CIS 6930 Special Topics in Large Language Models

LLM Fine-Tuning

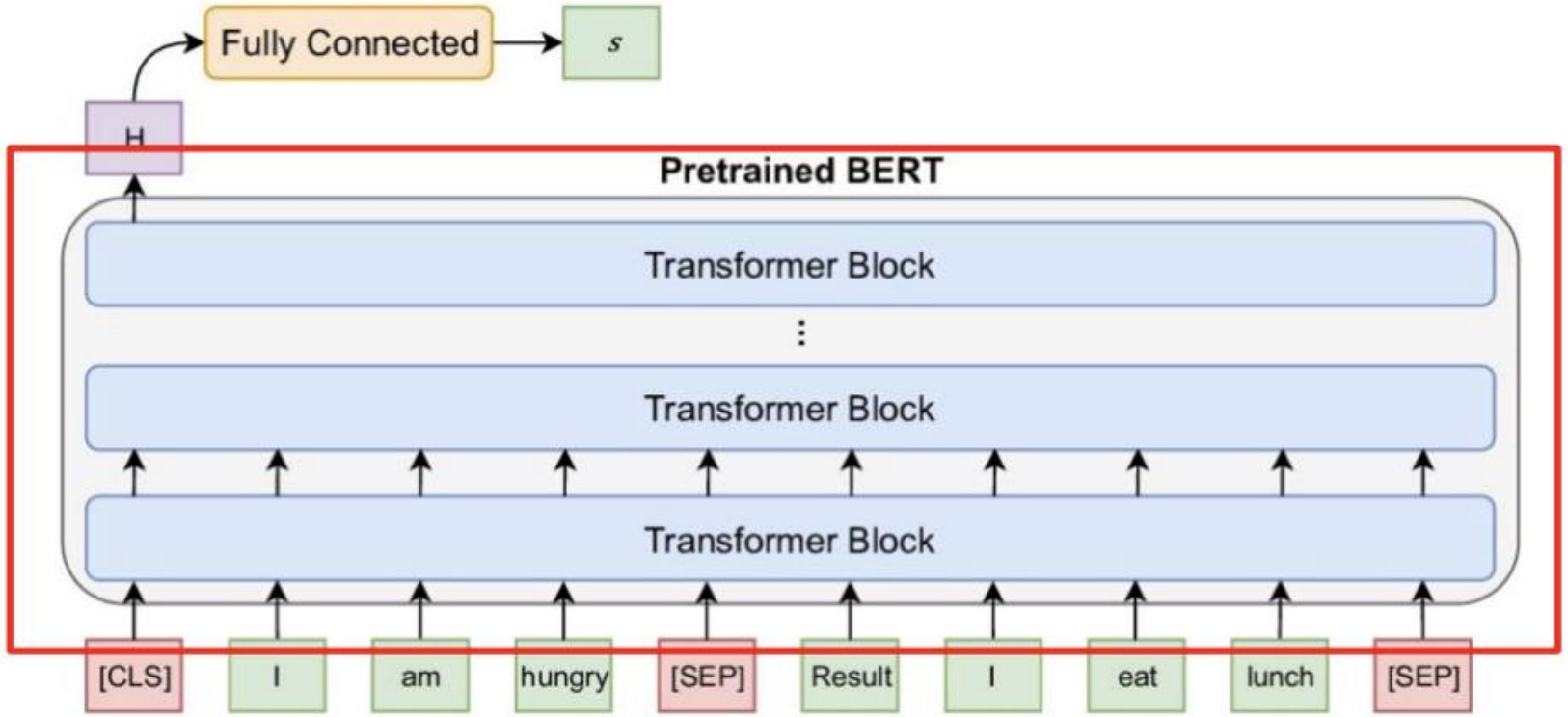
Outline

- Classical Fine-Tuning
- Prompt-based Fine-Tuning
- Parameter Efficient Fine-Tuning
  - Adapter
  - Infused Adapter IA3
  - Soft Prompt Tuning
  - Prefix Tuning
  - P-Tuning
  - LLaMA-Adapter
  - LoRA

Fine-Tuning

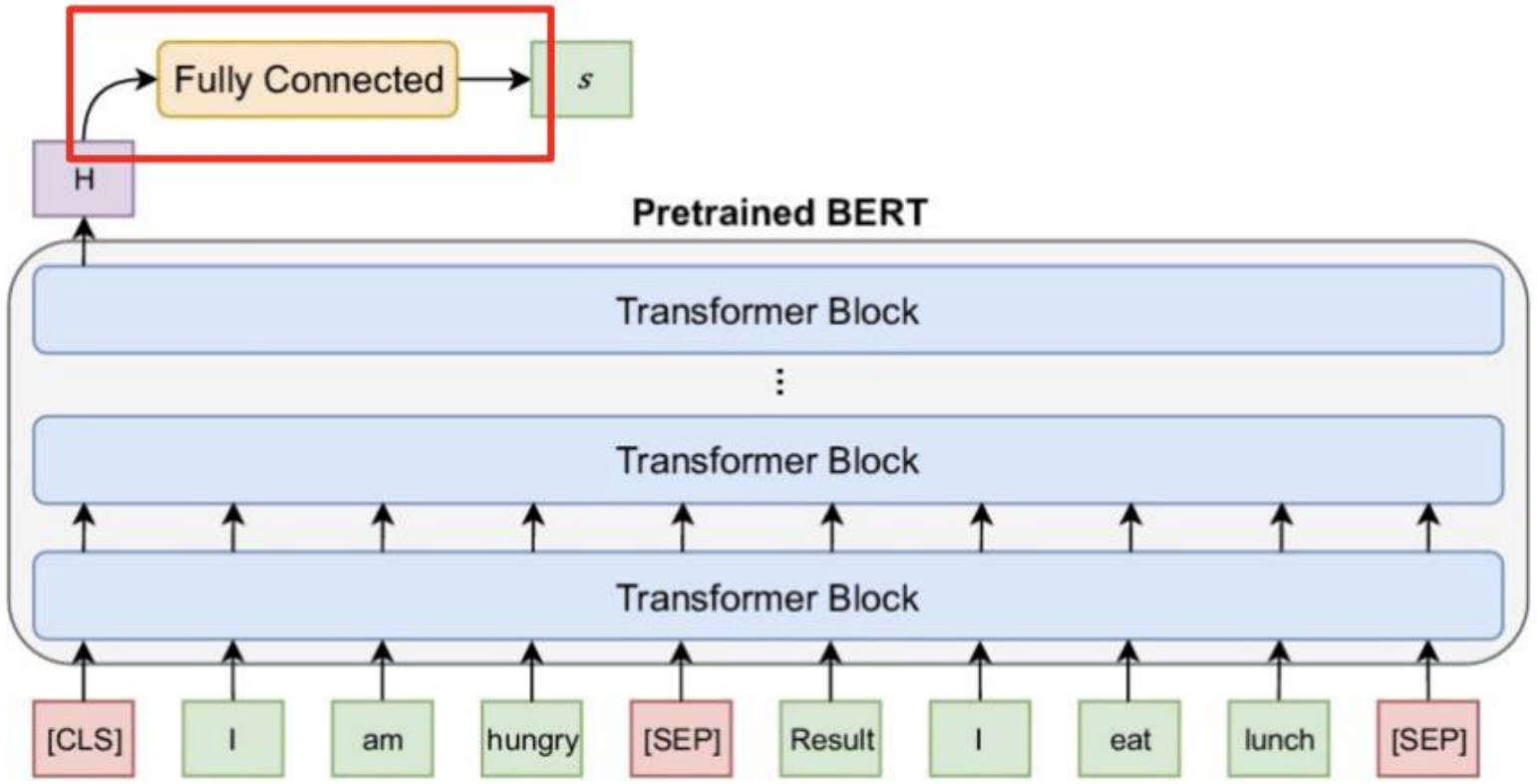
Fine-Tuning

- Step 1: Pre-train a language model on a variety of tasks



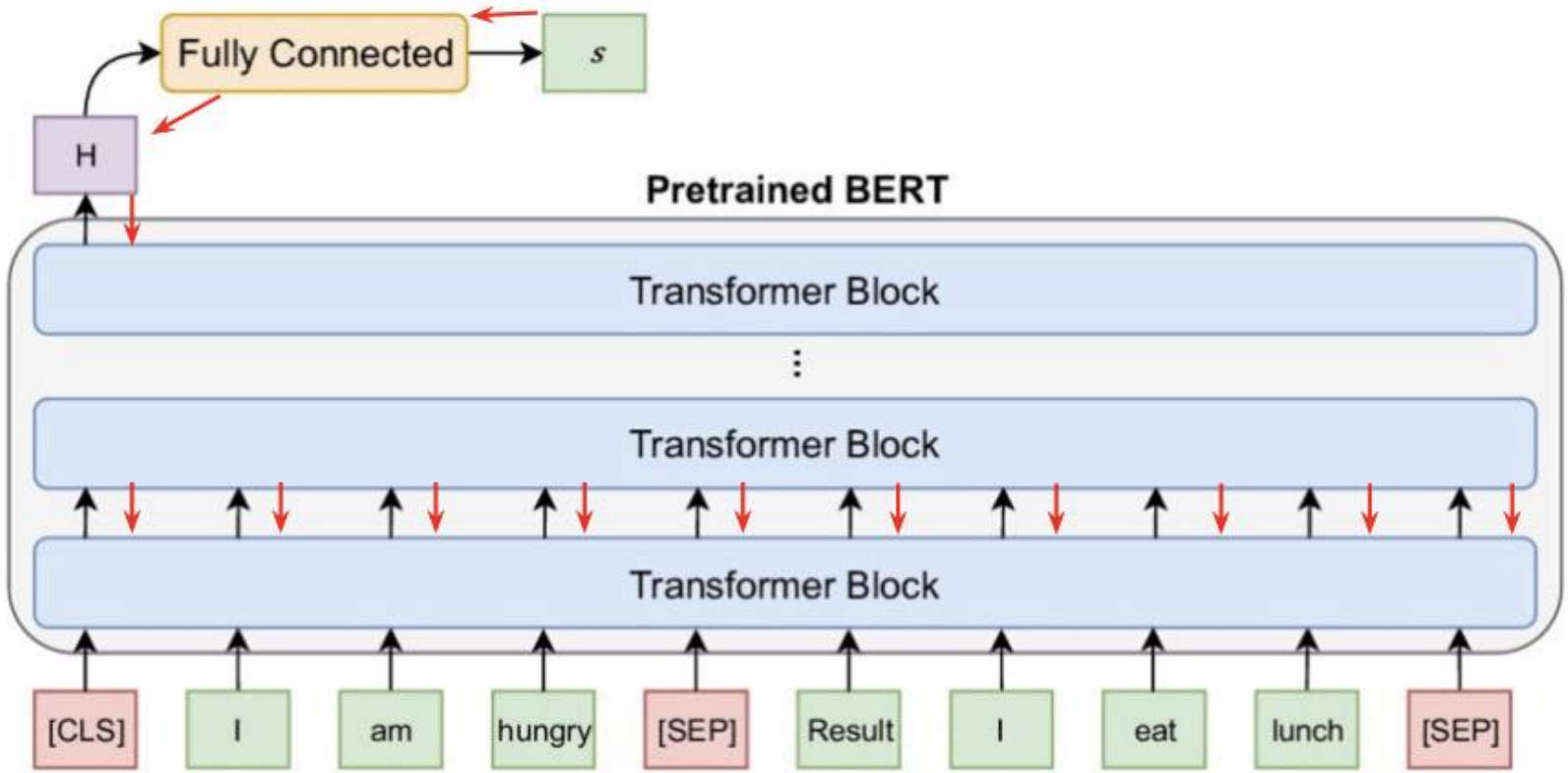
Fine-Tuning

- Step 2: Attach a small task specific layer (classification head)



Fine-Tuning

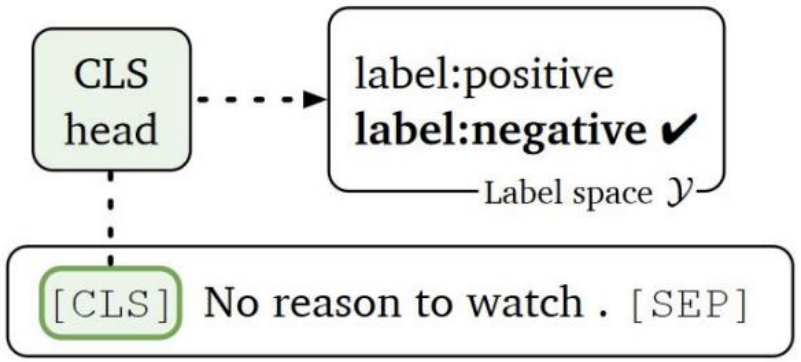
- Step 3: Fine-tune the weights of **full neural network** by propagating gradients on a downstream task



