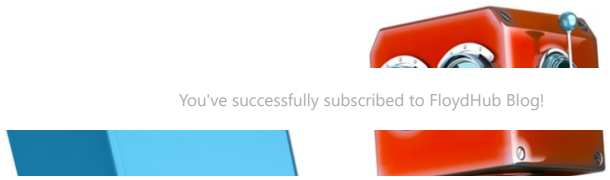
NLP

Tokenizers: How machines read

We will cover often-overlooked concepts vital to NLP, such as Byte Pair Encoding, and discuss how understanding them leads to better models.

CATHAL HORAN
28 JAN 2020 • 34 MIN READ

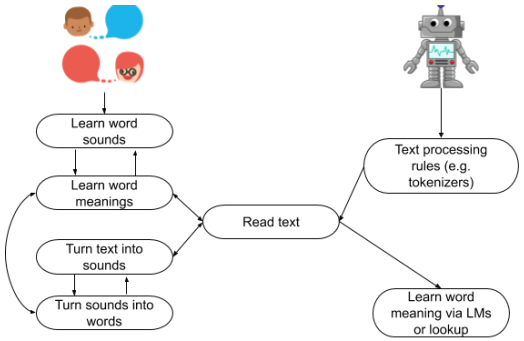


The world of Deep Learning (DL) Natural Language Processing (NLP) is evolving at a rapid pace. We tried to capture some of these trends in an earlier post which you can check out if you want more background into these developments. Two of the most important trends are the Transformer (2017) architecture and the BERT (2018) language model, which is the most famous model to take advantage of the former architecture. These two developments, in particular,

they need to perform well in complex linguistic tasks. We often skip this part of the process in order to get to the "meatier" core of the cool new model. It turns out that these input steps are a whole research field of their own, with an abundance of complex algorithms all working simply to enable the larger language models to learn higher level linguistic tasks. Think of it like the sorting algorithms needed to order vast arrays of numbers. While you only use the sort function in your Python script there is a whole industry whose sole goal is to optimize for better and better sorting algorithms. But before we dive into the specifics of these algorithms, we first need to understand why reading text is a difficult task for a machine.

Why is reading difficult for machines?

You could understand language before you learned to read. When you started school you could already talk to your classmates even though you didn't know the difference between a noun and a verb. After that, you learned to turn your phonetic language into a written language so that you could read and write. Once you had learned to turn text into sounds, you were able to access your previously learned bank of word meanings.



Computers (i.e. Language Models (LMs) or lookup programs (WordNet)) do not learn to speak before they learn to read so they cannot lean on a previous memory bank of learned word meanings. They need to find another way of discovering word meaning.

anything about language we need to develop systems that enable them to process text without the ability, like humans, of already being able to associate sounds with the meanings of words. It's the classic "chicken and egg" problem: how can machines start processing text if they know nothing about grammar, sounds, words or sentences? You can create rules that tell a machine to process text to enable it to perform a dictionary-type lookup. However, in this scenario the machine isn't learning anything, and you would need to have a static dataset of every possible combination of words and all their grammatical variants.

Instead of training a machine to lookup fixed dictionaries, we want to teach machines to recognize and "read" text in such a way that it can learn from this action itself. In other words, the more it reads the more it learns. Humans do this by leveraging how they previously learned phonetic sounds. Machines don't have this knowledge to

sequences of text are broken into smaller parts, or "tokens", and then

to use tokenization at all.

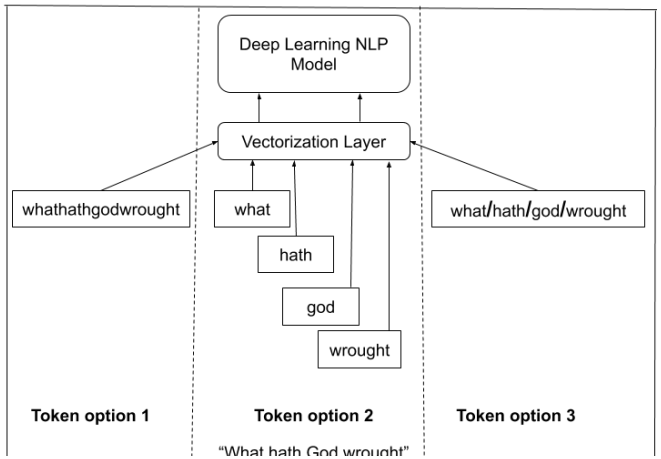
Do We Need Tokenizers?

To teach a DL model like BERT or GPT-2 to perform well at NLP tasks we need to feed it lots and lots of text. Hopefully, through the specific design of the architecture, the model will learn some level of syntactic or semantic understanding. It is still an area of active research as to what level of semantic understanding these models learn. It's thought that they learn syntactic knowledge at the lower levels of the neural network and then semantic knowledge at the higher levels as they begin to hone in on more specific language domain signals, e.g. medical vs. technical training texts.

The specific type of architecture used will have a significant impact on what tasks the model can deal with, how quickly it can learn and how well it performs. GPT2 uses a decoder architecture, for example,

BERT uses an encoder type architecture since it is trained for a larger

designed, they all need to be fed text via their input layers to perform any type of learning.



The above diagram shows that we can tokenize input text in different ways. Option 1 is not ideal since all the words are simply bunched together into one token. Option 2 breaks the input

One simple way to do this would be to simply feed the text as it appears in your training dataset. This sounds easy but there is a problem. We need to find a way to represent the words mathematically for the neural network to process it.

