# Classifying Duplicate Questions from Quora with Keras

TENSORFLOW/KERAS NATURAL LANGUAGE PROCESSING

In this post we will use Keras to classify duplicated questions from Quora. Our implementation is inspired by the Siamese Recurrent Architecture, with modifications to the similarity measure and the embedding layers (the original paper uses pre-trained word vectors)

AUTHOR
Daniel Falbel

AFFILIATION
Curso-R

PUBLISHED
Jan. 8, 2018

CITATION
Falbel, 2018

## Introduction

In this post we will use Keras to classify duplicated questions from Quora. The dataset first appeared in the Kaggle competition 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, with modifications to the similarity measure and the embedding layers (the original paper uses pre-trained word vectors). Using this kind of architecture dates back to 2005 with Le Cun et al and is useful for verification tasks. 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.

After the competition, Quora also described their approach to this problem in this blog post.

## Dowloading data

Data can be downloaded from the Kaggle dataset webpage or from Quora's release of the dataset:

```
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
```

```
# 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
Epoch 2/10
363861/363861 [==============================] - 88s 243us/step - loss: 0.5528 - acc
```

```
Epoch 3/10
363861/363861 [==============================] - 88s 242us/step - loss: 0.5428 - acc
Epoch 4/10
363861/363861 [==============================] - 88s 242us/step - loss: 0.5353 - acc
Epoch 5/10
363861/363861 [==============================] - 88s 242us/step - loss: 0.5299 - acc
Epoch 6/10
363861/363861 [==============================] - 88s 242us/step - loss: 0.5256 - acc
Epoch 7/10
363861/363861 [==============================] - 88s 242us/step - loss: 0.5211 - acc
Epoch 8/10
363861/363861 [==============================] - 88s 242us/step - loss: 0.5173 - acc
Epoch 9/10
363861/363861 [==============================] - 88s 242us/step - loss: 0.5138 - acc
Epoch 10/10
363861/363861 [==============================] - 88s 242us/step - loss: 0.5092 - acc
```

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):

```
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     )
)
```

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 here.

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:

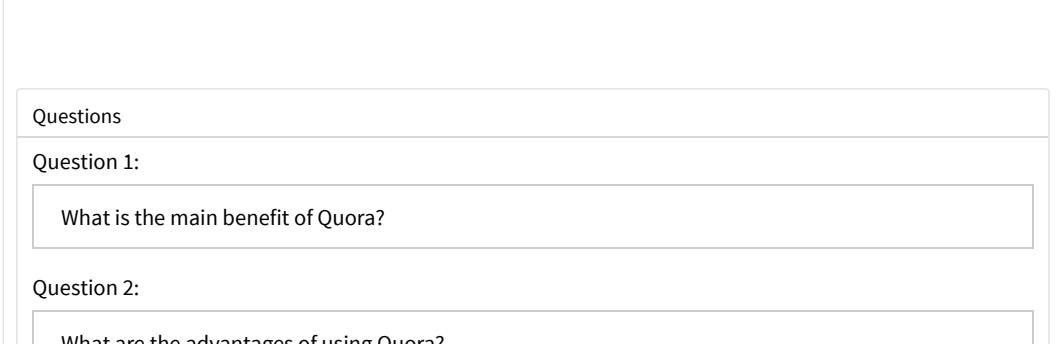
```
predict_question_pairs(
  model ,
  tokenizer,
  "What's R programming?",
  "What's R in programming?"
)
```

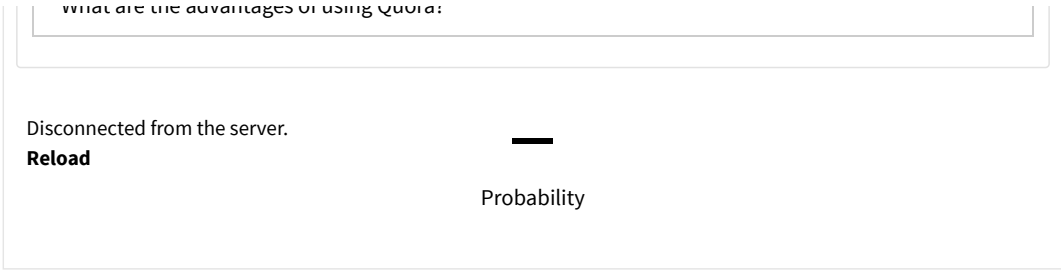
```
[1] 0.9784008
```

Prediction is quite fast (~40 milliseconds).

# Deploying the model

To demonstrate deployment of the trained model, we created a simple Shiny application, where you can paste 2 questions from Quora and find the probability of them being duplicated. Try changing the questions below or entering two entirely different questions.






The shiny application can be found at https://jjallaire.shinyapps.io/shiny-quora/ and it's source code at https://github.com/dfalbel/shiny-quora-question-pairs.

Note that when deploying a Keras model you only need to load the previously saved model file and tokenizer (no training data or model training steps are required).

# Wrapping up

- We trained a Siamese LSTM that gives us reasonable accuracy (84%). Quora's state of the art is 87%.
- We can improve our model by using pre-trained word embeddings on larger datasets. For example, try using what's described in this example. Quora uses their own complete corpus to train the word embeddings.
- After training we deployed our model as a Shiny application which given two Quora questions calculates the probability of their being duplicates.

**Citation**

For attribution, please cite this work as

```
Falbel (2018, Jan. 9). RStudio AI Blog: Classifying Duplicate Questions from Quora with Keras.
Retrieved from https://blogs.rstudio.com/tensorflow/posts/2018-01-09-keras-duplicate-questions-
quora/
```

BibTeX citation

```
@misc{falbel2018classifying,
  author = {Falbel, Daniel},
  title = {RStudio AI Blog: Classifying Duplicate Questions from Quora with Keras},
  url = {https://blogs.rstudio.com/tensorflow/posts/2018-01-09-keras-duplicate-questions-quora/},
  year = {2018}
}
```