Machine Learning 2ST129 26605 HT2023 Assignment 5

Anonymous Student

December 10, 2023

Contents

General Information

- Time used for reading: 2 hours
- Time used for basic assignment 15 hours:
- Time used for extra assignment 6 hours:
- Good with lab: Its good that you had the implement the steps one at a time, which helps with learning how the models works.
- Things improve with lab: Since there is a newer version of Deep Learning with R, it would be nice if there were reading instructions for that version as well.

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

1 Task 1

1.1 1.1

First we start computing the query, key and value matrices for one attention head by implementing it in a function `qkv()`

```r
data("transformer_example")
 qkv <- function(X, Wq, Wk, Wv){
   Q<- X %*% Wq
   K <- X %*% Wk
   V <- X %*% Wv
   return(list(Q=Q, K=K, V=V))
 }


Wq <- transformer_example$Wq[,, 1]
 Wk <- transformer_example$Wk[,, 1]
 Wv <- transformer_example$Wv[,, 1]
X <- transformer_example$embeddings[1: 3,]

res <- qkv(X, Wq, Wk, Wv)
```

1.2 1.2

Now, based on the query, key and value, we want to compute the attention of that given attention head for the three chosen tokens. This procedures involes multiple steps.

```r
softmax <- function(X){
  value <- exp(X) / rowSums(exp(X))
  return(value)
  }
attention <- function(Q, K, V) {
  key_dim <- ncol(K)
  score_ratio <- (Q%*% t(K)) / sqrt(key_dim)

  attention <- softmax(score_ratio)


  Z <- attention %*% V

  results <- list(Z = Z, attention = attention)
  return(results)
}

attention(res$Q, res$K, res$V)
```

```
$Z
              [,1]          [,2]         [,3]
the    0.012395453 -0.0212420459  0.009404870
quick -0.003759269 -0.0008360029 -0.005108890
```

```
brown  0.002412222 -0.0088974612  0.001147999

$attention
            the     quick     brown
the   0.3601932 0.3080896 0.3317172
quick 0.3088780 0.3582373 0.3328847
brown 0.3300375 0.3360583 0.3339042
```

1.3 1.3

The resulting vector is one we can send along to the feed-forward neural network. The second row of the attention matrix, are the attention scores assigned to the second token, or quick in this case, with respect to with respect to all other tokens, including itself. For example, in the second row, the second element is the highest, which is the attention score of quick attending to itself.

1.4 1.4

Now we will implement a multi-head attention layer.

```
compute_attention <- function(X, Wq, Wk, Wv){
  qkv_res <- qkv(X, Wq, Wk, Wv)
  attention <- attention(qkv_res$Q, qkv_res$K, qkv_res$V)
  return(attention)
}

multi_head_self_attention <- function(X,Q, K , V, W0){
  n_weights <- dim(K)[3]
   attention_heads <- sapply(1:n_weights, FUN=function(i){
   compute_attention(X, Q[,,i], K[,,i], V[,,i])
  }, simplify = FALSE)
  combined_matrix<- do.call(cbind, lapply(attention_heads, FUN = function(sub_list){
     sub_list$Z
  }))
  results <- combined_matrix %*% W0
  return(results)
}
```

```
#on example data
multi_head_self_attention(X,transformer_example$Wq, transformer_example$Wk,
                                transformer_example$Wv, transformer_example$W0)
```

```
              [,1]          [,2]         [,3]
the   -0.014189613 -0.0040299008 -0.006756286
quick -0.009963516 -0.0010724342 -0.001996524
brown -0.006394562 -0.0006626115 -0.002219108
```

```
#On whole data
X <- transformer_example$embeddings
multi_head_self_attention(X,transformer_example$Wq, transformer_example$Wk,
                                transformer_example$Wv, transformer_example$W0)
```

```
             [,1]        [,2]       [,3]
the   -0.02705494 0.005860020 0.01218247
quick -0.02413116 0.008522431 0.02030744
brown -0.02158066 0.009040085 0.02021518
fox   -0.02744489 0.004881125 0.01531574
jumps -0.02348063 0.008342224 0.01498752
over  -0.02592150 0.006371230 0.01826652
the   -0.02705494 0.005860020 0.01218247
lazy  -0.02755128 0.006333208 0.02246126
```

```
dog   -0.02986621 0.005885128 0.01774406
```

