Generative AI - Text to Text

Instructor: Parivesh

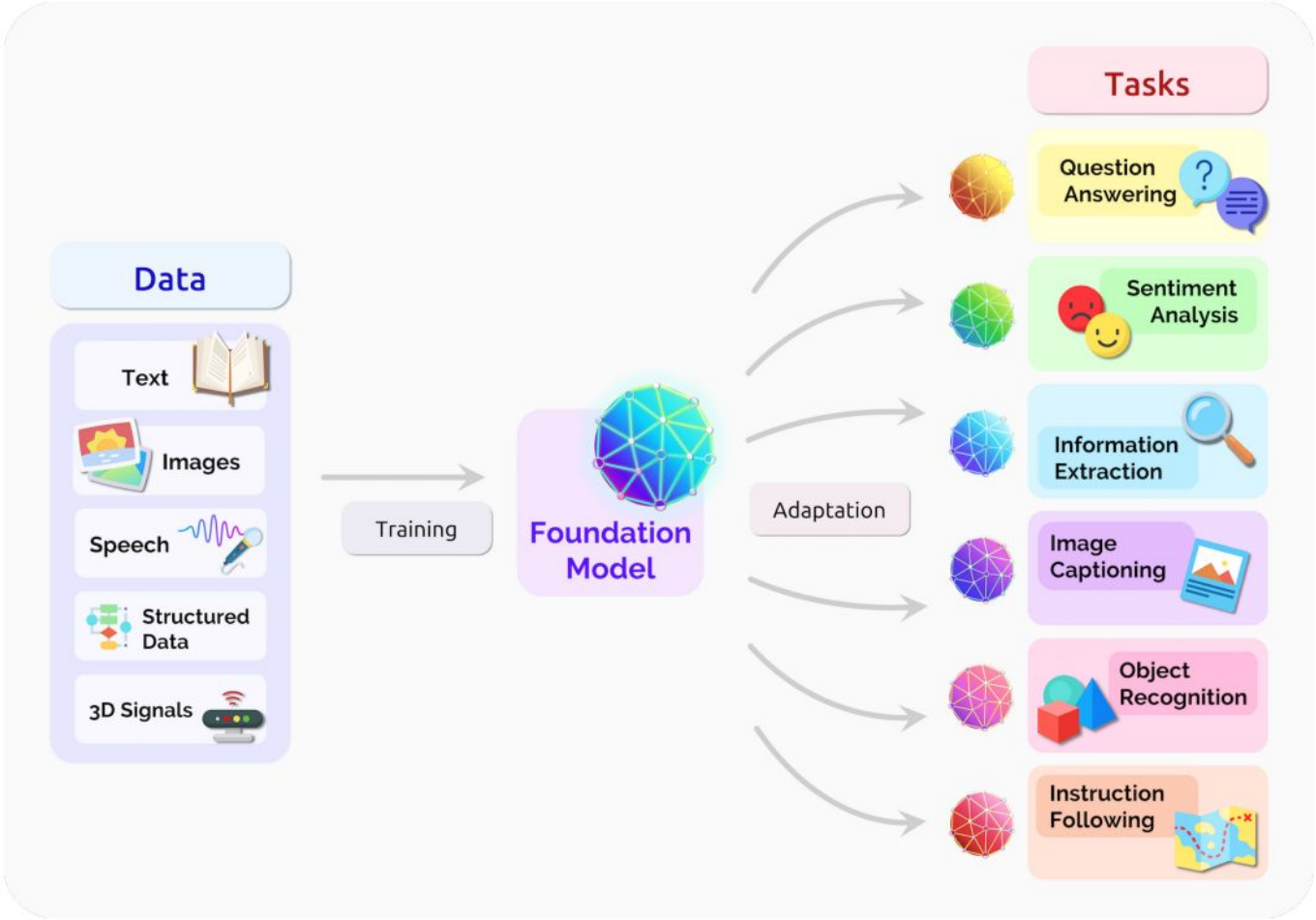
Introduction – Parivesh

Applied Scientist, Amazon
(Amazon Search)

Optimize Your Experience

- ✅ Ask questions in the class
- ✅ Don’t ignore the math (Whys and Hows)
- ✅ Fortify understanding by reading papers, walking through code implementations and trying out notebooks!

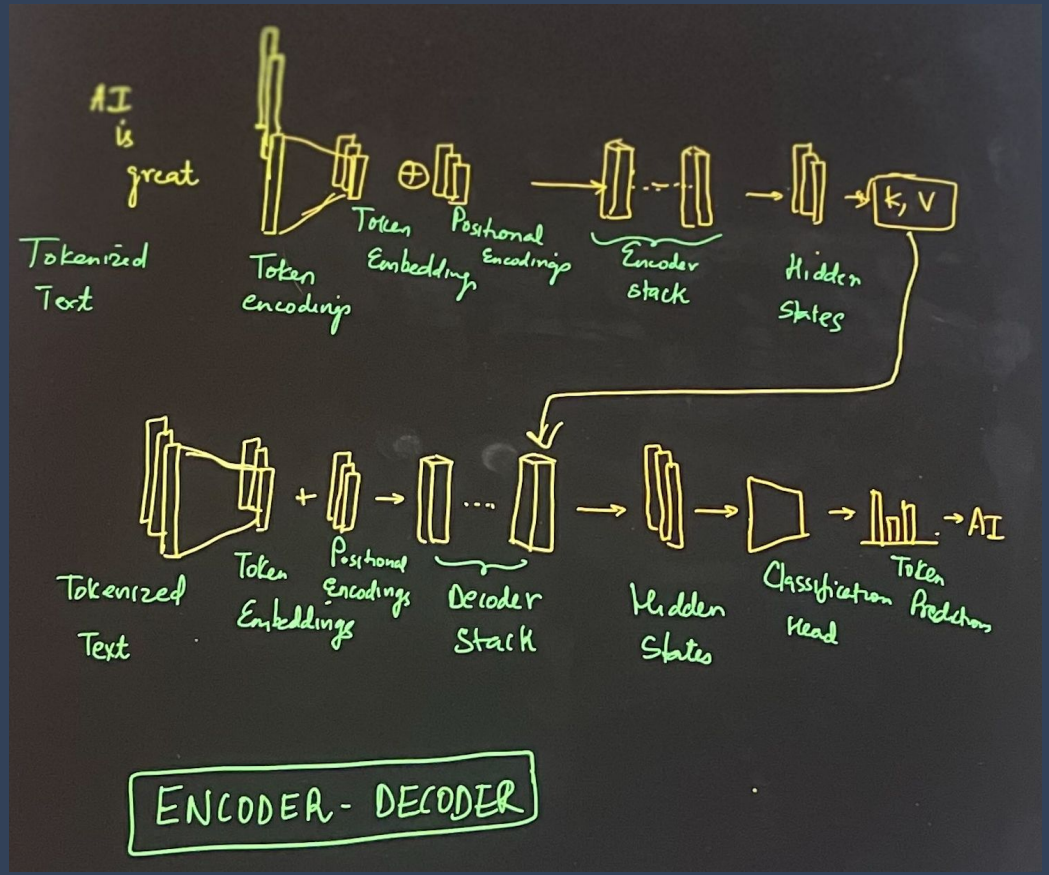
- ✅ Enjoy the subject. Study deeply, seek understanding, read papers, practice problems, ask questions
- ✅ Don’t spend time in self-doubt. Unlike real life tortoise & hare race, slow & steady literally wins the career race



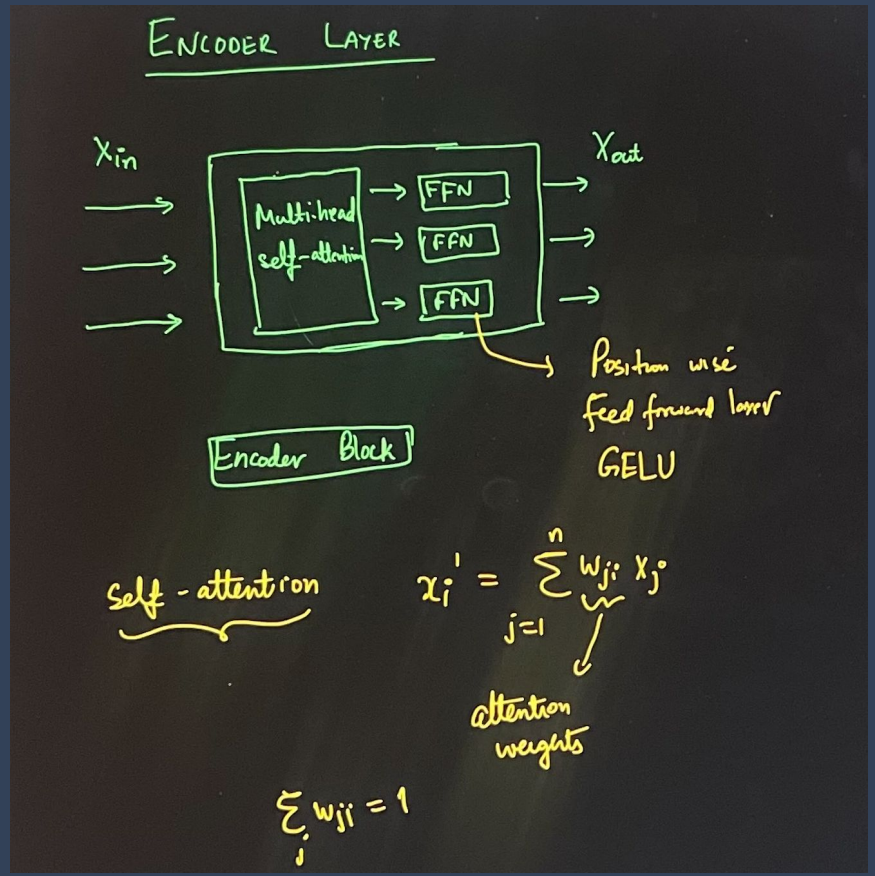
Objectives

- To gain a good grounding in the theoretical underpinnings of text to text generative models
- To understand the strengths & weaknesses of LLMs
- How are LLMs pre-trained and fine-tuned in practice
- Practical considerations while deploying generative models - inference, response quality, staleness, toxicity etc.
- Art of reading papers and staying abreast with the field

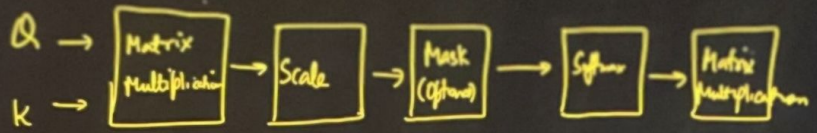
Transformers - For the nth time!



Encoder Layer - Closer Look

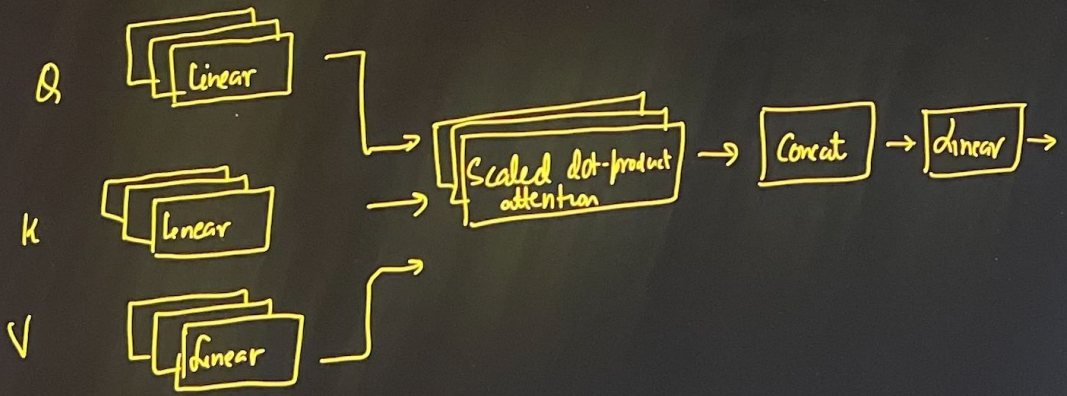


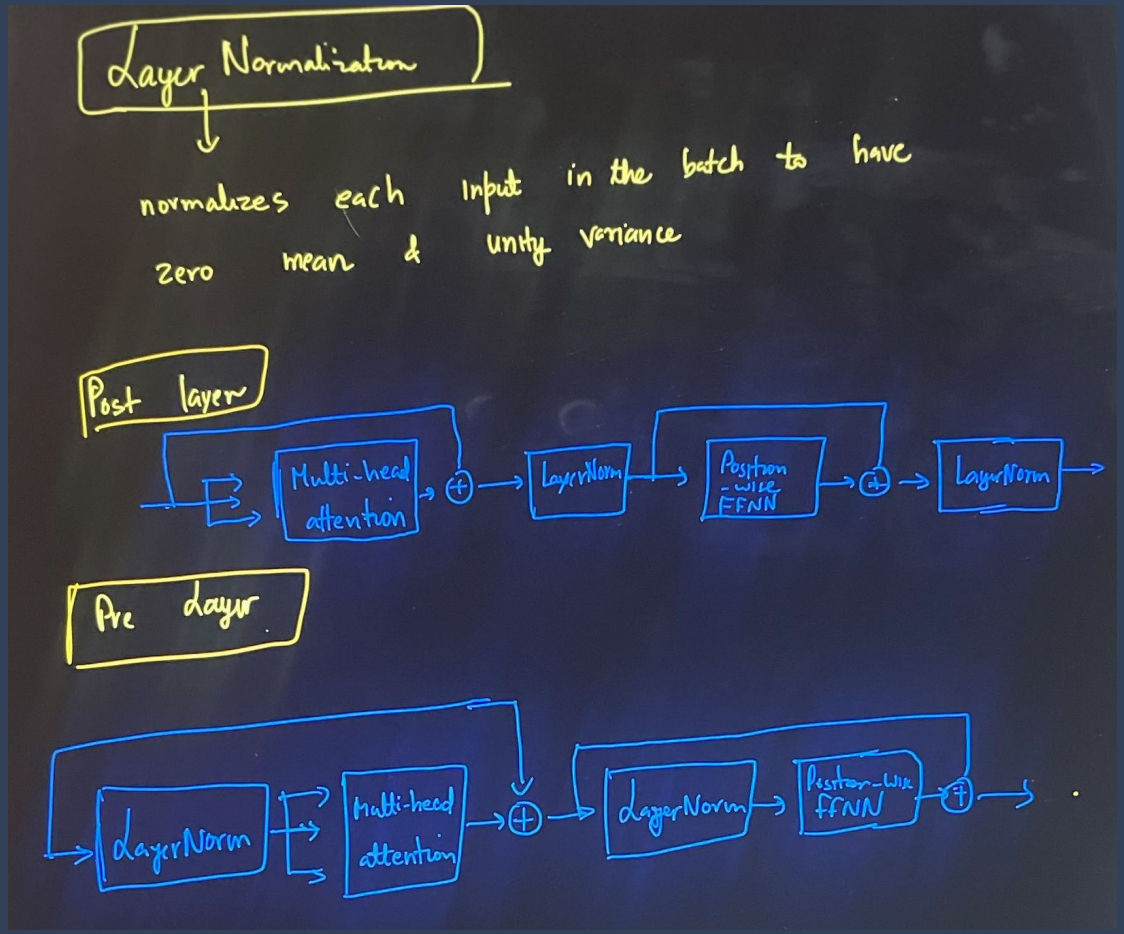
Scaled- dot - product attention



Multi-head Attention

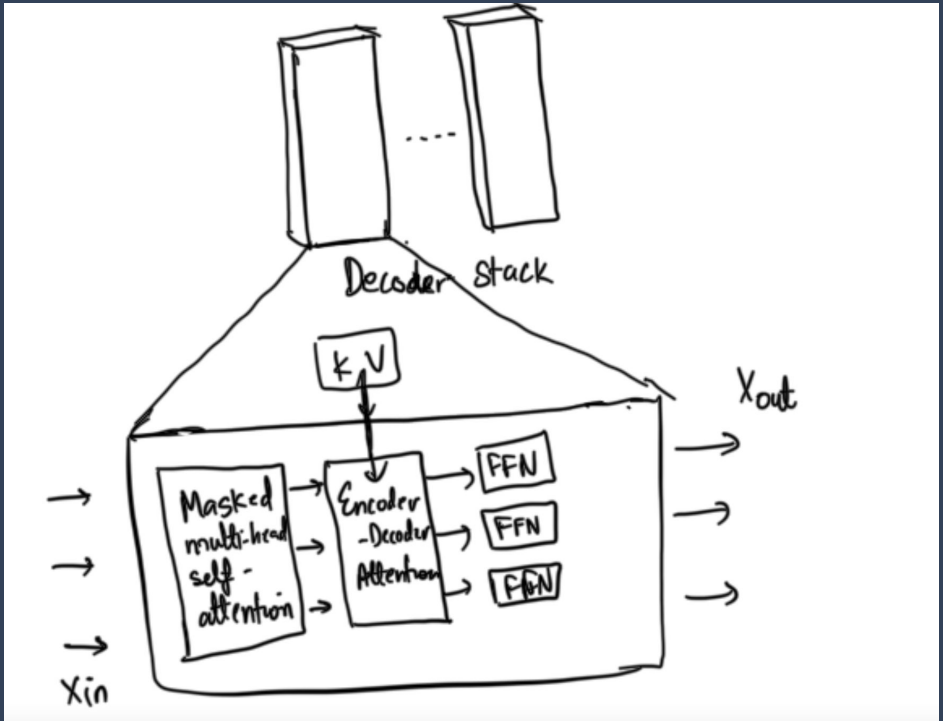
multiple attention heads





- Layer normalization normalizes across the feature dimension, thus enjoying benefits of scale independence and batch size independence.
- Batch normalization is usually empirically less effective than layer normalization in NLP tasks, where the inputs are often variable-length sequences.

