Package: mlr3torch (via r-universe)

October 30, 2026

Title Deep Learning with 'mlr3'

Version 0.4.0

Description Deep Learning library that extends the mlr3 framework by building upon the 'torch' package. It makes it convenient to build, train, and evaluate deep learning models without having to worry about low level details. Custom architectures can be created using the graph language defined in 'mlr3pipelines'.

License LGPL (>= 3)

BugReports https://github.com/mlr-org/mlr3torch/issues

URL https://mlr3torch.mlr-org.com/,
https://github.com/mlr-org/mlr3torch/

Depends mlr3 (>= 1.8.0), mlr3pipelines (>= 0.11.0), torch (>= 0.16.2), R (>= 4.2.0)

Imports backports, cli, checkmate (>= 2.2.0), coro, data.table, lgr, methods, mlr3misc (>= 0.23), paradox (>= 1.0.0), R6, withr

Suggests callr, curl, future, ggplot2, igraph, jsonlite, knitr, mirai, mlr3tuning (>= 1.0.0), progress, rmarkdown, rpart, viridis, visNetwork, testthat (>= 3.0.0), tibble, tfevents, torchvision (>= 0.10.0), waldo

Config/testthat/edition 3

NeedsCompilation no

ByteCompile yes

Encoding UTF-8

Collate 'CallbackSet.R' 'aaa.R' 'TorchCallback.R' 'CallbackSetCheckpoint.R' 'CallbackSetEarlyStopping.R' 'CallbackSetHistory.R' 'CallbackSetLRScheduler.R' 'CallbackSetProgress.R' 'CallbackSetTB.R' 'CallbackSetUnfreeze.R' 'ContextTorch.R' 'DataBackendLazy.R' 'utils.R' 'DataDescriptor.R' 'LearnerTorch.R' 'GraphLearnerTorch.R' 'LearnerFTTransformer.R' 'LearnerTorchFeatureless.R' 'LearnerTorchImage.R'

'LearnerTorchMLP.R' 'TorchDescriptor.R' 'TorchLoss.R'
'task_dataset.R' 'shape.R' 'PipeOpTorchIngress.R'
'LearnerTorchModel.R' 'LearnerTorchModule.R'
'LearnerTorchRealMLP.R' 'LearnerTorchTabM.R'
'LearnerTorchTabResNet.R' 'LearnerTorchVision.R'
'MeasureTorch.R' 'ModelDescriptor.R' 'PipeOpModule.R'
'PipeOpTorch.R' 'PipeOpTaskPreprocTorch.R'
'PipeOpTorchActivation.R' 'PipeOpTorchAdaptiveAvgPool.R'
'PipeOpTorchAdaptiveMaxPool.R' 'PipeOpTorchAvgPool.R'
'PipeOpTorchBatchNorm.R' 'PipeOpTorchBlock.R'
'PipeOpTorchCallbacks.R' 'PipeOpTorchConv.R'
'PipeOpTorchConvTranspose.R' 'PipeOpTorchDropout.R'
'PipeOpTorchFTCLS.R' 'PipeOpTorchFTTransformerBlock.R'
'PipeOpTorchFn.R' 'PipeOpTorchGroupNorm.R' 'PipeOpTorchHead.R'
'PipeOpTorchIdentity.R' 'PipeOpTorchLayerNorm.R'
'PipeOpTorchLinear.R' 'PipeOpTorchLoss.R'
'PipeOpTorchMaxPool.R' 'PipeOpTorchMerge.R'
'PipeOpTorchModel.R' 'PipeOpTorchMultiheadAttention.R'
'PipeOpTorchOptimizer.R' 'PipeOpTorchRecurrent.R'
'PipeOpTorchReshape.R' 'PipeOpTorchSoftmax.R'
'PipeOpTorchTokenizer.R' 'PipeOpTorchTransformerEncoderLayer.R'
'PredictionTorch.R' 'Select.R' 'TaskClassif_cifar.R'
'TaskClassif_lazy_iris.R' 'TaskClassif_melanoma.R'
'TaskClassif_mnist.R' 'TaskClassif_tiny_imagenet.R'
'TaskTorch.R' 'TorchOptimizer.R' 'bibentries.R' 'cache.R'
'lazy_tensor.R' 'learner_torch_methods.R' 'materialize.R'
'merge_graphs.R' 'multi_tensor_dataset.R' 'nn.R' 'nn_graph.R'
'paramset_torchlearner.R' 'pipeop_torch.R' 'preprocess.R'
'rd_info.R' 'with_torch_settings.R' 'zzz.R'

Config/roxygen2/version 8.1.0

Author Sebastian Fischer [cre, aut] (ORCID: <https://orcid.org/0000-0002-9609-3197>), Bernd Bischl [ctb] (ORCID: <https://orcid.org/0000-0001-6002-6980>), Lukas Burk [ctb] (ORCID: <https://orcid.org/0000-0001-7528-3795>), Martin Binder [aut], Florian Pfisterer [ctb] (ORCID: <https://orcid.org/0000-0001-8867-762X>), Carson Zhang [ctb]

Maintainer Sebastian Fischer <sebf.fischer@gmail.com>

Repository https://cran.r-universe.dev

Date/Publication 2026-10-30 19:34:27 UTC

RemoteUrl https://github.com/cran/mlr3torch

RemoteRef HEAD

RemoteSha ab1494873b713dedb1cb64290f2fdf2e6ca77190

Contents

mlr3torch-package 7

as_data_descriptor . . . 8
as_lazy_tensor . . . 9
as_learner_torch . . . 10
as_lr_scheduler . . . 12
as_task_torch . . . 12
as_torch_callback . . . 13
as_torch_callbacks . . . 14
as_torch_loss . . . 14
as_torch_optimizer . . . 16
assert_dim_in_range . . . 17
assert_known_dims . . . 18
assert_lazy_tensor . . . 19
assert_ndim . . . 19
assert_not_batch_dim . . . 20
assert_positive_extent . . . 21
assert_same_batch_size . . . 22
assert_same_ndim . . . 23
assert_shape . . . 23
assert_shapes . . . 24
auto_device . . . 25
batchgetter_categ . . . 26
batchgetter_num . . . 26
callback_set . . . 27
cross_entropy . . . 28
DataDescriptor . . . 29
encode_prediction . . . 32
get_batch_constructor . . . 33
get_batch_size . . . 34
get_target_batchgetter . . . 35
GraphLearnerTorch . . . 35
infer_shapes . . . 37
ingress_categ . . . 38
ingress_ltnsr . . . 39
ingress_num . . . 39
is_lazy_tensor . . . 40
lazy_shape . . . 40
lazy_tensor . . . 41
materialize . . . 41
mlr_backends_lazy . . . 43
mlr_callback_set . . . 46
mlr_callback_set.checkpoint . . . 49
mlr_callback_set.history . . . 51
mlr_callback_set.lr_scheduler . . . 53
mlr_callback_set.lr_scheduler_one_cycle . . . 55
mlr_callback_set.lr_scheduler_reduce_on_plateau . . . 56
mlr_callback_set.progress . . . 57
mlr_callback_set.tb . . . 59
mlr_callback_set.unfreeze . . . 61

mlr_context_torch . . . 63
mlr_learners.ft_transformer . . . 66
mlr_learners.mlp . . . 68
mlr_learners.module . . . 71
mlr_learners.realmlp . . . 74
mlr_learners.tab_resnet . . . 78
mlr_learners.tabm . . . 80
mlr_learners.torch_featureless . . . 84
mlr_learners.torchvision . . . 86
mlr_learners_torch . . . 90
mlr_learners_torch_image . . . 102
mlr_learners_torch_model . . . 104
mlr_measures_torch . . . 107
mlr_measures_torch.default . . . 108
mlr_pipeops_augment_center_crop . . . 109
mlr_pipeops_augment_color_jitter . . . 110
mlr_pipeops_augment_crop . . . 111
mlr_pipeops_augment_hflip . . . 111
mlr_pipeops_augment_random_affine . . . 112
mlr_pipeops_augment_random_choice . . . 113
mlr_pipeops_augment_random_crop . . . 114
mlr_pipeops_augment_random_horizontal_flip . . . 114
mlr_pipeops_augment_random_order . . . 115
mlr_pipeops_augment_random_resized_crop . . . 116
mlr_pipeops_augment_random_vertical_flip . . . 116
mlr_pipeops_augment_resized_crop . . . 117
mlr_pipeops_augment_rotate . . . 118
mlr_pipeops_augment_vflip . . . 119
mlr_pipeops_module . . . 119
mlr_pipeops_nn_adaptive_avg_pool1d . . . 122
mlr_pipeops_nn_adaptive_avg_pool2d . . . 124
mlr_pipeops_nn_adaptive_avg_pool3d . . . 125
mlr_pipeops_nn_adaptive_max_pool1d . . . 127
mlr_pipeops_nn_adaptive_max_pool2d . . . 128
mlr_pipeops_nn_adaptive_max_pool3d . . . 130
mlr_pipeops_nn_avg_pool1d . . . 131
mlr_pipeops_nn_avg_pool2d . . . 133
mlr_pipeops_nn_avg_pool3d . . . 134
mlr_pipeops_nn_batch_norm1d . . . 136
mlr_pipeops_nn_batch_norm2d . . . 138
mlr_pipeops_nn_batch_norm3d . . . 139
mlr_pipeops_nn_block . . . 141
mlr_pipeops_nn_celu . . . 143
mlr_pipeops_nn_conv_transpose1d . . . 144
mlr_pipeops_nn_conv_transpose2d . . . 146
mlr_pipeops_nn_conv_transpose3d . . . 148
mlr_pipeops_nn_conv1d . . . 149
mlr_pipeops_nn_conv2d . . . 151

mlr_pipeops_nn_conv3d . . . 153
mlr_pipeops_nn_dropout . . . 155
mlr_pipeops_nn_dropout2d . . . 156
mlr_pipeops_nn_dropout3d . . . 158
mlr_pipeops_nn_elu . . . 159
mlr_pipeops_nn_flatten . . . 161
mlr_pipeops_nn_fn . . . 162
mlr_pipeops_nn_ft_cls . . . 164
mlr_pipeops_nn_ft_transformer_block . . . 165
mlr_pipeops_nn_geglu . . . 167
mlr_pipeops_nn_gelu . . . 168
mlr_pipeops_nn_glu . . . 169
mlr_pipeops_nn_group_norm . . . 170
mlr_pipeops_nn_gru . . . 172
mlr_pipeops_nn_hardshrink . . . 173
mlr_pipeops_nn_hardsigmoid . . . 175
mlr_pipeops_nn_hardtanh . . . 176
mlr_pipeops_nn_head . . . 177
mlr_pipeops_nn_identity . . . 179
mlr_pipeops_nn_layer_norm . . . 180
mlr_pipeops_nn_leaky_relu . . . 182
mlr_pipeops_nn_linear . . . 183
mlr_pipeops_nn_log_sigmoid . . . 184
mlr_pipeops_nn_log_softmax . . . 186
mlr_pipeops_nn_lstm . . . 187
mlr_pipeops_nn_max_pool1d . . . 189
mlr_pipeops_nn_max_pool2d . . . 190
mlr_pipeops_nn_max_pool3d . . . 192
mlr_pipeops_nn_merge . . . 193
mlr_pipeops_nn_merge_cat . . . 195
mlr_pipeops_nn_merge_prod . . . 197
mlr_pipeops_nn_merge_sum . . . 198
mlr_pipeops_nn_multihead_attention . . . 199
mlr_pipeops_nn_prelu . . . 202
mlr_pipeops_nn_recurrent . . . 203
mlr_pipeops_nn_reglu . . . 205
mlr_pipeops_nn_relu . . . 206
mlr_pipeops_nn_relu6 . . . 207
mlr_pipeops_nn_reshape . . . 208
mlr_pipeops_nn_rnn . . . 210
mlr_pipeops_nn_rrelu . . . 212
mlr_pipeops_nn_selu . . . 213
mlr_pipeops_nn_sigmoid . . . 214
mlr_pipeops_nn_silu . . . 216
mlr_pipeops_nn_softmax . . . 217
mlr_pipeops_nn_softmax2d . . . 218
mlr_pipeops_nn_softmin . . . 220
mlr_pipeops_nn_softplus . . . 221

mlr_pipeops_nn_softshrink . . . 222
mlr_pipeops_nn_softsign . . . 224
mlr_pipeops_nn_squeeze . . . 225
mlr_pipeops_nn_tanh . . . 226
mlr_pipeops_nn_tanhshrink . . . 228
mlr_pipeops_nn_threshold . . . 229
mlr_pipeops_nn_tokenizer_categ . . . 230
mlr_pipeops_nn_tokenizer_num . . . 232
mlr_pipeops_nn_transformer_encoder_layer . . . 233
mlr_pipeops_nn_unflatten . . . 236
mlr_pipeops_nn_unsqueeze . . . 238
mlr_pipeops_preproc_torch . . . 239
mlr_pipeops_torch . . . 244
mlr_pipeops_torch_callbacks . . . 250
mlr_pipeops_torch_ingress . . . 251
mlr_pipeops_torch_ingress_categ . . . 253
mlr_pipeops_torch_ingress_ltnsr . . . 255
mlr_pipeops_torch_ingress_num . . . 257
mlr_pipeops_torch_loss . . . 259
mlr_pipeops_torch_model . . . 260
mlr_pipeops_torch_model_classif . . . 265
mlr_pipeops_torch_model_regr . . . 266
mlr_pipeops_torch_optimizer . . . 268
mlr_pipeops_trafo_adjust_brightness . . . 269
mlr_pipeops_trafo_adjust_gamma . . . 270
mlr_pipeops_trafo_adjust_hue . . . 271
mlr_pipeops_trafo_adjust_saturation . . . 271
mlr_pipeops_trafo_grayscale . . . 272
mlr_pipeops_trafo_nop . . . 273
mlr_pipeops_trafo_normalize . . . 273
mlr_pipeops_trafo_pad . . . 274
mlr_pipeops_trafo_reshape . . . 274
mlr_pipeops_trafo_resize . . . 275
mlr_pipeops_trafo_rgb_to_grayscale . . . 276
mlr_tasks_cifar . . . 276
mlr_tasks_lazy_iris . . . 277
mlr_tasks_melanoma . . . 278
mlr_tasks_mnist . . . 280
mlr_tasks_tiny_imagenet . . . 281
mlr_tasks_torch . . . 282
mlr3torch_callbacks . . . 284
mlr3torch_losses . . . 285
mlr3torch_optimizers . . . 285
model_descriptor_to_learner . . . 286
model_descriptor_to_module . . . 287
model_descriptor_union . . . 288
ModelDescriptor . . . 289
msr_torch . . . 290

nn . . . 292
nn_ft_cls . . . 293
nn_ft_transformer_block . . . 293
nn_geglu . . . 295
nn_graph . . . 296
nn_merge_cat . . . 297
nn_merge_prod . . . 298
nn_merge_sum . . . 298
nn_reglu . . . 298
nn_reshape . . . 299
nn_squeeze . . . 299
nn_tokenizer_categ . . . 300
nn_tokenizer_num . . . 300
nn_unsqueeze . . . 301
output_dim_for . . . 301
pipeop_preproc_torch . . . 302
pipeop_torch . . . 303
PredictionTorch . . . 306
reshape_output_shape . . . 308
Select . . . 309
shape_to_str . . . 310
t_clbk . . . 311
t_loss . . . 312
t_opt . . . 313
task_dataset . . . 314
torch_callback . . . 315
TorchCallback . . . 318
TorchDescriptor . . . 321
TorchIngressToken . . . 323
TorchLoss . . . 325
TorchOptimizer . . . 327

Index **331**

`mlr3torch-package` *mlr3torch: Deep Learning with 'mlr3'*

Description

Deep Learning library that extends the mlr3 framework by building upon the 'torch' package. It makes it convenient to build, train, and evaluate deep learning models without having to worry about low level details. Custom architectures can be created using the graph language defined in 'mlr3pipelines'.

Options

- `mlr3torch.cache`: Whether to cache the downloaded data (`TRUE`) or not (`FALSE`, default). This can also be set to a specific folder on the file system to be used as the cache directory. Without caching, every task that downloads its data fetches it again in each session.

Author(s)

Maintainer: Sebastian Fischer <sebf.fischer@gmail.com> (ORCID)

Authors:

- Sebastian Fischer <sebf.fischer@gmail.com> (ORCID)
- Martin Binder <mlr.developer@mb706.com>

Other contributors:

- Bernd Bischl <bernd_bischl@gmx.net> (ORCID) [contributor]
- Lukas Burk <github@quantenbrot.de> (ORCID) [contributor]
- Florian Pfisterer <pfistererf@googlemail.com> (ORCID) [contributor]
- Carson Zhang <carsonzhang4@gmail.com> [contributor]

See Also

Useful links:

- https://mlr3torch.mlr-org.com/
- https://github.com/mlr-org/mlr3torch/
- Report bugs at https://github.com/mlr-org/mlr3torch/issues

---

as_data_descriptor *Convert to Data Descriptor*

Description

Converts the input to a DataDescriptor.

Usage

```
as_data_descriptor(x, dataset_shapes, ...)
```

Arguments

x	(any) Object to convert.
dataset_shapes	(named list() of (integer() or NULL)) The shapes of the output. Names are the elements of the list returned by the dataset. If the shape is not NULL (unknown, e.g. for images of different sizes) the first dimension must be NA to indicate the batch dimension.
...	(any) Further arguments passed to the DataDescriptor constructor.

Examples

```
ds = dataset("example",
  initialize = function() self$iris = iris[, -5],
  .getitem = function(i) list(x = torch_tensor(as.numeric(self$iris[i, ]))),
  .length = function() nrow(self$iris)
)()
as_data_descriptor(ds, list(x = c(NA, 4L)))

# if the dataset has a .getbatch method, the shapes are inferred
ds2 = dataset("example",
  initialize = function() self$iris = iris[, -5],
  .getbatch = function(i) list(x = torch_tensor(as.matrix(self$iris[i, ]))),
  .length = function() nrow(self$iris)
)()
as_data_descriptor(ds2)
```

---

as_lazy_tensor *Convert to Lazy Tensor*

Description

Converts an object to a lazy_tensor. A dataset is wrapped as-is, a torch_tensor or a numeric() is first turned into a dataset whose first dimension indexes the observations.

Usage

```
as_lazy_tensor(x, ...)

## S3 method for class 'dataset'
as_lazy_tensor(x, dataset_shapes = NULL, ids = NULL, ...)
```

Arguments

x	(any) Object to convert to a lazy_tensor
...	(any) Additional arguments passed to the method.
dataset_shapes	(named list() of (integer() or NULL)) The shapes of the output. Names are the elements of the list returned by the dataset. If the shape is not NULL (unknown, e.g. for images of different sizes) the first dimension must be NA to indicate the batch dimension.
ids	(integer()) Which ids to include in the lazy tensor.

Examples

```
iris_ds = dataset("iris",
  initialize = function() {
    self$iris = iris[, -5]
  },
  .getbatch = function(i) {
    list(x = torch_tensor(as.matrix(self$iris[i, ])))
  },
  .length = function() nrow(self$iris)
)()
# no need to specify the dataset shapes as they can be inferred from the .getbatch method
# only first 5 observations
as_lazy_tensor(iris_ds, ids = 1:5)
# all observations
head(as_lazy_tensor(iris_ds))

iris_ds2 = dataset("iris",
  initialize = function() self$iris = iris[, -5],
  .getitem = function(i) list(x = torch_tensor(as.numeric(self$iris[i, ]))),
  .length = function() nrow(self$iris)
)()
# if .getitem is implemented we cannot infer the shapes as they might vary,
# so we have to annotate them explicitly
as_lazy_tensor(iris_ds2, dataset_shapes = list(x = c(NA, 4L)))[1:5]

# Convert a matrix
lt = as_lazy_tensor(matrix(rnorm(100), nrow = 20))
materialize(lt[1:5], rbind = TRUE)
```

---

as_learner_torch *Convert a Graph to a Torch Learner*

Description

Converts a Graph representing a deep learning pipeline into a GraphLearnerTorch. The graph either builds the network from PipeOpTorch operators and ends in a PipeOpTorchModel, or its base learner is a LearnerTorch, e.g. `po("scale") %>>% lrn("classif.mlp")`. The advantage over using as_learner() is that the resulting learner exposes methods like `$dataset()` and fields like `$network`.

Usage

```
as_learner_torch(x, ...)

## S3 method for class 'Graph'
as_learner_torch(x, id = NULL, ...)

## S3 method for class 'PipeOp'
```

```
as_learner_torch(x, ...)

## S3 method for class 'GraphLearner'
as_learner_torch(x, id = x$id, ...)
```

Arguments

x	(any) The object to convert, e.g. a Graph.
...	(any) Unused.
id	(character(1)) The id of the learner. Defaults to "<task_type>.graph".

Value

GraphLearner

See Also

Other Graph Network: ModelDescriptor(), TorchIngressToken(), mlr_learners_torch_model, mlr_pipeops_module, mlr_pipeops_torch, mlr_pipeops_torch_ingress, mlr_pipeops_torch_ingress_categ, mlr_pipeops_torch_ingress_ltnsr, mlr_pipeops_torch_ingress_num, model_descriptor_to_learner(), model_descriptor_to_module(), model_descriptor_union(), nn_graph(), pipeop_torch()

Other Learner: GraphLearnerTorch, mlr_learners.ft_transformer, mlr_learners.mlp, mlr_learners.module, mlr_learners.realmlp, mlr_learners.tab_resnet, mlr_learners.tabm, mlr_learners.torch_featureless, mlr_learners_torch, mlr_learners_torch_image, mlr_learners_torch_model

Examples

```
graph = po("scale") %>>%
  po("torch_ingress_num") %>>%
  nn("linear", out_features = 10) %>>%
  nn("relu") %>>%
  nn("head") %>>%
  po("torch_loss", "cross_entropy") %>>%
  po("torch_optimizer", "adam", lr = 0.1) %>>%
  po("torch_model_classif", epochs = 1, batch_size = 32)

learner = as_learner_torch(graph)
learner$id
learner$param_set$set_values(linear.out_features = 20, torch_optimizer.lr = 0.01)

task = tsk("iris")
learner$train(task)
learner$network
learner$predict(task)

learner$dataset(task, "train")
```

as_lr_scheduler *Convert to CallbackSetLRScheduler*

Description

Convert a `torch` scheduler generator to a `CallbackSetLRScheduler`.

Usage

```
as_lr_scheduler(x, step_on_epoch)
```

Arguments

`x`	(`function`) The `torch` scheduler generator defined using `torch::lr_scheduler()`.
`step_on_epoch`	(`logical(1)`) Whether the scheduler steps after every epoch

as_task_torch *Create a Generic Torch Task*

Description

Creates a `TaskTorch`, the general-purpose task type of `mlr3torch`, from a `data.frame()` or a `DataBackend`. See the *Custom Learning Problems* article for more information.

Usage

```
as_task_torch(x, target = NULL, id = deparse(substitute(x))[1L], ...)
```

Arguments

`x`	(`data.frame()` or `DataBackend`) The data.
`target`	(`character()` or `NULL`) The names of the target columns. `NULL` (default) for a task without a target, see section *Tasks without a Target* of `TaskTorch`.
`id`	(`character(1)`) The id of the task.
`...`	(any) Further arguments passed to `TaskTorch$new()`, such as `output_dim`, `default_encoder`, `default_target_batchgetter` or `default_measure`.

Value

TaskTorch

Examples

```
# multi-output regression
d = data.frame(x = rnorm(50))
d$y1 = d$x + rnorm(50)
d$y2 = 2 * d$x + rnorm(50)
as_task_torch(d, target = c("y1", "y2"))

# unsupervised: no target at all
as_task_torch(data.frame(a = rnorm(50), b = rnorm(50)))
```

---

as_torch_callback *Convert to a TorchCallback*

Description

Converts an object to a TorchCallback.

Usage

```
as_torch_callback(x, clone = FALSE, ...)
```

Arguments

x	(any) Object to be converted.
clone	(logical(1)) Whether to make a deep clone.
...	(any) Additional arguments

Value

TorchCallback.

See Also

Other Callback: TorchCallback, as_torch_callbacks(), callback_set(), mlr3torch_callbacks, mlr_callback_set, mlr_callback_set.checkpoint, mlr_callback_set.progress, mlr_callback_set.tb, mlr_callback_set.unfreeze, mlr_context_torch, t_clbk(), torch_callback()

as_torch_callbacks *Convert to a list of Torch Callbacks*

Description

Converts an object to a list of TorchCallback.

Usage

```
as_torch_callbacks(x, clone, ...)
```

Arguments

x	(any) Object to convert.
clone	(logical(1)) Whether to create a deep clone.
...	(any) Additional arguments.

Value

list() of TorchCallbacks

See Also

Other Callback: TorchCallback, as_torch_callback(), callback_set(), mlr3torch_callbacks, mlr_callback_set, mlr_callback_set.checkpoint, mlr_callback_set.progress, mlr_callback_set.tb, mlr_callback_set.unfreeze, mlr_context_torch, t_clbk(), torch_callback()

Other Torch Descriptor: TorchCallback, TorchDescriptor, TorchLoss, TorchOptimizer, as_torch_loss(), as_torch_optimizer(), mlr3torch_losses, mlr3torch_optimizers, t_clbk(), t_loss(), t_opt()

as_torch_loss *Convert to TorchLoss*

Description

Converts an object to a TorchLoss.

Usage

```
as_torch_loss(x, clone = FALSE, ...)
```

Arguments

x	(any) Object to convert to a TorchLoss.
clone	(logical(1)) Whether to make a deep clone.
...	(any) Additional arguments. Currently used to pass additional constructor arguments to TorchLoss for objects of type nn_loss.

Value

TorchLoss.

See Also

Other Torch Descriptor: TorchCallback, TorchDescriptor, TorchLoss, TorchOptimizer, as_torch_callbacks(), as_torch_optimizer(), mlr3torch_losses, mlr3torch_optimizers, t_clbk(), t_loss(), t_opt()

Examples

```
# Define a custom loss, here the quantile (pinball) loss for quantile regression.
# The class must contain "nn_loss", as this is what as_torch_loss() dispatches on.
nn_quantile_loss = nn_module(c("nn_quantile_loss", "nn_loss"),
  initialize = function(q = 0.5) {
    self$q = q
  },
  forward = function(input, target) {
    d = target - input
    torch_mean(torch_max(self$q * d, other = (self$q - 1) * d))
  }
)

# additional arguments are passed to the TorchLoss constructor
quantile_loss = as_torch_loss(nn_quantile_loss, task_types = "regr")
quantile_loss
# the parameters are inferred from the loss' initialize method
quantile_loss$param_set
# and can be configured when using the loss in a learner
lrn("regr.mlp", loss = quantile_loss, loss.q = 0.9)

# predefined losses can be converted from their key,
# this is the same as t_loss("mse")
as_torch_loss("mse")

# TorchLosses are returned as-is, unless clone is TRUE
loss = t_loss("mse")
identical(as_torch_loss(loss), loss)
identical(as_torch_loss(loss, clone = TRUE), loss)
```

`as_torch_optimizer` *Convert to TorchOptimizer*

Description

Converts an object to a `TorchOptimizer`.

Usage

```
as_torch_optimizer(x, clone = FALSE, ...)
```

Arguments

`x`	(any) Object to convert to a `TorchOptimizer`.
`clone`	(`logical(1)`) Whether to make a deep clone. Default is `FALSE`.
`...`	(any) Additional arguments. Currently used to pass additional constructor arguments to `TorchOptimizer` for objects of type `torch_optimizer_generator`.

Value

`TorchOptimizer`

See Also

Other Torch Descriptor: `TorchCallback`, `TorchDescriptor`, `TorchLoss`, `TorchOptimizer`, `as_torch_callbacks()`, `as_torch_loss()`, `mlr3torch_losses`, `mlr3torch_optimizers`, `t_clbk()`, `t_loss()`, `t_opt()`

Examples

```
# convert a `torch::torch_optimizer_generator`
as_torch_optimizer(optim_adamw)
# the id defaults to the name of the generator, but can be overwritten
as_torch_optimizer(optim_adamw, id = "my_adamw", label = "My AdamW")

# convert a key of mlr3torch_optimizers, this is the same as t_opt("adamw")
as_torch_optimizer("adamw")

# TorchOptimizers are returned as-is, unless clone is TRUE
opt = t_opt("adamw")
identical(as_torch_optimizer(opt), opt)
identical(as_torch_optimizer(opt, clone = TRUE), opt)
```

assert_dim_in_range *Assert that a Dimension Exists*

Description

Rejects a `dim` parameter that does not address a dimension of `shape`. Resolve negative indices with `resolve_dim()` first and pass both, so that the error message can report the value the user actually specified.

Usage

```
assert_dim_in_range(dim, true_dim, shape, id)
```

Arguments

`dim`	(`integer(1)`) The dimension as the user specified it: it may count back from the last dimension. Only used for the error message.
`true_dim`	(`integer(1)`) The same dimension resolved to a positive index, see `resolve_dim()`.
`shape`	(`integer()`) The shape that `true_dim` addresses.
`id`	(`character(1)`) The id of the `PipeOp`, which the error message names.

Value

The resolved dimension, invisibly.

See Also

Other Shape Assertions: `assert_known_dims()`, `assert_ndim()`, `assert_not_batch_dim()`, `assert_positive_extent()`, `assert_same_batch_size()`, `assert_same_ndim()`, `assert_shape()`, `assert_shapes()`

Examples

```
shape = c(NA, 3, 8, 8)
assert_dim_in_range(-1L, resolve_dim(-1L, shape), shape, id = "nn_squeeze")
try(assert_dim_in_range(-5L, resolve_dim(-5L, shape), shape, id = "nn_squeeze"))
```

assert_known_dims *Assert that Dimensions are Known*

Description

Ensure that a specific dimension is known, i.e. not NA.

Usage

```
assert_known_dims(shape, dims, what, id = NULL)
```

Arguments

shape	(integer()) The input shape.
dims	(integer()) Indices of the dimensions that must be known.
what	(character(1)) Describes those dimensions in the error message, e.g. "the channel dimension (dimension 2)".
id	(character(1) | NULL) The id of the PipeOp the assertion is made for, which the error message names. NULL when the assertion is not made for a PipeOp.

Value

The shape, invisibly.

See Also

Other Shape Assertions: assert_dim_in_range(), assert_ndim(), assert_not_batch_dim(), assert_positive_extent(), assert_same_batch_size(), assert_same_ndim(), assert_shape(), assert_shapes()

Examples

```
# a convolution needs the number of input channels, but not the spatial extent
assert_known_dims(c(NA, 3, NA, NA), 2, "the number of channels", id = "nn_conv2d")
try(assert_known_dims(c(NA, NA, 10, 10), 2, "the number of channels", id = "nn_conv2d"))
```

assert_lazy_tensor *Assert Lazy Tensor*

Description

Asserts whether something is a lazy tensor.

Usage

```
assert_lazy_tensor(x)
```

Arguments

x	(any) Object to check.

assert_ndim *Assert the Number of Dimensions of a Shape*

Description

Rejects a shape with the wrong number of dimensions. Give either `ndim`, the number(s) of dimensions the operator accepts, or the bounds `min` and `max` (either on its own), which is what an operator that accepts a range of them uses, as batch normalization does.

Usage

```
assert_ndim(shape, ndim = NULL, id, min = NULL, max = NULL)
```

Arguments

shape	(`integer()`) The input shape, with `NA` for the dimensions whose size is unknown.
ndim	(`integer()`) The number(s) of dimensions the operator accepts, the batch dimension included. Alternatively give `min` and/or `max`.
id	(`character(1)`) The id of the `PipeOp`, which the error message names.
min	(`integer(1)`) The smallest number of dimensions the operator accepts, `NULL` for no lower bound.
max	(`integer(1)`) The largest number of dimensions the operator accepts, `NULL` for no upper bound.

Value

The shape, invisibly.

See Also

Other Shape Assertions: assert_dim_in_range(), assert_known_dims(), assert_not_batch_dim(), assert_positive_extent(), assert_same_batch_size(), assert_same_ndim(), assert_shape(), assert_shapes()

Examples

```
assert_ndim(c(NA, 3, 32, 32), 4L, id = "nn_conv2d")
try(assert_ndim(c(NA, 3), 4L, id = "nn_conv2d"))
# batch normalization accepts a range
try(assert_ndim(c(NA, 3, 4, 5), id = "nn_batch_norm1d", min = 2L, max = 3L))
```

---

assert_not_batch_dim *Assert that a Dimension is not the Batch Dimension*

Description

Rejects an operation on the first dimension, which is the batch dimension. An operator that changes it would silently change the number of observations, which fails much later with a mismatch against the target.

Usage

```
assert_not_batch_dim(dim, shape, id)
```

Arguments

dim	(integer(1)) The resolved dimension that the operator changes, see resolve_dim().
shape	(integer()) The input shape, used for the error message.
id	(character(1)) The id of the PipeOp, which the error message names.

Value

The dimension, invisibly.

See Also

Other Shape Assertions: assert_dim_in_range(), assert_known_dims(), assert_ndim(), assert_positive_extent(), assert_same_batch_size(), assert_same_ndim(), assert_shape(), assert_shapes()

Examples

```
assert_not_batch_dim(2L, c(NA, 3, 8), id = "nn_squeeze")
try(assert_not_batch_dim(1L, c(NA, 3, 8), id = "nn_squeeze"))
```

`assert_positive_extent` *Assert that a Computed Output Size is Positive*

Description

Rejects a computed output size that no tensor can have, i.e. one that is not a positive number. Operators such as convolutions and pooling compute their spatial output sizes from `kernel_size`, `stride`, `padding` and `dilation`, a combination of which can produce a size of zero or less. Unknown (`NA`) sizes are accepted, since nothing can be said about them.

Usage

```
assert_positive_extent(extent, shape_in, id)
```

Arguments

`extent`	(`numeric()`) The sizes an operator computed for the dimensions it changes, e.g. the spatial dimensions of a convolution.
`shape_in`	(`integer()`) The input shape, used for the error message.
`id`	(`character(1)` | `NULL`) The id of the `PipeOp`, which the error message names. `NULL` when the assertion is not made for a `PipeOp`.

Value

The extent, invisibly.

See Also

Other Shape Assertions: `assert_dim_in_range()`, `assert_known_dims()`, `assert_ndim()`, `assert_not_batch_dim()`, `assert_same_batch_size()`, `assert_same_ndim()`, `assert_shape()`, `assert_shapes()`

Examples

```
assert_positive_extent(c(30, NA), c(NA, 3, 32, 32), id = "nn_conv2d")
try(assert_positive_extent(c(0, 4), c(NA, 3, 32, 32), id = "nn_conv2d"))
```

assert_same_batch_size *Assert that Shapes have the Same Batch Size*

Description

Rejects inputs whose batch sizes, i.e. first dimensions, disagree. An unknown (NA) batch size is compatible with any other, so only the known ones have to agree.

Unlike the other assertions this returns the common batch size rather than its input, because that is what the caller needs next: an operator that drops the batch dimension while it works has to put it back afterwards.

Usage

```
assert_same_batch_size(shapes, id)
```

Arguments

shapes	(list() of integer()) The input shapes.
id	(character(1)) The id of the PipeOp, which the error message names.

Value

(integer(1)) The common batch size, invisibly, or NA_integer_ if no input has a known one.

See Also

Other Shape Assertions: assert_dim_in_range(), assert_known_dims(), assert_ndim(), assert_not_batch_dim(), assert_positive_extent(), assert_same_ndim(), assert_shape(), assert_shapes()

Examples

```
assert_same_batch_size(list(c(8, 3), c(8, 5)), id = "nn_block")
# an unknown batch size is compatible with a known one, which is the one that is returned
assert_same_batch_size(list(c(NA, 3), c(8, 5)), id = "nn_block")
try(assert_same_batch_size(list(c(8, 3), c(4, 5)), id = "nn_block"))
```

assert_same_ndim *Assert that Shapes have the Same Number of Dimensions*

Description

Rejects inputs that do not all have the same number of dimensions, which the operators that combine several inputs require: shorter shapes are not left-padded with 1s, because the first dimension is the batch dimension.

Usage

```
assert_same_ndim(shapes, id)
```

Arguments

shapes	(list() of integer()) The input shapes.
id	(character(1)) The id of the PipeOp, which the error message names.

Value

The shapes, invisibly.

See Also

Other Shape Assertions: assert_dim_in_range(), assert_known_dims(), assert_ndim(), assert_not_batch_dim(), assert_positive_extent(), assert_same_batch_size(), assert_shape(), assert_shapes()

Examples

```
assert_same_ndim(list(c(NA, 3), c(NA, 5)), id = "nn_merge_cat")
try(assert_same_ndim(list(c(NA, 3), c(NA, 5, 2)), id = "nn_merge_cat"))
```

assert_shape *Assert a Shape*

Description

Checks that shape is a valid shape, i.e. an integer() with at least one dimension, NA where a dimension is unknown. See the "Shape Inference" section of PipeOpTorch for the conventions.

Usage

```
assert_shape(
  shape,
  null_ok = FALSE,
  coerce = TRUE,
  unknown_batch = NULL,
  len = NULL
)
```

Arguments

shape	(integer()) A shape, with NA for the dimensions whose size is unknown.
null_ok	(logical(1)) Whether NULL, i.e. a wholly unknown shape, is valid.
coerce	(logical(1)) Whether to coerce the input to an integer() if possible.
unknown_batch	(logical(1) | NULL) Whether the batch dimension **must** be unknown, i.e. NA. If left NULL (default), the first dimension can be NA or not.
len	(integer(1)) The required number of dimensions.

Value

The shape, coerced to an integer() if coerce is TRUE.

See Also

Other Shape Assertions: assert_dim_in_range(), assert_known_dims(), assert_ndim(), assert_not_batch_dim(), assert_positive_extent(), assert_same_batch_size(), assert_same_ndim(), assert_shapes()

Examples

```
assert_shape(c(NA, 3, 32, 32))
try(assert_shape("not a shape"))
```

---

assert_shapes *Assert a List of Shapes*

Description

Checks that shapes is a non-empty list() of valid shapes, see assert_shape().

Usage

```
assert_shapes(
  shapes,
  coerce = TRUE,
  named = FALSE,
  null_ok = FALSE,
  unknown_batch = NULL
)
```

Arguments

shapes	(list() of integer()) A list() of shapes.
coerce	(logical(1)) Whether to coerce the shapes to integer() if possible.
named	(logical(1)) Whether the shapes must be uniquely named.
null_ok	(logical(1)) Whether NULL, i.e. a wholly unknown shape, is valid.
unknown_batch	(logical(1) | NULL) Whether the batch dimension **must** be unknown, i.e. NA. If left NULL (default), the first dimension can be NA or not.

Value

The shapes, coerced to integer() if coerce is TRUE.

See Also

Other Shape Assertions: assert_dim_in_range(), assert_known_dims(), assert_ndim(), assert_not_batch_dim(), assert_positive_extent(), assert_same_batch_size(), assert_same_ndim(), assert_shape()

Examples

```
assert_shapes(list(c(NA, 3), c(NA, 5)))
```

---

auto_device *Auto Device*

Description

Resolves the device parameter of a learner. "auto" becomes "cuda" when a CUDA device is available and "cpu" otherwise; any other value is returned unchanged, except that an explicit "cuda" (or "cuda:<index>") without an available CUDA device with that index is an error rather than a failure deep inside libtorch later on.

Usage

```
auto_device(device = NULL)
```

Arguments

device	(character(1)) The device. If not NULL, is returned as is.

---

batchgetter_categ *Batchgetter for Categorical data*

Description

Converts a data frame of categorical data into a long tensor by converting the data to integers. No input checks are performed.

All columns are encoded with **1-based** codes, i.e. the values of a feature with cardinality k are 1, ..., k.

Usage

```
batchgetter_categ(data, ...)
```

Arguments

data	(data.table) data.table to be converted to a tensor.
...	(any) Unused.

---

batchgetter_num *Batchgetter for Numeric Data*

Description

Converts a data frame of numeric data into a float tensor by calling as.matrix(). No input checks are performed

Usage

```
batchgetter_num(data, ...)
```

Arguments

data	(data.table()) data.table to be converted to a tensor.
...	(any) Unused.

callback_set *Create a Set of Callbacks for Torch*

Description

Creates an `R6ClassGenerator` inheriting from `CallbackSet`. Additionally performs checks such as that the stages are not accidentally misspelled. To create a `TorchCallback` use `torch_callback()`.

In order for the resulting class to be cloneable, the private method `$deep_clone()` must be provided.

Usage

```
callback_set(
  classname,
  on_begin = NULL,
  on_end = NULL,
  on_exit = NULL,
  on_epoch_begin = NULL,
  on_before_valid = NULL,
  on_epoch_end = NULL,
  on_batch_begin = NULL,
  on_batch_end = NULL,
  on_after_backward = NULL,
  on_batch_valid_begin = NULL,
  on_batch_valid_end = NULL,
  on_valid_end = NULL,
  state_dict = NULL,
  load_state_dict = NULL,
  initialize = NULL,
  weight = NULL,
  public = NULL,
  private = NULL,
  active = NULL,
  parent_env = parent.frame(),
  inherit = CallbackSet,
  lock_objects = FALSE
)
```

Arguments

`classname` (`character(1)`)
The class name.

`on_begin`, `on_end`, `on_epoch_begin`, `on_before_valid`, `on_epoch_end`, `on_batch_begin`, `on_batch_end`, `on_after_backward`, `on_batch_valid_begin`, `on_batch_valid_end`, `on_valid_end`, `on_exit` (`function`)
Function to execute at the given stage, see section *Stages*.

`state_dict` (`function()`)
The function that retrieves the state dict from the callback. This is what will be available in the learner after training.

`load_state_dict` (`function(state_dict)`)
Function that loads a callback state.

`initialize` (`function()`)
The initialization method of the callback.

`weight` (`numeric(1)`)
Controls when the callback is called within a stage, see section *Ordering* of `CallbackSet`. Defaults to `0`. The name is reserved for this, so `initialize` must not take a `weight` argument: wrapping such a callback in a `TorchCallback` is an error.

`public`, `private`, `active` (`list()`)
Additional public, private, and active fields to add to the callback.

`parent_env` (`environment()`)
The parent environment for the `R6Class`.

`inherit` (`R6ClassGenerator`)
From which class to inherit. This class must either be `CallbackSet` (default) or inherit from it.

`lock_objects` (`logical(1)`)
Whether to lock the objects of the resulting `R6Class`. If `FALSE` (default), values can be freely assigned to `self` without declaring them in the class definition.

Value

`CallbackSet`

See Also

Other Callback: `TorchCallback`, `as_torch_callback()`, `as_torch_callbacks()`, `mlr3torch_callbacks`, `mlr_callback_set`, `mlr_callback_set.checkpoint`, `mlr_callback_set.progress`, `mlr_callback_set.tb`, `mlr_callback_set.unfreeze`, `mlr_context_torch`, `t_clbk()`, `torch_callback()`

---

cross_entropy *Cross Entropy Loss*

Description

The `cross_entropy` loss function selects the multi-class (`nn_cross_entropy_loss`) or binary (`nn_bce_with_logits_loss`) cross entropy loss based on the number of classes. Because of this, there is a slight reparameterization of the loss arguments, see *Parameters*.

Parameters

- class_weight:: torch_tensor
  The class weights. For multi-class problems, this must be a torch_tensor of length num_classes (and is passed as argument weight to nn_cross_entropy_loss). For binary problems, this must be a scalar (and is passed as argument pos_weight to nn_bce_with_logits_loss).

- ignore_index:: integer(1)
  Index of the class which to ignore and which does not contribute to the gradient. This is only available for multi-class loss.

- reduction :: character(1)
  The reduction to apply. Is either "mean" or "sum" and passed as argument reduction to either loss function. The default is "mean".

Examples

```
loss = t_loss("cross_entropy")
# multi-class
multi_ce = loss$generate(tsk("iris"))
multi_ce

# binary
binary_ce = loss$generate(tsk("sonar"))
binary_ce
```

DataDescriptor *Data Descriptor*

Description

A data descriptor is a rather internal data structure used in the lazy_tensor data type. In essence it is an annotated torch::dataset and a preprocessing graph (consisting mosty of PipeOpModule operators). The additional meta data (e.g. pointer, shapes) makes it possible to preprocess lazy_tensors in an mlr3pipelines::Graph just like any (non-lazy) data types. The preprocessing is applied when materialize() is called on the lazy_tensor.

To create a data descriptor, you can also use the as_data_descriptor() function.

Details

While it would be more natural to define this as an S3 class, we opted for an R6 class to avoid the usual trouble of serializing S3 objects. If each row contained a DataDescriptor as an S3 class, this would copy the object when serializing.

Public fields

`dataset` (`torch::dataset`)
The dataset.

`graph` (`Graph`)
The preprocessing graph.

`dataset_shapes` (named `list()` of (`integer()` or `NULL`))
The shapes of the output.

`input_map` (`character()`)
The input map from the dataset to the preprocessing graph.

`pointer` (`character(2)`)
The output pointer.

`pointer_shape` (`integer()` | `NULL`)
The shape of the output indicated by `pointer`.

`dataset_hash` (`character(1)`)
Hash for the wrapped dataset.

`hash` (`character(1)`)
Hash for the data descriptor.

`graph_input` (`character()`)
The input channels of the preprocessing graph (cached to save time).

`pointer_shape_predict` (`integer()` or `NULL`)
Internal use only.

Methods

Public methods:

- `DataDescriptor$new()`
- `DataDescriptor$print()`
- `DataDescriptor$clone()`

`DataDescriptor$new()`: Creates a new instance of this R6 class.

Usage:

```
DataDescriptor$new(
  dataset,
  dataset_shapes = NULL,
  graph = NULL,
  input_map = NULL,
  pointer = NULL,
  pointer_shape = NULL,
  pointer_shape_predict = NULL,
  clone_graph = TRUE
)
```

Arguments:

`dataset` (`torch::dataset`)
The torch dataset. It should return a named `list()` of `torch_tensor` objects.

`dataset_shapes` (named `list()` of (`integer()` or `NULL`))
: The shapes of the output. Names are the elements of the list returned by the dataset. If the shape is not `NULL` (unknown, e.g. for images of different sizes) the first dimension must be `NA` to indicate the batch dimension.

`graph` (`Graph`)
: The preprocessing graph. If left `NULL`, no preprocessing is applied to the data and `input_map`, `pointer`, `pointer_shape`, and `pointer_shape_predict` are inferred in case the dataset returns only one element.

`input_map` (`character()`)
: Character vector that must have the same length as the input of the graph. Specifies how the data from the `dataset` is fed into the preprocessing graph.

`pointer` (`character(2)` | `NULL`)
: Points to an output channel within `graph`: Element 1 is the `PipeOp`'s id and element 2 is that `PipeOp`'s output channel.

`pointer_shape` (`integer()` | `NULL`)
: Shape of the output indicated by `pointer`.

`pointer_shape_predict` (`integer()` or `NULL`)
: Internal use only. Used in a `Graph` to anticipate possible mismatches between train and predict shapes.

`clone_graph` (`logical(1)`)
: Whether to clone the preprocessing graph.

`DataDescriptor$print()`: Prints the object

Usage:

```
DataDescriptor$print(...)
```

Arguments:

`...` (any)
: Unused

`DataDescriptor$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DataDescriptor$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

ModelDescriptor, lazy_tensor

Examples

```
# Create a dataset
ds = dataset(
  initialize = function() self$x = torch_randn(10, 3, 3),
  .getitem = function(i) list(x = self$x[i, ]),
  .length = function() nrow(self$x)
)()
```

```
dd = DataDescriptor$new(ds, list(x = c(NA, 3, 3)))
dd
# is the same as using the converter:
as_data_descriptor(ds, list(x = c(NA, 3, 3)))
```

`encode_prediction` *Encode the Network Output as a Prediction*

Description

Converts the raw output of a network into a `list()` that can be passed to `mlr3::as_prediction_data()`, which is what the private `.encode_prediction()` method of a `LearnerTorch` has to return.

This is the default implementation that is used by `LearnerTorch` and `LearnerTorchModel`, i.e. by all learners that don't overwrite `.encode_prediction()`. When adding support for a custom task type, implement a method for the corresponding `Task` class, which makes the generic torch learners work for that task type.

For the network output that is expected for the built-in task types, see section *Network Head and Target Encoding* of `LearnerTorch`.

Usage

```
encode_prediction(task, network_output, predict_type, ...)
```

Arguments

`task`	(`Task`) The task to predict on. For classification, a `LearnerTorch` passes it with the classes in the order of its training task, which is the order that the columns of `network_output` refer to.
`network_output`	(`torch_tensor` or `list()` of them) The raw output of the network in evaluation mode. A network with more than one head – e.g. one predicting a mean and a standard deviation – returns a `list()` of tensors, which is passed on unchanged. The encodings of the built-in task types expect a single tensor.
`predict_type`	(`character(1)`) The predict type of the learner, e.g. `"response"` or `"prob"`.
`...`	(any) Additional arguments. Not used yet.

Value

named `list()`

get_batch_constructor *Batch Constructor for a Task*

Description

Returns the function that builds a whole batch of a `task`, i.e. both the features `x` and the target `y`. This is how `task_dataset` loads its batches.

The returned function takes the arguments

- `data`, a `data.table` with the feature *and* target columns of the batch, and
- `cache`, a rather internal `hashtab` that can be used as a cache when loading multiple `lazy_tensor` columns.

It returns a `list()` with elements `x` (a named `list()` of `torch_tensor`s) and `y` (a `torch_tensor` or `NULL`).

The default method applies the ingress tokens to obtain `x` and the target batchgetter to obtain `y`.

Usage

```
get_batch_constructor(
  task,
  feature_ingress_tokens,
  target_batchgetter = NULL,
  ...
)
```

Arguments

`task`	(`Task`) The task.
`feature_ingress_tokens`	(named `list()` of `TorchIngressToken`) The ingress tokens that define `x`. Their features are already resolved, i.e. they are `character()` vectors and not `Selector`s.
`target_batchgetter`	(`function()` or `NULL`) Converts the target columns of a batch into the target tensor `y` that the loss is applied to. Takes an argument `data`, a `data.table` with only the target columns, and optionally an argument `x`, the named list of feature tensors of the batch, which is what a target that is a function of the input needs, see `get_target_batchgetter()`. If `NULL` (default), the batches have no `y` element.
`...`	(any) Additional arguments. Not used yet.

Value

`function(data, cache) -> list(x = list<torch_tensor>, y = torch_tensor | NULL)`

Examples

```
task = tsk("iris")
token = TorchIngressToken(task$feature_names, batchgetter_num, c(NA, 4))
# the token's features are a `Selector`; `task_dataset()` resolves them for you
token$features = token$features(task)
batch_constructor = get_batch_constructor(
  task,
  feature_ingress_tokens = list(input = token),
  target_batchgetter = get_target_batchgetter(task)
)
batch = batch_constructor(data = task$data(rows = 1:2))
batch$x$input
batch$y
```

---

get_batch_size *Extract the Batch Size for a Given Phase*

Description

A LearnerTorch uses the batch_size parameter for both training and prediction, unless batch_size_predict is set, which then takes precedence during prediction. This helper resolves the batch size for one phase and is useful when overwriting the private .dataloader() method of a LearnerTorch.

Usage

```
get_batch_size(param_vals, phase)
```

Arguments

param_vals	(named list()) The parameter values, containing batch_size and/or batch_size_predict.
phase	(character(1)) Either "train" or "predict".

Value

(integer(1) or NULL)
The batch size for the given phase or NULL if none is set.

Examples

```
get_batch_size(list(batch_size = 16), "train")
get_batch_size(list(batch_size = 16, batch_size_predict = 32), "predict")
get_batch_size(list(batch_size_predict = 32), "train")
```

get_target_batchgetter *Target Batchgetter for a Task*

Description

Returns the function that converts the target column(s) of a `task` into the target tensor `y` of a batch, i.e. the tensor that the loss is applied to. The returned function takes an argument `data`, a `data.table` containing only the target column(s), and returns a `torch_tensor`. It is `NULL` for a task with no target at all, whose batches have no `y` element and whose loss is called as `loss(y_hat)`.

For the target encodings of the built-in task types, see section *Network Head and Target Encoding* of `LearnerTorch`.

When adding support for a custom task type, implement a method for the corresponding `Task` class.

Usage

```
get_target_batchgetter(task, ...)
```

Arguments

`task`	(`Task`) The task.
`...`	(any) Additional arguments. Not used yet.

Value

`function(data)`, `function(data, x)`, or `NULL` for a task with no target

Examples

```
batchgetter = get_target_batchgetter(tsk("iris"))
batchgetter(data.table::data.table(Species = factor(c("setosa", "virginica"))))
```

GraphLearnerTorch *Graph Learner for Torch Networks*

Description

The `GraphLearner` that `as_learner_torch()` returns. On top of a `GraphLearner` it has the `$network`, `$loss`, `$optimizer` and `$callbacks` fields and the `$dataset()` method of a `LearnerTorch`. If the graph contains a `PipeOpTorchModel`, the loss, optimizer and callbacks are those of its configuring operators, otherwise those of its base `LearnerTorch`. It is not in the `mlr_learners` dictionary, because it cannot be constructed without a graph.

Super classes

`mlr3::Learner` -> `mlr3pipelines::GraphLearner` -> `GraphLearnerTorch`

Active bindings

`network` (`nn_module`)
: The network of the trained learner, i.e. `$base_learner()$network`.

`loss` (`TorchLoss`)
: The torch loss, i.e. the one of the graph's `PipeOpTorchLoss` or of its base `LearnerTorch`. Read-only, the graph configures it.

`optimizer` (`TorchOptimizer`)
: The torch optimizer, i.e. the one of the graph's `PipeOpTorchOptimizer` or of its base `LearnerTorch`. Read-only, the graph configures it.

`callbacks` (`list()` of `TorchCallbacks`)
: The callbacks, i.e. those of the graph's `PipeOpTorchCallbacks` or of its base `LearnerTorch`. Read-only, the graph configures them.

Methods

Public methods:

- `GraphLearnerTorch$dataset()`
- `GraphLearnerTorch$clone()`

`GraphLearnerTorch$dataset()`: Create the dataset for a task, i.e. the tensors that are fed to the network.

Usage:

```
GraphLearnerTorch$dataset(
  task,
  stage = if (is.null(self$model)) "train" else "predict"
)
```

Arguments:

`task` `Task`
: The task.

`stage` (`character(1)`)
: Whether to create the dataset the way `$train()` does (`"train"`) or the way `$predict()` does (`"predict"`). Defaults to `"predict"` for a trained learner and to `"train"` otherwise, because the prediction phase reuses the state that the operators before the ingress fitted during training.

Returns: `dataset`

`GraphLearnerTorch$clone()`: The objects of this class are cloneable with this method.

