Machine Learning 2ST129 26605 HT2023 Assignment 3

Anonymous Student

November 26, 2023

Contents

1 Task 1 **3**
1.1 1.1 . . . 3
1.2 1.2 . . . 4
1.3 1.3 . . . 5
1.4 1.4 . . . 6
1.5 1.5 . . . 8
1.6 1.6 . . . 9
1.7 1.7 . . . 11
1.8 1.8 . . . 12
1.9 Task 2 . . . 13
1.10 2.1 . . . 13
1.11 2.2 . . . 13
1.12 2.3 . . . 13
1.13 2.4 . . . 14
1.14 2.5 . . . 14

General Information

- Time used for reading 3 hours :
- Time used for basic assignment 15 :
- Time used for extra assignment 3 hours
- Good with lab: Good introduction to Keras. Also fun to somewhat freely try to experiment with the model instead of only having to implement a given model.
- Things to improve in the lab: Maybe more about if there is some type of general strategy, or at least rough guidelines, on how to tune a model to get better accuracy. For example, when to implement regularizers on the layers or how to tune the learning rate. Now I just tested it randomly without any specific strategy until i was satisfied with the results.

```
#Libraries
 library(tidyverse)
 library(xtable)
 library(keras)
 library(tensorflow)
```

1 Task 1

1.1 1.1

First we instantiate and visualize the data.

```
 mnist <- dataset_mnist()
x_train <- mnist$train$x/255
x_test<- mnist$test$x/255

#' function to iterate through the different idx and plot the images
plot_images <- function(image_array, y, idx, pixel_dim = 28, ncol = 3, pred=FALSE) {
  par(mfrow =c(4,2))

  for (i in idx) {
    im <- image_array[i,,]
    im <- t(apply(im, 2, rev))
    if (isFALSE(pred)){
      main <- paste(y[i])
    } else{
main <- paste("Actual: ", paste0(y[i],","), "Predicted:", pred[i])
    }
  image(1:pixel_dim, 1:pixel_dim, im, col = gray((0:255)/255),
            xlab = "", ylab = "", xaxt = 'n', yaxt = 'n',
        main = main)
  }

  par(mfrow = c(1, 1))
}
```

```
plot_images(image_array = x_train, y = mnist$train$y, idx = c(1:8))
```

