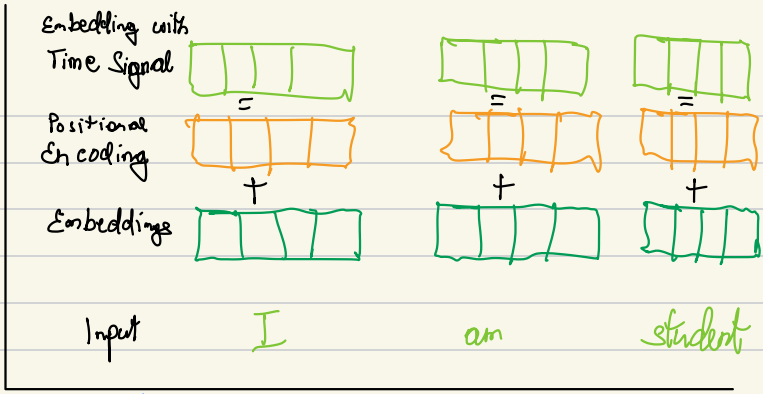


* We know that in a transformer, we have input embeddings and since, the tokens are processed in parallel, we add a positional encoding to give more information about the position.
* Positional Encoding is a vector of the same size as the input.
* Positional encoding is added to the input embedding to finally generate the time-aware embeddings.

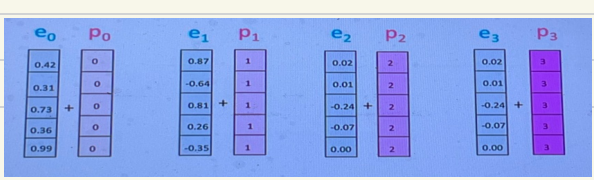
Q) Why sum, why not concat or any other operation?
A) No concrete reason, but sum will keep the space needed for storage low (4 in this case), whereas concatenation will need more space (8 in this case)

* The new x_1, x_2 & x_3 are position aware embeddings which will be fed to the 1st layer of the encoder or decoder.
* Also, positional encoding information is added to only the input before the 1st layer / transformer block. It is not added in subsequent stacked encoder / decoder blocks.

Types of positional encodings

Option 1

* Encode each position by its index



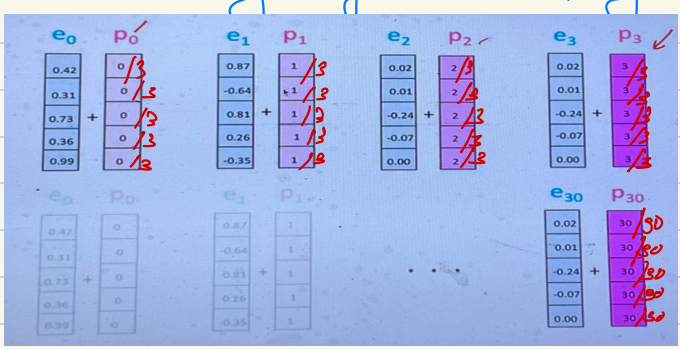
Issue

↳ Size of a sentence/input can vary. So, no pre-defined max value.

↳ If 2 sentences of diff. lengths (say 4 & 31) are encoded, the last word in both gets encoded with very different vectors (one with 3's & other with 30's) even though both represent the last position. Also, the model might misinterpret some information (like it might think that the 3rd index posit^n in the 31 length vector is similar to the last posit^n in the 4 length vector because both are encoded with 3's)

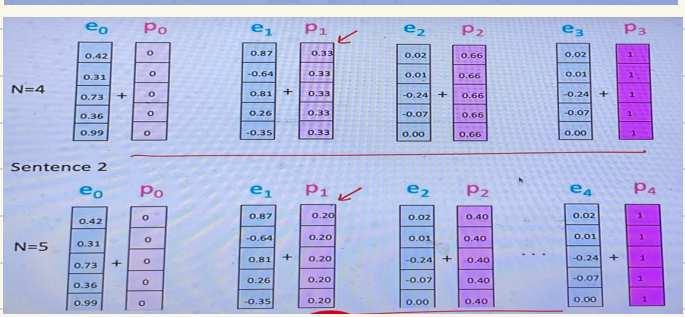
How to solve this?

- Simple solut^n is to normalize the positional encodings i.e., divide each positional encoding by the max length of that sequence
- So, the last positional encoding becomes 1 for all input sequences



Issues

↳ There is a dependency on the max length of the sequence. So, the same posit^n gets encoded with different values as shown in the figure beside



So, we want something that can generalise to arbitrary sequence lengths. We also may want to make attending to relative positions (eg: tokens in a local window to the current token) easier.
Also, the distance between two positions should be consistent with variable length inputs.

Intuitive Example: Binary Encoding

* Encode each position using its binary representation.
* This guarantees all positions are unique but adding binary values to a fractional embedding is not very meaningful.
* Similar problem as before i.e., we don't know the max length of the sequence.

0:	0	0	0	0	8:	1	0	0	0
1:	0	0	0	1	9:	1	0	0	1
2:	0	0	1	0	10:	1	0	1	0
3:	0	0	1	1	11:	1	0	1	1
4:	0	1	0	0	12:	1	1	0	0
5:	0	1	0	1	13:	1	1	0	1
6:	0	1	1	0	14:	1	1	1	0
7:	0	1	1	1	15:	1	1	1	1

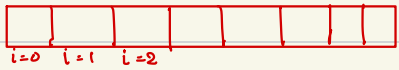
Even though we can't use this, we can get some motivation from the way binary representations work.

If we observe carefully, the least significant bit (LSB) of binary representation flips with a frequency of 1 (i.e., it changes for every new position), 2nd bit flips with a frequency of 2 (i.e., changes after every 2 numbers), 3rd bit flips with a frequency of 4 (i.e., after every 4 numbers) & so on.

It shows a kind of periodicity. And another way of representing periodic signals is sinusoidal representation.

Transformers Positional Encoding

*) So, for a positional encoding vector with each element in the vector denoted by an index i, the positional encoding for any position pos and any element within the vector can be defined as



$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

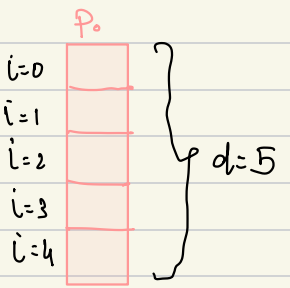
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

pos = posit^n of word in sequence

i = element index/dimension of the positional embedding vector

$d_{model} = 512$

↳ Size of positional embedding (& the word embedding)



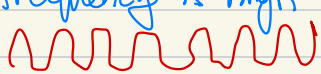
$PE(0,0) = \sin(0/10000^{2\times0/512})$

$PE(0,1) = \cos(0/10000^{2\times1/512})$

$\sin\left(\frac{pos}{10000^{2i/d}}\right)$ → This determines the frequency of the sine wave

d → fixed

for small i → frequency is high



for large i → frequency is low

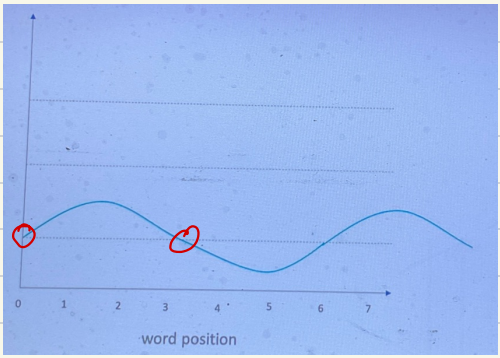
↳ (initial indices like i=0,1,2,...)

So, the upper part of vector is captured with high frequency sine wave & lower part of the vector is captured with

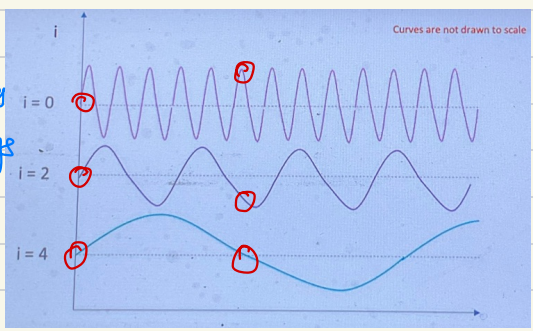
low frequency sine wave.

Q) Why do we need 'i'? Why not make it a funct^{n} of just the position?

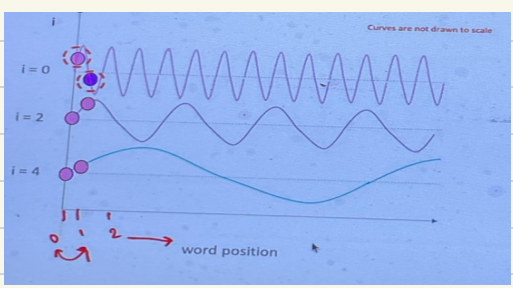
A) If we do this, the values will start repeating. In the figure alongside. The 0th & 6th word position will have same positional encoding. Similarly, 1 & 7 and so on.



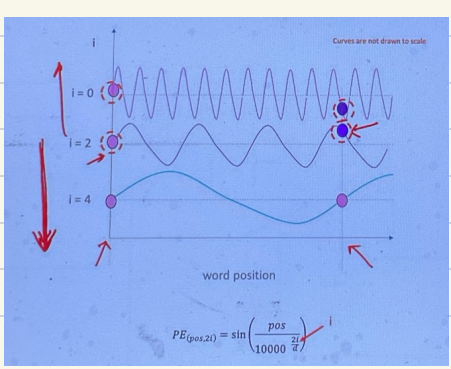
With i, we will have different sine waves with different frequencies as components of the sinusoidal embeddings and thus even if they match for one component, they will be different for others



Also, if we consider 2 nearby positions, we find that their low frequency bands are quite similar They will be seperated by the high frequency signals.

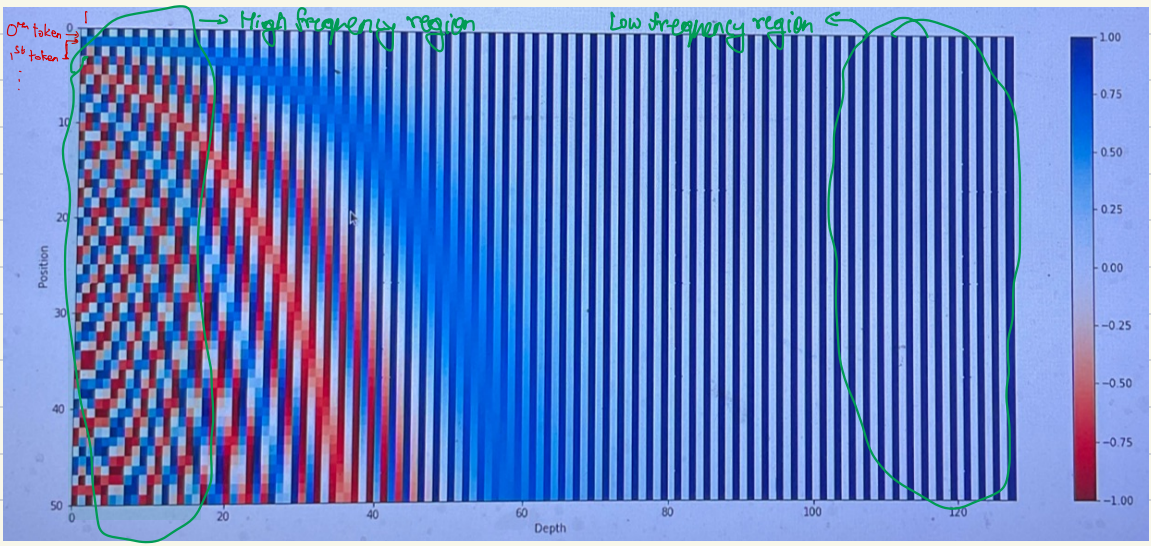


For the points which are far apart, the low frequency signals itself are capable enough of seperating them. The high frequency signals are also, obviously able to seperate them.

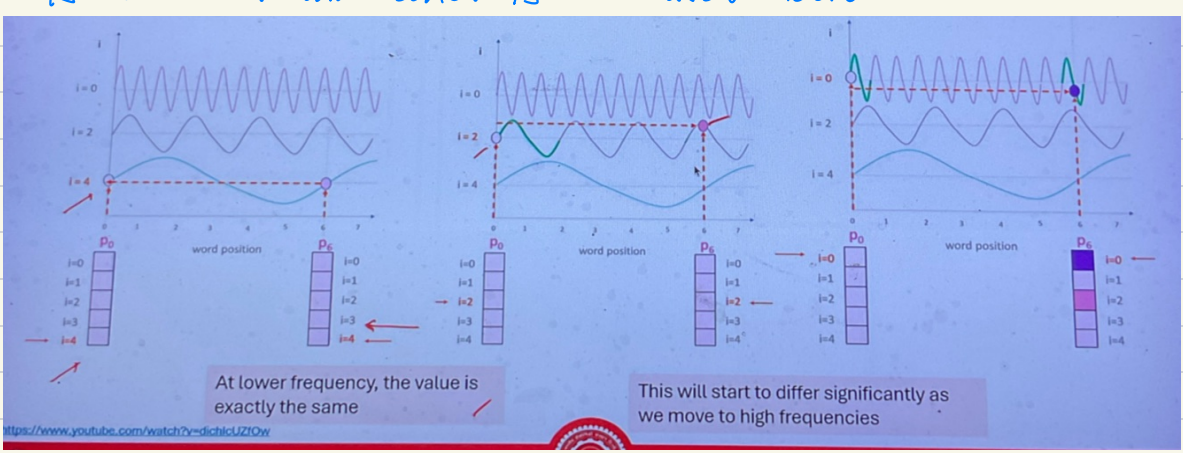


Position Vector Visualisation

(each row is the positional encoding of a 50-word sentence)
(each column represents a dimension of the vector i.e 'i' value)



* If we see, the 0th & 1st token, we see that their colors are different in the initial columns but more uniform as we move forward. And the initial columns are the high frequency regions so this is consistent with what is discussed above.



Example

for eg: for word w at pos $\in [0, L-1]$ in the input sequence $w = (w_0, \ldots, w_{L-1})$ with 4-dimensional embedding e_w & $d_{model} = 4$, the operation would be:

$$e'_w = e_w + \left[\sin\left(\frac{pos}{10000^0}\right), \cos\left(\frac{pos}{10000^0}\right), \sin\left(\frac{pos}{10000^{2/4}}\right), \cos\left(\frac{pos}{10000^{2/4}}\right)\right]$$

$$= e_w + \left[\sin(pos) + \cos(pos) + \sin\left(\frac{pos}{100}\right) + \cos\left(\frac{pos}{100}\right)\right]$$

Q) Does this address all the issues & satisfy all the properties we wanted?

↳ It is independent of sequence length

↳ Same position has same embedding in different sequences.

↳ 2 nearby words will be differenciated by high frequency regions whereas words that are far away will be differenciated by both high & low frequency regions. So, if we take a dot product we will see that nearby words have high similarity scores (for positional embeddings) whereas faraway words will not.

* Despite its intuitive flaws, many models use learned positional embeddings these days. Learned positional embeddings are shared across sequences. (They do not generalize well to longer sequences however)

General Properties of Positional Embeddings

We define the function $\phi(\cdot,\cdot)$ to measure the proximity between positional embeddings.

- Monotonicity: The proximity of position embeddings positions decreases when positions are further apart

$$\forall x, m, n \in \mathbb{N}: m > n \iff \phi(\vec{x}, \vec{x+m}) < \phi(\vec{x}, \vec{x+n})$$

- Translation Invariance: The proximity of embedded positions are translation invariant i.e., it shouldn't change regardless of the starting position (for the same distance)

$$\forall x_1, x_2 \dots x_n, m \in \mathbb{N}: \phi(\vec{x_1}, \vec{x_1+m}) = \phi(\vec{x_2}, \vec{x_2+m}) \dots \phi(\vec{x_n}, \vec{x_n+m})$$

In simple words, the relative distance b/w two positions should be the same, no matter where they are in the sequence.

- Symmetry: The proximity of embedded positions is symmetric

$$\forall x, y \in \mathbb{N}: \phi(\vec{x}, \vec{y}) = \phi(\vec{y}, \vec{x})$$

Rotary Positional Encoding (RoPE) – Precursors

* Published in 2023
* Used by models like PaLM, GPT-Neo, GPT-J, Llama 1 & 2.

Absolute Positional Encodings

eg → Learned from data
↳ Sinusoidal function (like original Transformer)

- Focus is on encoding absolute positions

$E_N = \{x_i\}_{i=1}^{N}$

$q_m = f_q(x_m, m)$
$k_n = f_k(x_n, n)$
$v_n = f_v(x_n, n)$

$$a_{m,n} = \frac{\exp\left(\frac{q_m^T k_n}{\sqrt{d}}\right)}{\sum_{j=1}^{N} \exp\left(\frac{q_m^T k_j}{\sqrt{d}}\right)}$$

$$o_m = \sum_{n=1}^{N} a_{m,n} v_n$$

- w_i are tokens and x_i are embeddings

$f_{t:t \in \{q,k,v\}}(x_i, i) := W_{t:t \in \{q,k,v\}}(x_i + p_i)$

$$\begin{cases} p_{i,2t} = \sin\left(k / 10000^{2t/d}\right) \\ p_{i,2t+1} = \cos\left(k / 10000^{2t/d}\right) \end{cases}$$

- Generate p_i using the sinusoidal function
  - Sinusoidal functions provide continuity between close positions
  - This also allows for the model to scale to virtually unlimited input sequence length.

Relative Positional Encoding

*) Here we don't care about absolute posit^n but only look at the relative posit^n b/w 2 tokens.

eg: The dog chased the pig
Once upon a time, the pig chased the dog.

*) Here is an example of one of the Relative Position Encodings before RoPE.

* clip b/w r_{max} & r_{min} if it is arbitrarily large or small.

** Authors felt they didn't need to address p_m because they had already the relative posit° b/w m & n with $\tilde{p}_{m-n}$. So, they substituted p_m with some bias term.

↳ query is not a funct° of posit°

$$f_q(x_m) := W_q x_m$$
$$f_k(x_n, n) := W_k (x_n + \tilde{p}_r^k)$$
$$f_v(x_n, n) := W_v (x_n + \tilde{p}_r^v)$$

} these $\tilde{p}$ positional embeddings & indicates relative distance b/w m & n

- $\tilde{p}_r^k, \tilde{p}_r^v \in \mathbb{R}^d$ are trainable relative position embeddings
- $r = clip(m-n, r_{min}, r_{max})$* is the relative distance between positions m & n.
  - Relative position info is not useful beyond a certain distance

Transformer XL

$$f_{t:t\in\{q,k,v\}}(x_i, i) := W_{t:t\in\{q,k,v\}}(x_i + p_i)$$

↳ original transformer

- Decompose $q_m^T k_n$ → $W_k(q_n + p_n)$ & $W_q(q_m + p_m)$ → q & k dot product

$$q_m^T k_n = \underbrace{x_m^T W_q^T W_k x_n}_{\text{has no positional info}} + \underbrace{x_m^T W_q^T W_k p_n}_{\text{has position info of key}} + \underbrace{p_m^T W_q^T W_k x_n}_{\text{has position info of query}} + \underbrace{p_m^T W_q^T W_k p_n}_{\text{only position info}}$$

- Replace absolute position embedding p_n with relative $\tilde{p}_{m-n}$
- Replace absolute position embedding p_m with 2 trainable vectors u & v independent of query positions.**
- W_k is distinguished for content-based & locat° based key vectors x_n & p_n, denoted as W_k & $\tilde{W}_k$

$$q_m^T k_n = x_m^T W_q^T W_k x_n + x_m^T W_q^T \tilde{W}_k \tilde{p}_{m-n} + u^T W_q^T W_k x_n + v^T W_q^T \tilde{W}_k \tilde{p}_{m-n}$$

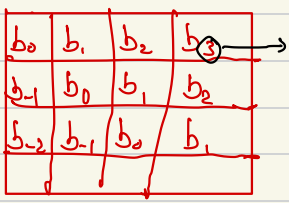
T5

- Uses a very simplified relative position embedding

$$q_m^T k_n = x_m^T W_q^T W_k x_n + b_{i,j}$$

→ combined all the terms with positional components and made it a learnable parameter

- $b_{i,j}$ is a trainable bias



- In another formulation, the paper suggested that learning all terms can be difficult so we keep 2 and learn the others

$$q_m^T k_n = x_m^T W_q^T W_k x_n + p_m^T U_q^T U_k p_n + b_{i,j}$$

DeBERTa

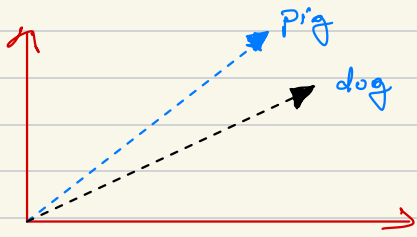
- Absolute position embeddings p_m & p_n are replaced with the relative position embeddings $\tilde{p}_{m-n}$

$$q_m^T k_n = x_m^T W_q^T W_k x_n + x_m^T W_q^T W_k \tilde{p}_{m-n} + \tilde{p}_{m-n}^T W_q^T W_k x_n$$

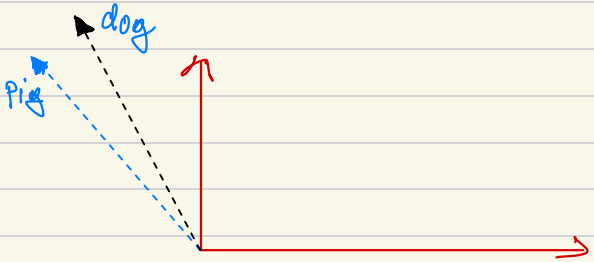
Combining both Relative & Absolute Positions

RoPE - 1st paper to leverage both relative & absolute position embedding.

Idea: The dog chased the pig
The pig chased the dog



The relative position of dog & pig is the same in both the above sentences & can be captured using the angle between the 2 vectors.



If the absolute position of the 2 vectors changes, keeping the relative position the same the embeddings will just rotate

eg: Once upon a time, the pig chased the dog.

Rotation takes care of the change in relative position & the angle takes care of the relative position.

In the RoPE paper, the authors mapped all the embeddings to the complex space.

Rotary Position Embedding (RoPE)

* Encodes absolute positions with a rotation matrix. (A matrix multiplication is just a linear transformation. A rotation matrix is a special matrix that doesn't change the magnitude of a vector but only rotates it)

eg:
$$\begin{bmatrix} \cos 90 & -\sin 90 \\ \sin 90 & \cos 90 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$
$$= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

→ Rotated the vector by 90° anticlockwise

* Incorporates explicit relative pos dependency in self-attention formulation i.e., during query & key dot product
* Requirement: $q_m^T k_n$ be a function (g) of only word embeddings x_m, x_n and their relative position $m-n$

$$\langle f_q(x_m, m), f_k(x_n, n) \rangle = g(x_m, x_n, m-n)$$

$f_q(x_m, m)$: query embedding, query position; $f_k(x_n, n)$: key embedding, key position

→ only a funct^n of x_m, x_n & $m-n$

Meaning – f_q is a funct^n of x_m & position 'm' and f_k is a funct^n of x_n & posit^n 'n' but their dot product should be such that it is a funct^n of x_m, x_n & only the relative position $m-n$

* So, what could be a good choice of f_q & f_k?

* For dimension $d=2$, a solution is

$$f_q(x_m, m) = (W_q x_m) e^{im\theta}$$
$$f_k(x_n, n) = (W_k x_n) e^{in\theta}$$
$$g(x_m, x_n, m-n) = \mathrm{Re}\left[(W_q x_m)(W_k x_n)^* e^{i(m-n)\theta}\right]$$

→ new term introduced for position; m,n are the positions; θ is the rotation value

- $\mathrm{Re}[\cdot]$ – real part of complex number
- $(W_k x_n)^*$ – complex conjugate of $W_k x_n$
- $\theta \in \mathbb{R}$ – preset non-zero constant

Thus, these choices of functions f_q & f_k satisfy the properties required as mentioned above

Dot product of 2 complex numbers in a polar coordinate

$\langle z_i, z_j \rangle = z_i z_j^*$ → complex conjugate

$r_i e^{\theta_i}$ $r_j e^{\theta_j}$

$= r_i r_j e^{(\theta_i - \theta_j)}$

need more clarity!! Is there no conjugate here because r_j is real so it's the same thing? So, also is $(W_k x_n)^*$ the same as $W_k x_n$ because it is also real?

Complex conjugate of a number is the number with its imaginary part negation

Rotation matrix ← → Transformation matrix → embedding

$$f_{\{q,k\}}(x_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} W^{(11)}_{\{q,k\}} & W^{(12)}_{\{q,k\}} \\ W^{(21)}_{\{q,k\}} & W^{(22)}_{\{q,k\}} \end{pmatrix} \begin{bmatrix} x^{(1)}_m \\ x^{(e)}_m \end{bmatrix}$$

Why?

Because Euler's formula: $e^{im\theta} = \cos(m\theta) + i\sin(m\theta)$

Matrix form of $e^{im\theta}$: $\begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix}$

$$\text{Re}[e^{im\theta}] = \cos(m\theta); \quad \text{Im}[e^{im\theta}] = \sin(m\theta)$$

* So, essentially, we are applying a rotation matrix with a transformation
* This matrix captures the rotation by $m\theta$
* Rotary Position embedding
  ↳ Rotate the affine transformed word embedding vector by amount of angle multiples of its position index

RoPE: General form

* Generalising from 2D to d-dimensions
  - Divide the d-dimension space into d/2 subspaces

$$f_{\{q,k\}}(x_m, m) = R^d_{\Theta,m} W_{\{q,k\}} x_m$$

$$R^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

→ what is essentially happening here?
We take every 2 elements of the vector & rotating

$\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, \dots, d/2]\}$ → similar to original transformer

Q) Are we learning something here?
A) No, everything is given Θ, m & n.

