Machine Learning 2ST129 26605 HT2023 Assignment 4

Anonymous Student

December 3, 2023

Contents

General Information

- Time used for reading: 2 hours:
- Time used for basic assignment: 18 hours
- Time used for extra assignment: 6 hours
- Good with lab: It was good that you had had to tune different hyper parameters and think about what they are doing.
- Things improve with lab: Maybe more explanation about how to plot images and how to code works. I do not really know what I am doing, I just copy paste the code given for some part and use other packages for another part.

1 Task 1

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

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

1.1 1.1

First we visualize the digits.

```
#' 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(1,3))

  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(1:3))
```

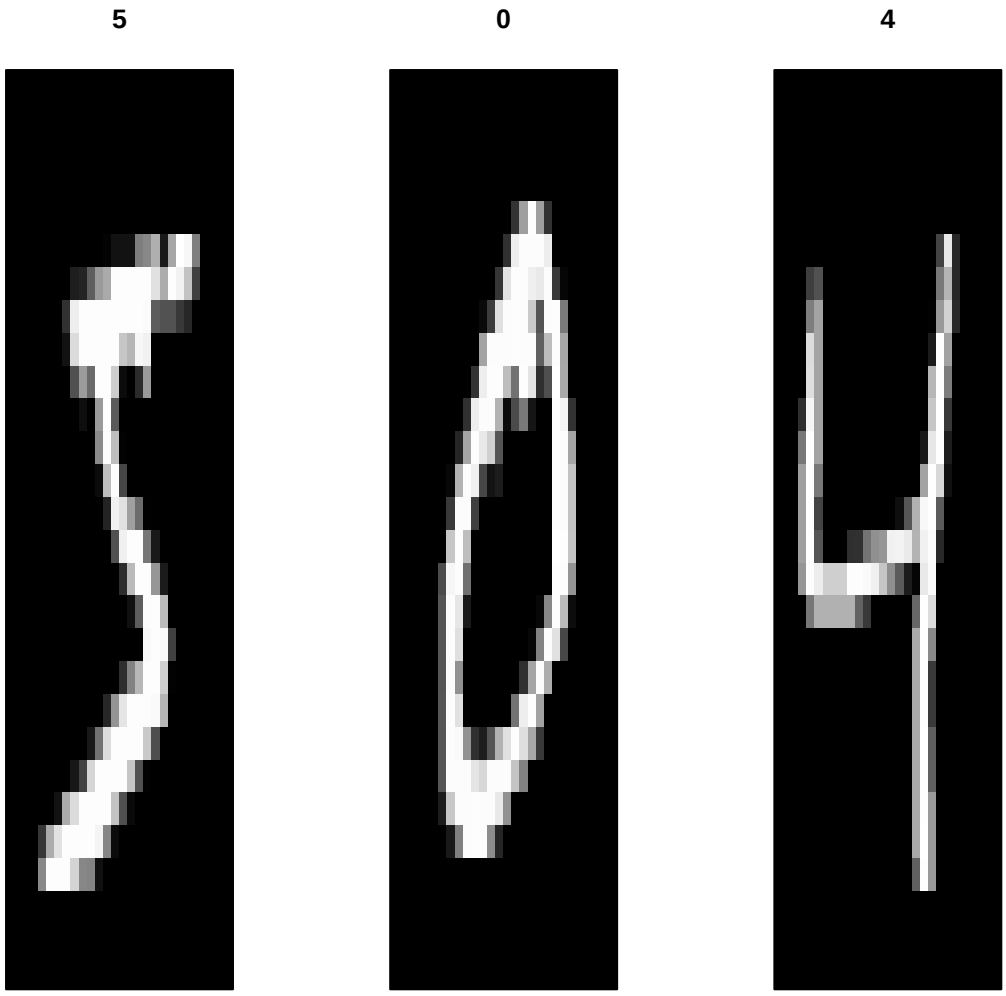


Figure 1: Visualization of first 3 digits in mnist data set

1.2 1.2 Convulutional neural network

```
model1 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = 'relu',
                input_shape = c(28, 28, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = 'relu') %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')


summary(model1)
```

```
Model: "sequential"
________________________________________________________________________________
 Layer (type)                        Output Shape                      Param #
================================================================================
```

```
 conv2d_1 (Conv2D)                   (None, 26, 26, 32)              320
 max_pooling2d (MaxPooling2D)        (None, 13, 13, 32)              0
 conv2d (Conv2D)                     (None, 11, 11, 32)              9248
 flatten (Flatten)                   (None, 3872)                    0
 dense_1 (Dense)                     (None, 64)                      247872
 dense (Dense)                       (None, 10)                      650
================================================================================
Total params: 258090 (1008.16 KB)
Trainable params: 258090 (1008.16 KB)
Non-trainable params: 0 (0.00 Byte)
--------------------------------------------------------------------------------
```

1.3 1.3

The reason there are 320 parameters in the first layer is due to the filter and kernel size. We have 32 filters where the filter size or kernel_size is 3×3. Hence the number of parameters becomes $(3 \times 3 + 1) \times 32 = 10 \times 32 = 320$ where the additional 1 is the bias (and hence 1 per filter) and the rest are the kernel weights.

1.4 1.4

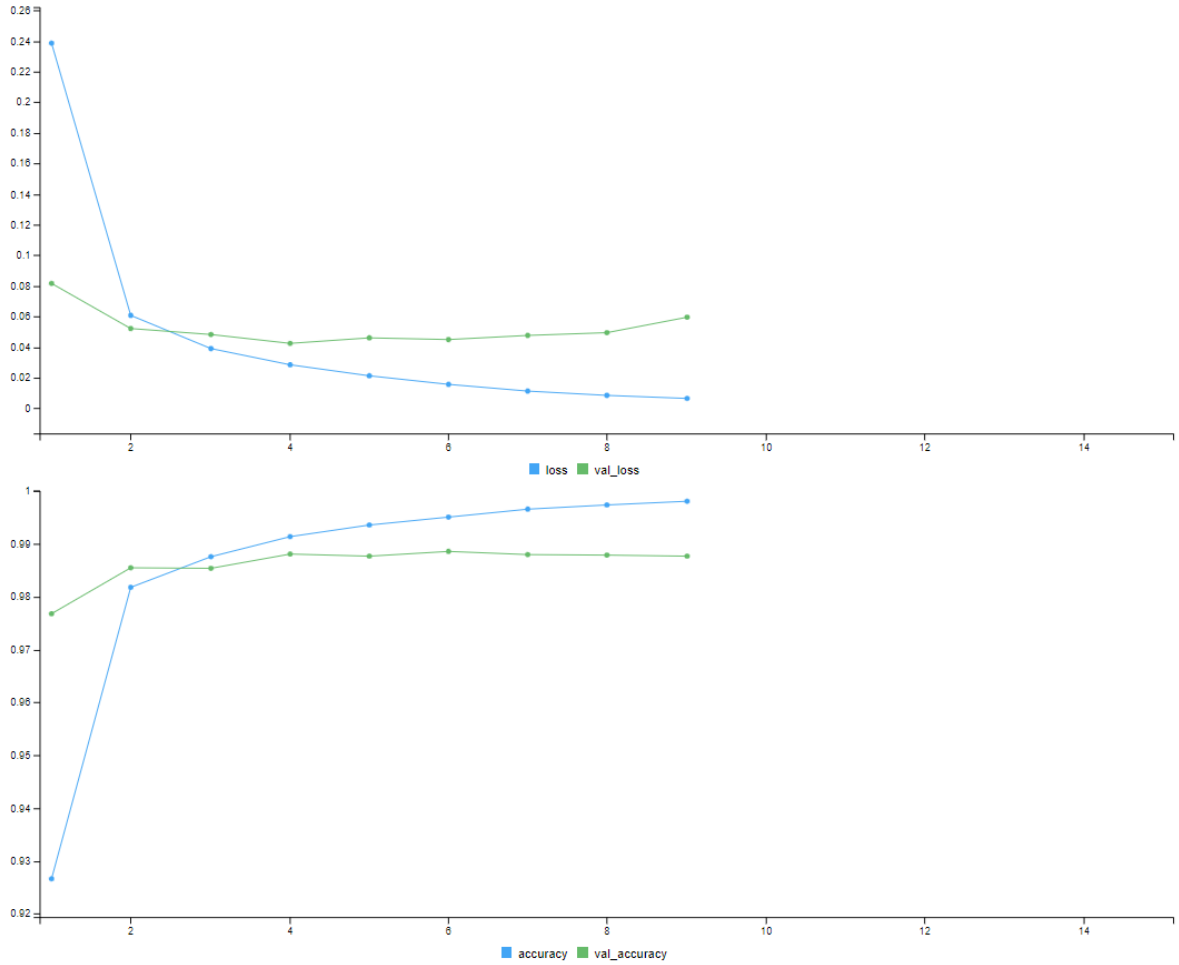
```
model1 %>% compile(
  optimizer = 'RMSprop',# RMSPROP instead of adam
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)

history <- model1 %>% fit(
  x_train, y_train,
  epochs = 10,batch_size = 128,
  validation_split = 0.2,
  callbacks = list(callback_early_stopping(patience = 5, monitor="val_loss"))
)
```