2 Task 2

2.1 Task 2.1

Now we are going to implement a one-layer recurrent neural network based on the rnn_example data. First we implement a RNN linear unit.

```
X <- rnn_example$embeddings[1,, drop=FALSE]
X
```

```
        [,1]    [,2]     [,3]       [,4]       [,5]
the 1.819735 1.37044 1.196318 -0.4068792 -0.6109788
```

```
hidden_dim <- 4

h_t_minus_one <- matrix(0, nrow = hidden_dim, ncol = 1)

rnn_unit <- function(W,U,b, h_t_minus_one, x_t){
  at <- b + W %*% h_t_minus_one + U %*% t(x_t)

  return(at)
}

a_t <- rnn_unit(h_t_minus_one, x_t=X, W = rnn_example$W, U = rnn_example$U, b = rnn_example$b)
a_t
```

```
            the
[1,]  0.5819145
[2,] -2.2686535
[3,] -0.6410312
[4,]  1.4891931
```

2.2 2.2

Now we implement the `tanh()` activation function

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$

```
activation <- function(x){
  return((exp(x) - exp(-x)) / (exp(x) + exp(-x)))
}

h_t <- activation(a_t)

h_t
```

```
            the
[1,]  0.5240555
[2,] -0.9788223
[3,] -0.5656013
[4,]  0.9031762
```

2.3 2.3

Now to implement the output function and the softmax function.

```r
output_rnn<- function(h_t, V, c){
  return(c + V %*% h_t)
}
softmax <- function(o){
  return(exp(o) / sum(exp(o)))
}


softmax(output_rnn(h_t, rnn_example$V, rnn_example$c) )
```

```
           the
[1,] 0.3063613
[2,] 0.2930885
[3,] 0.4005502
```

2.4 2.4

Now we implement the full recurrent layer.

```r
#' combines all the previous functions in one function
rnn_combined <- function(X, W, V, U, b, c, hidden_dim = 4, h_t_minus_one) {
  a_t <- rnn_unit(h_t_minus_one = h_t_minus_one, W = W, U = U, b = b, x_t = X)
  ht <- activation(a_t)
  o <- output_rnn(ht, V, c)

  y_hat <- softmax(o)
  return(list(ht = ht, y_hat = y_hat))
}


#Assuming the words are the rows of X
rnn_layer <- function(X, W, V, U, b, c, hidden_dim = 4){
  n_rows <- nrow(X)
  h_t_minus_one <- matrix(0, nrow = hidden_dim, ncol = 1)
  h_t <- matrix(NA, ncol=hidden_dim, nrow=n_rows)
  y_hat <- matrix(NA, ncol=nrow(V), nrow=n_rows)
  for(i in 1:n_rows){
    X_subset <- X[i,, drop=FALSE]
    iter_res <- rnn_combined(X_subset, W, V, U, b, c, hidden_dim, h_t_minus_one)
    h_t_minus_one <- h_t[i,] <- iter_res$ht
    y_hat[i,] <- iter_res$y_hat
  }
  rownames(h_t) <- rownames(y_hat) <- rownames(X)
  list <- list(ht=h_t, y_hat = y_hat)
  return(list)
}

#checking that it works for both cases

rnn_layer(X, W = rnn_example$W, V = rnn_example$V,
          U = rnn_example$U, rnn_example$b,c=  rnn_example$c)
```

```
$ht
         [,1]       [,2]       [,3]      [,4]
the 0.5240555 -0.9788223 -0.5656013 0.9031762

$y_hat
         [,1]      [,2]      [,3]
the 0.3063613 0.2930885 0.4005502
```

```
X_new <- rnn_example$embeddings[1: 3,, drop=FALSE]

rnn_layer(X_new, W = rnn_example$W, V = rnn_example$V,
          U = rnn_example$U, rnn_example$b,c=  rnn_example$c)

$ht
             [,1]        [,2]       [,3]       [,4]
the    0.52405551 -0.97882227 -0.5656013  0.9031762
quick -0.05951368  0.03988226  0.8241800 -0.6562744
brown -0.08984008  0.92822217 -0.1563247 -0.6657626

$y_hat
           [,1]      [,2]      [,3]
the   0.3063613 0.2930885 0.4005502
quick 0.2838013 0.3490452 0.3671536
brown 0.2878002 0.3666877 0.3455121
```

