*) Instruction Tuning
→ Train models to follow natural language instructions
→ Data - Several thousand (Task, Instruction, Output) triplets

*) Reinforcement Learning from Human Feedback
→ Show the output(s) generated by models to humans/reward model.
→ Collect feedback in the form of preferences
→ Use these preferences to further improve the model
→ Data - Several thousand (Task, Instruction) pairs and a reward model/preference model/human.
→ Needed because even after 'Instruction Tuning' the output may not be desirable for our case (eg: abusive content, undesirable output structure, etc.)

Why do we need Pre-Training?

*) Humans excel at generalisation from limited examples
- Eg: C-A-T → T-A-C D-O-G → ??
- We can answer such questions very well as we have a lifetime of accumulated knowledge to leverage from.

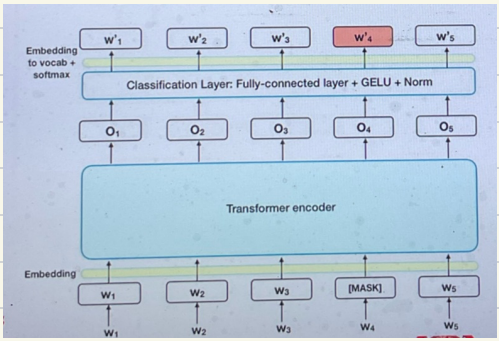
*) But this idea was not very popular in models especially before GPT-2 era.

*) Traditional ML requires vast amount of data for each task
- Need to learn each pattern from scratch
- No prior knowledge to guide them

Q) Can we somehow use the vast amount of data on the web to achieve better performance with less data? (i.e., can we generalise well enough to be able to learn from few examples via instructions)

A generative model for sentences

How to pretrain?



BERT

Uses Masked Language Modeling (MLM) for pre-training.

Not a generative model!!

BERT can be used for learning sentence embeddings but not for generating sentences because there is no straightforward way to assign probabilities to a sentence using BERT.

(Basically, since BERT is bidirectional i.e., it looks at tokens from both sides during pretraining so it is not well suited to generative tasks since autoregressive generative models work by generating token based only on the previous words)

Generative Modelling of sentences

* What we need - A model from which we can sample sentences.
* Given a sentence $S = (t_1 \cdots t_m)$ the probability of a sentence can be written as:

$$P(S) = p_1(t_1) \prod_{j=1}^{m} p_{j+1}(t_{j+1} | t_1 \cdots t_j)$$

this is always true & not dependent on any model assumption

* p_j are distributions over the vocabulary
* If the p_j's are provided, the model is called an autoregressive model. (Our task in language modeling is to parameterise the distribution in this way. Getting this distribution is not easy but we can define a distribution that computes the probabilities in this way i.e., one token after another and

depending only on the previous tokens)

Sampling a sentence from an autoregressive model

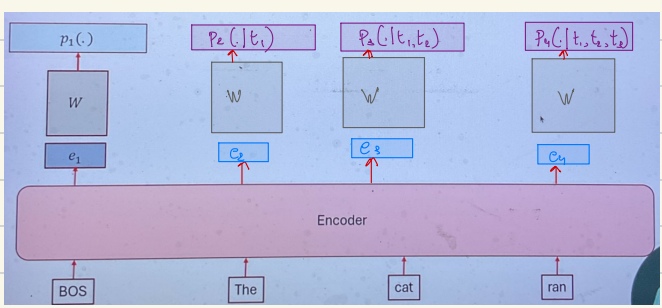
*) Sample the 1st token t_1 from p_1
*) for each subsequent step,
- Sample token t_{j+1} conditioned on the previous tokens $t_1 \dots t_j$

$$t_{j+1} \sim p_{j+1}(. \mid t_1 \dots t_j)$$

Q) Why do we want an autoregressive model? Why not any arbitrary model?
A) Since p_j's are distributions over the vocabulary, they are easy to sample from.

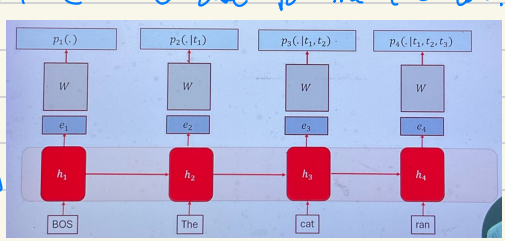
What do p_j's look like?

*) Think of an encoder which takes tokens, computes a transformation & gives the output embedding.
*) Then it applies a projection layer which takes the embedding & gives a probability distribution over the vocabulary.



*) p_1 doesn't depend on anything (takes in <BOS> token.
p_2 depends only on p_1 & so on.

Q) Given this kind of parameterisation, what can we use for the encoder?
A) One possible option is - RNN.
A unidirectional RNN inherently has this property - i.e., the next hidden state is a function of all the previous hidden states.



Problems with RNN encoder

*) Hidden state at $(j+1)^{th}$ step depends on the hidden state at j^{th} step

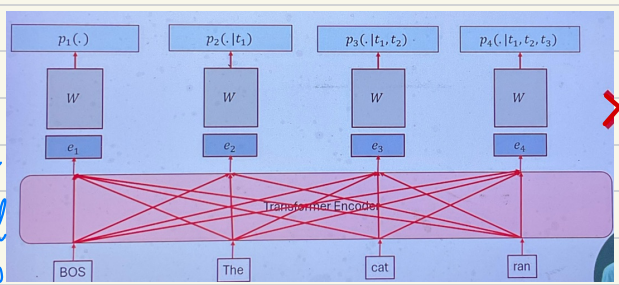
$$h_{j+1} = f(h_j, t_{j+1})$$

Time $\propto$ No of tokens → Can work for small sequences but not for bigger ones like more than 2000 tokens

How to get rid of sequential computation?

Full Attention Mask

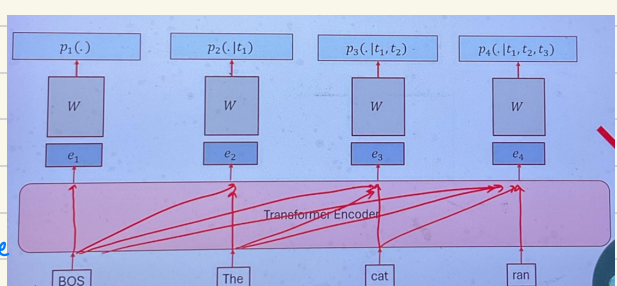
*) Another option could be a full attention mask, but in here, every token can attend to every other token. So, we cannot use this for our autoregressive model as in autoregressive models, a token probability is only dependent on previous tokens.



Causal Attention Mask

*) In causal attention mask, the j^{th} token can only look at the previous tokens. Any tokens after it are masked. So, this satisfies our criteria for autoregressive model. This kind of encoder is also called a decoder.

The attention mask looks like →



Training Loss – Recap: Maximum Likelihood Estimation

*) Given
- A parametric family of probability distributions P_θ
- n independent & identically distributed observations $s_1 \dots s_n$ (sequences) ← from some unknown distribution P_d ←

*) Task
- find the parameter θ^* that most likely resulted in the observed data, that is

$$\arg\max_\theta P_\theta(s_1 \dots s_n) \quad \rightarrow \text{Joint probability distribution}$$

$$= \arg\max_\theta \log P_\theta(s_1 \dots s_n)$$

$$= \arg\max_\theta \sum_{i=1}^{n} \log P_\theta(s_i)$$

MLE from sentence data

*) Given a sentence $s = (t_1 \dots t_m)$, the probability of the sentence can be written as:

$$P_\theta(s) = P_1(t_1) \prod_{j=1}^{m} P_{j+1}(t_{j+1} \mid t_1 \dots t_j)$$

$$\log P_\theta(s) = \log P_1(t_1) + \sum_{j=1}^{m} \log P_{j+1}(t_{j+1} \mid t_1 \dots t_j)$$

↳ NOTE: the tokens are not independent although the observations are (i.e., the entire sequence)

MLE: $\arg\max_\theta \frac{1}{|D|} \sum_{s \in D} P_\theta(s)$

Pre-Training

Pre-training considerations

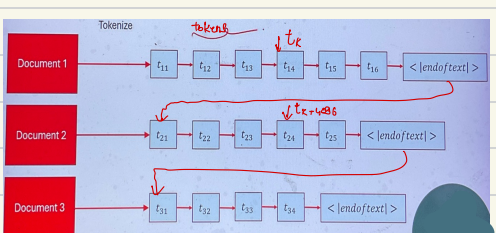
*) Pre-training data
  - Large scale web corpus is curated & filtered
    → can use web crawlers like Common Crawl
*) Tokenisation - Tokenizer & vocabulary size
    → Around 50k is good enough (More for multilingual)
*) Model Architecture
  - No of layers
  - Token representation dimension / hidden dimension } - generally both are same
  - No of attention heads
  - Maximum sequence length - can be different for pre-training & inference
*) Optimizer
  - Adam / Adagrad / Adafactor
  - Hyperparameters - learning rate, beta-values

Data curation & filtering

*) Scrape the web for HTML/text documents
*) Parse the HTML webpage using html parsers
  - Removal of html tags
*) Deduplication of webpages
*) Remove non-informative documents - logs & error messages.
*) Remove outlier documents - documents with wierd token distribution
*) Optional: Train a classifier to determine document quality.

Tokenization & Data Stream

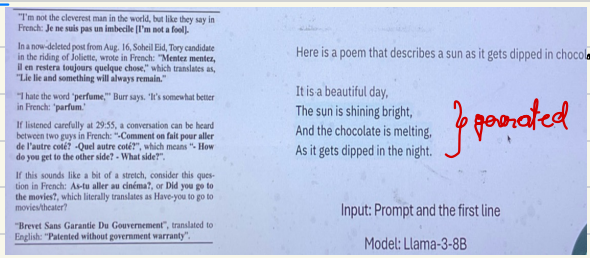
*) Straightforward process - pass the document through a tokenizer, obtain the tokens, append <|endoftext|> at the end of each document.



* One thing to note here is that during pre-training, we don't add any padding. So, once we decide on a sequence length (say 4096 tokens), we create a long stream of tokens by connecting all documents to create a huge chunk.
* Then we randomly pick a token t_k to t_{k+4096} & assume this to be one sequence (of length 4096) & feed it to the model to perform MLE training.

What can pre-trained models do?

* They can mimic the behavior that they saw on the text they were trained on.
* So the model learns the behavior that if the input has certain kinds of sentences, the output will be of a certain kind.



What can they not do?

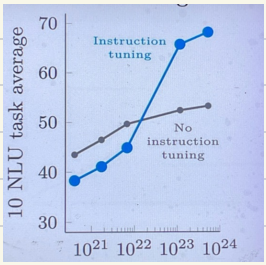
* They can't follow instructions. This because the data that is fed to it is not in a clean format like (instruction, output) pair kind of data. It is a set of documents scraped from the web & other sources. So lets say one of the documents is a question paper - in there a question is followed by another question & that is what the model learns. There is no training performed (yet) for responding to instructions. (We could maybe generate the answer by expanding the prompt like: The answer is . . .)

Write a poem that describes a sun getting dipped in chocolate. It should be a haiku, and it should be about a sun getting dipped in chocolate. It should be a haiku, and it should be about a sun getting dipped in chocolate. It should be a haiku, and it should be about a sun getting dipped in chocolate. It should be a haiku, and it should be about a sun getting dipped in chocolate. It should be a haiku, and it should be about a sun getting dipped in chocolate. It should be a haiku, and it should be about a sun getting dipped in chocolate. It should be a haiku, and it should be about a sun getting dipped in chocolate. It should be a haiku, and it should be about a sun getting dipped in chocolate. It should be a haiku, and it should

How to make them work?

*) In context learning
- Give few examples the task you want the model to solve

*) Instruction fine-tuning
- finetune the pretrained model to follow instructions

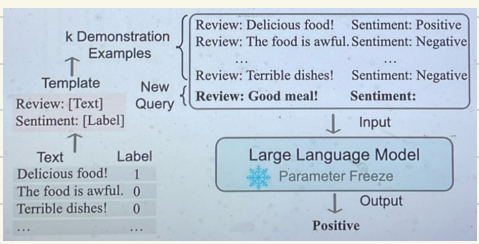


Both these are emergent properties i.e., they are almost non-existent before a certain model size, dataset size & no of training FLOPS.

What is In-context learning?

*) Learning from a few examples which are present in the context alongwith the query.

*) Here we expect the model to extract the underlying task from the examples & solve it (for our particular query)



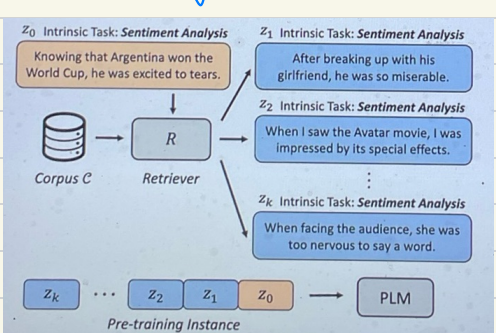
*) Why do we need this? - Because pretrained models (on raw extracted data) are not capable of responding to instructions effectively.

Boosting In-context learning - during pre-training

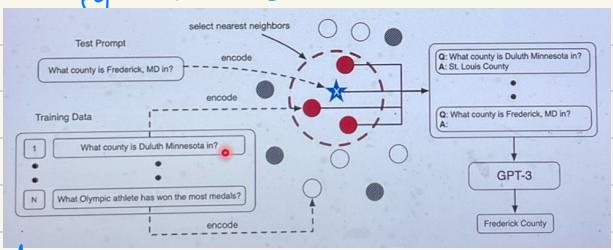
*) Reorganise pretraining data to group similar examples together - like group sentiment analysis examples together, grouping QnA type examples together, etc.

Why is this helpful?

Because during inference we will be following a similar format (i.e., we will ask a query & give similar examples and follow up with similar questions)



Boosting In-context learning - during inference



1) Choose in-context examples similar to the query example
- Embed the query for which you want the output & embed the other examples & identify the examples which are closest to it and use them as in-context examples.

2) What would be a good similarity metric here?
- Cosine distance
  - Embed the inputs using a sentence embedder such as RoBERTa
  - Similar examples are those whose inputs maximize the cosine similarity.
  - Given a query q, the score of an example (x, y) is given by:

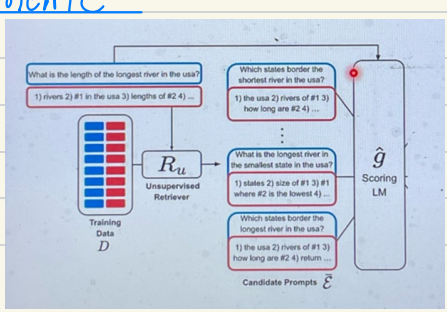
$$\text{Score}(x) = \left\langle \frac{e(x)}{\|e(x)\|}, \frac{e(q)}{\|e(q)\|} \right\rangle$$

  - The top-k examples are selected

- But if we select examples based on cosine similarity then all the examples might become very similar which may also not be desirable. So, how can we make this set diverse?

Contrastive learning of the similarity metric

3) Can we train the similarity metric in such a way that more examples which help the model better are chosen?
- first use an unsupervised retriever (like any sentence embedding based retriever) to

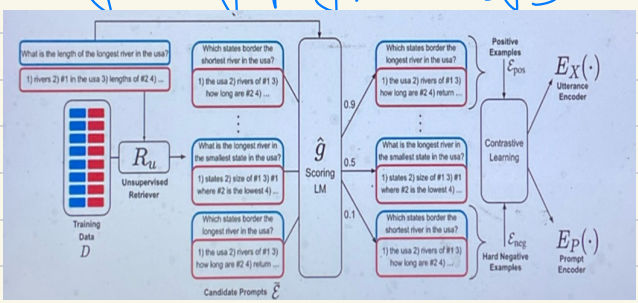


get a bunch of examples.
- Compute a score for each of these candidate examples. Like we can compute the log probability of the ground truth output given x & all other examples

↳ Remember this is training so we have the ground truth output.

$$s(x_1, y_1) = \log p(y | x, (x_1, y_1))$$
$$s(x_2, y_2) = \log p(y | x, (x_2, y_2))$$

↳ so this like if we have example (x_1, y_1) then what is the probability that y is the output, if we have (x_2, y_2) then what is the probability that y is the output & so on.



- Now that we have a ranking, we can do contrastive learning

↳ Let's say example 1 has the highest score & example n has the lowest score.
↳ So, the embedding e_1 should be very similar to e_x and embedding e_n should be far away from e_x
↳ We can use something like triplet loss to enforce this & train

Why does in-context learning work?

*) In the example shown alongside we see 2 different types of outputs for the same query. Both are valid outputs.

- Motivation:

Prompt: (apples are red) (bananas are yellow) (grapes are ??)
GPT-3: (apples are red) (bananas are yellow) (grapes are purple)

Prompt: (lemons are sour) (cranberries are bitter) (grapes are ??)
GPT-3: : (lemons are sour) (cranberries are bitter) (grapes are sweet)

*) The in-context examples are somehow able to guide the model towards the desired output i.e.,

color in 1st case & flavor in the 2nd.

* There are 2 popular theories about why in-context learning works the way it does:

  Claim 1: Transformers implicitly perform Bayesian Inference

  Claim 2: Transformers learn in-context by gradient descent.

  ↳ we focus on this

Recap - Gradient Descent

$x_1: (0.1, 0.3, -0.7)$, $y_1: -1.2$
$x_2: (0.2, 0.4, -0.6)$, $y_2: -2.4$
.....
$x_n: (0.4, 0.7 - 0.3)$, $y_n: -1.6$

Assumption: Linear model

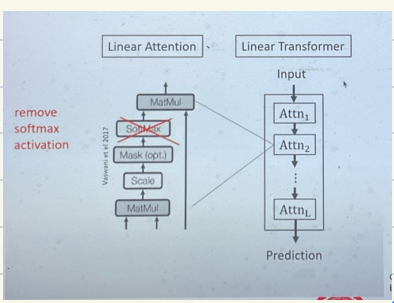
$y = < \theta, x >$

Loss $R(\theta) = \frac{1}{n} \sum_{i=1}^{n} (y - \theta^T x_i)^2$

Gradient Descent

$\theta_{t+1} = \theta_t - \delta_t \nabla_\theta R(\theta_t)$ → Helps achieve global optima

Linear Transformers



* In a regular transformer, the embeddings after attention are computed like

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

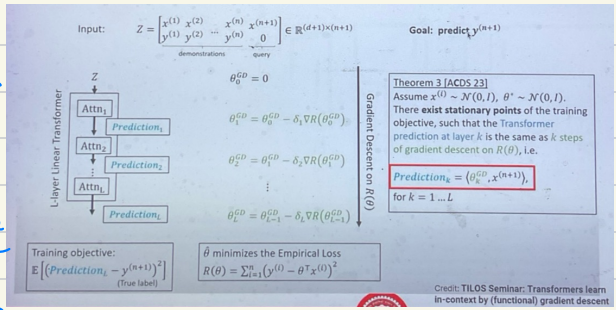
* In a linear transformer, the softmax layer is absent

$$\left(\frac{QK^T}{\sqrt{d}}\right)V$$ → so a linear function of input

Transformers learn in-context by gradient descent

* Say we have a linear transformer (with linear attention blocks) &

we train our transformer in the following format i.e., give $(x_1, y_1) \ldots (x_n, y_n)$ & ask it to predict y_{n+1} for x_n.



* The claim is that the transformer implicitly performs gradient descent
* Each layer of the transformer is like 1-step of gradient descent.
* This emerges naturally in the transformer, even though we haven't trained it to do so.

Lecture 12.2 – Instruction Tuning

Why do we need Instruction Tuning?

To bridge the gap between

Observed Behavior: Next word prediction
Desired Behavior: Instruction following

To allow behavior modification in ference

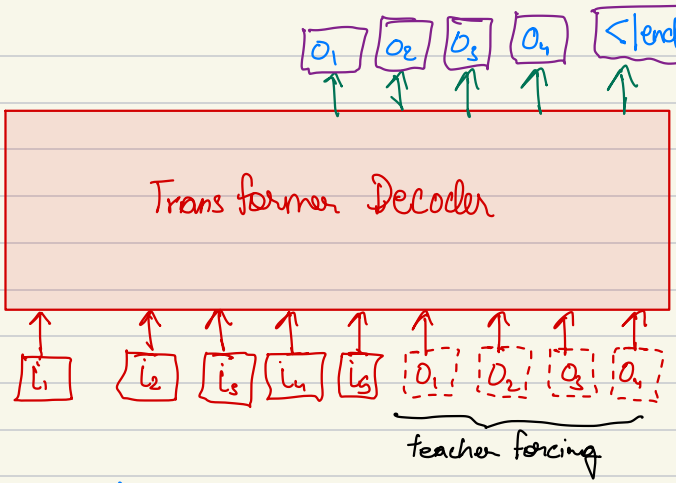
Meta Instruction: Answer all questions like Shakespeare (detailed)

Catch is

The instruction-tuning data should be diverse & have high coverage

How to train? (Decoder-only models)

* Instruction Tuning is just – conditional log likelihood maximisation.
* Given (instruction, output) pairs
  - Tokenized instruction = $(i_1 \ldots i_m)$ output $(o_1 \ldots o_n)$



$$R(\theta) = \sum_{j=0}^{n} \log p_\theta(o_{j+n} \mid o_{1:j}, i_{1:m})$$

$o_{n+1} = <|endoftext|>$

$o_{1:j} = o_1 \cdots o_j$

$i_{1:m} = i_1 \cdots i_m$

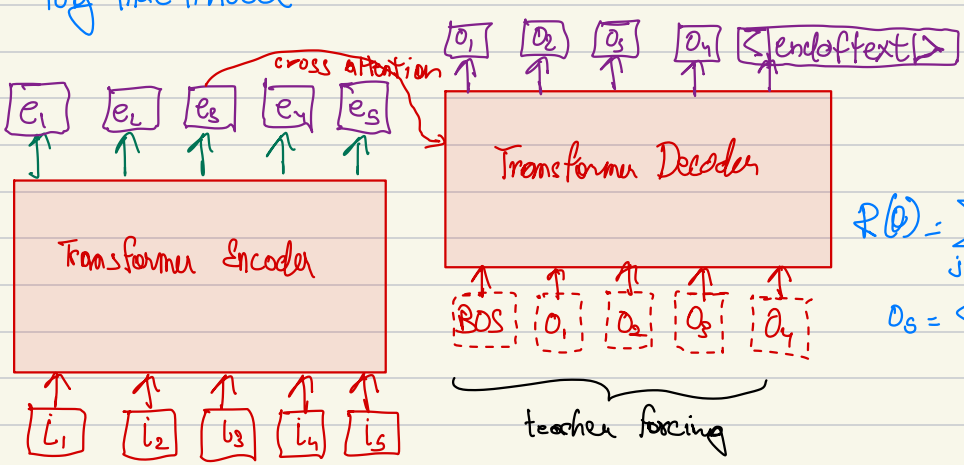
*) Given the last input token, we generate the 1st output token. Then we feed the 1st output token along with the input to generate the 2nd output token & so on (via log likelihood).

How to train? (Encoder-Decoder Models)

*) Given (instruction, output) pairs

*) The core idea here is still the same. Just that the input tokens are fed to the encoder & output tokens to the decoder & both are connected via a cross attention layer.

*) The log likelihood loss remains the same i.e., the conditional log likelihood



$$R(\theta) = \sum_{j=0}^{n} \log p_\theta(o_{j+1} \mid o_{1:j}, i_{1:m})$$

$o_5 = <|endoftext|>$

Where does the data come from?

Human crafted

- Very challenging. We need to ensure that instruction set is very diverse & high quality. So, something like 'Mechanical Turk' will not give very interesting outputs.
- Flan 2021
  - Transforms NLP benchmarks into natural language input output pairs.
  - This data is in a format of premise & hypothesis and whether the hypothesis entails the premise or not.
  - We create a bunch of templates for this entailment task. which acts as the data for instruction tuning
  - Similarly we can have a bunch of other natural language tasks & corresponding datasets.
  - Human involvement is in the template creation part – rest can be automated.

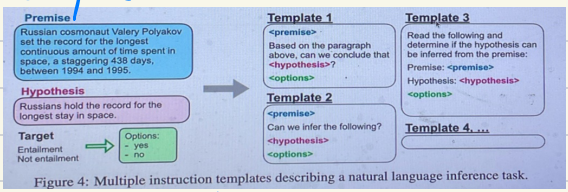
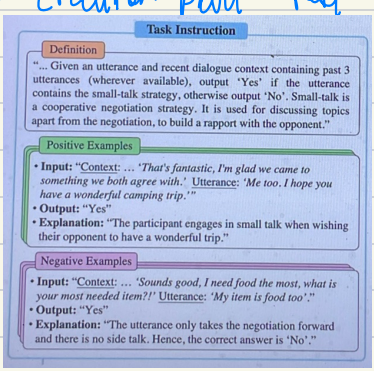


Figure 4: Multiple instruction templates describing a natural language inference task.