Here it stopped early. The bets results are from epoch 4. The loss and accuracy metrics are given by:

```
375/375 [==============================] - 11s 29ms/step
- loss: 0.0390 - accuracy: 0.9876 - val_loss: 0.0482 - val_accuracy: 0.9854
```

And the plot over the history of the accuracy and loss per epoch:

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



1.5 1.5

Now we test different hyper parameter values. The three hyper parameter I have chosen are the kernel size for the convolutional layers, the learning rate for the optimizer, and the Weight decay coefficient for the L_2 regularizer. The following code is structured such that using the function `build_model()` together with `call_existing_model()` samples new values for the hyper parameters and compiles a model, which is later fitted in the main `randomSearch()` function. The rest of the code is mostly to print the output and get the model attributes for each iteration.

```
  # function to return a compiled model given hyper parameters
call_existing_model <-  function(kernel_size, weight_decay, lr) {
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = kernel_size, activation = 'relu',
                kernel_regularizer = regularizer_l2(weight_decay),
                input_shape = c(28, 28, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 32, kernel_size = kernel_size, activation = 'relu',
                kernel_regularizer = regularizer_l2(weight_decay)) %>%
  layer_flatten() %>%
  layer_dense(units = 32, activation = 'relu',
              kernel_regularizer = regularizer_l2(weight_decay)) %>%
  layer_dense(units = 10, activation = 'softmax')

model %>%
 compile(
```

```
  optimizer = optimizer_rmsprop(learning_rate=lr),
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)
return(model)
}

#Function to sample hyperparameters and then build model
build_model <- function(){
  kernel_int <- sample(2:10, size=1) #sampling random integers 2-10
  kernel_size <- c(kernel_int, kernel_int)

  weight_decay <- runif(n=1, min=0.0001, max= 0.1)
    lr <- runif(n=1, min= 0.0001, max=0.01)

  model <- call_existing_model(kernel_size = kernel_size,
                               weight_decay=weight_decay,
                               lr=lr)
  return(model)
}


#function to obtain the values of the hyperparameters
#this assumes that the hyper parameters are the same for the different layers
#and thus just indexing the first layer
 hyperparameters_summary <- function(model){
   layers_configs <- lapply(model$layers, function(layer){
     layer$get_config()
   })
   parameter_vals <- list()
   kernel_int <- layers_configs[[1]]$kernel_size[[1]]
   kernel_size <- c(kernel_int, kernel_int)
   weight_decay <- layers_configs[[1]]$kernel_regularizer$config$l2

   learning_rate <- model$optimizer$get_config()$learning_rate

  parameter_vals <- list("kernel_size" = kernel_size,
                         "weight_decay" = weight_decay,
                         "learning_rate" = learning_rate )
   return(parameter_vals)
 }

 #just  a wrapper to fit a model
 fit_model <- function(model, x_train, y_train, epochs, batch_size,
                       validation_split,...){
   model %>%
   fit(x_train, y_train,
       epochs=epochs,
       batch_size=batch_size,
       validation_split=validation_split,
       ...)
 }

 #the print functions are only used withing the  RandomSearch function
 print_summary <- function(hyperparameters_summary){
   cat("Current Kernel size is ", hyperparameters_summary$kernel_size, "\n")
   cat("Current weight decay is:", hyperparameters_summary$weight_decay, "\n")
```

```
  cat("Current learning rate is", hyperparameters_summary$learning_rate, "\n")
}

#function to return the model_history results
#for simplicity sake this does not consider early stopping and just returns
# the last epoch results
get_model_results <- function(model_history){
  n_epochs <- model_history$params$epochs
  loss <- model_history$metrics$loss[n_epochs]
  accuracy <- model_history$metrics$accuracy[n_epochs]
  val_loss <- model_history$metrics$val_loss[n_epochs]
  val_accuracy <- model_history$metrics$val_accuracy[n_epochs]
  results <- list("loss"= loss, "accuracy" = accuracy,
                  "val_loss" = val_loss, "val_accuracy" = val_accuracy)
  return(results)
}


print_current_best <- function(current_best_list){
  cat("Current best model is model no.", current_best_list$index,
      " with the following hyperparameters: \n")
  cat("Hyperparameters:  \n", "Kernel_size : ",
      current_best_list$model_summary$kernel_size, "\n" )
  cat("weight decay: ", current_best_list$model_summary$weight_decay, "\n" )
  cat("learning rate:", current_best_list$model_summary$learning_rate, "\n")
  cat("Validation accuracy:", current_best_list$model_results$val_accuracy, "\n")
}

print_new_iter <- function(iter){
  line_breaks <- strrep("*", 25)
  cat(line_breaks, "\n", "New Model! \n", line_breaks, "\n" )
  cat("Current model number is:", iter, "\n")
}

#main function to implement the RandomSearch method
RandomSearch <- function(iterations, x_train, y_train, epochs,
                         batch_size, validation_split, ...){
  all_models <- list()
  current_best <- list()
  index <- 0
  for (i in 1:iterations){
    index <- index + 1
    model <- build_model()
    #print model summary
    hyper_summary <- hyperparameters_summary(model)
    print_new_iter(iter=index)
    print_summary(hyper_summary)


    model_history <- fit_model(model,  x_train, y_train, epochs=epochs,
                         batch_size=batch_size,
                         validation_split=validation_split)

    model_results <- get_model_results(model_history)

    iteration_results <- list("model" = model,
                              "model_summary" = hyper_summary,
```

```
                              "model_history" = model_history,
                              "model_results" = model_results,
                              "index" = index)
    all_models[[i]] <- iteration_results

    if(index==1) {
      current_best <- iteration_results} else{
        cur_best_acc <- current_best$model_results$val_accuracy
        iteration_accuracy <- model_results$val_accuracy
        if(iteration_accuracy > cur_best_acc){
          current_best <- iteration_results
        }
      }
    print_current_best(current_best)

  }
 all_results <- list(
    "all_models" = all_models,
    "best_model" = current_best
  )
   plot(all_results$best_model$model_history)
  return(all_results)
}
```

```
RS_results <- RandomSearch(iterations=10,
             x_train=x_train, y_train=y_train,
             epochs=10, batch_size=256,
             validation_split=0.2)

plot(RS_results$best_model$model_history)
```

Based on the results, the best model was the following:

```
RS_results$best_model$model_summary

$kernel_size
[1] 7 7

$weight_decay
[1] 0.009164357

$learning_rate
[1] 0.005879265
```

```
RS_results$best_model$model_results

$loss
[1] 0.2912685

$accuracy
[1] 0.9540833

$val_loss
[1] 0.2386363

$val_accuracy
[1] 0.9706666
```

Hence, kernel_size = (7,7), weight decay = 0.009 and learning rate =0.00059. The validation accuracy

was 0.97. This validation accuracy was lower than the one obtained in the previous task. However, a pattern was that the accuracy was generally increasing on average after each epoch. Hence, ideally it would have been better to run more epochs, but that would also not be as computationally efficient, which is why it was limited to 10 epochs in this case.