Remember, ***these models have no knowledge of language.*** So

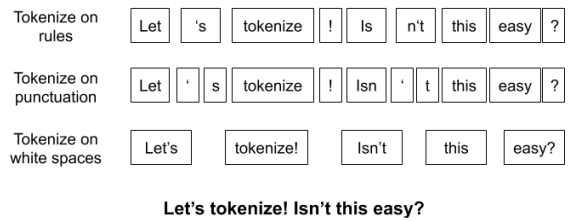
the structure of language. It will just appear like gibberish to the model and it won't learn anything. It won't understand where one word starts and another ends. It won't even know what constitutes a word. We get around this by first learning to understand spoken language and then learning to relate speech to written text. So we need to find a way to do **two** things to be able to feed our training data of text into our DL model:

1. **Split the input into smaller chunks**: The model doesn't know anything about the structure of language so we need to break it into chunks, or tokens, before feeding it into the model.
2. **Represent the input as a vector**: We want the model to learn the relationship between the words in a sentence or sequence of text. We do not want to encode grammatical rules into the model, as they would be restrictive and require specialist linguistic knowledge. Instead, we want the model to learn the relationship itself while training

as vectors where the model can encode meaning in any of the

they represent the contextual reference for the words. Alternatively, they can be fed into other layers as inputs to higher level NLP tasks such as text classification or used for transfer learning.

Before we can start training our model to produce better vectors, we first need to figure out which tokenization policy we need to implement in order to break our text into small chunks.



that breaking sentences into word-level chunks of tokens seems like the best approach. And they would be right to some extent. In the above diagram you can see that even for word tokenization there are some difficulties in creating tokens: do we ignore punctuation or include them, or do we write specific rules to end up with more coherent and useful word tokens?

So even if you work out a standard approach or ruleset that enables you to encode text into word tokens, you will still run into a few core problems:

1. **You need a big vocabulary**: When you are dealing with word tokens you can only learn those which are in your training vocab. Any words not in the training set will be treated as unknown words when using the model and identified by the dreaded "<UNK>" token. So even if you learned the word "cat" in your training set, the final model would not recognize the plural "cats". It does not break words

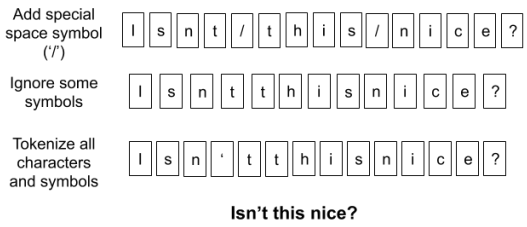
vs. "talked" and "talking". All of these would need individual

inefficient approach. Although you could address this by applying stemming or lemmatization to your input text to reduce the size of your vocabulary, you still end up with an extra step in your NLP pipeline and may be limited to certain languages.

2. **We combine words:** The other problem is that there may be some confusion about what exactly constitutes a word. We have compounded words such as “sun” and “flower” to make sunflower and hyphenated words such as “check-in” or “runner-up”. Are these one word or multiple? And we use text sequences such as “New York” or “bachelor of science” as one unit. They may not make sense as isolated words.
3. **Abbreviated words**: With the rise of social media we have shorthand for words such as “LOL” or “IMO”. Are these collections of words or new words?
4. **Some languages don’t segment by spaces**: It’s easy to

is not a trivial task.

OK, Let’s Tokenize Characters Instead of Words?



Character encoding rather than word encoding?

An alternative approach to using words is to simply tokenize the input text character by character. This way we avoid a number of pitfalls of word tokenization. For example, we can now avoid things

In the above diagram we can see that there are a number of different ways we can perform this character encoding. We can, for example, ignore spaces and simply treat every character and symbol as a single token and have a separate vector for each token. Alternatively, we may want to limit our character vocabulary to certain symbols and thus remove symbols like apostrophes. All of these ignore the spacing of words, so we may want to assign a symbol to a space and use this when creating the embedding vectors for each token.

Hopefully, using any of these approaches misspelled words or the unusual spelling of words (e.g. cooooooool) should appear as similar to each other as their learned embeddings will be close together. The same applies to different versions of the same verb, e.g. “walk”, “walk-ing”, “walk-ed”, and so on.

However, there are some drawbacks with this approach also:

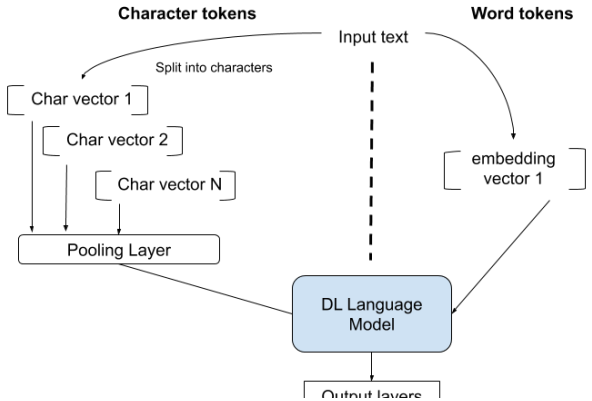
inherent meaning, so there is no guarantee that the resultant

combined into slightly unusual combinations which are not correct words. More training data should help alleviate this tendency, but we are still left with a situation where the model is losing the semantic-specific feature of words.

