Assifnment_1_rspake1

The Original Model (Part 1)

In this first section, we are building the original model by following the model built in the book. After this is done, we can go back and make the adjustments in part 2 that satisfy the assignment prompt.

```
library(keras)
```

```
## Warning: package 'keras' was built under R version 4.1.2
```

```
imdb <- dataset_imdb(num_words =10000)
```

```
## Loaded Tensorflow version 2.8.0
```

```
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% imdb
```

From above:

As illustrated in the book: The variables `train_data` and `test_data` are lists of reviews, each review being a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists of 0s and 1s, where 0 stands for “negative” and 1 stands for “positive”

```
##  int [1:218] 1 14 22 16 43 530 973 1622 1385 65 ...
```

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

Above is an example of the first entry, from the string of information, the train label indicates a “positive” review for the first entry.

Time to Prepare the Data

Bellow we will One-hot-encode the lists to transform them into vectors on 0 and 1 to be fed into our neural network.

```
vectorize_sequences <- function(sequences, dimension = 10000) {
  results <- matrix(0, nrow = length(sequences), ncol = dimension)
  for(i in 1:length(sequences))
    results[i, sequences[[i]]] <- 1
  results
}
```

Now we vectorize our train and test data.

```
x_train <- vectorize_sequences(train_data)
x_test <- vectorize_sequences(test_data)

y_train <- as.numeric(train_labels)
y_test <- as.numeric(test_labels)

str(x_train[1,])
```

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

Now that we have vectorized the train data, we can clearly see that the first two entries in the train data are classified as “positive” and the third is “negative”.

Now that it has been prepared, we can feed the data into the neural network

Building the Network

Bellow we are establishing our network as it is done in the book, with two relu layers at 16 units and one sigmoid layer at 1 unit. Doing it this way allows us to understand the output like a probability as the sigmoid layer makes the output fit the 0 or 1 framework.

```
library(keras)

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")

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

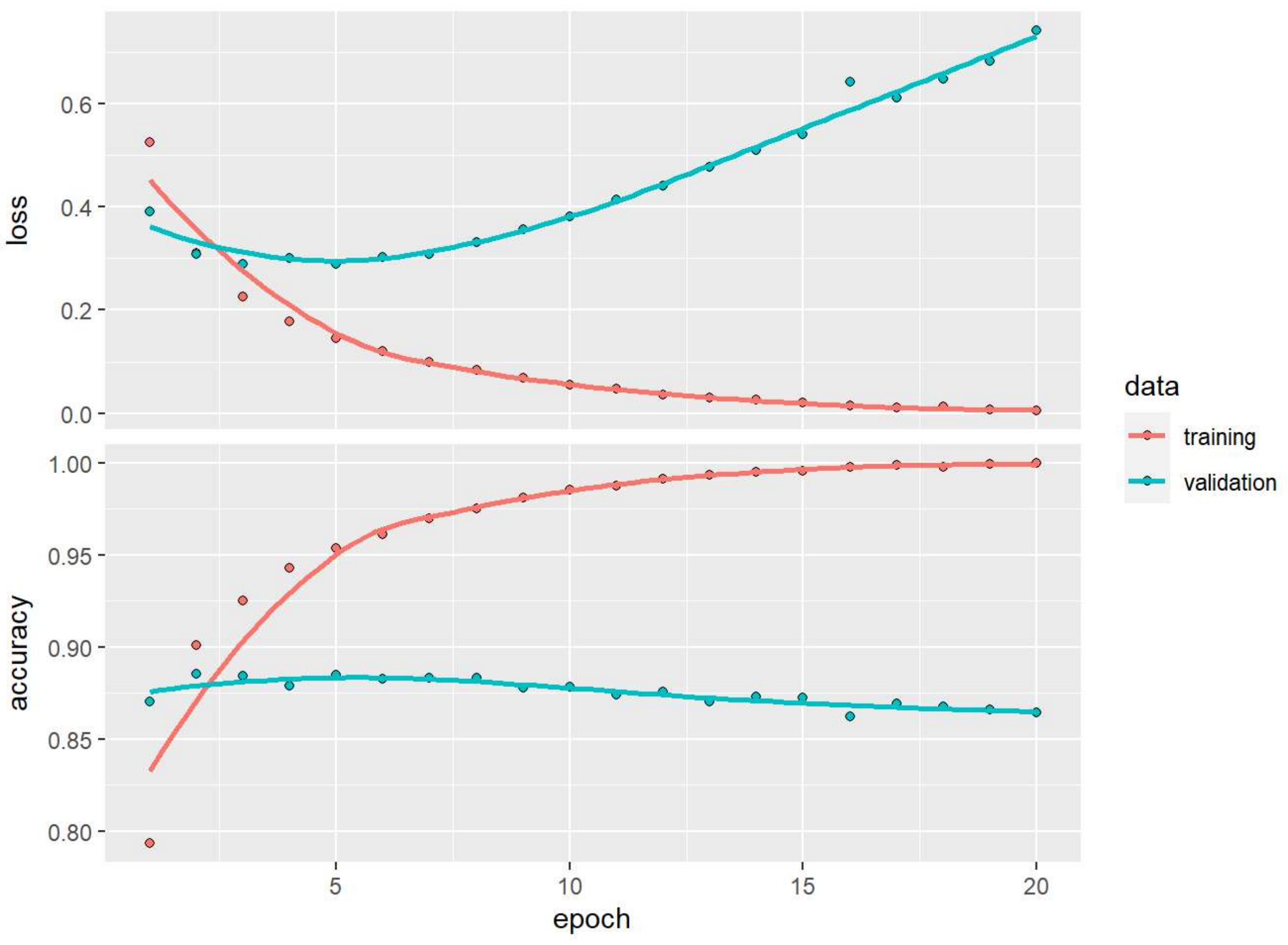
Validating our model