The full model summary and the results per epoch figure are:

```
RS_results$best_mode$model

 Layer (type)                       Output Shape                    Param #
================================================================================
 conv2d_11 (Conv2D)                 (None, 22, 22, 32)              1600
 max_pooling2d_5 (MaxPooling2D)     (None, 11, 11, 32)              0
 conv2d_10 (Conv2D)                 (None, 5, 5, 32)                50208
 flatten_5 (Flatten)                (None, 800)                     0
 dense_11 (Dense)                   (None, 32)                      25632
 dense_10 (Dense)                   (None, 10)                      330
================================================================================
Total params: 77770 (303.79 KB)
Trainable params: 77770 (303.79 KB)
Non-trainable params: 0 (0.00 Byte)
--------------------------------------------------------------------------------
```

```
plot(RS_results$best_model$model_history)
```

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

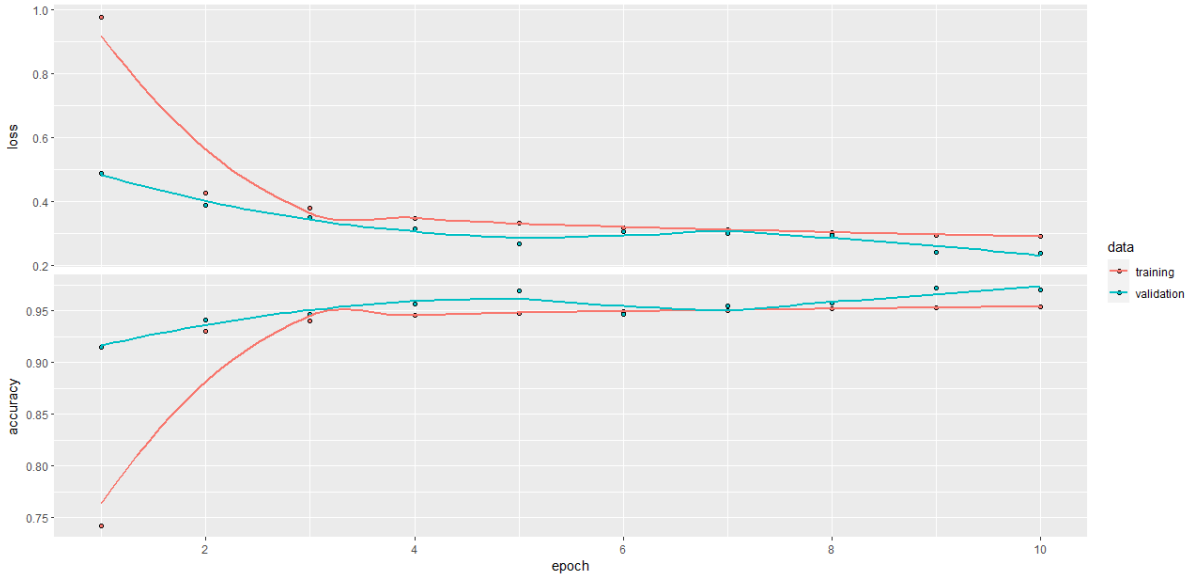


Figure 2: best model results

1.6 1.6

Now we use the cifar data set

```
cifar10 <- dataset_cifar10()

x_train <- cifar10$train$x/255
x_test <- cifar10$test$x/255
y_train <- cifar10$train$y
y_test <- cifar10$test$y
```

```
model_cifar <- keras_model_sequential() %>%
  # First hidden 2D convolutional layer
  layer_conv_2d(
    filters = 32,
    kernel_size = c(3, 3),
    padding = "valid",
    input_shape = c(32, 32, 3)
  ) %>%

  # Use max pooling
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  # Second hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), padding = "valid") %>%


  # Use max pooling once more
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  # Third hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(3, 3)) %>%


  # Flatten max filtered output into feature vector
  # and feed into dense layer
  layer_flatten() %>%

  # Fourth hidden layer
  layer_dense(units = 64) %>%


  # Outputs from dense layer are projected onto 10 unit output layer
  layer_dense(units = 10)

summary(model_cifar)
```

```
Model: "sequential_1"
_________________________________________________________________________________
 Layer (type)                        Output Shape                      Param #
=================================================================================
 conv2d_4 (Conv2D)                   (None, 30, 30, 32)                896
 max_pooling2d_2 (MaxPooling2D)      (None, 15, 15, 32)                0
 conv2d_3 (Conv2D)                   (None, 13, 13, 64)                18496
 max_pooling2d_1 (MaxPooling2D)      (None, 6, 6, 64)                  0
 conv2d_2 (Conv2D)                   (None, 4, 4, 64)                  36928
 flatten_1 (Flatten)                 (None, 1024)                      0
 dense_3 (Dense)                     (None, 64)                        65600
 dense_2 (Dense)                     (None, 10)                        650
=================================================================================
Total params: 122570 (478.79 KB)
Trainable params: 122570 (478.79 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________________________
```

running it yields:

```
opt <- optimizer_adamax(learning_rate = learning_rate_schedule_exponential_decay(
  initial_learning_rate = 5e-3,
  decay_rate = 0.96,
  decay_steps = 1500,
```

```
  staircase = TRUE
))

model_cifar %>% compile(
  loss = loss_sparse_categorical_crossentropy(from_logits = TRUE),
  optimizer = opt,
  metrics = "accuracy"
)


# Training --------------------------------------------------------------------
model_cifar %>% fit(
  x_train, y_train,
  batch_size = 128,
  epochs = 10,
  validation_data = list(x_test, y_test),
  shuffle = TRUE
)
```

```
Epoch 10/10
391/391 [==============================] - 23s 60ms/step
- loss: 0.9227 - accuracy: 0.6837 - val_loss: 1.0588 - val_accuracy: 0.6481
```

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

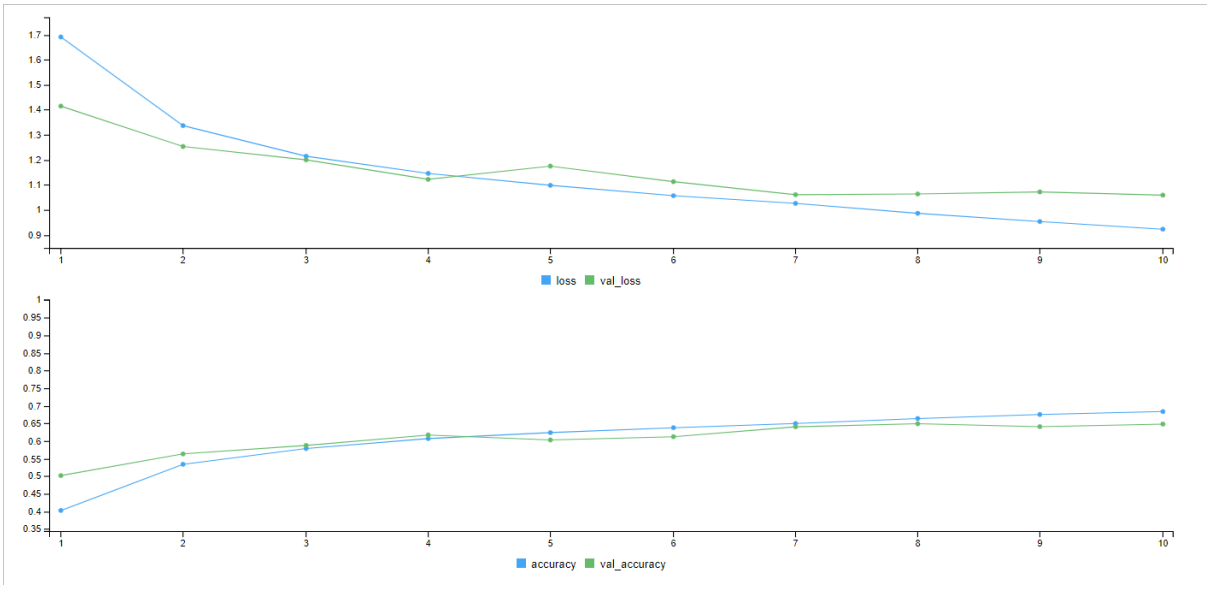


Figure 3: cifar model results

Looking at the results, we see that the validation accuracy is rather low compared to before with the mnist data since the accuracy is around 0.65. However, The accuracy was increasing for each epoch and we could probably have gotten it higher if we just ran it for more epochs.