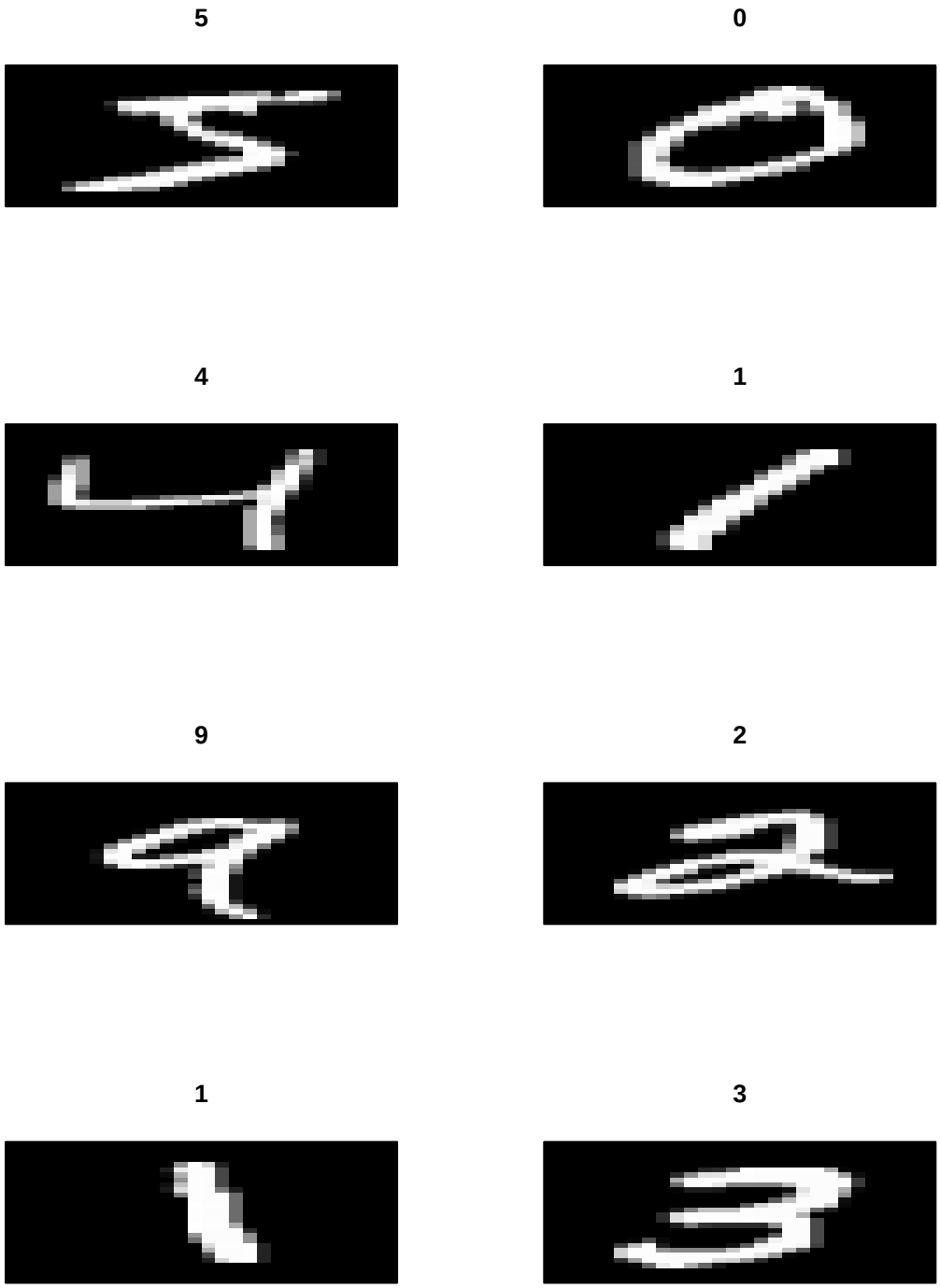

Figure 1: Visualization of digits

1.2 1.2

The data set contains of 60k 28x28 grayscale images of the 10 digits, as well as a test set with 10k images.

```
str(mnist$train)
```

```
List of 2
 $ x: int [1:60000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
 $ y: int [1:60000(1d)] 5 0 4 1 9 2 1 3 1 4 ...
```

```
str(mnist$test)
```

```
List of 2
 $ x: int [1:10000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
 $ y: int [1:10000(1d)] 7 2 1 0 4 1 4 9 5 9 ...
```

1.3 1.3

Now we want to implement a simple-feed-forward neural network with one hidden layer with 16 units and by using the sigmoid activation function. The steps done here to modify the data or model is as proposed by the guide Getting Started with Keras.

```
#Since the reponse variablbe y is an vector with integer values with 10 classes,
# we need to one-hot encode them into binary class matrices
#
y_train <- to_categorical(mnist$train$y, num_classes = 10)
y_test <- to_categorical(mnist$test$y, num_classes = 10)
```

```
# Model architecture
model <- keras_model_sequential(input_shape = c(28,28)) %>%
  layer_flatten() %>%
  layer_dense(units = 16, activation = 'sigmoid') %>%
  # Hidden layer with 16 units and sigmoid activation
  layer_dense(units = 10, activation = 'softmax')
# Output layer with softmax activation
```

```
print(model)
```

```
Model: "sequential"
_________________________________________________________________________________
 Layer (type)                        Output Shape                      Param #
=================================================================================
 flatten (Flatten)                   (None, 784)                       0
 dense_1 (Dense)                     (None, 16)                        12560
 dense (Dense)                       (None, 10)                        170
=================================================================================
Total params: 12730 (49.73 KB)
Trainable params: 12730 (49.73 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________________________
```

The model has 12730 parameters in total. The input layer has 12560 and the output layer has 170. Next we compile the model and fit it to compute the validation accuracy. For computational reasons we can set the `batch_size` to 128 to make it run faster, while also setting `validation_split()` to 0.2 so that it for every epoch uses 20 percent of the data as validation.

```
# Compile the model
#here we choose the "adam optimizer as well as categorical_crossentropy
model %>% compile(
  optimizer = 'adam',
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)

# Train the model
```

```
#here we choose batch_size = 128 and validaiton split = 0.2
#the number of samples used in each iteration of gradient descent
# which is more computationally efficient
# Validation is the fraction of the training data that is reserved for validation.

history <- model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
  #validation_data = list(x_test, y_test)
)
```

For the last epoch, we have the following results:

```
epoch 30/30
375/375 [==============================] - 1s 2ms/step
- loss: 0.1536 - accuracy: 0.9559 - val_loss: 0.1906 - val_accuracy: 0.9438
```

