

ChatGPT Overview & Tokenization

Lecture 2

EN.705.743: ChatGPT from Scratch

Notes from Office Hours

- Office hours was very uneventful this week
- Going forward I am only going to record / release the ones that actually have useful content
- Will also set up system to collect questions ahead of time via Canvas Discussions

One interesting question did come up:

Q: What is the difference between `Parameter()` and `Tensor(..., requires_grad=True)`?

A: Both of these will compute a gradient during backprop, however- **only the former will be automatically tracked by an optimizer.**

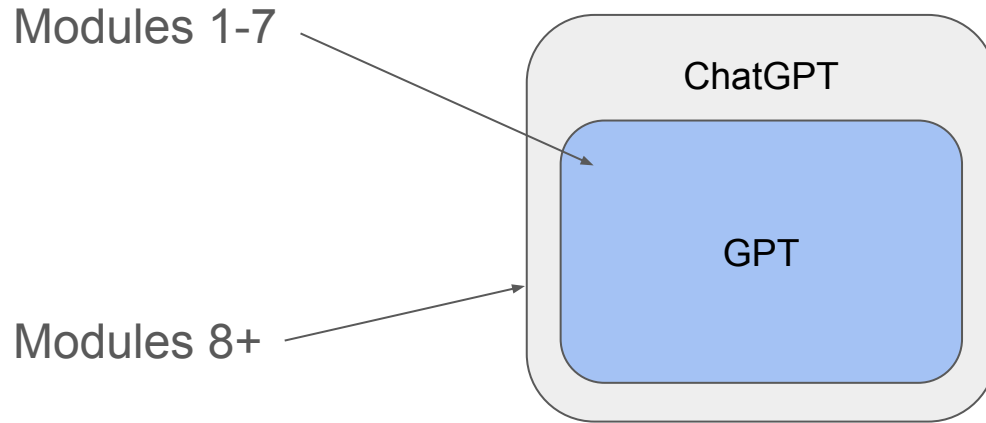
Lecture Outline

- ChatGPT, in more detail than last time.
 - Trace some data through the model (forward pass)
- What is Tokenization?
- Main Idea / Simple Approach (fixed vocabulary)
- Building a vocabulary
- Modern Tokenization: Byte-Pair Encoding
- Modern Tokenization: *Binary* Byte-Pair Encoding

GPT Model Data Trace

GPT vs ChatGPT

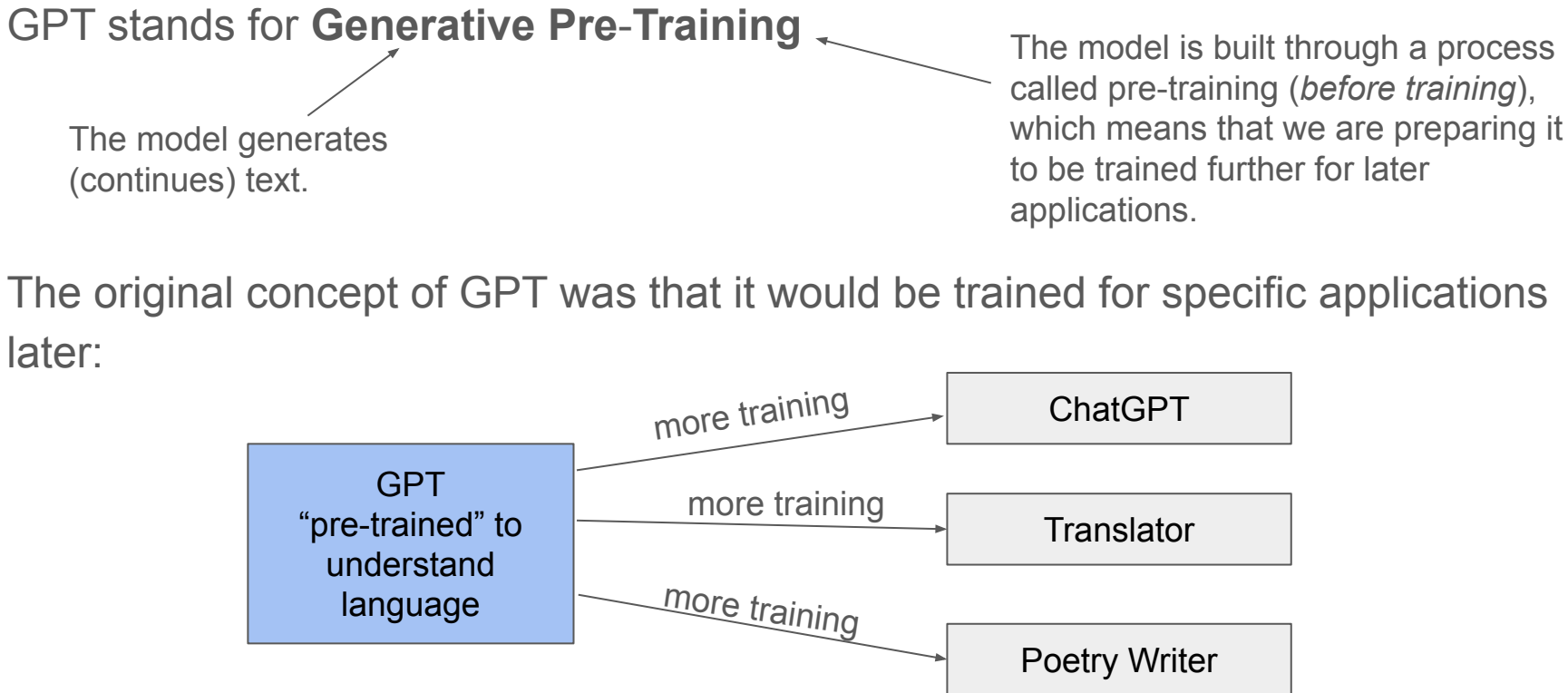
ChatGPT is a specialized model that is fine-tuned from a base model called GPT.