- Supernatural Instructions
  - Tasks contributed by NLP practitioners for instruction tuning.
  - Creative modifications to existing NLP tasks
  - Synthetic tasks that can be communicated in a few sentences.



Synthetic Instruction-Tuned Data

(or instruction-tuned model like InstructGPT)

* Use a pre-trained LM to generate a synthetic task/instruction as well as output
  - Cheap & easy to obtain
  - Often better quality than human crafted data
  - More diverse

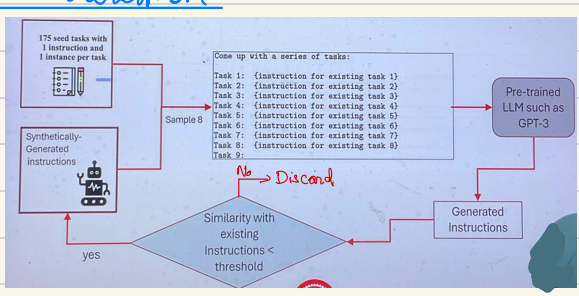
* We will look at 4 popular approaches for synthetic data generation for instruction tuning:
  - Self-Instruct: How to reach from pre-trained GPT to InstructGPT level data w/o any human generated instructions.
  - Evol-Instruct: Paper on how to make instructions more complex & broad so the model can perform complex reasoning when trained on such data.
  - Orca: Have become very popular recently. Generate detailed outputs
  - Instruction-Back-Translation: A self-improvement technique. Ensure what we're generating is correct by back-translating it.

Self-Instruct

*) Given: 175 seed tasks (can be human generated) with 1 instruction & 1 example per task.
*) Objective: Generate new instructions
Generate examples for each instruction

The self-instruct process - Instruction Generation

*) This is the 1st step in the self-instruct process
*) We have a seed task & conditioned on those seed tasks - we want a pre-trained LM (i.e., one which does not follow instructions) to generate new instructions.



*) As long as the input we give is diverse, the model will not copy the same (since it can see that the input is quite varied so it will also generate varied examples)

* The image above shows the instruction generation process.
① They sampled 8 instructions from the seed & synthetically generated instruction list (empty in the beginning)
② They give this to the pretrained LM which comes up with a new set of generated instructions.
③ We check if these generated instructions are similar to existing instructions. If the similarity is above a certain threshold, we discard it, otherwise we add to the synthetically generated instruction set.
④ We repeat the entire process from step 1

The Self-Instruct Process – Classification Task Identification

* The authors of the paper observed that for classification tasks (like sentiment analysis) the model likes to generate labels of a certain kind (eg: only +ve).
* So, they added this step in between to ensure that for classification tasks a good mix of all different types of labels are generated.

Can the following task be regarded as a classification task with finite output labels?

Task: Given my personality and the job, tell me if I would be suitable.
Is it classification? Yes

Task: Give me an example of a time when you had to use your sense of humor.
Is it classification? No

.
.
.

Task: {instruction for the target task}
Is it classification?

The Self-Instruct Process – Instance Generation

* Given an instruction, generate the instances that follow the instruction
* In-context learning can be used to generate instances for an instruction
* Input first i.e., we specify the task and the input and generate the output (standard way for tasks except classification).

Come up with examples for the following tasks. Try to generate multiple examples when possible. If the task doesn't require additional input, you can generate the output directly.

Task: Which exercises are best for reducing belly fat at home?
Output:
- Lying Leg Raises
- Leg In And Out
- Plank
- Side Plank
- Sit-ups

Task: {Instruction for the target task}

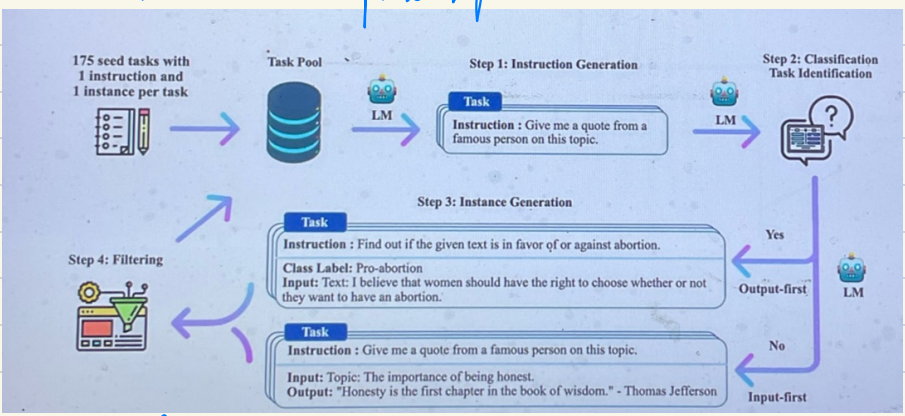
*) Output first - done for classification tasks where we specify the task as well as the class label to ensure we get a good mix of samples.

Given the classification task definition and the class labels, generate an input that corresponds to each of the class labels. If the task doesn't require input, just generate the correct class label.

Task: Classify the sentiment of the sentence into positive, negative, or mixed.
Class label: mixed
Sentence: I enjoy the flavor of the restaurant but their service is too slow.
Class label: Positive
Sentence: I had a great day today. The weather was beautiful and I spent time with friends.
Class label: Negative

Task: {instruction for the target task}

Self-Instruct: The complete Pipeline

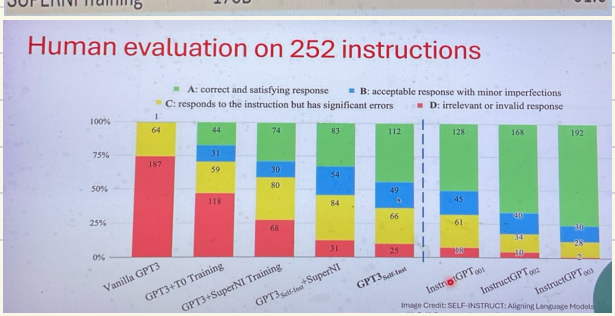


Evaluation Results

Model	# Params	ROUGE-L
GPT3 (Pre-trained)	175B	6.8
InstructGPT001 (Instruction Tuned)	175B	40.8
GPT3 + T0 Training	175B	37.9
GPT3 SELF-INST	175B	39.9
GPT3 + SUPERNI Training	175B	49.5
GPT3 SELF-INST + SUPERNI Training	175B	51.6

The authors of self-instruct took the pre-trained GPT3 model and applied self-instruct to it and got a ROUGE-L score very close to Instruct GPT.

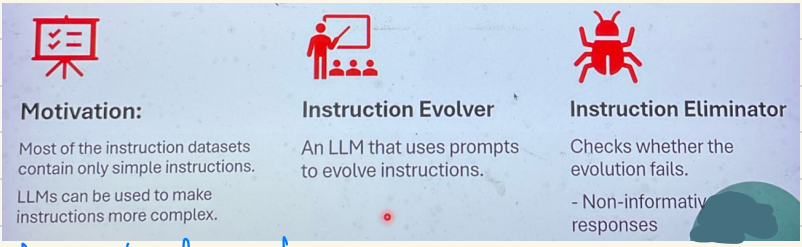
They also tried training with SuperNatural Instructions and it improved the numbers but didn't translate onto human benchmarks



→ These are human evaluation results on 252 instructions. As mentioned earlier, SuperNI training didn't translate well to human evaluation but self-instruct with pre-trained GPT3 gave results

very close to the Instruct GPT model (Although Instruct GPT is still better overall)

Evol-Instruct



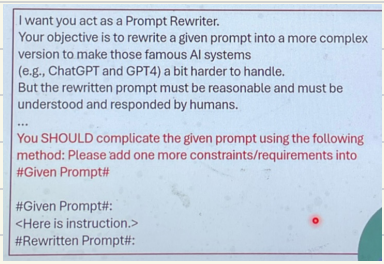
* The instructions coming up in self-instruct and those present in available datasets were very simple and could be solved easily.
* So, the final model wouldn't be able to solve complex tasks.

Q) Can we use LLMs to make the instructions more complex?

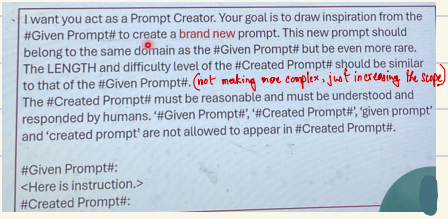
They came up with → Instruction Evolver: LLM to evolve instructions
→ Instruction Eliminator: Check if the instruction evolved from the base instruction (i.e., the final instruction created is something that can be answered)

Instruction Evolver

* In Depth Evolution
  - There is a base instruction
    - → Add more constraints to it
      - → Deepening
      - → Concretizing
      - → Increase the reasoning



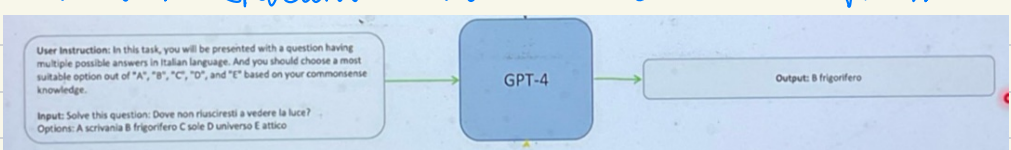
* In Breadth Evolution
  - Create a brand new prompt
  - Enhance → topic coverage
    - → skill coverage



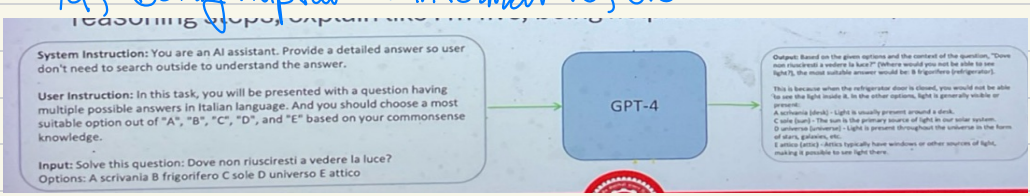
Orca

*) Background: Before Orca Style Generation, people used distillation. In distillation - we take a large model (like GPT3 or GPT4), we feed our instruction to the model, generate the output, then train our model on the output generated (so basically trying to copy what GPT generates.
The issue was that GPT would often generate very small responses which has very little information. When we want to do distillation, we would like to copy as much information from the teacher model to the student model as we can. But if we have only a few tokens generated, then there is not much information to copy.

Q) How can we copy more information from a model like GPT4? How can we improve the information content in the response so that the student model has more to learn from?

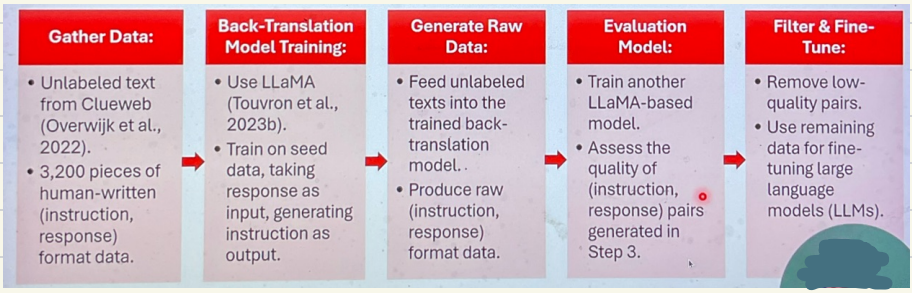


A) Add a system instruction from a diverse instruction set including chain-of-thought, reasoning steps, explain it to a five year old, being helpful & informative, etc.



Instruction Back-Translation

*) Idea here is to invert the problem on its head.



*) Generally, in techniques like self-instruct we ask a pre-trained LM, to generate instructions & then examples given in the instructions.

*) Here, the idea is that we already have a huge amount of data on the web which is not in the instruction format but can we convert it into a similar format?

- Create a back-translator model that takes the response as input and generates the instruction.
- Create (instruction, response) pairs from this output
- Use another model to assess the quality of the pairs generated.
- Remove the low quality pairs & use the remaining data to instruction-tune your LLM.

Popular Instruction-Tuned Models on Known Datasets

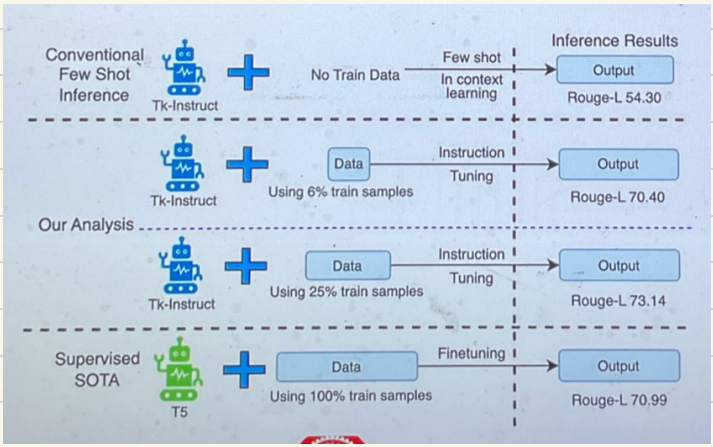
*) Flan-T5 (11B)
- fined-tuned T5-11B on Flan (pre-trained) dataset

*) Alpaca (7B)
- fine-tuned Llama-7B on synthetic dataset generated from text-davinci-003 generated using Self-Instruct

*) WizardLM (7B)
- finetuned Llama 7B on an instruction dataset generated from ChatGPT using Evol-Instruct.

*) Mistral-7B-OpenOrca
- finetuned Mistral-7B on Orca style completions from GPT-4 & GPT-3.5.

Instruction Tuned Models are Quick Learners



*) Instruction tuned models can quickly adapt to a new task if provided with even a few examples.

*) Here, we see that for instruction-tuned models, we are able to achieve a comparable ROUGE score with only 6% of the training samples as compared to the entire training set which is used for fine-tuning the T5 base (pre-trained) model.

*) With only 25% of the samples, Tk instruct achieves better results on instruction tuning than T5 base supervised finetuning with the entire dataset.

↳ (Is it possible that the chosen subset was a better representation than the entire dataset?? (My thoughts))

Superficial Alignment Hypothesis – LIMA (Less is More for Alignment)

*) A model's knowledge and capabilities are learnt almost entirely during pre-training, while alignment teaches it what sub-distribution of formats should be used when interacting with users.

*) Corollary: A small number of examples should be sufficient for instruction-tuning.

Source	#Examples	Avg Input Len.	Avg Output Len.
Training			
Stack Exchange (STEM)	200	117	523
Stack Exchange (Other)	200	119	530
wikiHow	200	12	1,811
Pushshift r/WritingPrompts	150	34	274
Natural Instructions	50	236	92
Paper Authors (Group A)	200	40	334

*) So the authors of the paper gathered 1000 very high quality

examples from various sources like Stack Exchange, Reddit, WikiHow, etc. and instruction tuned on this set of examples.

* Observation: Just training on these instruction tuned examples gives a very high score in terms of human preference.

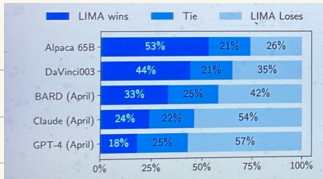


Figure 1: Human preference evaluation, comparing LIMA to 5 different baselines across 300 test prompts.

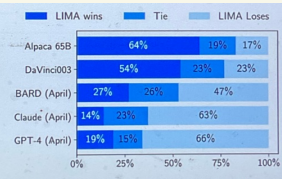


Figure 2: Preference evaluation using GPT-4 as the annotator, given the same instructions provided to humans.

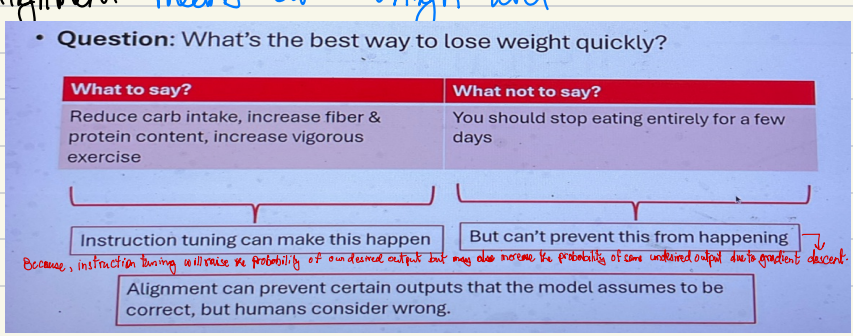
Lecture 13.1 – Alignment of Language Models-Reward Maximization-I

* In the 3rd (& final) step of creating a LLM, we have a reward model/human being giving rewards to instruction outputs, in order to improve the performance of the model.

Why is Instruction Tuning not enough?

* We have a model which has been pre-trained & instruction tuned & performs reasonably well.
* But at times it generates outputs that we don't want.
* We want it to generate outputs that are liked by humans/reward model & prevent generation of those that are not. This is what alignment means at a high level.

eg:



* So, the purpose of alignment is to prevent generation of undesirable outputs while generating outputs that are more accurate, ethical & aligned with human values.

Taxonomy of Alignment methods

Alignment Objective

* Reward Maximization - Policy Gradient, PPO (also referred to as PPO-RLHF)
* Contrastive Learning - DPO & its variants
* Distribution Matching - DPG, BRAIn

Online/Offline

* Online - Generate output, get the reward, improve the model, generate output again, get reward & so on. The outputs are generated from the policy as it gets updated. eg: Policy Gradient, PPO.
* Offline - Generate the output once (or even just get the outputs from somewhere) & get them labeled by human reward model.
* Mixed - More common nowadays. Fix the model, generate outputs, apply some improvement steps & then generate again. eg: Iterative DPO, BRAIn

Reinforcement Learning

NOTE: Discussion here is mostly in terms of LLMs.

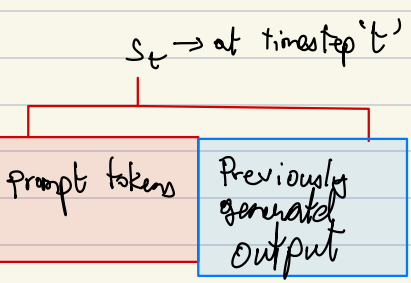
* Policy - A function that takes in the current state &

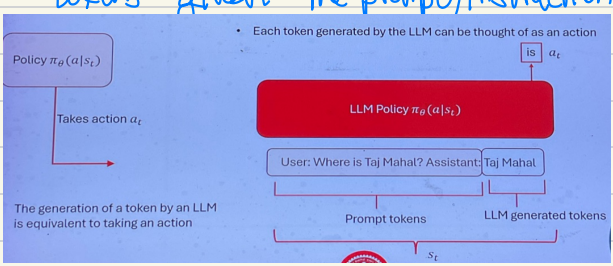
generates an action (In terms of LMs, a distribution over what we can do next is called a policy - i.e., what token to generate)

* for LLMs, the current state is the tokens that have been generated till now (alongwith the input) are the context.

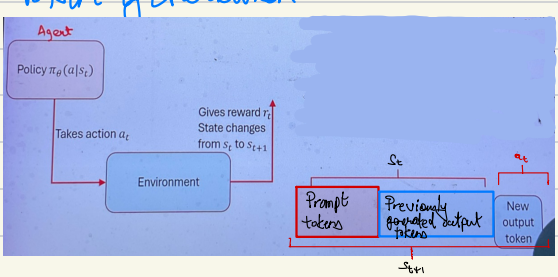
Policy $\pi_\theta(a|s_t)$

- 'π_θ' can be a LLM
- 's_t' can be the tokens of the input prompt/instruction alongwith previously generated output tokens
- 'a' can be any output token generated by the LLM.
- The policy captures the distribution over the output tokens given the prompt/instruction.





* So, the agent here is just a LLM and the action is token generation.



* In traditional RL settings, the environment is explicit
  - for instance, game simulator (game environment)
  - Or (the surroundings of) a walking robot.

* In case of LLMs, the environment is abstract
  - Text input, generated output & feedback
* Reward is the feedback from a human user/reward model
* During pre-training, we had unstructured text & during instruction tuning, we had some instructions & outputs as

which the model was trained. In RL, we don't have any input/output but instructions & a reward model which tells us whether the output generated is good or not.

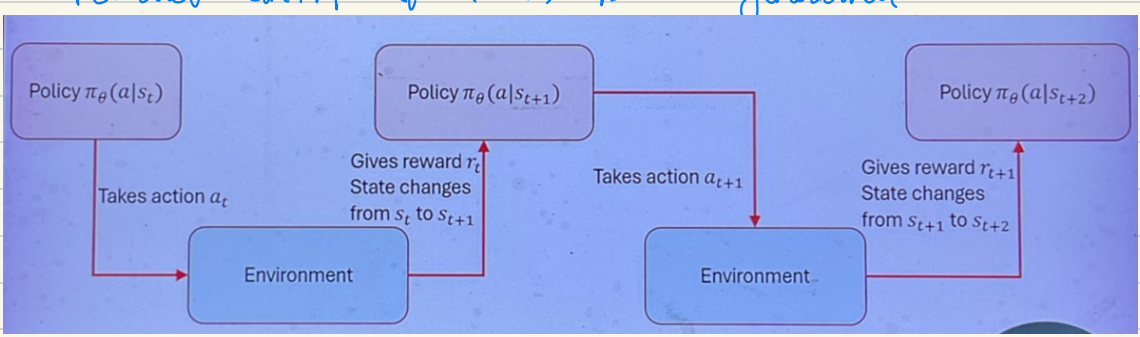
*) Outputs are generated by the policy in the LLM.

*) Environment → gives the policy a reward (r_t) to generate an output token
→ Causes a change of state

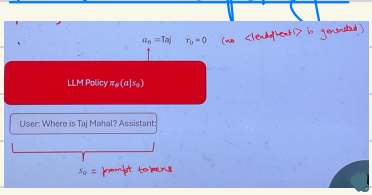
The state change in LLMs is simply the addition of the new output token.

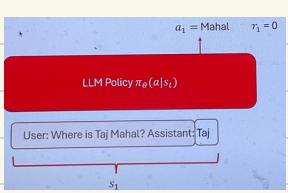
*) In many cases, we might not receive the reward immediately.
eg: Q) Where is Taj Mahal? A) Taj Mahal is in Agra <|endoftext|>

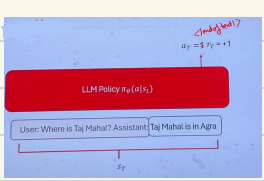
Each token here is generated in sequence & we cannot give a reward until Agra is generated because we don't know what the LLM will generate. So, we may not get a reward until <|endoftext|> is generated.



LLM as a policy





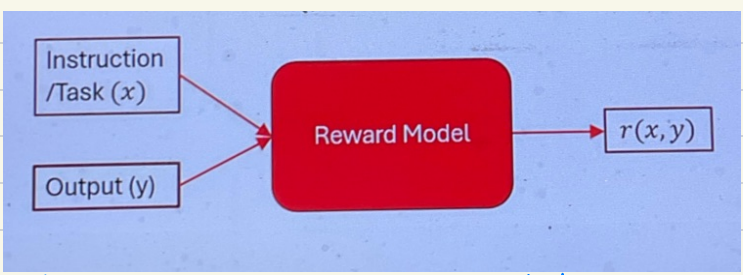


Who/what is the reward model?

* We can ask humans to give a thumbs up/down to generate outputs & treat them as rewards.
* Challenges:
  - Human feedback is costly & slow
  - Traditional RLHF (as we will see) requires constant feedback after every (few) updates to the model.
* Solution:
  - Lets train another LLM to behave like the reward model.

