Package ‘mxnet’
March 31, 2020

Type Package
Title MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems
Version 2.0.0
Date 2017-06-27
Author Tianqi Chen, Qiang Kou, Tong He, Anirudh Acharya <https://github.com/anirudhacharya>
Maintainer Qiang Kou <qkou@qkou.info>
Repository apache/incubator-mxnet
Description MXNet is a deep learning framework designed for both efficiency and flexibility. It allows you to mix the flavours of deep learning programs together to maximize the efficiency and your productivity.
License Apache License (== 2.0)
URL https://github.com/apache/incubator-mxnet/tree/master/R-package
BugReports https://github.com/apache/incubator-mxnet/issues
Imports methods,
Rcpp (>= 0.12.1),
DiagrammeR (>= 0.9.0),
visNetwork (>= 1.0.3),
data.table,
jsonlite,
magrittr,
stringr
Suggests testthat,
mlbench,
knitr,
rmarkdown,
imager,
covr
Depends R (>= 3.4.4)
LinkingTo Rcpp
VignetteBuilder knitr
RoxygenNote 7.1.0
Encoding UTF-8

R topics documented:

arguments ............... 15
as.array.MXNDArray ..... 16
as.matrix.MXNDArray ..... 16
children .............. 16
cxt .......................... 17
dim.MXNDArray ............ 17
graph.viz .................. 17
im2rec ................... 18
internals ............... 19
is.mx.context ............. 20
is.mx.dataiter .......... 20
is.mx.ndarray .......... 20
is.mx.symbol .......... 21
is.serialized ......... 21
length.MXNDArray ........ 22
mx.apply .......................... 22
mx.callback.early.stop ...... 23
mx.callback.log.speedometer ...... 23
mx.callback.log.train.metric ...... 24
mx.callback.save.checkpoint ...... 24
mx.cpu ...................... 25
mx.ctx.default .......... 25
mx.exec.backward ...... 25
mx.exec.forward ........ 26
mx.exec.update.arg.arrays ...... 26
mx.exec.update.aux.arrays ...... 26
mx.exec.update.grad.arrays ...... 27
mx.gpu ...................... 27
mx.infer.rnn ............. 28
mx.infer.rnn.one ........ 28
mx.infer.rnn.one.unroll ...... 29
mx.init.create .......... 29
mx.init.internal.default ...... 30
mx.init.normal ......... 30
mx.init.uniform .......... 30
mx.init.Xavier .......... 31
mx.io.arrayiter .......... 31
mx.io.bucket.iter .......... 32
mx.io.CSVIter .......... 32
mx.io.extract .......... 34
mx.io.ImageDetRecordIter ...... 34
mx.io.ImageRecordInt8Iter ...... 38
<table>
<thead>
<tr>
<th>Topics Documented:</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mx.io.ImageRecordIter</td>
<td>41</td>
</tr>
<tr>
<td>mx.io.ImageRecordIter_v1</td>
<td>44</td>
</tr>
<tr>
<td>mx.io.ImageRecordUint8Iter</td>
<td>47</td>
</tr>
<tr>
<td>mx.io.ImageRecordUint8Iter_v1</td>
<td>50</td>
</tr>
<tr>
<td>mx.io.LibSVMIter</td>
<td>53</td>
</tr>
<tr>
<td>mx.io.MNISTIter</td>
<td>55</td>
</tr>
<tr>
<td>mx.kv.create</td>
<td>56</td>
</tr>
<tr>
<td>mx.lr_scheduler.FactorScheduler</td>
<td>56</td>
</tr>
<tr>
<td>mx.lr_scheduler.MultiFactorScheduler</td>
<td>57</td>
</tr>
<tr>
<td>mx.metric.accuracy</td>
<td>58</td>
</tr>
<tr>
<td>mx.metric.custom</td>
<td>58</td>
</tr>
<tr>
<td>mx.metric.logistic_acc</td>
<td>58</td>
</tr>
<tr>
<td>mx.metric.logloss</td>
<td>59</td>
</tr>
<tr>
<td>mx.metric.mae</td>
<td>59</td>
</tr>
<tr>
<td>mx.metric.mse</td>
<td>59</td>
</tr>
<tr>
<td>mx.metric.Perplexity</td>
<td>60</td>
</tr>
<tr>
<td>mx.metric.rmse</td>
<td>60</td>
</tr>
<tr>
<td>mx.metric.rmsle</td>
<td>60</td>
</tr>
<tr>
<td>mx.metric.top_k_accuracy</td>
<td>61</td>
</tr>
<tr>
<td>mx.mlp</td>
<td>61</td>
</tr>
<tr>
<td>mx.model.buckets</td>
<td>62</td>
</tr>
<tr>
<td>mx.model.FeedForward.create</td>
<td>63</td>
</tr>
<tr>
<td>mx.model.init.params</td>
<td>65</td>
</tr>
<tr>
<td>mx.model.load</td>
<td>65</td>
</tr>
<tr>
<td>mx.model.save</td>
<td>66</td>
</tr>
<tr>
<td>mx.nd.abs</td>
<td>66</td>
</tr>
<tr>
<td>mx.nd.Activation</td>
<td>67</td>
</tr>
<tr>
<td>mx.nd.adam.update</td>
<td>67</td>
</tr>
<tr>
<td>mx.nd.add.n</td>
<td>68</td>
</tr>
<tr>
<td>mx.nd.all.finite</td>
<td>69</td>
</tr>
<tr>
<td>mx.nd.amp.cast</td>
<td>69</td>
</tr>
<tr>
<td>mx.nd.amp.multicast</td>
<td>70</td>
</tr>
<tr>
<td>mx.nd.arccos</td>
<td>70</td>
</tr>
<tr>
<td>mx.nd.arccosh</td>
<td>71</td>
</tr>
<tr>
<td>mx.nd.arcsin</td>
<td>71</td>
</tr>
<tr>
<td>mx.nd.arcsinh</td>
<td>72</td>
</tr>
<tr>
<td>mx.nd.arctan</td>
<td>72</td>
</tr>
<tr>
<td>mx.nd.arctanh</td>
<td>73</td>
</tr>
<tr>
<td>mx.nd.argmax</td>
<td>73</td>
</tr>
<tr>
<td>mx.nd.argmax.channel</td>
<td>74</td>
</tr>
<tr>
<td>mx.nd.argmin</td>
<td>75</td>
</tr>
<tr>
<td>mx.nd.argsort</td>
<td>75</td>
</tr>
<tr>
<td>mx.nd.array</td>
<td>76</td>
</tr>
<tr>
<td>mx.nd.batch.dot</td>
<td>77</td>
</tr>
<tr>
<td>mx.nd.batch.take</td>
<td>78</td>
</tr>
<tr>
<td>mx.nd.BatchNorm</td>
<td>78</td>
</tr>
<tr>
<td>mx.nd.BatchNorm_v1</td>
<td>80</td>
</tr>
<tr>
<td>mx.nd.BilinearSampler</td>
<td>81</td>
</tr>
</tbody>
</table>
R topics documented:

- mx.nd.BlockGrad
- mx.nd.broadcast.add
- mx.nd.broadcast.axes
- mx.nd.broadcast.axis
- mx.nd.broadcast.div
- mx.nd.broadcast.equal
- mx.nd.broadcast.greater
- mx.nd.broadcast.greater.equal
- mx.nd.broadcast.hypot
- mx.nd.broadcast.lesser
- mx.nd.broadcast.lesser.equal
- mx.nd.broadcast.like
- mx.nd.broadcast.logical.and
- mx.nd.broadcast.logical.or
- mx.nd.broadcast.logical.xor
- mx.nd.broadcast.maximum
- mx.nd.broadcast.minimum
- mx.nd.broadcast.minus
- mx.nd.broadcast.mod
- mx.nd.broadcast.mul
- mx.nd.broadcast.not.equal
- mx.nd.broadcast.plus
- mx.nd.broadcast.power
- mx.nd.broadcast.sub
- mx.nd.broadcast.to
- mx.nd.Cast
- mx.nd.cast
- mx.nd.cast.storage
- mx.nd.cbrt
- mx.nd.ceil
- mx.nd.choose.element.0index
- mx.nd.clip
- mx.nd.col2im
- mx.nd.Concat
- mx.nd.concat
- mx.nd.Convolution
- mx.nd.Convolution.v1
- mx.nd.copyto
- mx.nd.Correlation
- mx.nd.cos
- mx.nd.cosh
- mx.nd.Crop
- mx.nd.crop
- mx.nd.ctc.loss
- mx.nd.CTCLoss
- mx.nd.cumsum
- mx.nd.Custom
- mx.nd.Deconvolution
topics documented:

mx.nd.degrees .................................................. 118
mx.nd.depth.to.space ........................................ 119
mx.nd.diag ....................................................... 119
mx.nd.dot ........................................................ 120
mx.nd.Dropout .................................................... 122
mx.nd.ElementWiseSum ......................................... 123
mx.nd.elemwise.add ............................................. 123
mx.nd.elemwise.div ............................................. 124
mx.nd.elemwise.mul ............................................. 124
mx.nd.elemwise.sub ............................................ 125
mx.nd.Embedding ............................................... 125
mx.nd.erf ......................................................... 126
mx.nd.erfinv ..................................................... 127
mx.nd.exp ......................................................... 127
mx.nd.expdm ..................................................... 128
mx.nd.expm1 ..................................................... 128
mx.nd.fill.element.0index .................................... 129
mx.nd.fix ........................................................ 129
mx.nd.Flatten ................................................... 130
mx.nd.flatten .................................................... 130
mx.nd.flip ........................................................ 131
mx.nd.floor ....................................................... 131
mx.nd.ftml.update .............................................. 132
mx.nd.ftrl.update .............................................. 133
mx.nd.FullyConnected .......................................... 134
mx.nd.gamma ...................................................... 135
mx.nd.gammaMain ............................................... 135
mx.nd.gather.nd ............................................... 136
mx.nd.GridGenerator ........................................... 136
mx.nd.GroupNorm ............................................... 137
mx.nd.hard.sigmoid ............................................. 138
mx.nd.identity .................................................. 138
mx.nd.IdentityAttachKL.SparseReg .. .......................... 139
mx.nd.im2col .................................................... 139
mx.nd.InstanceNorm ............................................. 140
mx.nd.khatri.rao ............................................... 141
mx.nd.L2Normalization ......................................... 142
mx.nd.lamb.update.phase1 ..................................... 143
mx.nd.lamb.update.phase2 ..................................... 144
mx.nd.LayerNorm ............................................... 144
mx.nd.LeakyReLU ................................................ 145
mx.nd.linalg.det ............................................... 146
mx.nd.linalg.extractdiag ..................................... 147
mx.nd.linalg.extracttrian .................................... 148
mx.nd.linalg.gelqf ............................................. 149
mx.nd.linalg.gemm ............................................. 150
mx.nd.linalg.gemm2 ............................................ 151
mx.nd.linalg.inverse .......................................... 152
<table>
<thead>
<tr>
<th>Topics Documented</th>
</tr>
</thead>
<tbody>
<tr>
<td>mx.nd.linalg.makediag</td>
</tr>
<tr>
<td>mx.nd.linalg.maketrian</td>
</tr>
<tr>
<td>mx.nd.linalg.potrf</td>
</tr>
<tr>
<td>mx.nd.linalg.potri</td>
</tr>
<tr>
<td>mx.nd.linalg.slogdet</td>
</tr>
<tr>
<td>mx.nd.linalg.sumlogdiag</td>
</tr>
<tr>
<td>mx.nd.linalg.syrk</td>
</tr>
<tr>
<td>mx.nd.linalg.trmm</td>
</tr>
<tr>
<td>mx.nd.linalg.trsm</td>
</tr>
<tr>
<td>mx.nd.LinearRegressionOutput</td>
</tr>
<tr>
<td>mx.nd.load</td>
</tr>
<tr>
<td>mx.nd.log</td>
</tr>
<tr>
<td>mx.nd.log.softmax</td>
</tr>
<tr>
<td>mx.nd.log10</td>
</tr>
<tr>
<td>mx.nd.log1p</td>
</tr>
<tr>
<td>mx.nd.log2</td>
</tr>
<tr>
<td>mx.nd.logical.not</td>
</tr>
<tr>
<td>mx.nd.LogisticRegressionOutput</td>
</tr>
<tr>
<td>mx.nd.LRN</td>
</tr>
<tr>
<td>mx.nd.MAERegressionOutput</td>
</tr>
<tr>
<td>mx.nd.make.loss</td>
</tr>
<tr>
<td>mx.nd.MakeLoss</td>
</tr>
<tr>
<td>mx.nd.max</td>
</tr>
<tr>
<td>mx.nd.max.axis</td>
</tr>
<tr>
<td>mx.nd.mean</td>
</tr>
<tr>
<td>mx.nd.min</td>
</tr>
<tr>
<td>mx.nd.min.axis</td>
</tr>
<tr>
<td>mx.nd.moments</td>
</tr>
<tr>
<td>mx.nd.mp.lamb.update.phase1</td>
</tr>
<tr>
<td>mx.nd.mp.lamb.update.phase2</td>
</tr>
<tr>
<td>mx.nd.mp.nag.mom.update</td>
</tr>
<tr>
<td>mx.nd.mp.sgd.mom.update</td>
</tr>
<tr>
<td>mx.nd.mp.sgd.update</td>
</tr>
<tr>
<td>mx.nd.multi.all.finite</td>
</tr>
<tr>
<td>mx.nd.multi.lars</td>
</tr>
<tr>
<td>mx.nd.multi.mp.sgd.mom.update</td>
</tr>
<tr>
<td>mx.nd.multi.mp.sgd.update</td>
</tr>
<tr>
<td>mx.nd.multi.sgd.mom.update</td>
</tr>
<tr>
<td>mx.nd.multi.sgd.update</td>
</tr>
<tr>
<td>mx.nd.multi.sum.sq</td>
</tr>
<tr>
<td>mx.nd.nag.mom.update</td>
</tr>
<tr>
<td>mx.nd.nanprod</td>
</tr>
<tr>
<td>mx.nd.nansum</td>
</tr>
<tr>
<td>mx.nd.negative</td>
</tr>
<tr>
<td>mx.nd.norm</td>
</tr>
<tr>
<td>mx.nd.normal</td>
</tr>
<tr>
<td>mx.nd.one.hot</td>
</tr>
<tr>
<td>mx.nd.ones</td>
</tr>
</tbody>
</table>
R topics documented:

mx.nd.ones_like .......................................................... 187
mx.nd.Pad ................................................................. 188
mx.nd.pad ................................................................. 189
mx.nd.pick ................................................................. 190
mx.nd.Pooling ............................................................ 191
mx.nd.Pooling.v1 ......................................................... 193
mx.nd.preloaded.multi.mp.sgd.mom.update .......................... 194
mx.nd.preloaded.multi.mp.sgd.update ............................... 195
mx.nd.preloaded.multi.sgd.mom.update ............................. 195
mx.nd.preloaded.multi.sgd.update .................................. 196
mx.nd.prod ................................................................. 197
mx.nd.radians .............................................................. 197
mx.nd.random.exponential ............................................. 198
mx.nd.random.gamma ................................................... 199
mx.nd.random.generalized.negative.binomial ....................... 199
mx.nd.random.negative.binomial ..................................... 200
mx.nd.random.normal .................................................. 201
mx.nd.random.pdf.digichlet .......................................... 202
mx.nd.random.pdf.exponential ....................................... 202
mx.nd.random.pdf.gama ................................................. 203
mx.nd.random.pdf.generated.negative.binomial .................... 204
mx.nd.random.pdf.normal ............................................. 205
mx.nd.random.pdf.normal ............................................. 206
mx.nd.random.pdf.poisson ............................................. 207
mx.nd.random.pdf.uniform ............................................. 207
mx.nd.random.poisson .................................................. 208
mx.nd.random.randint ................................................. 209
mx.nd.random.uniform .................................................. 209
mx.nd.ravel.multi.index .............................................. 210
mx.nd.rcbrt .............................................................. 211
mx.nd.reciprocal ........................................................ 211
mx.nd.relu ............................................................... 212
mx.nd.repeat ............................................................ 212
mx.nd.reset.arrays ..................................................... 213
mx.nd.Reshape .......................................................... 213
mx.nd.reshape .......................................................... 215
mx.nd.reshape.like ..................................................... 216
mx.nd.reverse ........................................................... 217
mx.nd.rint ............................................................... 218
mx.nd.rmsprop.update .................................................. 218
mx.nd.rmspropalex.update ............................................ 219
mx.nd.RNN ............................................................... 221
mx.nd.ROIPooling ....................................................... 223
mx.nd.round .............................................................. 224
mx.nd.rsqrt ............................................................. 224
mx.nd.sample.exponential ............................................. 225
mx.nd.sample.gamma .................................................... 226
mx.nd.sample.generated.negative.binomial .......................... 227
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mx.nd.sample.multinomial</td>
<td>228</td>
</tr>
<tr>
<td>mx.nd.sample.negative.binomial</td>
<td>229</td>
</tr>
<tr>
<td>mx.nd.sample.normal</td>
<td>230</td>
</tr>
<tr>
<td>mx.nd.sample.poisson</td>
<td>231</td>
</tr>
<tr>
<td>mx.nd.sample.uniform</td>
<td>232</td>
</tr>
<tr>
<td>mx.nd.save</td>
<td>233</td>
</tr>
<tr>
<td>mx.nd.scatter.nd</td>
<td>233</td>
</tr>
<tr>
<td>mx.nd.SequenceLast</td>
<td>234</td>
</tr>
<tr>
<td>mx.nd.SequenceMask</td>
<td>235</td>
</tr>
<tr>
<td>mx.nd.SequenceReverse</td>
<td>236</td>
</tr>
<tr>
<td>mx.nd.sgd.mom.update</td>
<td>238</td>
</tr>
<tr>
<td>mx.nd.sgd.update</td>
<td>239</td>
</tr>
<tr>
<td>mx.nd.shape.array</td>
<td>240</td>
</tr>
<tr>
<td>mx.nd.shuffle</td>
<td>240</td>
</tr>
<tr>
<td>mx.nd.sigmoid</td>
<td>241</td>
</tr>
<tr>
<td>mx.nd.sign</td>
<td>241</td>
</tr>
<tr>
<td>mx.nd.sigmoids.gd.update</td>
<td>242</td>
</tr>
<tr>
<td>mx.nd.sigmum.update</td>
<td>242</td>
</tr>
<tr>
<td>mx.nd.sin</td>
<td>243</td>
</tr>
<tr>
<td>mx.nd.sinh</td>
<td>244</td>
</tr>
<tr>
<td>mx.nd.size.array</td>
<td>244</td>
</tr>
<tr>
<td>mx.nd.slice.axis</td>
<td>245</td>
</tr>
<tr>
<td>mx.nd.slice.like</td>
<td>246</td>
</tr>
<tr>
<td>mx.nd.SliceChannel</td>
<td>247</td>
</tr>
<tr>
<td>mx.nd.smooth.l1</td>
<td>248</td>
</tr>
<tr>
<td>mx.nd.Softmax</td>
<td>248</td>
</tr>
<tr>
<td>mx.nd.softmax</td>
<td>250</td>
</tr>
<tr>
<td>mx.nd.softmax.output</td>
<td>251</td>
</tr>
<tr>
<td>mx.nd.SoftmaxActivation</td>
<td>252</td>
</tr>
<tr>
<td>mx.nd.SoftmaxOutput</td>
<td>253</td>
</tr>
<tr>
<td>mx.nd.softmin</td>
<td>254</td>
</tr>
<tr>
<td>mx.nd.softsign</td>
<td>255</td>
</tr>
<tr>
<td>mx.nd.sort</td>
<td>256</td>
</tr>
<tr>
<td>mx.nd.space.to.depth</td>
<td>256</td>
</tr>
<tr>
<td>mx.nd.SpatialTransformer</td>
<td>257</td>
</tr>
<tr>
<td>mx.nd.split</td>
<td>258</td>
</tr>
<tr>
<td>mx.nd.sqrt</td>
<td>259</td>
</tr>
<tr>
<td>mx.nd.squeeze</td>
<td>259</td>
</tr>
<tr>
<td>mx.nd.stack</td>
<td>260</td>
</tr>
<tr>
<td>mx.nd.stop.gradient</td>
<td>261</td>
</tr>
<tr>
<td>mx.nd.sum</td>
<td>262</td>
</tr>
<tr>
<td>mx.nd.sum.axis</td>
<td>263</td>
</tr>
<tr>
<td>mx.nd.SVMOutput</td>
<td>264</td>
</tr>
<tr>
<td>mx.nd.swapaxes</td>
<td>264</td>
</tr>
<tr>
<td>mx.nd.SwapAxis</td>
<td>265</td>
</tr>
<tr>
<td>mx.nd.take</td>
<td>265</td>
</tr>
<tr>
<td>mx.nd.tan</td>
<td>266</td>
</tr>
</tbody>
</table>
mx.nd.tanh ................................................. 267
mx.nd.tile .................................................. 268
mx.nd.topk .................................................. 268
mx.nd.transpose ........................................... 269
mx.nd.trunc .................................................. 270
mx.nd.uniform .............................................. 271
mx.nd.unravel_index ......................................... 271
mx.nd.UpSampling ........................................... 272
mx.nd.where .................................................. 273
mx.nd.zeros .................................................. 274
mx.nd.zeros_like ............................................. 274
mx.opt.adadelta .............................................. 275
mx.opt.adagrad .............................................. 276
mx.opt.adam .................................................. 276
mx.opt.create ............................................... 277
mx.opt.get.updater .......................................... 278
mx.opt.nag ................................................... 278
mx.opt.rmsprop ............................................... 279
mx.opt.sgd .................................................... 280
mx.profiler.config ........................................... 280
mx.profiler.state ............................................ 281
mx.rnorm ....................................................... 281
mx.runif ........................................................ 282
mx.serialize ................................................... 283
mx.set.seed .................................................... 283
mx.simple.bind ............................................... 284
mx.symbol.abs ............................................... 284
mx.symbol.Activation ......................................... 285
mx.symbol.add_n .............................................. 285
mx.symbol.all_finite ......................................... 287
mx.symbol.amp_cast ........................................... 288
mx.symbol.amp_multicast ..................................... 288
mx.symbol.arccos ............................................. 289
mx.symbol.arccosh ............................................ 290
mx.symbol.arcsin ............................................ 290
mx.symbol.arcsinh ............................................ 291
mx.symbol.arctan ............................................ 292
mx.symbol.arctanh ............................................ 292
mx.symbol.argmax ............................................. 293
mx.symbol.argmax_channel ................................... 294
mx.symbol.argmin ............................................. 294
mx.symbol.argsort ............................................ 295
mx.symbol.BatchNorm .......................................... 296
mx.symbol.BatchNorm_v1 ....................................... 298
mx.symbol.batch_dot .......................................... 299
mx.symbol.batch_take ......................................... 300
mx.symbol.BilinearSampler ................................. 301
<table>
<thead>
<tr>
<th>mx.symbol.Blo...</th>
<th>302</th>
</tr>
</thead>
<tbody>
<tr>
<td>mx.symbol.broad...</td>
<td>303</td>
</tr>
<tr>
<td>mx.symbol.broadc...</td>
<td>304</td>
</tr>
<tr>
<td>mx.symbol.broadcast_a...</td>
<td>305</td>
</tr>
<tr>
<td>mx.symbol.broadcast_d...</td>
<td>306</td>
</tr>
<tr>
<td>mx.symbol.broadcast_equal</td>
<td>306</td>
</tr>
<tr>
<td>mx.symbol.broadcast_g...</td>
<td>307</td>
</tr>
<tr>
<td>mx.symbol.broadcast_g...</td>
<td>308</td>
</tr>
<tr>
<td>mx.symbol.broadcast_h...</td>
<td>308</td>
</tr>
<tr>
<td>mx.symbol.broadcast_l...</td>
<td>309</td>
</tr>
<tr>
<td>mx.symbol.broadcast_l...</td>
<td>310</td>
</tr>
<tr>
<td>mx.symbol.broadcast_l...</td>
<td>311</td>
</tr>
<tr>
<td>mx.symbol.broadcast_l...</td>
<td>312</td>
</tr>
<tr>
<td>mx.symbol.broadcast_l...</td>
<td>312</td>
</tr>
<tr>
<td>mx.symbol.broadcast_l...</td>
<td>313</td>
</tr>
<tr>
<td>mx.symbol.broadcast_m...</td>
<td>314</td>
</tr>
<tr>
<td>mx.symbol.broadcast_m...</td>
<td>314</td>
</tr>
<tr>
<td>mx.symbol.broadcast_m...</td>
<td>315</td>
</tr>
<tr>
<td>mx.symbol.broadcast_m...</td>
<td>316</td>
</tr>
<tr>
<td>mx.symbol.broadcast_m...</td>
<td>317</td>
</tr>
<tr>
<td>mx.symbol.broadcast_n...</td>
<td>317</td>
</tr>
<tr>
<td>mx.symbol.broadcast_p...</td>
<td>318</td>
</tr>
<tr>
<td>mx.symbol.broadcast_po...</td>
<td>319</td>
</tr>
<tr>
<td>mx.symbol.broadcast_s...</td>
<td>320</td>
</tr>
<tr>
<td>mx.symbol.broadcast_t...</td>
<td>321</td>
</tr>
<tr>
<td>mx.symbol.Ca...</td>
<td>322</td>
</tr>
<tr>
<td>mx.symbol.ca...</td>
<td>322</td>
</tr>
<tr>
<td>mx.symbol.cast_s...</td>
<td>323</td>
</tr>
<tr>
<td>mx.symbol.cb...</td>
<td>324</td>
</tr>
<tr>
<td>mx.symbol.cei...</td>
<td>324</td>
</tr>
<tr>
<td>mx.symbol.choose_e...</td>
<td>325</td>
</tr>
<tr>
<td>mx.symbol.cli...</td>
<td>326</td>
</tr>
<tr>
<td>mx.symbol.co...</td>
<td>327</td>
</tr>
<tr>
<td>mx.symbol.Conc...</td>
<td>328</td>
</tr>
<tr>
<td>mx.symbol.conc...</td>
<td>329</td>
</tr>
<tr>
<td>mx.symbol.Convolution</td>
<td>329</td>
</tr>
<tr>
<td>mx.symbol.Convolution_v1</td>
<td>331</td>
</tr>
<tr>
<td>mx.symbol.Corr...</td>
<td>332</td>
</tr>
<tr>
<td>mx.symbol.cos</td>
<td>334</td>
</tr>
<tr>
<td>mx.symbol.cosh</td>
<td>334</td>
</tr>
<tr>
<td>mx.symbol.Crop</td>
<td>335</td>
</tr>
<tr>
<td>mx.symbol.cro...</td>
<td>336</td>
</tr>
<tr>
<td>mx.symbol.CTCLo...</td>
<td>337</td>
</tr>
<tr>
<td>mx.symbol.ctc_lo...</td>
<td>338</td>
</tr>
<tr>
<td>mx.symbol.cumsum</td>
<td>340</td>
</tr>
<tr>
<td>mx.symbol.Custo...</td>
<td>340</td>
</tr>
<tr>
<td>mx.symbol.Deconvol...</td>
<td>341</td>
</tr>
<tr>
<td>mx.symbol.degrees</td>
<td>342</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>mx.symbol.depth_to_space</td>
<td>343</td>
</tr>
<tr>
<td>mx.symbol.diag</td>
<td>344</td>
</tr>
<tr>
<td>mx.symbol.dot</td>
<td>345</td>
</tr>
<tr>
<td>mx.symbol.Dropout</td>
<td>346</td>
</tr>
<tr>
<td>mx.symbol.ElementWiseSum</td>
<td>347</td>
</tr>
<tr>
<td>mx.symbol.elemwise_add</td>
<td>348</td>
</tr>
<tr>
<td>mx.symbol.elemwise_div</td>
<td>348</td>
</tr>
<tr>
<td>mx.symbol.elemwise_mul</td>
<td>349</td>
</tr>
<tr>
<td>mx.symbol.elemwise_sub</td>
<td>349</td>
</tr>
<tr>
<td>mx.symbol.Embedding</td>
<td>350</td>
</tr>
<tr>
<td>mx.symbol.erf</td>
<td>351</td>
</tr>
<tr>
<td>mx.symbol.erfinv</td>
<td>352</td>
</tr>
<tr>
<td>mx.symbol.exp</td>
<td>353</td>
</tr>
<tr>
<td>mx.symbol.expand_dims</td>
<td>353</td>
</tr>
<tr>
<td>mx.symbol.expm1</td>
<td>354</td>
</tr>
<tr>
<td>mx.symbol.fill_element_0index</td>
<td>355</td>
</tr>
<tr>
<td>mx.symbol.fix</td>
<td>355</td>
</tr>
<tr>
<td>mx.symbol.Flatten</td>
<td>356</td>
</tr>
<tr>
<td>mx.symbol.flatten</td>
<td>357</td>
</tr>
<tr>
<td>mx.symbol.flip</td>
<td>357</td>
</tr>
<tr>
<td>mx.symbol.floor</td>
<td>358</td>
</tr>
<tr>
<td>mx.symbol.html_update</td>
<td>359</td>
</tr>
<tr>
<td>mx.symbol.lamb_update_phase1</td>
<td>360</td>
</tr>
<tr>
<td>mx.symbol.lamb_update_phase2</td>
<td>361</td>
</tr>
<tr>
<td>mx.symbol.gamma</td>
<td>362</td>
</tr>
<tr>
<td>mx.symbol.gammaLn</td>
<td>362</td>
</tr>
<tr>
<td>mx.symbol.gather_nd</td>
<td>363</td>
</tr>
<tr>
<td>mx.symbol.GridGenerator</td>
<td>363</td>
</tr>
<tr>
<td>mx.symbol.Group</td>
<td>364</td>
</tr>
<tr>
<td>mx.symbol.GroupNorm</td>
<td>365</td>
</tr>
<tr>
<td>mx.symbol.hard_sigmoid</td>
<td>366</td>
</tr>
<tr>
<td>mx.symbol.identity</td>
<td>366</td>
</tr>
<tr>
<td>mx.symbol.IdentityAttachKLSparseReg</td>
<td>367</td>
</tr>
<tr>
<td>mx.symbol.im2col</td>
<td>367</td>
</tr>
<tr>
<td>mx.symbol.infer.shape</td>
<td>368</td>
</tr>
<tr>
<td>mx.symbol.InstanceNorm</td>
<td>369</td>
</tr>
<tr>
<td>mx.symbol.khatri_rao</td>
<td>370</td>
</tr>
<tr>
<td>mx.symbol.L2Normalization</td>
<td>371</td>
</tr>
<tr>
<td>mx.symbol.lamb_update_phase1</td>
<td>372</td>
</tr>
<tr>
<td>mx.symbol.lamb_update_phase2</td>
<td>373</td>
</tr>
<tr>
<td>mx.symbol.LayerNorm</td>
<td>374</td>
</tr>
<tr>
<td>mx.symbol.LeakyReLU</td>
<td>375</td>
</tr>
<tr>
<td>mx.symbol.linalg_det</td>
<td>376</td>
</tr>
<tr>
<td>mx.symbol.linalg_extractdiag</td>
<td>377</td>
</tr>
<tr>
<td>mx.symbol.linalg_extracttriangular</td>
<td>378</td>
</tr>
<tr>
<td>mx.symbol.linalg_gelqf</td>
<td>379</td>
</tr>
<tr>
<td>mx.symbol.linalg_gelqf</td>
<td>380</td>
</tr>
<tr>
<td>mx.symbol.linalg_gemm</td>
<td>381</td>
</tr>
<tr>
<td>mx.symbol.linalg_gemm2</td>
<td></td>
</tr>
</tbody>
</table>
topics documented:

- mx.symbol.linalg_inverse .......................................................... 382
- mx.symbol.linalg_makediag ......................................................... 383
- mx.symbol.linalg_maketrian ......................................................... 384
- mx.symbol.linalg_potrf .............................................................. 385
- mx.symbol.linalg_potri .............................................................. 386
- mx.symbol.linalg_slogdet ........................................................... 387
- mx.symbol.linalg_sumlogdiag ..................................................... 388
- mx.symbol.linalg_syrk ............................................................... 389
- mx.symbol.linalg_trmm ............................................................. 390
- mx.symbol.linalg_trsm .............................................................. 391
- mx.symbol.LinearRegressionOutput .............................................. 392
- mx.symbol.load ................................................................. 393
- mx.symbol.load.json ............................................................. 393
- mx.symbol.log ................................................................. 394
- mx.symbol.log10 ................................................................. 394
- mx.symbol.log1p ................................................................. 395
- mx.symbol.log2 ................................................................. 395
- mx.symbol.logical_not ............................................................ 396
- mx.symbol.LogisticRegressionOutput ......................................... 396
- mx.symbol.log_softmax ............................................................ 397
- mx.symbol.LRN ................................................................. 398
- mx.symbol.MAERegressionOutput ............................................... 399
- mx.symbol.MakeLoss .............................................................. 400
- mx.symbol.make_loss ............................................................. 401
- mx.symbol.max ................................................................. 401
- mx.symbol.max_axis .............................................................. 402
- mx.symbol.mean ................................................................. 403
- mx.symbol.moments ............................................................... 404
- mx.symbol.mp_lamb_update_phase1 ............................................. 404
- mx.symbol.mp_lamb_update_phase2 ............................................. 406
- mx.symbol.mp_nag_mom_update .................................................. 407
- mx.symbol.mp_sgd_mom_update .................................................. 408
- mx.symbol.mp_sgd_update ....................................................... 409
- mx.symbol.multi_all_finite ......................................................... 409
- mx.symbol.multi_lars ............................................................. 410
- mx.symbol.multi_mp_sgd_mom_update ......................................... 411
- mx.symbol.multi_mp_sgd_update ................................................ 412
- mx.symbol.multi_sgd_mom_update .............................................. 413
- mx.symbol.multi_sgd_update ..................................................... 414
- mx.symbol.multi_sum_sq .......................................................... 415
- mx.symbol.nag_mom_update ....................................................... 415
- mx.symbol.nanprod .............................................................. 416
- mx.symbol.nansum .............................................................. 417
- mx.symbol.negative .............................................................. 418
- mx.symbol.norm .............................................................. 418
- mx.symbol.normal .............................................................. 419
- mx.symbol.ones_like ............................................................. 420
- mx.symbol.one_hot .............................................................. 421
R topics documented:

mx.symbol.Pad ................................................................. 422
mx.symbol.pad ............................................................. 423
mx.symbol.pick ............................................................. 424
mx.symbol.Pooling ......................................................... 426
mx.symbol.Pooling_v1 ..................................................... 427
mx.symbol.preloaded_multi_mp_sgd_mom_update ..................... 429
mx.symbol.preloaded_multi_mp_sgd_update .......................... 430
mx.symbol.preloaded_multi_sgd_mom_update ........................ 430
mx.symbol.preloaded_multi_sgd_update .............................. 431
mx.symbol.prod ............................................................ 432
mx.symbol.radians .......................................................... 433
mx.symbol.random_exponential ......................................... 433
mx.symbol.random_gamma ................................................ 434
mx.symbol.random_generalized_negative_binomial ................. 435
mx.symbol.random_negative_binomial ............................... 436
mx.symbol.random_normal ............................................... 437
mx.symbol.random_pdf_dirichlet ..................................... 438
mx.symbol.random_pdf_exponential ................................... 439
mx.symbol.random_pdf_gamma .......................................... 440
mx.symbol.random_pdf_generalized_negative_binomial .......... 441
mx.symbol.random_pdf_negative_binomial .......................... 442
mx.symbol.random_pdf_normal .......................................... 443
mx.symbol.random_pdf_poisson ....................................... 444
mx.symbol.random_pdf_uniform ....................................... 445
mx.symbol.random_poisson ............................................. 446
mx.symbol.random.randint ............................................. 446
mx.symbol.ravel_multi_index ......................................... 448
mx.symbol.rcbrt .......................................................... 449
mx.symbol.reciprocal ................................................... 449
mx.symbol.relu ........................................................... 450
mx.symbol.repeat ......................................................... 451
mx.symbol.reset_arrays ................................................ 451
mx.symbol.Reshape ....................................................... 452
mx.symbol.reshape ....................................................... 453
mx.symbol.reshape_like ................................................ 455
mx.symbol.reverse ....................................................... 456
mx.symbol.rint ............................................................ 457
mx.symbol.rmspropalex_update ..................................... 457
mx.symbol.rmsprop_update ............................................ 459
mx.symbol.RNN ............................................................ 460
mx.symbol.ROIPooling .................................................... 462
mx.symbol.round ........................................................ 463
mx.symbol.rsqrt .......................................................... 464
mx.symbol.sample_exponential ...................................... 465
mx.symbol.sample_gamma ............................................... 466
mx.symbol.sample_generalized_negative_binomial .............. 467
mx.symbol.sample_multinomial ...................................... 468
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mx.symbol.sample_negative_binomial</td>
<td>469</td>
</tr>
<tr>
<td>mx.symbol.sample_normal</td>
<td>470</td>
</tr>
<tr>
<td>mx.symbol.sample_poisson</td>
<td>471</td>
</tr>
<tr>
<td>mx.symbol.sample_uniform</td>
<td>472</td>
</tr>
<tr>
<td>mx.symbol.save</td>
<td>473</td>
</tr>
<tr>
<td>mx.symbol.scatter_nd</td>
<td>473</td>
</tr>
<tr>
<td>mx.symbol.SequenceLast</td>
<td>474</td>
</tr>
<tr>
<td>mx.symbol.SequenceMask</td>
<td>475</td>
</tr>
<tr>
<td>mx.symbol.SequenceReverse</td>
<td>477</td>
</tr>
<tr>
<td>mx.symbol.sgd_mom_update</td>
<td>478</td>
</tr>
<tr>
<td>mx.symbol.sgd_update</td>
<td>479</td>
</tr>
<tr>
<td>mx.symbol.shape_array</td>
<td>480</td>
</tr>
<tr>
<td>mx.symbol.shuffle</td>
<td>481</td>
</tr>
<tr>
<td>mx.symbol.sigmoid</td>
<td>481</td>
</tr>
<tr>
<td>mx.symbol.sign</td>
<td>482</td>
</tr>
<tr>
<td>mx.symbol.signsgd_update</td>
<td>482</td>
</tr>
<tr>
<td>mx.symbol.signum_update</td>
<td>483</td>
</tr>
<tr>
<td>mx.symbol.sin</td>
<td>484</td>
</tr>
<tr>
<td>mx.symbol.sinh</td>
<td>485</td>
</tr>
<tr>
<td>mx.symbol.size_array</td>
<td>486</td>
</tr>
<tr>
<td>mx.symbol.slice</td>
<td>486</td>
</tr>
<tr>
<td>mx.symbol.SliceChannel</td>
<td>488</td>
</tr>
<tr>
<td>mx.symbol.slice_axis</td>
<td>489</td>
</tr>
<tr>
<td>mx.symbol.slice_like</td>
<td>490</td>
</tr>
<tr>
<td>mx.symbol.smooth_L1</td>
<td>491</td>
</tr>
<tr>
<td>mx.symbol.Softmax</td>
<td>491</td>
</tr>
<tr>
<td>mx.symbol.softmax</td>
<td>493</td>
</tr>
<tr>
<td>mx.symbol.SoftmaxActivation</td>
<td>494</td>
</tr>
<tr>
<td>mx.symbol.SoftmaxOutput</td>
<td>495</td>
</tr>
<tr>
<td>mx.symbol.softmax_cross_entropy</td>
<td>497</td>
</tr>
<tr>
<td>mx.symbol.softmin</td>
<td>498</td>
</tr>
<tr>
<td>mx.symbol.softsign</td>
<td>499</td>
</tr>
<tr>
<td>mx.symbol.sort</td>
<td>499</td>
</tr>
<tr>
<td>mx.symbol.space_to_depth</td>
<td>500</td>
</tr>
<tr>
<td>mx.symbol.SpatialTransformer</td>
<td>501</td>
</tr>
<tr>
<td>mx.symbol.split</td>
<td>502</td>
</tr>
<tr>
<td>mx.symbol.sqrt</td>
<td>503</td>
</tr>
<tr>
<td>mx.symbol.square</td>
<td>504</td>
</tr>
<tr>
<td>mx.symbol.squeeze</td>
<td>504</td>
</tr>
<tr>
<td>mx.symbol.stack</td>
<td>505</td>
</tr>
<tr>
<td>mx.symbol.stop_gradient</td>
<td>506</td>
</tr>
<tr>
<td>mx.symbol.sum</td>
<td>507</td>
</tr>
<tr>
<td>mx.symbol.sum_axis</td>
<td>508</td>
</tr>
<tr>
<td>mx.symbol.SVMOutput</td>
<td>509</td>
</tr>
<tr>
<td>mx.symbol.swapaxes</td>
<td>510</td>
</tr>
<tr>
<td>mx.symbol.SwapAxis</td>
<td>510</td>
</tr>
<tr>
<td>mx.symbol.take</td>
<td>511</td>
</tr>
<tr>
<td>mx.symbol.tan</td>
<td>512</td>
</tr>
</tbody>
</table>
arguments

Get the arguments of symbol.

Description

Get the arguments of symbol.

Usage

arguments(x)

Arguments

x The input symbol
as.array.MXNDArray  as.array operator overload of mx.ndarray

**Description**

as.array operator overload of mx.ndarray

**Usage**

```r
## S3 method for class 'MXNDArray'
as.array(nd)
```

**Arguments**

- `nd` The mx.ndarray

as.matrix.MXNDArray  as.matrix operator overload of mx.ndarray

**Description**

as.matrix operator overload of mx.ndarray

**Usage**

```r
## S3 method for class 'MXNDArray'
as.matrix(nd)
```

**Arguments**

- `nd` The mx.ndarray

custom

**Description**

Gets a new grouped symbol whose output contains inputs to output nodes of the original symbol.

**Usage**

```r
children(x)
```

**Arguments**

- `x` The input symbol
**ctx**

*Get the context of mx.ndarray*

### Description

Get the context of mx.ndarray

### Usage

```plaintext
ctx(nd)
```

### Arguments

- **nd**: The mx.ndarray

---

**dim.MXNDArray**

*Dimension operator overload of mx.ndarray*

### Description

Dimension operator overload of mx.ndarray

### Usage

```plaintext
## S3 method for class 'MXNDArray'
dim(nd)
```

### Arguments

- **nd**: The mx.ndarray

---

**graph.viz**

*Convert symbol to Graphviz or visNetwork visualisation.*

### Description

Convert symbol to Graphviz or visNetwork visualisation.
Usage

```r
graph.viz(
  symbol,
  shape = NULL,
  direction = "TD",
  type = "graph",
  graph.width.px = NULL,
  graph.height.px = NULL
)
```

Arguments

- **symbol**: a string representing the symbol of a model.
- **shape**: a numeric representing the input dimensions to the symbol.
- **direction**: a string representing the direction of the graph, either TD or LR.
- **type**: a string representing the rendering engine of the graph, either graph or vis.
- **graph.width.px**: a numeric representing the size (width) of the graph. In pixels.
- **graph.height.px**: a numeric representing the size (height) of the graph. In pixels.

Value

- a graph object ready to be displayed with the `print` function.

---

**im2rec**

Convert images into image recordio format

Description

Convert images into image recordio format

Usage

```r
im2rec(
  image_lst,
  root,
  output_rec,
  label_width = 1L,
  pack_label = 0L,
  new_size = -1L,
  nsplit = 1L,
  partid = 0L,
  center_crop = 0L,
  quality = 95L,
  color_mode = 1L,
)```
unchanged = 0L,
inter_method = 1L,
encoding = "jpg"
)

Arguments

image_lst          The image lst file
root               The root folder for image files
output_rec         The output rec file
label_width        The label width in the list file. Default is 1.
pack_label         Whether to also pack multi dimensional label in the record file. Default is 0.
new_size           The shorter edge of image will be resized to the newsize. Original images will be packed by default.
nsplit             It is used for part generation, logically split the image.lst to NSPLIT parts by position. Default is 1.
partid             It is used for part generation, pack the images from the specific part in image.lst. Default is 0.
center_crop        Whether to crop the center image to make it square. Default is 0.
quality            JPEG quality for encoding (1-100, default: 95) or PNG compression for encoding (1-9, default: 3).
color_mode         Force color (1), gray image (0) or keep source unchanged (-1). Default is 1.
unchanged          Keep the original image encoding, size and color. If set to 1, it will ignore the others parameters.
inter_method       NN(0), BILINEAR(1), CUBIC(2), AREA(3), LANCZOS4(4), AUTO(9), RAND(10). Default is 1.
encoding           The encoding type for images. It can be `.jpg` or `.png`. Default is `.jpg`.

---

**internals**

*Get a symbol that contains all the internals*

---

Description

Get a symbol that contains all the internals

Usage

`internals(x)`

Arguments

x                  The input symbol
is.mx.context        Check if the type is mxnet context.

Description
Check if the type is mxnet context.

Usage
is.mx.context(x)

Value
Logical indicator

is.mx.dataiter        Judge if an object is mx.dataiter

Description
Judge if an object is mx.dataiter

Usage
is.mx.dataiter(x)

Value
Logical indicator

is.mx.ndarray        Check if src.array is mx.ndarray

Description
Check if src.array is mx.ndarray

Usage
is.mx.ndarray(src.array)

Value
Logical indicator
Examples

```python
mat = mx.nd.array(1:10)
mat = mx.ndarray(mat)
mat2 = 1:10
is.mx.ndarray(mat2)
```

---

**is.mx.symbol**

Judge if an object is `mx.symbol`

**Description**

Judge if an object is `mx.symbol`

**Usage**

```python
is.mx.symbol(x)
```

**Value**

Logical indicator

---

**is.serialized**

Check if the model has been serialized into RData-compatible format.

**Description**

Check if the model has been serialized into RData-compatible format.

**Usage**

```python
is.serialized(model)
```

**Value**

Logical indicator
**length.MXNDArray**

*Length operator overload of mx.ndarray*

**Description**

Length operator overload of mx.ndarray

**Usage**

```r
## S3 method for class 'MXNDArray'
length(nd)
```

**Arguments**

- `nd` The mx.ndarray

---

**mx.apply**

*Apply symbol to the inputs.*

**Description**

Apply symbol to the inputs.

**Usage**

```r
mx.apply(x, ...)
```

**Arguments**

- `x` The symbol to be applied
- `kwargs` The keyword arguments to the symbol
mx.callback.early.stop

*Early stop with different conditions*

**Description**

Early stopping applying different conditions: hard thresholds or epochs number from the best score. Tested with "epoch.end.callback" function.

**Usage**

```r
mx.callback.early.stop(
  train.metric = NULL,
  eval.metric = NULL,
  bad.steps = NULL,
  maximize = FALSE,
  verbose = FALSE
)
```

**Arguments**

- `train.metric`: Numeric. Hard threshold for the metric of the training data set (optional)
- `eval.metric`: Numeric. Hard threshold for the metric of the evaluating data set (if set, optional)
- `bad.steps`: Integer. How much epochs should gone from the best score? Use this option with evaluation data set
- `maximize`: Logical. Do your model use maximizing or minimizing optimization?
- `verbose`: Logical

mx.callback.log.speedometer

*Calculate the training speed*

**Description**

Calculate the training speed

**Usage**

```r
mx.callback.log.speedometer(batch.size, frequency = 50)
```

**Arguments**

- `frequency`: The frequency of the training speed update
- `batch.size`: The batch size
mx.callback.log.train.metric

*Log training metric each period*

**Description**

Log training metric each period

**Usage**

```r
mx.callback.log.train.metric(period, logger = NULL)
```

**Arguments**

- **period**: The number of batch to log the training evaluation metric
- **logger**: The logger class

mx.callback.save.checkpoint

*Save checkpoint to files each period iteration.*

**Description**

Save checkpoint to files each period iteration.

**Usage**

```r
mx.callback.save.checkpoint(prefix, period = 1)
```

**Arguments**

- **prefix**: The prefix of the model checkpoint.
mx.cpu

Create a mxnet CPU context.

Description

Create a mxnet CPU context.

Arguments

dev.id optional, default=0 The device ID, this is meaningless for CPU, included for interface compatibility.

Value

The CPU context.

mx.ctx.default

Set/Get default context for array creation.

Description

Set/Get default context for array creation.

Usage

mx.ctx.default(new = NULL)

Arguments

new optional takes mx.cpu() or mx.gpu(id), new default ctx.

Value

The default context.

mx.exec.backward

Perform a backward on the executors This function will MUTATE the state of exec

Description

Perform a backward on the executors This function will MUTATE the state of exec

Usage

mx.exec.backward(exec, ...)

**mx.exec.forward**

*Perform an forward on the executors This function will MUTATE the state of exec*

**Description**

Perform an forward on the executors This function will MUTATE the state of exec

**Usage**

```r
mx.exec.forward(exec, is.train = TRUE)
```

---

**mx.exec.update.arg.arrays**

*Update the executors with new arrays This function will MUTATE the state of exec*

**Description**

Update the executors with new arrays This function will MUTATE the state of exec

**Usage**

```r
dxexec.update.arg.arrays(
  exec,
  arg.arrays,
  match.name = FALSE,
  skip.null = FALSE
)
```

---

**mx.exec.update.aux.arrays**

*Update the executors with new arrays This function will MUTATE the state of exec*

**Description**

Update the executors with new arrays This function will MUTATE the state of exec
**mx.exec.update.grad.arrays**

Usage

```r
mx.exec.update.aux.arrays(
  exec,
  arg.arrays,
  match.name = FALSE,
  skip.null = FALSE
)
```

Description

Update the executors with new arrays. This function will MUTATE the state of `exec`.

**mx.gpu**

Create a mxnet GPU context.

Description

Create a mxnet GPU context.

Arguments

- `dev.id` (optional, default=0) The GPU device ID, starts from 0.

Value

The GPU context.
mx.infer.rnn  Inference of RNN model

Description
Inference of RNN model

Usage
mx.infer.rnn(infer.data, model, ctx = mx.cpu())

Arguments
infer.data  DataIter
model  Model used for inference
ctx

mx.infer.rnn.one  Inference for one-to-one fusedRNN (CUDA) models

Description
Inference for one-to-one fusedRNN (CUDA) models

Usage
mx.infer.rnn.one(
    infer.data,
    symbol,
    arg.params,
    aux.params,
    input.params = NULL,
    ctx = mx.cpu()
)

Arguments
infer.data  Data iterator created by mx.io.bucket.iter
symbol  Symbol used for inference
ctx

mx.infer.rnn.one.unroll

Inference for one-to-one unroll models

Description

Inference for one-to-one unroll models

Usage

mx.infer.rnn.one.unroll(
    infer.data,
    symbol,
    num_hidden,
    arg.params,
    aux.params,
    init_states = NULL,
    ctx = mx.cpu()
)

Arguments

infer.data NDArray
symbol Model used for inference
num_hidden
cctx

mx.init.create Create initialization of argument like arg.array

Description

Create initialization of argument like arg.array

Usage

mx.init.create(initializer, shape.array, ctx = NULL, skip.unknown = TRUE)

Arguments

initializer The initializer.
shape.array A named list that represents the shape of the weights
cctx mx.context The context of the weights
skip.unknown Whether skip the unknown weight types
mx.init.uniform

Create a initializer that initialize the weight with uniform [-scale, scale]

Description
Create a initializer that initialize the weight with uniform [-scale, scale]

Usage
mx.init.uniform(scale)

Arguments
scale The scale of uniform distribution

mx.init.normal

Create a initializer that initialize the weight with normal(0, sd)

Description
Create a initializer that initialize the weight with normal(0, sd)

Usage
mx.init.normal(sd)

Arguments
sd The standard deviation of normal distribution

mx.init.internal.default

Internal default value initialization scheme.

Description
Internal default value initialization scheme.

Usage
mx.init.internal.default(name, shape, ctx, allow.unknown = FALSE)

Arguments
name the name of the variable.
shape the shape of the array to be generated.
**mx.init.Xavier**

**Xavier initializer**

**Description**
Create a initializer which initialize weight with Xavier or similar initialization scheme.

**Usage**
mx.init.Xavier(rnd_type = "uniform", factor_type = "avg", magnitude = 3)

**Arguments**
- `rnd_type`: A string of character indicating the type of distribution from which the weights are initialized.
- `factor_type`: A string of character.
- `magnitude`: A numeric number indicating the scale of random number range.

---

**mx.io.arrayiter**

Create MXDataIter compatible iterator from R’s array

**Description**
Create MXDataIter compatible iterator from R’s array.

**Usage**
mx.io.arrayiter(data, label, batch.size = 128, shuffle = FALSE)

**Arguments**
- `data`: The data array.
- `label`: The label array.
- `batch.size`: The batch size used to pack the array.
- `shuffle`: Whether shuffle the data
mx.io.bucket.iter  

Create Bucket Iter

Description

Create Bucket Iter

Usage

mx.io.bucket.iter(
    buckets,
    batch.size,
    data.mask.element = 0,
    shuffle = FALSE,
    seed = 123
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buckets</td>
<td>The data array.</td>
</tr>
<tr>
<td>batch.size</td>
<td>The batch size used to pack the array.</td>
</tr>
<tr>
<td>data.mask.element</td>
<td>The element to mask</td>
</tr>
<tr>
<td>shuffle</td>
<td>Whether shuffle the data</td>
</tr>
<tr>
<td>seed</td>
<td>The random seed</td>
</tr>
</tbody>
</table>

mx.io.CSVIter  

Returns the CSV file iterator.

Description

In this function, the 'data_shape' parameter is used to set the shape of each line of the input data. If a row in an input file is '1,2,3,4,5,6' and 'data_shape' is (3,2), that row will be reshaped, yielding the array [[[1,2],[3,4],[5,6]]] of shape (3,2).

Usage

mx.io.CSVIter(...)
**Arguments**

- **data.csv**
  - string, required
  - The input CSV file or a directory path.

- **data.shape**
  - Shape(tuple), required
  - The shape of one example.

- **label.csv**
  - string, optional, default='NULL'
  - The input CSV file or a directory path. If NULL, all labels will be returned as 0.

- **label.shape**
  - Shape(tuple), optional, default=[1]
  - The shape of one label.

- **batch.size**
  - int (non-negative), required
  - Batch size.

- **round.batch**
  - boolean, optional, default=1
  - Whether to use round robin to handle overflow batch or not.

- **prefetch.buffer**
  - long (non-negative), optional, default=4
  - Maximum number of batches to prefetch.

- **ctx**
  - 'cpu', 'gpu', optional, default='gpu'
  - Context data loader optimized for.

- **dtype**
  - None, 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', optional,
  - default='None'
  - Output data type. “None” means no change.

**Details**

By default, the `CSVIter` has ‘round_batch’ parameter set to “True”. So, if ‘batch_size’ is 3 and there are 4 total rows in CSV file, 2 more examples are consumed at the first round. If ‘reset’ function is called after first round, the call is ignored and remaining examples are returned in the second round.

If one wants all the instances in the second round after calling ‘reset’, make sure to set ‘round_batch’ to False.

If “data_csv = ’data/’” is set, then all the files in this directory will be read.

‘reset()’ is expected to be called only after a complete pass of data.

By default, the CSVIter parses all entries in the data file as float32 data type, if ‘dtype’ argument is set to be ‘int32’ or ‘int64’ then CSVIter will parse all entries in the file as int32 or int64 data type accordingly.

**Examples:**

// Contents of CSV file “data/data.csv”. 1,2,3 2,3,4 3,4,5 4,5,6
// Creates a ‘CSVIter’ with ‘batch_size’=2 and default ‘round_batch’=True. CSVIter = mx.io.CSVIter(data_csv = 'data/data.csv', data_shape = (3,), batch_size = 2)
// Two batches read from the above iterator are as follows: [[ 1. 2. 3.], [ 2. 3. 4.], [ 3. 4. 5.], [ 4. 5. 6.]]
// Creates a ‘CSVIter’ with default ‘round_batch’ set to True. CSVIter = mx.io.CSVIter(data_csv = 'data/data.csv', data_shape = (3,), batch_size = 3)
// Two batches read from the above iterator in the first pass are as follows: [[ 1. 2. 3.], [ 2. 3. 4.], [ 3. 4. 5.], [ 4. 5. 6.], [ 1. 2. 3.]]
// Now, ‘reset’ method is called. CSVIter.reset()
// Batch read from the above iterator in the second pass is as follows: [[ 3. 4. 5.], [ 4. 5. 6.], [ 1. 2. 3.]]
mx.io.ImageDetRecordIter

Create iterator for image detection dataset packed in recordio.

Description

Create iterator for image detection dataset packed in recordio.

Usage

mx.io.ImageDetRecordIter(...)

mx.io.extract

Extract a certain field from DataIter.

Description

Extract a certain field from DataIter.

Usage

mx.io.extract(iter, field)
Arguments

- **path.imglist**: string, optional, default="". Dataset Param: Path to image list.
- **path.imgrec**: string, optional, default="/data/imgrec.rec". Dataset Param: Path to image record file.
- **aug.seq**: string, optional, default='det_aug_default'. Augmentation Param: the augmenter names to represent sequence of augmenters to be applied, seperated by comma. Additional keyword parameters will be seen by these augmenters. Make sure you don’t use normal augmenters for detection tasks.
- **label.width**: int, optional, default='-1'. Dataset Param: How many labels for an image, -1 for variable label size.
- **preprocess.threads**: int, optional, default='4'. Backend Param: Number of thread to do preprocessing.
- **verbose**: boolean, optional, default=1. Auxiliary Param: Whether to output parser information.
- **num.parts**: int, optional, default='1'. partition the data into multiple parts
- **part.index**: int, optional, default='0'. the index of the part will read
- **shuffle.chunk.size**: long (non-negative), optional, default=0. the size(MB) of the shuffle chunk, used with shuffle=True, it can enable global shuffling
- **shuffle.chunk.seed**: int, optional, default='0'. the seed for chunk shuffling
- **label.pad.width**: int, optional, default='0'. pad output label width if set larger than 0, -1 for auto estimate
- **label.pad.value**: float, optional, default=-1. label padding value if enabled
- **shuffle**: boolean, optional, default=0. Augmentation Param: Whether to shuffle data.
- **seed**: int, optional, default='0'. Augmentation Param: Random Seed.
- **batch.size**: int (non-negative), required. Batch size.
- **round.batch**: boolean, optional, default=1. Whether to use round robin to handle overflow batch or not.
- **prefetch.buffer**: long (non-negative), optional, default=4. Maximum number of batches to prefetch.
- **ctx**: 'cpu', 'gpu', optional, default='gpu'. Context data loader optimized for.
- **dtype**: None, 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8',optional, default='None'. Output data type. “None“ means no change.
- **resize**: int, optional, default='1'. Augmentation Param: scale shorter edge to size before applying other augmentations, -1 to disable.
- **rand.crop.prob**: float, optional, default=0. Augmentation Param: Probability of random cropping, <= 0 to disable
- **min.crop.scales**: tuple of <float>, optional, default=[0] Augmentation Param: Min crop scales.
max.crop.scales

min.crop.aspect.ratios

max.crop.aspect.ratios

min.crop.overlaps
tuple of float, optional, default=[0] Augmentation Param: Minimum crop IOU between crop_box and ground-truths.

max.crop.overlaps

min.crop.sample.coverages
tuple of float, optional, default=[0] Augmentation Param: Minimum ratio of intersect/crop_area between crop box and ground-truths.

max.crop.sample.coverages

min.crop.object.coverages
tuple of float, optional, default=[0] Augmentation Param: Minimum ratio of intersect/gt_area between crop box and ground-truths.

max.crop.object.coverages

num.crop.sampler
int, optional, default=’1’ Augmentation Param: Number of crop samplers.

crop.emit.mode
’center’, ’overlap’,optional, default=’center’ Augmentation Param: Emition mode for invalid ground-truths after crop. center: emit if centroid of object is out of crop region; overlap: emit if overlap is less than emit_overlap_thresh.

emit.overlap.thresh
float, optional, default=0.300000012 Augmentation Param: Emit overlap thresh for emit mode overlap only.

max.crop.trials
Shape(tuple), optional, default=[25] Augmentation Param: Skip cropping if fail crop trial count exceeds this number.

rand.pad.prob
float, optional, default=0 Augmentation Param: Probability for random padding.

max.pad.scale
float, optional, default=1 Augmentation Param: Maximum padding scale.

max.random.hue
int, optional, default=’0’ Augmentation Param: Maximum random value of H channel in HSL color space.

random.hue.prob
float, optional, default=0 Augmentation Param: Probability to apply random hue.
max.random.saturation
    int, optional, default='0' Augmentation Param: Maximum random value of S channel in HSL color space.

random.saturation.prob
    float, optional, default=0 Augmentation Param: Probability to apply random saturation.

max.random.illumination
    int, optional, default='0' Augmentation Param: Maximum random value of L channel in HSL color space.

random.illumination.prob
    float, optional, default=0 Augmentation Param: Probability to apply random illumination.

max.random.contrast
    float, optional, default=0 Augmentation Param: Maximum random value of delta contrast.

random.contrast.prob
    float, optional, default=0 Augmentation Param: Probability to apply random contrast.

rand.mirror.prob
    float, optional, default=0 Augmentation Param: Probability to apply horizontal flip aka. mirror.

fill.value
    int, optional, default='127' Augmentation Param: Filled color value while padding.

inter.method
    int, optional, default='1' Augmentation Param: 0-NN 1-bilinear 2-cubic 3-area 4-lanczos4 9-auto 10-rand.

data.shape
    Shape(tuple), required Dataset Param: Shape of each instance generated by the DataIter.

resize.mode
    'fit', 'force', 'shrink', optional, default='force' Augmentation Param: How image data fit in data_shape. force: force reshape to data_shape regardless of aspect ratio; shrink: ensure each side fit in data_shape, preserve aspect ratio; fit: fit image to data_shape, preserve ratio, will upscale if applicable.

mean.img
    string, optional, default=” Augmentation Param: Mean Image to be subtracted.

mean.r
    float, optional, default=0 Augmentation Param: Mean value on R channel.

mean.g
    float, optional, default=0 Augmentation Param: Mean value on G channel.

mean.b
    float, optional, default=0 Augmentation Param: Mean value on B channel.

mean.a
    float, optional, default=0 Augmentation Param: Mean value on Alpha channel.

std.r
    float, optional, default=0 Augmentation Param: Standard deviation on R channel.

std.g
    float, optional, default=0 Augmentation Param: Standard deviation on G channel.

std.b
    float, optional, default=0 Augmentation Param: Standard deviation on B channel.

std.a
    float, optional, default=0 Augmentation Param: Standard deviation on Alpha channel.

scale
    float, optional, default=1 Augmentation Param: Scale in color space.
Value

iter The result mx.dataiter

mx.io.ImageRecordInt8Iter

Iterating on image RecordIO files

Description

.. note:: “ImageRecordInt8Iter” is deprecated. Use ImageRecordIter(dtype=’int8’) instead.

Usage

mx.io.ImageRecordInt8Iter(...)

Arguments

path.imglist string, optional, default=” Path to the image list (.lst) file. Generally created with tools/im2rec.py. Format (Tab separated): <index of record> <one or more labels> <relative path from root folder>.

path.imgrec string, optional, default=” Path to the image RecordIO (.rec) file or a directory path. Created with tools/im2rec.py.

path.imgidx string, optional, default=” Path to the image RecordIO index (.idx) file. Created with tools/im2rec.py.

aug.seq string, optional, default=’aug_default’ The augmenter names to represent sequence of augmenters to be applied, seperated by comma. Additional keyword parameters will be seen by these augmenters.

label.width int, optional, default=’1’ The number of labels per image.

preprocess.threads int, optional, default=’4’ The number of threads to do preprocessing.

verbose boolean, optional, default=1 If or not output verbose information.

num.parts int, optional, default=’1’ Virtually partition the data into these many parts.

part.index int, optional, default=’0’ The "i*-th virtual partition to be read.

device.id int, optional, default=’0’ The device id used to create context for internal NDArray. Setting device_id to -1 will create Context::CPU(0). Setting device_id to a valid positive device id will create Context::CPUPinned(device_id). Default is 0.

shuffle.chunk.size long (non-negative), optional, default=0 The data shuffle buffer size in MB. Only valid if shuffle is true.

shuffle.chunk.seed int, optional, default=’0’ The random seed for shuffling

seed.aug int or None, optional, default=’None’ Random seed for augmentations.
shuffle (boolean, default=0) Whether to shuffle data randomly or not.
seed (int, default=’0’) The random seed.
batch.size (int, default=1) Batch size.
round.batch (boolean, default=1) Whether to use round robin to handle overflow batch or not.
prefetch.buffer (long, default=4) Maximum number of batches to prefetch.
ctx (‘cpu’, ’gpu’, default=’gpu’) Context data loader optimized for.
resize (int, default=’-1’) Down scale the shorter edge to a new size before applying other augmentations.
rand.crop (boolean, default=0) If or not randomly crop the image
random.resized.crop (boolean, default=0) If or not perform random resized cropping on the image, as a standard preprocessing for resnet training on ImageNet data.
max.rotate.angle (int, default=’0’) Rotate by a random degree in “[-v, v]”
max.aspect.ratio (float, default=0) Change the aspect (namely width/height) to a random value. If min_aspect_ratio is None then the aspect ratio ins sampled from [1 - max_aspect_ratio, 1 + max_aspect_ratio], else it is in “[min_aspect_ratio, max_aspect_ratio]”
min.aspect.ratio (float or None, default=None) Change the aspect (namely width/height) to a random value in “[min_aspect_ratio, max_aspect_ratio]”
max.shear.ratio (float, default=0) Apply a shear transformation (namely “(x, y)->(x+my, y)”) with “m” randomly chose from “[max_shear_ratio, max_shear_ratio]”
max.crop.size (int, default=’-1’) Crop both width and height into a random size in “[min_crop_size, max_crop_size].”Ignored if “random_resized_crop” is True.
min.crop.size (int, default=’-1’) Crop both width and height into a random size in “[min_crop_size, max_crop_size].”Ignored if “random_resized_crop” is True.
max.random.scale (float, default=1) Resize into “[width*s, height*s]” with ‘s’ randomly chosen from “[min_random_scale, max_random_scale]”. Ignored if “random_resized_crop” is True.
min.random.scale (float, default=1) Resize into “[width*s, height*s]” with ‘s’ randomly chosen from “[min_random_scale, max_random_scale]”Ignored if “random_resized_crop” is True.
max.random.area (float, default=1) Change the area (namely width * height) to a random value in “[min_random_area, max_random_area]”. Ignored if “random_resized_crop” is False.
min.random.area
float, optional, default=1 Change the area (namely width * height) to a random value in “[min_random_area, max_random_area]”. Ignored if “random_resized_crop” is False.

max.img.size
float, optional, default=1e+10 Set the maximal width and height after all resize and rotate argumentation are applied

min.img.size
float, optional, default=0 Set the minimal width and height after all resize and rotate argumentation are applied

brightness
float, optional, default=0 Add a random value in “[−brightness, brightness]” to the brightness of image.

contrast
float, optional, default=0 Add a random value in “[−contrast, contrast]” to the contrast of image.

saturation
float, optional, default=0 Add a random value in “[−saturation, saturation]” to the saturation of image.

pca.noise
float, optional, default=0 Add PCA based noise to the image.

random.h
int, optional, default=’0’ Add a random value in “[−random_h, random_h]” to the H channel in HSL color space.

random.s
int, optional, default=’0’ Add a random value in “[−random_s, random_s]” to the S channel in HSL color space.

random.l
int, optional, default=’0’ Add a random value in “[−random_l, random_l]” to the L channel in HSL color space.

rotate
int, optional, default=’−1’ Rotate by an angle. If set, it overwrites the “max_rotate_angle” option.

fill.value
int, optional, default=’255’ Set the padding pixels value to “fill_value“.

data.shape
Shape(tuple), required The shape of a output image.

inter.method
int, optional, default=’1’ The interpolation method: 0-NN 1-bilinear 2-cubic 3-area 4-lanczos4 9-auto 10-rand.

pad
int, optional, default=’0’ Change size from “[width, height]” into “[pad + width + pad, pad + height + pad]” by padding pixels

Details
This iterator is identical to “ImageRecordIter” except for using “int8” as the data type instead of “float”.

Defined in src/io/iter_image_recordio_2.cc:L948

Value
iter The result mx.dataiter
mx.io.ImageRecordIter

Iterates on image RecordIO files

Usage

mx.io.ImageRecordIter(...)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path.imglist</td>
<td>string, optional, default=&quot;&quot; Path to the image list (.lst) file. Generally created with tools/im2rec.py. Format (Tab separated): &lt;index of record&gt; &lt;one or more labels&gt; &lt;relative path from root folder&gt;.</td>
</tr>
<tr>
<td>path.imgrec</td>
<td>string, optional, default=&quot;&quot; Path to the image RecordIO (.rec) file or a directory path. Created with tools/im2rec.py.</td>
</tr>
<tr>
<td>path.imgidx</td>
<td>string, optional, default=&quot;&quot; Path to the image RecordIO index (.idx) file. Created with tools/im2rec.py.</td>
</tr>
<tr>
<td>aug.seq</td>
<td>string, optional, default=’aug_default’ The augmenter names to represent sequence of augmenters to be applied, seperated by comma. Additional keyword parameters will be seen by these augmenters.</td>
</tr>
<tr>
<td>label.width</td>
<td>int, optional, default=’1’ The number of labels per image.</td>
</tr>
<tr>
<td>preprocess.threads</td>
<td>int, optional, default=’4’ The number of threads to do preprocessing.</td>
</tr>
<tr>
<td>verbose</td>
<td>boolean, optional, default=1 If or not output verbose information.</td>
</tr>
<tr>
<td>num.parts</td>
<td>int, optional, default=’1’ Virtually partition the data into these many parts.</td>
</tr>
<tr>
<td>part.index</td>
<td>int, optional, default=’0’ The <em>i</em>-th virtual partition to be read.</td>
</tr>
<tr>
<td>device.id</td>
<td>int, optional, default=’0’ The device id used to create context for internal NDArray. Setting device_id to -1 will create Context::CPU(0). Setting device_id to valid positive device id will create Context::CPUPinned(device_id). Default is 0.</td>
</tr>
<tr>
<td>shuffle.chunk.size</td>
<td>long (non-negative), optional, default=0 The data shuffle buffer size in MB. Only valid if shuffle is true.</td>
</tr>
<tr>
<td>shuffle.chunk.seed</td>
<td>int, optional, default=’0’ The random seed for shuffling</td>
</tr>
<tr>
<td>seed.aug</td>
<td>int or None, optional, default=’None’ Random seed for augmentations.</td>
</tr>
<tr>
<td>shuffle</td>
<td>boolean, optional, default=0 Whether to shuffle data randomly or not.</td>
</tr>
<tr>
<td>seed</td>
<td>int, optional, default=’0’ The random seed.</td>
</tr>
<tr>
<td>batch.size</td>
<td>int (non-negative), required Batch size.</td>
</tr>
<tr>
<td>round.batch</td>
<td>boolean, optional, default=1 Whether to use round robin to handle overflow batch or not.</td>
</tr>
<tr>
<td>prefetch.buffer</td>
<td>long (non-negative), optional, default=4 Maximum number of batches to prefetch.</td>
</tr>
</tbody>
</table>
ctx 'cpu', 'gpu', optional, default='gpu' Context data loader optimized for.
dtype None, 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', optional, default='None' Output data type. “None” means no change.
resize int, optional, default='-1' Down scale the shorter edge to a new size before applying other augmentations.
rand.crop boolean, optional, default=0 If or not randomly crop the image
random.resized.crop boolean, optional, default=0 If or not perform random resized cropping on the image, as a standard preprocessing for resnet training on ImageNet data.
max.rotate.angle int, optional, default='0' Rotate by a random degree in “[−v, v]”
max.aspect.ratio float, optional, default=0 Change the aspect (namely width/height) to a random value. If min_aspect_ratio is None then the aspect ratio ins sampled from [1 - max_aspect_ratio, 1 + max_aspect_ratio], else it is in “[min_aspect_ratio, max_aspect_ratio]”
min.aspect.ratio float or None, optional, default=None Change the aspect (namely width/height) to a random value in “[min_aspect_ratio, max_aspect_ratio]”
max.shear.ratio float, optional, default=0 Apply a shear transformation (namely “(x,y)->(x+my,y)”) with “m” randomly chose from “[−max_shear_ratio, max_shear_ratio]”
max.crop.size int, optional, default='-1' Crop both width and height into a random size in “[min_crop_size, max_crop_size]”. Ignored if “random_resized_crop” is True.
min.crop.size int, optional, default='-1' Crop both width and height into a random size in “[min_crop_size, max_crop_size]”. Ignored if “random_resized_crop” is True.
max.random.scale float, optional, default=1 Resize into “[width*s, height*s]” with “s” randomly chosen from “[min_random_scale, max_random_scale]”. Ignored if “random_resized_crop” is True.
min.random.scale float, optional, default=1 Resize into “[width*s, height*s]” with “s” randomly chosen from “[min_random_scale, max_random_scale]”. Ignored if “random_resized_crop” is True.
max.random.area float, optional, default=1 Change the area (namely width * height) to a random value in “[min_random_area, max_random_area]”. Ignored if “random_resized_crop” is False.
min.random.area float, optional, default=1 Change the area (namely width * height) to a random value in “[min_random_area, max_random_area]”. Ignored if “random_resized_crop” is False.
max.img.size float, optional, default=1e+10 Set the maximal width and height after all resize and rotate argumentation are applied
min.img.size float, optional, default=0 Set the minimal width and height after all resize and rotate argumentation are applied
brightness float, optional, default=0 Add a random value in “[−brightness, brightness]” to the brightness of image.

contrast float, optional, default=0 Add a random value in “[−contrast, contrast]” to the contrast of image.

saturation float, optional, default=0 Add a random value in “[−saturation, saturation]” to the saturation of image.

pca.noise float, optional, default=0 Add PCA based noise to the image.

random.h int, optional, default=’0’ Add a random value in “[−random_h, random_h]” to the H channel in HSL color space.

random.s int, optional, default=’0’ Add a random value in “[−random_s, random_s]” to the S channel in HSL color space.

random.l int, optional, default=’0’ Add a random value in “[−random_l, random_l]” to the L channel in HSL color space.

rotate int, optional, default=’−1’ Rotate by an angle. If set, it overwrites the “max_rotate_angle” option.

fill.value int, optional, default=’255’ Set the padding pixels value to “fill_value“.

data.shape Shape(tuple), required The shape of a output image.

inter.method int, optional, default=’1’ The interpolation method: 0-NN 1-bilinear 2-cubic 3-area 4-lanczos4 9-auto 10-rand.

pad int, optional, default=’0’ Change size from “[width, height]” into “[pad + width + pad, pad + height + pad]” by padding pixels.

mirror boolean, optional, default=0 Whether to mirror the image or not. If true, images are flipped along the horizontal axis.

rand.mirror boolean, optional, default=0 Whether to randomly mirror images or not. If true, 50

\itemmean imgstring, optional, default=” Filename of the mean image.
\itemmean.r float, optional, default=0 The mean value to be subtracted on the R channel
\itemmean.g float, optional, default=0 The mean value to be subtracted on the G channel
\itemmean.b float, optional, default=0 The mean value to be subtracted on the B channel
\itemmean.a float, optional, default=0 The mean value to be subtracted on the alpha channel
\itemstd.r float, optional, default=1 Augmentation Param: Standard deviation on R channel.
\itemstd.g float, optional, default=1 Augmentation Param: Standard deviation on G channel.
\itemstd.b float, optional, default=1 Augmentation Param: Standard deviation on B channel.
\itemstd.a float, optional, default=1 Augmentation Param: Standard deviation on Alpha channel.
\itemscale float, optional, default=1 Multiply the image with a scale value.
Iterating on image RecordIO files

Usage

mx.io.ImageRecordIter_v1(...)

Arguments

path.imglist string, optional, default="" Path to the image list (.lst) file. Generally created with tools/im2rec.py. Format (Tab separated): <index of record> <one or more labels> <relative path from root folder>.

path.imgrec string, optional, default="" Path to the image RecordIO (.rec) file or a directory path. Created with tools/im2rec.py.

path.imgidx string, optional, default="" Path to the image RecordIO index (.idx) file. Created with tools/im2rec.py.

aug.seq string, optional, default='aug_default' The augmenter names to represent sequence of augmenters to be applied, separated by comma. Additional keyword parameters will be seen by these augmenters.

label.width int, optional, default='1' The number of labels per image.

preprocess.threads int, optional, default='4' The number of threads to do preprocessing.
verbose: boolean, optional, default=1 If or not output verbose information.
num.parts: int, optional, default=1' Virtually partition the data into these many parts.
part.index: int, optional, default=0' The *i*-*th virtual partition to be read.
device.id: int, optional, default=0' The device id used to create context for internal NDArray. Setting device_id to -1 will create Context::CPU(0). Setting device_id to valid positive device id will create Context::CPU(device_id). Default is 0.
shuffle.chunk.size: long (non-negative), optional, default=0 The data shuffle buffer size in MB. Only valid if shuffle is true.
shuffle.chunk.seed: int, optional, default=0 The random seed for shuffling
seed.aug: int or None, optional, default=0' Random seed for augmentations.
shuffle: boolean, optional, default=0 Whether to shuffle data randomly or not.
seed: int, optional, default=0' The random seed.
batch.size: int (non-negative), required Batch size.
round.batch: boolean, optional, default=1 Whether to use round robin to handle overflow batch or not.
prefetch.buffer: long (non-negative), optional, default=4 Maximum number of batches to prefetch.
ctx: 'cpu', 'gpu',optional, default='gpu' Context data loader optimized for.
dtype: None, 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8',optional, default='None' Output data type. "None" means no change.
resize: int, optional, default=0' Down scale the shorter edge to a new size before applying other augmentations.
rand.crop: boolean, optional, default=0 If or not randomly crop the image
random.resized.crop: boolean, optional, default=0 If or not perform random resized cropping on the image, as a standard preprocessing for resnet training on ImageNet data.
max.rotate.angle: int, optional, default=0' Rotate by a random degree in “[-v, v]”
max.aspect.ratio: float, optional, default=0 Change the aspect (namely width/height) to a random value. If min_aspect_ratio is None then the aspect ratio ins sampled from [1 - max_aspect_ratio, 1 + max_aspect_ratio], else it is in “[min_aspect_ratio, max_aspect_ratio]”
min.aspect.ratio: float or None, optional, default=None Change the aspect (namely width/height) to a random value in “[min_aspect_ratio, max_aspect_ratio]”
max.shear.ratio: float, optional, default=0 Apply a shear transformation (namely “(x,y)->(x+my,y)”) with “m” randomly chose from “[-max_shear_ratio, max_shear_ratio]”
max.crop.size: int, optional, default=1' Crop both width and height into a random size in “[min_crop_size, max_crop_size].” Ignored if “random_resized_crop” is True.
min.crop.size  int, optional, default=-1 Crop both width and height into a random size in 
“[min_crop_size, max_crop_size]. “Ignored if “random_resized_crop” is True.

max.random.scale  float, optional, default=1 Resize into “[width*s, height*s]” with “s” randomly 
chosen from “[min_random_scale, max_random_scale]”. Ignored if “random_resized_crop” 
is True.

min.random.scale  float, optional, default=1 Resize into “[width*s, height*s]” with “s” randomly 
chosen from “[min_random_scale, max_random_scale]” Ignored if “random_resized_crop” 
is True.

max.random.area  float, optional, default=1 Change the area (namely width * height) to a random 
value in “[min_random_area, max_random_area]”. Ignored if “random_resized_crop” 
is False.

min.random.area  float, optional, default=1 Change the area (namely width * height) to a random 
value in “[min_random_area, max_random_area]”. Ignored if “random_resized_crop” 
is False.

max.img.size  float, optional, default=1e+10 Set the maximal width and height after all resize 
and rotate argumentation are applied

min.img.size  float, optional, default=0 Set the minimal width and height after all resize and 
rotate argumentation are applied

brightness  float, optional, default=0 Add a random value in “[-brightness, brightness]” to 
the brightness of image.

contrast  float, optional, default=0 Add a random value in “[-contrast, contrast]” to the 
contrast of image.

saturation  float, optional, default=0 Add a random value in “[saturation, saturation]” to 
the saturation of image.

pca.noise  float, optional, default=0 Add PCA based noise to the image.

random.h  int, optional, default=0 Add a random value in “[random_h, random_h]” to 
the H channel in HSL color space.

random.s  int, optional, default=0 Add a random value in “[random_s, random_s]” to 
the S channel in HSL color space.

random.l  int, optional, default=0 Add a random value in “[random_l, random_l]” to 
the L channel in HSL color space.

rotate  int, optional, default=-1 Rotate by an angle. If set, it overwrites the “max_rotate_angle” 
option.

fill.value  int, optional, default=255 Set the padding pixels value to “fill_value”.

data.shape  Shape(tuple), required The shape of a output image.

inter.method  int, optional, default=1 The interpolation method: 0-NN 1-bilinear 2-cubic 
3-area 4-lanczos4 9-auto 10-rand.

pad  int, optional, default=0 Change size from “[width, height]” into “[pad + width 
+ pad, pad + height + pad]” by padding pixes
mx.io.ImageRecordUInt8Iter

Iterating on image RecordIO files

Description

.. note:: ImageRecordUInt8Iter is deprecated. Use ImageRecordIter(dtype='uint8') instead.
Usage

```
mx.io.ImageRecordUInt8Iter(...)```

Arguments

- `path.imglist` string, optional, default="" Path to the image list (.lst) file. Generally created with tools/im2rec.py. Format (Tab separated): <index of record> <one or more labels> <relative path from root folder>.
- `path.imgrec` string, optional, default="" Path to the image RecordIO (.rec) file or a directory path. Created with tools/im2rec.py.
- `path.imgidx` string, optional, default="" Path to the image RecordIO index (.idx) file. Created with tools/im2rec.py.
- `aug.seq` string, optional, default="aug_default" The augmenter names to represent sequence of augmenters to be applied, separated by comma. Additional keyword parameters will be seen by these augmenters.
- `label.width` int, optional, default=1 The number of labels per image.
- `preprocess.threads` int, optional, default=4 The number of threads to do preprocessing.
- `verbose` boolean, optional, default=1 If or not output verbose information.
- `num.parts` int, optional, default=1 Virtually partition the data into these many parts.
- `part.index` int, optional, default=0 The *i*-th virtual partition to be read.
- `device.id` int, optional, default=0 The device id used to create context for internal NDArray. Setting device_id to -1 will create Context::CPU(0). Setting device_id to valid positive device id will create Context::CPUPinned(device_id). Default is 0.
- `shuffle.chunk.size` long (non-negative), optional, default=0 The data shuffle buffer size in MB. Only valid if shuffle is true.
- `shuffle.chunk.seed` int, optional, default=0 The random seed for shuffling
- `seed.aug` int or None, optional, default=None Random seed for augmentations.
- `shuffle` boolean, optional, default=0 Whether to shuffle data randomly or not.
- `seed` int, optional, default=0 The random seed.
- `batch.size` int (non-negative), required Batch size.
- `round.batch` boolean, optional, default=1 Whether to use round robin to handle overflow batch or not.
- `prefetch.buffer` long (non-negative), optional, default=4 Maximum number of batches to prefetch.
- `ctx` 'cpu', 'gpu', optional, default='gpu' Context data loader optimized for.
- `dtype` None, 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', optional, default=None Output data type. “None” means no change.
- `resize` int, optional, default=-1 Down scale the shorter edge to a new size before applying other augmentations.
**rand.crop**  
boolean, optional, default=0  
If or not randomly crop the image

**random.resized.crop**  
boolean, optional, default=0  
If or not perform random resized cropping on the image, as a standard preprocessing for resnet training on ImageNet data.

**max.rotate.angle**  
int, optional, default=’0’  
Rotate by a random degree in ‘[-v, v]’

**max.aspect.ratio**  
float, optional, default=0  
Change the aspect (namely width/height) to a random value.  
If min_aspect_ratio is None then the aspect ratio is sampled from  
[1 - max_aspect_ratio, 1 + max_aspect_ratio], else it is in “[min_aspect_ratio, max_aspect_ratio]”

**min.aspect.ratio**  
float or None, optional, default=None  
Change the aspect (namely width/height) to a random value in “[min_aspect_ratio, max_aspect_ratio]”

**max.shear.ratio**  
float, optional, default=0  
Apply a shear transformation (namely “(x,y)->(x+my,y)”)  
with “m” randomly chose from “[max_shear_ratio, max_shear_ratio]”

**max.crop.size**  
int, optional, default=’-1’  
Crop both width and height into a random size in  
“[min_crop_size, max_crop_size].”Ignored if “random_resized_crop” is True.

**min.crop.size**  
int, optional, default=’-1’  
Crop both width and height into a random size in  
“[min_crop_size, max_crop_size].”Ignored if “random_resized_crop” is True.

**max.random.scale**  
float, optional, default=1  
Resize into “[width*s, height*s]” with “s” randomly chosen from “[min_random_scale, max_random_scale]”. Ignored if “random_resized_crop” is True.

**min.random.scale**  
float, optional, default=1  
Resize into “[width*s, height*s]” with “s” randomly chosen from “[min_random_scale, max_random_scale]”Ignored if “random_resized_crop” is True.

**max.random.area**  
float, optional, default=1  
Change the area (namely width * height) to a random value in “[min_random_area, max_random_area]”. Ignored if “random_resized_crop” is False.

**min.random.area**  
float, optional, default=1  
Change the area (namely width * height) to a random value in “[min_random_area, max_random_area]”. Ignored if “random_resized_crop” is False.

**max.img.size**  
float, optional, default=1e+10  
Set the maximal width and height after all resize and rotate argumentation are applied

**min.img.size**  
float, optional, default=0  
Set the minimal width and height after all resize and rotate argumentation are applied

**brightness**  
float, optional, default=0  
Add a random value in “[−brightness, brightness]” to the brightness of image.

**contrast**  
float, optional, default=0  
Add a random value in “[−contrast, contrast]” to the contrast of image.
saturation float, optional, default=0 Add a random value in “[−saturation, saturation]” to the saturation of image.

c PCA.noise float, optional, default=0 Add PCA based noise to the image.

random.h int, optional, default=’0’ Add a random value in “[−random_h, random_h]” to the H channel in HSL color space.

random.s int, optional, default=’0’ Add a random value in “[−random_s, random_s]” to the S channel in HSL color space.

random.l int, optional, default=’0’ Add a random value in “[−random_l, random_l]” to the L channel in HSL color space.

rotate int, optional, default=’-1’ Rotate by an angle. If set, it overwrites the “max_rotate_angle” option.

fill.value int, optional, default=’255’ Set the padding pixels value to “fill_value“.

data.shape Shape(tuple), required The shape of a output image.

inter.method int, optional, default=’1’ The interpolation method: 0-NN 1-bilinear 2-cubic 3-area 4-lanczos4 9-auto 10-rand.

pad int, optional, default=’0’ Change size from “[width, height]” into “[pad + width + pad, pad + height + pad]” by padding pixes

Details

This iterator is identical to “ImageRecordIter” except for using “uint8” as the data type instead of “float”.

Defined in src/io/iter_image_recordio_2.cc:L930

Value

iter The result mx.dataiter

mx.io.ImageRecordUInt8Iter_v1

Iterating on image RecordIO files

Description

.. note::

Usage

mx.io.ImageRecordUInt8Iter_v1(...)
Arguments

path.imglist  string, optional, default="" Path to the image list (.lst) file. Generally created with tools/im2rec.py. Format (Tab separated): <index of record> <one or more labels> <relative path from root folder>.

path.imgrec  string, optional, default="" Path to the image RecordIO (.rec) file or a directory path. Created with tools/im2rec.py.

path.imgidx  string, optional, default="" Path to the image RecordIO index (.idx) file. Created with tools/im2rec.py.

aug.seq  string, optional, default='aug_default' The augmenter names to represent sequence of augmenters to be applied, seperated by comma. Additional keyword parameters will be seen by these augmenters.

label.width  int, optional, default='1' The number of labels per image.

preprocess.threads  int, optional, default='4' The number of threads to do preprocessing.

verbose  boolean, optional, default=1 If or not output verbose information.

num.parts  int, optional, default='1' Virtually partition the data into these many parts.

part.index  int, optional, default='0' The *i*-th virtual partition to be read.

device.id  int, optional, default='0' The device id used to create context for internal NDArray. Setting device_id to -1 will create Context::CPU(0). Setting device_id to valid positive device id will create Context::CPUPinned(device_id). Default is 0.

shuffle.chunk.size  long (non-negative), optional, default=0 The data shuffle buffer size in MB. Only valid if shuffle is true.

shuffle.chunk.seed  int, optional, default='0' The random seed for shuffling

seed.aug  int or None, optional, default='None' Random seed for augmentations.

shuffle  boolean, optional, default=0 Whether to shuffle data randomly or not.

seed  int, optional, default='0' The random seed.

batch.size  int (non-negative), required Batch size.

round.batch  boolean, optional, default=1 Whether to use round robin to handle overflow batch or not.

prefetch.buffer  long (non-negative), optional, default=4 Maximum number of batches to prefetch.

ctx  'cpu', 'gpu', optional, default='gpu' Context data loader optimized for.

dtype  None, 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8',optional, default='None' Output data type. “None“ means no change.

resize  int, optional, default='-1' Down scale the shorter edge to a new size before applying other augmentations.

rand.crop  boolean, optional, default=0 If or not randomly crop the image

random.resized.crop  boolean, optional, default=0 If or not perform random resized cropping on the image, as a standard preprocessing for resnet training on ImageNet data.
max.rotate.angle
int, optional, default=0' Rotate by a random degree in “[-v, v]”

max.aspect.ratio
float, optional, default=0 Change the aspect (namely width/height) to a random value. If min_aspect_ratio is None then the aspect ratio is sampled from [1 - max_aspect_ratio, 1 + max_aspect_ratio], else it is in “[min_aspect_ratio, max_aspect_ratio]”

min.aspect.ratio
float or None, optional, default=None Change the aspect (namely width/height) to a random value in “[min_aspect_ratio, max_aspect_ratio]”

max.shear.ratio
float, optional, default=0 Apply a shear transformation (namely “(x,y)->(x+my,y)”) with “m” randomly chosen from “[-max_shear_ratio, max_shear_ratio]”

max.crop.size
int, optional, default=-1’ Crop both width and height into a random size in “[min_crop_size, max_crop_size].” Ignored if “random_resized_crop” is True.

min.crop.size
int, optional, default=-1’ Crop both width and height into a random size in “[min_crop_size, max_crop_size].” Ignored if “random_resized_crop” is True.

max.random.scale
float, optional, default=1 Resize into “[width*s, height*s]” with ‘s’ randomly chosen from “[min_random_scale, max_random_scale].” Ignored if “random_resized_crop” is True.

min.random.scale
float, optional, default=1 Resize into “[width*s, height*s]” with ‘s’ randomly chosen from “[min_random_scale, max_random_scale].” Ignored if “random_resized_crop” is True.

max.random.area
float, optional, default=1 Change the area (namely width * height) to a random value in “[min_random_area, max_random_area].” Ignored if “random_resized_crop” is False.

min.random.area
float, optional, default=1 Change the area (namely width * height) to a random value in “[min_random_area, max_random_area].” Ignored if “random_resized_crop” is False.

max.img.size
float, optional, default=1e+10 Set the maximal width and height after all resize and rotate argumentation are applied.

min.img.size
float, optional, default=0 Set the minimal width and height after all resize and rotate argumentation are applied.

brightness
float, optional, default=0 Add a random value in “[-brightness, brightness]” to the brightness of image.

contrast
float, optional, default=0 Add a random value in “[-contrast, contrast]” to the contrast of image.

saturation
float, optional, default=0 Add a random value in “[-saturation, saturation]” to the saturation of image.

cpyca.noise
float, optional, default=0 Add PCA based noise to the image.

random.h
int, optional, default=0’ Add a random value in “[-random_h, random_h]” to the H channel in HSL color space.
mx.io.LibSVMIter

random.s
int, optional, default='0' Add a random value in "[-random_s, random_s]" to the S channel in HSL color space.

random.l
int, optional, default='0' Add a random value in "[-random_l, random_l]" to the L channel in HSL color space.

rotate
int, optional, default='-1' Rotate by an angle. If set, it overwrites the "max_rotate_angle" option.

fill.value
int, optional, default='255' Set the padding pixels value to "fill_value".

data.shape
Shape(tuple), required The shape of output image.

inter.method
int, optional, default='1' The interpolation method: 0-NN 1-bilinear 2-cubic 3-area 4-lanczos4 9-auto 10-rand.

pad
int, optional, default='0' Change size from "[width, height]" into "[pad + width + pad, pad + height + pad]" by padding pixels

Details

"ImageRecordU1Int8Iter_v1" is deprecated. Use "ImageRecordU1Int8Iter" instead.

This iterator is identical to "ImageRecordIter" except for using "uint8" as the data type instead of "float".

Defined in src/io/iter_image_recordio.cc:L377

Value

iter The result mx.dataiter

mx.io.LibSVMIter Returns the LibSVM iterator which returns data with ‘csr’ storage type. This iterator is experimental and should be used with care.

Description

The input data is stored in a format similar to LibSVM file format, except that the indices are expected to be zero-based instead of one-based, and the column indices for each row are expected to be sorted in ascending order. Details of the LibSVM format are available here. <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>

Usage

mx.io.LibSVMIter(...)
Arguments

- **data.libsvm**: string, required. The input zero-base indexed LibSVM data file or a directory path.
- **data.shape**: Shape(tuple), required. The shape of one example.
- **label.libsvm**: string, optional, default='NULL'. The input LibSVM label file or a directory path. If NULL, all labels will be read from “data_libsvm”.
- **label.shape**: Shape(tuple), optional, default=[1]. The shape of one label.
- **num.parts**: int, optional, default='1'. Partition the data into multiple parts.
- **part.index**: int, optional, default='0'. The index of the part will read.
- **batch.size**: int (non-negative), required. Batch size.
- **round.batch**: boolean, optional, default=1. Whether to use round robin to handle overflow batch or not.
- **prefetch.buffer**: long (non-negative), optional, default=4. Maximum number of batches to prefetch.
- **ctx**: 'cpu', 'gpu', optional, default='gpu'. Context data loader optimized for.
- **dtype**: None, 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', optional, default='None'. Output data type. “None” means no change.

Details

The ‘data_shape’ parameter is used to set the shape of each line of the data. The dimension of both ‘data_shape’ and ‘label_shape’ are expected to be 1.

The ‘data_libsvm’ parameter is used to set the path input LibSVM file. When it is set to a directory, all files in the directory will be read.

When ‘label_libsvm’ is set to “NULL“, both data and label are read from the file specified by ‘data_libsvm’. In this case, the data is stored in ‘csr’ storage type, while the label is a 1D dense array.

The ‘LibSVMIter’ only support ‘round_batch’ parameter set to “True”. Therefore, if ‘batch_size’ is 3 and there are 4 total rows in libsvm file, 2 more examples are consumed at the first round.

When ‘num_parts’ and ‘part_index’ are provided, the data is split into ‘num_parts’ partitions, and the iterator only reads the ‘part_index’-th partition. However, the partitions are not guaranteed to be even.

“reset()“ is expected to be called only after a complete pass of data.

Example:

```
# Contents of libsvm file “data.t”. 1.0 0:0.5 2:1.2 -2.0 -3.0 0:0.6 1:2.4 2:1.2 4 2:-1.2
# Creates a ‘LibSVMIter’ with ‘batch_size’=3. »> data_iter = mx.io.LibSVMIter(data_libsvm = 'data.t', data_shape = (3,), batch_size = 3) # The data of the first batch is stored in csr storage type »> batch = data_iter.next() »> csr = batch.data[0] <CSRNDArray 3x3 @cpu(0)> »> csr.asnumpy() 
[[ 0.5 0. 1.2 ] [ 0. 0. 0. ] [ 0.6 2.4 1.2]] # The label of first batch »> label = batch.label[0] »> label [ 1. -2. -3.]

»> second_batch = data_iter.next() # The data of the second batch »> second_batch.data[0].asnumpy() 
[[ 0. 0. -1.2] [ 0.5 0. 1.2] [ 0. 0. 0.]] # The label of the second batch »> second_batch.label[0].asnumpy() 
[ 4. 1. -2.]
```
mx.io.MNISTIter

Iterating on the MNIST dataset.

Description

One can download the dataset from http://yann.lecun.com/exdb/mnist/

Usage

mx.io.MNISTIter(...)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>string, optional, default='./train-images-idx3-ubyte' Dataset Param: Mnist image path.</td>
</tr>
<tr>
<td>label</td>
<td>string, optional, default='./train-labels-idx1-ubyte' Dataset Param: Mnist label path.</td>
</tr>
<tr>
<td>batch.size</td>
<td>int, optional, default='128' Batch Param: Batch Size.</td>
</tr>
<tr>
<td>shuffle</td>
<td>boolean, optional, default=1 Augmentation Param: Whether to shuffle data.</td>
</tr>
<tr>
<td>flat</td>
<td>boolean, optional, default=0 Augmentation Param: Whether to flat the data into 1D.</td>
</tr>
<tr>
<td>seed</td>
<td>int, optional, default='0' Augmentation Param: Random Seed.</td>
</tr>
<tr>
<td>silent</td>
<td>boolean, optional, default=0 Auxiliary Param: Whether to print out data info.</td>
</tr>
<tr>
<td>num.parts</td>
<td>int, optional, default='1' partition the data into multiple parts</td>
</tr>
<tr>
<td>part.index</td>
<td>int, optional, default='0' the index of the part will read</td>
</tr>
</tbody>
</table>
mx.lr_scheduler.FactorScheduler

Learning rate scheduler. Reduction based on a factor value.

Usage

    mx.lr_scheduler.FactorScheduler(
        step,
        factor_val,
        stop_factor_lr = 1e-08,
        verbose = TRUE
    )

mx.kv.create

Create a mxnet KVStore.

Description

Create a mxnet KVStore.

Arguments

type

    string(default="local") The type of kvstore.

