AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

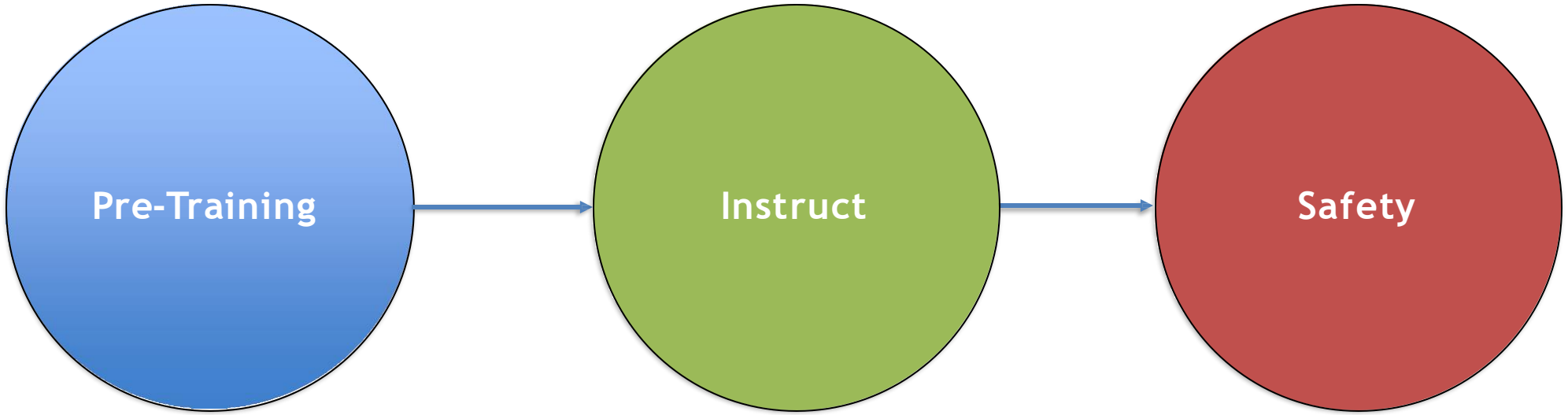
Fine Tuning

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

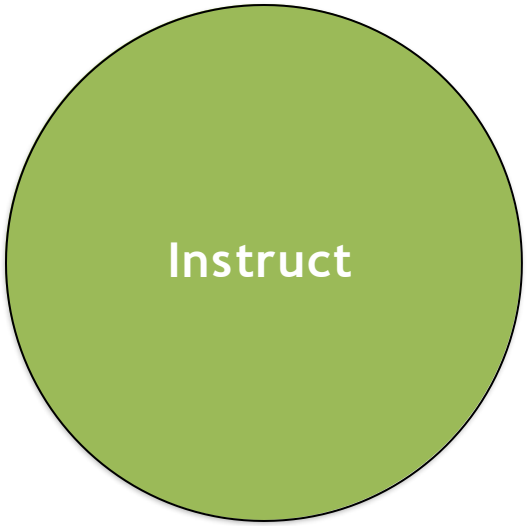
self-supervised

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



supervised fine-tuning

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.

This also includes RLHF.

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.

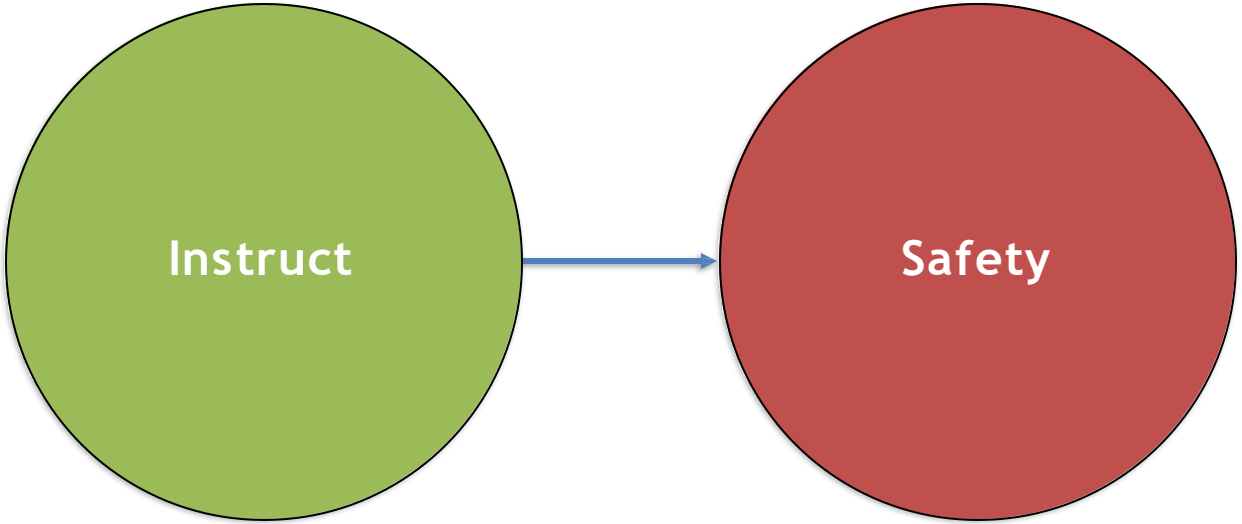
Process:
Involves further fine-tuning. We use RLHF to provide feedback on model outputs.

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

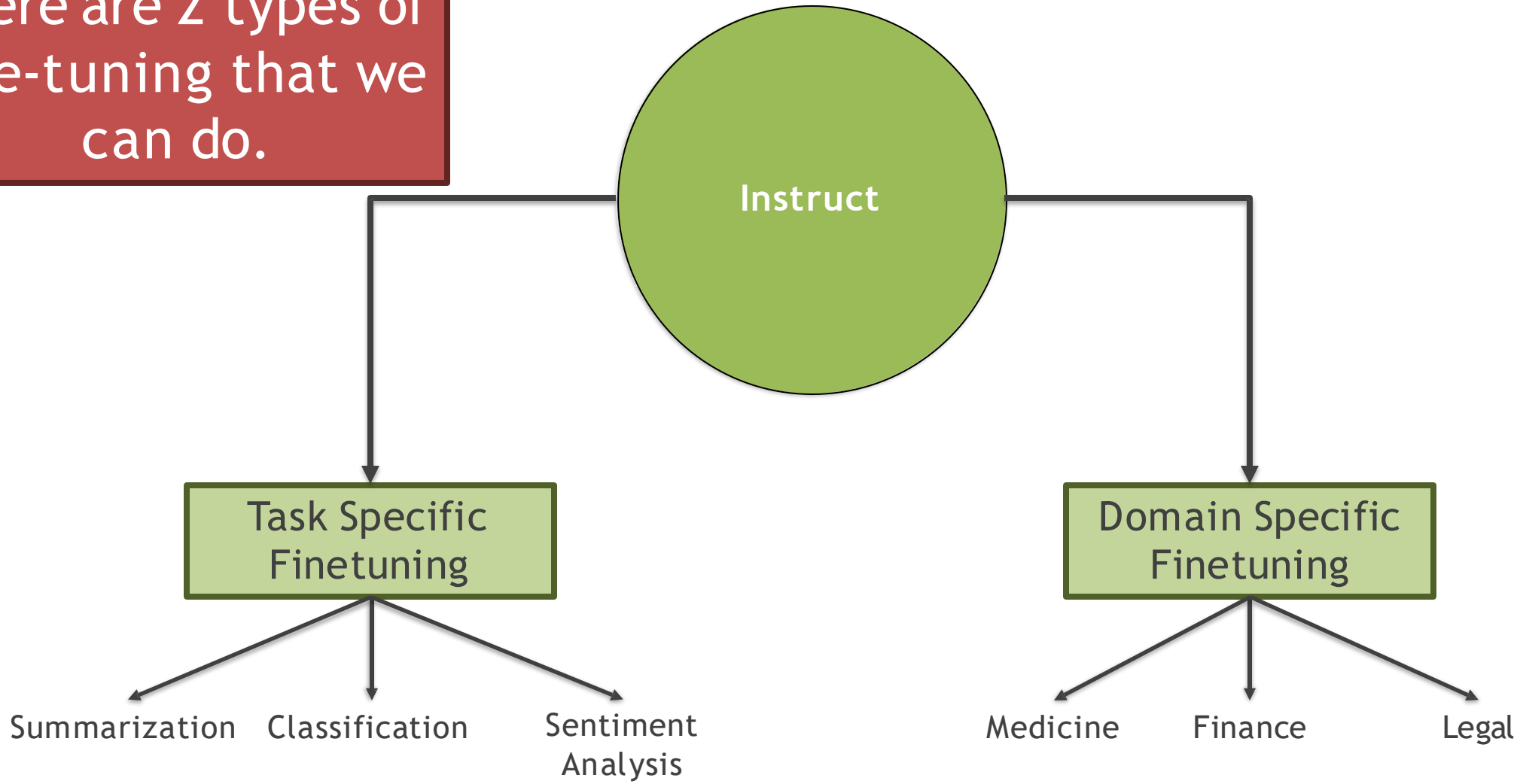
So, fine-tuning takes place in 2 stages.



In this lecture, we will be focusing on the Instruct stage of fine-tuning.

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
    - **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

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": "Summarize the following article in one sentence.",
  "input": "The global climate summit held this week saw leaders from over 100 countries commit
  "output": "World leaders pledged major carbon cuts by 2030 at this week's climate summit."
}
```

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.

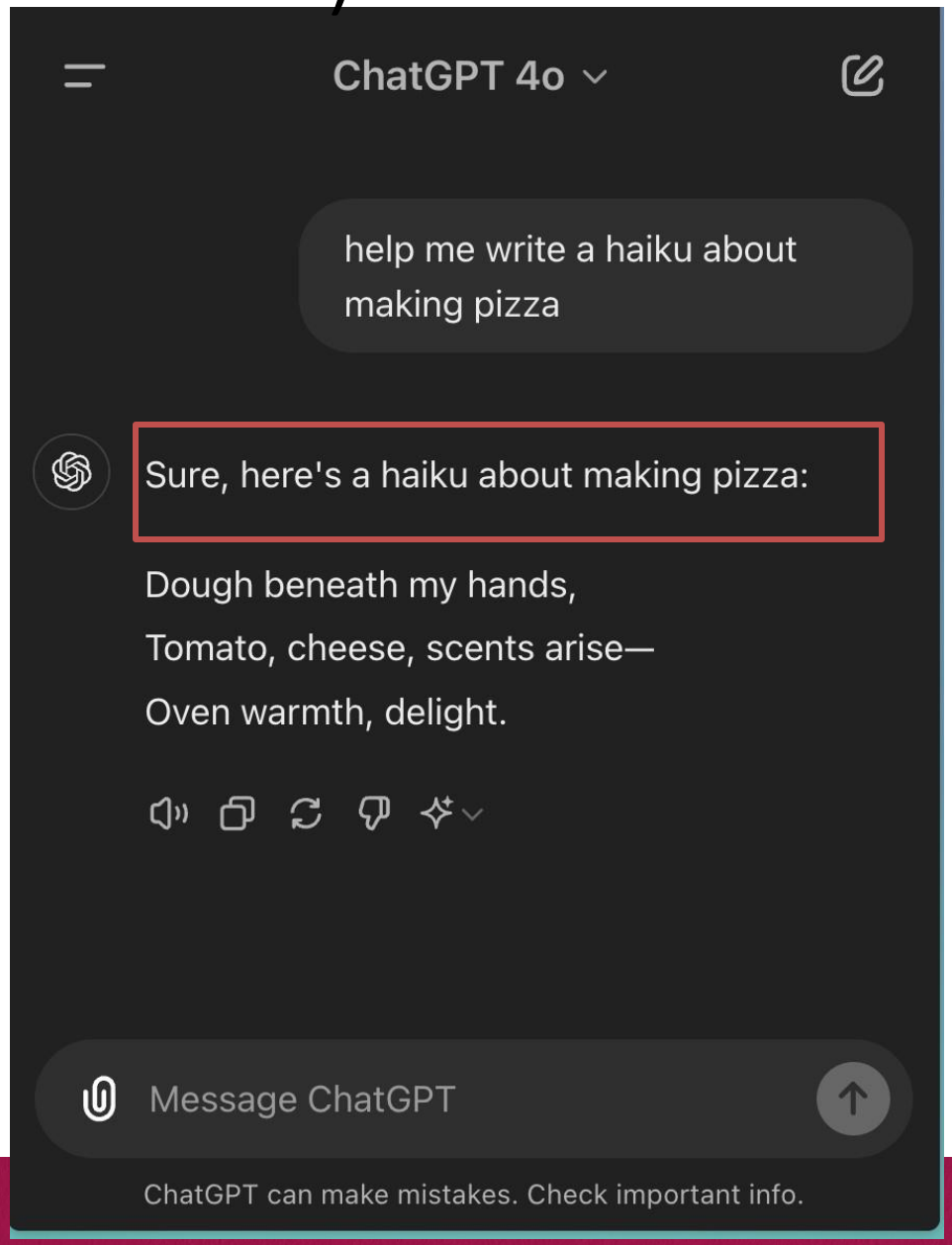
For e.g.: Summarize this, translate that, etc

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

Instruction-tuning (Full Parameter)

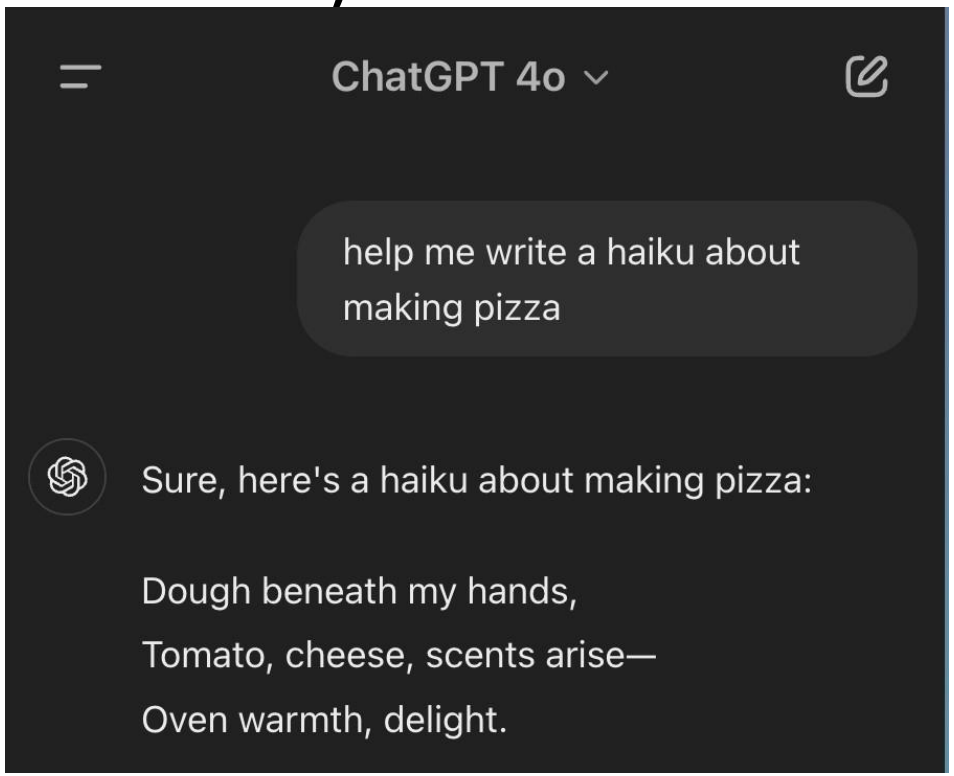
For e.g., if you train a model specifically for question answering:

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.

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)

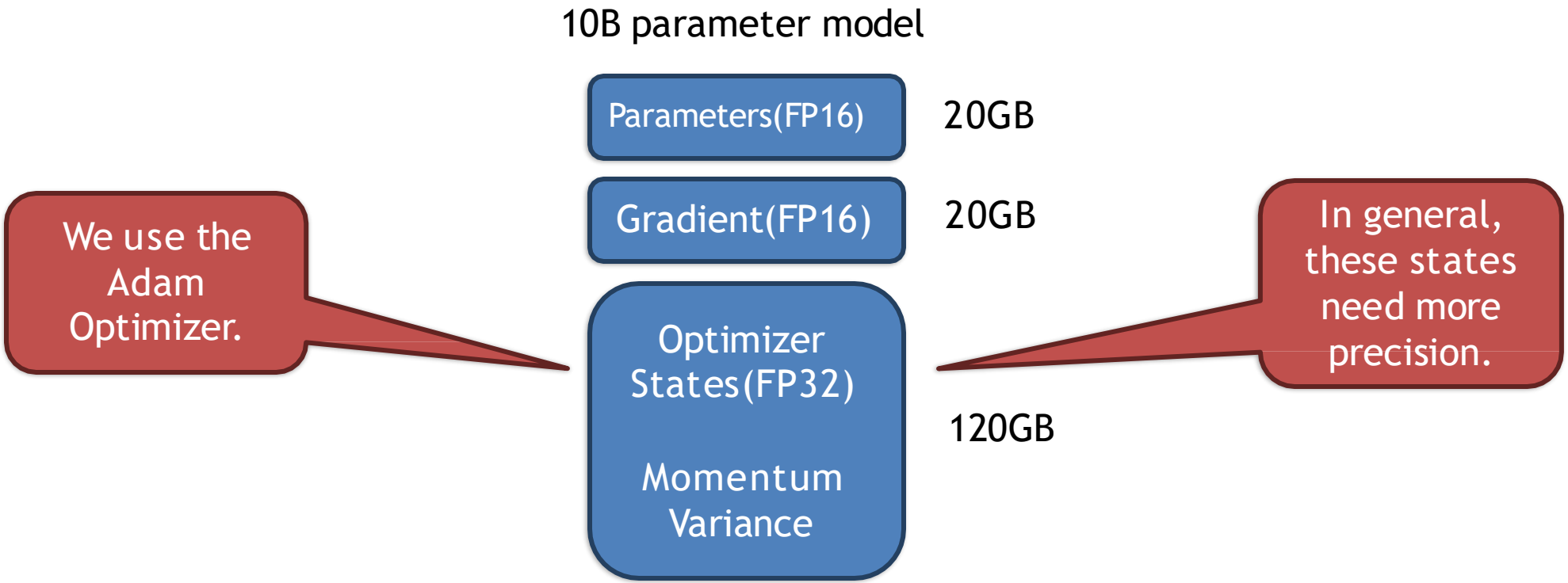
Let’s take a fine-tuning example now.

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.

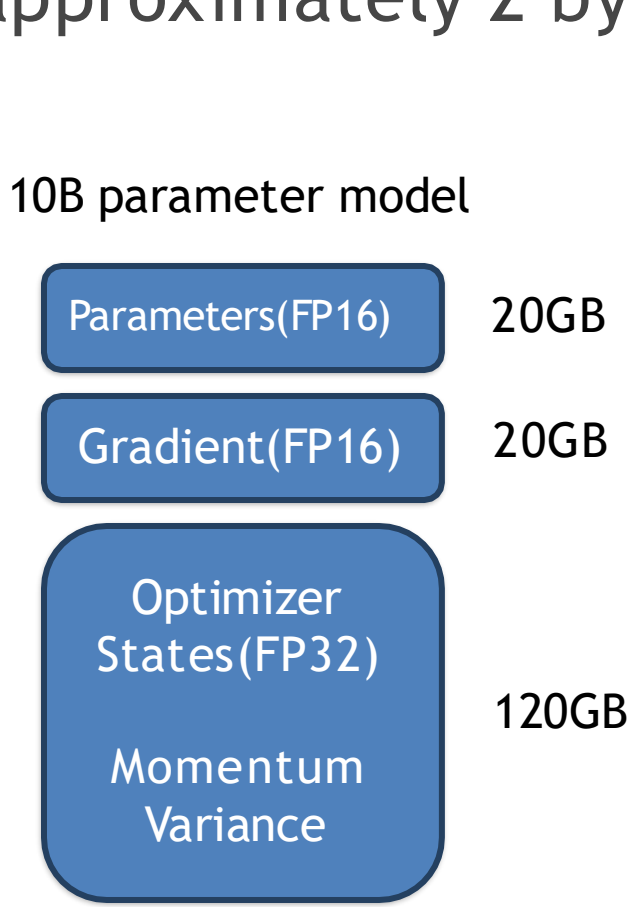
Instruction-tuning (Full Parameter)

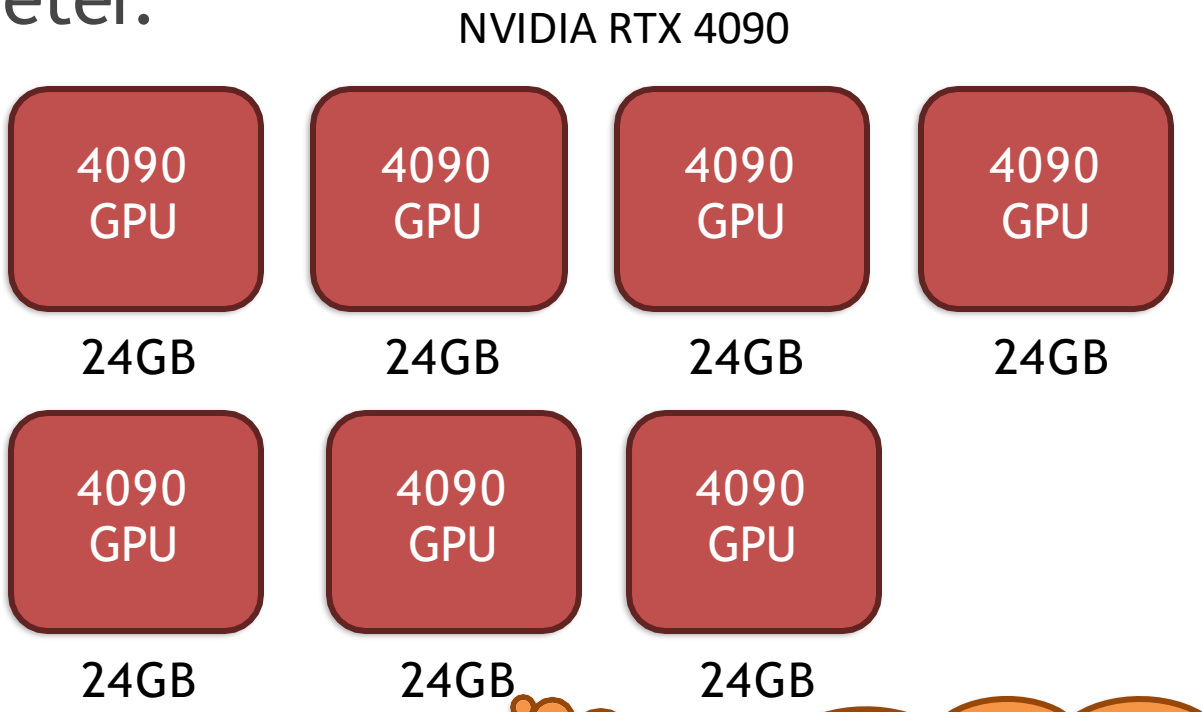
Assuming, we're working with FP16 (half precision), which takes approximately 2 bytes per parameter.



Instruction-tuning (Full Parameter)

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





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

Instruction-tuning (Full Parameter)

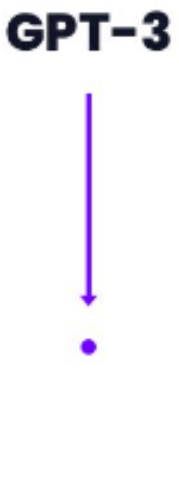
This makes full parameter finetuning inaccessible to normal folks like us.

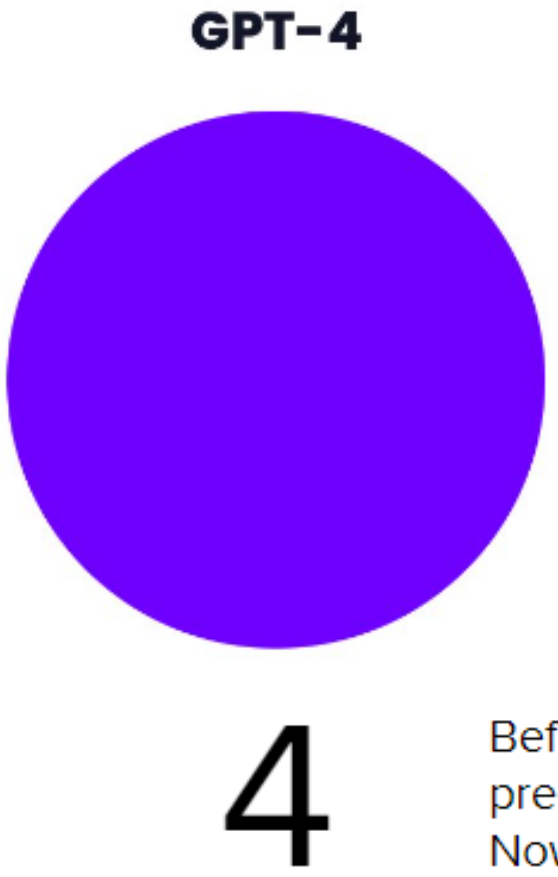
So, what can we do?

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
```

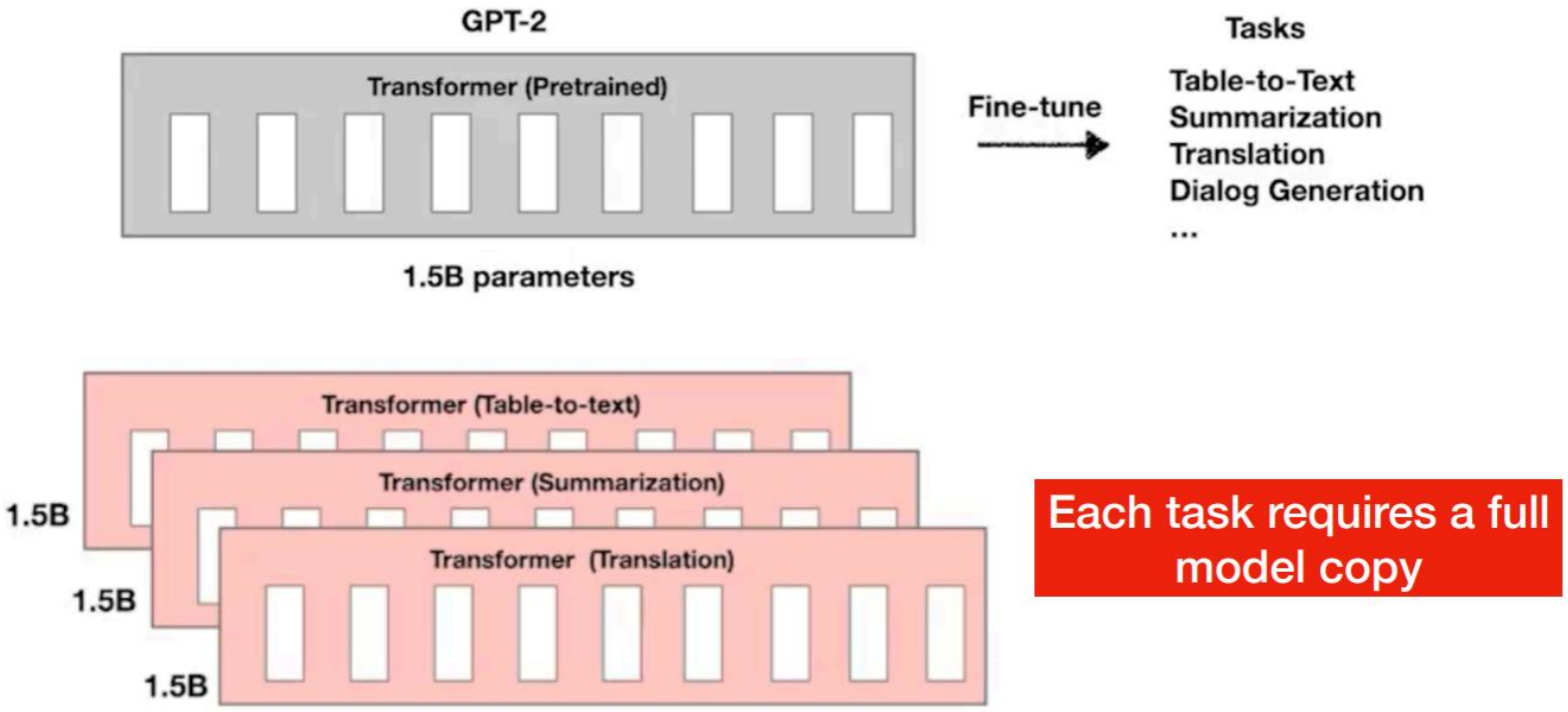
Does this work? Yes!

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 ($Adapter^H$)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 ($Adapter^H$)	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

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

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)

Instruction-tuning (PEFT)

PEFT stands for Parameter Efficient Finetuning.

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

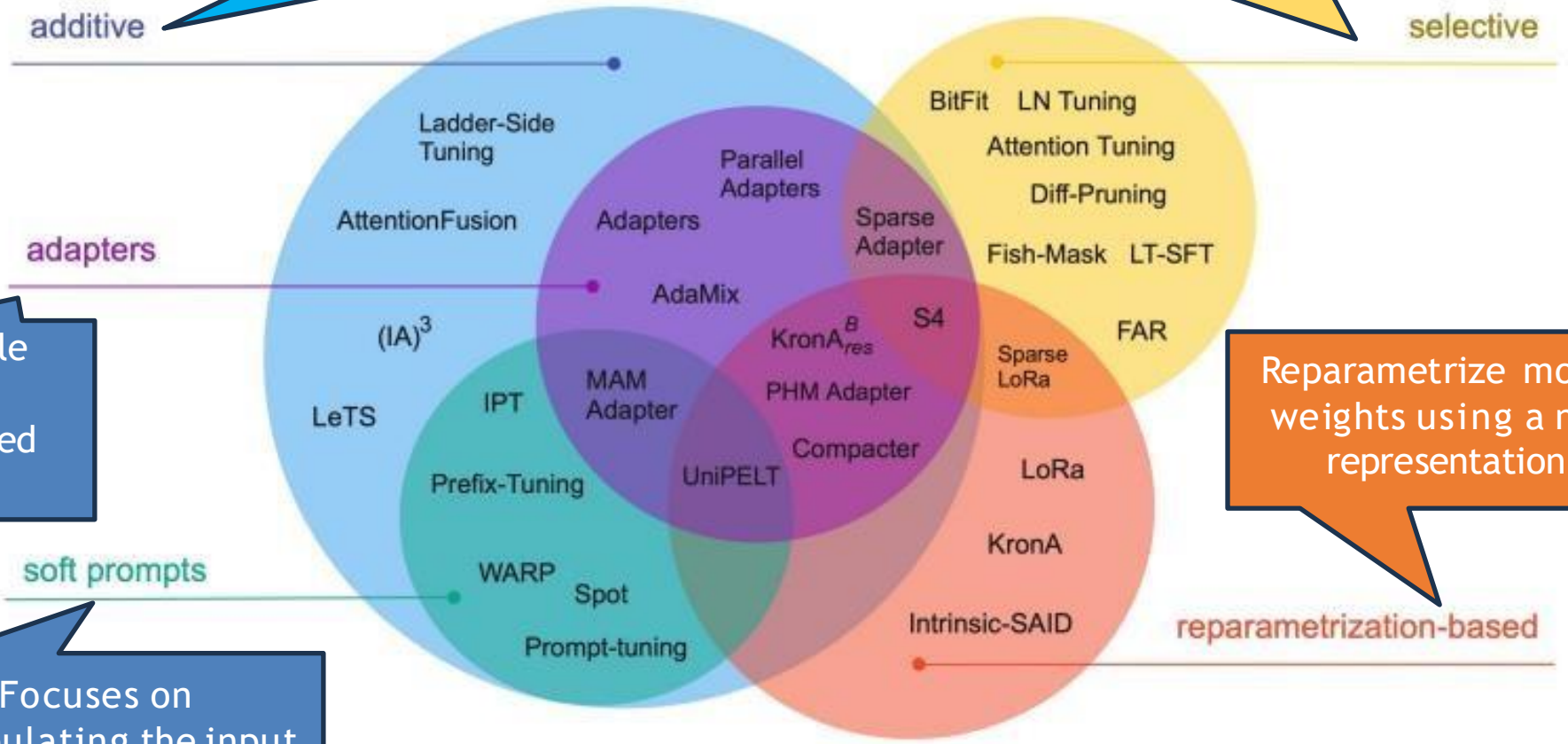
There are majorly three methods to do PEFT.

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]

- 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

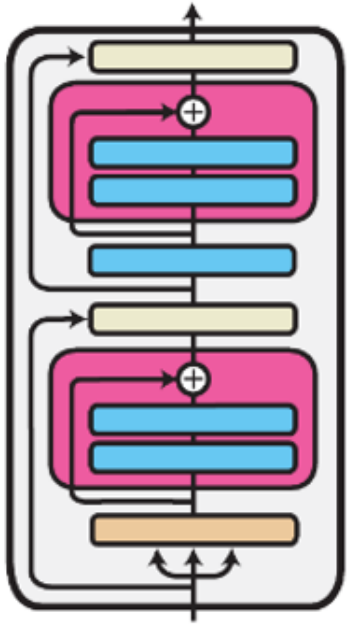
Additive Methods

- 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

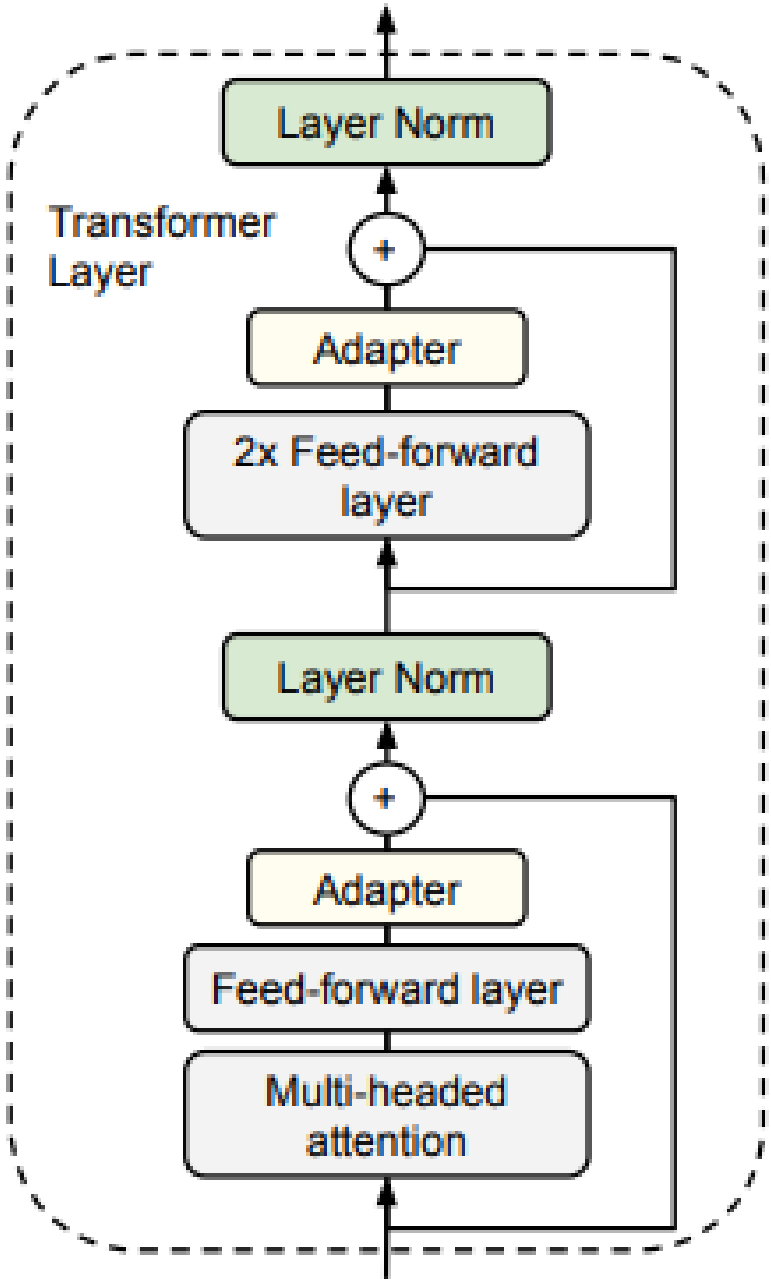
This technique was introduced in Houlsby et al [4]. The goal of adapters is to add small fully connected networks after Transformer sub-layers and learn those parameters

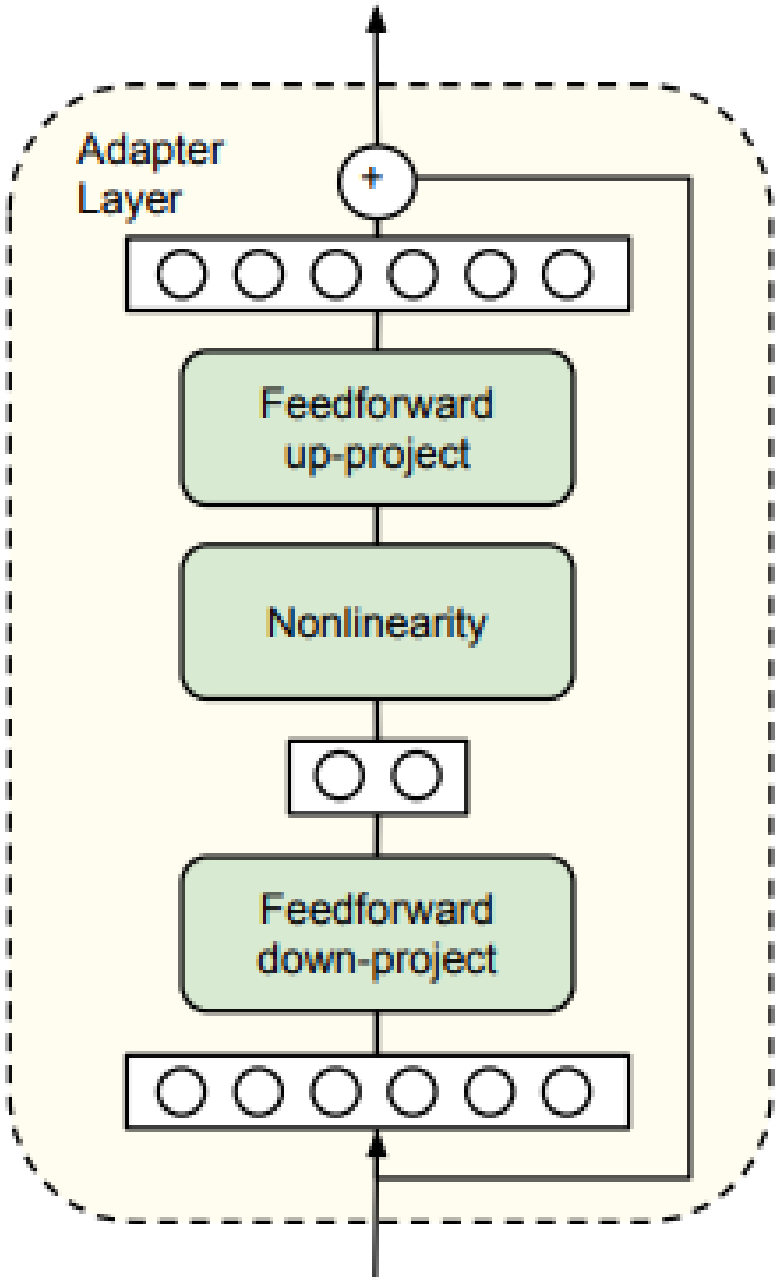
```
def transformer_block_adapter(x):
    """Pseudo code from [2] """
    residual = x
    x = self_attention(x)
    x = FFN(x)  # adapter
    x = layer_norm(x + residual)
    residual = x
    x = FFN(x)
    x = FFN(x)  # adapter
    x = layer_norm(x + residual)
    return x
```



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:

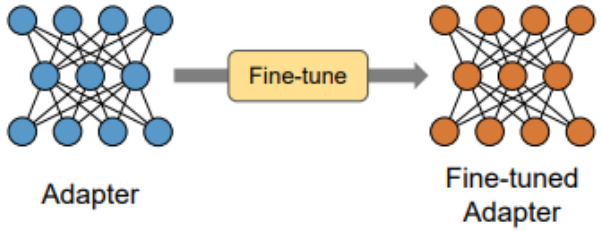
$$h' = W_{up} * z$$

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

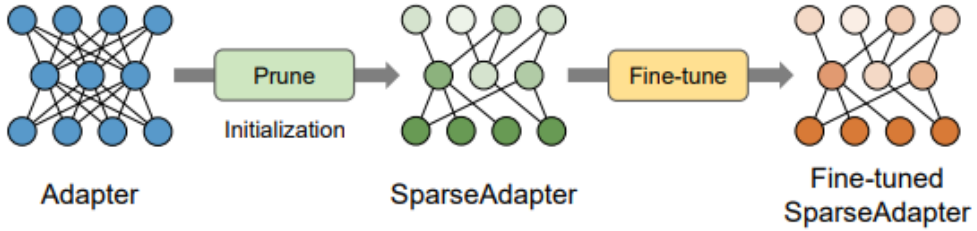
Outline

- 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

- 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).

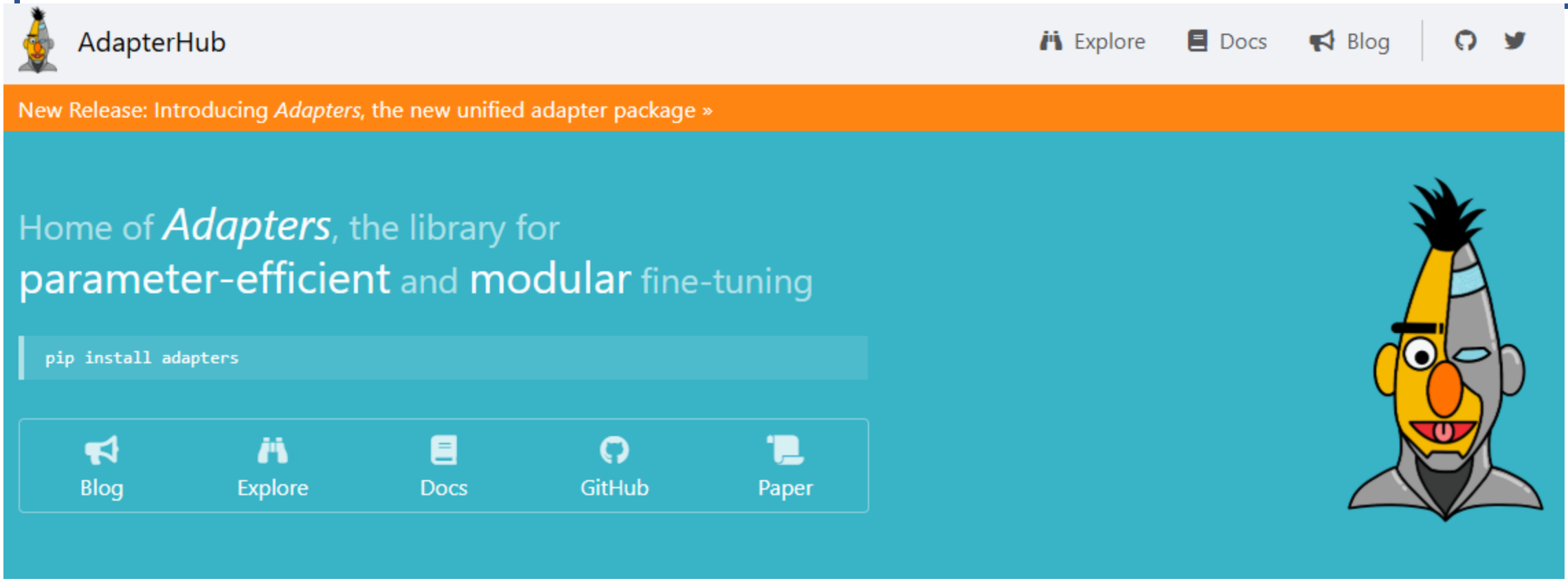
Choosing a Sparsity Technique:

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.



Outline

- 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 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

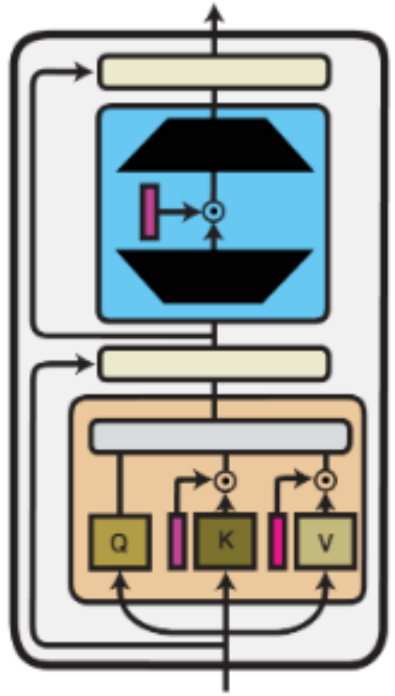
One-sentence idea:

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

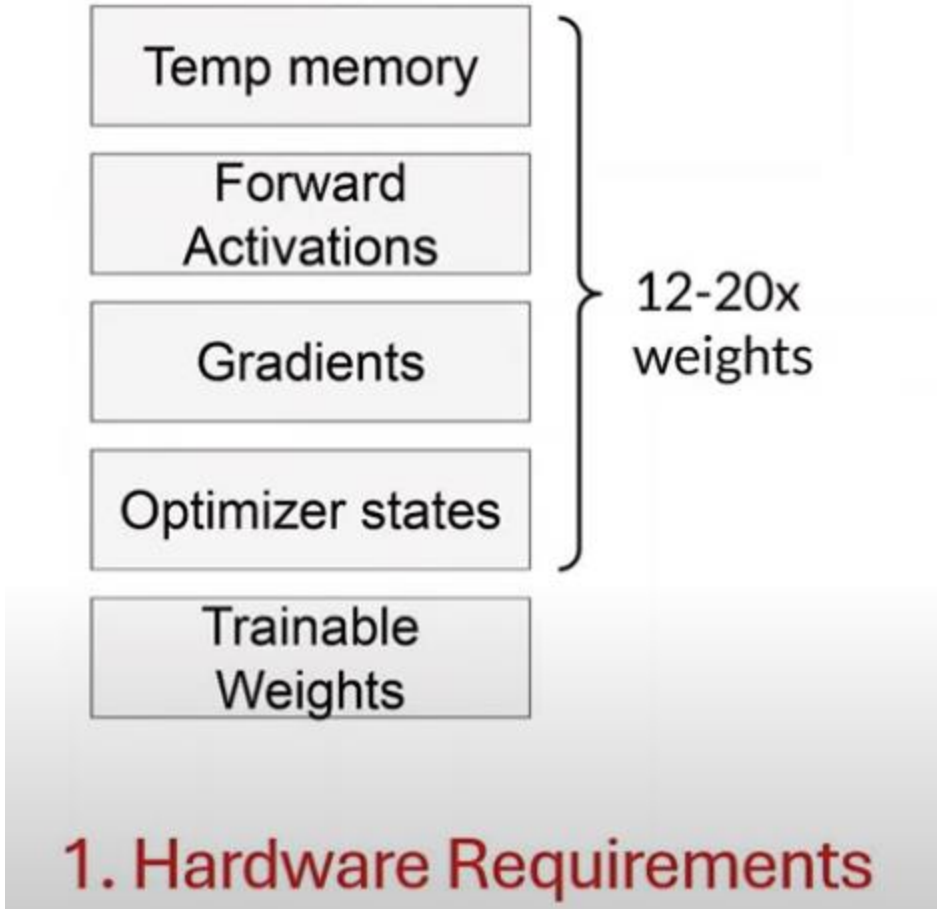
Outline

- 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

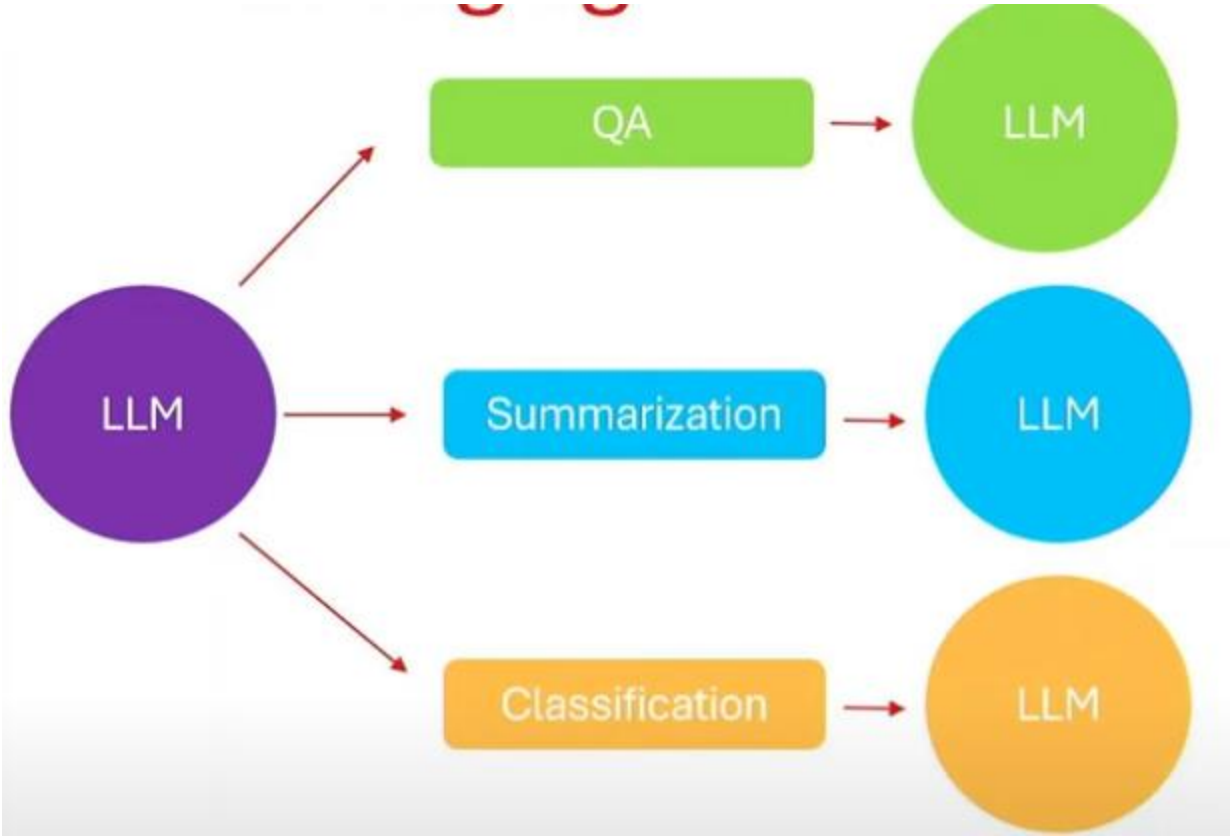
Downsides of In Context Learning

1. **Poor performance**: Prompting generally performs worse than fine-tuning [Brown et al., 2020].
2. **Sensitivity** to the wording of the prompt [Webson & Pavlick, 2022], order of examples [Zhao et al., 2021; Lu et al., 2022], etc.
3. **Lack of clarity** regarding what the model learns from the prompt. Even random labels work [Min et al., 2022]!
4. **Inefficiency:** The prompt needs to be processed *every time* the model makes a prediction.

Why is Full Finetuning in LLM Challenging?

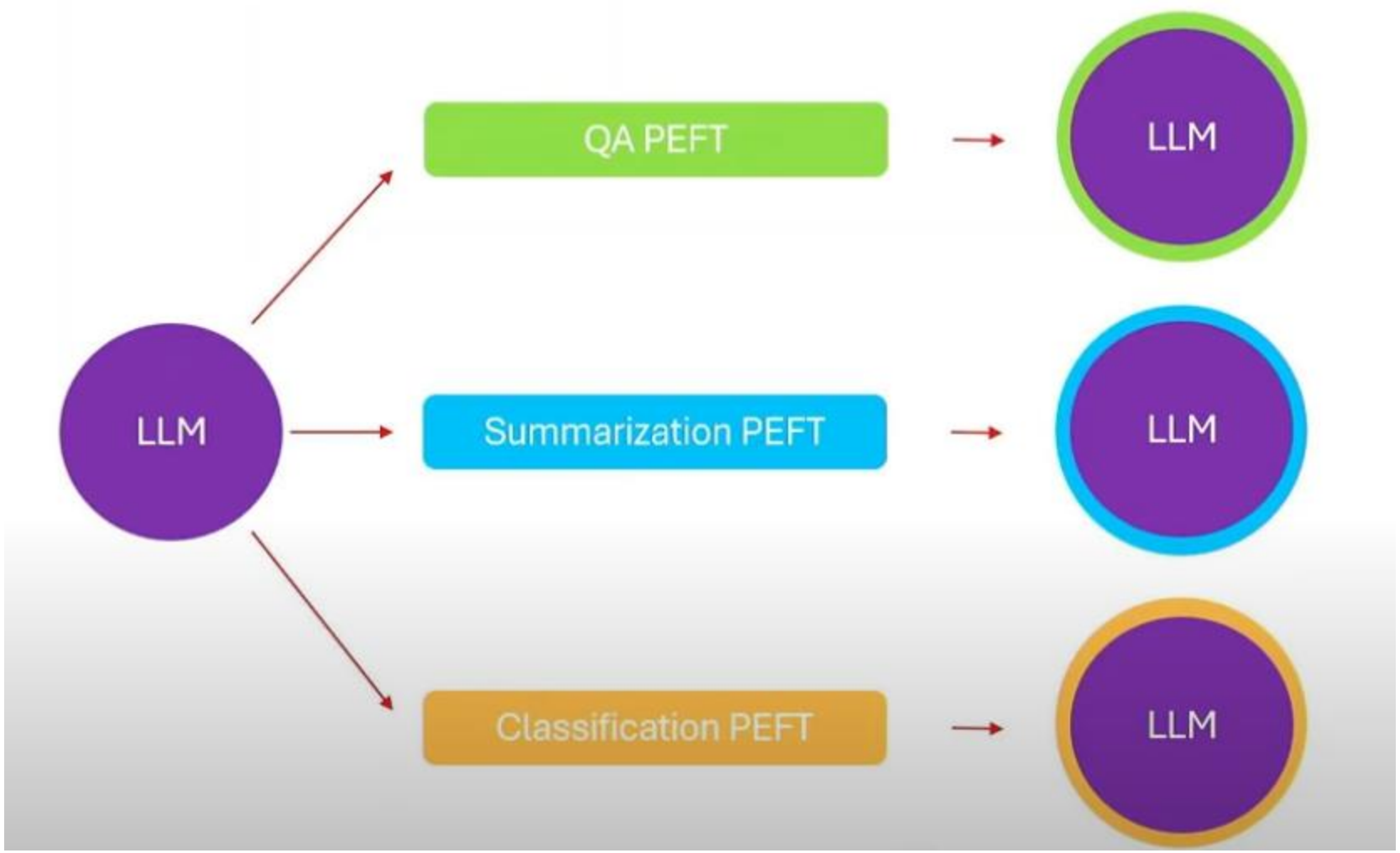


1. Hardware Requirements



2. Storage

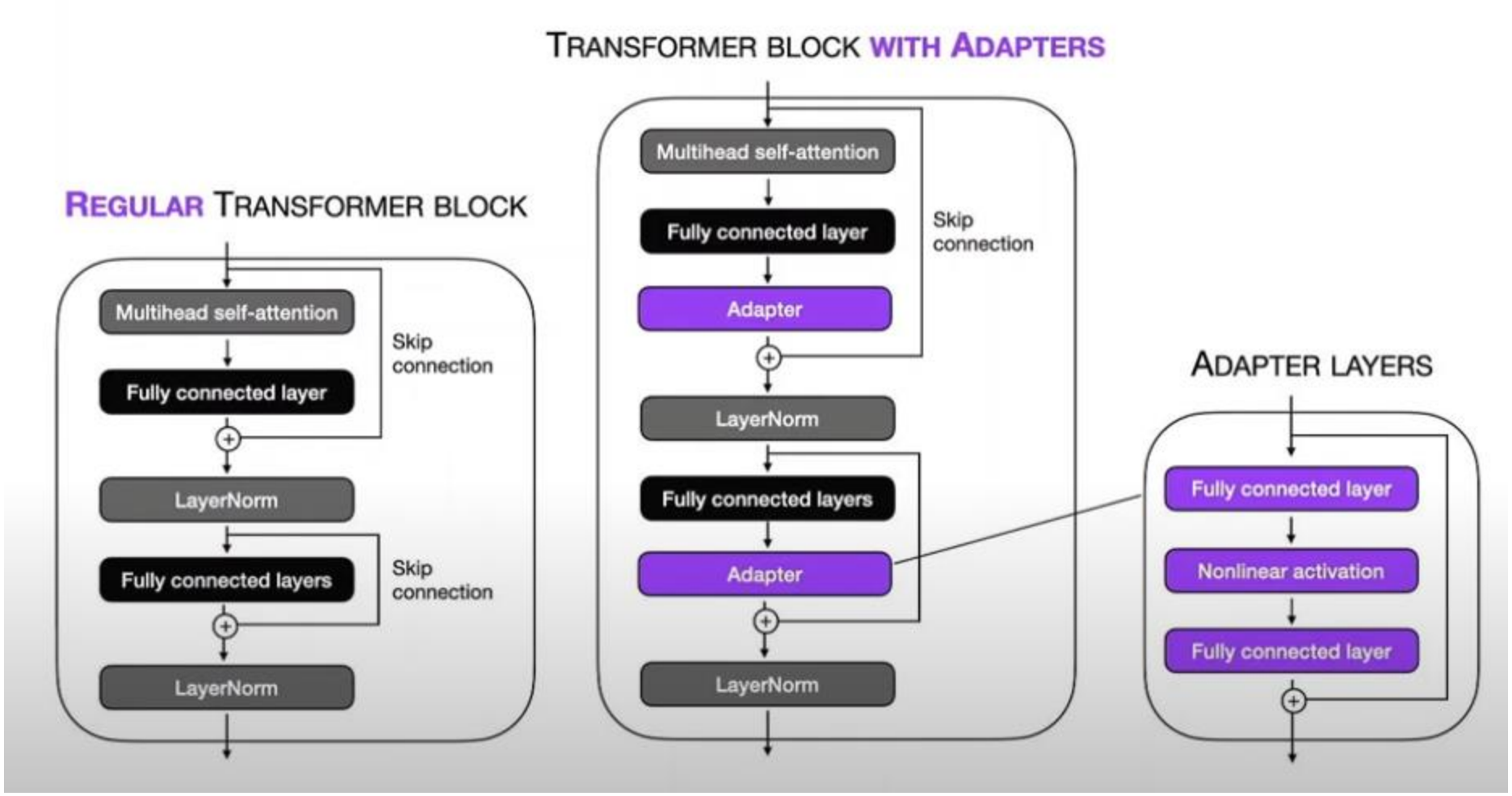
PEFT

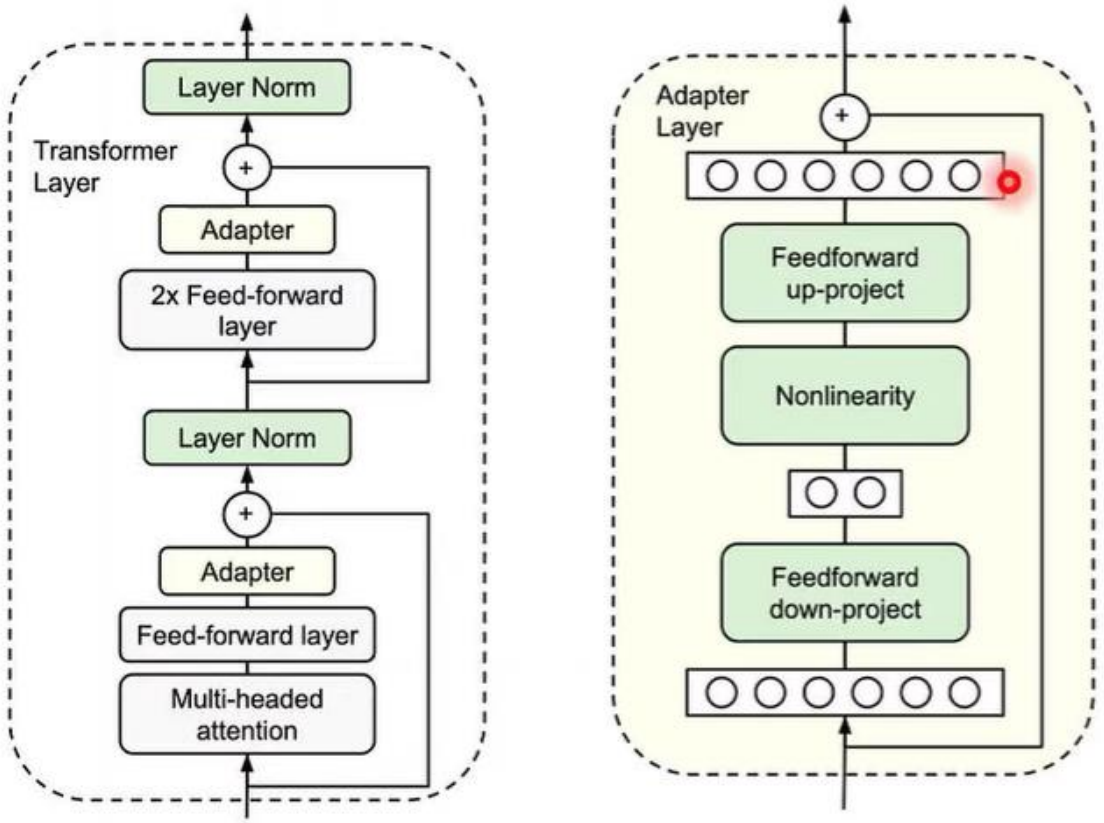

LLM
QA PEFT
Summarization PEFT
Classification PEFT
LLM
LLM
LLM

PEFT Advantages

- Reduced computational costs
  - requires fewer GPUs and GPU time
- Lower hardware requirements
  - works with smaller GPUs & less memory
- Better modelling performance
  - reduces overfitting by preventing catastrophic forgetting
- Less storage
  - majority of weights can be shared across different tasks

Adapters – Houlsby et al. 2019

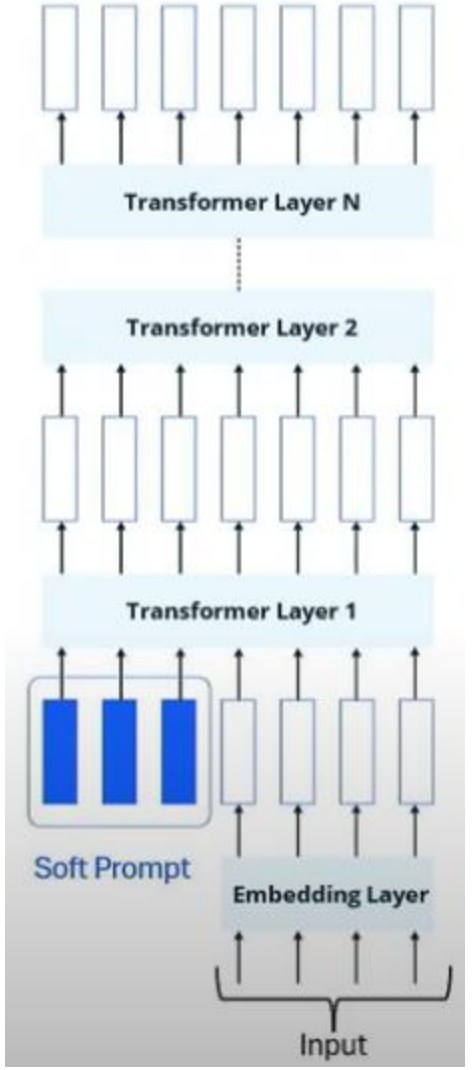




Architecture of adapter module and its integration with transformer [Houlsby et al., 2019]

(Soft) Prompt Tuning (Lester et al. 2021)

- In Context Tuning- hard prompting
- Would it be possible for the model to learn the prompt itself if we have large amount of input/output and a mechansism to measure how good/bad the prompt is?
- Reserve some special tokens in the input called soft prompts which are the only trainable parameters



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**

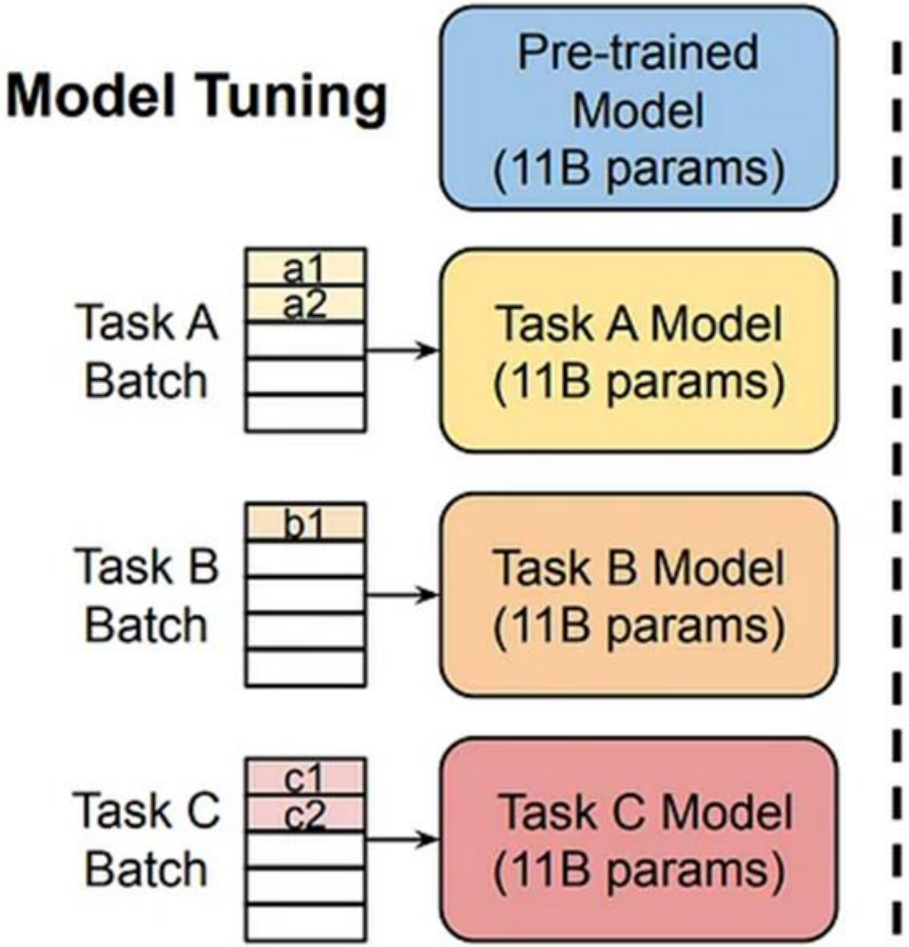
Depending on the technique, there are different methods for how the information is added to the network. The core idea is that the base model does not optimize the text itself but rather the **continuous representation (i.e. some type of learnable tensor) of the prompt text.** This can be some form of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

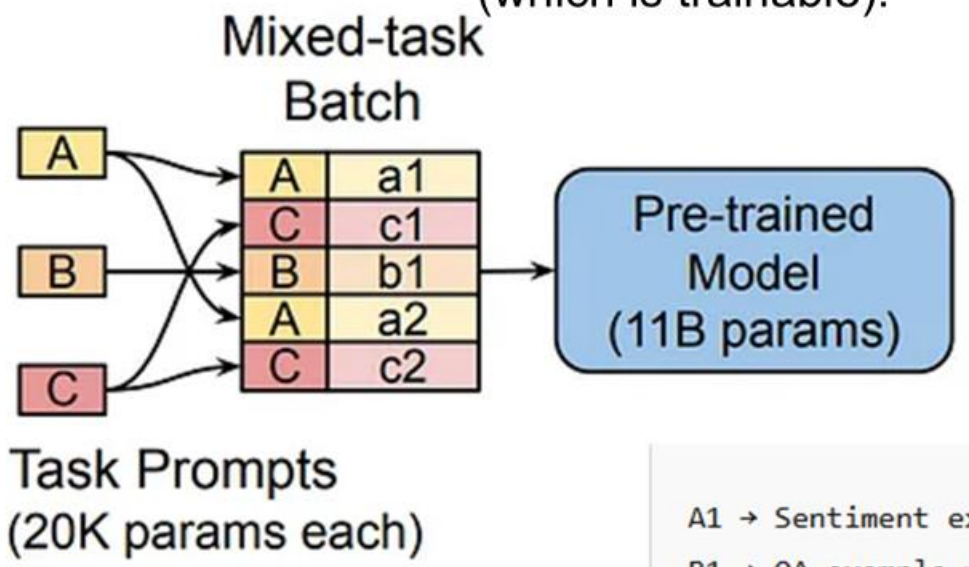
Prompt Tuning:

•**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)
```

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
```

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__")]

Soft Prompt

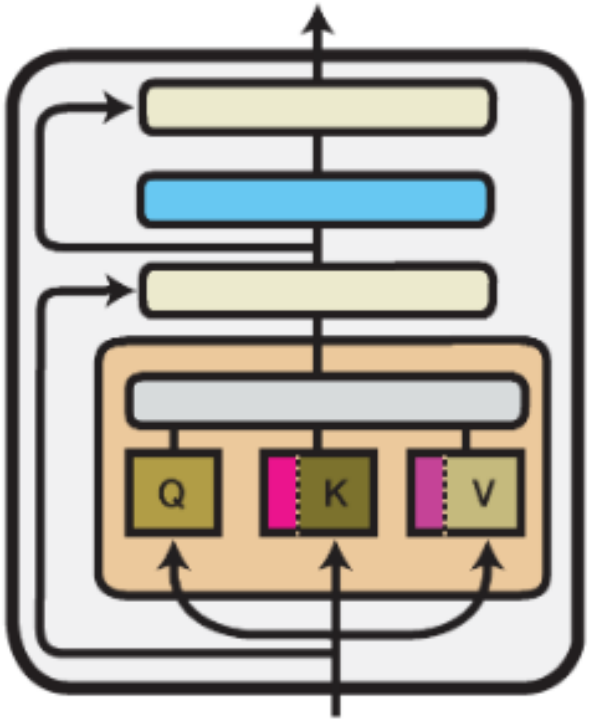
Additive PEFT: Prompt tuning

One-sentence idea:

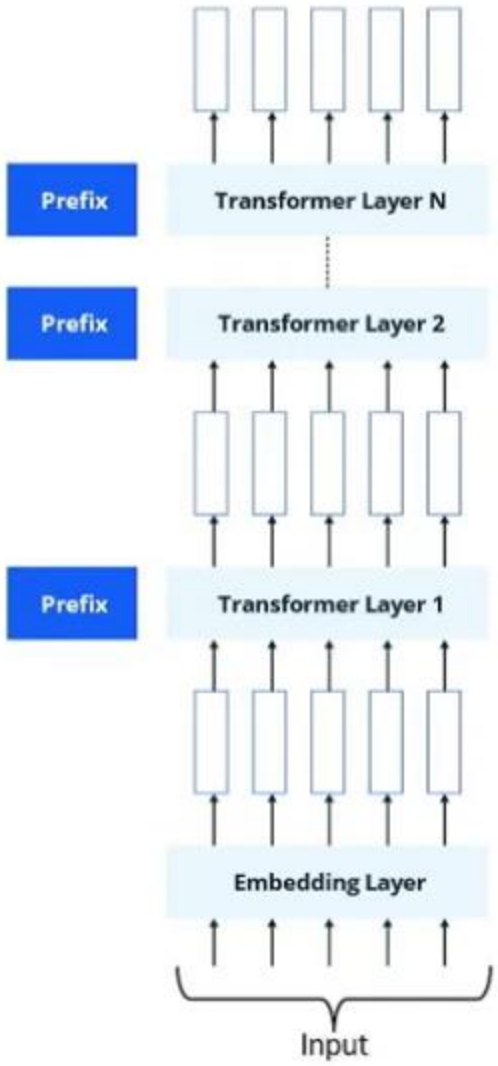
Fine-tune part of your attention input – soft prompt

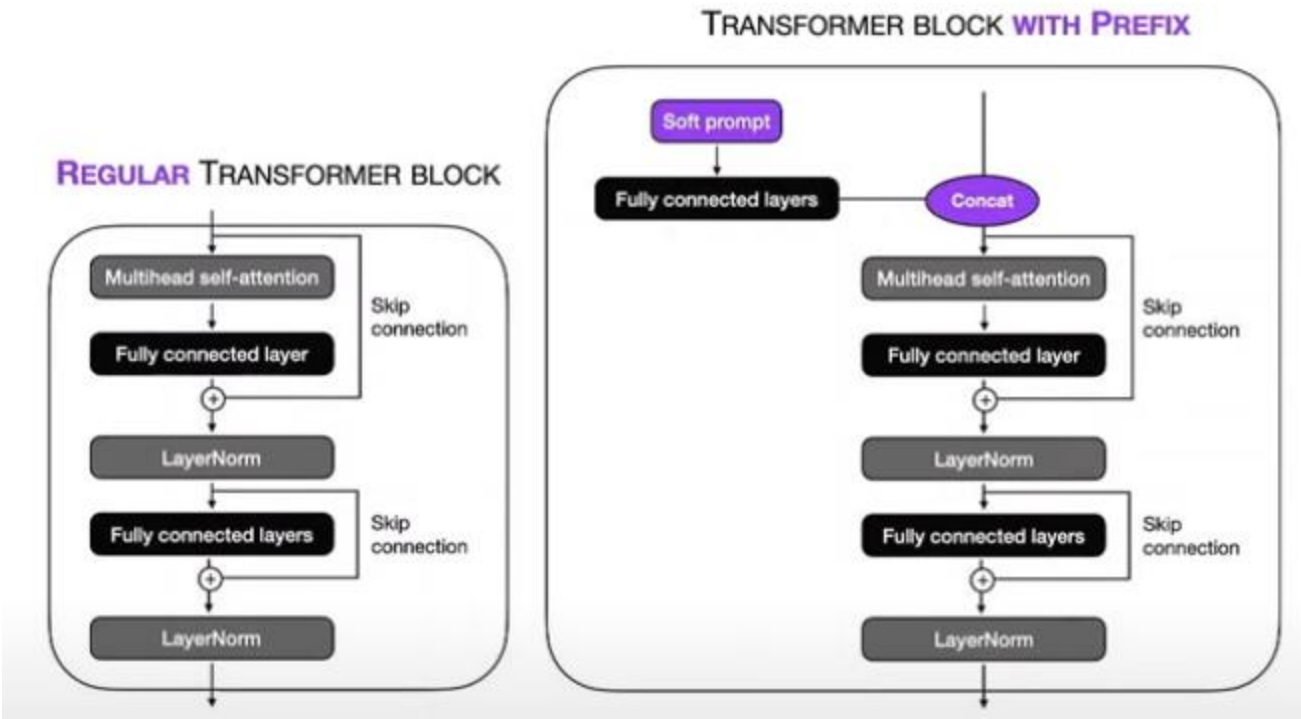
Pseudocode:

```
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
```



Prefix Tuning – Li & Lang 2021





Small set of trainable parameters in every layer of transformer

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]")]

VISHWA VIDYAPEETHAM

```
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.

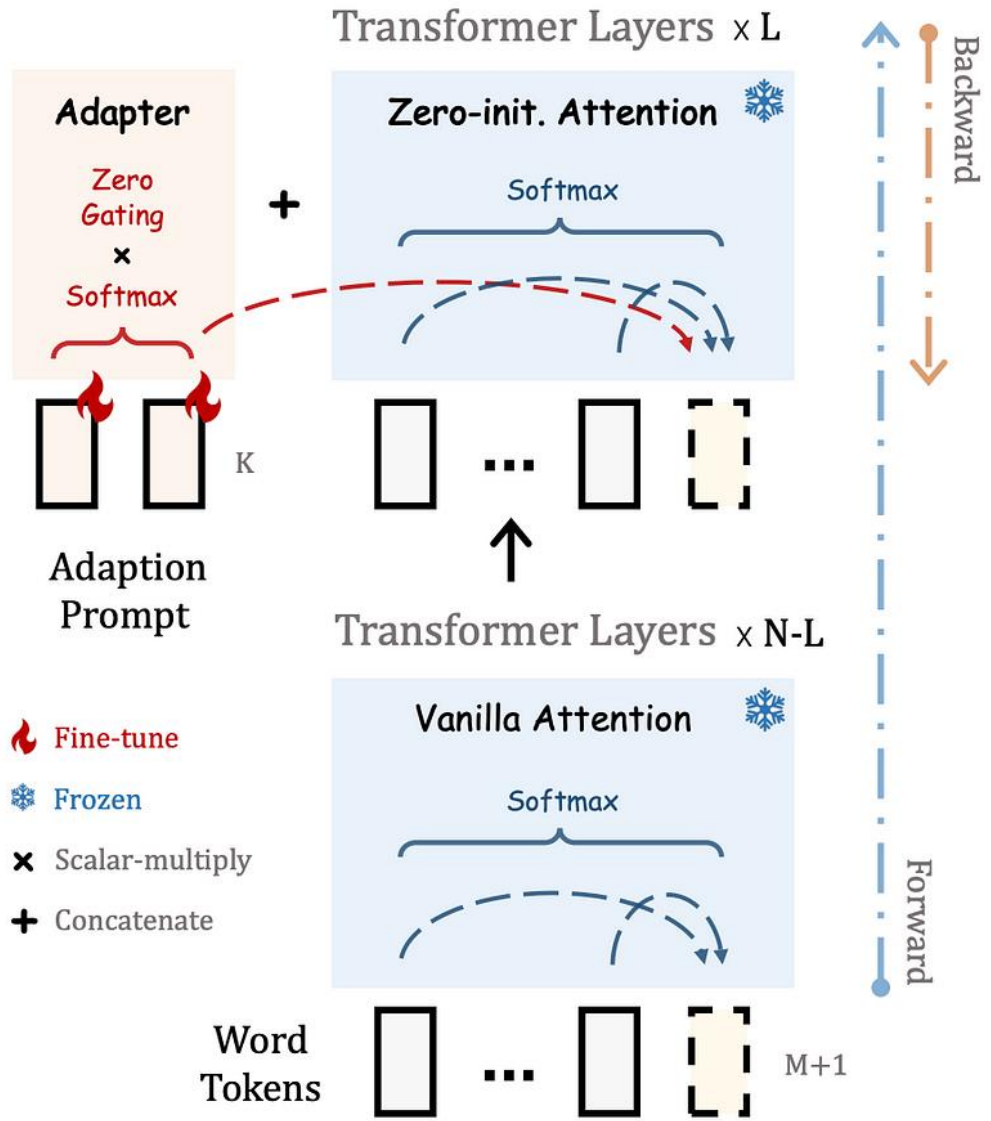
LLaMA adapter

```
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  # ze

    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
```



Outline

- 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

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)
```

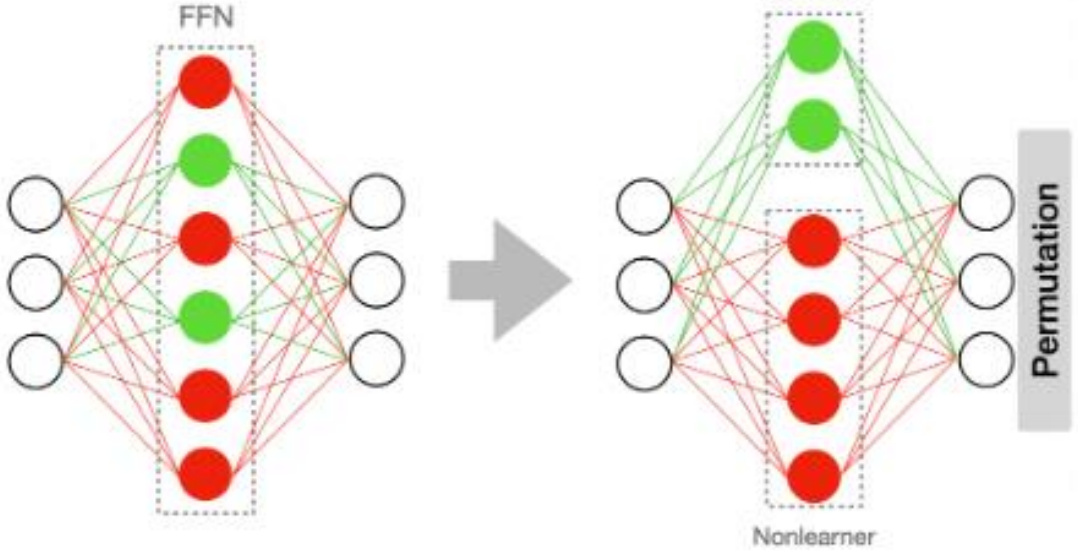
Selective PEFT: Freeze and Reconfigure

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



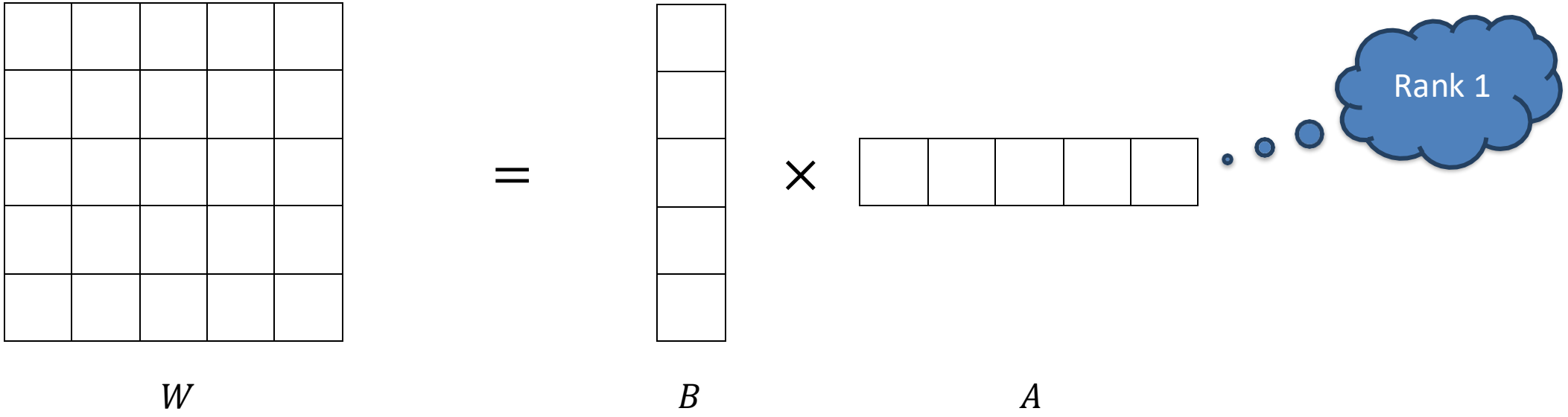
Freezing: This involves keeping certain layers of the pre-trained LLM's architecture unchanged during fine-tuning. These frozen layers typically represent **the LLM's core capabilities for understanding language**.
Reconfiguring: Specific parts of the LLM, particularly the later layers, are reconfigured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

Outline

- 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

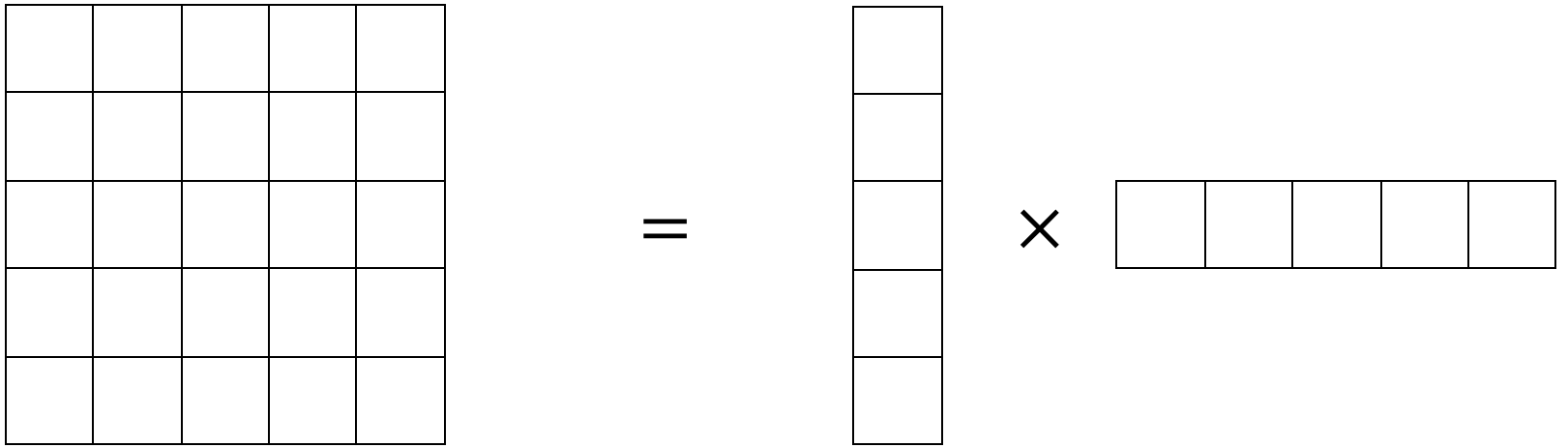
LoRA - Intuition

LoRA revolves around the idea that any matrix $W \in R^{m \times n}$ can be decomposed into $\mathbf{W = BA}$ where $B \in R^{m \times r}$ and $A \in R^{r \times n}$



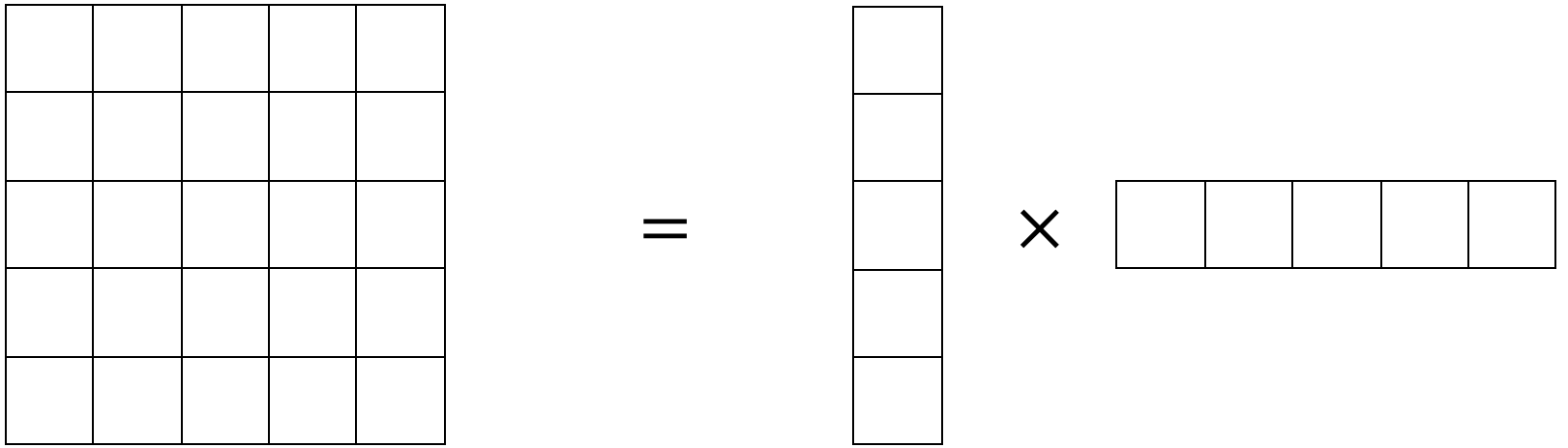
LoRA - Intuition

LoRA revolves around the idea that any matrix $W \in R^{m \times n}$ can be decomposed into $\mathbf{W = BA}$ where $B \in R^{m \times r}$ and $A \in R^{r \times n}$



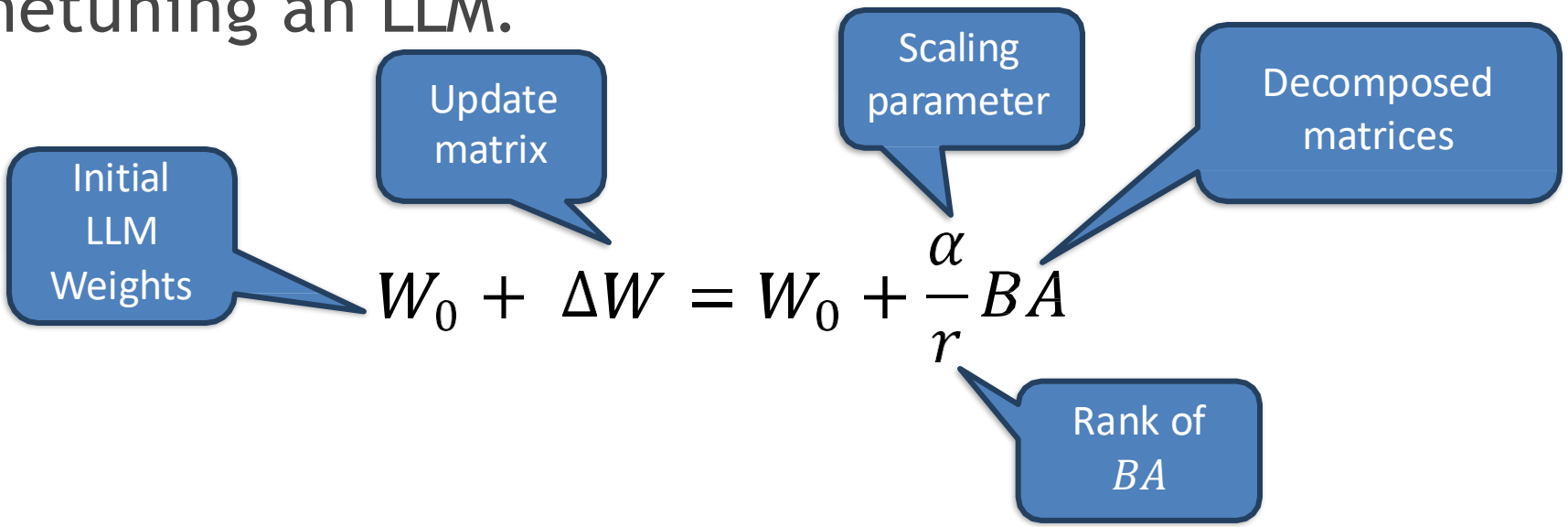
LoRA - Intuition

We can even increase the rank to get better performance.



LoRA - Working

Now, we use the same concept of matrix decomposition while finetuning an LLM.



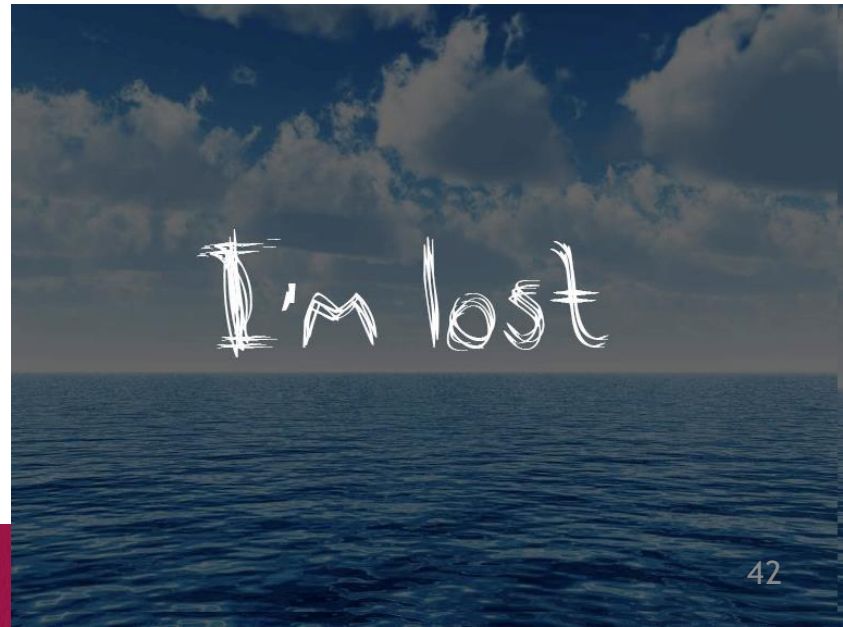
Remember, we are decomposing the update matix (ΔW), and not the original weights W_0.

LoRA - Working

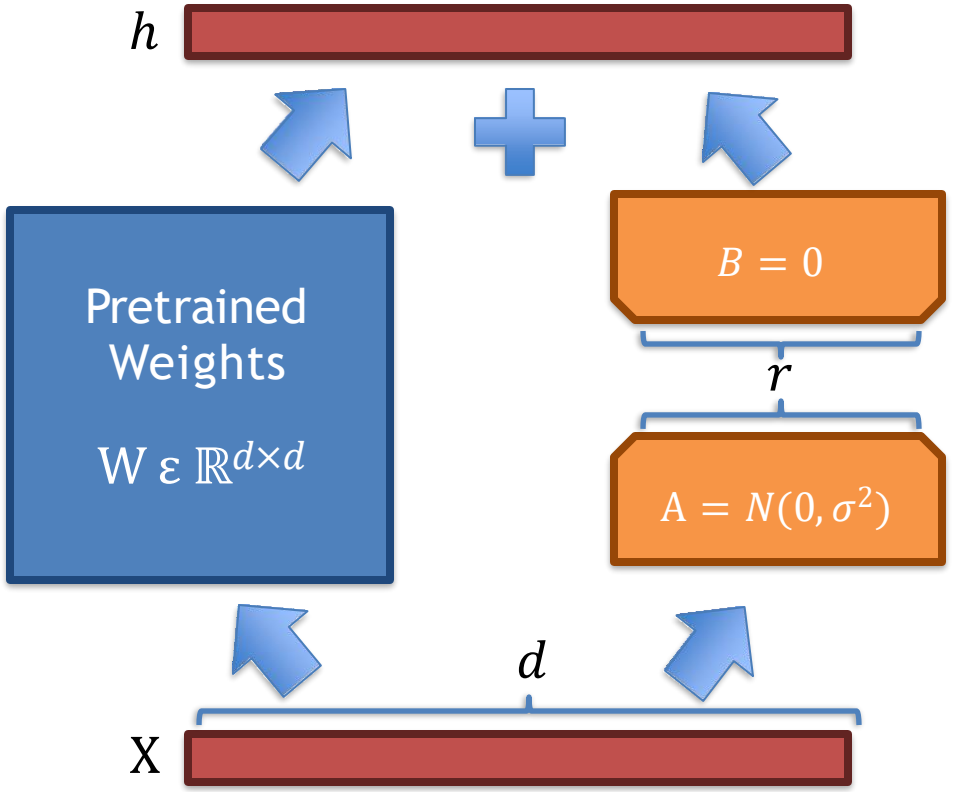
$$W_0 + \Delta W = W_0 + \frac{\alpha}{r} BA$$

We initialize B using a zero matrix, and A using a normal distribution.

Now, let's look at this diagrammatically.

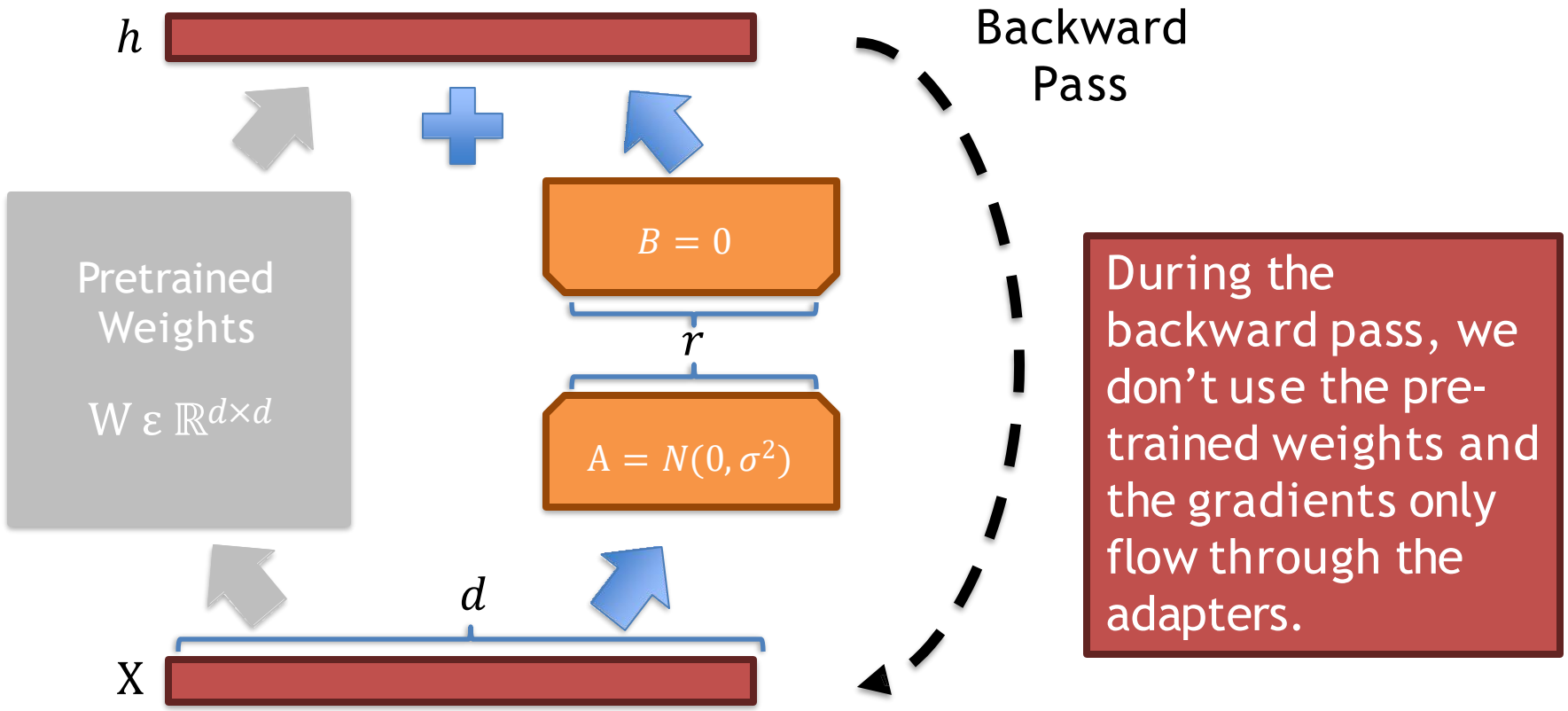


LoRA - Working



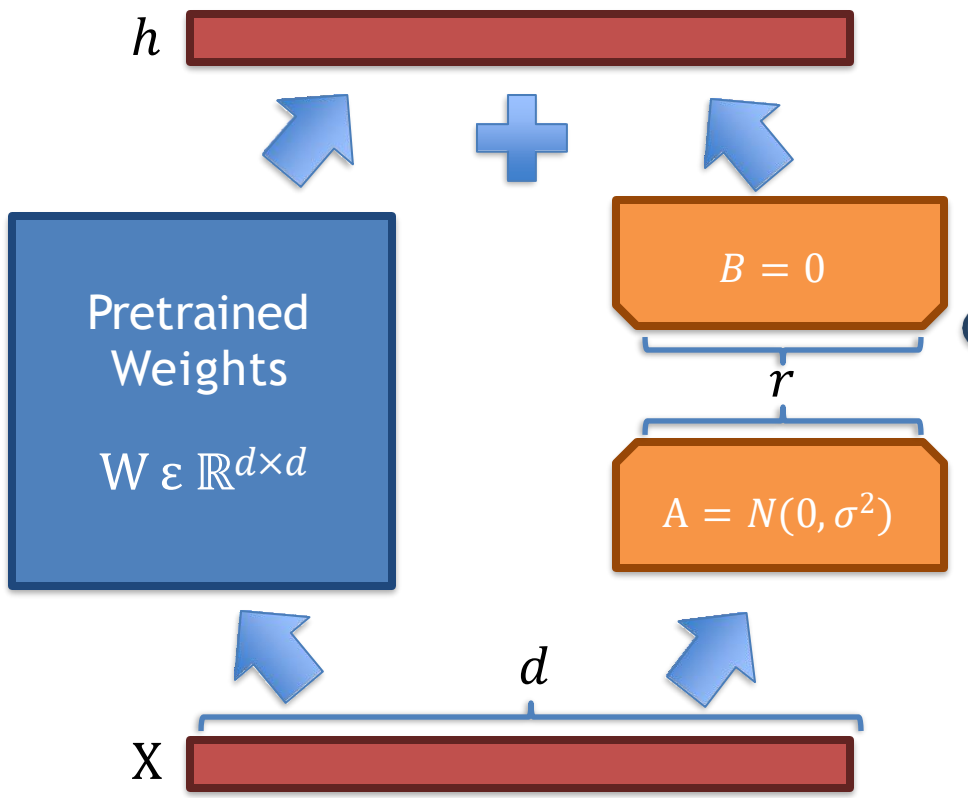
Notice how the reparameterization (LoRA) runs parallel to the original model.

LoRA - Working



Notice how the reparameterization (LoRA) runs parallel to the original model.

LoRA - Working



At inference time, the update matrix can be merged with the original model weights to make the process efficient.

Notice how the reparameterization (LoRA) runs parallel to the original model.

LoRA - Intuition

Let’s explore the scale at which **LoRA** can help reduce the number of parameters needed to achieve comparable performance!

LoRA - Intuition

Number of trainable parameters

Rank	Model 7B	Model 13B	Model 70B	Model 180B
1	167K	228K	529K	849K

LoRA - Intuition

Number of trainable parameters

Rank	Model 7B	Model 13B	Model 70B	Model 180B
1	167K	228K	529K	849K
2	334K	456K	1M	2M

LoRA - Intuition

Number of trainable parameters

Rank	Model 7B	Model 13B	Model 70B	Model 180B
1	167K	228K	529K	849K
2	334K	456K	1M	2M
8	1M	2M	4M	7M

LoRA - Intuition

Number of trainable parameters

Rank	Model 7B	Model 13B	Model 70B	Model 180B
1	167K	228K	529K	849K
2	334K	456K	1M	2M
8	1M	2M	4M	7M
16	3M	4M	8M	14M
512	86M	117M	270M	434M
1024	171M	233M	542M	869M
8192	1.4B	1.8B	4.3B	7B
Full	7B	13B	70B	180B

This is a generalization considering an LLM of one layer. LLMs are made up of multiple layers.

LoRA - Advantages

Compared to full parameter finetuning, LoRA has the following advantages:

1. Much faster
2. Finetuning can be achieved using less GPU memory
3. Cost efficient
4. Less prone to “catastrophic forgetting” since the original model weights are kept the same.

LoRA – Isn't it enough?

Full Parameter Fine Tuning

Optimizer State (FP32)

Base Model (FP16)

10B → 160GB

LoRA – Isn't it enough?

Full Parameter Fine Tuning

Optimizer State (FP32)

Base Model (FP16)

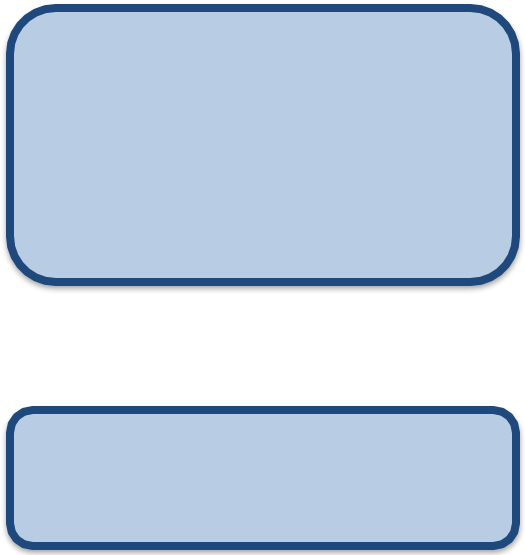
10B → 160GB

LoRA – Isn't it enough?

Full Parameter Fine Tuning

Optimizer State (FP32)

Base Model (FP16)



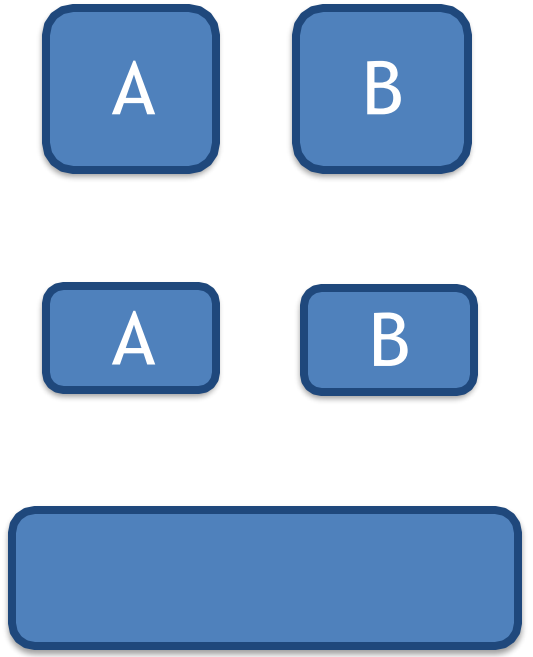
10B → 160GB

LoRA

Optimizer State (FP32)

LoRA Adapter (FP16)

Base Model (FP16)



10B → ~40GB

LoRA – Isn't it enough?

Full Parameter Fine Tuning

Optimizer State (FP32)

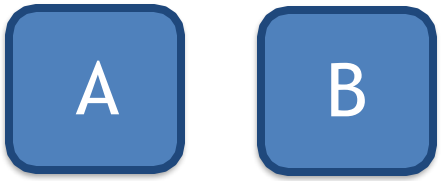
Base Model (FP16)

This will be frozen. So, no optimization, but the parameters still needs to be stored in memory for forward pass

10B → 160GB

LoRA

Optimizer State (FP32)



LoRA Adapter (FP16)

Base Model (FP16)

10B → ~40GB

LoRA – Isn't it enough?

As we can see below, LoRA's performance is comparative to full parameter fine-tuning and, in some cases, even outperforms it.

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.

These metrics are used for performance evaluation.

LoRA - Summary

- LoRA reduces the trainable parameters and memory requirements while maintaining good performance.
- LoRA adds pairs of rank decomposition weight matrices (called update matrices) to each layer of the LLM.
- Only the update matrices, which have significantly fewer parameters than the original model weights, are trained.

- QLoRA

QLoRA

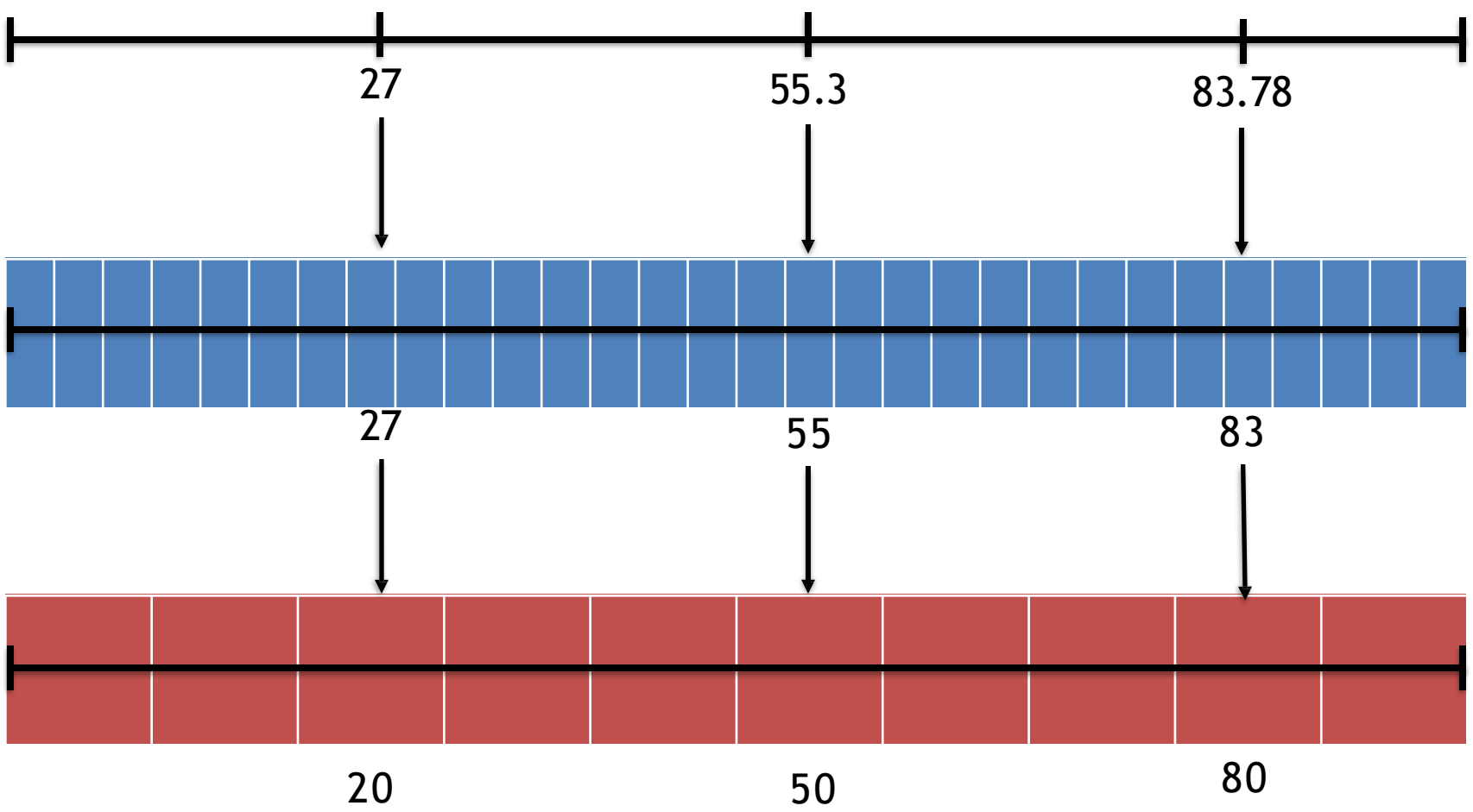
- QLoRA is the extended version of LoRA which works mainly by quantizing the precision of the network parameters.

- Before we dive into what QLoRA is, let's look at what quantization is.

Think of quantization as 'splitting range into buckets '.

QLoRA

Think of quantization as ‘splitting range into buckets ’.



QLoRA

Let's look at an example!

Let X^{FP32} be an array of values.

Here, FP32 refers to a 32-bit floating-point number.

1.5	2.3	3.7	4.1	5.6	6.8	7.9	8.4	9.2	10.2

What if we want to quantize from FP32 to Int8?

QLoRA

So, to quantize X^{FP32} to X^{Int8} :

Int8 values range from -127 to 127

$$X^{Int8} = round\left(\frac{127}{absmax(X^{FP32})} X^{FP32}\right)$$

Array of values

The absolute maximum value in the array, X^{FP32} .

QLoRA

So, to quantize X^{FP32} to X^{Int8} :

$$X^{Int8} = round\left(\frac{127}{absmax(X^{FP32})} X^{FP32}\right)$$

$$\downarrow$$

$$X^{Int8} = round(c^{FP32} X^{FP32})$$

QLoRA

So, to quantize X^{FP32} to X^{Int8} :

$$X^{Int8} = round\left(\frac{127}{absmax(X^{FP32})} X^{FP32}\right)$$

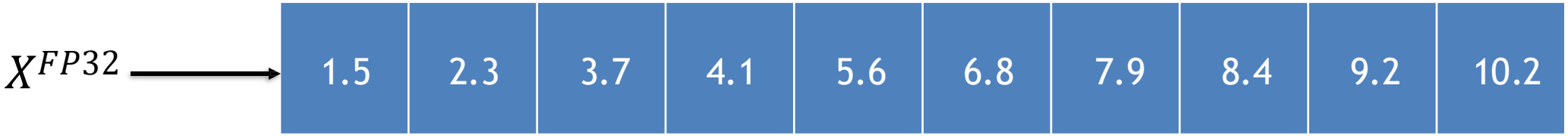
$$\downarrow$$

$$X^{Int8} = round(c^{FP32} X^{FP32})$$

QLoRA

$$X^{Int8} = round(c^{FP32} X^{FP32})$$

In our example,



$$c^{FP32} = \frac{127}{absmax(X^{FP32})} = \frac{127}{10.2} = 12.4509$$

Now, we combine the formula and the values that we have

QLoRA

$$X^{Int8} = round(12.4509 \text{ x } \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 1.5 & 2.3 & 3.7 & 4.1 & 5.6 & 6.8 & 7.9 & 8.4 & 9.2 & 10.2 \\ \hline \end{array})$$

$$X^{Int8} = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 18 & 29 & 46 & 51 & 69 & 85 & 98 & 105 & 115 & 127 \\ \hline \end{array}$$

Voila! That's how we quantize from FP32 to Int8 using the formula:

$$X^{Int8} = round(c^{FP32}X^{FP32})$$

QLoRA

$$X^{Int8} = round(c^{FP32} X^{FP32})$$

What if we want to dequantize and get back the original array, X^{FP32}?

To dequantize:

$$X^{FP32} = \frac{X^{Int8}}{c^{FP32}}$$

Now that we know what quantization is, let's look at how QLoRA works!

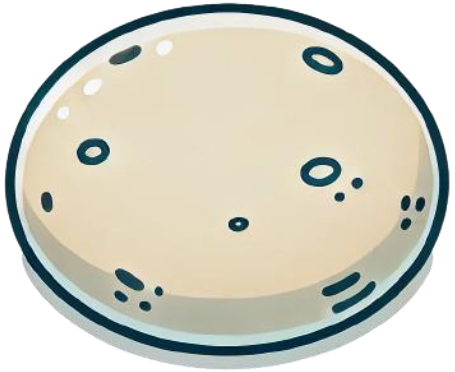
QLoRA – The Pizza

Imagine QLoRA to be a mouthwatering pizza.

Now, to make a pizza, we need to gather a few key ingredients!

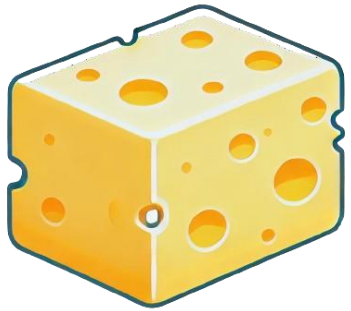
QLoRA – The Ingredients

There are 3 key ingredients which helps us make QLoRA:



4-Bit NormalFloat

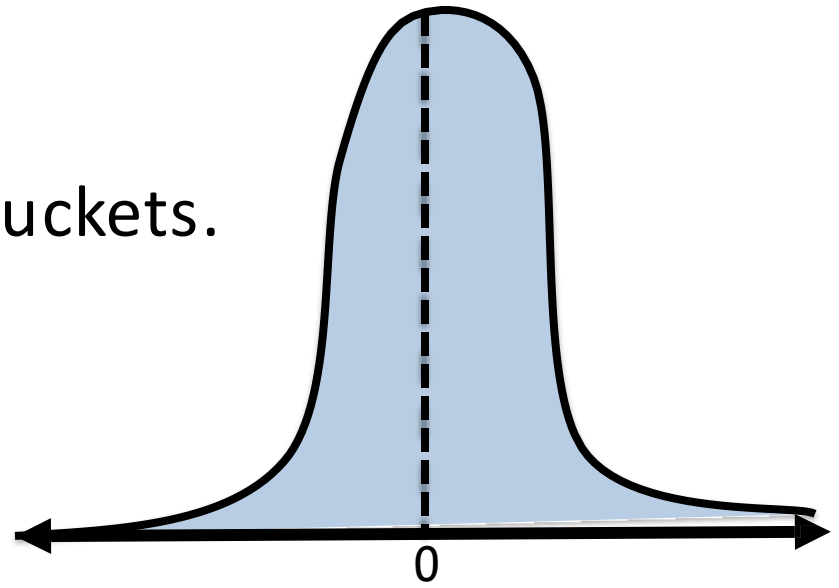
Double Quantization



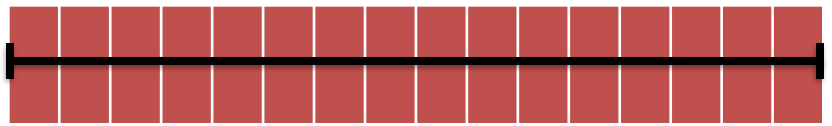
Paged Optimizer

QLoRA – Ingredient 1: 4-Bit NormalFloat

- 4-bit NormalFloat
- 4-bit NormalFloat is a clever way to split the buckets.



4-bit means we have
$2^4 = 16$ possible buckets for quantization.



Equally spaced buckets

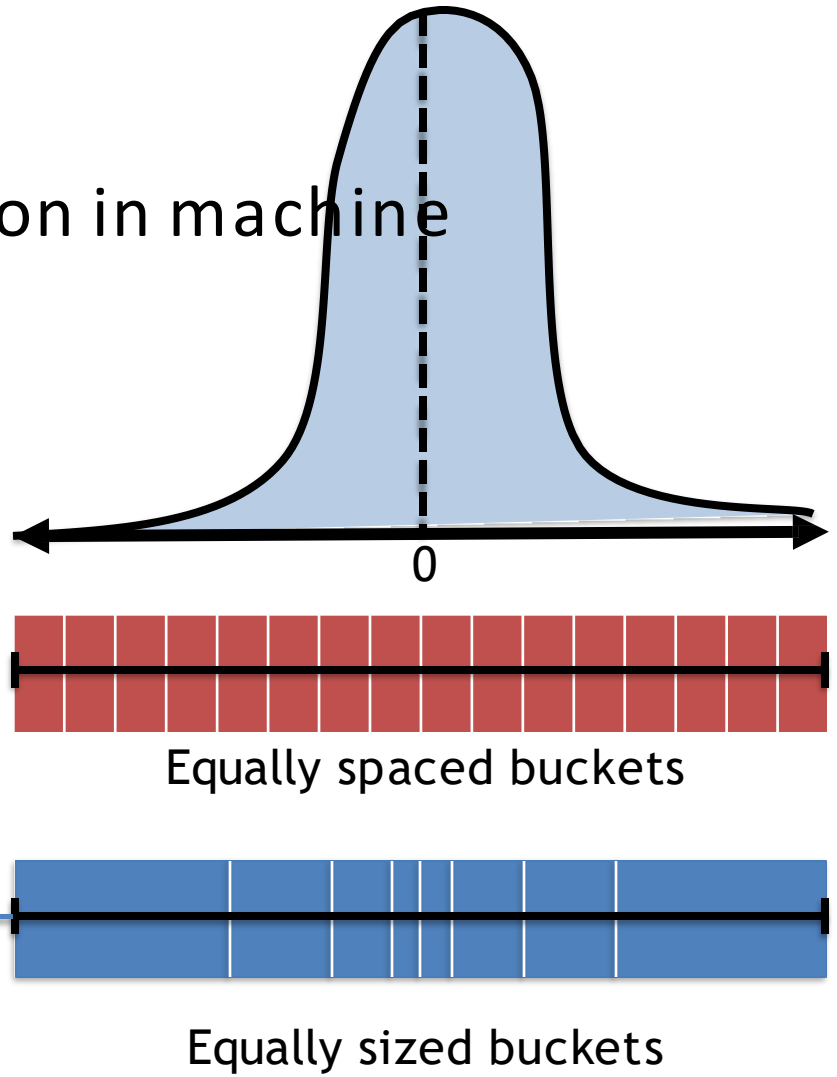
Equally sized buckets

This is an enhanced version of quantile quantization.

QLoRA – Ingredient 1: 4-Bit NormalFloat

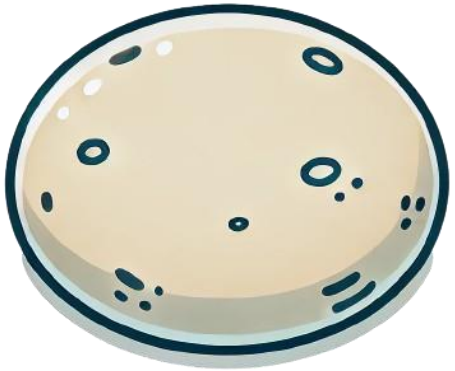
- Why use 4-bit NormalFloat
- Designed for efficient storage and computation in machine learning.

Most datasets in machine learning are normally distributed and precision around the mean is valuable.



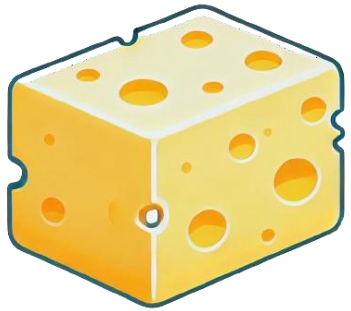
QLoRA – The Ingredients

There are 3 key ingredients which helps us make QLoRA:



4-Bit NormalFloat

Double Quantization



Paged Optimizer

QLoRA – Ingredient 2: Double Quantization

Remember this formula?

$$X^{Int8} = round(c^{FP32} X^{FP32})$$

Which is not an issue, as it's just 1constant. Right?

Now, if we think about this in terms of neural networks….

QLoRA – Ingredient 2: Double Quantization

Now, if we think about this in terms of neural networks….

Let’s take a 5x5 matrix to be the weights in a neural network:

Weight Tensor

-0.7	-0.3	0.0	-0.4	0.3
-1.0	0.2	0.7	1.7	-0.9
-0.1	-1.5	-0.1	0.8	0.5
1.2	-1.7	-0.9	-0.3	0.7
0.4	0.1	-1.4	2.2	-1.1

QLoRA – Ingredient 2: Double Quantization

Now, if we think about this in terms of neural networks….

Weight Tensor

-0.7	-0.3	0.0	-0.4	0.3
-1.0	0.2	0.7	1.7	-0.9
-0.1	-1.5	-0.1	0.8	0.5
1.2	-1.7	-0.9	-0.3	0.7
0.4	0.1	-1.4	2.2	-1.1

Rescale all parameters

$using\ c$

Rescaled Weight Tensor

-4	-2	0	-22	16
-60	10	40	99	-57
-5	-88	-9	48	27
72	-100	-50	-18	40
22	8	-81	127	-66

If we bring back the formula: $round(W^{FP32}c^{FP32}) = W^{Int8}$

QLoRA – Ingredient 2: Double Quantization

Now, if we think about this in terms of neural networks….

Weight Tensor

-0.7	-0.3	0.0	-0.4	0.3
-1.0	0.2	0.7	1.7	-0
-0.1	-1.5	-0.1	0.8	0
1.2	-1.7	-0.9	-0.3	0
0.4	0.1	-1.4	2.2	-1.1

Rescaled Weight Tensor

-4	-2	0	-22	16
		40	99	-57
		-9	48	27
		50	-18	40
22	8	-81	127	-66

We quantize to Int8 for simplicity but when we implement QLoRA we use **4-bit Normal Float.**

If we bring back the formula: $round(W^{FP32} c^{FP32}) = W^{Int8}$

QLoRA – Ingredient 2: Double Quantization

Now, if we think about this in terms of neural networks….

Weight Tensor

-0.7	-0.3	0.0	-0.4	0.3
-1.0	0.2	0.7	1.7	-0.9
-0.1	-1.5	-0.1	0.8	0.5
1.2	-1.7	-0.9	-0.3	0.7
0.4	0.1	-1.4	2.2	-1.1

Rescale all parameters

$using\ c$

Rescaled Weight Tensor

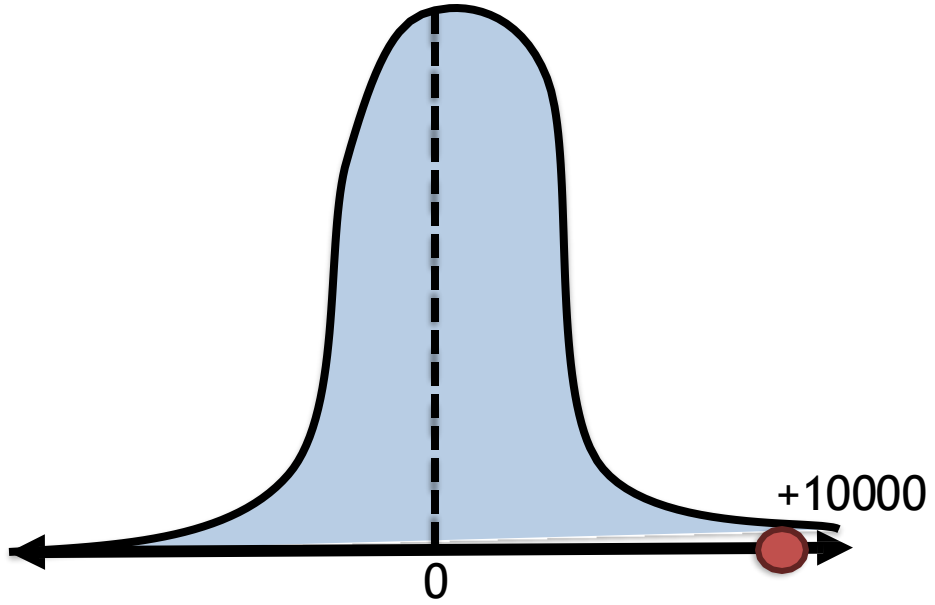
-4	-2	0	-22	16
-60	10	40	99	-57
-5	-88	-9	48	27
72	-100	-50	-18	40
22	8	-81	127	-66

Do you see a problem here?

If we bring back the formula: $round(W^{FP32}c^{FP32}) = W^{Int8}$

QLoRA – Ingredient 2: Double Quantization

Let's see how the weight tensors look like on the graph.

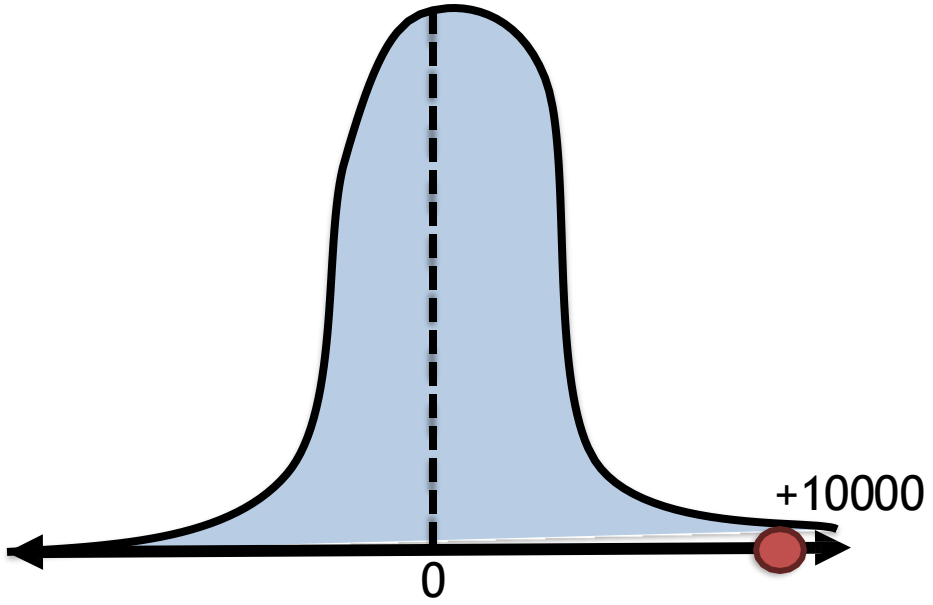


This is unbounded and could take up any maximum value (an outlier!).

$$W^{Int8} = round(\frac{127}{absmax(W^{FP32})} W^{FP32})$$

QLoRA – Ingredient 2: Double Quantization

Let's see how the weight tensors look like on the graph.



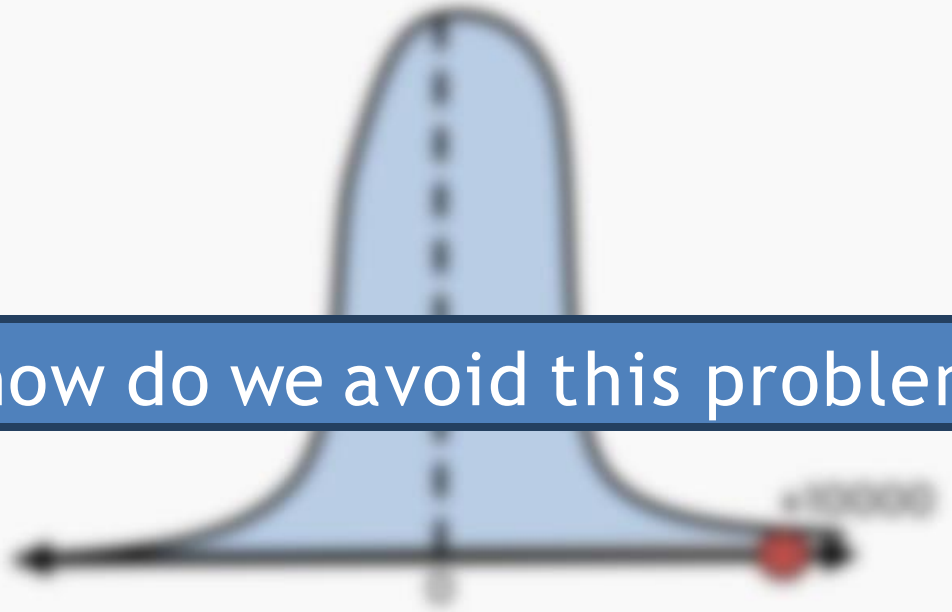
This is unbounded and could take up any maximum value (an outlier!).

This could introduce **bias** in our quantization process

$$W^{Int8} = round\left(\frac{127}{absmax(W^{FP32})} W^{FP32}\right)$$

QLoRA – Ingredient 2: Double Quantization

Let's see how the weight tensors look like on the graph.



So, how do we avoid this problem?

This is unbounded and could take up any maximum valu

This could introduce **bias** in our quantization process

$$W^{INT8} = round\left(\frac{127}{absmax(W^{FP32})} W^{FP32}\right)$$

QLoRA – Ingredient 2 : Double Quantization

The answer to that is: Block-wise Quantization, which is the first step in Double Quantization!

Let's look at an example to understand this concept.

We take the weight tensor that we saw in the previous slides.

Weight Tensor (W^{FP32})

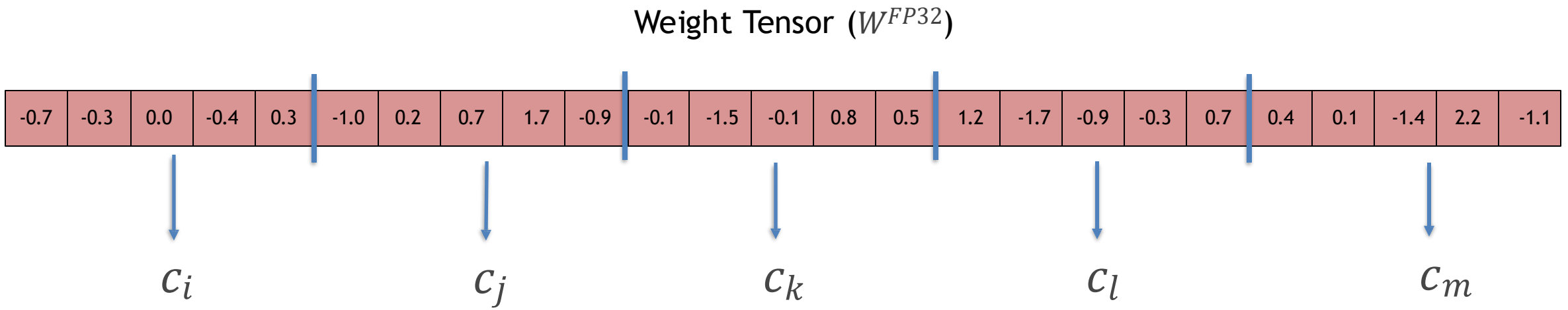
-0.7	-0.3	0.0	-0.4	0.3
-1.0	0.2	0.7	1.7	-0.9
-0.1	-1.5	-0.1	0.8	0.5
1.2	-1.7	-0.9	-0.3	0.7
0.4	0.1	-1.4	2.2	-1.1

QLoRA – Ingredient 2 : Double Quantization

We flatten the matrix as follows:

Now we divide it up into different blocks.

We calculate the quantization constants for each block.



If there are any outliers in a block, they won't affect the quantisation in the other blocks.

QLoRA – Ingredient 2 : Double Quantization

$$C^{Int8} = round(\frac{127}{absmax(C^{FP32})} C^{FP32})$$

Weight Tensor (W^{FP32})

-0.7	-0.3	0.0	-0.4	0.3	-1.0	0.2	0.7	1.7	-0.9	-0.1	-1.5	-0.1	0.8	0.5	1.2	-1.7	-0.9	-0.3	0.7	0.4	0.1	-1.4	2.2	-1.1

c_i	c_j	c_k	c_l	c_m
181.4	74.7	84.7	74.7	57.7

We now rescale all the parameters per block.

-127	-54	0	-73	54	-77	13	51	127	-74	-9	-127	-12	70	40	93	-127	-64	-24	52	22	8	-81	127	-66

Rescaled Weight Tensor (W^{Int8})

QLoRA – Ingredient 2 : Double Quantization

We now have a new array:

c_1^{FP32} → | c_i | c_j | c_k | c_l | c_m |

c_1^{FP32} is an array of all the constants from each block of the Weight Tensor.

Now, we repeat the same process of quantization for the quantization constants.

$$c_1^{Int8} = round\left(\frac{127}{absmax(c_1^{FP32})} c_1^{FP32}\right)$$

$$c_1^{Int8} = round(c_2^{FP32} c_1^{FP32})$$

Double Quantization

QLoRA – Ingredient 2 : Double Quantization

$$c_1^{Int8} = round(c_2^{FP32} c_1^{FP32})$$

Let's see the difference in memory usage before and after Double Quantization.

PROTOPAS

QLoRA – Ingredient 2 : Double Quantization

Before

All we had was a weight matrix containing FP32 values.

In our example, we had a 5x5 matrix.

Each value was 4 bytes in size.

So, the total memory used was: 25x4=100 bytes

-0.7	-0.3	0.0	-0.4	0.3
-1.0	0.2	0.7	1.7	-0.9
-0.1	-1.5	-0.1	0.8	0.5
1.2	-1.7	-0.9	-0.3	0.7
0.4	0.1	-1.4	2.2	-1.1

Weight Tensor (W^{FP32})

Next, let's look at the memory usage **after** Double Quantization.

QLoRA – Ingredient 2 : Double Quantization

Before 25x4=100 bytes

After

-127	-54	0	-73	54
-77	13	51	127	-74
-9	-127	-12	70	40
93	-127	-64	-24	52
22	8	-81	127	-66

Rescaled Weight Tensor
(W^{Int8})

25x1=25 bytes.

c_i	c_j	c_k	c_l	c_m

c_1^{Int8}

5x1=5bytes.

c_2^{FP32}

4 bytes

So, in total:

25 + 5 + 4 = **34 bytes**

QLoRA – Ingredient 2 : Double Quantization

After

-127	-54	0	-73	54
-77	13	51	127	-74
-9	-127	-12	70	40
93	-127	-64	-24	52
22	8	-81	127	-66

c_i	c_j	c_k	c_l	c_m

$$c_2^{FP32}$$

25 + 5 + 4 = 34 bytes

Before

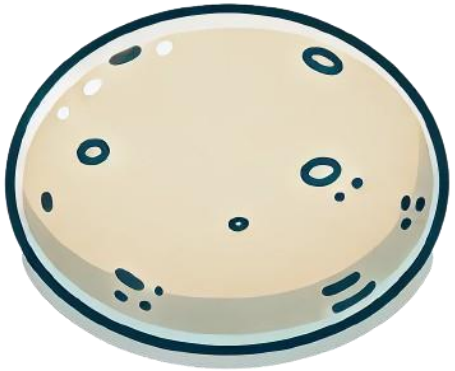
-0.7	-0.3	0.0	-0.4	0.3
-1.0	0.2	0.7	1.7	-0.9
-0.1	-1.5	-0.1	0.8	0.5
1.2	-1.7	-0.9	-0.3	0.7
0.4	0.1	-1.4	2.2	-1.1

25x4=100 bytes

That is an approximate 70% reduction in memory usage!!

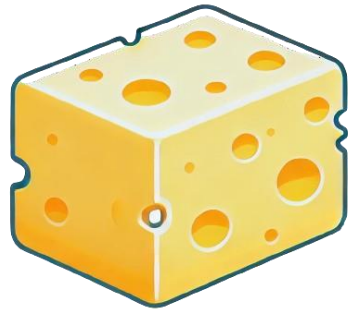
QLoRA – The Ingredients

There are 3 key ingredients which helps us make QLoRA:



4-Bit NormalFloat

Double Quantization



Paged Optimizer

QLoRA – Ingredient 3

Before we talk about the third ingredient in QLoRA, let's talk about a problem.

A problem which all of us have faced while training a Neural Network

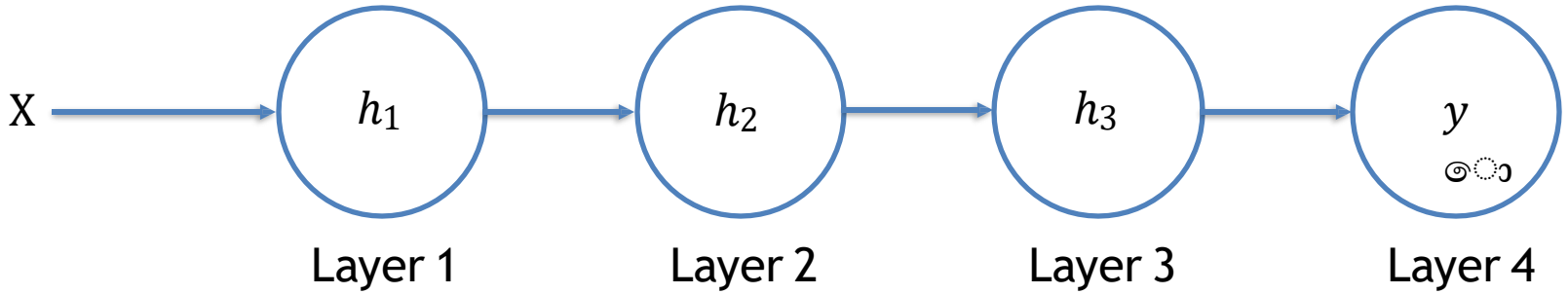
Running Out of Memory!

So, how do we train a modern Neural Networks without taking a hit on the memory?

We use gradient checkpointing.

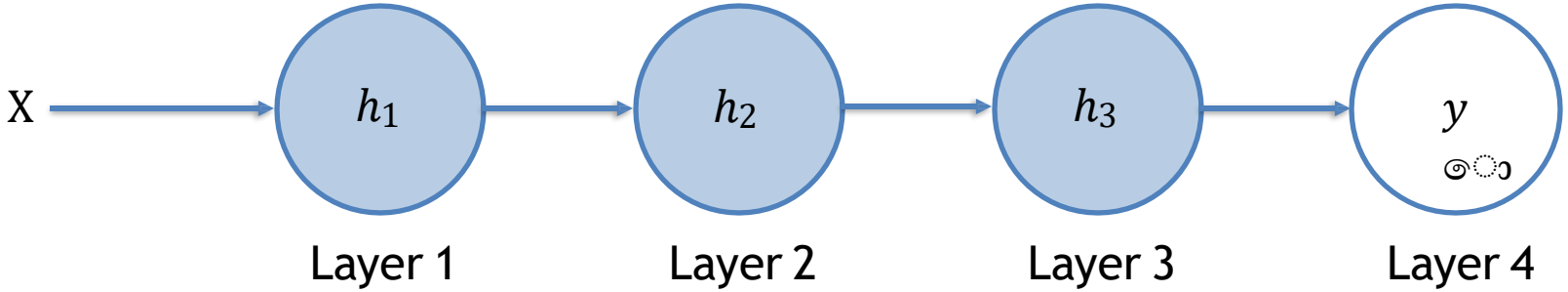
QLoRA – Ingredient 3

Imagine this simple neural network



When we do a forward-pass, we calculate the activations for each layer.

QLoRA – Ingredient 3



However, this takes up precious memory.

Modern-day computers have become very efficient at parallel processing. What they lack is memory.

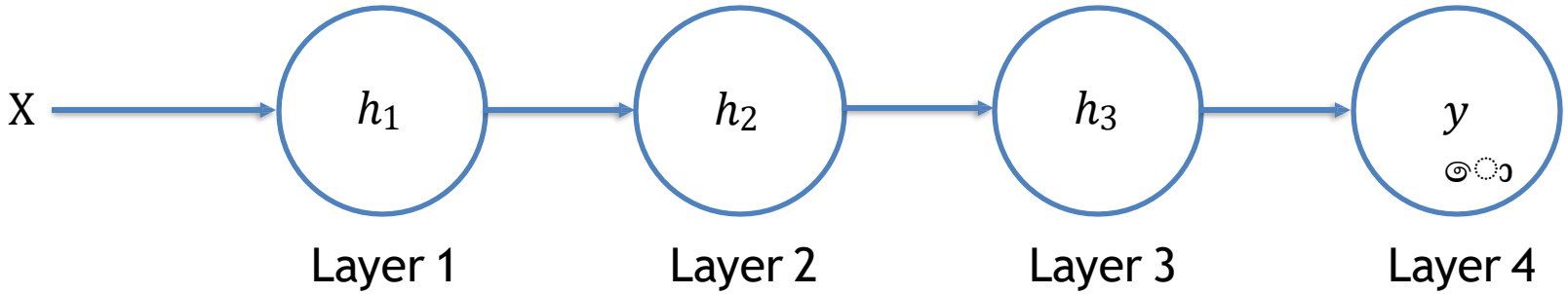
We don't need to store all the hidden states.

QLoRA – Ingredient 3

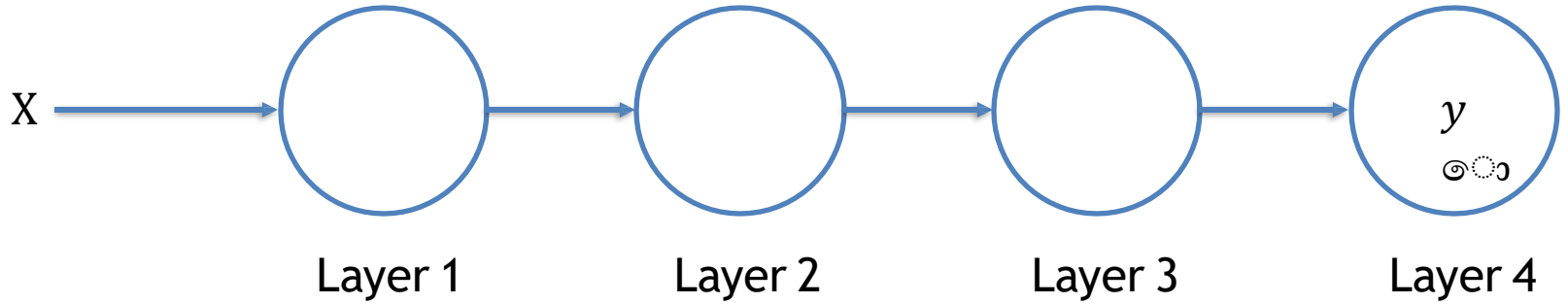
We only store in memory what is needed at the moment.

We keep discarding activations that have already been used to calculate the next dependent hidden state's activation.

So, let's see how it looks!



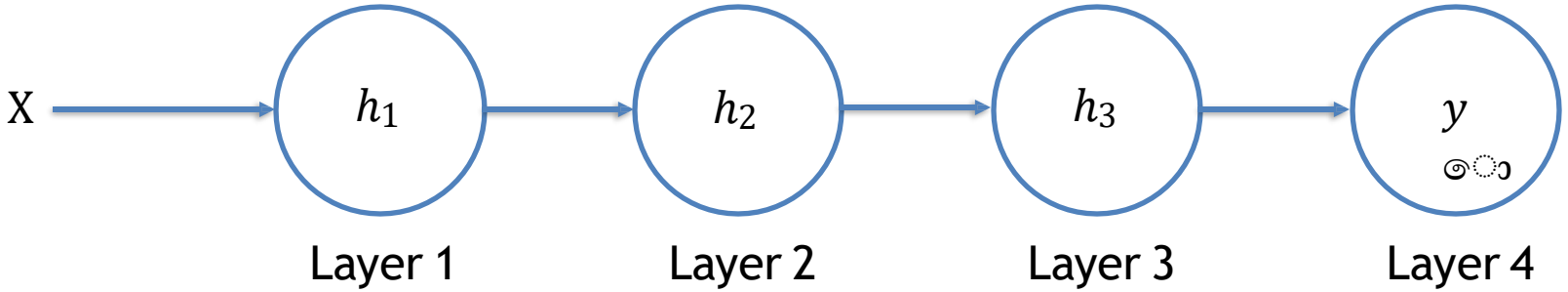
QLoRA – Ingredient 3



During backpropagation, we must recompute all the discarded activations.

To manage this, we introduce checkpoints in the middle.

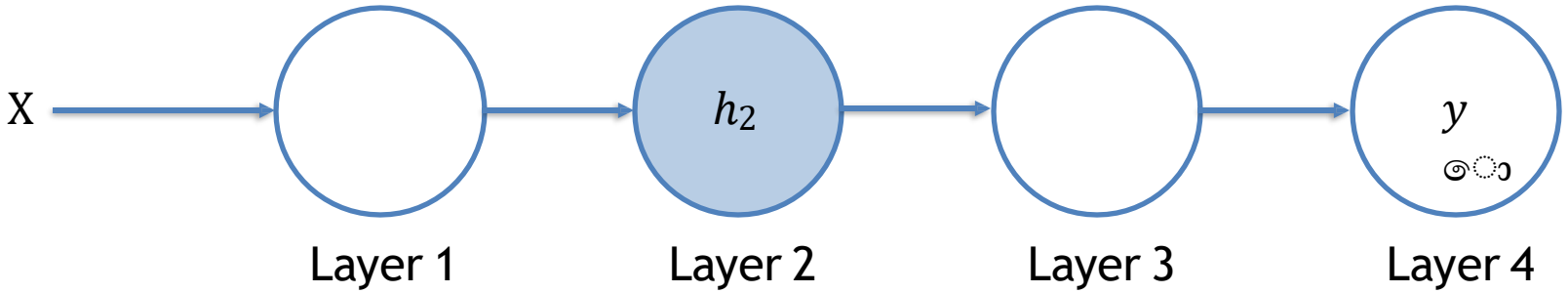
QLoRA – Ingredient 3



Checkpoints are usually placed at every $\sqrt{n}$ layer, considering we have a n-layer neural network.

So, now when we re-compute the activations for backward pass, we don't have to start from the beginning!

QLoRA – Ingredient 3



This allows us to mitigate the OOM (Out of memory) error to some extent, but it doesn't get rid of it!

We still see some memory spikes especially when we pass in long sequences in the batch.

QLoRA - Ingredient 3

This is where our third ingredient comes in!

This allows us to mitigate the (dreaded CUDA out of memory) error to some extent, but it doesn't get rid of it!

We still see some memory spikes especially when we pass in long sequences in the batch.

QLoRA – Ingredient 3 : Paged Optimizer

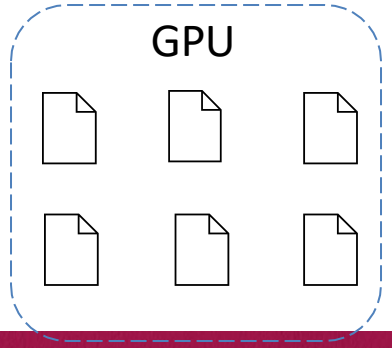
Paged Optimizer - Looping in your CPU

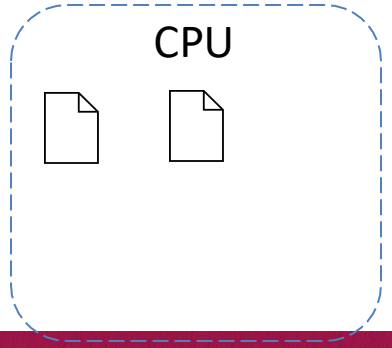
Paging is a memory management technique, where RAM is divided into fixed-size blocks called ‘pages’

It does automatic page-to-page transfers between CPU and GPU

Avoids the gradient checkpointing memory spikes that occur when processing a mini batch with a long sequence length.

GPU Memory has space now.

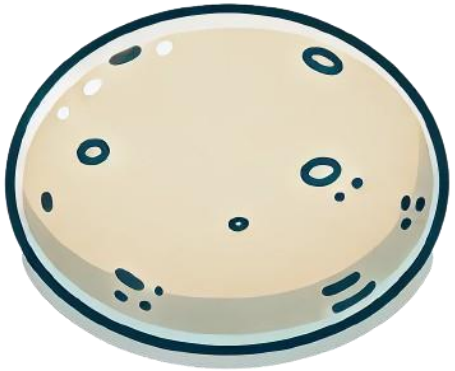




Now that the GPU has space, when a page moved to CPU is required, we move it back to GPU for computation.

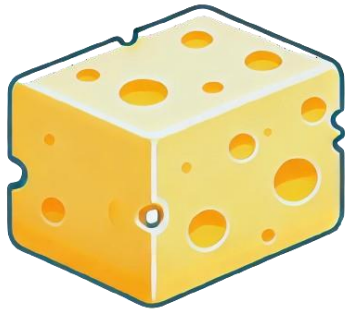
QLoRA – The Ingredients

We saw the 3 key ingredients needed to make QLoRA:



4-Bit NormalFloat

Double Quantization



Paged Optimizer

Let's bring it all together.

QLoRA – Putting it all together

Full Parameter Fine Tuning

Optimizer State (FP32)

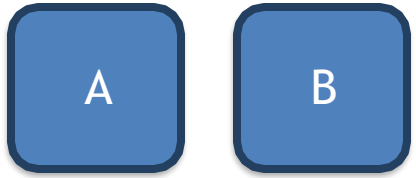
Base Model

FP16

10B => 160GB

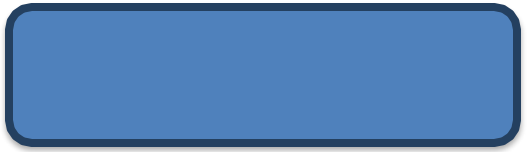
LoRA

Optimizer State (FP32)



LoRA Adapter (FP16)

Base Model



FP16

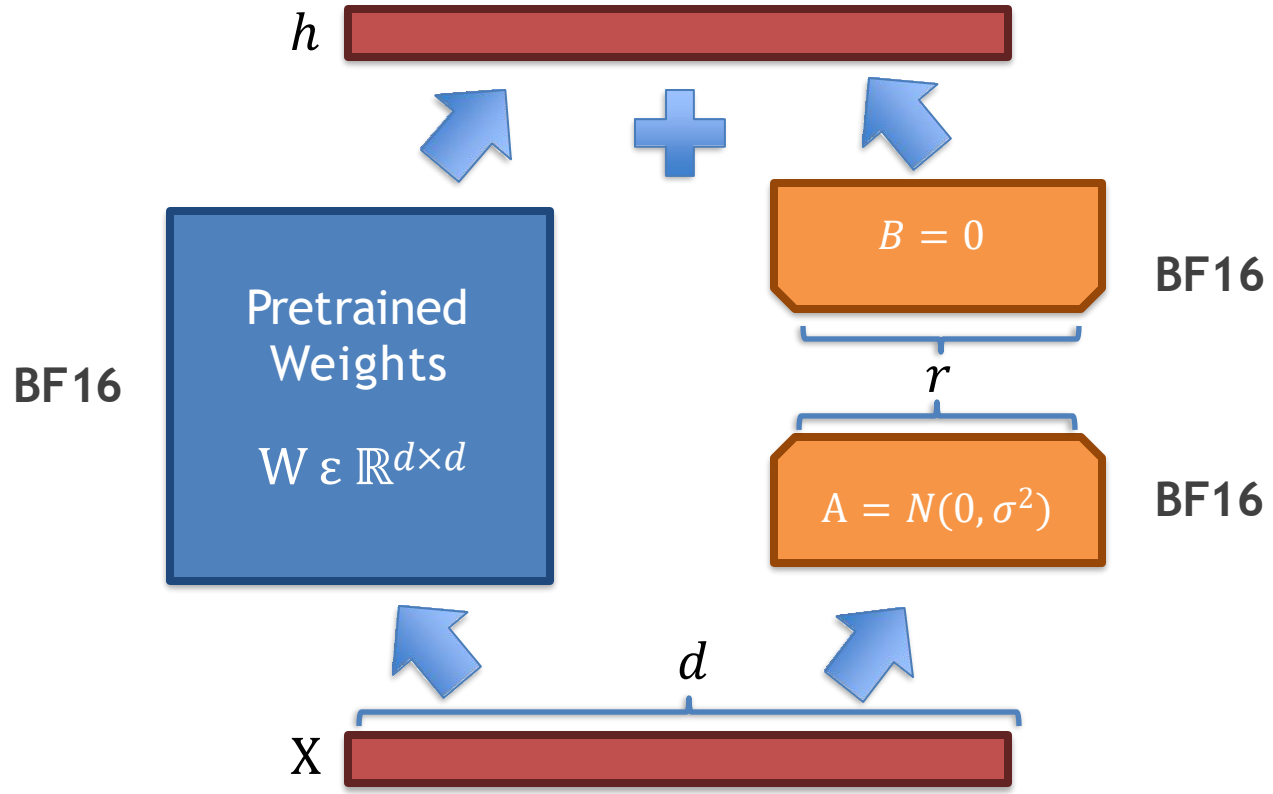
10B => ~40GB

QLoRA – Putting it all together

Before we talk about the 3 ingredients, there is another key difference that we should know.

In QLoRA we use BF16 (BrainFloat16) as compared to FP16 in LoRA.

This leads to a change in precision which is tailor-made for deep learning tasks.



1P
0R
8O
TO
PA
PA
S

QLoRA – Putting it all together

Ingredient 1:

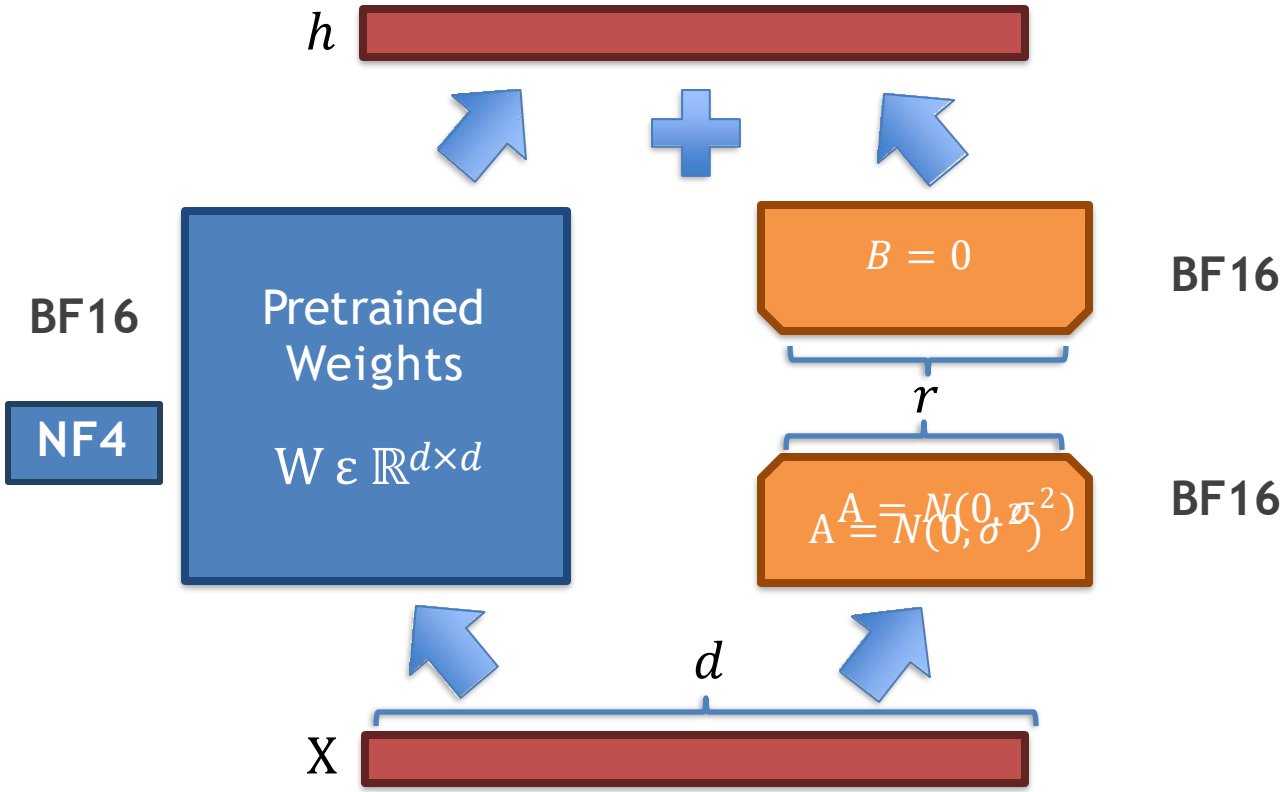
4-Bit NormalFloat

We store W, as 4-Bit NormalFloat

Ingredient 2:

Double Quantization

To convert and store, we make use of Double Quantization!



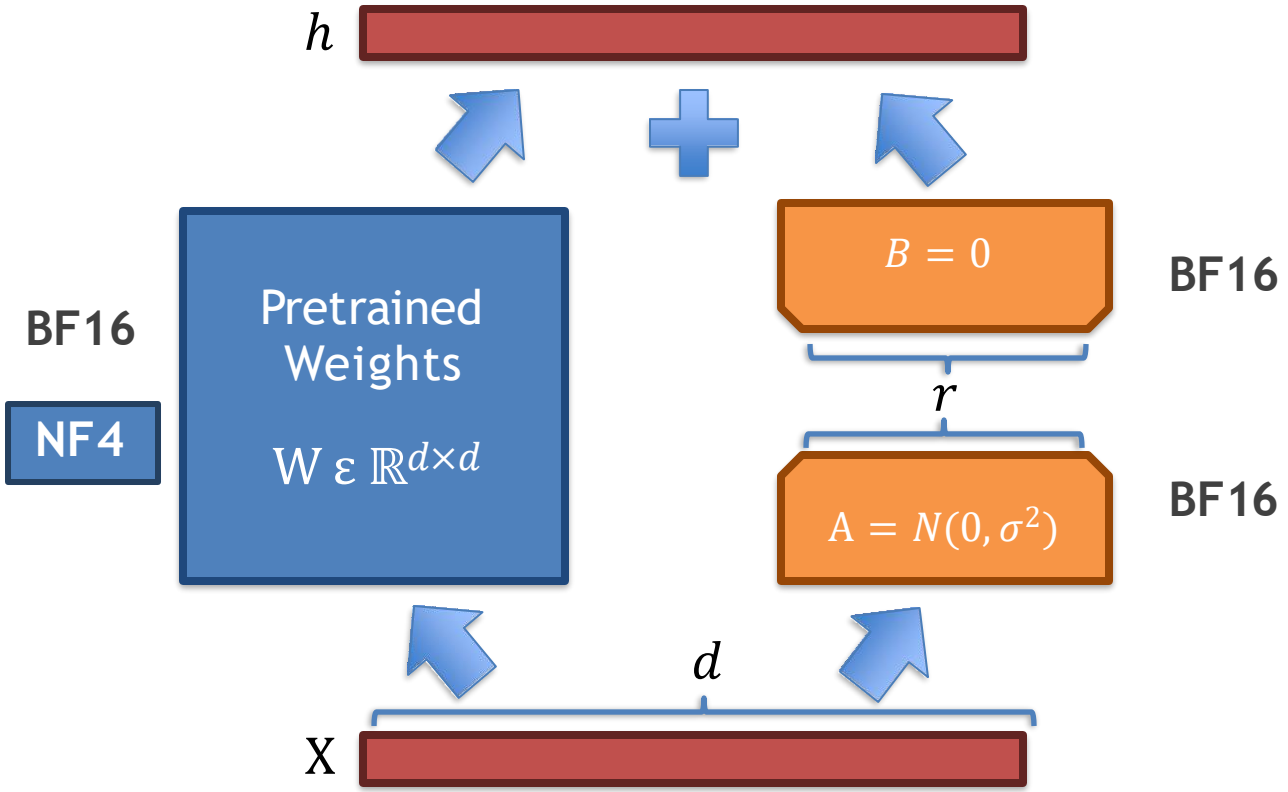
QLoRA – Putting it all together

Forward Pass

During the forward pass, we first dequantize the W weights from NF4 to BF16 for computation.

We then use the BF16 values of W, A and B to perform the required calculations.

The BF16 values of W is then deleted to save on storage!

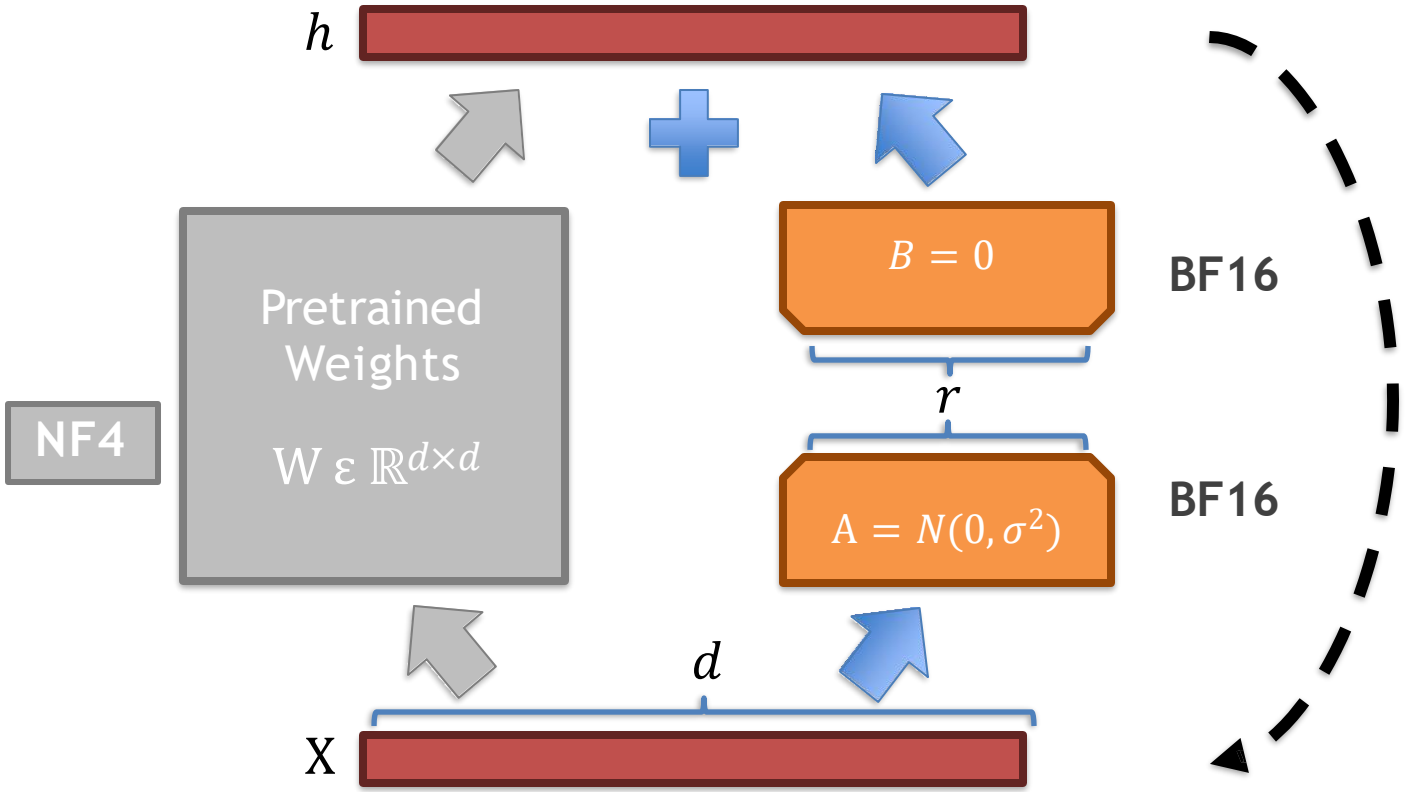


QLoRA – Putting it all together

Backward Pass

As in LoRA, we keep W weights frozen and allow the gradients to only flow through the adapters.

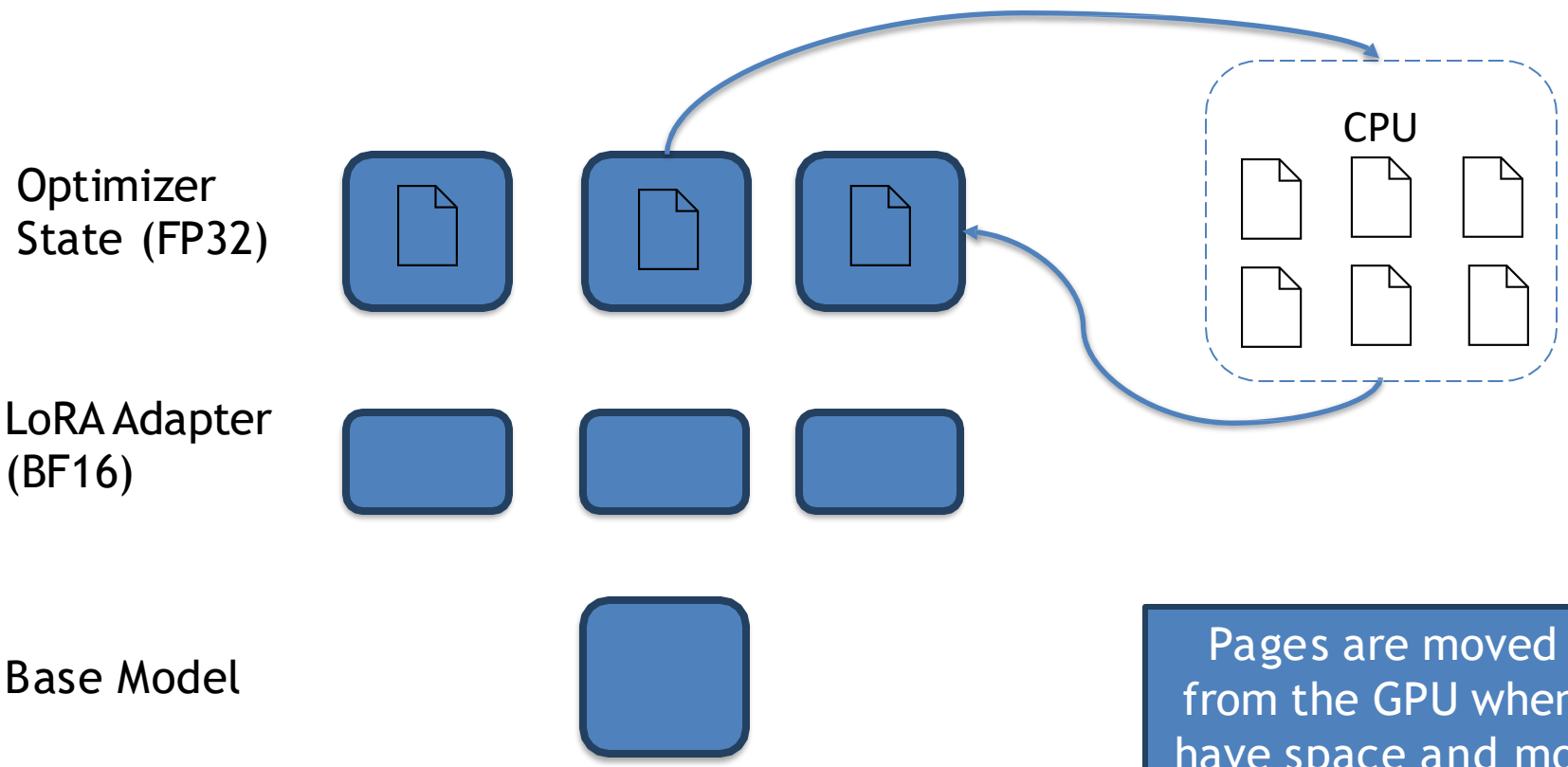
We then repeat the cycle of forward and backward passes till a minima is reached.



QLoRA – Putting it all together

Ingredient 3:

Paged Optimizer

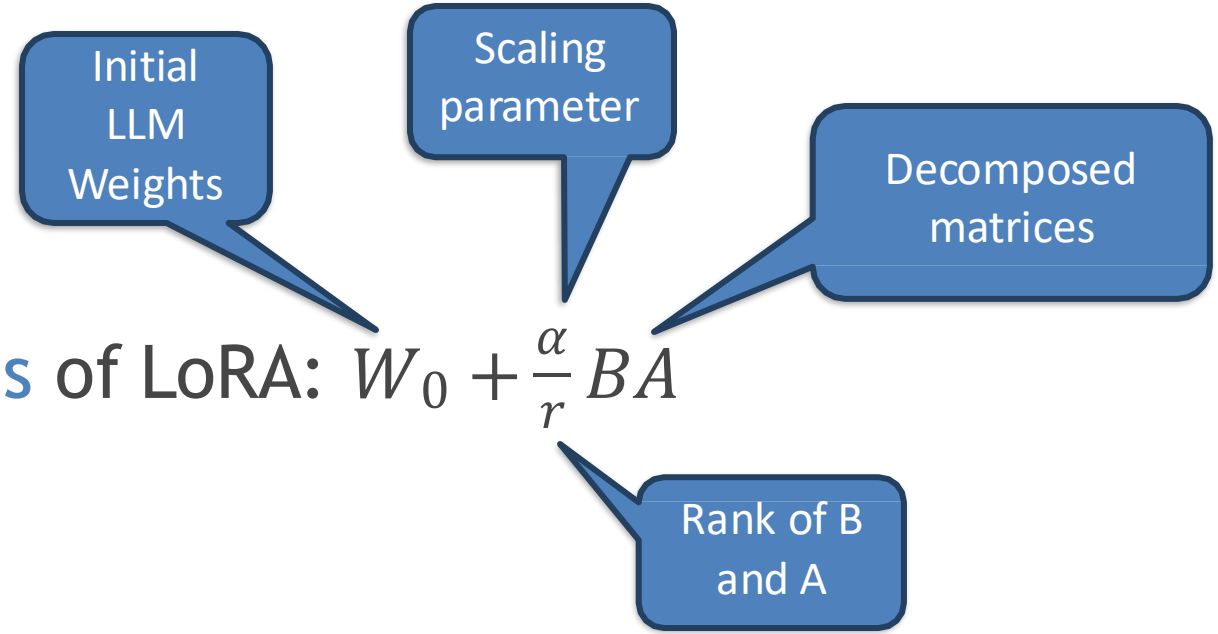


Pages are moved to the CPU from the GPU when it does not have space and moved back to GPU for when it's required and there is space.

QLoRA – Putting it all together

Putting it mathematically,

Let's start with LoRA:



Weights of LoRA: $W_0 + \frac{\alpha}{r} BA$

Forward pass in LoRA: $Y = XW_0 + \frac{\alpha}{r} XBA$

QLoRA – Putting it all together

$$Y = XW_0 + \frac{\alpha}{r} XBA$$

Let's expand the formula and see how it looks!

$$Y^{BF16} = X^{BF16} doubleDequant\left(c_1^{FP32}, c_2^{k-bit}, W_o^{NF4}\right) + \frac{\alpha}{r} X^{BF16} B^{BF16} A^{BF16}$$

where $doubleDequant\left(c_1^{FP32}, c_2^{k-bit}, W_o^{NF4}\right) = dequant\left(dequant\left(c_1^{FP32}, c_2^{k-bit}\right), W_o^{4bit}\right)$

$$= W^{BF16}$$

THANK YOU

Namah Shivaya

- https://medium.com/@shravankoninti/decoding-strategies-of-all-decoder-only-models-gpt-631faa4c449a#:~:text=In%20the%20previous%20blog%20we,steps%20and%20it%20generates%20outputs.
- https://medium.com/@shravankoninti/generative-pretrained-transformer-gpt-4b94d017a3f9
- https://medium.com/@shravankoninti/generative-pretrained-transformer-gpt-pre-training-fine-tuning-different-use-case-c93bb0553ffc
- https://medium.com/@shravankoninti/generative-pretrained-transformer-gpt-4b94d017a3f9
- https://medium.com/@YanAIx/step-by-step-into-gpt-70bc4a5d8714
- https://jalammar.github.io/illustrated-gpt2/
- https://learn.deeplearning.ai/courses/finetuning-large-language-models/lesson/1/introduction