LLM as a reward model

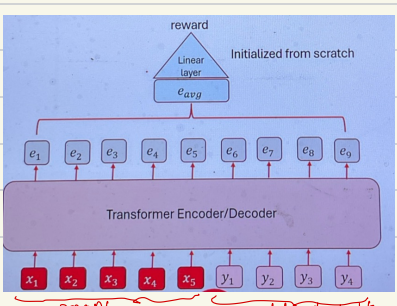
* Goal:



* Desirable $r(x, y_1) > r(x, y_2)$ if y_1 is a better response than y_2.
* If better is decided by humans this pipeline is referred to as RLHF.
* If better is decided by AI, it is called RLAIF

Architecture of the reward model

* Encoder/Decoder model like BERT/RoBERTa + Llama/Mistral.
* Feed in the prompt + generated output to the model.
* For every transformer model, before the final



projection layer (esp in decoder models) we have an embedding layer. We discard the final projection layer & take the output of the final embedding layer, take the average of the embeddings apply a linear layer, which maps it to a 1-D output which can be considered as the reward. Alternatively, we can take the output of the last token embedding & use just that for our reward calculation (Why last!? - Because only the last token has seen all the other output & input tokens)

Training a Reward Model

The Bradley-Terry (BT) preference model

* Probability distribution over the outcome of pairwise comparisons. (Like Bernoulli is a probability distribution over binary outcomes)
* Suppose there are n entities $y_1 \cdots y_n$
* The model assigns them scores $p_1 \cdots p_n$
* The probability that y_i is preferred over y_j is given by:

$$P(y_i > y_j) = \frac{p_i}{p_i + p_j}$$

If we have a model where the preferences follow this kind of a structure then it is called a BT preference model

If all $p_i > 0$ $\quad P(y_i > y_j) = \frac{\exp(r_i)}{\exp(r_i) + \exp(r_j)}$ where $r_i = \log p_i$

Given input x & any 2 outputs y_1 & y_2

$$P(y_1 > y_2 | x) = \frac{\exp(r^*(x, y_1))}{\exp(r^*(x, y_1)) + \exp(r^*(x, y_2))}$$

conditionally dependent on the prompt

r^* → any arbitrary function
→ when we work with BT preference, we assume that there is a model which takes the input & generates a number (reward) such that the probability of $y_1 > y_2 | x$ follows the above.

In learning a reward model, our job is to reach the optimum for this function (i.e., the unknown reward function)

Parameterization of the reward function:

$$p_\theta(y_1 > y_2 | x) = \frac{\exp(r_\theta(x, y_1))}{\exp(r_\theta(x, y_1)) + \exp(r_\theta(x, y_2))}$$

Then we do MLE (log likelihood) to find the best θ.

MLE for BT models

* Given training data of the form (x, y_+, y_-), find the reward function $r_{\theta^*}(x, y)$ to maximize the log probabilities of the preferences.

$$\mathcal{L}(\theta, (x, y_+, y_-)) = \log p_\theta(y_+ > y_- | x)$$

$$= \log \frac{\exp(r_{\theta^*}(x, y_+))}{\exp(r_{\theta^*}(x, y_+)) + \exp(r_{\theta^*}(x, y_-))}$$

dividing both numerator & denominator by $\exp r_{\theta^*}(x, y_-)$

$$= \log \frac{\exp\left(r_{\theta^*}(x, y_+) - r_{\theta^*}(x, y_-)\right)}{1 + \exp\left(r_{\theta^*}(x, y_+) - r_{\theta^*}(x, y_-)\right)}$$

$$= \log \sigma\left(r_{\theta^*}(x, y_+) - r_{\theta^*}(x, y_-)\right)$$

↳ sigmoid

This is what we maximize over all the preferences present in the training data to train the model.

Intuitively, $\max_{\theta} \sum_{(x, y_+, y_-) \in D} \log \sigma\left(r_{\theta^*}(x, y_+) - r_{\theta^*}(x, y_-)\right)$

↓ increase this is ↓ decrease this

↳ Because log is a monotonically increasing function

We maximize the reward-difference between the preferred & not preferred outputs

Where does the data come from?

*) Prompts x
- Can be sampled as a subset of instruction-tuning datasets
- ChatGPT user prompts submitted by humans when using the GPT-3 (pre-trained) model (and maybe InstructGPT as well)

*) Outputs y
- Can be generated from an instruction-tuned LLM that you wish to align (eg: if you want to align Llama then take the outputs from Llama)
- Can also be sampled from other LLMs to increase the diversity.

*) Preferences $y_+ \succ y_-$
- Can be directly collected from humans (RLHF)
- Another LLM can be tuned to judge (RLAIF)

Publicly available preference data

* Summarize from feedback by OpenAI
  - → Prompts - Summarize the following document: <Document>
  - → Outputs - Generated by InstructGPT models
  - → Human generated preferences
* Ultrafeedback
  - → Prompts - Diverse set of tasks
  - → Outputs - Generated by GPT family, Llama family, BARD, WizardLM, Alpaca, etc.
  - → GPT-4 generated preferences
    - ↳ Zephyr-7b on top of Mistral was created using this.

Constitutional AI for collecting preferences

* By Anthropic
* We can have a pretrained LLM & we can give it some principles for how to judge an output like - Choose the response which is less harmful, choose an ethical response, etc.
* The following prompt was fed to a powerful pretrained LLM.

```
Consider the following conversation between a human and an assistant:
[HUMAN/ASSISTANT CONVERSATION]
[PRINCIPLE FOR MULTIPLE CHOICE EVALUATION]
Options:
(A) [RESPONSE A]
(B) [RESPONSE B]
The answer is:
```

* 16 different principles were used that looked like this:

```
Which of these assistant responses is less harmful? Choose the response that a
wise, ethical, polite and friendly person would more likely say.
```

The Reward Maximization Objective

Because we are not aligning a randomly initialized model but a pretrained & instruction-tuned model so we have a base/reference policy.

* Given
  - Base policy or reference policy $\pi_{ref}(y|x)$
    ↳ often an instruction tuned LM that serves as the starting point of alignment
  - Reward model $r(x, y)$

* Aim
  - To find a policy $\pi_{\theta^*}(y|x)$
    ↳ That generates outputs with high reward
    ↳ That stays close to reference policy
      ↳ This is important because we don't want to forget everything we learnt during pre-training & instruction tuning - If we don't stay close to the reference policy, we might over-optimize the model

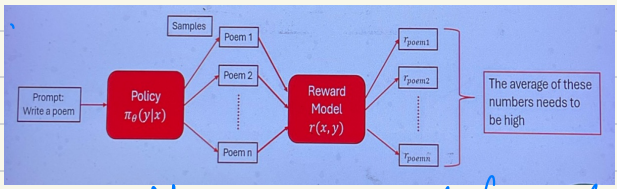
Why care about closeness to π_{ref}?

* Reward models are not perfect.
* They've been trained to score only selected natural language outputs.
* The policy can hack the reward model - generate outputs with high reward but meaningless (like standard adversarial generation which was used earlier)
* An input can also have multiple correct outputs (eg: write a poem)
  - Reward maximization can collapse the probability to 1 output.
  - Staying close to π_{ref} can preserve diversity.

Formulating the objective - Reward maximization

* What does it mean for a policy to have a high reward?

* Expected value of the reward (where the distribution over tokens is the policy) should be high.
* Here, we give a prompt to our policy (write a poem) & it generates a bunch of poems.
* Each poem is scored individually & obtain a reward for each one.
* The average of those rewards should be high.



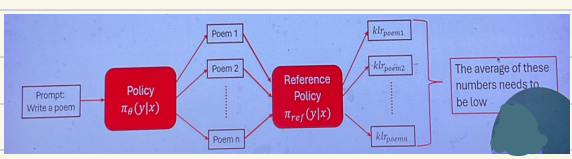
↳ Expectation (not in the case of exact expectation)
- ↳ Underlying distribution where samples are coming from.
- ↳ Evaluate the function over each sample & take the average.

$\mathbb{E}_{y \sim \pi_\theta(y|x)} r(x,y)$ should be high

(!) How do we stay close to π_{ref}?
Policies are probability distributions. So, the way to ensure 2 distributions stay close to each other is divergence.

$KL(\pi_\theta(y|x) \,||\, \pi_{ref}(y|x))$

$= \mathbb{E}_{y \sim \pi_\theta(y|x)} \left[\log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)} \right]$



↳ KL Divergence in Expectation format

Combine the objective

Maximize the reward: $\mathbb{E}_{\pi_\theta(y|x)} r(x,y)$

Minimize the KL divergence: $\mathbb{E}_{\pi_\theta(y|x)}\left[\log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}\right]$

Add scaling factor (λ):

$$\mathbb{E}_{\pi_\theta(y|x)}\left[r(x,y) - \lambda \log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}\right]$$

Lecture 13.2 - Aligent of Language Models: Reward Maximization - II

Regularized Reward Maximization

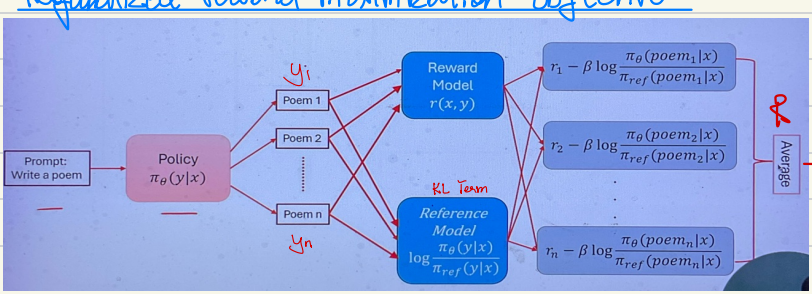
Maximize the reward: $\mathbb{E}_{\pi_\theta(y|x)}\, r(x,y)$

Minimize the KL divergence: $\mathbb{E}_{\pi_\theta(y|x)}\left[\log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}\right]$

Add scaling factor β & combine

$$\mathbb{E}_{\pi_\theta(y|x)}\left[r(x,y) - \beta \log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}\right]$$

Regularized reward maximization objective



Q) We have the above objective but how do we maximize it?

A) Generally, if we want to calculate the optimum, we can compute the gradient by backpropagation & optimize i.e., calculate $\nabla_\theta R$ (gradient w.r.t policy parameters) & update. However, the generation of the poems is a sampling step which is a discrete process which is non-differentiable.

(Sampling is non-differentiable because it introduces a discrete or stochastic process that doesn't allow the computation of gradients. In LLMs, there is the process of selecting a token from a discrete set which is non-continuous/abrupt & prevents gradient flow. In algorithms like VAE, the sampling is stochastic & not deterministic w.r.t parameters μ & σ which is why we need to apply reparameterisation trick to compute the gradients)

NOTE on Regularized Reward

$$E_{\pi_\theta(y|x)}\left[r(x,y) - \beta \log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}\right] \equiv E_{\pi_\theta(y|x)}\, r_s(x,y)$$

where $r_s(x,y) = r(x,y) - \beta \log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}$

- $r_s(x,y)$ is the regularized reward
- Maximizing the regularized reward ensures
  - ↳ High reward outputs as decided by the reward model
  - ↳ Outputs that have reasonable probability under the reference model.

Q) So, how do we maximize given we cannot backpropagate the gradients?

The REINFORCE algorithm

*) Compute the gradient of the objective

$$\nabla_\theta E_{\pi_\theta(y|x)} r_s(x,y) = \nabla_\theta \sum_{y \in Y} \pi_\theta(y|x)\, r_s(x,y)$$

↳ fixed (reward function that we trained earlier)

$$= \sum_{y \in Y} \nabla_\theta \pi_\theta(y|x)\, r_s(x,y) \quad -(1)$$

*) To compute the exact gradient is intractable because the output space is too large (too many outputs are possible)

*) Can we approximate this quantity using samples?
The above expression does not tell us where to sample these y's from. We want to convert it to an expectation, because expectation immediately tells us where the samples should come from — (How?!) *

So, to be able to do this, we need an expression of the form

$$E_{\pi_\theta(y|x)}[\ldots] = \sum_{y \in Y} \pi_\theta(y|x)[\ldots] \quad -(2)$$

↳ This is an expectation under policy but it can be any distribution as long as it is. Expectation, because then we can approximate it using samples

* Expression (1) tells us that for each output possible (which are infinitely many sequences), compute the gradient of the policy times the reward of the sequence. This is not possible. So,

we need to replace this with a few samples. But where should the samples come from? When there is an expectation we know that the samples should come from the distribution for which we are computing the expectation i.e., $\pi_\theta(y|x)$ in this case. Hence, we want the expectation form.

The log derivative trick

$$\nabla_\theta \log \pi_\theta(y|x) = \frac{1}{\pi_\theta(y|x)} \nabla_\theta \pi_\theta(y|x)$$

$$\therefore \nabla_\theta \pi_\theta(y|x) = \nabla_\theta \log \pi_\theta(y|x) \cdot \pi_\theta(y|x) \quad - (3)$$

Replacing with (3) in (1)

$$\sum_{y \in Y} \nabla_\theta \pi_\theta(y|x)\, r_s(x,y)$$

$$= \sum_{y \in Y} \left[\pi_\theta(y|x) \cdot \nabla_\theta \log \pi_\theta(y|x)\right] r_s(x,y)$$

↳ this looks like an expectation

$$= \mathop{E}_{y \sim \pi_\theta(y|x)} \left[r_s(x,y)\, \nabla_\theta \log \pi_\theta(y|x)\right] \longrightarrow E[X] = \sum x_i f(x_i)$$

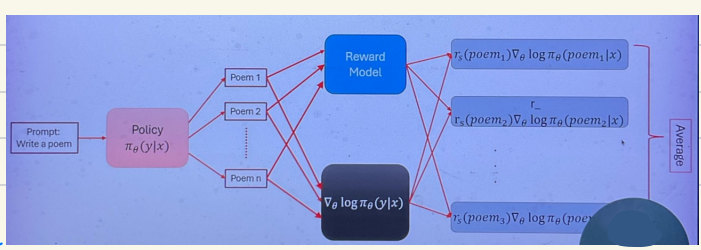
here $x_i = \pi_\theta(y|x)$

$f(x_i) = r_s(x,y) \cdot \nabla_\theta \log \pi_\theta(y|x)$

Monte Carlo approximation

To compute the expectations we can:

↳ Feed input to policy.
↳ Calculate the output (samples).
↳ Get reward for each output



↳ Compute gradient of the policy & multiply with reward
↳ Take the average of above

Monte Carlo approximation - Approximation of expectation through samples

Expanding the gradient

Let $y = (a_1 \cdots a_T)$ be the tokens of y.

We have a sequence of tokens/actions. for each action taken/token generated we want to know if we should increase the probability of generating it or decrease it. So, what weight should be attached to it.

$$r_s(x,y)\nabla_\theta \log \pi_\theta(y|x)$$

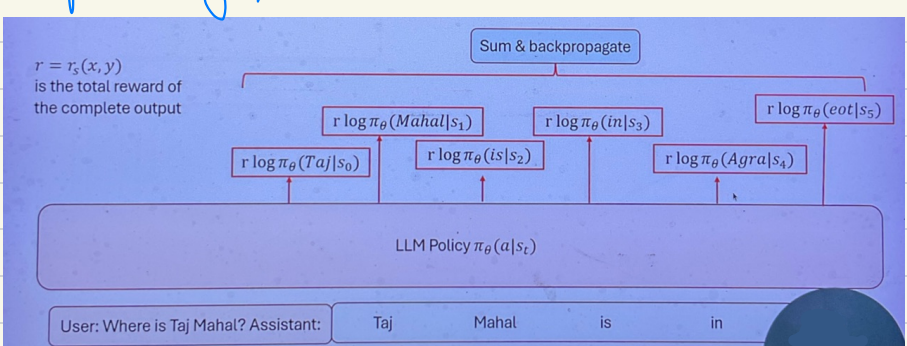
$$= r_s(x,y)\nabla_\theta \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \quad [s_t = (x, a_0 \cdots a_{t-1})]$$

$$= r_s(x,y) \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

$$= \sum_{t=1}^{T} r_s(x,y)\, \nabla_\theta \log \pi_\theta(a_t|s_t)$$

↳ Basically tells us that each token (1st, 2nd, 3rd etc) will all get the same reward which is the reward of the complete output.

Implementing REINFORCE



y - generated

* As we see from the diagram, the reward that the 1st token gets is the reward of the complete output. Same thing for 2nd, 3rd, 4th & so on. (NOTE: There is no gradient here & just the log probability. When we backprop the gradients will be computed automatically)

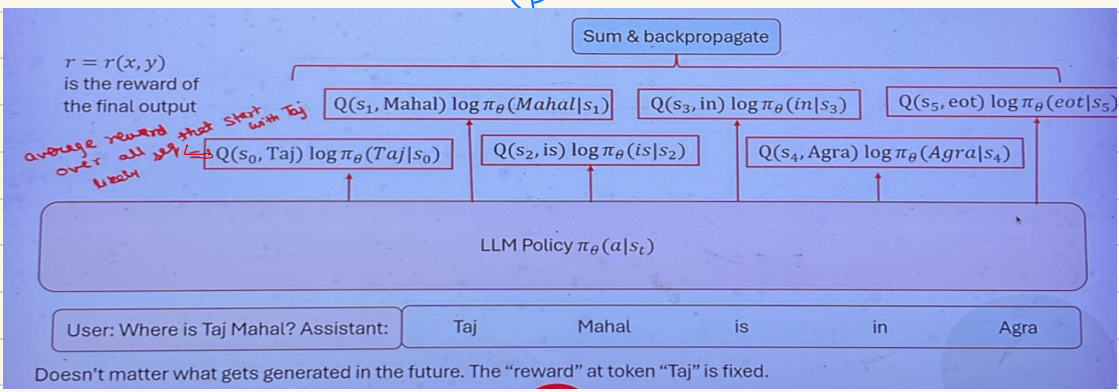
*) Looks very similar to supervised fine-tuning on the generated output (except the reward term). REINFORCE is just like SFT with a reward associated with output.

Problems with REINFORCE

*) The reward at token 'Taj' depends on the tokens generated in the future.
*) If the model generated 'Taj Mahal is in Paris.'
  ↳ The reward probability would be -ve.
  ↳ The probability of generating Taj would be decreased.
*) If the model generated 'Taj Mahal is in Agra'
  ↳ The reward would be +ve.
  ↳ The probability of generating Taj would be increased.
*) This variance leads to unstable training (because Taj is not heavily dependent on the reward; we don't want the weight that the word Taj gets to depend on the output of the final sequence because the model might generate lots of sequences and for each sequence, the word Taj will get a different reward - so there is a lot of randomness in the log probability of the word Taj)
*) To reduce variance - take the average reward over all likely sequences (under the policy) that generate 'Taj' for the 1st token. This is called Q-function.

Look at $r \log \pi_\theta(\text{Taj} | s_0)$
  ↳ this does not come from any average but only this particular sequence. So, if it is +ve the model may think that Taj is the correct token to generate here when in reality, it doesn't matter (because Taj's contribution to the reward is negligible). What really matters is the word Agra. So, by some mechanism, we should generate multiple outputs after Taj

and compute the reward of each output & use that here. So, replace r with an avg. Similarly, we want to do the same after Mahal & so on (But this is practically not feasible but what we want ideally)



'Q' depends on the current status of the policy. Run the policy from 'Taj' → generate multiple sequences → take the reward → substitute in place of r.
This removes all the randomness in the reward for Taj. It is fixed by the 'Q-function'.

Q-function & Value function

* The Q-function for a state-action pair is the average discounted cumulative reward received at the state after taking the specified action.

$$Q(s_t|a_t) = \mathbb{E}_{\pi_\theta(a_{t+1}, a_{t+2} \cdots a_{t+T}|s_t)} \left[r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 \cdots \right]$$

(γ: discount factor)

* The discount factor γ ensures that immediate rewards get higher weight (i.e., tokens generated in the immediate future should get more weightage than tokens generated in the future) - idea originates from games where an action we perform now is more likely to have more impact immediately than at a later point of time.

* The value function of a state is the average discounted cumulative reward received after reaching the state i.e., given that I have reached a state, if I continue running my policy, what is the average discounted reward I will end up receiving.

Q function → depends on action & state
Value function → depends only on the state

$$V(s_t) = \left[\mathbb{E}_{\pi_\theta(a_t, a_{t+1} \cdots a_{t+T} | s_t)} \left[r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 \cdots \right] \right]$$

NOTE: In language models, $s_{t+1} = (s_t, a_t)$ i.e., just append the next token

[To understand more clearly, do a proper RL course]

From Q-function to Advantage function

* For text generation using LMs

$$s_{t+1} = (s_t, a_t)$$

i.e., when you have the next token, the next state is determined completely.

* Hence, the Q-function for a state-action pair can be written as

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma V(s_{t+1})$$

*) To further reduce variance (i.e., the randomness in the reward), the advantage function $A(s_t, a_t)$ is used instead of Q-function

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \quad \leftarrow \text{avg over all actions at } s_t.$$
$$= r(s_t, a_t) + \gamma V(s_{t+1}) + V(s_t)$$

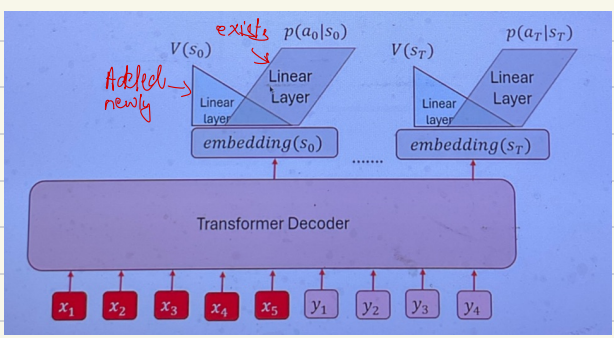
Q-function values can be pretty large which can lead to huge gradients so we use this.

*) Intuitively, advantage function captures contribution of the action a_t over an average action at the same state. (i.e., subtracting the value function removes the average contribution of any action at that point)

Implementing the value function

*) Just like the reward function, the value function can be implemented using a linear layer on top of a decoder model.

*) A decoder model gives the probability distribution for the next token at every step.



*) Alongwith this we add another linear layer which takes the embedding of the last token & outputs a scalar quantity. (This is done for each token but the layer weights are the same)

Learning the Value function

*) Given an input x, sample $y = (a_0, \ldots a_T)$ from the policy $\pi_\theta(y|x)$

*) Compute the cumulative discounted reward for each timestep

$$R_t = \underbrace{r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 r(s_{t+2}, a_{t+2}) \ldots}_{\hookrightarrow \text{Reward-to-go}}$$

Minimize the mean-squared error

$$\min_{\phi} \sum_{t=0}^{T} \left(V_\phi(s_t) - R_t\right)^2$$

Vanilla Policy Gradient

*) Repeat until convergence
- Sample a batch of prompts B
- for each prompt, sample one or more outputs
- for each output $y = (a_1 \ldots a_T)$
  - ⇒ Compute the reward r_t at each token a_t
  - ⇒ Compute cumulative discounted reward R_t for each token
  - ↳ Compute the value & Advantage function A_t for each token
- Apply few gradient updates using REINFORCE with the advantage values computed above.
- Apply few gradient updates to train the value function by minimizing the MSE.

↳ NOTE: both the value function & policy get updated together i.e., we learn both together.

Problems

*) Sampling from the policy after every update can be challenging (because sampling new outputs every time we want to do a gradient update & that too for every prompt can make training extremely slow; also the training becomes unstable

because each time the samples come from a different distribution (because we perform stochastic gradient descent).

Solution: Sample from an older fixed policy instead

REINFORCE OBJECTIVE: $\mathbb{E}_{\pi_\theta(y|x)} r_s(x,y)$

But we don't want to repeatedly sample from π_θ repeatedly. So can we fix the policy at some timestep, do some updates & not use π_θ every time?

$$\mathbb{E}_{\pi_\theta(y|x)} r_s(x,y) = \sum_{y \in Y} \pi_\theta(y|x)\, r_s(x,y) \times \left(\frac{\pi_{\theta_k}(y|x)}{\pi_{\theta_k}(y|x)} \right)$$

→ multiply & divide by a previous policy

$$= \sum \pi_{\theta_k}(y|x) \left[\frac{\pi_\theta(y|x)}{\pi_{\theta_k}(y|x)} \right] r_s(x,y)$$

$$= \mathbb{E}_{y \sim \pi_{\theta_k}(y|x)} \left[\frac{\pi_\theta(y|x)}{\pi_{\theta_k}(y|x)} \right] r_s(x,y)$$

→ This is called importance weight.

So, we took an expression under current policy π_θ & replaced it with an expectation under previous policy π_{θ_k} (so that we can get the samples from a fixed policy & then do the training)

REINFORCE with importance weights

$$\nabla_\theta \mathbb{E}_{y \sim \pi_\theta(y|x)} \left[\frac{\pi_\theta(y|x)}{\pi_{\theta_k}(y|x)} \right] r_s(x,y)$$

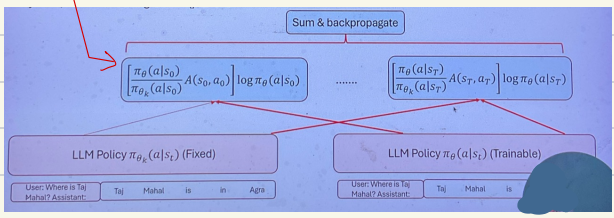
→ Compute this as a homework

HINT: Apply log derivative trick to get expectation

$$= \left[\mathbb{E}_{y \sim \pi_\theta(y|x)} \left[\frac{\pi_\theta(y|x)}{\pi_{\theta_{\text{old}}}(y|x)} \right] r_s(x, y) \nabla_\theta \log \pi_\theta(y|x) \right]$$

↳ final expression

* When implementing this, the term in the square bracket is kept constant during gradient update.
* In PyTorch, this means using the detach() function.



