

EE-258: NEURAL NETWORKS

Charles W. Davidson College of Engineering San Jose State University



Project #2

Tensorflow 2.0 Question & Answering systems,

Kaggle competition, 2019

Advisor: Birsen Sirkeci

By:

Shreeya Mahadevaswamy,
Mustafa Yavuz

The goal of this project is to implement the question answering system using the tensorflow 2.0 version to select the best relevant short and long answers from the model built by feeding the dataset of Wikipedia articles to the given questions as input.

DATA DESCRIPTION: The input dataset consists of the information from the Wikipedia articles for a set of questions framed to get long answers, short answers, yes/no answers or sometimes even no answer. The short answers are present in the long answer itself and are identified with special tokens. A brief statistical summarization for the dataset can be provided as follows:

- The dataset contains of about 307,372 training examples, 7,830 examples for development or validation, and a further 7,842 examples for testing. Out of the training examples provided, 152,148 of them have a long answer and 110,724 have a short answer. Short answers can be smaller sets of text such as a yes or no in the document (3,798) or of a smaller length such as of one or two sentences (106,926).
- But due to the software constraints such as poor CPU performance and the long duration taken to train the dataset, the training examples fed to the model is about 200,000. The validation set of about 50,000 has been considered as the test set to evaluate the test performance of the model as the test set provided by the Kaggle is unlabelled.
- Tokens are used to determine or differentiate between different candidate long and short answers in the document text as well as the html files. For example, taking two possible long answers for a question, we can see that the second answer is present in the first long answer. [1]

```
"long_answer_candidates": [  
  { "start_byte": 32, "end_byte": 106, "start_token": 5, "end_token": 22, "top_level": true },  
  { "start_byte": 65, "end_byte": 102, "start_token": 13, "end_token": 21, "top_level": false },]
```

The “top level” variable is a bool variable which lets the developer/user know if that particular long answer is nested or not. Therefore, the answers can be of four types:

1. Long answers
2. Short answers
3. Yes/ No answers
4. None (by default if no “yes”/”no”)

The dataset consists of the following fields:

- “document_text” which consists of the information or relevant text for the posed question
- “question_text” – text of the question to be responded
- “long_answer_candidates” - a file in the JSON format consisting of an array containing all of the possible long answers.
- “example_id” – a unique ID given for each of the samples.

The training dataset consists of the “annotations” field in addition to the above mentioned fields. This is the “label” provided to the machine to enable supervised learning. The label tells the machine to look for the corresponding long and short answers for the posed question. The answers can also be of type “yes”/ ”no” if there are no short answers.

- “annotations” - a JSON array consisting of all the right long and the short answers

- “document_url” – the web address or the URL for the full article is provided. This has been provided only for reference and cannot be used directly to verify the indices of the text.

The “annotation” data field present in the input dataset contains the label for the posed question to identify if it has a long answer or short answer or the type of answer. Each annotation defines a “long_answer” split slices, a list of short_answers, and a yes_no_answer. If the annotator points to the label of a long answer, then the long answer dictionary created categorises this long answer using an index into the list of long answer candidates, the byte offsets and the token offsets, and if the annotator signifies no long answer, then all the fields in the long answer dictionary are set to -1.

We used the kernel by Xhulu in Kaggle[2] for the baseline implementation along with the tensorflow Bert baseline [3].

BUILD DATASET: Utility functions are used to convert the input files in the json format to dataframes for both testing and training data. Only 1/S of the negative-labelled data is loaded along with all the positive labelled data where S is the sampling rate.

PRE-PROCESSING:

For pre-processing and cleaning of the given dataset, the Keras tokenizer has been used.

1. “tokenizer.fit_on_texts()”: To preprocess the data, it should be first fit into the tokenizer class of the Keras library. For this, a Keras tokenizer object is created and using that object, the “fit_on_texts()” function is called to convert the data into a list of samples or tokens.
2. “tokenizer.text_to_sequence()”: This function converts each text in the input dataset to a unique integer. This process is also known as tokenization. Tokenization is performed by splitting the data corpus into tokens of words and the function returns a list of the word tokens. For example, “I love songs because they lift my mood” becomes “5, 6,20,9,12,9,22,3” . Random dedicated integers are given for each word. The word “songs” is mapped to the integer “20”. The text of the word tokens is mapped to a stream of integers.
3. “pad_sequence(list_tokenized_train, maxlen=maxlen)”: The “pad_sequence” function takes two parameters; one is the tokenized text converted to integers and the other argument is the maximum possible length of a sentence in the data corpus. The “maxlen” parameter usually is set according to the length of the longest sentence in the dataset after removing the outliers. This function converts all the sentences in the input dataset to be of a particular uniform length by padding the sentences of smaller length than “maxlen” with zeroes and by truncating the sentences of longer length than “maxlen”.

After the pre-processing, the unique sentence tokens from the input data are stored in the tokenizer object dictionary and they can be accessed using the “tokenizer.word_index”. The words are the keys, and the corresponding integer values to which they are mapped to are the values of the dictionary.

