AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

श्रद्धावान् लभते ज्ञानम्

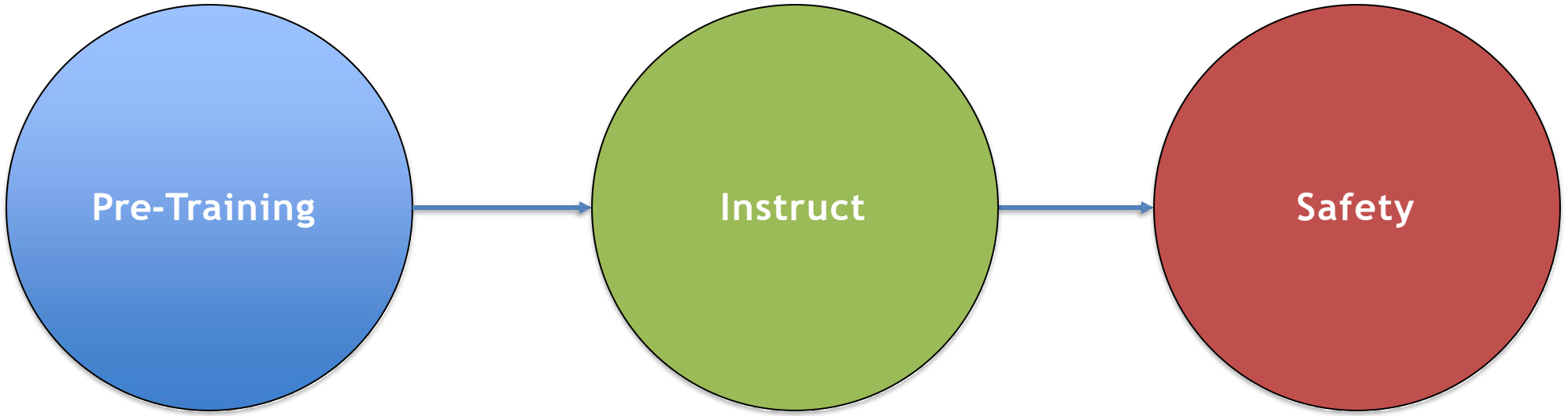
Fine Tuning

Outline

- Training Cycle – LLM
- Instruction-tuning
  - Full Parameter
  - PEFT
    - **Additive**
      - Adapters –
        - Sparse Adapters-IA3
      - Soft prompt-Prompt Tuning-Prefix Tuning-P-Tuning-LlaMA Adapter
    - **Selective**
      - BitFit-Freeze and Reconfigure
    - **Reparameterization** (LORA,QLORA)
    - **Hybrid method** that is a combination of Reparameterization and Selective

Training Cycle - LLM

The training cycle for a LLM consists of 3 main stages:



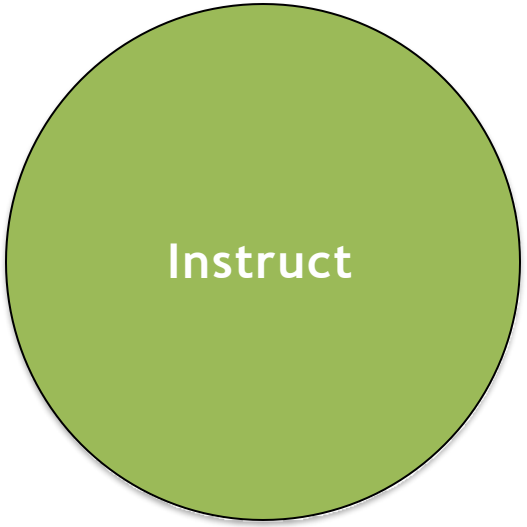
Training Cycle - LLM

Objective:
The goal of pre-training is to teach the model general language understanding.

Process:
The model is trained on a massive dataset of text from the internet and other sources.

Outcome:
A base model that has a general understanding of the language.

Training Cycle - LLM



Objective:
The goal is to make the model useful for specific tasks and improving its ability to follow instructions.

Process:
Fine-tuning the model on datasets that contain instructions and the desired outputs.

Outcome:
A model that becomes better at interpreting and following user instructions.

Training Cycle - LLM

Objective:
The goal is to make sure that the model outputs are safe and ethical.

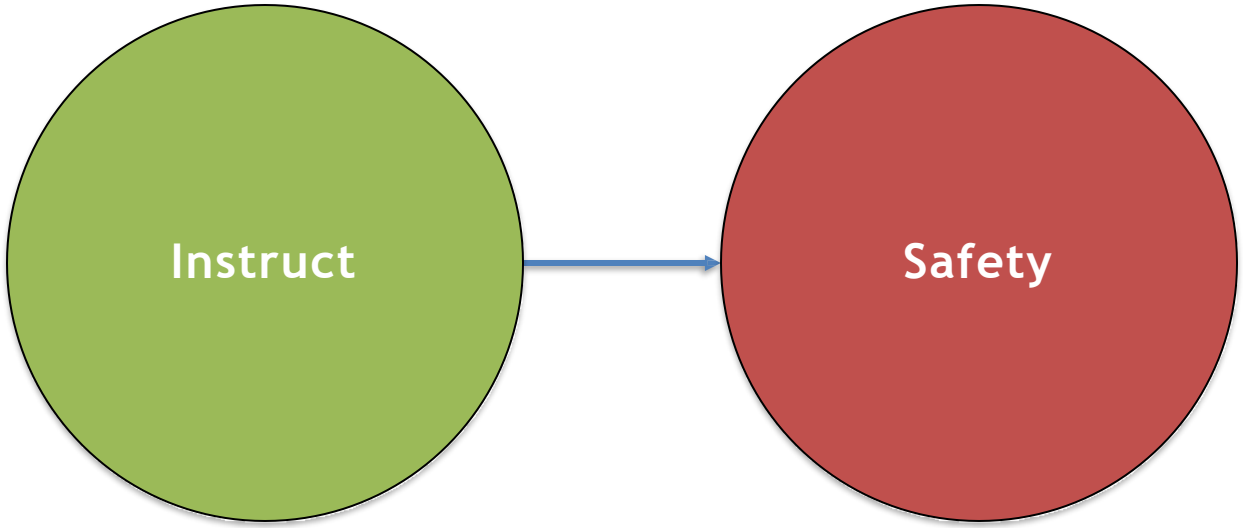
Process:
Involves further fine-tuning. We use RLHF to provide feedback on model outputs.

Outcome:
The model becomes safer reducing risk of biased content.

It's after this step that we get models like ChatGPT, Claude etc

Training Cycle - LLM

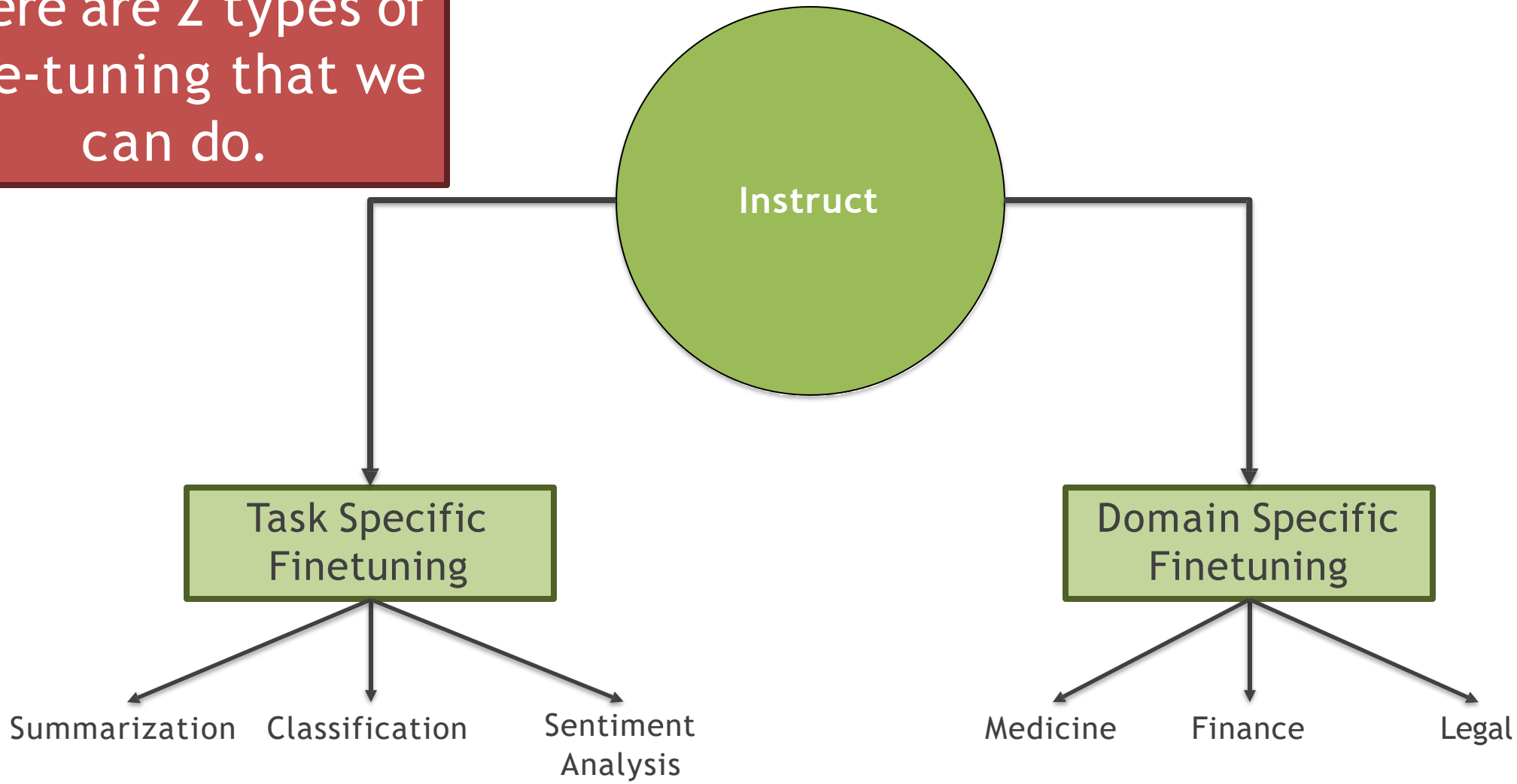
So, fine-tuning takes place in 2 stages.



•

Training Cycle - LLM

There are 2 types of fine-tuning that we can do.



Training Cycle - LLM

Another way of adapting LLMs for specific task, which is called "In-context learning.

In-context Learning	Fine-tuning
• A method of prompt engineering where the model is shown task demonstrations as part of the prompt. • No change in model	• A process of training the LLM on a labelled dataset specific to a particular task. • Change in model parameters.

Fine-tuning is a supervised process that leads to a new model, in contrast with in-context learning, which is considered "ephemeral."

Training Cycle - LLM

Before we go deeper into fine-tuning there is another way of adapting LLMs for specific task, which is called "in-context" learning.

In-context Learning	Fine-tuning
	• A process of training the LLM on a labelled dataset specific to a particular task. • Change in model parameters.

You may recall **in-context learning** from previous lecture with reference to prompting.

Let's focus on fine-tuning and how it makes our LLM better.

Fine-tuning is a supervised process that leads to a new model, in contrast with in-context learning, which is considered "ephemeral."

Outline

- Training Cycle – LLM
- Instruction-tuning
  - Full Parameter
  - PEFT
- LoRA
- QLoRA

Instruction-tuning (Full Parameter)

Fine-tuning very often means instruction fine-tuning.

An instruction dataset, comprising pairs of instructions, answers, and sometimes context, is required for such fine-tuning.

Instruction-tuning (Full Parameter)

Instruction	Context	Output
Suggest a good restaurant	Los Angeles, CA	In Los Angeles, CA, I suggest Rossoblu Italian Restaurant
Rewrite the sentence with more descriptive words	The game is fun	The game is exhilarating and enjoyable
Calculate the area of the triangle	Base: 5cm; Height: 6cm	The area of the triangle is $15\ cm^2$

This is an example of what an instruction dataset looks like.

Source: Alpaca-GPT4 dataset

Instruction-tuning (Full Parameter)

Task-specific fine-tuning:

This particular process involves training the model on a smaller, task-specific dataset.

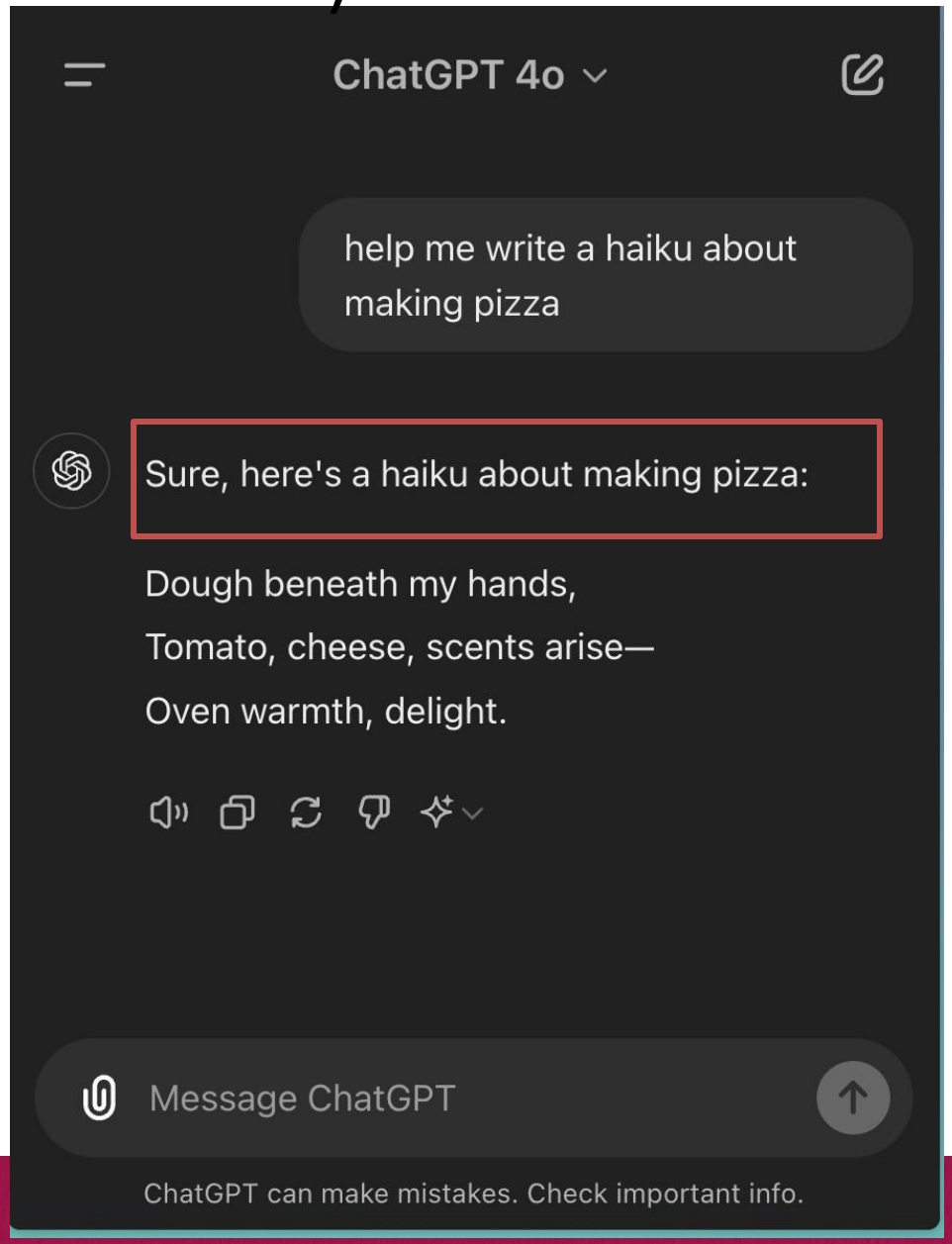
For e.g.: Summarize this, translate that, etc

This allows the model to learn the nuances, and specialized vocabulary relavant to the task.

