# Assigment 1 M3

Simon

17/11/2021

```
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.0.2
```

```
## Warning: package 'ggplot2' was built under R version 4.0.2
```

```
## Warning: package 'tibble' was built under R version 4.0.2
```

```
## Warning: package 'tidyr' was built under R version 4.0.2
```

```
## Warning: package 'dplyr' was built under R version 4.0.2
```

```
library(magrittr)
library(skimr)
```

```
## Warning: package 'skimr' was built under R version 4.0.2
```

```
data = read_csv(
  "https://github.com/SDS-AAU/SDS-master/raw/master/M3/assignments/find_elon.gz")
```

The data's column names are changed so 0 becomes text and 1 becomes y. Text being the tweet and y the label.

```
data= data %>%
  rename(text= "0", y="1")
```

We start by creating test and training data

```
library(rsample)
```

```
## Warning: package 'rsample' was built under R version 4.0.2
```

```
split= initial_split(data, prop = 0.75)

train_data= training(split)
test_data= testing(split)
```

Then we extract our x and y training and test data to be used in the neural networks.

```
x_train_data= train_data %>% pull(text)
y_train_data= train_data %>% pull(y)

x_test_data= test_data %>% pull(text)
y_test_data= test_data %>% pull(y)
```

Now it is time to load keras and make some adjustments to the data. The data are tweets so a lot of special characters are removed like # and @'s. And then we tokenize our data as we know from basic machine learning to get like a bag of words from our tweets and lastly we create a list where every tweet has a vector

which includes the words as a numerical charchter if the words contained in the tweets are among the 10000 most used words in the dataset.

```
library(keras)
```

```
## Warning: package 'keras' was built under R version 4.0.2
```

```
#for training data
tokenizer_train <- text_tokenizer(num_words = 10000,
                                  filters = "!\"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n" ) %>%
  fit_text_tokenizer(x_train_data)

sequences_train = texts_to_sequences(tokenizer_train, x_train_data)


#For test data

tokenizer_test <- text_tokenizer(num_words = 10000,
                                 filters = "!\"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n" ) %>%
  fit_text_tokenizer(x_test_data)

sequences_test = texts_to_sequences(tokenizer_test, x_test_data)
```

# Baseline model

## One-hot encoding

we use this function Daniel made to vectorize the sequences :)

```
vectorize_sequences <- function(sequences, dimension) {
  results <- matrix(0, nrow = length(sequences), ncol = dimension)
  for(i in 1:length(sequences)){
    results[i, sequences[[i]]] <- 1
  }
  return(results)
}
```

we use it on the training and test data

```
x_train <- sequences_train %>% vectorize_sequences(dimension = 10000)
x_test <- sequences_test %>% vectorize_sequences(dimension = 10000)

str(x_train[1,])
```

```
##  num [1:10000] 0 0 0 0 1 1 0 0 0 1 ...
```

What the above has done to the data is, that every tweet now is a row and every feature/word now is a column and then if the tweets has e.g. word 1 then it would have the value 1 otherwise zero. So we basically now have a matrix of size [2365x10000] [number of tweets in training set x number of words].

## The model

The above data is then used in our baseline model with an input shape of 10000 because that is the size of our input. Then we run it through two dense "relu" layers which is normal procedure for a baseline model. Lastly we have a dense layer with the output which is of unit 1 and is a "sigmoid" layer which means it returns a value between 0 and 1 as we want, as we wanna figure out if a tweet is fake or real.

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

We use baseline model compiling with optimizer “adam”, loss “binary” as we are dealing with a binary case and the metric we wanna maximize is accurary.

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

Here the structure of the model can be viewed

```
summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                         Output Shape                     Param #
## =========================================================================
## dense (Dense)                        (None, 16)                       160016
## _________________________________________________________________________
## dense_1 (Dense)                      (None, 16)                       272
## _________________________________________________________________________
## dense_2 (Dense)                      (None, 1)                        17
## =========================================================================
## Total params: 160,305
## Trainable params: 160,305
## Non-trainable params: 0
## _________________________________________________________________________
```

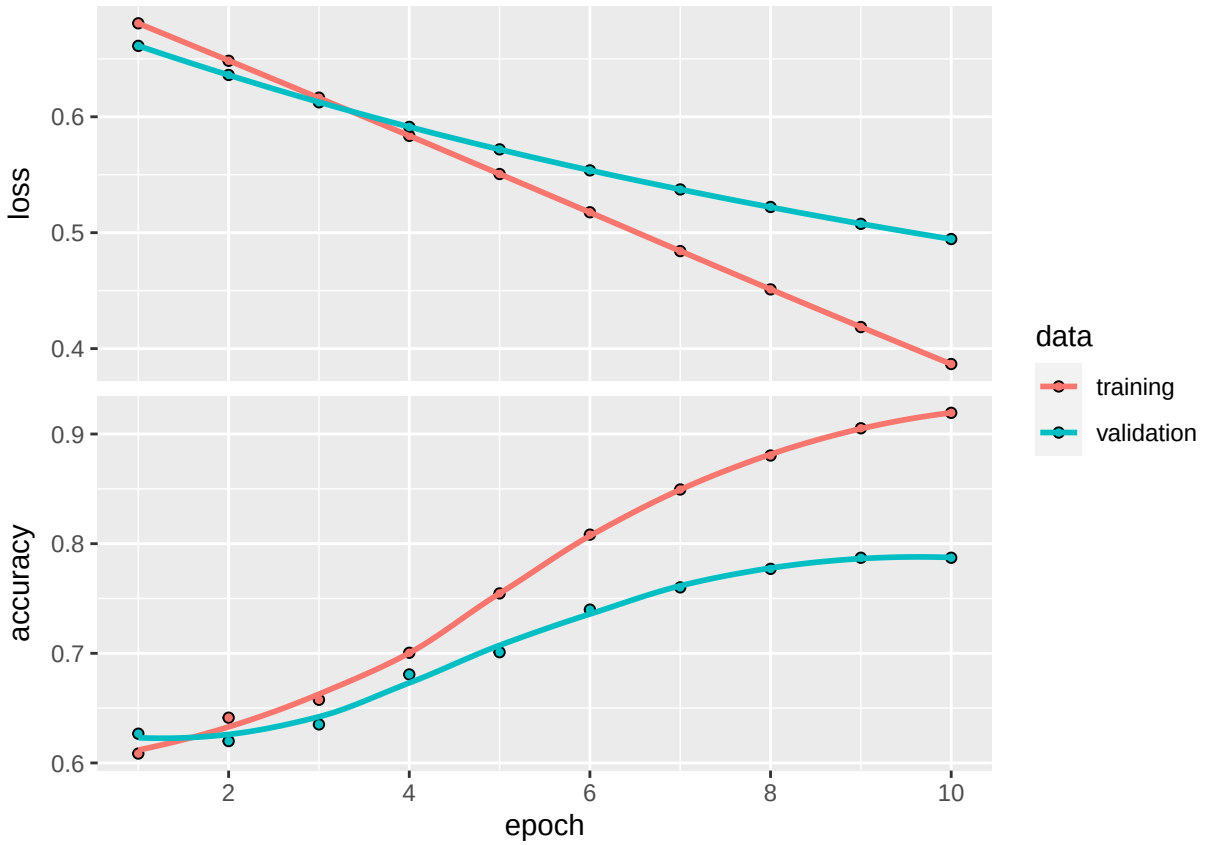
### Training data

And now the model is run 10 times with a batch size of 512

```
history_ann <- model %>% fit(
  x_train,
  y_train_data,
  epochs = 10,
  batch_size = 512,
  validation_split = 0.25
)
```

```
plot(history_ann)
```

```
## `geom_smooth()` using formula 'y ~ x'
```




Already after the first epochs we can see, that the validations set crosses our training set, which means our model is over fitted. After ten epochs the accuracy is kinda high at almost 90% and by traning the model again this should become even higher/better.

### Evaluation

```
history_ann
```

```
##
## Final epoch (plot to see history):
##         loss: 0.3867
##     accuracy: 0.9193
##     val_loss: 0.4945
## val_accuracy: 0.7872
```

Here we can see the metrics of the model in the final epoch.

Now we will try the data on a bit less naive model

## Rnn model with paded data

### Padding

In our first baseline model, we used a document-term matrix as inputs for training, with one-hot-encodings (= dummy variables) for the 10.000 most popular terms. This has a couple of disadvantages. Besides being a very large and sparse vector for every review, as a "bag-of-words", it did not take the word-order (sequence) into account.

This time, we use a different approach, therefore also need a different input data-structure. We now use

pad_sequences() to create a integer tensor of shape (samples, word_indices). However, tweets vary in length, which is a problem since Keras requieres the inputs to have the same shape across the whole sample. Therefore, we use the maxlen = 100 argument, to restrict ourselves to the first 100 words in every tweet also because a tweet maximally can be 280 characters.

The data is paded

```
x_train_pad <- sequences_train %>% pad_sequences(maxlen=100)
x_test_pad <- sequences_test %>% pad_sequences(maxlen=100)
```

```
glimpse(x_train_pad)
```

```
##  int [1:2365, 1:100] 0 0 0 0 0 0 0 0 0 0 ...
```

Now if the value in e.g. the first column of the first tweet is 0 it means that the first word in the first tweet is not one of the 10000 most used words and there for our model has no integer for it. If there is an integer e.g. "386" it means that that the 386 most commonly used word is the first word in the tweet.

### The model

setting up the model we will first use a layer_embedding to compress our initial one-hot-encoding vector of length 10.000 to a "meaning-vector" (=embedding) of the lower dimensionality of 32. Then we add a layer_simple_rnn on top, and finally a layer_dense for the binary prediction of review sentiment.

```
model_rnn <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 32) %>%
  layer_simple_rnn(units = 32, activation = "tanh") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Here the structure of the model can be seen

```
summary(model_rnn)
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## embedding (Embedding)               (None, None, 32)                320000
## ________________________________________________________________________________
## simple_rnn (SimpleRNN)              (None, 32)                      2080
## ________________________________________________________________________________
## dense_3 (Dense)                     (None, 1)                       33
## ================================================================================
## Total params: 322,113
## Trainable params: 322,113
## Non-trainable params: 0
## ________________________________________________________________________________
```

Again we use a basic setup for binary prediction.

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

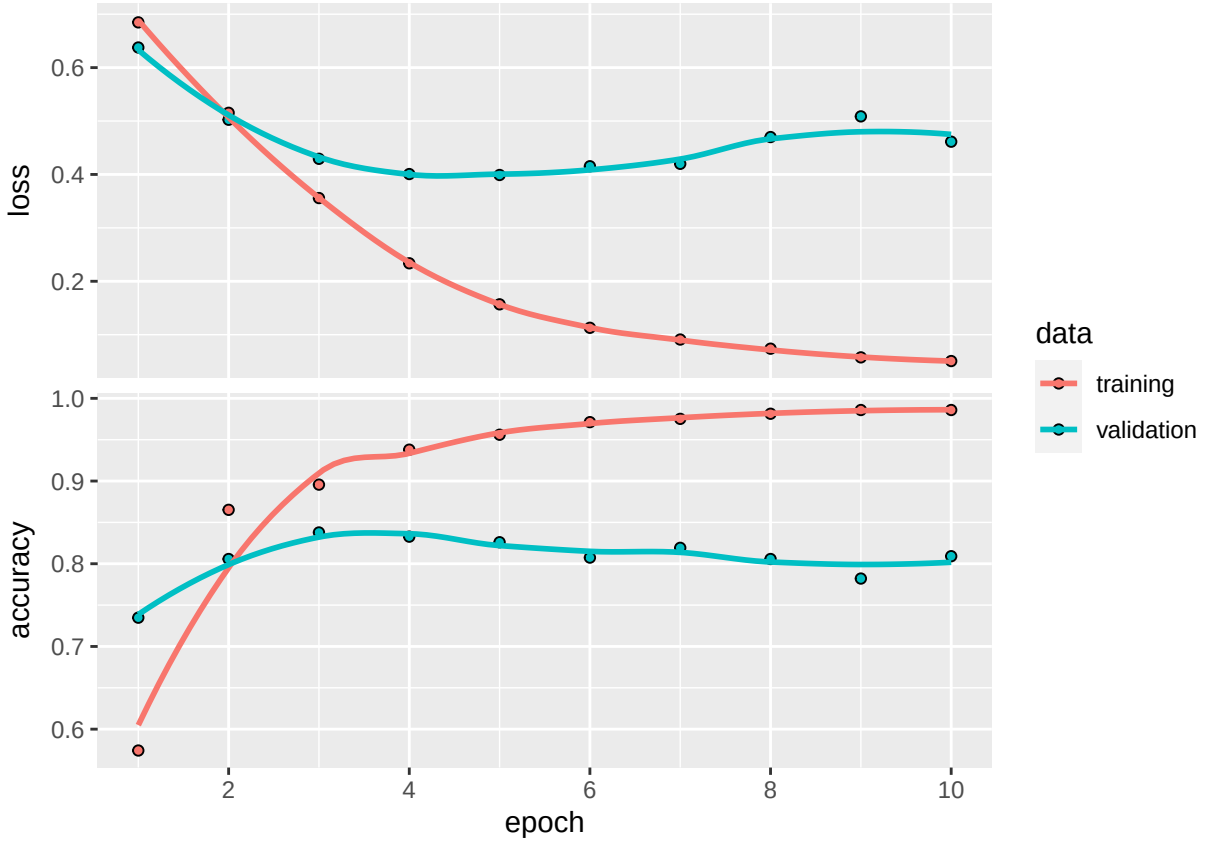
```
history_rnn <- model_rnn %>% fit(
```

