In this post we will use Keras to classify duplicated questions from Quora. The dataset first appeared in the Kaggle competition [Quora Question Pairs](https://www.kaggle.com/c/quora-question-pairs) and consists of approximately 400,000 pairs of questions along with a column indicating if the question pair is considered a duplicate.

Our implementation is inspired by the [Siamese Recurrent Architecture](https://dl.acm.org/citation.cfm?id=3016291), with modifications to the similarity measure and the embedding layers (the original paper uses pre-trained word vectors). The idea is to learn a function that maps input patterns into a target space such that a similarity measure in the target space approximates the “semantic” distance in the input space.

**Dowloading data**

Data can be downloaded from the Kaggle [dataset webpage](https://www.kaggle.com/quora/question-pairs-dataset) or from Quora’s [release of the dataset](https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs):

library(keras)

quora\_data <- get\_file(

"quora\_duplicate\_questions.tsv",

"https://qim.ec.quoracdn.net/quora\_duplicate\_questions.tsv"

)

We are using the Keras get\_file() function so that the file download is cached.

**Reading and preprocessing**

We will first load data into R and do some preprocessing to make it easier to include in the model. After downloading the data, you can read it using the readr read\_tsv() function.

library(readr)

df <- read\_tsv(quora\_data)

We will create a Keras tokenizer to transform each word into an integer token. We will also specify a hyperparameter of our model: the vocabulary size. For now let’s use the 50,000 most common words (we’ll tune this parameter later). The tokenizer will be fit using all unique questions from the dataset.

tokenizer <- text\_tokenizer(num\_words = 50000)

tokenizer %>% fit\_text\_tokenizer(unique(c(df$question1, df$question2)))

Let’s save the tokenizer to disk in order to use it for inference later.

save\_text\_tokenizer(tokenizer, "tokenizer-question-pairs")

We will now use the text tokenizer to transform each question into a list of integers.

question1 <- texts\_to\_sequences(tokenizer, df$question1)

question2 <- texts\_to\_sequences(tokenizer, df$question2)

Let’s take a look at the number of words in each question. This will helps us to decide the padding length, another hyperparameter of our model. Padding the sequences normalizes them to the same size so that we can feed them to the Keras model.

library(purrr)

questions\_length <- c(

map\_int(question1, length),

map\_int(question2, length)

)

quantile(questions\_length, c(0.8, 0.9, 0.95, 0.99))

80% 90% 95% 99%

14 18 23 31

We can see that 99% of questions have at most length 31 so we’ll choose a padding length between 15 and 30. Let’s start with 20 (we’ll also tune this parameter later). The default padding value is 0, but we are already using this value for words that don’t appear within the 50,000 most frequent, so we’ll use 50,001 instead.

question1\_padded <- pad\_sequences(question1, maxlen = 20, value = 50000 + 1)

question2\_padded <- pad\_sequences(question2, maxlen = 20, value = 50000 + 1)

We have now finished the preprocessing steps. We will now run a simple benchmark model before moving on to the Keras model.

**Simple benchmark**

Before creating a complicated model let’s take a simple approach. Let’s create two predictors: percentage of words from question1 that appear in the question2 and vice-versa. Then we will use a logistic regression to predict if the questions are duplicate.

perc\_words\_question1 <- map2\_dbl(question1, question2, ~mean(.x %in% .y))

perc\_words\_question2 <- map2\_dbl(question2, question1, ~mean(.x %in% .y))

df\_model <- data.frame(

perc\_words\_question1 = perc\_words\_question1,

perc\_words\_question2 = perc\_words\_question2,

is\_duplicate = df$is\_duplicate

) %>%

na.omit()

Now that we have our predictors, let’s create the logistic model. We will take a small sample for validation.

val\_sample <- sample.int(nrow(df\_model), 0.1\*nrow(df\_model))

logistic\_regression <- glm(

is\_duplicate ~ perc\_words\_question1 + perc\_words\_question2,

family = "binomial",

data = df\_model[-val\_sample,]

)

summary(logistic\_regression)

Call:

glm(formula = is\_duplicate ~ perc\_words\_question1 + perc\_words\_question2,

family = "binomial", data = df\_model[-val\_sample, ])

Deviance Residuals:

Min 1Q Median 3Q Max

-1.5938 -0.9097 -0.6106 1.1452 2.0292

Coefficients:

Estimate Std. Error z value Pr(>|z|)

