Code

Classifying movie reviews: a binary classification example

The IMDB dataset

We'll be working with "IMDB dataset", a set of 50,000 highly-polarized reviews from the Internet Movie Database. They are split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting in 50% negative and 50% positive reviews.

Hide

```
library(keras)

imdb <- dataset_imdb(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% imdb
```

The argument `num_words = 10000` means that we will only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows us to work with vector data of manageable size.

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":

Hide

```
str(train_data[[1]])
```

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

```
train_labels[[1]]
```

```
[1] 1
```

Since we restricted ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:

Hide

```
max(sapply(train_data, max))
```

```
[1] 9999
```

For kicks, here's how you can quickly decode one of these reviews back to English words:

Hide

```
# word_index is a dictionary mapping words to an integer index
word_index <- dataset_imdb_word_index()
# We reverse it, mapping integer indices to words
reverse_word_index <- names(word_index)
names(reverse_word_index) <- word_index
# We decode the review; note that our indices were offset by 3
# because 0, 1 and 2 are reserved indices for "padding", "start of sequence", and "unknown".
decoded_review <- sapply(train_data[[1]], function(index) {
  word <- if (index >= 3) reverse_word_index[[as.character(index - 3)]]
  if (!is.null(word)) word else "?"
})
```

```
cat(decoded_review)
```

```
? this film was just brilliant casting location scenery story direction everyone's really suited the part they played and yo
u could just imagine being there robert ? is an amazing actor and now the same being director ? father came from the same sc
ottish island as myself so i loved the fact there was a real connection with this film the witty remarks throughout the film
were great it was just brilliant so much that i bought the film as soon as it was released for ? and would recommend it to e
veryone to watch and the fly fishing was amazing really cried at the end it was so sad and you know what they say if you cry
at a film it must have been good and this definitely was also ? to the two little boy's that played the ? of norman and paul
they were just brilliant children are often left out of the ? list i think because the stars that play them all grown up are
such a big profile for the whole film but these children are amazing and should be praised for what they have done don't you
think the whole story was so lovely because it was true and was someone's life after all that was shared with us all
```

Preparing the data

You can't feed lists of integers into a neural network. You have to turn your lists into tensors.

- One-hot-encode your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence `[3, 5]` into a 10,000-dimensional vector that would be all zeros except for indices 3 and 5, which would be ones. Then you could use as the first layer in your network a dense layer, capable of handling floating-point vector data.

```
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
}

# Our vectorized training data
x_train <- vectorize_sequences(train_data)
# Our vectorized test data
x_test <- vectorize_sequences(test_data)
```

Here's what our samples look like now:

Hide

```
str(x_train[1,])
```

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

We should also vectorize our labels, which is straightforward:

Hide

```
# Our vectorized labels
y_train <- as.numeric(train_labels)
y_test <- as.numeric(test_labels)
```

Building our network

There are two key architecture decisions to be made about such stack of dense layers:

- How many layers to use.
- How many "hidden units" to chose for each layer.

In the following architecture: three intermediate layers with 64 hidden units each, and a third layer which will output the scalar prediction regarding the sentiment of the current review. The intermediate layers will use `tanh` as their "activation function", and the final layer will use a sigmoid activation so as to output a probability (a score between 0 and 1, indicating how likely the sample is to have the target "1", i.e. how likely the review is to be positive). Here's what our network looks like:

Hide

```
library(keras)

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

Lastly, we need to pick a loss function and an optimizer. Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it best to use the `binary_crossentropy` loss. It isn't the only viable choice: you could use, for instance, `mean_squared_error`. But crossentropy is usually the best choice when you are dealing with models that output probabilities. Crossentropy is a quantity from the field of Information Theory, that measures the "distance" between probability distributions, or in our case, between the ground-truth distribution and our predictions.

Here's the step where we configure our model with the `rmsprop` optimizer and the `mean_squared_error` loss function.

Hide

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

You're passing optimizer, loss function, and metrics as strings, which is possible because `rmsprop`, `binary_crossentropy`, and `accuracy` are packaged as part of Keras.

Hide