1.7 1.7

Now we visualize one image

```
library(EBImage)
class_labels <- c("Airplane", "Car", "Bird", "Cat", "Deer", "Dog",
                  "Frog", "Horse", "Ship", "Truck")
y_factor <- factor(y_train, levels = 0:9, labels = class_labels)
```

```
plot_images_2 <- function(idx, x_train, y_train){
  par(mfrow=c(1,2), mar = c(4, 2, 2, 2))

  fig_img <- list()
  for (i in 1:length(idx)){
    fig_mat <- x_train[idx[i],,,]
    fig_img[[i]] <- normalize(Image(transpose(fig_mat), dim=c(32,32,3), colormode='Color'))
    label <- y_factor[idx[i]]
    plot(fig_img[[i]])
    title(main=label)
  }
  par(mfrow=c(1,1))
}
```

```
plot_images_2(idx = c(13, 37), x_train, y_train)
```

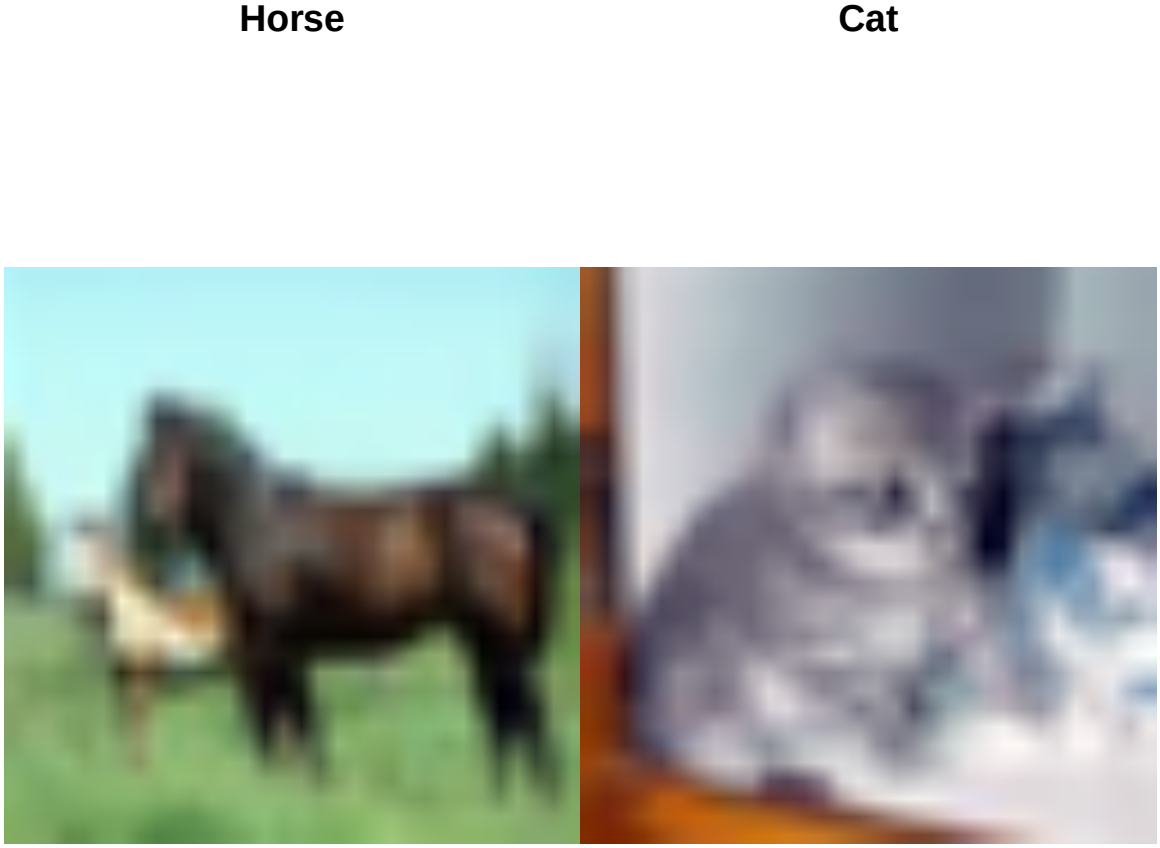


Figure 4: Images from the cifar data set

1.8 task 1.8

The reason we know have 896 parameters is because we know have the input channels the rgb for color, which is equal to 3. Hence, whereas we previously had $(3 \times 3 + 1) \times 32 = 10 \times 320 = 320$ we know have $(3 \times 3 \times 3 + 1) \times 32 = 28 \times 32 = 896$

1.9 task 1.9

First we start of by adding more hidden layers, both convolutional and dense layers, and increasing the number of hidden unis, which increases the representational capacity of the model. This is so we can potentially overfit. However, due to computational reasons, I will not go all out but simply try adding a relatively small change. We also increase the epochs to 15 to give it more time to converge.

```
model_cifar2 <- keras_model_sequential() %>%
  # First hidden 2D convolutional layer
  layer_conv_2d(
    filters = 32,
    kernel_size = c(3, 3),
    padding = "valid",
    input_shape = c(32, 32, 3)
  ) %>%

  # Use max pooling
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  # Second hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), padding = "valid") %>%

  # Use max pooling once more
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  # second convulutional hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), padding="valid") %>%

  #third convulutional hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), padding="valid") %>%

  # Flatten max filtered output into feature vector
  # and feed into dense layer
  layer_flatten() %>%

  # Fourth hidden layer
  layer_dense(units = 256, activation = "relu") %>%

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

  # 10 unit output layer
  layer_dense(units = 10)

summary(model_cifar2)
```

```
Model: "sequential_2"
_________________________________________________________________________________
 Layer (type)                        Output Shape                      Param #
=================================================================================
 conv2d_8 (Conv2D)                   (None, 30, 30, 32)                896
 max_pooling2d_4 (MaxPooling2D)      (None, 15, 15, 32)                0
 conv2d_7 (Conv2D)                   (None, 13, 13, 64)                18496
 max_pooling2d_3 (MaxPooling2D)      (None, 6, 6, 64)                  0
 conv2d_6 (Conv2D)                   (None, 4, 4, 64)                  36928
 conv2d_5 (Conv2D)                   (None, 2, 2, 64)                  36928
```

```
 flatten_2 (Flatten)                 (None, 256)                       0
 dense_6 (Dense)                     (None, 256)                       65792
 dense_5 (Dense)                     (None, 128)                       32896
 dense_4 (Dense)                     (None, 10)                        1290
================================================================================
Total params: 193226 (754.79 KB)
Trainable params: 193226 (754.79 KB)
Non-trainable params: 0 (0.00 Byte)
--------------------------------------------------------------------------------
```

```
model_cifar2 %>% compile(
  loss = loss_sparse_categorical_crossentropy(from_logits = TRUE),
  optimizer = opt,
  metrics = "accuracy"
)


# Training --------------------------------------------------------------------
model_cifar2 %>% fit(
  x_train, y_train,
  batch_size = 128,
  epochs = 15,
  validation_data = list(x_test, y_test),
  shuffle = TRUE,
  callbacks = list(callback_early_stopping(patience = 5, monitor="val_loss"))
)
```

The results are:

```
Epoch 8/15
391/391 [==============================] - 25s 64ms/step
- loss: 0.6989 - accuracy: 0.7521 - val_loss: 0.8948 - val_accuracy: 0.7108
```

```
knitr::include_graphics("cifar_model_2.png")
```