* Applying RoPE to self-attention

$$q_m^T k_n = (R_\Theta^d W_q x_m)^T (R_\Theta^d W_k x_n) = x_m^T W_q R_{\Theta, n-m}^d W_k x_n$$

$$R_{\Theta, n-m}^d = (R_{\Theta, m}^d)^T R_{\Theta, n}^d$$

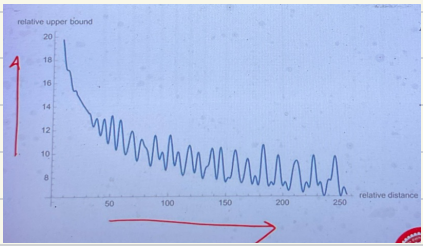
↳ funct^n of $n-m$
Thus, relative posit^n is preserved

* In contrast to earlier position embedding methods, RoPE is multiplicative → $e^{im\theta}$ is multiplied not added
* RoPE naturally incorporates relative pos info through rotation matrix product instead of altering terms in the expanded formulation of additive position encoding when applied with self-attention.

* RoPE
  - ↳ Represents token embeddings as complex numbers.
  - ↳ Represents their position as pure rotations
  - ↳ If we shift both the query & key by the same amount, changing absolute position but not relative position, this will lead both representations to be additionally rotated in the same manner, thus the angle between them will remain unchanged and thus the dot product will also remain unchanged.

Properties of RoPE

*) Long Term Decay
- Inner product decays when the relative position increase
- A pair of tokens with long relative distance should have less connection



*) Even though the rotation matrix looks very complex, most entries are 0. There is a computationally efficient way of realising the rotary matrix multiplication by segregating the cos & sin parts.

$$f_{\{q,k\}}(x_m, m) = R^d_{\Theta,m} W_{\{q,k\}} x_m$$

$$R^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

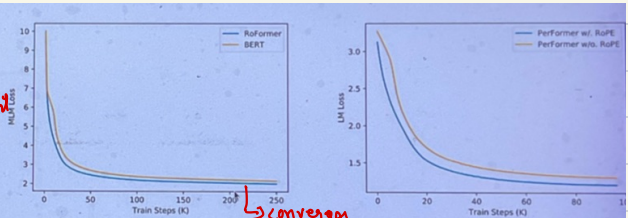
$$R^d_{\Theta,m} x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix}$$

RoPE Performance

Model	BLEU
Transformer-base Vaswani et al. [2017]	27.3
RoFormer	27.5

slight improvement

WMT 2014 English-to-German translation task



convergence faster

Language modeling pre-training. Left: training loss. Right: training loss for PerFormer with and without RoPE.

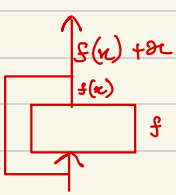
Table 2: Comparing RoFormer and BERT by fine tuning on downstream GLEU tasks.

Model	MRPC	SST-2	QNLI	STS-B	QQP	MNLI(m/mm)
BERTDevlin et al. [2019]	88.9	93.5	90.5	85.8	71.2	84.6/83.4
RoFormer	**89.5**	90.7	88.0	**87.0**	**86.4**	80.2/79.8

significant improvement

Residual Connections

*) A shortcut direct connection from the



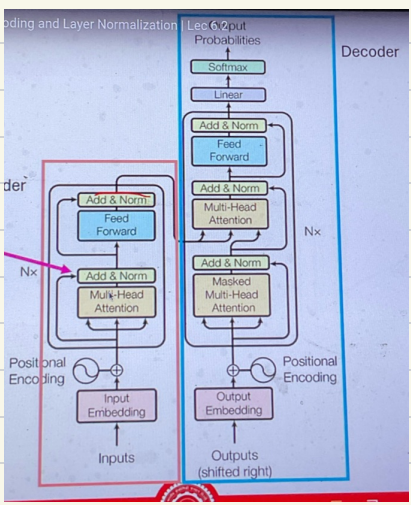
input to the output which directly adds the input to the output ($f(x) + x$), which is then fed to the next layer.

*) Helps address the vanishing gradient problem.

*) 2 residual connections in a transformer encoder block -
1) Across self-attention layer 2) Across the FFN

*) 3 residual connections in a transformer decoder block -
1) Across self-attention layer 2) Across cross-attention layer
3) Across the FFN

Add & Norm Layer



*) Final component of the transformer architecture

*) Add is the addition of the residual connection

*) Norm is layer normalisation.

Normalisation

*) Normalisation is not specific to transformers - it is used in almost all neural network models after each layer.

The patterns that we hope to maintain in a neural network (eg: values ranging b/w 0 & 1, values following a certain distribution, etc.) get destroyed because of the complex operations within a network. Normalisatn helps to deal with these issues.

Benefits of Normalization

- Stabilize Training: Stabilizes & speeds up the training

process by keeping the inputs to each layer in a similar range, ensuring that gradients don't explode or vanish during backpropagation.

- Accelerate convergence: Normalized inputs lead to faster convergence during training as features scaled to a similar range are learned more efficiently by the network.

- Prevent Overfitting: Techniques like dropout & batch normalisation also help prevent overfitting. Dropout randomly drops units during training, which forces the network to learn more robust features. Batch norm introduces a slight regularisation effect by adding noise to the inputs.

- Improve Gradient Flow: By normalizing the inputs, the gradients are more uniform and flow better through the network, which can help in training deeper networks.

Types of Normalization

* Batch normalisation
* Layer "
* Instance "
* Weight "
* Group "

Normalisation & Standardisation example

* Student loan eligibility with age & tuition as input features
  - the 2 values are on completely different scales
    - ↳ age will have a median value in the range 18-25
    - ↳ tuition could take values b/w 20k-50k for an academic year.

Normalisation works by mapping all values of a feature to be in the range $[0, 1]$ using the transformation

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$ where x is in the range $[x_{min}, x_{max}]$

Standardisation transforms the input values such that they follow a distribution with zero mean and unit variance

$$x_{std} = \frac{x - \mu}{\sigma}$$ $\mu \rightarrow$ mean $\sigma \rightarrow$ std deviation

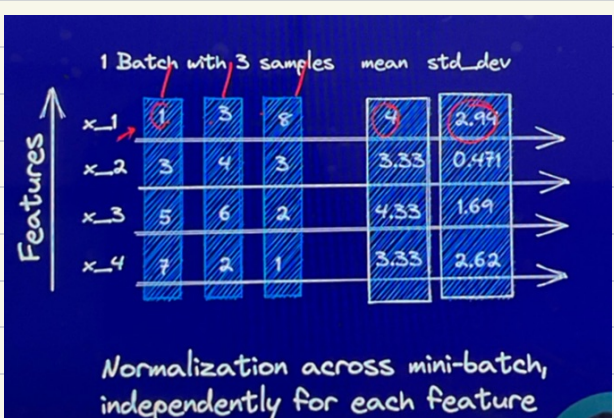
↳ done feature-wise

In practice, standardisation is also referred to as normalisation

Batch Normalisation

* Normalisation is performed across each batch, i.e., send an input batch, stack the outputs & perform normalisation

The input distribution at a particular layer keeps changing across batches.
The paper 'Accelerate Deep Network Training by Reducing Internal Covariate Shift' refers to this change in distribution of the input to a particular layer across batches as internal covariate shift



- The example here shows a batch with 3 instances. The column represents a feature vector
- We standardize across all the examples in the training batch for each individual feature i.e., calculate μ and σ for each feature and do $\frac{x - \mu}{\sigma}$ for each element for that feature in the batch.

* forcing all the pre-activations to be zero and unit standard deviation across all batches can be too restrictive.
* It may be the case that the fluctuant distributions are necessary for the network to learn certain classes better

$$\mu_b = \frac{1}{B}\sum_{i=1}^{B} x_i \quad (1)$$

$$\sigma_b^2 = \frac{1}{B}\sum_{i=1}^{B}(x_i - \mu_b)^2 \quad (2)$$

$$\hat{x}_i = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2}} \quad (3)$$

$$\text{or } \hat{x}_i = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}} \quad (3)$$

Adding ϵ helps when σ_b^2 is small

$$y_i = BN(x_i) = \gamma \cdot x_i + \beta \quad (4)$$

γ, β: Trainable parameters

Limitations of Batch Norm

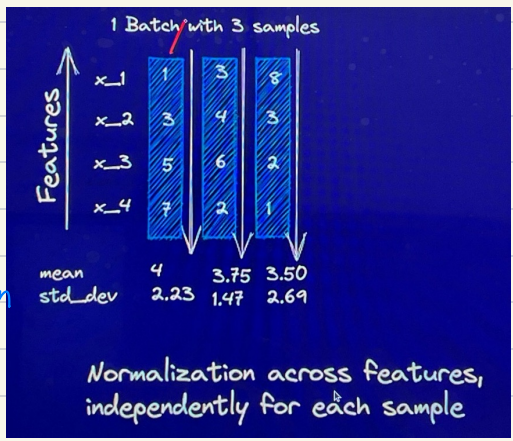
- Since we use batch statistics, when the batch size is small, sample mean and sample standard deviation are not representative enough of the actual distribution and the network cannot learn anything meaningful.
- Since it depends on batch statistics, batch norm is less suited for sequence models because in sequence models we may have sequences of potentially different lengths and smaller batch sizes corresponding to longer sequences.

Layer Normalisation

* All neurons in a particular layer effectively have the same distribution across all features for a given input

if each input has d features, it's a d-dimensional vector. If there are B elements in a batch, the normalisation is done along the length of the d-dimensional vector & not across the batch of size B.

* Normalizing across features for each input to a specific layer removes the dependence on batches.
* This makes layer norm well suited for sequence models such as transformers & RNNs.



Eg: Here we calculate the mean and std. deviation for each individual input sample across the features i.e., columnwise. Then we perform the standardisation as usual with $\frac{x-\mu}{\sigma}$ but along the column this time.

$$\mu_\ell = \frac{1}{d}\sum_{i=1}^{d} x_i \quad (1)$$

$$\sigma_\ell^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu_\ell)^2 \quad (2)$$

$$\hat{x}_i = \frac{x_i - \mu_\ell}{\sqrt{\sigma_\ell^2}} \quad (3) \quad \text{or} \quad \hat{x}_i = \frac{x_i - \mu_\ell}{\sqrt{\sigma_\ell^2 + \epsilon}} \quad (3)$$

Adding the ϵ helps when σ_ℓ^2 is small

$$y_i = LN(x_i) = \gamma \cdot x_i + \beta \quad (4)$$

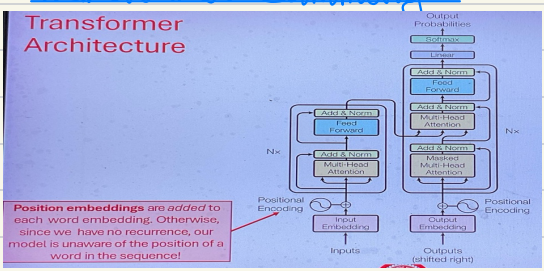
γ, β: Trainable parameters

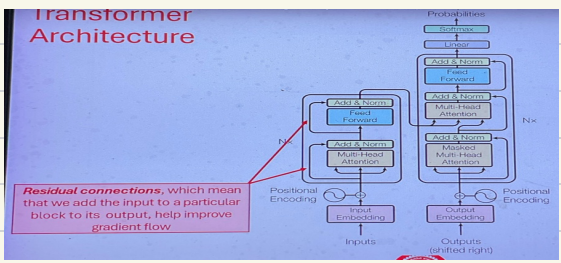
Batch vs Layer Norm

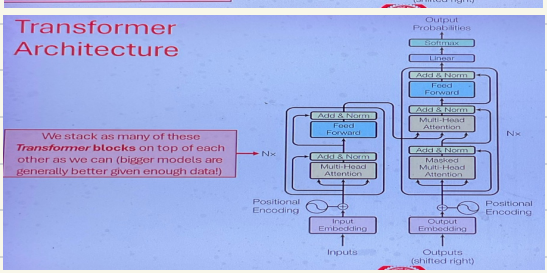
* Batch norm normalizes each feature across mini-batch. Layer norm normalizes each of the inputs in a batch across features.

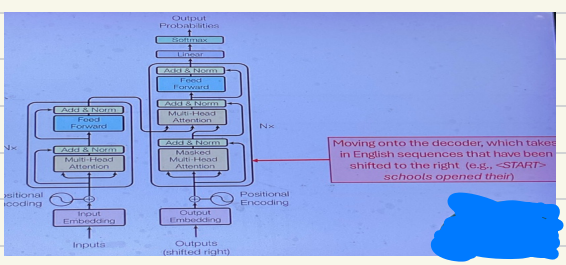
* Batch norm is dependent on batch size & not effective for small batch sizes. Layer norm is independent of batch size & can be applied to smaller batches as well.
* Batch norm requires different processing at training & inference times. Since, layer norm is done along the length of the input to a specific layer, the same set of operations can be used for training & inference times.

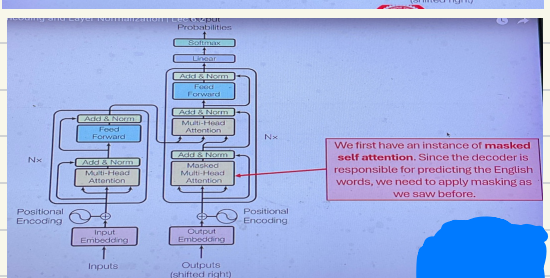
Transformer Summary

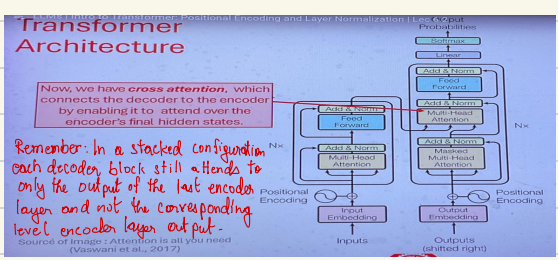


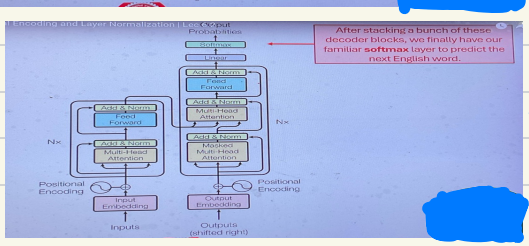












How is the model trained?

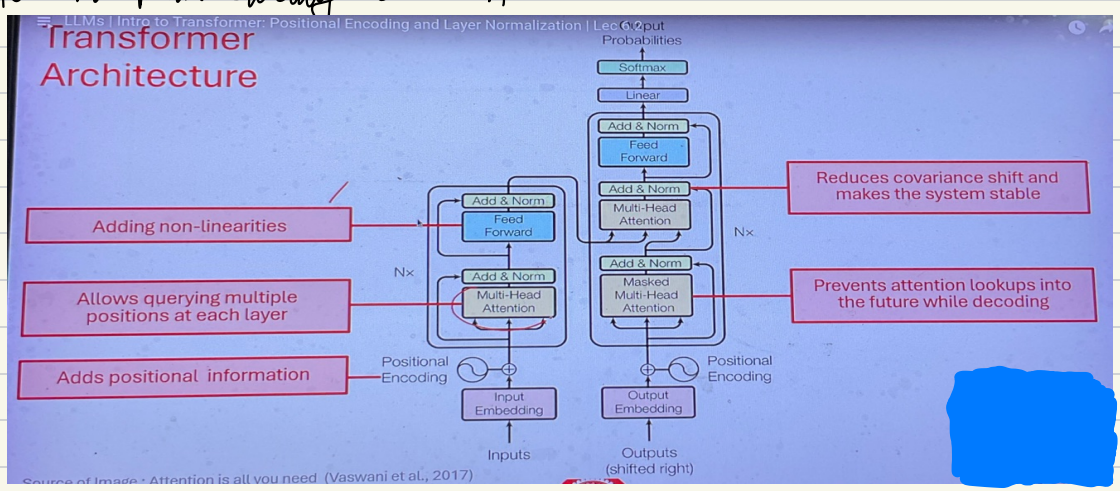
* So, when we perform a task, like machine translation in the

original transformer paper), we need to predict each token.

*) We feed <START> token, it generates a word, that goes to input of the next step which in turn generates the next token & so on.

Q) How do we parallelize this during training?

A) Since, we already have the translated version of the input, we use Teacher forcing to calculate the output of all the timesteps in parallel i.e., feed the <START> token, <START><FIRST_WORD>, <START><FIRST_WORD><SECOND_WORD> and so on to the model in parallel & calculate the errors from each of the outputs. Finally, we can sum it up & backpropagate all together or do it individually as well.



Lecture 7 - Pretraining Strategies : ELMo & BERT

"The complete meaning of a word is always contextual, and no study of meaning apart from a complete context can be taken seriously." (JR Firth 1935)

eg: I **record** the **record**.

The 2 instances of the word 'record' mean different things

However, methods like GLoVe & word2vec will give the same embeddings for both positions.

> The representation of a word should depend on the context in which it appears

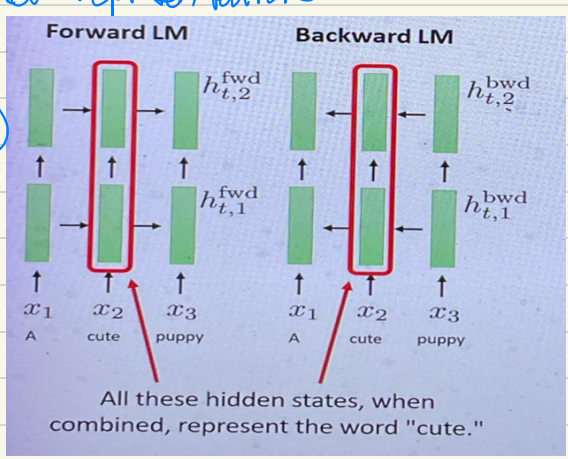
Solution: Train contextual representations on text corpus

A bat flew out of a cave → [-0.107, 0.109, -0.542, ...]

He hit the ball with a bat. → [0.349, 0.271, 0.139, ...]

Embeddings from Language Models (ELMo)

Paper: Deep contextualized word representations



* Non-Transformer based approach
* Relies on RNNs (could be LSTM or GRU)
* Hidden state outputs are used as the embeddings.
* RNN generally has 3 matrices W_h (hidden state), W_e (input) & U (to project outside).
* For now let's assume that there

is only 1 matrix W_h. We pass one sequence at a time (or in batches) & train using language model objective i.e., next word prediction. Then we backprop & update W_h.

* The claim is that the hidden state will serve as the embedding for the word at that position. (like 'cute' as marked in red in the above example)

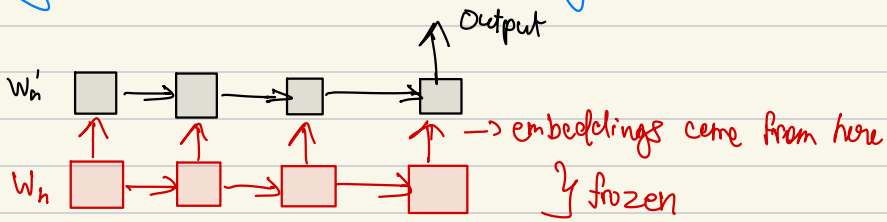
How?

- Lets assume that we have a dictionary with each row as the embedding of a particular token. (Assume word level tokenization & random initialisation).
- When we do a pass through this RNN, the inputs are fetched from the dictionary & fed into the RNN, we do backprop & update the hidden state weights.
- The new hidden state now replaces the previous embedding in the dictionary (for that word).
- The above steps are then repeated for every instance in the training set. (Can also be done in batches)

* This is a language model based RNN (No task involved so far)
* Now we freeze this RNN.

* Now, lets say we have a task & task specific data which we want to use to train/fine-tune the model further.
* We apply another RNN on top of this RNN which will be responsible for performing this task.

* Let's say our task is sentiment analysis.

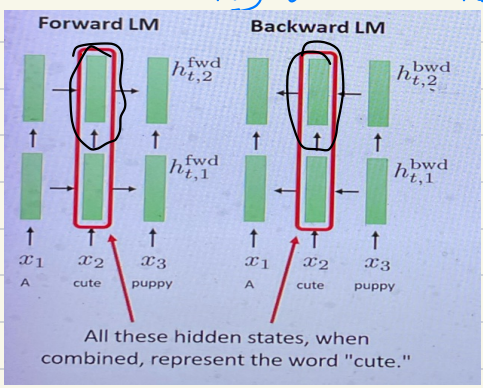


* Given the task specific input, all the token embeddings come from the frozen RNN layer and the 2nd layer has another weight matrix W_n' and it produces some output; we calculate the error which is then backpropagated and the weights are updated.
* During inference - we fetch the input embedding of each token from the dictionary, pass it (sequentially) through the first (frozen) RNN layer to generate the the next hidden state.
The current hidden state outputs from the frozen layer are very different from the stored hidden states and also contextual because the stored hidden states were passed through an RNN to generate them which makes them a function of the previous timesteps and thus contextual. (to clarify, the output of the hidden state for the same token will be different if used in a different context)
These hidden states are now moved to the upper task specific layer (which we have also trained) which will be used to perform the task.

This was a high level overview of ELMo. There are some minor differences in the actual implementation.

ELMO Architecture

* Unidirectional RNN is not a great approach. We use a bidirectional RNN in the actual implementation.
* The paper used a stacked 2-layer RNN (i.e., the bottom language model RNN was a stacked RNN) and 2 RNNs (one forward & one backward RNN)
* The final hidden state that will be stored in the dictionary (corresponding lets say to the word cute as shown in the alongside figure) can either be the concatenation of all 4 hidden states or only the top layers.



* The forward LM is a deep LSTM that goes over the sequence from start to end to predict token t_k based on the prefix $t_1 \cdots t_{k-1}$:

$$p(t_k | t_1 \cdots t_{k-1}; \Theta_{tc}, \overrightarrow{\Theta}_{LSTM}, \Theta_s)$$

Parameters: token embeddings Θ_{tc}, Θ_{LSTM}, softmax Θ_s
↳ same as W_h ↳ same as U

- All other parameters in the RNN (like W_e) remain the same.

* The backward LM is a deep LSTM that goes over the sequence from end to start to predict token t_k based on the suffix $t_{k+1} \cdots t_N$

$$p(t_k | t_{k+1} \cdots t_N; \Theta_{tc}, \overleftarrow{\Theta}_{LSTM}, \Theta_s)$$

Train these LMs jointly, with the same parameters for token representations and softmax layer (but not the LSTMs)

$$\sum_{k=1}^{N} (\log p(t_k | t_1 \cdots t_{k-1}; \Theta_{tc}, \overrightarrow{\Theta}_{LSTM}, \Theta_s) + \log p(t_k | t_{k+1} \cdots t_N; \Theta_{tc}, \overleftarrow{\Theta}_{LSTM}, \Theta_s))$$

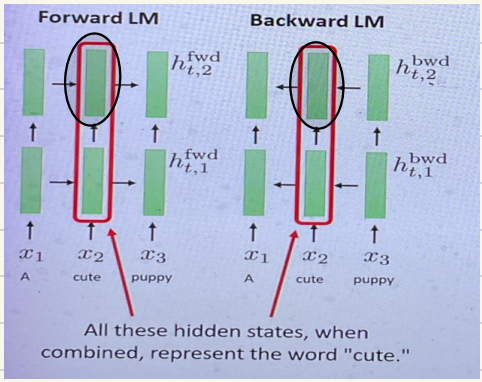
ELMo Token Representation

* Given a token representation x_k, each layer j of the LSTM language model computes a vector representation $h_{k,j}$ for every token k
* With L layers, ELMo represents each token as

$$R_k = \{x_k^{LM}, \overrightarrow{h}_{k,j}^{LM}, \overleftarrow{h}_{k,j}^{LM} \mid j = 1 \ldots L\}$$

$$= \{h_{k,j}^{LM} \mid j = 0 \ldots L\}$$

where $h_{k,j}^{LM} = [\overrightarrow{h}_{k,j}^{LM}; \overleftarrow{h}_{k,j}^{LM}]$ and $h_{k,0}^{LM} = x_k$



Instead of taking all 4 embeddings, we can only use the top layer embeddings because the top layer is directly followed by the 'U' layer & then softmax. Thus, during backprop, the top layer is affected the most.

* We can also add some parameters. Along with $h_{k,j}^{LM}$, we add a layer specific parameter 's' and task specific parameter ('γ').

When we apply a task specific layer on top of this, then instead of freezing the RNN completely, we can add some parameters which are dependent on the task.

$$ELMo_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} h_{k,j}^{LM}$$

These parameters will be updated when we fine-tune/train for the task & fetched during inference time. So, now the hidden states are somewhat dependent on the task.

*) ELMo token representations are purely character-based: a character CNN, followed by a linear projection to reduce dimensionality.
*) 2048 character n-gram convolutional filters with two highway layers, followed by a linear projection to 512 dimensions
*) Advantage over using fixed embeddings: no UNK tokens, any word can be represented.

Evaluation

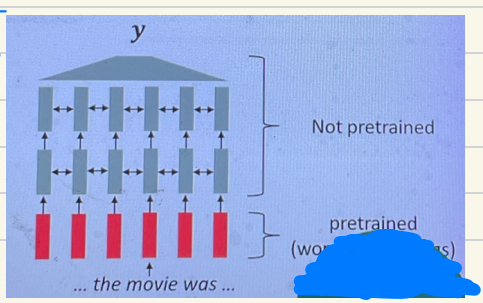
*) ELMo was one of the first large scale models (large was over 90 million)
*) ELMo gave improvements on a variety of tasks:
- question answering (SQuAD)
- entailment/Natural language Inference (SNLI)
- semantic role labeling (SRL)
- coreference resolution (Coref)
- named entity recognition (NER)
- sentiment analysis (SST-5)

TASK	PREVIOUS SOTA		OUR BASELINE	ELMO + BASELINE	INCREASE (ABSOLUTE/ RELATIVE)
SQuAD	Liu et al. (2017)	84.4	81.1	85.8	4.7 / 24.9%
SNLI	Chen et al. (2017)	88.6	88.0	88.7 ± 0.17	0.7 / 5.8%
SRL	He et al. (2017)	81.7	81.4	84.6	3.2 / 17.2%
Coref	Lee et al. (2017)	67.2	67.2	70.4	3.2 / 9.8%
NER	Peters et al. (2017)	91.93 ± 0.19	90.15	92.22 ± 0.10	2.06 / 21%
SST-5	McCann et al. (2017)	53.7	51.4	54.7 ± 0.5	3.3 / 6.8%

In fact, we can use ELMo embeddings to initialize transformers.

