# Advanced Machine Learning - Assignment 1

Julia Thacker

2/13/2022

```
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
```

Imported the IMDB dataset and limited to the top 10,000 most common words.

```
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
}

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

Vectorized the data.

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

Vectorized the labels.

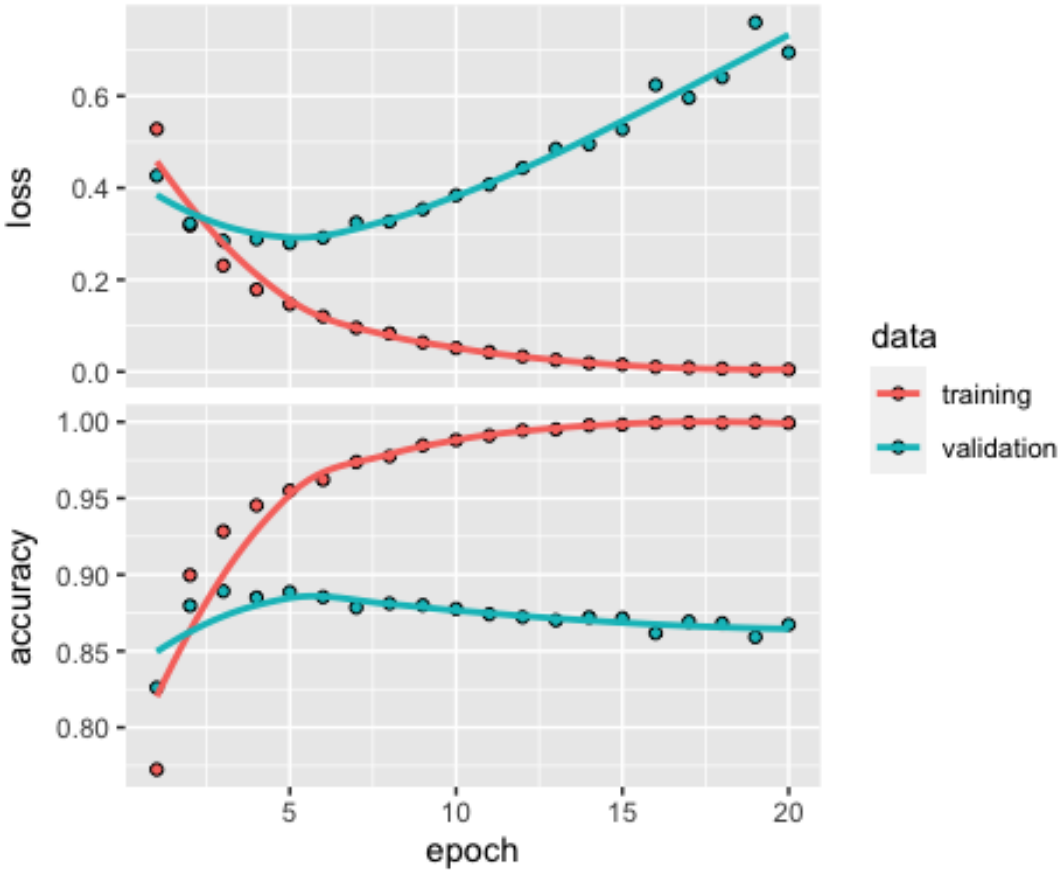
```
val_indices <- 1:10000

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]
```

Created a validation set consisting of 10,000 samples from the training data.

```
originalmodel <- 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")
```

Replicated the original model from the class example for comparison purposes.

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

history <- originalmodel %>% 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'
```




Visualized the original model.

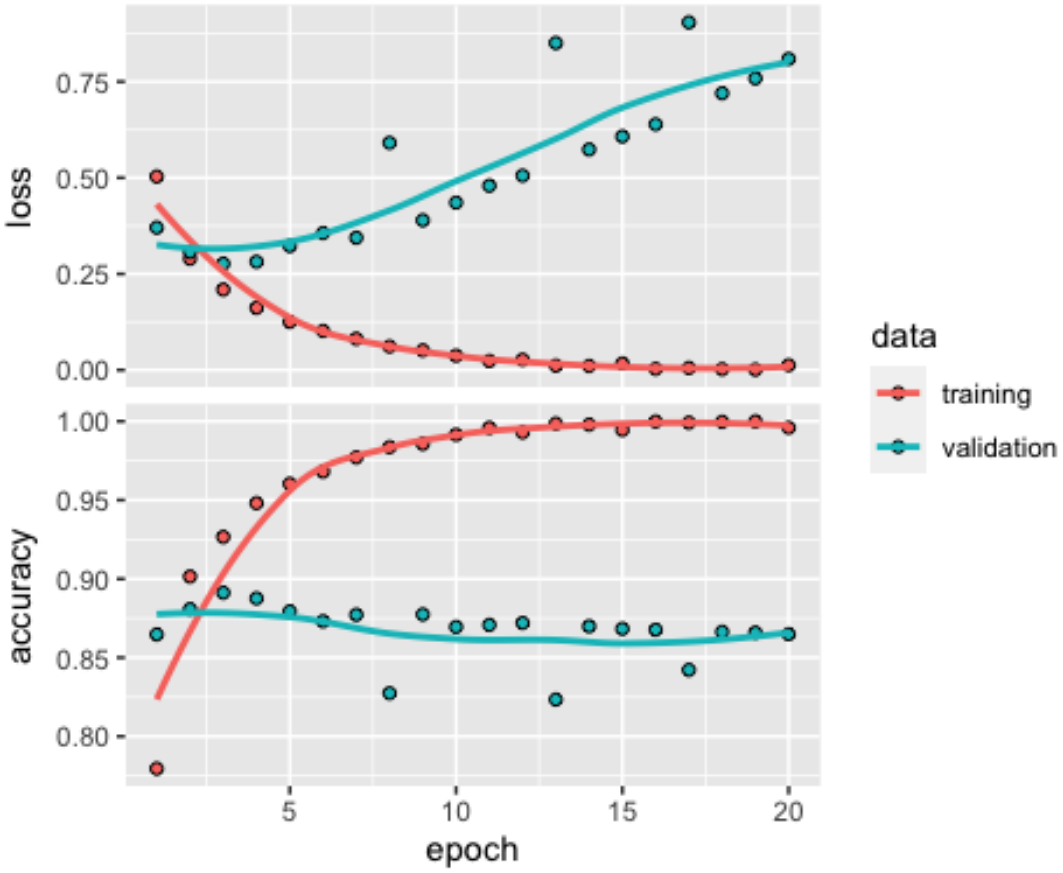
My first test: 3 layers instead of 2

```r
model2 <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

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

history2 <- model2 %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

str(history2)
```

```
## List of 2
##  $ params :List of 3
##   ..$ verbose: int 1
##   ..$ epochs : int 20
##   ..$ steps  : int 30
##  $ metrics:List of 4
##   ..$ loss        : num [1:20] 0.503 0.289 0.209 0.162 0.125 ...
##   ..$ accuracy    : num [1:20] 0.78 0.901 0.927 0.948 0.96 ...
##   ..$ val_loss    : num [1:20] 0.37 0.307 0.277 0.282 0.322 ...
##   ..$ val_accuracy: num [1:20] 0.865 0.881 0.891 0.888 0.879 ...
##  - attr(*, "class")= chr "keras_training_history"
```

```r
plot(history2)
```

