Formative Assessment: Exploring Deep Learning

Francois van Zyl - SN: 18620426

25/06/2020

Description of Problem

As we were told to find a problem that seemed interesting, I searched for various datasets used with deep learning applications and eventually opted for the IMDB movie review dataset that contained 50 000 reviews of a multitude of movies. The main reason for selecting this was because I thought it would be a cool problem to start my first deep learning project with. The dataset was downloaded from kaggle and is publicly available at the following link. [*https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews*].

This is a relatively well-known dataset, and typically people have used it as a benchmark for performing binary sentiment classificiation. This is due to it exceeding the size of previous benchmarks for binary classification, and the sheer magnitude of it makes it an excellent practice-problem for deep learning algorithms in the environment of text analysis and classification. It has effectively been used to inspect various models' performance, fine-tune and re-evaluate them.

```r
rm(list = ls())
set.seed(0)
setwd("/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/Formative/")
imdb<-read.csv(file = "IMDB Dataset.csv")
library(keras)
library(dplyr)
library(ggplot2)
```

```r
glimpse(imdb)
```

```
## Observations: 50,000
## Variables: 2
## $ review    <fct> "One of the other reviewers has mentioned that after watc...
## $ sentiment <fct> positive, positive, positive, negative, positive, positiv...
```

```r
anyNA(imdb)
```

```
## [1] FALSE
```

```r
summary(imdb$sentiment)
```

```
## negative positive
##    25000    25000
```

After reading the dataset in, I inspected the variables present within it. There are two attributes within this dataset, with 50 000 total observations in. The first attribute contains strings of reviews of certain movies, and the second attribute contains a *binary* rating of whether the specific review of a movie is considered *positive* or *negative*. I found that there are no missing values within this dataset which will need to be addressed during the pre-processing phase. I did note however, that the text does contain special characters that will need to be addressed.

My Approach

I adopted a similar approach to previous applications of this dataset. This implies my main goal is to perform the classification of the review as belonging to either a positive or a negative class based on the words in the review attribute. To ease the computational cost of this project, I concluded that the 1000 most occurring words within the review attribute should provide me with enough information to model the problem suitably. On a high-level, I will commence this project by vectorizing the words within the review as well as preprocessing the target attribute. Thereafter, I will use this data to create a model that is overfit on the training data, and then attempt to improve it's performance by 4 distinct regularization methods, namely architecture reduction, dropout, L2 regularization and early stopping.

Data pre-processing

As seen in the initial inspection of the data, the review and sentiment attributes are both currently factor data types. This part of the pre-processing will address correcting the sentiment attribute's data type and encoding. The sentiment attribute currently has the following levels (*unique classes*).

```r
levels(imdb$sentiment)
```

```
## [1] "negative" "positive"
```

Neural networks that perform binary classification should output the probability that a instance belongs to a class. This is usually accomplished by making use of the Sigmoid activation function, which ranges from [0, 1]. Therefore the sentiment attribute should be either [0,1] for this problem. This problem was addressed as follows, where the target features are all stored as numeric data types within the object y.

```r
levels(imdb$sentiment)[c(1:2)]<-c(0, 1)
y <- as.numeric(as.character(imdb$sentiment))
```

Now that the target attribute has been pre-processed, we can commence the pre-processing of the review attribute. The review attribute contains a multitude words that are unusable for a neural network in their current form. To address this problem, I converted this text into a sequence of integers by using a text tokenizer. This tokenizer proved quite handy, as it allowed me to remove all special characters and punctuation. The tokenizer was set up as follows, where it takes into account the top 1000 most-occurring words within the review dataset. To prevent duplication this tokenizer converts all words to lower-case. After setting up the tokenizer, I updated the tokenizer's vocabulary by fitting the review attribute's words to it.

```r
tokenizer <- text_tokenizer(num_words = 1000,
             filters = "!\"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n",
             lower = TRUE,
             split = " ",
             char_level = FALSE) %>% fit_text_tokenizer(imdb$review)
```

```
message("Documents: ", tokenizer$document_count)
```

```
## Documents: 50000
```

```
message("Total words: ", tokenizer$num_words)
```

```
## Total words: 1000
```

