Faith Chung
Net-id: faithc2
CS 410
Fall 2020

TensorFlow - NLP Tech Review

Introduction

Tokenization is a critical part of natural language processing (NLP), because it is responsible for grouping together sequences of characters from the text data into meaningful semantic units for processing. During tokenization, a given character sequence and defined document units are used to chop up the documents into smaller units, called tokens. For English, tokenization can be tricky due to (but not limited to) various uses of punctuation, capitalization, grouping of words, and variations of the same word. For example, “co-education” and “San Francisco” should count as their own tokens instead of being split into two tokens each. NLP tokenization methods that mitigate these mistakes include dropping stop words, which are extremely common words that offer little value in the vocabulary/text corpus, normalizing tokens to count equivalent tokens to the same term (such as ran, run, running), normalizing tokens to remove diacritics (which are accents that are placed on words), and reducing all letters to lowercase. This tech review covers all the tokenization methods offered by the TensorFlow Keras Text API and compares how the different tokenization offerings impact the accuracy of text classification, using a neural network. Keras was chosen because it is one of the most popular high level APIs used to interface with Tensorflow, allowing the user to utilize Tensorflow functionality with minimal user actions. The test data includes 200 quotes, and the model is used to classify whether or not each quote is motivational or demotivational. The data can be viewed in the motivational.txt and demotivation.txt files in the same directory as this tech review.

Text_to_word_sequence Review:

The Keras Text API provides 3 ways to preprocess text data. First, the “text_to_word_sequence” function simply takes 1 string of text input and 1 optional string of unwanted characters to filter and returns the text corpus for that inputted string of text. This function does not drop stop words or provide any normalization, but it automatically converts all text to lowercase and filters out punctuation, tabs, and new lines, and splits based on whitespaces. This is an extremely efficient method of retrieving the text corpus and getting the size of the vocabulary. However, this only works for a single string, so preprocessing must be done to concatenate all text data into one string. In “tokensTest.py” in the same directory as this tech review, lines 50 and 51 demonstrates the usage of this function, which revealed that the 200 quotes contained 899 unique words.

Code example:

Code	

```
motiv = self.flatten(self.motiv_quotes)
demotiv = self.flatten(self.demotiv_quotes)
```

	`self.vocab = set(text_to_word_sequence(motiv + " " + demotiv))` `self.vocab_size = len(self.vocab)`
Output	vocab size: 899

Hashing Methods for Text-Preprocessing:

In order to actually use the text data to train a model, the remaining 2 methods tokenizes the text, where one uses a hash function and the other assigns distinct values to each word. The hash function approach is achieved by using the "hashing_trick" or "one_hot functions", where the one_hot function is just a wrapper for hashing_trick. The arguments for hashing_trick are a single text string, vocabulary size, and a hash function. For this review, the 899 vocabulary count was multiplied by 1.5 to reduce the number of hashing collisions. Even with this larger count and using the stable 'md5' as the hash function, there was a total of 648 hash ids for the 899 words, and 197 of those hash ids had a hash collision with an average of 2-3 words each. My implementation can be found in lines 56-57 in "tokensTest.py", where hashing_trick was called to encode each of the 200 quotes, and a nested list containing all 200 encodings were stored in the self.encoded_quotes variable.

Code	`self.quotes = self.motiv_quotes + self.demotiv_quotes`
	`for quote in self.quotes:`
	`  self.encoded_quotes.append(hashing_trick(quote,`
	`  round(self.vocab_size*1.5), hash_function='md5'))`

Output	number of Hash Ids: 648 number of hash ids with collisions: 197 Snippet of the hash id and word mappings: 498: ['too', 'emotions'] 928: ['many'] 168: ['fears', 'fight'] 1201: ['interest', 'gives', 'solution'] 398: ['look', 'listen', '"furthermore'] 263: ['matters'] 1345: ['road'] 75: ['almost'] 330: ['exactly'] … Example of encoded quote: "Take up one idea make that one idea your life" [977 1251 528 747 675 1128 528 747 523 1141]

Tokenizer Class for Text-Preprocessing:

For the other tokenization method, Keras offers a Tokenizer class. The other text preprocessing methods were one-off convenience methods, but this Tokenizer class is much more sophisticated, taking list of strings as input, allowing the user to state the size of

vocabulary, and assign a default “out of value” token for any other tokens in a document that is not in the vocabulary.

Code	`tokenizer = Tokenizer(num_words=self.vocab_size, oov_token= '<OOV>')`
	`data = self.motiv_quotes + self.demotiv_quotes`
	`tokenizer.fit_on_texts(data)`
	`encoded_data = tokenizer.texts_to_sequences(data)`
	`print("word index: " + str(tokenizer.word_index))`

Output	word index snippet: {'<OOV>': 1, 'the': 2, 'you': 3, 'to': 4, 'is': 5, 'a': 6, 'and': 7, 'of': 8,...} Example of encoded quote: `"If you want to achieve greatness stop asking for permission"` [15 3 57 4 204 329 116 330 36 331]

Comparing the hashing and tokenizer methods, hashing_trick took 7.358074188232422 ms, while using the Tokenizer class took 7.977008819580078 ms. I can imagine as the test data increases, it will be more beneficial to use the Tokenizer class due to less time spent on concatenating all of the text into one string, but it will increase the total run time to process all the tokens instead of using the hash function. Using the “pad_sequences” function from `tensorflow.keras.preprocessing.sequence`, each of the encoded quotes for both the tokenized and hashed quotes were padded with 0’s to have uniform length to be used as training data for the neural network. The neural network has an embedded layer for the text with an embedded dimensionality of 16 for each word and where the input size is the length of each encoded quote, a hidden ReLu layer with 5 nodes, and a final node with a sigmoid activation function, where 0 means demotivational and 1 means motivational. For both the hashed and tokenized data, I saved 140 out of the 200 quotes (where 70 is motivational and 70 is demotivational) for training, and 60 of the quotes for testing (where 30 is motivational and 30 is demotivational). The code for building the model and training it is below:

```
# build model
self.embedding_dim = 16
model = tf.keras.Sequential([
tf.keras.layers.Embedding(self.vocab_size, self.embedding_dim,
                          input_length=self.max_len),
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(5, activation='relu'),
tf.keras.layers.Dense(1, activation='sigmoid')
]
)

# get Tokenized train and test data
```

```
self.token_train = (self.padded_token_quotes[0:70, 0:280]).tolist() +
                   (self.padded_token_quotes[130:200, 0:280]).tolist()
self.train_label = [1]*70 + [0]*70 # 1 = motivational, 0 = demotivational
self.token_test  = (self.padded_token_quotes[70:130, 0:280]).tolist()
self.test_label  = [1]*30 + [0]*30

       # train model with token data
       num_epochs = 20
       model.fit(
       self.token_train,
       self.train_label,
       epochs=num_epochs,
       validation_data = (self.token_test, self.test_label)
       )
```

Conclusion:

After fitting the models, the results showed that the tokenized data performed better for training the model than the hashed data. This was expected because of the hashing collisions, but it appears that both options are valid for NLP. The use cases for the “one_hot” and “hashing_trick” are for quick preprocessing for few documents, while the Tokenizer class is more ideal for large datasets. To view the complete log with test results, please run **python tokensTest.py**, which is in the same directory as this tech review.

Machine Learning Results

```
TRAINING WITH TOKENIZER DATA
Epoch 1/20
5/5 [==============================] - 0s 36ms/step - loss: 0.6948 - accuracy: 0.4714 - val_loss: 0.6939 - val_accuracy:
0.5000
...
Epoch 20/20
5/5 [==============================] - 0s 6ms/step - loss: 0.2940 - accuracy: 0.9786 - val_loss: 0.6274 - val_accuracy:
0.6667
TRAINING WITH HASHED DATA
Epoch 1/20
5/5 [==============================] - 0s 30ms/step - loss: 0.6982 - accuracy: 0.5071 - val_loss: 0.6943 - val_accuracy:
0.5000
...
Epoch 20/20
5/5 [==============================] - 0s 6ms/step - loss: 0.5742 - accuracy: 0.8143 - val_loss: 0.6782 - val_accuracy:
0.5500
```

References:

https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html

https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer

https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding