LLM Part 2 : Byte Pair Encoding

Reference text

https://www.manning.com/books/build-a-large-language-model-from-scratch

Text Corpus

The text we will tokenize for LLM training is a short story by Edith Wharton called The Verdict, which has been released into the public domain and is thus permitted to be used for LLM training tasks. The text is available on Wikisource at https://en.wikisource.org/wiki/The_Verdict,

Problem Statement :

In the first notebook, we discussed how to develop a Word Tokenizer step by step. In this notebook we will demonstrate An advanced tokenization method called Byte Pair Encoding

Additional special tokens that could have been implemented further in the tokenizer developed in Part 1

- [BOS] (beginning of sequence): This token marks the start of a text. It signifies to the LLM where a piece of content begins.
- [EOS] (end of sequence): This token is positioned at the end of a text, and is especially useful when concatenating multiple unrelated texts, similar to <|endoftext|>. For instance, when combining two different Wikipedia articles or books, the [EOS] token indicates where one article ends and the next one begins.
- [PAD] (padding): When training LLMs with batch sizes larger than one, the batch might contain texts of varying lengths. To ensure all texts have the same length, the shorter texts are extended or "padded" using the [PAD] token, up to the length of the longest text in the batch.

What does the tokenizer for GPT models use ?

- The tokenizer used for GPT models does not need any of these tokens mentioned above but only uses an <|endoftext|> token for simplicity.
- The <|endoftext|> is analogous to the [EOS] token mentioned above. Also, <|endoftext|> is used for padding as well. However, as we'll explore in subsequent chapters when training on batched inputs, we typically use a mask, meaning we don't attend to padded tokens. Thus, the specific token chosen for padding becomes inconsequential.

- Moreover, the tokenizer used for GPT models also doesn't use an <|unk|> token for out-of-vocabulary words.

- Instead, GPT models use a **byte pair encoding tokenizer**, which breaks down words into subword units, which we will discuss in the next section.

How is the Byte Pair Encoding used by GPT-2 superior ?

- it allows the model to break down words that aren't in its predefined vocabulary into smaller subword units or even individual characters, enabling it to handle out-of-vocabulary words
- For instance, if GPT-2's vocabulary doesn't have the word "unfamiliarword," it might tokenize it as ["unfam", "iliar", "word"] or some other subword breakdown, depending on its trained BPE merges
- The original BPE tokenizer can be found here: https://github.com/openai/gpt-2/blob/master/src/encoder.py

Concept Note : Byte Pair Encoding

REFERENCE https://www.geeksforgeeks.org/byte-pair-encoding-bpe-in-nlp/

Concepts related to BPE:

- **Vocabulary:** A set of subword units that can be used to represent a text corpus.
- **Byte:** A unit of digital information that typically consists of eight bits.
- **Character:** A symbol that represents a written or printed letter or numeral.
- **Frequency:** The number of times a byte or character occurs in a text corpus.
- **Merge:** The process of combining two consecutive bytes or characters to create a new subword unit.

Steps involved in BPE:

- Initialize the vocabulary with all the bytes or characters in the text corpus
- Calculate the frequency of each byte or character in the text corpus.
- Repeat the following steps until the desired vocabulary size is reached:
  - Find the most frequent pair of consecutive bytes or characters in the text corpus
  - Merge the pair to create a new subword unit.
  - Update the frequency counts of all the bytes or characters that contain the merged pair.
  - Add the new subword unit to the vocabulary.
- Represent the text corpus using the subword units in the vocabulary.

How does BPE Work - A simple example

Suppose we have a text corpus with the following four words: “ab”, “bc”, “bcd”, and “cde”. The initial vocabulary consists of all the bytes or characters in the text corpus:

{“a”, “b”, “c”, “d”, “e”}

Step 1 : Initialize the vocabulary

Vocabulary = {"a", "b", "c", "d", "e"}

Step 2 : Compute Frequency

Frequency = {"a": 1, "b": 3, "c": 3, "d": 2, "e": 1}

Repeat Steps 3 to 5 until the desired vocabulary size is reached.

- #### Step 3 : Find the most frequent pair of two characters

  ```
  The most frequent pair is "bc" with a frequency of 2.
  ```
- #### Step 4 : Merge the pair

  ```
  Merge "bc" to create a new subword unit "bc"
  ```
- #### Step 5: Update frequency counts

  ```
  Frequency = {"a": 1, "b": 2, "c": 3, "d": 2, "e": 1, "bc": 2}
  ```

Represent the text corpus using subword units

The resulting vocabulary consists of the following subword units: {"a", "b", "c", "d", "e", "bc", "cd", "de","ab","bcd","cde"}.

