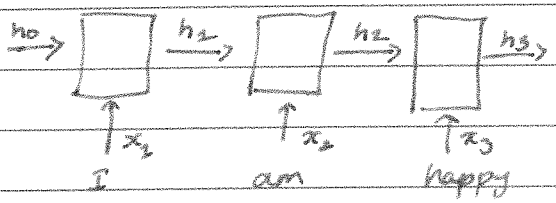
LARGE LANGUAGE MODELS

__/__/__

* Introduction to Transformer Architecture

→ In an encoder RNN, the sentence is given at one go. But the computation does not happen in one go.



x_i: Gives you the embedding from a corpus

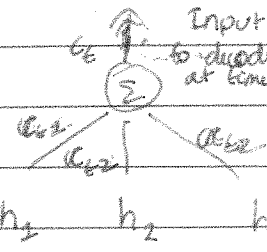
h_i: Gives you the embedding of the word in the context of the sentence

$h_1 = f(h_0, x_1)$, $h_2 = f(h_1, x_2)$

→ So basically, even though you are given the entire sentence in one go, the computation can only happen one step at a time. We do this to get the contextual representation of the word in the sentence.

→ The decoder RNN has to be sequential because the translated word is provided as input for the next word.

→ There is no notion of alignment or attention in Encoder-Decoder RNN we can add an attention mechanism the following way



$c_t = \sum_{i=1}^{n} \alpha_{ti} h_i$

Context vector for output y_t

Now, $\alpha_{ti} = \text{align}(y_t, h_i) = \dfrac{\exp(\text{Score}(s_{t-1}, h_i))}{\sum \exp(\text{Score}(s_{t-1}, h_i))}$

Function of the previous state of the decoder & the input vectors

i.e attention to be paid to input at timestep t was dependent of s_{t-1} & h_i

__/__/____

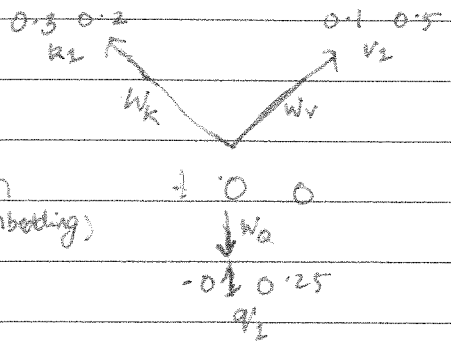
→ Can α_{ti} be computed in parallel for all i at timestep t?
↳ Yes; α_{ti} is dependent on s_{t-1} & h_i ⇒ Both available at t.

→ Can α_{ti} be computed in parallel for all t?
↳ No! α_{ti} is dependent on s_{t-1} which is dependent on $\alpha_{t-1,i}$

TAKEAWAY: For a given t, α_{ti} can be parallelized, however cannot be parallelized across t

§1.2 Attention is All you need!

§1.3 Self-Attention
↳ Previously, α_{ti} was a function of the decoder output from timestep $t-1$ & h_i.
↳ Now instead we use $\alpha_{ij} = f(h_i, h_j)$ - i.e the attention is calculated wrt to the input sentence (word embeddings)
↳ Since the input sentence is provided at one-go, and the recurrent connection is removed, this attention can be computed parallely for all t.



For each word create 3 linear transformations (Parallely)

Earlier we had Score (s_{t-1}, h_j)
Now we have Score (q_i, k_j) for t_j
i.e for a fixed query vector q_i we calculate the score of all words x_j wrt query q_i

__/__/__

↳ Score function is just the dot product

$$e_i = [q_i \cdot k_1 \quad q_i \cdot k_2 \quad \dots \quad q_i \cdot k_T]$$

↳ $\alpha_{ij} = \text{Softmax}(e_{ij}) = \frac{e^{e_{ij}}}{\sum_{t} e^{e_{ij}}}$

↳ $z_i = \sum_{j=1}^{T} \alpha_{ij} v_i$

⇒ All the query vectors can be computed in one go if you vectorize it in the following way:

- $Q = \underbrace{[q_1 \; q_2 \dots q_T]}_{\mathbb{R}^{d_k \times T}} = W_Q \underbrace{[h_1 \; h_2 \dots h_T]}_{\mathbb{R}^{d_{model} \times T}} \rightarrow \mathbb{R}^{d \times T}$

- You can do the same for Key & Value vectors

- You can now vectorize the z-output vector:

$$z = [z_1, z_2, \dots, z_T] = \text{Softmax}\left(\frac{Q^T K}{\sqrt{d_k}}\right) V^T$$

⇒ Alternate Notation

Input Sequence : $H \in \mathbb{R}^{T \times d_{model}}$

Projection Matrices : $W_Q, W_K, W_V \in \mathbb{R}^{d_{model} \times d_k}$, $d_q = d_k = d_v$

Query, Key, Value : $Q, K, V, \in \mathbb{R}^{T \times d_k}$

Attention : $\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \in \mathbb{R}^{T \times d_k}$

$Q = HW_Q$ where $H = \begin{bmatrix} — & h_1 & — \\ & \vdots & \\ — & h_T & — \end{bmatrix}$

__/__/____

21.4 Multi-Headed Attention

→ By using multiple heads, we can capture multiple representations of the contextual representation - Just like how in CNN's you can have one filter that blurs, another that detects edges, etc.

z_1 ... z_T ($T \times d_{out}$) → You can choose

Linear

$W_O : 512 \times d_{out}$

CONCATENATE ($T \times 512$)

$T \times 64 = d_k$ | $T \times 64 = d_k$ | $T \times 64 = d_k$

Matmul : V_0^T | Matmul : V_1^T | Matmul : V_8^T

Softmax | Softmax | Softmax

Scale : $\frac{1}{\sqrt{d_k}}$ | Scale : $\frac{1}{\sqrt{d_k}}$ | Scale : $\frac{1}{\sqrt{d_k}}$

Matmul : $Q_0^T K_0$ | Matmul : $Q_1^T K_1$ | Matmul : $Q_8^T K_8$

Dot Product Self-Attention

SCALED

$T \times d_k$: Q_0 K_0 V_0 | Q_1 K_1 V_1 | Q_8 K_8 V_8

$d_{model} \times d_k$: W_Q^0 W_K^0 W_V^0 | W_Q^1 W_Q^1 W_V^1 | W_Q^8 W_K^8 W_V^8

(Linear transformations)

$h_1, h_2 \ldots, h_T$ ($T \times d_{model}$) | $h_1, h_2 \ldots, h_T$ | $h_1, h_2 \ldots, h_T$

1 encoder layer:

FFN

MULTI-HEADED SELF-ATTENTION

h_1 h_2 ... h_T

__/__/__

4.2.1 Decoder Layer

→ Initially when you're training the transformer, it could produce garbage translations. Since the translated output at t-1 is fed as an input at t, these garbage translations would be recurrent & produce poor outputs for all timesteps

→ Thus, in Teacher forcing, instead of feeding in the output from time step t-1, the ground truth from t-1 is fed into it.
↳ loss is calculated wrt the predictions & backpropagated.

