10-423/10-623 Generative AI

Machine Learning Department
School of Computer Science
Carnegie Mellon University

Parameter Efficient Fine-Tuning

Matt Gormley & Henry Chai
Lecture 10
Sep. 30, 2024

Reminders

- Homework 2: Generative Models of Images
  - Out: Mon, Sep 23
  - Due: Mon, Oct 7 at 11:59pm

PARAMETER EFFICIENT FINE-TUNING

Few-shot Learning with LLMs

Suppose you have…

- a dataset $D = \{(x_i, y_i)\}_{i=1}^N$ and N is rather small (i.e. few-shot setting)
- a very large (billions of parameters) pre-trained language model

There are two ways to "learn"

Option A: Supervised fine-tuning

- **Definition:** fine-tune the LLM on the training data using…
  - a standard supervised objective
  - backpropagation to compute gradients
  - your favorite optimizer (e.g. Adam)
- **Pro:** fits into the standard ML recipe
- **Pro:** still works if N is large
- **Con:** backpropagation requires ~3x the memory and computation time as the forward computation
- **Con:** you might not have access to the model weights at all (e.g. because the model is proprietary)

Option B: In-context learning

- **Definition:**
  1. feed training examples to the LLM as a prompt
  2. allow the LLM to infer patterns in the training examples during inference (i.e. decoding)
  3. take the output of the LLM following the prompt as its prediction
- **Pro:** no backpropagation required and only one pass through the training data
- **Pro:** does not require model weights, only API access
- **Con:** for Transformers, a prompt (of length N) requires $O(N^2)$ time/space
- **Con:** the prompt might not fit into max context of a Transformer LM

Few-shot Learning with LLMs

Suppose you have…
- a dataset $D = \{(x_i, y_i)\}_{i=1}^N$ and N is rather small (i.e. few-shot setting)
- a very large (billions of parameters) pre-trained language model

There are two ways to "learn"

Option A: Supervised fine-tuning
- **Definition:** fine-tune the LLM on the training data using…
  - a standard supervised objective
  - backpropagation to compute gradients
  - your favorite optimizer (e.g. Adam)
- **Pro:** fits into the standard ML recipe
- **Pro:** still works if N is large
- **Con:** backpropagation requires ~3x the memory and computation time as the forward computation
- **Con:** you might not have access to the model weights at all (e.g. because the model is proprietary)

Option B: In-context learning

In this section, we consider the question:

How can we do supervised fine-tuning of a very large foundation model more efficiently?

to the LLM as a

atterns in the training

nce (i.e. decoding)

LLM following the

- **Pro:** no backpropagation required and only one pass through the training data
- **Pro:** does not require model weights, only API access
- **Con:** for Transformers, a prompt (of length N) requires $O(N^2)$ time/space
- **Con:** the prompt might not fit into max context of a Transformer LM

Fine-Tuning vs. In-Context Learning

- Why would we ever bother with fine-tuning if it's so inefficient?
- Because, even for very large LMs, fine-tuning often beats in-context learning

Method	MNLI-m (Val. Acc./%)	RTE (Val. Acc./%)
GPT-3 Few-Shot	40.6	69.0
GPT-3 Fine-Tuned	89.5	85.4

Question:

Why did fine-tuning of GPT-3 do so much better on these two tasks than few-shot learning?

Answer:

Figure from http://arxiv.org/abs/2106.09685

Fine-Tuning vs. In-Context Learning

- Why would we ever bother with fine-tuning if it's so inefficient?
- Because, even for very large LMs, fine-tuning often beats in-context learning
- In a fair comparison of fine-tuning (FT) and in-context learning (ICL), we find that FT outperforms ICL for most model sizes on RTE and MNLI

	FT						
ICL	125M	350M	1.3B	2.7B	6.7B	13B	30B
125M	−0.00	0.01	0.02	0.03	0.12	0.14	0.09
350M	−0.00	0.01	0.02	0.03	0.12	0.14	0.09
1.3B	−0.00	0.01	0.02	0.03	0.12	0.14	0.09
2.7B	−0.00	0.01	0.02	0.03	0.12	0.14	0.09
6.7B	−0.00	0.01	0.02	0.03	0.12	0.14	0.09
13B	−0.04	−0.02	−0.01	−0.00	0.09	0.11	0.05
30B	−0.11	−0.09	−0.08	−0.08	0.02	0.03	−0.02

(a) RTE

	FT						
ICL	125M	350M	1.3B	2.7B	6.7B	13B	30B
125M	−0.00	0.00	0.02	0.01	0.10	0.11	0.07
350M	−0.00	0.00	0.02	0.01	0.10	0.11	0.07
1.3B	−0.01	−0.00	0.01	0.01	0.10	0.11	0.07
2.7B	−0.01	−0.00	0.01	0.01	0.09	0.10	0.07
6.7B	−0.01	−0.01	0.01	0.00	0.09	0.10	0.06
13B	−0.03	−0.03	−0.02	−0.02	0.07	0.08	0.04
30B	−0.07	−0.07	−0.05	−0.06	0.03	0.04	0.00

(b) MNLI

Table 1: Difference between average **out-of-domain performance** of ICL and FT on RTE (a) and MNLI (b) across model sizes. We use 16 examples and 10 random seeds for both approaches. For ICL, we use the `gpt-3` pattern. For FT, we use pattern-based fine-tuning (PBFT) and select checkpoints according to in-domain performance. We perform a Welch's t-test and color cells according to whether: ICL performs significantly better than FT, FT performs significantly better than ICL. For cells without color, there is no significant difference.

Figure from https://aclanthology.org/2023.findings-acl.779.pdf

Fine-Tuning vs. In-Context Learning

- Why would we ever bother with fine-tuning if it’s so inefficient?
- Because, even for very large LMs, fine-tuning often beats in-context learning
- In a fair comparison of fine-tuning (FT) and in-context learning (ICL), we find that FT outperforms ICL for most model sizes on RTE and MNLI

ICL \ FT	125M	350M	1.3B	2.7B	6.7B	13B	30B
125M	−0.00	0.01	0.02	0.03	0.12	0.14	0.09
350M	−0.00	0.01	0.02	0.03	0.12	0.14	0.09
1.3B	−0.00	0.01	0.02	0.03	0.12	0.14	0.09
2.7B	−0.00	0.01	0.02	0.03	0.12	0.14	0.09
6.7B	−0.00	0.01	0.02	0.03	0.12	0.14	0.09
13B	−0.04	−0.02	−0.01	−0.00	0.09	0.11	0.05
30B	−0.11	−0.09	−0.08	−0.08	0.02	0.03	−0.02

(a) RTE

ICL \ FT	125M	350M	1.3B	2.7B	6.7B	13B	30B
125M	−0.00	0.00	0.02	0.01	0.10	0.11	0.07
350M	−0.00	0.00	0.02	0.01	0.10	0.11	0.07
1.3B	−0.01	−0.00	0.01	0.01	0.10	0.11	0.07
2.7B	−0.01	−0.00	0.01	0.01	0.09	0.10	0.07
6.7B	−0.01	−0.01	0.01	0.00	0.09	0.10	0.06
13B	−0.03	−0.03	−0.02	−0.02	0.07	0.08	0.04
30B	−0.07	−0.07	−0.05	−0.06	0.03	0.04	0.00

(b) MNLI

Table 1: Difference between average **out-of-domain performance** of ICL and FT on RTE (a) ar
model sizes. We use 16 examples and 10 random seeds for both approaches. For ICL, we use
For FT, we use pattern-based fine-tuning (PBFT) and select checkpoints according to in-do
We perform a Welch’s t-test and color cells according to whether: ICL performs significantly
performs significantly better than ICL. For cells without color, there is no significant difference

At least this was the general wisdom in 2023.

We might have a different story to tell now that it’s 2024. (See Lecture 19)

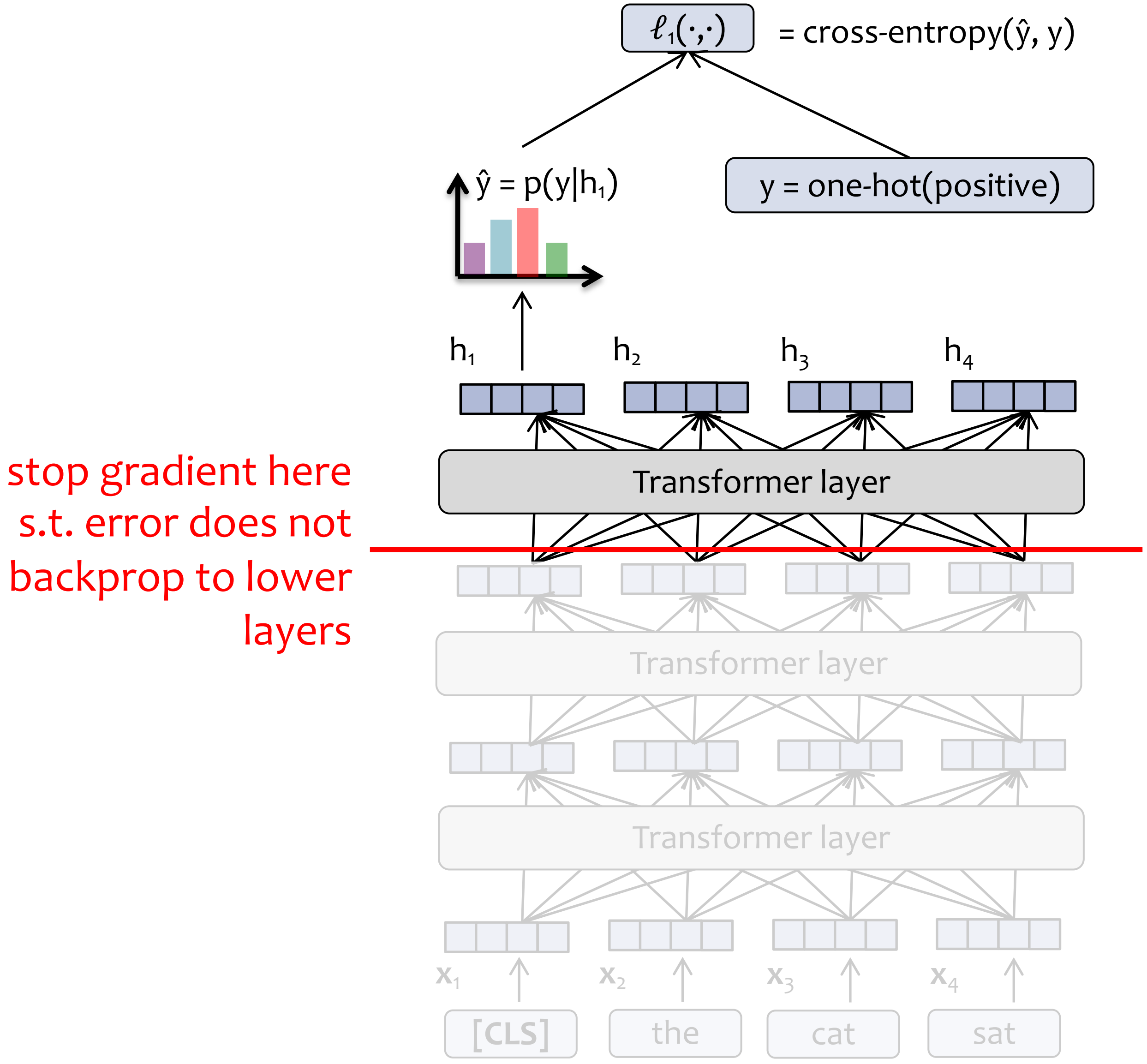
Figure from https://aclanthology.org/2023.findings-acl.779.pdf

Parameter Efficient Fine-Tuning

- **Goal**: perform fine-tuning of fewer parameters, but achieve performance on a downstream task that is comparable to fine-tuning of all parameters
- **Various approaches**:
  – **Subset**: Pick a subset of the parameters and fine-tune only those (e.g. only the top K layers of a K+L layer deep neural network)
  – **Adapters**: add additional layers that have few parameters and tune only the parameters of those layers, keeping all others fixed
  – **Prefix Tuning**: for a Transformer LM, pretend as if there exist many tokens that came before your sequence and tune the keys/values corresponding to those tokens
  – **LoRA**: learn a small delta for the each of the parameter matrices with the delta chosen to be low rank

Fine-Tuning the Top Layers Only

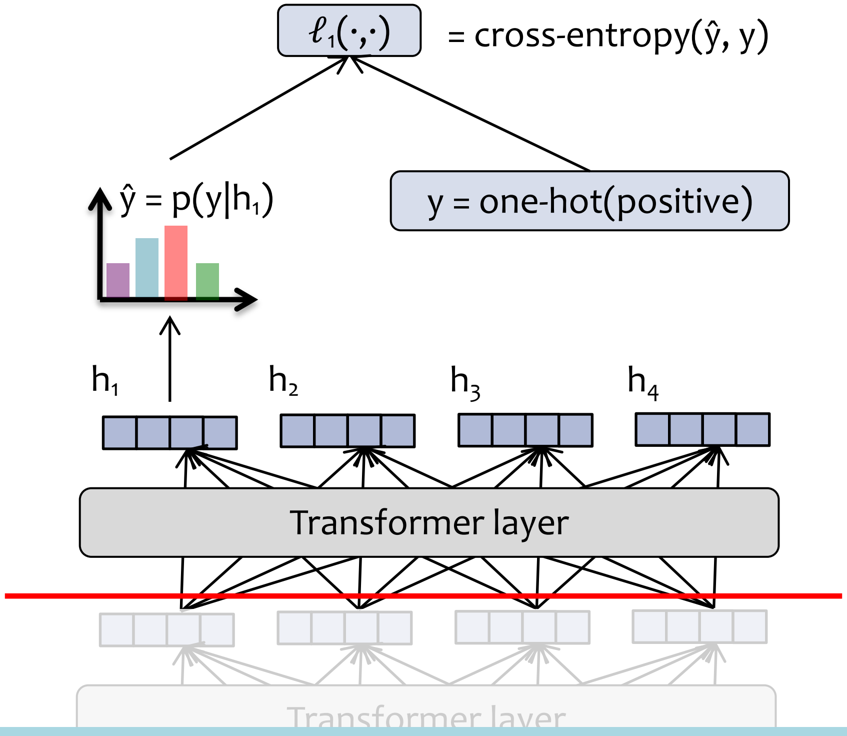
- Simple baseline for PEFT:
  - keep all parameters fixed except for the top K layers
  - gradients only need to flow through K layers instead of K+L total layers
  - reduced memory usage b/c we don't need to store the adjoints (gradient of the loss with respect to each parameter) for the full computation graph
- Can easily be applied to most deep neural networks



