Large Language Models: Tokenization

Nathanaël Fijalkow
CNRS, LaBRI, Bordeaux

Tokenization

- Basics of encoding
- Pre-tokenization
- Byte-Pair Encoding (BPE)
- WordPiece

Credits

Images and contents from Chapter 6 of Hugging Face's course on NLP:

https://huggingface.co/learn/nlp-course/chapter6

Basics

A **bit** = 0 or 1

A **byte** = typically an octet, meaning 8 bits

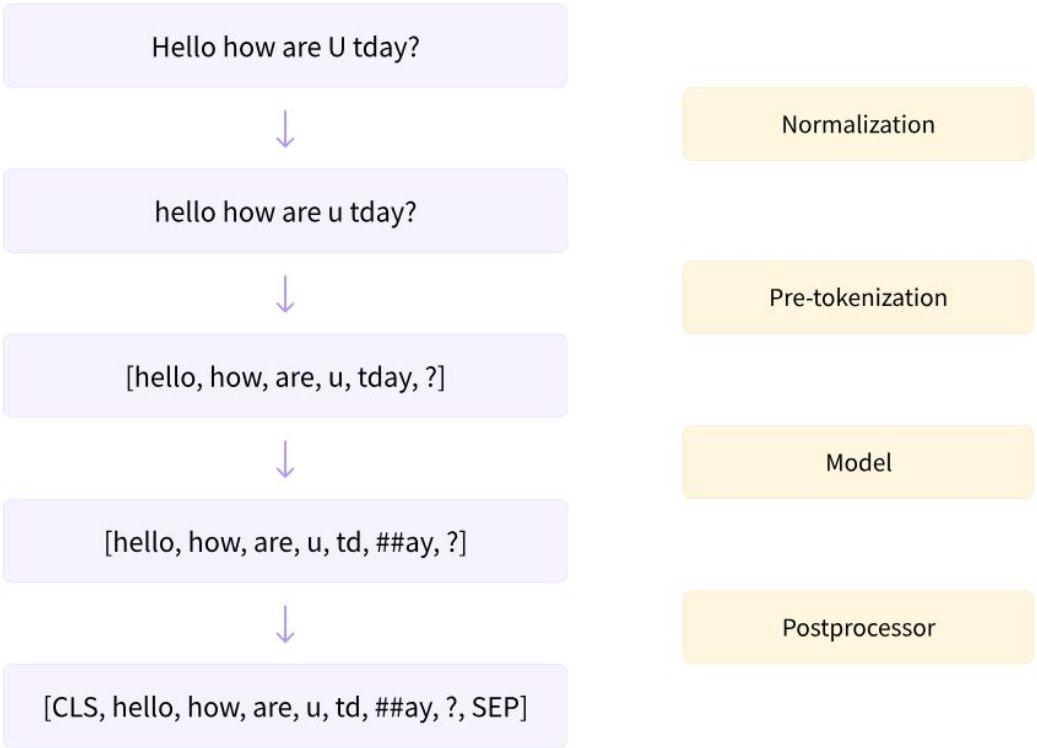
Character encodings:

- ASCII (code unit: 7 bits)
- Unicode: UTF-8, UTF-16, UTF-32 (code unit: 8,16,32 bits)

98% of WWW is UTF-8. Technically UTF is variable-length (so infinite…)

We are only considering “subword tokenization algorithms” but there are other tokenization algorithms…

The full tokenization pipeline



Normalization

The normalization step involves some general cleanup, such as removing needless whitespace, lowercasing, and/or removing accents.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
print(type(tokenizer.backend_tokenizer))
```

```
<class 'tokenizers.Tokenizer'>
```

```
print(tokenizer.backend_tokenizer.normalizer.normalize_str("Héllò hôw are ü?"))
```

```
hello how are u?
```

Pre-tokenization

Breaks a text into words (keeping the offsets):

```
tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str("Hello, how are  you?")
```

```
[('Hello', (0, 5)),
 (',', (5, 6)),
 ('how', (7, 10)),
 ('are', (11, 14)),
 ('you', (16, 19)),
 ('?', (19, 20))]
```

Pre-tokenization

Again there are many variants…

SentencePiece is a simple pre-tokenization algorithm:

- Treats everything as Unicode characters
- Replaces spaces with “_”

Tokenization algorithms

Two components:

- The *training* algorithm: preprocessing on a training set, to determine what will be the tokens

- The *tokenization* algorithm: at run time, transforming text inputs into sequences of tokens

Byte-Pair Encoding

Developed by OpenAI for GPT-2

Pre-tokenization adds “Ġ” before each word except the first:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str("Hello, how are  you?")
```

```
[('Hello', (0, 5)),
 (',', (5, 6)),
 ('Ġhow', (6, 10)),
 ('Ġare', (10, 14)),
 ('Ġ', (14, 15)),
 ('Ġyou', (15, 19)),
 ('?', (19, 20))]
```

