Post Block Assessment 1

Francois van Zyl : 18620426

06/07/2020

General Deep Learning Concepts [24]

Question 1) Considering this description, when does a neural network model become a deep learning model? [2]

A. When you add more hidden layers and increase depth of neural network

Question 2) Consider a problem where we want to classify a given picture as either a cat or a dog, which of the following architecture would you choose? [2]

C. Any one of these

Question 3) Consider a neural network where the number of nodes in the input layer is 10 and the hidden layer is 10. What is the maximum number of connections from the input layer to the hidden layer? [2]

A. 100

Question 4) Consider a simple multi-layer perceptron (MLP) model with the following architecture. There are 6 neurons in the input layer, 10 neurons in the hidden layer and a single neuron in the output layer. From the options below, what is the size of the weight matrices between the hidden layer and the output layer, as well as the input layer and the hidden layer? [2]

D. [10 x 1], [6 X 10]

Question 5) There are multiple steps in the training process of a neural network. What is the correct sequence of the following training tasks in a network? [2] 1. Initialize weights of neuron randomly 2. Go to the next batch of dataset 3. If the prediction does not closely approximate the output, change the weights 4. For a sample input, compute an output

D. 1, 4, 3, 2

Question 6) Consider a representation where we implement an AND function to a single neuron. Below is a tabular representation of an AND function: What would the weights and bias be for the neuron? [4]

A. Bias = -1.5, w1 = 1, w2 = 1

Question 7) When we stack neurons together, we form a neural network. Consider the example of a neural network simulating an XNOR function as indicated below. Suppose X1 is 0 and X2 is 1, what will the output be for the given neural network? [4]

Since the final class output prediction is f < 0, the answer is A. 0

Question 8) Model capacity refers to the ability of a neural network to approximate complex functions. Which of the following is statements true about model capacity? [2]

A. As number of hidden layers increase, model capacity increases

Question 9) Will the classification error of test data always decrease if you increase the number of hidden layers in a MLP model. [2]

B. No

Question 10) You want to map every possible image of size 64×64 to a binary category. Suppose an image has 3 channels and each pixel in each channel can take an integer value between (and including) 0 and 255. How many bits do you need to represent this mapping? [2]

(i) $[256^3]^{64*64}$

Vectorization [24]

Question 1) 1. Describe what vectorization in programming is? [3]

The vectorization of a piece of code refers to the setup and utilization of code that enables the utilization of the architecture of a vector to compute multiple data points in parallel instead of treating the vector as scalars that need to be looped through. The latter approach is called non-vectorized code, where, as an example, a for-loop is used to iterate through each entry within the vector. Since vectorized code exploits the vector architecture to perform multiple operations in parallel, non-vectorized code takes longer to execute. Therefore vectorized code is essential to deep learning, since deep learning applications are concerned with potentially millions of data points.

Question 2) Consider the two vectors a = [0,1,2] and b = [3,4,5]. Using: 1) a for loop, and 2) vectorization, add the two vectors to form a new vector c. Illustrate how you use vectorization versus using the for loop to add the two vectors by code and by visual means as part of your answer. Any programming language can be used. [6]

```
a <-  c(0,1,2)
b <- c(3,4,5)
start_time <- Sys.time()
c <- a + b #Vectorized approach
end_time <- Sys.time()
print(c)
```

```
## [1] 3 5 7
```

```
end_time - start_time
```

```
## Time difference of 0.001999855 secs
```

```
c <- NULL
start_time <- Sys.time()
for(i in 1:length(a))
{
  c[i] <- a[i] + b[i] #Non-vectorized approach
  print(c)
}
```

```
## [1] 3
## [1] 3 5
## [1] 3 5 7
```

```
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 0.004992008 secs
```

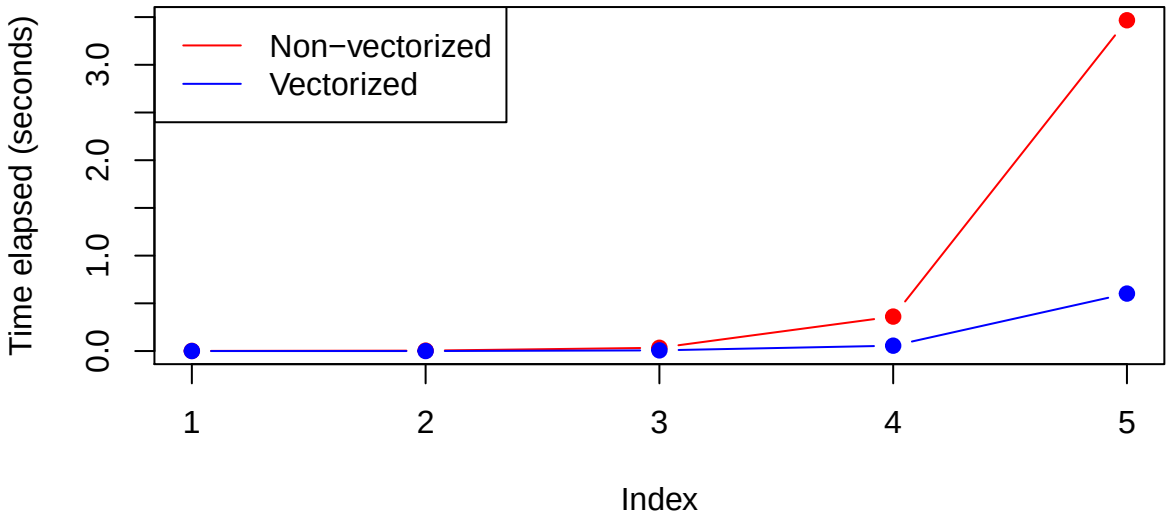
The non-vectorized code consistently takes longer to execute than the vectorized code. This difference in timing can be visually quantified by inspecting the resulting vector c during the non-vectorized implementation. The vector can be seen to operate on a single data point before moving to the next one as seen in the printout. This is in contrast to the vectorized approach, where the operations are performed in parallel by on all the data points and the result is immediate as seen in the printout

Question 3) Why is vectorization important in deep learning? As part of your answer, discuss how vectorization is used in the domain of natural language processing (NLP) within the field of deep learning. [5]

As seen in the previous question, the non-vectorized approached resulted in an execution time that was multiples larger than the vectorized approach. Note that this problem was simply the addition of two vectors with a length of 3. Deep learning systems utilize extremely large data sets with potentially millions of data points, which will ensure that non-vectorized approaches take significantly longer since each of these data points will have to be processed individually, instead of one parallel and direct vector operation. Furthermore, deep learning typically performs complex matrix multiplications, which have a longer execution time than simple vector addition. Therefore, the time difference between vectorized and non-vectorized approaches will be exacerbated further in the context of deep learning. Vectorization provides the field of NLP a way of utilizing the words and the relationships they share. Words are turned to vectors, after which algorithms can be applied to these vectors. For instance, one of the most common methods used to vectorize text is TF-IDF vectorization, which transforms text documents to vectors of numbers. This vectorization accounts for the frequency of a word within a single document, as well as assessing the frequency of this word within all the text documents. The combination of these two frequencies gives insight into how common and important a word is throughout all the text documents. Note that this one of the simpler methods of vectorization, more sophisticated methods exist such as word2vec. But the essence of the application remains the same, it converts words to vectors such that they can be utilized by models.

