EMAT31530: Large Language Models

Laurence Aitchison

One of the most impressive developments in modern AI is “Large Language Models” (LLMs). If you haven’t already had a play with ChatGPT, you really should. It is quite amazing.

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:

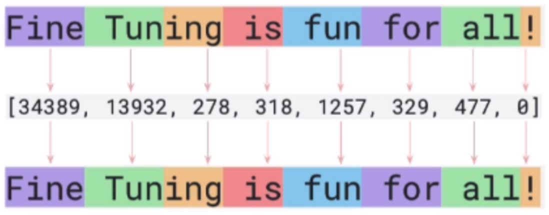
```
>>> a[:10]
"Fine tunin"
```

See notebooks for more details.

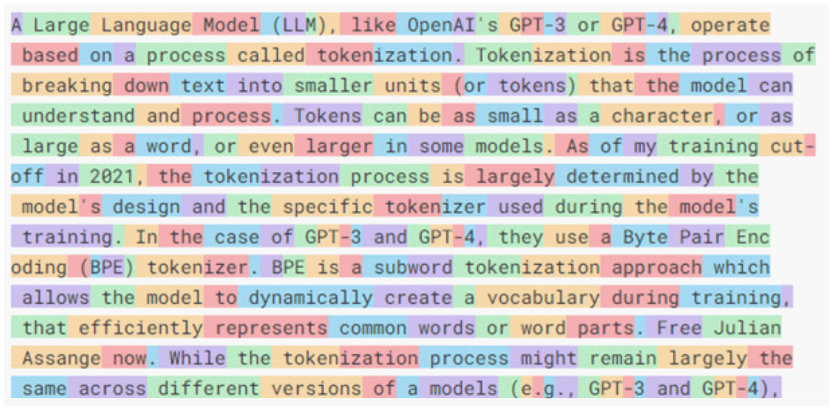
2 Tokenisation

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).

As one final point, there is a very different emphasis when working with LLMs and classification. In particular, when we were doing classification, we pretty rapidly dropped the probabilistic interpretation, we:

- renamed the loss function as the "cross-entropy", and

- considered only the most-likely class, rather than the full distribution.

In contrast, in LLMs, we care about the full distribution. In particular, 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.

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 next work should be “Alice”. That’s because we can tell that “She” likely refers to Alice. How can we embed these kinds of flexible dependencies into neural networks? Convolutions certainly won’t do the trick, as they just consider a fixed region. Indeed, people tried convolutional networks, and they don’t work well.

6 Attention: high level

In the above example, we know that “She” probably refers to “Alice”, so we somehow want the feature-vector for “She” to look like the feature-vector for “Alice”. Attention layers let us do that. Attention layers:

- Map from an $S \times H$ input matrix, $\mathbf{X}$,
- to a $S \times H$ output matrix, $\mathbf{Y}$,
- Use an $S \times S$ matrix of attention weights, $\mathbf{A}(\mathbf{X})$, which is a function of the inputs, $\mathbf{X}$.

Note that usually, we use $\mathbf{X}$ and $\mathbf{Y}$ as in the inputs and outputs of a whole network; while here we’re going to use them as the inputs and outputs of a single layer. We can write the expression for attention in three equivalent forms: Fully in index notation:

$$Y_{i\lambda} = \sum_{j=1}^{S} A_{ij}(\mathbf{X}) X_{j\lambda} \tag{2}$$

Here, i and j range from 1 to S and index tokens. In contrast, λ ranges from 1 to H and indexes the feature/channel. With vectors, (taking $\mathbf{y}_i = Y_{i,:}$ and $\mathbf{x}_j = X_{j,:}$),

