CSE 156 | Lecture 14: Parameter-Efficient Fine-Tuning (PEFT)

Ndapa Nakashole

November 14, 2024

Administrative matters

- **PA3**: due Nov 18 (short one!)
- Midterm - **done!**

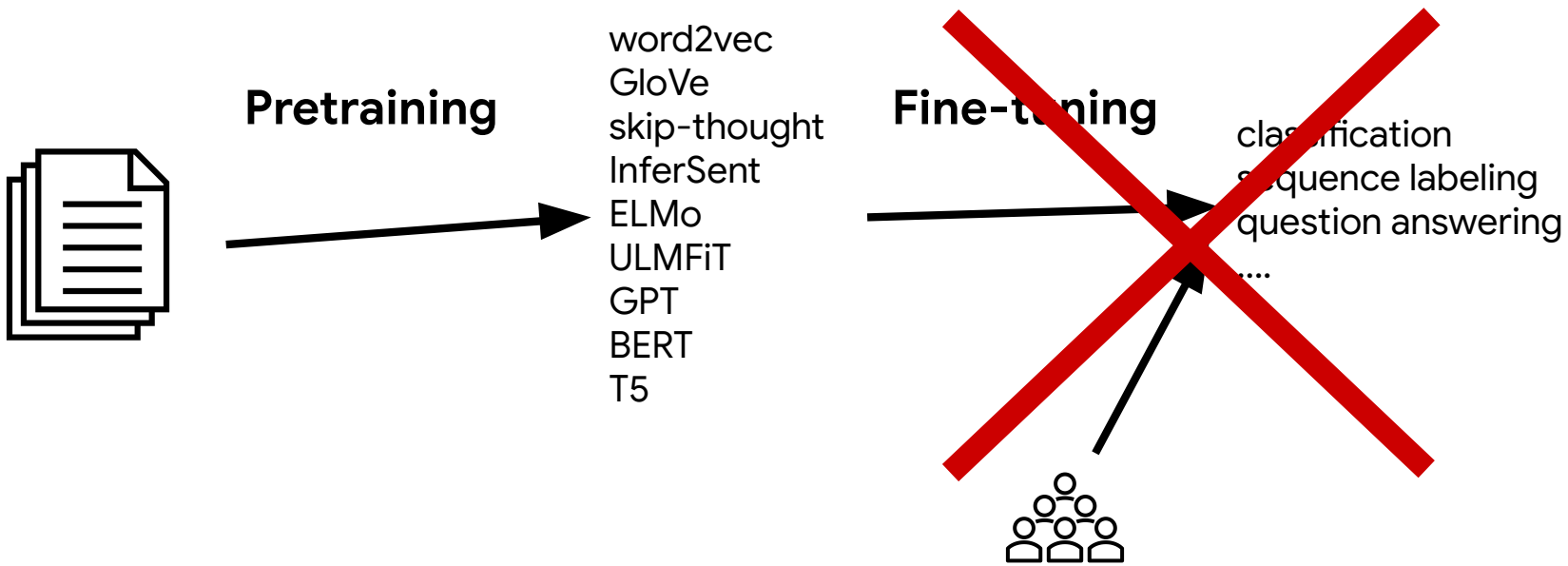
Today

1. PEFT Overview
2. Sparse Fine-Tuning
   - Lottery Ticket Hypothesis
3. Low Rank Adaptation (LoRA)
4. Adapter Functions

Parameter Efficient Fine-Tuning of LLMs

Transfer Learning breaks down on Very Large Models

- With large models, **fine-tuning is expensive**
- The pretrain-finetune transfer learning formula breaks down



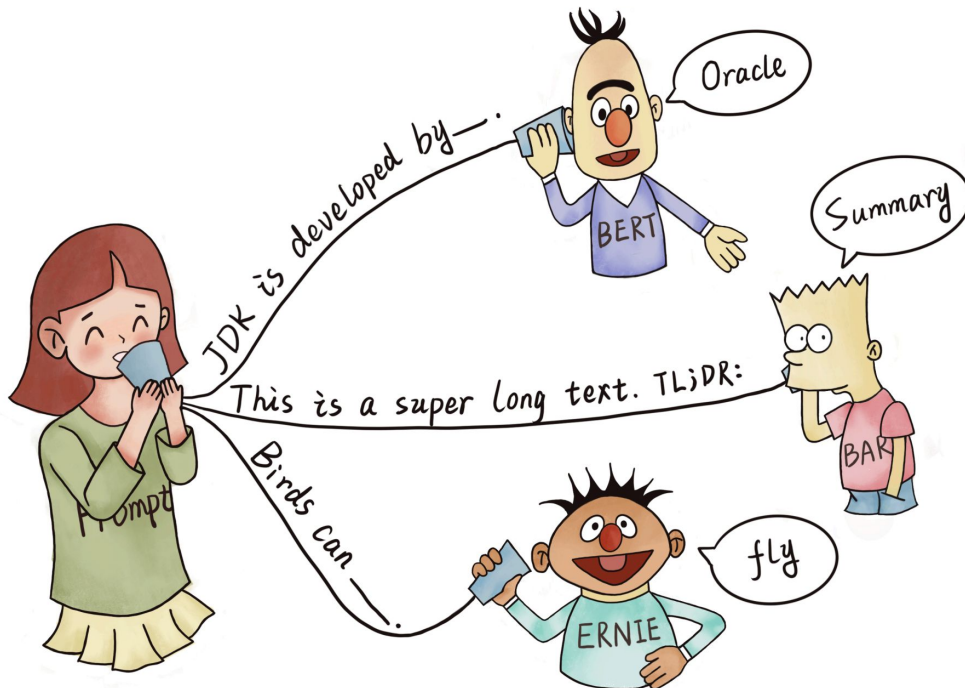
Credit: NAACL 2019 Transfer learning tutorial