Question 4) In this question, we will investigate the speed-up of vectorization versus for-loops when generating probabilities for a Gaussian function. Using the above equation, create a list of n probabilities where n = [1000,10000,100000,1000000,10000000] and where x is sampled from a uniform distribution between 10 and 20, i.e., x = U(10,20) . Any values may be used for mean and standard deviation as long as they are consistent across the two methods (for loops versus vectorization). Plot the time it takes to create the list of n probabilities for the various values of n for the two different approaches and comment on the difference in time as the magnitude of n increases. [10]

```
mu = 15
sigma = 3
N <- c(10^3, 10^4, 10^5, 10^6, 10^7)
diff.time.vec <- diff.time.nonvec <- vector(length = 5)
for(i in 1:length(N)){
  x <- runif(N[i], 10, 20)

  #Vec
  start_time <- Sys.time()
  vec <- dnorm(x, mean = mu, sd  = sigma)
  end_time <- Sys.time()
  diff.time.vec[i] <- end_time - start_time

  #Non-Vec
  g <- vector(length = N[i])
  start_time <- Sys.time()
  for(j in 1:length(x))  g[j] = (1/(sigma*sqrt(2*pi)))*exp(-0.5*((x[j] - mu)/sigma)^2)
  end_time <- Sys.time()
  diff.time.nonvec[i] <- end_time - start_time
}
plot(diff.time.nonvec, type = "b", pch = 19, col = "red", ylab = "Time elapsed (seconds)")
lines(diff.time.vec, type = "b", pch = 19, col = "blue")
legend("topleft", legend = c("Non-vectorized", "Vectorized"), lty = 1, col = c("red", "blue"))
```



Note that each index displayed corresponds to the increasing values of N as stated in the problem description. At low values of N *i.e. [1000,10000]*, the difference is marginal but as soon as the value of N increases, *i.e. the number of data points and operations increase*, the non-vectorized approach's elapsed time drastically increases. Between 1 000000 and 10 000000 data points, the difference in elapsed time can be seen to increase the fastest where there is a massive spike in the difference between the elapsed time. This trend will continue as the amount of data points and operations increase. This further ties in to why vectorized approaches are preferred in deep learning, since it is clear that non-vectorized approaches take much longer at performing exactly the same operation for exactly the same result.

```
knitr::kable(data.frame(Vectorized = head(vec, 2), NonVectorized = head(g, 2)), "markdown")
```

Vectorized	NonVectorized
0.0993691	0.0993691
0.0528075	0.0528075

Vanishing gradient problem and activation functions [49]

In this question we will explore the vanishing gradient problem encountered in a binary classification problem. Using the train and test data sets, implement the following:

Question 1) Load the data sets into your environment (trainX, trainy, testX, testy)

```
rm(list = ls())
setwd(dir = "C:/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA1/")
library(keras)
library(tensorflow)
library(ggplot2)
x_train <- as.matrix(read.csv("trainX.csv", header = F))
y_train <- as.matrix(read.csv("trainY.csv", header = F))
x_test <- as.matrix(read.csv("testX.csv", header = F))
y_test <- as.matrix(read.csv("testY.csv", header = F))
use_python("C:/Users/jd-vz/anaconda3")
use_condaenv("r-tensorflow")
use_condaenv("r-reticulate")
```

Question 2) Develop an MLP model with the following specifications: [10]

```
model.mlp <- keras_model_sequential() %>%    #Initialize model architecture
  layer_dense(units = 5, activation = "tanh", input_shape = c(2),
              kernel_initializer =
                initializer_random_uniform(minval = 0,
                                           maxval = 1,seed = 0)) %>%
  layer_dense(units = 1, activation = "sigmoid",
              kernel_initializer =
                initializer_random_uniform(minval = 0,
                                           maxval = 1, seed = 0))
model.mlp %>% compile(optimizer = optimizer_sgd(lr = 0.01, momentum = 0.9),
                      loss = "binary_crossentropy", metrics = "accuracy")
model.mlp
```

```
## Model
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                         Output Shape                      Param #
## =========================================================================
## dense (Dense)                        (None, 5)                         15
## _________________________________________________________________________
## dense_1 (Dense)                      (None, 1)                         6
## =========================================================================
## Total params: 21
## Trainable params: 21
## Non-trainable params: 0
## _________________________________________________________________________
```

The MLP model was set up as specified in the assignment description and a summary of it is displayed above.

Question 3) Train and test the model and plot the training and testing accuracy versus the training epochs of the respective data sets.

