BSDA: Assignment 3

November 26, 2023

Time used for reading: 8
Time used for the basic assignment: 9
Time used for extra assignment (VG): 4
Good with lab:
One really good thing with the lab was that I got to understand the concepts behind the feed-forward neural network with everything from activation function, number of hidden layers and so on.

Things to improve in the lab:
Overall really good lab, however one area of improvement would be to extend question 2 with one or two further subtasks where we implement functions with other, maybe more advanced, activation functions.

Anything that was difficult with the material?
When running summary (*model*) then we are able to see how many parameters there is in a layer in the network. Pretty difficult to fully understand where the parameters are coming from in the first hidden layer. I can understand the concept but I would not be able to explain it if I were to be asked.

Worth mentioning, very difficult part of the assignment was to be able to report the output since the output was based on some randomness. Thus I often refer to code and chunks instead of stating exact values.

Task 1: Feed-Forward Neural Network using Keras and TensorFlow

```
setwd("~/Desktop/Machine Learning")
library(knitr)
library(tensorflow)
library(keras)
```

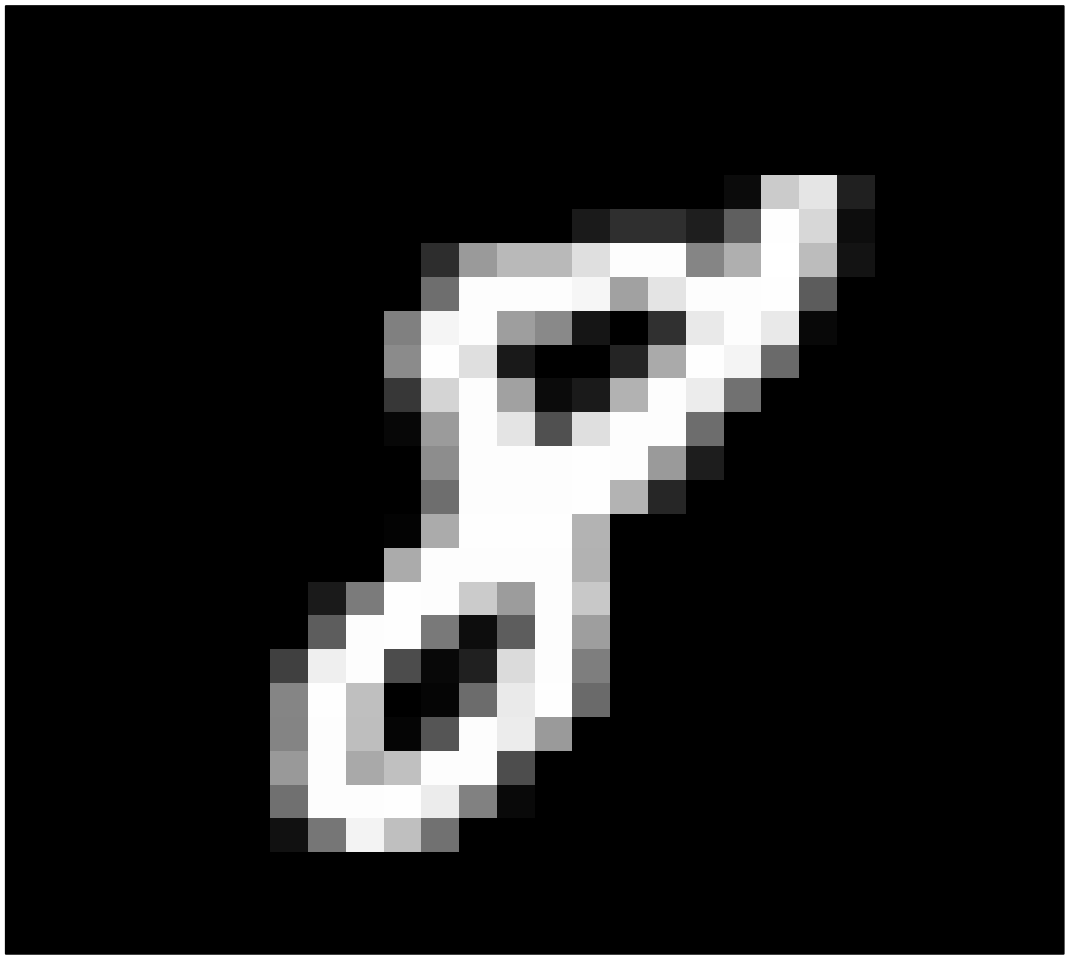
Below we load the MNIST dataset which is a dataset that consists of data on handwritten digits that we want to classify and we run the following code to access the data:

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

1.1

The image below visualize the first eight we could find in the training dataset. The 18th observation in the training data set contained of the first eight.

```r
idx <- min(which(mnist$train$y == 8)) # 18
im <- mnist$train$x[idx,,] # here we index the covariate training dataset for the first eight
# Transpose the image
im <- t(apply(im, 2, rev))
image(1:28, 1:28, im, col=gray((0:255)/255), xlab = "", ylab = "",
      xaxt='n', yaxt='n', main= paste(mnist$train$y[idx]))
```



1.2

When running the following code

```
nrow(mnist$train$x)
```

```
## [1] 60000
```

```
nrow(mnist$test$x)
```

```
## [1] 10000
```

we can see that the training dataset consist of 60 000 observations while the test dataset consists of 10 000 observations.

1.3

We will now implement a simple feed-forward neural network in R using TensorFlow and Keras. The implemented neural network will consist of one hidden layer with 16 units. We use sigmoid $\sigma(x) = \frac{e^x}{1+e^x}$ as the activation function for the hidden function. Further, the output layer will be of a softmax function defined by formula $\text{softmax}(\mathbf{z}) = \frac{\exp(z_j)}{\sum_j \exp(z_j)}$ for the jth group / outcome.

```
model <- keras_model_sequential(input_shape = c(28, 28)) %>%
  # the layer_flattern function which flattern the given input
  layer_flatten() %>%
  # here we have the hidden layer which is being assigned sigmoid as
  # the activation function. The hidden layer consist of 16 units
  layer_dense(16, activation = "sigmoid") %>%
  # this is the output layer which consist of 10 units from the range of
  # 0 to 9 and the activation function for the output layer is the softmax
  # function.
  layer_dense(units = nlevels( as.factor(y_train)), activation = "softmax")

summary(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)
## ________________________________________________________________________________
```

In total the Keras model consists of 12730 parameters where the input layer does not consist of any parameters, i.e 0 st, and the output layer consists of 170 parameters.

