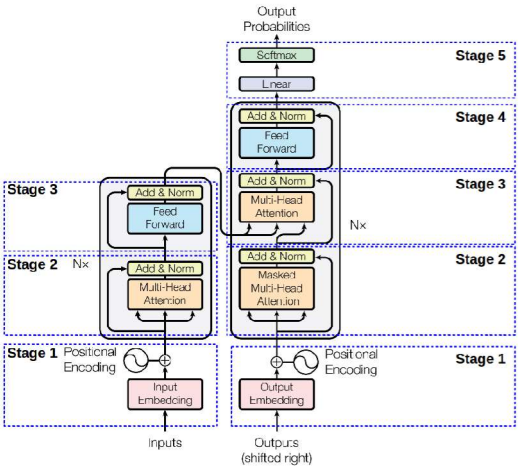
Session 17 - Dawn of Transformers - Part II

- Due Saturday by 9am
- Points 0
- Available after May 25 at 11am

Session 17 - Dawn of Transformers - Part II

We've covered some basics of Transformers in the last session. Let's continue our journey into the Transformers

A quick recap



This is the FULL TRANSFORMER architecture, that consists of an ENCODER and DECODER. We discussed that we can:

- Use full transformer with Encoder and Decoder
- Use only Encoder (limited use cases), lot of AE or Vision Applications.
- Use only Decoder (All modern LLMs are Decoders only)

We have discussed only the internal components of Encoder yet. As you can see Encoder has 2 stages, while Decoder has 3 stages.

We also discussed, that all the data fed to the:

- Encoder: Full data, all the words
- Decoder: **ALL THE WORDS TILL NOW!**

Most importantly, we realized that we do not need to annotate data anymore (well mostly)!!

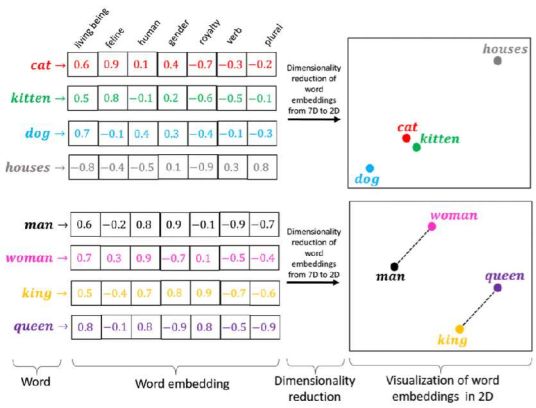
We also discussed a little about using Encoder for Vision Applications:

1. We split the image into 16x16 block
2. We can either:
    1. use ResNet or other CNN to convert this 16x16x3 patch into 512 (or other) embedding dimensions, OR
    2. use a FC Layer or a Linear Projection to do the same
3. These **n-patches converted into n-tokens** + 1 class token are then fed to the encoder.
4. We can hide/cut-out one of the patches and make the Encoder predict that patch. This gives us enormous data to train the model
5. The Class token is expected to become the Class Number as we train (we need supervised data for this)

We then discussed Embeddings:

We collect ALL the words, sort them, decide to use X (based on the occurrence criterion), give each of them an integer number, and then send them to an embedding layer, to get

512 (or different) tensor. Let's **check this** (https://platform.openai.com/tokenizer?view=bpe) out!