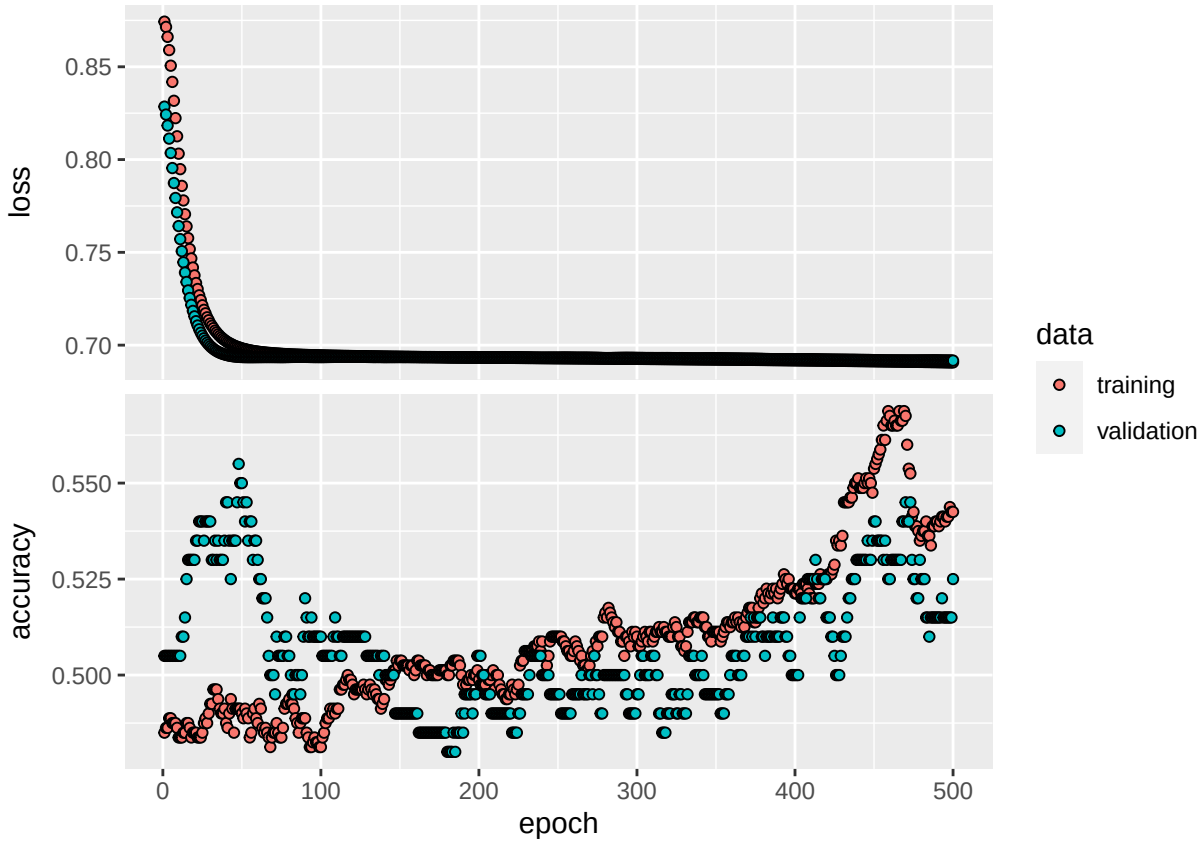
a. Report the training and testing accuracy and illustrate the plots. [7]

```
history <- model.mlp %>% fit(x_train, y_train, epochs=500, validation_data = list(x_test, y_test),
                             batch_size = 512) #Assuming 512 batch size
history
```

```
##
## Final epoch (plot to see history):
##         loss: 0.6907
##     accuracy: 0.5425
##     val_loss: 0.6917
## val_accuracy: 0.525
```

At the final epoch, this model achieved the relatively poor training and validation metrics as seen above. The previous question explicitly stated to use the test data as the validation data. The training and validation performance metric plots are illustrated on the next page.

```
plot(
  history,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", F),
  theme_bw = getOption("keras.plot.history.theme_bw", F)
)
```



b. Studying the plots and the accuracy metrics, do you think the current model suffers from a vanishing gradient problem? Provider reasons for your answer. [3]

As seen above, the loss plot seems to plateau completely, there seems to be no or extremely small changes in the loss of the network for both the training and validation sets. There is also quite a low final epoch accuracy for both of these data sets, which is quantifiable by noting the slow change in accuracy, or slow learning. This model improved it's training accuracy by around 4% at the first 100 epochs, then plateaued and the training accuracy appears to have increased by around 1% for the remaining epochs. Initially I thought this model suffers from the vanishing gradient problem, but upon noting that the vanishing gradient problem is due to repeated multiplications of derivatives and weights that are less than unity which harms the ability of feeding back an error signal, and since this model simply does not have enough layers to display this problem, I concluded that this model does not suffer from the vanishing gradient problem. This model therefore struggles to learn due to an unsophisticated architecture and poor initial selection of weights, instead of the vanishing gradient problem.

Question 4. Adding more hidden layers often improves the prediction capabilities of a model. Alter the existing model only by adding an additional 5 hidden layers (using the same specifications as the current hidden layer). The new model will therefore have 6 hidden layers, each with a hyperbolic tangent activation function and a Random Uniform kernel initializer with a range of [0, 1].

a. Report on the new model's training and testing accuracy as well as illustrating the plots (training and testing accuracy versus the training epochs) [7]

```
model.deeper <- keras_model_sequential() %>%    #Initialize model architecture
  layer_dense(units = 5, activation = "tanh", input_shape = c(2),
              kernel_initializer =
                initializer_random_uniform(minval = 0,
                                           maxval = 1,seed = 0)) %>%
  layer_dense(units = 5, activation = "tanh",
              kernel_initializer =
                initializer_random_uniform(minval = 0,
                                           maxval = 1,seed = 0)) %>%
  layer_dense(units = 5, activation = "tanh",
              kernel_initializer =
                initializer_random_uniform(minval = 0,
                                           maxval = 1,seed = 0)) %>%
  layer_dense(units = 5, activation = "tanh",
              kernel_initializer =
                initializer_random_uniform(minval = 0,
                                           maxval = 1,seed = 0)) %>%
  layer_dense(units = 5, activation = "tanh",
              kernel_initializer =
                initializer_random_uniform(minval = 0,
                                           maxval = 1,seed = 0)) %>%
  layer_dense(units = 5, activation = "tanh",
              kernel_initializer =
                initializer_random_uniform(minval = 0,
                                           maxval = 1,seed = 0)) %>%
  layer_dense(units = 1, activation = "sigmoid",
              kernel_initializer =
                initializer_random_uniform(minval = 0,
                                           maxval = 1, seed = 0))

model.deeper %>% compile(optimizer = optimizer_sgd(lr = 0.01, momentum = 0.9),
                         loss = "binary_crossentropy", metrics = "accuracy")
model.deeper
```

```
## Model
## Model: "sequential_1"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## dense_2 (Dense)                     (None, 5)                       15
## ________________________________________________________________________________
## dense_3 (Dense)                     (None, 5)                       30
## ________________________________________________________________________________
## dense_4 (Dense)                     (None, 5)                       30
## ________________________________________________________________________________
```

```
## dense_5 (Dense)                     (None, 5)                       30
## ________________________________________________________________________________
## dense_6 (Dense)                     (None, 5)                       30
## ________________________________________________________________________________
## dense_7 (Dense)                     (None, 5)                       30
## ________________________________________________________________________________
## dense_8 (Dense)                     (None, 1)                       6
## ================================================================================
## Total params: 171
## Trainable params: 171
## Non-trainable params: 0
## ________________________________________________________________________________
```