Instruction-tuning (Full Parameter)

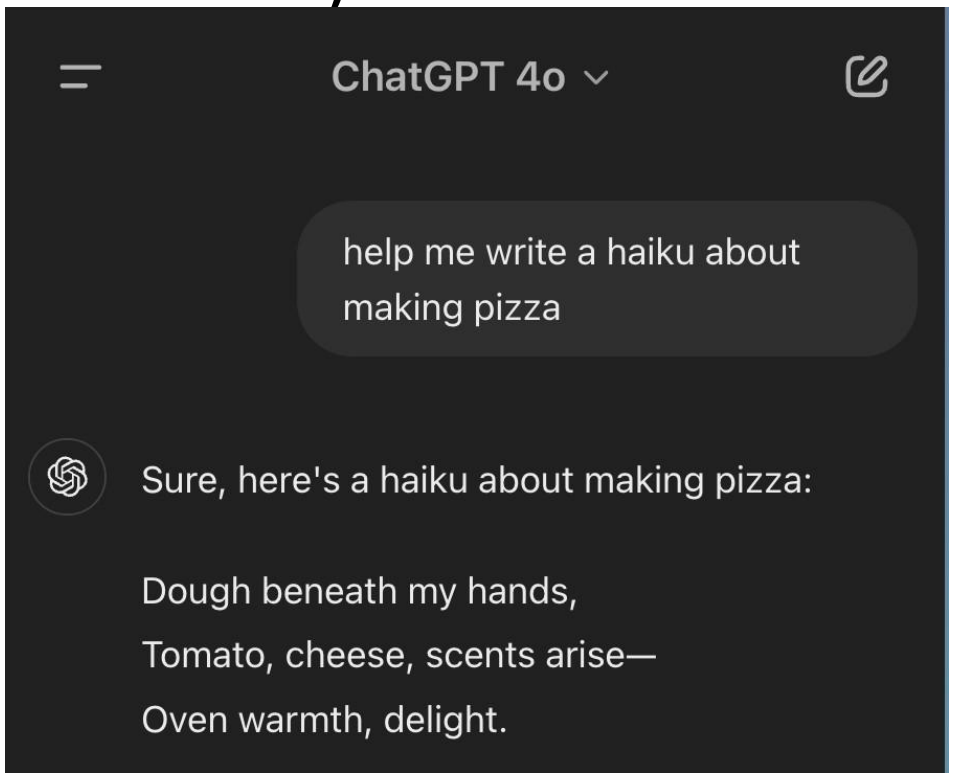
For e.g., if you train a model specifically for question answering:

Notice, how it answers requests, starting with ‘Sure…’.



Instruction-tuning (Full Parameter)

For e.g., if you train a model specifically for question answering:



Notice, how it answers requests, starting with ‘Sure…’.

This is opposed to how language models are trained (next word prediction), according to which the answer should just included the haiku directly.

Instruction-tuning (Full Parameter)

We have to be careful while doing task-specific finetuning to avoid catastrophic forgetting.

Catastrophic forgetting refers to the phenomenon where a model loses its ability to perform previously learned tasks when it is being fine-tuned on new tasks.

The key idea of catastrophic forgetting is that as the model learns new tasks, it may overwrite what it previously learned, leading to a loss in performance on earlier tasks.

Instruction-tuning (Full Parameter)

To mitigate the problem of catastrophic forgetting, we need to do multi-task finetuning.

This requires a lot of data, and training resources.

Replay or rehearsal Methods
Regularisation
PEFT

Instruction-tuning (Full Parameter)

- We need to update all the parameters while finetuning.
  - For a 7B model, we need to update 7 billion weights. For a 13 billion model, we need to update 13 billion weights.

- Storing and updating these weights require a lot of GPU memory.

Fun Fact: Did you know, training GPT-4 involved ~**25,000 A100 GPUs** over ~**90-100 days**, costing OpenAI nearly **$100 million!**

Instruction-tuning (Full Parameter)

Let’s take a fine-tuning example now.

Say we want to finetune a 10 billion parameter model. Let’s see how that looks in memory.

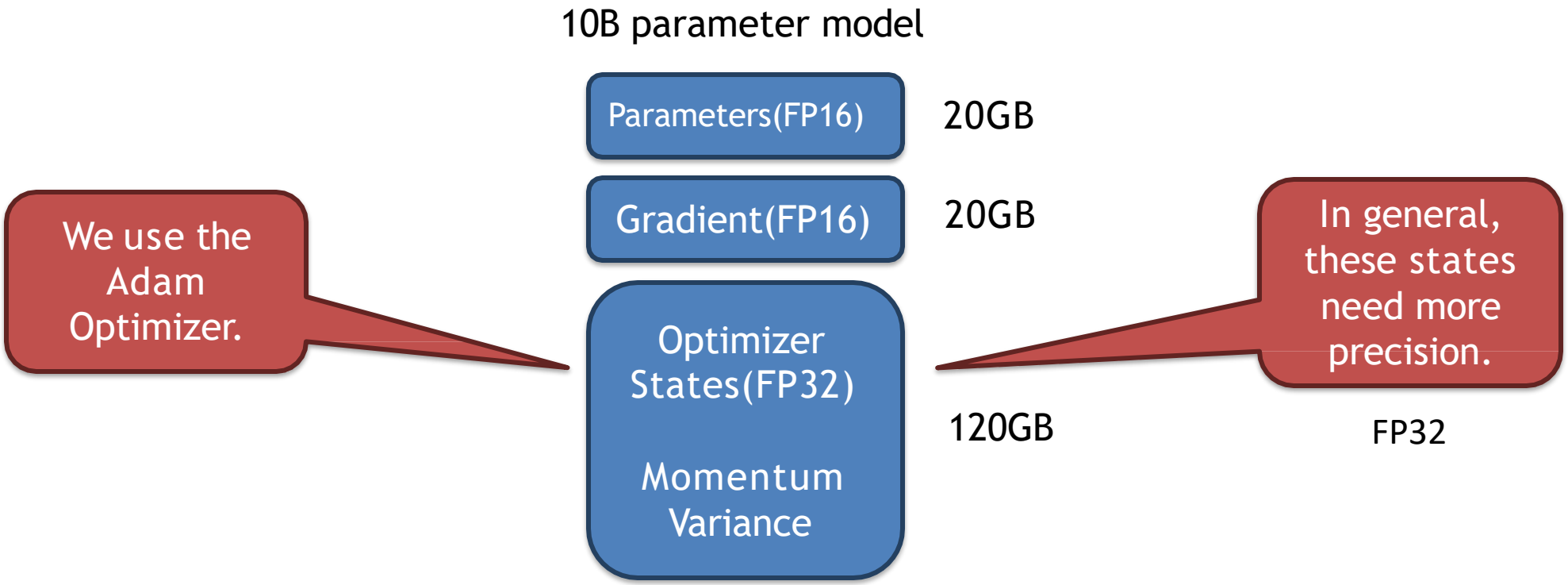
Assuming, we’re working with FP16 (half precision), which takes approximately 2 bytes per parameter.

4 bytes in FP32 (full precision).

When a GPU lists "**312 TFLOPS FP16**", it means it can perform 312 trillion **FP16 operations per second**.

Instruction-tuning (Full Parameter)

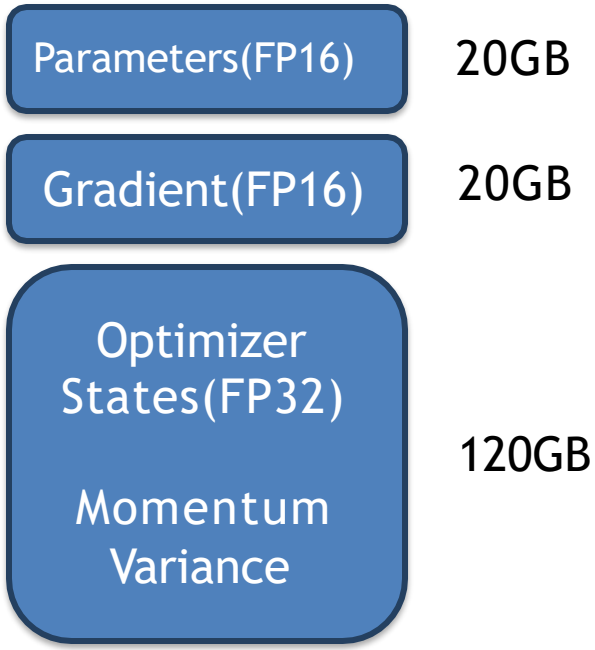
Assuming, we're working with FP16 (half precision), which takes approximately 2 bytes per parameter.



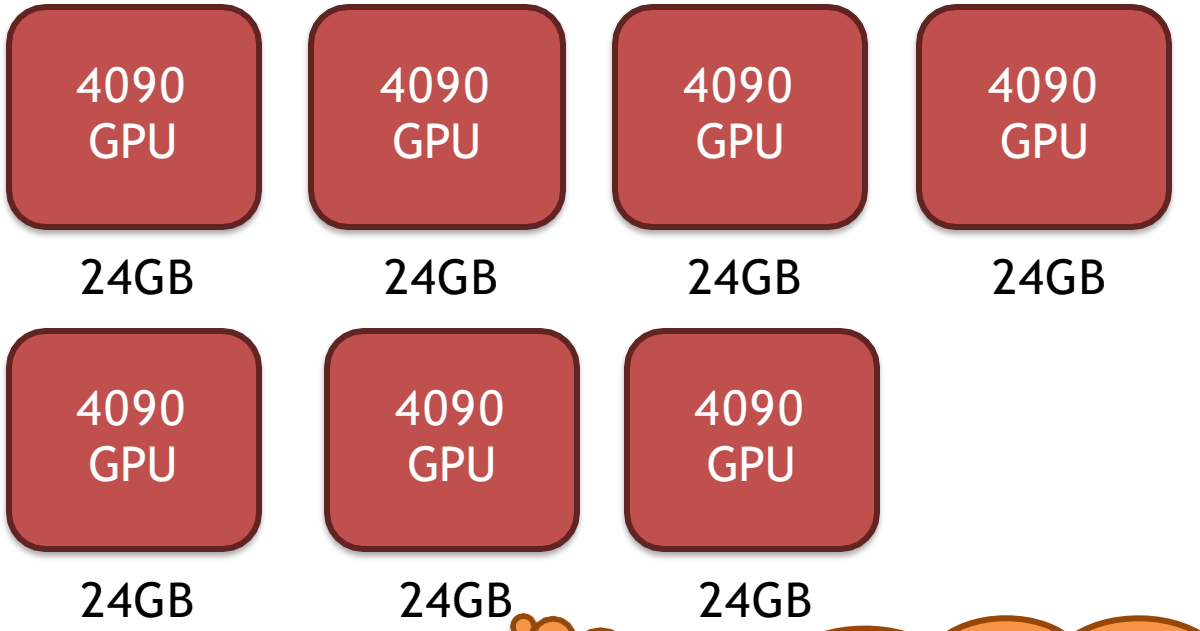
Instruction-tuning (Full Parameter)

Assuming, we're working with FP16 (half precision), which takes approximately 2 bytes per parameter.

10B parameter model



NVIDIA RTX 4090, a **consumer-grade GPU** with:
- **24 GB of GDDR6X VRAM**



This model needs at least 7 top-of-the-line consumer-grade GPU's to finetune.

Instruction-tuning (Full Parameter)

This makes full parameter finetuning inaccessible to normal folks like us.

So, what can we do?

Outline

- Training Cycle – LLM
- Instruction-tuning
  - Full Parameter
  - PEFT

Instruction-tuning (PEFT)

PEFT stands for Parameter Efficient Finetuning.

Unlike full parameter finetuning, PEFT preserves the vast majority of the model's original weights.

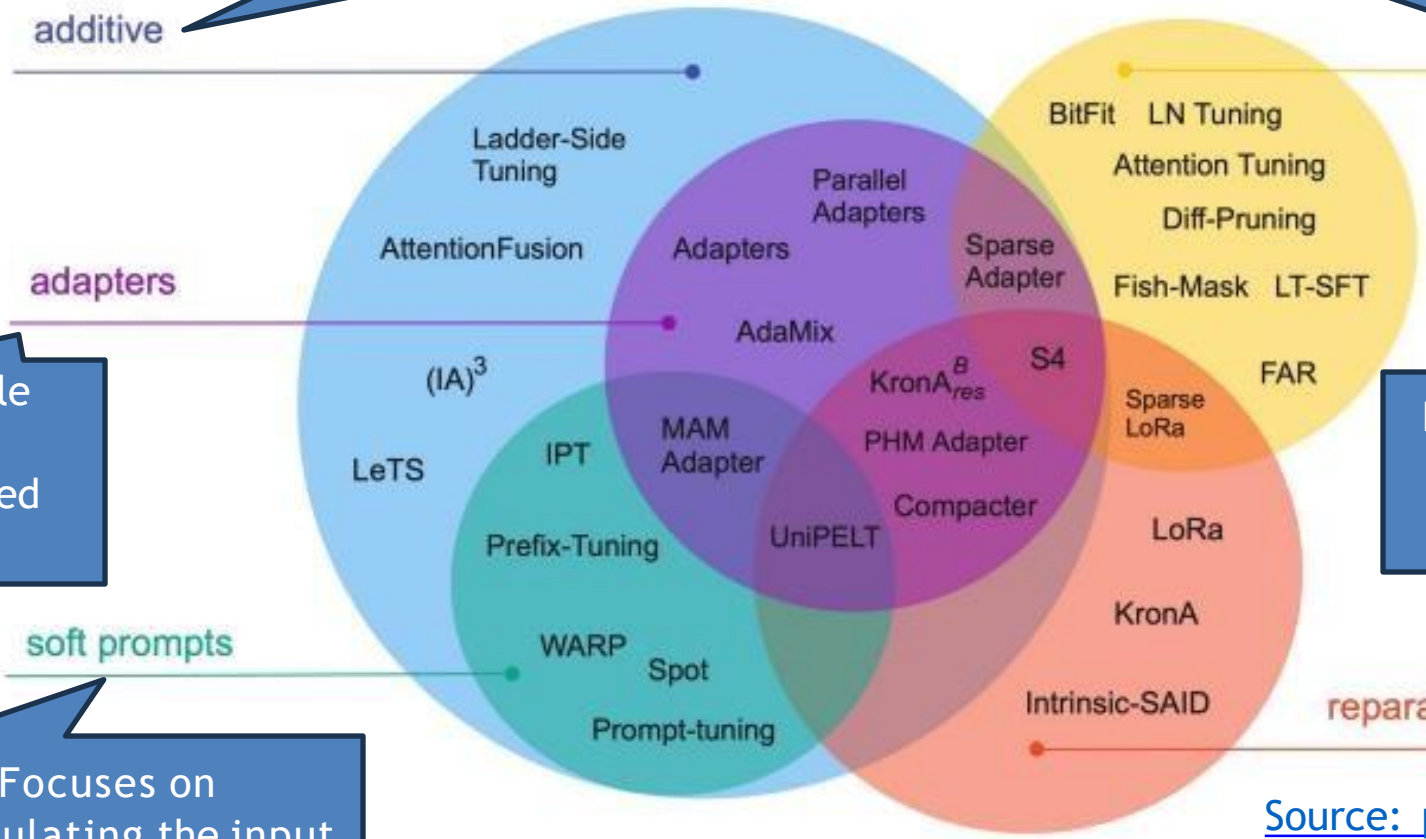
There are majorly three methods to do PEFT.

1. Additive
2. Selective
3. Reparameterization

Instruction-tuning (PEFT)

Add trainable layers or parameters to model

Subsets the parameters to finetune



Add new trainable layers to the architecture called ‘Adapters’

Reparametrize model weights using a new representation

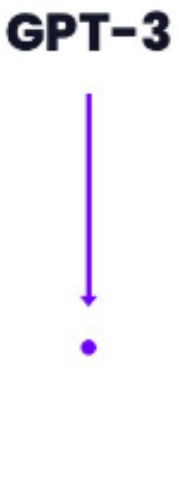
Focuses on manipulating the input (not the same as prompt engineering)

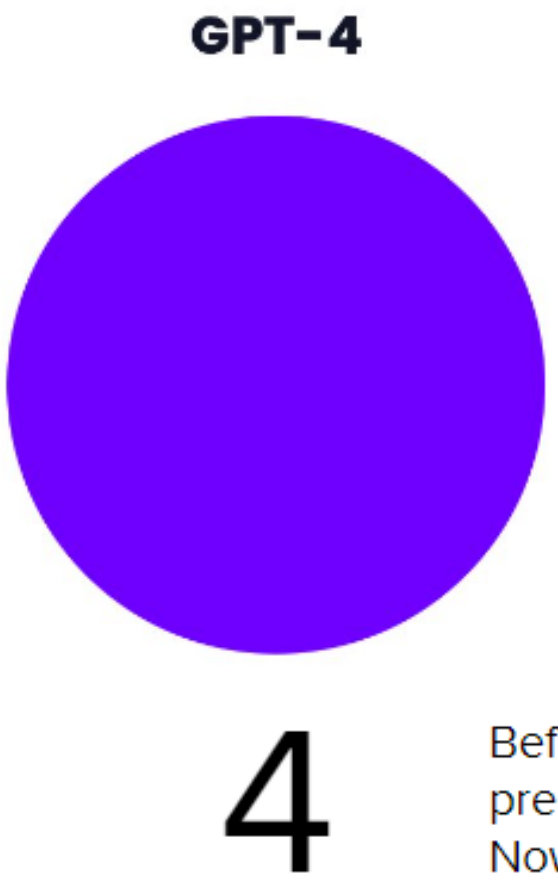
Source: paper “Scaling Down to Scale Up” (arxiv.org[3])[1]

Model sizes are still growing (?)

Publically available model sizes: 350M → 176B
Single-GPU RAM: 16Gb → 80Gb

Model size scales almost **two orders of magnitude** quicker than single-GPU memory





Before, we were worried we cannot pre-train models.
Now, can we even fine-tune them?

Large Language Models (LLMs) are quite large by name. These models usually have anywhere from **7 to 70 billion parameters.** To load a 70 billion parameter model in full precision would require **280 GB of GPU memory**! To train that model you would update billions of tokens over millions or billions of documents. The computation required is substantial for updating those parameters. The self-supervised training of these models is expensive, **costing companies up to $100 million**.

Parameter-Efficient Fine-Tuning (PEFT) can significantly help in adapting large language models (LLMs) for various tasks while overcoming limitations associated with traditional fine-tuning methods

Transformer - recap

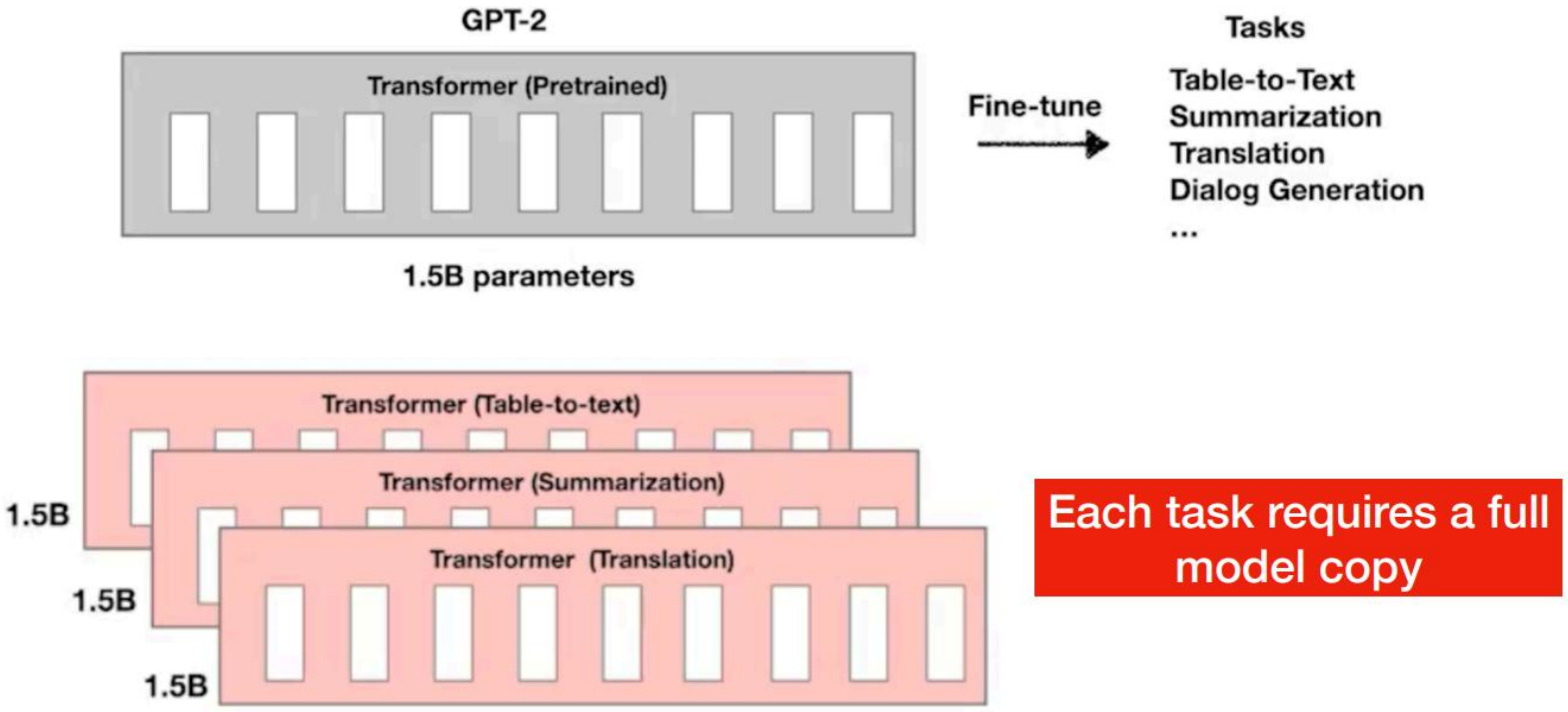
```
def self_attention(x):
    k = x @ W_k
    q = x @ W_q
    v = x @ W_v
    return softmax(q @ k.T) @ v

def transformer_block(x):
    """ Pseudo code by author based on [2] """
    residual = x
    x = self_attention(x)
    x = layer_norm(x + residual)
    residual = x
    x = FFN(x)
    x = layer_norm(x + residual)
    return x
```

Does this work? Yes!

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

Parameter efficient Fine-tuning (PEFT)



- Fine-tuning a model requires to store as many parameters as the original model
  - High storage complexity
- PEFT: An approach for finetuning large language models in a parameter-efficient manner

Source: peft (anoopsarkar.github.io)

- PEFT – classifications
  - Does the method introduce new parameters to the model?
  - Does it fine-tune a small subset of existing parameters?
  - Does the method aim to minimize memory footprint or storage efficiency?

- **Additive methods**
- **Selective methods**
- **Reparametrization-based methods**
- **Hybrid methods**

Lialin, V., Deshpande, V., & Rumshisky, A. (2023). Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*.

Outline

- Training Cycle – LLM
- Instruction-tuning
  - Full Parameter
  - PEFT
    - **Additive**
      - Adapters -Sparse Adapters-IA3
      - Soft prompt-Prompt Tuning-Prefix Tuning-P-Tuning-LlaMA Adapter
    - **Selective**
      - BitFit-Freeze and Reconfigure
    - **Reparameterization** (LORA,QLORA)
    - **Hybrid method** that is a combination of Reparameterization and Selective

Additive Method

Additive methods are probably the easiest to grasp.

The goal of additive methods is to **add an additional set of parameters or network layers** to augment the model.

When fine-tuning the data you **update the weights only of these newly added parameters**.

This makes training computationally easier and also adapts to smaller datasets

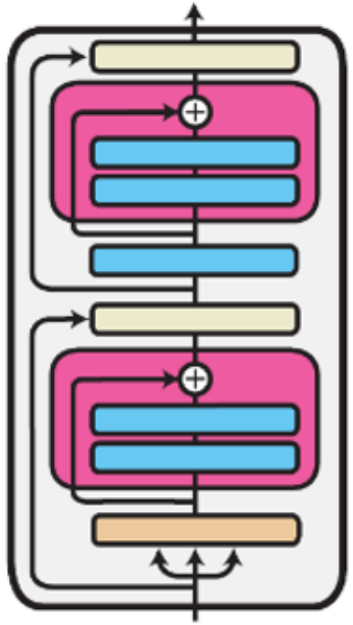
Additive Methods

- Key idea: **Augment the existing pre-trained model with additional parameters or layers and train only the added parameters**
- Introduce additional parameters. However, achieve significant training time and memory efficiency improvements
- Two approaches
  - Adapters
  - Soft prompts

Adapters

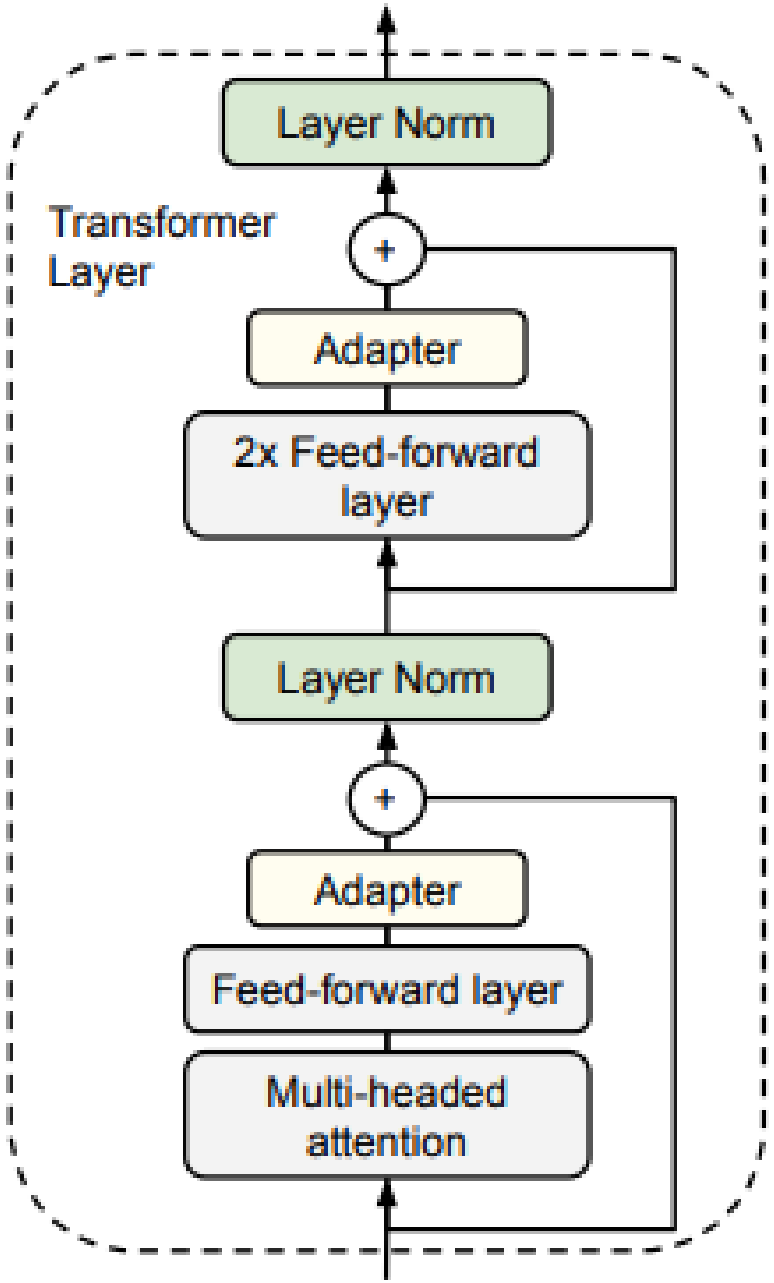
This technique was introduced in Houlsby et al [4]. The goal of adapters is to add small fully connected networks after Transformer sub-layers and learn those parameters

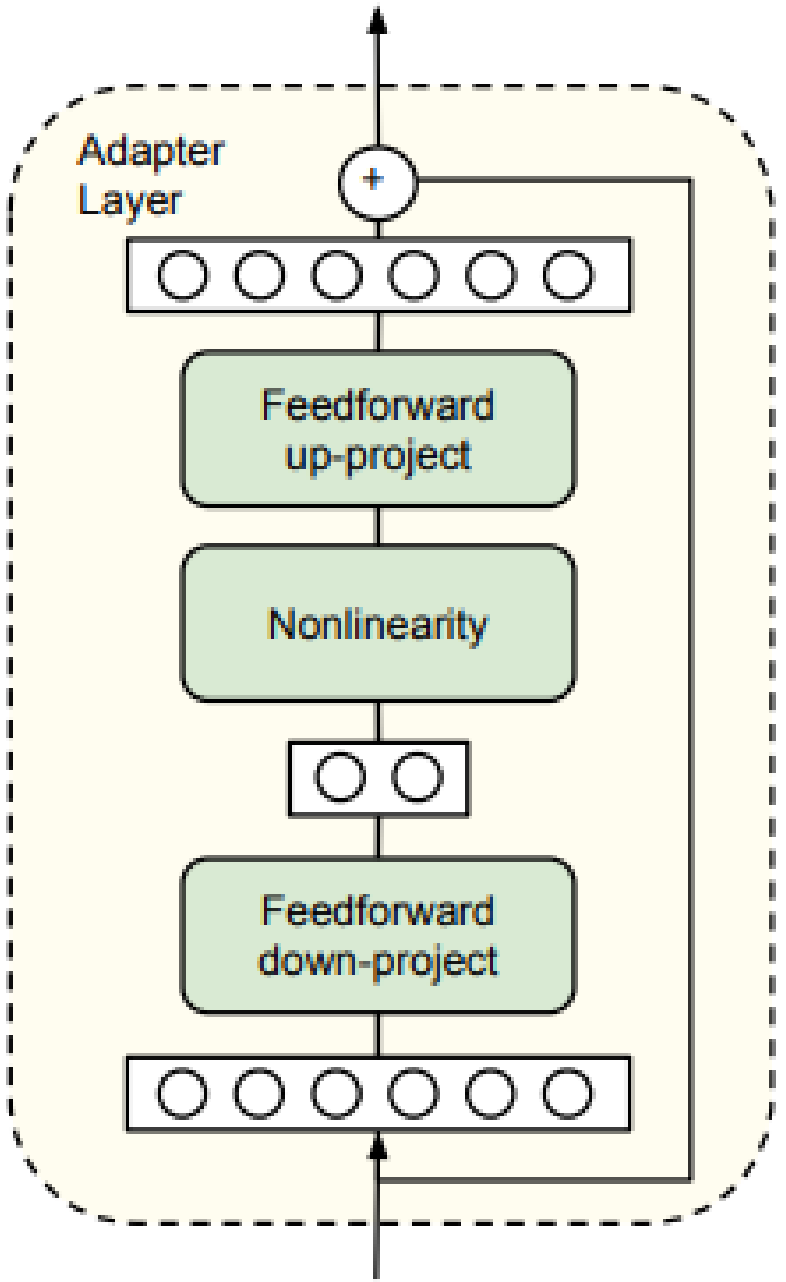
```
def transformer_block_adapter(x):
    """Pseudo code from [2] """
    residual = x
    x = self_attention(x)
    x = FFN(x)  # adapter
    x = layer_norm(x + residual)
    residual = x
    x = FFN(x)
    x = FFN(x)  # adapter
    x = layer_norm(x + residual)
    return x
```



https://arxiv.org/abs/1902.00751

Image source: adapterhub.ml





Source: Houlsby, Neil, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. "Parameter-efficient transfer learning for NLP." In *International Conference on Machine Learning*, pp. 2790-2799. PMLR, 2019.

Adapters

- Attach a small fully connected layer at every layer of the transformer
  - Inserts small modules (adapters) between transformer layers
- Adapter layer performs a down projection to project the input hidden layer information to a lower-dimensional space, followed by a non-linear activation function and an up projection
- A residual connection to generate the final form
- Only adapter layers are trainable, while the parameters of the original LLMs are frozen

$$h = h + f\left(hW_{down}\right)W_{up}$$

$$W_{up} \in \square^{r \times d} \qquad W_{down} \in \square^{d \times r}$$

Down projection: This is a linear transformation that reduces the dimensionality of the input hidden layer information (h) from the original LLM. Imagine squeezing a high-dimensional vector into a lower-dimensional one, focusing on the most relevant aspects for the specific task.

Let h be the input vector representing the hidden layer information from the original LLM (usually a high-dimensional vector).

We define a down-projection matrix (W_down) with dimensions appropriate for the desired reduction in dimensionality. This matrix essentially "compresses" the information.

The down projection is calculated by multiplying the input vector h with the down-projection matrix W_down:

z = W_down * h

Here, z represents the lower-dimensional representation of the input data.

Up projection: After processing in the lower-dimensional space, the data (z) is projected back to its original size (h'). This might seem redundant, but it allows the model to integrate the learned information from the lower-dimensional space back into the original context, enriching the representation.

Up Projection:

After processing in the lower-dimensional space, we want to project the data back to its original size.

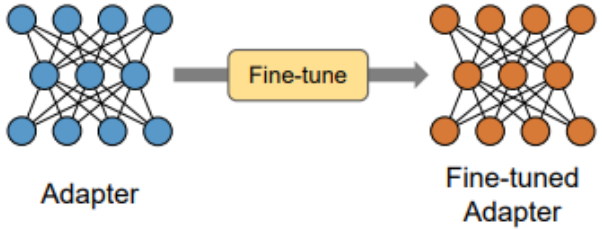
We define an up-projection matrix (W_up) with dimensions that allow us to expand the data back to the original size of the input vector h.

The up projection is calculated by multiplying the lower-dimensional vector z with the up-projection matrix W_up:

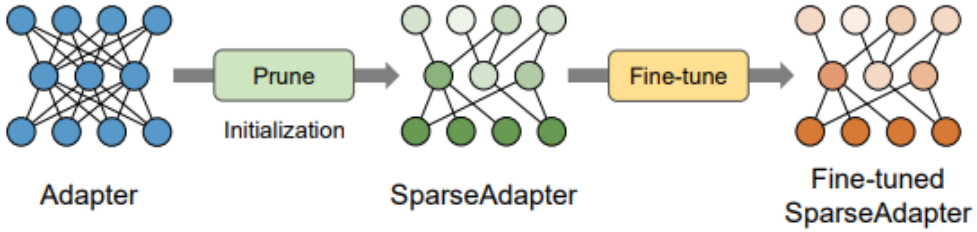
$$h' = W_{up} * z$$

Here, h' represents the data projected back to its original dimensionality.

- Sparse Adapter
  - Pruned adapters
  - Reduces the model size of neural networks by pruning redundant parameters and training the rest ones



(a) Standard Adapter Tuning.



(b) SparseAdapter Tuning.

Source:He, Shwai, Liang Ding, Daize Dong, Miao Zhang, and Dacheng Tao. "Sparseadapter: An easy approach for improving the parameter-efficiency of adapters." *arXiv preprint arXiv:2210.04284* (2022).

Implementing **sparse adaptation** involves incorporating sparsity techniques within the adapter layers of a PEFT (Parameter-Efficient Fine-Tuning) framework for large language models (LLMs).

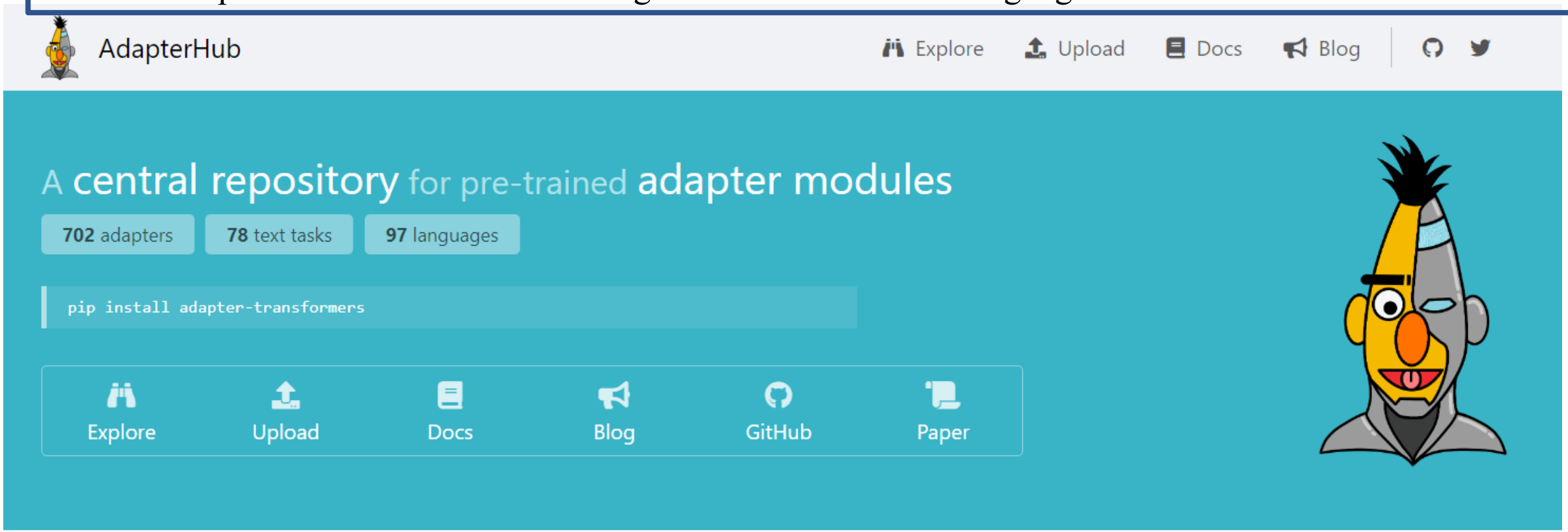
Choosing a Sparsity Technique:

There are several ways to introduce sparsity in adapter layers. Here are two common approaches:

- **Pruning:**
  - Start with a fully connected adapter layer (all weights have non-zero values).
  - During or after training, iteratively remove connections (set their weights to zero) considered unimportant based on specific criteria.
  - Common pruning algorithms include:
    - **Magnitude-based pruning:** Removes weights with values below a certain threshold.
    - **Gradient-based pruning:** Removes weights that contribute minimally to the gradient during training.
- **Magnitude Thresholding:**
  - Initialize all weights in the adapter layer with random values.
  - After training, set weights with absolute values below a certain threshold to zero, effectively making them inactive.

- AdapterHub
  - An easy-to-use and extensible adapter training and sharing framework for transformer-based model

AdapterHub is a framework designed to simplify the process of integrating, training, and using adapter modules for parameter-efficient fine-tuning of transformer-based language models.



Additive PEFT (IA3)- **Infused Adapter by Inhibiting and Amplifying Inner Activations**

Let's consider the scaled dot-product attention found in a normal transformer:

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

we are working with an additive method, we are seeking to add parameters to this network. We want the dimensionality to be quite small. $(IA)^3$ proposes the following **new vectors to be added to the attention mechanism:**

- We just added column vectors l_k and l_v and take the Hadamard product between the column vector and the matrix (multiply the column vector against all columns of the matrix).

$$\text{softmax}\left(\frac{Q(l_k \odot K^T)}{\sqrt{d_k}}\right)(l_v \odot V)$$

We also introduce one other learnable column vector l_{ff} that is added to the feed forward layers as follow:

$$(l_{ff} \odot \gamma(W_1 x))W_2$$

In this example, gamma is the activation function applied to the product between the weights and input

The Hadamard product ⊙,. Each element (c_ij) in the resulting matrix C is calculated by multiplying the corresponding elements (a_ij and b_ij) from matrices A and B at the same position (i, j).

Additive PEFT: (IA)3

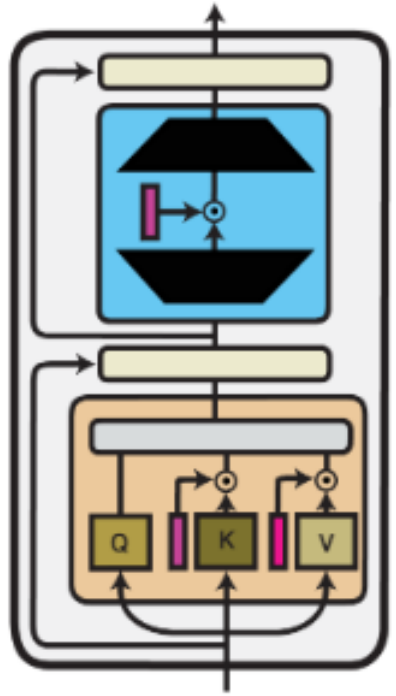
One-sentence idea:

Rescale key, value, and hidden FFN activations

Pseudocode:

```
def transformer_block_with_ia3(x):
    residual = x
    x = ia3_self_attention(x)
    x = LN(x + residual)
    residual = x
    x = x @ W_1           # FFN in
    x = l_ff * gelu(x)    # (IA)3 scaling
    x = x @ W_2           # FFN out
    x = LN(x + residual)
    return x

def ia3_self_attention(x):
    k, q, v = x @ W_k, x @ W_q, x @ W_v
    k = l_k * k
    v = l_v * v
    return softmax(q @ k.T) @ V
```



https://arxiv.org/abs/2205.05638

Image source: adapterhub.ml

Soft Prompting

With soft-prompts our goal is to add information to the base model that is more specific to our current task. With prompt tuning we accomplish this by creating a set of parameters for the **prompt tokens and injecting this at the beginning of the network**.

Soft-prompting is a technique that tries to avoid this dataset creation. In hard prompting, we are creating data in a discrete representation (picking words.) In soft-prompting, we seek a **continuous representation of the text we will input to the model**

Depending on the technique, there are different methods for how the information is added to the network. The core idea is that the base model does not optimize the text itself but rather the **continuous representation (i.e. some type of learnable tensor) of the prompt text.** This can be some form of embedding or some transformation applied to that embedding.

Prompt Tuning:

•**Trainable Prompt Parameters:** This approach focuses on creating a set of trainable parameters specifically for the prompt tokens. These parameters are essentially learned representations of the task that guide the LLM.

```
def prompt_tuning(seq_tokens, prompt_tokens):
    """ Pseudo code from [2]. """
    x = seq_embedding(seq_tokens)
    soft_prompt = prompt_embedding(prompt_tokens)
    model_input = concat([soft_prompt, x], dim=seq)
    return model(model_input)
```

Example sentiment Analysis

Imagine you want to fine-tune a pre-trained LLM for sentiment analysis. Here's how prompt tuning might work:

Define Prompt Template: You create a prompt template with special tokens representing the task and sentiment categories (positive, negative, neutral).

An example template could be:**[SENTIMENT] sentiment: __label__ , review: "This movie was fantastic!“**
[SENTIMENT] is a special token representing the task (sentiment analysis).__label__ is a placeholder that will be filled with the predicted sentiment during inference.“

This movie was fantastic!" is the actual review text you want the model to analyze.

Trainable Prompt Parameters: The model learns a set of parameters (embeddings) for each special token in the prompt template. These parameters capture the task-specific information.

Feeding the Input: During training and inference, you combine the prompt template with the actual review text, replacing __label__ with a special "unknown" token.

Sentiment Analysis (Soft Prompt)

Sentence:
"This movie was fantastic!“

Template Logic (masked-style):

[SENTIMENT] sentiment: __label__, review: "This movie was fantastic!"
Soft Prompt Encoding:

[p_1, p_2, p_3, p_4, p_5, embed("review"), embed(":"), embed(""), embed("This"), embed("movie"), embed("was"), embed("fantastic"), embed("!"), embed(""), embed("__label__")]

__label__
placeholders in the template — locations where the model is expected to **predict a missing token**.

Question Answering(Soft Prompt)

Sentence:
"The Eiffel Tower is located in Paris.“

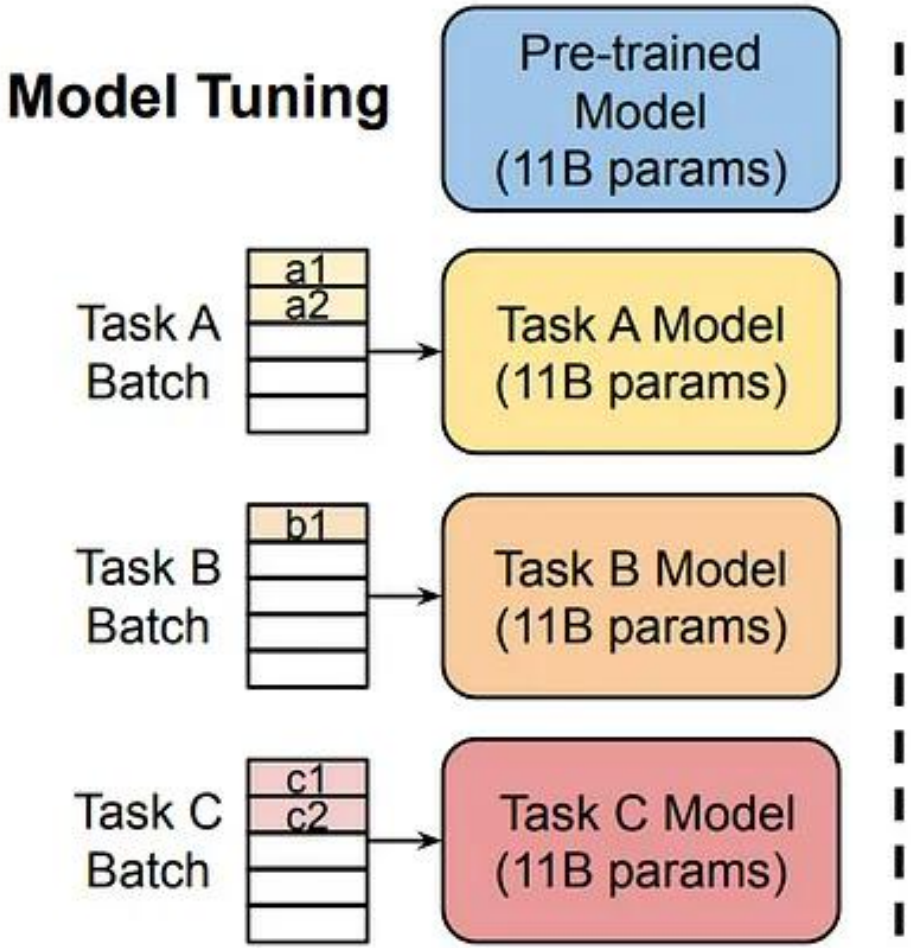
Template Logic:

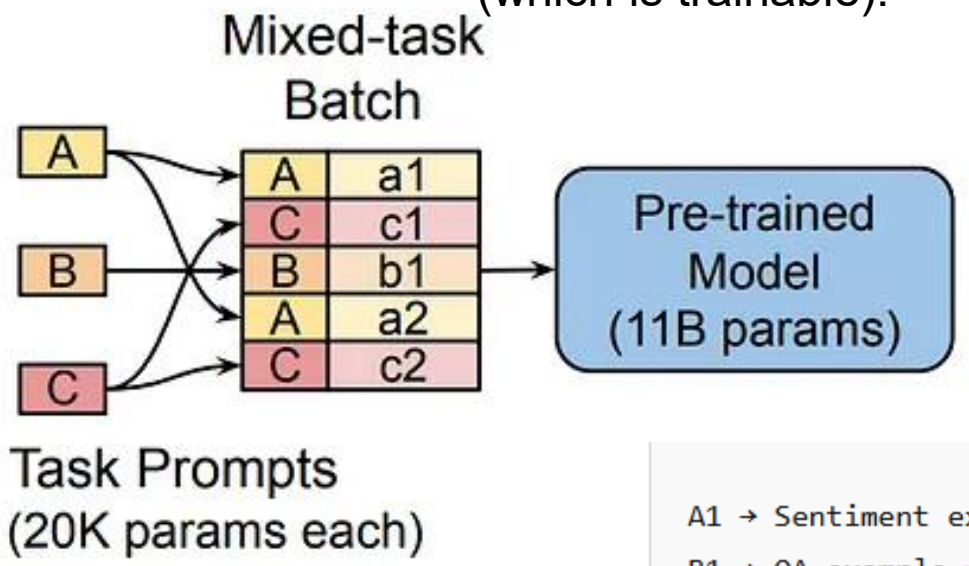
[QA] question: __label__, passage: "The Eiffel Tower is located in (__label__")].“

Soft Prompt Encoding:

[q_1, q_2, q_3, q_4, q_5, embed("passage"), embed(":"), embed(""), embed("The"), embed("Eiffel"), embed("Tower"), embed("is"), embed("located"), embed("in"), embed("Paris"), embed("."), embed(""), embed("__label__")]

Model Sharing for multiple tasks





- The **prompt tells the model what task to perform**
- All tasks are learned **together** using a single model.
- The only thing different per task is the **soft prompt** (which is trainable).

```
A1 → Sentiment example with soft prompt A
B1 → QA example with soft prompt B
C1 → Translation with soft prompt C
```

```
Task A → [prompt_A] + "This movie was great!" → predict: positive
Task B → [prompt_B] + "What is the capital of France?" → predict: Paris
Task C → [prompt_C] + "Translate: Hello" → predict: Bonjour
```

Soft Prompt

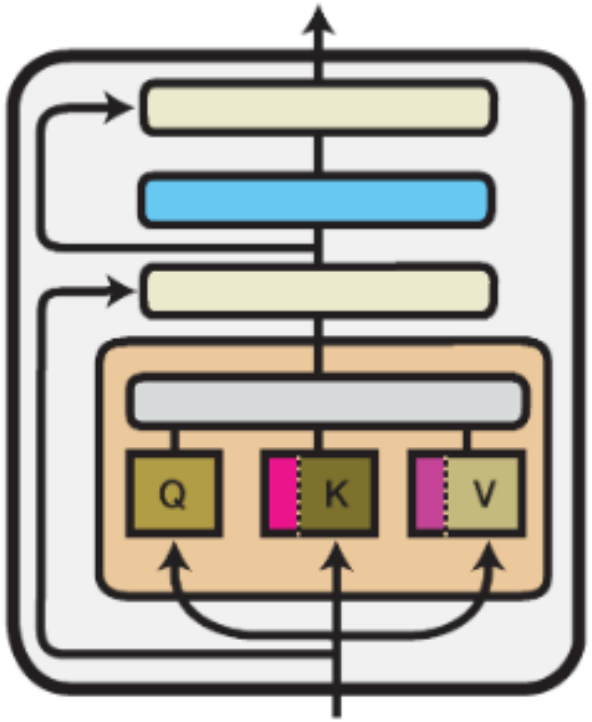
Additive PEFT: Prompt tuning

One-sentence idea:

Fine-tune part of your attention input – soft prompt

Pseudocode:

```
def prompt_tuning_attention(input_ids):
    q = x @ W_q
    k = cat([s_k, x]) @ W_k   # prepend a
    v = cat([s_v, x]) @ W_v   # soft prompt
    return softmax(q @ k.T) @ V
```



P-Tuning

Soft Prompting

P-Tuning is prompt-tuning but encoding the prompt using an LSTM.

Here prompt is a function that takes a context x and a target y and organizes itself into a template T. The authors provide the example sequence

“The capital of Britain is [MASK]”.

Here the prompt is “The capital of … is …”,
the context is “Britain” and the target is [MASK]

. We can use this formulation to create two sequences of tokens, everything before the context and everything after the context before the target.

- The sentence: "The capital of Britain is [MASK]"
- You want to replace "The capital of ... is" with a **soft prompt** of 5 tokens.
- Input "Britain" is a **real token**, embedded normally.

[p_1, p_2, p_3, p_4, p_5, embed("Britain"), embed("is"), embed("[MASK]")]

```
def p_tuning(seq_tokens, prompt_tokens):
    """Pseudo code for p-tuning created by Author."""
    h = prompt_embedding(prompt_tokens)
    h = LSMT(h, bidirectional=True)
    h = FFN(h)

    x = seq_embedding(seq_tokens)
    model_input = concat([h, x], dim=seq)

    return model(model_input)
```

LLaMA adapter

Soft Prompting

LLaMA adapter introduce Adaptation Prompts, which are soft-prompts appended with the input to the transformer layer. These adaption prompts are inserted in the L topmost of the N transformer layers.

With additive methods LLaMA adapter introduce a new set of parameters that have some random initialization over the weights. Because of this random noise added to the LM, we can potentially experience unstable fine-tuning which can cause a problem with large loss values at the early stages.

To solve this problem, the authors introduce a gating factor, initialized to 0, that is multiplied by the self attention mechanism. The product of the gating factor and self-attention is referred to as **zero-init attention**. The gating value is adaptively tuned over the training steps to create a smoother update of the network parameters.

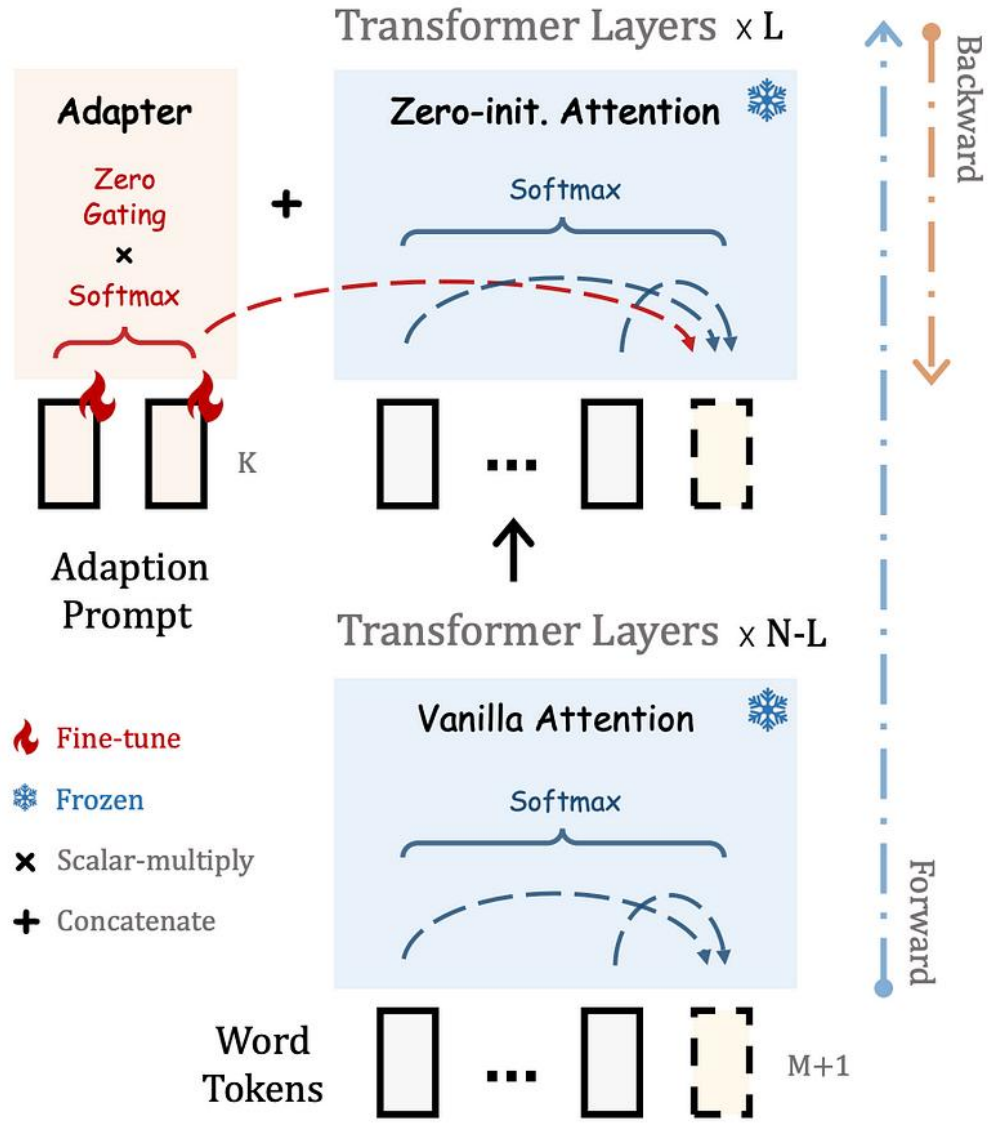
LLaMA adapter

```
def transformer_block_llama_adapter(x, soft_prompt, gating_factor):
    """LLaMA-Adapter pseudo code created by Author"""
    residual = x

    adaption_prompt = concat([soft_prompt, x], dim=seq)
    adaption_prompt = self_attention(adaption_prompt) * gating_factor  # zero-i

    x = self_attention(x)
    x = adaption_prompt * x
    x = layer_norm(x + residual)
    residual = x
    x = FFN(x)
    x = layer_norm(x + residual)

    return x
```



Selective Methods

- Selectively fine-tuning some parts of the network
  - Fine-tuning **only a few top layers** of a network
  - Based on the **type** of the layer
  - Internal structure of the network (tuning **only model biases**)
  - Tuning only **particular rows of the parameter matrix**
  - **Sparsity-**based approaches

Selective PEFT: BitFit

One-sentence idea:

Fine-tune only model biases

Pseudocode:

```
params = (p for n, p
          in model.named_parameters()
          if "bias" in n)
optimizer = Optimizer(params)
```

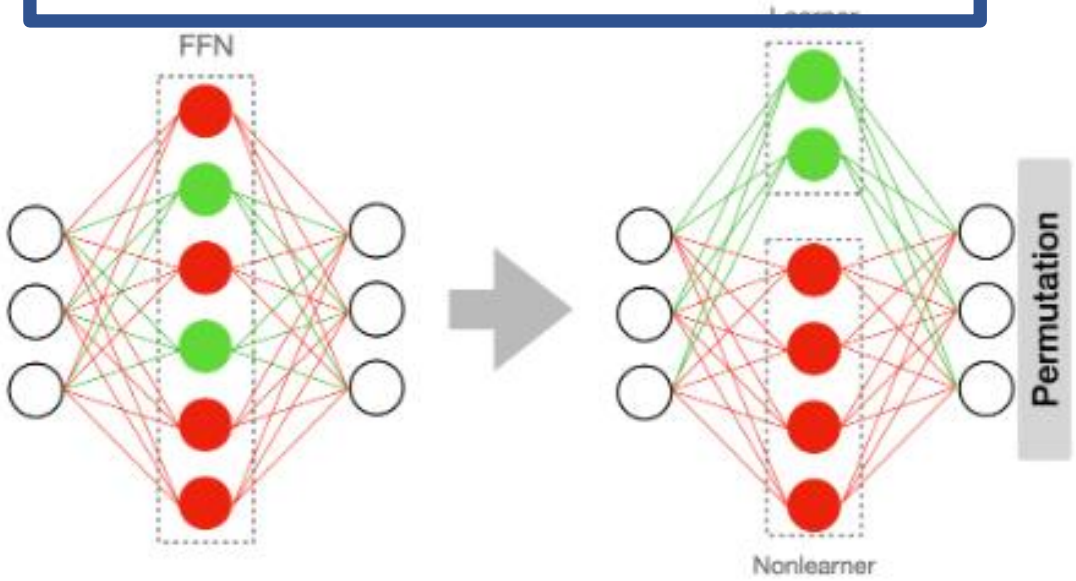
Selective PEFT: Freeze and Reconfigure

One-sentence idea:

Selects columns of parameter matrices to train and reconfigures linear layers into trainable and frozen

```
def far_layer(x):
    h1 = x @ W_t
    h2 = x @ W_f
    return concat([h1, h2], dim=-1)
```

Split your linear layer into 2 matrices based on information measure or gradient measure- Frozen matrix and Trainable matrix



Freezing: This involves keeping certain layers of the pre-trained LLM's architecture unchanged during fine-tuning. These frozen layers typically represent **the LLM's core capabilities for understanding language**.
Reconfiguring: Specific parts of the LLM, particularly the later layers, are reconfigured to **adapt to the target task**

Reparametrization-based methods- LoRA

Courtesy:, deeplearning.ai ,

Reparametrization-based methods

Reparameterization refers to transforming or reformulating the parameters of a model in a way that simplifies certain aspects of the training or optimization process.
This can involve:

- **Simplifying the Optimization Landscape**: Making the parameter space easier to navigate during training.
- **Improving Efficiency**: Reducing the number of parameters or computations required.
- **Introducing Constraints**: Incorporating prior knowledge or constraints more naturally into the model.

- Leverage low-rank representations to minimize the number of trainable parameters
- Basic intuition: neural networks have low dimensional representations
- Common approaches
  - Intrinsic SAID
  - LoRA
  - QLoRA

LoRA: Low-Rank Adaptation of Large Language Models

Edward Hu* **Yelong Shen*** **Phillip Wallis** **Zeyuan Allen-Zhu**
Yuanzhi Li **Shean Wang** **Lu Wang** **Weizhu Chen**
Microsoft Corporation
{edwardhu, yeshe, phwallis, zeyuana,
yuanzhil, swang, luw, wzchen}@microsoft.com
yuanzhil@andrew.cmu.edu
(Version 2)

Abstract

An important paradigm of natural language processing consists of large-scale pre-training on general domain data and adaptation to particular tasks or domains. As we pre-train larger models, full fine-tuning, which retrains all model parameters, becomes less feasible. Using GPT-3 175B as an example – deploying independent instances of fine-tuned models, each with 175B parameters, is prohibitively expensive. We propose **Lo**w-**R**ank **A**daptation, or LoRA, which freezes the pre-trained model weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture, 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. LoRA performs on-par or better than fine-tuning in model quality on RoBERTa, DeBERTa, GPT-2, and GPT-3, despite having fewer trainable parameters, a higher training throughput, and, unlike adapters, *no additional inference latency*. We also provide an empirical investigation into rank-deficiency in language model adaptation, which sheds light on the efficacy of LoRA. We release a package that facilitates the integration of LoRA with PyTorch models and provide our implementations and model checkpoints for RoBERTa, DeBERTa, and GPT-2 at https://github.com/microsoft/LoRA.

Low-Rank Adaptation, or LoRA

Low-Rank Adaptation, or LoRA, **freezes the pretrained model weights** and **injects trainable rank decomposition matrices** into each layer of the Transformer architecture, 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.
- LoRA performs on-par or better than finetuning in model quality on RoBERTa, DeBERTa, GPT-2, and GPT-3, despite having fewer trainable parameters,
- a higher training throughput, and, unlike adapters, no additional inference latency*

*Inference latency refers to the time it takes for a model to process an input and produce an output. It's a crucial factor in real-time applications where quick responses are necessary

Rank of a Matrix

The rank of the matrix gives the number of linearly independent column vectors of the matrix and this number also means the dimension of the linear space these vectors span. For example:

For example:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix has 3 column vectors which span the 3 dimensional space, they are it's trivial base vectors, so they are linearly independent...

Row-Echelon Form : *A non-zero matrix is said to be in a row-echelon form if all zero rows occur as bottom rows of the matrix and if the first non-zero element in any lower row occurs to the right of the first non-zero entry in the higher row.*

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix}$$

- Rank of a matrix is equal to the number of non-zero rows if it is in Echelon Form.

Let V be a vector space over a field (like $\mathbb{R}$ or $\mathbb{C}$).

A set of vectors $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\} \subset V$ is called a **basis** of V if:

1. **Span**: Every vector in V can be written as a linear combination of the basis vectors.

$$\text{Span}(\vec{v}_1, \ldots, \vec{v}_n) = V$$

2. **Linear Independence**: No vector in the set can be written as a linear combination of the others.

$$c_1\vec{v}_1 + \cdots + c_n\vec{v}_n = 0 \Rightarrow c_1 = \cdots = c_n = 0$$

There are many equivalent definitions of the rank of a matrix A.
The following two conditions are equivalent to each other

1. The largest linearly independent subset of columns of A has size k. That is, all n columns of A arise as linear combinations of only k of them.
2. The largest linearly independent subset of rows of A has size k. That is, all m rows of A arise as linear combinations of only k of them.

if $A = YZ^{T}$, then all of A's columns are linear combinations of the k columns of Y, and all of A's rows are linear combinations of the k rows of Z^{T}

The "rank" of a matrix is the dimension of that space spanned by the vectors it contains.

If we put the two vectors [7, 3, 5] and [-2, -1, 5] into a matrix:

$$\begin{bmatrix} 7 & -2 \\ 3 & -1 \\ 5 & 5 \end{bmatrix}$$

the rank of the matrix is the dimension of the space that you get by taking all combinations of the vectors. We've already done that, and saw that the space spanned by [7, 3, 5] and [-2, -1, 5] was a plane. In this case, the rank is 2 (because a plane is 2 dimensional).

Understanding Rank

In linear algebra, the rank of a matrix is a measure of the number of linearly independent rows or columns in the matrix. For a matrix $M \in \mathbb{R}^{m \times n}$:

- **Full Rank:** The matrix has full rank if its rank is equal to the smaller of the number of rows or columns, i.e., $\text{rank}(M) = \min(m, n)$.
- **Low Rank:** A matrix is considered low-rank if its rank is significantly less than the smaller dimension, i.e., $\text{rank}(M) \ll \min(m, n)$.

In practical terms, a low-rank matrix can be approximated by the product of two smaller matrices. For example, a matrix M of size $m \times n$ with rank r can be approximated as:

$$M \approx AB$$

where $A \in \mathbb{R}^{m \times r}$ and $B \in \mathbb{R}^{r \times n}$.

Full-Rank Matrix Example

Consider the following 3x3 matrix W:

$$W = \begin{pmatrix} 4 & 2 & 3 \\ 3 & 1 & 4 \\ 2 & 1 & 3 \end{pmatrix}$$

This matrix is full-rank if its rank is 3, which is the maximum for a 3x3 matrix.

Low-Rank Matrix Example

A low-rank matrix is one whose rank is less than the maximum possible. For example, consider the matrix M:

$$M = \begin{pmatrix} 2 & 4 & 4 \\ 1 & 2 & 2 \\ 1 & 2 & 2 \end{pmatrix}$$

This matrix is rank-1 (it has linearly dependent rows/columns), which is less than its dimension of 3.

Low-Rank Decomposition

Given a matrix W, we want to decompose it into two matrices A and B such that $W \approx AB$ where A and B have a smaller rank.

For our full-rank matrix W:

$$W = \begin{pmatrix} 4 & 2 & 3 \\ 3 & 1 & 4 \\ 2 & 1 & 3 \end{pmatrix}$$

Let's decompose W into A and B with a rank of 2.

Thus, the low-rank approximation of W is:

$$W \approx AB = \begin{pmatrix} -4.6 & -0.65 \\ -4.3 & 0.88 \\ -3.5 & -0.14 \end{pmatrix} \begin{pmatrix} -0.57 & -0.30 & -0.77 \\ -0.73 & 0.65 & 0.22 \end{pmatrix}$$

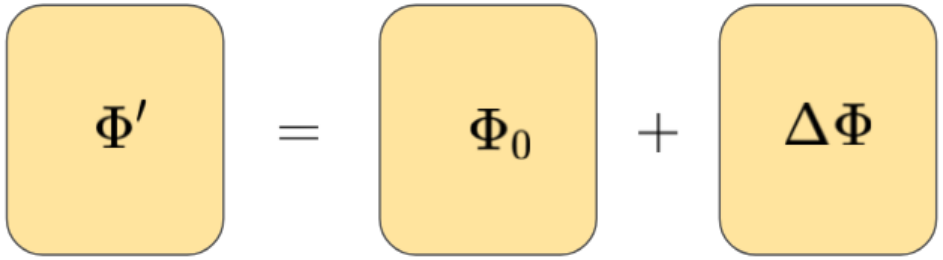
Verification

Multiplying A and B should yield an approximation of W:

$$AB = \begin{pmatrix} -4.6 & -0.65 \\ -4.3 & 0.88 \\ -3.5 & -0.14 \end{pmatrix} \begin{pmatrix} -0.57 & -0.30 & -0.77 \\ -0.73 & 0.65 & 0.22 \end{pmatrix} \approx \begin{pmatrix} 4.00 & 2.01 & 3.01 \\ 3.01 & 0.98 & 4.02 \\ 2.01 & 1.00 & 3.01 \end{pmatrix}$$

The resulting matrix is a close approximation of the original W, demonstrating the effectiveness of the low-rank decomposition.

- LoRA: Low-Rank Adaptation of Large Language Models
  - Try to achieve a **small number of task-specific parameters**
  - While the weights of a pre-trained model have full rank on the pre-trained tasks, the LoRA authors point out that pre-trained large language models have a low "intrinsic dimension"* when they are adapted to a new task



Decompose the weight changes, ΔW, into a lower-rank representation

*The intrinsic dimension refers to the minimum number of parameters required to effectively represent the information in a model.

Original Dense Layer

Consider a dense layer in a neural network with weights $W \in \mathbb{R}^{d \times d}$, where d is the dimension of the input and output. The operation performed by this layer is typically:

$$Y = WX + b$$

where:

- $X \in \mathbb{R}^{d \times n}$ is the input to the layer.
- $Y \in \mathbb{R}^{d \times n}$ is the output of the layer.
- $b \in \mathbb{R}^{d}$ is the bias term (which we can ignore for simplicity in our explanation).

Low-Rank Adaptation (LoRA)

Instead of updating W directly during fine-tuning, LoRA proposes to represent the change in W as a low-rank update. Specifically:

$$W' = W + \Delta W$$

where $\Delta W \approx AB$ with:

- $A \in \mathbb{R}^{d \times r}$
- $B \in \mathbb{R}^{r \times d}$
- r is the rank, which is much smaller than d.

So the new weight matrix after adaptation becomes:

$$W' = W + AB$$

Training Process

During training, instead of optimizing W' directly, we optimize the low-rank matrices A and B. The adaptation process can be described as follows:

1. **Forward Pass**: Compute the output using the modified weights.
   $$Y = (W + AB)X$$
2. **Backward Pass**: Compute gradients with respect to A and B and update these matrices accordingly.

Full Fine-Tuning

1. **Initialization:**
    - The model starts with pre-trained weights Φ_0.
2. **Updating Weights:**
    - During fine-tuning, the weights are updated to $\Phi_0 + \Delta\Phi$.
    - This update process involves adjusting the weights to better fit the new task's data.

Full Fine-Tuning

3. **Objective Function:**

- The goal is to maximize the conditional language modeling objective, which can be written as:

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

- Here, Z is the training dataset consisting of pairs (x, y), and the inner summation runs over the tokens in the output sequence y.
- $P_{\Phi}(y_t|x, y_{<t})$ is the probability of the token y_t given the input x and the previous tokens $y_{<t}$.

Z is context-target pairs

4. **Drawbacks:**

- Full fine-tuning requires learning a separate set of parameters $\Delta\Phi$ for each task.
- The size of $\Delta\Phi$ is the same as the original weights Φ_0, which can be very large (e.g., 175 billion parameters for GPT-3).
- Storing and deploying multiple fine-tuned models for different tasks is challenging due to the large size.

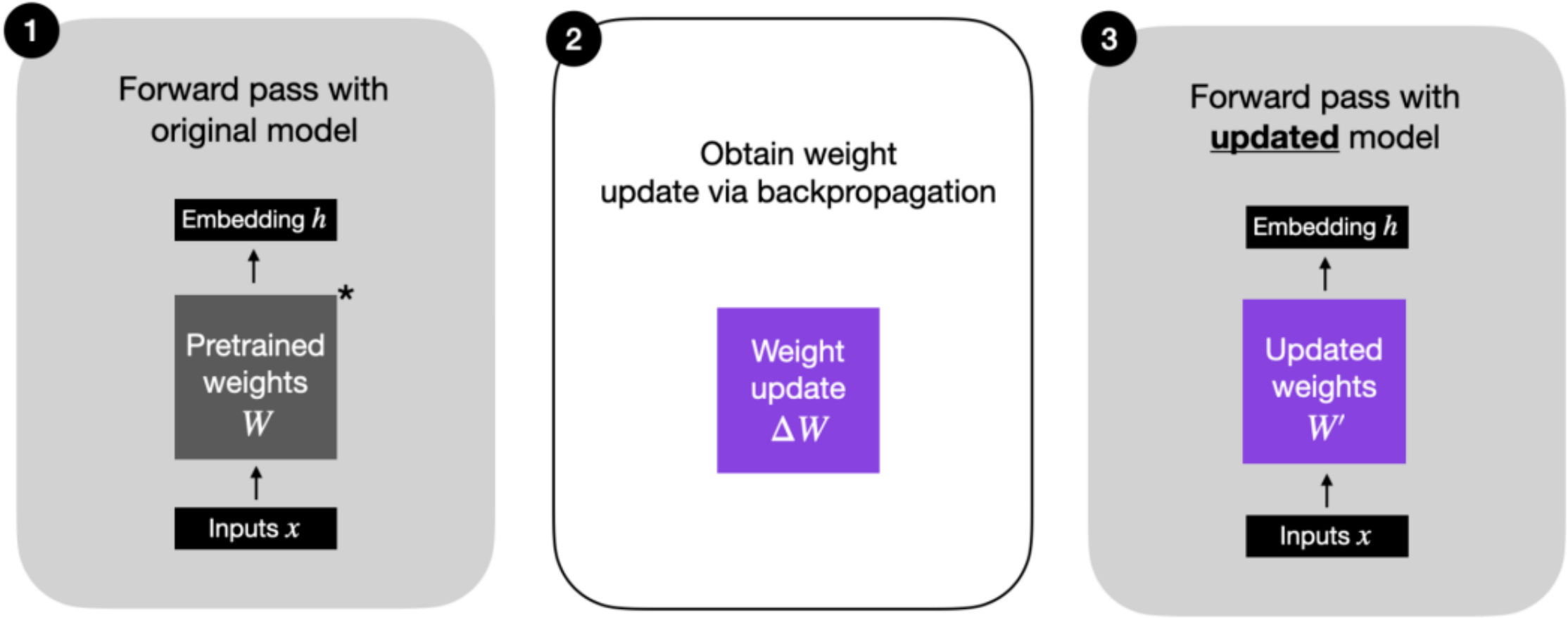
Parameter-Efficient Approach (LoRA)

1. **Reduced Parameter Set:**
   - Instead of learning a large $\Delta\Phi$ for each task, we encode the task-specific parameter increment $\Delta\Phi$ using a much smaller set of parameters Θ.
   - The size of Θ ($|\Theta|$) is much smaller than the size of the original weights ($|\Phi_0|$).
2. **Optimizing Over Θ:**
   - The task of finding $\Delta\Phi$ is now transformed into optimizing over the smaller set of parameters Θ:

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

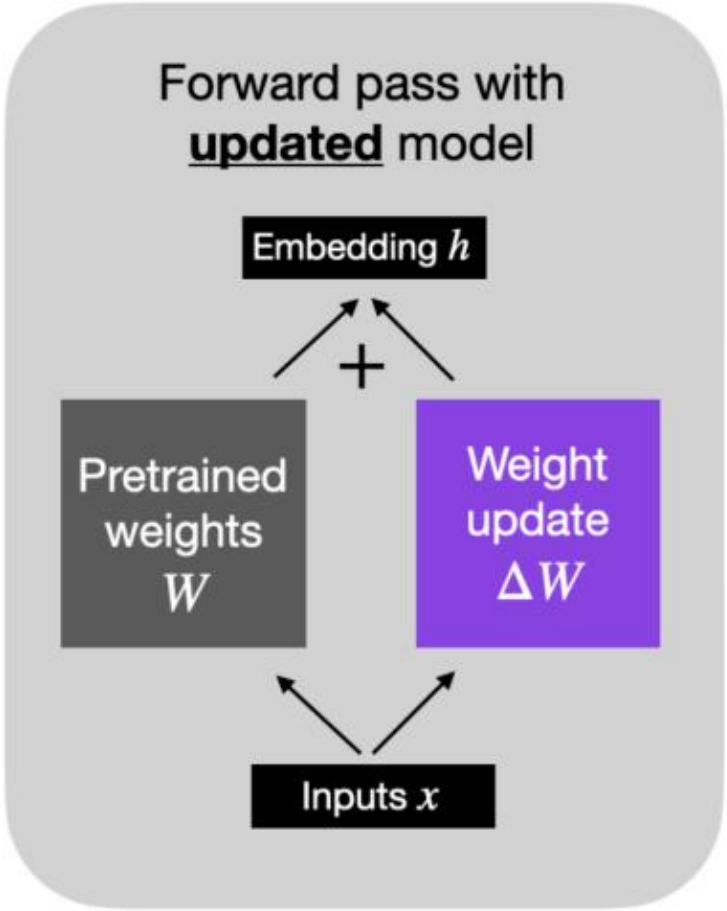
3. **Low-Rank Representation:**
   - $\Delta\Phi$ is encoded using a low-rank representation, which makes it both compute- and memory-efficient.
   - For example, if the pre-trained model is GPT-3 with 175 billion parameters, the number of trainable parameters Θ can be as small as 0.01% of the original size.

Regular Finetuning



people.cs.umass.edu/~miyyer/cs685/slides/multilingual.pdf

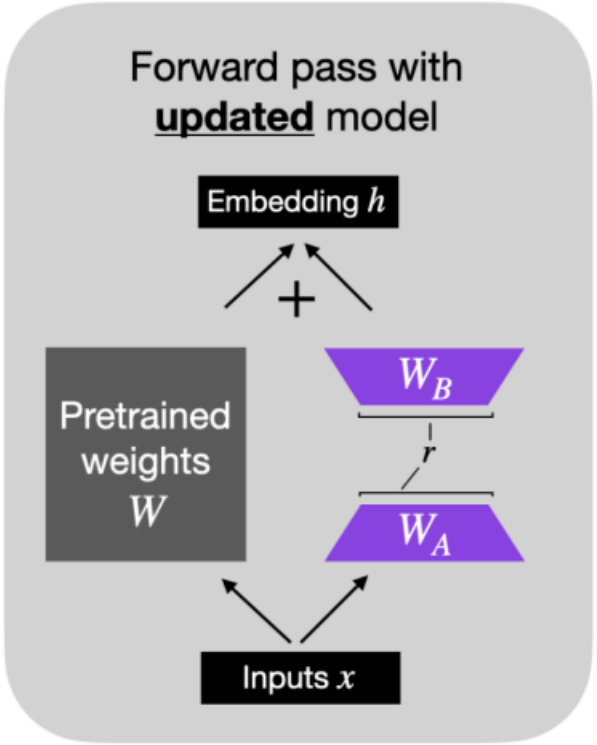
Alternative formulation (regular finetuning)

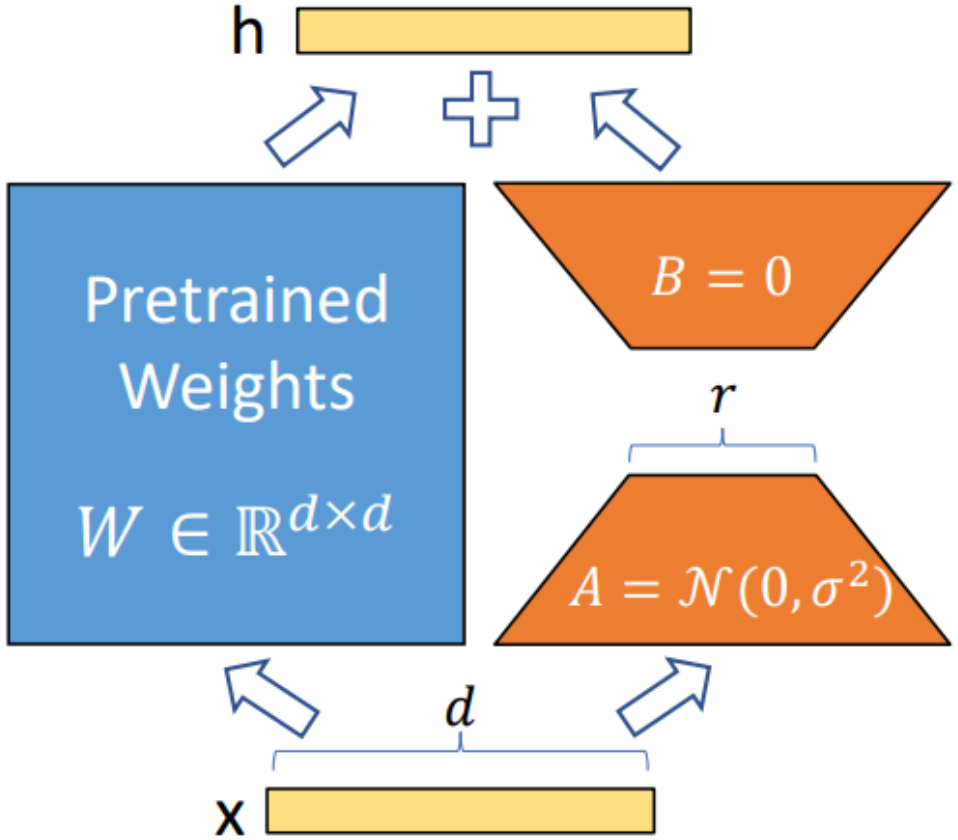


Pre-trained language models have a low intrinsic dimension

Possible to learn efficiently with a low-dimensional parameterization

LoRA weights, W_A and W_B, represent ΔW





Hu, Edward J., et al. "Lora: Low-rank adaptation of large language models." *arXiv preprint arXiv:2106.09685* (2021).

- The major downside of fine-tuning is that the new model contains as many parameters as in the original model

LoRA possesses several key advantages.

• A pre-trained model can be shared and used to build many small LoRA modules for different tasks. We can freeze the shared model and efficiently **switch tasks by replacing the matrices A and B** , reducing the storage requirement and task-switching overhead significantly.

• LoRA makes training more efficient and **lowers the hardware barrier** to entry by up to 3 times when using adaptive optimizers since we **do not need to calculate the gradients or maintain the optimizer states for most parameters**. Instead, we only optimize the injected, much smaller low-rank matrices.

• Simpler linear design allows us **to merge the trainable matrices with the frozen weights when deployed**, introducing no inference latency compared to a fully fine-tuned model, by construction.

• LoRA is orthogonal to many prior methods and **can be combined with many of them**, such as prefix-tuning

Intrinsic Dimension

Intrinsic Dimension

- **Intrinsic Dimension:** Refers to the minimal number of parameters or degrees of freedom required to effectively represent the essential information in a dataset or a model.
- **Full Dimension:** Refers to the actual number of parameters in the original matrix or model.

In the context of neural networks, even though the weight matrices have a high dimensionality, the actual effective transformations needed to adapt the model for specific tasks often lie in a lower-dimensional subspace. This is what we refer to as the intrinsic dimension.

LORA Aligns with Intrinsic Dimension

1. **Reduction to Essential Components:**
    - The low-rank approximation effectively captures the most significant components or directions of variation in the data or weight updates.
    - If W can be well-approximated by a rank-r decomposition, it suggests that the essential information in W is inherently low-dimensional.
2. **Parameter Efficiency:**
    - By using matrices A and B with dimensions $d \times r$ and $r \times d$ respectively, where $r \ll d$, we are significantly reducing the number of parameters.
    - This reduction is based on the premise that the updates needed for fine-tuning (captured by ΔW) lie within an r-dimensional subspace, which is the intrinsic dimension of the necessary adaptations.

The intrinsic dimension represents the minimal effective degrees of freedom.

By using a low-rank approximation, we are modeling the updates using only the essential parameters (aligned with the intrinsic dimension).

This approach avoids the redundancy of full-rank matrices and focuses on the core components that contribute most to the task-specific adaptation.

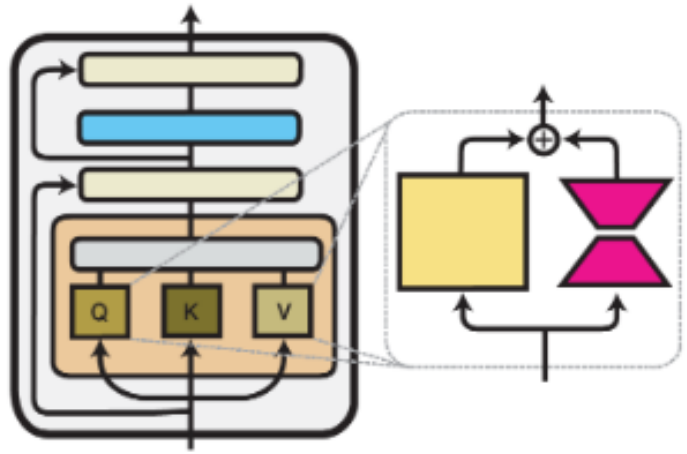
Reparametrization-based PEFT: LoRA

One-sentence idea:

Parameter update for a weight matrix in LoRA is decomposed into a product of two low-rank matrices

Pseudocode:

```
def lora_linear(x):
    h = x @ W # regular linear
    h += x @ W_A @ W_B # low-rank update
    return scale * h
```



https://arxiv.org/abs/2106.09685

Image source: adapterhub.ml

Model & Method	# Trainable Parameters	E2E NLG Challenge				
		BLEU	NIST	MET	ROUGE-L	CIDEr
GPT-2 M (FT)*	354.92M	68.2	8.62	46.2	71.0	2.47
GPT-2 M (Adapter^{L})*	0.37M	66.3	8.41	45.0	69.8	2.40
GPT-2 M (Adapter^{L})*	11.09M	68.9	8.71	46.1	71.3	2.47
GPT-2 M (Adapter^{H})	11.09M	$67.3_{\pm.6}$	$8.50_{\pm.07}$	$46.0_{\pm.2}$	$70.7_{\pm.2}$	$2.44_{\pm.01}$
GPT-2 M (FT^{Top2})*	25.19M	68.1	8.59	46.0	70.8	2.41
GPT-2 M (PreLayer)*	0.35M	69.7	8.81	46.1	71.4	2.49
GPT-2 M (LoRA)	0.35M	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.85}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{71.8}_{\pm.1}$	$\mathbf{2.53}_{\pm.02}$
GPT-2 L (FT)*	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L (Adapter^{L})	0.88M	$69.1_{\pm.1}$	$8.68_{\pm.03}$	$46.3_{\pm.0}$	$71.4_{\pm.2}$	$\mathbf{2.49}_{\pm.0}$
GPT-2 L (Adapter^{L})	23.00M	$68.9_{\pm.3}$	$8.70_{\pm.04}$	$46.1_{\pm.1}$	$71.3_{\pm.2}$	$2.45_{\pm.02}$
GPT-2 L (PreLayer)*	0.77M	70.3	8.85	46.2	71.7	2.47
GPT-2 L (LoRA)	0.77M	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.89}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{72.0}_{\pm.2}$	$2.47_{\pm.02}$

Table 3: GPT-2 medium (M) and large (L) with different adaptation methods on the E2E NLG Challenge. For all metrics, higher is better. LoRA outperforms several baselines with comparable or fewer trainable parameters. Confidence intervals are shown for experiments we ran. * indicates numbers published in prior works.

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB_{base} (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB_{base} (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB_{base} (Adpt^{D})*	0.3M	$87.1_{\pm.0}$	$94.2_{\pm.1}$	$88.5_{\pm1.1}$	$60.8_{\pm.4}$	$93.1_{\pm.1}$	$90.2_{\pm.0}$	$71.5_{\pm2.7}$	$89.7_{\pm.3}$	84.4
RoB_{base} (Adpt^{D})*	0.9M	$87.3_{\pm.1}$	$94.7_{\pm.3}$	$88.4_{\pm.1}$	$62.6_{\pm.9}$	$93.0_{\pm.2}$	$90.6_{\pm.0}$	$75.9_{\pm2.2}$	$90.3_{\pm.1}$	85.4
RoB_{base} (LoRA)	0.3M	$87.5_{\pm.3}$	$\mathbf{95.1}_{\pm.2}$	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm.3}$	$90.8_{\pm.1}$	$\mathbf{86.6}_{\pm.7}$	$\mathbf{91.5}_{\pm.2}$	**87.2**
$\text{RoB}_{\text{large}}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
$\text{RoB}_{\text{large}}$ (LoRA)	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.9}_{\pm1.2}$	$\mathbf{68.2}_{\pm1.9}$	$\mathbf{94.9}_{\pm.3}$	$91.6_{\pm.1}$	$\mathbf{87.4}_{\pm2.5}$	$\mathbf{92.6}_{\pm.2}$	**89.0**
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	3.0M	$90.2_{\pm.3}$	$96.1_{\pm.3}$	$90.2_{\pm.7}$	$\mathbf{68.3}_{\pm1.0}$	$\mathbf{94.8}_{\pm.2}$	$\mathbf{91.9}_{\pm.1}$	$83.8_{\pm2.9}$	$92.1_{\pm.7}$	88.4
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	0.8M	$\mathbf{90.5}_{\pm.3}$	$\mathbf{96.6}_{\pm.2}$	$89.7_{\pm1.2}$	$67.8_{\pm2.5}$	$\mathbf{94.8}_{\pm.3}$	$91.7_{\pm.2}$	$80.1_{\pm2.9}$	$91.9_{\pm.4}$	87.9
$\text{RoB}_{\text{large}}$ (Adpt^{H})†	6.0M	$89.9_{\pm.5}$	$96.2_{\pm.3}$	$88.7_{\pm2.9}$	$66.5_{\pm4.4}$	$94.7_{\pm.2}$	$92.1_{\pm.1}$	$83.4_{\pm1.1}$	$91.0_{\pm1.7}$	87.8
$\text{RoB}_{\text{large}}$ (Adpt^{H})†	0.8M	$90.3_{\pm.3}$	$96.3_{\pm.5}$	$87.7_{\pm1.7}$	$66.3_{\pm2.0}$	$94.7_{\pm.2}$	$91.5_{\pm.1}$	$72.9_{\pm2.9}$	$91.5_{\pm.5}$	86.4
$\text{RoB}_{\text{large}}$ (LoRA)†	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.2}_{\pm1.0}$	$68.2_{\pm1.9}$	$\mathbf{94.8}_{\pm.3}$	$91.6_{\pm.2}$	$\mathbf{85.2}_{\pm1.1}$	$\mathbf{92.3}_{\pm.5}$	**88.6**
DeB_{XXL} (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeB_{XXL} (LoRA)	4.7M	$\mathbf{91.9}_{\pm.2}$	$96.9_{\pm.2}$	$\mathbf{92.6}_{\pm.6}$	$\mathbf{72.4}_{\pm1.1}$	$\mathbf{96.0}_{\pm.1}$	$\mathbf{92.9}_{\pm.1}$	$\mathbf{94.9}_{\pm.4}$	$\mathbf{93.0}_{\pm.2}$	**91.3**

Table 2: $\text{RoBERTa}_{\text{base}}$, $\text{RoBERTa}_{\text{large}}$, and $\text{DeBERTa}_{\text{XXL}}$ with different adaptation methods on the GLUE benchmark. We report the overall (matched and mismatched) accuracy for MNLI, Matthew's correlation for CoLA, Pearson correlation for STS-B, and accuracy for other tasks. Higher is better for all metrics. * indicates numbers published in prior works. † indicates runs configured in a setup similar to Houlsby et al. (2019) for a fair comparison.

QLoRA

Courtesy:, deeplearning.ai , https://arxiv.org/pdf/2305.14314

what does Computation mean?

Computation refers to the **mathematical operations** that are performed on the weights and activations of the network during both the forward pass (when making predictions) and the backward pass (when updating the weights during training).

In a typical neural network, these computations are performed using **32-bit floating- point numbers**. This is because 32-bit floating-point numbers provide a good balance between **precision** (the ability to represent numbers accurately) and **range** (the range of numbers that can be represented). Using 32-bit floating-point numbers for all computations can be **memory-intensive.**

This is where quantization comes in.

Quantization is a technique to **reduce the precision** of the numbers used in the model. In the case of 4-bit quantization, the weights and activations of the network are compressed from 32-bit floating-point numbers to 4-bit integers. A 4-bit integer can range from -8 to 7.

QLoRA extends LoRA to enhance efficiency by quantizing weight values of the original network, from high-resolution data types, such as Float32, to lower-resolution data types like int4. This leads to reduced memory demands and faster calculations.

*Our best model family, which we name **Guanaco**, outperforms all previous openly released models on the Vicuna benchmark, reaching 99.3% of the performance level of ChatGPT while only requiring 24 hours of finetuning on a single GPU*

https://arxiv.org/pdf/2305.14314

QLoRA: Efficient Finetuning of Quantized LLMs

Tim Dettmers* **Artidoro Pagnoni*** **Ari Holtzman**

Luke Zettlemoyer

University of Washington
{dettmers,artidoro,ahai,lsz}@cs.washington.edu

Abstract

We present QLoRA, an efficient finetuning approach that reduces memory usage enough to finetune a 65B parameter model on a single 48GB GPU while preserving full 16-bit finetuning task performance. QLoRA backpropagates gradients through a frozen, 4-bit quantized pretrained language model into Low Rank Adapters (LoRA). Our best model family, which we name **Guanaco**, outperforms all previous openly released models on the Vicuna benchmark, reaching 99.3% of the performance level of ChatGPT while only requiring 24 hours of finetuning on a single GPU. QLoRA introduces a number of innovations to save memory without sacrificing performance: (a) 4-bit NormalFloat (NF4), a new data type that is information theoretically optimal for normally distributed weights (b) Double Quantization to reduce the average memory footprint by quantizing the quantization constants, and (c) Paged Optimizers to manage memory spikes. We use QLoRA to finetune more than 1,000 models, providing a detailed analysis of instruction following and chatbot performance across 8 instruction datasets, multiple model types (LLaMA, T5), and model scales that would be infeasible to run with regular finetuning (e.g. 33B and 65B parameter models). Our results show that QLoRA finetuning on a small high-quality dataset leads to state-of-the-art results, even when using smaller models than the previous SoTA. We provide a detailed analysis of chatbot performance based on both human and GPT-4 evaluations showing that GPT-4 evaluations are a cheap and reasonable alternative to human evaluation. Furthermore, we find that current chatbot benchmarks are not trustworthy to accurately evaluate the performance levels of chatbots. A lemon-picked analysis demonstrates where **Guanaco** fails compared to ChatGPT. We release all of our models and code, including CUDA kernels for 4-bit training.[2]

QLoRA

QLoRA is the extended version of LoRA which works by quantizing the precision of the weight parameters in the pre trained LLM to 4-bit precision. Typically, parameters of trained models are stored in a 32-bit format, but QLoRA compresses them to a 4-bit format. This reduces the memory footprint of the LLM, making it possible to finetune it on a single GPU. This method significantly **reduces the memory footprint, making it feasible to run LLM models on less powerful hardware, including consumer GPUs.**

According to QLoRA paper:

QLORA introduces **multiple innovations** designed to reduce memory use without sacrificing performance:

(1) 4-bit NormalFloat, an information theoretically optimal quantization data type for normally distributed data that yields better empirical results than 4-bit Integers and 4-bit Floats.

(2) Double Quantization, a method that quantizes the quantization constants, saving an average of about 0.37 bits per parameter (approximately 3 GB for a 65B model).

(3) Paged Optimizers, using NVIDIA unified memory to avoid the gradient checkpointing memory spikes that occur when processing a mini-batch with a long sequence length.

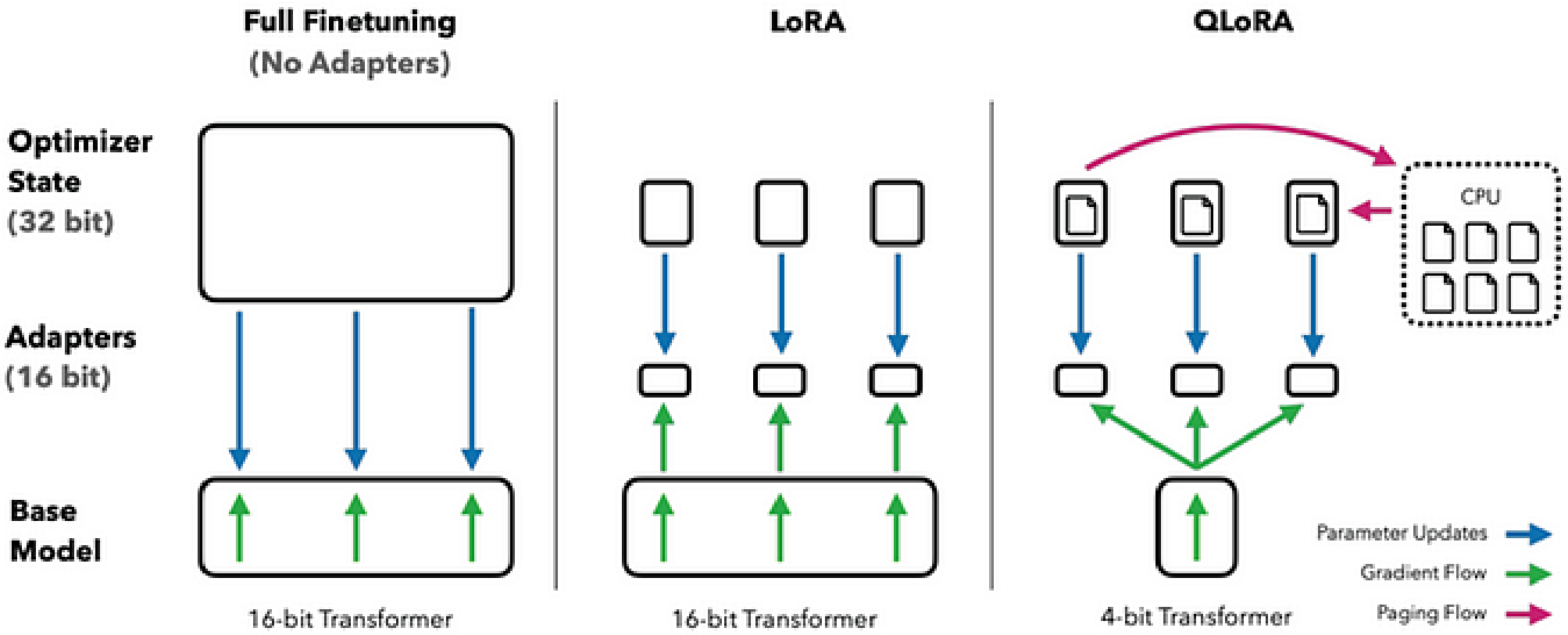


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

QLORA Finetuning

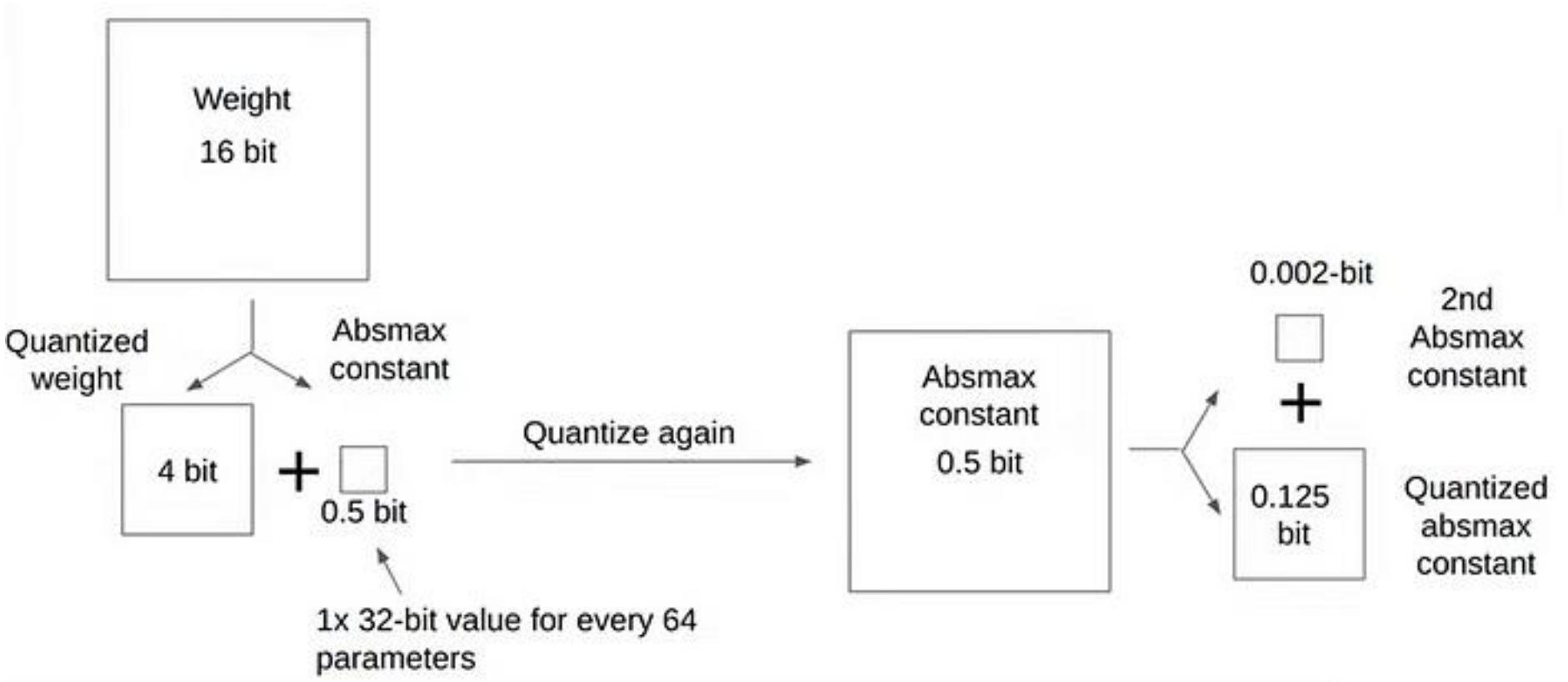


Image source: *Democratizing Foundation Models via k-bit Quantization by Tim Dettmers*

QLoRA Steps

- Normalisation
- Quantization
- Dequantization
- Double Quantization
  - First Level Quantization
  - Quantizing the quantization constants
  - Dequantize the constants
  - Dequantize the parameters using dequantized constants
- Unified Memory Paging/Paged Optimisers

Normalisation

1.Normalization

The weights of the model are first normalized to have zero mean and unit variance. This ensures that the weights are distributed around zero and fall within a certain range.

Quantization

Step 1: Original Weights

Assume we have a small block of weights:

$$W_{\text{original}} = \{0.35, -1.2, 2.7, -0.45\}$$

Step 2: Determine Quantization Range

Find the minimum and maximum values:

$$W_{\min} = -1.2, \quad W_{\max} = 2.7$$

Step 3: Calculate Scale Factor and Zero Point

1. **Calculate Scale Factor:**

   - The scale factor is calculated to map the range of weights to the range of 4-bit integers (0 to 15).
   - The range of the weights is:

$$\text{Range} = W_{\max} - W_{\min} = 2.7 - (-1.2) = 3.9$$

   - The scale factor is:

$$\text{Scale Factor} = \frac{\text{Range}}{15} = \frac{3.9}{15} \approx 0.26$$

Quantization

2. **Calculate Zero Point:**

- The zero point is the value that maps to the minimum weight:

$$\text{Zero Point} = \text{round}\left(\frac{-W_{\text{min}}}{\text{Scale Factor}}\right) = \text{round}\left(\frac{1.2}{0.26}\right) \approx 5$$

Quantization

QLoRA Steps

4. **Quantize the Parameters:**

- Using the scale factor and zero point, we convert the original values to 4-bit integers:

$$W_{\text{quantized}} = \text{round}\left(\frac{W_{\text{original}}}{\text{Scale Factor}} + \text{Zero Point}\right)$$

Applying this to each weight:

$$0.35 \rightarrow \text{round}\left(\frac{0.35}{0.26} + 5\right) = \text{round}(1.346 + 5) = 6$$

$$-1.2 \rightarrow \text{round}\left(\frac{-1.2}{0.26} + 5\right) = \text{round}(-4.615 + 5) = 0$$

$$2.7 \rightarrow \text{round}\left(\frac{2.7}{0.26} + 5\right) = \text{round}(10.385 + 5) = 15$$

$$-0.45 \rightarrow \text{round}\left(\frac{-0.45}{0.26} + 5\right) = \text{round}(-1.731 + 5) = 3$$

- So, the quantized values are:

$$W_{\text{quantized}} = \{6, 0, 15, 3\}$$

↓

5. **Dequantize the Parameters:**

- To convert the quantized values back to their approximate original values, use the scale factor and zero point:

$$W_{\text{dequantized}} = (W_{\text{quantized}} - \text{Zero Point}) \times \text{Scale Factor}$$

Applying this to each quantized weight:

$$6 \rightarrow (6 - 5) \times 0.26 = 1 \times 0.26 = 0.26$$
$$0 \rightarrow (0 - 5) \times 0.26 = -5 \times 0.26 = -1.3$$
$$15 \rightarrow (15 - 5) \times 0.26 = 10 \times 0.26 = 2.6$$
$$3 \rightarrow (3 - 5) \times 0.26 = -2 \times 0.26 = -0.52$$

$$W_{\text{original}} = \{0.35, -1.2, 2.7, -0.45\}$$

- The dequantized values approximate the original values, but with some loss of precision:

$$W_{\text{dequantized}} = \{0.26, -1.3, 2.6, -0.52\}$$

Double Quantization

Step 1: First Level Quantization (Recap)

We already quantized our original parameters:

1. **Original Parameters (High Precision):**

$$W_{\text{original}} = \{0.35, -1.2, 2.7, -0.45\}$$

2. **Determine Quantization Range:**

$$W_{\text{min}} = -1.2, \quad W_{\text{max}} = 2.7$$

3. **Calculate Scale Factor and Zero Point:**

$$\text{Scale Factor} = 0.26, \quad \text{Zero Point} = 5$$

4. **Quantize the Parameters:**

$$W_{\text{quantized}} = \{6, 0, 15, 3\}$$

Double Quantization

Step 2: Quantizing the Quantization Constants

Now, we'll apply a second level of quantization to the scale factor and zero point.

1. **Quantization Constants:**
    - Scale Factor: 0.26
    - Zero Point: 5
2. **Determine Range for Constants:**
    - Assume we need to store the scale factor and zero point with even lower precision. Let's represent these using 4-bit integers again.

Double Quantization

3. **Quantize the Scale Factor:**

- Suppose the possible range for the scale factor (0.26 in this case) is from 0 to 1.
- We divide this range into 16 steps (for 4-bit):

$$\text{Scale Factor Steps} = \frac{1-0}{15} = 0.0667$$

- Calculate quantized value for the scale factor:

$$\text{Quantized Scale Factor} = \text{round}\left(\frac{0.26}{0.0667}\right) = \text{round}(3.9) = 4$$

Double Quantization

4. **Quantize the Zero Point:**

- Assume the range for the zero point (5) is from 0 to 10.
- We divide this range into 16 steps (for 4-bit):

$$\text{Zero Point Steps} = \frac{10 - 0}{15} = 0.6667$$

- Calculate quantized value for the zero point:

$$\text{Quantized Zero Point} = \text{round}\left(\frac{5}{0.6667}\right) = \text{round}(7.5) = 8$$

Double Quantization

Step 3: Dequantize the Constants

To use these quantized constants, we need to dequantize them back:

1. **Dequantize the Scale Factor:**

$$\text{Dequantized Scale Factor} = 4 \times 0.0667 = 0.2668$$

2. **Dequantize the Zero Point:**

$$\text{Dequantized Zero Point} = 8 \times 0.6667 = 5.3336$$

Double Quantization

QLoRA Steps

Step 4: Dequantize the Parameters Using Dequantized Constants

Now, use the dequantized constants to dequantize the parameters:

1. **Dequantize the Parameters:**

$$W_{\text{dequantized}} = (W_{\text{quantized}} - \text{Dequantized Zero Point}) \times \text{Dequantized Scale Factor}$$

Applying this to each quantized weight:

$$\begin{aligned}
6 &\rightarrow (6 - 5.3336) \times 0.2668 = 0.6664 \times 0.2668 = 0.1778 \\
0 &\rightarrow (0 - 5.3336) \times 0.2668 = -5.3336 \times 0.2668 = -1.423 \\
15 &\rightarrow (15 - 5.3336) \times 0.2668 = 9.6664 \times 0.2668 = 2.579 \\
3 &\rightarrow (3 - 5.3336) \times 0.2668 = -2.3336 \times 0.2668 = -0.6225
\end{aligned}$$

The dequantized values are:

$$W_{\text{dequantized}} = \{0.1778, -1.423, 2.579, -0.6225\}$$

Paged Optimisers/Unified Memory paging

Paged optimizers are designed to efficiently manage memory usage, especially when training large models with long sequence lengths. They leverage memory paging techniques and NVIDIA's unified memory to avoid out-of-memory (OOM) errors and optimize performance

Paged Optimisers

Memory Challenges in Training LLMs

Training large language models involves handling massive amounts of data and numerous parameters, which can lead to significant memory usage. The main memory challenges include:

- **Memory Spikes**: These occur during gradient calculations, especially with long sequences, causing temporary peaks in memory usage.
- **Gradient Checkpointing**: This technique saves intermediate activations at specific points to reduce memory usage but can still cause spikes.

How Paged Optimizers Work

1. **Unified Memory**: Paged optimizers utilize NVIDIA's unified memory, which **allows a GPU to use both its own VRAM and the system RAM**. This approach provides a larger memory pool, reducing the risk of OOM errors when the GPU's VRAM is insufficient.

2. **Paging System**: Similar to how operating systems manage memory, paged optimizers use a **paging system to dynamically allocate and deallocate memory**. When processing a mini-batch, only the required portions of the model and data are loaded into the GPU memory, while the rest remain in the system RAM.

How Paged Optimizers Help

1. Avoiding Memory Spikes: By **spreading the memory usage more evenly between the GPU memory and system RAM**, paged optimizers help in mitigating memory spikes. This approach ensures that large models and long sequences can be processed without causing sudden peaks in memory usage that could lead to OOM errors.

2. Efficiency in Training: The optimizer **dynamically pages data in and out of GPU memory, ensuring that the GPU is not overwhelmed** by large memory demands at any given time. This dynamic management makes it possible to handle larger models or longer sequences than what would typically fit into the GPU's VRAM alone.

Benefits of Paged Optimizers

- **Increased Model Size**: Allows for training larger models than what the GPU VRAM alone would permit.
- **Handling Long Sequences**: Enables processing of longer sequences without running into OOM errors.
- **Efficient Memory Use**: Optimizes memory allocation dynamically, improving training efficiency.

Namah Shivaya