```
model %>% compile(
  optimizer = optimizer_rmsprop(lr=0.001),
  loss = "mse",
  metrics = c("accuracy")
)
```

Validating our approach

In order to monitor during training the accuracy of the model on data that it has never seen before, we will create a "validation set" by setting apart 10,000 samples from the original training data:

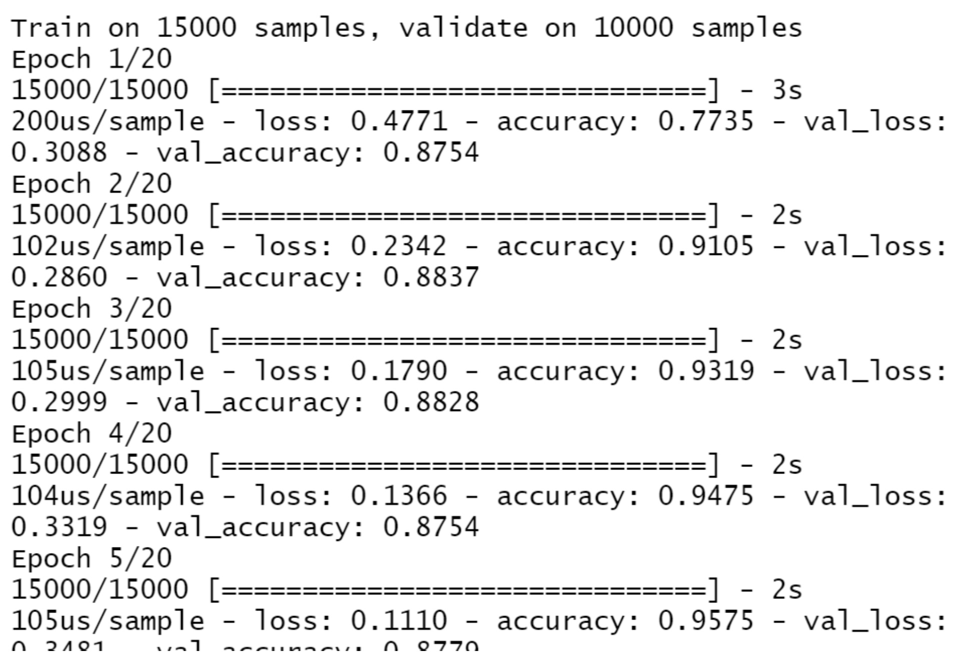
```r
val_indices <- 1:10000

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]
```

We will now train our model for 20 epochs (20 iterations over all samples in the `x_train` and `y_train` tensors), in mini-batches of 512 samples. At this same time we will monitor loss and accuracy on the 10,000 samples that we set apart. This is done by passing the validation data as the `validation_data` argument:

Hide

```r
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)
)
```

we call `fit()` returns a `history` object. Let's take a look at it:



```
Train on 15000 samples, validate on 10000 samples
Epoch 1/20
15000/15000 [==============================] - 3s
200us/sample - loss: 0.4771 - accuracy: 0.7735 - val_loss:
0.3088 - val_accuracy: 0.8754
Epoch 2/20
15000/15000 [==============================] - 2s
102us/sample - loss: 0.2342 - accuracy: 0.9105 - val_loss:
0.2860 - val_accuracy: 0.8837
Epoch 3/20
15000/15000 [==============================] - 2s
105us/sample - loss: 0.1790 - accuracy: 0.9319 - val_loss:
0.2999 - val_accuracy: 0.8828
Epoch 4/20
15000/15000 [==============================] - 2s
104us/sample - loss: 0.1366 - accuracy: 0.9475 - val_loss:
0.3319 - val_accuracy: 0.8754
Epoch 5/20
15000/15000 [==============================] - 2s
105us/sample - loss: 0.1110 - accuracy: 0.9575 - val_loss:
0.3481 - val_accuracy: 0.8779
```