Section A: Python Implementation of Byte Pair Encoding

We will define a series of functions to perfome the byte pair encoding as discussed above

1) get_vocab()

The get_vocab function is defined to take a list of strings (data) as input and return a dictionary mapping words (formatted as separated characters with an end token) to their frequency counts.

2) get_stats()

The get_stats function is defined to take a dictionary (vocab) as input and return a dictionary mapping tuples of character pairs to their frequency counts.

3) merge_vocab()

The merge_vocab function is defined to take a tuple of characters (pair) and a dictionary (v_in) as input, and return a new dictionary with the specified pair of characters merged.

byte pai

The byte_pair_encoding function is defined to take a list of strings (data) and an integer (n) as input, and return a dictionary representing the vocabulary with merged character pairs.

STEP 1. Define function get_vocab()

Note on creating the vocabulary dictionary**

- **vocab = defaultdict(int):** Initializes a defaultdict with int as the default factory, meaning any new key will have a default value of 0.
- The function iterates through each line in data and then through each word in the line.
- **vocab[' '.join(list(word)) + ' </w>'] += 1:**
- **list(word):** Converts the word into a list of characters.
- **' '.join(list(word)): Joins the characters with spaces.**
- **+ ' </w>': Adds an end token </w> to the end of the word.**
- The resulting string is used as a key in the vocab dictionary, and its value (frequency count) is incremented by 1.

In [1]:
```
from collections import defaultdict
from typing import List, Dict

def get_vocab(data: List[str]) -> Dict[str, int]:
    """
    Given a list of strings, returns a dictionary of words mapping to their
    count in the data.

    Parameters:
    data (List[str]): A list of strings where each string is a line of text.

    Returns:
    Dict[str, int]: A dictionary where keys are words with separated charact
                    an end token, and values are their frequency counts.
    """
    vocab = defaultdict(int)
    for line in data:
        for word in line.split():
            # Join the characters of the word with spaces and add an end tok
            vocab[' '.join(list(word)) + ' </w>'] += 1
    return vocab
```

In [2]:
```
# Example usage
data = [
    "this is a test",
    "this test is only a test",
    "a test this is"
]

vocab = get_vocab(data)
print(vocab)
```

```
defaultdict(<class 'int'>, {'t h i s </w>': 3, 'i s </w>': 3, 'a </w>': 3,
't e s t </w>': 4, 'o n l y </w>': 1})
```

STEP 2: Define function get_stats()

Note on Creating the Pairs Dictionary:

- **pairs = defaultdict(int):** Initializes a defaultdict with int as the default factory, meaning any new key will have a default value of 0.
- The function iterates through each word and its frequency in vocab.
- **symbols = word.split():** Splits the word into its component symbols (characters and end token).
- The nested loop iterates through adjacent symbol pairs in the list and increments their frequency count in the pairs dictionary.

In [3]:
```
from collections import defaultdict
from typing import Dict, Tuple

def get_stats(vocab: Dict[str, int]) -> Dict[Tuple[str, str], int]:
    """
    Given a vocabulary (dictionary mapping words to frequency counts), retu
    dictionary of tuples representing the frequency count of pairs of charac
    in the vocabulary.

    Parameters:
    vocab (Dict[str, int]): A dictionary where keys are words with separated
                            and an end token, and values are their frequency

    Returns:
    Dict[Tuple[str, str], int]: A dictionary where keys are tuples of charac
                                and values are their frequency counts in the
    """
    pairs = 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
```

In [4]:
```
# Example usage
vocab = {
    't h i s </w>': 3,
    'i s </w>': 2,
    'a </w>': 2,
    't e s t </w>': 2
}

stats = get_stats(vocab)
print(stats)
```

```
defaultdict(<class 'int'>, {('t', 'h'): 3, ('h', 'i'): 3, ('i', 's'): 5,
('s', '</w>'): 5, ('a', '</w>'): 2, ('t', 'e'): 2, ('e', 's'): 2, ('s',
't'): 2, ('t', '</w>'): 2})
```

STEP 3: Define function merge_vocab()

Notes on Creating the New Vocabulary Dictionary

- v_out = {}: Initializes an empty dictionary for the new vocabulary.

- bigram = re.escape(' '.join(pair)): Joins the pair with a space and escapes any special characters for use in a regular expression.
- p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)'): Compiles a regular expression pattern that matches the bigram as a whole word (using negative lookbehind (?<!\S) and negative lookahead (?!\S) to ensure it is not part of another word).
- The function iterates through each word in v_in.
  - w_out = p.sub(''.join(pair), word): Replaces occurrences of the bigram in the word with the merged pair.
  - v_out[w_out] = v_in[word]: Adds the modified word and its frequency to the new vocabulary dictionary.