Prompting

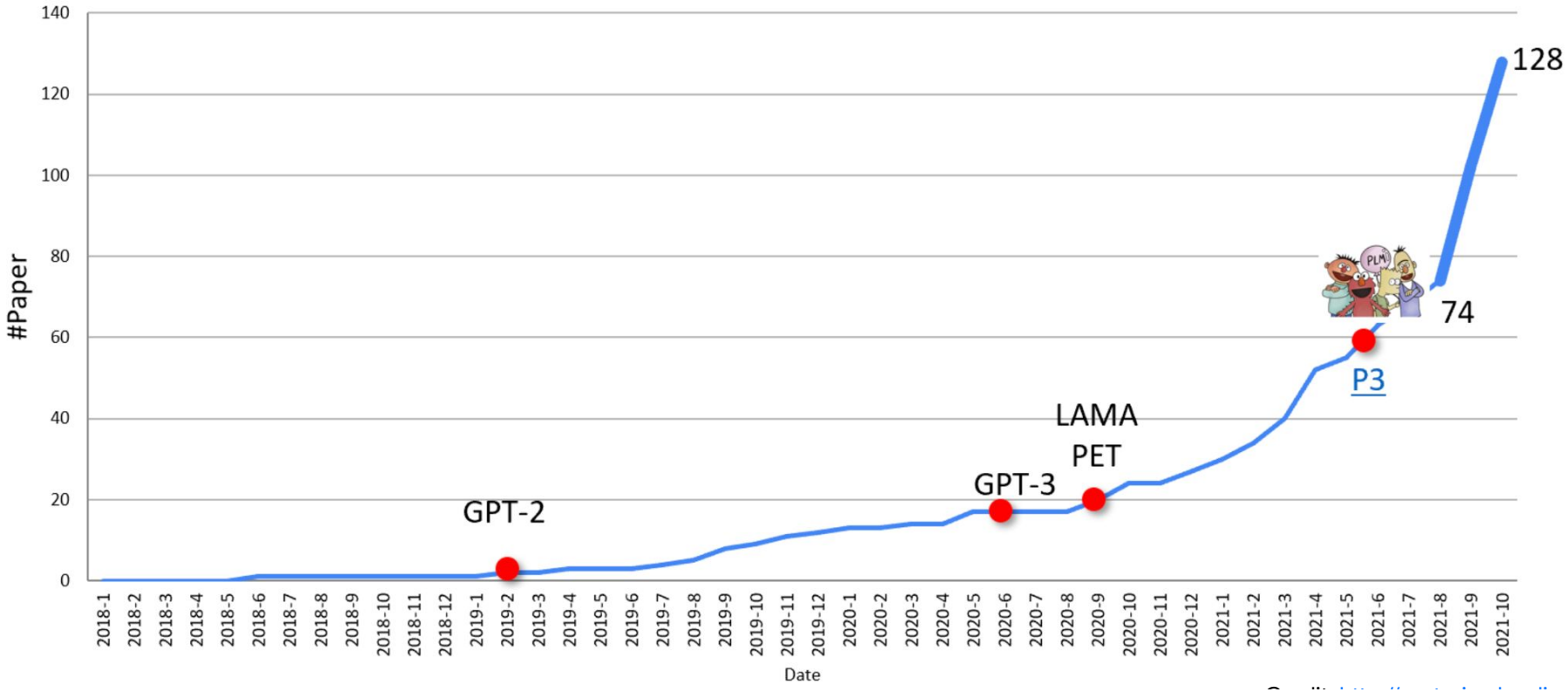
- In-context learning (prompting) has mostly replaced fine-tuning for very large models

Pretraining

BERT
BART
ERNIE
GPT-3
PaLM



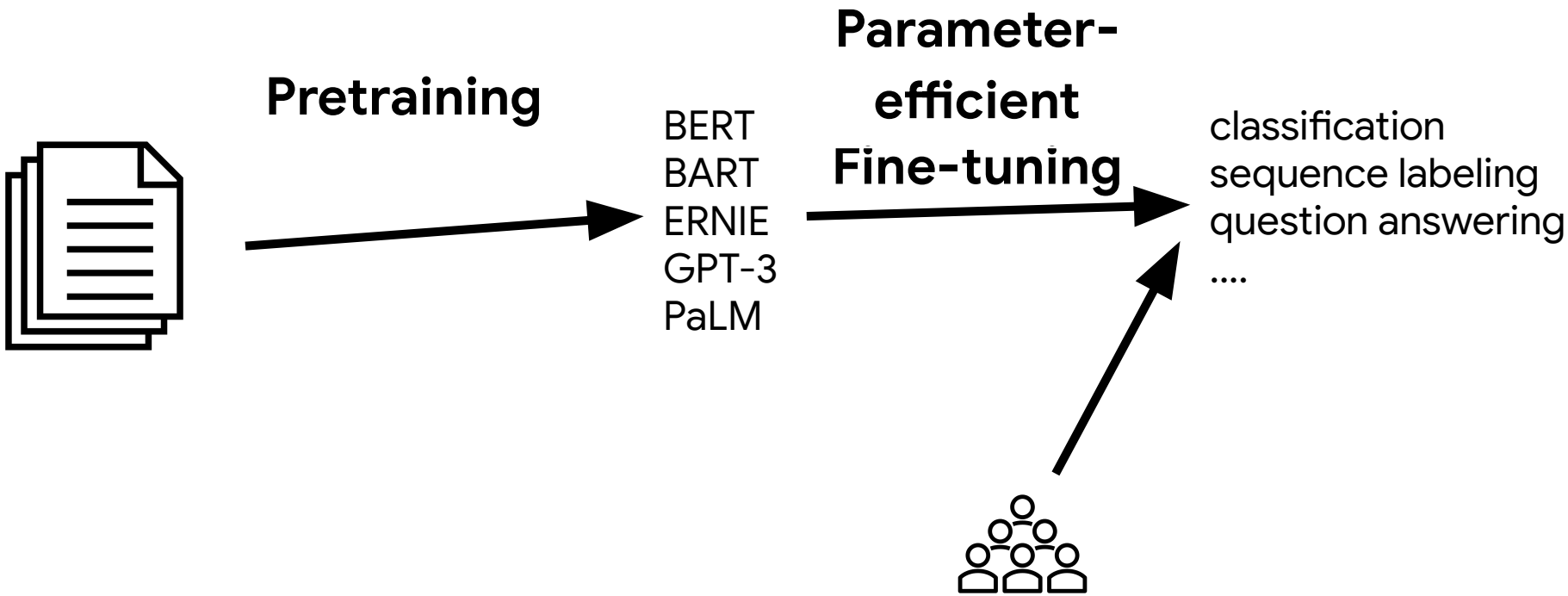
Prompt-based Learning increaded rapidly in NLP



Downsides of Prompt-based Learning

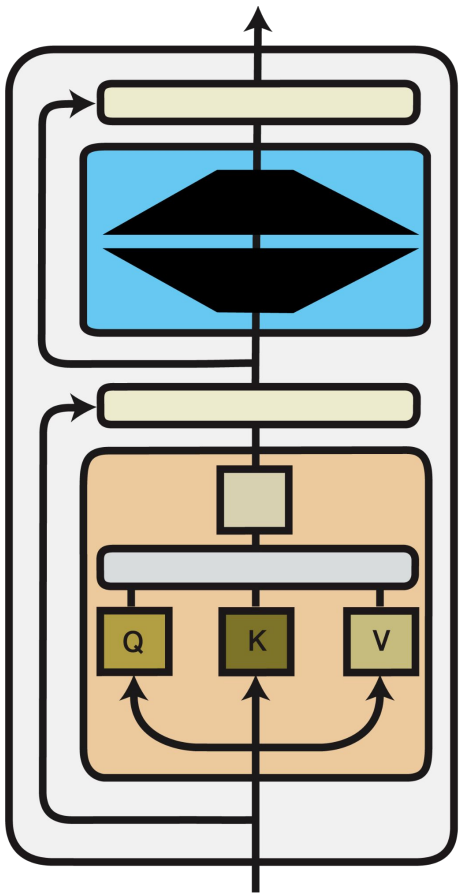
- **Inefficiency**: The prompt needs to be processed every time the model makes a prediction
- **Poor performance**: Prompting generally performs worse than fine-tuning [Brown et al., 2020] and **Sensitivity to the wording of the prompt** [Webson & Pavlick, 2022], order of examples [Zhao et al., 2021; Lu et al., 2022], etc.
- **Lack of clarity regarding what the model learns from the prompt**. Even random labels work [Min et al., 2022]