The tokenizer's word index is therefore set on the 1000 most occurring words across the 50 000 documents within the review attribute. This tokenizer can now be used to vectorize the texts to a binary matrix as follows, where each row corresponds to a document within review and each column to a unique word.

```
x <- texts_to_matrix(tokenizer, imdb$review, mode = "binary")
x[1:10, 1:10]
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    1    1    1    1    1    1    1    1     1
##  [2,]    0    1    1    1    1    1    1    1    0     1
##  [3,]    0    1    1    1    1    1    1    1    1     1
##  [4,]    0    1    1    1    1    1    1    1    1     0
##  [5,]    0    1    1    1    1    1    1    1    1     0
##  [6,]    0    1    1    1    1    1    1    0    1     1
##  [7,]    0    1    1    1    1    1    1    0    1     1
##  [8,]    0    1    1    1    1    1    1    1    1     1
##  [9,]    0    1    1    1    1    1    1    1    1     0
## [10,]    0    0    0    0    0    0    0    1    0     1
```

```
dim(x)
```

```
## [1] 50000  1000
```

This dataset does not require any further pre-processing *(such as normalization, feature extraction or missing value imputation)*. I concluded that I could move on to considering the model development phase of this project.

Model Development

I developed my models by making use of keras' sequential models. I started this development by first considering what I think the model should use, as seen below.

- Use sigmoid function as output neuron's activation function
  - Only one output neuron required since this is binary classification
  - Want to predict probability of class outcome
- Use relu function at intermediate hidden layers
  - Standard initial selection for many applications, including binary classification
- The input has a shape of 1000

  - There are 1000 inputs or words
- Use binary cross entropy as the loss function
  - The most inuitive choice, this is a binary classification problem
- Use accuracy as the performance metric
  - Standard initial selection for binary classification problems
- Start with batch size of 512
  - The same as the batch size in the twitter example from class
- RMSprop
  - Popular inital selection

Thereafter, I considered the parameter choices that I am less certain of.

- How many hidden units should the intermediate layers contain?
- How many hidden layers should there be?
- How many epochs should I evaluate the model over?

I decided to choose these values based on similar projects found online as well as the provided class material. I decided to drastically *overfit* my first model, and view the effect of it. I will thereafter reduce the complexity and apply other regularization techniques. I therefore decided to select three hidden layers with 512 units for 20 epochs for my first model. The model was set up as follows.

Once again, note that this is intentionally overfit.

```r
model <- keras_model_sequential() %>%
            layer_dense(units = 512, activation = "relu", input_shape = c(1000)) %>%
              layer_dense(units = 512, activation = "relu") %>%
                 layer_dense(units = 512, activation = "relu") %>%
                   layer_dense(units = 1, activation = "sigmoid")

model %>% compile(optimizer = "rmsprop",loss = "binary_crossentropy", metrics = "accuracy")
history <- model %>% fit(x, y, epochs = 20, batch_size = 512)
history
```

```
##
## Final epoch (plot to see history):
##     loss: 0.00006723
## accuracy: 1
```

The initial model's performance is displayed above. This model is definitely overfit. At it's final epoch it reached an accuracy of 99.54%, and a loss of approximately 0.04. Note that the training data used in this case is the entire dataset. This model will generalize poorly to new data. To improve this model, I proceeded to split my data into training, testing and validation sets.

Model Improvement

The training, testing and validation sets were set up as follows.

```
index.train<-sample(seq_len(nrow(x)), 0.80*nrow(x))
index.valid<-seq_len(0.2*length(index.train))
x_train <- x[index.train,]#Taken from master x
x_valid <- x_train[index.valid,]#Taken from train x
x_train <- x_train[-index.valid,]#Removed from train x
x_test <- x[-index.train,]
y_train <- y[index.train]
y_valid <- y_train[index.valid]#Taken from train y
y_train <- y_train[-index.valid]
y_test <- y[-index.train]
```

To take a quick peek at how our initial overfitted base model would have generalized to new data, I retrained the model on the above training data and tested it on the testing data. (*note no validation set is included yet*)

```
#Model 1 Lets see how initial model performs on unseen data
model <- keras_model_sequential() %>%
            layer_dense(units = 512, activation = "relu", input_shape = c(1000))%>%
                layer_dense(units = 512, activation = "relu") %>%
                 layer_dense(units = 512, activation = "relu") %>%
                   layer_dense(units = 1, activation = "sigmoid")
model %>% compile(optimizer = "rmsprop",loss = "binary_crossentropy", metrics = "accuracy")
history<-model %>% fit(x_train, y_train, epochs = 20, batch_size = 512)
model %>% summary()
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## dense_4 (Dense)                     (None, 512)                     512512
## ________________________________________________________________________________
## dense_5 (Dense)                     (None, 512)                     262656
## ________________________________________________________________________________
## dense_6 (Dense)                     (None, 512)                     262656
## ________________________________________________________________________________
## dense_7 (Dense)                     (None, 1)                       513
## ================================================================================
## Total params: 1,038,337
## Trainable params: 1,038,337
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
model.test.1 <- model %>% evaluate(x_test, y_test)
history#Training
```