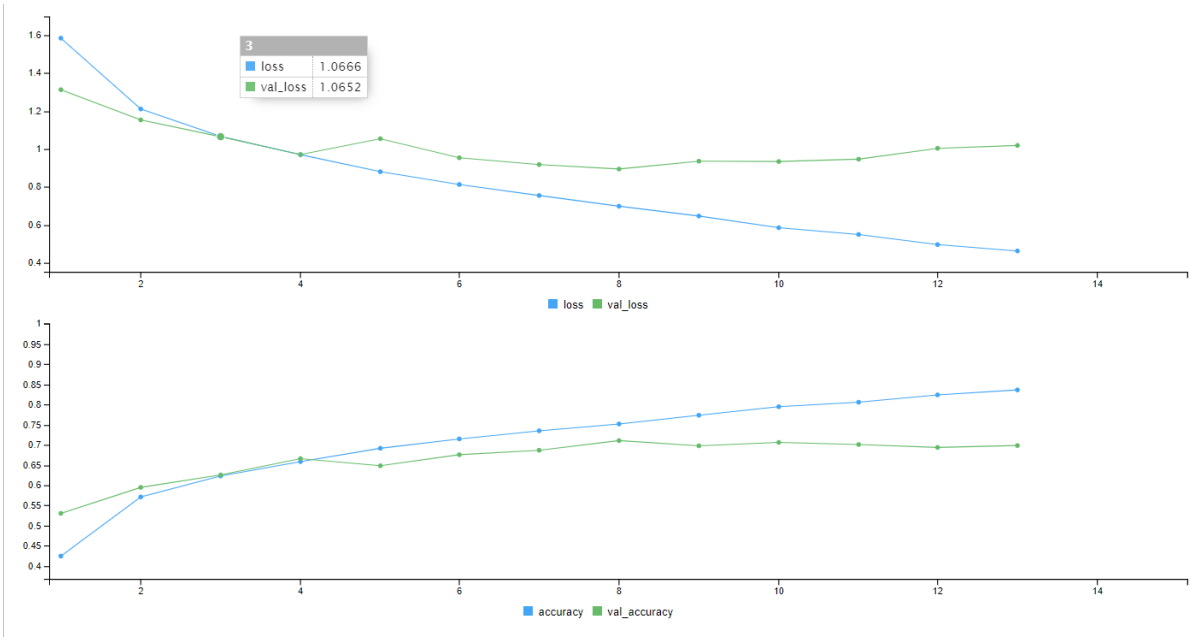


Figure 5: Results for second cifar model

Looking at the results, it ended at epoch 13 due to the early stopping. The best results were from epoch

8:

The accuracy improved slightly to 0.71

For the next model, we increase the batch size, since a batch with more samples results in more informative gradients with lower variance, while also being more computationally efficient. We also add implicit zero padding to keep the representation size large while also increasing the convolution kernel width to increase the number of parameters in the model . Furthermore, as inspired by the linked guide on the tensorflow webpage for this data set. we can add `layer_activation_leaky_relu(0.1)` which will allow a small, non-zero gradient when the input is negative and is supposed to helps prevent neurons becoming inactive and stop learning while training. We also increase the number of epochs little bit again and I will tune the learning rate manually as to lower it after 10 epochs, since an improper learning rate will result in a model with low effective capacity.

```
model_cifar3 <- keras_model_sequential() %>%
  # First hidden 2D convolutional layer
  layer_conv_2d(
    filters = 32,
    kernel_size = c(3, 3),
    padding = "valid",
    input_shape = c(32, 32, 3)
  ) %>%

  # Use max pooling
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  # Second hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), padding = "same") %>%
  layer_activation_leaky_relu(0.1) %>%
  # Use max pooling once more
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  # second convulutional hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), padding="same") %>%
  layer_activation_leaky_relu(0.1) %>%

  #third convulutional hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), padding="same") %>%
  layer_activation_leaky_relu(0.1) %>%

  # Flatten max filtered output into feature vector
  # and feed into dense layer
  layer_flatten() %>%

  # Fourth hidden layer
  layer_dense(units = 256, activation = "relu") %>%

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

  # 10 unit output layer
  layer_dense(units = 10)

print(model_cifar3)
```

```
Model: "sequential_3"
_________________________________________________________________________________
 Layer (type)                        Output Shape                      Param #
=================================================================================
 conv2d_12 (Conv2D)                  (None, 30, 30, 32)                896
 max_pooling2d_6 (MaxPooling2D)      (None, 15, 15, 32)                0
 conv2d_11 (Conv2D)                  (None, 15, 15, 64)                51264
```

```
 leaky_re_lu_2 (LeakyReLU)           (None, 15, 15, 64)              0
 max_pooling2d_5 (MaxPooling2D)      (None, 7, 7, 64)                0
 conv2d_10 (Conv2D)                  (None, 7, 7, 64)                102464
 leaky_re_lu_1 (LeakyReLU)           (None, 7, 7, 64)                0
 conv2d_9 (Conv2D)                   (None, 7, 7, 64)                102464
 leaky_re_lu (LeakyReLU)             (None, 7, 7, 64)                0
 flatten_3 (Flatten)                 (None, 3136)                    0
 dense_9 (Dense)                     (None, 256)                     803072
 dense_8 (Dense)                     (None, 128)                     32896
 dense_7 (Dense)                     (None, 10)                      1290
================================================================================
Total params: 1094346 (4.17 MB)
Trainable params: 1094346 (4.17 MB)
Non-trainable params: 0 (0.00 Byte)
--------------------------------------------------------------------------------
```

```
model_cifar3 %>% compile(
  optimizer = 'RMSprop',
  loss = loss_sparse_categorical_crossentropy(from_logits = TRUE),
  metrics = c('accuracy')
)


Scheduler <- function(epoch, lr) {
  if (epoch < 10) {
    return(lr)
  } else {
    return(lr * exp(-0.1))
  }
}

callback_list = list(callback_early_stopping(patience = 5),
                     callback_learning_rate_scheduler(Scheduler))

history_cifar_3 <- model_cifar3 %>%
  fit(
  x_train, y_train,
  epochs = 20 ,batch_size = 512,
  validation_data = list(x_test, y_test),
    shuffle = TRUE,
  callbacks =callback_list)
```

And the results are:

```
98/98 [==============================] - 61s 620ms/step
- loss: 0.6635 - accuracy: 0.7688 - val_loss: 0.8182 - val_accuracy: 0.7230
- lr: 0.0010
```

```
knitr::include_graphics("cifar_model_3.png")
```

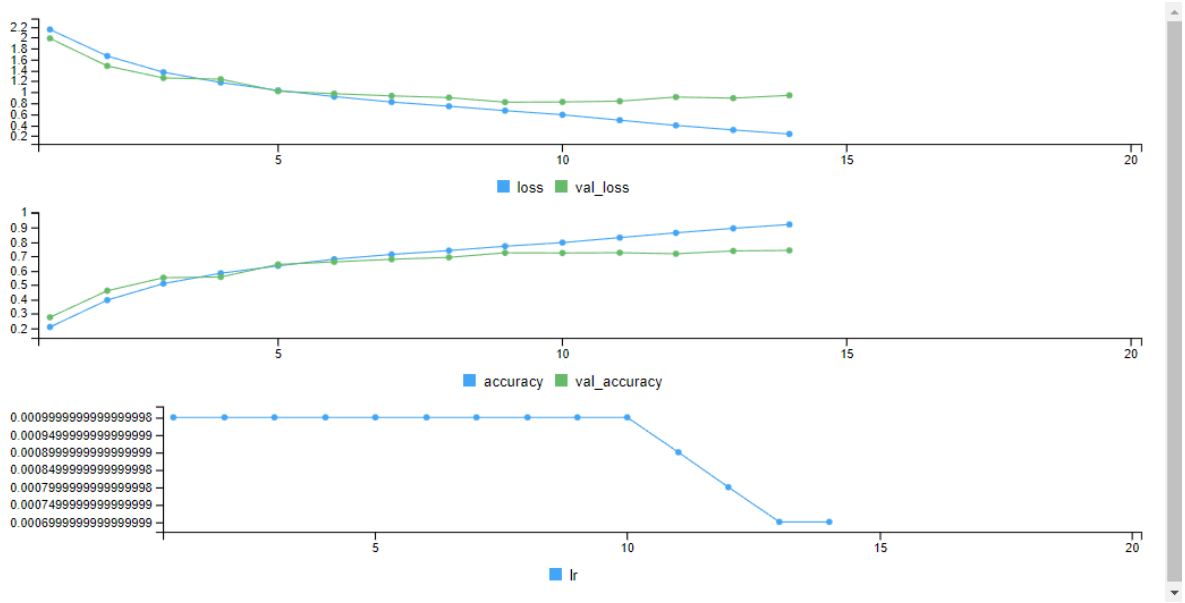


Figure 6: Results for the third model

This also stopped early. The validation accuracy was 0.72 and hence a small improvement, although very marginally.

