# Introduction to Machine Learning

Ismaël Lajaaiti

06/03/2022

Inspired from this tutorial (https://torch.mlverse.org/start/guess_the_correlation/).

# Set up

Installing the Machine Learning framework.

```
installed.pkg = installed.packages()[,1] # installed packages on your system

# If not installed, download the required packages.
required_pkg = c("torch", "torchvision", "luz") # list of required packages
for (pkg in required_pkg) {
  if (!is.element(pkg, installed.pkg)){install.packages(pkg)}
}
```

In addition download the dataset we will be using.

```
remotes::install_github("mlverse/torchdatasets")
```

```
## Skipping install of 'torchdatasets' from a github remote, the SHA1 (b33dbe2f) has no
t changed since last install.
##   Use `force = TRUE` to force installation
```

Load libraries.

```
library("torch")
library("torchvision")
library("luz")
library("torchdatasets")
```

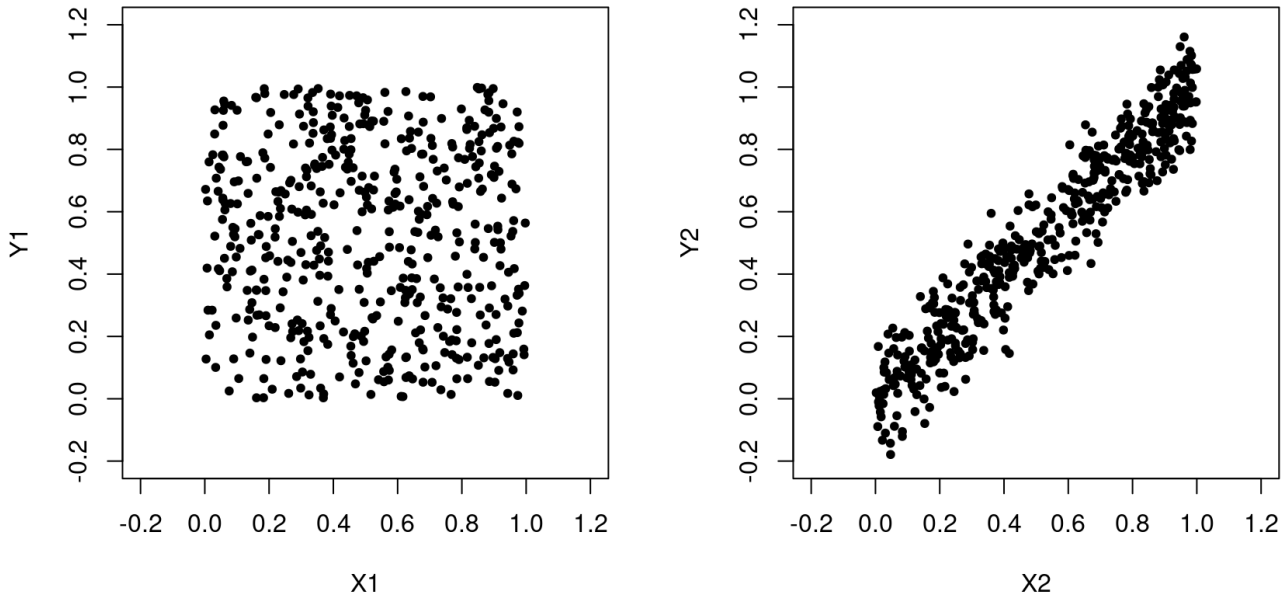
Quick test to check if torch is installed correctly:

```
torch_tensor(1)
## torch_tensor
##  1
## [ CPUFloatType{1} ]
```

# Prepare the dataset

*Guess the correlation* is a dataset that asks one to guess the correlation between two variables from their scatter plot.

For example can you guess the correlation between the variables displayed in the scatter plots below?




The answer is…

```
cat("cor(X1,Y1) = ", cor(X1,Y1))
## cor(X1,Y1) =  -0.04491401
cat("cor(X2,Y2) = ", cor(X2,Y2))
## cor(X2,Y2) =  0.9558691
```

This dataset contains contains 150,000 observations, where each observation contains: a scatter plot (input) and the corresponding correlation (output). As the dataset is large, to reduce the training time of the neural network we will restrict ourselves to a subset.

```
train_ind = 1:10000 # 10,000 observations for training
valid_ind = 10001:15000 # 5,000 observations for validation
test_ind  = 15001:20000 # 5,000 observations for testing
```

Let's download the data and pick the indices for the train set.

```
root     = file.path(tempdir(), "correlation")
train_ds = guess_the_correlation_dataset(root = root, indexes = train_ind,
                                         download = T)
```

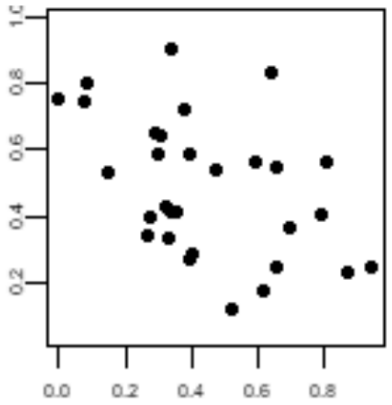
First, we will have a look at the first observation to see how it looks like. Each observation has two parts:

- `$x` = the input data, here the scatter plot
- `$y` = the expected output (or the 'label'), here the correlation

```
first_element = train_ds[1]
input  = first_element$x # torch tensor (150,150) = scatter plot
output = first_element$y # torch tensor (1,) = correlation coef.
```

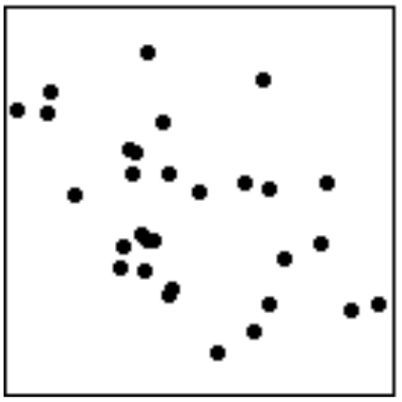
The scatter plot is encoded as a tensor, but we can plot it as follow:

```
input %>% as.array() %>% as.raster() %>% plot()
```




The scatter plot includes the axis that are unnecessary to infer the correlation, then let's remove them by cropping the picture.

```
cropped_input = input[0:130, 21:150] # input without axis
cropped_input %>% as.array() %>% as.raster() %>% plot() # plot to check
```



We have already create the training set above ( `train_ds` ). In addition, we create the validation and test sets. As the data was already download before, we do not need to download it again. We just pick different indices.

```
# Validation set.
valid_ds = guess_the_correlation_dataset(root = root, indexes = valid_ind,
    download = F)

# Test set.
test_ds = guess_the_correlation_dataset(root = root, indexes = test_ind,
    download = F)
```

Quick check of the size of our train, validation and test sets

```
length(train_ds)
## [1] 10000
length(valid_ds)
## [1] 5000
length(test_ds)
## [1] 5000
```

To make the neural network training faster, we will work with *batches*. A batch is a subset that contains a bunch of observations (here 64). During the training, instead of going over each observation individually, the neural network will process simultaneously the observations belonging to the same batch. In the torch framework, this process of splitting each set in batches is handled by the objects called `dataloader` .

```
train_dl = dataloader(train_ds, batch_size = 64, shuffle = T)
valid_dl = dataloader(valid_ds, batch_size = 64, shuffle = F)
test_dl  = dataloader(test_ds , batch_size = 64, shuffle = F)
```

The length of the dataloader is equal to its number of batches. That latter can be recovered by dividing the number of observations in the dataset by the batch size (here 64).

```
length(train_dl)
## [1] 157
ceiling(length(train_ds) / 64)
## [1] 157
```

Let's have look to the first batch.

```
first_batch <- dataloader_make_iter(train_dl) %>% dataloader_next()
dim(first_batch$x) # 64 first scatter plots (64, 150, 150)
## [1]  64 150 150
dim(first_batch$y) # 64 first correlations (64,)
## [1] 64
```

Our data is well-prepared and ready to be given to our neural network. Then we can now build the neural network.

# Build the neural network

In the torch framework, a neural network is defined by two functions:

1. `initialize` which describes the structure of the network i.e. the number of layers and the number of neurons per layer
2. `forward` which tells how the data goes through the different layers previously defined

```
torch_manual_seed(113) # set seed for reproducibility

# Create neural network.
dnn <- nn_module(

  "corr-dnn", # network name

  # Define layers.
  initialize = function(n_in, n_hidden, n_out) {
    self$fc1 <- nn_linear(in_features = n_in    , out_features = n_hidden)
    self$fc2 <- nn_linear(in_features = n_hidden, out_features = n_hidden)
    self$fc3 <- nn_linear(in_features = n_hidden, out_features = n_hidden)
    self$fc4 <- nn_linear(in_features = n_hidden, out_features = n_out)
  },

  # How input goes through the network.
  forward = function(x) {
    x %>% # input
      self$fc1() %>% # 1st layer
      nnf_relu() %>% # activation function

      self$fc2() %>% # 2nd layer
      nnf_relu() %>% # activation function

      self$fc3() %>% # etc.
      nnf_relu() %>% # ...

      self$fc4()     # output
  }
)
```

The `initialize` function takes three arguments:

1. `n_in` : the size of the input, here the number of pixels of the scatter plot, we flatten the 2d picture of size (height, width) into a 1d vector of size (width*height)
2. `n_hidden` : the number of neurons in the hidden layers
3. `n_out` : the size of output, here 1 as the output is the correlation

Arguments can be added, deleted or replaced depending on your needs. Here is just a simple example that fits most situations.

```
height = dim(cropped_input)[1] # height of the picture
width  = dim(cropped_input)[2] # width
n_in  = height * width # total number of pixels
n_out = 1 # expect one value in output: the correlation
n_hidden = 100 # play with this value to optimize the network
```

We can call the function and create our neural network, and check its structure.

```
network = dnn(n_in, n_hidden, n_out)
opt = optim_adam(params = network$parameters) # for param. opt. during training
network
## An `nn_module` containing 1,710,401 parameters.
##
## ── Modules ──────────────────────────────────────────────────────────────────
## • fc1: <nn_linear> #1,690,100 parameters
## • fc2: <nn_linear> #10,100 parameters
## • fc3: <nn_linear> #10,100 parameters
## • fc4: <nn_linear> #101 parameters
# ---------------------------------------
# WEIGHTS           + BIAS    = PARAMETERS
# ---------------------------------------
# node_in*n_out     + n_out
n_in     * n_hidden + n_hidden
## [1] 1690100
n_hidden * n_hidden + n_hidden
## [1] 10100
n_hidden * n_hidden + n_hidden
## [1] 10100
n_hidden * n_out    + n_out
## [1] 101
```

The network is ready to be trained.

# Training loop

One iteration of the training loop is called an *epoch* and can be summarized as follow:

1. Training over the batches of the training set
    a. Make prediction on a batch
    b. Compute the prediction error
    c. Change weight to reduce the prediction error
    d. Repeat for next batch, until the end of train set…
2. Validation over the batches of the validation set
    a. Make prediction on a batch
    b. Compute prediction error
    c. Repeat for next batch, until the end of validation set…
3. Early stopping check
    a. If error has decreased continue the training
    b. Else stop the training

To build that loop we specify first how the network should process batches during the training step…

```
trainStep = function(network, batch){
  opt$zero_grad()
  batch_size = dim(batch$x)[1]
  input = batch$x
  input = input[1:batch_size, 0:130, 21:150] # crop axis
  input = input$reshape(c(batch_size, 130*130)) # flatten input
  output.pred = network(input)$squeeze(2) # correlation predicted by the DNN
  output.true = batch$y # expected correlation
  loss = nnf_mse_loss(output.pred, output.true) # error between pred and true
  loss$backward() # backward propagation
  opt$step()
  loss$item() # get value
}
```

… then during the validation step…

```
validStep = function(network, batch){
  batch_size = dim(batch$x)[1]
  input = batch$x
  input = input[1:batch_size, 0:130, 21:150] # crop axis
  input = input$reshape(c(batch_size, 130*130)) # flatten input
  output.pred = network(input)$squeeze(2) # correlation predicted by the DNN
  output.true = batch$y # expected correlation
  loss = nnf_mse_loss(output.pred, output.true) # error between pred and true
  loss$item() # get value
}
```

… and now we can assemble the pieces.

```
trainingLoop = function(network, train_dl, valid_dl,
                        epoch_max = 5, patience = 2){

  # Initialize variables.
  epoch = 1
  count = 0
  loss.previous = 100

  # Training loop.
  while(epoch <= epoch_max & count < patience){

    # Train.
    network$train()
    Loss.train = c()
    coro::loop(for (batch in train_dl){
      loss.train = trainStep(network, batch) # go over the batch
      Loss.train = c(Loss.train, loss.train) # save loss value
    })
    cat(sprintf("Epoch %0.3d/%0.3d - Train - Loss = % 3.5f \n",
                epoch, epoch_max, mean(Loss.train))) # progress

    # Validation.
    network$eval()
    Loss.valid = c()
    coro::loop(for (batch in valid_dl){
      loss.valid = validStep(network, batch) # go over the batch
      Loss.valid = c(Loss.valid, loss.valid) # save loss value
    })
    cat(sprintf("Epoch %0.3d/%0.3d - Valid - Loss = % 3.5f \n",
                epoch, epoch_max, mean(Loss.valid))) # progress

    # Check for early stopping.
    loss.now = mean(Loss.valid) # mean loss at the current epoch
    if (loss.now > loss.previous){ # if loss has increase, increment counter
      count = count + 1}else{ # else, reset counter and loss baseline value
      count = 0
      loss.previous = loss.now
    }

    epoch = epoch + 1 # next epoch

  }
}
```

Training the network (it can take some times…)

```
trainingLoop(network, train_dl, valid_dl, epoch_max = 5)
## Epoch 001/005 - Train - Loss =  0.22488
## Epoch 001/005 - Valid - Loss =  0.17244
## Epoch 002/005 - Train - Loss =  0.11584
## Epoch 002/005 - Valid - Loss =  0.08343
## Epoch 003/005 - Train - Loss =  0.03894
## Epoch 003/005 - Valid - Loss =  0.08427
## Epoch 004/005 - Train - Loss =  0.02738
## Epoch 004/005 - Valid - Loss =  0.02900
## Epoch 005/005 - Train - Loss =  0.02200
## Epoch 005/005 - Valid - Loss =  0.01575
```

# Evaluate the network

Our network is now trained and ready to predict correlations from scatter plots it has *never* seen. That what the test set is for. Let's specify how the network is tested on one batch…

```
testStep = function(network, batch){
  batch_size = dim(batch$x)[1]
  input = batch$x
  input = input[1:batch_size, 0:130, 21:150] # crop axis
  input = input$reshape(c(batch_size, 130*130)) # flatten input
  output.pred = network(input)$squeeze(2) # correlation predicted by the DNN
  output.true = batch$y # expected correlation
  output = list(pred = output.pred, true = output.true)
  return(output)
}
```

… and then loop over all the batches of the test set.

