SentencePiece and Byte Pair Encoding

Introduction to Tokenization

In order to process text in neural network models, it is first required to **encode** text as numbers with ids (such as the embedding vectors we've been using in the previous assignments), since the tensor operations act on numbers. Finally, if the output of the network are words, it is required to **decode** the predicted tokens ids back to text.

To encode text, the first decision that has to be made is to what level of granularity are we going to consider the text? Because ultimately, from these **tokens**, features are going to be created about them. Many different experiments have been carried out using *words*, *morphological units*, *phonemic units*, *characters*. For example,

- Tokens are tricky. (raw text)
- Tokens are tricky . (words)
- Token s _ are _ trick _ y . (morphemes)
- t oʊ k ə n z _ ɑː _ ˈ t r ɪ k i. (phonemes, for STT)
- T o k e n s _ a r e _ t r i c k y . (character)

But how to identify these units, such as words, are largely determined by the language they come from. For example, in many European languages a space is used to separate words, while in some Asian languages there are no spaces between words. Compare English and Mandarin.

- Tokens are tricky. (original sentence)
- 令牌很棘手 (Mandarin)
- Lìng pái hěn jí shǒu (pinyin)
- 令牌 很 棘手 (Mandarin with spaces)

So, the ability to **tokenize**, i.e. split text into meaningful fundamental units is not always straight-forward.

Also, there are practical issues of how large our *vocabulary* of words, `vocab_size`, should be, considering memory limitations vs. coverage. A compromise between the finest-grained models employing characters which can be memory and more computationally efficient *subword* units such as n-grams or larger units need to be made.

In SentencePiece unicode characters are grouped together using either a unigram language model (used in this week's assignment) or BPE, **byte-pair encoding**. We will discuss BPE, since BERT and many of its variant uses a modified version of BPE and its pseudocode is easy to implement and understand... hopefully!

SentencePiece Preprocessing

NFKC Normalization

Unsurprisingly, even using unicode to initially tokenize text can be ambiguous, e.g.,

In [1]:
```
eaccent = '\u00E9'
e_accent = '\u0065\u0301'
print(f'{eaccent} = {e_accent} : {eaccent == e_accent}')
```

```
é = é : False
```

SentencePiece uses the Unicode standard Normalization form, NFKC, so this isn't an issue. Looking at our example from above again with normalization:

In [2]:
```
from unicodedata import normalize

norm_eaccent = normalize('NFKC', '\u00E9')
norm_e_accent = normalize('NFKC', '\u0065\u0301')
print(f'{norm_eaccent} = {norm_e_accent} : {norm_eaccent == norm_e_accent}')
```

```
é = é : True
```

Normalization has actually changed the unicode code point (unicode unique id) for one of these two characters.

In [3]:
```
def get_hex_encoding(s):
    return ' '.join(hex(ord(c)) for c in s)

def print_string_and_encoding(s):
    print(f'{s} : {get_hex_encoding(s)}')
```

In [4]:
```
for s in [eaccent, e_accent, norm_eaccent, norm_e_accent]:
    print_string_and_encoding(s)
```

```
é : 0xe9
é : 0x65 0x301
é : 0xe9
é : 0xe9
```

This normalization has other side effects which may be considered useful such as converting curly quotes “ to " their ASCII equivalent. (Although we *now* lose directionality of the quote...)

Lossless Tokenization*

SentencePiece also ensures that when you tokenize your data and detokenize your data the original position of white space is preserved. (However, tabs and newlines are converted to spaces, please try this experiment yourself later below.)

To ensure this **lossless tokenization** it replaces white space with _ (U+2581). So that a simple join of the replace underscores with spaces can restore the white space, even if there are consecutives symbols. But remember first to normalize and then replace spaces with _ (U+2581). As the following example shows.

In [5]:
```
s = 'Tokenization is hard.'
s_ = s.replace(' ', '\u2581')
s_n = normalize('NFKC', 'Tokenization is hard.')
```

In [6]:
```
print(get_hex_encoding(s))
print(get_hex_encoding(s_))
print(get_hex_encoding(s_n))
```

```
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x20 0x69 0x73 0x20 0x68 0x61 0x72 0x64 0x2e
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x2581 0x69 0x73 0x2581 0x68 0x61 0x72 0x64 0x2e
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x20 0x69 0x73 0x20 0x68 0x61 0x72 0x64 0x2e
```

So the special unicode underscore was replaced by the ASCII unicode. Reversing the order, we see that the special unicode underscore was retained.

