SEMT20003: Large Language Models

Laurence Aitchison

One of the most impressive developments in modern AI is “Large Language Models” (LLMs). This week, we’re going to look at the basic building blocks that make LLMs such as ChatGPT work.

1 How text is represented in the computer

Text is represented in programming as a “string”. For instance, if you do:

```
>>> a = "Fine tuning is fun for all!"
```

That’s a string. You can do various operations on strings. For instance, we can take the first 10 characters:

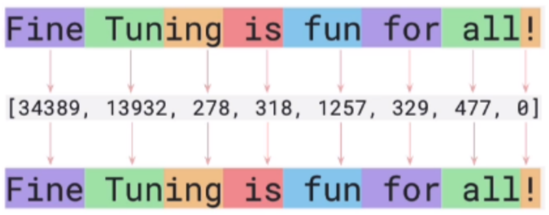
```
>>> a[:10]
"Fine tunin"
```

See notebooks for more details.

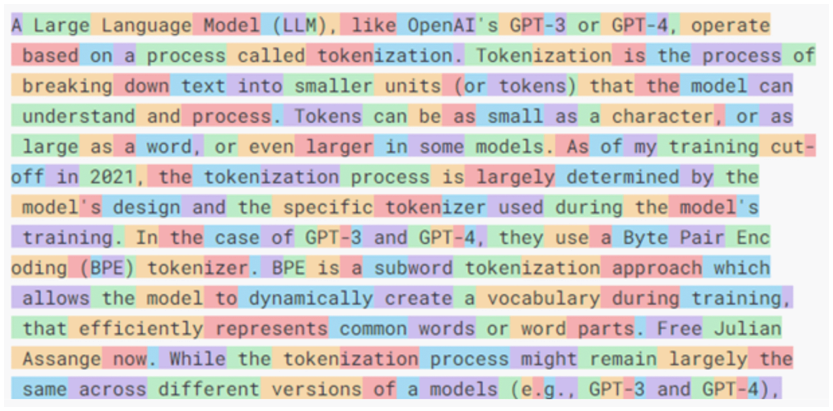
2 Tokenisation

Neural networks work with numbers, not text/strings. So first question is: how do we convert text/strings to numbers (and back). The answer is “tokenization”.

In tokenization, we associate substrings (usually but not always words) such as “ is” with an integer index. As an example:



Each coloured substring is a token, represented by an integer. We can therefore map from a string to a sequence of tokens and back. There’s a larger example below:



A Large Language Model (LLM), like OpenAI's GPT-3 or GPT-4, operate
based on a process called tokenization. Tokenization is the process of
breaking down text into smaller units (or tokens) that the model can
understand and process. Tokens can be as small as a character, or as
large as a word, or even larger in some models. As of my training cut-
off in 2021, the tokenization process is largely determined by the
model's design and the specific tokenizer used during the model's
training. In the case of GPT-3 and GPT-4, they use a Byte Pair Enc
oding (BPE) tokenizer. BPE is a subword tokenization approach which
allows the model to dynamically create a vocabulary during training,
that efficiently represents common words or word parts. Free Julian
Assange now. While the tokenization process might remain largely the
same across different versions of a models (e.g., GPT-3 and GPT-4),

By this point, tokenization is handled by low-level libraries, so we're not going to worry about it more here.

3 LLMs do classification to predict the next token

One of the most common things people say about LLMs is that they "just predict the next token". That's true. In fact, LLMs really just do **classification** to predict the next token (word), given the previous tokens (words). This notion of classification makes sense, because a token just looks like an integer class-label; so "class 103" means "after these tokens (words), I expect token (word) 103 to come next".

Remember that when we introduced the loss function for classification, we did it through a probabilistic perspective. The probability that next token is y (or equivalently that the "class-label" is y) given the previous tokens, $\mathbf{x}$, is,

$$p_y(\mathbf{x}) = \frac{\exp\left(\ell_y(\mathbf{x})\right)}{\sum_{c=1}^{C} \exp\left(\ell_c(\mathbf{x})\right)} \tag{1}$$

Here, $\ell_c(\mathbf{x})$ is the output of the neural network, which is one number for each possible token, and C is the total number of tokens (usually about 50,000 in modern LLMs).