2.5 2.5

Now we apply the function on the whole data and get the value of the hidden state for the token dog.

```
results <- rnn_layer(X= rnn_example$embeddings, W = rnn_example$W, V = rnn_example$V,
          U = rnn_example$U, rnn_example$b,c=  rnn_example$c)

results

$ht
             [,1]        [,2]       [,3]        [,4]
the    0.52405551 -0.97882227 -0.5656013  0.90317617
quick -0.05951368  0.03988226  0.8241800 -0.65627443
brown -0.08984008  0.92822217 -0.1563247 -0.66576260
fox   -0.43750124 -0.38442420  0.6070560  0.64156809
jumps  0.54934775 -0.24515537 -0.8748765  0.76466178
over  -0.62799359  0.82990292  0.6111049 -0.07918235
the    0.60957979 -0.97665249 -0.5596029  0.92482999
lazy  -0.92042949  0.90472482  0.9904783  0.03534373
dog   -0.29284645  0.18138447 -0.2190118  0.25923968

$y_hat
           [,1]      [,2]      [,3]
the   0.3063613 0.2930885 0.4005502
quick 0.2838013 0.3490452 0.3671536
brown 0.2878002 0.3666877 0.3455121
fox   0.2968252 0.3386681 0.3645066
jumps 0.3098598 0.3095035 0.3806367
over  0.2896910 0.3795378 0.3307712
the   0.3075896 0.2913470 0.4010634
lazy  0.2880155 0.3908604 0.3211241
dog   0.2956745 0.3461530 0.3581725

results$ht["dog", , drop = FALSE]

          [,1]      [,2]       [,3]      [,4]
dog -0.2928465 0.1813845 -0.2190118 0.2592397
```

3 Task 3

3.1 3.1

First we load the data.

```
imdb <- dataset_imdb(num_words = 10000) # we consider 10000 words as featers
maxlen <- 20 #cut off the text after 20 words
c(c(input_train, y_train), c(input_test, y_test)) %<-% imdb
input_train <- pad_sequences(input_train, maxlen = maxlen) #load data as list of integers
input_test <- pad_sequences(input_test, maxlen = maxlen) #turn the list of integers into a 2D intege
```

Now to create a neural network using the following:

- Emedding layer with 16 dimension
- Hidden layer with 32 hidden stats
- Validation split of 20 percent
- batch size of 128
- rmsprop and binary cross-entropy

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 16,
                  input_length = maxlen) %>%
  layer_flatten() %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

```
Model: "sequential"
_________________________________________________________________________________
 Layer (type)                        Output Shape                      Param #
=================================================================================
 embedding (Embedding)               (None, 20, 16)                    160000
 flatten (Flatten)                   (None, 320)                       0
 dense_1 (Dense)                     (None, 32)                        10272
 dense (Dense)                       (None, 1)                         33
=================================================================================
Total params: 170305 (665.25 KB)
Trainable params: 170305 (665.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________________________
```

```
  model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)



history <- model %>% fit(
  input_train, y_train,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.2
)
```

The result for the last epoch is:

```
Epoch 10/10
157/157 [==============================] - 1s 3ms/step
- loss: 0.0532 - acc: 0.9866 - val_loss: 0.9352 - val_acc: 0.7142
```

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

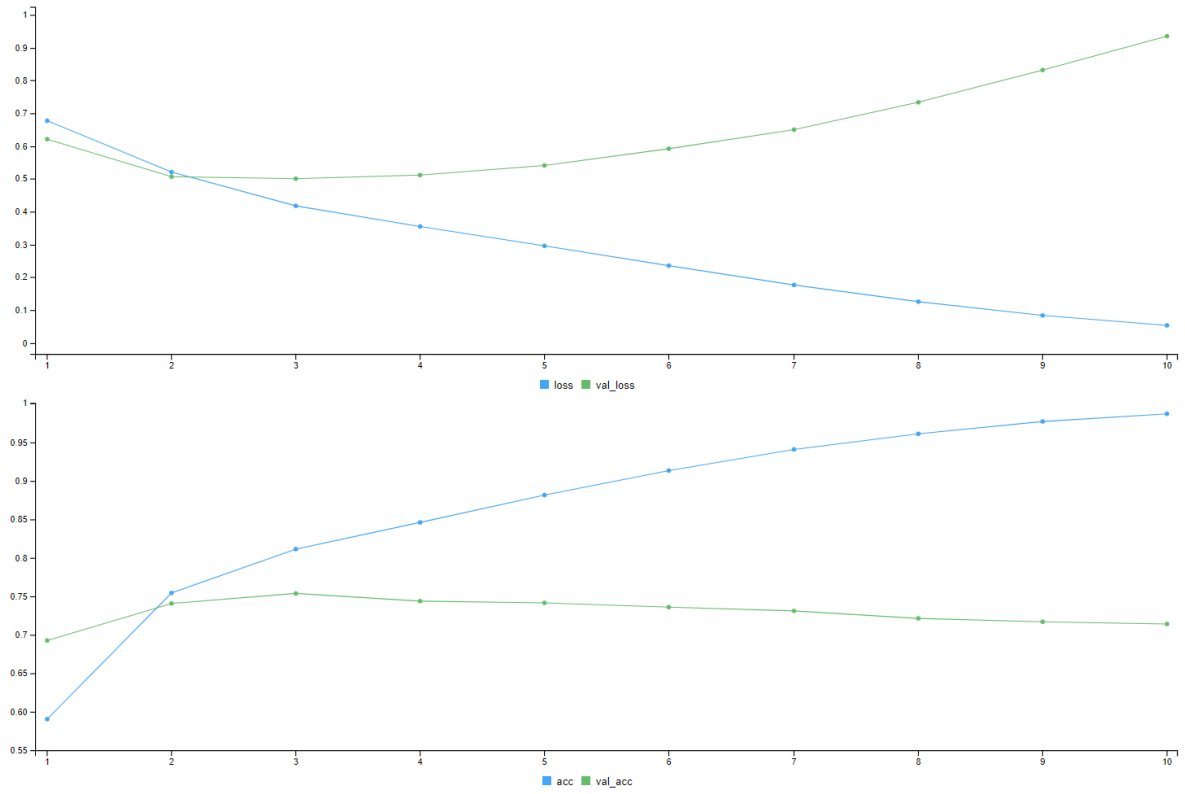


Figure 1: Results for the first model

As can be seen from the results, the validation accuracy for the last epoch is 0.7142.

3.2 3.2

The embedding layer uses two arguments, the number of possible tokens, which in this case equals to 10000 by the input_dim, and the dimension of the embedding as 16, as given by the output_dim. It can be seen as a dictionary such that it the indices for the integers are mapped to dense vectors. During its training process, it learns to associate each token or word with a specific vector. Then it works by using the input integers and searching for these integers in its internal dictionary and then return the corresponding vectors.

3.3 3.3

Now to setup a simple RNN with 32 hidden units based on the same embedding layer.

```
library(keras)

model2 <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 16, input_length = maxlen) %>%
  layer_simple_rnn(units = 32, activation = "tanh") %>%  # RNN layer with 32 hidden units
  layer_dense(units = 1, activation = "sigmoid")

summary(model2)
```

```
Model: "sequential_1"
________________________________________________________________________________
 Layer (type)                       Output Shape                    Param #
```

```
================================================================================
 embedding_1 (Embedding)            (None, 20, 16)                160000
 simple_rnn (SimpleRNN)             (None, 32)                    1568
 dense_2 (Dense)                    (None, 1)                     33
================================================================================
Total params: 161601 (631.25 KB)
Trainable params: 161601 (631.25 KB)
Non-trainable params: 0 (0.00 Byte)
--------------------------------------------------------------------------------
```

```
model2 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)

history2 <- model2 %>% fit(
  input_train, y_train,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.2
)
```

Now the results are:

```
Epoch 10/10
157/157 [==============================] - 1s 5ms/step
- loss: 0.0594 - acc: 0.9823 - val_loss: 1.0662 - val_acc: 0.6924
```