GPT

GPT stands for **Generative Pre-Training**

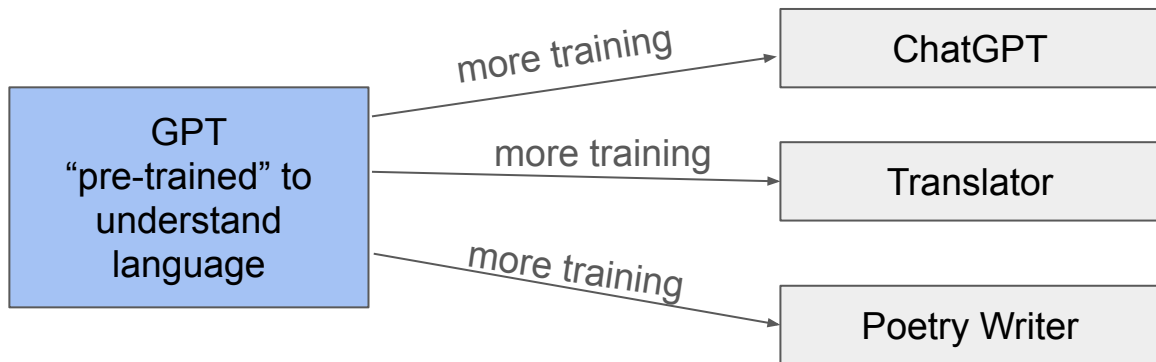
The model generates
(continues) text.



```
graph LR; A[GPT] --> B[The model generates (continues) text.]; A --> C[The model is built through a process called pre-training (before training), which means that we are preparing it to be trained further for later applications.]; A --> D[GPT "pre-trained" to understand language]; D -- "more training" --> E[ChatGPT]; D -- "more training" --> F[Translator]; D -- "more training" --> G[Poetry Writer];
```

The model is built through a process called pre-training (*before training*), which means that we are preparing it to be trained further for later applications.

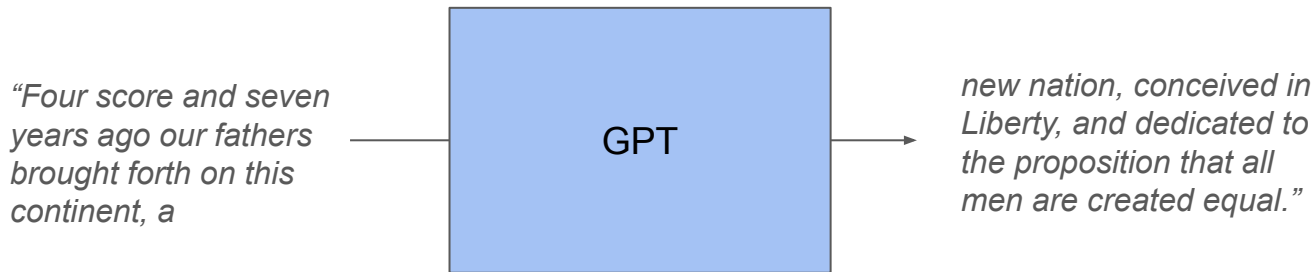
The original concept of GPT was that it would be trained for specific applications later:



A Common Base Problem

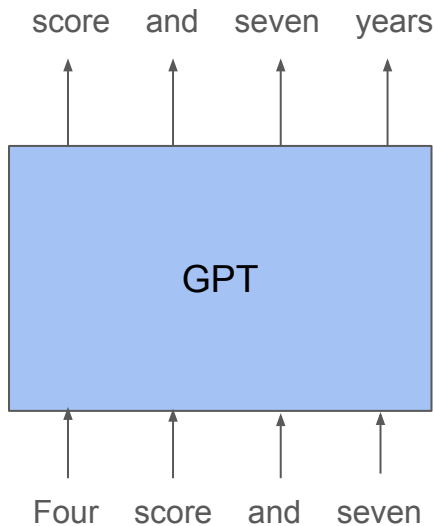
All the downstream applications have a common requirement that they take some text in, and generate other text that comes out.

So to prepare for all of these, we will make a model that just learns to continue text:



What does this model actually do?

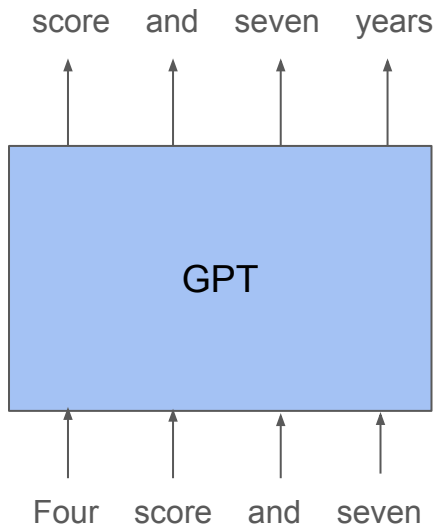
Specifically, a GPT model accepts a sequence of tokens, and outputs a sequence shifted by one position:



What does this model actually do?