2. **Increased input computation**: If you use word level tokens then you will spike a 7-word sentence into 7 input tokens. However, assuming an average of 5 letters per word (in the English language) you now have 35 inputs to process. This increases the complexity of the scale of the inputs you need to process
3. **Limits network choices**: Increasing the size of your input sequences at the character level also limits the type of neural networks you can use. It is difficult to use architectures which process input sequentially since your input sequences will be much longer. However, new models such as BERT are based on the Transformer architecture, which processes inputs in

difficult to perform some type of NLP tasks on characters. If

need to do more work than if you trained it at a word level to optimize for your task.



FLOYDHUB Tokenizers: How machines read Subscribe

If word-level tokens are not ideal, and character-level tokens seem to have their own issues, then what are the alternatives? One alternative that has proved popular is a balance between the character level and the word level known as the subword-level tokenization.

You've successfully subscribed to FloydHub Blog!

Subword Tokenization

FLOYDHUB Tokenizers: How machines read Subscribe

tokenization scheme that deals with an infinite potential vocabulary via a finite list of known words. Also, we don't want the extra complexity of breaking everything into single characters since character-level tokenization can lose some of the meaning and semantic niceties of the word level.

One way we can solve this problem is by thinking of how we can re-use words and create larger words from smaller ones. Think of words like "***any***" and "***place***" which make "***anyplace***" or compound words like "***anyhow***" or "***anybody***". You don't need an entry for each word in your vocabulary list. Instead you just need to remember a few words and put them together to create the other words. That requires much less memory and effort. This is the basic idea behind subword tokenization. Try and build up the smallest collection of subword chunks which would allow you to cover all the words in your

You've successfully subscribed to FloydHub Blog!

To make a more efficient system, the subword chunks do not even

FLOYDHUB Tokenizers: How machines read Subscribe

tokenisation will break the text into chunks based on the word frequency. In practice what happens is that common words will be tokenized generally as whole words, e.g. "***the***", "***at***", "***and***", etc., while rarer words will be broken into smaller chunks and can be used to create the rest of the words in the relevant dataset.

The other factor which applies here is the size of the vocabulary allowed. This is chosen by the person running the subword algorithm. The larger the vocabulary size the more common words you can tokenize. The smaller the vocabulary size the more subword tokens you need to avoid having to use the ***<UNK>*** token. It is this delicate balance that you can tinker with to try and find an optimum solution for your particular task.

Byte Pair Encoding (BPE)

You've successfully subscribed to FloydHub Blog!

data by finding common byte pair combinations. It can also be

can look at an example to see how BPE works in practice (I used code from Lei Mao's blog for the following example. You should check out the blog if you are interested in an even deeper dive into the world of BPE):

Imagine you have a text sample that contains the following words:

"***FloydHub is the fastest way to build, train and deploy deep learning models. Build deep learning models in the cloud. Train deep learning models.***"

First let's look at how often the individual words appear. The words in this text occur in the following frequency:

learning</w>	3	train</w>	1
the</w>	2	and</w>	1
models</w>	2	deploy</w>	1
Floydhub</w>	1	Build</w>	1
is</w>	1	models</w>	1
fastest</w>	1	in</w>	1
way</w>	1	cloud</w>	1
to</w>	1	Train</w>	1

The word count for our test paragraph

The first thing you can see here is that there is a "***</w>***" token at the end of each word. This is to identify a word boundary so that the algorithm knows where each word ends. This is important as the subword algorithm looks through each character in the text and tries to find the highest frequency character pairing. Next, let's look at the frequency of character level tokens:

2	e	16	**16**	m	3
3	d	12	**17**	.	3
4	l	11	**18**	b	2
5	n	10	**19**	h	2
6	i	9	**20**	F	1
7	a	8	**21**	H	1
8	o	7	**22**	f	1
9	s	6	**23**	w	1
10	t	6	**24**	,	1
11	r	5	**25**	B	1
12	u	4	**26**	c	1
13	p	4	**27**	T	1
14	y	3			

The character frequency of our test paragraph

A few points to note here. Look through the table and familiarize

the "***</w>***" character occurs 24 times, which makes sense as there are 24 words. The next most frequent token is the "***e***" character, which occurs 16 times. In total there are 27 tokens. Now the BPE algorithm looks at the most frequent pairing, merges them and does another iteration.

What is merging?

The main goal of the BPE subword algorithm is to find a way to represent your entire text dataset with the least amount of tokens. Similar to a compression algorithm, you want to find the best way to represent your image, text or whatever you are encoding, which uses the least amount of data, or in our case tokens. In the BPE algorithm merging is the way we try and "compress" the text into subword units.

Merging works by identifying the **most frequently represented byte pairs**. In our example here a character is the same as a byte,

keep it simple, a byte pair and a character pair are the same. There

already performed in our simple example).

1. Get the word **count** frequency
2. Get the **initial token count** and frequency (i.e. how many times each character occurs)
3. Merge the **most common byte pairing**
4. Add this to the list of tokens and **recalculate the frequency count** for each token; this will change with each merging step
5. **Rinse and repeat** until you have reached your defined token limit or a set number of iterations (as in our example)

We already have our word and token values in the above tables. Next we need to find the most common byte pair and merge both into one new token. After one iteration our output looks like this:

2	e	16 - 7 = 9	17	m	3
3	d	12 - 7 = 5	18	.	3
4	l	11	19	b	2
5	n	10	20	h	2
6	i	9	21	F	1
7	a	8	22	H	1
8	o	7	23	f	1
9	de	7	24	w	1
10	s	6	25	,	1
11	t	6	26	B	1
12	r	5	27	c	1
13	u	4	28	T	1
14	p	4			
15	y	3			

Iteration 1: Best pair: ('d', 'e')

After one iteration our most frequent pairing is "***d***" and "***e***". As a result we combined these to create our first subword token (which is not a single character) "***de***". How did we calculate this? If you remember the word frequencies we calculated earlier you can see how "***de***" is the most frequent pairing.

WORD	FREQUENCY	WORD	FREQUENCY
deep </w>	3	build </w>	1
learning </w>	3	train </w>	1
the </w>	2	and </w>	1
models </w>	2	deploy </w>	1
Floydhub </w>	1	Build </w>	1
is </w>	1	models </w>	1
fastest </w>	1	in </w>	1
to </w>	1	train </w>	1

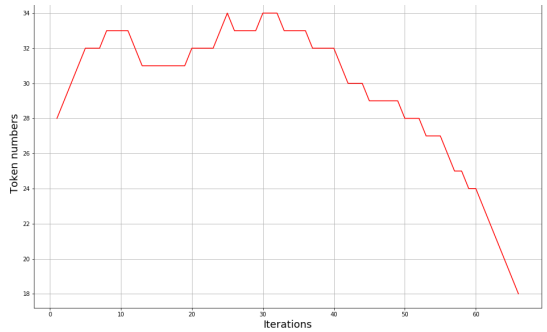
If you add up the frequency of the word in which "**de**" appears in you get 3 + 2 + 1 + 1 = 7 which is the frequency of our new "**de**" token. Since "**de**" is a new token we need to recalculate the counts for all our tokens. We do this by subtracting the frequency of the new "**de**" token, 7, from the frequency of the individual tokens before the merging operation. This makes sense if you think about it. We have just created a new token "**de**". This occurs 7 times in our dataset. Now we only want to count the times "**d**" and "**e**" occur when not paired together. To do this we subtract 7 from the original frequency of occurrence of "**e**", 16, to get 9. And we subtract 7 from the original frequency of "**d**", 12, to get 5. You can see this in the "iteration 1" table.

Let's do another iteration and see what the next most frequent

2	e	9	18	m	3
3	d	5	19	.	3
4	l	11	20	b	2
5	n	10 - 6 = 4	21	h	2
6	i	9 - 6 = 3	22	F	1
7	a	8	23	H	1
8	o	7	24	f	1
9	de	7	25	w	1
10	s	6	26	,	1
11	t	6	27	B	1
12	in	6	28	c	1
13	r	5	29	T	1
14	u	4			
15	p	4			
16	y	3			

Iteration 2, Best pair: ('i', 'n')

Again, we add a new token, bringing the number of tokens to 29, so we have actually increased the number of tokens after 2 iterations. This is common; as we start to create new merged pairs the number of tokens increases but then begins decreasing as we combine those together and remove other tokens. You can see this number change as we go through different iterations here:



Change in the number of token over 70 iterations

As you can see the number of tokens initially increases as we start

point subword units start to merge and we begin eliminating one of both of the merged pairs. We then build up our tokens into a format which can represent the entire dataset in the most efficient way. For our example here we stop at around 70 iterations with 18 tokens. In fact, we have recreated the original words from a starting point of individual character tokens. The final token list looks like:

WORD	FREQUENCY	WORD	FREQUENCY
deep </w>	3	build </w>	1
learning </w>	3	train </w>	1
the </w>	2	and </w>	1
models </w>	2	deploy </w>	1
Floydhub </w>	1	Build </w>	1
is </w>	1	models </w>	1
fastest </w>	1	in </w>	1
way </w>	1	cloud </w>	1
to </w>	1	Train </w>	1

original word list by starting with the individual characters and merging the most frequent byte pair tokens over a number of iterations (If you use smaller iterations you will see different token lists). While this may seem pointless remember that this was a toy dataset and the goal was to show the steps taken for subword tokenization. In a real world example the vocabulary size of your dataset should be much larger. Then you would not be able to have a token for each word in your vocabulary.

Probabilistic Subword Tokenization

For BPE we used the frequency of words to help identify which tokens to merge to create our token set. BPE ensures that the most common words will be represented in the new vocabulary as a single token, while less common words will be broken down into two or more subword tokens. To achieve this, BPE will go through every potential option at each step and pick the tokens to merge based on the highest

the best solution at each step in its iteration.

BPE algorithm is a token set like the one we generated earlier. This token set is used to encode the text for the input to your model. The problem occurs when there is more than one way to encode a particular word. How do you choose which subword units to use? You don't have any way to prioritize which subword tokens to use first. As a simple example, pretend our final token set for our toy example was the following subword tokens:

NUMBER	TOKEN	NUMBER	TOKEN
1	d	6	p
2	de	7	le
3	ee	8	ar
4	ep	9	n
5	e	10	ing

using our token set:

NUMBER	SUBWORDS (_ means space)	TEXT
1	de/ep/_le/ar/n/ing	deep learning
2	d/ee/p/_le/ar/n/ing	deep learning
3	d/e/ep/_le/ar/n/ing	deep learning

So while the input text is the same it can be represented by three different encodings. This is a problem for your language models as the embeddings generated will be different. These three different sequences will appear as three different input embeddings to be learned by your language model. This will impact the accuracy of your learned representations as your model will learn that the phrase "***deep learning***" appears in different context when in fact it should be the same relational context. To address this we need some way to rank or prioritize the encoding steps so that we end up with the same

probabilistic subword models such as unigram.