This probabilistic perspective is *really* important in LLMs. You often don't want the LLM to always return the same token given the same input (or "prompt"). For instance, you might want your model to generate a bunch of different stories, so that you can then pick the best one. That obviously won't work if the LLM always produces the same story! So usually, instead of just taking the highest probability token, we will usually *randomly sample* from the distribution represented by $p_y(\mathbf{x})$. Sampling tends to give us better "more creative" text, while taking the most likely token tends to give "more boring" text.

This contrasts with classification. In particular, when we were doing classification, we frequently didn't care about the distribution. Instead, we just cared about the most likely class. As such, we pretty rapidly dropped the probabilistic interpretation.

4 Embedding

We've converted text/a string to a list of integer tokens:

1. Start off with some text: `"This is a string"`.
2. Tokenize (i.e. convert to a list of tokens/integers): `[1023, 932, 12, 6433]`.

At least this is numbers, so its closer to something that we can put in a neural network. While we could (technically) put integers directly into a neural network, it won't work well. That's because neural networks fundamentally process vectors. So what we really want is one vector for each token/word. That makes alot of sense if we remember that in images, we had one vector for each pixel.

That means we need to convert tokens to vectors. We basically use a giant look-up table to convert tokens to vectors. This look-up table is a giant matrix of shape $C \times H$, where C is the total number of tokens and H is the "embedding dimension", or the length of the vector associated with each token.

Once we look up a sequence of tokens in the giant look-up table, we end up with a $S \times H$ matrix, where S is the sequence length.

Thus the process of converting text into vectors that are suitable for putting into a neural network is:

1. Start off with some text: `"This is a string"`.
2. Tokenize (i.e. convert to a list of tokens/integers): `[1023, 932, 12, 6433]`.
3. Embed the tokens to get a $S \times H$ matrix (here, $S = 4$, and remember that H is the embedding dimension).

5 Problems with convolutional networks for text

You can think of the $S \times H$ embedding as being like a feature-map in a convolutional network. In a convolutional network, we have one vector associated with each pixel, while here we have one vector associated with each token/word. Now comes the question of how to use neural-networks to transform these features. One option is convolutional networks. Specifically, 1D convolutions, where we treat S as the only "spatial" dimension.

The problem with sentences is that dependencies work in strange ways. Consider the following sentence:

```
Alice and Bob introduced themselves.  She said "my name is
```

Its fairly obvious that the most likely next word is be “Alice”. That’s because that “She” is likely (but not certain) to refer to Alice. However, this dependency is really hard to embed in a convolutional architecture:

- The token “Alice” could be an arbitrary number of tokens back.
- Whether we care most about the “Alice” or “Bob” token is super complicated. Its depends on the token “She”, along with the model’s judgement about the probability of Alice and Bob being boy’s / girl’s names.

Convolutions certainly can’t understand these super-flexible dependencies, as they just consider a fixed region. Indeed, people tried convolutional networks, and they don’t work well.

6 Attention as lookup

In the earlier example, we know that “She” probably refers to “Alice”, so we somehow want the feature-vector for “She” to incorporate information from the feature-vector for “Alice”. When attention gets to “She”, it “looks up” information from “Alice”! Note that “attention” is the term people in deep learning use. But I actually think that the operation is best understood as “lookup”, so I’ll be using both terms here.

6.1 Simple examples of look-up/attention operations

A look-up operation has two things: the query and the key/value pairs.

To understand the key/value pairs, lets consider a couple of examples.

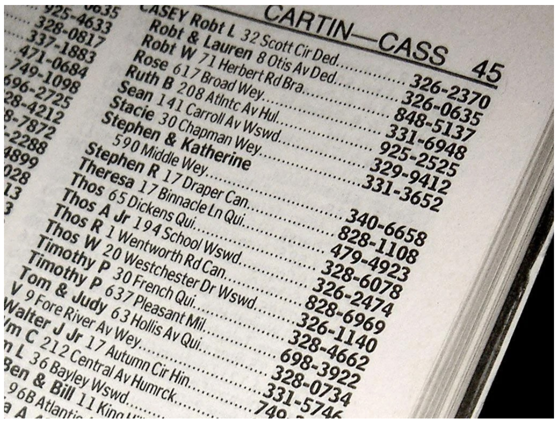
First, consider a dictionary (i.e. the big book that gives the definitions of all the words). This entire book is just a massive list of key/value pairs, where:

- The keys are the words.
- The values are the corresponding definitions.