Next, we can implement some regularization techniques. This will negatively affect the model fit on the training data, but with the aim of making if fit better for validation. More specifically, we will implement the L2 regularization which adds to the a cost to the models loss function of having large weights. Here we simply use the default value of 0.01 since I do not want to test multiple values due to computational reasons. Furthermore, we will also apply dropout to the layers which works by randomly dropping out a number of the output features of the layers during the training. lastly we can add normalization for the dense layer by normalizing the input of the layer, which can improve the generalization.

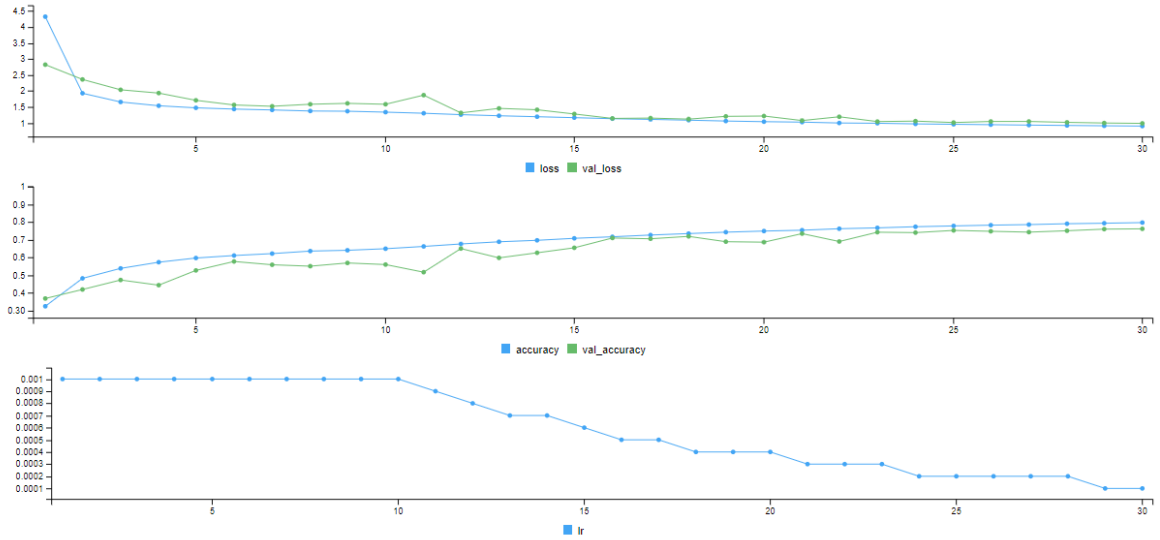
```
model_cifar4 <- keras_model_sequential() %>%
  # First hidden 2D convolutional layer
  layer_conv_2d(
    filters = 32,
    kernel_size = c(3, 3),
    padding = "valid",
    input_shape = c(32, 32, 3),
     kernel_regularizer = regularizer_l2()) %>%

  # Use max pooling
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
   layer_dropout(0.25) %>%
  # Second hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), padding = "same",
                kernel_regularizer = regularizer_l2()) %>%
  layer_activation_leaky_relu(0.1) %>%
  # Use max pooling once more
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
   layer_dropout(0.25) %>%

  # second convulutional hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), padding="same",
                kernel_regularizer = regularizer_l2()) %>%
  layer_activation_leaky_relu(0.1) %>%
```

```
  #third convulutional hidden layer
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), padding="same",
                kernel_regularizer = regularizer_l2()) %>%
  layer_activation_leaky_relu(0.1) %>%

  # Flatten max filtered output into feature vector
  # and feed into dense layer
  layer_flatten() %>%

  # Fourth hidden layer
  layer_dense(units = 256, activation = "relu",
              kernel_regularizer = regularizer_l2()) %>%
   layer_dropout(0.25) %>%
   layer_batch_normalization() %>%

  layer_dense(units=128, activation="relu",
              kernel_regularizer = regularizer_l2()) %>%
    layer_dropout(0.5) %>%

  # 10 unit output layer
  layer_dense(units = 10)
summary(model_cifar4)
```

```
Model: "sequential_4"
_________________________________________________________________________________
 Layer (type)                    Output Shape                Param #    Trainable
=================================================================================
 conv2d_16 (Conv2D)              (None, 30, 30, 32)          896        Y
 max_pooling2d_8 (MaxPooling2    (None, 15, 15, 32)          0          Y
 D)
 dropout_3 (Dropout)             (None, 15, 15, 32)          0          Y
 conv2d_15 (Conv2D)              (None, 15, 15, 64)          51264      Y
 leaky_re_lu_5 (LeakyReLU)       (None, 15, 15, 64)          0          Y
 max_pooling2d_7 (MaxPooling2    (None, 7, 7, 64)            0          Y
 D)
 dropout_2 (Dropout)             (None, 7, 7, 64)            0          Y
 conv2d_14 (Conv2D)              (None, 7, 7, 64)            102464     Y
 leaky_re_lu_4 (LeakyReLU)       (None, 7, 7, 64)            0          Y
 conv2d_13 (Conv2D)              (None, 7, 7, 64)            102464     Y
 leaky_re_lu_3 (LeakyReLU)       (None, 7, 7, 64)            0          Y
 flatten_4 (Flatten)             (None, 3136)                0          Y
 dense_12 (Dense)                (None, 256)                 803072     Y
 dropout_1 (Dropout)             (None, 256)                 0          Y
 batch_normalization (BatchNo    (None, 256)                 1024       Y
 rmalization)
 dense_11 (Dense)                (None, 128)                 32896      Y
 dropout (Dropout)               (None, 128)                 0          Y
 dense_10 (Dense)                (None, 10)                  1290       Y
=================================================================================
Total params: 1095370 (4.18 MB)
Trainable params: 1094858 (4.18 MB)
Non-trainable params: 512 (2.00 KB)
_________________________________________________________________________________
```

The results are:

```
Epoch 30/30
98/98 [==============================] - 73s 748ms/step
x- loss: 0.9004 - accuracy: 0.7981 - val_loss: 0.9849 - val_accuracy: 0.7628 - lr: 1.3534e-04
```

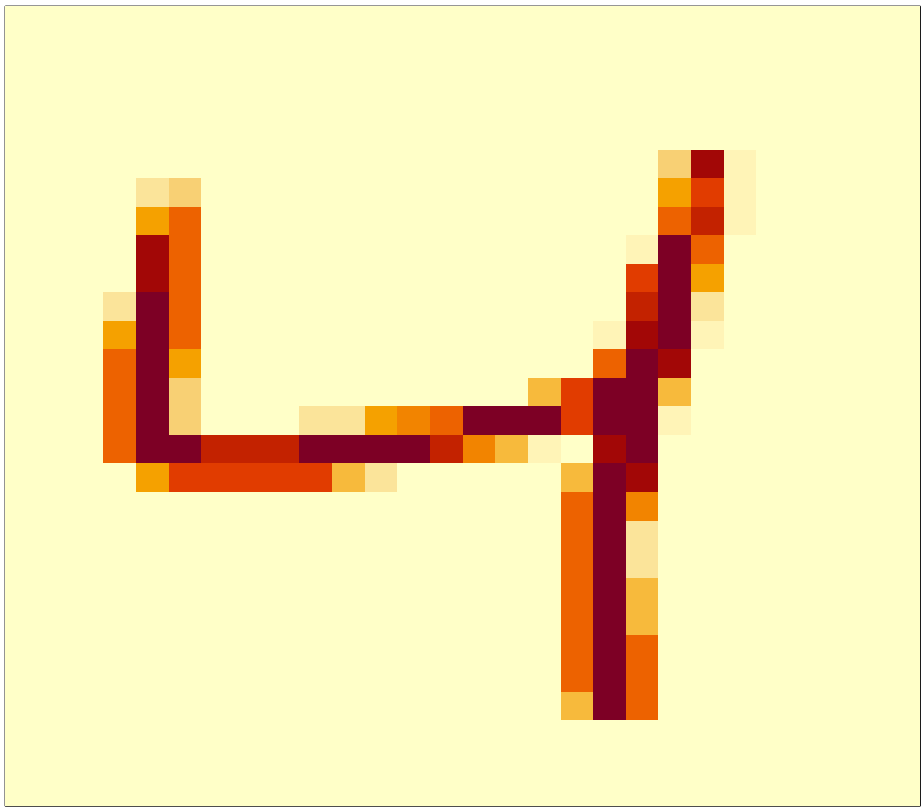


This did not stop early and the validation accuracy improved to 0.7628.