→ Since at inference we won't have the ground truth & instead only have the translations upto t-1 at timestep t, we use a masked Self-Attention so that the ground truth from t to T is masked & not made available to the transformer.

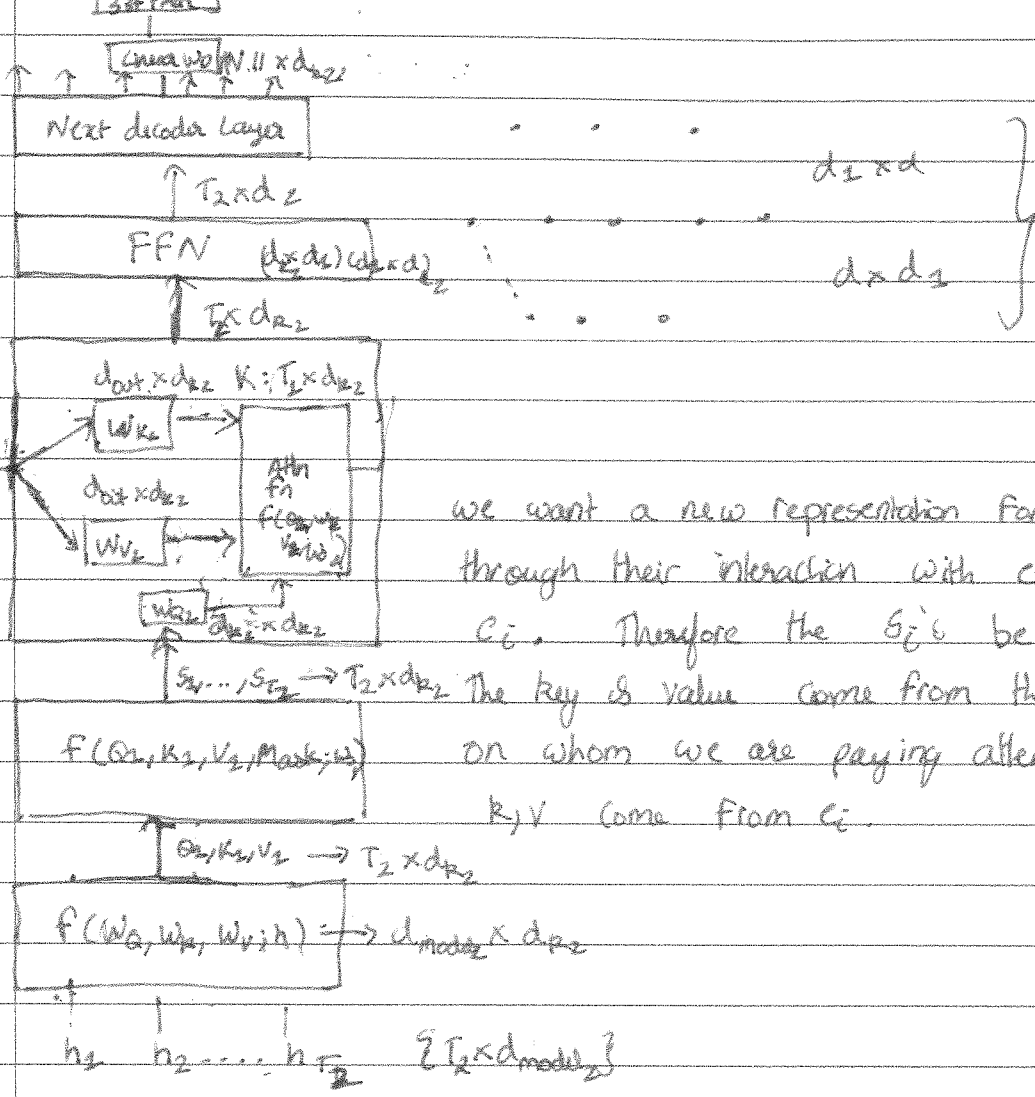
→ A mask M is therefore applied before the Softmax.
$Softmax(Q^T K + M)$. Here M will be an upper triangular $-\infty$ matrix so that the predicted probability is 0.

$$\begin{matrix} q_1 k_1 & -\infty & -\infty & -\infty \\ q_2 k_1 & q_2 k_2 & -\infty & -\infty \\ q_3 k_1 & q_3 k_2 & q_3 k_3 & -\infty \\ q_4 k_1 & q_4 k_2 & q_4 k_3 & q_4 k_4 \end{matrix}$$

} This is the Masked Self Attention.

__/__/____

2.2 Zooming in to the Decoder Layer



Cross Attention input from encoder

we want a new representation for s_i's here. through their interaction with encoders outputs c_i. Therefore the s_i's become the query The key & value come from the representations on whom we are paying attention. Hence K, V come from c_i.

→ In the subsequent layers we still need to use masking since the output from the first layer at 10^{th} position has seen words $1 \ldots, 9$. In the second layer if we don't use masking the representation for the third word may use the output 10^{th} word which has seen more than 2 words.

→ Teacher forcing should be used for the initial few epochs. Post which use output of the decoder

__/__/__

§2.3 Positional Encoding

→ When calculating the attention, we essentially are doing $z_i = \sum_i (q_i^T k_i) v_i$. This seem doesn't really take into account the position of the word. Whether you have "I am home today" or "I today home am" when you're calculating the attention for I, the sum is the same. There is no notion of position!

→ How do we embed positional information in the word embedding h_j (of dim = 512 maybe)?

↳ use a positional embedding vector (also dim = 512) & add it to word embedding.

i) Constant vector -

ii) One-hot embedding - Sparse, distance for all positions will be same. not relative

iii) Learnable Parameter with random init - is not encountered during training, & not in its during inference of a particular seq length

§2.4 Sinusoidal Embedding

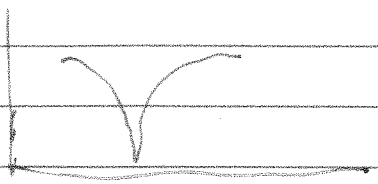
→ Embed a unique pattern of features for each position j which the model will learn to attend

↳

$$PE_{(j,i)} = \begin{cases} \sin\left(\frac{j}{10{,}000^{2i/d_{model}}}\right) & \text{if } i=0,2,\ldots \\ \cos\left(\frac{j}{10{,}000^{2i/d_{model}}}\right) & \text{if } i=0,2,\ldots \end{cases} \quad \begin{array}{l} i=0,1,\ldots,255 \\ j \text{ is the position for which we want the encoding : } j \in 1,\ldots,T \end{array}$$

$i \in 1, \ldots, d_{model}$ (512 in our case)

As i increases, frequency decreases as it will take more time to complete a cycle.



