

Neural networks for imbalanced classification: proposed AUM loss with mlr3torch in R

Toby Dylan Hocking
Professeur Agrégé, 2024–present,
Département d'Informatique
Université de Sherbrooke
toby.dylan.hocking@usherbrooke.ca
toby.hocking@r-project.org

April 7, 2025

Unbalanced classification, AUC, and proposed AUM

Basics of neural networks with torch in R

Implementing proposed AUM loss in mlr3torch framework

Conclusion


Learning two different functions using two data sets

Figure from chapter by Hocking TD, *Introduction to machine learning and neural networks* for book *Land Carbon Cycle Modeling: Matrix Approach, Data Assimilation, and Ecological Forecasting* edited by Luo Y (Taylor and Francis, 2022).

Learning Algorithm Train data Learned function Predictions on test data

Learn() $\rightarrow g$

$g(\text{0}) = 0$
 $g(\text{1}) = 1$
 $g(\text{1}) = 1$

Learn() $\rightarrow h$

$h(\text{shirt}) = 0$
 $h(\text{shirt}) = 0$
 $h(\text{pants}) = 1$

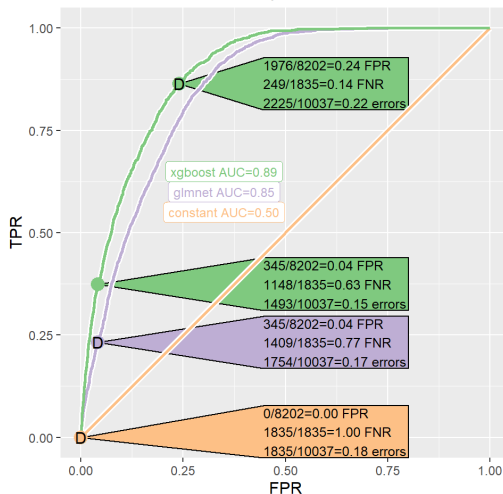
- ▶ Ten classes are equally distributed in these data (10% of each, balanced labels).
- ▶ What happens if there are 91% of one class, and 1% of each of the other 9 classes? (unbalanced labels)

How to deal with class imbalance?

- ▶ In binary classification, standard learning algorithms can yield sub-optimal prediction accuracy if train data have imbalanced labels.
- ▶ Predicting childhood autism (Lindly *et al.*), 3% autism, 97% not.
- ▶ Predicting presence of trees/burn in satellite imagery (Shenkin *et al.*, Thibault *et al.*), small percent of trees in deserts of Arizona, small percent of burned area out of total forested area in Quebec.
- ▶ Predicting fish spawning habitat in sonar imagery (Bodine *et al.*), small percent of suitable spawning habitat, out of total river bed.
- ▶ How do we adapt our learning algorithm, to handle the class imbalance? Re-weighting loss? Over/under-sampling?
- ▶ **New AUM loss for optimizing ROC curve.**

ROC curves: fair comparison with different default FPR

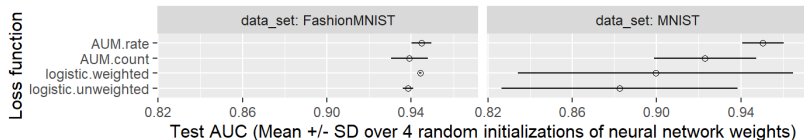
ROC curves, D=Default prediction threshold



ROC= Receiver Operating Characteristic curves show FPR, TPR for all cut points of the predicted probability.

- ▶ Imbalanced labels: 18% positive, 82% negative.
- ▶ At defaults (D), glmnet has fewer errors (misleading).
- ▶ At FPR=4%, xgboost has fewer errors (fair comparison).

Comparing AUM with weighted logistic loss



Hillman and Hocking, *Journal of Machine Learning Research* 2023.

- ▶ Two image classification data sets.
- ▶ LeNet5 convolutional neural network, batch size 1000.
- ▶ Step size from 10^{-4} to 10^2 (keep best).
- ▶ AUM rate uses Area Under Min of FPR/FNR, more accurate in these data than AUM count (FP/FN totals), .
- ▶ logistic unweighted is usual binary cross-entropy loss (uniform weight=1 for each sample).
- ▶ for logistic weighted, we compute class frequencies, $n_1 = \sum_{i=1}^N I[y_i = 1]$ and n_0 similar; then weights are $w_i = 1/n_{y_i}$ so that total weight of positive class equals total weight of negative class.