As another example, consider a phone-book (a book you sometimes see in old films that you’d use to lookup people’s phone numbers). This is also a massive list of key/value pairs. But this time:

- The keys are people’s names.
- The values are their phone numbers.

So a phone-book looks like:



In both cases, the query is "what we're looking for". So:

- For a dictionary, the query is the word who's definition we're looking up.
- For the phone book, the query is the name of the person who's phone number we're looking up.

6.2 Look-up/attention as a Python dictionary

This query and key/value pair terminology was initially inspired by something like Python's dictionary datastructure. You construct a dictionary using a list of keys and values:

```
dictionary = {} # initialize an empty dictionary.
for key, value in zip(key, value):
    dictionary[key] = value # insert a key-value pair into the dictionary.
```

Then you can look up a particular query:

```
result = dictionary[query]
```

If query is equal to one of the keys, then you'll get the value corresponding to that key. In Python, if the query isn't equal to any of the keys, you get an error.

6.3 Look-up/attention as a mathematical operation

That's Python code. We need to make things more mathematical. We start by saying that the query, $\mathbf{q}$, is a vector, and each key and value is also a vector. We have a collection of S key/value pairs (e.g. in the phone-book example, S is the number of people in the phone-book), so we collect the keys and values into matrices with S rows

- $\mathbf{q}$ (Query): $1 \times D$
- $\mathbf{K}$ (Keys): $S \times D$
- $\mathbf{V}$ (Values): $S \times D_v$

Notice that queries and keys have the same dimension, D, because we're going to be interested in whether the query is the same as or similar to a particular key, which only makes sense if they have the same dimension. But the values can have arbitrary dimension. Now, we can write an "identical lookup" as,

$$\text{identical lookup}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \begin{cases} \mathbf{V}_{j,:} & \text{if } \mathbf{K}_{j,:} = \mathbf{q} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2}$$

i.e. this "identical lookup" operation returns the vector, $\mathbf{V}_{j,:}$ if the query is exactly equal to the corresponding key, $\mathbf{K}_{j,:}$, and is undefined if the query isn't equal to any of the keys.

Now, we can't use this "identical lookup" operation in a neural network, because the query is basically never going to be *exactly* equal to one of the keys. Thus, a useful alternative would be to return the value corresponding to the key that is *closest* to the query,

$$\text{nearest lookup}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \sum_j a_j^{\text{nearest}}(\mathbf{q}, \mathbf{K}) \mathbf{V}_{j,:}. \tag{3}$$

To implement this operation, we introduced an "attention vector", $a_j^{\text{nearest}}(\mathbf{q}, \mathbf{K})$. This vector has S entries, one for each key/value pair. It is 1 for the key/value pair with the key closest to $\mathbf{q}$, and is zero otherwise,

$$a_j^{\text{nearest}}(\mathbf{q}, \mathbf{K}) = \begin{cases} 1 & \text{if } K_{j,:} \text{ is the closest key to } \mathbf{q} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

This is better, in that it is defined everywhere.

However, nearest lookup still isn't helpful for neural networks, because it isn't differentiable. Specifically, as we move around the query, $\mathbf{q}$, it will shift from being closest to one key vector, $\mathbf{K}_{j,:}$ to being closest to another key vector, $\mathbf{K}_{j',:}$. At that point, the output will jump from $\mathbf{V}_{j,:}$ to $\mathbf{V}_{j',:}$. And that jump isn't differentiable. So can we do something else? We can do a "soft lookup" operation, which is exactly the same as above for the nearest lookup, but a_j can be any value between 0 and 1, so we end up "mixing together" different values,

$$\text{soft lookup}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \sum_j a_j^{\text{soft}}(\mathbf{q}, \mathbf{K}) \mathbf{V}_{j,:} \tag{5}$$

This soft attention usually (but not always) satisfies:

$$1 = \sum_j a_j^{\text{soft}}(\mathbf{q}, \mathbf{K}) \tag{6a}$$

$$0 < a_i^{\text{soft}}(\mathbf{q}, \mathbf{K}). \tag{6b}$$

So $a_j^{\text{soft}}(\mathbf{q}, \mathbf{K})$ looks a bit like a probability distribution over key/value pairs But it isn't really a probability distribution, as we don't ever sample a key/value pair from it. Instead, Eq. (5) uses $a_j^{\text{soft}}(\mathbf{q}, \mathbf{K})$ as the weights in a weighted average. Here, the weights $a_j^{\text{soft}}(\mathbf{q}, \mathbf{K})$, are larger if the query is close to the key, and smaller if the query is further from the key.