Where we were: Pre-trained Word Vectors

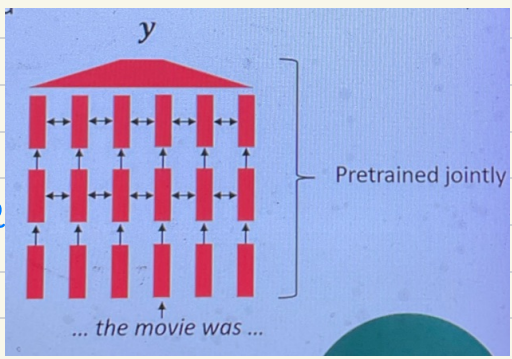
*) Start with pretrained word embeddings (no context!)
*) Learn how to incorporate context in an LSTM or Transformer while training on the task



* The training data we have for our downstream task (like Question Answering) must be sufficient to teach all contextual aspects of language.
* Most of the parameters in our network are randomly initialized.

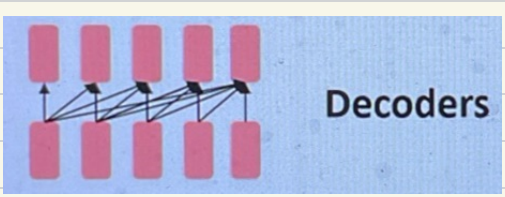
Pretrained Word Vectors → Pretrained Models

* All (or almost all) parameters in NLP networks are initialized via pretraining.
* Pretraining methods hide parts of the input from the model, and train the model to reconstruct those parts.
* This has been exceptionally effective at building strong:
  - representations of language
  - parameter initializations for strong NLP models
  - Probability distributions over language that we can sample from



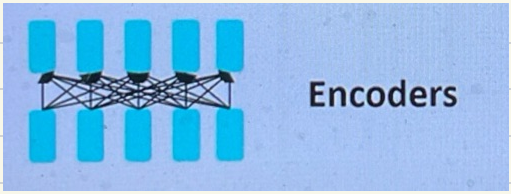
Pretraining for 3 types of architectures

The neural architecture influences the type of pretraining, and natural use cases



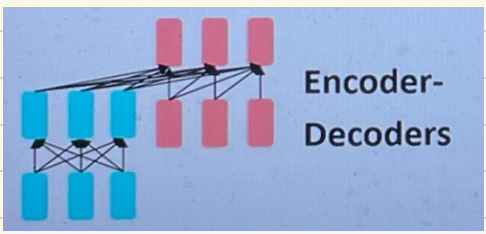
- Language models! what we've seen so far.
- Nice to generate from; can't condition on future words

eg: GPT



- Gets bidirectional context - can condition on future words!
- How to pretrain?

eg: BERT



- Good parts of encoders & decoders?
- What's the best way to pretrain them?

eg: T5, BART

Bidirectional Encoder Representations from Transformers (BERT)

Paper: BERT: Pre-training of Deep Bi-directional Transformers for Language Understanding

*) BERT is an encoder only pre-trained model.

Background - Bidirectional Context

*) Since, this an encoder only model, there is no masked self-attention i.e., each token can attend to all other tokens (i.e., bidirectional context)

*) Bidirectional context, unlike unidirectional context, takes into account both the left & right contexts

eg: Apple is my favorite fruit and I eat it all the time.

Left context | target | Right context

Masked Language Modeling (MLM)

*) Mask out k% of the input words, then predict the masked words (Usually k = 15%). Example:

I like going to the [MASK] in the evening.
↓
park

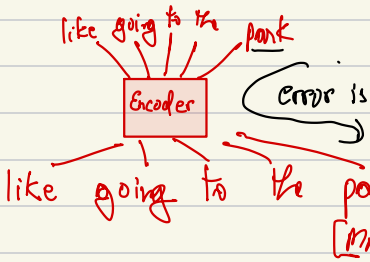
- Too little masking: Too expensive to train
- Too much masking: Not enough context

} 15% was found empirically good enough

*) The model needs to predict 15% of the words, but we don't replace with [MASK] 100% of the time. Instead:

- 80% of the time, replace with [MASK]
  eg. like going to the park → like going to the [MASK]
- 10% of the time, replace with random word
  eg: like going to the park → like going to the store
- 10% of the time, leave it as it is
  eg: like going to the park → like going to the park

Why do this?
- Because if we only teach the model to predict [MASK] token, the model will start forgetting the actual tokens.



error is only computed at this point not the other tokens

Next Sentence Prediction

* BERT was trained on 2 tasks - (1) MLM (2) Next Sentence Prediction
* Here we take 2 sentences S_1 & S_2 from the corpus & let the model predict if S_2 will follow S_1.

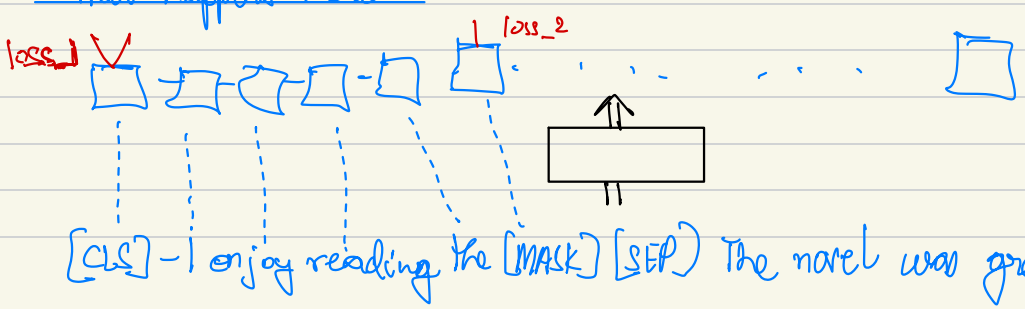
+ve samples → sentences that appear side-by-side.
-ve samples → random shuffling

eg: Input = [CLS] I enjoy read [MASK] book ##S [SEP]
I finish ##ed a [MASK] novel [SEP]
Label: Is Next

Input = [CLS] I enjoy read ##ing book [MASK] [SEP] The dog ran [MASK] the street [SEP]
Label: Not Next

* Both these tasks are performed simultaneously.
* The [SEP] token seperates the two sentences.
* Along with [SEP] and [MASK] we introduce another token called [CLS] which is prepended to the sequence (stands for classification)

What happens here?



- The sentence is passed through the encoder which tries to predict the output for each token [CLS] - I - enjoy ... and so on
- We use a task specific head (basically a linear layer U) on top of the CLS token which projects the output into 2 classes - Yes/No (for next sentence task)
- Now, we have 2 losses - loss from the output of the [CLS] token (for sentence task) & the loss from the output of the [MASK] token (for the word prediction task). These losses will be backpropagated.

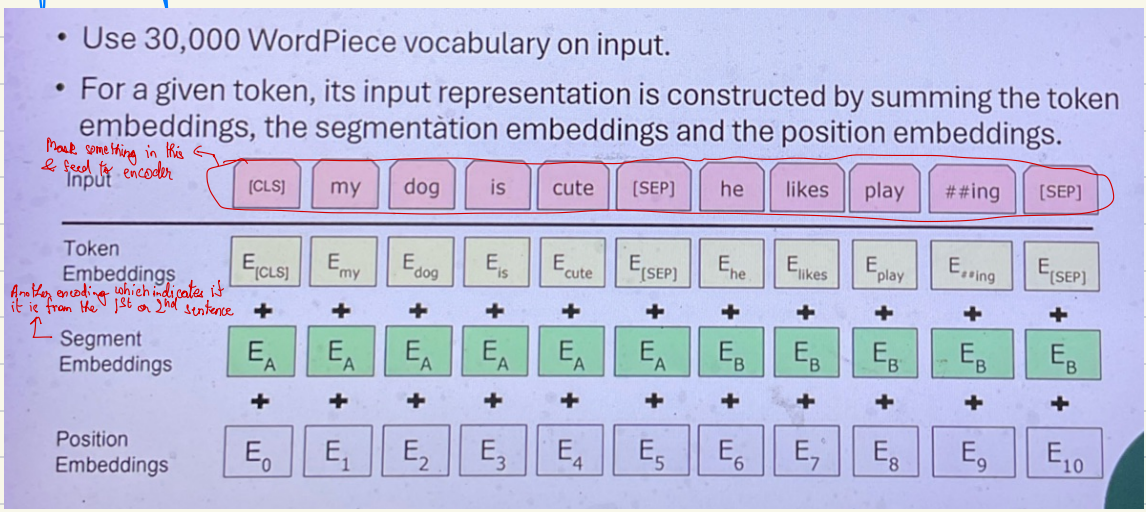
Q) Why do we need the CLS token?
Can't we just take the concatenation of embedding of all the tokens, pass through a linear layer followed by non-linearity?

A) CLS also acts as a normal token so it attends to all other tokens during self-attention. However, it doesn't indicate any meaning of a word because CLS is not part of the sentence. We can have a classification layer on top of 'I', 'read' etc. as they also attend to all tokens. However, they themselves have some meaning & are biased towards certain words. CLS is unbiased towards any token.
Also, on adding the linear layer on top of CLS token, we add a linear layer which has one dimension as the size of 1 embedding. If we concatenated all embeddings the size of 'U' would be much larger & inefficient to compute.

* Like ELMo, we can use BERT for contextual

word embeddings.

Input Representation

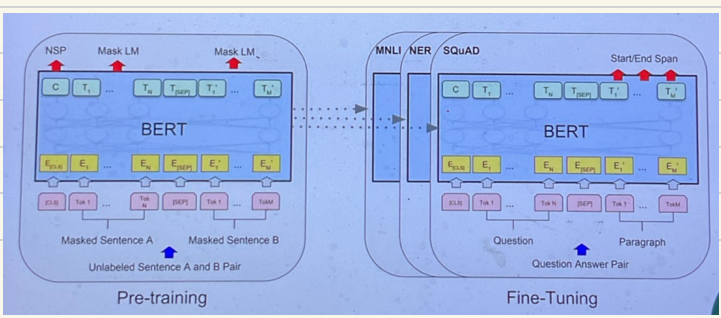


Training Details

- Data: Wikipedia (2.5B words) + Book Corpus (800M words)
- Batch Size: 131,072 words (1024 sequences * 128 length or 256 sequences * 512 length)
- Training time: 1M steps (~40 epochs)
- Optimizer: AdamW, 1e-4 learning rate, linear decay
- BERT-base: 12 layer, 768 hidden dim, 12-heads
- BERT-large: 24 layer, 1024 hidden dim, 16-heads
- Trained on 4x4 or 8x8 TPU slice for 4 days

fine-tuning procedure on QA task

*) We are given a question



q: Who played Vikram Rathore in Jawan?

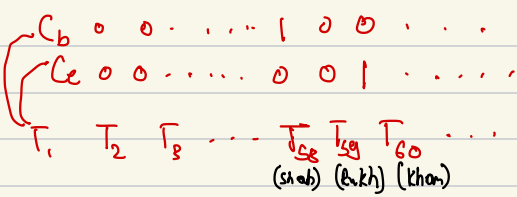
passage: ---------- the role of Vikram Rathore was played by Shah Rukh Khan ----------

span start, span end

Task → to retrieve

Q) How do we use BERT for this?

- During training, we feed the CLS - question - SEP - passage - SEP
- We add 2 classification layers on top of every token (Think of 'Shah Rukh Khan' as a span)
- 1st classification head keeps track of the beginning of the span (C_b) & 2nd keeps track of the end of the span (C_e).



- Example of C_b & C_e output values is shown in the image above (for the SRK example). This is the ground truth value that we want our model to predict.
- Now, we feed the input to the model, it produces the embeddings & these embeddings are fed to the classification layers which will essentially produce some probabilities (for beginning & end tokens) (And the hope is to predict 1 for Shah from C_b & 1 for Khan from C_e)
- The losses get backpropagated along the task specific layers accordingly (Pre-trained layers can also be updated)

* When we get a test sentence (question) & passage - we use our fine-tuned model - pass the sequence, produce P_b (corresponding to C_b) & P_e (corresponding to C_e) probabilities & our task would be to figure out 2 such positions such that

$$P_b \times P_e = \max$$

If we chose 2 positions i & j then

$i \leq j$

P_b P_e

↳ now we find P_b & P_e such that $P_b \times P_e = \max$

Q) What if the answer appears multiple times?
A) Generally, we consider the 1st appearance

BERT : Evaluation

BERT was massively popular and hugely versatile; finetuning BERT led to new state-of-the-art results on a broad range of tasks.

- **QQP:** Quora Question Pairs (detect paraphrase questions)
- **QNLI:** natural language inference over question answering data
- **SST-2:** sentiment analysis
- **CoLA:** corpus of linguistic acceptability (detect whether sentences are grammatical.)
- **STS-B:** semantic textual similarity
- **MRPC:** microsoft paraphrase corpus
- **RTE:** a small natural language inference corpus

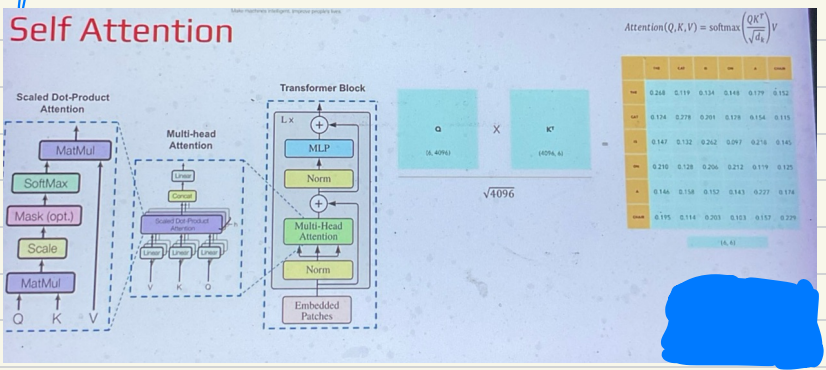
System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	Average -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$BERT_{BASE}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$BERT_{LARGE}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

* BERT was SOTA till 2020 & many variations like DeBERTa, RoBERTa followed it. It was also a baseline for a lot of future research.

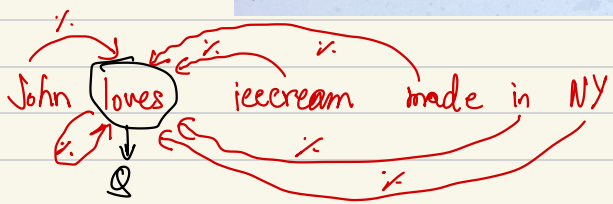
Lecture 8.1 – Advanced Attention Mechanisms

Self Attention : Recap

* Attention - To what extent one word is influenced by other words (in terms of a percentage)



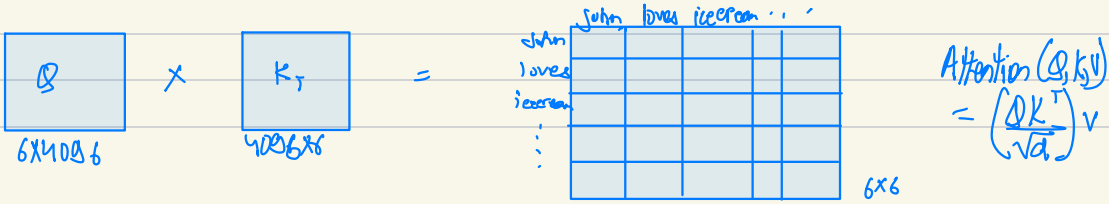
eg:



* This percentage is computed as a similarity b/w query of 'loves' & the key of the other words (like 'John')

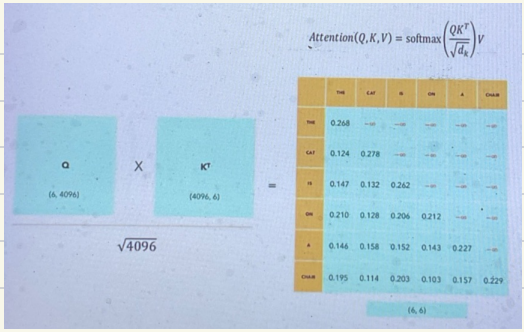
So we have 2 different forms of John → $John_K$ (which influences) ↵ $John_V$ (which is a representative)

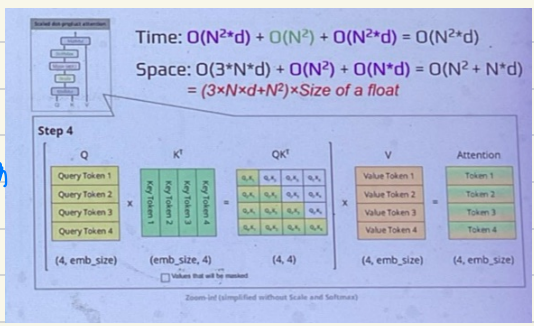
Q) Why do we have different $John_K$ & $John_V$?
One is a representative of John to check how influential it is to other words & the other is the content that is shared with other words if it is found influential



Causal (forward Masked) Attention

* Decoder only models have a forward mask because we try to restrict attention in a way such that a word cannot be influenced by future words.
* To do this, we put a $-\infty$ in the matrix in place of all future tokens (so it becomes 0 during softmax)
* We would now like to see if we can optimize the causal attention in terms of time & space.





Currently,

Time Complexity : $O(N^2 * d) + O(N^2) + O(N^2 * d) = O(N^2 * d)$

- $O(N^2 * d)$ ↓ $Q \times K^T$
- $O(N^2)$ ↓ softmax
- $O(N^2 * d)$ ↓ (softmax) $\times V$

N → No. of tokens in context
d → dimension of token

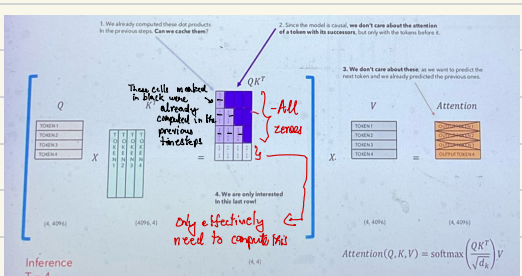
Space Complexity : $O(3 * N * d) + O(N^2) + O(N * d) = O(N^2 + Nd)$

- $O(3 * N * d)$ ↓ Q, K, V
- $O(N^2)$ ↓ softmax scores
- $O(N * d)$ ↓ output after attention

Byte size : $(3 * N * d + N^2) *$ size of float

Can we do better?

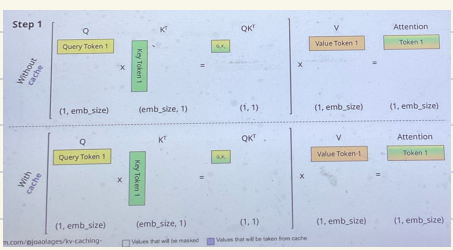
* Let's take this particular example where we do inference at timestep $t = 4$.
* We pass in all the 4 tokens' query & key & calculate the attention scores

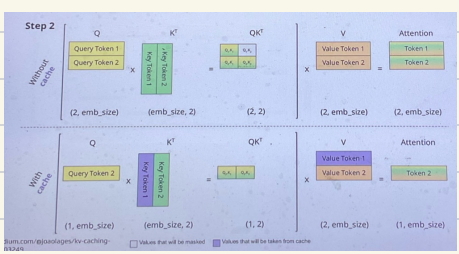


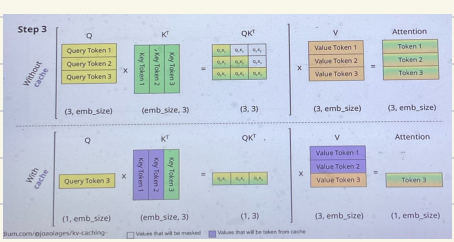
through matrix multiplication.

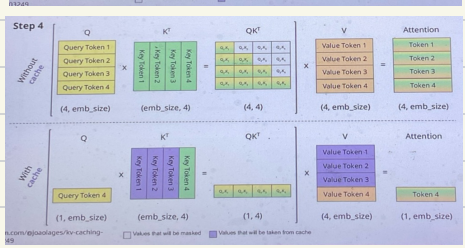
* But the scores were already calculated for the 1st 3 words at T = 1, 2 & 3 so recomputing them is a waste of compute. So, essentially, when we move from T = 3 to T = 4, even though we add 1 row & 1 column (3×3 → 4×4) to the attention matrix, the column is mostly 0's (since previous words cannot attend to future tokens) & we only need to compute the last row.
* Also, we don't need the output values recomputed as K, V were already calculated in the previous timesteps and are not subject to any change

KV Cache based (Forward Masked) Attention









KV Cache Storage : $O(N*d)$

$(2*N*d) \times$ size of float

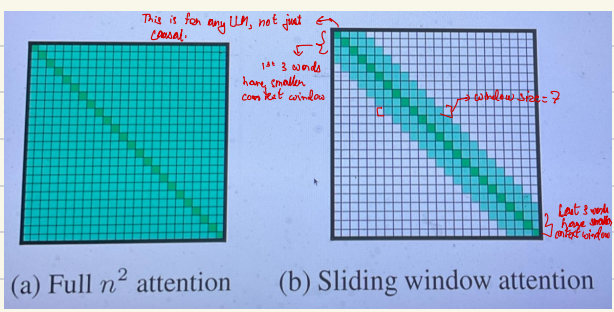
vs

$(3*N+N^2) \times$ size of float

* We get an idea of how K-V caching should work from the points mentioned above. We don't need to compute the entire attention matrix again. We only need to operate on the new token in the following manner:
  - Compute the new q, k, v rows for only the new token
  - New q will be used immediately (This is why there is no query cache)
    ↳ since query only used to compute the attention scores for that particular token & nowhere else.

- Append the new key, value entries to the K, V caches
- Compute new attention row by doing matrix-vector multiplication between new q row & k_cache.transpose()
- Compute new v row by doing matrix vector multiplication between new att row and v_cache.transpose()
- The output (which is just for the latest token) is passed to the next layer.
- This can proceed through subsequent layers & attention heads since we only care about the latest token.

Sliding Window Attention



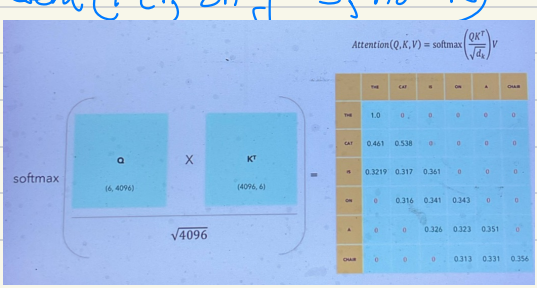
*) For a lot of NLP problems, we really don't need to get the influence from all other tokens.

*) In 2020, Allen Institute for A.I. came up with the idea of a smaller context window.

*) In the above diagram we see a context window of size = 7 (-3 to +3). If there are not enough words on any side of the current word then the window becomes shorter. eg: for the 1st 3 & last 3 words in this example

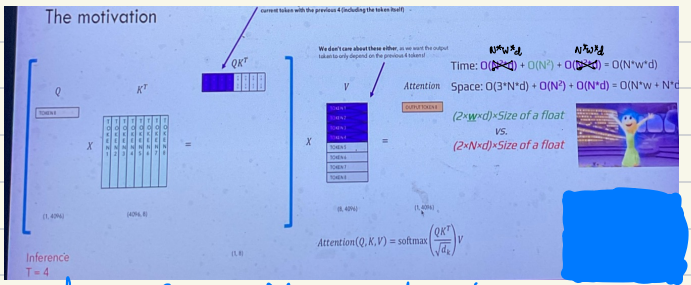
*) The above diagram is not for causal LMs. For causal LMs, the future tokens will not be present (i.e., only -3, no +3)

*) Here is an example of a causal LM with Sliding Window Attention. The window size = 3.



What about the KV cache?

* There is no such change change to the architecture that prevents us from applying KV caching.

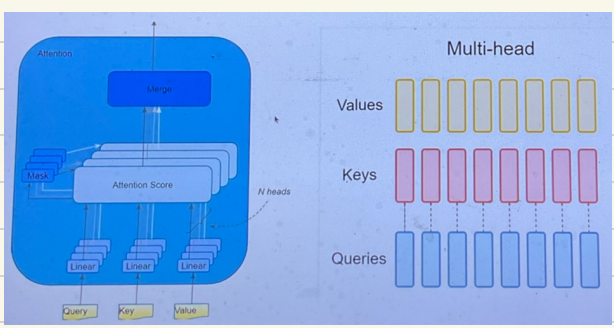


* The only difference is when fetching the key vectors from the cache to compute QK^T, we only fetch K's within the context window.
* Similarly, when calculating the final output, only the values of tokens within the context window are fetched

* However, sliding window attention doesn't perform well in many NLP tasks. eg: dialogue chaining

Multi head Self Attention

* The original transformers paper used multiple attention heads and the rationale behind it was that multiple heads could capture different kinds of relations between tokens that would not be possible with a single head.

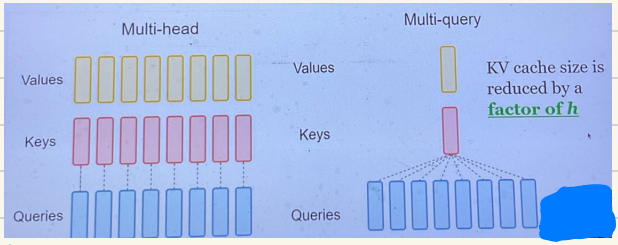


eg: for the sentence 'John loves icecream made in India'
- Head 1 could capture the fact that John & icecream appear together frequently
- Head 2 could capture that icecream - a common noun appears often in association with John - a proper noun

* Multiple heads could capture multiple syntactic & lexical associations
* This happens at the cost of increased computation overhead for each of the heads.

