Neural Machine Translation Using Byte Pair Encoding

AAKASH GUNDA (N14904310)
SAI VIKAS MANDADAPU (N12217800)

INDEX

1. INTRODUCTION:

Machine translation is the technology used to translate between human languages. We call the language input to the machine translation system the source language and call the output language the target language. Thus, machine translation can be described as the task of converting a sequence of words in the source and converting it into a sequence of words in the target. The goal of the machine translation expert is to come up with an effective model that allows us to perform this conversion accurately over a broad variety of languages and content. The aim of this project is to improve upon the current machine translation algorithms by introducing subword tokenization.

2. TECHNOLOGIES USED:

a. **TensorFlow:** TensorFlow is an end-to-end open-source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.
b. **Collaboratory:** Free Jupyter notebook environment that requires no setup and runs entirely in the cloud. It Provides GPU which is required for this project.
c. **Keras:** Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation.

3. NEURAL MACHINE TRANSLATION:

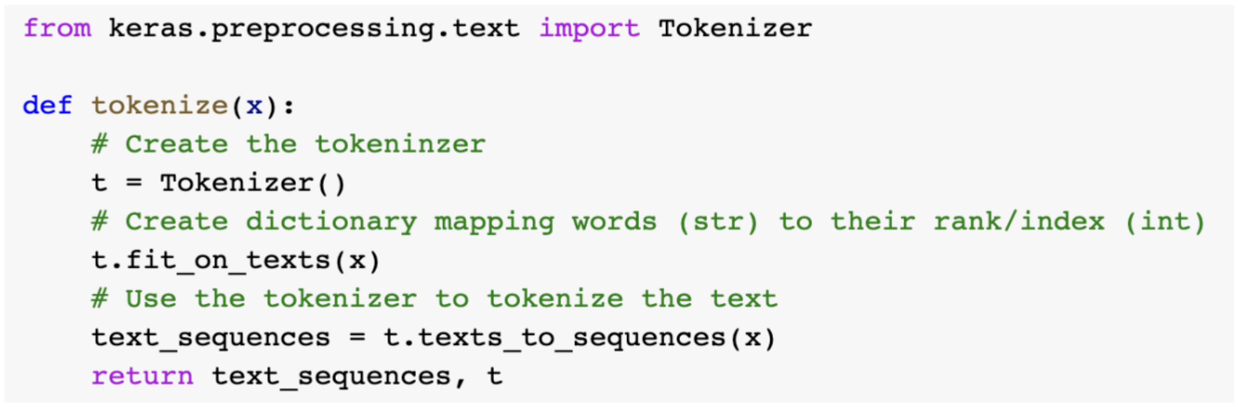
Neural Machine translation is the use of neural network models to learn a statistical model for machine translation. The key benefit to the approach is that a single system can be trained directly on source and target text. Multilayer Perceptron neural network models can be used for machine translation, although the models are limited by a fixed-length input sequence where the output must be the same length. These early models have been greatly improved upon recently using recurrent neural networks organized into an encoder-decoder architecture that allow for variable length input and output sequences. Key to the encoder-decoder architecture is the ability of the model to encode the source text into an internal fixed-length representation called the context vector. Interestingly, once encoded, different decoding systems could be used, in principle, to translate the context into different languages. The encoder-decoder recurrent neural network architecture with attention is currently the state-of-the-art on some benchmark problems for machine translation.

For a neural network to predict text data, it must first be converted into data that it can understand. The input data for a neural network must be numbers since it is a sequence of multiplication and addition operations. Hence, we won't use text data as input, instead will convert the text into sequences of integers using tokenization. Now after converting text data into a sequence of integers, we need to pad the sequence of integers with zeros to make the length of each sentence equal which is known as padding. After padding, we use Recurrent Neural Network models to convent English data to French data. The output of the RNN model is again ids which we will convert back to text using the tokenizer function. In this project we are using word tokenization from the Keras library and a subword tokenization method called byte pair encoding and comparing the accuracies of both tokenization methods by implementing them using an RNN model.