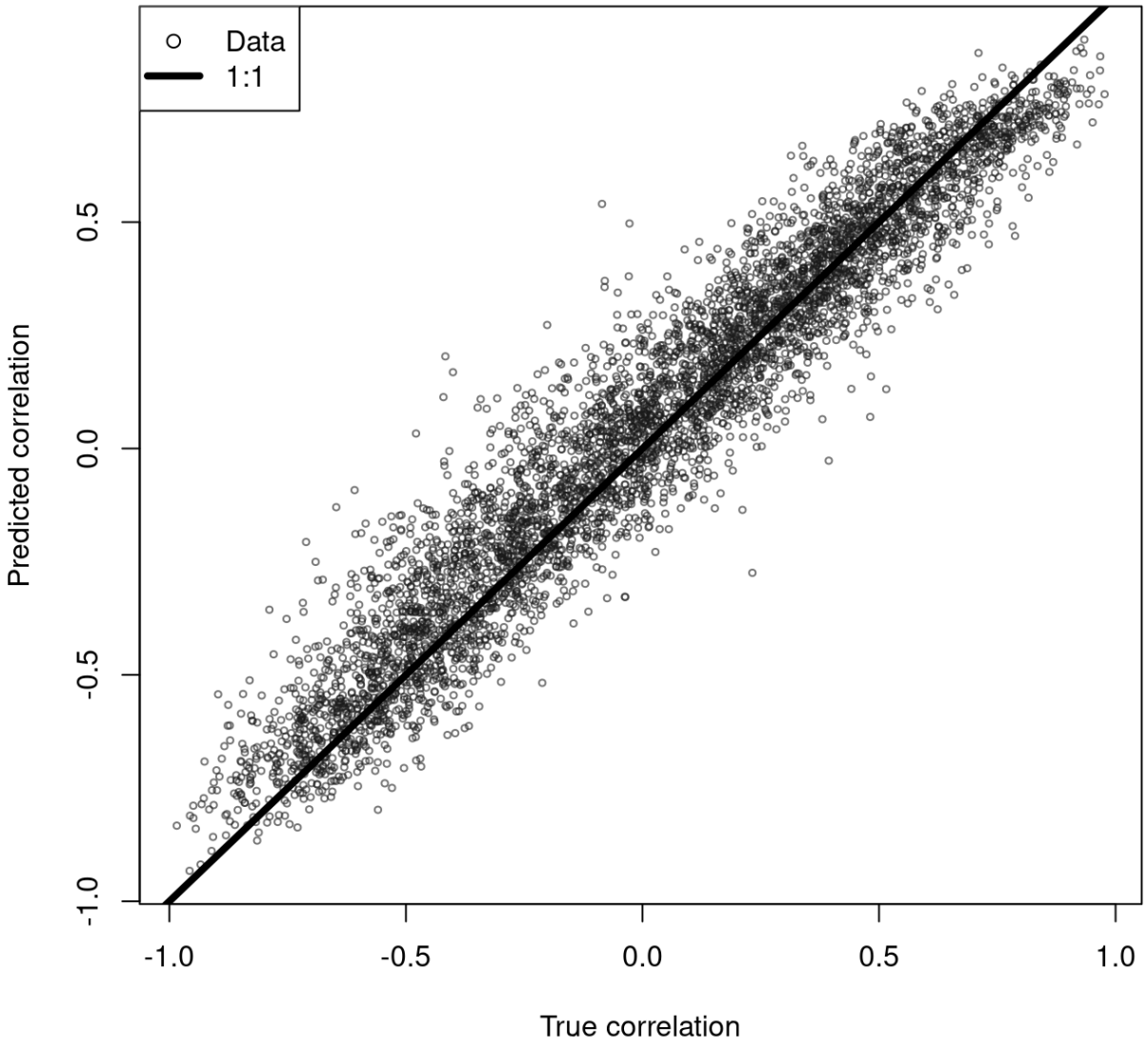
```
testLoop = function(network, test_dl){
  Pred = c() # store predicted correlations
  True = c() # store true correlations
  coro::loop(for (batch in test_dl){
      output = testStep(network, batch) # go over the batch
      Pred = c(Pred, output$pred %>% as.array()) # save pred. values
      True = c(True, output$true %>% as.array()) # save true values
    })
  out = list(true = True, pred = Pred)
  return(out)
}
```

Testing the network.

```
out = testLoop(network, test_dl)
```

The most direct way to have a quick idea of how well our network performs is to plot the predicted correlations against the true correlations.

```
# Plot Pred vs. True.
plot(out$true, out$pred, xlab = 'True correlation',
     ylab = 'Predicted correlation', cex = 0.5, col = rgb(.1,.1,.1, 0.6))
# Plot 1:1 line.
abline(0, 1, lw = 4)
# Add legend.
legend("topleft", legend = c("Data", "1:1"), lwd = c(NA, 4), pch = c(1, NA))
```




The performances of the network can be quantified more accurately (e.g. by decomposing the total error in variance and bias), but that's beyond the scope of this introduction.