

Efficiency I: Parameter Efficient Finetuning

CSE 5525: Foundations of Speech and Natural Language
Processing

<https://shocheen.github.io/courses/cse-5525-fall-2025>



THE OHIO STATE UNIVERSITY

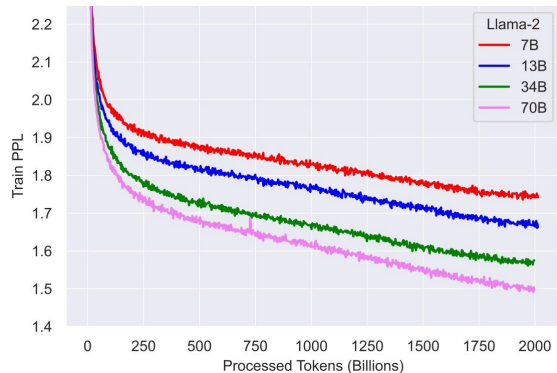
Logistics

- Homework 3.
- Project Proposal is graded
- Project mid-term report: due on Nov 12.
 - Will make a post on Canvas with everything that's required.

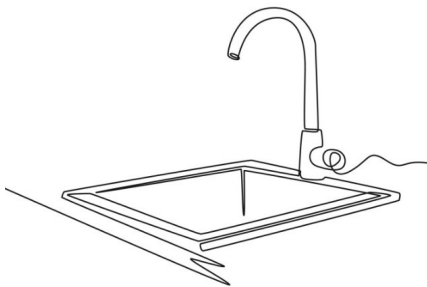
Last Class Recap: Benchmarking

- What is a benchmark?
- Quality of good benchmarks
- Benchmark and metrics, evaluation (closed and open-ended evaluation)
- Current evaluations of LLMs
- Issues with benchmarking

Current evaluation of LLM



Perplexity



Everything

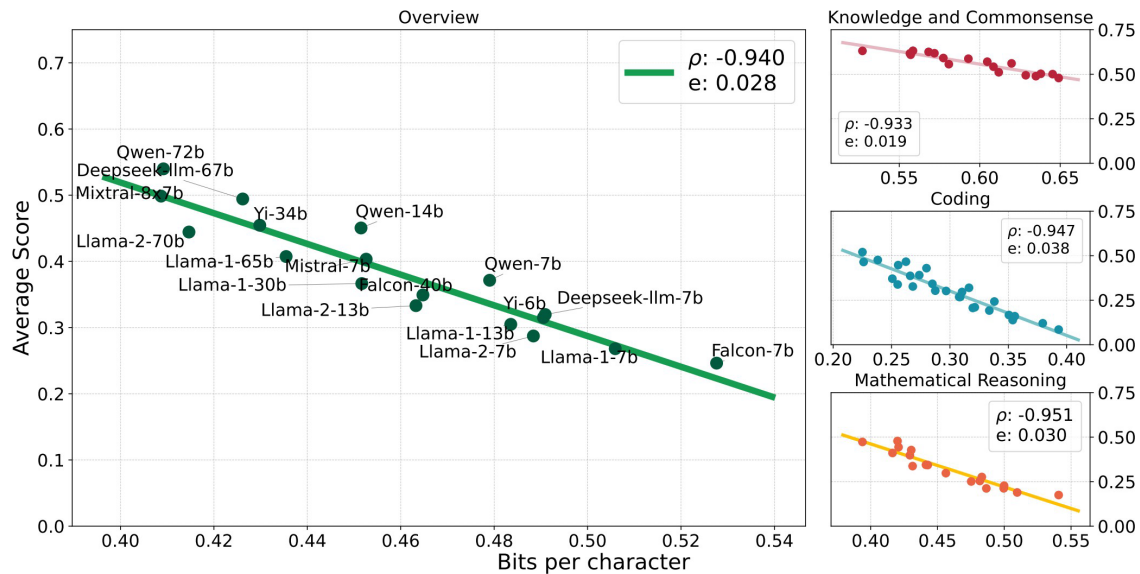


Arena-like

pretraining

finetuned

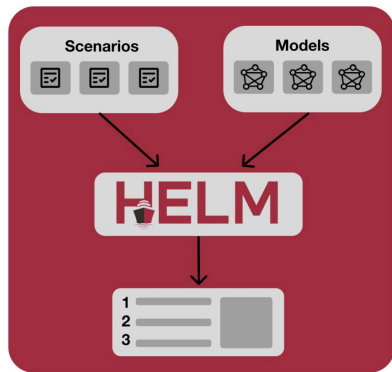
Perplexity



Perplexity is highly correlated with downstream performance

Everything: HELM, open-LLM leaderboard, and others

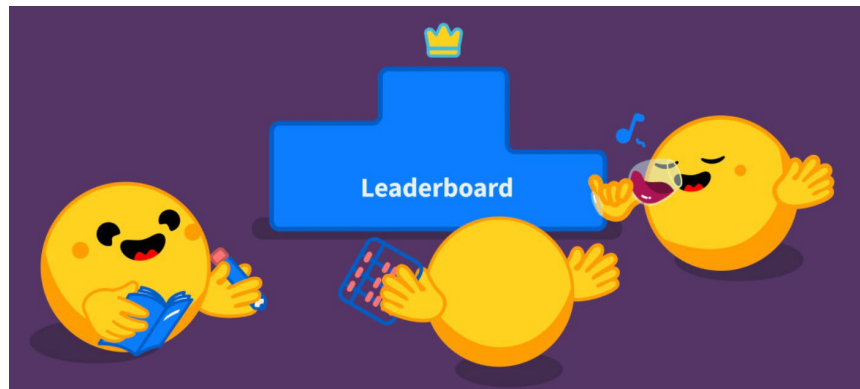
Holistic evaluation of language models (HELM)



Model	Mean win rate
GPT-4 (0613)	0.962
GPT-4 Turbo (1106 preview)	0.834
Palmyra X V3 (72B)	0.821
Palmyra X V2 (33B)	0.783
PaLM-2 (Unicorn)	0.776
Yi (34B)	0.772

SEE MORE

Huggingface open LLM leaderboard



collect many automatically evaluable benchmarks,
evaluate across them

What are common LM datasets?

- What do these benchmarks evaluate on?

- A huge mix of things!

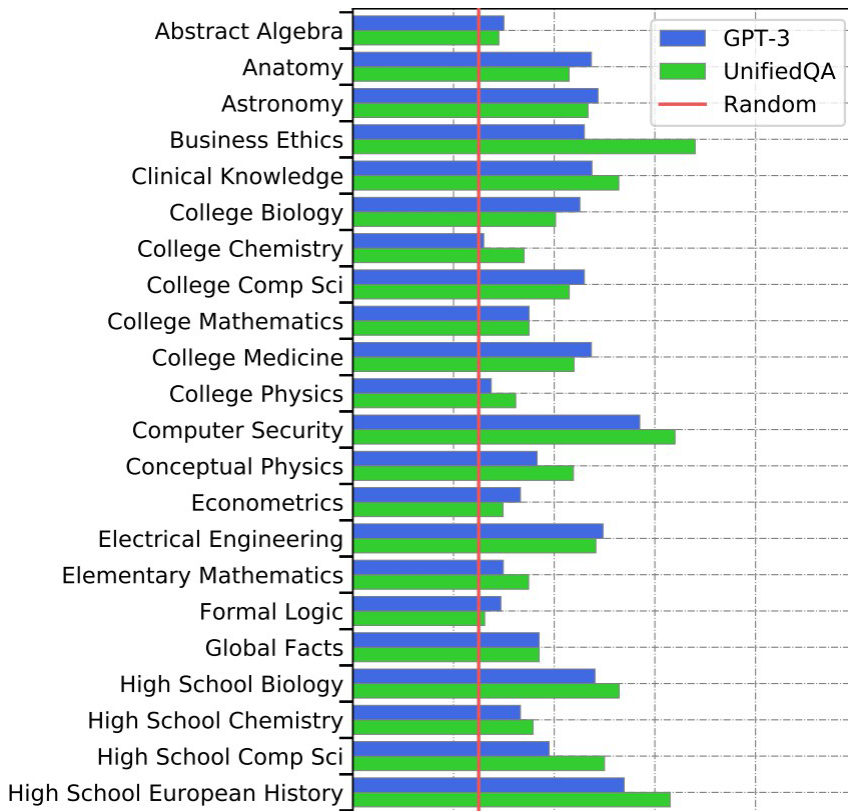
Scenario	Task	What	Who
NarrativeQA narrative_qa	short-answer question answering	passages are books and movie scripts, questions are unknown	annotators from summaries
NaturalQuestions (closed-book) natural_qa_closedbook	short-answer question answering	passages from Wikipedia, questions from search queries	web users
NaturalQuestions (open-book) natural_qa_openbook_longans	short-answer question answering	passages from Wikipedia, questions from search queries	web users
OpenbookQA openbookqa	multiple-choice question answering	elementary science	Amazon Mechanical Turk workers
MMLU (Massive Multitask Language Understanding) mmlu	multiple-choice question answering	math, science, history, etc.	various online sources
GSM8K (Grade School Math) gsm	numeric answer question answering	grade school math word problems	contractors on Upwork and Surge AI
MATH math_chain_of_thought	numeric answer question answering	math competitions (AMC, AIME, etc.)	problem setters
LegalBench legalbench	multiple-choice question answering	public legal and administrative documents, manually constructed questions	lawyers
MedQA med_qa	multiple-choice question answering	US medical licensing exams	problem setters
WMT 2014 wmt_14	machine translation	multilingual sentences	Europarl, news, Common Crawl, etc.

MMLU

Massive Multitask Language Understanding (MMLU)

[[Hendrycks et al., 2021](#)]

A benchmark for measuring LM performance on 57 diverse *knowledge intensive* tasks



Examples from MMLU

Astronomy

What is true for a type-Ia supernova?

- A. This type occurs in binary systems.
- B. This type occurs in young galaxies.
- C. This type produces gamma-ray bursts.
- D. This type produces high amounts of X-rays.