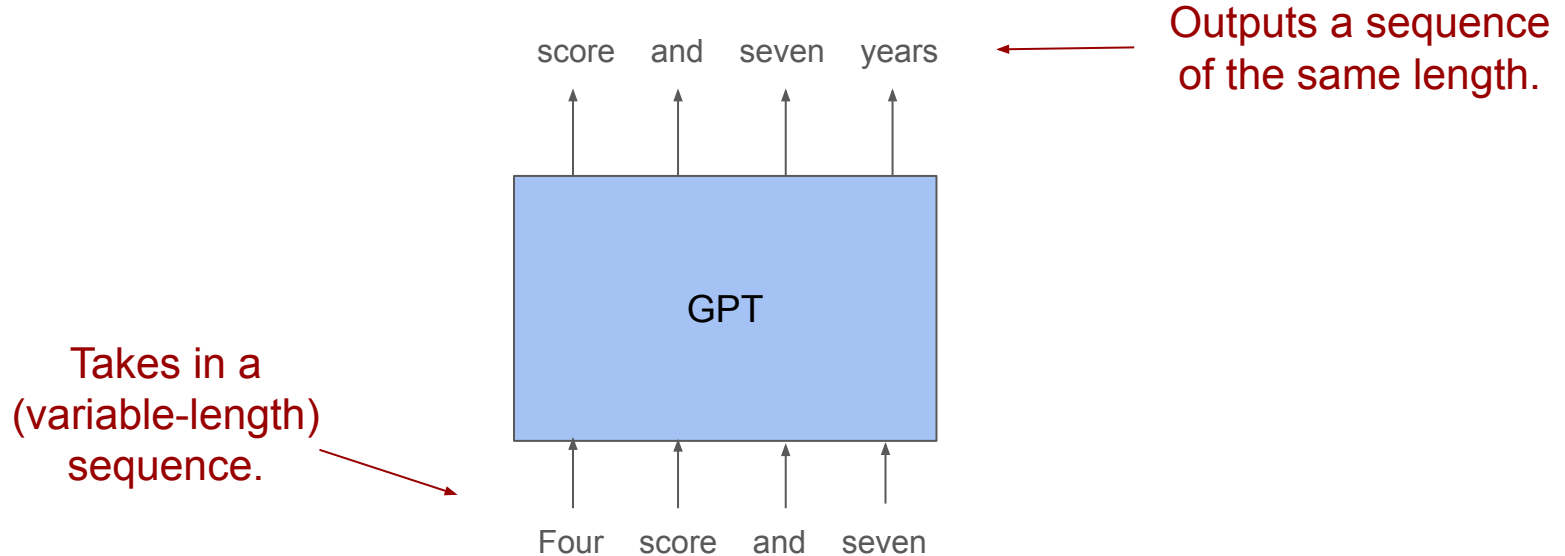
Specifically, a GPT model accepts a sequence of tokens, and outputs a sequence shifted by one position:

Takes in a
(variable-length)
sequence.



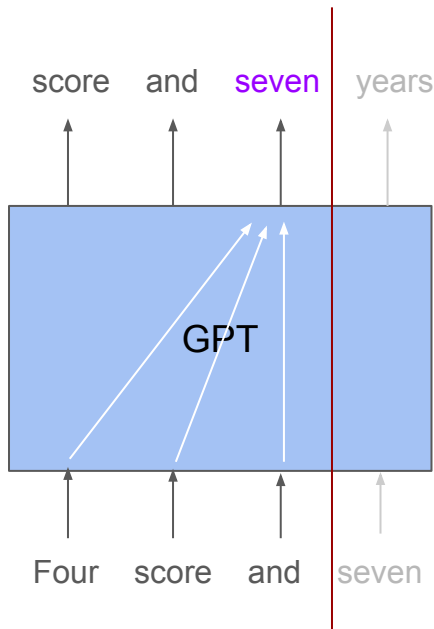
What does this model actually do?

Specifically, a GPT model accepts a sequence of tokens, and outputs a sequence shifted by one position:



What does this model actually do?

Specifically, a GPT model accepts a sequence of tokens, and outputs a sequence shifted by one position:



Each output is only
computed from prior
inputs.

Each output predicts:
what comes next in the
sequence?

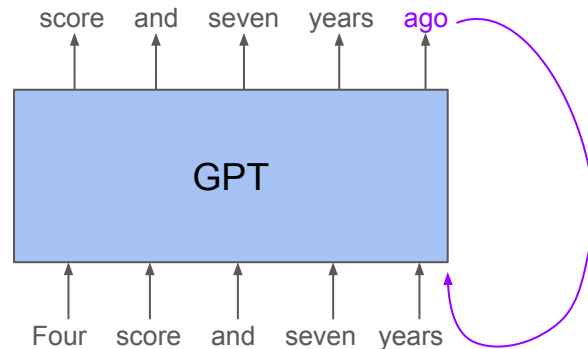
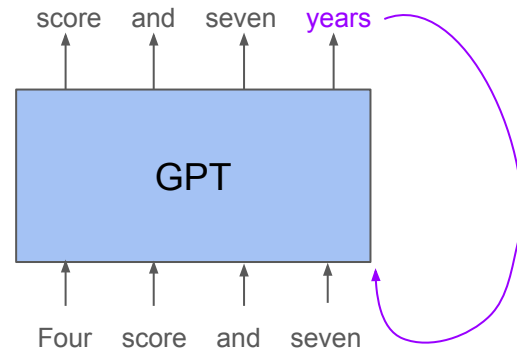
Autoregression

If we construct a model that does this, how would we use it?

