Machine Learning Assignment 2

16343261

03/04/2023

To begin this assignment, we will first load in our necessary packages.

```
library(keras)
library(tfruns)
library(reticulate)
library(jsonlite)
```

Exercise 1.

Part 1.

This multi-layer neural network has a size of 11 nodes, 3 of which are bias. With 2 hidden layers, it has a depth of 2. Hidden layer 1 has a width of 3 and Hidden layer 2 has a width of 3. The number of parameters here is 20.

Part 2.

First we will load in our input observations and the matrix for our input layer.

```
Input_obs <- c(-0.5,0.3)

Input_vec <- matrix(c(-0.5,0.4,-0.2, -0.3,-0.1,0.4), nrow = 2, ncol = 3,
byrow = TRUE, dimnames = list(c("row1", "row2"), c("C.1", "C.2", "C.3")))
Input_vec
```

```
##       C.1  C.2  C.3
## row1 -0.5  0.4 -0.2
## row2 -0.3 -0.1  0.4
```

Next we will define the matrix and vector for our respective hidden layers.

```
H1_vec <- matrix(c(0.3,-0.7,1.3, 0.5,-0.8,1.2), nrow = 3, ncol = 2, byrow =
TRUE,
dimnames = list(c("row1", "row2","row3"), c("C.1", "C.2")))
H2_vec <- c(2, 1.1)
H1_vec
```

```
##       C.1  C.2
## row1  0.3 -0.7
## row2  1.3  0.5
## row3 -0.8  1.2
```

And finally the vectors for our bias variables.

```
b1 <- c(1.1,-0.8,1.3)
b2 <- c(0.5,-0.8)
b3 <- 8
```

Next we will calculate the outputs for out input layer followed by the inputs into our sigmoid Hidden Layer 1.

```
out_1 <- Input_obs[1]*Input_vec[1,]
out_2 <- Input_obs[2]*Input_vec[2,]

HL1 <- NULL
for(i in 1:3){
HL1[i] <- out_1[i] + out_2[i]+b1[i]
}
```

Here we define our sigmoid function and find our outputs for our first hidden layer.

```
sigmoid <- function(x) {
  sigma <- 1/(1+exp(-x))
  return(sigma)
}
input_HL1 <- sigmoid(HL1)
input_HL1

## [1] 0.7790261 0.2630841 0.8205385
```

This is defined as “input_HL1” as these our inputs for our subsequent hidden layer which we will calculate now.

```
out_3 <- input_HL1[1]*H1_vec[1,]
out_4 <- input_HL1[2]*H1_vec[2,]
out_5 <- input_HL1[3]*H1_vec[3,]
HL2 <- NULL
for(i in 1:2){
HL2[i] <- out_3[i] + out_4[i]+ out_5[i]+b2[i]
}
```

Next we will define our Rectified Linear Unit function for our second hidden layer and calculate our outputs for our output layer.

```
Relu <- function(x) {
n <- length(x)
sigma <- NULL
for(i in 1:n){
  sigma[i] <- max(x[i],0)
}
  return(sigma)
}
input_HL2 <- Relu(HL2)
input_HL2
```

```
## [1] 0.4192864 0.0000000

out_6 <- input_HL2[1]*H2_vec[1]
out_7 <- input_HL2[2]*H2_vec[2]
```

And finally find our output.

```
OL <- out_6 + out_7 + b3
OL

## [1] 8.838573
```

Part 3.

Now let's define our target variable.

```
target <- 7
```

An appropriate loss function is the squared loss function. This is defined below.

Squared loss:

```
Squared_loss <- function(x,y) {
n <- length(x)
loss <- NULL
for(i in 1:n){
  loss[i] <- (y[i] - x[i])^2
  sum <- sum(loss)
}
  return(sum)
}
```

Our calculation.

```
Squared_loss(OL, target)

## [1] 3.38035
```

Exercise 2.

Part 1.

2048 units.

Part 2.

544 batches.

Part 3.

The task being employed here is a Multi-layer neural network. This network has a depth of 3 with 3 Leaky Rectified Linear Unit layers compiling the categorical cross entropy and classification accuracy.

Part 4.

Early Stopping Regularisation.

Exercise 3.

Part 1.