We have seen that using the frequency of subword patterns for tokenization can result in ambiguous final encodings. The problem is that we have no way to predict which particular token is more likely to be the best one when encoding any new input text. Luckily, needing to predict the most likely sequence of text is not a unique problem to tokenization. We can leverage this knowledge to build a better tokenizer.

The goal of a language model, which in some form or another underpins all current deep learning models such as BERT or GPT2, is to be able to predict a sequence of text given some initial state. For example, given the input text "***FloydHub is the fastest way to build, train and deploy deep ????***", can you predict the next text sequence? Is it "***deep ... sea***", "***deep ... space***", "***deep ... learning***" or "***deep ... sleep***". A well trained language model should be able to provide a probability for which is most likely given

take into account. However, the more words we take into account the

dimensionality of our LM and makes the conditional probability more difficult to calculate.

To address this complexity the simplest approach is the unigram model which only considers the probability of the current word. How likely it is that the next word is "***learning***" depends only on the probability of the word "***learning***" turning up in the training set. Now, this is not ideal when we are creating a model that is trying to predict a coherent sentence from some starting point. You would want to use a model with a larger training sequence such as a LM that looks at the preceding 2-3 words. This will have a better chance of generating a more coherent sentence as we can see even from our simple example. The goal for a subword model, however, is different from a LM that is trying to predict a full sentence. We only want something that generates unambiguous tokenization.

unigram LM approach to choose subword tokens. It is a great paper to check out since it describes the BPE approach as well and goes through its advantages and disadvantages. There is some math in the paper which covers the probability side of things but even that is well explained. The unigram approach differs from BPE in that it attempts to choose the most likely option rather than the best option at each iteration. To generate a unigram subword token set you need to first define the desired final size of your token set and also a starting seed subword token set. You can choose the seed subword token set in a similar way to BPE and choose the most frequently occurring substrings. Once you have this in place then you need to:

1. Work out the probability for each subword token
2. Work out a loss value which would result if each subwork token were to be dropped. The loss is worked out via an algorithm described in the paper (an expectation maximization algorithm).

choose a value here, e.g. drop the bottom 10% or 20% of

need to keep single characters to be able to deal with out of vocabulary words.

4. Repeat these steps until you reach your desired final vocabulary size or until there is no change in token numbers after successive iterations.

The Story so Far

So far we have explained why we need to tokenize input text sequences for deep learning NLP models. We then looked at some of the common approaches to tokenizing text and then we reviewed two recent models for subword tokenization in the form of BPE and unigram. Knowing something about both of these latter models means you should be able to understand nearly all of the tokenizer approaches currently used in deep learning NLP.

Most models will either use these directly or some variant of them

used in the latest and greatest NLP deep learning model. The reason

of. You might then think it was a waste of time knowing anything about unigram or BPE models. But don't worry, all is not lost! What happens in most cases is that after a little digging you begin to see that these "new" methods are actually pretty similar to either BPE or unigram.

An example of this is the tokenizer used in BERT, which is called "WordPiece". We will go through that algorithm and show how it is similar to the BPE model discussed earlier. We will finish up by looking at the "SentencePiece" algorithm which is used in the Universal Sentence Encoder Multilingual model released recently in 2019. SentencePiece brings together all of the concepts that we have spoken about, so it is a great way to summarize what we have covered so far. It also has a great open source repo that lets you take it for a test drive, so we can go through some code examples.

WordPiece

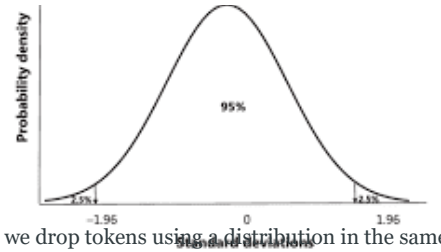
universe, evolving rapidly in a short space of time. So when BERT

to square one and need to figure out another subword model. However, WordPiece turns out to be very similar to BPE.

Think of WordPiece as an intermediary between the BPE approach and the unigram approach. BPE, if you remember, takes two tokens, looks at the frequency of each pair and then merges the pairs that have the highest combined frequency count. It only considers the most frequent pair combinations at each step, nothing else.

An alternate approach is to check the potential impact of merging that particular pair. You can do this using the probabilistic LM approach. At each iterative step, choose the character pair which will result in the largest increase in likelihood once merged. This is the difference between the probability of the new meged pair occurring minus the probability of both individual tokens occurring individually. For example, if "**de**" is more likely to occur than the

This is why I said WordPiece seems to be a bridge between both BPE



With distributions you can choose how much of the information you want to keep

In unigram we drop tokens using a distribution in the same way we would look for the skinny "tail" of a normal distribution. This has a lower density, or in our case would mean these are the tokens which are least likely to result in better tokenization.

Take A Breath!

Phew! I know, right? Who thought this part of the deep learning NLP process would be so difficult? And this is only the first step for these models- we haven't even got to the main part where we train them to

also uses the unigram approach to identify which to merge tokens.

You might be wondering - how is it different from the unigram model itself? That is a good question. The main difference is that WordPiece is a greedy approach. It still tries to build a tokenizer from the bottom up, picking the best pair at each iteration to merge. WordPiece uses the likelihood rather than count frequency but otherwise it is a similar approach. Unigram in contrast is a fully probabilistic approach which uses probability to both choose the pairs to merge and whether to merge them or not. It also removes tokens based on the fact that they add the least to the overall likelihood of the unigram model. Think of dropping the tokens which are at the tail ends of a normal distribution. Note that the individual character tokens will never be dropped since they will be needed to construct potentially out of vocabulary words.

best match at every iteration until it reaches the predefined vocabulary size.