From Fine-tuning to Parameter-efficient Fine-tuning

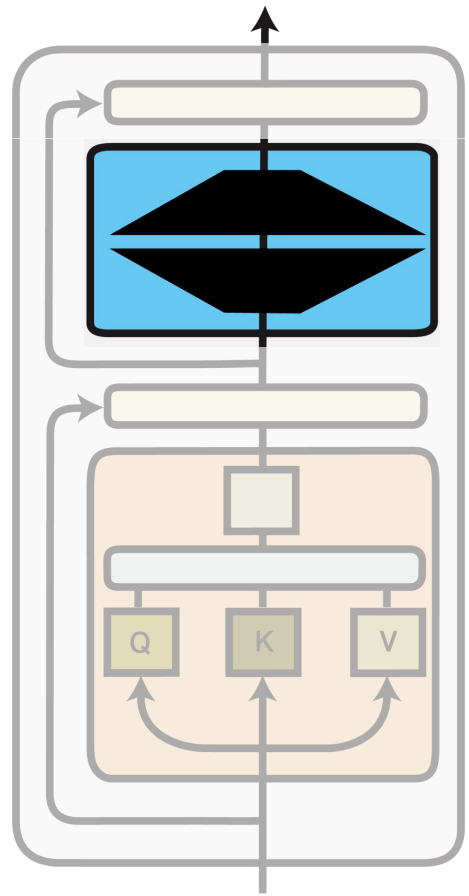


From Fine-tuning to Parameter-efficient Fine-tuning (PEFT)

- **Fine-tuning**: Update **all** parameters



- **PEFT**: Update a **small subset** of parameters



An old Idea

- In NLP, people experimented with static and non-static word embeddings [Kim, 2014]
- ELMo did not fine-tune contextualized word embeddings [Peters et al., 2018]

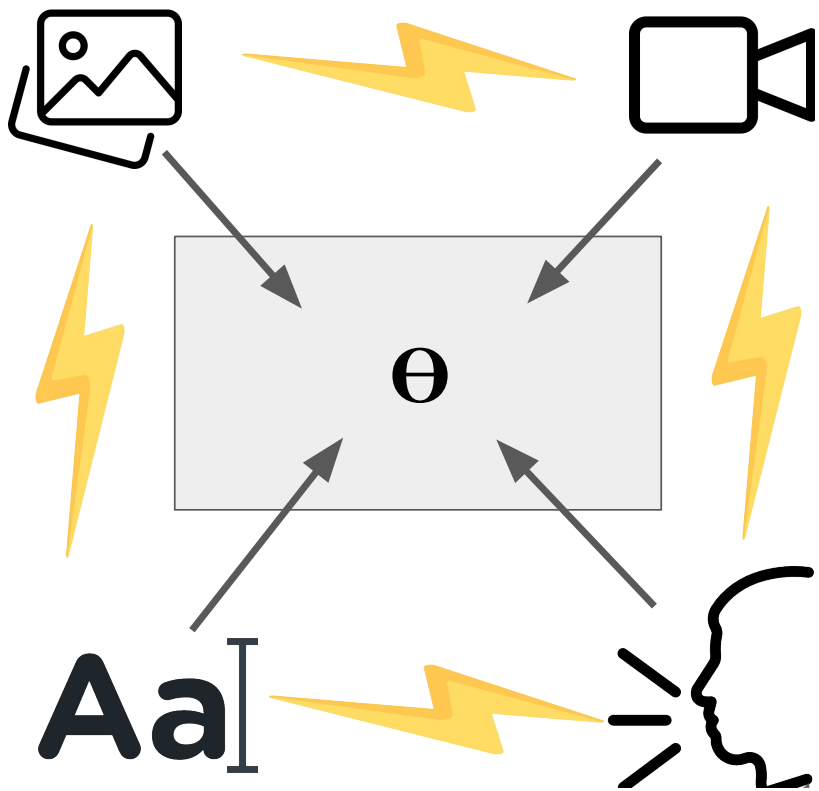
$\rightarrow$ Fine-tuning all representations performed generally better in practice (as in BERT)

Why go back to fine-tuning only some parameters?

- **Efficiency**: Fine-tuning all parameters is impractical with large models
- **State-of-the-art models are massively over-parameterized** $\rightarrow$ Parameter-efficient fine-tuning matches performance of full fine-tuning
- **Modular and compositional representations**

Why Modularity?

1. **Catastrophic forgetting and interference** $\rightarrow$ **Modular** representations can mitigate this



Neural Networks, Functions & Modules

- A neural network $f_\theta : \mathcal{X} \to \mathcal{Y}$ with ℓ layers can be thought of as a composition of functions:

$$f_\theta = f_{\theta_1} \odot f_{\theta_2} \odot \cdots \odot f_{\theta_l}$$

$\odot$ is the **function composition** operator

- Each f_{θ_i} has its own parameters θ_i, where $i = 1, \ldots, l$.

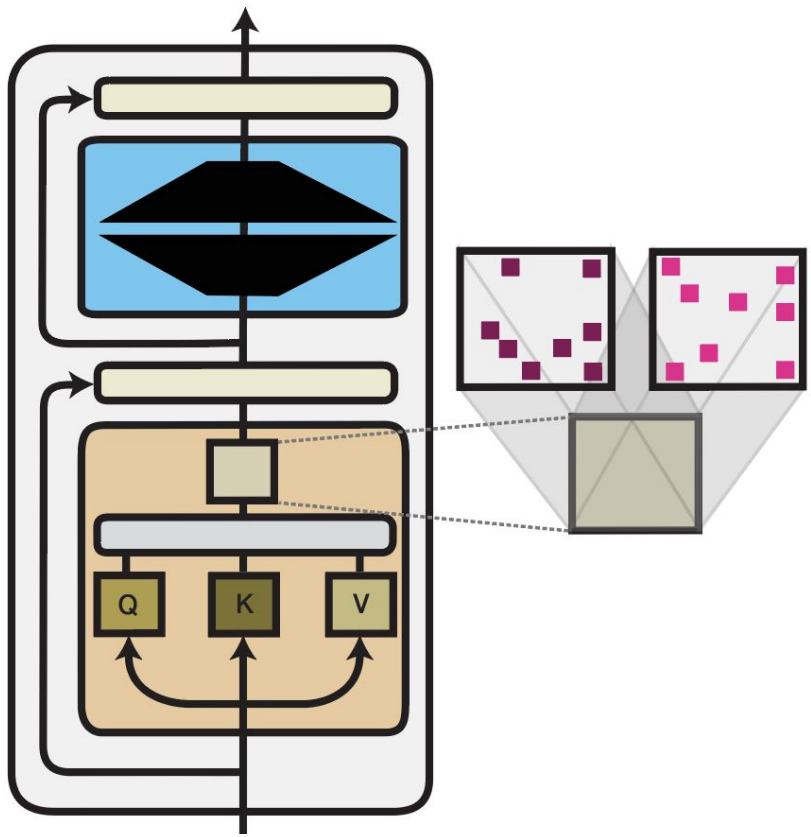
- A module with parameters ϕ can modify the i-th subfunction via parameter composition:

$$f'_i(\boldsymbol{x}) = f_{\theta_i \oplus \phi}(\boldsymbol{x})$$

$\oplus$ denotes the **parameter composition** operator such as element-wise multiplication