```
str(history)
```

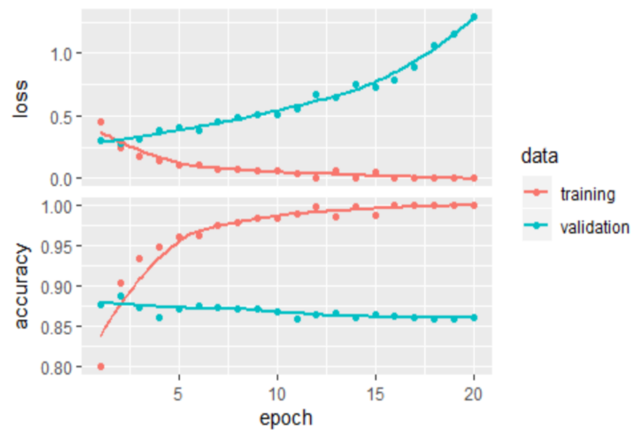
```
List of 2
 $ params :List of 7
  ..$ batch_size   : int 512
  ..$ epochs       : int 20
  ..$ steps        : num 30
  ..$ samples      : int 15000
  ..$ verbose      : int 0
  ..$ do_validation: logi TRUE
  ..$ metrics      : chr [1:4] "loss" "accuracy" "val_loss" "val_accuracy"
 $ metrics:List of 4
  ..$ loss        : num [1:20] 0.448 0.245 0.175 0.141 0.105 ...
  ..$ accuracy    : num [1:20] 0.799 0.904 0.934 0.949 0.962 ...
  ..$ val_loss    : num [1:20] 0.303 0.278 0.318 0.384 0.401 ...
  ..$ val_accuracy: num [1:20] 0.878 0.887 0.874 0.861 0.872 ...
 - attr(*, "class")= chr "keras_training_history"
```

The `history` object includes various parameters used to fit the model (`history$params`) as well as data for each of the metrics being monitored (`history$metrics`).

The `history` object has a `plot()` method that enables us to visualize the training and validation metrics by epoch:

Hide

```
plot(history)
```



The accuracy is plotted on the top panel and the loss on the bottom panel.

The dots are the training loss and accuracy, while the solid lines are the validation loss and accuracy.

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.

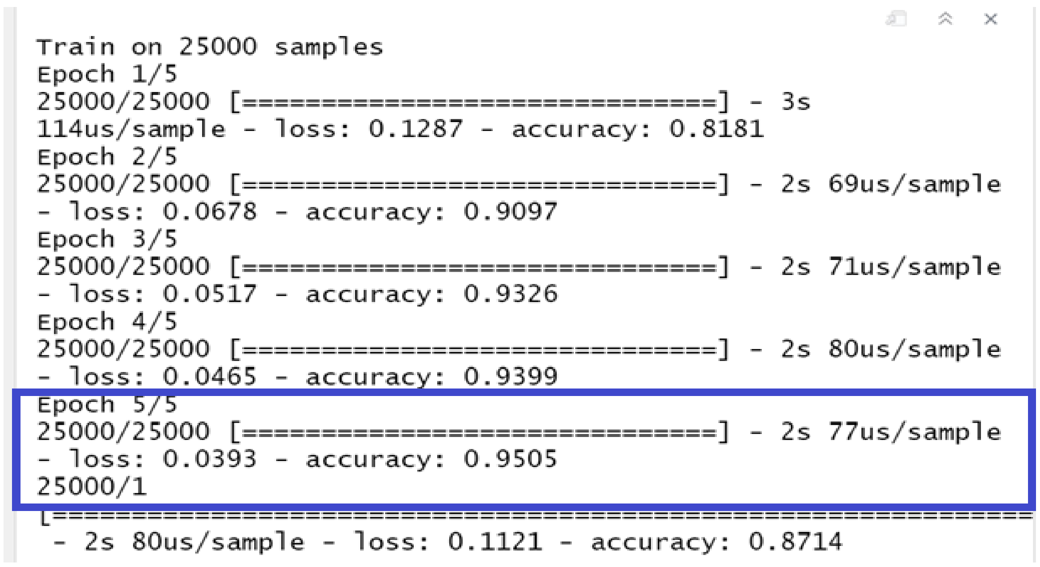
In this case, to prevent overfitting, you could stop training after three epochs. In general, you can use a range of techniques to mitigate overfitting,

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

Hyperparameter		Validation Accuracy
Number of hidden layers	3	.8754
Numbers of Units	64,64,64	.8837
Activation Function	Tanh	.8828
Epoch	20	.8754
Learning Rate	0.001	
Batch Size	512	