You take the new token, and feed it back in as an input.

This is called an autoregressive model: Each output at step t is a function of all previous step 0 to $t-1$.

Ideally, a model trained in this way would be good enough at understanding text that with a little more training we could use it for specific language problems later.



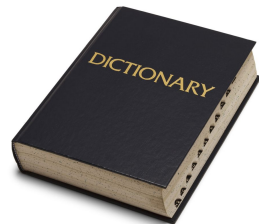
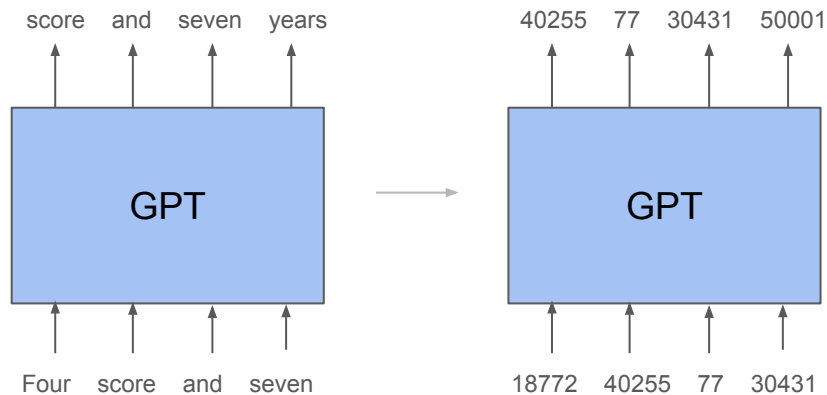
Words as Class ID's

How should we represent our text?

We can treat this as a really big classification problem!

Input: Id of each input word.

Output: Softmax'd prediction over all possible words.



=

0: aardvark

1: about

...

50256: zebra

50257: zero

Classification-Style Training

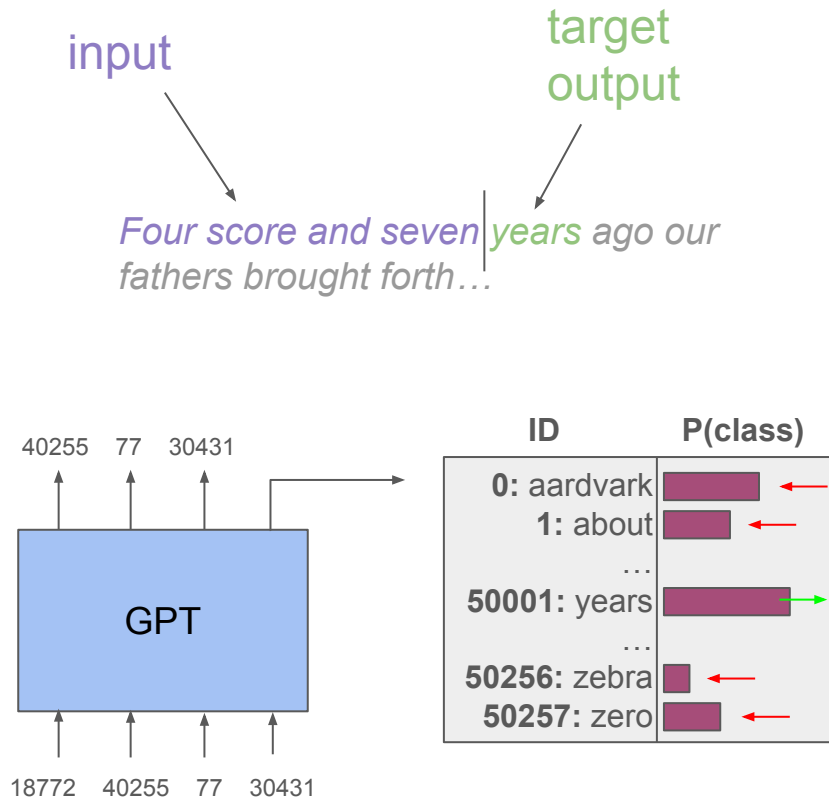
How do we train this model?

Given any snippet of text, we can imagine cutting it at some point.

We know (1) the preceding words, and (2) the next word.

This next word becomes our “label” for classification-style training.

We use standard cross-entropy loss to increase the probability of our ground-truth label.