Usage:

```
GraphLearnerTorch$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Learner: as_learner_torch(), mlr_learners.ft_transformer, mlr_learners.mlp, mlr_learners.module, mlr_learners.realmlp, mlr_learners.tab_resnet, mlr_learners.tabm, mlr_learners.torch_featureless, mlr_learners_torch, mlr_learners_torch_image, mlr_learners_torch_model

---

infer_shapes *Infer Shapes*

Description

Infer the shapes of the output of a function based on the shapes of the input. This works by running the function on the input and observing the results. For fully known input shapes this is always correct. For partially unknown shapes, the NAs are replaced with various concrete values and the output shape is computed from them. Note that this is a heuristic that might fail, so usually one wants to provide the shape (inference) explicitly.

Usage

```
infer_shapes(shapes_in, param_vals, output_names, fn, rowwise, id)
```

Arguments

shapes_in	(list()) A list of shapes of the input tensors.
param_vals	(list()) A list of named parameters for the function.
output_names	(character()) The names of the output tensors.
fn	(function()) The function to infer the shapes for.
rowwise	(logical(1)) Whether the function is rowwise.
id	(character(1)) The id of the PipeOp (for error messages).

Details

The inference is done as follows:

1. All NAs are replaced with three different values, which span a wide range: none of them is 1 (which broadcasts and is squeezed away), one of them is small (to detect operators that clamp to the input size, such as slicing or cropping) and the others are large (because operators such as a convolution with a large kernel need a minimum extent).
2. Three tensors are generated for the three shapes of step 1.

3. The function is called on these three tensors and the shapes are calculated. A call that fails is dropped, so that an operator is not rejected because of the smallest value; at least two of the three calls must succeed.
4. If:
   - the number of dimensions varies, an error is thrown.
   - the number of dimensions is the same, values are set to `NA` if the dimension is varying between the tensors and otherwise set to the unique value.

Value

(`list()`)
A list of shapes of the output tensors.

See Also

Other Shape Inference: `reshape_output_shape()`, `shape_to_str()`

`ingress_categ` *Ingress Token for Categorical Features*

Description

Represents an entry point representing a tensor containing all categorical (`factor()`, `ordered()`, `logical()`) features of a task.

Usage

```
ingress_categ(shape = NULL)
```

Arguments

shape	(`integer()` or `NULL`) Shape that `batchgetter` will produce. Batch-dimension should be included as `NA`.

Value

`TorchIngressToken`

ingress_ltnsr *Ingress Token for Lazy Tensor Feature*

Description

Represents an entry point representing a tensor containing a single lazy tensor feature.

Usage

```
ingress_ltnsr(feature_name = NULL, shape = NULL)
```

Arguments

feature_name	(character(1)) Which lazy tensor feature to select if there is more than one.
shape	(integer() or NULL) Shape that batchgetter will produce. Batch-dimension should be included as NA.

Value

TorchIngressToken

ingress_num *Ingress Token for Numeric Features*

Description

Represents an entry point representing a tensor containing all numeric (integer() and double()) features of a task.

Usage

```
ingress_num(shape = NULL)
```

Arguments

shape	(integer() or NULL) Shape that batchgetter will produce. Batch-dimension should be included as NA.

Value

TorchIngressToken

is_lazy_tensor *Check for lazy tensor*

Description

Checks whether an object is a lazy tensor.

Usage

```
is_lazy_tensor(x)
```

Arguments

x	(any) Object to check.

lazy_shape *Shape of Lazy Tensor*

Description

Shape of a lazy tensor. Might be NULL if the shapes is not known or varying between rows. Batch dimension is always NA.

Usage

```
lazy_shape(x)
```

Arguments

x	(lazy_tensor) Lazy tensor.

Value

(integer() or NULL)

Examples

```
lt = as_lazy_tensor(1:10)
lazy_shape(lt)
lt = as_lazy_tensor(matrix(1:10, nrow = 2))
lazy_shape(lt)
```

lazy_tensor *Create a lazy tensor*

Description

Creates a `lazy_tensor` vector. Because it is a vector, it can be stored in a `data.table`, which gives mlr3torch the ability to use arbitrary tensors in its task. It is 'lazy', because the tensors are not stored in-memory, but only loaded when calling `materialize()`. The vector itself only describes *how* to load the data. It is also possible to preprocess lazy_tensors, e.g. via `po("augment_<key>")`, and `po("trafo_<key>")`.

Usage

```
lazy_tensor(data_descriptor = NULL, ids = NULL)
```

Arguments

`data_descriptor`	(`DataDescriptor` or `NULL`) The data descriptor or `NULL` for a lazy tensor of length 0.
`ids`	(`integer()`) The elements of the `data_descriptor` to be included in the lazy tensor.

Examples

```
ds = dataset("example",
  initialize = function() self$iris = iris[, -5],
  .getitem = function(i) list(x = torch_tensor(as.numeric(self$iris[i, ]))),
  .length = function() nrow(self$iris)
)()
dd = as_data_descriptor(ds, list(x = c(NA, 4L)))
lt = as_lazy_tensor(dd)
```

materialize *Materialize Lazy Tensor Columns*

Description

This will materialize a `lazy_tensor()` or a `data.frame()` / `list()` containing – among other things – `lazy_tensor()` columns. I.e. the data described in the underlying `DataDescriptor`s is loaded for the indices in the `lazy_tensor()`, is preprocessed and then put unto the specified device. Because not all elements in a lazy tensor must have the same shape, a list of tensors is returned by default. If all elements have the same shape, these tensors can also be rbinded into a single tensor (parameter `rbind`).

Usage

```
materialize(x, device = "cpu", rbind = FALSE, ...)

## S3 method for class 'list'
materialize(x, device = "cpu", rbind = FALSE, cache = "auto", ...)
```

Arguments

x	(any) The object to materialize. Either a lazy_tensor or a list() / data.frame() containing lazy_tensor columns.
device	(character(1)) The torch device.
rbind	(logical(1)) Whether to rbind the lazy tensor columns (TRUE) or return them as a list of tensors (FALSE). In the second case, there is no batch dimension.
...	(any) Additional arguments.
cache	(character(1) or hashtab() or NULL) Optional cache for (intermediate) materialization results. Per default, caching will be enabled when the same dataset or data descriptor (with different output pointer) is used for more than one lazy tensor column.

Details

Materializing a lazy tensor consists of:

1. Loading the data from the internal dataset of the DataDescriptor.
2. Processing these batches in the preprocessing Graphs.
3. Returning the result of the PipeOp pointed to by the DataDescriptor (pointer).

With multiple lazy_tensor columns we can benefit from caching because: a) Output(s) from the dataset might be input to multiple graphs. b) Different lazy tensors might be outputs from the same graph.

For this reason it is possible to provide a cache, which is a hashtab(). The key for a) is list(dataset, indices), the key for b) is list(indices, dataset, graph, input_map). The dataset and the graph go into the key as the objects themselves, so keys are compared with identical() rather than being digested into a string, and two different keys can never share an entry.

Value

(list() of torch_tensors or a torch_tensor)

Examples

```
lt1 = as_lazy_tensor(torch_randn(10, 3))
materialize(lt1, rbind = TRUE)
materialize(lt1, rbind = FALSE)
lt2 = as_lazy_tensor(torch_randn(10, 4))
d = data.table::data.table(lt1 = lt1, lt2 = lt2)
materialize(d, rbind = TRUE)
materialize(d, rbind = FALSE)
```

---

mlr_backends_lazy *Lazy Data Backend*

Description

This lazy data backend wraps a constructor that lazily creates another backend, e.g. by downloading (and caching) some data from the internet. This backend should be used, when some metadata of the backend is known in advance and should be accessible before downloading the actual data. When the backend is first constructed, it is verified that the provided metadata was correct, otherwise an informative error message is thrown. After the construction of the lazily constructed backend, calls like `$data()`, `$missings()`, `$distinct()`, or `$hash()` are redirected to it.

Information that is available before the backend is constructed is:

- `nrow` - The number of rows (set as the length of the `rownames`).
- `ncol` - The number of columns (provided via the `id` column of `col_info`).
- `colnames` - The column names.
- `rownames` - The row names.
- `col_info` - The column information, which can be obtained via `mlr3::col_info()`.

Beware that accessing the backend's hash also contructs the backend.

Note that while in most cases the data contains `lazy_tensor` columns, this is not necessary and the naming of this class has nothing to do with the `lazy_tensor` data type.

Important

When the constructor generates `factor()` variables it is important that the ordering of the levels in data corresponds to the ordering of the levels in the `col_info` argument.

Super class

`mlr3::DataBackend` -> `DataBackendLazy`

Active bindings

`backend (DataBackend)`
: The wrapped backend that is lazily constructed when first accessed.

`nrow (integer(1))`
: Number of rows (observations).

`ncol (integer(1))`
: Number of columns (variables), including the primary key column.

`rownames (integer())`
: Returns vector of all distinct row identifiers, i.e. the contents of the primary key column.

`colnames (character())`
: Returns vector of all column names, including the primary key column.

`is_constructed (logical(1))`
: Whether the backend has already been constructed.

Methods

Public methods:

- `DataBackendLazy$new()`
- `DataBackendLazy$data()`
- `DataBackendLazy$head()`
- `DataBackendLazy$distinct()`
- `DataBackendLazy$missings()`
- `DataBackendLazy$print()`

`DataBackendLazy$new()`**:** Creates a new instance of this R6 class.

Usage:

```
DataBackendLazy$new(constructor, rownames, col_info, primary_key)
```

Arguments:

`constructor (function)`
: A function with argument `backend` (the lazy backend), whose return value must be the actual backend. This function is called the first time the field `$backend` is accessed.

`rownames (integer())`
: The row names. Must be a permutation of the rownames of the lazily constructed backend.

`col_info (data.table::data.table())`
: A data.table with columns `id`, `type` and `levels` containing the column id, type and levels. Note that the levels must be provided in the correct order.

`primary_key (character(1))`
: Name of the primary key column.

`DataBackendLazy$data()`**:** Returns a slice of the data in the specified format. The rows must be addressed as vector of primary key values, columns must be referred to via column names. Queries for rows with no matching row id and queries for columns with no matching column name are silently ignored. Rows are guaranteed to be returned in the same order as `rows`, columns may be returned in an arbitrary order. Duplicated row ids result in duplicated rows, duplicated column names lead to an exception.
Accessing the data triggers the construction of the backend.

Usage:

```
DataBackendLazy$data(rows, cols)
```

Arguments:

`rows (integer())`
: Row indices.

`cols (character())`
: Column names.

`DataBackendLazy$head()`: Retrieve the first `n` rows. This triggers the construction of the backend.

Usage:

```
DataBackendLazy$head(n = 6L)
```

Arguments:

`n (integer(1))`
: Number of rows.

Returns: `data.table::data.table()` of the first `n` rows.

`DataBackendLazy$distinct()`: Returns a named list of vectors of distinct values for each column specified. If `na_rm` is `TRUE`, missing values are removed from the returned vectors of distinct values. Non-existing rows and columns are silently ignored.
This triggers the construction of the backend.

Usage:

```
DataBackendLazy$distinct(rows, cols, na_rm = TRUE)
```

Arguments:

`rows (integer())`
: Row indices.

`cols (character())`
: Column names.

`na_rm (logical(1))`
: Whether to remove NAs or not.

Returns: Named `list()` of distinct values.

`DataBackendLazy$missings()`: Returns the number of missing values per column in the specified slice of data. Non-existing rows and columns are silently ignored.
This triggers the construction of the backend.

Usage:

```
DataBackendLazy$missings(rows, cols)
```

Arguments:

`rows (integer())`
: Row indices.

`cols (character())`
: Column names.

Returns: Total of missing values per column (named `numeric()`).

`DataBackendLazy$print()`: Printer.

Usage:

```
DataBackendLazy$print()
```

Examples

```
# We first define a backend constructor
constructor = function(backend) {
  cat("Data is constructed!\n")
  DataBackendDataTable$new(
    data.table(x = rnorm(10), y = rnorm(10), row_id = 1:10),
    primary_key = "row_id"
  )
}

# to wrap this backend constructor in a lazy backend, we need to provide the correct metadata for it
column_info = data.table(
  id = c("x", "y", "row_id"),
  type = c("numeric", "numeric", "integer"),
  levels = list(NULL, NULL, NULL)
)
backend_lazy = DataBackendLazy$new(
  constructor = constructor,
  rownames = 1:10,
  col_info = column_info,
  primary_key = "row_id"
)

# Note that the constructor is not called for the calls below
# as they can be read from the metadata
backend_lazy$nrow
backend_lazy$rownames
backend_lazy$ncol
backend_lazy$colnames
col_info(backend_lazy)

# Only now the backend is constructed
backend_lazy$data(1, "x")
# Is the same as:
backend_lazy$backend$data(1, "x")
```

`mlr_callback_set` *Base Class for Callbacks*

Description

Base class from which callbacks should inherit (see section *Inheriting*). A callback set is a collection of functions that are executed at different stages of the training loop. They can be used to gain more control over the training process of a neural network without having to write everything from scratch.

When used in a torch learner, the `CallbackSet` is wrapped in a `TorchCallback`. The latter's parameter set represents the arguments of the `CallbackSet`'s `$initialize()` method.

Inheriting

For each available stage (see section *Stages*) a public method `$on_<stage>()` can be defined. The evaluation context (a `ContextTorch`) can be accessed via `self$ctx`, which contains the current state of the training loop. This context is assigned at the beginning of the training loop and removed afterwards. Different stages of a callback can communicate with each other by assigning values to `self`.

State: To be able to store information in the `$model` slot of a `LearnerTorch`, callbacks support a state API. You can overload the `$state_dict()` public method to define what will be stored in `learner$model$callbacks$<id>` after training finishes. This then also requires to implement a `$load_state_dict(state_dict)` method that defines how to load a previously saved callback state into a different callback. Note that the `$state_dict()` should not include the parameter values that were used to initialize the callback.

In order to be able to resume a callback, it needs to work with `saveRDS`, so it cannot contain external pointers such as `torch_tensor`s. Convert them to R data, e.g. via `as.array()`.

For creating custom callbacks, the function `torch_callback()` is recommended, which creates a `CallbackSet` and then wraps it in a `TorchCallback`. To create a `CallbackSet` the convenience function `callback_set()` can be used. These functions perform checks such as that the stages are not accidentally misspelled.

Stages

- `begin` :: Run before the training loop begins.
- `epoch_begin` :: Run at the beginning of each epoch.
- `batch_begin` :: Run before the forward call.
- `after_backward` :: Run after the backward call.
- `batch_end` :: Run after the optimizer step.
- `before_valid` :: Run before the validation loop begins.
- `batch_valid_begin` :: Run before the forward call in the validation loop.
- `batch_valid_end` :: Run after the forward call in the validation loop.
- `valid_end` :: Run at the end of validation.
- `epoch_end` :: Run at the end of each epoch.
- `end` :: Run after last epoch.
- `exit` :: Run last, using `on.exit()`.

Ordering

Within a stage, callbacks are called in the order in which they were passed to the learner. A callback can override this via its `$weight` field: callbacks with a higher weight are called after those with a lower one, and callbacks with the same weight keep the order in which they were passed. The default weight is `0`. This matters for callbacks that observe what the others did, which is why `CallbackSetCheckpoint` has weight `Inf`: it always runs last and therefore saves the network and optimizer as the other callbacks left them at the end of the stage.

Terminate Training

If training is to be stopped, it is possible to set the field `$terminate` of `ContextTorch`. At the end of every epoch this field is checked and if it is `TRUE`, training stops. This can for example be used to implement custom early stopping.

Public fields

`ctx` (`ContextTorch` or `NULL`)
The evaluation context for the callback. This field should always be `NULL` except during the `$train()` call of the torch learner.

`weight` (`numeric(1)`)
Controls when this callback is called within a stage, see section *Ordering*.

Active bindings

`stages` (`character()`)
The active stages of this callback set.

Methods

Public methods:

- `CallbackSet$print()`
- `CallbackSet$state_dict()`
- `CallbackSet$load_state_dict()`
- `CallbackSet$clone()`

`CallbackSet$print()`: Prints the object.

Usage:

```
CallbackSet$print(...)
```

Arguments:

`...` (any)
Currently unused.

`CallbackSet$state_dict()`: Returns information that is kept in the `LearnerTorch`'s state after training. This information should be loadable into the callback using `$load_state_dict()` to be able to continue training. This returns `NULL` by default.

Usage:

```
CallbackSet$state_dict()
```

`CallbackSet$load_state_dict()`: Loads the state dict into the callback to continue training. A callback that keeps no state inherits this, which has nothing to load and therefore only warns when it is handed a state anyway – a callback that implements `$state_dict()` but not this method, so that a resuming run cannot give its state back.

Usage:

```
CallbackSet$load_state_dict(state_dict)
```

Arguments:

`state_dict` (any)
The state dict as retrieved via `$state_dict()`.

`CallbackSet$clone()`: The objects of this class are cloneable with this method.

Usage:

```
CallbackSet$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Callback: `TorchCallback`, `as_torch_callback()`, `as_torch_callbacks()`, `callback_set()`, `mlr3torch_callbacks`, `mlr_callback_set.checkpoint`, `mlr_callback_set.progress`, `mlr_callback_set.tb`, `mlr_callback_set.unfreeze`, `mlr_context_torch`, `t_clbk()`, `torch_callback()`

`mlr_callback_set.checkpoint` *Checkpoint Callback*

Description

Saves the optimizer, weights, and callback states every `freq` epochs as well as the final state. This can be used to later continue a training run via the `resume` parameter of `LearnerTorch`.

Checkpoints are written at the end of an epoch. For one written after epoch `<n>`, three files are created in `path`:

- `network<n>.pt` :: The `$state_dict()` of the network.
- `optimizer<n>.pt` :: The `$state_dict()` of the optimizer.
- `state<n>.rds` :: The epoch, the version of `mlr3torch` that wrote the checkpoint, the `$state_dict()`s of the training run's other callbacks, so that a later run can continue, as well as some other information. Additionally, there is `run.rds` which contains some additioanl global meta information.

Details

Saving the learner itself in the callback with a trained model is impossible, as the model slot is set *after* the last callback step is executed.

Resuming

This callback is special because it enables resuming a training run. Its own state is the folder it writes to, which `learner$model$callbacks$<id>$path` reports – the only way to learn where a `path` function sent a run, e.g. one fit of a `resample()`. That state is not part of a checkpoint and is not restored: a resuming run writes where its own `path` says.

Ordering

This callback has weight `Inf` and therefore runs last, so it captures all the changes other callbacks made.

Super class

`CallbackSet` -> `CallbackSetCheckpoint`

Public fields

`weight (numeric(1))`
Always `Inf`, see section *Ordering*.

Methods

Public methods:

- `CallbackSetCheckpoint$new()`
- `CallbackSetCheckpoint$state_dict()`
- `CallbackSetCheckpoint$on_begin()`
- `CallbackSetCheckpoint$on_epoch_end()`
- `CallbackSetCheckpoint$on_end()`
- `CallbackSetCheckpoint$clone()`

`CallbackSetCheckpoint$new()`**:** Creates a new instance of this R6 class.

Usage:

```
CallbackSetCheckpoint$new(path, freq)
```

Arguments:

`path (character(1) | function())`
The path to a folder where the models are saved, or a function of no arguments returning it. The latter is especially useful to create unique directories during `resample()` or `benchmark()` per fit. The folder must be new, empty, or already contain checkpoints. A half-written checkpoint – what a run killed mid-write leaves behind – may be written over, since a resuming run continues from the newest complete one.

`freq (integer(1))`
How often the model is saved, in epochs.

`CallbackSetCheckpoint$state_dict()`**:** Returns the folder this callback writes to so it can be accessed from the learner when the `path` was a function.

Usage:

```
CallbackSetCheckpoint$state_dict()
```

`CallbackSetCheckpoint$on_begin()`**:** Checks whether the checkpoint path is valid.

Usage:

```
CallbackSetCheckpoint$on_begin()
```

`CallbackSetCheckpoint$on_epoch_end()`**:** Saves the network and optimizer state dict. Does nothing if `freq` is not met.

Usage:

```
CallbackSetCheckpoint$on_epoch_end()
```

`CallbackSetCheckpoint$on_end()`: Saves the final network and optimizer, unless the last epoch was already saved.

Usage:

```
CallbackSetCheckpoint$on_end()
```

`CallbackSetCheckpoint$clone()`: The objects of this class are cloneable with this method.

Usage:

```
CallbackSetCheckpoint$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Callback: TorchCallback, as_torch_callback(), as_torch_callbacks(), callback_set(), mlr3torch_callbacks, mlr_callback_set, mlr_callback_set.progress, mlr_callback_set.tb, mlr_callback_set.unfreeze, mlr_context_torch, t_clbk(), torch_callback()

Examples

```
cb = t_clbk("checkpoint", freq = 1)
task = tsk("iris")

pth = tempfile()
learner = lrn("classif.mlp", epochs = 3, batch_size = 1, callbacks = cb)
learner$param_set$set_values(cb.checkpoint.path = pth)

learner$train(task)

list.files(pth)

# continue training for 3 more epochs, starting from the last checkpoint
learner_resumed = lrn("classif.mlp", epochs = 6, batch_size = 1, resume = pth)
learner_resumed$train(task)
learner_resumed$model$epochs
```

---

mlr_callback_set.history

History Callback

Description

Saves the training and validation history during training. The history is saved as a data.table where the validation measures are prefixed with "valid." and the training measures are prefixed with "train.".

Resuming

The epochs of a resumed run are appended to the history of the run it continues. A measure that only one of the two runs recorded is NA for the epochs of the other.

Super class

CallbackSet -> CallbackSetHistory

Methods

Public methods:

- CallbackSetHistory$on_begin()
- CallbackSetHistory$state_dict()
- CallbackSetHistory$load_state_dict()
- CallbackSetHistory$on_before_valid()
- CallbackSetHistory$on_epoch_end()
- CallbackSetHistory$clone()

CallbackSetHistory$on_begin(): Initializes lists where the train and validation metrics are stored.

Usage:

```
CallbackSetHistory$on_begin()
```

CallbackSetHistory$state_dict(): Converts the lists to data.tables.

Usage:

```
CallbackSetHistory$state_dict()
```

CallbackSetHistory$load_state_dict(): Remembers the history contained in the state dict, so that the epochs of the current run are appended to it by $state_dict().

Usage:

```
CallbackSetHistory$load_state_dict(state_dict)
```

Arguments:

state_dict (callback_state_history)
The state dict as retrieved via $state_dict().

CallbackSetHistory$on_before_valid(): Add the latest training scores to the history.

Usage:

```
CallbackSetHistory$on_before_valid()
```

CallbackSetHistory$on_epoch_end(): Add the latest validation scores to the history.

Usage:

```
CallbackSetHistory$on_epoch_end()
```

CallbackSetHistory$clone(): The objects of this class are cloneable with this method.

Usage:

```
CallbackSetHistory$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
cb = t_clbk("history")
task = tsk("iris")

learner = lrn("classif.mlp", epochs = 3, batch_size = 1,
  callbacks = t_clbk("history"), validate = 0.3)
learner$param_set$set_values(
  measures_train = msrs(c("classif.acc", "classif.ce")),
  measures_valid = msr("classif.ce")
)
learner$train(task)

print(learner$model$callbacks$history)
```

`mlr_callback_set.lr_scheduler`

Learning Rate Scheduling Callback

Description

Changes the learning rate based on the schedule specified by a `torch::lr_scheduler`.

As of this writing, the following are available:

- `torch::lr_cosine_annealing()`
- `torch::lr_lambda()`
- `torch::lr_multiplicative()`
- `torch::lr_one_cycle()` (where the default values for `epochs` and `steps_per_epoch` are the number of training epochs and the number of batches per epoch)
- `torch::lr_reduce_on_plateau()`
- `torch::lr_step()`
- Custom schedulers defined with `torch::lr_scheduler()`.

Resuming

The state of the wrapped `torch` scheduler is stored and restored, so a resumed run continues the schedule instead of starting it over. Creating a scheduler resets the optimizer's learning rate to the one the schedule started at, so the rate the restored schedule had reached is put back afterwards.

That state contains the scheduler's configuration as well as its progress, and restoring it overwrites what the resuming run was configured with. Resuming with different scheduler arguments, or a different `opt.lr`, which the schedule's base rates are derived from, therefore silently continues the schedule of the checkpointed run.

Super class

`CallbackSet` -> `CallbackSetLRScheduler`

Public fields

`scheduler_fn` (`lr_scheduler_generator`)
The `torch` function that creates a learning rate scheduler

`scheduler` (`LRScheduler`)
The learning rate scheduler wrapped by this callback

Methods

Public methods:

- `CallbackSetLRScheduler$new()`
- `CallbackSetLRScheduler$on_begin()`
- `CallbackSetLRScheduler$state_dict()`
- `CallbackSetLRScheduler$load_state_dict()`
- `CallbackSetLRScheduler$clone()`

`CallbackSetLRScheduler$new()`: Creates a new instance of this R6 class.

Usage:

```
CallbackSetLRScheduler$new(.scheduler, step_on_epoch, ...)
```

Arguments:

`.scheduler` (`lr_scheduler_generator`)
The `torch` scheduler generator (e.g. `torch::lr_step`).

`step_on_epoch` (`logical(1)`)
Whether the scheduler steps after every epoch (otherwise every batch).

`...` (any)
The scheduler-specific initialization arguments.

`CallbackSetLRScheduler$on_begin()`: Creates the scheduler using the optimizer from the context

Usage:

```
CallbackSetLRScheduler$on_begin()
```

`CallbackSetLRScheduler$state_dict()`: Returns the state of the wrapped `torch` scheduler, so that a later run can continue the schedule instead of starting it over. Returns `NULL` if the scheduler was not created yet, i.e. before the training loop began.

Usage:

```
CallbackSetLRScheduler$state_dict()
```

`CallbackSetLRScheduler$load_state_dict()`: Loads the state of the wrapped `torch` scheduler.

Usage:

```
CallbackSetLRScheduler$load_state_dict(state_dict)
```

Arguments:

`state_dict` (named `list()`)
The state dict as retrieved via `$state_dict()`.

`CallbackSetLRScheduler$clone()`: The objects of this class are cloneable with this method.

Usage:

```
CallbackSetLRScheduler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`mlr_callback_set.lr_scheduler_one_cycle`
OneCycle Learning Rate Scheduling Callback

Description

Changes the learning rate based on the 1cycle learning rate policy.

Wraps `torch::lr_one_cycle()`, where the default values for `epochs` and `steps_per_epoch` are the number of training epochs and the number of batches per epoch.

Resuming

As for `CallbackSetLRScheduler`, with one additional restriction: the 1cycle policy is defined over the total number of steps of the run, so a resumed run must be configured for the same number of steps as the one that wrote the checkpoint, i.e. the same `epochs` and the same number of batches per epoch. A run that is not errors before its first epoch rather than somewhere in the middle.

Super classes

`CallbackSet` -> `CallbackSetLRScheduler` -> `CallbackSetLRSchedulerOneCycle`

Methods

Public methods:

- `CallbackSetLRSchedulerOneCycle$new()`
- `CallbackSetLRSchedulerOneCycle$on_begin()`
- `CallbackSetLRSchedulerOneCycle$clone()`

`CallbackSetLRSchedulerOneCycle$new()`: Creates a new instance of this R6 class.

Usage:

```
CallbackSetLRSchedulerOneCycle$new(...)
```

Arguments:

`...` (any)
The scheduler-specific initialization arguments.

`CallbackSetLRSchedulerOneCycle$on_begin()`: Creates the scheduler using the optimizer from the context

Usage:

```
CallbackSetLRSchedulerOneCycle$on_begin()
```

`CallbackSetLRSchedulerOneCycle$clone()`: The objects of this class are cloneable with this method.

Usage:

```
CallbackSetLRSchedulerOneCycle$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`mlr_callback_set.lr_scheduler_reduce_on_plateau`
Reduce On Plateau Learning Rate Scheduler

Description

Reduces the learning rate when the first validation measure, i.e. the first entry of the learner's `measures_valid`, stops improving for `patience` evaluation steps. Wraps `torch::lr_reduce_on_plateau()`.

Note that `patience` counts *evaluation steps*, not epochs: when the learner's `eval_freq` is greater than 1, the scheduler only steps in the epochs in which the validation measures are evaluated, so `patience` evaluation steps correspond to `patience * eval_freq` epochs.

Unless `mode` is set, it is derived from the measure: `"min"` if the measure is minimized and `"max"` if it is maximized. Training fails when `mode` contradicts the measure. When the learner has no validation measure, e.g. in the final fit of an `AutoTuner` that tunes the epochs internally, a warning is given and the learning rate is not reduced.

Resuming

As for `CallbackSetLRScheduler`. For this schedule the restored state includes the best score seen so far and how long it has been stagnating, so `patience` keeps counting across runs instead of starting over.

Super classes

`CallbackSet` -> `CallbackSetLRScheduler` -> `CallbackSetLRSchedulerReduceOnPlateau`

Methods

Public methods:

- `CallbackSetLRSchedulerReduceOnPlateau$new()`
- `CallbackSetLRSchedulerReduceOnPlateau$on_begin()`
- `CallbackSetLRSchedulerReduceOnPlateau$clone()`

`CallbackSetLRSchedulerReduceOnPlateau$new()`: Creates a new instance of this R6 class.

Usage:

```
CallbackSetLRSchedulerReduceOnPlateau$new(...)
```

Arguments:

`...` (any)
The scheduler-specific initialization arguments.

`CallbackSetLRSchedulerReduceOnPlateau$on_begin()`: Creates the scheduler using the optimizer from the context, with the `mode` that fits the first validation measure.

Usage:

```
CallbackSetLRSchedulerReduceOnPlateau$on_begin()
```

`CallbackSetLRSchedulerReduceOnPlateau$clone()`: The objects of this class are cloneable with this method.

Usage:

```
CallbackSetLRSchedulerReduceOnPlateau$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`mlr_callback_set.progress` *Progress Callback*

Description

Prints a progress bar and the metrics for training and validation.

Resuming

This callback can be resumed without any problems.

Super class

`CallbackSet` -> `CallbackSetProgress`

Methods

Public methods:

- `CallbackSetProgress$new()`
- `CallbackSetProgress$on_begin()`
- `CallbackSetProgress$on_epoch_begin()`
- `CallbackSetProgress$on_batch_end()`
- `CallbackSetProgress$on_before_valid()`
- `CallbackSetProgress$on_batch_valid_end()`

- CallbackSetProgress$on_epoch_end()
- CallbackSetProgress$on_end()
- CallbackSetProgress$state_dict()
- CallbackSetProgress$load_state_dict()
- CallbackSetProgress$clone()

`CallbackSetProgress$new()`: Creates a new instance of this R6 class.

Usage:

```
CallbackSetProgress$new(digits = 2)
```

Arguments:

`digits integer(1)`
The number of digits to print for the measures.

`CallbackSetProgress$on_begin()`: Starts this run's timer.

Usage:

```
CallbackSetProgress$on_begin()
```

`CallbackSetProgress$on_epoch_begin()`: Initializes the progress bar for training.

Usage:

```
CallbackSetProgress$on_epoch_begin()
```

`CallbackSetProgress$on_batch_end()`: Increments the training progress bar.

Usage:

```
CallbackSetProgress$on_batch_end()
```

`CallbackSetProgress$on_before_valid()`: Creates the progress bar for validation.

Usage:

```
CallbackSetProgress$on_before_valid()
```

`CallbackSetProgress$on_batch_valid_end()`: Increments the validation progress bar.

Usage:

```
CallbackSetProgress$on_batch_valid_end()
```

`CallbackSetProgress$on_epoch_end()`: Prints a summary of the training and validation process.

Usage:

```
CallbackSetProgress$on_epoch_end()
```

`CallbackSetProgress$on_end()`: Prints the time at the end of training, and how long training took in total. A resumed run also reports how much of that total it contributed itself.

Usage:

```
CallbackSetProgress$on_end()
```

`CallbackSetProgress$state_dict()`: Returns the seconds trained so far, so that a resumed run reports the time of all runs together rather than only its own.

Usage:

```
CallbackSetProgress$state_dict()
```

`CallbackSetProgress$load_state_dict()`: Loads the time that the previous runs took.

Usage:

```
CallbackSetProgress$load_state_dict(state_dict)
```

Arguments:

`state_dict` (named `list()`)
The state dict as retrieved via `$state_dict()`.

`CallbackSetProgress$clone()`: The objects of this class are cloneable with this method.

Usage:

```
CallbackSetProgress$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Callback: `TorchCallback`, `as_torch_callback()`, `as_torch_callbacks()`, `callback_set()`, `mlr3torch_callbacks`, `mlr_callback_set`, `mlr_callback_set.checkpoint`, `mlr_callback_set.tb`, `mlr_callback_set.unfreeze`, `mlr_context_torch`, `t_clbk()`, `torch_callback()`

Examples

```
task = tsk("iris")

learner = lrn("classif.mlp", epochs = 5, batch_size = 1,
  callbacks = t_clbk("progress"), validate = 0.3)
learner$param_set$set_values(
  measures_train = msrs(c("classif.acc", "classif.ce")),
  measures_valid = msr("classif.ce")
)

learner$train(task)
```

---

mlr_callback_set.tb *TensorBoard Logging Callback*

Description

Logs training loss, training measures, and validation measures as events. To view them, use TensorBoard with `tensorflow::tensorboard()` (requires `tensorflow`) or the CLI.

Details

Logs events at most every epoch.

Resuming

This callback keeps no state of its own so it can trivially be resumed.

Super class

`CallbackSet` -> `CallbackSetTB`

Methods

Public methods:

- `CallbackSetTB$new()`
- `CallbackSetTB$on_epoch_end()`
- `CallbackSetTB$clone()`

`CallbackSetTB$new()`: Creates a new instance of this R6 class.

Usage:

```
CallbackSetTB$new(path, log_train_loss)
```

Arguments:

`path` (`character(1)`)
The path to a folder where the events are logged. Point TensorBoard to this folder to view them. The folder must be new, empty, or one that this callback already logged into, so that a resumed run can continue the log of the run it continues.

`log_train_loss` (`logical(1)`)
Whether we log the training loss.

`CallbackSetTB$on_epoch_end()`: Logs the training loss, training measures, and validation measures as TensorBoard events.

Usage:

```
CallbackSetTB$on_epoch_end()
```

`CallbackSetTB$clone()`: The objects of this class are cloneable with this method.

Usage:

```
CallbackSetTB$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Callback: `TorchCallback`, `as_torch_callback()`, `as_torch_callbacks()`, `callback_set()`, `mlr3torch_callbacks`, `mlr_callback_set`, `mlr_callback_set.checkpoint`, `mlr_callback_set.progress`, `mlr_callback_set.unfreeze`, `mlr_context_torch`, `t_clbk()`, `torch_callback()`

`mlr_callback_set.unfreeze` *Unfreezing Weights Callback*

Description

Unfreeze some weights (parameters of the network) after some number of steps or epochs.

Resuming

Which weights are trainable is stored and restored, so a resumed run does not freeze again what the run it continues had already unfrozen: `starting_weights` is applied first, and the restored weights are unfrozen on top of it.

Super class

`CallbackSet` -> `CallbackSetUnfreeze`

Methods

Public methods:

- `CallbackSetUnfreeze$new()`
- `CallbackSetUnfreeze$on_begin()`
- `CallbackSetUnfreeze$state_dict()`
- `CallbackSetUnfreeze$load_state_dict()`
- `CallbackSetUnfreeze$on_epoch_begin()`
- `CallbackSetUnfreeze$on_batch_begin()`
- `CallbackSetUnfreeze$clone()`

`CallbackSetUnfreeze$new()`**:** Creates a new instance of this R6 class.

Usage:

```
CallbackSetUnfreeze$new(starting_weights, unfreeze)
```

Arguments:

`starting_weights` (`Select`)
: A `Select` denoting the weights that are trainable from the start.

`unfreeze` (`data.table`)
: A `data.table` with a column `weights` (a list column of `Selects`) and a column `epoch` or `batch`. The selector indicates which parameters to unfreeze, while the `epoch` or `batch` column indicates when to do so. Its values must be positive integers: epochs and batches are counted from 1 (batches across epochs), and the weights are unfrozen at the start of the given epoch or batch, so they are already trained in it.

`CallbackSetUnfreeze$on_begin()`**:** Sets the starting weights

Usage:

```
CallbackSetUnfreeze$on_begin()
```

`CallbackSetUnfreeze$state_dict()`: Returns the names of the weights that are currently trainable, so that a later run does not freeze weights again that were already unfrozen.

Usage:

```
CallbackSetUnfreeze$state_dict()
```

`CallbackSetUnfreeze$load_state_dict()`: Marks the weights of the state dict as trainable.

Usage:

```
CallbackSetUnfreeze$load_state_dict(state_dict)
```

Arguments:

`state_dict` (named `list()`)
The state dict as retrieved via `$state_dict()`.

`CallbackSetUnfreeze$on_epoch_begin()`: Unfreezes weights if the training is at the correct epoch

Usage:

```
CallbackSetUnfreeze$on_epoch_begin()
```

`CallbackSetUnfreeze$on_batch_begin()`: Unfreezes weights if the training is at the correct batch

Usage:

```
CallbackSetUnfreeze$on_batch_begin()
```

`CallbackSetUnfreeze$clone()`: The objects of this class are cloneable with this method.

Usage:

```
CallbackSetUnfreeze$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Callback: `TorchCallback`, `as_torch_callback()`, `as_torch_callbacks()`, `callback_set()`, `mlr3torch_callbacks`, `mlr_callback_set`, `mlr_callback_set.checkpoint`, `mlr_callback_set.progress`, `mlr_callback_set.tb`, `mlr_context_torch`, `t_clbk()`, `torch_callback()`

Examples

```
task = tsk("iris")
cb = t_clbk("unfreeze")
mlp = lrn("classif.mlp", callbacks = cb,
 cb.unfreeze.starting_weights = select_invert(
   select_name(c("0.weight", "3.weight", "6.weight", "6.bias"))
 ),
 cb.unfreeze.unfreeze = data.table(
   epoch = c(2, 5),
   weights = list(select_name("0.weight"), select_name(c("3.weight", "6.weight")))
```

```
 ),
 epochs = 6, batch_size = 150, neurons = c(1, 1, 1)
)

mlp$train(task)
```

`mlr_context_torch` *Context for Torch Learner*

Description

Context for training a torch learner. This is the - mostly read-only - information callbacks have access to through the argument `ctx`. For more information on callbacks, see `CallbackSet`.

Public fields

`learner` (`Learner`)
: The torch learner.

`task_train` (`Task`)
: The training task.

`task_valid` (`Task` or `NULL`)
: The validation task.

`loader_train` (`torch::dataloader`)
: The data loader for training.

`loader_valid` (`torch::dataloader`)
: The data loader for validation.

`measures_train` (`list()` of `Measure`s)
: Measures used for training.

`measures_valid` (`list()` of `Measure`s)
: Measures used for validation.

`network` (`torch::nn_module`)
: The torch network.

`optimizer` (`torch::optimizer`)
: The optimizer.

`loss_fn` (`torch::nn_module`)
: The loss function.

`total_epochs` (`integer(1)`)
: The total number of epochs the learner is trained for.

`last_scores_train` (named `list()` or `NULL`)
: The scores from the last evaluated epoch, calculated over all rows that were trained on in that epoch. Names are the ids of the training measures. If `LearnerTorch` sets `eval_freq` different from `1`, this is `NULL` in all epochs that don't evaluate the model.

`last_scores_valid` (`list()`)
: The scores from the last evaluated epoch, calculated over the complete validation task. Names are the ids of the validation measures. If `LearnerTorch` sets `eval_freq` different from `1`, this is `NULL` in all epochs that don't evaluate the model.

`y_hat` (`torch_tensor`)
: The network's output for the current batch, or its first element if the network returns more than one tensor. Provided for the most common case where a network returns a single tensor. For the full output, see `y_hats`.

`y_hats` (`torch_tensor` or `list()`)
: The complete output of the network for the current batch, i.e. what the loss is applied to. This is a `list()` if the network returns more than one tensor and identical to `y_hat` otherwise.

`epoch` (`integer(1)`)
: The current epoch.

`step` (`integer(1)`)
: The current iteration, i.e. the index of the batch within the current epoch. This is reset to `0` at the beginning of every epoch, so a globally unique step is `(epoch - 1) * length(loader_train) + step`.

`prediction_encoder` (`function()`)
: The learner's prediction encoder.

`batch` (named `list()` of `torch_tensors`)
: The current batch.

`terminate` (`logical(1)`)
: If this field is set to `TRUE` at the end of an epoch, training stops.

`callbacks` (named `list()` of `CallbackSet`s)
: The callbacks that are active during training, named by their ids. This allows a callback to access the state of the other callbacks, which is for example what `CallbackSetCheckpoint` does to save them.

`device` (`torch::torch_device`)
: The device.

Active bindings

`last_loss` (`numeric(1)`)
: The loss from the last training batch. The training loop stores the (detached) loss tensor and it is only converted to a number when this field is accessed, which avoids synchronizing the device in every step.

Methods

Public methods:

- `ContextTorch$new()`
- `ContextTorch$clone()`

`ContextTorch$new()`**:** Creates a new instance of this R6 class.

Usage:

```
ContextTorch$new(
  learner,
  task_train,
  task_valid = NULL,
  loader_train,
  loader_valid = NULL,
  measures_train = NULL,
  measures_valid = NULL,
  network,
  optimizer,
  loss_fn,
  total_epochs,
  prediction_encoder,
  eval_freq = 1L,
  device
)
```

Arguments:

`learner` (Learner)
: The torch learner.

`task_train` (Task)
: The training task.

`task_valid` (Task or NULL)
: The validation task.

`loader_train` (torch::dataloader)
: The data loader for training.

`loader_valid` (torch::dataloader or NULL)
: The data loader for validation.

`measures_train` (list() of Measures or NULL)
: Measures used for training. Default is NULL.

`measures_valid` (list() of Measures or NULL)
: Measures used for validation.

`network` (torch::nn_module)
: The torch network.

`optimizer` (torch::optimizer)
: The optimizer.

`loss_fn` (torch::nn_module)
: The loss function.

`total_epochs` (integer(1))
: The total number of epochs the learner is trained for.

`prediction_encoder` (function())
: The learner's prediction encoder. See section *Inheriting* of LearnerTorch.

`eval_freq` (integer(1))
: The evaluation frequency.

`device` (character(1))
: The device.

`ContextTorch$clone()`: The objects of this class are cloneable with this method.

Usage:

```
ContextTorch$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Callback: `TorchCallback`, `as_torch_callback()`, `as_torch_callbacks()`, `callback_set()`, `mlr3torch_callbacks`, `mlr_callback_set`, `mlr_callback_set.checkpoint`, `mlr_callback_set.progress`, `mlr_callback_set.tb`, `mlr_callback_set.unfreeze`, `t_clbk()`, `torch_callback()`

`mlr_learners.ft_transformer`

FT-Transformer

Description

Feature-Tokenizer Transformer for tabular data that can either work on `lazy_tensor` inputs or on standard tabular features.

Some differences from the paper implementation: no attention compression, no option to have prenormalization in the first layer.

If training is unstable, consider a combination of standardizing features (e.g. using `po("scale")`), using an adaptive optimizer (e.g. Adam), reducing the learning rate, and using a learning rate scheduler (see `CallbackSetLRScheduler` for options).

Dictionary

This Learner can be instantiated using the sugar function `lrn()`:

```
lrn("classif.ft_transformer", ...)
lrn("regr.ft_transformer", ...)
```

Properties

- Supported task types: 'classif', 'regr'
- Predict Types:
  - classif: 'response', 'prob'
  - regr: 'response'
- Feature Types: "logical", "integer", "numeric", "factor", "ordered", "lazy_tensor"
- Required Packages: **mlr3**, **mlr3torch**, **torch**

Parameters

Parameters from LearnerTorch and PipeOpTorchFTTransformerBlock, as well as:

- n_blocks :: integer(1)
  The number of transformer blocks.
- d_token :: integer(1)
  The dimension of the embedding.
- cardinalities :: integer(1)
  The number of categories for each categorical feature. This only needs to be specified when working with lazy_tensor inputs.
- init_token :: character(1)
  The initialization method for the embedding weights. Either "uniform" or "normal". "Uniform" by default.
- ingress_tokens :: named list() or NULL
  A list of TorchIngressTokens. Only required when using lazy tensor features. The names are either "num.input" or "categ.input", and the values are lazy tensor ingress tokens constructed by, e.g. ingress_ltnsr(<num_feat_name>).

Super classes

mlr3::Learner -> LearnerTorch -> LearnerTorchFTTransformer

Methods

Public methods:

- LearnerTorchFTTransformer$new()
- LearnerTorchFTTransformer$clone()

LearnerTorchFTTransformer$new(): Creates a new instance of this R6 class.

Usage:

```
LearnerTorchFTTransformer$new(
  task_type,
  optimizer = NULL,
  loss = NULL,
  callbacks = list()
)
```

Arguments:

task_type (character(1))
: The task type, either "classif" or "regr".

optimizer (TorchOptimizer)
: The optimizer to use for training. Per default, *adam* is used.

loss (TorchLoss)
: The loss used to train the network. Per default, *mse* is used for regression and *cross_entropy* for classification.

callbacks (list() of TorchCallbacks)
: The callbacks. Must have unique ids.

`LearnerTorchFTTransformer$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LearnerTorchFTTransformer$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Gorishniy Y, Rubachev I, Khrulkov V, Babenko A (2021). "Revisiting Deep Learning for Tabular Data." *arXiv*, **2106.11959**.

See Also

Other Learner: `GraphLearnerTorch`, `as_learner_torch()`, `mlr_learners.mlp`, `mlr_learners.module`, `mlr_learners.realmlp`, `mlr_learners.tab_resnet`, `mlr_learners.tabm`, `mlr_learners.torch_featureless`, `mlr_learners_torch`, `mlr_learners_torch_image`, `mlr_learners_torch_model`

Examples

```
# Define the Learner and set parameter values
learner = lrn("classif.ft_transformer")
learner$param_set$set_values(
  epochs = 1, batch_size = 16, device = "cpu",
  n_blocks = 2, d_token = 32, ffn_d_hidden_multiplier = 4/3
)

# Define a Task
task = tsk("iris")

# Create train and test set
ids = partition(task)

# Train the learner on the training ids
learner$train(task, row_ids = ids$train)

# Make predictions for the test rows
predictions = learner$predict(task, row_ids = ids$test)

# Score the predictions
predictions$score()
```

`mlr_learners.mlp` *Multi Layer Perceptron*

Description

Fully connected feed forward network with dropout after each activation function. The features can either be a single `lazy_tensor` or one or more numeric columns (but not both).

Dictionary

This Learner can be instantiated using the sugar function `lrn()`:

```
lrn("classif.mlp", ...)
lrn("regr.mlp", ...)
```

Properties

- Supported task types: 'classif', 'regr'
- Predict Types:
  - classif: 'response', 'prob'
  - regr: 'response'
- Feature Types: "integer", "numeric", "lazy_tensor"
- Required Packages: **mlr3**, **mlr3torch**, **torch**

Parameters

Parameters from `LearnerTorch`, as well as:

- `activation` :: `nn_module`
  The activation function. Is initialized to `nn_relu`.
- `activation_args` :: named `list()`
  A named list with initialization arguments for the activation function. This is intialized to an empty list.
- `neurons` :: `integer()`
  The number of neurons per hidden layer. By default there is no hidden layer. Setting this to `c(10, 20)` would have a the first hidden layer with 10 neurons and the second with 20.
- `n_layers` :: `integer()`
  The number of layers. This parameter must only be set when `neurons` has length 1.
- `p` :: `numeric(1)`
  The dropout probability. Is initialized to `0.1`.
- `shape` :: `integer()` or `NULL`
  The input shape of length 2, e.g. `c(NA, 5)`. Only needs to be present when there is a lazy tensor input with unknown shape (`NULL`). Otherwise the input shape is inferred from the number of numeric features.

Super classes

`mlr3::Learner` -> `LearnerTorch` -> `LearnerTorchMLP`

Methods

Public methods:

- `LearnerTorchMLP$new()`
- `LearnerTorchMLP$clone()`

`LearnerTorchMLP$new()`: Creates a new instance of this R6 class.

Usage:

```
LearnerTorchMLP$new(
  task_type,
  optimizer = NULL,
  loss = NULL,
  callbacks = list()
)
```

Arguments:

`task_type` (`character(1)`)
: The task type, either `"classif"` or `"regr"`.

`optimizer` (`TorchOptimizer`)
: The optimizer to use for training. Per default, *adam* is used.

`loss` (`TorchLoss`)
: The loss used to train the network. Per default, *mse* is used for regression and *cross_entropy* for classification.

`callbacks` (`list()` of `TorchCallbacks`)
: The callbacks. Must have unique ids.

`LearnerTorchMLP$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LearnerTorchMLP$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Gorishniy Y, Rubachev I, Khrulkov V, Babenko A (2021). "Revisiting Deep Learning for Tabular Data." *arXiv*, **2106.11959**.

See Also

Other Learner: `GraphLearnerTorch`, `as_learner_torch()`, `mlr_learners.ft_transformer`, `mlr_learners.module`, `mlr_learners.realmlp`, `mlr_learners.tab_resnet`, `mlr_learners.tabm`, `mlr_learners.torch_featureless`, `mlr_learners_torch`, `mlr_learners_torch_image`, `mlr_learners_torch_model`

Examples

```
# Define the Learner and set parameter values
learner = lrn("classif.mlp")
learner$param_set$set_values(
  epochs = 1, batch_size = 16, device = "cpu",
  neurons = 10
)

# Define a Task
task = tsk("iris")
```

```
# Create train and test set
ids = partition(task)

# Train the learner on the training ids
learner$train(task, row_ids = ids$train)

# Make predictions for the test rows
predictions = learner$predict(task, row_ids = ids$test)

# Score the predictions
predictions$score()
```

`mlr_learners.module` *Learner Torch Module*

Description

Wraps a `torch::nn_module` into a `LearnerTorch`, so an architecture written directly in `torch` can be trained, resampled and tuned like any other mlr3 learner without going through a `Graph` of `PipeOpTorch` operators. This is the shortest path from an existing `torch` model to mlr3, and the natural choice for an architecture whose forward pass is easier to write as code than to assemble from operators. The module receives the task's features as its arguments, which `ingress_tokens` maps.

Dictionary

This Learner can be instantiated using the sugar function `lrn()`:

```
lrn("classif.module", ...)
lrn("regr.module", ...)
```

Properties

- Supported task types: 'classif', 'regr'
- Predict Types:
  - classif: 'response', 'prob'
  - regr: 'response'
- Feature Types: "logical", "integer", "numeric", "character", "factor", "ordered", "POSIXct", "Date", "lazy_tensor"
- Required Packages: **mlr3**, **mlr3torch**, **torch**

Super classes

`mlr3::Learner` -> `LearnerTorch` -> `LearnerTorchModule`

Methods

Public methods:

- LearnerTorchModule$new()
- LearnerTorchModule$clone()

`LearnerTorchModule$new()`: Creates a new instance of this R6 class.

Usage:

```
LearnerTorchModule$new(
  module_generator = NULL,
  param_set = NULL,
  ingress_tokens = NULL,
  task_type,
  properties = NULL,
  optimizer = NULL,
  loss = NULL,
  callbacks = list(),
  packages = character(0),
  feature_types = NULL,
  predict_types = NULL,
  target_batchgetter = NULL
)
```

Arguments:

`module_generator` (`function` or `nn_module_generator`)
: A `nn_module_generator` or `function` returning an `nn_module`. Both must take as argument the `task` for which to construct the network. Other arguments to its initialize method can be provided as parameters.

`param_set` (`NULL` or `ParamSet`)
: If provided, contains the parameters for the module_generator. If `NULL`, parameters will be inferred from the module_generator.

`ingress_tokens` (`list` of `TorchIngressToken()`)
: A list with ingress tokens that defines how the dataset will be defined. The names must correspond to the arguments of the network's forward method. For numeric, categorical, and lazy tensor features, you can use `ingress_num()`, `ingress_categ()`, and `ingress_ltnsr()` to create them.

`task_type` (`character(1)`)
: The task type, either `"classif"` or `"regr"`.

`task_type` (`character(1)`)
: The task type.

`properties` (`NULL` or `character()`)
: The properties of the learner. Defaults to the properties that every `LearnerTorch` has, i.e. `"marshal"`, `"validation"` and `"internal_tuning"` (and `"twoclass"` and `"multiclass"` for classification). Other properties such as `"weights"`, `"missings"`, `"new_levels"` or `"offset"` must only be claimed if the network and the ingress tokens handle them, as mlr3 then skips its checks for them.

`optimizer` (`TorchOptimizer`)
: The optimizer to use for training. Per default, *adam* is used.

`loss` (TorchLoss)
: The loss used to train the network. Per default, *mse* is used for regression and *cross_entropy* for classification.

`callbacks` (`list()` of TorchCallbacks)
: The callbacks. Must have unique ids.

`packages` (`character()`)
: The R packages this object depends on.

`feature_types` (`NULL` or `character()`)
: The feature types. Defaults to all available feature types.

`predict_types` (`character()`)
: The predict types. See `mlr_reflections$learner_predict_types` for available values.

`target_batchgetter` (`function()` or `NULL`)
: Converts the target columns of a batch into the target tensor `y` that the loss is applied to. Takes an argument `data`, a `data.table` with only the target columns, and optionally an argument `x`, the named list of feature tensors of the batch, which is what a target that is a function of the input needs, see `get_target_batchgetter()`. If `NULL` (default), it is taken from the task via `get_target_batchgetter()`, which the built-in task types provide, and a TaskTorch only if it has a `default_target_batchgetter` or no target at all, in which case the batches have no `y` element.

`LearnerTorchModule$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LearnerTorchModule$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Learner: GraphLearnerTorch, `as_learner_torch()`, `mlr_learners.ft_transformer`, `mlr_learners.mlp`, `mlr_learners.realmlp`, `mlr_learners.tab_resnet`, `mlr_learners.tabm`, `mlr_learners.torch_featureless`, `mlr_learners_torch`, `mlr_learners_torch_image`, `mlr_learners_torch_model`

Examples

```
nn_one_layer = nn_module("nn_one_layer",
  initialize = function(task, size_hidden) {
    self$first = nn_linear(task$n_features, size_hidden)
    self$second = nn_linear(size_hidden, output_dim_for(task))
  },
  # argument x corresponds to the ingress token x
  forward = function(x) {
    x = self$first(x)
    x = nnf_relu(x)
    self$second(x)
  }
)
learner = lrn("classif.module",
  module_generator = nn_one_layer,
  ingress_tokens = list(x = ingress_num()),
```

```
  epochs = 10,
  size_hidden = 20,
  batch_size = 16
)
task = tsk("iris")
learner$train(task)
learner$network
```

`mlr_learners.realmlp` *RealMLP*

Description

RealMLP is an MLP for tabular data with tuned defaults (RealMLP-TD), introduced by Holzmüller (2024). The learner implements the preprocessing, architecture and training recipe of the reference implementation.