```
val_indices <- 1:10000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
```

Training

```
history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

plot(history)
```

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



Train a new model using less epochs to avoid overfitting

```
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")

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

model %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results <- model %>% evaluate(x_test, y_test)
results
```

```
##      loss  accuracy
## 0.2897113 0.8854000
```

Predict using model

```
model %>% predict(x_test[1:10,])
```

```
##              [,1]
##  [1,] 0.168766707
##  [2,] 0.999184489
##  [3,] 0.915364146
##  [4,] 0.852203250
##  [5,] 0.951246023
##  [6,] 0.827501059
##  [7,] 0.999041200
##  [8,] 0.006707579
##  [9,] 0.956924200
## [10,] 0.992539525
```

Adjusted Model for Assignment (Part 2)

Now, using the outline of the model above, we will recreate the model, making adjustments to certain aspects. At the end, we will attempt to perform better on our validation.

In this section, we will be making the EXACT changes asked for in the prompt and seeing how they effect the performance of the model.

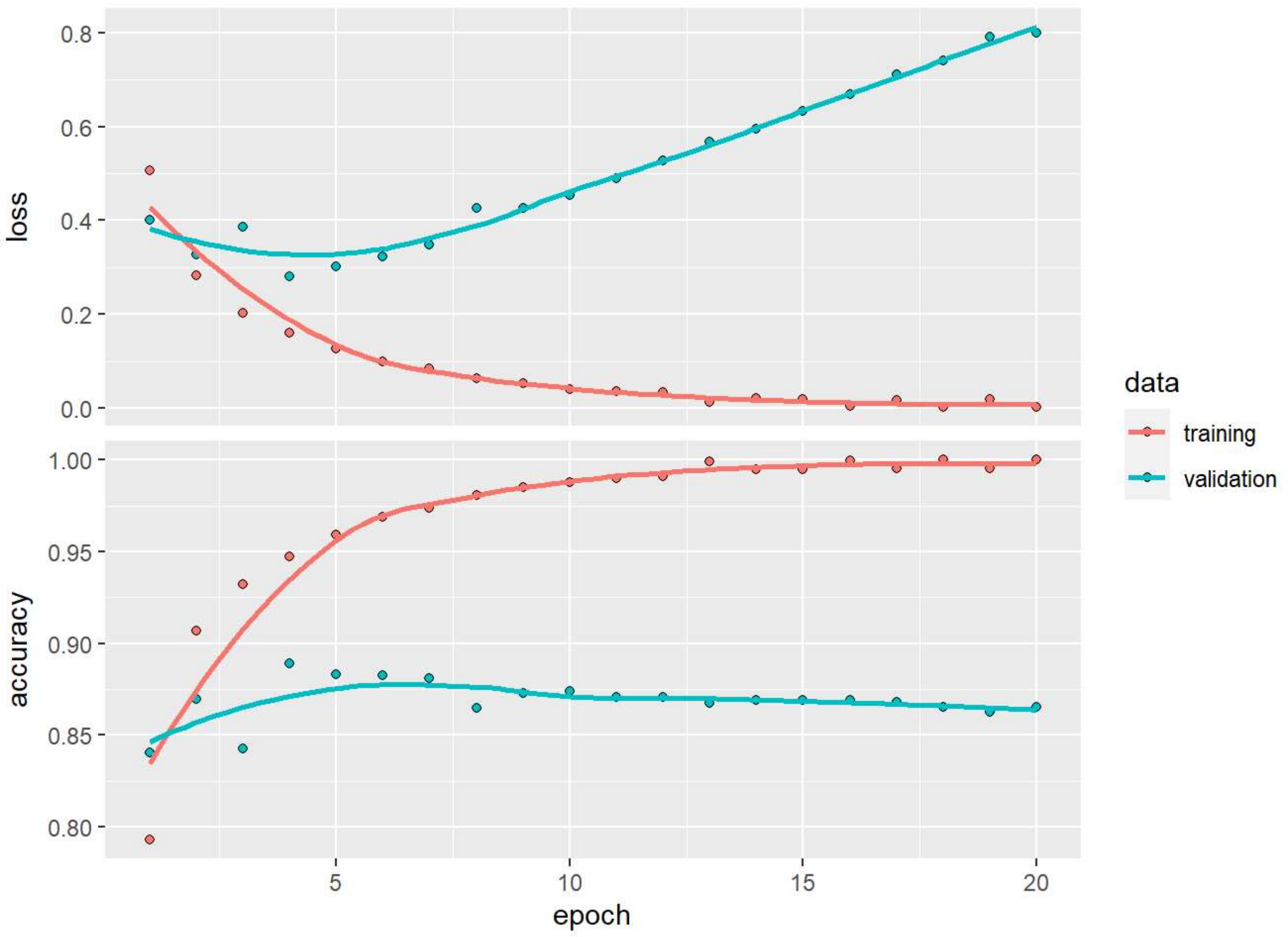
Number 1, Changing Amount of Layers

```r
library(keras)

model_2 <- keras_model_sequential() %>% layer_dense(units = 16, activation = "relu", input_shape
= c(10000)) %>% layer_dense(units = 16, activation = "relu") %>% layer_dense(units = 16, activat
ion = "relu") %>% layer_dense(units = 1, activation = "sigmoid")

model_2 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

```r
val_indices <- 1:10000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
```

```r
history2 <- model_2 %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

plot(history2)
```

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



```
model_2 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results <- model_2 %>% evaluate(x_test, y_test)
results
```

```
##      loss  accuracy
## 0.5209313 0.8590800
```

```
model_2 %>% predict(x_test[1:10,])
```

```
##                [,1]
##  [1,] 0.0225864649
##  [2,] 0.9999947548
##  [3,] 0.9278842211
##  [4,] 0.9701054096
##  [5,] 0.9928889871
##  [6,] 0.9976236820
##  [7,] 0.9998844862
##  [8,] 0.0006593764
##  [9,] 0.9752671123
## [10,] 0.9999897480
```

After we see the prediction and compare it to the original, it is clear to see that when we add a another layer we actually lose accuracy and have much higher loss than before.

Comparing the two predictions, it is obvious that the units that the model was confident on became more confident, but the units with low confidence became less confident. Looking at [3,] we see a positive change, however [1,] drastically negative change.

I believe this may be due to overfitting of the training data, where it picked up more noise from training.