In [7]:
```
s = 'Tokenization is hard.'
sn = normalize('NFKC', 'Tokenization is hard.')
sn_ = s.replace(' ', '\u2581')
```

In [8]:
```
print(get_hex_encoding(s))
print(get_hex_encoding(sn))
print(get_hex_encoding(sn_))
```

```
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x20 0x69 0x73 0x20 0x68 0x61 0x72 0x64 0x2e
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x20 0x69 0x73 0x20 0x68 0x61 0x72 0x64 0x2e
0x54 0x6f 0x6b 0x65 0x6e 0x69 0x7a 0x61 0x74 0x69 0x6f 0x6e 0x2581 0x69 0x73 0x2581 0x68 0x61 0x72 0x64 0x2e
```

BPE Algorithm

Now that we have discussed the preprocessing that SentencePiece performs we will go get our data, preprocess, and apply the BPE algorithm. We will show how this reproduces the tokenization produced by training SentencePiece on our example dataset (from this week's assignment).

Preparing our Data

First, we get our Squad data and process as above.

In [9]:
```
import ast

def convert_json_examples_to_text(filepath):
    example_jsons = list(map(ast.literal_eval, open(filepath))) # Read in the json from the example file
    texts = [example_json['text'].decode('utf-8') for example_json in example_jsons] # Decode the byte sequences
    text = '\n\n'.join(texts)       # Separate different articles by two newlines
    text = normalize('NFKC', text)  # Normalize the text

    with open('example.txt', 'w') as fw:
        fw.write(text)

    return text
```

In [10]:
```
text = convert_json_examples_to_text('data.txt')
print(text[:900])
```

```
Beginners BBQ Class Taking Place in Missoula!
Do you want to get better at making delicious BBQ? You will have the opportunity, put this on your calendar now. Thursday, S
eptember 22nd join World Class BBQ Champion, Tony Balay from Lonestar Smoke Rangers. He will be teaching a beginner level cl
ass for everyone who wants to get better with their culinary skills.
He will teach you everything you need to know to compete in a KCBS BBQ competition, including techniques, recipes, timeline
s, meat selection and trimming, plus smoker and fire information.
The cost to be in the class is $35 per person, and for spectators it is free. Included in the cost will be either a t-shirt
or apron and you will be tasting samples of each meat that is prepared.

Discussion in 'Mac OS X Lion (10.7)' started by axboi87, Jan 20, 2012.
I've got a 500gb internal drive and a 240gb SSD.
When trying to restore using di
```

In the algorithm the `vocab` variable is actually a frequency dictionary of the words. Further, those words have been prepended with an *underscore* to indicate that they are the beginning of a word. Finally, the characters have been delimited by spaces so that the BPE algorithm can group the most common characters together in the dictionary in a greedy fashion. We will see how that is exactly done shortly.

In [11]:
```
from collections import Counter

vocab = Counter(['\u2581' + word for word in text.split()])
vocab = {' '.join([l for l in word]): freq for word, freq in vocab.items()}
```

In [12]:
```
def show_vocab(vocab, end='\n', limit=20):
    shown = 0
    for word, freq in vocab.items():
        print(f'{word}: {freq}', end=end)
        shown +=1
        if shown > limit:
            break
```

In [13]:
```
show_vocab(vocab)
```

```
_ B e g i n n e r s: 1
_ B B Q: 3
_ C l a s s: 2
_ T a k i n g: 1
_ P l a c e: 1
_ i n: 15
_ M i s s o u l a !: 1
_ D o: 1
_ y o u: 13
_ w a n t: 1
_ t o: 33
_ g e t: 2
_ b e t t e r: 2
_ a t: 1
_ m a k i n g: 2
_ d e l i c i o u s: 1
_ B B Q ?: 1
_ Y o u: 1
_ w i l l: 6
_ h a v e: 4
_ t h e: 31
```

We check the size of the vocabulary (frequency dictionary) because this is the one hyperparameter that BPE depends on crucially on how far it breaks up a word into SentencePieces. It turns out that for our trained model on our small dataset that 60% of 455 merges of the most frequent characters need to be done to reproduce the upperlimit of a 32K `vocab_size` over the entire corpus of examples.

In [14]:
```
print(f'Total number of unique words: {len(vocab)}')
print(f'Number of merges required to reproduce SentencePiece training on the whole corpus: {int(0.60*len(vocab))}')
```

```
Total number of unique words: 455
Number of merges required to reproduce SentencePiece training on the whole corpus: 273
```

BPE Algorithm

Directly from the BPE paper we have the following algorithm.