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

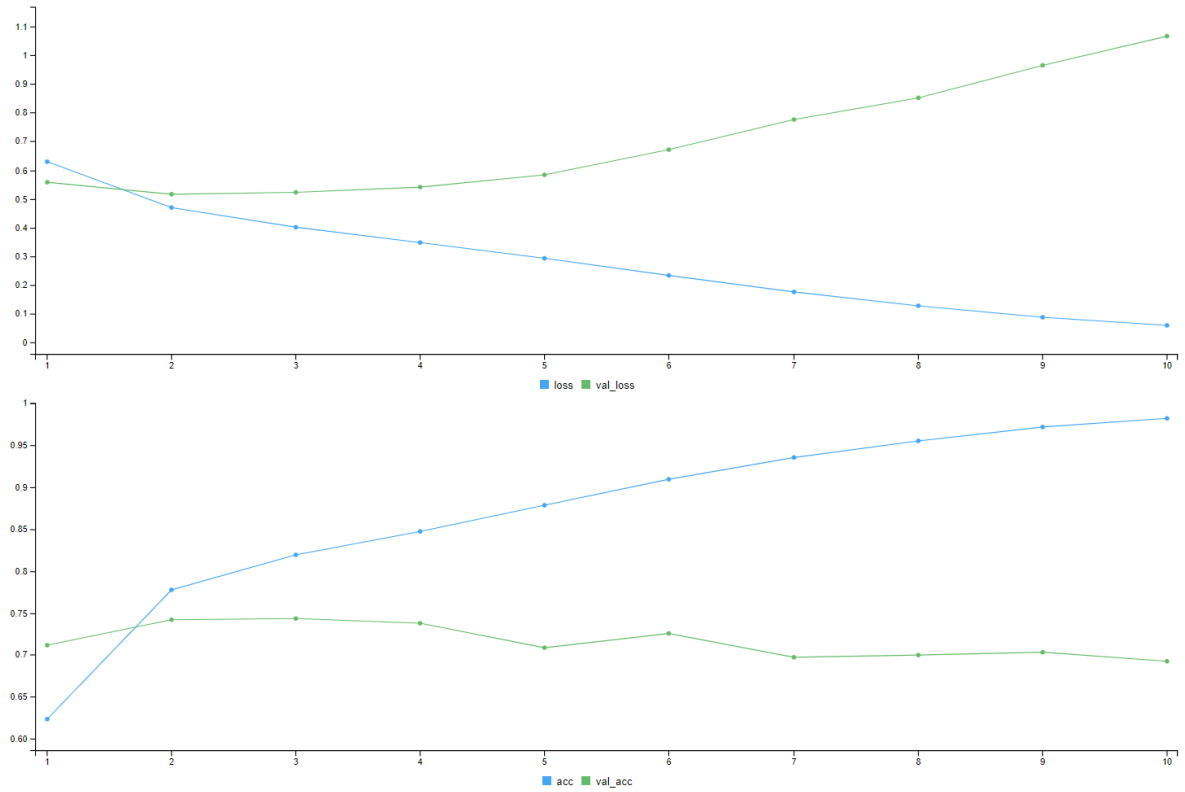


Figure 2: Results for RNN model

Now the validation accuracy is 0.6924. Hence this small rnn does not perform any better. Potentially it could be to the relatively small input length.

3.4 3.4

The model implemented is best described by Fig. 10.5 in Goodfellow et al. The reason is that this model corresponds to a time/unfolded recurrent network with a single output at the end of the sequence, while also having recurrent connections among the hidden units. This is in contrast to the figure in 10.3 where the recurrent networks have recurrent connections among the hidden units, while producing an output at each time point t. For example, in this model we have a dense layer with one unit, which means that the rnn gives a single output in the end.

3.5 3.5

The matrix $\mathbf{U}$ in a RNN is a weight matrix representing the hidden connections from the input to the hidden state. The individual elements in the matrix, U_{i_j} are the weights of the connection from a specific input feature j to the i:th hidden unit. The matrix $\mathbf{W}$ is a weight matrix for hidden-to-hidden connections. Here, each element $W_{i,j}$ represents the weight of the connection from the j:th hidden unit at the previous time time periods to the i:th hidden unit for the current time period. Hence $\mathbf{U}$ is related to the influence of the current input on the hidden state, whereas $\mathbf{W}$ is related to the influence of the previous hidden stat on the current hidden state.

In the model above, these wight matrices are not directly viewable. However, they exist within the embedding layer. The total number of parameters in this case is the inut_dim $\times$ output_dim which equals to $10000 \times 16 = 160000$ for U.

In the simple RNN layer `layer_simple_rnn()` we have $32 \times 32 + 16 \times 32 + 32 = 1568$ parameters, which contains both U and W.

3.6 3.6

The parameters V represents hidden-to-output connections, or the connections from the recurrent layer to the dense layer, which computes the final output. The number of parameters here is dependent on the number of units ht the simple RNN layer as well as the number of units in the dense layer. Here we have that the number of parameters equals to $32 \times 1 + 1 = 33$where 32 is the number of units in the simple RNN layer and 1 is the number of units the the dense layer, as well as an additional 1 for the bias.

3.7 3.7

New to implement a Long Short-Term Memory (LSTM) layer with 32 hidden units.

```
LSTM_model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 16) %>%
  layer_lstm(units = 32) %>%
  layer_dense(units = 1, activation = "sigmoid")
summary(LSTM_model)
```

```
Model: "sequential_2"
_________________________________________________________________________
 Layer (type)                        Output Shape                    Param #
=========================================================================
 embedding_2 (Embedding)             (None, None, 16)                160000
 lstm (LSTM)                         (None, 32)                      6272
 dense_3 (Dense)                     (None, 1)                       33
=========================================================================
Total params: 166305 (649.63 KB)
Trainable params: 166305 (649.63 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________________
```

```
LSTM_model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)
history_LSTM <- LSTM_model %>% fit(
  input_train, y_train,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.2
)
```

The results for the last epoch are:

```
Epoch 10/10
157/157 [==============================] - 2s 12ms/step
- loss: 0.2800 - acc: 0.8872 - val_loss: 0.5823 - val_acc: 0.7452
```

