`mlr3torch`

Deep Learning in R with mlr3 and torch

Sebastian Fischer, Martin Binder, Prof. Dr. Bernd Bischl, Lukas Burk and others

2024-07-10

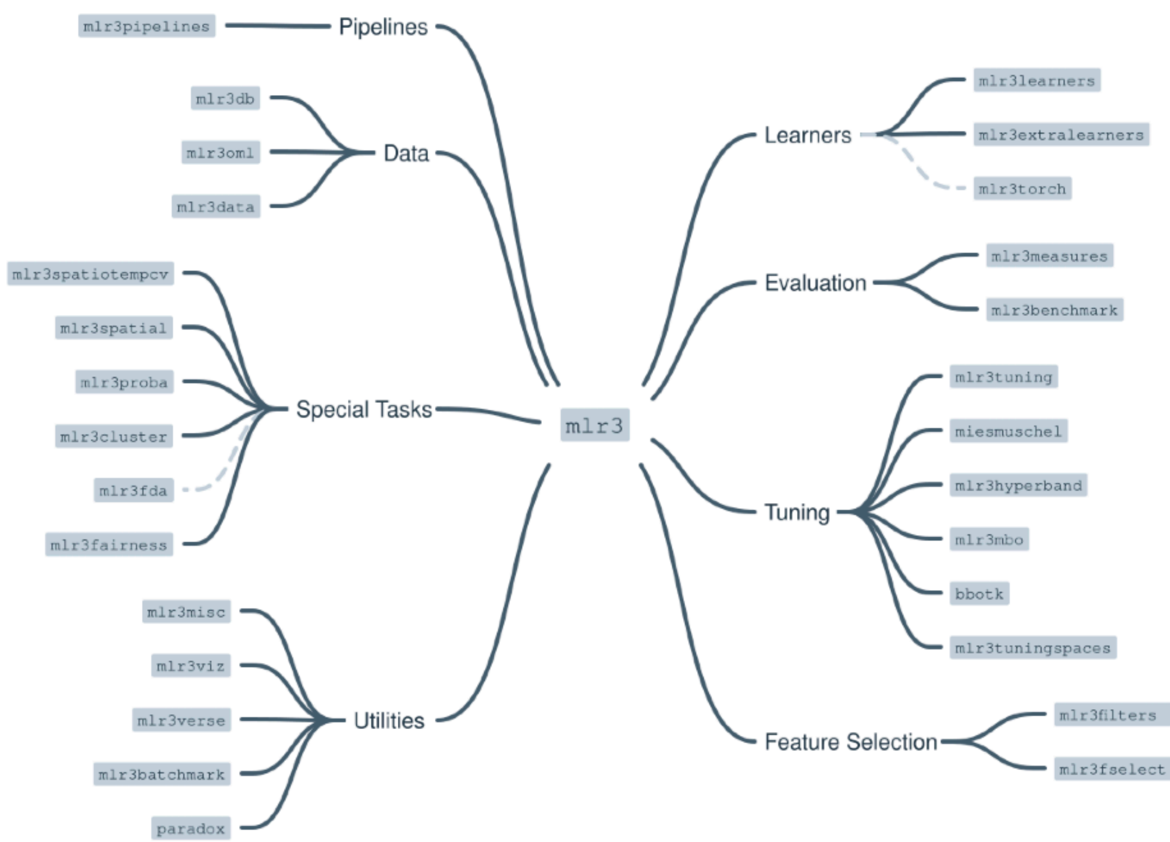
Introduction

- `mlr3torch` is a high level deep learning framework in R, built mainly on top of:
  - `mlr3` - A machine learning framework in R
  - `mlr3pipelines` - A *dataflow programming* toolkit
  - `torch` - A PyTorch implementation written in native R
- It allows to easily build, train and evaluate deep learning models
- GitHub: https://github.com/mlr-org/mlr3torch

Deep Learning in R

- `tensorflow`, `keras` & `rtorch` - use python libraries through reticulate
- `torch` - native R library
- `luz` - higher level deep learning library built on top of `torch`