```
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "tanh", input_shape = c(10000)) %>%
  layer_dense(units = 32, activation = "tanh")%>%
  layer_dense(units = 32, activation = "tanh") %>%
  layer_dense(units = 1, activation = "sigmoid")

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

model %>% fit(x_train, y_train, epochs = 5, batch_size = 512)
results <- model %>% evaluate(x_test, y_test)
```

Hide

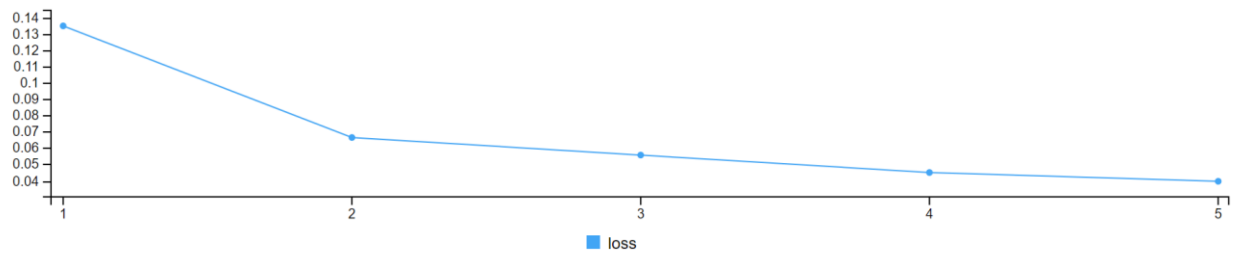


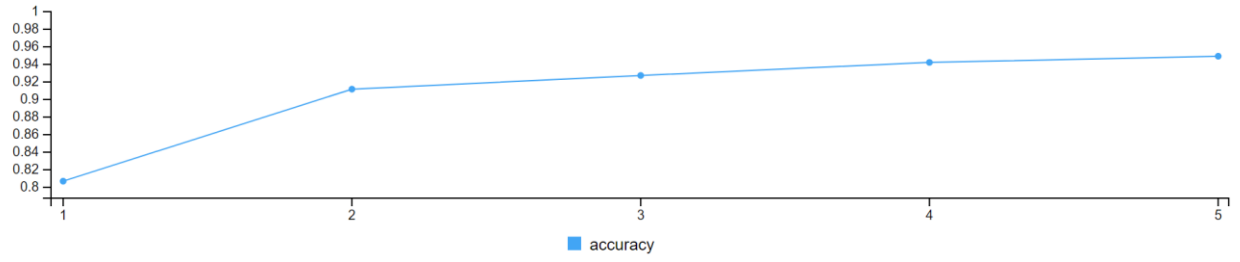
```
Train on 25000 samples
Epoch 1/5
25000/25000 [==============================] - 3s
114us/sample - loss: 0.1287 - accuracy: 0.8181
Epoch 2/5
25000/25000 [==============================] - 2s 69us/sample
- loss: 0.0678 - accuracy: 0.9097
Epoch 3/5
25000/25000 [==============================] - 2s 71us/sample
- loss: 0.0517 - accuracy: 0.9326
Epoch 4/5
25000/25000 [==============================] - 2s 80us/sample
- loss: 0.0465 - accuracy: 0.9399
Epoch 5/5
25000/25000 [==============================] - 2s 77us/sample
- loss: 0.0393 - accuracy: 0.9505
25000/1
[=============================================================
 - 2s 80us/sample - loss: 0.1121 - accuracy: 0.8714
```

Epoch5 Accuracy is max, and loss is least which is .0393 after epoch 5 accuracy keeps decreasing

