Parameter Efficient Fine-tuning

LoRA, Adaptors, Prefix-tuning

Finetuning LLMs

- Typical LLMs has 7B, 70B, 400B parameters.
  - Finetuning 70B will take around 1TB of VRAM with a batch size of 1.
    - My rule of thumb is usually params * 4 * size(float) for full finetune
      - Why? Optimizer states (momentum, etc), activation value, current weight value
    - Calculator https://github.com/manuelescobar-dev/LLM-Tools
    - Info on memory requirements https://blog.scottlogic.com/2023/11/24/llm-mem.html https://blog.eleuther.ai/transformer-math/ https://arxiv.org/abs/2404.10933
- This is not practical for most users.

Example Memory breakdown of LLM

		OPT-1.3B, 16bit-float, seq 512
cuDNN and CUDA		~1GB
Model weights	size(float)*N	2.6GB
Gradients	size(float)*$N_{trainable}$	2.6GB
Hidden state activations	~size(float) L (20H seq + 3 seq^2)	1 GB
Optimizer states	2*size(float)*$N_{trainable}$	5.2 GB
(Maybe) fp32 copy of the gradients	4*$N_{trainable}$	10.2 GB

Estimate 12.4 GB, actual 11.0 GB

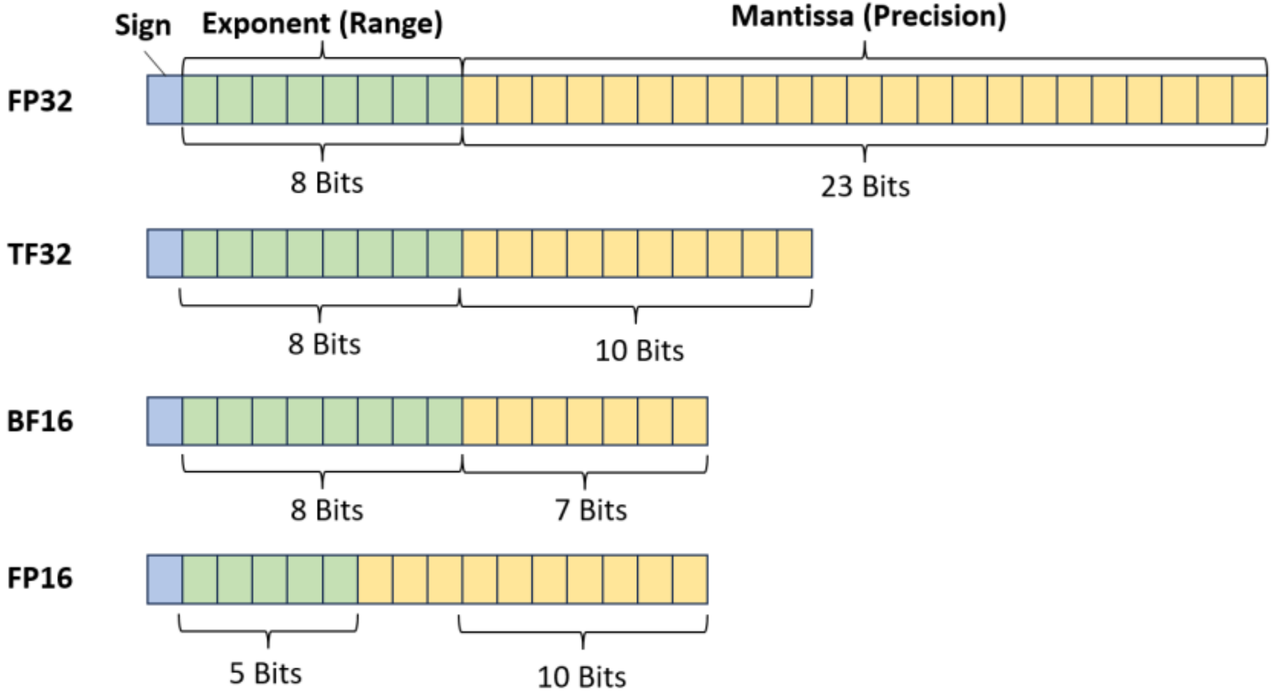
Tricks such as gradient/activation checkpointing can help reduce memory requirements for hidden states

https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/scaling/JAX/single_gpu_techniques.html#Gradient-Checkpointing-/-Activation-Recomputation

https://www.youtube.com/watch?v=StdrAJZsmw4

Notes on precision

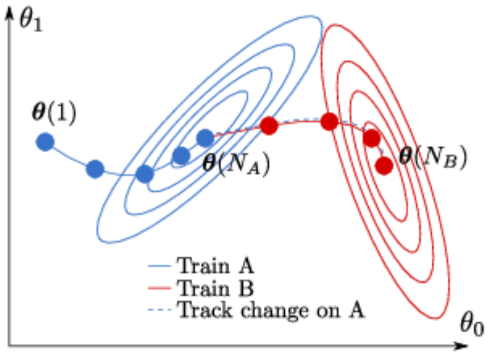
- Most transformer are trained in mixed precision. Usually BF16+FP32 or FP16+FP32

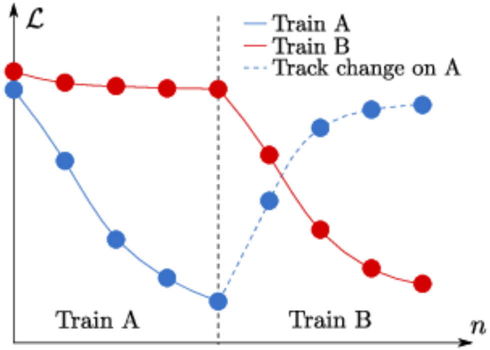


Catastrophic Forgetting

- Finetuning on a new dataset usually makes the model forgets its original capabilities.
- More likely that your model will be dumber if you finetune a model on a small dataset
    - Remember Chinchilla Scaling Law?

