#Import data set into R  
  
library(keras)  
  
imdb <- dataset\_imdb(num\_words = 10000)  
c(c(train\_data, train\_labels), c(test\_data, test\_labels)) %<-% imdb

#Vectorize variables  
  
vectorize\_sequences <- function(sequences, dimension = 10000) {  
 # Create an all-zero matrix of shape (len(sequences), dimension)  
 results <- matrix(0, nrow = length(sequences), ncol = dimension)  
 for (i in 1:length(sequences))  
 # Sets specific indices of results[i] to 1s  
 results[i, sequences[[i]]] <- 1  
 results  
}  
  
# Vectorized training data  
x\_train <- vectorize\_sequences(train\_data)  
# Our vectorized test data  
x\_test <- vectorize\_sequences(test\_data)

# Vectorized labels  
y\_train <- as.numeric(train\_labels)  
y\_test <- as.numeric(test\_labels)

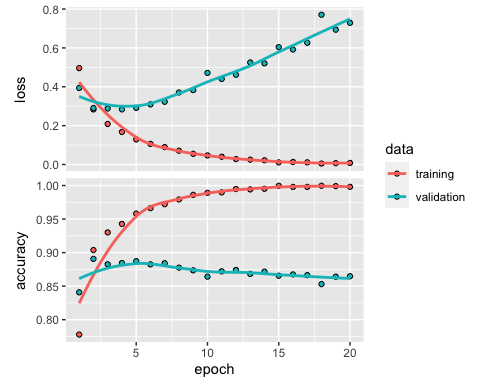
library(keras)  
  
model <- keras\_model\_sequential() %>%   
 layer\_dense(units = 16, activation = "relu", input\_shape = c(10000)) %>%   
 layer\_dense(units = 16, activation = "relu") %>%   
 layer\_dense(units = 1, activation = "sigmoid")

model %>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)

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]

model %>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)  
  
history <- model %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 20,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)

plot(history)



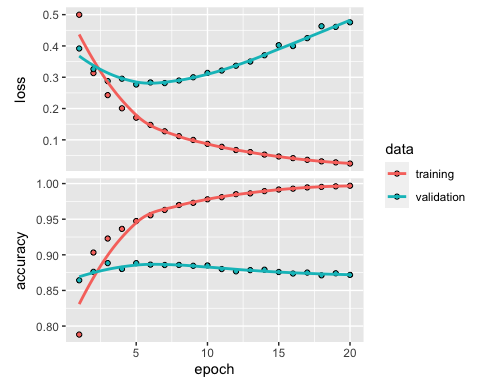
results <- model %>% evaluate(x\_test, y\_test)  
results

## loss accuracy   
## 0.7933121 0.8513200

Question 1: You used two hidden layers. Try using one or three hiddenlayers, and see how doing so affects validation and test accuracy.

model.1layer <- keras\_model\_sequential() %>%   
 layer\_dense(units = 16, activation = "relu", input\_shape = c(10000)) %>%   
 layer\_dense(units = 1, activation = "sigmoid")  
  
model.3layer <- 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")

#Model with one layer  
  
model.1layer %>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)  
  
history.1layer <- model.1layer %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 20,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)  
  
plot(history.1layer)

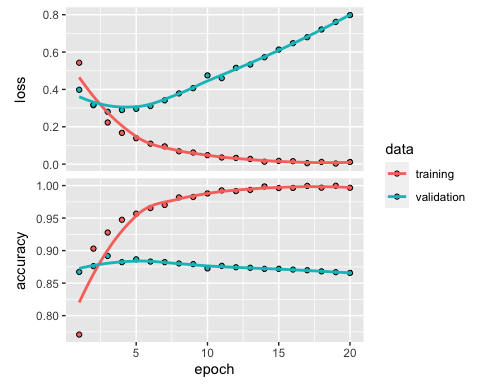


results.1layer <- model.1layer %>% evaluate(x\_test, y\_test)  
results.1layer

## loss accuracy   
## 0.5171219 0.8560800

#Model with one layer  
  
model.3layer %>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)  
  
history.3layer <- model.3layer %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 20,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)

plot(history.3layer)



results.3layer <- model.3layer %>% evaluate(x\_test, y\_test)  
results.3layer

## loss accuracy   
## 0.8703458 0.8494400

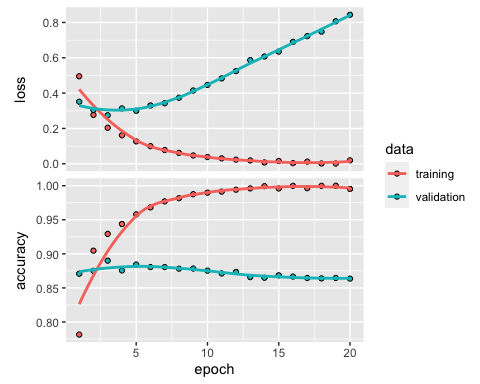
The Number of layers, (1,2 or 3) did not materially alter the accuracy on the validation or test sets. However, during validation, overfitting can be seen as more of a problem as layers ( memory) are increased. Beyond the layers, overfitting can be seen when epochs start to exceed for or five.