The `mlr3` Ecosystem



`mlr3`'s "Hello World"

- `tsk()` creates an example `Task`, where the goal is to predict the miles-per-galleon of cars
- `lrn()` defines a simple regression tree `Learner` from the {rpart} package
- `rsmp()` sets a `Resampling` strategy

- `resample()` runs the resample experiment
- `msr()` initializes an mlr3 performance `Measure`

```
tsk_mtcars = tsk("mtcars")
lrn_tree = lrn("regr.rpart")
rsmp_cv = rsmp("cv", folds = 3)

rr = resample(
  task       = tsk_mtcars,
  learner    = lrn_tree,
  resampling = rsmp_cv
)

rr$aggregate(msr("regr.mse"))
```

```
regr.mse
      19
```

Dataflow Programming with `mlr3pipelines`

- `mlr3pipelines`[1] allows to assemble new `Learners` by connecting `PipeOps` in a `Graph`, e.g. for preprocessing
- `PipeOps` are created via `po()`, and combined using the chain operator `%>>%`

```
library(mlr3pipelines)
library(mlr3learners)

graph = po("encode", method = "one-hot") %>>%
  po("pca") %>>%
  lrn("classif.log_reg")

learner = as_learner(graph)
learner
```

[1] M. Binder and B. Bischl et al, JMLR, 2021

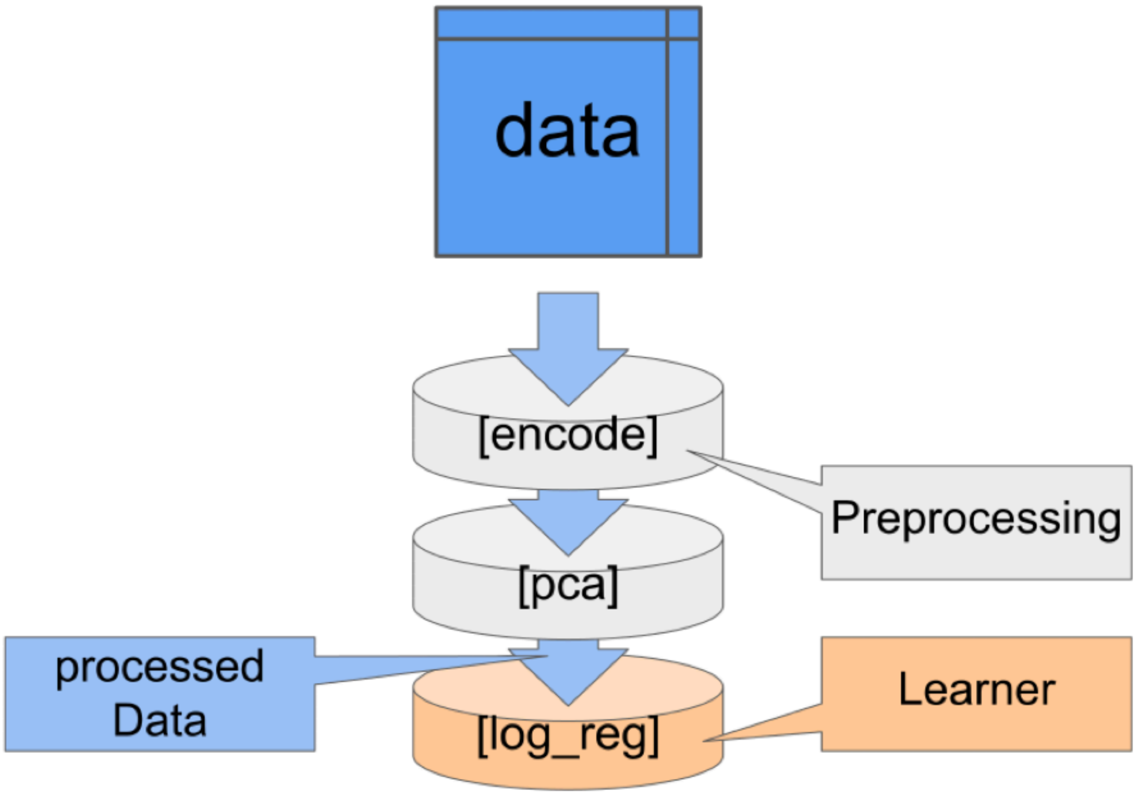


Figure 1: Sequential Preprocessing

torch's "Hello World"