Multi-Query Attention (MQA)

* Paper by Google in 2019.
* They came up with the idea that multiple query vectors can be used (just like asking multiple questions), to query the same key (just like having a single representation) & value.



* The hypothesis was that a single (key & value) vector can be used as a kind of universal representation for that token and multiple queries can be used to identify the different kinds of relations it has with the other tokens (it's like a person asking another person their relationship at different points - at home you are my neighbor, at school you are my classmate, etc. However, all those answers are given by the same person just like so similarly, all queries related to a particular token should be answerable by the same key (& value).

Do we lose out on something?

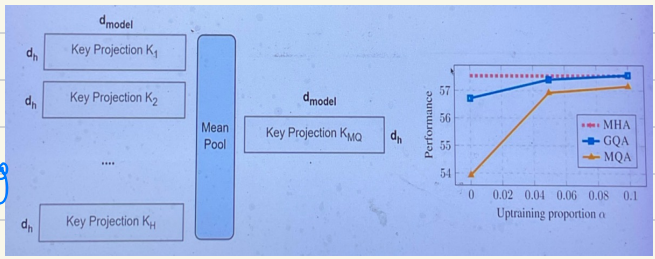
- Decline in performance quality
- Training instability

Why?

Because the assumption that the a single key value can answer all types of queries is wrong i.e., by the same analogy above - the answer to the home vs school question cannot be answered by the same me. It is 2 different personalities

of me which answers these questions seperately & independently.

Uptraining: Converting MHA to MQA



*) Paper in 2023 by GA Tech & Google Research.

*) Instead of randomly initializing the W_K & W_V matrices (which is used to calculate K & V), we can take an existing (maybe an MHA) model which is already trained, take its key matrices (for multiple heads), take a mean of it to create one matrix and initialize our MQA model with that.

*) We can do further pretraining after initializing.

*) So, if we want to train for say 'k' epochs then we start with the MHA model → train it for k' steps → take the key weights for all heads from it & do a mean pooling → initialize our MQA model with these weights & continue pretraining the MQA model for k-k' steps.

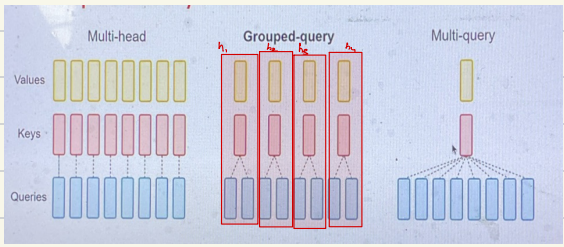
$$\alpha = \frac{k'}{k} \longrightarrow \text{i.e., proportion of MHA training}$$

$\alpha \to 0$ indicates only MQA training and higher α would mean a longer proportion of MHA training.

*) As we see in the above graph, when we slightly increase the proportion of MHA training, the performance improves drastically. So, we start to achieve more training stability. This process is called uptraining.

However, this improvement is only relative and we still see an overall decline in performance compared to the original MHA.

Grouped Query Attention (GQA)



*) Paper by Google Research in 2023.

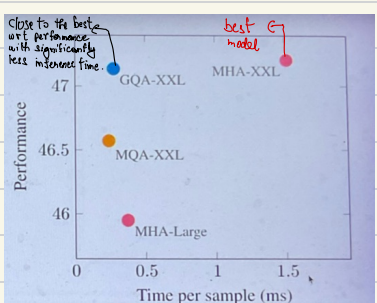
*) Slightly modified version of MQA.

*) Intuition: Instead of making this big assumption that only 1 k & v is enough to universally represent a token, let us relax the hypothesis & say that k & v do change but not as many times as the query

*) So we have a few keys & values i.e., less than that in MHA but more than MQA (which has only 1). We try to choose the best of both worlds.

*) We group a set of queries with with 1 set of key & value into 1 head. When there is only 1 group then GQA = MQA

What did we gain?

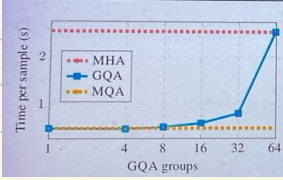


*) MQA & GQA models were tried & tested on multiple datasets for different kinds of tasks like summarisation, QnA, etc.

*) Results showed that the performance of GQA (& group XXL) was even better than some MHAs (MHA-Large) while the inference time while still slower than MQA was still significantly less than an MHA of the same size.

*) Inference time grows exponentially with increase in the no. of groups.

Model	T_{infer} s	Average	CNN R_1	arXiv R_1	PubMed R_1	MediaSum R_1	MultiNews R_1	WMT BLEU	TriviaQA F1
MHA-Large	0.37	46.0	42.9	44.6	46.2	35.5	46.6	27.7	78.2
MHA-XXL	1.51	47.2	43.8	45.6	47.5	36.4	46.9	28.4	81.9
MQA-XXL	0.24	46.6	43.0	45.0	46.9	36.1	46.5	28.5	81.3
GQA-8-XXL	0.28	47.1	43.5	45.4	47.7	36.3	47.2	28.4	81.6



Key Takeaways

- GQA/MQA aim: To reduce the need for storing a large amount of KV cache
- LLM server can now handle more requests, larger batch sizes & increased throughput.
- Cannot significantly reduce the computational load ↓ in terms of time complexity
- Quality degradation remains

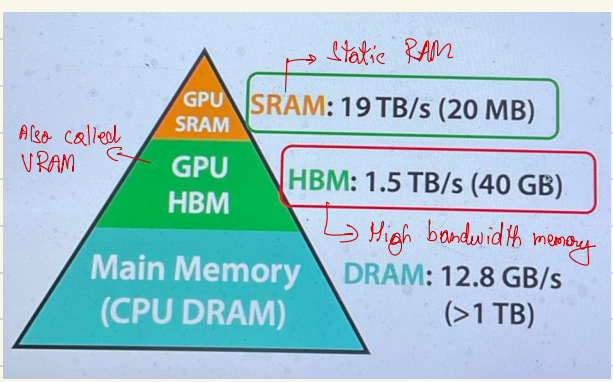
Lecture 8.2 - Advanced Attention Mechanisms - II

Here we look at a method to preserve the performance of any causal LM while bringing the time complexity to linear time.

Q) Can we optimize without performance degradation?
A) Yes → Flash Attention

A bit more about GPUs

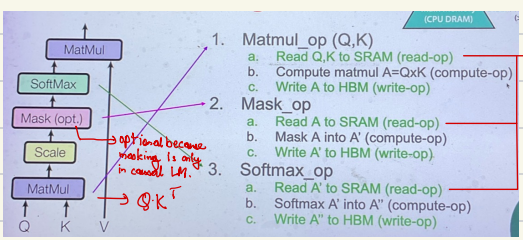
- Here is a hierarchy of memory in GPU systems.
- All the operations we studied until now (like KV caching, matmul, softmax, etc) was happening at the HBM level.
- HBM has decent processing power with good storage



*) However, we also have the GPU SRAM which is multiple times faster than the HBM but has very low memory. So, it cannot be used for the entire process of attention calculation etc.

Q) Can we somehow use the SRAM more while reducing our dependency on VRAM operations? We want to implement the exact same algorithm but harness the power of the SRAM even more.

What happens right now?



→ from the VRAM/HBM

Q & K are too large to be loaded into SRAM entirely & thus, the process is performed in blocks
→ Same thing with P & V.

$$S = QK^T \in \mathbb{R}^{N \times N}, \quad P = \text{softmax}(S) \in \mathbb{R}^{N \times N}; \quad O = PV \in \mathbb{R}^{N \times d},$$

Algorithm 0 Standard Attention Implementation

Require: Matrices $Q, K, V \in \mathbb{R}^{N \times d}$ in HBM.

1: Load Q, K by blocks from HBM, compute $S = QK^T$, write S to HBM.
2: Read S from HBM, compute $P = \text{softmax}(S)$, write P to HBM.
3: Load P and V by blocks from HBM, compute $O = PV$, write O to HBM.
4: Return O.

*) As we see from the above diagram, we constantly do read-write operations on the VRAM.
*) While the compute operations on the SRAM are superfast, the reads are slow & writes are even slower.
*) I/O awareness is missing i.e., there is a lot of unnecessary intermediate writes & reads.

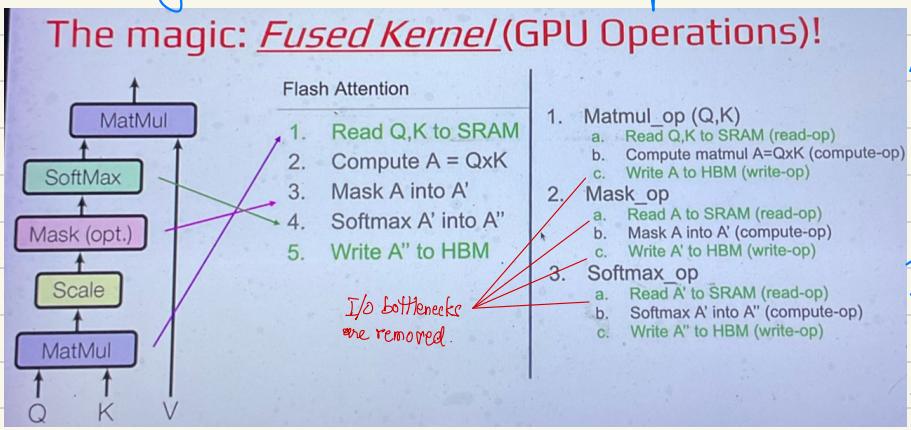
Q) Can we simplify this?

Flash Attention

↳ Stanford, SUNY Buffalo (2022; NeurIPS)

→ Not entirely from LCS2 lecture since it wasn't clear (Revisit other videos if necessary)

The magic: Fused Kernel (GPU operations)



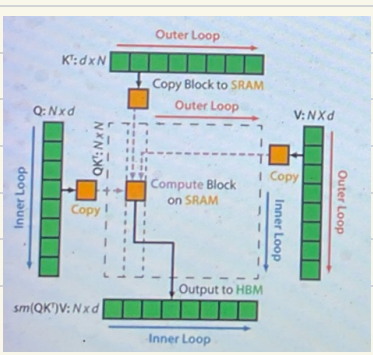
*) In flash attention, we remove all the intermediary read writes.
*) We read Q,K from V RAM → compute QK^T → perform masking → do softmax → write to HBM.

*) So, the only slow steps here are the 1st read & final write. We avoid all the intermediate write back to the HBM and subsequent reads, thereby avoiding I/O bottlenecks.
*) This process of eliminating intermediate read/writes is called fused kernel - (One operation inside the GPU is called kernel & we fuse multiple operations into a single kernel)

How do we do this?
- To avoid reading & writing the attention matrix to & from the HBM, we need to
  - ↳ compute softmax reduction without access to the whole input.
  - ↳ not store QK^T for backward pass (but still somehow do the backward pass)

Tiling

- Split the input into blocks & make several passes over input blocks, thus incrementally performing the softmax reduction.



- The dotted box in the above figure shows the QK^T matrix (which is never actually computed entirely in flash attention)
- Green blocks indicate the HBM & orange blocks are the faster SRAM. (so Q,K,V are stored in HBM).

Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.

- So, we run 2 for loops:
  - In the outer loop we load a block of K & V
  - Inner loop loads Q block-by-block. (eg: In the above diagram we load 2nd block of K,V & 4th block of Q)
  - From here we partially compute the entire O i.e., $(\text{softmax}(QK^T)).V$. We output this to HBM.
  - Next we load the next Q block & compute O for the next set of queries
- After iterating over all Q's we pick the next K,V & repeat to update the O.
- The final O cannot be calculated until we have processed all the positions (but this can be done incrementally)

Q But can we decompose softmax? How? Can we perform local aggregates & obtain the global softmax from that?

what is softmax? $\rightarrow [a \ b \ c] \Rightarrow \left[\frac{e^a}{\sum e^i} \quad \frac{e^b}{\sum e^i} \quad \frac{e^c}{\sum e^i} \right]$

In Tiling, we compute attention by blocks. We decompose the large softmax with scaling for numerical stability & compute as:

$$m(x) := \max_i x_i \qquad f(x) := \left[e^{x_1 - m(x)} \; \ldots \; e^{x_B - m(x)}\right]$$

$$l(x) := \sum_i f(x)_i \qquad \text{softmax}(x) := \frac{f(x)}{l(x)}$$

(meaning we do not have all the K,V vectors loaded to SRAM to compute QK^T entirely)

But we do not have the entire x when computing 'l' & 'm'.

For vectors $x^{(1)}, x^{(2)} \in \mathbb{R}^B$, we can decompose the softmax of the concatenated $x = [x^{(1)}, x^{(2)}] \in \mathbb{R}^{2B}$ as:

$$m(x) = m\left(\left[x^{(1)}, x^{(2)}\right]\right) = \max\left(m(x^{(1)}), m(x^{(2)})\right),$$

$$f(x) = \left[e^{m(x^{(1)}) - m(x)} f(x^{(1)}) \quad e^{m(x^{(2)}) - m(x)} f(x^{(2)})\right]$$

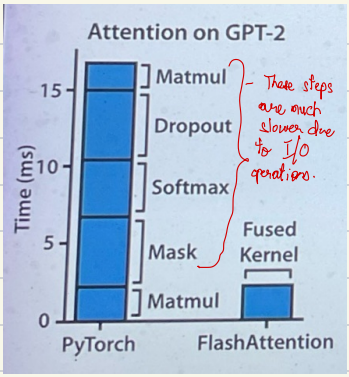
$$l(x) = l\left(\left[x^{(1)}, x^{(2)}\right]\right) = e^{m(x^{(1)}) - m(x)} l(x^{(1)}) + e^{m(x^{(2)}) - m(x)} l(x^{(2)})$$

$$\text{softmax}(x) = \frac{f(x)}{l(x)}$$

* Since max & sum are distributive in nature, so the overall softmax is algebraic.
* Keeping track of the extra statistics $(m(x), l(x))$ allows us to compute softmax one block at a time.
* We compute partial (O_i, l_i, m_i) & store it in the HBM & repeat it for every j-th block of K, V (outer loop) & keep updating (O_i, l_i, m_i) in the HBM (as per the method given above).
* The final j-th block update gives us the eventual global softmax for all vectors & the final output values
* The O_i, l_i, m_i statistics are enough for recomputation (of attention matrix) for gradient calculation in the backward pass.

More on Attention → https://paperswithcode.com/methods/category/attention-mechanisms

How well did they do?



BERT Implementation	Training time (minutes)
Nvidia MLPerf 1.1 [58]	20.0 ± 1.5
FLASHATTENTION (ours)	**17.4** ± 1.4

Time Complexity: $O(N*d)$
Space Complexity: $O(N*d)$

* A significant reduction in inference time on GPT-2.
* We were able to reduce the quadratic ($N*N$) time complexity to near linear time complexity ($N*d$; $N \gg d$)
* It outperformed the best techniques on NVIDIA MLPerf benchmarks.
* Recently, Flash Attention 2 was also released which improves the fusion of the kernel happens.

So, is the attention problem solved?

* Specific GPU requirements
  - Does GPU have CUDA? (If not we need to re-write in ROCm (AMD) or SYCL (Intel))
  - Need fast shared GPU memory (SRAM)
  - Need Tensor cores (specifically dedicated to matrix operations)
* Too much pro-NVIDIA (Ampere, Volta, etc.)
* If we have a new way to implement attention we will need to re-write the entire fused kernel

Key Takeaways → Avoid unnecessary HBM read/writes
→ Maximize SRAM computation

Lecture 9 - Tokenisation Strategies

* Tokenisation is important because
  - helps determine which words are similar (at a higher level)
  - helps understand unknown words
  - tokens are the inputs we give to many neural language models (like word2vec, RNN, Transformer, etc)

* Before the arrival of neural language models, delimiters were used to tokenize a running text.
* But this method was not very robust especially in test time when dealing with unknown/difficult to decipher token.
* We look at 3 strategies that have been used heavily in different kinds of LLMs.

Subword tokenisation

* Word level tokenisation is not useful because words are just a combination of characters and an exponential no. of words are possible.
  Neither is character level tokenisation because it is too limited in size & not very meaningful.

* Approach
  - A middle ground between word & character level tokenisation strategies
  - Frequently used words should not be split into smaller subwords
  - Rare words should be decomposed into meaningful subwords

Remember: Tokenisation is performed during training stage to identify the vocabulary by going through the entire corpus. This vocabulary will be used to parse all tokens in test time.

Q) Why is this useful?
Let's say we have a verb 'catch' & the word 'catching'. Say we have catch in our training set but not catching. But the test set has catching. With word level tokenization we will not be able to split it, but in subword level tokenisation we can split it into 'catch' & 'ing' & now the 1st part is same as catch so maybe the 2 words are related.
Similarly, say 'catching' & 'hugging' can be split into 'catch' & 'ing' and 'hug' & 'ing' so the second part of both words are similar so maybe they are similar (which they are in terms of their tense form i.e., continuous tense)

Tokenisation - still an area of active research

*) 3 types of tokenisation techniques
- Byte-Pair Encoding (BPE) (Sennrich et. al 2016)
- WordPiece (Schuster & Nakajima, 2012)
- Unigram Language modeling tokenisation (Kudo, 2018)
  ↳ used in sentencepiece

*) All modern LLMs use one of or a combination of these methods

*) These methods use data, guide the tokenisation process:
- A token learner that processes a raw training corpus to generate a vocabulary (a collection of tokens)
- A token segmenter that tokenizes a raw test sentence based on the generated vocabulary (ie a set of rules)

Byte-Pair Encoding (BPE)

Introduction

* Simplest & commonly used algorithm for tokenisation.
* Originally introduced as a text compression strategy.
* The algorithm depends on a pre-tokenizer that divides the training data into individual words

Widely adopted as a tokenisation technique in models like GPT, GPT-2, RoBERTa, BART & DeBERTa.

Breakdown of BPE

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens til k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

1. Pre tokenisation

- Input: The algorithm starts with a corpus of text data
- Pre-tokenisation: The corpus is pre-tokenized, usually by splitting the text into words. Pretokenisation can involve breaking the text at spaces, punctuation, or using more complex rules. → not fixed; varies from task-to-task
- After pretokenisation, the algorithm creates a list of all unique words in the corpus, along with their frequency of occurence.

2. Base Vocabulary

- The base vocabulary is initialized with all unique characters (or symbols) found in the list of unique words. For eg: if the word "hello" is in the corpus the symbols 'h', 'e', 'l', 'o' would be part of the

initial vocabulary
- Each word in the corpus is then represented as a sequence of symbols from this base vocabulary. for instance, "hello" would be represented as ['h', 'e', 'l', 'l', 'o']

3. Pair Merging
- Bigram counts: The algorithm counts the frequency of adjacent symbol pairs (bigrams) in the list of unique words. for eg: "hello" in the bigrams would be ('h', 'e'), ('e', 'l'), ('l', 'l'), ('l', 'o')
- Merging: The most frequent bigram is then merged into a new symbol, and the words in the corpus are updated to reflect this merge. eg: if ('l', 'l') is the most frequent bigram, it is merged into a new symbol, say 'll' and 'hello' would be updated to ['h', 'e', 'll', 'o']

This process continues iteratively until the desired vocabulary size is reached.

Example
- Lets us consider the toy corpus which consists of pre-tokenized text & its frequency.

Word	Frequency
cat	10
bat	5
bag	12
tag	4
cats	5

- Form the base vocabulary by taking all the characters that occur in the training corpus.
  - Base vocabulary: a, b, c, g, s, t
- Once we have the initial base vocabulary, we continue adding new tokens until we achieve the target vocabulary size by learning and applying merges.

- At each stage of the training, the BPE algorithm identifies the most frequent pair of existing tokens. This most frequent pair is then merged.

Tokens	Frequency
c, a, t	10
b, a, t	5
b, a, g	12
t, a, g	4
c, a, t, s	5

Token Pair	Frequency
ca	15
at	20
ba	17
ag	16
ts	5

- We split each word into its constituent characters (tokens) as per the base vocabulary.

Like in the above example we had : a, b, c, g, s, t
for all pairs of tokens like: aa, ab, ac, ag ... ba, bb .. ts, tt
we compute the pair with greatest frequency & merge it (like here a,t has a frequency = 20 (cat → 10, bat → 5, cats - 5))

- Select the most frequent pair (a, t) and merge them into a single symbol. Add this newly created symbol to the vocabulary [NOTE: we do not replace existing 'a' & 't' symbols. We just add a new 'at' symbol]
  - Vocabulary : a, b, c, g, s, t, at

Tokens	Frequency
c, at	10
b, at	5
b, a, g	12
t, a, g	4
c, at, s	5

Token Pair	Frequency
ag	16
cat	15
ba	12
bat	5
ats	5

- The 1st merge rule learned by the tokenizer is a, t → at and the pair should be merged in all the words of the corpus.

(As we keep expanding the vocabulary, we keep collecting the rules)

In the modified (token, frequency) table, we again check for all possible pairs & their frequencies, this time with the new vocab words (so (a, at), (b, at), (c, at), etc. will all be pairs)

- The most frequent pair at this stage is (a, g).

Tokens	Frequency
c, at	10
b, at	5
b, ag	12
t, ag	4
c, at, s	5

- The 2nd merge rule learned is a, g → ag. Adding that to the vocabulary & merging all leads us to:
  - Vocabulary: a, b, c, g, s, t, at, ag
- Now the most frequent pair is (c, at) so we learn the merge rule c, at → cat
- After 3 merges, the vocabulary and the corpus are as follows:
  - Vocabulary: a, b, c, g, s, t, at, ag, cat

Tokens	Frequency
cat	10
b, at	5
b, ag	12
t, ag	4
cat, s	5

- We keep iterating through these steps until the vocabulary reaches the desired size.

BPE Algorithm (Inference)

* To tokenize a text, we run each merge learned from the training data in a greedy manner, successively in the order we learned them, i.e.,

a, t → at
a, g → ag
c, at → cat

→ by token segmenter

* For example, the word "bags" would be tokenized as follows:
  - We begin by splitting the words into its constituent characters: bags → b, a, g, s
  - We go through the merge rules until we find one we can apply. Here we can apply the 2nd rule & merge a, g: b, ags → b, ags
  - When we have exhausted all the merge rules, the tokenization

process is complete. bags → b, ag, s
These are the tokens that will be fed to the model (like transformers, RNN, etc.)

Q) What if a token is not present in the vocabulary?
If the word being tokenized includes a character that was not present in training corpus, that character will be converted to the unknown token (<UNK>).

eg: moat → m, o, t → m, ot → <UNK>, at

* To avoid <UNK>, the base vocabulary must include every possible character or symbol. This can be extensive since there are about ~ 149K unicode symbols.
* GPT-2 & RoBERTa uses bytes as the base vocabulary (size 256) and then applies BPE on top of this sequence (with some rules to prevent certain types of merges)
* In practice, it is common to add a special end of word symbol "_" before space

WordPiece Tokenisation

Introduction

* Like BPE, WordPiece is used for subword tokenisation, but it employs a different approach to determine which symbol pairs to merge.

Unlike BPE, merge in WordPiece algorithm are determined by likelihood & not frequency

Adopted as a tokenization technique in language models like BERT, Distil BERT, Mobile BERT, Funnel Transformers & MPNET

Training Algorithm

* WordPiece algorithm uses special markers to indicate word-initial & word-internal tokens, which is model specific.
  - for BERT, ## is added as a prefix for any word-internal token
* To form the base vocabulary, split each word by adding the WordPiece prefix to all word-internal characters. For eg, the word "token" would be split as:
  - token → t, ##o, ##k, ##e, ##n
* At each stage, a score is computed for each pair of tokens in our vocabulary:

$$\text{score} = \frac{\text{freq of pair}}{\text{freq of 1st token} * \text{freq of 2nd token}}$$

The pair of tokens with the highest score is selected to be merged.

> How is this useful?
> If a pair is frequent, but so are the individual characters then the score is penalized. On the other hand if a pair is not frequent but neither are the individual characters, such pairs will still get a high score

* Add the merged pair to the vocabulary & continue to process until the vocabulary reaches the desired size.

Breakdown of WordPiece Algorithm

* Step 1 (Pre-tokenization) & 2 (Base Vocabulary) remain same as BPE.

3. Pair Merging:

- While BPE merges the most frequent symbol pair at each step, WordPiece chooses the symbols that maximize the likelihood of the training data when added to the vocabulary.
- Merge Criteria: The algorithm identifies the symbol pair whose merge would maximize the likelihood of the training data. This is determined by selecting the pair with the highest score, defined as the probability of the merged symbol divided by the product of probabilities of its individual components.
- The selected symbol pair is then merged into a new symbol & the vocabulary is updated accordingly.

The process is repeated iteratively until the size of the vocabulary is reached.