Unbalanced classification, AUC, and proposed AUM

Basics of neural networks with torch in R

Implementing proposed AUM loss in mlr3torch framework

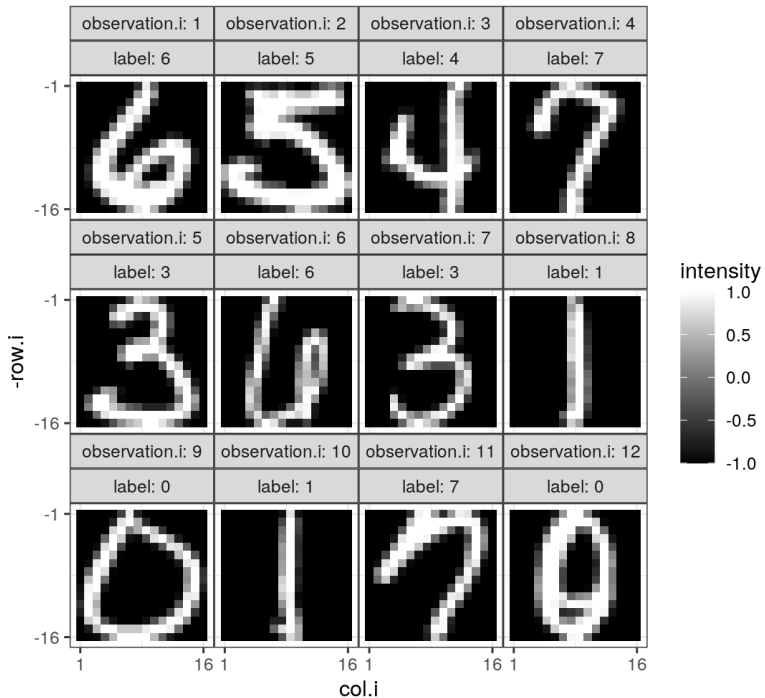
Conclusion

Advantages of torch

`library(torch)` in R provides two key features:

- ▶ Automatic gradient computation (auto-grad), which makes it easy to implement neural network learning, after having defined the network structure and loss function.
- ▶ Easy speedups using GPUs (if your model and data can fit into GPU memory).

R package has same features as python version of torch, but with less documentation and example code.



Representation of digits in CSV

- ▶ Each image/observation is one row.
- ▶ First column is output/label/class to predict.
- ▶ Other 256 columns are inputs/features (pixel intensity values).

Data from

<https://web.stanford.edu/~hastie/ElemStatLearn/datasets/zip.train.gz>

```
1:  6 -1 -1  ... -1.000 -1.000  -1
2:  5 -1 -1  ... -0.671 -0.828  -1
3:  4 -1 -1  ... -1.000 -1.000  -1
4:  7 -1 -1  ... -1.000 -1.000  -1
5:  3 -1 -1  ... -0.883 -1.000  -1
6:  6 -1 -1  ... -1.000 -1.000  -1
...
```

Demo: reading CSV, plotting digits,

<https://github.com/tdhock/2023-res-baz-az/blob/main/2023-04-19-deep-learning.Rmd>

Converting R data to torch tensors

Use array function with all columns except first as data.

```
zip.dt <- data.table::fread("zip.train.gz")
zip.X.array <- array(
  data = unlist(zip.dt[,-1]),
  dim = c(nrow(zip.dt), 1, 16, 16))
zip.X.tensor <- torch::torch_tensor(zip.X.array)
zip.y.tensor <- torch::torch_tensor(
  zip.dt$V1+1L, torch::torch_long())
```

Need to specify dimensions of input/X array:

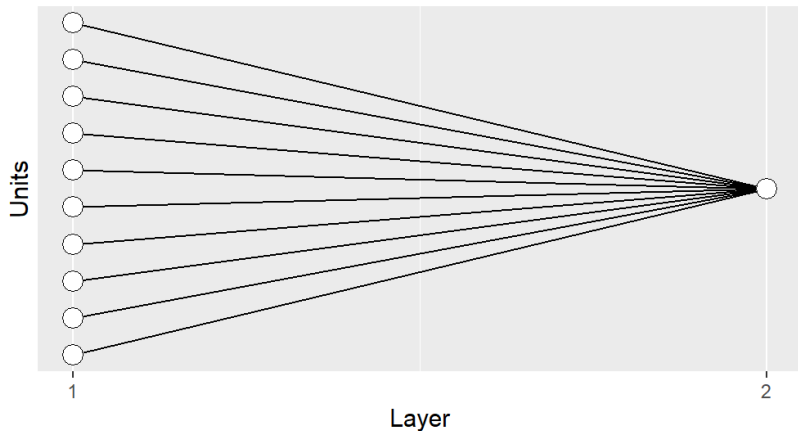
- ▶ Observations: same as the number of rows in the CSV table.
- ▶ Channels: 1 (greyscale image, would be 3 for RGB image).
- ▶ Pixels wide: 16.
- ▶ Pixels high: 16.

For output/y need to add 1 in R, and specify long int type.

Network diagram for linear model with 10 inputs/features

Neural network diagrams show how each unit (node) is computed by applying the weights (edges) to the values of the units at the previous layer.

Number of units: 10,1



Linear model R code

```
n.features <- 16*16
n.classes <- 10
linear.model <- torch::nn_sequential(
  torch::nn_flatten(),
  torch::nn_linear(n.features, n.classes))
pred.tensor <- linear.model(zip.X.tensor)
```

- ▶ First layer must specify shape of inputs (here 16x16x1).
- ▶ `nn_flatten` converts any shape to a single dimension of units (here, convert each image from 1x16x16-array to 256-vector).
- ▶ `nn_linear` uses all units/features in the previous layer (256) to predict each unit in the next layer (10).
- ▶ There are ten possible classes for an output.