Hence the accuracy is 0.956 and the validation accuracy is 0.944. The results is also shown in figure 2below

```
knitr::include_graphics("model1.png")
```

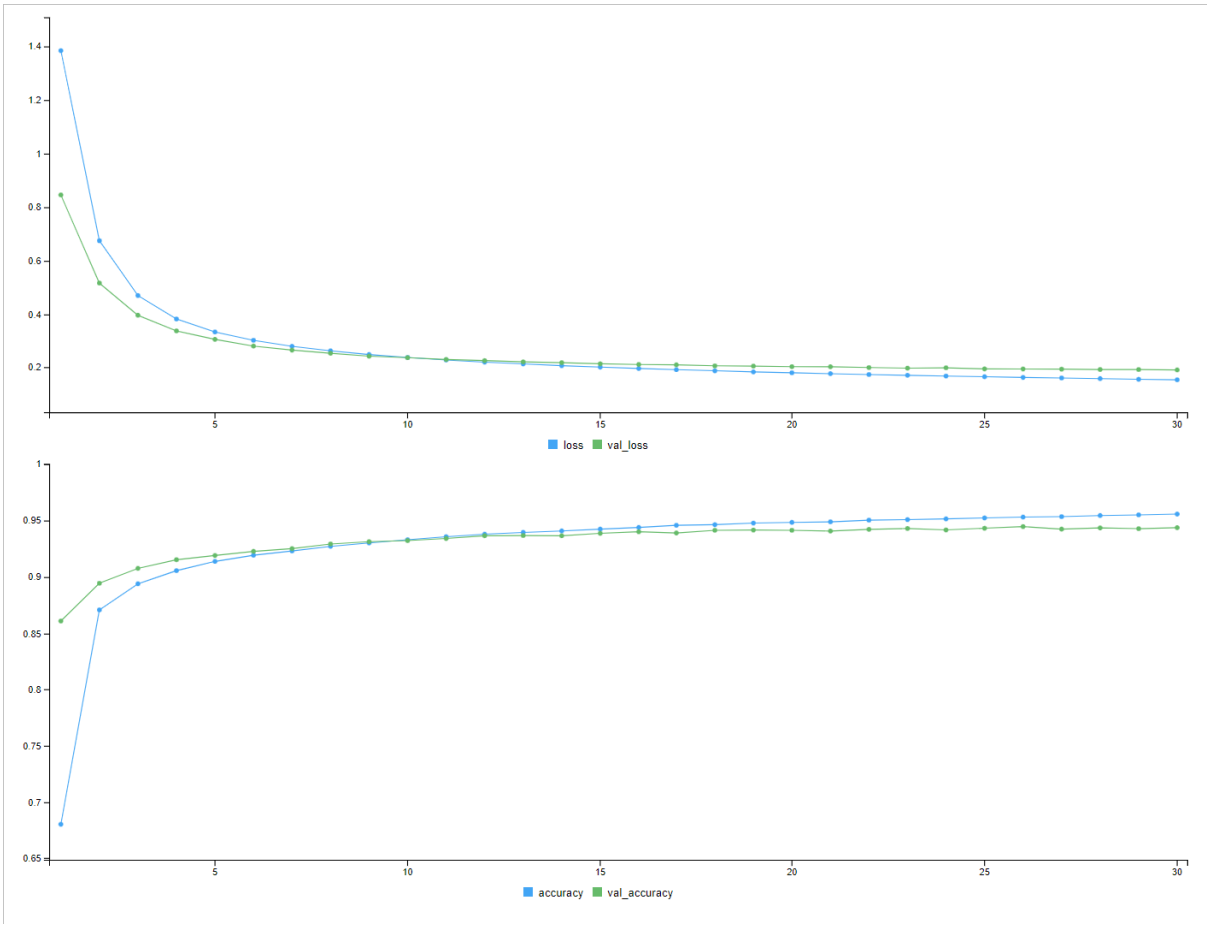

Figure 2: Results for model 1

1.4 1.4

Now we make some adjustments to the model by implementing the following:

- Increase the number of hidden units to 128.
- Change the activation function to reLU.
- Change the optimizer to RMSprop.
- Add a second layer with 128 hidden units.
- Add dropout with 0.2 dropout probability for both of hidden layers.
- Add batch normalization for both of hidden layers.

```
# Model architecture
model_new <- keras_model_sequential(input_shape = c(28,28)) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = 'relu') %>%  # Hidden layer with 128
  #units and relu activation. More learning capacity but also more overfitting
  layer_dropout(0.2) %>%  #dropping out random neurons with p=0.2
  layer_batch_normalization() %>%
  layer_dense(units=128, activation="relu") %>% #second layer
  layer_dropout(0.2) %>%
  layer_batch_normalization() %>%
  layer_dense(units = 10, activation = 'softmax')
# Output layer with softmax activation

  #batch normalization: reparametrizes the model in a way that introduces both
  #additive and multiplicative noise on the hidden units at training time

summary(model_new)
```

```
Model: "sequential_1"
_________________________________________________________________________________
 Layer (type)                    Output Shape                Param #     Trainable
=================================================================================
 flatten_1 (Flatten)             (None, 784)                 0           Y
 dense_4 (Dense)                 (None, 128)                 100480      Y
 dropout_1 (Dropout)             (None, 128)                 0           Y
 batch_normalization_1 (Batch    (None, 128)                 512         Y
 Normalization)
 dense_3 (Dense)                 (None, 128)                 16512       Y
 dropout (Dropout)               (None, 128)                 0           Y
 batch_normalization (BatchNo    (None, 128)                 512         Y
 rmalization)
 dense_2 (Dense)                 (None, 10)                  1290        Y
=================================================================================
Total params: 119306 (466.04 KB)
Trainable params: 118794 (464.04 KB)
Non-trainable params: 512 (2.00 KB)
_________________________________________________________________________________
```

```
model_new %>% compile(
  optimizer = 'RMSprop',# RMSPROP instead of adam
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)

history <- model_new %>% fit(
  x_train, y_train,
  epochs = 30,batch_size = 128,
  validation_split = 0.2
)
```

```
375/375 [==============================] - 2s 4ms/step
- loss: 0.0330 - accuracy: 0.9893 - val_loss: 0.0848 - val_accuracy: 0.9815
```

```
knitr::include_graphics("Model2.png")
```