4. TOKENIZATION:

Tokenization is the process of breaking down a phrase, sentence, paragraph, or entire test document into smaller units, such as words or terms called tokens. Here, tokens can be either words, characters, or subwords. Hence, tokenization can be broadly classified into 3 types – word, character, and subword (n-gram characters) tokenization. We can turn each character or words into a sequence of numbers or each word into a number. These are called character and word ids, respectively. Character ids are used for character-level models that generate text predictions for each character. A word-level model uses word ids that generate text predictions based on the sequence of characters. Word-level models are easy to understand, since they are lower in complexity, so they are widely used.

Word Tokenization splits a piece of text into individual words based on a certain delimiter. Depending upon delimiters, different word-level tokens are formed. One of the major issues with word tokens is dealing with Out of Vocabulary (OOV) words. OOV words refer to the new words which are not encountered at training. So, these new words do not exist in the vocabulary. Hence, these methods fail in handling OOV words.



```
from keras.preprocessing.text import Tokenizer

def tokenize(x):
    # Create the tokeninzer
    t = Tokenizer()
    # Create dictionary mapping words (str) to their rank/index (int)
    t.fit_on_texts(x)
    # Use the tokenizer to tokenize the text
    text_sequences = t.texts_to_sequences(x)
    return text_sequences, t
```

Character Tokenization splits a piece of text into a set of characters. It overcomes the drawbacks we saw above about Word Tokenization. Character Tokenizers handles OOV words coherently by preserving the information of the word. It breaks down the OOV word into characters and represents the word in terms of these characters. Character tokens solve the OOV problem, but the length of the input and output sentences increases rapidly as we are representing a sentence as a sequence of characters. As a result, it becomes challenging to learn the relationship between the characters to form meaningful words.

Another tokenization method is Subword Tokenization which is in between Word and Character tokenization. Subword Tokenization splits the piece of text into subwords (or n-gram characters). Subword tokenization allows the model to have a reasonable vocabulary size while being able to learn meaningful context-independent representations. In addition, subword tokenization enables the model to process words it has never seen before, by decomposing them into known subwords. Most popular Subword Tokenization algorithm is known as Byte Pair Encoding (BPE).

5. BYTE PAIR ENCODING:

5.1 Introduction:

The inputs to natural language processing models are sequences of sentences, such as “I went to New York last week.”. The sequence consists of tokens. In old language models, tokens are usually white-space separated words and punctuations, such as [“i”, “went”, “to”, “new”, “york”, “last”, “week”, “.”]. However, this has some drawbacks. For example, if the model learned the relationship between “old”, “older”, and “oldest”, it does not mean that it understands the relationship between “smart”, “smarter”, and “smartest”. However, if we use some subtokens such as “er” and “est”, and the model learned the relationship between “old”, “older”, and “oldest”, it will have some information to understand the relationship between “smart”, “smarter”, and “smartest” as they have common prefix. In information theory, byte pair encoding (BPE) or diagram coding is a simple form of data compression in which the most common pair of consecutive bytes of data is replaced with a byte that does not occur within that data.

5.2 Byte Pair Encoding Algorithm:

Source code for performing byte pair encoding was first published in the paper “Neural Machine Translation of Rare Words with Subword Units” in 2015 for a list of words. We count the frequency of each word shown in the list. For each word, we append a special stop token “</w>” at the end of the word. Then split the word into characters. Initially, the tokens of the word are all its characters plus the additional “</w>” token. For example, the tokens for the word “low” are [“l”, “o”, “w”, “</w>”] in order. So, from words in the input dataset we get a vocabulary as shown below,

{'l o w </w> l o w e r </w> n e w e s t </w> w i d e s t </w>'}

In every iteration we count the frequency of each consecutive byte pairs, take the most frequent one and merge them into a single token.

In the first iteration of the merge, because byte pairs "e" and "s" occurred 9 times which is the most frequent, so we merge these into a new token "es".