Summary

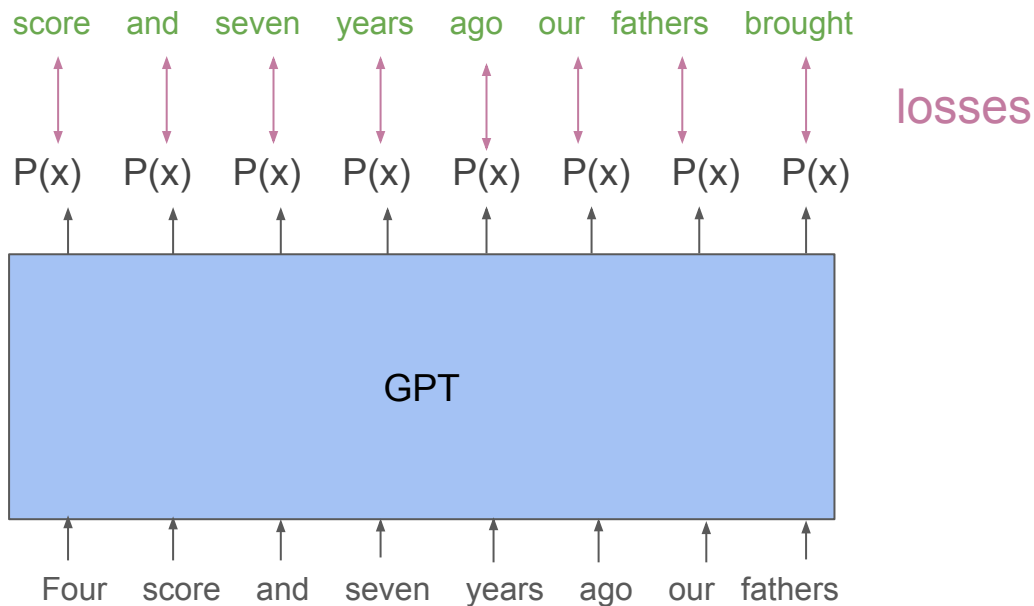
We take a bunch of text, and at any point in any sample we know (1) the proceeding text and (2) the next word.

We convert all our words into integer class id's according to some itemized list of words (our *vocabulary*).

We train the model just like a classifier! The output is a prediction over next words (classes), and since we know the true word that comes next, we can update the model to output that word with higher probability.

Important Note!

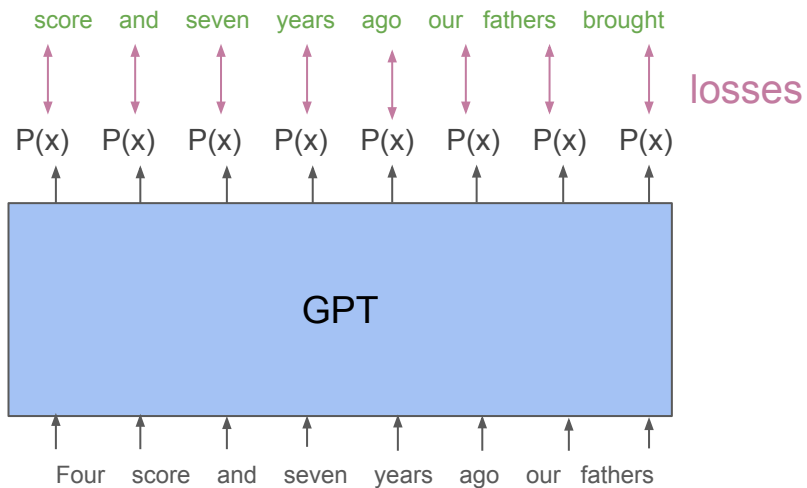
In reality, we do this training process in parallel and calculate a loss for an entire text sample at once. Every output has a known target output:



Important Note!

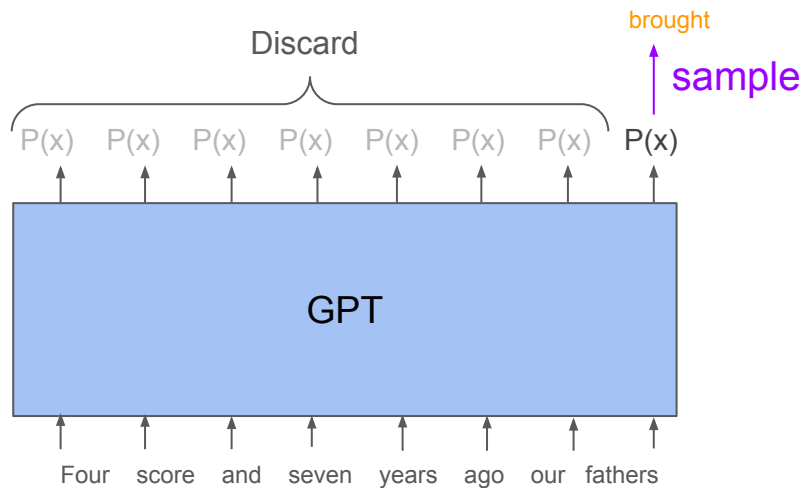
Training: Sequence to Sequence

Every output is a continuation of what came before.



Inference: Many to One (Effectively)

Only need the last output.