It is a special learner, as it implements it's own learning rate scheduling per parameter group, as well as custom weight decay handling. Configuring a custom learning reate scheduler as a callback or setting the weight decay of an optimizer will therefore lead to unexpected results.

Dictionary

This Learner can be instantiated using the sugar function `lrn()`:

```
lrn("classif.realmlp", ...)
lrn("regr.realmlp", ...)
```

Properties

- Supported task types: 'classif', 'regr'
- Predict Types:
  - classif: 'response', 'prob'
  - regr: 'response'
- Feature Types: "logical", "integer", "numeric", "factor", "ordered"
- Required Packages: **mlr3**, **mlr3torch**, **torch**

Parameters

Changed defaults of `LearnerTorch`:

- `epochs` = `256`
- `batch_size` = `256`
- `batch_size_predict` = `1024`
- `drop_last` = `TRUE` and the optimizer is *adam* with

- `betas` = `c(0.9, 0.95)`
- `lr` = `0.04` (classification) or `lr` = `0.2` (regression).

Parameters from `LearnerTorch`, as well as:

- `n_hidden_layers` :: `integer(1)`
  The number of hidden layers. Default is `3`.
- `hidden_width` :: `integer(1)`
  The width of the hidden layers. Default is `256`.
- `act` :: `character(1)`
  The activation function, one of `"selu"`, `"mish"`, `"relu"`, `"silu"`, `"gelu"` and `"elu"`. Default is `"selu"` for classification and `"mish"` for regression.
- `use_parametric_act` :: `logical(1)`
  Whether to use parametric activations $x + \alpha(f(x) - x)$ with a learned α per neuron, initialized with one. Default is `TRUE`.
- `p_drop` :: `numeric(1)`
  The dropout probability. Default is `0.15`.
- `p_drop_sched` :: `character(1)`
  The schedule of the dropout probability. Default is `"flat_cos"`.
- `wd` :: `numeric(1)`
  The decoupled weight decay. Biases are not decayed. Default is `0.02`.
- `wd_sched` :: `character(1)`
  The schedule of the weight decay. Default is `"flat_cos"`.
- `lr_sched` :: `character(1)`
  The schedule of the learning rate. Default is `"coslog4"`.
- `bias_lr_factor`, `act_lr_factor`, `scale_lr_factor`, `plr_lr_factor` :: `numeric(1)`
  The learning rate factors of the biases (default `0.1`), the parametric activations (default `0.1`), the scaling layer (default `6`) and the periodic embeddings (default `0.1`). All other parameters have the factor `1`.
- `add_front_scale` :: `logical(1)`
  Whether the first layer starts with a learned elementwise scaling. Default is `TRUE`.
- `num_emb_type` :: `character(1)`
  The embedding of the numerical features, one of:
  - `"pbld"` (default): $\mathrm{concat}(W_2 \cos(2\pi w_1 x + b_1) + b_2, x)$.
  - `"pblrd"`: as `"pbld"`, with a ReLU after the second linear layer.
  - `"pl"`: $W_2 \mathrm{concat}(\cos(2\pi w_1 x), \sin(2\pi w_1 x)) + b_2$.
  - `"plr"`: as `"pl"`, with a ReLU after the second linear layer.
  - `"none"`: no embedding.
- `plr_sigma` :: `numeric(1)`
  The standard deviation of the initialization of the frequencies w_1. Default is `0.1`.
- `plr_hidden_1` :: `integer(1)`
  The number of frequencies (`"pbld"`, `"pblrd"`) or twice the number of frequencies (`"pl"`, `"plr"`, must be even) per feature. Default is `16`.

- `plr_hidden_2` :: `integer(1)`
  The embedding size per feature, including the appended input of `"pbld"` and `"pblrd"`. Default is `4`.
- `max_one_hot_cat_size` :: `integer(1)`
  Categorical features with at most `max_one_hot_cat_size - 1` levels are one-hot encoded, all others are embedded. Default is `9`.
- `embedding_size` :: `integer(1)`
  The embedding size of the categorical features. Default is `8`.

Only for classification:

- `ls_eps` :: `numeric(1)`
  The label smoothing parameter. Requires the `cross_entropy` loss. The smoothing term ignores the loss's `class_weight` and `ignore_index`. Default is `0.1`.

Only for regression:

- `normalize_output` :: `logical(1)`
  Whether to standardize the target during training. Default is `TRUE`.
- `clamp_output` :: `logical(1)`
  Whether to clamp predictions to the range of the training target. Default is `TRUE`.

The schedules, as functions of the fraction t of training steps already taken, are:

- `"coslog4"`: $(1 - \cos(2\pi \log_2(1 + 15t)))/2$, four cycles of increasing length.
- `"flat_cos"`: one for $t < 0.5$, followed by a cosine decay to zero.
- `"cos"`: a cosine decay from one to zero.
- `"linear"`: $1 - t$.
- `"constant"`: one.

Super classes

`mlr3::Learner` -> `LearnerTorch` -> `LearnerTorchRealMLP`

Methods

Public methods:

- `LearnerTorchRealMLP$new()`
- `LearnerTorchRealMLP$clone()`

`LearnerTorchRealMLP$new()`**:** Creates a new instance of this R6 class.

Usage:

```
LearnerTorchRealMLP$new(
  task_type,
  optimizer = NULL,
  loss = NULL,
  callbacks = list()
)
```

Arguments:

`task_type` (`character(1)`)
The task type, either `"classif"` or `"regr"`.

`optimizer` (`TorchOptimizer`)
The optimizer to use for training. Per default, *adam* is used.

`loss` (`TorchLoss`)
The loss used to train the network. Per default, *mse* is used for regression and *cross_entropy* for classification.

`callbacks` (`list()` of `TorchCallback`s)
The callbacks. Must have unique ids.

`LearnerTorchRealMLP$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LearnerTorchRealMLP$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Holzmüller D, Grinsztajn L, Steinwart I (2024). "Better by Default: Strong Pre-Tuned MLPs and Boosted Trees on Tabular Data." In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37. 2407.04491.

See Also

Other Learner: `GraphLearnerTorch`, `as_learner_torch()`, `mlr_learners.ft_transformer`, `mlr_learners.mlp`, `mlr_learners.module`, `mlr_learners.tab_resnet`, `mlr_learners.tabm`, `mlr_learners.torch_featureless`, `mlr_learners_torch`, `mlr_learners_torch_image`, `mlr_learners_torch_model`

Examples

```
# Define the Learner and set parameter values
learner = lrn("classif.realmlp")
learner$param_set$set_values(
  epochs = 1, batch_size = 16, device = "cpu",
  n_hidden_layers = 2, hidden_width = 32
)

# Define a Task
task = tsk("iris")

# Create train and test set
ids = partition(task)

# Train the learner on the training ids
learner$train(task, row_ids = ids$train)

# Make predictions for the test rows
predictions = learner$predict(task, row_ids = ids$test)
```

```
# Score the predictions
predictions$score()
```

`mlr_learners.tab_resnet` *Tabular ResNet*

Description

A residual network for tabular data, following Gorishniy (2021).

The features are first projected to dimension `d_block` by a linear layer, followed by `n_blocks` residual blocks. Each block computes `x + Dropout(Linear(Dropout(ReLU(Linear(BatchNorm(x))))))`, where the two dropout ratios are `dropout1` and `dropout2`. The prediction is `Linear(ReLU(BatchNorm(x)))` of the output `x` of the last block.

Dictionary

This Learner can be instantiated using the sugar function `lrn()`:

```
lrn("classif.tab_resnet", ...)
lrn("regr.tab_resnet", ...)
```

Properties

- Supported task types: 'classif', 'regr'
- Predict Types:
  - classif: 'response', 'prob'
  - regr: 'response'
- Feature Types: "integer", "numeric", "lazy_tensor"
- Required Packages: **mlr3**, **mlr3torch**, **torch**

Parameters

Parameters from `LearnerTorch`, as well as:

- `n_blocks` :: `integer(1)`
  The number of blocks.
- `d_block` :: `integer(1)`
  The input and output dimension of a block.
- `d_hidden` :: `integer(1)`
  The latent dimension of a block.
- `d_hidden_multiplier` :: `numeric(1)`
  Alternative way to specify the latent dimension as `d_block * d_hidden_multiplier`.

- dropout1 :: numeric(1)
  First dropout ratio.
- dropout2 :: numeric(1)
  Second dropout ratio.
- shape :: integer() or NULL
  Shape of the input tensor. Only needs to be provided if the input is a lazy tensor with unknown shape.

Super classes

mlr3::Learner -> LearnerTorch -> LearnerTorchTabResNet

Methods

Public methods:

- LearnerTorchTabResNet$new()
- LearnerTorchTabResNet$clone()

LearnerTorchTabResNet$new(): Creates a new instance of this R6 class.

Usage:

```
LearnerTorchTabResNet$new(
  task_type,
  optimizer = NULL,
  loss = NULL,
  callbacks = list()
)
```

Arguments:

task_type (character(1))
: The task type, either "classif" or "regr".

optimizer (TorchOptimizer)
: The optimizer to use for training. Per default, *adam* is used.

loss (TorchLoss)
: The loss used to train the network. Per default, *mse* is used for regression and *cross_entropy* for classification.

callbacks (list() of TorchCallbacks)
: The callbacks. Must have unique ids.

LearnerTorchTabResNet$clone(): The objects of this class are cloneable with this method.

Usage:

```
LearnerTorchTabResNet$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

References

Gorishniy Y, Rubachev I, Khrulkov V, Babenko A (2021). "Revisiting Deep Learning for Tabular Data." *arXiv*, **2106.11959**.

See Also

Other Learner: GraphLearnerTorch, as_learner_torch(), mlr_learners.ft_transformer, mlr_learners.mlp, mlr_learners.module, mlr_learners.realmlp, mlr_learners.tabm, mlr_learners.torch_featureless, mlr_learners_torch, mlr_learners_torch_image, mlr_learners_torch_model

Examples

```
# Define the Learner and set parameter values
learner = lrn("classif.tab_resnet")
learner$param_set$set_values(
  epochs = 1, batch_size = 16, device = "cpu",
  n_blocks = 2, d_block = 10, d_hidden = 20, dropout1 = 0.3, dropout2 = 0.3
)

# Define a Task
task = tsk("iris")

# Create train and test set
ids = partition(task)

# Train the learner on the training ids
learner$train(task, row_ids = ids$train)

# Make predictions for the test rows
predictions = learner$predict(task, row_ids = ids$test)

# Score the predictions
predictions$score()
```

---

mlr_learners.tabm *TabM*

Description

TabM is an MLP-based tabular deep learning model that represents an ensemble of k MLPs: the k submodels are trained in parallel on the same batches and share most of their weights. The network produces k predictions per observation; the learner averages the predicted *probabilities* (classification) or the predicted values (regression) over the k submodels, and its loss function trains all k submodels jointly.

Numerical features are used as-is, or – if the num_embeddings parameter is set – embedded feature-wise first. Categorical features are one-hot encoded.

Dictionary

This Learner can be instantiated using the sugar function lrn():

```
lrn("classif.tabm", ...)
lrn("regr.tabm", ...)
```

Properties

- Supported task types: 'classif', 'regr'
- Predict Types:
  - classif: 'response', 'prob'
  - regr: 'response'
- Feature Types: "logical", "integer", "numeric", "factor", "ordered"
- Required Packages: **mlr3**, **mlr3torch**, **torch**

Parameters

Parameters from `LearnerTorch`, as well as:

- `arch_type` :: `character(1)`
  The architecture type, one of:
  - `"tabm"` (default) – BatchEnsemble with the TabM initialization, i.e. all multiplicative adapters except the very first one are initialized with ones.
  - `"tabm-mini"` – all non-shared parameters are concentrated in a single elementwise affine transformation applied to the input.
- `k` :: `integer(1)`
  The number of ensemble members. Default is `32`.
- `n_blocks` :: `integer(1)`
  The number of blocks of the MLP backbone. If unset, `2` is used when `num_embeddings` is set and `3` otherwise.
- `d_block` :: `integer(1)`
  The width of the MLP backbone. Default is `512`.
- `dropout` :: `numeric(1)`
  The dropout rate. Default is `0.1`.
- `activation` :: `character(1)`, `nn_module_generator` or `function`
  The activation function of the MLP backbone. Either the name of an activation of the `torch` package (e.g. `"relu"`, `"nn_relu"` or `"ReLU"`), an `nn_module_generator` such as `nn_relu`, or a function returning an `nn_module`. Default is `"relu"`.
- `start_scaling_init` :: `character(1)`
  The initialization of the very first (non-shared) scaling, either `"random-signs"` or `"normal"`. If unset, `"normal"` is used when `num_embeddings` is set and `"random-signs"` otherwise.

Parameters of the embeddings for the numerical features. The embedding types are those of Gorishniy (2022), see the *References* section for the paper that introduces and compares them:

- `num_embeddings` :: `character(1)`
  The type of the numerical feature embeddings, one of `"none"` (default), `"linear_relu"`, `"periodic"` or `"piecewise_linear"`.
- `d_embedding` :: `integer(1)`
  The embedding size. If unset, `32` is used for `"linear_relu"`, `24` for `"periodic"` and `16` for `"piecewise_linear"`.

- `n_frequencies` :: `integer(1)`
  `"periodic"` only: the number of frequencies per feature. Default is `48`.
- `frequency_init_scale` :: `numeric(1)`
  `"periodic"` only: the initialization scale of the frequencies. This is an important hyperparameter. Default is `0.01`.
- `lite` :: `logical(1)`
  `"periodic"` only: whether the outer linear layer is shared between all features. Default is `FALSE`.
- `embedding_activation` :: `logical(1)`
  `"periodic"` and `"piecewise_linear"` only: whether a ReLU is applied at the end of the embedding. If unset, `TRUE` is used for `"periodic"` and `FALSE` for `"piecewise_linear"`.
- `n_bins` :: `integer(1)`
  `"piecewise_linear"` only: the number of quantile bins, computed from the training data. Must be smaller than the number of training observations. Default is `48`.

Loss and Prediction

The network output has shape `(batch, k, d_out)`. At training time the learner therefore applies the configured loss to the `k` predictions separately: the ensemble dimension is folded into the batch dimension and each target is repeated `k` times. `$loss` itself is left untouched and stays whatever was configured. For prediction, the per-submodel probabilities (softmax for multiclass, sigmoid for binary) are averaged over the `k` submodels; for regression the outputs are averaged.

Super classes

`mlr3::Learner` -> `LearnerTorch` -> `LearnerTorchTabM`

Methods

Public methods:

- `LearnerTorchTabM$new()`
- `LearnerTorchTabM$clone()`

`LearnerTorchTabM$new()`**:** Creates a new instance of this R6 class.

Usage:

```
LearnerTorchTabM$new(
  task_type,
  optimizer = NULL,
  loss = NULL,
  callbacks = list()
)
```

Arguments:

`task_type` (`character(1)`)
: The task type, either `"classif"` or `"regr"`.

`optimizer` (`TorchOptimizer`)
: The optimizer to use for training. Per default, *adam* is used.

loss (TorchLoss)
: The loss used to train the network. Per default, *mse* is used for regression and *cross_entropy* for classification.

callbacks (list() of TorchCallbacks)
: The callbacks. Must have unique ids.

`LearnerTorchTabM$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LearnerTorchTabM$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

References

Gorishniy Y, Kotelnikov A, Babenko A (2025). "TabM: Advancing Tabular Deep Learning with Parameter-Efficient Ensembling." In *The Thirteenth International Conference on Learning Representations (ICLR)*. 2410.24210, https://openreview.net/forum?id=Sd4wYYOhmY.

Wen Y, Tran D, Ba J (2020). "BatchEnsemble: An Alternative Approach to Efficient Ensemble and Lifelong Learning." In *The Eighth International Conference on Learning Representations (ICLR)*. 2002.06715, https://openreview.net/forum?id=Sklf1yrYDr.

Gorishniy Y, Rubachev I, Babenko A (2022). "On Embeddings for Numerical Features in Tabular Deep Learning." In *Advances in Neural Information Processing Systems 35 (NeurIPS)*. 2203.05556, https://arxiv.org/abs/2203.05556.

See Also

Other Learner: GraphLearnerTorch, as_learner_torch(), mlr_learners.ft_transformer, mlr_learners.mlp, mlr_learners.module, mlr_learners.realmlp, mlr_learners.tab_resnet, mlr_learners.torch_featureless, mlr_learners_torch, mlr_learners_torch_image, mlr_learners_torch_model

Examples

```
# Define the Learner and set parameter values
learner = lrn("classif.tabm")
learner$param_set$set_values(
  epochs = 1, batch_size = 16, device = "cpu",
  k = 4, n_blocks = 2, d_block = 32
)

# Define a Task
task = tsk("iris")

# Create train and test set
ids = partition(task)

# Train the learner on the training ids
learner$train(task, row_ids = ids$train)

# Make predictions for the test rows
```

```
predictions = learner$predict(task, row_ids = ids$test)

# Score the predictions
predictions$score()
```

`mlr_learners.torch_featureless`
Featureless Torch Learner

Description

Featureless torch learner. Output is a constant weight that is learned during training. For classification, this should (asymptoptically) result in a majority class prediction when using the standard cross-entropy loss. For regression, this should result in the median for L1 loss and in the mean for L2 loss.

Dictionary

This Learner can be instantiated using the sugar function `lrn()`:

```
lrn("classif.torch_featureless", ...)
lrn("regr.torch_featureless", ...)
```

Properties

- Supported task types: 'classif', 'regr'
- Predict Types:
  - classif: 'response', 'prob'
  - regr: 'response'
- Feature Types: "logical", "integer", "numeric", "character", "factor", "ordered", "POSIXct", "Date", "lazy_tensor"
- Required Packages: **mlr3**, **mlr3torch**, **torch**

Parameters

Only those from `LearnerTorch`.

Super classes

`mlr3::Learner` -> `LearnerTorch` -> `LearnerTorchFeatureless`

Methods

Public methods:

- LearnerTorchFeatureless$new()
- LearnerTorchFeatureless$clone()

LearnerTorchFeatureless$new(): Creates a new instance of this R6 class.

Usage:

```
LearnerTorchFeatureless$new(
  task_type,
  optimizer = NULL,
  loss = NULL,
  callbacks = list()
)
```

Arguments:

task_type (character(1))
: The task type, either "classif" or "regr".

optimizer (TorchOptimizer)
: The optimizer to use for training. Per default, *adam* is used.

loss (TorchLoss)
: The loss used to train the network. Per default, *mse* is used for regression and *cross_entropy* for classification.

callbacks (list() of TorchCallbacks)
: The callbacks. Must have unique ids.

LearnerTorchFeatureless$clone(): The objects of this class are cloneable with this method.

Usage:

```
LearnerTorchFeatureless$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Learner: GraphLearnerTorch, as_learner_torch(), mlr_learners.ft_transformer, mlr_learners.mlp, mlr_learners.module, mlr_learners.realmlp, mlr_learners.tab_resnet, mlr_learners.tabm, mlr_learners_torch, mlr_learners_torch_image, mlr_learners_torch_model

Examples

```
# Define the Learner and set parameter values
learner = lrn("classif.torch_featureless")
learner$param_set$set_values(
  epochs = 1, batch_size = 16, device = "cpu"

)

# Define a Task
```

```
task = tsk("iris")

# Create train and test set
ids = partition(task)

# Train the learner on the training ids
learner$train(task, row_ids = ids$train)

# Make predictions for the test rows
predictions = learner$predict(task, row_ids = ids$test)

# Score the predictions
predictions$score()
```

`mlr_learners.torchvision`

Image Classification Networks from torchvision

Description

Well-known image classification networks from the `torchvision` package.

Each of these learners wraps one of the `model_*()` generators of `torchvision`. The section *Available Learners* below lists for every learner the original architecture that it implements, the paper that introduced this architecture, and the file in `torchvision` that contains the implementation, so the architecture can be inspected directly.

Available Learners

The table below lists for each learner the original architecture that it implements, the paper that introduced this architecture, and the file in `torchvision` that contains the implementation.

Learner	Architecture	Reference	Implementation
`classif.alexnet`	AlexNet	Krizhevsky (2017) and Krizhevsky (2014)	models-alexnet.R
`classif.convnext_tiny_1k`	ConvNeXt	Liu (2022)	models-convnext
`classif.convnext_tiny_22k`	ConvNeXt	Liu (2022)	models-convnext
`classif.convnext_small_22k`	ConvNeXt	Liu (2022)	models-convnext
`classif.convnext_base_1k`	ConvNeXt	Liu (2022)	models-convnext
`classif.convnext_base_22k`	ConvNeXt	Liu (2022)	models-convnext
`classif.convnext_large_1k`	ConvNeXt	Liu (2022)	models-convnext
`classif.convnext_large_22k`	ConvNeXt	Liu (2022)	models-convnext
`classif.efficientnet_b0`	EfficientNet	Tan (2019)	models-efficientn
`classif.efficientnet_b1`	EfficientNet	Tan (2019)	models-efficientn
`classif.efficientnet_b2`	EfficientNet	Tan (2019)	models-efficientn
`classif.efficientnet_b3`	EfficientNet	Tan (2019)	models-efficientn
`classif.efficientnet_b4`	EfficientNet	Tan (2019)	models-efficientn
`classif.efficientnet_b5`	EfficientNet	Tan (2019)	models-efficientn

classif.efficientnet_b6	EfficientNet	Tan (2019)	models-efficientn
classif.efficientnet_b7	EfficientNet	Tan (2019)	models-efficientn
classif.efficientnet_v2_s	EfficientNetV2	Tan (2021)	models-efficientn
classif.efficientnet_v2_m	EfficientNetV2	Tan (2021)	models-efficientn
classif.efficientnet_v2_l	EfficientNetV2	Tan (2021)	models-efficientn
classif.inception_v3	Inception v3	Szegedy (2016)	models-inception
classif.maxvit	MaxViT	Tu (2022)	models-maxvit.R
classif.mobilenet_v2	MobileNetV2	Sandler (2018)	models-mobilene
classif.mobilenet_v3_large	MobileNetV3	Howard (2019)	models-mobilene
classif.mobilenet_v3_small	MobileNetV3	Howard (2019)	models-mobilene
classif.resnet18	ResNet	He (2016)	models-resnet.R
classif.resnet34	ResNet	He (2016)	models-resnet.R
classif.resnet50	ResNet	He (2016)	models-resnet.R
classif.resnet101	ResNet	He (2016)	models-resnet.R
classif.resnet152	ResNet	He (2016)	models-resnet.R
classif.resnext50_32x4d	ResNeXt	He (2016) and Xie (2017)	models-resnet.R
classif.resnext101_32x8d	ResNeXt	He (2016) and Xie (2017)	models-resnet.R
classif.wide_resnet50_2	Wide ResNet	He (2016) and Zagoruyko (2016)	models-resnet.R
classif.wide_resnet101_2	Wide ResNet	He (2016) and Zagoruyko (2016)	models-resnet.R
classif.vgg11	VGG	Simonyan (2014)	models-vgg.R
classif.vgg11_bn	VGG	Simonyan (2014)	models-vgg.R
classif.vgg13	VGG	Simonyan (2014)	models-vgg.R
classif.vgg13_bn	VGG	Simonyan (2014)	models-vgg.R
classif.vgg16	VGG	Simonyan (2014)	models-vgg.R
classif.vgg16_bn	VGG	Simonyan (2014)	models-vgg.R
classif.vgg19	VGG	Simonyan (2014)	models-vgg.R
classif.vgg19_bn	VGG	Simonyan (2014)	models-vgg.R
classif.vit_b_16	Vision Transformer (ViT)	Dosovitskiy (2021)	models-vit.R
classif.vit_b_32	Vision Transformer (ViT)	Dosovitskiy (2021)	models-vit.R
classif.vit_l_16	Vision Transformer (ViT)	Dosovitskiy (2021)	models-vit.R
classif.vit_l_32	Vision Transformer (ViT)	Dosovitskiy (2021)	models-vit.R
classif.vit_h_14	Vision Transformer (ViT)	Dosovitskiy (2021)	models-vit.R

Parameters

Parameters from `LearnerTorchImage` and

- `pretrained` :: `logical(1)`
  Whether to use the pretrained model. The final linear layer will be replaced with a new `nn_linear` with the number of classes inferred from the `Task`.

`classif.inception_v3` additionally has

- `aux_logits` :: `logical(1)`
  Whether to enable the auxiliary classifier, which acts as a regularizer during training and is not applied when predicting. Defaults to `FALSE`. The configured loss is applied to the predictions of both classifiers and does not have to be changed for this. Note that the auxiliary classifier raises the minimum input size from 75x75 to 299x299, because it pools and convolves the feature map further than the main classifier does.

- `aux_weight` :: `numeric(1)`
  The weight with which the loss of the auxiliary classifier is added to the loss of the main classifier. Defaults to `0.4`, the value used by the Inception v3 paper. Only has an effect if `aux_logits` is `TRUE`.

Properties

- Supported task types: `"classif"`
- Predict Types: `"response"` and `"prob"`
- Feature Types: `"lazy_tensor"`
- Required packages: `"mlr3torch"`, `"torch"`, `"torchvision"`

Super classes

`mlr3::Learner` -> `LearnerTorch` -> `LearnerTorchImage` -> `LearnerTorchVision`

Methods

Public methods:

- `LearnerTorchVision$new()`
- `LearnerTorchVision$clone()`

`LearnerTorchVision$new()`: Creates a new instance of this R6 class.

Usage:

```
LearnerTorchVision$new(
  name,
  module_generator,
  label,
  optimizer = NULL,
  loss = NULL,
  callbacks = list(),
  jittable = FALSE,
  extra_param_set = NULL,
  network_args = character(0)
)
```

Arguments:

`name` (`character(1)`)
: The name of the network.

`module_generator` (`function(pretrained, num_classes)`)
: Function that generates the network.

`label` (`character(1)`)
: The label of the network.

`optimizer` (`TorchOptimizer`)
: The optimizer to use for training. Per default, *adam* is used.

`loss` (`TorchLoss`)
: The loss used to train the network. Per default, *mse* is used for regression and *cross_entropy* for classification.

`callbacks` (`list()` of `TorchCallbacks`)
The callbacks. Must have unique ids.

`jittable` (`logical(1)`)
Whether to use jitting.

`extra_param_set` (`ParamSet` or `NULL`)
Parameters that this network has in addition to `pretrained`, or `NULL` if it has none. They are added to the learner's `ParamSet`.

`network_args` (`character()`)
The ids of those parameters of `extra_param_set` that are passed on to `module_generator`. The remaining ones are interpreted by the learner itself, such as `aux_weight`, which is used to wrap the loss and never reaches the network.

`LearnerTorchVision$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LearnerTorchVision$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Krizhevsky A, Sutskever I, Hinton G (2017). "Imagenet classification with deep convolutional neural networks." *Communications of the ACM*, **60**(6), 84–90. Krizhevsky A (2014). "One weird trick for parallelizing convolutional neural networks." *arXiv preprint arXiv:1404.5997*. Liu Z, Mao H, Wu C, Feichtenhofer C, Darrell T, Xie S (2022). "A ConvNet for the 2020s." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 11976–11986. Tan M, Le Q (2019). "EfficientNet: Rethinking model scaling for convolutional neural networks." In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 6105–6114. PMLR. Tan M, Le Q (2021). "EfficientNetV2: Smaller models and faster training." In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, 10096–10106. PMLR. Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z (2016). "Rethinking the inception architecture for computer vision." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2818–2826. Tu Z, Talebi H, Zhang H, Yang F, Milanfar P, Bovik A, Li Y (2022). "MaxViT: Multi-axis vision transformer." In *Proceedings of the European Conference on Computer Vision (ECCV)*, 459–479. Sandler M, Howard A, Zhu M, Zhmoginov A, Chen L (2018). "Mobilenetv2: Inverted residuals and linear bottlenecks." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 4510–4520. Howard A, Sandler M, Chu G, Chen L, Chen B, Tan M, Wang W, Zhu Y, Pang R, Vasudevan V, Le Q, Adam H (2019). "Searching for MobileNetV3." In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 1314–1324. He K, Zhang X, Ren S, Sun J (2016). "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778. Xie S, Girshick R, Dollár P, Tu Z, He K (2017). "Aggregated residual transformations for deep neural networks." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 1492–1500. Zagoruyko S, Komodakis N (2016). "Wide residual networks." In *Proceedings of the British Machine Vision Conference (BMVC)*. 1605.07146, https://arxiv.org/abs/1605.07146. Simonyan K, Zisserman A (2014). "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556*. Dosovitskiy A, Beyer L, Kolesnikov A, Weissenborn D, Zhai X, Unterthiner T, Dehghani M, Minderer M, Heigold G, Gelly S, Uszkoreit J, Houlsby N (2021). "An image is worth

16x16 words: Transformers for image recognition at scale." In *The Ninth International Conference on Learning Representations (ICLR)*. 2010.11929, https://arxiv.org/abs/2010.11929.

`mlr_learners_torch` *Base Class for Torch Learners*

Description

This base class provides the basic functionality for training and prediction of a neural network. All torch learners should inherit from this class.

Validation

To specify the validation data, you can set the `$validate` field of the Learner, which can be set to:

- `NULL`: no validation
- `ratio`: only proportion `1 - ratio` of the task is used for training and `ratio` is used for validation.
- `"test"` means that the `"test"` task of a resampling is used and is not possible when calling `$train()` manually.
- `"predefined"`: This will use the predefined `$internal_valid_task` of a `mlr3::Task`.

This validation data can also be used for early stopping, see the description of the `Learner`'s parameters.

Saving a Learner

In order to save a `LearnerTorch` for later usage, it is necessary to call the `$marshal()` method on the `Learner` before writing it to disk, as the object will otherwise not be saved correctly. After loading a marshaled `LearnerTorch` into R again, you then need to call `$unmarshal()` to transform it into a useable state.

Early Stopping and Internal Tuning

In order to prevent overfitting, the `LearnerTorch` class supports early stopping via the `patience` and `min_delta` parameters, see the `Learner`'s parameters. When tuning a `LearnerTorch` it is also possible to combine the explicit tuning via `mlr3tuning` and the `LearnerTorch`'s internal tuning of the epochs via early stopping. To do so, you just need to include `epochs = to_tune(upper = <upper>, internal = TRUE)` in the search space, where `<upper>` is the maximally allowed number of epochs, and configure the early stopping.

Checkpointing and Resuming

It is possible to save intermediate results from a run via the `t_clbk("checkpoint")` callback. It is then possible to train for more epochs by setting the `resume` parameter of the `LearnerTorch`. This parameter can either be a path or `TRUE` which will use the path of the provided checkpoint callback. Only the number of epochs should be changed between resumed runs, other parameter changes are considered undefined behavior. Also, make sure to use the same train-validation split. When the latest written checkpoint was for `n1` epochs, the learner needs to be configured to be trained for `n >= n1` epochs and the training will run for `n2 = n - n1` epochs. With `n = n1` the checkpointed run is already finished, so nothing is trained and the model of the checkpoint is returned – which is what lets a script that restarts itself be run again after it succeeded, and what recovers the model of a run that was killed after its last epoch. Configuring `n < n1` is an error. Resuming will load the network weights, optimizer states and callback states. For some callbacks, training for `n1` and then `n2` epochs via resuming is not the same as training for `n` epochs from the start. This is for example the case for learning rate schedulers that depend on the total number of epochs to train for. The callbacks document their behavior under a corresponding *Resuming* section in their documentation. Furthermore, rng states are not restored, which constitutes another difference between a full and a resumed training run.

Network Head and Target Encoding

Torch learners are expected to have the following output:

- binary classification: `(batch_size, 1)`, representing the logits for the positive class.
- multiclass classification: `(batch_size, n_classes)`, representing the logits for all classes.
- regression: `(batch_size, 1)` representing the response prediction.

A network may return more than one tensor, in which case it returns a `list()` of them. There are two typical reasons for this:

- Networks with auxiliary classifiers such as `Inception v3` return additional predictions that only exist to contribute to the loss during training. Here, every element has the shape given above and the first one is the prediction of interest.
- A prediction that consists of several quantities – e.g. a mean and a standard deviation – is expressed by returning one tensor per quantity, which the prediction encoding combines.

In both cases the complete output is what the rest of the learner works with:

- The loss is applied to it. Because the configured loss expects a single tensor, a learner whose network returns a list has to wrap it in `.setup_training()`, see the list of methods below. `ContextTorch` makes the output available as `ctx$y_hats`.
- The prediction is encoded from it, both when predicting and when calculating the training and validation scores, so `.encode_prediction()` always receives the complete network output. Such a learner needs an `encode_prediction()` method for its task type, or has to overload the private `.encode_prediction()` method, because the encodings of the built-in task types expect a single tensor.

Note that the complete output is whatever the network returned in the mode it was called in, so a network whose extra tensors exist only during training – as auxiliary classifiers do – returns a different structure during training than during prediction, and `.encode_prediction()` has to handle both. `classif.inception_v3` does this by encoding only the prediction of the main classifier.

Furthermore, the target encoding is expected to be as follows:

- regression: The `numeric` target variable of a `TaskRegr` is encoded as a `torch_float` with shape `c(batch_size, 1)`.
- binary classification: The `factor` target variable of a `TaskClassif` is encoded as a `torch_float` with shape `(batch_size, 1)` where the positive class (`Task$positive`, which is also ensured to be the first factor level) is `1` and the negative class is `0`.
- multi-class classification: The `factor` target variable of a `TaskClassif` is a label-encoded `torch_long` with shape `(batch_size)` where the label-encoding goes from `1` to `n_classes`.

Predicting Tensors

The predict type `"lazy_tensor"`, available for the task type `"torch"`, hands back what the network produced – a `lazy_tensor` with one element per observation – instead of asking the task's `default_encoder` to turn it into a response. It is how to get at the logits of a classifier or the reconstruction of an autoencoder, and a task predicted this way needs no encoder at all. Unlike `"prob"` and `"se"` it is not opt-in: every learner for this task type has it among its `$predict_types`, because handing the output back does not depend on how the learner encodes a prediction. Set `learner$predict_type = "lazy_tensor"` to use it. A network with more than one head hands back one `lazy_tensor` per head, held in a `data.table` with one column per head so that the prediction is still one row per observation; `as.data.table()` spreads it into `lazy_tensor.<head>` columns.

Two things to know before using it:

- **Such a prediction does not survive `saveRDS()`.** It holds `torch` tensors, which are external pointers: saving *succeeds*, and the object then fails with *external pointer is not valid* the next time the tensors are touched, in this session or in another. This applies to a `ResampleResult` holding one as well – its row ids and scores survive, its tensors do not.
- Nothing about it is lazy. A `lazy_tensor` built from a tensor holds that tensor, so a prediction of this type is the network's output in memory, and `resample()` holds every fold's – combining the folds concatenates them, since lazy tensors from different networks share no data descriptor and cannot be concatenated lazily.

Important Runtime Considerations

There are a few hyperparameters settings that can have a considerable impact on the runtime of the learner. These include:

- `device`: Use a GPU if possible.
- `num_threads`: Set this to the number of CPU cores available if training on CPU. When resampling, benchmarking or tuning in parallel, each worker uses `num_threads` threads, so divide the available cores among the workers instead to avoid oversubscribing the machine.
- `tensor_dataset`: Set this to `TRUE` (or `"device"` if on a GPU) if the dataset fits into memory. This loads and stacks every batch once up front, so it must not be used with a `lazy_tensor` that applies random data augmentation – the augmentation would then be drawn only once.
- `batch_size`: Especially for very small models, choose a larger batch size.
- `batch_size_predict`: Prediction has no backward pass and so fits larger batches than training.

- `jit_trace`: Set this to `TRUE` to remove the per-batch R interpreter overhead, but only for a network whose control flow and shapes do not depend on the data. See the parameter's own entry.
- `num_workers`: Load batches in parallel worker processes.
- `pin_memory`: With a GPU, this speeds up the host-to-device copy of each batch.

Also, see the *Early Stopping and Internal Tuning* section for how to terminate training early.

Multiple GPUs

A single model is trained on one device. Set `device` to `"cuda"` for the default GPU, or to `"cuda:<index>"`, e.g. `"cuda:1"`, for a specific one.

To spread resampling, benchmarking or tuning over several GPUs, start one `mirai` daemon per GPU and restrict each daemon to its own GPU with the environment variable `CUDA_VISIBLE_DEVICES`. Inside a daemon, its GPU is then the only one, so `device = "cuda"` is correct for every resampling iteration. A device index such as `"cuda:1"` does not help here, because all iterations share the same parameter values. mlr3 uses the daemons of the compute profile `"mlr3_parallelization"`, and a daemon inherits the environment variables of the R session that launches it:

```
library(mirai)
gpus = c("0", "1")
daemons(url = local_url(), .compute = "mlr3_parallelization")
for (id in gpus) {
  Sys.setenv(CUDA_VISIBLE_DEVICES = id)
  launch_local(1, .compute = "mlr3_parallelization")
}
Sys.unsetenv("CUDA_VISIBLE_DEVICES")

learner = lrn("classif.mlp", device = "cuda", epochs = 50, batch_size = 64)
rr = resample(tsk("spam"), learner, rsmp("cv", folds = 10))
daemons(0, .compute = "mlr3_parallelization")
```

Each resampling iteration goes to the next free daemon, so all GPUs stay busy until fewer iterations than GPUs remain.

Things to avoid:

- `daemons(n)` alone: every daemon sees all GPUs, and `"cuda"` is the first GPU in each of them, so all iterations share that one GPU.
- Forking a process that has already initialized CUDA, e.g. with `future::plan("multicore")`.
- A high `num_threads`: data loading and preprocessing still run on the CPU in all daemons at once, so divide the CPU cores among them.

Several daemons per GPU only pay off when one model leaves the GPU underused, e.g. because it is small and training is bound by the per-batch overhead in R. Then all models must also fit into that GPU's memory together.

Model

The Model is a list of class `"learner_torch_model"` with the following elements:

- `network` :: The trained network.
- `optimizer` :: The `$state_dict()` optimizer used to train the network.
- `loss_fn` :: The `$state_dict()` of the loss used to train the network.
- `callbacks` :: The callbacks used to train the network.
- `seed` :: The seed that was / is used for training and prediction.
- `epochs` :: How many epochs the model was trained for (early stopping).
- `task_col_info` :: A `data.table()` containing information about the train-task.

Parameters

General:

The parameters of the optimizer, loss and callbacks, prefixed with `"opt."`, `"loss."` and `"cb.<callback id>."` respectively, as well as:

- `epochs` :: `integer(1)`
  The number of epochs.
- `device` :: `character(1)`
  The device. One of `"auto"`, `"cpu"`, or `"cuda"` or other values defined in `mlr_reflections$torch$devices`. A specific device can be selected by appending its index, e.g. `"cuda:1"` for the second GPU. The value is initialized to `"auto"`, which will select `"cuda"` if possible, then try `"mps"` and otherwise fall back to `"cpu"`.
- `num_threads` :: `integer(1)`
  The number of threads for intraop parallelization (if `device` is `"cpu"`). This value is initialized to 1. When resampling, benchmarking or tuning in parallel, each worker uses this many threads, so divide the available cores among the workers instead of setting this to the number of cores.
- `num_interop_threads` :: `integer(1)`
  The number of threads for interop parallelization (if `device` is `"cpu"`). Note that this can only be set **once** per session, so setting this for one learner also changes the behavior of other learners, and a later learner asking for a different value errors. `NULL` (default) uses whatever is set. In order to use different values for this parameter, use encapsulation to train the learners in separate R sessions.
- `seed` :: `integer(1)` or `"random"` or `NULL`
  The torch seed that is used during training and prediction. This value is initialized to `"random"`, which means that a random seed will be sampled at the beginning of the training phase. This seed (either set or randomly sampled) is available via `$model$seed` after training and used during prediction. Note that by setting the seed during the training phase this will mean that by default (i.e. when `seed` is `"random"`), clones of the learner will use a different seed. If set to `NULL`, no seeding will be done. This parameter only seeds torch's random number generator, it does **not** seed R's. Anything that is drawn from R's RNG is therefore unaffected by it, so to make those parts reproducible you need to seed R's RNG as well, e.g. via `set.seed()`.

- tensor_dataset :: logical(1) | "device"
  Whether to load all batches at once at the beginning of training and stack them. This is initialized to FALSE. If set to "device", the device of the tensors will be set to the value of device, which can avoid unnecessary moving of tensors between devices. When your dataset fits into memory this will make the loading of batches faster. Note that this should not be set for datasets that contain lazy_tensors with random data augmentation, as this augmentation will only be applied once at the beginning of training.
- cache_task_data :: logical(1)
  Whether to retrieve the data from the task once and to take the batches from it, instead of retrieving every batch from the task, which is slow. This is initialized to TRUE. Set it to FALSE if a copy of the task data does not fit into memory (lazy tensor columns only contain indices and are small).
- jit_trace :: logical(1)
  Whether to trace the network with torch::jit_trace() once at the start of training and then train the traced module instead of the original one. Not all learners support this. A traced network takes its inputs by position and ignores their names (mlverse/torch#1495). The learner passes the inputs in the right order, but when calling the traced $network of a trained learner with several inputs directly, they must be passed by position in the order given by attr(learner$network, "input_order").

Evaluation:

- measures_train :: Measure or list() of Measures
  Measures to be evaluated during training.
- measures_valid :: Measure or list() of Measures
  Measures to be evaluated during validation.
- eval_freq :: integer(1)
  How often the train / validation predictions are evaluated using measures_train / measures_valid. This is initialized to 1. Note that the final model is always evaluated.

Resuming:

- resume :: character(1) or TRUE
  Continues training from a checkpoint written by t_clbk("checkpoint"), either the folder it wrote to or TRUE, which takes that folder from the checkpoint callback of this learner. Note that epochs is the *total* number of epochs, i.e. it includes the epochs the checkpoint was already trained for: resuming a checkpoint from epoch 5 with epochs = 8 trains 3 more epochs.

Early Stopping:

- patience :: integer(1)
  This activates early stopping using the validation scores. If the performance of a model does not improve for patience evaluation steps, training is ended. Note that this counts *evaluation steps*, not epochs: when eval_freq is greater than 1, patience evaluation steps correspond to patience * eval_freq epochs. Note that the final model is stored in the learner, not the best model, unless restore_best_weights is set to TRUE. This is initialized to 0, which means no early stopping. The first entry from measures_valid is used as the metric. A missing (NA) score counts as no improvement, and every score improves on a missing one. If all scores are missing, the first evaluated epoch is reported as the best one. This also requires to specify the $validate field of the Learner, as well as measures_valid.

- `min_delta` :: `double(1)`
  The minimum improvement threshold for early stopping. Is initialized to 0.
- `restore_best_weights` :: `logical(1)`
  Whether to restore the weights of the best epoch when training ends, instead of keeping those of the last epoch that was trained. Like `min_delta`, this only has an effect when early stopping is active. Is initialized to `FALSE`, i.e. the network of the last epoch is stored. Setting this to `TRUE` makes the stored network the one of the epoch that `$internal_tuned_values` reports, and costs one additional copy of the network's parameters in memory.

Dataloader:

- `batch_size` :: `integer(1)`
  The batch size used by the training and prediction dataloader. It is required for training (unless a `batch_sampler` is provided, which already determines the batches) and it is required for prediction (unless `batch_size_predict` is set).
- `batch_size_predict` :: `integer(1)`
  The batch size used by the prediction dataloader (this includes the validation data during training). When set, it overrides `batch_size` for prediction. The batch size does not change the predictions, but smaller batches take longer and require less memory.
- `shuffle` :: `logical(1)`
  Whether to shuffle the instances in the dataset. This is initialized to `TRUE`, which differs from the default (`FALSE`). It is ignored when a `sampler` or `batch_sampler` is provided.
- `sampler` :: `torch::sampler`
  Object that defines how the dataloader draws samples, i.e. the order in which the observations are drawn. This must be the sampler *generator* (as returned by `torch::sampler()`), not an instance, as it is instantiated with the training dataset internally.
- `batch_sampler` :: `torch::sampler`
  Object that defines how the dataloader draws batches. As for `sampler`, this must be the generator. When it is provided, the parameters `batch_size`, `shuffle` and `drop_last` are ignored during training, because the batch sampler already determines the batches.
- `num_workers` :: `integer(1)`
  The number of workers for data loading (batches are loaded in parallel). The default is `0`, which means that data will be loaded in the main process.
- `collate_fn` :: `function`
  How to merge a list of samples to form a batch.
- `pin_memory` :: `logical(1)`
  Whether the dataloader copies tensors into CUDA pinned memory before returning them.
- `drop_last` :: `logical(1)`
  Whether to drop the last training batch in each epoch during training. Default is `FALSE`. If it is `TRUE`, the batch size is capped at the number of training observations, so that not all of them are dropped. It is ignored when a `batch_sampler` is provided.
- `timeout` :: `numeric(1)`
  The timeout value for collecting a batch from workers. Negative values mean no timeout and the default is `-1`.
- `worker_init_fn` :: `function(id)`
  A function that receives the worker id (in `[1, num_workers]`) and is executed after seeding on the worker but before data loading.

- worker_globals :: list() | character()
  When loading data in parallel, this makes it possible to export globals to the workers. If this is a character vector, the objects in the global environment with those names are copied to the workers.
- worker_packages :: character()
  Which packages to load on the workers.

Also see torch::dataloader for more information.

Inheriting

There are no separate classes for classification and regression to inherit from. Instead, the task_type must be specified as a construction argument. Any task type that is registered in mlr_reflections$task_types can be used. Support for a task type that **mlr3torch** does not know is added by implementing methods for the three S3 generics that hold the task-type-specific behaviour: output_dim_for() (how many output neurons the network needs), get_target_batchgetter() (how the target is turned into a tensor) and encode_prediction() (how the network's output is turned back into a prediction). Such a learner also has to be given a loss explicitly, unless a default loss is registered for the task type by adding the id of a loss from mlr3torch_losses to mlr_reflections$torch$default_losses, e.g. mlr_reflections$torch$default_losses$surv = "my_surv_loss". See the *Adding a New Task Type* article for a complete example. This class can also be used for custom task types, see TaskTorch and the *Custom Learning Problems* article for more information.

When inheriting from this class, one should overload the following methods:

- .network(task, param_vals)
  (Task, list()) -> nn_module
  Construct a torch::nn_module object for the given task and parameter values, i.e. the neural network that is trained by the learner. Note that a specific output shape is expected from the returned network, see section *Network Head and Target Encoding*. That section also describes when a network can return more than one tensor. You can use output_dim_for() to obtain the correct output dimension for a given task.
- .setup_training(ctx, param_vals)
  (ContextTorch, list()) -> NULL
  Customizes a training run by modifying the context in place; the default does nothing. It is called once the data loaders, the network and the configured loss (self$loss$generate(task)) are created, and before training (or resuming) starts. At this point, ctx$optimizer is NULL and ctx$callbacks contains the configured callbacks. The method may:
  - replace ctx$network. This is also the network that is used for prediction.
  - replace ctx$loss_fn, e.g. to wrap the configured loss when the network returns more than one prediction, see the aux_logits parameter of classif.inception_v3.
  - set ctx$optimizer, e.g. to self$optimizer$generate(groups) with parameter groups for parameter-specific learning rates. Otherwise, the optimizer is generated from the final ctx$network afterwards. Use self$optimizer$generate(), so that the optimizer's parameters (including param_groups) are respected, and set the optimizer after replacing the network.
  - add callbacks to ctx$callbacks. The configured callbacks may not be removed or replaced, and the id "early_stopping" is reserved.

Call `super$.setup_training(ctx, param_vals)` first. The method must be deterministic given `param_vals`, so that resuming from a checkpoint can restore the optimizer and callback states. With `jit_trace = TRUE`, `ctx$network` is already traced.

- `.ingress_tokens(task, param_vals)`
  (`Task`, `list()`) -> named `list()` with `TorchIngressToken`s
  Create the `TorchIngressToken`s that are passed to the `task_dataset` constructor. The number of ingress tokens must correspond to the number of input parameters of the network. If there is more than one input, the names must correspond to the inputs of the network. See `ingress_num`, `ingress_categ`, and `ingress_ltnsr` on how to easily create the correct tokens. For more flexibility, you can also directly implement the `.dataset(task, param_vals)` method, see below.
- `.dataset(task, param_vals)`
  (`Task`, `list()`) -> `torch::dataset`
  Create the dataset for the task. Don't implement this if the `.ingress_tokens()` method is defined. The dataset must return a named list where:
  - `x` is a list of torch tensors that are the input to the network. For networks with more than one input, the names must correspond to the inputs of the network.
  - `y` is the target tensor.
  - `.index` are the indices of the batch (`integer()` or a `torch_int()`).

  For information on the expected target encoding of `y`, see section *Network Head and Target Encoding*. Moreover, one needs to pay attention respect the row ids of the provided task. It is recommended to relu on `task_dataset` for creating the `dataset`.

It is also possible to overwrite the private `.dataloader()` method. This must respect the dataloader parameters from the `ParamSet`.

- `.dataloader(dataset, param_vals)`
  (`dataset`, `list()`) -> `torch::dataloader`
  Create a dataloader from the dataset. Needs to respect at least `batch_size` and `shuffle` (otherwise predictions will be incorrectly ordered). Use `get_batch_size(param_vals, "train")` to obtain the batch size for the respective phase, which takes the `batch_size_predict` parameter into account.

To change the predict types, it is possible to overwrite the method below:

- `.encode_prediction(network_output, task)`
  (`torch_tensor` or `list()` of them, `Task`) -> `list()`
  Take in the raw predictions from `self$network` (`network_output`) and encode them into a format that can be converted to valid `mlr3` predictions using `mlr3::as_prediction_data()`. For classification, `task` has its classes in the order of the training task. It is a `list()` of tensors when the network returns more than one, see section *Network Head and Target Encoding*. This method must take `self$predict_type` into account.

While it is possible to add parameters by specifying the `param_set` construction argument, it is currently not possible to remove existing parameters, i.e. those listed in section *Parameters*. None of the parameters provided in `param_set` can have an id that starts with `"loss."`, `"opt."`, or `"cb."`, as these are preserved for the dynamically constructed parameters of the optimizer, the loss function, and the callbacks.

To perform additional input checks on the task, the private `.check_train_task(task, param_vals)` and `.check_predict_task(task, param_vals)` can be overwritten. These should return `TRUE` if the input task is valid and otherwise a string with an error message.

For learners that have other construction arguments that should change the hash of a learner, it is required to implement the private `$.additional_phash_input()`.

Super class

`mlr3::Learner` -> `LearnerTorch`

Active bindings

`validate` How to construct the internal validation data. This parameter can be either `NULL`, a ratio in $(0, 1)$, `"test"`, or `"predefined"`.

`loss` (`TorchLoss`)
The torch loss.

`optimizer` (`TorchOptimizer`)
The torch optimizer.

`callbacks` (`list()` of `TorchCallback`s)
List of torch callbacks. The ids will be set as the names.

`internal_valid_scores` Retrieves the internal validation scores of the current network as a named `list()`. This is the *last* epoch, unless `restore_best_weights` is `TRUE`. Specify the `$validate` field and the `measures_valid` parameter to configure this. Returns `NULL` if learner is not trained yet.

`best_valid_scores` Retrieves the internal validation scores of the *best* epoch as a named `list()`. Only available when doing early stopping. Note that *best* here means with respect to the metric that was used for early stopping.

`internal_tuned_values` When early stopping is active, this returns a named list with the early-stopped epochs, otherwise an empty list is returned. Returns `NULL` if learner is not trained yet.

`marshaled` (`logical(1)`)
Whether the learner is marshaled.

`network` (`nn_module()`)
Shortcut for `learner$model$network`.

`param_set` (`ParamSet`)
The parameter set

`hash` (`character(1)`)
Hash (unique identifier) for this object.

`phash` (`character(1)`)
Hash (unique identifier) for this partial object, excluding some components which are varied systematically during tuning (parameter values).

Methods

Public methods:

- LearnerTorch$new()
- LearnerTorch$format()
- LearnerTorch$print()
- LearnerTorch$marshal()
- LearnerTorch$unmarshal()
- LearnerTorch$dataset()
- LearnerTorch$clone()

`LearnerTorch$new()`: Creates a new instance of this R6 class.

Usage:

```
LearnerTorch$new(
  id,
  task_type,
  param_set,
  properties = character(),
  man,
  label,
  feature_types,
  optimizer = NULL,
  loss = NULL,
  packages = character(),
  predict_types = NULL,
  callbacks = list(),
  jittable = FALSE
)
```

Arguments:

`id` (`character(1)`)
: The id for of the new object.

`task_type` (`character(1)`)
: The task type.

`param_set` (`ParamSet` or `alist()`)
: Either a parameter set, or an `alist()` containing different values of self, e.g. `alist(private$.param_set1, private$.param_set2)`, from which a `ParamSet` collection should be created.

`properties` (`character()`)
: The properties of the object. See `mlr_reflections$learner_properties` for available values.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

`label` (`character(1)`)
: Label for the new instance.

`feature_types` (`character()`)
: The feature types. See `mlr_reflections$task_feature_types` for available values, Additionally, `"lazy_tensor"` is supported.

optimizer (NULL or TorchOptimizer)
: The optimizer to use for training. Defaults to adam.

loss (NULL or TorchLoss)
: The loss to use for training. Defaults to the loss registered for the task type in mlr_reflections$torch$default_los which is MSE for regression and cross entropy for classification. For other task types there is no default unless one is registered, and the loss has to be given, because which loss is appropriate depends on the learning problem.

packages (character())
: The R packages this object depends on.

predict_types (character())
: The predict types. See mlr_reflections$learner_predict_types for available values. For regression, the default is "response". For classification, this defaults to "response" and "prob". For the task type "torch", it defaults to "response". For other task types, it defaults to all predict types that are registered for the task type. To deviate from the defaults, it is necessary to overwrite the private $.encode_prediction() method, see section *Inheriting*.

callbacks (list() of TorchCallbacks)
: The callbacks to use for training. Defaults to an empty list(), i.e. no callbacks. Within a stage they are called in the order in which they are provided, unless a callback requests otherwise via its $weight, see section *Ordering* of CallbackSet.

jittable (logical(1))
: Whether the model can be jit-traced. Default is FALSE.

LearnerTorch$format(): Helper for print outputs.

Usage:

```
LearnerTorch$format(...)
```

Arguments:

... (ignored).

LearnerTorch$print(): Prints the object.

Usage:

```
LearnerTorch$print(...)
```

Arguments:

... (any)
: Currently unused.

LearnerTorch$marshal(): Marshal the learner.

Usage:

```
LearnerTorch$marshal(...)
```

Arguments:

... (any)
: Additional parameters.

Returns: self

LearnerTorch$unmarshal(): Unmarshal the learner.

Usage:

```
LearnerTorch$unmarshal(...)
```

Arguments:

`...` (any)
Additional parameters.