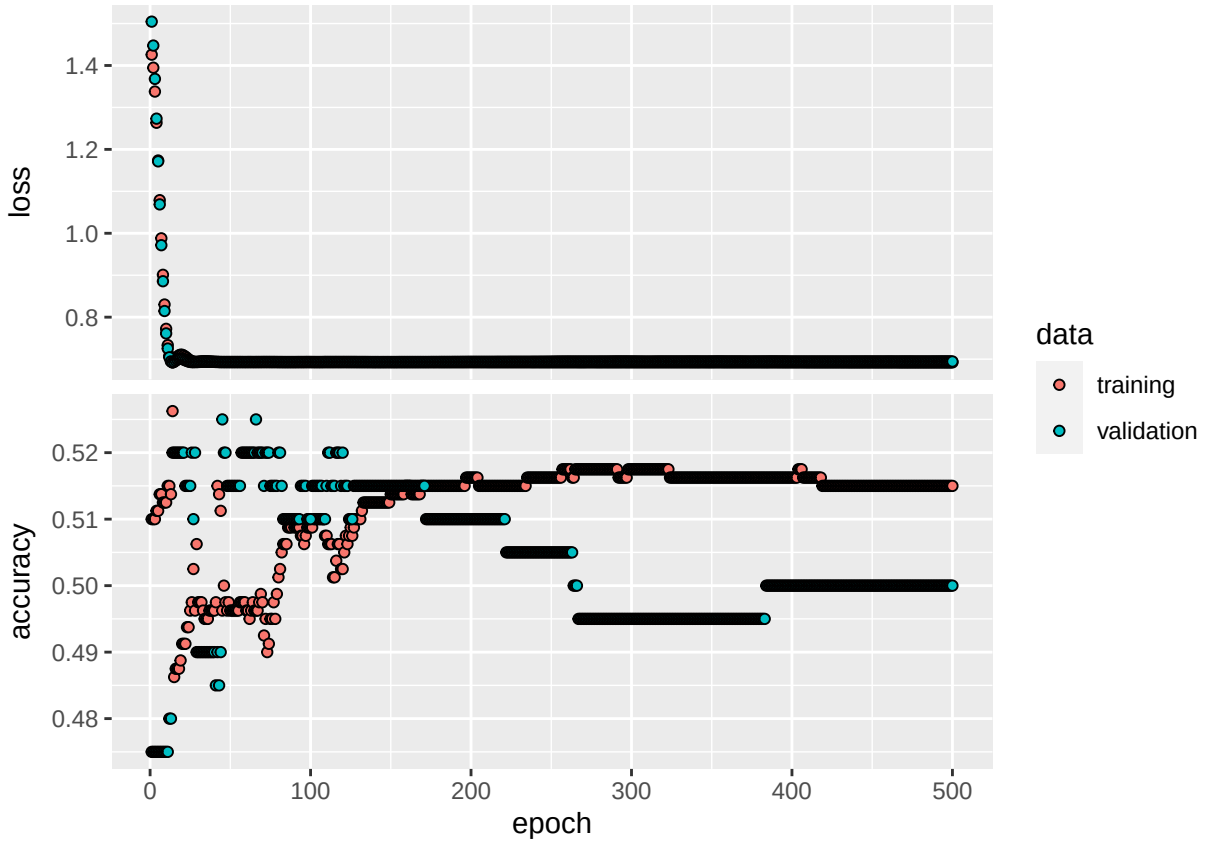
The model was set up as specified in the assignment detail, and a summary of it is displayed above

```
history <- model.deeper %>% fit(x_train, y_train, epochs=500, validation_data =
                                list(x_test, y_test),
                                batch_size = 512) #Assuming 512 batch size
history
```

```
##
## Final epoch (plot to see history):
##         loss: 0.6922
##     accuracy: 0.515
##     val_loss: 0.6944
## val_accuracy: 0.5
```

The training and validation final performance metrics are displayed above. The model can be seen to perform even worse than the previous simpler MLP did at its final epoch. This seems counterintuitive, since an unsophisticated model is performing better more powerful model *without* any signs of overfitting (as seen in the plots on the next page). This model achieves a low validation accuracy due to the vanishing gradient problem which will be discussed on the next page. Once again, the testing data was used as validation data as initially specified. The training and validation metrics were plotted as follows.

```
plot(
  history,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", F),
  theme_bw = getOption("keras.plot.history.theme_bw", F)
)
```



b. Studying the plots and the accuracy metrics, do you think the current model suffers from a vanishing gradient problem? Provider reasons for your answer. [3]

Yes, this model definitely suffers from a vanishing gradient problem. Unlike the previous model, this model has enough layers to fully show the effect of the poor initialization of the weights and the improper choice of activation functions. For this problem, we added more layers with hyperbolic tangent activation functions (*recalling the derivative of the hyperbolic tangent function is quite narrow ~[-2,2]*) and thereby essentially allocated enough space for the effect of vanishing gradients to come forth. The chain rule will ensure that the gradient is minimized and this can be confirmed by noting that the learning stops earlier and the accuracy and loss for the training sets barely change now. During the training of models, we typically would expect some sort of overfitting to appear at this point or at least some indication of learning, especially since this model has a decent amount of computational power for this problem, but the vanishing gradient ensures that the model cannot continue learning. Therefore, the too simple model from the first question is outperforming this complicated model.

Question 5) Setup [3]

```
model.final <- keras_model_sequential() %>%    #Initialize model architecture
  layer_dense(units = 5, activation = "relu", input_shape = c(2),
              kernel_initializer =
                initializer_he_uniform(seed = 0)) %>%
  layer_dense(units = 5, activation = "relu",
              kernel_initializer =
                initializer_he_uniform(seed = 0)) %>%
  layer_dense(units = 5, activation = "relu",
              kernel_initializer =
                initializer_he_uniform(seed = 0)) %>%
  layer_dense(units = 5, activation = "relu",
              kernel_initializer =
                initializer_he_uniform(seed = 0)) %>%
  layer_dense(units = 5, activation = "relu",
              kernel_initializer =
                initializer_he_uniform(seed = 0)) %>%
  layer_dense(units = 5, activation = "relu",
              kernel_initializer =
                initializer_he_uniform(seed = 0)) %>%
  layer_dense(units = 1, activation = "sigmoid",
              kernel_initializer =
                initializer_he_uniform(seed = 0))

model.final %>% compile(optimizer = optimizer_sgd(lr = 0.01, momentum = 0.9),
                        loss = "binary_crossentropy", metrics = "accuracy")

model.final
```

```
## Model
## Model: "sequential_2"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense_9 (Dense)              (None, 5)                 15
## _________________________________________________________________________
## dense_10 (Dense)             (None, 5)                 30
## _________________________________________________________________________
## dense_11 (Dense)             (None, 5)                 30
## _________________________________________________________________________
## dense_12 (Dense)             (None, 5)                 30
## _________________________________________________________________________
## dense_13 (Dense)             (None, 5)                 30
## _________________________________________________________________________
## dense_14 (Dense)             (None, 5)                 30
## _________________________________________________________________________
## dense_15 (Dense)             (None, 1)                 6
## =========================================================================
## Total params: 171
## Trainable params: 171
## Non-trainable params: 0
## _________________________________________________________________________
```