Decoder Block




```
def scaled_dot_product_attention(query, key, value, mask=None):
    dim_k = query.size(-1)
    scores = torch.bmm(query, key.transpose(1,2)) / sqrt(dim_k)
    if mask is not None:
        scores = scores.masked_fill(mask==0, float("-inf"))
    weights = F.softmax(scores, dim=-1)
    return weights.bmm(value)
```

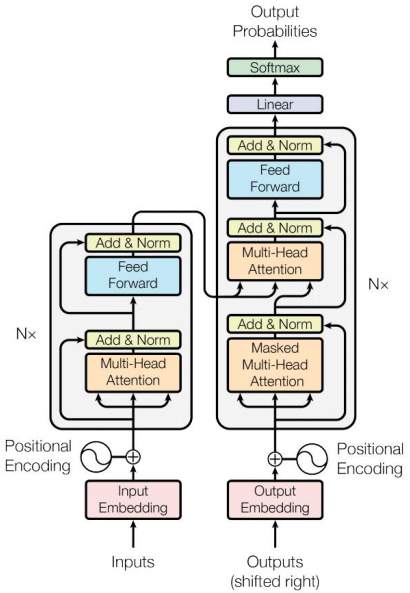
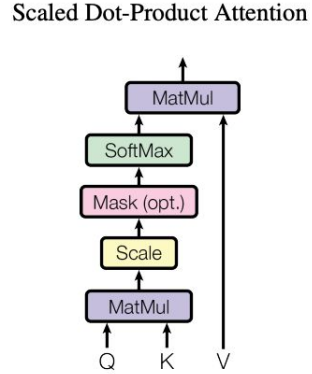
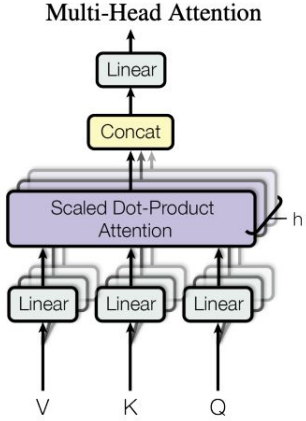


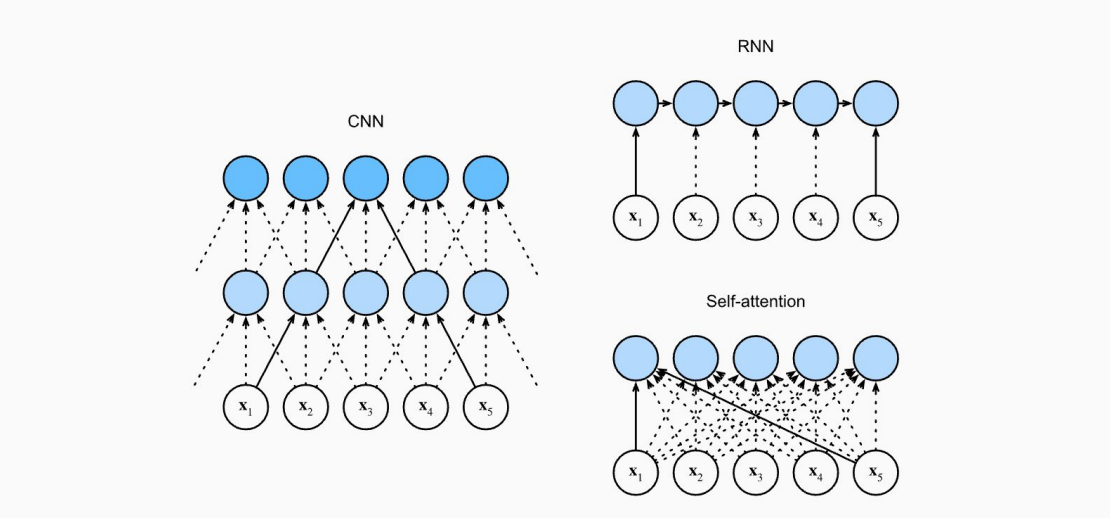
Figure 1: The Transformer - model architecture.





Illustrations from Attention is all you need (Vaswani et al)

Why did the LLMs take the world by storm?



- computational complexity, sequential operations, and maximum path lengths if we are mapping a sequence of length n to length n ?
- dimensional of the hidden state - d
- Transformers —> not sequential and can be parallelized, maximum path length O(1), computational complexity $O(n^2 d)$
- RNNs —> Sequential with length O(n), computational complexity $O(nd^2)$

Pre-training using self-supervision

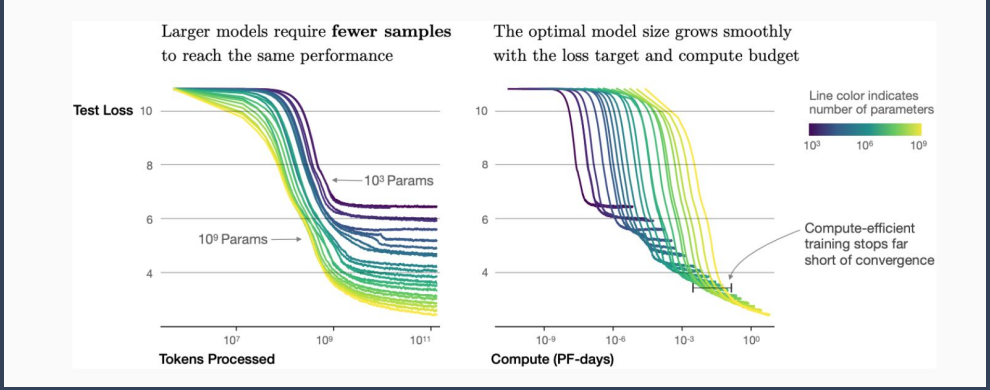
- Training encoder - decoder models needs <source, target> pairs
- Example - Neural Machine Translation, Spelling Correction, Summarization, Q&A
- But gathering human labels is hard. To learn semantic relationships on internet-scale data, human labeling quickly becomes infeasible
- Solution - ?
- BERT !! - Introduced MLM (Masked Language Model) paradigm

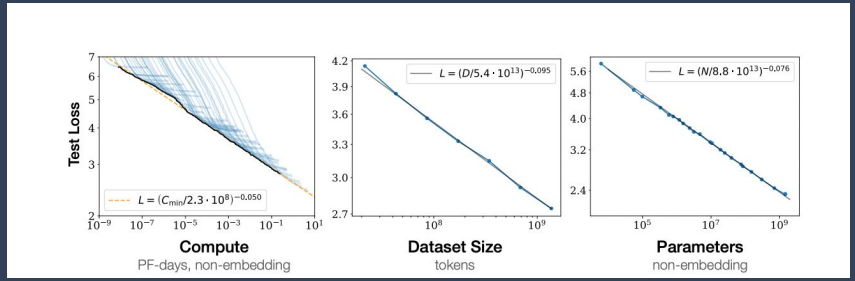
- MLM enables/enforces bidirectional learning from text by masking (hiding) a word in a sentence and forcing BERT to bidirectionally use the words on either side of the covered word to predict the masked word.

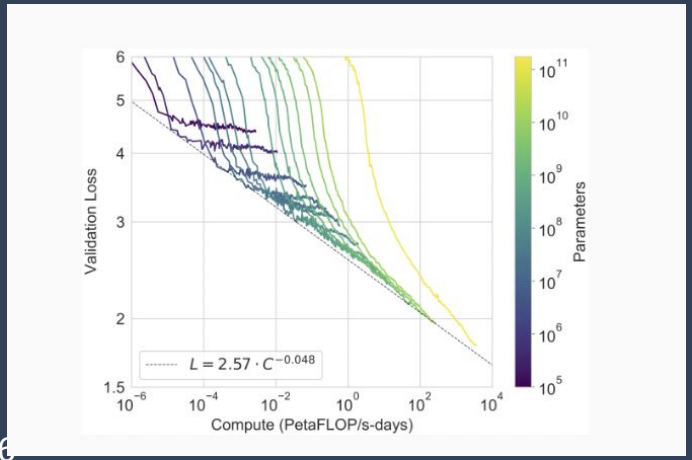
- NSP (Next Sentence Prediction) is used to help BERT learn about relationships between sentences by predicting if a given sentence follows the previous sentence or not.

- Given larger data for pretraining, the Transformer architecture performs better with an increased model size and training compute, demonstrating superior *scaling* behavior

- Performance of Transformer-based language models scales as a power law with the amount of model parameters, training tokens, and training compute



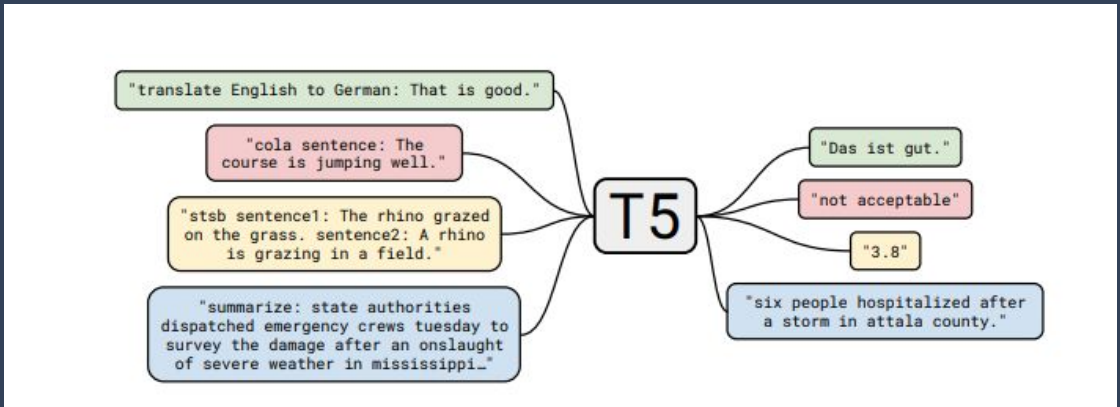




Scaling Laws - https://arxiv.org/pdf/2001.08361.pdf

T5

- Text to Text Transfer Transformer
- Multitask Learning but with a trick → using task prefixes!



Single-task learning: $\mathcal{D} = \{(\mathbf{x}, \mathbf{y})_k\}$

[supervised] $\min_\theta \mathcal{L}(\theta, \mathcal{D})$

Typical loss: negative log likelihood

$$\mathcal{L}(\theta, \mathcal{D}) = -\mathbb{E}_{(x,y)\sim\mathcal{D}}[\log f_\theta(\mathbf{y} \mid \mathbf{x})]$$

What is a task? (more formally this time)

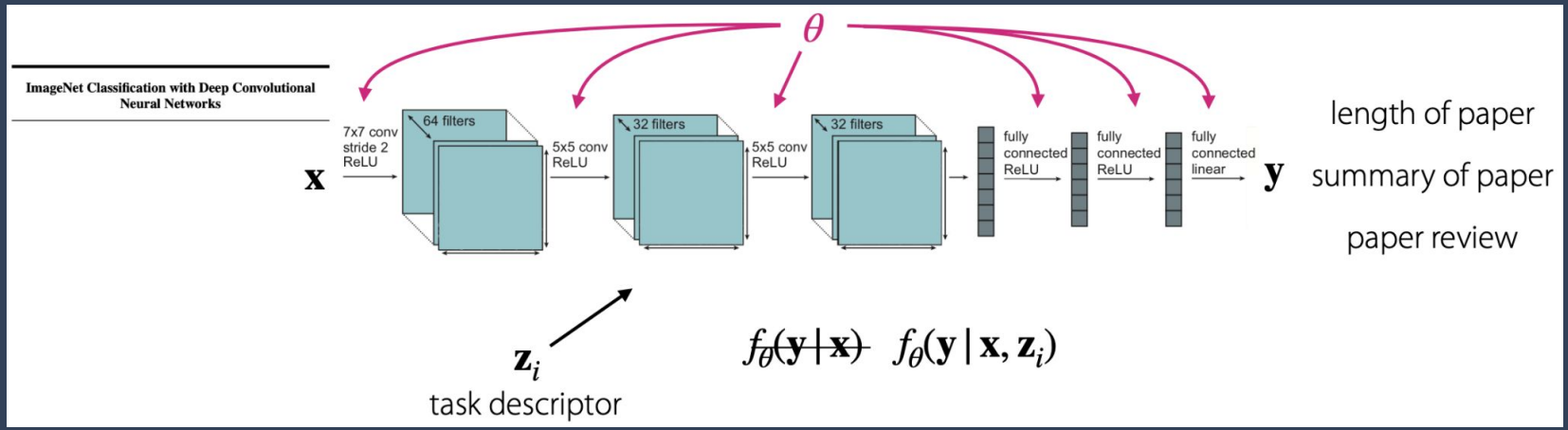
A task: $\mathcal{T}_i \triangleq \{p_i(\mathbf{x}), p_i(\mathbf{y} \mid \mathbf{x}), \mathcal{L}_i\}$

data generating distributions

Corresponding datasets: $\mathcal{D}_i^{tr}$ $\mathcal{D}_i^{test}$

will use $\mathcal{D}_i$ as shorthand for $\mathcal{D}_i^{tr}$:

5



Split θ into shared parameters θ^{sh} and task-specific parameters θ^i

Then, our objective is: $\min_{\theta^{sh},\theta^1,\ldots,\theta^T} \sum_{i=1}^{T} \mathscr{L}_i(\{\theta^{sh}, \theta^i\}, \mathscr{D}_i)$

Vanilla MTL objective:

$$\min_{\theta} \sum_{i=1}^{T} \mathscr{L}_i(\theta, \mathscr{D}_i)$$

Often want to weight tasks differently:

$$\min_{\theta} \sum_{i=1}^{T} w_i \mathscr{L}_i(\theta, \mathscr{D}_i)$$

How to choose w_i?

- manually based on importance or priority
- *dynamically* adjust throughout training

a. various heuristics

encourage gradients to have similar magnitudes
(Chen et al. GradNorm. ICML 2018)

b. optimize for the worst-case task loss

$$\min_{\theta} \max_{i} \mathscr{L}_i(\theta, \mathscr{D}_i)$$

(e.g. for task robustness, or for fairness)

Optimizing the objective

Vanilla MTL Objective: $\min_{\theta} \sum_{i=1}^{T} \mathscr{L}_i(\theta, \mathscr{D}_i)$

Basic Version:

1. Sample mini-batch of tasks $\mathscr{B} \sim \{\mathscr{T}_i\}$
2. Sample mini-batch datapoints for each task $\mathscr{D}_i^b \sim \mathscr{D}_i$
3. Compute loss on the mini-batch: $\hat{\mathscr{L}}(\theta, \mathscr{B}) = \sum_{\mathscr{T}_k \in \mathscr{B}} \mathscr{L}_k(\theta, \mathscr{D}_k^b)$
4. Backpropagate loss to compute gradient $\nabla_\theta \hat{\mathscr{L}}$
5. Apply gradient with your favorite neural net optimizer (e.g. Adam)

Note: This ensures that tasks are sampled uniformly, regardless of data quantities.

Tip: For regression problems, make sure your task labels are on the same scale!

Challenges with multi-task learning

- Negative transfer due to optimization challenges (tasks may learn at different rates, cross-task interference) and limited representational capacity (model size much bigger than single task model)—> Share less across tasks!

$$\min_{\theta^{sh},\theta^1,\ldots,\theta^T} \sum_{i=1}^{T} \mathscr{L}_i(\{\theta^{sh},\theta^i\},\mathscr{D}_i) + \lambda \underbrace{\sum_{i'=1}^{T} \|\theta^i - \theta^{i'}\|}_{\text{“soft parameter sharing”}}$$

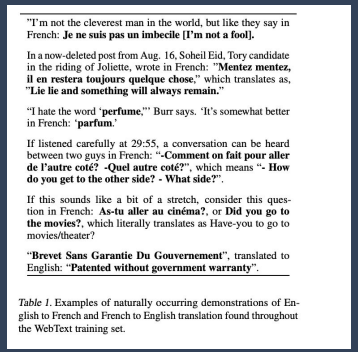
- Overfitting due to not sharing enough → Share more!

- What if there are a lot of tasks → Which tasks are similar? Should you train all of them together?

GPT & GPT-2

- Generative Pre-training
- $$p(x) = \prod_{i=1}^{n} p(s_n|s_1, ..., s_{n-1})$$
- But we should model based not just on tokens but tokens & tasks
  p(output|input, task)
- This conditional probability has been studied in meta-learning & multi-task learning settings
- We employ the trick that language itself can specify what to do in its prompts in supervised way in T5

- Translation training example can be written as the sequence (translate to french, english text, french text).
- Reading comprehension training example can be written as (answer the question, document, question, answer).
- Hard to get data for each task in a supervised way
- Language modeling on web-scale data will automatically be multitask due to presence of examples like below



"I'm not the cleverest man in the world, but like they say in French: **Je ne suis pas un imbecile [I'm not a fool].**

In a now-deleted post from Aug. 16, Soheil Eid, Tory candidate in the riding of Joliette, wrote in French: "**Mentez mentez, il en restera toujours quelque chose**," which translates as, "**Lie lie and something will always remain.**"

"I hate the word '**perfume**,"' Burr says. 'It's somewhat better in French: '**parfum**.'

If listened carefully at 29:55, a conversation can be heard between two guys in French: "**-Comment on fait pour aller de l'autre coté? -Quel autre coté?**", which means "**- How do you get to the other side? - What side?**".

If this sounds like a bit of a stretch, consider this question in French: **As-tu aller au cinéma?**, or **Did you go to the movies?**, which literally translates as Have-you to go to movies/theater?

"**Brevet Sans Garantie Du Gouvernement**", translated to English: "**Patented without government warranty**".

Table 1. Examples of naturally occurring demonstrations of English to French and French to English translation found throughout the WebText training set.

Unsupervised Multitask Learning by Language Models

Challenges with generative models and some solutions

- Decoding (response) is probabilistic and hence can be unhinged (meaningless / low quality, toxic, false, hallucination etc.) → Chain of Thought prompting, RLHF, DPO, RLAIF
- Longer context (prompt) → Takes longer to compute (why?)
- What if we need domain adaptation? → Supervised Fine-tuning, Instruction Fine-tuning
- Deploying in production is a major issue due to model size, attention is quadratic in sequence length → Distillation, Specific kernels, Runtimes, Hardware, Flash Attention, Hyper Attention

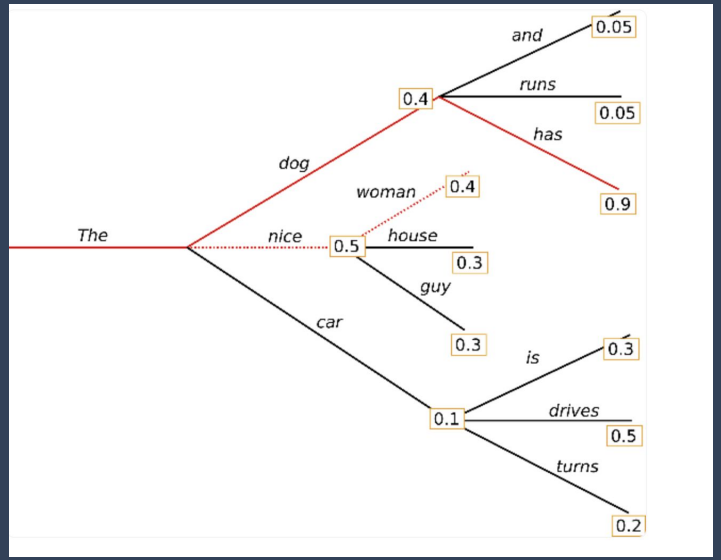
Decoding Mechanisms

$$P(w_{1:T}|W_0) = \prod_{t=1}^{T} P(w_t|w_{1:t-1}, W_0) \text{ ,with } w_{1:0} = \emptyset,$$

- Greedy Search - Emit the most likely token at every time step

$$w_t = argmax_w P(w|w_{1:t-1}) \text{ at each timestep } t$$

- Beam Search - Keep the most likely num_beams of hypotheses at each time step and eventually choosing the hypothesis that has the overall highest probability