Fine-Tuning the Top Layers Only

- Simple baseline for PEFT:
  - keep all parameters fixed except for the top K layers
  - gradients only need to flow through K layers instead of K+L total layers
  - reduced memory usage b/c we don't need to store the adjoints (gradient of the loss with respect to each parameter) for the full

stop gradient here s.t. error does not backprop to lower layers



Question:

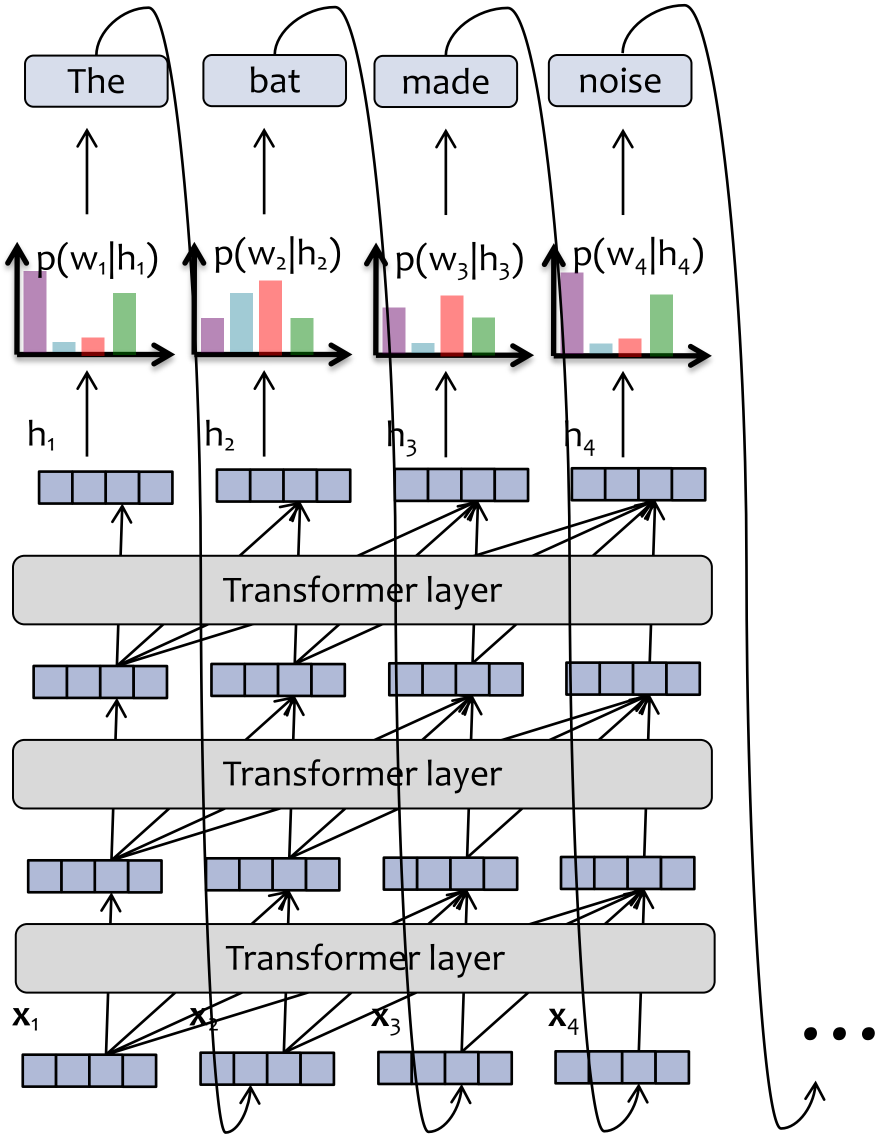
Why does this work at all?

Shouldn't it do a poor job fitting the underlying trends in our data?

Answer:

ADAPTERS

Decoder-only Transformer



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

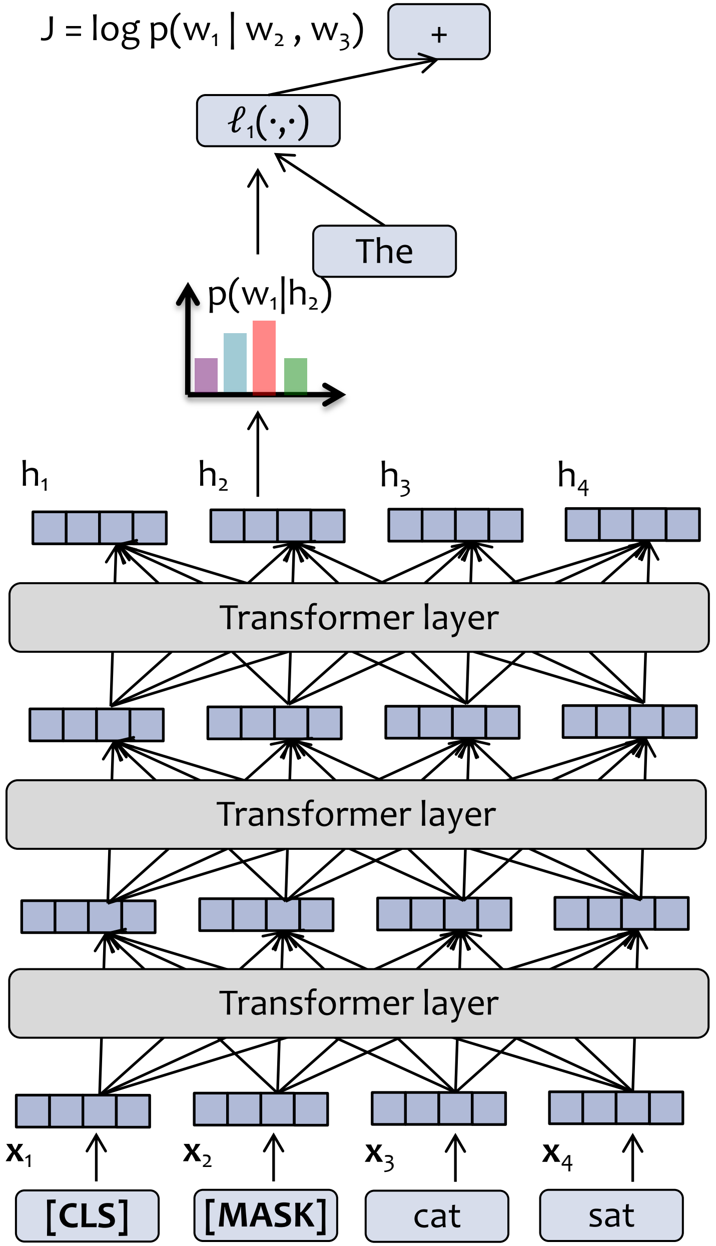
The language model part is just like an RNN-LM.

Encoder-only Transformer

BERT popularized this encoder-only Transformer architecture and style of pretraining

MLM Pretraining:

- Rather than trying to predict the next word from the previous ones…
- …mask out a word (or a few words) and predict the missing words from the remaining ones



…

Each layer of an encoder-only Transformer consists of several **sublayers:**

1. non-causal attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks at the the hidden vectors of **all timesteps in the previous layer.**

The distribution over words is used for **masked language model (MLM) pre-training** (cf. BERT)

Adapters Module

- An adapter layer is simply a feed-forward neural network with one hidden layer, and a residual connection
- For input dimension, d, the adapter layer also has output dimension d, but bottlenecks to a lower dimension m in the middle

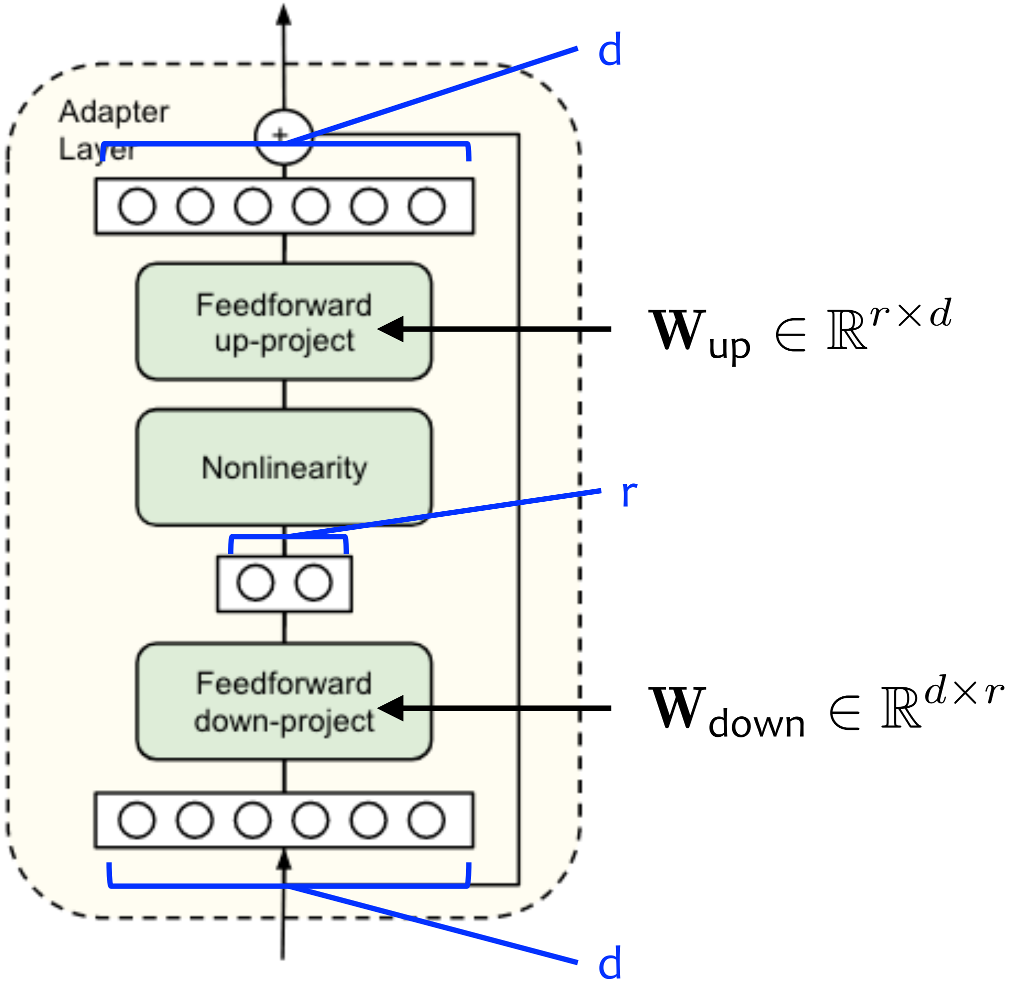
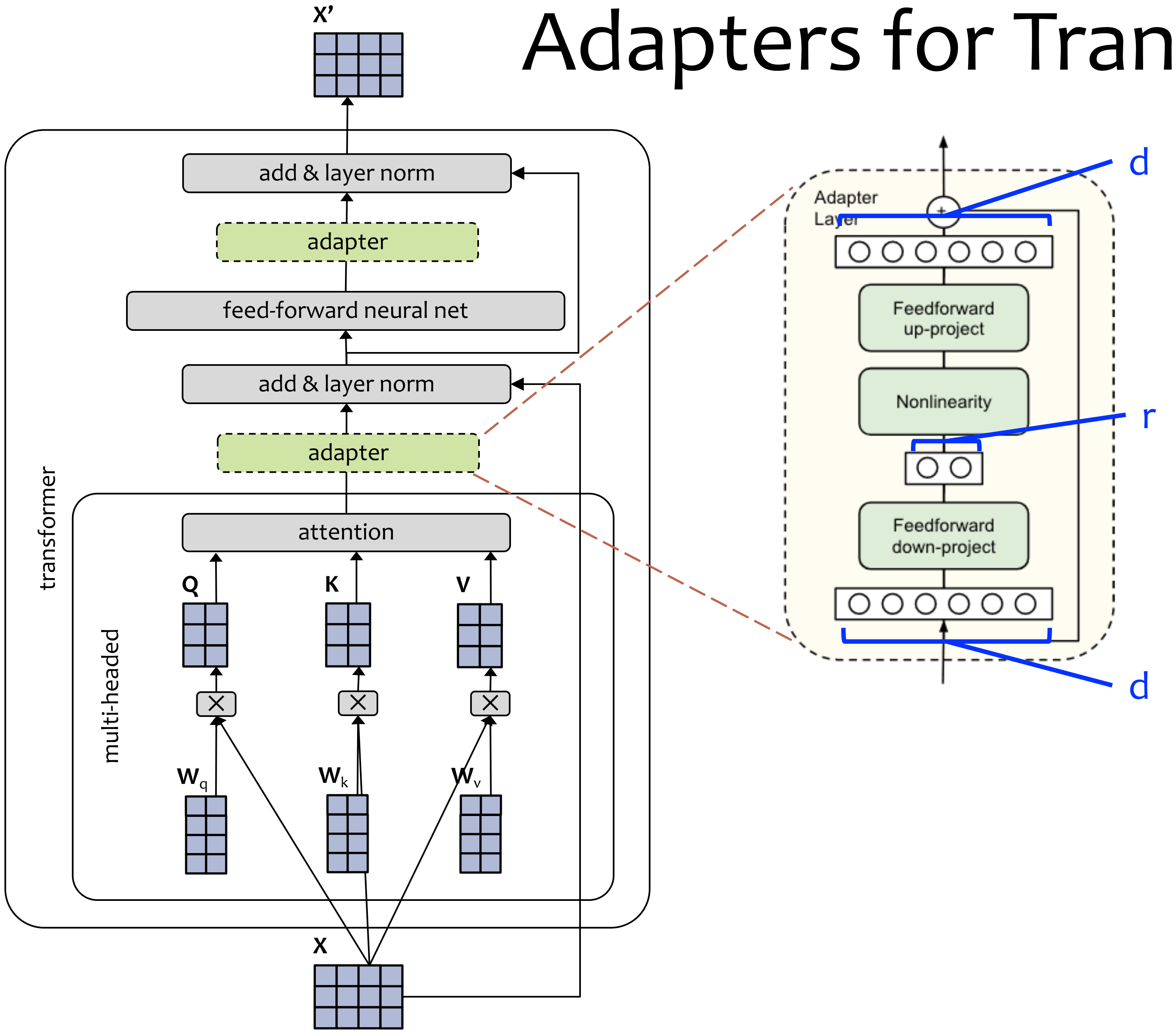


Figure from https://arxiv.org/pdf/1902.00751

Adapters for Transformer



- In practice, r is chosen s.t. r << d and the adapter layers contain only 0.5% – 8% of the total parameters
- When added to a deep neural network (e.g. Transformer) all the other parameters of the pretrained model are kept fixed, and only the adapter layer parameters are fine-tuned
- Interesting: it works even though the grey modules are kept fixed!