Moving forward, I continue various parameters variations on two layers.I will still use 20 epochs in order to continue to see where overfitting is starting / compare to other models.

2.Try using layers with more hidden units or fewer hidden units: 32 units, 64 units, andso on.

model.32u <- keras\_model\_sequential() %>%   
 layer\_dense(units = 32, activation = "relu", input\_shape = c(10000)) %>%   
 layer\_dense(units = 32, activation = "relu") %>%   
 layer\_dense(units = 1, activation = "sigmoid")  
  
  
model.32u %>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)  
  
history.32u <- model.32u %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 20,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)

plot(history.32u)

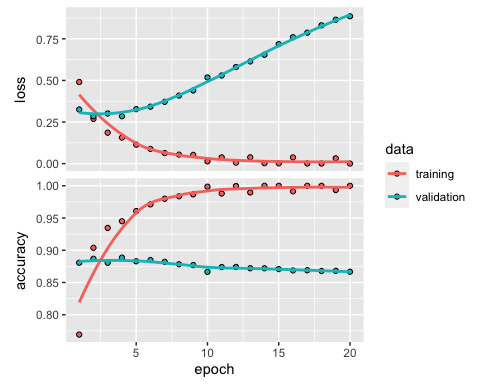


results.32u<- model.32u %>% evaluate(x\_test, y\_test)  
results.32u

## loss accuracy   
## 0.9320678 0.8493200

model.64u <- 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")  
  
  
model.64u %>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)  
  
history.64u <- model.64u %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 20,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)

plot(history.64u)

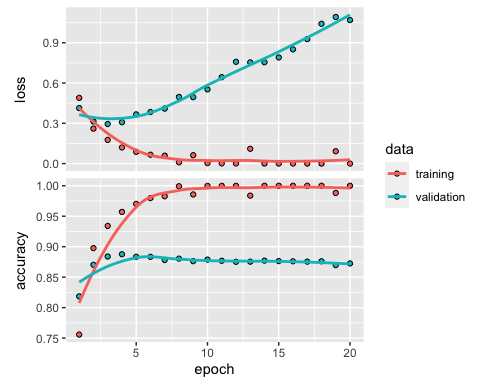


results.64u<- model.64u %>% evaluate(x\_test, y\_test)  
results.64u

## loss accuracy   
## 0.9601758 0.8542000

model.128u <- keras\_model\_sequential() %>%   
 layer\_dense(units = 128, activation = "relu", input\_shape = c(10000)) %>%   
 layer\_dense(units = 128, activation = "relu") %>%   
 layer\_dense(units = 1, activation = "sigmoid")  
  
  
model.128u %>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)  
  
history.128u <- model.128u %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 20,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)

plot(history.128u)



results.128u<- model.128u %>% evaluate(x\_test, y\_test)  
results.128u

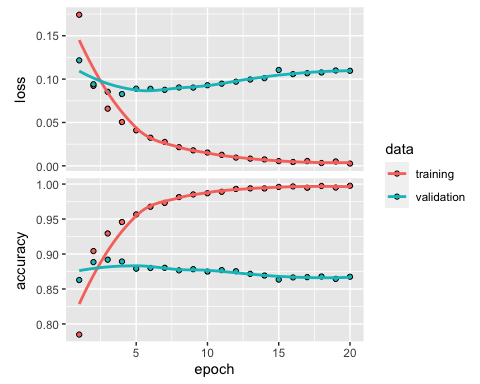
## loss accuracy   
## 1.16621 0.85672

As the more units are added to our to layer model, you can see that validationn accuracy is frequently achieves 100% accuracy as it moves through various epochs. However, again this is likely due to overfitting. This can be seen in results for Test accuracy where 16 units = 85.4%, 32 = 84.5%, 64 = 85.4% and 128 = 85.8%. The accuracy is very similar amongst all test number of hidden units. Due to principles of overfitting and regulaization, I will continue hyper parameter tuning with a model that uses 16 hiden units.