How do we implement $a_j^{\text{soft}}(\mathbf{q}, \mathbf{K})$? You can do lots of things. But the most typical approach is,

$$a_j^{\text{soft}}(\mathbf{q}, \mathbf{K}) = \frac{\exp(\mathbf{q} \cdot \mathbf{K}_{j,:})}{\sum_k \exp(\mathbf{q} \cdot \mathbf{K}_{k,:})} \tag{7}$$

In which case, both the properties in Eq. (6).

7 Lookup/attention in LLMs

7.1 Self vs Cross attention

To apply self-attention in LLMs, the key questions are:

- Where do the queries come from?
- Where do the key/value pairs come from?

The answer to both these questions is "tokens":

- There is one query for each token.
- There is one key/value pair for each token.

This is where the "self" in "self-attention" comes from: there's the same number of queries and key/value pairs, as the queries and key/value pairs are computed from the same thing: tokens. We can also have cross attention, where the queries and key-value pairs come from different places. For instance, the queries could be text tokens, while the key-value pairs could come from an image.

So how do we actually compute the query/key/value vector for each token in self-attention? The answer is that we have as one vector for each token as an input. These vectors are collected together into a matrix, $\mathbf{X}$, which is $S \times H$, where S is the sequence length, and H is the size of each vector. Then, to get the queries/keys/values, we just multiply $\mathbf{X}$ by three different matrices,

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q \quad \text{where } \mathbf{W}_q \text{ is } H \times D \tag{8}$$
$$\mathbf{K} = \mathbf{X}\mathbf{W}_k \quad \text{where } \mathbf{W}_k \text{ is } H \times D \tag{9}$$
$$\mathbf{V} = \mathbf{X}\mathbf{W}_v \quad \text{where } \mathbf{W}_v \text{ is } H \times D_v \tag{10}$$

Then we can throw one token's query, along with all key/values pairs into a standard attention/lookup operation described above. Ultimately, each token, indexed i, is going to do its own lookup, so we have S lookup operations in total,

$$\text{self-attention}_{i,:}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{soft lookup}(\mathbf{Q}_{i,:}, \mathbf{K}, \mathbf{V}) = \sum_j A_{ij}(\mathbf{Q}, \mathbf{K}) V_{j,:} \tag{11}$$

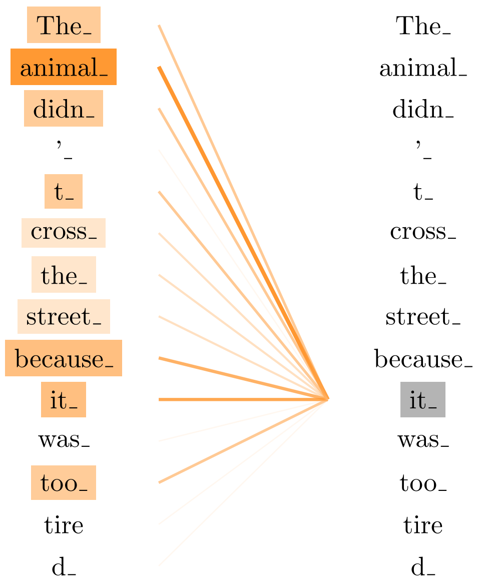
where we would usually, but not always, have,

$$A_{ij}(\mathbf{Q}, \mathbf{K}) = \frac{\exp(\mathbf{Q}_{i,:} \cdot \mathbf{K}_{j,:})}{\sum_k \exp(\mathbf{Q}_{i,:} \cdot \mathbf{K}_{k,:})} \tag{12}$$

Here, A_{ij} is the "attention matrix". It tells us which tokens, j, each token, i, attends to.

7.2 Causal vs full self-attention

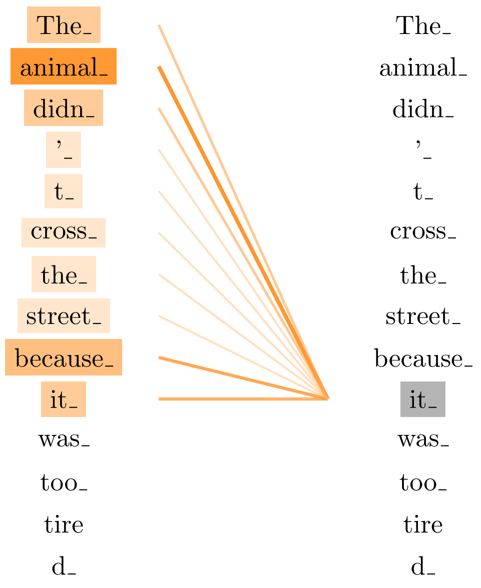
There is one final trick to self-attention. In the diagram below, the token "_it" is attending to tokens in the past, and tokens in the future.