Figure inspired by He et al. (2022) https://arxiv.org/pdf/2110.04366

Adapter Results

- **Pretrained Model:** BERT-Large
- **Baseline Method:** fine-tune only the top K layers of BERT-Large
- Adapters achieve nearly the performance (i.e. 0% delta) of full fine-tuning but with substantially fewer parameters
- Sometimes adapters even outperform full fine-tuning

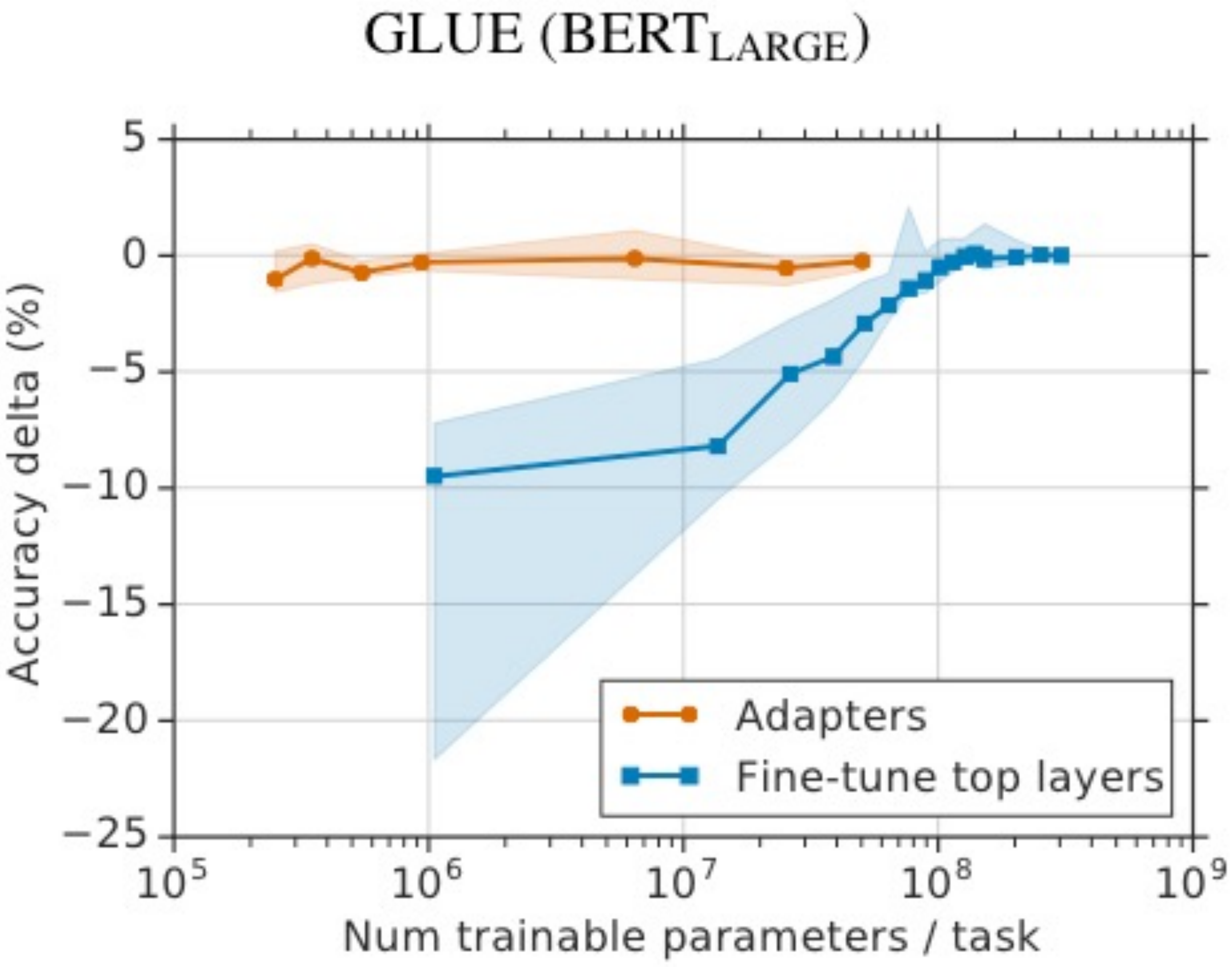


Figure from https://arxiv.org/pdf/1902.00751

Adapter Results

- **Pretrained Model:** BERT-Large
- **Baseline Method:** fine-tune only the top K layers of BERT-Large
- Adapters achieve nearly the performance (i.e. 0% delta) of full fine-tuning but with substantially fewer parameters
- Sometimes adapters even outperform full fine-tuning

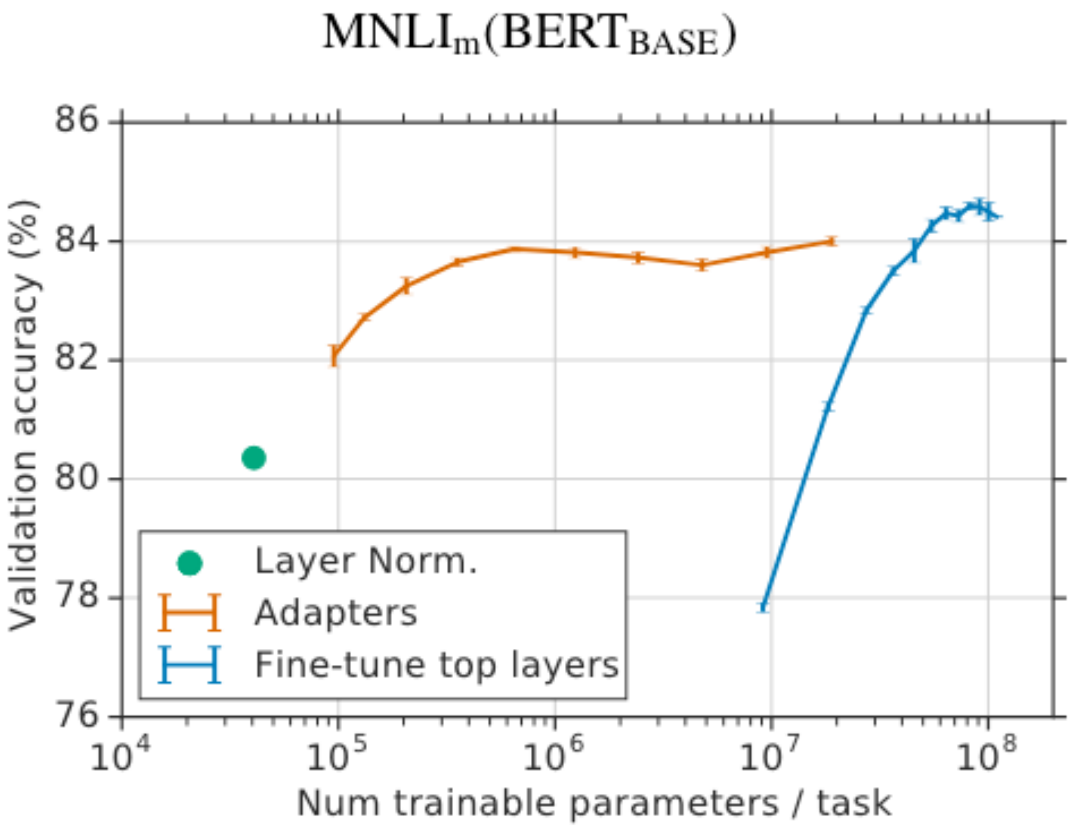


Figure from https://arxiv.org/pdf/1902.00751

Adapter Results

- **Pretrained Model:** BERT-Large
- **Baseline Method:** fine-tune only the top K layers of BERT-Large
- Adapters achieve nearly the performance (i.e. 0% delta) of full fine-tuning but with substantially fewer parameters
- Sometimes adapters even outperform full fine-tuning

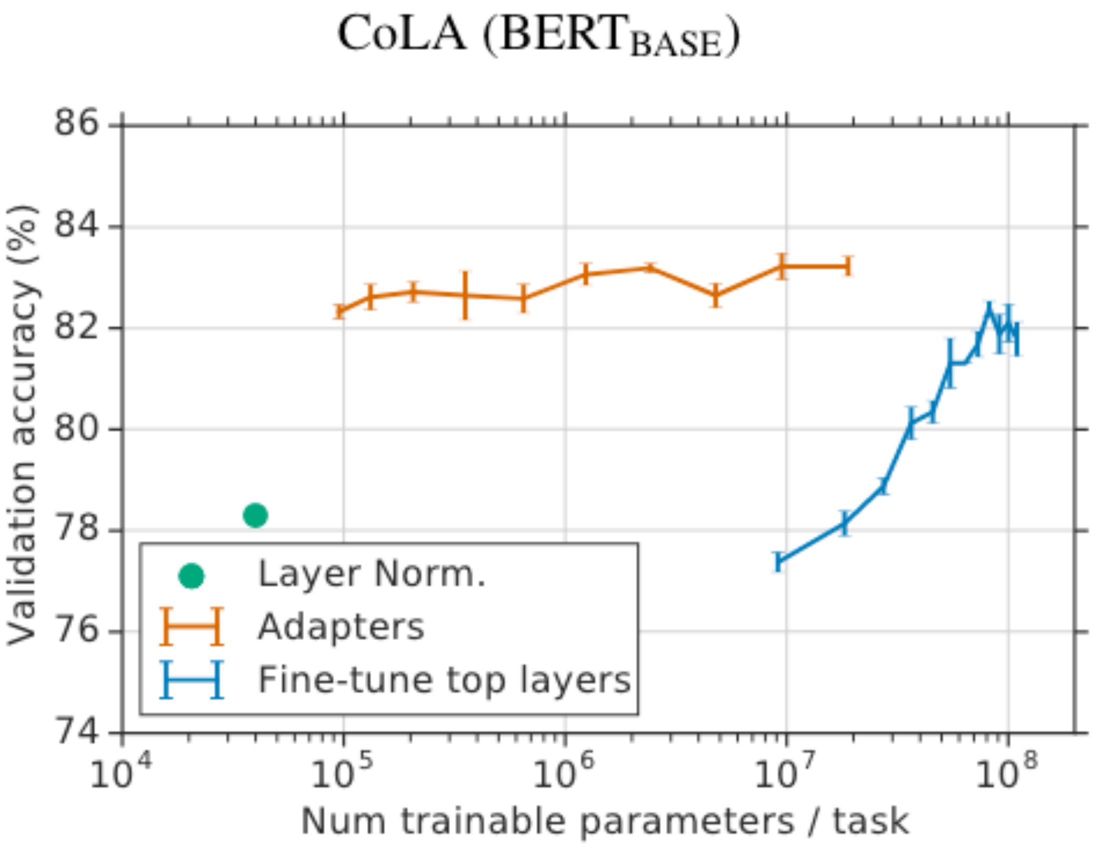
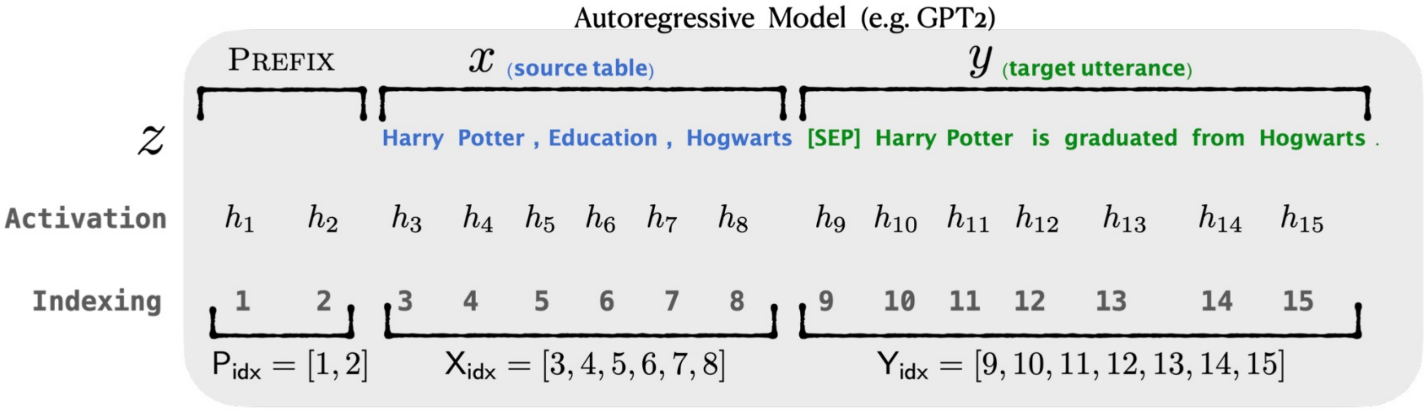