Returns: self

`LearnerTorch$dataset()`: Create the dataset for a task.

Usage:

```
LearnerTorch$dataset(task)
```

Arguments:

`task` Task
The task

Returns: dataset

`LearnerTorch$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LearnerTorch$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Learner: GraphLearnerTorch, as_learner_torch(), mlr_learners.ft_transformer, mlr_learners.mlp, mlr_learners.module, mlr_learners.realmlp, mlr_learners.tab_resnet, mlr_learners.tabm, mlr_learners.torch_featureless, mlr_learners_torch_image, mlr_learners_torch_model

---

mlr_learners_torch_image

Image Learner

Description

Base Class for Image Learners. The features are assumed to be a single lazy_tensor column in RGB format.

Parameters

Parameters include those inherited from LearnerTorch and the param_set construction argument.

Super classes

mlr3::Learner -> LearnerTorch -> LearnerTorchImage

Methods

Public methods:

- LearnerTorchImage$new()
- LearnerTorchImage$clone()

`LearnerTorchImage$new()`: Creates a new instance of this R6 class.

Usage:

```
LearnerTorchImage$new(
  id,
  task_type,
  param_set = ps(),
  label,
  optimizer = NULL,
  loss = NULL,
  callbacks = list(),
  packages,
  man,
  properties = NULL,
  predict_types = NULL,
  jittable = FALSE
)
```

Arguments:

`id` (`character(1)`)
: The id for of the new object.

`task_type` (`character(1)`)
: The task type.

`param_set` (`ParamSet`)
: The parameter set.

`label` (`character(1)`)
: Label for the new instance.

`optimizer` (`TorchOptimizer`)
: The torch optimizer.

`loss` (`TorchLoss`)
: The loss to use for training.

`callbacks` (`list()` of `TorchCallback`s)
: The callbacks used during training. Must have unique ids. They are executed in the order in which they are provided, unless a callback requests otherwise via its `$weight`, see section *Ordering* of `CallbackSet`.

`packages` (`character()`)
: The R packages this object depends on.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

`properties` (`character()`)
: The properties of the object. See `mlr_reflections$learner_properties` for available values.

`predict_types` (`character()`)
The predict types. See `mlr_reflections$learner_predict_types` for available values.

`jittable` (`logical(1)`)
Whether the model can be jit-traced.

`LearnerTorchImage$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LearnerTorchImage$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Learner: `GraphLearnerTorch`, `as_learner_torch()`, `mlr_learners.ft_transformer`, `mlr_learners.mlp`, `mlr_learners.module`, `mlr_learners.realmlp`, `mlr_learners.tab_resnet`, `mlr_learners.tabm`, `mlr_learners.torch_featureless`, `mlr_learners_torch`, `mlr_learners_torch_model`

---

mlr_learners_torch_model

Learner Torch Model

Description

Create a torch learner from an instantiated `nn_module()`. For classification, the output of the network must be the scores (before the softmax).

Parameters

See `LearnerTorch`

Super classes

`mlr3::Learner` -> `LearnerTorch` -> `LearnerTorchModel`

Active bindings

`ingress_tokens` (named `list()` with `TorchIngressToken` or `NULL`)
The ingress tokens. Must be non-`NULL` when calling `$train()`.

Methods

Public methods:

- `LearnerTorchModel$new()`
- `LearnerTorchModel$clone()`

`LearnerTorchModel$new()`: Creates a new instance of this R6 class.

Usage:

```
LearnerTorchModel$new(
  network = NULL,
  ingress_tokens = NULL,
  task_type,
  properties = NULL,
  optimizer = NULL,
  loss = NULL,
  callbacks = list(),
  packages = character(0),
  feature_types = NULL,
  target_batchgetter = NULL,
  predict_types = NULL
)
```

Arguments:

`network` (`nn_module`)
: An instantiated `nn_module`. Is not cloned during construction. For classification, outputs must be the scores (before the softmax).

`ingress_tokens` (`list` of `TorchIngressToken()`)
: A list with ingress tokens that defines how the dataloader will be defined.

`task_type` (`character(1)`)
: The task type.

`properties` (`NULL` or `character()`)
: The properties of the learner. Defaults to the properties that every `LearnerTorch` has, i.e. `"marshal"`, `"validation"` and `"internal_tuning"` (and `"twoclass"` and `"multiclass"` for classification). Other properties such as `"weights"`, `"missings"`, `"new_levels"` or `"offset"` must only be claimed if the network and the ingress tokens handle them, as mlr3 then skips its checks for them.

`optimizer` (`TorchOptimizer`)
: The torch optimizer.

`loss` (`TorchLoss`)
: The loss to use for training.

`callbacks` (`list()` of `TorchCallbacks`)
: The callbacks used during training. Must have unique ids. They are executed in the order in which they are provided, unless a callback requests otherwise via its `$weight`, see section *Ordering* of `CallbackSet`.

`packages` (`character()`)
: The R packages this object depends on.

`feature_types` (`NULL` or `character()`)
: The feature types. Defaults to all available feature types.

`target_batchgetter` (`function()` or `NULL`)
: Converts the target columns of a batch into the target tensor `y` that the loss is applied to. Takes an argument `data`, a `data.table` with only the target columns, and optionally an argument `x`, the named list of feature tensors of the batch, which is what a target that is a function of the input needs, see `get_target_batchgetter()`. If `NULL` (default), it is taken from the task via `get_target_batchgetter()`, which the built-in task types provide, and

a TaskTorch only if it has a default_target_batchgetter or no target at all, in which case the batches have no y element.

predict_types (character())
The predict types. See mlr_reflections$learner_predict_types for available values.

LearnerTorchModel$clone(): The objects of this class are cloneable with this method.

Usage:

```
LearnerTorchModel$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Learner: GraphLearnerTorch, as_learner_torch(), mlr_learners.ft_transformer, mlr_learners.mlp, mlr_learners.module, mlr_learners.realmlp, mlr_learners.tab_resnet, mlr_learners.tabm, mlr_learners.torch_featureless, mlr_learners_torch, mlr_learners_torch_image

Other Graph Network: ModelDescriptor(), TorchIngressToken(), as_learner_torch(), mlr_pipeops_module, mlr_pipeops_torch, mlr_pipeops_torch_ingress, mlr_pipeops_torch_ingress_categ, mlr_pipeops_torch_ingres mlr_pipeops_torch_ingress_num, model_descriptor_to_learner(), model_descriptor_to_module(), model_descriptor_union(), nn_graph(), pipeop_torch()

Examples

```
# We show the learner using a classification task

# The iris task has 4 features and 3 classes
network = nn_linear(4, 3)
task = tsk("iris")

# This defines the dataloader.
# It loads all 4 features, which are also numeric.
# The shape is (NA, 4) because the batch dimension is generally NA
ingress_tokens = list(
  input = TorchIngressToken(task$feature_names, batchgetter_num, c(NA, 4))
)

# Creating the learner and setting required parameters
learner = lrn("classif.torch_model",
  network = network,
  ingress_tokens = ingress_tokens,
  batch_size = 16,
  epochs = 1,
  device = "cpu"
)

# A simple train-predict
ids = partition(task)
learner$train(task, ids$train)
learner$predict(task, ids$test)
```

`mlr_measures_torch` *Measure for a Generic Torch Task*

Description

Wraps a plain R function into a `Measure` that scores the predictions of a `TaskTorch`. Use `msr_torch()` to construct one. See the *Custom Learning Problems* article for how to create and use such measures.

Super class

`mlr3::Measure` -> `MeasureTorch`

Active bindings

`hash (character(1))`
The hash of the measure.

Methods

Public methods:

- `MeasureTorch$new()`
- `MeasureTorch$clone()`

`MeasureTorch$new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureTorch$new(
  id,
  fun,
  minimize = NA,
  range = c(-Inf, Inf),
  predict_type = "response",
  properties = character(),
  label = NA_character_,
  obs_loss = NULL
)
```

Arguments:

`id (character(1))`
The id of the measure.

`fun (function())`
The scoring function. It receives whichever of the arguments `truth`, `response`, `prob`, `se`, `lazy_tensor`, `prediction`, `task`, `learner`, `train_set` and `weights` it declares, and must return a single number. Asking for anything else is an error.

`minimize (logical(1))`
Whether a smaller score is better. `NA` (default) means the direction is unknown.

`range` (`numeric(2)`)
: The range of possible scores.

`predict_type` (`character(1)`)
: The predict type the measure requires: `"response"` (default), `"prob"`, `"se"` or `"lazy_tensor"`. A measure asking for one it did not declare here still receives it, if the prediction has it.

`properties` (`character()`)
: Properties of the measure, see `Measure`. The `"requires_task"`, `"requires_learner"`, `"requires_train_set"` and `"weights"` properties are added automatically when `fun` declares the corresponding argument.

`label` (`character(1)`)
: The label of the measure.

`obs_loss` (`function()` or `NULL`)
: The per-observation loss. Declared like `fun`, except that `train_set` is not available here, and if specified adds the `"obs_loss"` property. It must return one number per observation.

`MeasureTorch$clone()`: The objects of this class are cloneable with this method.

Usage:

```
MeasureTorch$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Measure: `mlr_measures_torch.default`

Examples

```
d = data.frame(x = rnorm(10), y = rnorm(10))
task = as_task_torch(d, target = "y")
measure = msr_torch("mse", function(truth, response) mean((truth - response)^2))
measure
```

---

mlr_measures_torch.default

Default Measure of a Generic Torch Task

Description

This is a simple placeholder measure and extracts the actual value from the `$default_measure` of a `TaskTorch`.

Super class

`mlr3::Measure` -> `MeasureTorchDefault`

Methods

Public methods:

- `MeasureTorchDefault$new()`
- `MeasureTorchDefault$clone()`

`MeasureTorchDefault$new()`: Creates a new instance of this R6 class.

Usage:

```
MeasureTorchDefault$new()
```

`MeasureTorchDefault$clone()`: The objects of this class are cloneable with this method.

Usage:

```
MeasureTorchDefault$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Measure: `mlr_measures_torch`

Examples

```
d = data.frame(x = rnorm(10), y = rnorm(10))
task = as_task_torch(d, target = "y",
 default_measure = msr_torch("mse", function(truth, response) mean((truth - response)^2)))
msr("torch.default")
```

`mlr_pipeops_augment_center_crop`

Center Crop Augmentation

Description

Calls `torchvision::transform_center_crop`, see there for more information on the parameters. The preprocessing is applied to the whole batch.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("augment_center_crop")
```

Parameters

Id	Type	Default	Levels
size	untyped	-	
stages	character	-	train, predict, both
affect_columns	untyped	selector_all()	

`mlr_pipeops_augment_color_jitter`
Color Jitter Augmentation

Description

Calls `torchvision::transform_color_jitter`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("augment_color_jitter")
```

Parameters

Id	Type	Default	Levels	Range
brightness	numeric	0		$[0, \infty)$
contrast	numeric	0		$[0, \infty)$
saturation	numeric	0		$[0, \infty)$
hue	numeric	0		$[0, \infty)$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

mlr_pipeops_augment_crop

Crop Augmentation

Description

Calls `torchvision::transform_crop`, see there for more information on the parameters. The preprocessing is applied to the whole batch.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("augment_crop")
```

Parameters

Id	Type	Default	Levels	Range
top	integer	-		$[1, \infty)$
left	integer	-		$[1, \infty)$
height	integer	-		$[0, \infty)$
width	integer	-		$[0, \infty)$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

mlr_pipeops_augment_hflip

Horizontal Flip Augmentation

Description

Calls `torchvision::transform_hflip`, see there for more information on the parameters. The preprocessing is applied to the whole batch.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

R6Class inheriting from PipeOpTaskPreprocTorch.

Construction

```
po("augment_hflip")
```

Parameters

Id	Type	Default	Levels
stages	character	-	train, predict, both
affect_columns	untyped	selector_all()	

---

mlr_pipeops_augment_random_affine

Random Affine Augmentation

Description

Calls torchvision::transform_random_affine, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being an augment_ operator, its stages parameter starts out as "train", so the augmentation is applied while training and not when predicting. Set stages = "both" to apply it in both phases. See PipeOpTaskPreprocTorch for the stages parameter and this naming convention.

Format

R6Class inheriting from PipeOpTaskPreprocTorch.

Construction

```
po("augment_random_affine")
```

Parameters

Id	Type	Default	Levels	Range
degrees	untyped	-		-
translate	untyped	NULL		-
scale	untyped	NULL		-
resample	integer	0		$(-\infty, \infty)$
fillcolor	untyped	0		-
stages	character	-	train, predict, both	-

affect_columns	untyped	selector_all()	-

`mlr_pipeops_augment_random_choice`
Random Choice Augmentation

Description

Calls `torchvision::transform_random_choice`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("augment_random_choice")
```

Parameters

Id	Type	Default	Levels
transforms	untyped	-	
stages	character	-	train, predict, both
affect_columns	untyped	selector_all()	

mlr_pipeops_augment_random_crop

Random Crop Augmentation

Description

Calls `torchvision::transform_random_crop`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("augment_random_crop")
```

Parameters

Id	Type	Default	Levels
size	untyped	-	
padding	untyped	NULL	
pad_if_needed	logical	FALSE	TRUE, FALSE
fill	untyped	0L	
padding_mode	character	constant	constant, edge, reflect, symmetric
stages	character	-	train, predict, both
affect_columns	untyped	selector_all()	

mlr_pipeops_augment_random_horizontal_flip

Random Horizontal Flip Augmentation

Description

Calls `torchvision::transform_random_horizontal_flip`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

R6Class inheriting from PipeOpTaskPreprocTorch.

Construction

```
po("augment_random_horizontal_flip")
```

Parameters

Id	Type	Default	Levels	Range
p	numeric	0.5		$[0, 1]$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

`mlr_pipeops_augment_random_order`

Random Order Augmentation

Description

Calls `torchvision::transform_random_order`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See PipeOpTaskPreprocTorch for the `stages` parameter and this naming convention.

Format

R6Class inheriting from PipeOpTaskPreprocTorch.

Construction

```
po("augment_random_order")
```

Parameters

Id	Type	Default	Levels
transforms	untyped	-	
stages	character	-	train, predict, both
affect_columns	untyped	selector_all()	

mlr_pipeops_augment_random_resized_crop

Random Resized Crop Augmentation

Description

Calls `torchvision::transform_random_resized_crop`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("augment_random_resized_crop")
```

Parameters

Id	Type	Default	Levels	Range
size	untyped	-		-
scale	untyped	c(0.08, 1)		-
ratio	untyped	c(3/4, 4/3)		-
interpolation	integer	2		$[0, 3]$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

mlr_pipeops_augment_random_vertical_flip

Random Vertical Flip Augmentation

Description

Calls `torchvision::transform_random_vertical_flip`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

R6Class inheriting from PipeOpTaskPreprocTorch.

Construction

```
po("augment_random_vertical_flip")
```

Parameters

Id	Type	Default	Levels	Range
p	numeric	0.5		$[0, 1]$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

---

mlr_pipeops_augment_resized_crop

Resized Crop Augmentation

Description

Calls torchvision::transform_resized_crop, see there for more information on the parameters. The preprocessing is applied to the whole batch.

Being an augment_ operator, its stages parameter starts out as "train", so the augmentation is applied while training and not when predicting. Set stages = "both" to apply it in both phases. See PipeOpTaskPreprocTorch for the stages parameter and this naming convention.

Format

R6Class inheriting from PipeOpTaskPreprocTorch.

Construction

```
po("augment_resized_crop")
```

Parameters

Id	Type	Default	Levels	Range
top	integer	-		$[1, \infty)$
left	integer	-		$[1, \infty)$
height	integer	-		$[0, \infty)$
width	integer	-		$[0, \infty)$
size	untyped	-		-

interpolation	integer	2		$[0, 3]$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

`mlr_pipeops_augment_rotate`
Rotate Augmentation

Description

Calls `torchvision::transform_rotate`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("augment_rotate")
```

Parameters

Id	Type	Default	Levels	Range
angle	untyped	-		-
resample	integer	0		$(-\infty, \infty)$
expand	logical	FALSE	TRUE, FALSE	-
center	untyped	NULL		-
fill	untyped	NULL		-
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

mlr_pipeops_augment_vflip *Vertical Flip Augmentation*

Description

Calls `torchvision::transform_vflip`, see there for more information on the parameters. The preprocessing is applied to the whole batch.

Being an `augment_` operator, its `stages` parameter starts out as `"train"`, so the augmentation is applied while training and not when predicting. Set `stages = "both"` to apply it in both phases. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("augment_vflip")
```

Parameters

Id	Type	Default	Levels
stages	character	-	train, predict, both
affect_columns	untyped	selector_all()	

mlr_pipeops_module *Class for Torch Module Wrappers*

Description

`PipeOpModule` wraps an `nn_module` or `function` that is being called during the `train` phase of this `mlr3pipelines::PipeOp`. By doing so, this makes it possible to assemble `PipeOpModules` in a computational `mlr3pipelines::Graph` that represents either a neural network or a preprocessing graph of a `lazy_tensor`. In most cases it is easier to create such a network by creating a graph that generates this graph.

In most cases it is easier to create such a network by creating a structurally related graph consisting of nodes of class `PipeOpTorchIngress` and `PipeOpTorch`. This graph will then generate the graph consisting of `PipeOpModules` as part of the `ModelDescriptor`.

Input and Output Channels

The number and names of the input and output channels can be set during construction. They input and output `"torch_tensor"` during training, and `NULL` during prediction as the prediction phase currently serves no meaningful purpose.

State

The state is the value calculated by the public method `shapes_out()`.

Parameters

No parameters.

Internals

During training, the wrapped `nn_module` / `function` is called with the provided inputs in the order in which the channels are defined. Arguments are **not** matched by name.

Super class

`mlr3pipelines::PipeOp` -> `PipeOpModule`

Public fields

`module` (`nn_module`)
The torch module that is called during the training phase.

Methods

Public methods:

- `PipeOpModule$new()`
- `PipeOpModule$clone()`

`PipeOpModule$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpModule$new(
  id = "module",
  module = nn_identity(),
  inname = "input",
  outname = "output",
  param_vals = list(),
  packages = character(0)
)
```

Arguments:

`id` (`character(1)`)
The id for of the new object.

`module` (`nn_module` or `function()`)
The torch module or function that is being wrapped.

`inname` (`character()`)
The names of the input channels.

`outname` (`character()`)
The names of the output channels. If this parameter has length 1, the parameter module must return a tensor. Otherwise it must return a `list()` of tensors of corresponding length.

`param_vals` (named `list()`)
Parameter values to be set after construction.

`packages` (`character()`)
The R packages this object depends on.

`PipeOpModule$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpModule$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Graph Network: `ModelDescriptor()`, `TorchIngressToken()`, `as_learner_torch()`, `mlr_learners_torch_model`, `mlr_pipeops_torch`, `mlr_pipeops_torch_ingress`, `mlr_pipeops_torch_ingress_categ`, `mlr_pipeops_torch_ingres`, `mlr_pipeops_torch_ingress_num`, `model_descriptor_to_learner()`, `model_descriptor_to_module()`, `model_descriptor_union()`, `nn_graph()`, `pipeop_torch()`

Examples

```
## creating an PipeOpModule manually

# one input and output channel
po_module = po("module",
  id = "linear",
  module = torch::nn_linear(10, 20),
  inname = "input",
  outname = "output"
)
x = torch::torch_randn(16, 10)
# This calls the forward function of the wrapped module.
y = po_module$train(list(input = x))
str(y)

# multiple input and output channels
nn_custom = torch::nn_module("nn_custom",
  initialize = function(in_features, out_features) {
    self$lin1 = torch::nn_linear(in_features, out_features)
    self$lin2 = torch::nn_linear(in_features, out_features)
  },
  forward = function(x, z) {
    list(out1 = self$lin1(x), out2 = torch::nnf_relu(self$lin2(z)))
  }
)
```

```
module = nn_custom(3, 2)
po_module = po("module",
  id = "custom",
  module = module,
  inname = c("x", "z"),
  outname = c("out1", "out2")
)
x = torch::torch_randn(1, 3)
z = torch::torch_randn(1, 3)
out = po_module$train(list(x = x, z = z))
str(out)

# How such a PipeOpModule is usually generated
graph = po("torch_ingress_num") %>>% nn("linear", out_features = 10L)
result = graph$train(tsk("iris"))
# The PipeOpTorchLinear generates a PipeOpModule and adds it to a new (module) graph
result[[1]]$graph
linear_module = result[[1L]]$graph$pipeops$nn_linear
linear_module
formalArgs(linear_module$module)
linear_module$input$name

# Constructing a PipeOpModule using a simple function
po_add1 = po("module",
  id = "add_one",
  module = function(x) x + 1
)
input = list(torch_tensor(1))
po_add1$train(input)$output
```

`mlr_pipeops_nn_adaptive_avg_pool1d`
1D Adaptive Average Pooling

Description

Applies a 1D adaptive average pooling over an input signal composed of several input planes.

nn_module

Calls `nn_adaptive_avg_pool1d()` during training.

Parameters

- `output_size` :: `integer(1)`
  The target output size. A single number.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchAdaptiveAvgPool -> PipeOpTorchAdaptiveAvgPool1D

Methods

Public methods:

- PipeOpTorchAdaptiveAvgPool1D$new()
- PipeOpTorchAdaptiveAvgPool1D$clone()

PipeOpTorchAdaptiveAvgPool1D$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchAdaptiveAvgPool1D$new(
  id = "nn_adaptive_avg_pool1d",
  param_vals = list()
)
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchAdaptiveAvgPool1D$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchAdaptiveAvgPool1D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("adaptive_avg_pool1d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_adaptive_avg_pool2d` *2D Adaptive Average Pooling*

Description

Applies a 2D adaptive average pooling over an input signal composed of several input planes.

nn_module

Calls `nn_adaptive_avg_pool2d()` during training.

Parameters

- `output_size` :: `integer()`
  The target output size. Can be a single number or a vector.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchAdaptiveAvgPool` -> `PipeOpTorchAdaptiveAvgPool2D`

Methods

Public methods:

- `PipeOpTorchAdaptiveAvgPool2D$new()`
- `PipeOpTorchAdaptiveAvgPool2D$clone()`

`PipeOpTorchAdaptiveAvgPool2D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchAdaptiveAvgPool2D$new(
  id = "nn_adaptive_avg_pool2d",
  param_vals = list()
)
```

Arguments:

`id` (`character(1)`)
Identifier of the resulting object.

param_vals (list())
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchAdaptiveAvgPool2D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchAdaptiveAvgPool2D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("adaptive_avg_pool2d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_adaptive_avg_pool3d`
3D Adaptive Average Pooling

Description

Applies a 3D adaptive average pooling over an input signal composed of several input planes.

nn_module

Calls `nn_adaptive_avg_pool3d()` during training.

Parameters

- `output_size` :: `integer()`
  The target output size. Can be a single number or a vector.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchAdaptiveAvgPool` -> `PipeOpTorchAdaptiveAvgPool3D`

Methods

Public methods:

- `PipeOpTorchAdaptiveAvgPool3D$new()`
- `PipeOpTorchAdaptiveAvgPool3D$clone()`

`PipeOpTorchAdaptiveAvgPool3D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchAdaptiveAvgPool3D$new(
  id = "nn_adaptive_avg_pool3d",
  param_vals = list()
)
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchAdaptiveAvgPool3D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchAdaptiveAvgPool3D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("adaptive_avg_pool3d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_adaptive_max_pool1d`
1D Adaptive Max Pooling

Description

Applies a 1D adaptive max pooling over an input signal composed of several input planes.

nn_module

Calls `nn_adaptive_max_pool1d()` during training.

Parameters

- `output_size` :: `integer(1)`
  The target output size. A single number.

Input and Output Channels

There is one input channel `"input"`. The module has two outputs, `"output"` and `"indices"`, of which only `"output"` is an output channel by default. Set `$outputs` to also (or only) get the indices, e.g. `outputs = c("output", "indices")`, which are only computed when they are among `$outputs`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchAdaptiveMaxPool` -> `PipeOpTorchAdaptiveMaxPool1D`

Methods

Public methods:

- `PipeOpTorchAdaptiveMaxPool1D$new()`
- `PipeOpTorchAdaptiveMaxPool1D$clone()`

`PipeOpTorchAdaptiveMaxPool1D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchAdaptiveMaxPool1D$new(
  id = "nn_adaptive_max_pool1d",
  param_vals = list()
)
```

Arguments:

`id` (`character(1)`)
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchAdaptiveMaxPool1D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchAdaptiveMaxPool1D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("adaptive_max_pool1d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_adaptive_max_pool2d`
2D Adaptive Max Pooling

Description

Applies a 2D adaptive max pooling over an input signal composed of several input planes.

nn_module

Calls `nn_adaptive_max_pool2d()` during training.

Parameters

- `output_size` :: `integer()`
  The target output size. Can be a single number or a vector.

State

The state is the value calculated by the public method `$shapes_out()`.

Input and Output Channels

There is one input channel `"input"`. The module has two outputs, `"output"` and `"indices"`, of which only `"output"` is an output channel by default. Set `$outputs` to also (or only) get the indices, e.g. `outputs = c("output", "indices")`, which are only computed when they are among `$outputs`. For an explanation see `PipeOpTorch`.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchAdaptiveMaxPool -> PipeOpTorchAdaptiveMaxPool2D

Methods

Public methods:

- PipeOpTorchAdaptiveMaxPool2D$new()
- PipeOpTorchAdaptiveMaxPool2D$clone()

`PipeOpTorchAdaptiveMaxPool2D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchAdaptiveMaxPool2D$new(
  id = "nn_adaptive_max_pool2d",
  param_vals = list()
)
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchAdaptiveMaxPool2D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchAdaptiveMaxPool2D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("adaptive_max_pool2d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_adaptive_max_pool3d` *3D Adaptive Max Pooling*

Description

Applies a 3D adaptive max pooling over an input signal composed of several input planes.

nn_module

Calls `nn_adaptive_max_pool3d()` during training.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `output_size` :: `integer()`
  The target output size. Can be a single number or a vector.

Input and Output Channels

There is one input channel `"input"`. The module has two outputs, `"output"` and `"indices"`, of which only `"output"` is an output channel by default. Set `$outputs` to also (or only) get the indices, e.g. `outputs = c("output", "indices")`, which are only computed when they are among `$outputs`. For an explanation see `PipeOpTorch`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchAdaptiveMaxPool` -> `PipeOpTorchAdaptiveMaxPool3D`

Methods

Public methods:

- `PipeOpTorchAdaptiveMaxPool3D$new()`
- `PipeOpTorchAdaptiveMaxPool3D$clone()`

`PipeOpTorchAdaptiveMaxPool3D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchAdaptiveMaxPool3D$new(
  id = "nn_adaptive_max_pool3d",
  param_vals = list()
)
```

Arguments:

`id` (`character(1)`)
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchAdaptiveMaxPool3D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchAdaptiveMaxPool3D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("adaptive_max_pool3d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_avg_pool1d` *1D Average Pooling*

Description

Applies a 1D average pooling over an input signal composed of several input planes.

nn_module

Calls `nn_avg_pool1d()` during training.

Parameters

- `kernel_size` :: `integer()`
  The size of the window. Can be a single number or a vector.
- `stride` :: `integer()`
  The stride of the window. Can be a single number or a vector. Default: `kernel_size`.
- `padding` :: `integer()`
  Implicit zero paddings on both sides of the input. Can be a single number or a vector. Default: 0.
- `ceil_mode` :: `logical(1)`
  When `TRUE`, will use ceil instead of floor to compute the output shape. Default: `FALSE`.
- `count_include_pad` :: `logical(1)`
  When `TRUE`, will include the zero-padding in the averaging calculation. Default: `TRUE`.

- `divisor_override` :: `numeric(1)`
  If specified, it will be used as divisor, otherwise size of the pooling region will be used. Default: NULL. Only available for `nn_avg_pool2d` and `nn_avg_pool3d`, i.e. this parameter does not exist for `nn_avg_pool1d`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchAvgPool` -> `PipeOpTorchAvgPool1D`

Methods

Public methods:

- `PipeOpTorchAvgPool1D$new()`
- `PipeOpTorchAvgPool1D$clone()`

`PipeOpTorchAvgPool1D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchAvgPool1D$new(id = "nn_avg_pool1d", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchAvgPool1D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchAvgPool1D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("avg_pool1d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_avg_pool2d` *2D Average Pooling*

Description

Applies 2D average-pooling operation in $kH * kW$ regions by step size $sH * sW$ steps. The number of output features is equal to the number of input planes.

nn_module

Calls `nn_avg_pool2d()` during training.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `kernel_size` :: `integer()`
  The size of the window. Can be a single number or a vector.
- `stride` :: `integer()`
  The stride of the window. Can be a single number or a vector. Default: `kernel_size`.
- `padding` :: `integer()`
  Implicit zero paddings on both sides of the input. Can be a single number or a vector. Default: 0.
- `ceil_mode` :: `logical(1)`
  When `TRUE`, will use ceil instead of floor to compute the output shape. Default: `FALSE`.
- `count_include_pad` :: `logical(1)`
  When `TRUE`, will include the zero-padding in the averaging calculation. Default: `TRUE`.
- `divisor_override` :: `numeric(1)`
  If specified, it will be used as divisor, otherwise size of the pooling region will be used. Default: NULL. Only available for `nn_avg_pool2d` and `nn_avg_pool3d`, i.e. this parameter does not exist for `nn_avg_pool1d`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchAvgPool` -> `PipeOpTorchAvgPool2D`

Methods

Public methods:

- `PipeOpTorchAvgPool2D$new()`
- `PipeOpTorchAvgPool2D$clone()`

`PipeOpTorchAvgPool2D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchAvgPool2D$new(id = "nn_avg_pool2d", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchAvgPool2D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchAvgPool2D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("avg_pool2d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_avg_pool3d` *3D Average Pooling*

Description

Applies 3D average-pooling operation in $kT * kH * kW$ regions by step size $sT * sH * sW$ steps. The number of output features is equal to $\lfloor \frac{\text{input planes}}{sT} \rfloor$.

Internals

Calls `nn_avg_pool3d()` during training.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Parameters

- kernel_size :: integer()
  The size of the window. Can be a single number or a vector.
- stride :: integer()
  The stride of the window. Can be a single number or a vector. Default: kernel_size.
- padding :: integer()
  Implicit zero paddings on both sides of the input. Can be a single number or a vector. Default: 0.
- ceil_mode :: logical(1)
  When TRUE, will use ceil instead of floor to compute the output shape. Default: FALSE.
- count_include_pad :: logical(1)
  When TRUE, will include the zero-padding in the averaging calculation. Default: TRUE.
- divisor_override :: numeric(1)
  If specified, it will be used as divisor, otherwise size of the pooling region will be used. Default: NULL. Only available for nn_avg_pool2d and nn_avg_pool3d, i.e. this parameter does not exist for nn_avg_pool1d.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchAvgPool -> PipeOpTorchAvgPool3D

Methods

Public methods:

- PipeOpTorchAvgPool3D$new()
- PipeOpTorchAvgPool3D$clone()

PipeOpTorchAvgPool3D$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchAvgPool3D$new(id = "nn_avg_pool3d", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchAvgPool3D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchAvgPool3D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("avg_pool3d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_batch_norm1d` *1D Batch Normalization*

Description

Applies Batch Normalization for each channel across a batch of data.

nn_module

Calls `torch::nn_batch_norm1d()`. The parameter `num_features` is inferred as the second dimension of the input shape.

Parameters

- `eps` :: `numeric(1)`
  A value added to the denominator for numerical stability. Default: `1e-5`.
- `momentum` :: `numeric(1)`
  The value used for the running_mean and running_var computation. Can be set to `NULL` for cumulative moving average (i.e. simple average). Default: 0.1
- `affine` :: `logical(1)`
  a boolean value that when set to `TRUE`, this module has learnable affine parameters. Default: `TRUE`
- `track_running_stats` :: `logical(1)`
  a boolean value that when set to `TRUE`, this module tracks the running mean and variance, and when set to `FALSE`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `TRUE`

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchBatchNorm` -> `PipeOpTorchBatchNorm1D`

Methods

Public methods:

- `PipeOpTorchBatchNorm1D$new()`
- `PipeOpTorchBatchNorm1D$clone()`

`PipeOpTorchBatchNorm1D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchBatchNorm1D$new(id = "nn_batch_norm1d", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchBatchNorm1D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchBatchNorm1D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("batch_norm1d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_batch_norm2d` *2D Batch Normalization*

Description

Applies Batch Normalization for each channel across a batch of data.

nn_module

Calls `torch::nn_batch_norm2d()`. The parameter `num_features` is inferred as the second dimension of the input shape.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `eps` :: `numeric(1)`
  A value added to the denominator for numerical stability. Default: `1e-5`.
- `momentum` :: `numeric(1)`
  The value used for the running_mean and running_var computation. Can be set to `NULL` for cumulative moving average (i.e. simple average). Default: 0.1
- `affine` :: `logical(1)`
  a boolean value that when set to `TRUE`, this module has learnable affine parameters. Default: `TRUE`
- `track_running_stats` :: `logical(1)`
  a boolean value that when set to `TRUE`, this module tracks the running mean and variance, and when set to `FALSE`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `TRUE`

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchBatchNorm` -> `PipeOpTorchBatchNorm2D`

Methods

Public methods:

- `PipeOpTorchBatchNorm2D$new()`
- `PipeOpTorchBatchNorm2D$clone()`

`PipeOpTorchBatchNorm2D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchBatchNorm2D$new(id = "nn_batch_norm2d", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchBatchNorm2D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchBatchNorm2D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("batch_norm2d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_batch_norm3d`

3D Batch Normalization

Description

Applies Batch Normalization for each channel across a batch of data.

nn_module

Calls `torch::nn_batch_norm3d()`. The parameter `num_features` is inferred as the second dimension of the input shape.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `eps` :: `numeric(1)`
  A value added to the denominator for numerical stability. Default: `1e-5`.
- `momentum` :: `numeric(1)`
  The value used for the running_mean and running_var computation. Can be set to `NULL` for cumulative moving average (i.e. simple average). Default: 0.1
- `affine` :: `logical(1)`
  a boolean value that when set to `TRUE`, this module has learnable affine parameters. Default: `TRUE`
- `track_running_stats` :: `logical(1)`
  a boolean value that when set to `TRUE`, this module tracks the running mean and variance, and when set to `FALSE`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `TRUE`

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchBatchNorm` -> `PipeOpTorchBatchNorm3D`

Methods

Public methods:

- `PipeOpTorchBatchNorm3D$new()`
- `PipeOpTorchBatchNorm3D$clone()`

`PipeOpTorchBatchNorm3D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchBatchNorm3D$new(id = "nn_batch_norm3d", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchBatchNorm3D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchBatchNorm3D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("batch_norm3d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_block` *Block Repetition*

Description

Repeat a block `n_blocks` times by concatenating it with itself (via `%>>%`).

Naming

For the generated module graph, the IDs of the modules are generated by prefixing the IDs of the `n_blocks` layers with the ID of the `PipeOpTorchBlock` and postfixing them with `__<layer>`.

Parameters

The parameters available for the provided `block`, as well as

- `n_blocks` :: `integer(1)`
  How often to repeat the block.
- `trafo` :: `function(i, param_vals, param_set) -> list()`
  A function that transforms the parameter values of each layer (`block`). Here,
  - `i` :: `integer(1)`
    is the index of the layer, ranging from `1` to `n_blocks`.
  - `param_vals` :: named `list()`
    are the parameter values of the layer `i`.
  - `param_set` :: `ParamSet`
    is the parameter set of the whole `PipeOpTorchBlock`.

  The function must return the modified parameter values for the given layer. This, e.g., allows for special behavior of the first or last layer.

Input and Output Channels

The `PipeOp` sets its input and output channels to those from the `block` (Graph) it received during construction.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchBlock`

Active bindings

`block` (`Graph`)
The neural network segment that is repeated by this `PipeOp`.

Methods

Public methods:

- PipeOpTorchBlock$new()
- PipeOpTorchBlock$clone()

`PipeOpTorchBlock$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchBlock$new(block, id = "nn_block", param_vals = list())
```

Arguments:

`block` (Graph)
: A graph consisting primarily of PipeOpTorch objects that is to be repeated.

`id` (`character(1)`)
: The id for of the new object.

`param_vals` (named `list()`)
: Parameter values to be set after construction.

`PipeOpTorchBlock$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchBlock$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# repeat a simple linear layer with ReLU activation 3 times, but set the bias for the last
# layer to `FALSE`
block = nn("linear") %>>% nn("relu")

blocks = nn("block", block,
  linear.out_features = 10L, linear.bias = TRUE, n_blocks = 3,
  trafo = function(i, param_vals, param_set) {
    if (i  == param_set$get_values()$n_blocks) {
      param_vals$linear.bias = FALSE
    }
    param_vals
  })
graph = po("torch_ingress_num") %>>%
  blocks %>>%
  nn("head")
md = graph$train(tsk("iris"))[[1L]]
network = model_descriptor_to_module(md)
network
```

`mlr_pipeops_nn_celu` *CELU Activation Function*

Description

Applies element-wise, $CELU(x) = max(0, x) + min(0, \alpha * (exp(x\alpha) - 1))$.

nn_module

Calls `torch::nn_celu()` when trained.

Parameters

- `alpha` :: `numeric(1)`
  The alpha value for the ELU formulation. Default: 1.0
- `inplace` :: `logical(1)`
  Whether to do the operation in-place. Default: `FALSE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchCELU`

Methods

Public methods:

- `PipeOpTorchCELU$new()`
- `PipeOpTorchCELU$clone()`

`PipeOpTorchCELU$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchCELU$new(id = "nn_celu", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchCELU$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchCELU$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("celu")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_conv_transpose1d` *Transpose 1D Convolution*

Description

Applies a 1D transposed convolution operator over an input signal composed of several input planes, sometimes also called "deconvolution".

nn_module

Calls `nn_conv_transpose1d`. The parameter `in_channels` is inferred as the second dimension of the input tensor.

Parameters

- `out_channels` :: `integer(1)`
  Number of output channels produce by the convolution.
- `kernel_size` :: `integer()`
  Size of the convolving kernel.
- `stride` :: `integer()`
  Stride of the convolution. Default: 1.
- `padding` :: `integer()`
  `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of the input. Default: 0.
- `output_padding` :: `integer()`
  Additional size added to one side of the output shape. Default: 0.
- `groups` :: `integer()`
  Number of blocked connections from input channels to output channels. Default: 1

- `bias` :: `logical(1)`
  If `True`, adds a learnable bias to the output. Default: `TRUE`.
- `dilation` :: `integer()`
  Spacing between kernel elements. Default: 1.
- `padding_mode` :: `character(1)`
  The padding mode. One of `"zeros"`, `"reflect"`, `"replicate"`, or `"circular"`. Default is `"zeros"`.

State

The state is the value calculated by the public method `$shapes_out()`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchConvTranspose` -> `PipeOpTorchConvTranspose1D`

Methods

Public methods:

- `PipeOpTorchConvTranspose1D$new()`
- `PipeOpTorchConvTranspose1D$clone()`

`PipeOpTorchConvTranspose1D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchConvTranspose1D$new(id = "nn_conv_transpose1d", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchConvTranspose1D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchConvTranspose1D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("conv_transpose1d", kernel_size = 3, out_channels = 2)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_conv_transpose2d`
Transpose 2D Convolution

Description

Applies a 2D transposed convolution operator over an input image composed of several input planes, sometimes also called "deconvolution".

nn_module

Calls `nn_conv_transpose2d`. The parameter `in_channels` is inferred as the second dimension of the input tensor.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `out_channels` :: `integer(1)`
  Number of output channels produce by the convolution.
- `kernel_size` :: `integer()`
  Size of the convolving kernel.
- `stride` :: `integer()`
  Stride of the convolution. Default: 1.
- `padding` :: `integer()`
  `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of the input. Default: 0.
- `output_padding` :: `integer()`
  Additional size added to one side of the output shape. Default: 0.
- `groups` :: `integer()`
  Number of blocked connections from input channels to output channels. Default: 1

- bias :: logical(1)
  If True, adds a learnable bias to the output. Default: TRUE.
- dilation :: integer()
  Spacing between kernel elements. Default: 1.
- padding_mode :: character(1)
  The padding mode. One of "zeros", "reflect", "replicate", or "circular". Default is "zeros".

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchConvTranspose -> PipeOpTorchConvTranspose2D

Methods

Public methods:

- PipeOpTorchConvTranspose2D$new()
- PipeOpTorchConvTranspose2D$clone()

PipeOpTorchConvTranspose2D$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchConvTranspose2D$new(id = "nn_conv_transpose2d", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchConvTranspose2D$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchConvTranspose2D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("conv_transpose2d", kernel_size = 3, out_channels = 2)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_conv_transpose3d` *Transpose 3D Convolution*

Description

Applies a 3D transposed convolution operator over an input image composed of several input planes, sometimes also called "deconvolution"

nn_module

Calls `nn_conv_transpose3d`. The parameter `in_channels` is inferred as the second dimension of the input tensor.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `out_channels` :: `integer(1)`
  Number of output channels produce by the convolution.
- `kernel_size` :: `integer()`
  Size of the convolving kernel.
- `stride` :: `integer()`
  Stride of the convolution. Default: 1.
- `padding` :: `integer()`
  `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of the input. Default: 0.
- `output_padding` :: `integer()`
  Additional size added to one side of the output shape. Default: 0.
- `groups` :: `integer()`
  Number of blocked connections from input channels to output channels. Default: 1
- `bias` :: `logical(1)`
  If `True`, adds a learnable bias to the output. Default: `TRUE`.
- `dilation` :: `integer()`
  Spacing between kernel elements. Default: 1.
- `padding_mode` :: `character(1)`
  The padding mode. One of `"zeros"`, `"reflect"`, `"replicate"`, or `"circular"`. Default is `"zeros"`.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchConvTranspose -> PipeOpTorchConvTranspose3D

Methods

Public methods:

- PipeOpTorchConvTranspose3D$new()
- PipeOpTorchConvTranspose3D$clone()

`PipeOpTorchConvTranspose3D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchConvTranspose3D$new(id = "nn_conv_transpose3d", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchConvTranspose3D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchConvTranspose3D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("conv_transpose3d", kernel_size = 3, out_channels = 2)
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_conv1d *1D Convolution*

Description

Applies a 1D convolution over an input signal composed of several input planes.

nn_module

Calls torch::nn_conv1d() when trained. The paramter `in_channels` is inferred from the second dimension of the input tensor.

Parameters

- `out_channels` :: `integer(1)`
  Number of channels produced by the convolution.
- `kernel_size` :: `integer()`
  Size of the convolving kernel.
- `stride` :: `integer()`
  Stride of the convolution. The default is 1.
- `padding` :: `integer()`
  `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of the input. Default: 0.
- `groups` :: `integer()`
  Number of blocked connections from input channels to output channels. Default: 1
- `bias` :: `logical(1)`
  If `TRUE`, adds a learnable bias to the output. Default: `TRUE`.
- `dilation` :: `integer()`
  Spacing between kernel elements. Default: 1.
- `padding_mode` :: `character(1)`
  The padding mode. One of `"zeros"`, `"reflect"`, `"replicate"`, or `"circular"`. Default is `"zeros"`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchConv` -> `PipeOpTorchConv1D`

Methods

Public methods:

- `PipeOpTorchConv1D$new()`
- `PipeOpTorchConv1D$clone()`

`PipeOpTorchConv1D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchConv1D$new(id = "nn_conv1d", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

param_vals (list())
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchConv1D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchConv1D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("conv1d", kernel_size = 10, out_channels = 1)
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_conv2d *2D Convolution*

Description

Applies a 2D convolution over an input image composed of several input planes.

nn_module

Calls `torch::nn_conv2d()` when trained. The paramter `in_channels` is inferred from the second dimension of the input tensor.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `out_channels` :: `integer(1)`
  Number of channels produced by the convolution.
- `kernel_size` :: `integer()`
  Size of the convolving kernel.

- `stride` :: `integer()`
  Stride of the convolution. The default is 1.
- `padding` :: `integer()`
  `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of the input. Default: 0.
- `groups` :: `integer()`
  Number of blocked connections from input channels to output channels. Default: 1
- `bias` :: `logical(1)`
  If `TRUE`, adds a learnable bias to the output. Default: `TRUE`.
- `dilation` :: `integer()`
  Spacing between kernel elements. Default: 1.
- `padding_mode` :: `character(1)`
  The padding mode. One of `"zeros"`, `"reflect"`, `"replicate"`, or `"circular"`. Default is `"zeros"`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchConv` -> `PipeOpTorchConv2D`

Methods

Public methods:

- `PipeOpTorchConv2D$new()`
- `PipeOpTorchConv2D$clone()`

`PipeOpTorchConv2D$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchConv2D$new(id = "nn_conv2d", param_vals = list())
```

Arguments:

`id` (`character(1)`)
Identifier of the resulting object.

`param_vals` (`list()`)
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchConv2D$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchConv2D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("conv2d", kernel_size = 10, out_channels = 1)
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_conv3d *3D Convolution*

Description

Applies a 3D convolution over an input image composed of several input planes.

nn_module

Calls `torch::nn_conv3d()` when trained. The paramter `in_channels` is inferred from the second dimension of the input tensor.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `out_channels` :: `integer(1)`
  Number of channels produced by the convolution.
- `kernel_size` :: `integer()`
  Size of the convolving kernel.
- `stride` :: `integer()`
  Stride of the convolution. The default is 1.
- `padding` :: `integer()`
  `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of the input. Default: 0.
- `groups` :: `integer()`
  Number of blocked connections from input channels to output channels. Default: 1
- `bias` :: `logical(1)`
  If `TRUE`, adds a learnable bias to the output. Default: `TRUE`.
- `dilation` :: `integer()`
  Spacing between kernel elements. Default: 1.

- padding_mode :: character(1)
  The padding mode. One of "zeros", "reflect", "replicate", or "circular". Default is "zeros".

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchConv -> PipeOpTorchConv3D

Methods

Public methods:

- PipeOpTorchConv3D$new()
- PipeOpTorchConv3D$clone()

PipeOpTorchConv3D$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchConv3D$new(id = "nn_conv3d", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchConv3D$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchConv3D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("conv3d", kernel_size = 10, out_channels = 1)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_dropout` *Dropout*

Description

During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution.

nn_module

Calls `torch::nn_dropout()` when trained.

Parameters

- p :: `numeric(1)`
  Probability of an element to be zeroed. Default: 0.5.
- `inplace` :: `logical(1)`
  If set to `TRUE`, will do this operation in-place. Default: `FALSE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchDropout`

Methods

Public methods:

- `PipeOpTorchDropout$new()`
- `PipeOpTorchDropout$clone()`

`PipeOpTorchDropout$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchDropout$new(id = "nn_dropout", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchDropout$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchDropout$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("dropout")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_dropout2d` *2D Dropout*

Description

Randomly zero out entire channels (a channel is a 2D feature map, e.g., the j-th channel of the i-th sample in the batched input is a 2D tensor $input[i, j]$) of the input tensor). Each channel will be zeroed out independently on every forward call with probability `p` using samples from a Bernoulli distribution.

nn_module

Calls `torch::nn_dropout2d()` when trained.

Parameters

- `p` :: `numeric(1)`
  Probability of a channel to be zeroed. Default: 0.5.
- `inplace` :: `logical(1)`
  If set to `TRUE`, will do this operation in-place. Default: `FALSE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchDropoutNd` -> `PipeOpTorchDropout2D`

Methods

Public methods:

- `PipeOpTorchDropout2D$new()`
- `PipeOpTorchDropout2D$clone()`

`PipeOpTorchDropout2D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchDropout2D$new(id = "nn_dropout2d", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchDropout2D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchDropout2D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("dropout2d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_dropout3d` *3D Dropout*

Description

Randomly zero out entire channels (a channel is a 3D feature map, e.g., the j-th channel of the i-th sample in the batched input is a 3D tensor $input[i, j]$) of the input tensor). Each channel will be zeroed out independently on every forward call with probability `p` using samples from a Bernoulli distribution.

nn_module

Calls `torch::nn_dropout3d()` when trained.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `p` :: `numeric(1)`
  Probability of a channel to be zeroed. Default: 0.5.
- `inplace` :: `logical(1)`
  If set to `TRUE`, will do this operation in-place. Default: `FALSE`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchDropoutNd` -> `PipeOpTorchDropout3D`

Methods

Public methods:

- `PipeOpTorchDropout3D$new()`
- `PipeOpTorchDropout3D$clone()`

`PipeOpTorchDropout3D$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchDropout3D$new(id = "nn_dropout3d", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchDropout3D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchDropout3D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("dropout3d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_elu` *ELU Activation Function*

Description

Applies element-wise,

$$ELU(x) = max(0, x) + min(0, \alpha * (exp(x) - 1))$$

.

nn_module

Calls `torch::nn_elu()` when trained.

Parameters

- `alpha` :: `numeric(1)`
  The alpha value for the ELU formulation. Default: 1.0
- `inplace` :: `logical(1)`
  Whether to do the operation in-place. Default: `FALSE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchELU`

Methods

Public methods:

- `PipeOpTorchELU$new()`
- `PipeOpTorchELU$clone()`

`PipeOpTorchELU$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchELU$new(id = "nn_elu", param_vals = list())
```

Arguments:

`id` (`character(1)`)
: Identifier of the resulting object.

`param_vals` (`list()`)
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchELU$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchELU$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("elu")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_flatten` *Flattens a Tensor*

Description

Flattens a contiguous range of dimensions into one, leaving the others untouched. With the default `start_dim = 2`, every dimension after the batch dimension is collapsed into a single feature dimension.

nn_module

Calls `torch::nn_flatten()` when trained.

Parameters

`start_dim :: integer(1)`
At which dimension to start flattening. Default is 2. `end_dim :: integer(1)`
At which dimension to stop flattening. Default is -1.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchFlatten`

Methods

Public methods:

- `PipeOpTorchFlatten$new()`
- `PipeOpTorchFlatten$clone()`

`PipeOpTorchFlatten$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchFlatten$new(id = "nn_flatten", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchFlatten$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchFlatten$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("flatten")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_fn` *Custom Function*

Description

Applies a user-supplied function to a tensor.

Parameters

By default, these are inferred as all but the first arguments of the function `fn`. It is also possible to specify these more explicitly via the `param_set` constructor argument.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchFn`

Methods

Public methods:

- `PipeOpTorchFn$new()`
- `PipeOpTorchFn$clone()`

`PipeOpTorchFn$new()`: Creates a new instance of this `R6` class.

Usage:

```
PipeOpTorchFn$new(
  fn,
  id = "nn_fn",
  param_vals = list(),
  param_set = NULL,
  shapes_out = NULL
)
```

Arguments:

`fn` (`function`)
: The function to be applied. Takes a `torch` tensor as first argument and returns a `torch` tensor.

`id` (`character(1)`)
: Identifier of the resulting object.

`param_vals` (`list()`)
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`param_set` (`ParamSet` or `NULL`)
: A ParamSet wrapping the arguments to `fn`. If omitted, then the ParamSet for this PipeOp will be inferred from the function signature.

`shapes_out` (`function` or `NULL`)
: A function that computes the output shapes of the `fn`. See PipeOpTorch's `.shapes_out()` method for details on the parameters, and PipeOpTaskPreprocTorch for details on how the shapes are inferred when this parameter is `NULL`.

`PipeOpTorchFn$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchFn$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
custom_fn =  function(x, a) x / a
obj = nn("fn", fn = custom_fn, a = 2)
obj$param_set

graph = po("torch_ingress_ltnsr") %>>% obj

task = tsk("lazy_iris")$filter(1)
tnsr = materialize(task$data()$x)[[1]]

md_trained = graph$train(task)
trained = md_trained[[1]]$graph$train(tnsr)

trained[[1]]

custom_fn(tnsr, a = 2)
```

`mlr_pipeops_nn_ft_cls` *CLS Token for FT-Transformer*

Description

Concatenates a CLS token to the input as the last feature. The input shape is expected to be `(batch, n_features, d_token)` and the output shape is `(batch, n_features + 1, d_token)`.

This is used in the `LearnerTorchFTTransformer`.

nn_module

Calls `nn_ft_cls()` when trained.

Parameters

Id	Type	Default	Levels
initialization	character	-	uniform, normal

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchFTCLS`

Methods

Public methods:

- `PipeOpTorchFTCLS$new()`
- `PipeOpTorchFTCLS$clone()`

`PipeOpTorchFTCLS$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchFTCLS$new(id = "nn_ft_cls", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchFTCLS$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchFTCLS$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("ft_cls")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_ft_transformer_block`

Single Transformer Block for the FT-Transformer

Description

A transformer block consisting of a multi-head self-attention mechanism followed by a feed-forward network.

This is used in `LearnerTorchFTTransformer`.

nn_module

Calls `nn_ft_transformer_block()` when trained.

Parameters

The parameters of `nn_ft_transformer_block()`.

Id	Type	Default	Levels	Range
attention_n_heads	integer	-		$[1, \infty)$
attention_dropout	numeric	-		$[0, 1]$
attention_initialization	character	-	kaiming, xavier	-
attention_normalization	untyped	-		-
ffn_d_hidden	integer	-		$[1, \infty)$
ffn_d_hidden_multiplier	numeric	-		$[0, \infty)$
ffn_dropout	numeric	-		$[0, 1]$
ffn_activation	untyped	-		-
ffn_normalization	untyped	-		-
residual_dropout	numeric	-		$[0, 1]$
prenormalization	logical	-	TRUE, FALSE	-
is_first_layer	logical	-	TRUE, FALSE	-

query_idx	untyped	-		-
attention_bias	logical	-	TRUE, FALSE	-
ffn_bias_first	logical	-	TRUE, FALSE	-
ffn_bias_second	logical	-	TRUE, FALSE	-

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchFTTransformerBlock`

Methods

Public methods:

- `PipeOpTorchFTTransformerBlock$new()`
- `PipeOpTorchFTTransformerBlock$clone()`

`PipeOpTorchFTTransformerBlock$new()`: Create a new instance of this R6 class.

Usage:

```
PipeOpTorchFTTransformerBlock$new(
  id = "nn_ft_transformer_block",
  param_vals = list()
)
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchFTTransformerBlock$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchFTTransformerBlock$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("ft_transformer_block")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_geglu` *GeGLU Activation Function*

Description

Gaussian Error Linear Unit Gated Linear Unit (GeGLU) activation function, see `nn_geglu` for details.

Parameters

No parameters.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchGeGLU`

Methods

Public methods:

- `PipeOpTorchGeGLU$new()`
- `PipeOpTorchGeGLU$clone()`

`PipeOpTorchGeGLU$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchGeGLU$new(id = "nn_geglu", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchGeGLU$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchGeGLU$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("geglu")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_gelu` *GELU Activation Function*

Description

Gelu

nn_module

Calls `torch::nn_gelu()` when trained.