- Instead of finetuning the entire model, let's focus on parts of the model instead





https://arxiv.org/abs/1906.02568

Parameter Efficient Fine-Tuning

What if we train on less parameters

Train on 0.2M parameters

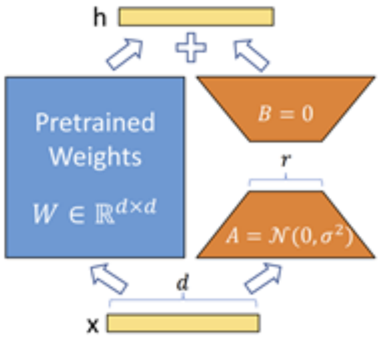
		OPT-1.3B, 16bit-float, seq 512
cuDNN and CUDA		~1GB
Model weights	size(float)*N	2.6GB
Gradients	size(float)*$N_{trainable}$	0.4MB
Hidden state activations	~size(float) L (20H seq + 3 seq^2)	1 GB
Optimizer states	2*size(float)*$N_{trainable}$	0.8MB
(Maybe) fp32 copy of the gradients	4*$N_{trainable}$	1.6MB

Estimate 4.6 GB, actual 5.7 GB

https://www.youtube.com/watch?v=StdrAJZsmw4

Parameter Efficient Fine-tuning

0. In-context learning (Prompt Engineering)
1. Prefix-tuning
    a. Append learnable tokens in the input
2. Adapter Module
    a. Insert a small number of layers that are relatively small compared to the entire model.
3. Select parts of network to update
    a. BiTFit, freeze and reconfigure
4. Low-Rank Adaptation (LoRA)
    a. Represent an adaptation weight (gradient) with a low-rank matrix.



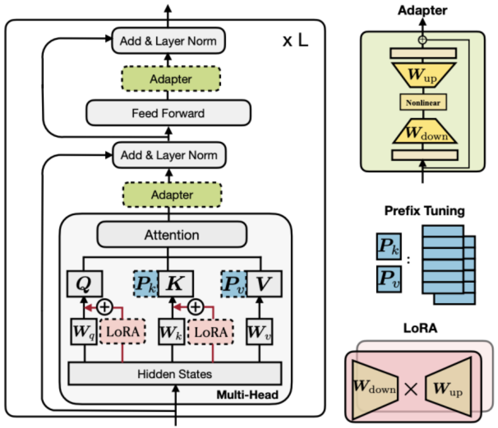
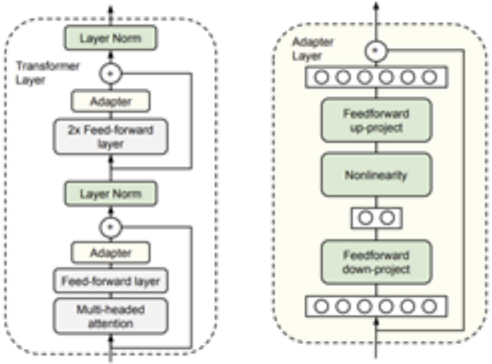


Figure 1: Illustration of the transformer architecture and several state-of-the-art parameter-efficient tuning methods. We use blocks with dashed borderlines to represent the added modules by those methods.



https://openreview.net/forum?id=0RDcd5Axok

Adapters

- Add small adapter layers after attention and feed forward layers.
- Only update these layers

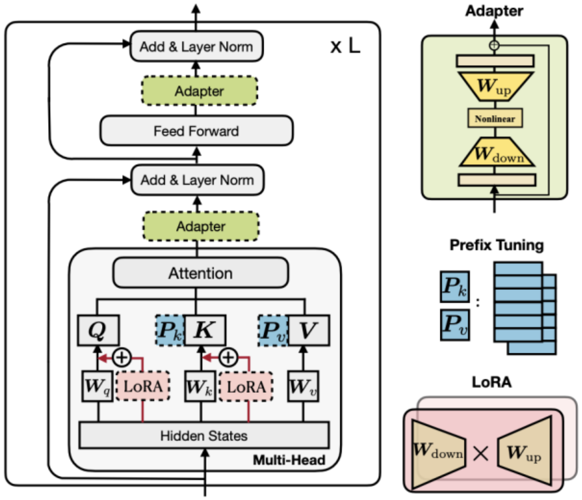
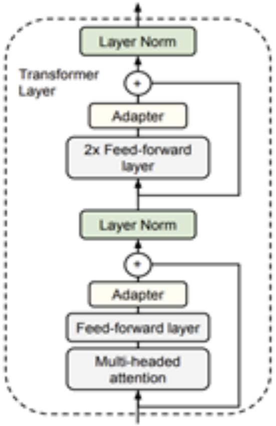
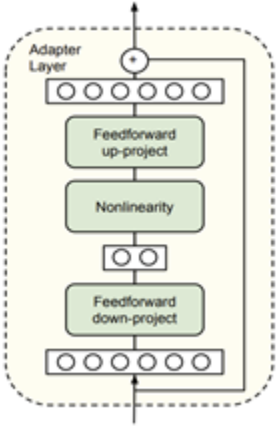


Figure 1: Illustration of the transformer architecture and several state-of-the-art parameter-efficient tuning methods. We use blocks with dashed borderlines to represent the added modules by those methods.

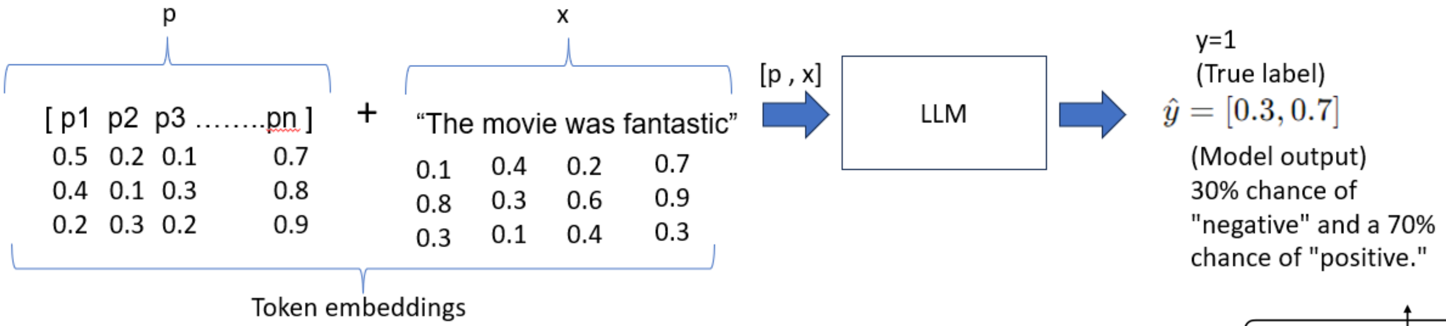




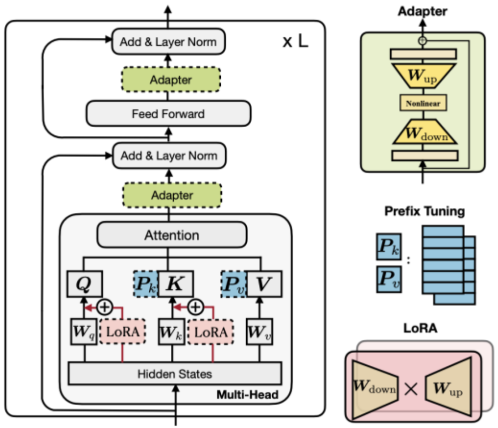
https://arxiv.org/abs/1902.00751

Prompt tuning / Prefix tuning

- Original idea: add a learnable token to the input text instead of prompt engineering your prompt



- Modern versions only append the key and value tokens (prefix tuning)
- Some people refer to this as a **soft prompt**



https://developer.ibm.com/articles/awb-how-prompt-tuning-works/
https://arxiv.org/pdf/2104.08691

Ladder Side-Tuning (LST)

- Add another branch to the original network.
  - Think of original model as multiple feature extractors
- Reduce the requirement to backprop over the entire network

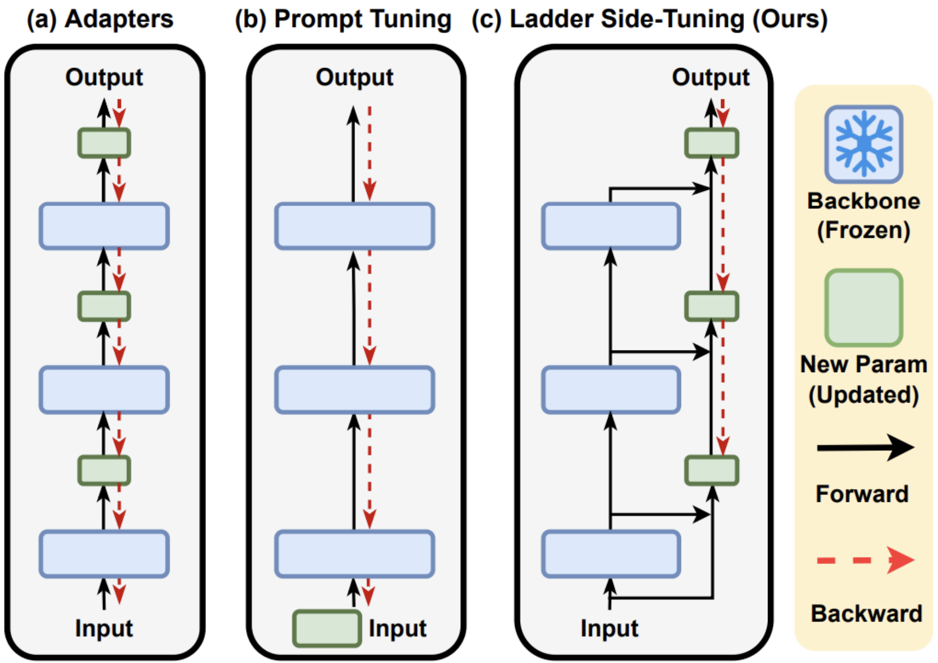


Figure 2: Comparison between transfer learning with (a) Adapters, (b) Prompt Tuning, and our (b) Ladder Side-Tuning (LST). LST reduces memory usage by removing the need of backpropgation through backbone networks.

https://arxiv.org/abs/2206.06522

BitFit

- A kind of selective finetuning
- Finetune only the **bias** of the network
- Works well on the paper (BERT) but doesn't work well on LLM scale