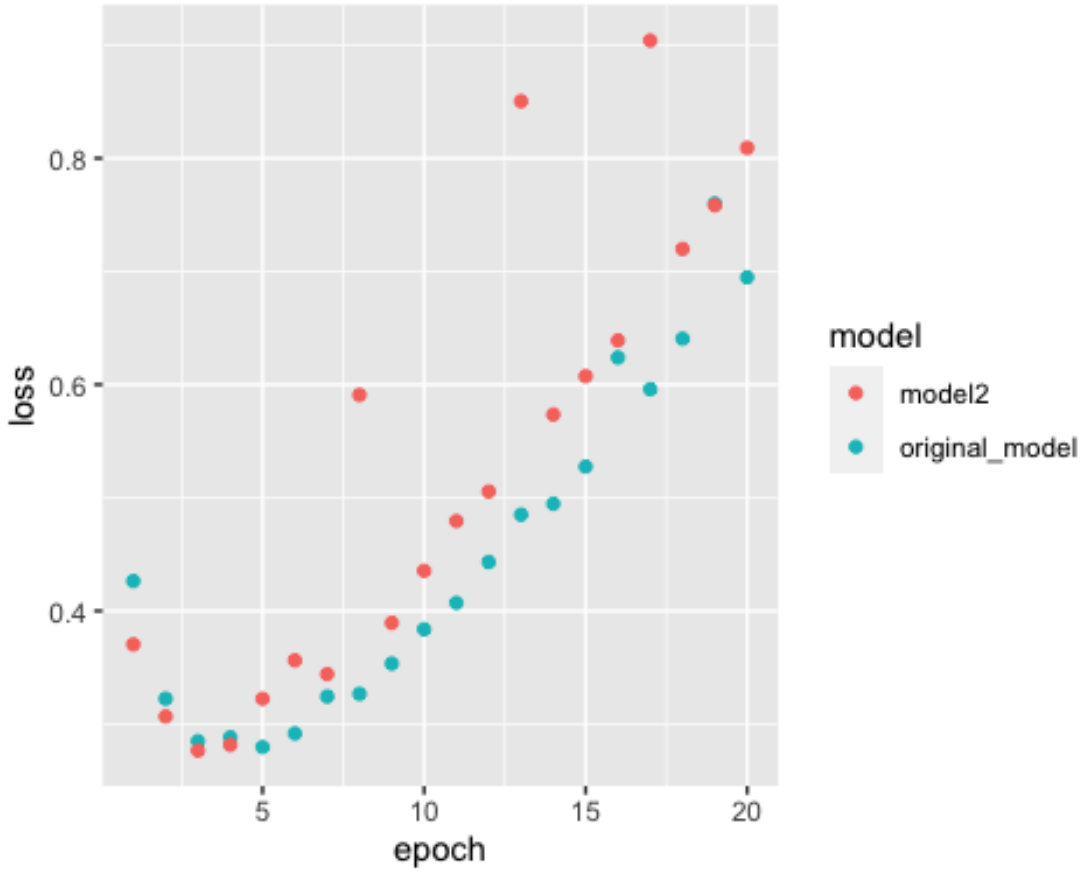
```
## `geom_smooth()` using formula 'y ~ x'
```




```
library(ggplot2)
library(tidyr)
plot_training_losses <- function(losses) {
  loss_names <- names(losses)
  losses <- as.data.frame(losses)
  losses$epoch <- seq_len(nrow(losses))
  losses %>%
    gather(model, loss, loss_names[[1]], loss_names[[2]]) %>%
    ggplot(aes(x = epoch, y = loss, colour = model)) +
    geom_point()
}
```

Plotted the results compared to the original model.

```
plot_training_losses(losses = list(
  original_model = history$metrics$val_loss,
 model2 = history2$metrics$val_loss
))
```



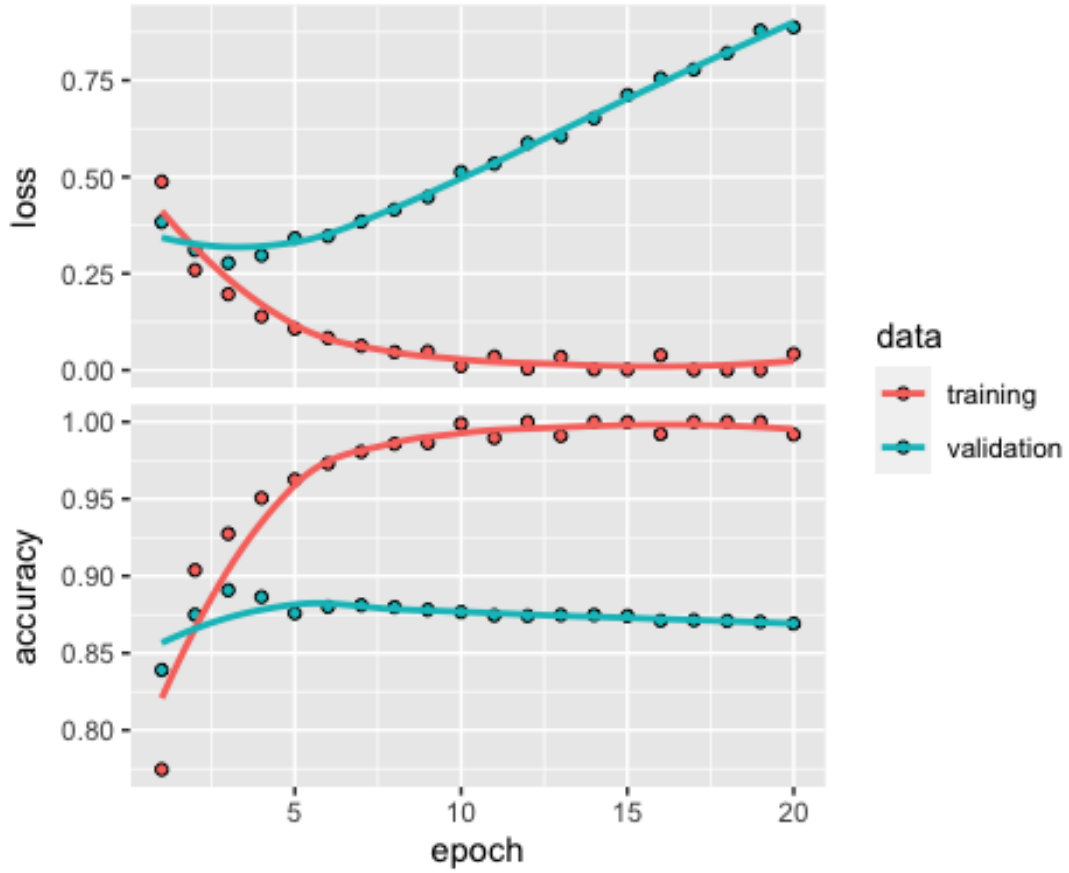

Increasing the number of layers did not improve the model.

```
model3 <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

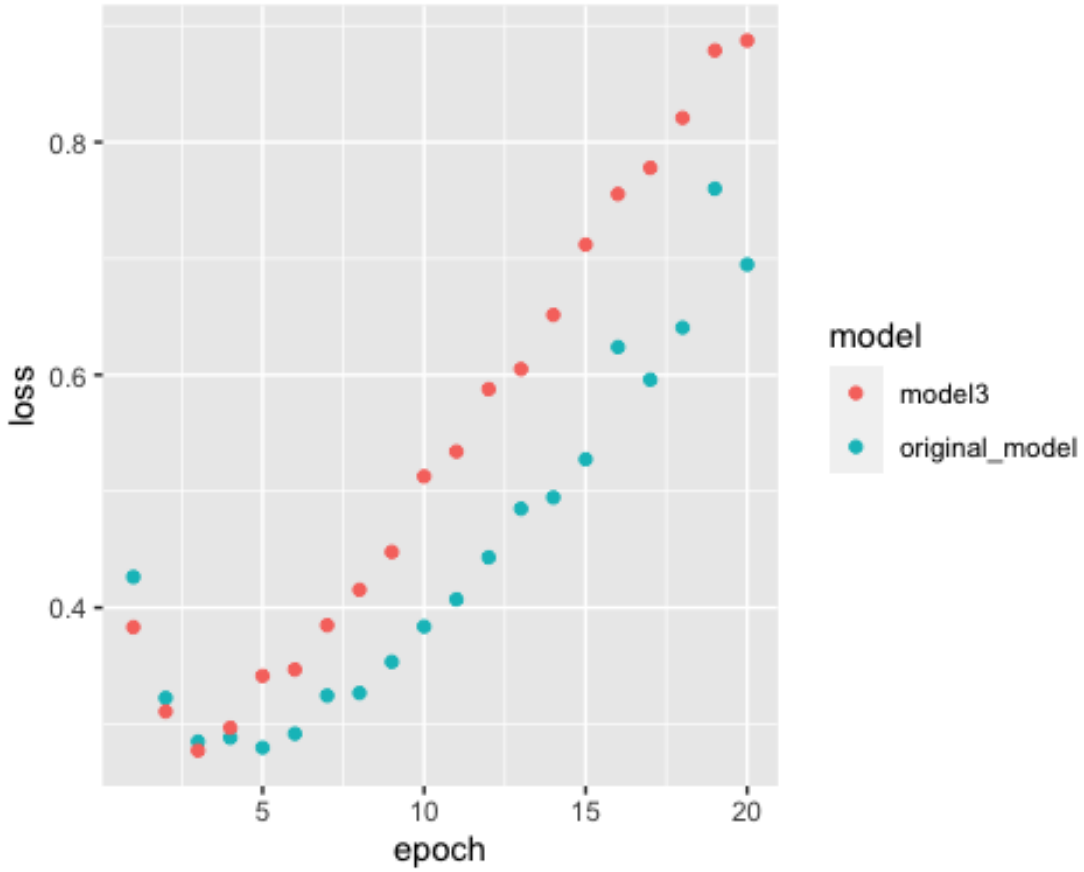
Created a model with 64 units instead of the original 16.

```
model3 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history3 <- model3 %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

str(history3)
```

```
## List of 2
##  $ params :List of 3
##   ..$ verbose: int 1
##   ..$ epochs : int 20
##   ..$ steps  : int 30
##  $ metrics:List of 4
##   ..$ loss        : num [1:20] 0.488 0.259 0.197 0.139 0.107 ...
##   ..$ accuracy    : num [1:20] 0.774 0.904 0.927 0.951 0.963 ...
##   ..$ val_loss    : num [1:20] 0.383 0.311 0.277 0.297 0.341 ...
##   ..$ val_accuracy: num [1:20] 0.839 0.875 0.891 0.886 0.876 ...
##  - attr(*, "class")= chr "keras_training_history"

plot(history3)

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




```
plot_training_losses(losses = list(
  original_model = history$metrics$val_loss,
 model3 = history3$metrics$val_loss
))
```



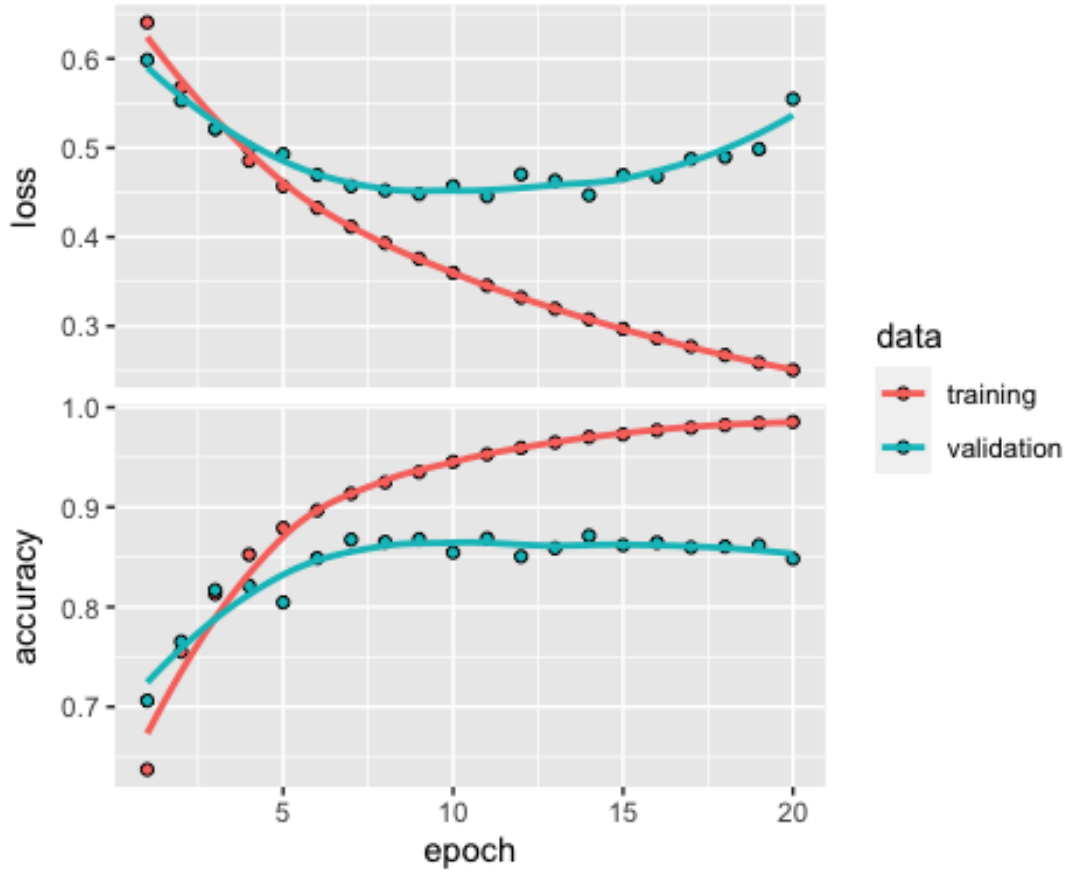

Visualized the model. Increasing the number of hidden units also had a negative impact on the performance of the model.

```
model3b <- keras_model_sequential() %>%