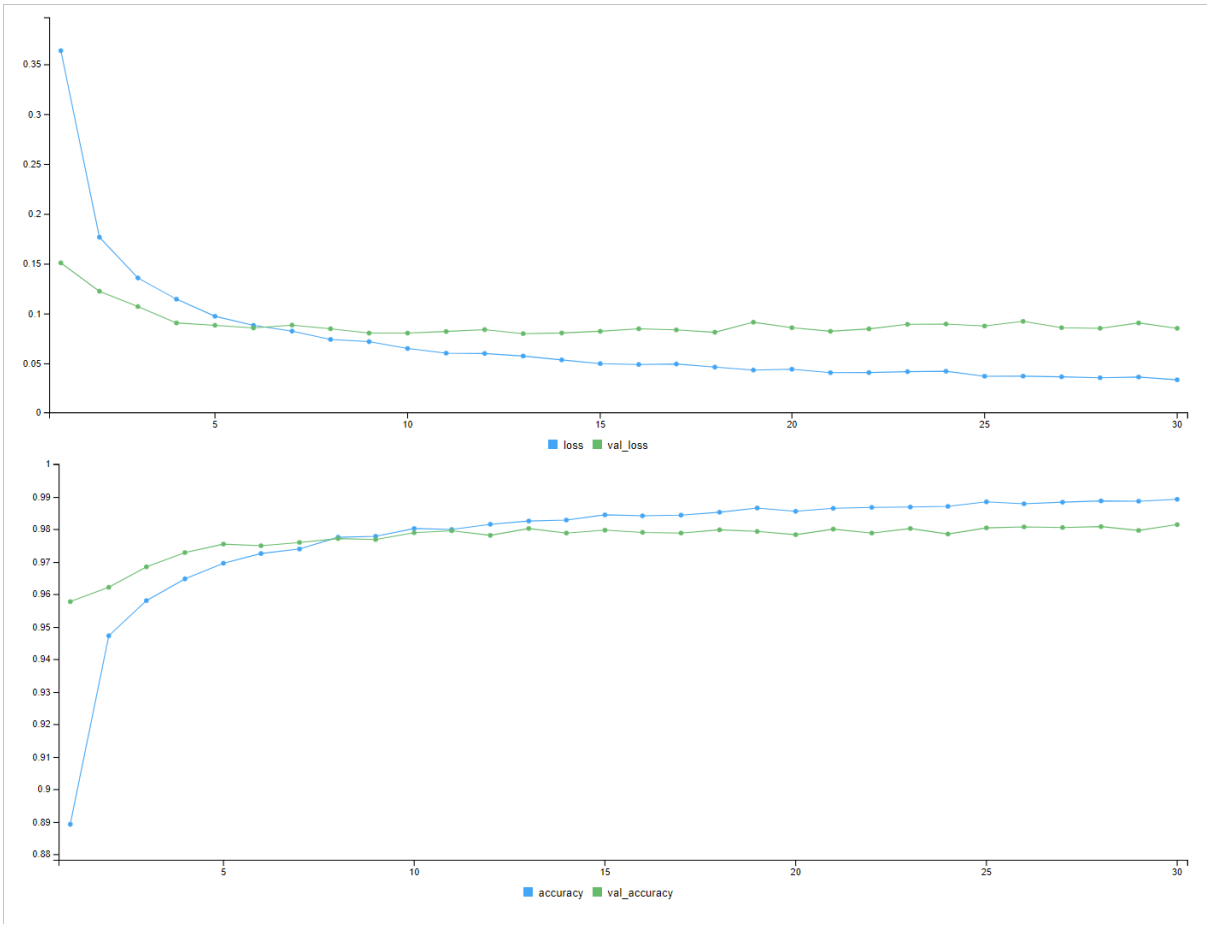

Figure 3: Results for second model

Looking at the results from the output as well as figure 3, then the accuracy is 98.9 and the validation accuracy is 0.982.

1.5 1.5

If i were to use early stopping, then for each epoch we keep track of the parameter values as well as the accuracy. Then in order to obtain a model with better validation error, I would return the parameter setting for the point in which the validation set accuracy was the highest. If had do to implement it from scratch, then to the best of my knowledge I would have to write my own callback. But otherwise I probably can just use the defined method `Earlystopping` and something along the lines of:

```
callback <- list(callback_early_stopping(patience = 5))
history <- model %>% fit(
  #rest of arguments here
  #....
  callbacks = callback
)
```

1.6 1.6

For this task we want to find the best possible model. I tried to vary different arguments and parameters. In the end I just decided to stick with the following changes:

- Added 3 new hidden layers, each with relu activation and changed the number of units for all layers
- Added L_2 regularizer for all hidden layers in order to apply penalties on layer parameters or layer activity during optimization.
- Specified that the images are in greyscale by adding 1 in input shape. But i do not know if it actually will affect performance.
- Created a function `Scheduler()` to decrease the learning rate exponentially after 10 epochs
- Implemented Early stopping.
- increased batch size to 256

```
model_best <- keras_model_sequential() %>%
  layer_flatten(input_shape = c(28, 28, 1)) %>%
  layer_dense(units = 1024, activation = 'relu',
              kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_dropout(0.3) %>%
  layer_batch_normalization() %>%
  layer_dense(units = 512, activation = 'relu',
              kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_dropout(0.3) %>%
  layer_batch_normalization() %>%
  layer_dense(units = 256, activation = 'relu',
              kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_dropout(0.3) %>%
  layer_batch_normalization() %>%
  layer_dropout(0.3) %>%
   layer_dense(units=128, activation="relu",
               kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_batch_normalization() %>%
  layer_dropout(0.3) %>%
   layer_dense(units=64, activation="relu",
               kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_batch_normalization() %>%
  layer_dropout(0.3) %>%