3.Try using the mse loss function instead of binary\_crossentropy.

model.mse <- 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.mse %>% compile(  
 optimizer = "rmsprop",  
 loss = "mse",  
 metrics = c("accuracy")  
)  
  
history.mse <- model.mse %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 20,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)

plot(history.mse)



results.mse<- model.mse %>% evaluate(x\_test, y\_test)  
results.mse

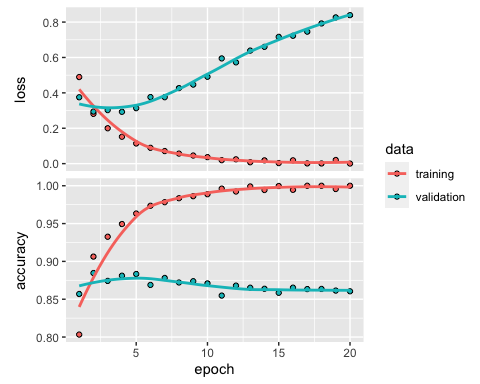
## loss accuracy   
## 0.1219433 0.8518800

There was not a large differece between using the mse or cross enotrpy as the loss function. We still see loss increasing after 5 epochs on the validation set and accuracy declinging slightly.

1. Use the tahn activton function.

model.tanh <- 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")  
  
model.tanh %>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)  
  
history.tanh <- model.tanh %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 20,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)

plot(history.tanh)



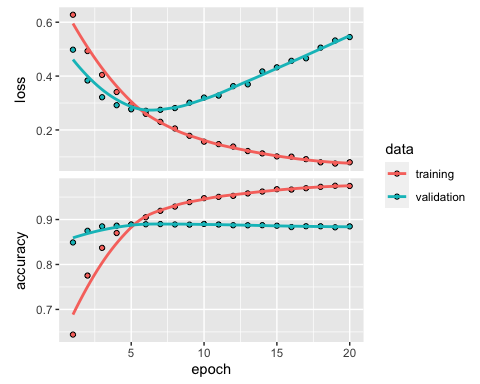
results.tanh<- model.tanh %>% evaluate(x\_test, y\_test)  
results.tanh

## loss accuracy   
## 0.9475284 0.8458400

I will now conduct the drop out method for regularization, which based on the litterature is one of the best methods for model regularization

model.drop.5 <- keras\_model\_sequential() %>%   
 layer\_dense(units = 16, activation = "relu", input\_shape = c(10000)) %>%   
 layer\_dropout(rate = .5) %>%  
 layer\_dense(units = 16, activation = "relu") %>%   
 layer\_dropout(rate = .5) %>%  
 layer\_dense(units = 1, activation = "sigmoid")  
  
  
model.drop.5%>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)  
  
history.drop.5 <- model.drop.5 %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 20,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)

plot(history.drop.5)

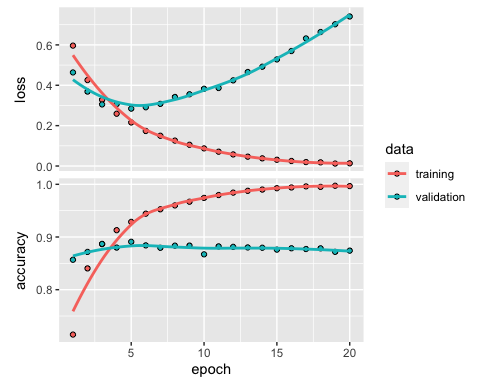


results.drop.5<- model.drop.5 %>% evaluate(x\_test, y\_test)  
results.drop.5

## loss accuracy   
## 0.5821295 0.8743200

model.drop.2 <- keras\_model\_sequential() %>%   
 layer\_dense(units = 16, activation = "relu", input\_shape = c(10000)) %>%   
 layer\_dropout(rate = .2) %>%  
 layer\_dense(units = 16, activation = "relu") %>%   
 layer\_dropout(rate = .2) %>%  
 layer\_dense(units = 1, activation = "sigmoid")  
  
  
model.drop.2%>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)  
  
history.drop.2 <- model.drop.2 %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 20,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)

plot(history.drop.2)



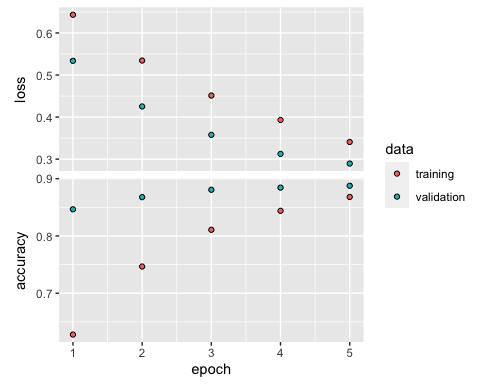
results.drop.2 <- model.drop.2 %>% evaluate(x\_test, y\_test)  
results.drop.2