Positional encoding captures relative distance. diff between pos 42 & 39 will be same as 39 & 54

__/__/____

§ 2.5 Batch Normalization

- How do we ensure gradient flows across the network?
  → Residual Connections

- How do we speed up training process? Normalization
  → Let x_i^j denote the activation of i^{th} Neuron for j^{th} Sample
  → Let d indicate an accumulator of the d^{th} layer storing the activations of batch inputs

13/02/24

63.1 Introduction to Language Modelling

→ So far we looked at transformers in the context of Machine Translation However they can be used for any text generation task like Summarization, Q&A, Chat is also for sentiment analysis

→ All of them can be abstracted into an input-output form with the transformer acting as a function mapping inputs to outputs

→ However, we would have to train the transformer from scratch for each task using a dataset specific to the task ⇒ Long time
↳ Preparing labelled datasets specific to each task is expensive both in terms of time & money.
↳ Large amounts of unlabelled data available for free. Can we use them & then finetune for specific tasks?

→ Question : i) Can a model develop basic understanding of language by getting exposure to a large amount of raw text? (Pre-Training)

ii) Can it perform well on downstream tasks with min supervision? (Supervised Fine-Tuning)

13/02/24

§3.2 Language Modelling

→ A sentence is a sequence of words $x_1, \ldots, x_n$ where $x_i \in V$. Given a large corpus, we expect some sentences to appear more frequently than others (i.e. more probable)

→ Given a sequence input, $f: (x_1, \ldots, x_n) \to [0, 1]$ assigns a probability. Such a function is called a language model.

↳ we can think of the sequence as

$$P(x_1) \cdot P(x_2 | x_1) \cdot P(x_3 | x_2, x_1) \cdots$$

(Naive) ↳ if we assume x_i's are independent, $P(x_1, \ldots, x_n) = \prod P(x_i)$

→ we can train a model the task of predicting the next token in a sequence using sequences freely available on the internet. Given a sentence of length k, we can provide $k-1$ tokens & ask it to predict the next token.

Language Modelling

§3.3 Transformers for Language Modelling

→ Pass an input sequence into the transformer & predict a probability distribution & learn parameters such that the probability of the most likely word is the highest

Causal Language Modelling ↳ Auto-regressive models: conditional probabilities are given by parametrised functions with fixed #params (like transformers)

↳ we are looking for f_θ such that $P(x_i | x_1, \ldots, x_{i-1}) = f_\theta(x_i | x_1, \ldots, x_{i-1})$

i) Encoder only models

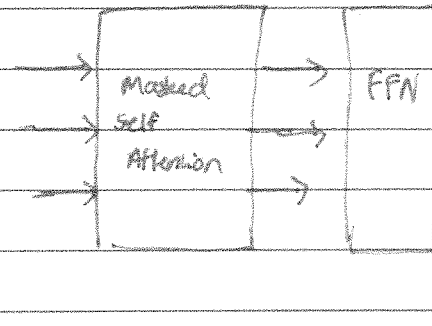
ii) Decoder only models (GPT)

iii) Encoder - Decoder Model

__/__/__

6.3.4 Causal Language Modelling

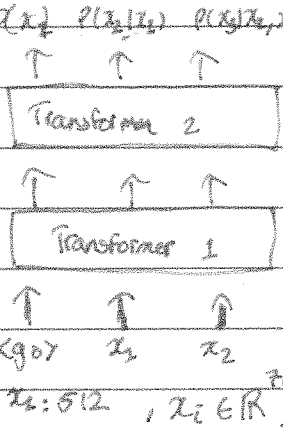
→ In a decoder only model, we do not have cross attention.



- Probabilities are determined by the parameters of the model
  $= P(x_1;\theta) \cdot P(x_2|x_1;\theta) \cdots$
- Objective: Max $L(\theta)$

6.3.5 Generative Pretrained Transformer (GPT)

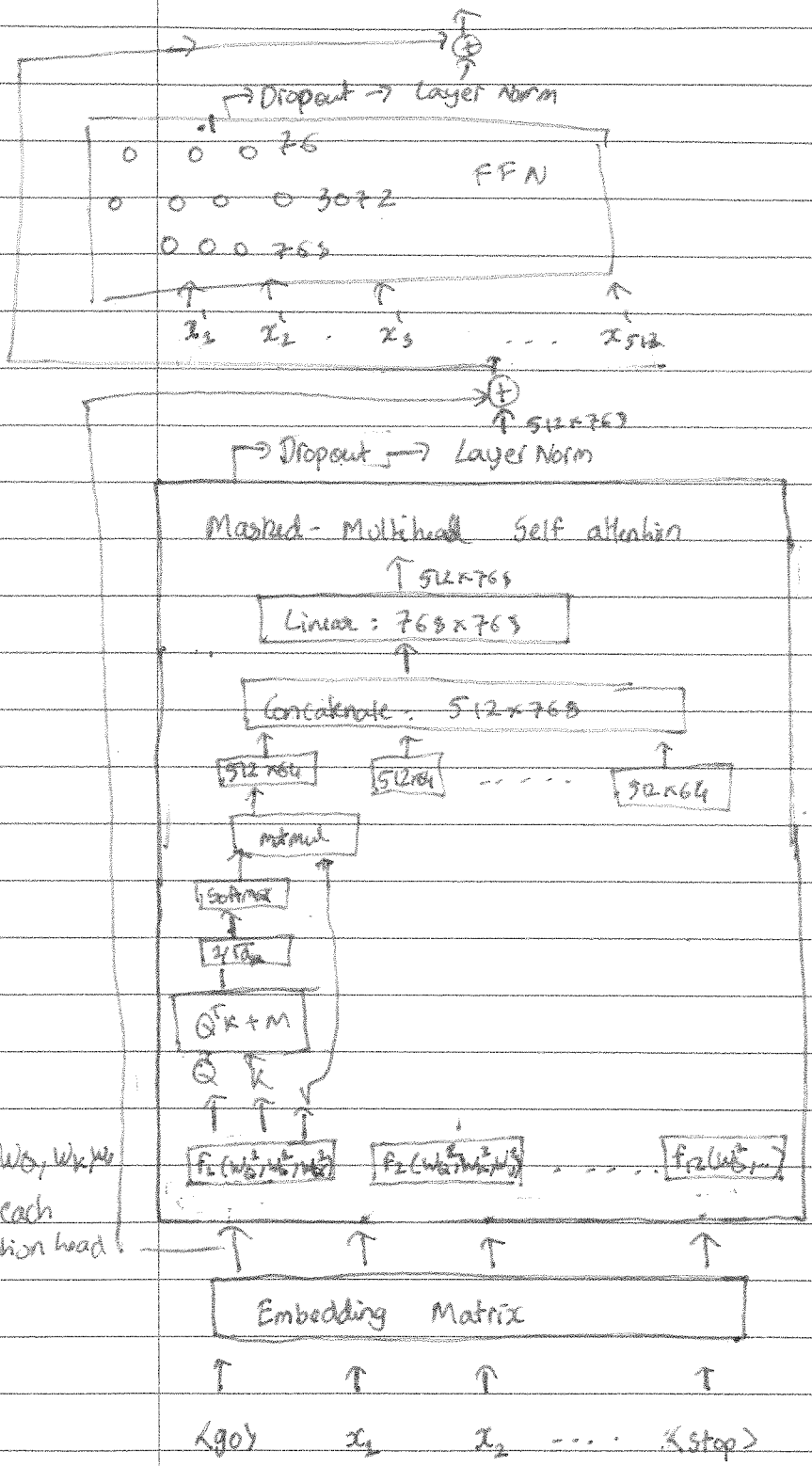
→ We can stack up n blocks as seen above.