Answer: A

High School Biology

In a population of giraffes, an environmental change occurs that favors individuals that are tallest. As a result, more of the taller individuals are able to obtain nutrients and survive to pass along their genetic information. This is an example of

- A. directional selection.
- B. stabilizing selection.
- C. sexual selection.
- D. disruptive selection

Answer: A

Other capabilities: code

Nice feature of code: evaluate
vs test cases

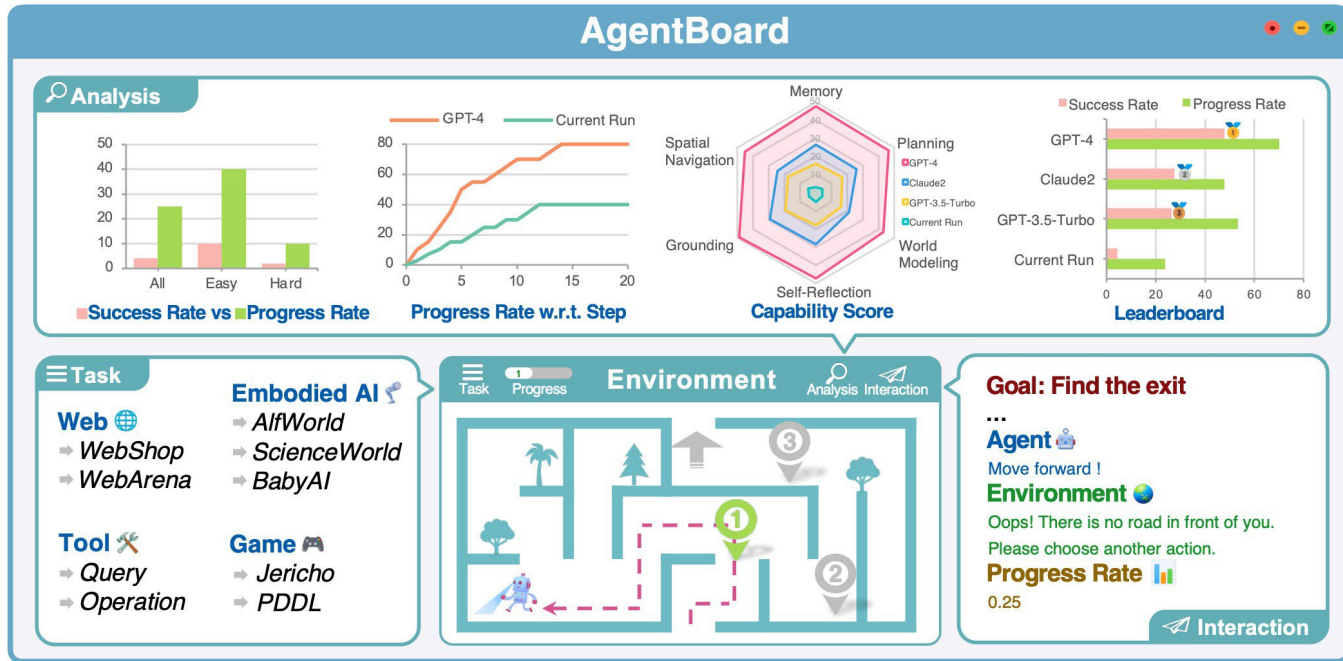
Metric: Pass@1 (Pass @ k
means one of k outputs pass)

GPT4: ~67%

```
def solution(lst):  
    """Given a non-empty list of integers, return the sum of all of the odd elements  
    that are in even positions.  
  
    Examples  
    solution([5, 8, 7, 1]) ==>12  
    solution([3, 3, 3, 3, 3]) ==>9  
    solution([30, 13, 24, 321]) ==>0  
    """  
    return sum(lst[i] for i in range(0, len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

```
def encode_cyclic(s: str):  
    """  
    returns encoded string by cycling groups of three characters.  
    """  
    # split string to groups. Each of length 3.  
    groups = [s[(3 * i):min((3 * i + 3), len(s))]] for i in range((len(s) + 2) // 3)]  
    # cycle elements in each group. Unless group has fewer elements than 3.  
    groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups]  
    return "".join(groups)  
  