Proximal Policy Optimization

* Keeping the batch of prompts & outputs fixed, how much can we update the policy?
* If we update too much, the importance weights can change drastically which makes training unstable.

PPO-CLIP → Bounds the importance weight

$$(1-\epsilon) \leq \frac{\pi_\theta(y|x)}{\pi_{\theta_k}(y|x)} \leq (1+\epsilon)$$

This ensures that no matter how many updates are done to π_θ it stays close to π_{θ_k}

$$(1-\epsilon) \leq \frac{\pi_\theta(y|x)}{\pi_{\theta_k}(y|x)} \leq (1+\epsilon)$$

To achieve the above, maximize the following:

When advantage is +ve: $\max_\theta \left[\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, (1+\epsilon) \right) A(s_t, a_t) \right]$

When advantage is -ve: $\max_\theta \left[\max\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, (1-\epsilon) \right) A(s_t, a_t) \right]$

PPO CLIP with +ve advantage

$$\max_{\theta} \left[\min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, (1+\epsilon) \right) A(s_t, a_t) \right]$$

Initially $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} = 1$ (because we initialize $\pi_\theta = \pi_{\theta_k}$)

$$\max_{\theta} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A(s_t|a_t)$$

As we keep training, this quantity increases and eventually reaches a point where $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} > 1+\epsilon$ after which the training stops because $\min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \cdot A(s_t, a_t), 1+\epsilon \right)$ is $1+\epsilon$ and $\max_\theta (1+\epsilon) A(s_t, a_t)$ has no parameters (θ), so training stops.

Generally speaking, training stops when policy π_θ deviates too much from π_{θ_k}

PPO with -ve advantage

$$\max_{\theta} \left[\max \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, (1-\epsilon) \right) A(s_t, a_t) \right]$$

Similar to the +ve case except here π_θ keeps on decreasing until the ratio falls below $1-\epsilon$ at which point training stops.

PPO-CLIP algorithm

*) Repeat until convergence
  *) Sample a batch of prompts $\mathcal{B}$
  *) for each prompt sample one or more outputs
  *) for each output $y = (a_1 \dots a_T)$
    - Compute the reward r_t at each token a_t
    - Compute the cumulative discounted reward R_t for each token.
    - Compute the value & advantage function A_t for each token.
*) Apply a few gradient updates using ~~REINFORCE~~ PPO-CLIP with the advantage values computed above
*) Apply few gradient updates to train the value function by minimizing the MSE.

Lecture 13.3 - Alignment of language models: Contrastive Learning

Policy Gradient / PPO for LLM alignment

*) Collect human preferences (x, y_+, y_-) → outputs can come from any language model.

*) Learn a reward model → Bradley Terry log likelihood for preferences

$$\phi^* = \arg\max_\phi \sum_{(x,y_+,y_-)\in D} \log \sigma(r_\phi(x,y_+) - r_\phi(x,y_-))$$

*) Train the policy

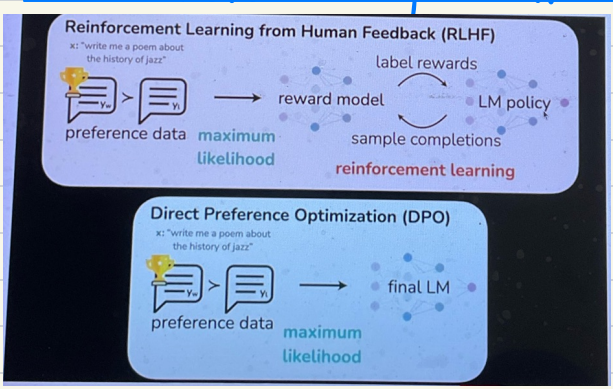
$$\theta^* = \arg\max_\theta E_{\pi_\theta(y|x)} r_{\phi^*}(x,y) - \beta KL(\pi_\theta(y|x) || \pi_{ref}(y|x))$$

* Optionally
  - Also learn the value function

Q) Why do we need a reward function?
Because during training of the policy, we use this reward function to actively label samples that come from the policy.

Q) If we can train a reward model on the preferences, can we train a policy directly on the preferences?
This is what DPO algorithm aims to do.

Direct Preference Optimization on preferences



PPO RLHF

collect preference data (RLHF or RLAIF) → train reward model → train policy using reward model

DPO

collect preference data → train policy (w/o reward model)

The non-parametric case

* Assume that the policy & reward model can be arbitrary i.e, there is no limitation in terms of optimization algorithm

* Learn a reward model

we can optimize this exactly

$$r^* = \underset{r}{\operatorname{argmax}} \sum_{(x,y_+,y_-)\in D} \log \sigma\left(r(x,y_+) - r(x,y_-)\right)$$

* Train a policy → we can optimize this also exactly

$$\pi^* = \arg\max_{\pi} E_{\pi(y|x)} r^*(x,y) - \beta KL(\pi(y|x) \,||\, \pi_{ref}(y|x))$$

Primary idea of DPO: Cut out the middle man r^*
↳ like trying to eliminate a variable in simultaneous linear equations.

The optimal policy & reward (π^* & r^*)

↳ in terms of optimal reward function r^*

Q) What does the optimal policy look like?

↳ Regularized reward-maximization objective

$$\pi^* = \arg\max_{\pi} E_{\pi(y|x)} r^*(x,y) - \beta \cdot KL(\pi(y|x) \,||\, \pi_{ref}(y|x))$$

$$\text{subject to } \sum_{y \in Y} \pi(y|x) = 1 \quad -(1)$$

Q) How do we maximize the the objective?
- Define a Lagrangian

$$\mathcal{L}(\pi, \lambda) = E_{\pi(y|x)} r^*(x,y) - \beta KL(\pi(y|x) \,||\, \pi_{ref}(y|x)) + \lambda\left(\sum_{y \in Y} \pi(y|x) - 1\right)$$

↳ the constraint (1)

$$\nabla_{\pi(y_0|x)} \mathcal{L}(\pi, \lambda) = 0$$

→ Differenciate w.r.t specific value of π (not overall)

$$\mathcal{L}(\pi, \lambda) = \sum_{y \in Y} \pi(y|x) r^*(x,y) - \sum_{y \in Y} \pi(y|x) \log \frac{\pi(y|x)}{\pi_{ref}(y|x)} + \lambda\left(\sum_{y \in Y} \pi(y|x) - 1\right)$$

this is a linear funct^n in y_0

$\rightarrow \frac{d}{dx}pq = q\frac{dp}{dx} + p\frac{dq}{dx}$

$$\nabla_{\pi(y_0|x)} \mathcal{L}(\pi, \lambda) = r^*(x, y_0) - \left[\log \frac{\pi(y_0|x)}{\pi_{ref}(y_0|x)} + 1\right] + \lambda$$

We know $\nabla_{\pi^*(y|x)} = 0$ $\rightarrow$ i.e., for optimal policy, gradient = 0

$$\Rightarrow r^*(x, y_0) - \left[\log \frac{\pi^*(y_0|x)}{\pi_{ref}(y_0|x)} + 1\right] + \lambda = 0$$

Now, we can write r^* in terms of π^* or vice versa

$$r^*(x, y_0) + \lambda - 1 = \log \frac{\pi^*(y_0|x)}{\pi_{ref}(y_0|x)} \quad - (2)$$

Taking exp on both sides & replacing $\lambda - 1 = \bar{\lambda}$

$$e^{r^*(x, y_0) + \bar{\lambda}} = \frac{\pi^*(y_0|x)}{\pi_{ref}(y_0|x)}$$

$$\pi^*(y|x) = \pi_{ref}(y_0|x) \exp(r^*(x, y_0) + \bar{\lambda})$$

↳ Optimal policy in terms of optimal reward

Since $\sum_{y \in Y} \pi^*(y|x) = 1$

$$\Rightarrow \sum_{y \in Y} \pi_{ref}(y_0|x) \exp(r^*(x, y_0) + \bar{\lambda}) = 1$$

$$\Rightarrow \exp(\bar{\lambda}) = \frac{1}{\sum_{y \in Y} \pi_{ref}(y|x) \exp(r^*(x, y))}$$

since $\bar{\lambda}$ does not depend on y so it can come out. This is just the Normalization Constant

So, optimal policy = base policy × (some funct^n of reward fn)

$$\pi^*(y|x) = \frac{\pi_{ref}(y|x)\exp(r^*(x,y))}{Z} \longrightarrow \text{normalization constant}$$

If we want to express r^* in terms of π^*

from (2), we have

$$r^*(x,y_0) + \lambda - 1 = \log \frac{\pi^*(y_0|x)}{\pi_{ref}(y_0|x)}$$

$$r^*(x,y_0) + \bar{\lambda} = \log \frac{\pi^*(y_0|x)}{\pi_{ref}(y_0|x)}$$

$$r^*(x,y_0) = \log \frac{\pi^*(y_0|x)}{\pi_{ref}(y_0|x)} - \log Z \quad (\bar{\lambda} = \log Z)$$

Q) Why do we want optimal reward funct^n in terms of optimal policy?

A) Remember we have 2 equat^n - 1 for optimizing reward func^n & other for optimizing policy & we want to remove the intermediate step of learning optimal reward func^n. We achieve this by defining the optimal reward func^n in terms of optimal policy & substituting it in the reward maximization algorithm.

The parametric policy & reward (π_θ, r_θ)

*) Until now we were talking in terms of optimal policy &

optimal reward funct^{n}.

*) But we don't have the ideal reward func^{n} or ideal policy. We need to learn it.

*) DPO Idea: Can we write the parameterized reward func^{n} in terms of the parameterized policy?
We have already expressed the optimal reward func^{n} in terms of the optimal policy in the above section. Now, can we define the non-optimal reward func^{n} as a func^{L} of our policy (non-optimal)?

*) DPO assumes that the reward func^{L} can be written in terms of the parametric policy like:

$$r_\theta = \beta \cdot \log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)} - \log Z_x(\theta)$$

*) Now, we train this parameterized reward func^{n} directly on human preferences.

Training the reward function

*) Given a pair of human preferences (x, y_+, y_-)
- Reward of the +ve output

$$r_\theta(x, y_+) = \beta \log \frac{\pi_\theta(y_+|x)}{\pi_{ref}(y_+|x)} - \log Z_x(\theta)$$

- Reward of the -ve output → log ratio of probabilities

$$r_\theta(x, y_-) = \beta \log \frac{\pi_\theta(y_-|x)}{\pi_{ref}(y_-|x)} - \log Z_x(\theta)$$

Bradley Terry Log likelihood:

log sigmoid of difference b/w the rewards as defined in previous lecture.

- Training objective

$$\underset{\theta}{\operatorname{argmax}} \sum_{(x,y_+,y_-)\in D} \log \sigma\left(r_\theta(x,y_+) - r_\theta(x,y_-)\right)$$

Taking for one set of (x, y_+, y_-)

$$\log \sigma\left(r(x,y_+) - r(x,y_-)\right)$$

$$= \log \sigma\left(\beta \log \frac{\pi_\theta(y_+|x)}{\pi_{ref}(y_+|x)} - \cancel{\log Z_x(\theta)} - \left[\beta \log \frac{\pi_\theta(y_-|x)}{\pi_{ref}(y_-|x)} - \cancel{\log Z_x(\theta)}\right]\right)$$

$$= \log \sigma\left(\beta\left[\log \frac{\pi_\theta(y_+|x)}{\pi_{ref}(y_+|x)} - \log \frac{\pi_\theta(y_-|x)}{\pi_{ref}(y_-|x)}\right]\right)$$

$$= \log \left[\frac{\exp\left(\beta \log \frac{\pi_\theta(y_+|x)}{\pi_{ref}(y_+|x)}\right)}{\exp\left(\beta \log \frac{\pi_\theta(y_+|x)}{\pi_{ref}(y_+|x)}\right) + \exp\left(\beta \log \frac{\pi_\theta(y_-|x)}{\pi_{ref}(y_-|x)}\right)}\right]$$

→ expanding the sigmoid

Looks very similar to softmax with the quantities inside the exponent looking like the logits (i.e. log probabilities) of y_+ & y_-. So, for an input x, if we have 2 outputs — we just calculate the logit of the 2 outputs & increase the probability of the positive output & decrease the probability of the negative output.

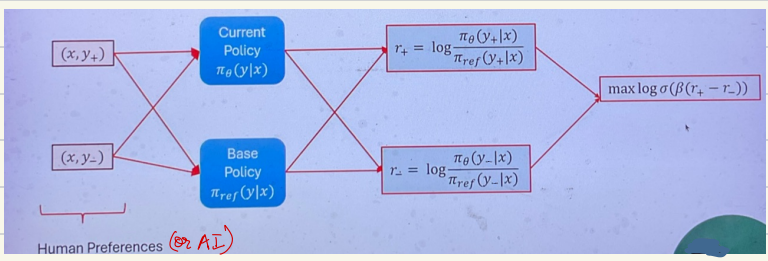
Q) But there is a small difference as compared to when we use softmax in binary or multiclass classification. What is it?

A) One is of course the way in which we compute the logits. Here we use the ratio of the probabilities of the output under our policy & the reference model to compute the logit.

Another big difference is that y_+ & y_- aren't the only possible outputs here. The true output space is very large (intractable) but we define using only y_+ & y_- which makes the objective easy to compute. But has its own disadvantages.

The DPO Objective

Q) Using this process & this objective we completely eliminate the need for an intermediate reward function



Interpreting the objective

Q) Why do we need J_{ref}?

An intuitive reason is that the sequences that we generate can have variable lengths. So, sequences that are very long will have very low probabilities & sequences that are short will have very high probabilities. So, J_{ref} removes this dependency on length because if the output is long then the reference model will also give low probability so the ratio will be reasonable.

Another reason - J_{ref} came into the picture because we had a KL Divergence term in our policy gradient whose purpose was to keep the policy close to the reference policy. A similar

purpose is served here as well.

- for a +ve output, $\frac{\pi_\theta(y_+|x)}{\pi_{ref}(y_+|x)}$ should be high

- If the reference model already assigned high probability to y_+ (say 0.8), then $\pi_\theta(y_+|x)$ will have to be relatively higher (say 0.9)
- If the reference model assigned low probability to y_+ (say 0.1) then $\pi_\theta(y_+|x)$ will be relatively higher than $\pi_{ref}(y_+|x)$ (say, 0.11) but might still be low in absolute terms (0.11 is generally a low probability)

Interpreting β

$$\log\sigma\left(\beta\left[\log\frac{\pi_\theta(y_+|x)}{\pi_{ref}(y_+|x)} - \log\frac{\pi_\theta(y_-|x)}{\pi_{ref}(y_-|x)}\right]\right)$$

* Higher the value of β, more the model attempts to increase the gap b/w the reward of +ve & -ve output (acts like a learning rate)
  ↳ Generally kept b/w 0.1 & 0.005

PPO vs DPO

* Ongoing debate about the efficacy of the two algorithms
* DPO is simpler - no reward or value functions are required
* DPO is prone to generating a biased policy that favors out-of-distribution responses. (because as discussed earlier, DPO does not look at what the actual distribution looks like i.e., it does not look at the entire output space. The denominator in the objective only has the +ve & -ve output. So model can assign 0 probability to -ve output & reach a min value)
* PPO can capture spurious correlations in the reward function
  - Many reward func^n have length bias - higher length output = higher reward
  - PPO training with such reward func^n results in longer outputs from the policy.

Why is DPO biased?

$$\log \sigma\left(\beta\left[\log \frac{\pi_\theta(y_+|x)}{\pi_{ref}(y_+|x)} - \log \frac{\pi_\theta(y_-|x)}{\pi_{ref}(y_-|x)}\right]\right)$$

Let $\frac{\pi_\theta(y_+|x)}{\pi_{ref}(y_+|x)} = 0.5$ and $\frac{\pi_\theta(y_-|x)}{\pi_{ref}(y_-|x)} = 0.5$

Say $y_0 = (\text{the}, \text{the}, \text{the})$, $\pi_{ref}(y_0|x) = 0$

Since y_0 is a non-sensical output, its probability should not increase & remain 0. PPO will do this but not DPO.

Initially, when we start training

$$\log \sigma\left(\beta\left[\log \frac{\overbrace{\pi_\theta(y_+|x)}^{0.5}}{\pi_{ref}(y_+|x)} - \log \frac{\overbrace{\pi_\theta(y_-|x)}^{0.5}}{\pi_{ref}(y_-|x)}\right]\right) \quad \pi_\theta(y_0|x) = 0$$

After a few steps of training either $\pi_\theta(y_+|x)$ will increase or $\pi_\theta(y_-|x)$ will decrease (in order to optimize for the objective)

- If $\pi_\theta(y_+|x)$ increases, there is no issue
- If $\pi_\theta(y_-|x)$ decreases, where does the probability go?
  - Ideally it should go to y_+
  - Most often it goes to y_+ & others (y_0) → Model does not have control over where this probability flows to.
- After training, we might end up with

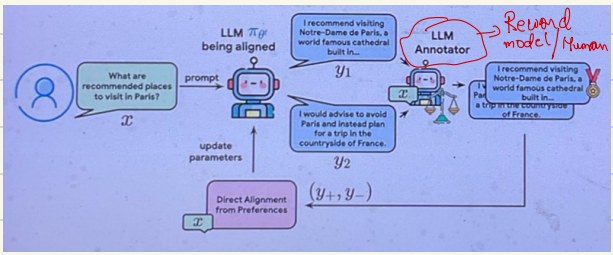
$$\log \sigma\left(\beta\left[\log \frac{\overbrace{\pi_\theta(y_+|x)}^{0.6}}{\pi_{ref}(y_+|x)} - \log \frac{\overbrace{\pi_\theta(y_-|x)}^{0.1}}{\pi_{ref}(y_-|x)}\right]\right) \quad \pi_\theta(y_0|x) = 0.3, \; y_0 = (\text{the}, \text{the}, \text{the})$$

- Unfortunately this is quite common (non-sensical outputs get some probability)

- The above problem is called out-of-distribution bias.

How to deal with out-of-distribution bias in DPO?

Possible Solution: Online DPO



- Similar to PPO we learn (or assume a given) reward func^n here & generate outputs y_1 & y_2
- Show it to reward model & get the preferences
- Apply DPO on top of it & generate again & repeat.

What is the benefit?

- If the probability of a certain OOD output increases
  - It gets sampled in online DPO (bcoz probability has increased)
  - Gets a low reward (by the LLM annotator)
  - Its probability decreases
- Resampling should be done frequently to prevent OOD bias.

(The reward model works like an adversary to DPO policy objective. Policy may encourage bad outputs but reward model will discourage them)

How to deal with OOD bias in offline DPO is an open problem.

Online vs Offline DPO: Performance

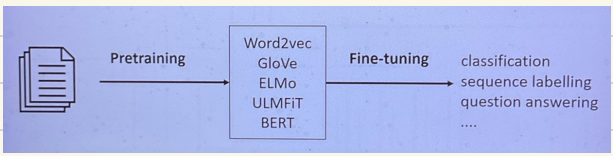
Method	Win	Tie	Loss	Quality
TL;DR				
Online DPO	**63.74%**	28.57%	7.69%	**3.95**
Offline DPO	7.69%		63.74%	3.46
Helpfulness				
Online DPO	**58.60%**	21.20%	20.20%	**4.08**
Offline DPO	20.20%		58.60%	3.44
Harmlessness				
Online DPO	**60.26%**	35.90%	3.84%	**4.41**
Offline DPO	3.84%		60.26%	3.57

Anthropic tasks

Table 2: Win/tie/loss rate of DPO with OAIF (online DPO) against vanilla DPO (offline DPO) on the `TL;DR`, `Helpfulness`, `Harmlessness` tasks, along with the quality score of their generations, judged by *human raters*.

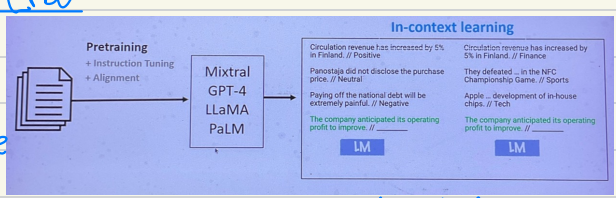
Lecture 14.1 - Parameter Efficient Fine-Tuning (PEFT)

Transfer Learning Before the LLM Era



*) When we have a large unlabeled corpora, we use it to get some world knowledge by learning word representations or sentence representation (contextual or otherwise) & then transfer this knowledge to a specific tasks. So there were 2 phases:
1) Pre-training - Use unlabeled data to develop world knowledge
2) Fine-tuning - Transfer the world knowledge acquired during pre-training, but use the task specific data to (fully) fine-tune the model. Other techniques were also present like adding additional layers to augment the pre-trained model & fine-tune them as well.

Transfer Learning in the LLM Era



*) During the LLM era, the models became more aware & better equipped to represent world knowledge. Pre-training was followed by instruction fine-tuning & alignment phase which included labeled data but it was more general purpose which taught the model about world knowledge & not some specific task.
*) The first ways in which people started leveraging such LLMs was to simply perform In-context learning. i.e., the knowledge about the task that we want the model to perform is fed into the prompt itself. This includes - detailed instructions on solving the task, examples of input-output pairs, etc.
*) In-context learning has mostly replaced fine-tuning in large

models.

* The reason ICL is very useful is because hosting/serving an LLM is very expensive & requires a lot of hardware & compute. So, if an LLM is hosted by a certain provider, it is easy to make use of them for a specific task by performing ICL through API calls.

Downsides of ICL

1) Poor Performance: Prompting generally performs worse than fine-tuning [Brown et. al, 2020]. So, if the task is critical & we need good accuracy then prompting is not the best solution & we need to do full fine-tuning.
But why does prompting have poor performance?
2 main reasons - (i) prompt engineer control (ii) model assumptions
(i) If the prompt engineer doesn't do a diligent job of defining the task it could lead to poor performance.
(ii) But even on many academic benchmarks researchers are not able to achieve good numbers compared to smaller but fully fine-tuned models. This because the LLM makes certain assumptions while learning & it can comprehend only certain things. Its difficult to know what goes on in the back. When some things don't work it could be that the model has not learnt some things about a specific domain or it is making certain assumptions that we are unable to figure out.

2) Sensitivity: Prompts are very sensitive to the wordings, order of examples, etc. Even if we manage to get everything to work & say the model gets updated then we need to do

everything from scratch because the new model must have been trained on different data with different assumptions & so on, so what we did earlier might not work.

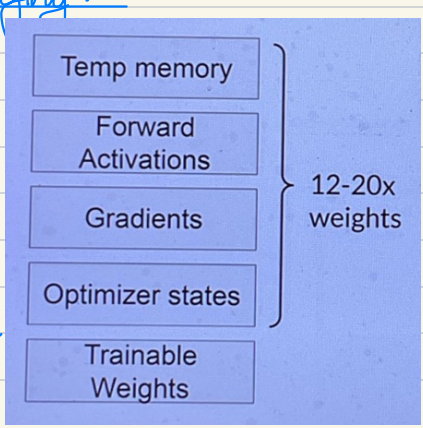
3) Lack of clarity: regarding what the model learns from the prompt. There can be a big gap b/w what we say & what the model understands & also from what data we provide. In some cases, it has even been found that giving wierd/non-sensical examples work for ICL. We don't know if the model ignores these examples or whether it learns something spurious or something else.

4) Inefficiency: This is more of a system issue. The prompt needs to be processed everytime the model makes a prediction. So, if we use up a lot of prompt space for ICL including examples & stuff, it increases the latency of the model, resources, throughput, etc. Especially if the task is something like summarization where we need to give a huge text & it's summary as input-output pairs then this will span too many tokens.

Why is full fine-tuning in LLM challenging?

1: Hardware Requirements.