To understand what's going on first take a look at the third function `get_sentence_piece_vocab`. It takes in the current `vocab` word-frequency dictionary and the fraction of the total `vocab_size` to merge characters in the words of the dictionary, `num_merges` times. Then for each *merge* operation it `get_stats` on how many of each pair of character sequences there are. It gets the most frequent *pair* of symbols as the `best` pair. Then it merges those pair of symbols (removes the space between them) in each word in the `vocab` that contains this `best` (= `pair`). Consquently, `merge_vocab` creates a new `vocab`, `v_out`. This process is repeated `num_merges` times and the result is the set of SentencePieces (keys of the final `sp_vocab`).

Please feel free to skip the below if the above description was enough.

In a little more detail then, we can see in `get_stats` we initially create a list of bigram frequencies (two character sequence) from our vocabulary. Later, this may include (trigrams, quadgrams, etc.). Note that the key of the `pairs` frequency dictionary is actually a 2-tuple, which is just shorthand notation for a pair.

In `merge_vocab` we take in an individual `pair` (of character sequences, note this is the most frequency `best` pair) and the current `vocab` as `v_in`. We create a new `vocab`, `v_out`, from the old by joining together the characters in the pair (removing the space), if they are present in the a word of the dictionary. Warning: the expression `(?<!\S)` means that either whitespace character follows before the `bigram` or there is nothing before (beginning of word) the bigram, similarly for `(?!\S)` for preceding whitespace or end of word.

In [15]:
```
import re, collections

def get_stats(vocab):
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols) - 1):
            pairs[symbols[i], symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
    for word in v_in:
        w_out = p.sub(''.join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out

def get_sentence_piece_vocab(vocab, frac_merges=0.60):
    sp_vocab = vocab.copy()
    num_merges = int(len(sp_vocab)*frac_merges)

    for i in range(num_merges):
        pairs = get_stats(sp_vocab)
        best = max(pairs, key=pairs.get)
        sp_vocab = merge_vocab(best, sp_vocab)

    return sp_vocab
```

In [16]:
```
sp_vocab = get_sentence_piece_vocab(vocab)
show_vocab(sp_vocab)
```

```
_B e g in n ers: 1
_BBQ: 3
_Cl ass: 2
_T ak ing: 1
_P la ce: 1
_in: 15
_M is s ou la !: 1
_D o: 1
_you: 13
_w an t: 1
_to: 33
_g et: 2
_be t ter: 2
_a t: 1
_mak ing: 2
_d e l ic i ou s: 1
_BBQ ?: 1
__ Y ou: 1
_will: 6
_have: 4
_the: 31
```

Train SentencePiece BPE Tokenizer on Example Data

Explore SentencePiece Model

First let us explore the SentencePiece model provided with this week's assignment. Remember you can always use Python's built in `help` command to see the documentation for any object or method.

In [17]:
```
import sentencepiece as spm
sp = spm.SentencePieceProcessor(model_file='sentencepiece.model')
```

In [18]:
```
help(sp)
```

```
Help on SentencePieceProcessor in module sentencepiece object:

class SentencePieceProcessor(builtins.object)
 |  SentencePieceProcessor(model_file=None, model_proto=None, out_type=<class 'int'>, add_bos=False, add_eos=False, reverse=False, enable_sampling=False, nbest_size=-1, alpha=0.1)
 |
 |  Methods defined here:
 |
 |  Decode(self, input)
 |      Decode processed id or token sequences.
 |
 |  DecodeIds(self, ids)
 |
 |  DecodeIdsAsSerializedProto(self, ids)
 |
 |  DecodePieces(self, pieces)
 |
 |  DecodePiecesAsSerializedProto(self, pieces)
 |
 |  Detokenize = Decode(self, input)
```

Let's work with the first sentence of our example text.

In [19]:
```
s0 = 'Beginners BBQ Class Taking Place in Missoula!'
```

In [20]:
```
# encode: text => id
print(sp.encode_as_pieces(s0))
print(sp.encode_as_ids(s0))

# decode: id => text
```

```
print(sp.decode_pieces(sp.encode_as_pieces(s0)))
print(sp.decode_ids([12847, 277]))
```

```
['_Beginn', 'ers', '_BBQ', '_Class', '_', 'Taking', '_Place', '_in', '_Miss', 'oul', 'a', '!']
[12847, 277, 15068, 4501, 3, 12297, 3399, 16, 5964, 7115, 9, 55]
Beginners BBQ Class Taking Place in Missoula!
Beginners
```

Notice how SentencePiece breaks the words into seemingly odd parts, but we've seen something similar from our work with BPE. But how close were we to this model trained on the whole corpus of examles with a `vocab_size` of 32,000 instead of 455? Here you can also test what happens to white space, like '\n'.