Figure from https://arxiv.org/pdf/1902.00751

PROMPT TUNING & PREFIX TUNING

Prefix Tuning



For a Transformer, we will say the *activation* of token i in some layer/head is given by its key/value vectors:

$$\mathbf{h}_i = [\mathbf{k}_i^T, \mathbf{v}_i^T]^T$$

1. inject (dummy) prefix tokens, indexed by P_{idx}, before the real tokens
2. represent i'th prefix token's activation by trainable parameters:
   $$\mathbf{h}_i = \mathbf{P}_\theta[i, :]$$
3. for each i let
   $$\mathbf{P}_\theta[i, :] = MLP(\mathbf{Q}_\theta[i, :])$$
   because having $\mathbf{Q}_\theta$ of lower dimension than $\mathbf{P}_\theta$ improves stability during training
4. during training, keep all Transformer parameters fixed, except for θ

Figure from http://arxiv.org/abs/2101.00190

Prefix Tuning

Also works for encoder-only Transformer models, but we inject prefix tokens before both the source tokens **x** and the target tokens **y**

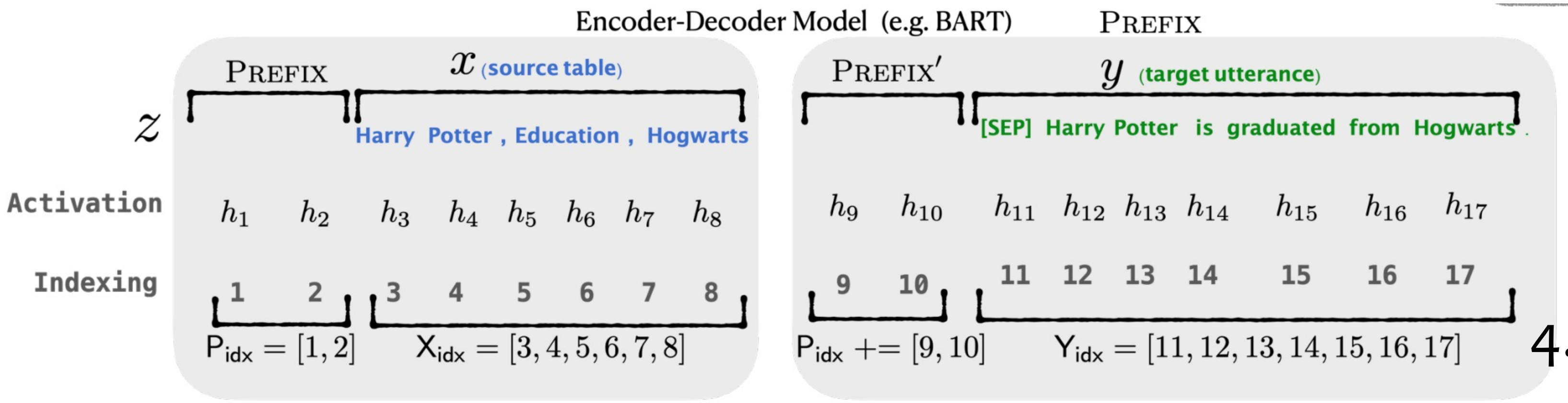
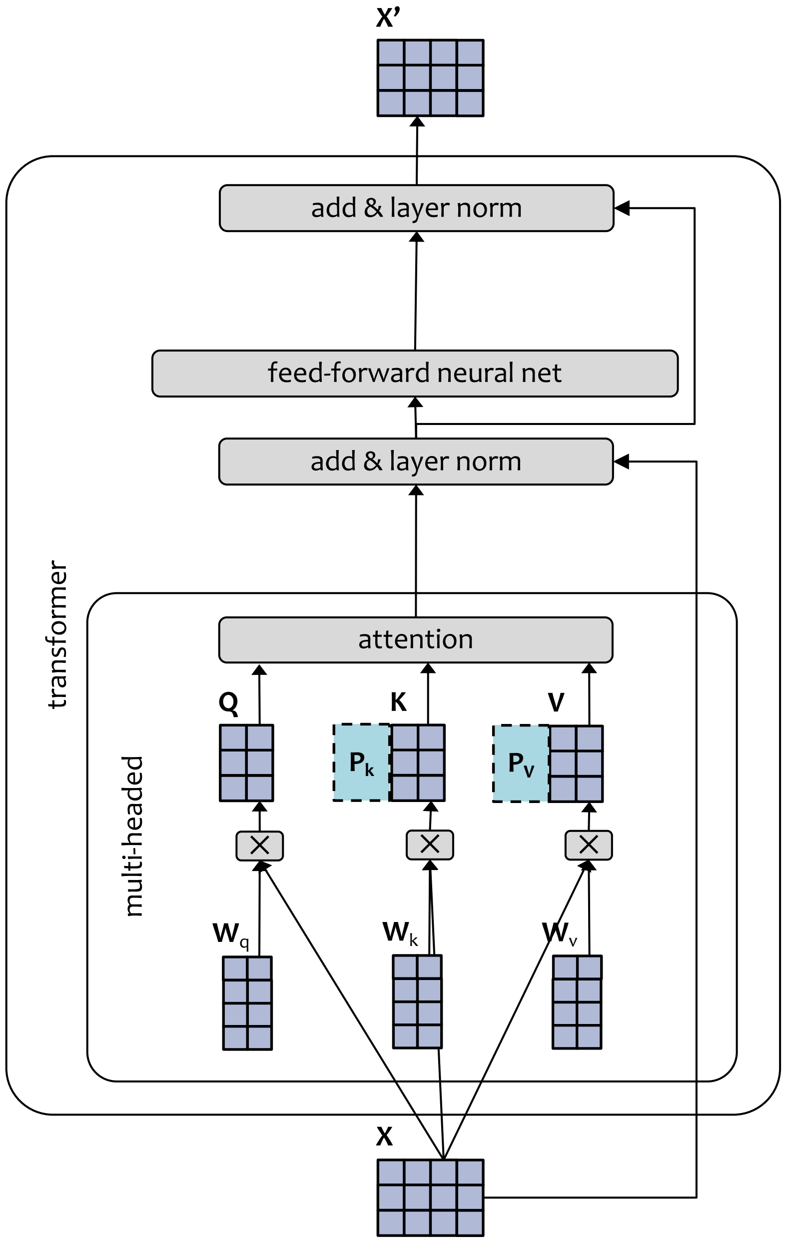


Figure from http://arxiv.org/abs/2101.00190

1. inject (dummy) prefix tokens, indexed by P_{idx}, before the real tokens
2. represent i'th prefix token's activation by trainable parameters:
   $\mathbf{h}_i = \mathbf{P}_\theta[i, :]$
3. for each i let
   $\mathbf{P}_\theta[i, :] = \mathrm{MLP}(\mathbf{Q}_\theta[i, :])$
   because having $\mathbf{Q}_\theta$ of lower dimension than $\mathbf{P}_\theta$ improves stability during training
4. during training, keep all Transformer parameters fixed, except for θ

Prefix Tuning



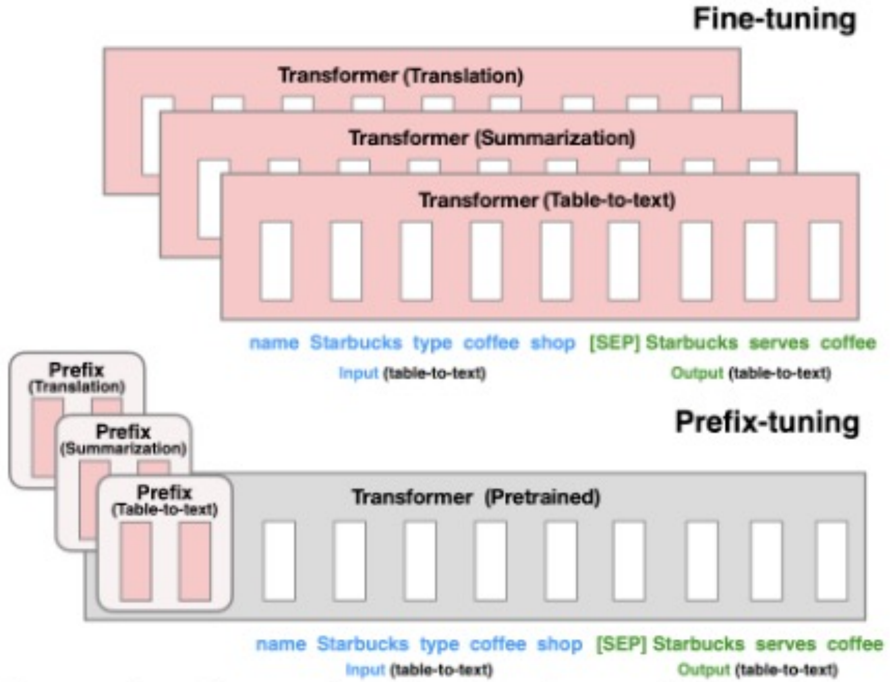


Figure 1: Fine-tuning (top) updates all Transformer parameters (the red Transformer box) and requires storing a full model copy for each task. We propose prefix-tuning (bottom), which freezes the Transformer parameters and only optimizes the prefix (the red prefix blocks). Consequently, we only need to store the prefix for each task, making prefix-tuning modular and space-efficient. Note that each vertical block denote transformer activations at one time step.

Figure inspired by He et al. (2022) https://arxiv.org/pdf/2110.04366

Figure from http://arxiv.org/abs/2101.00190

LOW-RANK ADAPTATION (LORA)

How large are LLMs?

Recall…

Comparison of some recent **large language models** (LLMs)

Model	Creators	Year of release	Training Data (# tokens)	Model Size (# parameters)
GPT-2	OpenAI	2019	~10 billion (40Gb)	1.5 billion
GPT-3 (cf. ChatGPT)	OpenAI	2020	300 billion	175 billion
PaLM	Google	2022	780 billion	540 billion
Chinchilla	DeepMind	2022	1.4 trillion	70 billion
LaMDA (cf. Bard)	Google	2022	1.56 trillion	137 billion
LLaMA	Meta	2023	1.4 trillion	65 billion
LLaMA-2	Meta	2023	2 trillion	70 billion
GPT-4	OpenAI	2023	?	? (1.76 trillion)
Gemini (Ultra)	Google	2023	?	? (1.5 trillion)
LLaMA-3	Meta	2024	15 trillion	405 billion

How large are the linear layers in a Transformer?

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-paramet[illegible] (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 [illegible]ion tokens.

Size of linear layer in GPT-3:
12k * 12k

Figure from https://arxiv.org/pdf/2005.14165.pdf

Fine-Tuning LLMs without Regularization

Method	MNLI-m (Val. Acc./%)	RTE (Val. Acc./%)
GPT-3 Few-Shot	40.6	69.0
GPT-3 Fine-Tuned	89.5	85.4

Question:

Why don't LLMs overfit when we fine-tune them without regularization?

Hypothesis:

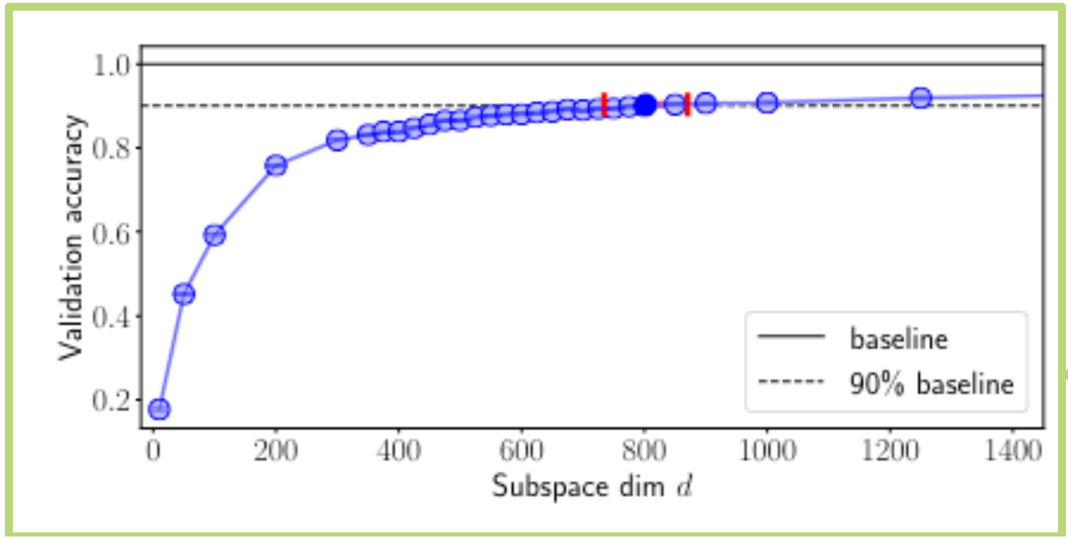
They are intrinsically low dimensional

Figure from http://arxiv.org/abs/1804.08838

Intrinsic Dimensionality

Motivation

- Maybe the number of parameters in a model is **not** a great measure of how many degrees of freedom are needed to successfully learn some problem
- How could we measure the number of degrees of freedom you **really** need?