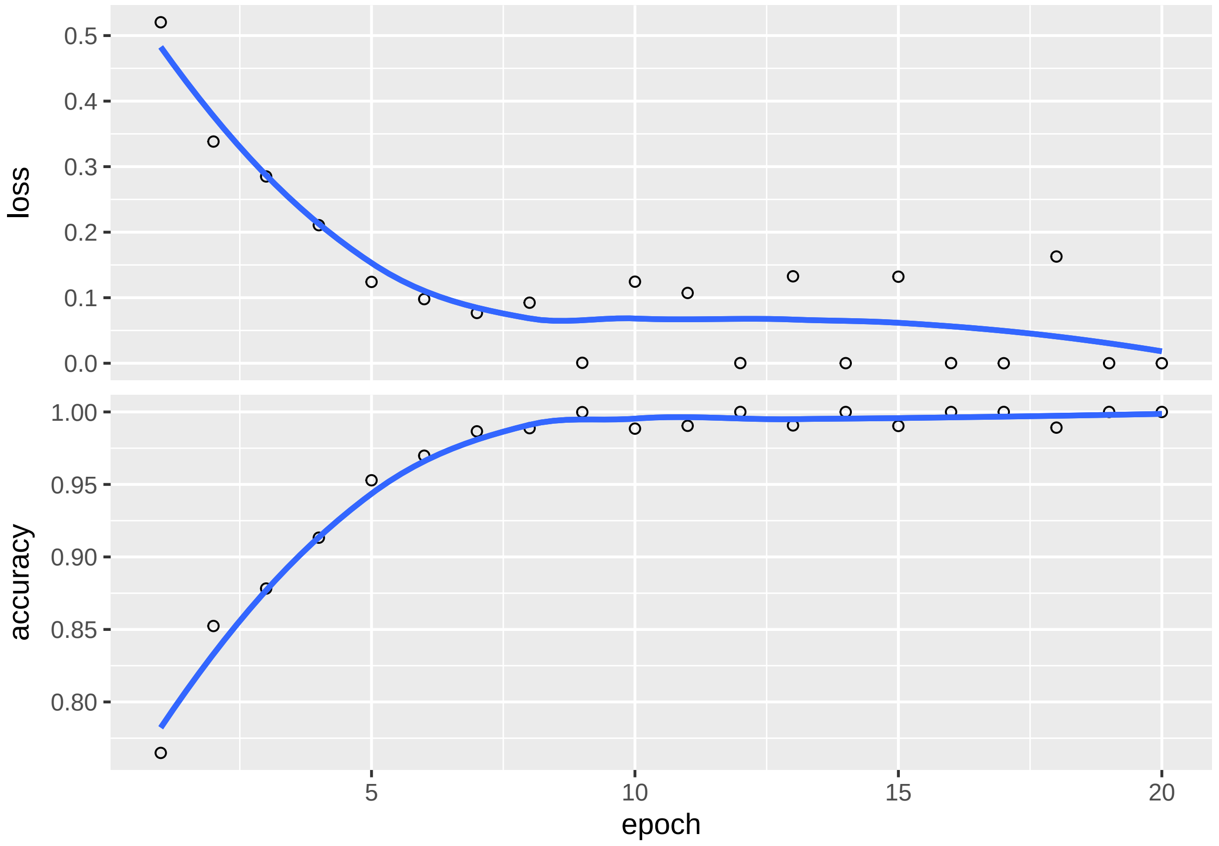
```
##
## Final epoch (plot to see history):
##     loss: 0.000005043
## accuracy: 1
```

```r
model.test.1 #Testing
```

```
##      loss accuracy
## 1.910669 0.835400
```

As seen above this model has more than 1 million total parameters. This is definitely far too much computational cost for a model that that has 1000 inputs. The model achieved a a near perfect training accuracy and exceptionally low loss value. This strikes a contrast to its testing data, where it had a substantially lower accuracy and higher loss, as is typical for overfitting. The model's performance metrics are displayed below and it can be seen that the training validation loss is increasing while the training error is decreasing. This is indicative that the model is still overfit, and it's training time should be limited. This will be addressed in the final regularization technique applied.

```r
plot(history, method =  "ggplot2") + geom_smooth(se = F, method = 'loess')
```