- $h_0 = X \in \mathbb{R}^{T \times d}$ where X is the input sequence
- h_l = transformer_block(h_{l-1})
- $h_n[i]$ is the i^{th} output vector from the last block
- $P(x_i) = \text{softmax}(h_n[i] W_e)$
- $L = \sum_i^T \log(P(x_i | x_1, \ldots, x_{i-1}))$

$x_i : 512$, $x_i \in \mathbb{R}^{768}$, $X \in \mathbb{R}^{512 \times 768}$

→ 12 transformer blocks, Context Size = 512 (↑), Sequence Length), 12 Attention heads, FFN Hidden Layer Size: $768 \times 4 = 3072$ ($768 \rightarrow 3072 \rightarrow 768$), GeLU Activation in FFN, Dropout, Layer Norm, residual Connections



→ $(768 \times 3072) + (3072 \times 768)$

$+ 768 + 3072 = 4.7M$

for FFN

→ Since we have 12 heads, each head must output a 768/12 = 64 dimension vector

- $W_Q = W_K = W_V = 768 \times 64$
- For 12 heads : $12 \times 3 \times 768 \times 64$
- Linear : $W_O : 768 \times 768$

→ Token Embedding : $|V| \times d_{model}$

$= 40,478 \times 768 \approx 31M$

→ Positional Embed : context length $\times d_{model}$

$= 512 \times 768$

20/02/24

6.3.6 Pre-Training & Fine Tuning

→ Pre-Training: · Input is Batch (B) × Sequence Length (T) × d_{model} (C) = 64 × 512 × 768

- $L = -\sum_{x \in V} \sum_{i=1}^{T} y_i \log(\hat{y}_i)$
- Adam with Cosine Scheduler
- Use Teacher Forcing for quicker and stable convergence

→ Fine-Tuning: Fine-tuning involves adapting model for various downstream tasks (with minimal change in architecture)

6.4.1 Decoding Strategies

→ Since all the calculations are deterministic, given x_1, the output produced will be the same no matter how many times we run it.

Decoding strategies

Deterministic	Stochastic
- Exhaustive Search	- Top-K Sampling
- Greedy Search	- Top-p (Nucleus Sampling)
- Beam Search	
- Contrastive decoding	

6.4.2 Exhaustive Search

Exhaustively search for all possible sequences & their associated prob & output the sequence with the highest probability.

$V = \{x_1, x_2, x_3\} \Rightarrow x_1 x_2 x_3, x_2 x_1 x_3, x_3 \ldots$

Check the prob of every permutation.

Timestep – 1

Sequence Length : # of times the decoder is run

20/02/22

64.3 Greedy & Beam Search (Degenerative)

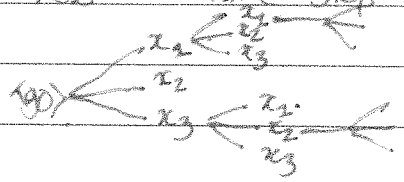
→ Whilst in exhaustive search, we look at all $|V|^{\text{seq length}}$ possible combinations, in Greedy search we only look at the most probable token at a time step.

↳ Input go ⇒ $p(x_1)$, $p(x_2|x_1)$, $p(x_3|x_2,x_1)$

So only 1 sequence is generated & greedily the most probable token is selected.

↳ Note that the most probable sequence found using exhaustive search maynot be generated by greedy search.

→ In Beam search with Parameter k, we generate all probabilities in a time step & choose the top-k to feed into the next time step

↳ 

↳ When k=1, it is equivalent to greedy search & when k=|V| it is equivalent to exhaustive search.

↳ good for translation where we want deterministic output.

↳ Normalize the sequence probability by dividing by the sequence length. Otherwise long sequences will have low probability. From the k sequences output the one with highest Normalized Probability.

20/02/22

6.4.4 Top-K Sampling

→ Input (go) & you get a probability distribution over |V|. Choose the top-k words. For k=3 suppose $P(x_{i_1}) = 0.1$, $P(x_{i_2}) = 0.2$, $P(x_{i_3}) = 0.3$

i) Normalize the probabilities as follows $P(x_{i_i}) = \frac{P(x_{i_i})}{\sum P(x_{i_i})}$

Thus $P(x_{i_1}) = \frac{0.1}{0.6} = 1/6$, $P(x_{i_2}) = 2/6$, $P(x_{i_3}) = 3/6$

ii) Now sample from these words with their associated Probs

→ This introduces stochasticity & a variety of sequences can now be produced.

6.4.5 Top-P Sampling (Nucleus)

→ In Top-K, we fix the number of tokens from which we sample. Suppose we have two distributions over the vocab as follows

i) [illegible] ii) [illegible].

In the first case summing up the top-5 tokens probabilities might give a mass of 0.5 whereas for the second distribution the top-2 alone may contribute to a mass of 0.6.

→ Thus a fixed number of tokens does not always make sense. Top-p sets the threshold parameter p and as many (few) tokens required to meet that is selected.

→ From this the Probs are normalized & we sample a token from that.

20/02/22

6.6 Masked Language Modelling - BERT

→ Predict the masked words using context words in both directions.
As the decoder is unidirectional, it cannot be used here.
↳ $P(y_i \mid x_1, x_2, \ldots, x_i = [\text{mask}], \ldots, x_T)$

↳ Independence of Predicted Tokens: If 3 out of 20 tokens are masked, each masked token produces an output assuming the other two are masked. There is no sequence i.e a leftmost occuring mask is not necessarily predicted first as its predictions are not used for the prediction of the remaining masked token.

→ We cannot use the same mask matrix used in CLM as the predicted probability of those tokens become 0. In CLM that made sense, however in this case we want to predict at the masked position.
↳ The task is to recover the original token
↳ Use [mask] token instead.

→ Only a select M words are masked as $L = \frac{1}{M} \sum_{i \in M} -\ln(\hat{y}_i)$

→ Typically 15% of the sequence is masked.
↑ masking ⇒ ↑ loss, Poor Representations
↓ masking ⇒ ↑ time, small gradients
Of the 15% 80% replaced with [mask],
10% with random word, 10% same word.

21/02/22

6.4.7 Next Sentence Prediction (NSP)

→ During training pairs of sentences (A,B) are given with special tokens [Sep.] in between & [CLS] at the start. [CLS] outputs whether the two sentences follow each other.

→ 50-50 split of consequent & random sentences. Pretraining with NSP improved performance of QA and other similar tasks.
↳ Recent Studies show NSP doesn't really benefit much

→ Token embeddings ($|V| \times d_{model}$) + Segment Embeddings ($2 \times d_{model}$) (2 to indicate the two sentences)
+ Positional Encoding ($T \times d_{model}$)

$$\min L = \frac{1}{n}\sum -\ln(\hat{y}_i) + L_{CLS}, \quad \hookrightarrow = -\ln(\hat{y})$$

6.4.9 Pretraining & Parameters

→ $|V| = 30{,}000$, $T = 512$, 12 layers, $d_{model} = 768$, 12 heads, 3072 FFN

Embedding Layer:
i) Token Embeddings : $|V| \times d_{model} = 30{,}000 \times 768 \approx 23M$
ii) Segment Embedding : $2 \times d_{model} = 2 \times 768 \approx 1536$
iii) Positional Encoding : $T \times d_{model} = 512 \times 768 \approx 0.4M$

1 Transformer layer:
i) W_Q, W_K, W_V : $3 \times (T \times \frac{d_{model}}{\#heads}) \times \#heads = 3 \times (512 \times 64) \times 12$
ii) Linear W_O : $d_{model} \times d_{model} = 768 \times 768 \approx 0.6M$
iii) FFN : $(d_{model} \times hidden) + (hidden \times d_{model}) + d_{hidden} + d_{model}$
$= (768 \times 3072) + (3072 \times 768) + 3072 + 768 \approx 4.7M$

21/02/24

6.4.9 Adapting to Downstream tasks.

15/03/2024

05.1 Tokenization - Challenges

→ So far we have considered each word as a token. In this case we split the text into words using whitespace and add all unique words in the corpus to the vocabulary.

i) enjoy, enjoyed, enjoying will be treated as different tokens & will have their own representation. We would want some relation between them

ii) Languages like Japanese have no word delimiters

→ If we instead use each character as a token, then the size of the vocab will be small. What should be the ideal size of vocab?

→ Larger vocab ⇒ larger embed matrix ⇒ greater computation & storage

→ How to deal with out of vocab words?

→ If a certain word did not occur during training, during inference it will have a default <unknown> representation. This may not be good take arso-phobia. Perhaps we can represent phobia and arso could be unknown.

→ Handling misspelled words?

→ Erroneous words should not be considered as unique

→ Open Vocab

→ In languages like german you can keep extending the word. In principle corpus is infinite. Need to handle this.

__/__/____

65-2 Motivation for Subword tokenization

→ The problem with character based tokenization is that a simple sentence of 20 words becomes 120 token sequence. So a lot of attention will have to be calculated for a simple sentence.
  ↳ for a context length of 512 tokens, we would only be able to pack in about 4-5 sentences.
  ↳ Essentially feeding in very little information

→ Word based tokenization & Character based tokenization have their issues. What we need is "sub-word" tokenization
  ↳ "don't" gets tokenized as "do" and "n't" as doing, does, doesn't; they all contain do and are related.
  ↳ Similarly "n't" appears in can't, won't, don't, etc

→ Three popular Subword tokenization methods
  i) Byte-Pair Encoding (BPE)
  ii) Word Piece
  iii) SentencePiece

__/__/__

§5.3 Byte Pair Encoding

→ First the text is normalized i.e lowercase all, remove accents, eliminate multiple whitespaces, handle HTML tags, etc. Splitting the word by whitespace was traditionally called tokenization however it is now considered pre-tokenization when used with Subword tokenization algorithm.

Input text → normalize → Pre-tokenize → Subword tokenization

→ In word based & character based tokenization, determining the vocab is easy as it is a set of all unique characters or words in the corpus. How do we determine the vocab for Subword? i.e how do we decide where to split the word?

→ Algorithm BPE:

i) Start with dictionary that contains words & their frequency

(Append words with </w>)

ii) Compute the character frequency for all words

iii) Count the frequency of all pairs of characters. Suppose (i,n) has the highest frequency, then add in to the vocab & adjust the character frequencies

iv) Repeat the same process (considering all pairs of characters including (i,n)

v) Repeat until some set number of merges (usually 32K)

→ Knowing is split into know, ing ∴ the fertility of knowing is 2 for character based tokenization, the fertility is the length of the word & for word based tokenization it is 1. In general aim for overall fertility rate between 1.5 & 3.

↳ The algorithm offers a way to adjust size of vocab as a function of number of merges.

__/__/____

05.4 Word-Piece Tokenizer

→ In BPE, if two character combinations (i,n), and (a,g) have the same frequency the we merge whichever appears first. However this has other repercussions & choosing the other could lead to a very different path.

→ So instead in WordPiece Tokenizer we calculate the score for pair of characters α, β as follows

$$\text{Score} = \frac{\text{Count}(\alpha, \beta)}{\text{Count}(\alpha)\ \text{Count}(\beta)}$$

05.5 SentencePiece Tokenizer

→ In BPE, a given word can have numerous ways with which it can be segmented even with the same vocab.

↳ BPE is greedy & deterministic (use dropout to make it stochastic)

↳ Given subword sequences, find the one that maximizes likelihood of the word.

→ Algorithm:

1. Start with a large vocab using BPE
2. E-Step: Estimate Probability of every token using freq counts
3. M-Step: For every word compute probabilities of all segments and return optimal segment that maximizes likelihood and segment the corpus accordingly.

Computationally expensive

4. Compute likelihood for each subword again & drop bottom 2x tokens

Repeat 2-4 until desired vocab size is reached.

→ CHECK VIDEO FROM 14:30 to understand a worked out example

__/__/__

§7.1 Intro to BART (Encoder-Decoder Model)

→ BERT is good at comprehension tasks like QSA because of its bidirectional nature. GPT is good at text generation because of its unidirectional nature.

→ A vanilla transformer with GELU activation function. The input sequence is corrupted in multiple ways: random masking, deletion, document rotation, etc.

→ The encoder takes in the corrupted sequence & the decoder predicts the entire sequence and computes the loss. (Think of it like a denoising AE)

§7.2 GPT-2: Prompting the model

→ So far we have studied pretraining a language model & then performing supervised finetuning over downstream tasks. However even this is computationally expensive.

→ Can we adapt a pre-trained language model to downstream tasks without any supervision? (ZERO SHOT TRANSFER)

↳ Yes! with a simple tweak to the input.

↳ Prompts are just words. Just change the single task I'm trying to P(output | input) ⇒ P(output | input, task)

where task is just a sequence of words prepended (or appended) to the input during inference.

→ LLM's are unsupervised multitask learners.

__/__/____

§ 3.3 Choices that affects Model Performance

→ Scale of the model, # training steps, LR, optimizer, objective (mum / cum)

1. Can we have the same objective for both pretraining & finetuning?
   ↳ For BERT, pre-training obj is MLM, and the finetuning objective for sentiment analysis is to use the [CLS] token for prediction i.e pretraining & finetuning objective is diff

2. Can we have an architecture that uses both CLM & MLM obj?

3. Can we use a pre-training dataset which is as large as possible (to understand the effect of size of unlabelled data)?

§ 3.4 Baseline Model

→ 12 encoder-decoder stacks with denoising objectives. Contiguous tokens corrupted.
   ↳ The denoising objective reduces the training time as it doesn't have to generate the full sequence. Only the uncorrupted tokens
   ↳ Batch size = 2^7 = 128, Sequence length = 2^9 = 512
   ↳ # of Training steps = 2^{19} = 524,288
   ↳ Total # of tokens in pretraining = $2^{19} \times 2^7 \times 2^9 = 2^{35}$ = 34 Billion

→ For finetuning, 2^{18} steps so 17B tokens were seen

→ Pretraining improves model performance for small sized datasets across a variety of tasks. Supervised pretraining has a slight edge over pretraining with big datasets.