Parameters

- `approximate` :: `character(1)`
  Whether to use an approximation algorithm. Default is `"none"`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchGELU`

Methods

Public methods:

- `PipeOpTorchGELU$new()`
- `PipeOpTorchGELU$clone()`

`PipeOpTorchGELU$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchGELU$new(id = "nn_gelu", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchGELU$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchGELU$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("gelu")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_glu` *GLU Activation Function*

Description

The gated linear unit. Computes:

nn_module

Calls `torch::nn_glu()` when trained.

Parameters

- `dim` :: `integer(1)`
  Dimension on which to split the input. Default: -1

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchGLU

Methods

Public methods:

- PipeOpTorchGLU$new()
- PipeOpTorchGLU$clone()

PipeOpTorchGLU$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchGLU$new(id = "nn_glu", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchGLU$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchGLU$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("glu")
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_group_norm

Group Normalization

Description

Applies Group Normalization for last certain number of dimensions.

nn_module

Calls torch::nn_group_norm() when trained. The parameter num_channels is inferred as the second dimension of the input shape.

Parameters

- num_groups :: integer(1)
  The number of groups to separate the channels into. Must divide the number of channels. Setting it to 1 normalizes over all channels at once (layer normalization), setting it to the number of channels normalizes each channel on its own (instance normalization).
- eps :: numeric(1)
  A value added to the denominator for numerical stability. Default: 1e-5.
- affine :: logical(1)
  Whether to learn per-channel affine parameters initialized to 1 for weights and to 0 for biases. Default: TRUE.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchGroupNorm

Methods

Public methods:

- PipeOpTorchGroupNorm$new()
- PipeOpTorchGroupNorm$clone()

PipeOpTorchGroupNorm$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchGroupNorm$new(id = "nn_group_norm", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchGroupNorm$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchGroupNorm$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("group_norm", num_groups = 1)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_gru` *Gated Recurrent Unit*

Description

For each element in the input sequence, each layer computes the following function:

nn_module

Calls `torch::nn_gru()` when trained, where the parameter `input_size` is inferred as the last dimension of the input tensor and `batch_first` is always `TRUE`, see section *Tensor Layout*.

State

The state is the value calculated by the public method `$shapes_out()`.

Tensor Layout

Input and output are `(batch, sequence, feature)`, i.e. the `batch_first` layout of `torch::nn_rnn()`, which is fixed and not a hyperparameter. `torch` defaults to `(sequence, batch, feature)`, but the first dimension of every shape has to be the batch dimension here.

The final hidden state is `(layers * directions, batch, hidden_size)` in `torch`, even in the batch-first layout. It is transposed to `(batch, layers * directions, hidden_size)` here, so that the batch dimension comes first for it as well.

Parameters

The parameters are those of `nn("rnn")` without `nonlinearity`, whose activation functions an LSTM cell fixes.

Input and Output Channels

There is one input channel `"input"`, the sequence to run over.

The module has two outputs: `"output"`, the output sequence of shape `(batch, sequence, hidden_size * directions)`, and `"h_n"`, the final hidden state of shape `(batch, num_layers * directions, hidden_size)`. Only `"output"` is an output channel by default. Set `$outputs` to also (or only) get the final state, e.g. `outputs = c("output", "h_n")`.

For an explanation see `PipeOpTorch`.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchRecurrent -> PipeOpTorchGRU

Methods

Public methods:

- PipeOpTorchGRU$new()
- PipeOpTorchGRU$clone()

PipeOpTorchGRU$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchGRU$new(id = "nn_gru", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchGRU$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchGRU$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("gru", hidden_size = 10)
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_hardshrink

Hard Shrink Activation Function

Description

Applies the hard shrinkage function element-wise

nn_module

Calls torch::nn_hardshrink() when trained.

Parameters

- `lambd` :: `numeric(1)`
  The lambda value for the Hardshrink formulation. Default 0.5.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchHardShrink`

Methods

Public methods:

- `PipeOpTorchHardShrink$new()`
- `PipeOpTorchHardShrink$clone()`

`PipeOpTorchHardShrink$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchHardShrink$new(id = "nn_hardshrink", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchHardShrink$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchHardShrink$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("hardshrink")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_hardsigmoid` *Hard Sigmoid Activation Function*

Description

Applies the element-wise function Hardsigmoid$(x) = \frac{ReLU6(x+3)}{6}$

nn_module

Calls `torch::nn_hardsigmoid()` when trained.

Parameters

No parameters.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchHardSigmoid`

Methods

Public methods:

- `PipeOpTorchHardSigmoid$new()`
- `PipeOpTorchHardSigmoid$clone()`

`PipeOpTorchHardSigmoid$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchHardSigmoid$new(id = "nn_hardsigmoid", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchHardSigmoid$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchHardSigmoid$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("hardsigmoid")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_hardtanh`

Hard Tanh Activation Function

Description

Applies the HardTanh function element-wise.

nn_module

Calls `torch::nn_hardtanh()` when trained.

Parameters

- `min_val` :: `numeric(1)`
  Minimum value of the linear region range. Default: -1.
- `max_val` :: `numeric(1)`
  Maximum value of the linear region range. Default: 1.
- `inplace` :: `logical(1)`
  Can optionally do the operation in-place. Default: `FALSE`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchHardTanh`

Methods

Public methods:

- PipeOpTorchHardTanh$new()
- PipeOpTorchHardTanh$clone()

`PipeOpTorchHardTanh$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchHardTanh$new(id = "nn_hardtanh", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchHardTanh$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchHardTanh$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("hardtanh")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_head` *Output Head*

Description

Output head for classification and regresssion.

Details

When the method `$shapes_out()` does not have access to the task, it returns `c(NA, NA)`. When this PipeOp is trained however, the model descriptor has the correct output shape.

nn_module

Calls `torch::nn_linear()` with the input features inferred from the input shape and the output features from the task, via `output_dim_for()`. For

- binary classification, the output dimension is 1.
- multiclass classification, the output dimension is the number of classes.
- regression, the output dimension is 1.

Parameters

- `bias` :: `logical(1)`
  Whether to use a bias. Default is `TRUE`.

Supporting Other Task Types

The output dimension is not hard-coded here: `PipeOpTorchHead` asks the generic `output_dim_for()` how many output neurons the task needs, and **mlr3torch** implements methods for `TaskClassif` and `TaskRegr`. You can add support to your custom task type by implementing a method for your class.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchHead`

Methods

Public methods:

- `PipeOpTorchHead$new()`
- `PipeOpTorchHead$clone()`

`PipeOpTorchHead$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchHead$new(id = "nn_head", param_vals = list())
```

Arguments:

`id` (`character(1)`)
: Identifier of the resulting object.

`param_vals` (`list()`)
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchHead$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchHead$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("head")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_identity` *Identity Layer*

Description

A placeholder identity operator that is argument-insensitive.

nn_module

Calls `torch::nn_identity()` when trained, which passes the input unchanged to the output.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchIdentity`

Methods

Public methods:

- `PipeOpTorchIdentity$new()`
- `PipeOpTorchIdentity$clone()`

`PipeOpTorchIdentity$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchIdentity$new(id = "nn_identity", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchIdentity$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchIdentity$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("identity")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_layer_norm`

Layer Normalization

Description

Applies Layer Normalization for last certain number of dimensions.

nn_module

Calls `torch::nn_layer_norm()` when trained. The parameter `normalized_shape` is inferred as the dimensions of the last `dims` dimensions of the input shape.

Parameters

- `dims` :: `integer(1)`
  The number of dimensions over which will be normalized (starting from the last dimension).
- `elementwise_affine` :: `logical(1)`
  Whether to learn affine-linear parameters initialized to `1` for weights and to `0` for biases. The default is `TRUE`.
- `eps` :: `numeric(1)`
  A value added to the denominator for numerical stability.

State

The state is the value calculated by the public method `$shapes_out()`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchLayerNorm`

Methods

Public methods:

- `PipeOpTorchLayerNorm$new()`
- `PipeOpTorchLayerNorm$clone()`

`PipeOpTorchLayerNorm$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchLayerNorm$new(id = "nn_layer_norm", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchLayerNorm$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchLayerNorm$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("layer_norm", dims = 1)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_leaky_relu` *Leaky ReLU Activation Function*

Description

Applies element-wise, $LeakyReLU(x) = max(0, x) + negative_slope * min(0, x)$

nn_module

Calls `torch::nn_leaky_relu()` when trained.

Parameters

- `negative_slope` :: `numeric(1)`
  Controls the angle of the negative slope. Default: 1e-2.
- `inplace` :: `logical(1)`
  Can optionally do the operation in-place. Default: `FALSE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchLeakyReLU`

Methods

Public methods:

- `PipeOpTorchLeakyReLU$new()`
- `PipeOpTorchLeakyReLU$clone()`

`PipeOpTorchLeakyReLU$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchLeakyReLU$new(id = "nn_leaky_relu", param_vals = list())
```

Arguments:

`id` (`character(1)`)
Identifier of the resulting object.

param_vals (list())
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchLeakyReLU$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchLeakyReLU$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("leaky_relu")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_linear` *Linear Layer*

Description

Applies a linear transformation to the incoming data: $y = xA^T + b$.

nn_module

Calls `torch::nn_linear()` when trained where the parameter `in_features` is inferred as the second to last dimension of the input tensor.

Parameters

- `out_features` :: `integer(1)`
  The output features of the linear layer.
- `bias` :: `logical(1)`
  Whether to use a bias. Default is `TRUE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchLinear

Methods

Public methods:

- PipeOpTorchLinear$new()
- PipeOpTorchLinear$clone()

PipeOpTorchLinear$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchLinear$new(id = "nn_linear", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchLinear$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchLinear$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("linear", out_features = 10)
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_log_sigmoid

Log Sigmoid Activation Function

Description

Applies element-wise $LogSigmoid(x_i) = log(\frac{1}{1+exp(-x_i)})$

nn_module

Calls torch::nn_log_sigmoid() when trained.

Parameters

No parameters.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchLogSigmoid

Methods

Public methods:

- PipeOpTorchLogSigmoid$new()
- PipeOpTorchLogSigmoid$clone()

PipeOpTorchLogSigmoid$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchLogSigmoid$new(id = "nn_log_sigmoid", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchLogSigmoid$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchLogSigmoid$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("log_sigmoid")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_log_softmax` *Log Softmax*

Description

Applies a softmax followed by a logarithm.

nn_module

Calls `torch::nn_log_softmax()` when trained.

Parameters

- `dim` :: `integer(1)`
  A dimension along which LogSoftmax will be computed.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchSoftmaxDim` -> `PipeOpTorchLogSoftmax`

Methods

Public methods:

- `PipeOpTorchLogSoftmax$new()`
- `PipeOpTorchLogSoftmax$clone()`

`PipeOpTorchLogSoftmax$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchLogSoftmax$new(id = "nn_log_softmax", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchLogSoftmax$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchLogSoftmax$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("log_softmax")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_lstm` *Long Short-Term Memory*

Description

For each element in the input sequence, each layer computes the following function:

nn_module

Calls `torch::nn_lstm()` when trained, where the parameter `input_size` is inferred as the last dimension of the input tensor and `batch_first` is always `TRUE`, see section *Tensor Layout*.

Parameters

The parameters are those of `nn("rnn")` without `nonlinearity`, whose activation functions an LSTM cell fixes.

Input and Output Channels

There is one input channel `"input"`, the sequence to run over.

The module has three outputs: `"output"`, the output sequence of shape `(batch, sequence, hidden_size * directions)`, and `"h_n"` and `"c_n"`, the final hidden state and the final cell state, both of shape `(batch, num_layers * directions, hidd`
Only `"output"` is an output channel by default. Set `$outputs` to also (or only) get the final states, e.g. `outputs = c("output", "h_n", "c_n")`.

For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Tensor Layout

Input and output are `(batch, sequence, feature)`, i.e. the `batch_first` layout of `torch::nn_rnn()`, which is fixed and not a hyperparameter. `torch` defaults to `(sequence, batch, feature)`, but the first dimension of every shape has to be the batch dimension here.

The final hidden state is `(layers * directions, batch, hidden_size)` in `torch`, even in the batch-first layout. It is transposed to `(batch, layers * directions, hidden_size)` here, so that the batch dimension comes first for it as well.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchRecurrent` -> `PipeOpTorchLSTM`

Methods

Public methods:

- `PipeOpTorchLSTM$new()`
- `PipeOpTorchLSTM$clone()`

`PipeOpTorchLSTM$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchLSTM$new(id = "nn_lstm", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchLSTM$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchLSTM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("lstm", hidden_size = 10)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_max_pool1d` *1D Max Pooling*

Description

Applies a 1D max pooling over an input signal composed of several input planes.

nn_module

Calls `torch::nn_max_pool1d()` during training.

Parameters

- `kernel_size` :: `integer()`
  The size of the window. Can be single number or a vector.
- `stride` :: `integer()`
  The stride of the window. Can be a single number or a vector. Default: `kernel_size`
- `padding` :: `integer()`
  Implicit zero paddings on both sides of the input. Can be a single number or a tuple (padW,). Default: 0
- `dilation` :: `integer()`
  Controls the spacing between the kernel points; also known as the a trous algorithm. Default: 1
- `ceil_mode` :: `logical(1)`
  When True, will use ceil instead of floor to compute the output shape. Default: `FALSE`

Input and Output Channels

There is one input channel `"input"`. The module has two outputs, `"output"` and `"indices"`, of which only `"output"` is an output channel by default. Set `$outputs` to also (or only) get the indices, e.g. `outputs = c("output", "indices")`, which are only computed when they are among `$outputs`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchMaxPool` -> `PipeOpTorchMaxPool1D`

Methods

Public methods:

- `PipeOpTorchMaxPool1D$new()`
- `PipeOpTorchMaxPool1D$clone()`

`PipeOpTorchMaxPool1D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchMaxPool1D$new(id = "nn_max_pool1d", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchMaxPool1D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchMaxPool1D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("max_pool1d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_max_pool2d` *2D Max Pooling*

Description

Applies a 2D max pooling over an input signal composed of several input planes.

nn_module

Calls `torch::nn_max_pool2d()` during training.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `kernel_size` :: `integer()`
  The size of the window. Can be single number or a vector.
- `stride` :: `integer()`
  The stride of the window. Can be a single number or a vector. Default: `kernel_size`
- `padding` :: `integer()`
  Implicit zero paddings on both sides of the input. Can be a single number or a tuple (padW,). Default: 0
- `dilation` :: `integer()`
  Controls the spacing between the kernel points; also known as the a trous algorithm. Default: 1
- `ceil_mode` :: `logical(1)`
  When True, will use ceil instead of floor to compute the output shape. Default: `FALSE`

Input and Output Channels

There is one input channel `"input"`. The module has two outputs, `"output"` and `"indices"`, of which only `"output"` is an output channel by default. Set `$outputs` to also (or only) get the indices, e.g. `outputs = c("output", "indices")`, which are only computed when they are among `$outputs`. For an explanation see `PipeOpTorch`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchMaxPool` -> `PipeOpTorchMaxPool2D`

Methods

Public methods:

- `PipeOpTorchMaxPool2D$new()`
- `PipeOpTorchMaxPool2D$clone()`

`PipeOpTorchMaxPool2D$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchMaxPool2D$new(id = "nn_max_pool2d", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchMaxPool2D$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchMaxPool2D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("max_pool2d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_max_pool3d` *3D Max Pooling*

Description

Applies a 3D max pooling over an input signal composed of several input planes.

nn_module

Calls `torch::nn_max_pool3d()` during training.

State

The state is the value calculated by the public method `$shapes_out()`.

Parameters

- `kernel_size` :: `integer()`
  The size of the window. Can be single number or a vector.
- `stride` :: `integer()`
  The stride of the window. Can be a single number or a vector. Default: `kernel_size`
- `padding` :: `integer()`
  Implicit zero paddings on both sides of the input. Can be a single number or a tuple (padW,). Default: 0
- `dilation` :: `integer()`
  Controls the spacing between the kernel points; also known as the a trous algorithm. Default: 1
- `ceil_mode` :: `logical(1)`
  When True, will use ceil instead of floor to compute the output shape. Default: `FALSE`

Input and Output Channels

There is one input channel `"input"`. The module has two outputs, `"output"` and `"indices"`, of which only `"output"` is an output channel by default. Set `$outputs` to also (or only) get the indices, e.g. `outputs = c("output", "indices")`, which are only computed when they are among `$outputs`. For an explanation see `PipeOpTorch`.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchMaxPool -> PipeOpTorchMaxPool3D

Methods

Public methods:

- PipeOpTorchMaxPool3D$new()
- PipeOpTorchMaxPool3D$clone()

PipeOpTorchMaxPool3D$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchMaxPool3D$new(id = "nn_max_pool3d", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchMaxPool3D$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchMaxPool3D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("max_pool3d")
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_merge *Merge Operation*

Description

Base class for merge operations such as addition (PipeOpTorchMergeSum), multiplication (PipeOpTorchMergeProd or concatenation (PipeOpTorchMergeCat).

Parameters

See the respective child class.

State

The state is the value calculated by the public method `shapes_out()`.

Input and Output Channels

`PipeOpTorchMerges` has either a *vararg* input channel if the constructor argument `innum` is not set, or input channels `"input1"`, ..., `"input<innum>"`. There is one output channel `"output"`. For an explanation see `PipeOpTorch`.

Internals

Per default, the `private$.shapes_out()` method outputs the shape that the inputs broadcast to. There are two things to be aware of:

1. Broadcasting is generalized to unknown (`NA`) sizes: per dimension a known size that is not 1 wins, and the result is only unknown when every input is either unknown or 1, because an unknown size may turn out to be greater than 1 and would then determine the size.
2. Tensors are expected to have the same number of dimensions, i.e. missing dimensions are not filled with 1s. The reason is that the first dimension should be the batch dimension. This private method can be overwritten by `PipeOpTorch`s inheriting from this class.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchMerge`

Methods

Public methods:

- `PipeOpTorchMerge$new()`
- `PipeOpTorchMerge$clone()`

`PipeOpTorchMerge$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchMerge$new(
  id,
  module_generator,
  param_set = ps(),
  innum = 0,
  param_vals = list()
)
```

Arguments:

`id` (`character(1)`)
: Identifier of the resulting object.

`module_generator` (`nn_module_generator`)
: The torch module generator.

`param_set` (`ParamSet`)
: The parameter set.

`innum (integer(1))`
: The number of inputs. Default is 0 which means there is one *vararg* input channel.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchMerge$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchMerge$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`mlr_pipeops_nn_merge_cat`

Merge by Concatenation

Description

Concatenates multiple tensors on a given dimension. No broadcasting rules are applied here, you must reshape the tensors before to have the same shape.

nn_module

Calls `nn_merge_cat()` when trained.

Parameters

- `dim` :: `integer(1)`
  The dimension along which to concatenate the tensors. The default is -1, i.e., the last dimension.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

`PipeOpTorchMerges` has either a *vararg* input channel if the constructor argument `innum` is not set, or input channels `"input1"`, ..., `"input<innum>"`. There is one output channel `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchMerge` -> `PipeOpTorchMergeCat`

Methods

Public methods:

- `PipeOpTorchMergeCat$new()`
- `PipeOpTorchMergeCat$speak()`
- `PipeOpTorchMergeCat$clone()`

`PipeOpTorchMergeCat$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchMergeCat$new(id = "nn_merge_cat", innum = 0, param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`innum (integer(1))`
: The number of inputs. Default is 0 which means there is one *vararg* input channel.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchMergeCat$speak()`: What does the cat say?

Usage:

```
PipeOpTorchMergeCat$speak()
```

`PipeOpTorchMergeCat$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchMergeCat$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("merge_cat")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_merge_prod` *Merge by Product*

Description

Calculates the product of all input tensors.

nn_module

Calls `nn_merge_prod()` when trained.

Parameters

No parameters.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

`PipeOpTorchMerges` has either a *vararg* input channel if the constructor argument `innum` is not set, or input channels `"input1"`, ..., `"input<innum>"`. There is one output channel `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchMerge` -> `PipeOpTorchMergeProd`

Methods

Public methods:

- `PipeOpTorchMergeProd$new()`
- `PipeOpTorchMergeProd$clone()`

`PipeOpTorchMergeProd$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchMergeProd$new(id = "nn_merge_prod", innum = 0, param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`innum (integer(1))`
: The number of inputs. Default is 0 which means there is one *vararg* input channel.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchMergeProd$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchMergeProd$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("merge_prod")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_merge_sum` *Merge by Summation*

Description

Calculates the sum of all input tensors.

nn_module

Calls `nn_merge_sum()` when trained.

Parameters

No parameters.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

`PipeOpTorchMerges` has either a *vararg* input channel if the constructor argument `innum` is not set, or input channels `"input1"`, ..., `"input<innum>"`. There is one output channel `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchMerge -> PipeOpTorchMergeSum

Methods

Public methods:

- PipeOpTorchMergeSum$new()
- PipeOpTorchMergeSum$clone()

PipeOpTorchMergeSum$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchMergeSum$new(id = "nn_merge_sum", innum = 0, param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

innum (integer(1))
: The number of inputs. Default is 0 which means there is one *vararg* input channel.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchMergeSum$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchMergeSum$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("merge_sum")
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_multihead_attention
Multi-Head Attention

Description

Multi-head attention as described in *Attention Is All You Need*.

This is a thin wrapper around torch::nn_multihead_attention() that makes it usable as a building block of a Graph of tensor operations, where both self-attention and cross-attention can be expressed, see section *Input and Output Channels*.

Tensor Layout

All inputs and outputs are `(batch, sequence, feature)`, i.e. the `batch_first` layout of `torch::nn_multihead_attentio` which is fixed and not a hyperparameter. `torch` defaults to `(sequence, batch, feature)`, but the first dimension of every shape has to be the batch dimension here.

nn_module

Calls `torch::nn_multihead_attention()` when trained, where the parameters `embed_dim`, `kdim` and `vdim` are inferred as the last dimension of the query, key and value tensors respectively, and `batch_first` is always `TRUE`, see section *Tensor Layout*.

Parameters

- `num_heads` :: `integer(1)`
  Number of parallel attention heads. The embedding dimension must be divisible by `num_heads`.
- `dropout` :: `numeric(1)`
  Dropout probability on the attention weights. Default is `0`.
- `bias` :: `logical(1)`
  Whether to add a bias to the input and output projections. Default is `TRUE`.
- `add_bias_kv` :: `logical(1)`
  Whether to add a bias to the key and value sequences at dimension 1. Default is `FALSE`.
- `add_zero_attn` :: `logical(1)`
  Whether to add a new batch of zeros to the key and value sequences at dimension 1. Default is `FALSE`.
- `avg_weights` :: `logical(1)`
  Whether the returned attention weights are averaged over the attention heads. Default is `TRUE`. Only has an effect when `"weights"` is among `$outputs`.

Note that `embed_dim`, `kdim` and `vdim` are *not* parameters, as they are inferred from the shapes of the input tensors, and that `batch_first` is *not* a parameter either, as it is fixed to `TRUE`, see section *Tensor Layout*.

Input and Output Channels

The number of input channels is determined by the construction argument `mode`:

- `mode = "self"` (default): one input channel `"input"`, which is used as query, key and value, i.e. the `PipeOp` performs *self-attention*.
- `mode = "cross"`: input channels `"query"` and `"key_value"`, i.e. the `PipeOp` performs *cross-attention*, where the second input is used as both key and value.
- `mode = "general"`: input channels `"query"`, `"key"` and `"value"`, i.e. the `PipeOp` performs *cross-attention* with separate key and value inputs.

The module has two outputs, `"output"`, containing the attention output, and `"weights"`, containing the attention weights, of which only `"output"` is an output channel by default. Set `$outputs` to also (or only) get the weights, e.g. `outputs = c("output", "weights")`, which are only computed when they are among `$outputs`.

For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchMultiheadAttention`

Methods

Public methods:

- `PipeOpTorchMultiheadAttention$new()`
- `PipeOpTorchMultiheadAttention$clone()`

`PipeOpTorchMultiheadAttention$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchMultiheadAttention$new(
  id = "nn_multihead_attention",
  mode = "self",
  param_vals = list()
)
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`mode (character(1))`
: The attention mode, which determines the input channels. One of `"self"`, `"cross"` or `"general"`. This is a *construction* argument (and not a hyperparameter), because it determines the structure of the `Graph`. The default is `"self"`, which means that the `PipeOp` performs self-attention. See section *Input and Output Channels* for more information.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchMultiheadAttention$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchMultiheadAttention$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A, Kaiser Ł, Polosukhin I (2017). "Attention is all you need." *Advances in neural information processing systems*, **30**.

Examples

```
# Construct the PipeOp
pipeop = nn("multihead_attention", num_heads = 4)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_prelu` *PReLU Activation Function*

Description

Applies element-wise the function $PReLU(x) = max(0, x) + weight * min(0, x)$ where weight is a learnable parameter.

nn_module

Calls `torch::nn_prelu()` when trained.

Parameters

- `num_parameters` :: `integer(1)`
  Number of a parameters to learn. Although it takes an integer as input, only two values are legitimate: 1, or the number of channels of the input. Default: 1.
- `init` :: `numeric(1)`
  The initial value of a. Default: 0.25.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchPReLU`

Methods

Public methods:

- PipeOpTorchPReLU$new()
- PipeOpTorchPReLU$clone()

`PipeOpTorchPReLU$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchPReLU$new(id = "nn_prelu", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchPReLU$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchPReLU$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("prelu")
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_recurrent

Recurrent Layer

Description

Base class for the recurrent layers PipeOpTorchRNN, PipeOpTorchLSTM and PipeOpTorchGRU.

Parameters

See the respective child class.

State

The state is the value calculated by the public method `shapes_out()`.

Input and Output Channels

One input channel "input", and the output channels described in the child classes. For an explanation see PipeOpTorch.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchRecurrent

Methods

Public methods:

- PipeOpTorchRecurrent$new()
- PipeOpTorchRecurrent$clone()

PipeOpTorchRecurrent$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchRecurrent$new(id, type, param_set, param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

type (character(1))
: Which recurrent layer to build, one of "rnn", "lstm" or "gru".

param_set (ParamSet)
: The parameter set.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchRecurrent$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchRecurrent$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other PipeOps: mlr_pipeops_torch_model

`mlr_pipeops_nn_reglu` *ReGLU Activation Function*

Description

Rectified Gated Linear Unit (ReGLU) activation function. See `nn_reglu` for details.

Parameters

No parameters.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchReGLU`

Methods

Public methods:

- `PipeOpTorchReGLU$new()`
- `PipeOpTorchReGLU$clone()`

`PipeOpTorchReGLU$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchReGLU$new(id = "nn_reglu", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchReGLU$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchReGLU$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("reglu")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_relu` *ReLU Activation Function*

Description

Applies the rectified linear unit function element-wise.

nn_module

Calls `torch::nn_relu()` when trained.

Parameters

- `inplace` :: `logical(1)`
  Whether to do the operation in-place. Default: `FALSE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchReLU`

Methods

Public methods:

- `PipeOpTorchReLU$new()`
- `PipeOpTorchReLU$clone()`

`PipeOpTorchReLU$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchReLU$new(id = "nn_relu", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchReLU$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchReLU$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("relu")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_relu6` *ReLU6 Activation Function*

Description

Applies the element-wise function $ReLU6(x) = min(max(0, x), 6)$.

nn_module

Calls `torch::nn_relu6()` when trained.

Parameters

- `inplace` :: `logical(1)`
  Whether to do the operation in-place. Default: `FALSE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchReLU6

Methods

Public methods:

- PipeOpTorchReLU6$new()
- PipeOpTorchReLU6$clone()

PipeOpTorchReLU6$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchReLU6$new(id = "nn_relu6", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchReLU6$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchReLU6$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("relu6")
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_reshape

Reshape a Tensor

Description

Reshape a tensor to the given shape.

nn_module

Calls nn_reshape() when trained. This internally calls torch::torch_reshape() with the given shape.

Parameters

- shape :: integer() | function()
  The desired output shape. One dimension at most can be -1, which torch infers from the number of elements. The first dimension is the batch dimension.

  It can also be a function(shape) that is called on the input shape and returns the output shape, e.g. \(shape) c(shape[1:2], 10). This expresses a reshape for inputs whose sizes are not known in advance, because the function is called again on the shape of the actual tensor when the network runs. Note that it is called with a shape that can contain NAs during shape inference. This is e.g. useful when there are multiple unknown dimensions such as (batch, sequence, ...).

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchReshape

Methods

Public methods:

- PipeOpTorchReshape$new()
- PipeOpTorchReshape$clone()

PipeOpTorchReshape$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchReshape$new(id = "nn_reshape", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchReshape$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchReshape$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("reshape")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_rnn` *Simple Recurrent Layer*

Description

Applies a multi-layer Elman RNN with $\tanh$ or ReLU non-linearity to an input sequence.

Tensor Layout

Input and output are `(batch, sequence, feature)`, i.e. the `batch_first` layout of `torch::nn_rnn()`, which is fixed and not a hyperparameter. `torch` defaults to `(sequence, batch, feature)`, but the first dimension of every shape has to be the batch dimension here.

The final hidden state is `(layers * directions, batch, hidden_size)` in `torch`, even in the batch-first layout. It is transposed to `(batch, layers * directions, hidden_size)` here, so that the batch dimension comes first for it as well.

nn_module

Calls `torch::nn_rnn()` when trained, where the parameter `input_size` is inferred as the last dimension of the input tensor and `batch_first` is always `TRUE`, see section *Tensor Layout*.

Parameters

- `hidden_size` :: `integer(1)`
  The number of features of the hidden state.
- `num_layers` :: `integer(1)`
  The number of stacked recurrent layers. Default is `1`.
- `bias` :: `logical(1)`
  Whether to use bias weights. Default is `TRUE`.
- `dropout` :: `numeric(1)`
  Dropout probability on the output of each layer but the last. Only has an effect when `num_layers` is greater than `1`. Default is `0`.
- `bidirectional` :: `logical(1)`
  Whether to run the sequence in both directions and concatenate the results, which doubles the size of the feature dimension of the output. Default is `FALSE`.
- `nonlinearity` :: `character(1)`
  The activation function, either `"tanh"` or `"relu"`. Default is `"tanh"`.

Note that `input_size` is *not* a parameter, as it is inferred from the shape of the input tensor, and that `batch_first` is *not* a parameter either, as it is fixed to `TRUE`, see section *Tensor Layout*.

Input and Output Channels

There is one input channel "input", the sequence to run over.

The module has two outputs: "output", the output sequence of shape (batch, sequence, hidden_size * directions), and "h_n", the final hidden state of shape (batch, num_layers * directions, hidden_size). Only "output" is an output channel by default. Set $outputs to also (or only) get the final state, e.g. outputs = c("output", "h_n").

For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchRecurrent -> PipeOpTorchRNN

Methods

Public methods:

- PipeOpTorchRNN$new()
- PipeOpTorchRNN$clone()

PipeOpTorchRNN$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchRNN$new(id = "nn_rnn", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchRNN$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchRNN$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("rnn", hidden_size = 10)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_rrelu` *RReLU Activation Function*

Description

Randomized leaky ReLU.

nn_module

Calls `torch::nn_rrelu()` when trained.

Parameters

- `lower`:: `numeric(1)`
  Lower bound of the uniform distribution. Default: 1/8.
- `upper`:: `numeric(1)`
  Upper bound of the uniform distribution. Default: 1/3.
- `inplace` :: `logical(1)`
  Whether to do the operation in-place. Default: `FALSE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchRReLU`

Methods

Public methods:

- `PipeOpTorchRReLU$new()`
- `PipeOpTorchRReLU$clone()`

`PipeOpTorchRReLU$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchRReLU$new(id = "nn_rrelu", param_vals = list())
```

Arguments:

`id` (`character(1)`)
Identifier of the resulting object.

param_vals (list())
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchRReLU$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchRReLU$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("rrelu")
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_selu *SELU Activation Function*

Description

Applies element-wise,

$$SELU(x) = scale * (max(0, x) + min(0, \alpha * (exp(x) - 1)))$$

, with $\alpha = 1.67326324235437728481704299167 17$ and $scale = 1.05070098735548049341933498529 46$.

nn_module

Calls `torch::nn_selu()` when trained.

Parameters

- inplace :: logical(1)
  Whether to do the operation in-place. Default: FALSE.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchSELU

Methods

Public methods:

- PipeOpTorchSELU$new()
- PipeOpTorchSELU$clone()

PipeOpTorchSELU$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchSELU$new(id = "nn_selu", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchSELU$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchSELU$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("selu")
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_sigmoid

Sigmoid Activation Function

Description

Applies element-wise $Sigmoid(x_i) = \frac{1}{1+exp(-x_i)}$

nn_module

Calls torch::nn_sigmoid() when trained.

Parameters

No parameters.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchSigmoid

Methods

Public methods:

- PipeOpTorchSigmoid$new()
- PipeOpTorchSigmoid$clone()

PipeOpTorchSigmoid$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchSigmoid$new(id = "nn_sigmoid", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchSigmoid$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchSigmoid$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("sigmoid")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_silu` *SiLU Activation Function*

Description

Applies the Sigmoid Linear Unit (SiLU) function, element-wise. See `nn_silu()` for more information.

nn_module

Calls `torch::nn_silu()` when trained.

Parameters

- `inplace` :: `logical(1)`
  Whether to do the operation in-place. Default: `FALSE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchSiLU`

Methods

Public methods:

- `PipeOpTorchSiLU$new()`
- `PipeOpTorchSiLU$clone()`

`PipeOpTorchSiLU$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchSiLU$new(id = "nn_silu", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchSiLU$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchSiLU$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("silu")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_softmax` *Softmax*

Description

Applies a softmax function.

nn_module

Calls `torch::nn_softmax()` when trained.

Parameters

- `dim` :: `integer(1)`
  A dimension along which Softmax will be computed (so every slice along dim will sum to 1).

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchSoftmaxDim` -> `PipeOpTorchSoftmax`

Methods

Public methods:

- PipeOpTorchSoftmax$new()
- PipeOpTorchSoftmax$clone()

`PipeOpTorchSoftmax$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchSoftmax$new(id = "nn_softmax", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchSoftmax$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchSoftmax$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("softmax")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_softmax2d`
2D Softmax

Description

Applies SoftMax over features to each spatial location. When given an image of `Channels x Height x Width`, it will apply `Softmax` to each location $(Channels, h_i, w_j)$

nn_module

Calls `torch::nn_softmax2d()` when trained. The softmax is applied over the feature dimension (dimension 2), so that every spatial position sums to 1.

Parameters

No parameters.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchSoftmax2D

Methods

Public methods:

- PipeOpTorchSoftmax2D$new()
- PipeOpTorchSoftmax2D$clone()

PipeOpTorchSoftmax2D$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchSoftmax2D$new(id = "nn_softmax2d", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchSoftmax2D$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchSoftmax2D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("softmax2d")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_softmin` *Softmin*

Description

Applies a softmin function.

nn_module

Calls `torch::nn_softmin()` when trained.

Parameters

- `dim` :: `integer(1)`
  A dimension along which Softmin will be computed (so every slice along dim will sum to 1).

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchSoftmaxDim` -> `PipeOpTorchSoftmin`

Methods

Public methods:

- `PipeOpTorchSoftmin$new()`
- `PipeOpTorchSoftmin$clone()`

`PipeOpTorchSoftmin$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchSoftmin$new(id = "nn_softmin", param_vals = list())
```

Arguments:

`id` (`character(1)`)
Identifier of the resulting object.

`param_vals` (`list()`)
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchSoftmin$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchSoftmin$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("softmin")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_softplus`

SoftPlus Activation Function

Description

Applies element-wise, the function $Softplus(x) = 1/\beta * log(1 + exp(\beta * x))$.

nn_module

Calls `torch::nn_softplus()` when trained.

Parameters

- `beta` :: `numeric(1)`
  The beta value for the Softplus formulation. Default: 1
- `threshold` :: `numeric(1)`
  Values above this revert to a linear function. Default: 20

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchSoftPlus`

Methods

Public methods:

- `PipeOpTorchSoftPlus$new()`
- `PipeOpTorchSoftPlus$clone()`

`PipeOpTorchSoftPlus$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchSoftPlus$new(id = "nn_softplus", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchSoftPlus$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchSoftPlus$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("softplus")
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_softshrink

Soft Shrink Activation Function

Description

Applies the soft shrinkage function elementwise

nn_module

Calls `torch::nn_softshrink()` when trained.

Parameters

- `lambd` :: `numeric(1)`
  The lambda (must be no less than zero) value for the Softshrink formulation. Default: 0.5

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchSoftShrink

Methods

Public methods:

- PipeOpTorchSoftShrink$new()
- PipeOpTorchSoftShrink$clone()

PipeOpTorchSoftShrink$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchSoftShrink$new(id = "nn_softshrink", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchSoftShrink$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchSoftShrink$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("softshrink")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_softsign` *SoftSign Activation Function*

Description

Applies element-wise, the function $SoftSign(x) = x/(1 + |x|$

nn_module

Calls `torch::nn_softsign()` when trained.

Parameters

No parameters.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchSoftSign`

Methods

Public methods:

- `PipeOpTorchSoftSign$new()`
- `PipeOpTorchSoftSign$clone()`

`PipeOpTorchSoftSign$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchSoftSign$new(id = "nn_softsign", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchSoftSign$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchSoftSign$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("softsign")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_squeeze` *Squeeze a Tensor*

Description

Squeezes a tensor by calling `torch::torch_squeeze()` with the given dimension `dim`.

nn_module

Calls `nn_squeeze()` when trained.

Parameters

- `dim` :: `integer()`
  The dimensions to squeeze. Negative values are interpreted downwards from the last dimension. A dimension whose size is not known to be 1 is kept, because it may be larger at runtime.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchSqueeze`

Methods

Public methods:

- `PipeOpTorchSqueeze$new()`
- `PipeOpTorchSqueeze$clone()`

`PipeOpTorchSqueeze$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchSqueeze$new(id = "nn_squeeze", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchSqueeze$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchSqueeze$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("squeeze")
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_tanh *Tanh Activation Function*

Description

Applies the element-wise function:

nn_module

Calls `torch::nn_tanh()` when trained.

Parameters

No parameters.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchTanh

Methods

Public methods:

- PipeOpTorchTanh$new()
- PipeOpTorchTanh$clone()

PipeOpTorchTanh$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchTanh$new(id = "nn_tanh", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchTanh$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchTanh$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("tanh")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_tanhshrink` *Tanh Shrink Activation Function*

Description

Applies element-wise, $Tanhshrink(x) = x - Tanh(x)$

nn_module

Calls `torch::nn_tanhshrink()` when trained.

Parameters

No parameters.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchTanhShrink`

Methods

Public methods:

- `PipeOpTorchTanhShrink$new()`
- `PipeOpTorchTanhShrink$clone()`

`PipeOpTorchTanhShrink$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchTanhShrink$new(id = "nn_tanhshrink", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchTanhShrink$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchTanhShrink$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("tanhshrink")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_threshold`

Threshold Activation Function

Description

Thresholds each element of the input Tensor.

nn_module

Calls `torch::nn_threshold()` when trained.

Parameters

- `threshold` :: `numeric(1)`
  The value to threshold at.
- `value` :: `numeric(1)`
  The value to replace with.
- `inplace` :: `logical(1)`
  Can optionally do the operation in-place. Default: `FALSE`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchThreshold`

Methods

Public methods:

- `PipeOpTorchThreshold$new()`
- `PipeOpTorchThreshold$clone()`

`PipeOpTorchThreshold$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchThreshold$new(id = "nn_threshold", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchThreshold$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchThreshold$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("threshold", threshold = 1, value = 2)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_tokenizer_categ`
Categorical Tokenizer

Description

Tokenizes categorical features into a dense embedding. For an input of shape `(batch, n_features)` the output shape is `(batch, n_features, d_token)`.

nn_module

Calls `nn_tokenizer_categ()` when trained where the parameter `cardinalities` is inferred. The output shape is `(batch, n_features, d_token)`.

Parameters

- `d_token` :: `integer(1)`
  The dimension of the embedding.
- `bias` :: `logical(1)`
  Whether to use a bias. Is initialized to `TRUE`.
- `initialization` :: `character(1)`
  The initialization method for the embedding weights. Possible values are `"uniform"` (default) and `"normal"`.
- `cardinalities` :: `integer()`
  The number of categories for each feature. Only needs to be provided when working with `lazy_tensor` inputs.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchTokenizerCateg`

Methods

Public methods:

- `PipeOpTorchTokenizerCateg$new()`
- `PipeOpTorchTokenizerCateg$clone()`

`PipeOpTorchTokenizerCateg$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchTokenizerCateg$new(id = "nn_tokenizer_categ", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchTokenizerCateg$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchTokenizerCateg$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("tokenizer_categ", d_token = 10)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_tokenizer_num`
Numeric Tokenizer

Description

Tokenizes numeric features into a dense embedding. For an input of shape `(batch, n_features)` the output shape is `(batch, n_features, d_token)`.

nn_module

Calls `nn_tokenizer_num()` when trained where the parameter `n_features` is inferred. The output shape is `(batch, n_features, d_token)`.

Parameters

- `d_token` :: `integer(1)`
  The dimension of the embedding.
- `bias` :: `logical(1)`
  Whether to use a bias. Is initialized to `TRUE`.
- `initialization` :: `character(1)`
  The initialization method for the embedding weights. Possible values are `"uniform"` (default) and `"normal"`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchTokenizerNum`

Methods

Public methods:

- `PipeOpTorchTokenizerNum$new()`
- `PipeOpTorchTokenizerNum$clone()`

`PipeOpTorchTokenizerNum$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchTokenizerNum$new(id = "nn_tokenizer_num", param_vals = list())
```

Arguments:

`id` (`character(1)`)
Identifier of the resulting object.

`param_vals` (`list()`)
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchTokenizerNum$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchTokenizerNum$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("tokenizer_num", d_token = 10)
pipeop
# The available parameters
pipeop$param_set
```

---

mlr_pipeops_nn_transformer_encoder_layer
Transformer Encoder Layer

Description

A single transformer encoder layer as described in *Attention Is All You Need*, consisting of a multi-head self-attention block and a position-wise feed-forward network, each wrapped in a residual connection and layer normalization.

This is a thin wrapper around `torch::nn_transformer_encoder_layer()` that makes it usable as a building block of a `Graph` of tensor operations.

Tensor Layout

Input and output are `(batch, sequence, feature)`, i.e. the `batch_first` layout of `torch::nn_transformer_encoder_la` which is fixed and not a hyperparameter. `torch` defaults to `(sequence, batch, feature)`, but the first dimension of every shape has to be the batch dimension here.

The layer preserves the shape of its input, so the output shape is the input shape. Only the feature dimension has to be known, the sequence length may be `NA`.

nn_module

Calls `torch::nn_transformer_encoder_layer()` when trained, where the parameter `d_model` is inferred as the last dimension of the input tensor and `batch_first` is always `TRUE`, see section *Tensor Layout*.

Parameters

- `nhead` :: `integer(1)`
  Number of parallel attention heads. The feature dimension must be divisible by `nhead`.
- `dim_feedforward` :: `integer(1)`
  The hidden dimension of the feed-forward network. Default is `2048`.
- `dropout` :: `numeric(1)`
  Dropout probability, applied to both the attention weights and the feed-forward network. Default is `0.1`.
- `activation` :: `character(1)` | `function`
  The activation function of the feed-forward network, either `"relu"`, `"gelu"`, or a function mapping a `torch_tensor` to a `torch_tensor`. Default is `"relu"`.
- `layer_norm_eps` :: `numeric(1)`
  A value added to the denominator of the layer normalizations for numerical stability. Default is `1e-5`.
- `norm_first` :: `logical(1)`
  Whether to apply layer normalization before the attention and feed-forward sublayers (pre-norm) instead of after them (post-norm). Default is `FALSE`.
- `bias` :: `logical(1)`
  Whether the linear layers use a bias and the layer normalizations learn affine parameters. Default is `TRUE`.
- `is_causal` :: `logical(1)`
  Whether to apply a causal mask, i.e. whether a position may only attend to the positions up to and including itself. This cannot be combined with the `"src_mask"` input channel. Default is `FALSE`.

Note that `d_model` is *not* a parameter, as it is inferred from the shape of the input tensor, and that `batch_first` is *not* a parameter either, as it is fixed to `TRUE`, see section *Tensor Layout*.

Input and Output Channels

There is always one input channel `"input"`, which is the sequence the layer attends over, and one output channel `"output"` of the same shape.

The construction arguments `src_mask` and `src_key_padding_mask` each add a further input channel of that name, which is passed to the corresponding argument of the module's `forward()` method. They are construction arguments and not hyperparameters, because they determine the structure of the `Graph`:

- `"src_mask"`: an additive or boolean mask over the sequence, of shape `(sequence, sequence)` or `(batch * nhead, sequence, sequence)`. Positions that are `TRUE` are *not* attended to.
- `"src_key_padding_mask"`: a mask of shape `(batch, sequence)` marking the padding positions of each observation, which are `TRUE`.

Note that a `(sequence, sequence)` `"src_mask"` has no batch dimension, unlike every other tensor in a network, so its first dimension is the sequence length rather than the batch size.

For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchTransformerEncoderLayer`

Methods

Public methods:

- `PipeOpTorchTransformerEncoderLayer$new()`
- `PipeOpTorchTransformerEncoderLayer$clone()`

`PipeOpTorchTransformerEncoderLayer$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchTransformerEncoderLayer$new(
  id = "nn_transformer_encoder_layer",
  src_mask = FALSE,
  src_key_padding_mask = FALSE,
  param_vals = list()
)
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`src_mask (logical(1))`
: Whether the attention mask is provided as an additional input channel `"src_mask"`. This is a *construction* argument (and not a hyperparameter), because it determines the structure of the `Graph`. The default is `FALSE`, i.e. the layer attends over the full sequence. See section *Input and Output Channels* for more information.

`src_key_padding_mask (logical(1))`
: Whether the padding mask is provided as an additional input channel `"src_key_padding_mask"`. This is a *construction* argument (and not a hyperparameter), because it determines the structure of the `Graph`. The default is `FALSE`, i.e. no position is treated as padding. See section *Input and Output Channels* for more information.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchTransformerEncoderLayer$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchTransformerEncoderLayer$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A, Kaiser Ł, Polosukhin I (2017). "Attention is all you need." *Advances in neural information processing systems*, **30**.

Examples

```
# Construct the PipeOp
pipeop = nn("transformer_encoder_layer", nhead = 4)
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_unflatten` *Unflattens a Tensor*

Description

Unflattens a tensor dim expanding it to a desired shape. For use with [nn_sequential.

nn_module

Calls `torch::nn_unflatten()` when trained.

Parameters

- `dim` :: `integer(1)`
  The dimension to unflatten. Negative values are interpreted downwards from the last dimension.
- `unflattened_size` :: `integer()`
  The sizes that replace that dimension. One of them at most can be `-1`, which torch infers from the size of the dimension being unflattened.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method $shapes_out().

Super classes

mlr3pipelines::PipeOp -> PipeOpTorch -> PipeOpTorchUnflatten

Methods

Public methods:

- PipeOpTorchUnflatten$new()
- PipeOpTorchUnflatten$clone()

PipeOpTorchUnflatten$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchUnflatten$new(id = "nn_unflatten", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchUnflatten$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchUnflatten$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("unflatten", dim = 2, unflattened_size = c(2, 2))
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_nn_unsqueeze` *Unsqueeze a Tensor*

Description

Unsqueezes a tensor by calling `torch::torch_unsqueeze()` with the given dimension `dim`.

nn_module

Calls `nn_unsqueeze()` when trained. This internally calls `torch::torch_unsqueeze()`.

Parameters

- `dim` :: `integer(1)`
  The dimension which to unsqueeze. Negative values are interpreted downwards from the last dimension.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is the value calculated by the public method `$shapes_out()`.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorch` -> `PipeOpTorchUnsqueeze`

Methods

Public methods:

- `PipeOpTorchUnsqueeze$new()`
- `PipeOpTorchUnsqueeze$clone()`

`PipeOpTorchUnsqueeze$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchUnsqueeze$new(id = "nn_unsqueeze", param_vals = list())
```

Arguments:

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchUnsqueeze$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchUnsqueeze$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Construct the PipeOp
pipeop = nn("unsqueeze")
pipeop
# The available parameters
pipeop$param_set
```

`mlr_pipeops_preproc_torch`

Base Class for Lazy Tensor Preprocessing

Description

This `PipeOp` can be used to preprocess (one or more) `lazy_tensor` columns contained in an `mlr3::Task`. The preprocessing function is specified as construction argument `fn` and additional arguments to this function can be defined through the `PipeOp`'s parameter set. The preprocessing is done per column, i.e. the number of lazy tensor output columns is equal to the number of lazy tensor input columns.

To create custom preprocessing `PipeOps` you can use `pipeop_preproc_torch`.

Inheriting

In addition to specifying the construction arguments, you can overwrite the private `.shapes_out()` method. If you don't overwrite it, the output shapes are assumed to be unknown (`NULL`).

- `.shapes_out(shapes_in, param_vals, task)`
  (`list()`, `list()`, `Task` or `NULL`) -> `list()`
  This private method calculates the output shapes of the lazy tensor columns that are created from applying the preprocessing function with the provided parameter values (`param_vals`). The `task` is very rarely needed, but if it is it should be checked that it is not `NULL`.

  This private method only has the responsibility to calculate the output shapes for one input column, i.e. the input `shapes_in` can be assumed to have exactly one shape vector for which it must calculate the output shapes and return it as a `list()` of length 1. It can also be assumed that the shape is not `NULL` (i.e. unknown). Any dimension of that shape can be `NA`, i.e. unknown, not only the first (batch) one, so it must handle `NA`s and assert those dimensions it actually needs to be known, see `assert_known_dims()` and the "Shape Inference" section of `PipeOpTorch`.

Input and Output Channels

See PipeOpTaskPreproc.

State

In addition to state elements from PipeOpTaskPreprocSimple, the state also contains the $param_vals that were set during training.

Parameters

In addition to the parameters inherited from PipeOpTaskPreproc as well as those specified during construction as the argument param_set there are the following parameters:

- stages :: character(1)
  The stages during which to apply the preprocessing. Can be one of "train", "predict" or "both". The initial value of this parameter is set to "train" when the PipeOp's id starts with "augment_" and to "both" otherwise. Note that the preprocessing that is applied during $predict() uses the parameters that were set during $train() and not those that are set when performing the prediction.

Internals

During $train() / $predict(), a PipeOpModule with one input and one output channel is created. The pipeop applies the function fn to the input tensor while additionally passing the parameter values (minus stages and affect_columns) to fn. The preprocessing graph of the lazy tensor columns is shallowly cloned and the PipeOpModule is added. This is done to avoid modifying user input and means that identical PipeOpModules can be part of different preprocessing graphs. This is only possible, because the created PipeOpModule is stateless.

At a later point in the graph, preprocessing graphs will be merged if possible to avoid unnecessary computation. This is best illustrated by example: One lazy tensor column's preprocessing graph is A -> B. Then, two branches are created B -> C and B -> D, creating two preprocessing graphs A -> B -> C and A -> B -> D. When loading the data, we want to run the preprocessing only once, i.e. we don't want to run the A -> B part twice. For this reason, task_dataset() will try to merge graphs and cache results from graphs. However, only graphs using the same dataset can currently be merged.

Also, the shapes created during $train() and $predict() might differ. To avoid the creation of graphs where the predict shapes are incompatible with the train shapes, the hypothetical predict shapes are already calculated during $train() (this is why the parameters that are set during train are also used during predict) and the PipeOpTorchModel will check the train and predict shapes for compatibility before starting the training.

Otherwise, this mechanism is very similar to the ModelDescriptor construct.

Super classes

mlr3pipelines::PipeOp -> mlr3pipelines::PipeOpTaskPreproc -> PipeOpTaskPreprocTorch

Active bindings

`fn` The preprocessing function.

`rowwise` Whether the preprocessing is applied rowwise.

Methods

Public methods:

- `PipeOpTaskPreprocTorch$new()`
- `PipeOpTaskPreprocTorch$shapes_out()`
- `PipeOpTaskPreprocTorch$clone()`

`PipeOpTaskPreprocTorch$new()`: Creates a new instance of this `R6` class.

Usage:

```
PipeOpTaskPreprocTorch$new(
  fn,
  id = "preproc_torch",
  param_vals = list(),
  param_set = ps(),
  packages = character(0),
  rowwise = FALSE,
  stages_init = NULL,
  tags = NULL
)
```

Arguments:

`fn` (`function` or `character(2)`)
The preprocessing function. Must not modify its input in-place. If it is a `character(2)`, the first element should be the namespace and the second element the name. When the preprocessing function is applied to the tensor, the tensor will be passed by position as the first argument. If the `param_set` is inferred (left as `NULL`) it is assumed that the first argument is the `torch_tensor`.

`id` (`character(1)`)
The id for of the new object.

`param_vals` (named `list()`)
Parameter values to be set after construction.

`param_set` (`ParamSet`)
In case the function `fn` takes additional parameters besides a `torch_tensor` they can be specified as parameters. None of the parameters can have the `"predict"` tag. All tags should include `"train"`.

`packages` (`character()`)
The packages the preprocessing function depends on.

`rowwise` (`logical(1)`)
Whether the preprocessing function is applied rowwise (and then concatenated by row) or directly to the whole tensor. In the first case there is no batch dimension.

`stages_init` (`character(1)`)
Initial value for the `stages` parameter.

tags (character())
: Tags for the pipeop.

`PipeOpTaskPreprocTorch$shapes_out()`: Calculates the output shapes that would result in applying the preprocessing to one or more lazy tensor columns with the provided shape. Names are ignored and only order matters. It uses the parameter values that are currently set.

Usage:

```
PipeOpTaskPreprocTorch$shapes_out(shapes_in, stage = NULL, task = NULL)
```

Arguments:

shapes_in (list() of (integer() or NULL))
: The input shapes of the lazy tensors. NULL indicates that the shape is unknown. If it is not NULL, the first (batch) dimension must be NA; any of the other dimensions may be NA as well.

stage (character(1))
: The stage: either "train" or "predict".

task (Task or NULL)
: The task, which is very rarely needed.

Returns: list() of (integer() or NULL)

`PipeOpTaskPreprocTorch$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTaskPreprocTorch$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Creating a simple task
d = data.table(
  x1 = as_lazy_tensor(matrix(rnorm(10), ncol = 1)),
  x2 = as_lazy_tensor(matrix(rnorm(10), ncol = 1)),
  x3 = as_lazy_tensor(matrix(as.double(1:10), ncol = 1)),
  y = rnorm(10)
)

taskin = as_task_regr(d, target = "y")

# Creating a simple preprocessing pipeop
po_simple = po("preproc_torch",
  # get rid of environment baggage
  fn = mlr3misc::crate(function(x, a) x + a),
  param_set = paradox::ps(a = paradox::p_int(tags = c("train", "required")))
)

po_simple$param_set$set_values(
  a = 100,
  affect_columns = selector_name(c("x1", "x2")),
  stages = "both" # use during train and predict
)
```

```
taskout_train = po_simple$train(list(taskin))[[1L]]
materialize(taskout_train$data(cols = c("x1", "x2")), rbind = TRUE)

taskout_predict_noaug = po_simple$predict(list(taskin))[[1L]]
materialize(taskout_predict_noaug$data(cols = c("x1", "x2")), rbind = TRUE)

po_simple$param_set$set_values(
  stages = "train"
)

# transformation is not applied
taskout_predict_aug = po_simple$predict(list(taskin))[[1L]]
materialize(taskout_predict_aug$data(cols = c("x1", "x2")), rbind = TRUE)

# Creating a more complex preprocessing PipeOp
PipeOpPreprocTorchPoly = R6::R6Class("PipeOpPreprocTorchPoly",
 inherit = PipeOpTaskPreprocTorch,
 public = list(
   initialize = function(id = "preproc_poly", param_vals = list()) {
     param_set = paradox::ps(
       n_degree = paradox::p_int(lower = 1L, tags = c("train", "required"))
     )
     param_set$set_values(
       n_degree = 1L
     )
     fn = mlr3misc::crate(function(x, n_degree) {
       torch::torch_cat(
         lapply(seq_len(n_degree), function(d) torch::torch_pow(x, d)),
         dim = 2L
       )
     })

     super$initialize(
       fn = fn,
       id = id,
       packages = character(0),
       param_vals = param_vals,
       param_set = param_set,
       stages_init = "both"
     )
   }
 ),
 private = list(
   .shapes_out = function(shapes_in, param_vals, task) {
     # shapes_in is a list of length 1 containing the shapes
     checkmate::assert_true(length(shapes_in[[1L]]) == 2L)
     if (shapes_in[[1L]][2L] != 1L) {
       stop("Input shape must be (NA, 1)")
     }
     list(c(NA, param_vals$n_degree))
   }
 )
```

```
)

po_poly = PipeOpPreprocTorchPoly$new(
  param_vals = list(n_degree = 3L, affect_columns = selector_name("x3"))
)

po_poly$shapes_out(list(c(NA, 1L)), stage = "train")

taskout = po_poly$train(list(taskin))[[1L]]
materialize(taskout$data(cols = "x3"), rbind = TRUE)
```

`mlr_pipeops_torch` *Base Class for Torch Module Constructor Wrappers*

Description

`PipeOpTorch` is the base class for all `PipeOp`s that represent neural network layers in a `Graph`. If you want to create your own `PipeOpTorch`, check out the `pipeop_torch()` helper function and the article Writing your own PipeOpTorch. During **training**, it generates a `PipeOpModule` that wraps an `nn_module` and attaches it to the architecture, which is also represented as a `Graph` consisting mostly of `PipeOpModule`s an `PipeOpNOP`s.