The model was set up as specified and a summary of it's architecture is displayed above

a. Report on the new model's training and testing accuracy as well as illustrating the plots (training and testing accuracy versus the training epochs) [7]

```
history <- model.final %>% fit(x_train, y_train, epochs=500,
                               validation_data = list(x_test, y_test),
                               batch_size = 512) #Assuming 512 batch size
history
```

```
##
## Final epoch (plot to see history):
##         loss: 0.5613
##     accuracy: 0.7088
##     val_loss: 0.5824
## val_accuracy: 0.705
```

The model's final validation and training metrics are displayed above. The accuracy is higher and the loss is lower when compared to both of the previous models. The training and validation metric plots are displayed on the following page. It can be seen that the loss and accuracy no longer plateau as before, and learning is still occuring. Due to the choice of activation function and weight initialization, I do not believe this model is suffering from a vanishing gradient. The other models' validation performance are displayed below and the RELU model can be seen to perform the best, with the lowest loss and highest accuracy.

```
model.final %>% evaluate(x_test, y_test) #Relu model validation results
```

```
##      loss  accuracy
## 0.5823627 0.7050000
```

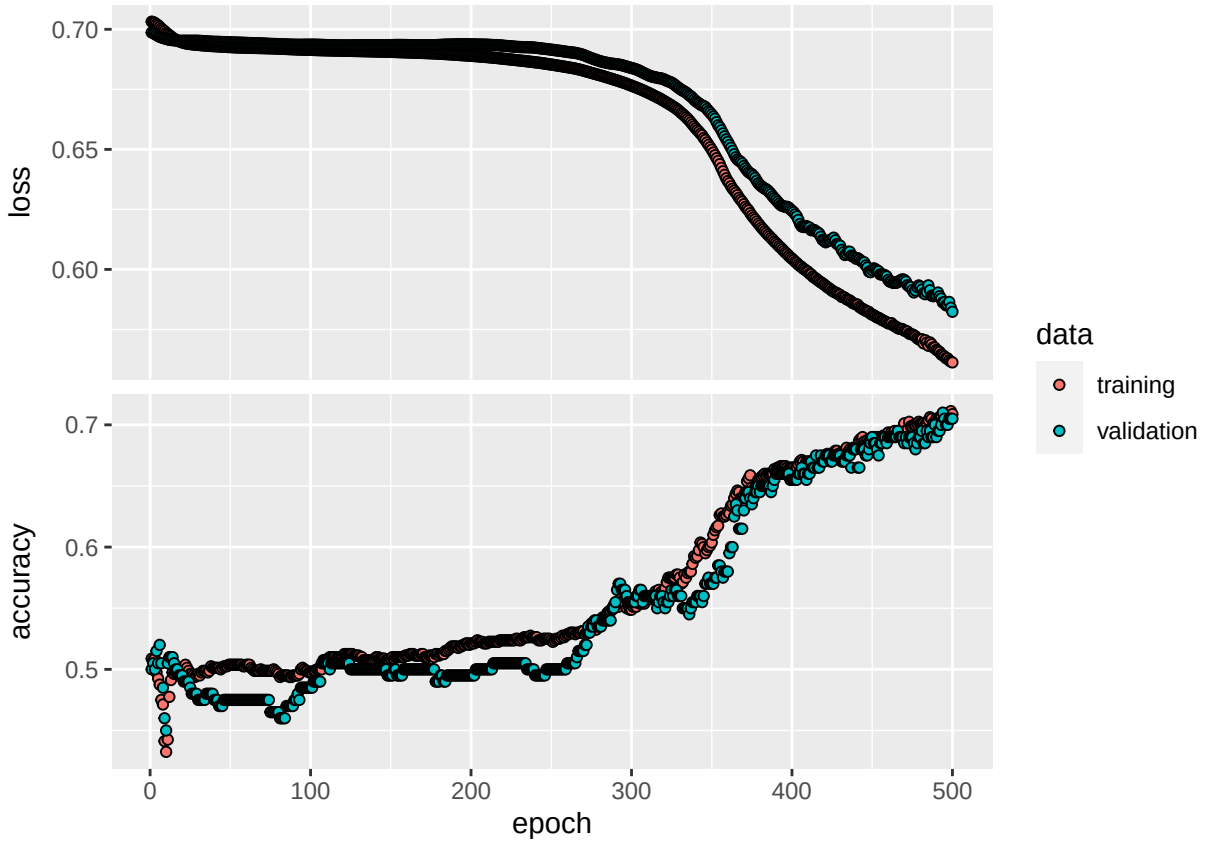
```
model.mlp %>% evaluate(x_test, y_test) #Intial model validation results
```

```
##      loss  accuracy
## 0.6916769 0.5250000
```

```
model.deeper %>% evaluate(x_test, y_test) #Deeper model validation results
```

```
##      loss  accuracy
## 0.6944346 0.5000000
```

```
plot(
  history,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", F),
  theme_bw = getOption("keras.plot.history.theme_bw", F)
)
```

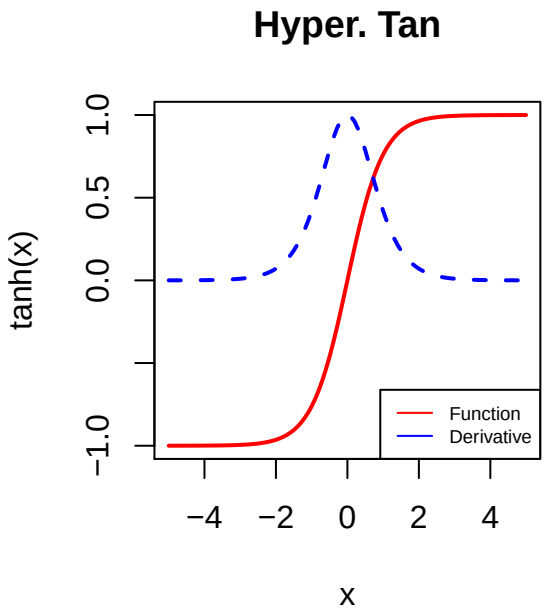


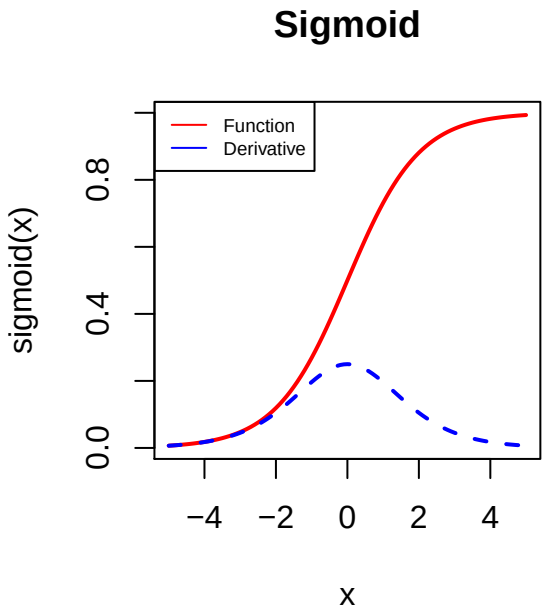
b. What impact did changing the activation function and kernel initializer have? [3]

The hyperbolic and sigmoid activation functions and their relative derivatives are displayed below. Recalling from class, the derivative of the cost function is used to change the weights and it is directly proportional to the derivative of the activation function as well as the weights of the network. These derivatives are multiplied due to the chain rule, and therefore by repeatedly multiplying numbers smaller than one, the product will diminish and no learning will occur. The initial and deeper models also implemented small initial weights, and the repeated multiplication of the small weights ensured that the process of learning was slowed down even further. Therefore, as soon as multiple layers are added the vanishing gradient problem quickly revealed itself. Upon changing the activation function to RELU and the initial weights to a wider range, the repeated multiplication of derivatives and weights smaller than unity was mitigated. This is because the RELU function can be seen to have a derivative of either 0 or either 1, and the weights are initialized between positive and negative sqrt(6/#input units in weight tensor)

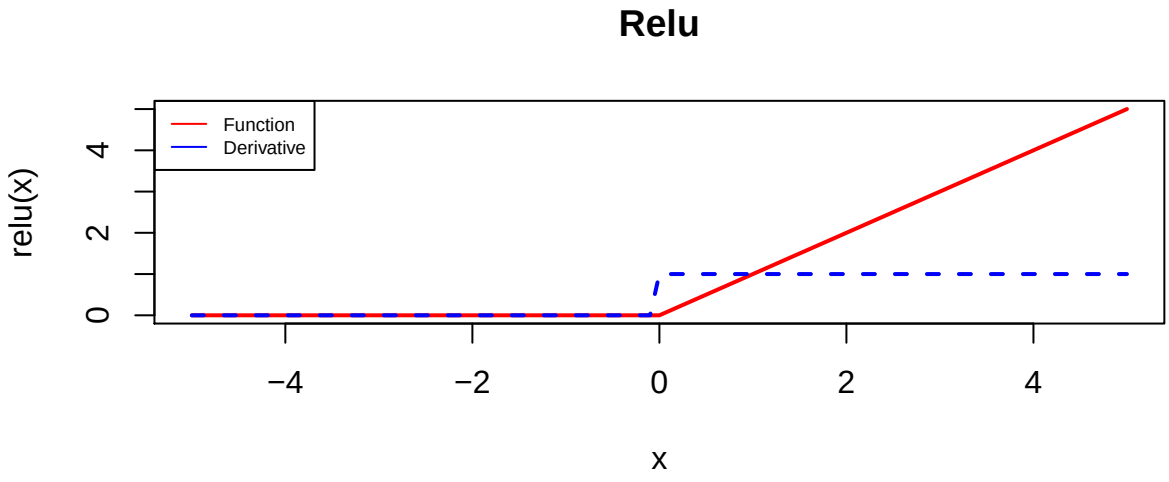
```
x <- seq(-5, 5, 0.1)
par(mfrow=c(1,2))
plot(x, tanh(x), type = "l", lwd=2, col = "red", main = "Hyper. Tan")
lines(x, (1-(tanh(x))^2), lty = 2, lwd=2, col = "blue") #deriv
legend("bottomright", legend = c("Function", "Derivative"), lty = 1,
       col = c("red", "blue"), cex = 0.6)
sigmoid = function(x)  1/(1+exp(-x))
plot(x, sigmoid(x), type = "l", col = "red", lwd=2, main = "Sigmoid")
lines(x, (sigmoid(x)-(sigmoid(x))^2),  lwd=2, lty = 2, col = "blue") #deriv
```

```
legend("topleft", legend = c("Function", "Derivative"), lty = 1,
       col = c("red", "blue"), cex = 0.6)
```






```
par(mfrow=c(1,1))
x <- seq(-5, 5, 0.1)
relu <- function(x) ifelse(x < 0, 0, x)
ddxrelu <- function(x) ifelse(x < 0, 0, 1)
plot(x, relu(x), type = "l", lwd=2, col = "red", main = "Relu")
lines(x, ddxrelu(x), lty = 2, lwd=2, col = "blue") #deriv
legend("topleft", legend = c("Function", "Derivative"), lty = 1, col = c("red", "blue"), cex = 0.6)
```



c. Studying the plots and the accuracy metrics, do you think the current model suffers from a vanishing gradient problem? Provider reasons for your answer. [3]

No, the vanishing gradient problem is characterized by slow or no learning and early stopping of training improvement. This RELU model can be seen to reduce its loss and increase it's accuracy substantially, especially at the later epochs, and therefore the model is still learning and is not suffering from the vanishing gradient problem. This model implements RELU activation functions which as seen above, have a derivative of either 0 or 1. This ensures that the feedback of gradients will not be diminished due to the chain rule. The RELU model also intelligently initializes its' weights. This initialization will also resolve the issue mentioned previously, where the small weights reduced the derivative of the cost function and replicated the effect of the vanishing gradient problem.

d. Compare the performance of the latest model with the initial 3-layer MLP model. [3]

At the final epoch the intial 3-layer MLP performed worse than the RELU model, since it achieved a higher training and validation loss than the RELU model, as well as a lower training and validation accuracy. The RELU model outperformed the initial MLP with a final lower training and validation loss, as well as a higher training and validation accuracy. This difference in losses and accuracies can be quantified by inspecting the relevant performance plots, where it can be seen that the initial model is learning slowly and only managed to increase its testing accuracy by around 5%, and training accuracy by around 10% for the duration of the 500 epochs. The RELU model managed to increase both its training and testing accuracy by around 20% for the duration of the 500 epochs. The RELU model is undoubtedly the better model. The validation results for the trained models are displayed below.

```
model.final %>% evaluate(x_test, y_test) #Relu model validation results
```

```
##      loss  accuracy
## 0.5823627 0.7050000
```

```
model.mlp %>% evaluate(x_test, y_test) #Intial model validation results
```

```
##      loss  accuracy
## 0.6916769 0.5250000
```

Overfitting/Underfitting in Deep Learning [50]