To begin this exercise, let's load in our Epileptic dataset.

```
setwd("C:/Users/matth/Documents/Machine Learning & AI/Assignment2")
load("C:/Users/matth/Documents/Machine Learning &
AI/Assignment2/data_epileptic.RData")
```

To begin deploying our neural networks, we must first define our 2 separate models. Our first model will contain 3 hidden layers. Our first layer using sigmoid activation and our other 2 using relu.

```
# Model1 Code.
FLAGS <- flags(
  flag_numeric("dropout", 0.4),
  flag_numeric("lambda", 0.01),
  flag_numeric("lr", 0.01),
  flag_numeric("bs", 100)
)

# model1 configuration
model1 <- keras_model_sequential() %>%
  layer_dense(units = 128, input_shape = V, activation = "sigmoid", name =
"layer_1",
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = 64, activation = "relu", name = "layer_2",
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = 32, activation = "relu", name = "layer_3",
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = ncol(y), activation = "softmax", name = "layer_out")
%>%
  compile(loss = "categorical_crossentropy", metrics = "accuracy",
          optimizer = optimizer_adam(learning_rate = FLAGS$lr),
  )
```

```
# training and evaluation
fit1 <- model1 %>% fit(
  x = x, y = y,
  validation_data = list(x_val, y_val),
  epochs = 100,
  batch_size = FLAGS$bs,
  verbose = 1,
  callbacks = callback_early_stopping(monitor = "val_accuracy", patience =
20)
)

# store accuracy on test set for each run
score1 <- model1 %>% evaluate(
  x_test, y_test,
  verbose = 0
)

runs_model1 <- tuning_run("model1.R",
runs_dir = "runs_model1",
flags = list(
dropout = dropout_set,
lambda = lambda_set,
lr = lr_set,
bs = bs_set
),
sample = 0.1)
```

Our second model will use 2 hidden layers. Our first being a Relu layer and sigmoid being our second layer.

```
# Model2 Code.
FLAGS <- flags(
  flag_numeric("dropout", 0.4),
  flag_numeric("lambda", 0.01),
  flag_numeric("lr", 0.01),
  flag_numeric("bs", 100)
)

# model configuration
model2 <- keras_model_sequential() %>%
  layer_dense(units = 128, input_shape = V, activation = "relu", name =
"layer_1",
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = 32, activation = "sigmoid", name = "layer_2",
              kernel_regularizer = regularizer_l2(FLAGS$lambda)) %>%
  layer_dropout(rate = FLAGS$dropout) %>%
  layer_dense(units = ncol(y), activation = "softmax", name = "layer_out")
%>%
```

```
  compile(loss = "categorical_crossentropy", metrics = "accuracy",
          optimizer = optimizer_adam(learning_rate = FLAGS$lr),
  )

fit2 <- model2 %>% fit(
  x = x, y = y,
  validation_data = list(x_val, y_val),
  epochs = 100,
  batch_size = FLAGS$bs,
  verbose = 1,
  callbacks = callback_early_stopping(monitor = "val_accuracy", patience =
20)
)

# store accuracy on test set for each run
score2 <- model2 %>% evaluate(
  x_test, y_test,
  verbose = 0
)

runs_model2 <- tuning_run("model2.R",
runs_dir = "runs_model2",
flags = list(
dropout = dropout_set,
lambda = lambda_set,
lr = lr_set,
bs = bs_set
),
sample = 0.1)
```

Our below function will allow the extraction of values from the stored runs and plot the corresponding validation learning curves.