- Support for GPU accelerated tensor operations
- An autograd system
- Provides many tensor operations and optimizers
- Extension for images via `torchvision`
- Developed by Daniel Falbel (Posit)
- GitHub: https://github.com/mlverse/torch

```r
library(torch)
x = torch_tensor(1, requires_grad = TRUE)
w = torch_tensor(2, requires_grad = TRUE)
b = torch_tensor(3, requires_grad = TRUE)
y = w * x + b
y$backward()
x$grad
```

```
torch_tensor
 2
[ CPUFloatType{1} ]
```

```r
w$grad
```

```
torch_tensor
 1
[ CPUFloatType{1} ]
```

```r
b$grad
```

```
torch_tensor
 1
[ CPUFloatType{1} ]
```

`mlr3torch` in a Nutshell

- Task Types:
  - Classification
  - Regression
- Learners:
  - Off-the-shelf architectures as predefined `Learner`s
  - Build architectures as `mlr3pipelines::Graph`s
  - Customization of training via a callback mechanism
- Data Types:
  - Tabular data
  - Generic `torch_tensor`s
  - Multi-modal data

Predefined architectures

- First, we construct and resample a multi layer perceptron with one hidden layer of size 10 and `ReLU` activation
- Then, we define a simple benchmark experiment that compares the neural network with the decision tree from earlier
- For this task, the neural network seems to be the wrong choice!

```
library(mlr3torch)
lrn_mlp = lrn("regr.mlp",
  activation     = torch::nn_relu,
  neurons        = 10,
  batch_size     = 32,
  epochs         = 100,
  loss           = t_loss("mse"),
  optimizer      = t_opt("adam", lr = 0.5),
  callbacks      = t_clbk("history")
)
```

```r
design = benchmark_grid(
  tsk_mtcars, list(lrn_mlp, lrn_tree), rsmp_cv
)
bmr = benchmark(design)
bmr$aggregate(msr("regr.mse"))
```

```
   nr task_id learner_id resampling_id iters regr.mse
1:  1  mtcars   regr.mlp            cv     3      140
2:  2  mtcars regr.rpart            cv     3       21
Hidden columns: resample_result
```

Neural Networks as `mlr3pipelines::Graphs`

- We can build the same architecture as before by connecting `PipeOps` in a `Graph`
- This `Graph` is fully interoperable with other `PipeOps`