  layer_dense(units = 10, activation = 'softmax')

summary(model_best)
```

```
Model: "sequential_2"
_________________________________________________________________________________
 Layer (type)                    Output Shape                Param #     Trainable
=================================================================================
 flatten_2 (Flatten)             (None, 784)                 0           Y
 dense_10 (Dense)                (None, 1024)                803840      Y
 dropout_7 (Dropout)             (None, 1024)                0           Y
 batch_normalization_6 (Batch    (None, 1024)                4096        Y
 Normalization)
 dense_9 (Dense)                 (None, 512)                 524800      Y
 dropout_6 (Dropout)             (None, 512)                 0           Y
 batch_normalization_5 (Batch    (None, 512)                 2048        Y
 Normalization)
 dense_8 (Dense)                 (None, 256)                 131328      Y
 dropout_5 (Dropout)             (None, 256)                 0           Y
 batch_normalization_4 (Batch    (None, 256)                 1024        Y
```

```
 Normalization)
 dropout_4 (Dropout)           (None, 256)               0         Y
 dense_7 (Dense)               (None, 128)               32896     Y
 batch_normalization_3 (Batch  (None, 128)               512       Y
 Normalization)
 dropout_3 (Dropout)           (None, 128)               0         Y
 dense_6 (Dense)               (None, 64)                8256      Y
 batch_normalization_2 (Batch  (None, 64)                256       Y
 Normalization)
 dropout_2 (Dropout)           (None, 64)                0         Y
 dense_5 (Dense)               (None, 10)                650       Y
================================================================================
Total params: 1509706 (5.76 MB)
Trainable params: 1505738 (5.74 MB)
Non-trainable params: 3968 (15.50 KB)
--------------------------------------------------------------------------------
```

```
model_best %>% compile(
  optimizer = 'RMSprop',
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)


Scheduler <- function(epoch, lr) {
  if (epoch < 15) {
    return(lr)
  } else {
    return(lr * exp(-0.1))
  }
}

callback_list = list(callback_early_stopping(patience = 10),
                     callback_learning_rate_scheduler(Scheduler))

history <- model_best %>% fit(
  x_train, y_train,
  epochs = 75,batch_size = 256,
  validation_split = 0.2,
  callbacks =callback_list,
  verbose = 1)
```