BPE IN ONE SLIDE

The goal is to learn merge rules, of the form:

(“Amer”, “ica”) -> “America”

Training: starting from characters, we create rules by merging the most frequent pairs, until we reach the budget number of tokens

Processing: to process an input text we apply rules greedily

Example corpus

```
corpus = [
    "This is the Hugging Face Course.",
    "This chapter is about tokenization.",
    "This section shows several tokenizer algorithms.",
    "Hopefully, you will be able to understand how they are trained and generate tokens.",
]
```

BPE training algorithm, step 0: Compute frequencies

```python
from collections import defaultdict

word_freqs = defaultdict(int)

for text in corpus:
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    new_words = [word for word, offset in words_with_offsets]
    for word in new_words:
        word_freqs[word] += 1

print(word_freqs)
```

```
defaultdict(<class 'int'>, {'This': 3, 'Ġis': 2, 'Ġthe': 1, 'ĠHugging': 1, 'ĠFace': 1, 'ĠCourse': 1, '.': 4, 'Ġcha
pter': 1, 'Ġabout': 1, 'Ġtokenization': 1, 'Ġsection': 1, 'Ġshows': 1, 'Ġseveral': 1, 'Ġtokenizer': 1, 'Ġalgorithm
s': 1, 'Hopefully': 1, ',': 1, 'Ġyou': 1, 'Ġwill': 1, 'Ġbe': 1, 'Ġable': 1, 'Ġto': 1, 'Ġunderstand': 1, 'Ġhow': 1,
'Ġthey': 1, 'Ġare': 1, 'Ġtrained': 1, 'Ġand': 1, 'Ġgenerate': 1, 'Ġtokens': 1})
```

BPE training algorithm, step 1: collect characters

```
alphabet = []

for word in word_freqs.keys():
    for letter in word:
        if letter not in alphabet:
            alphabet.append(letter)
alphabet.sort()

print(alphabet)
```

```
[',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'r',
's', 't', 'u', 'v', 'w', 'y', 'z', 'Ġ']
```

```
vocab = ["<|endoftext|>"] + alphabet.copy()
```

“<|endoftext|>” is a special token

BPE training algorithm, step 2: Compute pair frequencies

```
splits = {word: [c for c in word] for word in word_freqs.keys()}
```

```
def compute_pair_freqs(splits):
    pair_freqs = defaultdict(int)
    for word, freq in word_freqs.items():
        split = splits[word]
        if len(split) == 1:
            continue
        for i in range(len(split) - 1):
            pair = (split[i], split[i + 1])
            pair_freqs[pair] += freq
    return pair_freqs
```

```
pair_freqs = compute_pair_freqs(splits)

for i, key in enumerate(pair_freqs.keys()):
    print(f"{key}: {pair_freqs[key]}")
    if i >= 5:
        break
```

```
('T', 'h'): 3
('h', 'i'): 3
('i', 's'): 5
('Ġ', 'i'): 2
('Ġ', 't'): 7
('t', 'h'): 3
```

```
best_pair = ""
max_freq = None

for pair, freq in pair_freqs.items():
    if max_freq is None or max_freq < freq:
        best_pair = pair
        max_freq = freq

print(best_pair, max_freq)
```

```
('Ġ', 't') 7
```

BPE training algorithm, step 3: Add a merge rule

```
merges = {("Ġ", "t"): "Ġt"}
vocab.append("Ġt")
```

```
def merge_pair(a, b, splits):
    for word in word_freqs:
        split = splits[word]
        if len(split) == 1:
            continue

        i = 0
        while i < len(split) - 1:
            if split[i] == a and split[i + 1] == b:
                split = split[:i] + [a + b] + split[i + 2 :]
            else:
                i += 1
        splits[word] = split
    return splits
```

```
splits = merge_pair("Ġ", "t", splits)
print(splits["Ġtrained"])
```

```
['Ġt', 'r', 'a', 'i', 'n', 'e', 'd']
```