def decode_cyclic(s: str):  
    """  
    takes as input string encoded with encode_cyclic function. Returns decoded string.  
    """  
    # split string to groups. Each of length 3.  
    groups = [s[(3 * i):min((3 * i + 3), len(s))]] for i in range((len(s) + 2) // 3)]  
    # cycle elements in each group.  
    groups = [(group[-1] + group[:-1]) if len(group) == 3 else group for group in groups]  
    return "".join(groups)
```

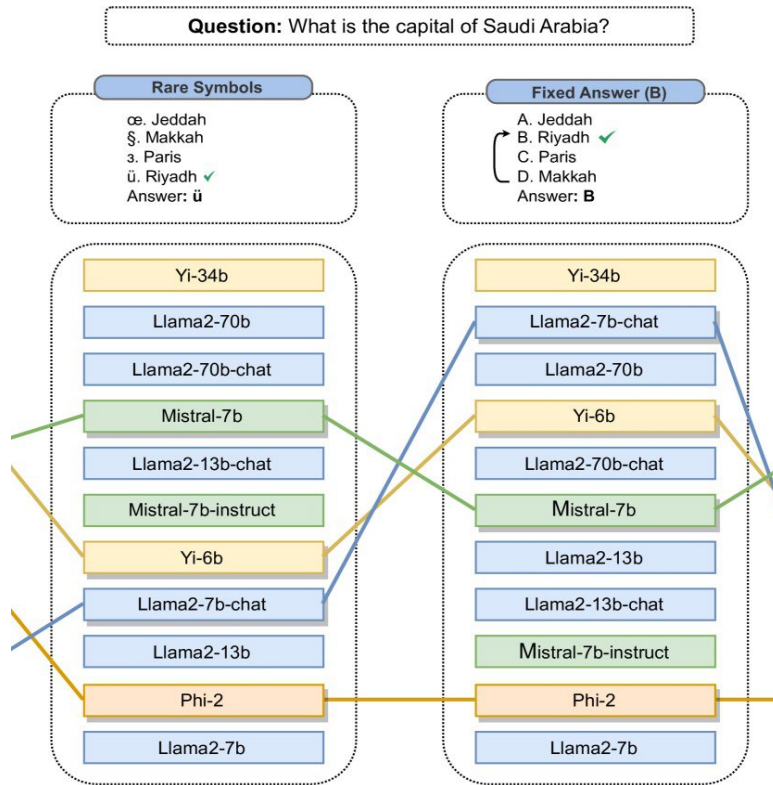
Other capabilities: agents



- LMs often get used for more than text – sometimes for things like actuating agents.
- Challenge:** evaluation need to be done in sandbox environments

Issues and challenges with evaluation

Consistency issues

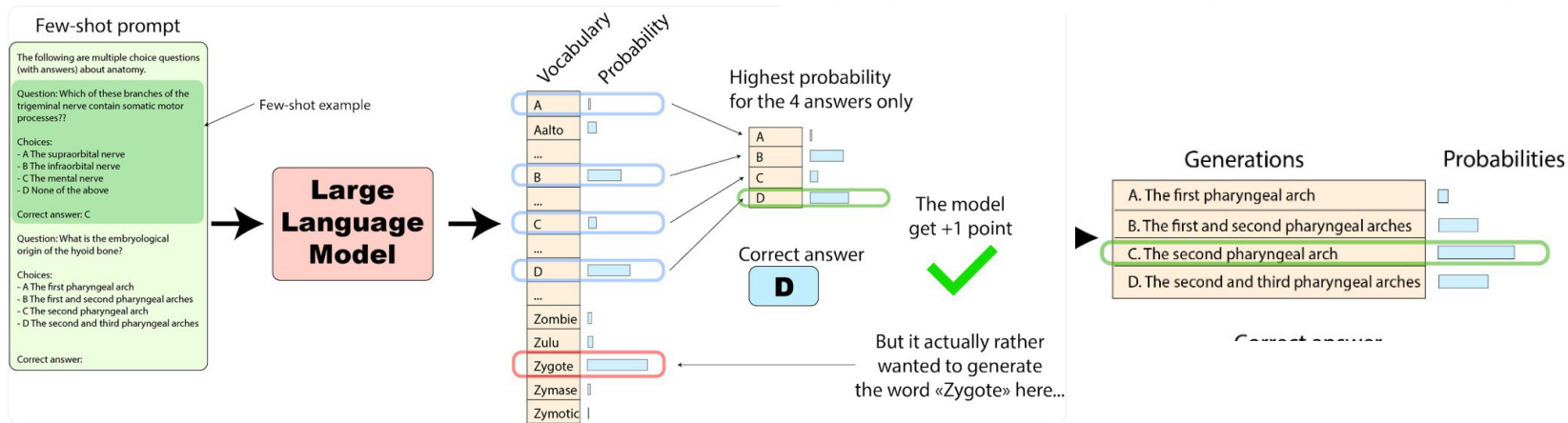


Consistency issues: MMLU

- MMLU has many implementations:

- Different prompts
- Different generations
 - Most likely valid choice
 - Probability of gen. answer
 - Most likely choice

	MMLU (HELM)	MMLU (Harness)	MMLU (Original)
llama-65b	0.637	0.488	0.636
tiiuae/falcon-40b	0.571	0.527	0.558
llama-30b	0.583	0.457	0.584
EleutherAI/gpt-neox-20b	0.256	0.333	0.262
llama-13b	0.471	0.377	0.47
llama-7b	0.339	0.342	0.351
tiiuae/falcon-7b	0.278	0.35	0.254



Contamination and overfitting issues



Horace He
@cHHillee

I suspect GPT-4's performance is influenced by data contamination, at least on Codeforces.

Of the easiest problems on Codeforces, it solved 10/10 pre-2021 problems and 0/10 recent problems.

This strongly points to contamination.

1/4

g's Race	implementation, math	🚀 ⭐	greedy, implementation	🚀 ⭐
nd Chocolate	implementation, math	🚀 ⭐	at?	🚀 ⭐
triangle!	brute force, geometry, math	🚀 ⭐	Actions	🚀 ⭐
	greedy, implementation, math	🚀 ⭐	Interview Problem	🚀 ⭐
			brute force, implementation, strings	

...



Susan Zhang ✓
@suchenzang

I think Phi-1.5 trained on the benchmarks. Particularly, GSM8K.



Susan Zhang ✓ @suchenzang · Sep 12

Let's take [github.com/openai/grade-s...](https://github.com/openai/grade-school-math)

If you truncate and feed this question into Phi-1.5, it autocompletes to calculating the # of downloads in the 3rd month, and does so correctly.

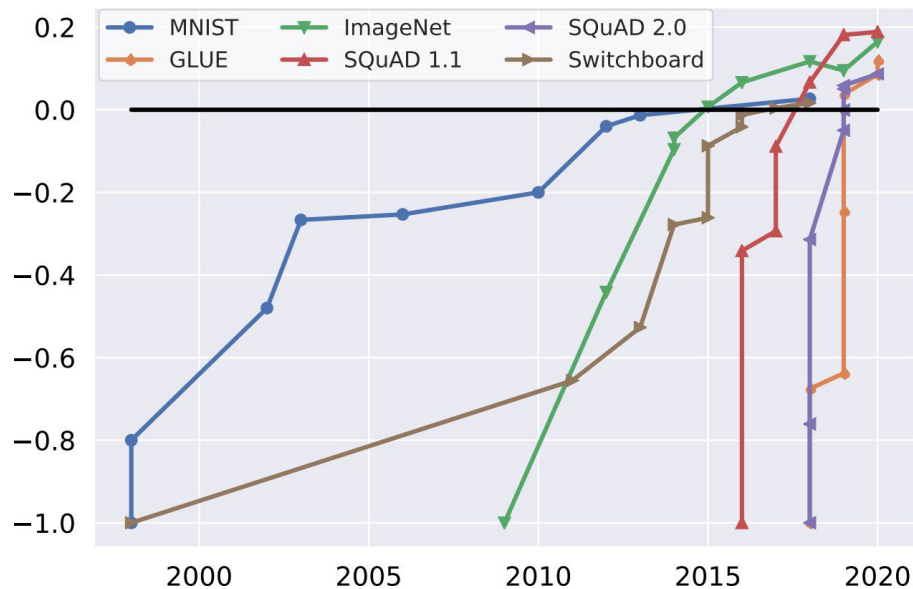
Change the number a bit, and it answers correctly as well.

1/ 🤖



Closed models + pretraining: hard to know that benchmarks are truly 'new'

Overfitting issue

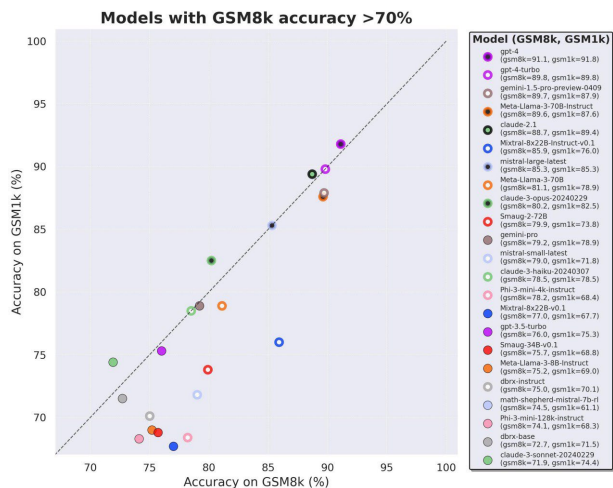


Reach “human-level” performance too quickly

Alleviating overfitting

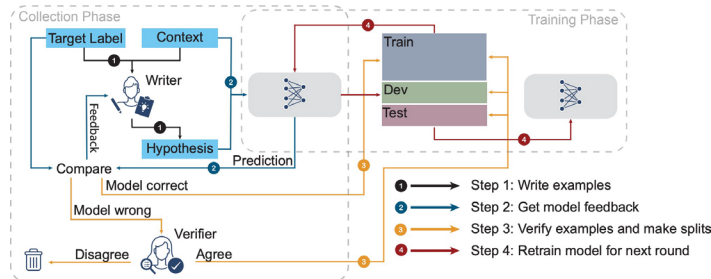
Private test set

- Control the number of times one can see the test set



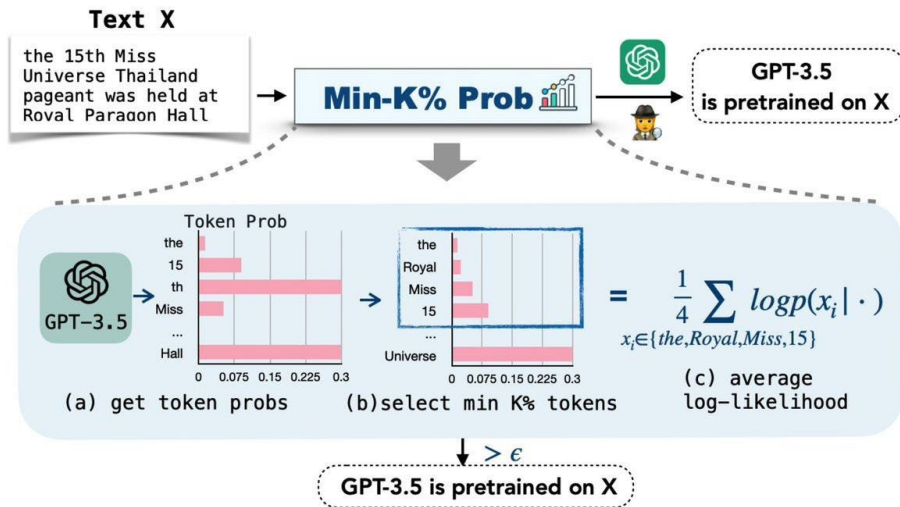
Dynamic test set