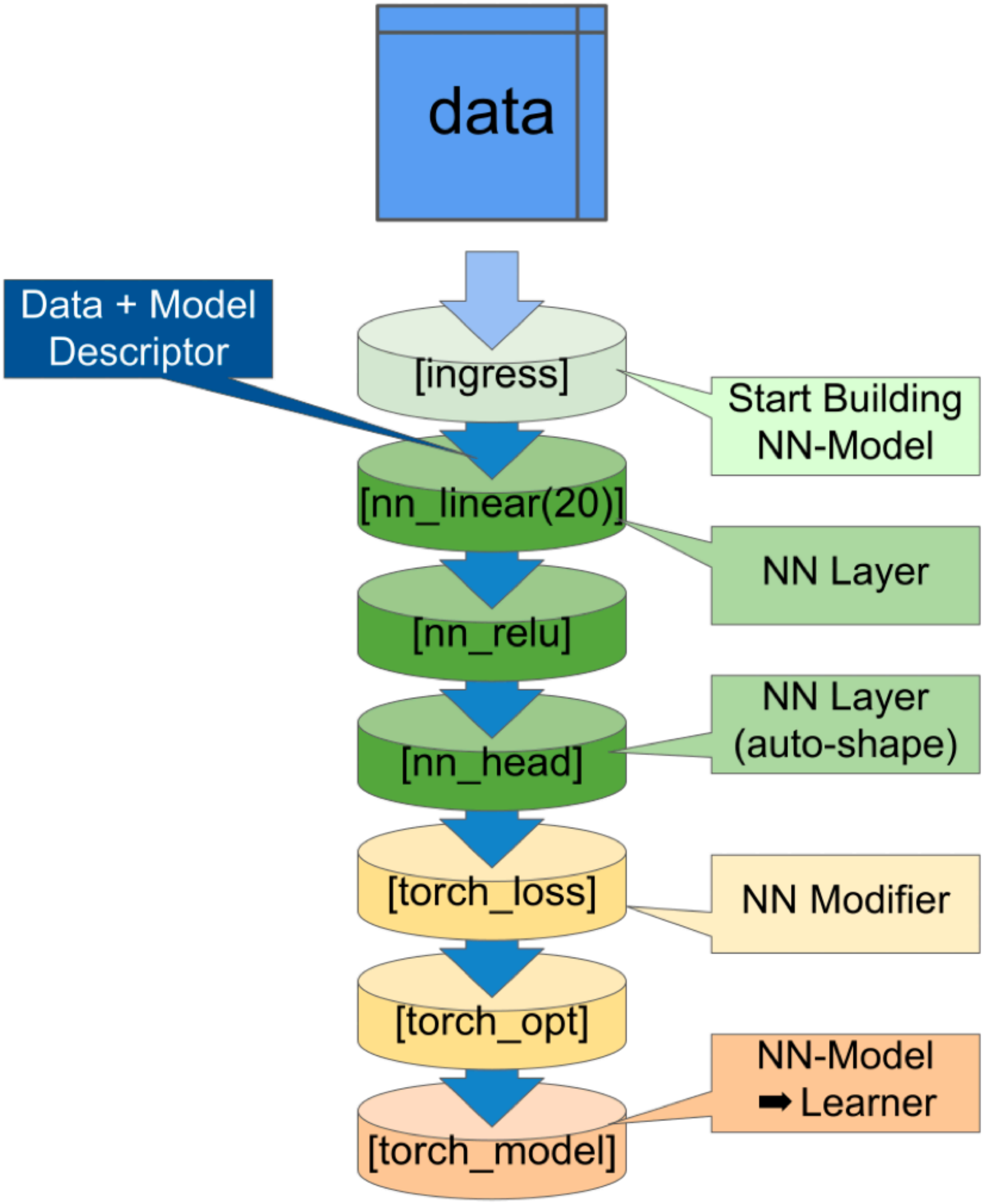
```r
mlp_graph = po("torch_ingress_ltnsr") %>>%
  po("nn_linear", out_features = 20) %>>%
  po("nn_relu") %>>%
  po("nn_head") %>>%
  po("torch_loss", t_loss("cross_entropy")) %>>%
  po("torch_optimizer", t_opt("adam", lr = 0.1)) %>>%
  po("torch_model_classif", batch_size = 16, epochs = 5)

lrn_mlp_graph = as_learner(mlp_graph)
```

Non-tabular data as `lazy_tensors`

- The images of the MNIST task are represented as `lazy_tensors`, which wrap a `torch::dataset`
- We have to reshape the MNIST images to work with our MLP that expects 2d inputs
- The preprocessing of `lazy_tensors` happens lazily by internally building up a preprocessing `Graph`
- We can combine the flattening step with our previous graph into a new `GraphLearner`

```r
tsk_mnist = tsk("mnist")
tsk_mnist$head(3)
```