Parameter Composition

1. Sparse Subnetworks ←
2. Structured Composition
3. Low-rank Composition



Sparse Subnetworks for PEFT

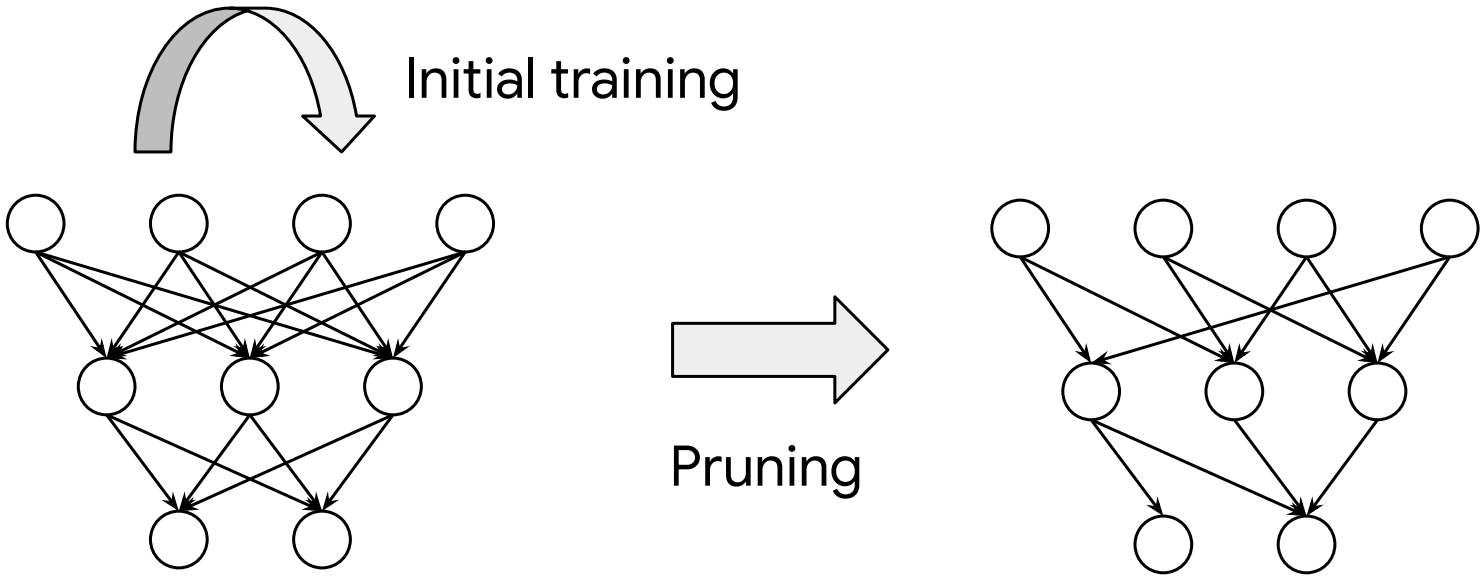
- Most common sparsity method: **pruning**
- Pruning can be seen as applying a binary mask

$$\boldsymbol{b} \in \{0,1\}^{|\theta|}$$

that selectively keeps or removes each connection in a model and produces a subnetwork

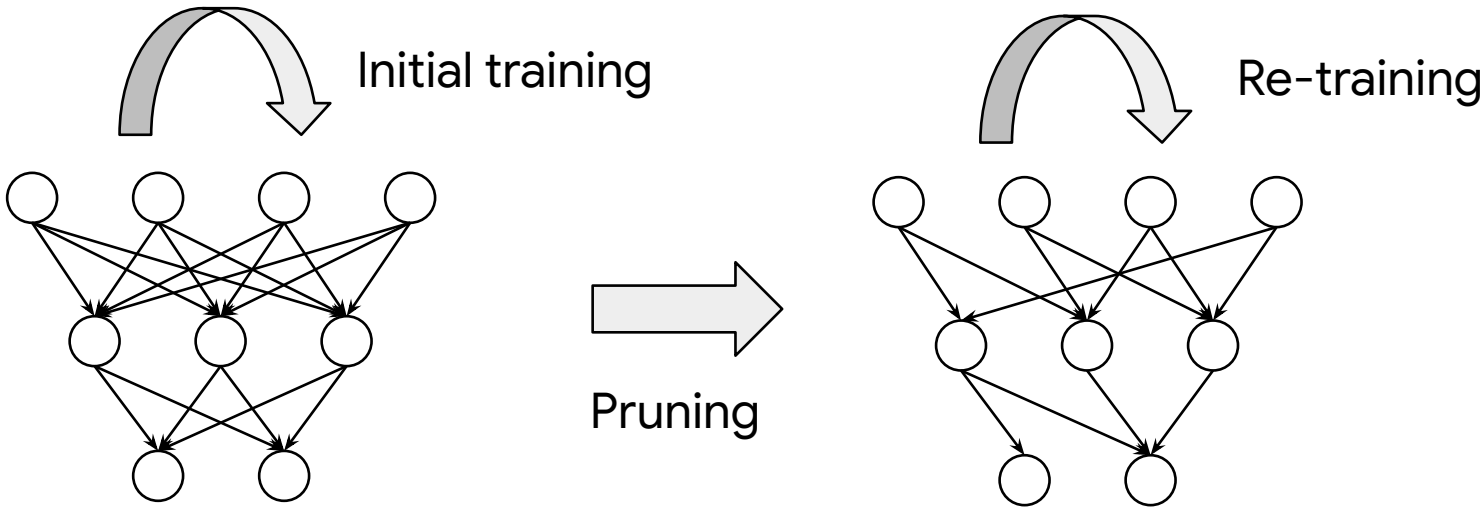
- Most common pruning criterion: **weight magnitude** [Han et al., 2017]

Pruning



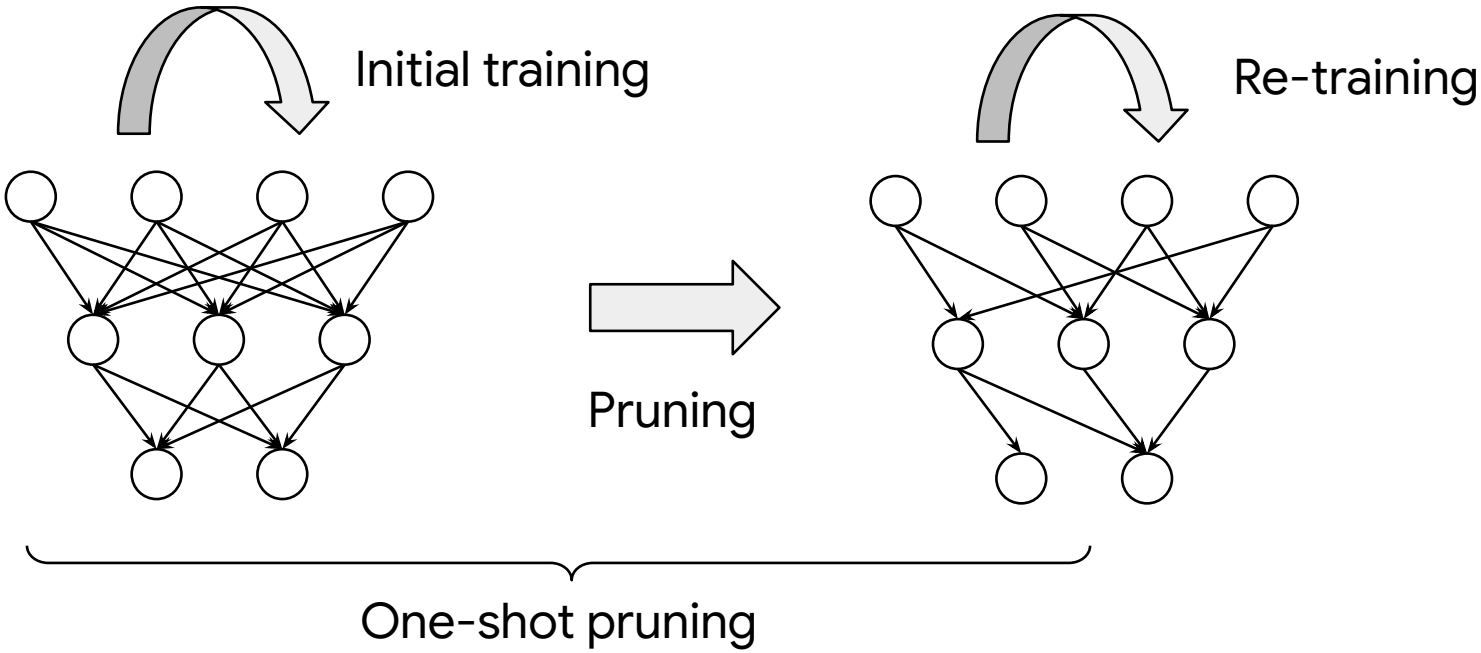
- **Pruning**: a fraction of the **lowest-magnitude weights** are removed

Pruning and Re-training



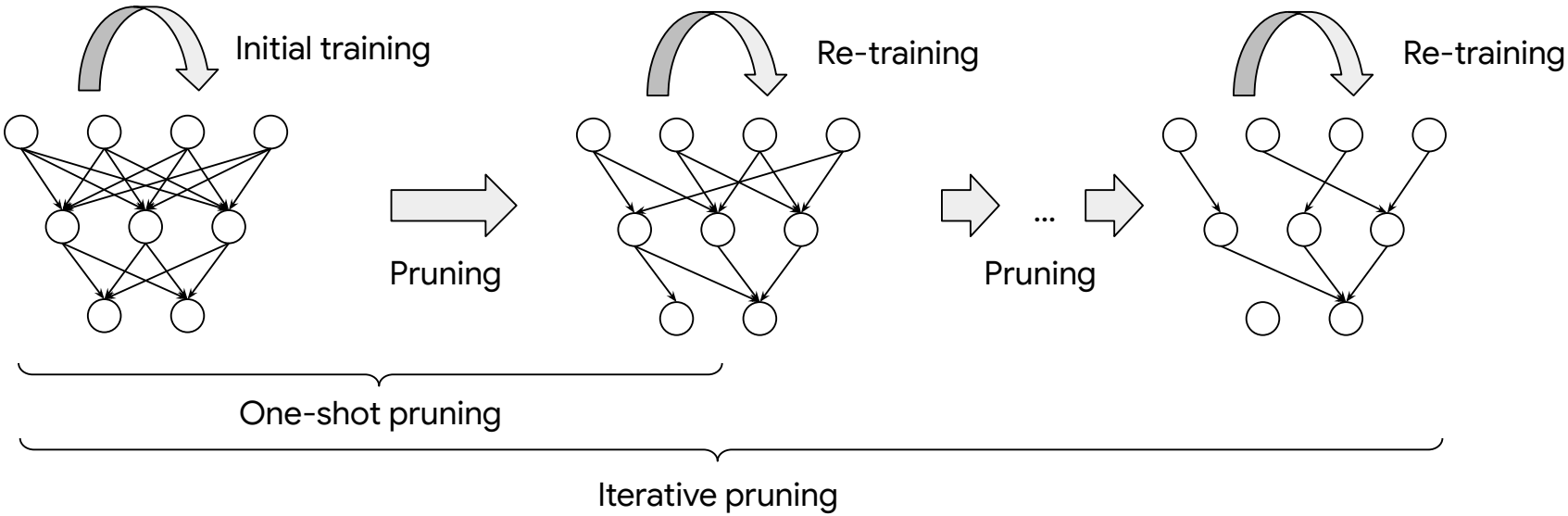
- **Pruning**: a fraction of the **lowest-magnitude weights** are removed
- The non-pruned weights are **re-trained**

One Shot Pruning



- One-shot pruning: **prune once** and **re-train**

Iterative Pruning



- Pruning for multiple iterations is more common [Frankle & Carbin, 2019]
  - Provides multiple opportunities to re-evaluate which weights are essential
  - Leads to a more accurate identification of the best subnetworks "winning ticket"

Another Perspective on Pruning

Pruning as Parameter Adjustment

- Can also view pruning as adding a task-specific vector ϕ to the parameters of an existing model

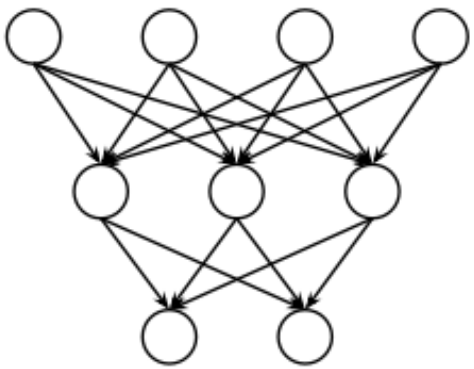
$$f'_\theta = f_{\theta+\phi}$$

- **Sparse Model with Binary Mask**: If the final model should be sparse, can multiply the existing weights with the binary mask to set the pruned weights to 0

$$f'_\theta = f_{\theta \circ b+\phi}$$

- But how do we find the right b, i.e., the right subnetwork?

Finding the Right Subnetwork



Starting with a randomly initialized, dense neural network...

- Is there a subnetwork with the following properties?
  - Same or better results
  - Fewer parameters
  - Shorter training time
  - Trainable from beginning?

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks

Jonathan Frankle
MIT CSAIL
jfrankle@csail.mit.edu

Michael Carbin
MIT CSAIL
mcarbin@csail.mit.edu

Lottery Ticket Hypothesis

A randomly-initialized, dense neural network **contains a sub-network ("winning ticket")** that is **initialized** such that-when trained in isolation-it **can match the test accuracy of the original network** after training for at most the same number of iterations.

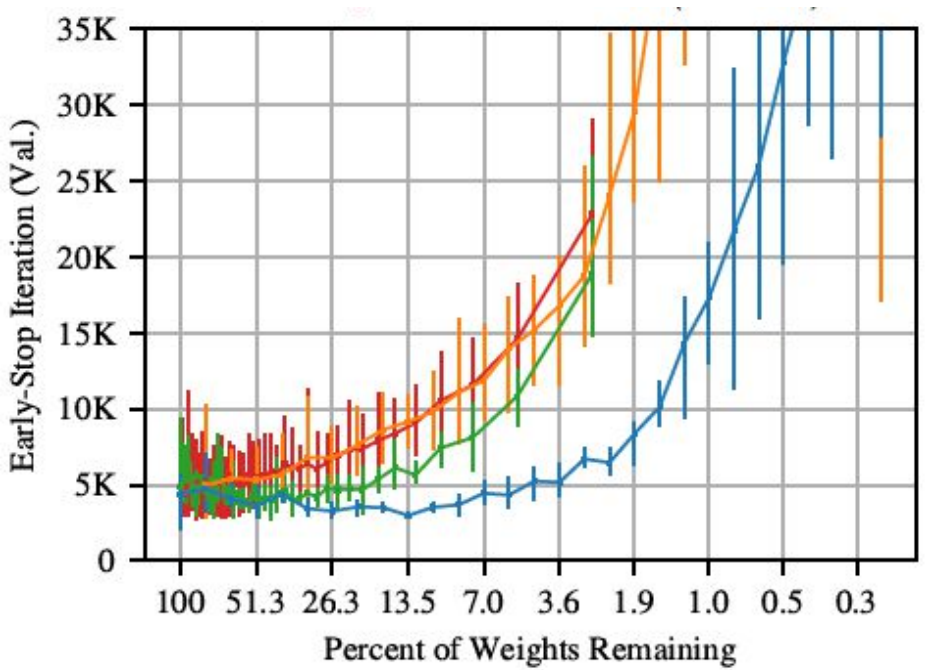
Identifying Winning Tickets

1. Randomly initialize a neural network $f(x; \theta_0)$
2. Train the network for j iterations, arriving at parameters θ_j
3. Prune $p\%$ of the parameters in θ_j, creating a mask m
4. **Reset** the remaining parameters to their values in θ_0, creating the winning ticket $f(x; m \odot \theta_0)$

