/

Neural Networks Using Torch Package

Lab 14 Solutions - MATH 4322

Instructions

- You can print this out and write on this or write/type on a separate sheet. I also provide a Quarto version of this if you desire.
- Upload your answers in Canvas as you do with your homework.
- The questions are in red.
- This lab we will work on the `Hitters` data and `MNIST` data.
- We use the `luz` package, which interfaces to the `torch` package which in turn links to efficient `C++` code in the LibTorch library. Need to install these packages first before starting this lab.
- This version of the lab was produced by Daniel Falbel and Sigrid Keydana, both data scientists at **Rstudio** where these packages were produced.
- An advantage over the `keras` implementation is that this version does not require a separate `python` installation.

Setting Up Hitters Data

We initially set up the data and separate into training and test set.

```r
library(ISLR2)
Gitters <- na.omit(Hitters)
n <- nrow(Gitters)
set.seed(13)
ntest <- trunc(n / 3)
```

```r
testid <- sample(1:n, ntest)
```

Linear Regression

We will first look at the linear model and results from that to compare.

```r
lfit <- lm(Salary ~ ., data = Gitters[-testid, ])
lpred <- predict(lfit, Gitters[testid, ])
with(Gitters[testid, ], mean(abs(lpred - Salary)))
```

```
[1] 254.6687
```

Notice the use of the `with()` command: the first argument is a dataframe, and the second an expression that can refer to elements of the dataframe by name. In this instance the dataframe corresponds to the test data and the expression computes the mean absolute prediction error on this data.

Question 1: What is the test MAE?

254.6687

Question 2: Interpret this value.

The linear model will be off on on average of 254.6687 thousands of dollars.

Fitting the Single Layer Neural Network

To fit the neural network, we first set up a model structure that describes the network.

```r
library(torch)
library(luz) # high-level interface for torch
library(torchvision) # for datasets and image transformation
library(torchdatasets) # for datasets we are going to use
library(zeallot)
torch_manual_seed(13)
x <- scale(model.matrix(Salary ~ . - 1, data = Gitters))
y <- Gitters$Salary
```

- We will create a model called `modnn` by defining the `initialize()` and `forward()` functions and passing them to the `nn_module()` function.

- The `initialize()` function is responsible for initializing the submodules that are used by the model.
- In the `forward` method we implement what happens when the model is called on input data.
- In this case we use the layers we defined in `initialize()` in that specific order.
- `self` is a list-like special object that is used to share information between the methods of the `nn_module()`. When you assign an object to `self` in `initialize()`, it can then be accessed by `forward()`.
- The `pipe` operator `%>%` passes the previous term as the first argument to the next function, and returns the result.

```
modnn <- nn_module(
  initialize = function(input_size) {
    self$hidden <- nn_linear(input_size, 50)
    self$activation <- nn_relu()
    self$dropout <- nn_dropout(0.4)
    self$output <- nn_linear(50, 1)
  },
  forward = function(x) {
    x %>%
      self$hidden() %>%
      self$activation() %>%
      self$dropout() %>%
      self$output()
  }
)
```

Example Using the Pipe Operator

We want to make a matrix and each of the variables. We can do so by

```
x <- scale(model.matrix(Salary ~ . - 1, data = Gitters))
```

Compound expressions like this can be difficult to parse. We could have obtained the same result using the pipe operator:

```
x <- model.matrix(Salary ~ . - 1, data = Gitters) %>% scale()
```

Using the pipe operator makes it easier to follow the sequence of operations.

Fitting the Neural Network

- We now return to our neural network.
- The object `modnn` has a single hidden layer
  - with *50 hidden units*, and
  - a **ReLU activation** function
- It then has a dropout layer, in which a random 40% of the 50 activations from the previous layer are set to zero during each iteration of the stochastic gradient descent algorithm (SGD).
- Finally, the output layer has just one unit with no activation function, indicating that the model provides a single quantitative output.
- Next we add details to `modnn` that control the fitting algorithm.
  - We minimize squared-error loss.
  - The algorithm tracks the mean absolute error on the training data, and on validation data if it is supplied.

```
modnn <- modnn %>%
  setup(
    loss = nn_mse_loss(),
    optimizer = optim_rmsprop,
    metrics = list(luz_metric_mae())
  ) %>%
  set_hparams(input_size = ncol(x))
```

- In the previous line, the pipe operator passes `modnn` as the first argument to `setup()`.
- The `setup()` function embeds these specification into a new model object.
- We also use `set_hparam()` to specify the arguments that should be passed to the `initialize()` method of `modnn`.