The convenient way to construct such a `PipeOp` is the `nn()` helper, which prefixes the given key with `"nn_"` to look it up in the `mlr_pipeops` dictionary and uses the unprefixed key as the id of the resulting `PipeOp`: `nn("linear", out_features = 10)` is equivalent to `po("nn_linear", id = "linear", out_features = 10)`. Because ids must be unique within a `Graph`, repeated layers can be disambiguated with a `_<n>` suffix, e.g. `nn("linear_1")` and `nn("linear_2")`.

While the former `Graph` operates on `ModelDescriptor`s, the latter operates on tensors.

The relationship between a `PipeOpTorch` and a `PipeOpModule` is similar to the relationshop between a `nn_module_generator` (like `nn_linear`) and a `nn_module` (like the output of `nn_linear(...)`). A crucial difference is that the `PipeOpTorch` infers auxiliary parameters (like `in_features` for `nn_linear`) automatically from the intermediate tensor shapes that are being communicated through the `ModelDescriptor`.

During **prediction**, `PipeOpTorch` takes in a `Task` in each channel and outputs the same new `Task` resulting from their feature union in each channel. If there is only one input and output channel, the task is simply piped through.

Parameters

The `ParamSet` is specified by the child class inheriting from `PipeOpTorch`. Usually the parameters are the arguments of the wrapped `nn_module` minus the auxiliary parameter that can be automatically inferred from the shapes of the input tensors.

Inheriting

When inheriting from this class, one should overload either the `private$.shapes_out()` and the `private$.shape_dependent_params()` methods, or overload `private$.make_module()`.

- `.make_module(shapes_in, param_vals, task)`
  (`list()`, `list()`, `Task` or `NULL`) -> `nn_module`
  This private method is called to generate the `nn_module` that is passed as argument `module` to `PipeOpModule`. It must be overwritten, when no `module_generator` is provided. If left as is, it calls the provided `module_generator` with the arguments obtained by the private method `.shape_dependent_params()`.
- `.shapes_out(shapes_in, param_vals, task)`
  (`list()`, `list()`, `Task` or `NULL`) -> named `list()`
  This private method gets a list of `integer` vectors (`shapes_in`), the parameter values (`param_vals`), as well as an (optional) `Task`. The `shapes_in` list indicates the shape of input tensors that will be fed to the module's `$forward()` function. The list has one item per input tensor, typically only one. The function should return a list of shapes of tensors that are created by the module. The `shapes_in` are named after the input channels of the `PipeOp` and are in the same order. It returns one shape per output name given during construction, in that order, regardless of `$outputs`: the public `$shapes_out()` keeps those of `$outputs`. In case the output shapes depends on the task (as is the case for `PipeOpTorchHead`), the function should return valid output shapes (possibly containing `NA`s) whether or not the `task` argument is provided. Any dimension of `shapes_in` can be `NA`, i.e. unknown, so this method must not assume that a dimension it reads is known. It has to assert the dimensions it actually needs and propagate the `NA`s it can live with.

  The inference should generally be **permissive**. For example, applying a convolutional layer to an input of shape `c(NA, 3, NA, NA)` should assume that the last two dimensions are big enough for the given kernel size. This can sometimes lead to runtime errors but this is preferable over rejecting valid architectures.

  There are various assertion helpers for common input checks, such as `assert_known_dims()` and the "See also" links on its page. There are also `shape_helpers`, which provide the shape arithmetic (broadcasting, resolving negative dimension indices).
- `.module_outputs()`
  () -> `character()`
  The names of the outputs of the module that `.make_module()` creates, in the order in which it returns them. By default, these are all output names given during construction. A subclass whose module can leave out outputs that are not among `$outputs` (e.g. by not computing attention weights) can override this to save computation. It must include `$outputs`.
- `.additional_phash_input()`
  () -> `any`
  See `PipeOp`. A subclass that overrides it must include the result of `super$.additional_phash_input()`, which reflects `$outputs`.
- `.shape_dependent_params(shapes_in, param_vals, task)`
  (`list()`, `list()`, `Task` or `NULL`) -> named `list()`
  This private method has the same inputs as `.shapes_out`. If `.make_module()` is not overwritten, it constructs the arguments passed to `module_generator`. Usually this means that it must infer the auxiliary parameters that can be inferred from the input shapes and add it to the user-supplied parameter values (`param_vals`).

Input and Output Channels

During *training*, all inputs and outputs are of class `ModelDescriptor`. During *prediction*, all input and output channels are of class `Task`.

Shape Inference

A network is assembled without any data flowing through it, so **mlr3torch** tracks the shape of the tensors instead: it starts from the shape the ingress announces and hands each `PipeOp` the shapes of its inputs, which is how auxiliary parameters such as `in_features` of `nn_linear` are filled in automatically.

A shape is an `integer()` whose first entry is the batch dimension, e.g. `c(NA, 3, 32, 32)` for a batch of RGB images of 32 by 32 pixels. A dimension whose size is not known in advance is `NA`, and any dimension can be `NA`, not only the batch dimension: the sequence length of a transformer or the height and width of an image may equally well vary.

The public `$shapes_out()` method answers the same question for a single `PipeOp`, without building a network around it, which is the quickest way to see what an operator makes of a given input shape. It is also what reports the problem when an operator needs a dimension that is unknown, naming the `PipeOp` and the shape it was given.

Implementing a `PipeOp` that computes its own output shapes is described under `.shapes_out()` in the "Inheriting" section above.

State

The state is the value calculated by the public method `shapes_out()`.

Internals

During training, the `PipeOpTorch` creates a `PipeOpModule` for the given parameter specification and the input shapes from the incoming `ModelDescriptor`s using the private method `.make_module()`. The input shapes are provided by the slot `pointer_shape` of the incoming `ModelDescriptor`s. The channel names of this `PipeOpModule` are identical to the channel names of the generating `PipeOpTorch`.

A model descriptor union of all incoming `ModelDescriptor`s is then created. Note that this modifies the `graph` of the first `ModelDescriptor` **in place** for efficiency. The `PipeOpModule` is added to the `graph` slot of this union and the edges that connect the sending `PipeOpModule`s to the input channel of this `PipeOpModule` are addeded to the graph. This is possible because every incoming `ModelDescriptor` contains the information about the `id` and the `channel` name of the sending `PipeOp` in the slot `pointer`.

The new graph in the `model_descriptor_union` represents the current state of the neural network architecture. It is structurally similar to the subgraph that consists of all pipeops of class `PipeOpTorch` and `PipeOpTorchIngress` that are ancestors of this `PipeOpTorch`.

For the output, a shallow copy of the `ModelDescriptor` is created and the `pointer` and `pointer_shape` are updated accordingly. The shallow copy means that all `ModelDescriptor`s point to the same `Graph` which allows the graph to be modified by-reference in different parts of the code.

Super class

`mlr3pipelines::PipeOp` -> `PipeOpTorch`

Public fields

`module_generator` (`nn_module_generator` or `NULL`)
The module generator wrapped by this `PipeOpTorch`. If `NULL`, the private method `private$.make_module(shapes_in param_vals)` must be overwritten, see section 'Inheriting'. Do not change this after construction.

Active bindings

`outputs` (`character()`)
The output channels of the `PipeOp`, a subset of the outputs of the wrapped module. By default, there is one output channel for every output of the module, unless a subclass restricts them during construction, e.g. to leave out the indices of a max pooling.

Methods

Public methods:

- `PipeOpTorch$new()`
- `PipeOpTorch$shapes_out()`
- `PipeOpTorch$clone()`

`PipeOpTorch$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorch$new(
  id,
  module_generator,
  param_set = ps(),
  param_vals = list(),
  inname = "input",
  outname = "output",
  packages = "torch",
  tags = NULL
)
```

Arguments:

`id` (`character(1)`)
Identifier of the resulting object.

`module_generator` (`nn_module_generator`)
The torch module generator.

`param_set` (`ParamSet`)
The parameter set.

`param_vals` (`list()`)
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`inname` (`character()`)
: The names of the `PipeOp`'s input channels, `"input"` by default. These will be the input channels of the generated `PipeOpModule`. The tensors are passed to the wrapped `nn_module` by position, i.e. the order of the input channels determines which argument of the forward method they end up in. Unless the forward method has the argument `...`, naming the input channels after its arguments is therefore recommended, as it avoids any ambiguity.

`outname` (`character()`)
: The names of the output channels, `"output"` by default. These will be the ouput channels of the generated `PipeOpModule` and therefore also the names of the list returned by its `$train()`. In case there is more than one output channel, the `nn_module` that is constructed by this `PipeOp` during training must return a `list()` whose elements are in the order of the output channels. The field `$outputs` can restrict the output channels to a subset of these.

`packages` (`character()`)
: The R packages this object depends on.

`tags` (`character()`)
: The tags of the `PipeOp`. The tags `"torch"` is always added.

`PipeOpTorch$shapes_out()`: Calculates the output shapes for the given input shapes, parameters and task.

Usage:

```
PipeOpTorch$shapes_out(shapes_in, task = NULL)
```

Arguments:

`shapes_in` (`list()` of `integer()`)
: The input shapes, which must be in the same order as the input channel names of the `PipeOp`.

`task` (`Task` or `NULL`)
: The task, which is very rarely used (default is `NULL`). An exception is `PipeOpTorchHead`.

Returns: A named `list()` containing the output shapes. The names are the names of the output channels of the `PipeOp`, i.e. `$outputs`.

`PipeOpTorch$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorch$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Graph Network: `ModelDescriptor()`, `TorchIngressToken()`, `as_learner_torch()`, `mlr_learners_torch_model`
`mlr_pipeops_module`, `mlr_pipeops_torch_ingress`, `mlr_pipeops_torch_ingress_categ`, `mlr_pipeops_torch_ingre`
`mlr_pipeops_torch_ingress_num`, `model_descriptor_to_learner()`, `model_descriptor_to_module()`,
`model_descriptor_union()`, `nn_graph()`, `pipeop_torch()`

Examples

```
# A `PipeOpTorch` does not compute anything itself: it adds a module to the network that is
# being assembled, and reports the shape of the tensors that module will produce.
```

```
task = tsk("iris")

po_linear = po("nn_linear", out_features = 10)

# what the operator makes of an input shape, without building a network around it.
# `NA` is an unknown dimension -- the batch size here -- and any dimension may be unknown.
po_linear$shapes_out(list(c(NA, 4)))

# a dimension the operator does need is reported, naming the PipeOp and the shape it was given
try(po_linear$shapes_out(list(c(NA, NA))))

# the graph of `PipeOpTorch`s describes the network, `model_descriptor_to_module()` builds it.
# `in_features` of the linear layer was inferred from the task, `out_features` of the head from
# the number of classes.
graph = po("torch_ingress_num") %>>% po_linear %>>% po("nn_relu") %>>% po("nn_head")
md = graph$train(task)[[1L]]
network = model_descriptor_to_module(md)
network

x = torch_tensor(as.matrix(task$data(1:2, task$feature_names)))
with_no_grad(network(torch_ingress_num.input = x))


## What happens during training

# A `PipeOpTorch` operates on `ModelDescriptor`s, which we here build by hand: two networks that
# each start from one half of the iris task.
task1 = task$clone()$select(paste0("Sepal.", c("Length", "Width")))
task2 = task$clone()$select(paste0("Petal.", c("Length", "Width")))
ingress = gunion(list(po("torch_ingress_num_1"), po("torch_ingress_num_2")))
mds_in = ingress$train(list(task1, task2), single_input = FALSE)

mds_in[[1L]][c("graph", "task", "ingress", "pointer", "pointer_shape")]
mds_in[[2L]][c("graph", "task", "ingress", "pointer", "pointer_shape")]

# Training a `PipeOpTorch` on them creates the `PipeOpModule` that wraps the constructed
# `nn_module`, adds it to the network and connects it to both ingress operators.
po_merge = nn("merge_cat", innum = 2)
md_out = po_merge$train(list(input1 = mds_in[[1L]], input2 = mds_in[[2L]]))[[1L]]

# Note that, for efficiency, the graph of the first input is modified in-place.
identical(md_out$graph, mds_in[[1L]]$graph)
md_out$graph$edges

# The task is the feature union of the incoming tasks and `ingress` collects the ingress tokens
# of all incoming `ModelDescriptor`s.
md_out$task
md_out$ingress

# `pointer` and `pointer_shape` now refer to the output of the new module.
md_out$pointer
md_out$pointer_shape
```

```
# During prediction, no network is built: the `PipeOpTorch` receives a `Task` in each channel
# and outputs their feature union.
po_merge$predict(list(input1 = task1, input2 = task2))[[1L]]

# Writing an operator of your own is described in the article "Writing your own PipeOpTorch" on
# the package website and in the section 'Inheriting' above.
```

`mlr_pipeops_torch_callbacks` *Callback Configuration*

Description

Configures the callbacks of a deep learning model.

Parameters

The parameters are defined dynamically from the callbacks, where the id of the respective callbacks is the respective set id.

Input and Output Channels

There is one input channel `"input"` and one output channel `"output"`. During *training*, the channels are of class `ModelDescriptor`. During *prediction*, the channels are of class `Task`.

State

The state is the value calculated by the public method `shapes_out()`.

Internals

During training the callbacks are cloned and added to the `ModelDescriptor`.

Super class

`mlr3pipelines::PipeOp` -> `PipeOpTorchCallbacks`

Methods

Public methods:

- `PipeOpTorchCallbacks$new()`
- `PipeOpTorchCallbacks$clone()`

`PipeOpTorchCallbacks$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchCallbacks$new(
  callbacks = list(),
  id = "torch_callbacks",
  param_vals = list()
)
```

Arguments:

`callbacks` (`list` of `TorchCallbacks`)
: The callbacks (or something convertible via `as_torch_callbacks()`). Must have unique ids. All callbacks are cloned during construction.

`id` (`character(1)`)
: Identifier of the resulting object.

`param_vals` (`list()`)
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchCallbacks$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchCallbacks$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Model Configuration: `ModelDescriptor()`, `mlr_pipeops_torch_loss`, `mlr_pipeops_torch_optimizer`, `model_descriptor_union()`

Examples

```
po_cb = po("torch_callbacks", "checkpoint")
po_cb$param_set
mdin = po("torch_ingress_num")$train(list(tsk("iris")))
mdin[[1L]]$callbacks
mdout = po_cb$train(mdin)[[1L]]
mdout$callbacks
# Can be called again
po_cb1 = po("torch_callbacks", t_clbk("progress"))
mdout1 = po_cb1$train(list(mdout))[[1L]]
mdout1$callbacks
```

---

mlr_pipeops_torch_ingress

Entrypoint to Torch Network

Description

Use this as entry-point to mlr3torch-networks. Unless you are an advanced user, you should not need to use this directly but `PipeOpTorchIngressNumeric`, `PipeOpTorchIngressCategorical` or `PipeOpTorchIngressLazyTensor`.

Parameters

Defined by the construction argument `param_set`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is set to the input shape.

Internals

Creates an object of class `TorchIngressToken` for the given task. The purpuse of this is to store the information on how to construct the torch dataloader from the task for this entry point of the network.

Super class

`mlr3pipelines::PipeOp` -> `PipeOpTorchIngress`

Active bindings

`feature_types (character(1))`
The features types that can be consumed by this `PipeOpTorchIngress`.

Methods

Public methods:

- `PipeOpTorchIngress$new()`
- `PipeOpTorchIngress$clone()`

`PipeOpTorchIngress$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchIngress$new(
  id,
  param_set = ps(),
  param_vals = list(),
  packages = character(0),
  feature_types
)
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_set (ParamSet)`
The parameter set.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`packages (character())`
The R packages this object depends on.

`feature_types (character())`
The feature types. See `mlr_reflections$task_feature_types` for available values, Additionally, `"lazy_tensor"` is supported.

`PipeOpTorchIngress$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchIngress$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Graph Network: `ModelDescriptor()`, `TorchIngressToken()`, `as_learner_torch()`, `mlr_learners_torch_model`, `mlr_pipeops_module`, `mlr_pipeops_torch`, `mlr_pipeops_torch_ingress_categ`, `mlr_pipeops_torch_ingress_ltnsr`, `mlr_pipeops_torch_ingress_num`, `model_descriptor_to_learner()`, `model_descriptor_to_module()`, `model_descriptor_union()`, `nn_graph()`, `pipeop_torch()`

`mlr_pipeops_torch_ingress_categ`

Torch Entry Point for Categorical Features

Description

Ingress PipeOp that represents a categorical (`factor()`, `ordered()` and `logical()`) entry point to a torch network.

Parameters

- `select` :: `logical(1)`
  Whether `PipeOp` should selected the supported feature types. Otherwise it will err on receiving tasks with unsupported feature types.

Internals

Uses `batchgetter_categ()`.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is set to the input shape.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorchIngress -> PipeOpTorchIngressCategorical

Methods

Public methods:

- PipeOpTorchIngressCategorical$new()
- PipeOpTorchIngressCategorical$clone()

PipeOpTorchIngressCategorical$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchIngressCategorical$new(
  id = "torch_ingress_categ",
  param_vals = list()
)
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchIngressCategorical$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchIngressCategorical$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Graph Network: ModelDescriptor(), TorchIngressToken(), as_learner_torch(), mlr_learners_torch_model, mlr_pipeops_module, mlr_pipeops_torch, mlr_pipeops_torch_ingress, mlr_pipeops_torch_ingress_ltnsr, mlr_pipeops_torch_ingress_num, model_descriptor_to_learner(), model_descriptor_to_module(), model_descriptor_union(), nn_graph(), pipeop_torch()

Examples

```
graph = po("select", selector = selector_type("factor")) %>>%
  po("torch_ingress_categ")
task = tsk("german_credit")
# The output is a model descriptor
md = graph$train(task)[[1L]]
ingress = md$ingress[[1L]]
ingress$batchgetter(task$data(1, ingress$features(task)), "cpu")
```

`mlr_pipeops_torch_ingress_ltnsr`
Ingress for Lazy Tensor

Description

Ingress for a single `lazy_tensor` column.

Parameters

- `shape` :: `integer()` | `NULL` | `"infer"`
  The shape of the tensor, where the first dimension (batch) must be `NA`. When it is not specified, the lazy tensor input column needs to have a known shape. When it is set to `"infer"`, the shape is inferred from an example batch.

Internals

The returned batchgetter materializes the lazy tensor column to a tensor.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is set to the input shape.

Super classes

`mlr3pipelines::PipeOp` -> `PipeOpTorchIngress` -> `PipeOpTorchIngressLazyTensor`

Methods

Public methods:

- PipeOpTorchIngressLazyTensor$new()
- PipeOpTorchIngressLazyTensor$clone()

`PipeOpTorchIngressLazyTensor$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchIngressLazyTensor$new(
  id = "torch_ingress_ltnsr",
  param_vals = list()
)
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchIngressLazyTensor$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchIngressLazyTensor$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Graph Network: ModelDescriptor(), TorchIngressToken(), as_learner_torch(), mlr_learners_torch_model, mlr_pipeops_module, mlr_pipeops_torch, mlr_pipeops_torch_ingress, mlr_pipeops_torch_ingress_categ, mlr_pipeops_torch_ingress_num, model_descriptor_to_learner(), model_descriptor_to_module(), model_descriptor_union(), nn_graph(), pipeop_torch()

Examples

```
po_ingress = po("torch_ingress_ltnsr")
task = tsk("lazy_iris")

md = po_ingress$train(list(task))[[1L]]

ingress = md$ingress
x_batch = ingress[[1L]]$batchgetter(data = task$data(1, "x"), cache = NULL)
x_batch

# Now we try a lazy tensor with unknown shape, i.e. the shapes between the rows can differ

ds = dataset(
  initialize = function() self$x = list(torch_randn(3, 10, 10), torch_randn(3, 8, 8)),
```

```
  .getitem = function(i) list(x = self$x[[i]]),
  .length = function() 2)()

task_unknown = as_task_regr(data.table(
  x = as_lazy_tensor(ds, dataset_shapes = list(x = NULL)),
  y = rnorm(2)
), target = "y", id = "example2")

# this task (as it is) can NOT be processed by PipeOpTorchIngressLazyTensor
# It therefore needs to be preprocessed
po_resize = po("trafo_resize", size = c(6, 6))
task_unknown_resize = po_resize$train(list(task_unknown))[[1L]]

# printing the transformed column still shows unknown shapes,
# because the preprocessing pipeop cannot infer them,
# however we know that the shape is now (3, 10, 10) for all rows
task_unknown_resize$data(1:2, "x")
po_ingress$param_set$set_values(shape = c(NA, 3, 6, 6))

md2 = po_ingress$train(list(task_unknown_resize))[[1L]]

ingress2 = md2$ingress
x_batch2 = ingress2[[1L]]$batchgetter(
  data = task_unknown_resize$data(1:2, "x"),
  cache = NULL
)

x_batch2
```

`mlr_pipeops_torch_ingress_num`

Torch Entry Point for Numeric Features

Description

Ingress PipeOp that represents a numeric (`integer()` and `numeric()`) entry point to a torch network.

Internals

Uses `batchgetter_num()`.

Input and Output Channels

One input channel called `"input"` and one output channel called `"output"`. For an explanation see `PipeOpTorch`.

State

The state is set to the input shape.

Super classes

mlr3pipelines::PipeOp -> PipeOpTorchIngress -> PipeOpTorchIngressNumeric

Methods

Public methods:

- PipeOpTorchIngressNumeric$new()
- PipeOpTorchIngressNumeric$clone()

`PipeOpTorchIngressNumeric$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchIngressNumeric$new(id = "torch_ingress_num", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchIngressNumeric$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchIngressNumeric$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Graph Network: ModelDescriptor(), TorchIngressToken(), as_learner_torch(), mlr_learners_torch_model, mlr_pipeops_module, mlr_pipeops_torch, mlr_pipeops_torch_ingress, mlr_pipeops_torch_ingress_categ, mlr_pipeops_torch_ingress_ltnsr, model_descriptor_to_learner(), model_descriptor_to_module(), model_descriptor_union(), nn_graph(), pipeop_torch()

Examples

```
graph = po("select", selector = selector_type(c("numeric", "integer"))) %>>%
  po("torch_ingress_num")
task = tsk("german_credit")
# The output is a model descriptor
md = graph$train(task)[[1L]]
ingress = md$ingress[[1L]]
ingress$batchgetter(task$data(1:5, ingress$features(task)), "cpu")
```

mlr_pipeops_torch_loss *Loss Configuration*

Description

Configures the loss of a deep learning model.

Input and Output Channels

One input channel called "input" and one output channel called "output". For an explanation see PipeOpTorch.

State

The state is the value calculated by the public method shapes_out().

Parameters

The parameters are defined dynamically from the loss set during construction.

Internals

During training the loss is cloned and added to the ModelDescriptor.

Super class

mlr3pipelines::PipeOp -> PipeOpTorchLoss

Methods

Public methods:

- PipeOpTorchLoss$new()
- PipeOpTorchLoss$clone()

PipeOpTorchLoss$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchLoss$new(loss, id = "torch_loss", param_vals = list())
```

Arguments:

loss (TorchLoss or character(1) or nn_loss)
: The loss (or something convertible via as_torch_loss()).

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchLoss$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchLoss$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Model Configuration: `ModelDescriptor()`, `mlr_pipeops_torch_callbacks`, `mlr_pipeops_torch_optimizer`, `model_descriptor_union()`

Examples

```
po_loss = po("torch_loss", loss = t_loss("cross_entropy"))
po_loss$param_set
mdin = po("torch_ingress_num")$train(list(tsk("iris")))
mdin[[1L]]$loss
mdout = po_loss$train(mdin)[[1L]]
mdout$loss
```

---

mlr_pipeops_torch_model

PipeOp Torch Model

Description

Builds a Torch Learner from a `ModelDescriptor` and trains it with the given parameter specification.

This is `po("torch_model")`, which works on the general-purpose task type of `mlr3torch`, see `TaskTorch`. For classification and regression see `PipeOpTorchModelClassif` and `PipeOpTorchModelRegr` respectively.

Parameters

General:

The parameters of the optimizer, loss and callbacks, prefixed with `"opt."`, `"loss."` and `"cb.<callback id>."` respectively, as well as:

- `epochs` :: `integer(1)`
  The number of epochs.
- `device` :: `character(1)`
  The device. One of `"auto"`, `"cpu"`, or `"cuda"` or other values defined in `mlr_reflections$torch$devices`. A specific device can be selected by appending its index, e.g. `"cuda:1"` for the second GPU. The value is initialized to `"auto"`, which will select `"cuda"` if possible, then try `"mps"` and otherwise fall back to `"cpu"`.

- num_threads :: integer(1)
  The number of threads for intraop parallelization (if device is "cpu"). This value is initialized to 1. When resampling, benchmarking or tuning in parallel, each worker uses this many threads, so divide the available cores among the workers instead of setting this to the number of cores.
- num_interop_threads :: integer(1)
  The number of threads for interop parallelization (if device is "cpu"). Note that this can only be set **once** per session, so setting this for one learner also changes the behavior of other learners, and a later learner asking for a different value errors. NULL (default) uses whatever is set. In order to use different values for this parameter, use encapsulation to train the learners in separate R sessions.
- seed :: integer(1) or "random" or NULL
  The torch seed that is used during training and prediction. This value is initialized to "random", which means that a random seed will be sampled at the beginning of the training phase. This seed (either set or randomly sampled) is available via $model$seed after training and used during prediction. Note that by setting the seed during the training phase this will mean that by default (i.e. when seed is "random"), clones of the learner will use a different seed. If set to NULL, no seeding will be done. This parameter only seeds torch's random number generator, it does **not** seed R's. Anything that is drawn from R's RNG is therefore unaffected by it, so to make those parts reproducible you need to seed R's RNG as well, e.g. via set.seed().
- tensor_dataset :: logical(1) | "device"
  Whether to load all batches at once at the beginning of training and stack them. This is initialized to FALSE. If set to "device", the device of the tensors will be set to the value of device, which can avoid unnecessary moving of tensors between devices. When your dataset fits into memory this will make the loading of batches faster. Note that this should not be set for datasets that contain lazy_tensors with random data augmentation, as this augmentation will only be applied once at the beginning of training.
- cache_task_data :: logical(1)
  Whether to retrieve the data from the task once and to take the batches from it, instead of retrieving every batch from the task, which is slow. This is initialized to TRUE. Set it to FALSE if a copy of the task data does not fit into memory (lazy tensor columns only contain indices and are small).
- jit_trace :: logical(1)
  Whether to trace the network with torch::jit_trace() once at the start of training and then train the traced module instead of the original one. Not all learners support this. A traced network takes its inputs by position and ignores their names (mlverse/torch#1495). The learner passes the inputs in the right order, but when calling the traced $network of a trained learner with several inputs directly, they must be passed by position in the order given by attr(learner$network, "input_order").

Evaluation:

- measures_train :: Measure or list() of Measures
  Measures to be evaluated during training.
- measures_valid :: Measure or list() of Measures
  Measures to be evaluated during validation.

- `eval_freq` :: `integer(1)`
  How often the train / validation predictions are evaluated using `measures_train` / `measures_valid`. This is initialized to 1. Note that the final model is always evaluated.

Resuming:

- `resume` :: `character(1)` or `TRUE`
  Continues training from a checkpoint written by `t_clbk("checkpoint")`, either the folder it wrote to or `TRUE`, which takes that folder from the checkpoint callback of this learner. Note that `epochs` is the *total* number of epochs, i.e. it includes the epochs the checkpoint was already trained for: resuming a checkpoint from epoch 5 with `epochs = 8` trains 3 more epochs.

Early Stopping:

- `patience` :: `integer(1)`
  This activates early stopping using the validation scores. If the performance of a model does not improve for `patience` evaluation steps, training is ended. Note that this counts *evaluation steps*, not epochs: when `eval_freq` is greater than 1, `patience` evaluation steps correspond to `patience * eval_freq` epochs. Note that the final model is stored in the learner, not the best model, unless `restore_best_weights` is set to `TRUE`. This is initialized to `0`, which means no early stopping. The first entry from `measures_valid` is used as the metric. A missing (`NA`) score counts as no improvement, and every score improves on a missing one. If all scores are missing, the first evaluated epoch is reported as the best one. This also requires to specify the `$validate` field of the Learner, as well as `measures_valid`.
- `min_delta` :: `double(1)`
  The minimum improvement threshold for early stopping. Is initialized to 0.
- `restore_best_weights` :: `logical(1)`
  Whether to restore the weights of the best epoch when training ends, instead of keeping those of the last epoch that was trained. Like `min_delta`, this only has an effect when early stopping is active. Is initialized to `FALSE`, i.e. the network of the last epoch is stored. Setting this to `TRUE` makes the stored network the one of the epoch that `$internal_tuned_values` reports, and costs one additional copy of the network's parameters in memory.

Dataloader:

- `batch_size` :: `integer(1)`
  The batch size used by the training and prediction dataloader. It is required for training (unless a `batch_sampler` is provided, which already determines the batches) and it is required for prediction (unless `batch_size_predict` is set).
- `batch_size_predict` :: `integer(1)`
  The batch size used by the prediction dataloader (this includes the validation data during training). When set, it overrides `batch_size` for prediction. The batch size does not change the predictions, but smaller batches take longer and require less memory.
- `shuffle` :: `logical(1)`
  Whether to shuffle the instances in the dataset. This is initialized to `TRUE`, which differs from the default (`FALSE`). It is ignored when a `sampler` or `batch_sampler` is provided.
- `sampler` :: `torch::sampler`
  Object that defines how the dataloader draws samples, i.e. the order in which the observations are drawn. This must be the sampler *generator* (as returned by `torch::sampler()`), not an instance, as it is instantiated with the training dataset internally.

- batch_sampler :: torch::sampler
  Object that defines how the dataloader draws batches. As for sampler, this must be the generator. When it is provided, the parameters batch_size, shuffle and drop_last are ignored during training, because the batch sampler already determines the batches.
- num_workers :: integer(1)
  The number of workers for data loading (batches are loaded in parallel). The default is 0, which means that data will be loaded in the main process.
- collate_fn :: function
  How to merge a list of samples to form a batch.
- pin_memory :: logical(1)
  Whether the dataloader copies tensors into CUDA pinned memory before returning them.
- drop_last :: logical(1)
  Whether to drop the last training batch in each epoch during training. Default is FALSE. If it is TRUE, the batch size is capped at the number of training observations, so that not all of them are dropped. It is ignored when a batch_sampler is provided.
- timeout :: numeric(1)
  The timeout value for collecting a batch from workers. Negative values mean no timeout and the default is -1.
- worker_init_fn :: function(id)
  A function that receives the worker id (in [1, num_workers]) and is executed after seeding on the worker but before data loading.
- worker_globals :: list() | character()
  When loading data in parallel, this makes it possible to export globals to the workers. If this is a character vector, the objects in the global environment with those names are copied to the workers.
- worker_packages :: character()
  Which packages to load on the workers.

Also see torch::dataloader for more information.

Input and Output Channels

There is one input channel "input" that takes in ModelDescriptor during training and a Task of the specified task_type during prediction. The output is NULL during training and a Prediction of given task_type during prediction.

State

A trained LearnerTorchModel.

Internals

A LearnerTorchModel is created by calling model_descriptor_to_learner() on the provided ModelDescriptor that is received through the input channel. Then the parameters are set according to the parameters specified in PipeOpTorchModel and its $train() method is called on the Task stored in the ModelDescriptor.

Super classes

mlr3pipelines::PipeOp -> mlr3pipelines::PipeOpLearner -> PipeOpTorchModel

Methods

Public methods:

- PipeOpTorchModel$new()
- PipeOpTorchModel$clone()

PipeOpTorchModel$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchModel$new(
  task_type = "torch",
  id = "torch_model",
  param_vals = list(),
  target_batchgetter = NULL,
  predict_types = NULL
)
```

Arguments:

task_type (character(1))
: The task type of the model. Defaults to "torch".

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

target_batchgetter (function() or NULL)
: Converts the target columns of a batch into the target tensor y that the loss is applied to. Takes an argument data, a data.table with only the target columns, and optionally an argument x, the named list of feature tensors of the batch, which is what a target that is a function of the input needs, see get_target_batchgetter(). If NULL (default), it is taken from the task via get_target_batchgetter(), which the built-in task types provide, and a TaskTorch only if it has a default_target_batchgetter or no target at all, in which case the batches have no y element.

predict_types (character())
: The predict types. See mlr_reflections$learner_predict_types for available values.

PipeOpTorchModel$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchModel$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other PipeOps: mlr_pipeops_nn_recurrent

`mlr_pipeops_torch_model_classif` *PipeOp Torch Classifier*

Description

Builds a torch classifier and trains it.

Parameters

See `LearnerTorch`

Input and Output Channels

There is one input channel `"input"` that takes in `ModelDescriptor` during training and a `Task` of the specified `task_type` during prediction. The output is `NULL` during training and a `Prediction` of given `task_type` during prediction.

State

A trained `LearnerTorchModel`.

Internals

A `LearnerTorchModel` is created by calling `model_descriptor_to_learner()` on the provided `ModelDescriptor` that is received through the input channel. Then the parameters are set according to the parameters specified in `PipeOpTorchModel` and its `$train()` method is called on the `Task` stored in the `ModelDescriptor`.

Super classes

`mlr3pipelines::PipeOp` -> `mlr3pipelines::PipeOpLearner` -> `PipeOpTorchModel` -> `PipeOpTorchModelClassif`

Methods

Public methods:

- `PipeOpTorchModelClassif$new()`
- `PipeOpTorchModelClassif$clone()`

`PipeOpTorchModelClassif$new()`: Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchModelClassif$new(id = "torch_model_classif", param_vals = list())
```

Arguments:

`id (character(1))`
Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchModelClassif$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchModelClassif$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# simple logistic regression

# configure the model descriptor
md = as_graph(po("torch_ingress_num") %>>%
  nn("head") %>>%
  po("torch_loss", "cross_entropy") %>>%
  po("torch_optimizer", "adam"))$train(tsk("iris"))[[1L]]

print(md)

# build the learner from the model descriptor and train it
po_model = po("torch_model_classif", batch_size = 50, epochs = 1)
po_model$train(list(md))
po_model$state
```

---

mlr_pipeops_torch_model_regr

Torch Regression Model

Description

Builds a torch regression model and trains it.

Parameters

See `LearnerTorch`

Input and Output Channels

There is one input channel `"input"` that takes in `ModelDescriptor` during training and a `Task` of the specified `task_type` during prediction. The output is `NULL` during training and a `Prediction` of given `task_type` during prediction.

State

A trained `LearnerTorchModel`.

Internals

A LearnerTorchModel is created by calling model_descriptor_to_learner() on the provided ModelDescriptor that is received through the input channel. Then the parameters are set according to the parameters specified in PipeOpTorchModel and its $train() method is called on the Task stored in the ModelDescriptor.

Super classes

mlr3pipelines::PipeOp -> mlr3pipelines::PipeOpLearner -> PipeOpTorchModel -> PipeOpTorchModelRegr

Methods

Public methods:

- PipeOpTorchModelRegr$new()
- PipeOpTorchModelRegr$clone()

PipeOpTorchModelRegr$new(): Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchModelRegr$new(id = "torch_model_regr", param_vals = list())
```

Arguments:

id (character(1))
: Identifier of the resulting object.

param_vals (list())
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

PipeOpTorchModelRegr$clone(): The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchModelRegr$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# simple linear regression

# build the model descriptor
md = as_graph(po("torch_ingress_num") %>>%
  nn("head") %>>%
  po("torch_loss", "mse") %>>%
  po("torch_optimizer", "adam"))$train(tsk("mtcars"))[[1L]]

print(md)

# build the learner from the model descriptor and train it
po_model = po("torch_model_regr", batch_size = 20, epochs = 1)
po_model$train(list(md))
po_model$state
```

`mlr_pipeops_torch_optimizer` *Optimizer Configuration*

Description

Configures the optimizer of a deep learning model.

Parameters

The parameters are defined dynamically from the optimizer that is set during construction.

Input and Output Channels

There is one input channel `"input"` and one output channel `"output"`. During *training*, the channels are of class `ModelDescriptor`. During *prediction*, the channels are of class `Task`.

State

The state is the value calculated by the public method `shapes_out()`.

Internals

During training, the optimizer is cloned and added to the `ModelDescriptor`. Note that the parameter set of the stored `TorchOptimizer` is reference-identical to the parameter set of the pipeop itself.

Super class

`mlr3pipelines::PipeOp` -> `PipeOpTorchOptimizer`

Methods

Public methods:

- `PipeOpTorchOptimizer$new()`
- `PipeOpTorchOptimizer$clone()`

`PipeOpTorchOptimizer$new()`**:** Creates a new instance of this R6 class.

Usage:

```
PipeOpTorchOptimizer$new(
  optimizer = t_opt("adam"),
  id = "torch_optimizer",
  param_vals = list()
)
```

Arguments:

`optimizer` (`TorchOptimizer` or `character(1)` or `torch_optimizer_generator`)
The optimizer (or something convertible via `as_torch_optimizer()`).

`id (character(1))`
: Identifier of the resulting object.

`param_vals (list())`
: List of hyperparameter settings, overwriting the hyperparameter settings that would otherwise be set during construction.

`PipeOpTorchOptimizer$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PipeOpTorchOptimizer$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Model Configuration: `ModelDescriptor()`, `mlr_pipeops_torch_callbacks`, `mlr_pipeops_torch_loss`, `model_descriptor_union()`

Examples

```
po_opt = po("torch_optimizer", "sgd", lr = 0.01)
po_opt$param_set
mdin = po("torch_ingress_num")$train(list(tsk("iris")))
mdin[[1L]]$optimizer
mdout = po_opt$train(mdin)
mdout[[1L]]$optimizer
```

---

mlr_pipeops_trafo_adjust_brightness

Adjust Brightness Transformation

Description

Calls `torchvision::transform_adjust_brightness`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being a `trafo_` operator, its `stages` parameter starts out as `"both"`, so the transformation is applied both while training and when predicting. Set `stages = "train"` to restrict it to training. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("trafo_adjust_brightness")
```

Parameters

Id	Type	Default	Levels	Range
brightness_factor	numeric	-		$[0, \infty)$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

`mlr_pipeops_trafo_adjust_gamma`
Adjust Gamma Transformation

Description

Calls `torchvision::transform_adjust_gamma`, see there for more information on the parameters. The preprocessing is applied to the whole batch.

Being a `trafo_` operator, its `stages` parameter starts out as `"both"`, so the transformation is applied both while training and when predicting. Set `stages = "train"` to restrict it to training. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("trafo_adjust_gamma")
```

Parameters

Id	Type	Default	Levels	Range
gamma	numeric	-		$[0, \infty)$
gain	numeric	1		$(-\infty, \infty)$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

`mlr_pipeops_trafo_adjust_hue`
Adjust Hue Transformation

Description

Calls `torchvision::transform_adjust_hue`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being a `trafo_` operator, its `stages` parameter starts out as `"both"`, so the transformation is applied both while training and when predicting. Set `stages = "train"` to restrict it to training. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("trafo_adjust_hue")
```

Parameters

Id	Type	Default	Levels	Range
hue_factor	numeric	-		$[-0.5, 0.5]$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

`mlr_pipeops_trafo_adjust_saturation`
Adjust Saturation Transformation

Description

Calls `torchvision::transform_adjust_saturation`, see there for more information on the parameters. The preprocessing is applied to each element of a batch individually.

Being a `trafo_` operator, its `stages` parameter starts out as `"both"`, so the transformation is applied both while training and when predicting. Set `stages = "train"` to restrict it to training. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("trafo_adjust_saturation")
```

Parameters

Id	Type	Default	Levels	Range
saturation_factor	numeric	-		$(-\infty, \infty)$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

---

mlr_pipeops_trafo_grayscale

Grayscale Transformation

Description

Calls `torchvision::transform_grayscale`, see there for more information on the parameters. The preprocessing is applied to the whole batch.

Being a `trafo_` operator, its `stages` parameter starts out as `"both"`, so the transformation is applied both while training and when predicting. Set `stages = "train"` to restrict it to training. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("trafo_grayscale")
```

Parameters

Id	Type	Default	Levels	Range
num_output_channels	integer	-		$[1, 3]$
stages	character	-	train, predict, both	-
affect_columns	untyped	selector_all()		-

mlr_pipeops_trafo_nop *No Transformation*

Description

Transforms the input via the identity function.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

mlr_pipeops_trafo_normalize *Normalization Transformation*

Description

Normalizes the channels by subtracting `mean` and dividing by `std`. The channels are the third-to-last dimension, e.g. `C` for images of shape `(N, C, H, W)` or videos of shape `(N, T, C, H, W)`, as in `torchvision::transform_normalize`, which this is the same as for these inputs. For inputs with fewer dimensions, e.g. `(N, C)` or `(N, C, L)`, the channels are the second dimension. Inputs that are not floating point, e.g. images stored as integers, are converted to the default floating point type (`torch_get_default_dtype()`) first. The preprocessing is applied to the whole batch.

Being a `trafo_` operator, its `stages` parameter starts out as `"both"`, so the transformation is applied both while training and when predicting. Set `stages = "train"` to restrict it to training. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("trafo_normalize")
```

Parameters

Id	Type	Default	Levels
mean	untyped	-	
std	untyped	-	
stages	character	-	train, predict, both
affect_columns	untyped	selector_all()	

mlr_pipeops_trafo_pad *Padding Transformation*

Description

Calls `torchvision::transform_pad`, see there for more information on the parameters. The preprocessing is applied to the whole batch.

Being a `trafo_` operator, its `stages` parameter starts out as `"both"`, so the transformation is applied both while training and when predicting. Set `stages = "train"` to restrict it to training. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("trafo_pad")
```

Parameters

Id	Type	Default	Levels
padding	untyped	-	
fill	untyped	0	
padding_mode	character	constant	constant, edge, reflect, symmetric
stages	character	-	train, predict, both
affect_columns	untyped	selector_all()	

mlr_pipeops_trafo_reshape *Reshaping Transformation*

Description

Reshapes the tensor according to the parameter `shape`, by calling `torch_reshape()`. This preprocessing function is applied batch-wise.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Parameters

- shape :: integer()
  The desired output shape. The first dimension is the batch dimension and should usually be -1.

---

mlr_pipeops_trafo_resize

Resizing Transformation

Description

Calls torchvision::transform_resize, see there for more information on the parameters. The preprocessing is applied to the whole batch.

Being a trafo_ operator, its stages parameter starts out as "both", so the transformation is applied both while training and when predicting. Set stages = "train" to restrict it to training. See PipeOpTaskPreprocTorch for the stages parameter and this naming convention.

Format

R6Class inheriting from PipeOpTaskPreprocTorch.

Construction

```
po("trafo_resize")
```

Parameters

Id	Type	Default	Levels
size	untyped	-	
interpolation	character	2	Undefined, Bartlett, Blackman, Bohman, Box, Catrom, Cosine, Cubic, Gaussian
stages	character	-	train, predict, both
affect_columns	untyped	selector_all()	

mlr_pipeops_trafo_rgb_to_grayscale *RGB to Grayscale Transformation*

Description

Calls `torchvision::transform_rgb_to_grayscale`, see there for more information on the parameters. The preprocessing is applied to the whole batch.

Being a `trafo_` operator, its `stages` parameter starts out as `"both"`, so the transformation is applied both while training and when predicting. Set `stages = "train"` to restrict it to training. See `PipeOpTaskPreprocTorch` for the `stages` parameter and this naming convention.

Format

`R6Class` inheriting from `PipeOpTaskPreprocTorch`.

Construction

```
po("trafo_rgb_to_grayscale")
```

Parameters

Id	Type	Default	Levels
stages	character	-	train, predict, both
affect_columns	untyped	selector_all()	

mlr_tasks_cifar *CIFAR Classification Tasks*

Description

The CIFAR-10 and CIFAR-100 datasets. A subset of the 80 million tiny images dataset with noisy labels was supplied to student labelers, who were asked to filter out incorrectly labeled images. The images are have datatype `torch_long()`.

CIFAR-10 contains 10 classes. CIFAR-100 contains 100 classes, which may be partitioned into 20 superclasses of 5 classes each. The CIFAR-10 and CIFAR-100 classes are mutually exclusive. See Chapter 3.1 of the technical report for more details.

The data is obtained from `torchvision::cifar10_dataset()` (or `torchvision::cifar100_dataset()`).

Format

R6::R6Class inheriting from mlr3::TaskClassif.

Construction

```
tsk("cifar10")
tsk("cifar100")
```

Download

The task's backend is a `DataBackendLazy` which will download the data once it is requested. Other meta-data is already available before that. You can cache these datasets by setting the `mlr3torch.cache` option to `TRUE` or to a specific path to be used as the cache directory.

Properties

- Task type: "classif"
- Properties: "multiclass"
- Has Missings: no
- Target: "class"
- Features: "image"
- Data Dimension: 60000x4

References

Krizhevsky A (2009). "Learning Multiple Layers of Features from Tiny Images." *Master's thesis, Department of Computer Science, University of Toronto*.

Examples

```
task_cifar10 = tsk("cifar10")
task_cifar100 = tsk("cifar100")
```

`mlr_tasks_lazy_iris` *Iris Classification Task*

Description

A classification task for the popular datasets::iris data set. Just like the iris task, but the features are represented as one lazy tensor column.

Format

R6::R6Class inheriting from mlr3::TaskClassif.

Construction

```
tsk("lazy_iris")
```

Properties

- Task type: “classif”
- Properties: “multiclass”
- Has Missings: no
- Target: “Species”
- Features: “x”
- Data Dimension: 150x3

Source

https://en.wikipedia.org/wiki/Iris_flower_data_set

References

Anderson E (1936). “The Species Problem in Iris.” *Annals of the Missouri Botanical Garden*, **23**(3), 457. doi:10.2307/2394164.

Examples

```
task = tsk("lazy_iris")
task
df = task$data()
materialize(df$x[1:6], rbind = TRUE)
```

---

mlr_tasks_melanoma *Melanoma Image classification*

Description

Classification of melanoma tumor images. The data is a preprocessed version of the 2020 SIIM-ISIC challenge where the images have been reshaped to size $(3, 128, 128)$.

By default only the training rows are active in the task, but the test data (that has no targets) is also included. Whether an observation is part of the train or test set is indicated by the column `"test"`.

There are no labels for the test rows, so by default, these observations are inactive, which means that the task uses only 32701 of the 43683 observations that are defined in the underlying data backend.

The data backend also contains a more detailed `diagnosis` of the specific type of tumor.

Columns:

- `outcome` (factor): the target variable. Whether the tumor is benign or malignant (the positive class)

- `anatom_site_general_challenge` (factor): the location of the tumor on the patient's body
- `sex` (factor): the sex of the patient
- `age_approx` (int): approximate age of the patient at the time of imaging
- `image` (lazy_tensor): The image (shape $(3, 128, 128)$) of the tumor. ee `split` (character): Whether the observation os part of the train or test set.

Construction

```
tsk("melanoma")
```

Download

The task's backend is a `DataBackendLazy` which will download the data once it is requested. Other meta-data is already available before that. You can cache these datasets by setting the `mlr3torch.cache` option to `TRUE` or to a specific path to be used as the cache directory.

Properties

- Task type: "classif"
- Properties: "twoclass", "groups"
- Has Missings: no
- Target: "outcome"
- Features: "sex", "anatom_site_general_challenge", "age_approx", "image"
- Data Dimension: 43683x11

Source

https://huggingface.co/datasets/carsonzhang/ISIC_2020_small

References

Rotemberg V, Kurtansky N, Betz-Stablein B, Caffery L, Chousakos E, Codella N, Combalia M, Dusza S, Guitera P, Gutman D, Halpern A, Helba B, Kittler H, Kose K, Langer S, Lioprys K, Malvehy J, Musthaq S, Nanda J, Reiter O, Shih G, Stratigos A, Tschandl P, Weber J, Soyer P (2021). "A patient-centric dataset of images and metadata for identifying melanomas using clinical context." *Scientific Data*, **8**, 34. doi:10.1038/s4159702100815z.

Examples

```
task = tsk("melanoma")
```

`mlr_tasks_mnist` *MNIST Image classification*

Description

Classic MNIST image classification.

The underlying `DataBackend` contains columns `"label"`, `"image"`, `"row_id"`, `"split"`, where the last column indicates whether the row belongs to the train or test set.

The first 60000 rows belong to the training set, the last 10000 rows to the test set.

Construction

```
tsk("mnist")
```

Download

The task's backend is a `DataBackendLazy` which will download the data once it is requested. Other meta-data is already available before that. You can cache these datasets by setting the `mlr3torch.cache` option to `TRUE` or to a specific path to be used as the cache directory.

Properties

- Task type: "classif"
- Properties: "multiclass"
- Has Missings: no
- Target: "label"
- Features: "image"
- Data Dimension: 70000x4

Source

https://torchvision.mlverse.org/reference/mnist_dataset.html

References

Lecun Y, Bottou L, Bengio Y, Haffner P (1998). "Gradient-based learning applied to document recognition." *Proceedings of the IEEE*, **86**(11), 2278-2324. doi:10.1109/5.726791.

Examples

```
task = tsk("mnist")
```

`mlr_tasks_tiny_imagenet` *Tiny ImageNet Classification Task*

Description

Subset of the famous ImageNet dataset. The data is obtained from `torchvision::tiny_imagenet_dataset()`.

The underlying `DataBackend` contains columns `"class"`, `"image"`, `"..row_id"`, `"split"`, where the last column indicates whether the row belongs to the train, validation or test set that are provided in torchvision.

There are no labels for the test rows, so by default, these observations are inactive, which means that the task uses only 110000 of the 120000 observations that are defined in the underlying data backend.

Construction

```
tsk("tiny_imagenet")
```

Download

The task's backend is a `DataBackendLazy` which will download the data once it is requested. Other meta-data is already available before that. You can cache these datasets by setting the `mlr3torch.cache` option to `TRUE` or to a specific path to be used as the cache directory.

Properties

- Task type: "classif"
- Properties: "multiclass"
- Has Missings: no
- Target: "class"
- Features: "image"
- Data Dimension: 120000x4

References

Deng J, Dong W, Socher R, Li L, Li K, Fei-Fei L (2009). "Imagenet: A large-scale hierarchical image database." In *2009 IEEE conference on computer vision and pattern recognition*, 248–255. IEEE.

Examples

```
task = tsk("tiny_imagenet")
```

`mlr_tasks_torch` *Generic Torch Task*

Description

A general-purpose `Task` that can be used for modeling arbitrary problems, including supervised and unsupervised problems. The article on *Custom Learning Problems* covers all of this in more detail.

The problem this generic task solves is that it is rather complicated to register new task types with `mlr3`, so this class makes this easier. The price of this flexibility is the loss of some compatibility checks.

Super class

`mlr3::Task` -> `TaskTorch`

Active bindings

`hash` (`character(1)`)
The hash of the task.

`default_encoder` (`function()` or `NULL`)
The default prediction encoder. Read-only.

`default_target_batchgetter` (`function()` or `NULL`)
See the construction argument. Read-only, for the same reason as `default_encoder`.

`default_measure` (`Measure` or `NULL`)
See the construction argument. Read-only, for the same reason as `default_encoder`.

`output_dim` (`function()` or `NULL`)
See the construction argument. Called by `output_dim_for()`.

Methods

Public methods:

- `TaskTorch$new()`
- `TaskTorch$truth()`
- `TaskTorch$clone()`

`TaskTorch$new()`: Creates a new instance of this R6 class.

Usage:

```
TaskTorch$new(
  id,
  backend,
  target = NULL,
  label = NA_character_,
  output_dim = NULL,
  default_encoder = NULL,
  default_target_batchgetter = NULL,
```