For iterative pruning, steps 2-4 are repeated for n rounds

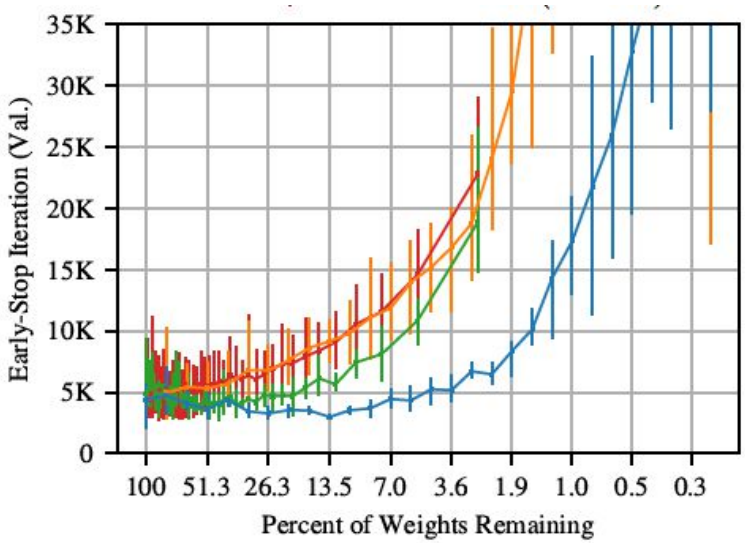
Winning Tickets

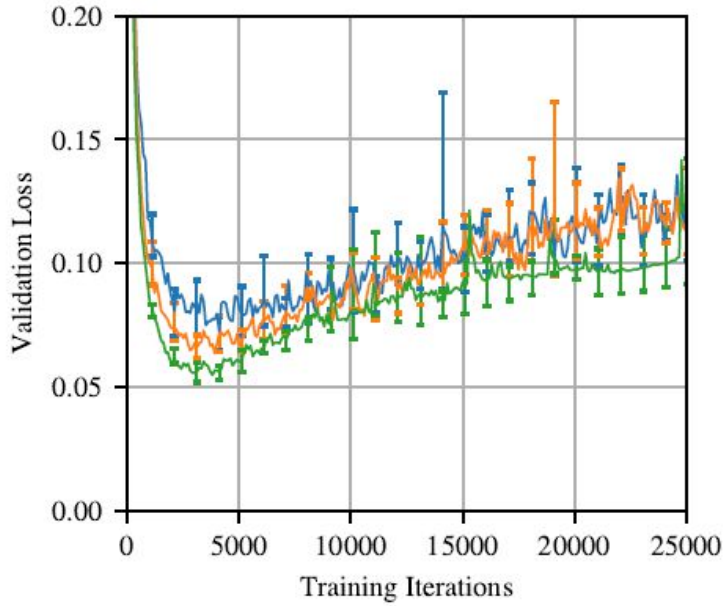
[Frankle & Carbin, 2019]



- Each color represents a different model (Lenet, Conv-4, etc.)
- Average of x trials
- Error bars for the
  - Minimum value
  - Maximum value