Tokenization

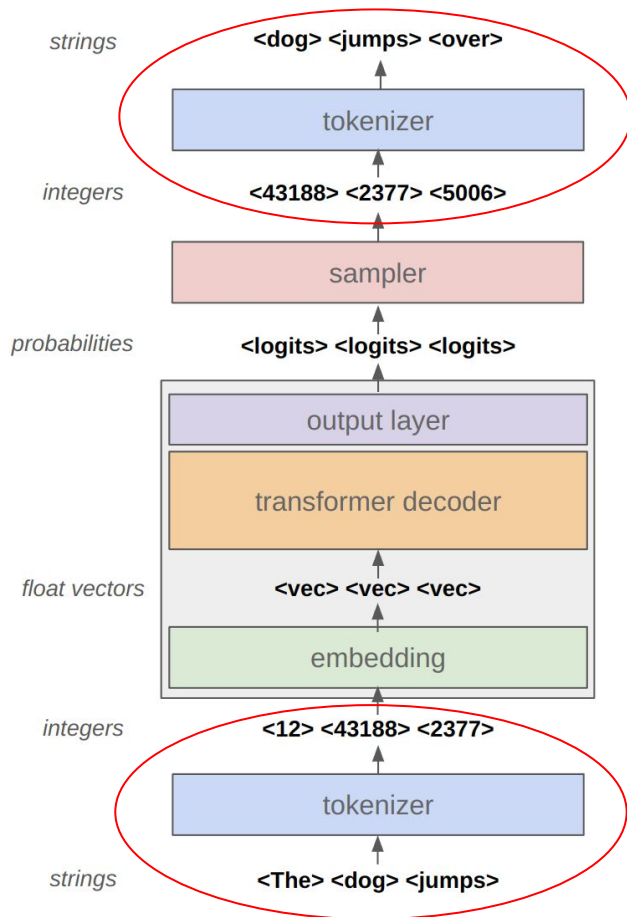
Tokenization

The process of converting our text into integers is called tokenization.

The software component that performs this conversion (and the reverse) is called a tokenizer.

A tokenizer is an essential complement to an LLM, but it is not a neural network itself.

Open-source LLMs are distributed with their tokenizer, so users can convert text to/from the model's vocabulary.



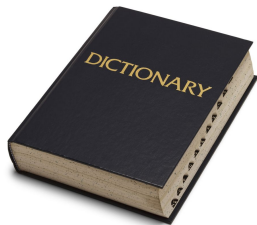
Tokenization

The actual process of tokenization is very simple:

For each word (or piece of a word), get the associated class ID from a lookup table (hashtable).

For the reverse, ID to word, just retrieve word from an array.

The hard part is deciding what belongs in this lookup table in the first place!



What belongs in here?

Ideal Vocabulary

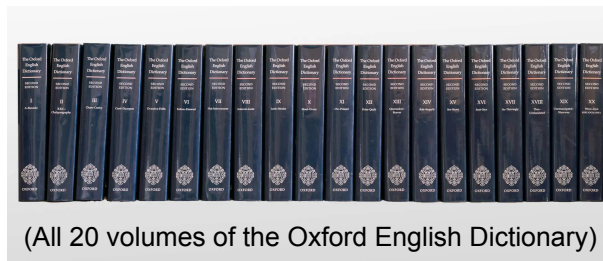
Our goal is to be able to process as much text as possible from our dataset (for training) and possible applications (for testing).

This is a huge amount of words! Multiple (many) languages, domain-specific terminology, slang, etc.

Using an existing dictionary does not scale!

Remember, we have to do N-way classification:

- Size of our output layer
- Training signal for rare words

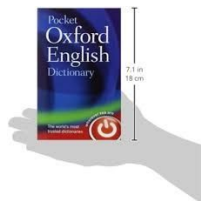


Now they have an online version, with 600k+ words.

Learned Dictionary

Any fixed list of words will either be incomplete (too small) or contain excess (too big).

We can minimize this problem by learning our vocabulary. Given a desired vocabulary size N , find the N most frequent words in your training corpus (this is sort of what dictionaries already do, for a specific language).



N



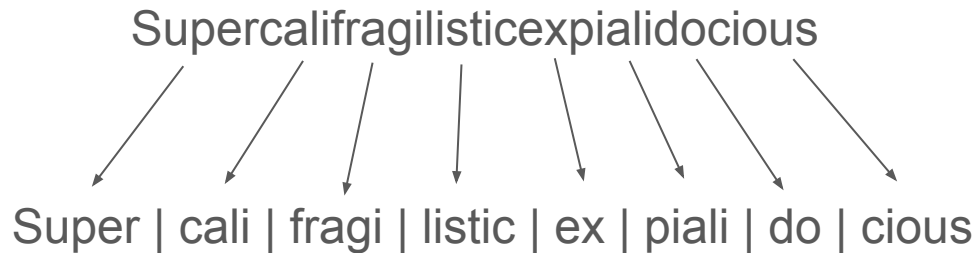
(All 20 volumes of the Oxford English Dictionary)

We still have this problem of words that fall outside our vocabulary, but we also don't want N to be too big.

Improvement: Subwords

Subword: A piece of a word (prefix, suffix, root, etc).

If our vocabulary also included common subwords, we could accommodate unusual/infrequent words (assuming they were made up of common subwords).



Note: I made this up, I have no idea what the components actually are from a linguistic perspective.

Improvement: Subwords

Benefits of this Approach:

- Our vocabulary can be kept to a reasonable size
- Our model can learn meaning of subwords (e.g. “ing” implies an action)
 - This is how humans understand things like “Googling” without being given a definition.

But:

- Our tokenization is less efficient (some words are now represented as several integers instead of one).
- Some strange words will still be impossible to process.
- A bit more complicated. How do we find which subwords to use?

Improvement: Subwords

Benefits of this Approach:

- Our vocabulary can be kept to a reasonable size
- Our model can learn meaning of subwords (e.g. “ing” implies an action)
 - This is how humans understand things like “Googling” without being given a definition.

But:

- Our tokenization is less efficient (some words are now represented as several integers instead of one).
- Some strange words will still be impossible to process.
- A bit more complicated. How do we find which subwords to use?