But first let us note that SentencePiece encodes the SentencePieces, the tokens, and has reserved some of the ids as can be seen in this week's assignment.

In [21]:
```
uid = 15068
spiece = "\u2581BBQ"
unknown = "__MUST_BE_UNKNOWN__"

# id <=> piece conversion
print(f'SentencePiece for ID {uid}: {sp.id_to_piece(uid)}')
print(f'ID for Sentence Piece {spiece}: {sp.piece_to_id(spiece)}')

# returns 0 for unknown tokens (we can change the id for UNK)
print(f'ID for unknown text {unknown}: {sp.piece_to_id(unknown)}')
```

```
SentencePiece for ID 15068: _BBQ
ID for Sentence Piece _BBQ: 15068
ID for unknown text __MUST_BE_UNKNOWN__: 2
```

In [22]:
```
print(f'Beginning of sentence id: {sp.bos_id()}')
print(f'Pad id: {sp.pad_id()}')
print(f'End of sentence id: {sp.eos_id()}')
print(f'Unknown id: {sp.unk_id()}')
print(f'Vocab size: {sp.vocab_size()}')
```

```
Beginning of sentence id: -1
Pad id: 0
End of sentence id: 1
Unknown id: 2
Vocab size: 32000
```

We can also check what are the ids for the first part and last part of the vocabulary.

In [23]:
```
print('\nId\tSentP\tControl?')
print('------------------------')
# <unk>, <s>, </s> are defined by default. Their ids are (0, 1, 2)
# <s> and </s> are defined as 'control' symbol.
for uid in range(10):
    print(uid, sp.id_to_piece(uid), sp.is_control(uid), sep='\t')

# for uid in range(sp.vocab_size()-10,sp.vocab_size()):
#     print(uid, sp.id_to_piece(uid), sp.is_control(uid), sep='\t')
```

```
Id      SentP   Control?
------------------------
0       <pad>   True
1       </s>    True
2       <unk>   False
3       _       False
4       X       False
5       .       False
6       ,       False
7       s       False
8       _the    False
9       a       False
```

Train SentencePiece BPE model with our example.txt

Finally, let's train our own BPE model directly from the SentencePiece library and compare it to the results of our implemention of the algorithm from the BPE paper itself.

In [24]:
```
spm.SentencePieceTrainer.train('--input=example.txt --model_prefix=example_bpe --vocab_size=450 --model_type=bpe')
sp_bpe = spm.SentencePieceProcessor()
sp_bpe.load('example_bpe.model')

print('*** BPE ***')
print(sp_bpe.encode_as_pieces(s0))
```

```
*** BPE ***
['_B', 'e', 'ginn', 'ers', '_BBQ', '_Cl', 'ass', '_T', 'ak', 'ing', '_P', 'la', 'ce', '_in', '_M', 'is', 's', 'ou', 'la', '!']
```

In [25]:
```
show_vocab(sp_vocab, end = ', ')
```

```
_B e g in n ers: 1, _BBQ: 3, _Cl ass: 2, _T ak ing: 1, _P la ce: 1, _in: 15, _M is s ou la !: 1, _D o: 1, _you: 13, _w an t: 1, _to: 33, _g et: 2, _be t ter: 2, _a t: 1, _mak ing: 2, _d e l ic i ou s: 1, _BBQ ?: 1, _ Y ou: 1, _will: 6, _have: 4, _the: 31,
```

Our implementation of BPE's code from the paper matches up pretty well with the library itself! Difference are probably accounted for by the `vocab_size`. There is also another technical difference in that in the SentencePiece implementation of BPE a priority queue is used to more efficiently keep track of the *best pairs*. Actually, there is a priority queue in the Python standard library called `heapq` if you would like to give that a try below!

In [26]:
```
from heapq import heappush, heappop
```

In [27]:
```
def heapsort(iterable):
    h = []
    for value in iterable:
        heappush(h, value)
    return [heappop(h) for i in range(len(h))]
```

In [28]:
```
a = [1,4,3,1,3,2,1,4,2]
heapsort(a)
```

Out[28]:
```
[1, 1, 1, 2, 2, 3, 3, 4, 4]
```

> For a more extensive example consider looking at the SentencePiece repo. The last section of this code is repurposed from that tutorial. Thanks for your participation!