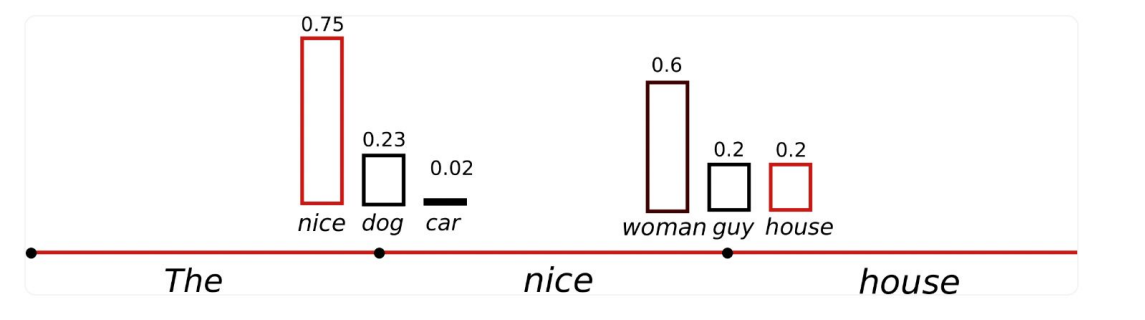


- Greedy would emit nice first, followed by house
- Beam Search - dog, has

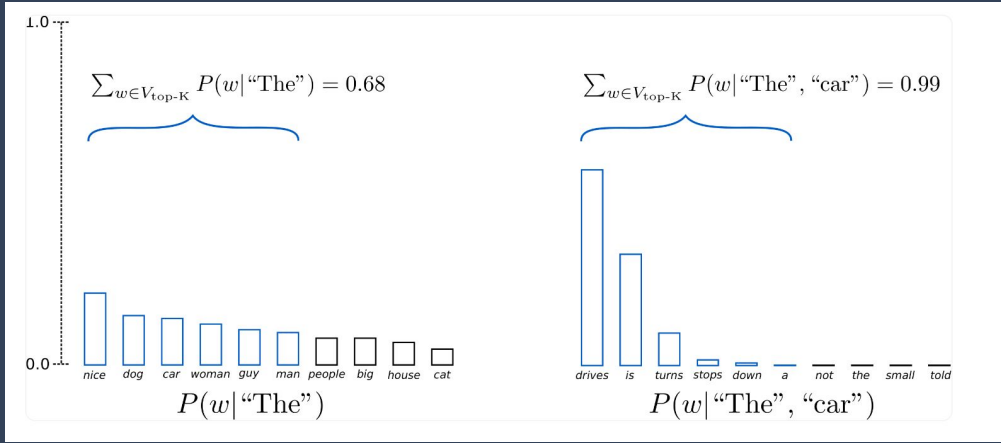
- Sampling based - Randomly pick the next word from the conditional probability distribution

$$w_t \sim P(w|w_{1:t-1})$$

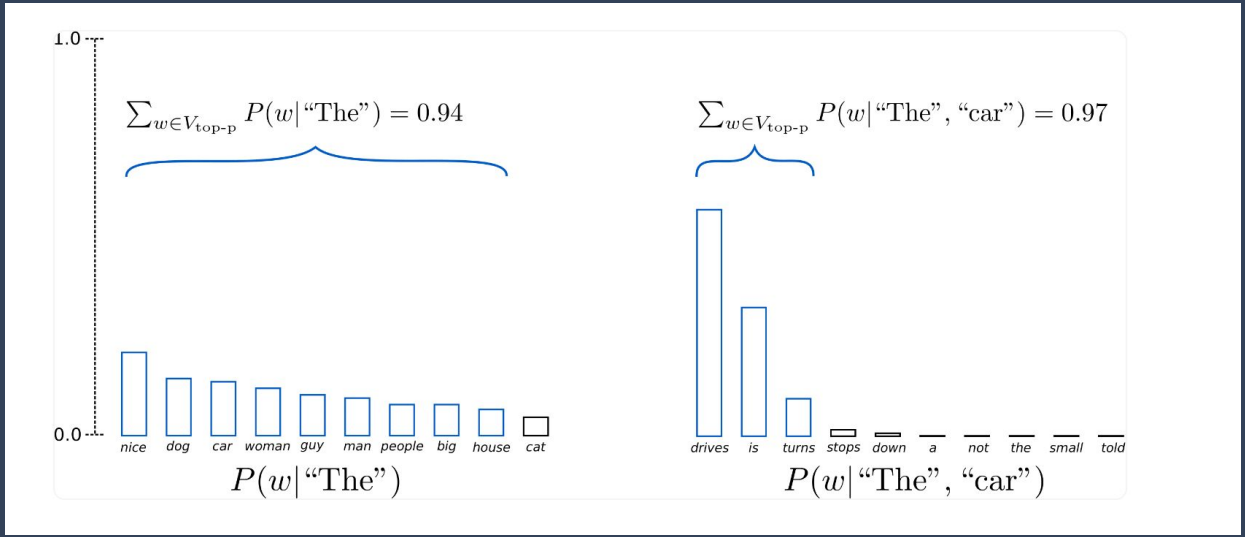
- This will lead to probabilistic responses (not fixed anymore!)



- Top-K sampling - *K* most likely next words are filtered and the probability mass is redistributed among only those *K* next words. Then sample! GPT-2 used this and was successful in story generation tasks.



- Nucleus Sampling - Choose from the smallest possible set of words whose cumulative probability exceeds the probability *p*



DEMO - Decoding Strategies

Generating better responses

META-LEARNING and IN-CONTEXT LEARNING

Given data from $\mathcal{T}_1, \ldots, \mathcal{T}_n$, solve new task $\mathcal{T}_{\text{test}}$ more quickly / proficiently / stably

Key assumption: meta-training tasks and meta-test task drawn i.i.d. from same task distribution

$$\mathcal{T}_1, \ldots, \mathcal{T}_n \sim p(\mathcal{T}), \mathcal{T}_{\text{test}} \sim p(\mathcal{T})$$

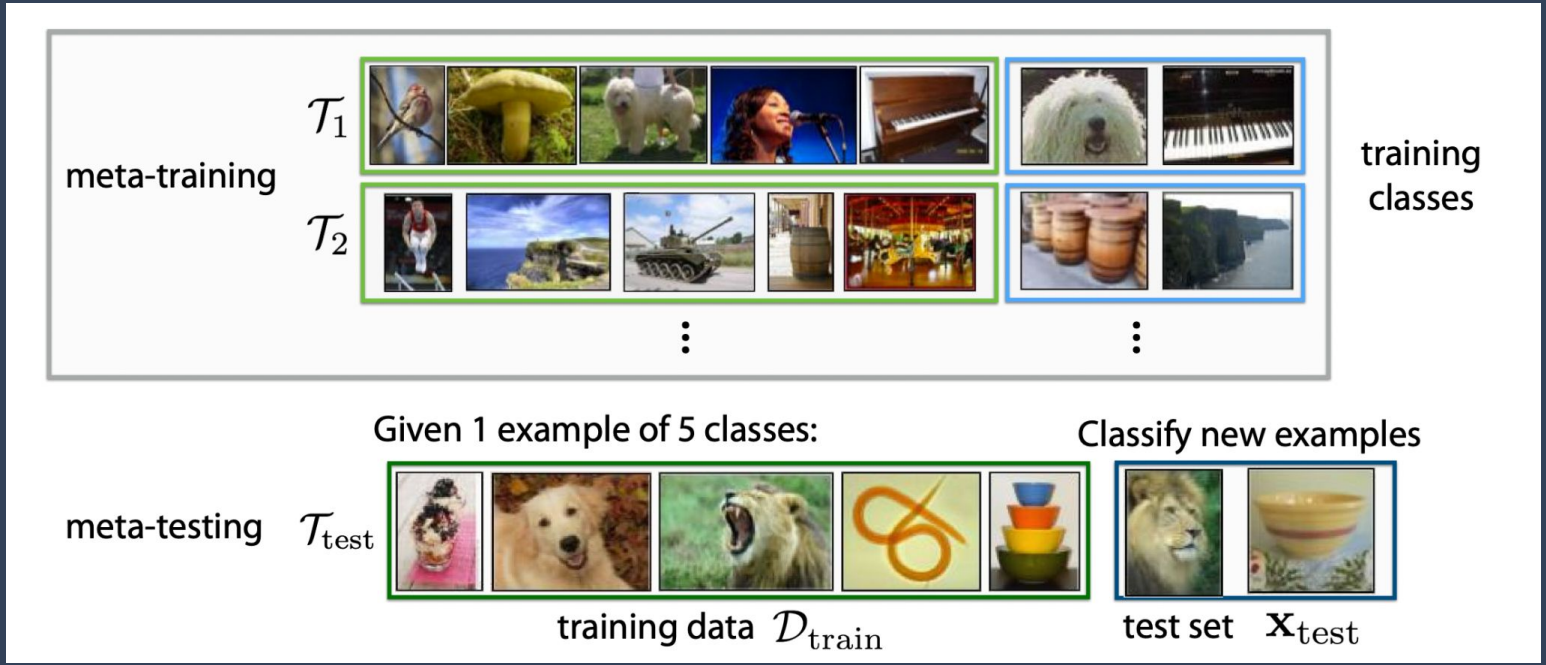
Like before, tasks must share structure.

Given 1 example of 5 classes:

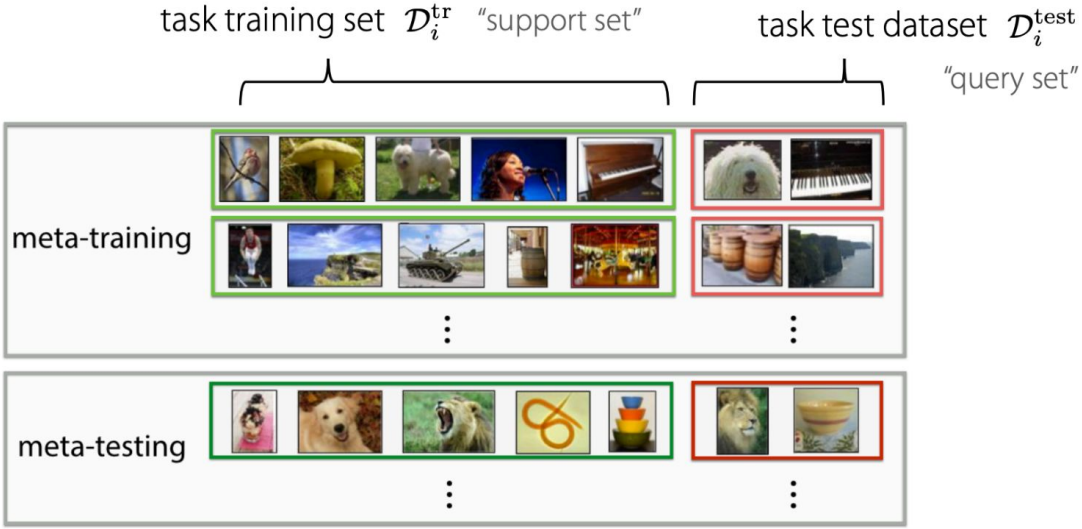
training data $\mathcal{D}_{\text{train}}$

Classify new examples

test set $\mathbf{x}_{\text{test}}$



- You can use this for language generation too!



k-shot learning: learning with **k** examples per class
(or **k** examples total for regression)

N-way classification: choosing between N classes

Supervised Learning:

Inputs: $\mathbf{x}$ Outputs: $\mathbf{y}$

$$\mathbf{y} = g_\phi(\mathbf{x})$$

Data: $\{(\mathbf{x}, \mathbf{y})_i\}$

Meta Supervised Learning:

Inputs: $\mathcal{D}^{\text{tr}}$ $\mathbf{x}^{\text{ts}}$ Outputs: $\mathbf{y}^{\text{ts}}$

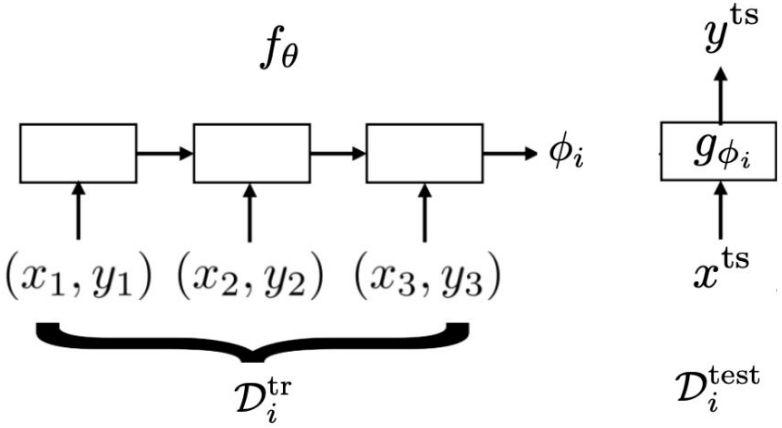
$$\{(\mathbf{x}, \mathbf{y})_{1:K}\} \quad \mathbf{y}^{\text{ts}} = f_\theta(\mathcal{D}^{\text{tr}}, \mathbf{x}^{\text{ts}})$$

Data: $\{\mathcal{D}_i\}$

$$\mathcal{D}_i : \{(\mathbf{x}, \mathbf{y})_j\}$$

Key idea: Train a neural network to represent $\phi_i = f_\theta(\mathcal{D}_i^{\text{tr}})$ "learner"

Predict test points with $\mathbf{y}^{\text{ts}} = g_{\phi_i}(\mathbf{x}^{\text{ts}})$

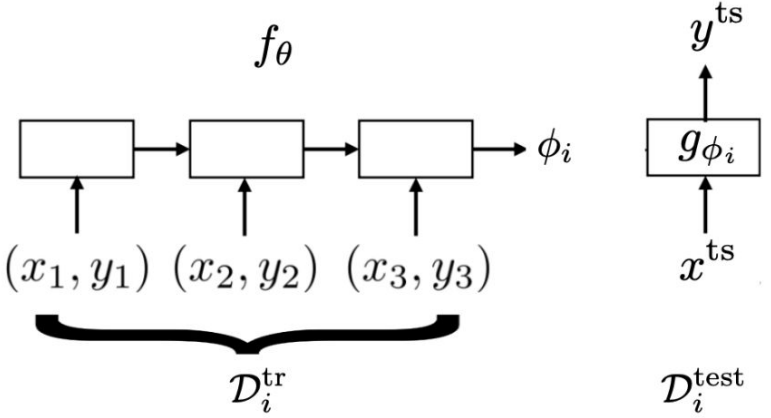


Train with standard supervised learning!

$$\min_\theta \sum_{\mathcal{T}_i} \underbrace{\sum_{(x,y)\sim\mathcal{D}_i^{\text{test}}} -\log g_{\phi_i}(y \mid x)}_{\mathcal{L}(\phi_i, \mathcal{D}_i^{\text{test}})}$$

$$\min_\theta \sum_{\mathcal{T}_i} \mathcal{L}(f_\theta(\mathcal{D}_i^{\text{tr}}), \mathcal{D}_i^{\text{ts}})$$

Key idea: Train a neural network to represent $\phi_i = f_\theta(\mathcal{D}_i^{\text{tr}})$.



1. Sample task $\mathcal{T}_i$ *(or mini batch of tasks)*
2. Sample disjoint datasets $\mathcal{D}_i^{\text{tr}}, \mathcal{D}_i^{\text{test}}$ from $\mathcal{D}_i$
3. Compute $\phi_i \leftarrow f_\theta(\mathcal{D}_i^{\text{tr}})$
4. Update θ using $\nabla_\theta \mathcal{L}(\phi_i, \mathcal{D}_i^{\text{test}})$

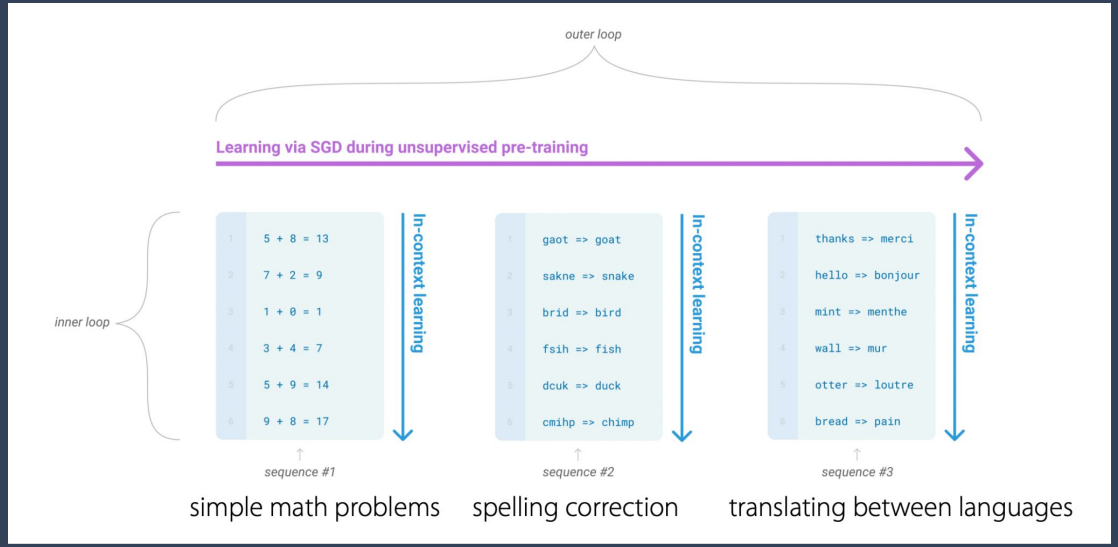
$\mathcal{D}_i^{\text{tr}}$

$\mathcal{D}_i^{\text{test}}$

GPT-3

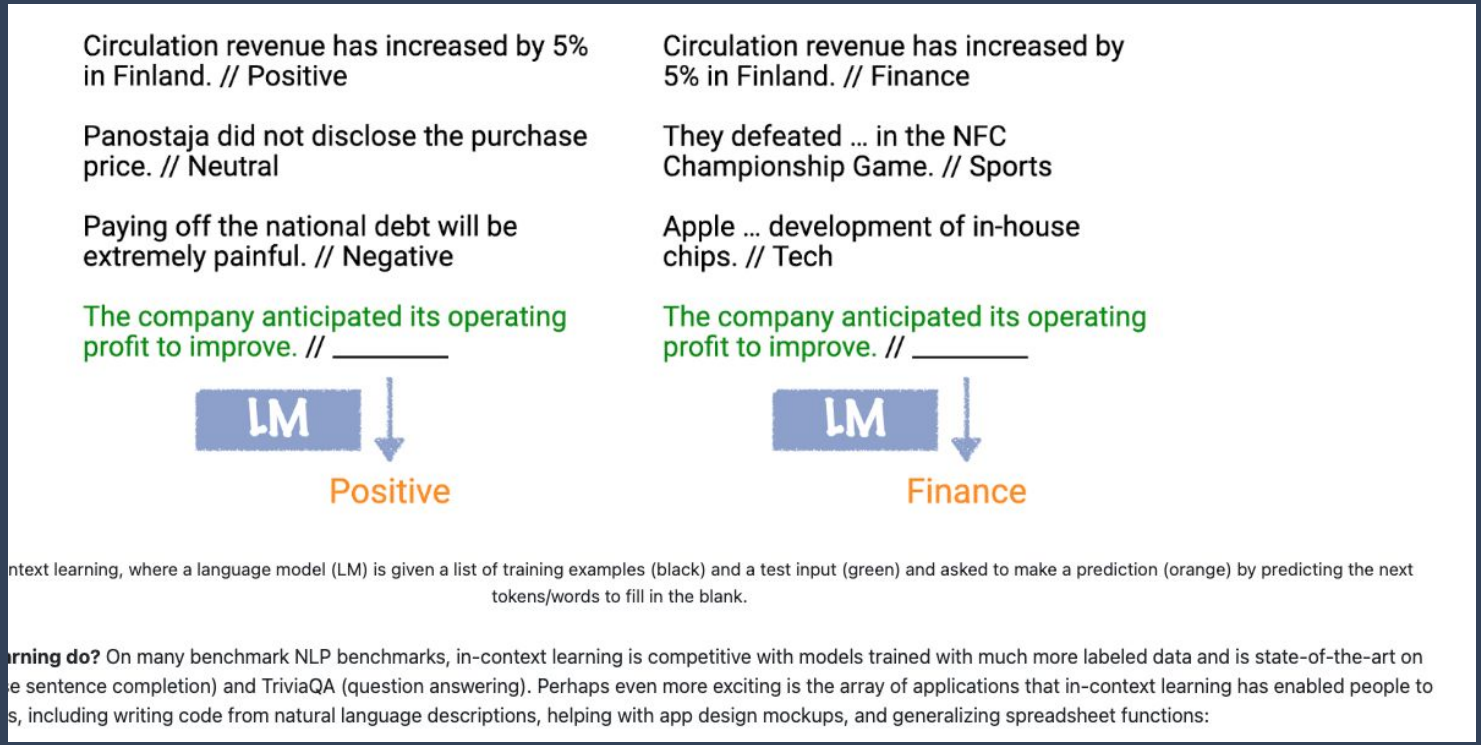
- black-box meta-learner trained on language generation tasks
- $\mathcal{D}_i^{\text{tr}}$: sequence of characters $\mathcal{D}_i^{\text{ts}}$: the following sequence of characters
- Datasets - crawled data from the internet, English-language Wikipedia, two books corpora
- Architecture - Transformer, 175B parameters, 96M batch size, 96 layers
- Big model, more data, clean data, better attention mechanism, train using model parallelism (divide layers width & depth wise across GPUs to train in a distributed manner while minimizing weight sync)
- During unsupervised pre-training, LLM develops a broad set of skills and pattern recognition abilities. It then uses these abilities at inference time to rapidly adapt to or recognize the desired task.