{'l o w </w> l o w e r </w> n e w es t </w> w i d es t </w>'}

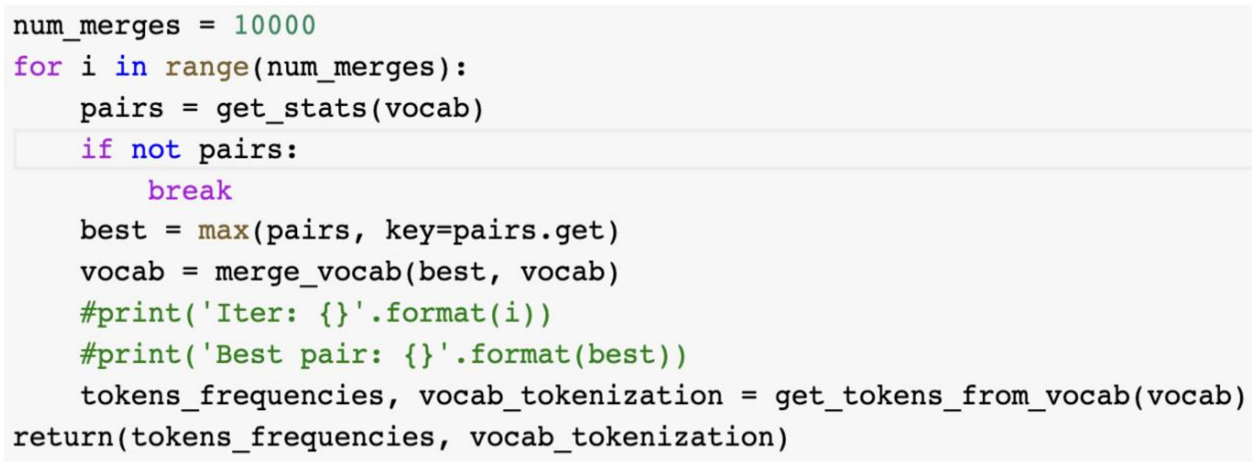
In the second iteration of merge, token "es" and "t" occurred 9 times, which is the most frequent. We merge these too into a new token "est". Note that token "es" is gone.

{'l o w </w> l o w e r </w> n e w est </w> w i d est </w>'}

In the third iteration of the merge, token "est" and "</w>" pair is the most frequent pair. Based on the dataset we alter the number of iterations required for our dataset. For our input dataset. Number of iterations is 10000.

Stop token "</w>" is also important. Without "</w>", say if there is a token "st", this token could be in the word "st ar", or the word "wide st", however, the meaning of them is quite different. With "</w>", if there is a token "st</w>", the model immediately know that it is the token for the wold "wide st</w>" but not "st ar</w>".

After each merge, there could be three scenarios, the number of tokens decreases by one, remains the same, or increases by one. But in practice, as the number of merges increases, usually the number of tokens first increases then decreases. After merging all the frequently occurring byte pairs, we will assign ids to each token available in the final dictionary.

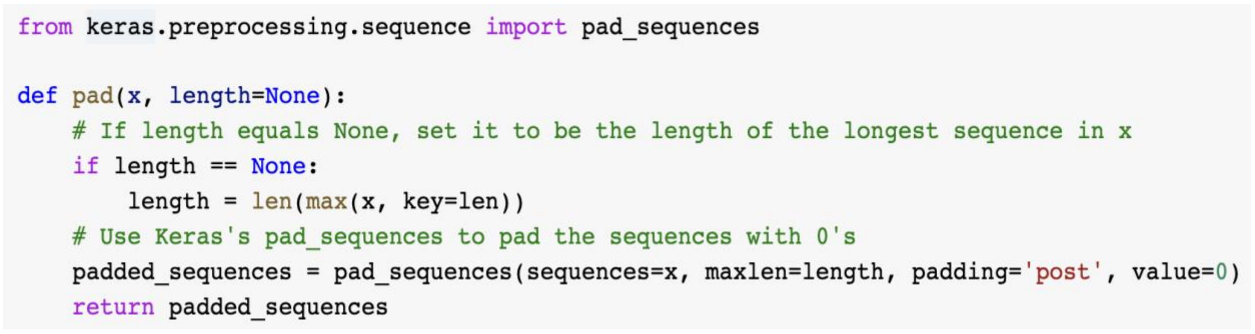


```
num_merges = 10000
for i in range(num_merges):
    pairs = get_stats(vocab)
    if not pairs:
        break
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
    #print('Iter: {}'.format(i))
    #print('Best pair: {}'.format(best))
    tokens_frequencies, vocab_tokenization = get_tokens_from_vocab(vocab)
return(tokens_frequencies, vocab_tokenization)
```

BPE brings the perfect balance between character and word-level hybrid representations which makes it capable of managing large lists of words. This behavior also enables the encoding of any rare words in the vocabulary with appropriate subword tokens without introducing any "unknown" tokens. This especially applies to foreign languages like German where the presence of many compound words can make it hard to learn a rich vocabulary otherwise.