Q1.1. Explain what effect the following operations generally will have on the bias and variance of your model. Fill in one of 'increases', 'decreases' or 'no change' in each of the cells: [12]

Operation	Bias	Variance
Regularizing the weights	Increase	Decrease
Increasing the size of the layers	Decrease	Increase
Using dropout to train a deep neural network	Increase	Decrease
More training data	No change	Decrease
Implementing early stopping	No change	Decrease
Combining multiple outputs from neural networks trained in parallel	Decrease	Decrease

Q1.2. Which of the following techniques perform similar operations as dropout in a neural network? [2]

A. Bagging

Q1.3. The red curve above denotes training accuracy with respect to each epoch in a deep learning algorithm. Both the green and blue curves denote validation accuracy. [2]

B) Blue Curve

Q1.4. Suppose you are using early stopping mechanism with patience as 2, at which point will the neural network model stop training? [2]

C) 4

Question 5) In this question, we will investigate a dataset that is prone to overfitting due to some of its features. You will start with a base model and implement certain features to prevent overfitting.

i) Load the training set called trainX_overfitting.csv into your environment.

```
rm(list = ls())
library(keras) #The question explicitly states to use the training dataset
x_train <- as.matrix(read.csv("trainX_overfitting.csv", header = F))
y_train <- as.matrix(read.csv("trainY_overfitting.csv", header = F))
```

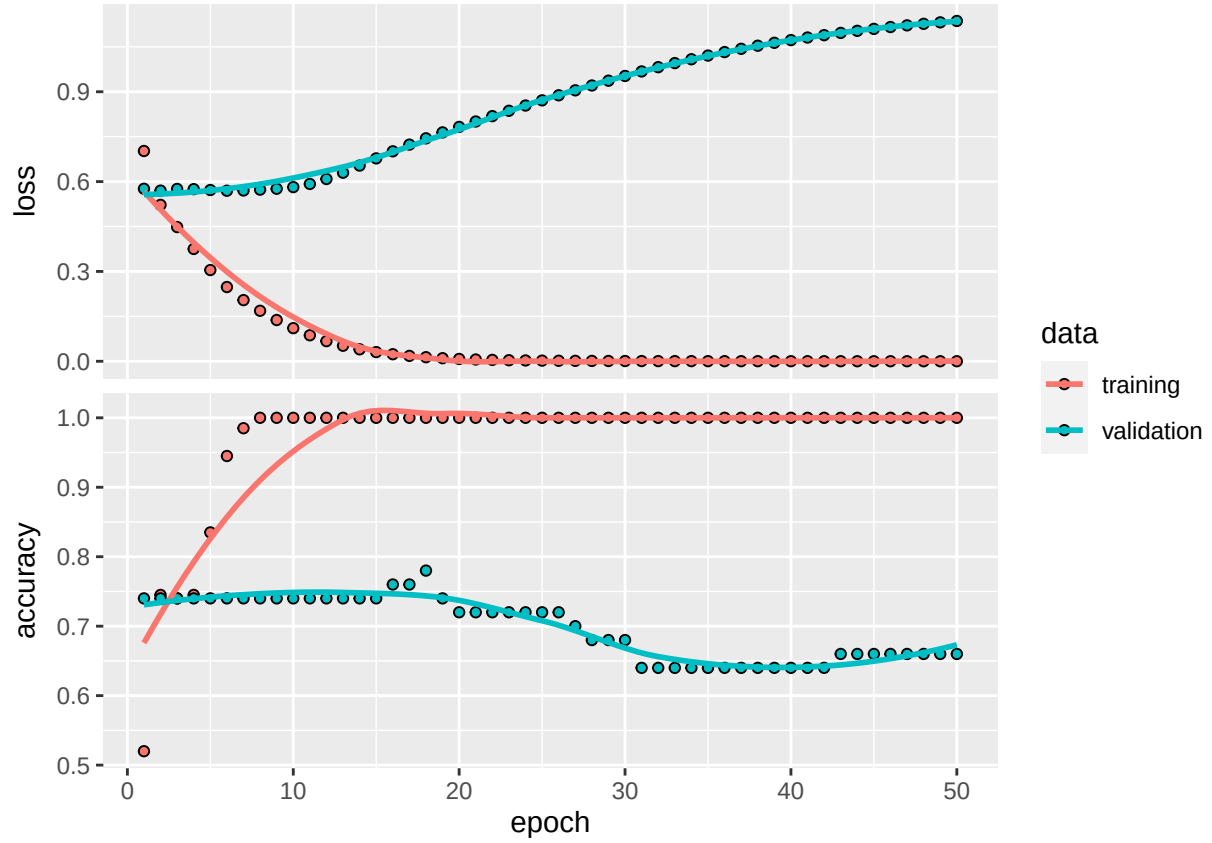
ii) For the base model, develop the following deep network. [10]

```
model <- keras_model_sequential() %>%    #Initialize model architecture
  layer_dense(units = 300, activation = "relu", input_shape = c(300)) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(optimizer = 'adam',
                  loss = "binary_crossentropy", metrics = "accuracy")
```

iii) Train and evaluate the model by printing the training and validation accuracy. What observations can you make from the plot? Is the model currently overfitting? If so, provide reasons for your answer. [5]

```
history <- model %>% fit(x_train, y_train, epochs=50, validation_split = 0.2,
                         batch_size = 512) #Assuming 512 batch size
```

```
plot(
  history,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", T),
  theme_bw = getOption("keras.plot.history.theme_bw", F)
)
```



This model is overfitting, the graph above shows that the validation loss, *i.e. how well the model will generalize to new data is increasing*, while the training loss is decreasing. That is a textbook example of overfitting, and furthermore the second plot displays that the validation accuracy remains relatively the same while the training accuracy increases, and not slightly, it increases the training accuracy to a value of 100% while the validation accuracy remains at around 70%. Therefore the model is undoubtedly learning the noise and other sorts of crooks and nooks in the data patterns that we are not interested in modelling. The model is clearly also trained too long which contributed to its overfitting nature, and it achieved the final-epoch metrics displayed below

```
history
```

```
##
## Final epoch (plot to see history):
##         loss: 0.0001383
```

```
##     accuracy: 1
##     val_loss: 1.136
## val_accuracy: 0.66
```

