Lecture 9: LLM-3 Finetuning

AC215

Pavlos Protopapas
SEAS/Harvard

Outline

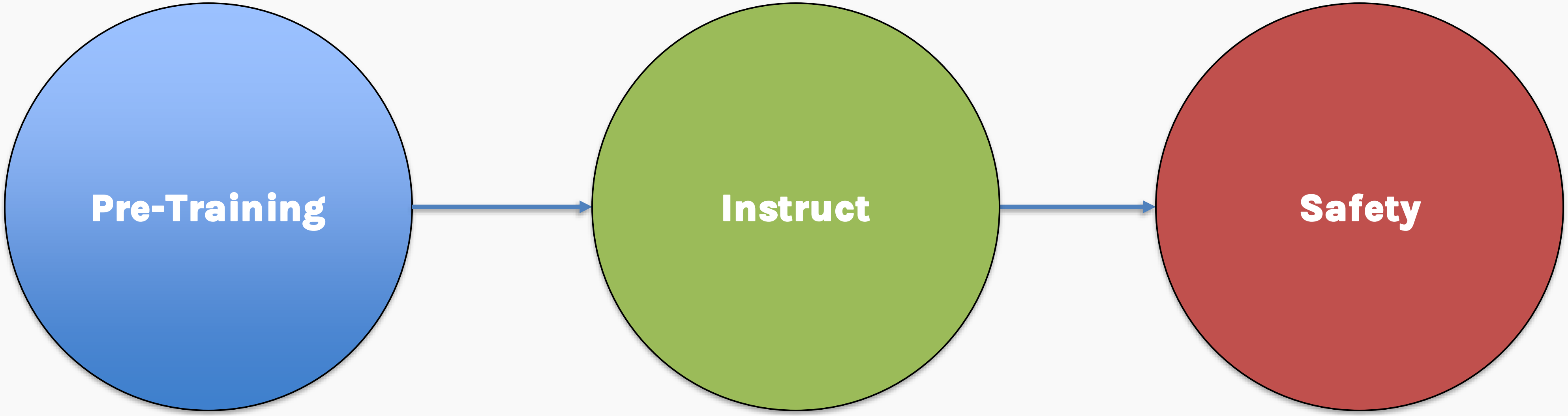
- Training Cycle – LLM
- Instruction-tuning
    - Full Parameter
    - PEFT
- LoRA
- QLoRA

Outline

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

Training Cycle - LLM

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



Training Cycle - LLM

Pre-Training

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.

This is what we've learned when we talked about how GPT works

Training Cycle - LLM

Instruct

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

Safety

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

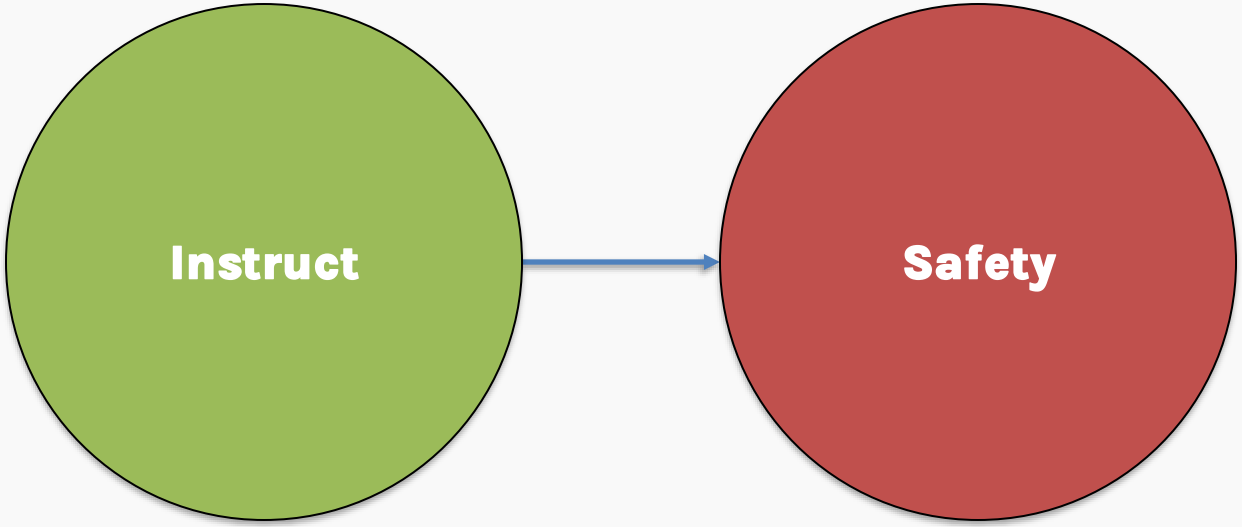
Process:
Involves further fine-tuning. We use RLHF to provide feedback on model outputs.

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

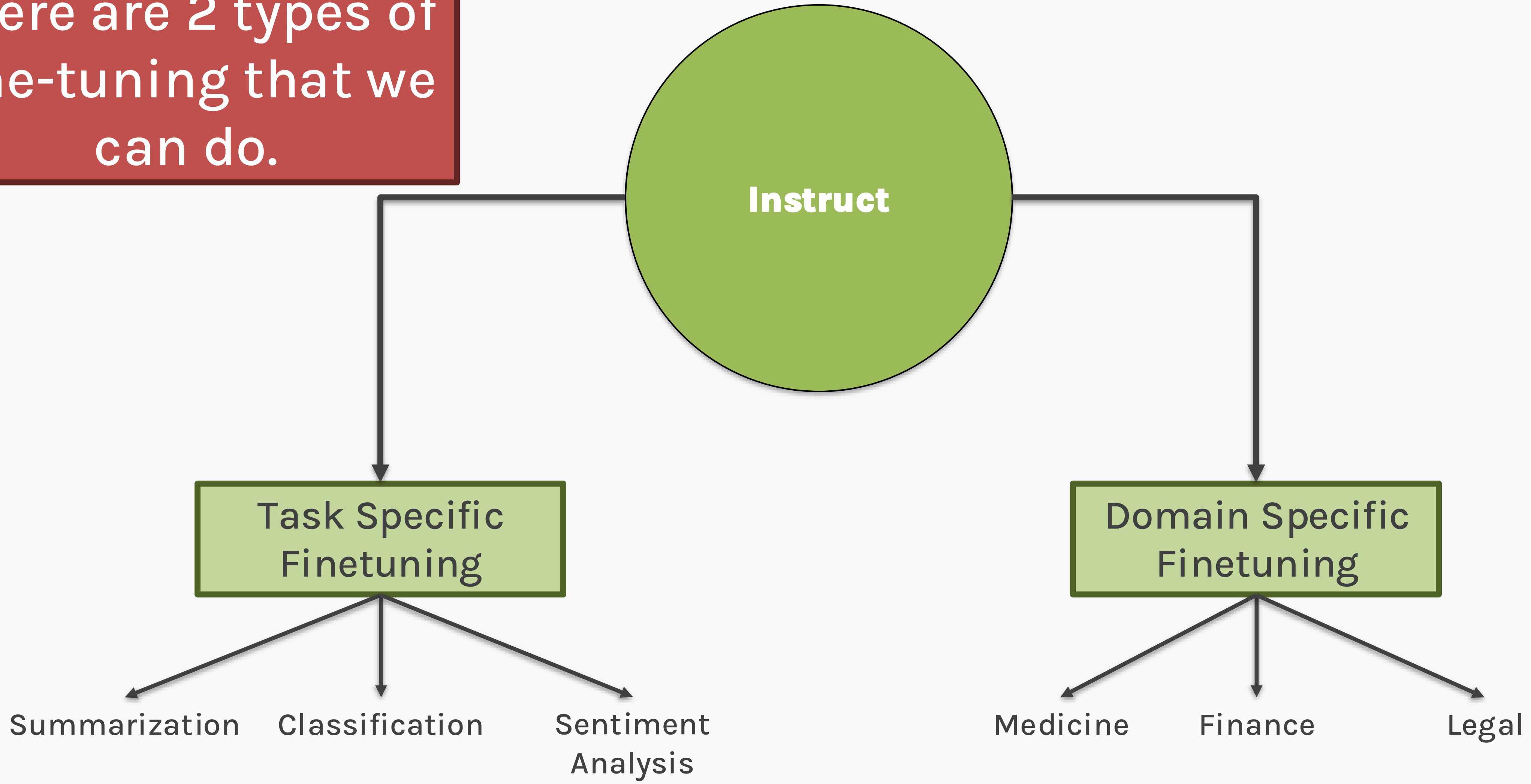
So, fine-tuning takes place in 2 stages.



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

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 method of prompt engineering where the model is shown task demonstrations as part of the prompt. • No change in model parameters.	• 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.

You may recall **in-context learning** from a previous lecture about 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.

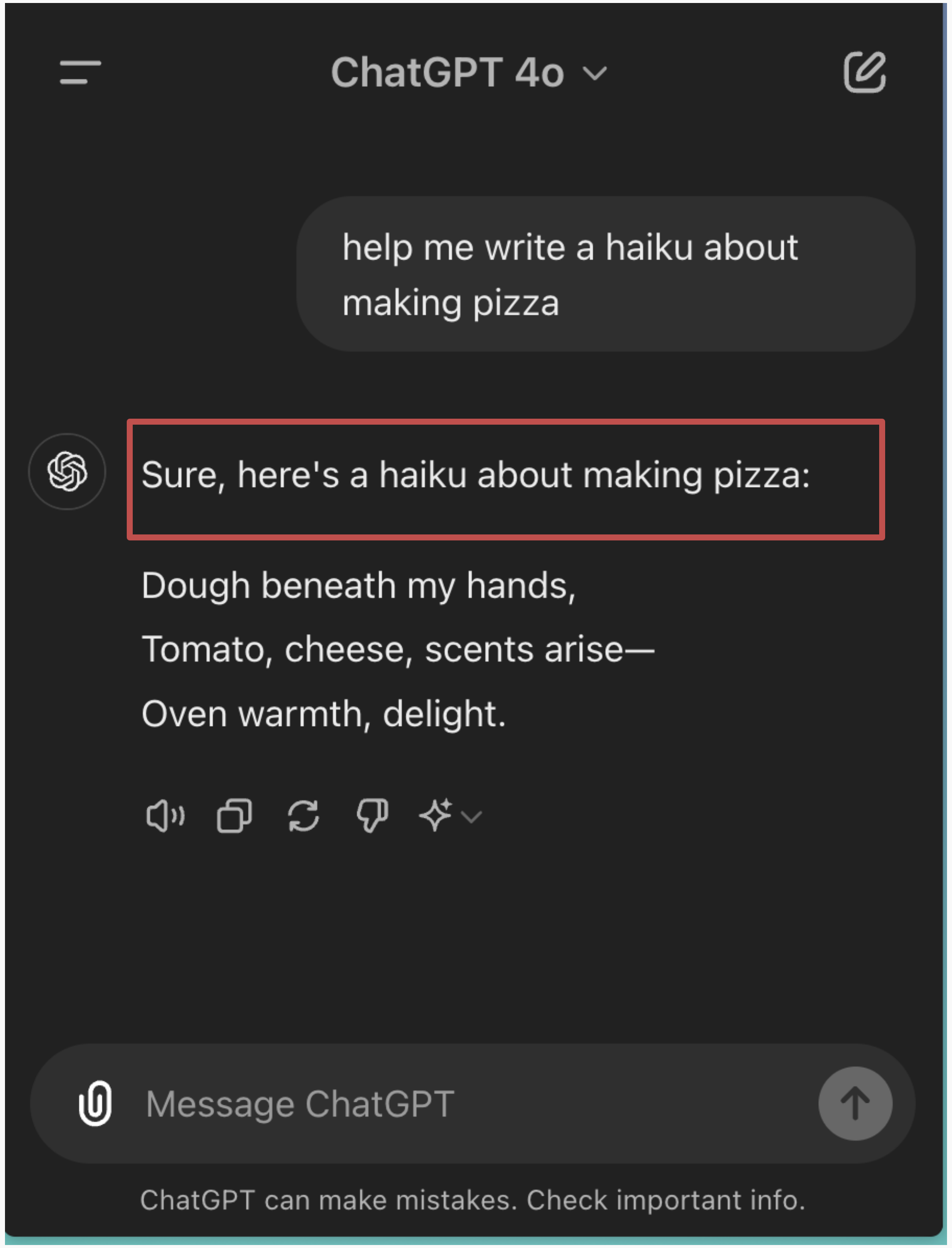
For e.g.: Summarize this, translate that, etc

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

Instruction-tuning (Full Parameter)

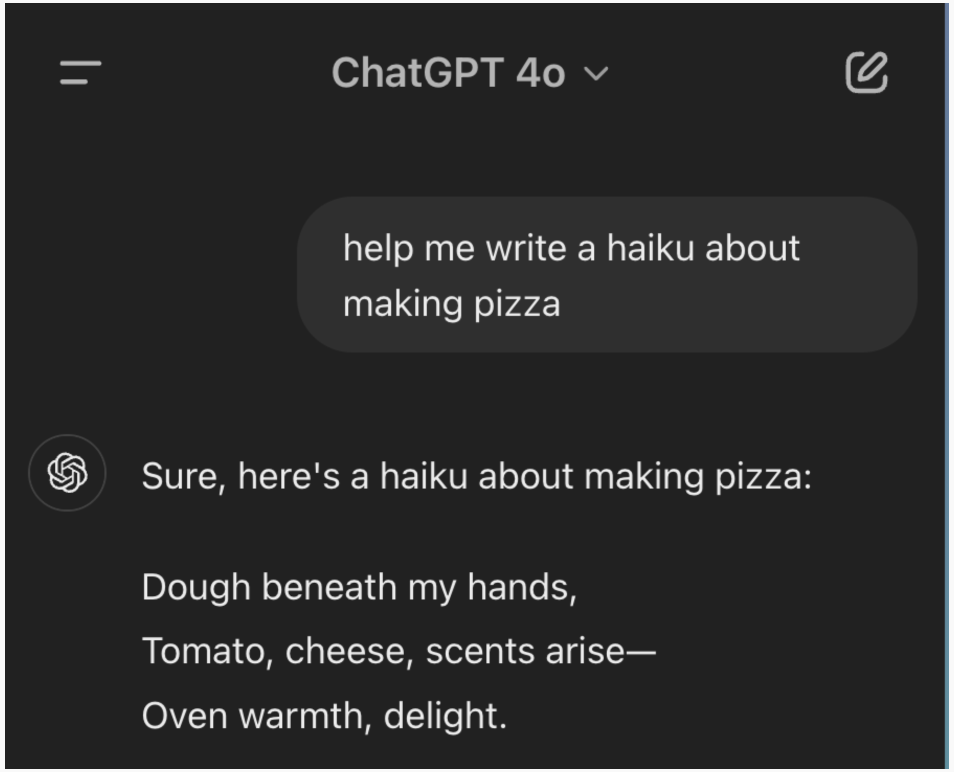
For e.g., if you train a model specifically for question answering:

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 include 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 and so on.

- 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)

Not so fun Fact:

🎉 ~~**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)

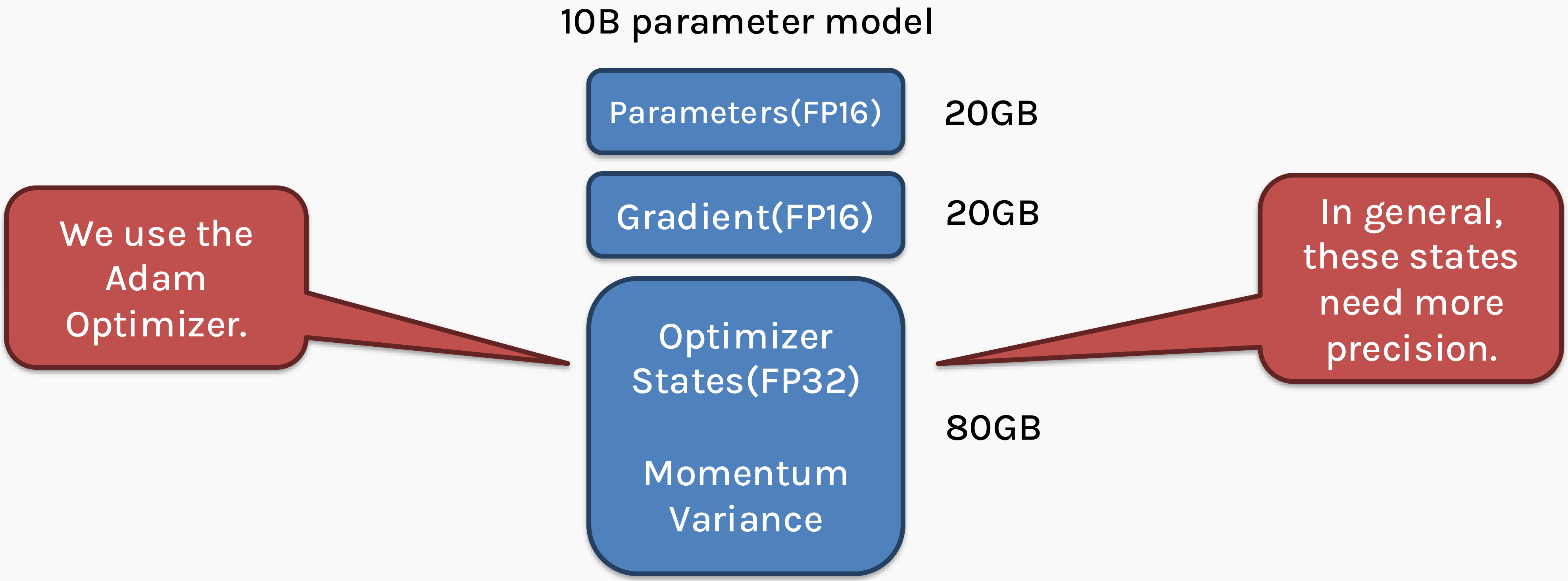
Let's take a fine-tuning example now.

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.

10B parameter model

Parameters(FP16)	20GB
Gradient(FP16)	20GB
Optimizer States(FP32) Momentum Variance	80GB

4090 GPU 24GB

4090 GPU 24GB

4090 GPU 24GB

4090 GPU 24GB

4090 GPU 24GB

This model needs at least 5 top-of-the-line consumer-grade GPU's to finetune.

Instruction-tuning (Full Parameter)

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

10B parameter model

Parameters(FP16) 20GB

Gradient(FP16) 20GB

Optimizer States(FP32)

Momentum Variance

>80GB

4090 GPU 24GB

4090 GPU 24GB

4090 GPU 24GB

4090 GPU 24GB

4090 GPU 24GB

4090 GPU 24GB

4090 GPU 24GB

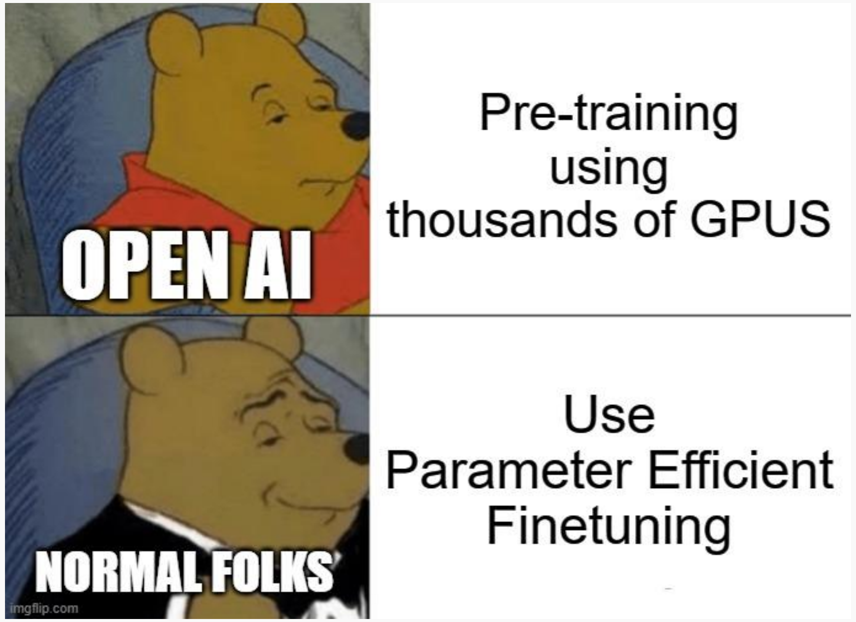
>80GB because we need to save activations and a few other things

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**
- LoRA
- QLoRA

Instruction-tuning (PEFT)

PEFT stands for Parameter Efficient Finetuning.

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

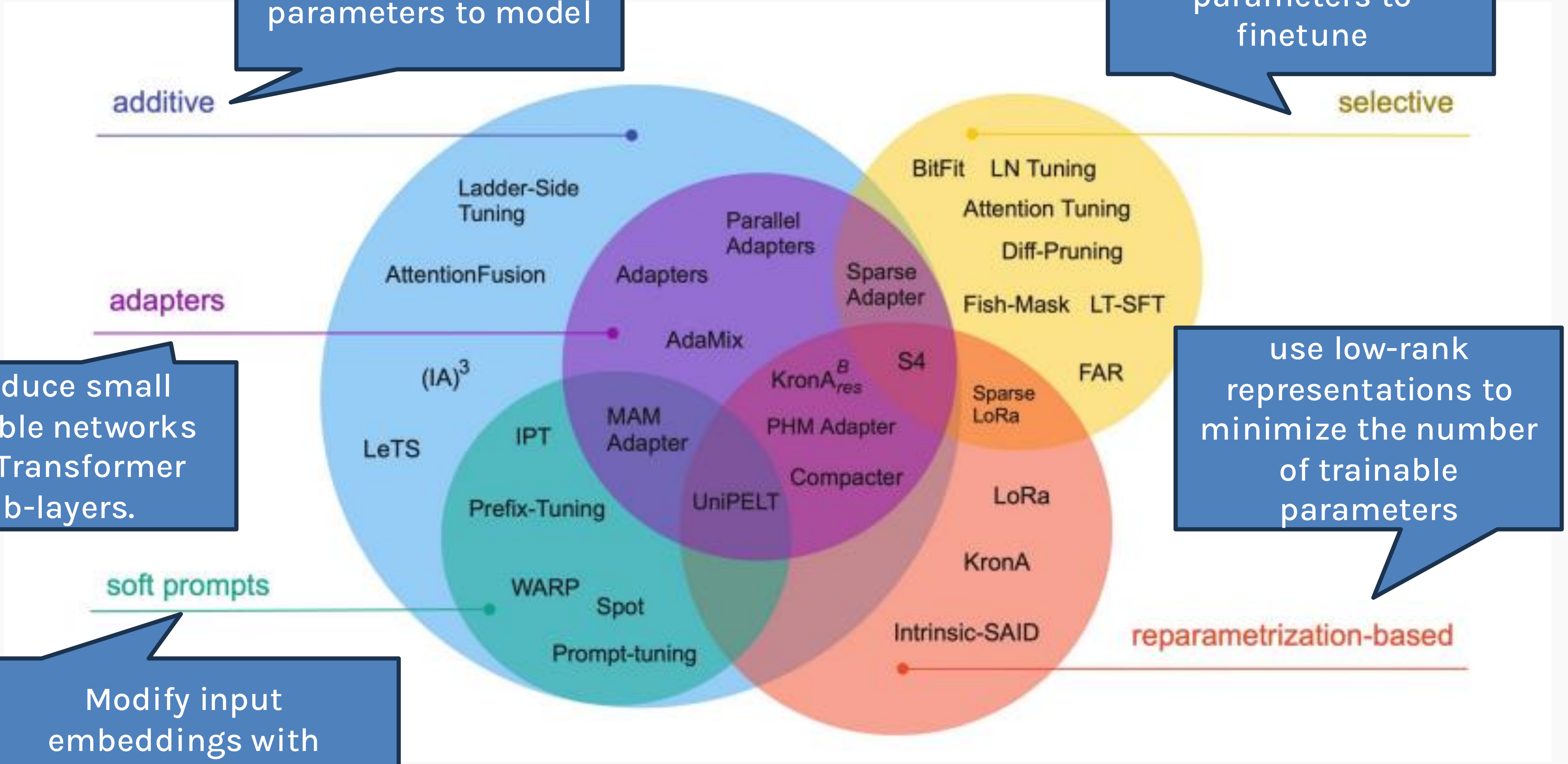
There are majorly three methods to do PEFT.

1. Additive
2. Selective
3. Reparameterization

Instruction-tuning (PEFT)

Add trainable layers or parameters to model

Subsets the existing parameters to finetune



Introduce small trainable networks **after** Transformer sub-layers.

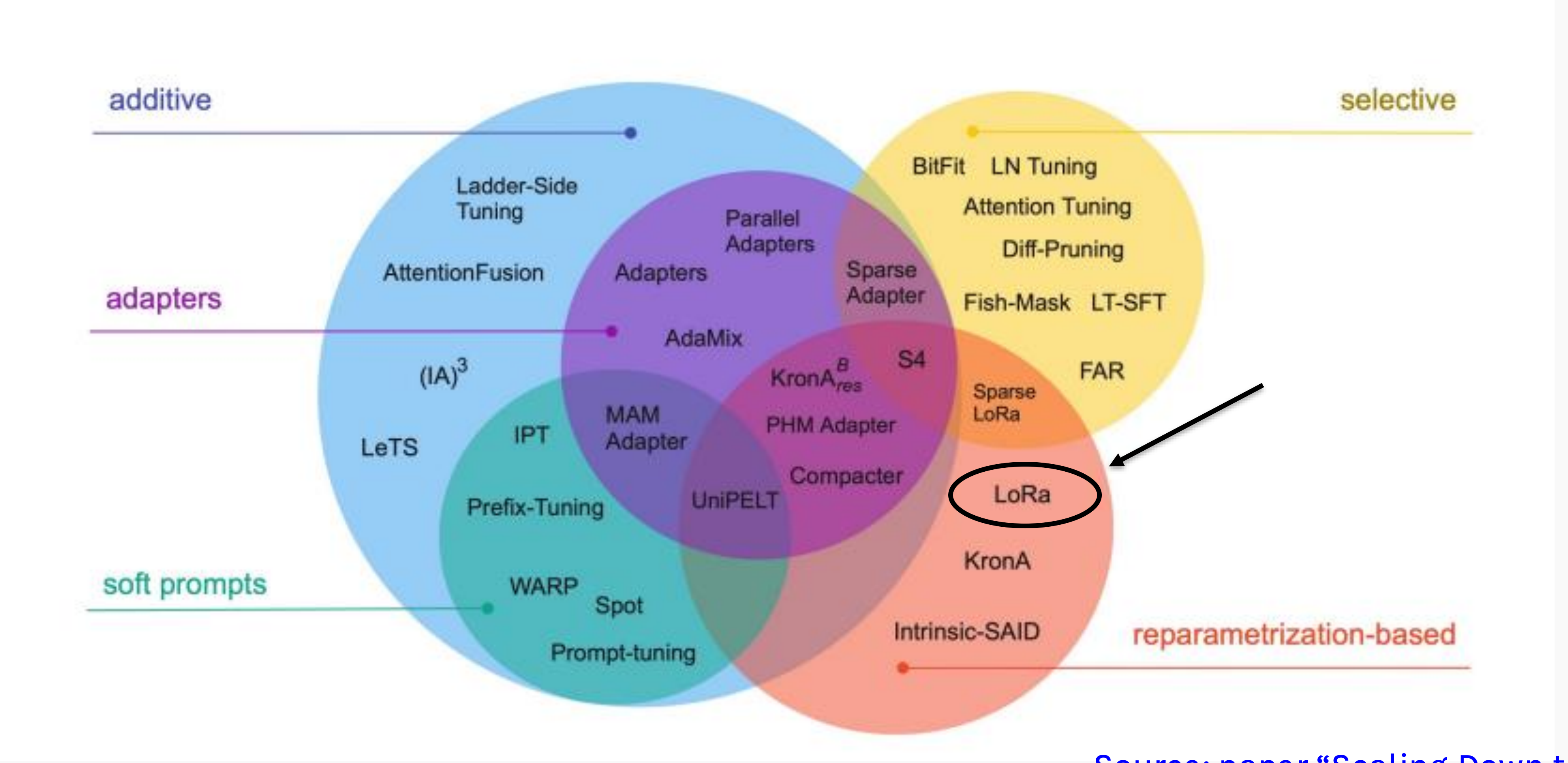
use low-rank representations to minimize the number of trainable parameters

Modify input embeddings with trainable tensors to steer the model's behavior

[Source: paper "Scaling Down to Scale Up" (arxiv.org)]

Instruction-tuning (PEFT)

There are a lot of techniques. We're interested in LoRA, which is one of the most popular.



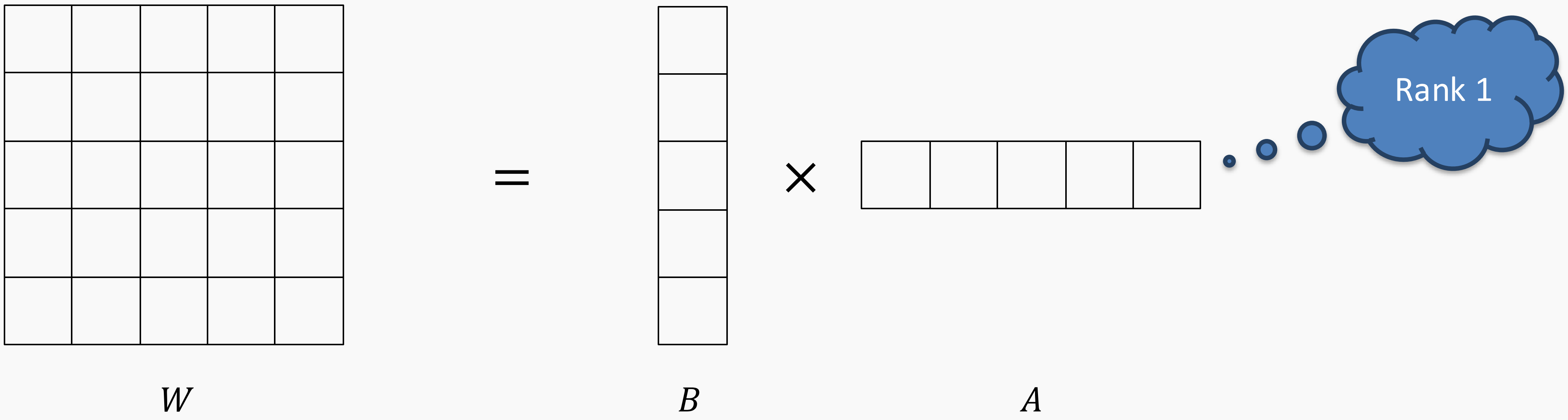
[Source: paper "Scaling Down to Scale Up" (arxiv.org)]

Outline

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

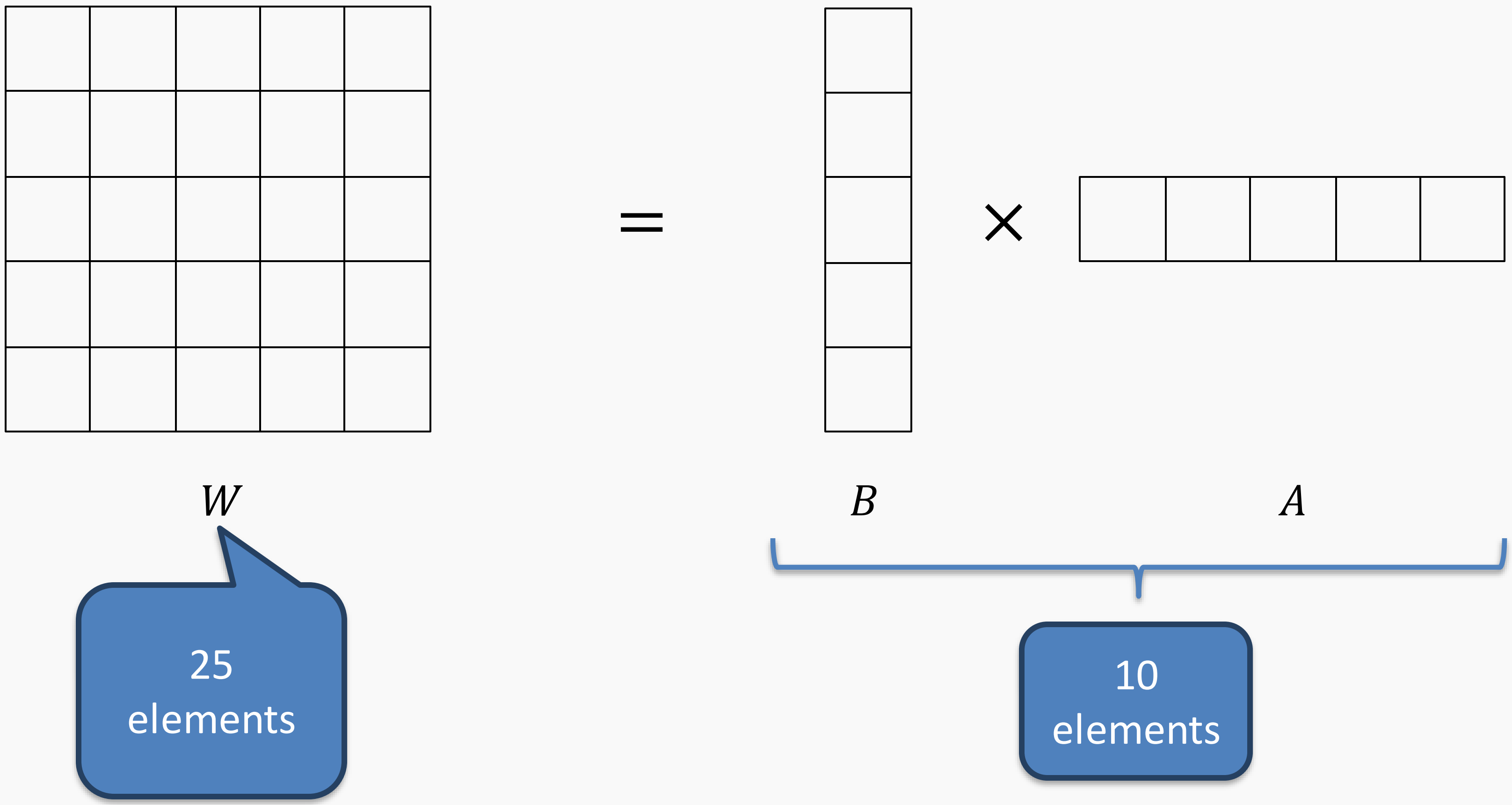
LoRA - Intuition

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



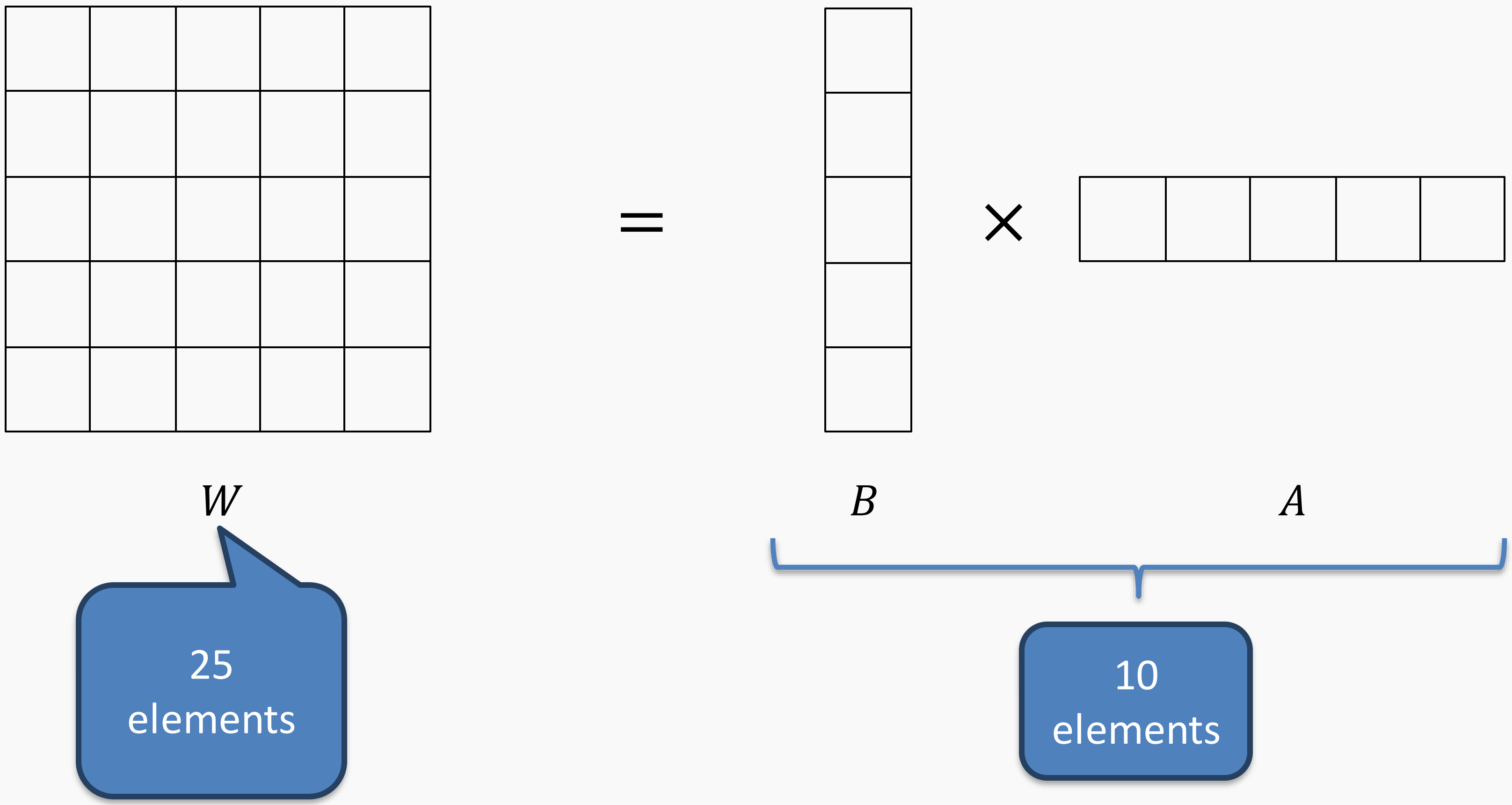
LoRA - Intuition

LoRA revolves around the idea that any matrix $W \in R^{m\times n}$ can be decomposed into $\mathbf{W} = \boldsymbol{BA}$ 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.

Initial LLM Weights

Update matrix

Scaling parameter

Decomposed matrices

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

Rank of BA

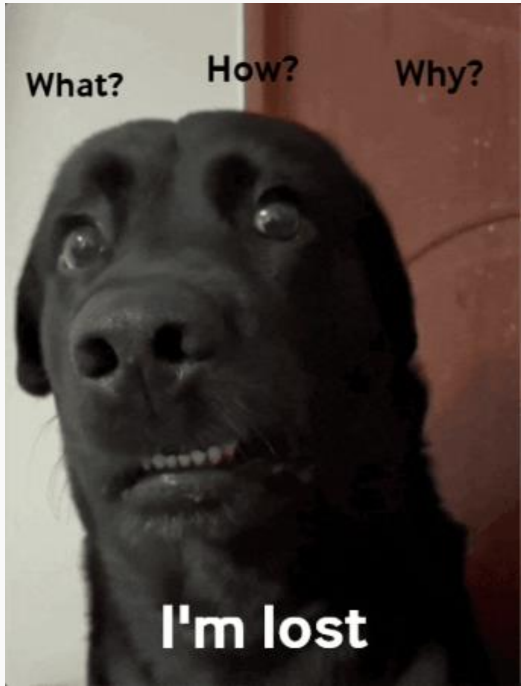
Remember, we are decomposing the update matix (ΔW), and not the original weights W_0.

LoRA - Working

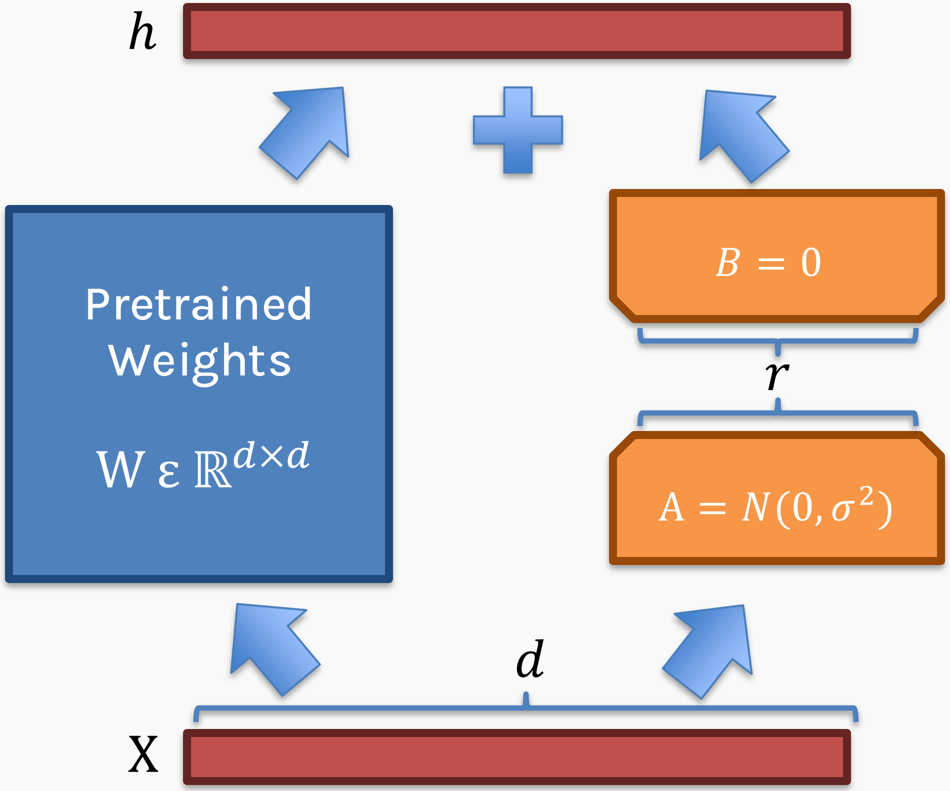
$$W_0 + \Delta W = W_0 + \frac{\alpha}{r} BA$$

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

Now, let's look at this diagrammatically.

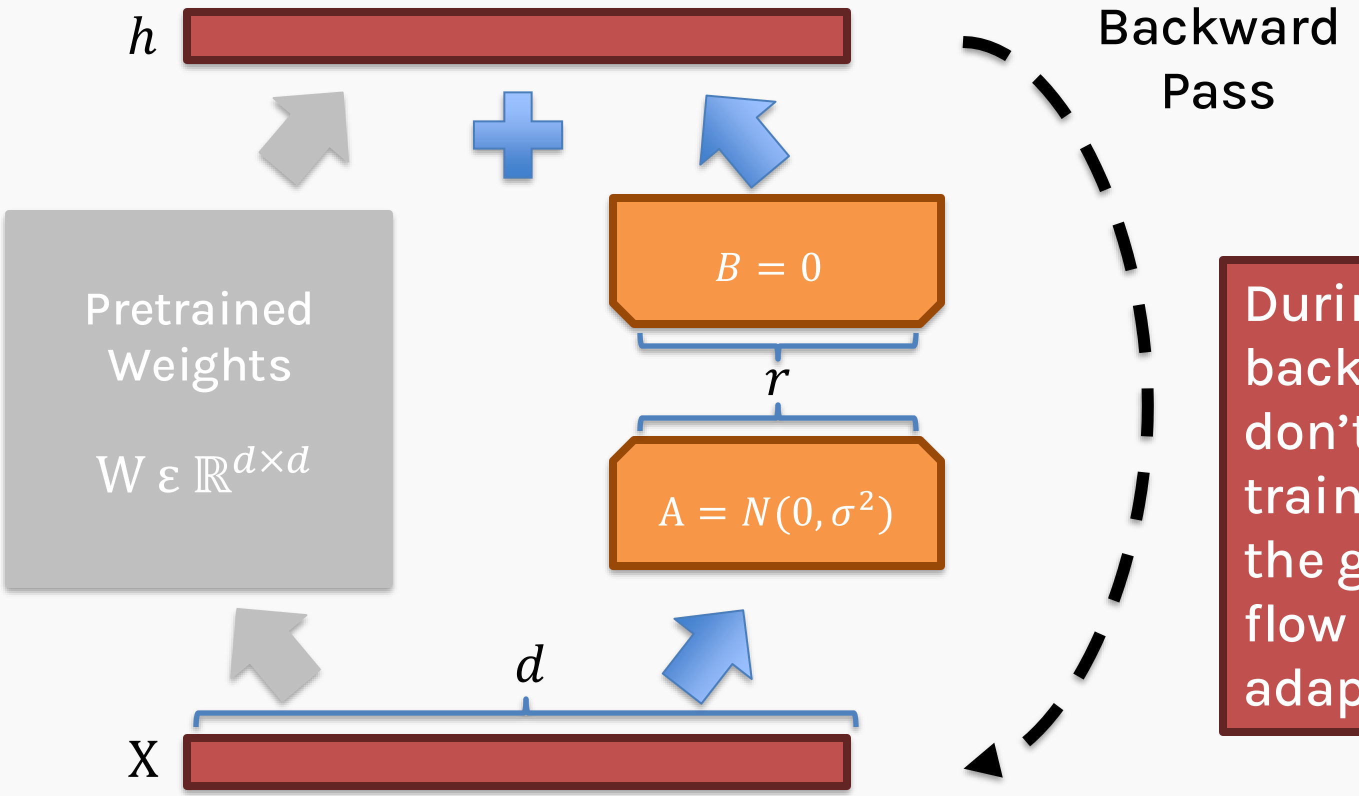


LoRA - Working



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

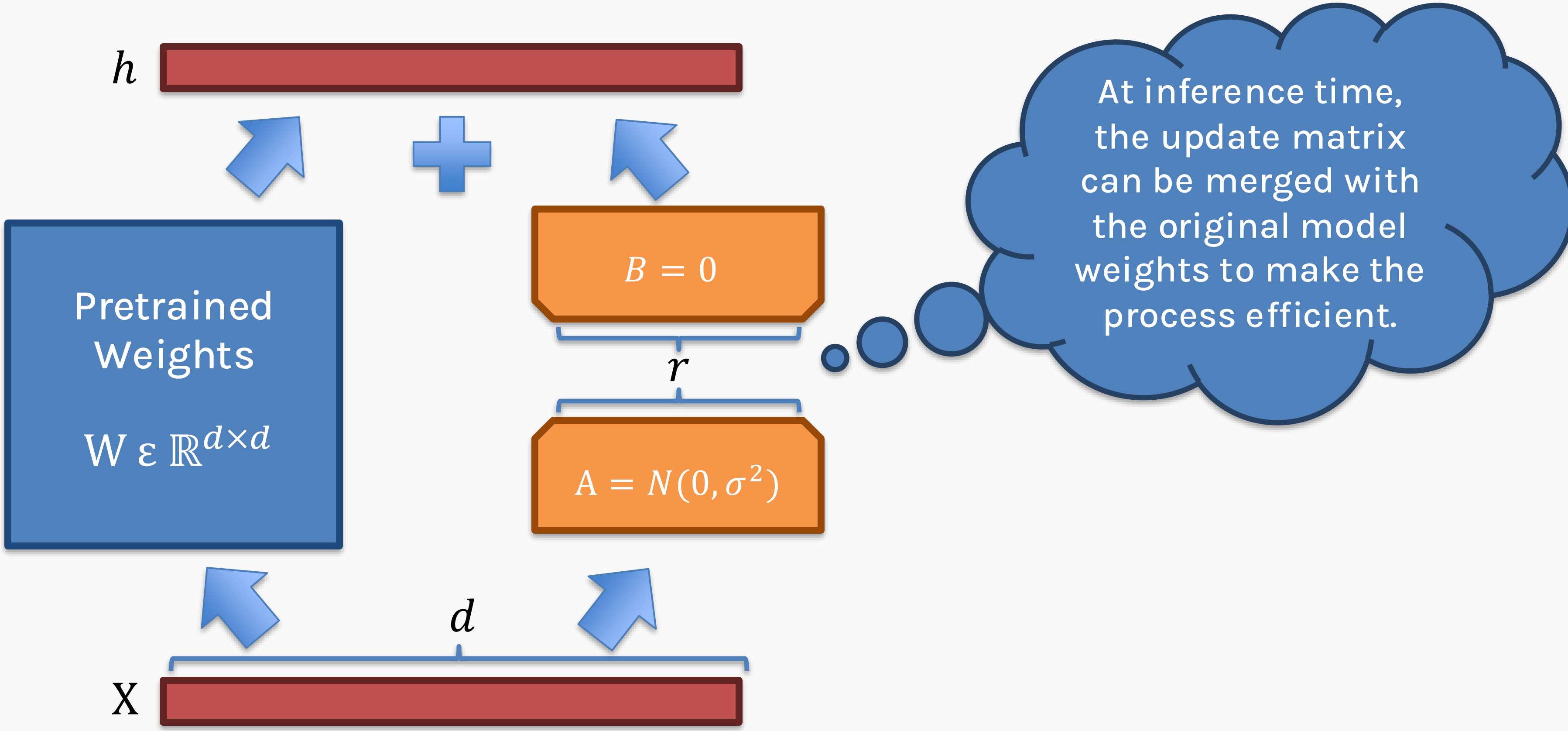
LoRA - Working



During the backward pass, we don't use the pre-trained weights and the gradients only flow through the adapters.

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 - 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:

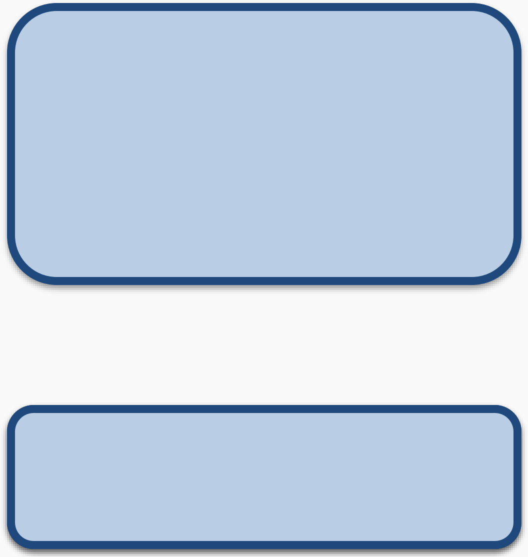
1. Much faster
2. Finetuning can be achieved using less GPU memory
3. Cost efficient
4. Less prone to “catastrophic forgetting” since the original model weights are kept the same.

LoRA – Isn’t it enough?

Full Parameter Fine Tuning

Optimizer State (FP32)

Base Model (FP16)



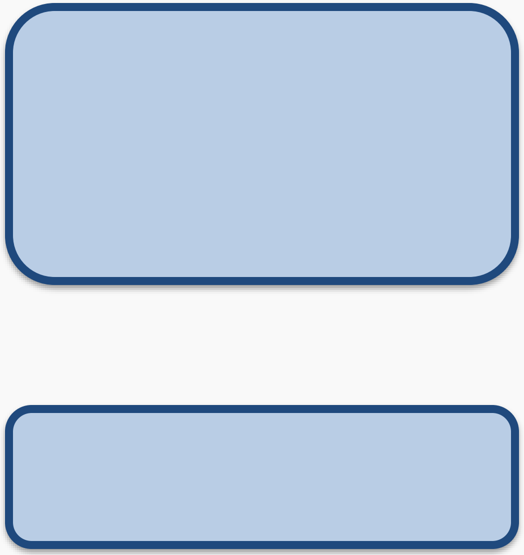
10B → 120GB

LoRA – Isn’t it enough?

Full Parameter Fine Tuning

Optimizer State (FP32)

Base Model (FP16)



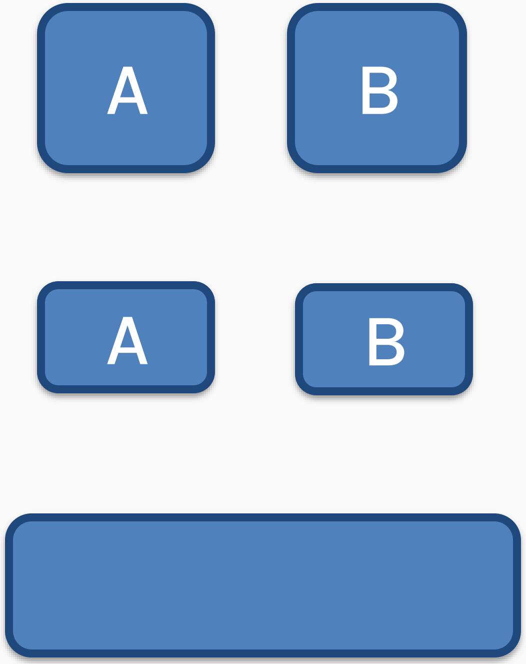
10B → 120GB

LoRA

Optimizer State (FP32)

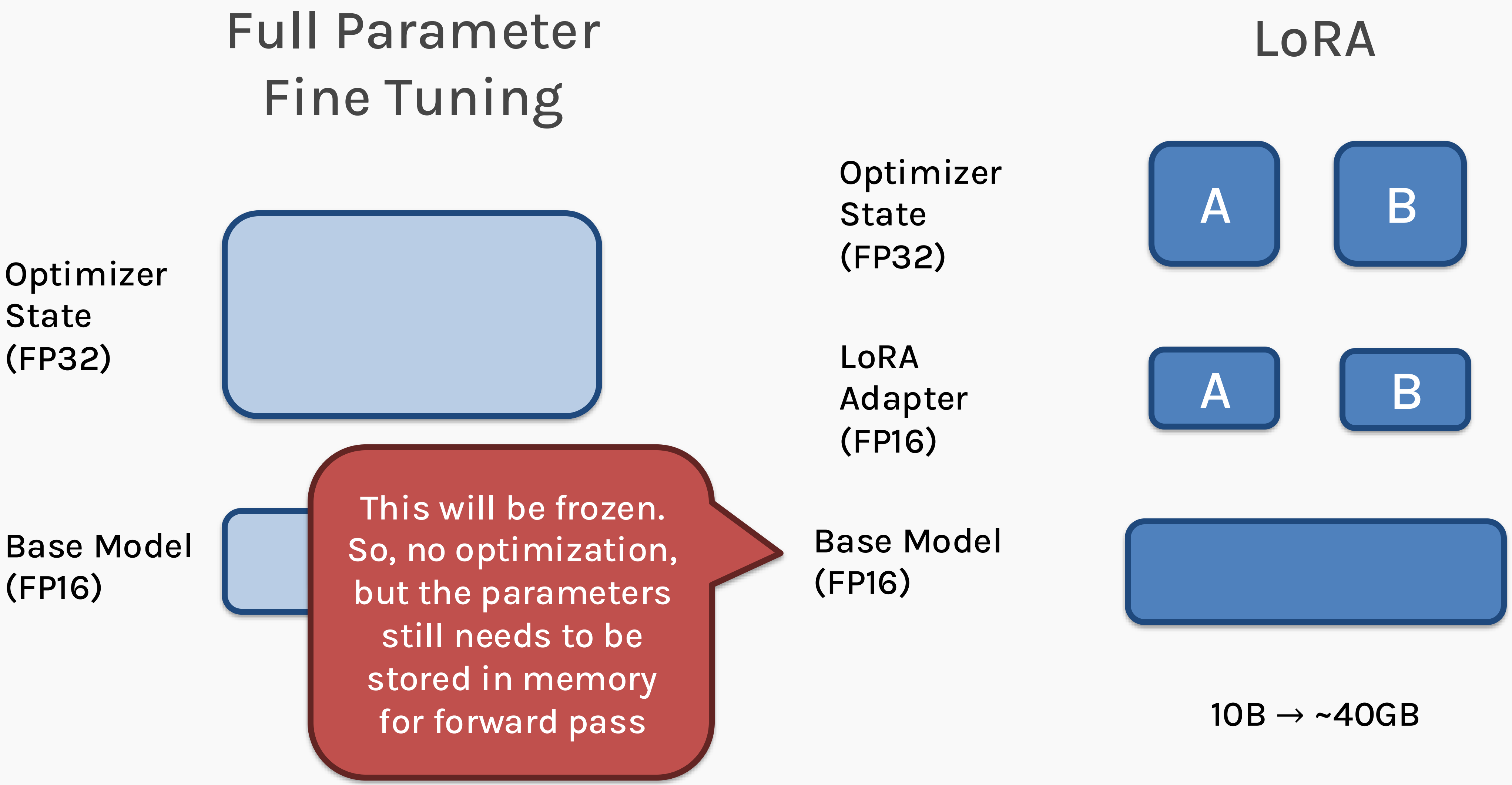
LoRA Adapter (FP16)

Base Model (FP16)



10B → ~40GB

LoRA – Isn’t it enough?



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.

Outline

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

QLoRA

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

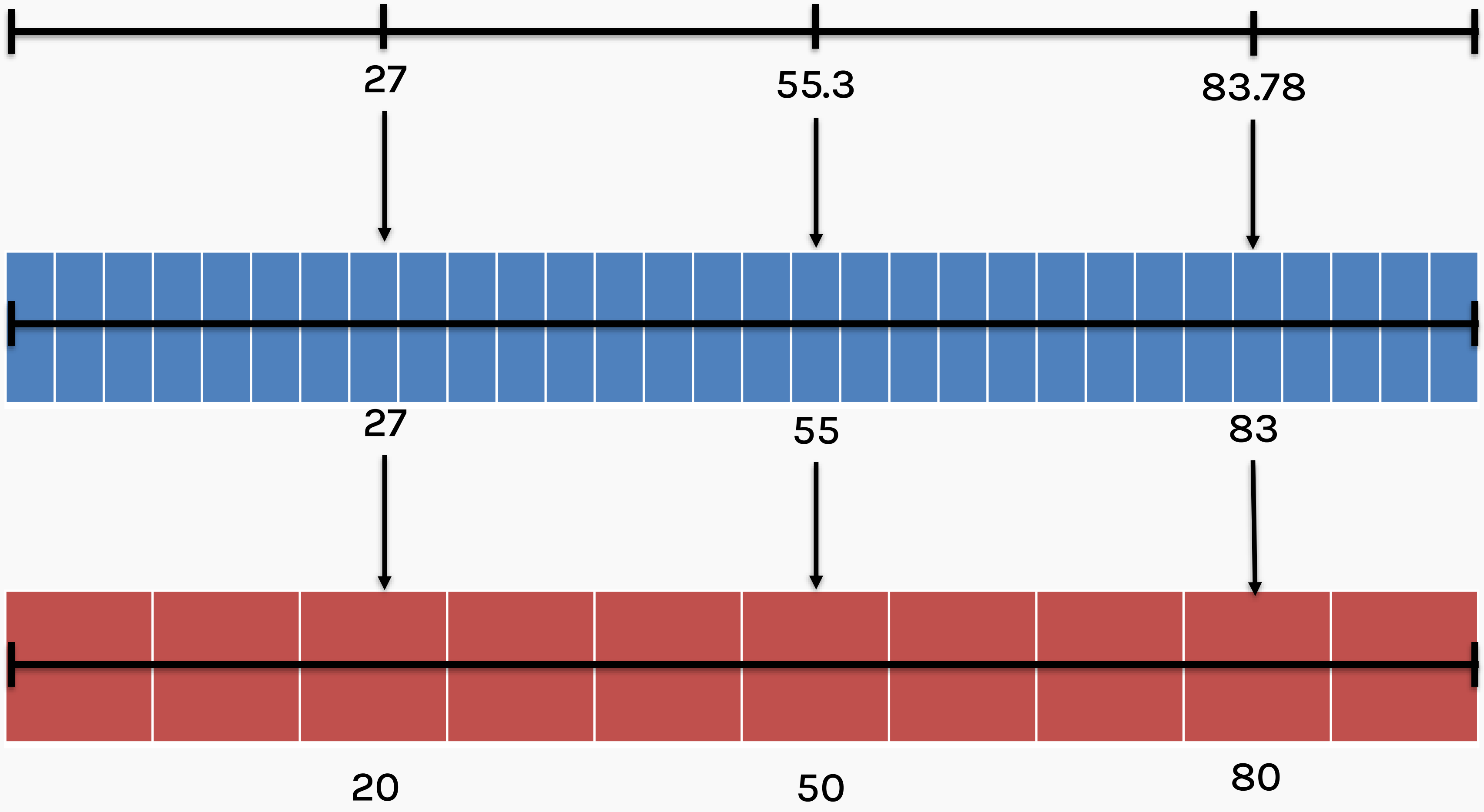
- Before we dive into what QLoRA is, let's look at what quantization is.

Think of quantization as ' splitting range into buckets '.

QLoRA

Think of quantization as ‘ splitting range into buckets ’.

Any number between
0 and **100**



1 num =
infinite bytes

Quantized by
whole numbers

1 num =
0.875 bytes

Quantized by
10s

1 num =
0.5 bytes
(3 bits ~
0.375 bytes)

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,

$X^{FP32} \longrightarrow$

1.5	2.3	3.7	4.1	5.6	6.8	7.9	8.4	9.2	10.2

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

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:

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

There will always be some Dequantization error here

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

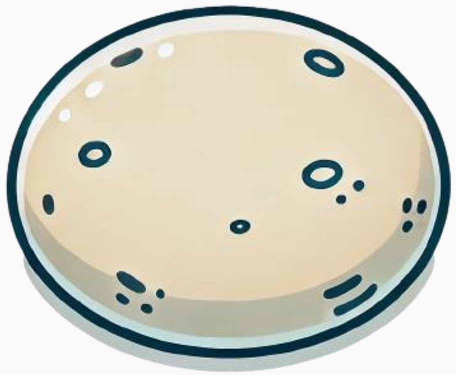
QLoRA – The Pizza

Imagine QLoRA to be a mouthwatering pizza.

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

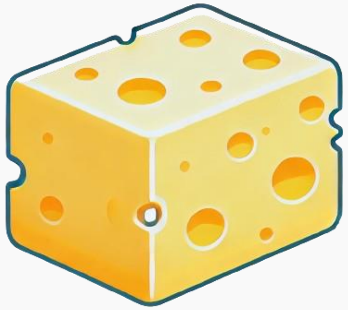
QLoRA – The Ingredients

There are 3 key ingredients which helps us make QLoRA:



4-Bit NormalFloat

Double Quantization



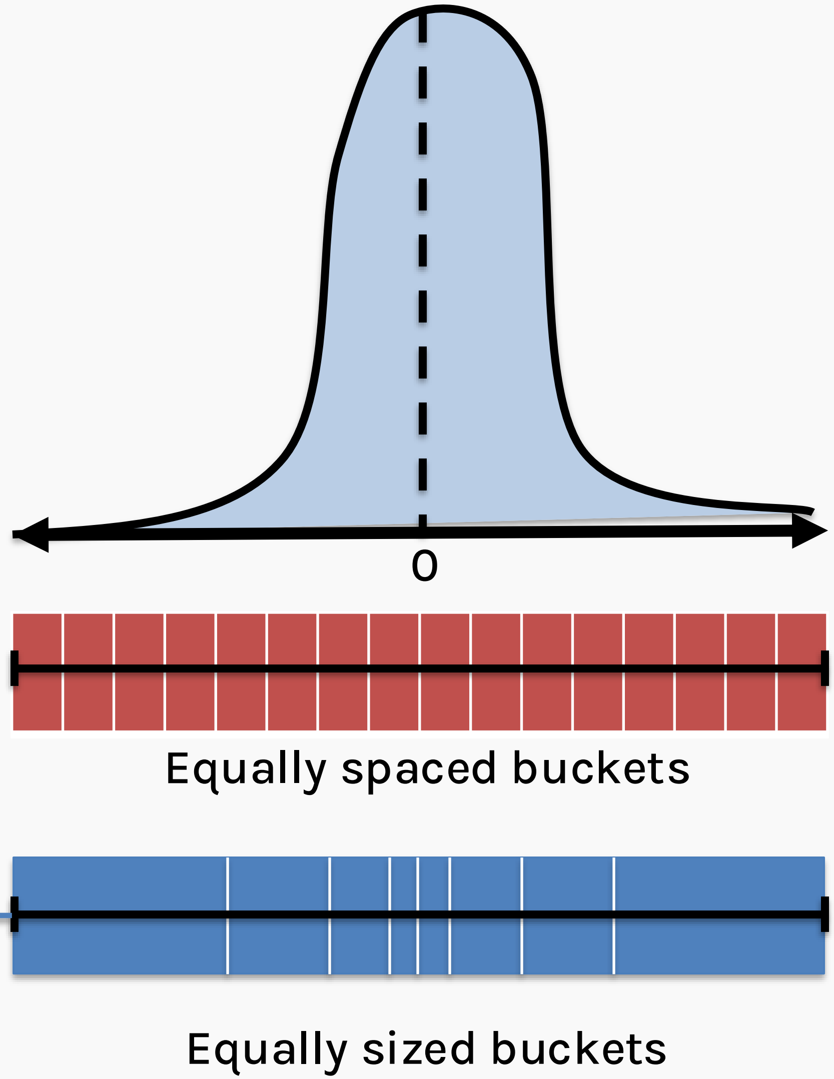
Paged Optimizer

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.



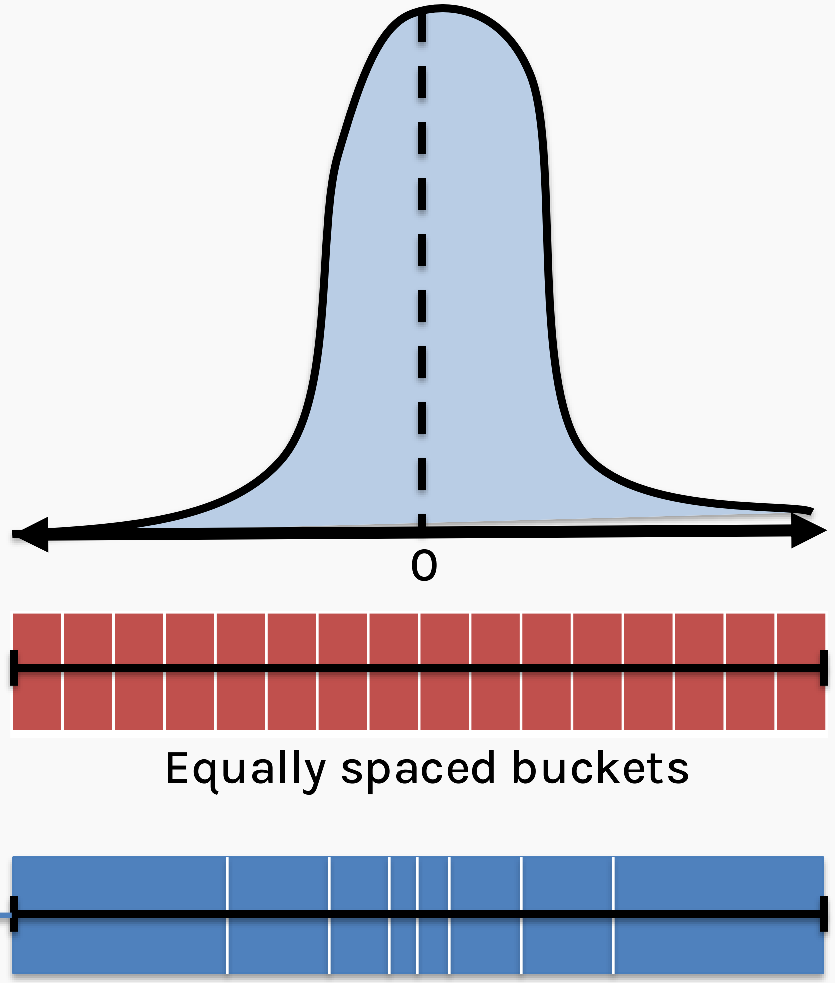
This is an enhanced version of **quantile quantization**.

QLoRA – Ingredient 1: 4-Bit NormalFloat

Why use 4-bit NormalFloat

Designed for efficient storage and computation in machine learning.

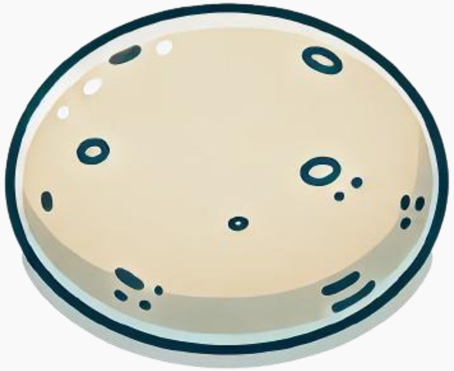
Most datasets in machine learning are normally distributed and precision around the mean is valuable.



This is an enhanced version of **quantile quantization**.

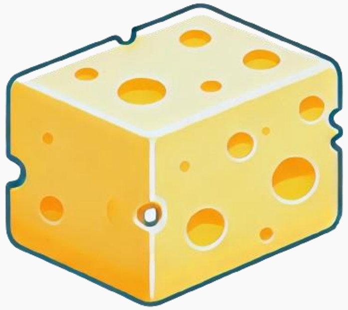
QLoRA – The Ingredients

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?

Takes up memory

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

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

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

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

Weight Tensor

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

Rescaled Weight Tensor

-4	-2	0	-22	16
[illegible]	[illegible]	40	99	-57
[illegible]	[illegible]	-9	48	27
[illegible]	[illegible]	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	-0.9
-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

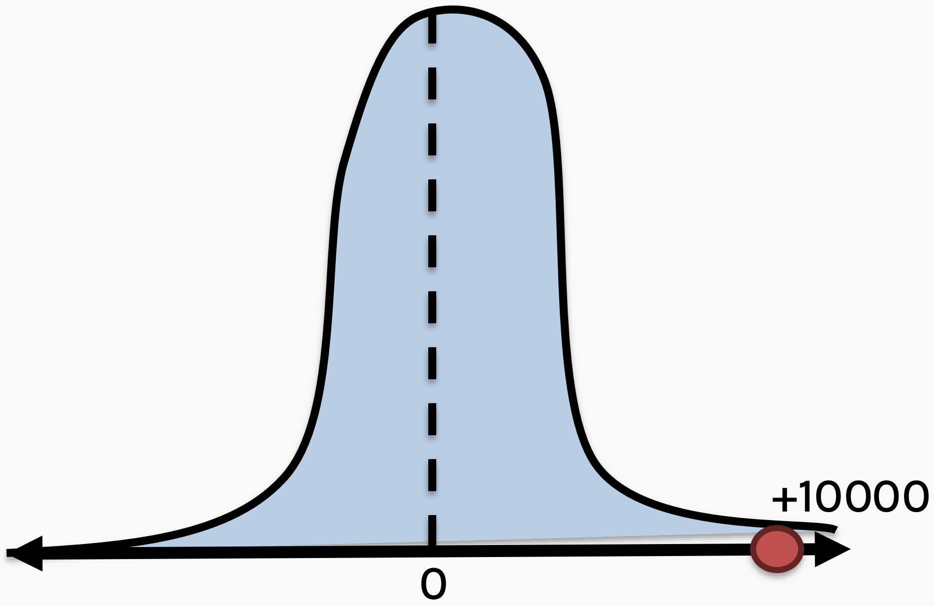
-4	-2	0	-22	16
-60	10	40	99	-57
-5	-88	-9	48	27
72	-100	-50	-18	40
22	8	-81	[illegible]	[illegible]

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.

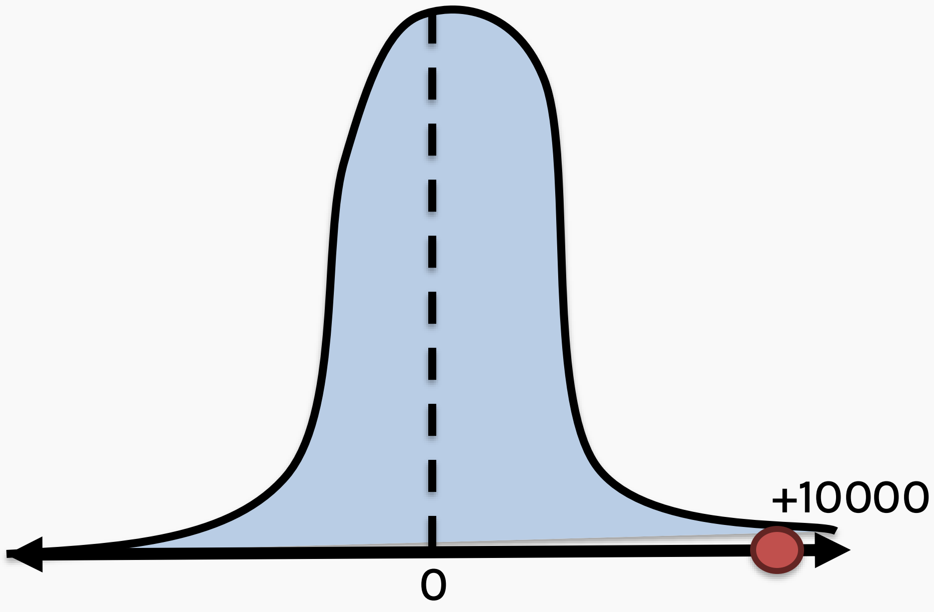


This is unbounded and could take up any maximum value (an outlier!).

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

QLoRA – Ingredient 2: Double Quantization

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



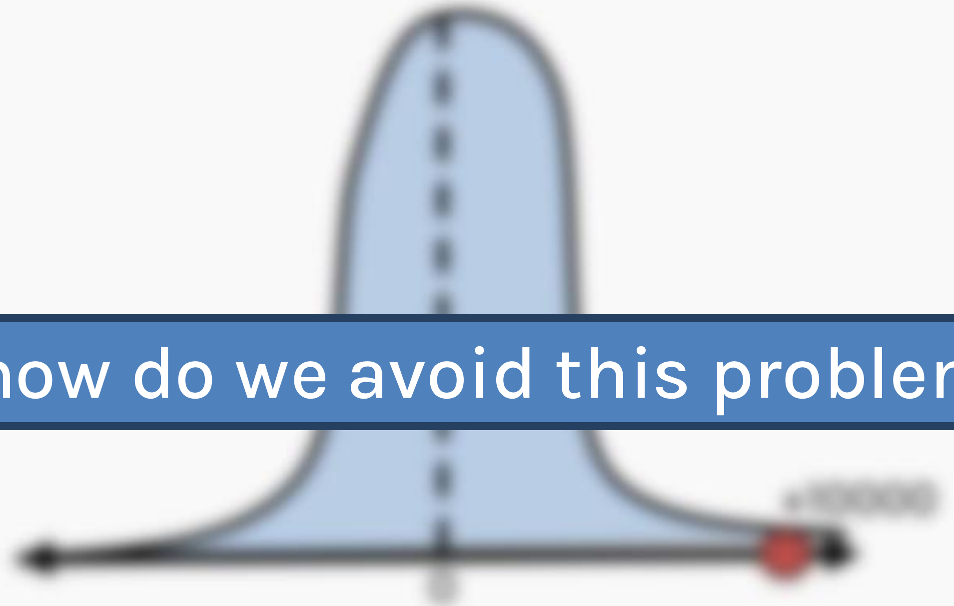
This is **unbounded** and could take up any **maximum valu**

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

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

QLoRA - Ingredient 2: Double Quantization

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



So, how do we avoid this problem?

This is unbounded and could take up any maximum value

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

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

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})

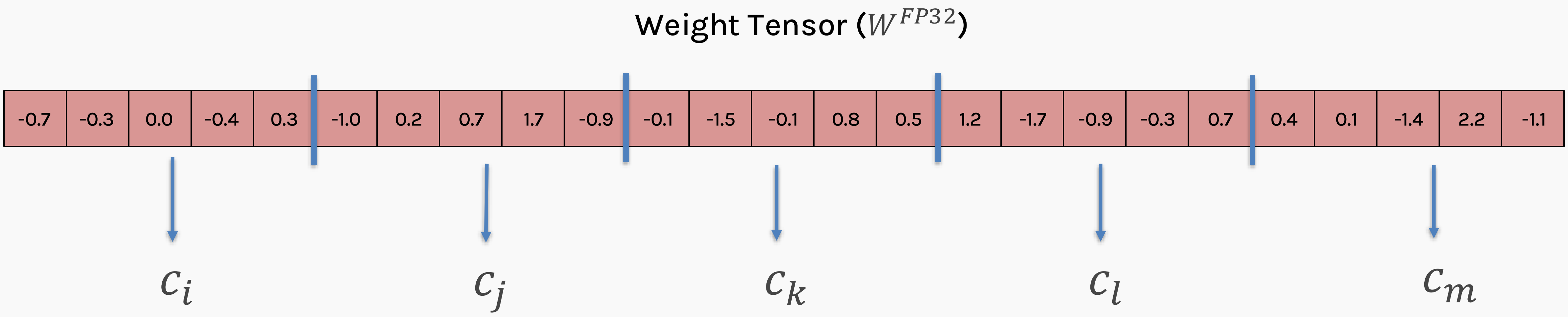
-0.7	-0.3	0.0	-0.4	0.3
-1.0	0.2	0.7	1.7	-0.9
-0.1	-1.5	-0.1	0.8	0.5
1.2	-1.7	-0.9	-0.3	0.7
0.4	0.1	-1.4	2.2	-1.1

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

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

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:

$c_1^{FP32} \longrightarrow$

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.

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

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

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.

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

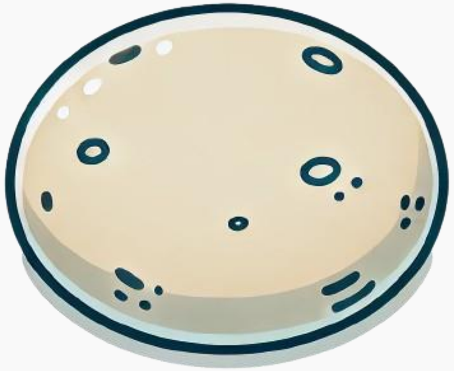
-0.7	-0.3	0.0	-0.4	0.3
-1.0	0.2	0.7	1.7	-0.9
-0.1	-1.5	-0.1	0.8	0.5
1.2	-1.7	-0.9	-0.3	0.7
0.4	0.1	-1.4	2.2	-1.1

25x4=100 bytes

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

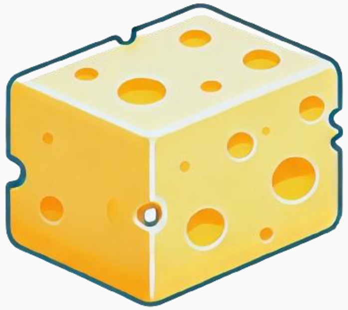
QLoRA – The Ingredients

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

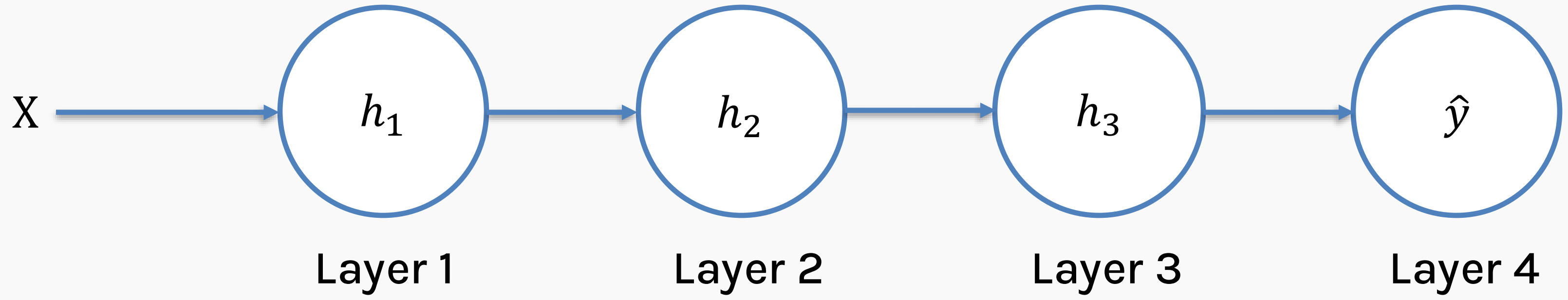
Running Out of Memory!

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

We use gradient checkpointing.

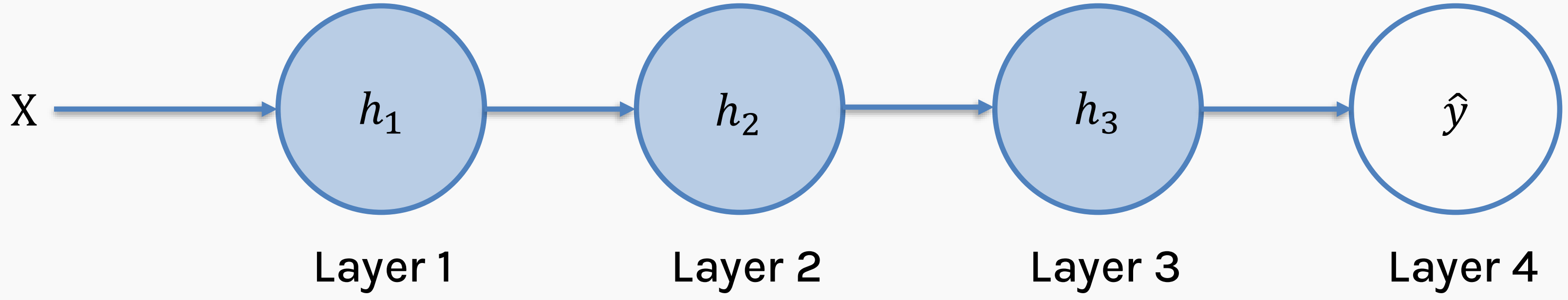
QLoRA – Ingredient 3

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.

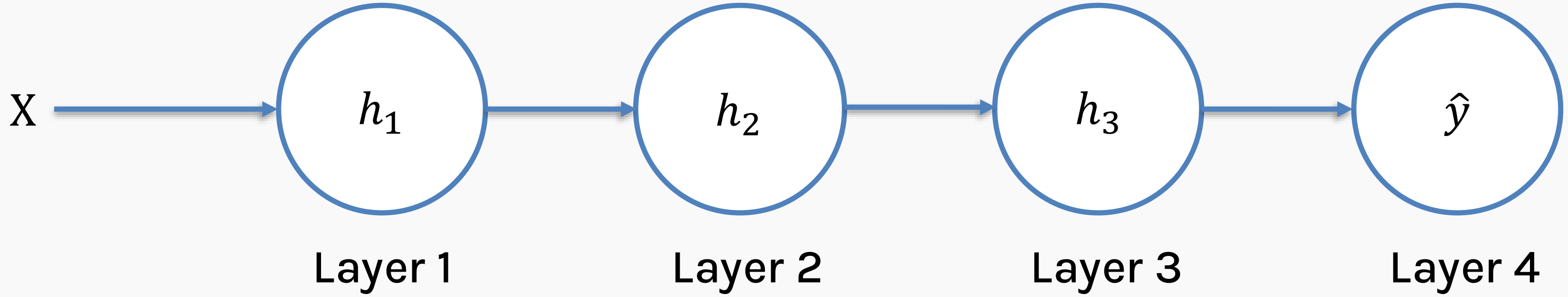
We don't need to store all the hidden states.

QLoRA – Ingredient 3

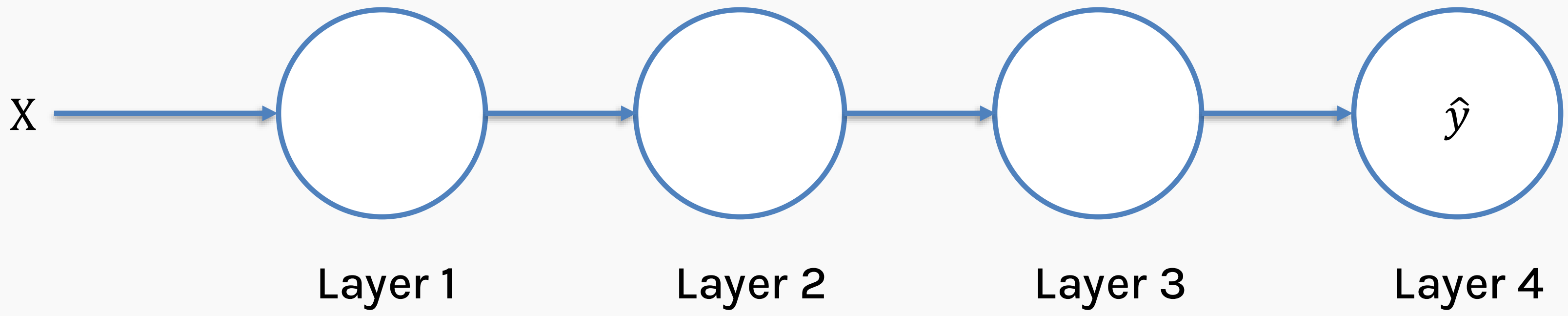
We only store in memory what is needed at the moment.

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!



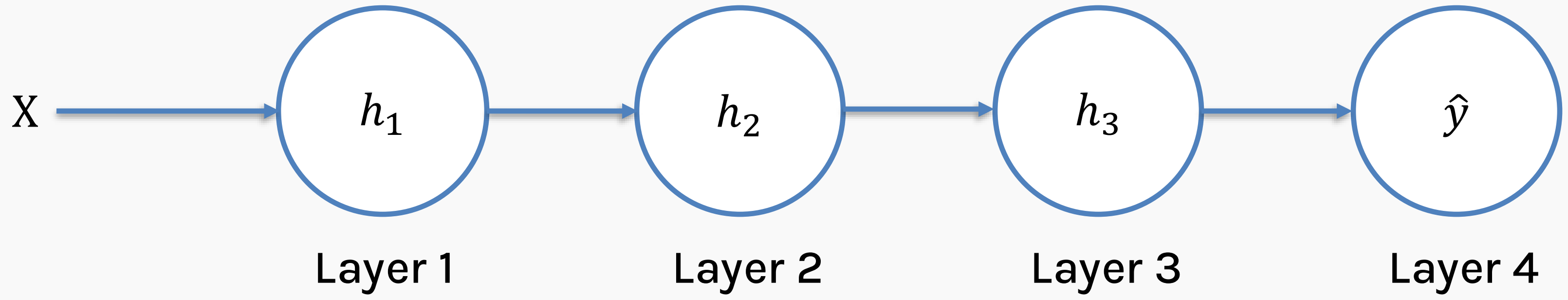
QLoRA – Ingredient 3



During backpropagation, we must recompute all the discarded activations.

To manage this, we introduce checkpoints in the middle.

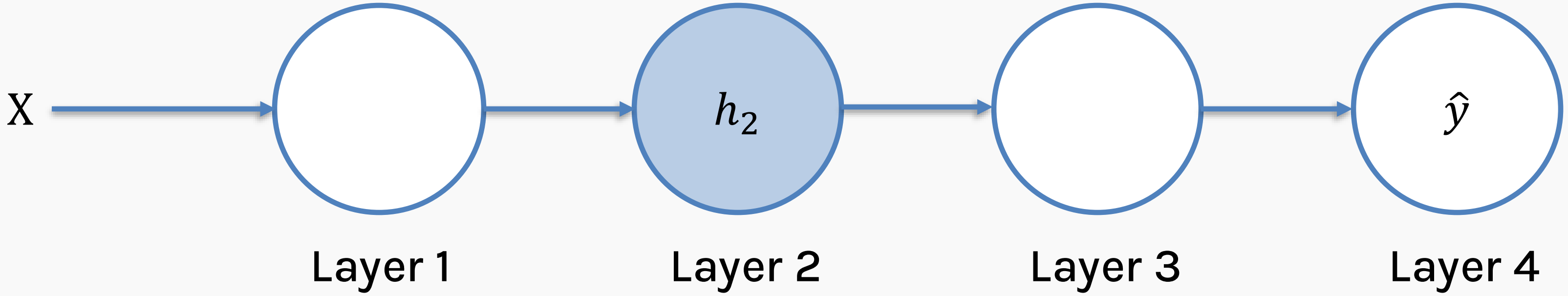
QLoRA – Ingredient 3



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 larger batches.

QLoRA - Ingredient 3

This is where our third ingredient comes in!

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 : Paged Optimizer

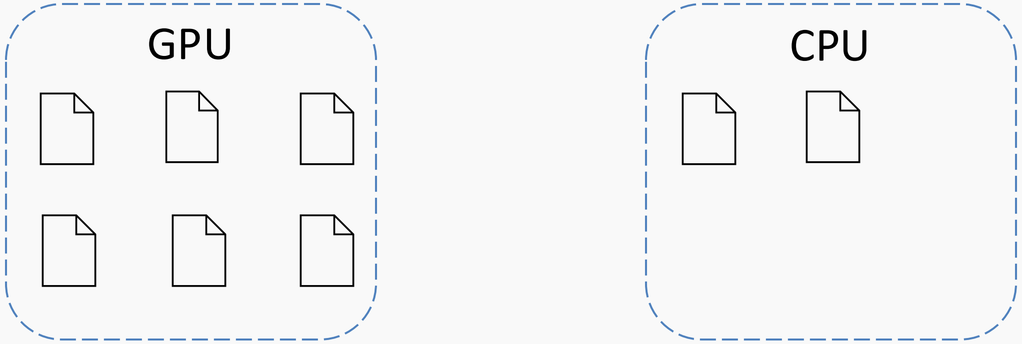
Paging is a memory management technique, where RAM is divided into fixed-size blocks called 'pages'

Paged Optimizer - Looping in your CPU

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.

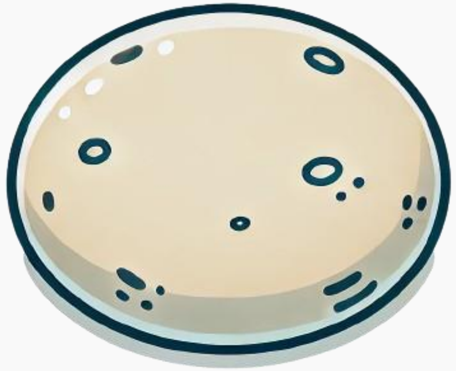
GPU Memory has space now.



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

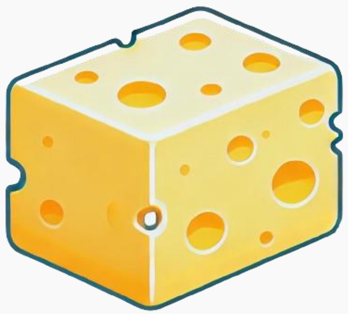
QLoRA – The Ingredients

We saw the 3 key ingredients needed to make QLoRA:



4-Bit NormalFloat

Double Quantization



Paged Optimizer

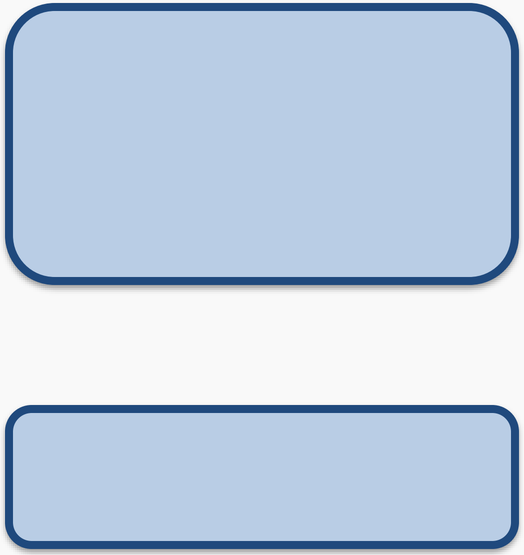
Let's bring it all together.

QLoRA – Putting it all together

Full Parameter Fine Tuning

Optimizer State (FP32)

Base Model



FP16

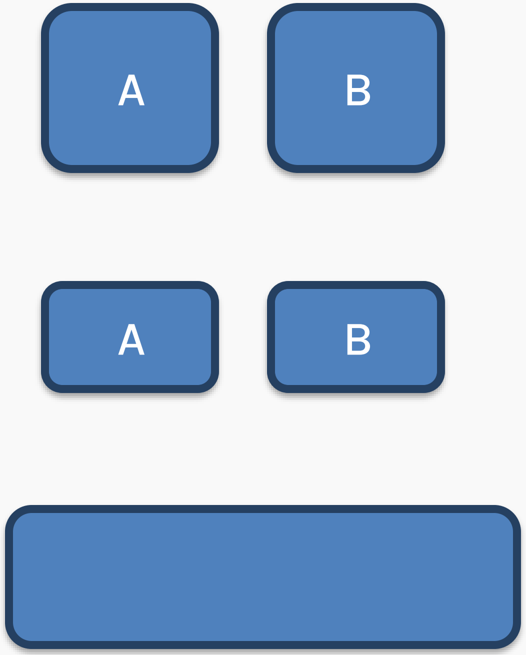
10B => 120GB

LoRA

Optimizer State (FP32)

LoRA Adapter (FP16)

Base Model



FP16

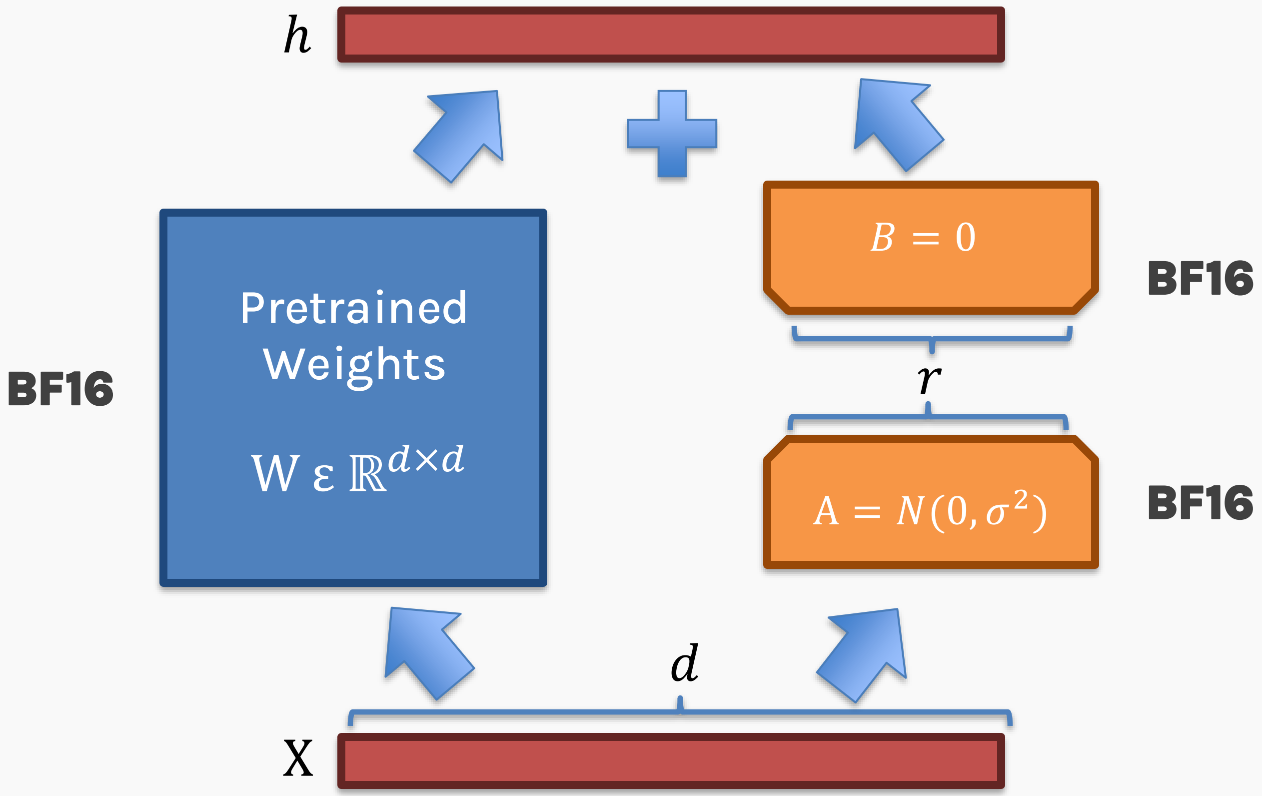
10B => ~40GB

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.

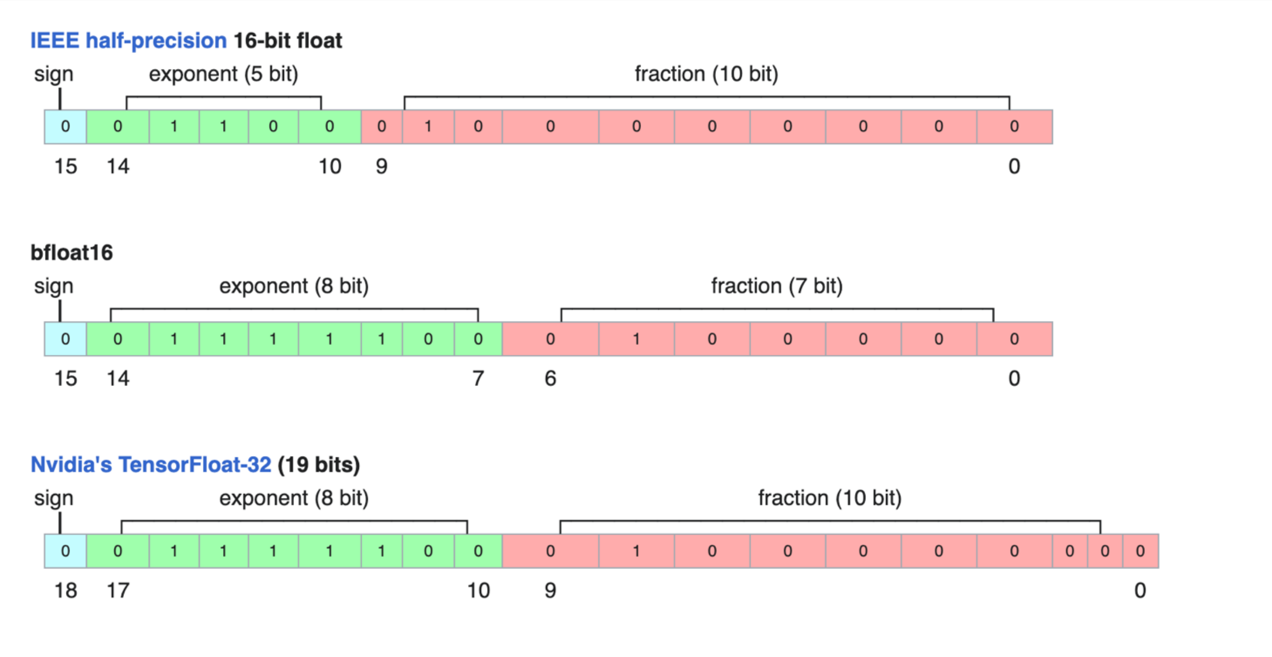


QLoRA – Putting it all together

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.

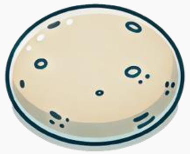


A 16-bit floating-point format keeping a wide range like 32-bit floats but with lower precision to speed up training.

Image Source

QLoRA – Putting it all together

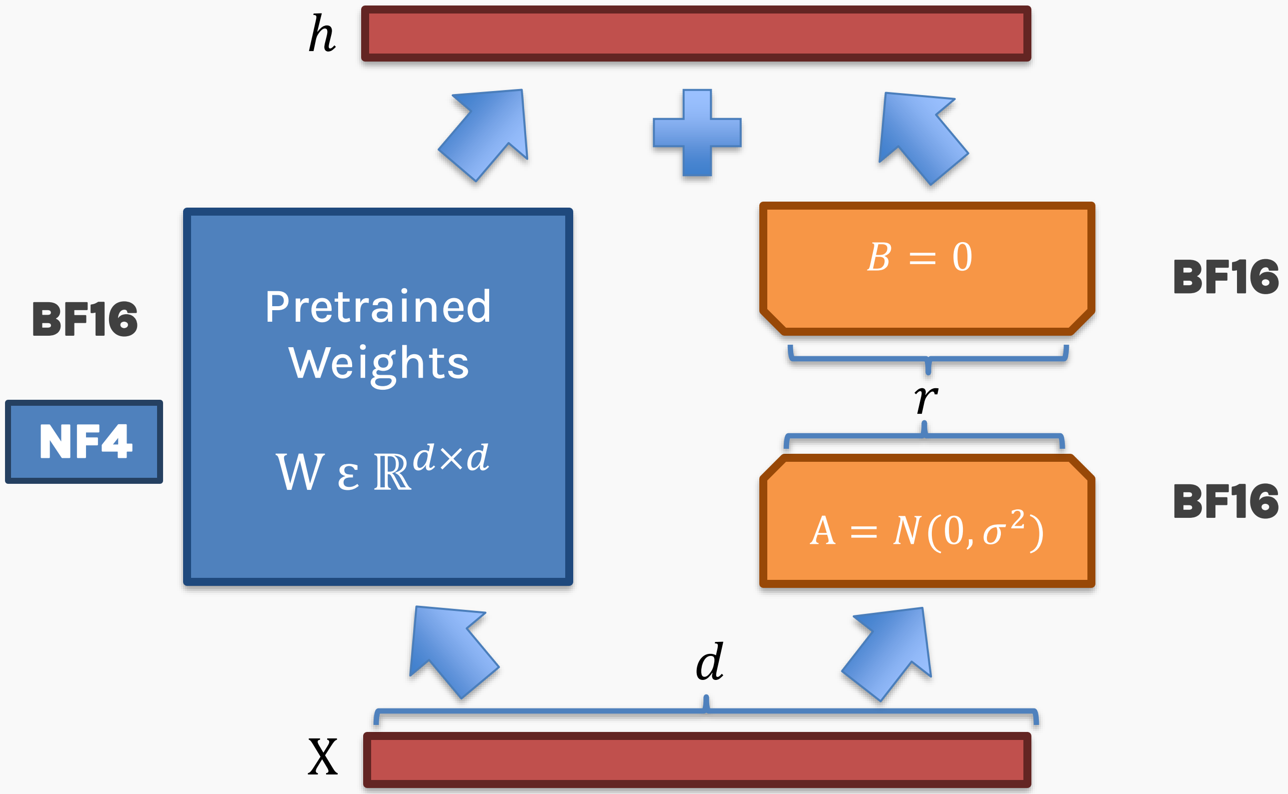
Ingredient 1:



4-Bit NormalFloat

Ingredient 2:

Double Quantization



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

We store W (original model parameters) as 4-Bit NormalFloat

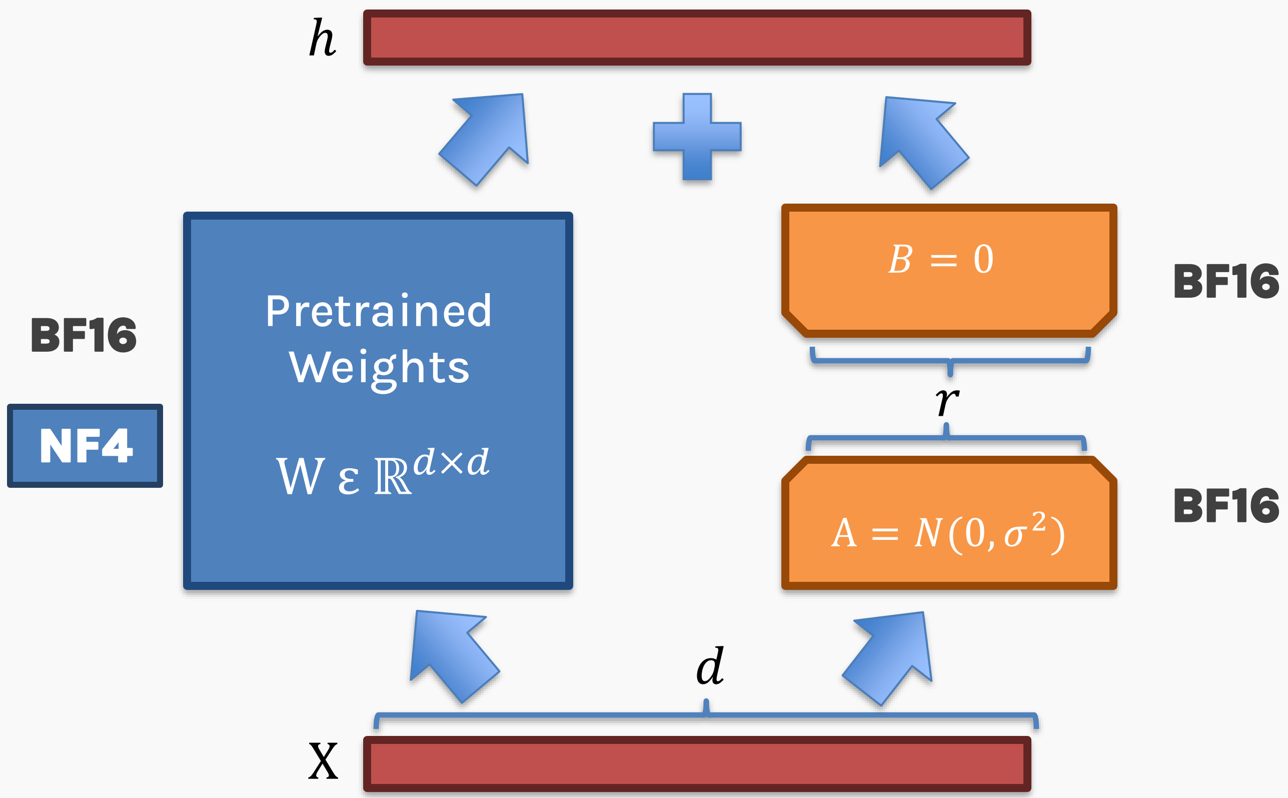
QLoRA – Putting it all together

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!

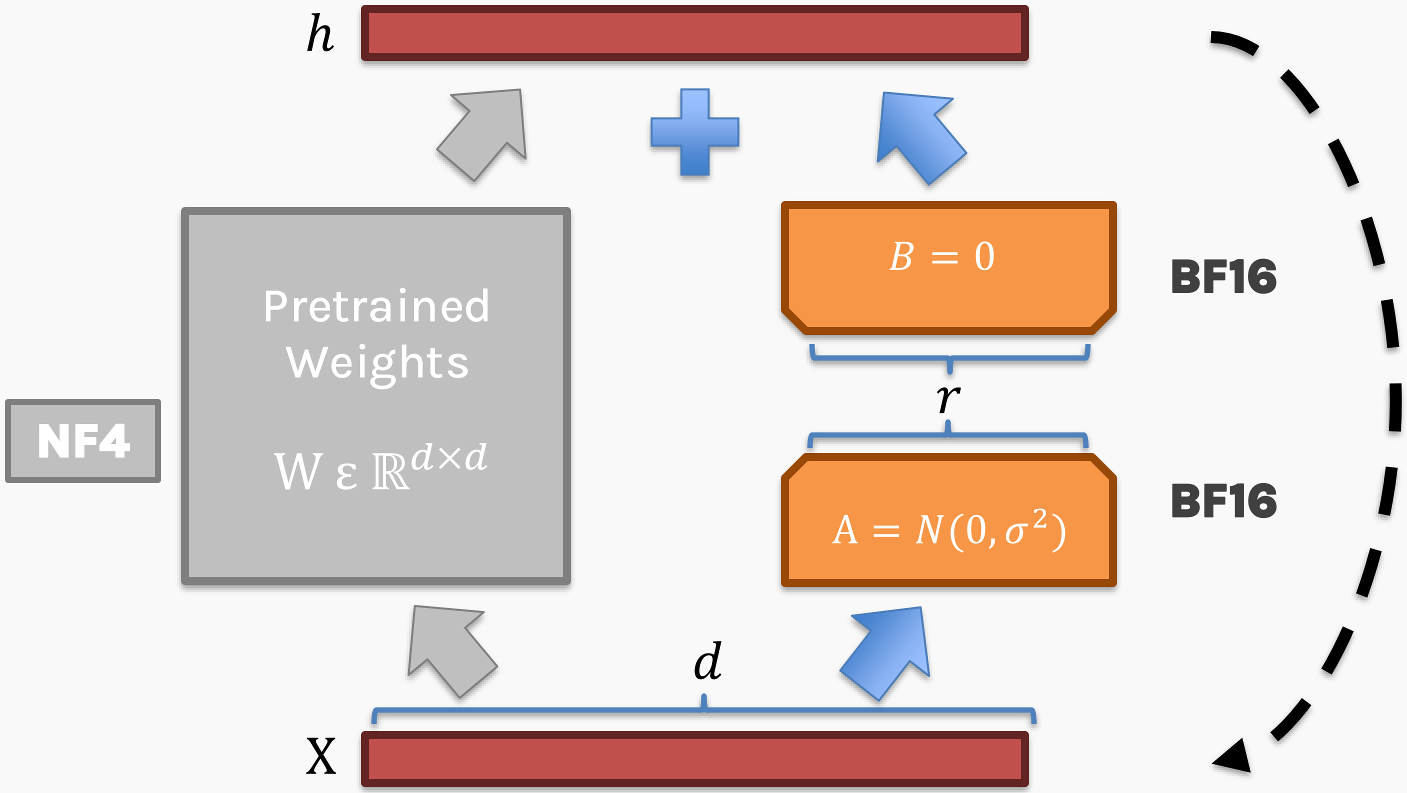


QLoRA – Putting it all together

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:

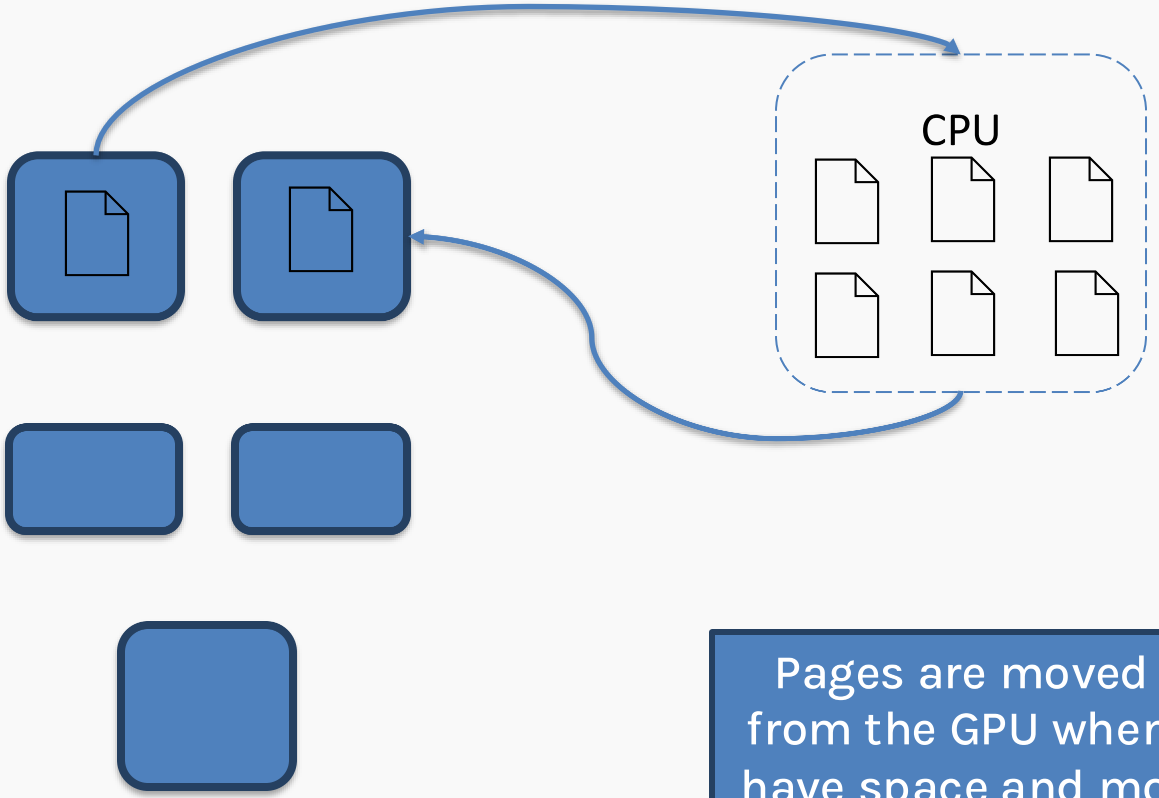
Paged Optimizer

QLoRA

Optimizer State (FP32)

LoRA Adapter (BF16)

Base Model



4bit NormalFloat

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.

10B => ~12GB

QLoRA – Putting it all together

Putting it mathematically,

Let's start with LoRA:

Initial LLM Weights

Scaling parameter

Decomposed matrices

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

Rank of B and A

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

QLoRA – Putting it all together

$$\mathrm{Y} = \mathrm{X}W_0 + \frac{\alpha}{r} XBA$$

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 $doubleDequant\left(c_1^{FP32}, c_2^{k-bit}, W_o^{NF4}\right) = dequant\left(dequant\left(\mathrm{c}_1^{\mathrm{FP32}}, \mathrm{c}_2^{\mathrm{k-bit}}\right), \mathrm{W}_o^{\mathrm{4bit}}\right)$
$= \mathrm{W}^{\mathrm{BF16}}$

Tutorial 10: Finetuning

In this tutorial, we will fine-tune a large language model (LLM) to respond like a cheese expert named Pavlos – who else?

The first step is to create question-answer pairs that reflect the knowledge and tone of Pavlos talking – using common phrases he uses. While this is usually done by a human, we'll use an LLM to help generate these pairs. We will use the dataset-creator container for that.
After preparing the dataset, we will fine-tune the LLM using QLORA.   For this, we'll use a tool called the Gemini finetuner to complete the process.

https://github.com/dlops-io/llm-finetuning/tree/main

After completing Finetuning:

THANK YOU