6. PADDING:

After getting the word ids we need to batch them together. When batching the sequence of word ids, each sequence needs to be the same length. Since sentences are dynamic in length, we can pad zeros to the end of the sequences to make them the same length. We will make sure that all the English sequences have the same length, and all the French sequences have the same length by adding zeros to the end of each sequence using Keras's pad_sequences function.

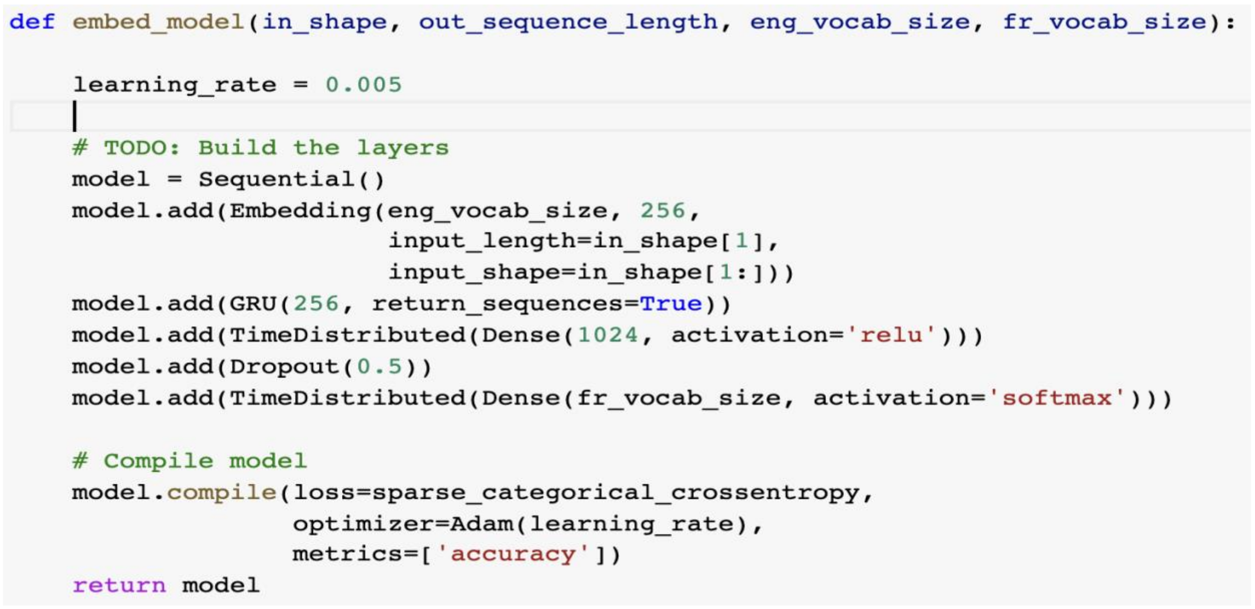


```
from keras.preprocessing.sequence import pad_sequences

def pad(x, length=None):
    # If length equals None, set it to be the length of the longest sequence in x
    if length == None:
        length = len(max(x, key=len))
    # Use Keras's pad_sequences to pad the sequences with 0's
    padded_sequences = pad_sequences(sequences=x, maxlen=length, padding='post', value=0)
    return padded_sequences
```

7. RECURRENT NEURAL NETWORKS:

Recurrent neural networks (RNNs) are a type of neural network that excels at modeling sequence data including time series and natural language. An RNN layer iterates over the timesteps of a sequence using a for loop, while keeping an internal state that encodes information about the timesteps it has seen so far. The output of the RNN model will be the ids which we have to convert back into texts for which we again use the tokenizer function available in Keras and wrote a function logit_to_text to convert the ids back into words.



```
def embed_model(in_shape, out_sequence_length, eng_vocab_size, fr_vocab_size):

    learning_rate = 0.005

    # TODO: Build the layers
    model = Sequential()
    model.add(Embedding(eng_vocab_size, 256,
                        input_length=in_shape[1],
                        input_shape=in_shape[1:]))
    model.add(GRU(256, return_sequences=True))
    model.add(TimeDistributed(Dense(1024, activation='relu')))
    model.add(Dropout(0.5))
    model.add(TimeDistributed(Dense(fr_vocab_size, activation='softmax')))

    # Compile model
    model.compile(loss=sparse_categorical_crossentropy,
                  optimizer=Adam(learning_rate),
                  metrics=['accuracy'])
    return model
```

