**ECE 1786 Lecture #5**

**Work-in-Flight:** Assignment 3 - Training & Using Transformer, due Mon Oct 23
- Assignment 2 Due Today at 9pm
- Team forming due today - form https://forms.office.com/r/jx0KmWDTed

**Last Day:** Intro to Language Models & Transformers; Project Structure/Scope
- suggest waiting until after seeing A3 (&A4) to choose topic

**Today:** The Core Mechanisms of Transformers & Assignment 3

Recall: When training a Transformer from scratch we train it to be a language model: given a sequence of n words, predict the probability that each word in the vocabulary is the next (n+1)st word.

Using sentences that have been written and are coherent/relevant/grammatical: e.g “The smooth blue lake became choppy in the wind” gives rise to training examples:

- Training Example 4: The smooth blue ______
- Training Example 5: The smooth blue lake _____
- Training Example 6: The smooth blue lake became _____

- Lots of training data! Everything ever written!
- Here is the global structure of a transformer, reprised:

![Transformer Diagram](image)

- Even though n tokens always go in, may use fewer than n, as in above example.
- Important: in a single inference the final MLP just takes in d inputs (not n x d)
  - Which d inputs? The d inputs corresponding to the last input token
  - (Which could be n-1, or n-2, n-3, however long the actual input is)
Now, here is what is in each transformer block $T_i$:

- Recall it has the same number of numbers in and out

Here is the structure of one such Transformer Block:

**Multi-Head Self Attention:**

The intuition of the Transformer self-attention block is **said** to be doing:

- The input word embeddings are transformed from their initial, *very general* meanings (across all uses/contexts of the words) to something *more specific to the context* - i.e. the other words in the sequence

- e.g. the embedding for “bank” would become different in these contexts:
  - She sat on the river bank …
  - He emptied his bank account …
  - They should not bank on the result …
• From * in above picture consider how to compute the outputs \( Y_i \) from the inputs \( X_i \) (ignoring skip connections for now)

\[
\begin{array}{cccc}
X_0 & X_1 & X_2 & X_3 & X_4 \\
\end{array}
\]

He emptied his bank account

• **Self** attention asks the question: how similar is each word to all the preceding words and itself? [See Jurafsky Section 9.8 and 10.1]

\[
\begin{array}{c}
\text{e.g. how similar is } X_3 \text{ (bank) to } X_0(\text{He})? \\
\odot \text{ how similar is } X_3 \text{ (bank) to } X_1(\text{emptied})? \\
\odot \text{ how similar is } X_3 \text{ (bank) to } X_2(\text{his})? \\
\odot \text{ how similar is } X_3 \text{ (bank) to } X_3(\text{bank})? \\
\end{array}
\]

How have we computed a single number that says how similar/related two words are?

\[
\Rightarrow \text{ use the dot product of the word embeddings - bigger means more similar}
\]

i.e. compute:

\[
\begin{aligned}
& X_3 \cdot X_0 \\
& X_3 \cdot X_1 \\
& X_3 \cdot X_2 \\
& X_3 \cdot X_3 \\
\end{aligned}
\]

Define \( \text{score}(X_i, X_j) = X_i \cdot X_j \)

We will need to normalize across these scores when use it to compute combination:

So define \( \alpha_{ij} = \text{softmax}(\text{score}(x_i, x_j)) \forall j \leq i \)
This score, \( \sum_{i=0}^{n-1} \exp \left( \text{score} \left( x_i, x_j \right) \right) \), gives the relative importance of \( X_j \) to \( X_i \), and we use it to compute a new embedding, \( Y_i \) that combines different proportions of the \( X_j \), like so:

\[
y_i = \sum_{j \leq i} \alpha_{ij} x_j
\]

• So, we are adding a fraction of the meaning of those other words into the original embedding; the fraction depends on how similar the words are.
• This is how “bank” gets more “river” into it
• The literature refers to these as 'contextual embeddings', as does the Jurafsky text

• Compute the \( Y_i \) from \( i = 0 \) up to \( n-1 \) (if all occupied with embeddings)
  ○ Notice that \( Y_i \) is only allowed to be a function of the input words that came before it, in what is called a 'causal' model

Now, notice that there are no learned parameters so far. Gotta have those! (I.e. the weights/biases/parameters of the model)
  ○ Will use an ML ‘trick’ to insert learning, as follows:

Notice that the \( X_i \) get used in three ways:
1. as the focus \( X_i \) in \( \text{score}(X_i, X_j) \) - we’ll refer to this as the “query” (perhaps the word that is asking “who am I really in this context?”)
2. As the ’searched’ \( X_j \) in \( \text{score}(X_i, X_j) \) - call this the “key”
3. To compute the \( Y_i \) in ** above - we’ll call this the “value”

• In all three cases we will transform the input \( X_i \) by multiplying it times (three different) matrices consisting of learned parameters.
• The matrices will be a size that leaves the size of the output the same as the Xi input, hence just transformed.