```
results
```

```
$loss
[1] 0.09937261

$accuracy
[1] 0.87152
```





Hyperparameter		Accuracy	Testing Accuracy
Number of hidden layers	3	.8181	87.15
Numbers of Units	64,32,32	.9097	
Activation Function	Tanh	.9326	
Epoch	5	.9399	
Learning Rate	0.001	.9505	
Batch Size	512		

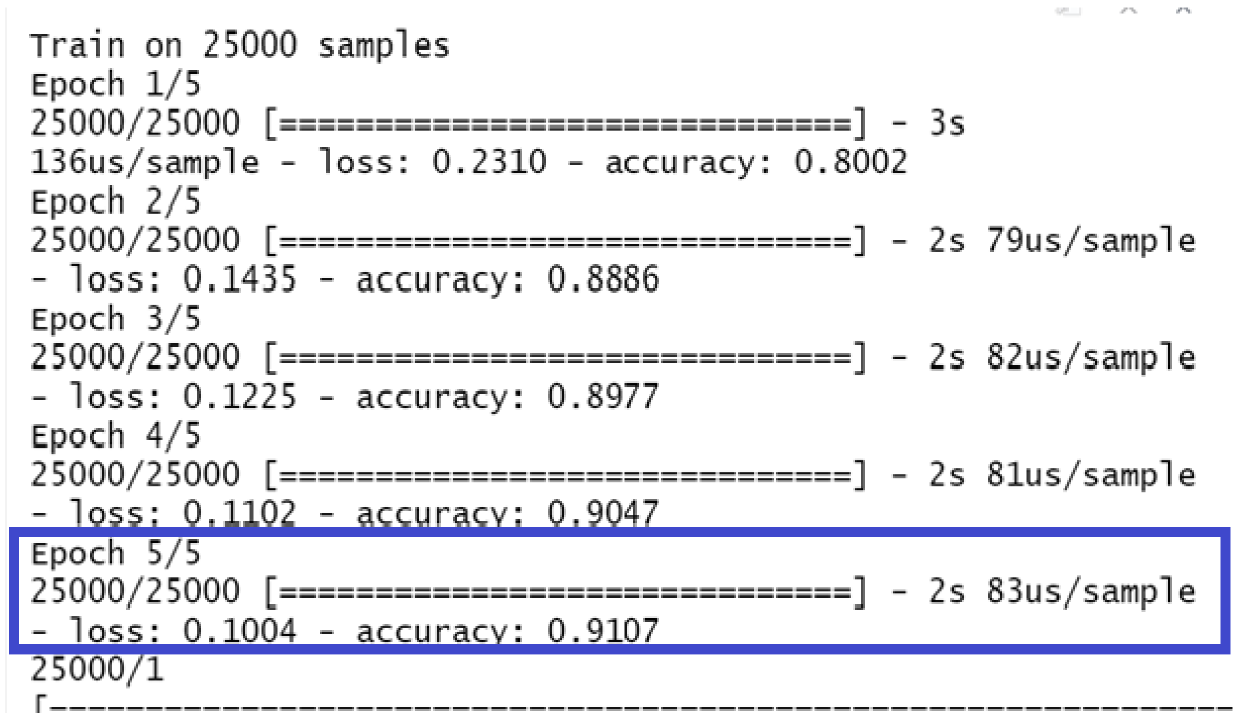
Add regularization

