

Solution 1:

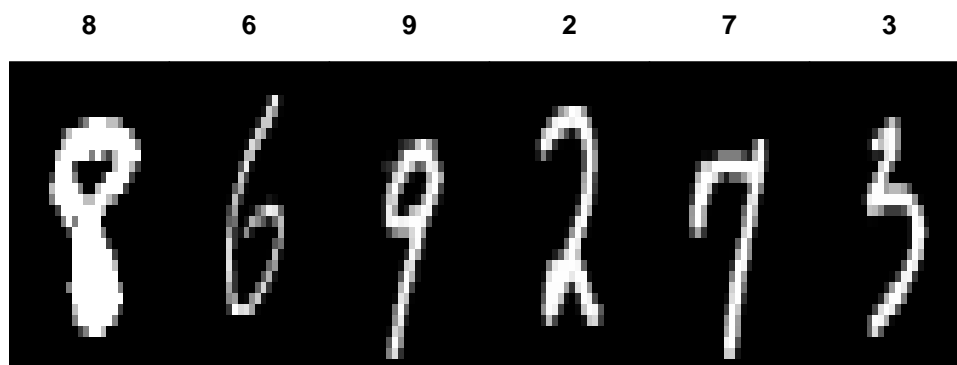
- (a) Read in the MNIST data set

```
library(keras)
mnist <- dataset_mnist()
```

- (b) Visualize the data like

```
library(keras)
mnist <- dataset_mnist()
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y

# visualize the digits
par(mfcol=c(1,6))
par(mar=c(0, 0, 3, 0), xaxs='i', yaxs='i')
for (idx in sample(1:NROW(x_train), 6)) {
  im <- x_train[idx,,]
  im <- t(apply(im, 2, rev))
  image(1:28, 1:28, im, col=gray((0:255)/255),
        xaxt='n', main=paste(y_train[idx]),
        yaxt='n')
}
```



- (c) Convert the features to a (pandas) data frame, by flattening the 28x28 images to a 784-entry-long vector, which represents one row in your data frame. Divide the intensity values of each pixel (each column) by 255 to get a value between 0 and 1.

```
library(tibble)
# reshape
dim(x_train) <- c(nrow(x_train), 784)
dim(x_test) <- c(nrow(x_test), 784)
# rescale
x_train <- x_train / 255
x_test <- x_test / 255
```

```
# convert to data.frame
x_train <- as_tibble(as.data.frame(x_train))
x_test  <- as_tibble(as.data.frame(x_test))
```

(d) Softmax regression

```
library(nnet)
data <- cbind(y = as.factor(y_train), x_train)
# note: takes some time and requires quite some memory
# also you need to set the maximum number of weights to get it running
# we will further restrict the maximum number of iterations
# to avoid overfitting (explanation is given later)
model <- multinom(y ~ -1 + ., data = data, MaxNWts = 7860, maxit = 20)
```

```
## # weights:  7850 (7056 variable)
## initial  value 138155.105580
## iter   10 value 30790.232712
## iter   20 value 23682.853837
## final   value 23682.853837
## stopped after 20 iterations
```

Look at the larger weights:

```
summary(model$wts)

##      Min.   1st Qu.   Median     Mean   3rd Qu.    Max.
## -0.754937 -0.036317  0.000000 -0.003706  0.009061  0.842105

which.max(abs(model$wts))

## [1] 1192

dim(coef(model))

## [1]  9 784
```

There seem to be a few very large coefficients

(e) Use keras:

```
library(dplyr)

##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##   filter, lag
## The following objects are masked from 'package:base':
##
##   intersect, setdiff, setequal, union

library(keras)

# convert outcome using one-hot encoding
y_train_one_hot <- to_categorical(y_train)
```

```

y_test_one_hot <- to_categorical(y_test)

neural_network <- keras_model_sequential()

neural_network %>%
  layer_dense(units = 10, # corresponding to the number of classes
              activation = "softmax",
              input_shape = list(784)) %>%
  compile(
    optimizer = "adam",
    loss      = "categorical_crossentropy",
    metric    = "accuracy"
  )

history_minibatches <- fit(
  object      = neural_network,
  x           = as.matrix(x_train),
  y           = y_train_one_hot,
  batch_size  = 24,
  epochs      = 80,
  validation_split = 0.2,
  callbacks   = list(callback_early_stopping(patience = 10)),
  verbose     = FALSE, # set this to TRUE to get console output
  view_metrics = FALSE # set this to TRUE to get a dynamic graphic output in RStudio
)

```

Look at the network weights

```

library(tensorflow)
tensor_weights <- as.matrix(tf$add(neural_network$weights[[1]],0))
summary(c(tensor_weights))

```

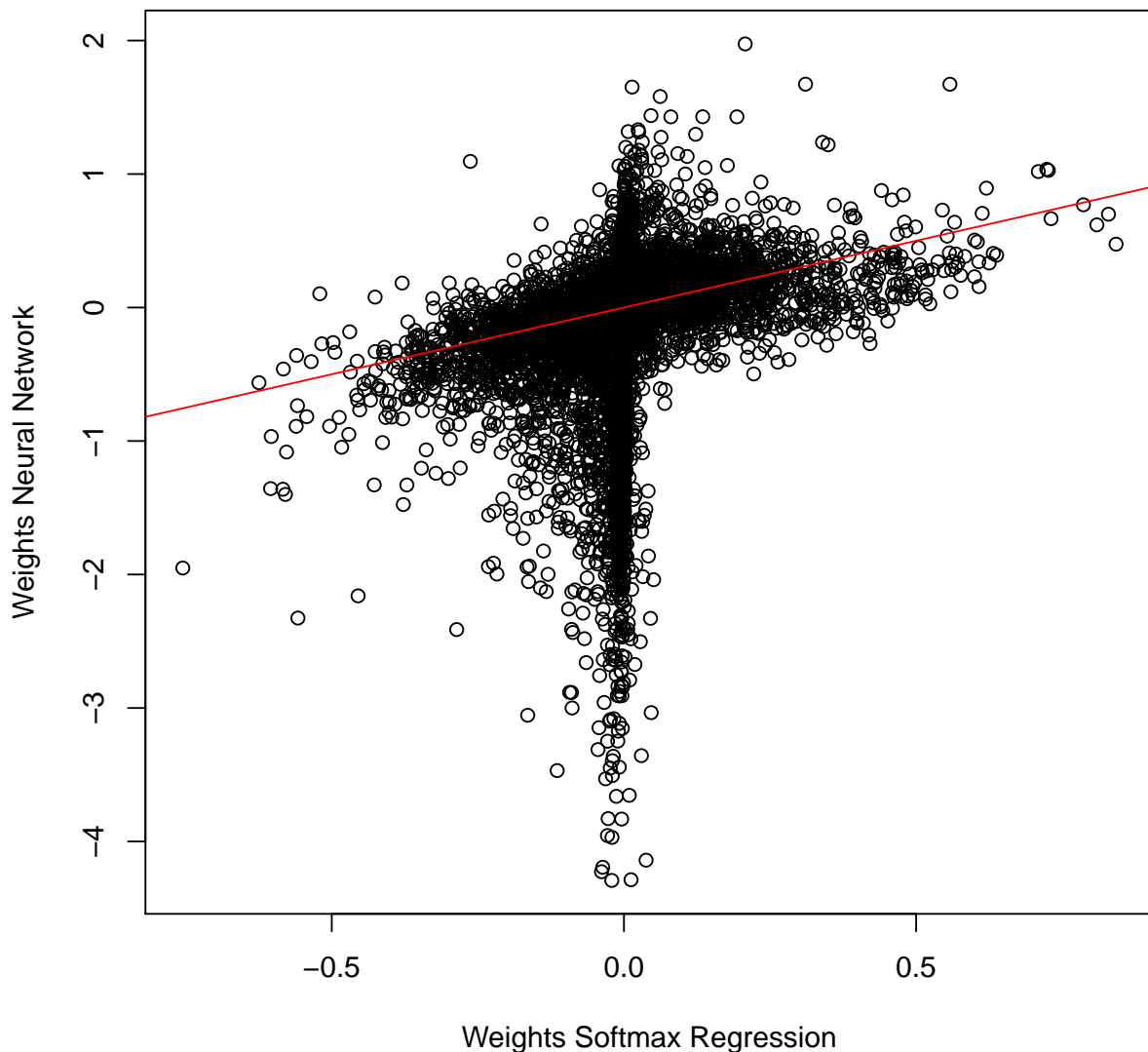
##	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
##	-4.29159	-0.46365	-0.07501	-0.24104	0.09048	1.97477

and compare to the ones from multinomial logistic regression:

```

plot(c(tensor_weights[, -1]) ~ c(t(coef(model))),
     xlab = "Weights Softmax Regression", ylab = "Weights Neural Network")
abline(0,1, col="red")

```



As both models de facto are based on neural networks (here the implementation of the softmax regression is actually done by fitting a neural network with the very same network structure), their similarity depends on how the network is trained. While clearly the implementation calling Python with backend **TensorFlow** (the **keras** fit) is much much faster, the network also converges more quickly due to a small batch size while the multinomial logistic regression calls a network fitting algorithm that uses batch size equal to the number of observations (which is usually a bad idea).