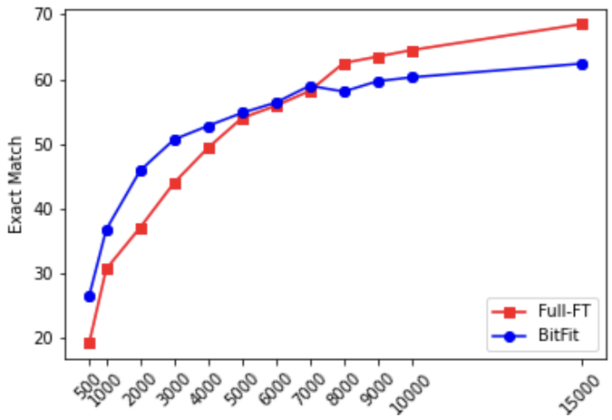
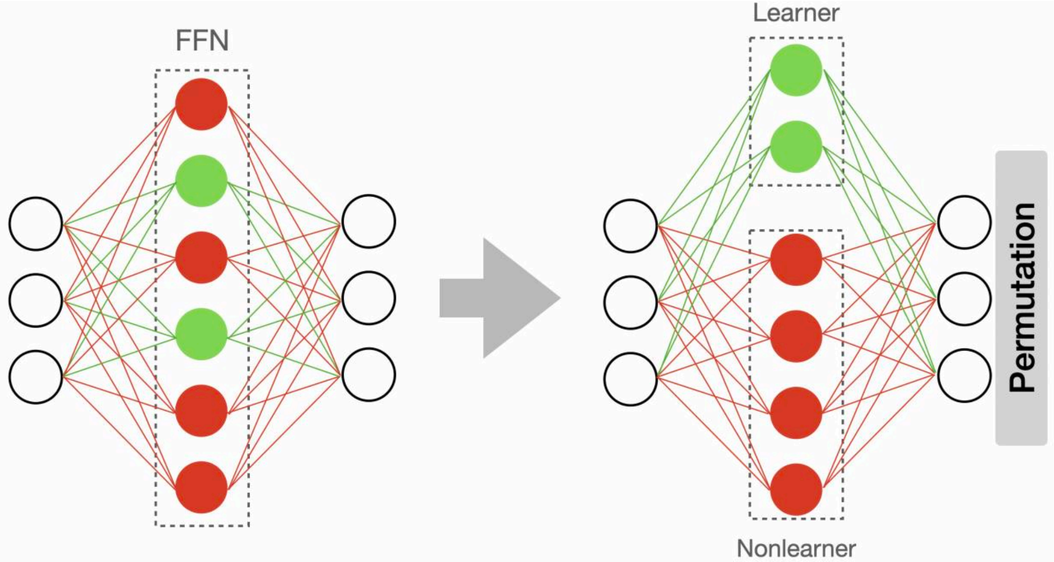


Figure 2: Comparison of BitFit and Full-FT with $\text{BERT}_{\text{BASE}}$ exact match score on SQuAD validation set.

https://arxiv.org/abs/2106.10199

Squeeze and reconfigure

- Selects part of the network based on some criterion.
  - In the paper they used the size of the change in weight in full finetuning
- Only learn on that part

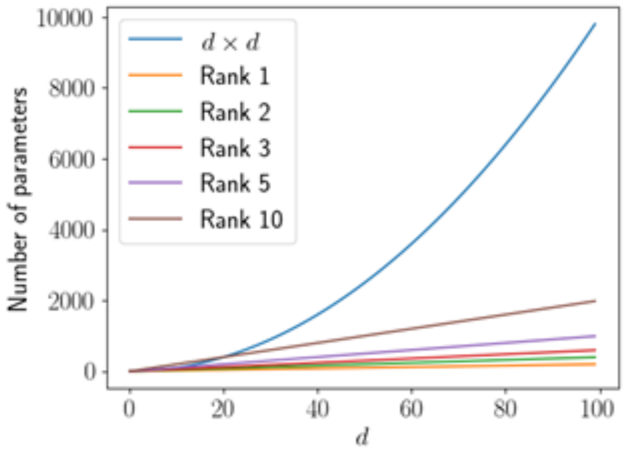


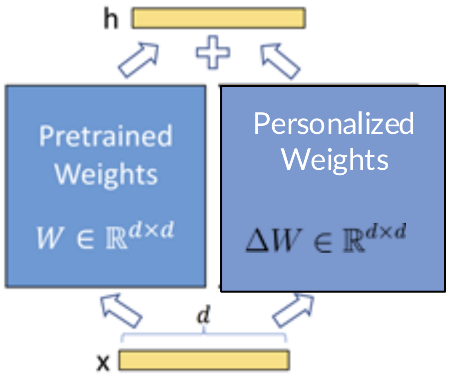
https://arxiv.org/abs/2205.01541

Low-Rank Adaptation

LoRA compress the update weights using low-rank decomposition. You are essentially updating all weights in a low parameter space.

$$\Delta W = BA \qquad B \in \mathbb{R}^{d \times r} A \in \mathbb{R}^{r \times d}$$

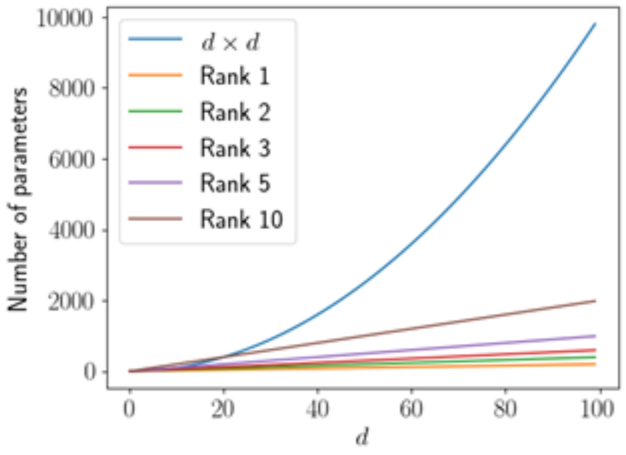


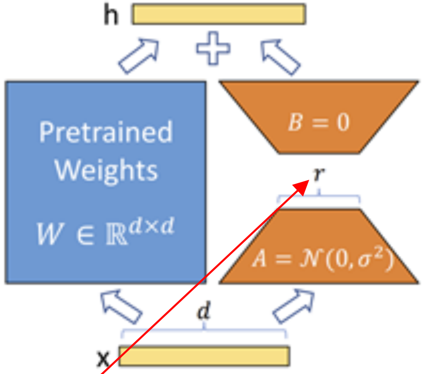


Low-Rank Adaptation

LoRA compress the update weights using low-rank decomposition. You are essentially updating all weights in a low parameter space.

$$\Delta W = BA \qquad B \in \mathbb{R}^{d\times r} A \in \mathbb{R}^{r\times d}$$





r – rank parameter in LoRA

Low-Rank Adaptation

In the initialization process, we use a random Gaussian initialization for A and zero for B.

$$h = W_0x + \underset{0}{\cancel{\Delta W}}x = W_0x + BAx$$

Therefore, the model is initialized to be identical to the pretrained weights.