2 Task 2

2.1 2.1

```
library(uuml)
data("mnist_example")
im <- mnist_example[["4"]]

image_func <- function(img, digit=4){
  image(1: ncol(img), 1: nrow(img), img, xlab = "", ylab = "",
      xaxt='n' , yaxt='n' , main=paste0("digit = ", digit))
}
image_func(im)
```



2.2 2.2

```
X <- mnist_example[["4"]][12: 15, 12: 15]
K <- matrix(c(-0.5, -0.5, -0.5, 1, 1, 1, -0.5, -0.5, -0.5) , nrow = 3, byrow = TRUE)
K
```

```
     [,1] [,2] [,3]
[1,] -0.5 -0.5 -0.5
[2,]  1.0  1.0  1.0
[3,] -0.5 -0.5 -0.5
```

```
X
```

```
     [,1] [,2] [,3] [,4]
[1,]   56  250  116    0
[2,]    0  240  144    0
[3,]    0  198  150    0
[4,]    0  143  241    0
```

```
kernel_func <- function(X_block, K){
  res <- sum(X_block*K)
  return(res)
}
```

```
convolution <- function(X, K) {
  k_dim <- nrow(K)
  n_xrows <- nrow(X)
  n_xcols <- ncol(X)
  steps_down <- n_xrows - k_dim + 1
  steps_right <- n_xcols - k_dim + 1

  result <- matrix(0, nrow = steps_down, ncol = steps_right)

  for (i in 1:steps_down) {
       row_subset <- i:(i + k_dim - 1)
    for (j in 1:steps_right) {
      col_subset <- j:(j + k_dim - 1)
      X_block <- X[row_subset, col_subset]
      result[i, j] <- kernel_func(X_block, K)
    }
  }

  return(result)
}

convolution(X,K)
```

```
     [,1] [,2]
[1,]   -1   27
[2,]  -36  -36
```

2.3 2.3

Applying to function to the MNIST example 4 digit yields:

```
conv_res <- convolution(mnist_example[["4"]], K)
image_func(conv_res)
```

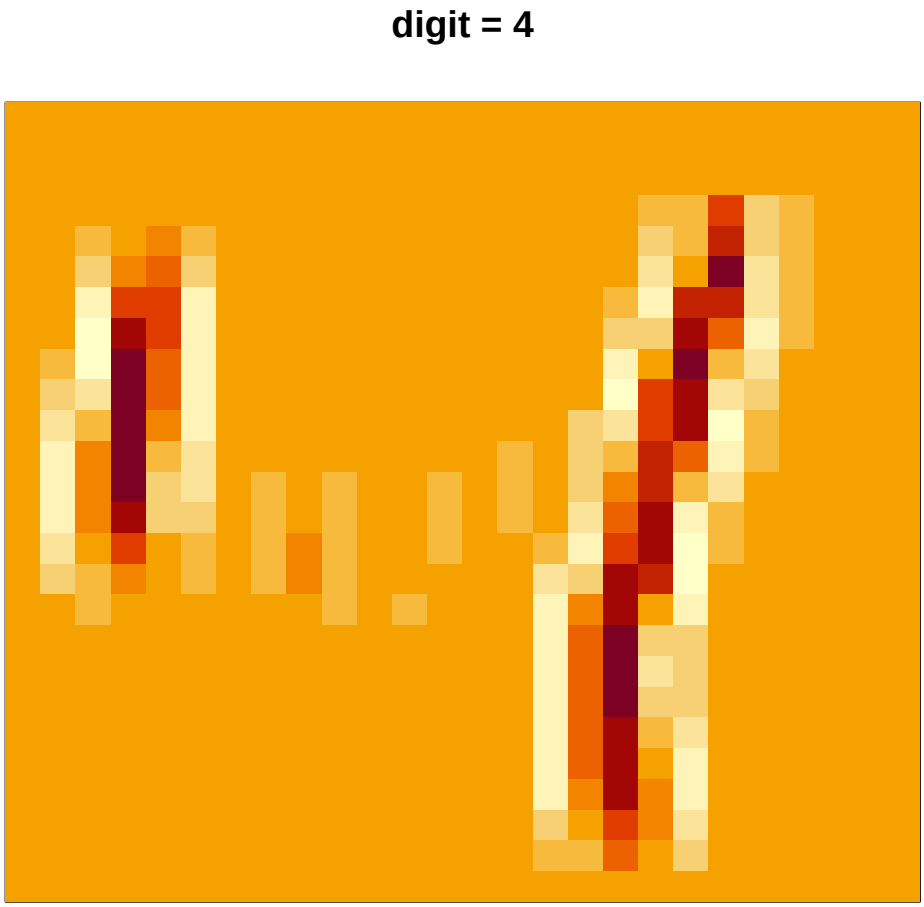


Figure 7: Results from convulution function on mnist example

2.4 2.4

Now we add more functionality for bias as well as an activation function

```
relu <- function(x) max(0, x)

convolutional_layer <- function(X, K, b, activation){
  conv_result <- convolution(X,K)
  conv_bias <- conv_result + b

  output <-apply(conv_bias, MARGIN = c(1, 2), FUN = activation)


  return(output)

  }
convolutional_layer(X, K, 100, relu)
```

```
     [,1] [,2]
```

```
[1,]   99  127
[2,]   64   64
```

2.5 2.5

Now we apply the function on digit 4 and visualize the result.

```
conv_layer_res <- convolutional_layer(X= mnist_example[["4"]], K, b= -150, relu)
```

```
image_func(conv_layer_res)
```

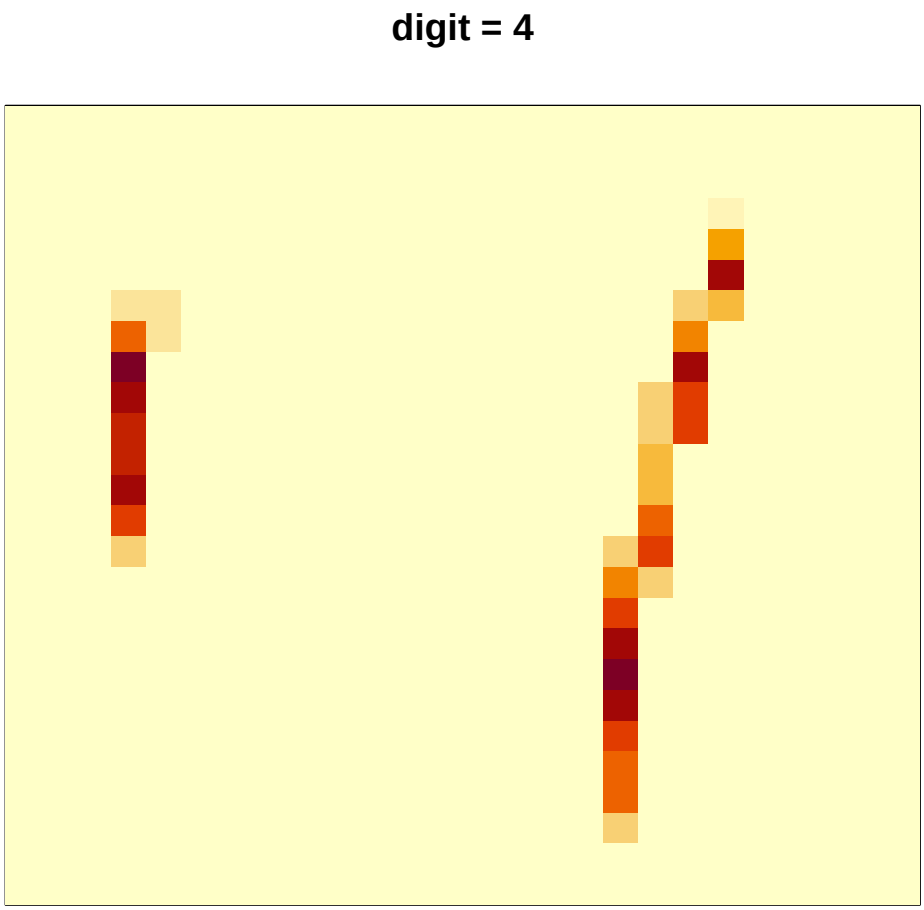


Figure 8: Visualization of the convolutional layer result

This seems to visualize the vertical lines in the digit 4.