Prompt-based Fine Tuning

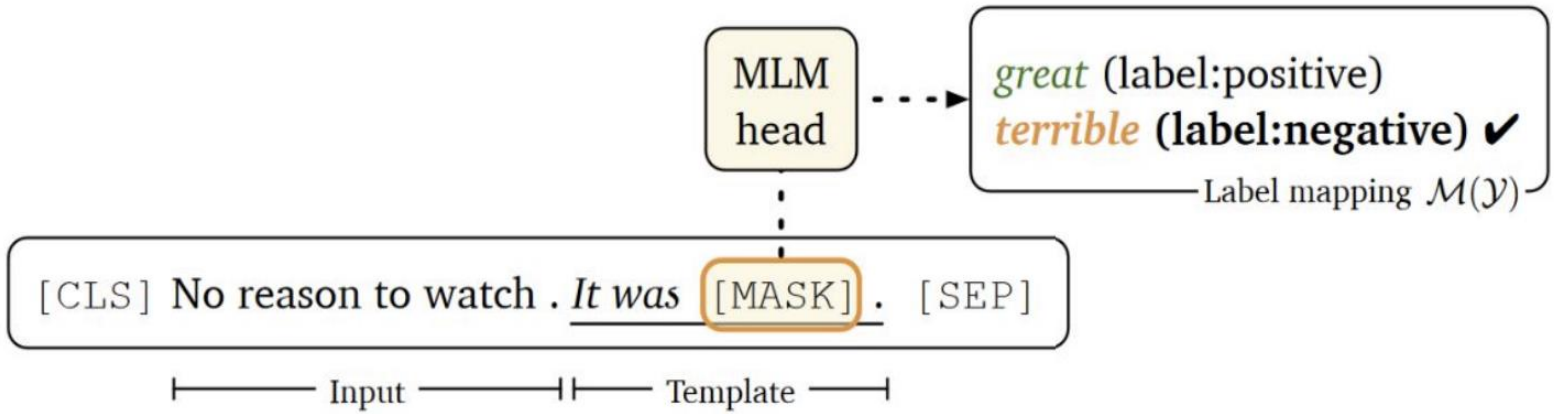
Prompt-based Fine-Tuning

classical head-based fine-tuning



Label as numeric: 0 / 1

Prompt-based fine-tuning



1. Prompt template
2. Label words

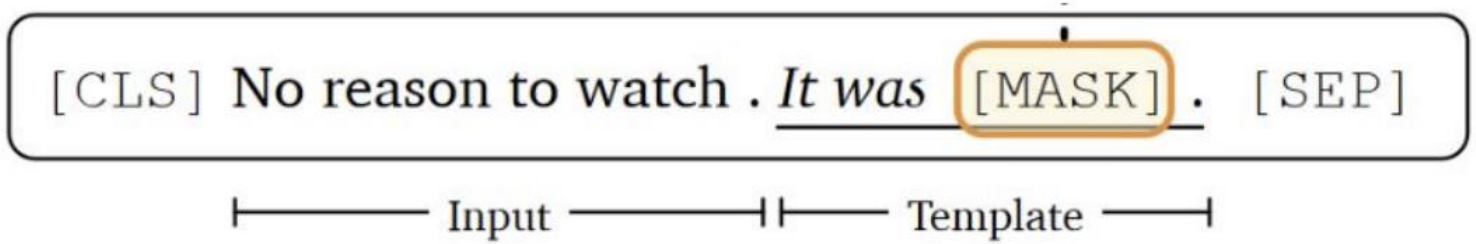
Classical Fine-Tuning vs Prompt-based Fine-Tuning

	Head-based	Prompt-based
New parameters?	Yes. hidden_size * num_classes	No
Few-shot friendly?	😓	😊

Prompt-based Fine-Tuning

Input: x_1 = No reason to watch.

Step 1. Formulate the downstream task into a (Masked) LM problem using a *template:*



Prompt-based Fine-Tuning

Input: x_1 = No reason to watch.

Step 1. Formulate the downstream task into a (Masked) LM problem using a *template:*

[CLS] No reason to watch . *It was* [MASK] . [SEP]

Input — Template

Step 2. Choose a *label word mapping* $\mathcal{M}$, which maps task labels to individual words.

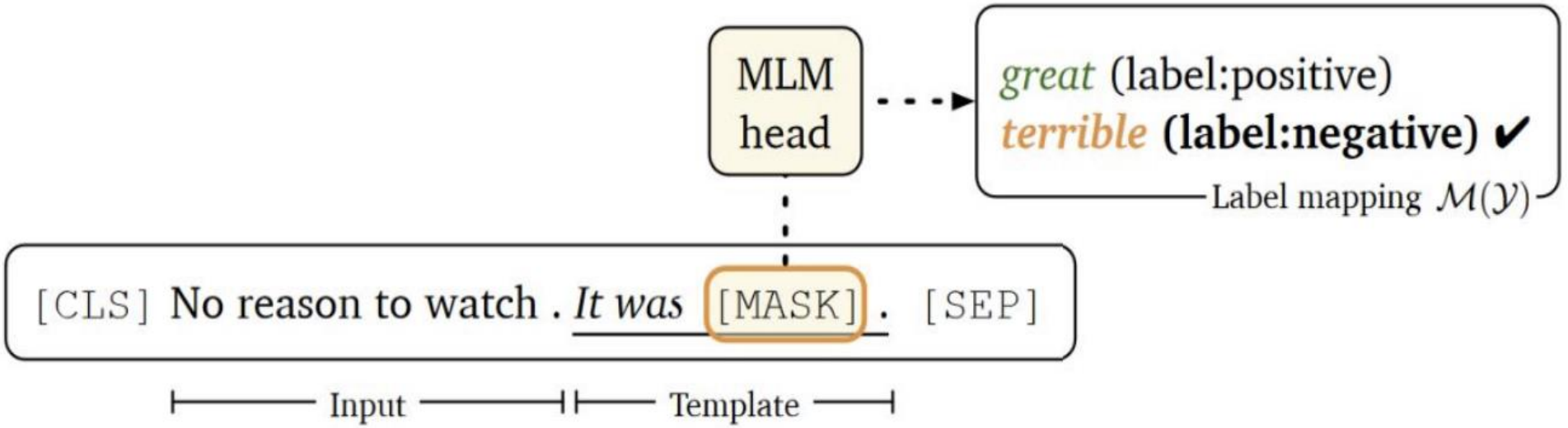
great (label:positive)
terrible **(label:negative)** ✔

Label mapping $\mathcal{M}(\mathcal{Y})$

Prompt-based Fine-Tuning

Step 3. Fine-tune the LM to fill in the correct label word.

$$p(y \mid x_{\text{in}}) = p\left(\texttt{[MASK]} = \mathcal{M}(y) \mid x_{\text{prompt}}\right)$$
$$= \frac{\exp\left(\mathbf{w}_{\mathcal{M}(y)} \cdot \mathbf{h}_{\texttt{[MASK]}}\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(\mathbf{w}_{\mathcal{M}(y')} \cdot \mathbf{h}_{\texttt{[MASK]}}\right)},$$



How to use Prompt-based Fine-Tuning

- Single Sentence Task

SST-2: sentiment analysis.

- E.g. **S1** = “The movie is ridiculous”. **Label**: negative.
- Manual prompt:

Template	Label words
$<S_1>$ It was `[MASK]` .	great/terrible

How to use Prompt-based Fine-Tuning

- Sentence Pair Task

SNLI: Natural Language Inference

- **S1** = "A soccer game with multiple males playing". **S2** = "Some men are playing sport". **Label**: Entailment.
- Manual prompt:

Template	Label words
$<S_1>$? [MASK] , $<S_2>$	Yes/Maybe/No

Prompt-based Fine-Tuning

Prompt-based fine-tuning involves:

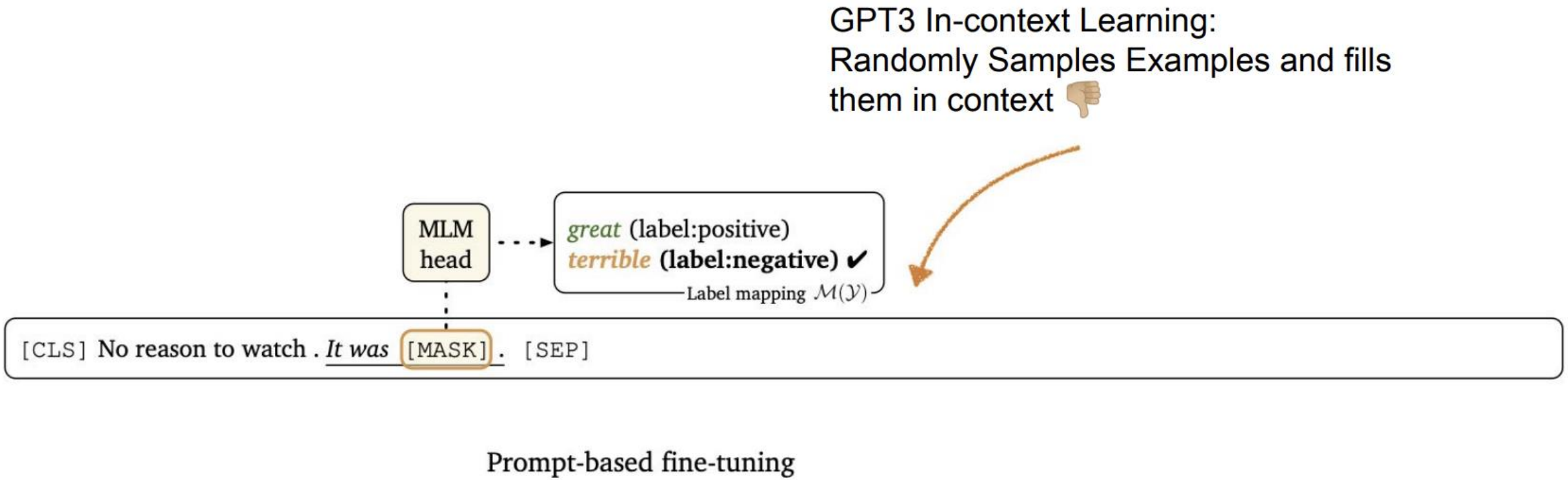
- a **template** which turns the downstream task into a (masked) language modelling problem, and
- a set of **label words** that map the textual output of the LM to the classification labels

Prompt-based Fine-Tuning

Prompt-based fine-tuning has the advantages:

- **No new parameters**: In this way, we don't need to introduce any new parameters so all the pre-trained parameters can be fine-tuned more sample-efficiently.
- **Few-shot learning capability**: It outperforms head-based fine-tuning in low-data setting because head-based fine-tuning introduces new randomly-initialized parameters (often more than 1k), which are hard to learn well from only a few examples.

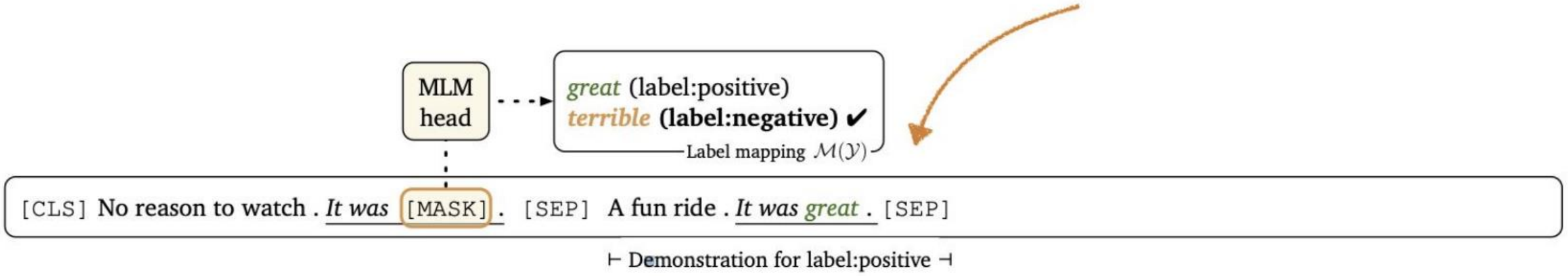
Prompt-based Fine-Tuning with Demonstrations



Prompt-based fine-tuning

Prompt-based Fine-Tuning with Demonstrations

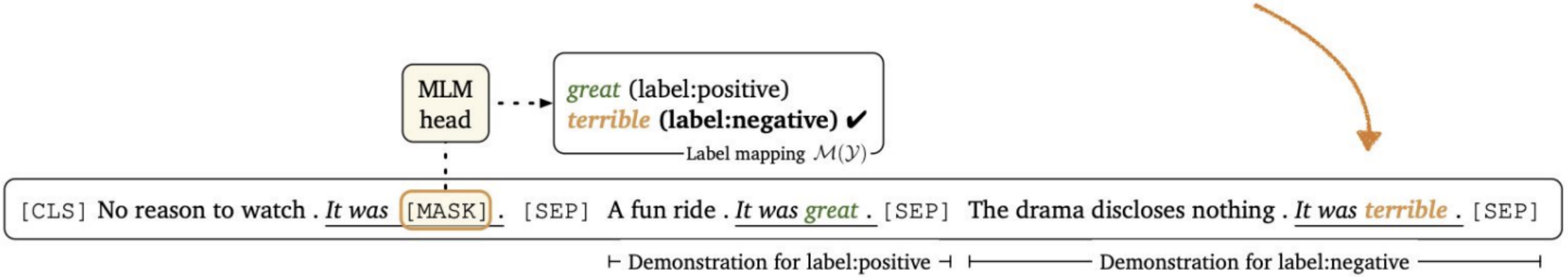
Improved: Selective Sampling, ie. for this example sample from then positive class 😎



Prompt-based fine-tuning with demonstrations

Prompt-based Fine-Tuning with Demonstrations

And we can also sample one from a negative training instance



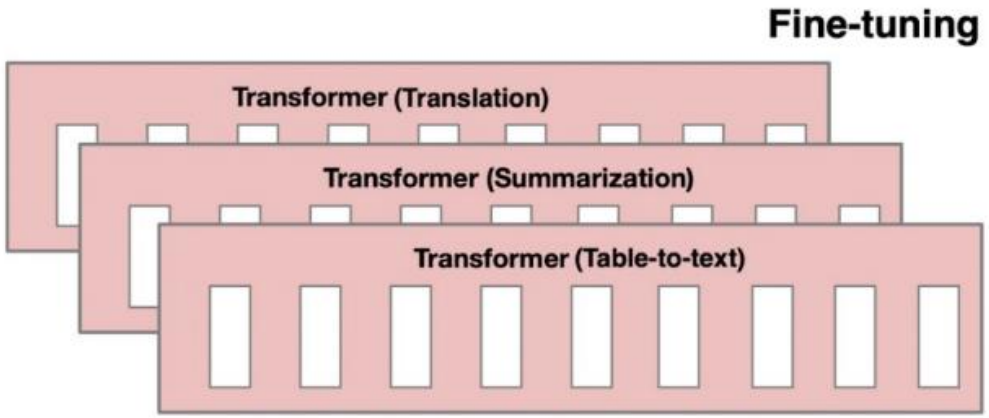
Prompt-based fine-tuning with demonstrations

Parameter Efficient Fine-Tuning

Problems of Classical Fine-Tuning

- **Computation Inefficiency:** high computational cost and time required to fine-tune them for each task

- **Storage Inefficiency:** With classical fine-tuning, we need to make a new copy of the model for each task.



Parameter Efficient Fine-Tuning

- Definition: PEFT means fine tuning only **a small subset of parameters** for each task, instead of updating all the parameters in the model

- Goal: PEFT aims to **reduce** the computation and storage requirements for fine-tuning while **maintaining** the model's performance on task-specific problems.

Advantages of Parameter Efficient Fine-Tuning

- **Reduced Training Cost:** because only a small portion of the model is updated, leading to faster iteration cycles and lower infrastructure cost
- **Resource Efficiency:** PEFT makes it possible to fine-tune large models on in resource-constrained device like mobile devices
- **Scalability:** easier to scale LLMs across various applications, because model storage requirement is significantly reduced for each task
- **Multi-Task Learning:** PEFT allows a single model to be adapted for multiple tasks efficiently, improves multi-tasking capabilities

Methods for Parameter Efficient Fine-Tuning

- Adapter
- Infused Adapter (IA)3
- Soft Prompt Tuning
- Prefix Tuning
- P-Tuning
- LLaMA-Adapter
- LoRA (Low Rank Adaptation)

Modifying only a small portion of the model while maintaining model performance on downstream tasks

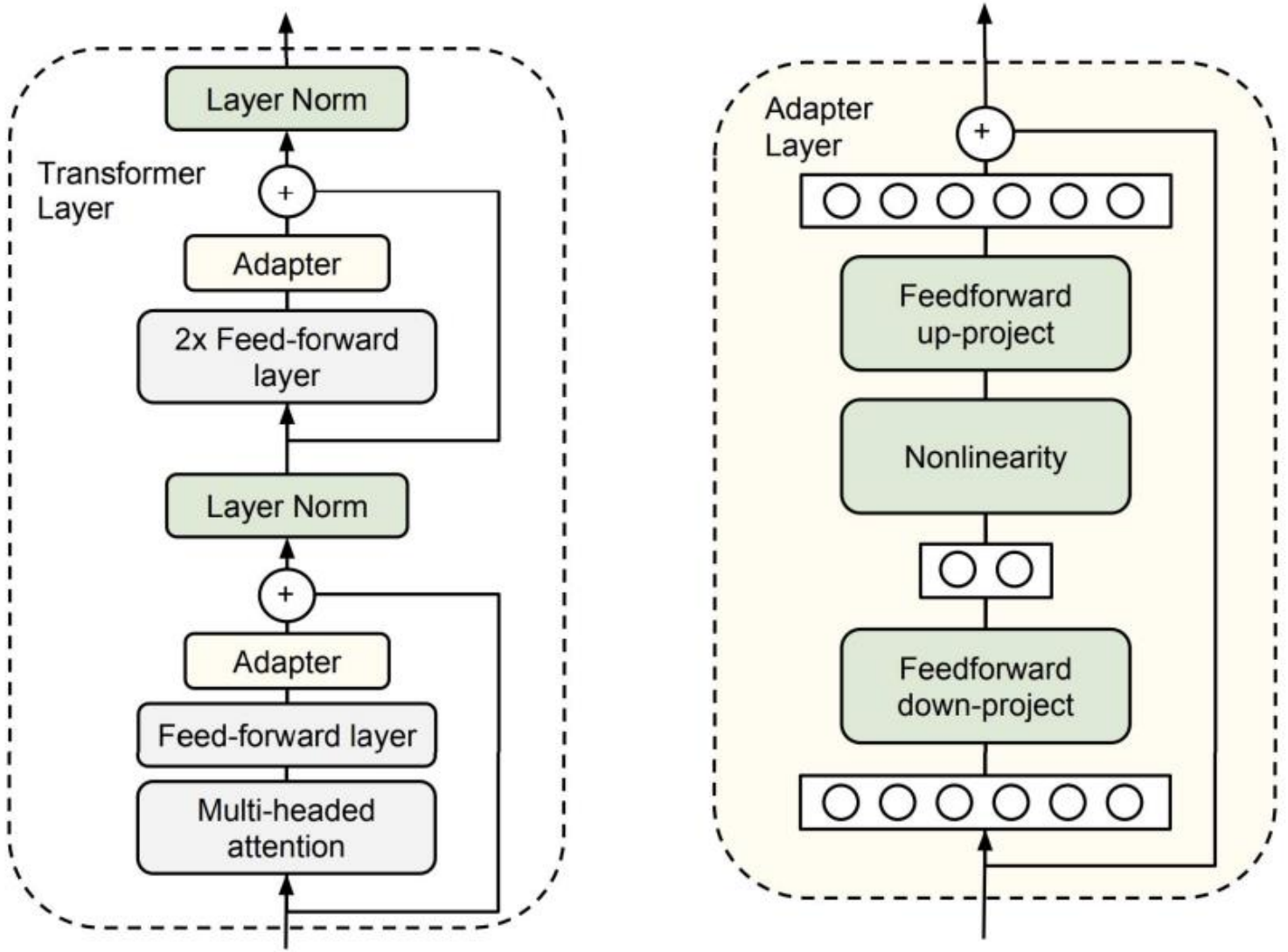
PEFT: Adapter

Adapter

- **Adapters** are small, trainable neural networks inserted between the layers of a pre-trained model.

- Not fine-tuning the entire model, only these adapter layers are tuned, while the remaining model's parameters are frozen.

Adapter


Layer Norm
Transformer Layer
Adapter
2x Feed-forward layer
Layer Norm
Adapter
Feed-forward layer
Multi-headed attention
Adapter Layer
Feedforward up-project
Nonlinearity
Feedforward down-project

Adapter Benefits

- Significantly **reduce** the number of parameters – Only 3.6% of learnable parameters needed

- The adapter layers learn **task-specific representations**, allowing the model to adapt to new tasks with minimal updates.

- Adapters are particularly useful in **multitask learning** where the same model needs to adapt to different domains or languages efficiently.

Code Implementation

```
def self_attention(x):
    k = x @ W_k
    q = x @ W_q
    v = x @ W_v
    return softmax(q @ k.T) @ v

def transformer_block(x):
    """ Pseudo code by author based on [2] """
    residual = x
    x = self_attention(x)
    x = layer_norm(x + residual)
    residual = x
    x = FFN(x)
    x = layer_norm(x + residual)
    return x
```

Pseudo code for the transformer block

Code Implementation

```
def transformer_block_adapter(x):
    """Pseudo code from [2] """
    residual = x
    x = self_attention(x)
    x = FFN(x)  # adapter
    x = layer_norm(x + residual)
    residual = x
    x = FFN(x)
    x = FFN(x)  # adapter
    x = layer_norm(x + residual)
    return x
```

Pseudo code for the adapter

PEFT: Infused Adapter IA3

Infused Adapter IA3

- Infused Adapter is also a method with additive parameters, augmenting the transformer block with some new parameters

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \longrightarrow \text{softmax}\left(\frac{Q(l_k \odot K^T)}{\sqrt{d_k}}\right)(l_v \odot V)$$

Attention in normal transformer

Adding new column vectors into attention

Infused Adapter IA3

- Infused Adapter is also a method with additive parameters, augmenting the transformer block with some new parameters

$$\gamma(W_1 x)W_2 \longrightarrow (l_{ff} \odot \gamma(W_1 x))W_2$$

Feedforward layers in normal transformer

Adding new column vector into feedforward layer

Infused Adapter IA3

```
def self_attention_ia3(x):
    k = x @ W_k
    q = x @ W_q
    v = x @ W_v

    k = l_k @ k  # ia3
    v = l_v @ v  # ia3

    return softmax(q @ k.T) @ v

def transformer_block_ia3(x):
    """Pseudo code from [2]"""
    residual = x
    x = self_attention_ia3(x)
    x = layer_norm(x + residual)
    residual = x
    x = x @ W_1  # normal transformer
    x = l_ff * gelu(x)  # ia3
    x = x @ W_2
    x = layer_norm(x + residual)
    return x
```

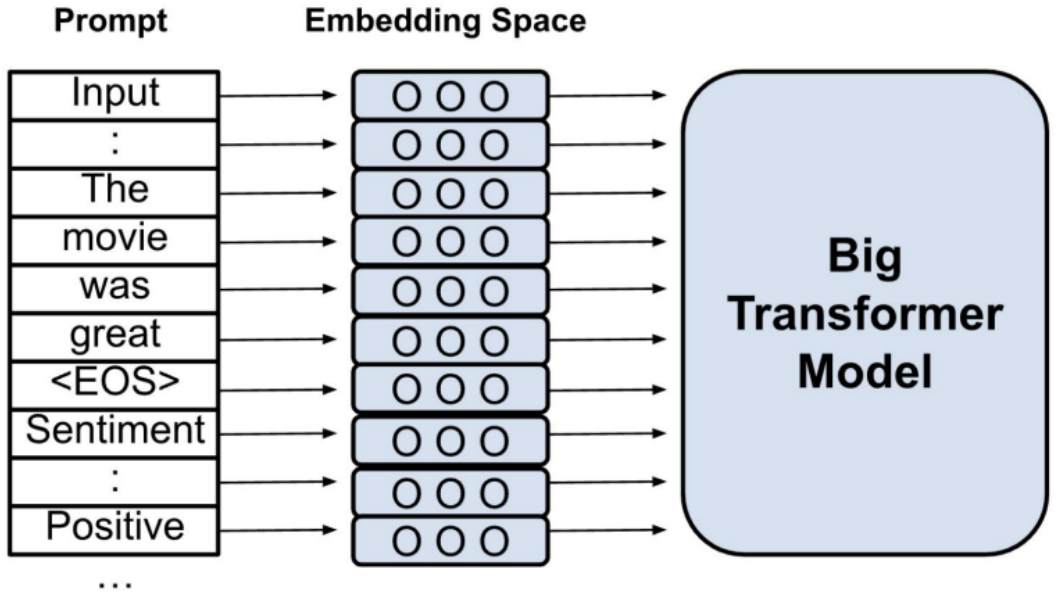
$$\text{softmax}\left(\frac{Q(l_k \odot K^T)}{\sqrt{d_k}}\right)(l_v \odot V)$$

$$(l_{ff} \odot \gamma(W_1 x))W_2$$

PEFT: Soft Prompt Tuning

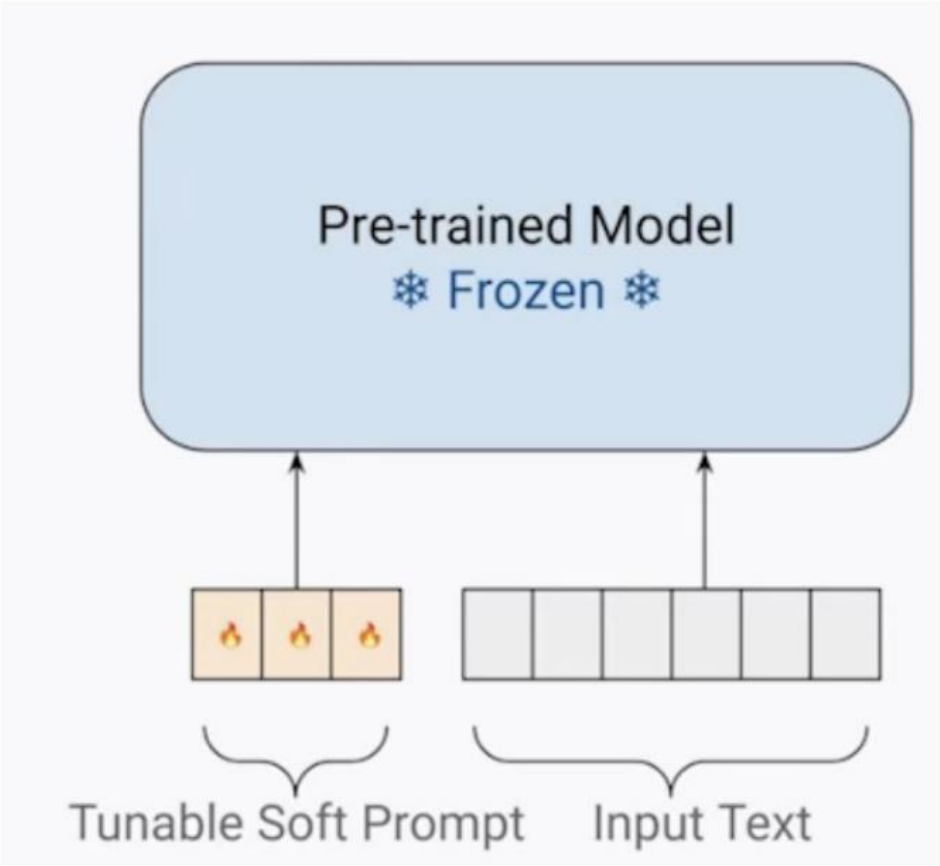
Motivation of Soft Prompt Tuning

- Recall prompt-based fine-tuning, the discrete prompt is optimized manually for prompt design.
- Optimization in discrete space is hard!
- Idea: What if we can optimize the prompt in the **continuous embedding space**?



Soft Prompt Tuning

- Concatenate **soft tokens** (represented as **continuous embedding**) with the input sequence
- Update embeddings of only these soft tokens via backpropagation. Keep the rest of model parameters fixed



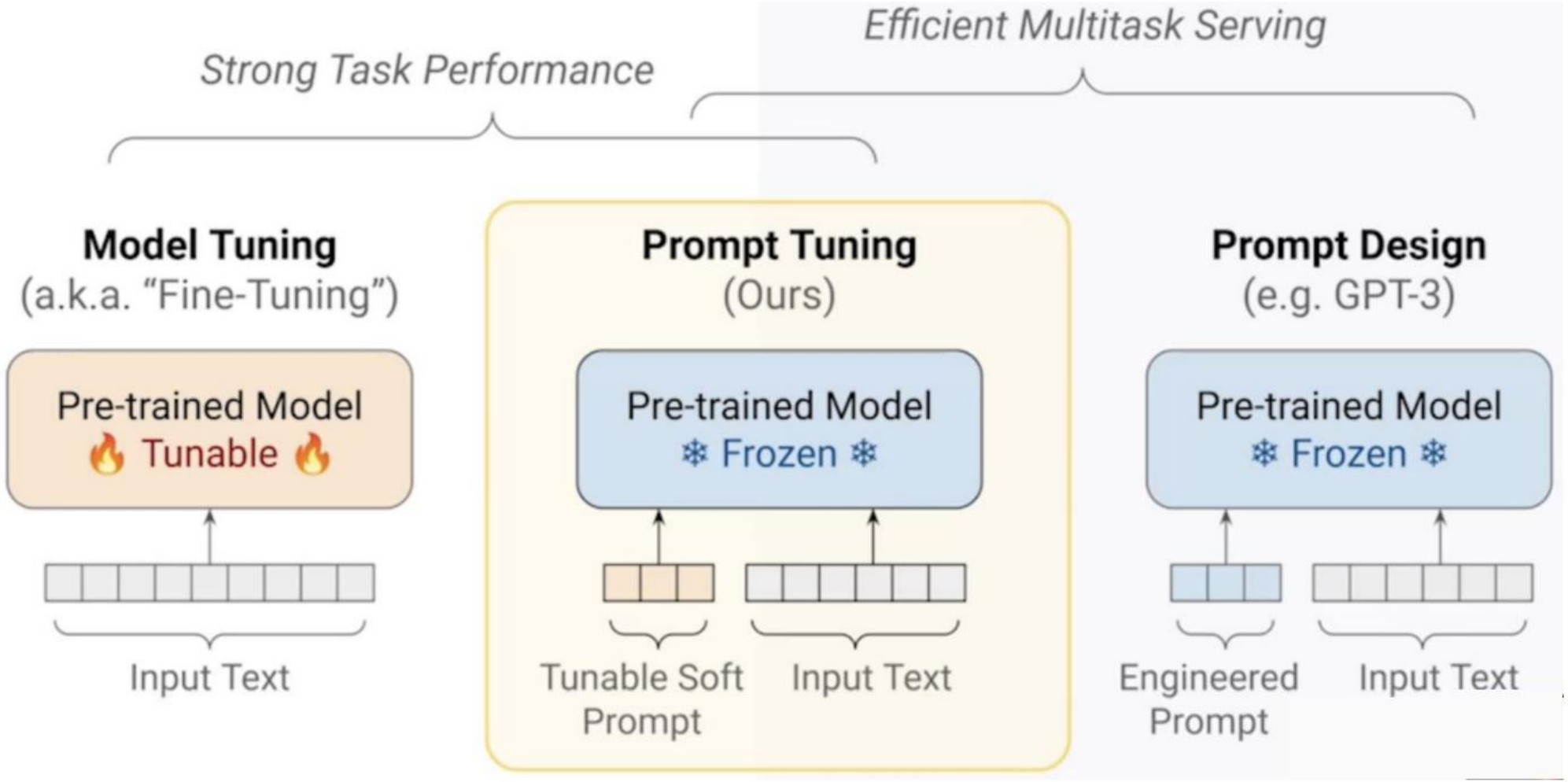
Soft Prompt Tuning

```
def prompt_tuning(seq_tokens, prompt_tokens):
    """ Pseudo code from [2]. """
    x = seq_embedding(seq_tokens)
    soft_prompt = prompt_embedding(prompt_tokens)
    model_input = concat([soft_prompt, x], dim=seq)
    return model(model_input)
```

Input text sequence

Soft prompt tokens

Soft Prompt Tuning



Soft Prompt Tuning

- Scalability
  - Model tuning: fairly good → poor
  - Prompt tuning: very good
  - Prompt design: poor
- Reason
  - Model tuning: possibly over-parameterized when model size is large
  - Prompt tuning: reserve general understanding and include tunable parameter
  - Prompt design: not fitting downstream tasks well

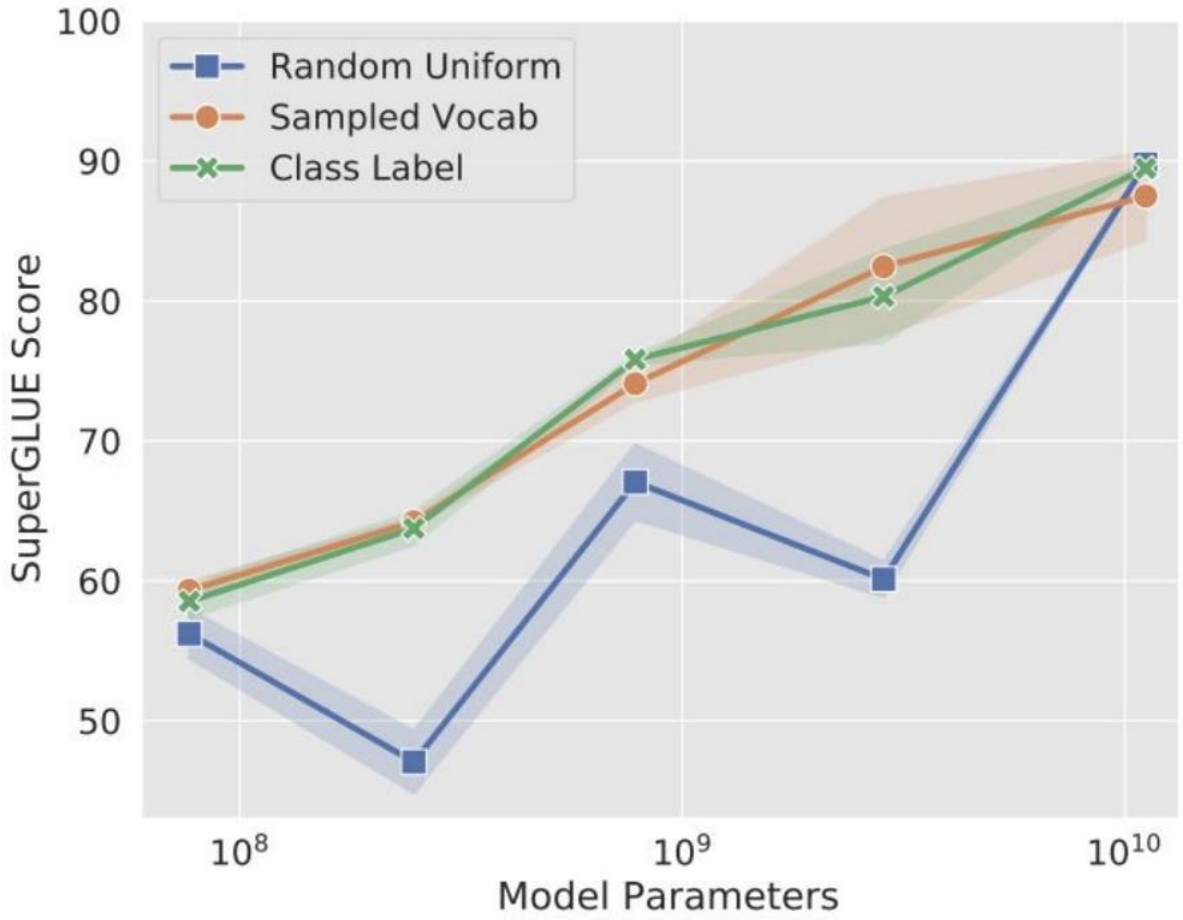
Soft Prompt Tuning – soft prompt initialization

- **Random initialization:** train soft prompt representations from scratch
- **Sampled vocabulary:** initialize each soft prompt to an embedding drawn from the model's vocabulary
- **Class label:** initialize soft prompt with embeddings of the output classes

Soft Prompt Tuning – soft prompt initialization

Observation:

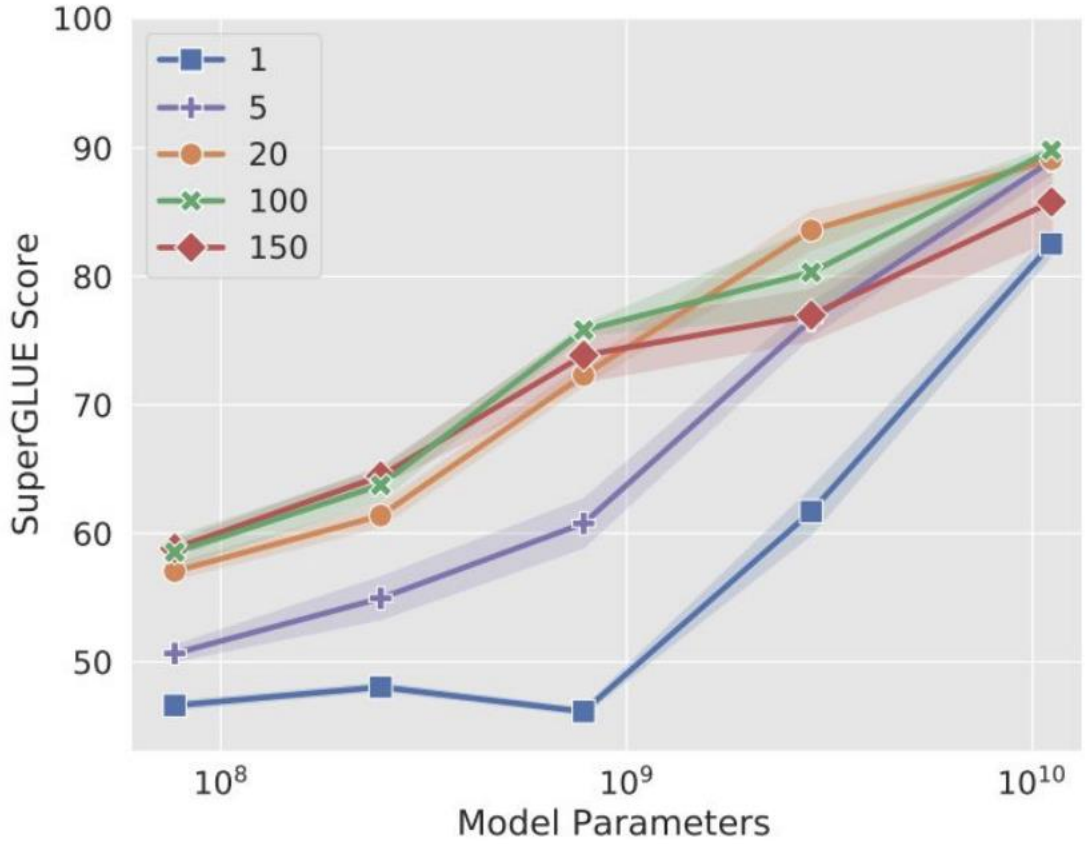
- Class label-based initialization performs best
- The gaps between different initializations disappear when the model is scaled to XXL size



Soft Prompt Tuning – soft prompt length

Observation:

- Increasing prompt length is critical to achieve good performance
- The large model still gives strong results with a single-token prompt
- Increasing beyond 20 soft tokens only yields marginal gains



PEFT: Prefix Tuning

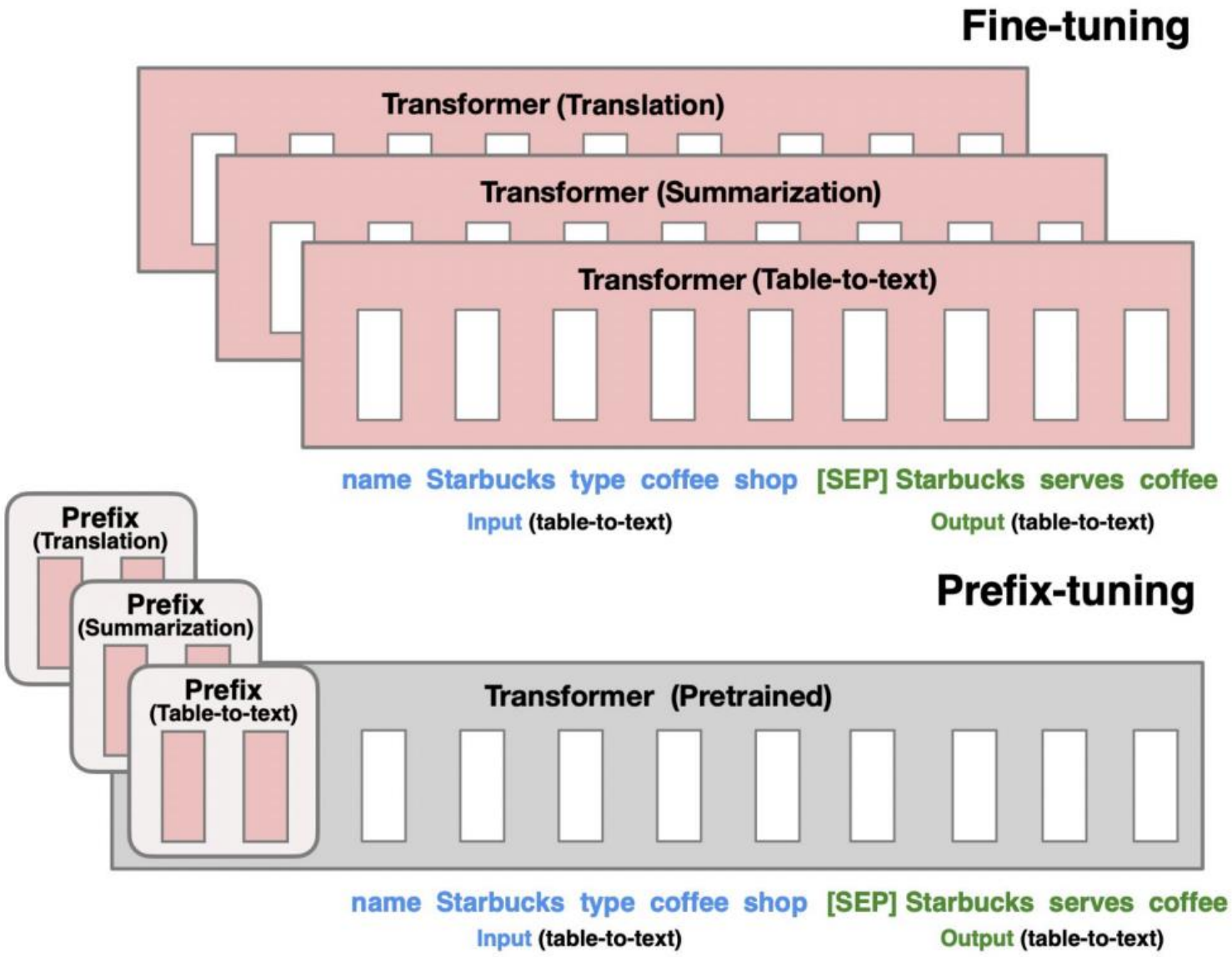
Prefix Tuning

- **Similar** to Soft Prompt Tuning: find a **continuous representation** from a separate set of prompt tokens that are input into the base model.

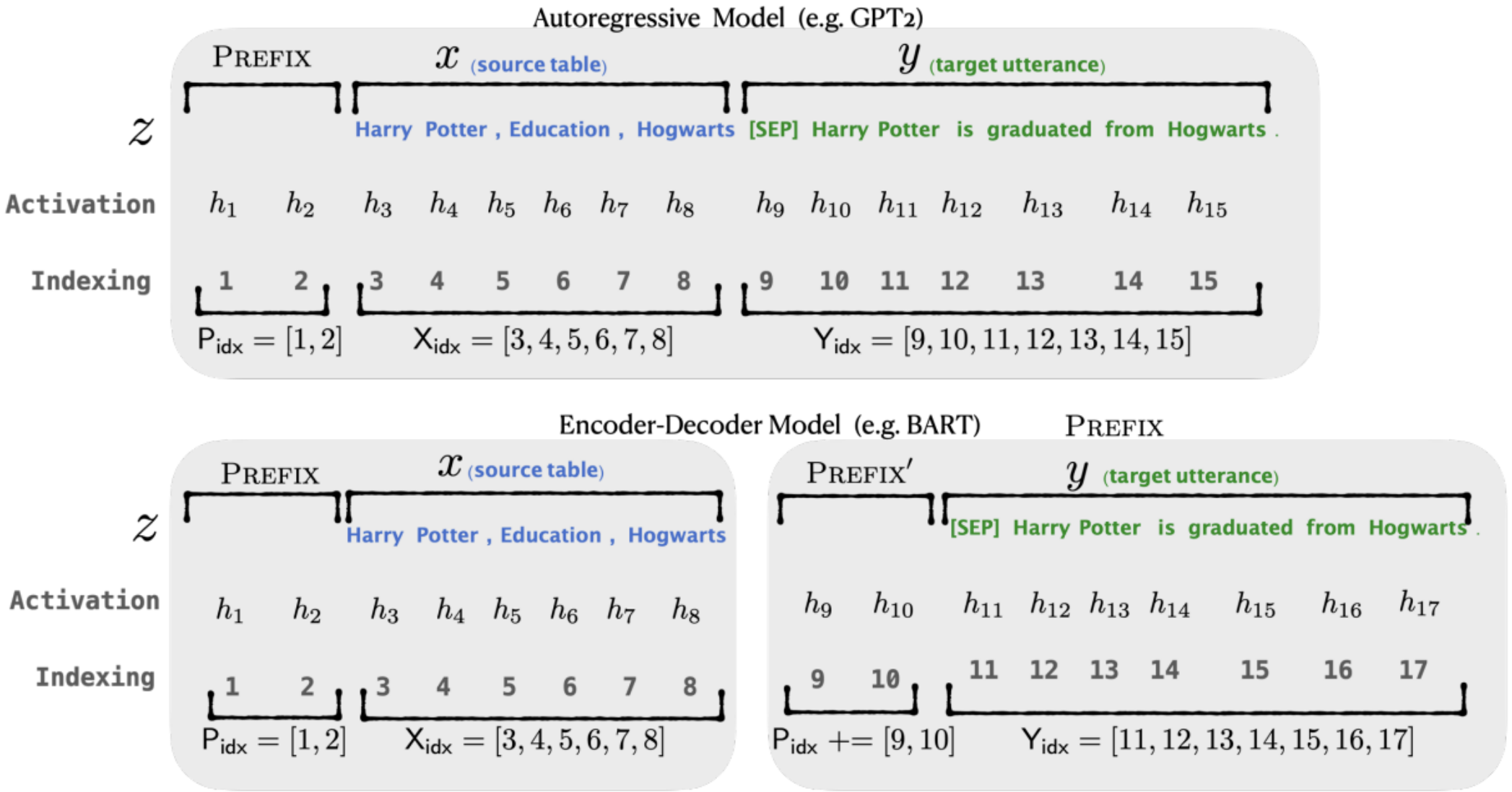
- **Extend** Soft Prompt Tuning: the representation is fed to **all layers** of the transformer whereas prompt tuning was only concatenated with the **initial embedding layer**

Prefix Tuning

Only 0.1% of parameters need to be tuned



Prefix Tuning (decoder and encoder-decoder LM)



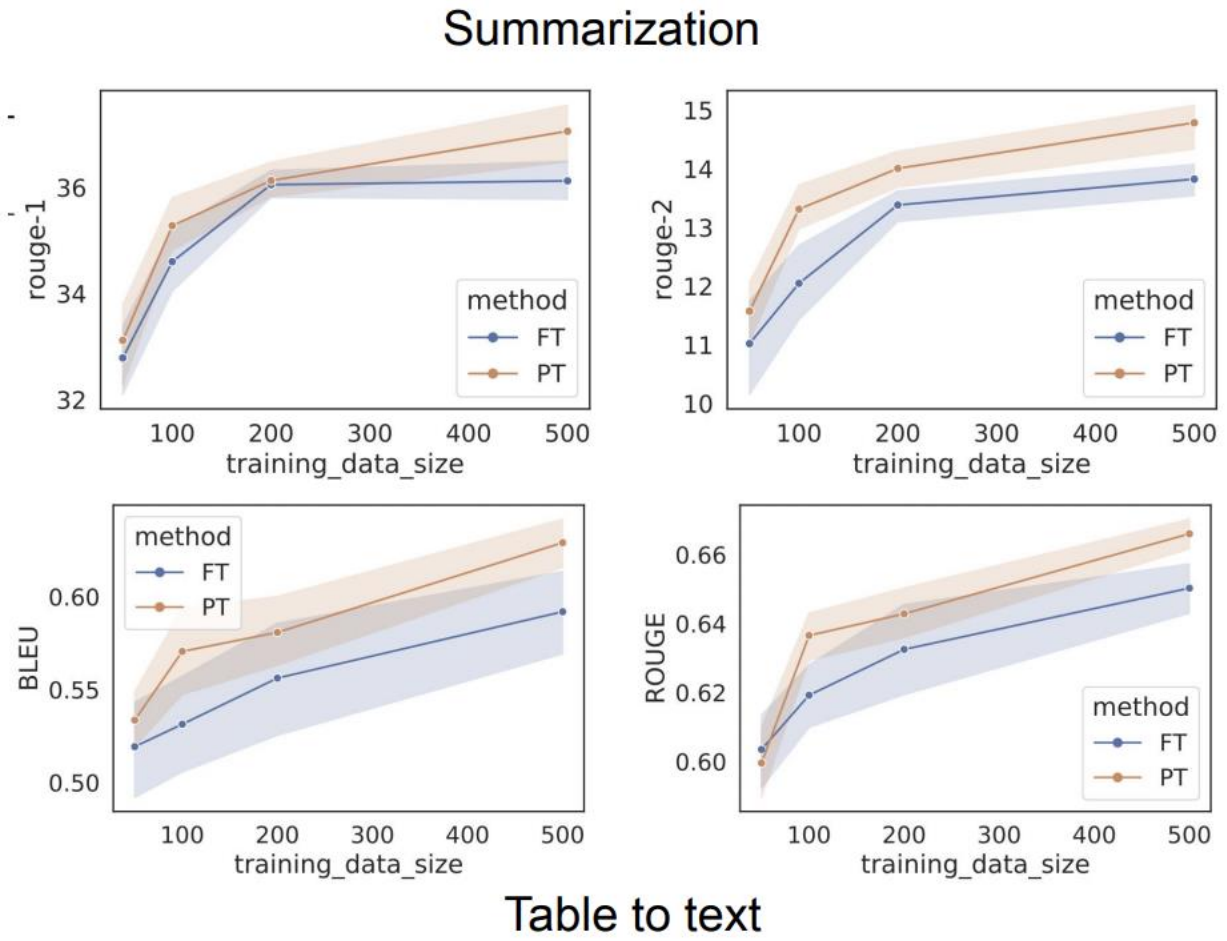
Prefix Tuning

```
def transformer_block_prefix_tuning(x, soft_prompt):
    """ Pseudo code from [2] """
    soft_prompt = FFN(soft_prompt)
    model_input = concat([soft_prompt, x], dim=seq)
    return model(model_input)
```

Prefix Tuning – low data setting

Observation

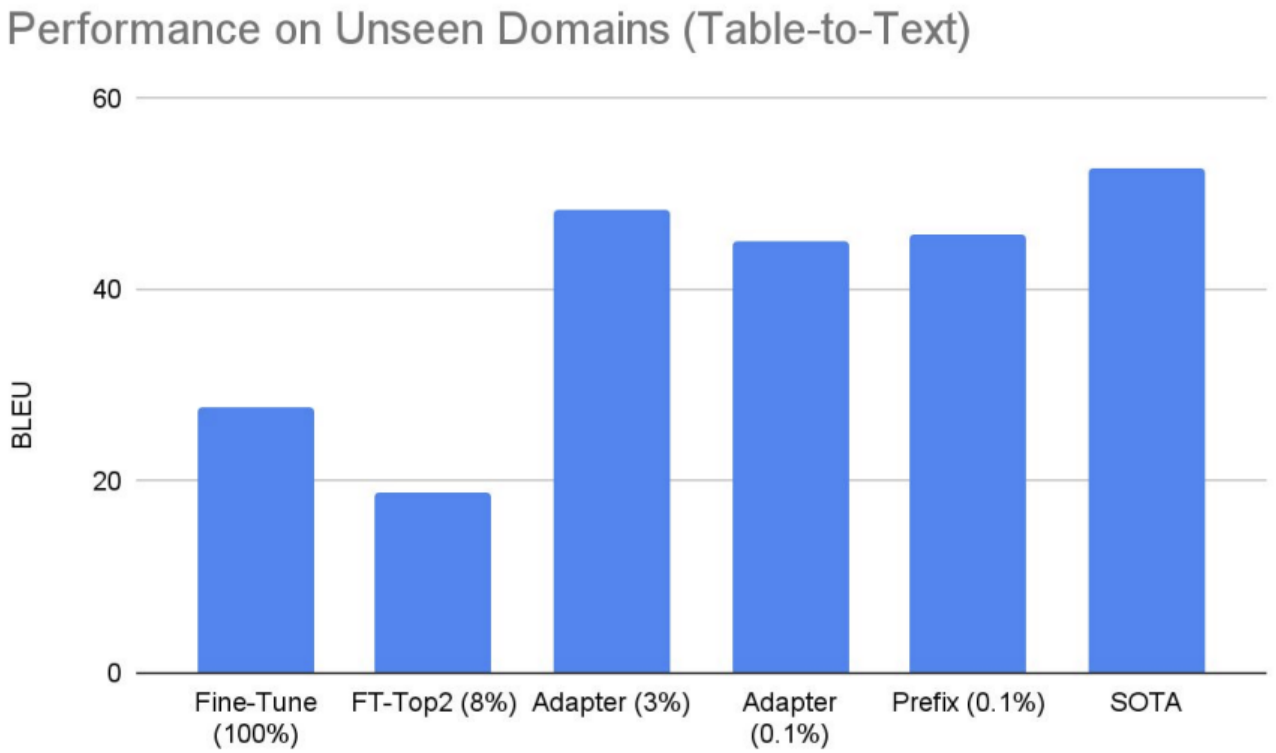
- Prefix tuning outperforms fine-tuning when training data size is low



Prefix Tuning – generalize to new domain

Observation

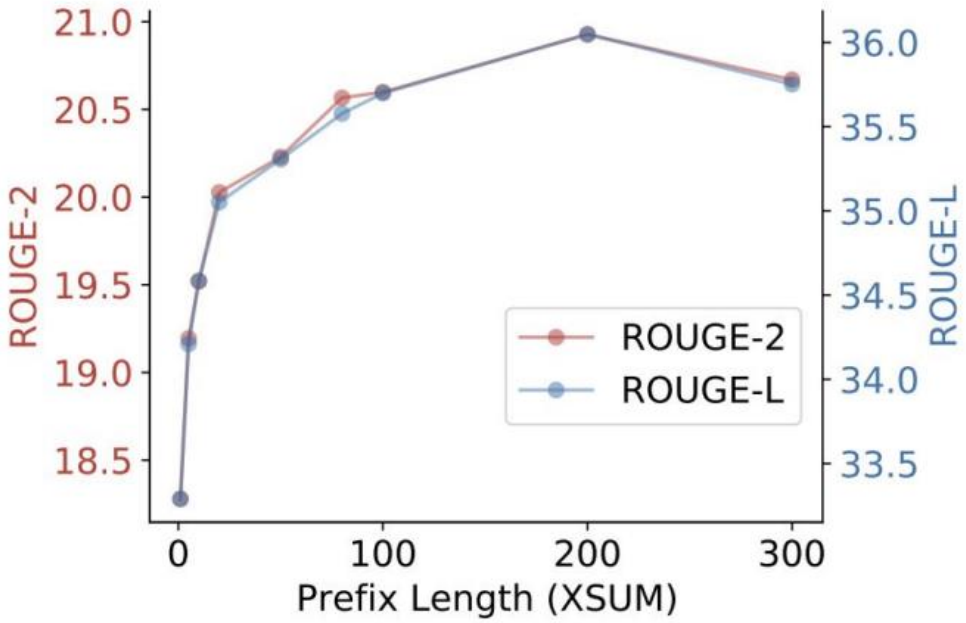
- Prefix tuning outperforms fine-tuning on generalization to different domains

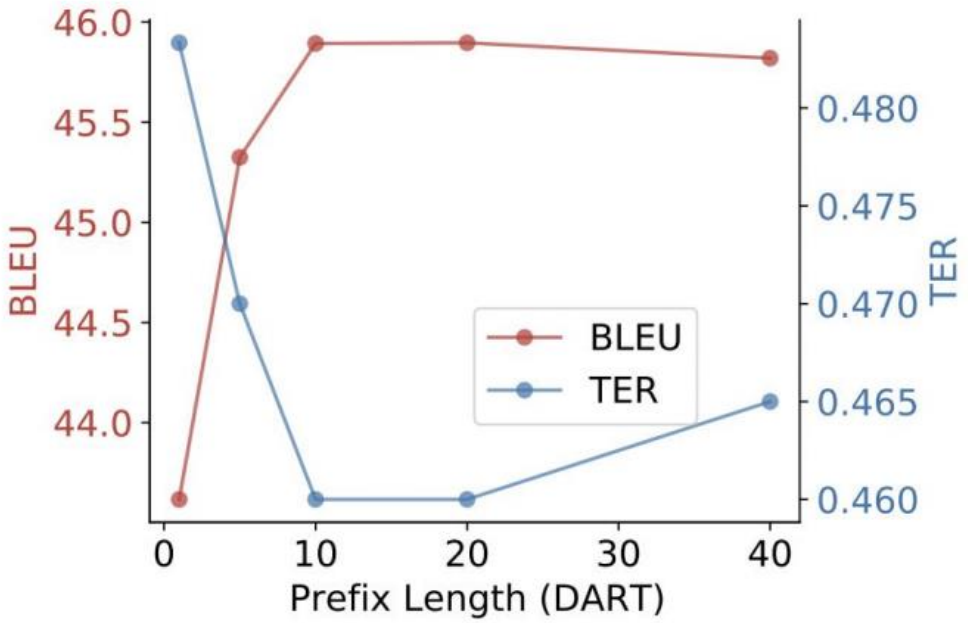


Prefix Tuning – prefix length

Observation

- As the tunable prefix-length increases, performance increases, with decreasing marginal benefits. (left: summarization, right: table to text)

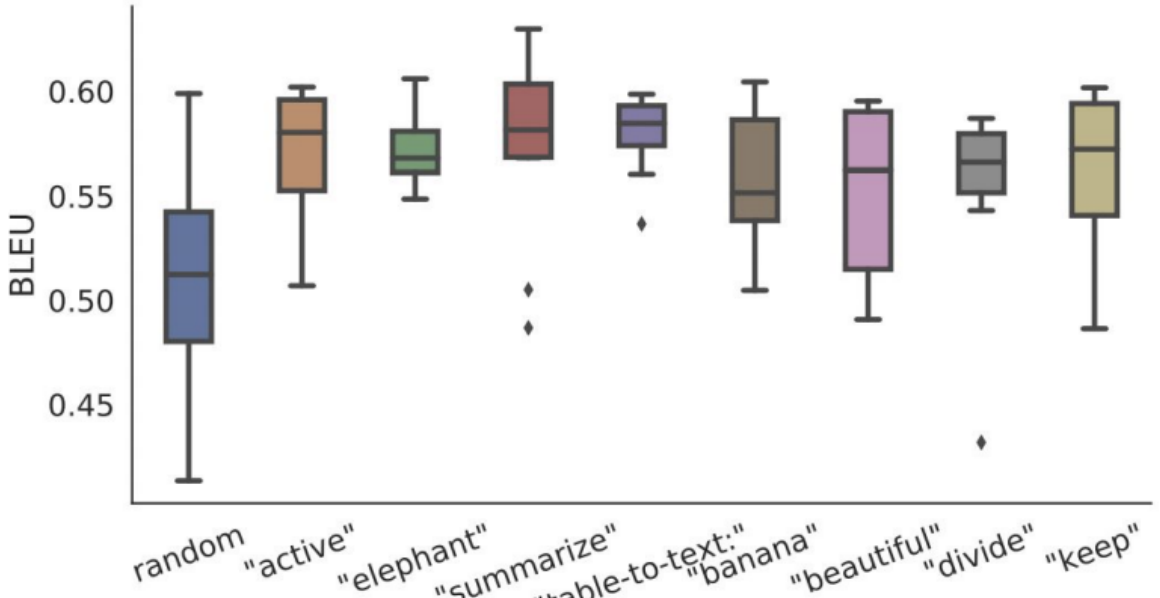




Prefix Tuning – prefix initialization

Observation

- Initializing randomly performs poorly and has high variance
- It's better to initialize with words in the LM's vocabulary
- It's even better to initialize with task specific words (summarize / table-to-text)



PEFT: P-Tuning

P-Tuning

- **Similar** to Prompt / Prefix Tuning: tune the **continuous representation** (soft prompt) instead of tuning on discrete words (hard text prompt)

- **Difference**:

  1. P-Tuning adds **LSTM** on soft tokens
  2. P-Tuning adds **anchor words** to prompt

P-Tuning – adding LSTM on soft tokens

- **Motivation:** soft tokens are technically independent of each other in prompt-tuning and prefix-tuning.

- **Solution:** P-Tuning adds **LSTM** on soft tokens to made the soft tokens dependent on each other

P-Tuning – adding anchor words

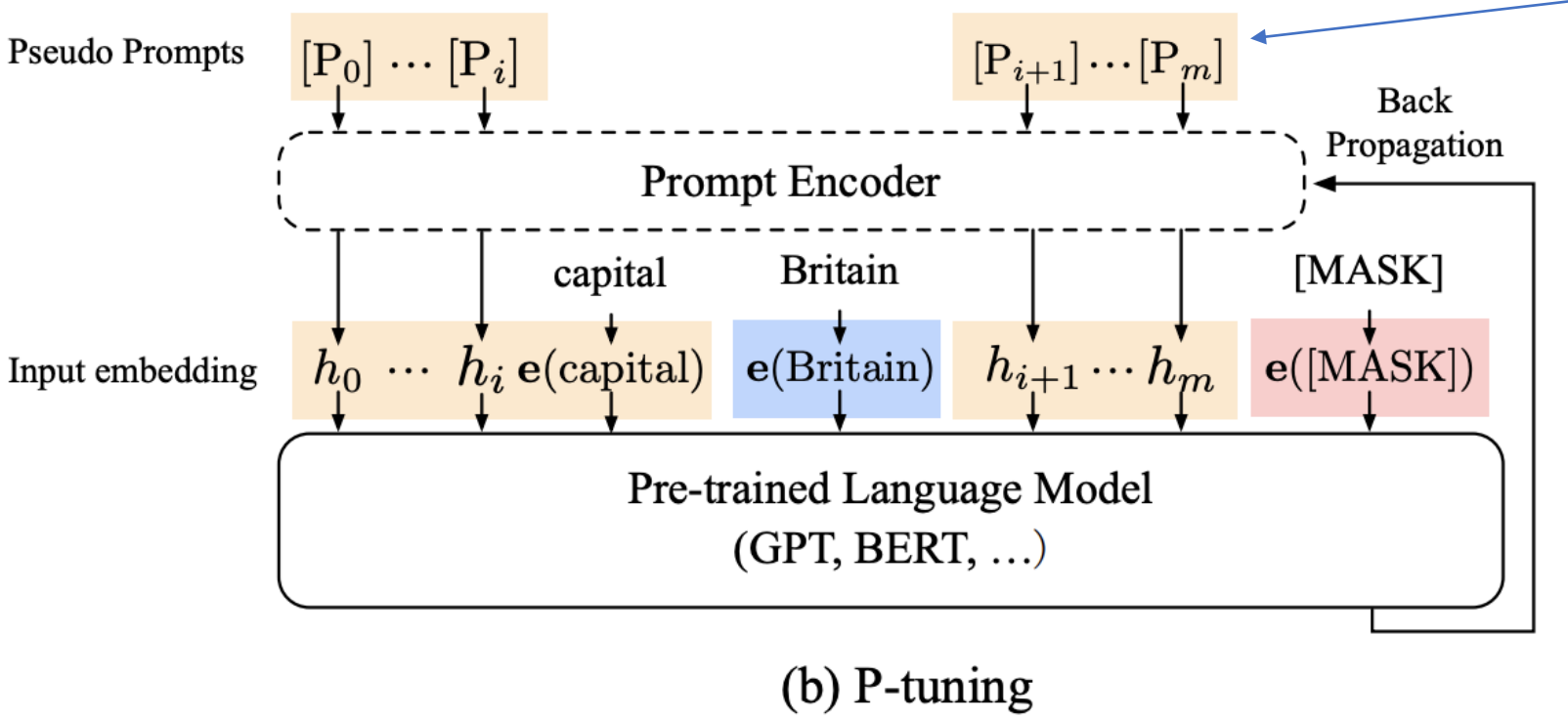
- **Motivation:** the same semantics can be expressed in various ways, but anchor words remain the same

Example: “The capital of [X] is [Y].” e.g. (Britain, London)

Prompt	P@1 w/o PT	P@1 w/ PT
[X] is located in [Y]. *(original)*	31.3	57.8
[X] is located in which country or state? [Y].	19.8	57.8
[X] is located in which country? [Y].	31.4	58.1
[X] is located in which country? In [Y].	51.1	58.1

P-Tuning – adding anchor words

- **Solution:** keep salient words as discrete tokens in prompt, and concatenate with soft prompt tokens (that replace unimportant tokens)



(b) P-tuning

remember LSTM layer here

P-Tuning

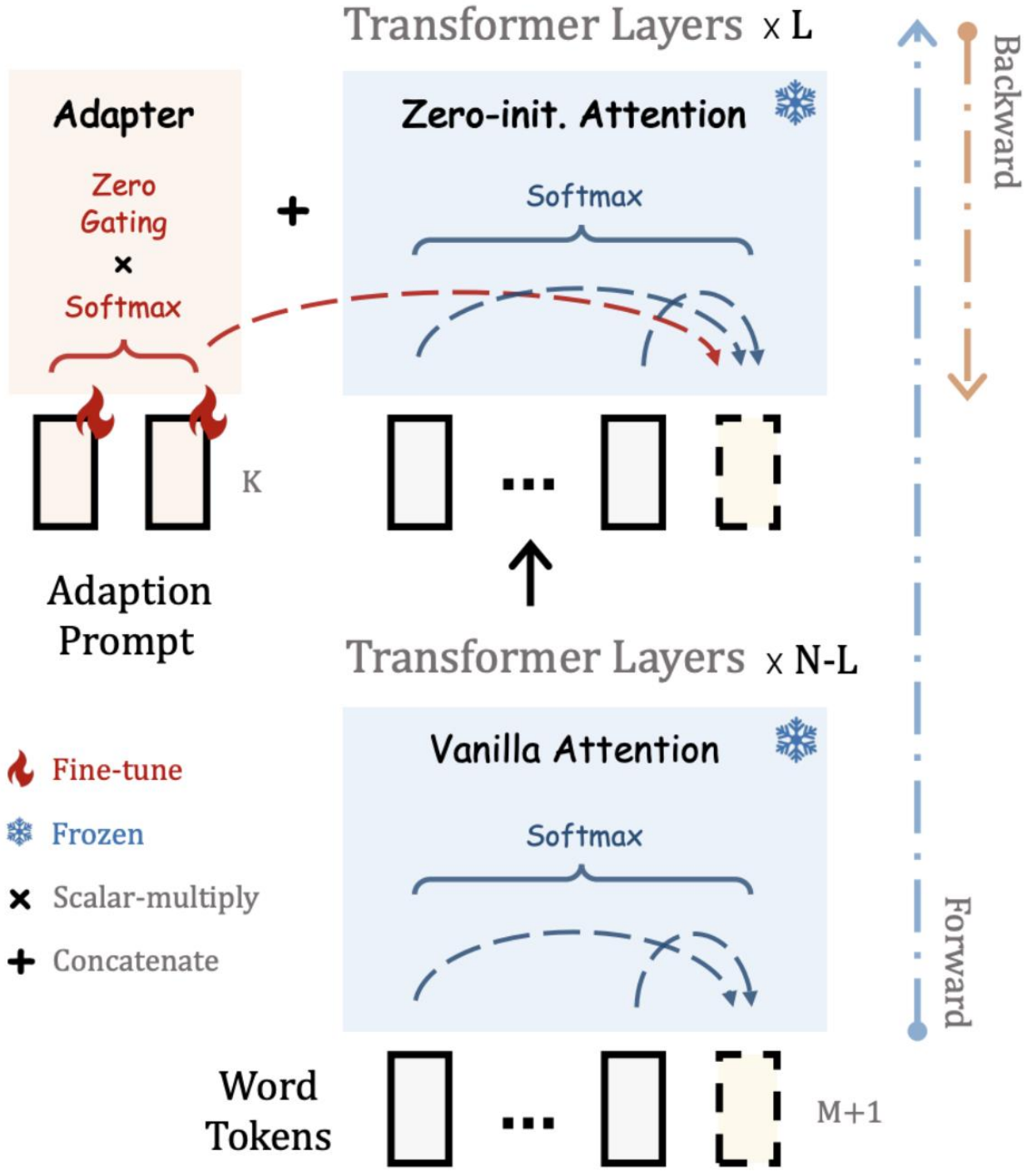
```
def p_tuning(seq_tokens, prompt_tokens):
    """Pseudo code for p-tuning created by Author."""
    h = prompt_embedding(prompt_tokens)
    h = LSMT(h, bidirectional=True)
    h = FFN(h)

    x = seq_embedding(seq_tokens)
    model_input = concat([h, x], dim=seq)

    return model(model_input)
```

PEFT: LLaMA-Adapter

LLaMA-Adapter

- **Similar** to Prefix Tuning: concatenate continuous representation into (L topmost) transformer layers

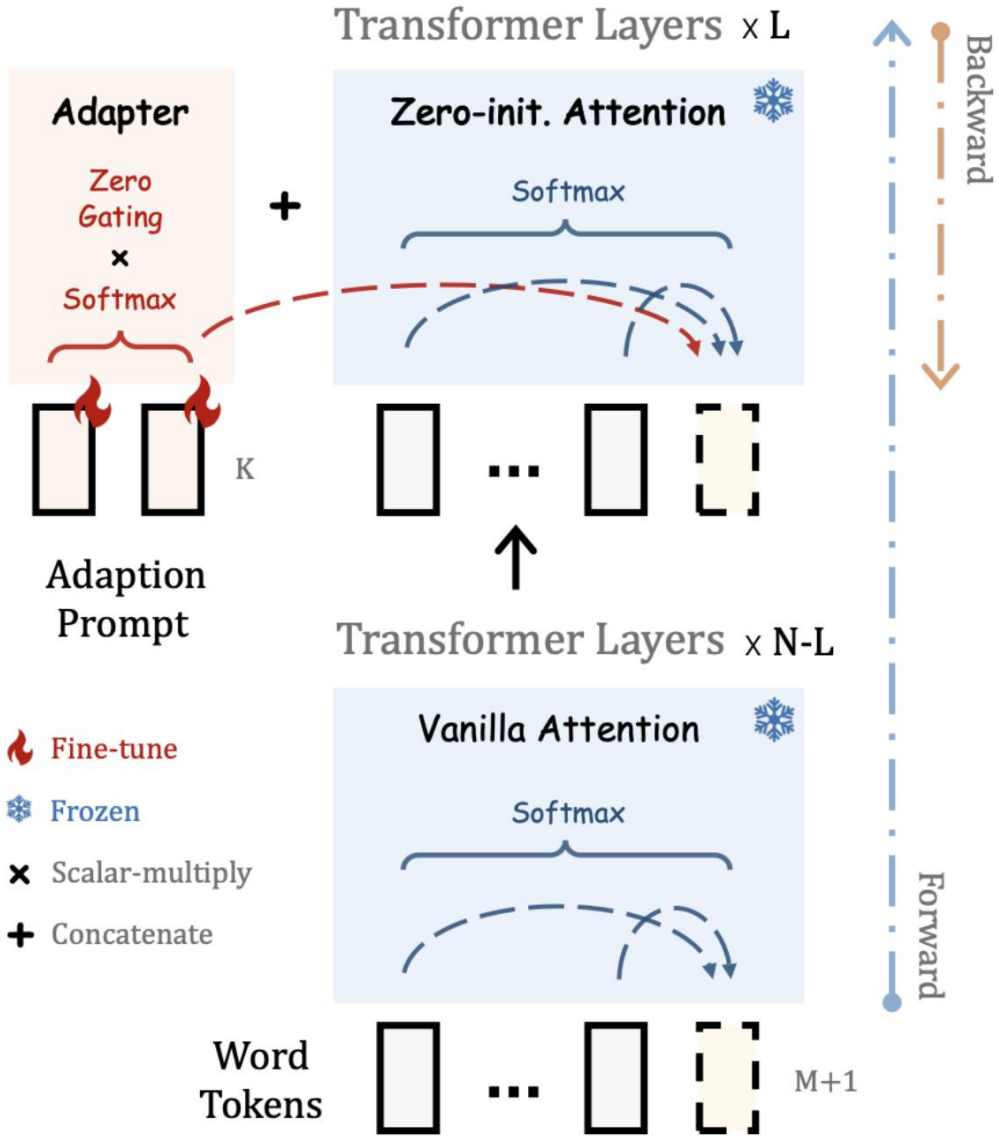
- **Difference** with Prefix Tuning: introduce **zero initialized attention**



LLaMA-Adapter – zero initialized attention

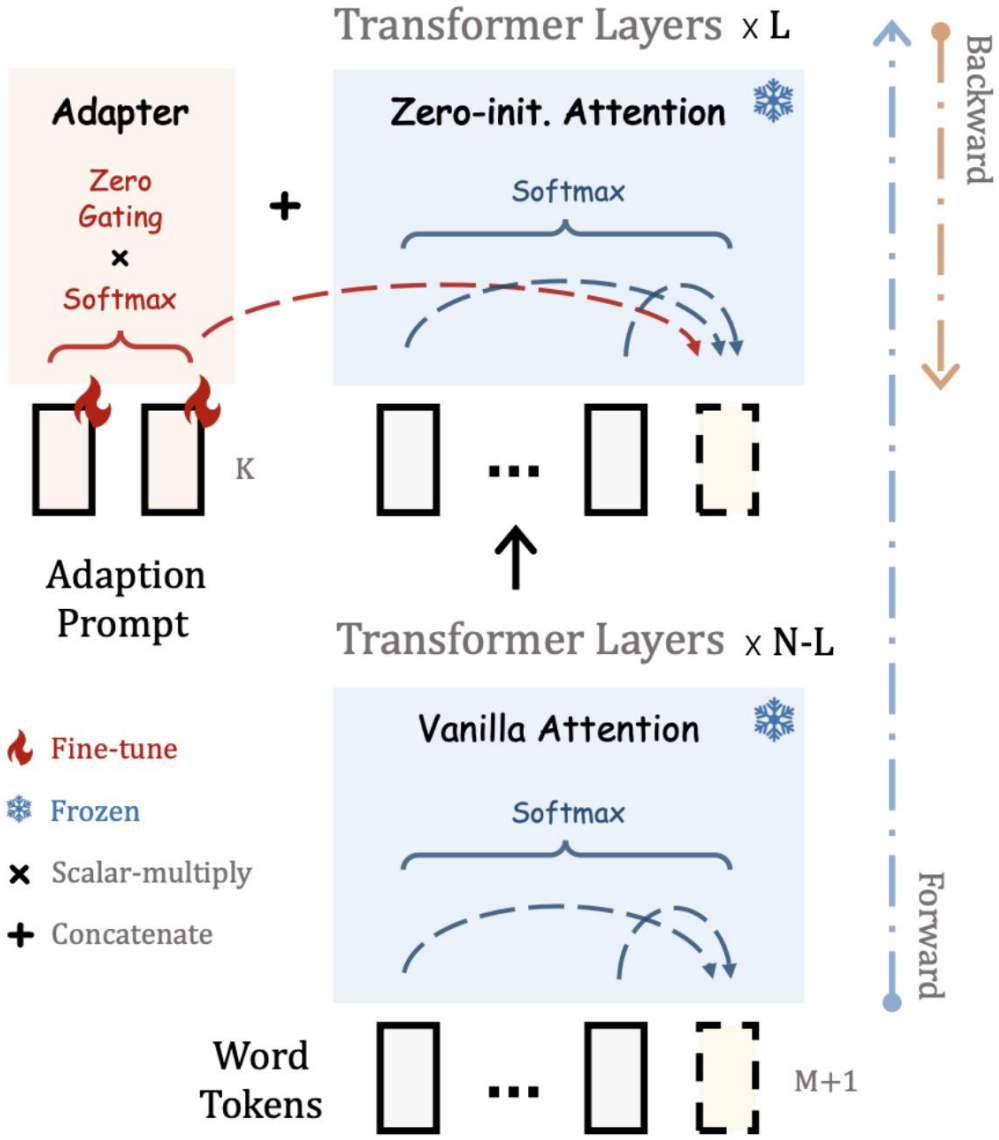
- **Motivation:** newly added parameters have random initialization, which introduce **random noise** to LM, causing **large loss** values at early stages and making finetuning **unstable**

- **Solution:** Zero Initialized Attention



LLaMA-Adapter – zero initialized attention

- Introducing a **gating factor**, initialized to 0, that is multiplied by the self attention mechanism.
- **Zero initialized attention** = product of **gating factor** and **self-attention**
- The **gating factor** is adaptively tuned over training to create a smoother update of the network parameters



LLaMA-Adapter

```
def transformer_block_llama_adapter(x, soft_prompt, gating_factor)
    """LLaMA-Adapter pseudo code created by Author"""
    residual = x

    adaption_prompt = concat([soft_prompt, x], dim=seq)
    adaption_prompt = self_attention(adaption_prompt) * gating_fac

    x = self_attention(x)
    x = adaption_prompt * x
    x = layer_norm(x + residual)
    residual = x
    x = FFN(x)
    x = layer_norm(x + residual)

    return x
```

Zero-initialized attention
Adding gating factor

PEFT: LoRA (Low Rank Adaptation)

Recap different types of PEFT

- Adapter-Based Methods (Adapter, Infused Adapter): introduce **new set of parameters** inserted between the layers of a pre-trained LM.

- Soft Prompt-Based Methods (Soft Prompt Tuning, Prefix Tuning, P Tuning): prepend **continuous representation** into transformer layers

- Reparameterization-Based Methods (LoRA): use **low dimensional representations** to learn the **weight difference** after fine-tuning

LoRA – weight difference

Idea:

- The weight matrix (like K, Q, V) in pre-trained LM is $W \in R^{dxk}$
- The weight matrix after fine-tuning is $W + \Delta W \in R^{dxk}$
- LoRa updates a weight matrix W by learning a separate matrix ΔW which represents the **weight difference** gained from fine-tuning.

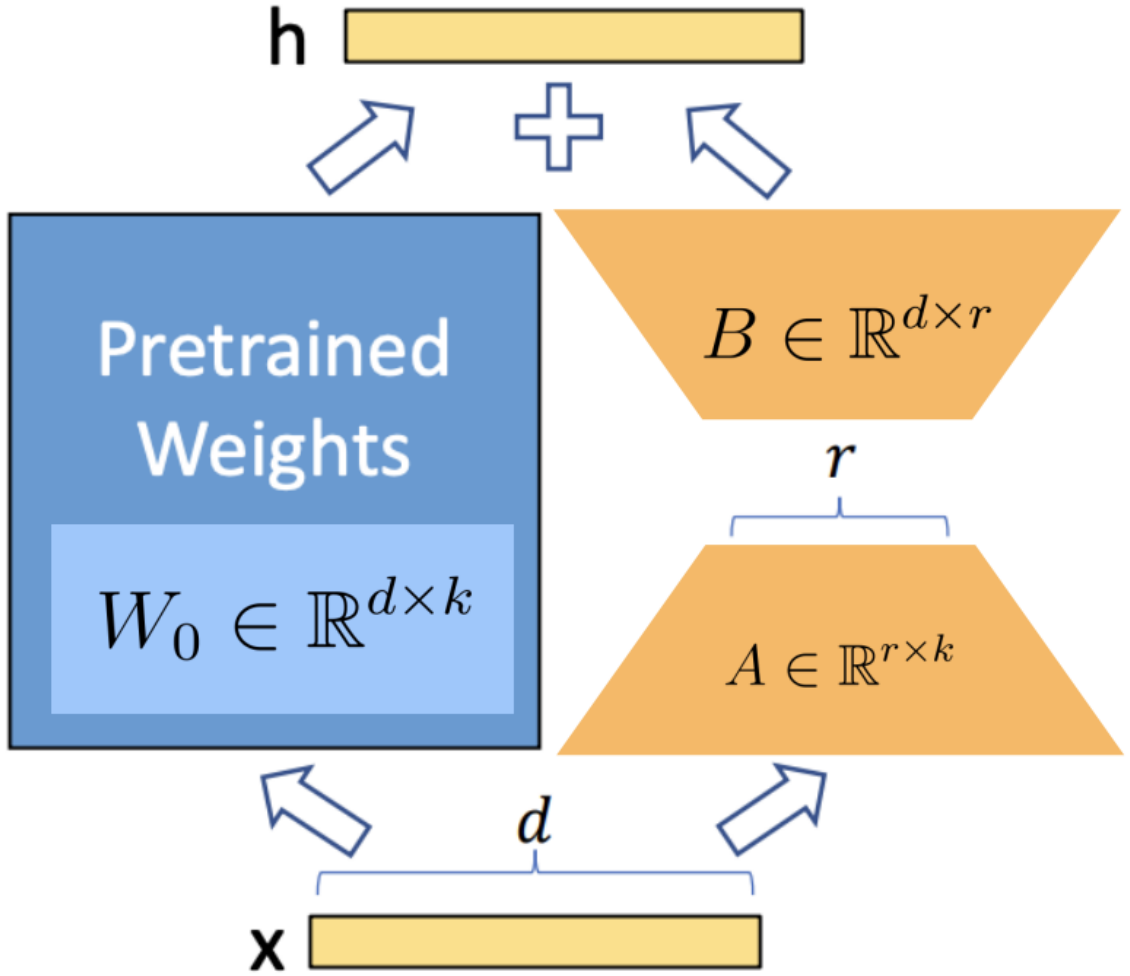
LoRA – low dimensional representation

Idea:

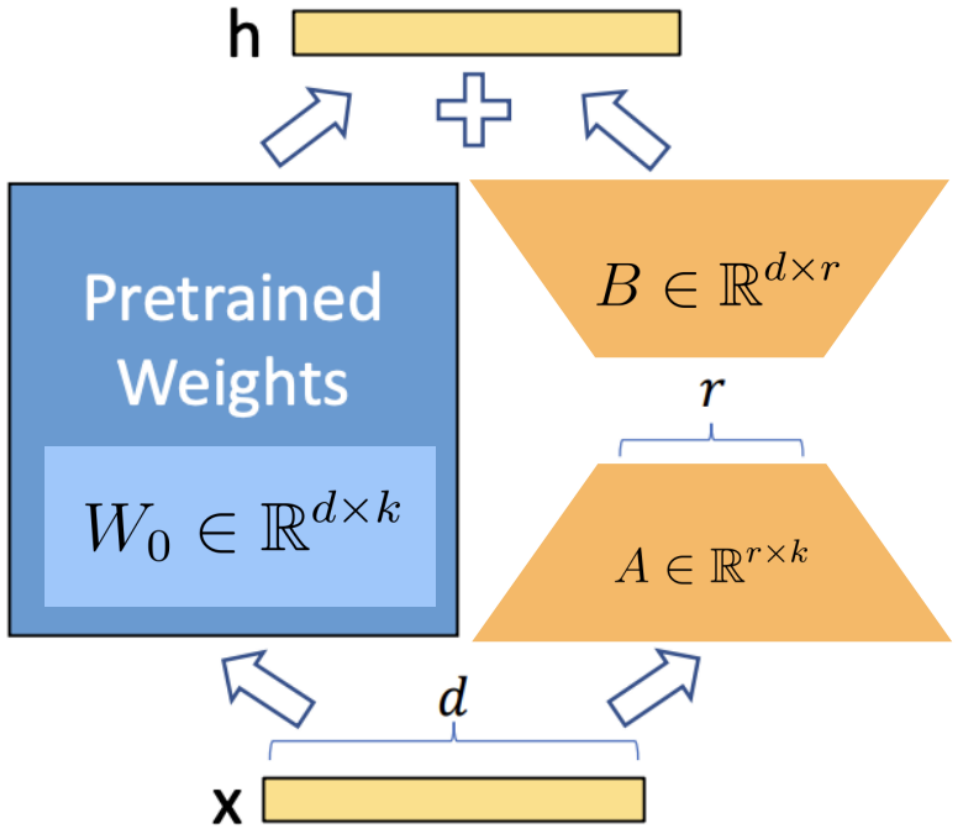
- LoRa updates a weight matrix $W \in R^{dxk}$ by learning a separate matrix ΔW which represents the **weight difference** gained from fine-tuning.
- Create two **lower dimension** weight matrices $A \in R^{dxr}$ and $B \in R^{rxk}$ to represent this difference. $\Delta W = AB \in R^{dxk}, r \ll d, k$
- By creating lower dimension weight matrices A and B , we have $d * k - d * r - r * k$ less parameters to learn. r is a small number, like 4 or 8

LoRA

- Only A, B are trainable params (only 0.1%-0.5% params)
- **No additional inference latency:** when switching to a different task, recover W by subtracting A, B and adding $A'B'$
- LoRA is often applied to weight matrices in the **self-attention** module, like K, Q, V matrices



LoRA




```python
def lora_linear(x, W):
    scale = 1 / r  # r is rank
    h = x @ W
    h += x @ W_a @ W_b  # W_a,W_b determined based on W
    return scale * h

def self_attention_lora(x):
    """ Pseudo code from Lialin et al. [2]."""

    k = lora_linear(x, W_k)
    q = x @ W_q
    v = lora_linear(x, W_v)
    return softmax(q @ k.T) @ v
```

LoRA Performance

LoRA outperforms several baselines with comparable or fewer parameters

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	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.85}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{71.8}_{\pm.1}$	$\mathbf{2.53}_{\pm.02}$
GPT-2 L (FT)*	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L (Adapter^{L})	0.88M	$69.1_{\pm.1}$	$8.68_{\pm.03}$	$46.3_{\pm.0}$	$71.4_{\pm.2}$	$\mathbf{2.49}_{\pm.0}$
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	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.89}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{72.0}_{\pm.2}$	$2.47_{\pm.02}$

LoRA – apply LoRA to which weight matrix

Adapting both Q and V matrix in self-attention gives the best performance

	# 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**

LoRA – choose the value of rank r

LoRA already performs competitively with a very small r

	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

LoRA Summary

- Instead of directly updating weight matrix W in pre-trained LM,
- we learn the **weight difference** ΔW between pre-trained LM and fine-tuned LM
- via two **lower-dimension** matrix A, B with fewer parameters


Thank you!
UF
Herbert Wertheim
College of Engineering
UNIVERSITY of FLORIDA
LEADING THE CHARGE, CHARGING AHEAD