Example

- Lets look at the toy corpus, which we will use to train the WordPiece tokenizer.
- To create the initial vocabulary, we break down each word in the training corpus into its constituent letters and prepend the WordPiece prefix to the word-internal characters.
- By retaining only a single occurence of each element, we get the following vocabulary:
  - ##d, ##c, ##f, ##g, ##i, ##l, ##n, ##o, ##r, ##s, ##u, ##w, $, s

Word	Frequency
sunflower	1
sun	2
flower	1
flow	1
flowers	1
flowing	2
flows	2
flowed	1

Word	Frequency
s, ##u, ##n, ##f, ##l, ##o, ##w, ##e, ##r	1
s, ##u, ##n	2
f, ##l, ##o, ##w, ##e, ##r	1
f, ##l, ##o, ##w	1
f, ##l, ##o, ##w, ##e, ##r, ##s	1
f, ##l, ##o, ##w, ##i, ##n, ##g	2
f, ##l, ##o, ##w, ##s	2
f, ##l, ##o, ##w, ##e, ##d	1

- We calculate the score for each pair.
- Select the pair with the highest score to be merged. → S, ##u

Token Pair	Score
S, ##u	0.33 → $\frac{2}{6\times3} = \frac{1}{6}$
##u, ##n	0.2
##n, ##f	0.2
##f, ##l	0.11
##l, ##o	0.11
...	...
##e, ##r	0.25
##e, ##d	0.25

- S + ##u → Su

NOTE: If we merge (S, ##u), the resultant token is 'Su' & not '##su' because S is the 1st token. On the other hand, if we had merged 2 internal tokens like (##u, ##n) the resultant token would've been ##un

↳ NOTE: S & ##S are counted as same word for the purpose of single word frequency calculation

- We add the merged pair to the vocabulary and apply the merge to the words in our corpus.
- Vocabulary: ##d, ##e, ##f, ##g, ##i, ##l, ##n, ##o, ##r, ##s, ##u, ##w, $, S, Su

Word	Frequency
su, ##n, ##f, ##l, ##o, ##w, ##e, ##r	1
su, ##n	2
f, ##l, ##o, ##w, ##e, ##r	1
f, ##l, ##o, ##w	1
f, ##l, ##o, ##w, ##e, ##r, ##s	1
f, ##l, ##o, ##w, ##i, ##n, ##g	2
f, ##l, ##o, ##w, ##s	2
f, ##l, ##o, ##w, ##e, ##d	1

Corpus ↑

- We continue this process for a few more steps
- Compute score for all pair of tokens
- The best score is shared by (##e, ##r) & (##e, ##d). Let's say we select (##e, ##r) as the best pair & merge them

##e + ##r = ##er

- So, now we have
- Vocabulary: ##d, ##e, ##f, ##g, ##i, ##l, ##n, ##o, ##r, ##s, ##u, ##w, $, S, Su, ##er

Token Pair	Score
su, ##n	0.2
##n, ##f	0.2
##f, ##l	0.11
##e, ##r	0.25
...	...
##e, ##d	0.25

Corpus ↓

Word	Frequency
su, ##n, ##f, ##l, ##o, ##w, ##er	1
su, ##n	2
f, ##l, ##o, ##w, ##er	1
f, ##l, ##o, ##w	1
f, ##l, ##o, ##w, ##er, ##s	1
f, ##l, ##o, ##w, ##i, ##n, ##g	2
f, ##l, ##o, ##w, ##s	2
f, ##l, ##o, ##w, ##e, ##d	1

- Next (##e, ##d) has the highest score & is merged

  ##e + ##d = ##ed

Token Pair	Score
su, ##n	0.2
##n, ##f	0.2
##f, ##l	0.11
...	...
##e, ##d	1

- Vocabulary. ##d, ##c, ##f, ##g, ##i, ##h, ##n, ##o, ##r, ##s, ##u, ##w, f, s, su, ##er, ##ed

- We continue this process until we reach the desired vocabulary size.

Tokenization Algorithm (Inference)

* Tokenization in WordPiece differs from BPE
* In WordPiece, we only retain the final vocabulary and not the merged rules learned during the process.
* To tokenize a word, WordPiece identifies the longest subword available in the vocabulary and then performs the split based on that subword.

Example

- Lets take the word - fused
- If we use the vocabulary learned in the example, for the word "fused" the longest subword starting from the beginning that is present in the vocabulary is 'f', so we split there & get f, ##used
- for ##used, the longest subword in the vocabulary is '##u', so we split it into [##u, ##sed]
- Next for ##sed, the longest subword in the vocabulary is '##s' so we split into [##s, ##ed]
- Finally, ##ed is a complete token in the vocabulary so no further split is needed.

- So, the full breakdown of "fused" will be:
[f, ##u, ##s, ##ed]

- If it is not possible to find a subword in the vocabulary, the whole word is tokenized as unknown

funny ⇒ f, ##u, ##n, ##n, ##y
present absent
↳ so funny is replaced by <UNK>

- In BPE, only the missing token is replaced by <UNK>
funny ⇒ f, ##u, ##n, ##n, <UNK>

Unigram Tokenization

Introduction

* Observation: There exists ambiguities in subword segmentation, as a single word or sentence can be divided into different subwords even when using the same vocabulary.
* Problem: The BPE algorithm doesn't support multiple segmentations because it uses a greedy & deterministic approach.
* Solution: Introduce multiple subword candidates during the training.
* The algorithm is based on Expectation-Maximization (EM) algorithm.

Commonly employed in SentencePiece, the tokenization method used by models such as ALBERT, T5, mBART, Big Bird & XLNET

Breakdown of Unigram LM Tokenization

1. Initializing the Base Vocabulary.

- This method is different from the previous tokenisation methods. In BPE & WordPiece, we have a base vocabulary & our goal is to keep expanding the vocabulary until we reach a limit. Here, we have a massive base vocabulary & we keep on shrinking the vocabulary until we reach a threshold.
- There are various methods for creating the seed vocabulary. A common approach is to include all characters and the most frequent substrings found in the corpus. (Can also run BPE exhaustively to create this)

2. Log-Likelihood Loss Computation

- Unigram Language Model: At each training step, the algorithm uses a unigram language model, which assumes that each token in a vocabulary is independent of the others.
- The algorithm calculates the log-likelihood loss over the entire training data based on the current vocabulary. The loss measures how the current set of symbols can represent the training data.
- The goal is to minimize this loss, meaning the algorithm aims to find the smallest & most effective vocabulary that is still adequate to represent the training data.

3. Finding the candidates for removal

- For each symbol in the vocabulary, the algorithm calculates the potential increase in log-likelihood loss that would occur if that symbol were removed.
- Symbols that contribute less to the overall representation of the data (i.e., those that cause the smallest increase in

loss when removed)

4. Progressive Vocabulary Pruning
- The algorithm removes a small percentage (typically 10% - 20%) of the symbols that have the least impact on the log-likelihood loss. These are the symbols for which the increase in the training loss is lowest if they're removed.

This pruning process is repeated iteratively. After each round of pruning, the log-likelihood loss is recalculated with the updated vocabulary, and the process continues till the vocabulary reaches the desired size.

Example
- Let us consider this toy corpus

Word	Frequency
run	3
bug	5
fun	13
sun	10

- We include all the possible strict substrings in the initial vocabulary.
  - Vocabulary: r, u, n, ru, un, b, g, bu, ug, f, fu, s, su
- We will start the first iteration of the algorithm.
- We compute the frequency of different tokens in the vocabulary.

Token	r	u	n	ru	un	b	g	bu	ug	f	fu	s	su
Freq	3	31	26	3	26	5	5	5	5	13	13	10	10

- The probability of a specific token is calculated by dividing its frequency in the original corpus by the total sum of frequencies of all tokens in the vocabulary.
  eg: the probability of the subword 'su' is $\frac{10}{155}$

Token	r	u	n	ru	un	b	g	bu	ug	f	fu	s	su
Freq	0.0194	0.2	0.1677	0.0194	0.1677	0.0323	0.0323	0.0323	0.0323	0.0839	0.0839	0.0645	0.0645

- To tokenize a given word using the Unigram model, we first consider all possible segmentations of the word into tokens.
- Since the Unigram model treats all tokens as independent, the probability of a specific segmentation is simply the product of the probabilities of each token in that segmentation.

- Let us consider the word run (→ All the tokens in this split are already vocabulary entries)
  - Possible segmentations: (r, u, n); (r, un); (ru, n)
  - Probability:

$P(r,u,n) = P(r) \times P(u) \times P(n) = 0.0194 \times 0.2 \times 0.1677 = 0.000650676$ (→ from the probability table above)

$P(ru,n) = P(ru) \times P(n) = 0.0194 \times 0.1677 = 0.00325338$

$P(r,un) = P(r) \times P(un) = 0.0194 \times 0.1677 = 0.00325338$

- The tokenization of a word using the Unigram model is chosen as the one with the highest probability among all possible segmentations.
  - The word run could be tokenized as either (ru, n) or (r, un), Lets say we select (ru, n)

In practice, finding all possible segmentation is very inefficient. The Viterbi Algorithm is used to find the most probable segmentation of a word/sentence from the set of possible segmentation candidates

- At each stage of training, the loss is calculated by tokenizing every word in

Word	Freq	Split	Score
run	3	ru, n	0.00325338
bug	5	bu, g	0.00104329
fun	13	fu, n	0.01407003
sun	10	su, n	0.01081665

the corpus using the current vocabulary.

- $Loss = \sum freq * (-\log(P(word)))$

$= 3 \times (-\log(0.00326338)) + 5 \times (-\log(0.00104329)) + 13 \times (-\log(0.01407003)) + 10 \times (-\log(0.01081665)) = 66.102$

↳ loss for the entire training corpus

M-Step (EM algorithm) (Maximization)

- Our goal is to reduce the vocabulary size.
- To determine which token to remove, we will calculate the associated loss for each token in the vocabulary that is not an elementary token, then compare these losses

For example, lets remove the token un

Token	r	u	n	ru	b	g	bu	ug	f	fu	s	su
Freq	0.0194	0.2	0.1677	0.0194	0.0323	0.0323	0.0323	0.0323	0.0839	0.0839	0.0645	0.0645

- Possible segmentations for the word run: (r,u,n), (ru,n), ~~(r,un)~~
  - Probability:
    - $P(r,u,n) = P(r) \times P(u) \times P(n) = 0.00065076$
    - $P(ru,n) = P(ru) \times P(n) = 0.00325338$

- Loss (after removal of token un)

$= 3 \times (-\log(0.00326338)) + 5 \times (-\log(0.00104329)) + 13 \times (-\log(0.01407003)) + 10 \times (-\log(0.01081665)) = 66.102$ (unchanged)

Word	Freq	Split	Score
run	3	ru, n	0.00325338
bug	5	bu, g	0.00104329
fun	13	fu, n	0.01407003
sun	10	su, n	0.01081665

↳ so removal of 'un' doesnt affect the log loss so we can remove it.

- For the 1st iteration, removing any token would not affect the loss.
- So, we randomly select the token 'un' and remove it from the vocabulary. Then we proceed with the 2nd iteration

Token removed from vocabulary	Loss
ru	66.102
un	66.102
bu	66.102
ug	66.102
fu	66.102
su	66.102

↳ In practice, the algorithm removes 10-20% of vocabulary entries at one go.

↳ Expectation

E-Step (EM Algorithm)

- We recompute the probabilities after removal of token 'un'

Token	r	u	n	ru	b	g	bu	ug	f	fu	s	su
Freq	3	31	26	3	5	5	5	5	13	13	10	10

← Bigram probabilities will change because we removed a token so the denominator changes

Token	r	u	n	ru	b	g	bu	ug	f	fu	s	su
Freq	0.0233	0.2403	0.2016	0.0233	0.0388	0.0388	0.0388	0.0388	0.1008	0.1008	0.0775	0.0775

Word	Freq	Split	Score
run	3	ru, n	0.00469728
bug	5	bu, g	0.00150544
fun	13	fu, n	0.02032128
sun	10	su, n	0.015624

↳ Best splits after removing 'un'

$$\text{loss} = 3\times(-\log(0.00469728)) + 5\times(-\log(0.00150544)) + 13\times(-\log(0.02032128)) + 10\times(-\log(0.015644)) = 61.155$$

↳ current loss after removing 'un'

M-Step

- Again, we compute the impact of each token on the loss
- Assuming we are removing only one token at each step, we can remove either 'bu' or 'ug' from the vocabulary at this iteration.

Token removed from vocabulary	Loss
ru	63.0126
bu	61.155
ug	61.155
fu	69.205
su	67.347

↳ since they don't affect the loss at all

These kinds of tokenization strategies are very useful in handling low resource languages i.e., languages that are not present during training time & the model is unaware of them. If we don't have the resources to pre-train a model from scratch, we can use these tokenization techniques.

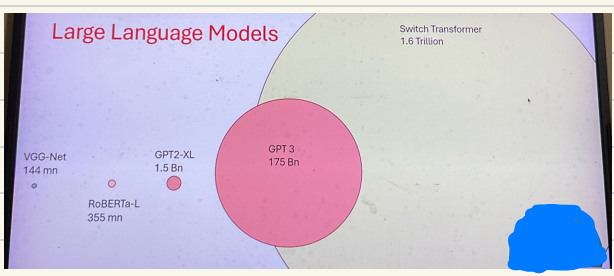
Lecture 10.1 - Mixture of Experts (MoE)

LLMs

* As the LLM race began with with the advent of transformers, a big paradigm shift was moving towards pre-training & fine-tuning.

- Pre training - Train a model on a huge dataset with a huge amount of compute.
- fine tuning - fine tune the pretrained model on a specific task using a small dataset (using efficient methods like PEFT, LoRA)

* This paradigm already existed in CV where we had VGG Net, ResNet etc which were first trained on a huge ImageNet dataset & later fine-tuned for specific tasks by using the same backbone architecture with a classifier on top of it.
* This method became even more popular with transformers



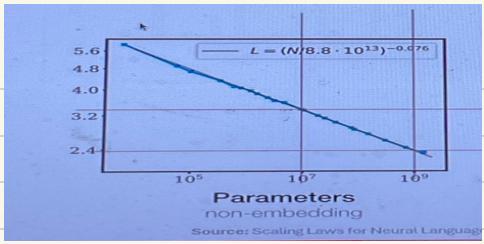
Model	Size	Approx Training Time
RoBERTa-L	356-M	1k10 A100 days (1k10 A100s on 1 day)
GPT-3	175 B	60k A100 days (2k A100 on 30 days)
Switch Transformer	1.6T	540k A100 days (2k A100 in 9 months)

These trillion parameter models are neither feasible nor efficient

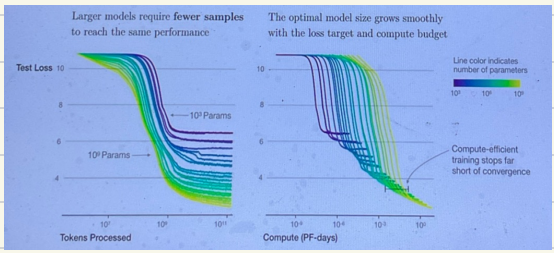
Q) Why is large model size even important?

Neural Scaling Laws

↳ Performance improves smoothly as we increase the compute, dataset size, or the model size



↳ Large models are more sample efficient - given a fixed computing budget, training a larger model for fewer steps is better than training a smaller model for more steps

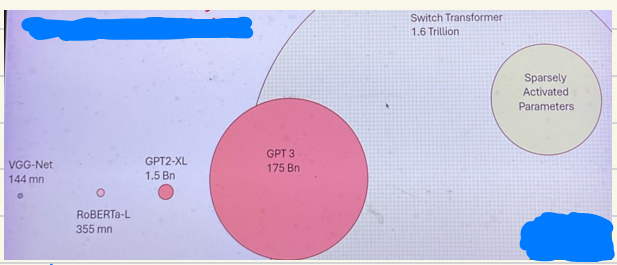


② How to efficiently increase model size?

A) Mixture of Experts (MoE)

Idea - We have a lot of parameters but we will not activate all of them - neither in training nor during inference

At any point in time, only a subset of params are active for a given input.

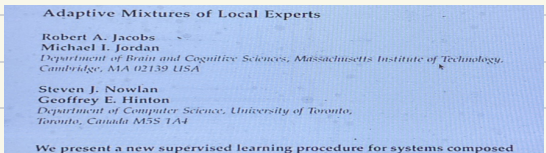


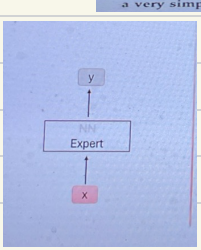
Mixture of Experts (MoE)

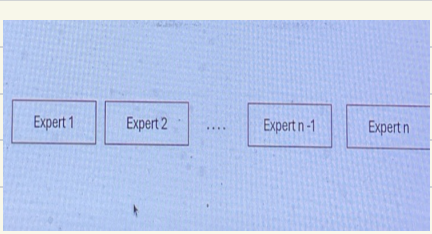
*) Concept was introduced in 1991

*) Introduced by Geoffrey Hinton's group

*) Typically consists of a Neural Net (or any other model) called expert which takes in an input X and gives an output Y.



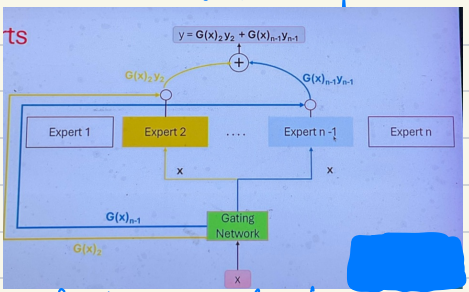




*) In a MoE, we have n such experts (which could be simple MLPs, SVMs, etc.)

*) Additionally, we have a Gating Network.

*) When we receive an input (say x), the Gating Network selects which expert to activate (like expert 2 & expert n-1 in this image)

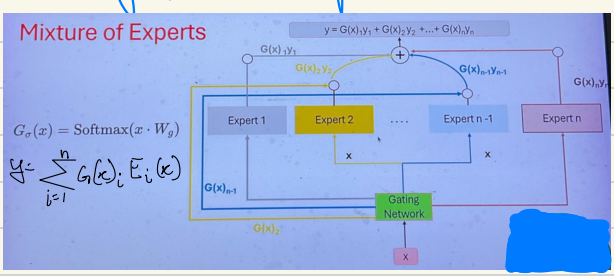


*) The input is passed to these 2 experts which compute their respective outputs (y_2 & y_{n-1}).

*) These outputs are then multiplied with their Gating Probabilities ($G(x)_2$ & $G(x)_{n-1}$) & add them to get the output.

*) In general, when all the outputs are active we have

$y = G(x)_1 y_1 + G(x)_2 y_2 \cdots G(x)_n y_n$



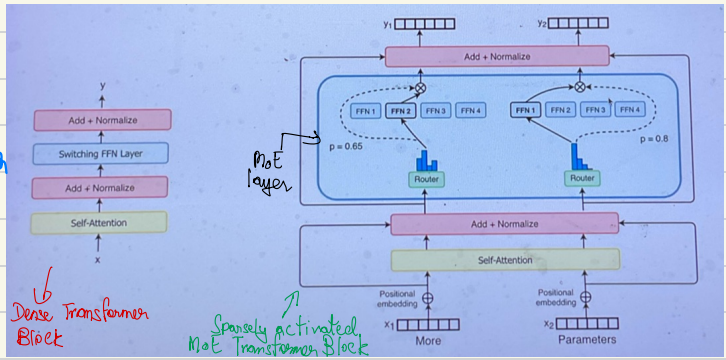
↳ This was the architecture introduced in 1991. Their idea was not to reduce the computation cost but to increase the capacity of the model.

MoE - Chronology

- Mixture of Experts Model [Jacobs et al., 1991; Jordan and Jacobs, 1994; Jordan et al., 1997; Tresp, 2001; Collobert et al., 2002;]
- MoE layer in Deep Learning [Eigen et al., 2013; Shazeer et al., 2017; Dauphin et al., 2017; Vaswani et al., 2017]
- MoE layer in Transformer based LLMs [Fedus et al., 2021; Du, Nan, et al., 2021]
- Mixtral-8x7B [Jiang et al. 2024] - Apache 2.0 license surpasses GPT-3.5 Turbo, Claude-2.1, Gemini Pro, and Llama 2 70B - chat model on human benchmarks

Content credits: https://www.youtube.com/watch?v=Tw

MoE as a layer



* The image on the left shows a transformer block.
* We first perform self-attention & then the residual connection add & normalization.
* The outputs are then passed through a router which will assign a probability of selecting an expert for both the tokens (independently). eg: for 'more' expert 2 is selected with a probability of 0.65 & for 'parameters' the expert 1 with probability 0.8.
* Then we multiply the outputs with the probabilities & pass it to the next layer which again performs residual add & normalize (independently for each token) & we get the outputs y_1 & y_2.

Q) How do we reduce the computational cost?
For an embedding, only 1 of the 4 experts are active. So, the computational cost is almost the same as in dense network (except for the router)

MoE layer - What happens underneath?

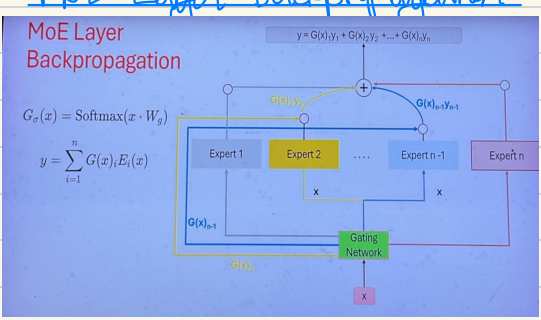
MoE's were introduced to increase the model capacity & not to speed up training. Hence, the router would choose which expert to go to in the initial steps itself. This didn't improve the training efficiency in any way. However, when MoE is used as a layer (with multiple transformer blocks), i.e., when performed on individual tokens rather than on the inputs itself (to choose one model or another), then there is a chance that each FFN will need to

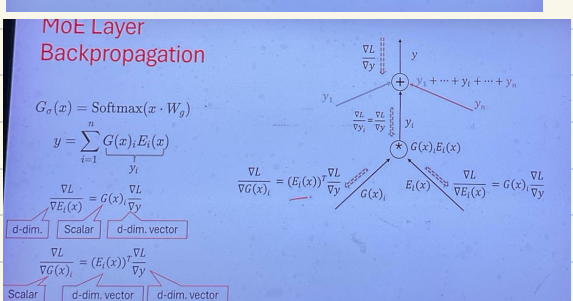
process only a subset of tokens (eg: if there are 1000 tokens & 100 experts then each expert might need to process only 100 tokens if uniformly distributed). Each token might get assigned to a different FFN expert in each transformer block.

Q) Why are the parameters increasing?
Because FFNs are dense NN layers which constitute a major portion of the parameters of a transformer. So, increasing the no. of FFN experts significantly increases the no of parameters.

Q) Why is training efficient?
Lets say each FFN has 1M parameters & we have 8 such expert FFNs (so 8M params). But only 1 gets activated at any point of time so the no of matrix multiplications is reduced (Also called conditional activation)

MoE Layer Backpropagation





* Lets go back to the dense MoE model.
* Here, all the experts are active.
* Gating network $G(x)$ gives us the distribution probability for each token (for selecting an expert)

Output $y = \underbrace{\sum_{i=1}^{n} G(x)_i E_i(x)}_{y_i}$

* Here is the computational graph of the MoE

During backprop,

$$\frac{\nabla L}{\nabla y_i} = \frac{\nabla L}{\nabla y} \quad \text{(for each } i\text{)}$$

$\frac{\nabla L}{\nabla y_i}$ is same for all y_i because $y = y_1 + y_2 \cdots y_i \cdots y_n$

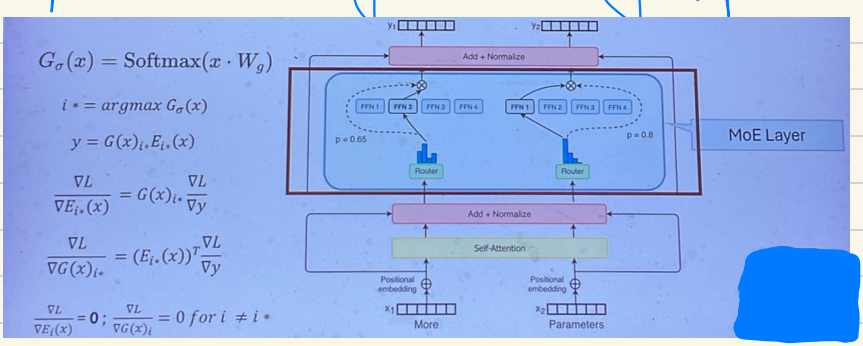
$$\frac{\nabla L}{\nabla y_i} = \frac{\nabla L}{\nabla y} \cdot \left(\frac{\nabla y}{\nabla y_i}\right) \rightarrow 1 \quad \left(\frac{\nabla y}{\nabla y_i} = \frac{\nabla (y_1 + y_2 \cdots y_i \cdots y_n)}{\nabla y_i} = \frac{\nabla y_i}{\nabla y_i} = 1\right)$$

$y_i = G(x)_i \, E_i(x)$

$$\frac{\nabla L}{\nabla E_i(x)} = \frac{\nabla L}{\nabla y_i} \frac{\nabla y_i}{\nabla E_i(x)} = G(x)_i \frac{\nabla L}{\nabla y_i} = G(x)_i \frac{\nabla L}{\nabla y}$$

Similarly, $\frac{\nabla L}{\nabla G(x)_i} = (E_i(x))^T \frac{\nabla L}{\nabla y}$

Sparse MoE Layer – Greedy Expert Selection



$i*$ is the expert with max probability

The derivatives w.r.t E and G are:

$$\frac{\nabla L}{\nabla E_{i*}(x)} = G(x)_{i*} \frac{\nabla L}{\nabla y}$$

$$\frac{\nabla L}{\nabla G(x)_{i*}} = (E_{i*}(x))^T \frac{\nabla L}{\nabla y}$$

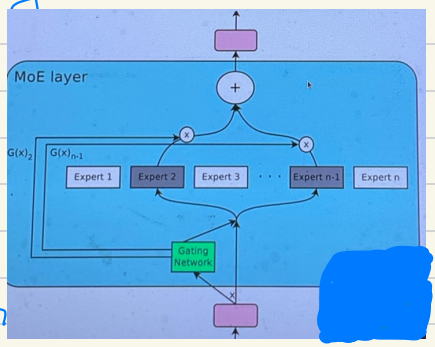
$\frac{\nabla L}{\nabla E_i(x)} = 0; \frac{\nabla L}{\nabla G(x)_i} = 0$ for $i \neq i*$

↳ since derivatives for all other i's are 0 so this speeds up the backprop & the training.

The above sparse MoE layer was used in Switch Transformer.

Sparse MoE Layer – Noisy Top-K Gating

*) Introduced by Noam Shazeer in LSTM
*) Came before Switch Transformers.



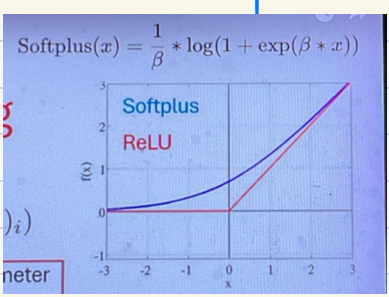
Previously,

$H(x)_i = (x \cdot W_g)_i$ → i^{th} logit of the Gating Router

Introduced a noise and a learnable parameter to control the amount of noise. So now

$$H(x)_i = (x \cdot W_g)_i + \text{StandardNormal}() \cdot \text{Softplus}((x \cdot W_{noise})_i)$$

Learnable parameter (similar to mean)



↳ Learnable parameter (similar to std. dev)

↓

Std deviation should be +ve so we pass it through softplus

Q) Why do we add the noise?
A) The claim was that it improved the learnability of MoE making it more stable.

$$\text{KeepTopK}(v,k)_i = \begin{cases} v_i & \text{if } v_i \text{ is in the top k elements of } v \\ -\infty & \text{otherwise} \end{cases}$$

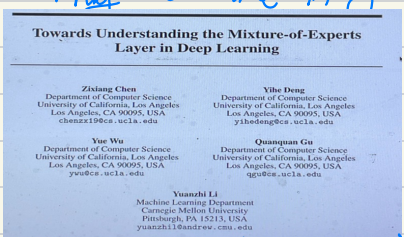
↳ ensures 0 probability after softmax

$$G(x) = \text{Softmax}(\text{KeepTopK}(H(x),k))$$ → Has k non-zero elements

Q) Why don't MoE's collapse in practice when they all have:
(i) Same architecture
(ii) Same initialization scheme (randomly though)
(iii) Trained with same optimizer

What's stopping all of them from learning the same thing?

A) There is a paper which tries to understand the MoE layer in deep learning & they address this question.


Towards Understanding the Mixture-of-Experts Layer in Deep Learning

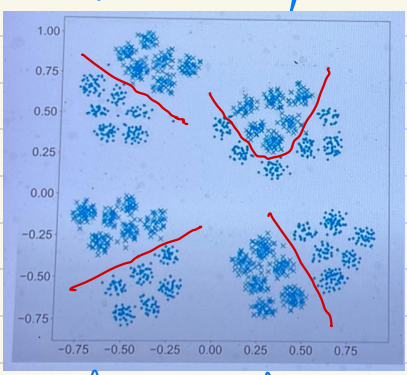
Zixiang Chen
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
chenzx19@cs.ucla.edu

Yihe Deng
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
yihedeng@cs.ucla.edu

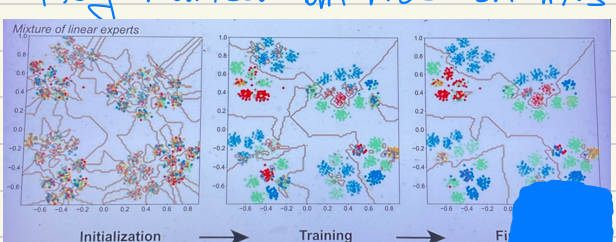
Yue Wu
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
yuw@cs.ucla.edu

Quanquan Gu
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
qgu@cs.ucla.edu

Yuanzhi Li
Machine Learning Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
yuanzhil@andrew.cmu.edu

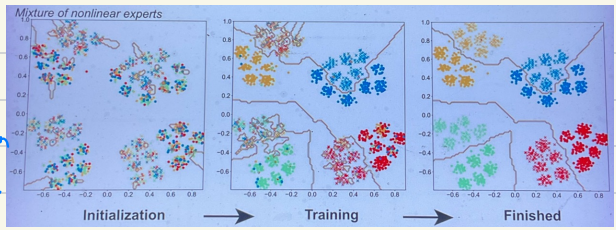


For this, they create a synthetic dataset (dim=50) which looks like this image in the 2-D space. It consists of 4 clusters which are linearly seperable (in higher dimension). Ideally, an MoE of 4 experts should be able to deal with this problem with each expert able to classify one type cluster & router should be able to identify which expert to send the data point to.
They trained an MoE on this dataset.



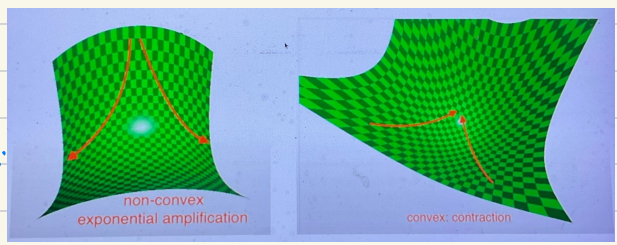
Initially, the considered a linear MoE (color of dot indicates which expert it was assigned to and the lines are the boundaries learned)
At first, we see that the router has not learned meaningfully. We see points of different (true) clusters get assigned to different same experts & vice versa.

On the other hand, when some non-linearity is introduced, we see that there is a clear seperation. The router has learned correctly to assign each data point to the right expert & the boundaries are well defined.



Q) So what happened here?

A) They proved theoretically that non-linearity is essential: If we have a linear layer, the loss function becomes convex & all experts even though initialized differently, converge to the same minimum.



On the other hand, with non-linearity, even if we start with similar points, non-linearity ensures that they diverge to different points in space.

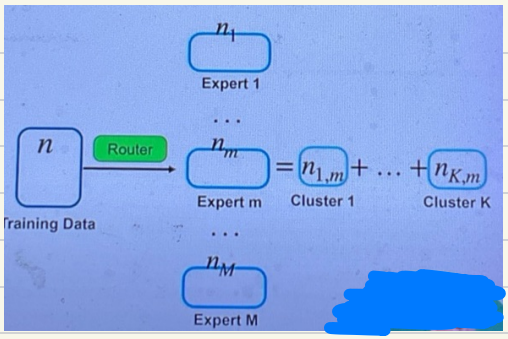
Learning Dynamics of Expert & Router

Q) Does the expert gets trained first & then the router learns to assign each data point to the right expert or will the router learn to assign points first & then the experts learn their respective boundaries.

A) The paper showed empirically that learning of MoE's happens in 2 stages:

1) Exploration Stage: Router is nearly untrained & randomly assigns data points. Experts start to diverge.

2) Router learning stage: Router learns to dispatch the points to the right expert.



Q) How do they quantify if the router is routing to the right expert?

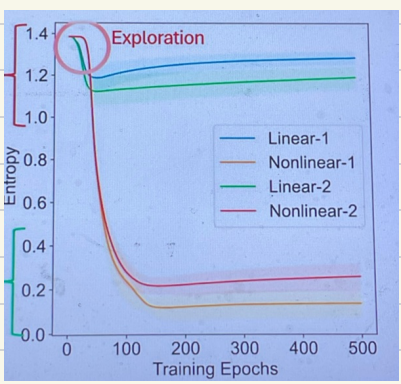
A) They use the notion of entropy i.e., the amount of randomness. If a router is assigning equal probability to all experts then entropy is high & vice versa.

$n_{k,m}$ = # of samples from cluster k routed to expert m

n_m = # of samples routed to expert $m = \sum_{k=1}^{K} n_{k,m}$

n = # of total samples $= \sum_{m=1}^{M} n_m$

$$\text{entropy} = -\sum_{m=1, n_m \neq 0}^{M} \frac{n_m}{n} \sum_{k=1}^{K} \frac{n_{k,m}}{n_m} \cdot \log\left(\frac{n_{k,m}}{n_m}\right)$$



From the graph here we see that initially, the entropy is quite high (meaning router is routing points randomly to experts) and then as the learning progresses, the entropy reduces (meaning now the router is routing a given sample to only one of the experts)

Pros & Cons of Sparse MoE Layer

Pros

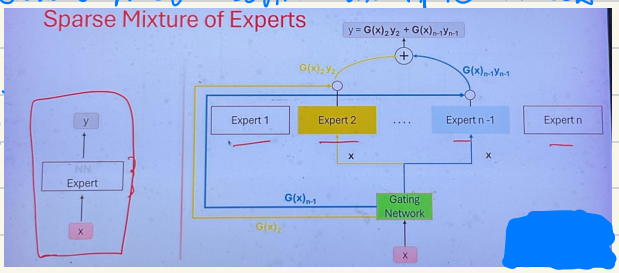
- Increased model params
- Efficient pre-training due to conditional (sparse) activation
- Faster inference (same reason i.e., sparse activation)

Cons

- Unstable training
  - → Same architecture, different random initialization may cause huge difference in training loss (Designing stable MoE is crucial)
  - → Router collapse still possible - i.e., router sends all data to 1 expert & others are useless
- High memory requirement - all params need to be loaded into v-RAM.

Lecture 10.2 - Mixture of Experts - II

* In MoE, we replace 1 Neural Net with multiple models (called experts)
* Model size has increased 'n' times.
* The task of the Gating Network is to assign probabilities to different experts to choose accordingly.



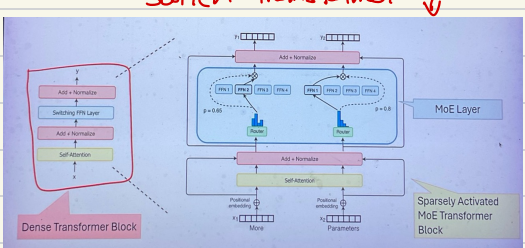
* In this image, expert 2 & expert n-1 are selected. We pass the input through the experts to get the output. Then we multiply the gating probabilities (i.e., the probability of choosing that expert) to the outputs & sum them up to get the final output.

Compute increase: $C \rightarrow 2C + \text{Routing}$
- ↳ compute for 1 model
- ↳ Here we choose 2 experts so 2C
- ↳ This is negligible

Model Size increase: $M \rightarrow n \times M$

Switch Transformer ↓

* MoE's are used in Transformers by replacing the FFN layer by an MoE (or sparse MoE) layer in the Transformer block. These blocks are then stacked on top of each other like any transformer. The gating network & experts in each block are independent.



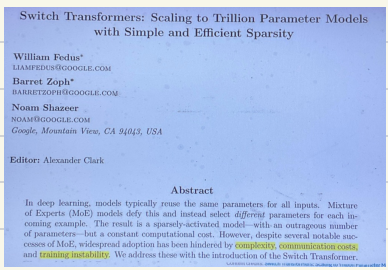
Model Size increase: $M \sim 4M$ → since most params are in the FFN layer itself (Attention layers are much smaller)

Compute Increase: $C \rightarrow C + \text{Routing}$
- ↳ Almost negligible

*) If we stack 8 transformer blocks on top of each other then the total no of experts (assuming 4 experts in each block as above) we have is 32 (4x8). Each token has 4^8 possible paths
*) A token can get routed to different experts in different layers to get the optimal path.

Switch Transformer Layer

*) Introduced by the Google Brain team.
*) Highlight 3 issues that they try to address in the paper - complexity, communication costs & training instability.
*) Routes greedily to only 1 expert.


Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

William Fedus*
LIAMFEDUS@GOOGLE.COM
Barret Zoph*
BARRETZOPH@GOOGLE.COM
Noam Shazeer
NOAM@GOOGLE.COM
Google, Mountain View, CA 94043, USA

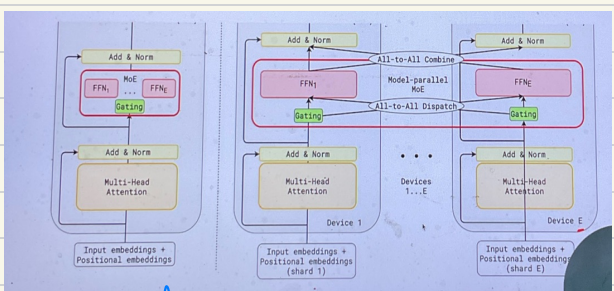
Editor: Alexander Clark

Abstract

In deep learning, models typically reuse the same parameters for all inputs. Mixture of Experts (MoE) models defy this and instead select *different* parameters for each incoming example. The result is a sparsely-activated model—with an outrageous number of parameters—but a constant computational cost. However, despite several notable successes of MoE, widespread adoption has been hindered by complexity, communication costs, and training instability. We address these with the introduction of the Switch Transformer.

Expert parallel for sparse MoEs

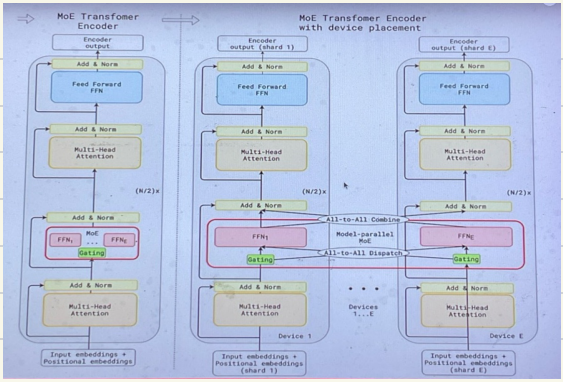
How are these huge experts trained?



*) On the left side image, all experts are on the same device. If we keep on increasing the experts, it becomes infeasible to put them on a single device.
*) So they scale it to a number of devices. If there are E experts, they use E devices (right side image)
*) Each expert (shard) is loaded onto device. The other layers are replicated.
*) This increases the communication cost since tokens are independently routed in each block so we need to collect them all at the end of the block.

How does the entire architecture look like?

* They don't replace all the FFNs with MoEs but do it alternatively. The image here shows one block, which consists of 2 transformer blocks, which is replicated N/2 times to get N layers.



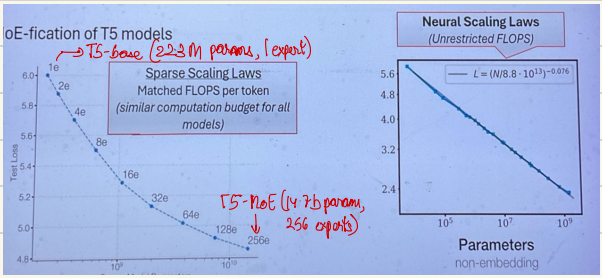
Results

- MoE-fication of T5 models
- The image here shows how test loss decreases as the model size increases.

But they also ensure that the no of computations also remains similar. How?

They matched flops per token - meaning - because each token passes through only 1 expert so the no of activated neurons is almost the same (Even though there is additional overhead communication cost, but the computation cost is approximately the same)

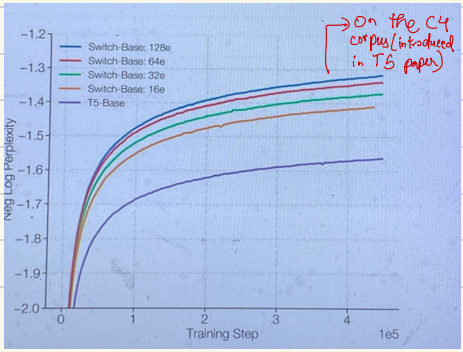
- On the other hand, if we increase the no of params with unrestricted FLOPS then even though we get better performance according to Neural Scaling Laws it requires more compute.



Sample Efficiency

- Sample efficiency - ability of a model to predict accurately with a minimal amount of samples.
- Here we measure the perplexity against the no of training steps

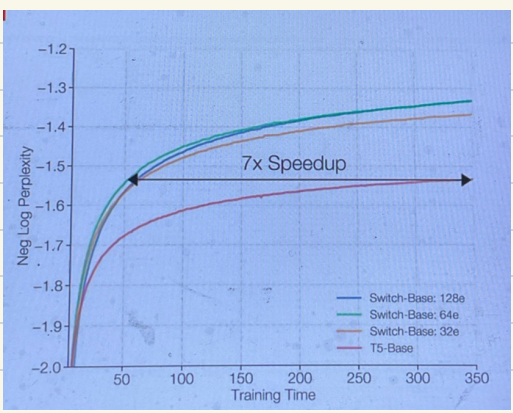
(Remember we want perplexity low; so -ve of perplexity should be high)



- Here we see that sparse models are much more sample efficient (i.e., for the same no of steps (& thus same no of training samples) it has a higher -ve log perplexity)

Switch transformer has
- → Better asymptotic performance
- → Improved sample efficiency
- → Diminishing returns as we increase # of experts

- Even with diminishing returns, given a fixed amount of compute, sparse MoEs perform well.



Training Time

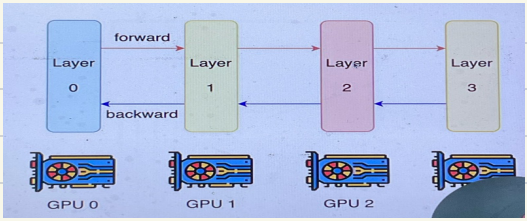
- flops per token are matched but there is additional clock time due to:
  1. Extra computation cost
  2. Router computation

- Still if we consider the same level of performance (i.e., perplexity) a Switch Transformer (128e) achieves the same levels in 7x faster time.

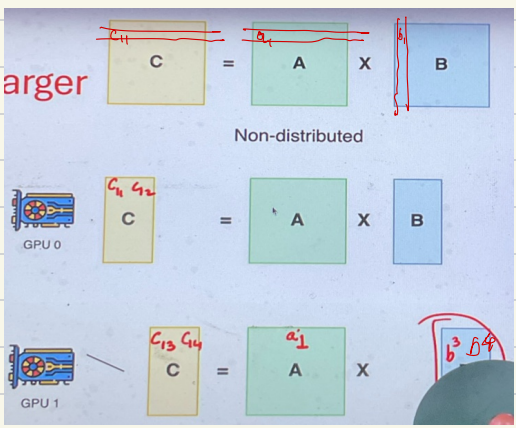
Comparison with larger dense model

- In expert parallel, we anyways use multiple devices instead of one by sharding the experts so we are increasing the no of resources.
- In model parallelism, the MoE itself is modular so we are able to shard it i.e., it has 'n' modular components each of which can be loaded into a different device. (Same can be done for a single FFN as well)

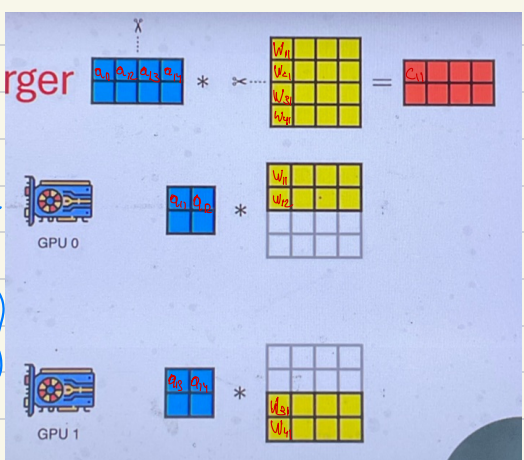
Model Parallelism for Large Dense Model



* Pipeline Parallelism
  - Shard the model layer-by-layer
  - Layer 1 on 1 device, layer 2 on 2nd device & so on

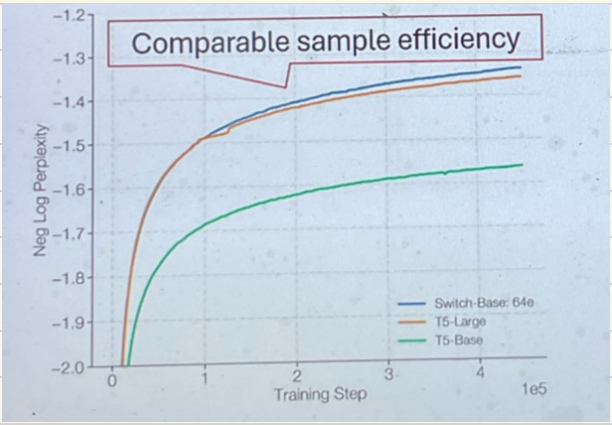


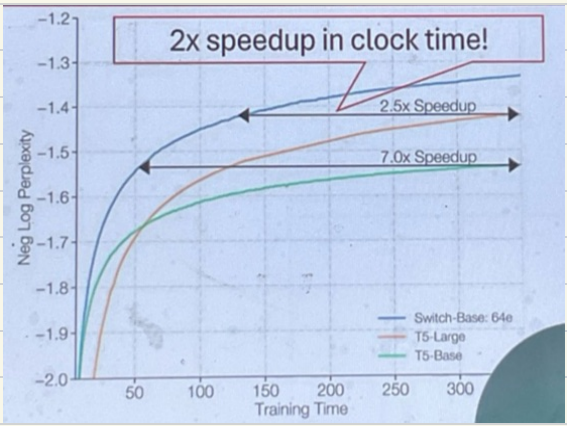
* Tensor Parallelism
  1) Columnwise splitting
  - Any FFN typically has matrix multiplications
  $C_{11} = a_1$ (row) $\times b_1$ (column)
  - In columnwise splitting, the weight is split into 2 parts columnwise
  Half the columns on device 1 and the rest on another. A is replicated.
  So, as per the image, C_{11} & C_{12} are computed on GPU-0 & C_{13}, C_{14} are computed on GPU-1.



  2) Row-wise splitting
  - Both the input & weight matrices are split.
  - To calculate any position (say $C_{11} = a_{11}W_{11} + a_{12}W_{21} + a_{13}W_{31} + a_{14}W_{41}$)
  - One half of the computation ($a_{11}W_{11} + a_{12}W_{21}$) happens on GPU-0 & second half ($a_{13}W_{31} + a_{14}W_{41}$) happens on GPU-1. (The same thing happens for any C_{ij})
  - finally, we add the 2 parts to get the output

Switch Transformer - Comparison with similar size dense models





→ Comparison with T5-large (FFN), similar model size but uses model parallelism, 3.5x more FLOPS per token

→ Comparable sample efficiency

→ Training time is much faster however for switch transformer. This is why, even if we have higher compute, we would want to train a sparse model compared to a dense one – comparable sample efficiency & faster clock time. (Even though this isn't exactly a fair comparison since the dense model has significantly more FLOPS per token i.e., more computations)

Switch Transformer - Issues Addressed

1) Complexity of MoE } - How? : Greedy Top 1 routing
2) Communication cost }

↳ If the output exists on multiple devices then we need to collect them to add and get the final output. But by using only 1 expert we avoid the need for this communication

3) Training instability
↳ Improved training techniques
(i) Differentiable load balancing loss (avoids router collapse)

(ii) Selective Precision
(iii) Reduced initialisation scale
(iv) Slower learning rate warmup
(v) Higher regularization of experts

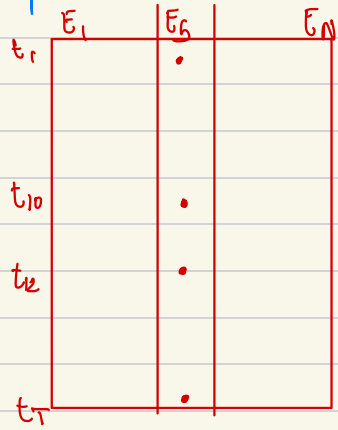
Load Balancing Loss

*) Work on load balancing loss already existed in Noam Shazeer's paper on MoE with LSTM layers with noisy top 2 routing. But it was complicated.
*) Switch transformer simplified this
*) Intuitively speaking, switch transformer changes the loss fn to ensure that all experts are activated equally i.e., still only 1 expert for 1 token but when training with a batch of tokens, then, on average, we want all experts should be activated roughly equally.
*) To ensure this (and thus avoid router collapse), we add an auxiliary loss term which minimizes only when routing is uniform

How to do this?

N → experts; T → tokens in a batch $\mathcal{B}$
f_i → fraction of tokens dispatched to expert i

$$f_i = \frac{1}{T} \sum_{x \in \mathcal{B}} 1\{\text{argmax}(p(x)) = i\}$$



P_i = Expected probability of selecting expert i.

$$P_i = \frac{1}{T} \sum_{x \in \mathcal{B}} p_i(x)$$

for any token x_j, if we consider the expert e, then the routing probability is $p(e|x_j)$

$$p(e) = \sum p(x_j)\, p(e|x_j)$$

↓ expected probability of routing to e^{th} expert

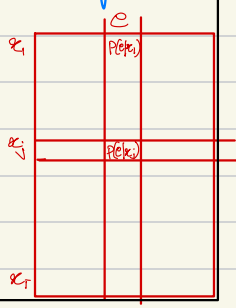
↳ x_j is sampled from the entire universe
↳ This is nothing but $E_{x \sim \text{corpus}}(p(e|x))$

Q) How do we compute expectation when we can't exactly compute the probabilities of selecting a token?
To empirically estimate the avg, we take IID samples and take a simple average

$$\therefore E_{x \sim \text{corpus}}(p(e|x)) = \frac{1}{T}\left(p(e|x_1) + p(e|x_2) \cdots p(e|x_T)\right)$$

↳ which is the avg of the column of probabilities for expert e

$$\therefore p(e) = \frac{1}{T}\sum_{j=1}^{T} p(e|x_j)$$



$$\text{loss} = \alpha \cdot N \cdot \sum_{i=1}^{N} f_i \cdot P_i$$

→ Q) Why does minimizing this ensure uniform probability?

↳ $\sum f_i P_i = f_1 P_1 + f_2 P_2 \cdots f_N P_N$

If f_i is high (i.e., more tokens are going to expert 1), then to minimize this P_i should reduce. So, ideally the optimizer should reduce P_i.
Also, we know that $\sum P_i = 1$ $\quad\quad$ $\sum f_i = 1$

So, if P_i reduces, then something must be compensated to ensure that the sum is still 1. Optimizer should choose to increase the P_i with the lowest value (Basically, to attain equilibrium f_i & P_i should be uniformly distributed)

The above simplified load balancing loss:
↳ Prevents router collapse
↳ Improves training efficiency by using all devices equally (as each expert is on a seperate device)

Selective Precision

* Training is typically done in bfloat16 as it reduces the memory footprint by 50%.
* But this increases the instability (compared to float32) - A common practice is conditional quantisation i.e., weights will be in bfloat16 but optimizer in float32.
* Here they observed that exponent is sensitive to small errors so they cast router to float32 (since it has a softmax operation)

Model (precision)	Quality (Neg. Log Perp.) (↑)	Speed (Examples/sec) (↑)
Switch-Base (float32)	-1.718	1160
Switch-Base (bfloat16)	-3.780 [*diverged*]	**1390**
Switch-Base (Selective precision)	**-1.716**	1390

* They observed that when training everything in bf16, the model loss diverged.
* To converge the training they used selective precision
* Mostly empirical work.
* Also, router computation cost is mostly negligible so not much effect on overall speed.

Reduced initialization scale

* Again mostly empirical work
* Default initialization: Sampling with $\mu = 0$; $\sigma = 1/\sqrt{d}$ (d ↓ input dimension), resample if beyond 2σ

* Recommended initialization: sampling with $\mu = 0, \sigma = \sqrt{0.1/d}$ resample if beyond 2σ

Model (Initialization scale)	Average Quality (Neg. Log Perp.)	Std. Dev. of Quality (Neg. Log Perp.)
Switch-Base (0.1x-init)	**-2.72**	**0.01**
Switch-Base (1.0x-init)	-3.60	0.68

Performance of 32 expert model after 3.5k steps (3 random seeds)

Higher Regularisation of experts

Typically, a uniform dropout is used to regularise the model & its value is 0.1 (10%). But MoEs increase the model size so high chance of overfitting. To solve this, they increase the dropouts for the experts.

Model (dropout)	GLUE	CNNDM	SQuAD	SuperGLUE
T5-Base (d=0.1)	82.9	**19.6**	83.5	72.4
Switch-Base (d=0.1)	84.7	19.1	**83.7**	**73.0**
Switch-Base (d=0.2)	84.4	19.2	**83.9**	**73.2**
Switch-Base (d=0.3)	83.9	19.6	83.4	70.7
Switch-Base (d=0.1, ed=0.4)	**85.2**	**19.6**	**83.7**	**73.0**

Dropout in other layers ↑ ↓ dropout in expert layers

- Pretrained on 34B tokens; Uniform dropout performs worse;
- Low dropout for non-experts and high dropout for expert layers perform the best

* This is done typically while fine-tuning on downstream tasks.

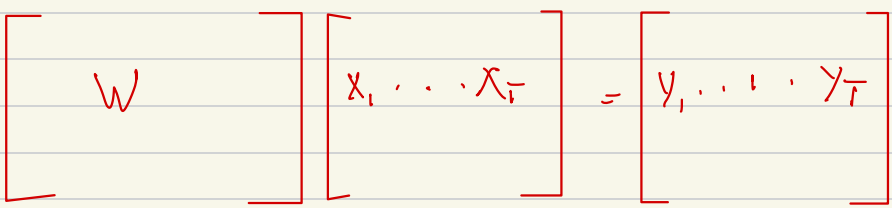
→ On one hand we want the distribution of points to be skewed (i.e., in the 4 clusters experiment described previously, we wanted to reduce the entropy/randomness of the router) for a particular data point (or token).

→ On the other hand we want all the routers to be activated equally i.e., load balancing (meaning if we take the same 4 clusters example & choose 4 routers in our experiment then we

would ideally want 2500 samples from each cluster (considering total 10k samples) so that the activation probability of each expert is 1/4.

Distributed Switch Implementation

* Trained on TPUs using Mesh Tensorflow
  - facilitates efficient model-parallel architectures (i.e., experts on different cores)
  - Statically compiled computational graph → fixed tensor shapes but dynamic computation.



When working with a batch of tokens, in PyTorch the batch size can be dynamic as it creates a dynamic computational graph. In Tensorflow, the computation graph is pre-compiled i.e., alongwith the weight matrix W, the batch size is also pre-defined (because the computation graph is static & pre-compiled)

So, what's the issue? – We cannot ensure uniform distribution of tokens to different experts.

eg: if we have a batch of 10k tokens & 100 experts and we fix that each expert can get 10 tokens. But lets say in 1 batch we get 20 tokens that go to one particular expert. So how do we compute this? (We've already fixed the computation graph)

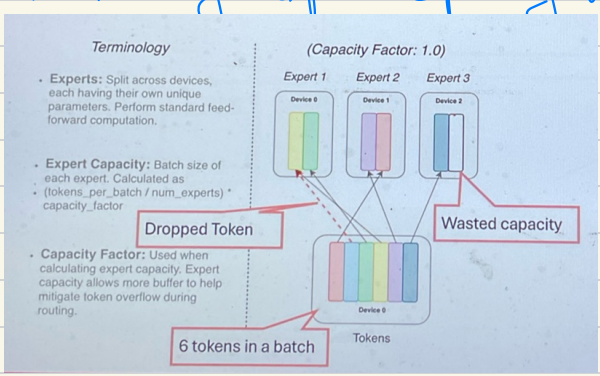
So, in this we define something called the Capacity factor. for the above example, each expert should ideally process 10 tokens so 10 indicates a capacity factor (CF) of 1. If we want a buffer, so lets say we allow each expert to process 15 tokens, then CF = 1.5

How is expert capacity calculated?
(The no of tokens processed by each expert)

$$\text{expert capacity} = \left(\frac{\text{tokens per batch}}{\text{no of experts}}\right) \times \text{capacity factor}$$

↑ uniform distribution of tokens to all experts

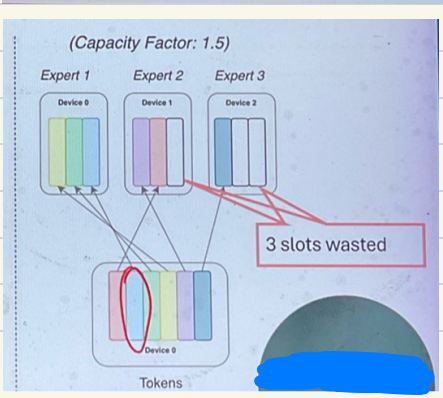
Modulating Expert Capacity via Capacity factor



*) In here, the 2nd, 3rd & 4th tokens all get routed to expert 1, but we can't do anything about it as we've fixed the CF to 1.

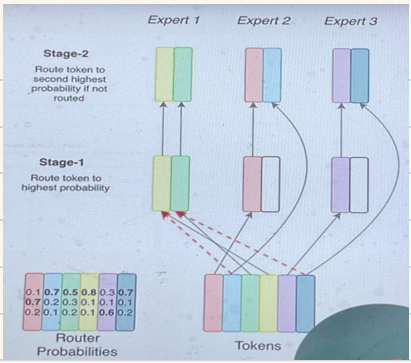
*) So, we drop a token.

*) Also, we have empty space in one expert which is space wasted.



*) Now if we take a CF = 1.5 then we can retain & process all the tokens but now we have 3 empty wasted slots.

No token left behind strategy



* Two stage routing
  - Stage 1: Route to the highest probability expert
  - Stage 2: Route the dropped tokens to the second best expert

*) Can also be done iteratively till we place every token.

*) Observations → Doesn't work empirically
  → Tokens prefer to be routed to the same expert
  → Maybe token dropping introduces regularisation so it works better.

Benchmarking Switch (top-1) versus MoE (noisy top-2)

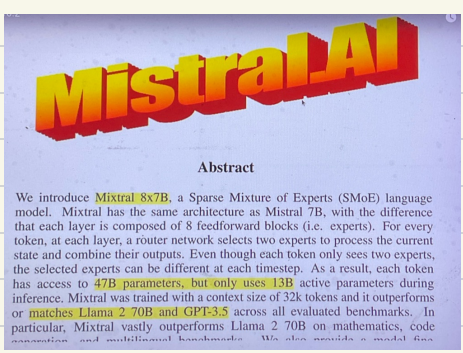
Time to reach -1.5 Neg. Log Perplexity

- 128 experts
- Alternate layers

Increase hidden dim. & no. of heads till it matches speed of top-2 routing

Model	Capacity Factor	Quality after 100k steps (↑) (Neg. Log Perp.)	Time to Quality Threshold (↓) (hours)	Speed (↑) (examples/sec)
T5-Base	—	-1.731	Not achieved†	1600
T5-Large	—	-1.550	131.1	470
MoE-Base	2.0	-1.547	68.7	840
Switch-Base	2.0	-1.554	72.8	860
MoE-Base	1.25	-1.559	80.7	790
Switch-Base	1.25	-1.553	65.0	910
MoE-Base	1.0	-1.572	80.1	860
Switch-Base	1.0	-1.561	**62.8**	1000
Switch-Base+	1.0	**-1.534**	67.6	780

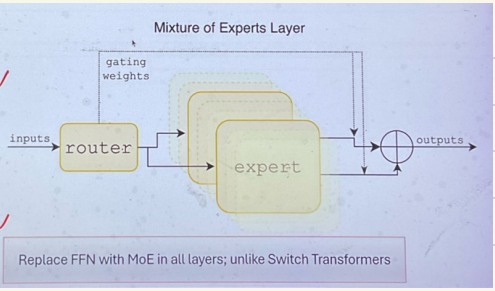
Mixtral (Mistral.AI)



*) First created a 7B dense model (Mistral 7B) and then they created a correspond MoE model where they repeated it 8 times (Mistral 8x7B)
  - Size: 8x7 = 56B ??
  - Actual Size: 47B → because only the FFN different; rest of the params are shared across the 8 replicas.

*) The no. of active parameters at any time is 13B at a time during inference.

Parameter	Value
dim	4096
n_layers	32
head_dim	128
hidden_dim	14336
n_heads	32
n_kv_heads	8
context_len	32768
vocab_size	32000
num_experts	8
top_k_experts	2



*) Uses top-2 routing
*) 32 transformer blocks
*) 8 experts in each layer (not in alternate layers like the switch transformer)
*) No noisy top-2 gating like the previous MoE.

$$G(x) := \text{Softmax}(\text{TopK}(x \cdot W_g))$$

$$y = \sum_{i=0}^{n-1} G(x)_i \cdot E_i(x)$$

instead of linear layer they use this SwiGLU layer

$$y = \sum_{i=0}^{n-1} \text{Softmax}(\text{Top2}(x \cdot W_g))_i \cdot \text{SwiGLU}_i(x)$$

↳ Combines Swish Activation & Gated Linear Unit (GLU)

$$\text{SwiGLU}(x) = x * \text{sigmoid}(\text{beta} * x) + (1 - \text{sigmoid}(\text{beta} * x)) * (Wx + b)$$

↳ This has a kind of inbuilt gating mechanism
sigmoid(beta*x) → probability with which 'x' is chosen
1 - sigmoid(beta*x) → " " " 'Wx+b' is chosen

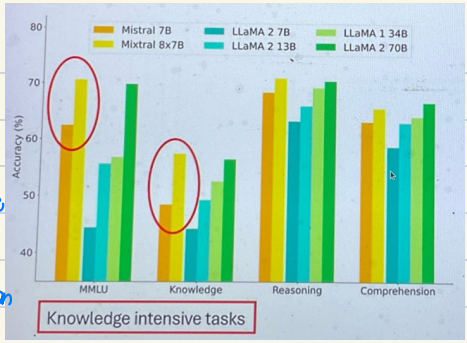
Reasoning vs Knowledge intensive tasks

*) In general, it is hypothesized that:
- FFN layers account for knowledge
- Attention layers account for reasoning or algorithms

*) To test this they compared the output of the Mistral base model along with the other Llama models to the sparse MoE

Mixtral model.

*) The results showed that the sparse MoE Mixtral heavily outperforms the base Mistral 7B model on knowledge intensive tasks whereas the performance is comparable for reasoning & comprehension tasks.



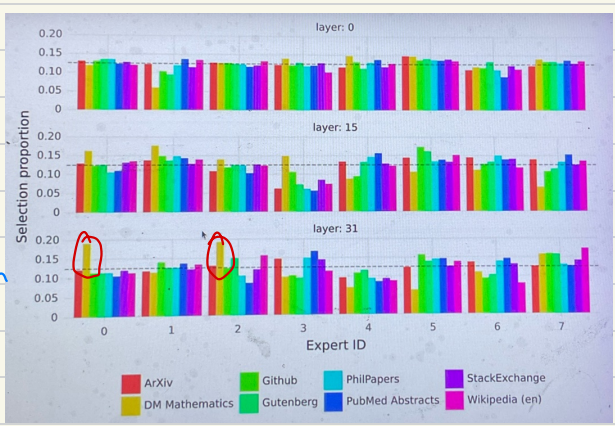
Interpreting routing decisions

*) Self-attention is always used as an interpretation tool -
- Which token in the input are we attending to while generating the next token?

*) Can we use routing decisions for interpreting the model?
- Which tokens are routed to a particular expert?

↳ eg: a specific task like all mathematical questions being routed to a particular expert.

*) Here we have:
- Validation split of Pile dataset
- Proportion of tokens assigned to each expert on different domains.
- Done for layer 0, 15 & 31.
- The distribution is clearly not uniform in the later layers.
- For eg: many mathematics related tokens get routed to expert 0 & 2.



Routing of consecutive tokens

How many times two consecutive tokens are routed to the same expert?

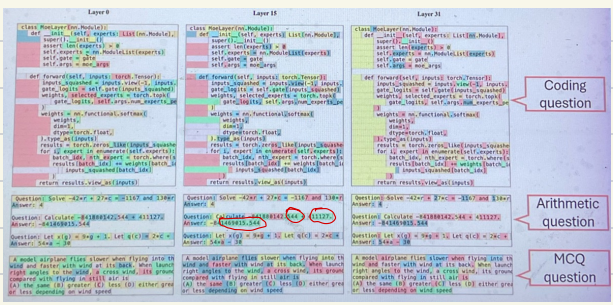
- Repititions at the 1st layer are close to random.
- Significantly higher at layers 15&31.
- The high no. of repetitions shows expert choice exhibits high temporal locality at these layers.

	First choice		
	Layer 0	Layer 15	Layer 31
ArXiv	14.0%	27.9%	22.7%
DM Mathematics	14.1%	28.4%	19.7%
Github	14.9%	28.1%	19.7%
Gutenberg	13.9%	26.1%	26.3%
PhilPapers	13.6%	25.3%	22.1%
PubMed Abstracts	14.2%	24.6%	22.0%
StackExchange	13.6%	27.2%	23.6%
Wikipedia (en)	14.4%	23.6%	25.3%

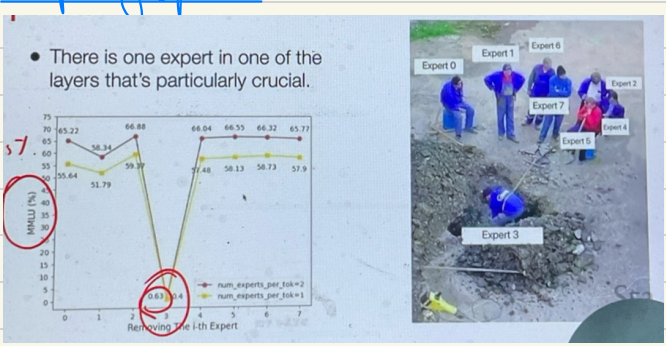
Which experts are active for different tokens?

*) Colors represent different experts

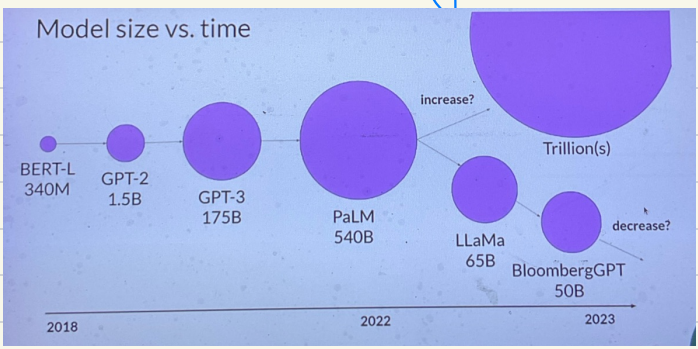
*) Experts do not specialize in any domain like coding, or maths. (But there are some patterns like most digits got routed to the same expert)



Interpreting experts

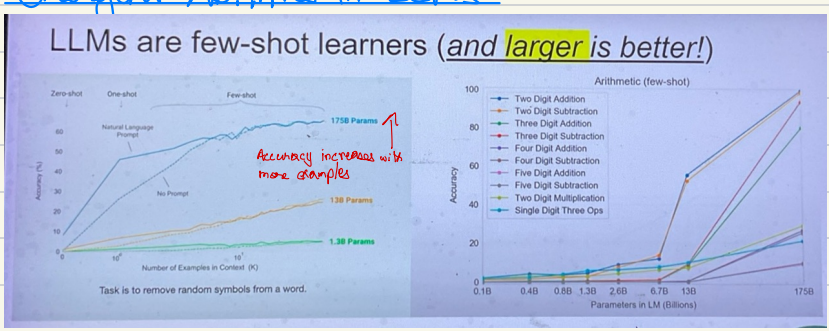


Lecture 11 – Scaling Laws

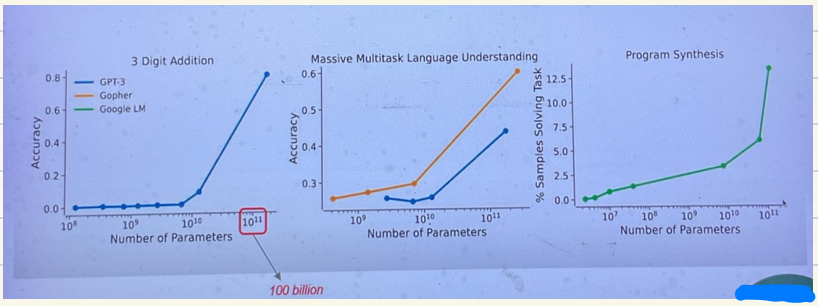


* Models started becoming very large after 2018.
* But after 2022, we see 2 diverging directions – one towards even larger models & other towards smaller models.

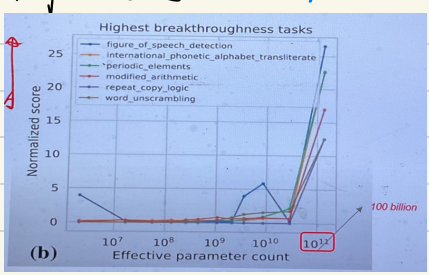
'Emergent' Abilities in LLMs



* 2 important things that came out of the famous GPT-2 paper were:
  - LLMs are few shot learners (meaning that they perform much better when given a few examples of the task you expect it to perform)
  - But we also observe that this phenomena of improving performance exists only when we climb up to larger models. (like here we see the sudden jump from 13B → 175B params but not so much in 1.3 → 13B models)
  - Same happens even with one-shot
  - We something similar for arithmetic tasks as well i.e., the sudden spikes.
  - Also, we see that the spikes occur much earlier when the task is much earlier as compared to more difficult tasks (like here 2-digit vs 4-digit addition)

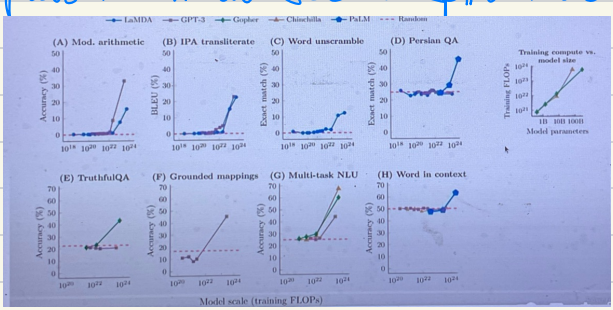


* We see similar kinds of spikes for various tasks and on various benchmarks and with multiple different models as we increase the number of parameters.
* Interestingly, we observe that in all these experiments we observe that there is a range between 10B—100B parameters where this spike occurs.
* Similar spikes occur on different types of NLP tasks between the same range of parameters (10B-100B).



Q) So we clearly see that models start to perform better when we increase the no. of parameters. But what about training steps? What happens if we increase the training steps & the compute? Will we see a spike there as well?

A) Yes. Given a model if we keep on increasing the training or compute, we will again see the spike after some time. And this too happens with various models.



Google BIG-BENCH Benchmark ——→ Quantifying Emergence (i.e., the spike)

* Consider a model family eg: PaLM
* Let x_n be the scale of n^{th} family member eg: PaLM 540B
* Let y_n be the family member's score on some Task & Metric

* Sort the pairs (x_n, y_n) by model scale x_n, smallest to largest.

$$\text{Emergence Score}\left(\{x_n, y_n\}_{n=1}^{N}\right) = \frac{\text{sign}(\text{argmax}_i\, y_i - \text{argmin}_i\, y_i)(\max_i y_i - \min_i y_i)}{\sqrt{\text{median}(\{(y_i - y_{i-1})^2\}_i)}}$$

↳ Basically quantifies the spike for a given model.

$i \rightarrow$ size of a given model

$\max_i y_i - \min_i y_i \rightarrow$ max score given to a model − min score given to the model (for different sizes)

sign of argmax is taken because we want to see if the model performs better when the params are increasing (so if this is +ve (i.e., if larger model has better score) then emergence score is also +ve & vice versa)

NOTE: Emergent ability

Emergent doesn't mean that the capability suddenly emerged in larger models while it was absent in smaller ones. We are just referring to the spike - i.e., the sudden jump in performance

When we talk about scaling laws in the ML community, usually people think about In-Context Learning (ICL)

What is ICL?

↳ Ask the model to do something it was never trained on and it still comes up with something

eg: if the model has learned to make 'Chai' & we prompt it to make 'coffee' & it still comes up with something because at a latent level it has associated both with similar concepts (like hot beverage)

Of course, the unknown task must be within the known classes of problems. You can't ask the model to teach you to play 'Cricket'

This is ICL in general although the task may not always be unknown.

Q) What did we realise from this?

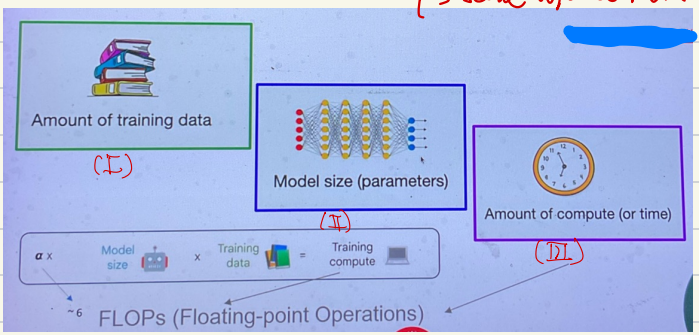
LLMs seem to be getting more intelligent with the following:

(i) Amount of training data

(ii) Model size

(iii) Amount of compute

↳ scale w/o bottleneck



FLOPs = α × model size × Training data

↓ measure of compute

↳ constant ~6 ↳ like a measure of the basic operation during fwd & bwd pass including matmul, add, etc.

*) An important thing to note here → as we try to scale one of these, the model should not run out of the other 2 params to perform the experiment.

eg: If we want to see what happens to accuracy as we keep increasing the training data, then during the experiment if we feel that we need a bigger model/more compute to see if the spike is happening, then we can freely do that. So, the model size is not a constraint (or a bottleneck). Similarly, for the other 2 params as well. If we want to experiment across model sizes & if we need more data in order to look for that spike then we will add that.

Recap on Parameter size & FLOPs

Operation	Parameters (→ context length)	FLOPs per Token
Embed	$(n_{\text{vocab}} + n_{\text{ctx}})\, d_{\text{model}}$	$4d_{\text{model}}$
Attention: QKV	$n_{\text{layer}} d_{\text{model}} 3d_{\text{attn}}$	$2n_{\text{layer}} d_{\text{model}} 3d_{\text{attn}}$ (→ Q,K,V)
Attention: Mask	—	$2n_{\text{layer}} n_{\text{ctx}} d_{\text{attn}}$
Attention: Project	$n_{\text{layer}} d_{\text{attn}} d_{\text{model}}$	$2n_{\text{layer}} d_{\text{attn}} d_{\text{embd}}$
Feedforward	$n_{\text{layer}} 2d_{\text{model}} d_{\text{ff}}$ (→ dimension of FFN sandwiched b/w 2 attention layers)	$2n_{\text{layer}} 2d_{\text{model}} d_{\text{ff}}$
De-embed	—	$2d_{\text{model}} n_{\text{vocab}}$
Total (Non-Embedding)	$N = 2d_{\text{model}} n_{\text{layer}} (2d_{\text{attn}} + d_{\text{ff}})$	$C_{\text{forward}} = 2N + 2n_{\text{layer}} n_{\text{ctx}} d_{\text{attn}}$

Peta flops day ←

one PF-day $= 10^{15} \times 24 \times 3600 = 8.64 \times 10^{19}$ floating point operations

Q) Are emergent abilities unpredictable?

A) It was the notion for a long time that the emergent abilities are unpredictable. But if that is true then, we never get to know the following:

- Which abilities (and when) exactly will emerge?
- What controls the trigger?
- Can we make desirable abilities emerge faster? → research area for small LMs
- Can we suppress undesirable abilities? → we don't want to release a model which when scaled opens a Pandora's box.

Q) Is the value of scaling laws only in predicting?

- It is also very important for budgeting (ML Ops)
- How much return for a given compute (resource) budget?
- How to allocate the compute budget - model size vs dataset size?

Can we fit "emergence" with a curve?

Intuition:

Input: $x_1 \dots x_n \sim N(\mu, \sigma^2)$

Task: estimate the average as $\hat{\mu} = \frac{\sum_i x_i}{n}$

What is the error? By standard arguments..

$$E\left[(\hat{\mu} - \mu)^2\right] = \frac{\sigma^2}{n} \rightarrow \text{expected deviation}$$

$$\log(\text{Error}) = -\log n + 2 \log \sigma$$

↳ This is a scaling law. More generally, any polynomial rate $1/n^\alpha$ is a scaling law. Taking a log makes it linear.

* If loss follows a power law then so should accuracy i.e., if loss drops according to a power law then accuracy should also go up according to a power law.

What about 'emergence' in a non-parametric setting?

Neural Nets can approximate arbitrary functions. Lets turn that into an example.

Input: $x_1 \ldots x_n$ uniform in 2D unit box. $y_i = f(x_i) + N(0,1)$
Task: estimate $f(x)$

Approach: cut up 2D space into boxes with length $n^{-1/4}$ average in each box.

What's our estimation error?
Informally, we have $\sqrt{n}$ boxes, each box has $\sqrt{n}$ samples

$$\text{Error} \approx \frac{1}{\sqrt{n}} + (\text{other smoothness terms})$$

In d-dimensions, this becomes $\text{Error} = n^{-1/d}$. This means scaling is

$$y = -\frac{1}{d}x + C$$

Takeaway: flexible 'non-parametric' learning has dimension dependent scaling laws

What if the loss-drop (i.e., emergence) follows power law?

$y = ax^k$ Power law: can be -ve $\quad \text{Loss}(x) \propto \frac{1}{x^{\alpha_x}}$

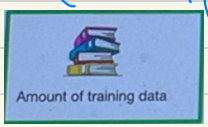
The authors of a paper (Scaling Laws for Neural Language Models - Jared Kaplan et. al - Johns Hopkins, Open AI) came up with a hypothesis:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N}$$

N_c → scaling factor; N → model size

$\alpha_N \sim 0.076$ $\quad N_c \sim 8.8 \times 10^3$ (non-embedding parameters) ← excluding embedding parameters

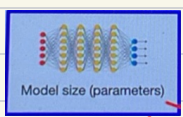
α_N, N_c — empirically determined



$$L(D) \approx \left(\frac{D_c}{D}\right)^{\alpha_D}$$

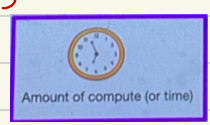
$\alpha_D \sim 0.095$ $\quad D_c \sim 5.4 \times 10^3$ (tokens)

D_c → depends on the tokenization method (will be different for say BPE vs WordPiece)



$$L(C) \approx \left(\frac{C_c}{C}\right)^{\alpha_C}$$

$\alpha_C = 0.057$ $\quad C_c = 1.6 \times 10^7$ Pf-days



$$L(C_{min}) = \left(C_c^{min} / C_{min}\right)^{\alpha_C^{min}}$$

$\alpha_C^{min} \sim 0.050$ $\quad C_c^{min} \sim 3.1 \times 10^8$ (Pf-days)

C_{min} → The minimum amount of compute required to achieve a target loss.

These 3 laws are known as **Kaplan Laws.**

$$L(C_{min}) = \left(C_c^{min} / C_{min}\right)^{\alpha_C^{min}}$$ → A more reliable version $L(C) \approx \left(\frac{C_c}{C}\right)^{\alpha_C}$

$\alpha_C^{min} \sim 0.050$, $C_c^{min} \sim 3.1 \times 10^8$ (Pf-days)

$$C_{min}(C) \equiv \frac{C}{1 + B/B_{crit}(L)}$$ (minimum compute, at $B \ll B_{crit}$)

This is the minimum compute that you would've required. Using this compute threshold value instead of actual 'C' (which might be more than what's needed) is more reliable. The paper recommends to scale according to C_{min} rather than C.

B_{crit} as computed below

Noise Batch

→ Plugin the loss to get B_{crit} batch

$$B_{crit}(L) \approx \frac{B_*}{L^{1/\alpha_B}}$$

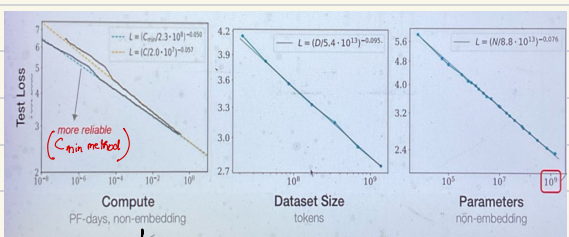
$B_* \sim 2 \cdot 10^8$ tokens, $\alpha_B \sim 0.21$

Parameters	Data	Compute	Batch Size
Optimal	∞	C	Fixed
↓	↓ vs.	↓ convert	↓
N_{opt}	D_{opt}	C_{min}	$B \ll B_{crit}$ *

Compute Efficient Value	Power Law	Scale
$N_{opt} = N_e \cdot C_{min}^{p_N}$	$p_N = 0.73$	$N_e = 1.3 \times 10^9$ params
$B \ll B_{crit} = \frac{B_*}{L^{1/\alpha_B}} = B_e C_{min}^{p_B}$	$p_B = 0.24$	$B_e = 2.0 \times 10^6$ tokens

* Take B much less than B_{crit} because results show that increasing the batch size above B_{crit} (which achieves that loss), we will get worse returns. So, it is recommended to keep the batch size within the limit.

*) It turned out that these laws (which started out more as hypotheses) and associated scaling equations were following the results very closely.
*) So scale is indeed predictable.
*) An important thing to note here is that when these laws came out, the largest scale models were around 1B (vs the one seen in earlier experiments with GPT-3 & others which are 100B or more).
*) So, the assumption here is that the same laws will be followed when we go to even larger models (like 100B, 500B, etc.)

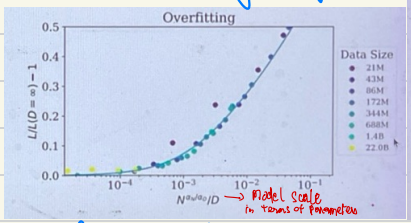


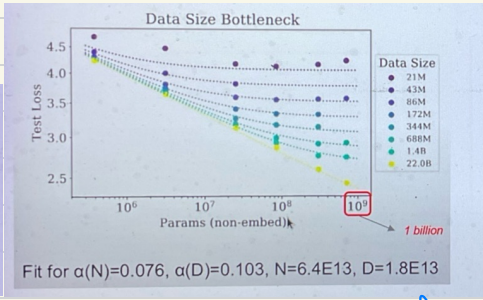
↳ log-log plotting

Observation 1a: Universality of Overfitting

Intuition: Bigger models love to work with more data

* But what if we keep scaling the parameters but don't supply the adequately large amount of training data.
* We observe that the test loss gets saturated & at a point it plateaus (no matter how many parameters you increase)

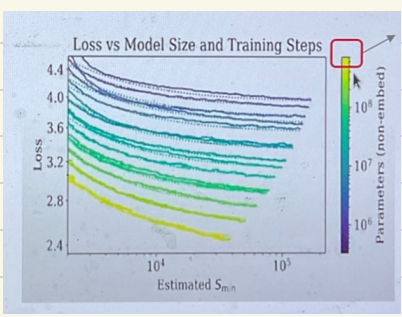


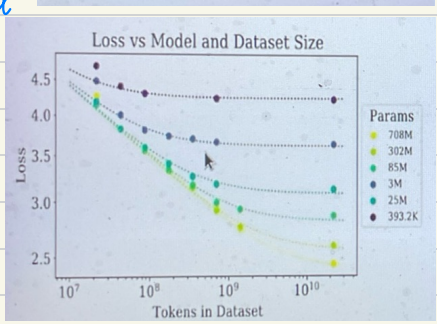


* So, the bigger models just overfit & memorize the training data instead of generalising well.
* So, if we've run out of training data, it doesn't make much sense to scale the model in terms of parameters.
* Universal because it is independent of the type of model we choose. If we don't increase the dataset size we're bound to fall into this trap.

Observation 16: Sample Efficiency

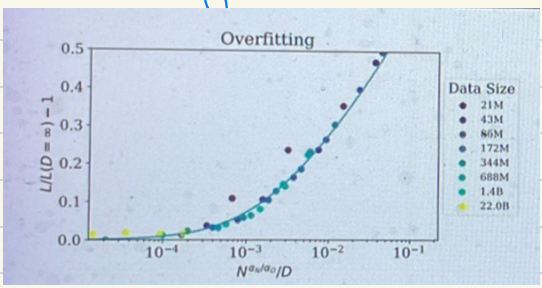
* Conversely, if we switch the roles, i.e., if we keep increasing the dataset size but work with only a smaller model. Will the smaller model show emergence?
* The results show that this doesn't happen & emergence only happens as we increase the model size
* So, if we have a fixed model size (no matter what model it is), we can't continuously keep reducing the loss beyond a certain point, no matter how much data we add.





Key Takeaway 1: Both parameters & dataset need to be scaled

* We cannot keep one fixed & keep scaling the other & hope to see emergence.



$$L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha_N} \qquad L(D) \approx \left(\frac{D_c}{D}\right)^{\alpha_D}$$

$$L(N,D) = \left[\left(\frac{N_c}{N}\right)^{\alpha_N} + \frac{D_c}{D}\right]^{\alpha_D}$$

→ Joint Scaling Law

↳ Power law must be dependent on both N & D. Not just 1 individually.

$$D \propto N^{\frac{\alpha_N}{\alpha_D}} \sim N^{0.74}$$

↳ Relationship b/w data scale & parameter scale.

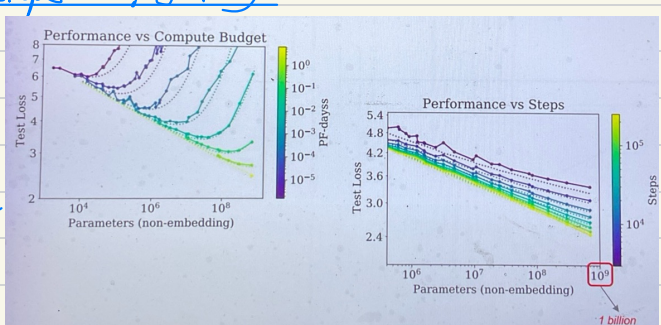
↓

Performance penalty is $N^{0.75/D}$

- If model increases 8x, dataset must increase 5x (scaling together)

Observation 2: Training time (Steps & FLOPS)

* Here, they observed the parameters against the compute.
* Similar to previous observations, we see that with less compute (i.e., training time or steps), we don't improve the test loss much, no matter how much we increase the parameters. (Strangely in fact, we see that the test loss starts to increase if we keep scaling the parameters without scaling compute)
* Another way of looking at this is the no. of training steps/iterations & the results are as observed in the 2nd graph.



Key Takeaway 2: Universality of training

* It is not good enough to have a joint loss that can characterise parameters & dataset. We need to characterise parameters & compute / steps.

$$L(N, S) = \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{S_c}{S_{min}(S)}\right)^{\alpha_S}$$

$$L(N, D) = \left[\left(\frac{N_c}{N}\right)^{\frac{\alpha_N}{\alpha_D}} + \frac{D_c}{D}\right]^{\alpha_D}$$

$S_c \approx 2.1 \times 10^3$ and $\alpha_S \approx 0.76$ and $S_{min}(S)$ is the minimum possible number of optimization steps.

$$S_{min}(S) \equiv \frac{S}{1 + B_{crit}(L)/B}$$ (minimum steps at $B \gg B_{crit}$)

↳ they kept $B \gg B_{crit}$ to have no worse performance & then started to increase the steps to see if they can drag down the loss.

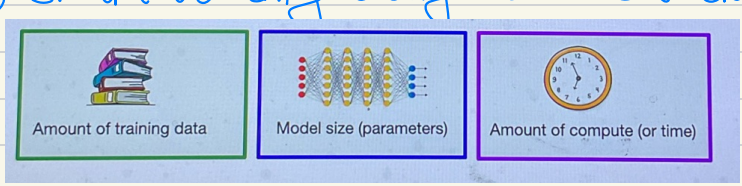
$$B_{crit}(L) \approx \frac{B_*}{L^{1/\alpha_B}}$$

$B_* = 2 \cdot 10^8$ tokens
$\alpha_B \sim 0.21$

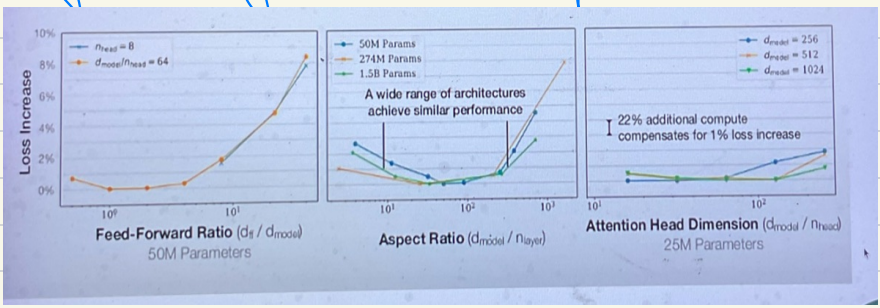
Parameter	α_N	α_D	N_c	D_c
Value	0.076	0.103	6.4×10^{13}	1.8×10^{13}

Parameter	α_N	α_S	N_c	S_c
Value	0.077	0.76	6.5×10^{13}	2.1×10^3

Q) Should we only worry about 3 factors?

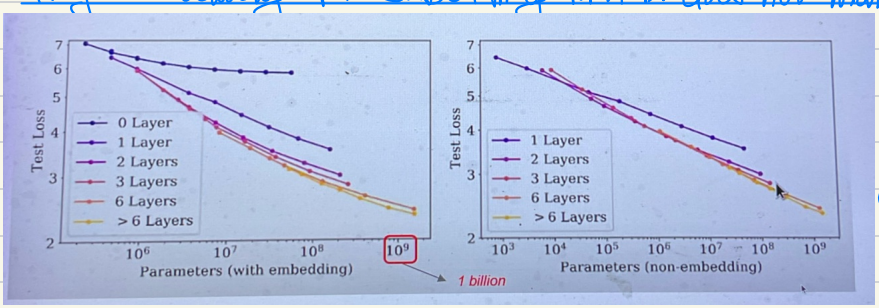


Key Takeaway 3: Model shape does not matter!



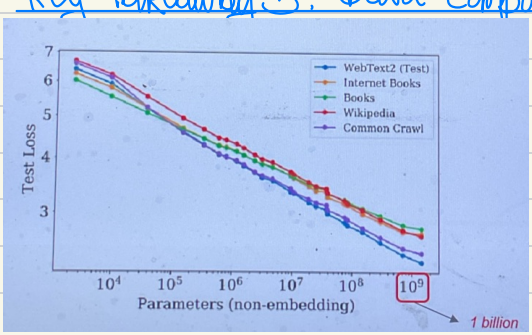
*) The effect of parameters like attention heads, aspect ratio, ratio of FFN to dmodel was very weak.

Key Takeaway 4: Embedding matrix does not matter



*) While they initially ignored the embedding parameters, the Kaplan group later tried the experiments with the embedding matrix as well and found no significant difference. (Only seems to be a big difference with the 0-layer vs 1-layer)
Both still show the scaling law (embedding & non-embedding)

Key Takeaway 5: Data composition does not matter



*) In a follow up study, they found that even for different types of datasets, the same power law is followed.

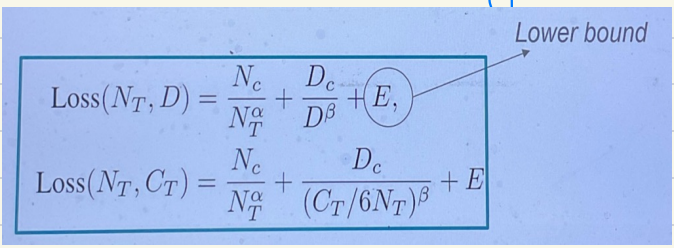
Kaplan Scaling Laws at a glance:

Power Law	Scale (tokenization-dependent)
$\alpha_N = 0.076$	$N_c = 8.8 \times 10^{13}$ params (non-embed)
$\alpha_D = 0.095$	$D_c = 5.4 \times 10^{13}$ tokens
$\alpha_C = 0.057$	$C_c = 1.6 \times 10^7$ PF-days
$\alpha_C^{min} = 0.050$	$C_c^{min} = 3.1 \times 10^8$ PF-days
$\alpha_B = 0.21$	$B_* = 2.1 \times 10^8$ tokens
$\alpha_S = 0.76$	$S_c = 2.1 \times 10^3$ steps

Parameters	Data	Compute	Batch Size	Equation
N	∞	∞	Fixed	$L(N) = (N_c/N)^{\alpha_N}$
∞	D	Early Stop	Fixed	$L(D) = (D_c/D)^{\alpha_D}$
Optimal	∞	C	Fixed	$L(C) = (C_c/C)^{\alpha_C}$ (naive)
N_{opt}	D_{opt}	C_{min}	$B \ll B_{crit}$	$L(C_{min}) = (C_c^{min}/C_{min})^{\alpha_C^{min}}$
N	D	Early Stop	Fixed	$L(N,D) = \left[\left(\frac{N_c}{N}\right)^{\frac{\alpha_N}{\alpha_D}} + \frac{D_c}{D}\right]^{\alpha_D}$
N	∞	S steps	B	$L(N,S) = \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{S_c}{S_{min}(S,B)}\right)^{\alpha_S}$

Joint

Chinchilla (Hoffman) Scaling Law



* Became very popular recently.
* Paper came out in 2022 in NeurIPS (Paper by DeepMind)

T → Total parameters (incl. embedding matrix)

E → loss intrinsic to the dataset

Empirically determined

$$\text{Loss}(N_T, D) = \frac{N_c}{N_T^\alpha} + \frac{D_c}{D^\beta} + E \longrightarrow L(N,D) = 1.69 + \frac{406.4}{N^{0.34}} + \frac{410.7}{D^{0.28}}$$

$$N_{opt}(C) = G(C/6)^a \qquad D_{opt}(C) = G^{-1}(C/6)^b$$

where $G = \left(\frac{\alpha A}{\beta B}\right)^{\frac{1}{\alpha+\beta}}$ $\qquad a = \frac{\beta}{\alpha+\beta}$ $\qquad b = \frac{\alpha}{\alpha+\beta}$

Fitting the constants yields: $\alpha \approx \beta$ → i.e., the best curve fit
i.e., equal scaling of N & D
↳ model parameters
↳ no of tokens

Kaplan vs Chinchilla Scaling Law

Kaplan: $N_{\setminus E}^* \propto C_{\setminus E}^{0.73}$ (↳ w/o embedding matrix)

Chinchilla: $N_T^* \propto C_T^{0.50}$

Slightly different from Kaplan which states that if model increases 8x, dataset should increase by 5x. Here they say 8x for both

$$N_T = N_E + N_{\setminus E}, \quad C_T = 6 N_T D = 6 (N_E + N_{\setminus E}) D,$$

$$N_E = (h+V)d, \quad C_{\setminus E} = 6 N_{\setminus E} D.$$

(context ↔ ↳ vocab)

(Revised) Chinchilla Scaling Law

*) By Epoch AI

$$L(N,D) = 1.69 + \frac{406.4}{N^{0.34}} + \frac{410.7}{D^{0.28}} \rightarrow L(N,D) = 1.82 + \frac{514.0}{N^{0.35}} + \frac{2115.2}{N^{0.37}}$$

scaling factor is a lot higher ↑

↳ better fit

Q) Is there a problem with our point of view?

Scaling Laws – A Mirage?

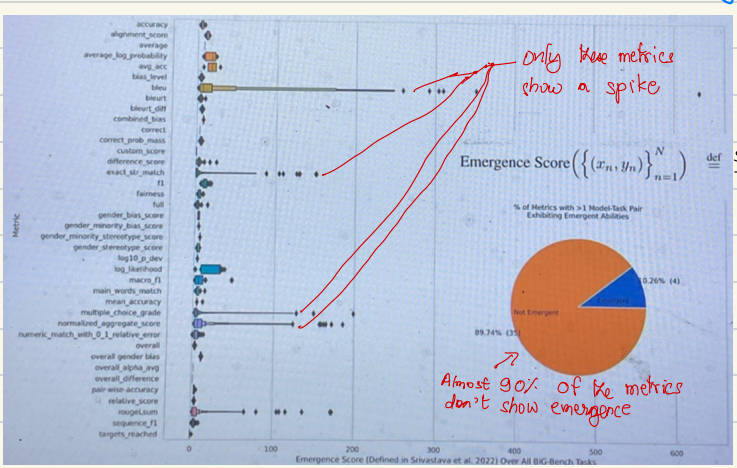
*) Recent paper from Stanford in 2023.

*) LLMs seem to be getting more intelligent with:
(i) Training data (ii) Model size (iii) Compute

↓

Lets look at this observe if there is truly a spike

Motivation: Not all metrics score same (Emergence Score)



$$\text{Emergence Score}\left(\{(x_n, y_n)\}_{n=1}^{N}\right) \stackrel{\text{def}}{=} \frac{\text{sign}(\text{argmax}_i\, y_i - \text{argmin}_i\, y_i)(\max_i y_i - \min_i y_i)}{\sqrt{\text{Median}\left(\{y_i - y_{i-1})^2\}_i\right)}}$$

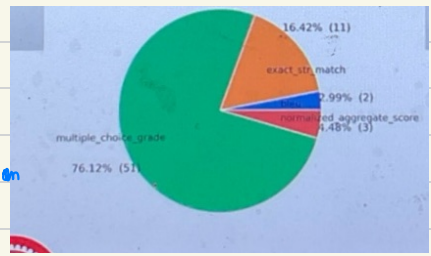
Q) So is the devil in the metric?

- The above observation motivated them to delve deeper into

The 10% metrics that show the spike.

→ which Kaplan used

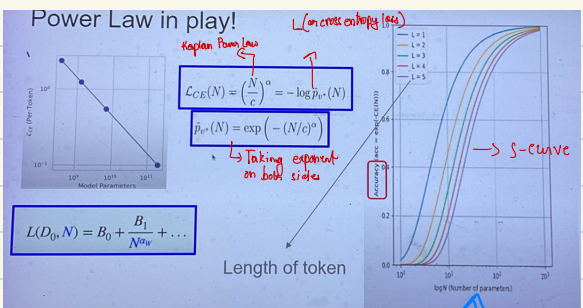
> 92% of BIG BENCH:



→ Multiple Choice Grade = $\begin{cases} 1 \text{ if highest probability mass on correct option} \\ 0 \text{ otherwise} \end{cases}$

→ Exact String Match = $\begin{cases} 1 \text{ if output string exactly matches target string} \\ 0 \text{ otherwise} \end{cases}$

→ Very strict 0/1 accuracy measure
→ Too challenging for smaller models!
→ Is it even worth it?



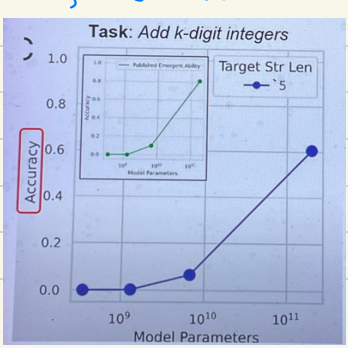
*) So they re-look at the power laws & equate it with the cross entropy loss (i.e., the L)

$$\text{So, } \left(\frac{N}{C}\right)^{\alpha} = -\log \hat{p}_{v^*}(N)$$

$$\hat{p}_{v^*} = \exp\left(-(N/C)^{\alpha}\right)$$

So, this becomes an S-curve

*) They looked at the accuracy in terms of the length of token. Predicting token L=1 is easier compared to L=5, so we need more time to get that right.



Problem with Non linear Measures like exact string match

Task: add k-digit integers

↳ 1 if all k+1 digits in model's output are correct
0 otherwise

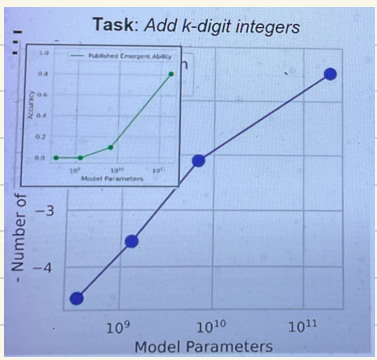
with this accuracy measure we get this curve (like Kaplan)

$$\hat{p}_{v^*} = \exp\left(-(N/c)^{\alpha}\right)$$

$$\text{Accuracy}(N) \approx p_N(\text{single token correct})^{\text{num of tokens}}$$

Change of perspective: Measure - Edit Distance

$$\text{Edit Distance}(N) \approx L\left(1 - p_N(\text{single token correct})\right)$$



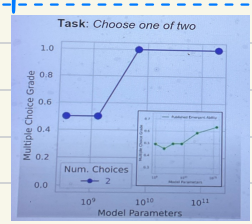
Here we reward intermediary probabilities as well (eg: 0.98, 0.99 will be penalized less)

Now, the graph looks more linear. The emergence has disappeared.

Problems with Discontinuous Measure: eg - Multiple Choice Grade

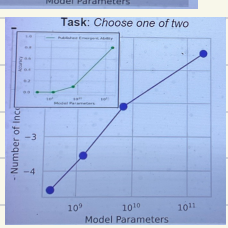
Task: Choose one of two

↳ 1 if highest probability mass on correct option
0 otherwise



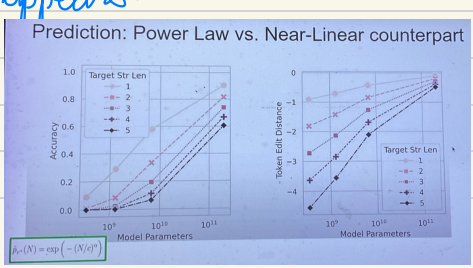
$$\hat{p}_{v^*}(N) = \exp\left(-(N/c)^{\alpha}\right)$$

$$\text{Brier Score} = 1 - (\text{probability mass on correct option})^2$$

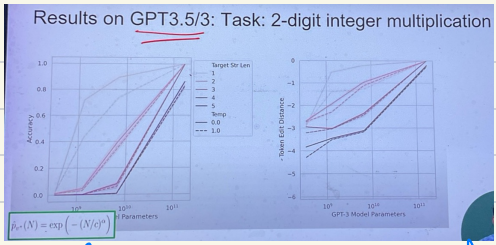


↳ Here too, if we change the metric to Brier Score (a softer metric), then emergence disappears.

*They tried this with multiple examples and observed similar results by changing the perspective.



* Even tried it on the latest models (at the time) like GPT-3/3.5 and observed the results.



* Even worked with BIG BENCH benchmark with different metrics like Brier Score (i.e., spikes disappeared)

Key Takeaways

* Want to predict without the theatrics? Choose a metric that's soft (in a continuous sense)
* There's no sudden jump in reality ("most" can be predicted on a linear scale)
* Do we really need the power law of scale? Maybe not!
  ↳ But then there may be cases where a soft measure doesn't make much sense

---

Lecture 12.1 - Pre-training of causal LMs & In-context learning

How to make a Large Language Model (like GPT/ChatGPT)?

* Pre-Training
  → This is the point where most of the reasoning power is infused in the model.
  → Data - Billions of tokens of unstructured text from the internet (html webpages, books, etc.)
    ↳ Most imp part