- Different tasks - Spelling correction, Translation, Simple math problems, many other tasks
- How to solve all these tasks using 1 architecture? Put them all in form of text
- Getting meta-training data for these tasks is easy!



Few-Shot

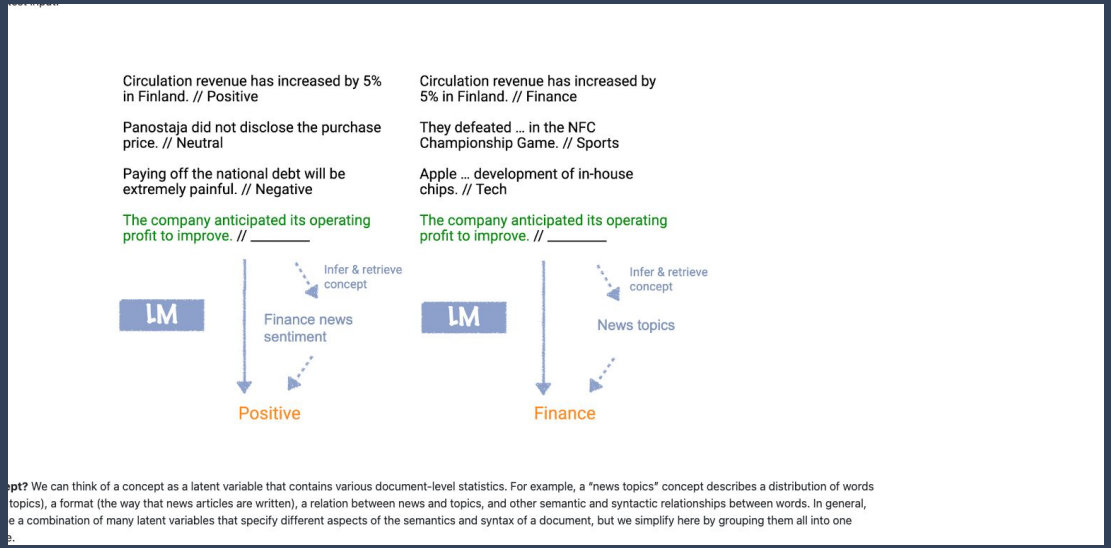
- No weight update allowed during inference. Not trained to learn from examples
- The inner loop of this process occurs within the forward-pass upon each sequence.
- Only used during inference for conditional generation



ntext learning, where a language model (LM) is given a list of training examples (black) and a test input (green) and asked to make a prediction (orange) by predicting the next tokens/words to fill in the blank.

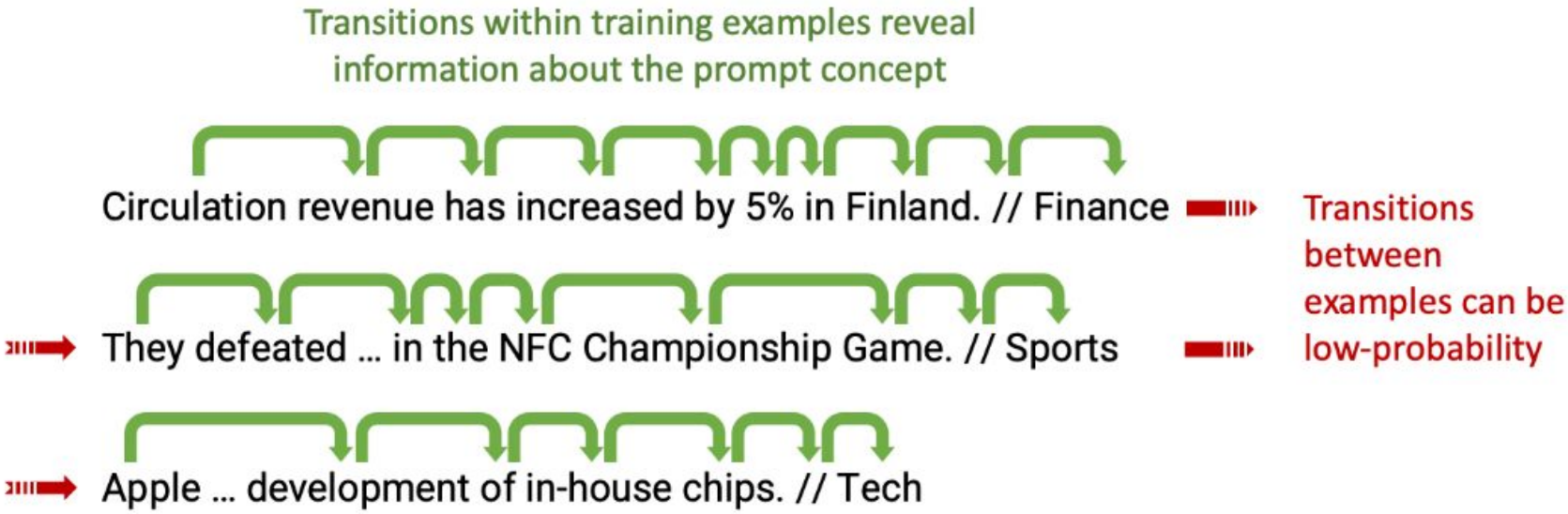
rning do? On many benchmark NLP benchmarks, in-context learning is competitive with models trained with much more labeled data and is state-of-the-art on e sentence completion) and TriviaQA (question answering). Perhaps even more exciting is the array of applications that in-context learning has enabled people to s, including writing code from natural language descriptions, helping with app design mockups, and generalizing spreadsheet functions:

- There's seemingly a mismatch between pretraining (what it's trained to do, which is next token prediction) and in-context learning (what we're asking it to do).



- A document is generated by first sampling a latent concept, and then the document is generated by conditioning on the latent concept
- In-context learning prompts are lists of IID (independent and identically distributed) training examples concatenated together with one test input. Each example in the prompt is drawn as a sequence conditioned on the same *prompt concept*, which describes the task to be learned.

$$p(\text{output}|\text{prompt}) = \int_{\text{concept}} p(\text{output}|\text{concept}, \text{prompt}) p(\text{concept}|\text{prompt}) d(\text{concept})$$



- Training examples provide signal
- Transitions between training examples can be low-probability (noise)
- In-context learning is robust to some noise

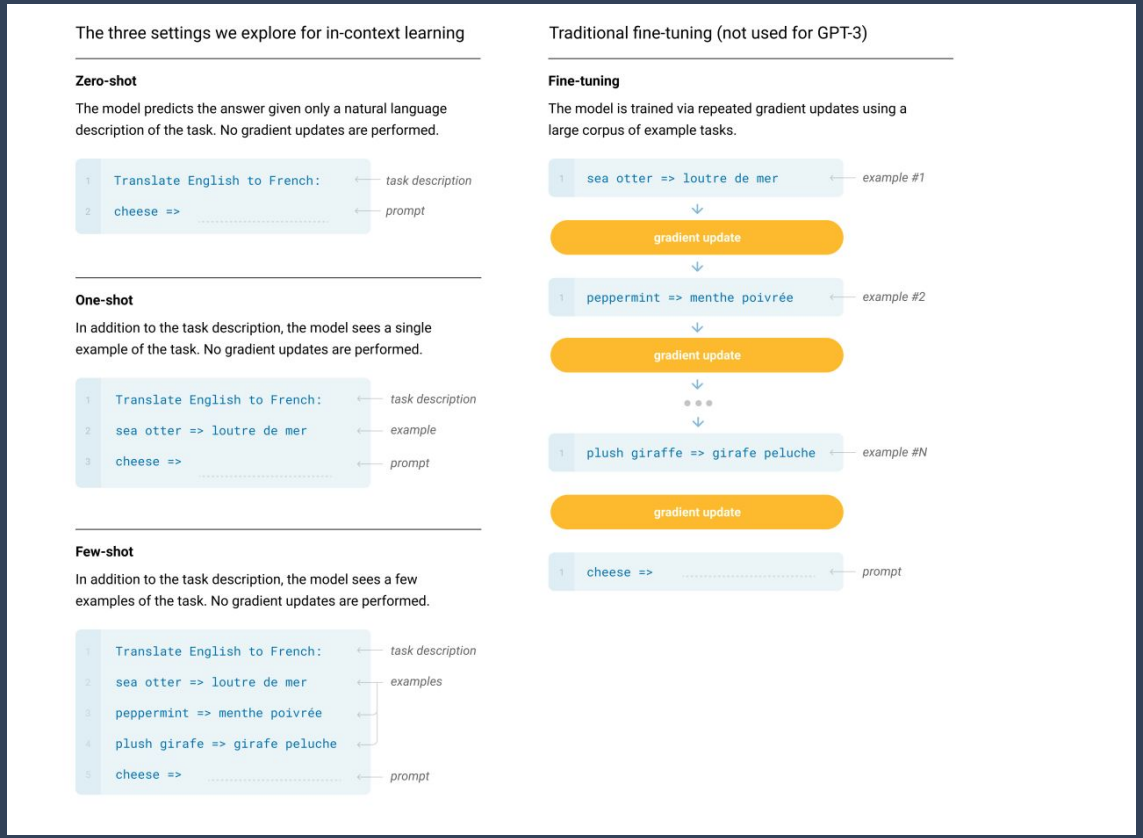


Illustration Source: [Language Models are Few-Shot Learners]

Some Results

One-shot learning from dictionary definitions:

To "screeg" something is to swing a sword at it. An example of a sentence that uses the word screeg is:
We screeghed at each other for several minutes and then we went outside and ate ice cream.

Few-shot language editing:

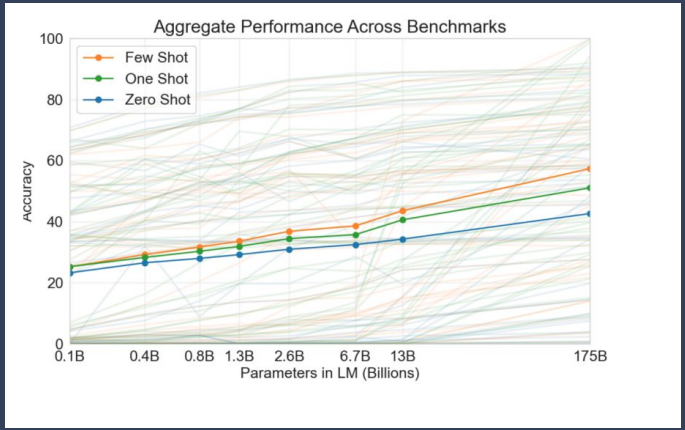
Poor English input: I eated the purple berries.
Good English output: I ate the purple berries.
Poor English input: Thank you for picking me as your designer. I'd appreciate it.
Good English output: Thank you for choosing me as your designer. I appreciate it.
Poor English input: The mentioned changes have done. or I did the alteration that you requested. or I changed things you wanted and did the modifications.
Good English output: The requested changes have been made. or I made the alteration that you requested. or I changed things you wanted and made the modifications.
Poor English input: I'd be more than happy to work with you in another project.
Good English output: I'd be more than happy to work with you on another project.

Poor English input: Please provide me with a short brief of the design you're looking for and that'd be nice if you could share some examples or project you did before.
Good English output: Please provide me with a brief description of the design you're looking for and that would be nice if you could share some examples or projects you have done before.

Non-few-shot learning tasks:

Title: United Methodists Agree to Historic Split
Subtitle: Those who oppose gay marriage will form their own denomination
Article: **After two days of intense debate, the United Methodist Church has agreed to a historic split - one that is expected to end in the creation of a new denomination, one that will be "theologically and socially conservative," according to The Washington Post. The majority of delegates attending the church's annual General Conference in May voted to strengthen a ban on the ordination of LGBTQ clergy and to write new rules that will "discipline" clergy who officiate at same-sex weddings. But those who opposed these measures have a new plan: They say they will form a separate denomination by 2020, calling their church the Christian Methodist**

- Model is impressive but has room to improve



- Model fails in inexplicable ways

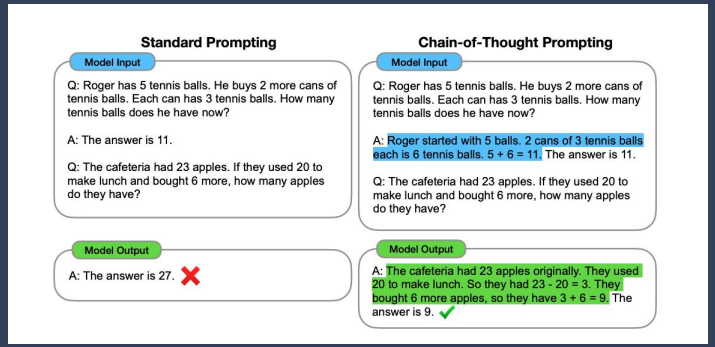
```
Q: How many eyes does a giraffe have?
A: A giraffe has two eyes.

Q: How many eyes does my foot have?
A: Your foot has two eyes.

Q: How many eyes does a spider have?
A: A spider has eight eyes.

Q: How many eyes does the sun have?
A: The sun has one eye.
```

Source: https://lacker.io/ai/2020/07/06/giving-gpt-3-a-turing-test.html

Core insight - We need better choice of  at test time. Better prompts!

Chain-of-thought prompting

- A series of intermediate reasoning steps—significantly improves the ability of large language models to perform complex reasoning.
- A few chain of thought examples are provided as in prompting.



- Handle complex arithmetic, commonsense, and symbolic reasoning tasks

- LLMs offer the prospect of in-context few-shot learning via prompting.
- Decomposes a multi-step layered problem into intermediate steps
- Interpretability

QUESTION: Mike plays ping pong for 40 minutes. In the first 20 minutes, he scores 4 points. In the second 20 minutes, he scores 25% more points. How many total points did he score?

EQUATION ONLY (WRONG ANSWER): (4 + 20 * 0.25) = 6. The answer is 6.

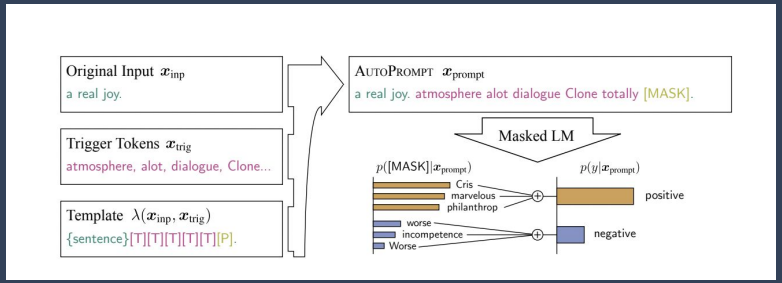
CHAIN OF THOUGHT (CORRECT): Mike played ping pong for 40 minutes. In the first 20 minutes, he scored 4 points. In the second 20 minutes, he scored 25% more points. So he scored 25% more in the second 20 minutes. 4 x 1.25 = 5. So he scored 5 points in the second 20 minutes. So he scored 9 points in total. The answer is 9.

Automatic Prompt Tuning

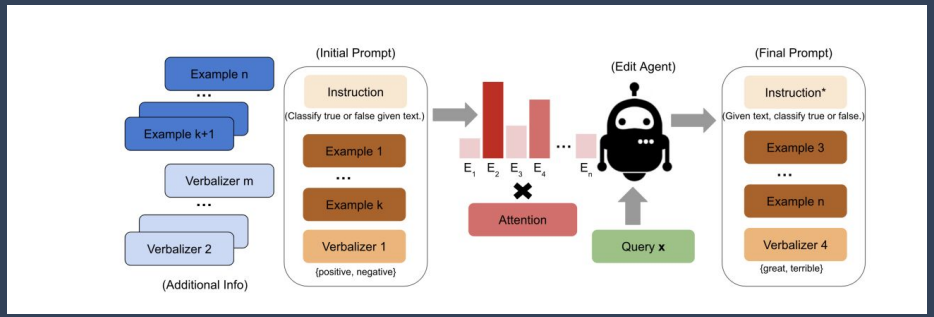
- Task description - needs human involvement
- Response quality is sensitive to task description and exemplars
- Writing prompts in natural language remains a manual trial-and-error process requiring significant human effort
- Limitation - How many exemplars can fit into the context window?
- Can we automate the task description and the exemplars ?

Some methods

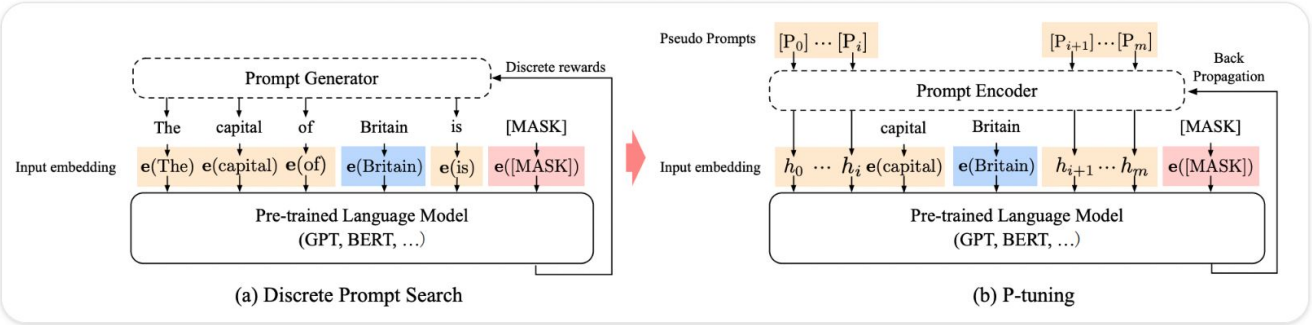
- AutoPrompt



- TEMPERA - Test Time Prompt Editing Via Reinforcement Learning



- Prompt tokens have their own parameters that are updated independently.
- We can keep the pre-trained model's parameters frozen, and only update the gradients of the prompt token embeddings.
- P Tuning → Soft, Work with the encoder embeddings. Automatically search and optimize for better prompts in a continuous space. Differentiable and learnable

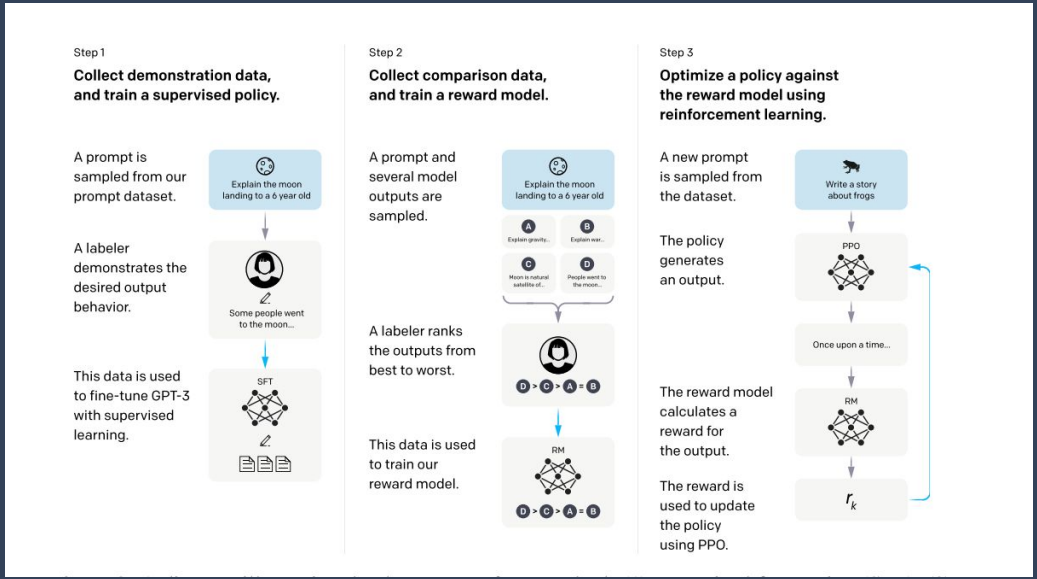


Source - [GPT Understands, Too]

P Tuning Demo

RLHF

- Responses could be meaningless as generation process is probabilistic
- Customer Facing applications should not emit toxic / factually incorrect answers
- Jailbreak LLMs
- Solutions - ?



- Supervised Fine-tuning : Finetune the pre-trained model on instruction dataset
- Learn a reward function: A reward function emits a scalar score given a response and prompt. Human beings rank many responses for a given prompt and do it for a variety of prompts. Reward functions are neural networks and we aim to approximate the ranking methods / preference of humans
- Policy Optimization: Learned reward function is used to guide the generation by reinforcing the decoder to emit high quality response for a given prompt

Reinforcement Learning Terms (Brief Intro)

- Agent
- Environment
- State - State of the world.
- Action - Given a state, a function which yields which action to take or which actions are more likely?
- Policy - Deterministic / Stochastic
- Value Functions - V(s) - What is the worth of a given state? Given a state of a chess game, all my potential moves from here, what payoff I can expect for each of them?
- Reward - A scalar award you get after taking an action a(t) at time step t after being in state s(t)

- Q-Values - Q(s,a) : Payoff I can expect for an action, state tuple
- Learning a Policy - Learning how to take actions by observing rewards associated with a given action and state
- On Policy - Update your course of action real time by looking at a reward signal, state and updating your belief of its payoff
- Off Policy - Learn policy offline and use it to decide your course of action real time
- Model Free - You don't know the model of the system. Example: Helicopter flying. Given its current position and velocity, air condition etc., where will it be next? Hard to model
- Model driven - A chess game. You know all possible next states you can go to from a given state
- PPO (Proximal Policy Optimization) - Online. You observe a reward for your action and change your policy accordingly

- How can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so far that we accidentally cause performance collapse?
- One huge reward should not lead us to a blind alley but we still want to update our beliefs!
- Trust Region Policy Optimization

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}(\theta_k, \theta) \quad \text{s.t. } \bar{D}_{KL}(\theta||\theta_k) \leq \delta$$

$$\mathcal{L}(\theta_k, \theta) = \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right]$$

$$\bar{D}_{KL}(\theta||\theta_k) = \underset{s\sim\pi_{\theta_k}}{\mathrm{E}} \left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right) \right].$$

KL Divergence is calculated on all states visited in the trajectory by the older policy

PPO (RLHF's secret sauce)

$$\theta_{k+1} = \arg\max_{\theta} \mathop{\mathrm{E}}_{s,a\sim\pi_{\theta_k}} \left[L(s, a, \theta_k, \theta) \right],$$

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a),\quad g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right),$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0. \end{cases}$$

- Clipping
- Updated policy does not benefit by going far away from the old policy.
- Updated policy does not benefit by going far away from the old policy.

Pseudocode

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

RLHF from InstructGPT Deconstructed

- Loss Function of the reward model - Deep Learning model parameterized by phi. D: human labeled dataset. y_w is the human label

$$\mathcal{L}_r(\phi) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}[\log \sigma(r_\phi(x,y_w) - r_\phi(x,y_l))]$$

where σ is the sigmoid function.

- Randomly choose a prompt, generate reward for the response, use KL penalty on a token level to keep the effect of reward in check

$$\max_\theta \mathbb{E}[r_\phi(y|x) - \beta\mathbb{D}_{KL}(\pi_\theta^{RL}(y|x)||\pi^{SFT}(y|x))]$$

RLAIF - Scaling Reinforcement Learning from Human Feedback with AI Feedback

- Core Idea - Gathering human labeled data for learning rewards for a wide variety of prompts is costly and time consuming
- How can we automate this?
- Use another LLM to generate labels. Reward model learned on AI preferences!
- Use LLM to generate y given a prompt x. Use softmax to rank the responses so that reward function can be learned

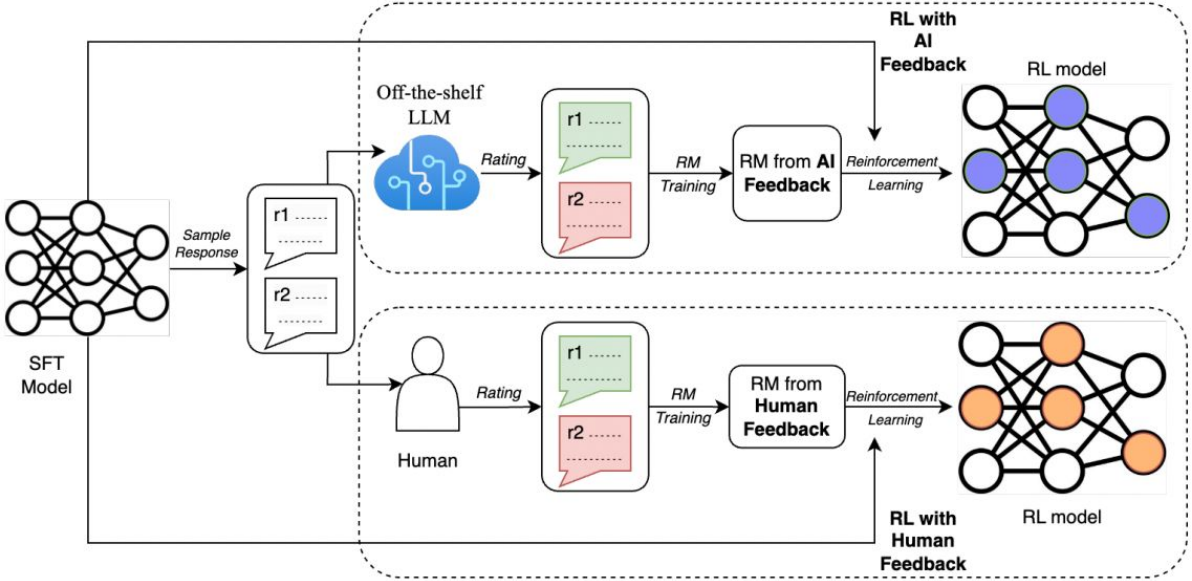
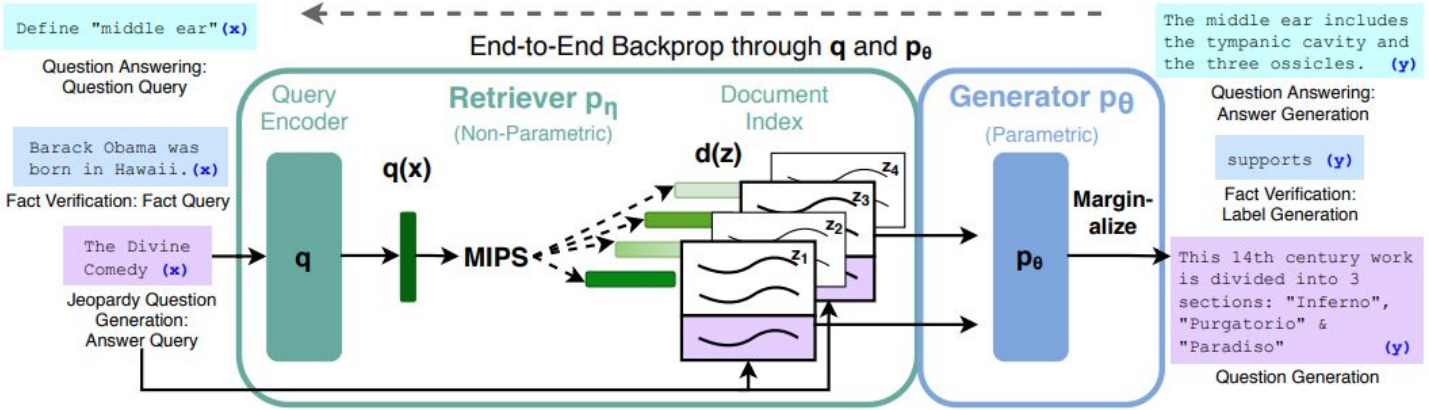


Figure 2: A diagram depicting RLAIF (top) vs. RLHF (bottom)

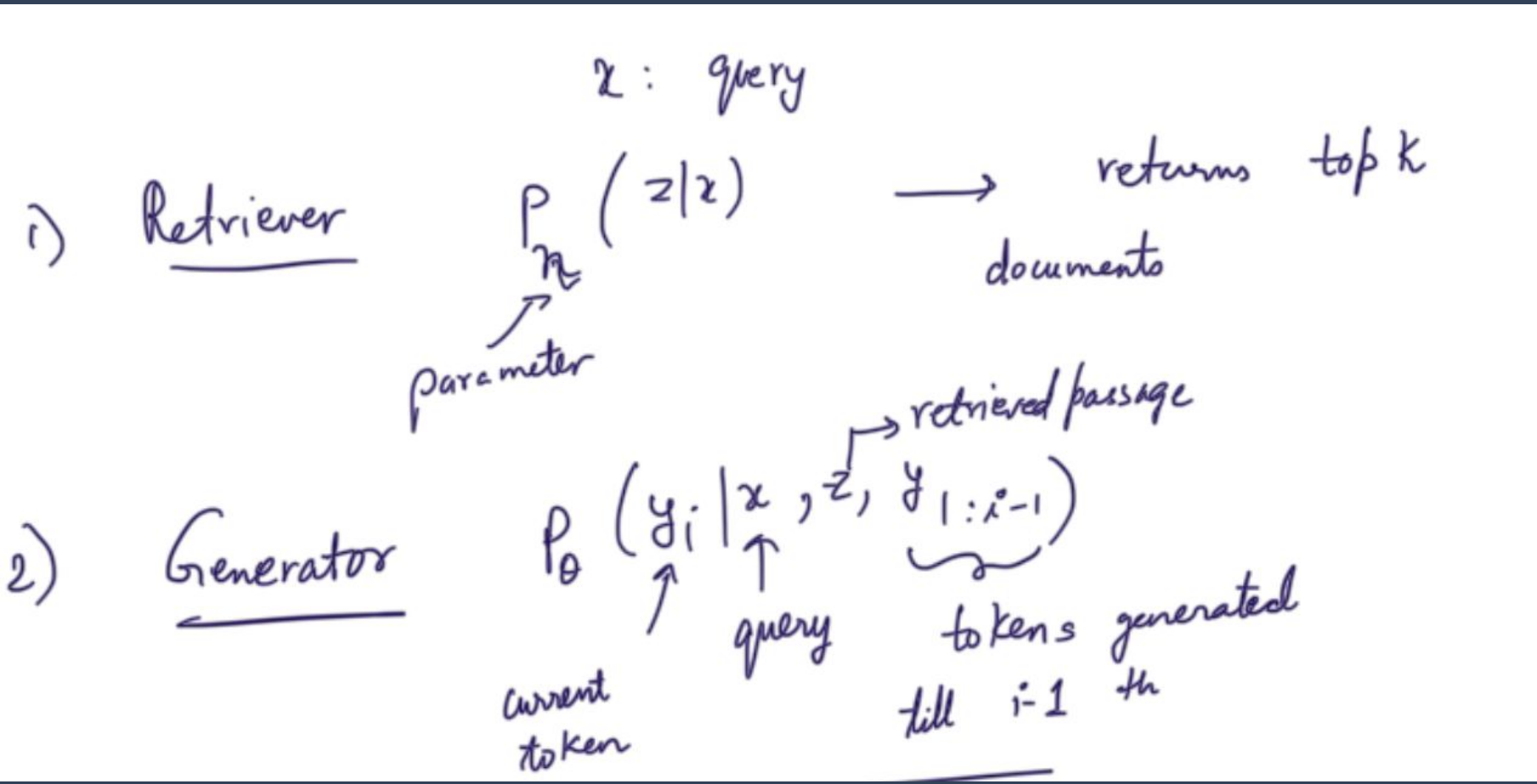
Source - RLAIF

Retrieval Augmented Generation

- Hallucinations
- Pre-trained LLMs don’t revise their memory frequently and hence get stale. How to keep it dynamic?
- Might need to condition on domain context - Example: Recommend suggestions for a given query given what the customer has searched for in the session. Autocomplete ni to nike shoes if the customer has searched for running gears in the session.
- Parametric Seq2Seq combined with non-parametric retrieval to generate using latent context



- Both generator and retriever are first initialized from pre-trained models and then fine-tuned jointly so that both retriever and generator can adapt to downstream tasks
- This is helpful for knowledge intensive tasks where one needs to tap into external knowledge source
- Retrieved document → Latent Variable



How to produce the sequence given the retrieved documents ??

① RAG-Sequence

a) Retrieve top k documents

b) Generator produces the output sequence probability for each document

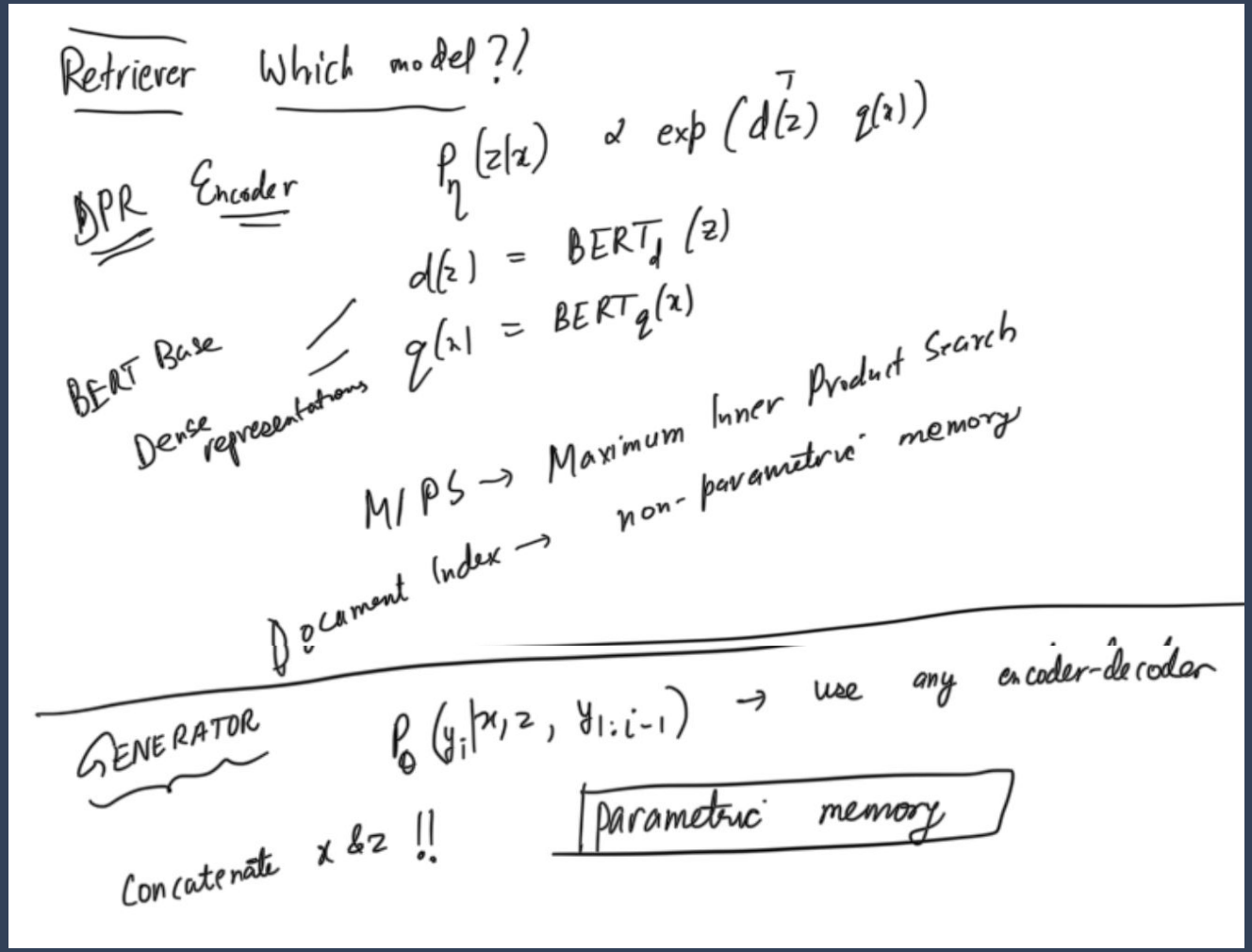
c) Marginalize

$$P_{RAG\text{-}sequence}(y|x) \approx \sum_{z \in top\text{-}k} p_\eta(z|x)\, p_\theta(y|x,z)$$

$$= \sum_{z} p_\eta(z|x) \prod_{i}^{N} p_\theta(y_i|x,z,y_{1:i-1})$$

② RAG-Token

$$p_{RAG\text{-}Token}(y|x) \approx \prod_{i=1}^{N} \sum_{z \in top\text{-}k} p_\eta(z|x)\, p(y_i|x,z,y_{1:i-1})$$



- Keep the document index fixed
- The query encoder is trained along with the generator
- Need supervised training dataset consisting of pairs (x_i, y_i)