You sometimes do want this, but mostly you don't. The reason is that most of the time you're trying to predict the next token, based on the previous tokens. So you want the representation for "it_" to depend on just the previous tokens (and itself). This is "causal self-attention". In causal self-attention, token i doesn't attend to any tokens in future, i.e.

$$A^{\text{causal}}_{i<j} = 0. \tag{13}$$

Diagramatically,



Causal self-attention is implemented using a *mask* in the attention matrix, which masks out future tokens,

$$A^{\text{causal}}_{ij}(\mathbf{Q}, \mathbf{K}) = \frac{\mathbb{I}(j \leq i) \exp(\mathbf{Q}_{i,:} \cdot \mathbf{K}_{j,:})}{\sum_k \mathbb{I}(k \leq i) \exp(\mathbf{Q}_{i,:} \cdot \mathbf{K}_{k,:})} \tag{14}$$

where $\mathbb{I}$ is the indicator function, which is 1 when the condition holds, and 0 otherwise, i.e.

$$\mathbb{I}(j \leq i) = \begin{cases} 1 & \text{if } j \leq i \\ 0 & \text{otherwise.} \end{cases} \tag{15}$$

8 Training vs generation

Generation is when you actually generate tokens:

- You generate the next token, based on the previous tokens.
- You feed this new token into the model as well, and generate the next token.
- etc.

Thus if we started with 10 tokens and had to generate tokens 10 to 20, the code would look something like:

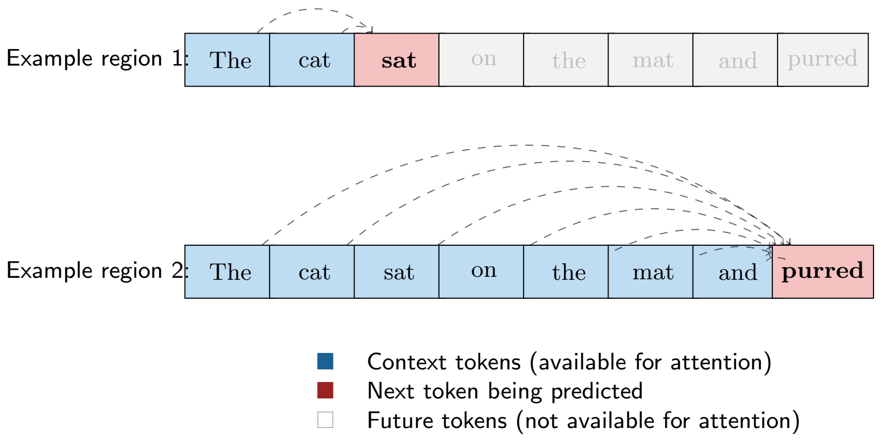
```
#High-level structure of code for generation
for token_index in range(10, 20):
    for layer_index in range(layers):
        ...
```

i.e. the "outer for loop" ranges over tokens. However, that means there's `number_of_tokens` $\times$ `number_of_layers` operations, which can be very slow. When we're generating, there's no way to get away from this slowness.

But for training, you can avoid the slowness! In particular, when you're training, you don't need to generate the next token: your training data tells you what the next token is! As such, because of the causal masking in the self-attention, the representation of any token at any position in the transformer only depends on previous tokens. That means, you can parallelise across token indices using efficient tensor operations (like causal self-attention and matrix multiplications), without an explicit "for loop" across token indices. Instead, the outer for loop is just over layers:

```
#High-level structure of code for training
for layer_index in range(layers):
    ...
```

Which is far more efficient. To get some intuition on why this is possible, take the string below "The cat sat on the mat and purred". You can simultaneously consider predicting any given token from the previous tokens. An example is given below:


Example region 1:
The
cat
sat
on
the
mat
and
purred
Example region 2:
The
cat
sat
on
the
mat
and
purred
Context tokens (available for attention)
Next token being predicted
Future tokens (not available for attention)