Now we fit the model.

- We supply the training data and the number of `epochs`.
- By default, at each step of SGD, the algorithm randomly selects 32 training observations for the computation of the gradient.
- An epoch amounts to the number of SGD steps required to process n observations.
  - Since the training set has $n = 176$, an epoch is $176/32 = 5.5$ SGD steps. * The `fit()` function has an argument `valid_data`; these data are not used in the fitting, but can be used to track the progress of the model (in this case reporting mean absolute error).

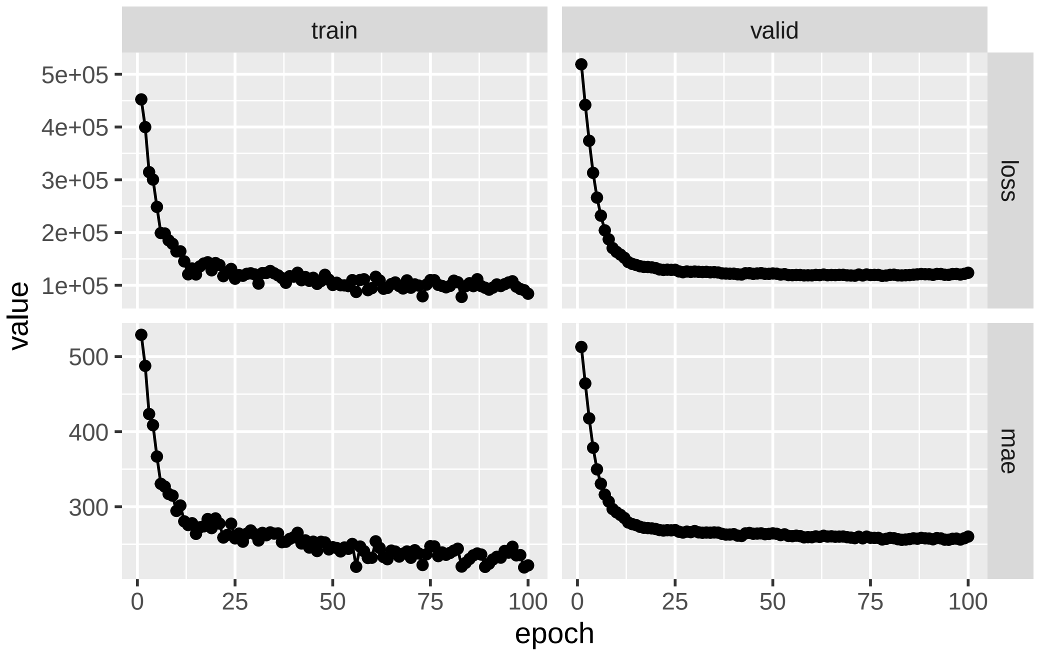
- We do supply the test data for this example so we can see mean absolute error of both the training data and test data as the epochs proceed.
- To see more options for fitting, use `?fit.luz_module_generator`.

```r
fitted <- modnn %>%
  fit(
    data = list(x[-testid, ], matrix(y[-testid], ncol = 1)),
    valid_data = list(x[testid, ], matrix(y[testid], ncol = 1)),
    epochs = 100 #100
  )
```

(Here and elsewhere we have reduced the number of epochs to make runtimes manageable; users can of course change back)

We can plot the `fitted` model to display the mean absolute error for the training and test data.

```r
plot(fitted)
```



Finally, we predict from the final model, and evaluate its performance on the test data. Due to the use of SGD, the results vary slightly with each fit.

```r
npred <- predict(fitted, x[testid, ])
mean(abs(y[testid] - as.matrix(npred)))
```

```
[1] 260.2138
```

Question 3: What is the MAE from this neural network? Compare that to the linear regression model.
MAE = 270.3537, this is higher than the linear regression model.