Value

The kvstore.

mx.lr_scheduler.FactorScheduler

Learning rate scheduler. Reduction based on a factor value.
mx.lr_scheduler.MultiFactorScheduler

Arguments

step (integer) Schedule learning rate after n updates
factor (double) The factor for reducing the learning rate

Value

scheduler function

mx.lr_scheduler.MultiFactorScheduler

Multifactor learning rate scheduler. Reduction based on a factor value at different steps.

Description

Multifactor learning rate scheduler. Reduction based on a factor value at different steps.

Usage

mx.lr_scheduler.MultiFactorScheduler(
    step,
    factor_val,
    stop_factor_lr = 1e-08,
    verbose = TRUE
)

Arguments

step (array of integer) Schedule learning rate after n updates
factor (double) The factor for reducing the learning rate

Value

scheduler function
mx.metric.accuracy  

*Accuracy metric for classification*

**Description**

Accuracy metric for classification

**Usage**

`mx.metric.accuracy`

**Format**

An object of class `mx.metric` of length 3.

mx.metric.custom  

*Helper function to create a customized metric*

**Description**

Helper function to create a customized metric

**Usage**

`mx.metric.custom(name, feval)`

mx.metric.logistic_acc  

*Accuracy metric for logistic regression*

**Description**

Accuracy metric for logistic regression

**Usage**

`mx.metric.logistic_acc`

**Format**

An object of class `mx.metric` of length 3.
mx.metric.logloss  

LogLoss metric for logistic regression

Description

LogLoss metric for logistic regression

Usage

mx.metric.logloss

Format

An object of class mx.metric of length 3.

mx.metric.mae  

MAE (Mean Absolute Error) metric for regression

Description

MAE (Mean Absolute Error) metric for regression

Usage

mx.metric.mae

Format

An object of class mx.metric of length 3.

mx.metric.mse  

MSE (Mean Squared Error) metric for regression

Description

MSE (Mean Squared Error) metric for regression

Usage

mx.metric.mse

Format

An object of class mx.metric of length 3.
mx.metric.Perplexity  

**Description**

Perplexity metric for language model

**Usage**

mx.metric.Perplexity

**Format**

An object of class mx.metric of length 3.

---

mx.metric.rmse  

**Description**

RMSE (Root Mean Squared Error) metric for regression

**Usage**

mx.metric.rmse

**Format**

An object of class mx.metric of length 3.

---

mx.metric.rmsle  

**Description**

RMSLE (Root Mean Squared Logarithmic Error) metric for regression

**Usage**

mx.metric.rmsle

**Format**

An object of class mx.metric of length 3.
mx.metric.top_k_accuracy

Top-k accuracy metric for classification

Description
Top-k accuracy metric for classification

Usage
mx.metric.top_k_accuracy

Format
An object of class mx.metric of length 3.

mx.mlp

Convenience interface for multiple layer perceptron

Description
Convenience interface for multiple layer perceptron

Usage
mx.mlp(
    data,
    label,
    hidden_node = 1,
    out_node,
    dropout = NULL,
    activation = "tanh",
    out_activation = "softmax",
    ctx = mx.ctx.default(),
    ...
)

Arguments

data the input matrix. Only mx.io.DataIter and R array/matrix types supported.
label the training label. Only R array type supported.
hidden_node a vector containing number of hidden nodes on each hidden layer as well as the output layer.
out_node the number of nodes on the output layer.
mx.model.buckets

dropout a number in \([0,1)\) containing the dropout ratio from the last hidden layer to the output layer.

activation either a single string or a vector containing the names of the activation functions.

out_activation a single string containing the name of the output activation function.

ctx whether train on cpu (default) or gpu.

... other parameters passing to \texttt{mx.model.FeedForward.create/}

eval.metric the evaluation metric/

**Examples**

```r
require(mlbench)
data(Sonar, package="mlbench")
Sonar[,61] = as.numeric(Sonar[,61])-1
train.ind = c(1:50, 100:150)
train.x = data.matrix(Sonar[train.ind, 1:60])
train.y = Sonar[train.ind, 61]
test.x = data.matrix(Sonar[-train.ind, 1:60])
test.y = Sonar[-train.ind, 61]
model = mx.mlp(train.x, train.y, hidden_node = 10, out_node = 2, out_activation = "softmax",
learning.rate = 0.1)
preds = predict(model, test.x)
```

mx.model.buckets \hspace{1cm} \textit{Train RNN with bucket support}

**Description**

Train RNN with bucket support

**Usage**

```r
mx.model.buckets(
  symbol,
  train.data,
  eval.data = NULL,
  metric = NULL,
  arg.params = NULL,
  aux.params = NULL,
  fixed.params = NULL,
  num.round = 1,
  begin.round = 1,
  initializer = mx.init.uniform(0.01),
  optimizer = "sgd",
  ctx = NULL,
  batch.end.callback = NULL,
)```
Arguments
symbol | Symbol or list of Symbols representing the model
train.data | Training data created by mx.io.bucket.iter
eval.data | Evaluation data created by mx.io.bucket.iter
num.round | int, number of epoch
verbose

Description
Create a MXNet Feedforward neural net model with the specified training.

Usage
mx.model.FeedForward.create(
    symbol,
    X,
    y = NULL,
    ctx = NULL,
    begin.round = 1,
    num.round = 10,
    optimizer = "sgd",
    initializer = mx.init.uniform(0.01),
    eval.data = NULL,
    eval.metric = NULL,
    epoch.end.callback = NULL,
    batch.end.callback = NULL,
    array.batch.size = 128,
    array.layout = "auto",
    kvstore = "local",
    verbose = TRUE,
    arg.params = NULL,
    aux.params = NULL,
    input.names = NULL,
    output.names = NULL,
mx.model.FeedForward.create

fixed.param = NULL,
allow.extra.params = FALSE,
metric_cpu = TRUE,
...
)

Arguments

symbol
The symbolic configuration of the neural network.

X
mx.io.DataIter or R array/matrix The training data.

y
R array, optional label of the data This is only used when X is R array.

ctx
mx.context or list of mx.context, optional The devices used to perform training.

begin.round
integer (default=1) The initial iteration over the training data to train the model.

num.round
integer (default=10) The number of iterations over training data to train the model.

optimizer
string, default="sgd" The optimization method.

initializer
, initializer object. default=mx.init.uniform(0.01) The initialization scheme for parameters.

eval.data
mx.io.DataIter or list(data=R.array, label=R.array), optional The validation set used for validation evaluation during the progress.

eval.metric
function, optional The evaluation function on the results.

epoch.end.callback
function, optional The callback when iteration ends.

batch.end.callback
function, optional The callback when one mini-batch iteration ends.

array.batch.size
integer (default=128) The batch size used for R array training.

array.layout
can be "auto", "colmajor", "rowmajor", (default=auto) The layout of array. "rowmajor" is only supported for two dimensional array. For matrix, "rowmajor" means dim(X) = c(nexample, nfeatures), "colmajor" means dim(X) = c(nfeatures, nexample) "auto" will auto detect the layout by match the feature size, and will report error when X is a square matrix to ask user to explicitly specify layout.

e_kvstore
string (default="local") The parameter synchronization scheme in multiple devices.

verbose
logical (default=TRUE) Specifies whether to print information on the iterations during training.

arg.params
list, optional Model parameter, list of name to NDArray of net’s weights.

aux.params
list, optional Model parameter, list of name to NDArray of net’s auxiliary states.

input.names
optional The names of the input symbols.

output.names
optional The names of the output symbols.

fixed.param
The parameters to be fixed during training. For these parameters, not gradients will be calculated and thus no space will be allocated for the gradient.

allow.extra.params
Whether allow extra parameters that are not needed by symbol. If this is TRUE, no error will be thrown when arg_params or aux_params contain extra parameters that is not needed by the executor.
Value

model A trained mxnet model.

mx.model.init.params  
**Parameter initialization**

Description

Parameter initialization

Usage

```
xm.model.init.params(symbol, input.shape, output.shape, initializer, ctx)
```

Arguments

- `symbol`: The symbolic configuration of the neural network.
- `input.shape`: The shape of the input for the neural network.
- `output.shape`: The shape of the output for the neural network. It can be NULL.
- `initializer`: initializer object. The initialization scheme for parameters.
- `ctx`: mx.context. The devices used to perform initialization.

mx.model.load  
**Load model checkpoint from file.**

Description

Load model checkpoint from file.

Usage

```
xm.model.load(prefix, iteration)
```

Arguments

- `prefix`: string prefix of the model name
- `iteration`: integer Iteration number of model we would like to load.
**mx.model.save**

_Save model checkpoint into file._

**Description**

Save model checkpoint into file.

**Usage**

```python
mx.model.save(model, prefix, iteration)
```

**Arguments**

- `model` (The feedforward model to be saved.)
- `prefix` (string prefix of the model name)
- `iteration` (integer Iteration number of model we would like to load.)

**mx.nd.abs**

_Returns element-wise absolute value of the input._

**Description**

Example::

**Arguments**

- `data` (NDArray-or-Symbol The input array.)

**Details**

```python
abs([-2, 0, 3]) = [2, 0, 3]
```

The storage type of “abs” output depends upon the input storage type:

- abs(default) = default - abs(row_sparse) = row_sparse - abs(csr) = csr

Defined in `src/operator/tensor/elemwise_unary_op_basic.cc:L720`

**Value**

- `out` (The result mx.ndarray)
mx.nd.Activation

Applies an activation function element-wise to the input.

Description

The following activation functions are supported:

Arguments

data NDArray-or-Symbol The input array.
act.type 'relu', 'sigmoid', 'softrelu', 'softsign', 'tanh', required Activation function to be applied.

Details

- 'relu': Rectified Linear Unit, :math:'y = \max(x, 0)' - 'sigmoid': :math:'y = \frac{1 + \exp(-x)}{1 + \exp(x)}' - 'tanh': Hyperbolic tangent, :math:'y = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}' - 'softrelu': Soft ReLU, or SoftPlus, :math:'y = \log(1 + \exp(x))' - 'softsign': :math:'y = \frac{x}{1 + |x|}'

Defined in src/operator/nn/activation.cc:L165

Value

out The result mx.ndarray

mx.nd.adam.update

Update function for Adam optimizer. Adam is seen as a generalization of AdaGrad.

Description

Adam update consists of the following steps, where g represents gradient and m, v are 1st and 2nd order moment estimates (mean and variance).

Arguments

weight NDArray-or-Symbol Weight
grad NDArray-or-Symbol Gradient
mean NDArray-or-Symbol Moving mean
var NDArray-or-Symbol Moving variance
lr float, required Learning rate
beta1 float, optional, default=0.899999976 The decay rate for the 1st moment estimates.
beta2 float, optional, default=0.999000013 The decay rate for the 2nd moment estimates.

epsilon float, optional, default=9.99999994e-09 A small constant for numerical stability.

wd float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.

rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.

clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).

lazy.update boolean, optional, default=1 If true, lazy updates are applied if gradient’s stype is row_sparse and all of w, m and v have the same stype.

Details

.. math::
   g_t = \nabla J(W_{t-1}) \ m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \ v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \ W_t = W_{t-1} - \alpha \frac{m_t}{\sqrt v_t + \epsilon}

It updates the weights using::

m = beta1*m + (1-beta1)*grad v = beta2*v + (1-beta2)*(grad**2) w += - learning_rate * m / (sqrt(v) + epsilon)

However, if grad’s storage type is “row_sparse”, “lazy_update” is True and the storage type of weight is the same as those of m and v, only the row slices whose indices appear in grad.indices are updated (for w, m and v):


Defined in src/operator/optimizer_op.cc:L679

Value

out The result mx.ndarray

mx.nd.add.n Adds all input arguments element-wise.

Description

.. math:: add\_n(a_1, a_2, ..., a_n) = a_1 + a_2 + ... + a_n

Arguments

args NDArray-or-Symbol[] Positional input arguments
mx.nd.all.finite

**Details**

“add_n” is potentially more efficient than calling “add” by ‘n’ times.

The storage type of “add_n” output depends on storage types of inputs

- add_n(row_sparse, row_sparse, ..) = row_sparse
- add_n(default, csr, default) = default
- add_n(any input combinations longer than 4 (>4) with at least one default type) = default
- otherwise, “add_n” falls all inputs back to default storage and generates default storage

Defined in src/operator/tensor/elemwise_sum.cc:L155

**Value**

out The result mx.ndarray

---

mx.nd.all.finite **Check if all the float numbers in the array are finite (used for AMP)**

**Description**

Defined in src/operator/contrib/all_finite.cc:L101

**Arguments**

data NDArray Array

init.output boolean, optional, default=1 Initialize output to 1.

**Value**

out The result mx.ndarray

---

mx.nd.amp.cast **Cast function between low precision float/FP32 used by AMP.**

**Description**

It casts only between low precision float/FP32 and does not do anything for other types.

**Arguments**

data NDArray-or-Symbol The input.

dtype 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', required

**Details**

Defined in src/operator/tensor/amp_cast.cc:L121
mx.nd.arccos

Returns element-wise inverse cosine of the input array.

Description

The input should be in range ‘[-1, 1]’. The output is in the closed interval :math:`[0, \pi]`

Arguments

- data NDArray-or-Symbol The input array.

Details

.. math:: \text{arccos}([-1, -0.707, 0, 0.707, 1]) = [\pi, 3\pi/4, \pi/2, \pi/4, 0]

The storage type of “arccos” output is always dense

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L233

Value

out The result mx.ndarray

mx.nd.amp.multicast

Cast function used by AMP, that casts its inputs to the common widest type.

Description

It casts only between low precision float/FP32 and does not do anything for other types.

Arguments

- data NDArray-or-Symbol[] Weights
- num.outputs int, required Number of input/output pairs to be casted to the widest type.
- cast.narrow boolean, optional, default=0 Whether to cast to the narrowest type

Details

Defined in src/operator/tensor/amp_cast.cc:L165

Value

out The result mx.ndarray
mx.nd.arccosh

Returns the element-wise inverse hyperbolic cosine of the input array, computed element-wise.

Description
The storage type of “arccosh” output is always dense

Arguments
- data NDArray-or-Symbol The input array.

Details
Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L535

Value
- out The result mx.ndarray

mx.nd.arcsin

Returns element-wise inverse sine of the input array.

Description
The input should be in the range ‘[-1, 1]’. The output is in the closed interval of [:math:`\{-\pi/2, \pi/2\}`.

Arguments
- data NDArray-or-Symbol The input array.

Details
.. math:: 
\begin{align*} 
arcsin([-1, -0.707, 0, 0.707, 1]) &= [-\pi/2, -\pi/4, 0, \pi/4, \pi/2] 
\end{align*}

The storage type of “arcsin” output depends upon the input storage type:
- arcsin(default) = default
- arcsin(row_sparse) = row_sparse
- arcsin(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L187

Value
- out The result mx.ndarray
mx.nd.arcsinh

Returns the element-wise inverse hyperbolic sine of the input array, computed element-wise.

Description
The storage type of “arcsinh” output depends upon the input storage type:

Arguments
data NDArray-or-Symbol The input array.

Details
- arcsinh(default) = default - arcsinh(row_sparse) = row_sparse - arcsinh(csr) = csr
Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L494

Value
out The result mx.ndarray

mx.nd.arctan

Returns element-wise inverse tangent of the input array.

Description
The output is in the closed interval :math:`[-\pi/2, \pi/2]`

Arguments
data NDArray-or-Symbol The input array.

Details
.. math:: \arctan([-1, 0, 1]) = [-\pi/4, 0, \pi/4]
The storage type of “arctan” output depends upon the input storage type:
- arctan(default) = default - arctan(row_sparse) = row_sparse - arctan(csr) = csr
Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L282

Value
out The result mx.ndarray
**mx.nd.arctanh**  
*Returns the element-wise inverse hyperbolic tangent of the input array, computed element-wise.*

**Description**

The storage type of “arctanh” output depends upon the input storage type:

**Arguments**

- `data`  
  NDArray-or-Symbol The input array.

**Details**

- `arctanh(default) = default`  
  - `arctanh(row_sparse) = row_sparse`  
  - `arctanh(csr) = csr`

Defined in `src/operator/tensor/elemwise_unary_op_trig.cc:L579`

**Value**

- `out` The result `mx.ndarray`

**mx.nd.argmax**  
*Returns indices of the maximum values along an axis.*

**Description**

In the case of multiple occurrences of maximum values, the indices corresponding to the first occurrence are returned.

**Arguments**

- `data`  
  NDArray-or-Symbol The input
- `axis`  
  int or None, optional, default=`None` The axis along which to perform the reduction. Negative values means indexing from right to left. “Requires axis to be set as int, because global reduction is not supported yet.”
- `keepdims`  
  boolean, optional, default=0 If this is set to ‘True’, the reduced axis is left in the result as dimension with size one.
Details

Examples::

data = [[0., 1., 2.], [3., 4., 5.]]

// argmax along axis 0 argmax(data, axis=0) = [1., 1., 1.]

// argmax along axis 1 argmax(data, axis=1) = [2., 2.]

// argmax along axis 1 keeping same dims as an input array argmax(data, axis=1, keepdims=True) = [[2.], [2.]]

Defined in src/operator/tensor/broadcast_reduce_op_index.cc:L52

Value

out The result mx.ndarray

mx.nd.argmax.channel  Returns argmax indices of each channel from the input array.

Description

The result will be an NDArray of shape (num_channel,).

Arguments

data NDArray-or-Symbol The input array

Details

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples::

data = [[0., 1., 2.], [3., 4., 5.]]

argmax_channel(data) = [2., 2.]

Defined in src/operator/tensor/broadcast_reduce_op_index.cc:L97

Value

out The result mx.ndarray
### Description

In the case of multiple occurrences of minimum values, the indices corresponding to the first occurrence are returned.

### Arguments

- **data**: NDArray-or-Symbol The input
- **axis**: int or None, optional, default=`None` The axis along which to perform the reduction. Negative values mean indexing from right to left. “Requires axis to be set as int, because global reduction is not supported yet.”
- **keepdims**: boolean, optional, default=0 If this is set to ‘True’, the reduced axis is left in the result as dimension with size one.

### Details

Examples:

```python
x = [[ 0., 1., 2.], [ 3., 4., 5.]]
# argmin along axis 0 argmin(x, axis=0) = [ 0., 0., 0.]
# argmin along axis 1 argmin(x, axis=1) = [ 0., 0.]
# argmin along axis 1 keeping same dims as an input array argmin(x, axis=1, keepdims=True) = [[ 0.], [ 0.]]
```

Defined in `src/operator/tensor/broadcast_reduce_op_index.cc:L77`

### Value

- **out**: The result mx.ndarray

---

### Description

This function performs sorting along the given axis and returns an array of indices having same shape as an input array that index data in sorted order.

### Return

- **mx.nd.argsort**: Returns the indices that would sort an input array along the given axis.
Arguments

data NDArray-or-Symbol The input array
axis int or None, optional, default=-1 Axis along which to sort the input tensor. If
not given, the flattened array is used. Default is -1.
is.ascend boolean, optional, default=1 Whether to sort in ascending or descending order.
dtype 'float16', 'float32', 'float64', 'int32', 'int64', 'uint8', optional, default='float32' DType of the output indices. It is only valid when ret_typ is "indices" or "both". An error will be raised if the selected data type cannot precisely represent the indices.

Details

Examples:

x = [[[ 0.3, 0.2, 0.4], [ 0.1, 0.3, 0.2]]
// sort along axis -1 argsort(x) = [[[ 1., 0., 2.], [ 0., 2., 1.]]
// sort along axis 0 argsort(x, axis=0) = [[[ 1., 0., 1.], [ 0., 1., 0.]]
// flatten and then sort argsort(x, axis=None) = [ 3., 1., 5., 0., 4., 2.]

Defined in src/operator/tensor/ordering_op.cc:L185

Value

out The result mx.ndarray

---

mx.nd.array Create a new mx.ndarray that copies the content from src on ctx.

Description

Create a new mx.ndarray that copies the content from src on ctx.

Usage

mx.nd.array(src.array, ctx = NULL)

Arguments

src.array Source array data of class array, vector or matrix.
ctx optional The context device of the array. mx.ctx.default() will be used in default.

Value

An mx.ndarray

An Rcpp_MXNDArray object
Examples

mat = mx.nd.array(x)
mat = 1 - mat + (2 * mat)/(mat + 0.5)
as.array(mat)

mx.nd.batch.dot

"batch_dot" is used to compute dot product of "x" and "y" when "x" and "y" are data in batch, namely N-D (N >= 3) arrays in shape of `(B0, ..., B_i, :,:)`.

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lhs</td>
<td>NDArray-or-Symbol</td>
<td>The first input</td>
</tr>
<tr>
<td>rhs</td>
<td>NDArray-or-Symbol</td>
<td>The second input</td>
</tr>
<tr>
<td>transpose.a</td>
<td>boolean, optional</td>
<td>If true then transpose the first input before dot.</td>
</tr>
<tr>
<td>transpose.b</td>
<td>boolean, optional</td>
<td>If true then transpose the second input before dot.</td>
</tr>
<tr>
<td>forward.stype</td>
<td>None, 'csr', 'default', 'row_sparse', optional, default='None'</td>
<td>The desired storage type of the forward output given by user, if the combination of input storage types and this hint does not match any implemented ones, the dot operator will perform fallback operation and still produce an output of the desired storage type.</td>
</tr>
</tbody>
</table>

Details

For example, given "x" with shape `(B_0, ..., B_i, N, M)` and "y" with shape `(B_0, ..., B_i, M, K)`, the result array will have shape `(B_0, ..., B_i, N, K)`, which is computed by:

```
batch_dot(x,y)[b_0, ..., b_i, :, :] = dot(x[b_0, ..., b_i, :, :], y[b_0, ..., b_i, :, :])
```

Defined in src/operator/tensor/dot.cc:L127

Value

out The result mx.ndarray
mx.nd.BatchNorm

Batch normalization.

Description

Normalizes a data batch by mean and variance, and applies a scale “gamma” as well as offset “beta”.

Arguments

data  NDArray-or-Symbol Input data to batch normalization

gamma  NDArray-or-Symbol gamma array

beta  NDArray-or-Symbol beta array

moving.mean  NDArray-or-Symbol running mean of input

moving.var  NDArray-or-Symbol running variance of input

eps  double, optional, default=0.0010000000474974513 Epsilon to prevent div 0. Must be no less than CUDNN_BN_MIN_EPSILON defined in cudnn.h when using cudnn (usually 1e-5)

mx.nd.batch.take

Takes elements from a data batch.

Description

.. note:: ‘batch_take’ is deprecated. Use ‘pick’ instead.

Arguments

a  NDArray-or-Symbol The input array

indices  NDArray-or-Symbol The index array

Details

Given an input array of shape “(d0, d1)” and indices of shape “(i0,)”, the result will be an output array of shape “(i0,)” with:

\[
\text{output}[i] = \text{input}[i, \text{indices}[i]]
\]

Examples:

\[
x = [[1., 2.], [3., 4.], [5., 6.]]
\]

// takes elements with specified indices
batch_take(x, [0,1,0]) = [1. 4. 5.]

Defined in src/operator/tensor/indexing_op.cc:L841

Value

out The result mx.ndarray
**mx.nd.BatchNorm**

- **momentum** float, optional, default=0.899999976 Momentum for moving average
- **fix.gamma** boolean, optional, default=1 Fix gamma while training
- **use.global.stats** boolean, optional, default=0 Whether use global moving statistics instead of local batch-norm. This will force change batch-norm into a scale shift operator.
- **output.mean.var** boolean, optional, default=0 Output the mean and inverse std
- **axis** int, optional, default='1' Specify which shape axis the channel is specified
- **cudnn.off** boolean, optional, default=0 Do not select CUDNN operator, if available
- **min.calib.range** float or None, optional, default=None The minimum scalar value in the form of float32 obtained through calibration. If present, it will be used to by quantized batch norm op to calculate primitive scale. Note: this calib_range is to calib bn output.
- **max.calib.range** float or None, optional, default=None The maximum scalar value in the form of float32 obtained through calibration. If present, it will be used to by quantized batch norm op to calculate primitive scale. Note: this calib_range is to calib bn output.

**Details**

Assume the input has more than one dimension and we normalize along axis 1. We first compute the mean and variance along this axis:

\[
\begin{align*}
\text{data\_mean}[i] &= \text{mean}(\text{data}[:,i,:,:\ldots]) \\
\text{data\_var}[i] &= \text{var}(\text{data}[:,i,:,:\ldots])
\end{align*}
\]

Then compute the normalized output, which has the same shape as input, as following:

\[
\begin{align*}
\text{out}[:,i,:,:\ldots] &= \frac{\text{data}[:,i,:,:\ldots]}{\sqrt{\text{data\_var}[i] + \epsilon}} \cdot \text{gamma}[i] + \text{beta}[i]
\end{align*}
\]

Both *mean* and *var* returns a scalar by treating the input as a vector.

Assume the input has size *k* on axis 1, then both “gamma” and “beta” have shape *(k,)*. If “output_mean_var” is set to be true, then outputs both “data_mean” and the inverse of “data_var”, which are needed for the backward pass. Note that gradient of these two outputs are blocked.

Besides the inputs and the outputs, this operator accepts two auxiliary states, “moving_mean” and “moving_var”, which are *k*-length vectors. They are global statistics for the whole dataset, which are updated by:

\[
\begin{align*}
\text{moving\_mean} &= \text{moving\_mean} \cdot \text{momentum} + \text{data\_mean} \cdot (1 - \text{momentum}) \\
\text{moving\_var} &= \text{moving\_var} \cdot \text{momentum} + \text{data\_var} \cdot (1 - \text{momentum})
\end{align*}
\]

If “use_global_stats” is set to be true, then “moving_mean” and “moving_var” are used instead of “data_mean” and “data_var” to compute the output. It is often used during inference.

The parameter “axis” specifies which axis of the input shape denotes the 'channel' (separately normalized groups). The default is 1. Specifying -1 sets the channel axis to be the last item in the input shape.
Both "gamma" and "beta" are learnable parameters. But if “fix_gamma” is true, then set “gamma”
to 1 and its gradient to 0.

.. Note:: When “fix_gamma“ is set to True, no sparse support is provided. If “fix_gamma is“ set to
False, the sparse tensors will fallback.

Defined in src/operator/nn/batch_norm.cc:L545

Value

out The result mx.ndarray

mx.nd.BatchNorm.v1  Batch normalization.

Description

This operator is DEPRECATED. Perform BatchNorm on the input.

Arguments

data NDArray-or-Symbol Input data to batch normalization
gamma NDArray-or-Symbol gamma array
beta NDArray-or-Symbol beta array
eps float, optional, default=0.00100000005 Epsilon to prevent div 0
momentum float, optional, default=0.899999976 Momentum for moving average
fix.gamma boolean, optional, default=1 Fix gamma while training
use.global.stats boolean, optional, default=0 Whether use global moving statistics instead of
local batch-norm. This will force change batch-norm into a scale shift operator.
output.mean.var boolean, optional, default=0 Output All,normal mean and var

Details

Normalizes a data batch by mean and variance, and applies a scale “gamma” as well as offset “beta”. Assume the input has more than one dimension and we normalize along axis 1. We first compute the mean and variance along this axis:

.. math::
\text{data}_{\text{mean}}[i] = \text{mean}(\text{data[:,i,:,:,...])} \ \ \ \ \ \text{data}_{\text{var}}[i] = \text{var}(\text{data[:,i,:,:,...])}

Then compute the normalized output, which has the same shape as input, as following:

.. math::
\text{out[:,i,:,:,...]} = \frac{\text{data[:,i,:,:,...]} - \text{data}_{\text{mean}}[i]}{\sqrt{\text{data}_{\text{var}}[i]} + \epsilon} \times \text{gamma}[i] + \text{beta}[i]

Both *mean* and *var* returns a scalar by treating the input as a vector.
Assume the input has size *(k,)* on axis 1, then both “gamma” and “beta” have shape *(k,)*. If “output_mean_var” is set to be true, then outputs both “data_mean” and “data_var” as well, which are needed for the backward pass.

Besides the inputs and the outputs, this operator accepts two auxiliary states, “moving_mean” and “moving_var”, which are *(k,)*-length vectors. They are global statistics for the whole dataset, which are updated by:

\[
moving\_mean = moving\_mean \ast momentum + data\_mean \ast (1 - momentum)\]
\[
moving\_var = moving\_var \ast momentum + data\_var \ast (1 - momentum)\]

If “use\_global\_stats” is set to be true, then “moving\_mean” and “moving\_var” are used instead of “data\_mean” and “data\_var” to compute the output. It is often used during inference.

Both “gamma” and “beta” are learnable parameters. But if “fix\_gamma” is true, then set “gamma” to 1 and its gradient to 0.

There’s no sparse support for this operator, and it will exhibit problematic behavior if used with sparse tensors.

Defined in src/operator/batch_norm_v1.cc:L95

**Value**

out The result mx.ndarray

---

**Description**

Bilinear Sampling is the key of [NIPS2015] "Spatial Transformer Networks". The usage of the operator is very similar to remap function in OpenCV, except that the operator has the backward pass.

**Arguments**

data NDArray-or-Symbol Input data to the BilinearsamplerOp.

grid NDArray-or-Symbol Input grid to the BilinearsamplerOp.grid has two channels: x\_src, y\_src

cudnn\_off boolean or None, optional, default=None whether to turn cudnn off

**Details**

Given :math:`\text{data}` and :math:`\text{grid}`, then the output is computed by

.. math::
   \text{input}[\text{batch}, \text{channel}, \text{y\_src}, \text{x\_src}] = G(\text{data}[\text{batch}, \text{channel}, \text{y\_dst}, \text{x\_dst}])

:math:`\text{x\_src}`, :math:`\text{y\_src}` enumerate all spatial locations in :math:`\text{output}`, and :math:`\text{G}` denotes the bilinear interpolation kernel. The out-boundary points will be padded with zeros. The shape of the output will be *(data.shape[0], data.shape[1], grid.shape[2], grid.shape[3])*. 

The operator assumes that :math:`data` has 'NCHW' layout and :math:`grid` has been normalized to [-1, 1].

BilinearSampler often cooperates with GridGenerator which generates sampling grids for BilinearSampler. GridGenerator supports two kinds of transformation: “affine” and “warp”. If users want to design a CustomOp to manipulate :math:`grid`, please firstly refer to the code of GridGenerator.

Example 1::

```python
## Zoom out data two times
data = array([[[[1, 4, 3, 6], [1, 8, 8, 9], [0, 4, 1, 5], [1, 0, 1, 3]]]])
affine_matrix = array([[2, 0, 0], [0, 2, 0]])
affine_matrix = reshape(affine_matrix, shape=(1, 6))
grid = GridGenerator(data=affine_matrix, transform_type='affine', target_shape=(4, 4))
out = BilinearSampler(data, grid)
out [[[ 0, 0, 0, 0], [ 0, 3.5, 6.5, 0], [ 0, 1.25, 2.5, 0], [ 0, 0, 0, 0]]]
```

Example 2::

```python
## shift data horizontally by -1 pixel
data = array([[[[4, 3, 6, 0], [8, 8, 9, 0], [4, 1, 5, 0], [0, 1, 3, 0]]]])
warp_matrix = array([[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]])
grid = GridGenerator(data=warp_matrix, transform_type='warp')
out = BilinearSampler(data, grid)
out [[[ 4, 3, 6, 0], [8, 8, 9, 0], [4, 1, 5, 0], [0, 1, 3, 0]]]
```

Defined in src/operator/bilinear_sampler.cc:L256

**Value**

`out` The result `mx.ndarray`

---

**mx.nd.BlockGrad**

*stops gradient computation.*

**Description**

Stops the accumulated gradient of the inputs from flowing through this operator in the backward direction. In other words, this operator prevents the contribution of its inputs to be taken into account for computing gradients.

**Arguments**

- `data` NDArray-or-Symbol The input array.
mx.nd.broadcast.add

Details

Example::

v1 = [1, 2] v2 = [0, 1] a = Variable(‘a’) b = Variable(‘b’) b_stop_grad = stop_gradient(3 * b) loss = MakeLoss(b_stop_grad + a)
executor = loss.simple_bind(ctx=cpu(), a=(1,2), b=(1,2)) executor.forward(is_train=True, a=v1, b=v2)
executor.outputs [ 1. 5.]
executor.backward() executor.grad_arrays [ 0. 0.] [ 1. 1.]
Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L325

Value

out The result mx.ndarray

mx.nd.broadcast.add

Returns element-wise sum of the input arrays with broadcasting.

Description

‘broadcast_plus’ is an alias to the function ‘broadcast_add’.

Arguments

lhs NDArray-or-Symbol First input to the function
rhs NDArray-or-Symbol Second input to the function

Details

Example::

x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_add(x, y) = [[ 1., 1., 1.], [ 2., 2., 2.]]
broadcast_plus(x, y) = [[ 1., 1., 1.], [ 2., 2., 2.]]
Supported sparse operations:
broadcast_add(csr, dense(1D)) = dense broadcast_add(dense(1D), csr) = dense
Defined in src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L58

Value

out The result mx.ndarray
mx.nd.broadcast.axis

Broadcasts the input array over particular axes.

Description

Broadcasting is allowed on axes with size 1, such as from `(2,1,3,1)` to `(2,8,3,9)`. Elements will be duplicated on the broadcasted axes.

Arguments

- **data**: NDArray-or-Symbol The input
- **axis**: Shape(tuple), optional, default=[] The axes to perform the broadcasting.
- **size**: Shape(tuple), optional, default=[] Target sizes of the broadcasting axes.

Details

'broadcast_axes' is an alias to the function 'broadcast_axis'.

Example:

```python
// given x of shape (1,2,1) x = [[[ 1.], [ 2.]]]
// broadcast x on on axis 2 broadcast_axis(x, axis=2, size=3) = [[[ 1., 1., 1.], [ 2., 2., 2.]]] // broadcast x on on axes 0 and 2 broadcast_axis(x, axis=(0,2), size=(2,3)) = [[[ 1., 1., 1.], [ 2., 2., 2.]], [[ 1., 1., 1.], [ 2., 2., 2.]]]
```

Defined in src/operator/tensor/broadcast_reduce_op_value.cc:L93

Value

out The result mx.ndarray
mx.nd.broadcast.div

Details

‘broadcast_axes’ is an alias to the function ‘broadcast_axis’.  

Example::

// given x of shape (1,2,1) x = [[[ 1.], [ 2.]]]
// broadcast x on axis 2 broadcast_axis(x, axis=2, size=3) = [[[ 1. 1. 1.], [ 2. 2. 2.]]] // broadcast
// x on on axes 0 and 2 broadcast_axis(x, axis=0,2), size=(2,3)) = [[[ 1. 1.], [ 2. 2.]], [[ 1. 1.], [ 2. 2.]]]  

Defined in src/operator/tensor/broadcast_reduce_op_value.cc:L93

Value

out The result mx.ndarray

mx.nd.broadcast.div

Returns element-wise division of the input arrays with broadcasting.

Description

Example::

Arguments

lhs NDArray-or-Symbol First input to the function

rhs NDArray-or-Symbol Second input to the function

Details

x = [[ 6., 6., 6.], [ 6. 6. 6.]]
y = [[ 2.], [ 3.]]
broadcast_div(x, y) = [[ 3., 3., 3.], [ 2. 2. 2.]]

Supported sparse operations:
broadcast_div(csr, dense(1D)) = csr  

Defined in src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L187

Value

out The result mx.ndarray
mx.nd.broadcast.equal

*Returns the result of element-wise \*==\* (==) comparison operation with broadcasting.*

**Description**

Example::

**Arguments**

<table>
<thead>
<tr>
<th>lhs</th>
<th>NDArray-or-Symbol First input to the function</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs</td>
<td>NDArray-or-Symbol Second input to the function</td>
</tr>
</tbody>
</table>

**Details**

\[
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
\]

\[
y = [[ 0.], [ 1.]]
\]

broadcast_equal(x, y) = [[ 0., 0., 0.], [ 1., 1., 1.]]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L46

**Value**

out The result mx.ndarray

mx.nd.broadcast.greater

*Returns the result of element-wise \*>\* (>) comparison operation with broadcasting.*

**Description**

Example::

**Arguments**

<table>
<thead>
<tr>
<th>lhs</th>
<th>NDArray-or-Symbol First input to the function</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs</td>
<td>NDArray-or-Symbol Second input to the function</td>
</tr>
</tbody>
</table>

**Details**

\[
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
\]

\[
y = [[ 0.], [ 1.]]
\]

broadcast_greater(x, y) = [[ 1., 1., 1.], [ 0., 0., 0.]]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L82
mx.nd.broadcast.greater.equal

Returns the result of element-wise **greater than or equal to** (>=) comparison operation with broadcasting.

Value

out The result mx.ndarray

Arguments

lhs NDArray-or-Symbol First input to the function

rhs NDArray-or-Symbol Second input to the function

Details

x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]

broadcast_greater_equal(x, y) = [[ 1., 1., 1.], [ 1., 1., 1.]]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L100

mx.nd.broadcast.hypot

Returns the hypotenuse of a right angled triangle, given its "legs" with broadcasting.

Description

It is equivalent to doing :math:`\sqrt{x_1^2 + x_2^2}`.

Arguments

lhs NDArray-or-Symbol First input to the function

rhs NDArray-or-Symbol Second input to the function
mx.nd.broadcast.lesser

Details

Example::

x = [[ 3., 3., 3.]]
y = [[ 4.], [ 4.]]
broadcast_hypot(x, y) = [[ 5., 5., 5.], [ 5., 5., 5.]]
z = [[ 0.], [ 4.]]
broadcast_hypot(x, z) = [[ 3., 3., 3.], [ 5., 5., 5.]]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_extended.cc:L158

Value

out The result mx.ndarray

mx.nd.broadcast.lesser

Returns the result of element-wise **lesser than** (<) comparison operation with broadcasting.

Description

Example::

Arguments

lhs NDArray-or-Symbol First input to the function
rhs NDArray-or-Symbol Second input to the function

Details

x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_lesser(x, y) = [[ 0., 0., 0.], [ 0., 0., 0.]]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L118

Value

out The result mx.ndarray
mx.nd.broadcast.lesser.equal

Returns the result of element-wise **less than or equal to** (<=) comparison operation with broadcasting.

**Description**

Example::

**Arguments**

- **lhs**: NDArray-or-Symbol First input to the function
- **rhs**: NDArray-or-Symbol Second input to the function

**Details**

\[ x = [[1., 1., 1.], [1., 1., 1.]] \]
\[ y = [[0.], [1.]] \]
\[ \text{broadcast	extunderscore lesser	extunderscore equal}(x, y) = [[0., 0., 0.], [1., 1., 1.]] \]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L136

**Value**

- **out**: The result mx.ndarray

---

mx.nd.broadcast.like

Broadcasts lhs to have the same shape as rhs.

**Description**

Broadcasting is a mechanism that allows NDArrays to perform arithmetic operations with arrays of different shapes efficiently without creating multiple copies of arrays. Also see, `Broadcasting <https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>`_ for more explanation.

**Arguments**

- **lhs**: NDArray-or-Symbol First input.
- **rhs**: NDArray-or-Symbol Second input.
- **lhs.\ axes**: Shape or None, optional, default=None Axes to perform broadcast on in the first input array
- **rhs.\ axes**: Shape or None, optional, default=None Axes to copy from the second input array
mx.nd.broadcast.logical.and

Details

Broadcasting is allowed on axes with size 1, such as from `(2,1,3,1)` to `(2,8,3,9)`. Elements will be duplicated on the broadcasted axes.

For example:

\[
\text{broadcast_like}([[1,2,3]], [[5,6,7],[7,8,9]]) = [[1., 2., 3.], [1., 2., 3.]]
\]

\[
\text{broadcast_like}(9, [1,2,3,4,5], \text{lhs_axes}=(0,), \text{rhs_axes}=(-1,)) = [9,9,9,9,9]
\]

Defined in src/operator/tensor/broadcast_reduce_op_value.cc:L171

Value

out The result mx.nd.array
**mx.nd.broadcast.logical.or**

Returns the result of element-wise **logical or** with broadcasting.

**Description**

Example::

**Arguments**

- `lhs` NDArray-or-Symbol First input to the function
- `rhs` NDArray-or-Symbol Second input to the function

**Details**

\[
x = \begin{bmatrix}
1., 1., 0. \\
1., 1., 0.
\end{bmatrix}
\]

\[
y = \begin{bmatrix}
1. \\
0.
\end{bmatrix}
\]

broadcast_logical_or(x, y) = \[
\begin{bmatrix}
1., 1., 1. \\
1., 1., 0.
\end{bmatrix}
\]

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_logic.cc`:L172

**Value**

- `out` The result mx.ndarray

**mx.nd.broadcast.logical.xor**

Returns the result of element-wise **logical xor** with broadcasting.

**Description**

Example::

**Arguments**

- `lhs` NDArray-or-Symbol First input to the function
- `rhs` NDArray-or-Symbol Second input to the function

**Details**

\[
x = \begin{bmatrix}
1., 1., 0. \\
1., 1., 0.
\end{bmatrix}
\]

\[
y = \begin{bmatrix}
1. \\
0.
\end{bmatrix}
\]

broadcast_logical_xor(x, y) = \[
\begin{bmatrix}
0., 0., 1. \\
1., 1., 0.
\end{bmatrix}
\]

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_logic.cc`:L190
mx.nd.broadcast.minimum

Returns element-wise minimum of the input arrays with broadcasting.

Description

This function compares two input arrays and returns a new array having the element-wise minima.

Arguments

- **lhs**: NDArray-or-Symbol First input to the function
- **rhs**: NDArray-or-Symbol Second input to the function

Details

Example::

x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_minimum(x, y) = [[ 1., 1., 1.], [ 1., 1., 1.]]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_extended.cc:L81

mx.nd.broadcast.minimum

Returns element-wise minimum of the input arrays with broadcasting.

Description

This function compares two input arrays and returns a new array having the element-wise minima.

Arguments

- **lhs**: NDArray-or-Symbol First input to the function
- **rhs**: NDArray-or-Symbol Second input to the function
mx.nd.broadcast.minus

Details

Example::
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_maximum(x, y) = [[ 0., 0., 0.], [ 1., 1., 1.]]
Defined in src/operator/tensor/elewise_binary_broadcast_op_extended.cc:L117

Value

out The result mx.ndarray

mx.nd.broadcast.minus  Returns element-wise difference of the input arrays with broadcasting.

Description

‘broadcast_minus’ is an alias to the function ‘broadcast_sub’.

Arguments

lhs NDArray-or-Symbol First input to the function  
rhs NDArray-or-Symbol Second input to the function

Details

Example::
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_sub(x, y) = [[ 1., 1., 1.], [ 0., 0., 0.]]
broadcast_minus(x, y) = [[ 1., 1., 1.], [ 0., 0., 0.]]
Supported sparse operations:
broadcast_sub/minus(csr, dense(1D)) = dense broadcast_sub/minus(dense(1D), csr) = dense
Defined in src/operator/tensor/elewise_binary_broadcast_op_basic.cc:L106

Value

out The result mx.ndarray
**mx.nd.broadcast.mod**

*Returns element-wise modulo of the input arrays with broadcasting.*

**Description**

Example::

**Arguments**

- **lhs**: NDArray-or-Symbol First input to the function
- **rhs**: NDArray-or-Symbol Second input to the function

**Details**

\[x = [[8., 8., 8.], [8., 8., 8.]]\]
\[y = [[2.], [3.]]\]
\[broadcast\_mod(x, y) = [[0., 0., 0.], [2., 2., 2.]]\]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L222

**Value**

- **out**: The result mx.ndarray

---

**mx.nd.broadcast.mul**

*Returns element-wise product of the input arrays with broadcasting.*

**Description**

Example::

**Arguments**

- **lhs**: NDArray-or-Symbol First input to the function
- **rhs**: NDArray-or-Symbol Second input to the function

**Details**

\[x = [[1., 1., 1.], [1., 1., 1.]]\]
\[y = [[0.], [1.]]\]
\[broadcast\_mul(x, y) = [[0., 0., 0.], [1., 1., 1.]]\]

Supported sparse operations:
\[broadcast\_mul(csr, dense(1D)) = csr\]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L146
mx.nd.broadcast.not.equal

Returns the result of element-wise not equal to (=) comparison operation with broadcasting.

Description

Example::

Arguments

(lhs) NDArray-or-Symbol First input to the function
(rhs) NDArray-or-Symbol Second input to the function

Details

x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_not_equal(x, y) = [[ 1., 1., 1.], [ 0., 0., 0.]]
Defined in src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L64

mx.nd.broadcast.plus

Returns element-wise sum of the input arrays with broadcasting.

Description

'broadcast_plus' is an alias to the function 'broadcast_add'.

Arguments

(lhs) NDArray-or-Symbol First input to the function
(rhs) NDArray-or-Symbol Second input to the function
Details

Example::

```python
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_add(x, y) = [[ 1., 1., 1.], [ 2., 2., 2.]]
broadcast_plus(x, y) = [[ 1., 1., 1.], [ 2., 2., 2.]]
```

Supported sparse operations:

```python
broadcast_add(csr, dense(1D)) = dense broadcast_add(dense(1D), csr) = dense
```

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L58`

Value

```python
out The result mx.ndarray
```

```
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_power(x, y) = [[ 2., 2., 2.], [ 4., 4., 4.]]
```

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_extended.cc:L45`

Value

```python
out The result mx.ndarray
```
mx.nd.broadcast.sub

Returns element-wise difference of the input arrays with broadcasting.

Description

`broadcast_minus` is an alias to the function `broadcast_sub`.

Arguments

- **lhs**: NDArray-or-Symbol First input to the function
- **rhs**: NDArray-or-Symbol Second input to the function

Details

Example:
```
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_sub(x, y) = [[ 1., 1., 1.], [ 0., 0., 0.]]
broadcast_minus(x, y) = [[ 1., 1., 1.], [ 0., 0., 0.]]
```

Supported sparse operations:
```
broadcast_sub/minus(csr, dense(1D)) = dense broadcast_sub/minus(dense(1D), csr) = dense
```

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L106`

Value

- **out**: The result mx.ndarray

mx.nd.broadcast.to

Broadcasts the input array to a new shape.

Description

Broadcasting is a mechanism that allows NDArrays to perform arithmetic operations with arrays of different shapes efficiently without creating multiple copies of arrays. Also see, ‘Broadcasting <https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>‘ for more explanation.

Arguments

- **data**: NDArray-or-Symbol The input
- **shape**: Shape(tuple), optional, default=[] The shape of the desired array. We can set the dim to zero if it’s same as the original. E.g 'A = broadcast_to(B, shape=(10, 0, 0))’ has the same meaning as ‘A = broadcast_axis(B, axis=0, size=10)’.
Details

Broadcasting is allowed on axes with size 1, such as from '(2,1,3,1)' to '(2,8,3,9)'. Elements will be duplicated on the broadcasted axes.

For example::

    broadcast_to([[1,2,3]], shape=(2,3)) = [[ 1., 2., 3.], [ 1., 2., 3.]]

The dimension which you do not want to change can also be kept as ‘0’ which means copy the original value. So with ‘shape=(2,0)’ we will obtain the same result as in the above example.

Defined in src/operator/tensor/broadcast_reduce_op_value.cc:L117

Value

    out The result mx.ndarray

mx.nd.Cast

Casts all elements of the input to a new type.

Description

.. note:: "Cast" is deprecated. Use "cast" instead.

Arguments

    data NDArray-or-Symbol The input.

    dtype 'bfloat16', 'bool', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', required Output data type.

Details

Example::

    cast([0.9, 1.3], dtype='int32') = [0, 1] cast([1e20, 11.1], dtype='float16') = [inf, 11.09375] cast([300, 11.1, 10.9, -1, -3], dtype='uint8') = [44, 11, 10, 255, 253]

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L664

Value

    out The result mx.ndarray
**mx.nd.cast**

**Casts all elements of the input to a new type.**

**Description**

.. note:: “Cast” is deprecated. Use “cast” instead.

**Arguments**

- **data**
  - NDArray-or-Symbol
  - The input.

- **dtype**
  - 'bfloat16', 'bool', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', required Output data type.

**Details**

Example::

   cast([[0.9, 1.3], dtype='int32')] = [0, 1]  cast([[1e20, 11.1], dtype='float16')] = [inf, 11.09375]  cast([[300, 11.1, 10.9, -1, -3], dtype='uint8')] = [44, 11, 10, 255, 253]

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L664

**Value**

- **out**
  - The result mx.ndarray

---

**mx.nd.cast.storage**

**Casts tensor storage type to the new type.**

**Description**

When an NDArray with default storage type is cast to csr or row_sparse storage, the result is compact, which means:

**Arguments**

- **data**
  - NDArray-or-Symbol
  - The input.

- **stype**
  - 'csr', 'default', 'row_sparse', required Output storage type.
**mx.nd.cbrt**

Returns element-wise cube-root value of the input.

**Description**

.. math:: cbrt(x) = \sqrt[3]{x}

**Arguments**

- **data**: NDArray-or-Symbol The input array.

**Details**

Example:

cbrt([1, 8, -125]) = [1, 2, -5]

The storage type of “cbrt” output depends upon the input storage type:

- cbrt(default) = default - cbrt(row_sparse) = row_sparse - cbrt(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_pow.cc:L270

**Value**

- **out**: The result mx.nd.array
mx.nd.ceil

Returns element-wise ceiling of the input.

**Description**

The ceil of the scalar \( x \) is the smallest integer \( i \), such that \( i \geq x \).

**Arguments**

- **data** NDArray-or-Symbol The input array.

**Details**

Example:

\[
\text{ceil([-2.1, -1.9, 1.5, 1.9, 2.1]}) = [-2., -1., 2., 2., 3.]
\]

The storage type of "ceil" output depends upon the input storage type:

- ceil(default) = default - ceil(row_sparse) = row_sparse - ceil(csr) = csr

Defined in src/operator/tensor/elemwise Unary_op_basic.cc:L817

**Value**

- **out** The result mx.ndarray

mx.nd.choose.element

Picks elements from an input array according to the input indices along the given axis.

**Description**

Given an input array of shape "(d0, d1)" and indices of shape "(i0,)", the result will be an output array of shape "(i0,)" with:

**Arguments**

- **data** NDArray-or-Symbol The input array
- **index** NDArray-or-Symbol The index array
- **axis** int or None, optional, default='-1' int or None. The axis to picking the elements. Negative values means indexing from right to left. If is 'None', the elements in the index w.r.t the flattened input will be picked.
- **keepdims** boolean, optional, default=0 If true, the axis where we pick the elements is left in the result as dimension with size one.
- **mode** 'clip', 'wrap',optional, default='clip' Specify how out-of-bound indices behave. Default is "clip". "clip" means clip to the range. So, if all indices mentioned are too large, they are replaced by the index that addresses the last element along an axis. "wrap" means to wrap around.
Details

\[
\text{output}[i] = \text{input}[i, \text{indices}[i]]
\]

By default, if any index mentioned is too large, it is replaced by the index that addresses the last element along an axis (the ‘clip’ mode).

This function supports n-dimensional input and (n-1)-dimensional indices arrays.

Examples::

\[
x = [[1., 2.], [3., 4.], [5., 6.]]
\]

// picks elements with specified indices along axis 0
pick(x, y=[0,1], 0) = [1., 4.]

// picks elements with specified indices along axis 1
pick(x, y=[0,1,0], 1) = [1., 4., 5.]

// picks elements with specified indices along axis 1 using 'wrap' mode // to place indicies that would normally be out of bounds
pick(x, y=[2,-1,-2], 1, mode='wrap') = [1., 4., 5.]

\[
y = [[1.], [0.], [2.]]
\]

// picks elements with specified indices along axis 1 and dims are maintained
pick(x, y, 1, keepdims=True) = [[2.], [3.], [6.]]

Defined in src/operator/tensor/broadcast_reduce_op_index.cc:L151

Value

out The result mx.ndarray

mx.nd.clip

Clips (limits) the values in an array. Given an interval, values outside the interval are clipped to the interval edges. Clipping “x” between ‘a_min’ and ‘a_max’ would be:

\[
\max(\min(x, a_{max}), a_{min})
\]

Example::

\[
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
\]

clip(x, 1, 8) = [1., 1., 2., 3., 4., 5., 6., 7., 8., 8.]

The storage type of “clip” output depends on storage types of inputs and the a_min, a_max \( \text{parameter values:} \)

- clip(default) = default - clip(row_sparse, a_min <= 0, a_max >= 0) = row_sparse - clip(csr, a_min <= 0, a_max >= 0) = csr

Description

Defined in src/operator/tensor/matrix_op.cc:L677

Arguments

- data NDArray-or-Symbol Input array.
- a.min float, required Minimum value
- a.max float, required Maximum value
Value

out The result mx.ndarray

mx.nd.col2im

Combining the output column matrix of im2col back to image array.

Description

Like :class:`~mxnet.ndarray.im2col`, this operator is also used in the vanilla convolution implementation. Despite the name, col2im is not the reverse operation of im2col. Since there may be overlaps between neighbouring sliding blocks, the column elements cannot be directly put back into image. Instead, they are accumulated (i.e., summed) in the input image just like the gradient computation, so col2im is the gradient of im2col and vice versa.

Arguments

data NDArray-or-Symbol Input array to combine sliding blocks.

output.size Shape(tuple), required The spatial dimension of image array: (w,), (h, w) or (d, h, w).

kernel Shape(tuple), required Sliding kernel size: (w,), (h, w) or (d, h, w).

stride Shape(tuple), optional, default= [] The stride between adjacent sliding blocks in spatial dimension: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.

dilate Shape(tuple), optional, default= [] The spacing between adjacent kernel points: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.

pad Shape(tuple), optional, default= [] The zero-value padding size on both sides of spatial dimension: (w,), (h, w) or (d, h, w). Defaults to no padding.

Details

Using the notation in im2col, given an input column array of shape :math:`(N, C \times \prod(k), W)`; this operator accumulates the column elements into output array of shape :math:`(N, C, \text{output_size}[0], \text{output_size}[1], \ldots)`. Only 1-D, 2-D and 3-D of spatial dimension is supported in this operator.

Defined in src/operator/nn/im2col.cc:L182

Value

out The result mx.ndarray
mx.nd.Concat

**Joins input arrays along a given axis.**

**Description**

.. note:: ‘Concat’ is deprecated. Use ‘concat’ instead.

**Arguments**

- **data**: NDArray-or-Symbol[] List of arrays to concatenate
- **num.args**: int, required Number of inputs to be concated.
- **dim**: int, optional, default=’1’ the dimension to be concated.

**Details**

The dimensions of the input arrays should be the same except the axis along which they will be concatenated. The dimension of the output array along the concatenated axis will be equal to the sum of the corresponding dimensions of the input arrays.

The storage type of “concat” output depends on storage types of inputs
- **concat(csr, csr, ..., csr, dim=0) = csr** - otherwise, “concat” generates output with default storage

**Example::**

```
x = [[1,1],[2,2]] y = [[3,3],[4,4],[5,5]] z = [[6,6],[7,7],[8,8]]
concat(x,y,z,dim=0) = [[ 1., 1.], [ 2., 2.], [ 3., 3.], [ 4., 4.], [ 5., 5.], [ 6., 6.], [ 7., 7.], [ 8., 8.]]
```

Note that you cannot concat x,y,z along dimension 1 since dimension 0 is not the same for all the input arrays.

```
concat(y,z,dim=1) = [[ 3., 3., 6., 6.], [ 4., 4., 7., 7.], [ 5., 5., 8., 8.]]
```

Defined in src/operator/nn/concat.cc:L385

**Value**

- **out**: The result mx.ndarray
mx.nd.Convolution

Arguments

- **data**: NDArray-or-Symbol[] List of arrays to concatenate
- **num. args**: int, required Number of inputs to be concated.
- **dim**: int, optional, default='1' the dimension to be concated.

Details

The dimensions of the input arrays should be the same except the axis along which they will be concatenated. The dimension of the output array along the concatenated axis will be equal to the sum of the corresponding dimensions of the input arrays.

The storage type of “concat” output depends on storage types of inputs

- `concat(csr, csr, ..., csr, dim=0) = csr` - otherwise, “concat” generates output with default storage

Example:

```python
x = [[1,1],[2,2]]  
y = [[3,3],[4,4]]  
z = [[6,6],[7,7],[8,8]]

concat(x,y,z,dim=0) = [[1.,1.],[2.,2.],[3.,3.],[4.,4.],[5.,5.],[6.,6.],[7.,7.],[8.,8.]]
```

Note that you cannot concat x,y,z along dimension 1 since dimension 0 is not the same for all the input arrays.

```python
concat(y,z,dim=1) = [[3.,3.,6.,6.],[4.,4.,7.,7.],[5.,5.,8.,8.]]
```

Defined in src/operator/nn/concat.cc:L385

Value

- **out**: The result mx.ndarray

---

mx.nd.Convolution

*Compute *N*-D convolution on *(N+2)*-D input.*

Description

In the 2-D convolution, given input data with shape *(batch_size, channel, height, width)*, the output is computed by

Arguments

- **data**: NDArray-or-Symbol Input data to the ConvolutionOp.
- **weight**: NDArray-or-Symbol Weight matrix.
- **bias**: NDArray-or-Symbol Bias parameter.
- **kernel**: Shape(tuple), required Convolution kernel size: (w), (h, w) or (d, h, w)
- **stride**: Shape(tuple), optional, default=[1] Convolution stride: (w), (h, w) or (d, h, w). Defaults to 1 for each dimension.
- **dilate**: Shape(tuple), optional, default=[1] Convolution dilate: (w), (h, w) or (d, h, w). Defaults to 1 for each dimension.
**pad**
Shape(tuple), optional, default=[] Zero pad for convolution: (w,), (h, w) or (d, h, w). Defaults to no padding.

**num.filter**
int (non-negative), required Convolution filter(channel) number

**num.group**
int (non-negative), optional, default=1 Number of group partitions.

**workspace**
long (non-negative), optional, default=1024 Maximum temporary workspace allowed (MB) in convolution. This parameter has two usages. When CUDNN is not used, it determines the effective batch size of the convolution kernel. When CUDNN is used, it controls the maximum temporary storage used for tuning the best CUDNN kernel when ‘limited_workspace’ strategy is used.

**no.bias**
boolean, optional, default=0 Whether to disable bias parameter.

**cudnn.tune**

**cudnn.off**
boolean, optional, default=0 Turn off cudnn for this layer.

**layout**
None, ‘NCDHW’, ‘NCHW’, ‘NCW’, ‘NDHWC’, ‘NHWC’, optional, default=’None’ Set layout for input, output and weight. Empty for default layout: NCW for 1d, NCHW for 2d and NCDHW for 3d. NHWC and NDHWC are only supported on GPU.

**Details**

.. math::
   \text{out}[n,i,:,:] = \text{bias}[i] + \sum_j=0^\text{channel} \text{data}[n,j,:,:] \star \text{weight}[i,j,:,:]

where :math:`\star` is the 2-D cross-correlation operator.

For general 2-D convolution, the shapes are

- **data**: *(batch_size, channel, height, width)*
- **weight**: *(num_filter, channel, kernel[0], kernel[1])*  
- **bias**: *(num_filter,)*
- **out**: *(batch_size, num_filter, out_height, out_width)*.

Define::

\[ f(x,k,p,s,d) = \text{floor}((x+2*p-d*(k-1)-1)/s)+1 \]

then we have::

\[ \text{out}_{\text{height}}=f(\text{height, kernel}[0], \text{pad}[0], \text{stride}[0], \text{dilate}[0]) \]
\[ \text{out}_{\text{width}}=f(\text{width, kernel}[1], \text{pad}[1], \text{stride}[1], \text{dilate}[1]) \]

If “no.bias” is set to be true, then the “bias” term is ignored.

The default data “layout” is *NCHW*, namely *(batch_size, channel, height, width)*. We can choose other layouts such as *NWC*.

If “num_group” is larger than 1, denoted by *g*, then split the input “data” evenly into *g* parts along the channel axis, and also evenly split “weight” along the first dimension. Next compute the convolution on the *i*-th part of the data with the *i*-th weight part. The output is obtained by concatenating all the *g* results.

1-D convolution does not have *height* dimension but only *width* in space.

- **data**: *(batch_size, channel, width)*
- **weight**: *(num_filter, channel, kernel[0])*  
- **bias**: *(num_filter,)*
- **out**: *(batch_size, num_filter, out_width)*.
3-D convolution adds an additional *depth* dimension besides *height* and *width*. The shapes are:
- **data**: *(batch_size, channel, depth, height, width)*
- **weight**: *(num_filter, channel, kernel[0], kernel[1], kernel[2])*
- **bias**: *(num_filter,)*
- **out**: *(batch_size, num_filter, out_depth, out_height, out_width)*.

Both “weight” and “bias” are learnable parameters.

There are other options to tune the performance.

- **cudnn_tune**: enable this option leads to higher startup time but may give faster speed. Options are:
  - **off**: no tuning
  - **limited_workspace**: run test and pick the fastest algorithm that doesn’t exceed workspace limit.
  - **fastest**: pick the fastest algorithm and ignore workspace limit.
  - **None** (default): the behavior is determined by environment variable “MXNET_CUDNN_AUTOTUNE_DEFAULT“. 0 for off, 1 for limited workspace (default), 2 for fastest.
- **workspace**: A large number leads to more (GPU) memory usage but may improve the performance.

Defined in src/operator/nn/convolution.cc:L469

Value

out The result mx.ndarray

mx.nd.Convolution.v1

This operator is DEPRECATED. Apply convolution to input then add a bias.

Description

This operator is DEPRECATED. Apply convolution to input then add a bias.

Arguments

data NDArray-or-Symbol Input data to the ConvolutionV1Op.
weight NDArray-or-Symbol Weight matrix.
bias NDArray-or-Symbol Bias parameter.
kernel Shape(tuple), required convolution kernel size: (h, w) or (d, h, w)
stride Shape(tuple), optional, default=[] convolution stride: (h, w) or (d, h, w)
dilate Shape(tuple), optional, default=[] convolution dilate: (h, w) or (d, h, w)
pad Shape(tuple), optional, default=[] pad for convolution: (h, w) or (d, h, w)
num.filter int (non-negative), required convolution filter(channel) number
num.group int (non-negative), optional, default=1 Number of group partitions. Equivalent to slicing input into num_group partitions, apply convolution on each, then concatenate the results
**workspace**

long (non-negative), optional, default=1024 Maximum temporary workspace allowed for convolution (MB). This parameter determines the effective batch size of the convolution kernel, which may be smaller than the given batch size. Also, the workspace will be automatically enlarged to make sure that we can run the kernel with batch_size=1.

**no.bias**

boolean, optional, default=0 Whether to disable bias parameter.

**cudnn.tune**

None, 'fastest', 'limited_workspace', 'off', optional, default='None' Whether to pick convolution algo by running performance test. Leads to higher startup time but may give faster speed. Options are: 'off': no tuning 'limited_workspace': run test and pick the fastest algorithm that doesn’t exceed workspace limit. 'fastest': pick the fastest algorithm and ignore workspace limit. If set to None (default), behavior is determined by environment variable MXNET_CUDNN_AUTOTUNE_DEFAULT: 0 for off, 1 for limited workspace (default), 2 for fastest.

**cudnn.off**

boolean, optional, default=0 Turn off cudnn for this layer.

**layout**

None, 'NCDHW', 'NCHW', 'NDHWC', 'NHWC', optional, default='None' Set layout for input, output and weight. Empty for default layout: NCHW for 2d and NCDHW for 3d.

---

**mx.nd.copyto**

Generate an mx.ndarray object on ctx, with data copied from src

**Description**

Generate an mx.ndarray object on ctx, with data copied from src

**Usage**

mx.nd.copyto(src, ctx)

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>The source mx.ndarray object.</td>
</tr>
<tr>
<td>ctx</td>
<td>The target context.</td>
</tr>
</tbody>
</table>
The correlation layer performs multiplicative patch comparisons between two feature maps.

**Arguments**

- **data1**
  NDArray-or-Symbol Input data1 to the correlation.

- **data2**
  NDArray-or-Symbol Input data2 to the correlation.

- **kernel.size**
  int (non-negative), optional, default=1 kernel size for Correlation must be an odd number

- **max.displacement**
  int (non-negative), optional, default=1 Max displacement of Correlation

- **stride1**
  int (non-negative), optional, default=1 stride1 quantize data1 globally

- **stride2**
  int (non-negative), optional, default=1 stride2 quantize data2 within the neighborhood centered around data1

- **pad.size**
  int (non-negative), optional, default=0 pad for Correlation

- **is.multiply**
  boolean, optional, default=1 operation type is either multiplication or subduction

**Details**

Given two multi-channel feature maps :math:`f_1, f_2`, with :math:`w`, :math:`h`, and :math:`c` being their width, height, and number of channels, the correlation layer lets the network compare each patch from :math:`f_1` with each patch from :math:`f_2`.

For now we consider only a single comparison of two patches. The ‘correlation’ of two patches centered at :math:`x_1` in the first map and :math:`x_2` in the second map is then defined as:

.. math::
   c(x_1, x_2) = \sum_o \in [-k,k] \times [-k,k] \langle f_1(x_1 + o), f_2(x_2 + o) \rangle

for a square patch of size :math:`K:=2k+1`.

Note that the equation above is identical to one step of a convolution in neural networks, but instead of convolving data with a filter, it convolves data with other data. For this reason, it has no training weights.

Computing :math:`c(x_1, x_2)` involves :math:`c \times K^2` multiplications. Comparing all patch combinations involves :math:`w^2 \times h^2 \times K^2` such computations.

Given a maximum displacement :math:`D`, for each location :math:`x_1` it computes correlations :math:`c(x_1, x_2)` only in a neighborhood of size :math:`D:=2d+1`, by limiting the range of :math:`x_2`. We use strides :math:`s_1, s_2`, to quantize :math:`x_1` globally and to quantize :math:`x_2` within the neighborhood centered around :math:`x_1`.

The final output is defined by the following expression:

.. math::
   out[n, q, i, j] = c(x_i, j, x_q)
where \(i\) and \(j\) enumerate spatial locations in \(f_1\), and \(q\) denotes the \(q^{th}\) neighborhood of \(x_{i,j}\).
Defined in src/operator/correlation.cc:L198

Value

out The result mx.ndarray

mx.nd.cos

*Computes the element-wise cosine of the input array.*

**Description**

The input should be in radians (2\(\pi\) rad equals 360 degrees).

**Arguments**

data NDArray-or-Symbol The input array.

**Details**

\[
\cos([0, \pi/4, \pi/2]) = [1, 0.707, 0]
\]
The storage type of “cos” output is always dense
Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L90

Value

out The result mx.ndarray

mx.nd.cosh

*Returns the hyperbolic cosine of the input array, computed element-wise.*

**Description**

\[
\cosh(x) = 0.5 \times (\exp(x) + \exp(-x))
\]

**Arguments**

data NDArray-or-Symbol The input array.

**Details**

The storage type of “cosh” output is always dense
Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L409
mx.nd.Crop

**Value**

out The result mx.ndarray

---

**Description**

Crop the 2nd and 3rd dim of input data, with the corresponding size of h\_w or with width and height of the second input symbol, i.e., with one input, we need h\_w to specify the crop height and width, otherwise the second input symbol’s size will be used.

**Arguments**

- **data** Symbol or Symbol[] Tensor or List of Tensors, the second input will be used as crop\_like shape reference.
- **num.args** int, required Number of inputs for crop, if equals one, then we will use the h\_w for crop height and width, else if equals two, then we will use the height and width of the second input symbol, we name crop\_like here.
- **offset** Shape(tuple), optional, default=[0,0] crop offset coordinate: (y, x)
- **h\_w** Shape(tuple), optional, default=[0,0] crop height and width: (h, w)
- **center.crop** boolean, optional, default=0 If set to true, then it will use be the center\_crop, or it will crop using the shape of crop\_like

**Details**

Defined in src/operator/crop.cc:L50

**Value**

out The result mx.ndarray
mx.nd.crop

Slices a region of the array. .. note:: “crop” is deprecated. Use “slice” instead. This function returns a sliced array between the indices given by ‘begin’ and ‘end’ with the corresponding ‘step’. For an input array of “shape=(d_0, d_1, ..., d_n-1)“, slice operation with “begin=(b_0, b_1...b_m-1)“, “end=(e_0, e_1, ..., e_m-1)“, and “step=(s_0, s_1, ..., s_m-1)“, where m <= n, results in an array with the shape “((e_0-b_0)/s_0, ..., (e_m-1-b_m-1)/s_m-1), d_m, ..., d_n-1)“. The resulting array’s *k*-th dimension contains elements from the *k*-th dimension of the input array starting from index “b_k” (inclusive) with step “s_k” until reaching “e_k” (exclusive). If the *k*-th elements are ‘None’ in the sequence of ‘begin’, ‘end’, and ‘step’, the following rule will be used to set default values. If ‘s_k’ is ‘None’, set ‘s_k=1’. If ‘s_k > 0’, set ‘b_k=0’, ‘e_k=d_k’; else, set ‘b_k=d_k-1’, ‘e_k=-1’. The storage type of “slice” output depends on storage types of inputs - slice(csr) = csr - otherwise, “slice” generates output with default storage .. note:: When input data storage type is csr, it only supports step=(), or step=(None,), or step=(1,) to generate a csr output. For other step parameter values, it falls back to slicing a dense tensor. Example:: x = [[ 1., 2., 3., 4.], [ 5., 6., 7., 8.], [ 9., 10., 11., 12.]] slice(x, begin=(0,1), end=(2,4)) = [[ 2., 3., 4.], [ 6., 7., 8.]] slice(x, begin=(None, 0), end=(None, 3), step=(-1, 2)) = [[9., 11.], [5., 7.], [1., 3.]]

Description

Defined in src/operator/tensor/matrix_op.cc:L482

Arguments

data NDArray-or-Symbol Source input

begin Shape(tuple), required starting indices for the slice operation, supports negative indices.

end Shape(tuple), required ending indices for the slice operation, supports negative indices.

step Shape(tuple), optional, default=[] step for the slice operation, supports negative values.

Value

out The result mx.ndarray
mx.nd.ctc.loss

Connectionist Temporal Classification Loss.

Description

.. note:: The existing alias "contrib_CTCLoss" is deprecated.

Arguments

data : NDArray-or-Symbol Input ndarray

label : NDArray-or-Symbol Ground-truth labels for the loss.

data.lengths : NDArray-or-Symbol Lengths of data for each of the samples. Only required when use_data_lengths is true.

label.lengths : NDArray-or-Symbol Lengths of labels for each of the samples. Only required when use_label_lengths is true.

use.data.lengths : boolean, optional, default=0 Whether the data lengths are decided by 'data_lengths'. If false, the lengths are equal to the max sequence length.

use.label.lengths : boolean, optional, default=0 Whether the label lengths are decided by 'label_lengths', or derived from 'padding_mask'. If false, the lengths are derived from the first occurrence of the value of 'padding_mask'. The value of 'padding_mask' is "0" when first CTC label is reserved for blank, and "-1" when last label is reserved for blank. See 'blank_label'.

blank.label : 'first', 'last', optional, default='first' Set the label that is reserved for blank label. If "first", 0-th label is reserved, and label values for tokens in the vocabulary are between "1" and "alphabet_size-1", and the padding mask is "-1". If "last", last label value "alphabet_size-1" is reserved for blank label instead, and label values for tokens in the vocabulary are between "0" and "alphabet_size-2", and the padding mask is "0".

Details

The shapes of the inputs and outputs:

- **data**: (sequence_length, batch_size, alphabet_size)
- **label**: (batch_size, label_sequence_length)
- **out**: (batch_size)

The 'data' tensor consists of sequences of activation vectors (without applying softmax), with i-th channel in the last dimension corresponding to i-th label for i between 0 and alphabet_size-1 (i.e always 0-indexed). Alphabet size should include one additional value reserved for blank label. When 'blank_label' is ""first"", the "0"-th channel is be reserved for activation of blank label, or otherwise if it is "last", "(alphabet_size-1)"-th channel should be reserved for blank label.

"label" is an index matrix of integers. When 'blank_label' is ""first"", the value 0 is then reserved for blank label, and should not be passed in this matrix. Otherwise, when 'blank_label' is ""last"", the value 'alphabet_size-1)" is reserved for blank label.
mx.nd.CTCLoss

If a sequence of labels is shorter than *label_sequence_length*, use the special padding value at the end of the sequence to conform it to the correct length. The padding value is '0' when 'blank_label' is "first", and '-1' otherwise.

For example, suppose the vocabulary is '[a, b, c]', and in one batch we have three sequences 'ba', 'cbb', and 'abac'. When 'blank_label' is "first", we can index the labels as 'a': 1, 'b': 2, 'c': 3, and we reserve the 0-th channel for blank label in data tensor. The resulting 'label' tensor should be padded to be:

[[2, 1, 0, 0], [3, 2, 2, 0], [1, 2, 1, 3]]

When 'blank_label' is "last", we can index the labels as 'a': 0, 'b': 1, 'c': 2, and we reserve the channel index 3 for blank label in data tensor. The resulting 'label' tensor should be padded to be:

[[1, 0, -1, -1], [2, 1, 1, -1], [0, 1, 0, 2]]

"out" is a list of CTC loss values, one per example in the batch.

See *Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks*, A. Graves et al, for more information on the definition and the algorithm. Defined in src/operator/nn/ctc_loss.cc:L100

Value

out The result mx.ndarray

mx.nd.CTCLoss Connectionist Temporal Classification Loss.

Description

.. note:: The existing alias "contrib_CTCloss" is deprecated.

Arguments

data NDArray-or-Symbol Input ndarray

label NDArray-or-Symbol Ground-truth labels for the loss.

data.lengths NDArray-or-Symbol Lengths of data for each of the samples. Only required when use_data_lengths is true.

label.lengths NDArray-or-Symbol Lengths of labels for each of the samples. Only required when use_label_lengths is true.

use.data.lengths boolean, optional, default=0 Whether the data lengths are decided by 'data_lengths'. If false, the lengths are equal to the max sequence length.

use.label.lengths boolean, optional, default=0 Whether the label lengths are decided by 'label_lengths', or derived from 'padding_mask'. If false, the lengths are derived from the first occurrence of the value of 'padding_mask'. The value of 'padding_mask' is "0" when first CTC label is reserved for blank, and "-1" when last label is reserved for blank. See 'blank_label'.
blank.label  'first', 'last', optional, default='first' Set the label that is reserved for blank label. If "first", 0-th label is reserved, and label values for tokens in the vocabulary are between "1" and "alphabet_size-1", and the padding mask is "-1". If "last", last label value "alphabet_size-1" is reserved for blank label instead, and label values for tokens in the vocabulary are between "0" and "alphabet_size-2", and the padding mask is "0".

Details

The shapes of the inputs and outputs:

- **data**: '(sequence_length, batch_size, alphabet_size)' - **label**: '(batch_size, label_sequence_length)' - **out**: '(batch_size)'

The ‘data’ tensor consists of sequences of activation vectors (without applying softmax), with i-th channel in the last dimension corresponding to i-th label for i between 0 and alphabet_size-1 (i.e always 0-indexed). Alphabet size should include one additional value reserved for blank label. When ‘blank_label’ is "first", the ‘0’-th channel is be reserved for activation of blank label, or otherwise if it is "last", '(alphabet_size-1)'-th channel should be reserved for blank label.

‘label’ is an index matrix of integers. When ‘blank_label’ is "first", the value 0 is then reserved for blank label, and should not be passed in this matrix. Otherwise, when ‘blank_label’ is "last", the value '(alphabet_size-1)' is reserved for blank label.

If a sequence of labels is shorter than *label_sequence_length*, use the special padding value at the end of the sequence to conform it to the correct length. The padding value is ‘0’ when ‘blank_label’ is "first", and ‘-1’ otherwise.

For example, suppose the vocabulary is ['a', 'b', 'c'], and in one batch we have three sequences 'ba', 'cbb', and 'abac'. When ‘blank_label’ is "first", we can index the labels as 'a': 1, 'b': 2, 'c': 3, and we reserve the 0-th channel for blank label in data tensor. The resulting ‘label’ tensor should be padded to be:

```
[[2, 1, 0, 0], [3, 2, 2, 0], [1, 2, 1, 3]]
```

When ‘blank_label’ is "last", we can index the labels as 'a': 0, 'b': 1, 'c': 2, and we reserve the 0-th channel for blank label in data tensor. The resulting ‘label’ tensor should be padded to be:

```
[[1, 0, -1, -1], [2, 1, 1, -1], [0, 1, 0, 2]]
```

“out” is a list of CTC loss values, one per example in the batch.

See *Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks*, A. Graves *et al*. for more information on the definition and the algorithm.

Defined in src/operator/nn/ctc_loss.cc:L100

Value

out The result mx.ndarray
**mx.nd.cumsum**

Return the cumulative sum of the elements along a given axis.

**Description**

Defined in src/operator/numpy/np_cumsum.cc:L70

**Arguments**

- **a**: NDArray-or-Symbol Input ndarray
- **axis**: int or None, optional, default=’None’ Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.
- **dtype**: None, ‘float16’, ‘float32’, ‘float64’, ‘int32’, ‘int64’, ‘int8’, optional, default=’None’ Type of the returned array and of the accumulator in which the elements are summed. If dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

**Value**

- **out**: The result mx.ndarray

---

**mx.nd.Custom**

Apply a custom operator implemented in a frontend language (like Python).

**Description**

Custom operators should override required methods like ‘forward’ and ‘backward’. The custom operator must be registered before it can be used. Please check the tutorial here: https://mxnet.incubator.apache.org/api/faq/new_op

**Arguments**

- **data**: NDArray-or-Symbol[] Input data for the custom operator.
- **op.type**: string Name of the custom operator. This is the name that is passed to ‘mx.operator.register’ to register the operator.

**Details**

Defined in src/operator/custom/custom.cc:L547

**Value**

- **out**: The result mx.ndarray
Computes 1D or 2D transposed convolution (aka fractionally strided convolution) of the input tensor. This operation can be seen as the gradient of Convolution operation with respect to its input. Convolution usually reduces the size of the input. Transposed convolution works the other way, going from a smaller input to a larger output while preserving the connectivity pattern.

Description

Computes 1D or 2D transposed convolution (aka fractionally strided convolution) of the input tensor. This operation can be seen as the gradient of Convolution operation with respect to its input. Convolution usually reduces the size of the input. Transposed convolution works the other way, going from a smaller input to a larger output while preserving the connectivity pattern.

Arguments

data
NDArray-or-Symbol Input tensor to the deconvolution operation.

weight
NDArray-or-Symbol Weights representing the kernel.

bias
NDArray-or-Symbol Bias added to the result after the deconvolution operation.

kernel
Shape(tuple), required Deconvolution kernel size: (w,), (h, w) or (d, h, w). This is same as the kernel size used for the corresponding convolution

stride
Shape(tuple), optional, default=[] The stride used for the corresponding convolution: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.

dilate
Shape(tuple), optional, default=[] Dilation factor for each dimension of the input: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.

pad
Shape(tuple), optional, default=[] The amount of implicit zero padding added during convolution for each dimension of the input: (w,), (h, w) or (d, h, w). “(kernel-1)/2” is usually a good choice. If ‘target_shape’ is set, ‘pad’ will be ignored and a padding that will generate the target shape will be used. Defaults to no padding.

adj
Shape(tuple), optional, default=[] Adjustment for output shape: (w,), (h, w) or (d, h, w). If ‘target_shape’ is set, ‘adj’ will be ignored and computed accordingly.

target.shape
Shape(tuple), optional, default=[] Shape of the output tensor: (w,), (h, w) or (d, h, w).

num.filter
int (non-negative), required Number of output filters.

num.group
int (non-negative), optional, default=1 Number of groups partition.

workspace
long (non-negative), optional, default=512 Maximum temporary workspace allowed (MB) in deconvolution. This parameter has two usages. When CUDNN is not used, it determines the effective batch size of the deconvolution kernel. When CUDNN is used, it controls the maximum temporary storage used for tuning the best CUDNN kernel when ‘limited_workspace’ strategy is used.
no.bias boolean, optional, default=1 Whether to disable bias parameter.
cudnn.tune None, 'fastest', 'limited_workspace', 'off', optional, default='None' Whether to pick convolution algorithm by running performance test.
cudnn.off boolean, optional, default=0 Turn off cudnn for this layer.
layout None, 'NCDHW', 'NCHW', 'NCW', 'NDHWC', 'NHWC', optional, default='None' Set layout for input, output and weight. Empty for default layout, NCW for 1d, NCHW for 2d and NCDHW for 3d. NHWC and NDHWC are only supported on GPU.

Value

out The result mx.ndarray

mx.nd.degrees

Converting each element of the input array from radians to degrees.

Description

.. math:: degrees([0, \pi/2, \pi, 3\pi/2, 2\pi]) = [0, 90, 180, 270, 360]

Arguments

data NDArray-or-Symbol The input array.

Details

The storage type of “degrees“ output depends upon the input storage type:

- degrees(default) = default - degrees(row_sparse) = row_sparse - degrees(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L332

Value

out The result mx.ndarray
**mx.nd.depth.to.space**

Rearranges (permutes) data from depth into blocks of spatial data. Similar to ONNX DepthToSpace operator: https://github.com/onnx/onnx/blob/master/docs/Operators.md#DepthToSpace. The output is a new tensor where the values from depth dimension are moved in spatial blocks to height and width dimension. The reverse of this operation is "space_to_depth".  

\[
\begin{align*}
  \mathbf{x}' &= \text{reshape}(x, [N, \text{block}\_\text{size}, \text{block}\_\text{size}, C / (\text{block}\_\text{size}^2), H * \text{block}\_\text{size}, W * \text{block}\_\text{size}]) \\
  \mathbf{x}'' &= \text{transpose}(x', [0, 3, 4, 1, 5, 2]) \\
  y &= \text{reshape}(x'', [N, C / (\text{block}\_\text{size}^2), H * \text{block}\_\text{size}, W * \text{block}\_\text{size}]) \\
\end{align*}
\]

where \(\mathbf{x}'\) is an input tensor with default layout as \([N, C, H, W]\) : \([\text{batch, channels, height, width}]\) and \(\mathbf{y}\) is the output tensor of layout \([N, C / (\text{block}\_\text{size}^2), H * \text{block}\_\text{size}, W * \text{block}\_\text{size}]\).  

Example:

\[
\begin{align*}
  x &= \begin{bmatrix}
     [0, 1, 2] & [3, 4, 5] \\
     [6, 7, 8] & [9, 10, 11] \\
     [12, 13, 14] & [15, 16, 17] \\
     [18, 19, 20] & [21, 22, 23] \\
   \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
  \text{depth_to_space}(x, 2) &= \begin{bmatrix}
     [0, 6, 1, 7, 2, 8] \\
     [12, 18, 13, 19, 14, 20] \\
     [3, 9, 4, 10, 5, 11] \\
     [15, 21, 16, 22, 17, 23] \\
   \end{bmatrix}
\end{align*}
\]

**Description**

Defined in src/operator/tensor/matrix_op.cc:L972

**Arguments**

- **data**
  - NDArray-or-Symbol
  - Input ndarray

- **block.size**
  - int, required
  - Blocks of \([\text{block}\_\text{size, block}\_\text{size}]\) are moved

**Value**

- **out**
  - The result mx.nd.array

---

**mx.nd.diag**

Extracts a diagonal or constructs a diagonal array.

**Description**

“diag”’s behavior depends on the input array dimensions:

**Arguments**

- **data**
  - NDArray-or-Symbol
  - Input ndarray

- **k**
  - int, optional, default=’0’
  - Diagonal in question. The default is 0. Use k>0 for diagonals above the main diagonal, and k<0 for diagonals below the main diagonal. If input has shape \([S0 S1]\) k must be between \(-S0\) and \(S1\)
mx.nd.dot

axis1 int, optional, default='0' The first axis of the sub-arrays of interest. Ignored when the input is a 1-D array.

axis2 int, optional, default='1' The second axis of the sub-arrays of interest. Ignored when the input is a 1-D array.

Details

- 1-D arrays: constructs a 2-D array with the input as its diagonal, all other elements are zero. - N-D arrays: extracts the diagonals of the sub-arrays with axes specified by “axis1” and “axis2”. The output shape would be decided by removing the axes numbered “axis1” and “axis2” from the input shape and appending to the result a new axis with the size of the diagonals in question.

For example, when the input shape is '(2, 3, 4, 5)', “axis1” and “axis2” are 0 and 2 respectively and “k” is 0, the resulting shape would be '(3, 5, 2)'.

Examples::

```python
x = [[1, 2, 3], [4, 5, 6]]
diag(x) = [1, 5]
diag(x, k=1) = [2, 6]
diag(x, k=-1) = [4]
x = [1, 2, 3]
diag(x) = [[1, 0, 0], [0, 2, 0], [0, 0, 3]]
diag(x, k=1) = [[0, 1, 0], [0, 0, 2], [0, 0, 0]]
diag(x, k=-1) = [[0, 0, 0], [1, 0, 0], [0, 2, 0]]
x = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
diag(x) = [[1, 7], [2, 8]]
diag(x, k=1) = [[3, 4]]
diag(x, axis1=-2, axis2=-1) = [[1, 4], [5, 8]]
```

Defined in src/operator/tensor/diag_op.cc:L87

Value

out The result mx.ndarray

mx.nd.dot

Dot product of two arrays.

Description

“dot”’s behavior depends on the input array dimensions:
**Arguments**

- **lhs**: NDArray-or-Symbol The first input
- **rhs**: NDArray-or-Symbol The second input
- **transpose.a**: boolean, optional, default=0 If true then transpose the first input before dot.
- **transpose.b**: boolean, optional, default=0 If true then transpose the second input before dot.
- **forward_stype**: None, 'csr', 'default', 'row_sparse',optional, default=None The desired storage type of the forward output given by user, if the combination of input storage types and this hint does not match any implemented ones, the dot operator will perform fallback operation and still produce an output of the desired storage type.

**Details**

- 1-D arrays: inner product of vectors - 2-D arrays: matrix multiplication - N-D arrays: a sum product over the last axis of the first input and the first axis of the second input

For example, given 3-D “x” with shape ‘(n,m,k)’ and “y” with shape ‘(k,r,s)’, the result array will have shape ‘(n,m,r,s)’. It is computed by:

\[
\text{dot}(x,y)[i,j,a,b] = \sum x[i,j,:] \times y[:,a,b]
\]

**Example:**

\[
x = \text{reshape}([0,1,2,3,4,5,6,7], \text{shape}=(2,2,2))
y = \text{reshape}([7,6,5,4,3,2,1,0], \text{shape}=(2,2,2))
\text{dot}(x,y)[0,0,1,1] = 0
\]

The storage type of “dot” output depends on storage types of inputs, transpose option and forward_stype option for output storage type. Implemented sparse operations include:

- \text{dot}(\text{default}, \text{default}, \text{transpose_a}=\text{True}/\text{False}, \text{transpose_b}=\text{True}/\text{False}) = \text{default}
- \text{dot}(\text{csr}, \text{default}, \text{transpose_a}=\text{True}) = \text{row_sparse}
- \text{dot}(\text{default}, \text{csr}) = \text{default}
- \text{dot}(\text{default}, \text{csr}, \text{forward_stype}=\text{''default'')} = \text{default}

If the combination of input storage types and forward_stype does not match any of the above patterns, “dot” will fallback and generate output with default storage.

**Note:**

If the storage type of the lhs is "csr", the storage type of gradient w.r.t rhs will be "row_sparse". Only a subset of optimizers support sparse gradients, including SGD, AdaGrad and Adam. Note that by default lazy updates is turned on, which may perform differently from standard updates. For more details, please check the Optimization API at: https://mxnet.incubator.apache.org/api/python/optimization/optimization.html

Defined in src/operator/tensor/dot.cc:L77

**Value**

- **out**: The result mx.ndarray
mx.nd.Dropout

Applies dropout operation to input array.

Description

- During training, each element of the input is set to zero with probability p. The whole array is rescaled by \(1/(1-p)\) to keep the expected sum of the input unchanged.

Arguments

- **data**: NDArray-or-Symbol Input array to which dropout will be applied.
- **p**: float, optional, default=0.5 Fraction of the input that gets dropped out during training time.
- **mode**: ‘always’, ‘training’, optional, default=’training’ Whether to only turn on dropout during training or to also turn on for inference.
- **axes**: Shape(tuple), optional, default=[] Axes for variational dropout kernel.
- **cudnn.off**: boolean or None, optional, default=0 Whether to turn off cudnn in dropout operator. This option is ignored if axes is specified.

Details

- During testing, this operator does not change the input if mode is ‘training’. If mode is ‘always’, the same computation as during training will be applied.

Example:

```python
random.seed(998) input_array = array([[3., 0.5, -0.5, 2., 7.], [2., -0.4, 7., 3., 0.2]]) a = symbol.Variable('a') dropout = symbol.Dropout(a, p = 0.2) executor = dropout.simple_bind(a = input_array.shape)
```

```python
## If training executor.forward(is_train = True, a = input_array) executor.outputs [[ 3.75 0.625 -0.2.5 8.75 ] [ 2.5 -0.5 8.75 3.75 0. ]]
## If testing executor.forward(is_train = False, a = input_array) executor.outputs [[ 3. 0.5 -0.5 2. 7. ] [ 2. -0.4 7. 3. 0.2 ]]
```

Defined in src/operator/nn/dropout.cc:L96

Value

- **out**: The result mx.ndarray
**mx.nd.ElementWiseSum**

**Adds all input arguments element-wise.**

**Description**

.. math:: \text{add\_n}(a_1, a_2, \ldots, a_n) = a_1 + a_2 + \ldots + a_n

**Arguments**

- **args** NDArray-or-Symbol[] Positional input arguments

**Details**

“add\_n” is potentially more efficient than calling “add” by ‘n’ times.

The storage type of “add\_n” output depends on storage types of inputs

- add\_n(row_sparse, row_sparse, ..) = row_sparse
- add\_n(default, csr, default) = default
- add\_n(csr, default) = default
- add\_n(csr, csr) = csr
- add\_n(rsp, default) = default
- add\_n(rsp, default) = default
- otherwise, “add\_n” generates output with default storage

Defined in src/operator/tensor/elemwise_sum.cc:L155

**Value**

- **out** The result mx.ndarray

---

**mx.nd.elemwise.add**

**Adds arguments element-wise.**

**Description**

The storage type of “elemwise_add” output depends on storage types of inputs

**Arguments**

- **lhs** NDArray-or-Symbol first input
- **rhs** NDArray-or-Symbol second input

**Details**

- elemwise_add(row_sparse, row_sparse) = row_sparse
- elemwise_add(csr, csr) = csr
- elemwise_add(default, csr) = default
- elemwise_add(csr, default) = default
- elemwise_add(default, rsp) = default
- elemwise_add(rsp, default) = default
- otherwise, “elemwise_add” generates output with default storage

**Value**

- **out** The result mx.ndarray
**mx.nd.elemwise.div**  
*Divides arguments element-wise.*

**Description**

The storage type of “elemwise_div” output is always dense.

**Arguments**

- `lhs`  
  NDArray-or-Symbol first input
- `rhs`  
  NDArray-or-Symbol second input

**Value**

- `out`  
  The result mx.ndarray

---

**mx.nd.elemwise.mul**  
*Multiplies arguments element-wise.*

**Description**

The storage type of “elemwise_mul” output depends on storage types of inputs.

**Arguments**

- `lhs`  
  NDArray-or-Symbol first input
- `rhs`  
  NDArray-or-Symbol second input

**Details**

- `elemwise_mul(default, default) = default`
- `elemwise_mul(row_sparse, row_sparse) = row_sparse`
- `elemwise_mul(default, row_sparse) = row_sparse`
- `elemwise_mul(row_sparse, default) = row_sparse`
- `elemwise_mul(csr, csr) = csr`
  - otherwise, “elemwise_mul” generates output with default storage

**Value**

- `out`  
  The result mx.ndarray
**mx.nd.elemwise.sub**  
*Subtracts arguments element-wise.*

**Description**

The storage type of "elemwise_sub" output depends on storage types of inputs.

**Arguments**

- **lhs**: NDArray-or-Symbol first input
- **rhs**: NDArray-or-Symbol second input

**Details**

- `elemwise_sub(row_sparse, row_sparse) = row_sparse - elemwise_sub(csr, csr) = csr - elemwise_sub(default, csr) = default - elemwise_sub(csr, default) = default - elemwise_sub(default, rsp) = default - elemwise_sub(rsp, default) = default - otherwise, "elemwise_sub" generates output with default storage`

**Value**

- **out**: The result mx.ndarray

**mx.nd.Embedding**  
*Maps integer indices to vector representations (embeddings).*

**Description**

This operator maps words to real-valued vectors in a high-dimensional space, called word embeddings. These embeddings can capture semantic and syntactic properties of the words. For example, it has been noted that in the learned embedding spaces, similar words tend to be close to each other and dissimilar words far apart.

**Arguments**

- **data**: NDArray-or-Symbol The input array to the embedding operator.
- **weight**: NDArray-or-Symbol The embedding weight matrix.
- **input_dim**: int, required Vocabulary size of the input indices.
- **output_dim**: int, required Dimension of the embedding vectors.
- **dtype**: 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', optional, default='float32' Data type of weight.
- **sparse.grad**: boolean, optional, default=0 Compute row sparse gradient in the backward calculation. If set to True, the grad’s storage type is row_sparse.
Details
For an input array of shape (d1, ..., dK), the shape of an output array is (d1, ..., dK, output_dim).
All the input values should be integers in the range [0, input_dim).
If the input_dim is ip0 and output_dim is op0, then shape of the embedding weight matrix must be
(ip0, op0).
When "sparse_grad" is False, if any index mentioned is too large, it is replaced by the index that
addresses the last vector in an embedding matrix. When "sparse_grad" is True, an error will be
raised if invalid indices are found.
Examples::
input_dim = 4 output_dim = 5
// Each row in weight matrix y represents a word. So, y = (w0,w1,w2,w3) y = [[ 0., 1., 2., 3., 4.],
// Input array x represents n-grams(2-gram). So, x = [(w1,w3), (w0,w2)] x = [[ 1., 3.], [ 0., 2.]]
// Mapped input x to its vector representation y. Embedding(x, y, 4, 5) = [[[ 5., 6., 7., 8., 9.],
[ 15., 16., 17., 18., 19.]],
[[ 0., 1., 2., 3., 4.], [ 10., 11., 12., 13., 14.]]]
The storage type of weight can be either row_sparse or default.
.. Note::
If "sparse_grad" is set to True, the storage type of gradient w.r.t weights will be "row_sparse". Only
a subset of optimizers support sparse gradients, including SGD, AdaGrad and Adam. Note that by
default lazy updates is turned on, which may perform differently from standard updates. For more
details, please check the Optimization API at: https://mxnet.incubator.apache.org/api/python/optimization/optimization.html
Defined in src/operator/tensor/indexing_op.cc:L603

Value
out The result mx.ndarray

mx.nd.erf

Returns element-wise gauss error function of the input.

description
Example::

Arguments

data NDArray-or-Symbol The input array.

Details
erf([0., -1., 10.]) = [0., -0.8427, 1.]
Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L886
**mx.nd.erfinv**

Returns element-wise inverse gauss error function of the input.

**Description**

Example:

```
args = [0., 0.5., -1.]
result = erfinv(args)
```

**Arguments**

- `data` (NDArray-or-Symbol) The input array.

**Details**

```
erfinv([0, 0.5, -1]) = [0., 0.4769, -inf]
```

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L908

**Value**

- `out` The result mx.ndarray

**mx.nd.exp**

Returns element-wise exponential value of the input.

**Description**

.. math:: \exp(x) = e^x \approx 2.718^x

**Arguments**

- `data` (NDArray-or-Symbol) The input array.

**Details**

```
exp([0, 1, 2]) = [1., 2.71828175, 7.38905621]
```

The storage type of “exp” output is always dense

Defined in src/operator/tensor/elemwise_unary_op_logexp.cc:L64

**Value**

- `out` The result mx.ndarray
**mx.nd.expand_dims**

Inserts a new axis of size 1 into the array shape. For example, given “x” with shape “(2,3,4)”, then “expand_dims(x, axis=1)” will return a new array with shape “(2,1,3,4)”. 

**Description**

Defined in src/operator/tensor/matrix_op.cc:L395

**Arguments**

- **data**: NDArray-or-Symbol, Source input
- **axis**: int, required, Position where new axis is to be inserted. Suppose that the input ‘NDArray’’s dimension is ‘ndim’, the range of the inserted axis is ‘[-ndim, ndim]’

**Value**

- **out**: The result mx.ndarray

---

**mx.nd.expm1**

Returns “exp(x) - 1” computed element-wise on the input.

**Description**

This function provides greater precision than “exp(x) - 1” for small values of “x”.

**Arguments**

- **data**: NDArray-or-Symbol, The input array.

**Details**

The storage type of “expm1” output depends upon the input storage type:
- expm1(default) = default - expm1(row_sparse) = row_sparse - expm1(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_logexp.cc:L244

**Value**

- **out**: The result mx.ndarray
mx.nd.fill.element.0index

Fill one element of each line (row for python, column for R/Julia) in lhs according to index indicated by rhs and values indicated by mhs. This function assume rhs uses 0-based index.

Description

Fill one element of each line (row for python, column for R/Julia) in lhs according to index indicated by rhs and values indicated by mhs. This function assume rhs uses 0-based index.

Arguments

- lhs  NDArray Left operand to the function.
- mhs  NDArray Middle operand to the function.
- rhs  NDArray Right operand to the function.

Value

- out The result mx.ndarray

mx.nd.fix

Returns element-wise rounded value to the nearest integer towards zero of the input.

Description

Example::

Arguments

- data  NDArray-or-Symbol The input array.

Details

- fix([-2.1, -1.9, 1.9, 2.1]) = [-2, -1, 1, 2]
  The storage type of “fix” output depends upon the input storage type:
  - fix(default) = default - fix(row_sparse) = row_sparse - fix(csr) = csr
  Defined in src/operator/tensor/elementwise_unary_op_basic.cc:L874

Value

- out The result mx.ndarray
mx.nd.Flatten

Flattens the input array into a 2-D array by collapsing the higher dimensions. .. note:: ‘Flatten’ is deprecated. Use ‘flatten’ instead. For an input array with shape “(d1, d2, ..., dk)”, ‘flatten’ operation reshapes the input array into an output array of shape “(d1, d2*...*dk)”. Note that the behavior of this function is different from numpy.ndarray.flatten, which behaves similar to mxnet.nd.array.reshape((-1,)). Example::