In addition, the paper proposes to scale low-rank matrices by alpha/r, claiming that it helps reduce the need to adjust hyperparameters when varying r.

Works well. Popularized by stable diffusion finetuning.

Can be seen as Learning Rate

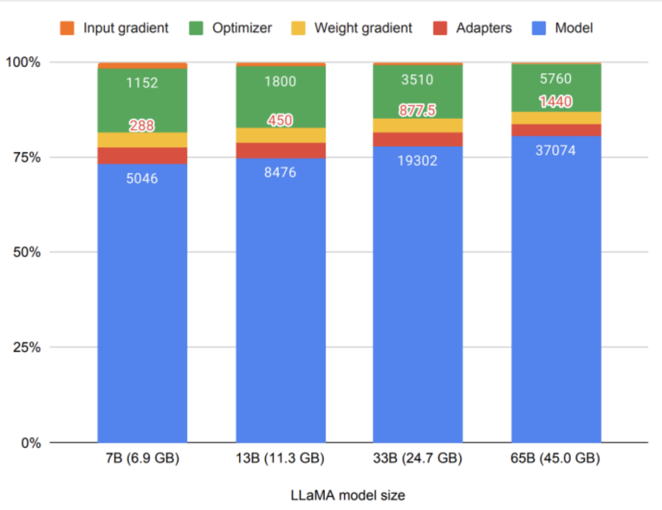
$$h = W_0x + \frac{\boxed{\alpha}}{r}BAx$$

Newer variants introduce dropout (peft library drops input x). New paper drops B and A columns/rows.
PiSSA initializes the weight matrixes with SVD and works better.

https://arxiv.org/abs/2106.09685 LORA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS 2021
https://openreview.net/forum?id=c4498OydLP
https://arxiv.org/abs/2404.02948

QLoRA

- QLoRA is an implementation of LoRA that focuses on compute efficiency
- Quantize both the value and the quantization scaling (double quantization)
- LoRA that is done on a quantized weights of the original model
- Uses 4-bit NormalFloat (for weight storage) and BF16 (for compute)



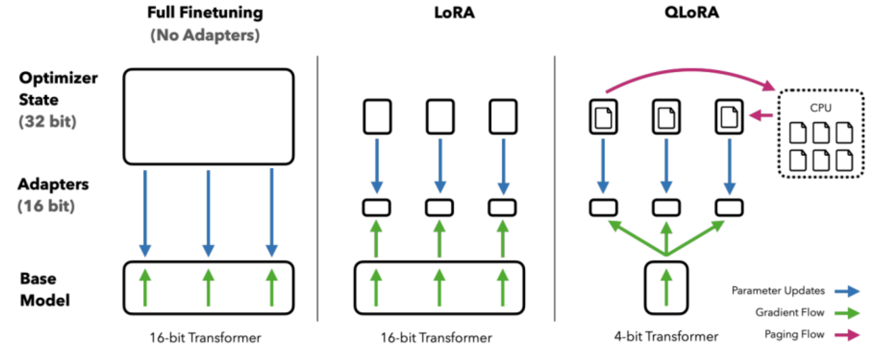
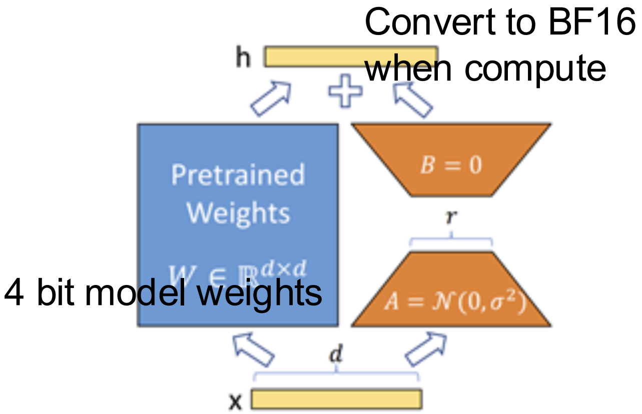


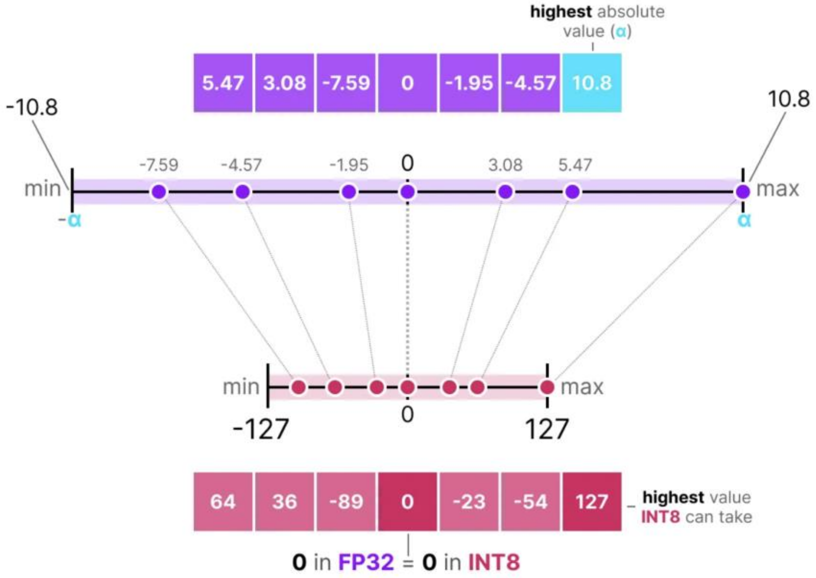
Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.