Now we will look at the classification accuracy of the network using the compile and fit function in R from the keras package. Also, the classification accuracy will be based on training dataset and we will set epochs to 10 when fitting the model.

```
loss_fn <- loss_sparse_categorical_crossentropy(from_logits = TRUE)

model %>% compile(
  optimizer = "adam",
  loss = loss_fn,
  metrics = "accuracy"
)

object<- model %>% fit(
  x_train, y_train,
  epochs = 10, validation_split = 0.2)
```

Below we can see the validation accuracy for the simple feed forward neural network:

```
object$metrics$val_accuracy[length(object$metrics$val_accuracy)]
```

```
## [1] 0.9444166
```

Further, figure 1 below presents the validation accuracy and validation error per epoch which automatically is being generated when fitting the neural network:

```
knitr::include_graphics("FigureMod1.png")
```

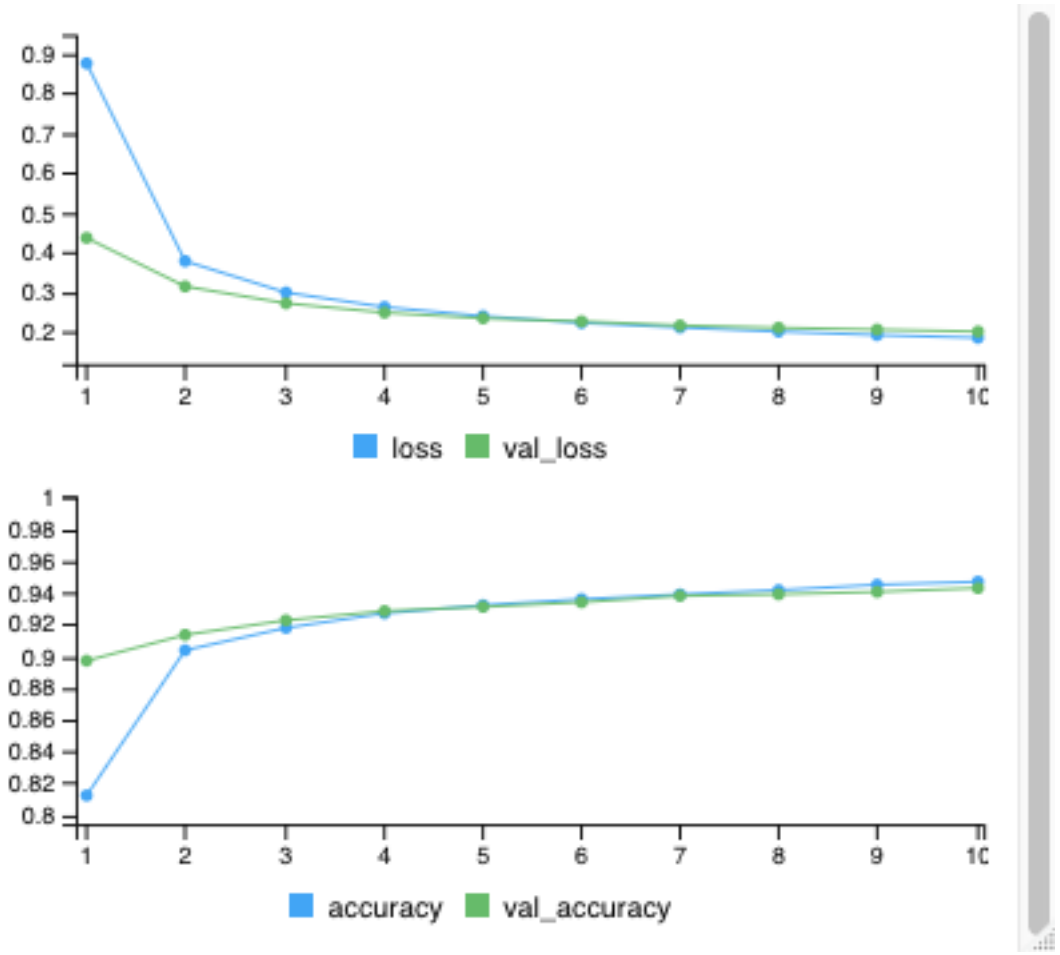


Figure 1: Result for the first model

1.4

For subtask 1.4 we will play around with the some essential concepts in feed-forward networks starting with the simple neural network created above. We will play around with concepts such as the number of hidden units, activation functions, optimizer among others. All the concepts can be found in the instructions for Assignment 3 in Machine Learning.

This will be done in 6 different steps starting with the simple feed forward neural network created above in section 1.3. For each of the 6 different steps we will report the validation accuracy / validation loss as well as a plot for validation accuracy / validation loss per epoch that is being automatically generated when fitting the model, i.e we will report as similar to before but for each step now.

step 1: Increase the number of hidden units to 128

```
model.step1 <- keras_model_sequential(input_shape = c(28, 28)) %>%
  # the layer_flattern function which flattern the given input
  layer_flatten() %>%
  # here we have the hidden layer which is being assigned sigmoid as
```

```
  # the activation function. The hidden layer consist of 16 units
  layer_dense(128, activation = "sigmoid") %>%
  # this is the output layer which consist of 10 units from the range of
  # 0 to 9 and the activation function for the output layer is the softmax
  # function.
  layer_dense(units = nlevels(as.factor(y_train)), activation = "softmax")

model.step1 %>% compile(
  optimizer = "adam",
  loss = loss_fn,
  metrics = "accuracy"
)

object.step1 <- model.step1 %>% fit(
  x_train, y_train,
  epochs = 10, validation_split = 0.2)
```

The validation accuracy and the validation loss, in that order, for epochs = 10 is being presented below and corresponds to:

```
object.step1$metrics$val_accuracy[length(object.step1$metrics$val_accuracy)]
```

```
## [1] 0.9723333
```

```
object.step1$metrics$val_loss[length(object.step1$metrics$val_loss)]
```

```
## [1] 0.09120017
```

and corresponding figure for the validation accuracy / error per epoch can be found below

```
knitr::include_graphics("Figurestep1.png")
```