```
  x_train_pad, y_train_data,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.25
)
```

```
plot(history_rnn)
```

**Training data**

```
## `geom_smooth()` using formula 'y ~ x'
```




Again after a couple of epochs our model seems to be overfitted as the validation set crosses with our traning set, but our traning set has a rather high accuracy at the point of crossing so overall a good model.

**Evaluation**

```
history_rnn
```

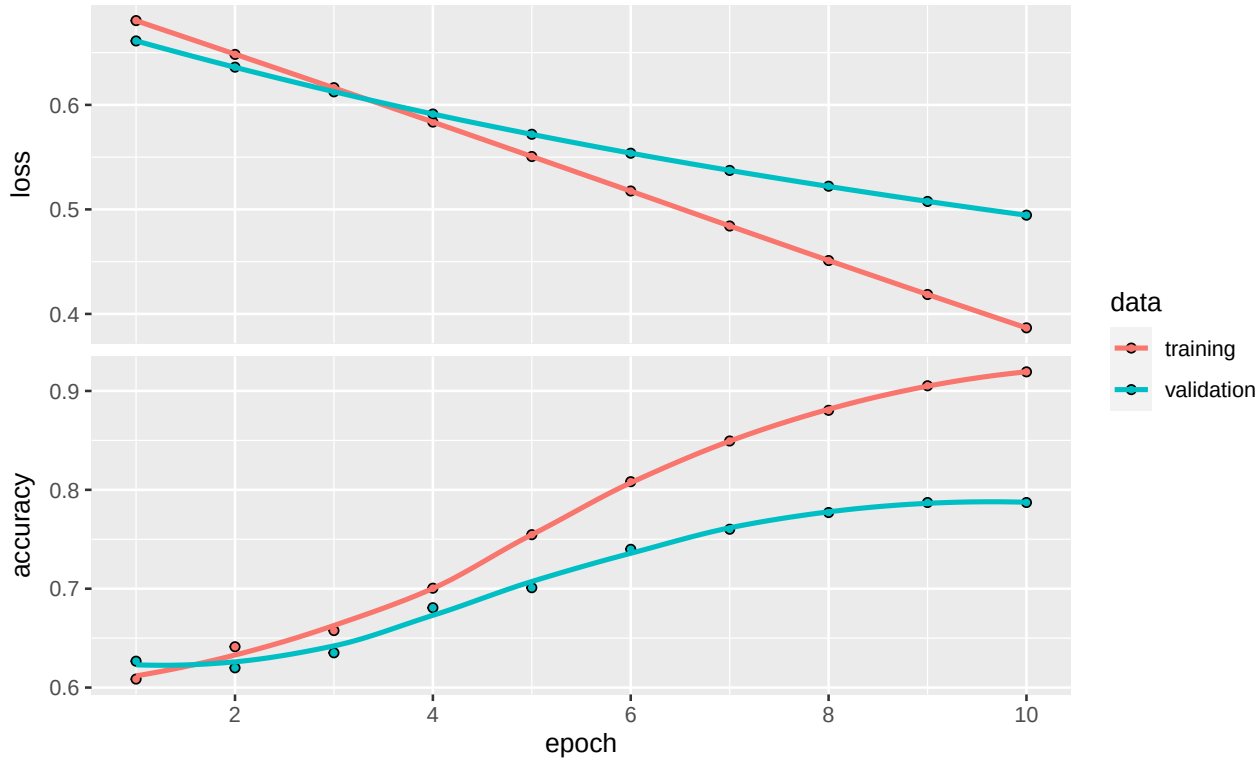
```