  layer_dense(units = 2, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 2, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

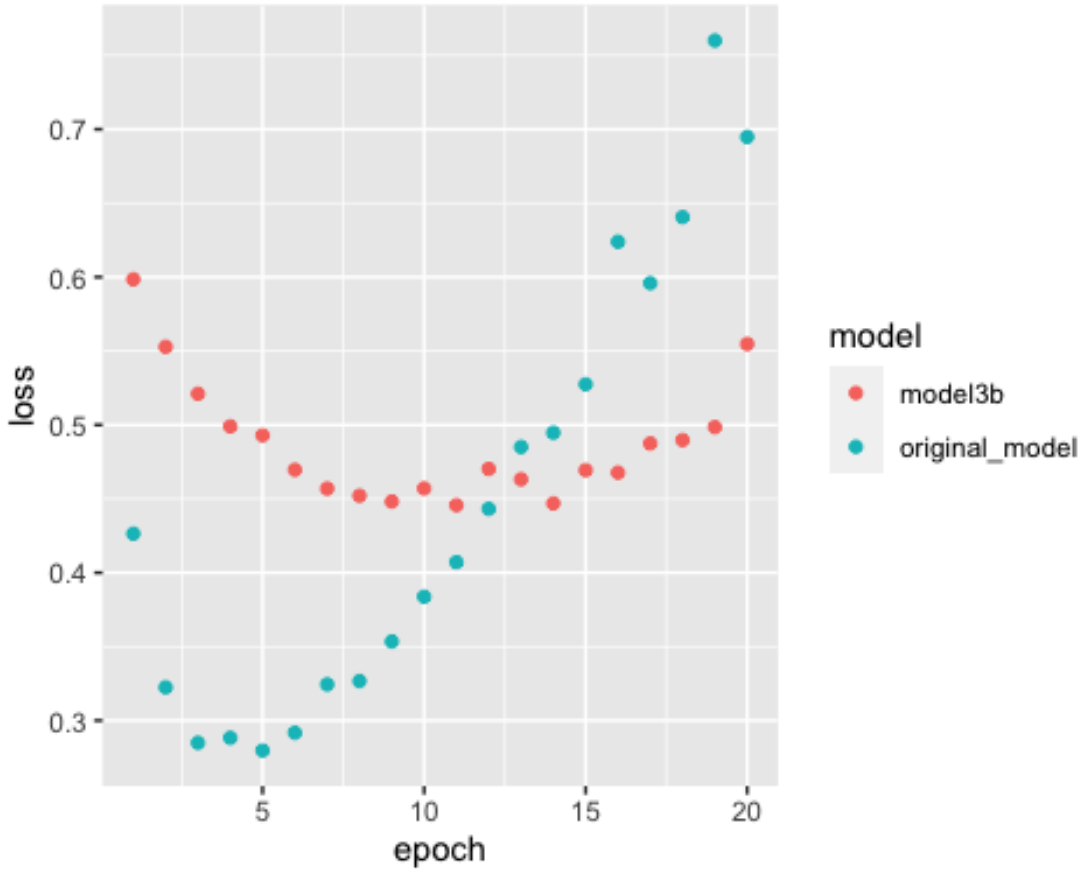
Adjusted the number of hidden units again, decreasing to only 2, to see the results.

```
model3b %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history3b <- model3b %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

str(history3b)
```

```
## List of 2
##  $ params :List of 3
##   ..$ verbose: int 1
##   ..$ epochs : int 20
##   ..$ steps  : int 30
##  $ metrics:List of 4
##   ..$ loss        : num [1:20] 0.641 0.568 0.521 0.486 0.457 ...
##   ..$ accuracy    : num [1:20] 0.637 0.755 0.813 0.852 0.879 ...
##   ..$ val_loss    : num [1:20] 0.599 0.553 0.521 0.499 0.493 ...
##   ..$ val_accuracy: num [1:20] 0.706 0.765 0.817 0.821 0.804 ...
##  - attr(*, "class")= chr "keras_training_history"

plot(history3b)

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




```
plot_training_losses(losses = list(
  original_model = history$metrics$val_loss,
 model3b = history3b$metrics$val_loss
))
```



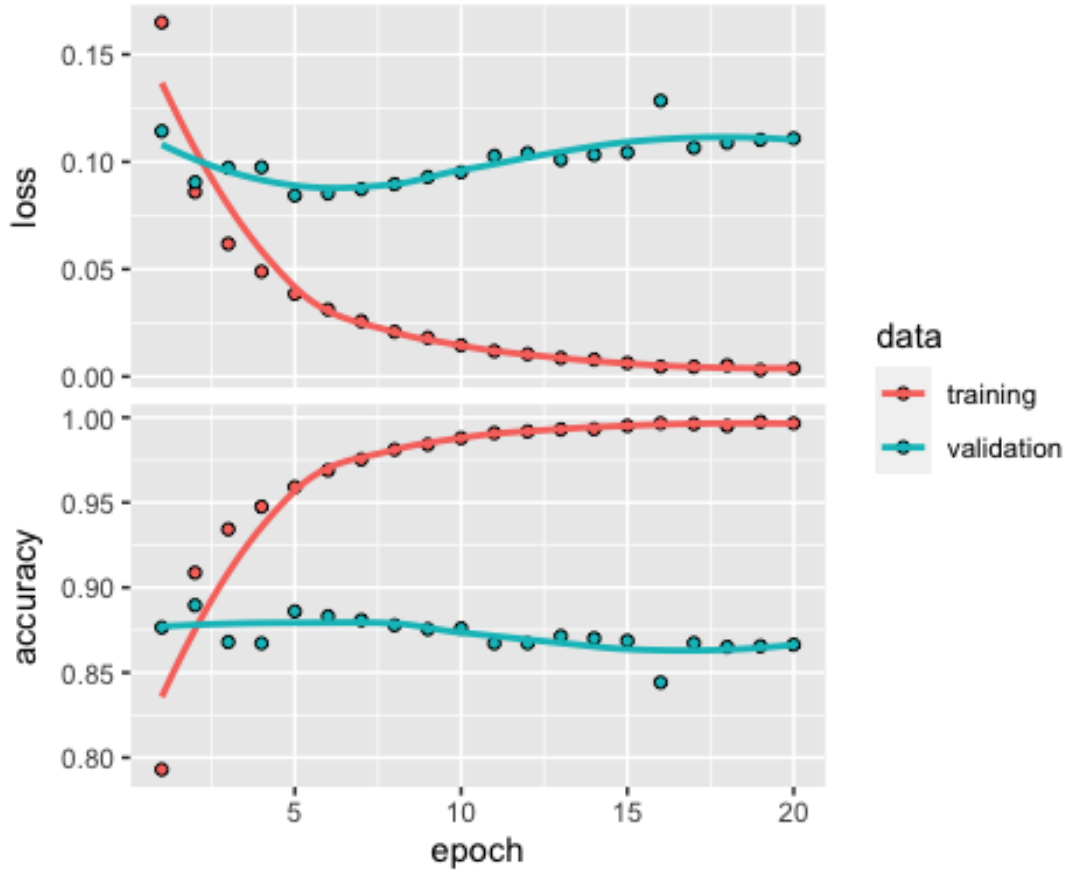

The smaller number of hidden units reduced the validation loss.

```
model4 <- 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")

model4 %>% compile(
  optimizer = "rmsprop",
  loss = "mse",
  metrics = c("accuracy")
)
```

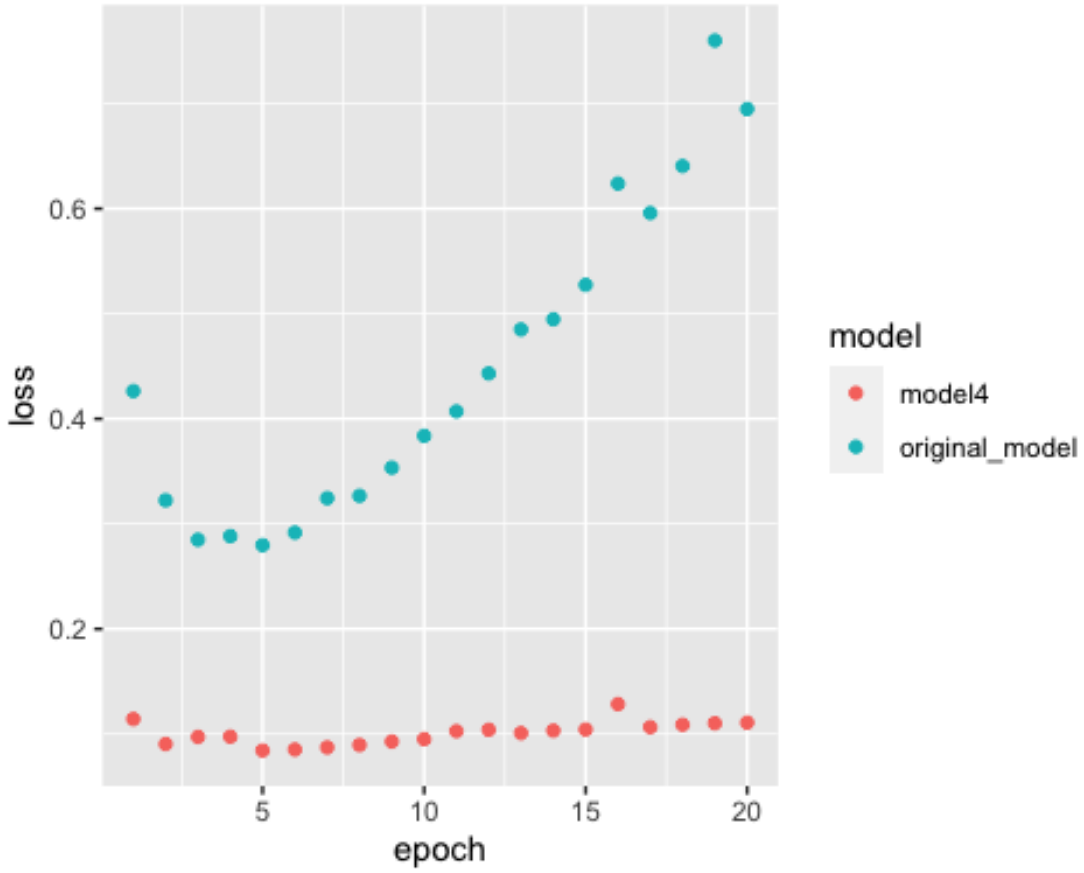
Performed the mse loss function instead of binary_crossentropy.

```
history4 <- model4 %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
str(history4)
```

```
## List of 2
##  $ params :List of 3
##   ..$ verbose: int 1
##   ..$ epochs : int 20
##   ..$ steps  : int 30
##  $ metrics:List of 4
##   ..$ loss        : num [1:20] 0.1649 0.086 0.0618 0.0489 0.0385 ...
##   ..$ accuracy    : num [1:20] 0.793 0.909 0.934 0.948 0.959 ...
##   ..$ val_loss    : num [1:20] 0.1143 0.0905 0.0972 0.0975 0.0843 ...
##   ..$ val_accuracy: num [1:20] 0.877 0.89 0.868 0.867 0.886 ...
##  - attr(*, "class")= chr "keras_training_history"

plot(history4)

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