- Constantly change the inputs



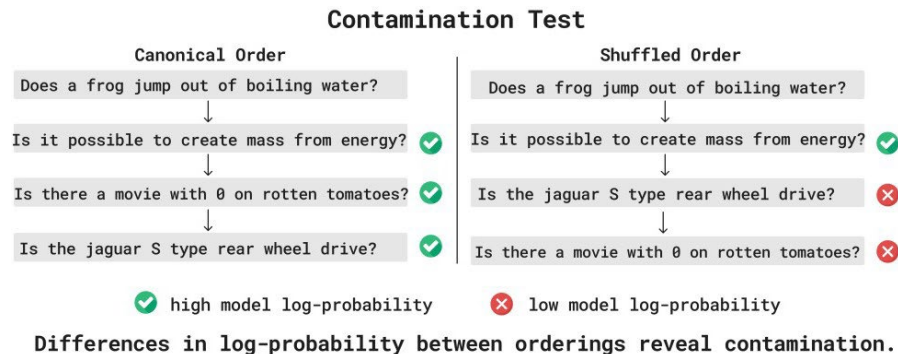
Alleviating contamination: detectors

Min-k-prob



- Detect if models trained on a benchmark by checking if probabilities are 'too high' (what is too high?). Often heuristic.

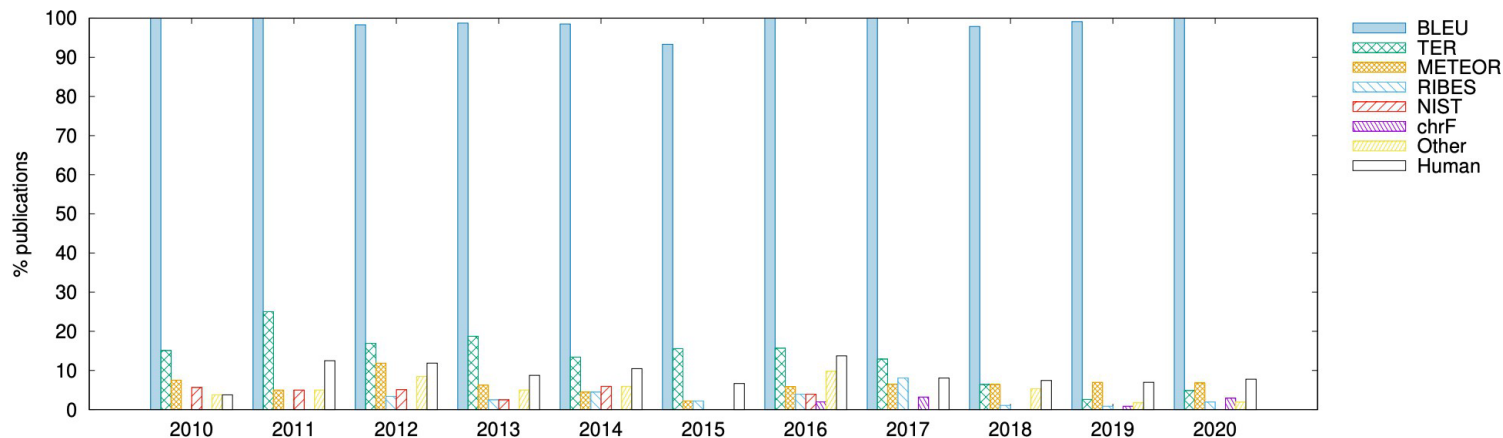
Exchangeability test



- Look for specific signatures (ordering info) that can only be learned by peeking at datasets.

The challenges of challenges: status quo issue

- Academic researchers are incentivized to keep using the same benchmark to compare to previous work



- 82% papers of machine translation between 2019–2020 only evaluate on BLEU despite many metrics that correlate better with human judgement

Reduce single metric issue

- Performance is not all we care about:
 - Computational efficiency
 - Biases
 - ...
- Taking averages for aggregation is unfair for minorized groups
- Different preferences for different people

Evaluation: Takeaways

- Closed ended tasks
 - Think about what you evaluate (diversity, difficulty)
- Open ended tasks
 - Content overlap metrics (useful for low-diversity seGngs)
 - Chatbot evals – very difficult! Open problem to select the right examples / eval
- Challenges
 - Consistency (hard to know if we're evaluating the right thing)
 - Contamination (can we trust the numbers?)
 - Biases
- In many cases, the best judge of output quality is **YOU!**
 - **Look at your model generations. Don't just rely on numbers!**

Parameter Efficient Finetuning

Instruction-tuning

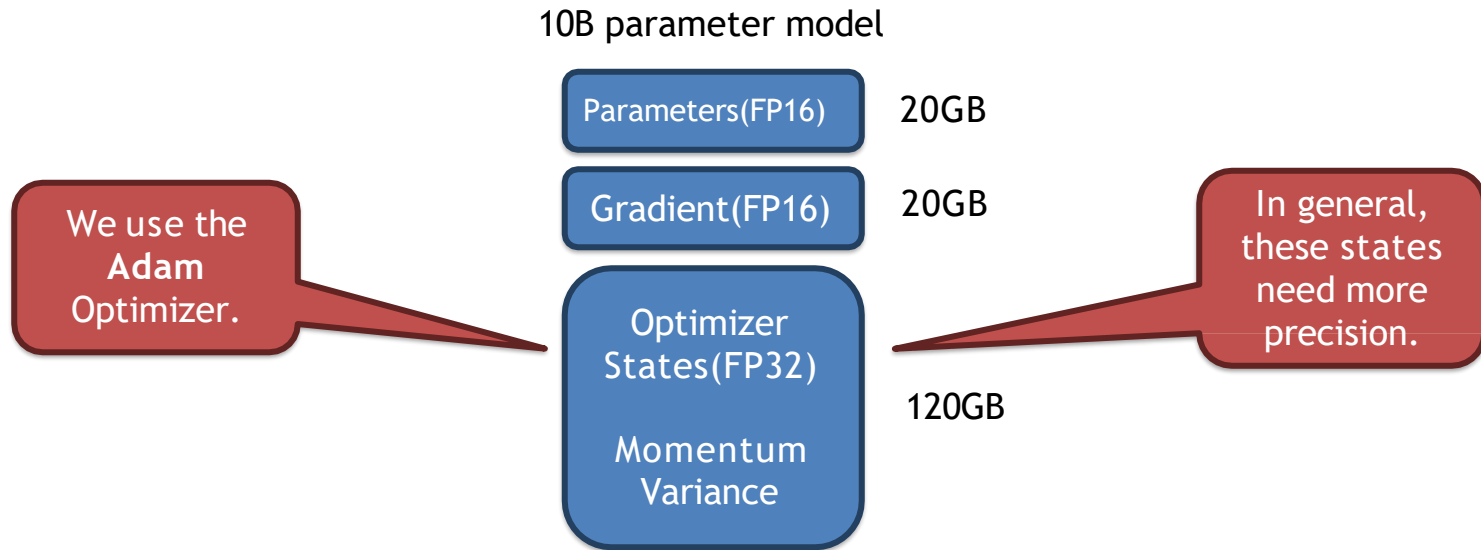
Let's take a fine-tuning example.

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

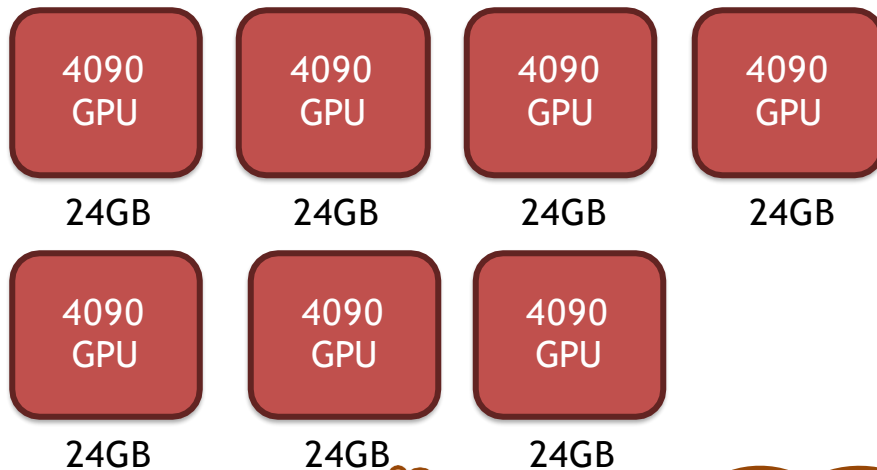
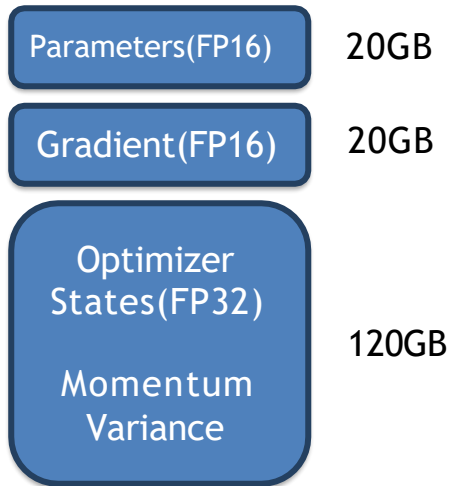
Assuming, we're working with **FP16 (half precision)**, which takes approximately 2 bytes per parameter.



Instruction-tuning

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

10B parameter model

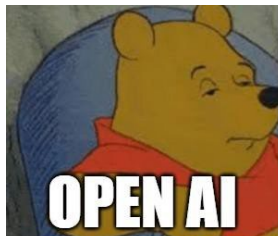


This model needs at least 7 decent GPU's to finetune.

Instruction-tuning

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

So, what can we
do?



Pre-training
using
thousands of GPUS



Use
Parameter Efficient
Finetuning

Outline

- Instruction-tuning
 - Full Parameter
 - PEFT
- LoRA
- Quantized LoRA

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

additive

selective

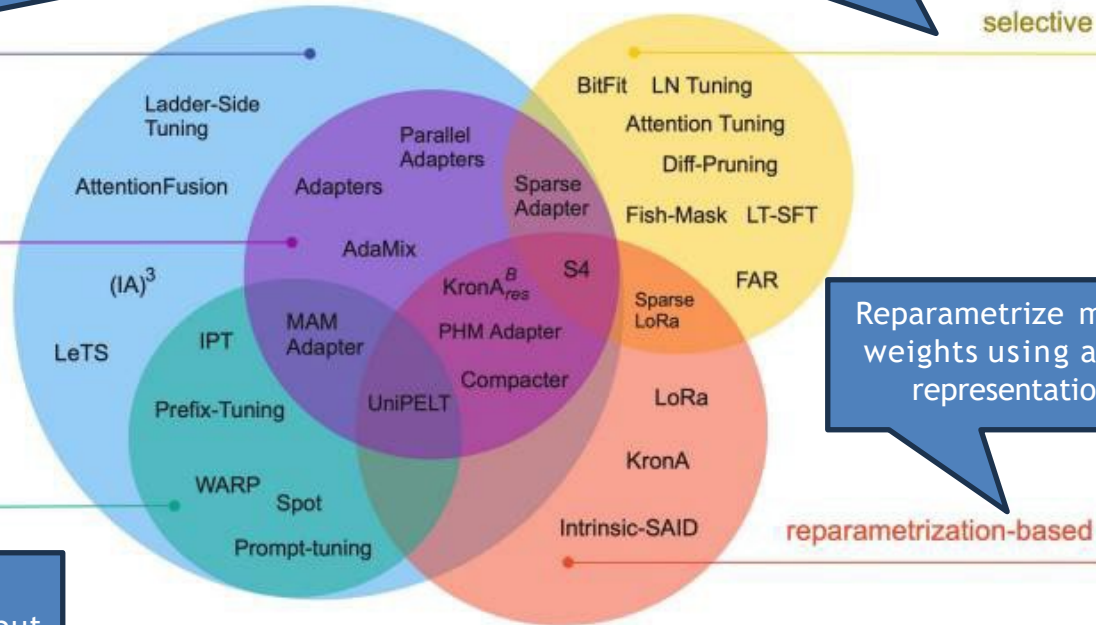
adapters

Add new trainable layers to the architecture called 'Adapters'

Reparametrize model weights using a new representation

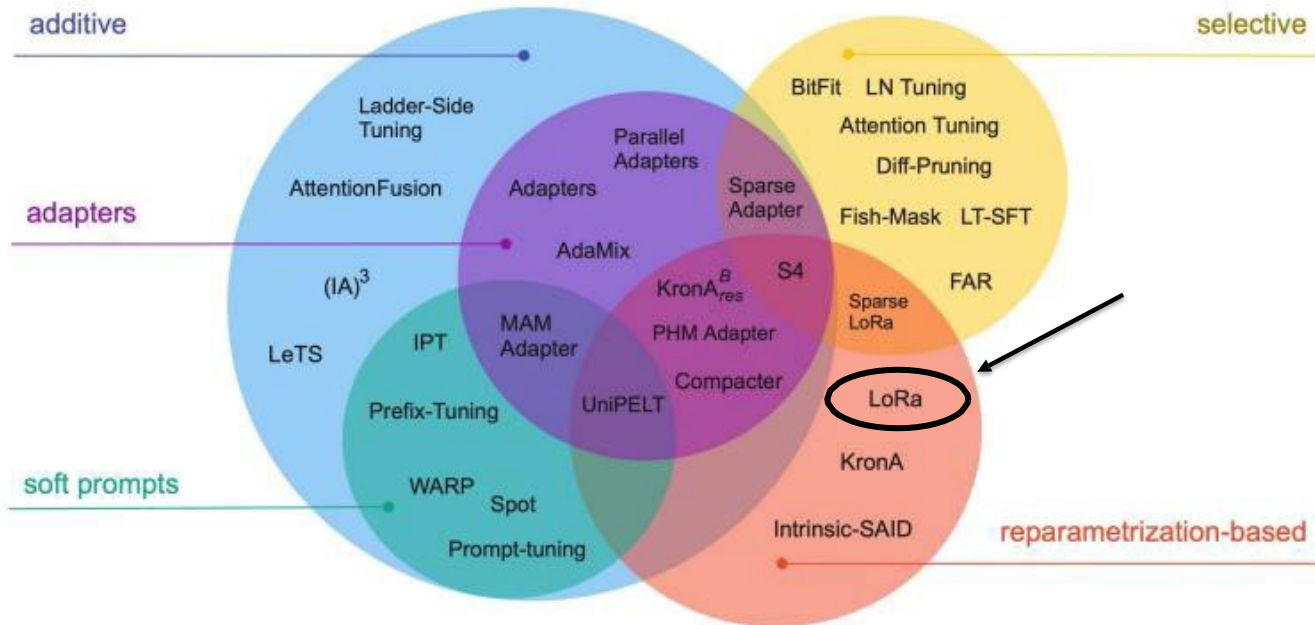
soft prompts

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