Question 4: Repeat this with 100 epochs. What is the MAE with 100 epochs.

MAE = 257.8595

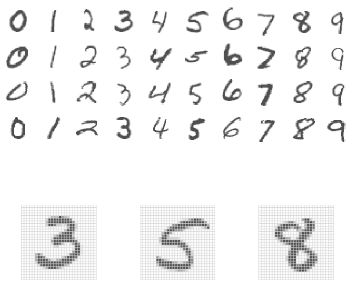
We had to convert the `npred` object to a matrix, since the current predict method returns an object of class `torch_tensor`.

```r
class(npred)
```

```
[1] "torch_tensor" "R7"
```

MNIST Dataset

- See LeCun, Cortes, and Burges (2010) "The MNIST database of handwritten digits", available at http://yann.lecun.com/exdb/mnist.



Examples of handwritten digits from the `MNIST` corpus. Each grayscale image has 28 × 28 pixels, each of which is an eight-bit number (0–255) which represents how dark that pixel is. The first 3, 5, and 8 are enlarged to show their 784 individual pixel values.

- The `torchvision` package comes with a number of example datasets, including the `MNIST` digit data.

- Our first step is to load the `MNIST` data. The `mnist_dataset()` function is provided for this purpose.
- This functions returns a `dataset()`, a data structure implemented in `torch` allowing one to represent any dataset without making assumptions on where the data is stored and how the data is organized. Usually, torch datasets also implement the data acquisition process, like downloading and caching some files on disk.

```r
train_ds <- mnist_dataset(root = ".", train = TRUE, download = TRUE)
test_ds <- mnist_dataset(root = ".", train = FALSE, download = TRUE)

str(train_ds[1])
```

```
List of 2
 $ x: int [1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
 $ y: int 6
```

```r
str(test_ds[2])
```

```
List of 2
 $ x: int [1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
 $ y: int 3
```

```r
length(train_ds)
```

```
[1] 60000
```

```r
length(test_ds)
```

```
[1] 10000
```

Question 5: How many observations are in the training set and how many are in the test?

60,000 in the training set and 10,000 in the test set.

- The images are 28×28, and stored as matrix of pixels. We need to transform each one into a vector.

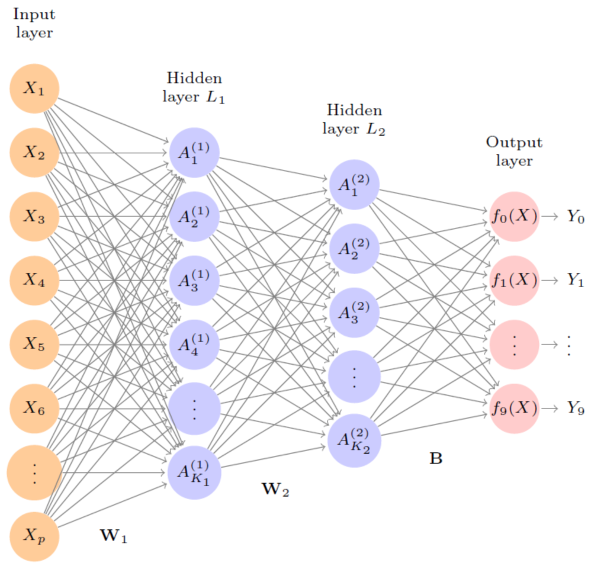
- Here the inputs are eight-bit grayscale values between 0 and 255, so we rescale to the unit interval. (Note: eight bits means 2^8, which equals 256. Since the convention is to start at 0, the possible values range from 0 to 255.)
- To apply these transformations we will re-define `train_ds` and `test_ds`, now passing a the `transform` argument that will apply a transformation to each of the image inputs.

```r
transform <- function(x) {
  x %>%
    torch_tensor() %>%
    torch_flatten() %>%
    torch_div(255)
}
train_ds <- mnist_dataset(
  root = ".",
  train = TRUE,
  download = TRUE,
  transform = transform
)
test_ds <- mnist_dataset(
  root = ".",
  train = FALSE,
  download = TRUE,
  transform = transform
)
```

Now we are ready to fit our neural network. This is a multilayer network

- It has two hidden layers L1 (256 units) and L2 (128 units) rather than one.
- It has ten output variables, rather than one. In this case the ten variables really represent a single qualitative variable and so are quite dependent. (We have indexed them by the digit class 0–9 rather than 1–10, for clarity.) More generally, in multi-task learning one can premulti- task dict different responses simultaneously with a single network; they all learning have a say in the formation of the hidden layers.
- The loss function used for training the network is tailored for the multiclass classification task.