Intrinsic Dimension

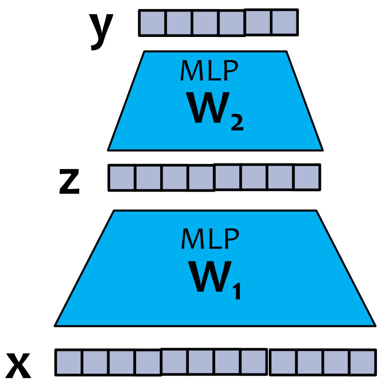
Definition from Li et al. (2018):

- Learn a neural network with *D* parameters in a random lower dimensional subspace, *d*
- Then repeat, gradually increasing the dimensionality, *d*
- Let the **intrinsic dimension** be the value of *d* when good solutions (above 90% threshold of full parameterization) start to appear

For MNIST digit recognition, original neural network has *D*=199,210 parameters but the intrinsic dimension is only *d*=750

Intrinsic Dimensionality

How do we learn in a low dimensional subspace?



$$\theta^{(D)} = \text{concat}(\text{flatten}(\mathbf{W}_1), \text{flatten}(\mathbf{W}_2))$$

Standard optimization, which we will refer to hereafter as the *direct* method of training, entails evaluating the gradient of a loss with respect to $\theta^{(D)}$ and taking steps directly in the space of $\theta^{(D)}$. To train in a random subspace, we instead define $\theta^{(D)}$ in the following way:

$$\theta^{(D)} = \theta_0^{(D)} + P\theta^{(d)} \tag{2}$$

where P is a randomly generated $D \times d$ projection matrix[1] and $\theta^{(d)}$ is a parameter vector in a generally smaller space $\mathbb{R}^d$. $\theta_0^{(D)}$ and P are randomly generated and frozen (not trained), so the system has only d degrees of freedom. We initialize $\theta^{(d)}$ to a vector of all zeros, so initially $\theta^{(D)} = \theta_0^{(D)}$.

Figure from http://arxiv.org/abs/1804.08838

Intrinsic Dimensionality

- Using similar techniques, Aghajanyan et al. (2020) measure the intrinsic dimension of LLMs
- Empirical results suggest that pre-training finds parameters that have low intrinsic dimensionality
- Number of parameters:
  - BERT-Base: 110 million
  - BERT-Large: 345 million

	SAID		DID	
Model	MRPC	QQP	MRPC	QQP
BERT-Base	1608	8030	1861	9295
BERT-Large	1037	1200	2493	1389
RoBERTa-Base	896	896	1000	1389
RoBERTa-Large	**207**	**774**	322	**774**

Table 1: Estimated d_{90} intrinsic dimension for a set of sentence prediction tasks and common pre-trained models. We present both the *SAID* and *DID* methods.

Figure from http://arxiv.org/abs/1804.08838

LoRA

- **Motivation #1:**
  "We take inspiration from Li et al. (2018a); Aghajanyan et al. (2020) which show that the learned over-parametrized models in fact reside on a low intrinsic dimension."
- **Motivation #2:**
  Directly optimizing the prompt, as in prefix tuning, leads to non-monotonic changes in performance as the number of parameters increases (we want more parameters to mean better performance!)
- **Motivation #3:**
  Adapters and related methods introduce inference latency at test time that is non-trivial!

Batch Size Sequence Length $\|\Theta\|$	32 512 0.5M	16 256 11M	1 128 11M
Fine-Tune/LoRA	1449.4±0.8	338.0±0.6	19.8±2.7
Adapter^{L}	1482.0±1.0 (+2.2%)	354.8±0.5 (+5.0%)	23.9±2.1 (+20.7%)
Adapter^{H}	1492.2±1.0 (+3.0%)	366.3±0.5 (+8.4%)	25.8±2.2 (+30.3%)

Table 1: Infernece latency of a single forward pass in GPT-2 medium measured in milliseconds, averaged over 100 trials. We use an NVIDIA Quadro RTX8000. "$|\Theta|$" denotes the number of trainable parameters in adapter layers. Adapter^{L} and Adapter^{H} are two variants of adapter tuning, which we describe in Section 5.1. The inference latency introduced by adapter layers can be significant in an online, short-sequence-length scenario. See the full study in Appendix B.

LoRA

Key Idea

- Keep the original pretrained parameters $\mathbf{W}_0$ fixed during fine-tuning
- Learn an additive modification to those parameters $\Delta\mathbf{W}$
- Define $\Delta\mathbf{W}$ via a low rank decomposition:

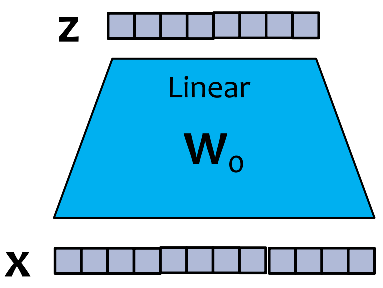
$$\Delta\mathbf{W} = \mathbf{B}\mathbf{A}$$

where **BA** has rank r, which is **much less** than the input dimension k or the output dimension d

Standard Linear Layer

$$\mathbf{z} = \mathbf{W}_0\mathbf{x}$$

$$\mathbf{W}_0 \in \mathbb{R}^{d\times k}, \mathbf{x} \in \mathbb{R}^{k}, \mathbf{z} \in \mathbb{R}^{d}$$

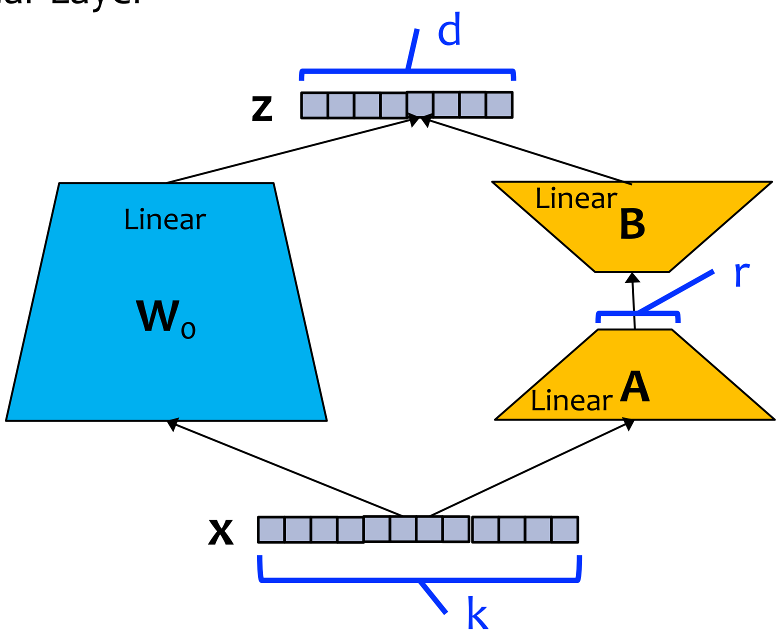


LoRA Linear Layer

$$\mathbf{z} = \mathbf{W}_0\mathbf{x} + \mathbf{B}\mathbf{A}\mathbf{x}$$
$$= (\mathbf{W}_0 + \mathbf{B}\mathbf{A})\mathbf{x}$$

$$\mathbf{W}_0 \in \mathbb{R}^{d\times k},$$
$$\mathbf{A} \in \mathbb{R}^{r\times k}, \mathbf{B} \in \mathbb{R}^{d\times r}$$

where $r << \min(d, k)$



LoRA

Initialization

- We initialize the trainable parameters:

  $A_{ij} \sim \mathcal{N}(0, \sigma^2), \forall i, j$

  $\mathbf{B} = 0$

- This ensures that, at the start of fine tuning, the parameters have their pretrained values:

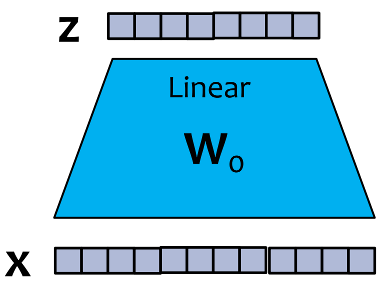
  $\Delta \mathbf{W} = \mathbf{BA} = 0$

  $\mathbf{W}_0 + \mathbf{BA} = \mathbf{W}_0$

Standard Linear Layer

$$\mathbf{z} = \mathbf{W}_0 \mathbf{x}$$

$$\mathbf{W}_0 \in \mathbb{R}^{d \times k}, \mathbf{x} \in \mathbb{R}^k, \mathbf{z} \in \mathbb{R}^d$$



LoRA Linear Layer

$$\mathbf{z} = \mathbf{W}_0 \mathbf{x} + \mathbf{BAx}$$
$$= (\mathbf{W}_0 + \mathbf{BA})\mathbf{x}$$

$$\mathbf{W}_0 \in \mathbb{R}^{d \times k},$$
$$\mathbf{A} \in \mathbb{R}^{r \times k}, \mathbf{B} \in \mathbb{R}^{d \times r}$$

where $r << \min(d, k)$

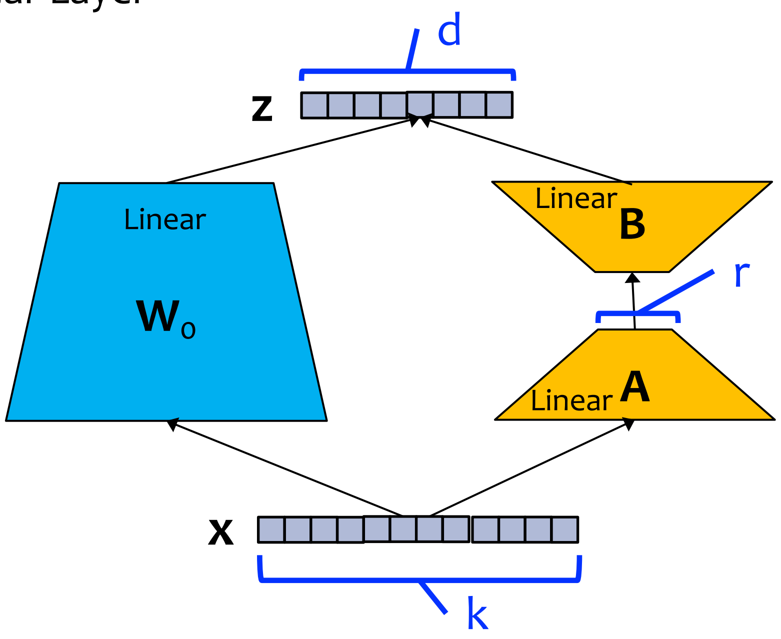


Figure inspired by He et al. (2022)

LoRA

Hot Swapping Parameters

- $\mathbf{W}_0$ and **BA** have the same dimension, so we can ”swap” the LoRA parameters in and out of a Standard Linear Layer
- To get a Standard Linear Layer with parameters **W** that includes our LoRA fine tuning:

$$\mathbf{W} \leftarrow \mathbf{W}_0 + \mathbf{B}\mathbf{A}$$

- To remove the LoRA fine tuning from that Standard Linear Layer:

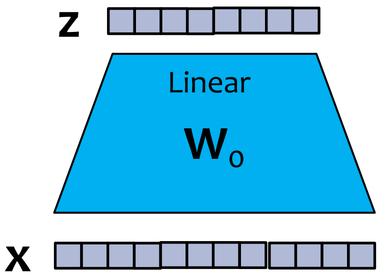
$$\mathbf{W} \leftarrow \mathbf{W} - \mathbf{B}\mathbf{A} = \mathbf{W}_0$$

- If we do LoRA training on two tasks s.t. the parameters **B’A’** are for task 1 and **B’’A’’** are for task 2, then we can swap back and forth between them

Standard Linear Layer

$$\mathbf{z} = \mathbf{W}_0\mathbf{x}$$

$$\mathbf{W}_0 \in \mathbb{R}^{d \times k}, \mathbf{x} \in \mathbb{R}^{k}, \mathbf{z} \in \mathbb{R}^{d}$$



LoRA Linear Layer

$$\mathbf{z} = \mathbf{W}_0\mathbf{x} + \mathbf{B}\mathbf{A}\mathbf{x}$$
$$= (\mathbf{W}_0 + \mathbf{B}\mathbf{A})\mathbf{x}$$

$$\mathbf{W}_0 \in \mathbb{R}^{d \times k},$$
$$\mathbf{A} \in \mathbb{R}^{r \times k}, \mathbf{B} \in \mathbb{R}^{d \times r}$$
where $r << \min(d, k)$