BPE training algorithm: the loop

```
vocab_size = 50

while len(vocab) < vocab_size:
    pair_freqs = compute_pair_freqs(splits)
    best_pair = ""
    max_freq = None
    for pair, freq in pair_freqs.items():
        if max_freq is None or max_freq < freq:
            best_pair = pair
            max_freq = freq
    splits = merge_pair(*best_pair, splits)
    merges[best_pair] = best_pair[0] + best_pair[1]
    vocab.append(best_pair[0] + best_pair[1])
```

```
print(merges)
```

```
{('Ġ', 't'): 'Ġt', ('i', 's'): 'is', ('e', 'r'): 'er', ('Ġ', 'a'): 'Ġa', ('Ġt', 'o'): 'Ġto', ('e', 'n'): 'en',
('T', 'h'): 'Th', ('Th', 'is'): 'This', ('o', 'u'): 'ou', ('s', 'e'): 'se', ('Ġto', 'k'): 'Ġtok', ('Ġtok', 'en'):
'Ġtoken', ('n', 'd'): 'nd', ('Ġ', 'is'): 'Ġis', ('Ġt', 'h'): 'Ġth', ('Ġth', 'e'): 'Ġthe', ('i', 'n'): 'in', ('Ġa',
'b'): 'Ġab', ('Ġtoken', 'i'): 'Ġtokeni'}
```

```
print(vocab)
```

```
['<|endoftext|>', ',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n',
'o', 'p', 'r', 's', 't', 'u', 'v', 'w', 'y', 'z', 'Ġ', 'Ġt', 'is', 'er', 'Ġa', 'Ġto', 'en', 'Th', 'This', 'ou', 's
e', 'Ġtok', 'Ġtoken', 'nd', 'Ġis', 'Ġth', 'Ġthe', 'in', 'Ġab', 'Ġtokeni']
```

BPE Tokenization algorithm

```
def tokenize(text):
    pre_tokenize_result = tokenizer._tokenizer.pre_tokenizer.pre_tokenize_str(text)
    pre_tokenized_text = [word for word, offset in pre_tokenize_result]
    splits = [[l for l in word] for word in pre_tokenized_text]
    for pair, merge in merges.items():
        for idx, split in enumerate(splits):
            i = 0
            while i < len(split) - 1:
                if split[i] == pair[0] and split[i + 1] == pair[1]:
                    split = split[:i] + [merge] + split[i + 2 :]
                else:
                    i += 1
            splits[idx] = split

    return sum(splits, [])
```

```
tokenize("This is not a token.")
```

```
['This', 'Ġis', 'Ġ', 'n', 'o', 't', 'Ġa', 'Ġtoken', '.']
```

BPE Tokenization algorithm can fail?

What happens if there’s an unknown character? This code would fail…

In actual (byte-level) implementations, it cannot happen.

In practice

Tiktoken implements BPE:

https://github.com/openai/tiktoken

Wordpiece

Developed by Google for BERT (but never open sourced!)

The pre-tokenizer feels a lot more civilized:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str("Hello, how are you?")
```

```
[('Hello', (0, 5)),
 (',', (5, 6)),
 ('how', (7, 10)),
 ('are', (11, 14)),
 ('you', (15, 18)),
 ('?', (18, 19))]
```

Wordpiece in one slide

The goal is to learn merge rules, of the form:

(“Amer”, “ica”) -> “America”

Training: starting from characters, we create tokens by merging pairs with highest score, until we reach the budget number of tokens

Processing: to process an input text we look for the longest token and continue recursively (not using rules!)

WordPiece training algorithm, step 0: compute characters

```python
from collections import defaultdict

word_freqs = defaultdict(int)
for text in corpus:
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    new_words = [word for word, offset in words_with_offsets]
    for word in new_words:
        word_freqs[word] += 1

word_freqs
```

```
defaultdict(int,
            {'This': 3,
             'is': 2,
             'the': 1,
             'Hugging': 1,
             'Face': 1,
             'Course': 1,
             '.': 4,
             'chapter': 1,
             'about': 1,
             'tokenization': 1,
             'section': 1,
             'shows': 1,
             'several': 1,
             'tokenizer': 1,
             'algorithms': 1,
             'Hopefully': 1,
```

WordPiece training algorithm, step 1: compute frequencies

```
alphabet = []
for word in word_freqs.keys():
    if word[0] not in alphabet:
        alphabet.append(word[0])
    for letter in word[1:]:
        if f"##{letter}" not in alphabet:
            alphabet.append(f"##{letter}")

alphabet.sort()
alphabet

print(alphabet)
```

```
['##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '##i', '##k', '##l', '##m', '##n', '##o', '##p', '##r',
'##s', '##t', '##u', '##v', '##w', '##y', '##z', ',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'g', 'h', 'i', 's',
't', 'u', 'w', 'y']
```

```
vocab = ["[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]"] + alphabet.copy()
```

```
splits = {
    word: [c if i == 0 else f"##{c}" for i, c in enumerate(word)]
    for word in word_freqs.keys()
}
splits
```