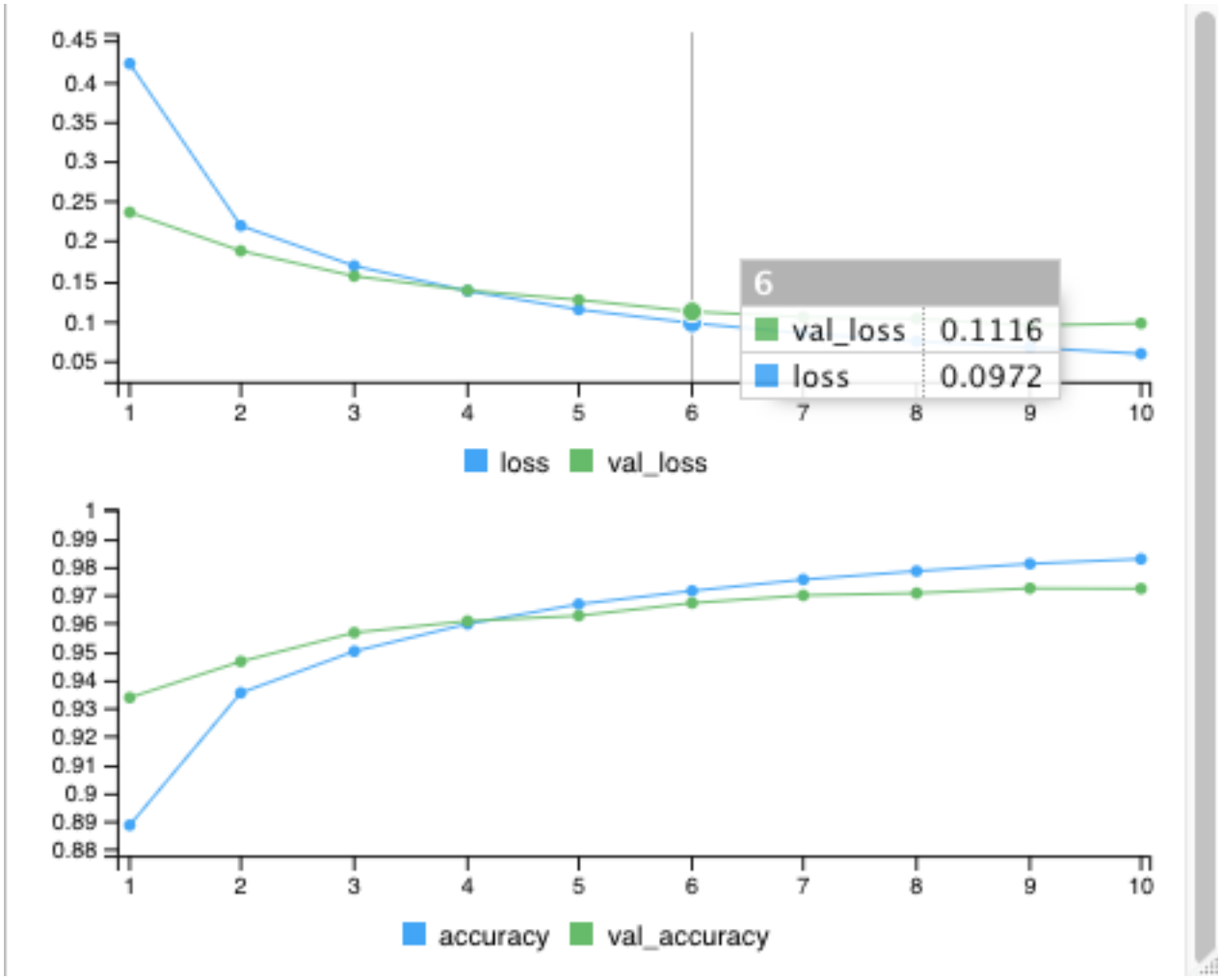


Figure 2: Result when increasing the nr of hidden units to 128

This will be done for each of the 6 different steps. Also, what is being done for each of the following steps can be found in the instructions for Assignment 3 in Machine learning or by looking at the heading for each subsection.

step 2: Change the activation function to reLU

```
model.step2 <- keras_model_sequential(input_shape = c(28, 28)) %>%
  # the layer_flattern function which flattern the given input
  layer_flatten() %>%
  # here we have the hidden layer which is being assigned sigmoid as
  # the activation function. The hidden layer consist of 16 units
  layer_dense(128, activation = "relu") %>%
  # this is the output layer which consist of 10 units from the range of
  # 0 to 9 and the activation function for the output layer is the softmax
  # function.
  layer_dense(units = nlevels( as.factor(y_train)), activation = "softmax")
```

```
model.step2 %>% compile(
  optimizer = "adam",
  loss = loss_fn,
  metrics = "accuracy"
)

object.step2 <- model.step2 %>% fit(
  x_train, y_train,
  epochs = 10, validation_split = 0.2)
```

validation accuracy and the validation loss for step 2, in that order, is being presented below:

```
object.step2$metrics$val_accuracy[length(object.step2$metrics$val_accuracy)]
```

```
## [1] 0.9785833
```

```
object.step2$metrics$val_loss[length(object.step2$metrics$val_loss)]
```

```
## [1] 0.0842077
```