The simplest and most intuitive regularization method would be to to reduce the model architecture. The model architecture was reduced by removing one of the three hidden layers, as well as taking the amount of hidden units within the layers down from 512 to a more conservative 64. Note that this selection of hidden units was based on the class example. The model was retrained as follows.

```r
#Model 2. Model with reduced model architecture
model <- keras_model_sequential() %>%
      layer_dense(units = 64, activation = "relu", input_shape = c(1000)) %>%
         layer_dense(units = 64, activation = "relu") %>%
          layer_dense(units = 1, activation = "sigmoid")

model %>% compile(optimizer = "rmsprop",loss = "binary_crossentropy", metrics = "accuracy")
history <- model %>% fit(x_train, y_train, epochs = 20, batch_size = 512,
                validation_data = list(x_valid, y_valid))
model %>% summary()
```

```
## Model: "sequential_2"
## ________________________________________________________________________________
## Layer (type)                      Output Shape                      Param #
## ================================================================================
## dense_8 (Dense)                   (None, 64)                        64064
## ________________________________________________________________________________
## dense_9 (Dense)                   (None, 64)                        4160
## ________________________________________________________________________________
## dense_10 (Dense)                  (None, 1)                         65
## ================================================================================
## Total params: 68,289
## Trainable params: 68,289
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
model.test.2 <- model %>% evaluate(x_test, y_test)
history#Training
```

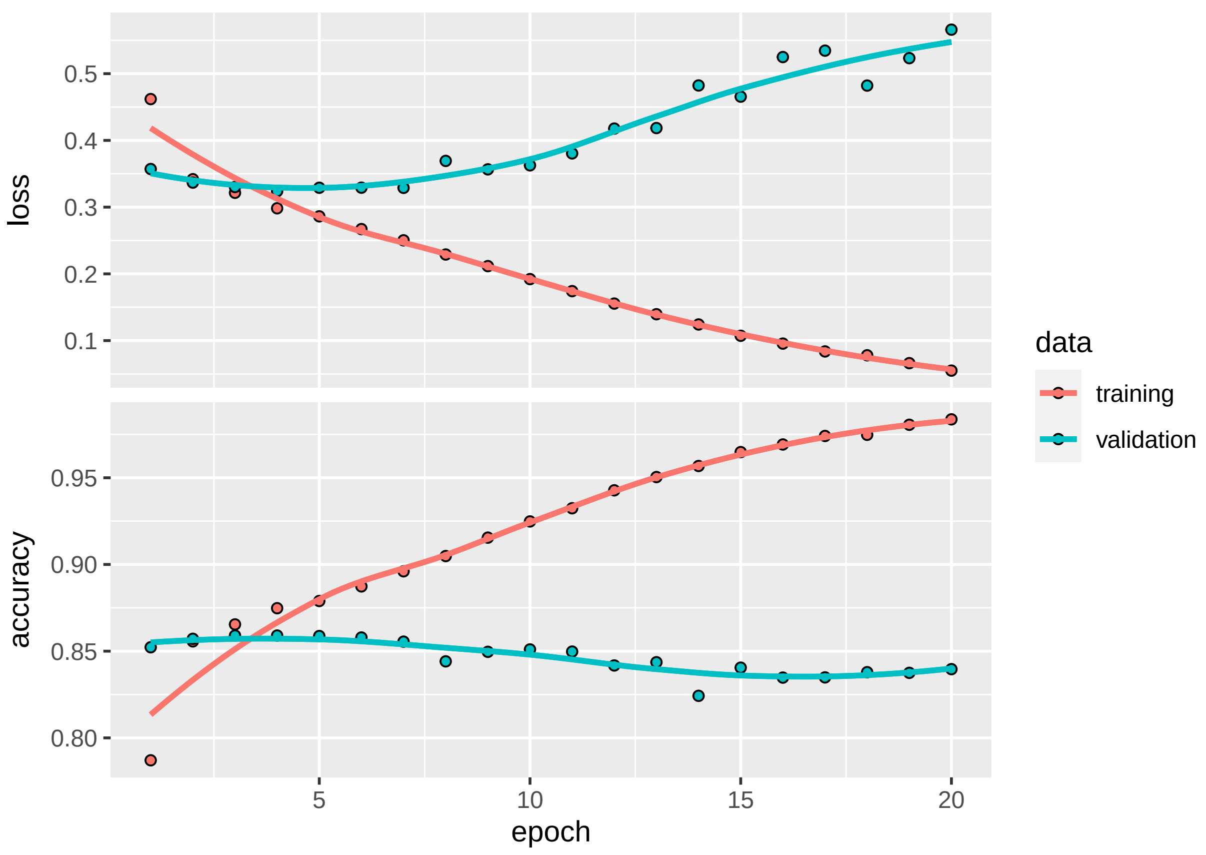
```
##
## Final epoch (plot to see history):
##         loss: 0.05498
##     accuracy: 0.9837
##     val_loss: 0.5659
## val_accuracy: 0.8396
```

```
model.test.2 #Testing
```

```
##      loss  accuracy
## 0.5511383 0.8480000
```

The model can now be seen to have a lower training accuracy and higher loss, but a better testing loss when compared to the initial overfitted model. This model can now be accepted as the baseline. Note that the testing accuracy did not improve, this is due to model still being trained to long. It is important to note that this model is still overfit. This can be seen in the figure below, where the validation loss is increasing steeply and the training loss is decreasing. Therefore, the model requires more work.

```
plot(history, method =  "ggplot2") + geom_smooth(se = F, method = 'loess')
```



As a next regularization technique, I considered adding dropout to the system. A dropout probability of 0.5 was selected based on material found online and in the class material provided to us. Note that this model is now based on the current baseline, which is that of the reduced architecture. The dropout was implemented as follows.

```r
#Model 3. Dropout + Reduced Architecture
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = c(1000)) %>%
    layer_dropout(rate = 0.5) %>%
      layer_dense(units = 64, activation = "relu") %>%
        layer_dropout(rate = 0.5) %>%
          layer_dense(units = 1, activation = "sigmoid")

model %>% compile(optimizer = "rmsprop",loss = "binary_crossentropy", metrics = "accuracy")
history <- model %>% fit(x_train, y_train, epochs = 20, batch_size = 512,
                         validation_data = list(x_valid, y_valid))
model %>% summary()
```

```
## Model: "sequential_3"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                      Param #
## ================================================================================
## dense_11 (Dense)                    (None, 64)                        64064
## ________________________________________________________________________________
## dropout (Dropout)                   (None, 64)                        0
## ________________________________________________________________________________
## dense_12 (Dense)                    (None, 64)                      4160
## ________________________________________________________________________________
## dropout_1 (Dropout)                 (None, 64)                      0
## ________________________________________________________________________________
## dense_13 (Dense)                    (None, 1)                       65
## ================================================================================
## Total params: 68,289
## Trainable params: 68,289
## Non-trainable params: 0
## ________________________________________________________________________________
```

```r
model.test.3 <- model %>% evaluate(x_test, y_test)
history#Training
```

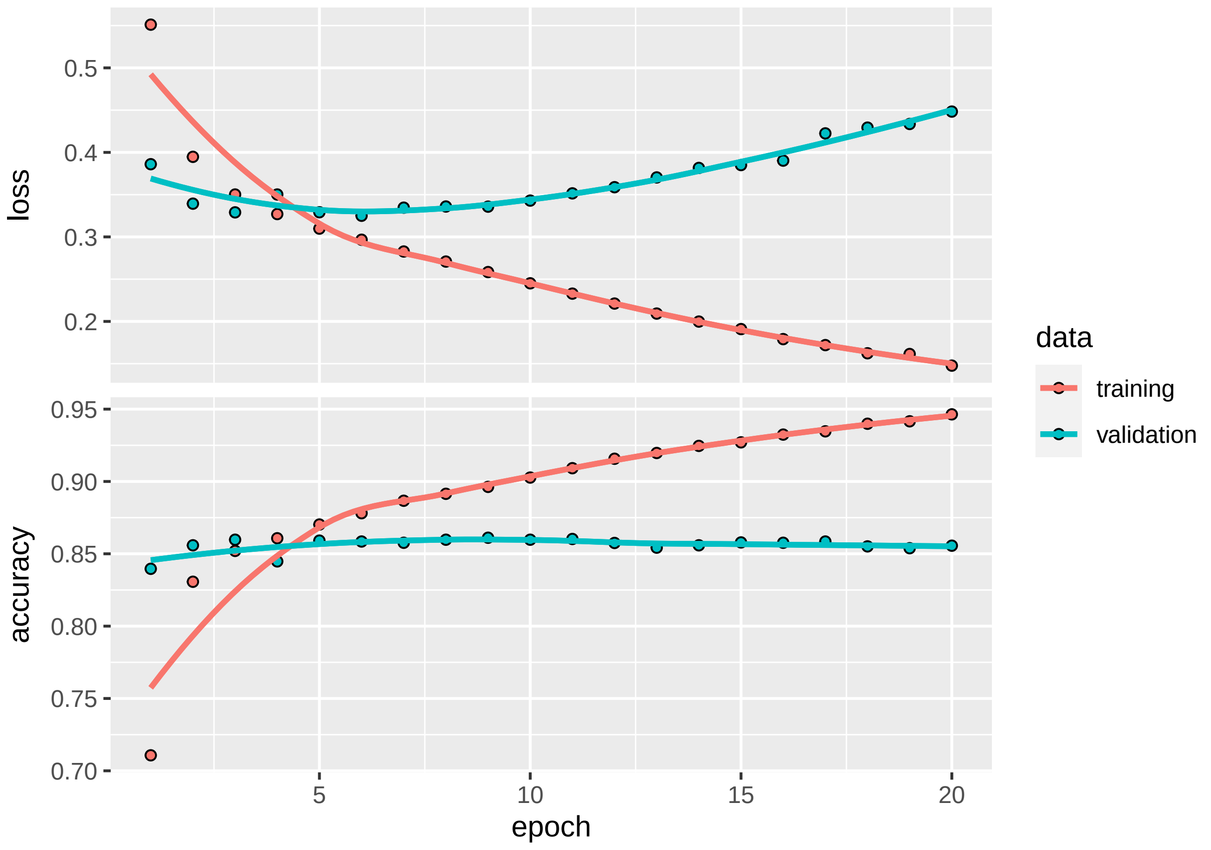
```
##
## Final epoch (plot to see history):
##         loss: 0.1477
##     accuracy: 0.9464
##     val_loss: 0.4483
## val_accuracy: 0.8556
```

```r
model.test.3 #Testing
```

```
##      loss  accuracy
## 0.4345258 0.8549000
```