__/__/__

§7.5 Experiment with different architectures

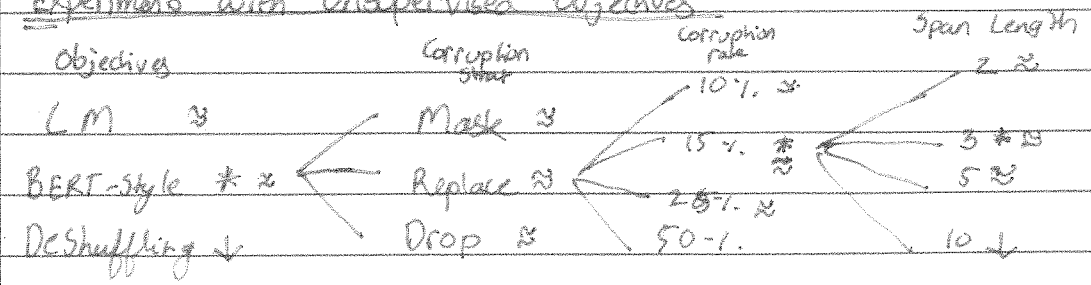
→ Architectural Variants : [Denoising (mLM) ; [LM ; LM with Prefix]

→ In Prefix LM we want the input sequence to be attended bidirectionally and the task output sequence autoregressively.
↳ Consider QA : The context & query should always be paid attention to (bidirection) & the generated tokens causally.
↳ Use encoder for input sequence & decoder for output
Or use decoder with prefix-mask.

→ How do we compare across encoder-only, decoder-only or encoder-decoder models?
i) # of Parameters
ii) amount of compute across an input & target sequences
However, you can't have both. The number computations of an L-layer encoder and L-layer decoder model is roughly the same as that of a L-layer decoder only model.
↳ In encoder-decoder input goes through enc & output goes through dec. } same comp
↳ For dec only, inp & out both go through dec

→ Denoising objective always results in better downstream task performance compared to Language Modelling

§7.6 Experiments with unsupervised objectives



objective	Input	Target
Prefix LM	Thank you for	inviting me ...
DeShuffling	Jumbled input	Original text
BERT-STYLE	Thank You <M> to <M>	original text

__/__/____

§ 7.7 Effect of size of pretraining dataset

⇒ Do the scale & quality of the pretraining dataset affect the performance in downstream tasks?

i) Just adding a lot of data if quality is not good is not useful ⇒ Quality is important

ii) Pretraining on in-domain unlabeled data can improve performance on downstream tasks.

→ Repeating the tokens degrades performance as the model might have memorized the pretraining dataset.

↳ Using more unique content is better than repeating

§ 7.8 Finetuning Strategies

⇒ Finetuning all parameters in the model is computationally expensive

i) Adaptive Layers: Add an adapter layer after FFN block in each layer. It is simply a dense-ReLU-dense layer with inner dimension d & residual connections.

↳ During finetuning only update the parameters for the adapter & layer Norm.

↳ Significant drop in performance over finetuning all parameters. Need higher value for d to achieve reasonable performance

ii) Gradual Unfreezing: Freeze the parameters of all the layers. Gradually unfreeze parameters starting from the top.

↳ Happens simultaneously for k^{th} layer encoder & decoder

↳ only one layer at a time is updated.

↳ Drop in performance, not as bad as Adaptive Layers.

__/__/__

§7.9 Multi-task Learning

→ In general multitask learning pretrain using the labelled dataset meant for finetuning as well.
  ↳ Summarize : " " } Everything becomes text-in
  ↳ Translate : " " } text-out

→ However, the general language modelling dataset will constitute the biggest share of the total data ⇒ Use Sampling
  ↳ K : Artificial limit to the size of datatasets (total # of i^{th} task)
  e_n : # of samples in each task where $n \in \{1, 2, \ldots, N\}$
  then P(seeing sample from m^{th} task) is

$$r_m = \frac{\min(e_m, K)}{\sum \min(e_m, K)}$$

  ↳ Temperature Mixing :
  Instead of using r_m, use $(r_m)^{\frac{1}{T}}$ as $T \to \infty \Rightarrow$ Uniform

→ Pretraining + Finetuning is still superior to Multitask pretraining. However Multitask pretraining + Joint Finetuning works well. Leave-one-out was only slightly worse suggesting that a model trained on variety of tasks can adapt well to new tasks

§7.10 Pushing the limits.

→ If we have more compute available should we increase the model size or train it on a larger number of tokens?
  ↳ Scaling parameters requires more compute during fine-tuning and also leads to a longer inference time.

Larger model trained for a larger number of steps is optimal.

__/__/____

E2-1 Big Picture – Road Ahead

__/__/____

§8.2 Motivation for scaling data size

→ We know from the TS paper that the models scales with data.

1. Is it necessary to scale even further?
2. How do we source the data?
3. What aspects do we need to consider when building a dataset?
   ↳ Replication, Toxicity, Bias

→ The TS study demonstrated that the model overfits if data is repeated over many epochs ⇒ need to scale the dataset.

→ Given the number of non-embedding parameters (N) and dataset (D), the test loss is

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

↳ This suggests that if # Parameters $\gg$ # of tokens, training error can be driven to 0, However the test loss changes according to scaling laws

↳ An increase in either data size or # parameters leads to an improvement in test loss only upto a certain point. Both need to be scaled appropriately.

→ Not just the size, but diversity & data quality are also important.

__/__/____

Ch 3 + Ch 4 **Data Sourcing, Cleaning & Pre-Processing Pipeline**

→ The pipeline : www ⇒ Source Data (webpage, Arxiv, wiki) ⇒ Clean (deduplication, quality fil) ⇒ Dataset (CH, mCH) ...

↳ The Cleaning pipeline is not available for all datasets

→ To get a high quality dataset, the pipeline needs to

1. Detect language : There exists many libraries like FastText that can detect language

2. Deduplicate : There are two kinds of deduplication

	Exact (Expensive)	Fuzzy
When	If sentence 1 = Sentence 2	If Sentence 1 ≈ Sentence 2
How	Suffix Array	Similarity measure using hashing/ML ↳ MinHash ; SimHash, Locality sensitive Hash

3. High Quality : Is usually ensured with LM model trained on Wiki or Books ; Simple Heuristics

↳ CCNet Pipeline : Designed for extracting HQ monolingual pretraining data

As BookCorpus & Wiki are primarily english, we need to use CC dumps & clean them . After Paragraph level deduplication, divide page into three parts and obtain quality score for each portion using pretrained LM on a targeted Domain.

4. Remove Toxic Content : Any page that contains a bad word is removed. 5. Bias Removal

→ Perfect deduplication is difficult. However duplication allows the model to memorize, wrong test perf, privacy risks . ↑ Parameters ⇒ ↑ effect of duplicated content . A sequence present 10 times in the data ⇒ 1000x generated more often . Performance can get halved if 0-1/. is repeated 100 times .