figure for step 2

```
knitr::include_graphics("FigureStep2.png")
```

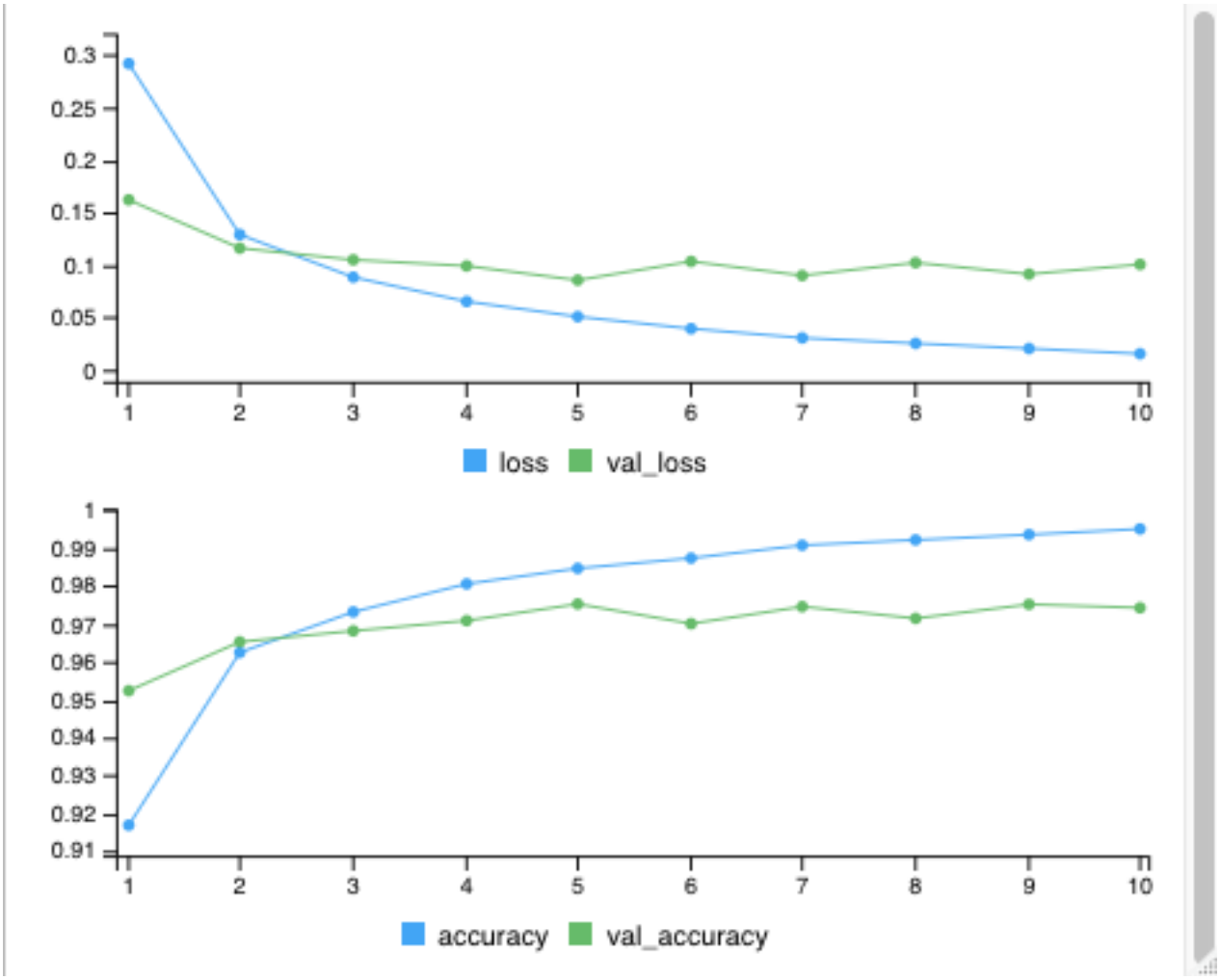


Figure 3: Result when changing the activation function to reLU

step 3: Change the optimizer to RMSprop

```
model.step3 <- keras_model_sequential(input_shape = c(28, 28)) %>%
  # the layer_flattern function whihc flattern the given input
  layer_flatten() %>%
  # here we have the hidden layer which is being assigned sigmoid as
  # the activation function. The hidden layer consist of 16 units
  layer_dense(128, activation = "relu") %>%
  # this is the output layer which consist of 10 units from the range of
  # 0 to 9 and the activation function for the output layer is the softmax
  # function.
  layer_dense(units = nlevels( as.factor(y_train)), activation = "softmax")

model.step3 %>% compile(
  # here we change the optimizer from
  # adam to rmsprop
```

```
  optimizer = "rmsprop",
  loss = loss_fn,
  metrics = "accuracy"
)

object.step3 <- model.step3 %>% fit(
  x_train, y_train,
  epochs = 10, validation_split = 0.2)
```

validation accuracy and the validation loss for step 3, in that order, is being presented below:

```
object.step3$metrics$val_accuracy[length(object.step3$metrics$val_accuracy)]
```

```
## [1] 0.9735
```

```
object.step3$metrics$val_loss[length(object.step3$metrics$val_loss)]
```

```
## [1] 0.1040162
```

figure for step 3

```
knitr::include_graphics("FigureStep3.png")
```

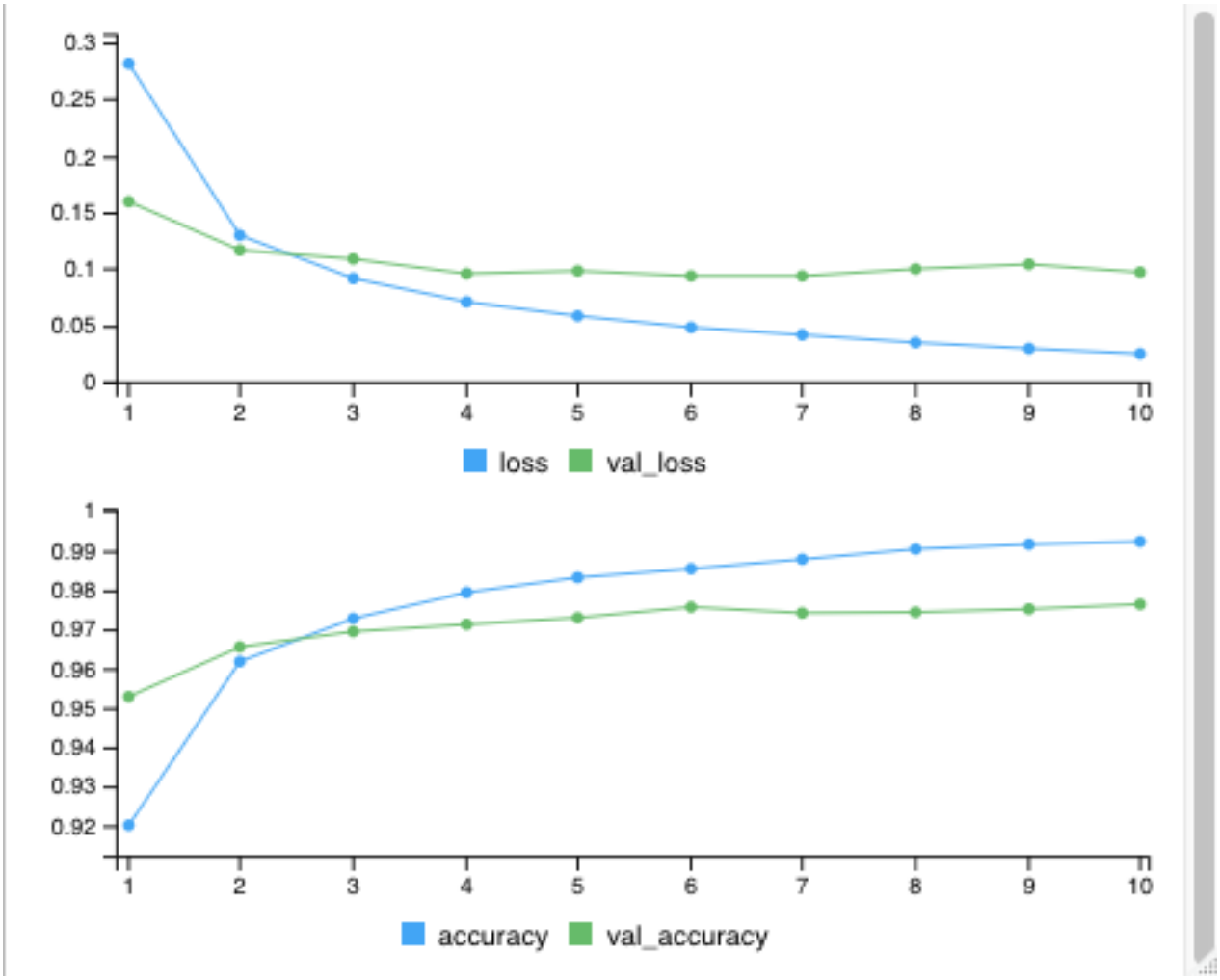


Figure 4: Result when changing the optimizer to RMSprop

step 4: Add a second layer with 128 hidden units

```
model.step4 <- keras_model_sequential(input_shape = c(28, 28)) %>%
  # the layer_flattern function whihc flattern the given input
  layer_flatten() %>%
  # here we have the hidden layer which is being assigned sigmoid as
  # the activation function. The hidden layer consist of 16 units
  layer_dense(128, activation = "relu") %>%
  # adding an extra hiddden layer with 128 hidden units
  # which is being asssigned relu as the activation function
  layer_dense(units = 128, activation = "relu") %>%
  # this is the output layer which consist of 10 units from the range of
  # 0 to 9 and the activation function for the output layer is the softmax
  # function.
  layer_dense(units = nlevels( as.factor(y_train)), activation = "softmax")
```

```
model.step4 %>% compile(
  # here we change the optimizer from
  # adam to rmsprop
  optimizer = "rmsprop",
  loss = loss_fn,
  metrics = "accuracy"
)

object.step4 <- model.step4 %>% fit(
  x_train, y_train,
  epochs = 10, validation_split = 0.2)
```

