AML1

Mark Christian

2/22/2021

```
#install.packages("keras")
#install_keras(tensorflow = "gpu")
memory.limit(24000)
```

#using 1000 to keep limited word count and make sure that R does not crash

```
library(keras)
```

```
## Warning: package 'keras' was built under R version 4.0.3
```

```
library(tensorflow)
imdb <- dataset_imdb(num_words = 1000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% imdb
```

#Vectorizing the data here

```
vectorize_sequences <- function(sequences, dimension = 1000) {
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)
y_train <- as.numeric(train_labels)
y_test <- as.numeric(test_labels)
```

#2 layers and 16 units result. Binary and relu

```
library(tensorflow)
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(1000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

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

#now lets validate and then see the results

```
val_indices <- 1:1000
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 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results <- model %>% evaluate(x_test, y_test)
results
```

```
##      loss  accuracy
## 0.3272121 0.8609600
```

#using 3 hidden layers with 16 units and comparing the results. binary and relu

```
model1 <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(1000)) %>%
  layer_dense(units = 16, activation = "relu") %>% layer_dense(units = 16, activation = "relu") %>% lay
```

```
model1 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy"))
```

#validating and results

```
val_indices1 <- 1:1000
x_val1 <- x_train[val_indices,]
partial_x_train1 <- x_train[-val_indices,]
y_val1 <- y_train[val_indices]
partial_y_train1 <- y_train[-val_indices]

model1 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results1 <- model1 %>% evaluate(x_test, y_test)
results1
```

```
##      loss  accuracy
## 0.3308595 0.8599200
```

#using 2 layers, the accuracy is 0.6674 while 3 hidden layers end up being better and more accurate with 0.8580

#Try using layers with more hidden units or fewer hidden units: 32 units, 64 units, and so on. Here I will use 64 hidden units

#binary and relu 64 units

```
model3 <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = c(1000)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

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

#lets validate and see the result

```
val_indices <- 1:1000
x_val3 <- x_train[val_indices,]
partial_x_train3 <- x_train[-val_indices,]
y_val3 <- y_train[val_indices]
partial_y_train3 <- y_train[-val_indices]

model3 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results3 <- model3 %>% evaluate(x_test, y_test)
results3
```

```
##      loss  accuracy
## 0.3240926 0.8598400
```

#the accuracy using 64 hidden units is 0.8551, loss was 0.3369

#Try using the mse loss function instead of binary_crossentropy. and seeing the results after validating. Hidden value used here is 64

```
model5 <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = c(1000)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

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

val_indices <- 1:1000
x_val5 <- x_train[val_indices,]
partial_x_train5 <- x_train[-val_indices,]
y_val5 <- y_train[val_indices]
partial_y_train5 <- y_train[-val_indices]

model5 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results5 <- model5 %>% evaluate(x_test, y_test)
results5
```

```
##      loss  accuracy
## 0.1005371 0.8596000
```

#mse and relu is 0.8596 accuracy and 0.10 loss which is better than binary and relu(0.8551 and 0.3369)

#Try using the tanh activation (an activation that was popular in the early days of neural networks) instead of relu units= 64

```
model6 <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "tanh", input_shape = c(1000)) %>%
  layer_dense(units = 64, activation = "tanh") %>%
  layer_dense(units = 1, activation = "sigmoid")

model6 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy"))

val_indices <- 1:1000
x_val6 <- x_train[val_indices,]
partial_x_train6 <- x_train[-val_indices,]
y_val6 <- y_train[val_indices]
partial_y_train6 <- y_train[-val_indices]

model6 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results6 <- model6 %>% evaluate(x_test, y_test)
results6
```

```
##      loss  accuracy
## 0.3220223 0.8596400
```

#using tanh instead of relu, the accuracy is 0.8615 and loss is 0.3196 while the accuray was 0.8551 with the loss of 0.3368.

#the highest accuracy was tanh as the activation and binary crossentropy as the loss. when I kept the hidden units, 64 units. the accuracy was 0.8615

#Regulization tanh and binary while hidden unit will be 64.

```
set.seed(123)
model7 <- keras_model_sequential() %>%
  layer_dense(units = 64, kernel_regularizer = regularizer_l2(0.001), activation = "tanh", input_shape =
  layer_dense(units = 64, kernel_regularizer = regularizer_l2(0.001), activation = "tanh") %>%
  layer_dense(units = 1, activation = "sigmoid")

model7 %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy"))

val_indices <- 1:1000
x_val7 <- x_train[val_indices,]
partial_x_train7 <- x_train[-val_indices,]
y_val7 <- y_train[val_indices]
partial_y_train7 <- y_train[-val_indices]

model7 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results7 <- model7 %>% evaluate(x_test, y_test)
results7
```

```
##      loss  accuracy
## 0.4217378 0.8596800
```

#I am also going to make another model with relu and mean_squared_error because the loss was so less just so compare the results.

```
set.seed(123)
model8 <- keras_model_sequential() %>%
  layer_dense(units = 64, kernel_regularizer = regularizer_l2(0.001), activation = "relu", input_shape =
  layer_dense(units = 64, kernel_regularizer = regularizer_l2(0.001), activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model8 %>% compile(
  optimizer = "rmsprop",
  loss = "mean_squared_error",
  metrics = c("accuracy"))

val_indices <- 1:1000
x_val8 <- x_train[val_indices,]
partial_x_train8 <- x_train[-val_indices,]
y_val8 <- y_train[val_indices]
partial_y_train8 <- y_train[-val_indices]

model8 %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results8 <- model8 %>% evaluate(x_test, y_test)
results8
```

```
##      loss  accuracy
## 0.1580272 0.8562000
```

#I would rather go with MSE and Relu as the model with least loss and comparatively higher number than other.Also, tanh is an old method and which means that there had to me more nugs and issues related to it.