Computing loss, gradient, parameter updates

```
loss.fun <- torch::nn_cross_entropy_loss()
loss.tensor <- loss.fun(pred.tensor, zip.y.tensor)
step.size <- 0.1#also known as learning rate.
optimizer <- torch::optim_sgd(
  linear.model$parameters, lr=step.size)
optimizer$zero_grad()
loss.tensor$backward()
optimizer$step()
```

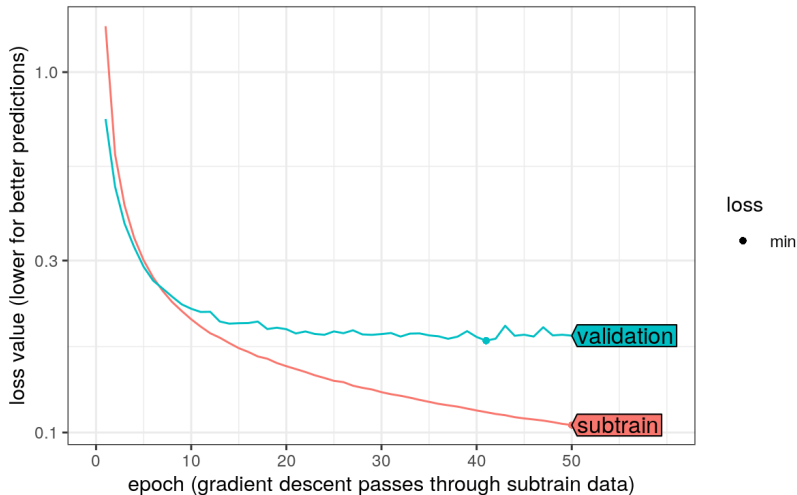
- ▶ `loss.fun` is the cross-entropy loss for multi-class classification, which is directly optimized/minimized in each iteration of the gradient descent learning algorithm.
- ▶ `optimizer` is the version of the gradient descent learning algorithm to use.
- ▶ `backward` method computes gradients.
- ▶ `step` method updates model parameters based on gradients.

Gradient Descent learning algorithm

```
gradient_descent <-  
  function(index.list, model, n_epochs, gradient.set){  
    loss.dt.list <- list()  
    for(epoch in seq(1, n_epochs)){  
      take_steps(index.list[[gradient.set]], model)  
      epoch.loss.dt <- loss_each_set(index.list, model)  
      loss.dt.list[[paste(epoch)]] <-  
        data.table(epoch, epoch.loss.dt)  
    }  
    rbindlist(loss.dt.list)  
  }
```

- ▶ `take_steps` sub-routine updates model parameters.
- ▶ `loss_each_set` computes loss and error rate on gradient set and held-out set.

Linear model

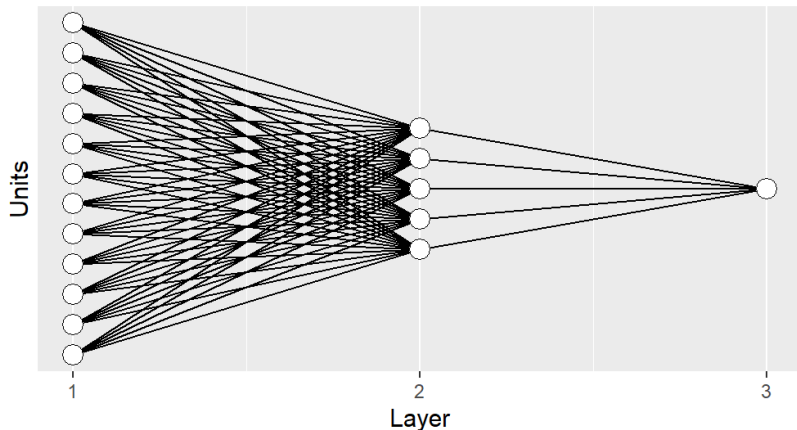


Demo: splitting data, gradient descent loop.

Network diagram for one hidden layer

Neural network diagrams show how each unit (node) is computed by applying the weights (edges) to the values of the units at the previous layer.

Number of units: 12,5,1



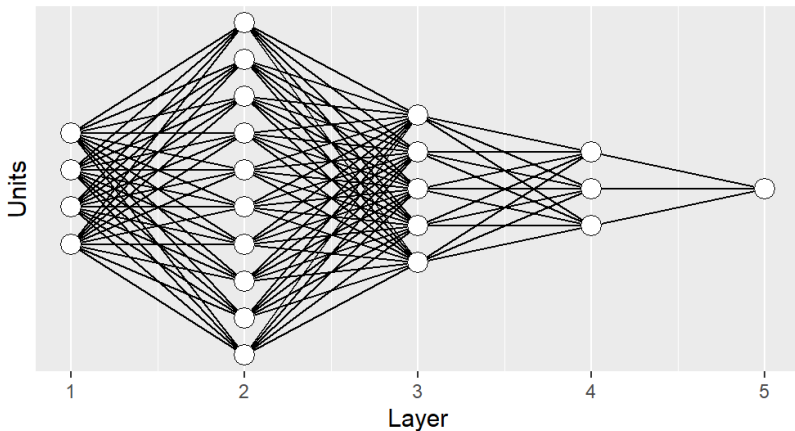
Dense (fully connected) neural network R code

```
n.hidden.units <- 50
one.hidden.layer <- torch::nn_sequential(
  torch::nn_flatten(),
  torch::nn_linear(n.features, n.hidden.units),
  torch::nn_relu(),
  torch::nn_linear(n.hidden.units, n.classes))
```

Network diagram for multiple hidden layers

Neural network diagrams show how each unit (node) is computed by applying the weights (edges) to the values of the units at the previous layer.