## loss accuracy   
## 0.8056458 0.8619200

Final Model

model.final <- keras\_model\_sequential() %>%   
 layer\_dense(units = 16, activation = "relu", input\_shape = c(10000)) %>%   
 layer\_dropout(rate = .5) %>%  
 layer\_dense(units = 16, activation = "relu") %>%   
 layer\_dropout(rate = .5) %>%  
 layer\_dense(units = 1, activation = "sigmoid")  
  
  
model.final%>% compile(  
 optimizer = "rmsprop",  
 loss = "binary\_crossentropy",  
 metrics = c("accuracy")  
)  
  
history.final <- model.final %>% fit(  
 partial\_x\_train,  
 partial\_y\_train,  
 epochs = 5,  
 batch\_size = 512,  
 validation\_data = list(x\_val, y\_val)  
)

plot(history.final)



results.final <- model.final %>% evaluate(x\_test, y\_test)  
results.final

## loss accuracy   
## 0.3019246 0.8840000

The accuracy is plotted on the top panel and the loss on the bottom panel. Note that your own results may vary slightly due to a different random initialization of your network.

The dots are the training loss and accuracy, while the solid lines are the validation loss and accuracy. Note that your own results may vary slightly due to a different random initialization of your network.

As you can see, the training loss decreases with every epoch, and the training accuracy increases with every epoch. That’s what you would expect when running a gradient-descent optimization – the quantity you’re trying to minimize should be less with every iteration. But that isn’t the case for the validation loss and accuracy: they seem to peak at the fourth epoch. This is an example of what we warned against earlier: a model that performs better on the training data isn’t necessarily a model that will do better on data it has never seen before. In precise terms, what you’re seeing is *overfitting*: after the second epoch, you’re over-optimizing on the training data, and you end up learning representations that are specific to the training data and don’t generalize to data outside of the training set.

In this case, to prevent overfitting, you could stop training after three epochs. In general, you can use a range of techniques to mitigate overfitting, which we’ll cover in chapter 4.

Let’s train a new network from scratch for four epochs and then evaluate it on the test data.

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   
## 0.319159 0.871240

Our fairly naive approach achieves an accuracy of 88%. With state-of-the-art approaches, one should be able to get close to 95%.

## Using a trained network to generate predictions on new data

After having trained a network, you’ll want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the predict method:

model %>% predict(x\_test[1:10,])

## [,1]  
## [1,] 0.141053110  
## [2,] 0.997071326  
## [3,] 0.504561484  
## [4,] 0.499099582  
## [5,] 0.889461279  
## [6,] 0.654364705  
## [7,] 0.996646821  
## [8,] 0.006696522  
## [9,] 0.928028226  
## [10,] 0.961185634

As you can see, the network is very confident for some samples (0.99 or more, or 0.02 or less) but less confident for others.

## Further experiments

* We were using 2 hidden layers. Try to use 1 or 3 hidden layers and see how it affects validation and test accuracy.
* Try to use layers with more hidden units or less hidden units: 32 units, 64 units…
* Try to use the mse loss function instead of binary\_crossentropy.
* Try to use the tanh activation (an activation that was popular in the early days of neural networks) instead of relu.

These experiments will help convince you that the architecture choices we have made are all fairly reasonable, although they can still be improved!

## Conclusions

Here’s what you should take away from this example:

* You usually need to do quite a bit of preprocessing on your raw data in order to be able to feed it – as tensors – into a neural network. Sequences of words can be encoded as binary vectors, but there are other encoding options, too.
* Stacks of dense layers with relu activations can solve a wide range of problems (including sentiment classification), and you’ll likely use them frequently.
* In a binary classification problem (two output classes), your network should end with a dense layer with one unit and a sigmoid activation. That is, the output of your network should be a scalar between 0 and 1, encoding a probability.
* With such a scalar sigmoid output on a binary classification problem, the loss function you should use is binary\_crossentropy.
* The rmsprop optimizer is generally a good enough choice, whatever your problem. That’s one less thing for you to worry about.
* As they get better on their training data, neural networks eventually start *overfitting* and end up obtaining increasingly worse results on data they’ve never seen before. Be sure to always monitor performance on data that is outside of the training set.