(f) First define the metrics

```
# Classification error (how many of the predictions are wrong)
classifiererror <- function(actual, predicted) {
  return(mean(actual != predicted))
}

# Accuracy (how many of the predictions are correct)
accuracy <- function(actual, predicted) {
  return(1 - classifiererror(actual, predicted))
}
```

```

# As we will usually have probabilistic predictions,
# we need to convert those to classes for the above
# metrics using the class with the max probability
probs_to_class <- function(probvec) {
  which.max(probvec)-1
}

# MC Brier score
mcbrier <- function(actual_one_hot, prob) {
  rowSums((actual_one_hot-prob)^2)
}

# Cross-Entropy loss
crossentropy <- function(actual_one_hot, prob) {
  rowSums( -log(prob) * actual_one_hot )
}

# negative log-likelihood of multinomial distribution
loglikmultinom <- function(actual_one_hot, prob) {
  sapply(1:nrow(actual_one_hot), function(i)
    dmultinom(actual_one_hot[i,],
              size = 1, prob[i,], log = TRUE))
}

```

Now we get the predictions:

```

pred_multinom <- predict(model, x_test, type = "probs")
pred_nn <- predict(neural_network, as.matrix(x_test))
str(pred_multinom, 1)

## num [1:10000, 1:10] 0.001922 0.019599 0.000202 0.98602 0.00456 ...
## - attr(*, "dimnames")=List of 2

str(pred_nn, 1)

## num [1:10000, 1:10] 1.03e-08 1.60e-05 3.93e-07 1.00 4.58e-04 ...

```

Let's first look at the confusion matrix (in this case for the multinomial regression):

```

table(y_test, apply(pred_multinom,1,probs_to_class))

##
## y_test      0      1      2      3      4      5      6      7      8      9
##      0  904      0      0      3      1     55     12      2      3      0
##      1      0 1108      2     10      1      1      5      2      6      0
##      2      8     11   874     34      9      6     20     15     47      8
##      3      6      0     11   926      3     15      5      9     26      9
##      4      0      5      2      4   901      2     13      4      3     48
##      5     11      1      3     54     13   703     23     10     62     12
##      6      9      3      4      1      8     16    913      2      2      0
##      7      4     14      9     11      8      1      1    938      3     39
##      8     10     14      5     48     12     22     16     12    813     22
##      9      8      5      1     19     41      6      1     19      7    902

```

Now the metrics. Classification error:

```
cbind(multinom = classiferror(y_test, apply(pred_multinom,1,probs_to_class)),
      neural = classiferror(y_test, apply(pred_nn,1,probs_to_class))
)

##      multinom neural
## [1,]    0.1018 0.0743
```

Accuracy:

```
# Classification error
cbind(multinom = accuracy(y_test, apply(pred_multinom,1,probs_to_class)),
      neural = accuracy(y_test, apply(pred_nn,1,probs_to_class))
)

##      multinom neural
## [1,]    0.8982 0.9257
```

MC Brier score (note that we look at the mean, because the definition of the loss is on an observation basis):

```
cbind(multinom = mean(mcbrier(y_test_one_hot, pred_multinom)),
      neural = mean(mcbrier(y_test_one_hot, pred_nn))
)

##      multinom      neural
## [1,] 0.1658749 0.1139387
```

Cross-entropy (mean):

```
cbind(multinom = mean(crossentropy(y_test_one_hot, pred_multinom)),
      neural = mean(crossentropy(y_test_one_hot, pred_nn))
)

##      multinom      neural
## [1,] 0.383929 0.2752445
```

Mean negative log-likelihood of multinomial distribution:

```
cbind(multinom = mean(loglikmultinom(y_test_one_hot, pred_multinom)),
      neural = mean(loglikmultinom(y_test_one_hot, pred_nn))
)

##      multinom      neural
## [1,] -0.383929 -0.2752445
```