data
Data + Model
Descriptor
[ingress]
Start Building
NN-Model
[nn_linear(20)]
NN Layer
[nn_relu]
NN Layer
(auto-shape)
[nn_head]
[torch_loss]
NN Modifier
[torch_opt]
NN-Model
➡ Learner
[torch_model]

```
   label              image
1:     5 <tnsr[1x28x28]>
2:     0 <tnsr[1x28x28]>
3:     4 <tnsr[1x28x28]>
```

```
flattener = po("trafo_reshape", shape = c(-1, 28 * 28))
flattener$train(list(tsk_mnist))[[1L]]$head(3)
```

```
   label        image
1:     5 <tnsr[784]>
2:     0 <tnsr[784]>
3:     4 <tnsr[784]>
```

```
lrn_flat_mlp = as_learner(flattener %>>% mlp_graph)
lrn_flat_mlp$train(tsk_mnist)
```

Hyperparameter Tuning

- The resulting `GraphLearner` has a parameter set representing all configuration options, which can be tuned!
- Optimize the latent dimension of the network and the learning rate of the optimizer using Bayesian Optimization from `mlr3mbo`.

```
library(mlr3tuning)
library(mlr3mbo)

as.data.table(lrn_flat_mlp$param_set)[c(5, 12), 1:4]
```

```
                        id    class lower upper
1: nn_linear.out_features ParamInt     1   Inf
2:     torch_optimizer.lr ParamDbl     0   Inf
```

```
lrn_flat_mlp$param_set$set_values(
  nn_linear.out_features = to_tune(10, 100),
  torch_optimizer.lr = to_tune(1e-05, 1e-03, logscale = TRUE)
)
```

```r
tune(
  learner = lrn_flat_mlp,
  task = tsk_mnist,
  tuner = tnr("mbo"),
  term_evals = 100L,
  resampling = rsmp("holdout")
)
```