```
plot_training_losses(losses = list(
  original_model = history$metrics$val_loss,
 model4 = history4$metrics$val_loss
))
```



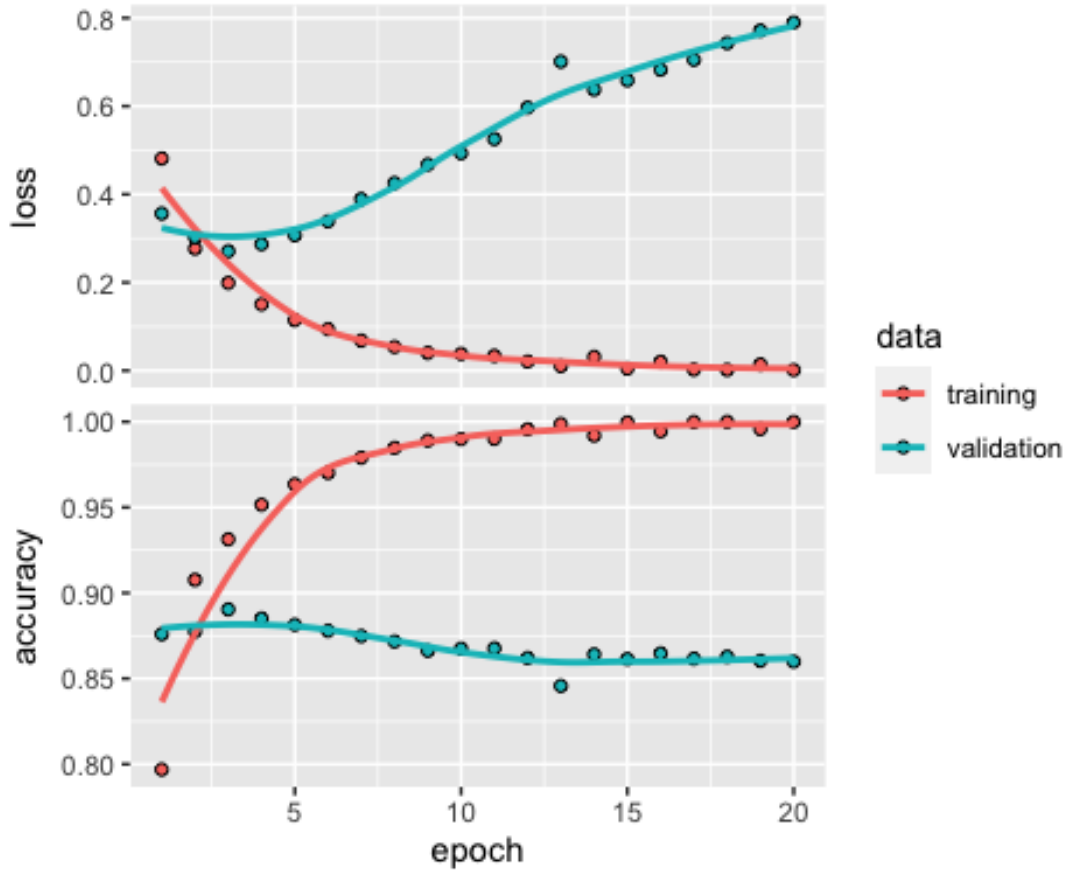

Using the mse loss function has produced the best results so far. This model has less validation loss and is not experiencing as severe overfitting as the original model.

```
model5 <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "tanh", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "tanh") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

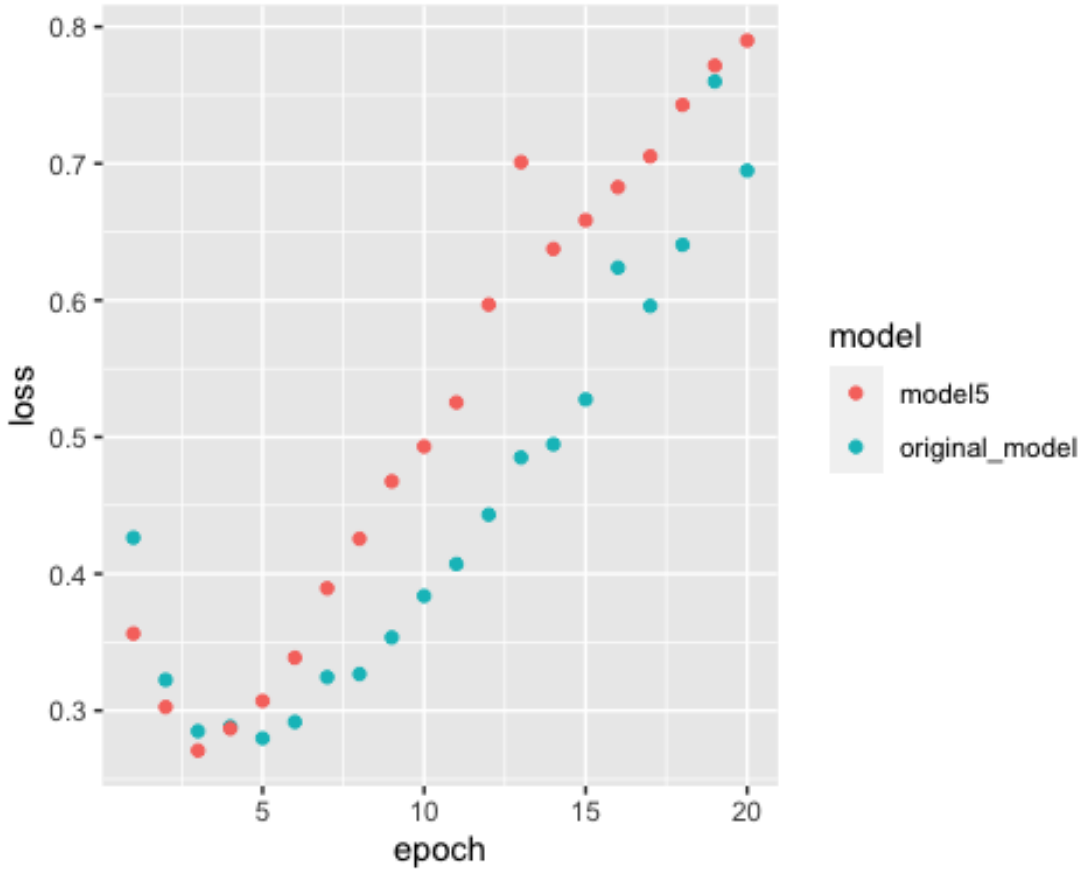
Created a model using the tanh activation method instead of relu.

```
model5 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history5 <- model5 %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
str(history5)
```

```
## List of 2
##  $ params :List of 3
##   ..$ verbose: int 1
##   ..$ epochs : int 20
##   ..$ steps  : int 30
##  $ metrics:List of 4
##   ..$ loss        : num [1:20] 0.481 0.277 0.199 0.15 0.114 ...
##   ..$ accuracy    : num [1:20] 0.797 0.908 0.931 0.952 0.963 ...
##   ..$ val_loss    : num [1:20] 0.356 0.303 0.271 0.287 0.307 ...
##   ..$ val_accuracy: num [1:20] 0.876 0.877 0.89 0.885 0.881 ...
##  - attr(*, "class")= chr "keras_training_history"

plot(history5)

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