Instruction-tuning (PEFT)

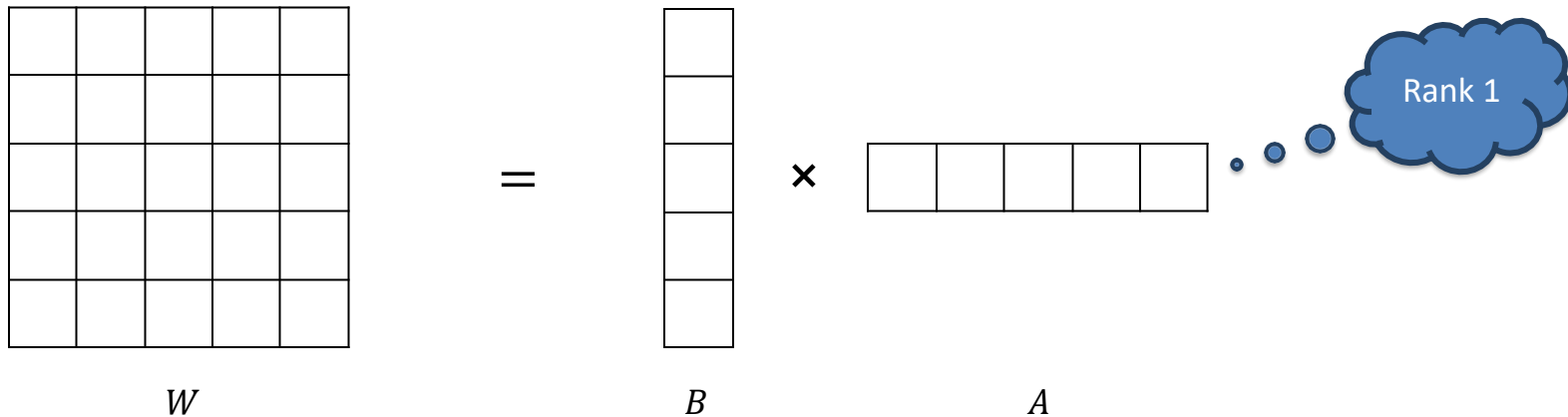
There are a lot of techniques. We're interested in [LoRA](#), which is one of the most popular.



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

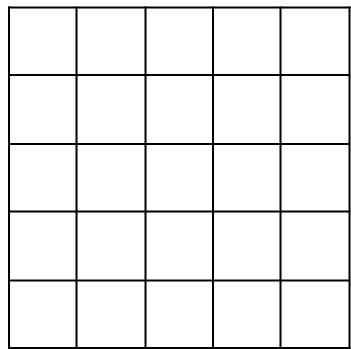
LoRA - Intuition

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



W

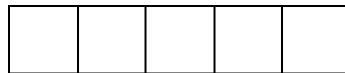
25
elements

=



B

\times

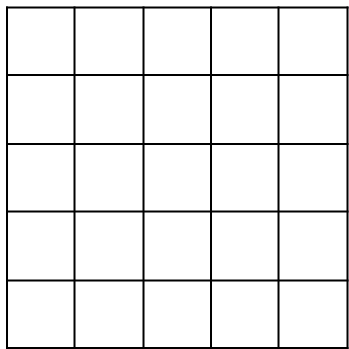


A

10
elements

LoRA - Intuition

We can even increase the rank to get better performance.



W

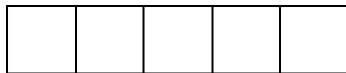
25
elements

=



B

\times



A

10
elements

LoRA - Working

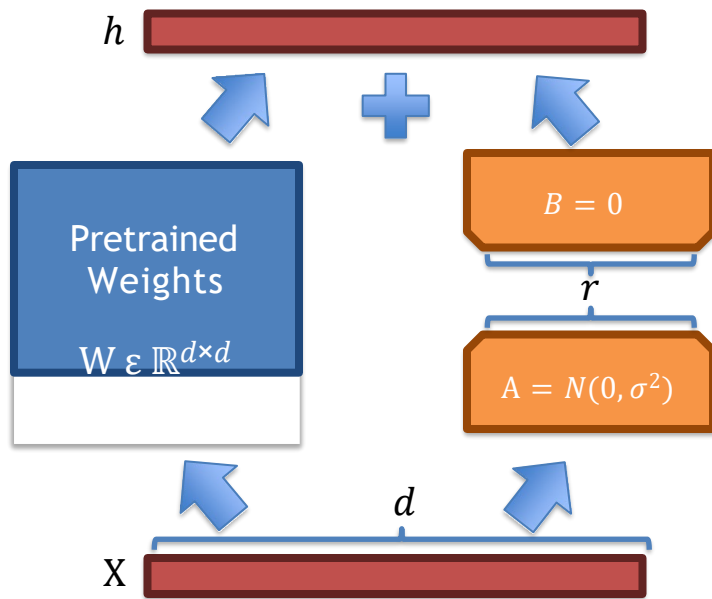
Now, we use the same concept of matrix decomposition while finetuning an LLM.

The diagram illustrates the LoRA equation: $W_0 + \Delta W = W_0 + \frac{\alpha}{r} BA$. It features five blue callout boxes with pointers to specific parts of the equation:

- Initial LLM Weights**: Points to W_0 on the left side of the equation.
- Update matrix**: Points to ΔW in the first part of the equation.
- Scaling parameter**: Points to the fraction $\frac{\alpha}{r}$.
- Decomposed matrices**: Points to the product BA .
- Rank of BA** : Points to the denominator r .

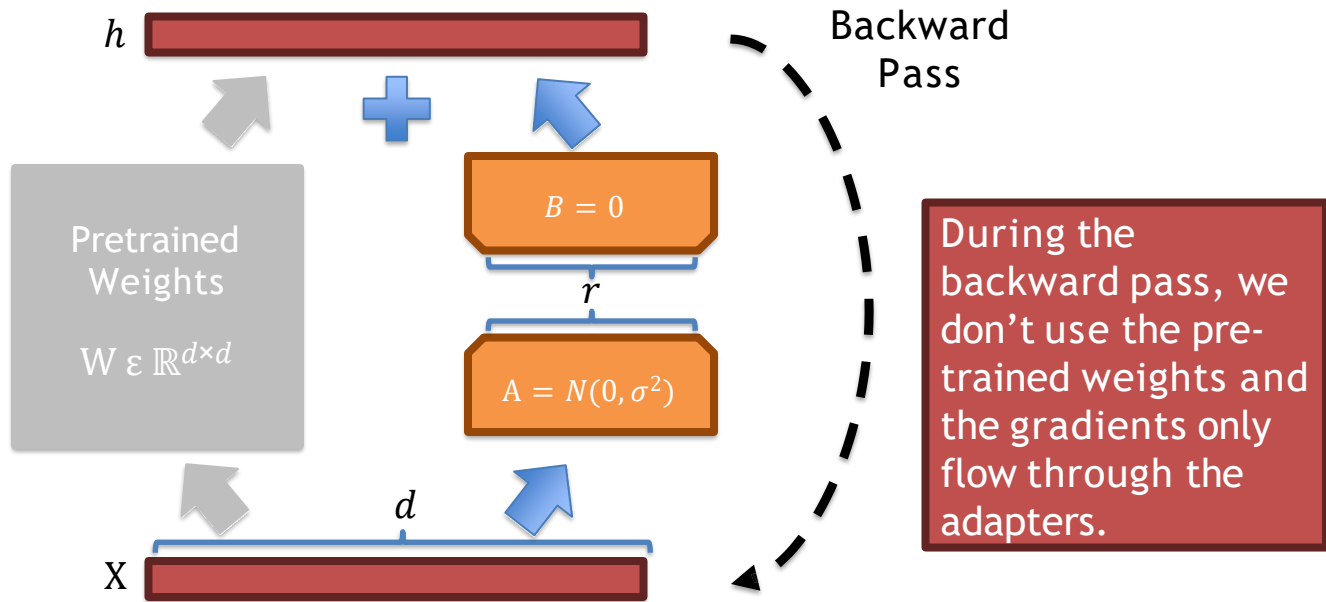
Remember, we are decomposing the update matrix (ΔW), and not the original weights W_0 .

LoRA - Working



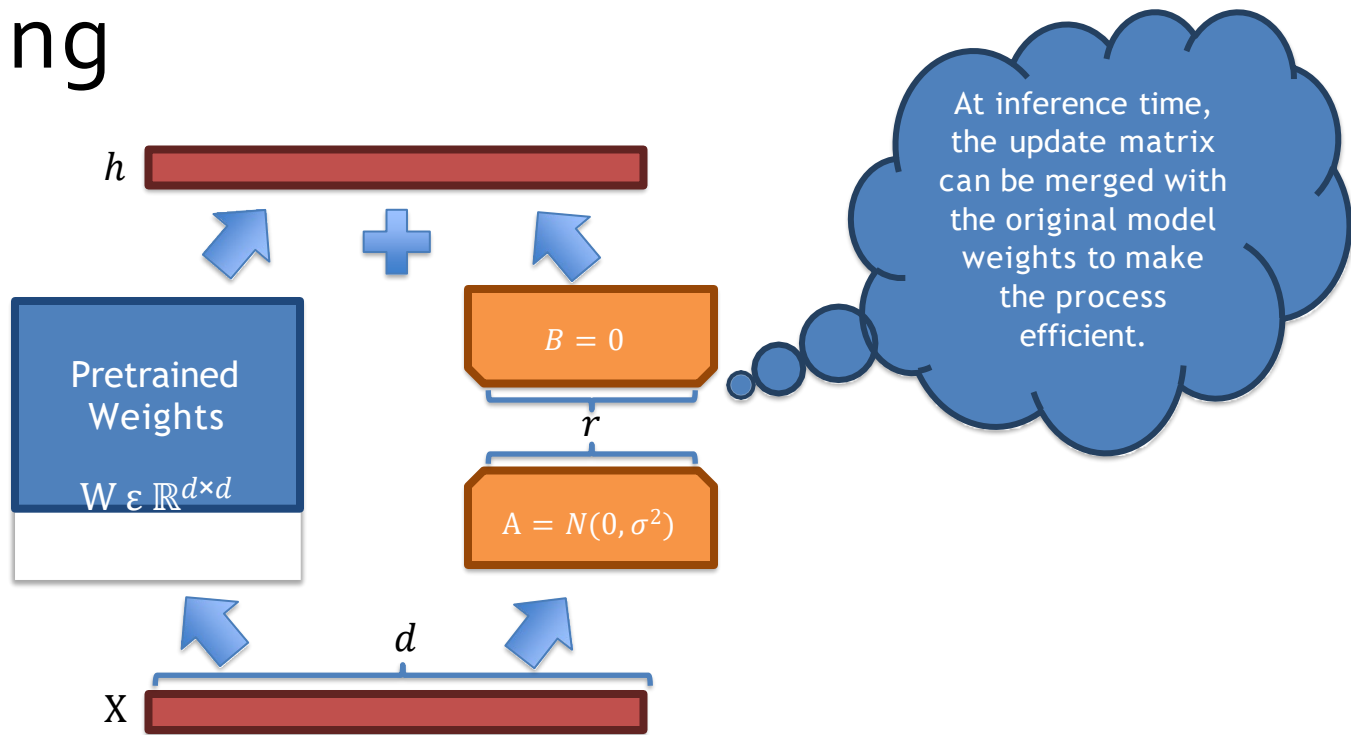
Notice how the reparameterization (LoRA) runs parallel to the original model.

LoRA - Working



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

LoRA - Working



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

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.

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

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

Optimizer
State
(FP32)

LoRA
Adapter
(FP16)

Base Model
(FP16)

LoRA



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

Optimizer
State
(FP32)

LoRA
Adapter
(FP16)

Base Model
(FP16)

LoRA



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	70.4\pm.1	8.85\pm.02	46.8\pm.2	71.8\pm.1	2.53\pm.02
<hr/>						
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	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	70.4\pm.1	8.89\pm.02	46.8\pm.2	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 network parameters.
- Before we dive into what QLoRA is, let's look at what quantization is.

Think of quantization as ‘**splitting range into buckets**’.

Floating point numbers

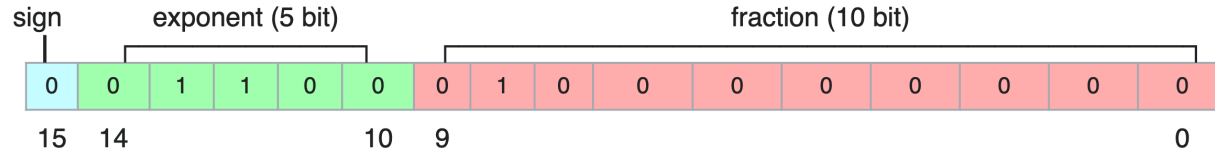
Floating point number is stored as $(-1)^s M 2^E$