2.6 2.6

Next we tranpose the filter and run the convolutional layer yet again with bias = -150

```
conv_layer_res_t <- convolutional_layer(X= mnist_example[["4"]],
                                        t(K), b= -150, relu)
```

```
image_func(conv_layer_res_t)
```

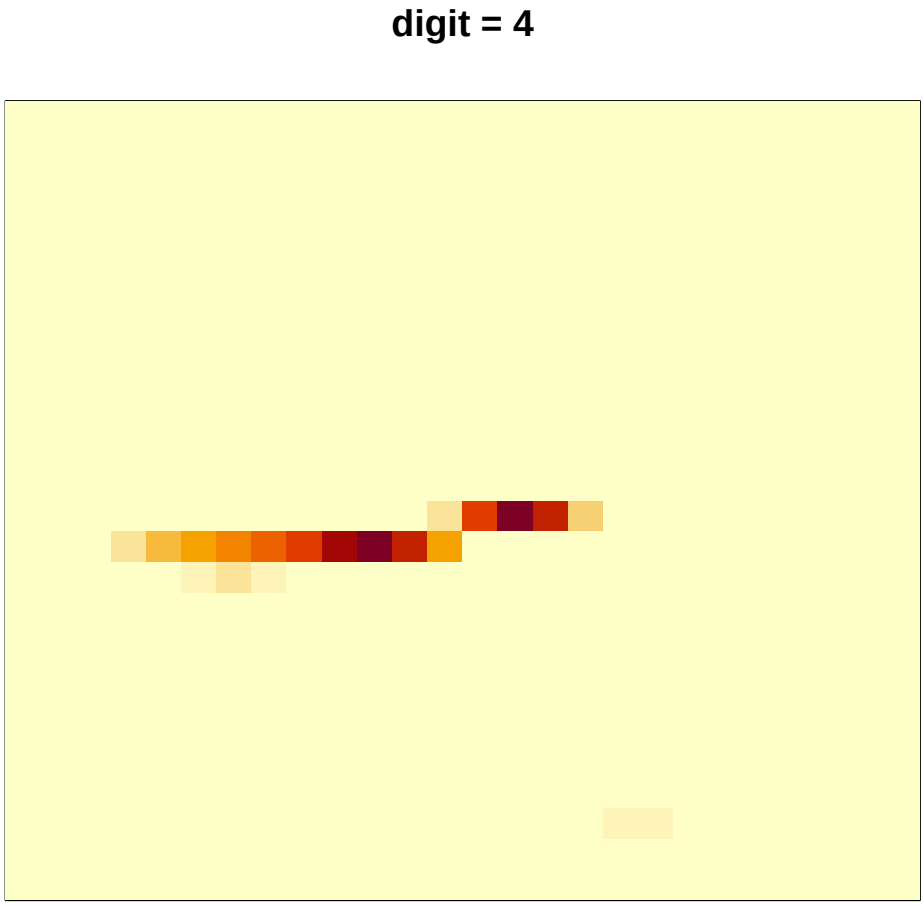


Figure 9: Visualization of convolutional layer results with tranposed kernel

Looking at the image, this transposed filter seems to illustrate the horizontal line in the digit 4.

2.7 2.7

Now we implement a two by two, two stride max-pooling layer.

```
X <- mnist_example[["4"]][12: 15, 12: 15]
X
```

```
     [,1] [,2] [,3] [,4]
[1,]   56  250  116    0
[2,]    0  240  144    0
[3,]    0  198  150    0
[4,]    0  143  241    0
```

```
maxpool_layer <- function(X, size=2){
  n_xcols <- ncol(X)
```

```
  n_xrows <- nrow(X)

  steps_down <- n_xrows %/% size
  steps_right <- n_xcols  %/% size

  #indeces where to start each row and col subset
  row_start <- seq(from=1, to=n_xrows, by=size)
  col_start <- seq(from=1, to=n_xcols, by=size)

    results <- matrix(NA, nrow=steps_down, ncol=steps_right)
    for(i in 1:steps_down){

      r_subset <- row_start[i]:(row_start[i]+size-1)

    for(j in 1:steps_right){
      c_subset <- col_start[j]:(col_start[j]+size-1)
      results[i,j] <- max(X[r_subset, c_subset])
      }
    }
  return(results)
}

maxpool_layer(X)
```

```
     [,1] [,2]
[1,]  250  144
[2,]  198  241
```

2.8 2.8

Now to put it all together and visualize the final output.

```
X <- mnist_example[["4"]]

relu <- function(x) max(0, x)
output <- maxpool_layer(convolutional_layer(X, K, b=-320, relu))
```

```
image_func(output)
```

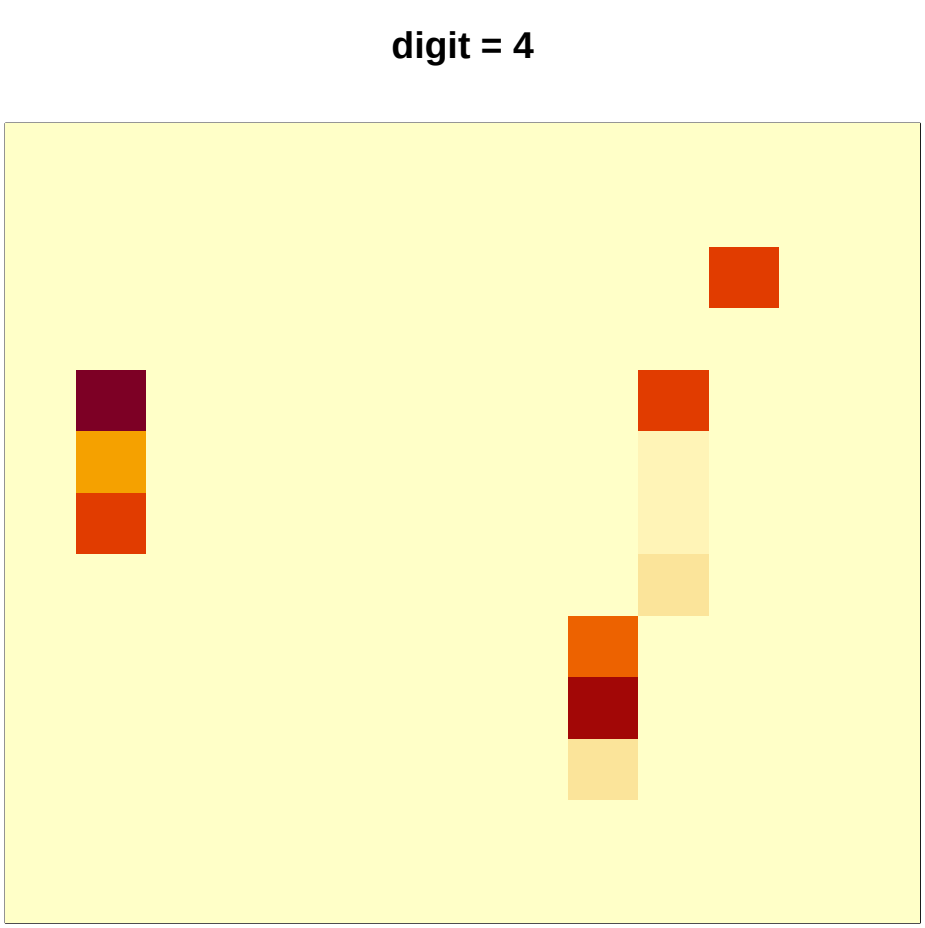


Figure 10: Visualization of final result

Looking at the results, then it seems like this is capturing the rough outlines of the vertical bars in the digit 4.