```
  default_measure = NULL
)
```

Arguments:

`id` (`character(1)`)
: The id of the task.

`backend` (`DataBackend` or `data.frame()`)
: The data.

`target` (`character()` or `NULL`)
: The names of the target columns. `NULL` (default) for a task without a target, see section *Tasks without a Target* of `TaskTorch`.

`label` (`character(1)`)
: The label of the task.

`output_dim` (`function()` or `NULL`)
: Returns the number of output units the network needs. Takes an argument `task` and returns a single positive integer. May be `NULL` (default), in which case any caller of `output_dim_for()` errors.

`default_encoder` (`function()` or `NULL`)
: The default prediction encoder for the task. This can be overwritten by a learner's private `$.encode_prediction` method. See `LearnerTorch` for more information.

`default_target_batchgetter` (`function()` or `NULL`)
: The default way to turn the target columns of a batch into the target tensor `y`, returned by `get_target_batchgetter()`. Takes an argument `data`, a `data.table` with only the target columns, and optionally an argument `x`, the named list of feature tensors of the batch. What `y` has to look like follows from the loss, so this can be overwritten by the `target_batchgetter` of a learner such as `lrn("torch.module")`. If `NULL` (default), a task with a target has no default and the learner has to provide one.

`default_measure` (`Measure` or `NULL`)
: The default measure of the task, i.e. what `msr("torch.default")` resolves to.

`TaskTorch$truth()`: The ground truth, see section *Scoring*. Might return `NULL` for unsupervised problems.

Usage:

```
TaskTorch$truth(rows = NULL)
```

Arguments:

`rows` (`integer()`)
: The rows to return the truth for. All rows if `NULL`.

`TaskTorch$clone()`: The objects of this class are cloneable with this method.

Usage:

```
TaskTorch$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# multi-label classification: one logical column per label
d = data.frame(x1 = rnorm(50), x2 = rnorm(50))
d$a = d$x1 > 0
d$b = d$x2 > 0
task = as_task_torch(d, target = c("a", "b"), id = "labels",
  output_dim = function(task) length(task$target_names),
  default_encoder = function(task, network_output, predict_type) {
    prob = as.matrix(torch::nnf_sigmoid(network_output)$cpu())
    colnames(prob) = task$target_names
    list(response = prob > 0.5, prob = if (predict_type == "prob") prob)
  },
  default_target_batchgetter = function(data) {
    torch::torch_tensor(as.matrix(data), dtype = torch::torch_float())
  })
task
output_dim_for(task)
```

---

mlr3torch_callbacks *Dictionary of Torch Callbacks*

Description

A mlr3misc::Dictionary of torch callbacks. Use t_clbk() to conveniently retrieve callbacks. Can be converted to a data.table using as.data.table.

Usage

```
mlr3torch_callbacks
```

See Also

Other Callback: TorchCallback, as_torch_callback(), as_torch_callbacks(), callback_set(), mlr_callback_set, mlr_callback_set.checkpoint, mlr_callback_set.progress, mlr_callback_set.tb, mlr_callback_set.unfreeze, mlr_context_torch, t_clbk(), torch_callback()

Other Dictionary: mlr3torch_losses, mlr3torch_optimizers, t_opt()

Examples

```
mlr3torch_callbacks$get("checkpoint")
# is the same as
t_clbk("checkpoint")
# convert to a data.table
as.data.table(mlr3torch_callbacks)
```

mlr3torch_losses *Loss Functions*

Description

Dictionary of torch loss descriptors. See `t_loss()` for conveniently retrieving a loss function. Can be converted to a `data.table` using `as.data.table`.

Usage

```
mlr3torch_losses
```

Available Loss Functions

cross_entropy, l1, mse

See Also

Other Torch Descriptor: `TorchCallback`, `TorchDescriptor`, `TorchLoss`, `TorchOptimizer`, `as_torch_callbacks()`, `as_torch_loss()`, `as_torch_optimizer()`, `mlr3torch_optimizers`, `t_clbk()`, `t_loss()`, `t_opt()`

Other Dictionary: `mlr3torch_callbacks`, `mlr3torch_optimizers`, `t_opt()`

Examples

```
mlr3torch_losses$get("mse")
# is equivalent to
t_loss("mse")
# convert to a data.table
as.data.table(mlr3torch_losses)
```

mlr3torch_optimizers *Optimizers*

Description

Dictionary of torch optimizers. Use `t_opt` for conveniently retrieving optimizers. Can be converted to a `data.table` using `as.data.table`.

Usage

```
mlr3torch_optimizers
```

Available Optimizers

adagrad, adam, adamw, rmsprop, sgd

See Also

Other Torch Descriptor: TorchCallback, TorchDescriptor, TorchLoss, TorchOptimizer, as_torch_callbacks(), as_torch_loss(), as_torch_optimizer(), mlr3torch_losses, t_clbk(), t_loss(), t_opt()

Other Dictionary: mlr3torch_callbacks, mlr3torch_losses, t_opt()

Examples

```
mlr3torch_optimizers$get("adam")
# is equivalent to
t_opt("adam")
# convert to a data.table
as.data.table(mlr3torch_optimizers)
```

---

model_descriptor_to_learner

Create a Torch Learner from a ModelDescriptor

Description

First a nn_graph is created using model_descriptor_to_module and then a learner is created from this module and the remaining information from the model descriptor, which must include the optimizer and loss function and optionally callbacks.

Usage

```
model_descriptor_to_learner(model_descriptor)
```

Arguments

model_descriptor	(ModelDescriptor) The model descriptor.

Value

Learner

See Also

Other Graph Network: ModelDescriptor(), TorchIngressToken(), as_learner_torch(), mlr_learners_torch_model mlr_pipeops_module, mlr_pipeops_torch, mlr_pipeops_torch_ingress, mlr_pipeops_torch_ingress_categ, mlr_pipeops_torch_ingress_ltnsr, mlr_pipeops_torch_ingress_num, model_descriptor_to_module(), model_descriptor_union(), nn_graph(), pipeop_torch()

model_descriptor_to_module *Create a nn_graph from ModelDescriptor*

Description

Creates the nn_graph from a ModelDescriptor. Mostly for internal use, since the ModelDescriptor is in most circumstances harder to use than just creating nn_graph directly.

Usage

```
model_descriptor_to_module(
  model_descriptor,
  output_pointers = NULL,
  list_output = FALSE
)
```

Arguments

model_descriptor	(ModelDescriptor) Model Descriptor. pointer is ignored, instead output_pointer values are used. $graph member is modified by-reference.
output_pointers	(list of character) Collection of pointers that indicate what part of the model_descriptor$graph is being used for output. Entries have the format of ModelDescriptor$pointer.
list_output	(logical(1)) Whether output should be a list of tensors. If FALSE, then length(output_pointers) must be 1.

Value

nn_graph

See Also

Other Graph Network: ModelDescriptor(), TorchIngressToken(), as_learner_torch(), mlr_learners_torch_model mlr_pipeops_module, mlr_pipeops_torch, mlr_pipeops_torch_ingress, mlr_pipeops_torch_ingress_categ, mlr_pipeops_torch_ingress_ltnsr, mlr_pipeops_torch_ingress_num, model_descriptor_to_learner(), model_descriptor_union(), nn_graph(), pipeop_torch()

`model_descriptor_union` *Union of ModelDescriptors*

Description

This is a mostly internal function that is used in `PipeOpTorch`s with multiple input channels.

It creates the union of multiple `ModelDescriptor`s:

- `graph`s are combinded (if they are not identical to begin with). The first entry's `graph` is modified by reference.
- `PipeOp`s with the same ID must be identical. No new input edges may be added to `PipeOp`s.
- Drops `pointer` / `pointer_shape` entries.
- The new task is the feature union of the two incoming tasks.
- The `optimizer` and `loss` of both `ModelDescriptor`s must be identical.
- Ingress tokens and callbacks are merged, where objects with the same `"id"` must be identical.

Usage

```
model_descriptor_union(md1, md2)
```

Arguments

`md1`	(`ModelDescriptor`) The first `ModelDescriptor`.
`md2`	(`ModelDescriptor`) The second `ModelDescriptor`.

Details

The requirement that no new input edgedes may be added to `PipeOp`s is not theoretically necessary, but since we assume that ModelDescriptor is being built from beginning to end (i.e. `PipeOp`s never get new ancestors) we can make this assumption and simplify things. Otherwise we'd need to treat "..."-inputs special.)

Value

`ModelDescriptor`

See Also

Other Graph Network: `ModelDescriptor()`, `TorchIngressToken()`, `as_learner_torch()`, `mlr_learners_torch_model`, `mlr_pipeops_module`, `mlr_pipeops_torch`, `mlr_pipeops_torch_ingress`, `mlr_pipeops_torch_ingress_categ`, `mlr_pipeops_torch_ingress_ltnsr`, `mlr_pipeops_torch_ingress_num`, `model_descriptor_to_learner()`, `model_descriptor_to_module()`, `nn_graph()`, `pipeop_torch()`

Other Model Configuration: `ModelDescriptor()`, `mlr_pipeops_torch_callbacks`, `mlr_pipeops_torch_loss`, `mlr_pipeops_torch_optimizer`

`ModelDescriptor` *Represent a Model with Meta-Info*

Description

Represents a *model*; possibly a complete model, possibly one in the process of being built up.

This model takes input tensors of shapes `shapes_in` and pipes them through `graph`. Input shapes get mapped to input channels of `graph`. Output shapes are named by the output channels of `graph`; it is also possible to represent no-ops on tensors, in which case names of input and output should be identical.

`ModelDescriptor` objects typically represent partial models being built up, in which case the `pointer` slot indicates a specific point in the graph that produces a tensor of shape `pointer_shape`, on which the graph should be extended. It is allowed for the `graph` in this structure to be modified by-reference in different parts of the code. However, these modifications may never add edges with elements of the `Graph` as destination. In particular, no element of `graph$input` may be removed by reference, e.g. by adding an edge to the `Graph` that has the input channel of a `PipeOp` that was previously without parent as its destination.

In most cases it is better to create a specific `ModelDescriptor` by training a `Graph` consisting (mostly) of operators `PipeOpTorchIngress`, `PipeOpTorch`, `PipeOpTorchLoss`, `PipeOpTorchOptimizer`, and `PipeOpTorchCallbacks`.

A `ModelDescriptor` can be converted to a `nn_graph` via `model_descriptor_to_module`.

Usage

```
ModelDescriptor(
  graph,
  ingress,
  task,
  optimizer = NULL,
  loss = NULL,
  callbacks = NULL,
  pointer = NULL,
  pointer_shape = NULL
)
```

Arguments

`graph`	(`Graph`) `Graph` of `PipeOpModule` and `PipeOpNOP` operators.
`ingress`	(uniquely named `list` of `TorchIngressToken`) List of inputs that go into `graph`. Names of this must be a subset of `graph$input$name`.
`task`	(`Task`) (Training)-Task for which the model is being built. May be necessary for some aspects of what loss to use etc.

optimizer (TorchOptimizer | NULL)
Additional info: what optimizer to use.

loss (TorchLoss | NULL)
Additional info: what loss to use.

callbacks (A list of CallbackSet or NULL)
Additional info: what callbacks to use.

pointer (character(2) | NULL)
Indicating an element on which a model is. Points to an output channel within graph: Element 1 is the PipeOp's id and element 2 is that PipeOp's output channel.

pointer_shape (integer | NULL)
Shape of the output indicated by pointer.

Value

(ModelDescriptor)

See Also

Other Model Configuration: mlr_pipeops_torch_callbacks, mlr_pipeops_torch_loss, mlr_pipeops_torch_optimize
model_descriptor_union()

Other Graph Network: TorchIngressToken(), as_learner_torch(), mlr_learners_torch_model, mlr_pipeops_module, mlr_pipeops_torch, mlr_pipeops_torch_ingress, mlr_pipeops_torch_ingress_categ, mlr_pipeops_torch_ingress_ltnsr, mlr_pipeops_torch_ingress_num, model_descriptor_to_learner(), model_descriptor_to_module(), model_descriptor_union(), nn_graph(), pipeop_torch()

---

msr_torch *Create a Measure for a Generic Torch Task*

Description

Short form for constructing a MeasureTorch. See the *Custom Learning Problems* article for how to create and use such measures.

Usage

```
msr_torch(
  id,
  fun,
  minimize = NA,
  range = c(-Inf, Inf),
  predict_type = "response",
  properties = character(),
  label = NA_character_,
  obs_loss = NULL
)
```

Arguments

id	(character(1)) The id of the measure.
fun	(function()) The scoring function. It receives whichever of the arguments truth, response, prob, se, lazy_tensor, prediction, task, learner, train_set and weights it declares, and must return a single number. Asking for anything else is an error. An argument that the prediction does not have – weights on a task without a weights_measure column, or prob on a response-only prediction – is not passed at all, so a default declared for it is what the function sees. weights are also withheld when the measure's $use_weights is set to "ignore", so give the argument a default if the measure should score with and without weights, and none if it cannot do without them – asking for weights that do not exist is an error, not a score.
minimize	(logical(1)) Whether a smaller score is better. NA (default) means the direction is unknown.
range	(numeric(2)) The range of possible scores.
predict_type	(character(1)) The predict type the measure requires: "response" (default), "prob", "se" or "lazy_tensor". A measure asking for one it did not declare here still receives it, if the prediction has it.
properties	(character()) Properties of the measure, see Measure. The "requires_task", "requires_learner", "requires_train_set" and "weights" properties are added automatically when fun declares the corresponding argument. "requires_model" is not: a learner argument only says that the learner object is needed, and mlr3 hands that over even when the model was not stored – after a resample() with store_models = FALSE, learner$network is then NULL and a measure reading it scores whatever an empty model gives. Pass properties = "requires_model" yourself whenever the measure reaches for the trained network, so that mlr3 refuses to score instead.
label	(character(1)) The label of the measure.
obs_loss	(function() or NULL) The per-observation loss. Declared like fun, except that train_set is not available here, and if specified adds the "obs_loss" property. It must return one number per observation: a multi-target loss reduces over the targets (rowMeans()), not over the observations (mean()), and returning a single number is an error rather than a column of that number repeated.

Value

MeasureTorch

Examples

```
m = msr_torch("hamming", function(truth, response) mean(as.matrix(truth) != response))
m$properties

# with a per-observation loss
m = msr_torch("mse", function(truth, response) mean((truth - response)^2),
  obs_loss = function(truth, response) (truth - response)^2)
m$properties
```

---

nn *Create a Neural Network Layer*

Description

Retrieve a neural network layer from the mlr_pipeops dictionary.

The id of the returned PipeOp is .key. Because the ids within a Graph must be unique, .key can be suffixed with _<n> to disambiguate repeated layers, e.g. nn("linear_1") and nn("linear_2").

Usage

```
nn(.key, ...)
```

Arguments

.key	(character(1)) The key of the layer in the dictionary, optionally followed by a _<n> suffix.
...	(any) Additional parameters, constructor arguments or fields.

Examples

```
po1 = po("nn_linear", id = "linear")
# is the same as:
po2 = nn("linear")

# the `_<n>` suffix is part of the id, but not of the dictionary key
nn("linear_1")$id
```

nn_ft_cls *CLS Token for FT-Transformer*

Description

Concatenates a CLS token to the input as the last feature. The input shape is expected to be `(batch, n_features, d_token)` and the output shape is `(batch, n_features + 1, d_token)`.

This is used in the `LearnerTorchFTTransformer`.

Usage

```
nn_ft_cls(d_token, initialization)
```

Arguments

`d_token`	`(integer(1))` The dimension of the embedding.
`initialization`	`(character(1))` The initialization method for the embedding weights. Possible values are `"uniform"` and `"normal"`.

References

Devlin J, Chang M, Lee K, Toutanova K (2018). "Bert: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805*.

nn_ft_transformer_block *Single Transformer Block for FT-Transformer*

Description

A transformer block consisting of a multi-head self-attention mechanism followed by a feed-forward network.

This is used in `LearnerTorchFTTransformer`.

Usage

```
nn_ft_transformer_block(
  d_token,
  attention_n_heads,
  attention_dropout,
  attention_initialization,
  ffn_d_hidden = NULL,
```

```
  ffn_d_hidden_multiplier = NULL,
  ffn_dropout,
  ffn_activation,
  residual_dropout,
  prenormalization,
  is_first_layer,
  attention_normalization,
  ffn_normalization,
  query_idx = NULL,
  attention_bias,
  ffn_bias_first,
  ffn_bias_second
)
```

Arguments

`d_token`	(`integer(1)`) The dimension of the embedding.
`attention_n_heads`	(`integer(1)`) Number of attention heads.
`attention_dropout`	(`numeric(1)`) Dropout probability in the attention mechanism.
`attention_initialization`	(`character(1)`) Initialization of the query, key and value projections, either "`kaiming`" or "`xavier`".
`ffn_d_hidden`	(`integer(1)`) Hidden dimension of the feed-forward network. Multiplied by 2 if using ReGLU or GeGLU activation.
`ffn_d_hidden_multiplier`	(`numeric(1)`) Alternative way to specify the hidden dimension of the feed-forward network as `d_token` * `d_hidden_multiplier`. Also multiplied by 2 if using RegLU or GeGLU activation.
`ffn_dropout`	(`numeric(1)`) Dropout probability in the feed-forward network.
`ffn_activation`	(`nn_module`) Activation function for the feed-forward network.
`residual_dropout`	(`numeric(1)`) Dropout probability for residual connections.
`prenormalization`	(`logical(1)`) Whether to apply normalization before attention and FFN (`TRUE`) or after (`FALSE`).
`is_first_layer`	(`logical(1)`) Whether this is the first layer in the transformer stack.

attention_normalization
: (nn_module)
Normalization module to use for attention.

ffn_normalization
: (nn_module)
Normalization module to use for the feed-forward network.

query_idx
: (integer() or NULL)
Indices of the tensor to apply attention to. Should not be set manually. If NULL, then attention is applied to the entire tensor. In the last block in a stack of transformers, this is set to -1 so that attention is applied only to the embedding of the CLS token.

attention_bias
: (logical(1))
Whether attention has a bias.

ffn_bias_first
: (logical(1))
Whether the first layer in the FFN has a bias.

ffn_bias_second
: (logical(1))
Whether the second layer in the FFN has a bias.

References

Devlin J, Chang M, Lee K, Toutanova K (2018). "Bert: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805*. Gorishniy Y, Rubachev I, Khrulkov V, Babenko A (2021). "Revisiting Deep Learning for Tabular Data." *arXiv*, **2106.11959**.

---

nn_geglu *GeGLU Module*

Description

This module implements the Gaussian Error Linear Unit Gated Linear Unit (GeGLU) activation function. It computes GeGLU$(x, g) = x \cdot \text{GELU}(g)$ where \(x\) and \(g\) are created by splitting the input tensor in half along the last dimension.

Usage

```
nn_geglu()
```

References

Shazeer N (2020). "GLU Variants Improve Transformer." 2002.05202, https://arxiv.org/abs/2002.05202.

Examples

```
x = torch::torch_randn(10, 10)
glu = nn_geglu()
glu(x)
```

nn_graph *Graph Network*

Description

Represents a neural network using a Graph that contains mostly PipeOpModules.

Usage

```
nn_graph(graph, shapes_in, output_map = graph$output$name, list_output = FALSE)
```

Arguments

graph	(Graph) The Graph to wrap. Is **not** cloned.
shapes_in	(named integer) Shape info of tensors that go into graph. Names must be graph$input$name, possibly in different order.
output_map	(character) Which of graph's outputs to use. Must be a subset of graph$output$name.
list_output	(logical(1)) Whether output should be a list of tensors. If FALSE (default), then length(output_map) must be 1.

Value

nn_graph

Fields

- graph :: Graph
  The graph (consisting primarily of PipeOpModules) that is wrapped by the network. When it consists only of PipeOpModules and PipeOpNOPs, the order in which the modules are called is determined once and $forward() calls them directly, which is much faster than running the graph. This is determined anew when PipeOps are added to or removed from the graph, but not when only its edges are changed, so the graph should not be modified in place. The field cannot be set to another graph, because the parameters of the modules are registered when the network is created. Create a new nn_graph instead.
- input_map :: character()
  The names of the input arguments of the network. Unnamed arguments of $forward() are matched in this order.
- shapes_in :: list()
  The shapes of the input tensors of the network.
- output_map :: character()
  Which output elements of the graph are returned by the $forward() method.

- `list_output` :: `logical(1)`
  Whether the output is a list of tensors.
- `module_list` :: `nn_module_list`
  The list of modules in the network.
- `list_output` :: `logical(1)`
  Whether the output is a list of tensors.

See Also

Other Graph Network: `ModelDescriptor()`, `TorchIngressToken()`, `as_learner_torch()`, `mlr_learners_torch_model`, `mlr_pipeops_module`, `mlr_pipeops_torch`, `mlr_pipeops_torch_ingress`, `mlr_pipeops_torch_ingress_categ`, `mlr_pipeops_torch_ingress_ltnsr`, `mlr_pipeops_torch_ingress_num`, `model_descriptor_to_learner()`, `model_descriptor_to_module()`, `model_descriptor_union()`, `pipeop_torch()`

Examples

```
graph = mlr3pipelines::Graph$new()
graph$add_pipeop(po("module_1", module = nn_linear(10, 20)), clone = FALSE)
graph$add_pipeop(po("module_2", module = nn_relu()), clone = FALSE)
graph$add_pipeop(po("module_3", module = nn_linear(20, 1)), clone = FALSE)
graph$add_edge("module_1", "module_2")
graph$add_edge("module_2", "module_3")

network = nn_graph(graph, shapes_in = list(module_1.input = c(NA, 10)))

x = torch_randn(16, 10)

network(module_1.input = x)
```

---

nn_merge_cat *Concatenates multiple tensors*

Description

Concatenates multiple tensors on a given dimension. No broadcasting rules are applied here, you must reshape the tensors before to have the same shape.

Usage

```
nn_merge_cat(dim = -1)
```

Arguments

`dim`	(`integer(1)`) The dimension for the concatenation.

nn_merge_prod *Product of multiple tensors*

Description

Calculates the product of all input tensors.

Usage

```
nn_merge_prod()
```

nn_merge_sum *Sum of multiple tensors*

Description

Calculates the sum of all input tensors.

Usage

```
nn_merge_sum()
```

nn_reglu *ReGLU Module*

Description

Rectified Gated Linear Unit (ReGLU) module. Computes the output as $\text{ReGLU}(x, g) = x \cdot \text{ReLU}(g)$ where \(x\) and \(g\) are created by splitting the input tensor in half along the last dimension.

Usage

```
nn_reglu()
```

References

Shazeer N (2020). "GLU Variants Improve Transformer." 2002.05202, https://arxiv.org/abs/2002.05202.

Examples

```
x = torch::torch_randn(10, 10)
reglu = nn_reglu()
reglu(x)
```

nn_reshape *Reshape*

Description

Reshape a tensor to the given shape.

Usage

```
nn_reshape(shape)
```

Arguments

shape	(integer() | function()) The desired output shape, or a function(shape) that is called on the shape of the input tensor and returns it.

nn_squeeze *Squeeze*

Description

Squeezes a tensor by calling torch::torch_squeeze() with the given dimension dim.

Usage

```
nn_squeeze(dim)
```

Arguments

dim	(integer()) The dimension to squeeze.

nn_tokenizer_categ *Categorical Tokenizer*

Description

Tokenizes categorical features into a dense embedding. For an input of shape `(batch, n_features)` the output shape is `(batch, n_features, d_token)`.

Usage

```
nn_tokenizer_categ(cardinalities, d_token, bias, initialization)
```

Arguments

`cardinalities`	(`integer()`) The number of categories for each feature.
`d_token`	(`integer(1)`) The dimension of the embedding.
`bias`	(`logical(1)`) Whether to use a bias.
`initialization`	(`character(1)`) The initialization method for the embedding weights. Possible values are `"uniform"` and `"normal"`.

References

Gorishniy Y, Rubachev I, Khrulkov V, Babenko A (2021). "Revisiting Deep Learning for Tabular Data." *arXiv*, **2106.11959**.

nn_tokenizer_num *Numeric Tokenizer*

Description

Tokenizes numeric features into a dense embedding. For an input of shape `(batch, n_features)` the output shape is `(batch, n_features, d_token)`.

Usage

```
nn_tokenizer_num(n_features, d_token, bias, initialization)
```

Arguments

`n_features`	(`integer(1)`) The number of features.
`d_token`	(`integer(1)`) The dimension of the embedding.
`bias`	(`logical(1)`) Whether to use a bias.
`initialization`	(`character(1)`) The initialization method for the embedding weights. Possible values are `"uniform"` and `"normal"`.

References

Gorishniy Y, Rubachev I, Khrulkov V, Babenko A (2021). "Revisiting Deep Learning for Tabular Data." *arXiv*, **2106.11959**.

`nn_unsqueeze` *Unsqueeze*

Description

Unsqueezes a tensor by calling `torch::torch_unsqueeze()` with the given dimension `dim`.

Usage

```
nn_unsqueeze(dim)
```

Arguments

`dim`	(`integer(1)`) The dimension to unsqueeze.

`output_dim_for` *Network Output Dimension*

Description

Calculates the output dimension of a neural network for a given task that is expected by **mlr3torch**. For classification, this is the number of classes (unless it is a binary classification task, where it is 1). For regression, it is 1.

This is an S3 generic and the single place where **mlr3torch** decides how many output neurons a task needs: it is what `PipeOpTorchHead` and the `LearnerTorch`s that build their own head ask. Adding a method for a new task type is therefore the way to support it, see the "Supporting Other Task Types" section of `PipeOpTorchHead`.

Usage

```
output_dim_for(x, ...)
```

Arguments

x	(any) The task.
...	(any) Additional arguments. Not used yet.

Value

(integer(1)) The number of output neurons.

See Also

PipeOpTorchHead

---

pipeop_preproc_torch *Create Torch Preprocessing PipeOps*

Description

Function to create objects of class PipeOpTaskPreprocTorch in a more convenient way. Start by reading the documentation of PipeOpTaskPreprocTorch.

Usage

```
pipeop_preproc_torch(
  id,
  fn,
  shapes_out = NULL,
  param_set = NULL,
  packages = character(0),
  rowwise = FALSE,
  parent_env = parent.frame(),
  stages_init = NULL,
  tags = NULL
)
```

Arguments

id	(character(1)) The id for of the new object.
fn	(function) The preprocessing function.

shapes_out (function or NULL or "infer")
The private .shapes_out(shapes_in, param_vals, task) method of PipeOpTaskPreprocTorch (see section Inheriting). Special values are NULL and "infer": If NULL, the output shapes are unknown. Option "infer" uses infer_shapes. Method "infer" should be correct in most cases, but might fail in some edge cases.

param_set (ParamSet or NULL)
The parameter set. If this is left as NULL (default) the parameter set is inferred in the following way: All parameters but the first and ... of fn are set as untyped parameters with tags 'train' and those that have no default value are tagged as 'required' as well. Default values are not annotated.

packages (character())
The R packages this object depends on.

rowwise (logical(1))
Whether the preprocessing is applied row-wise.

parent_env (environment)
The parent environment for the R6 class.

stages_init (character(1))
Initial value for the stages parameter. If NULL (default), will be set to "both" in case the id starts with "trafo" and to "train" if it starts with "augment". Otherwise it must specified.

tags (character())
Tags for the pipeop

Value

An R6Class instance inheriting from PipeOpTaskPreprocTorch

Examples

```
PipeOpPreprocExample = pipeop_preproc_torch("preproc_example", function(x, a) x + a)
po_example = PipeOpPreprocExample$new()
po_example$param_set
```

---

pipeop_torch *Create a PipeOpTorch*

Description

Helper function to create a custom PipeOpTorch class for the most common cases. A practical guide to this function is the article Writing your own PipeOpTorch. For more information and the more general case, see the *Inheriting* section of PipeOpTorch. The function works similarly to nn_module(), except that $initialize() can take two further arguments that the PipeOp supplies: shapes_in and task and that it also needs to implement the method $shapes_out() which belongs to the PipeOpTorch.

Usage

```
pipeop_torch(
  id,
  initialize = NULL,
  forward,
  shapes_out,
  param_set = NULL,
  in_channels = NULL,
  out_channels = 1L,
  packages = character(0),
  tags = NULL,
  classname = NULL,
  parent_env = parent.frame()
)
```

Arguments

id	(character(1)) The id for of the new object.
initialize	(function or NULL) The $initialize() method of the module. Its arguments become the hyperparameters of the PipeOp, except for shapes_in (the input shapes, named after the input channels) and task (the Task or NULL), which are supplied by the PipeOp and passed only if the function declares them.
forward	(function) The $forward() method of the module.
shapes_out	(function) The shapes of the tensors that the module produces, as a function of shapes_in, param_vals and task – only the arguments that are declared are passed, so an operator whose output shape depends on nothing else is written as function(shapes_in). Any dimension of shapes_in can be NA, i.e. unknown, so this function must not assume that a dimension it reads is known; see the *Inheriting* section of PipeOpTorch.
param_set	(ParamSet or NULL) The parameter set. If left as NULL (default), it is inferred from the arguments of initialize: All arguments but shapes_in and task become an untyped parameter tagged "train", and additionally "required" if it has no default.
in_channels	(character() or integer(1) or NULL) The input channels, either as names or as a count, where 0 means a single *vararg* channel. If NULL (default), the arguments of forward are used.
out_channels	(character() or integer(1)) The output channels, either as names or as a count, 1 by default. A module with more than one output channel must return a list(), in the order of the channels.
packages	(character()) The R packages this object depends on.

tags (character())
Tags for the PipeOp. The tag "torch" is always added.

classname (character(1))
The class name of the generated R6Class. By default it is derived from id: a leading "nn_" is dropped, the remaining _-separated words are capitalized and pasted together, and "PipeOpTorch" is prepended, so the id "nn_scale" gives "PipeOpTorchScale".

parent_env (environment)
The environment in which the module's methods are evaluated, the calling environment by default, as for nn_module(). initialize and forward become methods of the module and are therefore evaluated in this environment rather than in the one they were written in; shapes_out, which is not a method of the module, keeps its own environment. The two only differ when the functions are written somewhere other than the caller of pipeop_torch(), e.g. in a function that wraps it.

Value

An R6Class generator inheriting from PipeOpTorch.

See Also

Other Graph Network: ModelDescriptor(), TorchIngressToken(), as_learner_torch(), mlr_learners_torch_model, mlr_pipeops_module, mlr_pipeops_torch, mlr_pipeops_torch_ingress, mlr_pipeops_torch_ingress_categ, mlr_pipeops_torch_ingress_ltnsr, mlr_pipeops_torch_ingress_num, model_descriptor_to_learner(), model_descriptor_to_module(), model_descriptor_union(), nn_graph()

Examples

```
# A layer that scales its input by a learned factor
# Note that the number of features is read from the input shape
PipeOpTorchCustomScale = pipeop_torch("nn_custom_scale",
  initialize = function(shapes_in, init = 1) {
    self$weight = nn_parameter(torch_full(tail(shapes_in[[1L]], 1L), init))
  },
  forward = function(input) input * self$weight,
  shapes_out = function(shapes_in) shapes_in # scaling leaves the shape as it is
)

po_custom_scale = PipeOpTorchCustomScale$new()
# `init` is a hyperparameter, the number of features is not
po_custom_scale$param_set$ids()
po_custom_scale$shapes_out(list(c(NA, 4)))
# the operator can now be used like any other, and the module is built with 4 features
md = po("torch_ingress_num") %>>% po_custom_scale %>>% po("nn_head")
network = model_descriptor_to_module(md$train(tsk("iris"))[[1L]])
network

# To use it via `nn("custom_scale")` or `po("nn_custom_scale")` we could run the line below:
# mlr_pipeops$add("nn_custom_scale", PipeOpTorchCustomScale)
```

PredictionTorch *Prediction Object for a Generic Torch Task*

Description

The Prediction object returned by learners that were trained on a TaskTorch.

Because a TaskTorch can represent very different learning problems, this class does not prescribe much about how truth, response, prob and se are stored. This is defined by the task's prediction encoder, where you need to ensure that the first dimension indexes the observations. Within that, an element may be an atomic vector, a matrix(), an array() of any dimensionality, a data.table or a lazy_tensor. truth is whatever task$truth() returned – a vector for one target, a data.table for several, a lazy_tensor for a lazy tensor column, and nothing at all for a task without a target.

When a prediction is converted to a data.table, which is e.g. used for the printer, the conversion depends on the type of the object. A prob *matrix* spreads into one column per class, the way it does for a classification prediction. Everything else that is wider than one value per observation becomes a single column whose cells hold that observation's own array, printed as its shape – <array[3]> for a response matrix with three columns, <array[3x224x224]> for a prob with a class dimension and two spatial ones.

Missing Predictions

Only a response with a single value per observation can report a *missing* prediction, so $missing can only be TRUE for scalars and is always FALSE for parially missing predictions.

Super class

mlr3::Prediction -> PredictionTorch

Active bindings

response (any)
: The predicted response.

prob (any)
: The predicted probabilities.

se (any)
: The standard errors of the prediction.

lazy_tensor (lazy_tensor or data.table of them)
: The output of the network, for the predict type "lazy_tensor".

Methods

Public methods:

- PredictionTorch$new()
- PredictionTorch$clone()

PredictionTorch$new(): Creates a new instance of this R6 class.

Usage:

```
PredictionTorch$new(
  task = NULL,
  row_ids = task$row_ids,
  truth = if (!is.null(task)) task$truth(row_ids),
  response = NULL,
  prob = NULL,
  se = NULL,
  lazy_tensor = NULL,
  weights = NULL,
  check = TRUE
)
```

Arguments:

`task` (`TaskTorch`)
: The task that was predicted on. Used to extract the default `row_ids` and the `truth`.

`row_ids` (`integer()`)
: The row ids of the predicted observations.

`truth` (any)
: The ground truth, i.e. what `task$truth()` returned.

`response` (any)
: The predicted response.

`prob` (any)
: The predicted probabilities.

`se` (any)
: The standard errors of the prediction.

`lazy_tensor` (`lazy_tensor` or `data.table` of them)
: The output of the network, see the predict type `"lazy_tensor"` of `LearnerTorch`. A network with more than one head produces one column per head.

`weights` (`numeric()` or `NULL`)
: The measure weights of the predicted observations, i.e. the `weights_measure` column of the task. `mlr3` fills this in, so it rarely has to be passed by hand.

`check` (`logical(1)`)
: Whether to check the consistency of the prediction data.

`PredictionTorch$clone()`: The objects of this class are cloneable with this method.

Usage:

```
PredictionTorch$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
d = data.frame(x = rnorm(10), y1 = rnorm(10), y2 = rnorm(10))
task = as_task_torch(d, target = c("y1", "y2"))
PredictionTorch$new(task, response = as.matrix(d[, c("y1", "y2")]))
```

`reshape_output_shape` *Output Shape of a Reshape*

Description

The shape of `torch_reshape(x, shape)`, which is what `nn_reshape()` infers. The dimension that `torch` infers from the number of elements is resolved here whenever that number is known, and stays unknown otherwise.

Usage

```
reshape_output_shape(shape_in, shape, id)
```

Arguments

`shape_in`	(`integer()`) The input shape, with `NA` for the dimensions whose size is unknown.
`shape`	(`integer()` | `function()`) The target shape, where `-1` marks the dimension that `torch` infers from the number of elements. A `function(shape)` of the input shape is called on it and must return such a vector, which lets a reshape be expressed for inputs whose sizes are not known in advance, e.g. `\(shape) c(shape[1:2], 10)`.
`id`	(`character(1)`) The id of the `PipeOp`, which the error messages name.

Value

(`integer()`) The output shape.

See Also

Other Shape Inference: `infer_shapes()`, `shape_to_str()`

Examples

```
reshape_output_shape(c(NA, 3, 4), c(-1, 12), id = "nn_reshape")
reshape_output_shape(c(NA, 3, 4), \(shape) c(shape[1], 12), id = "nn_reshape")
```

`Select` *Selector Functions for Character Vectors*

Description

A `Select` function subsets a character vector. They are used by the callback `CallbackSetUnfreeze` to select parameters to freeze or unfreeze during training.

Usage

```
select_all()

select_none()

select_grep(pattern, ignore.case = FALSE, perl = FALSE, fixed = FALSE)

select_name(param_names, assert_present = TRUE)

select_invert(select)
```

Arguments

`pattern`	See `grep()`
`ignore.case`	See `grep()`
`perl`	See `grep()`
`fixed`	See `grep()`
`param_names`	The names of the parameters that you want to select
`assert_present`	Whether to check that `param_names` is a subset of the full vector of names
`select`	A `Select`

Functions

- `select_all()`: `select_all` selects all elements
- `select_none()`: `select_none` selects no elements
- `select_grep()`: `select_grep` selects elements with names matching a regular expression
- `select_name()`: `select_name` selects elements with names matching the given names
- `select_invert()`: `select_invert` selects the elements NOT selected by the given selector

Examples

```
select_all()(c("a", "b"))
select_none()(c("a", "b"))
select_grep("b$")(c("ab", "ac"))
select_name("a")(c("a", "b"))
select_invert(select_all())(c("a", "b"))
```

shape_to_str *Helpers for Shape Inference*

Description

Helpers for writing the `private$.shapes_out()` method of a `PipeOpTorch`. They all propagate unknown (`NA`) dimensions, see the "Shape Inference" section of `PipeOpTorch` for the shape conventions.

Usage

```
shape_to_str(x)

broadcast_shapes(shapes, id)

resolve_dim(dim, shape, insert = FALSE)
```

Arguments

`x`	(`integer()` | `list()` of `integer()` | `NULL`) The shape(s) to format. `NULL` stands for an unknown shape.
`shapes`	(`list()` of `integer()`) The input shapes, all with the same number of dimensions.
`id`	(`character(1)`) The id of the `PipeOp`, which the error message names.
`dim`	(`integer()`) The dimension(s) the operator addresses. Negative ones count back from the last dimension: `-1` is the last dimension, `-2` the one before it, and so on.
`shape`	(`integer()`) The input shape.
`insert`	(`logical(1)`) Whether a dimension is inserted rather than addressed, as by `nn_unsqueeze()`: there is then one more position than the shape has dimensions, `-1` appending a new last one.

Details

- `broadcast_shapes()` applies the broadcasting rules of `torch`, generalized to shapes that may contain `NA`. Per dimension a known size that is not 1 wins; if all known sizes are 1 and some input is unknown, the result is unknown, because the unknown one may be greater than 1 and would then determine the size. The shapes must already have the same number of dimensions: shorter ones are not left-padded with 1s, because the first dimension is the batch dimension.
- `resolve_dim()` resolves dimension indices that count from the end, as in `torch`, to positive ones. Indices that are out of range stay out of range, so that `assert_dim_in_range()` reports them.
- `shape_to_str()` formats a shape, or a `list()` of them, for an error message.

Value

broadcast_shapes() returns an integer() shape, resolve_dim() an integer() of the same length as dim, and shape_to_str() a character(1).

See Also

Other Shape Inference: infer_shapes(), reshape_output_shape()

Examples

```
broadcast_shapes(list(c(NA, 1), c(NA, 6)), id = "nn_merge_sum")
resolve_dim(-1, c(NA, 3, 8))
shape_to_str(c(NA, 3, 8))
```

---

t_clbk *Sugar Function for Torch Callback*

Description

Retrieves one or more TorchCallbacks from mlr3torch_callbacks. Works like mlr3::lrn() and mlr3::lrns().

Usage

```
t_clbk(.key, ...)

t_clbks(.keys)
```

Arguments

.key	(character(1)) The key of the torch callback.
...	(any) See description of dictionary_sugar_get().
.keys	(character()) The keys of the callbacks.

Value

TorchCallback

list() of TorchCallbacks

See Also

Other Callback: TorchCallback, as_torch_callback(), as_torch_callbacks(), callback_set(), mlr3torch_callbacks, mlr_callback_set, mlr_callback_set.checkpoint, mlr_callback_set.progress, mlr_callback_set.tb, mlr_callback_set.unfreeze, mlr_context_torch, torch_callback()

Other Torch Descriptor: TorchCallback, TorchDescriptor, TorchLoss, TorchOptimizer, as_torch_callbacks(), as_torch_loss(), as_torch_optimizer(), mlr3torch_losses, mlr3torch_optimizers, t_loss(), t_opt()

Examples

```
t_clbk("progress")
```

---

t_loss *Loss Function Quick Access*

Description

Retrieve one or more TorchLosses from mlr3torch_losses. Works like mlr3::lrn() and mlr3::lrns().

Usage

```
t_loss(.key, ...)

t_losses(.keys, ...)
```

Arguments

.key	(character(1)) Key of the object to retrieve.
...	(any) See description of dictionary_sugar_get.
.keys	(character()) The keys of the losses.

Value

A TorchLoss

See Also

Other Torch Descriptor: TorchCallback, TorchDescriptor, TorchLoss, TorchOptimizer, as_torch_callbacks(), as_torch_loss(), as_torch_optimizer(), mlr3torch_losses, mlr3torch_optimizers, t_clbk(), t_opt()

Examples

```
t_loss("mse", reduction = "mean")
# get the dictionary
t_loss()


t_losses(c("mse", "l1"))
# get the dictionary
t_losses()
```

---

t_opt *Optimizers Quick Access*

Description

Retrieves one or more TorchOptimizers from mlr3torch_optimizers. Works like mlr3::lrn() and mlr3::lrns().

Usage

```
t_opt(.key, ...)

t_opts(.keys, ...)
```

Arguments

.key	(character(1)) Key of the object to retrieve.
...	(any) See description of dictionary_sugar_get.
.keys	(character()) The keys of the optimizers.

Value

A TorchOptimizer

See Also

Other Torch Descriptor: TorchCallback, TorchDescriptor, TorchLoss, TorchOptimizer, as_torch_callbacks(), as_torch_loss(), as_torch_optimizer(), mlr3torch_losses, mlr3torch_optimizers, t_clbk(), t_loss()

Other Dictionary: mlr3torch_callbacks, mlr3torch_losses, mlr3torch_optimizers

Examples

```
t_opt("adam", lr = 0.1)
# get the dictionary
t_opt()


t_opts(c("adam", "sgd"))
# get the dictionary
t_opts()
```

`task_dataset` *Create a Dataset from a Task*

Description

Creates a torch dataset from an mlr3 `Task`. The resulting dataset's `$.getbatch()` method returns a list with elements `x`, `y` and `.index`:

- `x` is a list with tensors, whose content is defined by the parameter `feature_ingress_tokens`.
- `y` is the target variable and its content is defined by the parameter `target_batchgetter`.
- `.index` is the index of the batch in the task's data.

The data is returned on the device specified by the parameter `device`.

Usage

```
task_dataset(
  task,
  feature_ingress_tokens,
  target_batchgetter = NULL,
  cache_data = TRUE
)
```

Arguments

`task`	(`Task`) The task for which to build the dataset.
`feature_ingress_tokens`	(named `list()` of `TorchIngressToken`) Each ingress token defines one item in the `$x` value of a batch with corresponding names.
`target_batchgetter`	(`function()` or `NULL`) Converts the target columns of a batch into the target tensor `y` that the loss is applied to. Takes an argument `data`, a `data.table` with only the target columns, and optionally an argument `x`, the named list of feature tensors of the batch, which is what a target that is a function of the input needs, see `get_target_batchgetter()`. If `NULL` (default), the batches have no `y` element.

cache_data (logical(1))
Whether to retrieve the data from the task once (in $initialize()) and to take the batches from it, instead of retrieving every batch from the task, which is slow. This needs memory for a copy of the task data (lazy tensor columns only contain indices and are small).

Value

torch::dataset

Examples

```
task = tsk("iris")
sepal_ingress = TorchIngressToken(
  features = c("Sepal.Length", "Sepal.Width"),
  batchgetter = batchgetter_num,
  shape = c(NA, 2)
)
petal_ingress = TorchIngressToken(
  features = c("Petal.Length", "Petal.Width"),
  batchgetter = batchgetter_num,
  shape = c(NA, 2)
)
ingress_tokens = list(sepal = sepal_ingress, petal = petal_ingress)

target_batchgetter = function(data) {
  torch_tensor(data = data[[1L]], dtype = torch_float32())$unsqueeze(2)
}
dataset = task_dataset(task, ingress_tokens, target_batchgetter)
batch = dataset$.getbatch(1:10)
batch
```

---

torch_callback *Create a Callback Descriptor*

Description

Convenience function to create a custom TorchCallback. All arguments that are available in callback_set() are also available here. For more information on how to correctly implement a new callback, see CallbackSet.

Usage

```
torch_callback(
  id,
  classname = paste0("CallbackSet", capitalize(id)),
  param_set = NULL,
  packages = NULL,
```

```
  label = capitalize(id),
  man = NULL,
  on_begin = NULL,
  on_end = NULL,
  on_exit = NULL,
  on_epoch_begin = NULL,
  on_before_valid = NULL,
  on_epoch_end = NULL,
  on_batch_begin = NULL,
  on_batch_end = NULL,
  on_after_backward = NULL,
  on_batch_valid_begin = NULL,
  on_batch_valid_end = NULL,
  on_valid_end = NULL,
  state_dict = NULL,
  load_state_dict = NULL,
  initialize = NULL,
  weight = NULL,
  public = NULL,
  private = NULL,
  active = NULL,
  parent_env = parent.frame(),
  inherit = CallbackSet,
  lock_objects = FALSE
)
```

Arguments

id	(character(1)) ‘ The id for the torch callback.
classname	(character(1)) The class name.
param_set	(ParamSet) The parameter set, if not present it is inferred from the $initialize() method.
packages	(character()) The packages the callback depends on. Default isNULL‘.
label	(character(1)) The label for the torch callback. Defaults to the capitalized id.
man	(character(1)) String in the format [pkg]::[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help(). The default is NULL.
on_begin, on_end, on_epoch_begin, on_before_valid, on_epoch_end, on_batch_begin, on_batch_end, on_after_backward, on_batch_valid_begin, on_batch_valid_end, on_valid_end, on_exit	(function) Function to execute at the given stage, see section *Stages*.

`state_dict` (`function()`)
The function that retrieves the state dict from the callback. This is what will be available in the learner after training.

`load_state_dict` (`function(state_dict)`)
Function that loads a callback state.

`initialize` (`function()`)
The initialization method of the callback.

`weight` (`numeric(1)`)
Controls when the callback is called within a stage, see section *Ordering* of `CallbackSet`. Defaults to `0`. The name is reserved for this, so `initialize` must not take a `weight` argument: wrapping such a callback in a `TorchCallback` is an error.

`public`, `private`, `active` (`list()`)
Additional public, private, and active fields to add to the callback.

`parent_env` (`environment()`)
The parent environment for the `R6Class`.

`inherit` (`R6ClassGenerator`)
From which class to inherit. This class must either be `CallbackSet` (default) or inherit from it.

`lock_objects` (`logical(1)`)
Whether to lock the objects of the resulting `R6Class`. If `FALSE` (default), values can be freely assigned to `self` without declaring them in the class definition.

Value

`TorchCallback`

Internals

It first creates an `R6` class inheriting from `CallbackSet` (using `callback_set()`) and then wraps this generator in a `TorchCallback` that can be passed to a torch learner.

Stages

- `begin` :: Run before the training loop begins.
- `epoch_begin` :: Run at the beginning of each epoch.
- `batch_begin` :: Run before the forward call.
- `after_backward` :: Run after the backward call.
- `batch_end` :: Run after the optimizer step.
- `before_valid` :: Run before the validation loop begins.
- `batch_valid_begin` :: Run before the forward call in the validation loop.
- `batch_valid_end` :: Run after the forward call in the validation loop.
- `valid_end` :: Run at the end of validation.

- epoch_end :: Run at the end of each epoch.
- end :: Run after last epoch.
- exit :: Run last, using on.exit().

Ordering

Within a stage, callbacks are called in the order in which they were passed to the learner. A callback can override this via its $weight field: callbacks with a higher weight are called after those with a lower one, and callbacks with the same weight keep the order in which they were passed. The default weight is 0. This matters for callbacks that observe what the others did, which is why CallbackSetCheckpoint has weight Inf: it always runs last and therefore saves the network and optimizer as the other callbacks left them at the end of the stage.

See Also

Other Callback: TorchCallback, as_torch_callback(), as_torch_callbacks(), callback_set(), mlr3torch_callbacks, mlr_callback_set, mlr_callback_set.checkpoint, mlr_callback_set.progress, mlr_callback_set.tb, mlr_callback_set.unfreeze, mlr_context_torch, t_clbk()

Examples

```
custom_tcb = torch_callback("custom",
  initialize = function(name) {
    self$name = name
  },
  on_begin = function() {
    cat("Hello", self$name, ", we will train for ", self$ctx$total_epochs, "epochs.\n")
  },
  on_end = function() {
    cat("Training is done.")
  }
)

learner = lrn("classif.torch_featureless",
  batch_size = 16,
  epochs = 1,
  callbacks = custom_tcb,
  cb.custom.name = "Marie",
  device = "cpu"
)
task = tsk("iris")
learner$train(task)
```

TorchCallback *Torch Callback*

Description

This wraps a CallbackSet and annotates it with metadata, most importantly a ParamSet. The callback is created for the given parameter values by calling the $generate() method.

This class is usually used to configure the callback of a torch learner, e.g. when constructing a learner of in a ModelDescriptor.

For a list of available callbacks, see mlr3torch_callbacks. To conveniently retrieve a TorchCallback, use t_clbk().

Parameters

Defined by the constructor argument param_set. If no parameter set is provided during construction, the parameter set is constructed by creating a parameter for each argument of the wrapped callback, where the parameters are then of type ParamUty.

Super class

TorchDescriptor -> TorchCallback

Active bindings

weight (numeric(1) or NULL)
: Overwrites the $weight of the generated CallbackSet, see its section *Ordering*.

Methods

Public methods:

- TorchCallback$new()
- TorchCallback$generate()
- TorchCallback$clone()

TorchCallback$new(): Creates a new instance of this R6 class.

Usage:

```
TorchCallback$new(
  callback_generator,
  param_set = NULL,
  id = NULL,
  label = NULL,
  packages = NULL,
  man = NULL,
  additional_args = NULL,
  weight = NULL
)
```

Arguments:

callback_generator (R6ClassGenerator)
: The class generator for the callback that is being wrapped.

param_set (ParamSet or NULL)
: The parameter set. If NULL (default) it is inferred from callback_generator. Whether inferred or given, it must not contain a weight: that is the callback's ordering weight, which is set via the weight argument.

id (character(1))
: The id for of the new object.

label (character(1))
: Label for the new instance.

packages (character())
: The R packages this object depends on.

man (character(1))
: String in the format [pkg]::[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().

additional_args (any)
: Additional arguments if necessary. For learning rate schedulers, this is the torch::LRScheduler.

weight (numeric(1) or NULL)
: Overwrites the $weight of the generated CallbackSet, see its section *Ordering*. If NULL (default), the callback's own weight is kept.

TorchCallback$generate(): Generates the CallbackSet, applying $weight if it is set.

Usage:

```
TorchCallback$generate()
```

TorchCallback$clone(): The objects of this class are cloneable with this method.

Usage:

```
TorchCallback$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Callback: as_torch_callback(), as_torch_callbacks(), callback_set(), mlr3torch_callbacks, mlr_callback_set, mlr_callback_set.checkpoint, mlr_callback_set.progress, mlr_callback_set.tb, mlr_callback_set.unfreeze, mlr_context_torch, t_clbk(), torch_callback()

Other Torch Descriptor: TorchDescriptor, TorchLoss, TorchOptimizer, as_torch_callbacks(), as_torch_loss(), as_torch_optimizer(), mlr3torch_losses, mlr3torch_optimizers, t_clbk(), t_loss(), t_opt()

Examples

```
# Create a new torch callback from an existing callback set
torch_callback = TorchCallback$new(CallbackSetCheckpoint)
# The parameters are inferred
torch_callback$param_set

# Retrieve a torch callback from the dictionary
torch_callback = t_clbk("checkpoint",
```

```
  path = tempfile(), freq = 1
)
torch_callback
torch_callback$label
torch_callback$id

# open the help page of the wrapped callback set
# torch_callback$help()

# Create the callback set
callback = torch_callback$generate()
callback
# is the same as
CallbackSetCheckpoint$new(
  path = tempfile(), freq = 1
)

# Use in a learner
learner = lrn("regr.mlp", callbacks = t_clbk("checkpoint"))
# the parameters of the callback are added to the learner's parameter set
learner$param_set
```

`TorchDescriptor` *Base Class for Torch Descriptors*

Description

Abstract Base Class from which `TorchLoss`, `TorchOptimizer`, and `TorchCallback` inherit. This class wraps a generator (R6Class Generator or the torch version of such a generator) and annotates it with metadata such as a `ParamSet`, a label, an ID, packages, or a manual page.

The parameters are the construction arguments of the wrapped generator and the parameter `$values` are passed to the generator when calling the public method `$generate()`.

Parameters

Defined by the constructor argument `param_set`. All parameters are tagged with `"train"`, but this is done automatically during initialize.

Public fields

`label (character(1))`
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`param_set (ParamSet)`
: Set of hyperparameters.

`packages (character(1))`
: Set of required packages. These packages are loaded, but not attached.

`id` (`character(1)`)
: Identifier of the object. Used in tables, plot and text output.

`generator` The wrapped generator that is described.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object.

Active bindings

`phash` (`character(1)`)
: Hash (unique identifier) for this partial object, excluding some components which are varied systematically (e.g. the parameter values).

Methods

Public methods:

- `TorchDescriptor$new()`
- `TorchDescriptor$print()`
- `TorchDescriptor$generate()`
- `TorchDescriptor$help()`
- `TorchDescriptor$clone()`

`TorchDescriptor$new()`: Creates a new instance of this R6 class.

Usage:

```
TorchDescriptor$new(
  generator,
  id = NULL,
  param_set = NULL,
  packages = NULL,
  label = NULL,
  man = NULL,
  additional_args = NULL
)
```

Arguments:

`generator` The wrapped generator that is described.

`id` (`character(1)`)
: The id for of the new object.

`param_set` (`ParamSet`)
: The parameter set.

`packages` (`character()`)
: The R packages this object depends on.

`label` (`character(1)`)
: Label for the new instance.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

additional_args (list())
: Additional arguments if necessary. For learning rate schedulers, this is the torch::LRScheduler.

`TorchDescriptor$print()`: Prints the object

Usage:

```
TorchDescriptor$print(...)
```

Arguments:

`...` any

`TorchDescriptor$generate()`: Calls the generator with the given parameter values.

Usage:

```
TorchDescriptor$generate()
```

`TorchDescriptor$help()`: Displays the help file of the wrapped object.

Usage:

```
TorchDescriptor$help()
```

`TorchDescriptor$clone()`: The objects of this class are cloneable with this method.