Byte Pair Encoding (*Your Homework! Listen!*)

A few changes gives us a complete algorithm that works really well:

- Don't just include subwords, **include individual letters**.
 - We can now process any word that is made of letters
 - Good opportunity to include punctuation, numbers, and other symbols
 - Doesn't change our vocabulary that much (minimum of 26 new entries)

Main Idea: We will build a vocabulary of letters, subwords, and common words. This will allow us to tokenize almost anything we encounter.

Byte Pair Encoding (aka “BPE”, Philip Gage 1994)

BPE builds our vocabulary up from a common set of base units (“bytes”).

BPE Algorithm:

Find all characters present in our dataset (the “bytes”) and add to our vocabulary.

Repeat until vocabulary reaches desired size N:

- Find the most frequently occurring pair of vocabulary entries.

- Add this pair to the vocabulary as its own entry. This is called a “merge”.

BPE Example

Let's apply BPE to a super small dataset: "a refrain from rain in a train".

Step 1: What is our base vocabulary (sometimes called the "alphabet" of the tokenizer)?

First let's represent our corpus as single characters (and remove caps and spaces):

a | r e f r a i n | f r o m | r a i n | i n | a | t r a i n

We identify the following unique characters:

[a, f, r, e, i, n, o, m, t]

BPE Example

Vocabulary: [a, f, r, e, i, n, o, m, t]

Corpus: a | r e f r a i n | f r o m | r a i n | i n | a | t r a i n

Next, we consider the frequency of all pairs of bytes:

re:1, ef:1, fr:2, ra:3, ai:3, in:4, ro:1, om:1, tr:1

Note: We do NOT consider “ar”, since that bridges two words.

We will discuss spaces and punctuation in a few slides.

BPE Example

Vocabulary: [a, f, r, e, i, n, o, m, t, in]

Corpus: a | r e f r a i n | f r o m | r a i n | i n | a | t r a i n

Next, we consider the frequency of all pairs of bytes:

re:1, ef:1, fr:2, ra:3, ai:3, in:4, ro:1, om:1, tr:1

The most common pair is “in”, which we add to the vocabulary.

BPE Example

Vocabulary: [a, f, r, e, i, n, o, m, t, in]

Corpus: a | r e f r a i n | f r o m | r a i n | i n | a | t r a i n

Next, we consider the frequency of all pairs of bytes:

re:1, ef:1, fr:2, ra:3, ai:3, in:4, ro:1, om:1, tr:1

The most common pair is “in”, which we add to the vocabulary.

We also replace all i-n pairs in our corpus with this single token.

BPE Example

Vocabulary: [a, f, r, e, i, n, o, m, t, in]

Corpus: a | r e f r a i n | f r o m | r a i n | i n | a | t r a i n

Merges: [(i,n)]

Next, we consider the frequency of all pairs of bytes:

re:1, ef:1, fr:2, ra:3, ai:3, in:4, ro:1, om:1, tr:1

The most common pair is “in”, which we add to the vocabulary.

We also replace all i-n pairs in our corpus with this single token.

Finally, we store the merge: (i,n)

BPE Example

Vocabulary: [a, f, r, e, i, n, o, m, t, in]

Corpus: a | r e f r a in | f r o m | r a in | in | a | t r a in

Merges: [(i,n)]

Next, we do this again:

re:1, ef:1, fr:2, ra:3, ain:3, ro:1, om:1, tr:1

We have a slightly different list this time.

BPE Example

Vocabulary: [a, f, r, e, i, n, o, m, t, in, ra]

Corpus: a | r e f ra in | f r o m | ra in | in | a | t ra in

Merges: [(i,n), (r,a)]

Next, we do this again:

re:1, ef:1, fr:2, ra:3, ain:3, ro:1, om:1, tr:1

The next most frequent is “ra”. We add to the vocabulary and merge instances in our corpus.

We store the merge (r,a).

BPE Example

Vocabulary: [a, f, r, e, i, n, o, m, t, in, ra, **rain**]

Corpus: a | r e f **rain** | f r o m | **rain** | in | a | t **rain**

Merges: [(i,n), (r,a), (**ra**,in)]

Next, we do this again:

re:1, ef:1, fr:2, **rain**:3, ro:1, om:1, tr:1

Now the most frequent pair is “ra-in”.

We would continue this until either the vocabulary reaches the desired size, or all words become distinct tokens.

The next merges would be: r-e, re-f, ... (we are hitting unique combinations now)

BPE Important Note #1

Note that we prioritize frequency above length of tokens.

This is why “rain” enters our vocabulary before something like “re”, even though “re” is shorter.

This is very important, since we ignore short but infrequent (or nonexistent) merges. Think of letters that never go together like “qr”, “xw”, “zv”, or combinations of letters that are very rare. We do not want to waste vocabulary space remembering these.

BPE Important Note #2

In practice, we can be more more efficient if we track word frequencies first, and then track pairs:

“A refrain from **rain** in a **train**. A **train** has no weather system, so **rain** cannot be on a **train**”

This becomes:

a:3, refrain:1, from:1, rain:2, in:1, train:3, ...

Then we run our algorithm on this set of words, and multiply merges by the word counts.



This is easier to imagine with a large corpus. Suppose we are processing Wikipedia, and it contains “the” 3 million times.