```
model <- keras_model_sequential() %>%
  layer_dense(units = 64, kernel_regularizer = regularizer_l2(0.001), activation = "tanh", input_shape = c(10000)) %>%
  layer_dense(units = 32, kernel_regularizer = regularizer_l2(0.001), activation = "tanh", input_shape = c(10000)) %>%
  layer_dense(units = 1, activation = "sigmoid")

final<-model %>% compile(
  optimizer = "rmsprop",
  loss = "mse",
  metrics = c("accuracy")
)
model %>% fit(x_train, y_train, epochs = 5, batch_size = 512)
results <- model %>% evaluate(x_test, y_test)
```

Hide

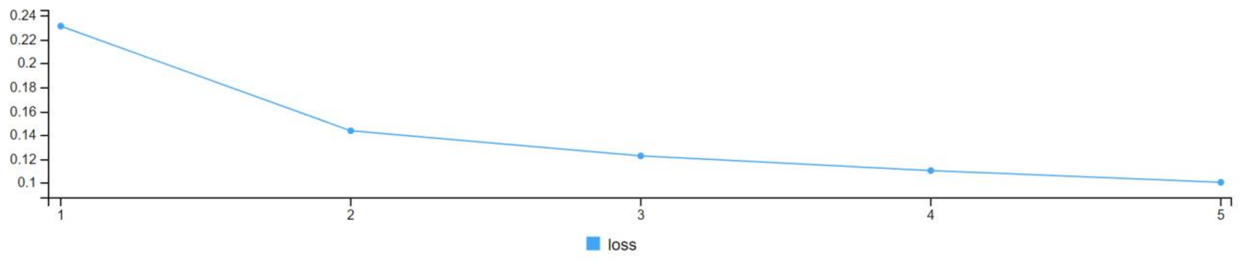
```
results
```

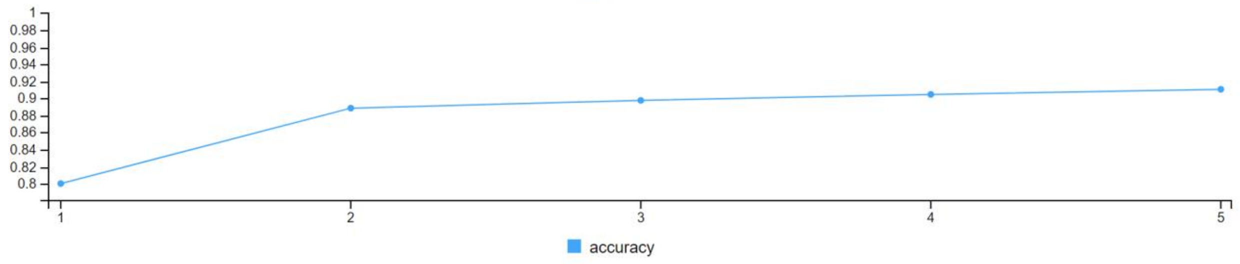
```
$loss
[1] 0.1143071