8. RESULTS & CONCLUSIONS:

The accuracy of 92.27 can be obtained from the initial 87.59 percent just by changing the tokenization method to byte pair encoding from word tokenization. This accuracy is subjected to change a bit based on the train test split's randomness. This accuracy can be further increased to 97.12 percent just by changing the layers in RNN model as demonstrated in the python file. Byte pair encoding works even better when the size of data set increases.

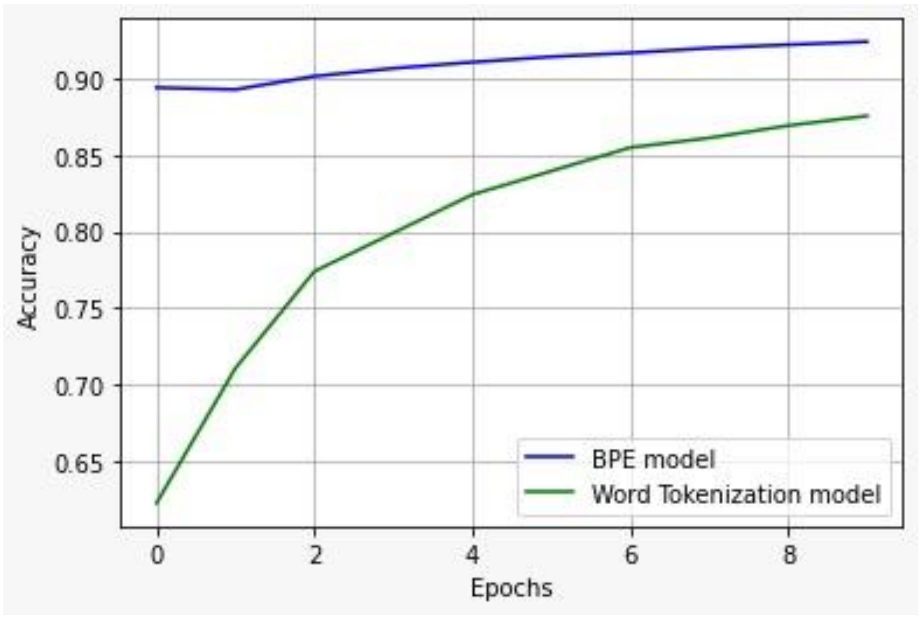


Fig.1 Accuracy comparison

REFERENCES

[1] **https://arxiv.org/pdf/1703.01619.pdf** (Graham Neubig. Neural machine translation and sequence-to-sequence models: A tutorial. arXiv preprint arXiv:1703.01619 (2017))

[2] **http://ethen8181.github.io/machine-learning/deep_learning/subword/bpe.html**

[3] **https://leimao.github.io/blog/Byte-Pair-Encoding/**

[4] **https://towardsdatascience.com/byte-pair-encoding-the-dark-horse-of-modern-nlp-eb36c7df4f10**

[5] **https://towardsdatascience.com/neural-machine-translation-with-python-c2f0a34f7dd**

[6] **https://stackabuse.com/python-for-nlp-neural-machine-translation-with-seq2seq-in-keras/**

[7] **https://www.analyticsvidhya.com/blog/2020/05/what-is-tokenization-nlp/#:~:text=A%20Quick%20Rundown%20of%20Tokenization,-Tokenization%20is%20a&text=Tokenization%20is%20a%20way%20of,n%2Dgram%20characters)%20tokenization.**

[8] **https://machinelearningmastery.com/introduction-neural-machine-translation/#:~:text=Machine%20translation%20is%20the%20task,of%20symbols%20in%20another%20language.**

[9] **https://machinelearningmastery.com/develop-neural-machine-translation-system-keras/**