This terminated early at epoch 59 with the following result:

```
Epoch 59/75
188/188 [==============================] - 7s 40ms/step
- loss: 0.0368 - accuracy: 0.9977 - val_loss: 0.1097 - val_accuracy: 0.9854 - lr: 1.2277e-05
```

However, since it used early stopping, I am not sure about which epoch exactly it decided on in the end. Either wait, the accuracy is around 0.997 and the validation accuracy is around 0.985. The results can also be seen in figure 4 below:

```
knitr::include_graphics("model3.png")
```

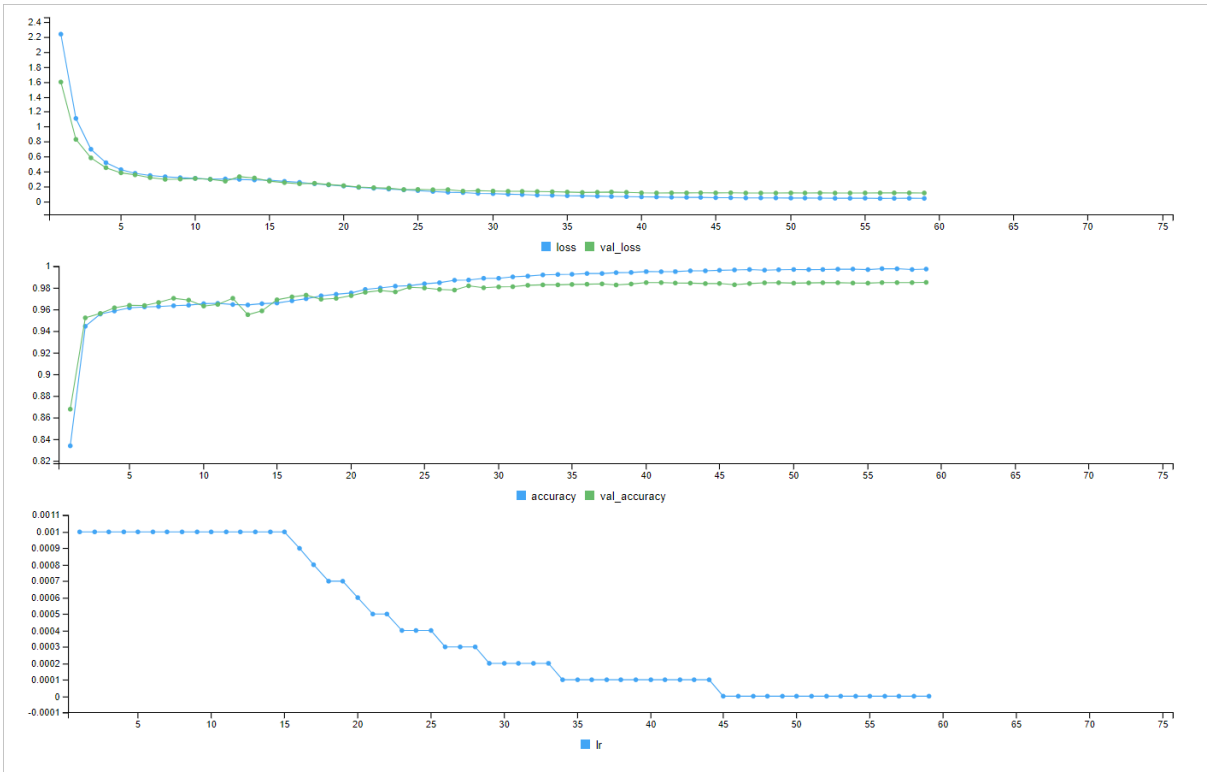

Figure 4: Results for the third model

1.7 1.7

Now we identify two digits that the network has classified incorrectly.

```
predictions <- model_best %>% predict(x_test)
predicted_labels <- max.col(predictions) - 1
true_labels <- max.col(y_test) - 1
# Gives the column for which the number is 1, which corresponds to the digit.
#  subtract 1 since numbers   starts at 0


incorrect_indices <- which(predicted_labels != true_labels)

incorrect_images <- x_test[incorrect_indices, ,]

plot_images(image_array = x_test, , y=mnist$test$y,pred = predicted_labels,
            idx = incorrect_indices[1:8])
#they look really scuffed
```

```
knitr::include_graphics("wrong_preds.png")
```

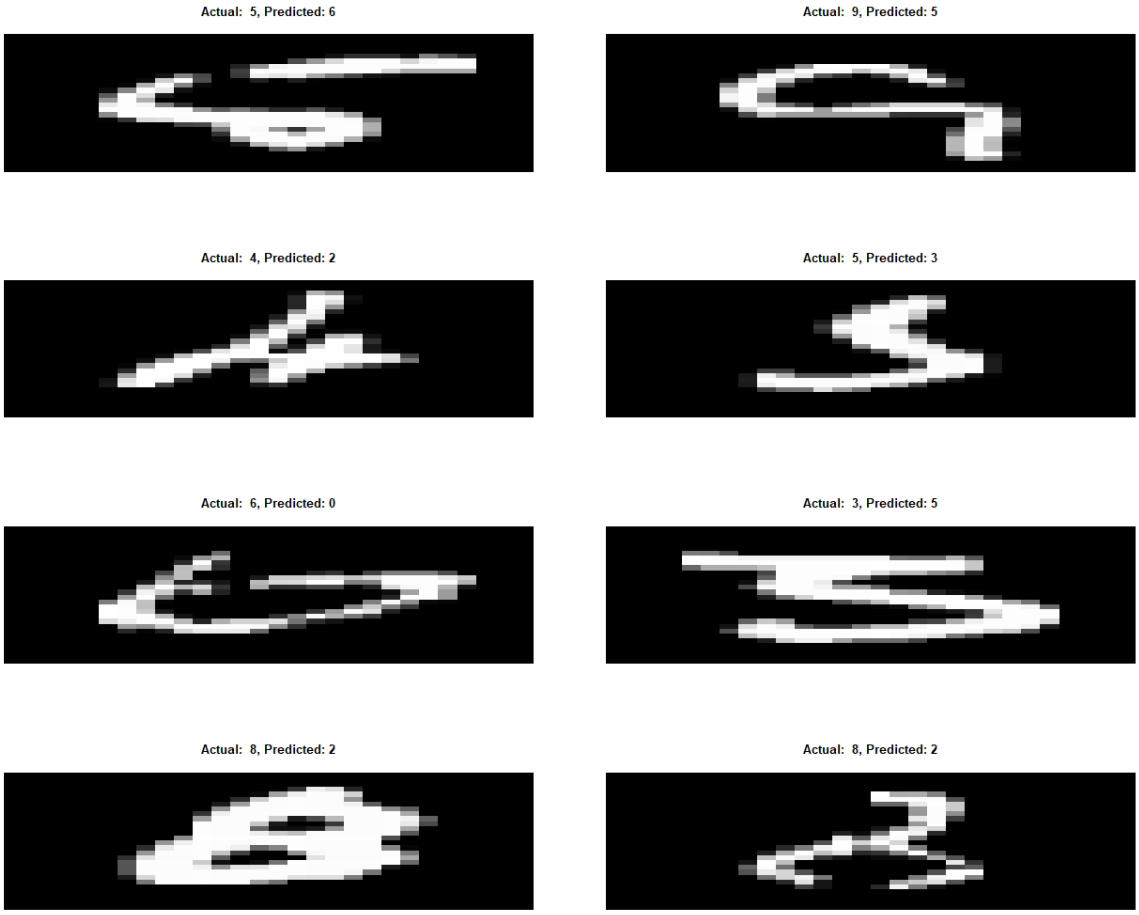

Figure 5: Actual and predicted values for wrongly classified images

Figure 5 shows the some of the wrongly classified images with the predicted values above each image. Looking at them, Then based on the images i would say its not really that surprising seeing as some of them look really scuffed and even I cannot really tell what some of them are supposed to be just by looking at them.

1.8 1.8

Now we compute the accuracy on the hold-out test set.

```
model_best %>% evaluate(x_test, y_test)

13/313 [==============================] - 1s 4ms/step - loss: 0.1017 - accuracy: 0.9866
     loss  accuracy
0.1017455 0.9866000
```

It seems that it is 0.987, which is around the as the validation accuracy from the model when training it. I would say it makes sense in overall since the validation accuracy is supposed to be the sample that is not used in the training for each epoch. However, it is still lower than the normal accuracy at 99.7 percent, which could indicate some overfitting with respect to the whole training model. However, in my opinion, the test accuracy is still rather close so it should not be of any major concern. Perhaps the accuracy could also be explained partly due to the images being very similar to each other, and thus for the most part the model could easily be applied on the test data as well.

1.9 Task 2

1.10 2.1

```
W <- matrix(1, nrow = 2, ncol = 2)
c <- matrix(c(0, -1) , ncol = 2, nrow = 4, byrow = TRUE)
X <- matrix(c(0, 0, 1, 1, 0, 1, 0, 1), ncol = 2)
w <- matrix(c(1, -2) , ncol = 1)
b <- 0

RLF <- function(i) return(max(0,i))

mini_net <- function(X, W, c, w, b){

  RLF_input <-  X%*%W + c
 transformed_res  <- apply(RLF_input, MARGIN = c(1,2), FUN= RLF)

  return(transformed_res %*% w + b)
}
```

```
mini_net(X, W, c, w, b)
```

```
     [,1]
[1,]    0
[2,]    1
[3,]    1
[4,]    0
```

```
mini_net(X, W*0.9, c, w, b)
```

```
     [,1]
[1,]  0.0
[2,]  0.9
[3,]  0.9
[4,]  0.2
```

1.11 2.2

Now we change $W_{1,1}$ to 0.

```
W[1,1]<- 0

mini_net(X, W, c, w, b)
```

```
     [,1]
[1,]    0
[2,]    1
[3,]    0
[4,]   -1
```

```
mini_net(X, W*0.9, c, w, b)
```

```
     [,1]
[1,]  0.0
[2,]  0.9
[3,]  0.0
[4,] -0.7
```

1.12 2.3

The current output functions is a linear function, but the first part captures the nonlinear information so I think its still reasonable. However, perhaps one could consider something else such as the sigmoid function since it is suitable for 0-1 binary values.

1.13 2.4

Now we implement a squared loss function.

```
XOR <- function(x1, x2){
  if(x1 ==1 & x2!=1 | x1!= 1 & x2 ==1){
  return(1)} else{
    return(0)
  }
}

#changing W back to original values
W <- matrix(1, nrow = 2, ncol = 2)
mini_net_loss <- function(X, W, c, w, b){
  mini_res <- mini_net(X, W, c, w, b)
xor_res <- apply(X, MARGIN=1, FUN=function(row){ XOR(row[1], row[2])})
summand <- sum((xor_res -mini_res)^2)
average <- summand / nrow(X)
return(average)
}

mini_net_loss(X, W, c, w, b)
```

```
[1] 0
```

```
mini_net_loss(X, 0.9*W, c, w, b)
```

```
[1] 0.015
```

```
## Originally I did not see it was supposed to be with a given Y argument,
#but since i already implemented the abvove one, we might as well just compare
#the results.
mini_net_loss_2<- function(y, X, W, c, w, b){
  mini_res <- mini_net(X=X, W=W, c=c, w=w, b=b)
  summand <- sum((y-mini_res)^2)
  average <- summand/nrow(X)
  return(average)
}

y <- c(0, 1, 1, 0)
mini_net_loss_2(y, X, W, c, w, b)
```

```
[1] 0
```

```
mini_net_loss_2(y, X, 0.9*W, c, w, b)
```

```
[1] 0.015
```

1.14 2.5

```
#change W[1,1] again
W[1,1] <- 0

mini_net_loss(X, W, c, w, b)
```

```
[1] 0.5
```

```
mini_net_loss_2(y, X, W, c, w, b)
```

```
[1] 0.5
```

```
mini_net_loss(X, 0.9*W, c, w, b)
```

```
[1] 0.375
```

```
mini_net_loss_2(y, X, 0.9*W, c, w, b)
```

```
[1] 0.375
```