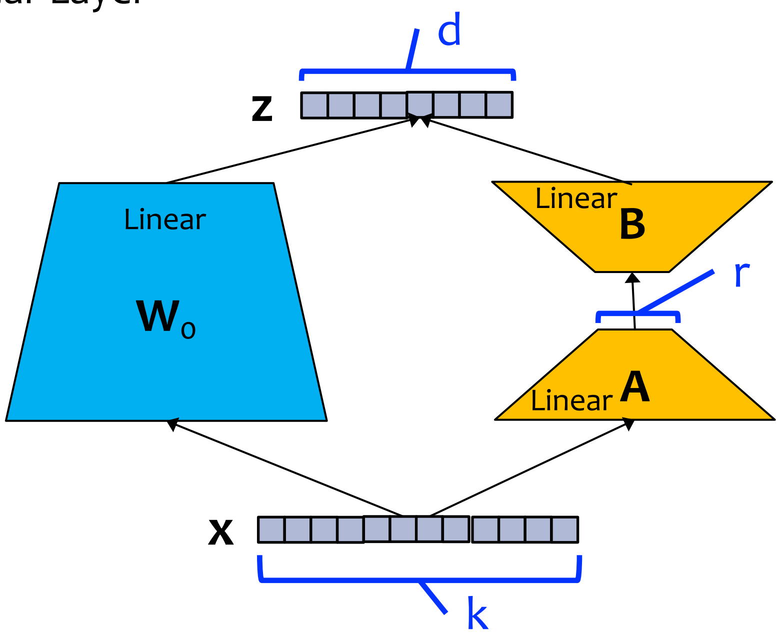
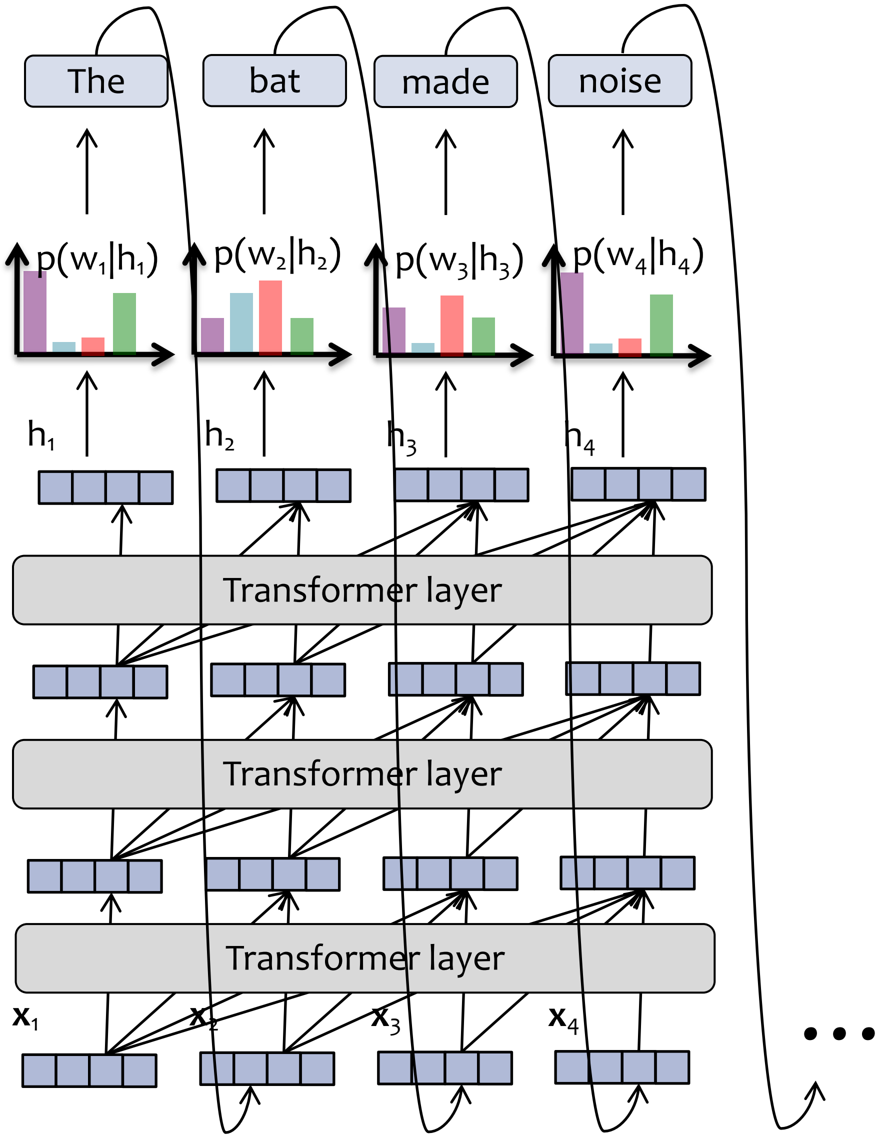


Figure inspired by He et al. (2022)

Transformer Language Model



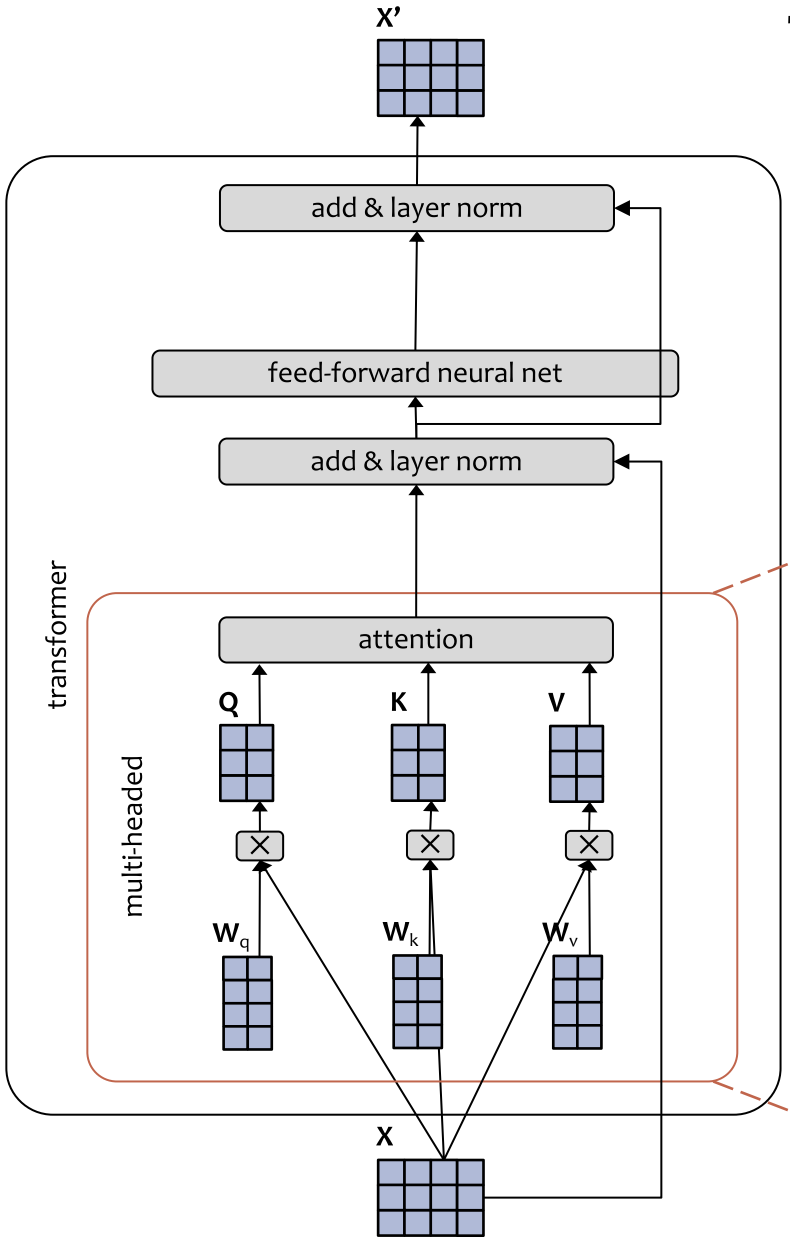
Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

The language model part is just like an RNN-LM.

Transformer Layer



multi-headed attention

$$\mathbf{X}'' = \text{concat}(\mathbf{X}'^{(1)}, \ldots, \mathbf{X}'^{(h)})$$

$$\mathbf{X}'^{(i)} = \text{softmax}\left(\frac{\mathbf{Q}^{(i)}(\mathbf{K}^{(i)})^T}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}^{(i)}$$

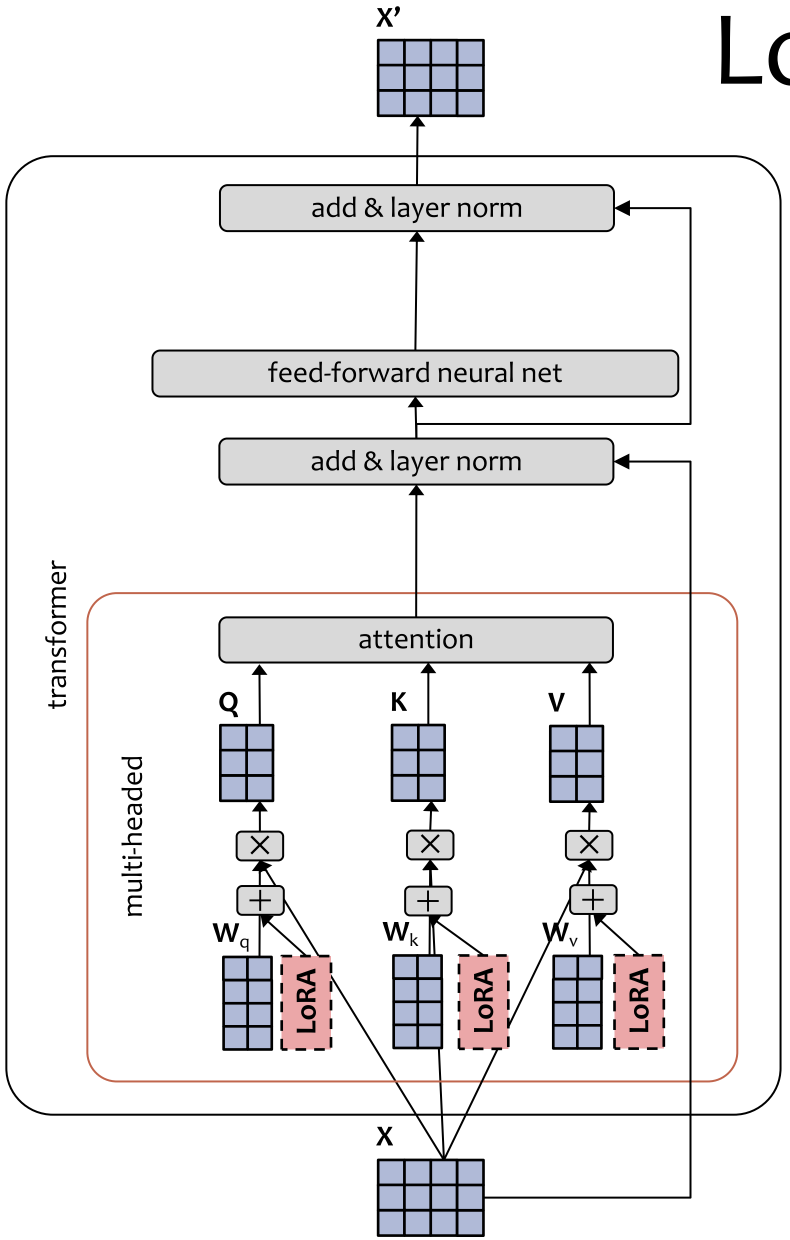
$$\mathbf{Q}^{(i)} = \mathbf{X}\mathbf{W}_q^{(i)}$$

$$\mathbf{K}^{(i)} = \mathbf{X}\mathbf{W}_k^{(i)}$$

$$\mathbf{V}^{(i)} = \mathbf{X}\mathbf{W}_v^{(i)}$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N]^T$$

Figure inspired by He et al. (2022)



LoRA for Transformer

- LoRA linear layers could replace *every* linear layer in the Transformer layer
- But the original paper only applies LoRA to the attention weights

LoRA Linear Layer

$$\mathbf{z} = \mathbf{W}_0\mathbf{x} + \mathbf{B}\mathbf{A}\mathbf{x}$$
$$= (\mathbf{W}_0 + \mathbf{B}\mathbf{A})\mathbf{x}$$

$\mathbf{W}_0 \in \mathbb{R}^{d \times k}$,

$\mathbf{A} \in \mathbb{R}^{r \times k}, \mathbf{B} \in \mathbb{R}^{d \times r}$

where $r << \min(d, k)$

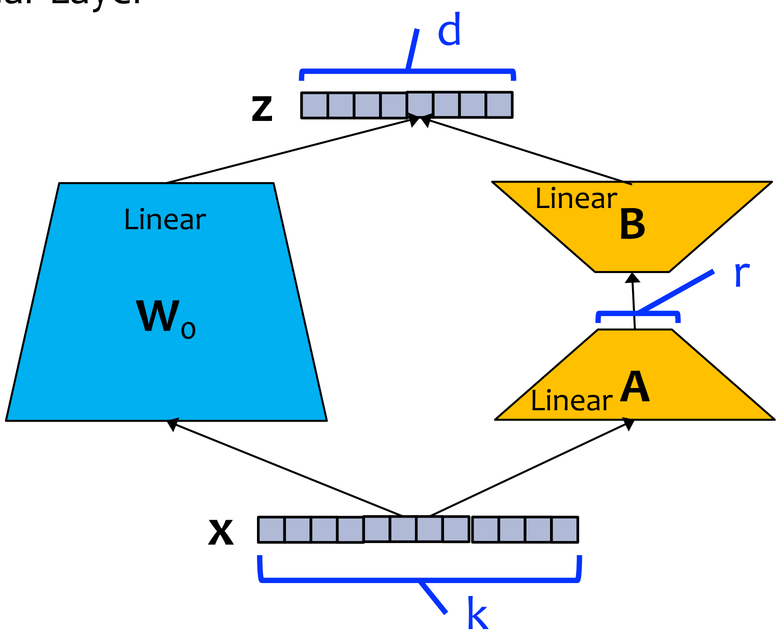
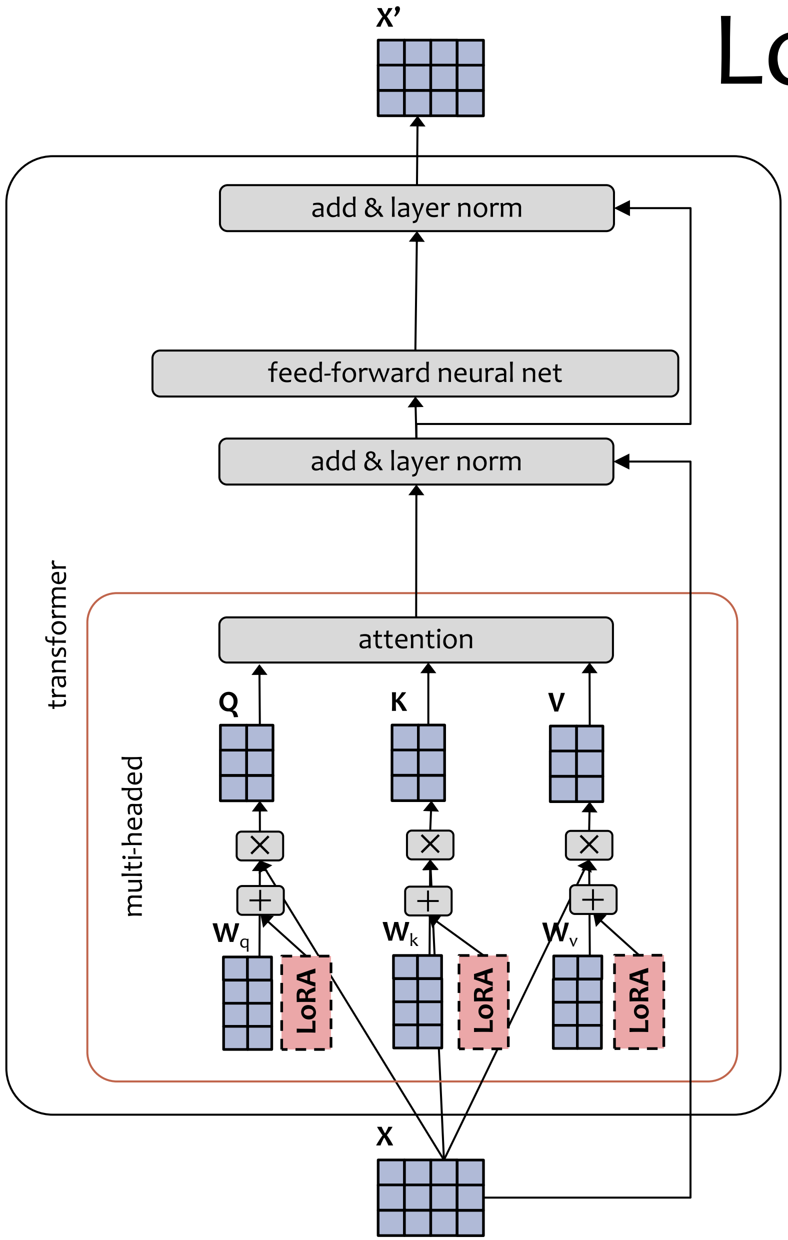


Figure inspired by He et al. (2022)

LoRA for Transformer



- LoRA linear layers could replace *every* linear layer in the Transformer layer
- But the original paper only applies LoRA to the attention weights

$$\mathbf{Q} = \text{LoRALinear}(\mathbf{X}; \mathbf{W}_q, \mathbf{A}_q, \mathbf{B}_q)$$
$$\mathbf{K} = \text{LoRALinear}(\mathbf{X}; \mathbf{W}_k, \mathbf{A}_k, \mathbf{B}_k)$$
$$\mathbf{V} = \text{LoRALinear}(\mathbf{X}; \mathbf{W}_v, \mathbf{A}_v, \mathbf{B}_v)$$

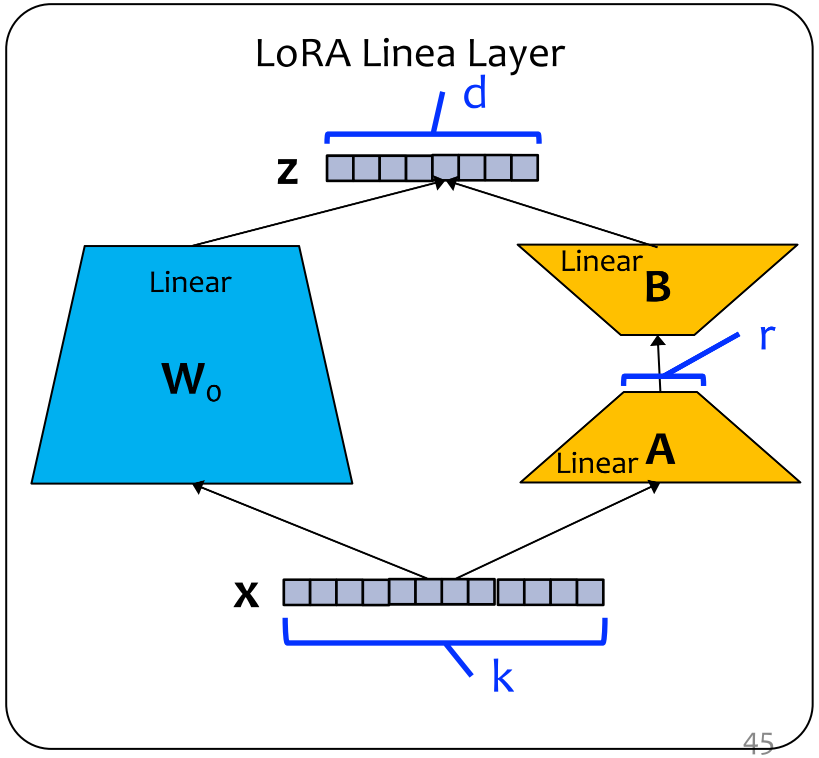
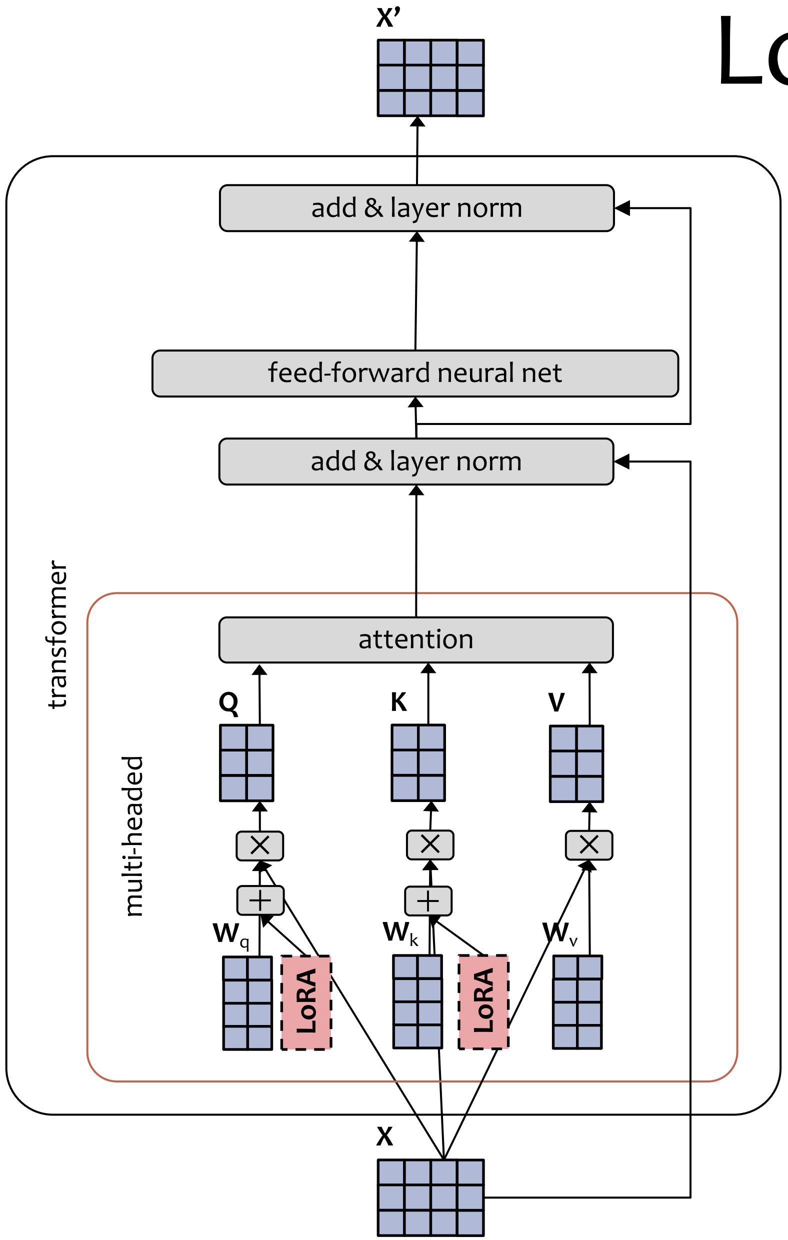


Figure inspired by He et al. (2022)

LoRA for Transformer



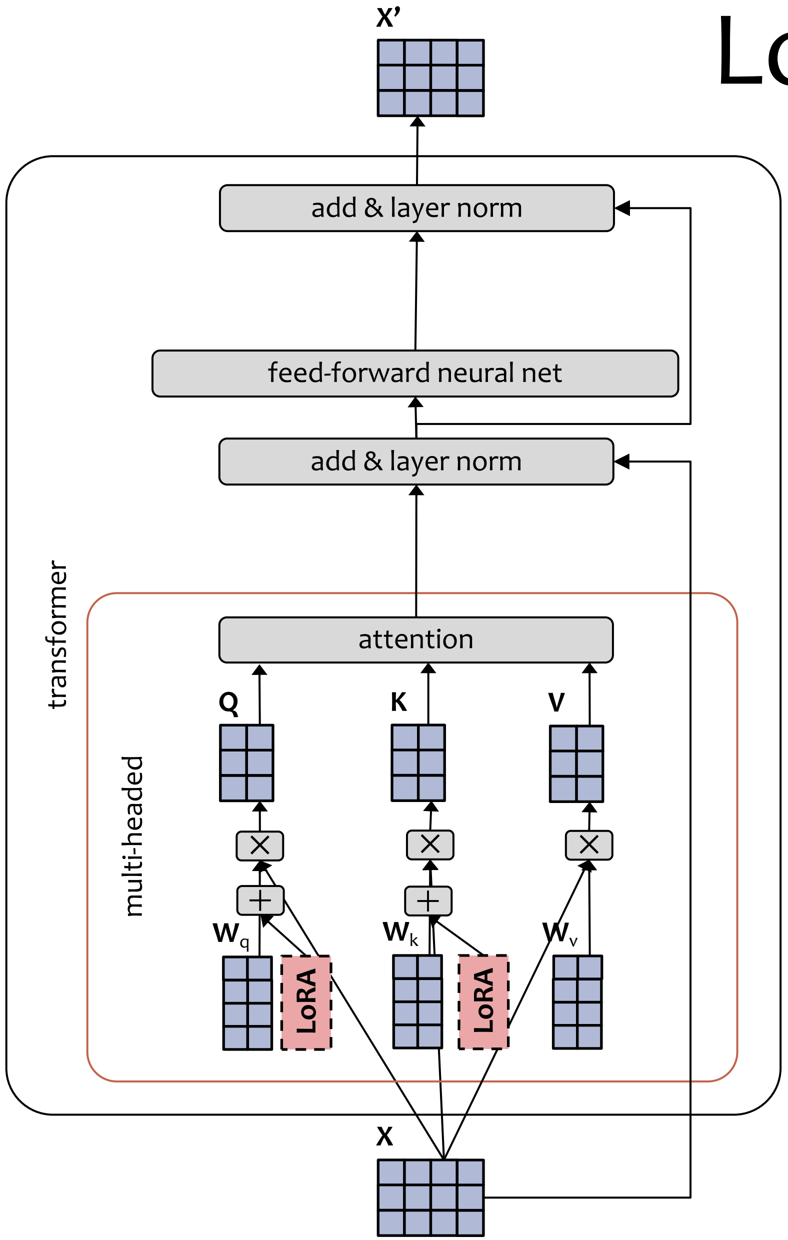
- LoRA linear layers could replace *every* linear layer in the Transformer layer
- But the original paper only applies LoRA to the attention weights
- Empirically, for GPT-3, they also find that it is most efficient to include LoRA **only on the query and value** linear layers

	# of Trainable Parameters = 18M						
Weight Type Rank r	W_q 8	W_k 8	W_v 8	W_o 8	W_q, W_k 4	W_q, W_v 4	W_q, W_k, W_v, W_o 2
WikiSQL ($\pm 0.5\%$)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI ($\pm 0.1\%$)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

Table 5: Validation accuracy on WikiSQL and MultiNLI after applying LoRA to different types of attention weights in GPT-3, given the same number of trainable parameters. Adapting both W_q and W_v gives the best performance overall. We find the standard deviation across random seeds to be consistent for a given dataset, which we report in the first column.

Table from http://arxiv.org/abs/2106.09685

LoRA for Transformer



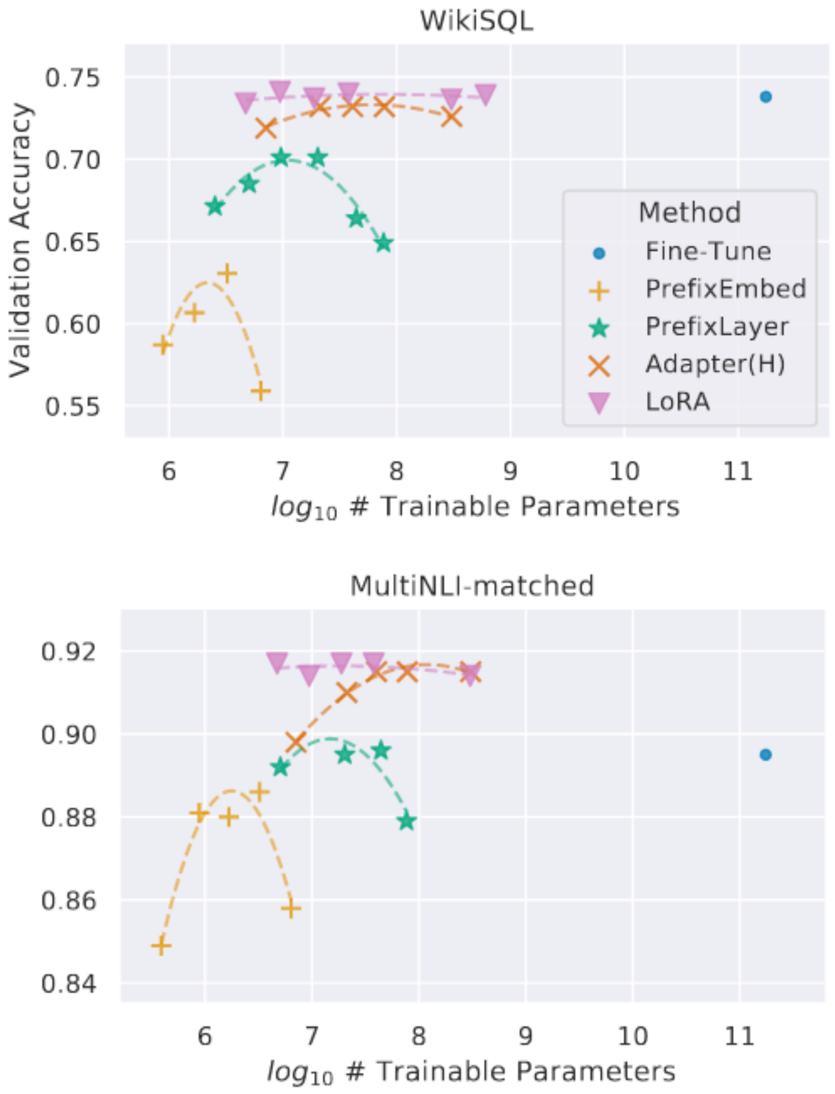
- LoRA linear layers could replace *every* linear layer in the Transformer layer
- But the original paper only applies LoRA to the attention weights
- Empirically, for GPT-3, they also find that it is most efficient to include LoRA **only on the query and value** linear layers
- During training only the new LoRA parameters are fine-tuned, all the other parameters are kept fixed

Table from http://arxiv.org/abs/2106.09685

LoRA Results

Takeaways

- Applied to GPT-3, LoRA achieves performance almost as good as full fine-tuning, but with far fewer parameters
- On some tasks it even outperforms full fine-tuning
- For some datasets a rank of r=1 is sufficient
- LoRA performs well when the dataset is large or small



LoRA Results

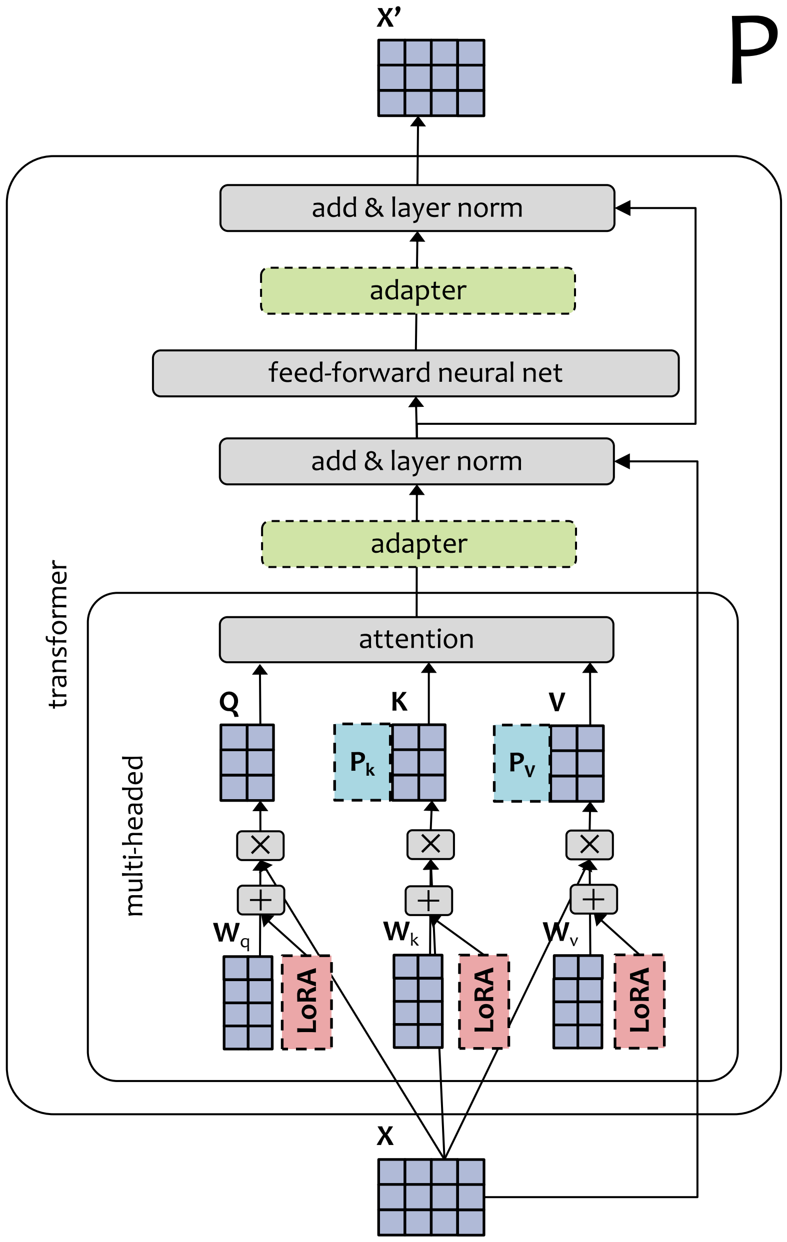
Takeaways

- Applied to GPT-3, LoRA achieves performance almost as good as full fine-tuning, but with far fewer parameters
- On some tasks it even outperforms full fine-tuning
- For some datasets a rank of r=1 is sufficient
- LoRA performs well when the dataset is large or small

	Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL($\pm 0.5\%$)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI ($\pm 0.1\%$)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4

Method	MNLI(m)-100	MNLI(m)-1k	MNLI(m)-10k	MNLI(m)-392K
GPT-3 (Fine-Tune)	60.2	**85.8**	88.9	89.5
GPT-3 (PrefixEmbed)	37.6	75.2	79.5	88.6
GPT-3 (PrefixLayer)	48.3	82.5	85.9	89.6
GPT-3 (LoRA)	**63.8**	85.6	**89.2**	**91.7**

PEFT for Transformer



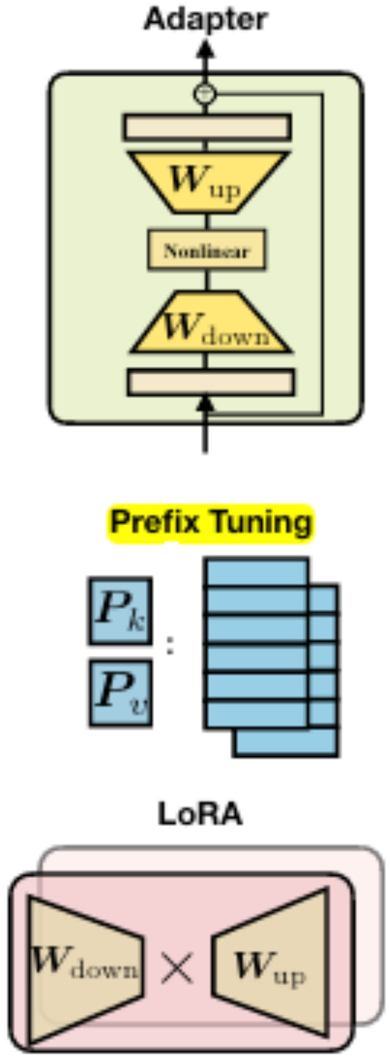


Figure inspired by / copied from He et al. (2022)

PEFT FOR VISION TRANSFORMER

PEFT for Vision Transformer

- Since Vision Transfomer is just another transformer model, we can apply LoRA directly to it
- (LNLoRA is just a variant that includes LayerNorm in the LoRALinear module.)

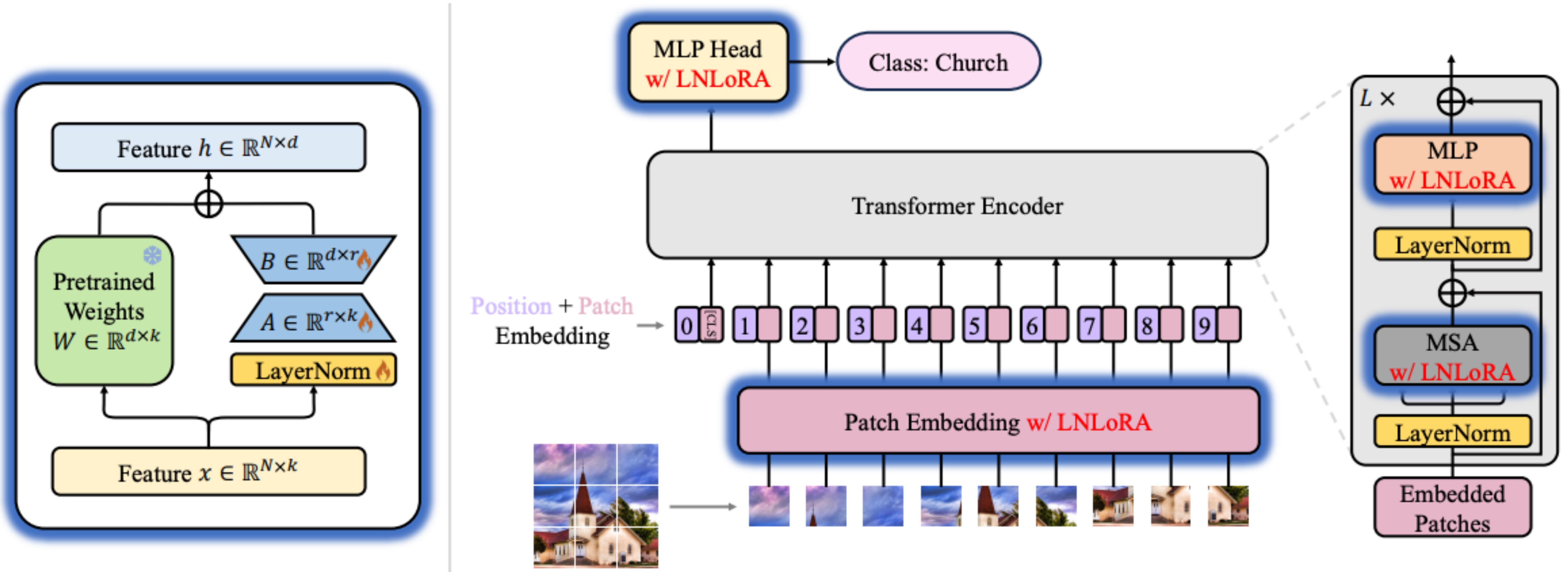
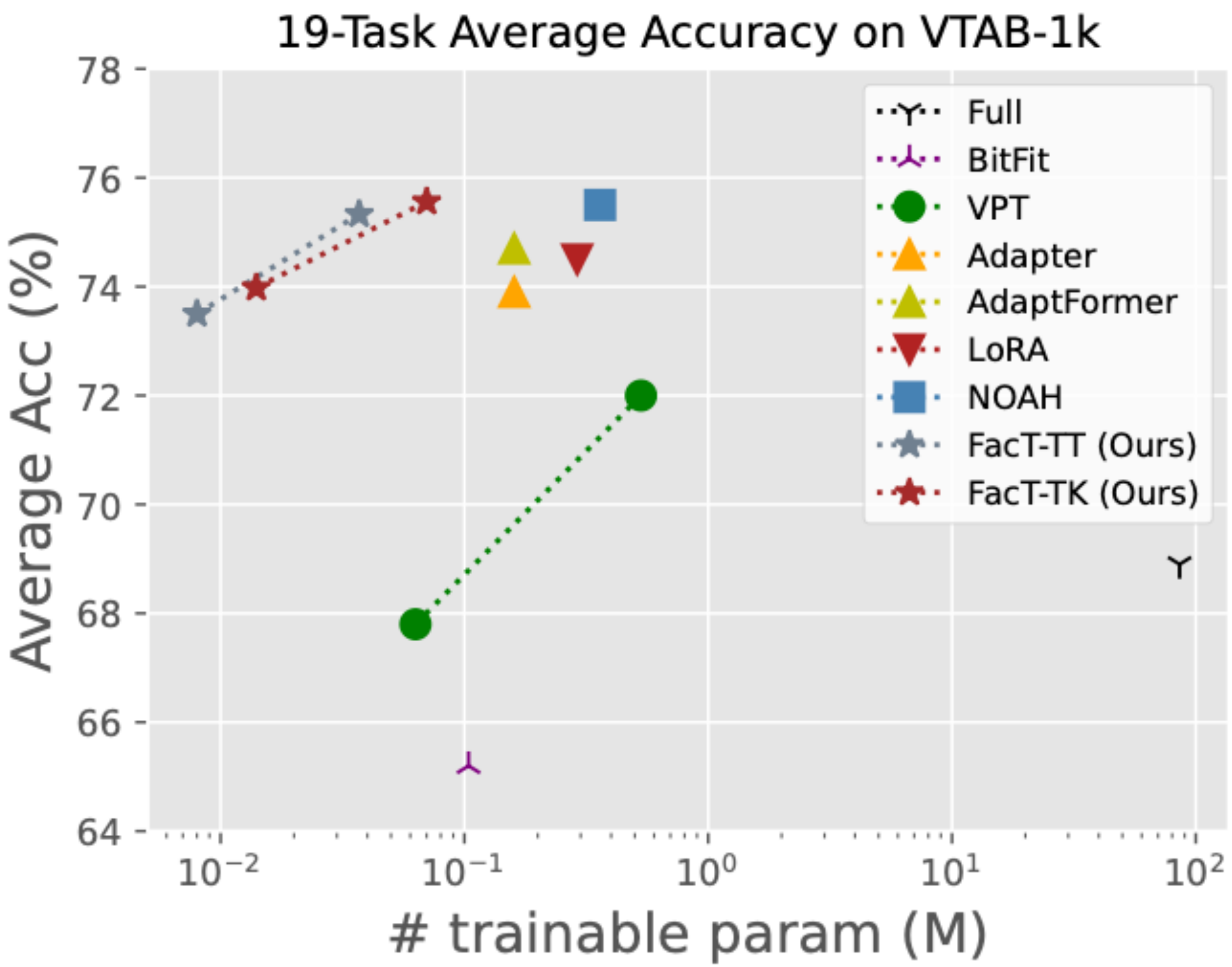


Figure from https://arxiv.org/pdf/2401.01752.pdf

PEFT for Vision Transformer



- For various computer vision tasks, parameter efficient transfer-learning (PETL) is sometimes **better** than full fine-tuning!
- VTAB-1k is a collection of 19 different vision tasks; here we're seeing average performance across tasks
- (FacT is another low-rank method capable of dramatically reducing the number of parameters tuned.)

Figure from https://ojs.aaai.org/index.php/AAAI/article/view/25187