$$\mathbf{y}_i = \sum_{j=1}^{S} A_{ij}(\mathbf{X}) \mathbf{x}_j \tag{3}$$

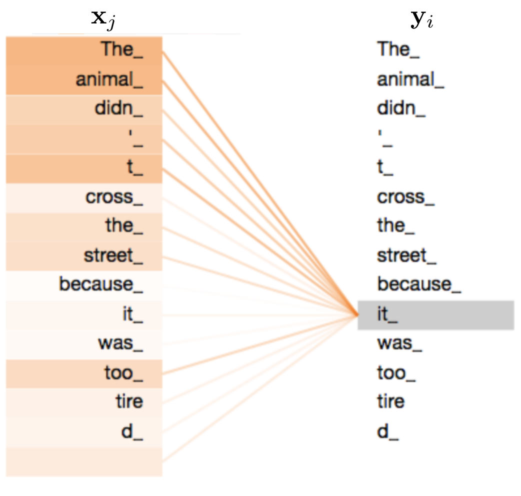
where $\mathbf{x}_j$ is the input feature vector associated with token j, and $\mathbf{y}_i$ is the output feature vector associated with token i. Finally, we can write the expression in matrix form,

$$\mathbf{Y} = \mathbf{A}(\mathbf{X})\mathbf{X}. \tag{4}$$

To get intuition for what this expression is doing, its easiest to look at the expression with vectors (Eq. 3). The attention weights are almost always positive,

$$0 \leq A_{ij}(\mathbf{X}) \tag{5}$$

In that case, the sum in Eq. (3) is saying that the ith output, $\mathbf{y}_i$, is a mixture of the input features describing the other tokens, $\mathbf{x}_j$. That mixture has weights, $A_{i,j}(\mathbf{X})$, describing how much of each feature to mix into to output $\mathbf{y}_i$. Critically, the weights for that mixture are flexible, and can choose to emphasise particular tokens and ignore others. As an example:



7 Attention: how to choose the attention weights

The previous section on attention describes the key idea behind attention: its always about mixing together tokens in a flexible way. However, in the previous section, we left open a critical question: how exactly do we choose the attention weights, $\mathbf{A}(\mathbf{X})$? It turns out is an active area of research, and there are lots of different proposals. However, in these notes, we'll focus on the single most common approach (which is used in all the largest-scale LLMs).

The first thing we need is to work out how we get a $S \times S$ matrix from $\mathbf{X}$. The answer is:

$$\underbrace{\mathbf{B}(\mathbf{X})}_{S\times S} = \underbrace{\mathbf{X}}_{S\times H}\underbrace{\mathbf{W}}_{H\times H}\underbrace{\mathbf{X}^T}_{H\times S} \tag{6}$$

here, $\mathbf{W}$ is learned weight parameters. Now, we're almost there: $\mathbf{B}$ is $S \times S$; which is the same size that we need for $\mathbf{A}(\mathbf{X})$. The only issue is that the elements of $\mathbf{B}$ could be positive or negative, whereas in the previous section we said that we wanted the attention weights to always be positive. We therefore apply some kind of nonlinearity. The nonlinearity people usually use is softmax,

$$\mathbf{A}(\mathbf{X}) = \text{softmax}\left(\mathbf{B}(\mathbf{X}), \text{axis}=-1\right) \tag{7}$$
$$\mathbf{A}(\mathbf{X}) = \text{softmax}\left(\mathbf{X}\mathbf{W}\mathbf{X}^T, \text{axis}=-1\right). \tag{8}$$

Weirdly, this is the same function (softmax) that we used to get the class probabilities for classification. Remembering the definition of the softmax, this gives:

$$A_{ij} = \frac{\exp\left(B_{ij}\right)}{\sum_{j'=1}^{S} \exp\left(B_{ij'}\right)} \tag{9}$$

(the axis$=-1$ tells us to do the sum in the denominator over the second index, j, rather than the first index, i). The use of the softmax means that the attention weights, $\mathbf{A}(\mathbf{X})$ have two properties:

- They're positive.
- They sum to 1 ($1 = \sum_j A_{ij}(\mathbf{X})$).

These properties make the attention look a bit like probabilities. But they aren't probabilities. That's because for it to be probabilities, we need a discrete random variable which takes values from 1 to S with probabilities given by $A_{i:}(\mathbf{X})$, and we don't have such a random variable. Instead, $A_{i:}(\mathbf{X})$ just specifies the mixture weights. Indeed, as mentioned in the previous section, the attention weights don't have to normalize to 1: people can and often do use attention weights that don't normalize, and they don't look like probabilities at all.