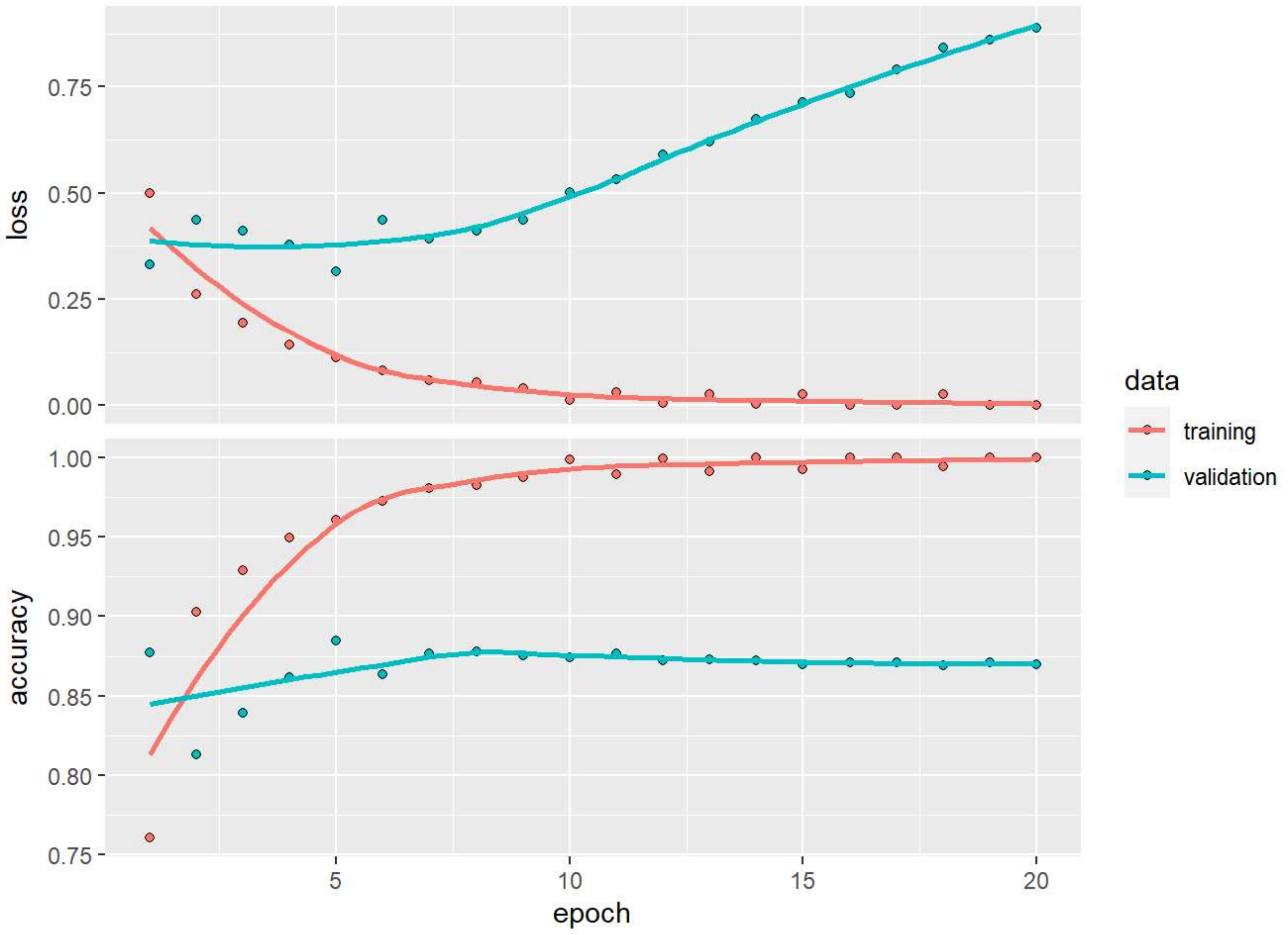
Number 2, Changing Layer Units

```
library(keras)

model_3 <- keras_model_sequential() %>% layer_dense(units = 64, activation = "relu", input_shape
= c(10000)) %>% layer_dense(units = 64, activation = "relu") %>% layer_dense(units = 1, activati
on = "sigmoid")

model_3 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

```
val_indices <- 1:10000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
```

```
history3 <- model_3 %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

plot(history3)
```

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



```
model_3 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results <- model_3 %>% evaluate(x_test, y_test)
results
```