validation accuracy and the validation loss for step 4, in that order, is being presented below:

```
object.step4$metrics$val_accuracy[length(object.step4$metrics$val_accuracy)]
```

```
## [1] 0.9733334
```

```
object.step4$metrics$val_loss[length(object.step4$metrics$val_loss)]
```

```
## [1] 0.149727
```

figure for step 4

```
knitr::include_graphics("FigureStep4.png")
```

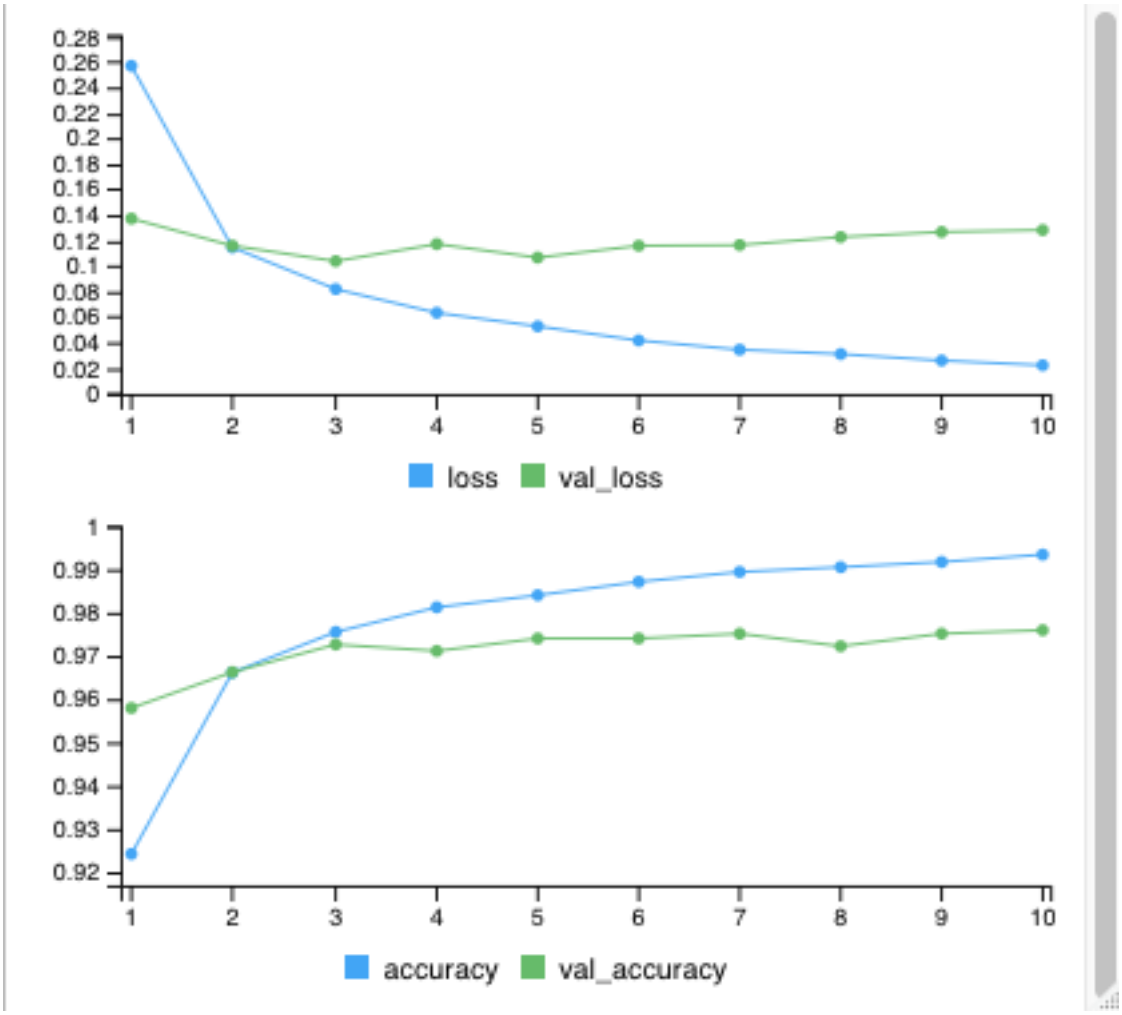


Figure 5: Result when adding a second layer with 128 hidden units

step 5: Adding 0.2 dropout probability for both of hidden layers

```
model.step5 <- keras_model_sequential(input_shape = c(28, 28)) %>%
  # the layer_flattern function whihc flattern the given input
  layer_flatten() %>%
  # here we have the hidden layer which is being assigned sigmoid as
  # the activation function. The hidden layer consist of 16 units
  layer_dense(128, activation = "relu") %>%
  # adding 0.2 dropout probability for first hidden layer
  layer_dropout(rate = 0.2) %>%
  # adding an extra hiddden layer with 128 hidden units
  # which is being asssigned relu as the activation function
  layer_dense(units = 128, activation = "relu") %>%
  # adding 0.2 dropout probability for second hidden layer
  layer_dropout(rate = 0.2) %>%
  # this is the output layer which consist of 10 units from the range of
```

```
  # 0 to 9 and the activation function for the output layer is the softmax
  # function.
  layer_dense(units = nlevels( as.factor(y_train)), activation = "softmax")

model.step5 %>% compile(
  # here we change the optimizer from
  # adam to rmsprop
  optimizer = "rmsprop",
  loss = loss_fn,
  metrics = "accuracy"
)

object.step5 <- model.step5 %>% fit(
  x_train, y_train,
  epochs = 10, validation_split = 0.2)
```

validation accuracy and the validation loss for step 5, in that order, is being presented below:

```
object.step5$metrics$val_accuracy[length(object.step5$metrics$val_accuracy)]
```

```
## [1] 0.9771667
```

```
object.step5$metrics$val_loss[length(object.step5$metrics$val_loss)]
```

```
## [1] 0.1179003
```

figure for step 5

```
knitr::include_graphics("FigureStep5.png")
```

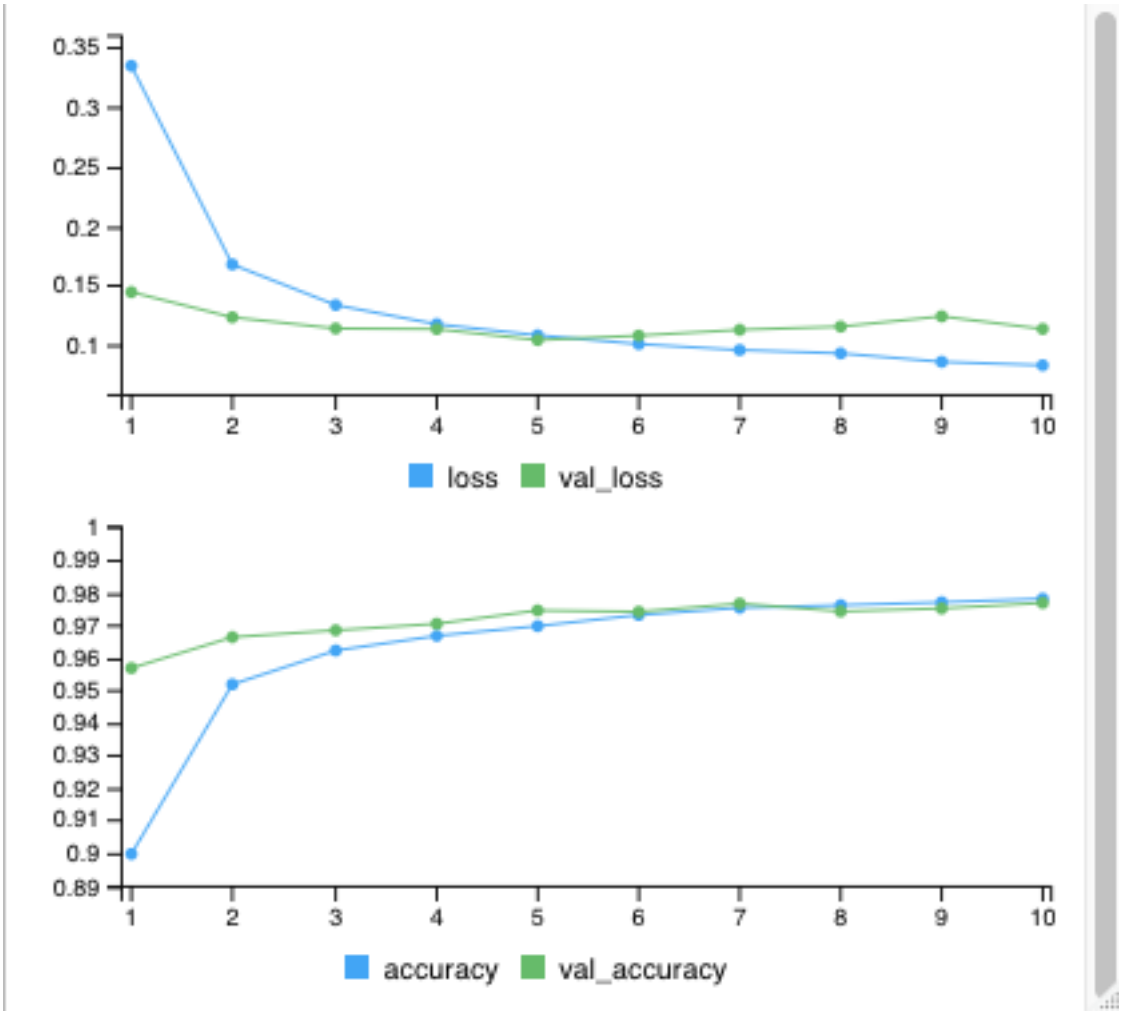


Figure 6: Adding 0.2 dropout for both of hidden layers

step 6: Add batch normalization for both of hidden layers

Now will we do the sixth and last step hence the output can be seen as the final neural network with all changes. For this assignment, this model will be called model2. The sixth step consists of adding a batch normalization for both hidden layers in the network and as before we will report the validation accuracy and the validation loss as well as the figure for the validation accuracy / loss per epoch.

```
model2 <- keras_model_sequential(input_shape = c(28, 28)) %>%
  # the layer_flattern function whihc flattern the given input
  layer_flatten() %>%
  # here we have the hidden layer which is being assigned sigmoid as
  # the activation function. The hidden layer consist of 16 units
  layer_dense(128, activation = "relu") %>%
  # adding 0.2 dropout probability for first hidden layer
  layer_dropout(rate = 0.2) %>%
  # adding a batch normalization for the first hidden layer
  layer_batch_normalization() %>%
  # adding an extra hiddden layer with 128 hidden units
  # which is being asssigned relu as the activation function
  layer_dense(units = 128, activation = "relu") %>%
  # adding 0.2 dropout probability for second hidden layer
  layer_dropout(rate = 0.2) %>%
  # adding a batch normalization for the second hidden layer
  layer_batch_normalization() %>%
  # this is the output layer which consist of 10 units from the range of
  # 0 to 9 and the activation function for the output layer is the softmax
  # function.
  layer_dense(units = nlevels( as.factor(y_train)), activation = "softmax")

model2 %>% compile(
  # here we change the optimizer from
  # adam to rmsprop
  optimizer = "rmsprop",
  loss = loss_fn,
  metrics = "accuracy"
)

object2 <- model2 %>% fit(
  x_train, y_train,
  epochs = 10, validation_split = 0.2)
```

validation accuracy and the validation loss for model2, in that order, is being presented below:

```
object2$metrics$val_accuracy[length(object2$metrics$val_accuracy)]
```

```
## [1] 0.9781666
```

```
object2$metrics$val_loss[length(object2$metrics$val_loss)]
```

```
## [1] 0.07858627
```

and the figure for model2 when applying step 6 looks like:

```
knitr::include_graphics("FigureModel2.png")
```

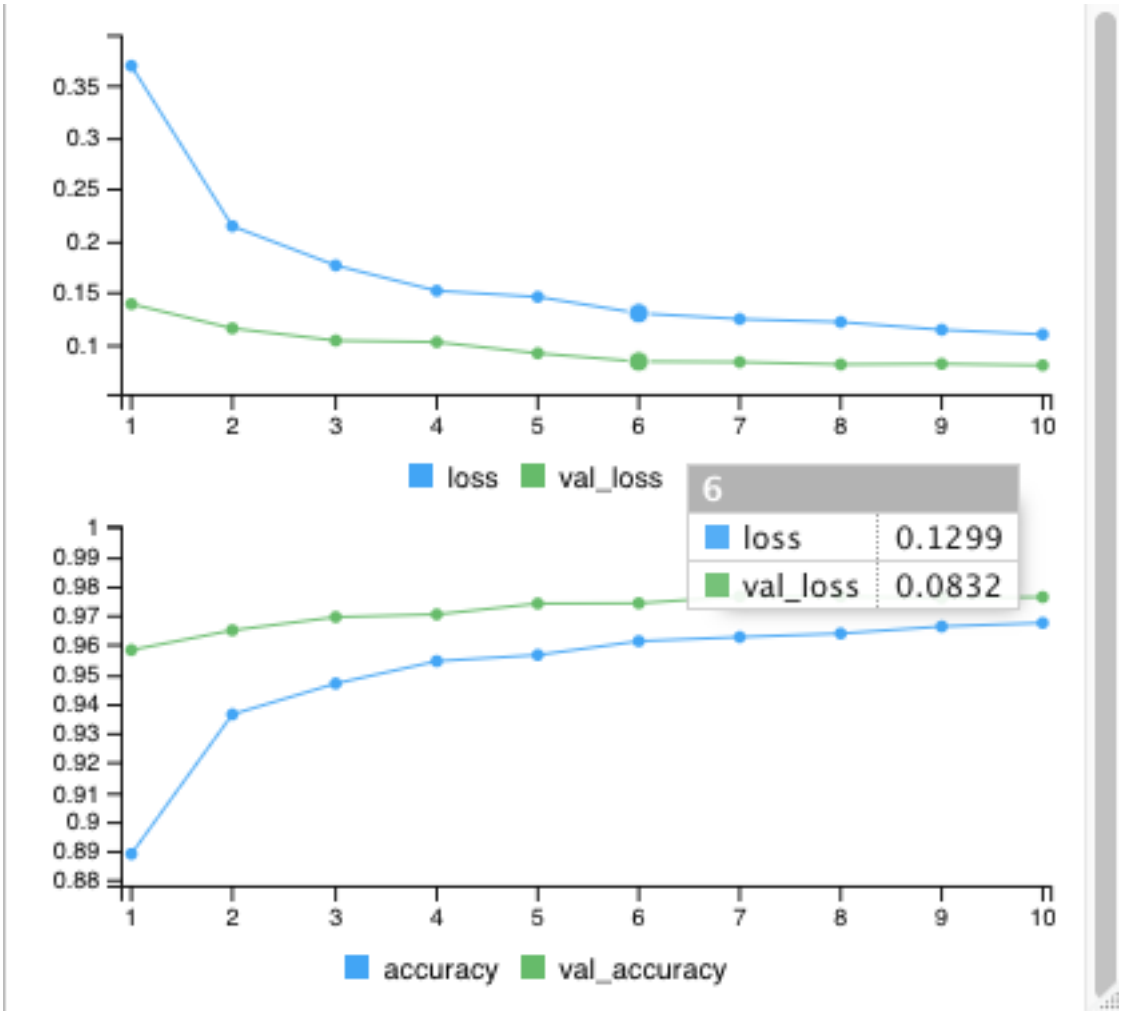


Figure 7: Adding a batch normalization for both of hidden layers

1.5

If we were to include a stopping regularization to the network then it would be to stop the training of the neural network at an earlier epoch by stopping the process if the training loss would be too small. The would be included to avoid overfitting the the data and making the network more generalized, hence performing better on the testing dataset.

1.6

Earlier we have created a feed forward neural network by starting at the simple case and then changing the structure of the neural network by playing around with different neural network concepts. Finally we created model2. Now we will try to build the best possible network architecture for the data, still within the framework of the feed forward neural network.

```
model3 <- keras_model_sequential(input_shape = c(28, 28)) %>%
  layer_flatten() %>%
```

```
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.11) %>%

  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.11) %>%

  layer_dense(units = 100, activation = "relu") %>%
  layer_dropout(rate = 0.11) %>%
  layer_batch_normalization() %>%

  layer_dense(units = 10, activation = "softmax")

model3 %>% compile(
  optimizer = "adam",
  loss = loss_fn,
  metrics = "accuracy"
)

object3 <- model3 %>% fit(
  x_train, y_train,
  epochs = 10, validation_split = 0.2)
```

Above we can see the final keras code for the best possible feed forward neural network we were able to build. The validation accuracy and the validation loss for model3, in that order, corresponds to the following values:

```
object3$metrics$val_accuracy[length(object3$metrics$val_accuracy)]
```

```
## [1] 0.97875
```

```
object3$metrics$val_loss[length(object3$metrics$val_loss)]
```

```
## [1] 0.07984529
```

Even if the neural network managed to reach a high validation accuracy and a low validation loss we do not believe that the network has yet reached the Bayes error, i.e the minimum possible error for the network.

We came up with the final keras code and architecture of the feed forward neural network by first creating a complicated network that was massively overfitting the data for later simplify the network.

Regarding the architecture of the network, we have built a feed forward neural network consisting of one input layer, one output layer and three hidden layers. The first, second and third hidden layer consist of 128, 128 and 100 hidden units, with "relu" as the activation function for all hidden layers. Further, we used a dropout with 0.11 dropout probability for all three hidden layers. The output layer is a softmax.

1.7

Based on the network created above named model3 in section 1.6 we will identify two digits that has been missclassified. We used the first two missclassified digits. Further, we will visualize the handwritten digits and include the digit number and what the network classified them as.

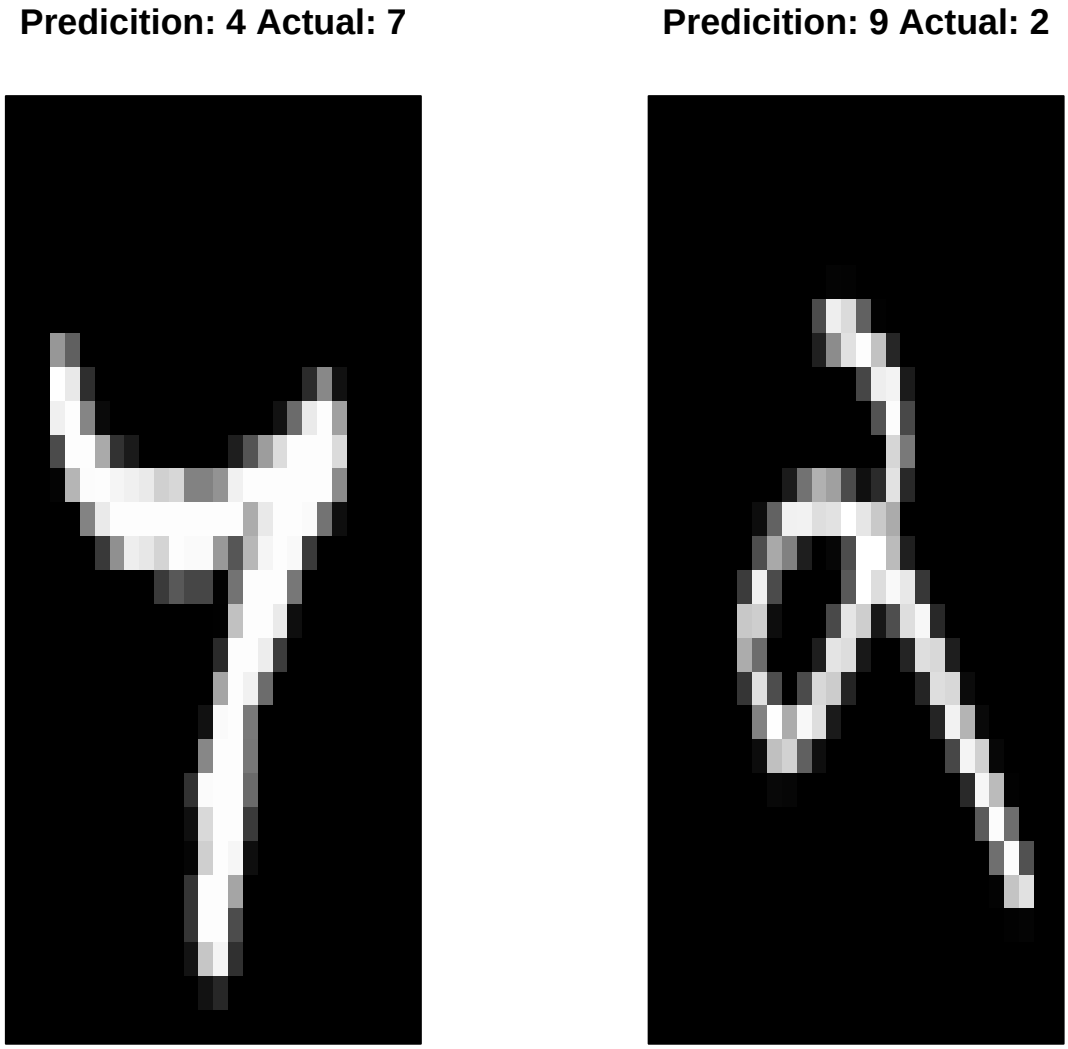
```
prediction <- model3 %>% predict(x_test)
```

```
## 313/313 - 1s - 696ms/epoch - 2ms/step
```

```
pred_classification <- apply(prediction, MARGIN = 1, which.max ) - 1
```

```
missclass <- which(pred_classification != y_test)
sample_missclass <- missclass[c(1,2)]

par(mfrow = c(1,2))
for (i in 1:2) {
idx <- sample_missclass[i]
im <- x_test[idx,,]
im <- t(apply(im, 2, rev))
image(1:28, 1:28, im, col=gray((0:255)/255), xlab = "", ylab = "",
      xaxt='n', yaxt='n',
      main=paste("Predicition:",pred_classification[idx],
                 "Actual:", y_test[idx]))
}
```



Above, we can see the first two digits where the neural network did misclassify the handwritten digits. When looking at the digits that the neural network has missclassified then it would be difficult for the human brain to classify them correctly, ie many people would probability get it wrong. Therefore, this odd hand styles is also difficult for the network to classify.

1.8

Now we will use the model to compute the accuracy of the hold out test set.

```
model3 %>% evaluate(x_test,
                    y_test, verbose = 2)
```

```
## 313/313 - 1s - loss: 0.0697 - accuracy: 0.9809 - 826ms/epoch - 3ms/step
##       loss   accuracy
## 0.06965498 0.98089999
```

Recall, the validation accuracy for the model when using the training data set. The validation for the hold out test set is around the same value as the validation accuracy for the training dataset. The same for the validation loss which is logical since the model is supposed to result in around the same validation accuracy for the sample that is not being included in the training dataset. However, the validation accuracy is still lower than the normal accuracy that was being computed based on the training dataset which indicates that the built model is overfitting the MNIST dataset.

Task 2: A simple neural network from scratch

2.1

The function below implement a neural network and consist of 5 function arguments and it takes as an input a 2×2 weight matrix W, a vector c, a vector w and a scalar b. (Reference: Goodfellow et al, chapter: 6, subchapter: 1)

The output of the network function will be based on the following equation:

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top g\left\{\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\right\} + b$$

where $\hat{y}$, $\boldsymbol{x}$, $\boldsymbol{c}$ is vectors of the ith element in $\hat{\boldsymbol{Y}}$, $\boldsymbol{X}$, $\boldsymbol{C}$ respectively. Also, $g(\cdot)$ denotes the activation function which is being applied element wise. For this case we will use the RELU activation function $g(z) = \text{ReLU}(z) = \max(0, z)$ which makes it possible to rewrite the equation:

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\left\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\right\} + b$$

Further, we denote the result of the implemented network as $\hat{\boldsymbol{y}}$ which corresponds to a vector.

```
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

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

  output <- matrix(NA, ncol = 1, nrow = nrow(X))

  for ( i in 1:nrow(X)) {
  RELU <- apply(cbind(0, t(W) %*% X[i,] + c[i,] ), 1 , max)
  output[i,] <- t(w)  %*%  RELU + b
  }

  return(output)
}
```

```
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
```

2.2

Now, we change the value of the element in the first row and column in the 2×2 weight matrix to 0, i.e $W_{1,1} = 0$ and calculating the predicted output.

```
W1 <-  W
W1[1,1] <- 0
mini_net(X, W1, c, w, b)
```

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

In matrix notation, when changing the value $W_{1,1} = 0$ we have that the result of the network corresponds to $\hat{\boldsymbol{y}} = (0, 1, 0, -1)'$.

2.3

The current output function is a simple feed forward neural network with one hidden layer and two hidden units where we set the activation function for the single hidden layer to RELU. We train the network of all four point in X consisting of two binary values. The simple feed forward neural network is structured in a such way that we have two function chained together i.e $f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)$, which looks like the following for output y:

$$y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$$

where $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$. We have that the $\boldsymbol{h}$ function are able to capture the nonlinear features in the training dataset since the output y is a linear function of $\boldsymbol{h}$ instead of $\boldsymbol{x}$ for the network. The network output function can be seen as reasonable since it is able to capture the nonlinear features in the data. However regarding the activation function $g(\cdot)$, we assigned the RELU to the network, which is linear for $x > 0$, but there is other (nonlinear) alternatives such as the sigmoid function $\sigma(z) = \frac{e^z}{1+e^z}$.

2.4

Now we will implement a function that computes the mean squared error loss function, denoted as $J(\boldsymbol{\theta})$, for the network and we will use the following formula to calculate the mean squared error loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=j}^{m}(y_j - \hat{y}_j)^2, j = (1, ..., m)$$

where $J(\boldsymbol{\theta})$ denotes the mean squared error loss function and m denotes the length of the $\hat{\boldsymbol{y}} : mx1$ vector.

```
i  = 1
y <- c(0,1,1,0)
mini_net_loss <- function(y, X, W, c, w, b) {
  predication  <-  mini_net(X, W, c, w, b)
  output <- sum((y - predication)^2) / length(predication)
  return(output)
}
mini_net_loss(y, X, W, c, w, b)
```

```
## [1] 0
```

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

```
## [1] 0.015
```

2.5

```
mini_net_loss(y, X, W1, c, w, b)
```

```
## [1] 0.5
```

When changing the value $W_{1,1}$ to 0 then we get that the value of the loss function corresponds to 0.5.