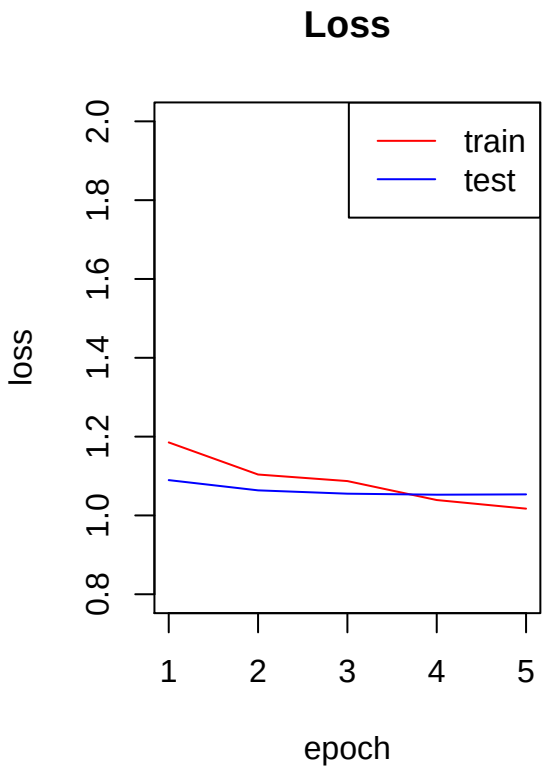
iv) Implement at least 3 measures preventing overfitting. Provide reasons for implementing the techniques you decide on. [7]

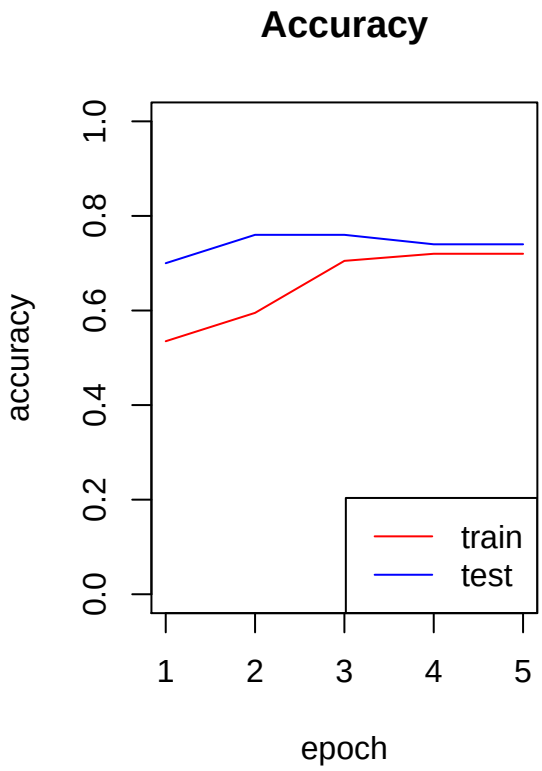
```
model.reg <- keras_model_sequential() %>%
  layer_dense(units = 300, activation = "relu", input_shape = c(300),
  kernel_regularizer = regularizer_l2(l = 0.001)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 64, activation = "relu",
  kernel_regularizer = regularizer_l2(l = 0.001)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 64, activation = "relu",
  kernel_regularizer = regularizer_l2(l = 0.001)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 1, activation = "sigmoid")

model.reg %>% compile(optimizer = 'adam',
                  loss = "binary_crossentropy", metrics = "accuracy")
history.reg <- model.reg %>% fit(x_train, y_train, epochs=15, validation_split = 0.2,
                                 callback = callback_early_stopping(monitor = "val_loss",
                                 patience = 1), batch_size = 512) #Assuming 512 batch size
```

Four measures were implemented to prevent the overfitting of the model. The first measure was the addition of dropout with a rate of 30%, which will allow the removal of units and ultimately assist in finding a more suitable and robust architecture for this problem. The second method implemented was L2 regularization, which will ensure that the weights are decayed according to the chosen lamdba. Furthermore the architecture of the network was reduced in an attempt to minimize the capability of the model to overfit, and also the final and simplest regularization method that was implemented is early stopping with a patience factor of 2.

v) Print the training and validation accuracy of the new model. Have you improved the overfitting nature of the original model? [6]

```
par(mfrow=c(1,2))
plot(history.reg$metrics$loss, main="Loss", xlab = "epoch", ylab="loss",
     col="red", lty=1, type = 'l', ylim = c(0.8,2))
lines(history.reg$metrics$val_loss, xlab = "epoch", ylab="loss",
      col="blue", lty=1, type = 'l')
legend("topright", c("train","test"), col=c("red", "blue"), lty=1)
plot(history.reg$metrics$accuracy, main="Accuracy", xlab = "epoch", ylab="accuracy",
     col="red", lty=1, type = 'l', ylim = c(0,1))
lines(history.reg$metrics$val_accuracy, xlab = "epoch", ylab="loss",
      col="blue", lty=1, type = 'l')
legend("bottomright", c("train","test"), col=c("red", "blue"), lty=1)
```






```
history.reg #Regularized model
```

```
##
## Final epoch (plot to see history):
##         loss: 1.017
##     accuracy: 0.72
##     val_loss: 1.053
## val_accuracy: 0.74
```

```
history #Overfit model
```

```
##
## Final epoch (plot to see history):
##         loss: 0.0001383
##     accuracy: 1
##     val_loss: 1.136
## val_accuracy: 0.66
```

The training and validation accuracy and loss are displayed above. By inspecting the metrics achieved at the final training epoch, we can note that the regularized model now does perform worse on the training data, but it generalizes better as seen in the validation set's increased accuracy and lowered loss. This is therefore the preferred model. I believe that the overfitting nature has been improved, since the validation and training metrics are very similar for the regularized model, with no large growth in in difference between the training and validation metrics, as opposed to the initial overfit model.

vi) Name and discuss 2 techniques to prevent overfitting in addition to the techniques you implemented. [4]

As mentioned previously, I implemented architecture reduction, L2 weight regularization, dropout and early stopping. Architecture reduction serves to reduce the capability of the network to overfit, L2 weight regularization forces the weights to decay and thereby reduces the complexity and capability of the network to overfit by effectively reducing the impact of certain neurons at specific points. Dropout forces the network to implement less complex layers, which also reduces it's potential to overfit. Early stopping simply keeps a check on the training epochs, to ensure the model is terminated at an optimal point. As an additional 2 techniques to discuss, I considered data augmentation with stochaistic noise and weight sharing. Data augmentation is quite relevant to this problem since there are only 250 training entries, and by implementing data augmentation we can increase the amount of data available by adding, for example Gaussian noise at input and hidden layers which would essentially train the network on more data than it actually has. As with all the previous methods of regularization, this is only active during the training phase of the model and no noise will be added during testing. Weight sharing might also be of benefit to this model, since it limits the total degrees of freedom for a model architecture. This is achieved by tying certain neurons parameter's together, which has the effect of making the model simpler with less capability of learning noise, which in turn should lead to the prevention of overfitting.