(Intercept) -2.259007 0.009668 -233.66 <2e-16 \*\*\*

perc\_words\_question1 1.517990 0.023038 65.89 <2e-16 \*\*\*

perc\_words\_question2 1.681410 0.022795 73.76 <2e-16 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 479158 on 363843 degrees of freedom

Residual deviance: 431627 on 363841 degrees of freedom

(17 observations deleted due to missingness)

AIC: 431633

Number of Fisher Scoring iterations: 3

Let’s calculate the accuracy on our validation set.

pred <- predict(logistic\_regression, df\_model[val\_sample,], type = "response")

pred <- pred > mean(df\_model$is\_duplicate[-val\_sample])

accuracy <- table(pred, df\_model$is\_duplicate[val\_sample]) %>%

prop.table() %>%

diag() %>%

sum()

accuracy

[1] 0.6573577

We got an accuracy of 65.7%. Not all that much better than random guessing. Now let’s create our model in Keras.

**Model definition**

We will use a Siamese network to predict whether the pairs are duplicated or not. The idea is to create a model that can embed the questions (sequence of words) into a vector. Then we can compare the vectors for each question using a similarity measure and tell if the questions are duplicated or not.

First let’s define the inputs for the model.

input1 <- layer\_input(shape = c(20), name = "input\_question1")

input2 <- layer\_input(shape = c(20), name = "input\_question2")

Then let’s the define the part of the model that will embed the questions in a vector.

word\_embedder <- layer\_embedding(

input\_dim = 50000 + 2, # vocab size + UNK token + padding value

output\_dim = 128, # hyperparameter - embedding size

input\_length = 20, # padding size,

embeddings\_regularizer = regularizer\_l2(0.0001) # hyperparameter - regularization

)

seq\_embedder <- layer\_lstm(

units = 128, # hyperparameter -- sequence embedding size

kernel\_regularizer = regularizer\_l2(0.0001) # hyperparameter - regularization

)

Now we will define the relationship between the input vectors and the embeddings layers. Note that we use the same layers and weights on both inputs. That’s why this is called a Siamese network. It makes sense, because we don’t want to have different outputs if question1 is switched with question2.

vector1 <- input1 %>% word\_embedder() %>% seq\_embedder()

vector2 <- input2 %>% word\_embedder() %>% seq\_embedder()

We then define the similarity measure we want to optimize. We want duplicated questions to have higher values of similarity. In this example we’ll use the cosine similarity, but any similarity measure could be used. Remember that the cosine similarity is the normalized dot product of the vectors, but for training it’s not necessary to normalize the results.

cosine\_similarity <- layer\_dot(list(vector1, vector2), axes = 1)

Next, we define a final sigmoid layer to output the probability of both questions being duplicated.

output <- cosine\_similarity %>%

layer\_dense(units = 1, activation = "sigmoid")

Now that let’s define the Keras model in terms of it’s inputs and outputs and compile it. In the compilation phase we define our loss function and optimizer. Like in the Kaggle challenge, we will minimize the logloss (equivalent to minimizing the binary crossentropy). We will use the Adam optimizer.

model <- keras\_model(list(input1, input2), output)

model %>% compile(

optimizer = "adam",

metrics = list(acc = metric\_binary\_accuracy),

loss = "binary\_crossentropy"

)

We can then take a look at out model with the summary function.

summary(model)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param # Connected to

=======================================================================================

input\_question1 (InputLayer (None, 20) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

input\_question2 (InputLayer (None, 20) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

embedding\_1 (Embedding) (None, 20, 128) 6400256 input\_question1[0][0]

input\_question2[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lstm\_1 (LSTM) (None, 128) 131584 embedding\_1[0][0]

embedding\_1[1][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dot\_1 (Dot) (None, 1) 0 lstm\_1[0][0]

lstm\_1[1][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 1) 2 dot\_1[0][0]

=======================================================================================

Total params: 6,531,842

Trainable params: 6,531,842

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Model fitting**

Now we will fit and tune our model. However before proceeding let’s take a sample for validation.

set.seed(1817328)

val\_sample <- sample.int(nrow(question1\_padded), size = 0.1\*nrow(question1\_padded))

train\_question1\_padded <- question1\_padded[-val\_sample,]

train\_question2\_padded <- question2\_padded[-val\_sample,]

train\_is\_duplicate <- df$is\_duplicate[-val\_sample]

val\_question1\_padded <- question1\_padded[val\_sample,]

val\_question2\_padded <- question2\_padded[val\_sample,]

val\_is\_duplicate <- df$is\_duplicate[val\_sample]