Usage:

```
TorchDescriptor$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Torch Descriptor: `TorchCallback`, `TorchLoss`, `TorchOptimizer`, `as_torch_callbacks()`, `as_torch_loss()`, `as_torch_optimizer()`, `mlr3torch_losses`, `mlr3torch_optimizers`, `t_clbk()`, `t_loss()`, `t_opt()`

TorchIngressToken *Torch Ingress Token*

Description

This function creates an S3 class of class `"TorchIngressToken"`, which is an internal data structure. It contains the (meta-)information of how a batch is generated from a `Task` and fed into an entry point of the neural network. It is stored as the `ingress` field in a `ModelDescriptor`.

Usage

```
TorchIngressToken(features, batchgetter, shape = NULL)
```

Arguments

features	(character or mlr3pipelines::Selector) Features on which the batchgetter will operate or a selector (such as mlr3pipelines::selector_type).
batchgetter	(function) Function with two arguments: data and device. This function is given the output of Task$data(rows = batch_indices, cols = features) and it should produce a tensor of shape shape_out.
shape	(integer) Shape that batchgetter will produce. Batch dimension must be included as NA (but other dimensions can also be NA, i.e., unknown).

Value

TorchIngressToken object.

See Also

Other Graph Network: ModelDescriptor(), as_learner_torch(), mlr_learners_torch_model, mlr_pipeops_module, mlr_pipeops_torch, mlr_pipeops_torch_ingress, mlr_pipeops_torch_ingress_categ, mlr_pipeops_torch_ingress_ltnsr, mlr_pipeops_torch_ingress_num, model_descriptor_to_learner(), model_descriptor_to_module(), model_descriptor_union(), nn_graph(), pipeop_torch()

Examples

```
# Define a task for which we want to define an ingress token
task = tsk("iris")

# We create an ingress token for two feature Sepal.Length and Petal.Length:
# We have to specify the features, the batchgetter and the shape
features = c("Sepal.Length", "Petal.Length")
# As a batchgetter we use batchgetter_num

batch_dt = task$data(rows = 1:10, cols =features)
batch_dt
batch_tensor = batchgetter_num(batch_dt, "cpu")
batch_tensor

# The shape is unknown in the first dimension (batch dimension)

ingress_token = TorchIngressToken(
  features = features,
  batchgetter = batchgetter_num,
  shape = c(NA, 2)
)
ingress_token
```

TorchLoss *Torch Loss*

Description

This wraps a `torch::nn_loss` and annotates it with metadata, most importantly a `ParamSet`. The loss function is created for the given parameter values by calling the `$generate()` method.

This class is usually used to configure the loss function of a torch learner, e.g. when construcing a learner or in a `ModelDescriptor`.

For a list of available losses, see `mlr3torch_losses`. Items from this dictionary can be retrieved using `t_loss()`.

Parameters

Defined by the constructor argument `param_set`. If no parameter set is provided during construction, the parameter set is constructed by creating a parameter for each argument of the wrapped loss function, where the parameters are then of type `ParamUty`.

Super class

`TorchDescriptor` -> `TorchLoss`

Public fields

`task_types (character())`
The task types this loss supports.

Methods

Public methods:

- `TorchLoss$new()`
- `TorchLoss$print()`
- `TorchLoss$generate()`
- `TorchLoss$clone()`

`TorchLoss$new()`: Creates a new instance of this R6 class.

Usage:

```
TorchLoss$new(
  torch_loss,
  task_types = NULL,
  param_set = NULL,
  id = NULL,
  label = NULL,
  packages = NULL,
  man = NULL
)
```

Arguments:

`torch_loss` (`nn_loss` or `function`)
The loss module or function that generates the loss module. Can have arguments `task` that will be provided when the loss is instantiated.

`task_types` (`character()`)
The task types supported by this loss. If `NULL` (default), the loss is applicable to `"classif"`, `"regr"` and `"torch"`.

`param_set` (`ParamSet` or `NULL`)
The parameter set. If `NULL` (default) it is inferred from `torch_loss`.

`id` (`character(1)`)
The id for of the new object.

`label` (`character(1)`)
Label for the new instance.

`packages` (`character()`)
The R packages this object depends on.

`man` (`character(1)`)
String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

`TorchLoss$print()`: Prints the object

Usage:

```
TorchLoss$print(...)
```

Arguments:

`...` any

`TorchLoss$generate()`: Instantiates the loss function.

Usage:

```
TorchLoss$generate(task = NULL)
```

Arguments:

`task` (`Task`)
The task. Must be provided if the loss function requires a task.

Returns: `torch_loss`

`TorchLoss$clone()`: The objects of this class are cloneable with this method.

Usage:

```
TorchLoss$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Torch Descriptor: `TorchCallback`, `TorchDescriptor`, `TorchOptimizer`, `as_torch_callbacks()`, `as_torch_loss()`, `as_torch_optimizer()`, `mlr3torch_losses`, `mlr3torch_optimizers`, `t_clbk()`, `t_loss()`, `t_opt()`

Examples

```
# Create a new torch loss
torch_loss = TorchLoss$new(torch_loss = nn_mse_loss, task_types = "regr")
torch_loss
# the parameters are inferred
torch_loss$param_set

# Retrieve a loss from the dictionary:
torch_loss = t_loss("mse", reduction = "mean")
# is the same as
torch_loss
torch_loss$param_set
torch_loss$label
torch_loss$task_types
torch_loss$id

# Create the loss function
loss_fn = torch_loss$generate()
loss_fn
# Is the same as
nn_mse_loss(reduction = "mean")

# open the help page of the wrapped loss function
# torch_loss$help()

# Use in a learner
learner = lrn("regr.mlp", loss = t_loss("mse"))
# The parameters of the loss are added to the learner's parameter set
learner$param_set
```

TorchOptimizer *Torch Optimizer*

Description

This wraps a `torch::torch_optimizer_generator` and annotates it with metadata, most importantly a `ParamSet`. The optimizer is created for the given parameter values by calling the `$generate()` method.

This class is usually used to configure the optimizer of a torch learner, e.g. when constructing a learner or in a `ModelDescriptor`.

For a list of available optimizers, see `mlr3torch_optimizers`. Items from this dictionary can be retrieved using `t_opt()`.

Parameters

Defined by the constructor argument `param_set`. If no parameter set is provided during construction, the parameter set is constructed by creating a parameter for each argument of the wrapped optimizer, where the parameters are then of type `ParamUty`.

In addition, every TorchOptimizer has the parameter:

- param_groups :: function(params) -> list()
  A function that receives the named list() of network parameters and returns a list() of parameter groups, in the format expected by the wrapped torch optimizer. This makes it possible to optimize different parts of the network differently, e.g. to give the first layer a different learning rate than the rest:

  ```
  param_groups = function(params) list(
    list(params = params[startsWith(names(params), "0.")], lr = 0.9),
    list(params = params[!startsWith(names(params), "0.")], lr = 0.001)
  )
  ```

  When a LearnerTorch is configured, this is reachable as opt.param_groups. Default is NULL, i.e. all parameters form a single group.

Super class

TorchDescriptor -> TorchOptimizer

Methods

Public methods:

- TorchOptimizer$new()
- TorchOptimizer$generate()
- TorchOptimizer$clone()

TorchOptimizer$new(): Creates a new instance of this R6 class.

Usage:

```
TorchOptimizer$new(
  torch_optimizer,
  param_set = NULL,
  id = NULL,
  label = NULL,
  packages = NULL,
  man = NULL
)
```

Arguments:

torch_optimizer (torch_optimizer_generator)
: The torch optimizer.

param_set (ParamSet or NULL)
: The parameter set. If NULL (default) it is inferred from torch_optimizer.

id (character(1))
: The id for of the new object.

label (character(1))
: Label for the new instance.

packages (character())
: The R packages this object depends on.

man (character(1))
String in the format [pkg]::[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().

TorchOptimizer$generate(): Instantiates the optimizer.

Usage:

```
TorchOptimizer$generate(params)
```

Arguments:

params (named list() of torch_tensors)
The parameters of the network.

Returns: torch_optimizer

TorchOptimizer$clone(): The objects of this class are cloneable with this method.

Usage:

```
TorchOptimizer$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Torch Descriptor: TorchCallback, TorchDescriptor, TorchLoss, as_torch_callbacks(), as_torch_loss(), as_torch_optimizer(), mlr3torch_losses, mlr3torch_optimizers, t_clbk(), t_loss(), t_opt()

Examples

```
# Create a new torch optimizer
torch_opt = TorchOptimizer$new(optim_ignite_adam, label = "adam")
torch_opt
# If the param set is not specified, parameters are inferred but are of class ParamUty
torch_opt$param_set

# open the help page of the wrapped optimizer
# torch_opt$help()

# Retrieve an optimizer from the dictionary
torch_opt = t_opt("sgd", lr = 0.1)
torch_opt
torch_opt$param_set
torch_opt$label
torch_opt$id

# Create the optimizer for a network
net = nn_linear(10, 1)
opt = torch_opt$generate(net$parameters)

# is the same as
optim_sgd(net$parameters, lr = 0.1)
```

```
# Use in a learner
learner = lrn("regr.mlp", optimizer = t_opt("sgd"))
# The parameters of the optimizer are added to the learner's parameter set
learner$param_set
```

Index

∗ **Callback**
as_torch_callback, 13
as_torch_callbacks, 14
callback_set, 27
mlr3torch_callbacks, 284
mlr_callback_set, 46
mlr_callback_set.checkpoint, 49
mlr_callback_set.progress, 57
mlr_callback_set.tb, 59
mlr_callback_set.unfreeze, 61
mlr_context_torch, 63
t_clbk, 311
torch_callback, 315
TorchCallback, 318
∗ **Dictionary**
mlr3torch_callbacks, 284
mlr3torch_losses, 285
mlr3torch_optimizers, 285
t_opt, 313
∗ **Graph Network**
as_learner_torch, 10
mlr_learners_torch_model, 104
mlr_pipeops_module, 119
mlr_pipeops_torch, 244
mlr_pipeops_torch_ingress, 251
mlr_pipeops_torch_ingress_categ, 253
mlr_pipeops_torch_ingress_ltnsr, 255
mlr_pipeops_torch_ingress_num, 257
model_descriptor_to_learner, 286
model_descriptor_to_module, 287
model_descriptor_union, 288
ModelDescriptor, 289
nn_graph, 296
pipeop_torch, 303
TorchIngressToken, 323
∗ **Learner**
as_learner_torch, 10
GraphLearnerTorch, 35
mlr_learners.ft_transformer, 66
mlr_learners.mlp, 68
mlr_learners.module, 71
mlr_learners.realmlp, 74
mlr_learners.tab_resnet, 78
mlr_learners.tabm, 80
mlr_learners.torch_featureless, 84
mlr_learners_torch, 90
mlr_learners_torch_image, 102
mlr_learners_torch_model, 104
∗ **Measure**
mlr_measures_torch, 107
mlr_measures_torch.default, 108
∗ **Model Configuration**
mlr_pipeops_torch_callbacks, 250
mlr_pipeops_torch_loss, 259
mlr_pipeops_torch_optimizer, 268
model_descriptor_union, 288
ModelDescriptor, 289
∗ **PipeOps**
mlr_pipeops_nn_recurrent, 203
mlr_pipeops_torch_model, 260
∗ **Prediction**
PredictionTorch, 306
∗ **Shape Assertions**
assert_dim_in_range, 17
assert_known_dims, 18
assert_ndim, 19
assert_not_batch_dim, 20
assert_positive_extent, 21
assert_same_batch_size, 22
assert_same_ndim, 23
assert_shape, 23
assert_shapes, 24
∗ **Shape Inference**
infer_shapes, 37
reshape_output_shape, 308
shape_to_str, 310

∗ **Task**
mlr_tasks_torch, 282
∗ **Torch Descriptor**
as_torch_callbacks, 14
as_torch_loss, 14
as_torch_optimizer, 16
mlr3torch_losses, 285
mlr3torch_optimizers, 285
t_clbk, 311
t_loss, 312
t_opt, 313
TorchCallback, 318
TorchDescriptor, 321
TorchLoss, 325
TorchOptimizer, 327

as.data.table, *284, 285*
as_data_descriptor, 8
as_data_descriptor(), *29*
as_lazy_tensor, 9
as_learner(), *10*
as_learner_torch, 10
as_learner_torch(), *35*, *37*, *68*, *70*, *73*, *77*, *80*, *83*, *85*, *102*, *104*, *106*, *121*, *248*, *253, 254*, *256*, *258*, *286–288*, *290*, *297*, *305*, *324*
as_lr_scheduler, 12
as_task_torch, 12
as_torch_callback, 13
as_torch_callback(), *14*, *28*, *49*, *51*, *59, 60*, *62*, *66*, *284*, *312*, *318*, *320*
as_torch_callbacks, 14
as_torch_callbacks(), *13*, *15, 16*, *28*, *49*, *51*, *59, 60*, *62*, *66*, *251*, *284–286*, *312, 313*, *318*, *320*, *323*, *326*, *329*
as_torch_loss, 14
as_torch_loss(), *14*, *16*, *259*, *285, 286*, *312, 313*, *320*, *323*, *326*, *329*
as_torch_optimizer, 16
as_torch_optimizer(), *14, 15*, *268*, *285, 286*, *312, 313*, *320*, *323*, *326*, *329*
assert_dim_in_range, 17
assert_dim_in_range(), *18*, *20–25*, *310*
assert_known_dims, 18
assert_known_dims(), *17*, *20–25*, *239*, *245*
assert_lazy_tensor, 19
assert_ndim, 19
assert_ndim(), *17, 18*, *20–25*
assert_not_batch_dim, 20
assert_not_batch_dim(), *17, 18*, *20–25*
assert_positive_extent, 21
assert_positive_extent(), *17, 18*, *20*, *22–25*
assert_same_batch_size, 22
assert_same_batch_size(), *17, 18*, *20, 21*, *23–25*
assert_same_ndim, 23
assert_same_ndim(), *17, 18*, *20–22*, *24, 25*
assert_shape, 23
assert_shape(), *17, 18*, *20–25*
assert_shapes, 24
assert_shapes(), *17, 18*, *20–24*
auto_device, 25
AutoTuner, *56*

batchgetter_categ, 26
batchgetter_categ(), *253*
batchgetter_num, 26
batchgetter_num(), *257*
broadcast_shapes *(*shape_to_str*)*, 310

callback_set, 27
callback_set(), *13, 14*, *47*, *49*, *51*, *59, 60*, *62*, *66*, *284*, *312*, *315*, *317, 318*, *320*
callbacks, *94*
CallbackSet, *27, 28*, *46*, *50*, *52*, *54–57*, *60, 61*, *63, 64*, *101*, *103*, *105*, *290*, *315*, *317*, *319, 320*
CallbackSet *(*mlr_callback_set*)*, 46
CallbackSetCheckpoint, *47*, *64*, *318*
CallbackSetCheckpoint *(*mlr_callback_set.checkpoint*)*, 49
CallbackSetHistory *(*mlr_callback_set.history*)*, 51
CallbackSetLRScheduler, *55, 56*, *66*
CallbackSetLRScheduler *(*mlr_callback_set.lr_scheduler*)*, 53
CallbackSetLRSchedulerOneCycle *(*mlr_callback_set.lr_scheduler_one_cycle*)*, 55
CallbackSetLRSchedulerReduceOnPlateau *(*mlr_callback_set.lr_scheduler_reduce_on_plateau*)*, 56
CallbackSetProgress *(*mlr_callback_set.progress*)*, 57
CallbackSetTB *(*mlr_callback_set.tb*)*, 59

CallbackSetUnfreeze
(mlr_callback_set.unfreeze), 61
classif.inception_v3, 97
ContextTorch, 47, 48, 91, 97
ContextTorch (mlr_context_torch), 63
cross_entropy, 28

data.table, 33, 35, 73, 92, 105, 264, 283–285, 306, 307, 314
data.table::data.table(), 44, 45
DataBackend, 12, 280, 281, 283
DataBackendLazy, 277, 279–281
DataBackendLazy (mlr_backends_lazy), 43
DataDescriptor, 8, 29, 41, 42
dataset, 9, 36, 98, 102, 314
datasets::iris, 277
dictionary_sugar_get, 312, 313
dictionary_sugar_get(), 311

encode_prediction, 32
encode_prediction(), 91, 97

feature union, 244, 288

get_batch_constructor, 33
get_batch_size, 34
get_target_batchgetter, 35
get_target_batchgetter(), 33, 73, 97, 105, 264, 283, 314
Graph, 10, 11, 30, 31, 42, 71, 141, 142, 199, 201, 233, 235, 244, 246, 289, 292, 296
graph, 246
GraphLearner, 11, 35
GraphLearnerTorch, 10, 11, 35, 68, 70, 73, 77, 80, 83, 85, 102, 104, 106

hashtab, 33
hashtab(), 42

infer_shapes, 37, 303
infer_shapes(), 308, 311
ingress_categ, 38, 98
ingress_categ(), 72
ingress_ltnsr, 39, 98
ingress_ltnsr(), 72
ingress_num, 39, 98
ingress_num(), 72
is_lazy_tensor, 40

lazy_shape, 40
lazy_tensor, 9, 29, 33, 40, 41, 42, 43, 66–68, 92, 95, 102, 119, 231, 239, 255, 261, 306, 307
lazy_tensor(), 41
Learner, 63, 65, 66, 69, 71, 74, 78, 80, 84, 286
LearnerTorch, 10, 32, 34–36, 47–49, 63–65, 67, 69, 71, 72, 74–76, 78, 79, 81, 82, 84, 88, 102, 104, 105, 265, 266, 283, 301, 307, 328
LearnerTorch (mlr_learners_torch), 90
LearnerTorchFeatureless
(mlr_learners.torch_featureless), 84
LearnerTorchFTTransformer, 164, 165, 293
LearnerTorchFTTransformer
(mlr_learners.ft_transformer), 66
LearnerTorchImage, 87, 88
LearnerTorchImage
(mlr_learners_torch_image), 102
LearnerTorchMLP (mlr_learners.mlp), 68
LearnerTorchModel, 32, 263, 265–267
LearnerTorchModel
(mlr_learners_torch_model), 104
LearnerTorchModule
(mlr_learners.module), 71
LearnerTorchRealMLP
(mlr_learners.realmlp), 74
LearnerTorchTabM (mlr_learners.tabm), 80
LearnerTorchTabResNet
(mlr_learners.tab_resnet), 78
LearnerTorchVision
(mlr_learners.torchvision), 86
loss, 94
lrn(), 66, 69, 71, 74, 78, 80, 84
lrn(torch.module), 283

materialize, 41
materialize(), 29, 41
Measure, 63, 65, 95, 107, 108, 261, 282, 283, 291
MeasureTorch, 290, 291
MeasureTorch (mlr_measures_torch), 107
MeasureTorchDefault
(mlr_measures_torch.default), 108
mirai, 93
mlr3::as_prediction_data(), 32, 98

mlr3::col_info(), *43*
mlr3::DataBackend, *43*
mlr3::Learner, *36*, *67*, *69*, *71*, *76*, *79*, *82*, *84*, *88*, *99*, *102*, *104*
mlr3::lrn(), *311–313*
mlr3::lrns(), *311–313*
mlr3::Measure, *107, 108*
mlr3::Prediction, *306*
mlr3::Task, *90*, *239*, *282*
mlr3::TaskClassif, *277*
mlr3misc::Dictionary, *284*
mlr3pipelines::Graph, *29*, *119*
mlr3pipelines::GraphLearner, *36*
mlr3pipelines::PipeOp, *119, 120*, *123, 124*, *126, 127*, *129, 130*, *132, 133*, *135*, *137, 138*, *140, 141*, *143*, *145*, *147*, *149, 150*, *152*, *154, 155*, *157, 158*, *160–162*, *164*, *166–168*, *170, 171*, *173–176*, *178, 179*, *181, 182*, *184–186*, *188, 189*, *191*, *193–195*, *197*, *199*, *201, 202*, *204–206*, *208, 209*, *211, 212*, *214–217*, *219–221*, *223–225*, *227–229*, *231, 232*, *235*, *237, 238*, *240*, *247*, *250*, *252*, *254, 255*, *258, 259*, *264, 265*, *267, 268*
mlr3pipelines::PipeOpLearner, *264, 265*, *267*
mlr3pipelines::PipeOpTaskPreproc, *240*
mlr3pipelines::selector_type, *324*
mlr3torch *(*mlr3torch-package*)*, 7
mlr3torch-package, 7
mlr3torch_callbacks, *13, 14*, *28*, *49*, *51*, *59, 60*, *62*, *66*, 284, *285, 286*, *311–313*, *318–320*
mlr3torch_losses, *14–16*, *97*, *284*, 285, *286*, *312, 313*, *320*, *323*, *325, 326*, *329*
mlr3torch_optimizers, *14–16*, *284*, 285, *285*, *312, 313*, *320*, *323*, *326, 327*, *329*
mlr_backends_lazy, 43
mlr_callback_set, *13, 14*, *28*, 46, *51*, *59, 60*, *62*, *66*, *284*, *312*, *318*, *320*
mlr_callback_set.checkpoint, *13, 14*, *28*, 49, *49*, *59, 60*, *62*, *66*, *284*, *312*, *318*, *320*
mlr_callback_set.history, 51
mlr_callback_set.lr_scheduler, 53
mlr_callback_set.lr_scheduler_one_cycle, 55
mlr_callback_set.lr_scheduler_reduce_on_plateau, 56
mlr_callback_set.progress, *13, 14*, *28*, *49*, *51*, 57, *60*, *62*, *66*, *284*, *312*, *318*, *320*
mlr_callback_set.tb, *13, 14*, *28*, *49*, *51*, 59, *59*, *62*, *66*, *284*, *312*, *318*, *320*
mlr_callback_set.unfreeze, *13, 14*, *28*, *49*, *51*, *59, 60*, 61, *66*, *284*, *312*, *318*, *320*
mlr_context_torch, *13, 14*, *28*, *49*, *51*, *59, 60*, *62*, 63, *284*, *312*, *318*, *320*
mlr_learners, *35*
mlr_learners.ft_transformer, *11*, *37*, 66, *70*, *73*, *77*, *80*, *83*, *85*, *102*, *104*, *106*
mlr_learners.mlp, *11*, *37*, 68, *68*, *73*, *77*, *80*, *83*, *85*, *102*, *104*, *106*
mlr_learners.module, *11*, *37*, *68*, *70*, 71, *77*, *80*, *83*, *85*, *102*, *104*, *106*
mlr_learners.realmlp, *11*, *37*, *68*, *70*, *73*, 74, *80*, *83*, *85*, *102*, *104*, *106*
mlr_learners.tab_resnet, *11*, *37*, *68*, *70*, *73*, *77*, 78, *83*, *85*, *102*, *104*, *106*
mlr_learners.tabm, *11*, *37*, *68*, *70*, *73*, *77*, 80, *80*, *85*, *102*, *104*, *106*
mlr_learners.torch_featureless, *11*, *37*, *68*, *70*, *73*, *77*, *80*, *83*, 84, *102*, *104*, *106*
mlr_learners.torchvision, 86
mlr_learners_torch, *11*, *37*, *68*, *70*, *73*, *77*, *80*, *83*, *85*, 90, *104*, *106*
mlr_learners_torch_image, *11*, *37*, *68*, *70*, *73*, *77*, *80*, *83*, *85*, 102, *102*, *106*
mlr_learners_torch_model, *11*, *37*, *68*, *70*, *73*, *77*, *80*, *83*, *85*, *102*, 104, *104*, *121*, *248*, *253, 254*, *256*, *258*, *286–288*, *290*, *297*, *305*, *324*
mlr_measures_torch, 107, *109*
mlr_measures_torch.default, 108, *108*
mlr_pipeops, *244*, *292*
mlr_pipeops_augment_center_crop, 109
mlr_pipeops_augment_color_jitter, 110
mlr_pipeops_augment_crop, 111
mlr_pipeops_augment_hflip, 111
mlr_pipeops_augment_random_affine, 112
mlr_pipeops_augment_random_choice, 113
mlr_pipeops_augment_random_crop, 114
mlr_pipeops_augment_random_horizontal_flip, 114

mlr_pipeops_augment_random_order, 115
mlr_pipeops_augment_random_resized_crop, 116
mlr_pipeops_augment_random_vertical_flip, 116
mlr_pipeops_augment_resized_crop, 117
mlr_pipeops_augment_rotate, 118
mlr_pipeops_augment_vflip, 119
mlr_pipeops_module, *11*, *106*, 119, *248*, *253*, *254*, *256*, *258*, *286–288*, *290*, *297*, *305*, *324*
mlr_pipeops_nn_adaptive_avg_pool1d, 122
mlr_pipeops_nn_adaptive_avg_pool2d, 124
mlr_pipeops_nn_adaptive_avg_pool3d, 125
mlr_pipeops_nn_adaptive_max_pool1d, 127
mlr_pipeops_nn_adaptive_max_pool2d, 128
mlr_pipeops_nn_adaptive_max_pool3d, 130
mlr_pipeops_nn_avg_pool1d, 131
mlr_pipeops_nn_avg_pool2d, 133
mlr_pipeops_nn_avg_pool3d, 134
mlr_pipeops_nn_batch_norm1d, 136
mlr_pipeops_nn_batch_norm2d, 138
mlr_pipeops_nn_batch_norm3d, 139
mlr_pipeops_nn_block, 141
mlr_pipeops_nn_celu, 143
mlr_pipeops_nn_conv1d, 149
mlr_pipeops_nn_conv2d, 151
mlr_pipeops_nn_conv3d, 153
mlr_pipeops_nn_conv_transpose1d, 144
mlr_pipeops_nn_conv_transpose2d, 146
mlr_pipeops_nn_conv_transpose3d, 148
mlr_pipeops_nn_dropout, 155
mlr_pipeops_nn_dropout2d, 156
mlr_pipeops_nn_dropout3d, 158
mlr_pipeops_nn_elu, 159
mlr_pipeops_nn_flatten, 161
mlr_pipeops_nn_fn, 162
mlr_pipeops_nn_ft_cls, 164
mlr_pipeops_nn_ft_transformer_block, 165
mlr_pipeops_nn_geglu, 167
mlr_pipeops_nn_gelu, 168
mlr_pipeops_nn_glu, 169
mlr_pipeops_nn_group_norm, 170
mlr_pipeops_nn_gru, 172
mlr_pipeops_nn_hardshrink, 173
mlr_pipeops_nn_hardsigmoid, 175
mlr_pipeops_nn_hardtanh, 176
mlr_pipeops_nn_head, 177
mlr_pipeops_nn_identity, 179
mlr_pipeops_nn_layer_norm, 180
mlr_pipeops_nn_leaky_relu, 182
mlr_pipeops_nn_linear, 183
mlr_pipeops_nn_log_sigmoid, 184
mlr_pipeops_nn_log_softmax, 186
mlr_pipeops_nn_lstm, 187
mlr_pipeops_nn_max_pool1d, 189
mlr_pipeops_nn_max_pool2d, 190
mlr_pipeops_nn_max_pool3d, 192
mlr_pipeops_nn_merge, 193
mlr_pipeops_nn_merge_cat, 195
mlr_pipeops_nn_merge_prod, 197
mlr_pipeops_nn_merge_sum, 198
mlr_pipeops_nn_multihead_attention, 199
mlr_pipeops_nn_prelu, 202
mlr_pipeops_nn_recurrent, 203, *264*
mlr_pipeops_nn_reglu, 205
mlr_pipeops_nn_relu, 206
mlr_pipeops_nn_relu6, 207
mlr_pipeops_nn_reshape, 208
mlr_pipeops_nn_rnn, 210
mlr_pipeops_nn_rrelu, 212
mlr_pipeops_nn_selu, 213
mlr_pipeops_nn_sigmoid, 214
mlr_pipeops_nn_silu, 216
mlr_pipeops_nn_softmax, 217
mlr_pipeops_nn_softmax2d, 218
mlr_pipeops_nn_softmin, 220
mlr_pipeops_nn_softplus, 221
mlr_pipeops_nn_softshrink, 222
mlr_pipeops_nn_softsign, 224
mlr_pipeops_nn_squeeze, 225
mlr_pipeops_nn_tanh, 226
mlr_pipeops_nn_tanhshrink, 228
mlr_pipeops_nn_threshold, 229
mlr_pipeops_nn_tokenizer_categ, 230
mlr_pipeops_nn_tokenizer_num, 232
mlr_pipeops_nn_transformer_encoder_layer, 233

mlr_pipeops_nn_unflatten, 236
mlr_pipeops_nn_unsqueeze, 238
mlr_pipeops_preproc_torch, 239
mlr_pipeops_torch, *11*, *106*, *121*, 244, *253*, *254*, *256*, *258*, *286–288*, *290*, *297*, *305*, *324*
mlr_pipeops_torch_callbacks, 250, *260*, *269*, *288*, *290*
mlr_pipeops_torch_ingress, *11*, *106*, *121*, *248*, 251, *254*, *256*, *258*, *286–288*, *290*, *297*, *305*, *324*
mlr_pipeops_torch_ingress_categ, *11*, *106*, *121*, *248*, 253, *253*, *256*, *258*, *286–288*, *290*, *297*, *305*, *324*
mlr_pipeops_torch_ingress_ltnsr, *11*, *106*, *121*, *248*, *253*, *254*, 255, *258*, *286–288*, *290*, *297*, *305*, *324*
mlr_pipeops_torch_ingress_num, *11*, *106*, *121*, *248*, *253*, *254*, *256*, 257, *286–288*, *290*, *297*, *305*, *324*
mlr_pipeops_torch_loss, *251*, 259, *269*, *288*, *290*
mlr_pipeops_torch_model, *204*, 260
mlr_pipeops_torch_model_classif, 265
mlr_pipeops_torch_model_regr, 266
mlr_pipeops_torch_optimizer, *251*, *260*, 268, *288*, *290*
mlr_pipeops_trafo_adjust_brightness, 269
mlr_pipeops_trafo_adjust_gamma, 270
mlr_pipeops_trafo_adjust_hue, 271
mlr_pipeops_trafo_adjust_saturation, 271
mlr_pipeops_trafo_grayscale, 272
mlr_pipeops_trafo_nop, 273
mlr_pipeops_trafo_normalize, 273
mlr_pipeops_trafo_pad, 274
mlr_pipeops_trafo_reshape, 274
mlr_pipeops_trafo_resize, 275
mlr_pipeops_trafo_rgb_to_grayscale, 276
mlr_reflections$learner_predict_types, *73*, *101*, *104*, *106*, *264*
mlr_reflections$learner_properties, *100*, *103*
mlr_reflections$task_feature_types, *100*, *253*
mlr_reflections$task_types, *97*
mlr_tasks_cifar, 276
mlr_tasks_cifar10 *(*mlr_tasks_cifar*)*, 276
mlr_tasks_cifar100 *(*mlr_tasks_cifar*)*, 276
mlr_tasks_lazy_iris, 277
mlr_tasks_melanoma, 278
mlr_tasks_mnist, 280
mlr_tasks_tiny_imagenet, 281
mlr_tasks_torch, 282
model descriptor union, *246*
model_descriptor_to_learner, 286
model_descriptor_to_learner(), *11*, *106*, *121*, *248*, *253*, *254*, *256*, *258*, *263*, *265*, *267*, *287*, *288*, *290*, *297*, *305*, *324*
model_descriptor_to_module, *286*, 287, *289*
model_descriptor_to_module(), *11*, *106*, *121*, *248*, *253*, *254*, *256*, *258*, *286*, *288*, *290*, *297*, *305*, *324*
model_descriptor_union, *246*, 288
model_descriptor_union(), *11*, *106*, *121*, *248*, *251*, *253*, *254*, *256*, *258*, *260*, *269*, *286*, *287*, *290*, *297*, *305*, *324*
ModelDescriptor, *119*, *240*, *244*, *246*, *250*, *259*, *260*, *263*, *265*, *267*, *268*, *286–288*, 289, *319*, *323*, *325*, *327*
ModelDescriptor(), *11*, *106*, *121*, *248*, *251*, *253*, *254*, *256*, *258*, *260*, *269*, *286–288*, *297*, *305*, *324*
module, *121*
msr(torch.default), *283*
msr_torch, 290
msr_torch(), *107*

network, *94*
nn, 292
nn(), *244*
nn(rnn), *172*, *187*
nn_adaptive_avg_pool1d(), *122*
nn_adaptive_avg_pool2d(), *124*
nn_adaptive_avg_pool3d(), *125*
nn_adaptive_max_pool1d(), *127*
nn_adaptive_max_pool2d(), *128*
nn_adaptive_max_pool3d(), *130*
nn_avg_pool1d(), *131*
nn_avg_pool2d(), *133*
nn_avg_pool3d(), *134*
nn_bce_with_logits_loss, *28*, *29*

nn_conv_transpose1d, *144*
nn_conv_transpose2d, *146*
nn_conv_transpose3d, *148*
nn_cross_entropy_loss, *28, 29*
nn_ft_cls, 293
nn_ft_cls(), *164*
nn_ft_transformer_block, 293
nn_ft_transformer_block(), *165*
nn_geglu, *167*, 295
nn_graph, *286, 287*, *289*, 296, *296*
nn_graph(), *11*, *106*, *121*, *248*, *253, 254*, *256*, *258*, *286–288*, *290*, *305*, *324*
nn_linear, *244*, *246*
nn_merge_cat, 297
nn_merge_cat(), *195*
nn_merge_prod, 298
nn_merge_prod(), *197*
nn_merge_sum, 298
nn_merge_sum(), *198*
nn_module, *36*, *69*, *81*, *97*, *105*, *119, 120*, *244*
nn_module(), *99*, *104*, *303*, *305*
nn_module_generator, *81*
nn_module_list, *297*
nn_reglu, *205*, 298
nn_relu, *69*, *81*
nn_reshape, 299
nn_reshape(), *208*, *308*
nn_sequential, *236*
nn_silu(), *216*
nn_squeeze, 299
nn_squeeze(), *225*
nn_tokenizer_categ, 300
nn_tokenizer_categ(), *230*
nn_tokenizer_num, 300
nn_tokenizer_num(), *232*
nn_unsqueeze, 301
nn_unsqueeze(), *238*, *310*

optimizer, *94*
output_dim_for, 301
output_dim_for(), *97*, *178*, *282, 283*

ParamSet, *72*, *89*, *98–100*, *103*, *141*, *163*, *194*, *204*, *241*, *244*, *247*, *253*, *303, 304*, *319*, *321, 322*, *325–327*
ParamUty, *327*
PipeOp, *17–23*, *42*, *177*, *244, 245*, *248*, *292*, *308*, *310*
pipeop_preproc_torch, *239*, 302
pipeop_torch, 303
pipeop_torch(), *11*, *106*, *121*, *244*, *248*, *253, 254*, *256*, *258*, *286–288*, *290*, *297*, *324*
PipeOpModule, *29*, *240*, *244–246*, *248*, *289*, *296*
PipeOpModule *(*mlr_pipeops_module*)*, 119
PipeOpNOP, *244*, *289*, *296*
PipeOpPreprocTorchAugmentCenterCrop *(*mlr_pipeops_augment_center_crop*)*, 109
PipeOpPreprocTorchAugmentColorJitter *(*mlr_pipeops_augment_color_jitter*)*, 110
PipeOpPreprocTorchAugmentCrop *(*mlr_pipeops_augment_crop*)*, 111
PipeOpPreprocTorchAugmentHflip *(*mlr_pipeops_augment_hflip*)*, 111
PipeOpPreprocTorchAugmentRandomAffine *(*mlr_pipeops_augment_random_affine*)*, 112
PipeOpPreprocTorchAugmentRandomChoice *(*mlr_pipeops_augment_random_choice*)*, 113
PipeOpPreprocTorchAugmentRandomCrop *(*mlr_pipeops_augment_random_crop*)*, 114
PipeOpPreprocTorchAugmentRandomHorizontalFlip *(*mlr_pipeops_augment_random_horizontal_flip*)*, 114
PipeOpPreprocTorchAugmentRandomOrder *(*mlr_pipeops_augment_random_order*)*, 115
PipeOpPreprocTorchAugmentRandomResizedCrop *(*mlr_pipeops_augment_random_resized_crop*)*, 116
PipeOpPreprocTorchAugmentRandomVerticalFlip *(*mlr_pipeops_augment_random_vertical_flip*)*, 116
PipeOpPreprocTorchAugmentResizedCrop *(*mlr_pipeops_augment_resized_crop*)*, 117
PipeOpPreprocTorchAugmentRotate *(*mlr_pipeops_augment_rotate*)*, 118
PipeOpPreprocTorchAugmentVflip *(*mlr_pipeops_augment_vflip*)*,

119
PipeOpPreprocTorchTrafoAdjustBrightness (mlr_pipeops_trafo_adjust_brightness), 269
PipeOpPreprocTorchTrafoAdjustGamma (mlr_pipeops_trafo_adjust_gamma), 270
PipeOpPreprocTorchTrafoAdjustHue (mlr_pipeops_trafo_adjust_hue), 271
PipeOpPreprocTorchTrafoAdjustSaturation (mlr_pipeops_trafo_adjust_saturation), 271
PipeOpPreprocTorchTrafoGrayscale (mlr_pipeops_trafo_grayscale), 272
PipeOpPreprocTorchTrafoNop (mlr_pipeops_trafo_nop), 273
PipeOpPreprocTorchTrafoNormalize (mlr_pipeops_trafo_normalize), 273
PipeOpPreprocTorchTrafoPad (mlr_pipeops_trafo_pad), 274
PipeOpPreprocTorchTrafoReshape (mlr_pipeops_trafo_reshape), 274
PipeOpPreprocTorchTrafoResize (mlr_pipeops_trafo_resize), 275
PipeOpPreprocTorchTrafoRgbToGrayscale (mlr_pipeops_trafo_rgb_to_grayscale), 276
PipeOpTaskPreproc, *240*
PipeOpTaskPreprocSimple, *240*
PipeOpTaskPreprocTorch, *109–119*, *163*, *269–276*, *302*, *303*
PipeOpTaskPreprocTorch (mlr_pipeops_preproc_torch), 239
PipeOpTorch, *23*, *119*, *123–130*, *132*, *133*, *135–143*, *145–164*, *166–176*, *178*, *179*, *181–189*, *191–195*, *197–202*, *204–209*, *211–217*, *219–221*, *223–225*, *227–229*, *231*, *232*, *235*, *237–239*, *244*, *246*, *252*, *254*, *255*, *257*, *259*, *288*, *289*, *303–305*, *310*
PipeOpTorch (mlr_pipeops_torch), 244
PipeOpTorchAdaptiveAvgPool1D (mlr_pipeops_nn_adaptive_avg_pool1d), 122
PipeOpTorchAdaptiveAvgPool2D (mlr_pipeops_nn_adaptive_avg_pool2d), 124
PipeOpTorchAdaptiveAvgPool3D (mlr_pipeops_nn_adaptive_avg_pool3d), 125
PipeOpTorchAdaptiveMaxPool1D (mlr_pipeops_nn_adaptive_max_pool1d), 127
PipeOpTorchAdaptiveMaxPool2D (mlr_pipeops_nn_adaptive_max_pool2d), 128
PipeOpTorchAdaptiveMaxPool3D (mlr_pipeops_nn_adaptive_max_pool3d), 130
PipeOpTorchAvgPool1D (mlr_pipeops_nn_avg_pool1d), 131
PipeOpTorchAvgPool2D (mlr_pipeops_nn_avg_pool2d), 133
PipeOpTorchAvgPool3D (mlr_pipeops_nn_avg_pool3d), 134
PipeOpTorchBatchNorm1D (mlr_pipeops_nn_batch_norm1d), 136
PipeOpTorchBatchNorm2D (mlr_pipeops_nn_batch_norm2d), 138
PipeOpTorchBatchNorm3D (mlr_pipeops_nn_batch_norm3d), 139
PipeOpTorchBlock (mlr_pipeops_nn_block), 141
PipeOpTorchCallbacks, *36*, *289*
PipeOpTorchCallbacks (mlr_pipeops_torch_callbacks), 250
PipeOpTorchCELU (mlr_pipeops_nn_celu), 143
PipeOpTorchConv1D (mlr_pipeops_nn_conv1d), 149
PipeOpTorchConv2D (mlr_pipeops_nn_conv2d), 151
PipeOpTorchConv3D (mlr_pipeops_nn_conv3d), 153

PipeOpTorchConvTranspose1D (mlr_pipeops_nn_conv_transpose1d), 144
PipeOpTorchConvTranspose2D (mlr_pipeops_nn_conv_transpose2d), 146
PipeOpTorchConvTranspose3D (mlr_pipeops_nn_conv_transpose3d), 148
PipeOpTorchDropout (mlr_pipeops_nn_dropout), 155
PipeOpTorchDropout2D (mlr_pipeops_nn_dropout2d), 156
PipeOpTorchDropout3D (mlr_pipeops_nn_dropout3d), 158
PipeOpTorchELU (mlr_pipeops_nn_elu), 159
PipeOpTorchFlatten (mlr_pipeops_nn_flatten), 161
PipeOpTorchFn (mlr_pipeops_nn_fn), 162
PipeOpTorchFTCLS (mlr_pipeops_nn_ft_cls), 164
PipeOpTorchFTTransformerBlock, *67*
PipeOpTorchFTTransformerBlock (mlr_pipeops_nn_ft_transformer_block), 165
PipeOpTorchGeGLU (mlr_pipeops_nn_geglu), 167
PipeOpTorchGELU (mlr_pipeops_nn_gelu), 168
PipeOpTorchGLU (mlr_pipeops_nn_glu), 169
PipeOpTorchGroupNorm (mlr_pipeops_nn_group_norm), 170
PipeOpTorchGRU, *203*
PipeOpTorchGRU (mlr_pipeops_nn_gru), 172
PipeOpTorchHardShrink (mlr_pipeops_nn_hardshrink), 173
PipeOpTorchHardSigmoid (mlr_pipeops_nn_hardsigmoid), 175
PipeOpTorchHardTanh (mlr_pipeops_nn_hardtanh), 176
PipeOpTorchHead, *245*, *248*, *301*, *302*
PipeOpTorchHead (mlr_pipeops_nn_head), 177
PipeOpTorchIdentity (mlr_pipeops_nn_identity), 179
PipeOpTorchIngress, *119*, *246*, *254*, *255*, *258*, *289*
PipeOpTorchIngress (mlr_pipeops_torch_ingress), 251
PipeOpTorchIngressCategorical, *252*
PipeOpTorchIngressCategorical (mlr_pipeops_torch_ingress_categ), 253
PipeOpTorchIngressLazyTensor, *252*
PipeOpTorchIngressLazyTensor (mlr_pipeops_torch_ingress_ltnsr), 255
PipeOpTorchIngressNumeric, *252*
PipeOpTorchIngressNumeric (mlr_pipeops_torch_ingress_num), 257
PipeOpTorchLayerNorm (mlr_pipeops_nn_layer_norm), 180
PipeOpTorchLeakyReLU (mlr_pipeops_nn_leaky_relu), 182
PipeOpTorchLinear (mlr_pipeops_nn_linear), 183
PipeOpTorchLogSigmoid (mlr_pipeops_nn_log_sigmoid), 184
PipeOpTorchLogSoftmax (mlr_pipeops_nn_log_softmax), 186
PipeOpTorchLoss, *36*, *289*
PipeOpTorchLoss (mlr_pipeops_torch_loss), 259
PipeOpTorchLSTM, *203*
PipeOpTorchLSTM (mlr_pipeops_nn_lstm), 187
PipeOpTorchMaxPool1D (mlr_pipeops_nn_max_pool1d), 189
PipeOpTorchMaxPool2D (mlr_pipeops_nn_max_pool2d), 190
PipeOpTorchMaxPool3D (mlr_pipeops_nn_max_pool3d), 192
PipeOpTorchMerge, *195*, *197*, *199*
PipeOpTorchMerge

(mlr_pipeops_nn_merge), 193
PipeOpTorchMergeCat, *193*
PipeOpTorchMergeCat (mlr_pipeops_nn_merge_cat), 195
PipeOpTorchMergeProd, *193*
PipeOpTorchMergeProd (mlr_pipeops_nn_merge_prod), 197
PipeOpTorchMergeSum, *193*
PipeOpTorchMergeSum (mlr_pipeops_nn_merge_sum), 198
PipeOpTorchModel, *10*, *35*, *240*, *265*, *267*
PipeOpTorchModel (mlr_pipeops_torch_model), 260
PipeOpTorchModelClassif, *260*
PipeOpTorchModelClassif (mlr_pipeops_torch_model_classif), 265
PipeOpTorchModelRegr, *260*
PipeOpTorchModelRegr (mlr_pipeops_torch_model_regr), 266
PipeOpTorchMultiheadAttention (mlr_pipeops_nn_multihead_attention), 199
PipeOpTorchOptimizer, *36*, *289*
PipeOpTorchOptimizer (mlr_pipeops_torch_optimizer), 268
PipeOpTorchPReLU (mlr_pipeops_nn_prelu), 202
PipeOpTorchRecurrent, *173*, *188*, *211*
PipeOpTorchRecurrent (mlr_pipeops_nn_recurrent), 203
PipeOpTorchReGLU (mlr_pipeops_nn_reglu), 205
PipeOpTorchReLU (mlr_pipeops_nn_relu), 206
PipeOpTorchReLU6 (mlr_pipeops_nn_relu6), 207
PipeOpTorchReshape (mlr_pipeops_nn_reshape), 208
PipeOpTorchRNN, *203*
PipeOpTorchRNN (mlr_pipeops_nn_rnn), 210
PipeOpTorchRReLU (mlr_pipeops_nn_rrelu), 212
PipeOpTorchSELU (mlr_pipeops_nn_selu), 213
PipeOpTorchSigmoid (mlr_pipeops_nn_sigmoid), 214
PipeOpTorchSiLU (mlr_pipeops_nn_silu), 216
PipeOpTorchSoftmax (mlr_pipeops_nn_softmax), 217
PipeOpTorchSoftmax2D (mlr_pipeops_nn_softmax2d), 218
PipeOpTorchSoftmin (mlr_pipeops_nn_softmin), 220
PipeOpTorchSoftPlus (mlr_pipeops_nn_softplus), 221
PipeOpTorchSoftShrink (mlr_pipeops_nn_softshrink), 222
PipeOpTorchSoftSign (mlr_pipeops_nn_softsign), 224
PipeOpTorchSqueeze (mlr_pipeops_nn_squeeze), 225
PipeOpTorchTanh (mlr_pipeops_nn_tanh), 226
PipeOpTorchTanhShrink (mlr_pipeops_nn_tanhshrink), 228
PipeOpTorchThreshold (mlr_pipeops_nn_threshold), 229
PipeOpTorchTokenizerCateg (mlr_pipeops_nn_tokenizer_categ), 230
PipeOpTorchTokenizerNum (mlr_pipeops_nn_tokenizer_num), 232
PipeOpTorchTransformerEncoderLayer (mlr_pipeops_nn_transformer_encoder_layer), 233
PipeOpTorchUnflatten (mlr_pipeops_nn_unflatten), 236
PipeOpTorchUnsqueeze (mlr_pipeops_nn_unsqueeze), 238
Prediction, *306*
PredictionTorch, 306

R6, *30*, *44*, *50*, *54*, *55*, *57*, *58*, *60*, *61*, *64*, *67*, *70*, *72*, *76*, *79*, *82*, *85*, *88*, *100*, *103*, *104*, *107*, *109*, *120*, *123*, *124*, *126*, *127*, *129*, *130*, *132*, *134*, *135*, *137*, *139*, *140*, *142*, *143*, *145*, *147*, *149*, *150*, *152*, *154*, *155*, *157*, *158*, *160–162*, *164*, *166–168*, *170*, *171*,

173–175, *177–179*, *181*, *182*, *184–186*, *188*, *190*, *191*, *193*, *194*, *196*, *197*, *199*, *201*, *203–206*, *208*, *209*, *211*, *212*, *214–216*, *218–220*, *222–224*, *226–228*, *230*, *231*, *233*, *235*, *237*, *238*, *241*, *247*, *250*, *252*, *254*, *256*, *258*, *259*, *264*, *265*, *267*, *268*, *282*, *306*, *319*, *322*, *325*, *328*
R6::R6Class, *277*
R6Class, *28*, *109–119*, *269–276*, *303*, *305*, *317*
ResampleResult, *92*
reshape_output_shape, 308
reshape_output_shape(), *38*, *311*
resolve_dim *(*shape_to_str*)*, 310
resolve_dim(), *17*, *20*

saveRDS(), *92*
Select, 309, *309*
select_all *(*Select*)*, 309
select_grep *(*Select*)*, 309
select_invert *(*Select*)*, 309
select_name *(*Select*)*, 309
select_none *(*Select*)*, 309
Selector, *33*
set.seed(), *94*, *261*
shape_helpers, *245*
shape_helpers *(*shape_to_str*)*, 310
shape_to_str, 310
shape_to_str(), *38*, *308*

t_clbk, 311
t_clbk(), *13–16*, *28*, *49*, *51*, *59*, *60*, *62*, *66*, *284–286*, *312*, *313*, *318–320*, *323*, *326*, *329*
t_clbk(checkpoint), *91*, *95*, *262*
t_clbks *(*t_clbk*)*, 311
t_loss, 312
t_loss(), *14–16*, *285*, *286*, *312*, *313*, *320*, *323*, *325*, *326*, *329*
t_losses *(*t_loss*)*, 312
t_opt, *285*, 313
t_opt(), *14–16*, *284–286*, *312*, *320*, *323*, *326*, *327*, *329*
t_opts *(*t_opt*)*, 313
Task, *32*, *33*, *35*, *36*, *63*, *65*, *87*, *97*, *98*, *102*, *242*, *244–246*, *248*, *250*, *263*, *265*, *267*, *268*, *282*, *289*, *304*, *314*, *323*
task, *277*, *279–281*
task_dataset, *33*, *98*, 314
task_dataset(), *240*
TaskClassif, *92*, *178*
TaskRegr, *92*, *178*
TaskTorch, *12*, *13*, *73*, *97*, *106–108*, *260*, *264*, *283*, *306*, *307*
TaskTorch *(*mlr_tasks_torch*)*, 282
tensor, *121*
tensors, *244*
torch::dataloader, *63*, *65*, *98*
torch::dataset, *29*, *30*, *98*, *315*
torch::jit_trace(), *95*, *261*
torch::lr_cosine_annealing(), *53*
torch::lr_lambda(), *53*
torch::lr_multiplicative(), *53*
torch::lr_one_cycle(), *53*, *55*
torch::lr_reduce_on_plateau(), *53*, *56*
torch::lr_scheduler(), *53*
torch::lr_step(), *53*
torch::nn_batch_norm1d(), *136*
torch::nn_batch_norm2d(), *138*
torch::nn_batch_norm3d(), *139*
torch::nn_celu(), *143*
torch::nn_conv1d(), *149*
torch::nn_conv2d(), *151*
torch::nn_conv3d(), *153*
torch::nn_dropout(), *155*
torch::nn_dropout2d(), *156*
torch::nn_dropout3d(), *158*
torch::nn_elu(), *159*
torch::nn_flatten(), *161*
torch::nn_gelu(), *168*
torch::nn_glu(), *169*
torch::nn_group_norm(), *170*
torch::nn_gru(), *172*
torch::nn_hardshrink(), *173*
torch::nn_hardsigmoid(), *175*
torch::nn_hardtanh(), *176*
torch::nn_identity(), *179*
torch::nn_layer_norm(), *180*
torch::nn_leaky_relu(), *182*
torch::nn_linear(), *178*, *183*
torch::nn_log_sigmoid(), *184*
torch::nn_log_softmax(), *186*
torch::nn_lstm(), *187*
torch::nn_max_pool1d(), *189*
torch::nn_max_pool2d(), *190*
torch::nn_max_pool3d(), *192*
torch::nn_module, *63*, *65*, *71*, *97*

torch::nn_multihead_attention(), *199*, *200*
torch::nn_prelu(), *202*
torch::nn_relu(), *206*
torch::nn_relu6(), *207*
torch::nn_rnn(), *172*, *188*, *210*
torch::nn_rrelu(), *212*
torch::nn_selu(), *213*
torch::nn_sigmoid(), *214*
torch::nn_silu(), *216*
torch::nn_softmax(), *217*
torch::nn_softmax2d(), *218*
torch::nn_softmin(), *220*
torch::nn_softplus(), *221*
torch::nn_softshrink(), *222*
torch::nn_softsign(), *224*
torch::nn_tanh(), *226*
torch::nn_tanhshrink(), *228*
torch::nn_threshold(), *229*
torch::nn_transformer_encoder_layer(), *233*, *234*
torch::nn_unflatten(), *236*
torch::optimizer, *63*, *65*
torch::sampler, *96*, *262*, *263*
torch::sampler(), *96*, *262*
torch::torch_reshape(), *208*
torch::torch_squeeze(), *225*, *299*
torch::torch_unsqueeze(), *238*, *301*
torch_callback, 315
torch_callback(), *13*, *14*, *27*, *28*, *47*, *49*, *51*, *59*, *60*, *62*, *66*, *284*, *312*, *320*
torch_float, *92*
torch_get_default_dtype(), *273*
torch_long, *92*
torch_tensor, *9*, *29*, *30*, *32*, *33*, *35*, *42*, *47*, *64*, *98*, *234*, *241*, *329*
TorchCallback, *13–16*, *27*, *28*, *36*, *46*, *47*, *49*, *51*, *59*, *60*, *62*, *66*, *67*, *70*, *73*, *77*, *79*, *83*, *85*, *89*, *99*, *101*, *103*, *105*, *251*, *284–286*, *311–313*, *315*, *317*, 318, *318*, *319*, *321*, *323*, *326*, *329*
TorchDescriptor, *14–16*, *285*, *286*, *312*, *313*, *319*, *320*, 321, *325*, *326*, *328*, *329*
TorchIngressToken, *33*, *38*, *39*, *98*, *252*, *314*, 323
TorchIngressToken(), *11*, *72*, *105*, *106*, *121*, *248*, *253*, *254*, *256*, *258*, *286–288*, *290*, *297*, *305*
TorchLoss, *14–16*, *36*, *67*, *70*, *73*, *77*, *79*, *83*, *85*, *88*, *99*, *101*, *103*, *105*, *259*, *285*, *286*, *290*, *312*, *313*, *320*, *321*, *323*, *325*, *329*
TorchOptimizer, *14–16*, *36*, *67*, *70*, *72*, *77*, *79*, *82*, *85*, *88*, *99*, *101*, *103*, *105*, *268*, *285*, *286*, *290*, *312*, *313*, *320*, *321*, *323*, *326*, 327
torchvision::cifar10_dataset(), *276*
torchvision::tiny_imagenet_dataset(), *281*
torchvision::transform_adjust_brightness, *269*
torchvision::transform_adjust_gamma, *270*
torchvision::transform_adjust_hue, *271*
torchvision::transform_adjust_saturation, *271*
torchvision::transform_center_crop, *109*
torchvision::transform_color_jitter, *110*
torchvision::transform_crop, *111*
torchvision::transform_grayscale, *272*
torchvision::transform_hflip, *111*
torchvision::transform_normalize, *273*
torchvision::transform_pad, *274*
torchvision::transform_random_affine, *112*
torchvision::transform_random_choice, *113*
torchvision::transform_random_crop, *114*
torchvision::transform_random_horizontal_flip, *114*
torchvision::transform_random_order, *115*
torchvision::transform_random_resized_crop, *116*
torchvision::transform_random_vertical_flip, *116*
torchvision::transform_resize, *275*
torchvision::transform_resized_crop, *117*
torchvision::transform_rgb_to_grayscale, *276*
torchvision::transform_rotate, *118*
torchvision::transform_vflip, *119*