2. **WordPiece**: Similar to BPE and uses frequency occurrences to identify potential merges but makes the final decision based on the likelihood of the merged token
3. **Unigram**: A fully probabilistic model which does not use frequency occurrences. Instead, it trains a LM using a probabilistic model, removing the token which improves the overall likelihood the least and then starting over until it reaches the final token limit.

Frequency V probability approaches

It is likely that you could use the BPE model with BERT instead of WordPiece and get similar results. Don't hold me to this but it should

which just might be the final step in this recent tokenization

will look at how and hopefully it will help us bring together all of the concepts we have discussed.

SentencePiece

SentencePiece basically tries to bring all the subword tokenization tools and techniques under one banner. It's kind of like the Swiss Army knife for subword tokenization. To be a Swiss Army-like tool something has to be capable of solving multiple problems. So what problems is SentencePiece addressing:

1. **All other models assume input is already tokenized**: BPE and Unigram are great models but they share one big disadvantage- they both need to have their input already tokenized. BPE needs to have the input tokenized so that every character (including word-boundary characters) are tokenized. Only then can BPE count frequencies and start to

with tokenization since not all languages are space segmented.

before it can start discarding tokens based on their probability distribution. SentencePiece deals with this by simply taking in an input in raw text and then doing everything (which we will discuss below) needed on that input to perform subword tokenization.

2. **Language agnostic**: Since all other subword algorithms need to have their input pre-tokenized, it limits their applicability to many languages. You have to create rules for different languages to be able to use them as input into your model. This gets very messy very quickly.
3. **Decoding is difficult**: Another problem which is caused by models like BPE and unigram requiring already tokenized inputs is you do not know what encoding rules were used. For example, how were spaces encoded in your tokens? Did the encoding rules differentiate between spaces and tabs? If you see two tokens like [new] and [york] together you cannot know

to reproduce results or confirm findings.

4. **No end to end solution**: These are just some of the issues, which means that BPE and unigram are not fully complete or end-to-end solutions. You cannot just plug in a raw input and get an output. Instead they are only part of a solution. SentencePiece gathers everything needed for an end-to-end solution under one neat package.

How does SentencePiece fix all these issues?

SentencePiece uses a number of features that address all of the above issues, which are outlined in detail in both the related paper and the corresponding GitHub repo. Both of these are great resources, but if you are short on time then skimming the repo might be the best way to get a quick overview of SentencePiece and all of its associated Swiss Army greatness.

uses to address the above shortcomings before diving into some code

1. **Encode everything as unicode ...**: SentencePiece first converts all the input into unicode characters. This means it doesn't have to worry about different languages or characters or symbols. If it uses unicode it can just treat all input in the same way, which allows it to be language agnostic
2. **... including the spaces**: To get around the word segmenting issues, SentencePiece simply encodes spaces as a unicode symbol. Specifically it encodes it as unicode value U+2581 (underscore '_' to those of us who don't speak unicode). This helps with the language agnostic issues and the decoding issue. Since spaces are unicode encoded then they can be easily reversed or decoded and treated (i.e learned) like a normal language character. It sounds like a simple approach and I guess it is, but the best ideas tend to seem that way in the end
3. **And it's faster**: Google famously noted that "speed isn't just

issues preventing other subword algorithms from being used

there lack of speed. If you processed input in real time and performed your tokenization on the raw input it would be too slow. SentencePiece addresses this by using a priority queue for the BPE algorithm to speed it up so that you can use it as part of an end-to-end solution.

Using these and other features, SentencePiece is able to provide fast and robust subword tokenization on any input language for any deep learning NLP model. Now let's look at it in action.

SentencePiece in action

The SentencePiece Github repo provides some great examples of how to use the library. We will try out some of these to showcase some of the subword algorithms we have spoken about already. However, there is much more to the SentencePiece library than we can go into in a few blog posts. We will aim to cover the main features here to

things we will cover are:

to train it on some data. SentencePiece suggests training on a novel called "Botchan". While this is fine, it is a Japanese novel written in 1906. So we can find something a bit more up to date. We can use one of the open source datasets made available by lionbridge.ai. It includes this list of datasets for NLP from which we can choose the Blogger Corpus of over 600,000 blog posts (we only need a fraction of this). It should contain a variety of data, common words, slang, misspellings, names and entities and so on.

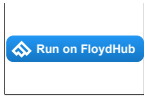
2. **Train a BPE model**: After creating our dataset we can train a BPE model so that we end up with a list of BPE tokens we can use for encoding.
3. **Train a Unigram model**: Similarly, we can train a unigram model on the same data so we can have unigram-specific tokens.
4. **Compare the models**: Using the trained models we will

models**Perform some sampling**: As we noted, the unigram

provides functions to sample from this distribution. These features are only available in the unigram model.

Get Your Training Data

Check out the **code repo** to see the how to use the blog corpus for training. Note that you can use any data that you like here. This is just a suggestion. All the code for the steps below are in the notebook so you can follow along with all the steps below via the notebook.