The testing loss of the model can be seen to be reduced further, and the accuracy is increased slightly at the expense of the training loss and training accuracy. This can be expected since we are busy reducing the models overfitting on the training data.

```r
plot(history, method =  "ggplot2") + geom_smooth(se = F, method = 'loess')
```



The model's training and validation accuracy and loss are displayed above, and overfitting is still present as we havent addressed the prolonged training epochs. However, this model performs better on new data than the model containing only the simplest architecture, and the gradient of the validation loss does not seem to be increasing as abruptly as previously. This model is now accepted as the new baseline for this project.

To see if I could reduce the effect of overfitting further, I implemented L2 regularization as a third regularization technique.

```
#Model 4. Dropout + Reduced + L2_Reg
model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu", input_shape = c(1000),
                kernel_regularizer = regularizer_l2(l = 0.001)) %>%
      layer_dropout(rate = 0.5) %>%
         layer_dense(units = 64, activation = "relu",
                     kernel_regularizer = regularizer_l2(l = 0.001)) %>%
            layer_dropout(rate = 0.5) %>%
              layer_dense(units = 1, activation = "sigmoid")

model %>% compile(optimizer = "rmsprop",loss = "binary_crossentropy", metrics = "accuracy")
history <- model %>% fit(x_train, y_train, epochs = 20, batch_size = 512,
                         validation_data = list(x_valid, y_valid))
model %>% summary()
```

```
## Model: "sequential_4"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## dense_14 (Dense)                    (None, 64)                      64064
## ________________________________________________________________________________
## dropout_2 (Dropout)                 (None, 64)                      0
## ________________________________________________________________________________
## dense_15 (Dense)                    (None, 64)                      4160
## ________________________________________________________________________________
## dropout_3 (Dropout)                 (None, 64)                      0
## ________________________________________________________________________________
## dense_16 (Dense)                    (None, 1)                       65
## ================================================================================
## Total params: 68,289
## Trainable params: 68,289
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
model.test.4 <- model %>% evaluate(x_test, y_test)
history#Training
```