```
x = [[[1,2,3],[4,5,6],[7,8,9]], [[1,2,3],[4,5,6],[7,8,9]]], flatten(x) = [[[1., 2., 3., 4., 5., 6., 7., 8., 9.], [1., 2., 3., 4., 5., 6., 7., 8., 9.]]
```

Description

Defined in src/operator/tensor/matrix_op.cc:L250

Arguments

data

NDArray-or-Symbol Input array.

Value

out The result mx.ndarray

mx.nd.flatten

Flattens the input array into a 2-D array by collapsing the higher dimensions. .. note:: ‘Flatten’ is deprecated. Use ‘flatten’ instead. For an input array with shape “(d1, d2, ..., dk)”, ‘flatten’ operation reshapes the input array into an output array of shape “(d1, d2*...*dk)”. Note that the behavior of this function is different from numpy.ndarray.flatten, which behaves similar to mxnet.nd.array.reshape((-1,)). Example::

```
x = [[[1,2,3],[4,5,6],[7,8,9]], [[1,2,3],[4,5,6],[7,8,9]]], flatten(x) = [[[1., 2., 3., 4., 5., 6., 7., 8., 9.], [1., 2., 3., 4., 5., 6., 7., 8., 9.]]
```

Description

Defined in src/operator/tensor/matrix_op.cc:L250

Arguments

data

NDArray-or-Symbol Input array.

Value

out The result mx.ndarray
**mx.nd.flip**

Reverses the order of elements along given axis while preserving array shape. Note: reverse and flip are equivalent. We use reverse in the following examples. Examples:

```
x = [[ 0., 1., 2., 3., 4.],
     [ 5., 6., 7., 8., 9.]]
reverse(x, axis=0) = [[ 5., 6., 7., 8., 9.],
                      [ 0., 1., 2., 3., 4.]]
reverse(x, axis=1) = [[ 4., 3., 2., 1., 0.],
                      [ 9., 8., 7., 6., 5.]]
```

**Description**

Defined in src/operator/tensor/matrix_op.cc:L832

**Arguments**

- **data**: NDArray-or-Symbol Input data array
- **axis**: Shape(tuple), required The axis which to reverse elements.

**Value**

- **out**: The result mx.ndarray

---

**mx.nd.floor**

Returns element-wise floor of the input.

**Description**

The floor of the scalar x is the largest integer i, such that i <= x.

**Arguments**

- **data**: NDArray-or-Symbol The input array.

**Details**

Example:

```
floor([-2.1, -1.9, 1.5, 1.9, 2.1]) = [-3., -2., 1., 1., 2.]
```

The storage type of “floor” output depends upon the input storage type:

- floor(default) = default - floor(row_sparse) = row_sparse - floor(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L836

**Value**

- **out**: The result mx.ndarray

Description

.. math::

Arguments

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **d**: NDArray-or-Symbol Internal state “d_t”
- **v**: NDArray-or-Symbol Internal state “v_t”
- **z**: NDArray-or-Symbol Internal state “z_t”
- **lr**: float, required Learning rate.
- **beta1**: float, optional, default=0.600000024 Generally close to 0.5.
- **beta2**: float, optional, default=0.999000013 Generally close to 1.
- **epsilon**: double, optional, default=9.9999999392252903e-09 Epsilon to prevent div 0.
- **t**: int, required Number of update.
- **wd**: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad * grad.
- **clip.grad**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).

Details

.. math::

Defined in src/operator/optimizer_op.cc:L631

Value

out The result mx.ndarray
mx.nd.ftrl.update  


Description

It updates the weights using::

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>NDArray-or-Symbol</td>
<td>Weight</td>
</tr>
<tr>
<td>grad</td>
<td>NDArray-or-Symbol</td>
<td>Gradient</td>
</tr>
<tr>
<td>z</td>
<td>NDArray-or-Symbol</td>
<td>z</td>
</tr>
<tr>
<td>n</td>
<td>NDArray-or-Symbol</td>
<td>Square of grad</td>
</tr>
<tr>
<td>lr</td>
<td>float, required</td>
<td>Learning rate</td>
</tr>
<tr>
<td>lamda1</td>
<td>float, optional, default=0.00999999978</td>
<td>The L1 regularization coefficient.</td>
</tr>
<tr>
<td>beta</td>
<td>float, optional, default=1</td>
<td>Per-Coordinate Learning Rate beta.</td>
</tr>
<tr>
<td>wd</td>
<td>float, optional, default=0</td>
<td>Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.</td>
</tr>
<tr>
<td>rescale.grad</td>
<td>float, optional, default=1</td>
<td>Rescale gradient to grad = rescale_grad*grad.</td>
</tr>
<tr>
<td>clip.gradient</td>
<td>float, optional, default=-1</td>
<td>Clip gradient to the range of [-clip_gradient, clip_gradient] if clip_gradient &lt;= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).</td>
</tr>
</tbody>
</table>

Details

rescaled_grad = clip(grad * rescale_grad, clip_gradient)  
z += rescaled_grad - (sqrt(n + rescaled_grad**2) - sqrt(n)) * weight / learning_rate  
n += rescaled_grad**2  
w = (sign(z) * lamda1 - z) / ((beta + sqrt(n)) / learning_rate + wd) * (abs(z) > lamda1)

If w, z and n are all of “row_sparse” storage type, only the row slices whose indices appear in grad.indices are updated (for w, z and n):::

for row in grad.indices:  
rescaled_grad[row] = clip(grad[row] * rescale_grad[row], clip_gradient)  
z[row] += rescaled_grad[row] - (sqrt(n[row] + rescaled_grad[row]**2) - sqrt(n[row])) * weight[row] / learning_rate  
n[row] += rescaled_grad[row]**2  
w[row] = (sign(z[row]) * lamda1 - z[row]) / ((beta + sqrt(n[row])) / learning_rate + wd) * (abs(z[row]) > lamda1)

Defined in src/operator/optimizer_op.cc:L867

Value

out The result mx.ndarray
mx.nd.FullyConnected

Applies a linear transformation: $Y = XW^T + b$.

Description

If “flatten” is set to be true, then the shapes are:

Arguments

data NDArray-or-Symbol Input data.
weight NDArray-or-Symbol Weight matrix.
bias NDArray-or-Symbol Bias parameter.
num.hidden int, required Number of hidden nodes of the output.
no.bias boolean, optional, default=0 Whether to disable bias parameter.
flatten boolean, optional, default=1 Whether to collapse all but the first axis of the input data tensor.

Details

- **data**: `(batch_size, x1, x2, ..., xn)` - **weight**: `(num_hidden, x1 * x2 * ... * xn)` - **bias**: `(num_hidden,)` - **out**: `(batch_size, num_hidden)`

If “flatten” is set to be false, then the shapes are:

- **data**: `(x1, x2, ..., xn, input_dim)` - **weight**: `(num_hidden, input_dim)` - **bias**: `(num_hidden,)` - **out**: `(x1, x2, ..., xn, num_hidden)`

The learnable parameters include both “weight” and “bias”.

If “no_bias” is set to be true, then the “bias” term is ignored.

.. Note::

The sparse support for FullyConnected is limited to forward evaluation with ‘row_sparse’ weight and bias, where the length of ‘weight.indices’ and ‘bias.indices’ must be equal to ‘num_hidden’. This could be useful for model inference with ‘row_sparse’ weights trained with importance sampling or noise contrastive estimation.

To compute linear transformation with ‘csr’ sparse data, sparse.dot is recommended instead of sparse.FullyConnected.

Defined in src/operator/nn/fully_connected.cc:L287

Value

out The result mx.ndarray
**mx.nd.gamma**

Returns the gamma function (extension of the factorial function to the reals), computed element-wise on the input array.

**Description**

The storage type of “gamma“ output is always dense

**Arguments**

- **data** NDArray-or-Symbol The input array.

**Value**

- **out** The result mx.ndarray

**mx.nd.gammaln**

Returns element-wise log of the absolute value of the gamma function of the input.

**Description**

The storage type of “gammaln“ output is always dense

**Arguments**

- **data** NDArray-or-Symbol The input array.

**Value**

- **out** The result mx.ndarray
### mx.nd.gather.nd

*Gather elements or slices from ‘data’ and store to a tensor whose shape is defined by ‘indices’.*

**Description**

Given ‘data’ with shape ‘(X_0, X_1, ..., X_N-1)’ and indices with shape ‘(M, Y_0, ..., Y_K-1)’, the output will have shape ‘(Y_0, ..., Y_K-1, X_M, ..., X_N-1)’, where ‘M <= N’. If ‘M == N’, output shape will simply be ‘(Y_0, ..., Y_K-1)’.

**Arguments**

- **data**: NDArray-or-Symbol data
- **indices**: NDArray-or-Symbol indices

**Details**

The elements in output is defined as follows:

\[
\text{output}[y_0, ..., y_{K-1}, x_M, ..., x_{N-1}] = \text{data}[\text{indices}[0, y_0, ..., y_{K-1}], ..., \text{indices}[M-1, y_0, ..., y_{K-1}], x_M, ..., x_{N-1}]
\]

**Examples**

- `data = \[[0, 1], [2, 3]\]` `indices = \[[1, 1, 0], [0, 1, 0]\]` `gather_nd(data, indices) = [2, 3, 0]`
- `data = \[[[1, 2], [3, 4]], [[5, 6], [7, 8]]\]` `indices = [[0, 1], [1, 0]]` `gather_nd(data, indices) = [[3, 4], [5, 6]]`

**Value**

- **out**: The result mx.ndarray

### mx.nd.GridGenerator

*Generates 2D sampling grid for bilinear sampling.*

**Description**

Generates 2D sampling grid for bilinear sampling.

**Arguments**

- **data**: NDArray-or-Symbol Input data to the function.
- **transform.type**: ‘affine’, ‘warp’, required The type of transformation. For ‘affine’, input data should be an affine matrix of size (batch, 6). For ‘warp’, input data should be an optical flow of size (batch, 2, h, w).
- **target.shape**: Shape(tuple), optional, default=[0,0] Specifies the output shape (H, W). This is required if transformation type is ‘affine’. If transformation type is ‘warp’, this parameter is ignored.
Value

out The result mx.ndarray

mx.nd.GroupNorm Group normalization.

Description

The input channels are separated into “num_groups” groups, each containing “num_channels / num_groups” channels. The mean and standard-deviation are calculated separately over each group.

Arguments

data NDArray-or-Symbol Input data

gamma NDArray-or-Symbol gamma array

beta NDArray-or-Symbol beta array

num.groups int, optional, default='1' Total number of groups.

eps float, optional, default=9.99999975e-06 An ‘epsilon’ parameter to prevent division by 0.

output.mean.var boolean, optional, default=0 Output the mean and std calculated along the given axis.

Details

.. math::

data = data.reshape((N, num_groups, C // num_groups, ...))

out = data - mean(data, axis)\sqrt{var(data, axis)} + \epsilon * gamma + beta

Both “gamma” and “beta” are learnable parameters.

Defined in src/operator/nn/group_norm.cc:L77

Value

out The result mx.ndarray
**mx.nd.hard.sigmoid**  Computes hard sigmoid of x element-wise.

**Description**

.. math:: y = \max(0, \min(1, \alpha x + \beta))

**Arguments**

- **data**  NDArray-or-Symbol  The input array.
- **alpha**  float, optional, default=0.200000003  Slope of hard sigmoid
- **beta**  float, optional, default=0.5  Bias of hard sigmoid.

**Details**

Defined in src/operator/tensor/elemwise UnaryOpBasic.cc:L161

**Value**

- **out**  The result mx.ndarray

---

**mx.nd.identity**  Returns a copy of the input.

**Description**

From:src/operator/tensor/Elemwise UnaryOpBasic.cc:244

**Arguments**

- **data**  NDArray-or-Symbol  The input array.

**Value**

- **out**  The result mx.ndarray
mx.nd.IdentityAttachKLSparseReg

Apply a sparse regularization to the output a sigmoid activation function.

Description

Apply a sparse regularization to the output a sigmoid activation function.

Arguments

- **data** NDArray-or-Symbol Input data.
- **sparseness.target** float, optional, default=0.1 The sparseness target
- **penalty** float, optional, default=0.001 The tradeoff parameter for the sparseness penalty
- **momentum** float, optional, default=0.899 The momentum for running average

Value

- **out** The result mx.ndarray

mx.nd.im2col

Extract sliding blocks from input array.

Description

This operator is used in vanilla convolution implementation to transform the sliding blocks on image to column matrix, then the convolution operation can be computed by matrix multiplication between column and convolution weight. Due to the close relation between im2col and convolution, the concept of **kernel**, **stride**, **dilate** and **pad** in this operator are inherited from convolution operation.

Arguments

- **data** NDArray-or-Symbol Input array to extract sliding blocks.
- **kernel** Shape(tuple), required Sliding kernel size: (w), (h, w) or (d, h, w).
- **stride** Shape(tuple), optional, default=[] The stride between adjacent sliding blocks in spatial dimension: (w), (h, w) or (d, h, w). Defaults to 1 for each dimension.
- **dilate** Shape(tuple), optional, default=[] The spacing between adjacent kernel points: (w), (h, w) or (d, h, w). Defaults to 1 for each dimension.
- **pad** Shape(tuple), optional, default=[] The zero-value padding size on both sides of spatial dimension: (w), (h, w) or (d, h, w). Defaults to no padding.
Details

Given the input data of shape \( (N, C, *) \), where \( N \) is the batch size, \( C \) is the channel size, and \( * \) is the arbitrary spatial dimension, the output column array is always with shape \( (N, C \times \prod(\text{kernel}), W) \), where \( C \times \prod(\text{kernel}) \) is the block size, and \( W \) is the block number which is the spatial size of the convolution output with same input parameters. Only 1-D, 2-D and 3-D of spatial dimension is supported in this operator.

Defined in src/operator/nn/im2col.cc:L100

Value

| out | The result mx.ndarray |

mx.nd.InstanceNorm

Applies instance normalization to the n-dimensional input array.

Description

This operator takes an n-dimensional input array where \( n > 2 \) and normalizes the input using the following formula:

\[
\text{out} = \frac{x - \text{mean}[\text{data}]}{\sqrt{\text{Var}[\text{data}] + \epsilon}} \times \text{gamma} + \text{beta}
\]

This layer is similar to batch normalization layer ('BatchNorm') with two differences: first, the normalization is carried out per example (instance), not over a batch. Second, the same normalization is applied both at test and train time. This operation is also known as 'contrast normalization'.

If the input data is of shape \( [\text{batch}, \text{channel}, \text{spacial_dim1}, \text{spacial_dim2}, ...] \), ‘gamma’ and ‘beta’ parameters must be vectors of shape \( [\text{channel}] \).

This implementation is based on this paper [1].

Examples:
// Input of shape (2,1,2) x = [[[ 1.1, 2.2]], [[ 3.3, 4.4]]]
// gamma parameter of length 1 gamma = [1.5]
// beta parameter of length 1 beta = [0.5]
// Instance normalization is calculated with the above formula InstanceNorm(x,gamma,beta) = [[[-0.9975277 , 1.99752665]], [[-0.99752653, 1.99752724]]]
Defined in src/operator/instance_norm.cc:L95

Value
out The result mx.ndarray
mx.nd.L2Normalization

Normalize the input array using the L2 norm.

Description
For 1-D NDArray, it computes:

Arguments
- **data**: NDArray-or-Symbol Input array to normalize.
- **eps**: float, optional, default=1.00000001e-10 A small constant for numerical stability.
- **mode**: 'channel', 'instance', 'spatial', optional, default='instance' Specify the dimension along which to compute L2 norm.

Details
\[
\text{out} = \frac{\text{data}}{\sqrt{\sum(\text{data}^2) + \text{eps}}}
\]

For N-D NDArray, if the input array has shape (N, N, ..., N),
with "mode" = "instance", it normalizes each instance in the multidimensional array by its L2 norm.:

\[
\text{for } i \in 0...N \text{ out}[i,;:,;,...] = \frac{\text{data}[i,;:,;,...]}{\sqrt{\sum(\text{data}[i,;:,;,...]^2) + \text{eps}}}
\]

with "mode" = "channel", it normalizes each channel in the array by its L2 norm.:

\[
\text{for } i \in 0...N \text{ out}[:,i,:,;,...] = \frac{\text{data}[:,i,:,;,...]}{\sqrt{\sum(\text{data}[:,i,:,;,...]^2) + \text{eps}}}
\]

with "mode" = "spatial", it normalizes the cross channel norm for each position in the array by its L2 norm.:

\[
\text{for dim in 2...N for } i \in 0...N \text{ out}[:;:;:;i,...] = \frac{\text{take(out, indices=i, axis=dim)}}{\sqrt{\sum(\text{take(out, indices=i, axis=dim)}^2) + \text{eps}}} -\text{dim}\n\]

Example:
\[
x = [[[1,2], [3,4]], [[2,2], [5,6]]]
\]
\[
\text{L2Normalization}(x, \text{mode}='\text{instance}') = [[[0.18257418 0.36514837] [0.54772252 0.73029673]] [[0.24077171 0.24077171] [0.60192931 0.7231513]]]
\]
\[
\text{L2Normalization}(x, \text{mode}='\text{channel}') = [[[0.31622776 0.44721359] [0.94868326 0.89442718]] [[0.37139067 0.31622776] [0.92847669 0.94868326]]]
\]
\[
\text{L2Normalization}(x, \text{mode}='\text{spatial}') = [[[0.44721359 0.89442718] [0.60000002 0.80000001]] [[0.70710677 0.70710677] [0.6401844 0.76822126]]]
\]

Defined in src/operator/l2_normalization.cc:L196

Value
- **out**: The result mx.ndarray
Phase I of lamb update it performs the following operations and returns g:

Description


Arguments

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **mean**: NDArray-or-Symbol Moving mean
- **var**: NDArray-or-Symbol Moving variance
- **beta1**: float, optional, default=0.899999976 The decay rate for the 1st moment estimates.
- **beta2**: float, optional, default=0.999000013 The decay rate for the 2nd moment estimates.
- **epsilon**: float, optional, default=9.99999997e-07 A small constant for numerical stability.
- **t**: int, required Index update count.
- **bias_correction**: boolean, optional, default=1 Whether to use bias correction.
- **wd**: float, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).

Details

.. math:: \begin{gather*}
\text{grad} = \text{grad} \times \text{rescale} \_\text{grad} \text{ if (grad} < -\text{clip} \_\text{gradient}) \text{ then grad} = -\text{clip} \_\text{gradient} \\
\text{if (grad} > \text{clip} \_\text{gradient}) \text{ then grad} = \text{clip} \_\text{gradient} \\
\text{mean} = \text{beta1} \times \text{mean} + (1 - \text{beta1}) \times \text{grad}; \text{variance} = \text{beta2} \times \text{variance} + (1 - \text{beta2}) \times \text{grad}^2; \\
\text{if (bias} \_\text{correction}) \text{ then mean} \_\text{hat} = \text{mean} / (1 - \text{beta1}^t) \text{; var} \_\text{hat} = \text{var} / (1 - \text{beta2}^t); \text{g} = \text{mean} \_\text{hat} / (\text{var} \_\text{hat}^{(1/2)} + \text{epsilon}) + \text{wd} \times \text{weight}; \text{else g} = \text{mean} / (\text{var} \_\text{data}^{(1/2)} + \text{epsilon}) + \text{wd} \times \text{weight}; \\
\end{gather*}

Defined in src/operator/optimizer_op.cc:L944

Value

- **out**: The result mx.ndarray
mx.nd.lamb.update.phase2

Phase II of lamb update it performs the following operations and updates grad.

Description


Arguments

- **weight**: NDArray-or-Symbol Weight
- **g**: NDArray-or-Symbol Output of lamb_update_phase 1
- **r1**: NDArray-or-Symbol r1
- **r2**: NDArray-or-Symbol r2
- **lr**: float, required Learning rate
- **lower.bound**: float, optional, default=-1 Lower limit of norm of weight. If lower_bound <= 0, Lower limit is not set
- **upper.bound**: float, optional, default=-1 Upper limit of norm of weight. If upper_bound <= 0, Upper limit is not set

Details

.. math:: \begin{gather*}
   \text{if (lower_bound >= 0) then } r1 = \max(r1, \text{lower_bound}) \\
   \text{if (upper_bound >= 0) then } r1 = \max(r1, \text{upper_bound}) \\
   \text{if (r1 == 0 or r2 == 0) then lr = lr else lr = lr * (r1/r2)} \text{ weight = weight - lr * g} \end{gather*}

Defined in src/operator/optimizer_op.cc:L983

Value

out The result mx.ndarray

mx.nd.LayerNorm

Layer normalization.

Description

Normalizes the channels of the input tensor by mean and variance, and applies a scale “gamma“ as well as offset “beta“.
**Arguments**

- **data**: NDArray-or-Symbol Input data to layer normalization
- **gamma**: NDArray-or-Symbol gamma array
- **beta**: NDArray-or-Symbol beta array
- **axis**: int, optional, default=-1 The axis to perform layer normalization. Usually, this should be axis of the channel dimension. Negative values means indexing from right to left.
- **eps**: float, optional, default=9.99999975e-06 An ‘epsilon’ parameter to prevent division by 0.
- **output.mean.var**: boolean, optional, default=0 Output the mean and std calculated along the given axis.

**Details**

Assume the input has more than one dimension and we normalize along axis 1. We first compute the mean and variance along this axis and then compute the normalized output, which has the same shape as input, as following:

\[
\text{out} = \frac{\text{data} - \text{mean}(\text{data}, \text{axis})}{\sqrt{\text{var}(\text{data}, \text{axis})}} + \epsilon \times \text{gamma} + \text{beta}
\]

Both “gamma” and “beta” are learnable parameters.

Unlike BatchNorm and InstanceNorm, the *mean* and *var* are computed along the channel dimension.

Assume the input has size *k* on axis 1, then both “gamma” and “beta” have shape *(k,)*. If “output_mean_var” is set to be true, then outputs both “data_mean” and “data_std“. Note that no gradient will be passed through these two outputs.

The parameter “axis” specifies which axis of the input shape denotes the ‘channel’ (separately normalized groups). The default is -1, which sets the channel axis to be the last item in the input shape.

Defined in src/operator/nn/layer_norm.cc:L158

**Value**

- **out**: The result mx.ndarray

---

**mx.nd.LeakyReLU**

Applies Leaky rectified linear unit activation element-wise to the input.

**Description**

Leaky ReLUs attempt to fix the "dying ReLU" problem by allowing a small ‘slope’ when the input is negative and has a slope of one when input is positive.
Arguments

- **data**: NDArray-or-Symbol. Input data to activation function.
- **gamma**: NDArray-or-Symbol. Input data to activation function.
- **act.type**: 'elu', 'gelu', 'leaky', 'prelu', 'rrelu', 'selu'. Optional, default='leaky'. Activation function to be applied.
- **slope**: float, optional, default=0.25. Init slope for the activation. (For leaky and elu only)
- **lower.bound**: float, optional, default=0.125. Lower bound of random slope. (For rrelu only)
- **upper.bound**: float, optional, default=0.333999991. Upper bound of random slope. (For rrelu only)

Details

The following modified ReLU Activation functions are supported:

- *elu*: Exponential Linear Unit. \( y = \begin{cases} x > 0 \ \Rightarrow \ x \cdot \text{slope} \cdot (\exp(x) - 1) \\ x \leq 0 \ \Rightarrow \ 0 \end{cases} \)
- *selu*: Scaled Exponential Linear Unit. \( y = \lambda \cdot \begin{cases} x > 0 \ \Rightarrow \ x \\ x \leq 0 \ \Rightarrow \ \alpha \cdot (\exp(x) - 1) \end{cases} \) where \( \lambda = 1.0507009873554804934193349852946 \) and \( \alpha = 1.6732632423543772848170429916717 \).
- *leaky*: Leaky ReLU. \( y = \begin{cases} x > 0 \ \Rightarrow \ x \\ x \leq 0 \ \Rightarrow \ \text{slope} \cdot x \end{cases} \)
- *prelu*: Parametric ReLU. This is same as *leaky* except that 'slope' is learnt during training.
- *rrelu*: Randomized ReLU. Same as *leaky* but the 'slope' is uniformly and randomly chosen from \([\text{lower bound}, \text{upper bound})\) for training, while fixed to be \((\text{lower bound} + \text{upper bound})/2\) for inference.

Defined in src/operator/leaky_relu.cc:L161

Value

- **out**: The result mx.ndarray

---

**mx.nd.linalg.det**

*Compute the determinant of a matrix. Input is a tensor \( \mathbf{A} \) of dimension \( n = 2 \).*

**Description**

If \( n = 2 \), \( \mathbf{A} \) is a square matrix. We compute:

**Arguments**

- **A**: NDArray-or-Symbol. Tensor of square matrix
Details

\*out\* = \*det(A)\*

If \*n>2\*, \*det\* is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only. .. note:: There is no gradient backwarded when A is non-invertible (which is equivalent to det(A) = 0) because zero is rarely hit upon in float point computation and the Jacobi’s formula on determinant gradient is not computationally efficient when A is non-invertible.

Examples::

Single matrix determinant A = [[1., 4.], [2., 3.]] det(A) = [-5.]
Batch matrix determinant A = [[[1., 4.], [2., 3.]], [[2., 3.], [1., 4.]]] det(A) = [-5., 5.]

Defined in src/operator/tensor/la_op.cc:L975

Value

out The result mx.ndarray

mx.nd.linalg.extractdiag

Extracts the diagonal entries of a square matrix. Input is a tensor \*A* of dimension \*n >= 2\*.

Description

If \*n=2\*, then \*A* represents a single square matrix which diagonal elements get extracted as a 1-dimensional tensor.

Arguments

A NDArray-or-Symbol Tensor of square matrices

offset int, optional, default=’0’ Offset of the diagonal versus the main diagonal. 0 corresponds to the main diagonal, a negative/positive value to diagonals below/above the main diagonal.

Details

If \*n>2\*, then \*A* represents a batch of square matrices on the trailing two dimensions. The extracted diagonals are returned as an \*n-1*-dimensional tensor.

.. note:: The operator supports float32 and float64 data types only.

Examples::

Single matrix diagonal extraction A = [[1.0, 2.0], [3.0, 4.0]]
extractdiag(A) = [1.0, 4.0]
extractdiag(A, 1) = [2.0]
Batch matrix diagonal extraction A = [[[1.0, 2.0], [3.0, 4.0]], [[5.0, 6.0], [7.0, 8.0]]]
extractdiag(A) = [[1.0, 4.0], [5.0, 8.0]]
Defined in src/operator/tensor/la_op.cc:L495

Value

out The result mx.ndarray

mx.nd.linalg.extracttrian
Extracts a triangular sub-matrix from a square matrix. Input is a tensor *A* of dimension *n >= 2*.

Description

If *n=2*, then *A* represents a single square matrix from which a triangular sub-matrix is extracted as a 1-dimensional tensor.

Arguments

A NDArray-or-Symbol Tensor of square matrices
offset int, optional, default='0' Offset of the diagonal versus the main diagonal. 0 corresponds to the main diagonal, a negative/positive value to diagonals below/above the main diagonal.
lower boolean, optional, default=1 Refer to the lower triangular matrix if lower=true, refer to the upper otherwise. Only relevant when offset=0

Details

If *n>2*, then *A* represents a batch of square matrices on the trailing two dimensions. The extracted triangular sub-matrices are returned as an *n-1*-dimensional tensor.

The *offset* and *lower* parameters determine the triangle to be extracted:
- When *offset = 0* either the lower or upper triangle with respect to the main diagonal is extracted depending on the value of parameter *lower*. - When *offset = k > 0* the upper triangle with respect to the k-th diagonal above the main diagonal is extracted. - When *offset = k < 0* the lower triangle with respect to the k-th diagonal below the main diagonal is extracted.

.. note:: The operator supports float32 and float64 data types only.

Examples::
Single triagonal extraction A = [[1.0, 2.0], [3.0, 4.0]]
eextracttrian(A) = [1.0, 3.0, 4.0] eextracttrian(A, lower=False) = [1.0, 2.0, 4.0] eextracttrian(A, 1) = [2.0] eextracttrian(A, -1) = [3.0]
Batch triagonal extraction A = [[[1.0, 2.0], [3.0, 4.0]], [[5.0, 6.0], [7.0, 8.0]]] eextracttrian(A) = [[[1.0, 3.0, 4.0], [5.0, 7.0, 8.0]]]
Definition in src/operator/tensor/la_op.cc:L605
mx.nd.linalg.gelqf

Value

out The result mx.ndarray

---

**mx.nd.linalg.gelqf**

*LQ* factorization for general matrix. Input is a tensor *A* of dimension 
*n >= 2*.

---

**Description**

If *n=2*, we compute the LQ factorization (LAPACK *gelqf*, followed by *orglq*). *A* must have shape *(x, y)* with *x <= y*, and must have full rank *=x*. The LQ factorization consists of *L* with shape *(x, x)* and *Q* with shape *(x, y)*, so that:

**Arguments**

A  
NDArray-or-Symbol Tensor of input matrices to be factorized

**Details**

*A* = *L* \* *Q*

Here, *L* is lower triangular (upper triangle equal to zero) with nonzero diagonal, and *Q* is row-orthonormal, meaning that

*Q* \* *Q*

is equal to the identity matrix of shape *(x, x)*.

If *n>2*, *gelqf* is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

**Examples::**

Single LQ factorization

```
A = [[1., 2., 3.], [4., 5., 6.]]  
Q, L = gelqf(A)  
Q = [[-0.26726124, -0.53452248, -0.80178373], [0.87287156, 0.21821789, -0.43643578]]  
L = [[-3.74165739, 0.], [-8.55235974, 1.96396101]]
```

Batch LQ factorization

```
A = [[[1., 2., 3.], [4., 5., 6.]], [[7., 8., 9.], [10., 11., 12.]]]  
Q, L = gelqf(A)  
Q = [[[[-0.26726124, -0.53452248, -0.80178373], [0.87287156, 0.21821789, -0.43643578]], 
     [0.50257071, -0.57436653, -0.64616234], [0.7620735, 0.05862104, -0.64483142]]]  
L = [[[-3.74165739, 0.], [-13.92838828, 0.], [-19.09768702, 0.52758934]], 
     [1.96396101], [1.96396101]]
```

Defined in src/operator/tensor/la_op.cc:L798

**Value**

out The result mx.ndarray
mx.nd.linalg.gemm

Performs general matrix multiplication and accumulation. Input are tensors \(\mathbf{A}, \mathbf{B}, \mathbf{C}\), each of dimension \(n \geq 2\) and having the same shape on the leading \(n-2 \) dimensions.

Description

If \(n=2\), the BLAS3 function \(\text{gemm}\) is performed:

Arguments

- **A**: NDArray-or-Symbol Tensor of input matrices
- **B**: NDArray-or-Symbol Tensor of input matrices
- **C**: NDArray-or-Symbol Tensor of input matrices
- \(\text{transpose.a}\): boolean, optional, default=0 Multiply with transposed of first input (A).
- \(\text{transpose.b}\): boolean, optional, default=0 Multiply with transposed of second input (B).
- \(\text{alpha}\): double, optional, default=1 Scalar factor multiplied with \(\mathbf{A}\mathbf{B}\).
- \(\text{beta}\): double, optional, default=1 Scalar factor multiplied with \(\mathbf{C}\).
- \(\text{axis}\): int, optional, default=-2 Axis corresponding to the matrix rows.

Details

\[
\text{out} = \alpha \cdot \text{op} (\mathbf{A}) \cdot \text{op} (\mathbf{B}) + \beta \cdot \mathbf{C}
\]

Here, \(\alpha\) and \(\beta\) are scalar parameters, and \(\text{op}\) is either the identity or matrix transposition (depending on \(\text{transpose.a}\), \(\text{transpose.b}\)).

If \(n>2\), \(\text{gemm}\) is performed separately for a batch of matrices. The column indices of the matrices are given by the last dimensions of the tensors, the row indices by the axis specified with the \(\text{axis}\) parameter. By default, the trailing two dimensions will be used for matrix encoding.

For a non-default axis parameter, the operation performed is equivalent to a series of swapaxes/gemmswapaxes calls. For example let \(\mathbf{A}, \mathbf{B}, \mathbf{C}\) be 5 dimensional tensors. Then \(\text{gemm}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \text{axis}=1)\) is equivalent to the following without the overhead of the additional swapaxis operations:

\[
\begin{align*}
\text{A1} &= \text{swapaxes}(\text{A}, \text{dim1}=1, \text{dim2}=3) \\
\text{B1} &= \text{swapaxes}(\text{B}, \text{dim1}=1, \text{dim2}=3) \\
\text{C} &= \text{swapaxes}(\text{C}, \text{dim1}=1, \text{dim2}=3) \\
\text{C} &= \text{gemm}(\text{A1}, \text{B1}, \text{C}) \\
\text{C} &= \text{swapaxis}(\text{C}, \text{dim1}=1, \text{dim2}=3)
\end{align*}
\]

When the input data is of type float32 and the environment variables MXNET_CUDA_ALLOW_TENSOR_CORE and MXNET_CUDA_TENSOR_OP_MATH_ALLOW_CONVERSION are set to 1, this operator will try to use pseudo-float16 precision (float32 math with float16 I/O) precision in order to use Tensor Cores on suitable NVIDIA GPUs. This can sometimes give significant speedups.

Examples:

Single matrix multiply-add \(\mathbf{A} = \begin{bmatrix} 1.0, 1.0 \end{bmatrix}, \begin{bmatrix} 1.0, 1.0 \end{bmatrix} \) \(\mathbf{B} = \begin{bmatrix} 1.0, 1.0, 1.0 \end{bmatrix}, \begin{bmatrix} 1.0, 1.0 \end{bmatrix}, \begin{bmatrix} 1.0, 1.0 \end{bmatrix} \) \(\mathbf{C} = \begin{bmatrix} 1.0, 1.0, 1.0, 1.0, 1.0 \end{bmatrix}, \begin{bmatrix} 1.0, 1.0 \end{bmatrix}, \begin{bmatrix} 1.0, 1.0, 1.0 \end{bmatrix} \) \(\text{gemm}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \text{transpose.b}=\text{True}, \alpha=2.0, \beta=10.0) = \begin{bmatrix} 14.0, 14.0, 14.0, 14.0 \end{bmatrix}, \begin{bmatrix} 14.0, 14.0, 14.0 \end{bmatrix}, \begin{bmatrix} 14.0, 14.0, 14.0 \end{bmatrix} \)

Batch matrix multiply-add $A = \begin{bmatrix} 1.0, 1.0 \\ 0.1, 0.1 \end{bmatrix}$ $B = \begin{bmatrix} 1.0, 1.0 \\ 0.1, 0.1 \end{bmatrix}$ $C = \begin{bmatrix} 10.0, 0.01 \end{bmatrix}$
gemm($A$, $B$, $C$, transpose_b=True, alpha=2.0, beta=10.0) = \begin{bmatrix} 104.0, 0.14 \end{bmatrix}$
Defined in src/operator/tensor/la_op.cc:L89

Value

out The result mx.ndarray

mx.nd.linalg.gemm2

Performs general matrix multiplication. Input are tensors $A^*$, $B^*$, each of dimension $n >= 2$ and having the same shape on the leading $n-2$ dimensions.

Description

If $n=2$, the BLAS3 function $\text{gemm}$ is performed:

Arguments

A NDArray-or-Symbol Tensor of input matrices
B NDArray-or-Symbol Tensor of input matrices
transpose.a boolean, optional, default=0 Multiply with transposed of first input (A).
transpose.b boolean, optional, default=0 Multiply with transposed of second input (B).
alpha double, optional, default=1 Scalar factor multiplied with A*B.
axis int, optional, default='-2' Axis corresponding to the matrix row indices.

details

$out^* = *\text{alpha}^* \cdot *\text{op}^*(A^*) \cdot *\text{op}^*(B^*)$

Here $\text{alpha}^*$ is a scalar parameter and $\text{op}(\cdot)$ is either the identity or the matrix transposition (depending on $\text{transpose.a}^*$, $\text{transpose.b}^*$).

If $n>2$, $\text{gemm}$ is performed separately for a batch of matrices. The column indices of the matrices are given by the last dimensions of the tensors, the row indices by the axis specified with the $\text{axis}^*$ parameter. By default, the trailing two dimensions will be used for matrix encoding.

For a non-default axis parameter, the operation performed is equivalent to a series of swapaxes/gemm/swapaxes calls. For example let $A^*$, $B^*$ be 5 dimensional tensors. Then gemm($A^*$, $B^*$, axis=1) is equivalent to the following without the overhead of the additional swapaxis operations::

$A1 = \text{swapaxes}(A, \text{dim1}=1, \text{dim2}=3)$
$B1 = \text{swapaxes}(B, \text{dim1}=1, \text{dim2}=3)$
$C = \text{gemm2}(A1, B1)$
$C = \text{swapaxis}(C, \text{dim1}=1, \text{dim2}=3)$

When the input data is of type float32 and the environment variables MXNET_CUDA_ALLOW_TENSOR_CORE and MXNET_CUDA_TENSOR_OP_MATH_ALLOW_CONVERSION are set to 1, this operator will try to use pseudo-float16 precision (float32 math with float16 I/O) precision in order to use Tensor Cores on suitable NVIDIA GPUs. This can sometimes give significant speedups.

.. note:: The operator supports float32 and float64 data types only.
Examples:
Single matrix multiply \( A = [[1.0, 1.0], [1.0, 1.0]] \)
\( B = [[1.0, 1.0], [1.0, 1.0], [1.0, 1.0]] \)
gemm2(A, B, transpose_b=True, alpha=2.0) = [[4.0, 4.0, 4.0], [4.0, 4.0, 4.0]]

Batch matrix multiply \( A = [[[1.0, 1.0]], [[0.1, 0.1]]] \)
\( B = [[[1.0, 1.0]], [[0.1, 0.1]]] \)
gemm2(A, B, transpose_b=True, alpha=2.0) = [[[4.0]], [[0.04]]]
Defined in src/operator/tensor/la_op.cc:L163

Value
out The result mx.ndarray

mx.nd.linalg.inverse

Compute the inverse of a matrix. Input is a tensor \( *A* \) of dimension \( *n >= 2* \).

Description
If \( *n=2* \), \( *A* \) is a square matrix. We compute:

Arguments

\( A \) NDArray-or-Symbol Tensor of square matrix

Details
\( *out* = *A*^{-1} \)

If \( *n>2* \), \( \text{inverse} \) is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

Examples:
Single matrix inverse \( A = [[1., 4.], [2., 3.]] \)
inverse(A) = [[-0.6, 0.8], [0.4, -0.2]]

Batch matrix inverse \( A = [[[1., 4.], [2., 3.]], [[1., 3.], [2., 4.]]] \)
inverse(A) = [[[[-0.6, 0.8], [0.4, -0.2]], [[-2., 1.5], [1., -0.5]]]]
Defined in src/operator/tensor/la_op.cc:L920

Value
out The result mx.ndarray
mx.nd.linalg.makediag

Constructs a square matrix with the input as diagonal. Input is a tensor \( A \) of dimension \( n \geq 1 \).

Description

If \( n=1 \), then \( A \) represents the diagonal entries of a single square matrix. This matrix will be returned as a 2-dimensional tensor. If \( n>1 \), then \( A \) represents a batch of diagonals of square matrices. The batch of diagonal matrices will be returned as an \( n+1 \)-dimensional tensor.

Arguments

- \( A \) : NDArray-or-Symbol Tensor of diagonal entries
- \( \text{offset} \) : int, optional, default='0' Offset of the diagonal versus the main diagonal. 0 corresponds to the main diagonal, a negative/positive value to diagonals below/above the main diagonal.

Details

.. note:: The operator supports float32 and float64 data types only.

Examples::

Single diagonal matrix construction

\[
A = \begin{bmatrix} 1.0 & 2.0 \\ \end{bmatrix}
\]

\[
makediag(A) = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 2.0 \\ \end{bmatrix}
\]

\[
makediag(A, 1) = \begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 2.0 \\ 0.0 & 0.0 & 0.0 \\ \end{bmatrix}
\]

Batch diagonal matrix construction

\[
A = \begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ \end{bmatrix}
\]

\[
makediag(A) = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 2.0 \\ \end{bmatrix}
\]

\[
makediag(A, 1) = \begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 2.0 \\ 0.0 & 0.0 & 0.0 \\ \end{bmatrix}
\]

Defined in src/operator/tensor/la_op.cc:L547

Value

- out : The result mx.ndarray

mx.nd.linalg.maketrian

Constructs a square matrix with the input representing a specific triangular sub-matrix. This is basically the inverse of linalg.extracttrian*. Input is a tensor \( A \) of dimension \( n \geq 1 \).

Description

If \( n=1 \), then \( A \) represents the entries of a triangular matrix which is lower triangular if \( \text{offset}<0 \) or \( \text{offset}=0, \text{lower}=\text{true} \). The resulting matrix is derived by first constructing the square matrix with the entries outside the triangle set to zero and then adding \( \text{offset} \)-times an additional diagonal with zero entries to the square matrix.
Arguments

A NDArray-or-Symbol Tensor of triangular matrices stored as vectors

offset int, optional, default='0' Offset of the diagonal versus the main diagonal. 0 corresponds to the main diagonal, a negative/positive value to diagonals below/above the main diagonal.

lower boolean, optional, default=1 Refer to the lower triangular matrix if lower=true, refer to the upper otherwise. Only relevant when offset=0

Details

If *n>1*, then *A* represents a batch of triangular sub-matrices. The batch of corresponding square matrices is returned as an *n+1*-dimensional tensor.

.. note:: The operator supports float32 and float64 data types only.

Examples::

Single matrix construction A = [1.0, 2.0, 3.0]
maketrian(A) = [[1.0, 0.0], [2.0, 3.0]]
maketrian(A, lower=false) = [[1.0, 2.0], [0.0, 3.0]]
maketrian(A, offset=1) = [[0.0, 1.0, 2.0], [0.0, 0.0, 3.0], [0.0, 0.0, 0.0]] maketrian(A, offset=-1) = [[0.0, 0.0, 0.0], [1.0, 0.0, 0.0], [2.0, 3.0, 0.0]]

Batch matrix construction A = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
maketrian(A) = [[[1.0, 2.0], [2.0, 3.0]], [[4.0, 0.0], [5.0, 6.0]]]
maketrian(A, offset=1) = [[[0.0, 1.0, 2.0], [0.0, 0.0, 3.0], [0.0, 0.0, 0.0]], [[0.0, 4.0, 5.0], [0.0, 0.0, 6.0], [0.0, 0.0, 0.0]]]

Defined in src/operator/tensor/la_op.cc:L673

Value

out The result mx.ndarray

mx.nd.linalg.potrf

Perform Cholesky factorization of a symmetric positive-definite matrix. Input is a tensor *A* of dimension *n >= 2*.

Description

If *n=2*, the Cholesky factor *B* of the symmetric, positive definite matrix *A* is computed. *B* is triangular (entries of upper or lower triangle are all zero), has positive diagonal entries, and:

Arguments

A NDArray-or-Symbol Tensor of input matrices to be decomposed
Details

\[A^* = B^* \cdot B^\top \text{ if } \text{lower} = \text{true} \]
\[A^* = B^\top \cdot B \text{ if } \text{lower} = \text{false}\]

If \(n \geq 2\), \text{potrf}\* is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

Examples::

Single matrix factorization \(A = \begin{bmatrix} 4.0, 1.0 \end{bmatrix}, \begin{bmatrix} 1.0, 4.25 \end{bmatrix}\)

\(\text{potrf}(A) = \begin{bmatrix} 2.0, 0 \end{bmatrix}, \begin{bmatrix} 0.5, 2.0 \end{bmatrix}\)

Batch matrix factorization \(A = \begin{bmatrix} \begin{bmatrix} 4.0, 1.0 \end{bmatrix}, \begin{bmatrix} 1.0, 4.25 \end{bmatrix} \end{bmatrix}, \begin{bmatrix} \begin{bmatrix} 16.0, 4.0 \end{bmatrix}, \begin{bmatrix} 4.0, 17.0 \end{bmatrix} \end{bmatrix}\)

\(\text{potrf}(A) = \begin{bmatrix} \begin{bmatrix} 2.0, 0 \end{bmatrix}, \begin{bmatrix} 0.5, 2.0 \end{bmatrix} \end{bmatrix}, \begin{bmatrix} \begin{bmatrix} 4.0, 0 \end{bmatrix}, \begin{bmatrix} 1.0, 4.0 \end{bmatrix} \end{bmatrix}\)

Defined in src/operator/tensor/la_op.cc:L214

Value

out The result mx.ndarray

mx.nd.linalg.potri

Performs matrix inversion from a Cholesky factorization. Input is a tensor \(A^*\) of dimension \(n \geq 2\).

Description

If \(n = 2\), \(A^*\) is a triangular matrix (entries of upper or lower triangle are all zero) with positive diagonal. We compute:

Arguments

\(A\)

NDArray-or-Symbol Tensor of lower triangular matrices

Details

\[\text{out}^* = A^\top \cdot A^\top \text{ if } \text{lower} = \text{true} \]
\[\text{out}^* = A^\top \cdot A \text{ if } \text{lower} = \text{false}\]

In other words, if \(A^*\) is the Cholesky factor of a symmetric positive definite matrix \(B^*\) (obtained by \text{potrf}\*), then

\[\text{out}^* = B^\top \cdot B\]

If \(n > 2\), \text{potri}\* is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: Use this operator only if you are certain you need the inverse of \(B^*\), and cannot use the Cholesky factor \(A^*\) (*potrf*), together with backsubstitution (*trsm*). The latter is numerically much safer, and also cheaper.

Examples::

Single matrix inverse \(A = [[2.0, 0], [0.5, 2.0]]\)

\(\text{potri}(A) = [[0.26563, -0.0625], [-0.0625, 0.25]]\)

Batch matrix inverse \(A = [[[[2.0, 0], [0.5, 2.0]], [[4.0, 0], [1.0, 4.0]]]\text{potri}(A) = [[[0.26563, -0.0625], [-0.0625, 0.25]], [[0.06641, -0.01562], [-0.01562, 0.0625]]\]

Defined in src/operator/tensor/la_op.cc:L275
Value

out The result mx.ndarray

mx.nd.linalg.slogdet Compute the sign and log of the determinant of a matrix. Input is a tensor *A* of dimension *n >= 2*.

Description

If *n=2*, *A* is a square matrix. We compute:

Arguments

A NDArray-or-Symbol Tensor of square matrix

Details

*sign* = *sign(det(A))* *logabsdet* = *log(abs(det(A)))*

If *n>2*, *slogdet* is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only. .. note:: The gradient is not properly defined on sign, so the gradient of it is not backwaded. .. note:: No gradient is backwaded when A is non-invertible. Please see the docs of operator det for detail.

Examples::

Single matrix signed log determinant A = [[2., 3.], [1., 4.]] sign, logabsdet = slogdet(A) sign = [1.] logabsdet = [1.609438]


Defined in src/operator/tensor/la_op.cc:L1034

Value

out The result mx.ndarray
mx.nd.linalg.sumlogdiag

Computes the sum of the logarithms of the diagonal elements of a square matrix. Input is a tensor \( A \) of dimension \( n \geq 2 \).

### Description

If \( n=2 \), \( A \) must be square with positive diagonal entries. We sum the natural logarithms of the diagonal elements, the result has shape (1,).

### Arguments

- **A**: NDArray-or-Symbol Tensor of square matrices

### Details

If \( n>2 \), \text{sumlogdiag} \ is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

Examples::

Single matrix reduction \( A = \begin{bmatrix} 1.0, 1.0 \end{bmatrix}, \begin{bmatrix} 1.0, 7.0 \end{bmatrix} \) \n\text{sumlogdiag}(A) = 1.9459

Batch matrix reduction \( A = \begin{bmatrix} \begin{bmatrix} 1.0, 1.0 \end{bmatrix}, \begin{bmatrix} 1.0, 7.0 \end{bmatrix} \end{bmatrix}, \begin{bmatrix} \begin{bmatrix} 3.0, 0 \end{bmatrix}, \begin{bmatrix} 0, 17.0 \end{bmatrix} \end{bmatrix} \) \n\text{sumlogdiag}(A) = [1.9459, 3.9318]

Defined in src/operator/tensor/la_op.cc:L445

### Value

- **out**: The result mx.ndarray

mx.nd.linalg.syrk

Multiplication of matrix with its transpose. Input is a tensor \( A \) of dimension \( n \geq 2 \).

### Description

If \( n=2 \), the operator performs the BLAS3 function \text{syrk}:

### Arguments

- **A**: NDArray-or-Symbol Tensor of input matrices
- **transpose**: boolean, optional, default=0 Use transpose of input matrix.
- **alpha**: double, optional, default=1 Scalar factor to be applied to the result.
Details

\[ \text{out} = \alpha \cdot A \cdot A^\top \]
if \( \text{transpose}=\text{False} \), or
\[ \text{out} = \alpha \cdot A^\top \cdot A \]
if \( \text{transpose}=\text{True} \).

If \( n>2 \), \text{syrk} \) is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

Examples:

Single matrix multiply

\[ A = \begin{bmatrix} 1., 2., 3. \\ 4., 5., 6. \end{bmatrix} \]
\[ \text{syrk}(A, \alpha=1., \text{transpose}=\text{False}) = \begin{bmatrix} 14., 32. \\ 32., 77. \end{bmatrix} \]
\[ \text{syrk}(A, \alpha=1., \text{transpose}=\text{True}) = \begin{bmatrix} 17., 22., 27. \\ 22., 29., 36. \\ 27., 36., 45. \end{bmatrix} \]

Batch matrix multiply

\[ A = \begin{bmatrix} 1., 1. \\ 0.1, 0.1 \end{bmatrix} \]
\[ \text{syrk}(A, \alpha=2., \text{transpose}=\text{False}) = \begin{bmatrix} 4. \\ 0.04 \end{bmatrix} \]

Defined in src/operator/tensor/la_op.cc:L730

Value

out The result mx.ndarray

**mx.nd.linalg.trmm**

Performs multiplication with a lower triangular matrix. Input are tensors \( A, B \), each of dimension \( n >= 2 \) and having the same shape on the leading \( n-2 \) dimensions.

**Description**

If \( n=2 \), \( A \) must be triangular. The operator performs the BLAS3 function \text{trmm}:

**Arguments**

- **A**: NDArray-or-Symbol Tensor of lower triangular matrices
- **B**: NDArray-or-Symbol Tensor of matrices
- **transpose**: boolean, optional, default=0 Use transposed of the triangular matrix
- **rightside**: boolean, optional, default=0 Multiply triangular matrix from the right to non-triangular one.
- **lower**: boolean, optional, default=1 True if the triangular matrix is lower triangular, false if it is upper triangular.
- **alpha**: double, optional, default=1 Scalar factor to be applied to the result.
Details

\[ *\text{out} = *\alpha \ast \ast \text{op} \ast (\ast \text{A} \ast) \ast \ast \text{B} \ast \]  
if \( *\text{rightside=False} \), or

\[ *\text{out} = *\alpha \ast \ast \text{B} \ast \ast \text{op} \ast (\ast \text{A} \ast) \]  
if \( *\text{rightside=True} \). Here, \( *\alpha \) is a scalar parameter, and \( *\text{op} \ast \) is either the identity or the matrix transposition (depending on \( *\text{transpose} \)).

If \( *n>2 \), \( *\text{trmm} \ast \) is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

Examples::

Single triangular matrix multiply
\[
A = \begin{bmatrix}
1.0 & 0 \\
1.0 & 1.0
\end{bmatrix}
\]
\[
B = \begin{bmatrix}
1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0
\end{bmatrix}
\]
\[
\text{trmm}(A, B, \alpha=2.0) = \begin{bmatrix}
2.0 & 2.0 & 2.0 \\
4.0 & 4.0 & 4.0
\end{bmatrix}
\]

Batch triangular matrix multiply
\[
A = \begin{bmatrix}
[[1.0, 0], [1.0, 1.0]] \\
[[1.0, 0], [1.0, 1.0]]
\end{bmatrix}
\]
\[
B = \begin{bmatrix}
[[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]] \\
[[0.5, 0.5, 0.5], [0.5, 0.5, 0.5]]
\end{bmatrix}
\]
\[
\text{trmm}(A, B, \alpha=2.0) = \begin{bmatrix}
[[2.0, 2.0, 2.0], [4.0, 4.0, 4.0]] \\
[[1.0, 1.0, 1.0], [2.0, 2.0, 2.0]]
\end{bmatrix}
\]

Defined in src/operator/tensor/la_op.cc:L333

Value

out The result mx.ndarray

mx.nd.linalg.trsm

Solves matrix equation involving a lower triangular matrix. Input are tensors \( *\text{A} \ast, *\text{B} \ast \), each of dimension \( *n \geq 2 \ast \) and having the same shape on the leading \( *n-2 \ast \) dimensions.

Description

If \( *n=2 \ast \), \( *\text{A} \ast \) must be triangular. The operator performs the BLAS3 function \( *\text{trsm} \ast \), solving for \( *\text{out} \ast \) in:

Arguments

- **A**
  - NDArray-or-Symbol Tensor of lower triangular matrices
- **B**
  - NDArray-or-Symbol Tensor of matrices
- **transpose**
  - boolean, optional, default=0 Use transposed of the triangular matrix
- **rightside**
  - boolean, optional, default=0 Multiply triangular matrix from the right to non-triangular one.
- **lower**
  - boolean, optional, default=1 True if the triangular matrix is lower triangular, false if it is upper triangular.
- **alpha**
  - double, optional, default=1 Scalar factor to be applied to the result.
**mx.nd.LinearRegressionOutput**

Computes and optimizes for squared loss during backward propagation. Just outputs “data” during forward propagation.

**Description**

If \( \hat{y}_i \) is the predicted value of the \( i \)-th sample, and \( y_i \) is the corresponding target value, then the squared loss estimated over \( n \) samples is defined as

\[
\text{SquaredLoss}(\textbf{Y}, \hat{\textbf{Y}}) = \frac{1}{n} \sum_{i=0}^{n-1} \| \textbf{y}_i - \hat{\textbf{y}}_i \|_2
\]

.. note:: Use the LinearRegressionOutput as the final output layer of a net.

The storage type of “label” can be “default” or “csr”

**Arguments**

- **data**: NDArray-or-Symbol Input data to the function.
- **label**: NDArray-or-Symbol Input label to the function.
- **grad.scale**: float, optional, default=1 Scale the gradient by a float factor

**Details**

The storage type of “label” can be “default” or “csr”
mx.nd.load

Load an mx.nd.array object on disk

Description
Load an mx.nd.array object on disk

Usage
mx.nd.load(filename)

Arguments
filename the filename (including the path)

Examples
mat = mx.nd.array(1:3)
mx.nd.save(mat, 'temp.mat')
mat2 = mx.nd.load('temp.mat')
as.array(mat)
as.array(mat2)

mx.nd.log
Returns element-wise Natural logarithmic value of the input.

Description
The natural logarithm is logarithm in base *e*, so that \( \log\exp(x) = x \)

Arguments
data NDArray-or-Symbol The input array.
mx.nd.log.softmax

Details

The storage type of “log” output is always dense

Defined in src/operator/tensor/elemwise_unary_op_logexp.cc:L77

Value

out The result mx.ndarray

mx.nd.log.softmax Computes the log softmax of the input. This is equivalent to computing softmax followed by log.

Description

Examples:

Arguments

data NDArray-or-Symbol The input array.
axis int, optional, default=’-1’ The axis along which to compute softmax.
temperature double or None, optional, default=None Temperature parameter in softmax
dtype None, ‘float16’, ‘float32’, ‘float64’, optional, default=None DType of the output in case this can’t be inferred. Defaults to the same as input’s dtype if not defined (dtype=None).
use_length boolean or None, optional, default=0 Whether to use the length input as a mask over the data input.

Details

>>> x = mx.nd.array([1, 2, .1]) >>> mx.nd.log_softmax(x).asnumpy() array([-1.41702998, -0.41702995, -2.31702995], dtype=float32)

Value

out The result mx.ndarray
**mx.nd.log10**

*Returns element-wise Base-10 logarithmic value of the input.*

**Description**

“\(10^{\log_{10}(x)} = x\)”

**Arguments**

- `data` (NDArray-or-Symbol) The input array.

**Details**

The storage type of “log10” output is always dense

Defined in `src/operator/tensor/elemwise Unary_op_logexp.cc:L94`

**Value**

- `out` The result mx.ndarray

---

**mx.nd.log1p**

*Returns element-wise “\(\log(1 + x)\)” value of the input.*

**Description**

This function is more accurate than “\(\log(1 + x)\)” for small “\(x\)” so that \(1+x\approx 1\)

**Arguments**

- `data` (NDArray-or-Symbol) The input array.

**Details**

The storage type of “log1p” output depends upon the input storage type:

- log1p(default) = default - log1p(row_sparse) = row_sparse - log1p(csr) = csr

Defined in `src/operator/tensor/elemwise Unary_op_logexp.cc:L199`

**Value**

- `out` The result mx.ndarray
### mx.nd.log2

Returns element-wise Base-2 logarithmic value of the input.

**Description**

\[2^{\log_2(x)} = x\]

**Arguments**

- **data**: NDArray-or-Symbol The input array.

**Details**

The storage type of “log2“ output is always dense

Defined in src/operator/tensor/elemwise_unary_op_logexp.cc:L106

**Value**

- **out**: The result mx.ndarray

---

### mx.nd.logical.not

Returns the result of logical NOT (!) function

**Description**

Example: `logical_not([-2., 0., 1.]) = [0., 1., 0.]`

**Arguments**

- **data**: NDArray-or-Symbol The input array.

**Value**

- **out**: The result mx.ndarray
mx.nd.LogisticRegressionOutput

Applies a logistic function to the input.

Description

The logistic function, also known as the sigmoid function, is computed as $\frac{1}{1+\exp(-\textbf{x})}$.

Arguments

data | NDArray-or-Symbol | Input data to the function.

label | NDArray-or-Symbol | Input label to the function.

grad.scale | float, optional, default=1 | Scale the gradient by a float factor

Details

Commonly, the sigmoid is used to squash the real-valued output of a linear model $w^T x + b$ into the [0,1] range so that it can be interpreted as a probability. It is suitable for binary classification or probability prediction tasks.

.. note:: Use the LogisticRegressionOutput as the final output layer of a net.

The storage type of “label” can be “default” or “csr”

- LogisticRegressionOutput(default, default) = default - LogisticRegressionOutput(default, csr) = default

The loss function used is the Binary Cross Entropy Loss:

$-(y \log(p) + (1 - y) \log(1 - p))$

Where ‘y’ is the ground truth probability of positive outcome for a given example, and ‘p’ the probability predicted by the model. By default, gradients of this loss function are scaled by factor ‘1/m’, where m is the number of regression outputs of a training example. The parameter ‘grad_scale’ can be used to change this scale to ‘grad_scale/m’.

Defined in src/operator/regression_output.cc:L152

Value

out The result mx.ndarray
mx.nd.LRN

Applies local response normalization to the input.

Description

The local response normalization layer performs "lateral inhibition" by normalizing over local input regions.

Arguments

data NDArray-or-Symbol Input data to LRN
alpha float, optional, default=9.99999975e-05 The variance scaling parameter $\alpha$ in the LRN expression.
beta float, optional, default=0.75 The power parameter $\beta$ in the LRN expression.
knorm float, optional, default=2 The parameter $k$ in the LRN expression.
nsize int (non-negative), required normalization window width in elements.

Details

If $a_{x,y}^{i}$ is the activity of a neuron computed by applying kernel $i$ at position $(x, y)$ and then applying the ReLU nonlinearity, the response-normalized activity $b_{x,y}^{i}$ is given by the expression:

$$ b_{x,y}^{i} = \frac{a_{x,y}^{i}}{k + \frac{\alpha n}{\sum_{j=\max(0, i-\frac{n}{2})}^{\min(N-1, i+\frac{n}{2})} (a_{x,y}^{j})^2}}^{\beta} $$

where the sum runs over $n$ "adjacent" kernel maps at the same spatial position, and $N$ is the total number of kernels in the layer.

Defined in src/operator/nn/lrn.cc:L158

Value

out The result mx.ndarray

mx.nd.MAERegressionOutput

Computes mean absolute error of the input.

Description

MAE is a risk metric corresponding to the expected value of the absolute error.
mx.nd.make.loss

Arguments

data NDArray-or-Symbol Input data to the function.
label NDArray-or-Symbol Input label to the function.
grad.scale float, optional, default=1 Scale the gradient by a float factor

Details

If $\hat{y}_i$ is the predicted value of the $i$-th sample, and $y_i$ is the corresponding target value, then the mean absolute error (MAE) estimated over $n$ samples is defined as:

$$\text{MAE}(\textbf{Y}, \hat{\textbf{Y}}) = \frac{1}{n} \sum_{i=0}^{n-1} |y_i - \hat{y}_i|$$

.. note:: Use the MAERegressionOutput as the final output layer of a net.
The storage type of “label” can be “default” or “csr”

- MAERegressionOutput(default, default) = default - MAERegressionOutput(default, csr) = default

By default, gradients of this loss function are scaled by factor $1/m$, where $m$ is the number of regression outputs of a training example. The parameter ‘grad_scale’ can be used to change this scale to ‘grad_scale/m’.

Defined in src/operator/regression_output.cc:L120

Value

out The result mx.ndarray

mx.nd.make.loss Make your own loss function in network construction.

Description

This operator accepts a customized loss function symbol as a terminal loss and the symbol should be an operator with no backward dependency. The output of this function is the gradient of loss with respect to the input data.

Arguments

data NDArray-or-Symbol The input array.

Details

For example, if you are making a cross entropy loss function. Assume “out” is the predicted output and “label” is the true label, then the cross entropy can be defined as:

cross_entropy = label * log(out) + (1 - label) * log(1 - out) loss = make_loss(cross_entropy)

We will need to use “make_loss” when we are creating our own loss function or we want to combine multiple loss functions. Also we may want to stop some variables’ gradients from backpropagation. See more detail in “BlockGrad” or “stop_gradient”.

mx.nd.MakeLoss

The storage type of “make_loss” output depends upon the input storage type:
- make_loss(default) = default - make_loss(row_sparse) = row_sparse

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L358

Value

out The result mx.ndarray

mx.nd.MakeLoss Make your own loss function in network construction.

Description

This operator accepts a customized loss function symbol as a terminal loss and the symbol should be an operator with no backward dependency. The output of this function is the gradient of loss with respect to the input data.

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>NDArray-or-Symbol Input array.</td>
</tr>
<tr>
<td>grad.scale</td>
<td>float, optional, default=1 Gradient scale as a supplement to unary and binary operators</td>
</tr>
<tr>
<td>valid.thresh</td>
<td>float, optional, default=0 clip each element in the array to 0 when it is less than “valid.thresh”. This is used when “normalization” is set to “valid”</td>
</tr>
<tr>
<td>normalization</td>
<td>’batch’, ’null’, ’valid’.optional, default=’null’ If this is set to null, the output gradient will not be normalized. If this is set to batch, the output gradient will be divided by the batch size. If this is set to valid, the output gradient will be divided by the number of valid input elements.</td>
</tr>
</tbody>
</table>

Details

For example, if you are making a cross entropy loss function. Assume “out” is the predicted output and “label” is the true label, then the cross entropy can be defined as:
cross_entropy = label * log(out) + (1 - label) * log(1 - out) loss = MakeLoss(cross_entropy)

We will need to use “MakeLoss” when we are creating our own loss function or we want to combine multiple loss functions. Also we may want to stop some variables’ gradients from backpropagation. See more detail in “BlockGrad” or “stop_gradient”.

In addition, we can give a scale to the loss by setting “grad.scale”, so that the gradient of the loss will be rescaled in the backpropagation.
.. note:: This operator should be used as a Symbol instead of NDArray.

Defined in src/operator/make_loss.cc:L71

Value

out The result mx.ndarray
Computes the max of array elements over given axes.

**Description**

Defined in src/operator/tensor/broadcast_reduce_op.h:L32

**Arguments**

- **data**: NDArray-or-Symbol The input
- **axis**: Shape or None, optional, default=None The axis or axes along which to perform the reduction. The default, ‘axis=’), will compute over all elements into a scalar array with shape ‘(1,)’. If ‘axis’ is int, a reduction is performed on a particular axis. If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple. If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead. Negative values means indexing from right to left.
- **keepdims**: boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.
- **exclude**: boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.

**Value**

- **out**: The result mx.ndarray

mx.nd.max.axis

Computes the max of array elements over given axes.

**Description**

Defined in src/operator/tensor/broadcast_reduce_op.h:L32

**Arguments**

- **data**: NDArray-or-Symbol The input
mx.nd.mean

<table>
<thead>
<tr>
<th>axis</th>
<th>Shape or None, optional, default=None The axis or axes along which to perform the reduction. The default, ‘axis=()’, will compute over all elements into a scalar array with shape ‘(1,)’. If ‘axis’ is int, a reduction is performed on a particular axis. If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple. If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead. Negative values means indexing from right to left.</th>
</tr>
</thead>
<tbody>
<tr>
<td>keepdims</td>
<td>boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.</td>
</tr>
<tr>
<td>exclude</td>
<td>boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.</td>
</tr>
</tbody>
</table>

Value

out The result mx.ndarray

mx.nd.mean Computes the mean of array elements over given axes.

Description

Defined in src/operator/tensor/./broadcast_reduce_op.h:L84

Arguments

data NDArray-or-Symbol The input
axis Shape or None, optional, default=None The axis or axes along which to perform the reduction. The default, ‘axis=()’, will compute over all elements into a scalar array with shape ‘(1,)’. If ‘axis’ is int, a reduction is performed on a particular axis. If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple. If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead. Negative values means indexing from right to left.
keepdims boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.
exclude boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.

Value

out The result mx.ndarray
Computes the min of array elements over given axes.

**Description**

Defined in src/operator/tensor/./broadcast_reduce_op.h:L47

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>NDArray-or-Symbol The input</td>
</tr>
<tr>
<td><strong>axis</strong></td>
<td>Shape or None, optional, default=None The axis or axes along which to perform the reduction. The default, ‘axis=’(), will compute over all elements into a scalar array with shape ‘(1,)’. If ‘axis’ is int, a reduction is performed on a particular axis. If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple. If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead. Negative values means indexing from right to left.</td>
</tr>
<tr>
<td><strong>keepdims</strong></td>
<td>boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.</td>
</tr>
<tr>
<td><strong>exclude</strong></td>
<td>boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.</td>
</tr>
</tbody>
</table>

**Value**

out The result mx.ndarray

---

Computes the min of array elements over given axes.

**Description**

Defined in src/operator/tensor/./broadcast_reduce_op.h:L47

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>NDArray-or-Symbol The input</td>
</tr>
</tbody>
</table>
mx.nd.moments

**axis**
Shape or None, optional, default=None The axis or axes along which to perform the reduction.
The default, ‘axis=()’, will compute over all elements into a scalar array with shape ‘(1,)’.
If ‘axis’ is int, a reduction is performed on a particular axis.
If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple.
If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead.
Negative values means indexing from right to left.

**keepdims**
boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.

**exclude**
boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.

**Value**
out The result mx.ndarray

---

**mx.nd.moments**
*Calculate the mean and variance of ‘data’.*

**Description**
The mean and variance are calculated by aggregating the contents of data across axes. If x is 1-D and axes = [0] this is just the mean and variance of a vector.

**Arguments**

data
NDArray-or-Symbol Input ndarray

axes
Shape or None, optional, default=None Array of ints. Axes along which to compute mean and variance.

keepdims
boolean, optional, default=0 produce moments with the same dimensionality as the input.

**Details**
Example:

```python
data = [[1, 2, 3], [4, 5, 6]]
mean = moments(data=data, axes=[0])
mean = [2.5, 3.5, 4.5]
var = moments(data=data, axes=[1])
var = [0.66666667, 0.66666667]
```

**Value**
out The result mx.ndarray
Mixed Precision version of Phase 1 of lamb update it performs the following operations and returns g:

**Description**


**Arguments**

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **mean**: NDArray-or-Symbol Moving mean
- **var**: NDArray-or-Symbol Moving variance
- **weight32**: NDArray-or-Symbol Weight32
- **beta1**: float, optional, default=0.899999976 The decay rate for the 1st moment estimates.
- **beta2**: float, optional, default=0.999000013 The decay rate for the 2nd moment estimates.
- **epsilon**: float, optional, default=9.99999997e-07 A small constant for numerical stability.
- **t**: int, required Index update count.
- **bias.correction**: boolean, optional, default=1 Whether to use bias correction.
- **wd**: float, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient]

If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).

**Details**

.. math:: \begin{gather*} grad32 = grad(float16) * rescale\_grad if (grad < -clip\_gradient) then grad = -clip\_gradient if (grad > clip\_gradient) then grad = clip\_gradient \\
mean = beta1 * mean + (1 - beta1) * grad; variance = beta2 * variance + (1 - beta2) * grad^2; \\
if (bias\_correction) then mean\_hat = mean / (1 - beta1^t); var\_hat = var / (1 - beta2^t); g = mean\_hat / (var\_hat^t + epsilon) + wd * weight32; else g = mean / (var\_data^t + epsilon) + wd * weight32; \end{gather*}

Defined in src/operator/optimizer_op.cc:L1024

**Value**

out The result mx.ndarray
Mixed Precision version Phase II of lamb update it performs the following operations and updates grad.

**Description**


**Arguments**

- **weight**: NDArray-or-Symbol Weight
- **g**: NDArray-or-Symbol Output of mp_lamb_update_phase 1
- **r1**: NDArray-or-Symbol r1
- **r2**: NDArray-or-Symbol r2
- **weight32**: NDArray-or-Symbol Weight32
- **lr**: float, required Learning rate
- **lower_bound**: float, optional, default=-1 Lower limit of norm of weight. If lower_bound <= 0, Lower limit is not set
- **upper_bound**: float, optional, default=-1 Upper limit of norm of weight. If upper_bound <= 0, Upper limit is not set

**Details**

```
.. math:: \begin{align*}
    &\text{if (lower_bound} \geq 0 \text{) then } r1 = \max(r1, \text{lower_bound}) \text{ if (upper_bound} \geq 0 \text{) then } r1 = \max(r1, \text{upper_bound}) \text{ }
    \text{if (r1} = 0 \text{ or r2} = 0 \text{) then } lr = lr \text{ else } lr = lr \times (r1/r2) \text{ weight32} = weight32 - lr \times g \text{ weight(float16)} \text{ }
    = weight32 \end{align*}
```

Defined in src/operator/optimizer_op.cc:L1066

**Value**

```
out The result mx.ndarray
```
mx.nd.mp.nag.mom.update

Update function for multi-precision Nesterov Accelerated Gradient (NAG) optimizer.

Description

Defined in src/operator/optimizer_op.cc:L736

Arguments

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **mom**: NDArray-or-Symbol Momentum
- **weight32**: NDArray-or-Symbol Weight32
- **lr**: float, required Learning rate
- **momentum**: float, optional, default=0 The decay rate of momentum estimates at each epoch.
- **wd**: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).

Value

- **out**: The result mx.ndarray

mx.nd.mp.sgd.mom.update

Updater function for multi-precision sgd optimizer

Description

Updater function for multi-precision sgd optimizer
### Arguments

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **mom**: NDArray-or-Symbol Momentum
- **weight32**: NDArray-or-Symbol Weight32
- **lr**: float, required Learning rate
- **momentum**: float, optional, default=0 The decay rate of momentum estimates at each epoch.
- **wd**: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- **lazy.update**: boolean, optional, default=1 If true, lazy updates are applied if gradient’s stype is row_sparse and both weight and momentum have the same stype.

### Value

- **out**: The result mx.ndarray

### Description

Updater function for multi-precision sgd optimizer

### Arguments

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol gradient
- **weight32**: NDArray-or-Symbol Weight32
- **lr**: float, required Learning rate
- **wd**: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- **lazy.update**: boolean, optional, default=1 If true, lazy updates are applied if gradient’s stype is row_sparse.
mx.nd.multi.all.finite

Check if all the float numbers in all the arrays are finite (used for AMP)

Value
out The result mx.ndarray

mx.nd.multi.all.finite

Description
Defined in src/operator/contrib/all_finite.cc:L133

Arguments
- data NDArray-or-Symbol[] Arrays
- num.arrays int, optional, default='1' Number of arrays.
- init.output boolean, optional, default=1 Initialize output to 1.

Value
out The result mx.ndarray

mx.nd.multi.lars

Compute the LARS coefficients of multiple weights and grads from their sums of square

Description
Defined in src/operator/contrib/multi_lars.cc:L37

Arguments
- lrs NDArray-or-Symbol Learning rates to scale by LARS coefficient
- weights.sum.sq NDArray-or-Symbol sum of square of weights arrays
- grads.sum.sq NDArray-or-Symbol sum of square of gradients arrays
- wds NDArray-or-Symbol weight decays
- eta float, required LARS eta
- eps float, required LARS eps
- rescale.grad float, optional, default=1 Gradient rescaling factor

Value
out The result mx.ndarray
Momentum update function for multi-precision Stochastic Gradient Descent (SGD) optimizer.

Description

Momentum update has better convergence rates on neural networks. Mathematically it looks like below:

Arguments

data NDArray-or-Symbol[] Weights
lrs tuple of <float>, required Learning rates.
wds tuple of <float>, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
momentum float, optional, default=0 The decay rate of momentum estimates at each epoch.
rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
num.weights int, optional, default=’1’ Number of updated weights.

Details

.. math::

\begin{align*}
v_1 &= \alpha \nabla J(W_0) \\
v_t &= \gamma v_{t-1} - \alpha \nabla J(W_{t-1}) \quad W_t = W_{t-1} + v_t
\end{align*}

It updates the weights using::

\begin{align*}
v &= \text{momentum} \times v - \text{learning_rate} \times \text{gradient weight} \\
\text{weight} &= \text{weight} + v
\end{align*}

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.
Defined in src/operator/optimizer_op.cc:L463

Value

out The result mx.ndarray
**mx.nd.multi.mp.sgd.update**

*Update function for multi-precision Stochastic Gradient Descent (SDG) optimizer.*

**Description**

It updates the weights using::

**Arguments**

- **data**: NDArray-or-Symbol[] Weights
- **lrs**: tuple of <float>, required Learning rates.
- **wds**: tuple of <float>, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- **num.weights**: int, optional, default='1’ Number of updated weights.

**Details**

weight = weight - learning_rate * (gradient + wd * weight)

Defined in src/operator/optimizer_op.cc:L408

**Value**

- out The result mx.ndarray

---

**mx.nd.multi.sgd.mom.update**

*Momentum update function for Stochastic Gradient Descent (SGD) optimizer.*

**Description**

Momentum update has better convergence rates on neural networks. Mathematically it looks like below:
Arguments

- **data**: NDArray-or-Symbol[] Weights, gradients and momentum
- **lrs**: tuple of <float>, required Learning rates.
- **wds**: tuple of <float>, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **momentum**: float, optional, default=0 The decay rate of momentum estimates at each epoch.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- **num.weights**: int, optional, default=’1’ Number of updated weights.

Details

.. math::
   v_1 = \alpha * \nabla J(W_0)\ v_t = \gamma v_{t-1} - \alpha * \nabla J(W_{t-1})\ W_t = W_{t-1} + v_t

It updates the weights using::

v = momentum * v - learning_rate * gradient weight += v

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.

Defined in src/operator/optimizer_op.cc:L365

Value

- **out**: The result mx.ndarray

---

**mx.nd.multi.sgd.update**

*Update function for Stochastic Gradient Descent (SDG) optimizer.*

---

Description

It updates the weights using::

Arguments

- **data**: NDArray-or-Symbol[] Weights
- **lrs**: tuple of <float>, required Learning rates.
- **wds**: tuple of <float>, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
**mx.nd.multi.sum.sq**

Compute the sums of squares of multiple arrays

**Description**

Defined in src/operator/contrib/multi_sum_sq.cc:L36

**Arguments**

- **data**
  - NDArray-or-Symbol[] Arrays
- **num.arrays**
  - int, required number of input arrays.

**Value**

- out The result mx.ndarray

---

**mx.nd.nag.mom.update**

Update function for Nesterov Accelerated Gradient (NAG) optimizer. It updates the weights using the following formula,

.. math:: v_t = \gamma v_{t-1} + \eta \nabla J(W_{t-1} - \gamma v_{t-1}) \quad W_t = W_{t-1} - v_t

**Description**

.. math:: v_t = \gamma v_{t-1} + \eta \nabla J(W_{t-1} - \gamma v_{t-1}) \quad W_t = W_{t-1} - v_t

**Arguments**

- **data**
  - NDArray-or-Symbol[] Arrays
- **num.arrays**
  - int, required number of input arrays.

**Value**

- out The result mx.ndarray
Arguments

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **mom**: NDArray-or-Symbol Momentum
- **lr**: float, required Learning rate
- **momentum**: float, optional, default=0 The decay rate of momentum estimates at each epoch.
- **wd**: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to \( \text{grad} = \text{rescale\_grad} \times \text{grad} \).
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of \([-\text{clip\_gradient}, \text{clip\_gradient}]\)
  If \(\text{clip\_gradient} \leq 0\), gradient clipping is turned off. \(\text{grad} = \max(\min(\text{grad}, \text{clip\_gradient}), -\text{clip\_gradient})\).

Details

Where \(\eta\) is the learning rate of the optimizer \(\gamma\) is the decay rate of the momentum estimate \(v_t\) is the update vector at time step \(t\) \(w_t\) is the weight vector at time step \(t\)

Defined in src/operator/optimizer_op.cc:L717

Value

out The result mx.ndarray

mx.nd.nanprod

**Computes the product of array elements over given axes treating Not a Numbers ("NaN") as one.**

Description

Computes the product of array elements over given axes treating Not a Numbers ("NaN") as one.

Arguments

- **data**: NDArray-or-Symbol The input
- **axis**: Shape or None, optional, default=None The axis or axes along which to perform the reduction.
  The default, ‘axis=()’, will compute over all elements into a scalar array with shape ‘(1,)’.
  If ‘axis’ is int, a reduction is performed on a particular axis.
  If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple.
  If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead.
  Negative values means indexing from right to left.
Computes the sum of array elements over given axes treating Not a Numbers (“NaN”) as zero.

**Arguments**
- **data**
  - NDArray-or-Symbol The input
- **axis**
  - Shape or None, optional, default=None The axis or axes along which to perform the reduction.
  - The default, ‘axis=()’, will compute over all elements into a scalar array with shape ‘(1,)’.
  - If ‘axis’ is int, a reduction is performed on a particular axis.
  - If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple.
  - If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead.
  - Negative values means indexing from right to left.
- **keepdims**
  - boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.
- **exclude**
  - boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.

**Details**
- Defined in src/operator/tensor/broadcast_reduce_sum_value.cc:L102

**Value**
- out The result mx.ndarray
**mx.nd.negative**

**Numerical negative of the argument, element-wise.**

**Description**

The storage type of “negative” output depends upon the input storage type:

**Arguments**

- **data**: NDArray-or-Symbol The input array.

**Details**

- negative(default) = default - negative(row_sparse) = row_sparse - negative(csr) = csr

**Value**

- **out**: The result mx.ndarray

---

**mx.nd.norm**

**Computes the norm on an NDArray.**

**Description**

This operator computes the norm on an NDArray with the specified axis, depending on the value of the ord parameter. By default, it computes the L2 norm on the entire array. Currently only ord=2 supports sparse ndarrays.

**Arguments**

- **data**: NDArray-or-Symbol The input
- **ord**: int, optional, default=’2’ Order of the norm. Currently ord=1 and ord=2 is supported.
- **axis**: Shape or None, optional, default=None The axis or axes along which to perform the reduction. The default, ‘axis=()’, will compute over all elements into a scalar array with shape ‘(1,)’. If ‘axis’ is int, a reduction is performed on a particular axis. If ‘axis’ is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed.
- **keepdims**: boolean, optional, default=0 If this is set to ‘True’, the reduced axis is left in the result as dimension with size one.
mx.nd.normal

Details

Examples:

```python
x = [[[1, 2], [3, 4]], [[2, 2], [5, 6]]
norm(x, ord=2, axis=1) = [[3.1622777 4.472136 ] [5.3851647 6.3245554]]
norm(x, ord=1, axis=1) = [[4., 6.], [7., 8.]]
rsp = x.cast_storage('row_sparse')
norm(rsp) = [5.47722578]
csr = x.cast_storage('csr')
norm(csr) = [5.47722578]
```

Defined in `src/operator/tensor/broadcast_reduce_norm_value.cc:L89`

Value

[out] The result mx.ndarray

mx.nd.normal

Description

.. note:: The existing alias “normal“ is deprecated.

Arguments

- **loc** float, optional, default=0 Mean of the distribution.
- **scale** float, optional, default=1 Standard deviation of the distribution.
- **shape** Shape(tuple), optional, default=None Shape of the output.
- **ctx** string, optional, default=”Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- **dtype** 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

Details

Samples are distributed according to a normal distribution parametrized by *loc* (mean) and *scale* (standard deviation).

Example:

```python
normal(loc=0, scale=1, shape=(2,2)) = [[ 1.89171135, -1.16881478], [-1.23474145, 1.55807114]]
```

Defined in `src/operator/random/sample_op.cc:L113`

Value

[out] The result mx.ndarray
mx.nd.one.hot

Returns a one-hot array.

Description

The locations represented by ‘indices’ take value ‘on_value’, while all other locations take value ‘off_value’.

Arguments

- **indices**: NDArray-or-Symbol array of locations where to set on_value
- **depth**: int, required Depth of the one hot dimension.
- **on_value**: double, optional, default=1 The value assigned to the locations represented by indices.
- **off_value**: double, optional, default=0 The value assigned to the locations not represented by indices.
- **dtype**: 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', optional, default='float32' DType of the output

Details

‘one_hot’ operation with ‘indices’ of shape “(i0, i1)” and ‘depth’ of “d” would result in an output array of shape “(i0, i1, d)” with:

\[
\text{output}[i,j,:] = \text{off\_value} \quad \text{output}[i,j,\text{indices}[i,j]] = \text{on\_value}
\]

Examples:

- `one_hot([[1,0,2,0], 3) = [[ 0. 1. 0.] [ 1. 0. 0.] [ 0. 0. 1.] [ 1. 0. 0.]]`
- `one_hot([[1,0,2,0], 3, on_value=8, off_value=1, dtype='int32') = [[1 8 1] [8 1 1] [1 1 8] [8 1 1]]`
- `one_hot([[1,0],[1,0],[2,0]], 3) = [[ 0. 1. 0.] [ 0. 0. 1.]]`

Defined in src/operator/tensor/indexing_op.cc:L888

Value

- **out**: The result mx.ndarray
mx.nd.ones

Generate an mx.ndarray object with ones

Description

Generate an mx.ndarray object with ones

Usage

mx.nd.ones(shape, ctx = NULL)

Arguments

shape the dimension of the mx.ndarray
ctx optional The context device of the array. mx.ctx.default() will be used in default.

Examples

mat = mx.nd.ones(10)
as.array(mat)
mat2 = mx.nd.ones(c(5,5))
as.array(mat)
mat3 = mx.nd.ones(c(3,3,3))
as.array(mat3)

mx.nd.ones.like

Return an array of ones with the same shape and type as the input array.

Description

Examples::

Arguments

data NDArray-or-Symbol The input

Details

x = [[ 0., 0., 0.], [ 0., 0., 0.]]
one_like(x) = [[ 1., 1., 1.], [ 1., 1., 1.]]

Value

out The result mx.ndarray
mx.nd.Pad

Pads an input array with a constant or edge values of the array.

Description

.. note:: ‘Pad’ is deprecated. Use ‘pad’ instead.

Arguments

data  NDArray-or-Symbol An n-dimensional input array.
mode 'constant', 'edge', 'reflect', required Padding type to use. "constant" pads with 'constant_value' "edge" pads using the edge values of the input array "reflect" pads by reflecting values with respect to the edges.
pad.width Shape(tuple), required Widths of the padding regions applied to the edges of each axis. It is a tuple of integer padding widths for each axis of the format “(before_1, after_1, ... , before_N, after_N)”. It should be of length “2*N” where “N” is the number of dimensions of the array. This is equivalent to pad_width in numpy.pad, but flattened.
constant.value double, optional, default=0 The value used for padding when ‘mode’ is "constant".

Details

.. note:: Current implementation only supports 4D and 5D input arrays with padding applied only on axes 1, 2 and 3. Expects axes 4 and 5 in ‘pad_width’ to be zero.

This operation pads an input array with either a ‘constant_value’ or edge values along each axis of the input array. The amount of padding is specified by ‘pad_width’.

‘pad_width’ is a tuple of integer padding widths for each axis of the format “(before_1, after_1, ... , before_N, after_N)”. The ‘pad_width’ should be of length “2*N” where “N” is the number of dimensions of the array.

For dimension “N” of the input array, “before_N” and “after_N” indicates how many values to add before and after the elements of the array along dimension “N”. The widths of the higher two dimensions “before_1”, “after_1”, “before_2”, “after_2” must be 0.

Example::

x = [[[ 1. 2. 3.] [ 4. 5. 6.]]
   [[ 7. 8. 9.] [10. 11. 12.]]
   [[[11. 12. 13.] [14. 15. 16.]]
   [[17. 18. 19.] [20. 21. 22.]]]
pad(x,mode="edge", pad_width=(0,0,0,1,1,1)) =
   [[[ 1. 1. 2. 3. 3. 1. 1. 2. 3. 3.] [ 4. 4. 5. 6. 6.] [ 4. 4. 5. 6. 6.]]
   [[ 7. 7. 8. 9. 9. 7. 7. 8. 9. 9.] [10. 10. 11. 12. 12.] [10. 10. 11. 12. 12.]]
mx.nd.pad

Pads an input array with a constant or edge values of the array.

Description

.. note:: 'Pad' is deprecated. Use 'pad' instead.

Arguments

data       NDArray-or-Symbol An n-dimensional input array.
mode       'constant', 'edge', 'reflect', required Padding type to use. "constant" pads with 'constant_value' "edge" pads using the edge values of the input array "reflect" pads by reflecting values with respect to the edges.

pad.width  Shape(tuple), required Widths of the padding regions applied to the edges of each axis. It is a tuple of integer padding widths for each axis of the format "(before_1, after_1, ... , before_N, after_N)". It should be of length "2*N" where "N" is the number of dimensions of the array.This is equivalent to pad_width in numpy.pad, but flattened.
constant.value  double, optional, default=0 The value used for padding when 'mode' is "constant".

Details

.. note:: Current implementation only supports 4D and 5D input arrays with padding applied only on axes 1, 2 and 3. Expects axes 4 and 5 in 'pad_width' to be zero.

This operation pads an input array with either a 'constant_value' or edge values along each axis of the input array. The amount of padding is specified by 'pad_width'.

'pad_width' is a tuple of integer padding widths for each axis of the format "(before_1, after_1, ... , before_N, after_N)". The 'pad_width' should be of length "2*N" where "N" is the number of dimensions of the array.
For dimension “N” of the input array, “before_N” and “after_N” indicates how many values to add before and after the elements of the array along dimension “N“. The widths of the higher two dimensions “before_1”, “after_1”, “before_2”, “after_2” must be 0.

Example::

```python
x = [[[ 1.  2.  3. ]  [ 4.  5.  6.]]
     [[ 7.  8.  9. ]  [10. 11. 12.]]
     [[11. 12. 13. ] [14. 15. 16.]]
     [[17. 18. 19. ] [20. 21. 22.]]]
pad(x,mode="edge", pad_width=(0,0,0,1,1,1,1,1)) =
 [[[ 1.  1.  2.  3.  3. ] [ 1.  1.  2.  3.  3. ] [ 4.  4.  5.  6.  6. ] [ 4.  4.  5.  6.  6. ]]
   [ 7.  7.  8.  9.  9. ] [ 7.  7.  8.  9.  9. ] [10. 10. 11. 12. 12. ] [10. 10. 11. 12. 12. ]]
   [[17. 17. 18. 19. 19. ] [17. 17. 18. 19. 19. ] [20. 20. 21. 22. 22. ] [20. 20. 21. 22. 22. ]]
```

```
pad(x, mode="constant", constant_value=0, pad_width=(0,0,0,1,1,1,1,1)) =
 [[[ 0.  0.  0.  0.  0. ] [ 0.  1.  2.  3.  0. ] [ 0.  4.  5.  6.  0. ] [ 0.  0.  0.  0.  0. ]]
   [ 0.  0.  0.  0.  0. ] [ 0.  7.  8.  9.  0. ] [ 0. 10. 11. 12. 0. ] [ 0.  0.  0.  0.  0. ]]
   [[ 0.  0.  0.  0.  0. ] [ 0. 11. 12. 13. 0. ] [ 0. 14. 15. 16. 0. ] [ 0.  0.  0.  0.  0. ]]
   [[ 0.  0.  0.  0.  0. ] [ 0. 17. 18. 19. 0. ] [ 0. 20. 21. 22. 0. ] [ 0.  0.  0.  0.  0. ]]]
```

Defined in src/operator/pad.cc:L766

Value

out The result mx.ndarray

mx.nd.pick

Picks elements from an input array according to the input indices along the given axis.

Description

Given an input array of shape “(d0, d1)\" and indices of shape “(i0,)\", the result will be an output array of shape “(i0,)\" with::

Arguments

data NDArray-or-Symbol The input array
index NDArray-or-Symbol The index array
axis int or None, optional, default=`-1` int or None. The axis to picking the elements. Negative values means indexing from right to left. If is ‘None’, the elements in the index w.r.t the flattened input will be picked.
keepdims boolean, optional, default=0 If true, the axis where we pick the elements is left in the result as dimension with size one.

mode 'clip', 'wrap', optional, default='clip' Specify how out-of-bound indices behave. Default is "clip". "clip" means clip to the range. So, if all indices mentioned are too large, they are replaced by the index that addresses the last element along an axis. "wrap" means to wrap around.

Details

output[i] = input[i, indices[i]]

By default, if any index mentioned is too large, it is replaced by the index that addresses the last element along an axis (the ‘clip’ mode).

This function supports n-dimensional input and (n-1)-dimensional indices arrays.

Examples::

x = [[ 1., 2.], [ 3., 4.], [ 5., 6.]]
// picks elements with specified indices along axis 0 pick(x, y=[0,1], 0) = [ 1., 4.]
// picks elements with specified indices along axis 1 pick(x, y=[0,1,0], 1) = [ 1., 4., 5.]
// picks elements with specified indices along axis 1 using 'wrap' mode // to place indices that would normally be out of bounds pick(x, y=[2,-1,-2], 1, mode='wrap') = [ 1., 4., 5.]
y = [[ 1.], [ 0.], [ 2.]]
// picks elements with specified indices along axis 1 and dims are maintained pick(x, y, 1, keepdims=True) = [[2.], [3.], [6.]]

Defined in src/operator/tensor/broadcast_reduce_op_index.cc:L151

Value

out The result mx.ndarray

mx.nd.Pooling

**Performs pooling on the input.**

Description

The shapes for 1-D pooling are

Arguments

data NDArray-or-Symbol Input data to the pooling operator.

kernel Shape(tuple), optional, default=[] Pooling kernel size: (y, x) or (d, y, x)

pool.type ‘avg’, ‘lp’, ‘max’, ‘sum’, optional, default=‘max’ Pooling type to be applied.

global.pool boolean, optional, default=0 Ignore kernel size, do global pooling based on current input feature map.
**mx.nd.Pooling**

- **data** and **out**: *(batch_size, channel, width)* (NCW layout) or *(batch_size, width, channel)* (NWC layout),

  The shapes for 2-D pooling are

- **data** and **out**: *(batch_size, channel, height, width)* (NCHW layout) or *(batch_size, height, width, channel)* (NHWC layout),

  out_height = f(height, kernel[0], pad[0], stride[0])
  out_width = f(width, kernel[1], pad[1], stride[1])

  The definition of f depends on “pooling_convention”, which has two options:

- **valid** (default):$
  f(x, k, p, s) = floor((x+2*p-k)/s)+1$

- **full**, which is compatible with Caffe::$
  f(x, k, p, s) = ceil((x+2*p-k)/s)+1$

  When “global_pool” is set to be true, then global pooling is performed. It will reset “kernel=(height, width)” and set the appropriate padding to 0.

  Three pooling options are supported by “pool_type”:  

- **avg**: average pooling - **max**: max pooling - **sum**: sum pooling - **lp**: Lp pooling

  For 3-D pooling, an additional *depth* dimension is added before *height*. Namely the input data and output will have shape *(batch_size, channel, depth, height, width)* (NCDHW layout) or *(batch_size, depth, height, width, channel)* (NDHWC layout).

  Notes on Lp pooling:
Lp pooling was first introduced by this paper: https://arxiv.org/pdf/1204.3968.pdf. L-1 pooling is simply sum pooling, while L-inf pooling is simply max pooling. We can see that Lp pooling stands between those two, in practice the most common value for p is 2.

For each window “X”, the mathematical expression for Lp pooling is:

\[ f(X) = \sqrt[p]{\sum_x^X x^p} \]

Defined in src/operator/nn/pooling.cc:L416

**Value**

out The result mx.ndarray

mx.nd.Pooling.v1 This operator is DEPRECATED. Perform pooling on the input.

**Description**

The shapes for 2-D pooling is

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>NDArray-or-Symbol Input data to the pooling operator.</td>
</tr>
<tr>
<td>kernel</td>
<td>Shape(tuple), optional, default=[] pooling kernel size: (y, x) or (d, y, x)</td>
</tr>
<tr>
<td>pool.type</td>
<td>'avg', 'max', 'sum',optional, default='max' Pooling type to be applied.</td>
</tr>
<tr>
<td>global.pool</td>
<td>boolean, optional, default=0 Ignore kernel size, do global pooling based on current input feature map.</td>
</tr>
<tr>
<td>pooling.convention</td>
<td>'full', 'valid',optional, default='valid' Pooling convention to be applied.</td>
</tr>
<tr>
<td>stride</td>
<td>Shape(tuple), optional, default=[] stride: for pooling (y, x) or (d, y, x)</td>
</tr>
<tr>
<td>pad</td>
<td>Shape(tuple), optional, default=[] pad for pooling: (y, x) or (d, y, x)</td>
</tr>
</tbody>
</table>

**Details**

- **data**: *(batch_size, channel, height, width)* - **out**: *(batch_size, num_filter, out_height, out_width)*, with:

out_height = f(height, kernel[0], pad[0], stride[0]) out_width = f(width, kernel[1], pad[1], stride[1])

The definition of *f* depends on "pooling_convention", which has two options:

- **valid** (default):

  f(x, k, p, s) = floor((x+2*p-k)/s)+1

- **full**, which is compatible with Caffe:

  f(x, k, p, s) = ceil((x+2*p-k)/s)+1

But "global_pool" is set to be true, then do a global pooling, namely reset "kernel=(height, width)". Three pooling options are supported by "pool_type":
- **avg**: average pooling  - **max**: max pooling  - **sum**: sum pooling

1-D pooling is a special case of 2-D pooling with *weight=1* and *kernel[1]=1*.

For 3-D pooling, an additional *depth* dimension is added before *height*. Namely the input data will have shape *(batch_size, channel, depth, height, width)*.

Defined in src/operator/pooling_v1.cc:L104

### Value

out The result mx.ndarray

---

mx.nd.preloaded.multi.mp.sgd.mom.update

*Momentum update function for multi-precision Stochastic Gradient Descent (SGD) optimizer.*

---

### Description

Momentum update has better convergence rates on neural networks. Mathematically it looks like below:

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>NDArray-or-Symbol[] Weights, gradients, momentums, learning rates and weight decays</td>
</tr>
<tr>
<td>momentum</td>
<td>float, optional, default=0 The decay rate of momentum estimates at each epoch.</td>
</tr>
<tr>
<td>rescale.grad</td>
<td>float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.</td>
</tr>
<tr>
<td>clip.gradient</td>
<td>float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient]</td>
</tr>
<tr>
<td>num.weights</td>
<td>int, optional, default='1' Number of updated weights.</td>
</tr>
</tbody>
</table>

### Details

Math:

\[
v_{t} = \gamma v_{t-1} - \alpha \nabla J(W_{t-1})
W_{t} = W_{t-1} + v_{t}
\]

It updates the weights using:

\[
v = \text{momentum} \cdot v - \text{learning rate} \cdot \text{gradient weight} + \gamma
\]

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.

Defined in src/operator/contrib/preloaded_multi_sgd.cc:L200

### Value

out The result mx.ndarray
Update function for multi-precision Stochastic Gradient Descent (SDG) optimizer.

Description

It updates the weights using::

Arguments

data NDArray-or-Symbol[] Weights, gradients, learning rates and weight decays
rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip.gradient, clip.gradient]
  If clip.gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip.gradient), -clip.gradient).
num.weights int, optional, default=’1’ Number of updated weights.

Details

weight = weight - learning_rate * (gradient + wd * weight)

Defined in src/operator/contrib/preloaded_multi_sgd.cc:L140

Value

out The result mx.ndarray

Momentum update function for Stochastic Gradient Descent (SGD) optimizer.

Description

Momentum update has better convergence rates on neural networks. Mathematically it looks like below:
**Arguments**

- **data**: NDArray-or-Symbol[] Weights, gradients, momentum, learning rates and weight decays.
- **momentum**: float, optional, default=0 The decay rate of momentum estimates at each epoch.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- **num.weights**: int, optional, default='1' Number of updated weights.

**Details**

.. math::

\begin{align*}
  v_1 &= \alpha \nabla J(W_0) \\
  v_t &= \gamma v_{t-1} - \alpha \nabla J(W_{t-1}) \\
  W_t &= W_{t-1} + v_t
\end{align*}

It updates the weights using::

\begin{align*}
  v &= \text{momentum} \cdot v \cdot \text{learning_rate} \cdot \text{gradient weight} + = v
\end{align*}

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.

Defined in src/operator/contrib/preloaded_multi_sgd.cc:L91

**Value**

- **out**: The result mx.ndarray

---

**mx.nd.preloaded.multi.sgd.update**

*Update function for Stochastic Gradient Descent (SDG) optimizer.*

**Description**

It updates the weights using::

**Arguments**

- **data**: NDArray-or-Symbol[] Weights, gradients, learning rates and weight decays
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- **num.weights**: int, optional, default='1' Number of updated weights.

**Details**

weight = weight - learning_rate * (gradient + wd * weight)

Defined in src/operator/contrib/preloaded_multi_sgd.cc:L42
mx.nd.prod

**Value**

out The result mx.ndarray

---

**mx.nd.prod**

*Computes the product of array elements over given axes.*

**Description**

Defined in src/operator/tensor/./broadcast_reduce_op.h:L31

**Arguments**

- **data**
  
  NDArray-or-Symbol The input

- **axis**
  
  Shape or None, optional, default=None The axis or axes along which to perform the reduction.
  The default, `axis=()`, will compute over all elements into a scalar array with shape `(1,)`.
  If `axis` is int, a reduction is performed on a particular axis.
  If `axis` is a tuple of ints, a reduction is performed on all the axes specified in the tuple.
  If `exclude` is true, reduction will be performed on the axes that are NOT in axis instead.
  Negative values means indexing from right to left.

- **keepdims**
  
  boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.

- **exclude**
  
  boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.

---

mx.nd.radians

*Converts each element of the input array from degrees to radians.*

**Description**

.. math:: radians([0, 90, 180, 270, 360]) = [0, \pi/2, \pi, 3\pi/2, 2\pi]

**Arguments**

- **data**
  
  NDArray-or-Symbol The input array.
Draw random samples from an exponential distribution.

**Description**

Samples are distributed according to an exponential distribution parametrized by *lambda* (rate).

**Arguments**

- `lam` float, optional, default=1 Lambda parameter (rate) of the exponential distribution.
- `shape` Shape(tuple), optional, default=None Shape of the output.
- `ctx` string, optional, default=" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- `dtype` 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

**Details**

Example::

    exponential(lam=4, shape=(2,2)) = [[ 0.0097189 , 0.08999364], [ 0.04146638, 0.31715935]]

Defined in src/operator/random/sample_op.cc:L137

**Value**

out The result mx.ndarray
mx.nd.random.gamma  
*Draw random samples from a gamma distribution.*

**Description**

Samples are distributed according to a gamma distribution parametrized by *alpha* (shape) and *beta* (scale).

**Arguments**

- **alpha**  
  float, optional, default=1 Alpha parameter (shape) of the gamma distribution.

- **beta**  
  float, optional, default=1 Beta parameter (scale) of the gamma distribution.

- **shape**  
  Shape(tuple), optional, default=None Shape of the output.

- **ctx**  
  string, optional, default="" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.

- **dtype**  
  'None', 'float16', 'float32', 'float64',optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

**Details**

Example::

    gamma(alpha=9, beta=0.5, shape=(2,2)) = [[ 7.10486984,  3.37695289], [ 3.91697288,  3.65933681]]  
Defined in src/operator/random/sample_op.cc:L125

**Value**

- **out** The result mx.ndarray

mx.nd.random.generalized.negative.binomial  
*Draw random samples from a generalized negative binomial distribution.*

**Description**

Samples are distributed according to a generalized negative binomial distribution parametrized by *mu* (mean) and *alpha* (dispersion). *alpha* is defined as *1/k* where *k* is the failure limit of the number of unsuccessful experiments (generalized to real numbers). Samples will always be returned as a floating point data type.
**Arguments**

- **mu**: float, optional, default=1 Mean of the negative binomial distribution.
- **alpha**: float, optional, default=1 Alpha (dispersion) parameter of the negative binomial distribution.
- **shape**: Shape(tuple), optional, default=None Shape of the output.
- **ctx**: string, optional, default="" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

**Details**

Example:

```python
generalized_negative_binomial(mu=2.0, alpha=0.3, shape=(2,2)) = [[ 2., 1.], [ 6., 4.]]
```

Defined in src/operator/random/sample_op.cc:L179

**Value**

- **out**: The result mx.ndarray

---

**mx.nd.random.negative.binomial**

*Draw random samples from a negative binomial distribution.*

**Description**

Samples are distributed according to a negative binomial distribution parametrized by *k* (limit of unsuccessful experiments) and *p* (failure probability in each experiment). Samples will always be returned as a floating point data type.

**Arguments**

- **k**: int, optional, default=’1’ Limit of unsuccessful experiments.
- **p**: float, optional, default=1 Failure probability in each experiment.
- **shape**: Shape(tuple), optional, default=None Shape of the output.
- **ctx**: string, optional, default="" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

**Details**

Example:

```python
negative_binomial(k=3, p=0.4, shape=(2,2)) = [[ 4., 7.], [ 2., 5.]]
```

Defined in src/operator/random/sample_op.cc:L164
mx.nd.random.normal

Value

out The result mx.ndarray

mx.nd.random.normal Draw random samples from a normal (Gaussian) distribution.

Description

.. note:: The existing alias “normal” is deprecated.

Arguments

*loc* float, optional, default=0 Mean of the distribution.
*scale* float, optional, default=1 Standard deviation of the distribution.
*shape* Shape(tuple), optional, default=None Shape of the output.
*ctx* string, optional, default=" Context of output, in format [cpu|gpucpu_pinned](n). Only used for imperative calls.
*dtype* 'None', 'float16', 'float32', 'float64',optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

Details

Samples are distributed according to a normal distribution parametrized by *loc* (mean) and *scale* (standard deviation).

Example::

    normal(loc=0, scale=1, shape=(2,2)) = [[ 1.89171135, -1.16881478], [-1.23474145, 1.55807114]]

Defined in src/operator/random/sample_op.cc:L113

Value

out The result mx.ndarray
mx.nd.random.pdf.dirichlet

Computes the value of the PDF of *sample* of Dirichlet distributions with parameter *alpha*.

**Description**

The shape of *alpha* must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *alpha*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the value of *alpha* at index *i*.

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample</td>
<td>NDArray-or-Symbol Samples from the distributions.</td>
</tr>
<tr>
<td>alpha</td>
<td>NDArray-or-Symbol Concentration parameters of the distributions.</td>
</tr>
<tr>
<td>is.log</td>
<td>boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.</td>
</tr>
</tbody>
</table>

**Details**

Examples::

random_pdf_dirichlet(sample=[[1.2],[2.3],[3.4]], alpha=[2.5, 2.5]) = [38.413498, 199.60245, 564.56085]
sample = [[[1, 2, 3], [10, 20, 30], [100, 200, 300]], [[0.1, 0.2, 0.3], [0.01, 0.02, 0.03], [0.001, 0.002, 0.003]]]
random_pdf_dirichlet(sample=sample, alpha=[0.1, 0.4, 0.9]) = [[2.3257459e-02, 5.8420084e-04, 1.4674458e-05], [9.2589635e-01, 3.6860607e+01, 1.4674468e+03]]

Defined in src/operator/random/pdf_op.cc:L316

**Value**

out The result mx.ndarray

mx.nd.random.pdf.exponential

Computes the value of the PDF of *sample* of exponential distributions with parameters *lam* (rate).
mx.nd.random.pdf.gamma

Description

The shape of *lam* must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *lam*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the value of *lam* at index *i*.

Arguments

- **sample**: NDArray-or-Symbol Samples from the distributions.
- **lam**: NDArray-or-Symbol Lambda (rate) parameters of the distributions.
- **is.log**: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.

Details

Examples::

    random_pdf_exponential(sample=[[1, 2, 3]], lam=[1]) = [[0.36787945, 0.13533528, 0.04978707]]
    sample = [[1,2,3], [1,2,3], [1,2,3]]
    random_pdf_exponential(sample=sample, lam=[1,0.5,0.25]) = [[0.36787945, 0.13533528, 0.04978707],
    [0.30326533, 0.18393973, 0.11156508], [0.1947002, 0.15163267, 0.11809164]]

Defined in src/operator/random/pdf_op.cc:L305

Value

- **out**: The result mx.ndarray

mx.nd.random.pdf.gamma

Computes the value of the PDF of *sample* of gamma distributions with parameters *alpha* (shape) and *beta* (rate).

Description

*alpha* and *beta* must have the same shape, which must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *alpha* and *beta*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the values of *alpha* and *beta* at index *i*. 
Arguments

- **sample**: NDArray-or-Symbol Samples from the distributions.
- **alpha**: NDArray-or-Symbol Alpha (shape) parameters of the distributions.
- **is.log**: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
- **beta**: NDArray-or-Symbol Beta (scale) parameters of the distributions.

Details

Examples:

```python
random_pdf_gamma(sample=[[1,2,3,4,5]], alpha=[5], beta=[1]) = [[0.01532831, 0.09022352, 0.16803136, 0.19536681, 0.17546739]]
```

```python
sample = [[1, 2, 3, 4, 5], [2, 3, 4, 5, 6], [3, 4, 5, 6, 7]]
random_pdf_gamma(sample=sample, alpha=[5,6,7], beta=[1,1,1]) = [[0.01532831, 0.09022352, 0.16803136, 0.19536681, 0.17546739], [0.03608941, 0.10081882, 0.15629345, 0.17546739, 0.16062315], [0.05040941, 0.10419563, 0.14622283, 0.16062315, 0.14900276]]
```

Defined in src/operator/random/pdf_op.cc:L303

Value

- **out**: The result mx.ndarray

mx.nd.random.pdf.generalized.negative.binomial

Computes the value of the PDF of *sample* of generalized negative binomial distributions with parameters *mu* (mean) and *alpha* (dispersion). This can be understood as a reparameterization of the negative binomial, where \( k^* = 1 / \alpha \) and \( p^* = 1 / (\mu \times \alpha + 1) \).

Description

*mu* and *alpha* must have the same shape, which must match the leftmost subshape of *sample*.

That is, *sample* can have the same shape as *mu* and *alpha*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the values of *mu* and *alpha* at index *i*.

Arguments

- **sample**: NDArray-or-Symbol Samples from the distributions.
- **mu**: NDArray-or-Symbol Means of the distributions.
- **is.log**: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
- **alpha**: NDArray-or-Symbol Alpha (dispersion) parameters of the distributions.
mx.nd.random.pdf.negative.binomial

Details

Examples::
random_pdf_generalized_negative_binomial(sample=[[1, 2, 3, 4]], alpha=[1], mu=[1]) = [[0.25, 0.125, 0.0625, 0.03125]]
sample = [[1,2,3,4], [1,2,3,4]] random_pdf_generalized_negative_binomial(sample=sample, alpha=[1, 0.6666], mu=[1, 1.5]) = [[0.25, 0.125, 0.0625, 0.03125], [0.26517063, 0.16573331, 0.09667706, 0.05437994]]

Defined in src/operator/random/pdf_op.cc:L314

Value

out The result mx.ndarray

mx.nd.random.pdf.negative.binomial

Computes the value of the PDF of samples of negative binomial distributions with parameters *k* (failure limit) and *p* (failure probability).

Description

*k* and *p* must have the same shape, which must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *k* and *p*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the values of *k* and *p* at index *i*.

Arguments

- **sample** NDArray-or-Symbol Samples from the distributions.
- **k** NDArray-or-Symbol Limits of unsuccessful experiments.
- **is.log** boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
- **p** NDArray-or-Symbol Failure probabilities in each experiment.

Details

Examples::
random_pdf_negative_binomial(sample=[[1,2,3,4]], k=[1], p=a[0.5]) = [[0.25, 0.125, 0.0625, 0.03125]]
# Note that k may be real-valued sample = [[1,2,3,4], [1,2,3,4]] random_pdf_negative_binomial(sample=sample, k=[1, 1.5], p=[0.5, 0.5]) = [[0.25, 0.125, 0.0625, 0.03125], [0.26516506, 0.16572815, 0.09667476, 0.05437956]]

Defined in src/operator/random/pdf_op.cc:L310
Value

out The result mx.ndarray

---

mx.nd.random.pdf.normal

Computes the value of the PDF of *sample* of normal distributions with parameters *mu* (mean) and *sigma* (standard deviation).

Description

*mu* and *sigma* must have the same shape, which must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *mu* and *sigma*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the values of *mu* and *sigma* at index *i*.

Arguments

sample NDArray-or-Symbol Samples from the distributions.
mu NDArray-or-Symbol Means of the distributions.
is.log boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
sigma NDArray-or-Symbol Standard deviations of the distributions.

Details

Examples::

    sample = [[-2, -1, 0, 1, 2]]
    random_pdf_normal(sample=sample, mu=[0], sigma=[1]) = [[0.05399097, 0.24197073, 0.3989423, 0.24197073, 0.05399097]]

    random_pdf_normal(sample=sample*2, mu=[0,0], sigma=[1,2]) = [[0.05399097, 0.24197073, 0.3989423, 0.24197073, 0.05399097], [0.12098537, 0.17603266, 0.19947115, 0.17603266, 0.12098537]]

Defined in src/operator/random/pdf_op.cc:L300

Value

out The result mx.ndarray
Computes the value of the PDF of *sample* of Poisson distributions with parameters *lam* (rate).

**Description**

The shape of *lam* must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *lam*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the value of *lam* at index *i*.

**Arguments**

- **sample**: NDArray-or-Symbol
  Samples from the distributions.

- **lam**: NDArray-or-Symbol
  Lambda (rate) parameters of the distributions.

- **is.log**: boolean, optional, default=0
  If set, compute the density of the log-probability instead of the probability.

**Details**

Examples:

```
random_pdf_poisson(sample=[[0,1,2,3]], lam=[1]) =
[[0.36787945, 0.36787945, 0.18393973, 0.06131324]]
```

```
sample = [[0,1,2,3], [0,1,2,3], [0,1,2,3]]
random_pdf_poisson(sample=sample, lam=[1,2,3]) =
[[0.36787945, 0.36787945, 0.18393973, 0.06131324],
[0.13533528, 0.27067056, 0.27067056, 0.18044704], [0.04978707, 0.14936121, 0.22404182, 0.22404182]]
```

Defined in src/operator/random/pdf_op.cc:L307

**Value**

- **out**: The result mx.ndarray

---

Computes the value of the PDF of *sample* of uniform distributions on the intervals given by *[low,high)*.

**Description**

*low* and *high* must have the same shape, which must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *low* and *high*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the values of *low* and *high* at index *i*.
mx.nd.random.poisson

Arguments

- **sample**: NDArray-or-Symbol Samples from the distributions.
- **low**: NDArray-or-Symbol Lower bounds of the distributions.
- **is.log**: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
- **high**: NDArray-or-Symbol Upper bounds of the distributions.

Details

Examples::

    random_pdf_uniform(sample=[1,2,3,4], low=[0], high=[10]) = [0.1, 0.1, 0.1, 0.1]
    sample = [[[1, 2, 3], [1, 2, 3]], [[1, 2, 3], [1, 2, 3]]] low = [[0, 0], [0, 0]] high = [[5, 10], [15, 20]]
    random_pdf_uniform(sample=sample, low=low, high=high) = [[[0.06667, 0.06667, 0.06667], [0.05, 0.05, 0.05]]

Defined in src/operator/random/pdf_op.cc:L298

Value

- **out**: The result mx.ndarray

mx.nd.random.poisson  Draw random samples from a Poisson distribution.

Description

Samples are distributed according to a Poisson distribution parametrized by *lambda* (rate). Samples will always be returned as a floating point data type.

Arguments

- **lam**: float, optional, default=1 Lambda parameter (rate) of the Poisson distribution.
- **shape**: Shape(tuple), optional, default=None Shape of the output.
- **ctx**: string, optional, default=’’ Context of output, in format [cpu|gpu|cpu_pinned](n).
  Only used for imperative calls.
- **dtype**: ’None’, ‘float16’, ‘float32’, ‘float64’,optional, default=’None’ Dtype of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=\text{None}).

Details

Example::

    poisson(lam=4, shape=(2,2)) = [[ 5., 2.], [ 4., 6.]]

Defined in src/operator/random/sample_op.cc:L150

Value

- **out**: The result mx.ndarray
mx.nd.random.randint  

**Draw random samples from a discrete uniform distribution.**

**Description**

Samples are uniformly distributed over the half-open interval *[low, high)* (includes *low*, but excludes *high*).

**Arguments**

- **low**  
  long, required Lower bound of the distribution.

- **high**  
  long, required Upper bound of the distribution.

- **shape**  
  Shape(tuple), optional, default=None Shape of the output.

- **ctx**  
  string, optional, default="" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.

- **dtype**  
  'None', 'int32', 'int64',optional, default='None' DType of the output in case this can’t be inferred. Defaults to int32 if not defined (dtype=None).

**Details**

Example::

```python
randint(low=0, high=5, shape=(2,2)) = [[ 0, 2], [ 3, 1]]
```

Defined in src/operator/random/sample_op.cc:L194

**Value**

out The result mx.ndarray

mx.nd.random.uniform  

**Draw random samples from a uniform distribution.**

**Description**

.. note:: The existing alias “uniform” is deprecated.

**Arguments**

- **low**  
  float, optional, default=0 Lower bound of the distribution.

- **high**  
  float, optional, default=1 Upper bound of the distribution.

- **shape**  
  Shape(tuple), optional, default=None Shape of the output.

- **ctx**  
  string, optional, default="" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.

- **dtype**  
  'None', 'float16', 'float32', 'float64',optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
mx.nd.ravel.multi.index

Details

Samples are uniformly distributed over the half-open interval *[low, high)* (includes *low*, but excludes *high*).

Example::

uniform(low=0, high=1, shape=(2,2)) = [[ 0.60276335, 0.85794562], [ 0.54488319, 0.84725171]]

Defined in src/operator/random/sample_op.cc:L96

Value

out The result mx.ndarray
mx.nd.rcbrt

Returns element-wise inverse cube-root value of the input.

Description

.. math:: rcbrt(x) = 1/\sqrt[3]x

Arguments

data NDArray-or-Symbol The input array.

Details

Example::
rcbrt([1,8,-125]) = [1.0, 0.5, -0.2]
Defined in src/operator/tensor/elemwise_unary_op_pow.cc:L323

Value

out The result mx.ndarray

mx.nd.reciprocal

Returns the reciprocal of the argument, element-wise.

Description

Calculates 1/x.

Arguments

data NDArray-or-Symbol The input array.

Details

Example::
reciprocal([-2, 1, 3, 1.6, 0.2]) = [-0.5, 1.0, 0.33333334, 0.625, 5.0]
Defined in src/operator/tensor/elemwise_unary_op_pow.cc:L43

Value

out The result mx.ndarray
**mx.nd.relu**

*Computes rectified linear activation.*

**Description**

.. math:: \max(\text{features}, 0)

**Arguments**

- **data**: NDArray-or-Symbol The input array.

**Details**

The storage type of “relu” output depends upon the input storage type:

- relu(default) = default - relu(row_sparse) = row_sparse - relu(csr) = csr

Defined in `src/operator/tensor/elemwise_unary_op_basic.cc:L85`

**Value**

- out The result mx.ndarray

---

**mx.nd.repeat**

*Repeats elements of an array. By default, “repeat“ flattens the input array into 1-D and then repeats the elements::

\[ \begin{bmatrix} 1, 2, 3, 4 \end{bmatrix} \]

\[ \text{repeat}(x, \text{repeats}=2) = \begin{bmatrix} 1., 2., 1., 2., 3., 4., 3., 4. \end{bmatrix} \]

The parameter “axis“ specifies the axis along which to perform repeat::

\[ \text{repeat}(x, \text{axis}=1) = \begin{bmatrix} 1., 1., 2., 2., 3., 3., 4., 4. \end{bmatrix} \]

**Description**

Defined in `src/operator/tensor/matrix_op.cc:L744`

**Arguments**

- **data**: NDArray-or-Symbol Input data array
- **repeats**: int, required The number of repetitions for each element.
- **axis**: int or None, optional, default=’None’ The axis along which to repeat values. The negative numbers are interpreted counting from the backward. By default, use the flattened input array, and return a flat output array.

**Value**

- out The result mx.ndarray
### mx.nd.reset.arrays

Set to zero multiple arrays

---

#### Description

Defined in src/operator/contrib/reset_arrays.cc:L36

#### Arguments

- **data**: NDArray-or-Symbol[] Arrays
- **num.arrays**: int, required number of input arrays.

#### Value

- **out**: The result mx.ndarray
mx.nd.Reshape

Reshapes the input array. .. note:: “Reshape” is deprecated, use “reshape” Given an array and a shape, this function returns a copy of the array in the new shape. The shape is a tuple of integers such as (2,3,4). The size of the new shape should be same as the size of the input array. Example:: reshape([1,2,3,4], shape=(2,2)) = [[1,2], [3,4]] Some dimensions of the shape can take special values from the set 0, -1, -2, -3, -4. The significance of each is explained below: - “0” copy this dimension from the input to the output shape. Example:: - input shape = (2,3,4), shape = (4,0,2), output shape = (4,3,2) - input shape = (2,3,4), shape = (2,0,0), output shape = (2,3,4) - “-1” infers the dimension of the output shape by using the remainder of the input dimensions keeping the size of the new array same as that of the input array. At most one dimension of shape can be -1. Example:: - input shape = (2,3,4), shape = (6,1,-1), output shape = (6,1,4) - input shape = (2,3,4), shape = (3,-1,8), output shape = (3,1,8) - input shape = (2,3,4), shape =(-1,1), output shape = (24,) - “-2” copy all/remainder of the input dimensions to the output shape. Example:: - input shape = (2,3,4), shape = (-2), output shape = (2,3,4) - input shape = (2,3,4), shape = (2,-2), output shape = (2,3,4) - input shape = (2,3,4), shape = (-2,1,1), output shape = (2,3,4,1,1) - “-3” use the product of two consecutive dimensions of the input shape as the output dimension. Example:: - input shape = (2,3,4), shape = (-3,4), output shape = (6,4) - input shape = (2,3,4), shape = (-3,3), output shape = (6,20) - input shape = (2,3,4), shape = (0,-3), output shape = (2,12) - input shape = (2,3,4), shape = (-3,-2), output shape = (6,4) - “-4” split one dimension of the input into two dimensions passed subsequent to -4 in shape (can contain -1). Example:: - input shape = (2,3,4), shape = (-4,1,2,-2), output shape = (1,2,3,4) - input shape = (2,3,4), shape = (2,-4,-1,-3,-2), output shape = (2,1,3,4) If the argument ‘reverse’ is set to 1, then the special values are inferred from right to left. Example:: - without reverse=1, for input shape = (10,5,4), shape = (-1,0), output shape would be (40,5) - with reverse=1, output shape will be (50,4).

Description

Defined in src/operator/tensor/matrix_op.cc:L175

Arguments

data NDArray-or-Symbol Input data to reshape.
shape Shape(tuple), optional, default=[] The target shape
reverse boolean, optional, default=0 If true then the special values are inferred from right to left
target.shape Shape(tuple), optional, default=[] (Deprecated! Use “shape” instead.) Target new shape. One and only one dim can be 0, in which case it will be inferred from the rest of dims
**mx.nd.reshape**

Reshapes the input array. .. note:: “Reshape” is deprecated, use “reshape”

Given an array and a shape, this function returns a copy of the array in the new shape. The shape is a tuple of integers such as (2,3,4). The size of the new shape should be same as the size of the

input array. Example: reshape([1,2,3,4], shape=(2,2)) = [[1,2],

[3,4]] Some dimensions of the shape can take special values from the set 0, -1, -2, -3, -4. The significance of each is explained below: - “0”
copy this dimension from the input to the output shape. Example:

- input shape = (2,3,4), shape = (4,0,2), output shape = (4,3,2) - input

shape = (2,3,4), shape = (2,0,0), output shape = (2,3,4) - “-1” infers
the dimension of the output shape by using the remainder of the input
dimensions keeping the size of the new array same as that of the input
array. At most one dimension of shape can be -1. Example:

- input shape = (2,3,4), shape = (6,1,-1), output shape = (6,1,4) - input

shape = (2,3,4), shape = (3,-1,8), output shape = (3,1,8) - input shape

= (2,3,4), shape=(1,-1,), output shape = (24,) - “-2” copy all/remainder
of the input dimensions to the output shape. Example:

- input shape = (2,3,4), shape = (-2,), output shape = (2,3,4) - input shape

= (2,3,4), shape = (2,-2), output shape = (2,3,4) - input shape

= (-2,1,1), output shape = (2,3,4,1) - “-3” use the product of two
consecutive dimensions of the input shape as the output dimension.
Example:

- input shape = (2,3,4), shape = (-3,4), output shape = (6,4) - input shape

= (2,3,4), shape = (2,3,4,5), shape = (-3,-3), output shape = (6,20) - input
shape = (2,3,4), shape = (0,-3), output shape = (2,12) - input
shape = (2,3,4), shape = (-3,-2), output shape = (6,4) - “-4” split one
dimension of the input into two dimensions passed subsequent to -4 in
shape (can contain -1). Example:

- input shape = (2,3,4), shape = (-4,-1,2,-2), output shape = (1,2,3,4) - input
shape = (2,3,4), shape = (2,-4,-1,3,-2), output shape = (2,1,3,4) If the argument ‘reverse’ is set
to 1, then the special values are inferred from right to left. Example:

- without reverse=1, for input shape = (10,5,4), shape = (-1,0), output
shape would be (40,5) - with reverse=1, output shape will be (50,4).

**Description**

Defined in src/operator/tensor/matrix_op.cc:L175
Arguments

- **data**: NDArray-or-Symbol Input data to reshape.
- **shape**: Shape(tuple), optional, default=[] The target shape
- **reverse**: boolean, optional, default=0 If true then the special values are inferred from right to left
- **target.shape**: Shape(tuple), optional, default=[] (Deprecated! Use “shape“ instead.) Target new shape. One and only one dim can be 0, in which case it will be inferred from the rest of dims
- **keep.highest**: boolean, optional, default=0 (Deprecated! Use “shape“ instead.) Whether keep the highest dim unchanged. If set to true, then the first dim in target_shape is ignored, and always fixed as input

Value

- **out**: The result mx.ndarray

---

**mx.nd.reshape_like**

*Reshape some or all dimensions of ‘lhs‘ to have the same shape as some or all dimensions of ‘rhs‘.*

Description

Returns a **view** of the ‘lhs‘ array with a new shape without altering any data.

Arguments

- **lhs**: NDArray-or-Symbol First input.
- **rhs**: NDArray-or-Symbol Second input.
- **lhs.begin**: int or None, optional, default=’None’ Defaults to 0. The beginning index along which the lhs dimensions are to be reshaped. Supports negative indices.
- **lhs.end**: int or None, optional, default=’None‘ Defaults to None. The ending index along which the lhs dimensions are to be used for reshaping. Supports negative indices.
- **rhs.begin**: int or None, optional, default=’None‘ Defaults to 0. The beginning index along which the rhs dimensions are to be used for reshaping. Supports negative indices.
- **rhs.end**: int or None, optional, default=’None‘ Defaults to None. The ending index along which the rhs dimensions are to be used for reshaping. Supports negative indices.
Details

Example::

x = [1, 2, 3, 4, 5, 6] y = [[0, -4], [3, 2], [2, 2]]
reshape_like(x, y) = [[1, 2], [3, 4], [5, 6]]

More precise control over how dimensions are inherited is achieved by specifying \ slices over the 'lhs' and 'rhs' array dimensions. Only the sliced 'lhs' dimensions \ are reshaped to the 'rhs' sliced dimensions, with the non-sliced 'lhs' dimensions staying the same.

Examples::

- lhs shape = (30,7), rhs shape = (15,2,4), lhs_begin=0, lhs_end=1, rhs_begin=0, rhs_end=2, output shape = (15,2,7) - lhs shape = (3, 5), rhs shape = (1,15,4), lhs_begin=0, lhs_end=2, rhs_begin=1, rhs_end=2, output shape = (15)

Negative indices are supported, and 'None' can be used for either 'lhs_end' or 'rhs_end' to indicate the end of the range.

Example::

- lhs shape = (30, 12), rhs shape = (4, 2, 2, 3), lhs_begin=-1, lhs_end=None, rhs_begin=1, rhs_end=None, output shape = (30, 2, 2, 3)

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L511

Value

out The result mx.ndarray
\textbf{mx.nd.rint} \hspace{1cm} \textit{Returns element-wise rounded value to the nearest integer of the input.}

\textbf{Description}

- For input “n.5” “rint” returns “n” while “round” returns “n+1”.
- For input “-n.5” both “rint” and “round” returns “-n-1”.

\textbf{Arguments}

- **data** \hspace{1cm} \text{NDArray-or-Symbol} The input array.

\textbf{Details}

Example:

\begin{verbatim}
rint([-1.5, 1.5, -1.9, 1.9, 2.1]) = [-2., 1., -2., 2., 2.]
\end{verbatim}

The storage type of “rint” output depends upon the input storage type:

- rint(default) = default
- rint(row_sparse) = row_sparse
- rint(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L798

\textbf{Value}

- **out** The result mx.ndarray

\textbf{mx.nd.rmsprop.update} \hspace{1cm} \textit{Update function for ‘RMSProp’ optimizer.}

\textbf{Description}

‘RMSprop’ is a variant of stochastic gradient descent where the gradients are divided by a cache which grows with the sum of squares of recent gradients?

\textbf{Arguments}

- **weight** \hspace{1cm} \text{NDArray-or-Symbol} Weight
- **grad** \hspace{1cm} \text{NDArray-or-Symbol} Gradient
- **n** \hspace{1cm} \text{NDArray-or-Symbol} n
- **lr** \hspace{1cm} \text{float, required} Learning rate
- **rho** \hspace{1cm} \text{float, optional, default=0.949999988} The decay rate of momentum estimates.
- **epsilon** \hspace{1cm} \text{float, optional, default=9.99999994e-09} A small constant for numerical stability.
- **wd** \hspace{1cm} \text{float, optional, default=0} Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
rescale.grad  float, optional, default=1  Rescale gradient to grad = rescale_grad*grad.
clip.gradient  float, optional, default=-1  Clip gradient to the range of [-clip_gradient, clip_gradient]
If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
clip.weights  float, optional, default=-1  Clip weights to the range of [-clip_weights, clip_weights]
If clip_weights <= 0, weight clipping is turned off. weights = max(min(weights, clip_weights), -clip_weights).

Details

‘RMSProp’ is similar to ‘AdaGrad’, a popular variant of ‘SGD’ which adaptively tunes the learning rate of each parameter. ‘AdaGrad’ lowers the learning rate for each parameter monotonically over the course of training. While this is analytically motivated for convex optimizations, it may not be ideal for non-convex problems. ‘RMSProp’ deals with this heuristically by allowing the learning rates to rebound as the denominator decays over time.

Define the Root Mean Square (RMS) error criterion of the gradient as :math:'RMS[g]_t = \sqrt{E[g^2]_t + \epsilon}', where :math:'g' represents gradient and :math:'E[g^2]_t' is the decaying average over past squared gradient.

The :math:'E[g^2]_t' is given by:

.. math:: E[g^2]_t = \rho * E[g^2]_{t-1} + (1-\rho) * g_t^2

The update step is

.. math:: \theta_{t+1} = \theta_t - \frac{\eta}{RMS[g]_t} g_t


Hinton suggests the momentum term :math:’\rho’ to be 0.9 and the learning rate :math:’\eta’ to be 0.001.

Defined in src/operator/optimizer_op.cc:L788

Value

out  The result mx.ndarray

mx.nd.rmspropalex.update  
Update function for RMSPropAlex optimizer.

Description

‘RMSPropAlex’ is non-centered version of ‘RMSProp’.
Arguments

weight NDArray-or-Symbol Weight

grad NDArray-or-Symbol Gradient

n NDArray-or-Symbol n

g NDArray-or-Symbol g

delta NDArray-or-Symbol delta

lr float, required Learning rate

rho float, optional, default=0.949999988 Decay rate.

momentum float, optional, default=0.999999976 Decay rate.

epsilon float, optional, default=9.99999994e-09 A small constant for numerical stability.

wd float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.

rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.

clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).

clip.weights float, optional, default=-1 Clip weights to the range of [-clip_weights, clip_weights] If clip_weights <= 0, weight clipping is turned off. weights = max(min(weights, clip_weights), -clip_weights).

Details

Define :math:`E[g^2]_t` is the decaying average over past squared gradient and :math:`E[g]_t` is the decaying average over past gradient.

.. math::
E[g^2]_t = \rho E[g^2]_{t-1} + (1 - \rho) g_t^2
E[g]_t = \rho E[g]_{t-1} + (1 - \rho) g_t
momentum_t = \gamma momentum_{t-1} - \frac \eta \sqrt{E[g^2]_t - E[g]_t^2} + \epsilon g_t

The update step is

.. math::
\theta_{t+1} = \theta_t + momentum_t

The RMSPropAlex code follows the version in http://arxiv.org/pdf/1308.0850v5.pdf Eq(38) - Eq(45) by Alex Graves, 2013.

Graves suggests the momentum term :math:`\rho` to be 0.95, :math:`\gamma` to be 0.9 and the learning rate :math:`\eta` to be 0.0001.

Defined in src/operator/optimizer_op.cc:L827

Value

out The result mx.nd.array
mx.nd.RNN

Applies recurrent layers to input data. Currently, vanilla RNN, LSTM and GRU are implemented, with both multi-layer and bidirectional support.

Description

When the input data is of type float32 and the environment variables MXNET_CUDA_ALLOW_TENSOR_CORE and MXNET_CUDA_TENSOR_OP_MATH_ALLOW_CONVERSION are set to 1, this operator will try to use pseudo-float16 precision (float32 math with float16 I/O) precision in order to use Tensor Cores on suitable NVIDIA GPUs. This can sometimes give significant speedups.

Arguments

data NDArray-or-Symbol Input data to RNN
parameters NDArray-or-Symbol Vector of all RNN trainable parameters concatenated
state NDArray-or-Symbol initial hidden state of the RNN
state.cell NDArray-or-Symbol initial cell state for LSTM networks (only for LSTM)
sequence.length NDArray-or-Symbol Vector of valid sequence lengths for each element in batch. (Only used if use_sequence_length kwarg is True)
state.size int (non-negative), required size of the state for each layer
num.layers int (non-negative), required number of stacked layers
bidirectional boolean, optional, default=0 whether to use bidirectional recurrent layers
mode 'gru', 'lstm', 'rnn_relu', 'rnn_tanh', required the type of RNN to compute
p float, optional, default=0 drop rate of the dropout on the outputs of each RNN layer, except the last layer.
state.outputs boolean, optional, default=0 Whether to have the states as symbol outputs.
projection.size int or None, optional, default=’None’ size of project size
lstm.state.clip.min double or None, optional, default=None Minimum clip value of LSTM states. This option must be used together with lstm_state_clip_max.
lstm.state.clip.max double or None, optional, default=None Maximum clip value of LSTM states. This option must be used together with lstm_state_clip_min.
lstm.state.clip.nan boolean, optional, default=0 Whether to stop NaN from propagating in state by clipping it to min/max. If clipping range is not specified, this option is ignored.
use.sequence.length boolean, optional, default=0 If set to true, this layer takes in an extra input parameter ‘sequence_length’ to specify variable length sequence
Details

**Vanilla RNN**
Applies a single-gate recurrent layer to input X. Two kinds of activation function are supported: ReLU and Tanh.
With ReLU activation function:

\[
\text{ReLU}: h_t = \text{relu}(W_{ih} \cdot x_t + b_{ih} + W_{hh} \cdot h_{(t-1)} + b_{hh})
\]
With Tanh activation function:

\[
\text{Tanh}: h_t = \tanh(W_{ih} \cdot x_t + b_{ih} + W_{hh} \cdot h_{(t-1)} + b_{hh})
\]

**LSTM**

\[
\begin{array}{ll}
\text{Math:} & i_t = \text{sigmoid}(W_{ii} \cdot x_t + b_{ii} + W_{hi} \cdot h_{(t-1)} + b_{hi}) \quad f_t = \text{sigmoid}(W_{if} \cdot x_t + b_{if} + W_{hf} \cdot h_{(t-1)} + b_{hf}) \quad g_t = \tanh(W_{ig} \cdot x_t + b_{ig} + W_{hc} \cdot h_{(t-1)} + b_{hg}) \quad o_t = \text{sigmoid}(W_{io} \cdot x_t + b_{io} + W_{ho} \cdot h_{(t-1)} + b_{ho}) \quad c_t = f_t \cdot c_{(t-1)} + i_t \cdot g_t \quad h_t = o_t \cdot \tanh(c_t)
\end{array}
\]
With the projection size being set, LSTM could use the projection feature to reduce the parameters size and give some speedups without significant damage to the accuracy.


\[
\begin{array}{ll}
\text{Math:} & i_t = \text{sigmoid}(W_{ii} \cdot x_t + b_{ii} + W_{ri} \cdot h_{(t-1)} + b_{ri}) \quad f_t = \text{sigmoid}(W_{if} \cdot x_t + b_{if} + W_{rf} \cdot r_{(t-1)} + b_{rf}) \quad g_t = \tanh(W_{ig} \cdot x_t + b_{ig} + W_{rc} \cdot r_{(t-1)} + b_{rg}) \quad o_t = \text{sigmoid}(W_{io} \cdot x_t + b_{io} + W_{ro} \cdot r_{(t-1)} + b_{ro}) \quad c_t = f_t \cdot c_{(t-1)} + i_t \cdot g_t \quad h_t = o_t \cdot \tanh(c_t) \quad r_t = W_{hr} \cdot h_t
\end{array}
\]

**GRU**

The definition of GRU here is slightly different from paper but compatible with CUDNN.

\[
\begin{array}{ll}
\text{Math:} & r_t = \text{sigmoid}(W_{ir} \cdot x_t + b_{ir} + W_{hr} \cdot h_{(t-1)} + b_{hr}) \quad z_t = \text{sigmoid}(W_{iz} \cdot x_t + b_{iz} + W_{hz} \cdot h_{(t-1)} + b_{hz}) \quad n_t = \tanh(W_{in} \cdot x_t + b_{in} + r_t \cdot (W_{hn} \cdot h_{(t-1)} + b_{hn})) \quad h_t = (1 - z_t) \cdot n_t + z_t \cdot h_{(t-1)}
\end{array}
\]
Defined in src/operator/rnn.cc:L368

Value

out The result mx.ndarray
mx.nd.ROIPooling

Performs region of interest (ROI) pooling on the input array.

Description

ROI pooling is a variant of a max pooling layer, in which the output size is fixed and region of interest is a parameter. Its purpose is to perform max pooling on the inputs of non-uniform sizes to obtain fixed-size feature maps. ROI pooling is a neural-net layer mostly used in training a ‘Fast R-CNN’ network for object detection.

Arguments

data NDArray-or-Symbol The input array to the pooling operator, a 4D Feature maps
rois NDArray-or-Symbol Bounding box coordinates, a 2D array of 
  [[batch_index, x1, y1, x2, y2]], where (x1, y1) and (x2, y2) are top left and bottom right corners of designated region of interest. ‘batch_index’ indicates the index of corresponding image in the input array
pooled.size Shape(tuple), required ROI pooling output shape (h,w)
spatial.scale float, required Ratio of input feature map height (or w) to raw image height (or w). Equals the reciprocal of total stride in convolutional layers

Details

This operator takes a 4D feature map as an input array and region proposals as ‘rois’, then it pools over sub-regions of input and produces a fixed-sized output array regardless of the ROI size.

To crop the feature map accordingly, you can resize the bounding box coordinates by changing the parameters ‘rois’ and ‘spatial_scale’.

The cropped feature maps are pooled by standard max pooling operation to a fixed size output indicated by a ‘pooled_size’ parameter. batch_size will change to the number of region bounding boxes after ‘ROIPooling’.

The size of each region of interest doesn’t have to be perfectly divisible by the number of pooling sections(‘pooled_size’).

Example::

// region of interest i.e. bounding box coordinates. y = [[0,0,0,4,4]]
// returns array of shape (2,2) according to the given roi with max pooling. ROIPooling(x, y, (2,2), 1.0) = [[[ 14., 16.], [ 26., 28.]]]
// region of interest is changed due to the change in ‘spacial_scale’ parameter. ROIPooling(x, y, (2,2), 0.7) = [[[ 7., 9.], [ 19., 21.]]]

Defined in src/operator/roi_pooling.cc:L225
mx.nd.round

Returns element-wise rounded value to the nearest integer of the input.

Description

Example::

Arguments

data NDArray-or-Symbol The input array.

Details

round([-1.5, 1.5, -1.9, 1.9, 2.1]) = [-2., 2., -2., 2., 2.]
The storage type of "round" output depends upon the input storage type:
- round(default) = default - round(row_sparse) = row_sparse - round(csr) = csr
Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L777

mx.nd.rsqrt

Returns element-wise inverse square-root value of the input.

Description

.. math:: rsqrt(x) = 1/\sqrt{x}

Arguments

data NDArray-or-Symbol The input array.

Details

rsqrt([4,9,16]) = [0.5, 0.33333334, 0.25]
The storage type of "rsqrt" output is always dense
Defined in src/operator/tensor/elemwise_unary_op_pow.cc:L221

Value

out The result mx.nd.array
mx.nd.sample.exponential

Concurrent sampling from multiple exponential distributions with parameters lambda (rate).

Description

The parameters of the distributions are provided as an input array. Let \([s]\) be the shape of the input array, \(n\) be the dimension of \([s]\), \([t]\) be the shape specified as the parameter of the operator, and \(m\) be the dimension of \([t]\). Then the output will be a \((n+m)\)-dimensional array with shape \([s]\times[t]\).

Arguments

- \(\text{lam}\) NDArray-or-Symbol Lambda (rate) parameters of the distributions.
- \(\text{shape}\) Shape(tuple), optional, default=\([]\) Shape to be sampled from each random distribution.
- \(\text{dtype}\) 'None', 'float16', 'float32', 'float64',optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

Details

For any valid \(n\)-dimensional index \(i\) with respect to the input array, \(output[i]\) will be an \(m\)-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input value at index \(i\). If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input array.

Examples:

\[
\text{lam} = [1.0, 8.5]
\]

// Draw a single sample for each distribution
\[
\text{sample\_exponential(lam)} = [0.51837951, 0.09994757]
\]

// Draw a vector containing two samples for each distribution
\[
\text{sample\_exponential(lam, shape=(2))} = [[0.51837951, 0.19866663], [0.09994757, 0.50447971]]
\]

Defined in src/operator/random/multisample_op.cc:L284

Value

- \(\text{out}\) The result mx.nd.array
**mx.nd.sample.gamma**  
Concurrent sampling from multiple gamma distributions with parameters *alpha* (shape) and *beta* (scale).

### Description

The parameters of the distributions are provided as input arrays. Let *[s]* be the shape of the input arrays, *[n]* be the dimension of *[s]*, *[t]* be the shape specified as the parameter of the operator, and *[m]* be the dimension of *[t]*. Then the output will be a *(n+m)*-dimensional array with shape *[s]*x*[t]*.

### Arguments

- **alpha**  
  NDArray-or-Symbol Alpha (shape) parameters of the distributions.

- **shape**  
  Shape(tuple), optional, default=[:] Shape to be sampled from each random distribution.

- **dtype**  
  'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

- **beta**  
  NDArray-or-Symbol Beta (scale) parameters of the distributions.

### Details

For any valid *[n]*-dimensional index *[i]* with respect to the input arrays, *[output][i]* will be an *[m]*-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input values at index *[i]*. If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input arrays.

#### Examples:

alpha = [ 0.0, 2.5 ] beta = [ 1.0, 0.7 ]  

// Draw a single sample for each distribution sample_gamma(alpha, beta) = [ 0.0, 2.5797319]  

// Draw a vector containing two samples for each distribution sample_gamma(alpha, beta, shape=(2)) = [[ 0.0, 0.0 ], [ 2.5797319, 1.70734084]]

Defined in src/operator/random/multisample_op.cc:L282

### Value

out The result mx.nd.array
**mx.nd.sample.generalized.negative.binomial**

Concurrent sampling from multiple generalized negative binomial distributions with parameters *mu* (mean) and *alpha* (dispersion).

---

**Description**

The parameters of the distributions are provided as input arrays. Let *[s]* be the shape of the input arrays, *[n]* be the dimension of *[s]*, *[t]* be the shape specified as the parameter of the operator, and *[m]* be the dimension of *[t]*. Then the output will be a *(n+m)*-dimensional array with shape *[s]x[t]*.

**Arguments**

- **mu** (NDArray-or-Symbol): Means of the distributions.
- **shape** (Shape(tuple), optional, default=[]) : Shape to be sampled from each random distribution.
- **dtype** ('None', 'float16', 'float32', 'float64', optional, default='None'): DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **alpha** (NDArray-or-Symbol): Alpha (dispersion) parameters of the distributions.

**Details**

For any valid *[n]*-dimensional index *[i]* with respect to the input arrays, *[output[i]]* will be an *[m]*-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input values at index *[i]*. If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input arrays.

Samples will always be returned as a floating point data type.

**Examples:**

```python
mu = [ 2.0, 2.5 ]
alpha = [ 1.0, 0.1 ]

// Draw a single sample for each distribution
sample_generalized_negative_binomial(mu, alpha) = [ 0., 3. ]

// Draw a vector containing two samples for each distribution
sample_generalized_negative_binomial(mu, alpha, shape=(2)) = [[ 0., 3.], [ 3., 1.]]
```

Defined in src/operator/random/multisample_op.cc:L293

**Value**

out The result mx.ndarray
mx.nd.sample.multinomial

Concurrent sampling from multiple multinomial distributions.

Description

*data* is an *n* dimensional array whose last dimension has length *k*, where *k* is the number of possible outcomes of each multinomial distribution. This operator will draw *shape* samples from each distribution. If shape is empty one sample will be drawn from each distribution.

Arguments

- **data**: NDArray-or-Symbol Distribution probabilities. Must sum to one on the last axis.
- **shape**: Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.
- **get.prob**: boolean, optional, default=0 Whether to also return the log probability of sampled result. This is usually used for differentiating through stochastic variables, e.g. in reinforcement learning.
- **dtype**: 'float16', 'float32', 'float64', 'int32', 'uint8', optional, default='int32' Dtype of the output in case this can’t be inferred.

Details

If *get_prob* is true, a second array containing log likelihood of the drawn samples will also be returned. This is usually used for reinforcement learning where you can provide reward as head gradient for this array to estimate gradient.

Note that the input distribution must be normalized, i.e. *data* must sum to 1 along its last axis.

Examples:

```python
probs = [[0, 0.1, 0.2, 0.3, 0.4], [0.4, 0.3, 0.2, 0.1, 0]]
// Draw a single sample for each distribution sample_multinomial(probs) = [3, 0]
// Draw a vector containing two samples for each distribution sample_multinomial(probs, shape=(2))
= [[4, 2], [0, 0]]
// requests log likelihood sample_multinomial(probs, get_prob=True) = [2, 1], [0.2, 0.3]
```

Value

out The result mx.ndarray
**mx.nd.sample.negative.binomial**

*Concurrent sampling from multiple negative binomial distributions with parameters *k* (failure limit) and *p* (failure probability).*

**Description**

The parameters of the distributions are provided as input arrays. Let *[s]* be the shape of the input arrays, *n* be the dimension of *[s]*, *[t]* be the shape specified as the parameter of the operator, and *m* be the dimension of *[t]*. Then the output will be a *(n+m)*-dimensional array with shape *[s]x[t]*.

**Arguments**

- **k**
  NDArray-or-Symbol, Limits of unsuccessful experiments.

- **shape**
  Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.

- **dtype**
  ‘None’, ‘float16’, ‘float32’, ‘float64’, optional, default=’None’ DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

- **p**
  NDArray-or-Symbol, Failure probabilities in each experiment.

**Details**

For any valid *n*-dimensional index *i* with respect to the input arrays, *output[i]* will be an *m*-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input values at index *i*. If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input arrays.

Samples will always be returned as a floating point data type.

**Examples:**

```python
k = [ 20, 49 ] p = [ 0.4 , 0.77 ]
// Draw a single sample for each distribution
sample_negative_binomial(k, p) = [ 15., 16.]

// Draw a vector containing two samples for each distribution
sample_negative_binomial(k, p, shape=(2)) = [[ 15., 50.], [ 16., 12.]]
```

Defined in src/operator/random/multisample_op.cc:L289

**Value**

- **out** The result mx.ndarray
Concurrent sampling from multiple normal distributions with parameters *mu* (mean) and *sigma* (standard deviation).

### Description

The parameters of the distributions are provided as input arrays. Let *s* be the shape of the input arrays, *n* be the dimension of *s*, *t* be the shape specified as the parameter of the operator, and *m* be the dimension of *t*. Then the output will be a *(n+m)*-dimensional array with shape *s*t.

### Arguments

- **mu**: NDArray-or-Symbol Means of the distributions.
- **shape**: Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **sigma**: NDArray-or-Symbol Standard deviations of the distributions.

### Details

For any valid *n*-dimensional index *i* with respect to the input arrays, *output[i]* will be an *m*-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input values at index *i*. If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input arrays.

#### Examples:

```python
mu = [0.0, 2.5] sigma = [1.0, 3.7]
// Draw a single sample for each distribution
sample_normal(mu, sigma) = [-0.56410581, 0.95934606]
// Draw a vector containing two samples for each distribution
sample_normal(mu, sigma, shape=(2)) = [[-0.56410581, 0.2928229], [0.95934606, 4.48287058]]
```

Defined in src/operator/random/multisample_op.cc:L279

### Value

- **out**: The result mx.ndarray
**Description**

The parameters of the distributions are provided as an input array. Let \( *[s] * \) be the shape of the input array, \( *n* \) be the dimension of \( *[s] * \), \( *[t] * \) be the shape specified as the parameter of the operator, and \( *m* \) be the dimension of \( *[t] * \). Then the output will be a \((n+m)\)-dimensional array with shape \( *[s]x*[t] * \).

**Arguments**

- **lam**: NDArray-or-Symbol Lambda (rate) parameters of the distributions.
- **shape**: Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.
- **dtype**: ‘None’, ‘float16’, ‘float32’, ‘float64’, optional, default=’None’ DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

**Details**

For any valid \( *n* \)-dimensional index \( *i* \) with respect to the input array, \( *\text{output}[i]* \) will be an \( *m* \)-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input value at index \( *i* \). If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input array.

Samples will always be returned as a floating point data type.

**Examples**:

```
lam = [ 1.0, 8.5 ]
// Draw a single sample for each distribution
sample_poisson(lam) = [ 0., 13.]

// Draw a vector containing two samples for each distribution
sample_poisson(lam, shape=(2)) = [[ 0., 4.], [ 13., 8.]]
```

Defined in src/operator/random/multisample_op.cc:L286

**Value**

- `out` The result mx.nd.array
mx.nd.sample.uniform

Concurrent sampling from multiple uniform distributions on the intervals given by *[low,high]*.

Description

The parameters of the distributions are provided as input arrays. Let *[s]* be the shape of the input arrays, *n* be the dimension of *[s]*, *[t]* be the shape specified as the parameter of the operator, and *m* be the dimension of *[t]*. Then the output will be a *(n+m)*-dimensional array with shape *[s]x[t]*.

Arguments

- **low**: NDArray-or-Symbol Lower bounds of the distributions.
- **shape**: Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **high**: NDArray-or-Symbol Upper bounds of the distributions.

Details

For any valid *n*-dimensional index *i* with respect to the input arrays, *output[i]* will be an *m*-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input values at index *i*. If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input arrays.

Examples:

```plaintext
low = [ 0.0, 2.5 ] high = [ 1.0, 3.7 ]
// Draw a single sample for each distribution sample_uniform(low, high) = [ 0.40451524, 3.18687344]
// Draw a vector containing two samples for each distribution sample_uniform(low, high, shape=(2)) = [[ 0.40451524, 0.18017688], [ 3.18687344, 3.68352246]]
```

Defined in src/operator/random/multisample_op.cc:L277

Value

- **out**: The result mx.ndarray
mx.nd.save  

Save an mx.nd.array object

Description

Save an mx.nd.array object

Usage

mx.nd.save(ndarray, filename)

Arguments

- ndarray: the mx.nd.array object
- filename: the filename (including the path)

Examples

mat = mx.nd.array(1:3)
mx.nd.save(mat, 'temp.mat')
mat2 = mx.nd.load('temp.mat')
as.array(mat)
as.array(mat2[[1]])

mx.nd.scatter.nd  

Scatters data into a new tensor according to indices.

Description

Given 'data' with shape '(Y_0, ..., Y_K-1, X_M, ..., X_N-1)' and indices with shape '(M, Y_0, ..., Y_K-1)', the output will have shape '(X_0, X_1, ..., X_N-1)', where 'M <= N'. If 'M == N', data shape should simply be '(Y_0, ..., Y_K-1)'.

Arguments

- data: NDArray-or-Symbol data
- indices: NDArray-or-Symbol indices
- shape: Shape(tuple), required Shape of output.
**mx.nd.SequenceLast**

Takes the last element of a sequence.

**Description**

This function takes an n-dimensional input array of the form \([\text{max\_sequence\_length}, \text{batch\_size}, \text{other\_feature\_dims}]\) and returns a \((n-1)\)-dimensional array of the form \([\text{batch\_size}, \text{other\_feature\_dims}]\).

**Arguments**

- **data**
  - NDArray-or-Symbol n-dimensional input array of the form \([\text{max\_sequence\_length}, \text{batch\_size}, \text{other\_feature\_dims}]\) where n>2

- **sequence.length**
  - NDArray-or-Symbol vector of sequence lengths of the form \([\text{batch\_size}]\)

- **use.sequence.length**
  - boolean, optional, default=0 If set to true, this layer takes in an extra input parameter ‘sequence.length’ to specify variable length sequence

- **axis**
  - int, optional, default=’0’ The sequence axis. Only values of 0 and 1 are currently supported.
mx.nd.SequenceMask

Details

Parameter `sequence_length` is used to handle variable-length sequences. `sequence_length` should be an input array of positive ints of dimension [batch_size]. To use this parameter, set `use_sequence_length` to `True`, otherwise each example in the batch is assumed to have the max sequence length.

.. note:: Alternatively, you can also use `take` operator.

Example::

    x = [[[ 1., 2., 3.], [ 4., 5., 6.], [ 7., 8., 9.]],
    // returns last sequence when sequence_length parameter is not used
    SequenceLast(x) = [[[ 19., 20., 21.], [ 22., 23., 24.], [ 25., 26., 27.]]]
    // sequence_length is used
    SequenceLast(x, sequence_length=[1,1,1], use_sequence_length=True) = [[[ 1., 2., 3.], [ 4., 5., 6.], [ 7., 8., 9.]]
    // sequence_length is used
    SequenceLast(x, sequence_length=[1,2,3], use_sequence_length=True) = [[[ 1., 2., 3.], [ 13., 14., 15.], [ 25., 26., 27.]]

Defined in src/operator/sequence_last.cc:L106

Value

out The result mx.ndarray

mx.nd.SequenceMask

Sets all elements outside the sequence to a constant value.

Description

This function takes an n-dimensional input array of the form `[max_sequence_length, batch_size, other_feature_dims]` and returns an array of the same shape.

Arguments

data NDArray-or-Symbol n-dimensional input array of the form `[max_sequence_length, batch_size, other_feature_dims]` where n>2

sequence_length NDArray-or-Symbol vector of sequence lengths of the form `[batch_size]`

use_sequence_length boolean, optional, default=0 If set to true, this layer takes in an extra input parameter `sequence_length` to specify variable length sequence

value float, optional, default=0 The value to be used as a mask.

axis int, optional, default=’0’ The sequence axis. Only values of 0 and 1 are currently supported.
Details

Parameter `sequence_length` is used to handle variable-length sequences. `sequence_length` should be an input array of positive ints of dimension `[batch_size]`. To use this parameter, set `use_sequence_length` to `True`, otherwise each example in the batch is assumed to have the max sequence length and this operator works as the `identity` operator.

Example::

```python
import mxnet as mx

x = [[[ 1., 2., 3.], [ 4., 5., 6.]],
     [[ 7., 8., 9.], [10., 11., 12.]],
     [[13., 14., 15.], [16., 17., 18.]]]

# Batch 1 B1 = [[[ 1., 2., 3.], [ 7., 8., 9.], [13., 14., 15.]]

# works as identity operator when sequence_length parameter is not used
SequenceMask(x) = [[[ 1., 2., 3.], [ 4., 5., 6.]],
                    [[ 7., 8., 9.], [10., 11., 12.]],
                    [[13., 14., 15.], [16., 17., 18.]]]

# sequence_length [1,1] means 1 of each batch will be kept // and other rows are masked with default mask value = 0
SequenceMask(x, sequence_length=[1,1], use_sequence_length=True) = [[[ 1., 2., 3.], [ 4., 5., 6.]],
                                                                   [[ 0., 0., 0.], [ 0., 0., 0.]],
                                                                   [[ 0., 0., 0.], [ 0., 0., 0.]]]

# sequence_length [2,3] means 2 of batch B1 and 3 of batch B2 will be kept // and other rows are masked with value = 1
SequenceMask(x, sequence_length=[2,3], use_sequence_length=True, value=1) = [[[ 1., 2., 3.], [ 4., 5., 6.]],
                                                                       [[ 7., 8., 9.], [10., 11., 12.]],
                                                                       [[ 1., 1., 1.], [16., 17., 18.]]]

Defined in src/operator/sequence_mask.cc:L186
```

Value

out The result mx.ndarray

**mx.nd.SequenceReverse**  
Reverses the elements of each sequence.

Description

This function takes an n-dimensional input array of the form `[max_sequence_length, batch_size, other_feature_dims]` and returns an array of the same shape.
**mx.nd.SequenceReverse**

**Arguments**

- **data**
  - NDArray-or-Symbol n-dimensional input array of the form [max_sequence_length, batch_size, other dims] where n>2

- **sequence.length**
  - NDArray-or-Symbol vector of sequence lengths of the form [batch_size]

- **use.sequence.length**
  - boolean, optional, default=0 If set to true, this layer takes in an extra input parameter 'sequence_length' to specify variable length sequence

- **axis**
  - int, optional, default='0' The sequence axis. Only 0 is currently supported.

**Details**

Parameter 'sequence_length' is used to handle variable-length sequences. 'sequence_length' should be an input array of positive ints of dimension [batch_size]. To use this parameter, set 'use_sequence_length' to 'True', otherwise each example in the batch is assumed to have the max sequence length.

Example::

```python
x = [[[ 1., 2., 3.], [ 4., 5., 6.]],
[[ 7., 8., 9.], [10., 11., 12.]],
[[13., 14., 15.], [16., 17., 18.]]]
// Batch 1 B1 = [[ 1., 2., 3.], [ 7., 8., 9.], [13., 14., 15.]]
// returns reverse sequence when sequence_length parameter is not used SequenceReverse(x) = [[[ 13., 14., 15.], [16., 17., 18.]],
[[ 7., 8., 9.], [10., 11., 12.]],
[[ 1., 2., 3.], [ 4., 5., 6.]]]
// sequence_length [2,2] means 2 rows of // both batch B1 and B2 will be reversed. SequenceRe-
verse(x, sequence_length=[2,2], use_sequence_length=True) = [[[ 7., 8., 9.], [10., 11., 12.]],
[[ 1., 2., 3.], [ 4., 5., 6.]],
[[13., 14., 15.], [16., 17., 18.]]]
// sequence_length [2,3] means 2 of batch B2 and 3 of batch B3 // will be reversed. SequenceRe-
verse(x, sequence_length=[2,3], use_sequence_length=True) = [[[ 7., 8., 9.], [16., 17., 18.]],
[[ 1., 2., 3.], [10., 11., 12.]],
[[13., 14., 15.], [ 4., 5., 6.]]]
```

Defined in src/operator/sequence_reverse.cc:L122

**Value**

- **out** The result mx.ndarray
Momentum update function for Stochastic Gradient Descent (SGD) optimizer.

Description

Momentum update has better convergence rates on neural networks. Mathematically it looks like below:

Arguments

weight NDArray-or-Symbol Weight

grad NDArray-or-Symbol Gradient

mom NDArray-or-Symbol Momentum

lr float, required Learning rate

momentum float, optional, default=0 The decay rate of momentum estimates at each epoch.

wd float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.

rescale.grad float, optional, default=1 Rescale gradient to grad = rescale.grad*grad.

clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip.gradient, clip.gradient] If clip.gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip.gradient), -clip.gradient).

lazy.update boolean, optional, default=1 If true, lazy updates are applied if gradient’s stype is row_sparse and both weight and momentum have the same stype.

Details

.. math::

   v_{t-1} = \alpha \nabla J(W_0) \quad v_t = \gamma v_{t-1} - \alpha \nabla J(W_{t-1}) \quad W_t = W_{t-1} + v_t

It updates the weights using:

v = momentum * v - learning_rate * gradient weight += v

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.

However, if grad’s storage type is “row_sparse”, “lazy_update” is True and weight’s storage type is the same as momentum’s storage type, only the row slices whose indices appear in grad.indices are updated (for both weight and momentum):


Defined in src/operator/optimizer_op.cc:L556

Value

out The result mx.ndarray
mx.nd.sgd.update

Update function for Stochastic Gradient Descent (SGD) optimizer.

Description

It updates the weights using::

Arguments

- weight: NDArray-or-Symbol Weight
- grad: NDArray-or-Symbol Gradient
- lr: float, required Learning rate
- wd: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- rescale.grad: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- clip.gradient: float, optional, default=-1 Clip gradient to the range of \([-\text{clip}\_\text{gradient}, \text{clip}\_\text{gradient}]\) If clip\_gradient <= 0, gradient clipping is turned off. \(\text{grad} = \max(\min(\text{grad}, \text{clip}\_\text{gradient}), -\text{clip}\_\text{gradient})\).
- lazy.update: boolean, optional, default=1 If true, lazy updates are applied if gradient’s stype is row\_sparse.

Details

\[\text{weight} = \text{weight} - \text{learning\_rate} \times (\text{gradient} + \text{wd} \times \text{weight})\]

However, if gradient is of “row\_sparse” storage type and “lazy\_update” is True, only the row slices whose indices appear in grad.indices are updated::

for row in gradient.indices: \(\text{weight}[\text{row}] = \text{weight}[\text{row}] - \text{learning\_rate} \times (\text{gradient}[\text{row}] + \text{wd} \times \text{weight}[\text{row}])\)

Defined in src/operator/optimizer_op.cc:L515

Value

out The result mx.nd.array
mx.nd.shape.array  

*Returns a 1D int64 array containing the shape of data.*

**Description**

Example::

**Arguments**

`data`  
NDArray-or-Symbol Input Array.

**Details**

`shape_array([[1,2,3,4], [5,6,7,8]]) = [2,4]`

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L573

**Value**

`out` The result mx.ndarray

mx.nd.shuffle  

*Randomly shuffle the elements.*

**Description**

This shuffles the array along the first axis. The order of the elements in each subarray does not change. For example, if a 2D array is given, the order of the rows randomly changes, but the order of the elements in each row does not change.

**Arguments**

`data`  
NDArray-or-Symbol Data to be shuffled.

**Value**

`out` The result mx.ndarray
mx.nd.sigmoid

Computes sigmoid of x element-wise.

Description

.. math:: y = 1 / (1 + \exp(-x))

Arguments

data NDArray-or-Symbol The input array.

Details

The storage type of “sigmoid“ output is always dense
Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L119

Value

out The result mx.ndarray

mx.nd.sign

Returns element-wise sign of the input.

Description

Example::

Arguments

data NDArray-or-Symbol The input array.

Details

sign([-2, 0, 3]) = [-1, 0, 1]
The storage type of “sign“ output depends upon the input storage type:
- sign(default) = default - sign(row_sparse) = row_sparse - sign(csr) = csr
Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L758

Value

out The result mx.ndarray
**Description**

.. math::

**Arguments**

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **lr**: float, required Learning rate
- **wd**: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).

**Details**

.. math::

It updates the weights using::

weight = weight - learning_rate * sign(gradient)

.. note:: - sparse ndarray not supported for this optimizer yet.

Defined in src/operator/optimizer_op.cc:L63

**Value**

out The result mx.ndarray
Arguments

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **mom**: NDArray-or-Symbol Momentum
- **lr**: float, required Learning rate
- **momentum**: float, optional, default=0 The decay rate of momentum estimates at each epoch.
- **wd**: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- **wd.lh**: float, optional, default=0 The amount of weight decay that does not go into gradient/momentum calculations otherwise do weight decay algorithmically only.

Details

\[ g_t = \nabla J(W_{t-1}); \ m_t = \beta m_{t-1} + (1 - \beta) g_t; \ W_t = W_{t-1} - \eta_t \text{sign}(m_t) \]

It updates the weights using:

- state = momentum * state + (1-momentum) * gradient
- weight = weight - learning_rate * sign(state)

Where the parameter "momentum" is the decay rate of momentum estimates at each epoch.

.. note:: - sparse ndarray not supported for this optimizer yet.

Defined in src/operator/optimizer_op.cc:L92

Value

- **out**: The result mx.ndarray

```
mx.nd.sin

Computes the element-wise sine of the input array.
```

Description

The input should be in radians (\(2\pi\text{ rad equals 360 degrees}\)).

Arguments

- **data**: NDArray-or-Symbol The input array.
**Details**

\[ \sin([0, \pi/4, \pi/2]) = [0, 0.707, 1] \]

The storage type of “sin” output depends upon the input storage type:
- \(\sin(\text{default}) = \text{default}\)
- \(\sin(\text{row_sparse}) = \text{row_sparse}\)
- \(\sin(\text{csr}) = \text{csr}\)

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L47

**Value**

out The result mx.ndarray

---

**mx.nd.sinh**

*Returns the hyperbolic sine of the input array, computed element-wise.*

**Description**

\[ \sinh(x) = 0.5 \times (\exp(x) - \exp(-x)) \]

**Arguments**

- **data**
  - NDArray-or-Symbol The input array.

**Details**

The storage type of “sinh” output depends upon the input storage type:
- \(\sinh(\text{default}) = \text{default}\)
- \(\sinh(\text{row_sparse}) = \text{row_sparse}\)
- \(\sinh(\text{csr}) = \text{csr}\)

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L371

**Value**

out The result mx.ndarray

---

**mx.nd.size.array**

*Returns a 1D int64 array containing the size of data.*

**Description**

Example:

**Arguments**

- **data**
  - NDArray-or-Symbol Input Array.
mx.nd.slice.axis

Details

size_array([[1,2,3,4], [5,6,7,8]]) = [8]

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L624

Value

out The result mx.ndarray

mx.nd.slice.axis Slices along a given axis. Returns an array slice along a given 'axis' starting from the 'begin' index to the 'end' index. Examples:

Out:

```
x = [[1., 2., 3., 4.], [5., 6., 7., 8.], [9., 10., 11., 12.]]
slice_axis(x, axis=0, begin=1, end=3) = [[5., 6., 7., 8.], [9., 10., 11., 12.]]
slice_axis(x, axis=1, begin=0, end=2) = [[1., 2.], [5., 6.], [9., 10.]]
slice_axis(x, axis=1, begin=-3, end=-1) = [[2., 3.], [6., 7.], [10., 11.]]
```

Description

Defined in src/operator/tensor/matrix_op.cc:L571

Arguments

data NDArray-or-Symbol Source input

axis int, required Axis along which to be sliced, supports negative indexes.

begin int, required The beginning index along the axis to be sliced, supports negative indexes.

end int or None, required The ending index along the axis to be sliced, supports negative indexes.

Value

out The result mx.ndarray
**mx.nd.slice_like**

Slices a region of the array like the shape of another array. This function is similar to “slice”, however, the ‘begin’ are always ‘0’s and ‘end’ of specific axes are inferred from the second input ‘shape_like’. Given the second ‘shape_like’ input of “shape=(d_0, d_1, ..., d_n-1)”, a “slice_like” operator with default empty ‘axes’, it performs the following operation: “out = slice(input, begin=(0, 0, ..., 0), end=(d_0, d_1, ..., d_n-1))”. When ‘axes’ is not empty, it is used to specify which axes are being sliced. Given a 4-d input data, “slice_like” operator with “axes=(0, 2, -1)” will perform the following operation: “out = slice(input, begin=(0, 0, 0, 0), end=(d_0, None, d_2, d_3))”. Note that it is allowed to have first and second input with different dimensions, however, you have to make sure the ‘axes’ are specified and not exceeding the dimension limits. For example, given ‘input_1’ with “shape=(2,3,4,5)” and ‘input_2’ with “shape=(1,2,3)”, it is not allowed to use: “out = slice_like(a, b)” because ndim of ‘input_1’ is 4, and ndim of ‘input_2’ is 3. The following is allowed in this situation: “out = slice_like(a, b, axes=(0, 2))”. Example:

```python
x = [[1., 2., 3., 4.],
     [5., 6., 7., 8.],
     [9., 10., 11., 12.]]
y = [[0., 0., 0.],
     [0., 0., 0.]]
slice_like(x, y) = [[1., 2., 3.], [5., 6., 7.]]
slice_like(x, y, axes=(0, 1)) = [[1., 2., 3.], [5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[1., 2., 3.], [5., 6., 7.], [9., 10., 11.]]
slice_like(x, y, axes=(-1)) = [[1., 2., 3., 4.], [5., 6., 7., 8.]]
slice_like(x, y, axes=(-1)) = [[1., 2., 3., 4.], [5., 6., 7., 8.], [9., 10., 11.]]
```

---

**Description**

Defined in src/operator/tensor/matrix_op.cc:L625

**Arguments**

- **data**
  - NDArray-or-Symbol Source input

- **shape_like**
  - NDArray-or-Symbol Shape like input

- **axes**
  - Shape(tuple), optional, default=[] List of axes on which input data will be sliced according to the corresponding size of the second input. By default will slice on all axes. Negative axes are supported.

**Value**

- **out** The result mx.nd.array
**mx.nd.SliceChannel**

Splits an array along a particular axis into multiple sub-arrays.

**Description**

.. note:: “SliceChannel“ is deprecated. Use “split” instead.

**Arguments**

- **data** : NDArray-or-Symbol The input
- **num_outputs** : int, required Number of splits. Note that this should evenly divide the length of the ‘axis’.
- **axis** : int, optional, default='1’ Axis along which to split.
- **squeeze.axis** : boolean, optional, default=0 If true, Removes the axis with length 1 from the shapes of the output arrays. **Note** that setting ‘squeeze_axis’ to “true” removes axis with length 1 only along the ‘axis’ which it is split. Also ‘squeeze_axis’ can be set to “true” only if “input.shape[axis] == num_outputs”.

**Details**

**Note** that ‘num_outputs’ should evenly divide the length of the axis along which to split the array.

Example::

x = [[[ 1.]] [[ 2.]] [[ 3.]] [[ 4.]] [[ 5.]] [[ 6.]]] x.shape = (3, 2, 1)
y = split(x, axis=1, num_outputs=2) // a list of 2 arrays with shape (3, 1, 1) y = [[[ 1.]] [[ 3.]] [[ 5.]]]
[[ 2.]] [[ 4.]] [[ 6.]]
y[0].shape = (3, 1, 1)
z = split(x, axis=0, num_outputs=3) // a list of 3 arrays with shape (1, 2, 1) z = [[[ 1.]] [[ 2.]]]
[[ 3.]] [[ 4.]]
[[ 5.]] [[ 6.]]]
z[0].shape = (1, 2, 1)
‘squeeze_axis=1‘ removes the axis with length 1 from the shapes of the output arrays. **Note** that setting ‘squeeze_axis’ to “1” removes axis with length 1 only along the ‘axis’ which it is split. Also ‘squeeze_axis’ can be set to true only if “input.shape[axis] == num_outputs”.

Example::

z = split(x, axis=0, num_outputs=3, squeeze_axis=1) // a list of 3 arrays with shape (2, 1) z = [[[ 1.]]
[[ 2.]]
[[ 3.]] [[ 4.]]
[[ 5.]] [[ 6.]]] z[0].shape = (2, 1)

Defined in src/operator/slice_channel.cc:L107
Value

out The result mx.ndarray

mx.nd.smooth.l1

*Calculate Smooth L1 Loss*(lhs, scalar) by summing*

Description

.. math::

Arguments

data

NDArray-or-Symbol source input

scalar

float scalar input

Details

\[
    f(x) = \begin{cases} \sigma x^2/2, & \text{if } x < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases}
\]

where :math:`x` is an element of the tensor *lhs* and :math:`\sigma` is the scalar.

Example::

    smooth_l1([1, 2, 3, 4]) = [0.5, 1.5, 2.5, 3.5]    smooth_l1([1, 2, 3, 4], scalar=1) = [0.5, 1.5, 2.5, 3.5]

Defined in src/operator/tensor/elemwise_binary_scalar_op_extended.cc:L108

Value

out The result mx.ndarray

mx.nd.Softmax

*Computes the gradient of cross entropy loss with respect to softmax output.*

Description

- This operator computes the gradient in two steps. The cross entropy loss does not actually need to be computed.
mx.nd.Softmax

Arguments

data : NDArray-or-Symbol
    Input array.

label : NDArray-or-Symbol
    Ground truth label.

grad.scale : float, optional, default=1
    Scales the gradient by a float factor.

ignore.label : float, optional, default=-1
    The instances whose *labels* == *ignore_label* will be ignored during backward, if `use.ignore` is set to “true”).

multi.output : boolean, optional, default=0
    If set to “true”, the softmax function will be computed along axis “1”. This is applied when the shape of input array differs from the shape of label array.

use.ignore : boolean, optional, default=0
    If set to “true”, the *ignore_label* value will not contribute to the backward gradient.

preserve.shape : boolean, optional, default=0
    If set to “true”, the softmax function will be computed along the last axis (“-1”).

normalization : 'batch', 'null', 'valid',optional, default='null'
    Normalizes the gradient.

out.grad : boolean, optional, default=0
    Multiplies gradient with output gradient element-wise.

smooth.alpha : float, optional, default=0
    Constant for computing a label smoothed version of cross-entropy for the backwards pass. This constant gets subtracted from the one-hot encoding of the gold label and distributed uniformly to all other labels.

Details

- Applies softmax function on the input array. - Computes and returns the gradient of cross entropy loss w.r.t. the softmax output.
- The softmax function, cross entropy loss and gradient is given by:
- Softmax Function:
  \[
  \text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}
  \]
- Cross Entropy Function:
  \[
  \text{CE}(label, output) = - \sum_i label_i \log(output_i)
  \]
- The gradient of cross entropy loss w.r.t softmax output:
  \[
  \text{gradient} = output - label
  \]
- During forward propagation, the softmax function is computed for each instance in the input array.
  For general *N*-D input arrays with shape \( (d_1, d_2, ..., d_n) \). The size is \( s = d_1 \cdot d_2 \cdot \cdot \cdot d_n \). We can use the parameters ‘preserve_shape’ and ‘multi_output’ to specify the way to compute softmax:
- By default, ‘preserve_shape’ is “false”. This operator will reshape the input array into a 2-D array with shape \( (d_1, \frac{s}{d_1}) \) and then compute the softmax function for each row in the reshaped array, and afterwards reshape it back to the original shape \( (d_1, d_2, ..., d_n) \).
- If ‘preserve_shape’ is “true”, the softmax function will be computed along the last axis (“axis” = “-1”). - If ‘multi_output’ is “true”, the softmax function will be computed along the second axis (“axis” = “1”).
- During backward propagation, the gradient of cross-entropy loss w.r.t softmax output array is computed. The provided label can be a one-hot label array or a probability label array.
- If the parameter ‘use_ignore’ is “true”, ‘ignore_label’ can specify input instances with a particular label to be ignored during backward propagation. **This has no effect when softmax ‘output’ has same shape as ‘label’**.

Example::

data = [[1,2,3,4],[2,2,2,2],[3,3,3,3],[4,4,4,4]] label = [1,0,2,3] ignore_label = 1 SoftmaxOutput(data=data, label = label, multi_output=true, use_ignore=true, ignore_label=ignore_label) ## forward softmax output 
[[ 0.0320586 0.08714432 0.23688284 0.64391428 ] [ 0.25 0.25 0.25 0.25 ] [ 0.25 0.25 0.25 0.25 ] [ 0.25 0.25 0.25 0.25 ]] ## backward gradient output 

## notice that the first row is all 0 because label[0] is 1, which is equal to ignore_label.

- The parameter ‘grad_scale’ can be used to rescale the gradient, which is often used to give each loss function different weights.
- This operator also supports various ways to normalize the gradient by ‘normalization’. The ‘normalization’ is applied if softmax output has different shape than the labels. The ‘normalization’ mode can be set to the followings:
  - “null” : do nothing. - “batch”: divide the gradient by the batch size. - “valid”: divide the gradient by the number of instances which are not ignored.

Defined in src/operator/softmax_output.cc:L231

Value

out The result mx.ndarray

mx.nd.softmax

Applies the softmax function.

Description

The resulting array contains elements in the range (0,1) and the elements along the given axis sum up to 1.

Arguments

data NDArray-or-Symbol The input array.
length NDArray-or-Symbol The length array.
axis int, optional, default=-1’ The axis along which to compute softmax.
temperature double or None, optional, default=None Temperature parameter in softmax
dtype None, ‘float16’, ‘float32’, ‘float64’,optional, default=None’ DType of the output in case this can’t be inferred. Defaults to the same as input’s dtype if not defined (dtype=None).
use.length boolean or None, optional, default=0 Whether to use the length input as a mask over the data input.
mx.nd.softmax.cross.entropy

Details

.. math:: \text{softmax}(\text{mathbf{z}}/t)_j = \frac{\text{exp}(\text{z}_j/t)}{\sum_{k=1}^{K} \text{exp}(\text{z}_k/t)}

for :math:`j = 1, ..., K`.

t is the temperature parameter in softmax function. By default, t equals 1.0

Example::

x = [[ 1. 1. 1.], [ 1. 1. 1.]]
softmax(x, axis=0) = [[ 0.5 0.5 0.5], [ 0.5 0.5 0.5]]
softmax(x, axis=1) = [[ 0.33333334, 0.33333334, 0.33333334], [ 0.33333334, 0.33333334, 0.33333334]]

Defined in src/operator/nn/softmax.cc:L134

Value

out The result mx.ndarray

mx.nd.softmax.cross.entropy

Calculate cross entropy of softmax output and one-hot label.

Description

- This operator computes the cross entropy in two steps:
  - Applies softmax function on the input array.
  - Computes and returns the cross entropy loss between the softmax output and the labels.

Arguments

data NDArray-or-Symbol Input data
label NDArray-or-Symbol Input label

Details

- The softmax function and cross entropy loss is given by:
  - Softmax Function:
    .. math:: \text{softmax}(\text{x})_i = \frac{\text{exp}(\text{x}_i)}{\sum_{j} \text{exp}(\text{x}_j)}
  - Cross Entropy Function:
    .. math:: \text{CE}(\text{label}, \text{output}) = - \sum_{i} \text{label}_i \log(\text{output}_i)

Example::

x = [[1, 2, 3], [11, 7, 5]]
label = [2, 0]
softmax(x) = [[0.09003057, 0.24472848, 0.66524094], [0.97962922, 0.01794253, 0.00242826]]
softmax_cross_entropy(data, label) = - \log(0.66524084) - \log(0.97962922) = 0.4281871

Defined in src/operator/loss_binary_op.cc:L59
mx.nd.SoftmaxActivation

Applies softmax activation to input. This is intended for internal layers.

Description
.. note::

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>NDArray-or-Symbol</td>
<td>The input array.</td>
</tr>
<tr>
<td>mode</td>
<td>'channel', 'instance', optional, default='instance'</td>
<td>Specifies how to compute the softmax. If set to &quot;instance&quot;, it computes softmax for each instance. If set to &quot;channel&quot;, It computes cross channel softmax for each position of each instance.</td>
</tr>
</tbody>
</table>

Details
This operator has been deprecated, please use ‘softmax’.

If ‘mode’ = “instance”, this operator will compute a softmax for each instance in the batch. This is the default mode.

If ‘mode’ = “channel”, this operator will compute a k-class softmax at each position of each instance, where ‘k’ = “num_channel”. This mode can only be used when the input array has at least 3 dimensions. This can be used for 'fully convolutional network', 'image segmentation', etc.

Example::

```python
gg> input_array = mx.nd.array([[3., 0.5, -0.5, 2., 7.],
gg> [2., -.4, 7., 3., 0.2]])
gg> softmax_act = mx.nd.SoftmaxActivation(input_array)
gg> print softmax_act.asnumpy() [[ 1.78322066e-02 1.46375655e-03 5.38485940e-04 6.56010211e-03 9.73605454e-01]
gg>  1.78379621e-02 1.08472735e-03]]
```

Defined in src/operator/nn/softmax_activation.cc:L59

Value

out The result mx.ndarray
**mx.nd.SoftmaxOutput**

Computes the gradient of cross entropy loss with respect to softmax output.

**Description**

- This operator computes the gradient in two steps. The cross entropy loss does not actually need to be computed.

**Arguments**

- **data**: NDArray-or-Symbol Input array.
- **label**: NDArray-or-Symbol Ground truth label.
- **grad.scale**: float, optional, default=1 Scales the gradient by a float factor.
- **ignore.label**: float, optional, default=-1 The instances whose 'labels' == 'ignore_label' will be ignored during backward, if 'use_ignore' is set to "true".
- **multi.output**: boolean, optional, default=0 If set to "true", the softmax function will be computed along axis "1". This is applied when the shape of input array differs from the shape of label array.
- **use.ignore**: boolean, optional, default=0 If set to "true", the 'ignore_label' value will not contribute to the backward gradient.
- **preserve.shape**: boolean, optional, default=0 If set to "true", the softmax function will be computed along the last axis ("-1").
- **normalization**: 'batch', 'null', 'valid',optional, default='null' Normalizes the gradient.
- **out.grad**: boolean, optional, default=0 Multiplies gradient with output gradient element-wise.
- **smooth.alpha**: float, optional, default=0 Constant for computing a label smoothed version of cross-entropy for the backwards pass. This constant gets subtracted from the one-hot encoding of the gold label and distributed uniformly to all other labels.

**Details**

- Applies softmax function on the input array. - Computes and returns the gradient of cross entropy loss w.r.t. the softmax output.
- The softmax function, cross entropy loss and gradient is given by:

  - Softmax Function:
    . math:: \text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}
  
  - Cross Entropy Function:
    . math:: \text{CE}(\text{label}, \text{output}) = - \sum_i \text{label}_i \log(\text{output}_i)
  
  - The gradient of cross entropy loss w.r.t softmax output:
    . math:: \text{gradient} = \text{output} - \text{label}
During forward propagation, the softmax function is computed for each instance in the input array.

For general $N$-D input arrays with shape $:(d_1, d_2, ..., d_n):$. The size is $:s=d_1 \cdot d_2 \cdot \cdot \cdot d_n:$. We can use the parameters ‘preserve_shape’ and ‘multi_output’ to specify the way to compute softmax:

- By default, ‘preserve_shape’ is “false”. This operator will reshape the input array into a 2-D array with shape $:(d_1, \frac{d_1}{2})$ and then compute the softmax function for each row in the reshaped array, and afterwards reshape it back to the original shape $:(d_1, d_2, ..., d_n):$.
- If ‘preserve_shape’ is “true”, the softmax function will be computed along the last axis (‘axis’ = “-1”). - If ‘multi_output’ is “true”, the softmax function will be computed along the second axis (‘axis’ = “1”).

During backward propagation, the gradient of cross-entropy loss w.r.t softmax output array is computed. The provided label can be a one-hot label array or a probability label array.

- If the parameter ‘use_ignore’ is “true”, ‘ignore_label’ can specify input instances with a particular label to be ignored during backward propagation. **This has no effect when softmax ‘output’ has same shape as ‘label’**.

Example:

data = [[1,2,3,4],[2,2,2,2],[3,3,3,3],[4,4,4,4]] label = [1,0,2,3] ignore_label = 1 SoftmaxOutput(data=data, label = label, multi_output=true, use_ignore=true, ignore_label=ignore_label) ## forward softmax output [[ 0.0320586 0.08714432 0.23688284 0.64391428] [ 0.25 0.25 0.25 0.25 ] [ 0.25 0.25 0.25 0.25 ] [ 0.25 0.25 0.25 0.25 ]] ## backward gradient output [[0. 0. 0. 0. [-0.75 0.25 0.25 0.25] [ 0.25 0.25 -0.75 0.25] [ 0.25 0.25 -0.75 0.25]]] ## notice that the first row is all 0 because label[0] is 1, which is equal to ignore_label.

- The parameter ‘grad_scale’ can be used to rescale the gradient, which is often used to give each loss function different weights.

- This operator also supports various ways to normalize the gradient by ‘normalization’. The ‘normalization’ is applied if softmax output has different shape than the labels. The ‘normalization’ mode can be set to the followings:
  - ”’null’”: do nothing. - ”’batch’”: divide the gradient by the batch size. - ”’valid’”: divide the gradient by the number of instances which are not ignored.

Defined in src/operator/softmax_output.cc:L231

Value

out The result mx.ndarray

mx.nd.softmax

Applies the softmax function.

Description

The resulting array contains elements in the range (0,1) and the elements along the given axis sum up to 1.
Arguments

data: NDArray-or-Symbol The input array.
axis: int, optional, default='-1' The axis along which to compute softmax.
temperature: double or None, optional, default=None Temperature parameter in softmax.
dtype: None, 'float16', 'float32', 'float64', optional, default=None DType of the output in case this can’t be inferred. Defaults to the same as input’s dtype if not defined (dtype=None).
use.length: boolean or None, optional, default=0 Whether to use the length input as a mask over the data input.

Details

.. math:: \text{softmin}(\mathbf{z}/t)_j = \frac{e^{-z_j}/t}{\sum_{k=1}^{K} e^{-z_k}/t}
for j = 1, ..., K

$\text{t}$ is the temperature parameter in softmax function. By default, $\text{t}$ equals 1.0

Example::

   x = [[ 1.  2.  3.]
   [ 3.  2.  1.]]
   softmin(x, axis=0) = [[ 0.88079703, 0.5, 0.11920292],
   [ 0.11920292, 0.5, 0.88079703]]
   softmin(x, axis=1) = [[ 0.66524094, 0.24472848, 0.09003057],
   [ 0.09003057, 0.24472848, 0.66524094]]

Defined in src/operator/nn/softmin.cc:L57

Value

out The result mx.ndarray

mx.nd.softsign

**Computes softsign of x element-wise.**

Description

.. math:: y = x / (1 + \text{abs}(x))

Arguments

data: NDArray-or-Symbol The input array.

Details

The storage type of “softsign” output is always dense

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L191

Value

out The result mx.ndarray
mx.nd.sort

Returns a sorted copy of an input array along the given axis.

Description

Examples:

Arguments

data NDArray-or-Symbol The input array
axis int or None, optional, default='-1' Axis along which to choose sort the input tensor. If not given, the flattened array is used. Default is -1.
is.ascend boolean, optional, default=1 Whether to sort in ascending or descending order.

Details

\[ x = \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} \]
// sorts along the last axis sort(x) = \[ 1., 4., 3., 1. \]
// flattens and then sorts sort(x, axis=None) = \[ 1., 1., 3., 4. \]
// sorts along the first axis sort(x, axis=0) = \[ 1., 1., 3., 4. \]
// in a descend order sort(x, is_ascend=0) = \[ 4., 1., 3., 1. \]

Defined in src/operator/tensor/ordering_op.cc:L133

Value

out The result mx.ndarray

mx.nd.space.to.depth

Rearranges(permutes) blocks of spatial data into depth. Similar to ONNX SpaceToDepth operator:
https://github.com/onnx/onnx/blob/master/docs/Operators.md#SpaceToDepth

The output is a new tensor where the values from height and width dimension are moved to the depth dimension. The reverse of this operation is “depth_to_space”. .. math:: \begin{align*}
    x' &= \text{reshape}(x, [N, C, H / \text{block\_size}, \text{block\_size}, W / \text{block\_size}, \text{block\_size}]) \\
    x'' &= \text{transpose}(x', [0, 3, 5, 1, 2, 4]) \\
    y &= \text{reshape}(x''', [N, C * (\text{block\_size}^2), H / \text{block\_size}, W / \text{block\_size}])
\end{align*}

where :math:`x` is an input tensor with default layout as :math:`[N, C, H, W]`: [batch, channels, height, width] and :math:`y` is the output tensor of layout :math:`[N, C * (\text{block\_size}^2), H / \text{block\_size}, W / \text{block\_size}]`. Example::

\[ x = \begin{bmatrix} \begin{bmatrix} 0, 6, 1, 7, 2, 8, 12, 18, 13, 19, 14, 20, 3, 9, 4, 10, 5, 11, 15, 21, 16, 22, 17, 23 \end{bmatrix} \end{bmatrix} \] space_to_depth(x, 2) = \[ \begin{bmatrix} \begin{bmatrix} 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23 \end{bmatrix} \end{bmatrix} \]
mx.nd.SpatialTransformer

Description

Defined in src/operator/tensor/matrix_op.cc:L1019

Arguments

data NDArray-or-Symbol Input ndarray
block.size int, required Blocks of [block_size, block_size] are moved

Value

out The result mx.ndarray

mx.nd.SpatialTransformer

Applies a spatial transformer to input feature map.

Description

Applies a spatial transformer to input feature map.

Arguments

data NDArray-or-Symbol Input data to the SpatialTransformerOp.
loc NDArray-or-Symbol localisation net, the output dim should be 6 when transform_type is affine. You should initialize the weight and bias with identity transform.
target.shape Shape(tuple), optional, default=[0,0] output shape(h, w) of spatial transformer: (y, x)
transform.type 'affine', required transformation type
sampler.type 'bilinear', required sampling type
cudnn.off boolean or None, optional, default=None whether to turn cudnn off

Value

out The result mx.ndarray
mx.nd.split

Splits an array along a particular axis into multiple sub-arrays.

Description

.. note:: “SliceChannel“ is deprecated. Use “split” instead.

Arguments

data
NDArray-or-Symbol The input

num.outputs
int, required Number of splits. Note that this should evenly divide the length of
the ‘axis’.

axis
int, optional, default=’1’ Axis along which to split.

squeeze.axis
boolean, optional, default=0 If true, Removes the axis with length 1 from the
shapes of the output arrays. **Note** that setting ‘squeeze_axis’ to “true” re-
moves axis with length 1 only along the ‘axis’ which it is split. Also ‘squeeze_axis’
can be set to “true” only if “input.shape[axis] == num_outputs”.

Details

**Note** that ‘num_outputs’ should evenly divide the length of the axis along which to split the
array.

Example::

x = [[[ 1.] [ 2.]] [[ 3.] [ 4.]] [[ 5.] [ 6.]]] x.shape = (3, 2, 1)
y = split(x, axis=1, num_outputs=2) // a list of 2 arrays with shape (3, 1, 1)
y = [[[ 1.]] [[ 3.]] [[ 5.]]]
[[[ 2.]] [[ 4.]] [[ 6.]]]
y[0].shape = (3, 1, 1)
z = split(x, axis=0, num_outputs=3) // a list of 3 arrays with shape (1, 2, 1)
z = [[[ 1.] [ 2.]]
[[ 3.] [ 4.]]
[[ 5.] [ 6.]]]
z[0].shape = (1, 2, 1)

‘squeeze_axis=1‘ removes the axis with length 1 from the shapes of the output arrays. **Note**
that setting ‘squeeze_axis’ to “1” removes axis with length 1 only along the ‘axis’ which it is split.
Also ‘squeeze_axis’ can be set to true only if “input.shape[axis] == num_outputs“.

Example::

z = split(x, axis=0, num_outputs=3, squeeze_axis=1) // a list of 3 arrays with shape (2, 1)
z = [[[ 1.]]
[[ 2.]]
[[ 3.] [ 4.]]
[[ 5.] [ 6.]]] z[0].shape = (2, 1 )

Defined in src/operator/slice_channel.cc:L107
mx.nd.sqrt

Value
out: The result mx.ndarray

mx.nd.sqrt
Returns element-wise square-root value of the input.

Description
.. math:: \sqrt{x} = \sqrt{x}

Arguments
data: NDArray-or-Symbol The input array.

Details
Example::
sqrt([4, 9, 16]) = [2, 3, 4]
The storage type of “sqrt” output depends upon the input storage type:
- sqrt(default) = default - sqrt(row_sparse) = row_sparse - sqrt(csr) = csr
Defined in src/operator/tensor/elementwise_unary_op_pow.cc:L170

mx.nd.square

Value
out: The result mx.ndarray

mx.nd.square
Returns element-wise squared value of the input.

Description
.. math:: \text{square}(x) = x^2

Arguments
data: NDArray-or-Symbol The input array.

Details
Example::
square([2, 3, 4]) = [4, 9, 16]
The storage type of “square” output depends upon the input storage type:
- square(default) = default - square(row_sparse) = row_sparse - square(csr) = csr
Defined in src/operator/tensor/elementwise_unary_op_pow.cc:L119
mx.nd.squeeze

Remove single-dimensional entries from the shape of an array. Same behavior of defining the output tensor shape as numpy.squeeze for the most of cases. See the following note for exception. Examples:

```python
data = [[0], [1], [2]]
squeeze(data) = [0, 1, 2]
squeeze(data, axis=0) = [[0], [1], [2]]
squeeze(data, axis=2) = [0, 1, 2]
```

Note:: The output of this operator will keep at least one dimension not removed. For example, `squeeze([[4]]) = [4]`, while in numpy.squeeze, the output will become a scalar.

Arguments

- **data**
  NDArray-or-Symbol data to squeeze
- **axis**
  Shape or None, optional, default=None Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.

Value

out The result mx.ndarray

mx.nd.stack

Join a sequence of arrays along a new axis. The axis parameter specifies the index of the new axis in the dimensions of the result. For example, if axis=0 it will be the first dimension and if axis=-1 it will be the last dimension. Examples:

```python
x = [1, 2] y = [3, 4] stack(x, y) = [[1, 2], [3, 4]]
stack(x, y, axis=1) = [[1, 3], [2, 4]]
```

Description

Join a sequence of arrays along a new axis. The axis parameter specifies the index of the new axis in the dimensions of the result. For example, if axis=0 it will be the first dimension and if axis=-1 it will be the last dimension. Examples:

```python
x = [1, 2] y = [3, 4] stack(x, y) = [[1, 2], [3, 4]]
stack(x, y, axis=1) = [[1, 3], [2, 4]]
```
**mx.nd.stop.gradient**

**Arguments**

- **data** NDArray-or-Symbol[] List of arrays to stack
- **axis** int, optional, default=`'0'` The axis in the result array along which the input arrays are stacked.
- **num. args** int, required Number of inputs to be stacked.

**Value**

- **out** The result mx.ndarray

---

**mx.nd.stop.gradient**  
*Stops gradient computation.*

**Description**

Stops the accumulated gradient of the inputs from flowing through this operator in the backward direction. In other words, this operator prevents the contribution of its inputs to be taken into account for computing gradients.

**Arguments**

- **data** NDArray-or-Symbol The input array.

**Details**

Example::

v1 = [1, 2] v2 = [0, 1] a = Variable('a') b = Variable('b') b_stop_grad = stop_gradient(3 * b) loss = MakeLoss(b_stop_grad + a)
executor = loss.simple_bind(ctx=cpu(), a=(1,2), b=(1,2)) executor.forward(is_train=True, a=v1, b=v2)
executor.outputs [ 1. 5.]
executor.backward() executor.grad_arrays [ 0. 0. ] [ 1. 1.]

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L325

**Value**

- **out** The result mx.ndarray
mx.nd.sum

Computes the sum of array elements over given axes.

Description

.. Note::

Arguments

data NDArray-or-Symbol The input
axis Shape or None, optional, default=none The axis or axes along which to perform the reduction.
The default, `axis=()` will compute over all elements into a scalar array with shape `(1,)`.
If `axis` is int, a reduction is performed on a particular axis.
If `axis` is a tuple of int, a reduction is performed on all the axes specified in the tuple.
If `exclude` is true, reduction will be performed on the axes that are NOT in axis instead.
Negative values means indexing from right to left.
keepdims boolean, optional, default=0 If this is set to `True`, the reduced axes are left in the result as dimension with size one.
exclude boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.

Details

`sum` and `sum_axis` are equivalent. For ndarray of csr storage type summation along axis 0 and axis 1 is supported. Setting keepdims or exclude to True will cause a fallback to dense operator.

Example::
data = [[1, 2], [2, 3], [1, 3]], [[1, 4], [4, 3], [5, 2]], [[7, 1], [7, 2], [7, 3]]
sum(data, axis=1) [[ 4.  8.]
  [10.  9.]
  [21.  6.]]
sum(data, axis=[1,2]) [ 12. 19. 27.]
data = [[1, 2, 0], [3, 0, 1], [4, 1, 0]]
csr = cast_storage(data, 'csr')
sum(csr, axis=0) [ 8.  3.  1.]
sum(csr, axis=1) [ 3.  4.  5.]

Defined in src/operator/tensor/broadcast_reduce_sum_value.cc:L67

Value

out The result mx.ndarray
mx.nd.sum.axis

Computes the sum of array elements over given axes.

Description

.. Note::

Arguments

data NDArray-or-Symbol The input

axis Shape or None, optional, default=None The axis or axes along which to perform the reduction.
The default, ‘axis=()’, will compute over all elements into a scalar array with shape ‘(1,)’.
If ‘axis’ is int, a reduction is performed on a particular axis.
If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple.
If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead.
Negative values means indexing from right to left.

keepdims boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.

exclude boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.

Details

‘sum’ and ‘sum_axis’ are equivalent. For ndarray of csr storage type summation along axis 0 and axis 1 is supported. Setting keepdims or exclude to True will cause a fallback to dense operator.

Example::

data = [[[1, 2], [2, 3], [1, 3]], [[1, 4], [4, 3], [5, 2]], [[7, 1], [7, 2], [7, 3]]]
sum(data, axis=1) [[4. 8.] [10. 9.] [21. 6.]]
sum(data, axis=[1,2]) [12. 19. 27.]
data = [[1, 2, 0], [3, 0, 1], [4, 1, 0]]
csr = cast_storage(data, 'csr')
sum(csr, axis=0) [8. 3. 1.]
sum(csr, axis=1) [3. 4. 5.]

Defined in src/operator/tensor/broadcast_reduce_sum_value.cc:L67

Value

out The result mx.ndarray
mx.nd.SVMOutput

Computes support vector machine based transformation of the input.

Description

This tutorial demonstrates using SVM as output layer for classification instead of softmax: https://github.com/dmlc/mxnet/tree/master/example/svm_mnist.

Arguments

- **data**: NDArray-or-Symbol Input data for SVM transformation.
- **label**: NDArray-or-Symbol Class label for the input data.
- **margin**: float, optional, default=1 The loss function penalizes outputs that lie outside this margin. Default margin is 1.
- **regularization.coefficient**: float, optional, default=1 Regularization parameter for the SVM. This balances the tradeoff between coefficient size and error.
- **use.linear**: boolean, optional, default=0 Whether to use L1-SVM objective. L2-SVM objective is used by default.

Value

- **out**: The result mx.ndarray

mx.nd.swapaxes

Interchanges two axes of an array.

Description

Examples:

Arguments

- **data**: NDArray-or-Symbol Input array.
- **dim1**: int, optional, default='0' the first axis to be swapped.
- **dim2**: int, optional, default='0' the second axis to be swapped.

Details

- x = [[1, 2, 3]]) swapaxes(x, 0, 1) = [[ 1], [ 2], [ 3]]
- x = [[[ 0, 1], [ 2, 3]], [[ 4, 5], [ 6, 7]]] // (2,2,2) array
- swapaxes(x, 0, 2) = [[[ 0, 4], [ 2, 6]], [[ 1, 5], [ 3, 7]]]

Defined in src/operator/swapaxis.cc:L70

Value

- **out**: The result mx.ndarray
\textbf{mx.nd.SwapAxis} \hspace{1cm} \textit{Interchanges two axes of an array.}

**Description**

Examples:

**Arguments**

- data \text{NDArray-or-Symbol} \text{Input array.}
- dim1 \text{int, optional, default='0'} \text{the first axis to be swapped.}
- dim2 \text{int, optional, default='0'} \text{the second axis to be swapped.}

**Details**

\texttt{x = [[1, 2, 3]], swapaxes(x, 0, 1) = [[1], [2], [3]]}
\texttt{x = [[[0, 1], [2, 3]], [[4, 5], [6, 7]]], (2,2,2) array}
\texttt{swapaxes(x, 0, 2) = [[[0, 4], [2, 6]], [[1, 5], [3, 7]]]}

Defined in src/operator/swapaxis.cc:L70

**Value**

- out \text{The result mx.ndarray}

---

\textbf{mx.nd.take} \hspace{1cm} \textit{Takes elements from an input array along the given axis.}

**Description**

This function slices the input array along a particular axis with the provided indices.

**Arguments**

- a \text{NDArray-or-Symbol} \text{The input array.}
- indices \text{NDArray-or-Symbol} \text{The indices of the values to be extracted.}
- axis \text{int, optional, default='0'} \text{The axis of input array to be taken. For input tensor of rank r, it could be in the range of [-r, r-1]}
- mode \text{\{'clip', 'raise', 'wrap', \}optional, default='clip'} \text{Specify how out-of-bound indices behave. Default is "clip". "clip" means clip to the range. So, if all indices mentioned are too large, they are replaced by the index that addresses the last element along an axis. "wrap" means to wrap around. "raise" means to raise an error when index out of range.}
Details

Given data tensor of rank r >= 1, and indices tensor of rank q, gather entries of the axis dimension of
data (by default outer-most one as axis=0) indexed by indices, and concatenates them in an output
tensor of rank q + (r - 1).

Examples::

```
x = [4. 5. 6.]
// Trivial case, take the second element along the first axis.
take(x, [1]) = [ 5. ]
// The other trivial case, axis=-1, take the third element along the first axis
take(x, [3], axis=-1, mode='clip') = [ 6. ]
x = [[ 1., 2.], [ 3., 4.], [ 5., 6.]]
// In this case we will get rows 0 and 1, then 1 and 2. Along axis 0
take(x, [[0,1],[1,2]]) = [[[ 1. , 2. ], [ 3. , 4. ]],
[[ 3. , 4. ], [ 5. , 6. ]]]
// In this case we will get rows 0 and 1, then 1 and 2 (calculated by wrapping around). // Along axis 1
take(x, [[0, 3], [-1, -2]], axis=1, mode='wrap') = [[[ 1. , 2. ], [ 2. , 1. ]],
[[ 3. , 4. ], [ 4. , 3. ]],
[[ 5. , 6. ], [ 6. , 5. ]]]
```

The storage type of “take“ output depends upon the input storage type:
- take(default, default) = default - take(csr, default, axis=0) = csr

Defined in src/operator/tensor/indexing_op.cc:L782

Value

```
out The result mx.ndarray
```

mx.nd.tan

*Computes the element-wise tangent of the input array.*

Description

The input should be in radians (:math:`2\pi` rad equals 360 degrees).

Arguments

```
data NDArray-or-Symbol The input array.
```
mx.nd.tanh

Details

.. math:: \tan([0, \pi/4, \pi/2]) = [0, 1, -\inf]

The storage type of “tan” output depends upon the input storage type:

- tan(default) = default - tan(row_sparse) = row_sparse - tan(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L140

Value

out The result mx.ndarray

mx.nd.tanh

Returns the hyperbolic tangent of the input array, computed element-wise.

Description

.. math:: \tanh(x) = \sinh(x) / \cosh(x)

Arguments

data NDArray-or-Symbol The input array.

Details

The storage type of “tanh“ output depends upon the input storage type:

- tanh(default) = default - tanh(row_sparse) = row_sparse - tanh(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L451

Value

out The result mx.ndarray
Repeats the whole array multiple times. If “reps” has length *d*, and input array has dimension of *n*. There are three cases: - **n=d**. Repeat *i*-th dimension of the input by “reps[i]” times::

```latex
\text{tile}(x, \text{reps}=(2,3)) = [[ 1., 2., 1., 2., 1.], [ 3., 4., 3., 4., 3., 4.], [ 1., 2., 1., 2., 1.], [ 3., 4., 3., 4., 3., 4.]]
```

- **n>d**. “reps” is promoted to length *n* by pre-pending 1’s to it. Thus for an input shape “(2,3)”, “reps=(2,)” is treated as “(1,2)”:·

```latex
\text{tile}(x, \text{reps}=(2)) = [[ 1., 2., 1., 2.], [ 3., 4., 3., 4.]]
```

- **n<d**. The input is promoted to be d-dimensional by prepending new axes. So a shape “(2,2)” array is promoted to “(1,2,2)” for 3-D replication:

```latex
```

mx.nd.topk

Returns the indices of the top *k* elements in an input array along the given axis (by default). If ret_type is set to ‘value’ returns the value of top *k* elements (instead of indices). In case of ret_type = ‘both’, both value and index would be returned. The returned elements will be sorted.

Examples::

Arguments

- **data**: NDArray-or-Symbol The input array.
- **axis**: int or None, optional, default='-1' Axis along which to choose the top k indices. If not given, the flattened array is used. Default is -1.
- **k**: int, optional, default='1' Number of top elements to select, should be always smaller than or equal to the element number in the given axis. A global sort is performed if set \( k < 1 \).
- **ret_typ**: 'both', 'indices', 'mask', 'value', optional, default='indices' The return type. "value" means to return the top k values, "indices" means to return the indices of the top k values, "mask" means to return a mask array containing 0 and 1. 1 means the top k values. "both" means to return a list of both values and indices of top k elements.
- **is_ascend**: boolean, optional, default=0 Whether to choose k largest or k smallest elements. Top K largest elements will be chosen if set to false.
- **dtype**: 'float16', 'float32', 'float64', 'int32', 'int64', 'uint8', optional, default='float32' Dtype of the output indices when ret_typ is "indices" or "both". An error will be raised if the selected data type cannot precisely represent the indices.

Details

```
x = [[0.3, 0.2, 0.4], [0.1, 0.3, 0.2]]
// returns an index of the largest element on last axis
topk(x) = [[2.], [1.]]
// returns the value of top-2 largest elements on last axis
topk(x, ret_typ='value', k=2) = [[0.4, 0.3], [0.3, 0.2]]
// returns the value of top-2 smallest elements on last axis
topk(x, ret_typ='value', k=2, is_ascend=1) = [[0.2, 0.3], [0.1, 0.2]]
// returns the value of top-2 largest elements on axis 0
//flattens and then returns list of both values and indices
topk(x, axis=0, ret_typ='both', k=2) = [[[0.4, 0.3], [0.3, 0.2]], [[2., 0.], [1., 2.]]]
```

Defined in src/operator/tensor/ordering_op.cc:L68

Value

- **out**: The result mx.ndarray

---

**mx.nd.transpose**

Permut the dimensions of an array. Examples:

```
x = [[1, 2], [3, 4]]
transpose(x) = [[1, 3], [2, 4]]
x = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
transpose(x) = [[[1, 5], [3, 7]], [[2, 6], [4, 8]]]
transpose(x, axes=(1,0,2)) = [[[1, 2], [5, 6]], [[3, 4], [7, 8]]]
```

Description

Defined in src/operator/tensor/matrix_op.cc:L328

Arguments

data NDArray-or-Symbol Source input
axes Shape(tuple), optional, default=[] Target axis order. By default the axes will be inverted.

Value

out The result mx.ndarray

mx.nd.trunc

Return the element-wise truncated value of the input.

Description

The truncated value of the scalar x is the nearest integer i which is closer to zero than x is. In short, the fractional part of the signed number x is discarded.

Arguments

data NDArray-or-Symbol The input array.

Details

Example::

trunc([-2.1, -1.9, 1.5, 1.9, 2.1]) = [-2., -1., 1., 1., 2.]

The storage type of “trunc” output depends upon the input storage type:
- trunc(default) = default - trunc(row_sparse) = row_sparse - trunc(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L856

Value

out The result mx.ndarray
mx.nd.uniform

**Draw random samples from a uniform distribution.**

**Description**

.. note:: The existing alias “uniform” is deprecated.

**Arguments**

- **low**: float, optional, default=0 Lower bound of the distribution.
- **high**: float, optional, default=1 Upper bound of the distribution.
- **shape**: Shape(tuple), optional, default=None Shape of the output.
- **ctx**: string, optional, default="" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

**Details**

Samples are uniformly distributed over the half-open interval *[low, high)* (includes *low*, but excludes *high*).

**Example:**

```
uniform(low=0, high=1, shape=(2,2)) = [[ 0.60276335, 0.85794562], [ 0.54488319, 0.84725171]]
```

Defined in src/operator/random/sample_op.cc:L96

**Value**

- **out**: The result mx.ndarray

mx.nd.unravel.index

**Converts an array of flat indices into a batch of index arrays.** The operator follows numpy conventions so a single multi index is given by a column of the output matrix. The leading dimension may be left unspecified by using -1 as placeholder.

**Description**

**Examples:**

**Arguments**

- **data**: NDArray-or-Symbol Array of flat indices
- **shape**: Shape(tuple), optional, default=None Shape of the array into which the multi-indices apply.
mx.nd.UpSampling

Details

A = [22,41,37] unravel(A, shape=(7,6)) = [[3,6,6],[4,5,1]]
unravel(A, shape=(-1,6)) = [[3,6,6],[4,5,1]]

Defined in src/operator/tensor/ravel.cc:L68

Value

out The result mx.ndarray

mx.nd.UpSampling Upsamples the given input data.

Description

Two algorithms (“sample_type“) are available for upsampling:

Arguments

data NDArray-or-Symbol[] Array of tensors to upsample. For bilinear upsampling, there should be 2 inputs - 1 data and 1 weight.
scale int, required Up sampling scale
num.filter int, optional, default='0' Input filter. Only used by bilinear sample_type.Since bilinear upsampling uses deconvolution, num_filters is set to the number of channels.
sample.type 'bilinear', 'nearest', required upsampling method
multi.input.mode 'concat', 'sum',optional, default='concat' How to handle multiple input. concat means concatenate upsampled images along the channel dimension. sum means add all images together, only available for nearest neighbor upsampling.
num.args int, required Number of inputs to be upsampled. For nearest neighbor upsampling, this can be 1-N; the size of output will be(scale*h_0, scale*w_0) and all other inputs will be upsampled to the same size. For bilinear upsampling this must be 2; 1 input and 1 weight.
workspace long (non-negative), optional, default=512 Tmp workspace for deconvolution (MB)

Details

- Nearest Neighbor - Bilinear

**Nearest Neighbor Upsampling**

Input data is expected to be NCHW.
Example::

x = [[[1. 1. 1.] [1. 1. 1.] [1. 1. 1.]]]
UpSampling(x, scale=2, sample_type='nearest') = [[[1. 1. 1. 1. 1. 1.] [1. 1. 1. 1. 1. 1.] [1. 1. 1. 1. 1. 1.]]]

**Bilinear Upsampling**

Uses ‘deconvolution’ algorithm under the hood. You need provide both input data and the kernel.
Input data is expected to be NCHW.
‘num_filter’ is expected to be same as the number of channels.

Example::

x = [[[1. 1. 1.] [1. 1. 1.] [1. 1. 1.]]]
w = [[[1. 1. 1. 1.] [1. 1. 1. 1.] [1. 1. 1. 1.] [1. 1. 1. 1.]]]
UpSampling(x, w, scale=2, sample_type='bilinear', num_filter=1) = [[[2. 2. 2. 2. 2. 1.] [2. 2. 2. 2. 2. 1.] [2. 2. 2. 2. 2. 1.]]]

Defined in src/operator/nn/upsampling.cc:L173

---

mx.nd.where

Return the elements, either from x or y, depending on the condition.

**Description**

Given three ndarrays, condition, x, and y, return an ndarray with the elements from x or y, depending on the elements from condition are true or false. x and y must have the same shape. If condition has the same shape as x, each element in the output array is from x if the corresponding element in the condition is true, and from y if false.

**Arguments**

- **condition**
  - NDArray-or-Symbol condition array
- **x**
  - NDArray-or-Symbol
- **y**
  - NDArray-or-Symbol

**Details**

If condition does not have the same shape as x, it must be a 1D array whose size is the same as x’s first dimension size. Each row of the output array is from x’s row if the corresponding element from condition is true, and from y’s row if false.

Note that all non-zero values are interpreted as “True“ in condition.

Examples::

x = [[1, 2], [3, 4]] y = [[5, 6], [7, 8]] cond = [[0, 1], [-1, 0]]
where(cond, x, y) = [[5, 2], [3, 8]]
csr_cond = cast_storage(cond, 'csr')
where(csr_cond, x, y) = [[5, 2], [3, 8]]
Defined in src/operator/tensor/control_flow_op.cc:L57

Value

out The result mx.ndarray

mx.nd.zeros

*Generate an mx.nd.array object with zeros*

**Description**

Generate an mx.nd.array object with zeros

**Usage**

mx.nd.zeros(shape, ctx = NULL)

**Arguments**

shape

the dimension of the mx.nd.array

cxt

optional The context device of the array. mx.ctx.default() will be used in default.

**Examples**

mat = mx.nd.zeros(10)
as.array(mat)
mat2 = mx.nd.zeros(c(5,5))
as.array(mat)
mat3 = mx.nd.zeros(c(3,3,3))
as.array(mat3)

mx.nd.zeros_like

*Return an array of zeros with the same shape, type and storage type as the input array.*

**Description**

The storage type of “zeros_like” output depends on the storage type of the input

**Arguments**

data

NDArray-or-Symbol The input
Details

- zeros_like(row_sparse) = row_sparse - zeros_like(csr) = csr - zeros_like(default) = default

Examples:

\[ x = \begin{bmatrix} 1., 1., 1. \\ 1., 1., 1. \end{bmatrix} \]
\[ \text{zeros_like}(x) = \begin{bmatrix} 0., 0., 0. \\ 0., 0., 0. \end{bmatrix} \]

Value

out The result mx.ndarray

mx.opt.adadelta  Create an AdaDelta optimizer with respective parameters.

Description


Usage

mx.opt.adadelta(
    rho = 0.9,
    epsilon = 1e-05,
    wd = 0,
    rescale.grad = 1,
    clip_gradient = -1
)

Arguments

rho float, default=0.90 Decay rate for both squared gradients and delta x.
epsilon float, default=1e-5 The constant as described in the thesis.
wd float, default=0.0 L2 regularization coefficient add to all the weights.
rescale.grad float, default=1 rescaling factor of gradient.
clip_gradient float, default=-1 (no clipping if < 0) clip gradient in range [-clip_gradient, clip_gradient].
mx.opt.adagrad

Create an AdaGrad optimizer with respective parameters. AdaGrad optimizer of Duchi et al., 2011,

Description

This code follows the version in http://arxiv.org/pdf/1212.5701v1.pdf Eq(5) by Matthew D. Zeiler, 2012. AdaGrad will help the network to converge faster in some cases.

Usage

mx.opt.adagrad(
    learning.rate = 0.05,
    epsilon = 1e-08,
    wd = 0,
    rescale.grad = 1,
    clip_gradient = -1,
    lr_scheduler = NULL
)

Arguments

learning.rate float, default=0.05 Step size.
epsilon float, default=1e-8
wd float, default=0.0 L2 regularization coefficient add to all the weights.
rescale.grad float, default=1.0 rescaling factor of gradient.
clip_gradient float, default=1.0 (no clipping if < 0) clip gradient in range [-clip_gradient, clip_gradient].
lr_scheduler function, optional The learning rate scheduler.

mx.opt.adam

Create an Adam optimizer with respective parameters. Adam optimizer as described in [King2014].

Description

Usage

mx.opt.adam(
    learning.rate = 0.001,
    beta1 = 0.9,
    beta2 = 0.999,
    epsilon = 1e-08,
    wd = 0,
    rescale.grad = 1,
    clip_gradient = -1,
    lr_scheduler = NULL
)

Arguments

learning.rate float, default=1e-3 The initial learning rate.
beta1 float, default=0.9 Exponential decay rate for the first moment estimates.
beta2 float, default=0.999 Exponential decay rate for the second moment estimates.
epsilon float, default=1e-8
wd float, default=0.0 L2 regularization coefficient add to all the weights.
rescale.grad float, default=1.0 rescaling factor of gradient.
clip_gradient float, optional, default=-1 (no clipping if < 0) clip gradient in range [-clip_gradient, clip_gradient].
lr_scheduler function, optional The learning rate scheduler.

mx.opt.create

Create an optimizer by name and parameters

Description

Create an optimizer by name and parameters

Usage

mx.opt.create(name, ...)

Arguments

name The name of the optimizer
... Additional arguments
mx.opt.get.updater

Get an updater closure that can take list of weight and gradient and return updated list of weight.

Description

Get an updater closure that can take list of weight and gradient and return updated list of weight.

Usage

mx.opt.get.updater(optimizer, weights, ctx)

Arguments

optimizer The optimizer
weights The weights to be optimized

mx.opt.nag

Create a Nesterov Accelerated SGD (NAG) optimizer.

Description


Usage

mx.opt.nag(
    learning.rate = 0.01,
    momentum = 0,
    wd = 0,
    rescale.grad = 1,
    clip_gradient = -1,
    lr_scheduler = NULL
)

Arguments

learning.rate float, default=0.01 The initial learning rate.
momentum float, default=0 The momentum value
wd float, default=0.0 L2 regularization coefficient added to all the weights.
rescale.grad float, default=1.0 rescaling factor of gradient.
clip_gradient float, optional, default=-1 (no clipping if < 0) clip gradient in range [-clip_gradient, clip_gradient].
lr_scheduler function, optional The learning rate scheduler.

**Description**


**Usage**

```r
mx.opt.rmsprop(
    learning.rate = 0.002,
    centered = TRUE,
    rho = 0.95,
    momentum = 0.9,
    epsilon = 1e-04,
    wd = 0,
    rescale.grad = 1,
    clip_gradient = -1,
    lr_scheduler = NULL
)
```

**Arguments**

- **learning.rate** float, default=0.002 The initial learning rate.
- **rho** float, default=0.95 decay factor of moving average for gradient, gradient^2.
- **momentum** float, default=0.9 "momentum" factor.
- **epsilon** float, default=1e-4
- **wd** float, default=0.0 L2 regularization coefficient add to all the weights.
- **rescale.grad** float, default=1.0 rescaling factor of gradient.
- **clip_gradient** float, optional, default=-1 (no clipping if < 0) clip gradient in range [-clip_gradient, clip_gradient].
- **lr_scheduler** function, optional The learning rate scheduler.
mx.opt.sgd

*Create an SGD optimizer with respective parameters. Perform SGD with momentum update*

**Description**

Create an SGD optimizer with respective parameters. Perform SGD with momentum update

**Usage**

```python
mx.opt.sgd(
    learning.rate = 0.01,
    momentum = 0,
    wd = 0,
    rescale.grad = 1,
    clip_gradient = -1,
    lr_scheduler = NULL
)
```

**Arguments**

- **learning.rate**: float, default=0.01 The initial learning rate.
- **momentum**: float, default=0 The momentum value
- **wd**: float, default=0.0 L2 regularization coefficient add to all the weights.
- **rescale.grad**: float, default=1.0 rescaling factor of gradient.
- **clip_gradient**: float, optional, default=-1 (no clipping if < 0) clip gradient in range [-clip_gradient, clip_gradient].
- **lr_scheduler**: function, optional The learning rate scheduler.

---

mx.profiler.config

*Set up the configuration of profiler*

**Description**

Set up the configuration of profiler.

**Usage**

```python
mx.profiler.config(params)
```
**Arguments**

flags

- list of key/value pair tuples. Indicates configuration parameters
  - profile_symbolic: boolean, whether to profile symbolic operators
  - profile_imperative: boolean, whether to profile imperative operators
  - profile_memory: boolean, whether to profile memory usage
  - profile_api: boolean, whether to profile the C API
  - file_name: string, output file for profile data
  - continuous_dump: boolean, whether to periodically dump profiling data to file
  - dump_period: float, seconds between profile data dumps

```
mx.profiler.state(state = MX.PROF.STATE$STOP)
```

**Description**

Set up the profiler state to record operator.

**Usage**

```
mx.profiler.state(state = MX.PROF.STATE$STOP)
```

**Arguments**

- state: Indicating whether to run the profiler, can be ’MX.PROF.STATE$RUN’ or ’MX.PROF.STATE$STOP’. Default is ’MX.PROF.STATE$STOP’.
- filename: The name of output trace file. Default is ’profile.json’

```
mx.rnorm(shape, mean = 0, sd = 1, ctx = NULL)
```

**Description**

Generate normal distribution with mean and sd.

**Usage**

```
mx.rnorm(shape, mean = 0, sd = 1, ctx = NULL)
```

**Arguments**

- shape: Dimension, The shape(dimension) of the result.
- mean: numeric, The mean of distribution.
- sd: numeric, The standard deviations.
- ctx: optional The context device of the array. mx.ctx.default() will be used in default.
Examples

```r
mx.set.seed(0)
as.array(mx.runif(2))
# 0.5488135 0.5928446
mx.set.seed(0)
as.array(mx.rnorm(2))
# 2.212206  1.163079
```

---

**mx.runif**

Generate uniform distribution in \([low, high)\) with specified shape.

### Description

Generate uniform distribution in \([low, high)\) with specified shape.

### Usage

```r
mx.runif(shape, min = 0, max = 1, ctx = NULL)
```

### Arguments

- **shape**: Dimension, The shape(dimension) of the result.
- **min**: numeric, The lower bound of distribution.
- **max**: numeric, The upper bound of distribution.
- **ctx**: optional The context device of the array. `mx.ctx.default()` will be used in default.

### Examples

```r
mx.set.seed(0)
as.array(mx.runif(2))
# 0.5488135 0.5928446
mx.set.seed(0)
as.array(mx.rnorm(2))
# 2.212206 1.163079
```
**mx.serialize**

Serialize MXNet model into RData-compatible format.

**Description**

Serialize MXNet model into RData-compatible format.

**Usage**

```r
mx.serialize(model)
```

**Arguments**

- `model` The mxnet model

---

**mx.set.seed**

Set the seed used by mxnet device-specific random number generators.

**Description**

Set the seed used by mxnet device-specific random number generators.

**Usage**

```r
mx.set.seed(seed)
```

**Arguments**

- `seed` the seed value to the device random number generators.

**Details**

We have a specific reason why `mx.set.seed` is introduced, instead of simply use `set.seed`. The reason that is that most of mxnet random number generator can run on different devices, such as GPU. We need to use massively parallel PRNG on GPU to get fast random number generations. It can also be quite costly to seed these PRNGs. So we introduced `mx.set.seed` for mxnet specific device random numbers.

**Examples**

```r
mx.set.seed(0)
as.array(mx.runif(2))
# 0.5488135 0.5928446
mx.set.seed(0)
as.array(mx.rnorm(2))
# 2.212206 1.163079
```
mx.simple.bind

*Simple bind the symbol to executor, with information from input shapes.*

**Description**

Simple bind the symbol to executor, with information from input shapes.

**Usage**

`mx.simple.bind(symbol, ctx, grad.req = "null", fixed.param = NULL, ...)`

mx.symbol.abs

*abs:* Returns element-wise absolute value of the input.

**Description**

Example::

**Usage**

`mx.symbol.abs(...)`

**Arguments**

- **data**: NDArray-or-Symbol The input array.
- **name**: string, optional Name of the resulting symbol.

**Details**

`abs([-2, 0, 3]) = [2, 0, 3]`

The storage type of “abs” output depends upon the input storage type:

- abs(default) = default - abs(row_sparse) = row_sparse - abs(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L720

**Value**

`out` The result mx.symbol
mx.symbol.Activation

**Activation**

Applies an activation function element-wise to the input.

**Description**

The following activation functions are supported:

- 'relu': Rectified Linear Unit, \( y = \max(x, 0) \)
- 'sigmoid': \( y = \frac{1}{1 + \exp(-x)} \)
- 'tanh': Hyperbolic tangent, \( y = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \)
- 'softrelu': Soft ReLU, or SoftPlus, \( y = \log(1 + \exp(x)) \)
- 'softsign': \( y = \frac{x}{1 + \abs{x}} \)

Defined in src/operator/nn/activation.cc:L165

**Usage**

```python
mx.symbol.Activation(...)```

**Arguments**

- **data**: NDArray-or-Symbol The input array.
- **act.type**: 'relu', 'sigmoid', 'softrelu', 'softsign', 'tanh', required Activation function to be applied.
- **name**: string, optional Name of the resulting symbol.

**Value**

- **out**: The result mx.symbol

mx.symbol.adam_update

**adam_update**

Update function for Adam optimizer. Adam is seen as a generalization of AdaGrad.

**Description**

Adam update consists of the following steps, where \( g \) represents gradient and \( m, v \) are 1st and 2nd order moment estimates (mean and variance).

**Usage**

```python
mx.symbol.adam_update(...)```
Arguments

weight NDArray-or-Symbol Weight
grad NDArray-or-Symbol Gradient
mean NDArray-or-Symbol Moving mean
var NDArray-or-Symbol Moving variance
lr float, required Learning rate
beta1 float, optional, default=0.899999976 The decay rate for the 1st moment estimates.
beta2 float, optional, default=0.999000013 The decay rate for the 2nd moment estimates.
epsilon float, optional, default=9.99999994e-09 A small constant for numerical stability.
wd float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
lazy.update boolean, optional, default=1 If true, lazy updates are applied if gradient’s stype is row_sparse and all of w, m and v have the same stype
name string, optional Name of the resulting symbol.

Details

.. math::

\begin{align*}
g_t &= \nabla J(W_{t-1}) \\
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
W_t &= W_{t-1} - \alpha \frac{m_t}{\sqrt{v_t} + \epsilon}
\end{align*}

It updates the weights using::

\begin{align*}
m &= \beta_1 m + (1 - \beta_1) g \n
v &= \beta_2 v + (1 - \beta_2) (g^2) w += - \text{learning_rate} \times \frac{m}{\sqrt{v} + \epsilon}
\end{align*}

However, if grad’s storage type is “row_sparse”, “lazy_update” is True and the storage type of weight is the same as those of m and v, only the row slices whose indices appear in grad.indices are updated (for w, m and v)::


Defined in src/operator/optimizer_op.cc:L679

Value

out The result mx.symbol
mx.symbol.add_n

add_n: Adds all input arguments element-wise.

Description

.. math:: \text{add\_n}(a_1, a_2, ..., a_n) = a_1 + a_2 + ... + a_n

Usage

mx.symbol.add_n(...)

Arguments

args NDArray-or-Symbol[] Positional input arguments
name string, optional Name of the resulting symbol.

Details

“add_n” is potentially more efficient than calling “add“ by ‘n‘ times.
The storage type of “add_n” output depends on storage types of inputs
- add_n(row_sparse, row_sparse, ..) = row_sparse
- add_n(default, csr, default) = default
- add_n(any input combinations longer than 4 (>4) with at least one default type) = default
- otherwise, “add_n“ falls all inputs back to default storage and generates default storage
Defined in src/operator/tensor/elemwise_sum.cc:L155

Value

out The result mx.symbol

mx.symbol.all_finite

all_finite: Check if all the float numbers in the array are finite (used for AMP)

Description

Defined in src/operator/contrib/all_finite.cc:L101

Usage

mx.symbol.all_finite(...)

Arguments

data NDArray Array
init.output boolean, optional, default=1 Initialize output to 1.
name string, optional Name of the resulting symbol.
**mx.symbol.amp_cast**

*amp_cast*: Cast function between low precision float/FP32 used by AMP.

**Description**

It casts only between low precision float/FP32 and does not do anything for other types.

**Usage**

`mx.symbol.amp_cast(...)`

**Arguments**

- **data**: NDArray-or-Symbol The input.
- **dtype**: 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', required Output data type.
- **name**: string, optional Name of the resulting symbol.

**Details**

Defined in src/operator/tensor/amp_cast.cc:L121

**Value**

out The result mx.symbol

---

mx.symbol.amp_multicast

*amp_multicast*: Cast function used by AMP, that casts its inputs to the common widest type.

**Description**

It casts only between low precision float/FP32 and does not do anything for other types.

**Usage**

`mx.symbol.amp_multicast(...)`
mx.symbol.arccos

Arguments

- **data**: NDArray-or-Symbol[\] Weights
- **num.outputs**: int, required Number of input/output pairs to be casted to the widest type.
- **cast.narrow**: boolean, optional, default=0 Whether to cast to the narrowest type
- **name**: string, optional Name of the resulting symbol.

Details

Defined in src/operator/tensor/amp_cast.cc:L165

Value

out The result mx.symbol

--

mx.symbol.arccos

arccos: Returns element-wise inverse cosine of the input array.

Description

The input should be in range ‘[-1, 1]’. The output is in the closed interval :math:`[0, \pi]`

Usage

mx.symbol.arccos(...)
**mx.symbol.arccosh**

**arccosh:** Returns the element-wise inverse hyperbolic cosine of the input array, computed element-wise.

**Description**

The storage type of “arccosh” output is always dense.

**Usage**

```python
mx.symbol.arccosh(...)
```

**Arguments**

- **data** (NDArray-or-Symbol): The input array.
- **name** (string, optional): Name of the resulting symbol.

**Details**

Defined in `src/operator/tensor/elemwise_unary_op_trig.cc:L535`

**Value**

- **out** (The result `mx.symbol`)

---

**mx.symbol.arcsin**

**arcsin:** Returns element-wise inverse sine of the input array.

**Description**

The input should be in the range ‘[-1, 1]’. The output is in the closed interval of \([-\pi/2, \pi/2]\).

**Usage**

```python
mx.symbol.arcsin(...)
```

**Arguments**

- **data** (NDArray-or-Symbol): The input array.
- **name** (string, optional): Name of the resulting symbol.
mx.symbol.arcsinh

Details

\[ \text{arcsin}([-1, -.707, 0, .707, 1]) = [-\pi/2, -\pi/4, 0, \pi/4, \pi/2] \]

The storage type of “arcsin” output depends upon the input storage type:

- arcsin(default) = default
- arcsin(row_sparse) = row_sparse
- arcsin(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L187

Value

out The result mx.symbol

mx.symbol.arcsinh

arcsinh: Returns the element-wise inverse hyperbolic sine of the input array, computed element-wise.

Description

The storage type of “arcsinh” output depends upon the input storage type:

Usage

mx.symbol.arcsinh(...)

Arguments

data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.

Details

- arcsinh(default) = default
- arcsinh(row_sparse) = row_sparse
- arcsinh(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L494

Value

out The result mx.symbol
mx.symbol.arctan

arctan: Returns element-wise inverse tangent of the input array.

Description

The output is in the closed interval $[-\pi/2, \pi/2]$.

Usage

mx.symbol.arctan(...)

Arguments

data NDArray-or-Symbol The input array.

name string, optional Name of the resulting symbol.

Details

.. math:: \text{arctan}([-1, 0, 1]) = [-\pi/4, 0, \pi/4]

The storage type of “arctan” output depends upon the input storage type:

- arctan(default) = default
- arctan(row_sparse) = row_sparse
- arctan(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L282

Value

out The result mx.symbol

mx.symbol.arctanh

arctanh: Returns the element-wise inverse hyperbolic tangent of the input array, \( \text{computed element-wise} \).

Description

The storage type of “arctanh” output depends upon the input storage type:

Usage

mx.symbol.arctanh(...)
mx.symbol.argmax

Details
- arctanh(default) = default - arctanh(row_sparse) = row_sparse - arctanh(csr) = csr
Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L579

Value
out The result mx.symbol
mx.symbol.argmax_channel

argmax_channel: Returns argmax indices of each channel from the input array.

Description

The result will be an NDArray of shape (num_channel,).

Usage

mx.symbol.argmax_channel(...)

Arguments

data NDArray-or-Symbol The input array
name string, optional Name of the resulting symbol.

Details

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples:

x = [[0., 1., 2.], [3., 4., 5.]]
argmax_channel(x) = [2., 2.]

Defined in src/operator/tensor/broadcast_reduce_op_index.cc:L97

Value

out The result mx.symbol

mx.symbol.argmin

argmin: Returns indices of the minimum values along an axis.

Description

In the case of multiple occurrences of minimum values, the indices corresponding to the first occurrence are returned.

Usage

mx.symbol.argmin(...)
**mx.symbol.argsort**

**Arguments**

- **data**: NDArray-or-Symbol
  - The input

- **axis**: int or None, optional, default=`None`
  - The axis along which to perform the reduction. Negative values means indexing from right to left. "Requires axis to be set as int, because global reduction is not supported yet."

- **keepdims**: boolean, optional, default=0
  - If this is set to ‘True’, the reduced axis is left in the result as dimension with size one.

- **name**: string, optional
  - Name of the resulting symbol.

**Details**

Examples::

    x = [[ 0., 1., 2.], [ 3., 4., 5.]]

    // argmin along axis 0
    argmin(x, axis=0) = [ 0., 0., 0.]

    // argmin along axis 1
    argmin(x, axis=1) = [ 0., 0.]

    // argmin along axis 1 keeping same dims as an input array
    argmin(x, axis=1, keepdims=True) = [[ 0.], [ 0.]]

Defined in `src/operator/tensor/broadcast_reduce_op_index.cc:L77`

**Value**

out The result
mx.symbol

```
mx.symbol.argsort
```

argsort: Returns the indices that would sort an input array along the given axis.

**Description**

This function performs sorting along the given axis and returns an array of indices having same shape as an input array that index data in sorted order.

**Usage**

mx.symbol.argsort(...)

**Arguments**

- **data**: NDArray-or-Symbol
  - The input array

- **axis**: int or None, optional, default=`-1`
  - Axis along which to sort the input tensor. If not given, the flattened array is used. Default is -1.

- **is.ascend**: boolean, optional, default=1
  - Whether to sort in ascending or descending order.
mx.symbol.BatchNorm

**Description**

Normalizes a data batch by mean and variance, and applies a scale “gamma” as well as offset “beta”.

**Usage**

```
mx.symbol.BatchNorm(...)  
```

**Arguments**

- `data` NDArray-or-Symbol Input data to batch normalization
- `gamma` NDArray-or-Symbol gamma array
- `beta` NDArray-or-Symbol beta array
- `moving.mean` NDArray-or-Symbol running mean of input
- `moving.var` NDArray-or-Symbol running variance of input
- `eps` double, optional, default=0.0010000000474974513 Epsilon to prevent div 0. Must be no less than CUDNN_BN_MIN_EPSILON defined in cudnn.h when using cudnn (usually 1e-5)
- `momentum` float, optional, default=0.899999976 Momentum for moving average
- `fix.gamma` boolean, optional, default=1 Fix gamma while training

**Details**

Examples::

```python
x = [[ 0.3, 0.2, 0.4], [ 0.1, 0.3, 0.2]]
// sort along axis -1 argsort(x) = [[ 1., 0., 2.], [ 0., 2., 1.]]
// sort along axis 0 argsort(x, axis=0) = [[ 1., 0., 1.], [ 0., 1., 0.]]
// flatten and then sort argsort(x, axis=None) = [ 3., 1., 5., 0., 4., 2.]  
```

Defined in src/operator/tensor/ordering_op.cc:L185

**Value**

```
out The result mx.symbol
```

```
mx.symbol.BatchNorm  BatchNorm:Batch normalization.  
```
**use.global.stats**
boolean, optional, default=0 Whether use global moving statistics instead of local batch-norm. This will force change batch-norm into a scale shift operator.

**output.mean.var**
boolean, optional, default=0 Output the mean and inverse std

**axis**
int, optional, default='1' Specify which shape axis the channel is specified

**cudnn.off**
boolean, optional, default=0 Do not select CUDNN operator, if available

**min.calib.range**
float or None, optional, default=None The minimum scalar value in the form of float32 obtained through calibration. If present, it will be used to by quantized batch norm op to calculate primitive scale. Note: this calib_range is to calib bn output.

**max.calib.range**
float or None, optional, default=None The maximum scalar value in the form of float32 obtained through calibration. If present, it will be used to by quantized batch norm op to calculate primitive scale. Note: this calib_range is to calib bn output.

**name**
string, optional Name of the resulting symbol.

**Details**

Assume the input has more than one dimension and we normalize along axis 1. We first compute the mean and variance along this axis:

\[
\text{data\_mean}[i] = \text{mean}(\text{data}[;i,:,:\ldots]) \quad \text{data\_var}[i] = \text{var}(\text{data}[;i,:,:\ldots])
\]

Then compute the normalized output, which has the same shape as input, as following:

\[
\text{out}[;i,:,:\ldots] = \frac{\text{data}[;i,:,:\ldots] - \text{data\_mean}[i]}{\sqrt{\text{data\_var}[i]} + \epsilon} \frac{\text{gamma}[i]}{\text{data\_var}[i]} + \text{beta}[i]
\]

Both \(*\text{mean}\)* and \(*\text{var}\)* returns a scalar by treating the input as a vector.

Assume the input has size \(*k*\) on axis 1, then both “gamma” and “beta” have shape \(*(k,)*\). If “output_mean_var” is set to be true, then outputs both “data_mean” and the inverse of “data_var”, which are needed for the backward pass. Note that gradient of these two outputs are blocked.

Besides the inputs and the outputs, this operator accepts two auxiliary states, “moving_mean” and “moving_var”, which are \(*k*\)-length vectors. They are global statistics for the whole dataset, which are updated by:

\[
\text{moving\_mean} = \text{moving\_mean} \times \text{momentum} + \text{data\_mean} \times (1 - \text{momentum}) \\
\text{moving\_var} = \text{moving\_var} \times \text{momentum} + \text{data\_var} \times (1 - \text{momentum})
\]

If “use_global_stats” is set to be true, then “moving_mean” and “moving_var” are used instead of “data_mean” and “data_var” to compute the output. It is often used during inference.

The parameter “axis” specifies which axis of the input shape denotes the ‘channel’ (separately normalized groups). The default is 1. Specifying -1 sets the channel axis to be the last item in the input shape.

Both “gamma” and “beta” are learnable parameters. But if “fix_gamma” is true, then set “gamma” to 1 and its gradient to 0.
mx.symbol.BatchNorm_v1

.. Note:: When “fix_gamma” is set to True, no sparse support is provided. If “fix_gamma is” set to False, the sparse tensors will fallback.

Defined in src/operator/nn/batch_norm.cc:L545

Value

out The result mx.symbol

mx.symbol.BatchNorm_v1

BatchNorm_v1:Batch normalization.

Description

This operator is DEPRECATED. Perform BatchNorm on the input.

Usage

mx.symbol.BatchNorm_v1(...) 

Arguments

data NDArray-or-Symbol Input data to batch normalization

gamma NDArray-or-Symbol gamma array

beta NDArray-or-Symbol beta array

eps float, optional, default=0.00100000005 Epsilon to prevent div 0

momentum float, optional, default=0.899999976 Momentum for moving average

fix.gamma boolean, optional, default=1 Fix gamma while training

use.global.stats boolean, optional, default=0 Whether use global moving statistics instead of local batch-norm. This will force change batch-norm into a scale shift operator.

output.mean.var boolean, optional, default=0 Output All,normal mean and var

name string, optional Name of the resulting symbol.

Details

Normalizes a data batch by mean and variance, and applies a scale “gamma“ as well as offset “beta“.
Assume the input has more than one dimension and we normalize along axis 1. We first compute the mean and variance along this axis:

.. math::

data\_mean[i] = mean(data[:,i,:,:,:]) \ data\_var[i] = var(data[:,i,:,:,:])

Then compute the normalized output, which has the same shape as input, as following:
out[:,i,:,...] = \frac{\text{data[:,i,:,...]} - \text{data_mean}[i]\sqrt{\text{data_var}[i]} + \epsilon * \text{gamma}[i] + \text{beta}[i]}{299}

Both *mean* and *var* returns a scalar by treating the input as a vector. Assume the input has size *k* on axis 1, then both “gamma” and “beta” have shape *(k,)*. If “output_mean_var” is set to be true, then outputs both “data_mean” and “data_var” as well, which are needed for the backward pass.

Besides the inputs and the outputs, this operator accepts two auxiliary states, “moving_mean” and “moving_var”, which are *k*-length vectors. They are global statistics for the whole dataset, which are updated by:

\[
\text{moving_mean} = \text{moving_mean} \times \text{momentum} + \text{data_mean} \times (1 - \text{momentum}) \\
\text{moving_var} = \text{moving_var} \times \text{momentum} + \text{data_var} \times (1 - \text{momentum})
\]

If “use_global_stats” is set to be true, then “moving_mean” and “moving_var” are used instead of “data_mean” and “data_var” to compute the output. It is often used during inference. Both “gamma” and “beta” are learnable parameters. But if “fix_gamma” is true, then set “gamma” to 1 and its gradient to 0.

There’s no sparse support for this operator, and it will exhibit problematic behavior if used with sparse tensors.

Defined in src/operator/batch_norm_v1.cc:L95

### Value

out The result mx.symbol

### Description

“batch_dot” is used to compute dot product of “x” and “y” when “x” and “y” are data in batch, namely N-D (N >= 3) arrays in shape of ‘(B0, ..., B_i, :, :)’.

### Usage

mx.symbol.batch_dot(...)
Details

For example, given “x” with shape ‘(B_0, ..., B_i, N, M)’ and “y” with shape ‘(B_0, ..., B_i, M, K)’, the result array will have shape ‘(B_0, ..., B_i, N, K)’, which is computed by:

\[ \text{batch\_dot}(x, y)[b_0, ..., b_i, :, :] = \text{dot}(x[b_0, ..., b_i, :, :], y[b_0, ..., b_i, :, :]) \]

Defined in src/operator/tensor/dot.cc:L127

Value

out The result mx.symbol

mx.symbol.batch_take

batch_take: Takes elements from a data batch.

Description

.. note:: ‘batch_take’ is deprecated. Use ‘pick’ instead.

Usage

mx.symbol.batch_take(...)

Arguments

a NDArray-or-Symbol The input array
indices NDArray-or-Symbol The index array
name string, optional Name of the resulting symbol.

Details

Given an input array of shape “(d0, d1)“ and indices of shape “(i0,)”, the result will be an output array of shape “(i0,)“ with:

\[ \text{output}[i] = \text{input}[i, \text{indices}[i]] \]

Examples:

\[ x = [[ 1., 2.], [ 3., 4.], [ 5., 6.]] \]

// takes elements with specified indices
\[ \text{batch\_take}(x, [0,1,0]) = [ 1. 4. 5.] \]

Defined in src/operator/tensor/indexing_op.cc:L841

Value

out The result mx.symbol
mx.symbol.BilinearSampler

_BilinearSampler: Applies bilinear sampling to input feature map._

**Description**

Bilinear Sampling is the key of [NIPS2015] "Spatial Transformer Networks". The usage of the operator is very similar to remap function in OpenCV, except that the operator has the backward pass.

**Usage**

```
mx.symbol.BilinearSampler(...)
```

**Arguments**

- **data** NDArray-or-Symbol Input data to the BilinearsamplerOp.
- **grid** NDArray-or-Symbol Input grid to the BilinearsamplerOp.grid has two channels:
  - `x_src`, `y_src`
- **cudnn.off** boolean or None, optional, default=None whether to turn cudnn off
- **name** string, optional Name of the resulting symbol.

**Details**

Given `:math:`\text{"data"}` and `:math:`\text{"grid"}`, then the output is computed by

\[
x_{\text{src}} = \text{grid[batch, 0, y_{\text{dst}}, x_{\text{dst}}]}
\quad y_{\text{src}} = \text{grid[batch, 1, y_{\text{dst}}, x_{\text{dst}}]}
\quad \text{output[batch, channel, y_{\text{dst}}, x_{\text{dst}}]} = G(\text{data[batch, channel, y_{\text{src}}, x_{\text{src}}])}
\]

`:math:`\text{"x_{\text{dst}}"}, :math:`\text{"y_{\text{dst}}"}` enumerate all spatial locations in `:math:`\text{"output"}`, and `:math:`\text{"G()"}` denotes the bilinear interpolation kernel. The out-boundary points will be padded with zeros. The shape of the output will be `(data.shape[0], data.shape[1], grid.shape[2], grid.shape[3])`.

The operator assumes that `:math:`\text{"data"}` has `\text{"NCHW"}` layout and `:math:`\text{"grid"}` has been normalized to [-1, 1].

BilinearSampler often cooperates with GridGenerator which generates sampling grids for BilinearSampler. GridGenerator supports two kinds of transformation: “affine” and “warp”. If users want to design a CustomOp to manipulate `:math:`\text{"grid"}`, please firstly refer to the code of GridGenerator.

**Example 1:**
```python
## Zoom out data two times
data = array([[[1, 4, 3, 6], [1, 8, 8, 9], [0, 4, 1, 5], [1, 0, 1, 3]]])

affine_matrix = array([[2, 0, 0], [0, 2, 0]])
affine_matrix = reshape(affine_matrix, shape=(1, 6))
grid = GridGenerator(data=affine_matrix, transform_type='affine', target_shape=(4, 4))
out = BilinearSampler(data, grid)
out [[[ 0, 0, 0, 0], [ 0, 3.5, 6.5, 0], [ 0, 1.25, 2.5, 0], [ 0, 0, 0, 0]]]
```
Example 2::

```python
## shift data horizontally by -1 pixel

data = array((([[1, 4, 3, 6], [1, 8, 9, 7], [0, 4, 1, 5], [1, 0, 1, 3]]))

warp_matrix = array((([[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]], [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]))

grid = GridGenerator(data=warp_matrix, transform_type='warp')
out = BilinearSampler(data, grid)

out = array(([[4, 3, 6, 0], [8, 8, 9, 0], [4, 1, 5, 0], [0, 1, 3, 0]])
```

Defined in src/operator/bilinear_sampler.cc:L256

Value

out The result mx.symbol

---

mx.symbol.BlockGrad  

**BlockGrad:** Stops gradient computation.

Description

Stops the accumulated gradient of the inputs from flowing through this operator in the backward direction. In other words, this operator prevents the contribution of its inputs to be taken into account for computing gradients.

Usage

mx.symbol.BlockGrad(…)

Arguments

data  NDArray-or-Symbol The input array.

name  string, optional Name of the resulting symbol.

Details

Example::

```python
v1 = [1, 2] v2 = [0, 1] a = Variable('a') b = Variable('b') b_stop_grad = stop_gradient(3 * b)
loss = MakeLoss(b_stop_grad + a)
executor = loss.simple_bind(ctx=cpu(), a=(1,2), b=(1,2))
executor.forward(is_train=True, a=v1, b=v2)
executor.backward()
executor.grad_arrays [ 0. 0. ] [ 1. 1. ]
```

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L325

Value

out The result mx.symbol
mx.symbol.broadcast_add

*broadcast_add*: Returns element-wise sum of the input arrays with broadcasting.

**Description**

'broadcast_plus' is an alias to the function 'broadcast_add'.

**Usage**

mx.symbol.broadcast_add(...)

**Arguments**

- **lhs**: NDArray-or-Symbol, First input to the function
- **rhs**: NDArray-or-Symbol, Second input to the function
- **name**: string, optional, Name of the resulting symbol.

**Details**

Example::

x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_add(x, y) = [[ 1., 1., 1.], [ 2., 2., 2.]]
broadcast_plus(x, y) = [[ 1., 1., 1.], [ 2., 2., 2.]]

Supported sparse operations:

broadcast_add(csr, dense(1D)) = dense broadcast_add(dense(1D), csr) = dense

Defined in src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L58

**Value**

out The result mx.symbol
mx.symbol.broadcast_axes

broadcast_axes: Broadcasts the input array over particular axes.

Description

Broadcasting is allowed on axes with size 1, such as from '(2,1,3,1)' to '(2,8,3,9)'. Elements will be duplicated on the broadcasted axes.

Usage

mx.symbol.broadcast_axes(...)  

Arguments

data NDArray-or-Symbol The input  
axis Shape(tuple), optional, default=[] The axes to perform the broadcasting.
size Shape(tuple), optional, default=[] Target sizes of the broadcasting axes.
name string, optional Name of the resulting symbol.

Details

'broadcast_axes' is an alias to the function 'broadcast_axis'.

Example:  
// given x of shape (1,2,1) x = [[[ 1.], [ 2.]]]
// broadcast x on axis 2 broadcast_axis(x, axis=2, size=3) = [[[ 1.], [ 1.], [ 2., 2., 2.]]]  
// broadcast with a different shape  
// broadcast x on axes 0 and 2 broadcast_axis(x, axis=(0,2), size=(2,3)) = [[[ 1.], [ 1.], [ 2., 2., 2.]], [[ 1., 1., 1.], [ 2., 2., 2.]]]

Defined in src/operator/tensor/broadcast_reduce_op_value.cc:L93

Value

out The result mx.symbol
mx.symbol.broadcast_axis

broadcast_axis: Broadcasts the input array over particular axes.

Description

Broadcasting is allowed on axes with size 1, such as from ‘(2,1,3,1)’ to ‘(2,8,3,9)’. Elements will be duplicated on the broadcasted axes.

Usage

mx.symbol.broadcast_axis(...)

Arguments

data NDArray-or-Symbol The input
axis Shape(tuple), optional, default=[] The axes to perform the broadcasting.
size Shape(tuple), optional, default=[] Target sizes of the broadcasting axes.
name string, optional Name of the resulting symbol.

Details

‘broadcast_axes’ is an alias to the function ‘broadcast_axis’.

Example::

// given x of shape (1,2,1) x = [[[ 1.], [ 2.]]]

// broadcast x on axis 2 broadcast_axis(x, axis=2, size=3) = [[[ 1., 1., 1.], [ 2., 2., 2.]]] // broadcast x on on axes 0 and 2 broadcast_axis(x, axis=(0,2), size=(2,3)) = [[[ 1., 1., 1.], [ 2., 2., 2.]], [[ 1., 1., 1.], [ 2., 2., 2.]]]

Defined in src/operator/tensor/broadcast_reduce_op_value.cc:L93

Value

out The result mx.symbol
mx.symbol.broadcast_div

broadcast_div: Returns element-wise division of the input arrays with broadcasting.

Description

Example::

Usage

mx.symbol.broadcast_div(...)

Arguments

lhs NDArray-or-Symbol First input to the function
rhs NDArray-or-Symbol Second input to the function
name string, optional Name of the resulting symbol.

Details

\[
\begin{align*}
x &= \begin{bmatrix} 6., 6., 6. \\ 6., 6., 6. \end{bmatrix} \\
y &= \begin{bmatrix} 2. \\ 3. \end{bmatrix} \\
\text{broadcast_div}(x, y) &= \begin{bmatrix} 3., 3., 3. \\ 2., 2., 2. \end{bmatrix}
\end{align*}
\]

Supported sparse operations:

broadcast_div(csr, dense(1D)) = csr

Defined in src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L187

Value

out The result mx.symbol

mx.symbol.broadcast_equal

broadcast_equal: Returns the result of element-wise **equal to** (==) comparison operation with broadcasting.

Description

Example::

Usage

mx.symbol.broadcast_equal(...)
**Arguments**

- `lhs` : NDArray-or-Symbol  
  First input to the function
- `rhs` : NDArray-or-Symbol  
  Second input to the function
- `name` : string  
  optional Name of the resulting symbol.

**Details**

\[
\begin{align*}
\mathbf{x} &= \begin{bmatrix} 1., 1., 1. \end{bmatrix}, \begin{bmatrix} 1., 1., 1. \end{bmatrix} \\
\mathbf{y} &= \begin{bmatrix} 0. \end{bmatrix}, \begin{bmatrix} 1. \end{bmatrix} \\
broadcast_{\text{equal}}(\mathbf{x}, \mathbf{y}) &= \begin{bmatrix} 0., 0., 0. \end{bmatrix}, \begin{bmatrix} 1., 1., 1. \end{bmatrix}
\end{align*}
\]

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L46`

**Value**

- `out` : The result mx.symbol

---

**Description**

Example::

**Usage**

```python
mx.symbol.broadcast_greater(...)```

**Arguments**

- `lhs` : NDArray-or-Symbol  
  First input to the function
- `rhs` : NDArray-or-Symbol  
  Second input to the function
- `name` : string  
  optional Name of the resulting symbol.

**Details**

\[
\begin{align*}
\mathbf{x} &= \begin{bmatrix} 1., 1., 1. \end{bmatrix}, \begin{bmatrix} 1., 1., 1. \end{bmatrix} \\
\mathbf{y} &= \begin{bmatrix} 0. \end{bmatrix}, \begin{bmatrix} 1. \end{bmatrix} \\
broadcast_{\text{greater}}(\mathbf{x}, \mathbf{y}) &= \begin{bmatrix} 1., 1., 1. \end{bmatrix}, \begin{bmatrix} 0., 0., 0. \end{bmatrix}
\end{align*}
\]

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L82`

**Value**

- `out` : The result mx.symbol
mx.symbol.broadcast_greater_equal

```
broadcast_greater_equal: Returns the result of element-wise **greater
than or equal to** (>=) comparison operation with broadcasting.
```

**Description**

Example::

**Usage**

```
mx.symbol.broadcast_greater_equal(...)
```

**Arguments**

- `lhs` (NDArray-or-Symbol): First input to the function.
- `rhs` (NDArray-or-Symbol): Second input to the function.
- `name` (string, optional): Name of the resulting symbol.

**Details**

```
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_greater_equal(x, y) = [[ 1., 1., 1.], [ 1., 1., 1.]]
```

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L100`

**Value**

```
out The result mx.symbol
```

mx.symbol.broadcast_hypot

```
broadcast_hypot: Returns the hypotenuse of a right angled triangle,
given its "legs" with broadcasting.
```

**Description**

It is equivalent to doing :math:`sqrt(x_1^2 + x_2^2)`.

**Usage**

```
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_hypot(x, y) = [[ 1., 1., 1.], [ 1., 1., 1.]]
```

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L100`
**mx.symbol.broadcast_lesser**

Returns the result of element-wise **lesser than** (`<`) comparison operation with broadcasting.

**Arguments**

- **lhs**: NDArray-or-Symbol First input to the function
- **rhs**: NDArray-or-Symbol Second input to the function
- **name**: string, optional Name of the resulting symbol.

**Details**

Example:

```python
def broadcast_hypot(x, y):
    return x < y
```

```python
x = [[ 3., 3., 3.],
     [ 4., 4.]]
y = [[ 0.],
     [ 1.]]
broadcast_hypot(x, y) = [[ 0., 0., 0.],
                         [ 0., 0., 0.]]
```

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L118`

**Value**

- **out**: The result mx.symbol

---

**mx.symbol.broadcast_lesser**

*broadcast_lesser:* Returns the result of element-wise **lesser than** (`<`) comparison operation with broadcasting.

**Description**

Example:

**Usage**

```python
mx.symbol.broadcast_lesser(...)```

**Arguments**

- **lhs**: NDArray-or-Symbol First input to the function
- **rhs**: NDArray-or-Symbol Second input to the function
- **name**: string, optional Name of the resulting symbol.

**Details**

```python
x = [[ 1., 1., 1.],
     [ 1., 1., 1.]]
y = [[ 0.],
     [ 1.]]
broadcast_lesser(x, y) = [[ 0., 0., 0.],
                         [ 0., 0., 0.]]
```

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L118`
mx.symbol.broadcast_lesser_equal

broadcast_lesser_equal: Returns the result of element-wise "less than or equal to" (<=) comparison operation with broadcasting.

Value

out The result mx.symbol

mx.symbol.broadcast_lesser_equal

Description

Example:

Usage

mx.symbol.broadcast_lesser_equal(…)

Arguments

lhs
NDArray-or-Symbol First input to the function
rhs
NDArray-or-Symbol Second input to the function
name
string, optional Name of the resulting symbol.

Details

x = [[[1., 1., 1.], [1., 1., 1.]]
y = [[[0.], [1.]]
broadcast_lesser_equal(x, y) = [[[0., 0., 0.], [1., 1., 1.]]

Defined in src/operator/tensor/elementwise_binary_broadcast_op_logic.cc:L136

Value

out The result mx.symbol
**mx.symbol.broadcast_like**

*broadcast_like:* Broadcasts lhs to have the same shape as rhs.

**Description**

Broadcasting is a mechanism that allows NDArrays to perform arithmetic operations with arrays of different shapes efficiently without creating multiple copies of arrays. Also see, ‘Broadcasting <https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>‘ for more explanation.

**Usage**

```python
mx.symbol.broadcast_like(...)```

**Arguments**

- `lhs` : NDArray-or-Symbol First input.
- `rhs` : NDArray-or-Symbol Second input.
- `lhs.axes` : Shape or None, optional, default=None Axes to perform broadcast on in the first input array
- `rhs.axes` : Shape or None, optional, default=None Axes to copy from the second input array
- `name` : string, optional Name of the resulting symbol.

**Details**

Broadcasting is allowed on axes with size 1, such as from `(2,1,3,1)` to `(2,8,3,9)`. Elements will be duplicated on the broadcasted axes.

For example:

```
broadcast_like([[1,2,3]], [[5,6,7],[7,8,9]]) = [[ 1., 2., 3.], [ 1., 2., 3.]]
broadcast_like([9], [1,2,3,4,5], lhs_axes=(0,), rhs_axes=(-1,)) = [9,9,9,9,9]
```

Defined in src/operator/tensor/broadcast_reduce_op_value.cc:L171

**Value**

```
out The result mx.symbol
```
mx.symbol.broadcast_logical_and

broadcast_logical_and: Returns the result of element-wise **logical and** with broadcasting.

Description

Example::

Usage

mx.symbol.broadcast_logical_and(...)

Arguments

lhs NDArray-or-Symbol First input to the function
rhs NDArray-or-Symbol Second input to the function
name string, optional Name of the resulting symbol.

Details

\[
\begin{align*}
x &= \begin{bmatrix} 1., 1., 1. \\ 1., 1., 1. \end{bmatrix} \\
y &= \begin{bmatrix} 0. \\ 1. \end{bmatrix} \\
broadcast\_logical\_and(x, y) &= \begin{bmatrix} 0., 0., 0. \\ 1., 1., 1. \end{bmatrix}
\end{align*}
\]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L154

Value

out The result mx.symbol

mx.symbol.broadcast_logical_or

broadcast_logical_or: Returns the result of element-wise **logical or** with broadcasting.

Description

Example::

Usage

mx.symbol.broadcast_logical_or(...)
**Arguments**

- **lhs**: NDArray-or-Symbol
  First input to the function

- **rhs**: NDArray-or-Symbol
  Second input to the function

- **name**: string, optional
  Name of the resulting symbol.

**Details**

\[
x = \begin{bmatrix} 1., 1., 0. \\ 1., 1., 0. \end{bmatrix}
\]
\[
y = \begin{bmatrix} 1. \\ 0. \end{bmatrix}
\]

\[
broadcast\_logical\_or(x, y) = \begin{bmatrix} 1., 1., 1. \\ 1., 1., 0. \end{bmatrix}
\]

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L172`

**Value**

- **out**: The result mx.symbol

---

**Description**

**Example**:::

**Usage**

`mx.symbol.broadcast_logical_xor(...)`

**Arguments**

- **lhs**: NDArray-or-Symbol
  First input to the function

- **rhs**: NDArray-or-Symbol
  Second input to the function

- **name**: string, optional
  Name of the resulting symbol.

**Details**

\[
x = \begin{bmatrix} 1., 1., 0. \\ 1., 1., 0. \end{bmatrix}
\]
\[
y = \begin{bmatrix} 1. \\ 1. \end{bmatrix}
\]

\[
broadcast\_logical\_xor(x, y) = \begin{bmatrix} 0., 0., 1. \\ 1., 1., 0. \end{bmatrix}
\]

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L190`

**Value**

- **out**: The result mx.symbol
mx.symbol.broadcast_maximum

**broadcast_maximum:** Returns element-wise maximum of the input arrays with broadcasting.

**Description**
This function compares two input arrays and returns a new array having the element-wise maxima.

**Usage**
```python
mx.symbol.broadcast_maximum(...)```

**Arguments**
- `lhs`: NDArray-or-Symbol First input to the function
- `rhs`: NDArray-or-Symbol Second input to the function
- `name`: string, optional Name of the resulting symbol.

**Details**
Example:
```python
x = [[[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[[ 0.], [ 1.]]
broadcast_maximum(x, y) = [[[ 1., 1., 1.], [ 1., 1., 1.]]
```

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_extended.cc:L81`

**Value**
- `out`: The result `mx.symbol`

mx.symbol.broadcast_minimum

**broadcast_minimum:** Returns element-wise minimum of the input arrays with broadcasting.

**Description**
This function compares two input arrays and returns a new array having the element-wise minima.

**Usage**
```python
mx.symbol.broadcast_minimum(...)```
mx.symbol.broadcast_minus

Arguments

- **lhs**: NDArray-or-Symbol, First input to the function.
- **rhs**: NDArray-or-Symbol, Second input to the function.
- **name**: string, optional, Name of the resulting symbol.

Details

Example::

```python
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_maximum(x, y) = [[ 0., 0., 0.], [ 1., 1., 1.]]
```

Defined in `src/operator/tensor/elemwise_binary_broadcast_op_extended.cc:L117`

Value

- **out**: The result mx.symbol

mx.symbol.broadcast_minus

`broadcast_minus`: Returns element-wise difference of the input arrays with broadcasting.

Description

‘broadcast_minus’ is an alias to the function ‘broadcast_sub’.

Usage

```python
mx.symbol.broadcast_minus(...)```

Arguments

- **lhs**: NDArray-or-Symbol, First input to the function.
- **rhs**: NDArray-or-Symbol, Second input to the function.
- **name**: string, optional, Name of the resulting symbol.

Details

Example::

```python
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_sub(x, y) = [[ 1., 1., 1.], [ 0., 0., 0.]]
broadcast_minus(x, y) = [[ 1., 1., 1.], [ 0., 0., 0.]]
```
Supported sparse operations:

\[
\text{broadcast\_sub/minus}(\text{csr, dense}(1D)) = \text{dense broadcast\_sub/minus(dense}(1D), \text{csr}) = \text{dense}
\]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L106

**Value**

\[
\text{out} \text{ The result mx.symbol}
\]

**mx.symbol.broadcast_mod**

\[
\text{broadcast\_mod: Returns element-wise modulo of the input arrays with broadcasting.}
\]

**Description**

Example:

**Usage**

\[
\text{mx.symbol.broadcast\_mod(...)}
\]

**Arguments**

- `lhs` NDArray-or-Symbol First input to the function
- `rhs` NDArray-or-Symbol Second input to the function
- `name` string, optional Name of the resulting symbol.

**Details**

\[
\begin{align*}
x &= \begin{bmatrix} 8., 8., 8. \\ 8., 8., 8. \end{bmatrix} \\
y &= \begin{bmatrix} 2. \\ 3. \end{bmatrix} \\
broadcast\_mod(x, y) &= \begin{bmatrix} 0., 0. \\ 2., 2. \end{bmatrix}
\end{align*}
\]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L222

**Value**

\[
\text{out} \text{ The result mx.symbol}
\]
**mx.symbol.broadcast_mul**

*broadcast_mul:* Returns element-wise product of the input arrays with broadcasting.

---

**Description**

Example::

**Usage**

```
mx.symbol.broadcast_mul(...)
```

**Arguments**

- `lhs` NDArray-or-Symbol First input to the function
- `rhs` NDArray-or-Symbol Second input to the function
- `name` string, optional Name of the resulting symbol.

**Details**

\[
x = \begin{bmatrix} 1., 1., 1. \\ 1., 1., 1. \end{bmatrix}
\]

\[
y = \begin{bmatrix} 0. \\ 1. \end{bmatrix}
\]

\[
broadcast_mul(x, y) = \begin{bmatrix} 0., 0., 0. \\ 1., 1., 1. \end{bmatrix}
\]

Supported sparse operations:

\[
broadcast_mul(csr, dense(1D)) = csr
\]

Defined in `src/operator/tensor elemwise_binary_broadcast_op_basic.cc:L146`

**Value**

```
out The result mx.symbol
```

---

**mx.symbol.broadcast_not_equal**

*broadcast_not_equal:* Returns the result of element-wise **not equal** to***(!=)** comparison operation with broadcasting.

---

**Description**

Example::

**Usage**

```
mx.symbol.broadcast_not_equal(...)
```
mx.symbol.broadcast_plus

Arguments

lhs  NDArray-or-Symbol First input to the function
rhs  NDArray-or-Symbol Second input to the function
name  string, optional Name of the resulting symbol.

Details

\[
x = \begin{bmatrix} 1., 1., 1. \\ 1., 1., 1. \end{bmatrix}
\]
\[
y = \begin{bmatrix} 0. \\ 1. \end{bmatrix}
\]
\[
broadcast\_not\_equal(x, y) = \begin{bmatrix} 1., 1., 1. \\ 0., 0., 0. \end{bmatrix}
\]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_logic.cc:L64

Value

out The result mx.symbol

mx.symbol.broadcast_plus

broadcast_plus: Returns element-wise sum of the input arrays with broadcasting.

Description

‘broadcast_plus’ is an alias to the function ‘broadcast_add’.

Usage

mx.symbol.broadcast_plus(…)

Arguments

lhs  NDArray-or-Symbol First input to the function
rhs  NDArray-or-Symbol Second input to the function
name  string, optional Name of the resulting symbol.

Details

Example::

\[
x = \begin{bmatrix} 1., 1., 1. \\ 1., 1., 1. \end{bmatrix}
\]
\[
y = \begin{bmatrix} 0. \\ 1. \end{bmatrix}
\]
\[
broadcast\_add(x, y) = \begin{bmatrix} 1., 1., 1. \\ 2., 2., 2. \end{bmatrix}
\]
\[
broadcast\_plus(x, y) = \begin{bmatrix} 1., 1., 1. \\ 2., 2., 2. \end{bmatrix}
\]

Supported sparse operations:

\[
broadcast\_add(csr, dense(1D)) = dense broadcast\_add(dense(1D), csr) = dense
\]

Defined in src/operator/tensor/elemwise_binary_broadcast_op_basic.cc:L58
Value

out The result mx.symbol

---

mx.symbol.broadcast_power

broadcast_power: Returns result of first array elements raised to powers from second array, element-wise with broadcasting.

---

Description

Example:

Usage

mx.symbol.broadcast_power(...)

Arguments

lhs NDArray-or-Symbol First input to the function
rhs NDArray-or-Symbol Second input to the function
name string, optional Name of the resulting symbol.

Details

x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_power(x, y) = [[ 2., 2., 2.], [ 4., 4., 4.]]
Defined in src/operator/tensor/elemwise_binary_broadcast_op_extended.cc:L45

Value

out The result mx.symbol
mx.symbol.broadcast_sub

\[ \text{broadcast_sub}: \text{Returns element-wise difference of the input arrays with broadcasting.} \]

---

**Description**

'broadcast_minus' is an alias to the function 'broadcast_sub'.

**Usage**

```python
mx.symbol.broadcast_sub(...)```

**Arguments**

- `lhs`: NDArray-or-Symbol First input to the function
- `rhs`: NDArray-or-Symbol Second input to the function
- `name`: string, optional Name of the resulting symbol.

**Details**

Example:

\[
x = [[ 1., 1., 1.], [ 1., 1., 1.]]
y = [[ 0.], [ 1.]]
broadcast_sub(x, y) = [[ 1., 1., 1.], [ 0., 0., 0.]]
broadcast_minus(x, y) = [[ 1., 1., 1.], [ 0., 0., 0.]]
\]

Supported sparse operations:

\[
broadcast_sub/minus(csr, dense(1D)) = dense\ \text{broadcast_sub/minus(dense(1D), csr)} = dense
\]

Defined in `src/operator/tensor/elewise_binary_broadcast_op_basic.cc:L106`

**Value**

- `out` The result mx.symbol
mx.symbol.broadcast_to

*broadcast_to*: Broadcasts the input array to a new shape.

**Description**

Broadcasting is a mechanism that allows NDArrays to perform arithmetic operations with arrays of different shapes efficiently without creating multiple copies of arrays. Also see, ‘Broadcasting &lt;https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html&gt;‘ for more explanation.

**Usage**

mx.symbol.broadcast_to(...)

**Arguments**

- **data**: NDArray-or-Symbol The input
- **shape**: Shape(tuple), optional, default=[] The shape of the desired array. We can set the dim to zero if it’s same as the original. E.g ‘A = broadcast_to(B, shape=(10, 0, 0))’ has the same meaning as ‘A = broadcast_axis(B, axis=0, size=10)’.
- **name**: string, optional Name of the resulting symbol.

**Details**

Broadcasting is allowed on axes with size 1, such as from ‘(2,1,3,1)’ to ‘(2,8,3,9)’. Elements will be duplicated on the broadcasted axes.

For example::

broadcast_to([[1,2,3]], shape=(2,3)) = [[ 1., 2., 3.], [ 1., 2., 3.]]

The dimension which you do not want to change can also be kept as ‘0’ which means copy the original value. So with ‘shape=(2,0)’, we will obtain the same result as in the above example.

Defined in src/operator/tensor/broadcast_reduce_op_value.cc:L117

**Value**

- **out**: The result mx.symbol
mx.symbol.cast

Cast: Casts all elements of the input to a new type.

Description

.. note:: “Cast” is deprecated. Use “cast” instead.

Usage

mx.symbol.cast(...)

Arguments

data NDArray-or-Symbol The input.
dtype 'bfloat16', 'bool', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', required Output data type.
name string, optional Name of the resulting symbol.

details

Example:

cast([0.9, 1.3], dtype='int32') = [0, 1]
cast([1e20, 11.1], dtype='float16') = [inf, 11.09375]
cast([300, 11.1, 10.9, -1, -3], dtype='uint8') = [44, 11, 10, 255, 253]

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L664

Value

out The result mx.symbol

mx.symbol.cast

cast: Casts all elements of the input to a new type.

Description

.. note:: “Cast” is deprecated. Use “cast” instead.

Usage

mx.symbol.cast(...)

Arguments

data NDArray-or-Symbol The input.
dtype 'bfloat16', 'bool', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', required Output data type.
name string, optional Name of the resulting symbol.
mx.symbol.cast_storage

Details

Example::

    cast([0.9, 1.3], dtype='int32') = [0, 1]
cast([1e20, 11.1], dtype='float16') = [inf, 11.09375]
cast([300, 11.1, 10.9, -1, -3], dtype='uint8') = [44, 11, 10, 255, 253]

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L664

Value

out The result mx.symbol

mx.symbol.cast_storage

cast_storage:Casts tensor storage type to the new type.

Description

When an NDArray with default storage type is cast to csr or row_sparse storage, the result is compact, which means:

Usage

mx.symbol.cast_storage(...)

Arguments

data NDArray-or-Symbol The input.
stype 'csr', 'default', 'row_sparse', required Output storage type.
name string, optional Name of the resulting symbol.

Details

- for csr, zero values will not be retained - for row_sparse, row slices of all zeros will not be retained

The storage type of “cast_storage” output depends on stype parameter:

- cast_storage(csr, 'default') = default - cast_storage(row_sparse, 'default') = default - cast_storage(default, 'csr') = csr - cast_storage(default, 'row_sparse') = row_sparse - cast_storage(csr, 'csr') = csr - cast_storage(row_sparse, 'row_sparse') = row_sparse

Example::

dense = [[ 0., 1., 0.], [ 2., 0., 3.], [ 0., 0., 0.], [ 0., 0., 0.]]
# cast to row_sparse storage type
rsp = cast_storage(dense, 'row_sparse')
rsp.indices = [0, 1]
rsp.values = [[ 0., 1., 0.], [ 2., 0., 3.]]
# cast to csr storage type
csr = cast_storage(dense, 'csr')
csr.indices = [1, 0, 2]
csr.values = [ 1., 2., 3.]
csr.indptr = [0, 1, 3, 3, 3]

Defined in src/operator/tensor/cast_storage.cc:L71
Value

out The result mx.symbol

mx.symbol.ceil

ceil: Returns element-wise ceiling of the input.

Description

The ceil of the scalar x is the smallest integer i, such that i >= x.

Usage

mx.symbol.ceil(...)

Arguments

data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.

Details

Example::
ceil([1, 8, -125]) = [1, 2, -5]
The storage type of “ceil” output depends upon the input storage type:
- ceil(default) = default - ceil(row_sparse) = row_sparse - ceil(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_pow.cc:L270

Value

out The result mx.symbol

mx.symbol.cbrt
cbrt: Returns element-wise cube-root value of the input.

Description

.. math:: cbrt(x) = \sqrt[3]{x}

Usage

mx.symbol.cbrt(...)
mx.symbol.choose_element_0index

Arguments

data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.

Details

Example::

  ceil([-2.1, -1.9, 1.5, 1.9, 2.1]) = [-2., -1., 2., 2., 3.]

The storage type of “ceil” output depends upon the input storage type:
- ceil(default) = default - ceil(row_sparse) = row_sparse - ceil(csr) = csr

Defined in src/operator/tensor/elemwise UnaryOpBasic.cc:L817

Value

out The result mx.symbol

mx.symbol.choose_element_0index

choose_element_0index: Picks elements from an input array according to the input indices along the given axis.

Description

Given an input array of shape “(d0, d1)” and indices of shape “(i0,)”, the result will be an output array of shape “(i0,)” with:

Usage

mx.symbol.choose_element_0index(...)  

Arguments

data NDArray-or-Symbol The input array
index NDArray-or-Symbol The index array
axis int or None, optional, default='-1' int or None. The axis to picking the elements. Negative values means indexing from right to left. If is 'None', the elements in the index w.r.t the flattened input will be picked.
keepdims boolean, optional, default=0 If true, the axis where we pick the elements is left in the result as dimension with size one.
mode 'clip', 'wrap',optional, default='clip' Specify how out-of-bound indices behave. Default is "clip". "clip" means clip to the range. So, if all indices mentioned are too large, they are replaced by the index that addresses the last element along an axis. "wrap" means to wrap around.
name string, optional Name of the resulting symbol.
### Details

`output[i] = input[i, indices[i]]`

By default, if any index mentioned is too large, it is replaced by the index that addresses the last element along an axis (the ‘clip’ mode).

This function supports n-dimensional input and (n-1)-dimensional indices arrays.

**Examples:**

```python
x = [[ 1., 2.], [ 3., 4.], [ 5., 6.]]
// picks elements with specified indices along axis 0
pick(x, y=[0,1], 0) = [[ 1., 4.]]
// picks elements with specified indices along axis 1
pick(x, y=[0,1,0], 1) = [[ 1., 4., 5.]]
// picks elements with specified indices along axis 1 using ‘wrap’ mode // to place indices that
would normally be out of bounds
pick(x, y=[2,-1,-2], 1, mode='wrap') = [[ 1., 4., 5.]]
y = [[ 1.], [ 0.], [ 2.]]
// picks elements with specified indices along axis 1 and dims are maintained
pick(x, y, 1, keepdims=True) = [[[ 2.], [ 3.], [ 6.]]]
```

Defined in `src/operator/tensor/broadcast_reduce_op_index.cc:L151`

---

### Value

**out** The result `mx.symbol`

---

**mx.symbol.clip**

`clip`: Clips (limits) the values in an array. Given an interval, values outside the interval are clipped to the interval edges. Clipping “x” between ‘a_min’ and ‘a_max’ would be:

\[
\text{clip}(x, a_{\text{min}}, a_{\text{max}}) = \max(\min(x, a_{\text{max}}), a_{\text{min}})
\]

**Examples:**

```python
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
clip(x,1,8) = [ 1., 1., 2., 3., 4., 5., 6., 7., 8., 8.]
```

The storage type of “clip” output depends on storage types of inputs and the `a_min`, `a_max` parameter values:

- clip(default) = default - clip(row_sparse, a_min <= 0, a_max >= 0) = row_sparse - clip(csr, a_min <= 0, a_max >= 0) = csr - clip(row_sparse, a_min < 0, a_max < 0) = default - clip(row_sparse, a_min > 0, a_max > 0) = default - clip(csr, a_min < 0, a_max < 0) = csr - clip(csr, a_min > 0, a_max > 0) = csr

---

### Description

Defined in `src/operator/tensor/matrix_op.cc:L677`

### Usage

```python
mx.symbol.clip(...)```
mx.symbol.col2im

Arguments

- **data**: NDArray-or-Symbol Input array.
- **a.min**: float, required Minimum value
- **a.max**: float, required Maximum value
- **name**: string, optional Name of the resulting symbol.

Value

- **out**: The result mx.symbol

---

**mx.symbol.col2im**

col2im: Combining the output column matrix of im2col back to image array.

---

**Description**

Like :class:`~mxnet.ndarray.im2col`, this operator is also used in the vanilla convolution implementation. Despite the name, col2im is not the reverse operation of im2col. Since there may be overlaps between neighbouring sliding blocks, the column elements cannot be directly put back into image. Instead, they are accumulated (i.e., summed) in the input image just like the gradient computation, so col2im is the gradient of im2col and vice versa.

**Usage**

mx.symbol.col2im(...)

Arguments

- **data**: NDArray-or-Symbol Input array to combine sliding blocks.
- **output.size**: Shape(tuple), required The spatial dimension of image array: (w,), (h, w) or (d, h, w).
- **kernel**: Shape(tuple), required Sliding kernel size: (w,), (h, w) or (d, h, w).
- **stride**: Shape(tuple), optional, default=[] The stride between adjacent sliding blocks in spatial dimension: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.
- **dilate**: Shape(tuple), optional, default=[] The spacing between adjacent kernel points: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.
- **pad**: Shape(tuple), optional, default=[] The zero-value padding size on both sides of spatial dimension: (w,), (h, w) or (d, h, w). Defaults to no padding.
- **name**: string, optional Name of the resulting symbol.
Details

Using the notation in im2col, given an input column array of shape \( (N, C \times \prod(\text{kernel}), W) \); this operator accumulates the column elements into output array of shape \( (N, C, \text{output_size}[0], \text{output_size}[1], \ldots) \). Only 1-D, 2-D and 3-D of spatial dimension is supported in this operator.

Defined in src/operator/nn/im2col.cc:L182

Value

out The result mx.symbol

mx.symbol.Concat

Perform an feature concat on channel dim (dim 1) over all the inputs.

Description

Perform an feature concat on channel dim (dim 1) over all the inputs.

Usage

mx.symbol.Concat(data, num.args, dim = NULL, name = NULL)

Arguments

data list, required List of tensors to concatenate
num.args int, required Number of inputs to be concatenated.
dim int, optional, default=’1’ the dimension to be concatenated.
name string, optional Name of the resulting symbol.

Value

out The result mx.symbol
mx.symbol.concat  Perform an feature concat on channel dim (dim 1) over all the inputs.

Description
Perform an feature concat on channel dim (dim 1) over all the inputs.

Usage
mx.symbol.concat(data, num.args, dim = NULL, name = NULL)

Arguments
- data: list, required List of tensors to concatenate
- num.args: int, required Number of inputs to be concated.
- dim: int, optional, default=’1’ the dimension to be concated.
- name: string, optional Name of the resulting symbol.

Value
out The result mx.symbol

mx.symbol.Convolution  Convolution:Compute *N*-D convolution on *(N+2)*-D input.

Description
In the 2-D convolution, given input data with shape *(batch_size, channel, height, width)*, the output is computed by

Usage
mx.symbol.Convolution(...)

Arguments
- data: NDArray-or-Symbol Input data to the ConvolutionOp.
- weight: NDArray-or-Symbol Weight matrix.
- bias: NDArray-or-Symbol Bias parameter.
- kernel: Shape(tuple), required Convolution kernel size: (w,), (h, w) or (d, h, w)
- stride: Shape(tuple), optional, default=[] Convolution stride: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.
- dilate: Shape(tuple), optional, default=[] Convolution dilate: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.
pad Shape(tuple), optional, default=[] Zero pad for convolution: (w,), (h, w) or (d, h, w). Defaults to no padding.

num.filter int (non-negative), required Convolution filter(channel) number

num.group int (non-negative), optional, default=1 Number of group partitions.

workspace long (non-negative), optional, default=1024 Maximum temporary workspace allowed (MB) in convolution. This parameter has two usages. When CUDNN is not used, it determines the effective batch size of the convolution kernel. When CUDNN is used, it controls the maximum temporary storage used for tuning the best CUDNN kernel when 'limited_workspace' strategy is used.

no.bias boolean, optional, default=0 Whether to disable bias parameter.

cudnn.tune None, 'fastest', 'limited_workspace', 'off', optional, default='None' Whether to pick convolution algo by running performance test.

cudnn.off boolean, optional, default=0 Turn off cudnn for this layer.

layout None, 'NCDHW', 'NCHW', 'NCW', 'NDHWC', 'NHWC', optional, default='None' Set layout for input, output and weight. Empty for default layout: NCW for 1d, NCHW for 2d and NCDHW for 3d. NHWC and NDHWC are only supported on GPU.

name string, optional Name of the resulting symbol.

Details

.. math::

   \text{out}[n,i,:,:] = \text{bias}[i] + \sum_{j=0}^{\text{channel}} \text{data}[n,j,:,:] \star \text{weight}[i,j,:,:]

where :math:`\star` is the 2-D cross-correlation operator.

For general 2-D convolution, the shapes are

- **data**: *(batch_size, channel, height, width)* - **weight**: *(num_filter, channel, kernel[0], kernel[1])* - **bias**: *(num_filter,)* - **out**: *(batch_size, num_filter, out_height, out_width)*.

Define::

   f(x,k,p,s,d) = floor((x+2*p-d*(k-1)-1)/s)+1

then we have::

   out_height=f(height, kernel[0], pad[0], stride[0], dilate[0]) out_width=f(width, kernel[1], pad[1], stride[1], dilate[1])

If "no_bias" is set to be true, then the "bias" term is ignored.

The default data "layout" is *NCHW*, namely *(batch_size, channel, height, width)*. We can choose other layouts such as *NWC*.

If "num_group" is larger than 1, denoted by *g*, then split the input "data" evenly into *g* parts along the channel axis, and also evenly split "weight" along the first dimension. Next compute the convolution on the *i*-th part of the data with the *i*-th weight part. The output is obtained by concatenating all the *g* results.

1-D convolution does not have *height* dimension but only *width* in space.

- **data**: *(batch_size, channel, width)* - **weight**: *(num_filter, channel, kernel[0])* - **bias**: *(num_filter,)* - **out**: *(batch_size, num_filter, out_width)*.
3-D convolution adds an additional *depth* dimension besides *height* and *width*. The shapes are:
- **data**: *(batch_size, channel, depth, height, width)*
- **weight**: *(num_filter, channel, kernel[0], kernel[1], kernel[2])*  
- **bias**: *(num_filter,)*
- **out**: *(batch_size, num_filter, out_depth, out_height, out_width)*.

Both "weight" and "bias" are learnable parameters.

There are other options to tune the performance.
- **cudnn_tune**: enable this option leads to higher startup time but may give faster speed. Options are:
  - **off**: no tuning
  - **limited_workspace**: run test and pick the fastest algorithm that doesn’t exceed workspace limit.
  - **fastest**: pick the fastest algorithm and ignore workspace limit.
  - **None** (default): the behavior is determined by environment variable "MXNET_CUDNN_AUTOTUNE_DEFAULT": 0 for off, 1 for limited workspace (default), 2 for fastest.
- **workspace**: A large number leads to more (GPU) memory usage but may improve the performance.

Defined in src/operator/nn/convolution.cc:L469

Value

out The result mx.symbol

mx.symbol.Convolution_v1

Convolution_v1:This operator is DEPRECATED. Apply convolution to input then add a bias.

Description

Convolution_v1:This operator is DEPRECATED. Apply convolution to input then add a bias.

Usage

mx.symbol.Convolution_v1(…)

Arguments

data          NDArray-or-Symbol Input data to the ConvolutionV1Op.
weight        NDArray-or-Symbol Weight matrix.
bias          NDArray-or-Symbol Bias parameter.
kernel        Shape(tuple), required convolution kernel size: (h, w) or (d, h, w)
stride        Shape(tuple), optional, default=[] convolution stride: (h, w) or (d, h, w)
dilate        Shape(tuple), optional, default=[] convolution dilate: (h, w) or (d, h, w)
pad           Shape(tuple), optional, default=[] pad for convolution: (h, w) or (d, h, w)
mx.symbol.Correlation

Correlation: Applies correlation to inputs.

**Value**

out The result mx.symbol

**Description**

The correlation layer performs multiplicative patch comparisons between two feature maps.

**Usage**

mx.symbol.Correlation(

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data1</td>
<td>NDArray-or-Symbol Input data1 to the correlation.</td>
</tr>
<tr>
<td>data2</td>
<td>NDArray-or-Symbol Input data2 to the correlation.</td>
</tr>
<tr>
<td>kernel.size</td>
<td>int (non-negative), optional, default=1 kernel size for Correlation must be an odd number</td>
</tr>
</tbody>
</table>
mx.symbol.Correlation

max.displacement  int (non-negative), optional, default=1 Max displacement of Correlation
stride1  int (non-negative), optional, default=1 stride1 quantize data1 globally
stride2  int (non-negative), optional, default=1 stride2 quantize data2 within the neighborhood centered around data1
pad.size  int (non-negative), optional, default=0 pad for Correlation
is.multiply  boolean, optional, default=1 operation type is either multiplication or subduction
name  string, optional Name of the resulting symbol.

Details

Given two multi-channel feature maps :math:`f_1, f_2`, with :math:`w`, :math:`h`, and :math:`c` being their width, height, and number of channels, the correlation layer lets the network compare each patch from :math:`f_1` with each patch from :math:`f_2`.

For now we consider only a single comparison of two patches. The 'correlation' of two patches centered at :math:`x_1` in the first map and :math:`x_2` in the second map is then defined as:

.. math::
   \mathbf{c}(x_1, x_2) = \sum_{o \in [-k,k] \times [-k,k]} \langle f_1(x_1 + o), f_2(x_2 + o) \rangle

for a square patch of size :math:`K:=2k+1`.

Note that the equation above is identical to one step of a convolution in neural networks, but instead of convolving data with a filter, it convolves data with other data. For this reason, it has no training weights.

Computing :math:`\mathbf{c}(x_1, x_2)` involves :math:`c \times K^2` multiplications. Comparing all patch combinations involves :math:`w^2 \times h^2` such computations.

Given a maximum displacement :math:`d`, for each location :math:`x_1` it computes correlations :math:`\mathbf{c}(x_1, x_2)` only in a neighborhood of size :math:`D:=2d+1`, by limiting the range of :math:`x_2`. We use strides :math:`s_1, s_2`, to quantize :math:`x_1` globally and to quantize :math:`x_2` within the neighborhood centered around :math:`x_1`.

The final output is defined by the following expression:

.. math::
   \mathbf{out}[n, q, i, j] = \mathbf{c}(x_i, j, x_q)


Defined in src/operator/correlation.cc:L198

Value

out The result mx.symbol
**mx.symbol.cos**

*Function: Computes the element-wise cosine of the input array.*

**Description**

The input should be in radians (:math:`2\pi` rad equals 360 degrees).

**Usage**

```python
mx.symbol.cos(...)
```

**Arguments**

- `data` NDArray-or-Symbol The input array.
- `name` string, optional Name of the resulting symbol.

**Details**

```
.. math:: \cos([0, \pi/4, \pi/2]) = [1, 0.707, 0]
```

The storage type of “cos” output is always dense

```
Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L90
```

**Value**

- `out` The result mx.symbol

---

**mx.symbol.cosh**

*Function: Returns the hyperbolic cosine of the input array, computed element-wise.*

**Description**

```
.. math:: \cosh(x) = 0.5 \times (\exp(x) + \exp(-x))
```

**Usage**

```python
mx.symbol.cosh(...)
```

**Arguments**

- `data` NDArray-or-Symbol The input array.
- `name` string, optional Name of the resulting symbol.
mx.symbol.Crop

Details

The storage type of “cosh” output is always dense
Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L409

Value

out The result mx.symbol

mx.symbol.Crop      Crop:

Description

.. note:: ‘Crop’ is deprecated. Use ‘slice’ instead.

Usage

mx.symbol.Crop(...)  

Arguments

data Symbol or Symbol[] Tensor or List of Tensors, the second input will be used as
    crop_like shape reference
num.args int, required Number of inputs for crop, if equals one, then we will use the
    h_w for crop height and width, else if equals two, then we will use the height and
    width of the second input symbol, we name crop_like here
offset Shape(tuple), optional, default=[0,0] crop offset coordinate: (y, x)
h.w Shape(tuple), optional, default=[0,0] crop height and width: (h, w)
center.crop boolean, optional, default=0 If set to true, then it will use be the center_crop, or
    it will crop using the shape of crop_like
name string, optional Name of the resulting symbol.

Details

Crop the 2nd and 3rd dim of input data, with the corresponding size of h_w or with width and height
of the second input symbol, i.e., with one input, we need h_w to specify the crop height and width,
otherwise the second input symbol’s size will be used
Defined in src/operator/crop.cc:L50

Value

out The result mx.symbol
mx.symbol.crop  

.crop: Slices a region of the array. .. note:: “crop” is deprecated. Use “slice” instead. This function returns a sliced array between the indices given by ‘begin’ and ‘end’ with the corresponding ‘step’. For an input array of “shape=(d_0, d_1, ..., d_n-1)”, slice operation with “begin=(b_0, b_1, ..., b_m-1)”, “end=(e_0, e_1, ..., e_m-1)”, and “step=(s_0, s_1, ..., s_m-1)”, where m <= n, results in an array with the shape “((e_0-b_0)/s_0, ..., (e_m-1-b_m-1)/s_m-1, d_m, ..., d_n-1)”. The resulting array’s *k*-th dimension contains elements from the *k*-th dimension of the input array starting from index “b_k” (inclusive) with step “s_k” until reaching “e_k” (exclusive). If the *k*-th elements are ‘None’ in the sequence of ‘begin’, ‘end’, and ‘step’, the following rule will be used to set default values. If ‘s_k’ is ‘None’, set ‘s_k=1’. If ‘s_k > 0’, set ‘b_k=0’, ‘e_k=d_k’; else, set ‘b_k=d_k-1’, ‘e_k=-1’. The storage type of “slice” output depends on storage types of inputs - slice(csr) = csr - otherwise, “slice” generates output with default storage .. note:: When input data storage type is csr, it only supports step=(0), or step=(None,), or step=(1,) to generate a csr output. For other step parameter values, it falls back to slicing a dense tensor. Example::

x = [[ 1., 2., 3., 4.], [ 5., 6., 7., 8.], [ 9., 10., 11., 12.]]

slice(x, begin=(0,1), end=(2,4)) = [[ 2., 3., 4.], [ 6., 7., 8.]]
slice(x, begin=(None, 0), end=(None, 3), step=(-1, 2)) = [[9., 11.], [5., 7.], [1., 3.]]

Description

Defined in src/operator/tensor/matrix_op.cc:L482

Usage

mx.symbol.crop(...)

Arguments

data NDArray-or-Symbol Source input

begin Shape(tuple), required starting indices for the slice operation, supports negative indices.

end Shape(tuple), required ending indices for the slice operation, supports negative indices.

step Shape(tuple), optional, default=[] step for the slice operation, supports negative values.

name string, optional Name of the resulting symbol.

Value

out The result mx.symbol
Description

.. note:: The existing alias "contrib_CTCLoss" is deprecated.

Usage

mx.symbol.CTCLoss(...)

Arguments

data NDArray-or-Symbol Input ndarray

label NDArray-or-Symbol Ground-truth labels for the loss.

data.lengths NDArray-or-Symbol Lengths of data for each of the samples. Only required when use_data_lengths is true.

label.lengths NDArray-or-Symbol Lengths of labels for each of the samples. Only required when use_label_lengths is true.

use.data.lengths boolean, optional, default=0 Whether the data lenghts are decided by 'data_lengths'. If false, the lengths are equal to the max sequence length.

use.label.lengths boolean, optional, default=0 Whether the label lenghts are decided by 'label_lengths', or derived from 'padding_mask'. If false, the lengths are derived from the first occurrence of the value of 'padding_mask'. The value of 'padding_mask' is "0" when first CTC label is reserved for blank, and "-1" when last label is reserved for blank. See 'blank_label'.

blank.label 'first', 'last', optional, default='first' Set the label that is reserved for blank label. If "first", 0-th label is reserved, and label values for tokens in the vocabulary are between "1" and "alphabet_size-1", and the padding mask is "-1". If "last", last label value "alphabet_size-1" is reserved for blank label instead, and label values for tokens in the vocabulary are between "0" and "alphabet_size-2", and the padding mask is "0".

name string, optional Name of the resulting symbol.

Details

The shapes of the inputs and outputs:

- **data**:: ‘(sequence_length, batch_size, alphabet_size)’ - **label**:: ‘(batch_size, label_sequence_length)’ - **out**:: ‘(batch_size)’

The 'data' tensor consists of sequences of activation vectors (without applying softmax), with i-th channel in the last dimension corresponding to i-th label for i between 0 and alphabet_size-1 (i.e always 0-indexed). Alphabet size should include one additional value reserved for blank label.
When 'blank_label' is ""first"", the “0”-th channel is be reserved for activation of blank label, or otherwise if it is "last", “(alphabet_size-1)”-th channel should be reserved for blank label. “label” is an index matrix of integers. When ‘blank_label’ is ""first"", the value 0 is then reserved for blank label, and should not be passed in this matrix. Otherwise, when ‘blank_label’ is ""last"", the value ‘(alphabet_size-1)’ is reserved for blank label.

If a sequence of labels is shorter than *label_sequence_length*, use the special padding value at the end of the sequence to conform it to the correct length. The padding value is ‘0’ when ‘blank_label’ is ""first"", and ‘-1’ otherwise.

For example, suppose the vocabulary is ‘[a, b, c]’, and in one batch we have three sequences ‘ba’, ’cbb’, and ’abac’. When ‘blank_label’ is ""first"", we can index the labels as ‘’a’: 1, ’b’: 2, ’c’: 3’, and we reserve the 0-th channel for blank label in data tensor. The resulting ‘label’ tensor should be padded to be::

[[2, 1, 0, 0], [3, 2, 2, 0], [1, 2, 1, 3]]

When ‘blank_label’ is ""last"", we can index the labels as ‘’a’: 0, ’b’: 1, ’c’: 2’, and we reserve the channel index 3 for blank label in data tensor. The resulting ‘label’ tensor should be padded to be::

[[1, 0, -1, -1], [2, 1, 1, -1], [0, 1, 0, 2]]

“out” is a list of CTC loss values, one per example in the batch.

See *Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks*, A. Graves *et al*. for more information on the definition and the algorithm.

Defined in src/operator/nn/ctc_loss.cc:L100

Value

out The result mx.symbol

mx.symbol.ctc_loss  

ctc_loss:Connectionist Temporal Classification Loss.

Description

.. note:: The existing alias “contrib_CTCLoss” is deprecated.

Usage

mx.symbol.ctc_loss(...)  

Arguments

- **data**: NDArray-or-Symbol Input ndarray
- **label**: NDArray-or-Symbol Ground-truth labels for the loss.
- **data.lengths**: NDArray-or-Symbol Lengths of data for each of the samples. Only required when use_data_lengths is true.
- **label.lengths**: NDArray-or-Symbol Lengths of labels for each of the samples. Only required when use_label_lengths is true.
use.data.lengths

boolean, optional, default=0 Whether the data lengths are decided by ‘data_lengths’. If false, the lengths are equal to the max sequence length.

use.label.lengths

boolean, optional, default=0 Whether the label lengths are decided by ‘label_lengths’, or derived from ‘padding_mask’. If false, the lengths are derived from the first occurrence of the value of ‘padding_mask’. The value of ‘padding_mask’ is “0” when first CTC label is reserved for blank, and “-1” when last label is reserved for blank. See ‘blank_label’.

blank.label

‘first’, ‘last’, optional, default=’first’ Set the label that is reserved for blank label. If ”first”, 0-th label is reserved, and label values for tokens in the vocabulary are between “1” and “alphabet_size-1”, and the padding mask is “-1”. If ”last”, last label value “alphabet_size-1” is reserved for blank label instead, and label values for tokens in the vocabulary are between “0” and “alphabet_size-2”, and the padding mask is “0”.

name

string, optional Name of the resulting symbol.

Details

The shapes of the inputs and outputs:

- **data**: ‘(sequence_length, batch_size, alphabet_size)’ - **label**: ‘(batch_size, label_sequence_length)’
- **out**: ‘(batch_size)’

The ‘data’ tensor consists of sequences of activation vectors (without applying softmax), with i-th channel in the last dimension corresponding to i-th label for i between 0 and alphabet_size-1 (i.e always 0-indexed). Alphabet size should include one additional value reserved for blank label. When ‘blank_label’ is “‘first’”, the “0”-th channel is be reserved for activation of blank label, or otherwise if it is ”last”, “(alphabet_size-1)”-th channel should be reserved for blank label.

“label” is an index matrix of integers. When ‘blank_label’ is “‘first’”, the value 0 is then reserved for blank label, and should not be passed in this matrix. Otherwise, when ‘blank_label’ is “‘last’”, the value “(alphabet_size-1)” is reserved for blank label.

If a sequence of labels is shorter than *label_sequence_length*, use the special padding value at the end of the sequence to conform it to the correct length. The padding value is ‘0’ when ‘blank_label’ is “‘first’”, and ‘-1’ otherwise.

For example, suppose the vocabulary is ‘[a, b, c]’, and in one batch we have three sequences ‘ba’, ‘cbb’, and ‘abac’. When ‘blank_label’ is “‘first’”, we can index the labels as ‘‘a’: 1, ‘b’: 2, ’c’: 3”, and we reserve the 0-th channel for blank label in data tensor. The resulting ‘label’ tensor should be padded to be::

[[2, 1, 0, 0], [3, 2, 2, 0], [1, 2, 1, 3]]

When ‘blank_label’ is “‘last’”, we can index the labels as ‘‘a’: 0, ‘b’: 1, ’c’: 2”, and we reserve the channel index 3 for blank label in data tensor. The resulting ‘label’ tensor should be padded to be::

[[1, 0, -1, -1], [2, 1, 1, -1], [0, 1, 0, 2]]

“out” is a list of CTC loss values, one per example in the batch.

See *Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks*, A. Graves *et al*., for more information on the definition and the algorithm.

Defined in src/operator/nn/ctc_loss.cc:L100
Value

out The result mx.symbol

---

mx.symbol.cumsum  

`cumsum:` Return the cumulative sum of the elements along a given axis.

Description

Defined in `src/operator/numpy/np_cumsum.cc:L70`

Usage

```
mx.symbol.cumsum(...)  
```

Arguments

- `a` NDArray-or-Symbol Input ndarray
- `axis` int or None, optional, default='None' Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.
- `dtype` None, 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', optional, default='None' Type of the returned array and of the accumulator in which the elements are summed. If dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.
- `name` string, optional Name of the resulting symbol.

Value

out The result mx.symbol

---

mx.symbol.Custom  

`Custom:` Apply a custom operator implemented in a frontend language (like Python).

Description

Custom operators should override required methods like ‘forward’ and ‘backward’. The custom operator must be registered before it can be used. Please check the tutorial here: https://mxnet.incubator.apache.org/api/faq/new_op

Usage

```
mx.symbol.Custom(...)  
```
mx.symbol.Deconvolution

Arguments

- **data**: NDArray-or-Symbol[] Input data for the custom operator.
- **op.type**: string Name of the custom operator. This is the name that is passed to `mx.operator.register` to register the operator.
- **name**: string, optional Name of the resulting symbol.

Details

Defined in src/operator/custom/custom.cc:L547

Value

- **out**: The result mx.symbol

---

Deconvolution: Computes 1D or 2D transposed convolution (aka fractionally strided convolution) of the input tensor. This operation can be seen as the gradient of Convolution operation with respect to its input. Convolution usually reduces the size of the input. Transposed convolution works the other way, going from a smaller input to a larger output while preserving the connectivity pattern.

Description

Deconvolution: Computes 1D or 2D transposed convolution (aka fractionally strided convolution) of the input tensor. This operation can be seen as the gradient of Convolution operation with respect to its input. Convolution usually reduces the size of the input. Transposed convolution works the other way, going from a smaller input to a larger output while preserving the connectivity pattern.

Usage

```python
mx.symbol.Deconvolution(...)```

Arguments

- **data**: NDArray-or-Symbol Input tensor to the deconvolution operation.
- **weight**: NDArray-or-Symbol Weights representing the kernel.
- **bias**: NDArray-or-Symbol Bias added to the result after the deconvolution operation.
- **kernel**: Shape(tuple), required Deconvolution kernel size: (w,), (h, w) or (d, h, w). This is same as the kernel size used for the corresponding convolution.
- **stride**: Shape(tuple), optional, default=[] The stride used for the corresponding convolution: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.
- **dilate**: Shape(tuple), optional, default=[] Dilation factor for each dimension of the input: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.
mx.symbol.degrees

degrees: Converts each element of the input array from radians to degrees.