https://arxiv.org/pdf/2305.14314

Quick slide on Quantization

- Symmetric quantization keeps track of the abs(max) for scaling to the quantized value



Note the [-127, 127] range of values represents the restricted range. The unrestricted range is [-128, 127] and depends on the quantization method.

$$s = \frac{2^{b-1}-1}{\alpha} \quad \text{(scale factor)}$$

$$X_{quantized} = \text{round}\left(s \cdot X\right) \quad \text{(quantization)}$$

Filling in the values would then give us the following:

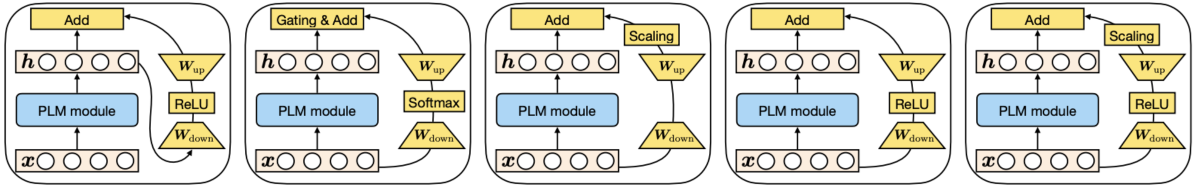
$$s = \frac{127}{10.8} = 11.76 \quad \text{(scale factor)}$$

$$X_{quantized} = \text{round}\left(11.76 \cdot \blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\right) \quad \text{(quantization)}$$

https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization

Hybrid approaches

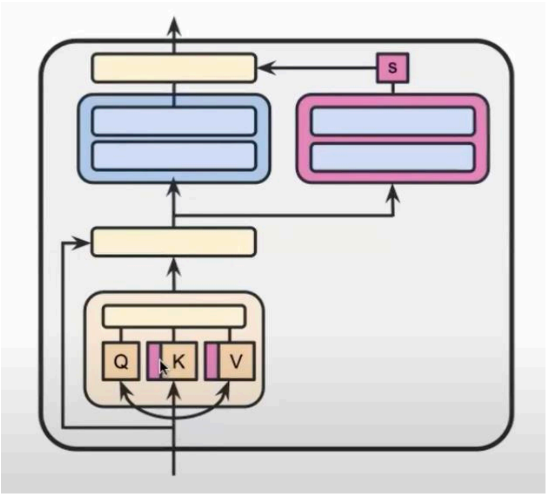
- There are MANY approaches that are based on these simple methods.
- Example MAM adaptors (Mix-and-Match adaptors)
  - Sacled parallel adapter + prefixed finetuning
- S4 finds optimal combinations of PEFT



(a) Adapter (b) Prefix Tuning (c) LoRA (d) Parallel Adapter (e) Scaled PA

Table 2: Accuracy on the dev set of MNLI and SST2. MAM Adapter is proposed in §4.6. Bitfit numbers are from Ben Zaken et al. (2021).

Method (# params)	MNLI	SST2
Full-FT (100%)	$87.6_{\pm.4}$	$94.6_{\pm.4}$
Bitfit (0.1 %)	84.7	93.7
Prefix (0.5%)	$86.3_{\pm.4}$	$94.0_{\pm.1}$
LoRA (0.5%)	$87.2_{\pm.4}$	$94.2_{\pm.2}$
Adapter (0.5%)	$87.2_{\pm.2}$	$94.2_{\pm.1}$
MAM Adapter (0.5%)	$\mathbf{87.4}_{\pm.3}$	$94.2_{\pm.3}$



https://arxiv.org/pdf/2110.04366
https://arxiv.org/abs/2301.01821

Performance comparison

- These depend on application and exact benchmarks, but people tends to fine LoRA to perform well.

Method	$T5_{LARGE}$	$T5_{3B}$	$T5_{11B}$
Additive methods			
Adapters (Houlsby)	$\mathbf{67.34}_{\pm 9.58}$	$\underline{74.66}_{\pm 1.68}$	$\mathbf{76.16}_{\pm 1.47}$
Adapters (Pfeiffer)	$62.93_{\pm 3.52}$	$\underline{69.92}_{\pm 5.61}$	$50.72_{\pm 1.69}$
Parallel Adapter	$\underline{66.78}_{\pm 3.85}$	$\underline{74.15}_{\pm 0.88}$	$\underline{68.74}_{\pm 12.73}$
IA3	$55.06_{\pm 1.80}$	$41.77_{\pm 0.50}$	$61.05_{\pm 3.42}$
Prefix Tuning	$45.05_{\pm 3.89}$	$48.90_{\pm 5.37}$	$51.93_{\pm 2.21}$
Prompt Tuning	$8.97_{\pm 30.91}$	$8.38_{\pm 0.50}$	-
Selective methods			
LN Tuning	$\underline{64.68}_{\pm 4.59}$	$72.95_{\pm 1.38}$	$\mathbf{73.77}_{\pm 0.93}$
Reparametrization-based methods			
LoRA (q and v)	$\mathbf{67.42}_{\pm 2.32}$	$\mathbf{75.49}_{\pm 1.71}$	$\mathbf{76.20}_{\pm 1.27}$
LoRA (all linear)	$\mathbf{68.76}_{\pm 1.83}$	$\mathbf{75.22}_{\pm 1.28}$	$\mathbf{76.58}_{\pm 2.16}$
KronA	$\underline{65.68}_{\pm 3.27}$	$71.98_{\pm 0.57}$	$\underline{72.13}_{\pm 7.30}$
Hybrid methods			
MAM	$46.90_{\pm 6.47}$	$45.57_{\pm 4.67}$	$51.49_{\pm 0.54}$
Compacter	$64.48_{\pm 1.81}$	$70.72_{\pm 0.87}$	$\mathbf{74.33}_{\pm 1.40}$
Compacter++	$64.78_{\pm 2.23}$	$71.00_{\pm 1.62}$	$\mathbf{74.72}_{\pm 0.82}$
Unipelt	$44.10_{\pm 15.48}$	$47.16_{\pm 4.84}$	$52.29_{\pm 3.09}$
Full tuning	67.22	74.83	73.25

