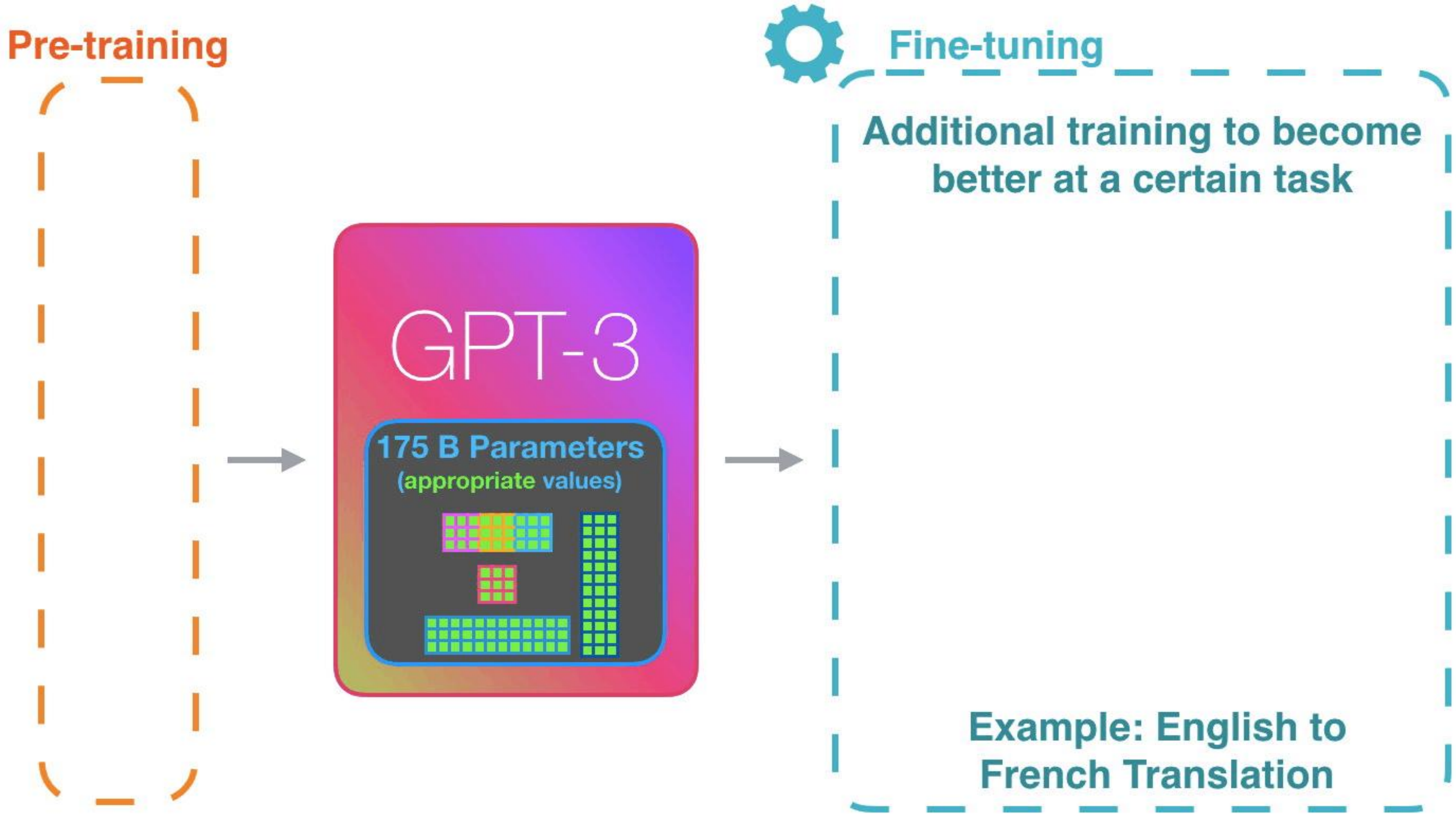
Parameter-Efficient Fine-Tuning (PEFT)

What is Fine-tuning

Fine-tuning is the process of taking a model that is already trained on some task and then tweaking it to perform a similar task. It is often used when a new dataset or task requires the model to have some modifications, or when the model is not performing well on a specific task.

For example, a model trained to generate stories can be fine-tuned to generate poems. This is possible because the model has already learned how to generate casual language and write stories, this skill can also be used to generate poems if the model is tweaked properly.

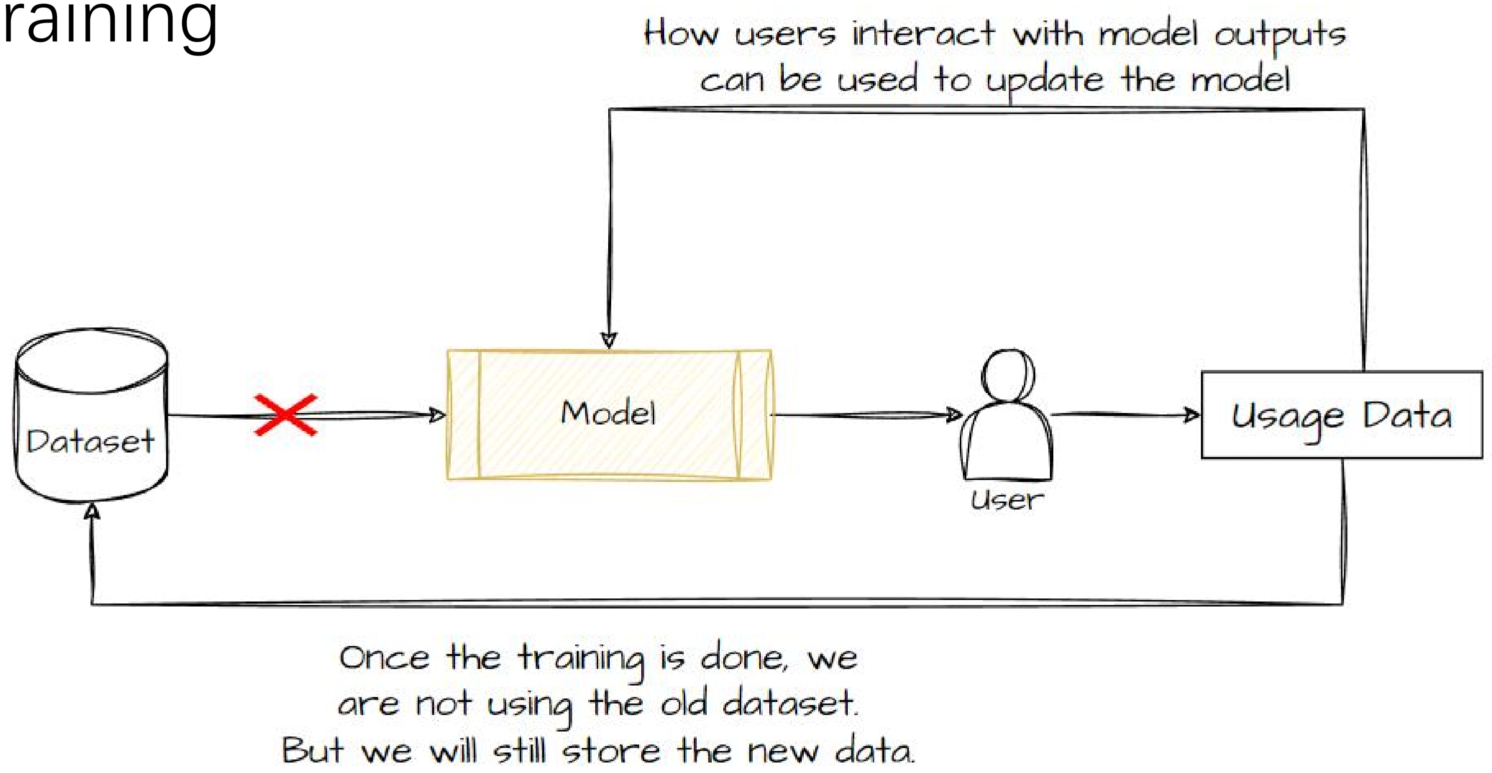
How does Fine-tuning work?


Pre-training
GPT-3
175 B Parameters
(appropriate values)
Fine-tuning
Additional training to become better at a certain task
Example: English to French Translation

taking the weights of the original model and adjusting them to fit a new task.

Why use Fine-Tuning?

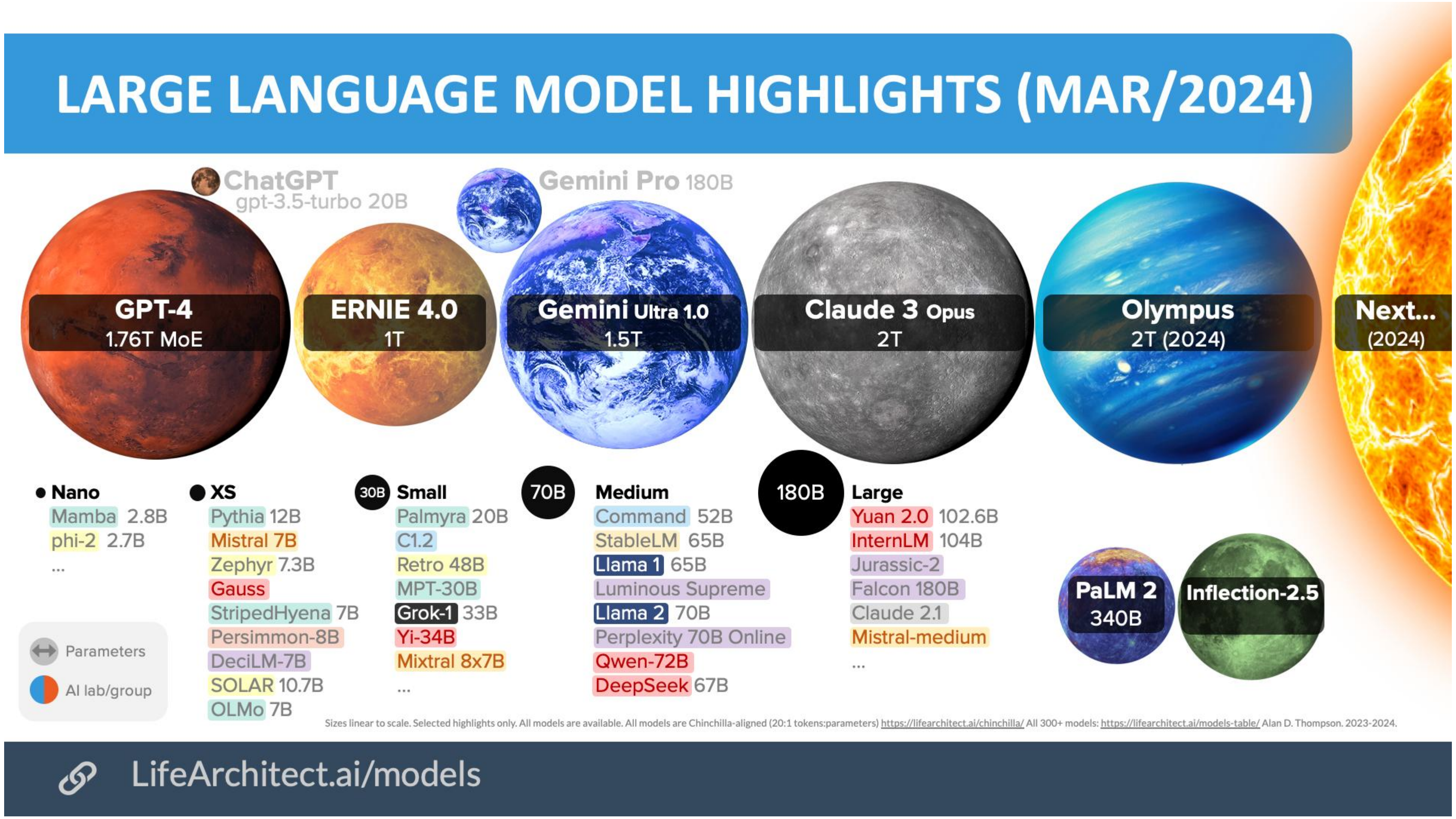
- Larger models generalize to downstream tasks well
- Cheaper than training a whole model
- Good for online training


How users interact with model outputs can be used to update the model
Dataset
Model
User
Usage Data
Once the training is done, we are not using the old dataset. But we will still store the new data.

Use a LLM when you get some data.

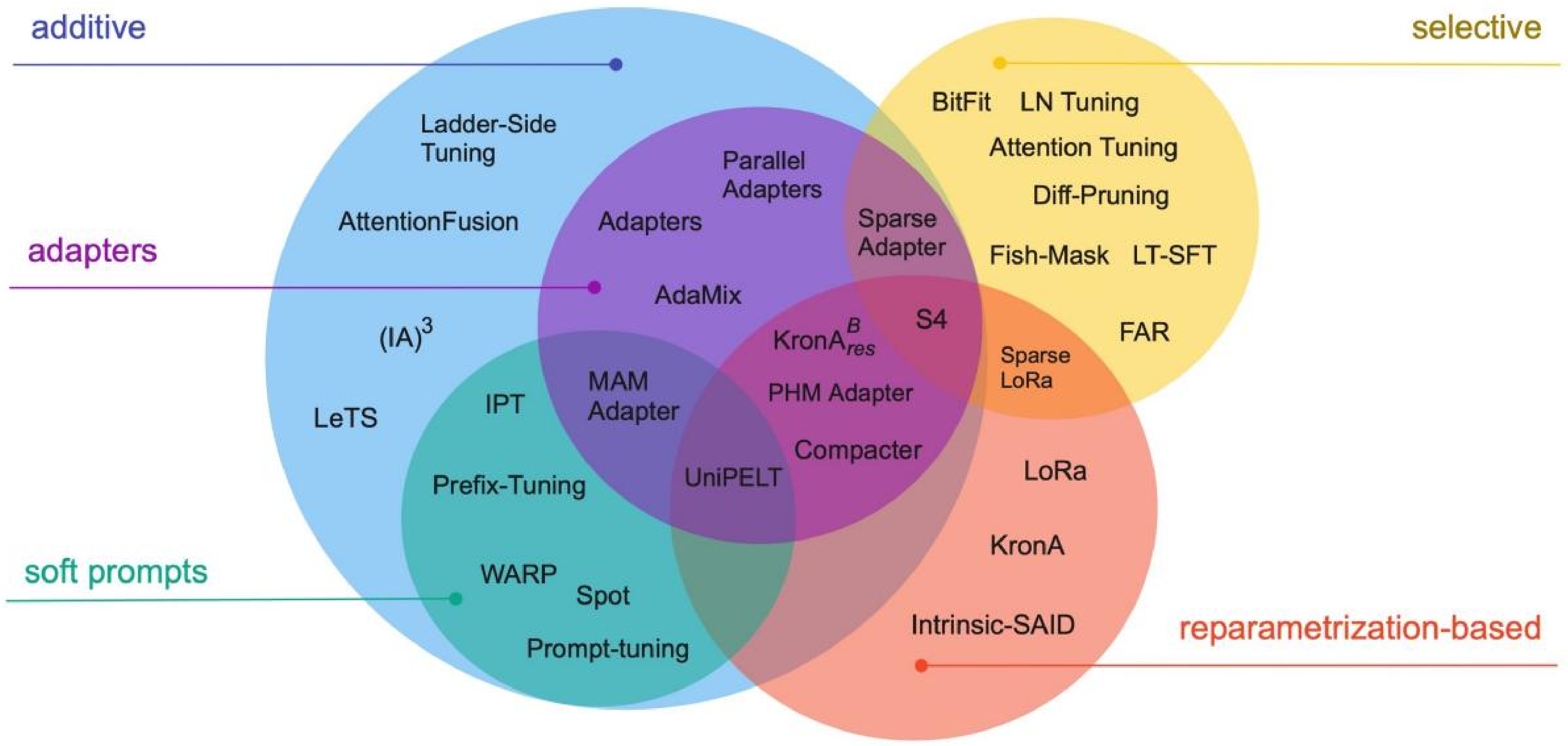
- Prompt engineering
  - Doesn’t always work
  - Tedious to find a good prompt
- Finetune the full LM on your data
  - Expensive
  - Overfitting / catastrophic forgetting on small datasets
  - Need to store one full set of model weights per task
- Parameter-efficient Fine-tuning

What is PEFT?



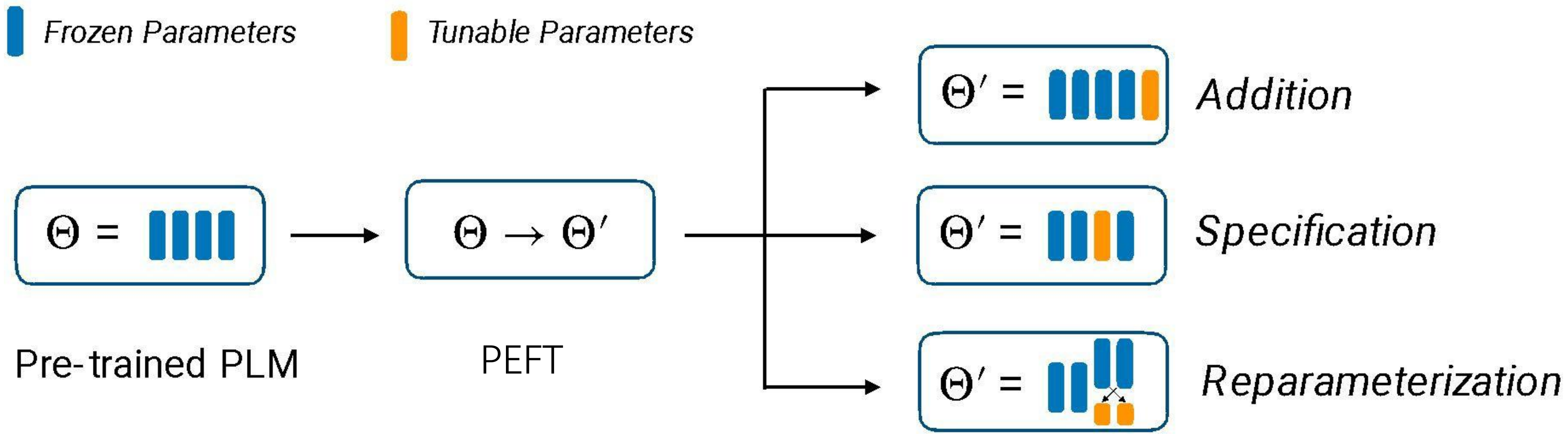
PEFT, Parameter Efficient Fine-Tuning, is a set of techniques or methods to fine-tune a large model in the most compute and time-efficient way possible, without losing any performance which you might see from full fine-tuning.

PEFT Methods



PEFT Methods

- **Addition:** What if we introduce additional trainable parameters to the neural network and just train those?
- **Specification:** What if we pick a small subset of the parameters of the neural network and just tune those?
- **Reparameterization:** What if we re-parameterize the model into something that is more efficient to train?



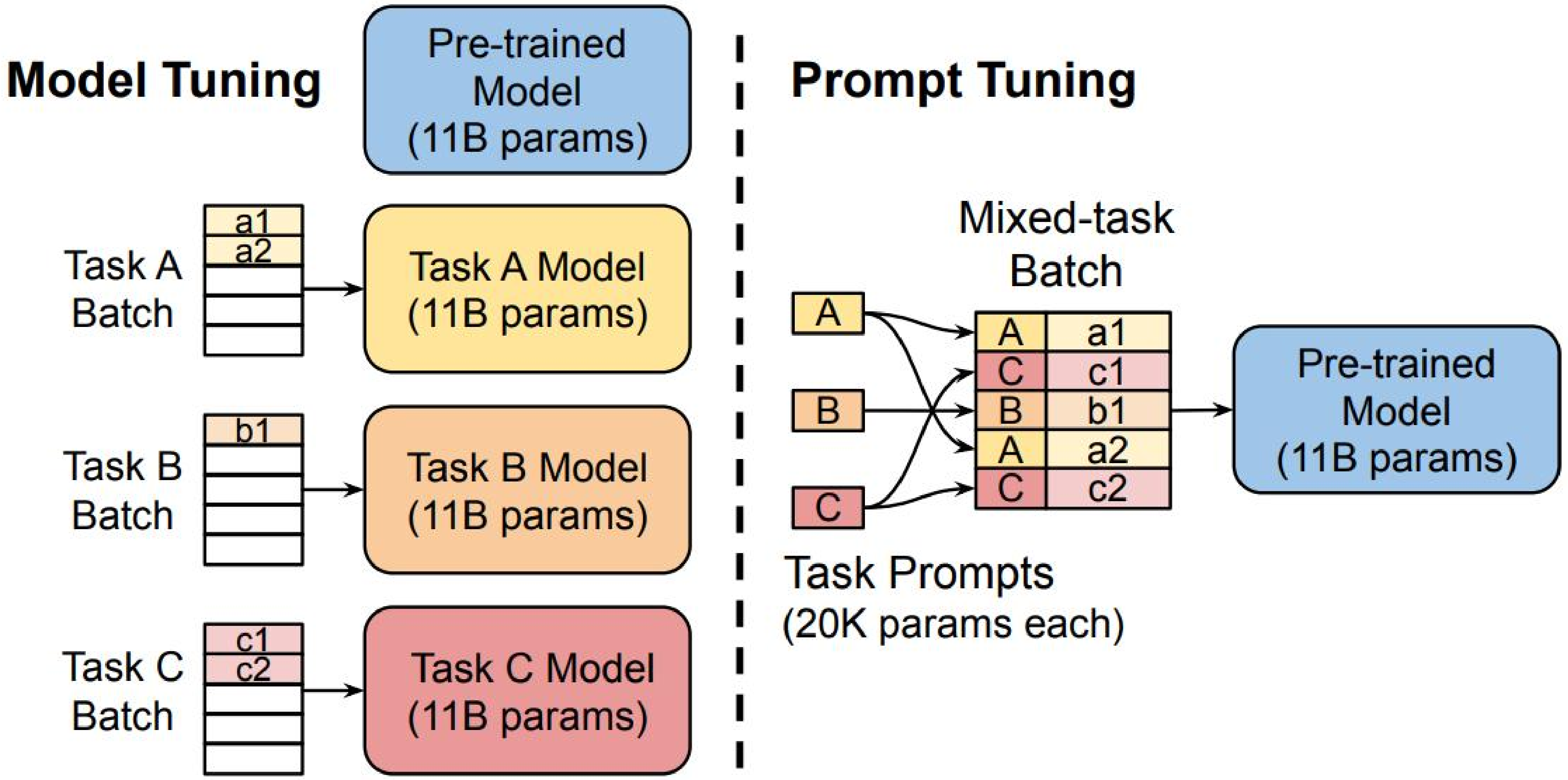
What if we introduce additional trainable parameters to the neural network and just train those?

Methods: prompt tuning, prefix tuning, adapters, and compacters

Prompt Tuning

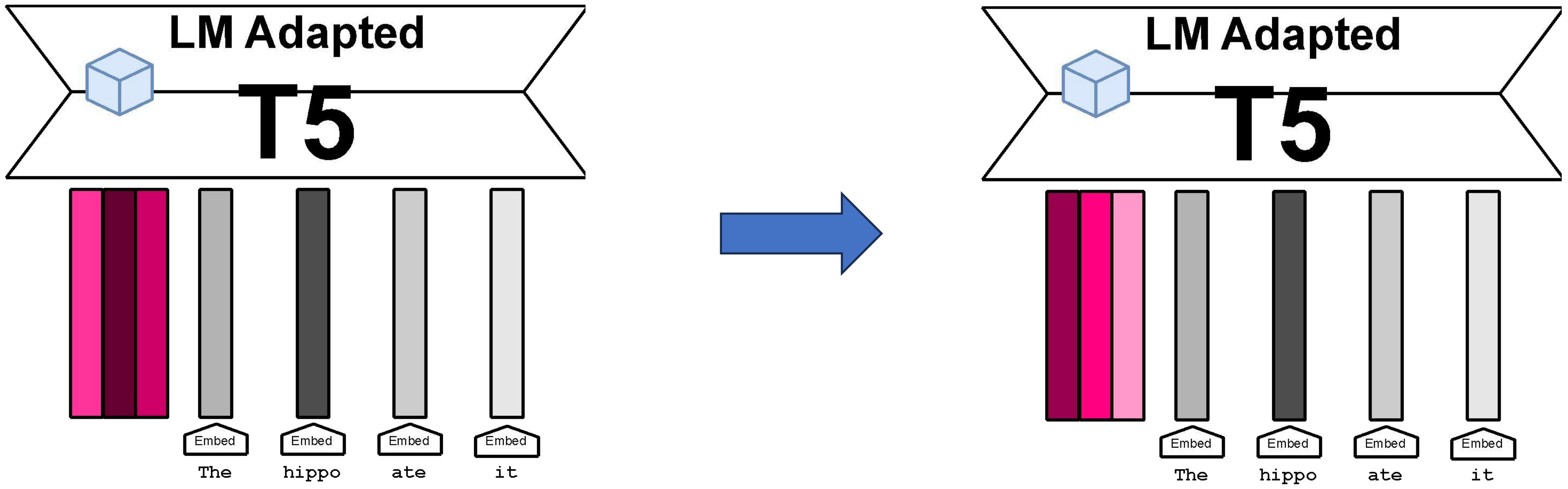
1. Finetune T5 to act a bit more like a traditional language model.
   - This only needs to be done once, and empirically makes prompt tuning working better.
   - This is probably because the span-corruption objective T5 was originally trained with isn't amenable to prompting.
2. Freeze the weights of T5. Set the first k input embeddings to be learnable.
   - k is a hyperparameter up to the choice of the implementer.
3. Initialize the k learnable embeddings. Some options include:
   - Random initialization
   - Initialize to values drawn from the vocabulary embedding matrix
4. Train on your task specific data,

Prompt Tuning

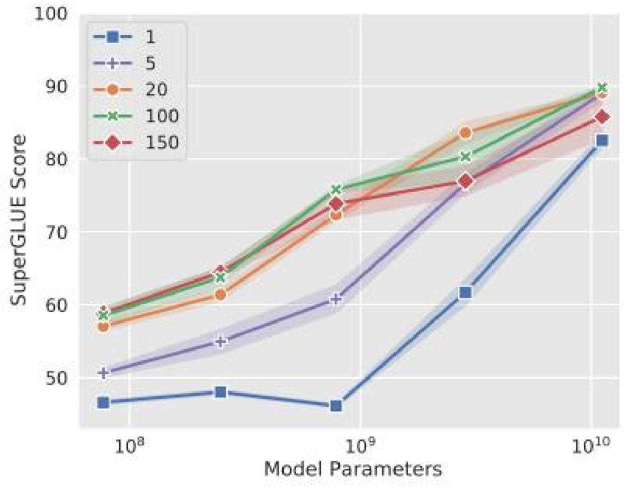


Model Tuning
Pre-trained Model (11B params)
Task A Batch
a1
a2
Task A Model (11B params)
Task B Batch
b1
Task B Model (11B params)
Task C Batch
c1
c2
Task C Model (11B params)
Prompt Tuning
Mixed-task Batch
A
B
C
A a1
C c1
B b1
A a2
C c2
Pre-trained Model (11B params)
Task Prompts (20K params each)

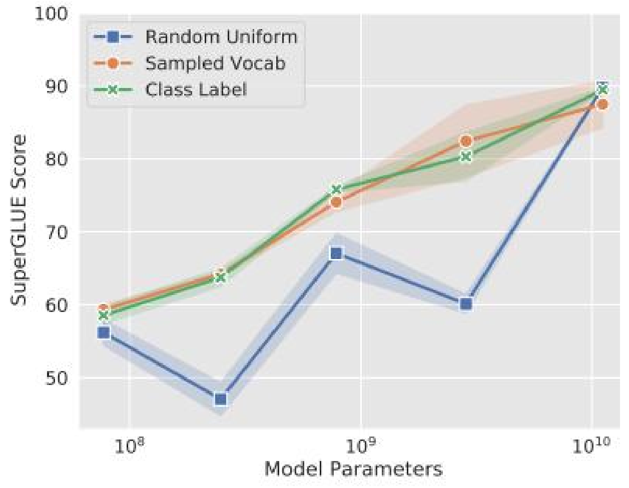
Prompt Tuning



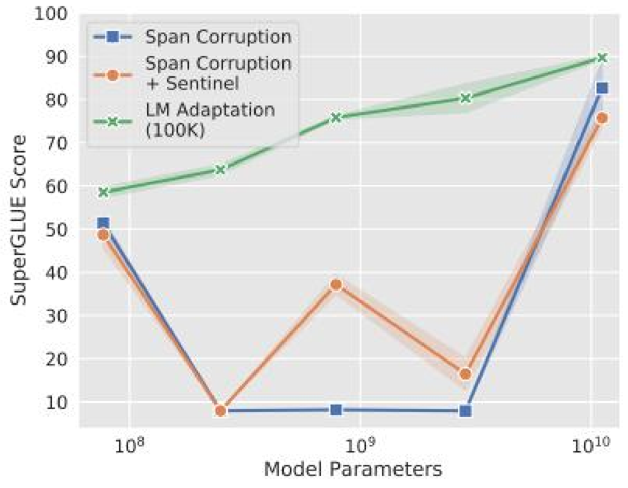
Efficiency of Prompt Tuning



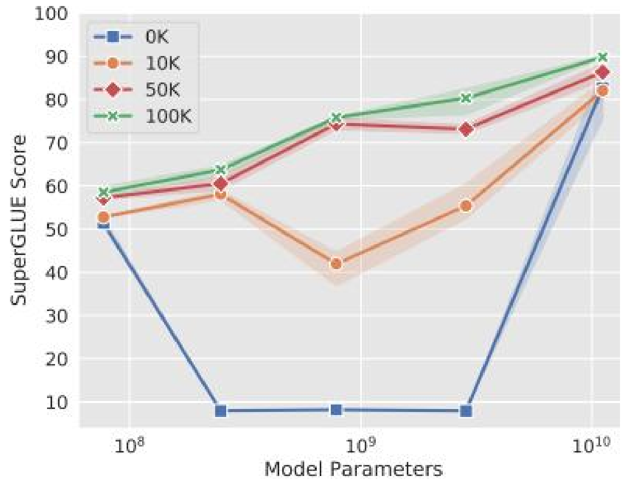
(a) Prompt length



(b) Prompt initialization

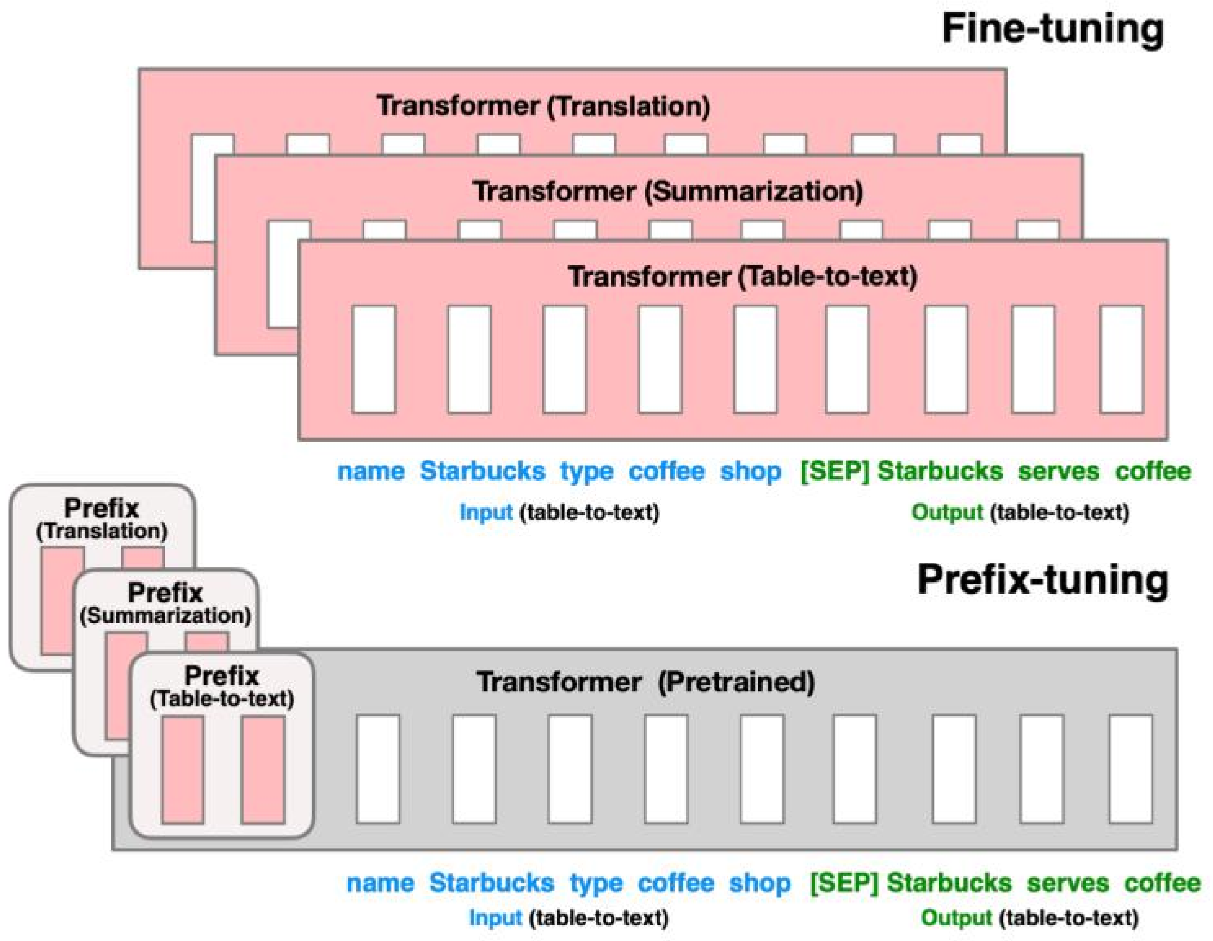


(c) Pre-training method

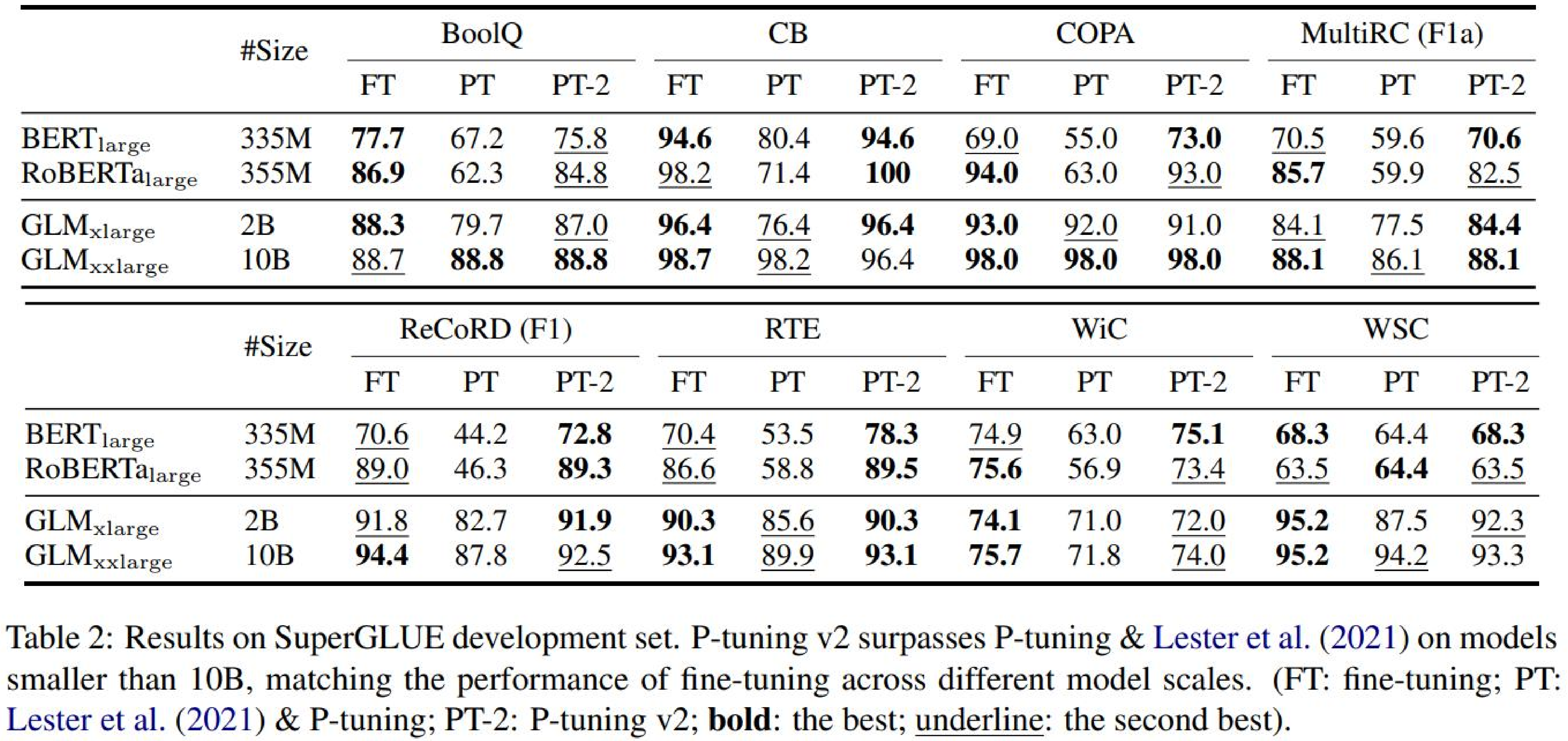


(d) LM adaptation steps

Prefix Tuning


Fine-tuning
Transformer (Translation)
Transformer (Summarization)
Transformer (Table-to-text)
name Starbucks type coffee shop [SEP] Starbucks serves coffee
Input (table-to-text)
Output (table-to-text)
Prefix-tuning
Prefix (Translation)
Prefix (Summarization)
Prefix (Table-to-text)
Transformer (Pretrained)
name Starbucks type coffee shop [SEP] Starbucks serves coffee
Input (table-to-text)
Output (table-to-text)

Efficiency of Prefix Tuning



	#Size	BoolQ			CB			COPA			MultiRC (F1a)		
		FT	PT	PT-2	FT	PT	PT-2	FT	PT	PT-2	FT	PT	PT-2
$\text{BERT}_{\text{large}}$	335M	**77.7**	67.2	75.8	**94.6**	80.4	**94.6**	69.0	55.0	**73.0**	70.5	59.6	**70.6**
$\text{RoBERTa}_{\text{large}}$	355M	**86.9**	62.3	84.8	98.2	71.4	**100**	**94.0**	63.0	93.0	**85.7**	59.9	82.5
$\text{GLM}_{\text{xlarge}}$	2B	**88.3**	79.7	87.0	**96.4**	76.4	**96.4**	**93.0**	92.0	91.0	84.1	77.5	**84.4**
$\text{GLM}_{\text{xxlarge}}$	10B	88.7	**88.8**	**88.8**	**98.7**	98.2	96.4	**98.0**	**98.0**	**98.0**	**88.1**	86.1	**88.1**

	#Size	ReCoRD (F1)			RTE			WiC			WSC		
		FT	PT	PT-2	FT	PT	PT-2	FT	PT	PT-2	FT	PT	PT-2
$\text{BERT}_{\text{large}}$	335M	70.6	44.2	**72.8**	70.4	53.5	**78.3**	74.9	63.0	**75.1**	**68.3**	64.4	**68.3**
$\text{RoBERTa}_{\text{large}}$	355M	89.0	46.3	**89.3**	86.6	58.8	**89.5**	**75.6**	56.9	73.4	63.5	**64.4**	63.5
$\text{GLM}_{\text{xlarge}}$	2B	91.8	82.7	**91.9**	**90.3**	85.6	**90.3**	**74.1**	71.0	72.0	**95.2**	87.5	92.3
$\text{GLM}_{\text{xxlarge}}$	10B	**94.4**	87.8	92.5	**93.1**	89.9	**93.1**	**75.7**	71.8	74.0	**95.2**	94.2	93.3

Table 2: Results on SuperGLUE development set. P-tuning v2 surpasses P-tuning & Lester et al. (2021) on models smaller than 10B, matching the performance of fine-tuning across different model scales. (FT: fine-tuning; PT: Lester et al. (2021) & P-tuning; PT-2: P-tuning v2; **bold**: the best; underline: the second best).

P-Tuning

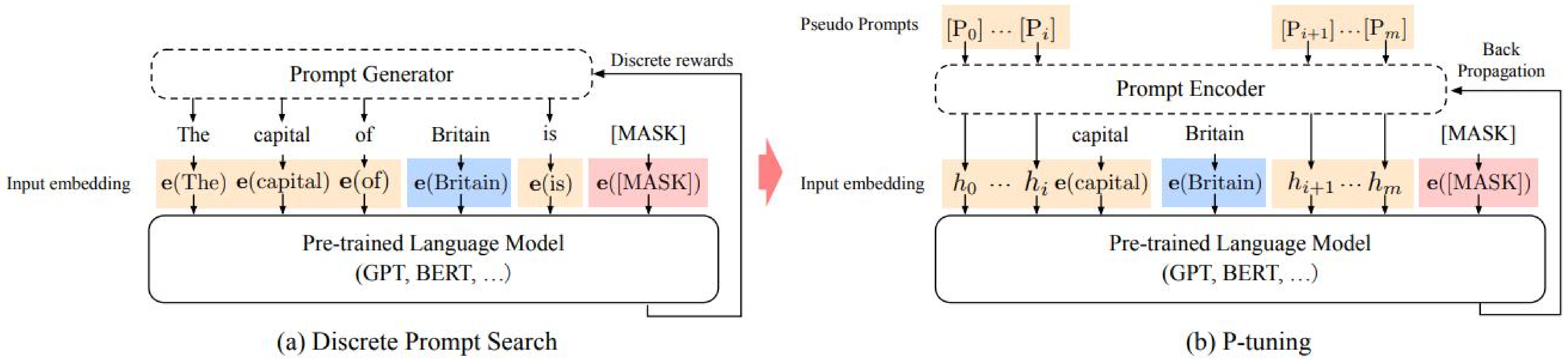
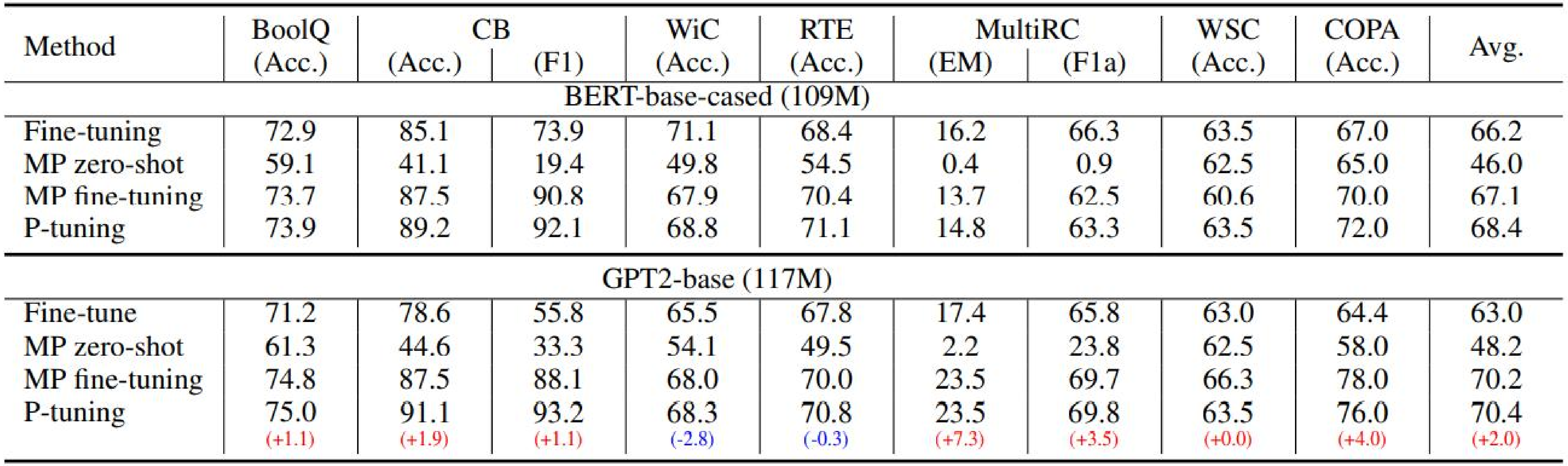


Figure 2. An example of prompt search for "The capital of Britain is [MASK]". Given the context (blue zone, "Britain") and target (red zone, "[MASK]"), the orange zone refer to the prompt tokens. In (a), the prompt generator only receives discrete rewards; on the contrary, in (b) the pseudo prompts and prompt encoder can be optimized in a differentiable way. Sometimes, adding few task-related anchor tokens (such as "capital" in (b)) will bring further improvement.

Efficiency of P-Tuning



Method	BoolQ (Acc.)	CB (Acc.)	CB (F1)	WiC (Acc.)	RTE (Acc.)	MultiRC (EM)	MultiRC (F1a)	WSC (Acc.)	COPA (Acc.)	Avg.
BERT-base-cased (109M)										
Fine-tuning	72.9	85.1	73.9	71.1	68.4	16.2	66.3	63.5	67.0	66.2
MP zero-shot	59.1	41.1	19.4	49.8	54.5	0.4	0.9	62.5	65.0	46.0
MP fine-tuning	73.7	87.5	90.8	67.9	70.4	13.7	62.5	60.6	70.0	67.1
P-tuning	73.9	89.2	92.1	68.8	71.1	14.8	63.3	63.5	72.0	68.4
GPT2-base (117M)										
Fine-tune	71.2	78.6	55.8	65.5	67.8	17.4	65.8	63.0	64.4	63.0
MP zero-shot	61.3	44.6	33.3	54.1	49.5	2.2	23.8	62.5	58.0	48.2
MP fine-tuning	74.8	87.5	88.1	68.0	70.0	23.5	69.7	66.3	78.0	70.2
P-tuning	75.0 (+1.1)	91.1 (+1.9)	93.2 (+1.1)	68.3 (-2.8)	70.8 (-0.3)	23.5 (+7.3)	69.8 (+3.5)	63.5 (+0.0)	76.0 (+4.0)	70.4 (+2.0)

Table 3. Fully-supervised learning on SuperGLUE dev with base-scale models. MP refers to manual prompt. For a fair comparison, MP zero-shot and MP fine-tuning report results of a single pattern, while anchors for P-tuning are selected from the same prompt. Subscript in red represents advantages of GPT with P-tuning over the best results of BERT.

Advantages of Prefix/Prompt Tuning

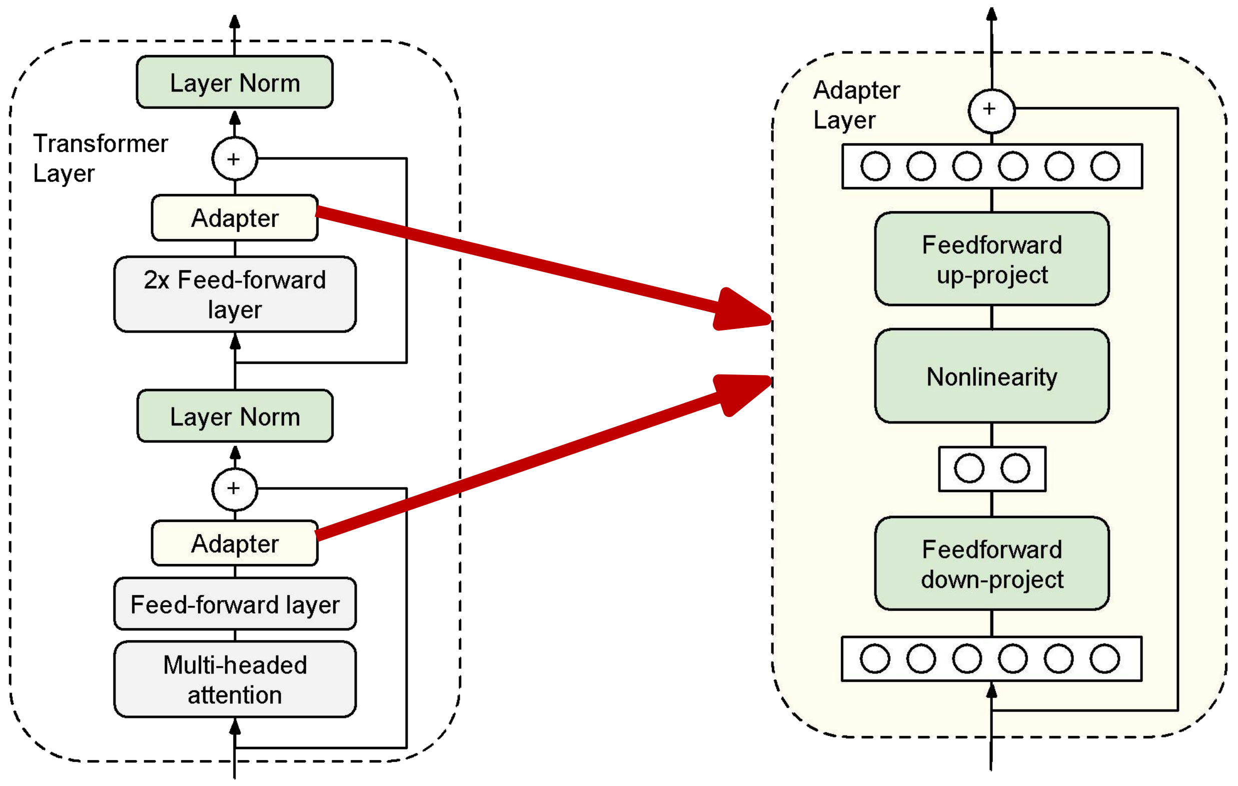
- The learned embeddings tend to be relatively small, just a few megabytes or less.
  - It is cheap to keep around one set of embeddings per task.
- The pre-trained LLM can be loaded into memory (such as on a server), and at inference time, the appropriate task-specific embeddings can be passed in.
  - Example use case: User customization

Pitfalls of Prefix and Prompt Tuning

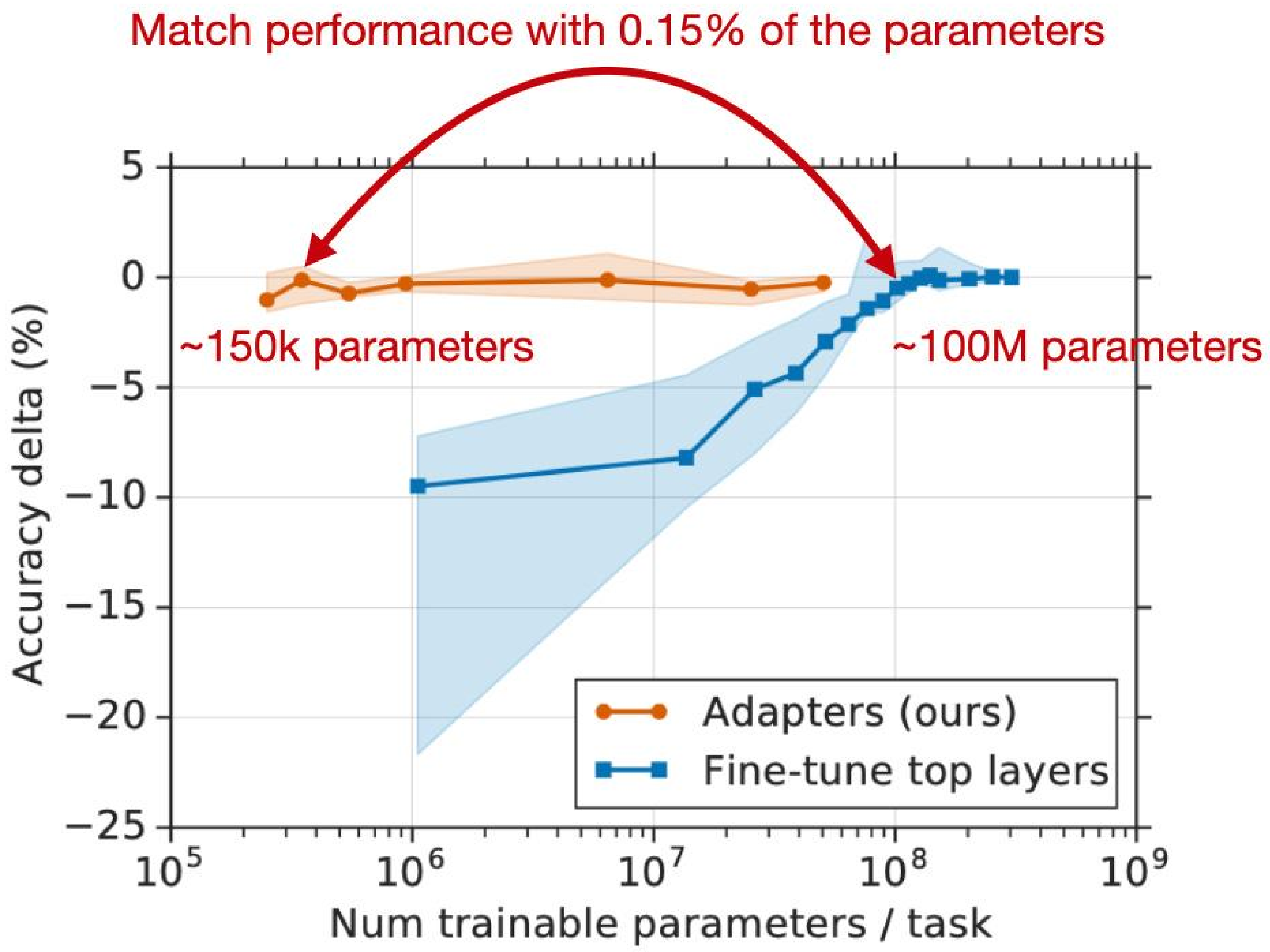
- In practice, these methods tend to converge significantly slower than full parameter fine-tuning.
- Unclear what the best prefix length is for any particular task.
  - Every sequence position you “spend” on the prefix is one less you have for your actual task.
- Learned embeddings are not very interpretable.

Adapters

- Adapters are **new modules** are added between layers of a pre-trained network.
- The original model weights are **fixed**; just the adapter modules are **tuned**.
- The adapters are initialized such that the output of the adapter-inserted module resembles the original model.



Adapters


Match performance with 0.15% of the parameters
~150k parameters
~100M parameters
5
0
−5
−10
−15
−20
−25
Accuracy delta (%)
10^5
10^6
10^7
10^8
10^9
Num trainable parameters / task
Adapters (ours)
Fine-tune top layers

Advantages of Adapter-Based Methods

- Have been shown to be quite effective in multi-task settings.
  - There are methods for training task-specific adapters and then combining the to leverage the cross-task knowledge.
- Tend to be faster to tune than full model finetuning.
- Possibly more robust to adversarial perturbations of the tuning data full model finetuning.

Pitfalls or Adapter Methods

- Adding in new layers means making inference slower.
- It also makes the model bigger (possibly harder to fit on available Accelerators, like GPUs, NPUs).
- Adapter layers need to be processed sequentially at inference time, which can break model parallelism.

What if we pick a small subset of the parameters of the neural network and just tune those?

Methods: layer freezing, BitFit, diff pruning

Layer Freezing

- Research has shown that earlier layers of the Transformer tend to capture linguistic phenomena and basic language understand; later layers are where the task-specific learning happens.
- This means we should be able to learn new tasks by freezing the earlier layers and just tuning the later ones.

BitFit: Bias-terms Fine-tuning

- Only tune the bias terms and final classification layer (if doing classification)
- Recall the equations for multi-head attention

$$\mathbf{Q}^{m,\ell}(\mathbf{x}) = \mathbf{W}_q^{m,\ell}\mathbf{x} + \mathbf{b}_q^{m,\ell}$$
$$\mathbf{K}^{m,\ell}(\mathbf{x}) = \mathbf{W}_k^{m,\ell}\mathbf{x} + \mathbf{b}_k^{m,\ell}$$
$$\mathbf{V}^{m,\ell}(\mathbf{x}) = \mathbf{W}_v^{m,\ell}\mathbf{x} + \mathbf{b}_v^{m,\ell}$$

  - ℓ is the layer index
  - m is the attention head index
  - Only the bias terms (shown in red) are updated.

Intuition for DiffPruning

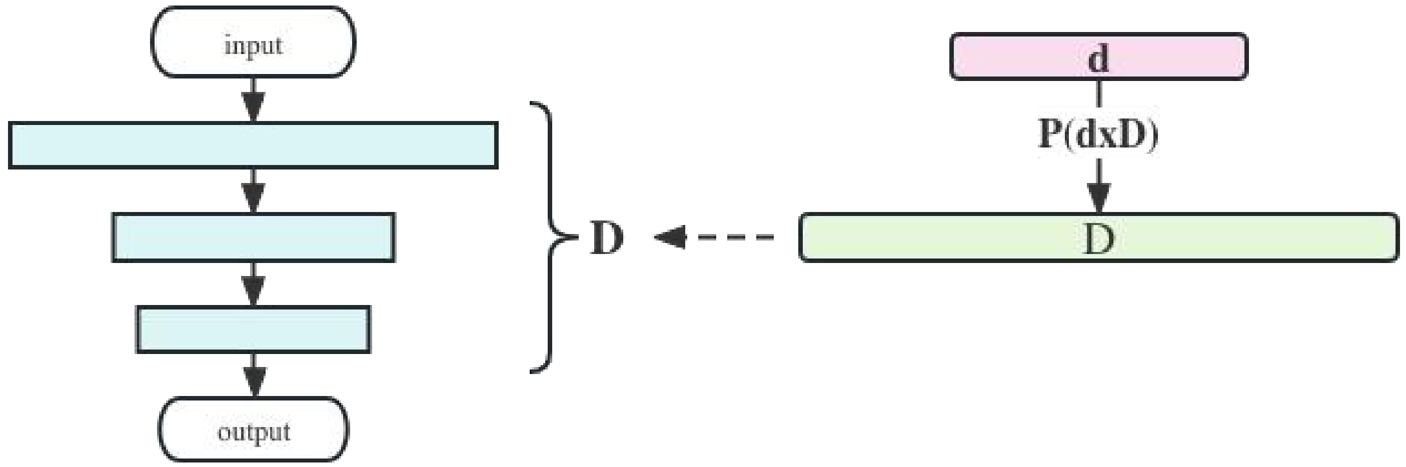
- In prior methods we discussed, the choice of what parameters to freeze and what parameters to tune was done manually.
- Why not learn this instead?
- Main idea:
  - For each parameter, finetune a learnable “delta” which gets added to the original parameter value.
  - Use an L_0-norm penalty to encourage sparsity in the deltas.

What if we re-parameterize the model into something that is more efficient to train?

Methods: LoRa, $(IA)^3$

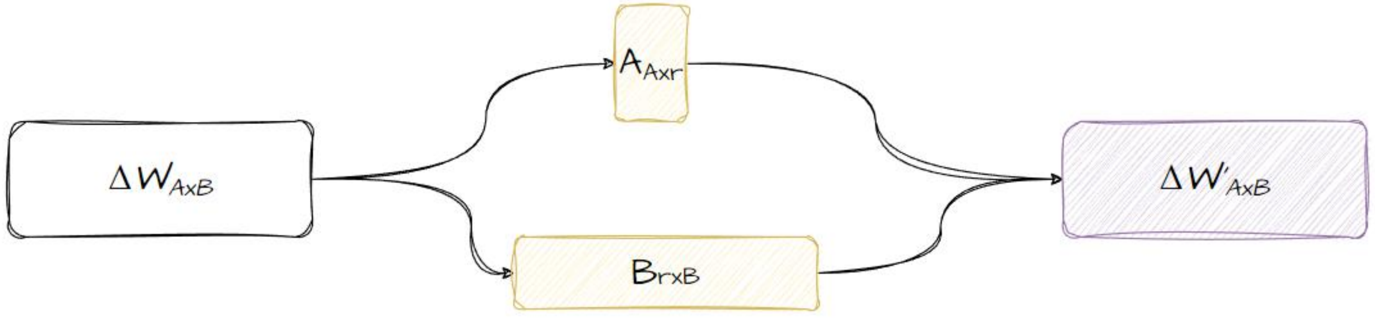
Intuition for Re-Parameterizing the Model

- Finetuning has a low intrinsic dimension, that is, the minimum number of parameters needed to be modified to reach satisfactory performance is not very large.
- This means, we can re-parameterize a subset of the original model parameters with low-dimensional proxy parameters, and just optimize the proxy.

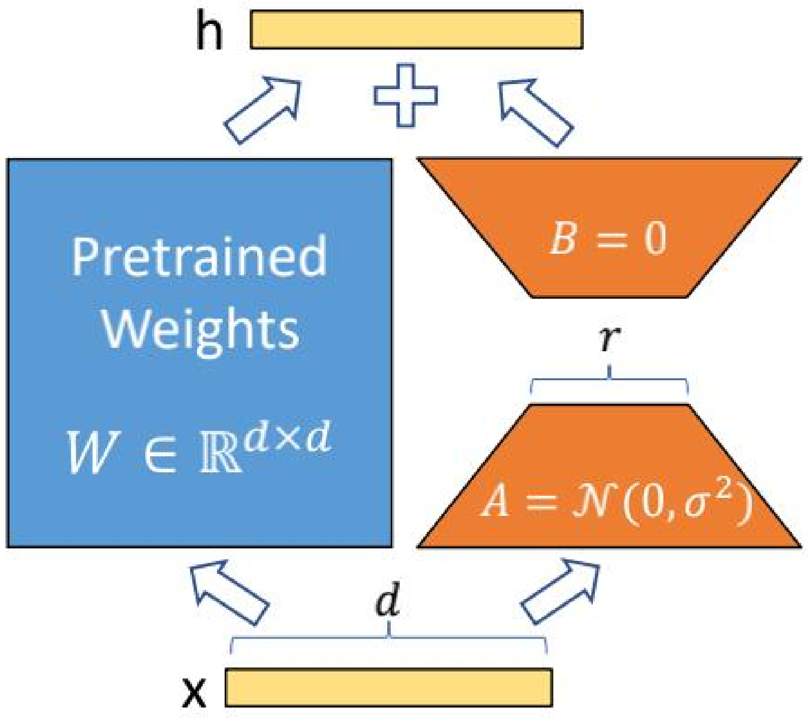


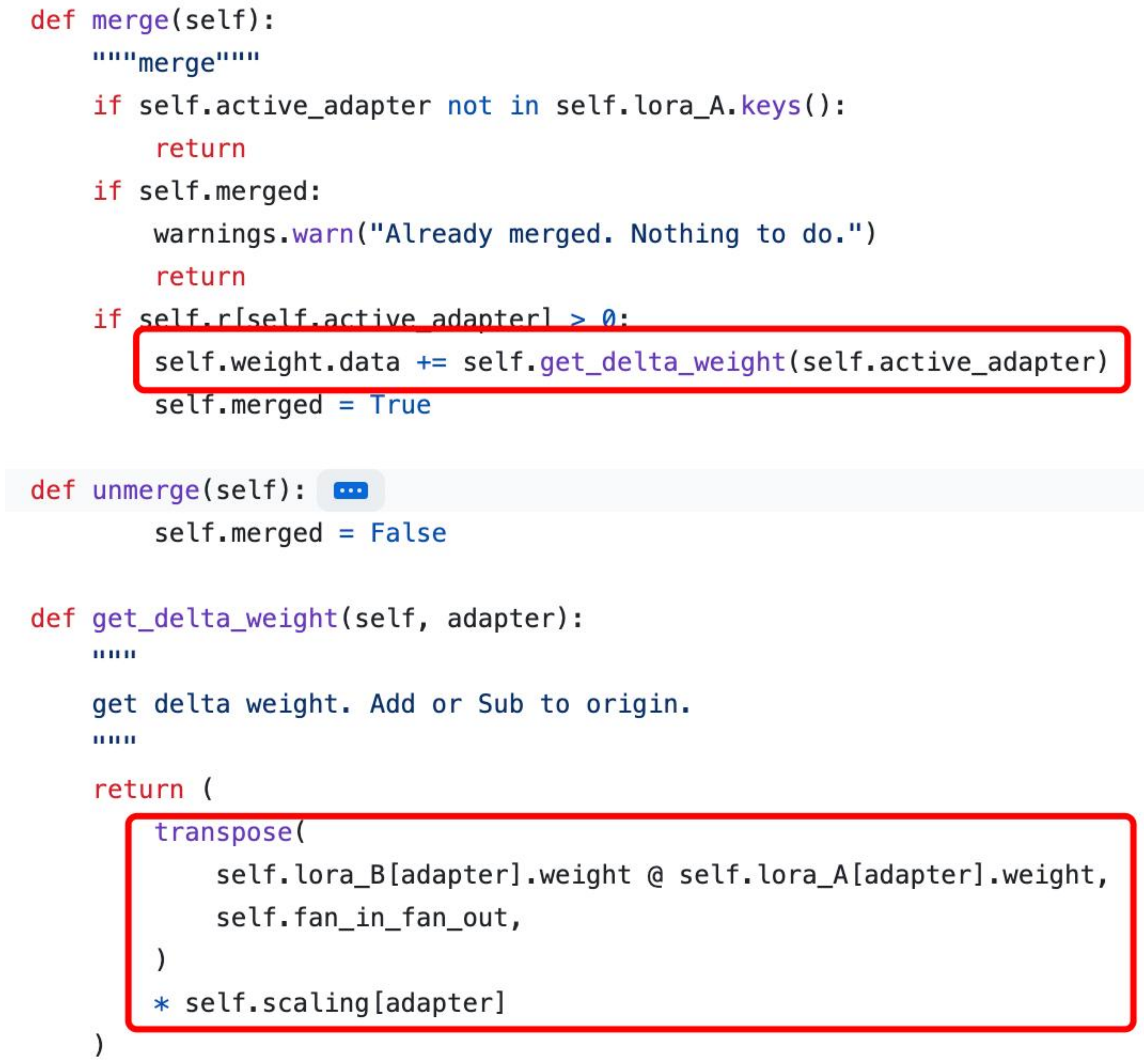
LoRA - Low-Rank Adaptation

- Suppose we have model parameters θ^D
  - D is the number of parameters.
- Instead of optimizing θ^D, we could instead optimize a smaller set of parameters θ^d where $d \ll D$.
- This is done through clever factorization:
  - $\theta^D = \theta_0^D + P(\theta^d)$ where $P: \mathbb{R}^d \rightarrow \mathbb{R}^D$
  - *P* is typically a linear projection: $\theta^D = \theta_0^D + \theta^d M$



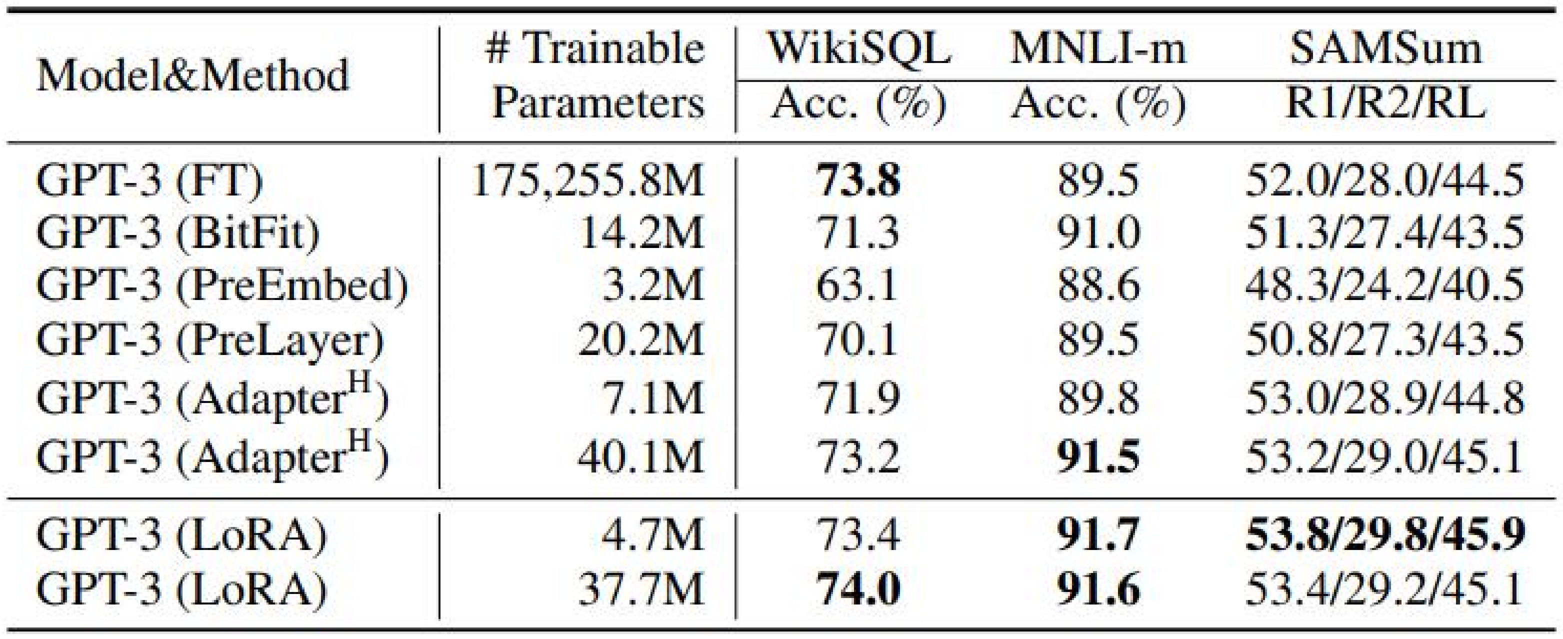
LoRA


h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x



```python
def merge(self):
    """merge"""
    if self.active_adapter not in self.lora_A.keys():
        return
    if self.merged:
        warnings.warn("Already merged. Nothing to do.")
        return
    if self.r[self.active_adapter] > 0:
        self.weight.data += self.get_delta_weight(self.active_adapter)
        self.merged = True

def unmerge(self): ...
        self.merged = False

def get_delta_weight(self, adapter):
    """

    get delta weight. Add or Sub to origin.
    """

    return (
        transpose(
            self.lora_B[adapter].weight @ self.lora_A[adapter].weight,
            self.fan_in_fan_out,
        )
        * self.scaling[adapter]
    )
```

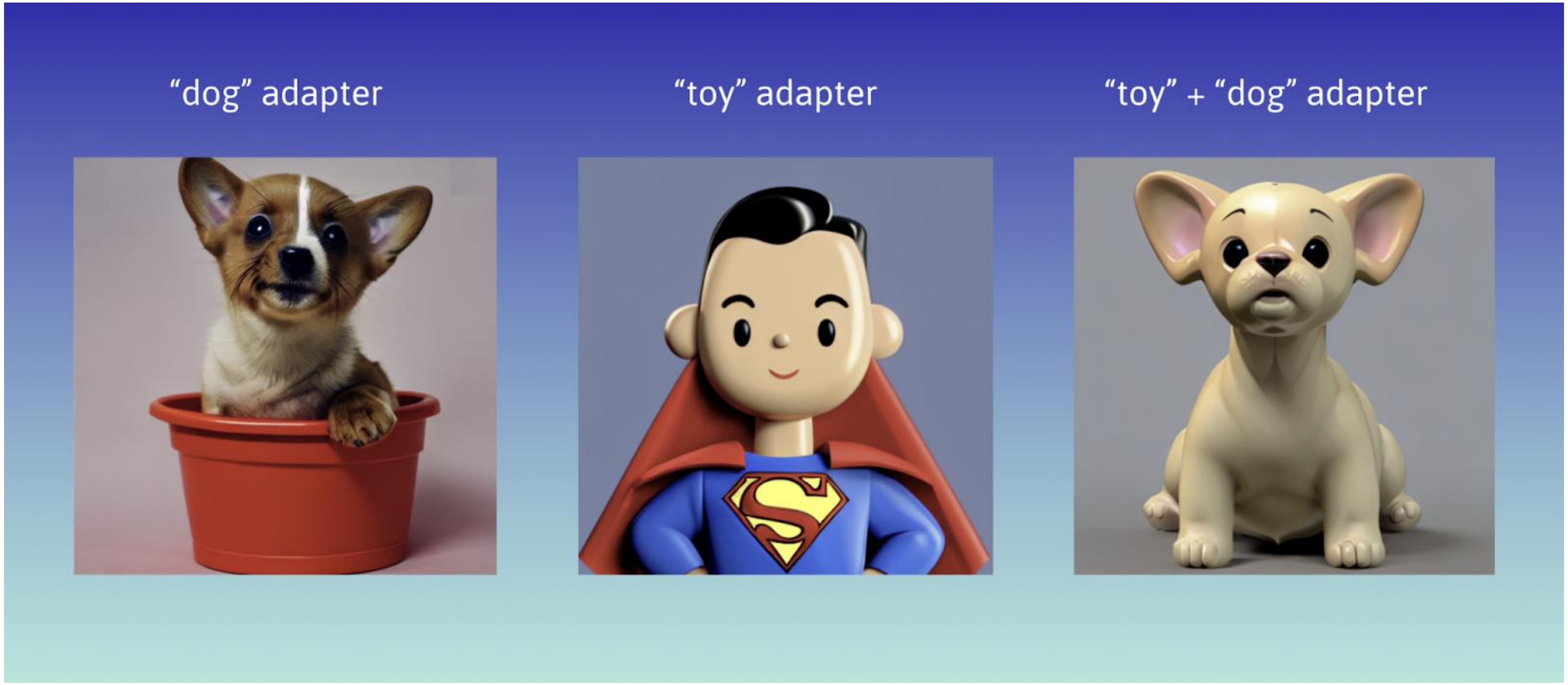
Efficiency of LoRA



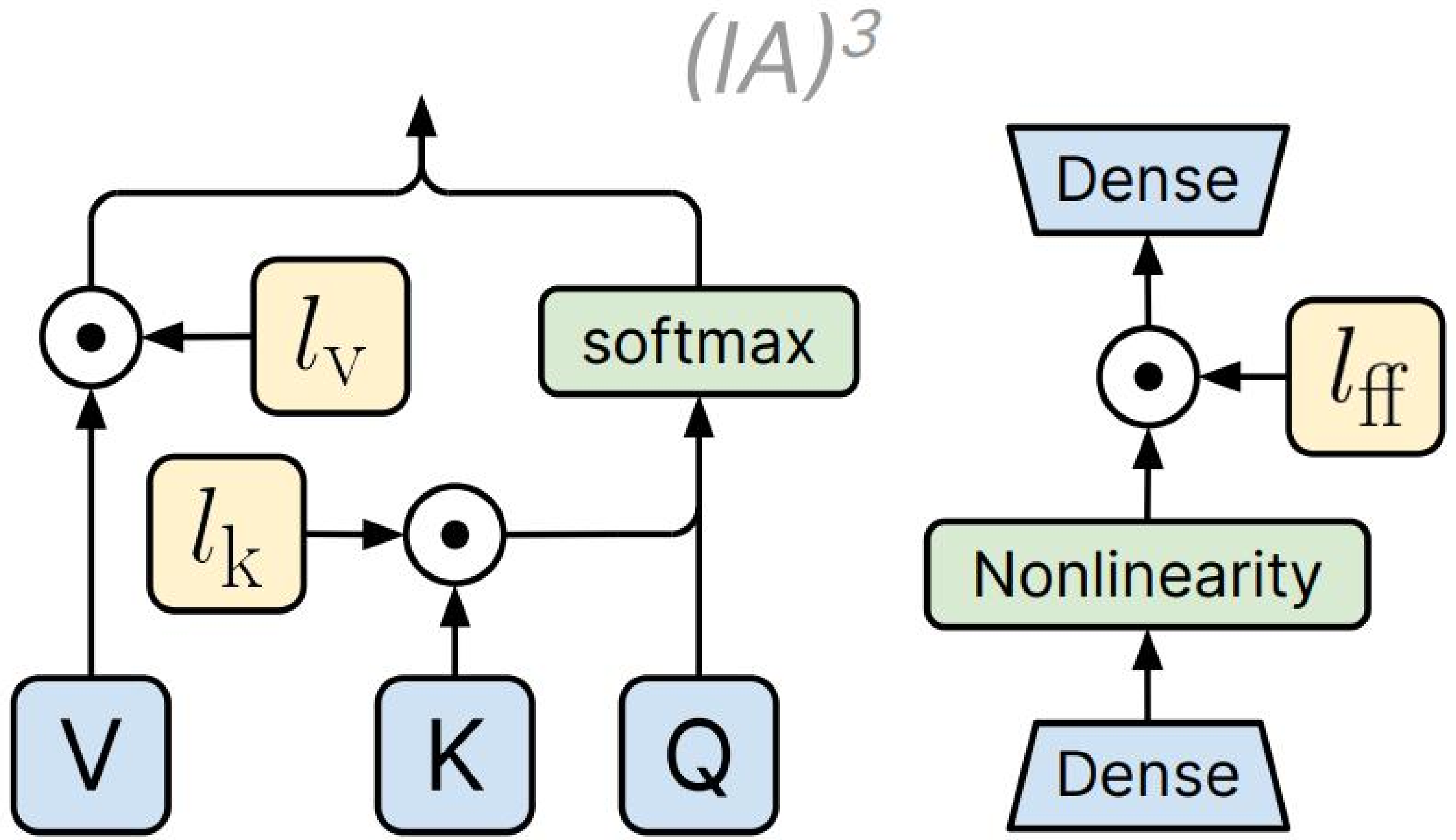
Model&Method	# Trainable Parameters	WikiSQL Acc. (%)	MNLI-m Acc. (%)	SAMSum R1/R2/RL
GPT-3 (FT)	175,255.8M	**73.8**	89.5	52.0/28.0/44.5
GPT-3 (BitFit)	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 (PreEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 (PreLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 (Adapter^H)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 (Adapter^H)	40.1M	73.2	**91.5**	53.2/29.0/45.1
GPT-3 (LoRA)	4.7M	73.4	**91.7**	**53.8/29.8/45.9**
GPT-3 (LoRA)	37.7M	**74.0**	**91.6**	53.4/29.2/45.1

Table 4: Performance of different adaptation methods on GPT-3 175B. We report the logical form validation accuracy on WikiSQL, validation accuracy on MultiNLI-matched, and Rouge-1/2/L on SAMSum. LoRA performs better than prior approaches, including full fine-tuning. The results on WikiSQL have a fluctuation around $\pm 0.5\%$, MNLI-m around $\pm 0.1\%$, and SAMSum around $\pm 0.2/\pm 0.2/\pm 0.1$ for the three metrics.

LoRA in Stable Diffusion



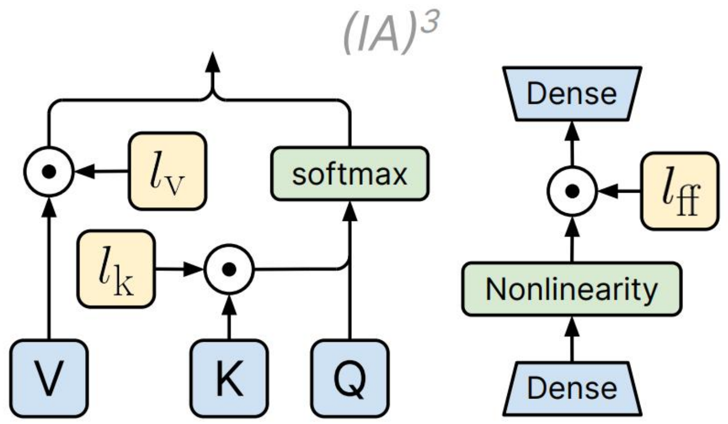
(IA)^3 - Infused Adapter by Inhibiting and Amplifying Inner Activations

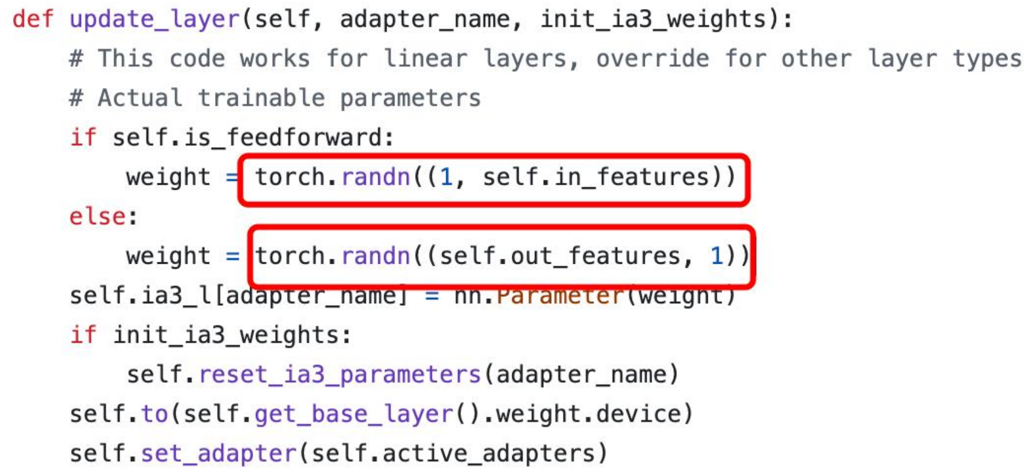


$(IA)^3$ - Infused Adapter by Inhibiting and Amplifying Inner Activations

- Intended to be an improved over LoRA
- Three goals:
  - must add or update as few parameters as possible to avoid incurring storage and memory costs
  - should achieve strong accuracy after training on only a few examples of a new tasks
  - must allow for mixed-task batches
- Main idea:
  - Rescale inner activations with lower-dimensional learned vectors, which are injected into the attention and feedforward modules
- Main differences from LoRA:
  - LoRA learns low-rank **updates** to the attention weights
  - $(IA)^3$ learns injectable vectors.

$(IA)^3$





```
def update_layer(self, adapter_name, init_ia3_weights):
    # This code works for linear layers, override for other layer types
    # Actual trainable parameters
    if self.is_feedforward:
        weight = torch.randn((1, self.in_features))
    else:
        weight = torch.randn((self.out_features, 1))
    self.ia3_l[adapter_name] = nn.Parameter(weight)
    if init_ia3_weights:
        self.reset_ia3_parameters(adapter_name)
    self.to(self.get_base_layer().weight.device)
    self.set_adapter(self.active_adapters)
```

Efficiency of IA3

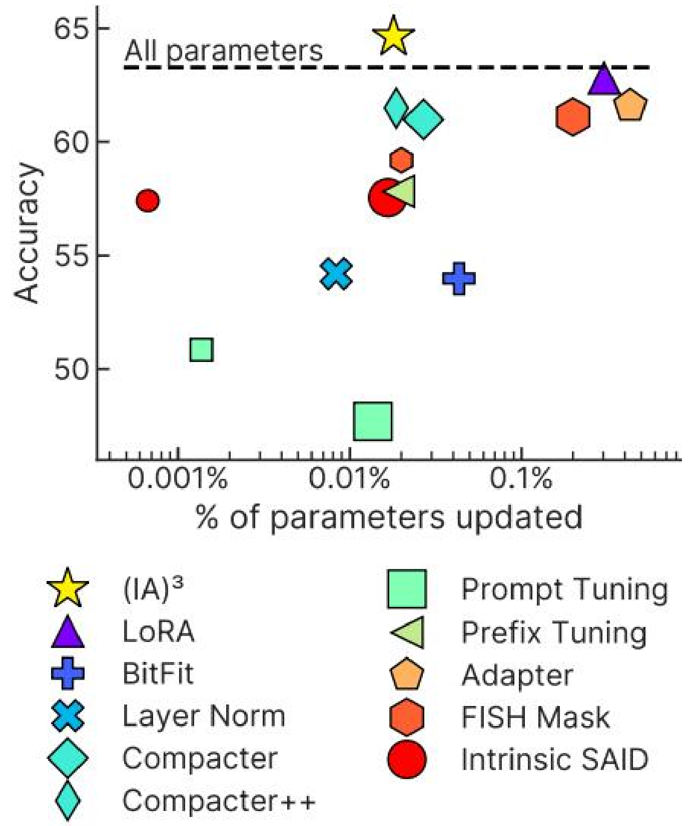


Figure 2: Accuracy of PEFT methods with L_{UL} and L_{LN} when applied to T0-3B. Methods that with variable parameter budgets are represented with larger and smaller markers for more or less parameters.

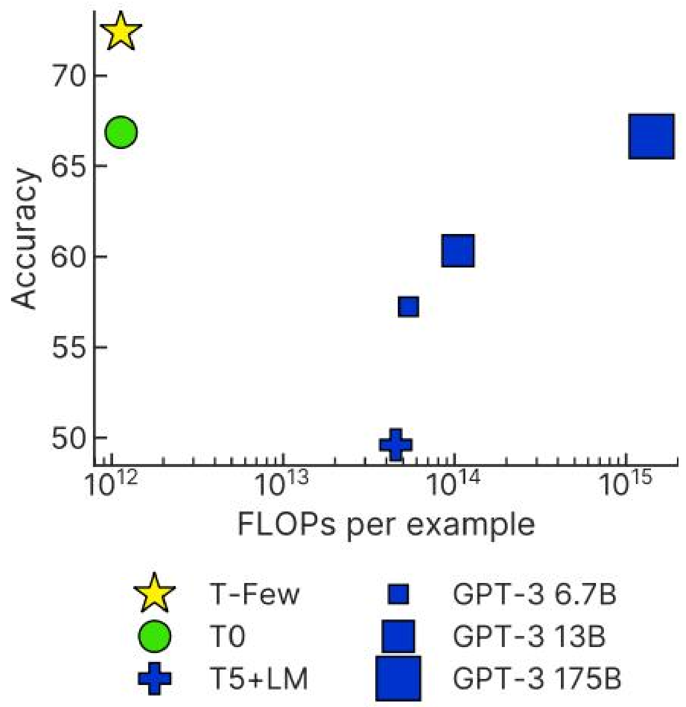


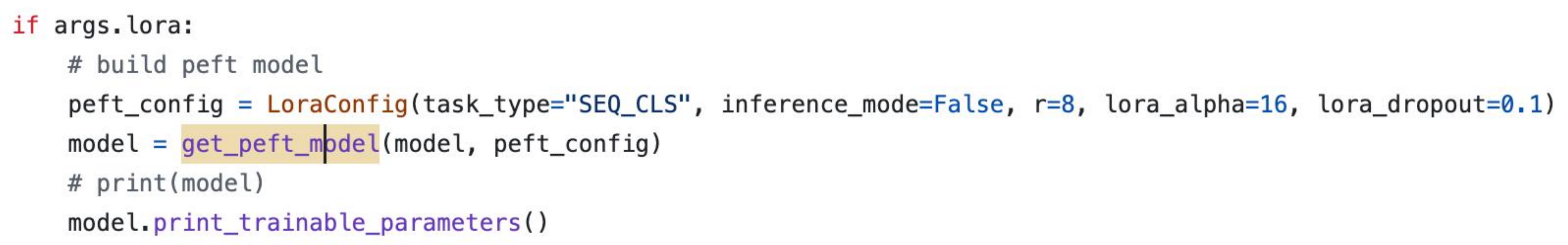
Figure 3: Accuracy of different few-shot learning methods. T-Few uses $(\mathtt{IA})^3$ for PEFT methods of T0, T0 uses zero-shot learning, and T5+LM and the GPT-3 variants use few-shot ICL. The x-axis corresponds to inference costs; details are provided in section 4.2.

Advantages of Re-Paramaterization Methods

- Training tends to be more memory-efficient, since we only need to calculate gradients and maintain optimizer state for a small number of parameters.
- These methods are faster to tune than standard full model finetuning.
- It is straight-forward to swap between tasks by swapping in and our just the tuned weights.

MindNLP Demo

- Just like huggingface peft



```python
if args.lora:
    # build peft model
    peft_config = LoraConfig(task_type="SEQ_CLS", inference_mode=False, r=8, lora_alpha=16, lora_dropout=0.1)
    model = get_peft_model(model, peft_config)
    # print(model)
    model.print_trainable_parameters()
```