$accuracy
[1] 0.8836
```



```
Train on 25000 samples
Epoch 1/5
25000/25000 [==============================] - 3s
136us/sample - loss: 0.2310 - accuracy: 0.8002
Epoch 2/5
25000/25000 [==============================] - 2s 79us/sample
- loss: 0.1435 - accuracy: 0.8886
Epoch 3/5
25000/25000 [==============================] - 2s 82us/sample
- loss: 0.1225 - accuracy: 0.8977
Epoch 4/5
25000/25000 [==============================] - 2s 81us/sample
- loss: 0.1102 - accuracy: 0.9047
Epoch 5/5
25000/25000 [==============================] - 2s 83us/sample
- loss: 0.1004 - accuracy: 0.9107
25000/1
[--------------------------------------------------------------
```

Epoch5 Accuracy is max, and loss is least which is .1004





Hyperparameter		Accuracy	Testing Accuracy
Number of hidden layers	2	.8002	88.36
Numbers of Units	64,32	.8886	
Activation Function	Tanh	.8977	
Epoch	5	.9047	
Learning Rate	0.001	.9107	
Batch Size	512		
Regularization			

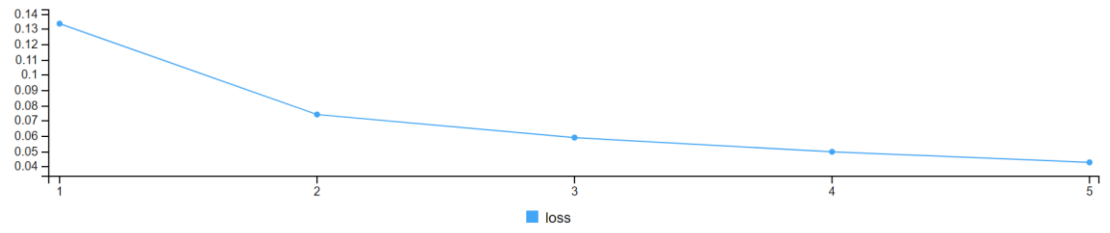
Add Dropout

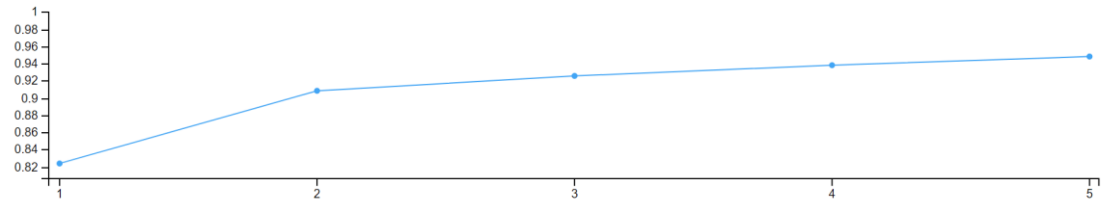
```
model <- keras_model_sequential() %>%
  layer_dense(units = 64,  activation = "tanh", input_shape =
c(10000)) %>%
  layer_dropout(rate = 0.5)%>%

  layer_dense(units = 1, activation = "sigmoid")

final<-model %>% compile(
  optimizer = "rmsprop",
  loss = "mse",
  metrics = c("accuracy")
)
model %>% fit(x_train, y_train, epochs = 5, batch_size = 512)

results <- model %>% evaluate(x_test, y_test)
```

```
Train on 25000 samples
Epoch 1/5
25000/25000 [==============================] - 5s
194us/sample - loss: 0.1334 - accuracy: 0.8238
Epoch 2/5
25000/25000 [==============================] - 3s
128us/sample - loss: 0.0740 - accuracy: 0.9084
Epoch 3/5
25000/25000 [==============================] - 3s
128us/sample - loss: 0.0589 - accuracy: 0.9257
Epoch 4/5
25000/25000 [==============================] - 3s
126us/sample - loss: 0.0496 - accuracy: 0.9382
Epoch 5/5
25000/25000 [==============================] - 3s
126us/sample - loss: 0.0427 - accuracy: 0.9483
25000/1
[============================================================
```

```
results
```

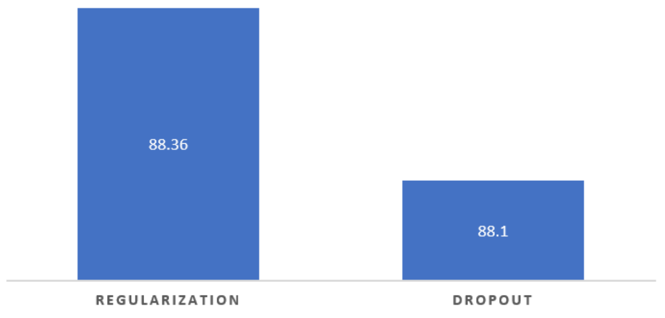
```
$loss
[1] 0.08839746

$accuracy
[1] 0.88108
```





Hyperparameter		Accuracy	Testing Accuracy
Number of hidden layers	2	.8238	88.10
Numbers of Units	64,32	.9084	
Activation Function	Tanh	.9257	
Epoch	5	.9382	
Learning Rate	0.001	.9482	
Batch Size	512		
Dropout			

Model Tuning (dropout/ regularization)



- Model performs better when Regularization is applied.