The validation accuracy is 0.7452. And the history of the results across all epochs:

```
knitr::include_graphics("LSTM_model1.png")
```

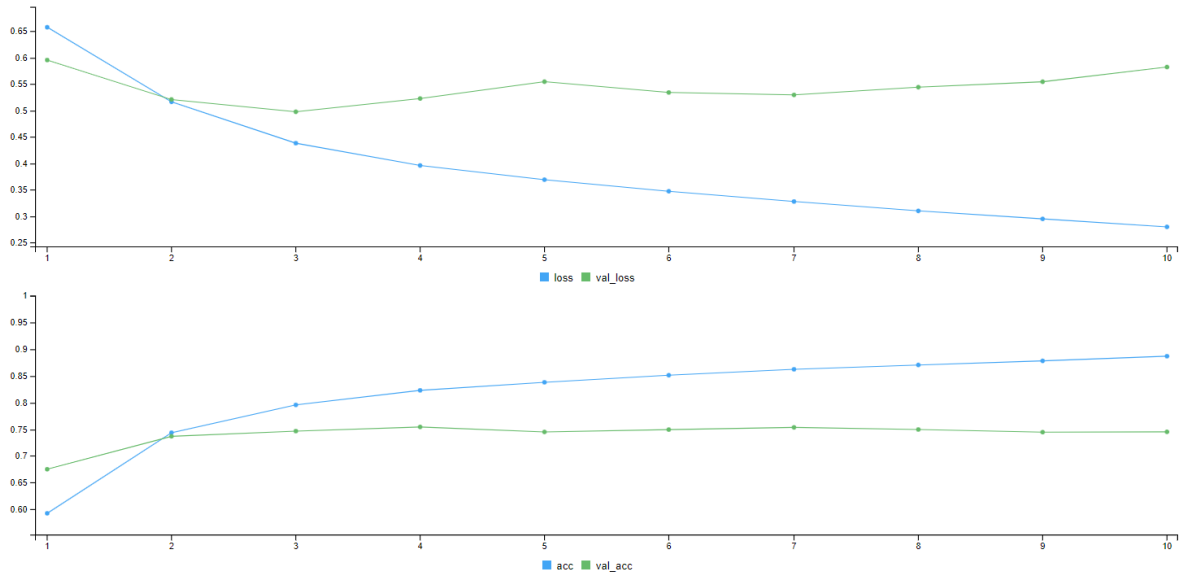


Figure 3: Results for first LSTM model

4 3.8

First we increase the number of words used to considers as features to 100 instead of 20. Then we also add an additional lstm layer to increase the capacity of the network. This requires the intermediate layer to return their full sequence of outputs, hence we have to specify `return_sequences=TRUE`. Then we also add dropout to help against overfitting. Hence we set to `dropout` argument to 0.2, which is specifying the dropout rate for the input units, as well as the `recurrent_dropout`, which corresponds to the dropout rate for the recurrent units. Lastly we also increase the number of epochs used, both because we made new changes in general and perhaps it will take longer to converge due to adding new layers and increasing the number of words. But also because networks regularized with dropout always take longer to converge.

```
maxlen <- 100 #cut off the text after 20 words
c(c(input_train, y_train), c(input_test, y_test)) %<-% imdb
```

```
input_train <- pad_sequences(input_train, maxlen = maxlen) #load data as list of integers
input_test <- pad_sequences(input_test, maxlen = maxlen) #turn the list of integers into a 2D intege

LSTM_model2 <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 16) %>%
  layer_lstm(units = 32, dropout = 0.2, recurrent_dropout = 0.2, return_sequences = TRUE) %>%
  layer_lstm(units = 32, dropout = 0.2, recurrent_dropout = 0.2) %>%
  layer_dense(units = 1, activation = "sigmoid")
summary(LSTM_model2)
```

```
Model: "sequential_3"
_________________________________________________________________________________
 Layer (type)                        Output Shape                      Param #
=================================================================================
 embedding_3 (Embedding)             (None, None, 16)                  160000
 lstm_2 (LSTM)                       (None, None, 32)                  6272
 lstm_1 (LSTM)                       (None, 32)                        8320
 dense_4 (Dense)                     (None, 1)                         33
=================================================================================
Total params: 174625 (682.13 KB)
Trainable params: 174625 (682.13 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________________________
```

```
LSTM_model2 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)
history_LSTM2 <- LSTM_model2 %>% fit(
  input_train, y_train,
  epochs = 20,
  batch_size = 128,
  validation_split = 0.2)
```

And the results are:

```
Epoch 20/20
157/157 [==============================] - 21s 136ms/step
- loss: 0.1003 - acc: 0.9677 - val_loss: 0.6459 - val_acc: 0.8116
```

```
knitr::include_graphics("lstm_model_2.png")
```

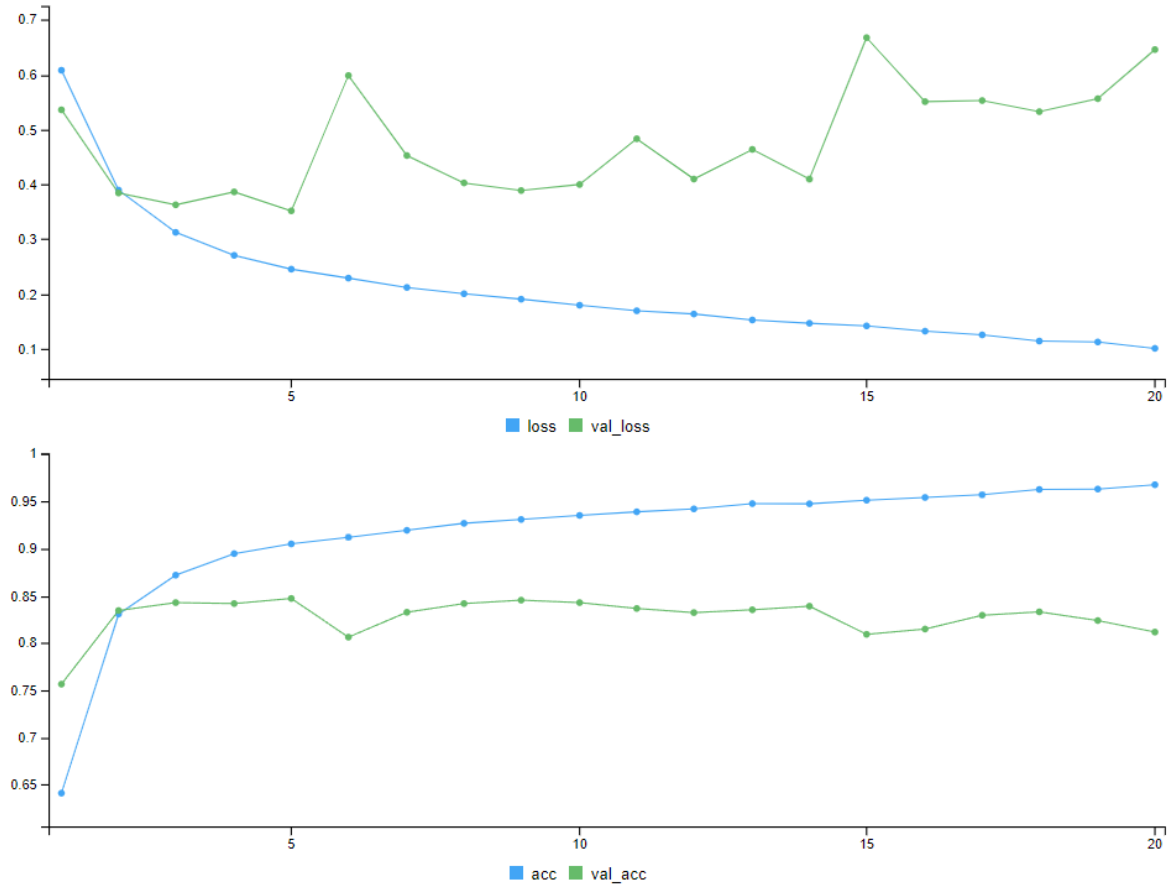


Figure 4: Results for second LSTM model

Looking at the results, then we see that the validation accuracy for the last epoch is 0.81, and looking at 4 it can be seen that this is were the accuracy seem to stabilize around already after a few epochs.

Now we can try some more minor adjustments to see if we can gain some additional accuracy. we can increase the number of words used to 500. Since this also increases the computational power required, then I will also increase the batch size to 256. Also, we can fine tune the learning rate to decrease after 5 epochs. The reason is that an improper learning rate will result in a model with low effective capacity due to optimization problems. The reason for choosing 5 is since the previous models seemed to have converged around that point.

```
maxlen <- 200 #cut off the text after 20 words
c(c(input_train, y_train), c(input_test, y_test)) %<-% imdb
input_train <- pad_sequences(input_train, maxlen = maxlen) #load data as list of integers
input_test <- pad_sequences(input_test, maxlen = maxlen) #turn the list of integers into a 2D intege

LSTM_model3 <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 16) %>%
  layer_lstm(units = 32, dropout = 0.2, recurrent_dropout = 0.2, return_sequences = TRUE) %>%
  layer_lstm(units = 32, dropout = 0.2, recurrent_dropout = 0.2) %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(LSTM_model3)
```

```
Model: "sequential_4"
_________________________________________________________________________________
 Layer (type)                        Output Shape                      Param #
=================================================================================
```

```
 embedding_4 (Embedding)             (None, None, 16)                160000
 lstm_4 (LSTM)                       (None, None, 32)                6272
 lstm_3 (LSTM)                       (None, 32)                      8320
 dense_5 (Dense)                     (None, 1)                       33
================================================================================
Total params: 174625 (682.13 KB)
Trainable params: 174625 (682.13 KB)
Non-trainable params: 0 (0.00 Byte)
--------------------------------------------------------------------------------
```

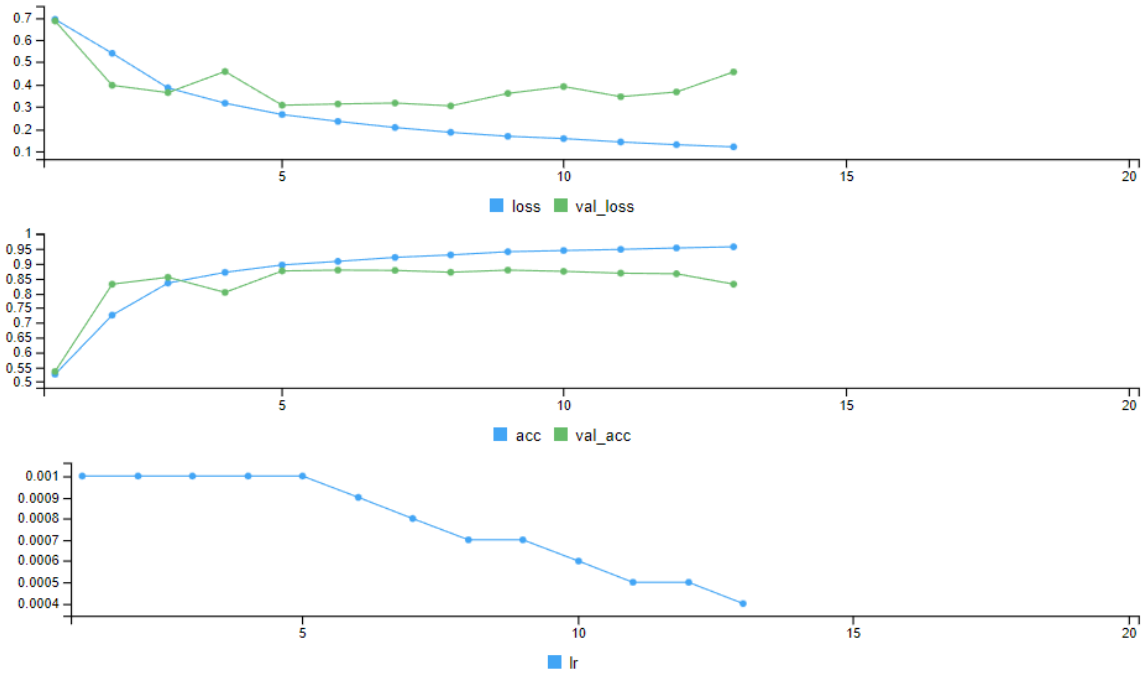
```
LSTM_model3 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)

Scheduler <- function(epoch, lr) {
  if (epoch < 5) {
    return(lr)
  } else {
    return(lr * exp(-0.1))
  }
}

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

history_LSTM3 <- LSTM_model3 %>%
  fit(
  input_train, y_train,
  epochs = 20,
  batch_size = 256,
  validation_split = 0.2,
    shuffle = TRUE,
  callbacks =callback_list)
```

This ended early due to early stopping. The state with the best results before ending it was:

```
79/79 [==============================] -
  51s 648ms/step - loss: 0.1860 - acc: 0.9304 - val_loss: 0.3046 - val_acc: 0.8716 - lr: 7.4082e-04
```

Here we see that the validation accuracy has increased to 0.87. The same as before, we can also plot the entire history.

```
knitr::include_graphics("lstm_model_3.png")
```



Now lastly, we can run this best-performing model on the test set to make check if we have over fitted on the validation set.

```
evaluation <- LSTM_model3 %>% evaluate(input_test, y_test)
```

```
782/782 [==============================] - 16s 20ms/step - loss: 0.4716 - acc: 0.8257
```

Looking at the results, then the accuracy for the test set was 0.825. Hence this could indicate that the previous model may have overfitted to a certain degree, since this accuracy is lower.