.. math:: degrees([0, \pi/2, \pi, 3\pi/2, 2\pi]) = [0, 90, 180, 270, 360]

Usage

mx.symbol.degrees(...)
mx.symbol.depth_to_space

Arguments

data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.

Details

The storage type of "degrees" output depends upon the input storage type:
- degrees(default) = default - degrees(row_sparse) = row_sparse - degrees(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L332

Value

out The result mx.symbol

mx.symbol.depth_to_space:Rearranges(permutes) data from depth into blocks of spatial data. Similar to ONNX DepthToSpace operator:
The output is a new tensor where the values from depth dimension are moved in spatial blocks to height and width dimension. The reverse of this operation is "space_to_depth". .. math::
\begin{align*}
x' &= \text{reshape}(x, [N, \text{block}_\text{size}, \text{block}_\text{size}, C / (\text{block}_\text{size}^2), H * \text{block}_\text{size}, W * \text{block}_\text{size}]) \\
x'' &= \text{transpose}(x', [0, 3, 4, 1, 5, 2]) \\
y &= \text{reshape}(x'', [N, C / (\text{block}_\text{size}^2), H * \text{block}_\text{size}, W * \text{block}_\text{size}])
\end{align*}

where :math:`'x'` is an input tensor with default layout as :math:`'[N, C, H, W]'`: [batch, channels, height, width] and :math:`'y'` is the output tensor of layout :math:`'[N, C / (\text{block}_\text{size}^2), H * \text{block}_\text{size}, W * \text{block}_\text{size}]'`

Example::

x = [[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]], [[12, 13, 14], [15, 16, 17]], [[18, 19, 20], [21, 22, 23]]] depth_to_space(x, 2) = [[[0, 6, 1, 7, 2, 8], [12, 18, 13, 19, 14, 20], [3, 9, 4, 10, 5, 11], [15, 21, 16, 22, 17, 23]]]

Description

Defined in src/operator/tensor/matrix_op.cc:L972

Usage

mx.symbol.depth_to_space(…)

Arguments

data NDArray-or-Symbol Input ndarray
block.size int, required Blocks of [block_size, block_size] are moved
name string, optional Name of the resulting symbol.
Value

out The result mx.symbol

mx.symbol.diag

diag: Extracts a diagonal or constructs a diagonal array.

Description

"diag"'s behavior depends on the input array dimensions:

Usage

mx.symbol.diag(...)

Arguments

data NDArray-or-Symbol Input ndarray

k int, optional, default='0' Diagonal in question. The default is 0. Use k>0 for diagonals above the main diagonal, and k<0 for diagonals below the main diagonal. If input has shape (S0 S1) k must be between -S0 and S1

axis1 int, optional, default='0' The first axis of the sub-arrays of interest. Ignored when the input is a 1-D array.

axis2 int, optional, default='1' The second axis of the sub-arrays of interest. Ignored when the input is a 1-D array.

name string, optional Name of the resulting symbol.

Details

- 1-D arrays: constructs a 2-D array with the input as its diagonal, all other elements are zero.
- N-D arrays: extracts the diagonals of the sub-arrays with axes specified by "axis1" and "axis2". The output shape would be decided by removing the axes numbered "axis1" and "axis2" from the input shape and appending to the result a new axis with the size of the diagonals in question.

For example, when the input shape is '(2, 3, 4, 5)', "axis1" and "axis2" are 0 and 2 respectively and "k" is 0, the resulting shape would be '(3, 5, 2)'.

Examples:

x = [[1, 2, 3], [4, 5, 6]]
diag(x) = [1, 5]
diag(x, k=1) = [2, 6]
diag(x, k=-1) = [4]
x = [1, 2, 3]
diag(x) = [[1, 0, 0], [0, 2, 0], [0, 0, 3]]
diag(x, k=1) = [[0, 1, 0], [0, 0, 2], [0, 0, 0]]
mx.symbol.dot

\[
\text{diag}(x, k=-1) = [[0, 0, 0], [1, 0, 0], [0, 2, 0]]
\]
\[
x = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
\]
\[
\text{diag}(x) = [[1, 7], [2, 8]]
\]
\[
\text{diag}(x, k=1) = [[3], [4]]
\]
\[
\text{diag}(x, \text{axis1}=-2, \text{axis2}=-1) = [[1, 4], [5, 8]]
\]

Defined in src/operator/tensor/diag_op.cc:L87

Value

out The result mx.symbol

mx.symbol.dot  dot:Dot product of two arrays.

Description

“dot”’s behavior depends on the input array dimensions:

Usage

mx.symbol.dot(...)

Arguments

lhs  NDArray-or-Symbol The first input
rhs  NDArray-or-Symbol The second input
transpose.a  boolean, optional, default=0 If true then transpose the first input before dot.
transpose.b  boolean, optional, default=0 If true then transpose the second input before dot.
forward.stype  None, 'csr', 'default', 'row_sparse',optional, default='None' The desired storage type of the forward output given by user, if the combination of input storage types and this hint does not match any implemented ones, the dot operator will perform fallback operation and still produce an output of the desired storage type.
name  string, optional Name of the resulting symbol.

Details

- 1-D arrays: inner product of vectors - 2-D arrays: matrix multiplication - N-D arrays: a sum product over the last axis of the first input and the first axis of the second input

For example, given 3-D “x” with shape ‘(n,m,k)’ and “y” with shape ‘(k,r,s)’, the result array will have shape ‘(n,m,r,s)’. It is computed by:
\[
dot(x,y)[i,j,a,b] = \text{sum}(x[i,j,:]*y[:,a,b])
\]

Example::
x = reshape([0,1,2,3,4,5,6,7], shape=(2,2,2)) y = reshape([7,6,5,4,3,2,1,0], shape=(2,2,2)) dot(x,y)[0,0,1,1] = 0 sum(x[0,0,:]*y[:,1,1]) = 0

The storage type of “dot” output depends on storage types of inputs, transpose option and forward_stype option for output storage type. Implemented sparse operations include:

- dot(default, default, transpose_a=True/False, transpose_b=True/False) = default
- dot(csr, default, transpose_a=True) = default
- dot(csr, default, transpose_a=True) = row_sparse
- dot(csr, default) = default
- dot(default, csr) = csr (CPU only)
- dot(default, csr, forward_stype='default') = default

If the combination of input storage types and forward_stype does not match any of the above patterns, “dot” will fallback and generate output with default storage.

.. Note::

If the storage type of the lhs is "csr", the storage type of gradient w.r.t rhs will be "row_sparse". Only a subset of optimizers support sparse gradients, including SGD, AdaGrad and Adam. Note that by default lazy updates is turned on, which may perform differently from standard updates. For more details, please check the Optimization API at: https://mxnet.incubator.apache.org/api/python/optimization/optimization.html

Defined in src/operator/tensor/dot.cc:L77

Value

out The result mx.symbol

mx.symbol.Dropout

Dropout:Applies dropout operation to input array.

Description

- During training, each element of the input is set to zero with probability p. The whole array is rescaled by :math:`1/(1-p)` to keep the expected sum of the input unchanged.

Usage

mx.symbol.Dropout(...)

Arguments

data

NDArray-or-Symbol Input array to which dropout will be applied.
p

float, optional, default=0.5 Fraction of the input that gets dropped out during training time.
mode

'always', 'training',optional, default='training' Whether to only turn on dropout during training or to also turn on for inference.
axes

Shape(tuple), optional, default=[] Axes for variational dropout kernel.
cudnn_off

boolean or None, optional, default=0 Whether to turn off cudnn in dropout operator. This option is ignored if axes is specified.
nname

string, optional Name of the resulting symbol.
mx.symbol.ElementWiseSum

Details

- During testing, this operator does not change the input if mode is 'training'. If mode is 'always', the same computation as during training will be applied.

Example:

```python
random.seed(998) input_array = array([[3., 0.5, -0.5, 2., 7.], [2., -0.4, 7., 3., 0.2]]) a = symbol.Variable('a') dropout = symbol.Dropout(a, p = 0.2) executor = dropout.simple_bind(a = input_array.shape)
```

```python
## If training executor.forward(is_train = True, a = input_array) executor.outputs [[ 3.75 0.625 -0.2.5 8.75 ] [ 2.5 -0.5 8.75 3.75 0. ]]
## If testing executor.forward(is_train = False, a = input_array) executor.outputs [[ 3. 0.5 -0.5 2. 7. ] [ 2. -0.4 7. 3. 0.2 ]] 
```

Defined in src/operator/nn/dropout.cc:L96

Value

out The result mx.symbol

mx.symbol.ElementWiseSum

.. math:: \text{add\_n}(a_1, a_2, ..., a_n) = a_1 + a_2 + ... + a_n

Description

.. math:: \text{add\_n}(a_1, a_2, ..., a_n) = a_1 + a_2 + ... + a_n

Usage

mx.symbol.ElementWiseSum(...)

Arguments

- **args**: NDArray-or-Symbol[] Positional input arguments
- **name**: string, optional Name of the resulting symbol.

Details

“add\_n” is potentially more efficient than calling “add” by ‘n’ times.
The storage type of “add\_n” output depends on storage types of inputs
- add\_n(row_sparse, row_sparse, ..) = row_sparse - add\_n(default, csr, default) = default - add\_n(any input combinations longer than 4 (>4) with at least one default type) = default - otherwise, “add\_n” falls all inputs back to default storage and generates default storage

Defined in src/operator/tensor/elementwise_sum.cc:L155

Value

out The result mx.symbol
mx.symbol.elemwise_add

**elemwise_add**: Adds arguments element-wise.

**Description**

The storage type of “elemwise_add” output depends on storage types of inputs

**Usage**

```python
mx.symbol.elemwise_add(...)
```

**Arguments**

- **lhs**: NDArray-or-Symbol first input
- **rhs**: NDArray-or-Symbol second input
- **name**: string, optional Name of the resulting symbol.

**Details**

- elemwise_add(row_sparse, row_sparse) = row_sparse - elemwise_add(csr, csr) = csr - elemwise_add(default, csr) = default - elemwise_add(csr, default) = default - elemwise_add(default, rsp) = default - elemwise_add(rsp, default) = default - otherwise, “elemwise_add” generates output with default storage

**Value**

- **out**: The result mx.symbol

mx.symbol.elemwise_div

**elemwise_div**: Divides arguments element-wise.

**Description**

The storage type of “elemwise_div” output is always dense

**Usage**

```python
mx.symbol.elemwise_div(...)
```

**Arguments**

- **lhs**: NDArray-or-Symbol first input
- **rhs**: NDArray-or-Symbol second input
- **name**: string, optional Name of the resulting symbol.
**mx.symbol.elemwise_mul**

*elemwise_mul:*Multiplies arguments element-wise.

---

**Description**

The storage type of “elemwise_mul” output depends on storage types of inputs

**Usage**

```python
mx.symbol.elemwise_mul(...)```

**Arguments**

- **lhs**: NDArray-or-Symbol first input
- **rhs**: NDArray-or-Symbol second input
- **name**: string, optional Name of the resulting symbol.

**Details**

- `elemwise_mul(default, default) = default`
- `elemwise_mul(row_sparse, row_sparse) = row_sparse`
- `elemwise_mul(default, row_sparse) = row_sparse`
- `elemwise_mul(row_sparse, default) = row_sparse`
- `elemwise_mul(csr, csr) = csr`

- otherwise, “elemwise_mul” generates output with default storage

**Value**

```python
out The result mx.symbol```

---

**mx.symbol.elemwise_sub**

*elemwise_sub:*Subtracts arguments element-wise.

---

**Description**

The storage type of “elemwise_sub” output depends on storage types of inputs

**Usage**

```python
mx.symbol.elemwise_sub(...)```
Arguments

lhs | NDArray-or-Symbol first input
 rhs | NDArray-or-Symbol second input
 name | string, optional Name of the resulting symbol.

Details

- elemwise_sub(row_sparse, row_sparse) = row_sparse - elemwise_sub(csr, csr) = csr - elemwise_sub(default, csr) = default - elemwise_sub(csr, default) = default - elemwise_sub(default, rsp) = default - elemwise_sub(rsp, default) = default - otherwise, "elemwise_sub" generates output with default storage

Value

out The result mx.symbol

mx.symbol.Embedding  

Embedding: Maps integer indices to vector representations (embeddings).

Description

This operator maps words to real-valued vectors in a high-dimensional space, called word embeddings. These embeddings can capture semantic and syntactic properties of the words. For example, it has been noted that in the learned embedding spaces, similar words tend to be close to each other and dissimilar words far apart.

Usage

mx.symbol.Embedding(...)

Arguments

data | NDArray-or-Symbol The input array to the embedding operator.
 weight | NDArray-or-Symbol The embedding weight matrix.
 input.dim | int, required Vocabulary size of the input indices.
 output.dim | int, required Dimension of the embedding vectors.
 dtype | 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8'.optional, default='float32' Data type of weight.
 sparse.grad | boolean, optional, default=0 Compute row sparse gradient in the backward calculation. If set to True, the grad’s storage type is row_sparse.
 name | string, optional Name of the resulting symbol.
Details

For an input array of shape (d1, ..., dK), the shape of an output array is (d1, ..., dK, output_dim).
All the input values should be integers in the range [0, input_dim).
If the input_dim is ip0 and output_dim is op0, then shape of the embedding weight matrix must be
(ip0, op0).
When "sparse_grad" is False, if any index mentioned is too large, it is replaced by the index that
addresses the last vector in an embedding matrix. When "sparse_grad" is True, an error will be
raised if invalid indices are found.

Examples::
input_dim = 4 output_dim = 5
// Each row in weight matrix y represents a word. So, y = (w0,w1,w2,w3) y = [[ 0., 1., 2., 3., 4.], [ 5., 6., 7., 8., 9.], [ 10., 11., 12., 13., 14.], [ 15., 16., 17., 18., 19.]]
// Input array x represents n-grams(2-gram). So, x = [(w1,w3), (w0,w2)] x = [[ 1., 3.], [ 0., 2.]]
// Mapped input x to its vector representation y. Embedding(x, y, 4, 5) = [[[ 5., 6., 7., 8., 9.], [ 15., 16., 17., 18., 19.]],
[[ 0., 1., 2., 3., 4.], [ 10., 11., 12., 13., 14.]]]
The storage type of weight can be either row_sparse or default.

.. Note::
If "sparse_grad" is set to True, the storage type of gradient w.r.t weights will be "row_sparse". Only
a subset of optimizers support sparse gradients, including SGD, AdaGrad and Adam. Note that by
default lazy updates is turned on, which may perform differently from standard updates. For more
details, please check the Optimization API at: https://mxnet.incubator.apache.org/api/python/optimization/optimization.html

Defined in src/operator/tensor/indexing_op.cc:L603

Value

out The result mx.symbol

---

mx.symbol.erf

\( \text{erf} \): Returns element-wise gauss error function of the input.

Description

Example::

Usage

mx.symbol.erf(...)

Arguments

data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.
Details

\[ \text{erf}([0, -1., 10.]) = [0., -0.8427, 1.] \]

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L886

Value

out The result mx.symbol

mx.symbol.erfinv  

\texttt{erfinv}: Returns element-wise inverse gauss error function of the input.

Description

Example::

Usage

\texttt{mx.symbol.erfinv(...)}

Arguments

\begin{itemize}
  \item \texttt{data} NDArray-or-Symbol The input array.
  \item \texttt{name} string, optional Name of the resulting symbol.
\end{itemize}

Details

\[ \text{erfinv}([0, 0.5., -1.]) = [0., 0.4769, -\text{inf}] \]

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L908

Value

out The result mx.symbol
mx.symbol.exp

exp: Returns element-wise exponential value of the input.

Description

.. math:: \exp(x) = e^x \approx 2.718^x

Usage

mx.symbol.exp(...)

Arguments

data: NDArray-or-Symbol The input array.

name: string, optional Name of the resulting symbol.

Details

Example:

```
\exp([0, 1, 2]) = [1., 2.71828175, 7.38905621]
```

The storage type of “exp” output is always dense

Defined in src/operator/tensor/elemwise_unary_op_logexp.cc:L64

Value

out The result mx.symbol

mx.symbol.expand_dims

expand_dims: Inserts a new axis of size 1 into the array shape. For example, given “x” with shape “(2,3,4)”, then “expand_dims(x, axis=1)” will return a new array with shape “(2,1,3,4)”.

Description

Defined in src/operator/tensor/matrix_op.cc:L395

Usage

mx.symbol.expand_dims(...)
mx.symbol.expm1

Arguments

- **data**: NDArray-or-Symbol Source input
- **axis**: int, required Position where new axis is to be inserted. Suppose that the input ‘NDArray’’s dimension is ‘ndim’, the range of the inserted axis is ‘[-ndim, ndim]’
- **name**: string, optional Name of the resulting symbol.

Value

- **out**: The result mx.symbol

---

mx.symbol.expm1

**expm1**: Returns “exp(x) - 1“ computed element-wise on the input.

Description

This function provides greater precision than “exp(x) - 1“ for small values of “x“.

Usage

- mx.symbol.expm1(...)

Arguments

- **data**: NDArray-or-Symbol The input array.
- **name**: string, optional Name of the resulting symbol.

Details

The storage type of “expm1“ output depends upon the input storage type:
- expm1(default) = default - expm1(row_sparse) = row_sparse - expm1(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_logexp.cc:L244

Value

- **out**: The result mx.symbol
mx.symbol.fill_element_0index

fill_element_0index: Fill one element of each line (row for python, column for R/Julia) in lhs according to index indicated by rhs and values indicated by mhs. This function assume rhs uses 0-based index.

Description
fill_element_0index: Fill one element of each line (row for python, column for R/Julia) in lhs according to index indicated by rhs and values indicated by mhs. This function assume rhs uses 0-based index.

Usage
mx.symbol.fill_element_0index(...)

Arguments
 lhs NDArray Left operand to the function.
 mhs NDArray Middle operand to the function.
 rhs NDArray Right operand to the function.
 name string, optional Name of the resulting symbol.

Value
out The result mx.symbol

mx.symbol.fix

fix: Returns element-wise rounded value to the nearest integer towards zero of the input.

Description
Example::

Usage
mx.symbol.fix(...)

Arguments
 data NDArray-or-Symbol The input array.
 name string, optional Name of the resulting symbol.
**mx.symbol.Flatten**

*Flatten:* Flattens the input array into a 2-D array by collapsing the higher dimensions. ..  

*note::* ‘Flatten’ is deprecated. Use ‘flatten’ instead. For an input array with shape “(d1, d2, ..., dk)”, ‘flatten’ operation reshapes the input array into an output array of shape “(d1, d2*...*dk)”. Note that the behavior of this function is different from numpy.ndarray.flatten, which behaves similar to mxnet.ndarray.reshape((-1,)).  

*Example:: x = [[ [1,2,3], [4,5,6], [7,8,9] ], [ [1,2,3], [4,5,6], [7,8,9] ]], flatten(x) = [[ 1., 2., 3., 4., 5., 6., 7., 8., 9.], [ 1., 2., 3., 4., 5., 6., 7., 8., 9.]]

**Description**

Defined in src/operator/tensor/matrix_op.cc:L250

**Usage**

mx.symbol.Flatten(...)

**Arguments**

- **data**: NDArray-or-Symbol Input array.
- **name**: string, optional Name of the resulting symbol.

**Value**

out The result mx.symbol
mx.symbol.flatten  

flatten: Flattens the input array into a 2-D array by collapsing the higher dimensions.  

.. note:: ‘Flatten’ is deprecated. Use ‘flatten’ instead. For an input array with shape “(d1, d2, ..., dk)”, 
‘flatten’ operation reshapes the input array into an output array of shape “(d1, d2*...*dk)”. Note that the behavior of this function is different from numpy.ndarray.flatten, which behaves similar to mxnet.ndarray.reshape((-1,)). Example:: x = [[ 1, 2, 3], [ 4, 5, 6], [ 7, 8, 9], [ 1, 2, 3], [ 4, 5, 6], [ 7, 8, 9]], flatten(x) = [[ 1, 2, 3, 4, 5, 6, 7, 8, 9], [ 1, 2, 3, 4, 5, 6, 7, 8, 9]]

Description

Defined in src/operator/tensor/matrix_op.cc:L250

Usage

mx.symbol.flatten(...)

Arguments

data NDArray-or-Symbol Input array.

name string, optional Name of the resulting symbol.

Value

out The result mx.symbol

mx.symbol.flip  

flip: Reverses the order of elements along given axis while preserving array shape. Note: reverse and flip are equivalent. We use reverse in the following examples. Examples:: x = [[ 0, 1, 2, 3, 4], [ 5, 6, 7, 8, 9]] reverse(x, axis=0) = [[ 5, 6, 7, 8, 9], [ 0, 1, 2, 3, 4]] reverse(x, axis=1) = [[ 4, 3, 2, 1, 0], [ 9, 8, 7, 6, 5]]

Description

Defined in src/operator/tensor/matrix_op.cc:L832

Usage

mx.symbol.flip(...
Arguments

- **data**: NDArray-or-Symbol Input data array
- **axis**: Shape(tuple), required The axis which to reverse elements.
- **name**: string, optional Name of the resulting symbol.

Value

- **out**: The result mx.symbol

---

**mx.symbol.floor**

Returns element-wise floor of the input.

Description

The floor of the scalar x is the largest integer i, such that i <= x.

Usage

mx.symbol.floor(...)

Arguments

- **data**: NDArray-or-Symbol The input array.
- **name**: string, optional Name of the resulting symbol.

Details

Example:

floor([-2.1, -1.9, 1.5, 1.9, 2.1]) = [-3., -2., 1., 1., 2.]

The storage type of “floor“ output depends upon the input storage type:

- floor(default) = default - floor(row_sparse) = row_sparse - floor(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L836

Value

- **out**: The result mx.symbol
mx.symbol.ftml_update


Description

.. math::

Usage

mx.symbol.ftml_update(...)

Arguments

weight NDArray-or-Symbol Weight
grad NDArray-or-Symbol Gradient
d NDArray-or-Symbol Internal state “d_t”
v NDArray-or-Symbol Internal state “v_t”
z NDArray-or-Symbol Internal state “z_t”
lr float, required Learning rate.
beta1 float, optional, default=0.600000024 Generally close to 0.5.
beta2 float, optional, default=0.999000013 Generally close to 1.
epsilon double, optional, default=9.9999999392252903e-09 Epsilon to prevent div 0.
t int, required Number of update.
wd float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.grad float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
name string, optional Name of the resulting symbol.

Details

\[ g_t = \nabla J(W_{t-1}) \backslash v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \backslash d_t = \frac{1 - \beta_1_t}{\sqrt{\frac{v_t}{1 - \beta_2^t} + \text{epsilon}}} \sigma_t \backslash \frac{\text{sqrt}}{\sigma_t = \frac{1}{1 - \beta_1_t}} d_t - \beta_1 d_{t-1} z_t = \beta_1 z_{t-1} + (1 - \beta_1^t) \]

\[ g_t - \text{sigma}_t W_{t-1} W_t = - \frac{z_t}{\text{sqrt}} d_t \]

Defined in src/operator/optimizer_op.cc:L631

Value

out The result mx.symbol
mx.symbol.ftrl_update


Description

It updates the weights using::

Usage

mx.symbol.ftrl_update(...)

Arguments

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **z**: NDArray-or-Symbol z
- **n**: NDArray-or-Symbol Square of grad
- **lr**: float, required Learning rate
- **lamda1**: float, optional, default=0.00999999978 The L1 regularization coefficient.
- **beta**: float, optional, default=1 Per-Coordinate Learning Rate beta.
- **wd**: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- **name**: string, optional Name of the resulting symbol.

Details

rescaled_grad = clip(grad * rescale_grad, clip_gradient) z += rescaled_grad - (sqrt(n + rescaled_grad**2) - sqrt(n)) * weight / learning_rate n += rescaled_grad**2 w = (sign(z) * lamda1 - z) / ((beta + sqrt(n)) / learning_rate + wd) * (abs(z) > lamda1)

If w, z and n are all of “row_sparse” storage type, only the row slices whose indices appear in grad.indices are updated (for w, z and n):


Defined in src/operator/optimizer_op.cc:L867
Value

out The result mx.symbol

mx.symbol.FullyConnected

FullyConnected: Applies a linear transformation: \( Y = XW^T + b \).

Description

If “flatten” is set to be true, then the shapes are:

Usage

mx.symbol.FullyConnected(...)

Arguments

data NDArray-or-Symbol Input data.
weight NDArray-or-Symbol Weight matrix.
bias NDArray-or-Symbol Bias parameter.
um.hidden int, required Number of hidden nodes of the output.
no.bias boolean, optional, default=0 Whether to disable bias parameter.
flatten boolean, optional, default=1 Whether to collapse all but the first axis of the input data tensor.
name string, optional Name of the resulting symbol.

Details

- **data**: ‘(batch_size, x1, x2, ..., xn)’ - **weight**: ‘(num_hidden, x1 * x2 * ... * xn)’ - **bias**: ‘(num_hidden,)’ - **out**: ‘(batch_size, num_hidden)’

If “flatten” is set to be false, then the shapes are:

- **data**: ‘(x1, x2, ..., xn, input_dim)’ - **weight**: ‘(num_hidden, input_dim)’ - **bias**: ‘(num_hidden,)’ - **out**: ‘(x1, x2, ..., xn, num_hidden)’

The learnable parameters include both “weight” and “bias”.

If “no.bias” is set to be true, then the “bias” term is ignored.

.. Note::

The sparse support for FullyConnected is limited to forward evaluation with ‘row_sparse’ weight and bias, where the length of ‘weight.indices’ and ‘bias.indices’ must be equal to ‘num_hidden’.

This could be useful for model inference with ‘row_sparse’ weights trained with importance sampling or noise contrastive estimation.

To compute linear transformation with ’csr’ sparse data, sparse.dot is recommended instead of sparse.FullyConnected.

Defined in src/operator/nn/fully_connected.cc:L287
mx.symbol.gamma

**Description**

The storage type of “gamma“ output is always dense

**Usage**

```
mx.symbol.gamma(...)```

**Arguments**

- `data` : NDArray-or-Symbol The input array.
- `name` : string, optional Name of the resulting symbol.

**Value**

out The result mx.symbol

mx.symbol.gammaln

**Description**

Returns element-wise log of the absolute value of the gamma function of the input.

**Usage**

```
mx.symbol.gammaln(...)```

**Arguments**

- `data` : NDArray-or-Symbol The input array.
- `name` : string, optional Name of the resulting symbol.

**Value**

out The result mx.symbol
mx.symbol.gather_nd

**Description**

Given ‘data’ with shape ‘(X_0, X_1, ..., X_N-1)’ and indices with shape ‘(M, Y_0, ..., Y_K-1)’, the output will have shape ‘(Y_0, ..., Y_K-1, X_M, ..., X_N-1)’, where ‘M <= N’. If ‘M == N’, output shape will simply be ‘(Y_0, ..., Y_K-1)’.

**Usage**

```python
mx.symbol.gather_nd(...)
```

**Arguments**

- `data` : NDArray-or-Symbol data
- `indices` : NDArray-or-Symbol indices
- `name` : string, optional Name of the resulting symbol.

**Details**

The elements in output is defined as follows:

```python
output[y_0, ..., y_K-1, x_M, ..., x_N-1] = data[indices[0, y_0, ..., y_K-1], ..., indices[M-1, y_0, ..., y_K-1], x_M, ..., x_N-1]
```

**Examples**:

```python
data = [[0, 1], [2, 3]] indices = [[1, 1, 0], [0, 1, 0]] gather_nd(data, indices) = [2, 3, 0]
data = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]] indices = [[0, 1], [1, 0]] gather_nd(data, indices) = [[3, 4], [5, 6]]
```

**Value**

```python
out The result mx.symbol
```

mx.symbol.GridGenerator

**Description**

GridGenerator:Generates 2D sampling grid for bilinear sampling.
Usage

mx.symbol.GridGenerator(...)

Arguments

data        NDArray-or-Symbol Input data to the function.
transform.type  'affine', 'warp', required The type of transformation. For 'affine', input data should be an affine matrix of size (batch, 6). For 'warp', input data should be an optical flow of size (batch, 2, h, w).
target.shape  Shape(tuple), optional, default=[0,0] Specifies the output shape (H, W). This is required if transformation type is 'affine'. If transformation type is 'warp', this parameter is ignored.
name         string, optional Name of the resulting symbol.

Value

out The result mx.symbol

mx.symbol.Group

Create a symbol that groups symbols together.

Description

Create a symbol that groups symbols together.

Usage

mx.symbol.Group(...)

Arguments

kwarg        Variable length of symbols or list of symbol.

Value

The result symbol
**mx.symbol.GroupNorm**

*GroupNorm: Group normalization.*

**Description**

The input channels are separated into “num_groups” groups, each containing “num_channels / num_groups” channels. The mean and standard-deviation are calculated separately over the each group.

**Usage**

```
mx.symbol.GroupNorm(...)
```

**Arguments**

- **data**  
  NDArray-or-Symbol Input data
- **gamma**  
  NDArray-or-Symbol gamma array
- **beta**  
  NDArray-or-Symbol beta array
- **num.groups**  
  int, optional, default='1' Total number of groups.
- **eps**  
  float, optional, default=9.99999975e-06 An ‘epsilon’ parameter to prevent division by 0.
- **output.mean.var**  
  boolean, optional, default=0 Output the mean and std calculated along the given axis.
- **name**  
  string, optional Name of the resulting symbol.

**Details**

.. math::

   data = data.reshape((N, num_groups, C // num_groups, ...))
   out = \frac{data - mean(data, axis)}{sqrtvar(data, axis)} + \epsilon * gamma + beta

Both “gamma” and “beta” are learnable parameters.

Defined in src/operator/nn/group_norm.cc:L77

**Value**

```
out The result mx.symbol
```

mx.symbol.hard_sigmoid

**hard_sigmoid:** Computes hard sigmoid of \( x \) element-wise.

**Description**

.. math:: y = \max(0, \min(1, \alpha \times x + \beta))

**Usage**

mx.symbol.hard_sigmoid(...)

**Arguments**

- **data**: NDArray-or-Symbol The input array.
- **alpha**: float, optional, default=0.200000003 Slope of hard sigmoid
- **beta**: float, optional, default=0.5 Bias of hard sigmoid.
- **name**: string, optional Name of the resulting symbol.

**Details**

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L161

**Value**

out The result mx.symbol

mx.symbol.identity

**identity:** Returns a copy of the input.

**Description**

From:src/operator/tensor/elemwise_unary_op_basic.cc:244

**Usage**

mx.symbol.identity(...)

**Arguments**

- **data**: NDArray-or-Symbol The input array.
- **name**: string, optional Name of the resulting symbol.

**Value**

out The result mx.symbol
mx.symbol.IdentityAttachKLSparseReg

IdentityAttachKLSparseReg: Apply a sparse regularization to the output a sigmoid activation function.

Description

IdentityAttachKLSparseReg: Apply a sparse regularization to the output a sigmoid activation function.

Usage

mx.symbol.IdentityAttachKLSparseReg(...)

Arguments

data NDArray-or-Symbol Input data.
sparseness.target float, optional, default=0.100000001 The sparseness target
penalty float, optional, default=0.00100000005 The tradeoff parameter for the sparseness penalty
momentum float, optional, default=0.899999976 The momentum for running average
name string, optional Name of the resulting symbol.

Value

out The result mx.symbol

mx.symbol.im2col

im2col: Extract sliding blocks from input array.

Description

This operator is used in vanilla convolution implementation to transform the sliding blocks on image to column matrix, then the convolution operation can be computed by matrix multiplication between column and convolution weight. Due to the close relation between im2col and convolution, the concept of **kernel**, **stride**, **dilate** and **pad** in this operator are inherited from convolution operation.

Usage

mx.symbol.im2col(...)
Arguments

- **data**: NDArray-or-Symbol Input array to extract sliding blocks.
- **kernel**: Shape(tuple), required Sliding kernel size: (w,), (h, w) or (d, h, w).
- **stride**: Shape(tuple), optional, default=[] The stride between adjacent sliding blocks in spatial dimension: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.
- **dilate**: Shape(tuple), optional, default=[] The spacing between adjacent kernel points: (w,), (h, w) or (d, h, w). Defaults to 1 for each dimension.
- **pad**: Shape(tuple), optional, default=[] The zero-value padding size on both sides of spatial dimension: (w,), (h, w) or (d, h, w). Defaults to no padding.
- **name**: string, optional Name of the resulting symbol.

Details

Given the input data of shape :math:`(N, C, \ast)`, where :math:`N` is the batch size, :math:`C` is the channel size, and :math:`\ast` is the arbitrary spatial dimension, the output column array is always with shape :math:`(N, C \times \prod(\text{kernel}), W)`, where :math:`C \times \prod(\text{kernel})` is the block size, and :math:`W` is the block number which is the spatial size of the convolution output with same input parameters. Only 1-D, 2-D and 3-D of spatial dimension is supported in this operator.

Defined in src/operator/nn/im2col.cc:L100

Value

- **out**: The result mx.symbol

---

**mx.symbol.infer.shape**  
Inference the shape of arguments, outputs, and auxiliary states.

Description

Inference the shape of arguments, outputs, and auxiliary states.

Usage

```python
mx.symbol.infer.shape(symbol, ...)
```

Arguments

- **symbol**: The mx.symbol object
**mx.symbol.InstanceNorm**

*InstanceNorm:* Applies instance normalization to the n-dimensional input array.

### Description

This operator takes an n-dimensional input array where (n>2) and normalizes the input using the following formula:

\[
\text{out} = \frac{x - \text{mean}[\text{data}]}{\sqrt{\text{Var}[\text{data}]}} + \epsilon \times \gamma + \beta
\]

This layer is similar to batch normalization layer (`BatchNorm`) with two differences: first, the normalization is carried out per example (instance), not over a batch. Second, the same normalization is applied both at test and train time. This operation is also known as 'contrast normalization'.

If the input data is of shape [batch, channel, spatial_dim1, spacial_dim2, ...], ‘gamma’ and ‘beta’ parameters must be vectors of shape [channel].

This implementation is based on this paper [1].


### Usage

```python
mx.symbol.InstanceNorm(...)```

### Arguments

- **data**: NDArray-or-Symbol An n-dimensional input array (n > 2) of the form [batch, channel, spatial_dim1, spatial_dim2, ...].
- **gamma**: NDArray-or-Symbol A vector of length 'channel', which multiplies the normalized input.
- **beta**: NDArray-or-Symbol A vector of length 'channel', which is added to the product of the normalized input and the weight.
- **eps**: float, optional, default=0.00100000005 An ‘epsilon’ parameter to prevent division by 0.
- **name**: string, optional Name of the resulting symbol.

### Details

```
.. math::
    \text{out} = \frac{x - \text{mean}[\text{data}]}{\sqrt{\text{Var}[\text{data}]}} + \epsilon \times \gamma + \beta
```

This layer is similar to batch normalization layer (`BatchNorm`) with two differences: first, the normalization is carried out per example (instance), not over a batch. Second, the same normalization is applied both at test and train time. This operation is also known as 'contrast normalization'.

If the input data is of shape [batch, channel, spatial_dim1, spacial_dim2, ...], ‘gamma’ and ‘beta’ parameters must be vectors of shape [channel].

This implementation is based on this paper [1].


### Examples

```
// Input of shape (2,1,2) x = [[[ 1.1, 2.2]], [[ 3.3, 4.4]]]
// gamma parameter of length 1 gamma = [1.5]
// beta parameter of length 1 beta = [0.5]
```
// Instance normalization is calculated with the above formula
InstanceNorm(x, gamma, beta) = [[[-0.997527, 1.99752665]], [[-0.99752653, 1.99752724]]]
Defined in src/operator/instance_norm.cc:L95

Value

out The result mx.symbol

mx.symbol.khatri_rao

khatri_rao: Computes the Khatri-Rao product of the input matrices.

Description

Given a collection of :math:`n` input matrices,

Usage

mx.symbol.khatri_rao(...)

Arguments

args NDArray-or-Symbol[] Positional input matrices
name string, optional Name of the resulting symbol.

Details

.. math:: A_1 \in \mathbb{R}^{M_1 \times N}, \ldots, A_n \in \mathbb{R}^{M_n \times N},

the (column-wise) Khatri-Rao product is defined as the matrix,

.. math:: X = A_1 \otimes \cdots \otimes A_n \in \mathbb{R}^{(M_1 \cdots M_n) \times N},

where the :math:`k` th column is equal to the column-wise outer product :math:`A_1_k \otimes \cdots \otimes A_n_k` where :math:`A_i_k` is the :math:`k`th column of the :math:`i`th matrix.

Example::

```python
>>> A = mx.nd.array([[1, -1], [2, -3]])
>>> B = mx.nd.array([[1, 4], [2, 5], [3, 6]])
>>> C = mx.nd.khatri_rao(A, B)
>>> print(C.asnumpy())
[[ 1. -4.]
 [ 2. -5.]
 [ 3. -6.]
 [ 2. -12.]
 [ 4. -15.]
 [ 6. -18.]]
```

Defined in src/operator/contrib/krprod.cc:L108

Value

out The result mx.symbol
mx.symbol.L2Normalization

L2Normalization: Normalize the input array using the L2 norm.

Description
For 1-D NDArray, it computes:

Usage
mx.symbol.L2Normalization(...)

Arguments
- **data**: NDArray-or-Symbol Input array to normalize.
- **eps**: float, optional, default=1.00000001e-10 A small constant for numerical stability.
- **mode**: 'channel', 'instance', 'spatial', optional, default='instance' Specify the dimension along which to compute L2 norm.
- **name**: string, optional Name of the resulting symbol.

Details
out = data / sqrt(sum(data ** 2) + eps)
For N-D NDArray, if the input array has shape (N, N, ..., N), with “mode“ = “instance“, it normalizes each instance in the multidimensional array by its L2 norm.: for i in 0...N out[i,:,:,:,:] = data[i,:,:,:,:] / sqrt(sum(data[i,:,:,:,:] ** 2) + eps) with “mode“ = “channel", it normalizes each channel in the array by its L2 norm.: for i in 0...N out[:,i,:,:,:,:] = data[:,i,:,:,:,:] / sqrt(sum(data[:,i,:,:,:,:] ** 2) + eps) with “mode“ = “spatial", it normalizes the cross channel norm for each position in the array by its L2 norm.: for dim in 2...N for i in 0...N out[.....,i,...] = take(out, indices=i, axis=dim) / sqrt(sum(take(out, indices=i, axis=dim) ** 2) + eps) -dim-
Example:
x = [[[1,2], [3,4]], [[2,2], [5,6]]]
L2Normalization(x, mode='instance') =[[[ 0.18257418 0.36514837] [ 0.54772252 0.73029673] [0.24077171 0.24077171] [ 0.60192931 0.72231513]]
L2Normalization(x, mode='channel') =[[[ 0.31622776 0.44721359] [ 0.94868326 0.89442718] [0.37139067 0.31622776] [ 0.92847669 0.94868326]]
L2Normalization(x, mode='spatial') =[[[ 0.44721359 0.89442718] [ 0.60000002 0.80000001] [0.70710677 0.70710677] [ 0.6401844 0.76822126]]
Defined in src/operator/l2_normalization.cc:L196
**Value**

out The result `mx.symbol`

---

```python
mx.symbol.lamb_update_phase1

lamb_update_phase1: Phase I of lamb update it performs the following operations and returns g:
```

---

**Description**


**Usage**

```python
mx.symbol.lamb_update_phase1(...)
```

**Arguments**

- `weight` : NDArray-or-Symbol Weight
- `grad` : NDArray-or-Symbol Gradient
- `mean` : NDArray-or-Symbol Moving mean
- `var` : NDArray-or-Symbol Moving variance
- `beta1` : float, optional, default=0.899999976 The decay rate for the 1st moment estimates.
- `beta2` : float, optional, default=0.999000013 The decay rate for the 2nd moment estimates.
- `epsilon` : float, optional, default=9.99999997e-07 A small constant for numerical stability.
- `t` : int, required Index update count.
- `bias_correction` : boolean, optional, default=1 Whether to use bias correction.
- `wd` : float, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- `rescale.grad` : float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- `clip.gradient` : float, optional, default=1 Clip gradient to the range of [-clip_gradient, clip_gradient]
  If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- `name` : string, optional Name of the resulting symbol.
mx.symbol.lamb_update_phase2

Details

.. math:: \begin{gather*} \text{grad} = \text{grad} \times \text{rescale\_grad} \text{ if (grad} < \text{-clip\_gradient}) \text{ then grad} = \text{-clip\_gradient} \\
\text{if (grad} > \text{clip\_gradient}) \text{ then grad} = \text{clip\_gradient} \\
\text{mean} = \beta_1 \times \text{mean} + (1 - \beta_1) \times \text{grad}; \text{variance} = \beta_2 \times \text{variance} + (1 - \beta_2) \times \text{grad}^2; \\
\text{if (bias\_correction)} \text{ then mean\_hat} = \text{mean} / (1 - \beta_1^t); \text{var\_hat} = \text{var} / (1 - \beta_2^t); \text{g} = \text{mean\_hat} / (\text{var\_hat}^{1/2} + \epsilon) + \text{wd} \times \text{weight}; \text{else g} = \text{mean} / (\text{var\_data}^{1/2} + \epsilon) + \text{wd} \times \text{weight}; \\
\end{gather*}

Defined in src/operator/optimizer_op.cc:L944

Value

out The result mx.symbol

mx.symbol.lamb_update_phase2

lamb_update_phase2: Phase II of lamb update it performs the following operations and updates grad.

Description


Usage

mx.symbol.lamb_update_phase2(...)

Arguments

weight NDArray-or-Symbol Weight

g NDArray-or-Symbol Output of lamb_update_phase 1

r1 NDArray-or-Symbol r1

r2 NDArray-or-Symbol r2

lr float, required Learning rate

lower\_bound float, optional, default=-1 Lower limit of norm of weight. If lower_bound <= 0, Lower limit is not set

upper\_bound float, optional, default=-1 Upper limit of norm of weight. If upper_bound <= 0, Upper limit is not set

name string, optional Name of the resulting symbol.

Details

.. math:: \begin{gather*} \text{if (lower\_bound} \geq 0) \text{ then r1} = \max(r1, \text{lower\_bound}) \text{ if (upper\_bound} \geq 0) \text{ then r1} = \max(r1, \text{upper\_bound}) \\
\text{if (r1} \geq 0 \text{ or r2} = 0) \text{ then lr} = \text{lr} \text{ else lr} = \text{lr} * (r1/r2) \text{ weight} = \text{weight} - \text{lr} \times \text{g} \end{gather*}

Defined in src/operator/optimizer_op.cc:L983
mx.symbol.LayerNorm

**Description**

Normalizes the channels of the input tensor by mean and variance, and applies a scale “gamma” as well as offset “beta”.

**Usage**

mx.symbol.LayerNorm(...)

**Arguments**

- **data**: NDArray-or-Symbol Input data to layer normalization
- **gamma**: NDArray-or-Symbol gamma array
- **beta**: NDArray-or-Symbol beta array
- **axis**: int, optional, default='-1' The axis to perform layer normalization. Usually, this should be axis of the channel dimension. Negative values means indexing from right to left.
- **eps**: float, optional, default=9.99999975e-06 An ‘epsilon’ parameter to prevent division by 0.
- **output.mean.var**: boolean, optional, default=0 Output the mean and std calculated along the given axis.
- **name**: string, optional Name of the resulting symbol.

**Details**

Assume the input has more than one dimension and we normalize along axis 1. We first compute the mean and variance along this axis and then compute the normalized output, which has the same shape as input, as following:

```math
out = \frac{data - mean(data, axis)}{sqrtvar(data, axis) + \epsilon} * gamma + beta
```

Both “gamma” and “beta” are learnable parameters.

Unlike BatchNorm and InstanceNorm, the *mean* and *var* are computed along the channel dimension.

Assume the input has size *k* on axis 1, then both “gamma” and “beta” have shape *(k,)*. If “output_mean_var” is set to be true, then outputs both “data_mean” and “data_std”. Note that no gradient will be passed through these two outputs.
The parameter “axis” specifies which axis of the input shape denotes the ‘channel’ (separately
normalized groups). The default is -1, which sets the channel axis to be the last item in the input
shape.

Defined in src/operator/nn/layer_norm.cc:L158

Value

out The result mx.symbol

mx.symbol.LeakyReLU  LeakyReLU:Applies Leaky rectified linear unit activation elementwise to the input.

Description

Leaky ReLUs attempt to fix the "dying ReLU" problem by allowing a small ‘slope’ when the input
is negative and has a slope of one when input is positive.

Usage

mx.symbol.LeakyReLU(…)

Arguments

data  NDArray-or-Symbol Input data to activation function.
gamma  NDArray-or-Symbol Input data to activation function.
act.type  'elu', 'gelu', 'leaky', 'prelu', 'rrelu', 'selu'.optional, default='leaky' Activation
function to be applied.
slope  float, optional, default=0.25 Init slope for the activation. (For leaky and elu only)
lower.bound  float, optional, default=0.125 Lower bound of random slope. (For rrelu only)
upper.bound  float, optional, default=0.333999991 Upper bound of random slope. (For rrelu
only)
name  string, optional Name of the resulting symbol.

Details

The following modified ReLU Activation functions are supported:
- *elu*: Exponential Linear Unit. ‘y = x > 0 ? x : slope * (exp(x)-1)’ - *elu*: Scaled Ex-
ponential Linear Unit. ‘y = lambda * (x > 0 ? x : alpha * (exp(x) - 1))’ where
*lambda = 1.0507009873554804934193349852946* and *alpha = 1.6732632423543772848170429916717*.
- *leaky*: Leaky ReLU. ‘y = x > 0 ? x : slope * x’ - *leaky*: Leaky ReLU. ‘y = x > 0 ? x : slope * x’ - *prelu*: Parametric ReLU. This is same as
*leaky* except that ‘slope’ is learnt during training. - *prelu*: Parametric ReLU. This is same as *leaky*
but the ‘slope’ is uniformly and randomly chosen from *[lower_bound, upper_bound)* for training,
while fixed to be *(lower_bound+upper_bound)/2* for inference.

Defined in src/operator/leaky_relu.cc:L161
Value

out The result mx.symbol

mx.symbol.linalg_det

linalg_det: Compute the determinant of a matrix. Input is a tensor *A* of dimension *n >= 2*.

Description

If *n=2*, *A* is a square matrix. We compute:

Usage

mx.symbol.linalg_det(...)

Arguments

A NDArray-or-Symbol Tensor of square matrix
name string, optional Name of the resulting symbol.

Details

*out* = *det(A)*

If *n>2*, *det* is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only. .. note:: There is no gradient
backwarded when A is non-invertible (which is equivalent to det(A) = 0) because zero is rarely hit
upon in float point computation and the Jacobi’s formula on determinant gradient is not computa-
tionally efficient when A is non-invertible.

Examples::

Single matrix determinant A = [[1., 4.], [2., 3.]] det(A) = [-5.]

Batch matrix determinant A = [[[1., 4.], [2., 3.]], [[2., 3.], [1., 4.]]] det(A) = [-5., 5.]

Defined in src/operator/tensor/la_op.cc:L975

Value

out The result mx.symbol
Description

If \( n=2 \), then \( A \) represents a single square matrix which diagonal elements get extracted as a 1-dimensional tensor.

Usage

\[
\text{mx.symbol.linalg_extractdiag(...)}
\]

Arguments

- **A**: NDArray-or-Symbol Tensor of square matrices
- **offset**: int, optional, default='0' Offset of the diagonal versus the main diagonal. 0 corresponds to the main diagonal, a negative/positive value to diagonals below/above the main diagonal.
- **name**: string, optional Name of the resulting symbol.

Details

If \( n>2 \), then \( A \) represents a batch of square matrices on the trailing two dimensions. The extracted diagonals are returned as an \( n-1 \)-dimensional tensor.

.. note:: The operator supports float32 and float64 data types only.

Examples:

Single matrix diagonal extraction

\[
A = \begin{bmatrix}1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}
\]

\[
\text{extractdiag}(A) = [1.0, 4.0]
\]

\[
\text{extractdiag}(A, 1) = [2.0]
\]

Batch matrix diagonal extraction

\[
A = \begin{bmatrix}\begin{bmatrix}1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}, \begin{bmatrix}5.0 & 6.0 \\ 7.0 & 8.0 \end{bmatrix}\end{bmatrix}
\]

\[
\text{extractdiag}(A) = \begin{bmatrix}1.0, 4.0, 5.0, 8.0\end{bmatrix}
\]

Defined in src/operator/tensor/la_op.cc:L495

Value

out The result mx.symbol
mx.symbol.linalg_extracttrian

`linalg_extracttrian`: Extracts a triangular sub-matrix from a square matrix. Input is a tensor \( A \) of dimension \( n \geq 2 \).

**Description**

If \( n=2 \), then \( A \) represents a single square matrix from which a triangular sub-matrix is extracted as a 1-dimensional tensor.

**Usage**

mx.symbol.linalg_extracttrian(...)

**Arguments**

- **A**: NDArray-or-Symbol Tensor of square matrices
- **offset**: int, optional, default='0' Offset of the diagonal versus the main diagonal. 0 corresponds to the main diagonal, a negative/positive value to diagonals below/above the main diagonal.
- **lower**: boolean, optional, default=1 Refer to the lower triangular matrix if lower=true, refer to the upper otherwise. Only relevant when offset=0
- **name**: string, optional Name of the resulting symbol.

**Details**

If \( n \geq 2 \), then \( A \) represents a batch of square matrices on the trailing two dimensions. The extracted triangular sub-matrices are returned as an \( (n-1) \)-dimensional tensor.

The **offset** and **lower** parameters determine the triangle to be extracted:

- When **offset = 0** either the lower or upper triangle with respect to the main diagonal is extracted depending on the value of parameter **lower**.  - When **offset = k > 0** the upper triangle with respect to the k-th diagonal above the main diagonal is extracted. - When **offset = k < 0** the lower triangle with respect to the k-th diagonal below the main diagonal is extracted.

.. note:: The operator supports float32 and float64 data types only.

**Examples**:

Single triagonal extraction

\[
A = \begin{bmatrix}
1.0 & 2.0 \\
3.0 & 4.0
\end{bmatrix}
\]

extracttrian(A) = [1.0, 3.0, 4.0] extracttrian(A, lower=False) = [1.0, 2.0, 4.0] extracttrian(A, 1) = [2.0] extracttrian(A, -1) = [3.0]

Batch triagonal extraction

\[
A = \begin{bmatrix}
[1.0, 2.0, [3.0, 4.0]], [5.0, 6.0], [7.0, 8.0]]
\]

extracttrian(A) = [[1.0, 3.0, 4.0], [5.0, 7.0, 8.0]]

Defined in src/operator/tensor/la_op.cc:L605

**Value**

out The result mx.symbol
mx.symbol.linalg_gelqf

**linalg_gelqf**: LQ factorization for general matrix. Input is a tensor *A* of dimension *n >= 2*.

**Description**

If *n=2*, we compute the LQ factorization (LAPACK *gelqf*, followed by *orglq*). *A* must have shape *(x, y)* with *x <= y*, and must have full rank *=x*. The LQ factorization consists of *L* with shape *(x, x)* and *Q* with shape *(x, y)*, so that:

**Usage**

mx.symbol.linalg_gelqf(...)

**Arguments**

- **A**
  NDArray-or-Symbol Tensor of input matrices to be factorized
- **name**
  string, optional Name of the resulting symbol.

**Details**

*A* = *L* \* *Q*

Here, *L* is lower triangular (upper triangle equal to zero) with nonzero diagonal, and *Q* is row-orthonormal, meaning that

*Q* \* *Q*sup:`T`

is equal to the identity matrix of shape *(x, x)*.

If *n>2*, *gelqf* is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

**Examples**:

Single LQ factorization A = [[[1., 2., 3.], [4., 5., 6.]]] Q, L = gelqf(A) Q = [[-0.26726124, -0.53452248, -0.80178373], [0.87287156, 0.21821789, -0.43643578]] L = [[-3.74165739, 0.], [-8.55235974, 1.96396101]]

Batch LQ factorization A = [[[1., 2., 3.], [4., 5., 6.]], [[7., 8., 9.], [10., 11., 12.]]] Q, L = gelqf(A) Q = [[[-0.26726124, -0.53452248, -0.80178373], [0.87287156, 0.21821789, -0.43643578]], [[-0.50257071, -0.57436653, -0.64616234], [0.7620735, 0.05862104, -0.64483142]]] L = [[[-3.74165739, 0.], [-13.92838828, 0.], [-19.09768702, 0.52758934]]

Defined in src/operator/tensor/la_op.cc:L798

**Value**

out The result mx.symbol
mx.symbol.linalg_gemm

linalg_gemm: Performs general matrix multiplication and accumulation. Input are tensors \(*A*, \*B*, \*C*\), each of dimension \(*n >= 2*\) and having the same shape on the leading \(*n-2*\) dimensions.

Description

If \(*n=2*\), the BLAS3 function \(*gemm*\) is performed:

Usage

mx.symbol.linalg_gemm(...)

Arguments

- **A**: NDArray-or-Symbol Tensor of input matrices
- **B**: NDArray-or-Symbol Tensor of input matrices
- **C**: NDArray-or-Symbol Tensor of input matrices
- **transpose.a**: boolean, optional, default=0 Multiply with transposed of first input (A).
- **transpose.b**: boolean, optional, default=0 Multiply with transposed of second input (B).
- **alpha**: double, optional, default=1 Scalar factor multiplied with A*B.
- **beta**: double, optional, default=1 Scalar factor multiplied with C.
- **axis**: int, optional, default=':0' Axis corresponding to the matrix rows.
- **name**: string, optional Name of the resulting symbol.

Details

\[*out* = \*alpha* \* \*op\*\(^*A*\) \* \*op\*\(^*B*\) + \*beta* \* \*C*\]

Here, \(*alpha*\) and \(*beta*\) are scalar parameters, and \(*op()*\) is either the identity or matrix transposition (depending on \(*transpose_a*\, \*transpose_b*\)).

If \(*n>2*\), \(*gemm*\) is performed separately for a batch of matrices. The column indices of the matrices are given by the last dimensions of the tensors, the row indices by the axis specified with the \(*axis*\) parameter. By default, the trailing two dimensions will be used for matrix encoding.

For a non-default axis parameter, the operation performed is equivalent to a series of swapaxes/gemm/swapaxes calls. For example let \(*A*, \*B*, \*C*\) be 5 dimensional tensors. Then \(\text{gemm}(*A*, \*B*, \*C*, \text{axis}=1)\) is equivalent to the following without the overhead of the additional swapaxis operations::

\[
A1 = \text{swapaxes}(A, \text{dim1}=1, \text{dim2}=3) \\
B1 = \text{swapaxes}(B, \text{dim1}=1, \text{dim2}=3) \\
C = \text{gemm}(A1, B1, C) \\
C = \text{swapaxis}(C, \text{dim1}=1, \text{dim2}=3)
\]

When the input data is of type float32 and the environment variables MXNET_CUDA_ALLOW_TENSOR_CORE and MXNET_CUDA_TENSOR_OP_MATH_ALLOW_CONVERSION are set to 1, this operator will try to use pseudo-float16 precision (float32 math with float16 I/O) precision in order to use Tensor Cores on suitable NVIDIA GPUs. This can sometimes give significant speedups.

.. note:: The operator supports float32 and float64 data types only.
Examples:
Single matrix multiply-add $A = \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \end{bmatrix}$ $B = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}$ $C = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}$
$\text{gemm}(A, B, C, \text{transpose}_b=True, \alpha=2.0, \beta=10.0) = \begin{bmatrix} 14.0 & 14.0 & 14.0 \\ 14.0 & 14.0 & 14.0 \end{bmatrix}$
Batch matrix multiply-add $A = \begin{bmatrix} [1.0, 1.0] \\ [0.1, 0.1] \end{bmatrix}$ $B = \begin{bmatrix} [1.0, 1.0] \\ [0.1, 0.1] \end{bmatrix}$ $C = \begin{bmatrix} [10.0] \\ [0.01] \end{bmatrix}$
$\text{gemm}(A, B, C, \text{transpose}_b=True, \alpha=2.0, \beta=10.0) = \begin{bmatrix} [104.0] \\ [0.14] \end{bmatrix}$
Defined in src/operator/tensor/la_op.cc:L89

Value
out The result mx.symbol

mx.symbol.linalg_gemm2

**linalg_gemm2**: Performs general matrix multiplication. Input are tensors $A$, $B$, each of dimension $n >= 2$ and having the same shape on the leading $n-2$ dimensions.

Description
If $n=2$, the BLAS3 function $\text{gemm}$ is performed:

Usage
mx.symbol.linalg_gemm2(...)

Arguments
- **A**: NDArray-or-Symbol Tensor of input matrices
- **B**: NDArray-or-Symbol Tensor of input matrices
- **transpose.a**: boolean, optional, default=0 Multiply with transposed of first input (A).
- **transpose.b**: boolean, optional, default=0 Multiply with transposed of second input (B).
- **alpha**: double, optional, default=1 Scalar factor multiplied with $A*B$.
- **axis**: int, optional, default=’-2’ Axis corresponding to the matrix row indices.
- **name**: string, optional Name of the resulting symbol.

Details
$out$ = $\alpha \times out$ $\times \text{op}(A) \times \text{op}(B)$
Here $\alpha$ is a scalar parameter and $\text{op}()$ is either the identity or the matrix transposition (depending on $\text{transpose}_a$, $\text{transpose}_b$).
If $n>2$, $\text{gemm}$ is performed separately for a batch of matrices. The column indices of the matrices are given by the last dimensions of the tensors, the row indices by the axis specified with the $\text{axis}$ parameter. By default, the trailing two dimensions will be used for matrix encoding.
For a non-default axis parameter, the operation performed is equivalent to a series of swapaxes/gemm/swapaxes calls. For example let \( A^*, B^* \) be 5 dimensional tensors. Then gemm\((A^*, B^*, \text{axis}=1)\) is equivalent to the following without the overhead of the additional swapaxis operations::

\[
A1 = \text{swapaxes}(A, \text{dim1}=1, \text{dim2}=3) \quad B1 = \text{swapaxes}(B, \text{dim1}=1, \text{dim2}=3) \quad C = \text{gemm2}(A1, B1) \quad C = \text{swapaxis}(C, \text{dim1}=1, \text{dim2}=3)
\]

When the input data is of type float32 and the environment variables MXNET_CUDA_ALLOW_TENSOR_CORE and MXNET_CUDA_TENSOR_OP_MATH_ALLOW_CONVERSION are set to 1, this operator will try to use pseudo-float16 precision (float32 math with float16 I/O) precision in order to use Tensor Cores on suitable NVIDIA GPUs. This can sometimes give significant speedups.

.. note:: The operator supports float32 and float64 data types only.

Examples::

Single matrix multiply

\[
A = \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \end{bmatrix} \quad B = \begin{bmatrix} 1.0 & 1.0 \end{bmatrix}
\]

\[
gemm2(A, B, \text{transpose_b}=\text{True}, \alpha=2.0) = \begin{bmatrix} 4.0, 4.0, 4.0 \end{bmatrix}
\]

Batch matrix multiply

\[
A = \begin{bmatrix} 1.0 & 1.0 \\ 0.1 & 0.1 \end{bmatrix} \quad B = \begin{bmatrix} 1.0 & 1.0 \end{bmatrix}
\]

\[
gemm2(A, B, \text{transpose_b}=\text{True}, \alpha=2.0) = \begin{bmatrix} 4.0 \end{bmatrix}, \begin{bmatrix} 0.04 \end{bmatrix}
\]

Defined in src/operator/tensor/la_op.cc:L163

**Value**

```
out
```

The result mx.symbol

---

mx.symbol.linalg_inverse

\[\text{linalg\_inverse}: \text{Compute the inverse of a matrix}. \text{ Input is a tensor } *A* of dimension *n >= 2*.\]

**Description**

If \( *n=2* \), \( *A* \) is a square matrix. We compute:

**Usage**

```
mx.symbol.linalg_inverse(...) 
```

**Arguments**

\[A\] NDArray-or-Symbol Tensor of square matrix

\[name\] string, optional Name of the resulting symbol.
mx.symbol.linalg_makediag

Details

\[ \text{out} = A^{-1} \]

If \( n \geq 2 \), \( \text{inverse} \) is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

Examples::

Single matrix inverse \( A = \begin{bmatrix} 1., 4. \\ 2., 3. \end{bmatrix} \) \( \text{inverse}(A) = \begin{bmatrix} -0.6, 0.8 \\ 0.4, -0.2 \end{bmatrix} \)

Batch matrix inverse \( A = \begin{bmatrix} \begin{bmatrix} 1., 4. \\ 2., 3. \end{bmatrix}, \begin{bmatrix} 1., 3. \\ 2., 4. \end{bmatrix} \end{bmatrix} \) \( \text{inverse}(A) = \begin{bmatrix} \begin{bmatrix} -0.6, 0.8 \\ 0.4, -0.2 \end{bmatrix}, \begin{bmatrix} -2., 1.5 \\ 1., -0.5 \end{bmatrix} \end{bmatrix} \)

Defined in src/operator/tensor/la_op.cc:L920

Value

out The result mx.symbol

mx.symbol.linalg_makediag

linalg_makediag: Constructs a square matrix with the input as diagonal. Input is a tensor \( A \) of dimension \( n \geq 1 \).

Description

If \( n=1 \), then \( A \) represents the diagonal entries of a single square matrix. This matrix will be returned as a 2-dimensional tensor. If \( n>1 \), then \( A \) represents a batch of diagonals of square matrices. The batch of diagonal matrices will be returned as an \( n+1 \)-dimensional tensor.

Usage

mx.symbol.linalg_makediag(...)
mx.symbol.linalg_maketrian

*linalg_maketrian*: Constructs a square matrix with the input representing a specific triangular sub-matrix. This is basically the inverse of *linalg.extracttrian*. Input is a tensor *A* of dimension *n >= 1*.

**Description**

If *n=1*, then *A* represents the entries of a triangular matrix which is lower triangular if *offset=0* or *offset=0*, *lower=true*. The resulting matrix is derived by first constructing the square matrix with the entries outside the triangle set to zero and then adding *offset*-times an additional diagonal with zero entries to the square matrix.

**Usage**

mx.symbol.linalg_maketrian(...)  

**Arguments**

- **A**: NDArray-or-Symbol Tensor of triangular matrices stored as vectors  
- **offset**: int, optional, default=`0` Offset of the diagonal versus the main diagonal. 0 corresponds to the main diagonal, a negative/positive value to diagonals below/above the main diagonal.  
- **lower**: boolean, optional, default=`1` Refer to the lower triangular matrix if lower=1, refer to the upper otherwise. Only relevant when offset=0  
- **name**: string, optional Name of the resulting symbol.

---

**Details**

.. note:: The operator supports float32 and float64 data types only.

**Examples::**

Single diagonal matrix construction  
\[ A = [1.0, 2.0] \]

\[
\text{makediag}(A) = [[1.0, 0.0], [0.0, 2.0]]
\]

\[
\text{makediag}(A, 1) = [[0.0, 1.0, 0.0], [0.0, 0.0, 2.0], [0.0, 0.0, 0.0]]
\]

Batch diagonal matrix construction  
\[ A = [[1.0, 2.0], [3.0, 4.0]] \]

\[
\text{makediag}(A) = [[[1.0, 0.0], [0.0, 2.0]], [[3.0, 0.0], [0.0, 4.0]]]
\]

Defined in src/operator/tensor/la_op.cc:L547
Details

If \( n > 1 \), then \( A \) represents a batch of triangular sub-matrices. The batch of corresponding square matrices is returned as an \( n+1 \)-dimensional tensor.

.. note:: The operator supports float32 and float64 data types only.

Examples:

Single matrix construction

\[
A = \begin{bmatrix} 1.0, 2.0, 3.0 \end{bmatrix}
\]

\[
maketrian(A) = \begin{bmatrix} [1.0, 0.0], [2.0, 3.0] \end{bmatrix}
\]

\[
maketrian(A, \text{lower}=false) = \begin{bmatrix} [1.0, 2.0], [0.0, 3.0] \end{bmatrix}
\]

\[
maketrian(A, \text{offset}=1) = \begin{bmatrix} [0.0, 1.0, 2.0], [0.0, 0.0, 3.0], [0.0, 0.0, 0.0] \end{bmatrix}
\]

\[
maketrian(A, \text{offset}=-1) = \begin{bmatrix} [0.0, 0.0, 0.0], [1.0, 0.0, 0.0], [2.0, 3.0, 0.0] \end{bmatrix}
\]

Batch matrix construction

\[
A = \begin{bmatrix} [1.0, 2.0, 3.0], [4.0, 5.0, 6.0] \end{bmatrix}
\]

\[
maketrian(A) = \begin{bmatrix} \begin{bmatrix} [1.0, 0.0], [2.0, 3.0] \end{bmatrix}, \begin{bmatrix} [4.0, 0.0], [5.0, 6.0] \end{bmatrix} \end{bmatrix}
\]

\[
maketrian(A, \text{offset}=1) = \begin{bmatrix} \begin{bmatrix} [0.0, 1.0, 2.0], [0.0, 0.0, 3.0], [0.0, 0.0, 0.0] \end{bmatrix}, \begin{bmatrix} [0.0, 4.0, 5.0], [0.0, 0.0, 6.0], [0.0, 0.0, 0.0] \end{bmatrix} \end{bmatrix}
\]

Defined in src/operator/tensor/la_op.cc:L673

Value

out The result mx.symbol

mx.symbol.linalg_potrf

\textit{linalg\_potrf}: Performs Cholesky factorization of a symmetric positive-definite matrix. Input is a tensor \( A \) of dimension \( n \geq 2 \).

Description

If \( n=2 \), the Cholesky factor \( B \) of the symmetric, positive definite matrix \( A \) is computed. \( B \) is triangular (entries of upper or lower triangle are all zero), has positive diagonal entries, and:

Usage

\[
\text{mx.symbol.linalg\_potrf}(\ldots)
\]

Arguments

- \( A \) NDArray-or-Symbol Tensor of input matrices to be decomposed
- name string, optional Name of the resulting symbol.
mx.symbol.linalg_potri

linalg_potri: Performs matrix inversion from a Cholesky factorization.
Input is a tensor \( \text{\texttt{A}} \) of dimension \( \text{n} \geq 2 \).

Description

If \( \text{n=2} \), \( \text{\texttt{A}} \) is a triangular matrix (entries of upper or lower triangle are all zero) with positive diagonal. We compute:

Usage

mx.symbol.linalg_potri(...) 

Arguments

- **A**: NDArray-or-Symbol Tensor of lower triangular matrices
- **name**: string, optional Name of the resulting symbol.

Details

\( \text{\texttt{out}} \) = \( \text{\texttt{A}} \)^{\text{T}}\text\(^{-1}\) if \( \text{\texttt{lower}} = \text{\texttt{true}} \) 
\( \text{\texttt{out}} \) = \( \text{\texttt{A}} \text\(^{-1}\)\text\(^{T}\) if \( \text{\texttt{lower}} = \text{\texttt{false}} \)

In other words, if \( \text{\texttt{A}} \) is the Cholesky factor of a symmetric positive definite matrix \( \text{\texttt{B}} \) (obtained by \( \text{\texttt{potrf}} \)), then

\( \text{\texttt{out}} \) = \( \text{\texttt{B}} \text\(^{-1}\)\text\(^{T}\)

If \( \text{n>2} \), \( \text{\texttt{potri}} \) is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.
mx.symbol.linalg_slogdet

.. note:: Use this operator only if you are certain you need the inverse of \(B\), and cannot use the Cholesky factor \(A\) (*potrf*), together with backsubstitution (*trsm*). The latter is numerically much safer, and also cheaper.

Examples::

Single matrix inverse \(A = \begin{bmatrix} 2.0, 0 \end{bmatrix}, \begin{bmatrix} 0.5, 2.0 \end{bmatrix}\)

\(\text{potri}(A) = \begin{bmatrix} 0.26563, -0.0625 \end{bmatrix}, \begin{bmatrix} -0.0625, 0.25 \end{bmatrix}\)

Batch matrix inverse \(A = \begin{bmatrix} \begin{bmatrix} 2.0, 0 \end{bmatrix}, \begin{bmatrix} 0.5, 2.0 \end{bmatrix}\, \begin{bmatrix} 4.0, 0 \end{bmatrix}, \begin{bmatrix} 1.0, 4.0 \end{bmatrix}\end{bmatrix}\)

\(\text{potri}(A) = \begin{bmatrix} \begin{bmatrix} 0.26563, -0.0625 \end{bmatrix}, \begin{bmatrix} -0.0625, 0.25 \end{bmatrix}\end{bmatrix}, \begin{bmatrix} \begin{bmatrix} 0.06641, -0.01562 \end{bmatrix}, \begin{bmatrix} -0.01562, 0.0625 \end{bmatrix}\end{bmatrix}\)

Defined in src/operator/tensor/la_op.cc:L275

Value

out The result mx.symbol

mx.symbol.linalg_slogdet

\textit{linalg_slogdet}: Compute the sign and log of the determinant of a matrix. Input is a tensor \(A\) of dimension \(n \geq 2\).

Description

If \(n=2\), \(A\) is a square matrix. We compute:

Usage

\texttt{mx.symbol.linalg_slogdet(...)}

Arguments

\(A\) NDArray-or-Symbol Tensor of square matrix

name string, optional Name of the resulting symbol.

Details

\(\text{\texttt{\*sign\*}} = \text{\texttt{\*sign\(det(A)\)\*}}, \text{\texttt{\*logabsdet\*}} = \text{\texttt{\*log\(abs(det(A))\)\*}}\)

If \(n>2\), \texttt{slogdet} is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only. .. note:: The gradient is not properly defined on sign, so the gradient of it is not backrounded. .. note:: No gradient is backrounded when \(A\) is non-invertible. Please see the docs of operator \texttt{det} for detail.

Examples::

Single matrix signed log determinant \(A = \begin{bmatrix} 2., 3. \end{bmatrix}, \begin{bmatrix} 1., 4. \end{bmatrix}\)

\(\text{sign, \texttt{logabsdet} = \texttt{slogdet(A)}\} \text{\texttt{sign = \[1\]}\} \text{\texttt{logabsdet = [1.609438]}\}}\)

Batch matrix signed log determinant \(A = \begin{bmatrix} \begin{bmatrix} 2., 3. \end{bmatrix}, \begin{bmatrix} 1., 4. \end{bmatrix}\, \begin{bmatrix} 1., 2. \end{bmatrix}, \begin{bmatrix} 2., 4. \end{bmatrix}\, \begin{bmatrix} 1., 2. \end{bmatrix}, \begin{bmatrix} 4., 3. \end{bmatrix}\end{bmatrix}\)

\(\text{sign, \texttt{logabsdet} = \texttt{slogdet(A)}\} \text{\texttt{sign = \[1., 0., -1\]}\} \text{\texttt{logabsdet = [1.609438, -inf, 1.609438]}\}}\)

Defined in src/operator/tensor/la_op.cc:L1034
mx.symbol.linalg_sumlogdiag

linalg_sumlogdiag: Computes the sum of the logarithms of the diagonal elements of a square matrix. Input is a tensor *A* of dimension *n* >= 2*.

Description

If *n=2*, *A* must be square with positive diagonal entries. We sum the natural logarithms of the diagonal elements, the result has shape (1,).

Usage

mx.symbol.linalg_sumlogdiag(...)

Arguments

- **A**: NDArray-or-Symbol Tensor of square matrices
- **name**: string, optional Name of the resulting symbol.

Details

If *n>2*, *sumlogdiag* is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

Examples::

Single matrix reduction A = [[1.0, 1.0], [1.0, 7.0]] sumlogdiag(A) = [1.9459]

Batch matrix reduction A = [[[1.0, 1.0], [1.0, 7.0]], [[3.0, 0], [0, 17.0]]] sumlogdiag(A) = [1.9459, 3.9318]

Defined in src/operator/tensor/la_op.cc:L445

Value

out The result mx.symbol
Description

If \( n=2 \), the operator performs the BLAS3 function \( \text{syrk} \):

Usage

\[
\text{mx.symbol.linalg_syrk}(...) 
\]

Arguments

- \( A \)  
  NDArray-or-Symbol Tensor of input matrices
- \( \text{transpose} \)  
  boolean, optional, default=0 Use transpose of input matrix.
- \( \text{alpha} \)  
  double, optional, default=1 Scalar factor to be applied to the result.
- \( \text{name} \)  
  string, optional Name of the resulting symbol.

Details

\[
\text{out} = \alpha \cdot A \cdot A^\top \quad \text{if } \text{transpose}=\text{False}, \text{or} \\
\text{out} = \alpha \cdot A^\top \cdot A \quad \text{if } \text{transpose}=\text{True}.
\]

If \( n>2 \), \( \text{syrk} \) is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

Examples::

Single matrix multiply \( A = \begin{bmatrix} 1. & 2. \\ 3. & 4. \end{bmatrix} \)  
\( \text{syrk}(A, \alpha=1., \text{transpose}=\text{False}) = \begin{bmatrix} 14. & 32. \\ 32. & 77. \end{bmatrix} \)  
\( \text{syrk}(A, \alpha=1., \text{transpose}=\text{True}) = \begin{bmatrix} 17. & 22. & 27. \\ 22. & 29. & 36. \\ 27. & 36. & 45. \end{bmatrix} \)

Batch matrix multiply \( A = \begin{bmatrix} \begin{bmatrix} 1. & 1. \\ 0.1 & 0.1 \end{bmatrix} \end{bmatrix} \)  
\( \text{syrk}(A, \alpha=2., \text{transpose}=\text{False}) = \begin{bmatrix} 4. \end{bmatrix} \)  
\( \text{syrk}(A, \alpha=2., \text{transpose}=\text{True}) = \begin{bmatrix} 4. \end{bmatrix} \)

Defined in src/operator/tensor/la_op.cc:L730

Value

out The result mx.symbol
mx.symbol.linalg_trmm

**linalg_trmm**: Performs multiplication with a lower triangular matrix. Input are tensors *A*, *B*, each of dimension *n* >= 2 and having the same shape on the leading *n-2* dimensions.

**Description**

If *n=2*, *A* must be triangular. The operator performs the BLAS3 function *trmm*:

**Usage**

```
mx.symbol.linalg_trmm(...)
```

**Arguments**

- **A**: NDArray-or-Symbol Tensor of lower triangular matrices
- **B**: NDArray-or-Symbol Tensor of matrices
- **transpose**: boolean, optional, default=0 Use transposed of the triangular matrix
- **rightside**: boolean, optional, default=0 Multiply triangular matrix from the right to non-triangular one.
- **lower**: boolean, optional, default=1 True if the triangular matrix is lower triangular, false if it is upper triangular.
- **alpha**: double, optional, default=1 Scalar factor to be applied to the result.
- **name**: string, optional Name of the resulting symbol.

**Details**

\[ \text{out} = \alpha \times \text{op}(A) \times B \]

if *rightside=False*, or

\[ \text{out} = \alpha \times B \times \text{op}(A) \]

if *rightside=True*. Here, \(\alpha\) is a scalar parameter, and \(\text{op()}\) is either the identity or the matrix transposition (depending on *transpose*).

If *n>2*, *trmm* is performed separately on the trailing two dimensions for all inputs (batch mode).

.. note:: The operator supports float32 and float64 data types only.

**Examples**:

Single triangular matrix multiply

\[ A = [[1.0, 0], [1.0, 1.0]] \]

\[ B = [[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]] \]

\[ \text{trmm}(A, B, \alpha=2.0) = [[2.0, 2.0, 2.0], [4.0, 4.0, 4.0]] \]

Batch triangular matrix multiply

\[ A = [[[1.0, 0], [1.0, 1.0]], [[1.0, 0], [1.0, 1.0]]] \]

\[ B = [[[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]], [[0.5, 0.5, 0.5], [0.5, 0.5, 0.5]]] \]

\[ \text{trmm}(A, B, \alpha=2.0) = [[[2.0, 2.0, 2.0], [4.0, 4.0, 4.0]], [[1.0, 1.0, 1.0], [2.0, 2.0, 2.0]]] \]

Defined in src/operator/tensor/la_op.cc:L333
Value

out The result mx.symbol

mx.symbol.linalg_trsm  linalg_trsm:Solves matrix equation involving a lower triangular matrix. Input are tensors *A*, *B*, each of dimension *n >= 2* and having the same shape on the leading *n-2* dimensions.

Description

If *n=2*, *A* must be triangular. The operator performs the BLAS3 function *trsm*, solving for *out* in:

Usage

mx.symbol.linalg_trsm(…)

Arguments

A                 NDArray-or-Symbol Tensor of lower triangular matrices
B                 NDArray-or-Symbol Tensor of matrices
transpose         boolean, optional, default=0 Use transposed of the triangular matrix
rightside         boolean, optional, default=0 Multiply triangular matrix from the right to non-triangular one.
lower             boolean, optional, default=1 True if the triangular matrix is lower triangular, false if it is upper triangular.
alpha             double, optional, default=1 Scalar factor to be applied to the result.
name              string, optional Name of the resulting symbol.

Details

*op*(A)* out* = *alpha* \* *B*
if *rightside=False*, or
*out* \* *op*(A) = *alpha* \* *B*
if *rightside=True*. Here, *alpha* is a scalar parameter, and *op()* is either the identity or the matrix transposition (depending on *transpose*).

If *n>2*, *trsm* is performed separately on the trailing two dimensions for all inputs (batch mode).

Examples:

Single matrix solve A = [[1.0, 0], [1.0, 1.0]] B = [[2.0, 2.0, 2.0], [4.0, 4.0, 4.0]] trsm(A, B, alpha=0.5) = [[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]]
Batch matrix solve A = [[[1.0, 0], [1.0, 1.0]], [[1.0, 0], [1.0, 1.0]]] B = [[[2.0, 2.0, 2.0], [4.0, 4.0, 4.0]], [[4.0, 4.0, 4.0], [8.0, 8.0, 8.0]]] trsm(A, B, alpha=0.5) = [[[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]], [[2.0, 2.0, 2.0], [2.0, 2.0, 2.0]]]

Defined in src/operator/tensor/la_op.cc:L396
Value
out The result mx.symbol

mx.symbol.LinearRegressionOutput

LinearRegressionOutput: Computes and optimizes for squared loss during backward propagation. Just outputs “data” during forward propagation.

Description
If \( \hat{y}_i \) is the predicted value of the \( i \)-th sample, and \( y_i \) is the corresponding target value, then the squared loss estimated over \( n \) samples is defined as

\[
\text{SquaredLoss} (\textbf{Y}, \hat{\textbf{Y}}) = \frac{1}{n} \sum_i=0^{n-1} \| \textbf{y}_i - \hat{\textbf{y}}_i \|_2
\]

.. note:: Use the LinearRegressionOutput as the final output layer of a net.

The storage type of “label” can be “default” or “csr”

- LinearRegressionOutput(default, default) = default - LinearRegressionOutput(default, csr) = default

By default, gradients of this loss function are scaled by factor ‘1/m’, where \( m \) is the number of regression outputs of a training example. The parameter ‘grad_scale’ can be used to change this scale to ‘grad_scale/m’.

Defined in src/operator/regression_output.cc:L92

Value
out The result mx.symbol
### mx.symbol.load

#### Load an mx.symbol object

**Description**

Load an mx.symbol object

**Usage**

`mx.symbol.load(file.name)`

**Arguments**

- `filename`: the filename (including the path)

**Examples**

```python
data = mx.symbol.Variable('data')
mx.symbol.save(data, 'temp.symbol')
data2 = mx.symbol.load('temp.symbol')
```

---

### mx.symbol.load.json

#### Load an mx.symbol object from a json string

**Description**

Load an mx.symbol object from a json string

**Arguments**

- `str`: the json str represent a mx.symbol
mx.symbol.log

log: Returns element-wise Natural logarithmic value of the input.

Description

The natural logarithm is logarithm in base $e$, so that $\log(\exp(x)) = x$.

Usage

mx.symbol.log(...)

Arguments

data: NDArray-or-Symbol The input array.
name: string, optional Name of the resulting symbol.

Details

The storage type of “log” output is always dense.
Defined in src/operator/tensor/elemwise_unary_op_logexp.cc:L77

Value

out The result mx.symbol

mx.symbol.log10

log10: Returns element-wise Base-10 logarithmic value of the input.

Description

$10^{\log_{10}(x)} = x$.

Usage

mx.symbol.log10(...)

Arguments

data: NDArray-or-Symbol The input array.
name: string, optional Name of the resulting symbol.

Details

The storage type of “log10” output is always dense.
Defined in src/operator/tensor/elemwise_unary_op_logexp.cc:L94
mx.symbol.log1p

Value

out The result mx.symbol

mx.symbol.log1p

log1p:Returns element-wise \( \log(1 + x) \) value of the input.

Description

This function is more accurate than \( \log(1 + x) \) for small \( x \) so that \( 1 + x \approx 1 \)

Usage

mx.symbol.log1p(...)

Arguments

data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.

details

The storage type of \( \log1p \) output depends upon the input storage type:
- \( \log1p(\text{default}) = \text{default} \)
- \( \log1p(\text{row_sparse}) = \text{row_sparse} \)
- \( \log1p(\text{csr}) = \text{csr} \)

Defined in src/operator/tensor/elemwise_unary_op_logexp.cc:L199

mx.symbol.log2

log2:Returns element-wise Base-2 logarithmic value of the input.

Description

\( 2^{\log2(x)} = x \)

Usage

mx.symbol.log2(...)

Arguments

data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.
**Details**

The storage type of “log2“ output is always dense

Defined in src/operator/tensor/elemwise_unary_op_logexp.cc:L106

**Value**

out The result mx.symbol

---

**mx.symbol.logical_not**

*logical_not:* Returns the result of logical NOT (!) function

**Description**

Example: logical_not([-2., 0., 1.]) = [0., 1., 0.]

**Usage**

mx.symbol.logical_not(...)

**Arguments**

- **data**: NDArray-or-Symbol The input array.
- **name**: string, optional Name of the resulting symbol.

**Value**

out The result mx.symbol

---

**mx.symbol.LogisticRegressionOutput**

*LogisticRegressionOutput:* Applies a logistic function to the input.

**Description**

The logistic function, also known as the sigmoid function, is computed as \( \frac{1}{1 + \exp(-x)} \).

**Usage**

mx.symbol.LogisticRegressionOutput(...)

---
mx.symbol.log_softmax

Arguments

- **data**: NDArray-or-Symbol Input data to the function.
- **label**: NDArray-or-Symbol Input label to the function.
- **grad.scale**: float, optional, default=1 Scale the gradient by a float factor
- **name**: string, optional Name of the resulting symbol.

Details

Commonly, the sigmoid is used to squash the real-valued output of a linear model \( w^T x + b \) into the \([0,1]\) range so that it can be interpreted as a probability. It is suitable for binary classification or probability prediction tasks.

.. note:: Use the LogisticRegressionOutput as the final output layer of a net.

The storage type of “label” can be “default” or “csr”

- LogisticRegressionOutput(default, default) = default - LogisticRegressionOutput(default, csr) = default

The loss function used is the Binary Cross Entropy Loss:

\[ - (y \log(p) + (1 - y) \log(1 - p)) \]

Where ‘y’ is the ground truth probability of positive outcome for a given example, and ‘p’ the probability predicted by the model. By default, gradients of this loss function are scaled by factor ‘1/m’, where \( m \) is the number of regression outputs of a training example. The parameter ‘grad_scale’ can be used to change this scale to ‘grad_scale/m’.

Defined in src/operator/regression_output.cc:L152

Value

- **out**: The result mx.symbol

mx.symbol.log_softmax

log_softmax Computes the log softmax of the input. This is equivalent to computing softmax followed by log.

Description

Examples::

Usage

mx.symbol.log_softmax(\ldots)
mx.symbol.LRN

Description

The local response normalization layer performs "lateral inhibition" by normalizing over local input regions.

Usage

mx.symbol.LRN(...)

Arguments

data  NDArray-or-Symbol Input data to LRN
alpha float, optional, default=9.99999975e-05 The variance scaling parameter :math:`\alpha` in the LRN expression.
beta float, optional, default=0.75 The power parameter :math:`\beta` in the LRN expression.
knorm float, optional, default=2 The parameter :math:`k` in the LRN expression.
nsize int (non-negative), required normalization window width in elements.
name string, optional Name of the resulting symbol.
Details

If \( a_x,y^i \) is the activity of a neuron computed by applying kernel \( i \) at position \( (x, y) \) and then applying the ReLU nonlinearity, the response-normalized activity \( b_x,y^i \) is given by the expression:

\[
   b_x,y^i = \frac{a_x,y^i}{k + \frac{\alpha}{n} \sum_j=\max(0, i-\frac{n}{2})^{\min(N-1, i+\frac{n}{2})} (a_x,y^j)^2} ^\beta
\]

where the sum runs over \( n \) "adjacent" kernel maps at the same spatial position, and \( N \) is the total number of kernels in the layer.

Defined in src/operator/nn/lrn.cc:L158

Value

out The result mx.symbol

mx.symbol.MAERegressionOutput

MAERegressionOutput: Computes mean absolute error of the input.

Description

MAE is a risk metric corresponding to the expected value of the absolute error.

Usage

mx.symbol.MAERegressionOutput(...)

Arguments

data NDArray-or-Symbol Input data to the function.
label NDArray-or-Symbol Input label to the function.
grad.scale float, optional, default=1 Scale the gradient by a float factor
name string, optional Name of the resulting symbol.

Details

If \( \hat{y}_i \) is the predicted value of the \( i \)-th sample, and \( y_i \) is the corresponding target value, then the mean absolute error (MAE) estimated over \( n \) samples is defined as:

\[
   \text{MAE}(\textbf{Y}, \hat{\textbf{Y}}) = \frac{1}{n} \sum_{i=0}^{n-1} \left\| \textbf{y}_i - \hat{\textbf{y}}_i \right\|_1
\]

.. note:: Use the MAERegressionOutput as the final output layer of a net.

The storage type of "label" can be "default" or "csr"

- MAERegressionOutput(default, default) = default - MAERegressionOutput(default, csr) = default

By default, gradients of this loss function are scaled by factor ‘1/m’, where m is the number of regression outputs of a training example. The parameter ‘grad_scale’ can be used to change this scale to ‘grad_scale/m’.

Defined in src/operator/regression_output.cc:L120
mx.symbol.MakeLoss

MakeLoss: Make your own loss function in network construction.

Description
This operator accepts a customized loss function symbol as a terminal loss and the symbol should be an operator with no backward dependency. The output of this function is the gradient of loss with respect to the input data.

Usage
mx.symbol.MakeLoss(...)

Arguments
- **data**: NDArray-or-Symbol Input array.
- **grad.scale**: float, optional, default=1 Gradient scale as a supplement to unary and binary operators
- **valid.thresh**: float, optional, default=0 clip each element in the array to 0 when it is less than "valid_thresh". This is used when "normalization" is set to "'valid'".
- **normalization**: 'batch', 'null', 'valid', optional, default='null' If this is set to null, the output gradient will not be normalized. If this is set to batch, the output gradient will be divided by the batch size. If this is set to valid, the output gradient will be divided by the number of valid input elements.
- **name**: string, optional Name of the resulting symbol.

Details
For example, if you are making a cross entropy loss function. Assume "out" is the predicted output and "label" is the true label, then the cross entropy can be defined as::

cross_entropy = label * log(out) + (1 - label) * log(1 - out) loss = MakeLoss(cross_entropy)

We will need to use "MakeLoss" when we are creating our own loss function or we want to combine multiple loss functions. Also we may want to stop some variables’ gradients from backpropagation. See more detail in “BlockGrad” or “stop_gradient”.

In addition, we can give a scale to the loss by setting “grad_scale“, so that the gradient of the loss will be rescaled in the backpropagation.

.. note:: This operator should be used as a Symbol instead of NDArray.

Defined in src/operator/make_loss.cc:L71

Value
out The result mx.symbol
**mx.symbol.make_loss**  

**Description**  
This operator accepts a customized loss function symbol as a terminal loss and the symbol should be an operator with no backward dependency. The output of this function is the gradient of loss with respect to the input data.

**Usage**  
mx.symbol.make_loss(...)  

**Arguments**  
- data: NDArray-or-Symbol The input array.  
- name: string, optional Name of the resulting symbol.

**Details**  
For example, if you are a making a cross entropy loss function. Assume “out“ is the predicted output and “label“ is the true label, then the cross entropy can be defined as:

cross_entropy = label * log(out) + (1 - label) * log(1 - out)  
loss = make_loss(cross_entropy)

We will need to use “make_loss“ when we are creating our own loss function or we want to combine multiple loss functions. Also we may want to stop some variables’ gradients from backpropagation. See more detail in “BlockGrad“ or “stop_gradient“.

The storage type of “make_loss“ output depends upon the input storage type:
- make_loss(default) = default - make_loss(row_sparse) = row_sparse

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L358

**Value**  
- out The result mx.symbol

---

**mx.symbol.max**  

**Description**  
Computes the max of array elements over given axes.

**Usage**  
mx.symbol.max(...)  

**Description**  
Defined in src/operator/tensor/./broadcast_reduce_op.h:L32

**Usage**  
mx.symbol.max(...)
mx.symbol.max_axis

Arguments

- **data**: NDArray-or-Symbol The input
- **axis**: Shape or None, optional, default=None The axis or axes along which to perform the reduction. The default, ‘axis=()', will compute over all elements into a scalar array with shape '(1,)'. If ‘axis’ is int, a reduction is performed on a particular axis. If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple. If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead. Negative values means indexing from right to left.
- **keepdims**: boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.
- **exclude**: boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.
- **name**: string, optional Name of the resulting symbol.

Value

- **out**: The result mx.symbol

mx.symbol.max_axis  
max_axis:Computes the max of array elements over given axes.

Description

Defined in src/operator/tensor/./broadcast_reduce_op.h:L32

Usage

mx.symbol.max_axis(...)
mx.symbol.mean

keepdims: boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.
exclude: boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.
name: string, optional Name of the resulting symbol.

Value

tout The result mx.symbol

mx.symbol.mean: Computes the mean of array elements over given axes.

Description

Defined in src/operator/tensor/./broadcast_reduce_op.h:L84

Usage

mx.symbol.mean(...)

Arguments

data: NDArray-or-Symbol The input
axis: Shape or None, optional, default=None The axis or axes along which to perform the reduction.
The default, ‘axis=()’, will compute over all elements into a scalar array with shape ‘(1,)’.
If ‘axis’ is int, a reduction is performed on a particular axis.
If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple.
If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead.
Negative values means indexing from right to left.

keepdims: boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.
exclude: boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.
name: string, optional Name of the resulting symbol.

Value

tout The result mx.symbol
**Description**

The mean and variance are calculated by aggregating the contents of data across axes. If x is 1-D and axes = [0] this is just the mean and variance of a vector.