STEP 3a). Creating Bigrams

Detailed explanation of:

```
bigram = re.escape(' '.join(pair)):
```

Objective: This line constructs a regular expression pattern for a bigram (pair of characters) and escapes any special characters.

Steps:

- pair is assumed to be a tuple containing two strings, representing the bigram.
- ' '.join(pair) combines the two strings with a space in between.
- re.escape() escapes any special characters in the resulting string to ensure they are treated as literal characters in the regex

In [5]:
```
### Example

import re
pair = ('a', 'b')
bigram = re.escape(' '.join(pair))
# ' '.join(pair) results in 'a b'
# re.escape('a b') results in 'a\\ b' (the space is escaped)
print(bigram)  # Output: 'a\\ b'
```

```
a\ b
```

STEP 3b) Compiling the Regex pattern

Detailed explanation of:

```
p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)'):
```

Objective: This line creates a compiled regular expression object p that matches the bigram when it appears as a whole word, not part of another word.

Components:

- r'(?<!\S)': A negative lookbehind assertion. \S matches any non-whitespace character. (?<!\S) asserts that what immediately precedes the current position is not a non-whitespace character (i.e., it ensures the bigram is not part of a larger word).
- bigram: The escaped bigram pattern from the previous step.
- r'(?!\S)': A negative lookahead assertion. \S matches any non-whitespace character. (?!\S) asserts that what immediately follows the current position is not a non-whitespace character (i.e., it ensures the bigram is not part of a larger word).

In [6]:
```
# Example

import re
pair = ('a', 'b')

bigram = re.escape(' '.join(pair))

p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')

text = "a b a b a b a bc"
matches = p.findall(text)
# Matches 'a b' only when it appears as a whole word, not part of 'a bc'
print(matches)  # Output: ['a b', 'a b', 'a b']
```

```
['a b', 'a b', 'a b']
```

STEP 3c Create output Dict

Detailed Explanation of below code snippet

```
 for word in v_in:
     w_out = p.sub(''.join(pair), word)
     v_out[w_out] = v_in[word]
```

Objective: The code snippet merges the character pair if the pair is find in any word string

Steps:

- Loop through the words
- for each word search for the character pair
- If a match is found substute the space separated pair by the joined pair (no spaces) add an new entry to the output dictionary (new key) with value equal to the frequency of the old dictionary before the substitution

STEP 3d) Let us code the merge_vocab function now!

In [7]:
```
import re
from typing import Tuple, Dict

def merge_vocab(pair: Tuple[str, str], v_in: Dict[str, int]) -> Dict[str, in
    """
    Given a pair of characters and a vocabulary, returns a new vocabulary wi
    pair of characters merged together wherever they appear.

    Parameters:
    pair (Tuple[str, str]): A tuple containing two characters to be merged.
    v_in (Dict[str, int]): A dictionary where keys are words with separated
                           and an end token, and values are their frequency

    Returns:
    Dict[str, int]: A new vocabulary dictionary with the pair of characters
    """
    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
```

In [8]:
```
# Example usage
v_in = {
    't h i s </w>': 3,
    'i s </w>': 2,
    'a </w>': 2,
    't e s t </w>': 2
}

pair = ('t', 'h')
merged_vocab = merge_vocab(pair, v_in)
print(merged_vocab)
```

```
{'th i s </w>': 3, 'i s </w>': 2, 'a </w>': 2, 't e s t </w>': 2}
```

STEP 4 : Create the Byte Pair Encoder Function

Putting it all together

Input Parameters

- A list of strings
- an integer n denoting how many merged pairs are to be returned

Output Parameter(s)

- A dictionary with the vocabulary post Byte Pair Encoding

Key Process Steps

- vocab = get_vocab(data): Initializes the vocabulary using the get_vocab function.
- loop through 'n' times, at each iteration:
    - determine frequency dict of character pairs
    - extract the most frequent character pair
    - merge the most frequent pair in the vocab list

In [9]:
```
from typing import List, Dict

def byte_pair_encoder(data: List[str], n: int) -> Dict[str, int]:
    """
    Given a list of strings and an integer n, returns a list of n merged pa
    of characters found in the vocabulary of the input data.
    Parameters:
    data (List[str]): A list of strings where each string is a line of text.
    n (int): The number of pairs of characters to merge.

    Returns:
    Dict[str, int]: A dictionary representing the vocabulary with merged cha
    """
    vocab = get_vocab(data)
    for i in range(n):
        pairs = get_stats(vocab)
        best = max(pairs, key=pairs.get)
        vocab = merge_vocab(best, vocab)
    return vocab
```