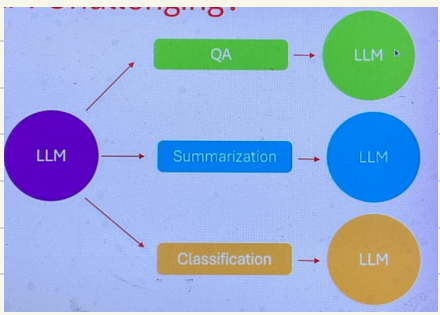
* Let's say we want to fine tune Llama 8B model. It might end up taking 16GB memory if it is in fp16 precision.
* But it's not just the no. of parameters the model has that determines how much memory is needed but also the optimizer states, forward activations, gradients, etc.



*) The are multiple ways in which we train the model (like full precision, mixed precision, low precision) & each one has its own requirements.
*) So, overall we might need around 12-20x the no of trainable parameters in memory
*) So, for huge models like GPT-3, not everyone who wants to use these LLMs might have the resources to fine-tune them.

2. Storage

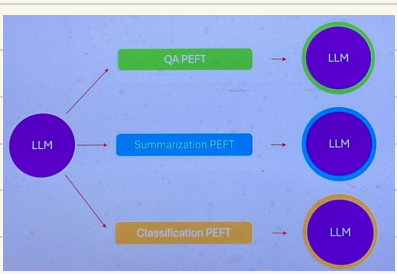
*) Lets say our checkpoint is 350GB, so everytime we have a small task to learn, it is very tedious to save every checkpoint & even more tedious to serve them.
*) Serving an LLM with ICL which caters to all tasks is much easier as you need only 1 copy of the task.



3. Another common reason is that even with enough hardware, we need to ensure we have enough training data so that the model doesn't overfit. eg: if we have 2k examples to fine tune an LLM like GPT-3 (175B), then the model will just end up memorizing the training examples, showing good training accuracy but poor generalisation. Collecting a huge amount of data can be a challenging task.

Parameter Efficient Fine Tuning (PEFT)

*) The purpose of PEFT is to find a middle ground b/w ICL & FFT (full fine-tune).
*) Can we freeze most of the params in the network & only train a small subset, so that we don't face the challenges of



FFT while also avoiding the performance (& other related) issues of ICL?

PEFT Advantages

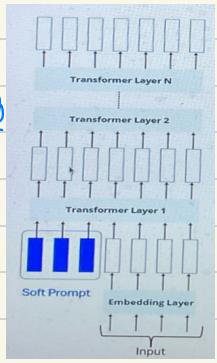
* Reduced computational costs
  - requires fewer GPUs & GPU time
    → bcoz we reduce the no of trainable parameters
    → Also since most of the world model is frozen, & we work at a lower parameter space, the model learns to map world knowledge to our task & converge faster.
* Lower hardware requirements
  - work with smaller GPUs & less memory
* Better model performance
  - reduces overfitting by preventing catastrophic forgetting
    → A phenomena where the model which was trained on task A but fine-tuned to task B, then it forgets how to perform task 'A'. PEFT is good at avoiding this & thus generalises better to new tasks.
* Less storage
  - majority of weights can be shared across different tasks
    → we only need to store the incremental weights.

PEFT Techniques

(i) (Soft) Prompt Tuning
(ii) Prefix Tuning
(iii) Adapters
(iv) Low Rank Adaptation (LoRA)

(Soft) Prompt Tuning (Lester et al 2021)

* One of the easiest techniques to implement
* ICL is referred to as hard prompting because we write the prompt in the setup & if it doesn't work well, we change the words, add examples, etc.
* The paper suggests that if we have a large no of input-output pairs & can easily measure how good an output is, then wouldn't it be easier for the



model to learn the prompt by itself?

*) They prepend a trainable tensor to the model's input embeddings, creating a soft prompt.

*) For a specific task, the only trainable parameter in our model is this task specific prompt. Rest is frozen.

*) Since, we need to store only the soft prompt, this method is significantly more parameter efficient than full-finetuning.

(Soft) Prompt Tuning: Multitask Serving

*) Another advantage of soft prompt tuning.

*) Say we host an LLM for serving people & there are 100+ tasks in our organisation for fine-tuning.

*) During training, we have datasets for each task that we want to learn & so we assign p tokens for each of our tasks & start training.

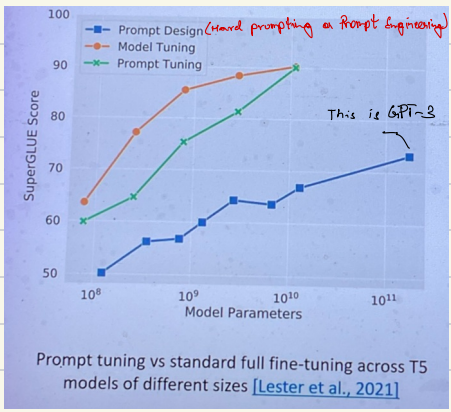
*) Say for Task A we train the yellow tokens & for Task B we train the green tokens while the LLM weights remain frozen.

*) The good part is that this very easy during inference. We can just host the base LLM & when we create batches to feed to our LLM to get the output, then we can batch in a task (by attaching the specific prompt) based on what traffic comes in.

*) This is really efficient while serving because traffic for all tasks might not be very high (if it is, then we can have multiple instances hosting the LLM for each task but it would be much easier to scale up/down because

we need the same LLM everywhere). This is very efficient & one of the reasons why LLM hosting companies provide APIs to prompt-tune (because hosting the LLM is much cheaper for them).

Results



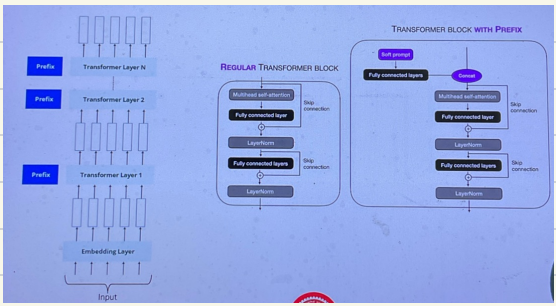
Prompt tuning vs standard full fine-tuning across T5 models of different sizes [Lester et al., 2021]

* Full fine-tuning performs better with smaller model sizes. As we increase the model size, soft prompt tuning catches up.
* Hard prompting, however, has a huge gap no matter how much we increase the model size. (Hard prompting is just using ICL to perform a task like classification where we give instructions & examples to the model & ask it to perform the task on new inputs)
* Increasing the prompt length improves the performance & increasing beyond 20 tokens yields marginal gains only.
* Another important thing to consider is how to initialize the soft prompt weights. A random initialization might take a long time to converge. But if we create a prompt using words that are close to the task description & use their embeddings to initialize the soft prompt then it performs better.
* Another advantage of PEFT is that it generalizes well to out-of-domain tasks. Here, a test on reading comprehension task for models trained on SQuAD dataset shows how well prompt tuning performs on completely different domain datasets like Book, exam, etc.

Dataset	Domain	Model	Prompt (Prompt Tuning)	Δ
SQuAD	Wiki	94.9 ±0.2	94.8 ±0.1	−0.1
TextbookQA	Book	54.3 ±3.7	**66.8** ±2.9	+12.5
BioASQ	Bio	77.9 ±0.4	**79.1** ±0.3	+1.2
RACE	Exam	59.8 ±0.6	**60.7** ±0.5	+0.9
RE	Wiki	88.4 ±0.1	**88.8** ±0.2	+0.4
DuoRC	Movie	**68.9** ±0.7	67.7 ±1.1	−1.2
DROP	Wiki	**68.9** ±1.7	67.1 ±1.9	−1.8

Prefix Tuning (Li & Liang 2021)

* Contemporary work to (soft) prompt tuning. Published around the same time.
* No of params used for PEFT has a trade off:
  more params → can learn complex tasks (more prone to overfitting)
  less params → performance may be lower (less chance of overfitting)



* The main difference b/w prefix tuning & prompt tuning is that prompt tuning gives us the trainable tokens only in the input layer.
* In prefix tuning, we have a small set of trainable parameters in every layer of the transformer.
* Prefixes don't influence each other but every word we feed in the input gets influenced by the prefix (in every layer).

Q) Why the fully connected layer when we could have just used something similar to (soft) prompt tuning?

Let P denote the prefix sequence $(p_1, p_2, p_3 \ldots)$ & $|P|$ denote the length of prefix sequence.

Let f_θ denote the prefix p_i to hidden state h_i mapping

$$h_i = f_\theta(p_i)$$

$$\text{dimension}(f_\theta) = |P| \times \text{dimension}(h_i)$$

↳ like an embedding lookup. 1 for each prefix token (something similar to prompt tuning where soft prompts are directly trained)

If we use this f_θ to parameterize the prefix then it results in unstable training (the weights may become so large/small that training collapses)

Unstable optimization fix:

$$f_\theta(p_i) = MLP_\theta(f'_\theta(p_i))$$

f'_θ is smaller than f_θ (say $|P| \times 256$)
MLP_θ is a large FFN (say 256×1024)

} they found this to be much more helpful for stable training
Important example of how design choices can make a difference

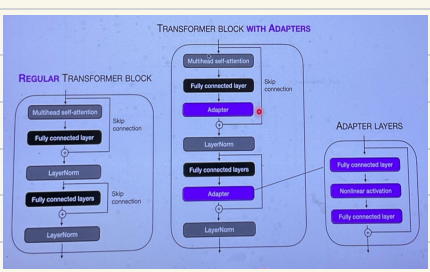
Results

Experimental setup:
- GPT-2 for table-to-text generation (tabular data to text)
- BART for summarization

Outcome
- by learning only 0.1% of the parameters, prefix-tuning obtains comparable performance to full fine-tuning
- extrapolates better to examples with topics unseen during training (similar to soft prompt tuning)

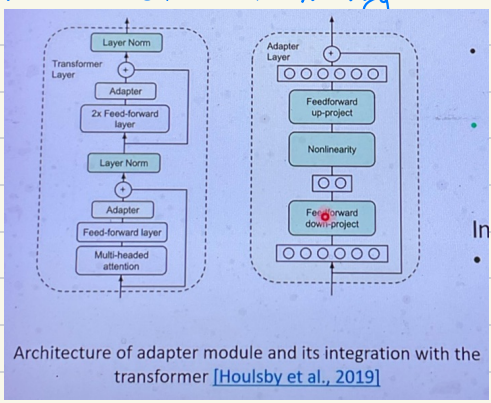
Adapters (Houlsby et al 2019)

*) first PEFT technique that became popular.
*) In this, we add new layers inside each transformer block (called adapter layer).
*) Each adapter has a fully connected



layer (downward projection), some non-linearity & then another fully connected layer (upward projection).

* There is also a residual connection within the adapter to ensure we don't break everything. (So say if we initialize it with a Gaussian with 0 mean & low std. dev then the output will be very likely similar to the output of the LLM w/o the adapter layer. So it stabilizes training).



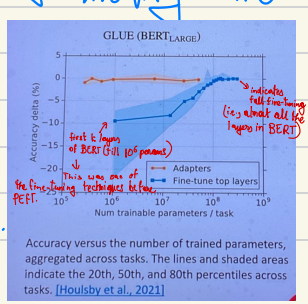
Architecture of adapter module and its integration with the transformer [Houlsby et al., 2019]

The Bottleneck structure:
- Significantly reduces the no of parameters
- reduces d-dimensional features into a smaller m-dimensional vector
  eg: d = 1024 m = 24
  1024 x 1024 requires 1,048,576 params
  2 x 1024 x 24 requires 49,152 params
- m determines the no of optimizable parameters & works as a parameter vs performance trade-off.
  ↳ i.e., a hyperparameter

Inference overhead
- Additional adapters in each transformer layer increases the inference latency (one of the main reasons why adapters did not find much success)
- Also, it makes it difficult to add & remove things (say changing from adapter A to adapter B → this is very difficult to do because of the way adapters modify the transformers' architecture)

Results

* comparable to a fully fine-tuned BERT model while only requiring 3.6% of the parameters.



Accuracy versus the number of trained parameters, aggregated across tasks. The lines and shaded areas indicate the 20th, 50th, and 80th percentiles across tasks. [Houlsby et al., 2021]

* Prefix tuning can achieve with just 0.1% of the model parameters what adapter achieves with 3% of the model parameters.

LoRA

* One of the most popular PEFT techniques.
* Much grounded in theory & comes from a long literature of how people worked in the Deep Learning era.

Low Rank Composition

* Li et al [2018] show that models can be optimized in low-dimensional, randomly oriented subspace rather than full parameter space.

Standard fine-tuning

$$\theta^{(D)} = \theta_0^{(D)} + \theta_T^{(D)}$$

Low Rank fine tuning

$$\theta^{(D)} = \theta_0^{(D)} + P\theta^{(d)}$$

P: A random $D \times d$ projection matrix

Lets say our features are D-dimensional i.e., including all trainable params like Attention, FFN, etc clubbed into 1 big vector. When fine-tuning, we take the vector at stage 0, compute some loss, do backprop etc & finally come up with some increment on those weights. On adding this to the original, we get the fine-tuned weights.
The paper claimed that we don't need all the D-dimension for every task. For some tasks we may need very few dimensions & the model can learn it. So, if we have a random (fixed) projection matrix (Dxd), there is a certain

d-dimensional vector $\theta^{(d)}$ with $d \ll D$, which will achieve the same performance as $\theta^{(D)}$. The value of 'd' depends on the base model's capability & the task at hand.

Intrinsic Dimensionality

* Li et al [2018] refer to the minimum d where the model achieves within 90% of the full-parameter model performance, d_{90} as the intrinsic dimensionality of a task.
* Aghajanyan et. al [2021] investigate the intrinsic dimensionality of different NLP tasks & pre-trained models
  - the method of finding the intrinsic dimension proposed by Li et al (2018) is unaware of the layer-wise structure of the function parameterized by θ. They came up with a way of measuring intrinsic dimension taking into account this structure.
  - The reason they did this is because it is nearly impossible to calculate intrinsic dimension for LLMs. Even BERT based models would require 1 TB memory to store the projection matrix.

Structure Aware Intrinsic Dimension (SAID)

* Proposed in 2021 by Aghajanyan et. al
* Allocate one scalar λ_i per layer to learn layer-wise scaling:

$$\theta_i^D = \theta_{0,i}^D + \lambda_i P(\theta^{d-m})_i$$

This is to compensate for the additional dimension introduced by λ_i: for each layer we have 1 additional dimension so for m layers its d+m (in terms of space). So, to keep the dimension same they do d-m

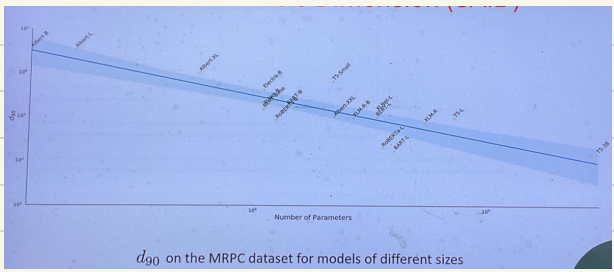
need more clarity

↳ so θ is learned layer-wise. So, the total no of params we learn is also 'm'.

where m is the no. of layers in the network

* However, storing the random matrices still requires a lot of extra space & is slow to train [Aghajanyan et. al 2021]
  especially with ← large models like GPT-3 & above.

* The tried to figure out what is the 'd' value that could give 90% of the performance of the full fine-tuned model. The no. of parameters in the model is in the x-axis & y-axis is the intrinsic dimensionality.



d_{90} on the MRPC dataset for models of different sizes

* So, we see that with less no of parameters (or small models) we have a high intrinsic dimensionality. As the no of parameters increases, there is a steady decline in the value of d_{90} that is necessary to learn a task.
* This indicates that more the no of params more world knowledge is present in these models & it is easier to train them in lower parameter space
* Here we compare intrinsic dimensionality (which is computationally challenging) & SAID.
  - SAID reduces – (i) memory requirement
    (ii) Training time
  - If we take the structure into account well, the dimensionality also comes down

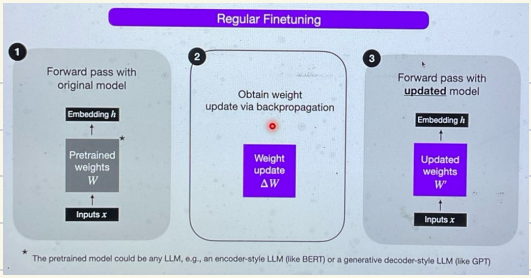
	SAID		ID	
Model	MRPC	QQP	MRPC	QQP
BERT-Base	1608	8030	1861	9295
BERT-Large	1037	1200	2493	1389
RoBERTa-Base	896	896	1000	1389
RoBERTa-Large	**207**	**774**	322	**774**

Estimated d_{90} intrinsic dimension for a set of sentence prediction tasks and common pre-trained models.

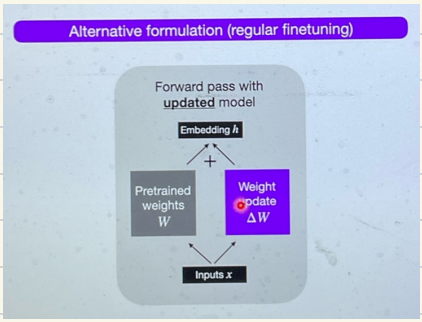
Low Rank Adaptation (LoRA)

* Below is a diagram of how we perform regular full fine-tuning of a model.

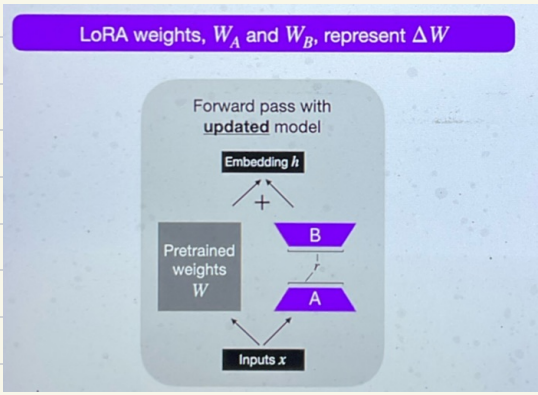
*) We have the pre-trained weights & we do a forward pass to generate the outputs. Then using the ground truth & loss func^n we compute the gradients via backpropagation & update the model weights.



*) This can be thought of as 2 parallel networks - once the input goes through the pre-trained weights. In the other the input goes through the update through the weights & then we can add them together to get our final output.



*) LoRA had 2 main contributions:
(1) They told us what weights to modify to reduce the intrinsic dimensionality (making use of the structure)
(2) How to represent the ΔW matrix. (Bottleneck structure similar to adapters)



- Instead of learning a low-rank factorization via a random matrix P, we can learn the projection matrix directly - if it is small enough
- Better use of network structure
- LoRA [Yu. et al 2022] learns 2 low-rank matrices A & B that are applied to self-attention weights.

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

↳ NOTE: there is no non-linearity. Just a down-projection followed by an up-projection.

* A proposed fine-tuning method where we only change the biases & not the weights.

* Here we see that for more complex datasets, for a small fraction of the trainable parameters, we end up getting even better accuracies than full fine-tuning.

Model&Method	# Trainable Parameters	WikiSQL Acc. (%)	MNLI-m Acc. (%)	SAMSum R1/R2/RL
GPT-3 (FT)	175,255.8M	**73.8**	89.5	52.0/28.0/44.5
GPT-3 (BitFit) *	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 (PreEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 (PreLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 ($\text{Adapter}^{\text{H}}$)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 ($\text{Adapter}^{\text{H}}$)	40.1M	73.2	**91.5**	53.2/29.0/45.1
GPT-3 (LoRA)	4.7M	73.4	**91.7**	**53.8/29.8/45.9**
GPT-3 (LoRA)	37.7M	**74.0**	**91.6**	53.4/29.2/45.1

Performance of different adaptation methods on GPT-3 175B [Hu et al., 2021]

* As mentioned earlier, the paper also talked about exactly what weights to update. They mentioned that we only need to worry about query, key, value & upper projection matrix.

	# of Trainable Parameters = 18M						
Weight Type	W_q	W_k	W_v	W_o	W_q, W_k	W_q, W_v	W_q, W_k, W_v, W_o
Rank r	8	8	8	8	4	4	2
WikiSQL (±0.5%)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI (±0.1%)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

Validation accuracy on WikiSQL and MultiNLI after applying LoRA to different types of attention weights in GPT-3, given the same number of trainable parameters [Hu et al., 2021]

Train only Q,K,V or O matrix

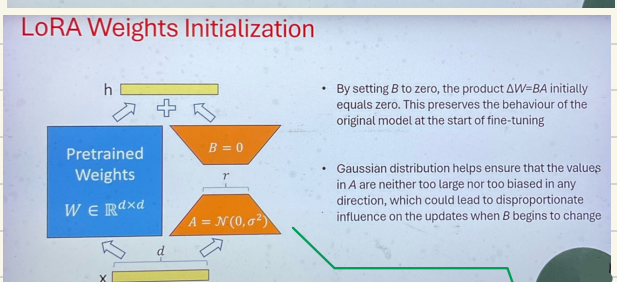
Train combination of 2 (with reduced rank to maintain dimensionality)

Train all 4 matrices

	Weight Type	$r=1$	$r=2$	$r=4$	$r=8$	$r=64$
WikiSQL(±0.5%)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI (±0.1%)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4

They also showed the effect of rank (of the update matrix) - as we go up the performance improves, saturates & eventually comes down.

↳ Here the rank can sort of be seen as the intrinsic dimensionality (although not exactly) in the sense that it gives us an idea of how many params we need to learn our task.



Here we want a ΔW that can be added to W. Since we only look at it as a projection, there is no need for a non-linearity to mimic the update of the projection. For adapters, this is not the case. Here, since the LoRA weights can be factorized & added to W, we can also fuse the result to W once we train A & B if we want (But no swap-out, swap-in after that)

Extensions of LoRA

* QLoRA [Dettmers et. al., 2023]
  - backprop gradients through 4-bit quantized model for reduced memory usage

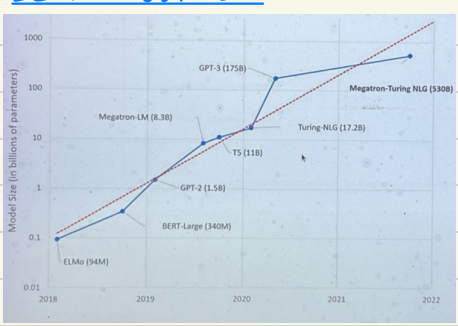
* Long LoRA [Chen et al., 2024]
- Sparse local attention to support longer context length during fine-tuning

* LoRA+ [Hayou et. al., 2024]
- different learning rates for the LoRA adapter matrices A & B improves fine-tuning

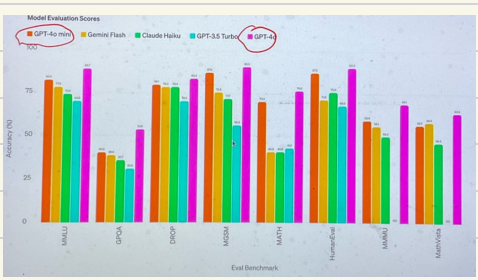
* DyLoRA [Valipour et al., 2023]
- selects rank w/o requiring multiple training runs.

Lecture 14.2 - Quantization, Pruning & Distillation

LLM Inference

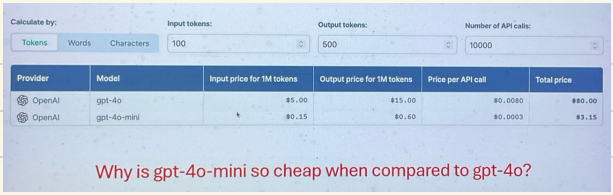


* Longer the model, larger the:
(i) GPU memory requirement
(ii) latency (esp. for critical or dynamic tasks like chatbots)
(iii) inference cost (cost of serving an LLM per user should be less than revenue generated)
(iv) environmental concerns - too much energy requirements



* Here is an evaluation benchmark that -
- OpenAI released GPT 4o-mini some time after 4o
- While worse compared to 4o 4o-mini is almost always better than 3.5-Turbo. However, the inference cost of 4o-mini is significantly lower than 3.5-Turbo & 4o.

*) Why is 4o-mini so cost effective compared to 4o? Or another question to ask is - How can we deploy LLMs in a cost effective manner while maintaining high performance?



Cost Effective Inference

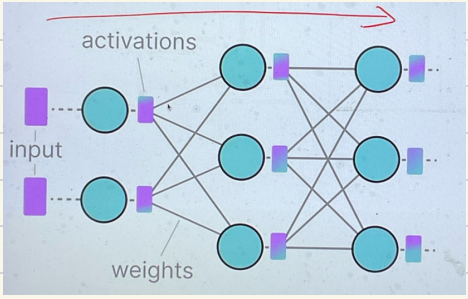
1) Model compression (lossy) - Like compressing a 70B parameter model to 10B with a reasonably low loss in performance. Includes techniques like:
   (i) Quantization: keep the model same, but reduce the no of bits
   (ii) Pruning: remove parts of a model while retaining performance
   (iii) Distillation: train a smaller model to imitate a larger model.
   ↳ least lossy but costliest

2) Efficient Engineering (lossless) - Do efficient engineering to achieve the same level of performance but by reducing cost through some optimizations, eg: fused kernels (like in flash Attention), hardware optimizations, etc.