Number of units: 4,10,5,3,1

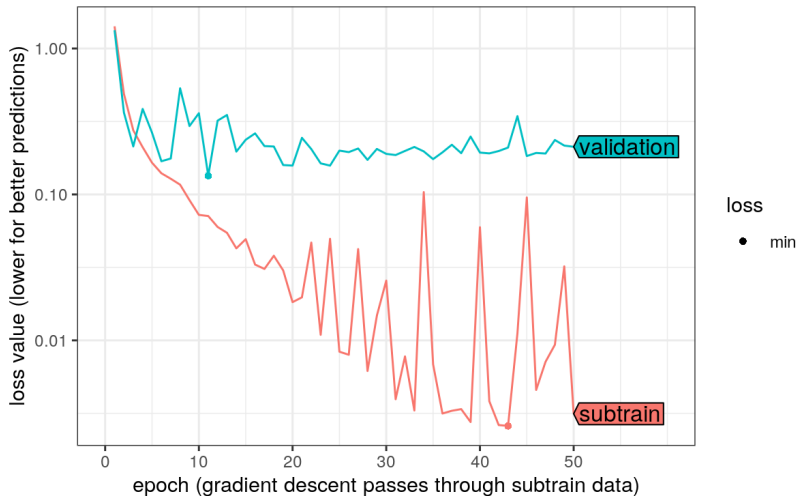


Use for loop to implement multiple hidden layers

```
new_fully_connected_units <- function(units.per.layer){  
  seq.args <- list(torch::nn_flatten())  
  for(output.i in seq(2, length(units.per.layer))){  
    input.i <- output.i-1  
    seq.args[[length(seq.args)+1]] <- torch::nn_linear(  
      units.per.layer[[input.i]],  
      units.per.layer[[output.i]])  
    if(output.i<length(units.per.layer)){  
      seq.args[[length(seq.args)+1]] <- torch::nn_relu()  
    }  
  }  
  do.call(torch::nn_sequential, seq.args)  
}
```

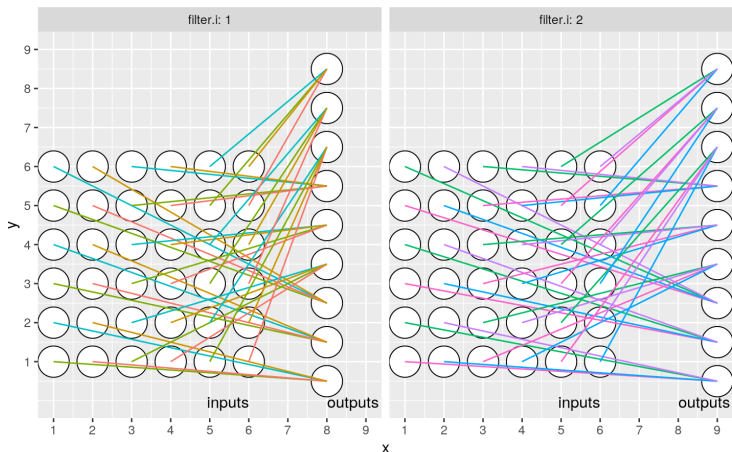
- ▶ input a vector of units per layer, for example `c(256,1000,100,10)`.
- ▶ Begin with flatten.
- ▶ Linear followed by relu in each layer except last.

Dense (fully connected) neural network with 8 hidden layers



2D convolutional kernel for 6x6 pixel image, kernel size=2

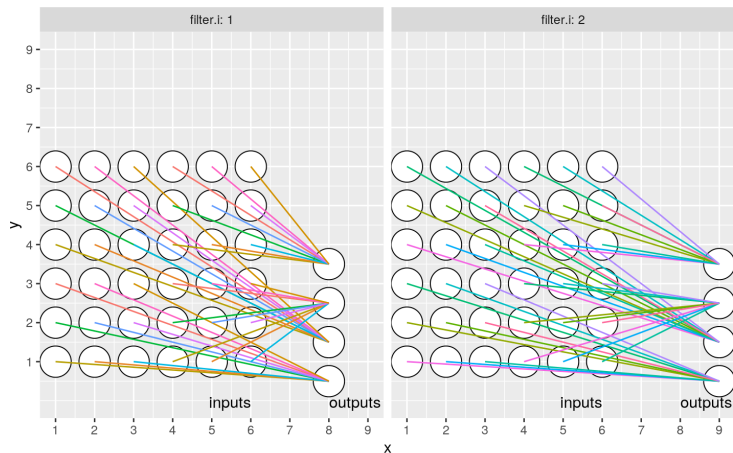
6x6 pixel image input, convolutional kernel size = stride = 2,
n.filters/channels = 2, weights per channel = 4, outputs per channel = 9



```
torch.nn_conv2d(  
    in_channels = 1, out_channels = 2, kernel_size = 2)
```