```
{'This': ['T', '##h', '##i', '##s'],
 'is': ['i', '##s'],
 'the': ['t', '##h', '##e'],
 'Hugging': ['H', '##u', '##g', '##g', '##i', '##n', '##g'],
 'Face': ['F', '##a', '##c', '##e'],
 'Course': ['C', '##o', '##u', '##r', '##s', '##e'],
```

WordPiece training algorithm, step 2: compute scores

WordPiece computes a score for each pair, using the following formula:

$$\text{freq\_of\_pair} / (\text{freq\_of\_first\_element} \times \text{freq\_of\_second\_element})$$

The algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary:

- It won't necessarily merge `("un", "##able")` even if that pair occurs very frequently in the vocabulary, because the two pairs `"un"` and `"##able"` will likely each appear in a lot of other words and have a high frequency.
- In contrast, a pair like `("hu", "##gging")` will probably be merged faster (assuming the word "hugging" appears often in the vocabulary) since `"hu"` and `"##gging"` are likely to be less frequent individually.

Wordpiece training algorithm, step 2: compute scores

```
def compute_pair_scores(splits):
    letter_freqs = defaultdict(int)
    pair_freqs = defaultdict(int)
    for word, freq in word_freqs.items():
        split = splits[word]
        if len(split) == 1:
            letter_freqs[split[0]] += freq
            continue
        for i in range(len(split) - 1):
            pair = (split[i], split[i + 1])
            letter_freqs[split[i]] += freq
            pair_freqs[pair] += freq
        letter_freqs[split[-1]] += freq

    scores = {
        pair: freq / (letter_freqs[pair[0]] * letter_freqs[pair[1]])
        for pair, freq in pair_freqs.items()
    }
    return scores
```

```
best_pair = ""
max_score = None
for pair, score in pair_scores.items():
    if max_score is None or max_score < score:
        best_pair = pair
        max_score = score

print(best_pair, max_score)
```

```
('a', '##b') 0.2
```

```
vocab.append("ab")
```

```
def merge_pair(a, b, splits):
    for word in word_freqs:
        split = splits[word]
        if len(split) == 1:
            continue
        i = 0
        while i < len(split) - 1:
            if split[i] == a and split[i + 1] == b:
                merge = a + b[2:] if b.startswith("##") else a + b
                split = split[:i] + [merge] + split[i + 2 :]
            else:
                i += 1
        splits[word] = split
    return splits
```

```
splits = merge_pair("a", "##b", splits)
splits["about"]
```

```
['ab', '##o', '##u', '##t']
```

WordPiece training algorithm: the Loop

```
vocab_size = 70
while len(vocab) < vocab_size:
    scores = compute_pair_scores(splits)
    best_pair, max_score = "", None
    for pair, score in scores.items():
        if max_score is None or max_score < score:
            best_pair = pair
            max_score = score
    splits = merge_pair(*best_pair, splits)
    new_token = (
        best_pair[0] + best_pair[1][2:]
        if best_pair[1].startswith("##")
        else best_pair[0] + best_pair[1]
    )
    vocab.append(new_token)
```

```
print(vocab)
```

```
['[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]', '##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '##i', '##
k', '##l', '##m', '##n', '##o', '##p', '##r', '##s', '##t', '##u', '##v', '##w', '##y', '##z', ',', '.', 'C', 'F',
'H', 'T', 'a', 'b', 'c', 'g', 'h', 'i', 's', 't', 'u', 'w', 'y', 'ab', '##fu', 'Fa', 'Fac', '##ct', '##ful', '##fu
ll', '##fully', 'Th', 'ch', '##hm', 'cha', 'chap', 'chapt', '##thm', 'Hu', 'Hug', 'Hugg', 'sh', 'th', 'is', '##thm
s', '##za', '##zat', '##ut']
```

Wordpiece tokenization algorithm

```
def encode_word(word):
    tokens = []
    while len(word) > 0:
        i = len(word)
        while i > 0 and word[:i] not in vocab:
            i -= 1
        if i == 0:
            return ["[UNK]"]
        tokens.append(word[:i])
        word = word[i:]
        if len(word) > 0:
            word = f"##{word}"
    return tokens
```

```
print(encode_word("Hugging"))
print(encode_word("HOgging"))
```

```
['Hugg', '##i', '##n', '##g']
['[UNK]']
```

```
def tokenize(text):
    pre_tokenize_result = tokenizer._tokenizer.pre_tokenizer.pre_tokenize_str(text)
    pre_tokenized_text = [word for word, offset in pre_tokenize_result]
    encoded_words = [encode_word(word) for word in pre_tokenized_text]
    return sum(encoded_words, [])
```

Summary for the two algorithms

Model	BPE	WordPiece
Training	Starts from a small vocabulary and learns rules to merge tokens	Starts from a small vocabulary and learns rules to merge tokens
Training step	Merges the tokens corresponding to the most common pair	Merges the tokens corresponding to the pair with the best score based on the frequency of the pair, privileging pairs where each individual token is less frequent
Learns	Merge rules and a vocabulary	Just a vocabulary
Encoding	Splits a word into characters and applies the merges learned during training	Finds the longest subword starting from the beginning that is in the vocabulary, then does the same for the rest of the word