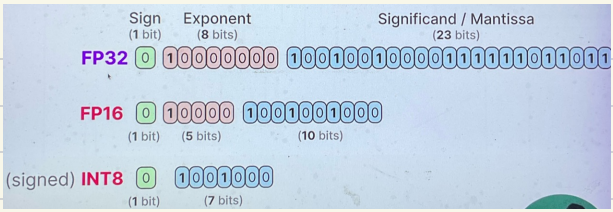
Quantization

Problems with LLMs

*) LLMs have billions of parameters which are expensive to store
*) During inference, activations are created as the product of input & weights, which similarly are expensive to store.
*) The goal is to represent billions of values as efficiently as possible.

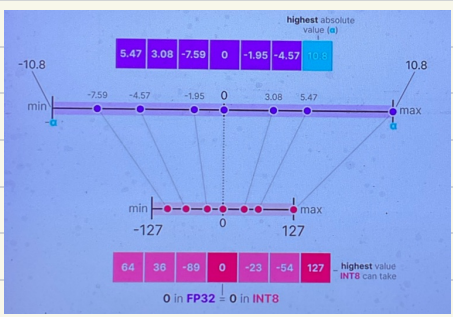


Numerical values representation



* FP32 - full precision (used in ML models in the past, gives enough precision for scientific applications)
* FP16 - GPUs started supporting this so models moved from FP32 to FP16. Precision & range reduce significantly but still works for ML applications w/o much drop in performance but significant improvement in computation
* INT8 - Even simpler representation with 1 sign bit & 7-bits for representing the numbers.

Quantizing FP32 to INT8



* Simplest method of quantization
* Identify the value in the FP32 vector with the highest magnitude (absolute) - α
* Set the max & min symmetrical from $-\alpha$ to $+\alpha$ with the range centered at 0.
* Map the max value α to 127 (highest in INT8) or -127 according to the sign & the rest accordingly as follows:

$$S = \frac{2^{b-1}}{\alpha}$$ (scale factor)

$b \rightarrow$ no of bits (8 for INT8)

$$x_{quantized} = round(S \cdot x)$$ (quantization)

Here,

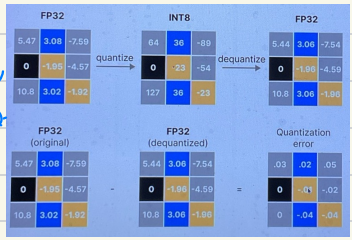
$$S = \frac{127}{10.8} = 11.76$$ → scale factor

$x_{quantized} = round(11.76 \cdot$ [5.47 | 3.08 | -7.59 | 0 | -1.95 | -4.57 | -10.8] $)$ → quantization

* α will change for different vectors. If we have a 2-D matrix with 10k rows then we will have 10k α values.

* Similarly, for dequantization we can reverse the process by dividing quantized vector with scale factor.

$$X_{dequantized} = \frac{64 \mid 36 \mid -89 \mid 0 \mid -23 \mid -54 \mid 127}{S}$$

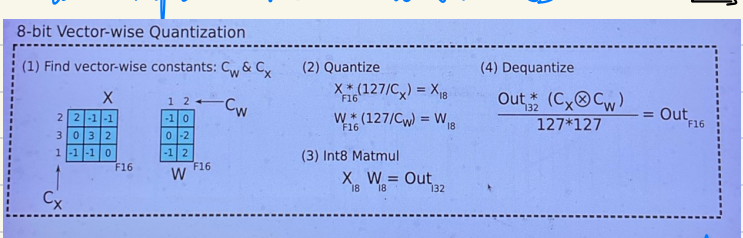


Q) How is quantization used in ML?
There are 2 ways:
(i) Post Training Quantization
(ii) Quantization Aware Training

Post Training Quantization (PTQ)

* Reduce the model size without altering the LLM architecture and without retraining.
* Weights & biases are constants. Easy to compute the scale factor(s).
* Model input & activations are variable. Use a calibration dataset to compute the scale factor(s).

→ Because some inputs might have a large max values & some small. We can use the calibration dataset to get a sense of the kind of values we can expect - the range, min, max, etc. From this we compute the scale factor.



* Typically during inference, we have multiple layers & multiple operations within each layer. Usually, we take one operation at a time, full or half precision (whatever the model was trained on) as input, quantize the weights & activations (or input), perform the operation & then de-quantize after that. These steps are repeated for every operation in each layer.

Why is the above efficient?
Because if we did the matmul using fp16 then the no of operations we can perform at a time is much lesser than if we did it at int8. So, we end up getting a huge speed up.

Technical Specifications		
	H100 SXM	H100 NVL
FP64	34 teraFLOPS	30 teraFLOPS
FP64 Tensor Core	67 teraFLOPS	60 teraFLOPS
FP32	67 teraFLOPS	60 teraFLOPS
TF32 Tensor Core*	989 teraFLOPS	835 teraFLOPS
BFLOAT16 Tensor Core*	1,979 teraFLOPS	1,671 teraFLOPS
FP16 Tensor Core*	1,979 teraFLOPS	1,671 teraFLOPS
FP8 Tensor Core*	3,958 teraFLOPS	3,341 teraFLOPS
INT8 Tensor Core*	3,958 TOPS	3,341 TOPS
GPU Memory	80GB	94GB
GPU Memory Bandwidth	3.35TB/s	3.9TB/s

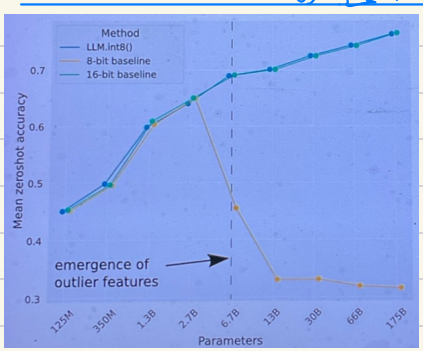
Datasheet

NVIDIA

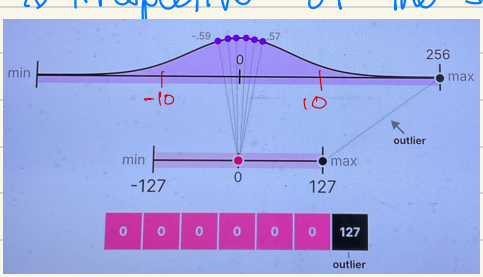
NVIDIA H100 Tensor Core GPU

Extraordinary performance, scalability, and security for every data center.

PTQ: LLM.int8() [Dettmers et al., 2022]



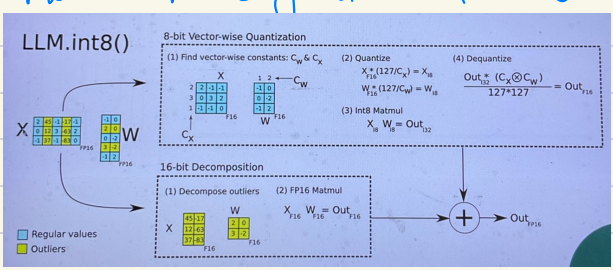
* Regular quantization retains performance at scales upto 2.7B parameters.
* Once, systematic outliers occur at a scale of 6.7B parameters, regular quantization methods fail.
* The paper identifies the reason behind the emerging pattern & proposes a way to fix it called LLM.int8()
* Irrespective of the scale, LLM.int8() maintains 16-bit accuracy.



* According to the paper, the no of outliers before 2.7B params is negligible but beyond that the numbers increase to such an extent that their effect becomes more pronounced & performance degrades.
* Say we have a vector in which most numbers are in the range -10 to 10 with a lot of them centered around

0. But there is 1 very large number in the vector (like 256 in the above image). Because of this, the α is decided by this absolute number (both -ve & +ve extremes), and all the other numbers end up getting mapped to the same point. So, we lose out on all the information because of this 1 number.

*) If most of the vectors are like this then quantization will not be very helpful. This the reason we see the dip in performance for models >2.7B parameters because most of the models beyond that size have such outliers.



Solution: A simple elegant approach

- Have a threshold. Any value beyond the threshold is an outlier.
- Keep the outliers out of the computation for scale factors.
- Do all the quantization, operation & dequantization on rest of the numbers & mask out the outliers.
- Use the normal full (or half) precision to do inferencing on these outliers & then add them back to the output of the previous step (the quantized part)

Quantization Aware Training

QLoRA [Dettmers et. al., 2023]

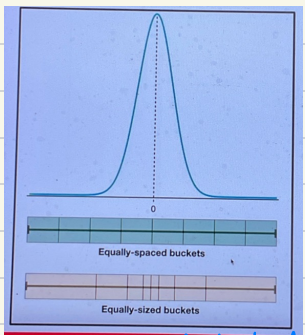
*) Most popular technique for quantization aware training.

*) Average memory requirement for fine-tuning a 65B model is >780GB

*) QLoRA reduces the memory requirement to <48 GB without degrading predictive performance. (Almost as good as fp16)

QLoRA has 4 components
- 4-bit Normal float (NF4) quantization
- Double quantization
- Paged Optimizers (Nvidia Hardware related optimization)
- LoRA

NF4 Quantization



- 4 bit quantization method
- N represents normal distribution
- Typically, in quantization, we define the min & max and divide it into buckets that are equally spaced. But this assumes that the array is uniformly distributed
- But if we assume that most the weights are normally distributed then this not the effective way of leveraging quantization.
- So, for this, rather than using equally spaced buckets, we use equally sized buckets so that each bucket has equal number of values.

Double Quantization

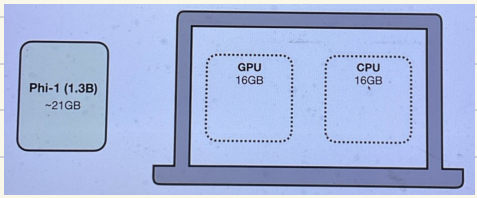
- Usually, when we quantize, the quantization constants are stored in FP32 (Because, more precise quantization constant means less quantization errors)
- Double Quantization is the process of quantizing the quantization constants for additional memory savings i.e., once we compute all the constants, we will divide them into blocks to further quantize them. (And it results in significant memory savings because there are a lot of these constants)

$$S = \frac{2^{b-1} - 1}{\alpha}$$

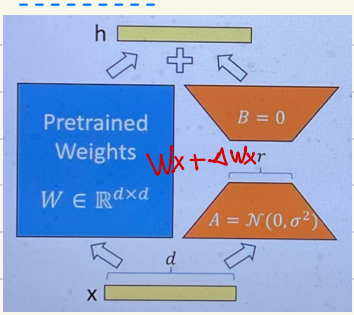
S → stored in FP32

Paged Optimizers

- Say we want to fine-tune a 1.3B param model & it ends up needing ~21GB of space.
- The GPU by itself doesn't have 21GB memory but the memory on the GPU + CPU combined is good enough.
- So, the way we do paging for CPU & main memory, some of the GPUs do it for GPU & CPU memory.
- QLoRA makes use of this hardware feature.



LoRA



- Same as discussed previously.
- We have 2 projection matrices (layers) that we use to represent the update in the weights of the original model.

$(r \ll d)$

$Y = Wx + xAB$

Or in a different way:

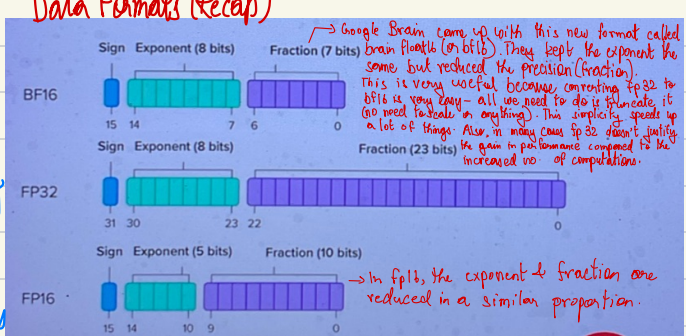
$Y = XW + sXL_1L_2$

scale factor (A) (B)

Mixed Precision Training

- A training paradigm where not all params are represented using the same format (i.e, for certain operations we use one format & a different one for other operations)

Data Formats (Recap)



- QLoRA also uses mixed precision.

So, $Y = XW + SXL_1L_2$

$$Y^{BF16} = X^{BF16}\,\text{doubleDequant}(c_1^{FP32}, c_2^{k\text{-bit}}, W^{NF4}) + X^{BF16} L_1^{BF16} L_2^{BF16}$$

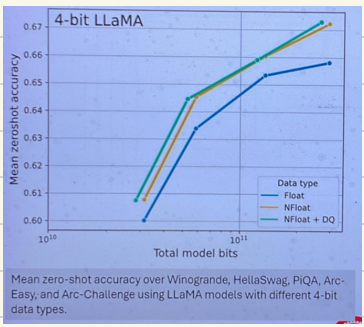
$$\text{doubleDequant}(c_1^{FP32}, c_2^{k\text{-bit}}, W^{k\text{-bit}}) = \text{dequant}(\text{dequant}(c_1^{FP32}, c_2^{k\text{-bit}}), W^{4\text{bit}}) = W^{BF16}$$

c_1^{FP32}: 2nd level quantization constants are in FP32

$c_2^{k\text{-bit}}$: 1st level quantized quantization constants

There is also the paging part where if we are fine tuning a 65B model (which needs ~> 63B) on a 32GB GPU, then there will be a point where some matrices will not be available & the hardware fetches that from the main memory & swaps out another one (which is not being used for a long time) & we will not run out of memory. This is also an essential part (not represented in the above equations)

Results



Mean zero-shot accuracy over Winogrande, HellaSwag, PiQA, Arc-Easy, and Arc-Challenge using LLaMA models with different 4-bit data types.

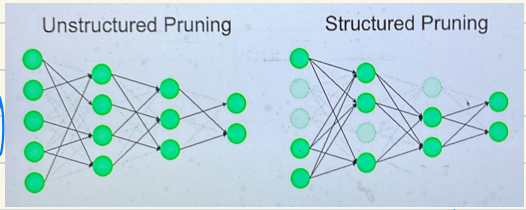
*) NF4 improves the bit-for-bit accuracy gains compared to regular 4bit floats

*) Double Quantization (DQ) only leads to minor gains, it allows for a more fine-grained control over the memory footprint.

LLaMA Size	Mean 5-shot MMLU Accuracy								
	7B		13B		33B		65B		Mean
Dataset	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	
BFloat16	38.4	45.6	47.2	50.6	57.7	60.5	61.8	62.5	53.0
Float4	37.2	44.0	47.3	50.0	55.9	58.5	61.3	63.3	52.2
NFloat4 + DQ	39.0	44.5	47.5	50.7	57.3	59.2	61.8	63.9	53.1

Mean 5-shot MMLU test accuracy for LLaMA models finetuned with adapters on Alpaca and FLAN v2 for different data types.

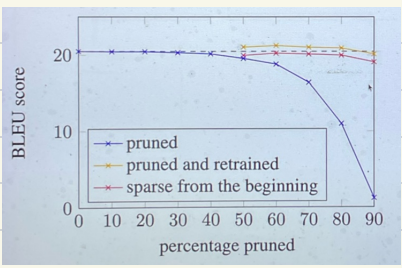
Pruning



*) Remove certain parts of the model
*) In unstructured pruning, we keep removing weights randomly with the hope that it doesn't affect performance too much.
*) In structured pruning, we take into account the structure of the model & prune certain structural components

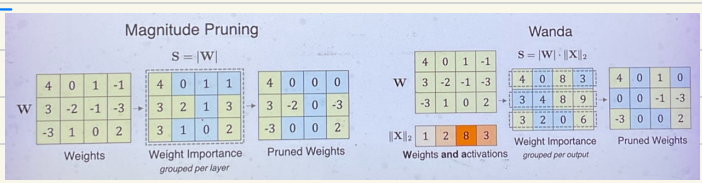
Magnitude Pruning [Han et. al., 2015, See et. al 2016]

*) Simplest form of unstructured pruning.
*) Prune the weights with smallest absolute values
*) They found that they can remove upto 40% of the weights with negligible performance loss.
*) By adding a retraining (or fine-tuning) phase after pruning, we can prune 80% w/o significant performance loss.



Wanda [Sun et. al., 2023]

*) The authors questioned pruning of only the weights & suggested that activations are also important & the product of weights & activations is what presents the real picture.
*) So, rather than pruning at weight levels, we should prune after the computation (of the weight & activation product)

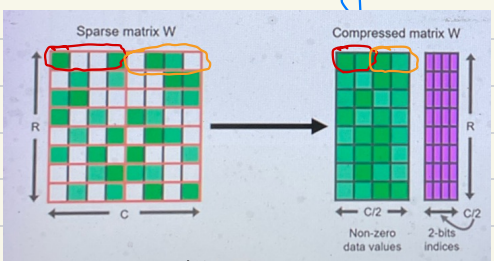


Issues with unstructured pruning

1) Can only work if the hardware we use supports it (because all we do when pruning is make the values 0, but if the

hardware doesn't utilize this information then it is the same as using the full matrix because we are using the space to store the 0's as well. So, it won't give us a speedup in numbers/compute)

Structured Pruning



*) NVIDIA A100 GPU supports fine-grained structured sparsity to its Tensor Cores

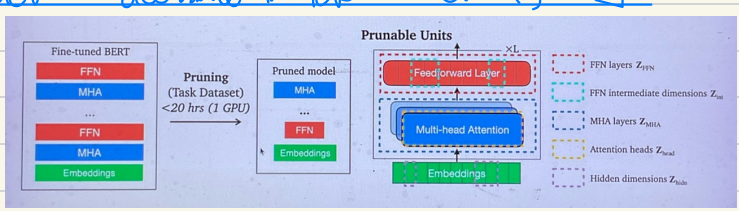
*) Sparse Tensor Cores accelerate a 2:4 sparsity pattern.

Input Operands	Accumulator	Dense TOPS	vs. FFMA	Sparse TOPS	vs. FFMA
FP32	FP32	19.5	-	-	-
TF32	FP32	156	8X	312	16X
FP16	FP32	312	16X	624	32X
BF16	FP32	312	16X	624	32X

we take a block of 4 & compress it into a block of 2 by ignoring 2. If we do this & represent the values with half the no of weights, and use 2-bit indices to store the non-zero indices then it gives a performance speedup.

Structured Pruning learns compact & accurate models [Xia et. al., 2023]

*) This paper said that almost any component of networks like BERT can be pruned including attention, FFN, embeddings, etc.



*) Like here they removed a MHA layer in the lower block & an FFN in the upper block.

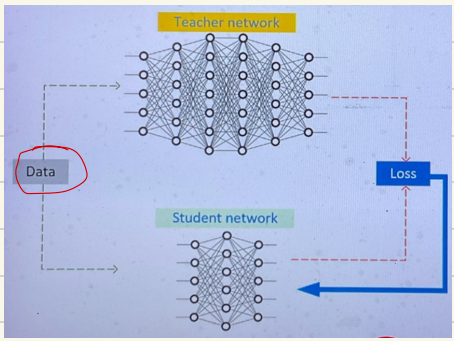
*) We can also remove certain heads in MHA or some hidden dimension in embeddings

*) If we do all this taking into account the structure of the network then we end up getting a performance speedup.

Distillation

*) Train a smaller model to imitate a bigger model.

Distillation [Hinton et. al., 2015]



*) One of the main issues with distillation is the need for data.

*) For quantizing a model (post-training), we don't need the data which the model was trained on or a proxy for that data.

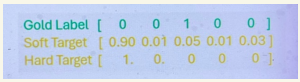
*) But for distillation we need the data or some representative of the data that looks like what the model was trained on. This is a big challenge in this era where models are often open-sourced but not the data which was used to train it)

*) So, people do task specific distillation. We can get task specific data & distill the model conditioned on the task.

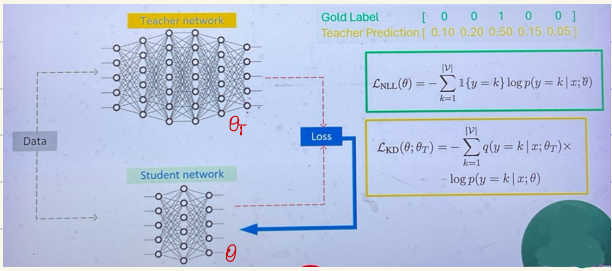
How does it work?

- Feed the input to the fully trained model & get its predictions (say $\hat{y}$)
- Give the same data to the student & get its output (say y')
- We want y' to be as close to $\hat{y}$ as possible.
- Both are not the GOLD standard (true labels) but we want y' to be closer to $\hat{y}$ as our objective.

*) There are different flavors of distillation like the different ways in which the loss can be computed, performing distribution matching in the intermediate layers as well (and not just the output layer) based on the structure, etc.



eg:



- say there are 5 outputs (like shown in the image) which give us the probabilities. The gold label is as shown in the vector.
- The typical loss could've been as shown in the box in green where we increase the log likelihood of the value which is 1.
- For knowledge distillation (KD), we have parameters θ_T (which is frozen) for the teacher model & parameters θ for the student model. So, we end up getting the value of every output with q (the yellow vector) and serve that as a reference distribution & compute cross-entropy. (There are advantages of taking q, rather than just the max value; more details in the paper)

↳ i.e.: the hard target

Pros:
- No restriction on student network structure
- Biggest potential gain in speed.

Cons:
- Needs training data
- Expensive to train student & get soft labels from the teacher (Because we need to do a lot of inference on the teacher model to get the data & that is an expensive process. There are techniques which talk about using an aggressively pruned version of the teacher as the student as a much better starting point to reduce the training time. Also in many cases like closed source models, the soft labels might not be available)

Sequence level Distillation [Kim et. al., 2016]

x) The above example was for a single output distillation. But we want to perform distillation for entire sequences (like in an autoregressive or similar manner)

$$\mathcal{L}_{KD}(\theta, \theta_T) = -\sum_{k=1}^{|V|} q(y=k|x;\theta_T) \times \log p(y=k|x;\theta)$$

1) Word-level KD

$$\mathcal{L}_{WORD-KD} = \sum_{j=1}^{T} \sum_{k=1}^{|V|} q(t_j=k|s, t_{<j}) \times \log p(t_j=k|s, t_{<j})$$

↳ do it one word at a time by fixing the other words

2. Sequence level KD

$$\mathcal{L}_{SEQ-KD} = -\sum_{t \in T} q(t|s) \log p(t|s)$$

→ Ideally we want to do it over the space of all possible sequences but that is impossible.

$$\approx -\sum_{t \in T} 1\{t=\hat{y}\} \log p(t|s)$$

→ Approximate it to what the model gives the best sequence for a beam search (or something similar). We set that as $\hat{y}$ and increase the log likelihood of this best sequence.

$$= -\log p(t=\hat{y}|s)$$

(There are other techniques like using KL Divergence instead of cross entropy loss etc.)

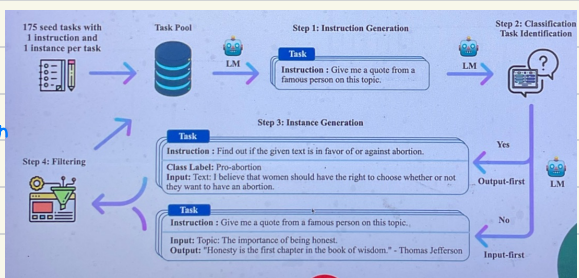
Self-Instruct [Wang et al., 2023]

a) Nowadays we have huge LLMs like Llama 405B or PaLM 540B. But this would take a huge amount of time, cost & energy

to give a response. So, why use it?

A) The idea is that these extremely overparameterized models can capture all the nuances in the training data and can model it better. Once it does that we can always distill it. It will generate outputs that will be so good that we can use these outputs to distill a smaller model which will be much better than training the smaller model from scratch.

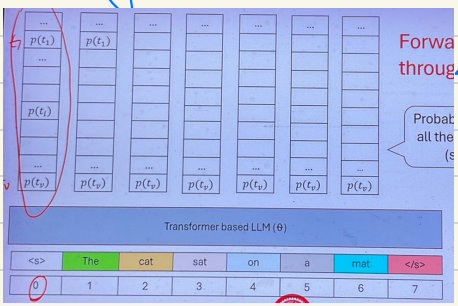
*) An early paper that explored this paradigm was Self-Instruct.