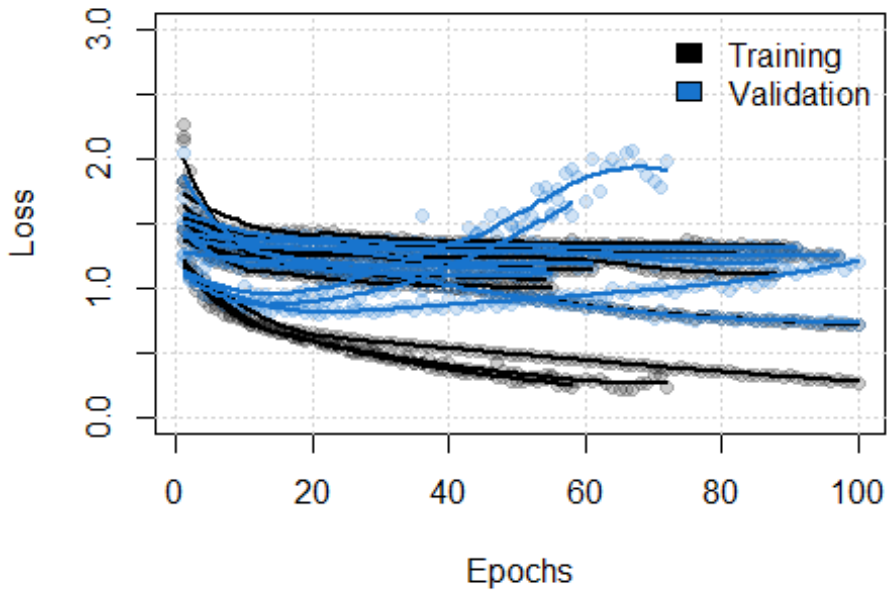
```
read_metrics <- function(path, files = NULL)
# 'path' is where the runs are --> e.g. "path/to/runs"
{
path <- paste0(path, "/")
if ( is.null(files) ) files <- list.files(path)
n <- length(files)
out <- vector("list", n)
for ( i in 1:n ) {
dir <- paste0(path, files[i], "/tfruns.d/")
out[[i]] <- jsonlite::fromJSON(paste0(dir, "metrics.json"))
out[[i]]$flags <- jsonlite::fromJSON(paste0(dir, "flags.json"))
out[[i]]$evaluation <- jsonlite::fromJSON(paste0(dir, "evaluation.json"))
}
return(out)
}
```

Function for smooth lines.

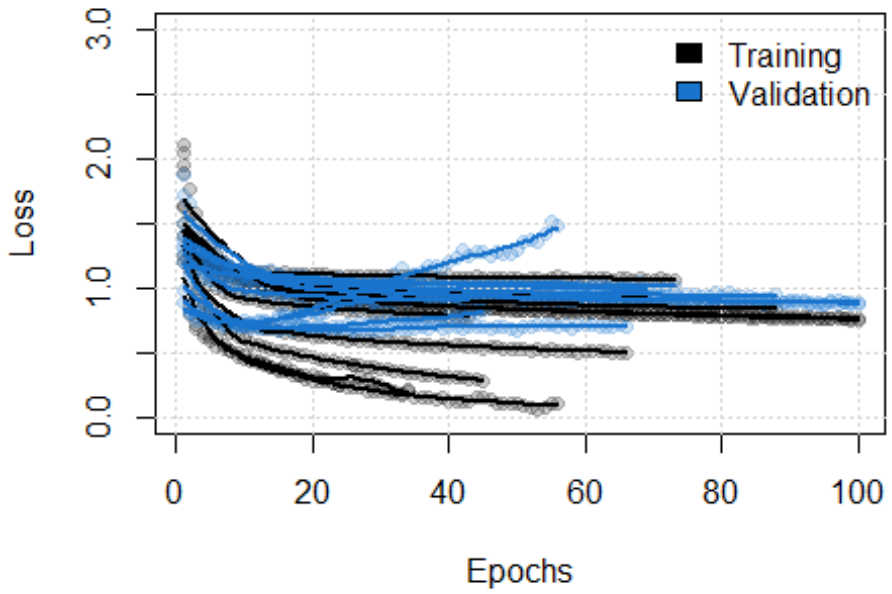
```
# to add a smooth line to points
smooth_line <- function(y, span = 0.3) {
x <- 1:length(y)
out <- predict( loess(y ~ x, span = span) )
return(out)
}

# extract results from folders
out_m1 <- read_metrics("runs_model1")
out_m2 <- read_metrics("runs_model2")

# extract training and validation scores
train_acc_m1 <- sapply(out_m1, "[[", "accuracy")
val_acc_m1 <- sapply(out_m1, "[[", "val_accuracy")
train_loss_m1 <- sapply(out_m1, "[[", "loss")
val_loss_m1 <- sapply(out_m1, "[[", "val_loss")

train_acc_m2 <- sapply(out_m2, "[[", "accuracy")
val_acc_m2 <- sapply(out_m2, "[[", "val_accuracy")
train_loss_m2 <- sapply(out_m2, "[[", "loss")
val_loss_m2 <- sapply(out_m2, "[[", "val_loss")

# select the top 10 runs by validation accuracy
sel <- 10
top_m1 <- order(apply(val_acc_m1, 2, max, na.rm = TRUE), decreasing =
TRUE)[1:sel]
top_m2 <- order(apply(val_acc_m2, 2, max, na.rm = TRUE), decreasing =
TRUE)[1:sel]
```

Now let's plot our loss curves for both of our models.

```
# plot loss curves to inspect training process and underfitting/overfitting
cols <- rep(c("black", "dodgerblue3"), each = sel)
out_loss_m1 <- cbind(train_loss_m1[,top_m1], val_loss_m1[,top_m1])
matplot(out_loss_m1, pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.2), ylim = c(0, 3))
grid()
tmp_m1 <- apply(out_loss_m1, 2, smooth_line, span = 0.5)
tmp_m1 <- sapply(tmp_m1, "length<-", 100 )
# set default length of 100 epochs
matlines(tmp_m1, lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
fill = unique(cols), bty = "n")
```

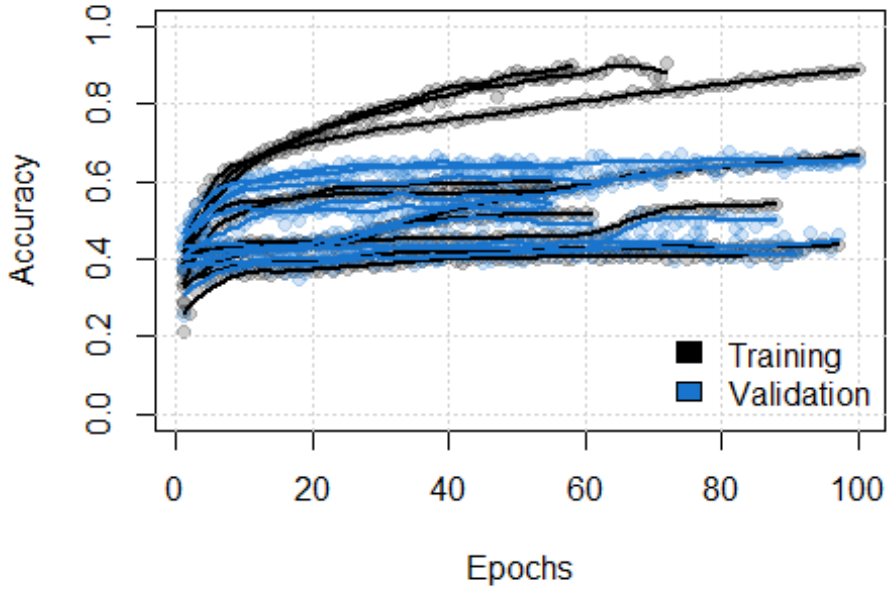



```
out_loss_m2 <- cbind(train_loss_m2[,top_m2], val_loss_m2[,top_m2])
matplot(out_loss_m2, pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.2), ylim = c(0, 3))
grid()
tmp_m2 <- apply(out_loss_m2, 2, smooth_line, span = 0.5)
tmp_m2 <- sapply(tmp_m2, "length<-", 100 )
# set default length of 100 epochs
matlines(tmp_m2, lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
fill = unique(cols), bty = "n")
```