BASELINE MODEL:

The baseline model constructed for this project is a bidirectional LSTM model with two layers which uses the fasttext word embeddings to create its word vector. Two LSTMs are trained, one on the questions and the other on the text, and their output is concatenated and fed to a fully-connected neural network. The sigmoid activation function is used to add non-linearity to the model for predicting the binary output.

Adam optimizer and the binary cross-entropy loss is used for regularization of the model.

➤ FAST-TEXT WORD EMBEDDINGS

The Fasttext dictionary consisting of word embeddings that are trained using “word2vec” is used to get the word-vectors. Fasttext pre-trained word embeddings are preferred over plain word2vec embeddings because it uses n-grams for each word, i.e. each word is seen as ‘n’ number of root words. For example, the word “single” with n=3 will have three roots “sin”, “ing” and “gle”. Therefore, it provides better precision for the results. These acquired embeddings are saved in a matrix variable “**embedding_matrix**”, whose index will be the corresponding integer of the word in the tokenizer’s **word_index** dictionary.

➤ Building Model with embedding matrix init weights

1. Built a two 2-layer bidirectional LSTM; one to read the questions, and the other one to read the text.
2. Concatenated the output of both the LSTMs and fed in 2-layer fully-connected neural networks.
3. Predict the binary output using the sigmoid activation function.

➤ Bidirectional LSTMs:

We decided upon building a bidirectional LSTM as a base model for this project as it is said to produce better results for sequence classification problems. Bidirectional LSTMs are preferred over LSTMs as it trains two LSTMs on the input data, instead of one. The first LSTM is trained on the input sequence as-is and the second one is trained on a inverted (copy) input data. This tends to provide better context to the LSTMs and effects in faster and more precise learning of the dataset. Below is a figure of a bidirectional LSTM that helped us understand its internal working better.

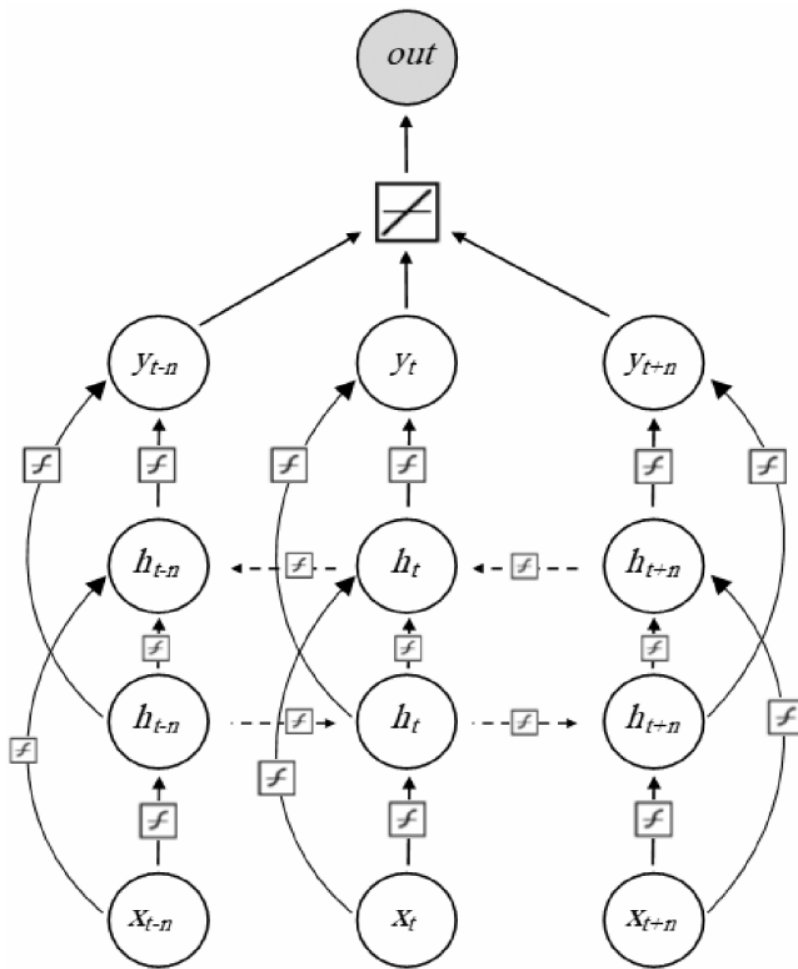


FIGURE 1: y_t : returned value of cell t from the learning process for each training sample, x_t : input data, and h_t : hidden layers. [4]

MODEL IMPROVEMENTS:

Global Max Pooling: To reduce the amount of parameters and the excessive computation in the LSTM, we used Global Max Pooling to progressively reduce the spatial size of the representation. Mathematically, the global max pooling can be defined as the ordinary max pooling layer with the size of pool equal to the size of the input.

OPTIMIZERS:

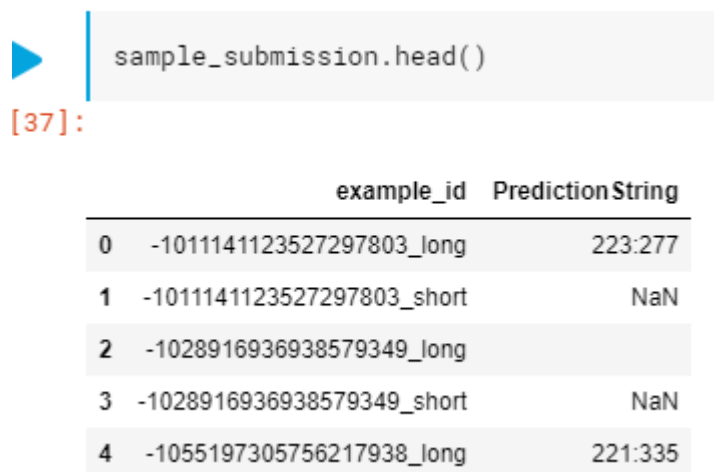
We tried a number of gradient descent optimization algorithms like Stochastic Gradient Descent, Adagrad and Adam optimizer to update weights. Adam optimizer and Adagrad gave the same performance metrics of precision and recall with the values of about 0.72 precision and 0.28 recall for the test set. But the Stochastic Gradient Descent performed way worse than the other tested optimizers as it maintains a fixed learning rate for all network weights unlike the other optimizers. It decreased the test accuracy to 0%. As a result, we stuck to using Adam optimizer in the model as it is known to produce better results fast.

BERT for long answers:

BERT (by Google) consists of pre-trained language representations that gives the comparatively better performance when applied on Natural Language Processing (NLP) tasks. BERT is known to perform better than all the other previous methods as it is a bidirectional system pre-trained on unsupervised data.

We tried to use the BERT embeddings instead of fasttext embeddings in the model to check the variance in performance metrics. As a result, we used the code from mmmarchetti [4] on Kaggle to check BERT's performance on predicting long answers and got a public score of 0.44 on Kaggle which is way higher than 0.17 that is observed in the case of using fast-text embeddings to predict long answers.

Results for the long answers only using BERT:



```
sample_submission.head()
```

[37]:

	example_id	PredictionString
0	-1011141123527297803_long	223:277
1	-1011141123527297803_short	NaN
2	-1028916936938579349_long	
3	-1028916936938579349_short	NaN
4	-1055197305756217938_long	221:335

To predict the short answers, the yes/ no answers and the blank answers, we built a new kernel using the BERT baseline model[3] and the BERT implementation code. Using the inbuilt enum in BERT to predict the type of answer and on identifying the special tokens “CLS” to identify the start and stop of statements, BERT is implemented.

BINARY CROSS-ENTROPY LOSS:

To train an NN, error measure between the outputs and the required target outputs of the training data is necessary. This neural network uses binary cross entropy error to train using the back-propagation algorithm. Binary cross entropy loss function is used in cases where the decision made is binary. Its output value which is a probability value increases as the output produced diverges from the label.

The output of both the LSTMs are then concatenated and fed into the fully connected dense layer using a sigmoid activation function.

RESULTS & CONCLUSION:

The Tensorflow 2.0 Q&A systems implemented to predict the long answers performs better using BERT than Fasttext pre-trained library. The public score on Kaggle for the Bert is about 0.44 whereas the score for Fasttext is about 0.17. Accuracy cannot be chosen as a performance metric in this case as the dataset is unbalanced. Therefore, precision and recall is considered. For the fasttext model, the precision is observed to be around 0.75 and the recall around 0.32.

As we implemented a different kernel to predict short answers, using BERT, the performance metrics is as follows:

FUTURE WORK:

- With better software conditions, i.e., with a better CPU or GPU(provided by Kaggle), more data can be fed to the model which will result in a better performance without much time consumption. We could not modify the built model much because it was very time consuming and the Kaggle kernel crashed due to extensive computations.
- Instead of using two different models to predict the short and long answers, we can combine to predict both using a single kernel.
- If more time is provided, the working and application of BERT can be learnt better and we can modify various parameters and fine tune it for a better performance.

REFERENCES:

1. <https://github.com/google-research-datasets/natural-questions/blob/master/README.md>
2. <https://www.kaggle.com/xhlulu/tf2-qa-lstm-for-long-answers-predictions>
3. <https://www.kaggle.com/prokaj/bert-joint-baseline-notebook>
4. <https://www.kaggle.com/mmmarchetti/tensorflow-2-0-bert-yes-no-answers>
5. <https://github.com/google-research/bert>
6. <https://books.google.com/books?id=k3VQDwAAQBAJ&pg=PA185&lpg=PA185&dq=Figure+4+illustrates+the+methodology+followed+to+build+the+BLSTM+network&source=bl&ots=fmMVTpPHDQ&sig=ACfU3U0Ax4x05b5O2UBQeVd6l0Z7v6uSNw&hl=en&sa=X&ved=2ahUKEwjDyrWhi7nmAhWDvp4KHSNyDV0Q6AEwAHoECACQAQ#v=onepage&q=Figure%20%20illustrates%20the%20methodology%20followed%20to%20build%20the%20BLSTM%20network&f=false>
7. <https://www.semanticscholar.org/paper/A-Unified-Tagging-Solution%3A-Bidirectional-LSTM-with-Wang-Qian/191dd7df9cb91ac22f56ed0dfa4a5651e8767a51>