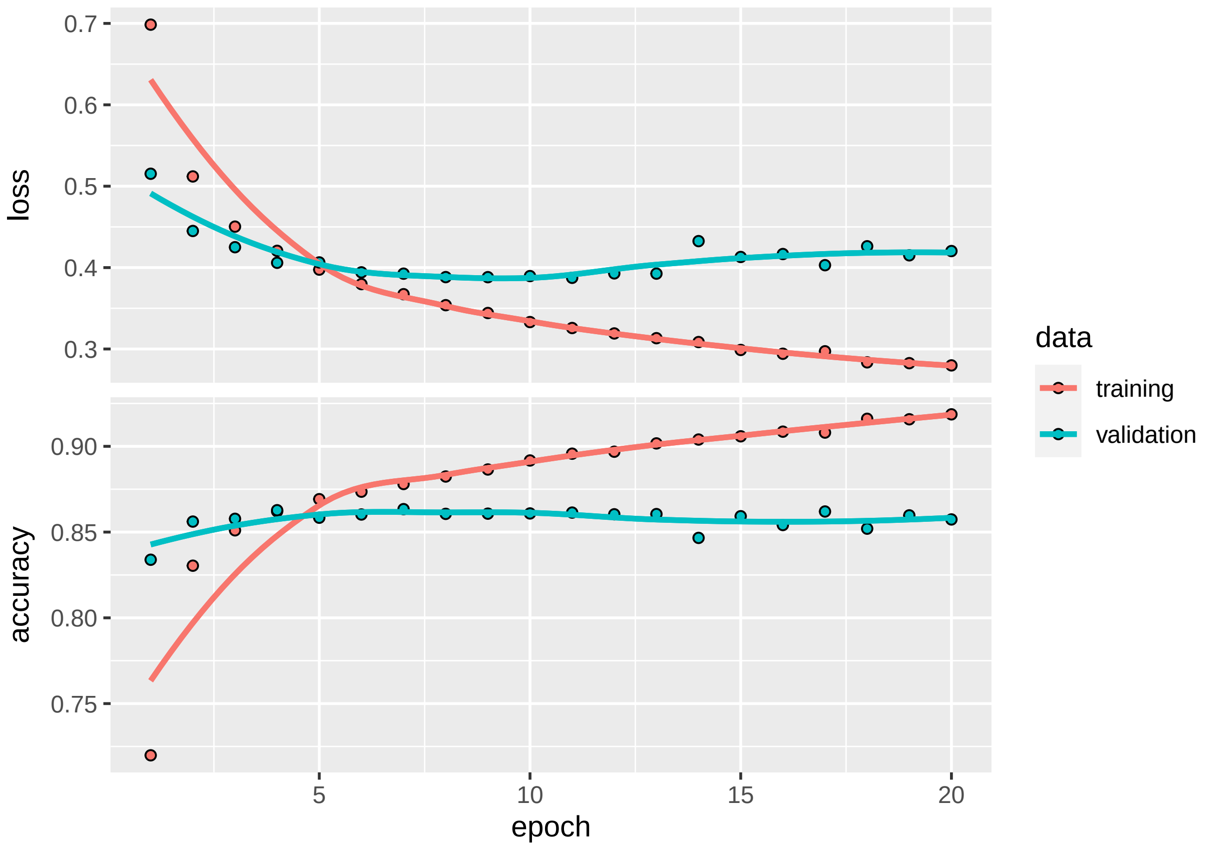
```
##
## Final epoch (plot to see history):
##         loss: 0.2799
##     accuracy: 0.9187
##     val_loss: 0.4203
## val_accuracy: 0.8574
```

```
model.test.4 #Testing
```

```
##      loss  accuracy
## 0.4063964 0.8605000
```

As seen above, the training loss has increased and the training accuracy has increased when compared to the previous model (*reduced architecture + dropout*). The testing accuracy has increased and the testing loss has decreased further than the previous model, and therefore this is accepted as the new baseline. The models training and validation accuracy and loss graphs are displayed below. The models can still be seen to overfit, and the gradient of the validation loss can be seen to decrease in the loss-vs-epoch graph. This is a good indication that the regularization is working.

```
plot(history, method =  "ggplot2") + geom_smooth(se = F, method = 'loess')
```



As a fourth and final regularization technique, I considered applying early stopping to the current baseline. A patience parameter of 2 was chosen, meaning the validation loss would wait for 2 epochs before terminating the training of the deep network.

```
#5. Early stop + Dropout + Reduced + L2_Reg
model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu", input_shape = c(1000),
                kernel_regularizer = regularizer_l2(l = 0.001)) %>%
      layer_dropout(rate = 0.5) %>%
         layer_dense(units = 64, activation = "relu",
                     kernel_regularizer = regularizer_l2(l = 0.001)) %>%
            layer_dropout(rate = 0.5) %>%
              layer_dense(units = 1, activation = "sigmoid")

model %>% compile(optimizer = "rmsprop",loss = "binary_crossentropy", metrics = "accuracy")
history <- model %>% fit(x_train, y_train, epochs = 20, batch_size = 512,
                         validation_data = list(x_valid, y_valid), callback =
                           callback_early_stopping(monitor = "val_loss", patience
                                                   = 1))
model %>% summary()
```

```
## Model: "sequential_5"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                        Param #
## ================================================================================
## dense_17 (Dense)                    (None, 64)                      64064
## ________________________________________________________________________________
## dropout_4 (Dropout)                 (None, 64)                      0
## ________________________________________________________________________________
## dense_18 (Dense)                    (None, 64)                      4160
## ________________________________________________________________________________
## dropout_5 (Dropout)                 (None, 64)                      0
## ________________________________________________________________________________
## dense_19 (Dense)                    (None, 1)                       65
## ================================================================================
## Total params: 68,289
## Trainable params: 68,289
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
model.test.5 <- model %>% evaluate(x_test, y_test)
history#Training
```

```
##
## Final epoch (plot to see history):
##         loss: 0.354
##     accuracy: 0.8835
##     val_loss: 0.4199
## val_accuracy: 0.8419
```

```
model.test.5 #Testing
```

```
##      loss  accuracy
## 0.4112824 0.8464000
```