Train a BPE Model

We can train a model quite easily using SentencePiece. For now we don't need to worry about the parameters we use in the training cmd


```
# train sentencepiece model from our blog corpus
```


The main parameter to note is the "**vocab_size**". We set this to 500 just as an example, but you can choose anything you like here. Remember, the larger your vocab size the more common words you will be able to store, but you may want a smaller vocab size for your application for performance reasons.

Once you have trained your model, you just need to load it and you are ready to go!

```
# makes segmenter instance and loads the BPE model file (bpe.model)
sp_bpe = spm.SentencePieceProcessor()
sp_bpe.load('bpe.model')
```

Train a Unigram Model

Now we just need to train the Unigram model and then we can compare the two. You can train the Unigram model in much the same way as the BPE model. Just remember to name them differently!


```
# train sentencepiece model from our blog corpus
```

```
# makes segmenter instance and loads the BPE model file (bpe.model)
sp_uni = spm.SentencePieceProcessor()
sp_uni.load('uni.model')
```

Let's Compare the Models

You can encode a sentence with the trained subword tokens by calling the "encode_as_pieces" function. Let's encode the following sentence: "***This is a test***".

```
print("BPE: {}".format(sp_bpe.encode_as_pieces('This is a test')))
print("UNI: {}".format(sp_uni.encode_as_pieces('This is a test')))
```

BPE: ['_This', '_is', '_a', '_t', 'est']

UNI: ['_Thi', 's', '_is', '_a', '_t', 'est']

The underscore indicates that there is a space with the token and it is

we did not put it there. It assumes there is one since that word is at

will be encoded in the same way.

Interesting things to note here are that there is no word for "***This***" or "***test***" in the Unigram model, but there is for "***This***" in BPE. In a different dataset these words might have been more common if we chose a larger vocab size. For a blog you would think "***this***" would be a popular word. Maybe it is the capital "***T***" which is causing it to be encoded differently? Let's try "***I think this is a test***".

BPE: ['_I', '_think', '_this', '_is', '_a', '_t', 'est']

UNI: ['_I', '_think', '_this', '_is', '_a', '_t', 'est']

So there is a word for "***this***"! It does not occur with a capital '***T***" enough to be encoded as a specific token for the Unigram model (or it did not increase the merging likelihood enough to be implemented). Since there is no word for "***test***" the model created it via the

UNI: ['_C', 'ar', 'b', 'on', '_d', 'i', 'o', 'x', 'id', 'e']

So it looks like this is not a common word and thus is made up from a collection of more common subword tokens and a few single letter tokens.

Let's See All Our Tokens

To see all the tokens that were created we can run the following code to see the full list.

```
vocabs = [sp_bpe.id_to_piece(id) for id in range(sp_bpe.get_piece_size())]
bpe_tokens = sorted(vocabs, key=lambda x: len(x), reverse=True)
bpe_tokens
```

We will get a list of 500 tokens (which was our predefined limit) which should represent the most common words followed by the most common subword combinations. You can see from the code that

into its corresponding textual representation.


```
'_because',
'_thought',
'_really',
.
.
.
'9',
'*',
'8',
'6',
'7',
'$']
```

BPE tokens (sorted by length of token string)

This is important since SentencePiece enables the subword process to be reversible. You can encode your test sentence in ID's or in subword tokens; what you use is up to you. The key is that you can decode either the IDs or the tokens perfectly back into the original sentences, including the original spaces. Previously this was not possible with

```
# decode: id => text
print("BPE {}".format(sp_bpe.decode_pieces(['_This', '_is', '_a', '_t', 'est'])))
print("BPE {}".format(sp_bpe.decode_ids([400, 61, 4, 3, 231])))

print("UNI {}".format(sp_uni.decode_pieces(['_Thi', 's', '_is', '_a', '_t', 'est'])))
print("UNI {}".format(sp_uni.decode_ids([284, 3, 37, 15, 78, 338])))
```

Reversing the encoding process

BPE This is a test

BPE This is a test

UNI This is a test

UNI This is a test

Anyway, back to our token list. It would be nice to look at the list of

0	__everything	__friend
1	__different	__somet
2	__actually	__thou
3	__remember	other
4	__anything	ittle

These are good examples of BPE's greedy approach. Do we really need a token for "**_somet**"? The Unigram model must have calculated that the overall benefit of using this is less than simply using "**_some**" along with some other subword units. But for BPE it just checks the most frequent pairing at each step.

Similarly for "**ittle**", is it efficient to have this token? If you look in the training data text "little" occurs 159 times. 156 of those are "**little**", the other remaining occasions are one "**belittle**" and two

"whittle", "tittle" and "skittle" then it might make sense. But given the

~~We can then simply get Unigram tokens which are not in~~ the BPE list and vice versa. It is interesting to see which tokens are missing from each set. This will tell us something about the different approaches of each subword model. Remember when looking at these tokens the "__" represents a space. When it does not appear it means the word is part of another one or attached to a full stop or comma or some symbol other than a space. When you start tinkering with the encoding this will become clearer.