```
##      loss  accuracy
## 0.5980617 0.8642000
```

```
model_3 %>% predict(x_test[1:10,])
```

```
##               [,1]
##  [1,] 1.577038e-02
##  [2,] 1.000000e+00
##  [3,] 9.998381e-01
##  [4,] 8.466014e-01
##  [5,] 9.976469e-01
##  [6,] 9.999073e-01
##  [7,] 1.000000e+00
##  [8,] 7.375389e-06
##  [9,] 9.985485e-01
## [10,] 1.000000e+00
```

Much like when adding a layer, adding units has dramatic impact on loss and accuracy.

This adjustment doubled the loss from before and decreased accuracy by “.02”.

We see another case where the model has extreme reactions to what it believes is correct and incorrect. Even comparing it to Number 1, this model is 100% confident in [2,],[7,], and [10,]. Yet, it is even less confident than before for [1,] and [3,].

From both of these observations, it seems that adding to these models in both layers and units makes the model heavily overfit with the train data and performs poorly on the validation and test sets.

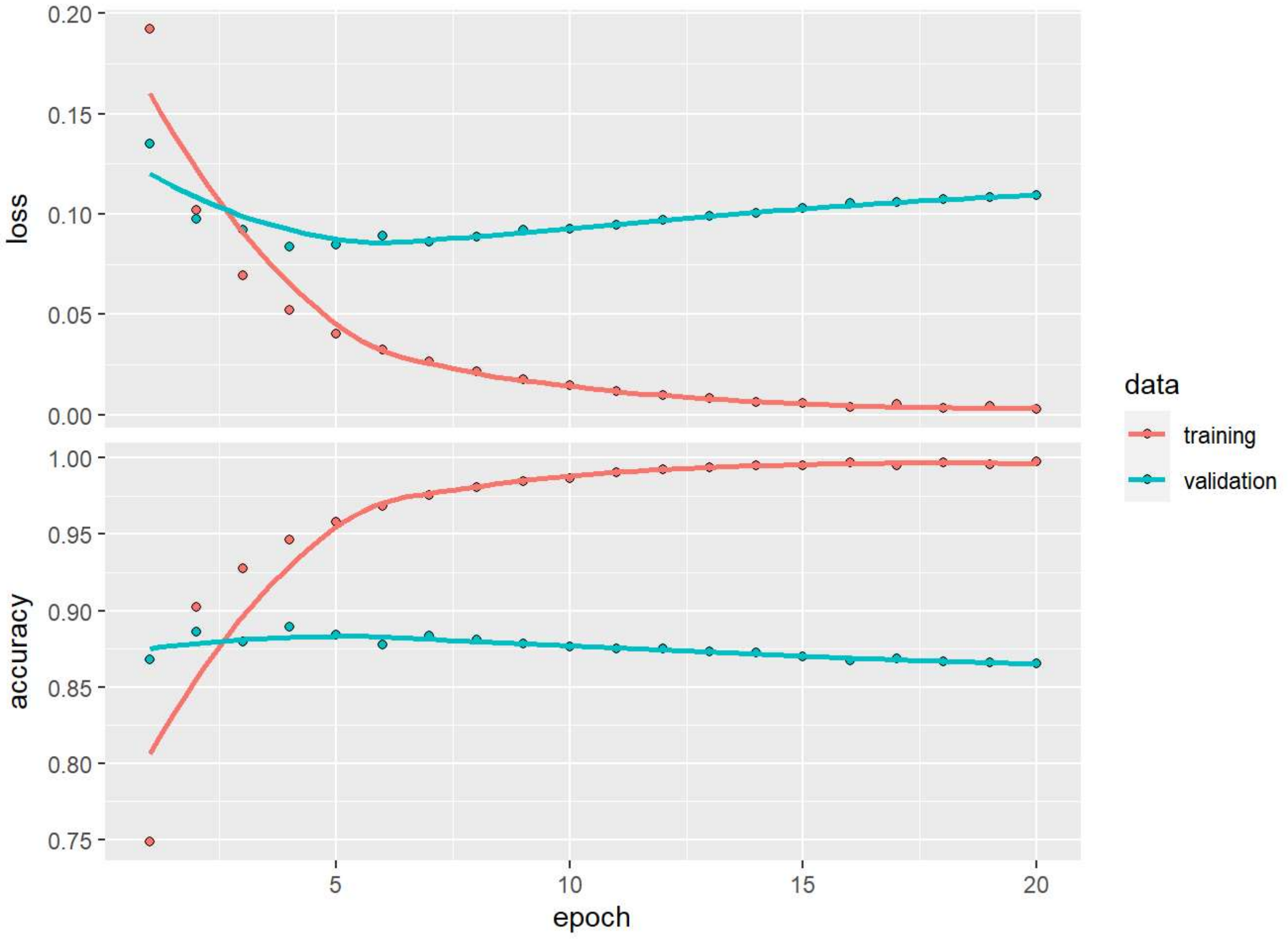
Number 3, mse