- Sign bit s
- Fractional part $M = \text{frac}$
- Exponential part $E = \text{exp} - \text{bias}$

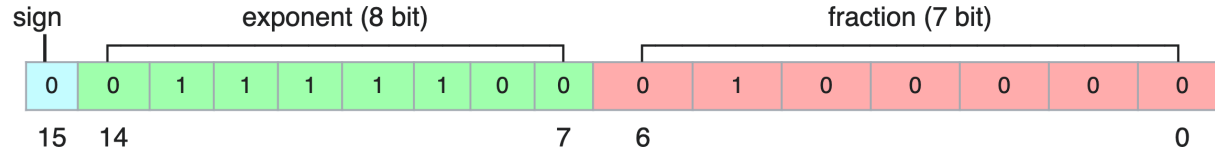


Reduced-precision floating point types

float16 (fp16)



bf16



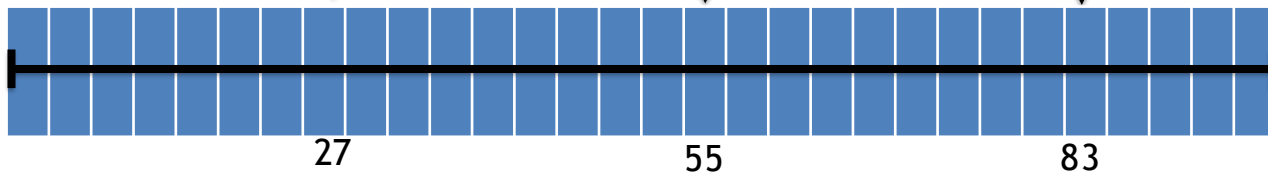
Quantization

Think of quantization as ‘splitting range into buckets’.

Any number between
0 and 100



Quantized by
whole numbers



Quantized by
10s



QLoRA

Let's look at an example!

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

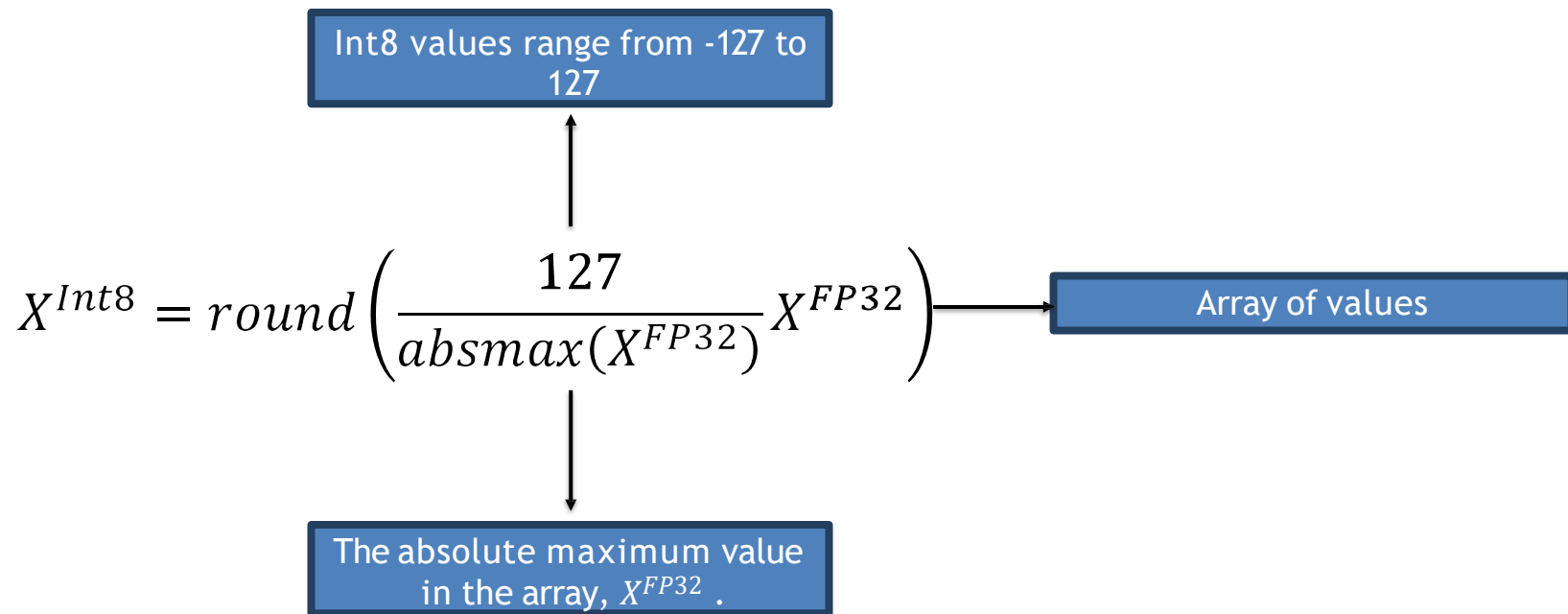
Here, FP32 refers to a 32-bit floating-point number.

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} :



QLoRA

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

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



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

QLoRA

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

In our example,



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

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

QLoRA

$$X^{Int8} = round(12.4509 \times \begin{bmatrix} 1.5 & 2.3 & 3.7 & 4.1 & 5.6 & 6.8 & 7.9 & 8.4 & 9.2 & 10.2 \end{bmatrix})$$

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

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

QLoRA

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

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

To dequantize:

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



QLoRA – Ingredient 1: 4-Bit

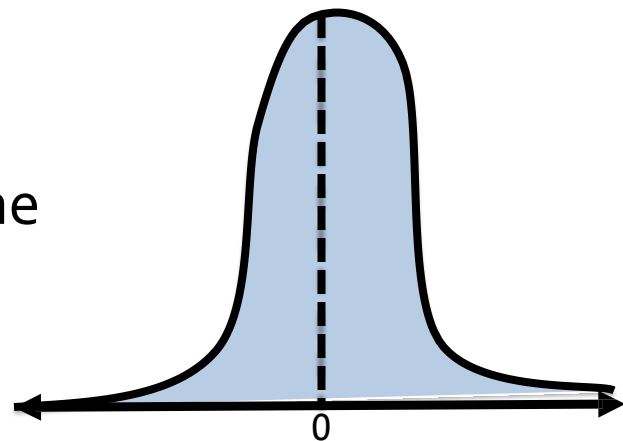


NormalFloat

- 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.

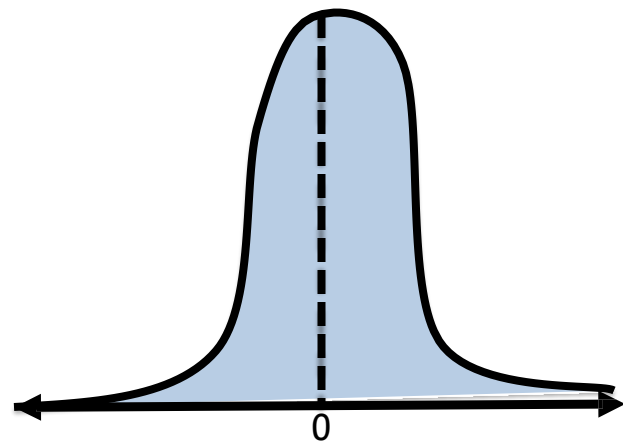
QLoRA – Ingredient 1: 4-Bit



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.



Equally spaced buckets



Equally sized buckets

This is an enhanced version of
quantile quantization.

QLoRA – Ingredient 2: Double Quantization



Remember this formula?



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

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:

$$\text{round}(W^{FP32} c^{FP32}) = W^{Int8}$$

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	0.3
-0.1	-1.5	-0.1	0.8	0.3
1.2	-1.7	-0.9	-0.3	0.3
0.4	0.1	-1.4	2.2	-1.1

Rescaled Weight Tensor

-4	-2	0	-22	16
-10	10	99	-57	10
-9	48	27	50	-18
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:

$$\text{round}(W^{FP32}_C^{FP32}) = W^{Int8}$$

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	48
22	8	-81	127	-66

Do you see a problem here?

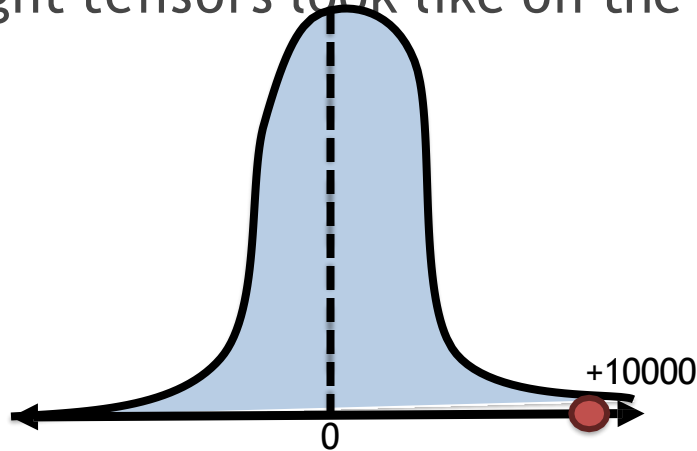
If we bring back the formula:

$$\text{round}(W^{FP32} c^{FP32}) = W^{Int8}$$

QLoRA – Ingredient 2: Double Quantization



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



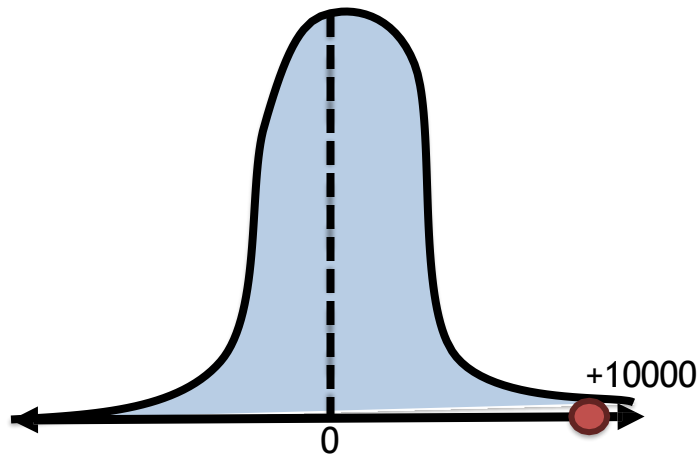
This is **unbounded** and could take up any **maximum value** (an outlier!).

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

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!).

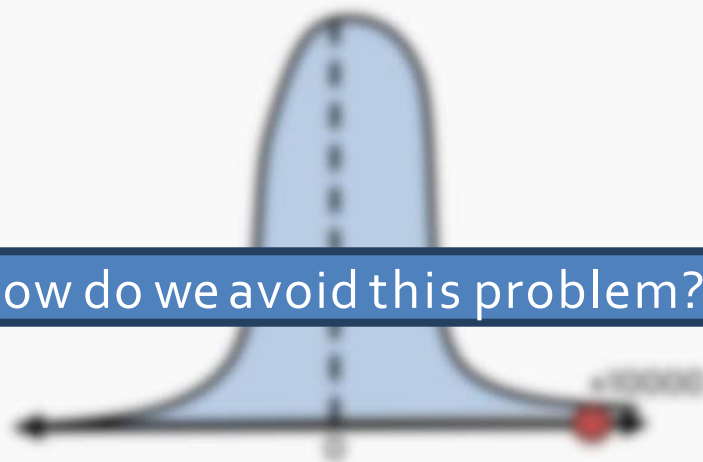
This could introduce bias in our quantization process

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

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**.



$$W^{int8} = \text{round}\left(\frac{127}{\text{absmax}(W^{FP32})} W^{FP32}\right)$$

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

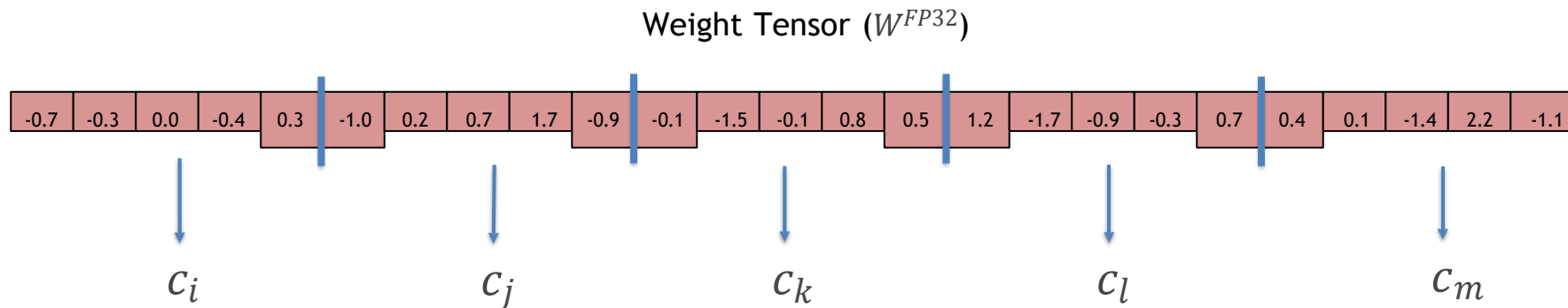


Quantization

We flatten the matrix as follows:

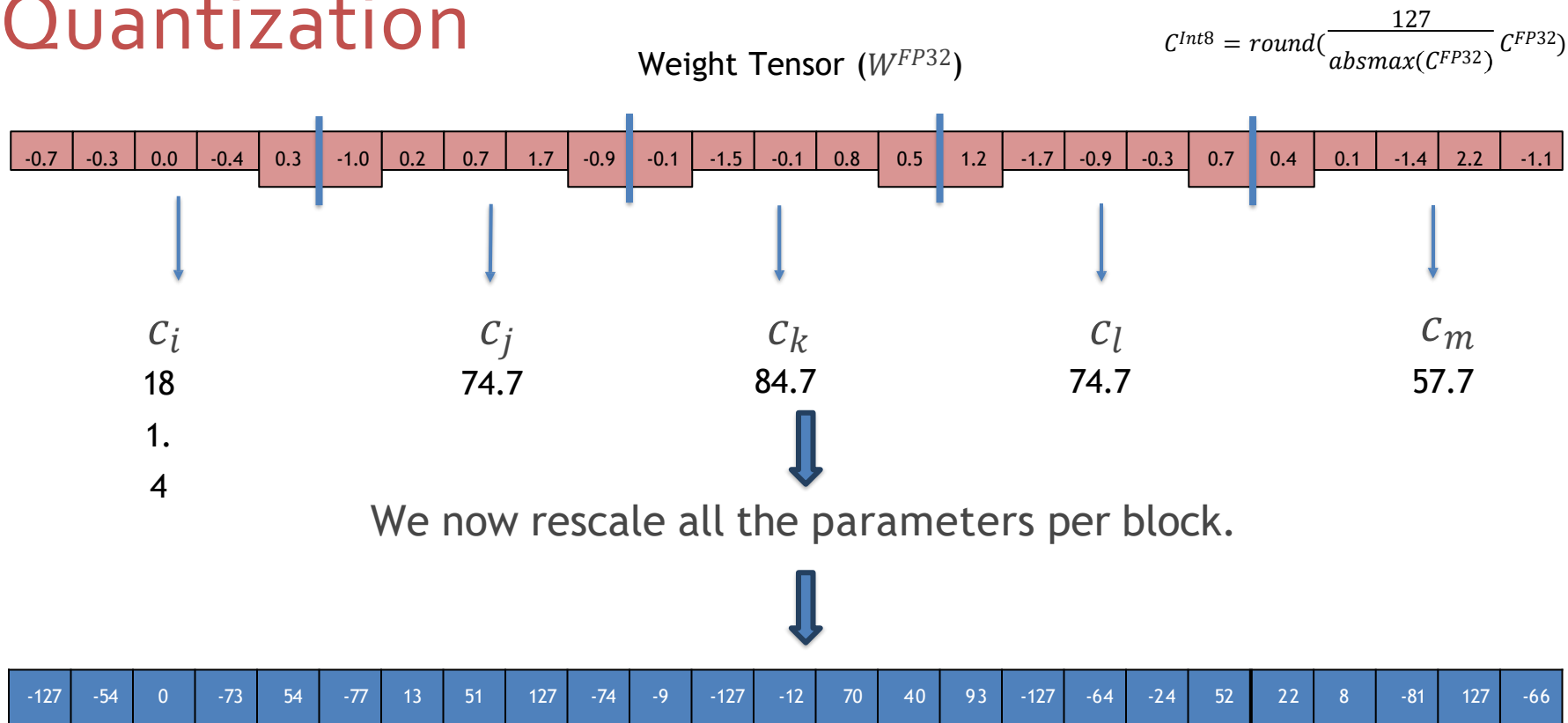
Now we divide it up into different blocks.

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



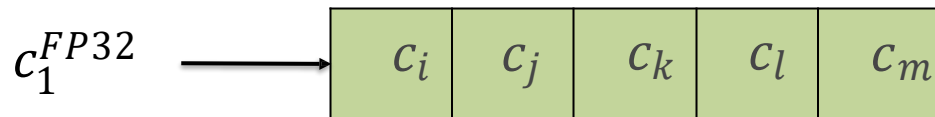
Rescaled Weight Tensor (W^{Int8})

QLoRA – Ingredient 2 : Double



Quantization

We now have a new array:



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



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: $25 \times 4 = 100$ 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

-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



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

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
-------	-------	-------	-------	-----

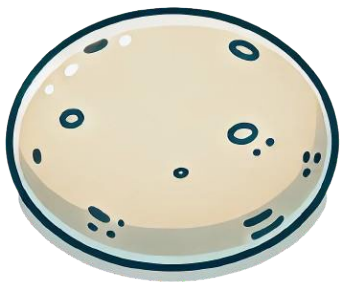
c_2^{FP32}

25 + 5 + 4 = 34 bytes

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

QLoRA – The Ingredients

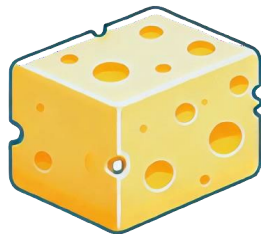
There are 3 key ingredients which helps us make QLoRA:



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!

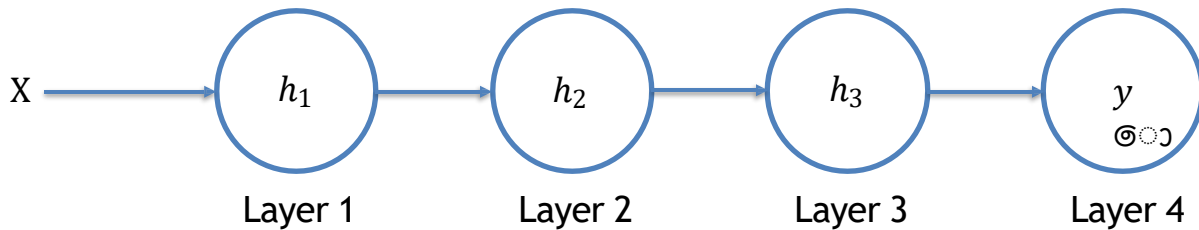
So, how do we train a modern Neural Networks without taking a hit on the memory?

We use [gradient checkpointing](#).

QLoRA – Ingredient 3

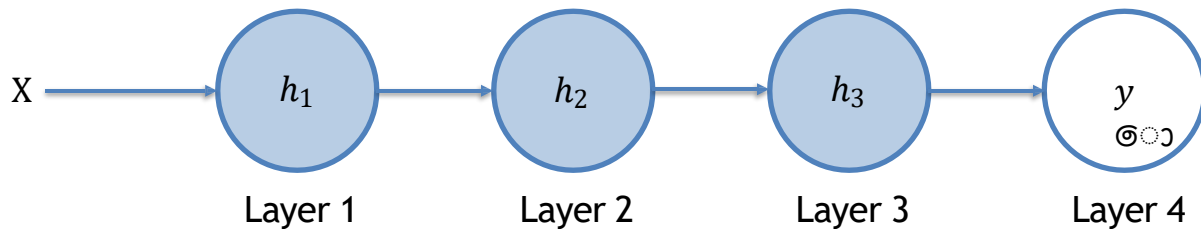


Imagine this simple neural network



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

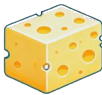
QLoRA – Ingredient 3



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.

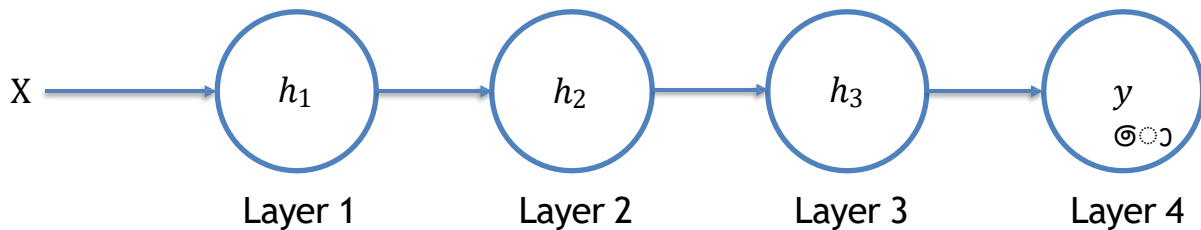


QLoRA – Ingredient 3

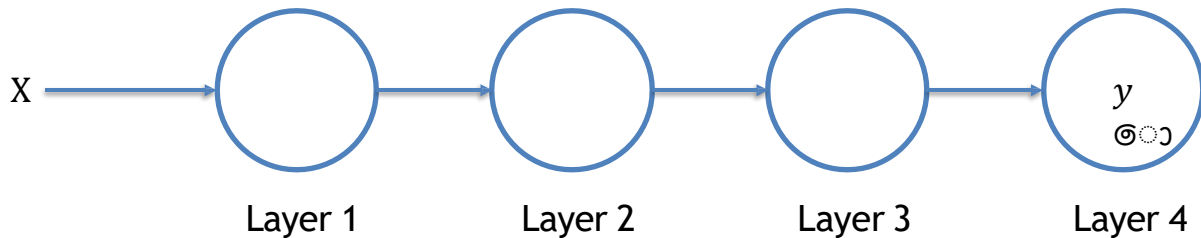
We only store in memory what is needed at the moment.

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!



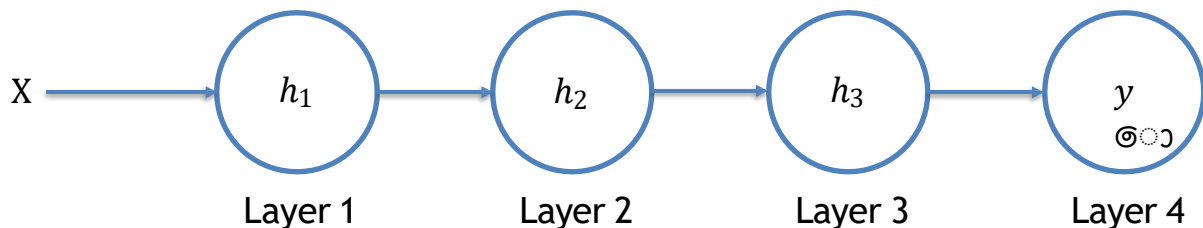
QLoRA – Ingredient 3



During backpropagation, we must recompute all the discarded activations.

To manage this, we introduce checkpoints in the middle.

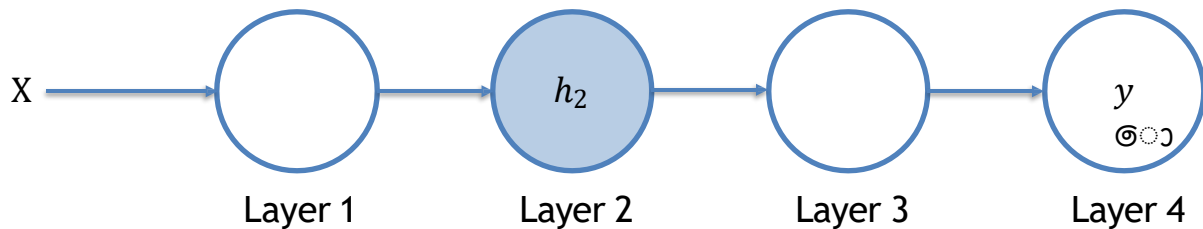
QLoRA – Ingredient 3