The training loss and accuracy are displayed above and it can be seen to have better testing performance than the previous baseline *(reduced loss, increased accuracy)*. The performance metrics of this final model are displayed on the following page.

- Resources
  - Deep Learning with R, ISBN 9781617295546
  - Class notes and examples

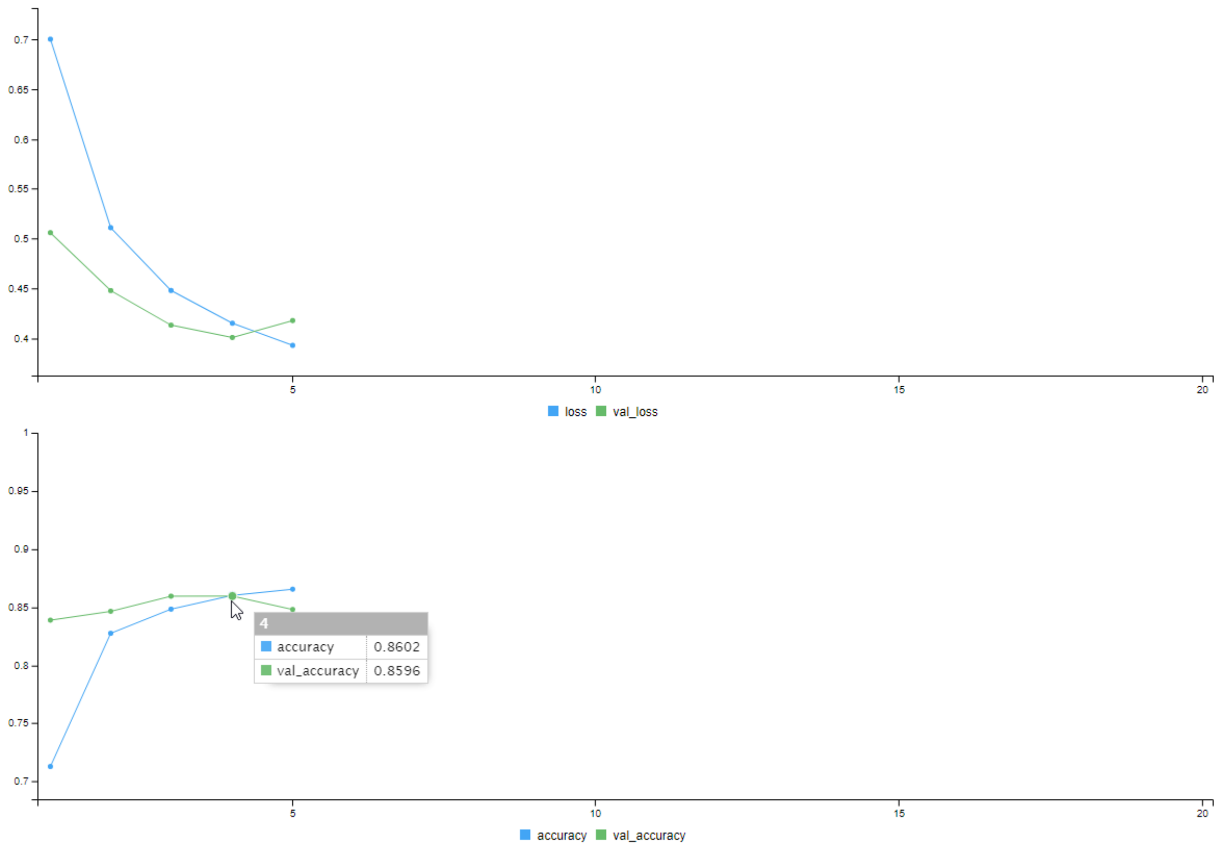


Figure 1: Final Model