```
plot_training_losses(losses = list(
  original_model = history$metrics$val_loss,
  model5 = history5$metrics$val_loss
))
```

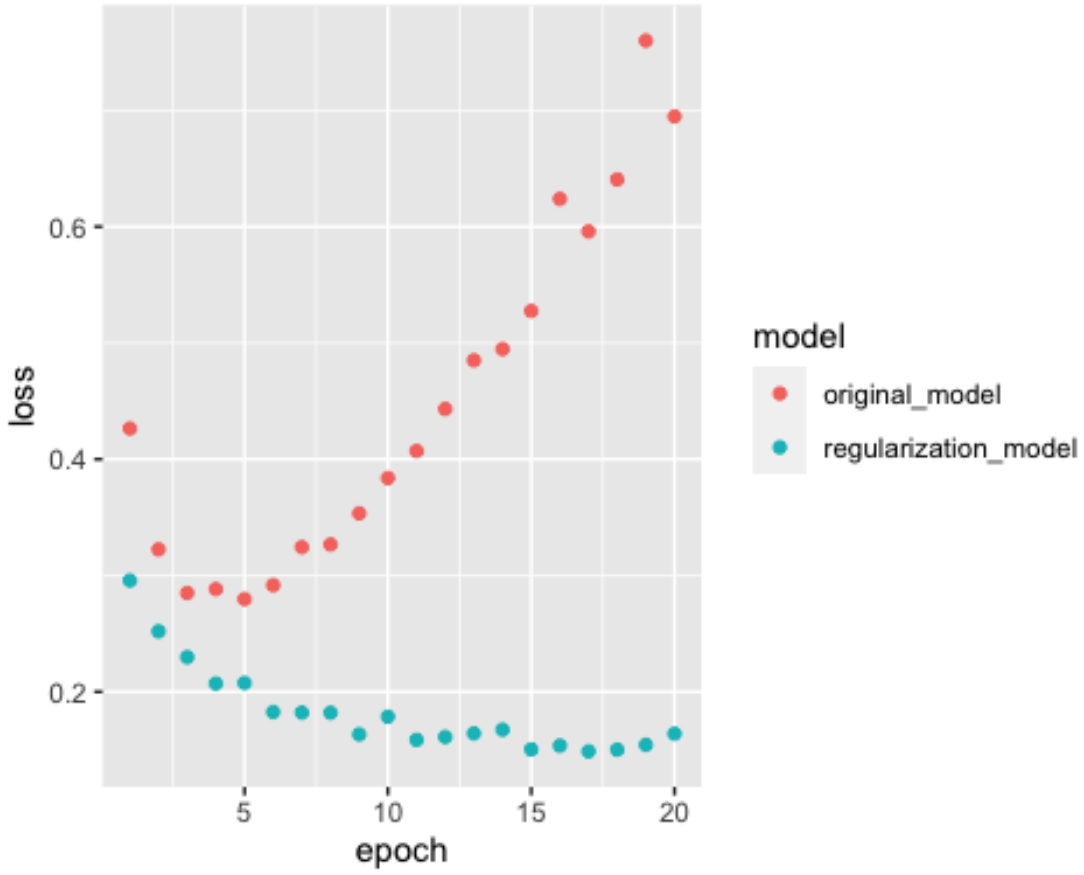



Using the tahn activation method still results in overfitting and additional validation loss compared to the original model.

In an attempt to get a model that could perform better on validation, I decided to use the regularization technique. I tried a few variations of this method before concluding that the code below produced the best results. My final decision was to use a weight of 0.01. I also decided to use the mse loss function instead of binary_crossentropy, because this method appeared to produce the best results during my prior trials.

```
regularization_model <- keras_model_sequential() %>%
  layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.01),
              activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.01),
              activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

regularization_model %>% compile(
  optimizer = "rmsprop",
  loss = "mse",
  metrics = c("acc")
)

regularization_model_hist <- regularization_model %>% fit(
  x_train, y_train,
```

```
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_test, y_test)
)

plot_training_losses(losses = list(
  original_model = history$metrics$val_loss,
  regularization_model = regularization_model_hist$metrics$val_loss
))
```




This final regularization model performed significantly better than the original model. Both the training and validation data in this model were resistant to overfitting and primarily showed a reduction in loss with every epoch. Although the accuracy of the validation data does dip at some points, the accuracy is not as low as some of the previous model attempts, and the accuracy of the validation data is much more similar to the accuracy of the training data than any of the previous iterations.