**Usage**

```python
mx.symbol.moments(...)```

**Arguments**

- **data**: NDArray-or-Symbol Input ndarray
- **axes**: Shape or None, optional, default=None Array of ints. Axes along which to compute mean and variance.
- **keepdims**: boolean, optional, default=0 produce moments with the same dimensionality as the input.
- **name**: string, optional Name of the resulting symbol.

**Details**

Example:

```python
x = [[1, 2, 3], [4, 5, 6]] mean, var = moments(data=x, axes=[0]) mean = [2.5, 3.5, 4.5] var = [2.25, 2.25, 2.25] mean, var = moments(data=x, axes=[1]) mean = [2.0, 5.0] var = [0.66666667, 0.66666667] mean, var = moments(data=x, axis=[0, 1]) mean = [3.5] var = [2.9166667]
```

Defined in src/operator/nn/moments.cc:L54

**Value**

- **out**: The result mx.symbol

---

**Description**


**Usage**

```python
mx.symbol.mp_lamb_update_phase1(...)```

**Description**

Mixed Precision version of Phase I of lamb update it performs the following operations and returns g:

**Usage**

```python
mx.symbol.mp_lamb_update_phase1(...)```
Arguments

weight  NDArray-or-Symbol  Weight
grad    NDArray-or-Symbol  Gradient
mean    NDArray-or-Symbol  Moving mean
var     NDArray-or-Symbol  Moving variance
weight32 NDArray-or-Symbol  Weight32
beta1   float, optional, default=0.899999976 The decay rate for the 1st moment estimates.
beta2   float, optional, default=0.999000013 The decay rate for the 2nd moment estimates.
epsilon float, optional, default=9.99999997e-07 A small constant for numerical stability.
t      int, required Index update count.
bias_correction boolean, optional, default=1 Whether to use bias correction.
wd      float, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
name    string, optional Name of the resulting symbol.

Details

.. math:: \begin{align*}
    grad32 = grad(float16) * rescale_grad \text{ if (grad < -clip_gradient) then grad = -clip_gradient if (grad > clip_gradient) then grad = clip_gradient } \\
    mean = beta1 * mean + (1 - beta1) * grad; variance = beta2 * variance + (1 - beta2) * grad^2; \\
    \text{if (bias_correction) then mean_hat = mean / (1 - beta1^t); var_hat = var / (1 - beta2^t); g = mean_hat / (var_hat^{(1/2)} + epsilon) + wd * weight32; else g = mean / (var_data^{(1/2)} + epsilon) + wd * weight32; }\end{align*}

Defined in src/operator/optimizer_op.cc:L1024

Value

out The result mx.symbol
mx.symbol.mp_lamb_update_phase2

**mp_lamb_update_phase2:** Mixed Precision version Phase II of lamb update it performs the following operations and updates grad.

**Description**


**Usage**

```python
mx.symbol.mp_lamb_update_phase2(...)
```

**Arguments**

- **weight**
  - NDArray-or-Symbol Weight
- **g**
  - NDArray-or-Symbol Output of mp_lamb_update_phase1
- **r1**
  - NDArray-or-Symbol r1
- **r2**
  - NDArray-or-Symbol r2
- **weight32**
  - NDArray-or-Symbol Weight32
- **lr**
  - float, required Learning rate
- **lower_bound**
  - float, optional, default=-1 Lower limit of norm of weight. If lower_bound <= 0, Lower limit is not set
- **upper_bound**
  - float, optional, default=-1 Upper limit of norm of weight. If upper_bound <= 0, Upper limit is not set
- **name**
  - string, optional Name of the resulting symbol.

**Details**

.. math:: \begin{gather*}
   \text{if (lower_bound >= 0) then } r1 = \max(r1, \text{lower_bound}) \text{ if (upper_bound >= 0) then } r1 = \max(r1, \text{upper_bound}) \\
   \text{if (r1 == 0 or r2 == 0) then } lr = lr \text{ else } lr = lr * (r1/r2) \text{ weight32} = \text{weight32} - lr * g \text{ weight(float16)}
\end{gather*}

Defined in src/operator/optimizer_op.cc:L1066

**Value**

- **out** The result mx.symbol
mx.symbol.mp_nag_mom_update

mp_nag_mom_update: Update function for multi-precision Nesterov Accelerated Gradient (NAG) optimizer.

Description

Defined in src/operator/optimizer_op.cc:L736

Usage

mx.symbol.mp_nag_mom_update(...) 

Arguments

weight NDArray-or-Symbol Weight
grad NDArray-or-Symbol Gradient
mom NDArray-or-Symbol Momentum
weight32 NDArray-or-Symbol Weight32
lr float, required Learning rate
momentum float, optional, default=0 The decay rate of momentum estimates at each epoch.
wd float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
name string, optional Name of the resulting symbol.

Value

out The result mx.symbol
### Description

*mp_sgd_mom_update*: Updater function for multi-precision sgd optimizer

### Usage

```python
mx.symbol.mp_sgd_mom_update(...)
```

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>NDArray-or-Symbol Weight</td>
</tr>
<tr>
<td>grad</td>
<td>NDArray-or-Symbol Gradient</td>
</tr>
<tr>
<td>mom</td>
<td>NDArray-or-Symbol Momentum</td>
</tr>
<tr>
<td>weight32</td>
<td>NDArray-or-Symbol Weight32</td>
</tr>
<tr>
<td>lr</td>
<td>float, required Learning rate</td>
</tr>
<tr>
<td>momentum</td>
<td>float, optional, default=0 The decay rate of momentum estimates at each epoch.</td>
</tr>
<tr>
<td>wd</td>
<td>float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.</td>
</tr>
<tr>
<td>rescale.grad</td>
<td>float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.</td>
</tr>
<tr>
<td>clip.gradient</td>
<td>float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient &lt;= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).</td>
</tr>
<tr>
<td>lazy.update</td>
<td>boolean, optional, default=1 If true, lazy updates are applied if gradient's stype is row_sparse and both weight and momentum have the same stype</td>
</tr>
<tr>
<td>name</td>
<td>string, optional Name of the resulting symbol.</td>
</tr>
</tbody>
</table>

### Value

```python
out The result mx.symbol
```
mx.symbol.mp_sgd_update

mp_sgd_update: Updater function for multi-precision sgd optimizer

Description

mp_sgd_update: Updater function for multi-precision sgd optimizer

Usage

mx.symbol.mp_sgd_update(...)

Arguments

weight   NDArray-or-Symbol Weight
grad     NDArray-or-Symbol gradient
weight32 NDArray-or-Symbol Weight32
lr       float, required Learning rate
wd       float, optional, default=0 Weight decay augments the objective function with a
          regularization term that penalizes large weights. The penalty scales with the
          square of the magnitude of each weight.
rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient]
             If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad,
             clip_gradient),-clip_gradient).
lazy.update boolean, optional, default=1 If true, lazy updates are applied if gradient’s stype
             is row_sparse.
name     string, optional Name of the resulting symbol.

Value

out The result mx.symbol

mx.symbol.multi_all_finite

multi_all_finite: Check if all the float numbers in all the arrays are
finite (used for AMP)

Description

Defined in src/operator/contrib/all_finite.cc:L133
mx.symbol.multi_lars

multi_lars: Compute the LARS coefficients of multiple weights and
grads from their sums of square

Description
Defined in src/operator/contrib/multi_lars.cc:L37

Usage
mx.symbol.multi_lars(...)

Arguments
- **lrs**: NDArray-or-Symbol Learning rates to scale by LARS coefficient
- **weights.sum.sq**: NDArray-or-Symbol sum of square of weights arrays
- **grads.sum.sq**: NDArray-or-Symbol sum of square of gradients arrays
- **wds**: NDArray-or-Symbol weight decays
- **eta**: float, required LARS eta
- **eps**: float, required LARS eps
- **rescale.grad**: float, optional, default=1 Gradient rescaling factor
- **name**: string, optional Name of the resulting symbol.

Value
out The result mx.symbol

Description

Momentum update has better convergence rates on neural networks. Mathematically it looks like:

\[ \begin{aligned}
\text{v}_1 &= \alpha \times \nabla J(W_0) \\
\text{v}_t &= \gamma \text{v}_{t-1} - \alpha \times \nabla J(W_{t-1}) \\
W_t &= W_{t-1} + \text{v}_t
\end{aligned} \]

It updates the weights using:

\[ \begin{aligned}
\text{v} &= \text{momentum} \times \text{v} - \text{learning_rate} \times \text{gradient weight} \\
\text{weight} &= \text{weight} + \text{v}
\end{aligned} \]

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.

Defined in src/operator/optimizer_op.cc:L463

Value

out The result mx.symbol
mx.symbol.multi_mp_sgd_update


Description

It updates the weights using:

Usage

mx.symbol.multi_mp_sgd_update(...)  

Arguments

data  NDArray-or-Symbol[] Weights
lrs   tuple of <float>, required Learning rates.
wds   tuple of <float>, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
rescale.grad  float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.gradient  float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient]
num.weights  int, optional, default='1’ Number of updated weights.
name      string, optional Name of the resulting symbol.

details

weight = weight - learning_rate * (gradient + wd * weight)

Defined in src/operator/optimizer_op.cc:L408

Value

out The result mx.symbol
**Description**

Momentum update has better convergence rates on neural networks. Mathematically it looks like below:

**Usage**

```python
mx.symbol.multi_sgd_mom_update(...)
```

**Arguments**

- `data` : NDArray-or-Symbol[] Weights, gradients and momentum
- `lrs` : tuple of <float>, required Learning rates.
- `wds` : tuple of <float>, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- `momentum` : float, optional, default=0 The decay rate of momentum estimates at each epoch.
- `rescale.grad` : float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- `clip.gradient` : float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient]
  If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- `num.weights` : int, optional, default=’1’ Number of updated weights.
- `name` : string, optional Name of the resulting symbol.

**Details**

```math
v_1 = \alpha \nabla J(W_0)
v_t = \gamma v_{t-1} - \alpha \nabla J(W_{t-1})W_t = W_{t-1} + v_t
```

It updates the weights using:

```python
v = momentum * v - learning_rate * gradient weight += v
```

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.

Defined in src/operator/optimizer_op.cc:L365

**Value**

```python
out The result mx.symbol
```
**Description**

It updates the weights using:

**Usage**

```python
mx.symbol.multi_sgd_update(...)
```

**Arguments**

- `data` : NDArray-or-Symbol[] Weights
- `lrs` : tuple of float, required Learning rates.
- `wds` : tuple of float, required Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- `rescale.grad` : float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- `clip.gradient` : float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- `num.weights` : int, optional, default='1' Number of updated weights.
- `name` : string, optional Name of the resulting symbol.

**Details**

```python
weight = weight - learning_rate * (gradient + wd * weight)
```

Defined in src/operator/optimizer_op.cc:1320

**Value**

```python
out The result mx.symbol
```
**mx.symbol.multi_sum_sq**

*multi_sum_sq:* Compute the sums of squares of multiple arrays

**Description**

Defined in src/operator/contrib/multi_sum_sq.cc:L36

**Usage**

```
mx.symbol.multi_sum_sq(...)  
```

**Arguments**

- `data` : NDArray-or-Symbol[] Arrays
- `num.arrays` : int, required number of input arrays.
- `name` : string, optional Name of the resulting symbol.

**Value**

```
out The result mx.symbol
```

**mx.symbol.nag_mom_update**

*nag_mom_update:* Update function for Nesterov Accelerated Gradient (NAG) optimizer. It updates the weights using the following formula,

\[
\begin{align*}
    v_t &= \gamma v_{t-1} + \eta \nabla J(W_{t-1} - \gamma v_{t-1}) \\
    W_t &= W_{t-1} - v_t
\end{align*}
\]

**Description**

.. math:: v_t = \gamma v_{t-1} + \eta \nabla J(W_{t-1} - \gamma v_{t-1}) \\
\quad W_t = W_{t-1} - v_t

**Usage**

```
mx.symbol.nag_mom_update(...)  
```

**Arguments**

- `weight` : NDArray-or-Symbol Weight
- `grad` : NDArray-or-Symbol Gradient
- `mom` : NDArray-or-Symbol Momentum
- `lr` : float, required Learning rate
- `momentum` : float, optional, default=0 The decay rate of momentum estimates at each epoch.
**Details**

Where $\eta$ is the learning rate of the optimizer $\gamma$ is the decay rate of the momentum estimate $v_t$ is the update vector at time step $t$ $W_t$ is the weight vector at time step $t$

Defined in src/operator/optimizer_op.cc:L717

**Value**

out The result mx.symbol

---

**Description**

nanprod: Computes the product of array elements over given axes treating Not a Numbers (“NaN”) as one.

**Usage**

mx.symbol.nanprod(...)

**Arguments**

data NDArray-or-Symbol The input
axis Shape or None, optional, default=None The axis or axes along which to perform the reduction.

The default, `axis=()`, will compute over all elements into a scalar array with shape `(1,)`.

If `axis` is int, a reduction is performed on a particular axis.

If `axis` is a tuple of ints, a reduction is performed on all the axes specified in the tuple.

If `exclude` is true, reduction will be performed on the axes that are NOT in axis instead.

Negative values means indexing from right to left.
`mx.symbol.nansum` 417

```
keepdims  boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in
the result as dimension with size one.
exclude  boolean, optional, default=0 Whether to perform reduction on axis that are NOT
in axis instead.
name  string, optional Name of the resulting symbol.
```

Details

Defined in src/operator/tensor/broadcast_reduce_prod_value.cc:L47

Value

out The result mx.symbol

```
xm.symbol.nansum  nansum:Computes the sum of array elements over given axes treating
Not a Numbers (“NaN”) as zero.
```

Description

nansum:Computes the sum of array elements over given axes treating Not a Numbers (“NaN”) as zero.

Usage

mx.symbol.nansum(...)

Arguments

data  NDArray-or-Symbol The input
axis  Shape or None, optional, default=0 The axis or axes along which to perform
the reduction.
The default, ‘axis=()’, will compute over all elements into a scalar array with
shape ’(1,)’.
If ‘axis’ is int, a reduction is performed on a particular axis.
If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in
the tuple.
If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis
instead.
Negative values means indexing from right to left.
keepdims  boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in
the result as dimension with size one.
exclude  boolean, optional, default=0 Whether to perform reduction on axis that are NOT
in axis instead.
name  string, optional Name of the resulting symbol.
mx.symbol.norm

Details
Defined in src/operator/tensor/broadcast_reduce_sum_value.cc:L102

Value
out The result mx.symbol

mx.symbol.negative negative:Numerical negative of the argument, element-wise.

Description
The storage type of “negative” output depends upon the input storage type:

Usage
mx.symbol.negative(…)

Arguments
data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.

Details
- negative(default) = default - negative(row_sparse) = row_sparse - negative(csr) = csr

Value
out The result mx.symbol

mx.symbol.norm norm:Computes the norm on an NDArray.

Description
This operator computes the norm on an NDArray with the specified axis, depending on the value of the ord parameter. By default, it computes the L2 norm on the entire array. Currently only ord=2 supports sparse ndarrays.

Usage
mx.symbol.norm(…)

**Arguments**

- **data**: NDArray-or-Symbol The input
- **ord**: int, optional, default='2' Order of the norm. Currently ord=1 and ord=2 is supported.
- **axis**: Shape or None, optional, default=None The axis or axes along which to perform the reduction. The default, ‘axis=()’, will compute over all elements into a scalar array with shape '(1,)'. If ‘axis’ is int, a reduction is performed on a particular axis. If ‘axis’ is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed.
- **out.dtype**: None, 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', optional, default='None' The data type of the output.
- **keepdims**: boolean, optional, default=0 If this is set to ‘True’, the reduced axis is left in the result as dimension with size one.
- **name**: string, optional Name of the resulting symbol.

**Details**

Examples:

```python
x = [[[1, 2], [3, 4]], [[2, 2], [5, 6]]
norm(x, ord=2, axis=1) = [3.1622777 4.472136 5.3851647 6.3245554]
norm(x, ord=1, axis=1) = [4., 6., 7., 8.]
srp = x.cast_storage('row_sparse')
norm(srp) = [5.47722578]
csr = x.cast_storage('csr')
norm(csr) = [5.47722578]
```

Defined in src/operator/tensor/broadcast_reduce_norm_value.cc:L89

**Value**

- **out**: The result mx.symbol

---

**mx.symbol.normal**

`normal`: Draw random samples from a normal (Gaussian) distribution.

**Description**

.. note:: The existing alias “normal” is deprecated.

**Usage**

```python
mx.symbol.normal(...)```
mx.symbol.ones_like

Arguments

- loc: float, optional, default=0 Mean of the distribution.
- scale: float, optional, default=1 Standard deviation of the distribution.
- shape: Shape(tuple), optional, default=None Shape of the output.
- ctx: string, optional, default="" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- dtype: 'None', 'float16', 'float32', 'float64', optional, default='None' Dtype of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- name: string, optional Name of the resulting symbol.

Details

Samples are distributed according to a normal distribution parametrized by *loc* (mean) and *scale* (standard deviation).

Example::

    normal(loc=0, scale=1, shape=(2,2)) = [[ 1.89171135, -1.16881478], [-1.23474145, 1.55807114]]

Defined in src/operator/random/sample_op.cc:L113

Value

out The result mx.symbol

mx.symbol.ones_like

ones_like:Return an array of ones with the same shape and type as the input array.

Description

Examples::

Usage

mx.symbol.ones_like(...)

Arguments

data: NDArray-or-Symbol The input

name: string, optional Name of the resulting symbol.

Details

x = [[ 0., 0., 0.], [ 0., 0., 0.]]
ones_like(x) = [[ 1., 1., 1.], [ 1., 1., 1.]]
Value

out The result mx.symbol

mx.symbol.one_hot

one_hot: Returns a one-hot array.

Description

The locations represented by ‘indices’ take value ‘on_value’, while all other locations take value ‘off_value’.

Usage

mx.symbol.one_hot(...)

Arguments

indices NDArray-or-Symbol array of locations where to set on_value

depth int, required Depth of the one hot dimension.

on.value double, optional, default=1 The value assigned to the locations represented by indices.

off.value double, optional, default=0 The value assigned to the locations not represented by indices.

dtype 'bfloat16', 'float16', 'float32', 'float64', 'int32', 'int64', 'int8', 'uint8', optional, default='float32' DType of the output

name string, optional Name of the resulting symbol.

Details

‘one_hot’ operation with ‘indices’ of shape “(i0, i1)” and ‘depth’ of “d” would result in an output array of shape “(i0, i1, d)” with:

output[i,j,:] = off_value output[i,j,indices[i,j]] = on_value

Examples::

one_hot([1,0,2,0], 3) = [[ 0. 1. 0. ] [ 1. 0. 0. ] [ 0. 0. 1. ] [ 1. 0. 0. ]]
one_hot([1,0,2,0], 3, on_value=8, off_value=1, dtype='int32') = [[1 8 1] [8 1 1] [1 1 8] [8 1 1]]

Defined in src/operator/tensor/indexing_op.cc:L888

Value

out The result mx.symbol
mx.symbol.Pad

Pad: Pads an input array with a constant or edge values of the array.

Description

.. note:: ‘Pad’ is deprecated. Use ‘pad’ instead.

Usage

mx.symbol.Pad(...)

Arguments

data
NDArray-or-Symbol An n-dimensional input array.

mode
'constant', 'edge', 'reflect', required Padding type to use. "constant" pads with
'constant_value' "edge" pads using the edge values of the input array "reflect"
pads by reflecting values with respect to the edges.

pad.width
Shape(tuple), required Widths of the padding regions applied to the edges of
each axis. It is a tuple of integer padding widths for each axis of the format “(be-
fore_1, after_1, ... , before_N, after_N)”. It should be of length “2*N” where
“N” is the number of dimensions of the array. This is equivalent to pad_width in
numpy.pad, but flattened.

central.value
double, optional, default=0 The value used for padding when ‘mode’ is "con-
stant".

name
string, optional Name of the resulting symbol.

Details

.. note:: Current implementation only supports 4D and 5D input arrays with padding applied only
on axes 1, 2 and 3. Expects axes 4 and 5 in ‘pad_width’ to be zero.

This operation pads an input array with either a ‘constant_value’ or edge values along each axis of
the input array. The amount of padding is specified by ‘pad_width’.

‘pad_width’ is a tuple of integer padding widths for each axis of the format “(before_1, after_1, ...
... , before_N, after_N)”. The ‘pad_width’ should be of length “2*N” where “N” is the number of
dimensions of the array.

For dimension “N” of the input array, “before_N” and “after_N” indicates how many values to
add before and after the elements of the array along dimension “N“. The widths of the higher two
dimensions “before_1”, “after_1”, “before_2”, “after_2” must be 0.

Example::
x = [[[ 1.  2.  3. ] [ 4.  5.  6. ]
  [ 7.  8.  9. ] [ 10. 11. 12. ]]
  [[[ 11.  12.  13. ] [ 14.  15.  16. ]
  [[ 17.  18.  19. ] [ 20.  21.  22. ]]]]
**mx.symbol.pad**

```
pad(x, mode="edge", pad_width=(0,0,0,1,1,1,1)) =
[[[ 1. 1. 2. 3. 3.][ 1. 1. 2. 3. 3.][ 4. 4. 5. 6. 6.][ 4. 4. 5. 6. 6.]]
[[ 7. 7. 8. 9. 9.][ 7. 7. 8. 9. 9.][10. 10. 11. 12. 12.][10. 10. 11. 12. 12.]]
```

```
pad(x, mode="constant", constant_value=0, pad_width=(0,0,0,1,1,1,1)) =
[[[ 0. 0. 0. 0. 0.][ 0. 1. 2. 3. 0.][ 0. 4. 5. 6. 0.][ 0. 0. 0. 0. 0.]]
[[ 0. 0. 0. 0. 0.][ 0. 7. 8. 9. 0.][ 0. 10. 11. 12. 0.][ 0. 0. 0. 0. 0.]]
[[ 0. 0. 0. 0. 0.][ 0. 11. 12. 13. 0.][ 0. 14. 15. 16. 0.][ 0. 0. 0. 0. 0.]]
[[ 0. 0. 0. 0. 0.][ 0. 17. 18. 19. 0.][ 0. 20. 21. 22. 0.][ 0. 0. 0. 0. 0.]]
```

Defined in src/operator/pad.cc:L766

Value

```
out The result mx.symbol
```
Details

.. note:: Current implementation only supports 4D and 5D input arrays with padding applied only on axes 1, 2 and 3. Expects axes 4 and 5 in 'pad_width' to be zero.

This operation pads an input array with either a 'constant_value' or edge values along each axis of the input array. The amount of padding is specified by 'pad_width'.

'pad_width' is a tuple of integer padding widths for each axis of the format ``(before_1, after_1, ..., before_N, after_N)``. The 'pad_width' should be of length ``2*N`` where ``N`` is the number of dimensions of the array.

For dimension ``N`` of the input array, ``before_N`` and ``after_N`` indicates how many values to add before and after the elements of the array along dimension ``N``. The widths of the higher two dimensions ``before_1``, ``after_1``, ``before_2``, ``after_2`` must be 0.

Example:

```python
x = [[[  1.  2.  3.]
     [  4.  5.  6.]]
    [[  7.  8.  9.]
     [ 10. 11. 12.]]
    [[ 11. 12. 13.]
     [ 14. 15. 16.]]
    [[ 17. 18. 19.]
     [ 20. 21. 22.]]]
pad(x, mode="edge", pad_width=(0,0,0,0,1,1,1,1)) =
[[[[  1.  2.  3.  3.]
    [  4.  5.  6.  6.]]
   [[  7.  8.  9.  9.]
    [ 10. 11. 12. 12.]]
   [[[ 11. 12. 13. 13.]
     [ 14. 15. 16. 16.]]
    [[ 17. 17. 18. 18.]]
    [ 20. 20. 21. 22.]]]
pad(x, mode="constant", constant_value=0, pad_width=(0,0,0,0,1,1,1,1)) =
[[[[  0.  0.  0.  0.]
    [  0.  1.  2.  3.]
    [  0.  4.  5.  6.]]
   [[  0.  0.  0.  0.]
    [  0.  7.  8.  9.]
    [ 10. 11. 12. 0.]]
   [[[  0.  0.  0.  0.]
     [  0.  11. 12. 13.]
     [  0.  14. 15. 16.]]
    [[  0.  0.  0.  0.]
     [  0.  17. 18. 19.]
     [  0.  20. 21. 22.]]]]
Defined in src/operator/pad.cc:L766
```

Value

out The result mx.symbol

mx.symbol.pick

```
pick: Picks elements from an input array according to the input indices along the given axis.
```

Description

Given an input array of shape ``(d0, d1)`` and indices of shape ``(i0,)``, the result will be an output array of shape ``(i0,)`` with::
Usage

mx.symbol.pick(...)

Arguments

data
NDArray-or-Symbol The input array

index
NDArray-or-Symbol The index array

axis
int or None, optional, default=':1' int or None. The axis to picking the elements. Negative values means indexing from right to left. If is 'None', the elements in the index w.r.t the flattened input will be picked.

keepdims
boolean, optional, default=0 If true, the axis where we pick the elements is left in the result as dimension with size one.

mode
'clip', 'wrap', optional, default='clip' Specify how out-of-bound indices behave. Default is "clip", "clip" means clip to the range. So, if all indices mentioned are too large, they are replaced by the index that addresses the last element along an axis. "wrap" means to wrap around.

name
string, optional Name of the resulting symbol.

Details

output[i] = input[i, indices[i]]

By default, if any index mentioned is too large, it is replaced by the index that addresses the last element along an axis (the 'clip' mode).

This function supports n-dimensional input and (n-1)-dimensional indices arrays.

Examples::

x = [[[ 1., 2.], [ 3., 4.], [ 5., 6.]]
// picks elements with specified indices along axis 0 pick(x, y=[0,1], 0) = [ 1., 4.]
// picks elements with specified indices along axis 1 pick(x, y=[0,1,0], 1) = [ 1., 4., 5.]
// picks elements with specified indices along axis 1 using 'wrap' mode // to place indicies that would normally be out of bounds pick(x, y=[2,-1,-2], 1, mode='wrap') = [ 1., 4., 5.]

y = [[[ 1.], [ 0.], [ 2.]]
// picks elements with specified indices along axis 1 and dims are maintained pick(x, y, 1, keepdims=True) = [[ [ 2.], [ 3.], [ 6.]]

Defined in src/operator/tensor/broadcast_reduce_op_index.cc:L151

Value

out The result mx.symbol
mx.symbol.Pooling

Pooling: Performs pooling on the input.

Description

The shapes for 1-D pooling are

Usage

mx.symbol.Pooling(...)

Arguments

data NDArray-or-Symbol Input data to the pooling operator.
kernel Shape(tuple), optional, default=[] Pooling kernel size: (y, x) or (d, y, x)
pool.type 'avg', 'lp', 'max', 'sum',optional, default='max' Pooling type to be applied.
global.pool boolean, optional, default=0 Ignore kernel size, do global pooling based on current input feature map.
cudnn.off boolean, optional, default=0 Turn off cudnn pooling and use MXNet pooling operator.
pooling.convention 'full', 'same', 'valid',optional, default='valid' Pooling convention to be applied.
stride Shape(tuple), optional, default=[] Stride: for pooling (y, x) or (d, y, x). Defaults to 1 for each dimension.
pad Shape(tuple), optional, default=[] Pad for pooling: (y, x) or (d, y, x). Defaults to no padding.
p.value int or None, optional, default='None' Value of p for Lp pooling, can be 1 or 2, required for Lp Pooling.
count.include.pad boolean or None, optional, default=None Only used for AvgPool, specify whether to count padding elements for average calculation. For example, with a 5*5 kernel on a 3*3 corner of an image, the sum of the 9 valid elements will be divided by 25 if this is set to true, or it will be divided by 9 if this is set to false. Defaults to true.
layout None, 'NCDHW', 'NCHW', 'NCW', 'NDHWC', 'NHWC', 'NWC',optional, default='None' Set layout for input and output. Empty for default layout: NCW for 1d, NCHW for 2d and NCDHW for 3d.
nname string, optional Name of the resulting symbol.
Details

- **data** and **out**: *(batch_size, channel, width)* (NCW layout) or *(batch_size, width, channel)* (NWC layout).

The shapes for 2-D pooling are

- **data** and **out**: *(batch_size, channel, height, width)* (NCHW layout) or *(batch_size, height, width, channel)* (NHWC layout),

out\_height = f(height, kernel\[0\], pad\[0\], stride\[0\])
out\_width = f(width, kernel\[1\], pad\[1\], stride\[1\])

The definition of *f* depends on “pooling\_convention”, which has two options:

- **valid** (default)::
  \[f(x, k, p, s) = \text{floor}(x + 2p - k)/s + 1\]
- **full**, which is compatible with Caffe::
  \[f(x, k, p, s) = \text{ceil}(x + 2p - k)/s + 1\]

When “global\_pool” is set to be true, then global pooling is performed. It will reset “kernel=(height, width)” and set the appropriate padding to 0.

Three pooling options are supported by “pool\_type”:

- **avg**: average pooling - **max**: max pooling - **sum**: sum pooling - **lp**: Lp pooling

For 3-D pooling, an additional *depth* dimension is added before *height*. Namely the input data and output will have shape *(batch_size, channel, depth, height, width)* (NCDHW layout) or *(batch_size, depth, height, width, channel)* (NDHWC layout).

Notes on Lp pooling:

Lp pooling was first introduced by this paper: [https://arxiv.org/pdf/1204.3968.pdf](https://arxiv.org/pdf/1204.3968.pdf). L-1 pooling is simply sum pooling, while L-inf pooling is simply max pooling. We can see that Lp pooling stands between those two, in practice the most common value for p is 2.

For each window “X”, the mathematical expression for Lp pooling is:

\[f(X) = \sqrt[p]{\sum_x^X x^p}\]

Defined in src/operator/nn/pooling.cc:L416

Value

out The result mx.symbol

---

*mx.symbol.Pooling\_v1*

Pooling\_v1: This operator is DEPRECATED. Perform pooling on the input.

---

Description

The shapes for 2-D pooling is

Usage

mx.symbol.Pooling\_v1(...)
Arguments

data : NDArray-or-Symbol
Input data to the pooling operator.

kernel : Shape(tuple), optional
pooling kernel size: (y, x) or (d, y, x)

pool.type : string
"avg", "max", "sum", optional, default="max" Pooling type to be applied.

global.pool : boolean
Ignore kernel size, do global pooling based on current input feature map.

pooling.convention : string
"full", "valid", optional, default="valid" Pooling convention to be applied.

stride : Shape(tuple), optional
stride: for pooling (y, x) or (d, y, x)

pad : Shape(tuple), optional
pad for pooling: (y, x) or (d, y, x)

name : string
optional Name of the resulting symbol.

Details

- **data**: *(batch_size, channel, height, width)* - **out**: *(batch_size, num_filter, out_height, out_width)*, with:

  out_height = f(height, kernel[0], pad[0], stride[0])
  out_width = f(width, kernel[1], pad[1], stride[1])

The definition of *f* depends on “pooling_convention”, which has two options:

- **valid** (default):
  
  f(x, k, p, s) = floor((x+2*p-k)/s)+1

- **full**, which is compatible with Caffe:
  
  f(x, k, p, s) = ceil((x+2*p-k)/s)+1

But “global_pool” is set to be true, then do a global pooling, namely reset “kernel=(height, width)“.

Three pooling options are supported by “pool_type“:

- **avg**: average pooling
- **max**: max pooling
- **sum**: sum pooling

1-D pooling is special case of 2-D pooling with *weight=1* and *kernel[1]=1*.

For 3-D pooling, an additional *depth* dimension is added before *height*. Namely the input data will have shape *(batch_size, channel, depth, height, width)*.

Defined in src/operator/pooling_v1.cc:L104

Value

out The result mx.symbol
**mx.symbol.preloaded_multi_mp_sgd_mom_update**

*preloaded_multi_mp_sgd_mom_update: Momentum update function for multi-precision Stochastic Gradient Descent (SGD) optimizer.*

---

**Description**

Momentum update has better convergence rates on neural networks. Mathematically it looks like below:

**Usage**

```
mx.symbol.preloaded_multi_mp_sgd_mom_update(...)  
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>NDArray-or-Symbol[]</td>
<td>Weights, gradients, momentums, learning rates and weight decays</td>
</tr>
<tr>
<td>momentum</td>
<td>float, optional, default=0</td>
<td>The decay rate of momentum estimates at each epoch.</td>
</tr>
<tr>
<td>rescale.grad</td>
<td>float, optional, default=1</td>
<td>Rescale gradient to grad = rescale_grad*grad.</td>
</tr>
<tr>
<td>clip.gradient</td>
<td>float, optional, default=1</td>
<td>Clip gradient to the range of [-clip_gradient, clip_gradient]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If clip_gradient &lt;= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).</td>
</tr>
<tr>
<td>num.weights</td>
<td>int, optional, default='1'</td>
<td>Number of updated weights.</td>
</tr>
<tr>
<td>name</td>
<td>string, optional</td>
<td>Name of the resulting symbol.</td>
</tr>
</tbody>
</table>

**Details**

.. math::

\[
\begin{align*}
    v_1 &= \alpha \nabla J(W_0) \\
    v_t &= \gamma v_{t-1} - \alpha \nabla J(W_{t-1}) \\
    W_t &= W_{t-1} + v_t
\end{align*}
\]

It updates the weights using::

\[
    v = \text{momentum} \times v - \text{learning_rate} \times \text{gradient} \
    weight += v
\]

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.

Defined in src/operator/contrib/preloaded_multi_sgd.cc:L200

**Value**

```
out The result mx.symbol
```

**Description**

It updates the weights using::

**Usage**

```
mx.symbol.preloaded_multi_mp_sgd_update(...)```

**Arguments**

- **data**
  - NDArray-or-Symbol[] Weights, gradients, learning rates and weight decays

- **rescale.grad**
  - float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.

- **clip.gradient**
  - float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient]
    If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).

- **num.weights**
  - int, optional, default='1’ Number of updated weights.

- **name**
  - string, optional Name of the resulting symbol.

**Details**

weight = weight - learning_rate * (gradient + wd * weight)

Defined in src/operator/contrib/preloaded_multi_sgd.cc:L140

**Value**

```
out The result mx.symbol
```

---

**Description**

Momentum update has better convergence rates on neural networks. Mathematically it looks like below:

**Usage**

```
mx.symbol.preloaded_multi_sgd_mom_update(...)```
Arguments

data NDArray-or-Symbol[] Weights, gradients, momentum, learning rates and weight decays
momentum float, optional, default=0 The decay rate of momentum estimates at each epoch.
rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
num.weights int, optional, default=’1’ Number of updated weights.
name string, optional Name of the resulting symbol.

Details

.. math::
\v_1 = \alpha \nabla J(W_0) \v_t = \gamma \v_{t-1} - \alpha \nabla J(W_{t-1}) \ W_t = W_{t-1} + \v_t

It updates the weights using::
\v = momentum * \v - learning_rate * gradient weight += \v

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.
Defined in src/operator/contrib/preloaded_multi_sgd.cc:L91

Value

out The result mx.symbol

mx.symbol.preloaded_multi_sgd_update

preloaded_multi_sgd_update:Update function for Stochastic Gradient Descent (SDG) optimizer.

Description

It updates the weights using::

Usage

mx.symbol.preloaded_multi_sgd_update(…)

Arguments

data NDArray-or-Symbol[] Weights, gradients, learning rates and weight decays
rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
num.weights int, optional, default=’1’ Number of updated weights.
name string, optional Name of the resulting symbol.
Details
weight = weight - learning_rate * (gradient + wd * weight)
Defined in src/operator/contrib/preloaded_multi_sgd.cc:L42

Value
out The result mx.symbol

mx.symbol.prod

prod: Computes the product of array elements over given axes.

Description
Defined in src/operator/tensor/./broadcast_reduce_op.h:L31

Usage
mx.symbol.prod(...)

Arguments
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>NDArray-or-Symbol The input</td>
</tr>
<tr>
<td>axis</td>
<td>Shape or None, optional, default=None The axis or axes along which to perform the reduction. The default, ‘axis=()’, will compute over all elements into a scalar array with shape ‘(1,)’. If ‘axis’ is int, a reduction is performed on a particular axis. If ‘axis’ is a tuple of ints, a reduction is performed on all the axes specified in the tuple. If ‘exclude’ is true, reduction will be performed on the axes that are NOT in axis instead. Negative values means indexing from right to left.</td>
</tr>
<tr>
<td>keepdims</td>
<td>boolean, optional, default=0 If this is set to ‘True’, the reduced axes are left in the result as dimension with size one.</td>
</tr>
<tr>
<td>exclude</td>
<td>boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.</td>
</tr>
<tr>
<td>name</td>
<td>string, optional Name of the resulting symbol.</td>
</tr>
</tbody>
</table>

Value
out The result mx.symbol
**mx.symbol.radians**

**radians**: Converts each element of the input array from degrees to radians.

**Description**

.. math:: \text{radians}([0, 90, 180, 270, 360]) = [0, \pi/2, \pi, 3\pi/2, 2\pi]

**Usage**

mx.symbol.radians(...)

**Arguments**

- **data**: NDArray-or-Symbol The input array.
- **name**: string, optional Name of the resulting symbol.

**Details**

The storage type of “radians“ output depends upon the input storage type:

- radians(default) = default
- radians(row_sparse) = row_sparse
- radians(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L351

**Value**

out The result mx.symbol

---

**mx.symbol.random_exponential**

**random_exponential**: Draw random samples from an exponential distribution.

**Description**

Samples are distributed according to an exponential distribution parametrized by *lambda* (rate).

**Usage**

mx.symbol.random_exponential(...)
mx.symbol.random_gamma

Arguments

Arguments

lam float, optional, default=1 Lambda parameter (rate) of the exponential distribution.
shape Shape(tuple), optional, default=None Shape of the output.
ctx string, optional, default="" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
dtype 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
name string, optional Name of the resulting symbol.

Details

Example::
exponential(lam=4, shape=(2,2)) = [[ 0.0097189 , 0.08999364], [ 0.04146638, 0.31715935]]
Defined in src/operator/random/sample_op.cc:L137

Value

out The result mx.symbol

Description

Samples are distributed according to a gamma distribution parametrized by *alpha* (shape) and *beta* (scale).

Usage

mx.symbol.random_gamma(...)
Details

Example:

\[
\text{gamma(alpha=9, beta=0.5, shape=(2,2)) = } \begin{bmatrix} 7.10486984, & 3.37695289 \end{bmatrix}, \begin{bmatrix} 3.91697288, & 3.65933681 \end{bmatrix}
\]

Defined in src/operator/random/sample_op.cc:L125

Value

out The result mx.symbol

mx.symbol.random_generalized_negative_binomial
random_generalized_negative_binomial:Draw random samples from a generalized negative binomial distribution.

Description

Samples are distributed according to a generalized negative binomial distribution parametrized by *mu* (mean) and *alpha* (dispersion). *alpha* is defined as *1/k* where *k* is the failure limit of the number of unsuccessful experiments (generalized to real numbers). Samples will always be returned as a floating point data type.

Usage

mx.symbol.random_generalized_negative_binomial(...)

Arguments

mu float, optional, default=1 Mean of the negative binomial distribution.
alpha float, optional, default=1 Alpha (dispersion) parameter of the negative binomial distribution.
shape Shape(tuple), optional, default=None Shape of the output.
ctx string, optional, default=" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
dtype 'None', 'float16', 'float32', 'float64',optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
name string, optional Name of the resulting symbol.

Details

Example:

generalized_negative_binomial(mu=2.0, alpha=0.3, shape=(2,2)) = \begin{bmatrix} 2., & 1. \end{bmatrix}, \begin{bmatrix} 6., & 4. \end{bmatrix}

Defined in src/operator/random/sample_op.cc:L179

Value

out The result mx.symbol
mx.symbol.random_negative_binomial

*random_negative_binomial*: Draw random samples from a negative binomial distribution.

Description

Samples are distributed according to a negative binomial distribution parametrized by \( k \) (limit of unsuccessful experiments) and \( p \) (failure probability in each experiment). Samples will always be returned as a floating point data type.

Usage

```python
mx.symbol.random_negative_binomial(...)```

Arguments

- **k**: int, optional, default='1' Limit of unsuccessful experiments.
- **p**: float, optional, default=1 Failure probability in each experiment.
- **shape**: Shape(tuple), optional, default=None Shape of the output.
- **ctx**: string, optional, default=’Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **name**: string, optional Name of the resulting symbol.

Details

Example::

```python
negative_binomial(k=3, p=0.4, shape=(2,2)) = [[ 4., 7.], [ 2., 5.]]
```

Defined in src/operator/random/sample_op.cc:L164

Value

```python
out The result mx.symbol
```
random_normal: Draw random samples from a normal (Gaussian) distribution.

Description

.. note:: The existing alias “normal” is deprecated.

Usage

.. code-block:: python

    mx.symbol.random_normal(...)

Arguments

- **loc**: float, optional, default=0 Mean of the distribution.
- **scale**: float, optional, default=1 Standard deviation of the distribution.
- **shape**: Shape(tuple), optional, default=None Shape of the output.
- **ctx**: string, optional, default=" Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **name**: string, optional Name of the resulting symbol.

Details

Samples are distributed according to a normal distribution parametrized by *loc* (mean) and *scale* (standard deviation).

Example::

    normal(loc=0, scale=1, shape=(2,2)) = [[ 1.89171135, -1.16881478], [-1.23474145, 1.55807114]]

Defined in src/operator/random/sample_op.cc:L113

Value

- **out**: The result mx.symbol
mx.symbol.random_pdf_dirichlet

random_pdf_dirichlet: Computes the value of the PDF of *sample* of Dirichlet distributions with parameter *alpha*.

**Description**

The shape of *alpha* must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *alpha*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the value of *alpha* at index *i*.

**Usage**

mx.symbol.random_pdf_dirichlet(...)

**Arguments**

- **sample**: NDArray-or-Symbol. Samples from the distributions.
- **alpha**: NDArray-or-Symbol. Concentration parameters of the distributions.
- **is.log**: boolean, optional, default=0. If set, compute the density of the log-probability instead of the probability.
- **name**: string, optional. Name of the resulting symbol.

**Details**

Examples:

```python
random_pdf_dirichlet(sample=[[1,2],[2,3],[3,4]], alpha=[2.5, 2.5]) = [38.413498, 199.60245, 564.56085]
sample = [[[1, 2, 3], [10, 20, 30], [100, 200, 300]], [[0.1, 0.2, 0.3], [0.01, 0.02, 0.03], [0.001, 0.002, 0.003]]]
random_pdf_dirichlet(sample=sample, alpha=[0.1, 0.4, 0.9]) = [[2.3257459e-02, 5.8420084e-04, 1.4674458e-05], [9.2589635e-01, 3.6860607e+01, 1.4674468e+03]]
```

Defined in src/operator/random/pdf_op.cc:L316

**Value**

- **out**: The result mx.symbol
mx.symbol.random_pdf_exponential

random_pdf_exponential: Computes the value of the PDF of *sample* of exponential distributions with parameters *lam* (rate).

Description

The shape of *lam* must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *lam*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the value of *lam* at index *i*.

Usage

mx.symbol.random_pdf_exponential(...) 

Arguments

    sample: NDArray-or-Symbol Samples from the distributions.
    lam: NDArray-or-Symbol Lambda (rate) parameters of the distributions.
    is.log: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
    name: string, optional Name of the resulting symbol.

Details

Examples:

    random_pdf_exponential(sample=[[1, 2, 3], lam=[1]) = [[0.36787945, 0.13533528, 0.04978707]]
    sample = [[1,2,3], [1,2,3], [1,2,3]]
    random_pdf_exponential(sample=sample, lam=[[0.36787945, 0.13533528, 0.04978707], [0.30326533, 0.18393973, 0.11156508], [0.1947002, 0.15163267, 0.11809164]]
    Defined in src/operator/random/pdf_op.cc:L305

Value

    out The result mx.symbol
mx.symbol.random_pdf_gamma

random_pdf_gamma: Computes the value of the PDF of *sample* of gamma distributions with parameters *alpha* (shape) and *beta* (rate).

Description
*alpha* and *beta* must have the same shape, which must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *alpha* and *beta*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the values of *alpha* and *beta* at index *i*.

Usage
mx.symbol.random_pdf_gamma(...)

Arguments
- **sample**: NDArray-or-Symbol Samples from the distributions.
- **alpha**: NDArray-or-Symbol Alpha (shape) parameters of the distributions.
- **is.log**: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
- **beta**: NDArray-or-Symbol Beta (scale) parameters of the distributions.
- **name**: string, optional Name of the resulting symbol.

Details
Examples:
```python
random_pdf_gamma(sample=[[1,2,3,4,5]], alpha=[5], beta=[1]) = [[0.01532831, 0.09022352, 0.16803136, 0.19536681, 0.17546739]]
sample = [[1, 2, 3, 4, 5], [2, 3, 4, 5, 6], [3, 4, 5, 6, 7]]
random_pdf_gamma(sample=sample, alpha=[5,6,7], beta=[1,1,1]) = [[0.01532831, 0.09022352, 0.16803136, 0.19536681, 0.17546739], [0.03608941, 0.10081882, 0.15629345, 0.17546739, 0.16062315], [0.05040941, 0.10419563, 0.14622283, 0.16062315, 0.14900276]]
```
Defined in src/operator/random/pdf_op.cc:L303

Value
- **out**: The result mx.symbol
random_pdf_generalized_negative_binomial: Computes the value of the PDF of *sample* of generalized negative binomial distributions with parameters *mu* (mean) and *alpha* (dispersion). This can be understood as a reparameterization of the negative binomial, where *k* = *1 / alpha* and *p* = *1 / (mu ∘ alpha + 1)*.

Description

*mu* and *alpha* must have the same shape, which must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *mu* and *alpha*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the values of *mu* and *alpha* at index *i*.

Usage

```
mx.symbol.random_pdf_generalized_negative_binomial(...)```

Arguments

- **sample**: NDArray-or-Symbol Samples from the distributions.
- **mu**: NDArray-or-Symbol Means of the distributions.
- **is.log**: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
- **alpha**: NDArray-or-Symbol Alpha (dispersion) parameters of the distributions.
- **name**: string, optional Name of the resulting symbol.

Details

Examples:
```
random_pdf_generalized_negative_binomial(sample=[[1, 2, 3, 4]], alpha=[1], mu=[1]) = [[0.25, 0.125, 0.0625, 0.03125]]
sample = [(1,2,3,4), (1,2,3,4)] random_pdf_generalized_negative_binomial(sample=sample, alpha=[1, 0.6666], mu=[1, 1.5]) = [[0.25, 0.125, 0.0625, 0.03125 ], [0.26517063, 0.16573331, 0.09667706, 0.05437994]]
```

Defined in src/operator/random/pdf_op.cc:L314

Value

```
out The result mx.symbol
```
mx.symbol.random_pdf_negative_binomial

**random_pdf_negative_binomial**: Computes the value of the PDF of samples of negative binomial distributions with parameters *k* (failure limit) and *p* (failure probability).

**Description**

*k* and *p* must have the same shape, which must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *k* and *p*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the values of *k* and *p* at index *i*.

**Usage**

mx.symbol.random_pdf_negative_binomial(...)

**Arguments**

- **sample**: NDArray-or-Symbol Samples from the distributions.
- **k**: NDArray-or-Symbol Limits of unsuccessful experiments.
- **is.log**: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
- **p**: NDArray-or-Symbol Failure probabilities in each experiment.
- **name**: string, optional Name of the resulting symbol.

**Details**

Examples:

```
random_pdf_negative_binomial(sample=[[1,2,3,4]], k=[1], p=[0.5]) = [[0.25, 0.125, 0.0625, 0.03125]]
```

# Note that k may be real-valued sample = [[1,2,3,4], [1,2,3,4]] random_pdf_negative_binomial(sample=sample, k=[1, 1.5], p=[0.5, 0.5]) = [[0.25, 0.125, 0.0625, 0.03125 ], [0.26516506, 0.16572815, 0.09667476, 0.05437956]]

Defined in src/operator/random/pdf_op.cc:L310

**Value**

out The result mx.symbol
mx.symbol.random_pdf_normal

random_pdf_normal: Computes the value of the PDF of *sample* of normal distributions with parameters *mu* (mean) and *sigma* (standard deviation).

Description

*mu* and *sigma* must have the same shape, which must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *mu* and *sigma*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the values of *mu* and *sigma* at index *i*.

Usage

mx.symbol.random_pdf_normal(...)

Arguments

- **sample**: NDArray-or-Symbol Samples from the distributions.
- **mu**: NDArray-or-Symbol Means of the distributions.
- **is.log**: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
- **sigma**: NDArray-or-Symbol Standard deviations of the distributions.
- **name**: string, optional Name of the resulting symbol.

Details

Examples:

```python
sample = [[-2, -1, 0, 1, 2]]
random_pdf_normal(sample=sample, mu=[0], sigma=[1]) = [[0.05399097, 0.24197073, 0.3989423, 0.24197073, 0.05399097]]
random_pdf_normal(sample=sample*2, mu=[0,0], sigma=[1,2]) = [[0.05399097, 0.24197073, 0.3989423, 0.24197073, 0.05399097], [0.12098537, 0.17603266, 0.19947115, 0.17603266, 0.12098537]]
```

Defined in src/operator/random/pdf_op.cc:L300

Value

- **out**: The result mx.symbol
mx.symbol.random_pdf_poisson

random_pdf_poisson: Computes the value of the PDF of *sample* of Poisson distributions with parameters *lam* (rate).

**Description**

The shape of *lam* must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *lam*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the value of *lam* at index *i*.

**Usage**

mx.symbol.random_pdf_poisson(...)

**Arguments**

- sample: NDArray-or-Symbol Samples from the distributions.
- lam: NDArray-or-Symbol Lambda (rate) parameters of the distributions.
- is.log: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
- name: string, optional Name of the resulting symbol.

**Details**

Examples:

```python
random_pdf_poisson(sample=[[0,1,2,3]], lam=[1]) = [0.36787945, 0.36787945, 0.18393973, 0.06131324]
sample = [[0,1,2,3], [0,1,2,3], [0,1,2,3]]
random_pdf_poisson(sample=sample, lam=[1,2,3]) = [0.36787945, 0.36787945, 0.18393973, 0.06131324],
[0.13533528, 0.27067056, 0.27067056, 0.18044704], [0.04978707, 0.14936121, 0.22404182, 0.22404182]
```

Defined in src/operator/random/pdf_op.cc:L307

**Value**

out The result mx.symbol
mx.symbol.random_pdf_uniform

**random_pdf_uniform:** Computes the value of the PDF of *sample* of uniform distributions on the intervals given by *[low,high]*.

**Description**

*low* and *high* must have the same shape, which must match the leftmost subshape of *sample*. That is, *sample* can have the same shape as *low* and *high*, in which case the output contains one density per distribution, or *sample* can be a tensor of tensors with that shape, in which case the output is a tensor of densities such that the densities at index *i* in the output are given by the samples at index *i* in *sample* parameterized by the values of *low* and *high* at index *i*.

**Usage**

mx.symbol.random_pdf_uniform(...)

**Arguments**

- **sample**: NDArray-or-Symbol Samples from the distributions.
- **low**: NDArray-or-Symbol Lower bounds of the distributions.
- **is.log**: boolean, optional, default=0 If set, compute the density of the log-probability instead of the probability.
- **high**: NDArray-or-Symbol Upper bounds of the distributions.
- **name**: string, optional Name of the resulting symbol.

**Details**

Examples:

```python
random_pdf_uniform(sample=[[1,2,3,4]], low=[0], high=[10]) = [0.1, 0.1, 0.1, 0.1]

sample = [[[1, 2, 3], [1, 2, 3]], [[1, 2, 3], [1, 2, 3]]] low = [[0, 0], [0, 0]] high = [[5, 10], [15, 20]] random_pdf_uniform(sample=sample, low=low, high=high) = [[0.06667, 0.06667, 0.06667], [0.05, 0.05, 0.05]]
```

Defined in src/operator/random/pdf_op.cc:L298

**Value**

out The result mx.symbol
**random_poisson:** Draw random samples from a Poisson distribution.

**Description**

Samples are distributed according to a Poisson distribution parametrized by \(*\text{lambda}*\) (rate). Samples will always be returned as a floating point data type.

**Usage**

```
mx.symbol.random_poisson(...)
```

**Arguments**

- `lam` float, optional, default=1 Lambda parameter (rate) of the Poisson distribution.
- `shape` Shape(tuple), optional, default=None Shape of the output.
- `ctx` string, optional, default="" Context of output, in format [cpu|gpus|cpu_pinned](n). Only used for imperative calls.
- `dtype` 'None', 'float16', 'float32', 'float64', optional, default='None' Dtype of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- `name` string, optional Name of the resulting symbol.

**Details**

Example::

```
poisson(lam=4, shape=(2,2)) = [[ 5., 2.], [ 4., 6.]]
```

Defined in src/operator/random/sample_op.cc:L150

**Value**

`out` The result `mx.symbol`

**random_randint:** Draw random samples from a discrete uniform distribution.

**Description**

Samples are uniformly distributed over the half-open interval \(*[\text{low}, \text{high})*\) (includes \(*\text{low})*, but excludes \(*\text{high})*).
mx.symbol.random_uniform

Usage

mx.symbol.random.randint(...)

Arguments

- **low**: long, required Lower bound of the distribution.
- **high**: long, required Upper bound of the distribution.
- **shape**: Shape(tuple), optional, default=None Shape of the output.
- **ctx**: string, optional, default=?” Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- **dtype**: 'None', 'int32', 'int64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to int32 if not defined (dtype=None).
- **name**: string, optional Name of the resulting symbol.

Details

Example::

randint(low=0, high=5, shape=(2,2)) = [[ 0, 2], [ 3, 1]]

Defined in src/operator/random/sample_op.cc:L194

Value

out The result mx.symbol

mx.symbol.random_uniform

random_uniform: Draw random samples from a uniform distribution.

Description

.. note:: The existing alias “uniform” is deprecated.

Usage

mx.symbol.random_uniform(...)

Arguments

- **low**: float, optional, default=0 Lower bound of the distribution.
- **high**: float, optional, default=1 Upper bound of the distribution.
- **shape**: Shape(tuple), optional, default=None Shape of the output.
- **ctx**: string, optional, default=?” Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **name**: string, optional Name of the resulting symbol.
mx.symbol.ravel_multi_index

ravel_multi_index: Converts a batch of index arrays into an array of flat indices. The operator follows numpy conventions so a single multi index is given by a column of the input matrix. The leading dimension may be left unspecified by using -1 as placeholder.

Description

Examples::

A = [[3,6,6],[4,5,1]] ravel(A, shape=(7,6)) = [22,41,37] ravel(A, shape=(-1,6)) = [22,41,37]

Usage

mx.symbol.ravel_multi_index(…)

Arguments

data NDArray-or-Symbol Batch of multi-indices
shape Shape(tuple), optional, default=None Shape of the array into which the multi-indices apply.
name string, optional Name of the resulting symbol.

Details

Defined in src/operator/tensor/ravel.cc:L42

Value

out The result mx.symbol
mx.symbol.rcbrt

rcbrt: Returns element-wise inverse cube-root value of the input.

Description

.. math:: \text{rcbrt}(x) = 1/\sqrt[3]{x}

Usage

mx.symbol.rcbrt(...)

Arguments

data
NDArray-or-Symbol The input array.

name
string, optional Name of the resulting symbol.

Details

Example:

rcbrt([1,8,-125]) = [1.0, 0.5, -0.2]

Defined in src/operator/tensor/elemwise_unary_op_pow.cc:L323

Value

out The result mx.symbol

mx.symbol.reciprocal

reciprocal: Returns the reciprocal of the argument, element-wise.

Description

Calculates 1/x.

Usage

mx.symbol.reciprocal(...)

Arguments

data
NDArray-or-Symbol The input array.

name
string, optional Name of the resulting symbol.
Details

Example::

reciprocal([-2, 1, 3, 1.6, 0.2]) = [-0.5, 1.0, 0.33333334, 0.625, 5.0]

Defined in src/operator/tensor/elemwise_unary_op_pow.cc:L43

Value

out The result mx.symbol

---

mx.symbol.relu

relu: Computes rectified linear activation.

Description

.. math:: \text{max}(\text{features}, 0)

Usage

mx.symbol.relu(...)

Arguments

data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.

Details

The storage type of “relu” output depends upon the input storage type:
- relu(default) = default - relu(row_sparse) = row_sparse - relu(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L85

Value

out The result mx.symbol
\texttt{mx.symbol.repeat} \hspace{1cm} \texttt{repeat:} Repeats elements of an array. By default, “repeat” flattens the input array into 1-D and then repeats the elements:\[ x = [[1, 2], [3, 4]] \]\[ \text{repeat}(x, \text{repeats}=2) = [1.0, 1.0, 2.0, 2.0, 3.0, 3.0, 4.0, 4.0] \] The parameter “axis” specifies the axis along which to perform repeat:\[ \text{repeat}(x, \text{repeats}=2, \text{axis}=1) = [[1.0, 2.0], [3.0, \text{repeats}=2, \text{axis}=-1] = [[1.0, 2.0], [3.0, 4.0], [3.0, 4.0]] \]

**Description**

Defined in src/operator/tensor/matrix_op.cc:L744

**Usage**

\texttt{mx.symbol.repeat(...)}

**Arguments**

- \texttt{data} \hspace{1cm} \texttt{NDArray-or-Symbol} Input data array
- \texttt{repeats} \hspace{1cm} \texttt{int, required} The number of repetitions for each element.
- \texttt{axis} \hspace{1cm} \texttt{int or None, optional, default=None} The axis along which to repeat values. The negative numbers are interpreted counting from the backward. By default, use the flattened input array, and return a flat output array.
- \texttt{name} \hspace{1cm} \texttt{string, optional} Name of the resulting symbol.

**Value**

\texttt{out} The result \texttt{mx.symbol}

---

\texttt{mx.symbol.reset_arrays} \hspace{1cm} \texttt{reset_arrays:} Set to zero multiple arrays

**Description**

Defined in src/operator/contrib/reset_arrays.cc:L36

**Usage**

\texttt{mx.symbol.reset_arrays(...)}
mx.symbol.Reshape

Arguments

- **data**: NDArray-or-Symbol[] Arrays
- **num.arrays**: int, required number of input arrays.
- **name**: string, optional Name of the resulting symbol.

Value

- **out**: The result mx.symbol

```
mx.symbol.Reshape
Reshape: Reshapes the input array. .. note:: “Reshape“ is deprecated, use “reshape“ Given an array and a shape, this function returns a copy of the array in the new shape. The shape is a tuple of integers such as (2,3,4). The size of the new shape should be same as the size of the input array. Example:: reshape([1,2,3,4], shape=(2,2)) = [[1,2],
[3,4]] Some dimensions of the shape can take special values from the set 0, -1, -2, -3, -4. The significance of each is explained below: - “0“ copy this dimension from the input to the output shape. Example:: - input shape = (2,3,4), shape = (4,0,2), output shape = (4,3,2) - input shape = (2,3,4), shape = (2,0,0), output shape = (2,3,4) - “-1“ infers the dimension of the output shape by using the remainder of the input dimensions keeping the size of the new array same as that of the input array. At most one dimension of shape can be -1. Example:: - input shape = (2,3,4), shape = (6,1,-1), output shape = (6,1,4) - input shape = (2,3,4), shape = (3,-1,8), output shape = (3,1,8) - input shape = (2,3,4), shape = (-1,), output shape = (24,) - “-2“ copy all/remainder of the input dimensions to the output shape. Example:: - input shape = (2,3,4), shape = (-2,), output shape = (2,3,4) - input shape = (2,3,4), shape = (2,-2), output shape = (2,3,4) - input shape = (-2,1,1), output shape = (2,3,4,1,1) - “-3“ use the product of two consecutive dimensions of the input shape as the output dimension. Example:: - input shape = (2,3,4), shape = (-3,4), output shape = (6,4) - input shape = (2,3,4,5), shape = (-3,-3), output shape = (6,20) - input shape = (2,3,4), shape = (0,-3), output shape = (2,12) - input shape = (2,3,4), shape = (-3,-2), output shape = (6,4) - “-4“ split one dimension of the input into two dimensions passed subsequent to -4 in shape (can contain -1). Example:: - input shape = (2,3,4), shape = (-4,1,2,-2), output shape = (1,2,3,4) - input shape = (2,3,4), shape = (2,-4,-1,3,-2), output shape = (2,1,3,4) If the argument ‘reverse‘ is set to 1, then the special values are inferred from right to left. Example:: - without reverse=1, for input shape = (10,5,4), shape = (-1,0), output shape would be (40,5) - with reverse=1, output shape will be (50,4).
```

Description

Defined in src/operator/tensor/matrix_op.cc:L175
Usage

```
mx.symbol.Reshape(...)
```

Arguments

- **data**: NDArray-or-Symbol Input data to reshape.
- **shape**: Shape(tuple), optional, default=[] The target shape
- **reverse**: boolean, optional, default=0 If true then the special values are inferred from right to left
- **target.shape**: Shape(tuple), optional, default=[] (Deprecated! Use “shape” instead.) Target new shape. One and only one dim can be 0, in which case it will be inferred from the rest of dims
- **keep.highest**: boolean, optional, default=0 (Deprecated! Use “shape” instead.) Whether keep the highest dim unchanged. If set to true, then the first dim in target_shape is ignored, and always fixed as input
- **name**: string, optional Name of the resulting symbol.

Value

```
out The result mx.symbol
```
**mx.symbol.reshape**

Reshapes the input array. .. note:: “Reshape” is deprecated, use “reshape” Given an array and a shape, this function returns a copy of the array in the new shape. The shape is a tuple of integers such as (2,3,4). The size of the new shape should be same as the size of the input array. Example:: reshape([1,2,3,4], shape=(2,2)) = [[1,2], [3,4]] Some dimensions of the shape can take special values from the set 0, -1, -2, -3, -4. The significance of each is explained below: - “0“ copy this dimension from the input to the output shape. Example:: - input shape = (2,3,4), shape = (4,0,2), output shape = (4,3,2) - input shape = (2,3,4), shape = (2,0,0), output shape = (2,3,4) - “-1“ infers the dimension of the output shape by using the remainder of the input dimensions keeping the size of the new array same as that of the input array. At most one dimension of shape can be -1. Example:: - input shape = (2,3,4), shape = (6,1,-1), output shape = (6,1,4) - input shape = (2,3,4), shape = (3,-1,8), output shape = (3,1,8) - input shape = (2,3,4), shape=-(-1), output shape = (24,) - “-2“ copy all/remainder of the input dimensions to the output shape. Example:: - input shape = (2,3,4), shape = (-2), output shape = (2,3,4) - input shape = (2,3,4), shape = (2,-2), output shape = (2,3,4) - input shape = (2,3,4), shape = (-2,1,1), output shape = (2,3,4,1) - “-3“ use the product of two consecutive dimensions of the input shape as the output dimension. Example:: - input shape = (2,3,4), shape = (-3,4), output shape = (6,4) - input shape = (2,3,4,5), shape = (-3,-3), output shape = (6,20) - input shape = (2,3,4), shape = (0,-3), output shape = (2,12) - input shape = (2,3,4), shape = (-3,-2), output shape = (6,4) - “-4“ split one dimension of the input into two dimensions passed subsequent to -4 in shape (can contain -1). Example:: - input shape = (2,3,4), shape = (-4,1,2,-2), output shape =(1,2,3,4) - input shape = (2,3,4), shape = (2,-4,-1,3,-2), output shape = (2,1,3,4) If the argument ‘reverse’ is set to 1, then the special values are inferred from right to left. Example:: - without reverse=1, for input shape = (10,5,4), shape = (-1,0), output shape would be (40,5) - with reverse=1, output shape will be (50,4).

**Description**

Defined in src/operator/tensor/matrix_op.cc:L175

**Usage**

mx.symbol.reshape(...)

**Arguments**

data NDArray-or-Symbol Input data to reshape.

shape Shape(tuple), optional, default=[] The target shape

reverse boolean, optional, default=0 If true then the special values are inferred from right to left
**mx.symbol.reshape_like**

**reshape_like:** Reshape some or all dimensions of ‘lhs’ to have the same shape as some or all dimensions of ‘rhs’.

### Description

Returns a **view** of the ‘lhs’ array with a new shape without altering any data.

### Usage

```python
mx.symbol.reshape_like(...)
```

### Arguments

- **lhs**
  - NDArray-or-Symbol: First input.
- **rhs**
  - NDArray-or-Symbol: Second input.
- **lhs.begin**
  - int or None, optional, default=’None’: Defaults to 0. The beginning index along which the lhs dimensions are to be reshaped. Supports negative indices.
- **lhs.end**
  - int or None, optional, default=’None’: Defaults to None. The ending index along which the lhs dimensions are to be used for reshaping. Supports negative indices.
- **rhs.begin**
  - int or None, optional, default=’None’: Defaults to 0. The beginning index along which the rhs dimensions are to be used for reshaping. Supports negative indices.
- **rhs.end**
  - int or None, optional, default=’None’: Defaults to None. The ending index along which the rhs dimensions are to be used for reshaping. Supports negative indices.
- **name**
  - string, optional: Name of the resulting symbol.
mx.symbol.reverse

**Details**

Example::

```python
x = [1, 2, 3, 4, 5, 6] y = [[0, -4], [3, 2], [2, 2]] reshape_like(x, y) = [[1, 2], [3, 4], [5, 6]]
```

More precise control over how dimensions are inherited is achieved by specifying \ slices over the ‘lhs’ and ‘rhs’ array dimensions. Only the sliced ‘lhs’ dimensions \ are reshaped to the ‘rhs’ sliced dimensions, with the non-sliced ‘lhs’ dimensions staying the same.

Examples::

- lhs shape = (30,7), rhs shape = (15,2,4), lhs_begin=0, lhs_end=1, rhs_begin=0, rhs_end=2, output shape = (15,2,7)
- lhs shape = (3,5), rhs shape = (1,15,4), lhs_begin=0, lhs_end=2, rhs_begin=1, rhs_end=2, output shape = (15)

Negative indices are supported, and ‘None’ can be used for either ‘lhs_end’ or ‘rhs_end’ to indicate the end of the range.

Example::

- lhs shape = (30, 12), rhs shape = (4, 2, 2, 3), lhs_begin=-1, lhs_end=None, rhs_begin=1, rhs_end=None, output shape = (30, 2, 2, 3)

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L511

**Value**

out The result mx.symbol

---

## mx.symbol.reverse

reverse: Reverses the order of elements along given axis while preserving array shape. Note: reverse and flip are equivalent. We use reverse in the following examples. Examples::

```python
x = [[ 0., 1., 2., 3., 4.], [ 5., 6., 7., 8., 9.]] reverse(x, axis=0) = [[ 5., 6., 7., 8., 9.], [ 0., 1., 2., 3., 4.]]
```

```python
reverse(x, axis=1) = [[ 4., 3., 2., 1., 0.], [ 9., 8., 7., 6., 5.]]
```

Defined in src/operator/tensor/matrix_op.cc:L832

**Usage**

mx.symbol.reverse(...)

**Arguments**

- **data**: NDArray-or-Symbol Input data array
- **axis**: Shape(tuple), required The axis which to reverse elements.
- **name**: string, optional Name of the resulting symbol.

**Value**

out The result mx.symbol
mx.symbol.rint

**rint:** Returns element-wise rounded value to the nearest integer of the input.

**Description**

.. note:: - For input “n.5” “rint” returns “n” while “round” returns “n+1”. - For input “-n.5” both “rint” and “round” returns “-n-1”.

**Usage**

mx.symbol.rint(...)

**Arguments**

- **data**: NDArray-or-Symbol The input array.
- **name**: string, optional Name of the resulting symbol.

**Details**

Example::

rint([-1.5, 1.5, -1.9, 1.9, 2.1]) = [-2, 1, -2, 2, 2]

The storage type of “rint” output depends upon the input storage type:
- rint(default) = default - rint(row_sparse) = row_sparse - rint(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L798

**Value**

out The result mx.symbol

mx.symbol.rmspropalex_update

**rmspropalex_update:** Update function for RMSPropAlex optimizer.

**Description**

‘RMSPropAlex’ is non-centered version of ‘RMSProp’.

**Usage**

mx.symbol.rmspropalex_update(...)
Arguments

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **n**: NDArray-or-Symbol n
- **g**: NDArray-or-Symbol g
- **delta**: NDArray-or-Symbol delta
- **lr**: float, required Learning rate
- **rho**: float, optional, default=0.949999988 Decay rate.
- **momentum**: float, optional, default=0.999999976 Decay rate.
- **epsilon**: float, optional, default=9.99999994e-09 A small constant for numerical stability.
- **wd**: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient]
  If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- **clip.weights**: float, optional, default=-1 Clip weights to the range of [-clip_weights, clip_weights]
  If clip_weights <= 0, weight clipping is turned off. weights = max(min(weights, clip_weights), -clip_weights).
- **name**: string, optional Name of the resulting symbol.

Details

Define \(E[g^2]_t\) is the decaying average over past squared gradient and \(E[g]_t\) is the decaying average over past gradient.

\[
\begin{align*}
E[g^2]_t &= \rho \times E[g^2]_{t-1} + (1 - \rho) \times g_t^2 \\
E[g]_t &= \rho \times E[g]_{t-1} + (1 - \rho) \times g_t
\end{align*}
\]

\[
momentum_t = \gamma \times momentum_{t-1} - \frac{\eta}{\sqrt{E[g^2]_t - E[g]_t^2}} + \epsilon g_t
\]

The update step is

\[
\theta_{t+1} = \theta_t + momentum_t
\]

The RMSPPropAlex code follows the version in http://arxiv.org/pdf/1308.0850v5.pdf Eq(38) - Eq(45) by Alex Graves, 2013.

Graves suggests the momentum term \(\rho\) to be 0.95, \(\gamma\) to be 0.9 and the learning rate \(\eta\) to be 0.0001.

Defined in src/operator/optimizer_op.cc:L827

Value

- **out**: The result mx.symbol
Description

‘RMSprop’ is a variant of stochastic gradient descent where the gradients are divided by a cache which grows with the sum of squares of recent gradients.

Usage

mx-symbol-rmsprop-update(...)

Arguments

- **weight**: NDArray-or-Symbol Weight
- **grad**: NDArray-or-Symbol Gradient
- **n**: NDArray-or-Symbol n
- **lr**: float, required Learning rate
- **rho**: float, optional, default=0.949999988 The decay rate of momentum estimates.
- **epsilon**: float, optional, default=9.99999994e-09 A small constant for numerical stability.
- **wd**: float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
- **rescale.grad**: float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
- **clip.gradient**: float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
- **clip.weights**: float, optional, default=-1 Clip weights to the range of [-clip_weights, clip_weights] If clip_weights <= 0, weight clipping is turned off. weights = max(min(weights, clip_weights), -clip_weights).
- **name**: string, optional Name of the resulting symbol.

Details

‘RMSProp’ is similar to ‘AdaGrad’, a popular variant of ‘SGD’ which adaptively tunes the learning rate of each parameter. ‘AdaGrad’ lowers the learning rate for each parameter monotonically over the course of training. While this is analytically motivated for convex optimizations, it may not be ideal for non-convex problems. ‘RMSProp’ deals with this heuristically by allowing the learning rates to rebound as the denominator decays over time.

Define the Root Mean Square (RMS) error criterion of the gradient as \( \text{RMS}[g]_t = \sqrt{E[g^2]_t + \epsilon} \), where ‘\( g \)’ represents gradient and ‘\( E[g^2]_t \)’ is the decaying average over past squared gradient.
The $E[g^{2}]_t$ is given by:
.. math:: E[g^{2}]_t = \rho \ast E[g^{2}]_{t-1} + (1-\rho) \ast g^{2}_t

The update step is
.. math:: \theta_{t+1} = \theta_{t} - \frac{\eta}{\text{RMS}[g]_t} g_t


Hinton suggests the momentum term $\\rho$ to be 0.9 and the learning rate $\eta$ to be 0.001.

Defined in src/operator/optimizer_op.cc:L788

Value

out The result mx.symbol

mx.symbol.RNN

RNN: Applies recurrent layers to input data. Currently, vanilla RNN, LSTM and GRU are implemented, with both multi-layer and bidirectional support.

Description

When the input data is of type float32 and the environment variables MXNET_CUDA_ALLOW_TENSOR_CORE and MXNET_CUDA_TENSOR_OP_MATH_ALLOW_CONVERSION are set to 1, this operator will try to use pseudo-float16 precision (float32 math with float16 I/O) precision in order to use Tensor Cores on suitable NVIDIA GPUs. This can sometimes give significant speedups.

Usage

mx.symbol.RNN(...)

Arguments

data NDArray-or-Symbol Input data to RNN
parameters NDArray-or-Symbol Vector of all RNN trainable parameters concatenated
state NDArray-or-Symbol initial hidden state of the RNN
state.cell NDArray-or-Symbol initial cell state for LSTM networks (only for LSTM)
sequence.length NDArray-or-Symbol Vector of valid sequence lengths for each element in batch. (Only used if use_sequence_length kwarg is True)
state.size int (non-negative), required size of the state for each layer
num.layers int (non-negative), required number of stacked layers
bidirectional boolean, optional, default=0 whether to use bidirectional recurrent layers
mode ‘gru’, ‘lstm’, ‘rnn_relu’, ‘rnn_tanh’, required the type of RNN to compute
`mx.symbol.RNN`

float, optional, default=0 drop rate of the dropout on the outputs of each RNN layer, except the last layer.

state.outputs boolean, optional, default=0 Whether to have the states as symbol outputs.

projection.size int or None, optional, default=None size of project size

lstm.state.clip.min double or None, optional, default=None Minimum clip value of LSTM states. This option must be used together with lstm_state_clip_max.

lstm.state.clip.max double or None, optional, default=None Maximum clip value of LSTM states. This option must be used together with lstm_state_clip_min.

lstm.state.clip.nan boolean, optional, default=0 Whether to stop NaN from propagating in state by clipping it to min/max. If clipping range is not specified, this option is ignored.

use.sequence.length boolean, optional, default=0 If set to true, this layer takes in an extra input parameter ‘sequence_length’ to specify variable length sequence

name string, optional Name of the resulting symbol.

**Details**

**Vanilla RNN**

Applies a single-gate recurrent layer to input X. Two kinds of activation function are supported: ReLU and Tanh.

With ReLU activation function:

.. math:: h_t = \text{relu}(W_ih * x_t + b_ih + W_hh * h_{(t-1)} + b_{hh})

With Tanh activation function:

.. math:: h_t = \tanh(W_ih * x_t + b_ih + W_hh * h_{(t-1)} + b_{hh})


**LSTM**


.. math:: \begin{array}{l}
i_t = \text{rsigmoid}(W_{ii} x_t + b_{ii} + W_{hi} h_{(t-1)} + b_{hi}) \\
f_t = \text{rsigmoid}(W_{if} x_t + b_{if} + W_{hf} h_{(t-1)} + b_{hf}) \\
g_t = \tanh(W_{ig} x_t + b_{ig} + W_{hc} h_{(t-1)} + b_{hg}) \\
o_t = \text{rsigmoid}(W_{io} x_t + b_{io} + W_{ho} h_{(t-1)} + b_{ho}) \\
c_t = f_t * c_{(t-1)} + i_t * g_t \\
h_t = o_t * \tanh(c_t) \\
r_t = W_{hr} h_t \end{array}


.. math:: \begin{array}{l}
i_t = \text{rsigmoid}(W_{ii} x_t + b_{ii} + W_{ri} r_{(t-1)} + b_{ri}) \\
f_t = \text{rsigmoid}(W_{if} x_t + b_{if} + W_{rf} r_{(t-1)} + b_{rf}) \\
g_t = \tanh(W_{ig} x_t + b_{ig} + W_{rc} r_{(t-1)} + b_{rg}) \\
o_t = \text{rsigmoid}(W_{io} x_t + b_{io} + W_{ro} r_{(t-1)} + b_{ro}) \\
c_t = f_t * c_{(t-1)} + i_t * g_t \\
h_t = o_t * \tanh(c_t) \\
r_t = W_{hr} h_t \end{array}
**GRU**


The definition of GRU here is slightly different from paper but compatible with CUDNN.

```
.. math:: \begin{array}{ll}
r_t = \mathrmsigmoid(W_{ir} x_t + b_{ir} + W_{hr} h_{(t-1)} + b_{hr}) \\
z_t = \mathrmsigmoid(W_{iz} x_t + b_{iz} + W_{hz} h_{(t-1)} + b_{hz}) \\
n_t = \tanh(W_{in} x_t + b_{in} + r_t \ast (W_{hn} h_{(t-1)} + b_{hn})) \\
h_t = (1 - z_t) \ast n_t + z_t \ast h_{(t-1)} \end{array}
```

Defined in src/operator/rnn.cc:L368

**Value**

out The result mx.symbol

---

```
mx.symbol.ROIPooling

ROIPooling: Performs region of interest (ROI) pooling on the input array.
```

**Description**

ROI pooling is a variant of a max pooling layer, in which the output size is fixed and region of interest is a parameter. Its purpose is to perform max pooling on the inputs of non-uniform sizes to obtain fixed-size feature maps. ROI pooling is a neural-net layer mostly used in training a ‘Fast R-CNN’ network for object detection.

**Usage**

`mx.symbol.ROIPooling(...)`

**Arguments**

- `data` NDArray-or-Symbol The input array to the pooling operator, a 4D Feature maps
- `rois` NDArray-or-Symbol Bounding box coordinates, a 2D array of [[batch_index, x1, y1, x2, y2]], where (x1, y1) and (x2, y2) are top left and bottom right corners of designated region of interest. ‘batch_index’ indicates the index of corresponding image in the input array
- `pooled.size` Shape(tuple), required ROI pooling output shape (h,w)
- `spatial.scale` float, required Ratio of input feature map height (or w) to raw image height (or w). Equals the reciprocal of total stride in convolutional layers
- `name` string, optional Name of the resulting symbol.
Details

This operator takes a 4D feature map as an input array and region proposals as ‘rois’, then it pools over sub-regions of input and produces a fixed-sized output array regardless of the ROI size.

To crop the feature map accordingly, you can resize the bounding box coordinates by changing the parameters ‘rois’ and ‘spatial_scale’.

The cropped feature maps are pooled by standard max pooling operation to a fixed size output indicated by a ‘pooled_size’ parameter. batch_size will change to the number of region bounding boxes after ‘ROIPooling’.

The size of each region of interest doesn’t have to be perfectly divisible by the number of pooling sections('pooled_size').

Example::

```python
[42., 43., 44., 45., 46., 47.]]]
// region of interest i.e. bounding box coordinates.
y = [0,0,0,4,4]
// returns array of shape (2,2) according to the given roi with max pooling. ROIPooling(x, y, (2,2),
1.0) = [[[ 14., 16.], [ 26., 28.]]]
// region of interest is changed due to the change in ‘spacial_scale’ parameter. ROIPooling(x, y,
(2,2), 0.7) = [[[ 7., 9.], [19., 21.]]]
```

Defined in src/operator/roi_pooling.cc:L225

Value

- `out` The result mx.symbol

---

**mx.symbol.round**

`round`:Returns element-wise rounded value to the nearest integer of the input.

---

Description

Example::

Usage

```python
mx.symbol.round(...)```

Arguments

- `data` NDArray-or-Symbol The input array.
- `name` string, optional Name of the resulting symbol.
mx.symbol.rsqrt

**Details**

\[
\text{round}([-1.5, 1.5, -1.9, 1.9, 2.1]) = [-2., 2., -2., 2., 2.]
\]

The storage type of “round” output depends upon the input storage type:
- round(default) = default - round(row_sparse) = row_sparse - round(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L777

**Value**

out The result mx.symbol

---

mx.symbol.rsqrt  
rsqrt: Returns element-wise inverse square-root value of the input.

**Description**

.. math:: \text{rsqrt}(x) = \frac{1}{\sqrt{x}}

**Usage**

mx.symbol.rsqrt(...)  

**Arguments**

data NDArray-or-Symbol The input array.

name string, optional Name of the resulting symbol.

**Details**

Example::

    rsqrt([4,9,16]) = [0.5, 0.33333334, 0.25]

The storage type of “rsqrt” output is always dense

Defined in src/operator/tensor/elemwise_unary_op_pow.cc:L221

**Value**

out The result mx.symbol
mx.symbol.sample_exponential

```
sample_exponential: Concurrent sampling from multiple exponential distributions with parameters lambda (rate).
```

### Description

The parameters of the distributions are provided as an input array. Let *s* be the shape of the input array, *n* be the dimension of *s*, *t* be the shape specified as the parameter of the operator, and *m* be the dimension of *t*. Then the output will be an *(n+m)*-dimensional array with shape *s x t*.

### Usage

```
mx.symbol.sample_exponential(...)  
```

### Arguments

- **lam**: NDArray-or-Symbol
  Lambda (rate) parameters of the distributions.

- **shape**: Shape(tuple), optional, default=\[
  \]
  Shape to be sampled from each random distribution.

- **dtype**: ‘None’, ‘float16’, ‘float32’, ‘float64’, optional, default=’None’
  DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).

- **name**: string, optional
  Name of the resulting symbol.

### Details

For any valid *n*-dimensional index *i* with respect to the input array, *output[i]* will be an *m*-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input value at index *i*. If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input array.

#### Examples::

- lam = [ 1.0, 8.5 ]
  - // Draw a single sample for each distribution
  - sample_exponential(lam) = [0.51837951, 0.09994757]
  - // Draw a vector containing two samples for each distribution
  - sample_exponential(lam, shape=(2)) = [[0.51837951, 0.19866663], [0.09994757, 0.50447971]]

Defined in `src/operator/random/multisample_op.cc:L284`

### Value

- **out**: The result mx.symbol
mx.symbol.sample_gamma

sample_gamma: Concurrent sampling from multiple gamma distributions with parameters *alpha* (shape) and *beta* (scale).

Description

The parameters of the distributions are provided as input arrays. Let *[s]* be the shape of the input arrays, *[n]* be the dimension of *[s]*, *[t]* be the shape specified as the parameter of the operator, and *[m]* be the dimension of *[t]*. Then the output will be a *([n]x[m]}-dimensional array with shape *[s]x[t]*.

Usage

```python
mx.symbol.sample_gamma(...)
```

Arguments

- **alpha**: NDArray-or-Symbol Alpha (shape) parameters of the distributions.
- **shape**: Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.
- **dtype**: ‘None’, ‘float16’, ‘float32’, ‘float64’.optional, default=’None’ DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **beta**: NDArray-or-Symbol Beta (scale) parameters of the distributions.
- **name**: string, optional Name of the resulting symbol.

Details

For any valid *[n]*-dimensional index *[i]* with respect to the input arrays, *[output][i]* will be an *[m]*-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input values at index *[i]*. If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input arrays.

Examples:
```
alpha = [ 0.0, 2.5 ]
beta = [ 1.0, 0.7 ]
// Draw a single sample for each distribution
sample_gamma(alpha, beta) = [ 0. , 2.25797319]
// Draw a vector containing two samples for each distribution
sample_gamma(alpha, beta, shape=(2)) = [[ 0. , 0. ], [ 2.25797319, 1.70734084]]
```

Defined in src/operator/random/multisample_op.cc:L282

Value

- **out**: The result mx.symbol
mx.symbol.sample_generalized_negative_binomial

Sample generalized negative binomial: Concurrent sampling from multiple generalized negative binomial distributions with parameters \( \mu \) (mean) and \( \alpha \) (dispersion).

Description

The parameters of the distributions are provided as input arrays. Let \( *[s]* \) be the shape of the input arrays, \( *n* \) be the dimension of \( *[s]*, *[t]* \) be the shape specified as the parameter of the operator, and \( *m* \) be the dimension of \( *[t]* \). Then the output will be a \( *(n+m)* \)-dimensional array with shape \( *[s]x[t]* \).

Usage

mx.symbol.sample_generalized_negative_binomial(...)  

Arguments

- **mu**: NDArray-or-Symbol Means of the distributions.  
- **shape**: Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.  
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).  
- **alpha**: NDArray-or-Symbol Alpha (dispersion) parameters of the distributions.  
- **name**: string, optional Name of the resulting symbol.

Details

For any valid \( n \)-dimensional index \( i \) with respect to the input arrays, \( output[i] \) will be an \( m \)-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input values at index \( i \). If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input arrays.

Samples will always be returned as a floating point data type.

Examples:

```python
mu = [ 2.0, 2.5 ] alpha = [ 1.0, 0.1 ]

# Draw a single sample for each distribution
sample_generalized_negative_binomial(mu, alpha) =
[ 0., 3.]

# Draw a vector containing two samples for each distribution
sample_generalized_negative_binomial(mu, alpha, shape=(2)) = [[ 0., 3.], [ 3., 1.]]
```

Defined in src/operator/random/multisample_op.cc:L293

Value

out The result mx.symbol
mx.symbol.sample_multinomial

sample_multinomial: Concurrent sampling from multiple multinomial distributions.

Description

*data* is an *n* dimensional array whose last dimension has length *k*, where *k* is the number of possible outcomes of each multinomial distribution. This operator will draw *shape* samples from each distribution. If shape is empty one sample will be drawn from each distribution.

Usage

mx.symbol.sample_multinomial(...)

Arguments

- **data**: NDArray-or-Symbol Distribution probabilities. Must sum to one on the last axis.
- **shape**: Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.
- **get.prob**: boolean, optional, default=0 Whether to also return the log probability of sampled result. This is usually used for differentiating through stochastic variables, e.g. in reinforcement learning.
- **dtype**: 'float16', 'float32', 'float64', 'int32', 'uint8', optional, default='int32' DType of the output in case this can't be inferred.
- **name**: string, optional Name of the resulting symbol.

Details

If *get_prob* is true, a second array containing log likelihood of the drawn samples will also be returned. This is usually used for reinforcement learning where you can provide reward as head gradient for this array to estimate gradient.

Note that the input distribution must be normalized, i.e. *data* must sum to 1 along its last axis.

Examples:

```python
probs = [[0, 0.1, 0.2, 0.3, 0.4], [0.4, 0.3, 0.2, 0.1, 0]]
// Draw a single sample for each distribution sample_multinomial(probs) = [3, 0]
// Draw a vector containing two samples for each distribution sample_multinomial(probs, shape=(2)) = [[4, 2], [0, 0]]
// requests log likelihood sample_multinomial(probs, get_prob=True) = [2, 1], [0.2, 0.3]
```

Value

out The result mx.symbol
mx.symbol.sample_negative_binomial

sample_negative_binomial: Concurrent sampling from multiple negative binomial distributions with parameters \(*k*\) (failure limit) and \(*p*\) (failure probability).

**Description**

The parameters of the distributions are provided as input arrays. Let \(*[s]*\) be the shape of the input arrays, \(*n*\) be the dimension of \(*[s]*\), \(*[t]*\) be the shape specified as the parameter of the operator, and \(*m*\) be the dimension of \(*[t]*\). Then the output will be a \((n+m)\)-dimensional array with shape \(*[s][t]*\).

**Usage**

mx.symbol.sample_negative_binomial(...)

**Arguments**

- **k**
  - NDArray-or-Symbol Limits of unsuccessful experiments.
- **shape**
  - Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.
- **dtype**
  - 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **p**
  - NDArray-or-Symbol Failure probabilities in each experiment.
- **name**
  - string, optional Name of the resulting symbol.

**Details**

For any valid \(*n*\)-dimensional index \(*i*\) with respect to the input arrays, \(*output[i]*\) will be an \(*m*\)-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input values at index \(*i*\). If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input arrays. Samples will always be returned as a floating point data type.

**Examples:**

```python
k = [ 20, 49 ] p = [ 0.4 , 0.77 ]
// Draw a single sample for each distribution sample_negative_binomial(k, p) = [ 15., 16.]
// Draw a vector containing two samples for each distribution sample_negative_binomial(k, p, shape=(2)) = [[ 15., 50.], [ 16., 12.]]
```

Defined in src/operator/random/multisample_op.cc:L289

**Value**

out The result mx.symbol
sample_normal: Concurrent sampling from multiple normal distributions with parameters *mu* (mean) and *sigma* (standard deviation).

**Description**

The parameters of the distributions are provided as input arrays. Let *[s]* be the shape of the input arrays, *[n]* be the dimension of *[s]*, *[t]* be the shape specified as the parameter of the operator, and *[m]* be the dimension of *[t]*. Then the output will be a *(n+m)*-dimensional array with shape *[s]x[t]*.

**Usage**

`mx.symbol.sample_normal(...)`

**Arguments**

- **mu**: NDArray-or-Symbol Means of the distributions.
- **shape**: Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **sigma**: NDArray-or-Symbol Standard deviations of the distributions.
- **name**: string, optional Name of the resulting symbol.

**Details**

For any valid *[n]*-dimensional index *[i]* with respect to the input arrays, *[output][i]* will be an *[m]*-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input values at index *[i]*. If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input arrays.

**Examples**:

```python
mu = [ 0.0, 2.5 ] sigma = [ 1.0, 3.7 ]
// Draw a single sample for each distribution
sample_normal(mu, sigma) = [-0.56410581, 0.95934606]
// Draw a vector containing two samples for each distribution
sample_normal(mu, sigma, shape=(2)) = [[-0.56410581, 0.2928229 ], [ 0.95934606, 4.48287058]]
```

**Value**

`out` The result `mx.symbol`
mx.symbol.sample_poisson

sample_poisson: Concurrent sampling from multiple Poisson distributions with parameters lambda (rate).

Description
The parameters of the distributions are provided as an input array. Let *[s]* be the shape of the input array, *n* be the dimension of *[s]*, *[t]* be the shape specified as the parameter of the operator, and *m* be the dimension of *[t]*. Then the output will be a *(n+m)*-dimensional array with shape *[s]x[t]*.

Usage
mx.symbol.sample_poisson(...)

Arguments
- **lam**: NDArray-or-Symbol Lambda (rate) parameters of the distributions.
- **shape**: Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.
- **dtype**: ‘None’, ‘float16’, ‘float32’, ‘float64’, optional, default=’None’ DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **name**: string, optional Name of the resulting symbol.

Details
For any valid *n*-dimensional index *i* with respect to the input array, *output[i]* will be an *m*-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input value at index *i*. If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input array.

Examples:
- lam = [1.0, 8.5]
  // Draw a single sample for each distribution sample_poisson(lam) = [0., 13.]
  // Draw a vector containing two samples for each distribution sample_poisson(lam, shape=(2)) = [[0., 4.], [13., 8.]]

Defined in src/operator/random/multisample_op.cc:L286

Value
out The result mx.symbol
**mx.symbol.sample_uniform**

Sample uniform: Concurrent sampling from multiple uniform distributions on the intervals given by *([low,high])*.

**Description**

The parameters of the distributions are provided as input arrays. Let *s* be the shape of the input arrays, *n* be the dimension of *s*, *t* be the shape specified as the parameter of the operator, and *m* be the dimension of *t*. Then the output will be an *s*×*t* dimensional array with shape *s*×*t*.

**Usage**

mx.symbol.sample_uniform(...)

**Arguments**

- **low**: NDArray-or-Symbol Lower bounds of the distributions.
- **shape**: Shape(tuple), optional, default=[] Shape to be sampled from each random distribution.
- **dtype**: 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- **high**: NDArray-or-Symbol Upper bounds of the distributions.
- **name**: string, optional Name of the resulting symbol.

**Details**

For any valid *n*-dimensional index *i* with respect to the input arrays, *output[i]* will be an *m*-dimensional array that holds randomly drawn samples from the distribution which is parameterized by the input values at index *i*. If the shape parameter of the operator is not set, then one sample will be drawn per distribution and the output array has the same shape as the input arrays.

Examples:

```python
low = [0.0, 2.5] high = [1.0, 3.7]
// Draw a single sample for each distribution sample_uniform(low, high) = [0.40451524, 3.18687344]
// Draw a vector containing two samples for each distribution sample_uniform(low, high, shape=(2)) = [[0.40451524, 0.18017688], [3.18687344, 3.68352246]]
```

Defined in src/operator/random/multisample_op.cc:L277

**Value**

out The result mx.symbol
**mx.symbol.save**

Save an `mx.symbol` object

**Usage**

```
mx.symbol.save(symbol, filename)
```

**Arguments**

- `symbol` the `mx.symbol` object
- `filename` the filename (including the path)

**Examples**

```python
data = mx.symbol.Variable('data')
mx.symbol.save(data, 'temp.symbol')
data2 = mx.symbol.load('temp.symbol')
```

---

**mx.symbol.scatter_nd**

`scatter_nd`: Scatters data into a new tensor according to indices.

**Description**

Given 'data' with shape `(Y_0, ..., Y_K-1, X_M, ..., X_N-1)` and indices with shape `(M, Y_0, ..., Y_K-1)`, the output will have shape `(X_0, X_1, ..., X_N-1)`, where `M <= N`. If `M == N`, data shape should simply be `(Y_0, ..., Y_K-1)`.

**Usage**

```
mx.symbol.scatter_nd(...)  
```

**Arguments**

- `data` NDArray-or-Symbol data
- `indices` NDArray-or-Symbol indices
- `shape` Shape(tuple), required Shape of output.
- `name` string, optional Name of the resulting symbol.
The elements in output is defined as follows:

\[
\text{output}[\text{indices}[0, y_0, ..., y_{K-1}], ..., \text{indices}[M-1, y_0, ..., y_{K-1}], \text{x}_M, ..., \text{x}_{N-1}] = \text{data}[y_0, ..., y_{K-1}, \text{x}_M, ..., \text{x}_{N-1}]
\]

all other entries in output are 0.

.. warning::

If the indices have duplicates, the result will be non-deterministic and the gradient of `scatter_nd` will not be correct!!

Examples::

\[
data = [2, 3, 0] \text{ indices } = [[1, 1, 0], [0, 1, 0]] \text{ shape } = (2, 2) \text{ scatter nd(data, indices, shape) } = [[0, 0], [2, 3]]
\]
\[
data = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]] \text{ indices } = [[0, 1], [1, 1]] \text{ shape } = (2, 2, 2) \text{ scatter nd(data, indices, shape) } = [[[0, 0], [0, 0]], [[1, 2], [3, 4]]], [[[0, 0], [0, 0]], [[5, 6], [7, 8]]]
\]

Value

\[
\text{out} \text{ The result mx.symbol}
\]
**mx.symbol.SequenceMask**

- **axis**
  int, optional, default='0' The sequence axis. Only values of 0 and 1 are currently supported.

- **name**
  string, optional Name of the resulting symbol.

**Details**

Parameter `sequence_length` is used to handle variable-length sequences. `sequence_length` should be an input array of positive ints of dimension [batch_size]. To use this parameter, set `use_sequence_length` to 'True', otherwise each example in the batch is assumed to have the max sequence length.

.. note:: Alternatively, you can also use `take` operator.