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

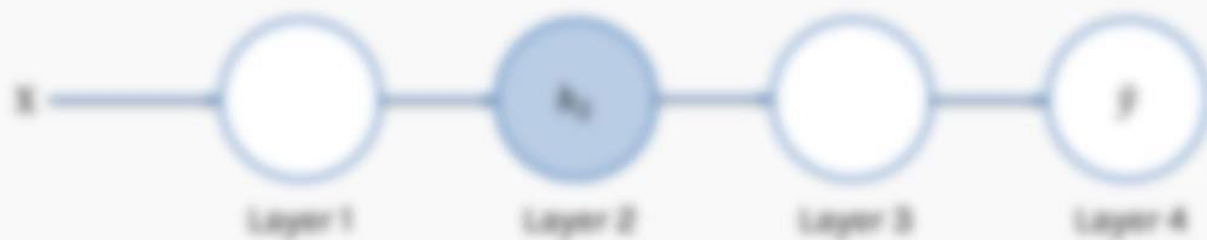
So, now when we re-compute the activations for **backward pass**, we don't have to start from the beginning!

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.



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'

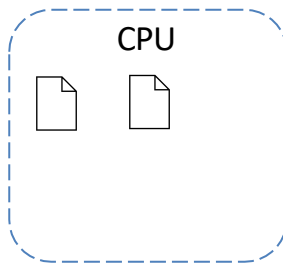
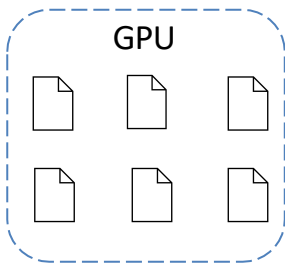
Paged Optimizer - Looping in your CPU

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.



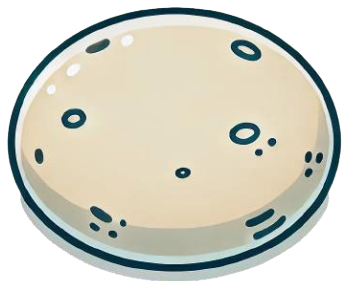
GPU Memory has
space now.



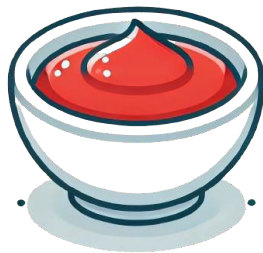
Now that the GPU has
space, when a page
moved to CPU is required,
we move it back to GPU
for computation.

QLoRA – The Ingredients

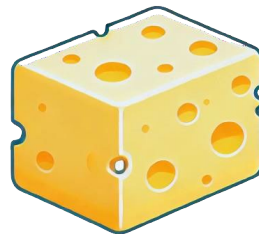
We saw the 3 key ingredients needed to make QLoRA:



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 => 160GB

LoRA

Optimizer
State
(FP32)



LoRA
Adapter
(FP16)



Base Model



FP16

10B => ~40GB

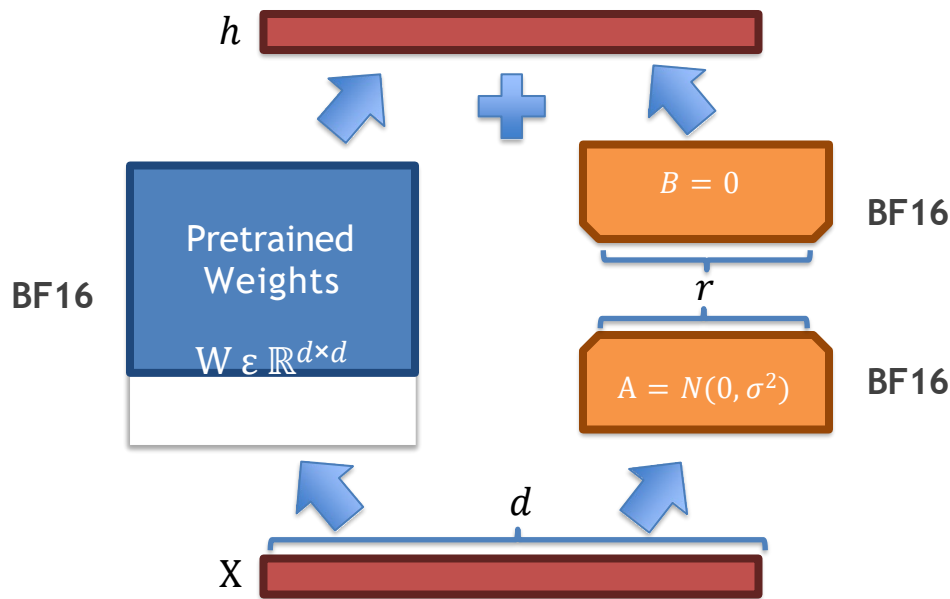


QLoRA – Putting it all together

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.





QLoRA – Putting it all together

Ingredient 1:



4-Bit NormalFloat

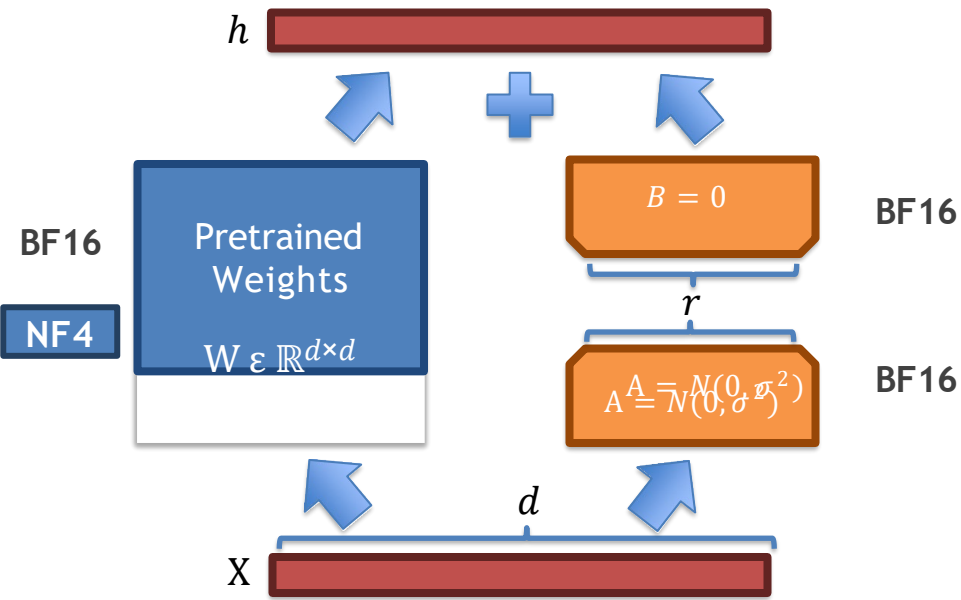
We store W , as 4-Bit NormalFloat

Ingredient 2:



Double Quantization

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





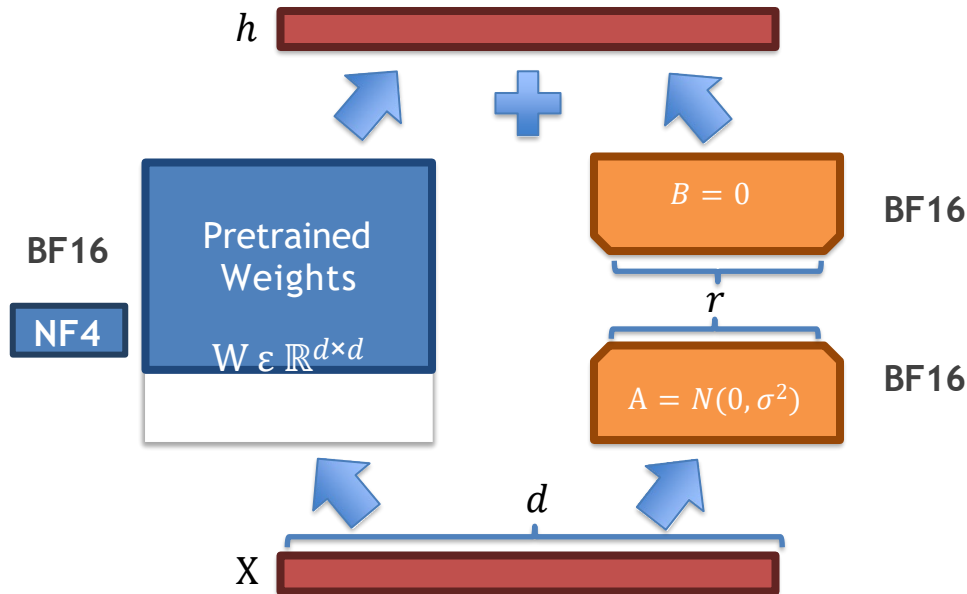
QLoRA – Putting it all together

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!



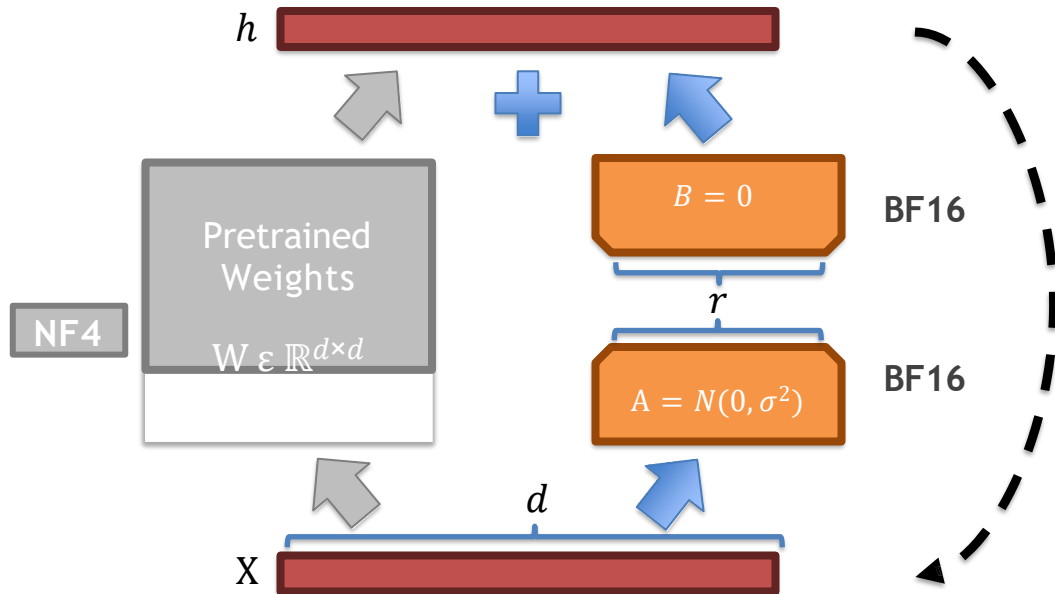


QLoRA – Putting it all together

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.

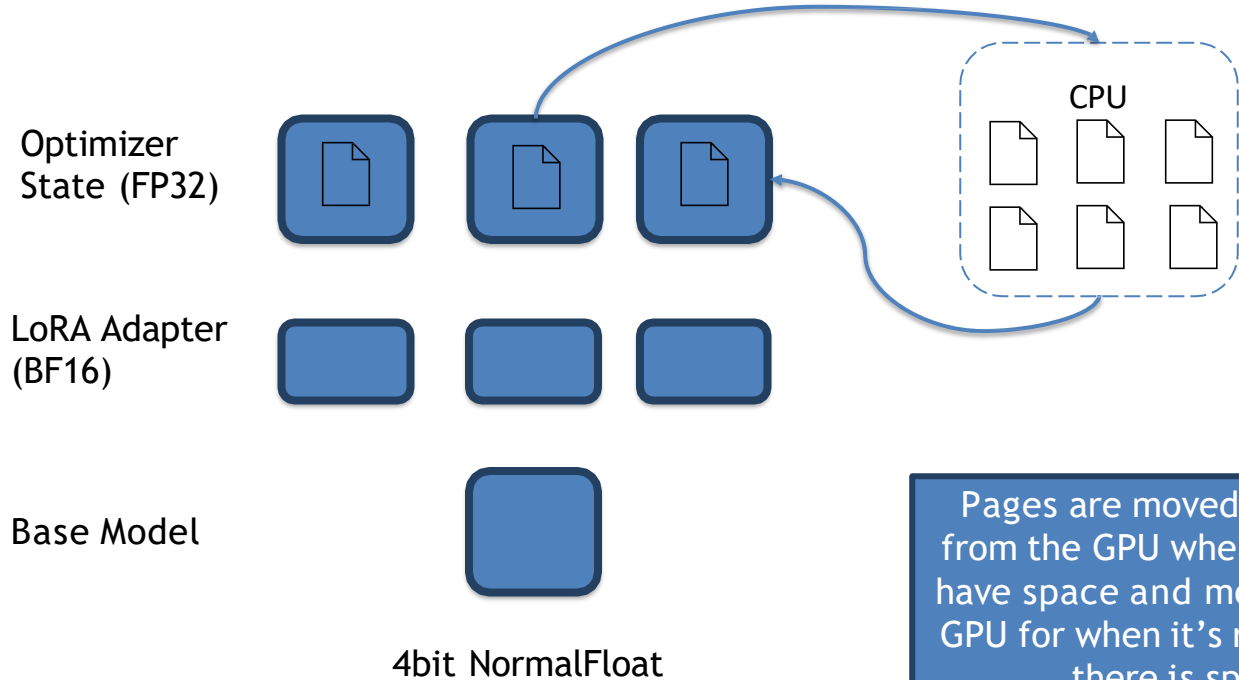




QLoRA – Putting it all together

Ingredient 3: 
Paged
Optimizer

QLoRA



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:

Initial LLM Weights

Scaling parameter

Decomposed matrices

Rank of B and A

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

The diagram illustrates the mathematical components of the LoRA weight update. The central equation is $W_0 + \frac{\alpha}{r}BA$. Four blue callout boxes point to specific parts of the equation: 'Initial LLM Weights' points to W_0 , 'Scaling parameter' points to α , 'Decomposed matrices' points to the product BA , and 'Rank of B and A' points to the denominator r .

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!

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

$$\begin{aligned} \text{where } \text{doubleDequant} (c_1^{FP32}, c_2^{k-bit}, W_o^{NF4}) &= \text{dequant} (\text{dequant} (c_1^{FP32}, c_2^{k-bit}), W_o^{4bit}) \\ &= W^{BF16} \end{aligned}$$

THANK
YOU