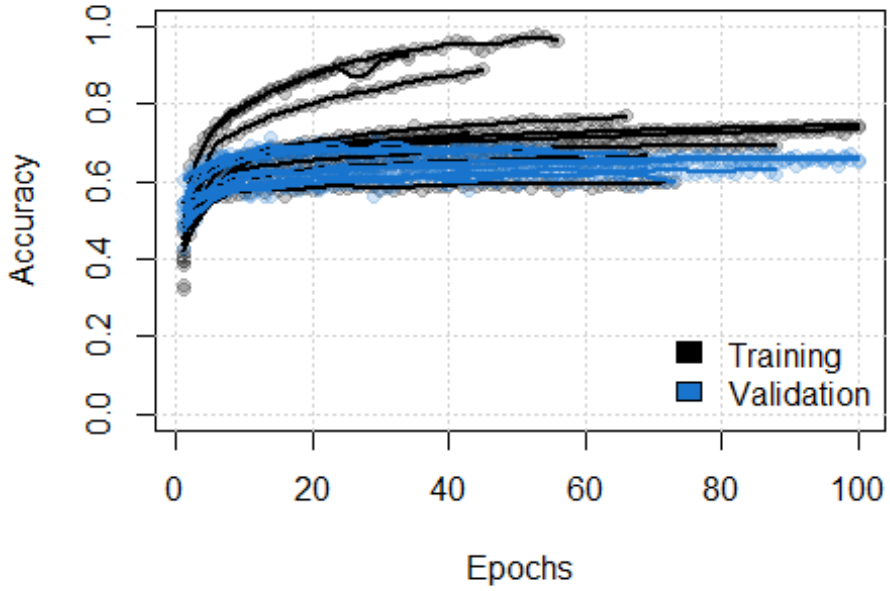


Next let's plot our accuracy curves to inspect our performance and overfitting.

```
# Accuracy curves
out_acc_m1 <- cbind(train_acc_m1[,top_m1], val_acc_m1[,top_m1])
matplot(out_acc_m1, pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.2), ylim = c(0, 1))
grid()
tmp_m1 <- apply(out_acc_m1, 2, smooth_line)
tmp_m1 <- sapply(tmp_m1, "length<-", 100 )
matlines(tmp_m1, lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = unique(cols), bty = "n")
```




```
out_acc_m2 <- cbind(train_acc_m2[,top_m2], val_acc_m2[,top_m2])
matplot(out_acc_m2, pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.2), ylim = c(0, 1))
grid()
tmp_m2 <- apply(out_acc_m2, 2, smooth_line)
tmp_m2 <- sapply(tmp_m2, "length<-", 100 )
matlines(tmp_m2, lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = unique(cols), bty = "n")
```



Here we see the training process does not look particularly good for either model, with considerable signs of overfitting as displayed in the loss curves plots for both models. Definitely a simpler full Relu architecture would be of help in this regard as a sigmoid hidden layers have a tendancy to cause overfitting. Furthermore extra regularisation could help in this regard. It is clear from each plot that model 2 has the better performance with both higher.

```
res_m1 <- ls_runs(metric_val_accuracy > 0.6,
runs_dir = "runs_model1", order = metric_val_accuracy)
res_m2 <- ls_runs(metric_val_accuracy > 0.6,
runs_dir = "runs_model2", order = metric_val_accuracy)

colu_m1 <- c("metric_val_accuracy", grep("flag", colnames(res_m1), value =
TRUE), "epochs_completed")
colu_m2 <- c("metric_val_accuracy", grep("flag", colnames(res_m2), value =
TRUE), "epochs_completed")

res_m1[1:5,colu_m1]

## Data frame: 5 x 6
##    metric_val_accuracy flag_dropout flag_lambda flag_lr flag_bs
## 1               0.6551          0.3           0   0.005   303.6
## 2               0.6522          0.0           0   0.001   101.2
## 3               0.6449          0.0           0   0.005   101.2
## 4               0.6290          0.0           0   0.010   202.4
## NA                  NA           NA          NA      NA      NA
##    epochs_completed
```

```
## 1                100
## 2                100
## 3                 58
## 4                 72
## NA                NA

res_m2[1:5,colu_m2]

## Data frame: 5 x 6
##   metric_val_accuracy flag_dropout flag_lambda flag_lr flag_bs
epochs_completed
## 1              0.6986          0.0 0.000000000   0.010   202.4
34
## 2              0.6826          0.0 0.000000000   0.001   101.2
45
## 3              0.6768          0.0 0.000000000   0.005   101.2
56
## 4              0.6739          0.3 0.000000000   0.005   303.6
66
## 5              0.6565          0.0 0.002478752   0.001   303.6
100
```

Overall validation for Model 1 is noticeably low with validation skewed considerably between 0.4 and 0.6 accuracy. Model 2 on the other hand appears to be considerably more accurate with most of our validation more than 0.6. For model 1, our best fitted model had a validation accuracy of 0.66 and for model 2 we had a validation accuracy of 0.7. In fact all of our top five models for model 2 had a validation accuracy higher than model 1. From this, we can assert that model 2 with only 2 hidden layers had a higher validation accuracy.

The best model of model 2 has no weight decay, a dropout rate of 0, a learning rate of 0.01 and a batch size of 202. Our top 5 configurations all have no dropout rate and lambda with varying learning rates and batch sizes. This question can be further answered in our part 3 answer.

Part 2.

It would appear from our findings that adding an additional hidden layer does not lead to an improvement in predictive performance as evidenced by the increased loss in our model 1 compared to our model 2.

Part 3.

Below we will continue the tuning procedure for both models and discover which has the best accuracy at predicting EEG signals of patients with tumour formations. For a reason out of my control there is an error that occurs when loading keras_model_sequential() into rmarkdown that does not occur in base R. I have included screenshots of my output in base r. All of this code should run smoothly and is replicatable in base r.

```
# deploy model using optimal hyperparameters
model1 <- keras_model_sequential() %>%
```

```
   layer_dense(units = 128, input_shape = V, activation = "sigmoid", name =
"layer_1") %>%
  layer_dropout(rate = res_m1$flag_dropout[1]) %>%
  layer_dense(units = 64, activation = "relu", name = "layer_2") %>%
  layer_dropout(rate = res_m1$flag_dropout[1]) %>%
  layer_dense(units = 32, activation = "relu", name = "layer_3",) %>%
  layer_dropout(rate = res_m1$flag_dropout[1]) %>%
  layer_dense(units = ncol(y), activation = "softmax", name = "layer_out")
%>%
  compile(loss = "categorical_crossentropy", metrics = "accuracy",
optimizer = optimizer_adam(learning_rate = res_m1$flag_lr[1]),
)

x_ <- rbind(x, x_val)
y_ <- rbind(y, y_val)
# fit on full data and test on test data
fit <- model1 %>% fit(
x = x_, y = y_,
validation_data = list(x_test, y_test),
epochs = 100,
batch_size = res_m1$flag_bs[1],
verbose = 1,
callbacks = callback_early_stopping(monitor = "val_accuracy", patience = 30)
)

# estimated classes and actual digits
class_hat <- model1 %>% predict(x_test) %>% max.col() - 1
y_labels_test <- max.col(y_test) - 1
# check performance and class-specific performance
tab <- table(y_labels_test, class_hat)
acc <- diag(tab)/rowSums(tab)
cbind(tab, acc)
```

Output:

```
> acc <- diag(tab)/rowSums(tab)
> cbind(tab, acc)
     1  2   3  4   5       acc
1 119 13   1  0   0 0.8947368
2   1 31 103  1   8 0.2152778
3   1 15 105  4  13 0.7608696
4   1  0   0 87  49 0.6350365
5   0  0  18  6 114 0.8260870
```

```
# deploy model using optimal hyperparameters
model2 <- keras_model_sequential() %>%
  layer_dense(units = 128, input_shape = V, activation = "relu", name =
"layer_1") %>%
  layer_dropout(rate = res_m2$flag_dropout[1]) %>%
  layer_dense(units = 32, activation = "sigmoid", name = "layer_2") %>%
```

```
  layer_dropout(rate = res_m2$flag_dropout[1]) %>%
  layer_dense(units = ncol(y), activation = "softmax", name = "layer_out")
%>%
  compile(loss = "categorical_crossentropy", metrics = "accuracy",
optimizer = optimizer_adam(learning_rate = res_m2$flag_lr[1]),
)
# merge train and validation data
x_ <- rbind(x, x_val)
y_ <- rbind(y, y_val)
# fit on full data and test on test data
fit <- model2 %>% fit(
x = x_, y = y_,
validation_data = list(x_test, y_test),
epochs = 100,
batch_size = res_m2$flag_bs[1],
verbose = 1,
callbacks = callback_early_stopping(monitor = "val_accuracy", patience = 30)
)

# estimated classes and actual digits
class_hat <- model2 %>% predict(x_test) %>% max.col() - 1
y_labels_test <- max.col(y_test) - 1
# check performance and class-specific performance
tab <- table(y_labels_test, class_hat)
acc <- diag(tab)/rowSums(tab)
cbind(tab, acc)
```

Output:

```
> acc <- diag(tab)/rowSums(tab)
> cbind(tab, acc)
    1  2  3   4  5       acc
1 124  6  3   0  0 0.9323308
2   6 87 43   0  8 0.6041667
3   4 40 80   3 11 0.5797101
4   0  0  3 106 28 0.7737226
5   0 10 11  25 92 0.6666667
```

Here we can find that our best model for predicting EEG activity of patients with tumour formations is again model 2 with 2 hidden layers. We can see this as our second most accurate model for model 2 with 0.604 accuracy and 87 values accurately identified from our samples.

Part 1 continued.

From this analysis we once again find model 2 to have our best accuracy with an overall higher digit specific accuracy for values 1, 2 and 4 and quite similar to model 1 for 3 and 5.