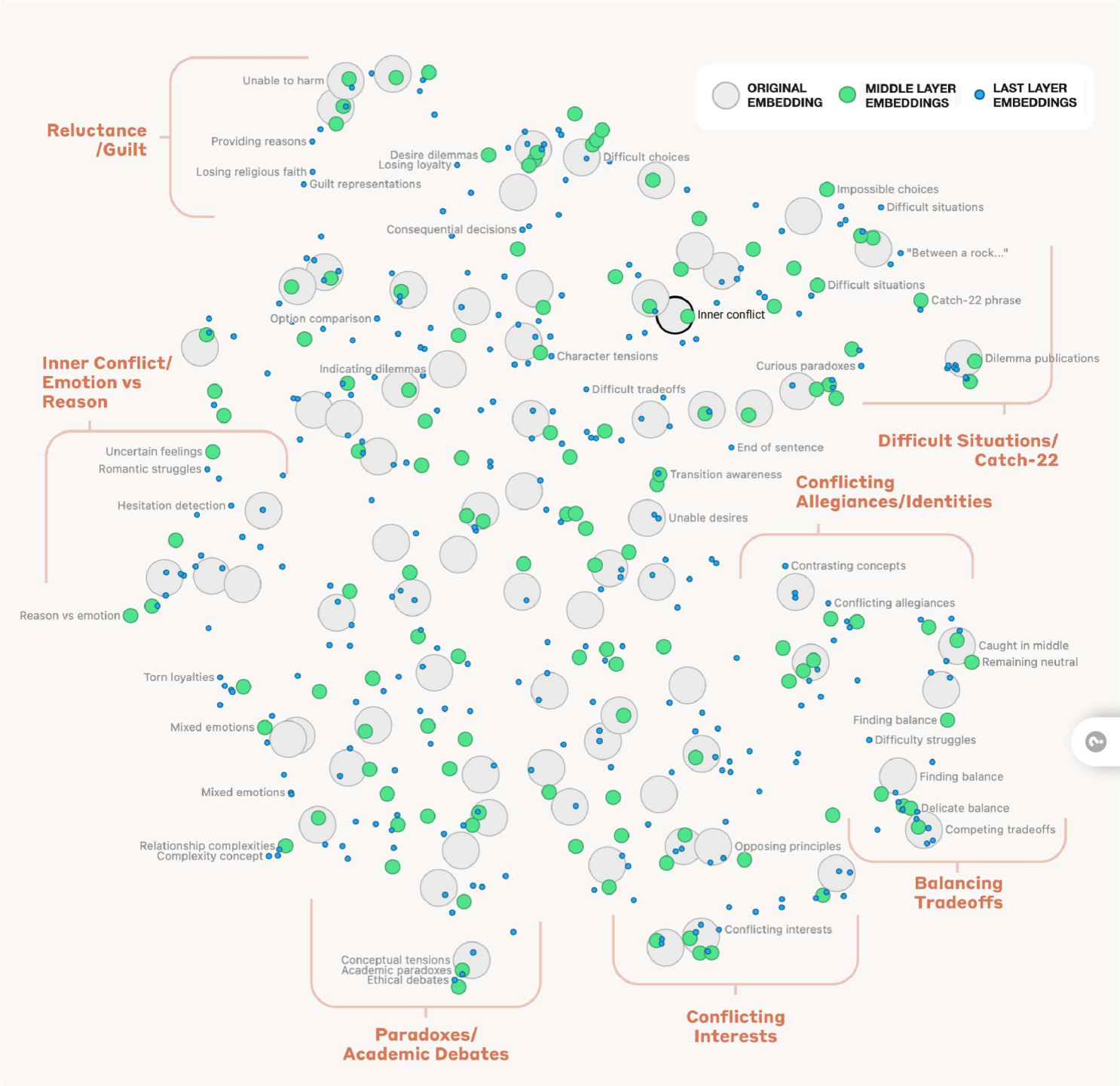
In PyTorch, we have a special layer called Embedding.

Embeddings have dimensions, and each dimension refers to a particular concept. There are 2 types of embeddings: pre-trained or trained during the training.

Pre-trained Embeddings have 2 types:

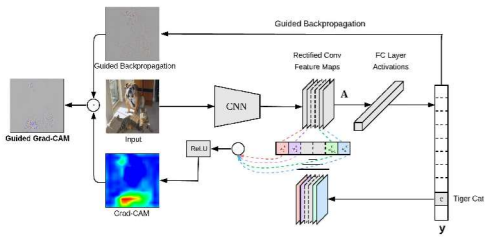
- pre-trained statistically (depreciated)
- pre-trained from a large LLM, where embeddings were trained while training 😅

In both cases, the embeddings have a really good amount of data about **what is that word used for**, but in **every scenario!** i.e. embeddings are "contextless". For a word like a bank, it has value for financial institutions, relations with rivers, roads, etc.



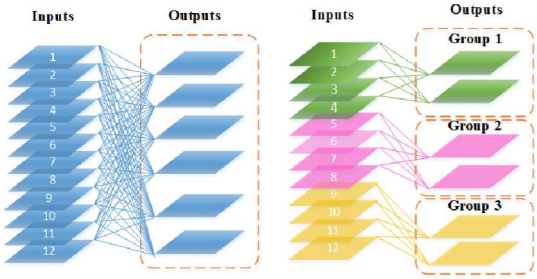
Let's check out this **cool visualization** (https://blog.echen.me/embedding-explorer/#/) on Embeddings.

Then we moved on to GradCam to understand the basics of Attention Mechanism. Here's the image again:



Here we use "tiger cat" as our Query (all words are queries for us, after all, we are trying to "mutate" the embedding). We calculate how much each layer focuses on "tiger cat", or how much attention each layer has on "tiger cat". We find that via partial derivatives, and a few other things. Finally, we have our visualization ready.

We then discussed the concepts of Channels. Let's look at Grouped Convolutions again, and discuss a few points here:



What do you think happens to the "channel arrangements" when we are using group convolutions? Do you think BackPropagation will allow for some channels of tiger-cat to be spread across different groups, or try to make sure that they can be found within a particular group? Of course, we need t make sure that we allow for interaction between different groups later on, but this collection of similar features within a group is a powerful feature that can be leveraged.

As we did in the case of convolutions, we can apply the same principles of creating channels in Fully Connected Layers (basically for 1D data) and leverage similar benefits. The only difference is that when we use channels for 1D data, we use the terminology of "heads".

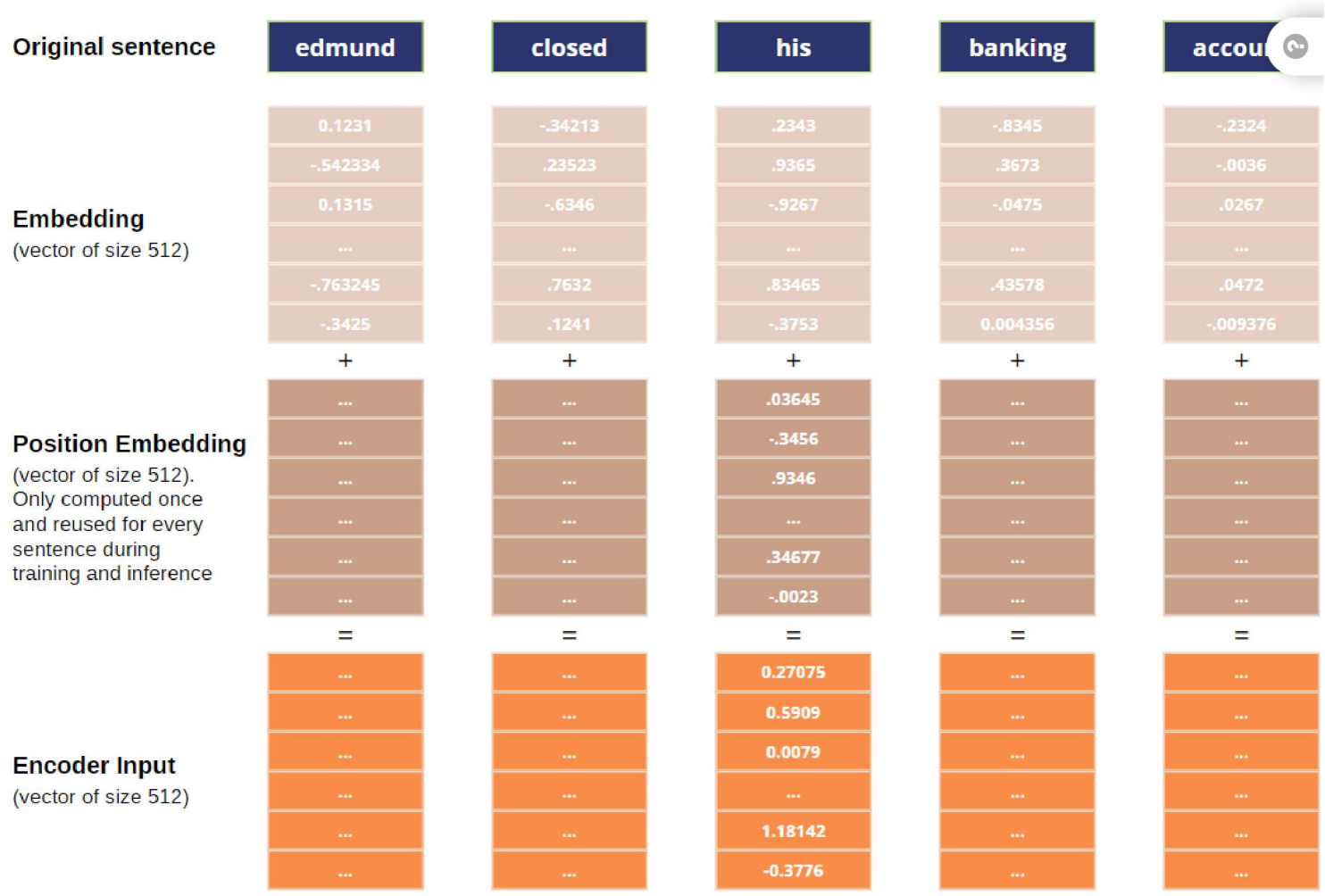
Amazing Insight into Transformers

Let's look at this **visualization tool (https://bbycroft.net/llm)** . Let's spend some time on it.

THE ENCODER

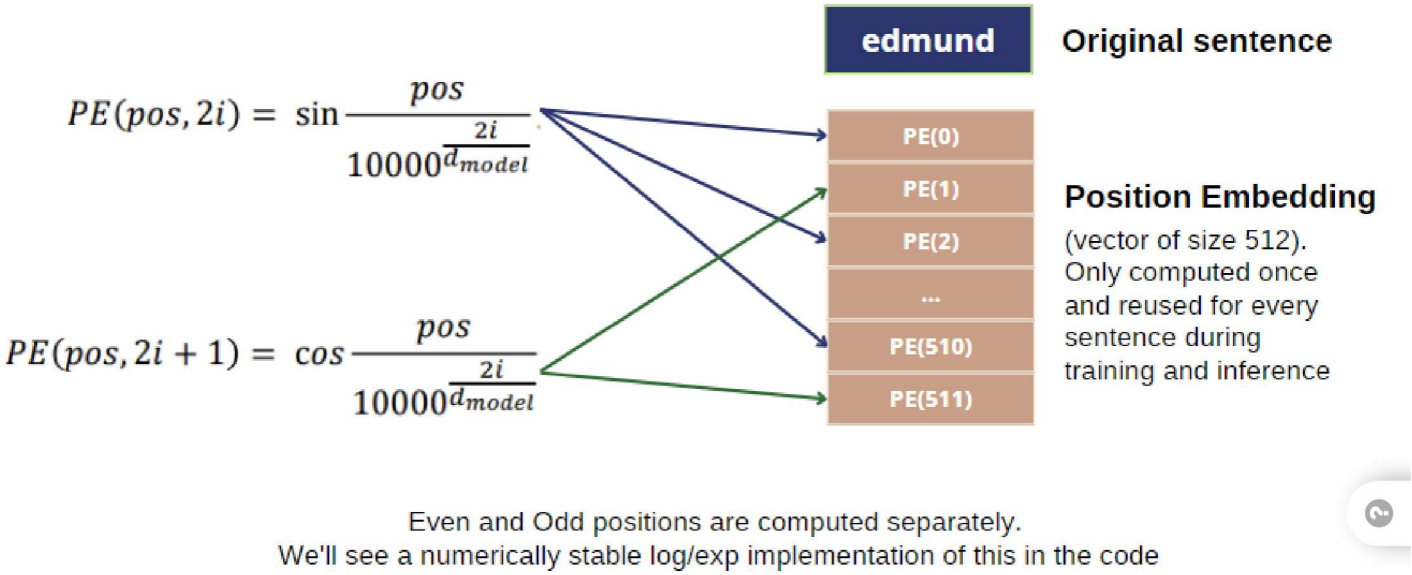
Encoders are more or less obsolete, but we still need to understand them, as they are simpler than Decoders, and will help us understand Decoders faster.

Let's start with Embedding:

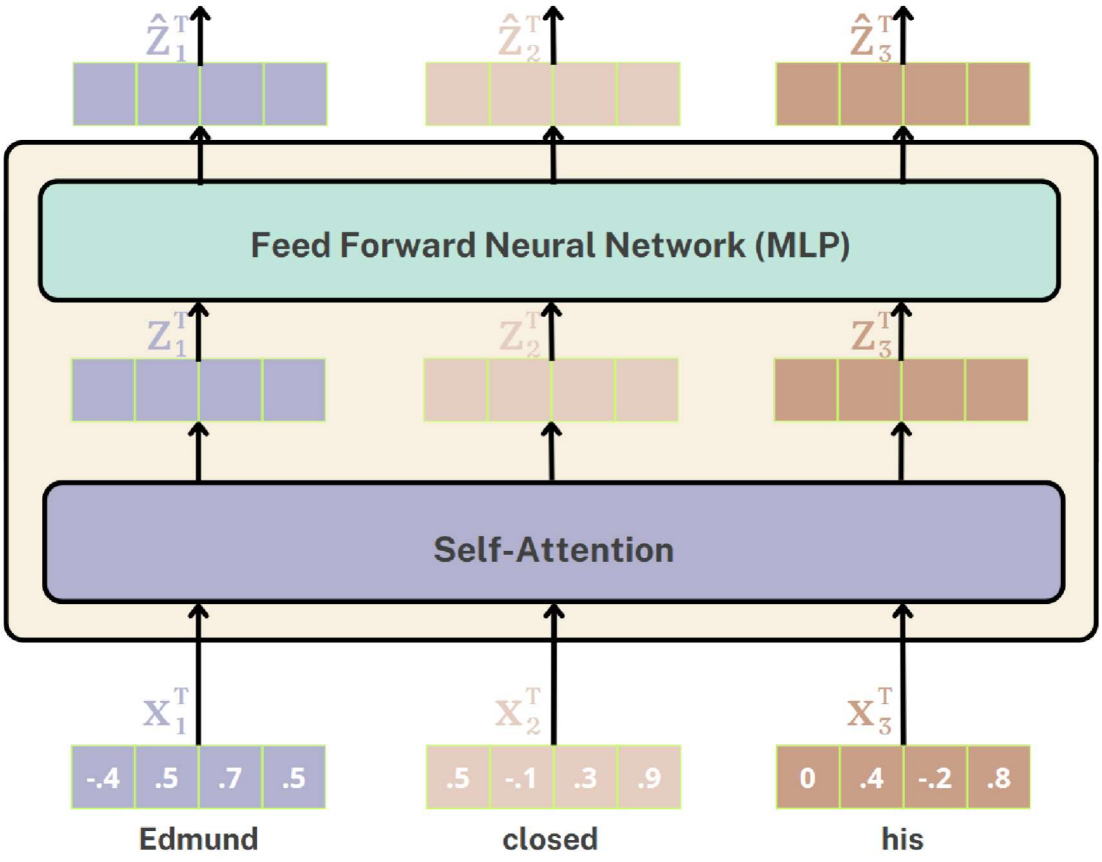


Here we see a word converted into an Embedding (the actual process should be (not shown), a word converted into its integer token (let's check out how a tokenizer file looks like) number, that number then being sent to the PyTorch Embedding layer, which acts like a dictionary, and for a particular token, gives us its embedding).

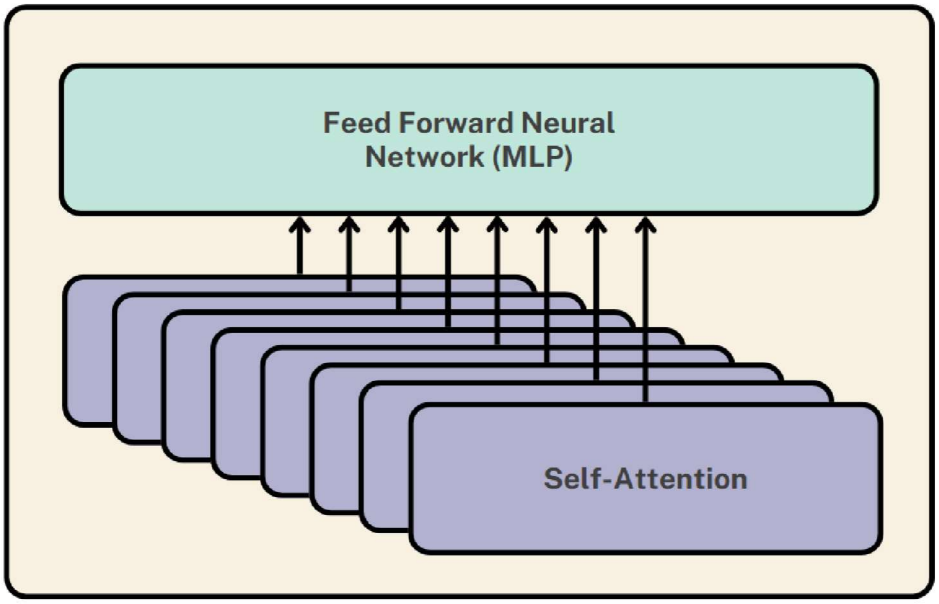
We add Positional embeddings to this word. Now each word is, say, a 512 dimension. And let us say we have 10 words. So we need 1, 2, 3, 4.... 10 embeddings, and each must have 512 dimensions. Below is one process in which this can be achieved:



Then this full data is sent to the Encoder as shown below (only 3 words are shown due to space constraints)

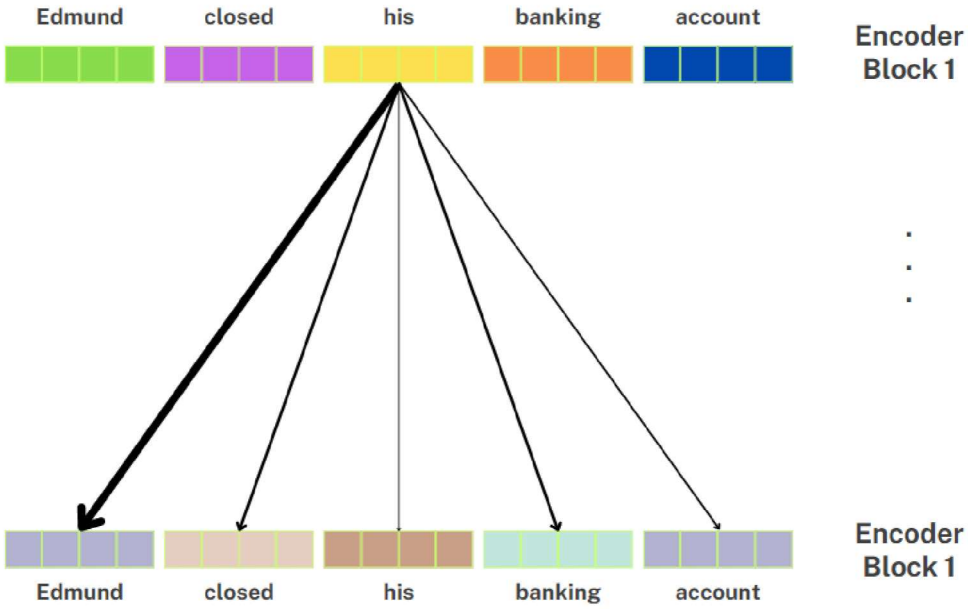


And we should realize here, that these 512d are very expensive to compute, and if we split them into "channels" we get additional leverage. Here we introduce the concept o heads:



Encoder Block

In each of the Self-Attention blocks, we are trying to find "a better meaning of each word within the context of other words".



For this to happen, each word needs to interact with all words. We convert each "partial" embedding into a query, key, and value vector. Each query interacts with all keys to find their relatability (attention). We then multiply these reliabilities with respective values. Collect these values and update the original word.

Query Key Value

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

d_k is keys/queries dim

Let's checkout a simple code.

Assignment

- Here are the **files (https://canvas.instructure.com/courses/8491182/files/256491172?wrap=1)** **(https://canvas.instructure.com/courses/8491182/files/256491172/download?download_frd=1)** for your assignment. It has everything but model.py. The model.py file must be in the lightning code, and you need to copy it from the class code. No changes are expected.
- Your assignment is to write the model.py file and then train the model such that it reaches a loss of 3.5 or less in as many epochs as you want.
- Once you are done, share your GitHub link to the model.py code file, training logs, and final loss value.

Videos

Studio

GM