Neural network diagram with two hidden layers and multiple outputs, suitable for the MNIST handwritten-digit problem. The input layer has $p = 784$ units, the two hidden layers $K1 = 256$ and $K2 = 128$ units respectively, and the output layer 10 units. Along with intercepts (referred to as biases in the deep-learning community) this network has $235,146$ parameters (referred to as weights).



```
modelnn <- nn_module(
  initialize = function() {
    self$linear1 <- nn_linear(in_features = 28*28, out_features = 256)
    self$linear2 <- nn_linear(in_features = 256, out_features = 128)
    self$linear3 <- nn_linear(in_features = 128, out_features = 10)

    self$drop1 <- nn_dropout(p = 0.4)
    self$drop2 <- nn_dropout(p = 0.3)

    self$activation <- nn_relu()
  },
  forward = function(x) {
    x %>%

      self$linear1() %>%
      self$activation() %>%
      self$drop1() %>%

      self$linear2() %>%
      self$activation() %>%
      self$drop2() %>%

      self$linear3()
  }
)
```

- We define the `intialize()` and `forward()` methods of the `nn_module()`.
- In `initialize` we specify all layers that are used in the model.
  - For example, `nn_linear(784, 256)` defines a dense layer that goes from $28 \times 28 = 784$ input units to a hidden layer of 256 units.
  - The model will have 3 of them, each one decreasing the number of output units. The last will have 10 output units, because each unit will be associated to a different class, and we have a 10-class classification problem.
- We also defined dropout layers using `nn_dropout()`. These will be used to perform dropout regularization.
- Finally we define the activation layer using `nn_relu()`.
- In `forward()` we define the order in which these layers are called. We call them in blocks like (linear, activation, dropout), except for the last layer that does not use an activation function or dropout.
- Finally, we use `print` to summarize the model, and to make sure we got it all right.

```
print(modelnn())
```

```
An `nn_module` containing 235,146 parameters.

-- Modules ---------------------------------------------------------------------
* linear1: <nn_linear> #200,960 parameters
* linear2: <nn_linear> #32,896 parameters
* linear3: <nn_linear> #1,290 parameters
* drop1: <nn_dropout> #0 parameters
* drop2: <nn_dropout> #0 parameters
* activation: <nn_relu> #0 parameters
```

The parameters for each layer include a bias term, which results in a parameter count of 235,146. For example, the first hidden layer involves $(784 + 1) \times 256 = 200{,}960$ parameters.

Next, we add details to the model to specify the fitting algorithm. We fit the model by minimizing the cross-entropy function given by:

$$-\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im} \log(f_m(x_i)),$$

for $m = 0, 1, \dots, 9$ and where $f_m(X)$ is the *softmax* activation function

$$f_m(X) = P(Y = m|X) = \frac{e^{Z_m}}{\sum_{l=0}^{9} e^{Z_l}}.$$

Notice that in `torch` the cross entropy function is defined in terms of the logits, for numerical stability and memory efficiency reasons. It does not require the target to be one-hot encoded.