Example:

```py
x = [[[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]],

// returns last sequence when sequence_length parameter is not used
SequenceLast(x) = [[[19., 20., 21.], [22., 23., 24.], [25., 26., 27.]]

// sequence_length is used
SequenceLast(x, sequence_length=[1,1,1], use_sequence_length=True) = [[[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]]

// sequence_length is used
SequenceLast(x, sequence_length=[1,2,3], use_sequence_length=True) = [[[1., 2., 3.], [13., 14., 15.], [25., 26., 27.]]

Defined in src/operator/sequence_last.cc:L106

**Value**

out The result mx.symbol

---

**mx.symbol.SequenceMask**

Sequences: Sets all elements outside the sequence to a constant value.

---

**Description**

This function takes an n-dimensional input array of the form [max_sequence_length, batch_size, other_feature_dims] and returns an array of the same shape.

**Usage**

mx.symbol.SequenceMask(...)
Arguments

data NDArray-or-Symbol n-dimensional input array of the form [max_sequence_length, batch_size, other_feature_dims] where n>2

sequence.length NDArray-or-Symbol vector of sequence lengths of the form [batch_size]

use.sequence.length boolean, optional, default=0 If set to true, this layer takes in an extra input parameter ‘sequence_length’ to specify variable length sequence

value float, optional, default=0 The value to be used as a mask.

axis int, optional, default=’0’ The sequence axis. Only values of 0 and 1 are currently supported.

name string, optional Name of the resulting symbol.

Details

Parameter ‘sequence_length’ is used to handle variable-length sequences. ‘sequence_length’ should be an input array of positive ints of dimension [batch_size]. To use this parameter, set ‘use_sequence_length’ to ‘True’, otherwise each example in the batch is assumed to have the max sequence length and this operator works as the ‘identity’ operator.

Example::

x = [[[ 1., 2., 3.], [ 4., 5., 6.]],
[[ 7., 8., 9.], [10., 11., 12.]],
[[13., 14., 15.], [16., 17., 18.]]]

// Batch 1 B1 = [[[ 1., 2., 3.], [ 7., 8., 9.], [13., 14., 15.]]

// works as identity operator when sequence_length parameter is not used SequenceMask(x) = [[[ 1., 2., 3.], [ 4., 5., 6.]],
[[ 7., 8., 9.], [10., 11., 12.]],
[[13., 14., 15.], [16., 17., 18.]]]

// sequence_length [1,1] means 1 of each batch will be kept // and other rows are masked with default mask value = 0 SequenceMask(x, sequence_length=[1,1], use_sequence_length=True) = [[[ 1., 2., 3.], [ 4., 5., 6.]],
[[ 0., 0., 0.], [ 0., 0., 0.]],
[[ 0., 0., 0.], [ 0., 0., 0.]]]

// sequence_length [2,3] means 2 of batch B1 and 3 of batch B2 will be kept // and other rows are masked with value = 1 SequenceMask(x, sequence_length=[2,3], use_sequence_length=True, value=1) = [[[ 1., 2., 3.], [ 4., 5., 6.]],
[[ 7., 8., 9.], [10., 11., 12.]],
[[ 1., 1., 1.], [16., 17., 18.]]]

Defined in src/operator/sequence_mask.cc:L186
mx.symbol.SequenceReverse

SequenceReverse: Reverses the elements of each sequence.

Description
This function takes an n-dimensional input array of the form \([\text{max\_sequence\_length}, \text{batch\_size}, \text{other\_feature\_dims}]\) and returns an array of the same shape.

Usage
mx.symbol.SequenceReverse(...)

Arguments
- **data**: NDArray-or-Symbol n-dimensional input array of the form \([\text{max\_sequence\_length}, \text{batch\_size}, \text{other\_dims}]\) where \(n>2\)
- **sequence.length**: NDArray-or-Symbol vector of sequence lengths of the form \([\text{batch\_size}]\)
- **use.sequence.length**: boolean, optional, default=0 If set to true, this layer takes in an extra input parameter ‘sequence_length’ to specify variable length sequence
- **axis**: int, optional, default='0' The sequence axis. Only 0 is currently supported.
- **name**: string, optional Name of the resulting symbol.

Details
Parameter ‘sequence_length’ is used to handle variable-length sequences. ‘sequence_length’ should be an input array of positive ints of dimension \([\text{batch\_size}]\). To use this parameter, set ‘use_sequence_length’ to ‘True’, otherwise each example in the batch is assumed to have the max sequence length.

Example:
x = [[[ 1., 2., 3.], [ 4., 5., 6.]],
[[ 7., 8., 9.], [ 10., 11., 12.]],
[[ 13., 14., 15.], [ 16., 17., 18.]]]
// Batch 1 B1 = [[[ 1., 2., 3.], [ 7., 8., 9.], [ 13., 14., 15.]]
// returns reverse sequence when sequence_length parameter is not used SequenceReverse(x) = [[[ 13., 14., 15.], [ 16., 17., 18.]],
[[ 7., 8., 9.], [ 10., 11., 12.]],
[[ 1., 2., 3.], [ 4., 5., 6.]]]
mx.symbol.sgd_mom_update

// sequence_length [2,2] means 2 rows of // both batch B1 and B2 will be reversed. SequenceRe-
verse(x, sequence_length=[2,2], use_sequence_length=True) = [[[ 7., 8., 9.], [ 10., 11., 12.]],
[[ 1., 2., 3.], [ 4., 5., 6.]]
[[ 13., 14., 15.], [ 16., 17., 18.]]

// sequence_length [2,3] means 2 of batch B2 and 3 of batch B3 // will be reversed. SequenceRe-
verse(x, sequence_length=[2,3], use_sequence_length=True) = [[[ 7., 8., 9.], [ 16., 17., 18.]],
[[ 1., 2., 3.], [ 10., 11., 12.]],
[[ 13., 14, 15.], [ 4., 5., 6.]]
 Defined in src/operator/sequence_reverse.cc:L122

Value

out The result mx.symbol

mx.symbol.sgd_mom_update

sgd_mom_update: Momentum update function for Stochastic Gradient Descent (SGD) optimizer.

Description

Momentum update has better convergence rates on neural networks. Mathematically it looks like below:

Usage

mx.symbol.sgd_mom_update(...)

Arguments

weight NDArray-or-Symbol Weight
grad NDArray-or-Symbol Gradient
mom NDArray-or-Symbol Momentum
lr float, required Learning rate
momentum float, optional, default=0 The decay rate of momentum estimates at each epoch.
wds float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.
rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.
clip.gradient float, optional, default=1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).
lazy.update boolean, optional, default=1 If true, lazy updates are applied if gradient’s stype is row_sparse and both weight and momentum have the same stype
name string, optional Name of the resulting symbol.
Details

.. math::
\begin{align*}
    v_{t-1} &= \alpha \nabla J(W_{t-1}) \\
    v_t &= \gamma v_{t-1} - \alpha \nabla J(W_t) \\
    W_t &= W_{t-1} + v_t
\end{align*}

It updates the weights using::

\[ v = \text{momentum} \cdot v - \text{learning\_rate} \cdot \text{gradient} \]
weight += v

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.

However, if grad’s storage type is “row\_sparse”, “lazy\_update” is True and weight’s storage type is
the same as momentum’s storage type, only the row slices whose indices appear in grad.indices are
updated (for both weight and momentum)::

for row in gradient.indices:
weight[row] += v[row]

Defined in src/operator/optimizer_op.cc:L556

Value

out The result mx.symbol

mx.symbol.sgd_update
sgd_update: Update function for Stochastic Gradient Descent (SGD) optimizer.

Description

It updates the weights using::

Usage

mx.symbol.sgd_update(...)

Arguments

weight NDArray-or-Symbol Weight
grad NDArray-or-Symbol Gradient
lr float, required Learning rate
wd float, optional, default=0 Weight decay augments the objective function with a
regularization term that penalizes large weights. The penalty scales with the
square of the magnitude of each weight.
rescale\_grad float, optional, default=1 Rescale gradient to grad = rescale\_grad*grad.
clip\_gradient float, optional, default=-1 Clip gradient to the range of [-clip\_gradient, clip\_gradient]
If clip\_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip\_gradient), -clip\_gradient).
lazy\_update boolean, optional, default=1 If true, lazy updates are applied if gradient’s stype
is row\_sparse.
name string, optional Name of the resulting symbol.
Details

weight = weight - learning_rate * (gradient + wd * weight)

However, if gradient is of “row_sparse” storage type and “lazy_update” is True, only the row slices whose indices appear in grad.indices are updated:


Defined in src/operator/optimizer_op.cc:L515

Value

out The result mx.symbol

mx.symbol.shape_array

shape_array:Returns a 1D int64 array containing the shape of data.

Description

Example::

Usage

mx.symbol.shape_array(...)

Arguments

data NDArray-or-Symbol Input Array.
nname string, optional Name of the resulting symbol.

Details

shape_array([[1,2,3,4], [5,6,7,8]]) = [2,4]

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L573

Value

out The result mx.symbol
mx.symbol.shuffle  
shuffle: Randomly shuffle the elements.

Description
This shuffles the array along the first axis. The order of the elements in each subarray does not change. For example, if a 2D array is given, the order of the rows randomly changes, but the order of the elements in each row does not change.

Usage
mx.symbol.shuffle(...)

Arguments
- data: NDArray-or-Symbol Data to be shuffled.
- name: string, optional Name of the resulting symbol.

Value
out The result mx.symbol

mx.symbol.sigmoid  
sigmoid: Computes sigmoid of x element-wise.

Description
.. math:: y = 1 / (1 + \exp(-x))

Usage
mx.symbol.sigmoid(...)

Arguments
- data: NDArray-or-Symbol The input array.
- name: string, optional Name of the resulting symbol.

Details
The storage type of "sigmoid" output is always dense
Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L119

Value
out The result mx.symbol
mx.symbol.sign

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mx.symbol.sign</td>
<td>sign: Returns element-wise sign of the input.</td>
</tr>
</tbody>
</table>

**Description**

Example:
```
Usage

mx.symbol.sign(...)  # Usage example
```

**Arguments**

- **data**: NDArray-or-Symbol. The input array.
- **name**: string, optional. Name of the resulting symbol.

**Details**

```
sign([-2, 0, 3]) = [-1, 0, 1]
```

The storage type of “sign” output depends upon the input storage type:
- sign(default) = default
- sign(row_sparse) = row_sparse
- sign(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L758

**Value**

```
out The result mx.symbol
```

mx.symbol.signsgd_update

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mx.symbol.signsgd_update</td>
<td>signsgd_update: Update function for SignSGD optimizer.</td>
</tr>
</tbody>
</table>

**Description**

.. math::

```
Usage

mx.symbol.signsgd_update(...)  # Usage example
```

**Usage**

```
x = mx.sym.Variable('x')
```

```
x_grad = sign(x)
```

```
x_grad = signsgd_update(x_grad, x)
```
**Arguments**

- **weight**  
  NDArray-or-Symbol Weight

- **grad**  
  NDArray-or-Symbol Gradient

- **lr**  
  float, required Learning rate

- **wd**  
  float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.

- **rescale.grad**  
  float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.

- **clip.gradient**  
  float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient]  
  If clip_gradient <= 0, gradient clipping is turned off.  
  \[ \text{grad} = \max(\min(\text{grad}, \text{clip} \text{gradient}), -\text{clip} \text{gradient}). \]

- **name**  
  string, optional Name of the resulting symbol.

**Details**

\[ g_t = \nabla J(W_{t-1}) \backslash W_t = W_{t-1} - \eta_t \textsign(g_t) \]

It updates the weights using::

weight = weight - learning_rate * sign(gradient)

.. note:: - sparse ndarray not supported for this optimizer yet.

Defined in src/operator/optimizer_op.cc:L63

**Value**

- **out**  
The result mx.symbol

---

**mx.symbol.signum_update**

*:SIGN momentUM (Signum) optimizer.*

**Description**

.. math::

**Usage**

mx.symbol.signum_update(...)
Arguments

weight NDArray-or-Symbol Weight

grad NDArray-or-Symbol Gradient

mom NDArray-or-Symbol Momentum

lr float, required Learning rate

momentum float, optional, default=0 The decay rate of momentum estimates at each epoch.

wd float, optional, default=0 Weight decay augments the objective function with a regularization term that penalizes large weights. The penalty scales with the square of the magnitude of each weight.

rescale.grad float, optional, default=1 Rescale gradient to grad = rescale_grad*grad.

clip.gradient float, optional, default=-1 Clip gradient to the range of [-clip_gradient, clip_gradient] If clip_gradient <= 0, gradient clipping is turned off. grad = max(min(grad, clip_gradient), -clip_gradient).

wd.lh float, optional, default=0 The amount of weight decay that does not go into gradient/momentum calculations otherwise do weight decay algorithmically only.

name string, optional Name of the resulting symbol.

Details

\[ g_t = \nabla J(W_{t-1}) \]

\[ m_t = \beta m_{t-1} + (1 - \beta) g_t \]

\[ W_t = W_{t-1} - \eta_t \text{sign}(m_t) \]

It updates the weights using:

\[ \text{state} = \text{momentum} * \text{state} + (1-\text{momentum}) * \text{gradient} \]

\[ \text{weight} = \text{weight} - \text{learning_rate} * \text{sign}(	ext{state}) \]

Where the parameter “momentum” is the decay rate of momentum estimates at each epoch.

.. note:: - sparse ndarray not supported for this optimizer yet.

Defined in src/operator/optimizer_op.cc:L92

Value

out The result mx.symbol

mx.symbol.sin

\text{sin}:\text{Computes the element-wise sine of the input array.}

Description

The input should be in radians (:math:`2\pi` rad equals 360 degrees).

Usage

mx.symbol.sin(\ldots)
Arguments

data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.

Details

.. math:: \sin([0, \pi/4, \pi/2]) = [0, 0.707, 1]

The storage type of “sin” output depends upon the input storage type:
- sin(default) = default - sin(row_sparse) = row_sparse - sin(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L47

Value

out The result mx.symbol

mx.symbol.sinh

sinh: Returns the hyperbolic sine of the input array, computed element-wise.

Description

.. math:: \sinh(x) = 0.5\times(e^x - e^{-x})

Usage

mx.symbol.sinh(...)

Arguments

data NDArray-or-Symbol The input array.
name string, optional Name of the resulting symbol.

Details

The storage type of “sinh” output depends upon the input storage type:
- sinh(default) = default - sinh(row_sparse) = row_sparse - sinh(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_trig.cc:L371

Value

out The result mx.symbol
mx.symbol.size_array

size_array: Returns a 1D int64 array containing the size of data.

Description

Example::

Usage

mx.symbol.size_array(...) 

Arguments

data NDArray-or-Symbol Input Array.
name string, optional Name of the resulting symbol.

Details

size_array([[1,2,3,4], [5,6,7,8]]) = [8]

Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L624

Value

out The result mx.symbol
mx.symbol.slice  
slice: Slices a region of the array.  

... note:: “crop” is deprecated. Use “slice” instead. This function returns a sliced array between the indices given by ‘begin’ and ‘end’ with the corresponding ‘step’. For an input array of “shape=(d_0, d_1, ..., d_n-1)”, slice operation with “begin=(b_0, b_1...b_m-1)”, “end=(e_0, e_1, ..., e_m-1)”, and “step=(s_0, s_1, ..., s_m-1)”, where m <= n, results in an array with the shape “(|e_0-b_0|/|s_0|, ..., |e_m-1-b_m-1|/|s_m-1|, d_m, ..., d_{n-1})”. The resulting array’s *k*-th dimension contains elements from the *k*-th dimension of the input array starting from index “b_k” (inclusive) with step “s_k” until reaching “e_k” (exclusive). If the *k*-th elements are ‘None’ in the sequence of ‘begin’, ‘end’, and ‘step’, the following rule will be used to set default values. If ‘s_k’ is ‘None’, set ‘s_k=1’. If ‘s_k > 0’, set ‘b_k=d_k=0’, ‘e_k=d_k+1’, else, set ‘b_k=d_k-1’, ‘e_k=1’. The storage type of “slice” output depends on storage types of inputs - slice(csr) = csr - otherwise, “slice” generates output with default storage.  

... note:: When input data storage type is csr, it only supports step=(), or step=(None,), or step=(1,) to generate a csr output. For other step parameter values, it falls back to slicing a dense tensor. Example::

```
x = [[ 1.,  2.,  3.,  4.],
     [ 5.,  6.,  7.,  8.],
     [ 9., 10., 11., 12.]]
slice(x, begin=(0,1), end=(2,4)) = [[ 2.,  3.,  4.],
                                             [ 6.,  7.,  8.]]
slice(x, begin=(None, 0), end=(None, 3), step=(-1, 2)) = [[9.,
                                                             11.],
                                                            [5., 7.],
                                                            [1., 3.]]
```

Description

Defined in src/operator/tensor/matrix_op.cc:L482

Usage

```
mx.symbol.slice(...)  
```

Arguments

- **data**: NDArray-or-Symbol  
  Source input
- **begin**: Shape(tuple), required starting indices for the slice operation, supports negative indices.
- **end**: Shape(tuple), required ending indices for the slice operation, supports negative indices.
- **step**: Shape(tuple), optional, default=() step for the slice operation, supports negative values.
- **name**: string, optional Name of the resulting symbol.

Value

```
out The result mx.symbol
```
mx.symbol.SliceChannel

SliceChannel: Splits an array along a particular axis into multiple sub-arrays.

Description

.. note:: “SliceChannel” is deprecated. Use “split” instead.

Usage

mx.symbol.SliceChannel(…)

Arguments

data NDArray-or-Symbol The input
num.outputs int, required Number of splits. Note that this should evenly divide the length of the ‘axis’.
axis int, optional, default=’1’ Axis along which to split.
squeeze.axis boolean, optional, default=0 If true, Removes the axis with length 1 from the shapes of the output arrays. **Note** that setting 'squeeze_axis' to “true” removes axis with length 1 only along the ‘axis’ which it is split. Also `squeeze_axis` can be set to “true” only if “input.shape[axis] == num_outputs”.
name string, optional Name of the resulting symbol.

Details

**Note** that `num_outputs` should evenly divide the length of the axis along which to split the array.

Example::

```python
x = [[[ 1.] [ 2.]] [[ 3.] [ 4.]] [[ 5.] [ 6.]]] x.shape = (3, 2, 1)
y = split(x, axis=1, num_outputs=2) // a list of 2 arrays with shape (3, 1, 1) y = [[[ 1.] [[ 3.]] [[ 5.]]] [[ 2.]] [[ 4.]] [[ 6.]]]
y[0].shape = (3, 1, 1)
z = split(x, axis=0, num_outputs=3) // a list of 3 arrays with shape (1, 2, 1) z = [[[ 1.] [ 2.]] [[[ 3.] [ 4.]]] [[ 5.] [ 6.]]]
z[0].shape = (1, 2, 1)
```

‘squeeze_axis=1’ removes the axis with length 1 from the shapes of the output arrays. **Note** that setting 'squeeze_axis' to “1” removes axis with length 1 only along the ‘axis’ which it is split. Also `squeeze_axis` can be set to true only if “input.shape[axis] == num_outputs”.

Example::
```python
z = split(x, axis=0, num_outputs=3, squeeze_axis=1)  # a list of 3 arrays with shape (2, 1)
[[1.]
 [2.]]
[[3.]
 [4.]]
[[5.]
 [6.]]
```

Defined in src/operator/slice_channel.cc:L107

**Value**

out The result mx.symbol

---

**mx.symbol.slice_axis**  
```
slice_axis:Slices along a given axis. Returns an array slice along a given 'axis' starting from the 'begin' index to the 'end' index. Examples::
x = [[1., 2., 3., 4.], [5., 6., 7., 8.], [9., 10., 11., 12.]]
slice_axis(x, axis=0, begin=1, end=3) = [[5., 6., 7., 8.], [9., 10., 11., 12.]]
slice_axis(x, axis=1, begin=0, end=2) = [[1., 2.], [5., 6.], [9., 10.]]
slice_axis(x, axis=1, begin=-3, end=-1) = [[2., 3.], [6., 7.], [10., 11.]]
```

Defined in src/operator/tensor/matrix_op.cc:L571

**Description**

Defined in src/operator/tensor/matrix_op.cc:L571

**Usage**

mx.symbol.slice_axis(...)

**Arguments**

- **data** (NDArray-or-Symbol): Source input
- **axis** (int, required): Axis along which to be sliced, supports negative indexes.
- **begin** (int, required): The beginning index along the axis to be sliced, supports negative indexes.
- **end** (int or None, required): The ending index along the axis to be sliced, supports negative indexes.
- **name** (string, optional): Name of the resulting symbol.

**Value**

out The result mx.symbol
mx.symbol.slice_like

slice_like: Slices a region of the array like the shape of another array. This function is similar to "slice", however, the 'begin' are always '0' s and 'end' of specific axes are inferred from the second input 'shape_like'. Given the second 'shape_like' input of "shape=(d_0, d_1, ..., d_n-1)" , a "slice_like" operator with default empty 'axes', it performs the following operation: " out = slice(input, begin=(0, 0, ..., 0), end=(d_0, d_1, ..., d_n-1))". When 'axes' is not empty, it is used to specify which axes are being sliced. Given a 4-d input data, "slice_like" operator with "axes=(0, 2, -1)" will perform the following operation: " out = slice(input, begin=(0, 0, 0, 0), end=(d_0, None, d_2, d_3))". Note that it is allowed to have first and second input with different dimensions, however, you have to make sure the 'axes' are specified and not exceeding the dimension limits. For example, given 'input_1' with "shape=(2,3,4,5)" and 'input_2' with "shape=(1,2,3)" , it is not allowed to use: " out = slice_like(a, b)" because ndim of 'input_1' is 4, and ndim of 'input_2' is 3. The following is allowed in this situation: " out = slice_like(a, b, axes=(0, 2))" Example::

```
x = [[ 1., 2., 3., 4.], [ 5., 6., 7., 8.], [ 9., 10., 11., 12.]]
y = [[ 0., 0., 0.], [ 0., 0., 0.]]
slice_like(x, y) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
slice_like(x, y, axes=(0)) = [[ 1., 2., 3.] [ 5., 6., 7.]]
```

Description

Defined in src/operator/tensor/matrix_op.cc:L625

Usage

mx.symbol.slice_like(...)

Arguments

data : NDArray-or-Symbol
Source input

shape_like : NDArray-or-Symbol
Shape like input

axes : Shape(tuple), optional, default=[ ] List of axes on which input data will be sliced according to the corresponding size of the second input. By default will slice on all axes. Negative axes are supported.

name : string, optional
Name of the resulting symbol.

Value

out : The result mx.symbol
mx.symbol.smooth_l1

**smooth_l1:** Calculate Smooth L1 Loss(lhs, scalar) by summing

**Description**

.. math:: ...

**Usage**

mx.symbol.smooth_l1(...)

**Arguments**

- **data**: NDArray-or-Symbol source input
- **scalar**: float scalar input
- **name**: string, optional Name of the resulting symbol.

**Details**

\[ f(x) = \begin{cases} \frac{(\sigma x)^2}{2}, & \text{if } x < \frac{1}{\sigma^2} \\ |x| - 0.5 \frac{1}{\sigma^2}, & \text{otherwise} \end{cases} \]

where :math:`x` is an element of the tensor `lhs` and :math:`\sigma` is the scalar.

**Example**::

smooth_l1([1, 2, 3, 4]) = [0.5, 1.5, 2.5, 3.5] smooth_l1([1, 2, 3, 4], scalar=1) = [0.5, 1.5, 2.5, 3.5]

Defined in src/operator/tensor/elemwise_binary_scalar_op_extended.cc:L108

**Value**

- **out**: The result mx.symbol

---

mx.symbol.Softmax

**Softmax:** Computes the gradient of cross entropy loss with respect to softmax output.

**Description**

- This operator computes the gradient in two steps. The cross entropy loss does not actually need to be computed.

**Usage**

mx.symbol.Softmax(...)
Arguments

- **data**: NDArray-or-Symbol Input array.
- **label**: NDArray-or-Symbol Ground truth label.
- **grad.scale**: float, optional, default=1 Scales the gradient by a float factor.
- **ignore.label**: float, optional, default=-1 The instances whose 'labels' == 'ignore_label' will be ignored during backward, if 'use_ignore' is set to "true".
- **multi.output**: boolean, optional, default=0 If set to "true", the softmax function will be computed along axis “1”. This is applied when the shape of input array differs from the shape of label array.
- **use.ignore**: boolean, optional, default=0 If set to "true", the ‘ignore_label’ value will not contribute to the backward gradient.
- **preserve.shape**: boolean, optional, default=0 If set to "true", the softmax function will be computed along the last axis (“-1”).
- **normalization**: 'batch', 'null', 'valid', optional, default='null' Normalizes the gradient.
- **out.grad**: boolean, optional, default=0 Multiplies gradient with output gradient element-wise.
- **smooth.alpha**: float, optional, default=0 Constant for computing a label smoothed version of cross-entropy for the backwards pass. This constant gets subtracted from the one-hot encoding of the gold label and distributed uniformly to all other labels.
- **name**: string, optional Name of the resulting symbol.

Details

- Applies softmax function on the input array. - Computes and returns the gradient of cross entropy loss w.r.t. the softmax output.
- The softmax function, cross entropy loss and gradient is given by:
  - **Softmax Function**:
    \[
    \text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}
    \]
  - **Cross Entropy Function**:
    \[
    \text{CE}(\text{label}, \text{output}) = - \sum_i \text{label}_i \log(\text{output}_i)
    \]
  - The gradient of cross entropy loss w.r.t softmax output:
    \[
    \text{gradient} = \text{output} - \text{label}
    \]
- During forward propagation, the softmax function is computed for each instance in the input array. For general \(N\)-D input arrays with shape \(d_1, d_2, ..., d_n\). The size is \(s = d_1 \cdot d_2 \cdot \ldots \cdot d_n\). We can use the parameters ‘preserve_shape’ and ‘multi_output’ to specify the way to compute softmax:
  - By default, ‘preserve_shape’ is “false”. This operator will reshape the input array into a 2-D array with shape \(d_1, \frac{d_1 \cdot d_2 \cdot \ldots \cdot d_n}{s}\) and then compute the softmax function for each row in the reshaped array, and afterwards reshape it back to the original shape \(d_1, d_2, ..., d_n\).
  - If ‘preserve_shape’ is “true”, the softmax function will be computed along the last axis (‘axis’ = “-1”).
  - If ‘multi_output’ is “true”, the softmax function will be computed along the second axis (‘axis’ = “1”).
- During backward propagation, the gradient of cross-entropy loss w.r.t softmax output array is computed. The provided label can be a one-hot label array or a probability label array.
- If the parameter ‘use_ignore’ is “true”, ‘ignore_label’ can specify input instances with a particular label to be ignored during backward propagation. **This has no effect when softmax ‘output’ has same shape as ‘label’**.

Example:

```python
data = [[1,2,3,4],[2,2,2,2],[3,3,3,3],[4,4,4,4]] label = [1,0,2,3] ignore_label = 1 SoftmaxOutput(data=data, label = label, multi_output=true, use_ignore=true, ignore_label=ignore_label)  ## forward softmax output 
[[0.0320586 0.08714432 0.23688284 0.64391428] [ 0.25 0.25 0.25 0.25 ] [ 0.25 0.25 0.25 0.25 ] [ 0.25 0.25 0.25 0.25 ] ]
## backward gradient output [[ 0.25 0.25 0.25 0.25 ] [-0.75 0.25 0.25 0.25] [ 0.25 0.25 -0.75 0.25] [ 0.25 0.25 0.25 -0.75]]
```  

- The parameter ‘grad_scale’ can be used to rescale the gradient, which is often used to give each loss function different weights.
- This operator also supports various ways to normalize the gradient by ‘normalization’, The ‘normalization’ is applied if softmax output has different shape than the labels. The ‘normalization’ mode can be set to the followings:
  - “null”: do nothing.  - “batch”: divide the gradient by the batch size.  - “valid”: divide the gradient by the number of instances which are not ignored.

Defined in src/operator/softmax_output.cc:L231

**Value**

out The result mx.symbol

**Description**

The resulting array contains elements in the range (0,1) and the elements along the given axis sum up to 1.

**Usage**

mx.symbol.softmax(...)

**Arguments**

data NDArray-or-Symbol The input array.
length NDArray-or-Symbol The length array.
axis int, optional, default=-1‘ The axis along which to compute softmax.
temperature double or None, optional, default=None Temperature parameter in softmax
mx.symbol.SoftmaxActivation

### Arguments

- **data**: NDArray-or-Symbol The input array.
- **mode**: 'channel', 'instance', optional, default='instance' Specifies how to compute the softmax. If set to "instance", it computes softmax for each instance. If set to "channel", it computes cross channel softmax for each position of each instance.
- **name**: string, optional Name of the resulting symbol.

### Description

*SoftmaxActivation*: Applies softmax activation to input. This is intended for internal layers.

### Usage

mx.symbol.SoftmaxActivation(...)

### Details

\[
\text{softmax}(\mathbf{z}/t)_j = \frac{e^{z_j}/t}{\sum_{k=1}^{K} e^{z_k}/t}
\]

for \( j = 1, ..., K \)

t is the temperature parameter in softmax function. By default, t equals 1.0

Example:

\[
x = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}
\]

\[
\text{softmax}(x, axis=0) = \begin{bmatrix} 0.33333334, 0.33333334, 0.33333334 \\ 0.33333334, 0.33333334, 0.33333334 \end{bmatrix}
\]

\[
\text{softmax}(x, axis=1) = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix}
\]

Defined in src/operator/nn/softmax.cc:L134

### Value

out The result mx.symbol
Details
This operator has been deprecated, please use ‘softmax’.

If ‘mode’ = “instance”, this operator will compute a softmax for each instance in the batch. This is the default mode.

If ‘mode’ = “channel”, this operator will compute a k-class softmax at each position of each instance, where ‘k’ = “num_channel”. This mode can only be used when the input array has at least 3 dimensions. This can be used for ‘fully convolutional network’, ‘image segmentation’, etc.

Example::
```python
gb input_array = mx.nd.array([[3., 0.5, -0.5, 2., 7.],
[2., -4., 7., 3., 0.2]])
gb softmax_act =
mx.nd.SoftmaxActivation(input_array) 
print softmax_act.asnumpy()
```
```text
[[ 1.78322066e-02 1.46375655e-03 5.38485940e-04 6.56010211e-03 9.73605454e-01]
[ 6.56221947e-03 5.95310994e-02 9.73919690e-01 1.7837962e-02 1.08472735e-03]]
```
Defined in src/operator/nn/softmax_activation.cc:L59

Value
out The result mx.symbol

mx.symbol.SoftmaxOutput

SoftmaxOutput: Computes the gradient of cross entropy loss with respect to softmax output.

Description
- This operator computes the gradient in two steps. The cross entropy loss does not actually need to be computed.

Usage
mx.symbol.SoftmaxOutput(...)

Arguments
- data: NDArray-or-Symbol Input array.
- label: NDArray-or-Symbol Ground truth label.
- grad.scale: float, optional, default=1 Scales the gradient by a float factor.
- ignore.label: float, optional, default=-1 The instances whose ‘labels’ == ‘ignore_label’ will be ignored during backward, if ‘use_ignore’ is set to “true”).
- multi.output: boolean, optional, default=0 If set to “true”, the softmax function will be computed along axis “1“. This is applied when the shape of input array differs from the shape of label array.
- use.ignore: boolean, optional, default=0 If set to “true”, the ‘ignore_label’ value will not contribute to the backward gradient.
mx.symbol.SoftmaxOutput

- Applies softmax function on the input array. - Computes and returns the gradient of cross entropy loss w.r.t. the softmax output.

- The softmax function, cross entropy loss and gradient is given by:

- Softmax Function:

  .. math:: \text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}

- Cross Entropy Function:

  .. math:: \text{CE}(\text{label}, \text{output}) = -\sum_i \text{label}_i \log(\text{output}_i)

- The gradient of cross entropy loss w.r.t softmax output:

  .. math:: \text{gradient} = \text{output} - \text{label}

- During forward propagation, the softmax function is computed for each instance in the input array. For general \(N\)-D input arrays with shape \(d_1, d_2, ..., d_n\). The size is \(s = d_1 \cdot d_2 \cdot \ldots \cdot d_n\). We can use the parameters 'preserve_shape' and 'multi_output' to specify the way to compute softmax:

  - By default, 'preserve_shape' is “false”. This operator will reshape the input array into a 2-D array with shape \(d_1', d_2', ..., d_n'\) and then compute the softmax function for each row in the reshaped array, and afterwards reshape it back to the original shape \(d_1, d_2, ..., d_n\).

  - If 'preserve_shape' is “true”, the softmax function will be computed along the last axis (‘axis’ = “-1”). - If 'multi_output' is “true”, the softmax function will be computed along the second axis (‘axis’ = “1”).

- During backward propagation, the gradient of cross-entropy loss w.r.t softmax output array is computed. The provided label can be a one-hot label array or a probability label array.

- If the parameter ‘use_ignore’ is "true", ‘ignore_label’ can specify input instances with a particular label to be ignored during backward propagation. **This has no effect when softmax ‘output’ has same shape as ‘label’**.

Example::

data = [[1,2,3,4],[2,2,2,2],[3,3,3,3],[4,4,4,4]] label = [1,0,2,3] ignore_label = 1 SoftmaxOutput(data=data, label = label, multi_output=true, use_ignore=true, ignore_label=ignore_label) ## forward softmax output [[0.0320586 0.08714432 0.23688284 0.64391428] [0.25 0.25 0.25 0.25] [0.25 0.25 0.25 0.25] [0.25 0.25 -0.75 0.25]] ## backward gradient output [[0. 0. 0. 0.] [-0.75 0.25 0.25] [0.25 0.25 -0.75 0.25] [0.25 0.25 -0.75]] ## notice that the first row is all 0 because label[0] is 1, which is equal to ignore_label.
mx.symbol.softmax_cross_entropy

- The parameter 'grad_scale' can be used to rescale the gradient, which is often used to give each loss function different weights.
- This operator also supports various ways to normalize the gradient by 'normalization'. The 'normalization' is applied if softmax output has different shape than the labels. The 'normalization' mode can be set to the followings:
  - ‘null’: do nothing.
  - ‘batch’: divide the gradient by the batch size.
  - ‘valid’: divide the gradient by the number of instances which are not ignored.
Defined in src/operator/softmax_output.cc:L231

Value

out The result mx.symbol

Description

- This operator computes the cross entropy in two steps:
  - Applies softmax function on the input array.
  - Computes and returns the cross entropy loss between the softmax output and the labels.

Usage

mx.symbol.softmax_cross_entropy(...)

Arguments

data NDArray-or-Symbol Input data
label NDArray-or-Symbol Input label
name string, optional Name of the resulting symbol.

Details

- The softmax function and cross entropy loss is given by:

  Softmax Function:
  .. math:: \text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}

  Cross Entropy Function:
  .. math:: \text{CE}(\text{label}, \text{output}) = - \sum_i \text{label}_i \log(\text{output}_i)

Example:

\[
x = [[1, 2, 3], [11, 7, 5]]
\]
\[
label = [2, 0]
\]
\[
\text{softmax}(x) = [[0.09003057, 0.24472848, 0.66524094], [0.97962922, 0.01794253, 0.00242826]]
\]
\[
\text{softmax} \_\text{cross} \_\text{entropy}(\text{data}, \text{label}) = - \log(0.66524084) - \log(0.97962922) = 0.4281871
\]
Defined in src/operator/loss_binary_op.cc:L59
**mx.symbol.softmin**

**Value**

out The result mx.symbol

---

mx.symbol.softmin  
**softmin**: Applies the softmin function.

---

**Description**

The resulting array contains elements in the range (0,1) and the elements along the given axis sum up to 1.

**Usage**

mx.symbol.softmin(...)

**Arguments**

- **data**: NDArray-or-Symbol The input array.
- **axis**: int, optional, default='-1' The axis along which to compute softmax.
- **temperature**: double or None, optional, default=None Temperature parameter in softmax
- **dtype**: None, 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to the same as input’s dtype if not defined (dtype=None).
- **use.length**: boolean or None, optional, default=0 Whether to use the length input as a mask over the data input.
- **name**: string, optional Name of the resulting symbol.

**Details**

.. math:: \text{softmin}(\mathbfz/t)_j = \frac{e^{-z_j/t}}{\sum_{k=1}^{K} e^{-z_k/t}}

for :math:`j = 1, ..., K`

\(t\) is the temperature parameter in softmax function. By default, \(t\) equals 1.0

Example::

    x = [[ 1.  2.  3.], [ 3.  2.  1.]]
    softmax(x, axis=0) = [[ 0.88079703, 0.5, 0.11920292], [ 0.11920292, 0.5, 0.88079703]]
    softmax(x, axis=1) = [[ 0.66524094, 0.24472848, 0.09003057], [ 0.09003057, 0.24472848, 0.66524094]]

Defined in src/operator/nn/softmin.cc:L57

**Value**

out The result mx.symbol
### mx.symbol.softsign

**softsign:** Computes softsign of \( x \) element-wise.

**Description**

.. math:: y = x / (1 + \text{abs}(x))

**Usage**

```python
mx.symbol.softsign(...)
```

**Arguments**

- `data` : NDArray-or-Symbol The input array.
- `name` : string, optional Name of the resulting symbol.

**Details**

The storage type of "softsign" output is always dense

Defined in `src/operator/tensor/elemwise_unary_op_basic.cc:L191`

**Value**

- `out` : The result `mx.symbol`

### mx.symbol.sort

**sort:** Returns a sorted copy of an input array along the given axis.

**Description**

Examples::

**Usage**

```python
mx.symbol.sort(...)
```

**Arguments**

- `data` : NDArray-or-Symbol The input array
- `axis` : int or None, optional, default=-1 Axis along which to choose sort the input tensor. If not given, the flattened array is used. Default is -1.
- `is.ascend` : boolean, optional, default=1 Whether to sort in ascending or descending order.
- `name` : string, optional Name of the resulting symbol.
Details

\[
\begin{align*}
\text{x} &= \begin{bmatrix}
1 & 4 \\ 3 & 1
\end{bmatrix} \\
// \text{sorts along the last axis} & \text{sort}(x) = \begin{bmatrix}
1. & 4. \\ 1. & 3.
\end{bmatrix} \\
// \text{flattens and then sorts} & \text{sort}(x, \text{axis=None}) = \begin{bmatrix}
1. , 1. , 3. , 4. 
\end{bmatrix} \\
// \text{sorts along the first axis} & \text{sort}(x, \text{axis=0}) = \begin{bmatrix}
1. , 1. , 3. , 4.
\end{bmatrix} \\
// \text{in a ascend order} & \text{sort}(x, \text{is_ascend=0}) = \begin{bmatrix}
4. , 1. , 3. , 1.
\end{bmatrix}
\end{align*}
\]

Defined in src/operator/tensor/ordering_op.cc:L133

Value

- out: The result mx.symbol

mx.symbol.space_to_depth

\[\text{space_to_depth}: \text{Rearranges(permutes) blocks of spatial data into depth. Similar to ONNX SpaceToDepth operator: https://github.com/onnx/onnx/blob/master/docs/Operators.md#SpaceToDepth} \]

The output is a new tensor where the values from height and width dimension are moved to the depth dimension. The reverse of this operation is "depth_to_space". .. math:: \begin{align*}
\text{x}' &= \text{reshape}(x, [N, C, H / block\_size, block\_size, W / block\_size, block\_size]) \\
\text{x}'' &= \text{transpose}(\text{x}' , [0, 3, 5, 1, 2, 4]) \\
y &= \text{reshape}(\text{x}'' , [N, C * (block\_size ^ 2), H / block\_size, W / block\_size])
\end{align*}

where \(x\)' is an input tensor with default layout as \(\text{[N, C, H, W]}\): \(\text{[batch, channels, height, width]}\) and \(y\)' is the output tensor of layout \(\text{[N, C * (block\_size ^ 2), H / block\_size, W / block\_size]}\)

Example:

\[
\begin{align*}
x &= \begin{bmatrix}
\begin{bmatrix}
0, 6, 1, 7, 2, 8 \\
12, 18, 13, 19, 14, 20
\end{bmatrix} \\
\begin{bmatrix}
3, 9, 4, 10, 5, 11 \\
15, 21, 16, 22, 17, 23
\end{bmatrix}
\end{bmatrix} \\
\text{space_to_depth}(x, 2) &= \begin{bmatrix}
\begin{bmatrix}
0, 1, 2 \\
3, 4, 5
\end{bmatrix} \\
\begin{bmatrix}
6, 7, 8 \\
9, 10, 11
\end{bmatrix} \\
\begin{bmatrix}
12, 13, 14 \\
15, 16, 17
\end{bmatrix} \\
\begin{bmatrix}
18, 19, 20 \\
21, 22, 23
\end{bmatrix}
\end{bmatrix}
\end{align*}
\]

Description

Defined in src/operator/tensor/matrix_op.cc:L1019

Usage

\[\text{mx.symbol.space_to_depth(...)}\]

Arguments

- data: NDArray-or-Symbol Input ndarray
- block.size: int, required Blocks of [block_size, block_size] are moved
- name: string, optional Name of the resulting symbol.
**Value**

out The result mx.symbol

---

**SpatialTransformer**

*Applies a spatial transformer to input feature map.*

**Description**

SpatialTransformer: Applies a spatial transformer to input feature map.

**Usage**

`mx.symbol.SpatialTransformer(...)`

**Arguments**

- **data**: NDArray-or-Symbol Input data to the SpatialTransformerOp.
- **loc**: NDArray-or-Symbol localisation net, the output dim should be 6 when transform_type is affine. You should initialize the weight and bias with identity transform.
- **target.shape**: Shape(tuple), optional, default=[0,0] output shape(h, w) of spatial transformer: (y, x)
- **transform.type**: 'affine', required transformation type
- **sampler.type**: 'bilinear', required sampling type
- **cudnn.off**: boolean or None, optional, default=None whether to turn cudnn off
- **name**: string, optional Name of the resulting symbol.

**Value**

out The result mx.symbol
mx.symbol.split

split:Splits an array along a particular axis into multiple sub-arrays.

Description

.. note:: “SliceChannel“ is deprecated. Use “split“ instead.

Usage

.. code-block::

   mx.symbol.split(...)

Arguments

- **data**
  - NDArray-or-Symbol The input
- **num.outputs**
  - int, required Number of splits. Note that this should evenly divide the length of the 'axis'.
- **axis**
  - int, optional, default='1' Axis along which to split.
- **squeeze.axis**
  - boolean, optional, default=0 If true, Removes the axis with length 1 from the shapes of the output arrays. **Note** that setting 'squeeze_axis' to 'true' removes axis with length 1 only along the 'axis' which it is split. Also 'squeeze_axis' can be set to “true” only if “input.shape[axis] == num_outputs”.
- **name**
  - string, optional Name of the resulting symbol.

Details

**Note** that 'num_outputs' should evenly divide the length of the axis along which to split the array.

Example::

   x = [[[ 1.] [ 2.]] [[ 3.] [ 4.]] [[ 5.] [ 6.]]] x.shape = (3, 2, 1)
   y = split(x, axis=1, num_outputs=2) // a list of 2 arrays with shape (3, 1, 1) y = [[[ 1.]] [[ 3.]] [[ 5.]]]
      [[[ 2.]] [[ 4.]] [[ 6.]]]
   y[0].shape = (3, 1, 1)
   z = split(x, axis=0, num_outputs=3) // a list of 3 arrays with shape (1, 2, 1) z = [[[ 1.]] [[ 2.]]]
      [[[ 3.]] [[ 4.]]]
      [[[ 5.]] [[ 6.]]]
   z[0].shape = (1, 2, 1)

'squeeze_axis=1' removes the axis with length 1 from the shapes of the output arrays. **Note** that setting 'squeeze_axis' to '1' removes axis with length 1 only along the 'axis' which it is split. Also 'squeeze_axis' can be set to true only if “input.shape[axis] == num_outputs”.

Example::

   z = split(x, axis=0, num_outputs=3, squeeze_axis=1) // a list of 3 arrays with shape (2, 1) z = [[ 1.]]
      [[ 2.]]
Value

out The result mx.symbol

mx.symbol.sqrt sqrt:Returns element-wise square-root value of the input.

Description

.. math:: \text{sqrt}(x) = \sqrt{x}

Usage

mx.symbol.sqrt(...)

Arguments

data NDArray-or-Symbol The input array.

name string, optional Name of the resulting symbol.

Details

Example::

  \text{sqrt}([4, 9, 16]) = [2, 3, 4]

The storage type of "sqrt" output depends upon the input storage type:

- \text{sqrt}(default) = default - \text{sqrt}(\text{row_sparse}) = \text{row_sparse} - \text{sqrt}(\text{csr}) = \text{csr}

Defined in src/operator/tensor/elemwise_unary_op_pow.cc:L170

Value

out The result mx.symbol
**mx.symbol.square**  

square: Returns element-wise squared value of the input.

**Description**

.. math:: \text{square}(x) = x^2

**Usage**

mx.symbol.square(...)

**Arguments**

- **data**: NDArray-or-Symbol The input array.
- **name**: string, optional Name of the resulting symbol.

**Details**

Example::

    square([2, 3, 4]) = [4, 9, 16]

The storage type of “square” output depends upon the input storage type:

- square(default) = default
- square(row_sparse) = row_sparse
- square(csr) = csr

Defined in src/operator/tensor/elemwise_unary_op_pow.cc:L119

**Value**

out The result mx.symbol

---

**mx.symbol.squeeze**  

squeeze: Remove single-dimensional entries from the shape of an array. Same behavior of defining the output tensor shape as numpy.squeeze for the most of cases. See the following note for exception. Examples::

    data = [[0], [1], [2]]
    squeeze(data) = [0, 1, 2]
    squeeze(data, axis=0) = [[0], [1], [2]]
    squeeze(data, axis=2) = [[0, 1, 2]]
    squeeze(data, axis=(0, 2)) = [0, 1, 2] .. Note:: The output of this operator will keep at least one dimension not removed. For example, squeeze([[[4]]]) = [4], while in numpy.squeeze, the output will become a scalar.
Description

squeeze: Remove single-dimensional entries from the shape of an array. Same behavior of defining the output tensor shape as numpy.squeeze for the most of cases. See the following note for exception. Examples:: data = [[[0], [1], [2]]] squeeze(data) = [0, 1, 2] squeeze(data, axis=0) = [[0], [1], [2]] squeeze(data, axis=2) = [[0, 1, 2]] squeeze(data, axis=(0, 2)) = [0, 1, 2] .. Note:: The output of this operator will keep at least one dimension not removed. For example, squeeze([[[4]]]) = [4], while in numpy.squeeze, the output will become a scalar.

Usage

mx.symbol.squeeze(...)

Arguments

data  NDArray-or-Symbol data to squeeze
axis  Shape or None, optional, default=None Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.
name  string, optional Name of the resulting symbol.

Value

out The result mx.symbol

Description

stack: Join a sequence of arrays along a new axis. The axis parameter specifies the index of the new axis in the dimensions of the result. For example, if axis=0 it will be the first dimension and if axis=-1 it will be the last dimension. Examples:: x = [1, 2] y = [3, 4] stack(x, y) = [[1, 2], [3, 4]] stack(x, y, axis=1) = [[1, 3], [2, 4]]

Usage

mx.symbol.stack(...)
Arguments

data NDArray-or-Symbol[] List of arrays to stack
axis int, optional, default='0' The axis in the result array along which the input arrays are stacked.
num_args int, required Number of inputs to be stacked.
name string, optional Name of the resulting symbol.

Value

out The result mx.symbol

Description

Stops the accumulated gradient of the inputs from flowing through this operator in the backward direction. In other words, this operator prevents the contribution of its inputs to be taken into account for computing gradients.

Usage

mx.symbol.stop_gradient(...)
**mx.symbol.sum**

**sum:** Computes the sum of array elements over given axes.

### Description

Note:

### Usage

```python
mx.symbol.sum(...)```

### Arguments

- **data**: NDArray-or-Symbol The input
- **axis**: Shape or None, optional, default=None The axis or axes along which to perform the reduction.
  - The default, `axis=()`, will compute over all elements into a scalar array with shape `(1,)`.
  - If `axis` is int, a reduction is performed on a particular axis.
  - If `axis` is a tuple of ints, a reduction is performed on all the axes specified in the tuple.
  - If `exclude` is true, reduction will be performed on the axes that are NOT in axis instead.
  - Negative values means indexing from right to left.
- **keepdims**: boolean, optional, default=0 If this is set to `True`, the reduced axes are left in the result as dimension with size one.
- **exclude**: boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.
- **name**: string, optional Name of the resulting symbol.

### Details

`sum` and `sum_axis` are equivalent. For ndarray of csr storage type summation along axis 0 and axis 1 is supported. Setting keepdims or exclude to True will cause a fallback to dense operator.

Example:

```python
data = [[[1, 2], [2, 3], [1, 3]], [[1, 4], [4, 3], [5, 2]], [[7, 1], [7, 2], [7, 3]]]
sum(data, axis=1) [[ 4.  8.] [10.  9.] [21.  6.]]
sum(data, axis=[1,2]) [12. 19. 27.]
data = [[1, 2, 0], [3, 0, 1], [4, 1, 0]]
csr = cast_storage(data, 'csr')
sum(csr, axis=0) [ 8.  3.  1.]
sum(csr, axis=1) [ 3.  4.  5.]
```

Defined in `src/operator/tensor/broadcast_reduce_sum_value.cc:L67`
mx.symbol.sum_axis  

**Value**

out  The result mx.symbol

**Description**

.. Note::

**Usage**

mx.symbol.sum_axis(...)

**Arguments**

data  
NDArray-or-Symbol The input

axis  
Shape or None, optional, default=None The axis or axes along which to perform the reduction.  
The default, 'axis=()', will compute over all elements into a scalar array with shape '(1,)'.  
If 'axis' is int, a reduction is performed on a particular axis.  
If 'axis' is a tuple of ints, a reduction is performed on all the axes specified in the tuple.  
If 'exclude' is true, reduction will be performed on the axes that are NOT in axis instead.  
Negative values means indexing from right to left.

keepdims  
boolean, optional, default=0 If this is set to 'True', the reduced axes are left in the result as dimension with size one.

eclude  
boolean, optional, default=0 Whether to perform reduction on axis that are NOT in axis instead.

name  
string, optional Name of the resulting symbol.

**Details**

‘sum’ and ‘sum_axis’ are equivalent. For ndarray of csr storage type summation along axis 0 and axis 1 is supported. Setting keepdims or exclude to True will cause a fallback to dense operator.

Example::

data = [[[[1, 2], [2, 3], [1, 3]], [[1, 4], [4, 3], [5, 2]], [[7, 1], [7, 2], [7, 3]]]
sum(data, axis=1) [[4, 8] [10, 9] [21, 6]]
sum(data, axis=[1,2]) [12, 19, 27]
data = [[[1, 2, 0], [3, 0, 1], [4, 1, 0]]
csr = cast_storage(data, 'csr')
mx.symbol.SVMOutput

sum(csr, axis=0) [ 8. 3. 1.]
sum(csr, axis=1) [ 3. 4. 5.]

Defined in src/operator/tensor/broadcast_reduce_sum_value.cc:L67

Value

out The result mx.symbol

mx.symbol.SVMOutput SVMOutput: Computes support vector machine based transformation of the input.

Description

This tutorial demonstrates using SVM as output layer for classification instead of softmax: https://github.com/dmlc/mxnet/tree/master/example/svm_mnist.

Usage

mx.symbol.SVMOutput(...)

Arguments

data NDArray-or-Symbol Input data for SVM transformation.
label NDArray-or-Symbol Class label for the input data.
margin float, optional, default=1 The loss function penalizes outputs that lie outside this margin. Default margin is 1.
regularization.coefficient float, optional, default=1 Regularization parameter for the SVM. This balances the tradeoff between coefficient size and error.
use.linear boolean, optional, default=0 Whether to use L1-SVM objective. L2-SVM objective is used by default.
name string, optional Name of the resulting symbol.

Value

out The result mx.symbol
mx.symbol.swapaxes

swapaxes: Interchanges two axes of an array.

Description

Examples::

Usage

mx.symbol.swapaxes(...)  

Arguments

data NDArray-or-Symbol Input array.
dim1 int, optional, default='0' the first axis to be swapped.
dim2 int, optional, default='0' the second axis to be swapped.
name string, optional Name of the resulting symbol.

Details

x = [[1, 2, 3]] swapaxes(x, 0, 1) = [[ 1], [ 2], [ 3]]
x = [[[ 0, 1], [ 2, 3]], [[ 4, 5], [ 6, 7]]] // (2,2) array  
swapaxes(x, 0, 2) = [[[ 0, 4], [ 2, 6]], [[ 1, 5], [ 3, 7]]]
Defined in src/operator/swapaxis.cc:L70

Value

out The result mx.symbol

mx.symbol.SwapAxis

SwapAxis: Interchanges two axes of an array.

Description

Examples::

Usage

mx.symbol.SwapAxis(...)

mx.symbol.SwapAxis

SwapAxis: Interchanges two axes of an array.

Description

Examples::

Usage

mx.symbol.SwapAxis(...)
**Arguments**

- **data**: NDArray-or-Symbol Input array.
- **dim1**: int, optional, default='0' the first axis to be swapped.
- **dim2**: int, optional, default='0' the second axis to be swapped.
- **name**: string, optional Name of the resulting symbol.

**Details**

\[
\begin{align*}
x &= [[1, 2, 3]] \quad \text{swapaxes}(x, 0, 1) = [[ 1], [ 2], [ 3]] \\
x &= [[[ 0, 1], [ 2, 3]], [[ 4, 5], [ 6, 7]]] \quad \text{// (2,2,2) array} \\
\text{swapaxes}(x, 0, 2) &= [[[ 0, 4], [ 2, 6]], [[ 1, 5], [ 3, 7]]]
\end{align*}
\]

Defined in src/operator/swapaxis.cc:L70

**Value**

- **out**: The result mx.symbol

**Description**

This function slices the input array along a particular axis with the provided indices.

**Usage**

```python
mx.symbol.take(...)```

**Arguments**

- **a**: NDArray-or-Symbol The input array.
- **indices**: NDArray-or-Symbol The indices of the values to be extracted.
- **axis**: int, optional, default='0' The axis of input array to be taken. For input tensor of rank r, it could be in the range of [-r, r-1]
- **mode**: 'clip', 'raise', 'wrap'.optional, default='clip' Specify how out-of-bound indices behave. Default is "clip". "clip" means clip to the range. So, if all indices mentioned are too large, they are replaced by the index that addresses the last element along an axis. "wrap" means to wrap around. "raise" means to raise an error when index out of range.
- **name**: string, optional Name of the resulting symbol.
mx.symbol.tan

details

Given data tensor of rank $r \geq 1$, and indices tensor of rank $q$, gather entries of the axis dimension of data (by default outer-most one as axis=0) indexed by indices, and concatenates them in an output tensor of rank $q + (r - 1)$.

Examples:

```python
x = [4., 5., 6.]
// Trivial case, take the second element along the first axis.
take(x, [1]) = [5.]
// The other trivial case, axis=-1, take the third element along the first axis
take(x, [3], axis=-1, mode='clip') = [6.]
x = [[1., 2.], [3., 4.], [5., 6.]]
// In this case we will get rows 0 and 1, then 1 and 2. Along axis 0
take(x, [[0,1],[1,2]]) = [[[1., 2.], [3., 4.]],
[[3., 4.], [5., 6.]]]
// In this case we will get rows 0 and 1, then 1 and 2 (calculated by wrapping around). // Along axis 1
take(x, [[0, 3], [-1, -2]], axis=1, mode='wrap') = [[[1., 2.], [2., 1.]],
[[3., 4.], [4., 3.]],
[[5., 6.], [6., 5.]]]
```

The storage type of “take” output depends upon the input storage type:

```text
- take(default, default) = default
- take(csr, default, axis=0) = csr
```

Defined in src/operator/tensor/indexing_op.cc:L782

value

```
out The result mx.symbol
```

tan: Computes the element-wise tangent of the input array.

description

The input should be in radians (\(2\pi\) rad equals 360 degrees).

Usage

```
mx.symbol.tan(...)
```

Arguments

```text
data NDArray-or-Symbol The input array.
nname string, optional Name of the resulting symbol.
```
mx.symbol.tanh

Details

.. math:: \text{tan([0, \pi/4, \pi/2])} = [0, 1, -\infty]

The storage type of “tan” output depends upon the input storage type:

- tan(default) = default
- tan(row_sparse) = row_sparse
- tan(csr) = csr

Defined in src/operator/tensor/elementwise_unary_op_trig.cc:L140

Value

out The result mx.symbol

mx.symbol.tanh

.. math:: \text{tanh} : \text{Returns the hyperbolic tangent of the input array, computed element-wise.}

Description

.. math:: \text{tanh}(x) = \frac{\text{sinh}(x)}{\text{cosh}(x)}

Usage

mx.symbol.tanh(...)

Arguments

data NDArray-or-Symbol The input array.

name string, optional Name of the resulting symbol.

Details

The storage type of “tanh” output depends upon the input storage type:

- tanh(default) = default
- tanh(row_sparse) = row_sparse
- tanh(csr) = csr

Defined in src/operator/tensor/elementwise_unary_op_trig.cc:L451

Value

out The result mx.symbol
**mx.symbol.tile**

Tile: Repeats the whole array multiple times. If “reps” has length *d*, and input array has dimension of *n*. There are three cases:

- **n=d**. Repeat *i*-th dimension of the input by “reps[i]” times::
  \[
  x = \begin{bmatrix}
  [1, 2], [3, 4]
  \end{bmatrix}
  \]
  \[
  \text{tile}(x, \text{reps}=(2,3)) = \begin{bmatrix}
  [1, 2, 1, 2, 1, 2], [3, 4, 3, 4, 3, 4], [1, 2, 1, 2, 1, 2], [3, 4, 3, 4, 3, 4]
  \end{bmatrix}
  \]

- **n>d**. “reps” is promoted to length *n* by pre-pending 1’s to it. Thus for an input shape “(2,3)”, “reps=(2,)” is treated as “(1,2)”: \[
  \text{tile}(x, \text{reps}=(2,)) = \begin{bmatrix}
  [1, 2], [3, 4]
  \end{bmatrix}
  \]

- **n<d**. The input is promoted to be d-dimensional by prepending new axes. So a shape “(2,2)” array is promoted to “(1,2,2)” for 3-D replication::
  \[
  \text{tile}(x, \text{reps}=(2,2,3)) = \begin{bmatrix}
  [1, 2, 1, 2, 1, 2], [3, 4, 3, 4, 3, 4], [1, 2, 1, 2, 1, 2], [3, 4, 3, 4, 3, 4]
  \end{bmatrix}
  \]

**Description**

Defined in src/operator/tensor/matrix_op.cc:L796

**Usage**

\[
\text{mx.symbol.tile}(\ldots)
\]

**Arguments**

- **data**: NDArray-or-Symbol Input data array
- **reps**: Shape(tuple), required The number of times for repeating the tensor a. Each dimension of reps must be a positive integer. If reps has length d, the result will have dimension of max(d, a.ndim); if a.ndim < d, a is promoted to be d-dimensional by prepending new axes. If a.ndim > d, reps is promoted to a.ndim by pre-pending 1’s to it.
- **name**: string, optional Name of the resulting symbol.

**Value**

out The result mx.symbol

**mx.symbol.topk**

topk: Returns the indices of the top *k* elements in an input array along the given axis (by default). If ret_type is set to ‘value’ returns the value of top *k* elements (instead of indices). In case of ret_type = ‘both’, both value and index would be returned. The returned elements will be sorted.
**Description**

Examples::

**Usage**

```
mx.symbol.topk(...)
```

**Arguments**

- **data**
  NDArray-or-Symbol The input array

- **axis**
  int or None, optional, default='-1' Axis along which to choose the top k indices. If not given, the flattened array is used. Default is -1.

- **k**
  int, optional, default='1' Number of top elements to select, should be always smaller than or equal to the element number in the given axis. A global sort is performed if set k < 1.

- **ret_typ**
  'both', 'indices', 'mask', 'value',optional, default='indices' The return type. "value" means to return the top k values, "indices" means to return the indices of the top k values, "mask" means to return a mask array containing 0 and 1. 1 means the top k values. "both" means to return a list of both values and indices of top k elements.

- **is_ascend**
  boolean, optional, default=0 Whether to choose k largest or k smallest elements. Top K largest elements will be chosen if set to false.

- **dtype**
  'float16', 'float32', 'float64', 'int32', 'int64', 'uint8',optional, default='float32' DType of the output indices when ret_typ is "indices" or "both". An error will be raised if the selected data type cannot precisely represent the indices.

- **name**
  string, optional Name of the resulting symbol.

**Details**

```
x = [[ 0.3, 0.2, 0.4], [ 0.1, 0.3, 0.2]]
// returns an index of the largest element on last axis topk(x) = [[ 2. ], [ 1. ]]
// returns the value of top-2 largest elements on last axis topk(x, ret_typ='value', k=2) = [[ 0.4, 0.3], [ 0.3, 0.2]]
// returns the value of top-2 smallest elements on last axis topk(x, ret_typ='value', k=2, is_ascend=1) = [[ 0.2 , 0.3], [ 0.1 , 0.2]]
// returns the value of top-2 largest elements on axis 0 topk(x, axis=0, ret_typ='value', k=2) = [[ 0.3, 0.3, 0.4], [ 0.1, 0.2, 0.2]]
// flattens and then returns list of both values and indices topk(x, ret_typ='both', k=2) = [[[ 0.4, 0.3], [ 0.3, 0.2]], [[ 2., 0.], [ 1., 2.]]]
```

Defined in src/operator/tensor/ordering_op.cc:L68

**Value**

```
out The result mx.symbol
```
**mx.symbol.transpose**  
transpose:Permutes the dimensions of an array. Examples::

```
x = [[ 1., 2.], [ 3., 4.]]
x = [[ 1., 2.], [ 3., 4.], [ 5., 6.], [ 7., 8.]]
transpose(x) = [[ 1., 3.], [ 2., 4.], [ 5., 7.], [ 6., 8.]]
x = [[[ 1., 2.], [ 3., 4.]], [[ 5., 6.], [ 7., 8.]]]
x = [[[ 1., 2.], [ 3., 4.], [ 5., 6.]], [[ 7., 8., 9.]]]
```

**Description**

Defined in src/operator/tensor/matrix_op.cc:L328

**Usage**

```
mx.symbol.transpose(...)
```

**Arguments**

- **data**: NDArray-or-Symbol Source input  
- **axes**: Shape(tuple), optional, default=[] Target axis order. By default the axes will be inverted.  
- **name**: string, optional Name of the resulting symbol.

**Value**

```
out The result mx.symbol
```

**mx.symbol.trunc**  
trunc:Return the element-wise truncated value of the input.

**Description**

The truncated value of the scalar x is the nearest integer i which is closer to zero than x is. In short, the fractional part of the signed number x is discarded.

**Usage**

```
mx.symbol.trunc(...)
```

**Arguments**

- **data**: NDArray-or-Symbol The input array.  
- **name**: string, optional Name of the resulting symbol.
Details

Example::
trunc([-2.1, -1.9, 1.5, 1.9, 2.1]) = [-2., -1., 1., 1., 2.]
The storage type of “trunc” output depends upon the input storage type:
- trunc(default) = default - trunc(row_sparse) = row_sparse - trunc(csr) = csr
Defined in src/operator/tensor/elemwise_unary_op_basic.cc:L856

Value
out The result mx.symbol

---

mx.symbol.uniform  
uniform:Draw random samples from a uniform distribution.

Description
.. note:: The existing alias “uniform” is deprecated.

Usage
mx.symbol.uniform(...)

Arguments
- low float, optional, default=0 Lower bound of the distribution.
- high float, optional, default=1 Upper bound of the distribution.
- shape Shape(tuple), optional, default=None Shape of the output.
- ctx string, optional, default=’” Context of output, in format [cpu|gpu|cpu_pinned](n). Only used for imperative calls.
- dtype 'None', 'float16', 'float32', 'float64', optional, default='None' DType of the output in case this can’t be inferred. Defaults to float32 if not defined (dtype=None).
- name string, optional Name of the resulting symbol.

Details
Samples are uniformly distributed over the half-open interval *[low, high)* (includes *low*, but excludes *high*).
Example::
uniform(low=0, high=1, shape=(2,2)) = [[ 0.60276335, 0.85794562], [ 0.54488319, 0.84725171]]
Defined in src/operator/random/sample_op.cc:L96

Value
out The result mx.symbol
mx.symbol.unravel_index

unravel_index: Converts an array of flat indices into a batch of index arrays. The operator follows numpy conventions so a single multi-index is given by a column of the output matrix. The leading dimension may be left unspecified by using -1 as placeholder.

Description

Examples:

Usage

mx.symbol.unravel_index(...)

Arguments

data NDArray-or-Symbol Array of flat indices
shape Shape(tuple), optional, default=None Shape of the array into which the multi-indices apply.
name string, optional Name of the resulting symbol.

Details

A = [22,41,37] unravel(A, shape=(7,6)) = [[3,6,6],[4,5,1]] unravel(A, shape=(-1,6)) = [[3,6,6],[4,5,1]]

Defined in src/operator/tensor/ravel.cc:L68

Value

out The result mx.symbol

mx.symbol.UpSampling

UpSampling: Upsamples the given input data.

Description

Two algorithms (“sample_type”) are available for upsampling:

Usage

mx.symbol.UpSampling(...)
Arguments

data NDArray-or-Symbol[] Array of tensors to upsample. For bilinear upsampling, there should be 2 inputs - 1 data and 1 weight.
scale int, required Up sampling scale
num.filter int, optional, default='0' Input filter. Only used by bilinear sample_type. Since bilinear upsampling uses deconvolution, num_filters is set to the number of channels.
sample.type 'bilinear', 'nearest', required upsampling method
multi.input.mode 'concat', 'sum', optional, default='concat' How to handle multiple input. concat means concatenate upsampled images along the channel dimension. sum means add all images together, only available for nearest neighbor upsampling.
num.args int, required Number of inputs to be upsampled. For nearest neighbor upsampling this can be 1-N; the size of output will be(scale*h_0, scale*w_0) and all other inputs will be upsampled to the same size. For bilinear upsampling this must be 2; 1 input and 1 weight.
workspace long (non-negative), optional, default=512 Tmp workspace for deconvolution (MB)
name string, optional Name of the resulting symbol.

Details

- Nearest Neighbor - Bilinear

**Nearest Neighbor Upsampling**
Input data is expected to be NCHW.
Example::
x = [[[1. 1. 1.] [1. 1. 1.] [1. 1. 1.]]]
UpSampling(x, scale=2, sample_type='nearest') = [[[1. 1. 1. 1. 1. 1.] [1. 1. 1. 1. 1. 1.] [1. 1. 1. 1. 1. 1.] [1. 1. 1. 1. 1. 1.]]

**Bilinear Upsampling**
Uses 'deconvolution' algorithm under the hood. You need provide both input data and the kernel. Input data is expected to be NCHW.
'num_filter' is expected to be same as the number of channels.
Example::
x = [[[1. 1. 1.] [1. 1. 1.]]]
w = [[[1. 1. 1. 1.] [1. 1. 1. 1.] [1. 1. 1. 1. ]] [1. 1. 1. 1.]]
UpSampling(x, w, scale=2, sample_type='bilinear', num_filter=1) = [[[2. 2. 2. 2. 2. 2. ] [2. 2. 2. 2. 2. 2. ] [2. 2. 2. 2. 2. 2.] [2. 2. 2. 2. 2. 2. ] [2. 2. 2. 2. 2. 2. ] [1. 1. 1. 1. 1. 1.]]
Defined in src/operator/nn/upsampling.cc:L173

Value

out The result mx.symbol
mx.symbol.Variable

Create a symbolic variable with specified name.

Description

Create a symbolic variable with specified name.

Arguments

name: string The name of the result symbol.

Value

The result symbol

mx.symbol.where

where:Return the elements, either from x or y, depending on the condition.

Description

Given three ndarrays, condition, x, and y, return an ndarray with the elements from x or y, depending on the elements from condition are true or false. x and y must have the same shape. If condition has the same shape as x, each element in the output array is from x if the corresponding element in the condition is true, and from y if false.

Usage

mx.symbol.where(...) 

Arguments

condition: NDArray-or-Symbol condition array
x: NDArray-or-Symbol
y: NDArray-or-Symbol
name: string, optional Name of the resulting symbol.
mx.symbol.zeros_like

Details

If condition does not have the same shape as x, it must be a 1D array whose size is the same as x’s first dimension size. Each row of the output array is from x’s row if the corresponding element from condition is true, and from y’s row if false.

Note that all non-zero values are interpreted as “True” in condition.

Examples::

x = [[1, 2], [3, 4]] y = [[5, 6], [7, 8]] cond = [[0, 1], [-1, 0]]
where(cond, x, y) = [[5, 2], [3, 8]]
csr_cond = cast_storage(cond, ’csr’)
where(csr_cond, x, y) = [[5, 2], [3, 8]]
Defined in src/operator/tensor/control_flow_op.cc:L57

Value

out The result mx.symbol

mx.symbol.zeros_like

describe:Return an array of zeros with the same shape, type and storage type as the input array.

Description

The storage type of “zeros_like“ output depends on the storage type of the input

Usage

mx.symbol.zeros_like(...)

Arguments

data NDArray-or-Symbol The input
name string, optional Name of the resulting symbol.

Details

- zeros_like(row_sparse) = row_sparse - zeros_like(csr) = csr - zeros_like(default) = default

Examples::

x = [[1, 1, 1], [1, 1, 1]]
zeros_like(x) = [[0, 0, 0], [0, 0, 0]]

Value

out The result mx.symbol
mx.unserialize

Unserialize MXNet model from Robject.

Description
Unserialize MXNet model from Robject.

Usage
mx.unserialize(model)

Arguments
model The mxnet model loaded from RData files.

mxnet

MXNet: Flexible and Efficient GPU computing and Deep Learning.

Description
MXNet is a flexible and efficient GPU computing and deep learning framework.

Details
It enables you to write seamless tensor/matrix computation with multiple GPUs in R.
It also enables you construct and customize the state-of-art deep learning models in R, and apply
them to tasks such as image classification and data science challenges.

mxnet.export

Internal function to generate mxnet_generated.R Users do not need to
call this function.

Description
Internal function to generate mxnet_generated.R Users do not need to call this function.

Usage
mxnet.export(path)

Arguments
path The path to the root of the package.
### Binary operator overloading of `mx.ndarray`

**Description**

Binary operator overloading of `mx.ndarray`

**Usage**

```r
## S3 method for class 'MXNDArray'
Ops(e1, e2)
```

**Arguments**

- `e1`: The second operand

---

### `outputs` (Get the outputs of a symbol)

**Description**

Get the outputs of a symbol.

**Usage**

```r
outputs(x)
```

**Arguments**

- `x`: The input symbol

---

### `predict.MXFeedForwardModel` (Predict the outputs given a model and dataset)

**Description**

Predict the outputs given a model and dataset.
Usage

```r
## S3 method for class 'MXFeedForwardModel'
predict(
  model,
  X,
  ctx = NULL,
  array.batch.size = 128,
  array.layout = "auto",
  allow.extra.params = FALSE
)
```

Arguments

- **model**: The MXNet Model.
- **X**: The dataset to predict.
- **ctx**: mx.cpu() or mx.gpu(). The device used to generate the prediction.
- **array.batch.size**: The batch size used in batching. Only used when `X` is R’s array.
- **array.layout**: can be "auto", "colmajor", "rowmajor", (default=auto) The layout of array. "rowmajor" is only supported for two dimensional array. For matrix, "rowmajor" means dim(X) = c(nexample, nfeatures), "colmajor" means dim(X) = c(nfeatures, nexample) "auto" will auto detect the layout by match the feature size, and will report error when `X` is a square matrix to ask user to explicitly specify layout.
- **allow.extra.params**: Whether allow extra parameters that are not needed by symbol. If this is TRUE, no error will be thrown when arg_params or aux_params contain extra parameters that is not needed by the executor.

---

**print.MXNDArray**  
*print operator overload of mx.ndarray*

Description

print operator overload of mx.ndarray

Usage

```r
## S3 method for class 'MXNDArray'
print(nd)
```

Arguments

- **nd**: The mx.ndarray
**rnn.graph**

Generate a RNN symbolic model - requires CUDA

**Description**

Generate a RNN symbolic model - requires CUDA

**Usage**

```r
rnn.graph(
    num_rnn_layer,
    input_size = NULL,
    num_embed = NULL,
    num_hidden,
    num_decode,
    dropout = 0,
    ignore_label = -1,
    bidirectional = F,
    loss_output = NULL,
    config,
    cell_type,
    masking = F,
    output_last_state = F,
    rnn.state = NULL,
    rnn.state.cell = NULL,
    prefix = ""
)
```

**Arguments**

- `num_rnn_layer` int, number of stacked layers
- `input_size` int, number of levels in the data - only used for embedding
- `num_embed` int, default = NULL - no embedding. Dimension of the embedding vectors
- `num_hidden` int, size of the state in each RNN layer
- `num_decode` int, number of output variables in the decoding layer
- `dropout` 0
- `ignore_label` -1
- `bidirectional` F
- `loss_output` NULL
- `config` Either seq-to-one or one-to-one
- `cell_type` Type of RNN cell: either gru or lstm
- `masking` F
- `output_last_state` F
- `rnn.state` NULL
- `rnn.state.cell` NULL
- `prefix` ""
Description

Unroll representation of RNN running on non CUDA device

Usage

rnn.graph.unroll(
    num_rnn_layer,
    seq_len,
    input_size = NULL,
    num_embed = NULL,
    num_hidden,
    num_decode,
    dropout = 0,
    ignore_label = -1,
    loss_output = NULL,
    init.state = NULL,
    config,
    cell_type = "lstm",
    masking = F,
    output_last_state = F,
    prefix = "",
    data_name = "data",
    label_name = "label"
)

Arguments

num_rnn_layer int, number of stacked layers
seq_len int, number of time steps to unroll
input_size int, number of levels in the data - only used for embedding
num_embed int, default = NULL - no embedding. Dimension of the embedding vectors
num_hidden int, size of the state in each RNN layer
num_decode int, number of output variables in the decoding layer
dropout
config Either seq-to-one or one-to-one
cell_type Type of RNN cell: either gru or lstm
Index

*Topic datasets
  mx.metric.accuracy, 58
  mx.metric.logistic_acc, 58
  mx.metric.logloss, 59
  mx.metric.mae, 59
  mx.metric.mse, 59
  mx.metric.Perplexity, 60
  mx.metric.rmse, 59
  mx.metric.rmsle, 60
  mx.metric.top_k_accuracy, 61

arguments, 15
as.array.MXNDArray, 16
as.matrix.MXNDArray, 16

children, 16
cxt, 17
dim.MXNDArray, 17
graph.viz, 17

im2rec, 18
internals, 19
is.mx.context, 20
is.mx.dataiter, 20
is.mx.ndarray, 20
is.mx.symbol, 21
is.serialized, 21

length.MXNDArray, 22
mx.apply, 22
mx.callback.early.stop, 23
mx.callback.log.speedometer, 23
mx.callback.log.train.metric, 24
mx.callback.save.checkpoint, 24
mx.cpu, 25
mx.ctx.default, 25
mx.exec.backward, 25
mx.exec.forward, 26

mx.exec.update.arg.arrays, 26
mx.exec.update.aux.arrays, 26
mx.exec.update.grad.arrays, 27
mx.gpu, 27
mx.infer.rnn, 28
mx.infer.rnn.one, 28
mx.infer.rnn.one.unroll, 29
mx.init.create, 29
mx.init.internal.default, 30
mx.init.normal, 30
mx.init.uniform, 30
mx.init.Xavier, 31
mx.io.arrayiter, 31
mx.io.bucket.iter, 32
mx.io.CSVIter, 32
mx.io.extract, 34
mx.io.ImageDetRecordIter, 34
mx.io.ImageRecordInt8Iter, 38
mx.io.ImageRecordIter, 41
mx.io.ImageRecordIter_v1, 44
mx.io.ImageRecordUInt8Iter, 47
mx.io.ImageRecordUInt8Iter_v1, 50
mx.io.LibSVMIter, 53
mx.io.MNISTIter, 55
mx.kv.create, 56
mx.lr_scheduler.FactorScheduler, 56
mx.lr_scheduler.MultiFactorScheduler, 57
mx.metric.accuracy, 58
mx.metric.custom, 58
mx.metric.logistic_acc, 58
mx.metric.logloss, 59
mx.metric.mae, 59
mx.metric.mse, 59
mx.metric.Perplexity, 60
mx.metric.rmse, 60
mx.metric.rmsle, 60
mx.metric.top_k_accuracy, 61
mx.mlp, 61
mx.model.buckets, 62
mx.model.FeedForward.create, 63
mx.model.init.params, 65
mx.model.load, 65
mx.model.save, 66
mx.nd.abs, 66
mx.nd.Activation, 67
mx.nd.adam.update, 67
mx.nd.add.n, 68
mx.nd.all.finite, 69
mx.nd.amp.cast, 69
mx.nd.amp.multicast, 70
mx.nd.arccos, 70
mx.nd.arccosh, 71
mx.nd.arcsin, 71
mx.nd.arcsinh, 72
mx.nd.arctan, 72
mx.nd.arctanh, 73
mx.nd.argmax, 73
mx.nd.argmax.channel, 74
mx.nd.argmin, 75
mx.nd.argsort, 75
mx.nd.array, 76
mx.nd.batch.dot, 77
mx.nd.batch.take, 78
mx.nd.BatchNorm, 78
mx.nd.BatchNorm.v1, 80
mx.nd.BilinearSampler, 81
mx.nd.BlockGrad, 82
mx.nd.broadcast.add, 83
mx.nd.broadcast.axes, 84
mx.nd.broadcast.axis, 84
mx.nd.broadcast.div, 85
mx.nd.broadcast.equal, 86
mx.nd.broadcast.greater, 86
mx.nd.broadcast.greater.equal, 87
mx.nd.broadcast.hypot, 87
mx.nd.broadcast.lesser, 88
mx.nd.broadcast.lesser.equal, 89
mx.nd.broadcast.like, 89
mx.nd.broadcast.logical.and, 90
mx.nd.broadcast.logical.or, 91
mx.nd.broadcast.logical.xor, 91
mx.nd.broadcast.maximum, 92
mx.nd.broadcast.minimum, 92
mx.nd.broadcast.minus, 93
mx.nd.broadcast.mod, 94
mx.nd.broadcast.mul, 94
mx.nd.broadcast.not.equal, 95
mx.nd.broadcast.plus, 95
mx.nd.broadcast.power, 96
mx.nd.broadcast.sub, 97
mx.nd.broadcast.to, 97
mx.nd.Cast, 98
mx.nd.cast, 99
mx.nd.cast.storage, 99
mx.nd.cbrt, 100
mx.nd.ceil, 101
mx.nd.choose.element.0index, 101
mx.nd.clip, 102
mx.nd.col2im, 103
mx.nd.Concat, 104
mx.nd.concat, 104
mx.nd.Convolution, 105
mx.nd.Convolution.v1, 107
mx.nd.copyto, 108
mx.nd.Correlation, 109
mx.nd.cos, 110
mx.nd.cosh, 110
mx.nd.Crop, 111
mx.nd.crop, 112
mx.nd.ctc.loss, 113
mx.nd.CTCLoss, 114
mx.nd.cumsum, 116
mx.nd.Custom, 116
mx.nd.Deconvolution, 117
mx.nd.degrees, 118
mx.nd.depth.to.space, 119
mx.nd.diag, 119
mx.nd.dot, 120
mx.nd.Dropout, 122
mx.nd.ElementWiseSum, 123
mx.nd.elemwise.add, 123
mx.nd.elemwise.div, 124
mx.nd.elemwise.mul, 124
mx.nd.elemwise.sub, 125
mx.nd.Embedding, 125
mx.nd.erf, 126
mx.nd.erfinv, 127
mx.nd.exp, 127
mx.nd.expand.dims, 128
mx.nd.expm1, 128
mx.nd.fill.element.0index, 129
mx.nd.fix, 129
mx.nd.Flatten, 130
mx.nd.flatten, 130
INDEX

mx.nd.flip, 131
mx.nd.floor, 131
mx.nd.fft1.update, 132
mx.nd.fft1.update, 133
mx.nd.FullyConnected, 134
mx.nd.gamma, 135
mx.nd.gammaln, 135
mx.nd.gather.nd, 136
mx.nd.GridGenerator, 136
mx.nd.GroupNorm, 137
mx.nd.hard.sigmoid, 138
mx.nd.IdentityAttachKLSparseReg, 139
mx.nd.im2col, 139
mx.nd.InstanceNorm, 140
mx.nd.khatri.rao, 141
mx.nd.L2Normalization, 142
mx.nd.lamb.update.phase1, 143
mx.nd.lamb.update.phase2, 144
mx.nd.LayerNorm, 144
mx.nd.LeakyReLU, 145
mx.nd.linalg.det, 146
mx.nd.linalg.extractdiag, 147
mx.nd.linalg.extracttrian, 148
mx.nd.linalg.gelqf, 149
mx.nd.linalg.gemm, 150
mx.nd.linalg.gemm2, 151
mx.nd.linalg.inverse, 152
mx.nd.linalg.makediag, 153
mx.nd.linalg.maketrian, 153
mx.nd.linalg.potrf, 154
mx.nd.linalg.potri, 155
mx.nd.linalg.slogdet, 156
mx.nd.linalg.sumlogdiag, 157
mx.nd.linalg.syrk, 157
mx.nd.linalg.trmm, 158
mx.nd.linalg.trsm, 159
mx.nd.LinearRegressionOutput, 160
mx.nd.load, 161
mx.nd.log, 161
mx.nd.logsoftmax, 162
mx.nd.log10, 163
mx.nd.log1p, 163
mx.nd.log2, 164
mx.nd.logical.not, 164
mx.nd.LogisticRegressionOutput, 165
mx.nd.LRN, 166
mx.nd.MAERegressionOutput, 166
mx.nd.make.loss, 167
mx.nd.MakeLoss, 168
mx.nd.max, 169
mx.nd.max.axis, 169
mx.nd.mean, 170
mx.nd.min, 171
mx.nd.min.axis, 171
mx.nd.moments, 172
mx.nd.mp.lamb.update.phase1, 173
mx.nd.mp.lamb.update.phase2, 174
mx.nd.mp.nag.mom.update, 175
mx.nd.mp.sgd.mom.update, 175
mx.nd.mp.sgd.update, 176
mx.nd.multi.all.finite, 177
mx.nd.multi.lars, 177
mx.nd.multi.mp.sgd.mom.update, 178
mx.nd.multi.mp.sgd.update, 179
mx.nd.multi.sgd.mom.update, 179
mx.nd.multi.sgd.update, 180
mx.nd.multi.sum.sq, 181
mx.nd.nag.mom.update, 181
mx.nd.nanprod, 182
mx.nd.nansum, 183
mx.nd.negative, 184
mx.nd.norm, 184
mx.nd.normal, 185
mx.nd.one.hot, 186
mx.nd.ones, 187
mx.nd.ones_like, 187
mx.nd.Pad, 188
mx.nd.pad, 189
mx.nd.pick, 190
mx.nd.Pooling, 191
mx.nd.Pooling.v1, 193
mx.nd.preloaded.multi.mp.sgd.mom.update, 194
mx.nd.preloaded.multi.mp.sgd.update, 195
mx.nd.preloaded.multi.sgd.mom.update, 195
mx.nd.preloaded.multi.sgd.update, 196
mx.nd.prod, 197
mx.nd.radians, 197
mx.nd.random.exponential, 198
mx.nd.random.gamma, 199
mx.nd.random.generalized.negative.binomial, 199
mx.nd.random.negative.binomial, 200
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mx.nd.random.normal</td>
<td>201</td>
</tr>
<tr>
<td>mx.nd.random.pdf.dirichlet</td>
<td>202</td>
</tr>
<tr>
<td>mx.nd.random.pdf.exponential</td>
<td>202</td>
</tr>
<tr>
<td>mx.nd.random.pdf.gamma</td>
<td>203</td>
</tr>
<tr>
<td>mx.nd.random.pdf.generalized.negative.binomial</td>
<td>204</td>
</tr>
<tr>
<td>mx.nd.random.pdf.negative.binomial</td>
<td>205</td>
</tr>
<tr>
<td>mx.nd.random.pdf.normal</td>
<td>206</td>
</tr>
<tr>
<td>mx.nd.random.pdf.poisson</td>
<td>207</td>
</tr>
<tr>
<td>mx.nd.random.pdf.uniform</td>
<td>207</td>
</tr>
<tr>
<td>mx.nd.random.poisson</td>
<td>208</td>
</tr>
<tr>
<td>mx.nd.random.randint</td>
<td>209</td>
</tr>
<tr>
<td>mx.nd.random.uniform</td>
<td>209</td>
</tr>
<tr>
<td>mx.nd.ravel.multi.index</td>
<td>210</td>
</tr>
<tr>
<td>mx.nd.rcbrt</td>
<td>211</td>
</tr>
<tr>
<td>mx.nd.reciprocal</td>
<td>211</td>
</tr>
<tr>
<td>mx.nd.relu</td>
<td>212</td>
</tr>
<tr>
<td>mx.nd.repeat</td>
<td>212</td>
</tr>
<tr>
<td>mx.nd.reset.arrays</td>
<td>213</td>
</tr>
<tr>
<td>mx.nd.Reshape</td>
<td>213</td>
</tr>
<tr>
<td>mx.nd.reshape</td>
<td>215</td>
</tr>
<tr>
<td>mx.nd.reshape.like</td>
<td>216</td>
</tr>
<tr>
<td>mx.nd.reverse</td>
<td>217</td>
</tr>
<tr>
<td>mx.nd.rint</td>
<td>218</td>
</tr>
<tr>
<td>mx.nd.rmsprop.update</td>
<td>218</td>
</tr>
<tr>
<td>mx.nd.rmspropalex.update</td>
<td>219</td>
</tr>
<tr>
<td>mx.nd.RNN</td>
<td>221</td>
</tr>
<tr>
<td>mx.nd.ROIPooling</td>
<td>223</td>
</tr>
<tr>
<td>mx.nd.round</td>
<td>224</td>
</tr>
<tr>
<td>mx.nd.rsqrt</td>
<td>224</td>
</tr>
<tr>
<td>mx.nd.sample_exponential</td>
<td>225</td>
</tr>
<tr>
<td>mx.nd.sample.gamma</td>
<td>226</td>
</tr>
<tr>
<td>mx.nd.sample.generalized.negative.binomial</td>
<td>227</td>
</tr>
<tr>
<td>mx.nd.sample.multinomial</td>
<td>228</td>
</tr>
<tr>
<td>mx.nd.sample.negative.binomial</td>
<td>229</td>
</tr>
<tr>
<td>mx.nd.sample.normal</td>
<td>230</td>
</tr>
<tr>
<td>mx.nd.sample.poisson</td>
<td>231</td>
</tr>
<tr>
<td>mx.nd.sample.uniform</td>
<td>232</td>
</tr>
<tr>
<td>mx.nd.save</td>
<td>233</td>
</tr>
<tr>
<td>mx.nd.scatter.nd</td>
<td>233</td>
</tr>
<tr>
<td>mx.nd.SequenceLast</td>
<td>234</td>
</tr>
<tr>
<td>mx.nd.SequenceMask</td>
<td>235</td>
</tr>
<tr>
<td>mx.nd.SequenceReverse</td>
<td>236</td>
</tr>
<tr>
<td>mx.nd.sgd.mom.update</td>
<td>238</td>
</tr>
<tr>
<td>mx.nd.sgd.update</td>
<td>239</td>
</tr>
<tr>
<td>mx.nd.shape.array</td>
<td>240</td>
</tr>
<tr>
<td>mx.nd.shuffle</td>
<td>240</td>
</tr>
<tr>
<td>mx.nd.sigmoid</td>
<td>241</td>
</tr>
<tr>
<td>mx.nd.sign</td>
<td>241</td>
</tr>
<tr>
<td>mx.nd.sign.update</td>
<td>242</td>
</tr>
<tr>
<td>mx.nd.signsgd.update</td>
<td>242</td>
</tr>
<tr>
<td>mx.nd.signum.update</td>
<td>242</td>
</tr>
<tr>
<td>mx.nd.sin</td>
<td>243</td>
</tr>
<tr>
<td>mx.nd.sinh</td>
<td>244</td>
</tr>
<tr>
<td>mx.nd.size.array</td>
<td>244</td>
</tr>
<tr>
<td>mx.nd.slice.axis</td>
<td>245</td>
</tr>
<tr>
<td>mx.nd.slice.like</td>
<td>246</td>
</tr>
<tr>
<td>mx.nd.SliceChannel</td>
<td>247</td>
</tr>
<tr>
<td>mx.nd.smooth.11</td>
<td>248</td>
</tr>
<tr>
<td>mx.nd.Softmax</td>
<td>248</td>
</tr>
<tr>
<td>mx.nd.softmax</td>
<td>250</td>
</tr>
<tr>
<td>mx.nd.softmax.cross.entropy</td>
<td>251</td>
</tr>
<tr>
<td>mx.nd.SoftmaxActivation</td>
<td>252</td>
</tr>
<tr>
<td>mx.nd.SoftmaxOutput</td>
<td>253</td>
</tr>
<tr>
<td>mx.nd.softmin</td>
<td>254</td>
</tr>
<tr>
<td>mx.nd.softsign</td>
<td>255</td>
</tr>
<tr>
<td>mx.nd.sort</td>
<td>256</td>
</tr>
<tr>
<td>mx.nd.space.to.depth</td>
<td>256</td>
</tr>
<tr>
<td>mx.nd.SpatialTransformer</td>
<td>257</td>
</tr>
<tr>
<td>mx.nd.split</td>
<td>258</td>
</tr>
<tr>
<td>mx.nd.sqrt</td>
<td>259</td>
</tr>
<tr>
<td>mx.nd.square</td>
<td>259</td>
</tr>
<tr>
<td>mx.nd.squeeze</td>
<td>260</td>
</tr>
<tr>
<td>mx.nd.stack</td>
<td>260</td>
</tr>
<tr>
<td>mx.nd.stop.gradient</td>
<td>261</td>
</tr>
<tr>
<td>mx.nd.sum</td>
<td>262</td>
</tr>
<tr>
<td>mx.nd.sum.axis</td>
<td>263</td>
</tr>
<tr>
<td>mx.nd.SVMOutput</td>
<td>264</td>
</tr>
<tr>
<td>mx.nd.swapaxes</td>
<td>264</td>
</tr>
<tr>
<td>mx.nd.SwapAxis</td>
<td>265</td>
</tr>
<tr>
<td>mx.nd.take</td>
<td>265</td>
</tr>
<tr>
<td>mx.nd.tan</td>
<td>266</td>
</tr>
<tr>
<td>mx.nd.tanh</td>
<td>267</td>
</tr>
<tr>
<td>mx.nd.tile</td>
<td>268</td>
</tr>
<tr>
<td>mx.nd.topk</td>
<td>268</td>
</tr>
<tr>
<td>mx.nd.transpose</td>
<td>269</td>
</tr>
<tr>
<td>mx.nd.trunc</td>
<td>270</td>
</tr>
<tr>
<td>mx.nd.uniform</td>
<td>271</td>
</tr>
<tr>
<td>mx.nd.unravel.index</td>
<td>271</td>
</tr>
<tr>
<td>mx.nd.UpSampling</td>
<td>272</td>
</tr>
<tr>
<td>mx.nd.where</td>
<td>273</td>
</tr>
<tr>
<td>mx.nd.zeros</td>
<td>274</td>
</tr>
<tr>
<td>mx.nd.zeros.like</td>
<td>274</td>
</tr>
<tr>
<td>mx.opt.adadelta</td>
<td>275</td>
</tr>
<tr>
<td>mx.opt.adagrad</td>
<td>276</td>
</tr>
</tbody>
</table>
mx.opt.adam, 276
mx.opt.create, 277
mx.opt.get.updater, 278
mx.opt.nag, 278
mx.opt.rmsprop, 279
mx.opt.sgd, 280
mx.profiler.config, 280
mx.profiler.state, 281
mx.runif, 282
mx.serialize, 283
mx.set.seed, 283
mx.simple.bind, 284
mx.symbol.abs, 284
mx.symbol.Activation, 285
mx.symbol.adam_update, 285
mx.symbol.add_n, 287
mx.symbol.all_finite, 287
mx.symbol.amp_cast, 288
mx.symbol.amp_multicast, 288
mx.symbol.arccos, 289
mx.symbol.arccosh, 289
mx.symbol.arcsin, 290
mx.symbol.arcsinh, 290
mx.symbol.arctan, 290
mx.symbol.arctanh, 290
mx.symbol.argmax, 293
mx.symbol.argmax_channel, 294
mx.symbol.argmin, 294
mx.symbol.arsort, 295
mx.symbol.batch_dot, 299
mx.symbol.batch_take, 300
mx.symbol.BatchNorm, 296
mx.symbol.BatchNorm_v1, 298
mx.symbol.BilinearSampler, 301
mx.symbol.BlockGrad, 302
mx.symbol.broadcast_add, 303
mx.symbol.broadcast_axes, 304
mx.symbol.broadcast_axis, 305
mx.symbol.broadcast_div, 306
mx.symbol.broadcast_equal, 306
mx.symbol.broadcast_greater, 307
mx.symbol.broadcast_greater_equal, 308
mx.symbol.broadcast_hypot, 308
mx.symbol.broadcast_less, 309
mx.symbol.broadcast_less_equal, 310
mx.symbol.broadcast_like, 311
mx.symbol.broadcast_logical_and, 312
mx.symbol.broadcast_logical_or, 312
mx.symbol.broadcast_logical_xor, 313
mx.symbol.broadcast_maximum, 314
mx.symbol.broadcast_minimum, 314
mx.symbol.broadcast_minus, 315
mx.symbol.broadcast_mod, 316
mx.symbol.broadcast_mul, 317
mx.symbol.broadcast_not_equal, 317
mx.symbol.broadcast_plus, 318
mx.symbol.broadcast_power, 319
mx.symbol.broadcast_sub, 320
mx.symbol.broadcast_to, 321
mx.symbol.Cast, 322
mx.symbol.cast, 322
mx.symbol.cast_storage, 323
mx.symbol.cbrt, 324
mx.symbol.ceil, 324
mx.symbol.choose_element_0index, 325
mx.symbol.clip, 326
mx.symbol.col2im, 327
mx.symbol.Concat, 328
mx.symbol.concat, 329
mx.symbol.Convolution, 329
mx.symbol.Convolution_v1, 331
mx.symbol.Correlation, 332
mx.symbol.cos, 334
mx.symbol.cosh, 334
mx.symbol.Crop, 335
mx.symbol.crop, 336
mx.symbol.ctc_loss, 338
mx.symbol.CTCloss, 337
mx.symbol.cumsum, 340
mx.symbol.Custom, 340
mx.symbol.Deconvolution, 341
mx.symbol.degrees, 342
mx.symbol.depth_to_space, 343
mx.symbol.diag, 344
mx.symbol.dot, 345
mx.symbol.Dropout, 346
mx.symbol.ElementWiseSum, 347
mx.symbol.ElemwiseAdd, 348
mx.symbol.ElemwiseDiv, 348
mx.symbol.ElemwiseMul, 349
mx.symbol.ElemwiseSub, 349
mx.symbol.Embedding, 350
mx.symbol.erf, 351
mx.symbol.erfinv, 352
mx.symbol.exp, 353
mx.symbol.expand_dims, 353
mx.symbol.expm1, 354
mx.symbol.fill_element_0index, 355
mx.symbol.fix, 355
mx.symbol.Flatten, 356
mx.symbol.flatten, 357
mx.symbol.flip, 357
mx.symbol.floor, 358
mx.symbol-ftml_update, 359
mx.symbol.ftrl_update, 360
mx.symbol.FullyConnected, 361
mx.symbol.gamma, 362
mx.symbol.gammaLn, 362
mx.symbol.gather_nd, 363
mx.symbol.GridGenerator, 363
mx.symbol.Group, 364
mx.symbol.GroupNorm, 365
mx.symbol.hard_sigmoid, 366
mx.symbol.identity, 366
mx.symbol.IdentityAttachKLSparseReg, 367
mx.symbol.im2col, 367
mx.symbol.InferShape, 368
mx.symbol.InstanceNorm, 369
mx.symbol.khatri_rao, 370
mx.symbol.L2Normalization, 371
mx.symbol.lamb_update_phase1, 372
mx.symbol.lamb_update_phase2, 373
mx.symbol.LayerNorm, 374
mx.symbol.LeakyReLU, 375
mx.symbol.linalg_det, 376
mx.symbol.linalg_extractdiag, 377
mx.symbol.linalg_extracttrian, 378
mx.symbol.linalg_geqlf, 379
mx.symbol.linalg_gemm, 380
mx.symbol.linalg_gemm2, 381
mx.symbol.linalg_inverse, 382
mx.symbol.linalg_makediag, 383
mx.symbol.linalg_maketrian, 384
mx.symbol.linalg_potrf, 385
mx.symbol.linalg_potri, 386
mx.symbol.linalg_slogdet, 387
mx.symbol.linalg_sumlogdiag, 388
mx.symbol.linalg_syrk, 389
mx.symbol.linalg_trmm, 390
mx.symbol.linalg_trsm, 391
mx.symbol.LinearRegressionOutput, 392
mx.symbol.load, 393
mx.symbol.load.json, 393
mx.symbol.log, 394
mx.symbol.log10, 394
mx.symbol.log1p, 395
mx.symbol.log2, 395
mx.symbol.log_softmax, 397
mx.symbol.logical_not, 396
mx.symbol.LogisticRegressionOutput, 396
mx.symbol LRN, 398
mx.symbol.MAERegressionOutput, 399
mx.symbol.make_loss, 401
mx.symbol.MakeLoss, 400
mx.symbol.max, 401
mx.symbol.max_axis, 402
mx.symbol.mean, 403
mx.symbol.moments, 404
mx.symbol.mp_lamb_update_phase1, 404
mx.symbol.mp_lamb_update_phase2, 406
mx.symbol.mp_nag_mom_update, 407
mx.symbol.mp_sgd_mom_update, 408
mx.symbol.mp_sgd_update, 409
mx.symbol.multi_all_finite, 409
mx.symbol.multi_lars, 410
mx.symbol.multi_mp_sgd_mom_update, 411
mx.symbol.multi_mp_sgd_update, 412
mx.symbol.multi_sgd_mom_update, 413
mx.symbol.multi_sgd_update, 414
mx.symbol.multi_sum_sq, 415
mx.symbol.nag_mom_update, 415
mx.symbol.nanprod, 416
mx.symbol.nansum, 417
mx.symbol.negative, 418
mx.symbol.norm, 418
mx.symbol.normal, 419
mx.symbol.one_hot, 421
mx.symbol.ones_like, 420
mx.symbol.Pad, 422
mx.symbol.pad, 423
mx.symbol.pick, 424
mx.symbol.Pooling, 426
mx.symbol.Pooling_v1, 427
mx.symbol.preloaded_multi_mp_sgd_mom_update, 429
mx.symbol.preloaded_multi_mp_sgd_update, 430
mx.symbol.preloaded_multi_sgd_mom_update, 430
mx.symbol.preloaded_multi_sgd_update, 433
mx.symbol.prod, 432
mx.symbol.radians, 433
mx.symbol.random_exponential, 433
mx.symbol.random_gamma, 434
mx.symbol.random_generalized_negative_binomial, 435
mx.symbol.random_negative_binomial, 436
mx.symbol.random_normal, 437
mx.symbol.random_pdf_dirichlet, 438
mx.symbol.random_pdf_exponential, 439
mx.symbol.random_pdf_gamma, 440
mx.symbol.random_pdf_generalized_negative_binomial, 441
mx.symbol.random_pdf_negative_binomial, 442
mx.symbol.random_pdf_normal, 443
mx.symbol.random_pdf_uniform, 444
mx.symbol.random_pdf_uniform, 445
mx.symbol.random_poisson, 446
mx.symbol.random_randint, 446
mx.symbol.random_uniform, 447
mx.symbol.ravel_multi_index, 448
mx.symbol.rcbrt, 449
mx.symbol.reciprocal, 449
mx.symbol.relu, 450
mx.symbol.repeat, 451
mx.symbol.reset_arrays, 451
mx.symbol.reshape, 452
mx.symbol.reshape_like, 453
mx.symbol.reverse, 455
mx.symbol.rint, 457
mx.symbol.rmsprop_update, 459
mx.symbol.rmspropalex_update, 457
mx.symbol.RNN, 460
mx.symbol.ROIPooling, 462
mx.symbol.round, 463
mx.symbol.rsqrt, 464
mx.symbol.sample_exponential, 465
mx.symbol.sample_gamma, 466
mx.symbol.sample_generalized_negative_binomial, 467
mx.symbol.sample_multinomial, 468
mx.symbol.sample_negative_binomial, 469
mx.symbol.sample_normal, 470
mx.symbol.sample_poisson, 471
mx.symbol.sample_uniform, 472
mx.symbol.save, 473
mx.symbol.scatter_nd, 473
mx.symbol.SequenceLast, 474
mx.symbol.SequenceMask, 475
mx.symbol.SequenceReverse, 477
mx.symbol.tensorLogical_or, 478
mx.symbol.tensorLogical_and, 479
mx.symbol.tensor_logical_or, 480
mx.symbol.tensor_logical_and, 481
mx.symbol.sigmoid, 481
mx.symbol.sgn, 482
mx.symbol.sign, 482
mx.symbol.sigmoid_update, 482
mx.symbol.signsgn, 483
mx.symbol.sin, 484
mx.symbol.sinh, 485
mx.symbol.size_array, 486
mx.symbol.slice, 486
mx.symbol.slice_axis, 489
mx.symbol.slice_like, 490
mx.symbol.SliceChannel, 488
mx.symbol.softmax, 491
mx.symbol.softmax_cross_entropy, 497
mx.symbol.softmaxActivation, 494
mx.symbol.softmaxOutput, 495
mx.symbol.softplus, 498
mx.symbol.Softmax, 491
mx.symbol.softmax_cross_entropy, 497
mx.symbol.softmaxActivation, 494
mx.symbol.softmaxOutput, 495
mx.symbol.sigmoids, 498
mx.symbol.sigmoids, 499
mx.symbol.stop_gradient, 499
mx.symbol.space_to_depth, 500
mx.symbol.SpatialTransformer, 501
mx.symbol.split, 502
mx.symbol.sqrt, 503
mx.symbol.square, 504
mx.symbol.squeeze, 504
mx.symbol.stack, 505
mx.symbol.stop_gradient, 506
mx.symbol.sum, 507
mx.symbol.sum_axis, 508
mx.symbol.SVMOutput, 509
mx.symbol.swapaxes, 510
mx.symbol.SwapAxis, 510
mx.symbol.take, 511
mx.symbol.tan, 512
mx.symbol.tanh, 513
mx.symbol.tile, 514
mx.symbol.topk, 514
mx.symbol.transpose, 516
mx.symbol.trunc, 516
mx.symbol.uniform, 517
mx.symbol.unravel_index, 518
mx.symbol.UpSampling, 518
mx.symbol.Variable, 520
mx.symbol.where, 520
mx.symbol.zeros_like, 521
mx.unserialize, 522
mxnet, 522
mxnet.export, 522

Ops.MXNDArray, 523
outputs, 523

predict.MXFeedForwardModel, 523
print.MXNDArray, 524

rnn.graph, 525
rnn.graph.unroll, 526