##
## Final epoch (plot to see history):
##         loss: 0.05064
##     accuracy: 0.9859
##     val_loss: 0.4613
## val_accuracy: 0.8091
```

Here we can see the metrics of the model in the final epoch.

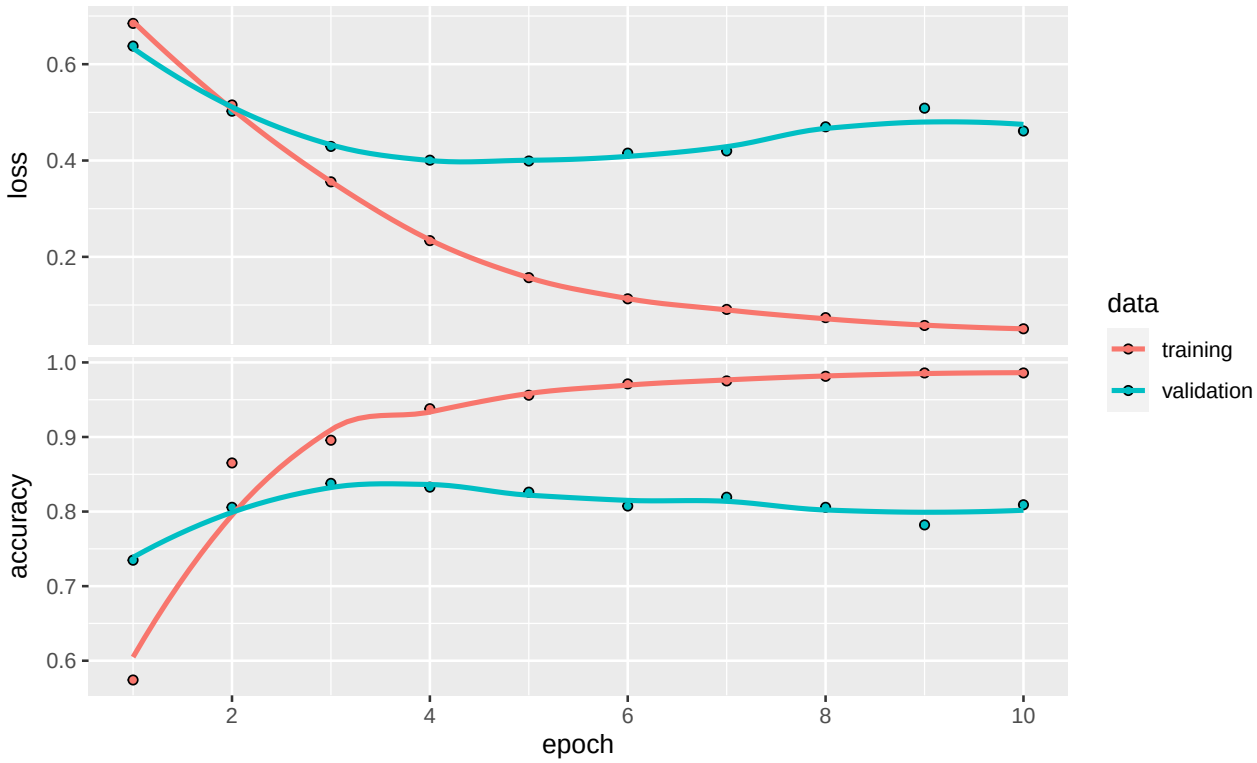
Let's try to put the models on top of each other

```
par(mfrow=c(2,1))
plot(history_ann); plot(history_rnn)
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
## `geom_smooth()` using formula 'y ~ x'
```




The first being the baseline and the second being the Recurrent neural network. Decide for yourself whcih one you like the most.