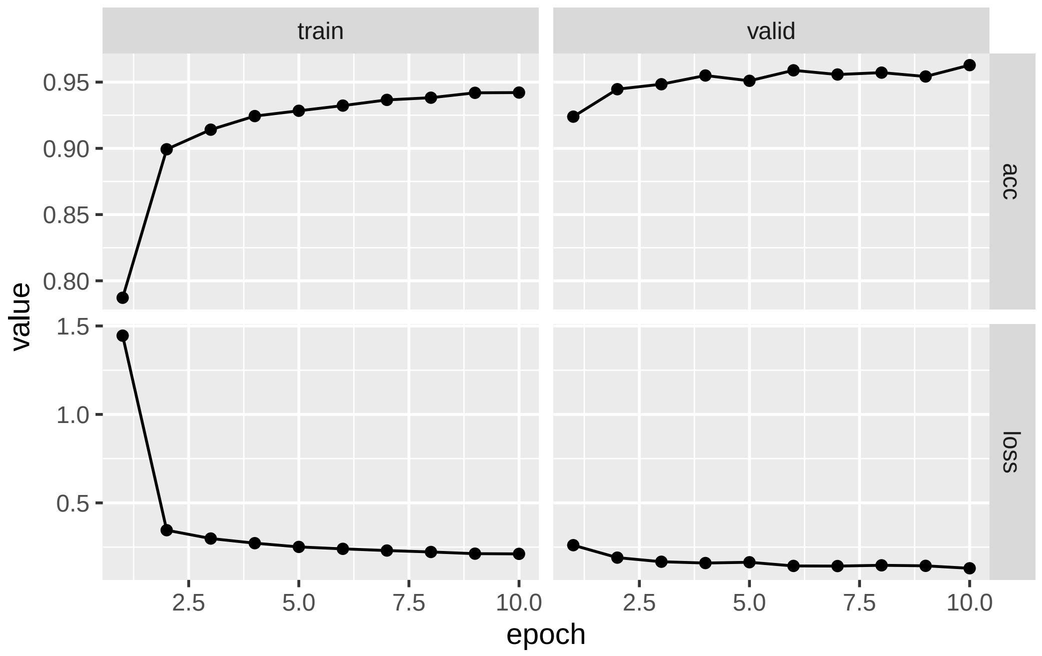
```r
modelnn <- modelnn %>%
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_rmsprop,
    metrics = list(luz_metric_accuracy())
  )
```

Now we are ready to go. The final step is to supply training data, and fit the model.

```r
system.time(
   fitted <- modelnn %>%
      fit(
        data = train_ds,
        epochs = 10, #15,
        valid_data = 0.2,
        dataloader_options = list(batch_size = 256),
        verbose = TRUE
      )
 )
```

```
   user  system elapsed 
 959.05  744.64  367.55 
```

```r
plot(fitted)
```



To obtain the test error, we first write a simple function `accuracy()` that compares predicted and true class labels, and then use it to evaluate our predictions.

```
accuracy <- function(pred, truth) {
   mean(pred == truth) }

# gets the true classes from all observations in test_ds.
truth <- sapply(seq_along(test_ds), function(x) test_ds[x][[2]])

fitted %>%
  predict(test_ds) %>%
  torch_argmax(dim = 2) %>%  # the predicted class is the one with higher 'logit'.
  as_array() %>% # we convert to an R object
  accuracy(truth)
```

```
[1] 0.9647
```

Question 6: What is the test accuracy from this model?

0.9631

Although packages such as `glmnet` can handle multiclass logistic regression, they are quite slow on this large dataset. It is much faster and quite easy to fit such a model using the `luz` software. We just have an input layer and output layer, and omit the hidden layers!

```r
modellr <- nn_module(
  initialize = function() {
    self$linear <- nn_linear(784, 10)
  },
  forward = function(x) {
    self$linear(x)
  }
)
print(modellr())
```

```
An `nn_module` containing 7,850 parameters.

-- Modules ---------------------------------------------------------------------
* linear: <nn_linear> #7,850 parameters
```

We fit the model just as before.

```r
fit_modellr <- modellr %>%
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_rmsprop,
    metrics = list(luz_metric_accuracy())
  ) %>%
  fit(
    data = train_ds,
    epochs = 5,
    valid_data = 0.2,
    dataloader_options = list(batch_size = 128)
  )

fit_modellr %>%
  predict(test_ds) %>%
  torch_argmax(dim = 2) %>%  # the predicted class is the one with higher 'logit'.
  as_array() %>% # we convert to an R object
  accuracy(truth)
```

```
[1] 0.9131
```

```r
# alternatively one can use the `evaluate` function to get the results
# on the test_ds
evaluate(fit_modellr, test_ds)
```

```
A `luz_module_evaluation`
-- Results ---------------------------------------------------------------------
loss: 0.3114
acc: 0.9131
```