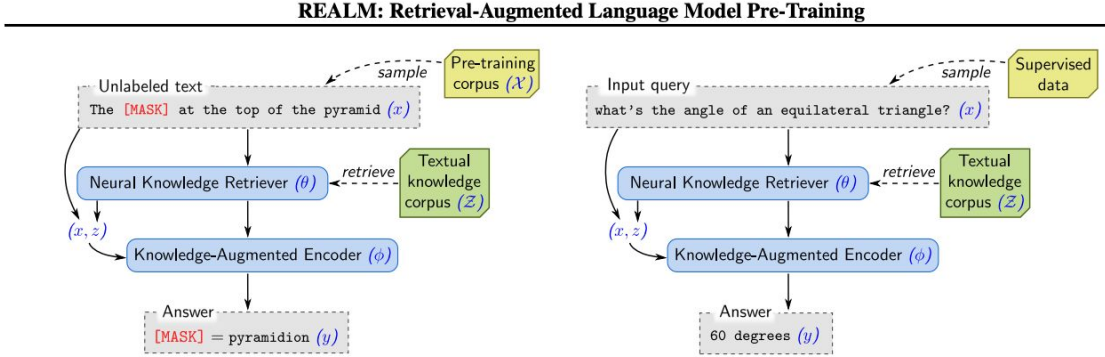
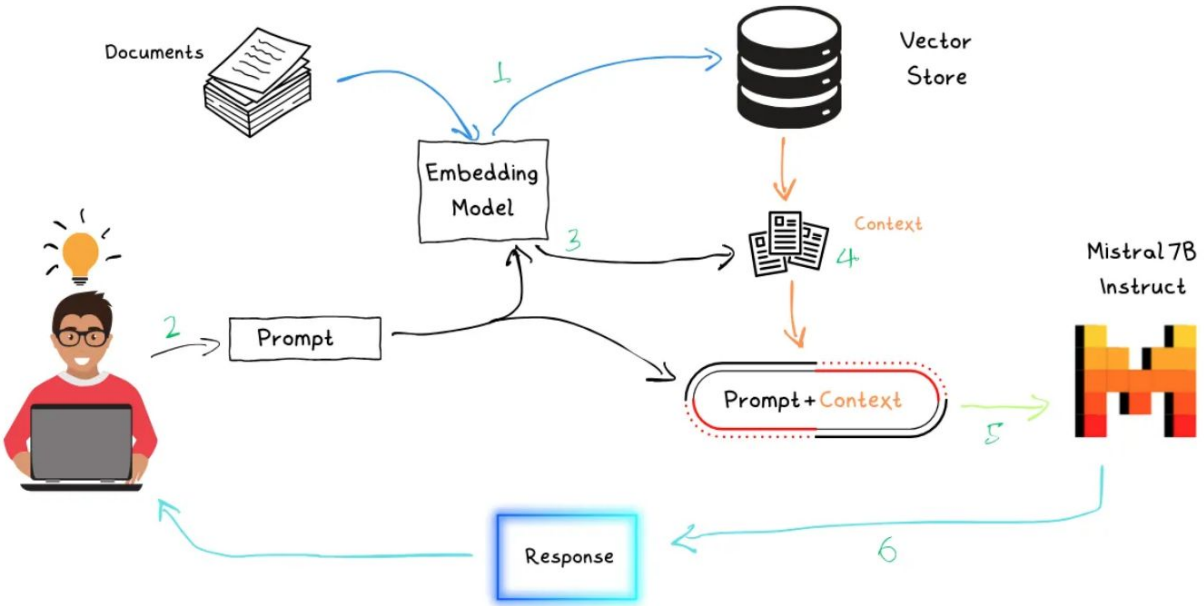


Figure 2. The overall framework of REALM. **Left:** *Unsupervised pre-training.* The knowledge retriever and knowledge-augmented encoder are jointly pre-trained on the unsupervised language modeling task. **Right:** *Supervised fine-tuning.* After the parameters of the retriever (θ) and encoder (ϕ) have been pre-trained, they are then fine-tuned on a task of primary interest, using supervised examples.

Source: [REALM: Retrieval-Augmented Language Model Pre-Training]



RAG Demo

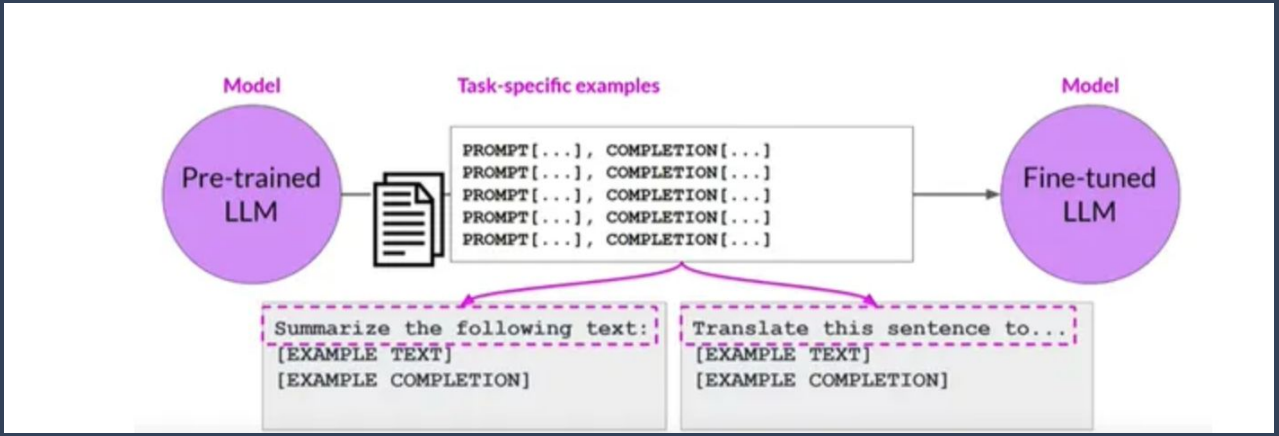
Domain Adaptation through instruction fine-tuning

Finetuning

- Pre-trained models encapsulate world knowledge due to the size and diversity of the data they have been trained on
- Due to this wide exposure to different data, they might not be experts in a given domain
- Can we fine-tune these LLMs to attune them to our advantage?
- Subtleties - a) What data to use to finetune b) In what format to feed the instructions c) How to learn new things without forgetting the old things d) How to fine-tune this massive model efficiently?

Simple finetune demo

Instruction Finetuning



- Training examples are more like directives / instructions
- Remember, in-context learning!
- Task diversity - The instructions could come from different tasks as opposed to regular finetuning which generally has data for 1 task.
- This helps generalize zero shot LLMs to new tasks

Self-Instruct

- Reliance on humans to come up with diverse & rich instructions - bottleneck!
- Can we use AI to generate these instructions for fine-tuning?
- Core idea → Sample task from a task pool and use some other LLM to curate these instructions! Filter out bad instructions and insert into the instruction fine-tuning dataset
- Self-Instruct, Alpaca etc. → Framework for doing this
- A set of 52000 such synthetic instructions is available publicly

```
[
    {
        "instruction": "Give three tips for staying healthy.",
        "input": "",
        "output": "1.Eat a balanced diet and make sure to include plenty of fruits and vegetables. \n2. Exercise regularly to keep your body active and strong. \n3. Get enough sleep and maintain a consistent sleep schedule."
    },
    {
        "instruction": "What are the three primary colors?",
        "input": "",
        "output": "The three primary colors are red, blue, and yellow."
    },
    {
        "instruction": "Describe the structure of an atom.",
        "input": "",
        "output": "An atom is made up of a nucleus, which contains protons and neutrons, surrounded by electrons that travel in orbits around the nucleus. The protons and neutrons have a positive charge, while the electrons have a negative charge, resulting in an overall neutral atom. The number of each particle determines the atomic number and the type of atom."
    },
    {
        "instruction": "How can we reduce air pollution?",
        "input": "",
        "output": "There are a number of ways to reduce air pollution, such as shifting to renewable energy sources, encouraging the use of public transportation, prohibiting the burning of fossil fuels, implementing policies to reduce emissions from industrial sources, and implementing vehicle emissions standards. Additionally, individuals can do their part to reduce air pollution by reducing car use, avoiding burning materials such as wood, and changing to energy efficient appliances."
    },
    {
        "instruction": "Describe a time when you had to make a difficult decision.",
        "input": "",
        "output": "I had to make a difficult decision when I was working as a project manager at a construction company. I was in charge of a project that needed to be completed by a certain date in order to meet the client\u2019s expectations. However, due to unexpected delays, we were not able to meet the deadline and so I had to make a difficult decision. I decided to extend the deadline, but I had to stretch the team\u2019s resources even further and increase the budget. Although it was a risky decision, I ultimately decided to go ahead with it to ensure that the project was completed on time and that the client\u2019s expectations were met. The project was eventually successfully completed and this was seen as a testament to my leadership and decision-making abilities."
    },
    {
        "instruction": "Identify the odd one out.",
        "input": "Twitter, Instagram, Telegram",
        "output": "Telegram"
    },
```


- A list of JSONs looking like the above

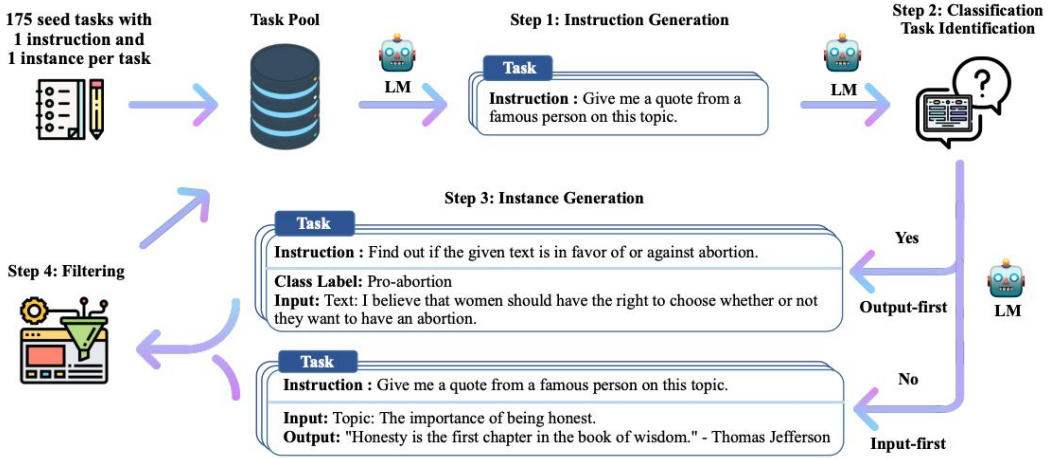


Figure 2: A high-level overview of SELF-INSTRUCT. The process starts with a small seed set of tasks as the task pool. Random tasks are sampled from the task pool, and used to prompt an off-the-shelf LM to generate both new instructions and corresponding instances, followed by filtering low-quality or similar generations, and then added back to the initial repository of tasks. The resulting data can be used for the instruction tuning of the language model itself later to follow instructions better. Tasks shown in the figure are generated by GPT3.

Source - SELF-INSTRUCT: Aligning Language Models with Self-Generated Instructions

PEFT - Parameter Efficient Fine-tuning

- Catastrophic forgetting → While finetuning, the new task erases / perturbs weights learned during fine-tuning
- Pre-trained LLMs are very big in size and hence fine-tuning them efficiently is challenging
- 2 birds with 1 stone - PEFT!
- Helps in retaining previous weights and learns additional small number of parameters to adjust to the task instructions

- In-context learning incurs substantial computational, memory, and storage costs because it involves processing all of the training examples every time a prediction is made

- ICL typically produces inferior performance compared to fine-tuning

- The exact formatting of the prompt (including the wording and ordering of examples) can have significant and unpredictable impact on the model's performance

- PEFT : offers an alternative paradigm where a small set of parameters are trained to enable a model to perform the new task.

- Core components - Adapter modules, sparse updates methods, prompt tuning, prefix tuning etc.

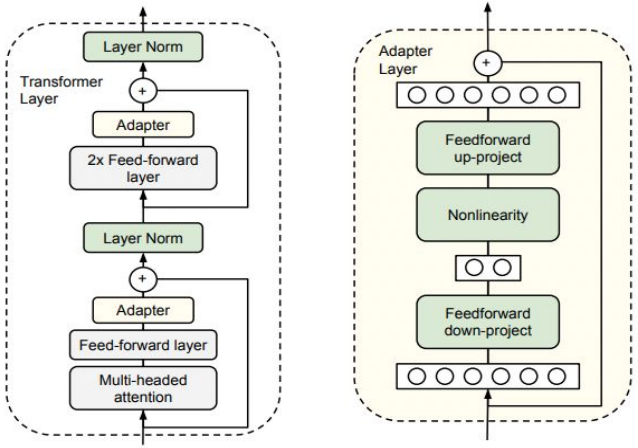


Figure 2. Architecture of the adapter module and its integration with the Transformer. **Left:** We add the adapter module twice to each Transformer layer: after the projection following multi-headed attention and after the two feed-forward layers. **Right:** The adapter consists of a bottleneck which contains few parameters relative to the attention and feedforward layers in the original model. The adapter also contains a skip-connection. During adapter tuning, the green layers are trained on the downstream data, this includes the adapter, the layer normalization parameters, and the final classification layer (not shown in the figure).

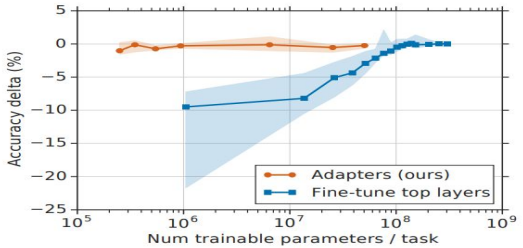


Figure 1. Trade-off between accuracy and number of trained task-specific parameters, for adapter tuning and fine-tuning. The y-axis is normalized by the performance of full fine-tuning, details in Section 3. The curves show the 20th, 50th, and 80th performance percentiles across nine tasks from the GLUE benchmark. Adapter-based tuning attains a similar performance to full fine-tuning with two orders of magnitude fewer trained parameters.

Source: [Parameter-Efficient Transfer Learning for NLP]

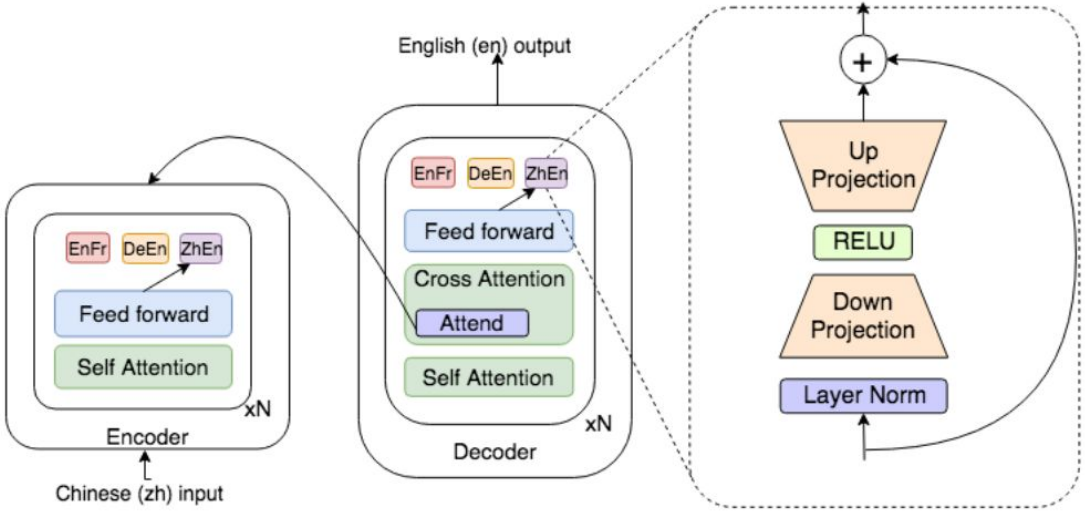


Figure 1: Diagrams depicting (i) Left: the proposed layout of a Transformer enhanced with language specific adapters (ii) Right: the architecture of each residual adapter layer. While the figure depicts use of adapters for multilingual NMT, the same formulation can be used for domain adaptation.

Source: [Simple, Scalable Adaptation for Neural Machine Translation]

Sparse Masks

- Stochastic gradient descent updates all of the model's parameters at each iteration during training → Costly!
- Is it possible to induce a fixed sparse mask on the model's parameters that selects a subset to update over many iterations?
- Construct the mask out of the k parameters with the largest Fisher information as a simple approximation as to which parameters are most important for the task at hand

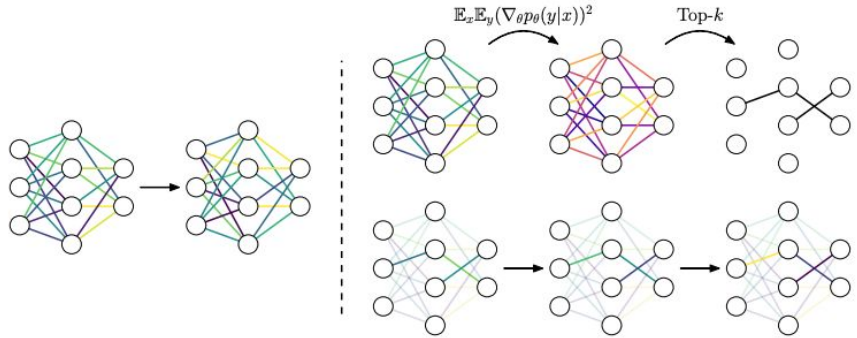


Figure 1: Diagram comparing our proposed method to standard SGD. In traditional gradient-based training (left), all of a model's parameters are updated at every iteration. We propose FISH Mask, a method for precomputing a sparse subset of parameters to update over many subsequent training iterations. To construct the FISH Mask, we find the k parameters with the largest Fisher information (right, top). Then, we train the model with traditional gradient descent, but only update those parameters chosen by the mask (right, bottom).

Source: [Training Neural Networks with Fixed Sparse Masks]

- pre-compute a sparse subset of existing parameters to update and keeping the subset fixed over many iterations of training

- Perturb x by a small amount and measure the difference in distribution

$$\mathbb{E}_x \mathrm{D}_{\mathrm{KL}}(p_\theta(y|x) \parallel p_{\theta+\delta}(y|x)) = \delta^{\mathrm{T}} F_\theta \delta + O(\delta^3)$$

- Fisher Information Matrix

$$F_\theta = \mathbb{E}_{x \sim p(x)} \left[\mathbb{E}_{y \sim p_\theta(y|x)} \nabla_\theta \log p_\theta(y|x) \nabla_\theta \log p_\theta(y|x)^{\mathrm{T}} \right]$$

- In practice, F becomes infeasible to compute as it is square matrix (theta x theta). Approximate to a vector by sampling N points! How much a parameter influences is the entry (positively correlated)!

$$\hat{F}_\theta = \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{y \sim p_\theta(y|x_i)} (\nabla_\theta \log p_\theta(y|x_i))^2$$

LoRA - Low Rank Adaptation

- Freeze pre-trained weights at each layer and add trainable low rank decomposition matrices at each layer
- This greatly reducing the number of trainable parameters for downstream tasks
- Compared to GPT-3 175B fine-tuned with Adam, LoRA can reduce the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times.
- No extra inference latency like adapters

- The learned over-parametrized models in fact reside on a low intrinsic dimension.
- SGD

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

- Pre-trained parameters: Φ_0
- Task specific parameter increment - $\Delta\Phi(\Theta)$, is encoded by a much smaller-sized set of parameters Θ. $|\Theta| \ll |\Phi_0|$.

$$\max_{\Theta} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(p_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t})\right)$$

- Constrain the weight change to be in lower dimension subspace (not in the same space as pre-trained weights)

$$W_0 + \Delta W = W_0 + BA,$$

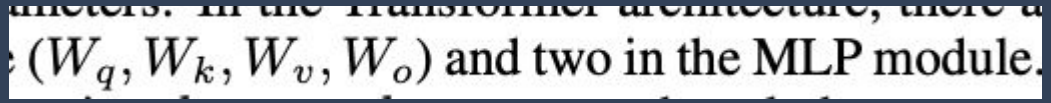
- $W_0 \in \mathbb{R}^{d \times k}$

$B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}$, and the rank $r \ll \min(d, k)$.