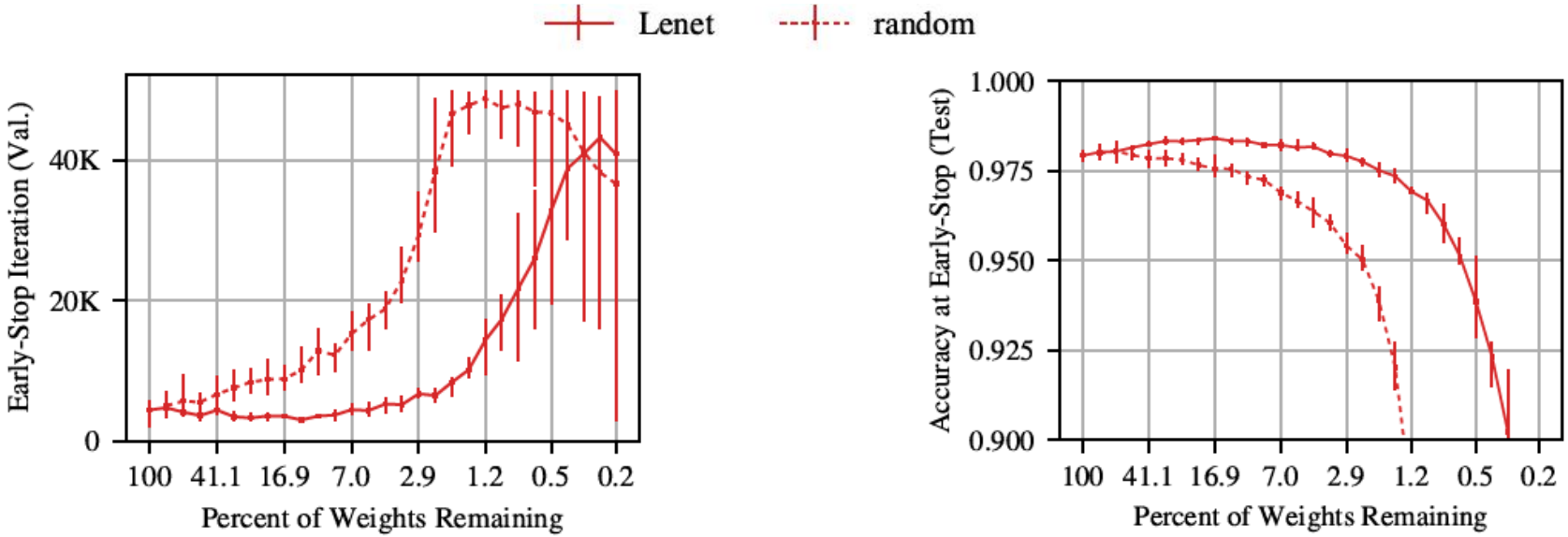
Early-Stopping Criterion Identification





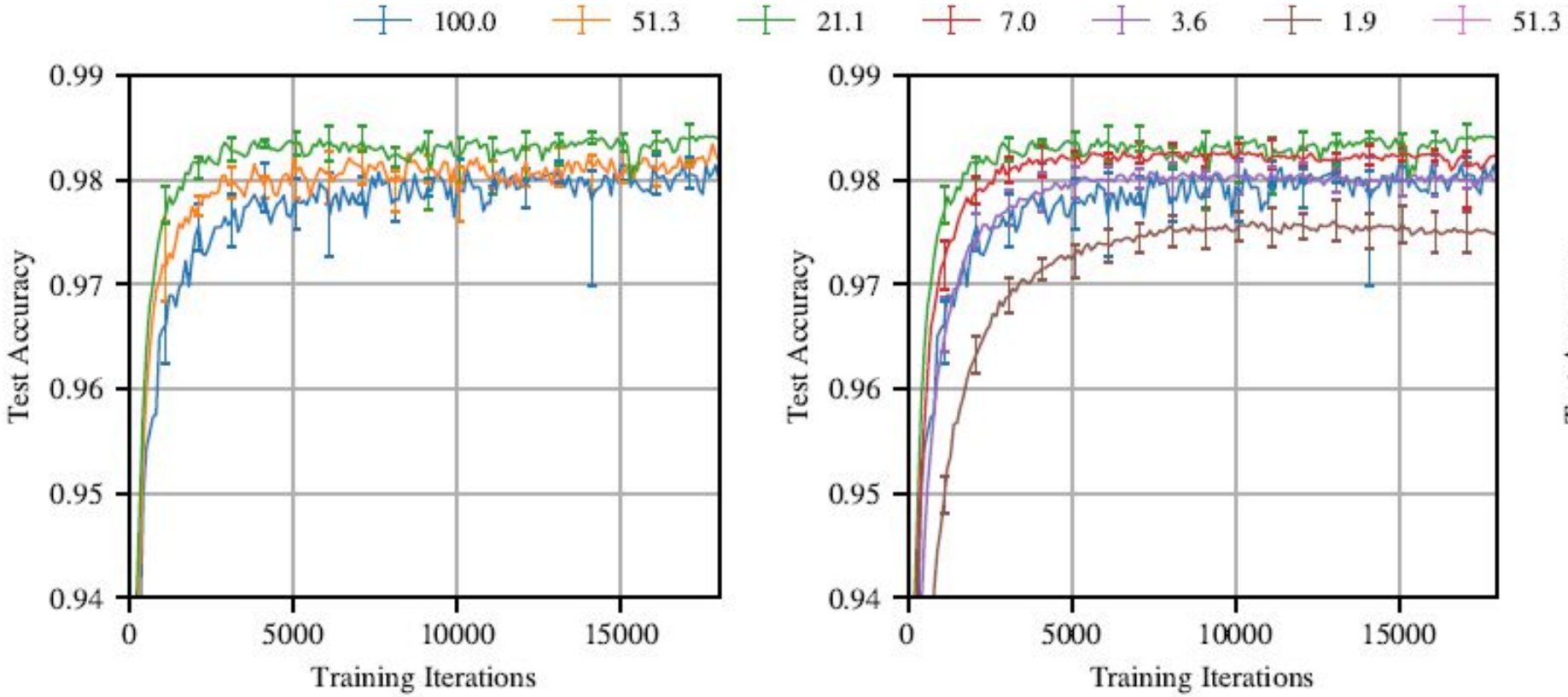
- Early stopping criterion is the iteration of minimum validation loss
- Validation loss initially drops, after which it forms a clear bottom and then begins increasing again. Early-stopping criterion identifies this bottom

Winning Tickets Vs Randomly Sampled ones



- Dashed lines are randomly sampled sparse networks (average of ten trials)
- Solid lines are winning tickets (average of five trials)

Winning Tickets in Fully-Connected Networks



- Labels: fraction of weights remaining in the network after pruning

The Lottery Ticket Hypothesis in Pre-trained Models

- Prior work [Chen et al., 2020; Prasanna et al., 2020] has found winning tickets in pre-trained models such as BERT
- Sparsity ratios: from 40% (SQuAD) to 90% (QQP and WNLI)
- Pre-trained init $\gg$ random init

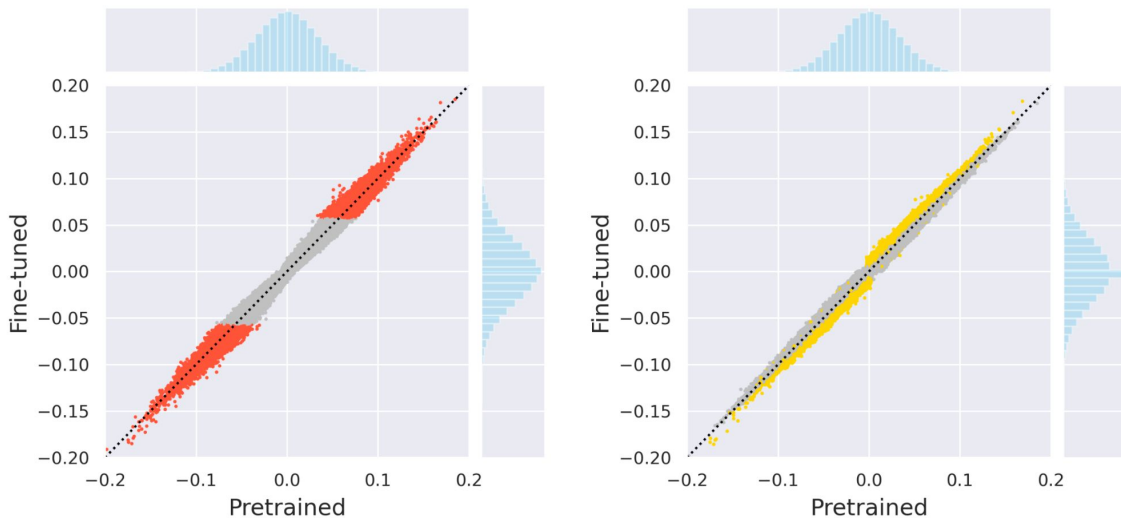
Insights from Lottery Ticket Hypothesis

- The weights that end up in the winning hypothesis **travel much further in the optimization space** compared to those not in the winning hypothesis
- This suggests that **the good network is not already present in the initialization**
- Instead, the good network is particularly well-suited to be optimized by SGD.

Pruning Pre-trained Models

- Pruning does not consider how weights change during fine-tuning
- **Magnitude pruning**: keep weights farthest from 0
- **Movement pruning** [Sanh et al., 2020]: keep weights that *move the most away from 0* during fine-tuning

Fine-tuned weights stay close to their their pre-trained values. Magnitude pruning (left) selects weights that are far from 0.



Movement pruning (right) selects weights that move away from 0.

Diff Pruning

- For a model with $f'_\theta = f_{\theta+\phi}$, we can perform pruning only based on the magnitude of the module parameters ϕ rather than the updated parameters $\theta + \phi$
- Diff pruning [Guo et al.. 2021] prunes the module parameters via magnitude pruning to make them sparse

Lottery Ticket Sparse Fine-Tuning (LT-SFT)

Phase 1: Fine-Tuning and Mask Creation

- Pretrained model parameters $\boldsymbol{\theta}^{(0)}$ are fully fine-tuned on target data $\mathcal{D}$, yielding $\boldsymbol{\theta}^{(1)}$.
- Parameters ranked by greatest absolute difference $\left|\theta_i^{(1)} - \theta_i^{(0)}\right|$.
- Top K parameters selected; binary mask $\boldsymbol{\mu}$ created

