Contents:

1. Neural Language modelling
2. RNN's and LSTMs
3. Transformers
4. Decoding
5. Pre-training
6. Scaling Laws of LLMs
7. LM Assistants and RLHF policies
8. PEFT Techinques (LORA, Adapters, pruning)

Neural language modeling, RNNs :-

- language model is a system that assigns probability to a piece of text.
- let us have texts $x^{(1)}, \ldots, x^{(T)}$, then probability of this text is

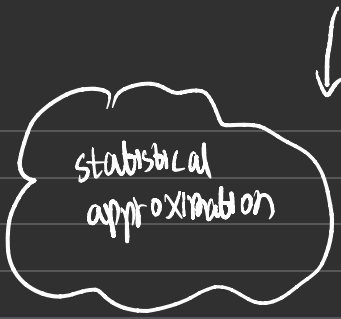
$$P(x^{(1)}, \ldots, x^{(T)}) = P(x^{(1)}) \times P(x^{(2)}/x^{(1)}) \times \ldots \times P(x^{(T)}/x^{(T-1)} \ldots x^{(1)})$$

$$= \prod_{t=1}^{T} P(x^{(t)}/x^{(t-1)}, \ldots, x^{(1)}).$$

N-gram language modeling :-

- $x^{(t+1)}$ depends only on the preceeding $n-1$ words.
- $P(x^{(t+1)}/x^{(t)}, \ldots, x^{(1)}) = P(x^{(t+1)}/x^{(t)}, \ldots, x^{(t-n+2)})$.

$$= \frac{P(x^{(t+1)}, x^{(t)}, \ldots, x^{(t-n+2)})}{P(x^{(t)}, \ldots, x^{(t-n+2)})}$$



$$\approx \frac{\text{count}(x^{(t+1)}, \ldots, x^{(t-n+2)})}{\text{count}(x^{(t)}, \ldots, x^{(t-n+2)})}$$

Ex :- students opened their ___ (w)

condition on this

use a 4-gram language model

$$P(w \mid \text{students opened their}) = \frac{\text{count}(\text{students opened their } w)}{\text{count}(\text{students opened their})}$$

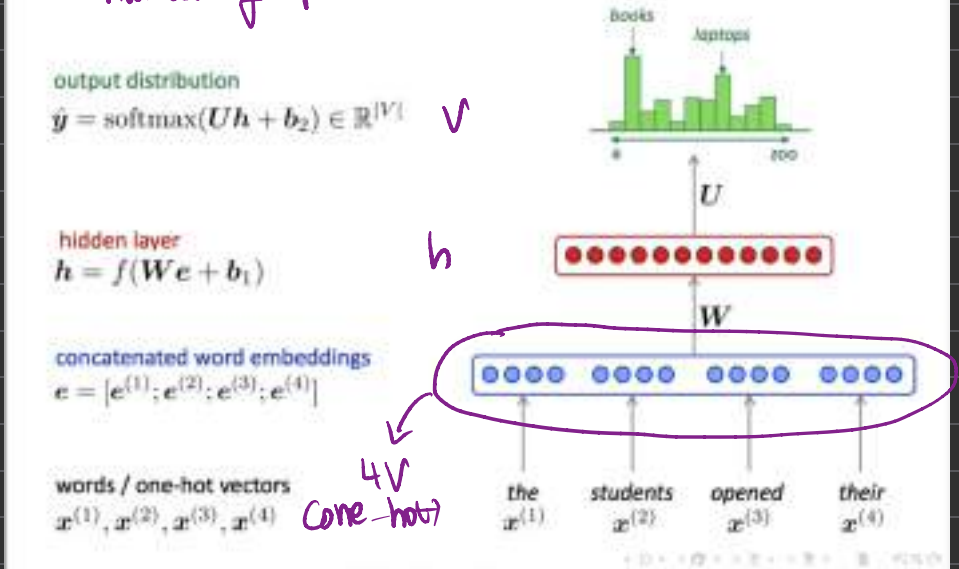
V^n → need to store count for all n-grams you saw in the corpus.

A fixed-window Neural language model :-

- It operates on a fixed-length sequence of words
- the model predicts the target word based on the probability distribution generated by it.

A fixed-window neural language model

number of parameters = $4Vh + Vh = 5Vh$



Problem in one-hot encoding :- the vectors are orthogonal, and there is no notion of similarity between one-hot vectors.

word2vec representation :-

Any word w_i in the corpus is given a distributed representation by an embedding

$$w_i \in \mathbb{R}^d$$ → a d-dimensional vector

Basic idea !

→ in a large corpus of text every word is represented by a vector.

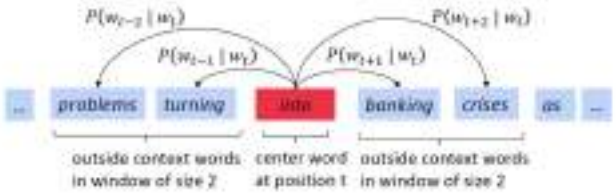
→ go through each position `t` in the text which has a center coord `c` and context coords `o`.

→ use the similarity of the coord vectors for c and o to calculate the probability of o given c.

→ keep adjusting the coord vectors to maximize this probability.

Word2Vec (Skip-gram) Overview

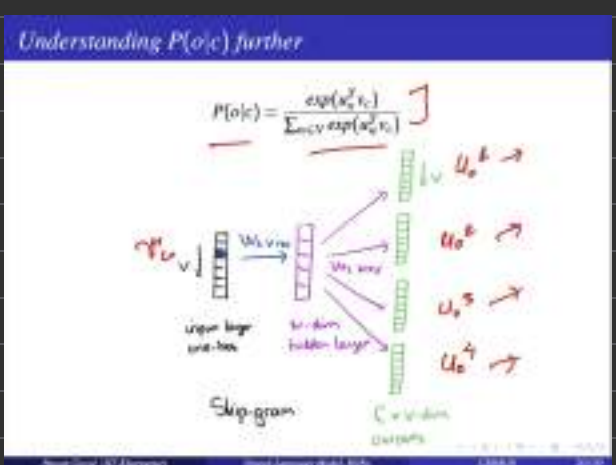
Example windows and process for computing $P(w_{t+j}|w_t)$



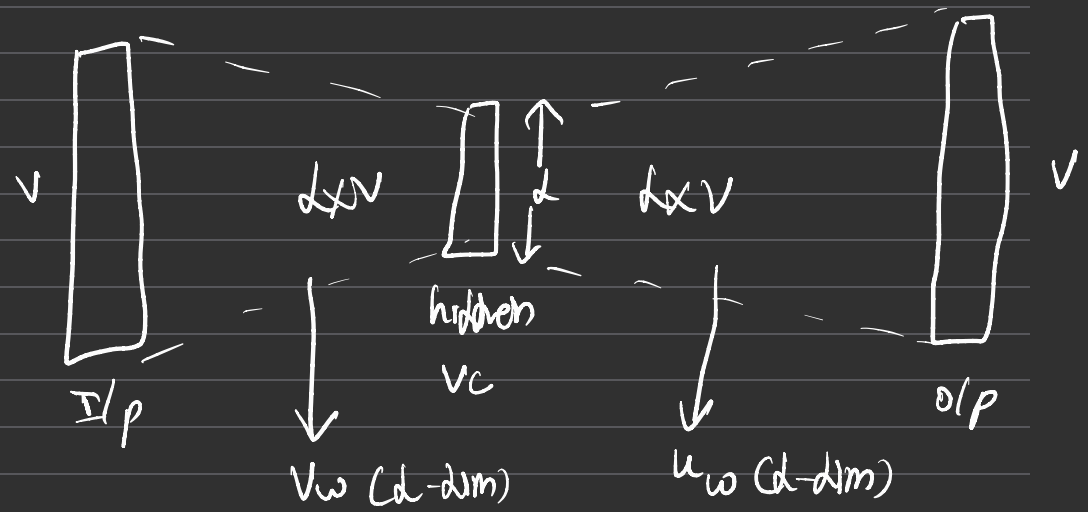
- $$J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log P(w_{t+j} \mid w_t; \theta)$$

we will use two vectors per word w:
- v_w when w is the center word
- u_w when w is the context word

$$P(O|C) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$



for a target word,

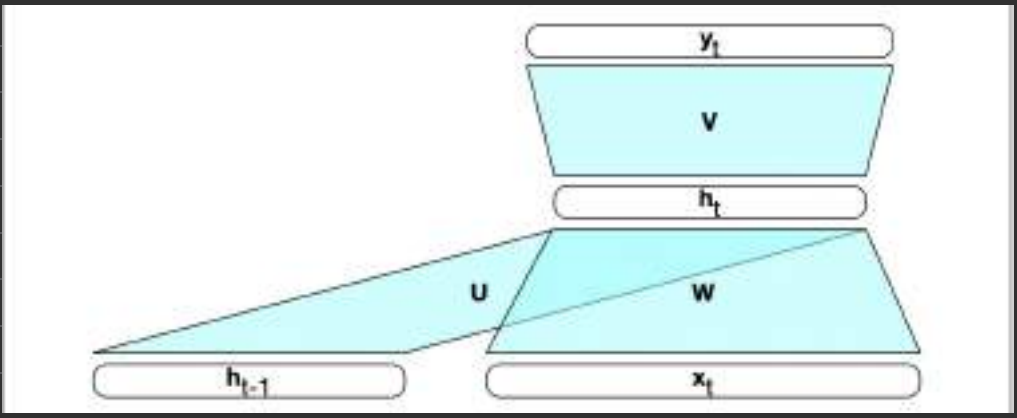


2dV → weights

Recurrent Neural Networks :-

- Apply the same weights repeatedly.
- We can process a sequence of vectors x by applying a recurrence formula at each step:

$$\underbrace{h_t}_{\text{new state}} = f_w(\underbrace{h_{t-1}}_{\text{old state}}, x_t)$$

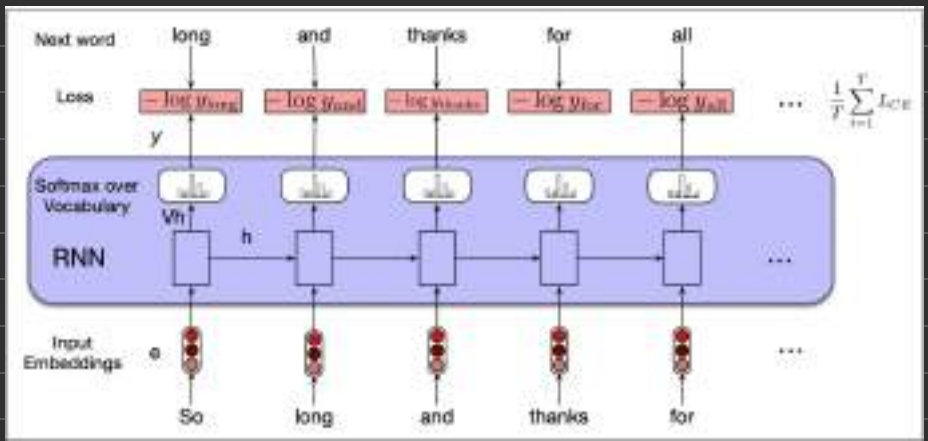


$h_t = f(Uh_{t-1} + Wx_t)$ $\quad W :- d_h \times d_{in}$

$y_t = \text{softmax}(Vh_t)$ $\quad U : d_h \times d_h$

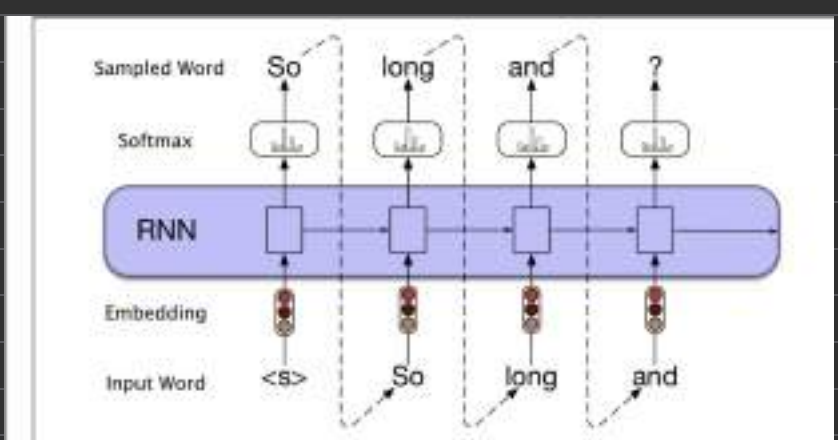
$V : d_{out} \times d_h$

- we simply train the model to minimize the error in predicting the true next word in the training sequence.



$$L_{CE}(\hat{y}_t, y_t) = -\log \hat{y}_t[w_{t+1}]$$

Autoregressive Generation with RNNs :-

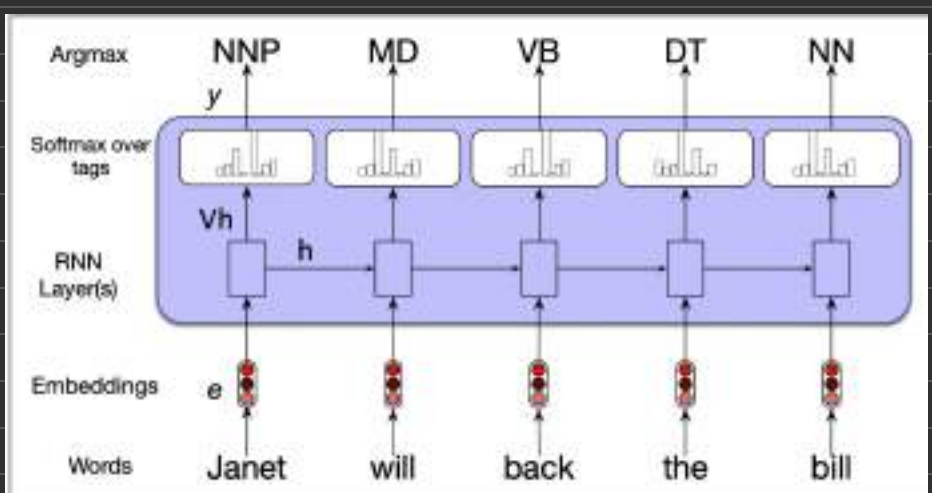


- All your parameters have already been trained.
- start with special token <s>
- Through forward propagation, obtain the probability distribution and sample a word.

- Feed the corresponding word vector as input for next time step

- continue generating until the end of sentence token is sampled or a fixed length of sentence has been reached.

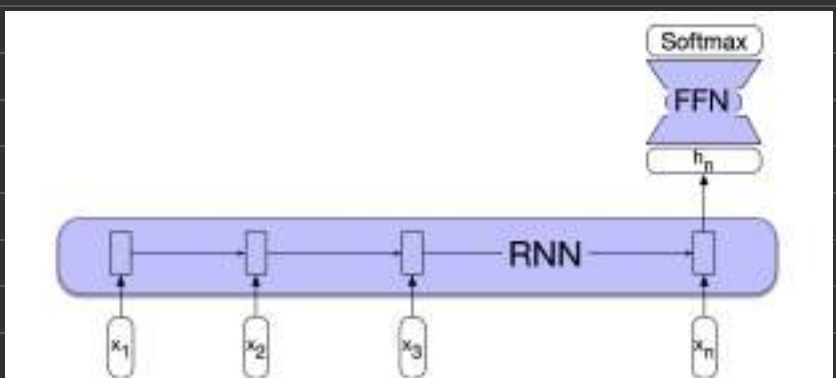
Sequence labeling and Sequence Classification with RNNs :



parts of speech tagging

The network's task is to assign a label chosen from a fixed set of labels to each element of a sequence.

Sequence classification :-



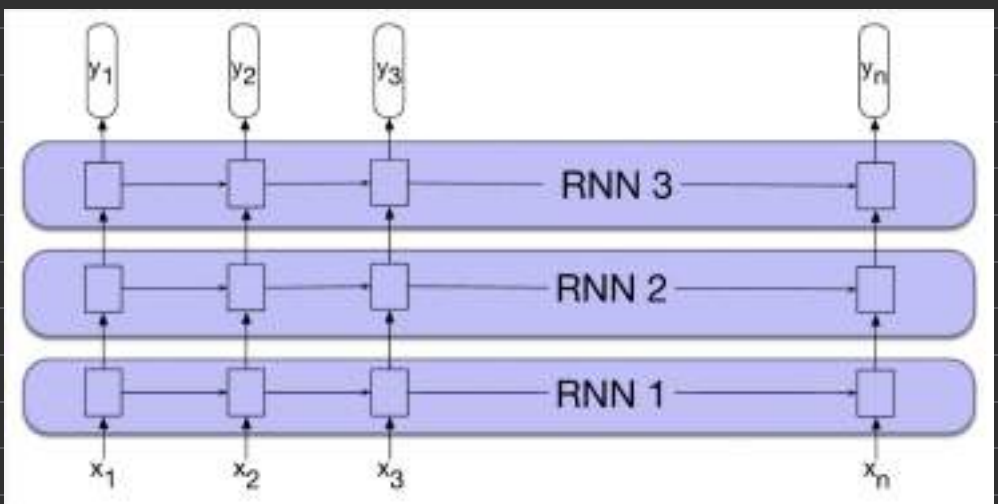
We pass the text to be classified through the RNN a word at a time generating a new hidden layer at each time step.

- The hidden layer of the last token, h_n, to constitute a compressed representation of entire sequence.

Alternate method :-

$$h_{mean} = \frac{1}{n} \sum_{i=1}^{n} h_i$$ → feed it into FFN

Stacked RNN's :-



The output of lower level serves as input to the higher levels with the output of the last network serving as final output.

Bi-Directional RNN :-

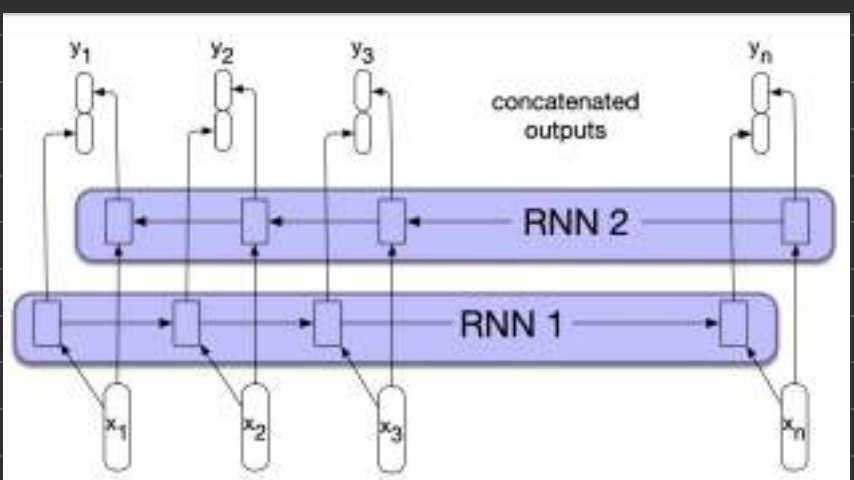
- in the left-to-right RNN

$$h_t^{\rightarrow} = RNN_{forward}(x_1, \dots, x_t)$$

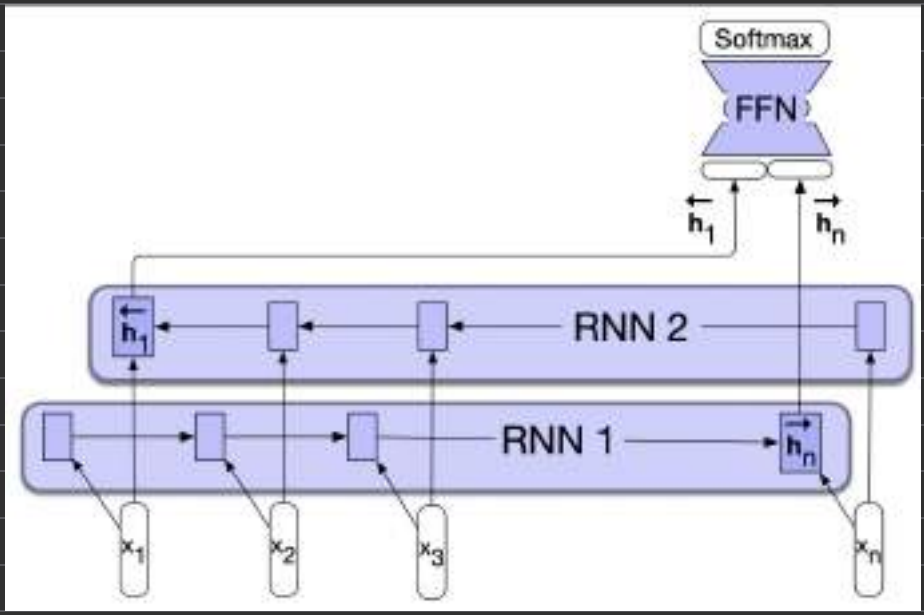
- in the right-to-left RNN

$$h_t^{\leftarrow} = RNN_{backward}(x_t, \dots, x_n)$$

$$h_t = h_t^{\rightarrow} \oplus h_t^{\leftarrow} \Rightarrow y_t = softmax(V h_t)$$



A bi-directional RNN for sequence classification:-



- A difficulty with the general approach is that the final state naturally reflects more information about the end of sentence than its beginning. So bidirectional-RNN will solve this problem.

Problems in RNNs :-

- model weights are updated only with respect to near effects, not long-term effects.

- Gradient signal from faraway is lost because its much smaller than gradient signal from close by.

- **syntactic recency** :- The writer of the books is (✓)

- **sequential recency** :- The writer of the books are (✗)

- Due to vanishing gradient RNN-LMs are better at learning from sequential recency than syntactic recency.

- In RNNs, the hidden state is constantly being rewritten

Long-short term memory :- (LSTM)

- LSTMs divide the context management problem into two subproblems :-
  1. removing information no longer needed from the context.
  2. Adding information likely to be needed for later decision making.

- it uses gates to achieve the above two tasks. it uses additional weights.

- The gates in LSTM share a common design pattern; each consists of a feed forward layer followed by a sigmoid activation function; followed by a pointwise multiplication.

forget gate :-

$$f_t = \sigma(v_f h_{t-1} + w_f x_t)$$
$$k_t = c_{t-1} \odot f_t$$

- extract the actual information from previous hidden state.

$$g_t = \tanh(v_g h_{t-1} + w_g x_t)$$

add gate :-

$$i_t = \sigma(v_i h_{t-1} + w_i x_t)$$

$$j_t = g_t \odot i_t$$

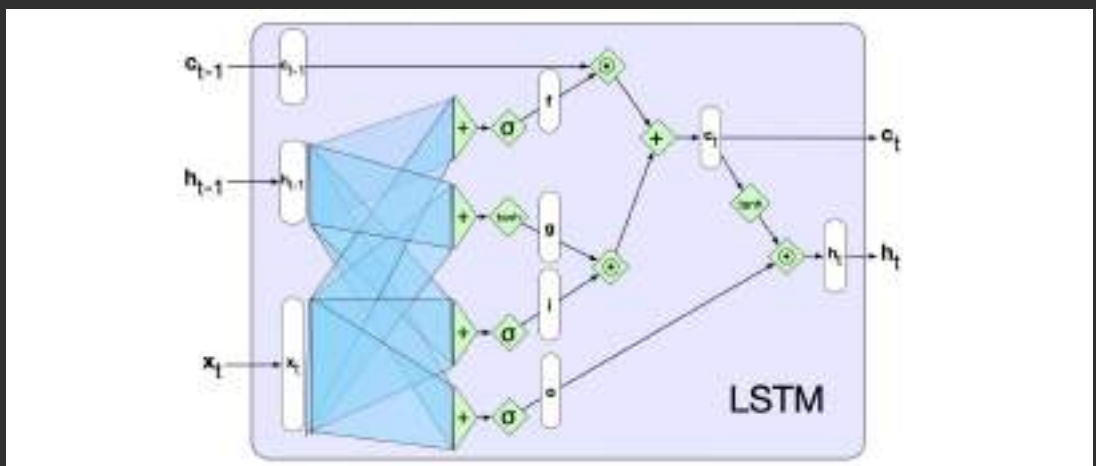
$$\Rightarrow \boxed{c_t = j_t + k_t}$$

output gate :-

$$o_t = \sigma(V_o h_{t-1} + W_o x_t)$$

$$h_t = o_t \odot \tanh(c_t)$$

- An LSTM as input the context layer, and hidden layer from the previous step.



Encoder-Decoder Networks :-

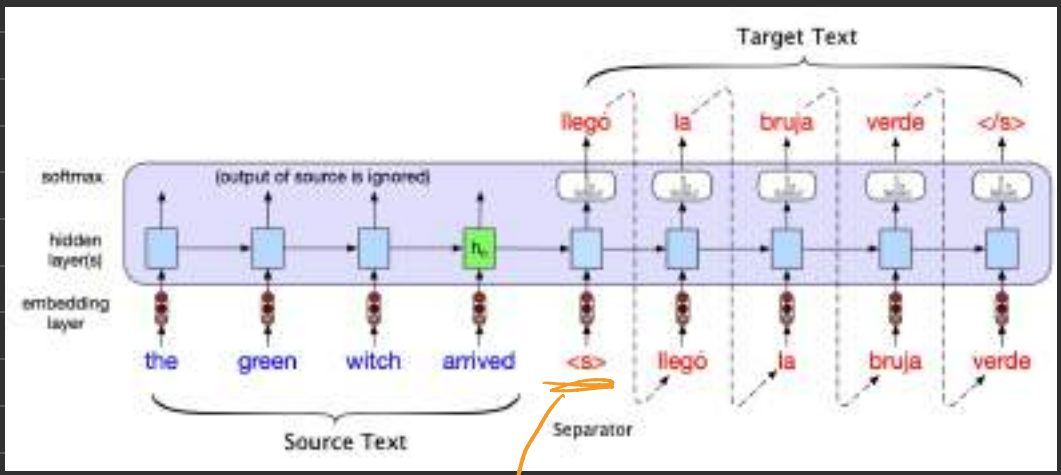
- Encoder :- accepts an input sequence $x_{1:n}$ and generates an corresponding sequence of $h_{1:n}$

- Context :- A context vector c, which is a function of $h_{1:n}$, acts as input to

decoder.

- **Decoder** :- who accepts C as input and generates $h_{1:m}$ and corresponding output $y_{1:m}$ is generated

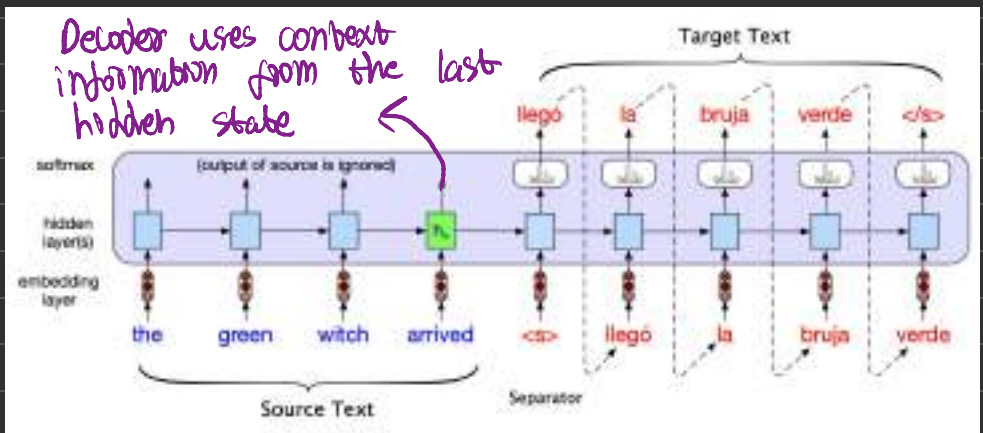
Machine Translation :-

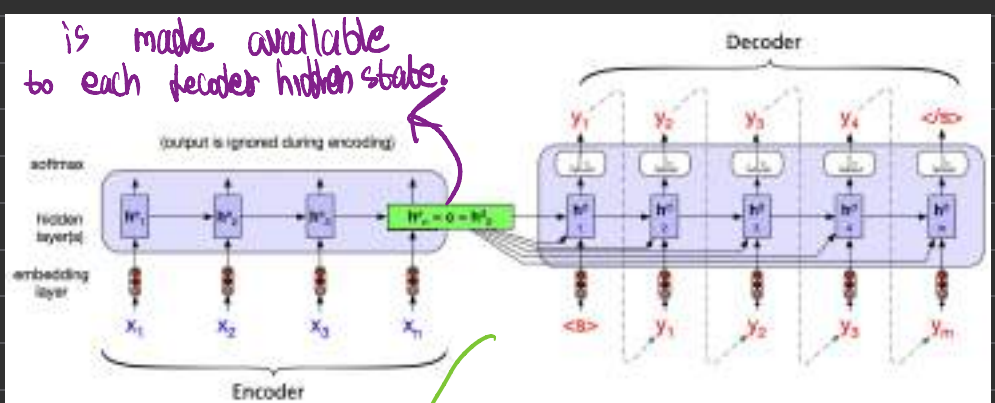


sentence seperator token

x → Source text <s> → sentence seperator.
y → target text

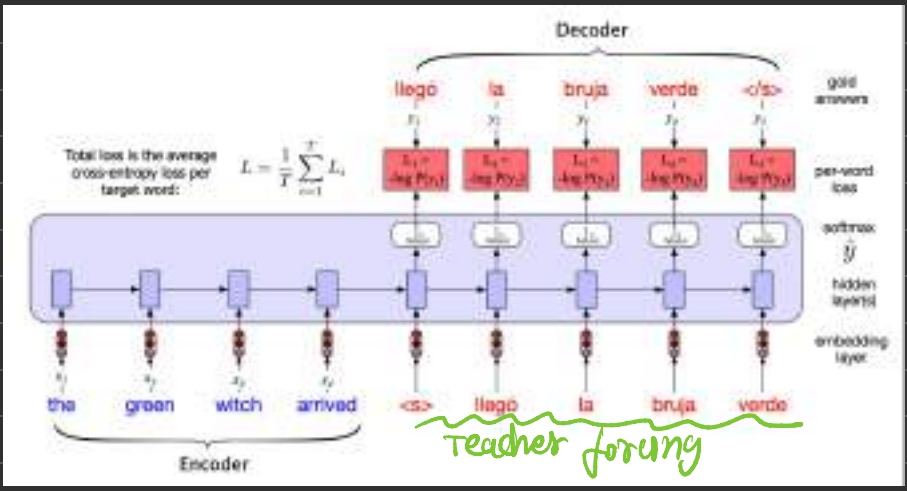
$$P(y/x) = P(y_1/x) \cdot P(y_2/y_1, x) \cdot P(y_3/y_1, y_2, x) \cdots P(y_m/y_1, \ldots, y_{m-1}, x)$$





$$h_t^d = g(y_{t-1}, h_{t-1}^d, c)$$

Training an Encoder-Decoder model :-



- It is trained autoregressively to predict the next word.

- Teacher forcing is used during training i.e. the system is forced to use the gold target token from training as the next input x_{t+1} rather than relying on the last decoder output.

Not during inference !

★ the last hidden state acts as the bottleneck as it has to represent absolutely everything about the source text.

Solution :- Attention mechanism :-

→ context vector, c is $c = f(h_1^e, \ldots, h_n^e)$, we can't directly use the entire set of encoder hidden states as they vary with the size of input.

for each decoder step ;

$$h_i^d = g(\hat{y}_{i-1}, h_{i-1}^d, c_i)$$

Calculation of c_i :-

① → score $(h_{i-1}^d, h_j^e) = h_{i-1}^d \cdot h_j^e$ → for each encoder hidden state 'j'

$$\alpha_{ij} = \text{Softmax}(\text{score}(h_{i-1}^d, h_j^e))$$

• finally, given the distribution

$$c_i = \sum_j \alpha_{ij} h_j^e$$

We can get a more powerful scoring function by parameterizing the score with its own set of weights W_s.

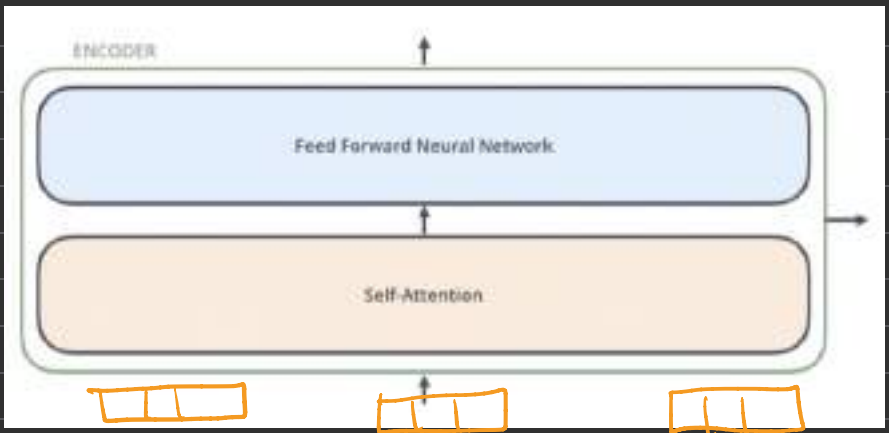
$$\text{Score}(h_{i-1}^d, h_j^e) = h_{t-1}^d W_s h_j^e$$

W_s are learnable

Issues with RNNs :-

- Hard to learn long-distance dependencies.
- Linear order of words (! Not right way to think)
- Future RNN hidden states can't be computed in full before past RNN hidden states have been computed. Lack of parallelizability !

- In Attention, all words can interact with each other and computation can be done in parallel.



- Each encoder recieves a list of vectors each of size 512.
- In the bottom encoder, this would be the word embeddings.

☆☆ The word in each position flows through it's own path in the encoder.

There are dependencies between these paths in the self attention layer but no in FFN.

Attention :-

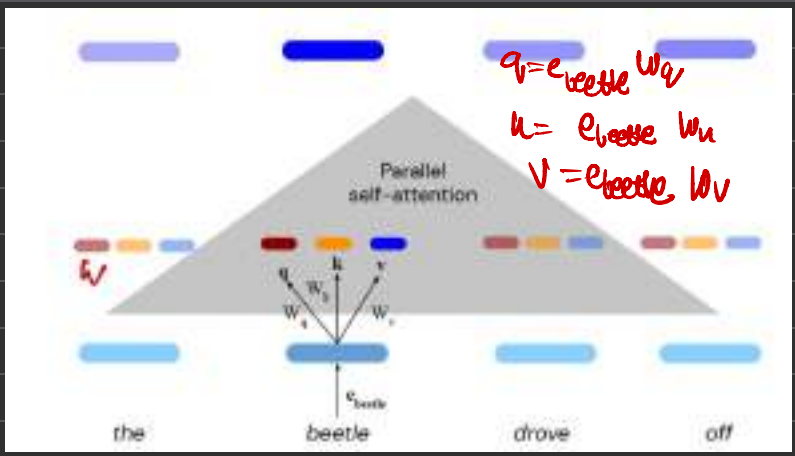
- in self-Attention the queries, keys, and values are drawn from the same source.

- the dot product self-Attention is

$$e_{ij} = q_i^T k_j \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

$$\text{output}_i = \sum_j \alpha_{ij} v_j$$

★ the number of queries can differ from the number of keys in practice.



$q = e_{beetle} W_q$

$k = e_{beetle} W_k$

$v = e_{beetle} W_v$

- Attention for all the words can be done in parallel!

Multi-Attention Heads :-

→ the different words in a sentence can relate to each other in many different ways simultaneously.

→ it could be difficult for a single self-Attention model to learn to capture all of the different kinds of parallel relations among its inputs

Concept of Multi-head Attention :-

We use distinct set of parameters, each head can learn different aspects of the relationships among inputs at the same level of abstraction.

- Each head has its own W_i^K, W_i^Q and W_i^V
- d → dimension of input and output

key → d_k query → d_k value → d_v

for each head i,

$W_i^Q \in R^{d \times d_k}$ $W_i^K \in R^{d \times d_k}$ $W_i^V \in R^{d \times d_v}$

These get multiplied by input X

outputs :- $Q \in R^{N \times d_k}$ $K \in R^{N \times d_k}$ $V \in R^{N \times d_v}$

for each head, output shape: $N \times d_v$

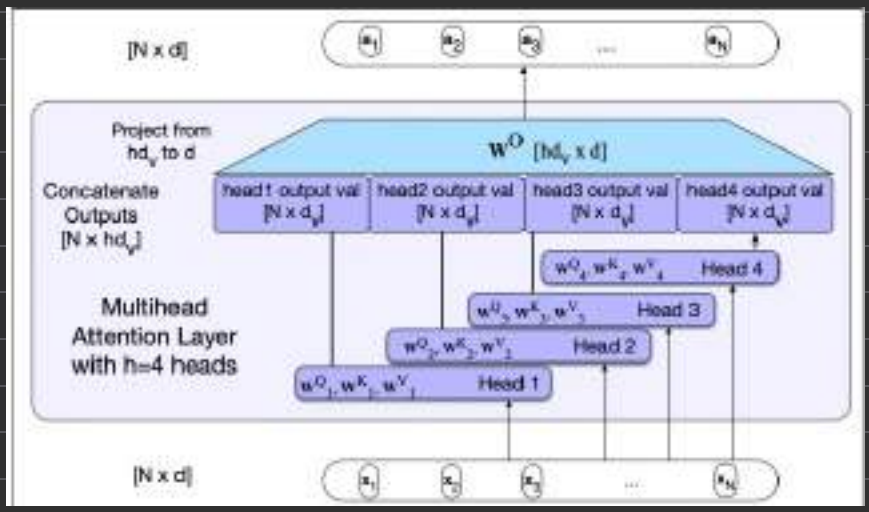
all the outputs are concatenated $\rightarrow N \times h d_v$

→ This is multiplied by $W^O \in R^{h d_v \times d}$ → to reshape the output to original dimensions

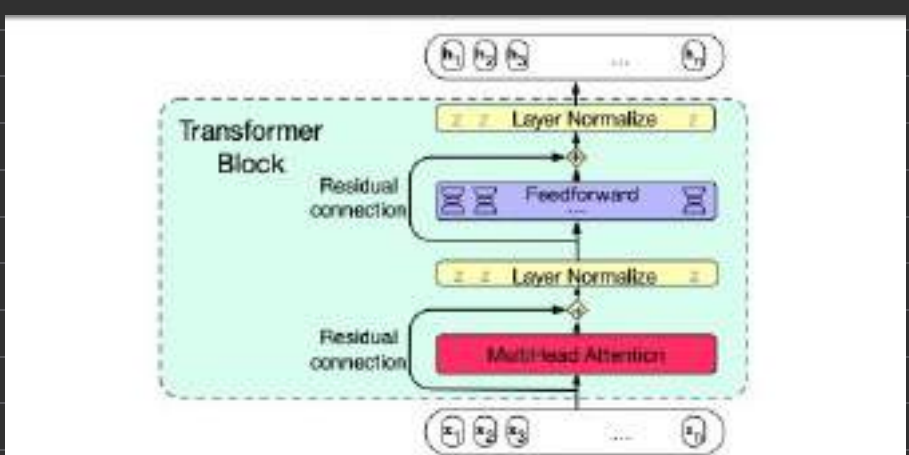
$Q = X W_i^Q$; $K = X W_i^K$; $V = X W_i^V$

$head_i = SelfAttention(Q, K, V)$

$MultiHeadAttention(X) = (head_1 \oplus head_2 \cdots \oplus head_h) W^O$



A transformer block showing all layers ↓



$$t_i^1 = \text{Multihead Attention}(x_i, x_1, \dots, x_N)$$

$$t_i^2 = t_i^1 + x_i$$

$$t_i^3 = \text{LayerNorm}(t_i^2)$$

$$t_i^4 = \text{FFN}(t_i^3)$$

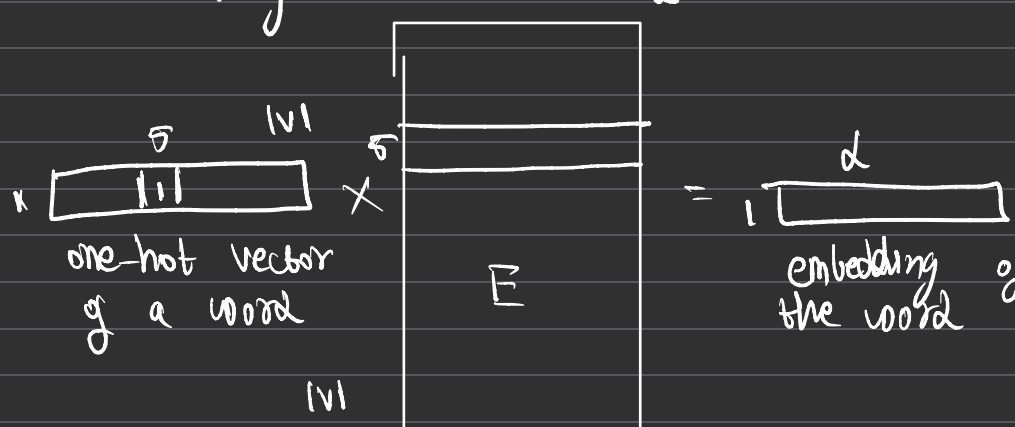
$$t_i^5 = t_i^4 + t_i^3$$

$$h_i = \text{LayerNorm}(t_i^5)$$

↳ output of transformer block

Embeddings :-

E → Embedding matrix



one-hot vector of a word

embedding of the word

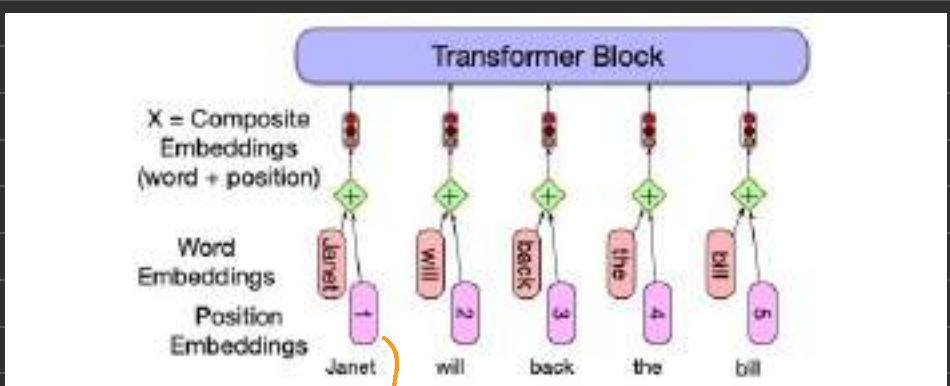
token-embeddings

These token embeddings are not position dependent. To represent the position of each token, we combine these token embedding's with positional embeddings

Simple Approach :-

- store the absolute position of the words.
- these positional embeddings are learned along with other parameters during training.

$E_{pos} \rightarrow N \times N$



↳ both are of [1x2] size

problem with the above approach :-

- there will be plenty of training examples for the initial positions in our inputs and correspondingly fewer at the outer length limits.

- this leads to poor generalization for latter embeddings.

Solution :- choose a static function that maps integer inputs to real-valued vectors that captures the inherent relationship among the positions.

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

- these position embeddings are added to word embeddings.

Vocabulary :-

→ we assume a fixed vocab of tens of thousands of words, built from the training set.

→ All novel words seen at test time are mapped to a single UNK.

unknown type

Byte-pair encoding Algorithm :-

- start with a vocabulary containing only characters and an "end of word" symbol.
- using a corpus of text, find the most common adjacent characters "a, b"; add "ab" as a subword.
- Replace instances of the character pair with the new subword; repeat until desired vocab size.

Layer Normalization :-

- Let $x = [x_1, \ldots, x_d]$ be an individual word vector in the model.

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i, \quad \mu \in R$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}, \quad \sigma \in R$$

Let $\gamma \in R^d$ and $\beta \in R^d$ can be learnable parameters.

$$\text{Layernorm} = \gamma\left(\frac{x-\mu}{\sigma+\epsilon}\right) + \beta$$

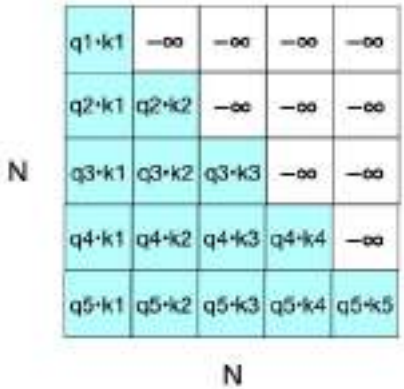
not important

Masked-Attention :-

The calculation in QKT results in a score for each query value to every key value including those that follow the query.

this is inappropriate in the setting of language modelling : guessing the next word is simple if you already knew it.

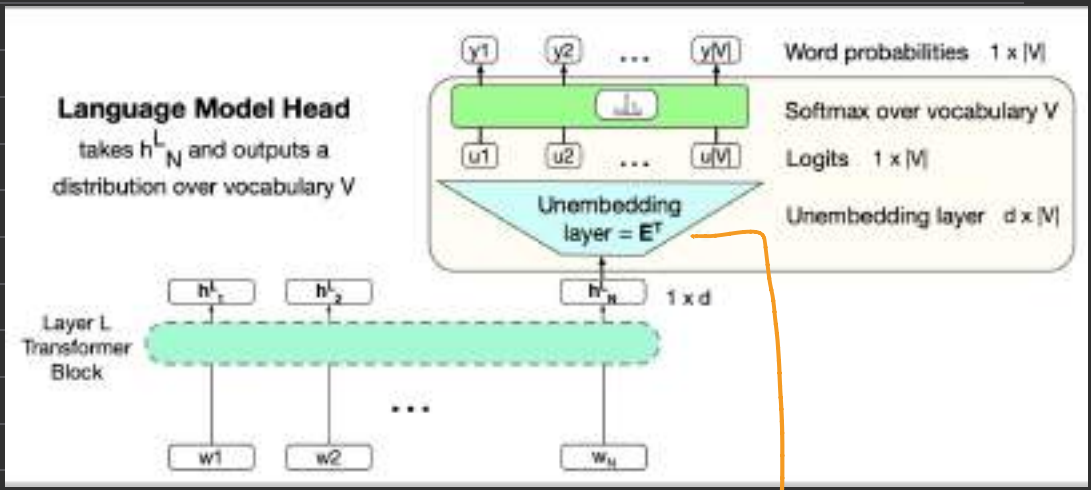
Solution



Encoder - Decoder Attention :-

- Let $h_1, \dots, h_n$ be output vectors from the transformer encoder ; $h_i \in \mathbb{R}^d$
- Let $z_1, \dots, z_n$ be input vectors from the transformer decoder ; $z_i \in \mathbb{R}^d$
- keys and values $\Rightarrow$ $k_i = K h_i$, $v_i = V h_i$ $\Rightarrow$ from encoder
- queries are drawn from decoder $\Rightarrow$ $q_i = Q z_i$

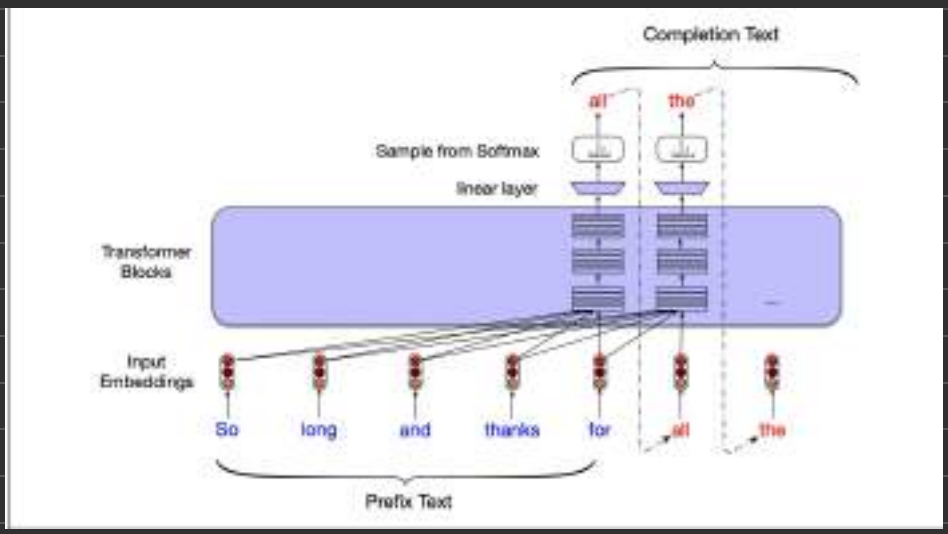
Transformers for Language Modeling :-



→ Language modelling head

This linear layer can be learned but more commonly we tie this matrix to the transpose of the embedding matrix. (E^T)

- In the learning process E will be optimized to be good at doing both of these mappings.

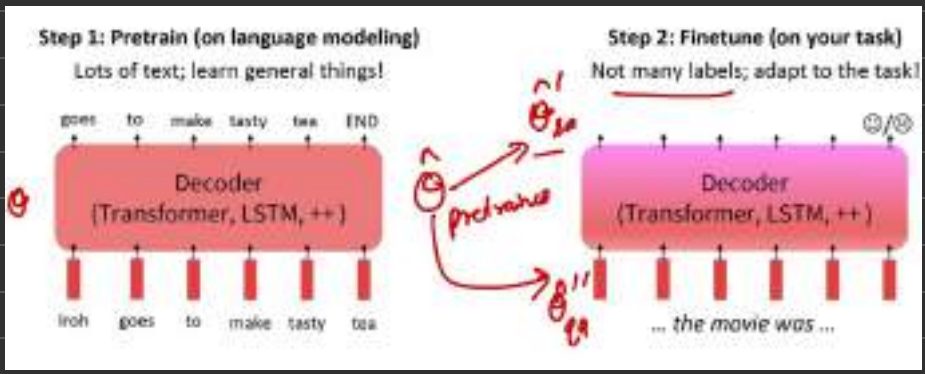


- As the generation proceeds, model has access to the context as well as it's previously generated tokens.

Pre-Training :-

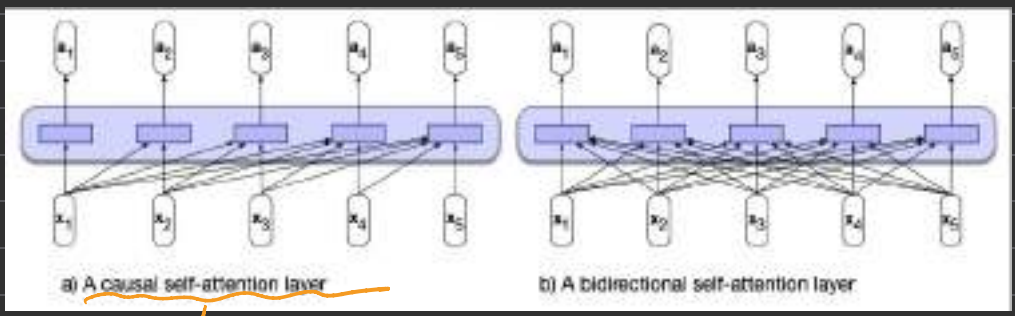
- pretraining methods hide parts of the input from the model and train the model to reconstruct these parts.
- This worked exceptionally well for :-
  - → representations of language
  - → parameter initializations for strong nlp models

→ probability distributions over language that we can sample from.



- pretraining loss ⇒ provides approximating parameters $\hat{\theta}$ by $\min_{\theta} L_{pretrain}(\theta)$
- now, finetuning loss ⇒ approximate $\min_{\theta} L_{finetune}(\theta)$, starting at $\hat{\theta}$
- The pretraining may matter because stochastic gradient descent stays relatively close to $\hat{\theta}$ during finetuning.
  - maybe the finetuning local minima near $\hat{\theta}$ tend to generalize well!
  - maybe the gradients of finetuning loss near $\hat{\theta}$ propagate nicely.

Bi- Directional Transformer Encoders :-



These models will not be able to solve problems like self-Attention and labelling which involves processing text from right to left also.

- Bi-directional encoders use self-Attention to map sequences of input embeddings $(x_1, \ldots, x_n)$ to sequences of output embeddings.

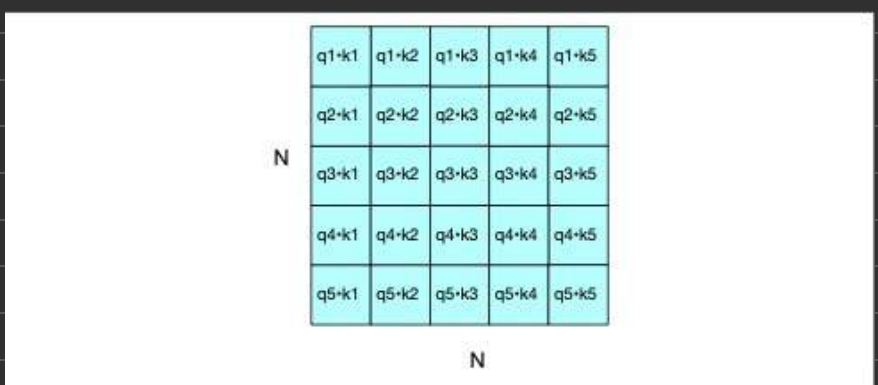
These output embeddings are contextualized representations of each input token that are generally useful across a range of downstream tasks.

The architecture for Bi-directional models :-

Bidirectional models use the same self-Attention mechanism as casual models

$$Q = XW^Q, \quad K = XW^K; \quad V = XW^V$$

$$X \in R^{N \times d_h}$$



QK^T matrix

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

The key architecture difference is in bidirectional

models, we don't mask the future. we simply mask the each token, allowing model to contextualize each token using information from the future.

- Hidden layers of size 768
- 12 layers of transformers blocks, with 12 multihead attention layers each.
- the resulting model has about 100M parameters.
- Bert and all it's descendants are based on subwords tokens rather than words.

Masking words :-

- the original approach to training bidirectional encoders is called masked language modelling.
- The model is presented with a series of sentences from the training corpus where a random sample of tokens from each training sequence is selected for use in the learning task.

Once chosen, a token is used as

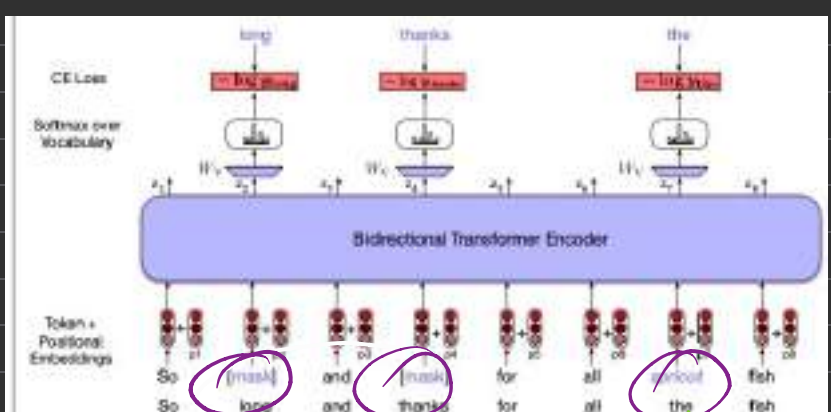
① it is replaced with unique vocabulary token [MASK]

② it is replaced with another token from the vocabulary, randomly sampled on the token unigram probabilities.

③ it is left unchanged.

In BERT, 15% of the input tokens are sampled for learning

of these, 80% replaced with [mask], 10% are replaced with randomly selected tokens, and the remaining 10% are left unchanged.



replaced with mask

replaced with other word

- All the inputs tokens play a role in the self Attention process, but only the sampled tokens are used for learning

$$y_i = \text{softmax}(W_v z_i)$$

$$L_{mLM} = \frac{-1}{|m|} \sum_{i \in m} \log p(x_i | z_i)$$

Next sentence prediction :-

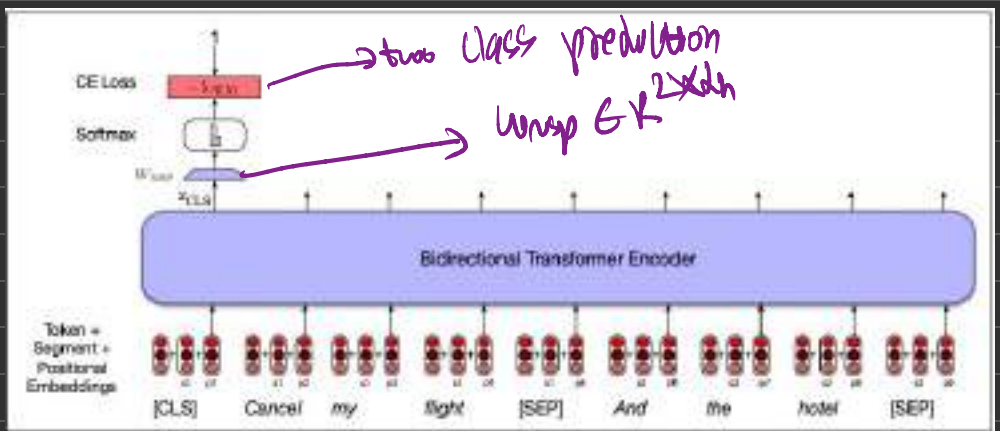
In this task, the model is presented with pairs of sentences and is asked to predict whether each pair consists of an actual pair of adjacent sentences from the training corpus or a pair of unrelated sentences

- To facilitate NSP training, BERT introduces two new tokens to input representation.

→ The token [CLS] is prepended to the input sequence

→ the token [sep] is placed between the sentences and after the final token of the second

Sentence.



Cross entropy is used to learn NSP loss

why NSP?

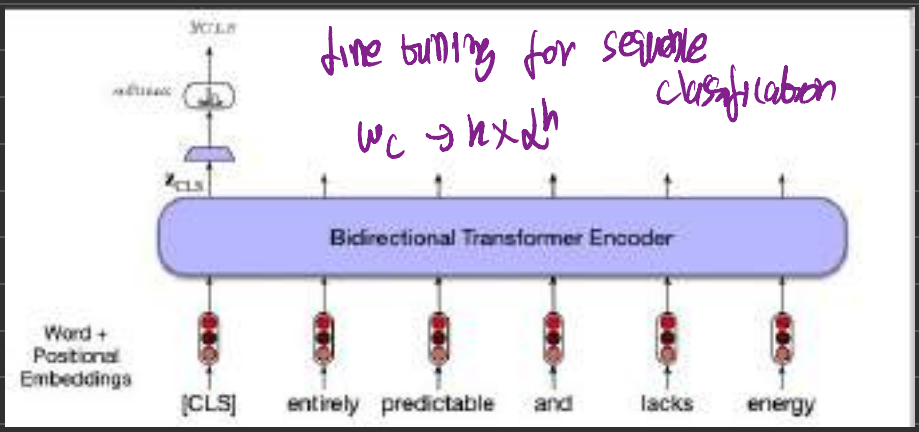
- Detecting if two sentences have similar meaning
- Detecting if the meanings of two sentences entail or contradict each other
- deciding if two sentences form a coherent discourse.

Fine-tuning :-

- Fine-tuning facilitates the creation of applications on top of pretrained models through the addition of small-set of application specific parameters.

- The fine-tuning process consists of using labeled data from the application to train those additional application-specific parameters.

- Typically this training will either freeze or make only minimal adjustments to the pretrained language model parameters.



Sequence labeling :-

Parts of speech tagging

(or)

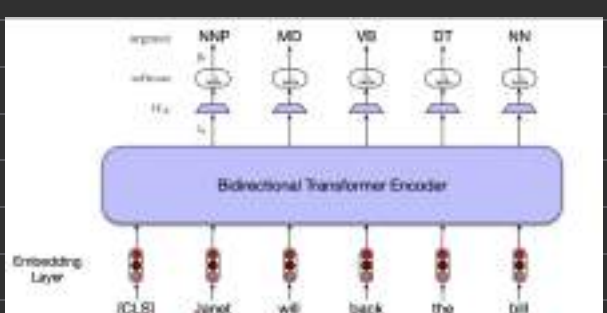
BIO - based named entity recognition

B → indicates the first token of a multi-token entity

I → indicates a token that is inside a multi-token entity

O → indicates a token that is not part of any entity.

Allows us to understand the types of entities in the text



$y_i = \text{Softmax}(W_u z_i)$

$t_i = \text{argmax}_k(y_{ik})$

Alternative :-

The distribution over labels provided by the softmax for each input token can be passed to a conditional random field (CRF) layer which can take global tag-level transitions into account.

→ they capture dependencies between input features and output labels in sequence data.

[Loc Mt. Sanitas] is in [Loc Sunshine Canyon]

The set of per-word BIO tags.

Mt.	Sanitas	is	in	Sunshine	Canyon.
B-Loc	I-Loc	O	O	B-Loc	I-Loc → O

The word piece tokenization for this sentence yields the following sequence of tokens

"Mt", ".", "San", "##itas", "is", "in", "sunshine", "canyon", "."

To deal with misalignment, we need a way to assign BIO tags to subword tokens

during training and a corresponding way to recover word-level tags from subwords during decoding.

→ for training, we can just assign the gold-standard tag associated with each word to all of the subword tokens derived from it.

→ for decoding, the simplest approach is to use argmax BIO tag associated with the first subword token of a word.

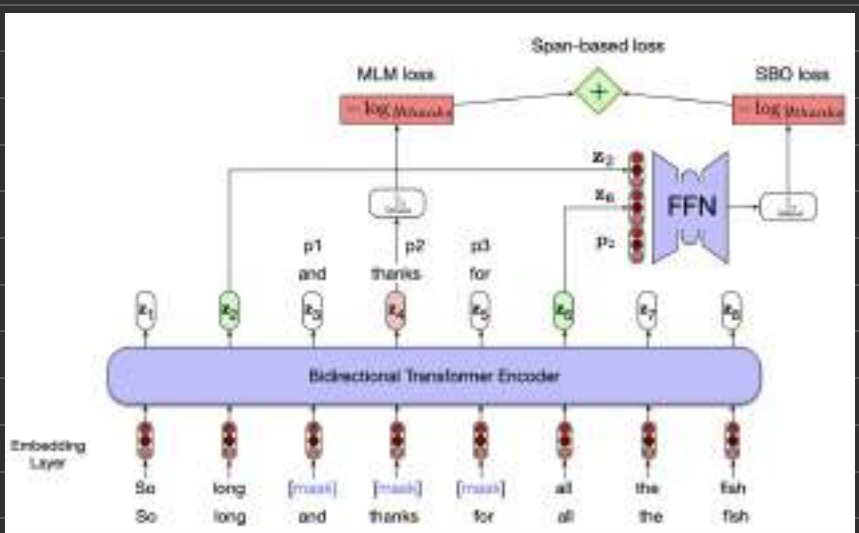
Fine-tuning for span based Applications :-

- once a span is choosen for masking, all the tokens within the span are substituted according to the same regime used in BERT:-
  - → 80% are replaced with [MASK]
  - → 10% are replaced by randomly sampled tokens
  - → 10% are kept as it is
- the spanBERT learning objective augments the MLM objective with a boundary oriented component called the span boundary objective (SBO).

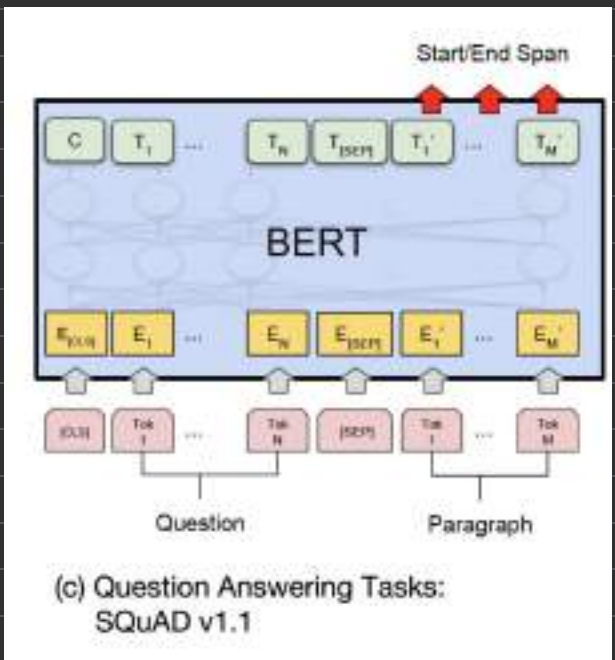
- $x_1, x_2, \ldots, x_n \rightarrow$ input
  $z_1, z_2, \ldots, z_n \rightarrow$ output

→ A token x_i in a masked span of tokens $(x_s, \ldots, x_e)$.

$$L_{SBO}(x_i) = -\log p(x_i / x_{s-1}, x_{e+1}, p_{i-s+1})$$



Fine-tuning for SQuAD Dataset :-



(c) Question Answering Tasks: SQuAD v1.1

- We only introduce a start vector $S \in \mathbb{R}^H$ and an end vector $E \in \mathbb{R}^H$ during fine tuning

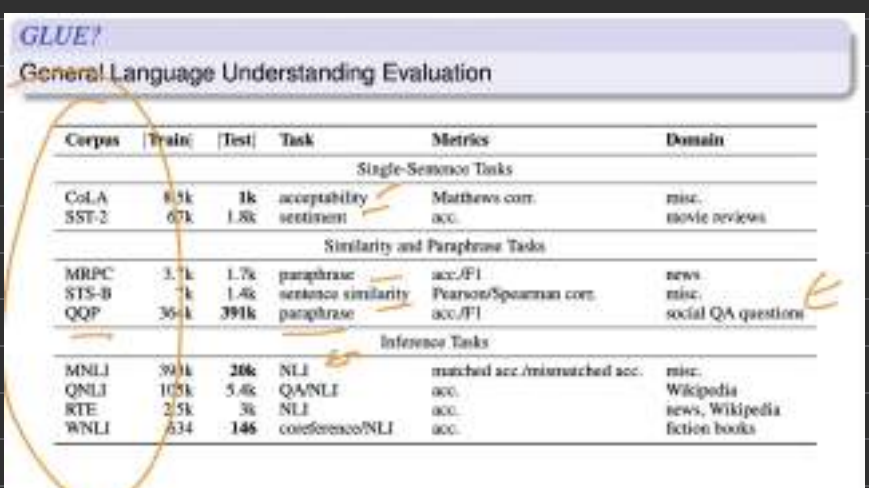
→ The probability of word i being the start of answer span is computed as dot product between T_i and S followed by Softmax over all of the words in the paragraph.

$$P_i = \frac{e^{S \cdot T_i}}{\sum_j e^{S \cdot T_j}}$$

the score of a candidate span from position i to position j is defined as $S \cdot T_i + E \cdot T_j$ and the maximum scoring span where $j \geq i$ is used as prediction.

* for practical purposes, span-based models often impose an application-specific length limit L so the legal spans are limited to those where $j - i < L$.

GLUE Benchmark :-



GLUE?

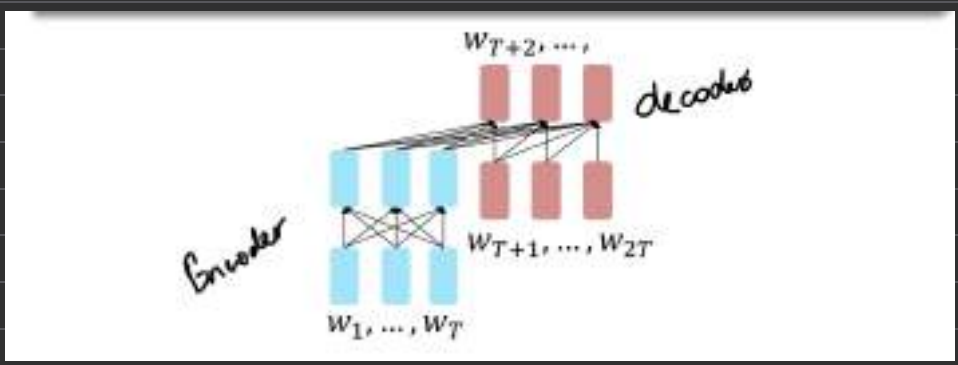
General Language Understanding Evaluation

Corpus	|Train|	|Test|	Task	Metrics	Domain
			Single-Sentence Tasks		
CoLA	8.5k	1k	acceptability	Matthews corr.	misc.
SST-2	67k	1.8k	sentiment	acc.	movie reviews
			Similarity and Paraphrase Tasks		
MRPC	3.7k	1.7k	paraphrase	acc./F1	news
STS-B	7k	1.4k	sentence similarity	Pearson/Spearman corr.	misc.
QQP	364k	391k	paraphrase	acc./F1	social QA questions
			Inference Tasks		
MNLI	393k	20k	NLI	matched acc./mismatched acc.	misc.
QNLI	105k	5.4k	QA/NLI	acc.	Wikipedia
RTE	2.5k	3k	NLI	acc.	news, Wikipedia
WNLI	634	146	coreference/NLI	acc.	fiction books

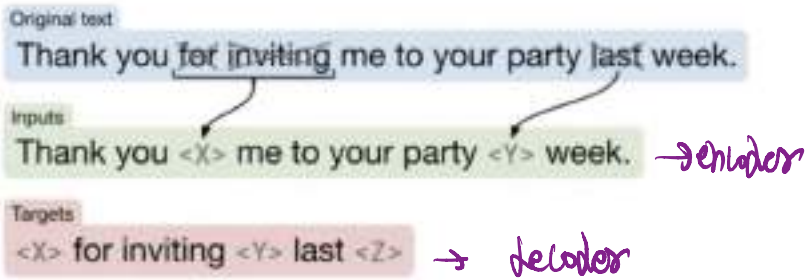
A key takeaway from Roberta paper :-

More compute, more data can improve pretrain even when not changing the underlying Transformer encoder.

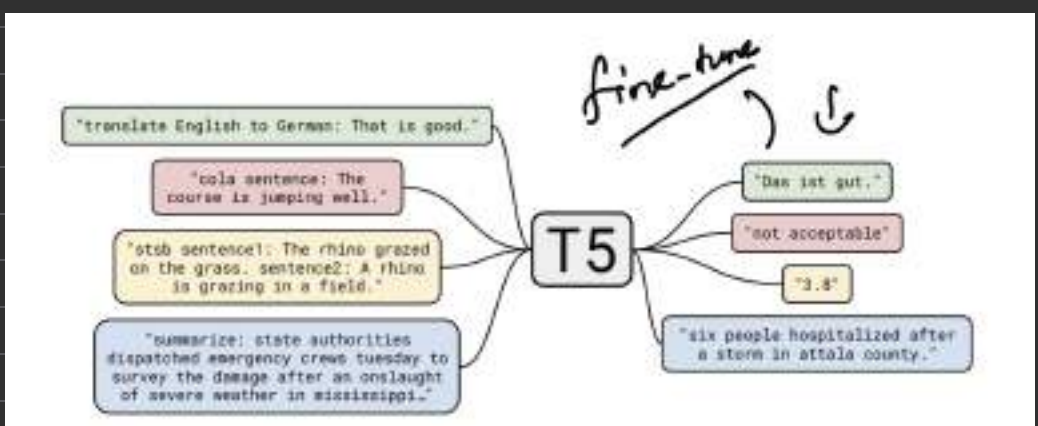
Pretraining encoder-Decoder models :-



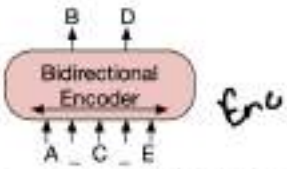
The encoder portion benefits from bi-directional context The decoder portion is used to train the whole model through language modeling.



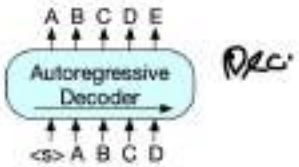
The same model loss function hyperparameters etc can be used across all tasks



BART



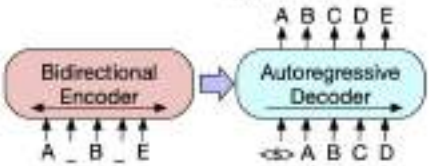
Enc



Dec.

(a) BERT: Random tokens are replaced with masks, and the document is encoded bidirectionally. Missing tokens are predicted independently, so BERT cannot easily be used for generation.

(b) GPT: Tokens are predicted auto-regressively, meaning GPT can be used for generation. However words can only condition on leftward context, so it cannot learn bidirectional interactions.



Enc. - dec.

(c) BART: Inputs to the encoder need not be aligned with decoder outputs, allowing arbitary noise transformations. Here, a document has been corrupted by replacing spans of text with mask symbols. The corrupted document (left) is encoded with a bidirectional model, and then the likelihood of the original document (right) is calculated with an autoregressive decoder. For fine-tuning, an uncorrupted document is input to both the encoder and decoder, and we use representations from the final hidden state of the decoder.

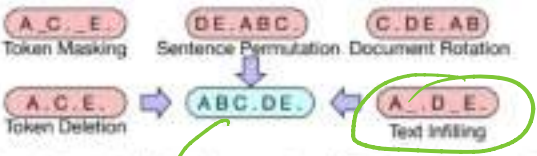


Figure 2: Transformations for noising the input that we experiment with. These transformations can be composed.

A number of text spans are sampled

A token is chosen and then noised.

How to encode an image as a sequence :-

- original image shape :- $H \times W \times C$
- N patches of dimensions $P \times P$: $N = HW / P*P$
- Each patch is mapped to D dimensions through a trainable layer $(P^2 \cdot C) \times D$
- standard learnable 1-D positional embeddings are used E_{pos} & $R^{(N+1) \times D}$
- A learnable class token is prepended and its representation at last layer is used as image representation.

Decoding :-

→ The task of choosing a word to generate based on model's probability is called decoding.

→ methods that give bit more weight to the middle probability words tend to be more creative and diverse but likely to be incoherent and less factual

Greedy decoding :-

at each time step t in generation, the output y_t is choosen by computing the probability for each word in vocabulary.

$$\hat{w}_t = \underset{w \in V}{\operatorname{argmax}} P(w / w_{<t})$$

Beam Search :-

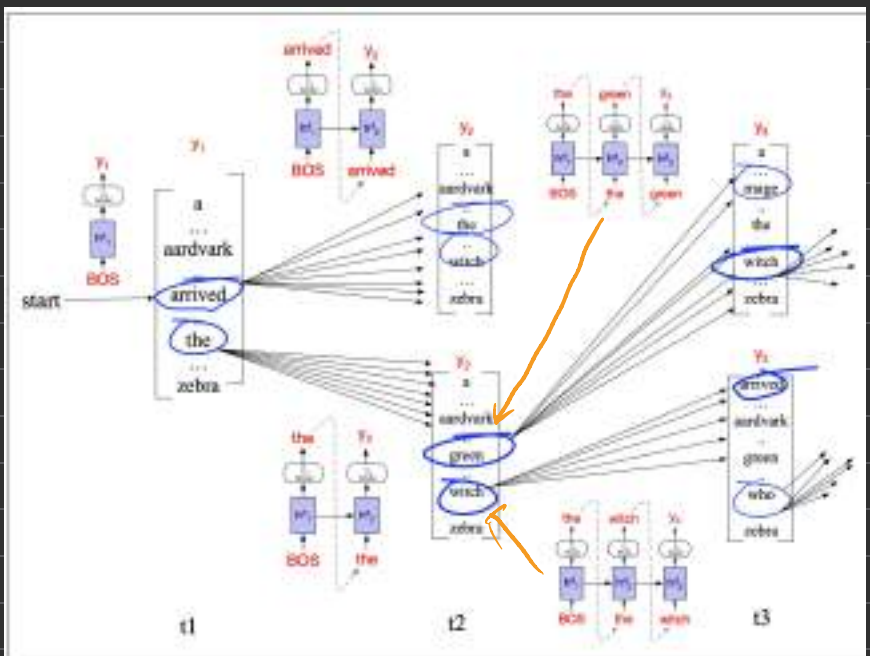
In beam search instead of choosing the best token to generate at each timestep, we keep k possible tokens at each step.

↓

beam width

- we select the k-best options from this softmax output.
- these k words are called hypotheses.

Schematic view of beam search :-



- At each time step, nxV hypotheses are scored by $P(y_i / x, y_{<i})$.
- select the best n hypotheses again.

never more than n decoders or n hypothesis at frontier of search

- once Eos is generated, we have found the candidate outputs.

$$\text{score}(y) = \log p(y/x)$$

$$\log (p(y_1/x)\, p(y_2/y_1, x) \dots \dots)$$

$$= \sum_{i=1}^{t} \log p(y_i/y_1, \dots, y_{i-1}, x)$$

Top-n sampling :-

- Choose in advance a number of words n
- For each word in the vocabulary V use the language model to compute the likelihood of this word given the context $p(w_t / w_{<t})$
- Sort the words by their likelihood and throw any word that is not one of the top n most probable words.
- Renormalise the score of n words to be a legitimate probability distribution
- Randomly sample a word from within these remaining n most-probable words according to its probability.

- keeping k-fixed is a big disadvantage, as the probability distribution density may change with different contexts.

Top-p sampling :-

Given a distribution $P(w_t | w_{<t})$ the top-p vocabulary $V^{(p)}$ is the smallest set of words such that

$$\sum_{w \in V^{(p)}} P(w | w_{<t}) \geq p$$

Temperature sampling :-

- we don't truncate the distribution, but reshape it

Thermodynamics intution :- when a system at high temperature at a high temperature it is flexible and explore many possible states

when a system at low temperature it is likely to explore a subset of lower energy states

modified softmax computation

$y = \text{softmax}(u/\tau)$

$\tau \rightarrow$ temperature

$\tau \rightarrow 1$ doesn't make any change

$\tau \leq 1 \Rightarrow$ inputs become larger $\Rightarrow$ probability distribution become more greedy

softmax tends to push high values to 1 and low values to 0

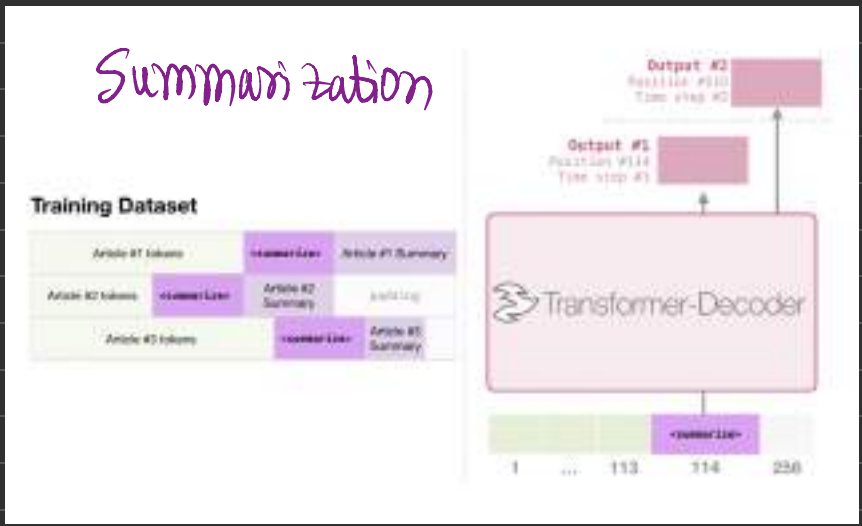
Scaling Laws :-

- The performance of a large language model is hugely affected by
  - model size
  - Training data size
  - Amount of compute used for training

formating the inputs for training on various tasks:

Machine Translation

Summarization



Rough estimation of parameters of a model :-

$$N \approx 2d\, n_{layer}\,(2d_{attn} + d_{ff})$$

$$N \approx 12\, n_{layer}\, d^2$$

$$d_{attn} = \frac{d_{ff}}{4} = d$$

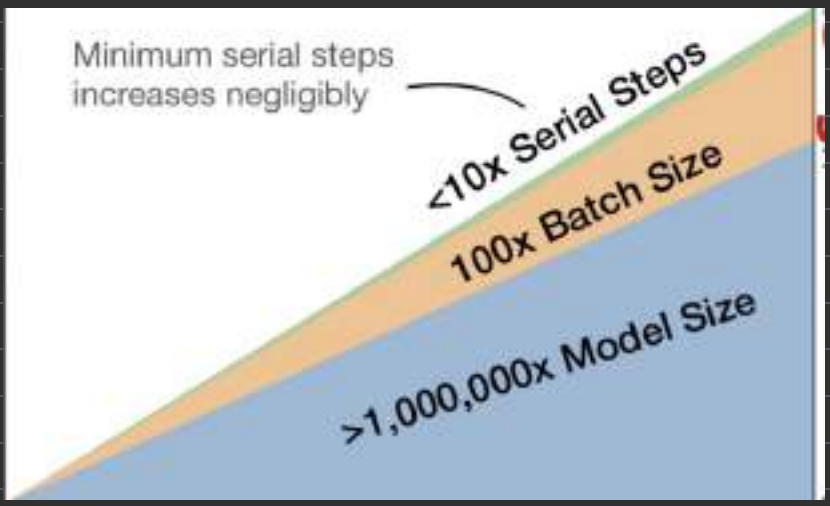
In-Context learning

- very large language models seem to perform some kind of learning without gradient steps simply from examples you provide within their contexts.

Prompting over finetuning

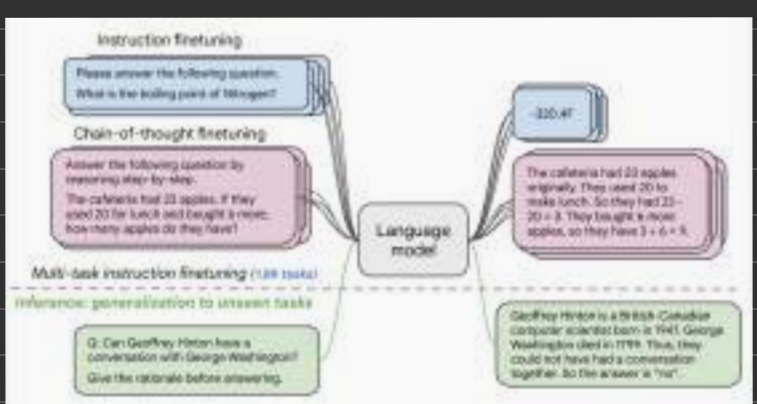
$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N} \qquad L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D} \qquad L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C}$$

Scaling Laws !



as compute increase

Instruction fine-tuning :-



Collect many pairs of (instruction / output) across many tasks and finetune an LLM.

Major Limitations :-

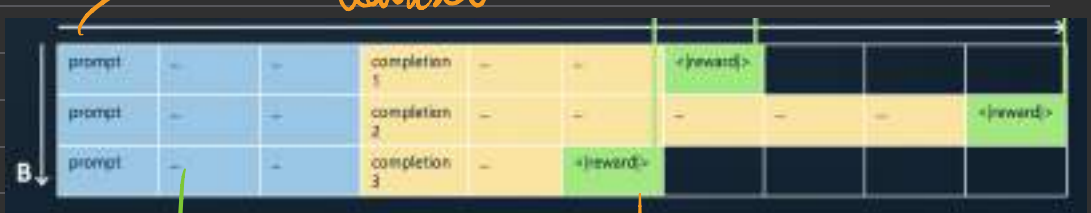
- expensive to collect data
- No standard answer for open-ended creative generation
- Language modelling penalizes all token-level mistakes equally, but some errors are worse than others.

LM Assistants :-

The base models cannot be used as LM assistants.

Reward Modelling Training :-

RM dataset



same prompt tokens

only this output is used

Goal :- predict rewards consistent with preferences

RL Training :- (Maximise the reward)

- The model is now trained on only yellow cells
- It predicts only the <reward>
- The sample tokens becomes labels the training objective is weighted by the "advantage". (normalized rewards)

optimize this using human preferences

for each LM generated sample s, imagine we had a way to obtain human reward of that summary. $R(s)$

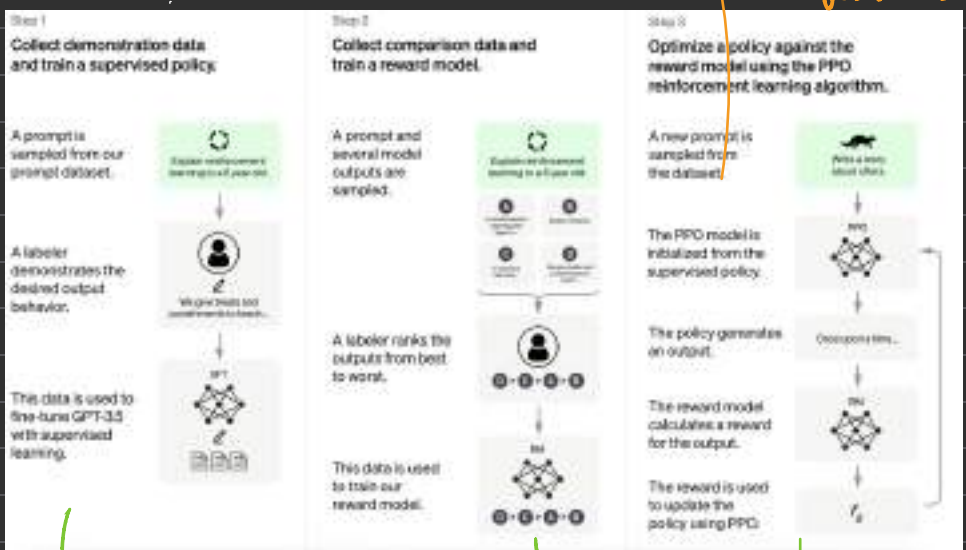
Goals :-

Maximize $E_{\hat{s} \sim P_\theta(s)}[R(\hat{s})]$

But how do we do this?

RLHF pipeline :- [Finetuning the model on human preferences]

Step 1 :-



policy to update the model

instruction tuning

Reward modelling

RL policy for maximizing the reward.

Reinforce Algorithm :- (ppo)

Goal : $E_{\hat{s} \sim P_\theta(s)}[R(\hat{s})]$ → maximize this

This may be a non-convex problem

So, better to use gradient ascent

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta_t} E_{\hat{s} \sim P_{\theta_t}(s)}[R(\hat{s})]$$

Reward function may be non-differentiable

$$\rightarrow \nabla_\theta E_{\hat{s} \sim P_\theta(s)}[R(\hat{s})] = \sum_s R(s) \nabla_\theta P_\theta(s)$$

$$P_\theta(s) \nabla_\theta \log P_\theta(s)$$

$$\sum_s P_\theta(s) R(s) \nabla_\theta \log P_\theta(s)$$

$$= E_{\hat{s} \sim P_\theta(s)}[R(\hat{s}) \nabla_\theta \log P_\theta(\hat{s})]$$

↓

This can be averaged over all the prompts

$$\frac{1}{m}\sum_{i=1}^{m} R(s_i)\,\nabla_\theta \log p_\theta(s_i)$$

↓

if reward is +ve, Take gradient steps to maximize $p_\theta(s_i)$

if reward is −ve Take gradient steps to minimize $p_\theta(s_i)$

Major issue :- HUMAN in the process

Model the Human preferences as a seperate NLP problem

LM → RM $p(s)$ → for optimising the reward

Problem with Human annotations :- Human annotations are highly noisy.

- provide annotations as pairwise comparisons / judgements so that a relative can be made.

Training objective

$$J_{RM}(\phi) = -\mathbb{E}_{(s^w, s^l)\sim D}\left[\log \sigma(RM_\phi(s^w) - RM_\phi(s^l))\right]$$

$RM_\phi(s^w)$ → winning sample (should score higher)

$RM_\phi(s^l)$ → losing sample

R_{LHF}

$p^{PT}(s)$ → pretrained LM

$RM_\phi(s)$ → reward model (trained)

$p_\theta^{R_L}(s)$ → copy of $p^{PT}(s)$ for PPO.

Use of KL Divergence here !

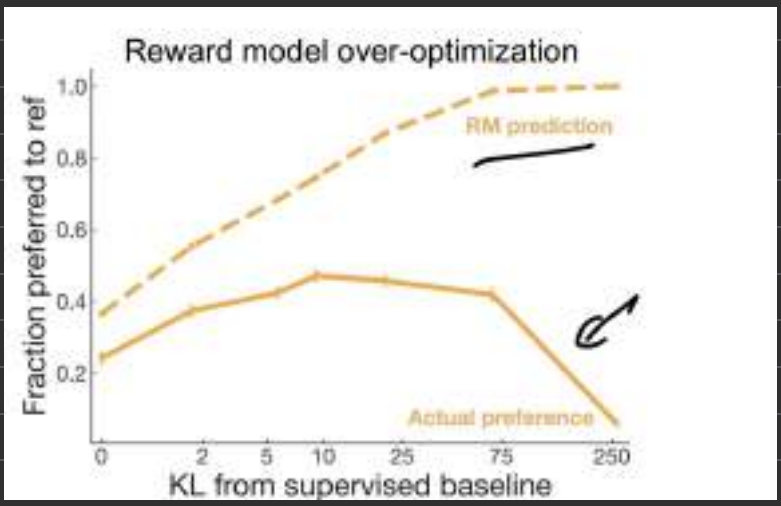
making sure that the model doesn't deviate by a large amount from its original weights.

To implement this, a reward penalty is used

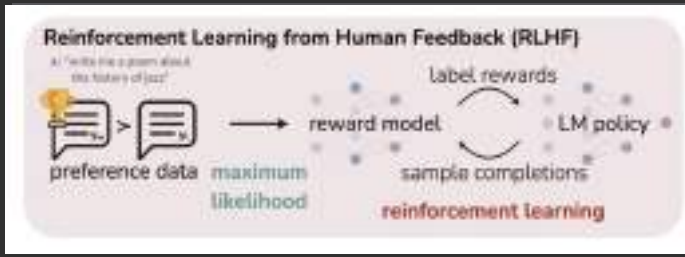
$$R(s) = RM_\phi(s) - \beta \log\left(\frac{P_\theta^{RL}(s)}{P_\theta^{PT}(s)}\right)$$

$$P_\theta^{RL}(s) > P_\theta^{PT}(s)$$

Penalty



Direct preference optimization :-



Dpo directly optimises for the policy best satisfying the preferences with a classification objective.

$$L_{DPO} = -E_{(x, y_w, y_l) \sim D}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)}\right)\right]$$

$(\pi_\theta; \pi_{ref})$

→ Dpo model

Parameter efficient fine tuning techniques :-

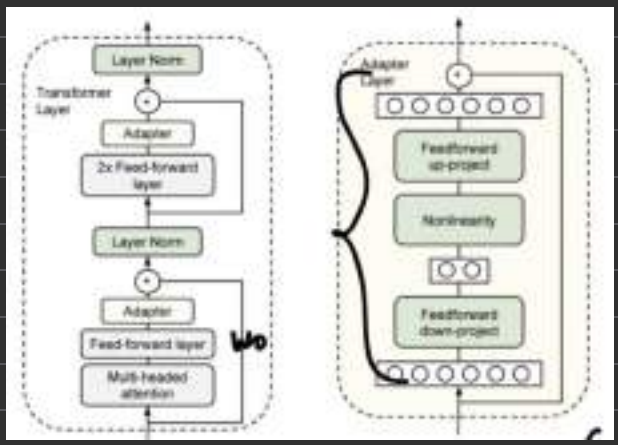
Adapters :-

Fine tuning onto a specific task can be composed as

$$f_i'(x) = f_{\theta_i}(x) \odot f_{\phi_i}(x)$$

The original parameters of the model are not changed.

- Insert new function f_ϕ between layers of pre trained model to adapt to a downstream task.



$W_D \in R^{k \times d}$

$W_U \in R^{d \times k}$

$$f_\phi(x) = W^U(\sigma(W^D x))$$

For one adapter :-

Feed-Forward down projection : $r \times d_{model} + r$

Feed-forward Upward projection : $d_{model} \times r + d_{model}$

→ for L layers in the decoder, there would be $2L$ adapters

→ number of parameters → $2L \times (2 \times d_{model} \times r + d_{model} + r)$

Pruning for pre-trained models :-

- A fraction of the lowest weights are removed.
- The non-pruned weights are re-trained.

Magnitude pruning : keep weights farthest from 0

Movement pruning : keep weights move the most away from 0.

Major objective of the tuning :-

Maximise the conditional language modelling objective :-

$$\max_{\phi} \sum_{(x,y)} \sum_{t=1}^{|y|} \log\left(P_{\phi}(y_t | x, y_{<t})\right)$$

LoRA :-

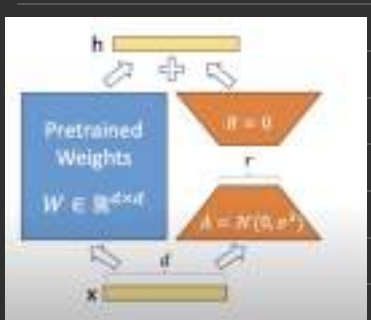
$$\Delta\phi = \Delta\phi(\Theta)$$

$\Theta \rightarrow$ task specific parameters

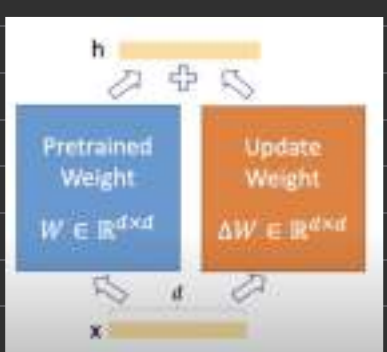
$$|\Theta| << |\phi_0|$$

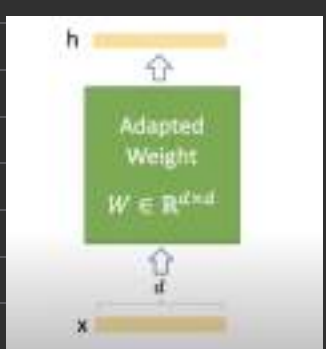
The task of finding $|\phi|$ becomes as

$$\max_{\Theta} \sum_{(x,y)} \sum_{t=1}^{|y|} \log\left(P_{\phi_0 + \Delta\phi(\Theta)}(y_t | x, y_{<t})\right)$$



Fine Tuning





inference time

$W_0 \in R^{d \times k}$ $\quad B \in R^{d \times r}$, $A \in R^{r \times k}$

only A and B contain trainable parameters

weight decomposition :- $W = W_0 + \Delta W$, $\Delta W = BA$

W_0 → frozen

Forward pass :- $W_0 x + \Delta W x$

$= W_0 x + BAx$

During inference time, $W = W_0 + BA$, no additional latency.

- If LoRA underperforms, increase the value of rank.

Quantization of a pretrained model :-

Ex:- quantizing a FP32 number to Int8 integer $\in [-127, 127]$

$$x^{int8} = round\left(\frac{127}{\underbrace{absmax(x^{FP32})}_{c}} \cdot x^{FP32} \right)$$

Example :- (2-bit)

Maintain a map of values rather than absolute bits to minimize information loss.

map : {Index : 0, 1, 2, 3 → {values :- 1, 0.3, 0.5, 1.0}

input tensor :- {10, −3, 5, 4}

Normalise with abs max : [10, −3, 5, 4]

↓

[1, −0.3, 0.5, 0.4]

Nearest values :- [1, 0.3, 0.5, 0.5]

Associated index :- [3, 1, 2, 2] → Store

↓ Dequantisation

[10, 3, 5, 5] → error

↳ Problem :- if an outlier appears in the input then some of the bins would not utilized well with few or no numbers in those bins

Block-wise K-bit Quantization

- Chunk into contiguous blocks that are independently quantized each with their quantization constant / c.
- NF4 Data type ensures every bin has equal number of values assigned from the input tensor.