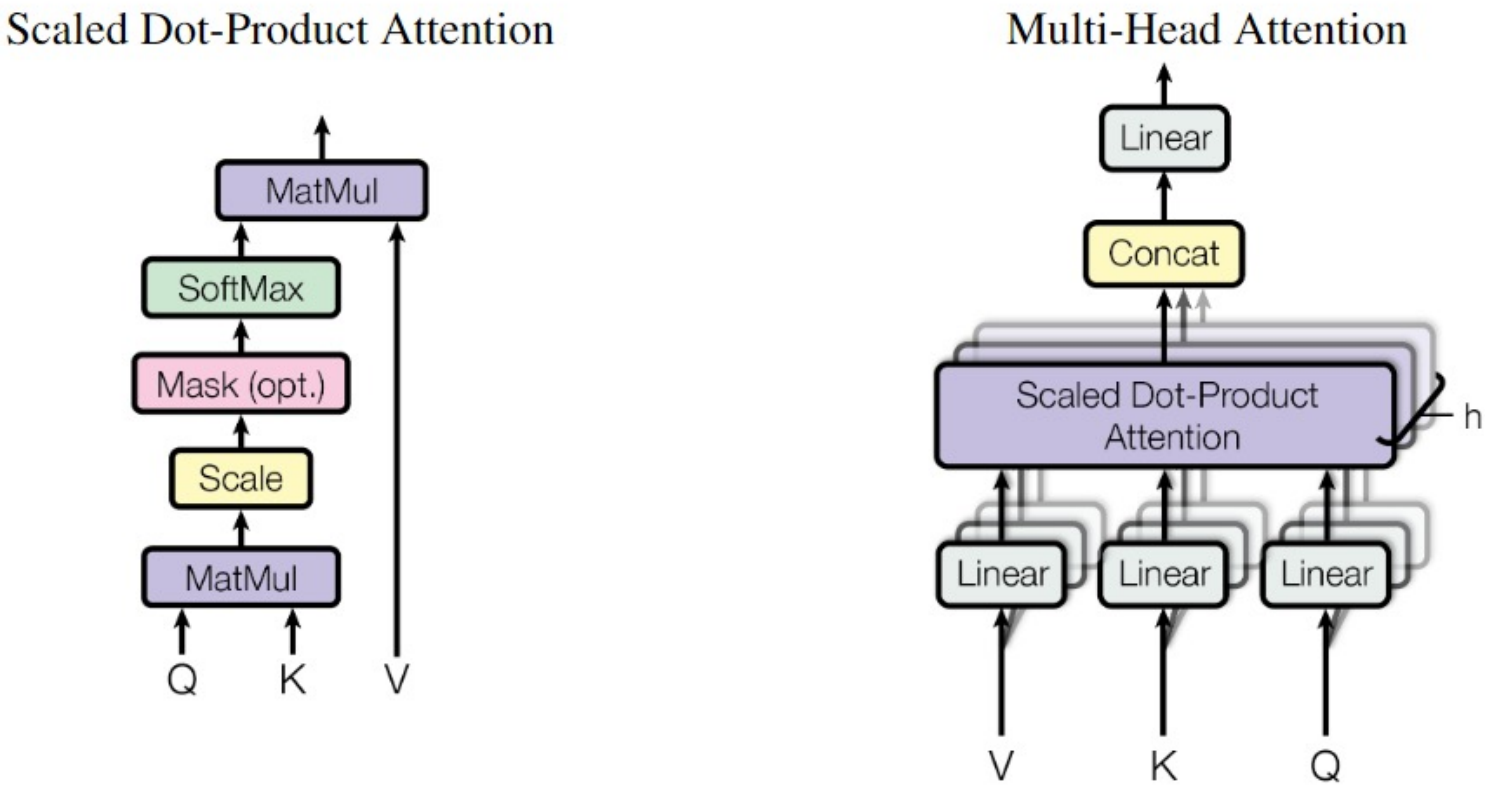
EECS 230 Deep Learning
Lecture 13: Training LLM

Recap: Attention Operation

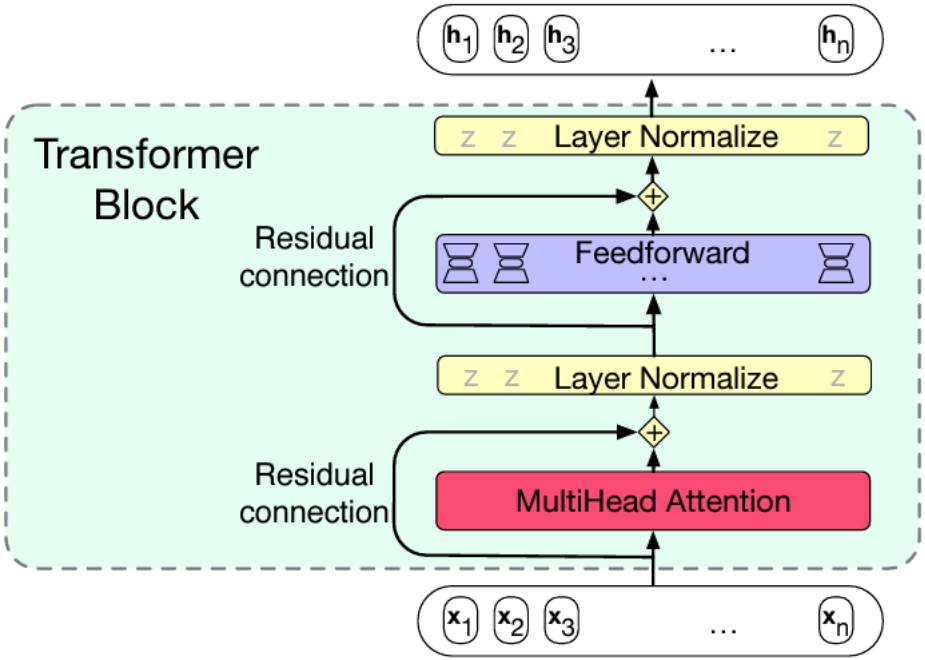
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Recap: Multi-head attention

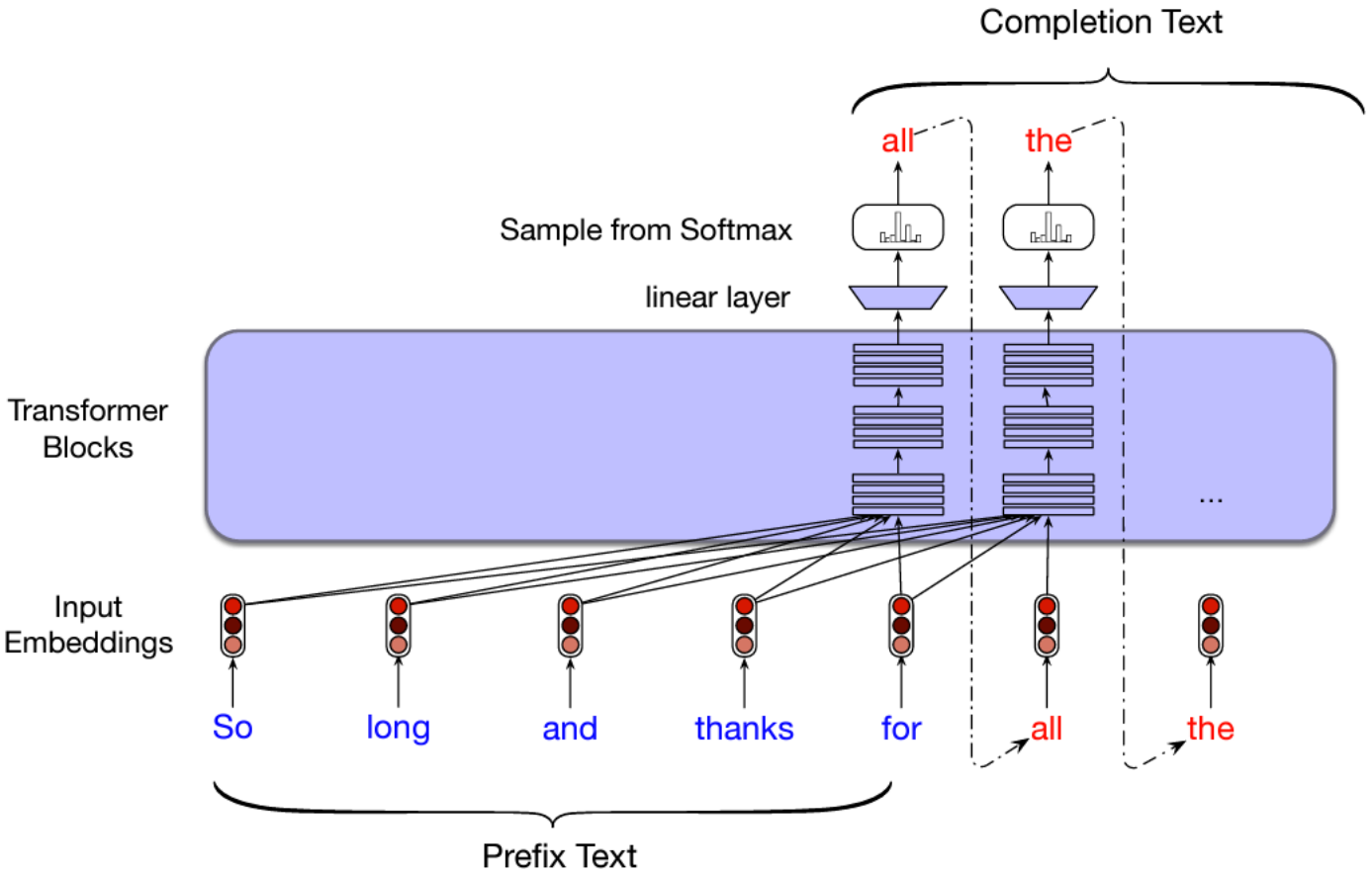


Recap: From Attention to Transformer Block

- A transformer block has
  - **Self Attention**: information exchange between tokens
  - **Feed forward network**: Information transform within tokens
    - E.g. a multi-layer perceptron with 1 hidden layer
  - **Normalization** (Layer normalization)
  - **Residual connection**



Recap:Transformer-based Large Language Model

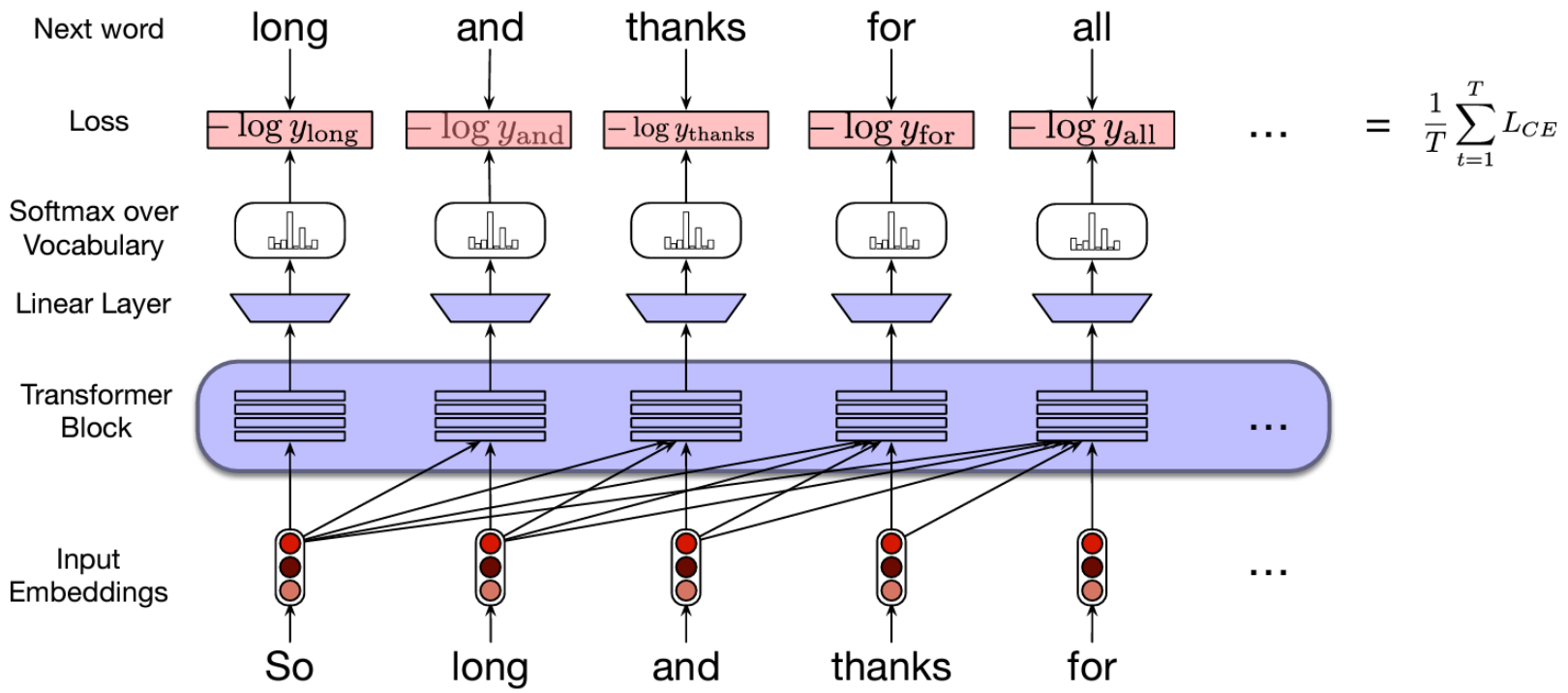


Outline

- ❑Training LLM
- ❑LLM beyond language modeling
- ❑Fine-tuning LLM

Self-supervised training algorithm

- Take a corpus of text as training material
- Minimize cross-entropy loss for next-word prediction

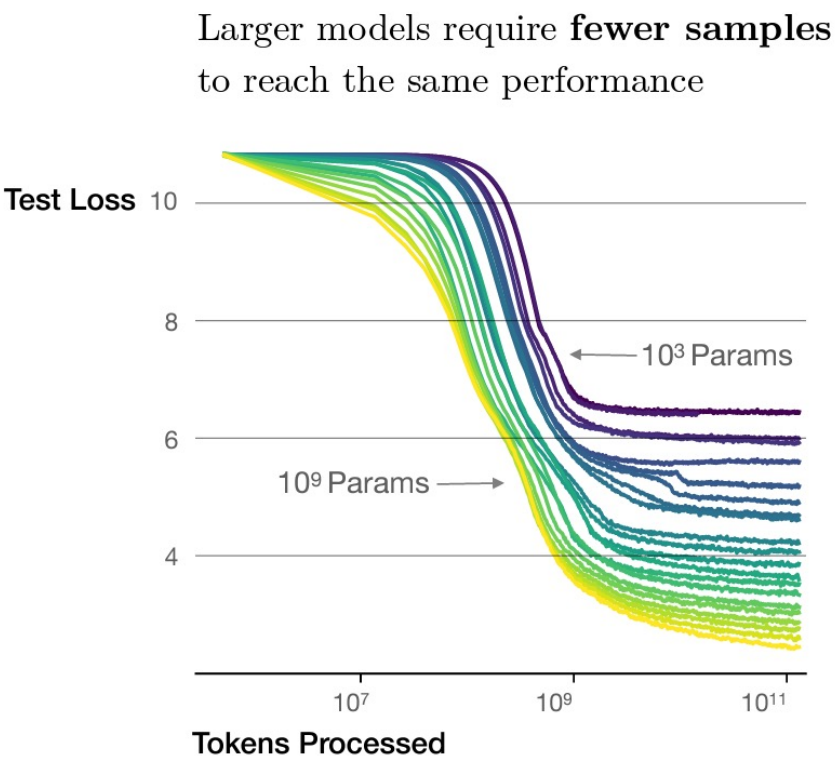


Training corpora for large language models

- commoncrawl.org
  - Over 250 billion pages spanning 17 years.
  - Free and open corpus since 2007.
  - Cited in over 10,000 research papers.
  - 3–5 billion new pages added each month.
- Training data for GPT-3
  - web (429 billion tokens)
  - some text from books (67 billion tokens)
  - Wikipedia (3 billion tokens).

Scaling laws

- Cross-entropy loss scales as a power-law with
  - model size
  - dataset size
  - amount of compute



$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N}$$

$$L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D}$$

$$L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C}$$

UCMERCED

Conditional generation using LLM

- Many practical NLP tasks can be cast as word prediction
- For example, sentiment classification
  - The sentiment of the sentence “I like Jackie Chan” is:
- Question answering
  - Q: Who wrote the book ‘‘The Origin of Species"? A:
- Summarization
  - Append token tl;dr

How large are 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	?	?

Fine-tuning LLM

Few-shot Learning with LLMs

- Suppose you have…
  - a dataset D = {(xi, yi)},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)
- Pros:
  - fits into the standard ML recipe
  - still works if N is large
- Cons:
  - backpropagation requires ~3x the memory and computation time as the forward computation
  - you might not have access to the model weights at all

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
- Cons:
  - the prompt may be very long and Transformer LMs require O(N2) time/space where N = length of context
- Pros:
  - no backpropagation required and only one pass through the training data
  - does not require model weights, only API access

Fine-Tuning vs. In-Context Learning

- Why would we ever bother with fine-tuning if it's so inefficient?
- 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

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

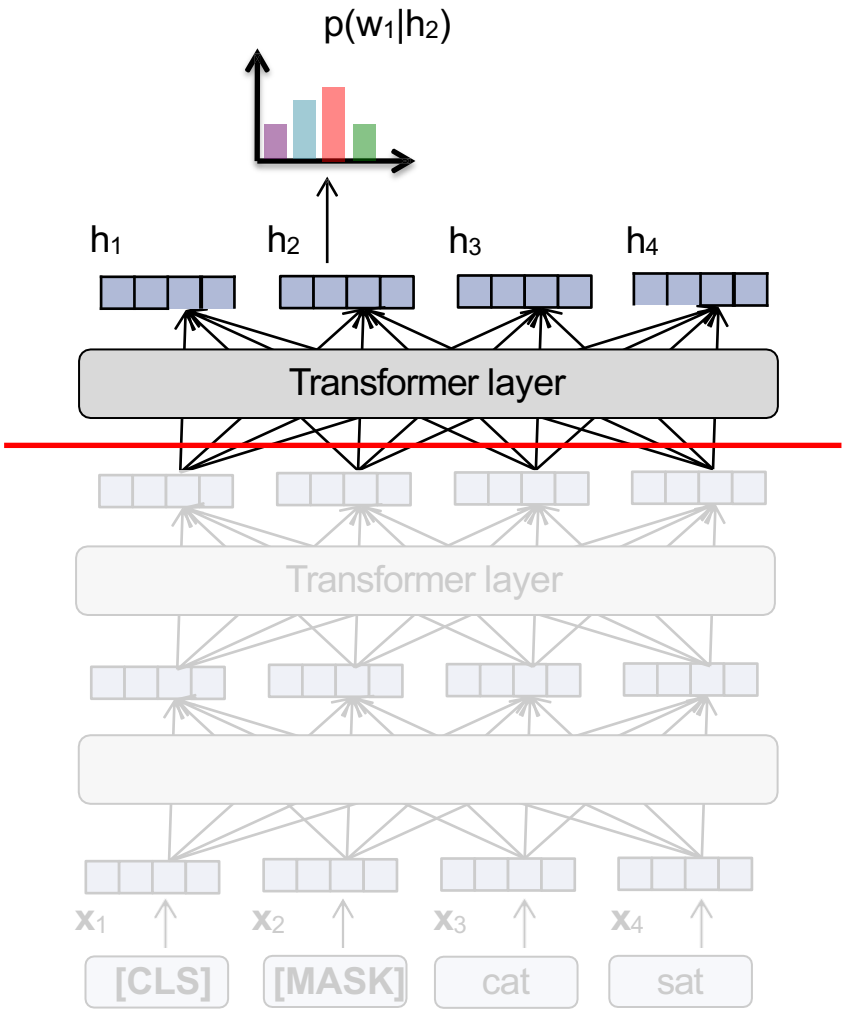
Parameter Efficient Fine-Tuning

- Goal: perform fine-tuning of fewer parameters, but achieve performance on a downstream task that is comparable to finetuning of all parameters
- Various approaches:
  - Subset: Pick a subset of the parameters and fine-tune only those
  - Adapters: add additional layers that have few parameters and tune only the parameters of those layers, keeping all others fixed
  - 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

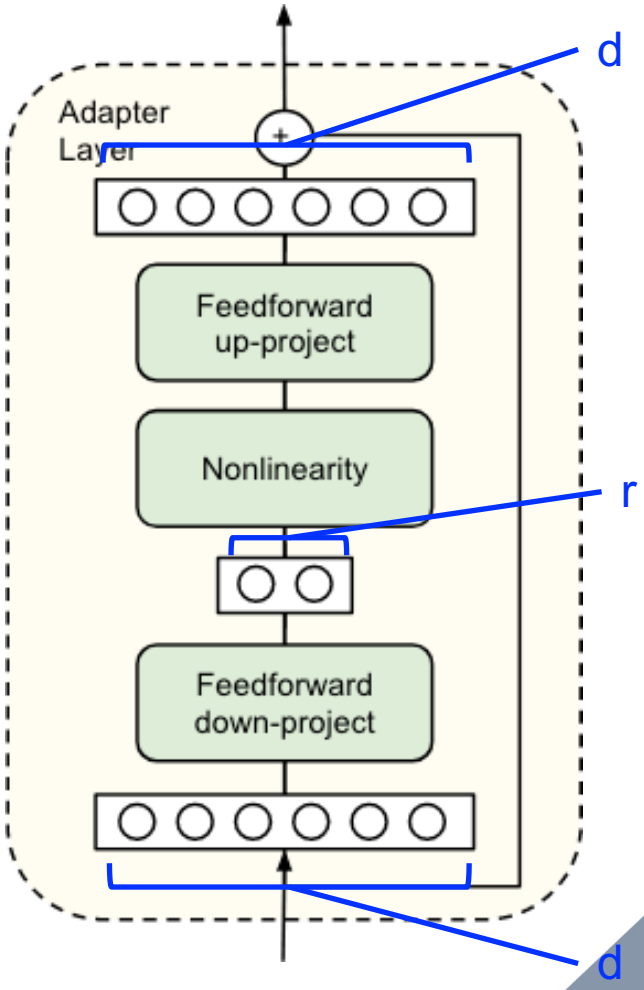
- keep all parameters fixed except for the top K layers

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



Adapter 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 r in the middle



Adapter for transformer

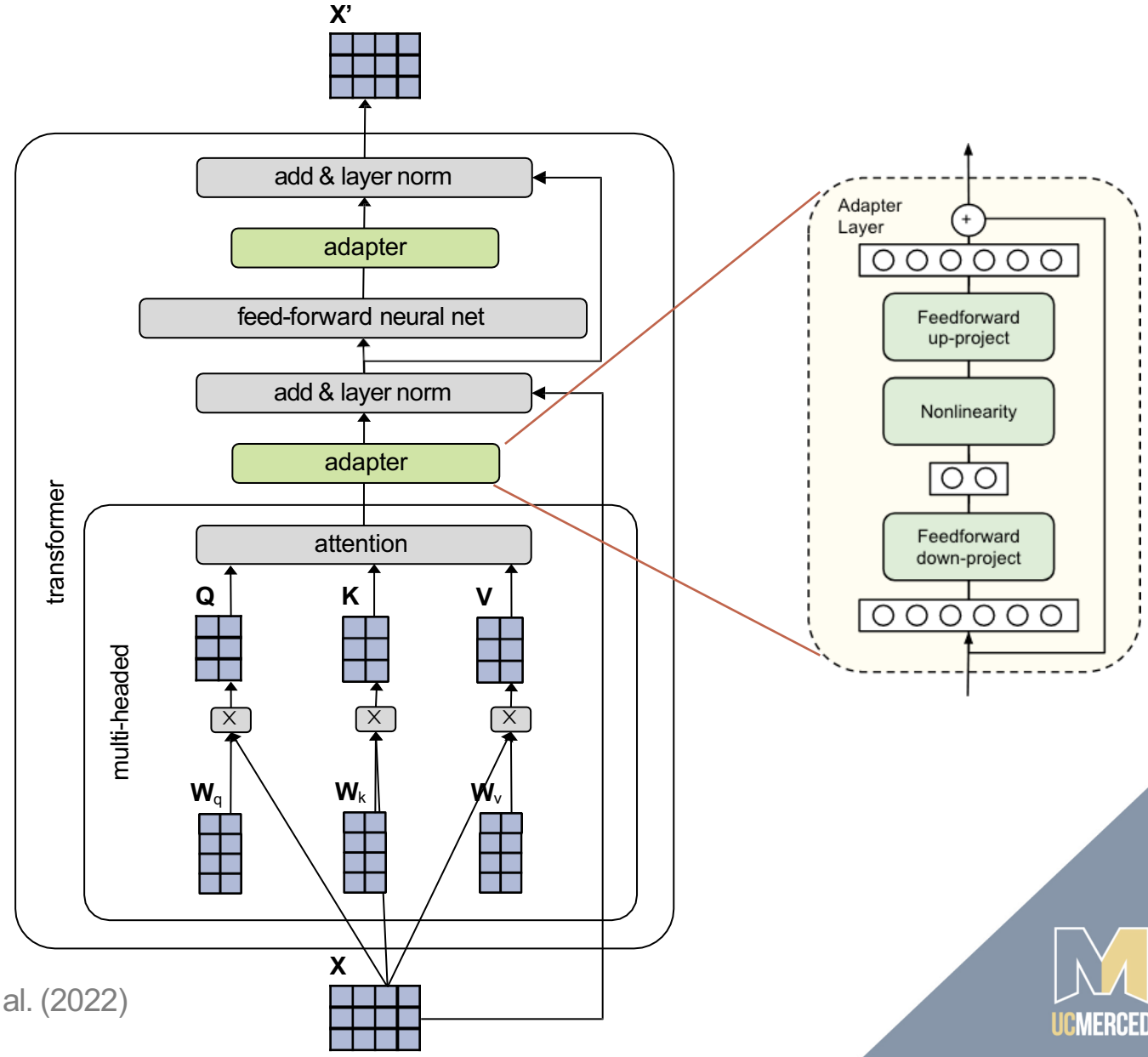
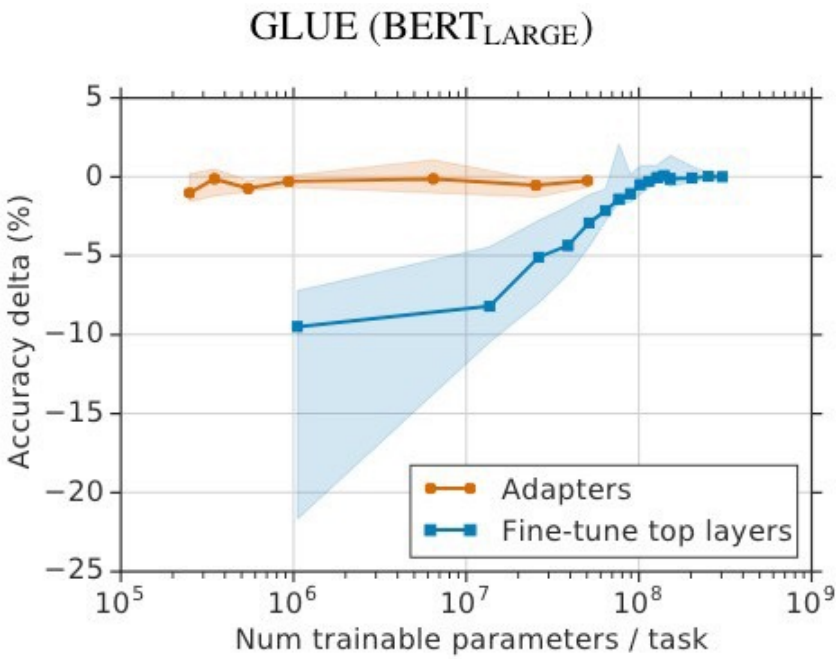


Figure inspired by He et al. (2022)

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



UCMERCED

Low-Rank Adaptation (LORA)

- 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
$\text{Adapter}^{\text{L}}$	1482.0±1.0 (+2.2%)	354.8±0.5 (+5.0%)	23.9±2.1 (+20.7%)
$\text{Adapter}^{\text{H}}$	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. $\text{Adapter}^{\text{L}}$ and $\text{Adapter}^{\text{H}}$ 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:

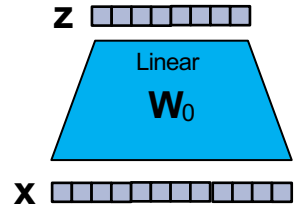
$$\Delta W = BA$$

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

Standard Linear Layer

$$z = W_0 x$$

$$W_0 \in R^{d \times k}, x \in R^k, z \in R^d$$



LoRA Linear Layer

$$z = W_0 x + BAx$$
$$= (W_0 + BA)x$$

$$W_0 \in R^{d \times k},$$

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

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

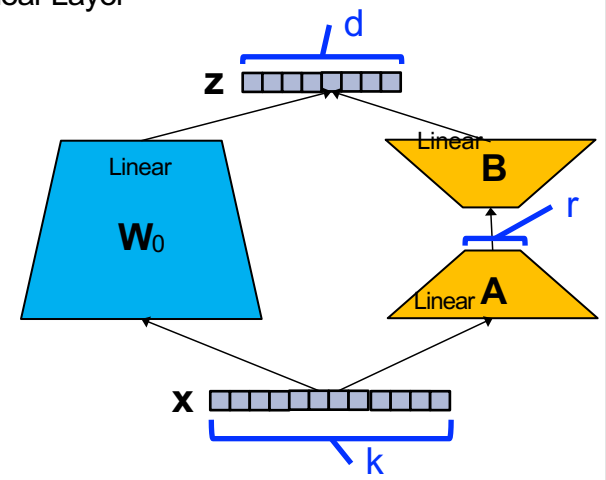


Figure inspired by He et al. (2022)

Slide from Matthew Gormley

LoRA

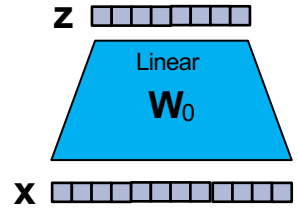
Initialization

- We initialize the trainable parameters:
  $A_{ij} \sim \mathrm{N}(0, \sigma^2), \forall i, j$
  $\mathrm{B} = 0$
- This ensures that, at the start of fine tuning, the parameters have their pretrained values:
  $\Delta \mathrm{W} = \mathrm{BA} = 0$
  $\mathrm{W}_0 + \mathrm{BA} = \mathrm{W}_0$

Standard Linear Layer

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

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



LoRA Linear Layer

$\mathrm{z} = \mathrm{W}_0\mathrm{x} + \mathrm{BAx}$

$= (\mathrm{W}_0 + \mathrm{BA})\mathrm{x}$

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

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

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

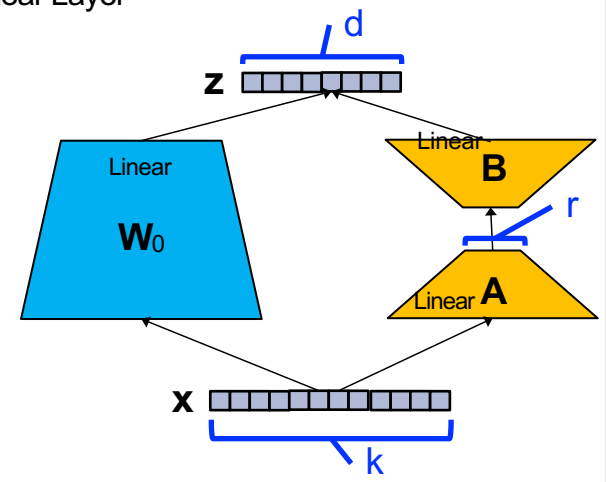
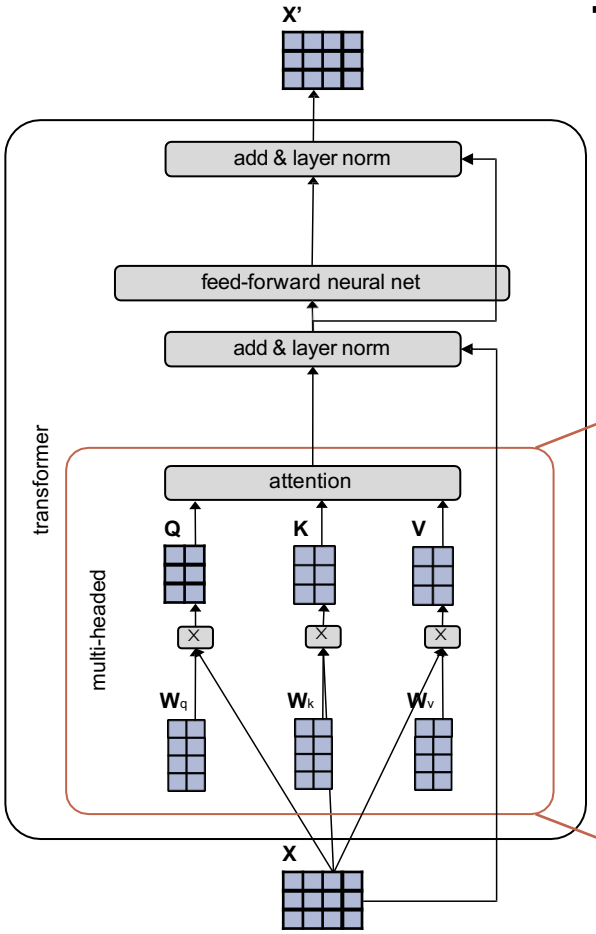


Figure inspired by He et al. (2022)

Slide from Matthew Gormley

Transformer Layer



multi-headed attention

$$X^{ll} = \text{concat}(X^{l(1)}, \ldots, X^{l(h)})$$

$$X^{l(i)} = \text{softmax}\left(\frac{Q^{(i)}(K^{(i)})^T}{\sqrt{d_k}} + M^l\right)V^{(i)}$$

$$Q^{(i)} = XW_q^{(i)}$$

$$K^{(i)} = XW_k^{(i)}$$

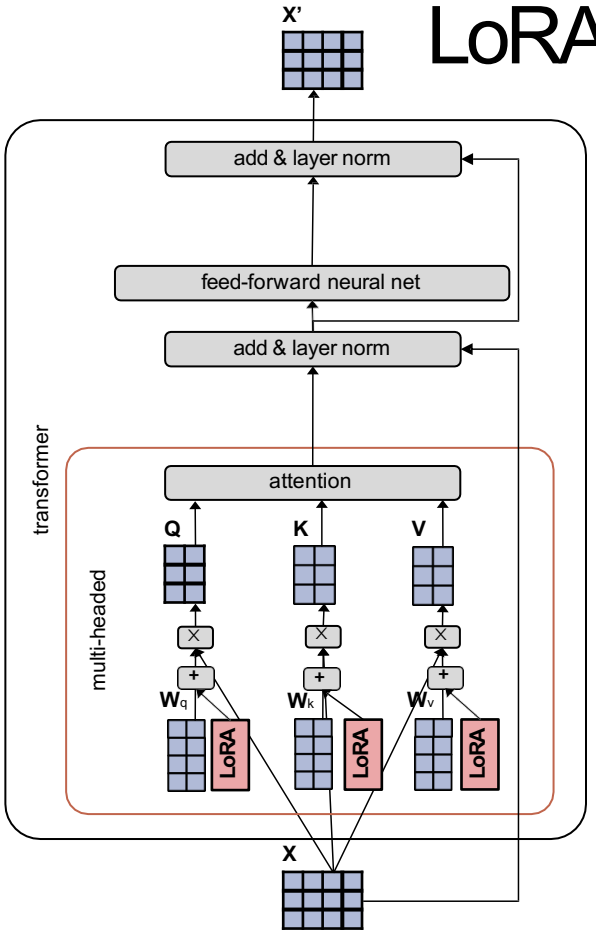
$$V^{(i)} = XW_v^{(i)}$$

$$X = [x_1, \ldots, x_N]^T$$

Figure inspired by He et al. (2022)

Slide from Matthew Gormley

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

$$z = W_0 x + BAx$$
$$= (W_0 + BA)x$$

$$W_0 \in R^{d \times k},$$
$$A \in R^{r \times k}, B \in R^{d \times r}$$
where $r << \min(d, k)$

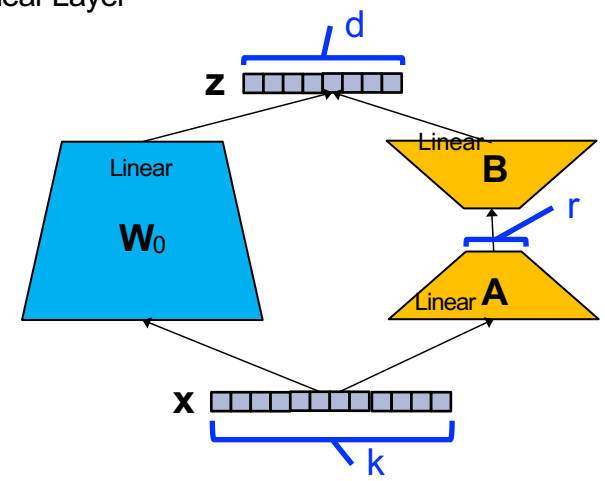
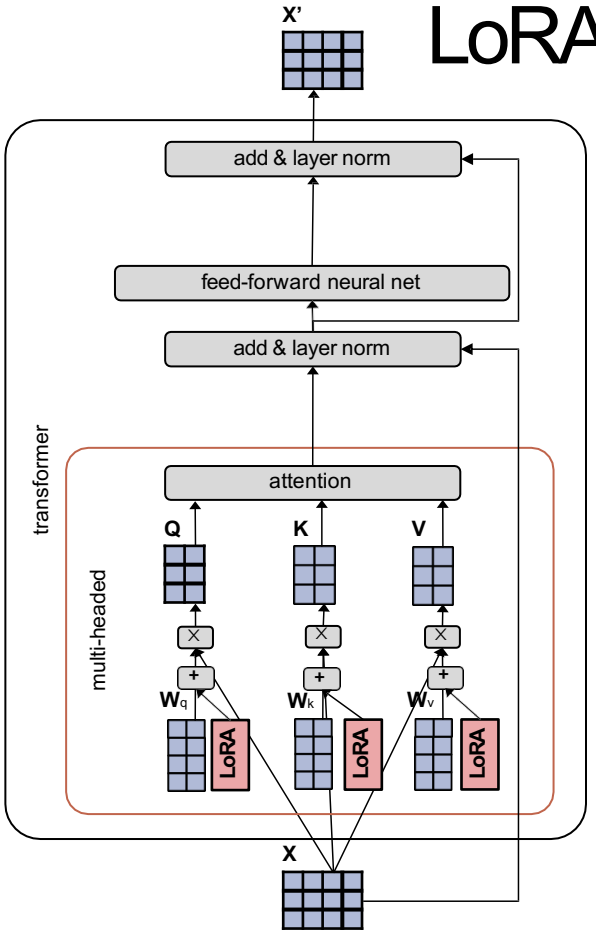


Figure inspired by He et al. (2022)

Slide from Matthew Gormley

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

$$Q = \text{LoRALinear}(X; W_q, A_q, B_q)$$
$$K = \text{LoRALinear}(X; W_k, A_k, B_k)$$
$$V = \text{LoRALinear}(X; W_v, A_v, B_v)$$

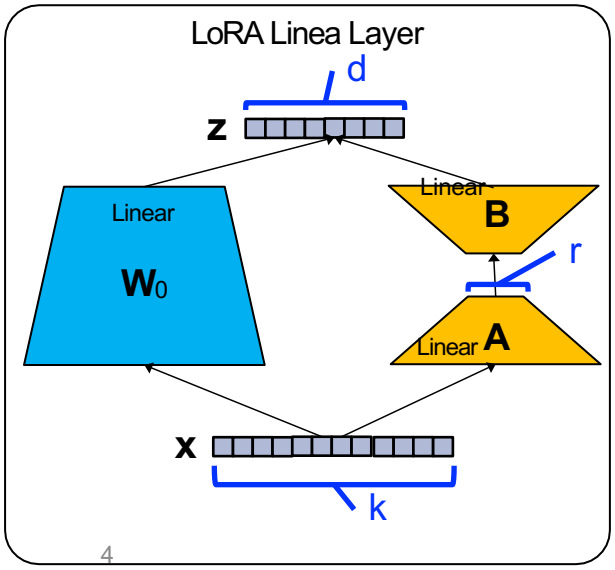
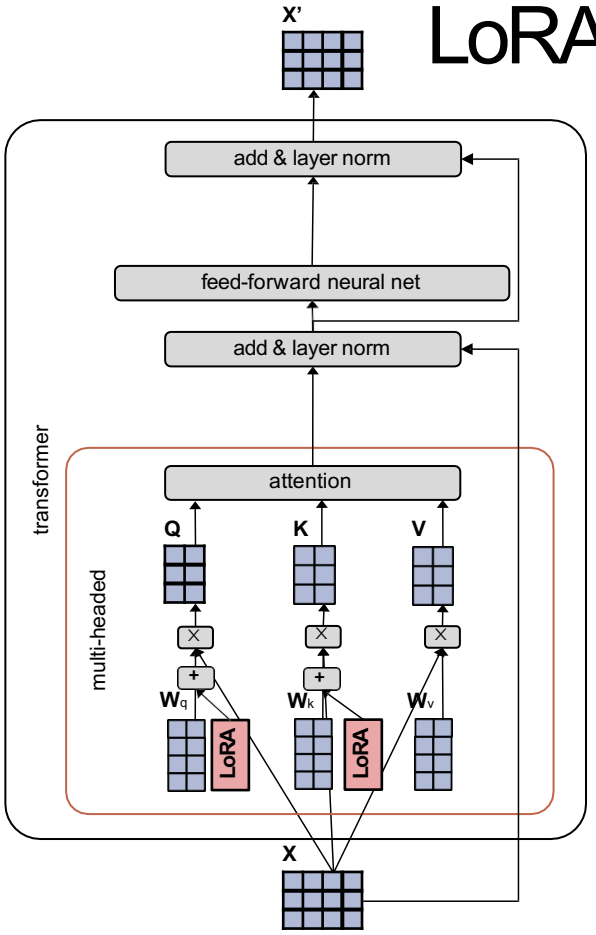


Figure inspired by He et al. (2022)

Slide from Matthew Gormley

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 key** linear layers:

	# of Trainable Parameters = 18M						
Weight Type	W_q	W_k	W_v	W_o	W_q, W_k	W_q, W_v	W_q, W_k, W_v, W_o
Rank r	8	8	8	8	4	4	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

Slide from Matthew Gormley

Summary

- ❑Training LLM
- ❑Parameter-efficient Fine-tuning
  - ❑Subset
  - ❑Adaptor
  - ❑LoRA