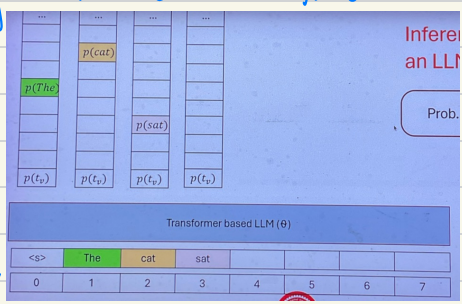
- Took 175 seed tasks with 1 instruction & 1 example (hand crafted tasks)
- Ask the LLM to generate new instructions from these.
- Check if the newly generated task is good enough (i.e., whether it makes sense & is different enough from input tasks)
- If it is good, we ask the LLM to generate an example with the class label & input given the instruction.
- By doing this, we can generate a lot of data.
- A base LLM (pre-trained & not-instruction fine-tuned) can still do this work with few shot learning
- So, we end up getting a lot of information from the model itself on how to instruct fine-tune it (Hence, the name self-instruct)
- Now, we can use this instruct fine-tuned dataset to instruct fine-tune the model itself.

Lecture 15.2 - Efficient LLM Decoding - I

Q) Why focus on efficient inference & not on training?



* During training (forward pass) we get a probability distribution over all the tokens at every step (simultaneously). The learning objective is to maximize the probability of the next token.



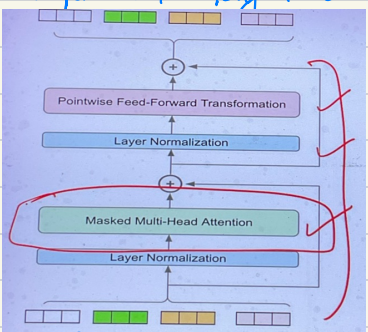
* During inference (forward pass), lets say we want to complete the sentence - The cat sat . . .
We run the forward pass & generate the distribution at each step. We sample the token from the distribution, append it to the sequence & run the forward pass again on this new sequence.
Here, we need to run the forward pass repeatedly (i.e., one pass to generate one token) & one after the other.

Training	Inference
* Single forward pass and all outputs are computed in parallel -> remember "teacher forcing"	* One forward pass for each token * Very expensive * Need techniques to make it workable

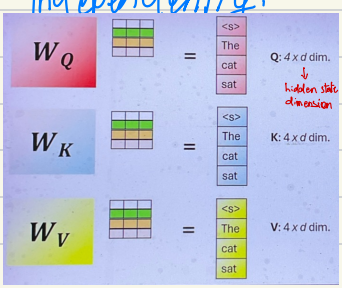
* In the above example, we run the fwd pass 4 times to generate 4 tokens.

*) This is not feasible at production scale.

Let us revisit the forward pass through & see if we can optimize. We focus on the attention layer as that is the bottleneck.

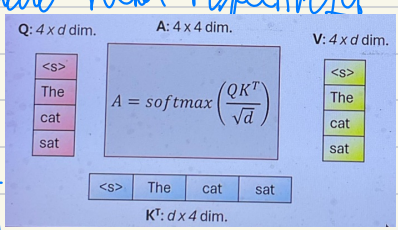


*) Multihead attention is the step where the tokens interact with each other. Every where else, they are processed independently.

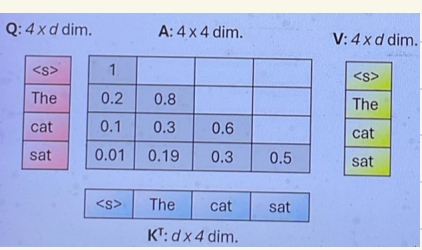


*) In multihead attention, we start with 3 matrices (W_Q, W_K, W_V) which when multiplied with the embeddings gives us the query, key & value vector respectively.

*) We compute the attention matrix which tells us how much a token depends on its previous tokens.

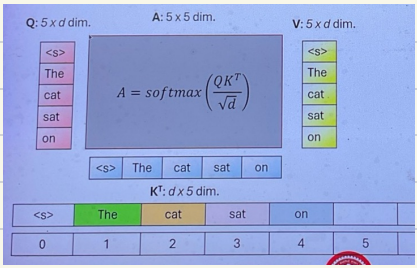


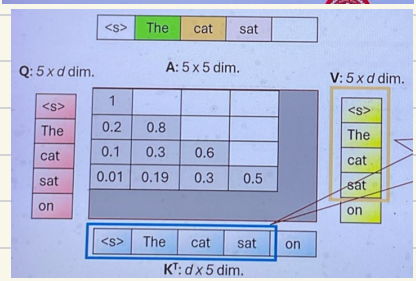
*) To do this, we multiply the query vector with the keys of the previous tokens to generate the attention scores. These scores are then passed through a softmax function which gives the importance weight of each token for the token being currently generated. These scores are multiplied with each of the respective value vectors & summed up to give the final output.



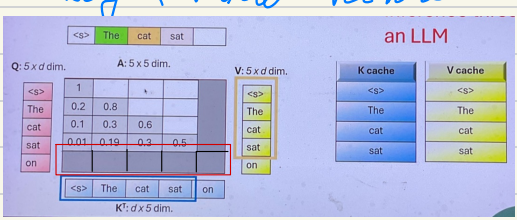
*) Finally, we pass the embedding of the last layer through a classifier to get a probability distribution over the tokens. Then we pick the token with the highest probability

(or sample it) as our next token.

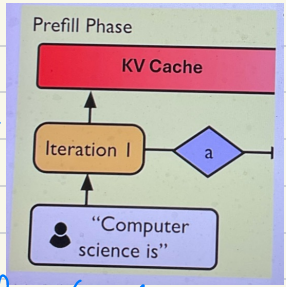




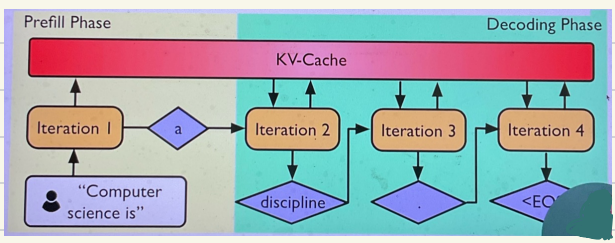
- When doing the computations with the new token 'on' do we need to compute everything again or can we leverage what we've already calculated?
  - We have already computed the attention matrix (partially)
  - The output embeddings have also been computed (or given) for the 1st 3 tokens
  - The keys & values have also been computed earlier
  - We don't need the queries because we only need to calculate the output for the current step/input ('on' is the current input in the above image)
  - We can cache the values of the key & value vectors for each of the tokens from the previous steps.
  - To compute the attention weights, we compute the query vector for the last generated token ('on' in this case), calculate the similarity score with the keys from the key cache, calculate the similarity score with itself (and store the key of itself in the cache), convert the scores to probabilities using softmax, compute the value vector (and store in the cache) and finally, take the weighted sum $\sum_i p_i v_i$ using the cached value vectors to calculate the final output embedding. This is then passed through a linear layer which gives a probability distribution over the vocabulary for generating the next token.
  - We discard the query & the process continues for the next tokens.



2 stages of LLM Inference



* 1st forward pass (Pre fill step): Highly Parallel
  - The entire prompt is embedded & encoded - High latency
  - Multihead attention computes the keys & values (KV-cache)
  - Large matrix multiplication, high usage of hardware accelerator.



* Remaining forward passes (Output generation): Sequential
  - The answer is generated one token at a time - Low latency per step
  - Each generated token is appended to the previous input
  - The process is repeated until the stopping criteria is met (eg: max length or EOS)
  - Low usage of hardware accelerator.

Memory Usage of KV Cache

$$2 * \text{precision} * N_{layers} * d_{model} * \text{seqlen} * \text{batch}$$

2: 2 matrices for K & V
precision: bytes per parameter (eg 4 for fp32)
N_{layers}: layers in the model (i.e., no of transformer blocks)
d_{model}: dimension of embeddings / hidden state dimension
seqlen: length of the context in tokens
batch: batch size

Eg: OPT-13B

2 x 2 bytes (fp16) x 40 layers x 5120 dim x 2048 tokens x 10 (per batch)

KV cache = 17GB

Model size = 26 GB

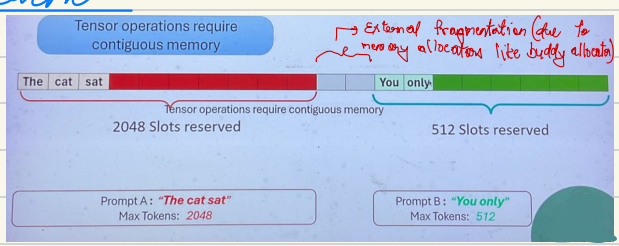
So, on a 40GB A100 GPU:

- 65% (26 GB) is used by model parameters
- 30% (12 GB) available for KV cache
- Expected throughput ~ 8 batch size of 2048 tokens

But this is still inefficient. Can we manage the KV cache better? Here we assume that there are 2048 tokens in each sequence but that is not true. So how can we optimize further?

Memory Management of KV cache

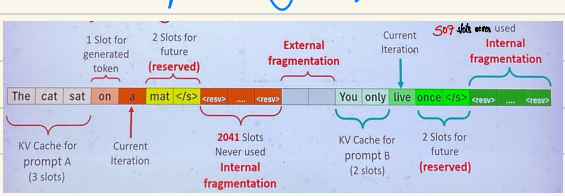
*) KV cache needs contiguous memory in the GPU because most of the tensor operations (like mat mul) requires it to be in contiguous memory.



*) In this example, we have 2048 slots reserved for the 1st prompt & 512 slots for the 2nd prompt.

*) There will be some external fragmentation depending upon the memory allocator we're using.

*) But beyond that, there is also a lot of token slots that might not end up getting used because the sequence might end much



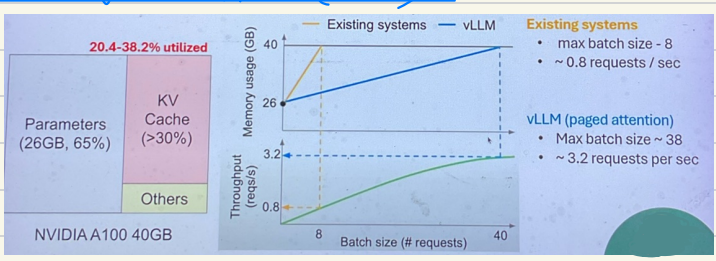
earlier (as shown in the diagram 20th & 50th tokens slots are never used)

* Chunk Pre-allocation Scheme
  - KV cache stored in contiguous memory
  - Chunks of memory allocated statically, based on max tokens.
  - Actual input or eventual output length ignored while allocating memory.

Results in 3 types of memory wastes -
- Reserved slots for future tokens
- Internal fragmentation due to over-provisioning for max sequence length
- External fragmentation from the memory allocator

Memory layout of 13B-OPT model on A100 (40GB)

* So for a 13B OPT model on a 40GB NVIDIA A100, aside from the model parameters (which need to be loaded all the time), we get roughly 30% for the KV cache & with the way we currently manage the memory, we max out at 8 batches



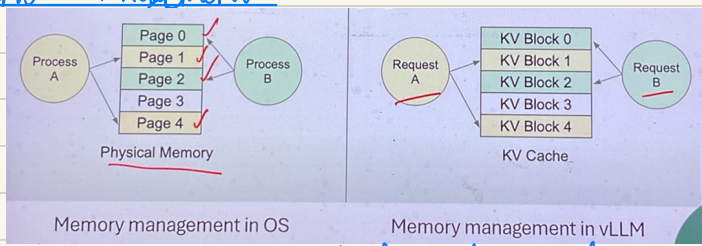
Q) Can we use some OS concepts to efficiently manage our KV cache?

vLLMs introduced a concept called paged attention which is able to increase the batch size upto 40 using OS concepts like paging to manage the KV cache.

vLLM: Efficient KV cache management

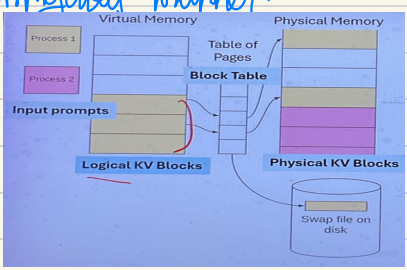
*) vLLMs are inspired by virtual memory & paging.

*) In OS, multiple processes are running & each one requires its own memory. It is not necessary that this memory will be allocated in contiguous chunks to these processes.



*) Analogous to this, if we have multiple request prompts then the KV cache (which will be divided into blocks) will not be allocated to each prompt in a contiguous manner.

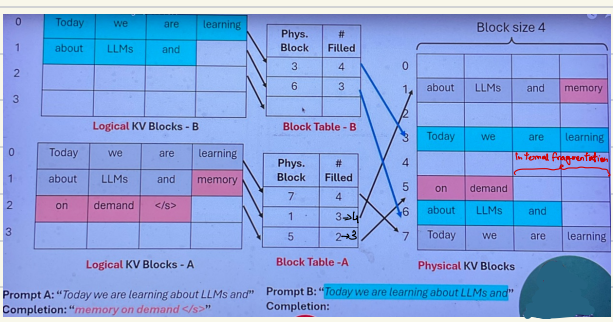
- Processes are seen as incoming requests (input to the model)
- Virtual memory as Logical KV blocks
- Physical memory as Physical KV blocks
- Page Table as Block Table



KV Blocks

*) Physical KV blocks are where the actual GPU memory is allocated.

*) For each request, we have a corresponding logical KV block and a Block Table which maps the logical KV blocks to physical KV blocks

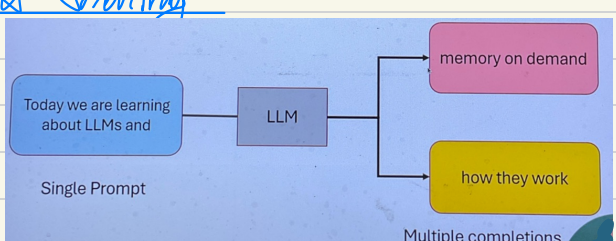


*) In the example A, we have the input prompt "Today we are learning about LLMs and."

*) The logical KV blocks are assigned to the tokens in a contiguous fashion. For input A, the 0th logical block is mapped to the 7th physical block.

* We also keep a track of how many slots are filled in a given block. (Here, all 4 slots are filled in block 0 & only 3 slots are filled in block 1 (when we only have the input)
* Once we generate the next token, we check if there is a slot available in the logical KV block, store it (the K & V vectors) & increment count of the # of slots filled.
* When the next token comes (say 'on' after 'memory' in the above example), and the current block is completely filled, then a new block from the physical KV will be allocated on demand (which may not necessarily be contiguous).
* The corresponding entry is added to the block table & store the token (vectors) there & then set the # filled count for this block to 1 & the process continues.
* Compared to the naive way of managing the KV cache, this method also suffers from internal fragmentation. However, the fragmentation is upper bounded by the block size.
* If we get another request, it will have it's own logical KV block, block table & the corresponding memory will be allocated on the physical KV block.

Dynamic block mapping enables sharing

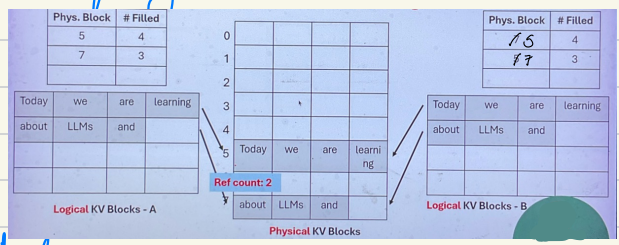


* In addition to efficient memory management, dynamic block mapping also enables block sharing.
* We often come across scenarios where we want multiple completions for the same prompt. (eg: the ChatGPT interface where it gives us two simultaneous outputs for the same prompt & asks us to choose which one we prefer; extensively used for RLHF tasks. Another example

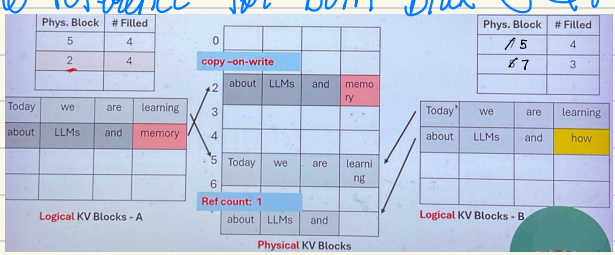
could be Beam Search)

Sharing KV blocks in parallel sampling

* We will have seperate logical KV blocks for the 2 requests.
* The physical KV blocks are shared so only block 5 & 7 are actually populated.
* When the next token is generated, both sequences will generate a different token. So how do we manage that?



In order to manage this, the KV cache needs to know how many requests are actually pointing to each block. So it needs a reference count. The reference for both block 5 & 7 is 2 initially. When the next token is generated for a request (say 'memory' for request A), then we check the reference count of the current block (to see if it is being shared). If it is more than 1 (i.e., shared by multiple), then we reduce the reference count by 1 & copy it to another block, write the generated coord (vectors) to this new block & update the block table with the new block address & increment #filled by 1 (NOTICE: Block 5 still has a reference count of 2 & shared). Now when the other request generates a token (like 'how' for request B), we again check the reference count. Since, it is 1 (i.e not shared (anymore)), we write the new word directly to the block itself & update #filled for that block in the block table.

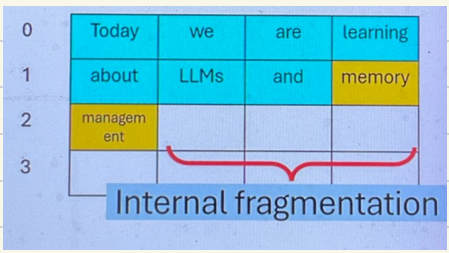


Q) So, if we have a big prompt (which spans like 5-6 blocks) then how many blocks will end up getting copied for multiple requests?

Only 1 i.e., the last one. Rest of the blocks can & will be shared. The next blocks will be allocated seperately for each requests.

Memory efficiency of vLLMs

*) The naive implementation of KV cache suffered from external fragmentation (due to memory allocator) as well as internal fragmentation (depending on the max_tokens specified, we over reserved the space).



*) Compared to this, in vLLMs we have no slots b/c 2 blocks so there is no external fragmentation. Also, the space that we reserve for future tokens is bounded by the block size (which is a lot lesser compared to max_tokens in general), because we allocate only 1 block at a time to any request, so the internal fragmentation is a lot lesser compared to the naive implementation (happens only in the last block)

Sequence : O(100) or O(1000) tokens
Block size : 16 or 32 tokens

*) On average, the wasted space <4% of the KV cache (compared to earlier where the utilization was only 20-40%)

Paged Attention

REMEMBER: we allocated memory in contiguous slots because

most tensor operations require contiguous memories (Like, at times in PyTorch, when we perform an operation on 2 concatenated tensors, it gives us an error that we need to perform .contiguous())

Q) So what do we need to do here to compute the next output token for a query token? How do we compute attention softmax across fragmented memory? (because in actual physical memory, they are only stored in blocks of 16 or 32 depending on our choice)

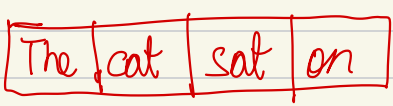
Idea → Paged Attention (Conceptually same as flash Attention)

$$\text{softmax}([A_1, A_2]) = [\alpha\,\text{softmax}(A_1), \beta\,\text{softmax}(A_2)]$$

$$\text{softmax}([A_1, A_2])\begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \alpha\,\text{softmax}(A_1) * V_1 + \beta\,\text{softmax}(A_2) * V_2$$

The higher level concept is the same as flash Attention which is that we can decompose attention softmax calculation to block level & calculate attention over blocks & recompute the final attention values

So, if we have a sentence:



Attention scores can be calculated independently: $q.k_1$, $q.k_2$, $q.k_3$, $q.k_4$

In fact a step further: $e^{q.k_1}v_1$, $e^{q.k_2}v_2$, $e^{q.k_3}v_3$, $e^{q.k_4}v_4$ can all be calculated.

So, we can calculate the numerators (for each block) for the softmax block wise & the sum seperately (i.e., the denominator for each block) $\sum_{i=1}^{2} e^{q.k_i}$, $\sum_{i=3}^{4} e^{q.k_i}$

Finally, we can aggregate

So, what will α & β be (for this example)?

$$\alpha = \frac{\sum_{i=1}^{2} e^{q_k i}}{\sum_{i=1}^{4} e^{q_k j}}$$

→ because in blockwise softmax, this must've already been divided in the block so we multiply by this & divide by total sum to calculate actual softmax score.

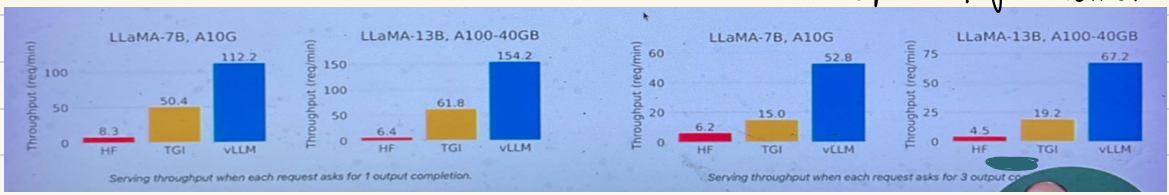
Similarly, $$\beta = \frac{\sum_{i=3}^{4} e^{q_k i}}{\sum_{i=1}^{4} e^{q_k j}}$$

Q) How does vLLM & Paged Attention result in efficient inference?

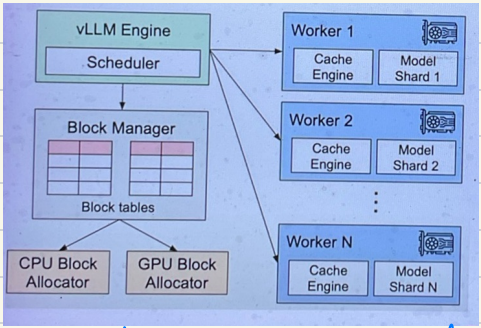
↳ Reduce memory fragmentation with paging
↳ further reduce memory usage with sharing

Results

* Upto 24x higher throughput than Hugging face (HF) - .generate()
* Upto 3.5x higher throughput than Text Generation Inference (TGI)
  ↳ by PyTorch
  ↳ Later incorporated Paged attention



System Architecture & Implementation



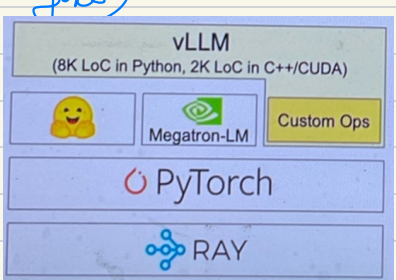
*) So, vLLM involves 2 concepts - Paging & Paged Attention

*) vLLM authors (from Stanford) started a company & launched it as a product called vLLM which is very popular as an end-to-end LLM serving engine. (pip install vllm; start server; get url; start sending requests)

*) Heavily used in research because loading an LLM individually on GPUs is very tedious process.

*) The end-to-end llm serving engine has 3 components
- A frontend
- A distributed model executor (models these days can't run on single GPU. We need multiple workers)
- A scheduler (schedules the different jobs)

A centralized engine manages the block tables.



*) Mostly written in PyTorch & some in CUDA/C++ (to implement Paged Attention).
- Megatron is to split a big model into multiple GPUs
- RAY - distributed parallel computing framework

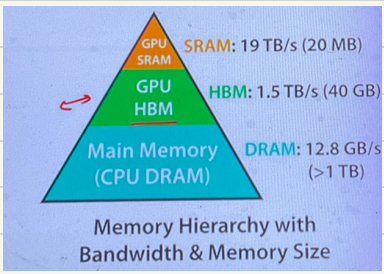
Q) Till now we have only looked at efficient LLM inferencing from the point-of-view of efficient memory management. Can we speed up attention computation? (Like Flash Attention) Also, can we generate multiple tokens at a time?