Table 4: Average model performance on our collection of datasets (Section 11.1) with 95% confidence intervals (two standard deviations). We **bold** values that outperform full-tuning by mean value. We underline values that achieve full-tuning performance within the confidence interval.

https://arxiv.org/abs/2303.15647

Some consideration on picking PEFT approaches

- Does it save storage size compared to full finetune (Disk)
- Whether it increases inference overhead (with additional parameters)
- Does it save memory when training (RAM)
- Some method does not require you to backprop through parts of the original model (BP)

Method	Type	Efficiency Disk	Efficiency RAM	Efficiency BP	Inference overhead
Adapters (Houlsby et al., 2019)	A	✓	✓	✗	+ FFN
AdaMix (Wang et al., 2022)	A	✓	✓	✗	+ FFN
SparseAdapter (He et al., 2022b)	AS	✓	✓	✗	+ FFN
Cross-Attn tuning (Gheini et al., 2021)	S	✓	✓	✗	No overhead
BitFit (Ben-Zaken et al., 2021)	S	✓	✓	✗	No overhead
DiffPruning (Guo et al., 2020)	S	✓	✗	✗	No overhead
Fish-Mask (Sung et al., 2021)	S	✓	✗[5]	✗	No overhead
LT-SFT (Ansell et al., 2022)	S	✓	✗[5]	✗	No overhead
Prompt Tuning (Lester et al., 2021)	A	✓	✓	✗	+ input
Prefix-Tuning (Li and Liang, 2021)	A	✓	✓	✗	+ input
Spot (Vu et al., 2021)	A	✓	✓	✗	+ input
IPT (Qin et al., 2021)	A	✓	✓	✗	+ FFN and input
MAM Adapter (He et al., 2022a)	A	✓	✓	✗	+ FFN and input
Parallel Adapter (He et al., 2022a)	A	✓	✓	✗	+ FFN
Intrinsinc SAID (Aghajanyan et al., 2020)	R	✓	✗	✗	No overhead
LoRa (Hu et al., 2022)	R	✓	✓	✗	No overhead
DoRA (Liu et al., 2024)	R	✓	✓	✗	No overhead
UniPELT (Mao et al., 2021)	AR	✓	✓	✗	+ FFN and input
Compacter (Karimi Mahabadi et al., 2021)	AR	✓	✓	✗	+ FFN
PHM Adapter (Karimi Mahabadi et al., 2021)	AR	✓	✓	✗	+ FFN
KronA (Edalati et al., 2022)	R	✓	✓	✗	No overhead
KronA$^{B}_{res}$ (Edalati et al., 2022)	AR	✓	✓	✗	+ linear layer
(IA)^3 (Liu et al., 2022)	A	✓	✓	✗	+ gating
Attention Fusion (Cao et al., 2022)	A	✓	✓	✓	+ decoder
LeTS (Fu et al., 2021)	A	✓	✓	✓	+ FFN
Ladder Side-Tuning (Sung et al., 2022)	A	✓	✓	✓	+ decoder
FAR (Vucetic et al., 2022)	S	✓	✓	✗	No overhead
S4-model (Chen et al., 2023)	ARS	✓	✓	✗	+ FFN and input

More on PeFT

- Refer to the paper https://arxiv.org/abs/2303.15647 for overview of other methods.
- https://github.com/synbol/Parameter-Efficient-Transfer-Learning-Benchmark for computer vision benchmarks

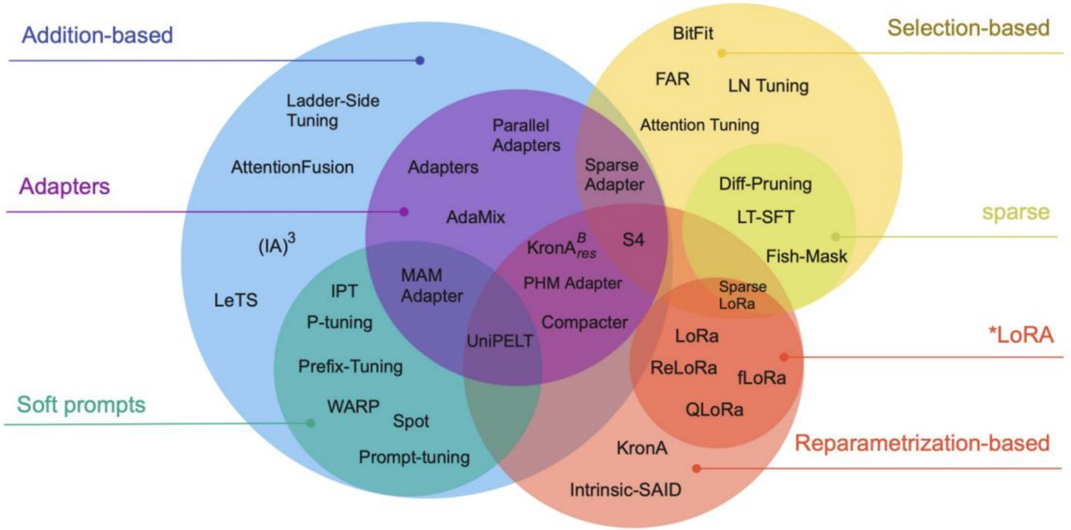


Figure 2: Parameter-efficient fine-tuning methods taxonomy. We identify three main classes of methods: **Addition**-based, **Selection**-based, and **Reparametrization**-based. Within additive methods, we distinguish two large included groups: **Adapter-like** methods and **Soft prompts**.

Scaling and PEFT

- Large/better pre-trained models require a small amount of parameters to be finetuned
    - The larger the model the smaller the amount of parameters need to be changed
    - Ex Typhoon-2 uses LoRA rank = 8 to finetune from base models (LLAMA 3 and Qwen2)

- Implications
    - PEFT should always be used in finetuning large models

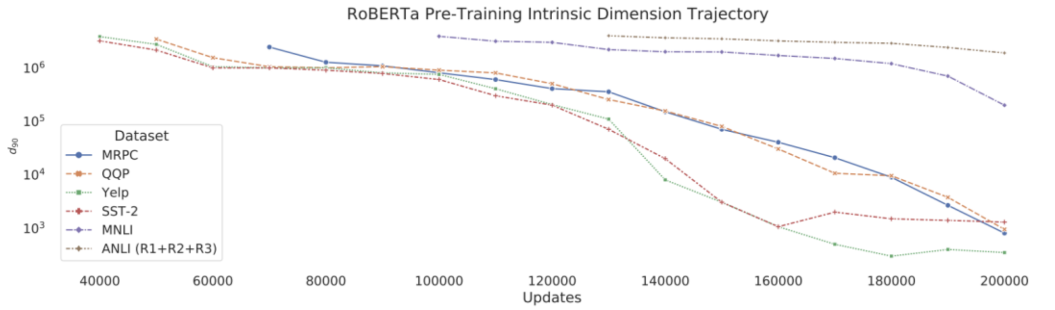


Figure 2: Every 10k updates of RoBERTa-Base that we trained from scratch, we compute d_{90} for six datasets; MRPC, QQP, Yelp Polarity, SST-2, MNLI, and ANLI. If we were unable to compute a d_{90} for a specific checkpoint, we do not plot the point, hence some datasets start at later points. Unable to compute means either we could not fine-tune the full checkpoint to accuracy above majority class or stabilize SAID training.

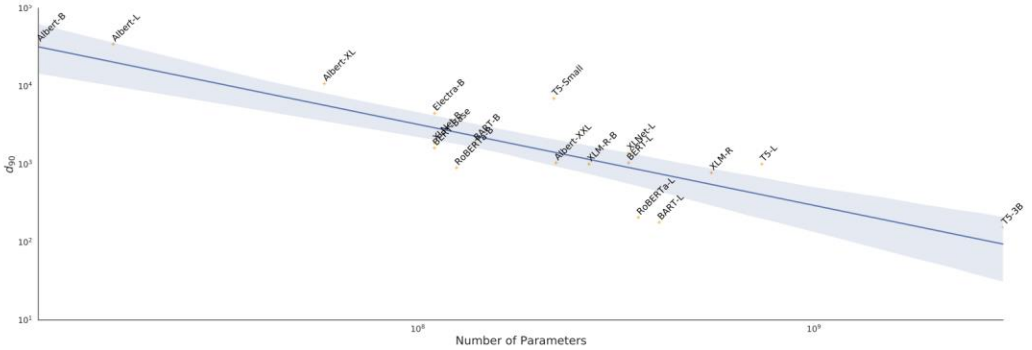
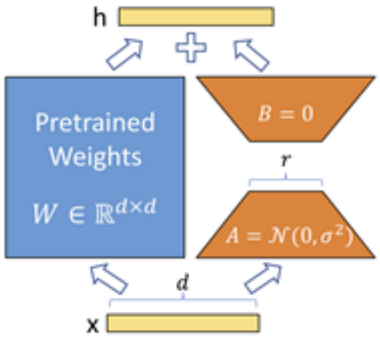


Figure 3: We calculate the intrinsic dimension for a large set of pre-trained models using the SAID method on the MRPC dataset.

https://arxiv.org/abs/2012.13255
https://arxiv.org/abs/2412.13702

Parameter Efficient Fine-tuning

0. In-context learning (Prompt Engineering)
1. Prefix-tuning
    a. Append learnable tokens in the input
2. Adapter Module
    a. Insert a small number of layers that are relatively small compared to the entire model.
3. Select parts of network to update
    a. BiTFit, freeze and reconfigure
4. Low-Rank Adaptation (LoRA)
    a. Represent an adaptation weight (gradient) with a low-rank matrix.



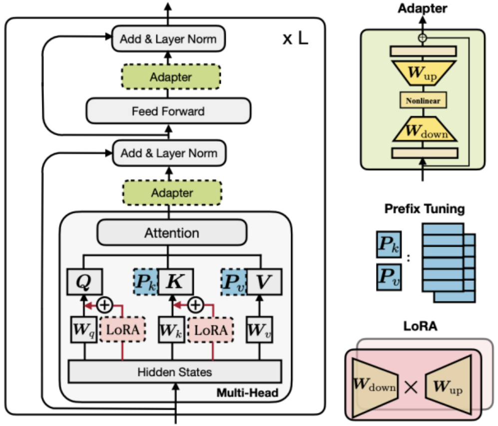
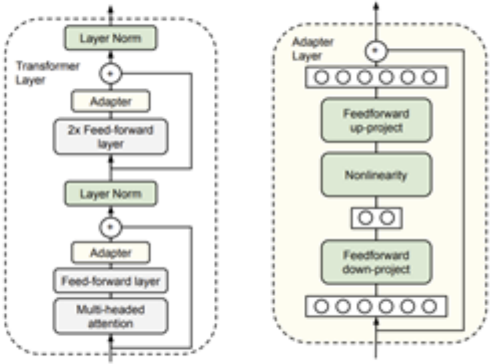


Figure 1: Illustration of the transformer architecture and several state-of-the-art parameter-efficient tuning methods. We use blocks with dashed borderlines to represent the added modules by those methods.



https://openreview.net/forum?id=0RDcd5Axok

Prompt engineering

- Prompt engineer does not learn any weight updates
- But you don’t have to train!
  - If you have a long prompt, you are paying for it in inference compute.
  - KV Caching can help.