```
library(keras)

model_4 <- keras_model_sequential() %>% layer_dense(units = 16, activation = "relu", input_shape
= c(10000)) %>% layer_dense(units = 16, activation = "relu") %>% layer_dense(units = 1, activati
on = "sigmoid")

model_4 %>% compile(
  optimizer = "rmsprop",
  loss = "mse",
  metrics = c("accuracy")
)
```

```
val_indices <- 1:10000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
```

```
history4 <- model_4 %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

plot(history4)
```

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



```
model_4 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results <- model_4 %>% evaluate(x_test, y_test)
results
```

```
##      loss  accuracy
## 0.1117132 0.8638000
```

```
model_4 %>% predict(x_test[1:10,])
```

```
##              [,1]
##  [1,] 0.022712499
##  [2,] 1.000000000
##  [3,] 0.997950494
##  [4,] 0.884446979
##  [5,] 0.999774694
##  [6,] 0.999063134
##  [7,] 0.999943256
##  [8,] 0.002414197
##  [9,] 0.956685483
## [10,] 0.999998093
```

This case is more interesting than the previous adjustments. By changing the loss function, we reduce loss greatly, at the expense of a little loss in accuracy. On the surface this would seem to be the obvious choice to use since it reduces loss by so much.

However, the real issue appears when you predict the test set. The model loses confidence similarly to how the other adjustments have, and [9,] even loses some confidence.

I believe this is due to how the loss functions measure loss. Since our model is designed to measure output a “probability”, crossentropy is better as it measures distance between probability distributions. MSE is Mean Squared Error which would be less applicable in our case since the output is forced into the binary form sure to sigmoid.

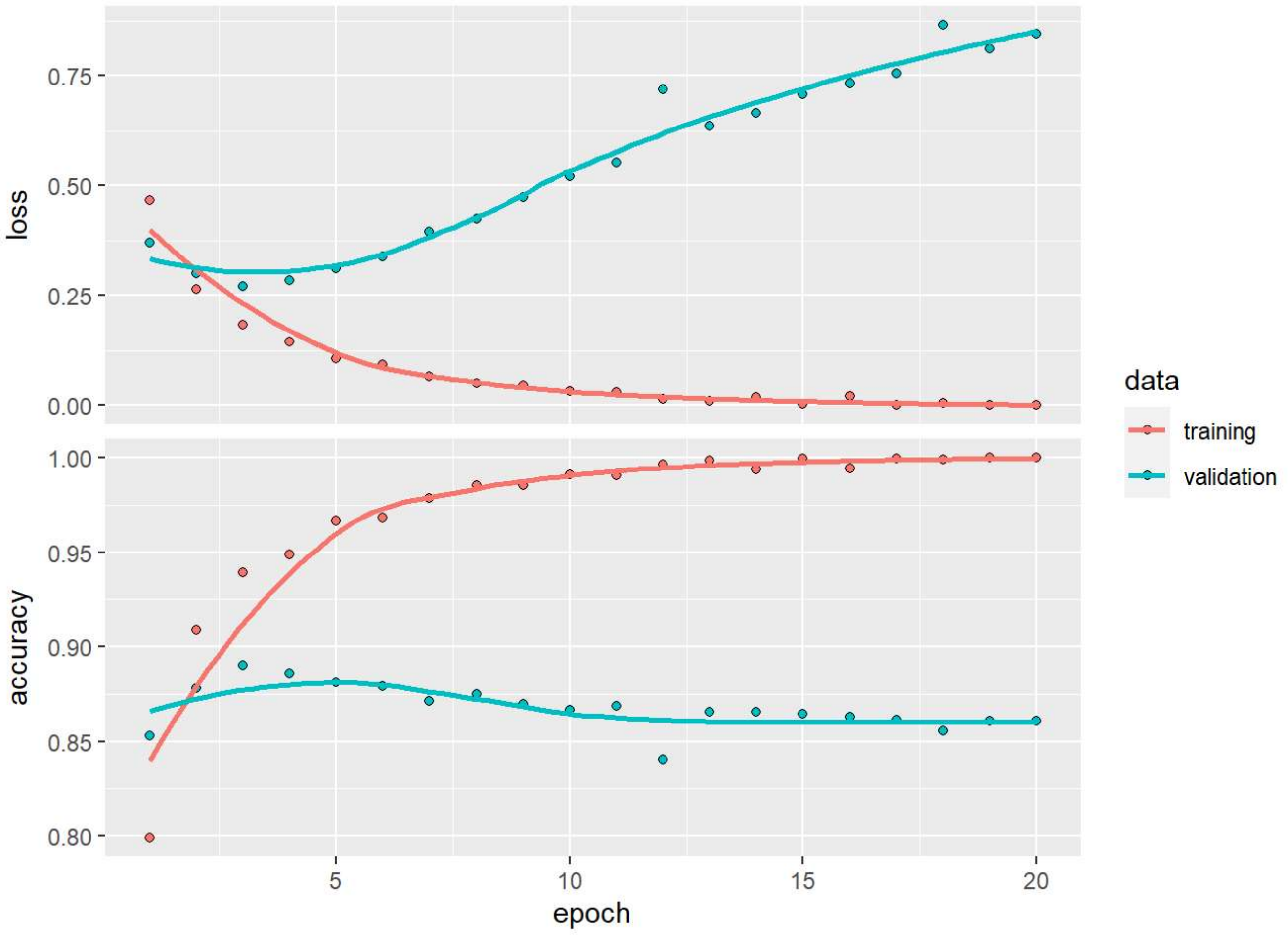
Number 4, tanh