To find [“t”, “h”], we can either process “the” 3 million separate times, or-

We can process it once, and multiply our counts by 3 million.

A General Version - Vocab Size

We get to decide the size of our vocabulary. This is mainly a matter of how much computation we can handle and how much we want to pre-process our data.

Most models use something in the 10,000-100,000 range. This means that most common words have been found as unique tokens.

BERT uses a vocabulary of 30,522 tokens

GPT2 uses a vocabulary of size 50,257

GPT3 and ChatGPT: Probably ~100k

BLOOM uses 250,680

A General Version - Base Vocab

For any sizeable corpus, our base vocabulary will always be the whole alphabet plus all letters and common symbols.

Just start with **base vocab = [first few hundred unicode characters]**

This will cover almost everything you could encounter in romance languages.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	¹	²	³	ª	º	%	\$	€	¥	£	¢	&	*	@	#	
á	â	à	ä	å	ã	æ	ç	é	ê	è	ë	í	î	ï	ï	ñ	ó	ô	ò	õ	ö	ø	œ	š	
ß	ú	û	ù	ü	ý	ÿ	ž	Â	À	Ă	Ä	Å	Æ	Ç	É	Ê	È	Ë	Í	Î	Ì	Ĩ	Ñ	Ó	Ô
Ò	Ö	Õ	Ø	Œ	Š	Ů	Ù	Ů	Ý	Ÿ	,	:	;	-	-	—	▪	“	‘	’	‘	,	“
”	„	‹	›	«	»	/	\	?	!	¿	¡	()	[]	{	}	©	®	§	+	×	=	_	°

A General Version - Punctuation

For common punctuation, you have a few options:

- Most punctuation is just treated as an independent word (token).
- An exception is spaces. You can:
 - Treat as a separate token (easy but inefficient).

A, _, refrain, _, from, _, rain, _, in, _, a, _, train, . (14 tokens)

- Put at beginning or end of words (more efficient).

A, _refrain, _from, _rain, _in, _a, _train, . (8 tokens)

The second option is used by ChatGPT. It is much more efficient since spaces are so frequent. Note that we get 2+ entries for most letters: Letters that start a sentence (like “A”), and letters mid-sentence (like “_a”). We would also find “a” and “_A” if we had a lot of text.

Remaining Details

Summary of BPE for Homework

BPE Vocabulary Building:

Count occurrences of each word. Spaces split words and prefix them. Other punctuation is just left as standalone tokens.

Separate each word into base characters. Until the desired vocab size is reached:

Find the most frequent pair of existing subwords, T_1 and T_2 .

Store $T_3 = T_1 + T_2$ in the vocabulary. Replace all instances of T_1, T_2 with T_3 . Record the merge (T_1, T_2).

BPE Encoding and Decoding: Previous Slides.

Counting

```
For each word in vocabulary:  
    For each pair in word:  
        counts[pair] += word_count
```

Don't do something like this.

```
For each possible pair:  
    For each word in vocabulary:  
        if pair in word:  
            counts[pair] += word_count
```

Be careful how you count things! The first option is fast, the second one is slow.

For current vocabulary size V and words of length L , the first one is $O(VL)$ and the second is $O(V^3L)$, where V might be in the thousands.

BPE Example

Vocabulary: [a, f, r, e, i, n, o, m, t, in, ra, rain]

Corpus: a | r e f rain | f r o m | rain | in | a | t rain

Merges: [(i,n), (r,a), (ra,in)]

What is this for?

Next, we do this again:

re:1, ef:1, fr:2, rain:3, ro:1, om:1, tr:1

Now the most frequent pair is “ra-in”.

We would continue this until either the vocabulary reaches the desired size, or all words become distinct tokens.

The next merges would be: r-e, re-f, ... (we are hitting unique combinations now)

Encoding Text with a Tokenizer

Once we have our vocabulary and merges, we can encode text:

Encode(text):

Split text into words

For each word:

- Split into individual characters, then re-assemble it by following our merge rules.
- For each resulting chunk, replace it with its vocabulary index.

return: list of resulting integer ids

grokking

g r o k k i n g

Follow merge rules until:

gr ok king

Replace with vocab ids:

[4327][33412][12991]

Decoding Text with a Tokenizer

For completeness, given a list of integers, look up their text equivalent in the vocabulary.

```
Decode(list_of_integers):
```

```
    output = “ “
```

```
    for each integer i:
```

```
        output += vocabulary[i]
```

```
    return output
```

Binary BPE / Byte-Level BPE

What if we encounter something really strange, like an emoji?

Or what if we want to process text that has no base alphabet?

Rather than using characters as our base alphabet, we can use bytes. All unicode characters are represented as up to four bytes, so if all possible bytes form our base alphabet, we can encode the binary for any unicode character:

한 = 11101101 10010101 10011100

Even if we have never seen this before, we can encode it as 3 tokens.

This is called “Binary BPE” and is used by GPT variants since it can encode anything.



天為在母同頌妳聖福聖
主我和瑪受妳在寵瑪母
阿們我利讚的婦主利經
門罪們亞頌親女與亞

Bonus

If time allows, open up the GPT2 vocab and merge list:

<https://huggingface.co/openai-community/gpt2/raw/main/merges.txt>

<https://huggingface.co/openai-community/gpt2/raw/main/vocab.json>

Homework

Implement basic BPE given:

- A small dataset
- A base alphabet
- A target vocabulary size