CS 6501 Natural Language Processing

Efficient Fine-tuning for LLMs

Yangfeng Ji

Information and Language Processing Lab
Department of Computer Science
University of Virginia
https://uvanlp.org/

Section I

Prefix Tuning

(Li and Liang, 2021)

Full Fine-tuning

For a text generation task, given the input x and output y, the full fine-tuning objective is defined as

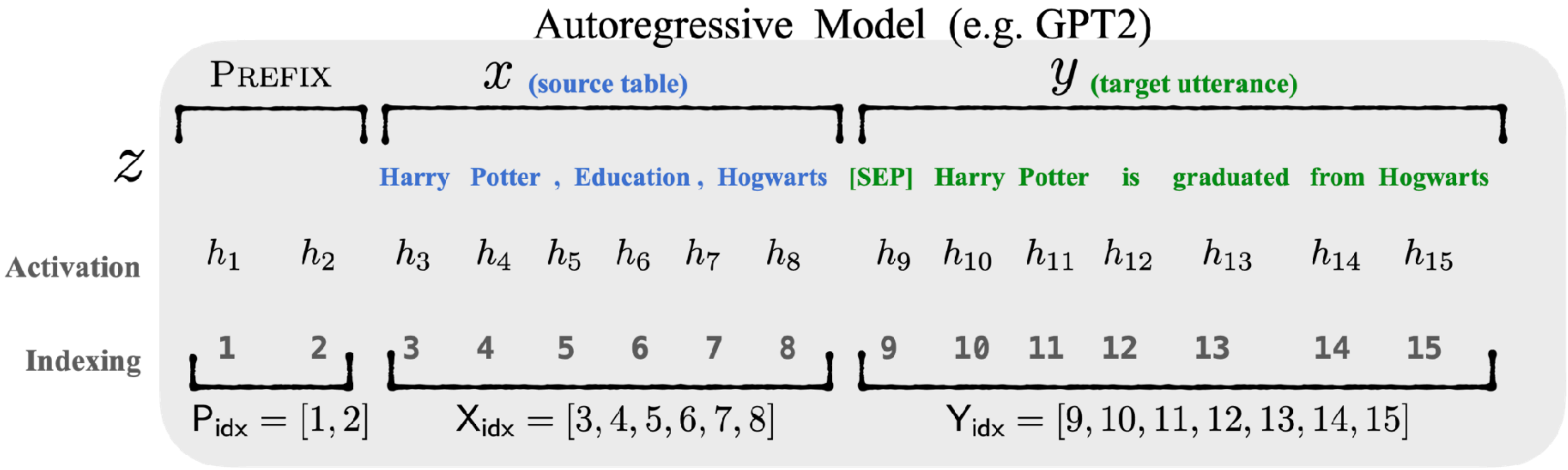
$$\max_{\phi} \sum_{i \in Y_{\mathrm{idx}}} \log p_{\phi}(y_i \mid h_{<i})$$

where

- Y_{idx}: indices of output tokens
- z_i is the i-th token of $z = [x, y]$
- $p_{\phi}(z_i \mid h_{<i}) = \mathrm{softmax}(W_{\phi} h_{i-1}^{\mathrm{top}})$

Prefix Tuning

Add k virtual tokens to the **latent representations** on **each layer**



- Number of virtual tokens k is a hyperparameter
    - $k = 2$ in this example

Latent Representations

Depending whether i is the virtual token or not, on the n-th layer, we have

$$h_i^n = \begin{cases} P_\theta[i, n, :] & \text{if } i \in P_{\text{idx}} \\ \mathrm{LM}_\phi(z_i, h_{<i}^n) & \text{otherwise} \end{cases}$$

- P_{idx}: prefix index set (e.g., $\{1, 2\}$)
- ϕ are fixed and θ are the only trainable parameters

Prefix Projection

The prefix embeddings can also be computed via

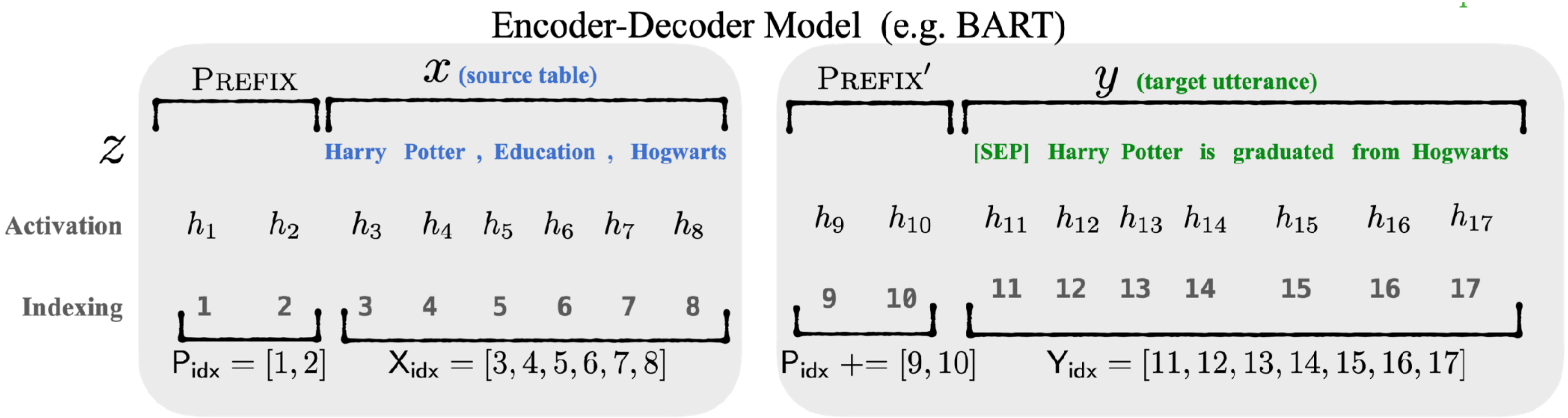
$$P[i, n, :] = \mathrm{MLP}_\theta(P'_\theta[i, :])$$

where

- $\mathrm{MLP}_\theta(\cdot)$ is a two-layer feedforward NN with the Tanh activation
- Empirically, this project produced more stable results than directly training the embedding $P_\theta[i, n, :]$

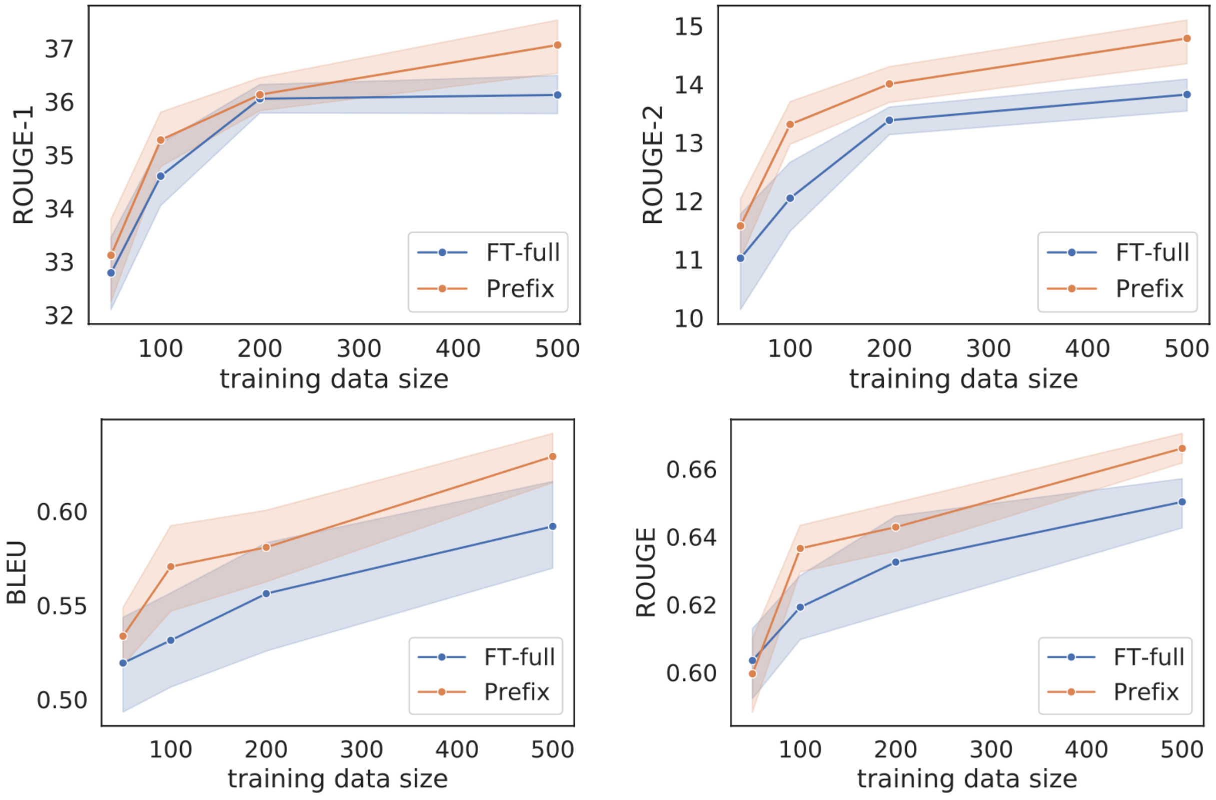
With Encoder-Decoder Framework

The idea of prefix tuning in the encoder-decoder framework is similar to the autoregressive framework, except the position of prefix embeddings



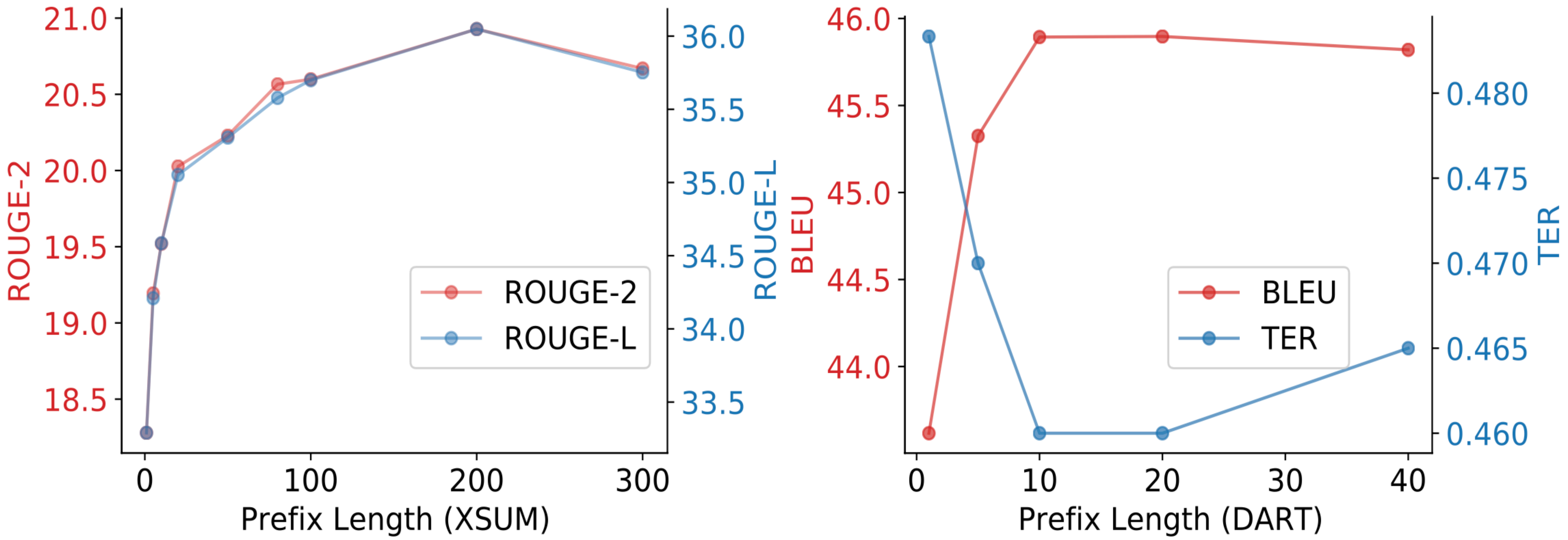
Experiment: Low-data Settings

In low-data setting, prefix-tuning is better than full fine-tuning (on both summarization and data-to-text generation)



Experiment: Prefix Length

Increase the size of prefix embeddings will increase the performance, until a certain threshold (task-dependent)



Experiment: Two Alternative Designs

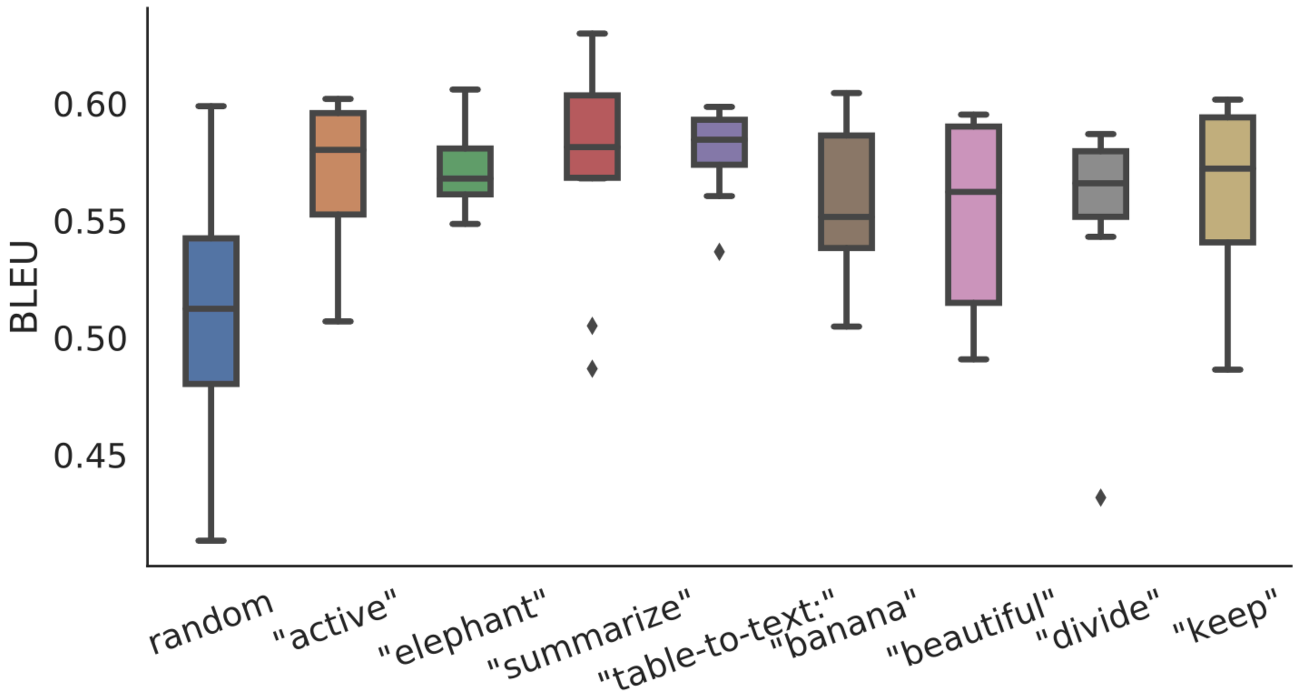
- Embedding only
  - Only use the prefix embeddings in the input layer
  - Higher-layer representations are computed by the Transformer
- Infix tuning
  - Add virtual tokens between the input and output, as

$$[x, \mathrm{Infix}, y]$$

Experiment: Results

	E2E				
	BLEU	NIST	MET	ROUGE	CIDEr
PREFIX	70.3	8.82	46.3	72.1	2.46
	Embedding-only: EMB-{PrefixLength}				
EMB-1	48.1	3.33	32.1	60.2	1.10
EMB-10	62.2	6.70	38.6	66.4	1.75
EMB-20	61.9	7.11	39.3	65.6	1.85
	Infix-tuning: INFIX-{PrefixLength}				
INFIX-1	67.9	8.63	45.8	69.4	2.42
INFIX-10	67.2	8.48	45.8	69.9	2.40
INFIX-20	66.7	8.47	45.8	70.0	2.42

Initialization



- Initialization with task-relevant words works better than task-irrelevant words
- Initialization with word embeddings works better than random initialization

Prefix Tuning vs. Discrete Prompt Optimization

Why prefix tuning is better

- From the initialization
- From the optimization perspective

Relation

$$\text{discrete prompt optimization} < \text{embedding-only} < \text{prefix-tuning}$$

Section II

Low-Rank Adaptation (LoRA)

(Hu et al., 2021)

Fine-tuning

Given a pair of example (x, y)

$$\max_{\theta} \log p_{\phi_0+\Delta\phi(\theta)}(y_t \mid x, y_{<t})$$

- ϕ_0: pre-trained model parameter
- $\Delta\phi(\theta)$: adapater produced by task-specific fine-tuning
  - $\Delta\phi(\theta)$ has the same size as ϕ_0
  - $\Delta\phi(\theta)$ is the function of θ
  - θ has a much smaller size than $\Delta\phi(\theta)$

What in ϕ_0?

One sub-layer in the Transformer module is the multi-head attention. With H heads

- For $i = 1, \ldots, H$
  - Compute

$$\mathrm{head}_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- Concatenate multiple heads as

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_H)W_O$$

Parameters

$$\{W_i^Q, W_i^K, W_i^V\}_{i=1}^H, W^O$$

What in ϕ_0? (II)

Another sub-layer in the Transformer encoder module

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Parameters

$$W_1, b_1, W_2, b_2$$

Llama-2

Print the model architecture using `model.parameters()`

```
(0-31): 32 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
                (q_proj): Linear8bitLt(in_features=4096, out_features=4096, bias=False)
                (k_proj): Linear8bitLt(in_features=4096, out_features=4096, bias=False)
                (v_proj): Linear8bitLt(in_features=4096, out_features=4096, bias=False)
                (o_proj): Linear8bitLt(in_features=4096, out_features=4096, bias=False)
                (rotary_emb): LlamaRotaryEmbedding()
        )
        (mlp): LlamaMLP(
                (gate_proj): Linear8bitLt(in_features=4096, out_features=11008, bias=False)
                (up_proj): Linear8bitLt(in_features=4096, out_features=11008, bias=False)
                (down_proj): Linear8bitLt(in_features=11008, out_features=4096, bias=False)
          (act_fn): SiLUActivation()
        )
        (input_layernorm): LlamaRMSNorm()
        (post_attention_layernorm): LlamaRMSNorm()
)
```

Low-Rank Adapter

For any parameter matrix $W_0 \in \mathbb{R}^{d \times k}$, the low-rank adapter $\Delta W \in \mathbb{R}^{d \times k}$

$$\Delta W = B \cdot A$$

where

- $B \in \mathbb{R}^{d \times r}$
- $A \in \mathbb{R}^{r \times k}$
- $r \ll \min(d, k)$ is the rank

Initialization

- Initialize A with a Gaussian distribution
- Initialize B as zero

Therefore, initially

$$\Delta W = A \cdot B = 0$$

Additionally, ΔW is scaled by $\frac{\alpha}{r}$

Applying LoRA to Transformer

- The discussion is limited to attention weights, e.g.,

$$W^Q, W^K, W^V, W^O$$

- Can be also used on other metrics, for example, the MLP sublayer

Results

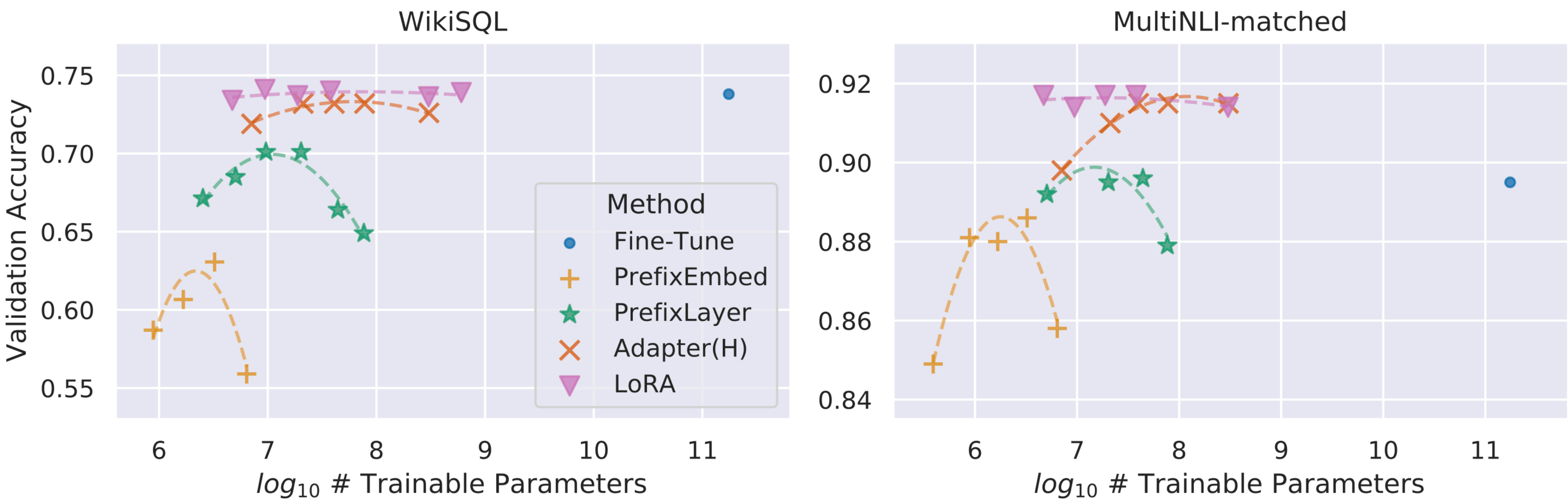


Figure 2: GPT-3 175B validation accuracy vs. number of trainable parameters of several adaptation methods on WikiSQL and MNLI-matched. LoRA exhibits better scalability and task performance.

Which Weight Matrices?

	# 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.

Optimal Rank?

	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

Table 6: Validation accuracy on WikiSQL and MultiNLI with different rank r. To our surprise, a rank as small as one suffices for adapting both W_q and W_v on these datasets while training W_q alone needs a larger r. We conduct a similar experiment on GPT-2 in Section H.2.

Last Comments

Pay attention to the variable names. For example, in `Falcon`

```
(0-31): 32 x FalconDecoderLayer(
        (self_attention): FalconAttention(
                (maybe_rotary): FalconRotaryEmbedding()
                (query_key_value): Linear8bitLt(in_features=4544, out_features=4672, bias=False)
                (dense): Linear8bitLt(in_features=4544, out_features=4544, bias=False)
                (attention_dropout): Dropout(p=0.0, inplace=False)
        )
        (mlp): FalconMLP(
                (dense_h_to_4h): Linear8bitLt(in_features=4544, out_features=18176, bias=False)
                (act): GELU(approximate='none')
                (dense_4h_to_h): Linear8bitLt(in_features=18176, out_features=4544, bias=False)
        )
        (input_layernorm): LayerNorm((4544,), eps=1e-05, elementwise_affine=True)
)
```

Thank You!