If we look at some of the BPE tokens which are not in the Unigam model, we see examples like "**_somet**" and "**ittle**".

```
diff_pairs = list(zip(uni_tok_diff, bpe_tok_diff))
diff_df = pd.DataFrame(diff_pairs,
                       columns=(["Unigram tokens not in BPE", "BPE tokens not
diff_df.head()
```


more efficient. This shows the benefit of using the Unigram approach if you want to have the most efficient subword vocabulary,

Let's Do Some Sampling

And last but not least, let's take a look at the Unigram sampling functionality. This is only available for the Unigram model since BPE is a frequency-based approach. The sampling functionality of SentencePiece allows you to set the sampling parameters. By default you get the most efficient tokenization, but you can change this if you like. These are very fine-grained and advanced settings which are mentioned in the original paper. The "**nbest**" parameter allows you to select from more segmentation options. The higher the parameter, the more options will be considered. While this is pretty advanced and it's difficult to know when you might need to change it, you can at least look at the different tokens returned when you alter these settings.

```
for n in range(10):
  print(sp_uni.sample_encode_as_pieces('remembers', -1, 0.1))
```

['_re', 'me', 'm', 'b', 'er', 's']

['_', 're', 'm', 'e', 'm', 'b', 'e', 'r', 's']

['_remember', 's']

['_remember', 's']

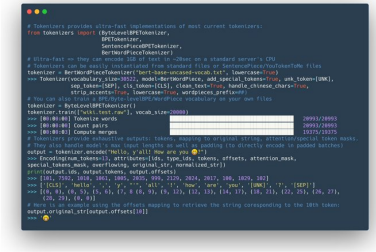
['_remember', 's']

['_', 're', 'me', 'm', 'b', 'er', 's']

['_', 'r', 'e', 'me', 'm', 'b', 'er', 's']

['_re', 'me', 'm', 'b', 'e', 'r', 's']

['_', 'r', 'e', 'me', 'm', 'b', 'er', 's']



951 10:23 PM - Jan 10, 2020

268 people are talking about this

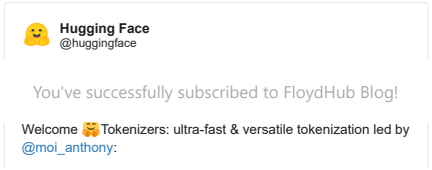
As we have shown, the SentencePiece library contains everything you need for the BPE and Unigram models. But if you want to use other models such as WordPiece you will need to set that up separately. HuggingFace have all of these under one handy GitHub roof. So let's train these tokens on our blog data and show how easy it is to use.

With a few lines of code you have access to a whole range of

You can read more about this in the **subword sampling section** of the paper where they discuss the "l-best" approach to sampling.

HuggingFace Tokenizers to the Rescue!

Those great people at HuggingFace have done it again. There latest addition to their already impressive NLP library is, yep, you guessed it, tokenizers. The handy thing about the HuggingFace tokenizers is that they hide much of the complicated details we discussed in this post (don't worry, it's still really useful to have a working knowledge of how they work and this will help you with libraries like the HuggingFace tokenizer, I promise).



it on the same test dataset and then simply print the tokens:

```
from tokenizers import (ByteLevelBPETokenizer,
                        BPETokenizer,
                        SentencePieceBPETokenizer,
                        BertWordPieceTokenizer)

tokenizer = SentencePieceBPETokenizer()
tokenizer.train(["../blog_test.txt"], vocab_size=500, min_frequency=2)

output = tokenizer.encode("This is a test")
print(output.tokens)
```

BPE: ['_Th', 'is', '_is', '_a', '_t', 'est']

You can see that this is different from the token for our BPE algorithm which we implemented via SentencePiece:

BPE: ['_This', '_is', '_a', '_t', 'est']

This is possibly due to the different parameters you can set or the

potential issues with using different libraries. *If you use*

same model trained on the same data.

Hopefully, it should be easy enough to align the parameters so that these libraries can be used interchangeably. Alternatively, whichever library you use it may be best to stick with that in all your projects for consistency. It's too early to tell but one of these libraries may just prove to be the defacto standard that most people use going forward.

Conclusion

With the rapid pace of development in Deep Learning, it can be easy to look only at the main, meaty core of models like BERT and XLNet. However, the format used to enable these models to process text is central to how they learn. Understanding the basics about subword tokenizers will give you a way to quickly get to grips with the latest innovations in this field. You do not need to start from scratch when reading up on the latest model hot off the academic press. It will also

way, knowing something about models such as SentencePiece will be

Further Reading

Tokenization is a surprisingly complex topic once you start to get into the finer details of each model. It seems like it is it's own separate research area outside of the more widely know areas such as the LM architecture and models like ELMo, BERT and the Transformer models. So I leant on a wide range of source to try and better understand the area. Here are some of the most helpful resources I found. So If you want to know more about tokenization (or you think I have got it all wrong !) then I recommend the following material:

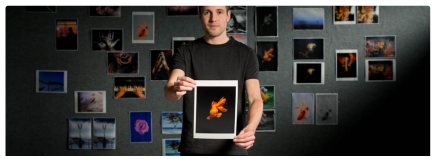
1. **A Deep Dive into the Wonderful World of Preprocessing in NLP**

5. **Tokenization tooling**
6. **Google SentencePiece repo**
7. **Unicode Normalization**

About Cathal Horan

Cathal is interested in the intersection of philosophy and technology, and is particularly fascinated by how technologies like deep learning can help augment and improve human decision making. He recently completed an MSc in business analytics. His primary degree is in electrical and electronic engineering, but he also boasts a degree in philosophy and an MPhil in psychoanalytic studies. He currently works at Intercom. Cathal is also a FloydHub AI Writer.

Intercom blog.



This Humans of Machine Learning interview covers Emil Wallner and his hero's journey, self-taught approach to education, experience with AI, and path to

ALESSIO GOZZOLI, EMIL WALLNER
20 JAN 2020 • 12 MIN READ