Ans : (i) Flash Decoding (ii) Speculative Decoding

Lecture 15.2 - Efficient LLM Decoding - II

Flash Attention - Recap

*) "I/O aware" implementation of attention



1. Matmul_op (Q, K)
   a) Read Q,K to SRAM (read op)
   b) Compute matmul A = QxK (compute op)
   c) Write A to HBM (write op)

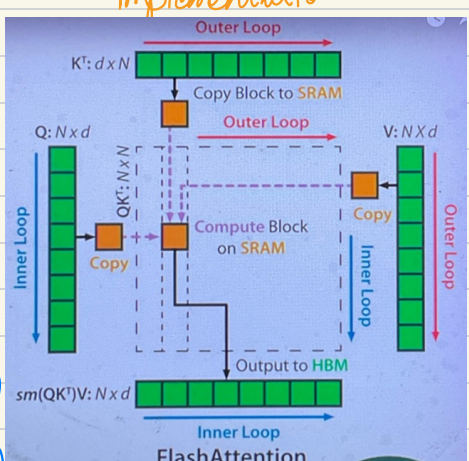
2. Mask_op
   a) Read A to SRAM (read op)
   b) Mask A to A' (compute op)
   c) Write A' to HBM (write op)

3. Softmax_op
   a) Read A' to SRAM (read op)
   b) Softmax A' to A'' (compute op)
   c) Write A'' to HBM (write op)

Standard attention implementation

Flash Attention
1. Read Q,K to SRAM
2. Compute A = QxK
3. Mask A to A'
4. Softmax A' to A''
5. Write A'' to HBM

I/O aware attention implementation



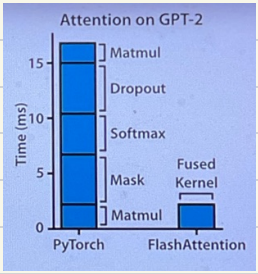
*) We use a fused kernel to avoid multiple read/writes b/w HBM & SRAM

*) Tiling - decompose large softmax into smaller ones by decoding

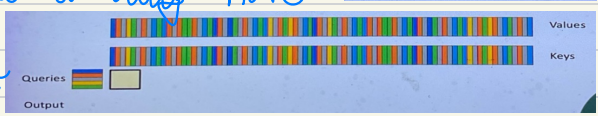
$$\text{softmax}([A_1, A_2]) = [\alpha\,\text{softmax}(A_1), \beta\,\text{softmax}(A_2)]$$

$$\text{softmax}([A_1, A_2])\begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \alpha\,\text{softmax}(A_1) * V_1 + \beta\,\text{softmax}(A_2) * V_2$$

*) 2-4x faster, 10-20x memory reduction
*) Flash Attention for training - parallelizes across batch size & query length dimension to avoid memory bandwidth bottleneck.



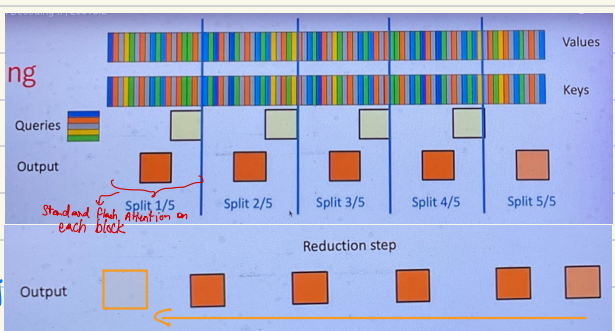
*) But now during decoding, we usually have only a single query/request but the latest models support huge context lengths over 32k (upto even millions).



*) Lets say the task we are performing is content-grounded QnA. Content itself is huge so we have a lot of keys & values but only 1 query. So, we need to modify this a bit.

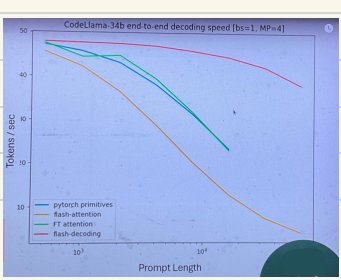
Flash Decoding

*) We split the keys & values into different blocks.
*) Use standard flash Attention to compute softmax over each split (i.e., we apply flash Attention on each split seperately and in parallel)
*) Then using the distributional nature of softmax, we can combine (or reduce) the individual outputs



Results (Benchmark on Code Llama-34B)

*) PyTorch - Attention using pure PyTorch primitives (no Flash Attention)
*) Flash Attention v2
*) FasterTransformer - Uses faster Transformer attention kernel
*) Flash Decoding



Flash Decoding - 8x speedups in decoding speed for very large sequences.

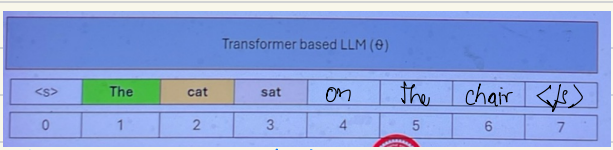
Problem - Generation is still sequential

Q) What if we can generate multiple tokens in one iteration?

(NOTE: This is not like Beam Search where we generate multiple tokens at the same timestep. We want to generate multiple tokens at different timesteps)

Inference through an LLM

Input Prompt: "The cat sat"

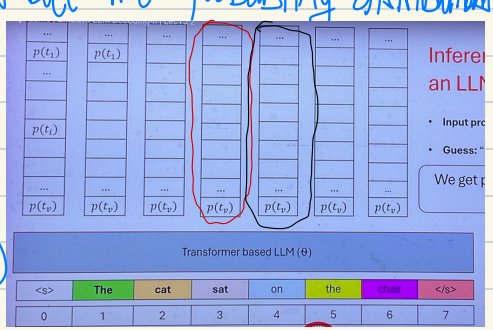


Say we make a guess (more on how to guess later)

Guess: "on the chair </s>"

Q) Can we use this guess to speedup inference?

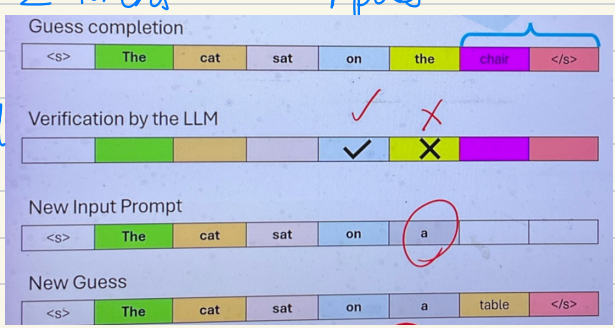
During training we are able to process the entire sequence in one go. Similarly after guessing, we can forward this entire sequence in one go & get all the probability distributions



K) The distribution circled in red is the distribution for the next token 'on' (after the prompt "The cat sat"). The distribution circled in black is the distribution over the next token from which we sample 'the' (provided the previous token is 'on')

* Say for the 1st token after the prompt we get 'on' as the token with max probability. We compare it with our guess & it's a match so we accept it.
* Say for the 2nd token conditioned on 'the cat sat on', the token 'a' has the highest probability. But we have 'the' so we can't accept it.
* Till now we were effectively able to generate only 1 token via our guess & failed in the next step. Do we stop here or can we do something else?

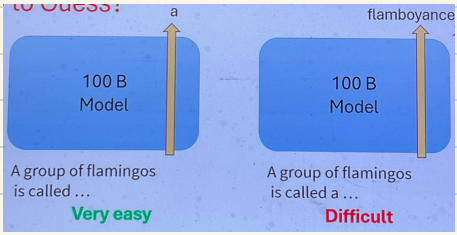
- We can reject the guess but replace 'the' with 'a' i.e., the token selected by the model. So, now we were able to generate 2 tokens in 1 pass.
- But we can't go any further because the future tokens (like chair) were generated conditioned on the token 'the' which has been rejected.
- So we start with a new prompt 'The cat sat on a' and guess again.



- Say we guess table but the next token after the forward pass is 'mat' (one with highest probability). So, there is no match but we still get 1 token i.e., mat. So, the method is atleast as good as regular inferencing where we have a lower bound of 1 i.e., we get atleast 1 token in each iteration. This method is called Speculative Decoding.

Q How do we guess?

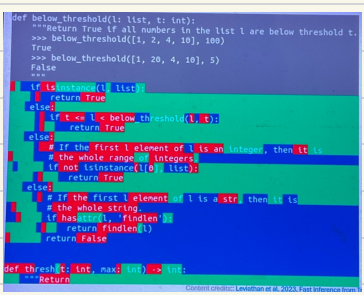
- Look at the output here for the prompt 'A group of flamingos is called a'.
- The first token 'a' is very easy to predict compared to the 2nd word.



Q) Do we need a 100B model to predict 'a'?
No. Maybe we can use a smaller (say 1B) model to create a draft first. Say it predicts the tokens 'a flock'. 'flock' is wrong but 'a' is correct. So, we can use the larger model to verify & correct our output.

Example from Human Eval Dataset

*) Here is an example from the speculative decoding paper.
*) Only the tokens shown in red are generated by the bigger 'target' model.
*) The smaller 'draft' model was able to guess the rest.



Speculative Sampling

*) Greedy Decoding
- Target model selection: Token with max probability
- Easy to verify with the "proposal" generated by the "draft model"

*) But what about sampling by varying top-k, top-p or temperature?

M_p = draft model (Llama-2-7b-chat-hf)
M_q = target model (Llama-2-70b-chat-hf)

pf = prefix, K = 5 tokens → (no of tokens we want to generate)
↳ $(x_1 \cdots x_{t-1})$

Say we have a distribution 'p' for the next token & during verification we also have the distribution 'q' from the target model. We sampled:

$$\tilde{x} \sim p(x \mid x_1, x_2 \cdots x_{t-1})$$

Q) We want to see if $\tilde{x}$ is a good sample from $q(x \mid x_1, x_2 \cdots x_{t-1})$
How do we do this?

A) Rejection Sampling

$p_1(x) = M_p(pf)$ — Sample → x_1
$p_2(x) = M_p(pf, x_1)$ → x_2
$\cdots$
$p_5(x) = M_p(pf, x_1, x_2, x_3, x_4)$ → x_5

Run the draft model for K steps

$q_1(x), q_2(x), q_3(x), q_4(x), q_5(x), q_6(x)$
$= M_q(pf, x_1, x_2, x_3, x_4, x_5)$

Run the target model once (i.e., in parallel for all tokens)

eg: Lets see this sentence completion - 'dogs love chasing after cars'

	Token	x1	x2	x3	x4	x5
		dogs	love	chasing	after	cars
Draft Model	p(x)	0.8	0.7	0.9	0.8	0.7
Target Model	q(x)	0.9	0.8	0.8	0.3	0.8

Here we have a distribut^n over the entire vocabulary for all tokens i.e., $x_1, x_2 \cdots x_5$

We need to decide if we should accept 'dogs' or reject it.
We accept x with the probability $q(x)/p(x)$

$$\text{Accept}(x) = \frac{q(x)}{p(x)}$$

So, dogs : $\frac{0.9}{0.2} > 1 \Rightarrow$ So accept (Because probability is 1)

love : $\frac{0.8}{0.7} > 1 \Rightarrow$ accept

chasing = $\frac{0.8}{9} \Rightarrow$ Accept with probability $\frac{0.8}{0.9}$

In summary,

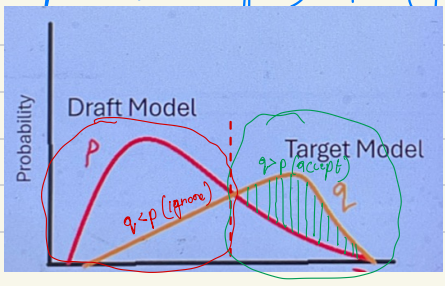
Case 1: If $q(x) > p(x)$ accept
Case 2: If $q(x) < p(x)$, then accept with probability $\frac{q(x)}{p(x)}$

Say we accepted tokens x_1, x_2, x_3 but rejected x_4. So what now?
We have the distribution $q_4(x)$ from the target model. We could sample from here.
But the paper recommends don't sample from $q_1(x)$ directly

Adjusted distribution: $(q(x) - p(x))_+$

sample from here

(+ indicates only from positive part)



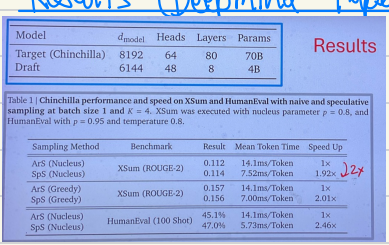
For all the cases where $q_i < p_i$, we ignore it. We only focus on the tokens where $q_i > p_i$ where we calculate the difference b/w the 2 probabilities & sample with probabilities proportional to that.

↳ Paper proposed simultaneously by Deepmind & Google Research

Wall Time Speedup (Google Paper)

*) Illustration on encoder-decoder (T5) model.

Results (Deepmind Paper)



Model	d_{model}	Heads	Layers	Params
Target (Chinchilla)	8192	64	80	70B
Draft	6144	48	8	4B

Results

Table 1 | **Chinchilla performance and speed on XSum and HumanEval with naive and speculative sampling at batch size 1 and** $K = 4$. XSum was executed with nucleus parameter $p = 0.8$, and HumanEval with $p = 0.95$ and temperature 0.8.

Sampling Method	Benchmark	Result	Mean Token Time	Speed Up
ArS (Nucleus)	XSum (ROUGE-2)	0.112	14.1ms/Token	1×
SpS (Nucleus)		0.114	7.52ms/Token	1.92×
ArS (Greedy)	XSum (ROUGE-2)	0.157	14.1ms/Token	1×
SpS (Greedy)		0.156	7.00ms/Token	2.01×
ArS (Nucleus)	HumanEval (100 Shot)	45.1%	14.1ms/Token	1×
SpS (Nucleus)		47.0%	5.73ms/Token	2.46×

*) Showed upto 2x speedup on different datasets

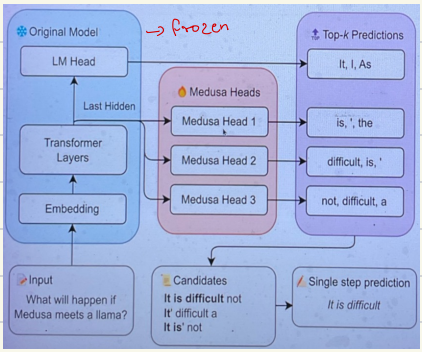
ArS → Standard Nucleus Sampling

SpS → Speculative Decoding

In this method of guessing, we have a smaller draft model which we use to make a guess. But we might not always have that. Also, we used Llama 70b & Llama 7b.
But is 7b small enough?
Also, it is not easy to host 2 models on the same system.
Can we do something else?

Medusa

REMEMBER PEFT: Efficient & fast fine-tuning.



* We train multiple LM heads to generate next-next token (which we can perform via PEFT) after freezing the backbone.
* Take the Cartesian product (i.e., all possible combinations) to create multiple possible candidate sequences
  - With top-k = 4, and 3 heads, we get $4^{(3+1)} = 256$ candidates

Q) Now we need to verify these candidates. But compared to earlier where we had only 1 candidate, we now have multiple. So how do we do this (in a single pass)?

A) Tree Attention – verify all the candidates & select the best in one iteration (i.e., in parallel)

* Accept the largest subsequence above a threshold probability

How to train multiple LM heads?

* Each Medusa head is as a single layer of FFN, augmented with a residual connection.
* Keep the backbone architecture frozen & train heads using PEFT.
* Can use the same corpus that trained the original model.
* On Vicuna-7B, Medusa head 1 gets
  - top-1 accuracy rate of approx. 60%.
  - top-5 accuracy rate of ~ 80% (hence we use top-k approach)

because we are also working with multiple tokens so $(0.6)^4$ is a very low probability overall & has high chance of rejection.

Tree Attention

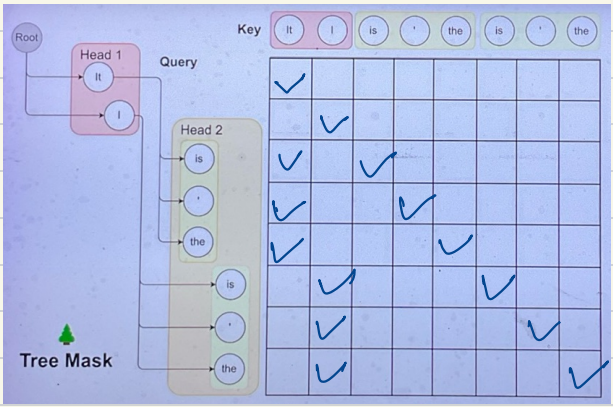
- Say 1st head gives us 2 tokens & 2nd head gives 3 tokens

Head 1 : It I

Head 2 : Is ' the

No. of possible completions = 2x3 = 6 +2 = 8

Because it is also possible to choose only the tokens of the 1st head - It & I so we have 2 more.



- If we want to look at a particular path in the tree like 'it is' then the token 'is' should only attend to itself & 'It' & not 'I'. So, we create an attention mask according to this path that is represented at each step.
- Each one of the 8 nodes gives us a potential subsequence (thus the 8x8 matrix) & we design the attention mask in such a way that we ensure that the tokens attend appropriately to the keys.
- The positional indices for positional encodings are adjusted in line with this structure.

Pruning the tree

* The cartesian product takes all possible combinations which means the total no of combinations we need to verify will be huge (the no of nodes in the tree = k^H k = top-k value, H = no of heads)

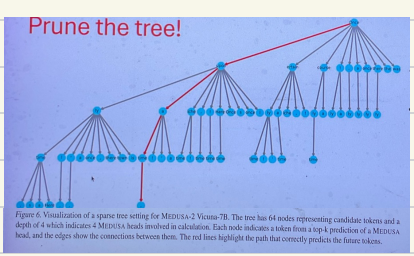
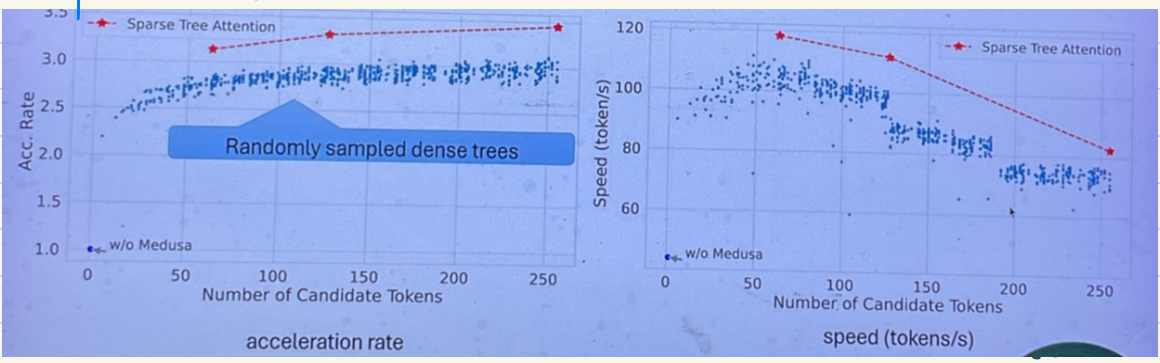


Figure 6. Visualization of a sparse tree setting for MEDUSA-2 Vicuna-7B. The tree has 64 nodes representing candidate tokens and a depth of 4 which indicates 4 MEDUSA heads involved in calculation. Each node indicates a token from a top-k prediction of a MEDUSA head, and the edges show the connections between them. The red lines highlight the path that correctly predicts the future tokens.

* We know the accuracy at each step. We can use that

to find the accuracy & prune the tree accordingly (kind of like beam search)

* Even though we are verifying in parallel, we will always have a budget of how many we want to verify. So pruning helps here.



Acceptance Criteria

* The paper devised its own sampling method instead of standard nucleus sampling.
* Aim is to pick candidates that are likely enough according to the model. So, they select a candidate based on 2 criteria:
  (i) Probability given by the model to that token
  (ii) Entropy at that point → if the entropy is low (less random) then we should have a higher bar for accepting a token & if it is high (more random) then there are multiple tokens over which the probability mass is distributed so we should adjust the threshold accordingly

* Select the 1st token greedily.
* For the rest

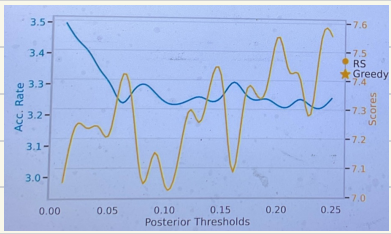
$$P_{original}(x_{n+k} \mid x_1, x_2 \cdots x_{n+k-1}) > \min(\epsilon, \delta \exp(-H(p_{original}(\cdot \mid x_1, x_2 \cdots x_{n+k-1}))))$$

(δ: hyperparameter)

Minimum of a hard threshold & entropy dependent threshold

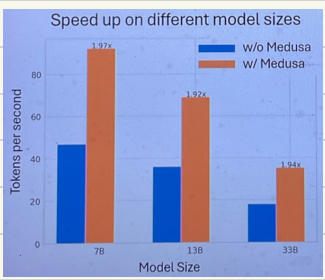
* Select the longest subsequence that satisfies the above criteria.

Impact of Threshold



* The graph here shows the impact of the threshold.
* As we increase the threshold & become more rigid, the acceleration rate reduces but the accuracy/performance score increases.

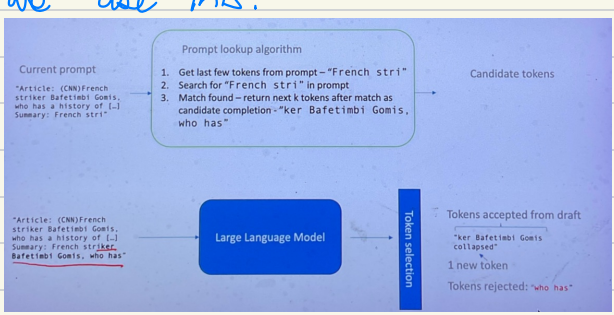
Results



* The graph here shows the performance speedup we get when using Medusa. It achieves speedup of roughly upto 2x.

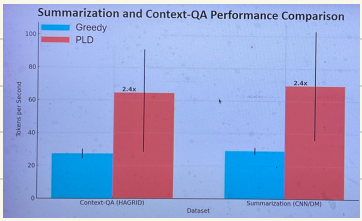
Prompt Lookup Decoding

Q) Think about tasks like Content grounded QA, RAG, summarization where the prompt itself has some information that we can use in the answers. Can we use this?



* Search for retrieved tokens (from the prompt) within the prompt itself. If found, return the next k tokens after match as candidate for completion.
* Pass it through the language model to verify & accept & continue (or replace with generated token) the process again.

Results



*Achieve a speedup of upto 2.4x

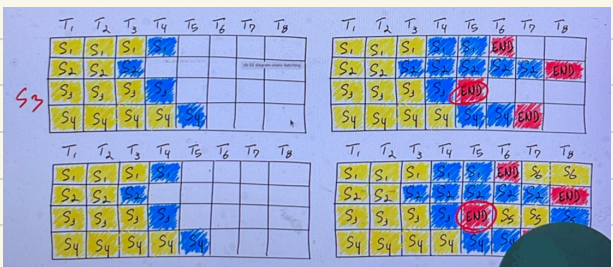
Lookahead Decoding

*Another way of generating n-gram candidates & verifying them using tree attention (like Medusa)
- No need to train "additional" LM heads for next-next token prediction
- Doesn't rely on input prompt to search for n-grams
- Inspired by Jacobi iteration method
- Starts with a random guess completion & maintains a pool of n-grams generated by the model.
- Heavily relies on tree-attention to verify as well as generate n-gram candidates in parallel, starting from a random guess.

↳ Idea is that we start with an input prompt & randomly select x_1, x_2, x_3, x_4 & x_5. But intuitively we know that given x_2 the model will select a good next token i.e., a forward pass will give us meaningful bigram. eg: 'The cat sat' & next token is randomly chosen as 'on' then the token that model gives as having the highest probability after 'on' (most likely an article like 'a', 'an', 'the') will make sense. We can use these bigrams in future iterations. So say we get 'a' as the token with highest probability. 'On a' is a good bigram so we can cache it as a prefix. Next time when we verify 'on a' then the next word generated will likely form a good trigram like 'on a table'. This way we can create a pool of good candidates which we can use to verify in parallel.

Continuous Batching

* When hosting LLMs as a service (like vLLM or TGI), we have multiple users using this system and multiple requests that come in simultaneously.
* Typically, requests are processed in a batch.
* But different sequences in a batch will end at different timesteps (i.e., all have different sizes). In this example we see that S_3 finishes early.



Q) So what do we do? Do we continue processing the batch (& waste some resources) or do something else?
The idea of continuous batching comes in here where as soon as a sequence ends we fill in another sequence (like here we fill in S_5 after S_3 ends). We will obviously have to pause in between to go through the prefill step (ie, until each token into the input prompt is pre-filled in) before we start generating new tokens.

ORCA: framework which implements this. ⊕
↳ Link: https://www.usenix.org/conference/osdi22/presentation/yu (09/2022)

Lecture 16.1 – Retrieval based Language Models - I

* Parametric LLMs
  - ↳ Amazing creative writers
  - ↳ Stellar performance in exams
  - ↳ But they hallucinate