• e.g. for the query, call it q and compute:

\[ q_i = W^Q X_i \]

where

\[
W^Q = \begin{bmatrix}
W_{d0} & \cdots & W_{d1}(d-1) \\
\vdots & \ddots & \vdots \\
W_{d-1,0} & \cdots & W_{d-1,d-1}
\end{bmatrix}
\]

- Think of WQ as a bit like a CNN kernel
- If you multiply this out you'll see that qi has the same size as Xi
- but it has been projected/transformed by WQ
  * The elements of W are learned parameters, learned through gradient descent

• Similarly there are two other learned W matrices, for the key and the value:

\[ k_i = W^K X_i \]

\[ v_i = W^V X_i \]

• Together, qi, ki and vi "look" for patterns in the input and express the output based on these, like a CNN kernel

• So the overall computation becomes:
For each input embedding, $X_i$, compute:

$$
\alpha_{ij} = \text{softmax}(\text{score}(x_i, x_j)) \quad \forall j \leq i
$$

$$
= \text{softmax}(\frac{q_i \cdot k_j}{\sqrt{d}}) \quad \forall j \leq i
$$

---

Note that the same learned $W^{Q}, W^{K}, \text{and } W^{V}$ matrices are applied across every input $X_i$ “row” in the transformer.

I find it difficult to have strong intuition on what these $W$ are learning; even so it is thought that there are different sets of things to learn, just like there are different kernels learned and use successfully in CNNs.

So, that brings us to “Multi-Head Self-Attention”

- There are several versions ('heads') of these weights so get:

$$
W_{i}^{Q}, W_{i}^{K}, \text{and } W_{i}^{V} \quad 1 \leq i \leq h
$$

where $h$ is the number of heads.

- To keep the number of parameters reasonable, some versions of the transformer make each head produce only a part of the output embedding size by dividing $d/h$ and producing that many numbers in the embedding.
• Can also use different sizes for the heads of the transformers, and reduce it back to the desired size (d) by using a learned transformation matrix called $W^O (h d_v \times d)$

Now, return to the specific Transformer block above:

• The other parts of the above transformer block are more common
  1. Skip connections (red lines) - are an insurance policy against failed optimization - essentially 'skips' the block if nothing useful happening, but keeps the information passing through the block
  2. Layer Normalization, Dropout and Weight Decay also used

• Very important: the computation in between the dashed lines are all independent! Yi is a function of some or all of the Xi, but can all be done in parallel! This speed-up was crucial to the ability to train against huge amounts of training data - trillions of tokens.

• Also, the Feed-forward MLPs are isolated - i.e. there are n separated MLPs, not one big one, and their parameters are all the same
• Think of the transform block as a set independently computed “rows”, where there is one “row” per input token/embedding.
  - Each “row” has the same trained parameters in it, including the layer norm
  - Similar to a CNN’s kernels - like how the kernel is used all over the image, attention, MLP, norm are applied the same on different input token rows

• Now, I mentioned that attention is “said” to be working as described, but to me this only really makes sense on the very first transformer block T0, and even there, just a the beginning attention
• Everything after that is the typical black-box of neural networks - the feed forward MLP for example
• THEN, the next transformer block mixes up all the input embeddings again, through a different set of learned Matrices on that layer, then MLP and so on through all layers.
• To me the key to the transformer is really how wide it is - it keeps the information flowing from the input embeddings flowing all the way to the end, versus RNNs which “pinched” that information after every word input

Notes on Assignment 3
• Code for Transformer is too complex to write from scratch
• So, A3 gives you Karpathy’s mingpt - a well written, simpler GPT style transformer
• You’ll have to read code and try to understand it
• we will use mingpt “nano” which has these parameters:
  - # Transformer blocks = 3 = n_layer
  - # Heads, h = 3 = n_head
  - Embedding dimension, d = 48 = n_embed

• The assignment is to train this transformer on a small, then large corpus (same as ones from A1)
• Re-use the language model as a sentiment classifier, after fine-tuning it
• Learn to use the Huggingface model hub/code to fine-tune GPT-2

• Missing from this lecture: Positional Embeddings - the answer to the question “how does the transformer know the order of the input words?”