- During training, W_{0} is frozen and does not receive gradient updates, while A and B contain trainable parameters.

- Modified forward pass - $$h = W_0 x + \Delta W x = W_0 x + BAx$$

- Initialize A and B with gaussian matrices such that BA = 0 before training
- Transformer parameters



- Only adapting the attention weights and freeze the MLP during fine-tuning

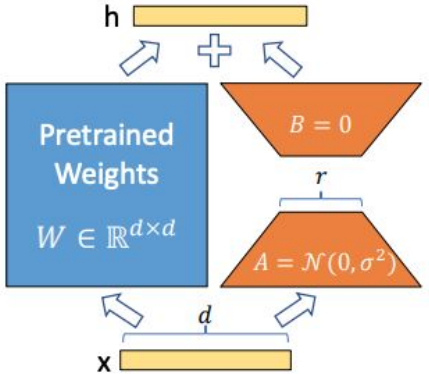


Figure 1: Our reparametrization. We only train A and B.

Source: LORA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS

DEMO

Training LLMs and Deploying them in production

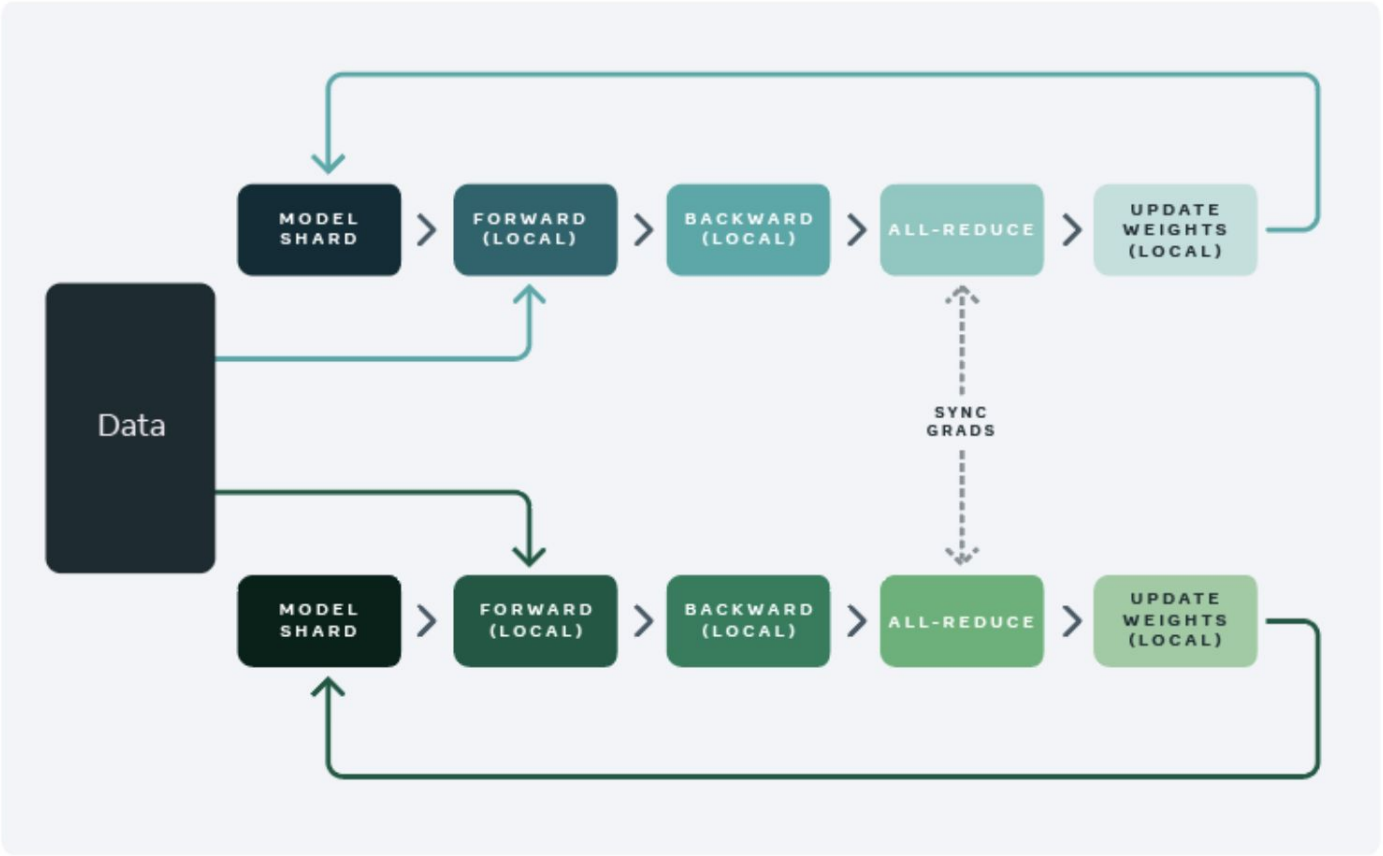
Practical challenges while deploying

- LLMs are typically very large (number of parameters is huge). Forward propagation will take longer compared to modestly sized models
- Attention calculation → Quadratic in sequence length
- Models are too big to fit on 1 GPU.
- Training such large models on internet-sized data - Challenging & costly!
- Serving these models - even harder!
- Controlling the staleness, bias, toxicity, meaning of outputs

Training very, very large models on large data

- Store layers on different GPUs
- Layers on 1 GPU will train on its own batches of data and periodically GPUs will aggregate weights
- Optimize network communication
- Minimize GPU idle time by structuring input tensors accordingly

Standard data parallel training



Distributed Training Strategies - Data Parallelism

- Traditional data parallelism - a per-GPU copy of a model's parameters, gradients and optimizer states
- FSDP (Fully Sharded Data Parallelism) - Shard model's parameters, gradients and optimizer states across data-parallel workers and can optionally offload the sharded model parameters to CPUs.

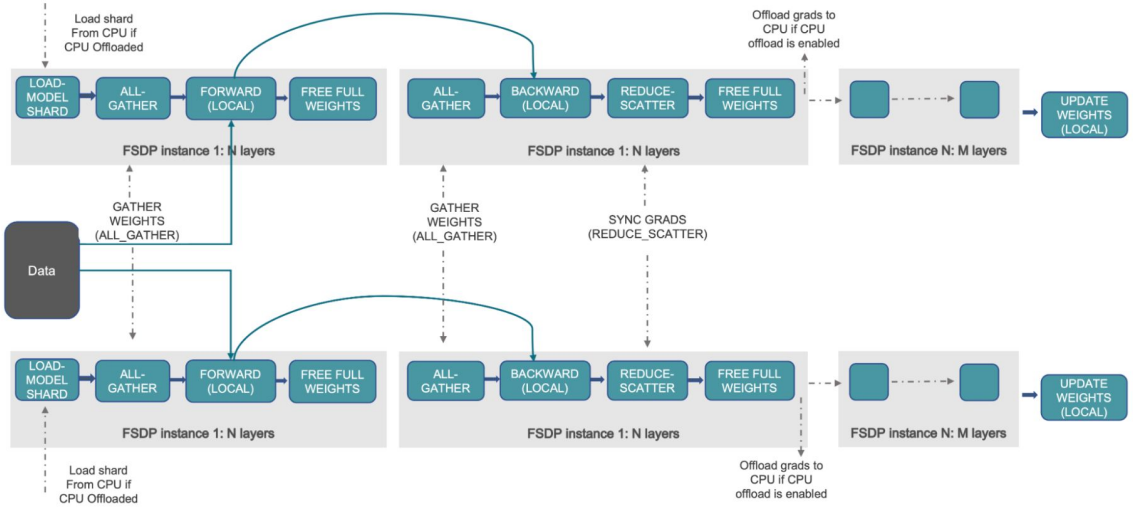
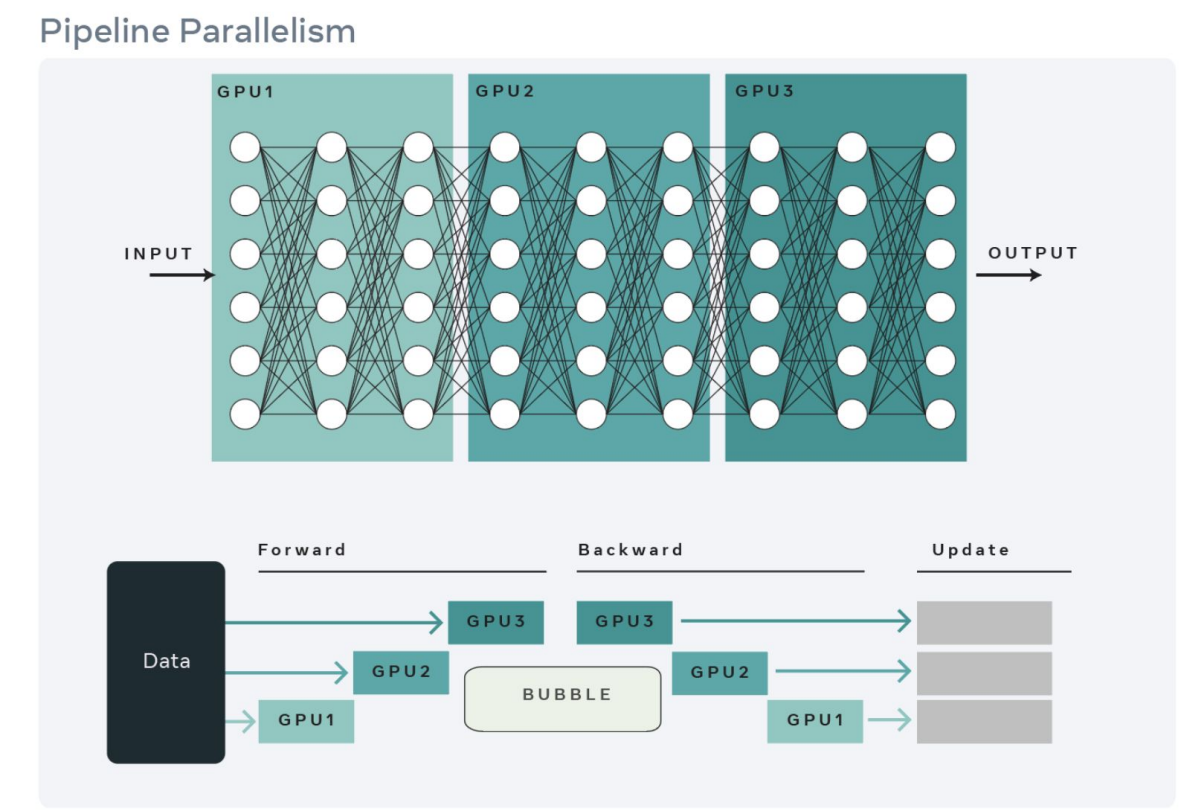


Figure 1. FSDP workflow

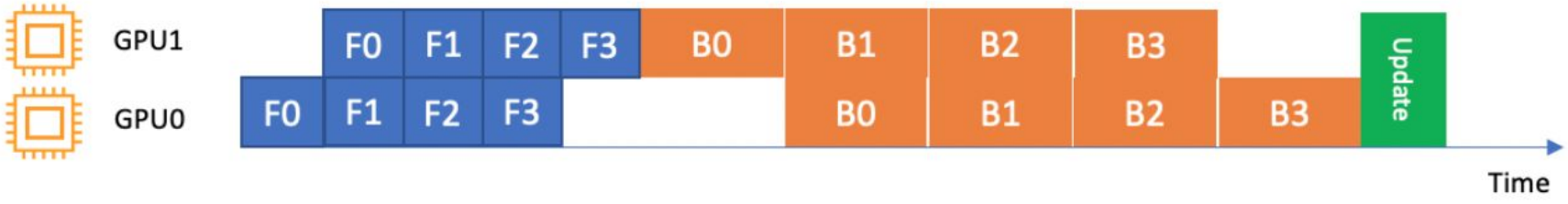
Pipeline Parallelism



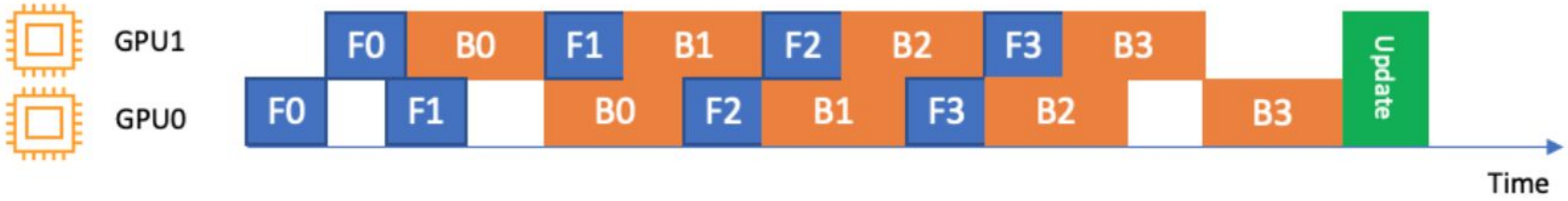
- first shard the model across different devices where each device hosts a shard of the model
- A shard can be a single layer or a series of layers
- Split a mini-batch of data into micro-batches and feeds it to the device hosting the first shard
- The layers on each device process the micro-batches and send the output to the following shard/device.
- By pipelining, reduce the idle time of devices
- Choice of size of micro-batches can affect GPU utilization. Large vs small?

- Sharding the model can also impact GPU utilization where layers with heavier computation can slow down the shards downstream.

Simple schedule - first run all forward prop before running backward prop

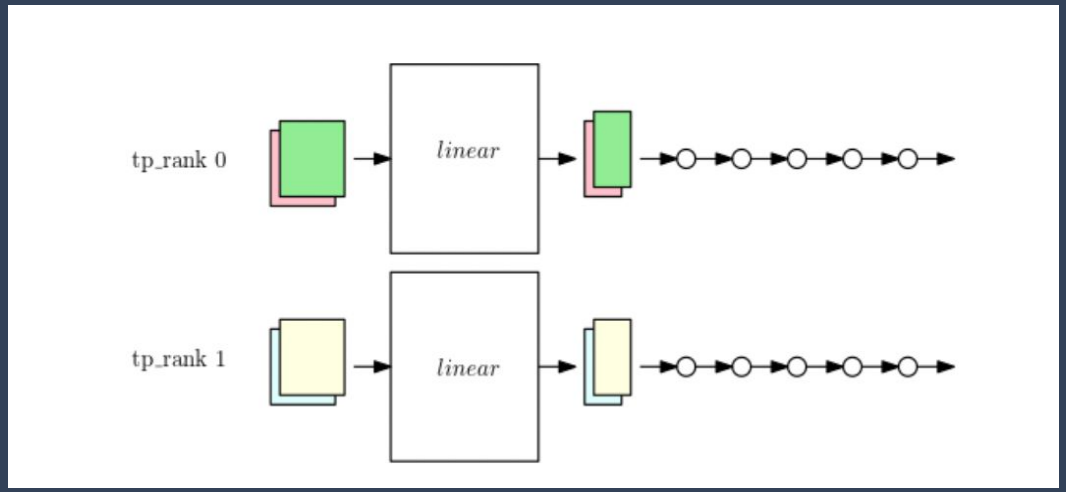


Interleaved schedule - backward execution of the microbatches is prioritized whenever possible.

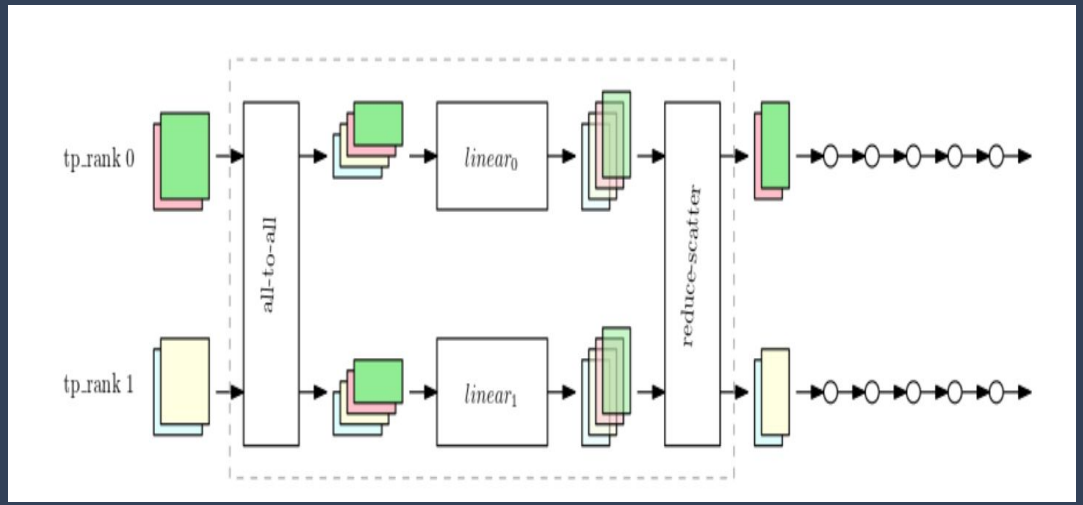


Tensor Parallelism

- When pipeline parallelism is not enough!
- Required when a single parameter consumes most of the GPU memory (such as large embedding tables with a large vocabulary size or a large softmax layer with a large number of classes). In this case, treating this large tensor or operation as an atomic unit is inefficient and impedes balance of the memory load.
- Split weights too! For very large attention matrices, vocabularies → calculation will benefit from optimally placing entries on the gpu in a distributed manner
- Useful for extremely large models in which a pure pipelining is simply not enough. For GPT-3 sized models, a pure microbatch pipelining is inefficient because the pipeline depth becomes too high and the overhead becomes prohibitively large.



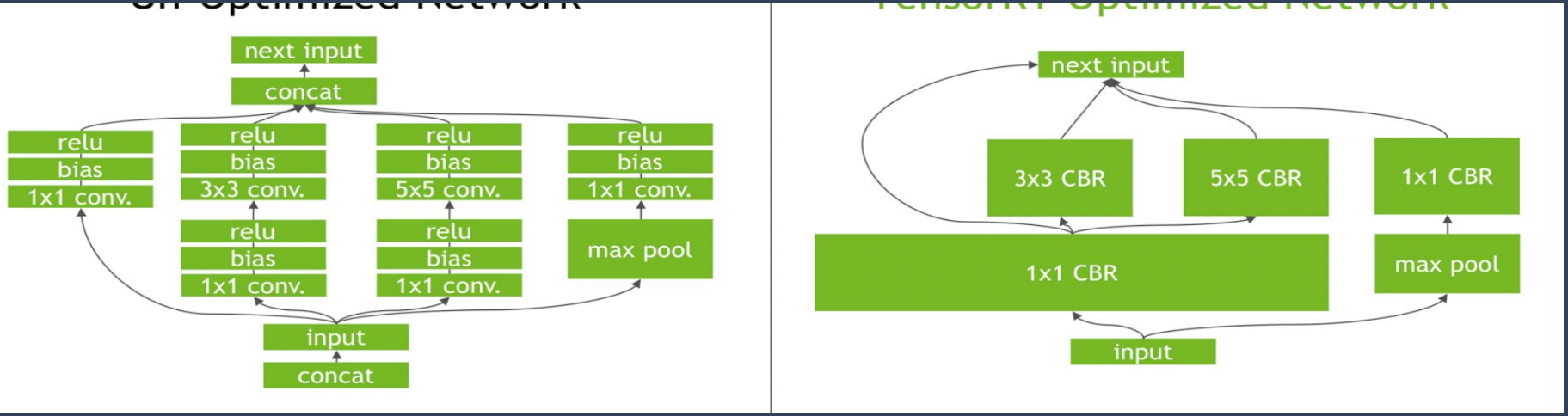
Data Parallelism: Small Model with a large nn.Linear module



Tensor Parallelism: Large Model with nn.Linear module split across tensor ranks

Faster Inferencing

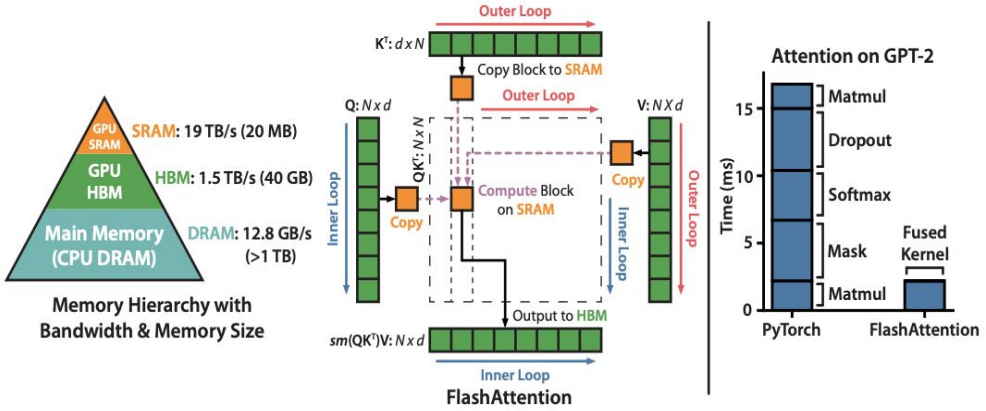
- Support high throughput, low latency, power efficient inference by the deployed models
- Use optimized hardware -> more memory, GPUs with more parallelism, better networking, optimized kernels
- Partition model parameters using model parallelism techniques to leverage the memory of multiple GPUs for inference
- Optimized Kernels - pick the implementation from a library of kernels that delivers the best performance for the target GPU, input data size, filter size, tensor layout, batch size and other parameters.
- Use quantization aware training (meaning, weights can stored with slightly less number of significant digits). This reduces the memory footprint of the model and also speeds up the calculation
- Reduce memory footprint and improve memory reuse by designating memory for each tensor only for the duration of its usage, avoiding memory allocation overhead for fast and efficient execution.



- Tensor and layer kernel fusion - On the left is the suboptimal computation graph and on the right is the optimized graph. Vertical fusion → sequential calculations
- Layer fusion reduces kernel launches and avoids writing into and reading from memory between layers
- Smaller, faster and more efficient graph with fewer layers and kernel launches
- Once the model is fully trained, inference computations can use half precision FP16 or even INT8 tensor operations, since gradient backpropagation is not required for inference. Using lower precision results in smaller model size, lower memory utilization.

Other ingenuities - Flash Attention

- Transformers are memory inefficient and have trouble handling long sequences. Self attention is quadratic
- Make attention algorithms IO aware—accounting for reads and writes between levels of GPU memory
- IO-aware exact attention algorithm that uses tiling to reduce the number of memory reads/writes between GPU high bandwidth memory (HBM) and GPU on-chip SRAM



- HBM - High Bandwidth memory
- SRAM - On chip Bottlenecks when frequent memory accesses
- HBM some orders of magnitude faster and many orders of magnitude smaller

- FlashAttention loops through blocks of the K and V matrices and loads them to fast on-chip SRAM
- In each block, FlashAttention loops over blocks of Q matrix (blue arrows), loading them to SRAM, and writing the output of the attention computation back to HBM.
- Speedup over the PyTorch implementation of attention on GPT-2. FlashAttention does not read and write the large $N \times N$ attention matrix to HBM, resulting in an 7.6× speedup on the attention computation.

- GPUs have a massive number of threads to execute an operation (called a kernel). Each kernel loads inputs from HBM to registers and SRAM, computes, then writes outputs to HBM.
- Compute-bound -> Computation costlier. Matrix multiplications, Convolution etc.
- Memory-bound -> Element-wise operations like dropout, reduction operations (softmax, pooling etc.). Time spent in memory accesses
- Kernel fusion: if there are multiple operations applied to the same input, the input can be loaded once from HBM, instead of multiple times for each operation. Compilers provide kernel fusion for element-wise operations
- But compilers might not be optimized to carry out efficient forward prop and backward prop

$$\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$$

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d},$$

attention and of our method (FLASHATTENTION).

Algorithm 0 Standard Attention Implementation

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

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

in Appendix B). This avoids repeatedly reading and writing of inputs and outputs from and to HBM.

Algorithm 1 FlashAttention

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

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

Theorem 2. *Let N be the sequence length, d be the head dimension, and M be size of SRAM with $d \le M \le Nd$. Standard attention (Algorithm 0) requires $\Theta(Nd + N^2)$ HBM accesses, while* FlashAttention *(Algorithm 1) requires $\Theta(N^2 d^2 M^{-1})$ HBM accesses.*

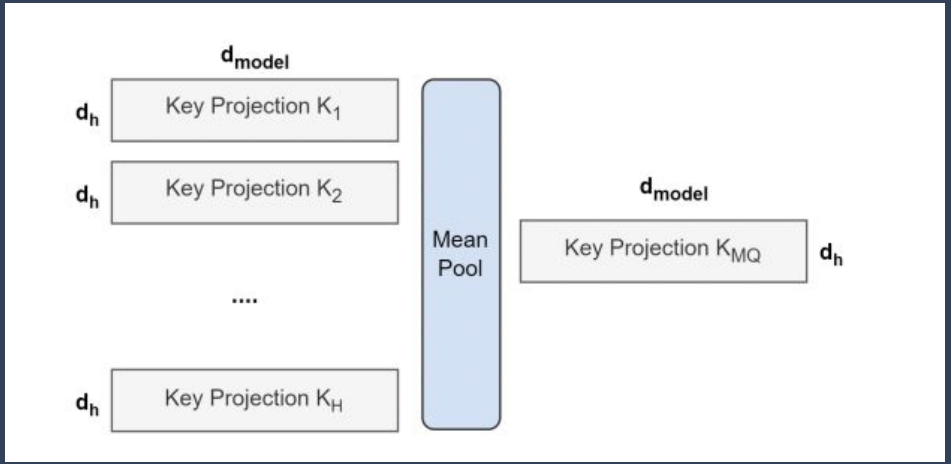
Key-Value Cache

- Happens in the decoder part
- Since the decoder is causal (i.e., the attention of a token only depends on its preceding tokens), at each generation step we are recalculating the same previous token attention, when we actually just want to calculate the attention for the new token.
- By caching the previous Keys and Values, we can focus on only calculating the attention for the new token.
- Faster inference due to faster decoding!

Multi-Query Attention

- Same large K, V will be loaded for all queries otherwise

- Multi-head attention consists of multiple attention layers (heads) in parallel with different linear transformations on the queries, keys, values and outputs. Multi-query attention is identical except that the different heads share a single set of keys and values.



Grouped Query Attention

- Multi-query attention (MQA) can lead to quality degradation and training instability, and it may not be feasible to train separate models optimized for quality and inference

- Grouped-query attention (GQA) is an interpolation of multi-head attention (MHA) and multi-query attention (MQA). It divides query heads into G groups, each of which shares a single key head and value head. This strikes a balance between the speed of MQA and the quality of MHA.

- For large models, more efficient than MHA

- More parallelizable than MQA

- More robust to noise than MQA

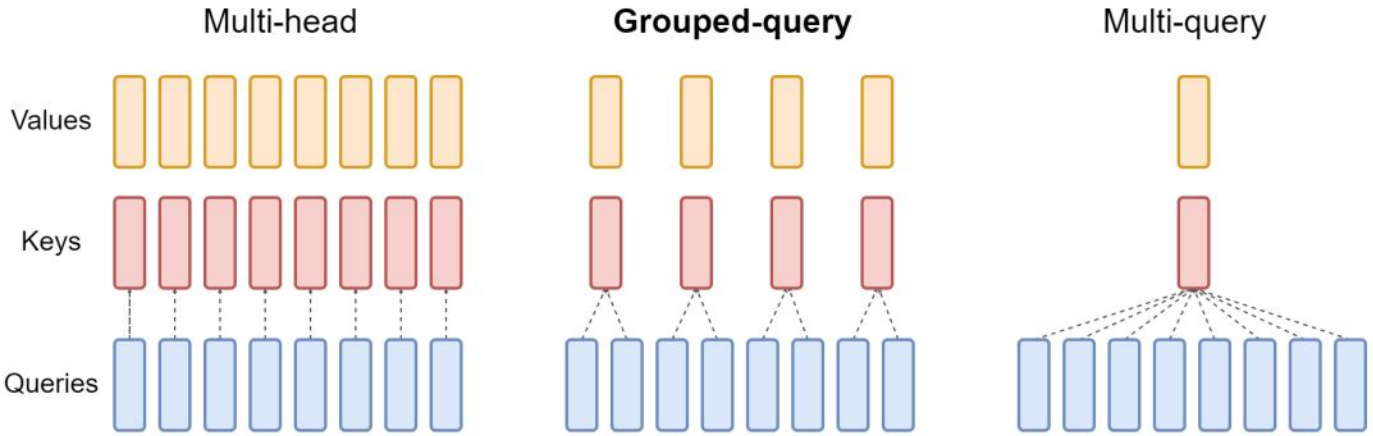


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

Source: [GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints]

HyperAttention

- Near linear time

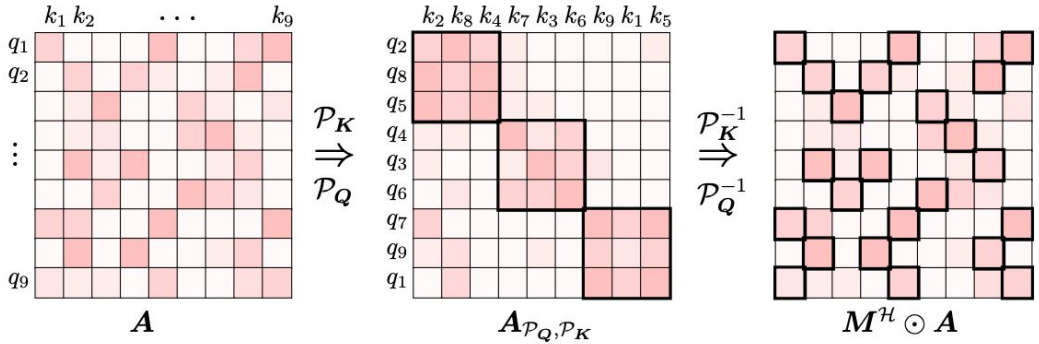


Figure 1: How sortLSH finds large entries of $\boldsymbol{A}$: (Left) Keys and queries undergo hashing using the Hamming ordered LSH $\mathcal{H}(\cdot)$. (Middle) Keys and queries are rearranged based on their hash buckets. Attention matrix after applying these row and column permutations is denoted as $\boldsymbol{A}_{\mathcal{P}_Q,\mathcal{P}_K}$. Large entries of $\boldsymbol{A}_{\mathcal{P}_Q,\mathcal{P}_K}$ are concentrated around the diagonal blocks. (Right) rows and columns permutations are reversed on the attention matrix and $\boldsymbol{M}^{\mathcal{H}} \odot \boldsymbol{A}$ is highlighted.

- First find out large entries in K, V matrices, then rearrange

$$A := \exp\left(QK^\top\right)$$

$$\mathbf{Att} = D^{-1}AV$$

Algorithm 1: sortLSH for locating large entries of A

input: matrices $Q, K \in \mathbb{R}^{n \times d}$, and block size b
Let $\mathcal{H}(\cdot)$ be a Hamming sorted LSH as per Definition 1 and hash rows of Q, K
Let $\mathcal{P}_K, \mathcal{P}_Q \in \mathrm{Sym}(n)$ be permutations satisfying $\mathcal{P}_K(i) < \mathcal{P}_K(j)$ if $\mathcal{H}(K_{i,:}) \leq \mathcal{H}(K_{j,:})$ and $\mathcal{P}_Q(i) < \mathcal{P}_Q(j)$ if $\mathcal{H}(Q_{i,:}) \leq \mathcal{H}(Q_{j,:})$
return Mask matrix $M^{\mathcal{H}} \in \{0,1\}^{n \times n}$ defined as $M^{\mathcal{H}}_{i,j} = \mathbf{1}_{\{\lfloor \mathcal{P}_Q(i)/b \rfloor = \lfloor \mathcal{P}_K(j)/b \rfloor\}}$

- Using rearranged A after applying LSH, obtain a mask matrix M using Algorithm 1
- Quickly approximate V and softmax matrix using another matrix S which has very few number of rows m. S is of size m x n

Source: [HyperAttention: Long-context Attention in Near-Linear Time]

Metrics - How good is the generation?

BertScore

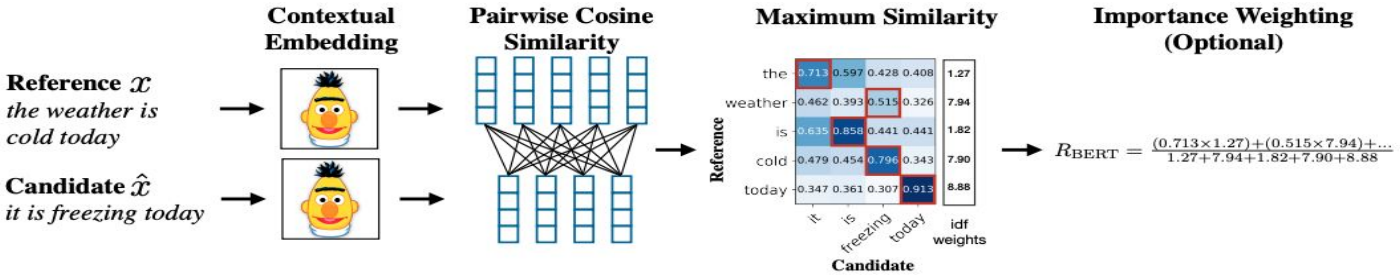


Figure 1: Illustration of the computation of the recall metric R_{BERT}. Given the reference x and candidate $\hat{x}$, we compute BERT embeddings and pairwise cosine similarity. We highlight the greedy matching in red, and include the optional idf importance weighting.

$$R_{\text{BERT}} = \frac{1}{|x|}\sum_{x_i \in x} \max_{\hat{x}_j \in \hat{x}} \mathbf{x}_i^\top \hat{\mathbf{x}}_j \quad , \quad P_{\text{BERT}} = \frac{1}{|\hat{x}|}\sum_{\hat{x}_j \in \hat{x}} \max_{x_i \in x} \mathbf{x}_i^\top \hat{\mathbf{x}}_j \quad , \quad F_{\text{BERT}} = 2\frac{P_{\text{BERT}} \cdot R_{\text{BERT}}}{P_{\text{BERT}} + R_{\text{BERT}}} \quad .$$

$$R_{\text{BERT}} = \frac{\sum_{x_i \in x} \text{idf}(x_i) \max_{\hat{x}_j \in \hat{x}} \mathbf{x}_i^\top \hat{\mathbf{x}}_j}{\sum_{x_i \in x} \text{idf}(x_i)} \quad .$$

Source: BERTSCORE: EVALUATING TEXT GENERATION WITH BERT

Mauve

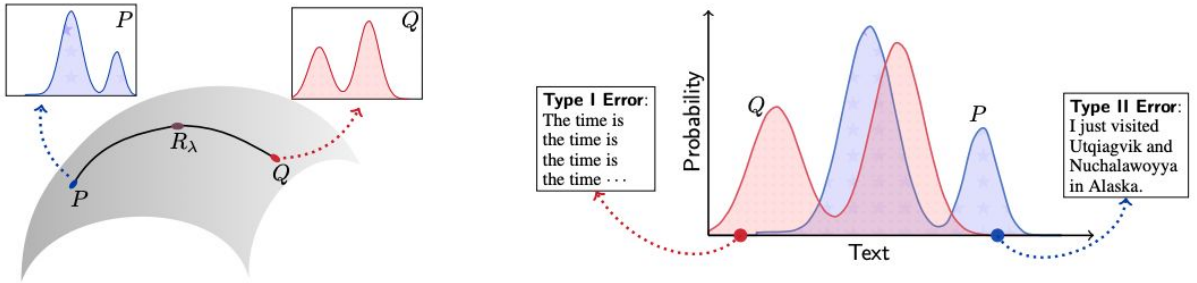


Figure 1: **Left**: MAUVE compares the machine text distribution Q to that of human text P by using the family of mixtures $R_\lambda = \lambda P + (1-\lambda)Q$ for $\lambda \in (0, 1)$. **Right**: Illustration of *Type I errors*, where Q produces degenerate, repetitive text which is unlikely under P, and, *Type II errors*, where Q cannot produce plausible human text due to truncation heuristics [26]. MAUVE measures these errors softly, by using the mixture distribution R_λ. Varying λ in $(0, 1)$ gives a divergence curve and captures a spectrum of soft Type I and Type II errors. MAUVE summarizes the entire divergence curve in a single scalar as the area under this curve.

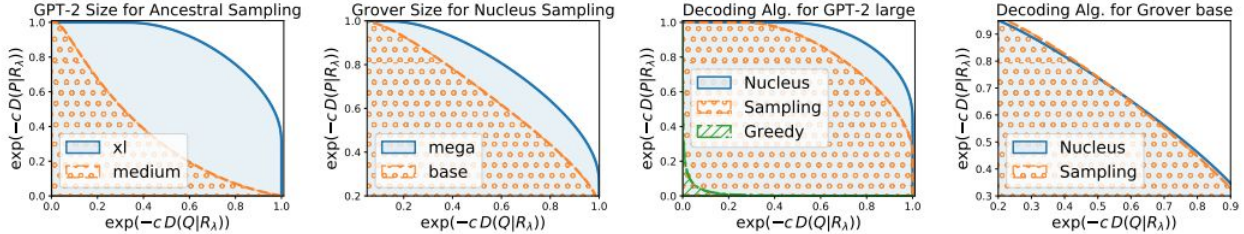


Figure 2: Divergence curves for different models (GPT-2 [45], Grover [61]) and decoding algorithms (greedy decoding, ancestral and nucleus sampling). MAUVE is computed as the area of the shaded region, and larger values of MAUVE indicate that Q is closer to P. In general, MAUVE indicates that generations from larger models and nucleus sampling are closer to human text. **Rightmost**: Nucleus sampling has a slightly smaller Type I error than ancestral sampling but a higher Type II error, indicating that ancestral sampling with Grover base produces more degenerate text while nucleus sampling does not effectively cover the human text distribution.

$$\mathcal{C}(P,Q) = \left\{ \left(\exp(-c\,\mathrm{KL}(Q|R_\lambda)), \exp(-c\,\mathrm{KL}(P|R_\lambda)) \right) \,:\, R_\lambda = \lambda P + (1-\lambda)Q,\ \lambda \in (0,1) \right\}, \tag{1}$$

- Area under the divergence curve C(P, Q)

Source: [MAUVE: Measuring the Gap Between Neural Text and Human Text using Divergence Frontiers]