Multi Modal Data

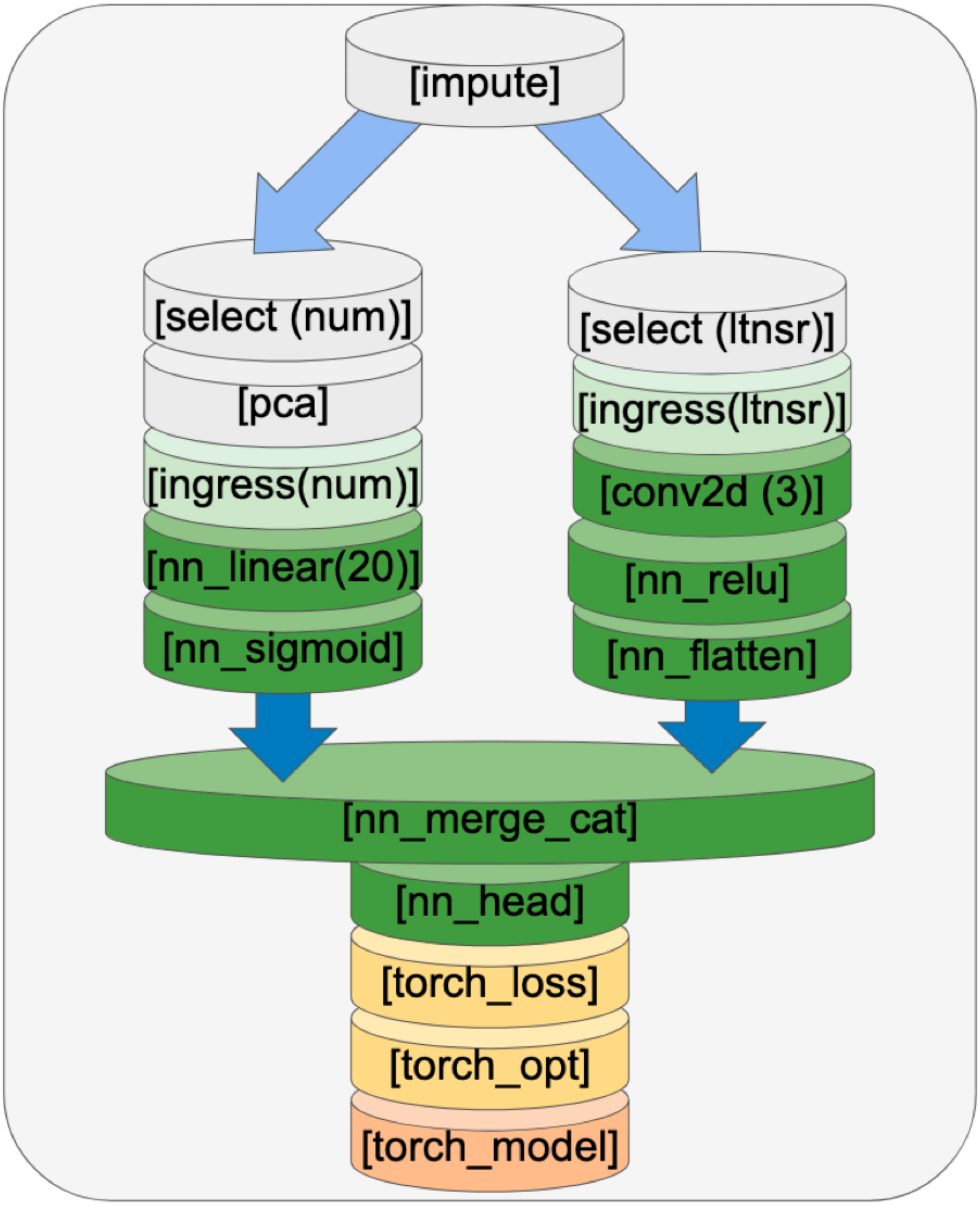
- **mlr3torch** naturally supports multi-modal data as **lazy_tensors** can be stored in **data.frames**
- Consider a classification task with some medical data about patients, as well as x-ray images
- **mlr3torch** allows to easily build neural networks with multiple inputs, e.g. each operating on a different subset of features

```r
medical_task
```

```
<TaskClassif:medical> (100 x 4): Medical Diagnosis
* Target: status
* Properties: twoclass
* Features (3):
  - dbl (2): age, lesion_size
  - lt (1): xray
```

```r
medical_task$head(3)
```

```
      status age lesion_size            xray
1:    benign  NA        0.99 <tnsr[3x64x64]>
2:    benign  21        0.56 <tnsr[3x64x64]>
3: malignant  56        0.71 <tnsr[3x64x64]>
```


[impute]
[select (num)]
[pca]
[ingress(num)]
[nn_linear(20)]
[nn_sigmoid]
[select (ltnsr)]
[ingress(ltnsr)]
[conv2d (3)]
[nn_relu]
[nn_flatten]
[nn_merge_cat]
[nn_head]
[torch_loss]
[torch_opt]
[torch_model]

Multi Modal Data

```
branch_num = po("select_1", selector = selector_type("numeric")) %>>%
  po("pca") %>>%
  po("torch_ingress_num") %>>%
  po("nn_linear", out_features = 20) %>>%
  po("nn_sigmoid")

branch_img = po("select_2", selector = selector_type("lazy_tensor")) %>>%
  po("torch_ingress_ltnsr") %>>%
  po("nn_conv2d", out_channels = 3) %>>%
  po("nn_relu") %>>%
  po("nn_flatten")

graph = po("imputeoor") %>>% list(branch_num, branch_img) %>>%
  po("nn_merge_cat") %>>%
  po("nn_head") %>>%
  po("torch_loss", t_loss("cross_entropy")) %>>%
  po("torch_optimizer", t_opt("adam", lr = 1e-4)) %>>%
  po("torch_model_classif", batch_size = 16, epochs = 50)
```

Learn More

- This presentation: https://github.com/sebffischer/mlr3torch-UseR-2024
- The mlr3 book: https://mlr3book.mlr-org.com/
- The mlr3 website: https://mlr-org.com/
- The mlr3torch package website https://mlr3torch.mlr-org.com/