In [10]:
```
# Example usage:

# Set corpus

corpus = '''Tokenization is the process of breaking down
a sequence of text into smaller units called tokens,
which can be words, phrases, or even individual characters.
Tokenization is often the first step in natural language processing tasks
such as text classification, named entity recognition, and sentiment analysi
The resulting tokens are typically used as input to further processing steps
such as vectorization, where the tokens are converted
into numerical representations for machine learning models to use.'''\


# split by sentence
data = corpus.split('.')
```

TEST BPE with n = 200

In [11]:
```
# define output count
n = 200

# call function
bpe_pairs = byte_pair_encoder(data, n)

# check
print(bpe_pairs)
```

```
{'Tokenization</w>': 2, 'is</w>': 2, 'the</w>': 3, 'process</w>': 1, 'of</w
>': 2, 'breaking</w>': 1, 'down</w>': 1, 'a</w>': 1, 'sequence</w>': 1, 'te
xt</w>': 2, 'into</w>': 2, 'smaller</w>': 1, 'units</w>': 1, 'called</w>':
1, 'tokens,</w>': 1, 'which</w>': 1, 'can</w>': 1, 'be</w>': 1, 'words,</w
>': 1, 'phrases,</w>': 1, 'or</w>': 1, 'even</w>': 1, 'individual</w>': 1,
'characters</w>': 1, 'often</w>': 1, 'first</w>': 1, 'step</w>': 1, 'in</w
>': 1, 'natural</w>': 1, 'language</w>': 1, 'processing</w>': 2, 'tasks</w
>': 1, 'such</w>': 2, 'as</w>': 3, 'classification,</w>': 1, 'named</w>':
1, 'entity</w>': 1, 'recognition,</w>': 1, 'and</w>': 1, 'sentiment</w>':
1, 'analysis</w>': 1, 'The</w>': 1, 'resulting</w>': 1, 'tokens</w>': 2, 'a
re</w>': 2, 'typically</w>': 1, 'used</w>': 1, 'input</w>': 1, 'to</w>': 2,
'further</w>': 1, 'steps,</w>': 1, 'vectorization,</w>': 1, 'where</w>': 1,
'conv er te d</w>': 1, 'n u m er ic al</w>': 1, 're pr es en t ation s</w
>': 1, 'f or</w>': 1, 'm a ch in e</w>': 1, 'l e ar n ing</w>': 1, 'm o d e
l s</w>': 1, 'us e</w>': 1}
```

TEST BPE with n = 210

In [12]:
```
# define output count
n = 210
```

```
# call function
bpe_pairs = byte_pair_encoder(data, n)

# check
print(bpe_pairs)
```

```
{'Tokenization</w>': 2, 'is</w>': 2, 'the</w>': 3, 'process</w>': 1, 'of</w
>': 2, 'breaking</w>': 1, 'down</w>': 1, 'a</w>': 1, 'sequence</w>': 1, 'te
xt</w>': 2, 'into</w>': 2, 'smaller</w>': 1, 'units</w>': 1, 'called</w>':
1, 'tokens,</w>': 1, 'which</w>': 1, 'can</w>': 1, 'be</w>': 1, 'words,</w
>': 1, 'phrases,</w>': 1, 'or</w>': 1, 'even</w>': 1, 'individual</w>': 1,
'characters</w>': 1, 'often</w>': 1, 'first</w>': 1, 'step</w>': 1, 'in</w
>': 1, 'natural</w>': 1, 'language</w>': 1, 'processing</w>': 2, 'tasks</w
>': 1, 'such</w>': 2, 'as</w>': 3, 'classification,</w>': 1, 'named</w>':
1, 'entity</w>': 1, 'recognition,</w>': 1, 'and</w>': 1, 'sentiment</w>':
1, 'analysis</w>': 1, 'The</w>': 1, 'resulting</w>': 1, 'tokens</w>': 2, 'a
re</w>': 2, 'typically</w>': 1, 'used</w>': 1, 'input</w>': 1, 'to</w>': 2,
'further</w>': 1, 'steps,</w>': 1, 'vectorization,</w>': 1, 'where</w>': 1,
'converted</w>': 1, 'numerical</w>': 1, 'repres en t ation s</w>': 1, 'f or
</w>': 1, 'm a ch in e</w>': 1, 'l e ar n ing</w>': 1, 'm o d e l s</w>':
1, 'us e</w>': 1}
```

Section B: Using Byte Pair Encoder from tiktoken

Note

We have seen how complex it is to implement BPE from ground up! and its also operationally expensive in compute sense.