```
library(keras)

model_5 <- keras_model_sequential() %>% layer_dense(units = 16, activation = "tanh", input_shape
= c(10000)) %>% layer_dense(units = 16, activation = "tanh") %>% layer_dense(units = 1, activati
on = "sigmoid")

model_5 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

```
val_indices <- 1:10000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
```

```
history5 <- model_5 %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

plot(history5)
```

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



```
model_5 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results <- model_5 %>% evaluate(x_test, y_test)
results
```

```
##      loss  accuracy 
## 0.4876024 0.8554400
```

```
model_5 %>% predict(x_test[1:10,])
```

```
##               [,1]
##  [1,] 0.0045495629
##  [2,] 0.9996865988
##  [3,] 0.9913512468
##  [4,] 0.9694321156
##  [5,] 0.9992716312
##  [6,] 0.9975706935
##  [7,] 0.9924060106
##  [8,] 0.0005244017
##  [9,] 0.9947632551
## [10,] 0.9995533228
```

Much like how we have seen issues with increased loss and decrease in accuracy with the other alterations, tanh also returns units that are more extreme in confidence (where not confident before, it is even LESS confident).

I believe this is due to “relu” being an activation function that zeros-out negative values. By not using “relu”, these negative values impact the model.

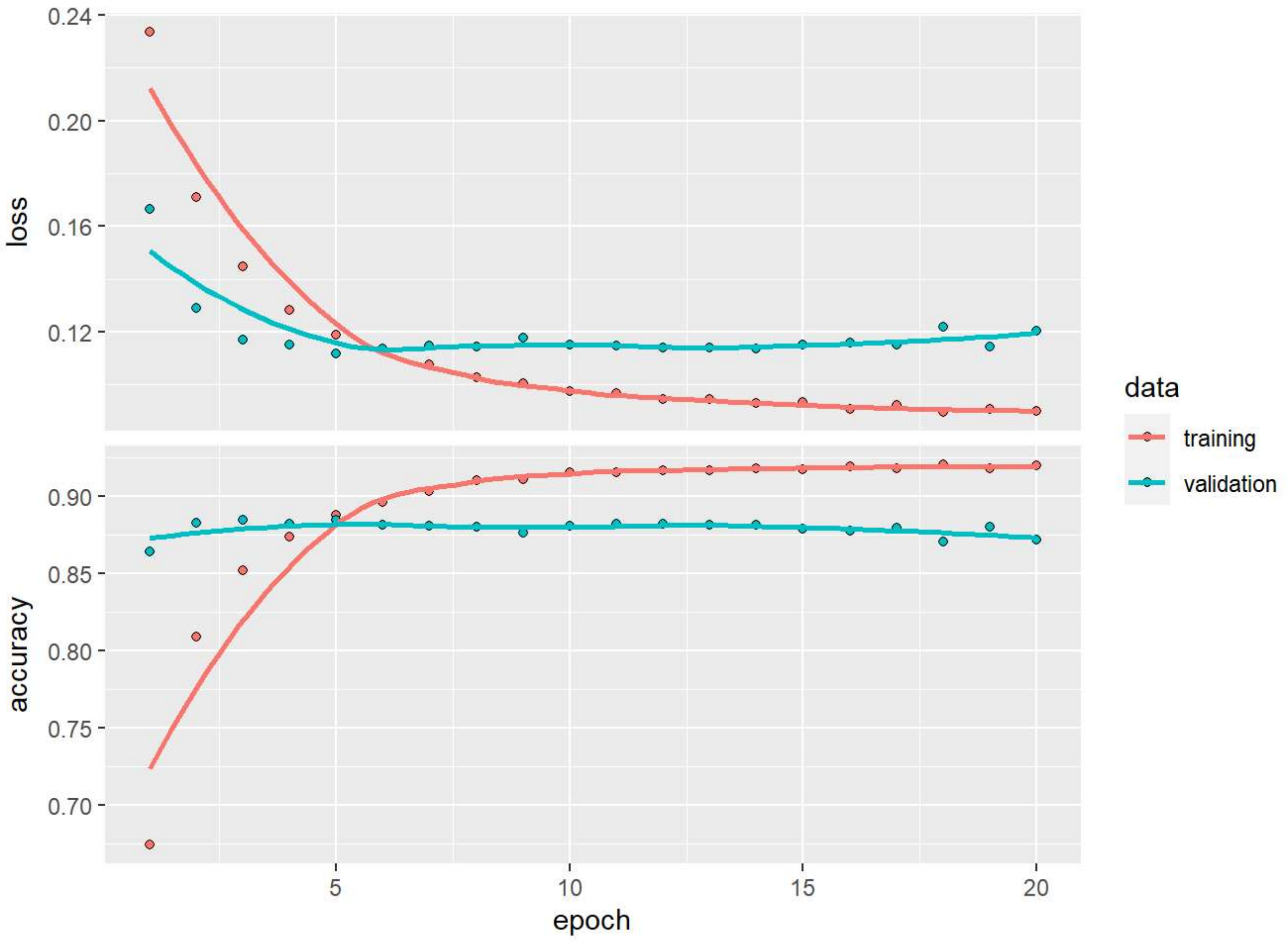
Number 5, Improvments to Validation

For this section we will be relying on mse, regularization, and dropout to try and improve the validation.

```
library(keras)

l2_regular_model <- keras_model_sequential() %>% layer_dense(units = 16, kernel_regularizer = re
gularizer_l2(0.001), activation = "relu", input_shape = c(10000)) %>% layer_dropout(rate = 0.5)
 %>% layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.001), activation = "relu")  %
>% layer_dropout(rate = 0.5) %>% layer_dense(units = 1, activation = "sigmoid")

l2_regular_model %>% compile(
  optimizer = "rmsprop",
  loss = "mse",
  metrics = c("accuracy")
)
```

```
l2_model_hist <- l2_regular_model %>% fit(
  x_train, y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_test, y_test)
)

plot(l2_model_hist)
```

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



During the validation set we can see that, using regularization, 4 epochs are the optimal number of epochs.

Using 4 epochs, the this model is slightly less accurate but reduces loss greatly.