2D convolutional kernel for 6x6 pixel image, kernel size=3

6x6 pixel image input, convolutional kernel size = stride = 3,
n.filters/channels = 2, weights per channel = 9, outputs per channel = 4



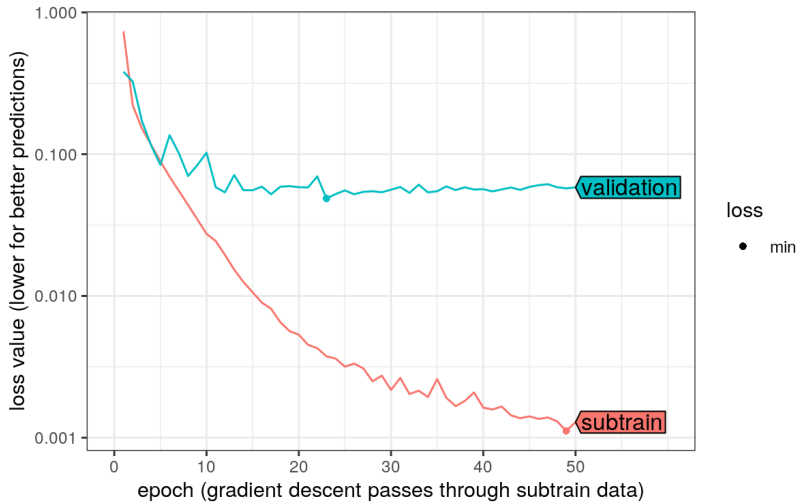
```
torch.nn_conv2d(  
    in_channels = 1, out_channels = 2, kernel_size = 3)
```

Convolutional model R code

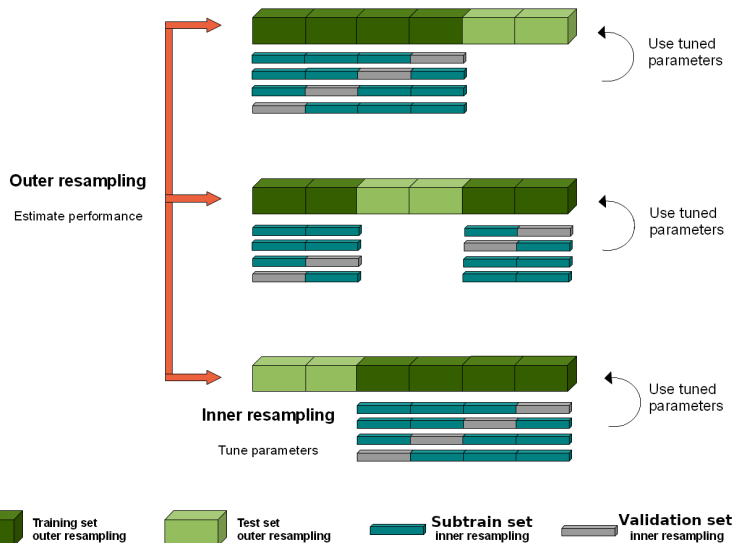
```
seq2flat <- torch::nn_sequential(  
  torch::nn_conv2d(  
    in_channels = 1, out_channels = 2, kernel_size = 3),  
  torch::nn_relu(),  
  torch::nn_flatten(),  
  torch::nn_linear(conv.hidden.units, last.hidden.units),  
  torch::nn_relu(),  
  torch::nn_linear(last.hidden.units, n.classes))
```

- ▶ Two hidden layers: one convolutional, one linear.
- ▶ Sparse: few inputs are used to predict each unit in `nn_conv2d`.
- ▶ Exploits structure of image data to make learning easier/faster.

Convolutional neural network

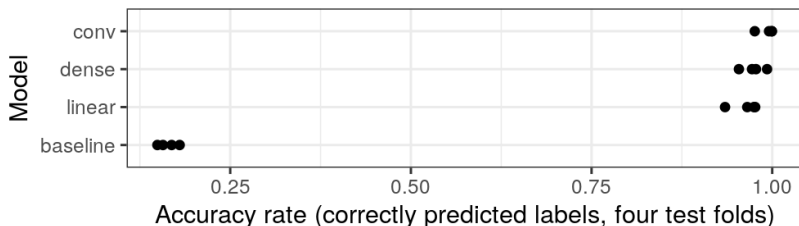


Two kinds of cross-validation must be used



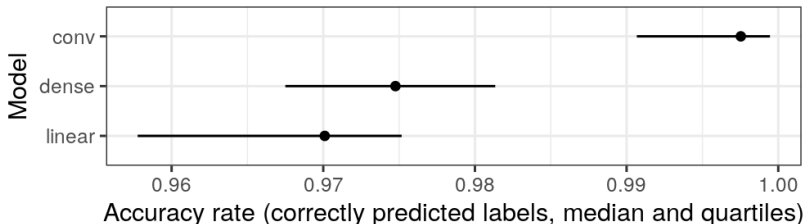
Source: https://mlr.mlr-org.com/articles/tutorial/nested_resampling.html

Accuracy rates for each test fold



- ▶ Always a good idea to compare with the trivial/featureless baseline model which always predicts the most frequent class in the train set. (ignoring all inputs/features)
- ▶ Here we see that the featureless baseline is much less accurate than the three learned models, which are clearly learning something non-trivial.
- ▶ Code for test accuracy figures:
<https://github.com/tdhock/2023-res-baz-az/blob/main/figure-test-accuracy.R>

Zoom to learned models



- ▶ Dense neural network slightly more accurate than linear model, convolutional significantly more accurate than others.
- ▶ Conclusion: convolutional neural network should be preferred for most accurate predictions in these data.
- ▶ Maybe not the same conclusion in other data sets, with the same models. (always need to do cross-validation experiments to see which model is best in any given data set)
- ▶ Maybe other models/algorithms would be even more accurate in these data. (more/less layers, more/less units, completely different algorithm such as random forests, boosting, etc)

Unbalanced classification, AUC, and proposed AUM

Basics of neural networks with torch in R

Implementing proposed AUM loss in mlr3torch framework

Conclusion

Computing loss and gradient descent updates in torch

```
loss.fun <- torch::nn_cross_entropy_loss() #multi-class.  
loss.fun <- torch::nn_bce_with_logits_loss() #binary.  
loss.fun <- Proposed_AUM #ROC optimization.
```

```
## Then the rest of the code is the same:  
loss.tensor <- loss.fun(pred.tensor, zip.y.tensor)  
step.size <- 0.1#also known as learning rate.  
optimizer <- torch::optim_sgd(  
  linear.model$parameters, lr=step.size)  
optimizer$zero_grad()  
loss.tensor$backward()  
optimizer$step()
```

ROC curve R torch code uses argsort

```
ROC_curve <- function(pred_tensor, label_tensor){
  sorted_indices = torch_argsort(-pred_tensor$flatten())
  ... # $cumsum() $diff() etc.
  list(FPR=FPR, FNR=FNR, TPR=1 - FNR,
       "min(FPR,FNR)"=torch_minimum(FPR, FNR),
       min_constant=torch_cat(c(torch_tensor(-Inf), uniq_thresh)),
       max_constant=torch_cat(c(uniq_thresh, torch_tensor(Inf))))
}
> L <- ROC_curve(torch_tensor(c(2,-3.5,-1,1.5)),
+               torch_tensor(c(0, 0, 1, 1)))
> data.frame(lapply(L, torch::as_array), check.names=FALSE)
  FPR FNR TPR min.FPR.FNR. min_constant max_constant
1 0.0 1.0 0.0          0.0          -Inf          -2.0
2 0.5 1.0 0.0          0.5          -2.0          -1.5
3 0.5 0.5 0.5          0.5          -1.5           1.0
4 0.5 0.0 1.0          0.0           1.0           3.5
5 1.0 0.0 1.0          0.0           3.5           Inf
```

<https://tdhock.github.io/blog/2024/auto-grad-overhead/>

R code for AUC and proposed AUM both use ROC curve

```
ROC_AUC <- function(pred_tensor, label_tensor){  
  roc = ROC_curve(pred_tensor, label_tensor)  
  FPR_diff = roc$FPR[2:N]-roc$FPR[1:-2]  
  TPR_sum = roc$TPR[2:N]+roc$TPR[1:-2]  
  torch_sum(FPR_diff*TPR_sum/2.0)  
}  
  
Proposed_AUM <- function(pred_tensor, label_tensor){  
  roc = ROC_curve(pred_tensor, label_tensor)  
  min_FPR_FNR = roc[["min(FPR,FNR)"]][2:-2]  
  constant_diff = roc$min_constant[2:N]$diff()  
  torch_sum(min_FPR_FNR * constant_diff)  
}
```

Can be used for ROC optimization in binary classification, instead of `torch::nn_bce_with_logits_loss!`

<https://tdhock.github.io/blog/2024/auto-grad-overhead/>

Advantages of mlr3 framework

- ▶ Provides standard interface to many popular learning algorithms in other R packages. (rpart, glmnet, torch, ...)
- ▶ Reference implementations of standard algorithms like cross-validation.
- ▶ Makes it easy to code benchmarks (no for loop necessary) to compare prediction accuracy of different algorithms, on different data sets, ...
- ▶ Easy parallelization of benchmarks on super-computer clusters. (100–1000x speedups)

Creating unbalanced MNIST data to demonstrate learning with AUM loss

- ▶ MNIST data sets are images with ten classes (0–9).
- ▶ Convert to binary problem: even versus odd.
- ▶ Two subsets: balanced (50% odd, 50% even), unbalanced (99% odd, 1% even).
- ▶ Can AUM loss be used for learning in unbalanced data?

	subset	target_prop	0	1	2	3
1:	balanced	0.01	3568	3939	3613	3570
2:	unbalanced	0.01	36	3938	37	3571

	4	5	6	7	8	9
1:	3528	3157	3554	3646	3528	3479
2:	35	3156	36	3647	35	3479

<https://tdhock.github.io/blog/2025/unbalanced/>

Data set is Task in mlr3

```
task_MNIST <- mlr3::TaskClassif$new(  
  "MNIST", task_data_table, target="binary_label")  
task_MNIST$col_roles$stratum <- "original_y"  
task_MNIST$col_roles$subset <- "balanced_or_not"  
task_MNIST$col_roles$feature <- as.character(0:783)
```

<https://tdhock.github.io/blog/2024/mlr3torch/>

Define AUM loss module sub-class

```
AUM_loss_module <- torch::nn_module(  
  "nn_AUM_loss",  
  inherit = torch::nn_mse_loss,  
  initialize = function() {  
    super$initialize()  
  },  
  forward = Proposed_AUM  
)
```

<https://tdhock.github.io/blog/2024/mlr3torch/>

Defining a linear model in mlr3torch

Number of epochs fixed at 400.

```
pipe_op_list <- list(  
  mlr3torch::PipeOpTorchIngressNumeric$new(),  
  mlr3torch::nn("linear", out_features=1),  
  mlr3pipelines::po("torch_loss", AUM_loss_module),  
  mlr3pipelines::po(  
    "torch_optimizer",  
    mlr3torch::t_opt("sgd", lr=0.1)),  
  mlr3pipelines::po(  
    "torch_model_classif",  
    batch_size = 100000,  
    predict_type="prob",  
    n.epochs=400))
```

<https://tdhock.github.io/blog/2024/mlr3torch/>

Computing subtrain/validation AUC at each epoch

```
measure_list = mlr3::msrs(c("classif.auc", "classif.acc"))
pipe_op_list = list(...)
  mlr3pipelines::po("torch_callbacks",
    mlr3torch::t_clbk("history")),
mlr3pipelines::po(
  "torch_model_classif",
  batch_size = 100000, patience=400,
  predict_type="prob",
  measures_train=measure_list,
  measures_valid=measure_list,
  n.epochs=paradox::to_tune(
    upper = 400, internal = TRUE)))
graph_obj = Reduce(
  mlr3pipelines::concat_graphs, pipe_op_list)
learner_obj = mlr3::as_learner(graph_obj)
mlr3::set_validate(learner_obj, validate = 0.5)
```

<https://tdhock.github.io/blog/2024/mlr3torch/>

Auto-tuner learns best number of epochs

```
pipe_op_list = list(...  
  mlr3pipelines::po(  
    "torch_model_classif",  
    measures_valid=measure_list,  
    n.epochs=paradox::to_tune(  
      upper = 400, internal = TRUE))  
  ...  
learner_auto = mlr3tuning::auto_tuner(  
  learner = learner_obj,  
  tuner = mlr3tuning::tnr("internal"),  
  resampling = mlr3::rsmp("insample"),  
  measure = mlr3::msr("internal_valid_score"),  
  term_evals = 1,  
  store_models = TRUE)
```

<https://tdhock.github.io/blog/2024/mlr3torch/>

Create a learner list and benchmark grid

Grid combines a set of learners, tasks, and cross-validation splits.

```
learner_list <- list(  
  mlr3learners::LearnerClassifCVGlmnet$new(),  
  mlr3::LearnerClassifFeatureless$new(),  
  learner_auto)  
SOAK <- mlr3resampling::ResamplingSameOtherSizesCV$new()  
(bench.grid <- mlr3::benchmark_grid(  
  list(task_MNIST),  
  learner_list,  
  SOAK))
```

<https://tdhock.github.io/blog/2024/mlr3torch/>

Run benchmark on your local computer

Uses all available CPUs on your computer: each combination of learner, task, and cross-validation split is run in parallel.

```
if(require(future))plan("multisession")  
bench.result <- mlr3::benchmark(  
  bench.grid, store_models = TRUE)
```

<https://tdhock.github.io/blog/2024/mlr3torch/>

Run benchmark on super-computer cluster

I typically use SLURM, which requires definition of how much time/memory to reserve per learner/task/split.

```
batchtools::makeExperimentRegistry(  
  file.dir = reg.dir, seed = 1,  
  packages = "mlr3verse")  
mlr3batchmark::batchmark(  
  bench.grid, store_models = TRUE)  
job.table <- batchtools::getJobTable()  
chunks <- data.frame(job.table, chunk=1)  
batchtools::submitJobs(chunks, resources=list(  
  walltime = 60*60*24,#seconds  
  memory = 8000,#megabytes per cpu  
  chunks.as.arrayjobs=TRUE))
```

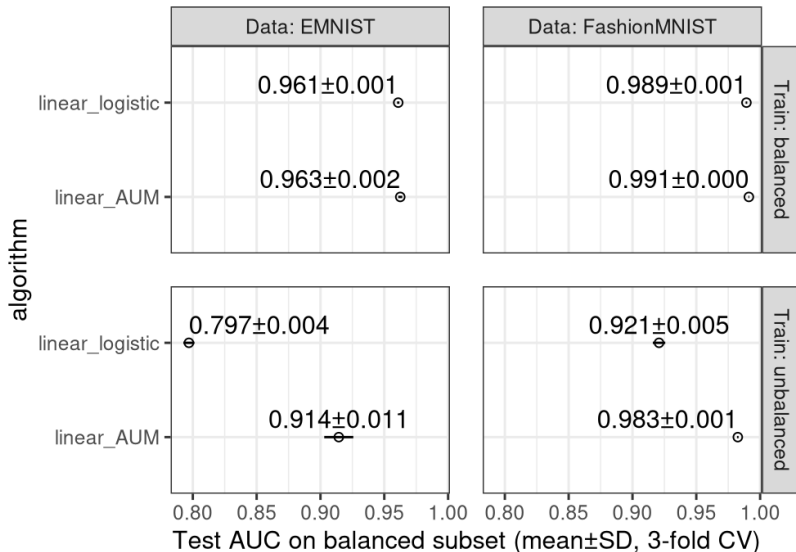
<https://tdhock.github.io/blog/2024/mlr3torch/>

Get results from super-computer cluster

```
jobs.after <- batchtools::getJobTable(reg=reg)
ids <- jobs.after[is.na(error), job.id]
keep_history <- function(x){
  learners <- x$learner_state$model$marshaled[[
    "tuning_instance"]$archive$learners
  x$learner_state$model <- if(is.function(learners)){
    L <- learners(1)[[1]]
    x$history <- L$model$torch_model_classif[[
      "model"]$callbacks$history
  }
  x
}
bench.result <- mlr3batchmark::reduceResultsBatchmark(
  ids, fun=keep_history)
```

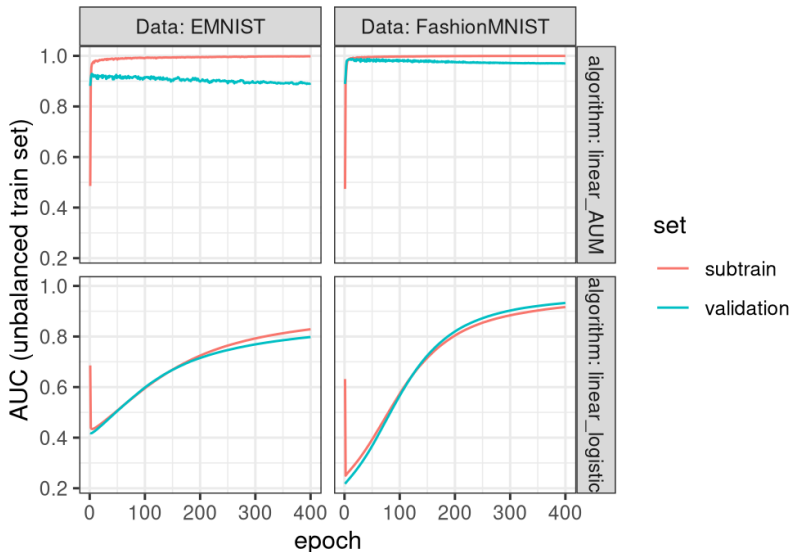
<https://tdhock.github.io/blog/2024/mlr3torch/>

Test AUC of linear models



- ▶ No difference for balanced training.
- ▶ AUM loss has larger test AUC using unbalanced training.

Subtrain/validation AUC of linear models



AUM loss reaches max validation AUC much faster than logistic.

Unbalanced classification, AUC, and proposed AUM

Basics of neural networks with torch in R

Implementing proposed AUM loss in mlr3torch framework

Conclusion

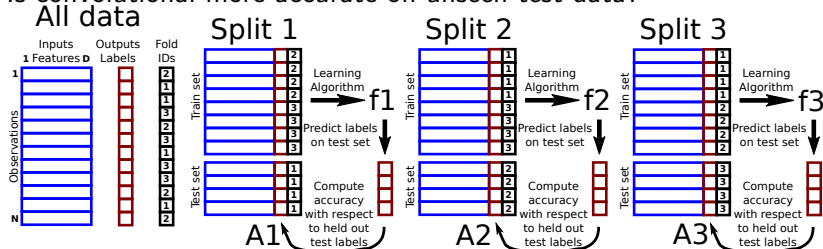
Thanks for participating!

- ▶ Contact: toby.dylan.hocking@usherbrooke.ca, toby.hocking@r-project.org
- ▶ ROC curves and AUC are used to fairly evaluate binary classification algorithms.
- ▶ Minimizing proposed AUM loss results in ROC optimization.
- ▶ Proposed AUM loss can be implemented in torch in R.
- ▶ Advantages of mlr3 framework include parallelization, simplified code (fewer nested loops).



K-fold cross-validation for model evaluation

Is convolutional more accurate on unseen test data?



- ▶ Randomly assign a fold ID from 1 to K to each observation.
- ▶ Hold out the observations with the Split ID as test set.
- ▶ Use the other observations as the train set.
- ▶ Run learning algorithm on train set (including hyper-parameter selection), outputs learned function (f1-f3).
- ▶ Finally compute and plot the prediction accuracy (A1-A3) with respect to the held-out test set.