In this section we will use an existing Python open-source library called tiktoken (https://github.com/openai/tiktoken), which implements the BPE algorithm very efficiently based on source code in **Rust**.

installs

In [13]:
```
#!pip install tiktoken
import tiktoken
```

Instantiate the BPE tokenizer from tiktoken

In [14]:
```
tokenizer = tiktoken.get_encoding("gpt2")
```

Check usage

Key Steps

- tokenize an input text to token ids and check
- convert token ids back to tokens and check

encode

In [15]:
```
# define text
text = (
    "Hello, do you want some coffee? <|endoftext|> In the shadows of large 
     "of someunknownPlace."
)

# tokenize
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})

# check
print(integers)
```

```
[15496, 11, 466, 345, 765, 617, 6891, 30, 220, 50256, 554, 262, 16187, 286,
1588, 18057, 7150, 1659, 617, 34680, 27271, 13]
```

decode

In [16]:
```
strings = tokenizer.decode(integers)

print(strings)
```

```
Hello, do you want some coffee? <|endoftext|> In the shadows of large palm
treesof someunknownPlace.
```

In [ ]:
```
```

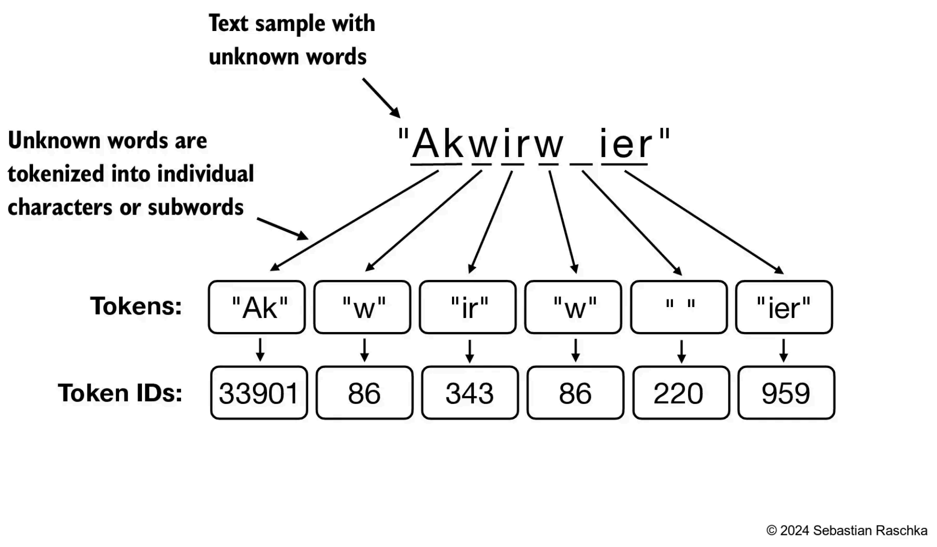
Observations from usage

- **First** The <|endoftext|> token is assigned a relatively large token ID, namely, 50256. In fact, the BPE tokenizer, which was used to train models such as GPT-2, GPT-3, and the original model used in ChatGPT, has a total vocabulary size of 50,257, with <|endoftext|> being assigned the largest token ID.
- **Second** The BPE tokenizer above encodes and decodes unknown words, such as "someunknownPlace" correctly. The BPE tokenizer can handle any unknown word. **How does it achieve this without using <|unk|> tokens?**

The Trick is as below

The algorithm underlying BPE breaks down words that aren't in its predefined vocabulary into smaller subword units or even individual characters, enabling it to handle out-of-vocabulary words. So, thanks to the BPE algorithm, if the tokenizer encounters an unfamiliar word during tokenization, it can represent it as a sequence of subword tokens or characters, as illustrated in Figure below.

Fig reference - Ch2 Reference text


Text sample with unknown words
"Akwirw ier"
Unknown words are tokenized into individual characters or subwords
Tokens:
"Ak"
"w"
"ir"
"w"
" "
"ier"
Token IDs:
33901
86
343
86
220
959
© 2024 Sebastian Raschka

Let us test the above

In [17]:
```
text = " Akwirw ier"

# tokenize
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})

# check
print(integers)


strings = tokenizer.decode(integers)

print(strings)
```

```
[9084, 86, 343, 86, 220, 959]
 Akwirw ier
```

Food for thought: - How does it recombine unknown words post byte pair encoding!!!

(guess we need to ask that to some one in open AI!!!)

End of notebook

In [ ]: