# ECE 1786 Lecture #5

**Last Day:** Project + Introduction to Transformers

**Work-in-flight:** Assignment #3: Understanding, Training & Fine Tuning Transformers for Classification
A2 due today
- team forming due today
- approval-in-principle due Oct 27

**Today:** The Core Mechanisms of Transformers & Assignment 3 (nothing)

or done

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

From: The smooth blue lake became choppy in the wind

| | To: input sequence of words | label |
|---|---|---|
| e.g. Training Example 3: | The smooth blue | lake |
| " " 4: | The smooth blue lake | became |
| 5: | The smooth blue lake became | choppy |

- lot's of training data! everything ever written

- here is the global structure of a Transformer configured in training mode:

INPUT WORD vectors $X_0, X_1, \ldots, X_{n-1}$ (each of dimension $d$; $n \times d$) → Transformer Block $T_0$ → ($n \times d$) → $T_1$ → $\cdots$ → $T_{N-1}$ → ($n \times d$) → Language "Head" Fully Connected MLP → $P(w_0)$, $P(w_1)$, $\ldots$, $P(w_{M-1})$

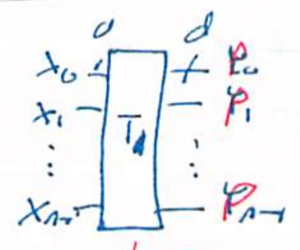
$M$ = size of vocabulary

→ 512, 1024..

- even though $n$ words/tokens always goes in to predictor, the ones after the final input length $\leq n$ are ignored ⇒ "Causal" language model
- done as part of the batch set-up

in, something but fewer in, as above.

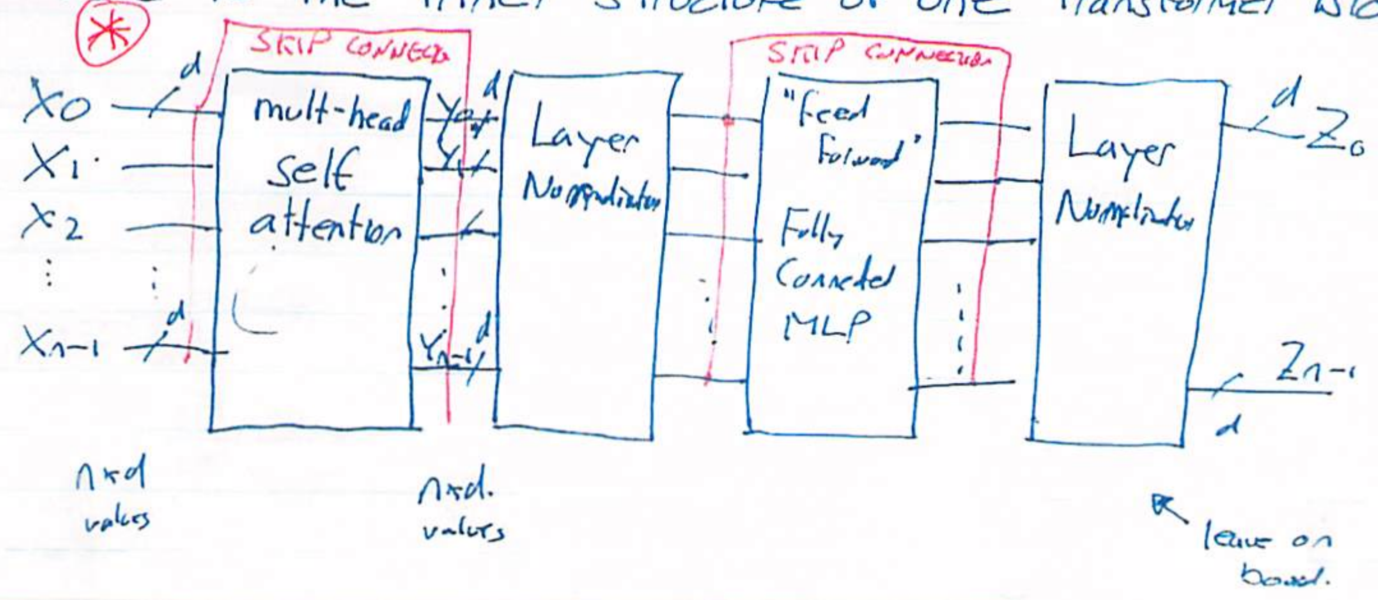
- Now, here is what is in each transformer block $T_i$:
- recall it is the same size in as out.




skeptical aside: we could ignore the inner structure of $T_i$ and just say "it is a big network with many parameters"
- it may be, to my mind, the underlying truth, & maybe a large MLP or CNN could do this same thing. (just bigger is better. & it does work.
- it may be that the particular $T_i$ structure is better.
- will present the structure & understanding per Jurafsky 9.2.
- however, the explanations only work for block $T_0$, to me.

Warning: This is complex!

- Here is the inner structure of one Transformer Block



Here is the core intuition given of what Transformer Attention Block is doing

- the input word embeddings are transformed from their initial, <u>very general</u> meanings (across all uses/contexts of the words) to something <u>more specific to the context</u> → i.e the other words in sequence (makes some sense.)
  ⇒ they call these "contextual embeddings"

- 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 (*), consider the output $y_i$, which we say is a transformation of word $x_i$ the input word.

e.g. $x_0$ $x_1$ $x_2$ [$x_3$ bank] $x_4$ $x_5$
He emptied his **bank** account. e.g. $i = 3$

- self attention asks the question: how similar is each word to all preceeding words and itself! [See Jurafsky Section 9.7]

e.g. how similar is $x_3$ (bank) to $x_0$ (He) ?
$x_3$ (bank) to $x_1$ (Emptied) ?
$x_3$ (bank) to $x_2$ (his) ?
$x_3$ (bank) to $x_3$ (bank) ?

- how have we computed a single number that says how similar/related two words are?
  ⟹ use dot product of word embeddings

Compute $x_0 \cdot x_3$ Can Define
$x_1 \cdot x_3$ $score(x_i, x_j) = x_i \cdot x_j$ for all $j \leq i$
$x_2 \cdot x_3$ ↳ but need to normalize
$x_3 \cdot x_3$. it, so ⟹

- do this for every word
- when computing the relative similarity of $x_i$ to everything that comes before, we calculate

$\alpha_{ij}$

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

$$= \frac{\exp(score(x_i, x_j))}{\sum_{k=0}^{i} \exp(score(x_i, x_k))} \quad \forall j \leq i$$

only previous & current word

- this score, $\alpha_{ij}$, gives the relative importance of $x_j$ to $x_i$, and we use it to compute the new embedding, $y_i$ that combines different proportions of the $x_j$ like so: → including the original $x_i$

$$y_i = \sum_{j \le i} \alpha_{ij} x_j \quad (**)$$

→ we are adding a fraction of the meaning of those other words into the original embedding.
→ this is how "bank" gets more "river" in it
→ "contextual embeddings" – makes sense on 1st layer

[redacted] → nothing is being learned; ML trick is to insert parameters to "enhance" this transformation.

Now, notice that there are no learned parameters so far
- also notice that $x_i$ gets used in 3 ways!
  1. as the focus $x_i$ in score$(x_i, x_j)$ — "query"
  2. as the "searched" $x_j$ in score$(x_i, x_j)$ — looking backwards — "key"
  3. to compute $y_i$ in (**) above — "value"

- for all 3 cases we will insert learned parameters to modify/project these three uses as follows, e.g. for 1

$$\text{let } q_i = W^Q x_i \quad \text{where } W^Q = \begin{bmatrix} w_{00} & \cdots & w_{0,d-1} \\ \vdots & & \vdots \\ w_{d-1,0} & \cdots & w_{d-1,d-1} \end{bmatrix} d \quad \begin{bmatrix} x_i^0 \\ \vdots \\ x_i^{d-1} \end{bmatrix} d.$$

(learned params ↓ d)

→ – this is a ML trick
– think of $W^Q$ like a Conv Kernel → something learned across the training data
– we'll (later have multiple)

- if you multiply this out you'll see that $q_i$ is the same vector size as $x_i = d$
- but it has been projected/transformed by $W^Q$.
  - the $W_{pq}$ are learned params through gradient descent

- Similarly, there are two other learned 2d ~~matr~~ matrices $W^K$ & $W^V$ to get

$$k_i = W^K x_i$$

$$v_i = W^V x_i$$

together $q_i$, $k_i$ & $v_i$ "look" for patterns in the input & express the output $y_i$ based on this — again like 1 big kernel in a CNN.

- So the overall computation becomes

For each input embedding $x_i$, compute:

$$\alpha_{ij} = \text{Score}(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d}} \quad \text{for all } j \le i$$

→ scaled to keep ~~[illegible]~~ sizes under control

$$y_i = \sum_{j \le i} \alpha_{ij} v_j$$

↑ the output embedding correspond to $x_i$

- the learned matrices $W^Q$, $W^K$, $W^V$ learn which values of the $x_i$ are important in these 3 roles and presumably emphasize them.

INTUITION:
- When the inputs look like this: $W^Q$ & $W^K$ then focus the output on that: $W^V$ — maybe.

Jurafsky: "Transformers allow us to create a more sophisticated way (using the learned weights) of representing how individual words can contribute to the representation of longer inputs"

- one set of weights $W^K, W^Q, W^V$ can be thought of looking for one set of relationships
  → a single self-"attention "head"

→ could be many such relationships. that will help NN succeed. | like conv kernels

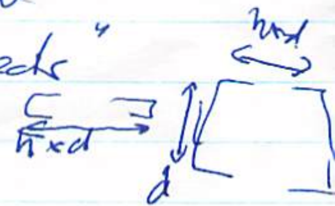
→ so rather than just one set of $W^K, W^Q, W^V$ we use several
  ⇒ called "Multi-head Self-Attention"
  ⇒ get $W_i^K \; W_i^Q \; W_i^V$ $\quad 1 \le i \le h$ ← # heads.

  constant in code

- in Assignment 3, use mingpt "nano" * Transformer Blocks = 3 = n-layer
  * Heads h = 3 = n_head
  d = 48 = n-embd

- Every head, though, increases the size of the output → don't want larger output than input

- So also train another weight matrix $W^O$ (learned parameter) that is shape $hd \times d$ which "projects" the hd output back down to d.




---

- the other parts of the transformer block are more common → see (*) on 5-3.
  ① Skip connections → "insurance policy against failed optimization" x → [ ] → (+) → [ ]

  ② Normalization, Dropout, weight decay

# Assignment A3

↗ Show Karpathy's Battleship v. Speedboat.

- the code for a Transformer is too complex to write
- instead we use Karpathy's mingpt
- still quite complex! Configurable to large Transformers
- nicely written software, but will require careful reading, thinking & looking up PyTorch funct.
- we have turned it into a language model + given you the code.
  ⇒ asked you questions about that code, to help you/prod you to understand it.
- train it with:

① SmallSimpleCorpus.txt from A1.
  - easier to see how Language Model works
  - look at predictions.

② LargeCorpus.txt
  → to create pretrained model

  ⇒ fine tune a classifier to do sentiment analysis

③ Use Huggingface → where the real models live, to also do sentiment analysis

- partially released tonight @ 9pm
- Fully released later this week.