__/__/____

§8.5 Pre-training Datasets

→ Instead of comparing datasets on raw size, one should compare it on the size of tokens (scaled to scaling law)

→ Utilizing high-quality and diverse datasets is crucial for achieving optimal downstream performance.

§11.2 Motivation for fast attention mechanism, time & space complexity

→ The models seen so far have a limited context length of size or seq
But what if we want to summarize/generate a long passage
↳ A 10s audio file at 8K SR ⇒ 80,000 sequence length
or a 100×100 px image ⇒ 30,000 sequence length
↳ Why is it difficult to scale the context length?
One of the primary reasons is the computational complexity of the self attention mechanism.

→ When getting the attention matrix, we need to calculate
i) QK^T : $O(T^2d) : (p \times q) \times (q \times r) \Rightarrow O(pqr)$
ii) $A = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)$: $O(T^2)$ Because we need to normalize each row of QK^T with each row taking $O(T)$ time
iii) AV : $O(T^2d)$: $(T \times T) \cdot (T \times d)$
⇒ Therefore the self-attention block is $O(T^2d)$

→ Memory for Attention
$mha = (Q_{T \times d} + K^T_{d \times T} + V_{T \times d})\, n_h = 3n_h(T \times d) = 3T \times d_{model}$
↳ For a batch of B samples : $3BTd_{model}$
↳ $Z_{T \times T} \Rightarrow (T^2)n_h \Rightarrow Bn_hT^2$
↳ $A_{T \times T} \Rightarrow (T^2)n_h \Rightarrow Bn_hT^2$

→ Memory Scales up linearly w.r.t batch size & quadratically w.r.t T

__/__/____

6.11.2 Fast Attention Mechanisms : Approaches

→ As we saw that the full attention mechanism has a quadratic $O(T^2d)$ complexity, we want something that is sub-quadratic like $O(cTd)$.

↳ A study on BERT concluded that

i) Neighbouring inner products are extremely important

ii) Not all heads attend to all tokens ⇒ Some paid attention to fullstops, some to [sep], others to next token

↳ Instead of all T^2d, the heads are learning something constrained. Can we localize the attention?

→ Strided Local Attention: Pay attention only to neighbouring tokens so now you have $O(cTd)$

→ Random Local Attention: Sometimes compute some local, otherwise compute attention wrt random tokens.

→ Sparse Block Attention: Blocks of tokens ie first n tokens only attend among themselves and the next n tokens only attend among themselves.

→ Global + Local attention: Most of the attention is in the neighbourhood except some tokens which attend to all tokens.

→ Flash Attention: A hardware aware attention mechanism.

__/__/____

11.3 Local Attention Mechanisms

→ In Strided Local attention, every token attends to a window of c words, $\lfloor \frac{c}{2} \rfloor$ to the left, $\lfloor \frac{c}{2} \rfloor$ to the right & one with itself
↳ So now the complexity is $O(cTd) \Rightarrow$ Linear
↳ We could vary the size of the window for each layer in the model such that the topmost layer uses the full global attention.

↳ We could also employ dilation.

→ In Global + Local attention, every word attends to c words similar to Strided Local attention, however a few tokens have global attention
↳ Why? Tokens like [CLS] require full attention to get good performance in downstream tasks.

↳ For c neighbours & k tokens that require global attention the complexity is $O((T-k)k + kTd) = O([cT+T]d)$

↳ A variant of this also pays attention to some random tokens in addition to the global and local attention.

__/__/__

6114 Sparse Block Attention

→ Recall that to compute the attention matrix we have

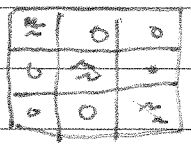
$$Q_{\text{first}} K^T_{\text{first}} = \begin{bmatrix} Q_0 \\ Q_1 \\ Q_2 \end{bmatrix} \begin{bmatrix} K_0^T & K_1^T & K_2^T \end{bmatrix} = \begin{bmatrix} Q_0K_0 & Q_0K_1 & Q_0K_2 \\ Q_1K_1 & & Q_1K_2 \\ & & Q_2K_2 \end{bmatrix}$$

↳ Therefore we can divide them into blocks. In the full attention, we compute all the blocks but in sparse ~~block~~ block attention we may compute only one block per row.

↳ If we are dividing the Q & K matrices into n blocks, the QK^T has n^2 blocks.

↳ The complexity of sparse block attention is $O(\frac{I^2}{n} d)$ i.e as you increase the number of blocks, the less time it will take.

→ Note that each head could have a different permutation to choose which blocks interact. Let $\Pi = \text{perm}(0, \dots, n-1)$, then $M_{ij} = 1$ if $\Pi(\lfloor \frac{i}{n} \rfloor) = \lfloor \frac{j}{n} \rfloor$ & $Z = \text{softmax}(QK^T \odot M)$



$\Pi = (0, 1, 2)$

$\Pi = (0, 2, 1)$

• Note that in the second example tokens from Block 0 interact with all tokens from block 0. However tokens from Block 1 interact with tokens from Block 2.

$Q = (Q_0, Q_1, Q_2)$

$K = (K_{\Pi(0)}, K_{\Pi(1)}, K_{\Pi(2)})$, $V = (V_{\Pi(0)}, V_{\Pi(1)}, V_{\Pi(2)})$

→ As n increases, the attention matrix becomes more sparse.

→ Blockwise sparsity capture both local & long-distance dependencies in a memory efficient manner using diff perm for diff head

→ Block-wise attn reduces training time, increases perplexity score (bad),

__/__/__

E115 Low Rank Approximation

→ The methods we have seen so far localize or sparsify the attention matrix in different ways where the theoretical time complexity is sub-quadratic $O(cTd)$.

→ Empirically it is observed that the self-attention matrix is low-rank especially in the top layers.

↳ For $A \in \mathbb{R}^{512 \times 512}$ ie when the sequence length was T, it was observed that the top 128 eigen-values were the most important.

↳ Since the SVD of A is given by $A = U \Sigma V^T \Rightarrow$ we don't need all 512 singular values. Using 128 we can approximate A.

→ Note that $Q \in \mathbb{R}^{T \times d}$, $K \in \mathbb{R}^{T \times d}$, $V \in \mathbb{R}^{T \times d}$

↳ Introduce two learnable projection matrices $E, F \in \mathbb{R}^{k \times T}$

↳ Now $A = \text{Softmax}\left(\frac{Q(EK)^T}{\sqrt{d}}\right) \Rightarrow Sm\left((T \times d)\underbrace{(k \times T)(T \times d)}_{O(kTd)}{}^T\right)$

Essentially the attention matrix is be approximated in $O(kTd)$ instead of $O(T^2d)$ $\quad Sm\left(\underbrace{(T \times d)(d \times k)}_{O(kTd)}\right)$

↳ $A \in \mathbb{R}^{T \times k} \Rightarrow A(FV) : \underbrace{(T \times k)}_{O(kTd)}\underbrace{((k \times T)(T \times d))}_{O(kTd)} = \underbrace{(T \times k)(k \times d)}_{O(kTd)} = T \times d$

→ The complexity is $O(kTd)$ where k is the rank which can be set according to the error.

__/__/____

E116 Fast Inference Mechanisms

→ During inference, you take the token [start] & give it to the model.

i) Its embedding h_0 is computed

ii) Using W_q, W_k, W_v we get q_1, k_1, v_1 & then also a new output.

iii) The [start] token and output from $t=0$ is supplied to the model at $t=1$. Again the embeddings are calc & again q_2, k_2, v_2 along with q_1, k_1, v_1 are computed.

↳ At each timestep we compute the key-values for previous queries as well which is a waste of compute & increases latency.

→ Simple idea is to cache keys & values. In future timesteps reload KV from cache.

↳ This way we tradeoff memory for compute which scales linearly $O(n)$

↳ Since KV caching uses GPU RAM where model weights are stored, we cannot let KV cache grow indefinitely.

↳ KV-Cache PER TOKEN:

$$\underset{(1 \text{ for } K,V)}{2} \times \#\text{layers} \times (\#\text{heads} \times d_k) \times \text{precision in bytes}$$

For 96 layer, 96 headed, $d_k = 128$ with 4 byte precision $\Rightarrow$ 9.4 MB per token

For then entire sequence of 2k $\Rightarrow$ 19.3 GB

↳ Either drop KV's from the cache that occurred in the distant past or do some model level modifications.

__/__/____

→ Multi-Query Attention (MQA): Since the KV-cache is dependent on # layers, #heads, d_k, precision; if reduce any of those we can reduce the amount of memory required.

↳ If we share the key & value vectors across all heads then #heads becomes 1. This greatly reduces cache memory however it degrades performance ⇒ Retrain with 5% training da

→ Grouped-Query Attention (GQA): On one end we have MHA where each head has its own K-V vectors. On the other end we have MQA where all heads share the same vector.

↳ GQA is a middle ground where a group of heads share K-V vectors

↳ So if there are 32 heads & Group size is 4, heads 0,1,2,3 share KV, then heads 4,5,6,7 share a KV

→ Paged Attention (PA): Typically fixed & contiguous memory is reserved for KV-caching (usually based on context length).

↳ If context length >> length of the sequence, memory is wasted.

↳ use blocks of non-contiguous memory which allows others to use unused memory.

__/__/____

§12-1 Motivation for Relative Positional Encoding

→ Previously we used positional embeddings to capture the order of tokens in input sequences. We used absolute (fixed) positions & added the position vector with the word embedding.

→ It is hypothesized that Absolute Positional Embedding does learn to attend to tokens by relative positions.
  ↳ Some studies were done wherein you supplied sentences which had positional Embedding encoded as (0-20) let's say.
  ↳ Then the same sentence was supplied with positional Encoding (100-120).
  ↳ As you shifted the position, the performance degraded ie the model is sensitive to absolute positional embeddings and was not learning the relative positions.
  ↳ This motivated the need to explicitly encode relative positions.

→ Another issue with APE is that the model cannot generalise beyond the context length ⇒ Suppose you trained a model with a context length of 512, and you want to produce an output sequence of 1024, PE's for 513 to 1023 do not exist.

→ Similarly if context length was 512 but all sequences during training were 128, then during inference APE will struggle with sequences longer than 128.

__/__/____

6.12 Relative Positional Encoding

→ In APE, each token has one positional encoding attached to it. In RPE, each token has T encodings attached to it since it encodes pairwise relationships between tokens.

↳ The orange goes from $[-(T-1), T-1]$ as here. There are a total of $2T-1$ PE.

↳ Recall that,

$$q_i = h_i W_Q = (x_i + p_i) W_Q \quad , \quad k_j = (x_j + p_j) W_K$$

$$e_{ij} = q_i k_j^T = (x_i + p_i) W_Q W_K^T (x_j + p_j)$$

$$= x_i W_Q W_K^T x_j + \boxed{x_i W_Q W_K^T p_j \quad p_i W_Q W_K^T x_j} + p_i W_Q W_K^T p_j$$

- $x_i W_Q W_K^T x_j$: Score of key is query w/o PE
- $x_i W_Q W_K^T p_j \quad p_i W_Q W_K^T x_j$: Correlation between word & PE (Does not make sense)
- $p_i W_Q W_K^T p_j$: Adding a constant that comes from PE

↳ PE just adds a constant to pre-attention score ⇒ We can inject position information in the attention layer instead of adding to the word embedding & doing loads of calculations.

→ $e_{ij} = x_i W_Q (x_j W_K + p_{ij}^K)^T$: Previously p_{ij} added to the word embeddings & was therefore was in $R^{d_{model}}$. Since now we are directly injecting, $p_{ij} \in R^{d_k}$

↳ This formulation still can't generalise to sequences longer than T.

↳ Clip the relative distance beyond a certain $k < T$ as k. i.e if $T = 512$, treat positions after 256 as k. Now even if sequence length is longer than T, it will work because k will be taken as the distance. ($2k+1$ embeddings) making it independent of T.

↳ Once softmax is taken, p_{ij}^V is also computed for the value vectors. p_{ij}^K & p_{ij}^V are typically shared across heads (not layers)

__/__/____

Slide Rotary Positional Embedding (RoPE)

→ Consider two sentences "the book" and "read the book". In APE, the representation for the is book in the first sentence is different from the representations for the is book in the 2nd sent. This is because (P_0, P_1) vs (P_1, P_2) gets added to the word embedding. ⇒ Not good.

→ In RoPE, the angle between the same words with same distance is preserved no matter the sequence.

↳ Angle between the is book preserved in both sentences since distance is the same. The vector representation might change but angle is preserved.

→ Previously we had,

$f_q(x_m, m) = x_m W_Q$ & $f_k(x_n, n) = x_n W_K$

and we performed $\langle f_q, f_k \rangle$

↳ If instead we use $f_q(x_m, m) = x_m W_Q e^{jm\theta}$ & $f_k(x_n, n) = x_n W_K e^{jn\theta}$

where j is in the imaginary space then

$\langle f_q, f_k \rangle$ will use the hermitian (complex conjugate) instead of transpose & you get $x_m W_Q (x_n W_K)^T e^{j(m-n)\theta}$

↳ So e^{*} remains the same across relative distances.

→ Unlike APE, RoPE needs to be added in every layer as doesn't need to be added to the v vector.

__/__/____

2024 ALiBi & NoPE

→ Length generalisation is the ability of a model to extrapolate to longer sequences than the one it has seen during training
↳ the ALiBi paper showed that pretty much all methods performed poorly.

→ ALiBi just adds a constant negative bias to the attention score based on position
↳ Each head has a specific scalar. The attention score computed is as follows: $\text{Softmax}(q_i K^T + m[-(i-1), \ldots, -2, -1, 0])$

↳ observe that since we are adding negative values, larger values are getting added for nearby tokens than distant tokens. This effectively acts as a local windowed attention that enables us to predict next token effectively from the past k tokens.

(L-1 x -128)

→ NoPE conjectures that using causal mask (decoder-only) implicitly encodes the position information. It can represent both absolute & relative PEs & seems to use relative PE in practice.