Now we use the fit() function to train the model:

model %>% fit(

list(train\_question1\_padded, train\_question2\_padded),

train\_is\_duplicate,

batch\_size = 64,

epochs = 10,

validation\_data = list(

list(val\_question1\_padded, val\_question2\_padded),

val\_is\_duplicate

)

)

Train on 363861 samples, validate on 40429 samples

Epoch 1/10

363861/363861 [==============================] - 89s 245us/step - loss: 0.5860 - acc: 0.7248 - val\_loss: 0.5590 - val\_acc: 0.7449

Epoch 2/10

363861/363861 [==============================] - 88s 243us/step - loss: 0.5528 - acc: 0.7461 - val\_loss: 0.5472 - val\_acc: 0.7510

Epoch 3/10

363861/363861 [==============================] - 88s 242us/step - loss: 0.5428 - acc: 0.7536 - val\_loss: 0.5439 - val\_acc: 0.7515

Epoch 4/10

363861/363861 [==============================] - 88s 242us/step - loss: 0.5353 - acc: 0.7595 - val\_loss: 0.5358 - val\_acc: 0.7590

Epoch 5/10

363861/363861 [==============================] - 88s 242us/step - loss: 0.5299 - acc: 0.7633 - val\_loss: 0.5358 - val\_acc: 0.7592

Epoch 6/10

363861/363861 [==============================] - 88s 242us/step - loss: 0.5256 - acc: 0.7662 - val\_loss: 0.5309 - val\_acc: 0.7631

Epoch 7/10

363861/363861 [==============================] - 88s 242us/step - loss: 0.5211 - acc: 0.7701 - val\_loss: 0.5349 - val\_acc: 0.7586

Epoch 8/10

363861/363861 [==============================] - 88s 242us/step - loss: 0.5173 - acc: 0.7733 - val\_loss: 0.5278 - val\_acc: 0.7667

Epoch 9/10

363861/363861 [==============================] - 88s 242us/step - loss: 0.5138 - acc: 0.7762 - val\_loss: 0.5292 - val\_acc: 0.7667

Epoch 10/10

363861/363861 [==============================] - 88s 242us/step - loss: 0.5092 - acc: 0.7794 - val\_loss: 0.5313 - val\_acc: 0.7654

After training completes, we can save our model for inference with the save\_model\_hdf5() function.

save\_model\_hdf5(model, "model-question-pairs.hdf5")

**Model tuning**

Now that we have a reasonable model, let’s tune the hyperparameters using the tfruns package. We’ll begin by adding FLAGS declarations to our script for all hyperparameters we want to tune (FLAGS allow us to vary hyperparmaeters without changing our source code):

Library(tfruns)

FLAGS <- [flags](https://tensorflow.rstudio.com/keras/reference/reexports.html)(

flag\_integer("vocab\_size", 50000),

flag\_integer("max\_len\_padding", 20),

flag\_integer("embedding\_size", 256),

flag\_numeric("regularization", 0.0001),

flag\_integer("seq\_embedding\_size", 512)

)

With this FLAGS definition we can now write our code in terms of the flags. For example:

input1 <- layer\_input(shape = c(FLAGS$max\_len\_padding))

input2 <- layer\_input(shape = c(FLAGS$max\_len\_padding))

embedding <- layer\_embedding(

input\_dim = FLAGS$vocab\_size + 2,

output\_dim = FLAGS$embedding\_size,

input\_length = FLAGS$max\_len\_padding,

embeddings\_regularizer = regularizer\_l2(l = FLAGS$regularization)

)

The full source code of the script with FLAGS can be found below:

|  |
| --- |
| library(readr) |
|  | library(keras) |
|  | library(purrr) |
|  |  |
|  | FLAGS <- flags( |
|  | flag\_integer("vocab\_size", 50000), |
|  | flag\_integer("max\_len\_padding", 20), |
|  | flag\_integer("embedding\_size", 256), |
|  | flag\_numeric("regularization", 0.0001), |
|  | flag\_integer("seq\_embedding\_size", 512) |
|  | ) |
|  |  |
|  | df <- read\_tsv("quora\_duplicate\_questions.tsv") |
|  |  |
|  | tokenizer <- text\_tokenizer(num\_words = FLAGS$vocab\_size) |
|  | fit\_text\_tokenizer(tokenizer, x = c(df$question1, df$question2)) |
|  |  |
|  | question1 <- texts\_to\_sequences(tokenizer, df$question1) |
|  | question2 <- texts\_to\_sequences(tokenizer, df$question2) |
|  |  |
|  | question1 <- pad\_sequences(question1, maxlen = FLAGS$max\_len\_padding, value = FLAGS$vocab\_size + 1) |
|  | question2 <- pad\_sequences(question2, maxlen = FLAGS$max\_len\_padding, value = FLAGS$vocab\_size + 1) |
|  |  |
|  | # keras model |
|  |  |
|  | input1 <- layer\_input(shape = c(FLAGS$max\_len\_padding)) |
|  | input2 <- layer\_input(shape = c(FLAGS$max\_len\_padding)) |
|  |  |
|  | embedding <- layer\_embedding( |
|  | input\_dim = FLAGS$vocab\_size + 2, |
|  | output\_dim = FLAGS$embedding\_size, |
|  | input\_length = FLAGS$max\_len\_padding, |
|  | embeddings\_regularizer = regularizer\_l2(l = FLAGS$regularization) |
|  | ) |
|  | seq\_emb <- layer\_lstm( |
|  | units = FLAGS$seq\_embedding\_size, |
|  | recurrent\_regularizer = regularizer\_l2(l = FLAGS$regularization) |
|  | ) |
|  |  |
|  | vector1 <- embedding(input1) %>% |
|  | seq\_emb() |
|  | vector2 <- embedding(input2) %>% |
|  | seq\_emb() |
|  |  |
|  | out <- layer\_dot(list(vector1, vector2), axes = 1) %>% |
|  | layer\_dense(1, activation = "sigmoid") |
|  |  |
|  | model <- keras\_model(list(input1, input2), out) |
|  | model %>% compile( |
|  | optimizer = "adam", |
|  | loss = "binary\_crossentropy", |
|  | metrics = list( |
|  | acc = metric\_binary\_accuracy |
|  | ) |
|  | ) |
|  |  |
|  | set.seed(1817328) |
|  | val\_sample <- sample.int(nrow(question1), size = 0.1\*nrow(question1)) |
|  |  |
|  | model %>% |
|  | fit( |
|  | list(question1[-val\_sample,], question2[-val\_sample,]), |
|  | df$is\_duplicate[-val\_sample], |
|  | batch\_size = 128, |
|  | epochs = 30, |
|  | validation\_data = list( |
|  | list(question1[val\_sample,], question2[val\_sample,]), df$is\_duplicate[val\_sample] |
|  | ), |
|  | callbacks = list( |
|  | callback\_early\_stopping(patience = 5), |
|  | callback\_reduce\_lr\_on\_plateau(patience = 3) |
|  | ) |
|  | ) |
|  |  |
|  | save\_model\_hdf5(model, "model-question-pairs.hdf5", include\_optimizer = TRUE) |
|  | save\_text\_tokenizer(tokenizer, "tokenizer-question-pairs.hdf5") |

We additionally added an early stopping callback in the training step in order to stop training if validation loss doesn’t decrease for 5 epochs in a row. This will hopefully reduce training time for bad models. We also added a learning rate reducer to reduce the learning rate by a factor of 10 when the loss doesn’t decrease for 3 epochs (this technique typically increases model accuracy).

model %>% fit(

...,

callbacks = list(

callback\_early\_stopping(patience = 5),

callback\_reduce\_lr\_on\_plateau(patience = 3)

)

)

We can now execute a tuning run to probe for the optimal combination of hyperparameters. We call the tuning\_run() function, passing a list with the possible values for each flag. The tuning\_run() function will be responsible for executing the script for all combinations of hyperparameters. We also specify the sample parameter to train the model for only a random sample from all combinations (reducing training time significantly).

library(tfruns)

runs <- tuning\_run(

"question-pairs.R",

flags = list(

vocab\_size = c(30000, 40000, 50000, 60000),

max\_len\_padding = c(15, 20, 25),

embedding\_size = c(64, 128, 256),

regularization = c(0.00001, 0.0001, 0.001),

seq\_embedding\_size = c(128, 256, 512)

),

runs\_dir = "tuning",

sample = 0.2

)

The tuning run will return a data.frame with results for all runs. We found that the best run attained 84.9% accuracy using the combination of hyperparameters shown below, so we modify our training script to use these values as the defaults:

FLAGS <- flags(

flag\_integer("vocab\_size", 50000),

flag\_integer("max\_len\_padding", 20),

flag\_integer("embedding\_size", 256),

flag\_numeric("regularization", 1e-4),

flag\_integer("seq\_embedding\_size", 512)

)

**Making predictions**

Now that we have trained and tuned our model we can start making predictions. At prediction time we will load both the text tokenizer and the model we saved to disk earlier.

library(keras)

model <- load\_model\_hdf5("model-question-pairs.hdf5", compile = FALSE)

tokenizer <- load\_text\_tokenizer("tokenizer-question-pairs")

Since we won’t continue training the model, we specified the compile = FALSE argument.

Now let`s define a function to create predictions. In this function we we preprocess the input data in the same way we preprocessed the training data:

predict\_question\_pairs <- function(model, tokenizer, q1, q2) {

q1 <- texts\_to\_sequences(tokenizer, list(q1))

q2 <- texts\_to\_sequences(tokenizer, list(q2))

q1 <- pad\_sequences(q1, 20)

q2 <- pad\_sequences(q2, 20)

as.numeric(predict(model, list(q1, q2)))

}

We can now call it with new pairs of questions, for example:

predict\_question\_pairs(

model,

tokenizer,

"What's R programming?",

"What's R in programming?"

)

[1] 0.9784008

Prediction is quite fast (~40 milliseconds).

**Deploying the model**

We deploy the model via Shiny apps. Below is the complete code for Shiny Apss:

# Load packages

library(shiny)

library(shinythemes)

library(dplyr)

library(readr)

# Load data

trend\_data <- read\_csv("data/trend\_data.csv")

trend\_description <- read\_csv("data/trend\_description.csv")

# Define UI

ui <- fluidPage(theme = shinytheme("lumen"),

titlePanel("Google Trend Index"),

sidebarLayout(

sidebarPanel(

# Select type of trend to plot

selectInput(inputId = "type", label = strong("Trend index"),

choices = unique(trend\_data$type),

selected = "Travel"),

# Select date range to be plotted

dateRangeInput("date", strong("Date range"), start = "2007-01-01", end = "2017-07-31",

min = "2007-01-01", max = "2017-07-31"),

# Select whether to overlay smooth trend line

checkboxInput(inputId = "smoother", label = strong("Overlay smooth trend line"), value = FALSE),

# Display only if the smoother is checked

conditionalPanel(condition = "input.smoother == true",

sliderInput(inputId = "f", label = "Smoother span:",

min = 0.01, max = 1, value = 0.67, step = 0.01,

animate = animationOptions(interval = 100)),

HTML("Higher values give more smoothness.")

)

),

# Output: Description, lineplot, and reference

mainPanel(

plotOutput(outputId = "lineplot", height = "300px"),

textOutput(outputId = "desc"),

tags$a(href = "https://www.google.com/finance/domestic\_trends", "Source: Google Domestic Trends", target = "\_blank")

)

)

)

# Define server function

server <- function(input, output) {

# Subset data

selected\_trends <- reactive({

req(input$date)

validate(need(!is.na(input$date[1]) & !is.na(input$date[2]), "Error: Please provide both a start and an end date."))

validate(need(input$date[1] < input$date[2], "Error: Start date should be earlier than end date."))

trend\_data %>%

filter(

type == input$type,

date > as.POSIXct(input$date[1]) & date < as.POSIXct(input$date[2]

))

})

# Create scatterplot object the plotOutput function is expecting

output$lineplot <- renderPlot({

color = "#434343"

par(mar = c(4, 4, 1, 1))

plot(x = selected\_trends()$date, y = selected\_trends()$close, type = "l",

xlab = "Date", ylab = "Trend index", col = color, fg = color, col.lab = color, col.axis = color)

# Display only if smoother is checked

if(input$smoother){

smooth\_curve <- lowess(x = as.numeric(selected\_trends()$date), y = selected\_trends()$close, f = input$f)

lines(smooth\_curve, col = "#E6553A", lwd = 3)

}

})

# Pull in description of trend

output$desc <- renderText({

trend\_text <- filter(trend\_description, type == input$type) %>% pull(text)

paste(trend\_text, "The index is set to 1.0 on January 1, 2004 and is calculated only for US search traffic.")

})

}

# Create Shiny object

shinyApp(ui = ui, server = server)