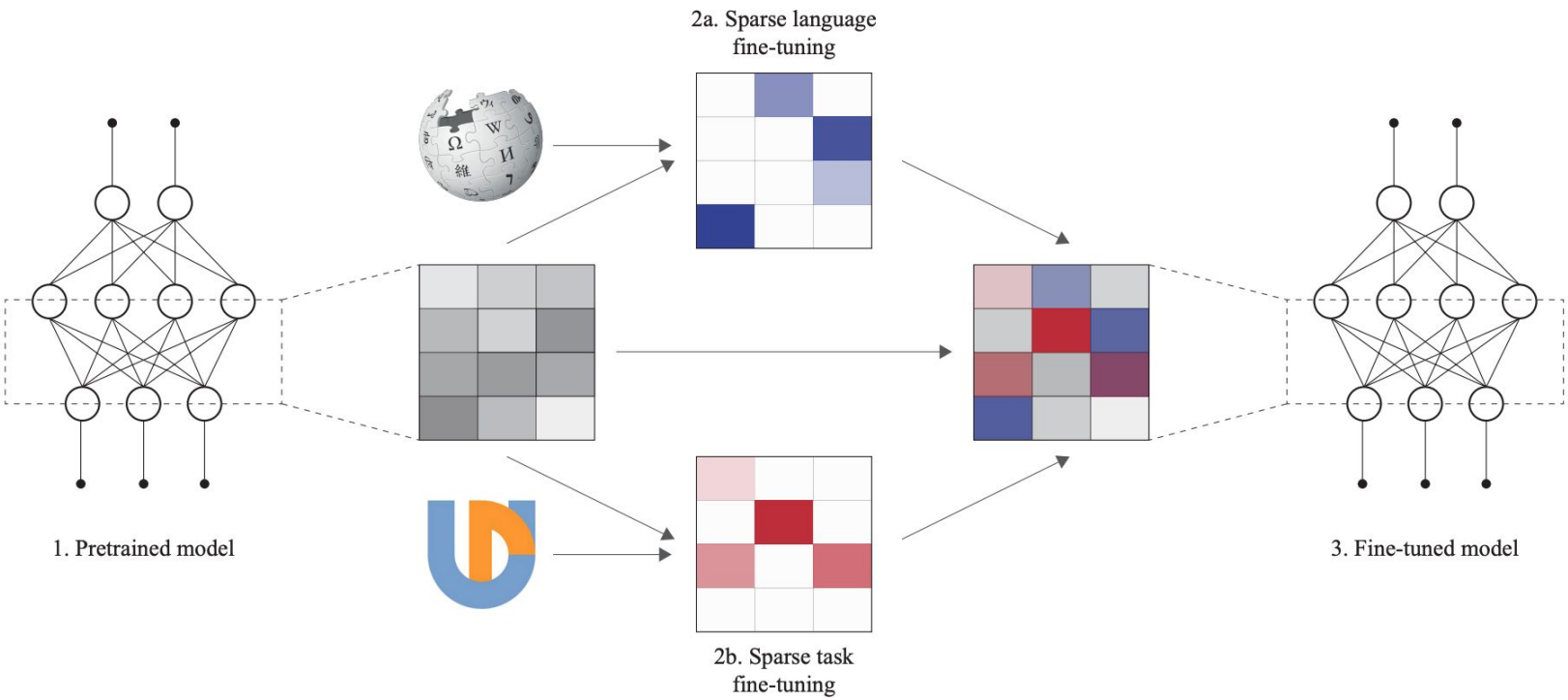
Phase 2: Sparse Fine-Tuning

- Reset parameters to $\boldsymbol{\theta}^{(0)}$.
- Fine-tune only the K selected parameters; others frozen.
- Obtain sparse vector of differences $\boldsymbol{\phi} = \boldsymbol{\theta}^{(2)} - \boldsymbol{\theta}^{(0)}$.

Lottery Ticket Sparse Fine-Tuning (LT-SFT)

- Keeping the pre-trained weights allows combining subnetworks for different settings:

$$f'_\theta = f_{\theta+\phi^{\mathrm{A}}+\phi^{\mathrm{B}}}$$



Summary

- Winning tickets, subnetworks, can be found in pre-trained models
- Sparse fine-tuning only updates subset of parameters
- Strengths and weaknesses of sparse fine-tuning:
  - **Training efficiency** (-): not so great, pruning requires re-training iterations
  - **Memory efficiency** (+): great, smaller model
  - **Inference efficiency** (+): great a smaller model is used
  - **Compositionality**(+): Subnetworks can be composed

Structured Composition & Low-rank Adaptation in Transformers

Structured Composition

- We can additionally impose a structure on the weights that we select
- Specifically, we can only modify the weights that are associated with a pre-defined group $\mathcal{G}$: $f_i' = f_{\theta_i + \phi_i} \forall f_i' \in \mathcal{G}$
- Most common setting: each group $\mathcal{G}$ corresponds to a layer; only update the parameters associated with certain layers
- Groups can also relate to more fine-grained components

Bias-only Fine-tuning: BitFit

A practical choice: updating only biases $\boldsymbol{b}$ Computing, [Zaken et al., 2022]

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

Query and second MLP layer biases are most important!

- BitFit peformance: degraded performance a bit but $< 0.1\%$ of the parameters are tuned

LoRA (Hu et al., 2021)

- LoRA: Low-rank Adaptation adds trainable low-rank matrices into transformer layers to approximate the weight updates
- For a pre-trained weight matrix $\boldsymbol{W}_0 \in \mathbb{R}^{d \times k}$,

$$h = W_0 x + \Delta W x = W_0 x + \boldsymbol{A}_{\text{down}} \boldsymbol{B}_{\text{up}} x$$

  where $\boldsymbol{A}_{\text{down}} \in \mathbb{R}^{d \times r}$ and $\boldsymbol{B}_{\text{up}} \in \mathbb{R}^{r \times k}$ are tunable parameters

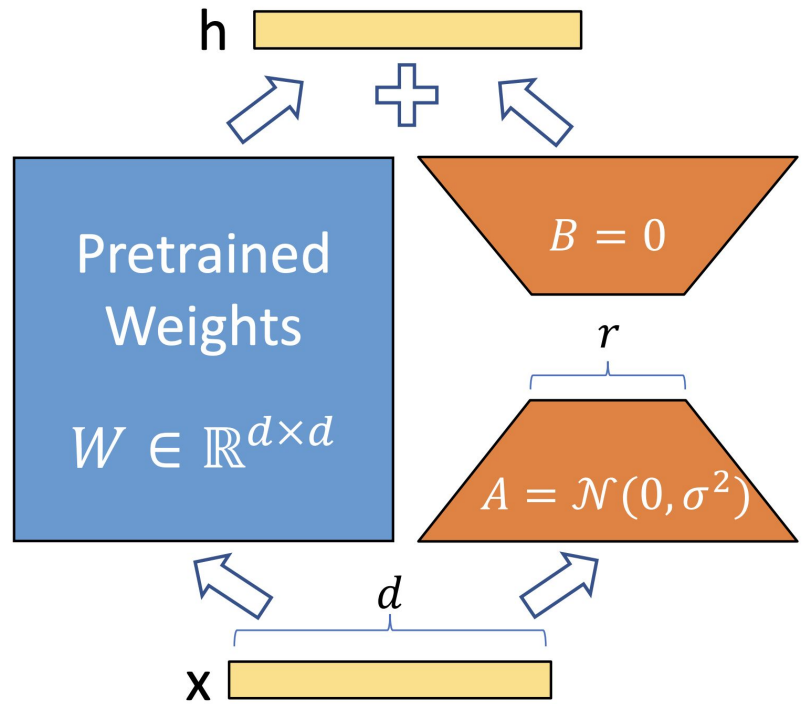
- LoRA applies this update to the query and value projection matrices $(\boldsymbol{W}_q, \boldsymbol{W}_v)$

Low-rank Adaptation (LoRA)

Hu et al. [2022]

$$h = W_0 x + \Delta W x = W_0 x + ABx$$

- Learn weight matrices as $(W + AB)$, where AB is a product of two low-rank matrices
- **Initialization**: use a random Gaussian initialization for A and zero for B, so $\Delta W = BA$ is zero at the beginning of training



What is the optimal rank for LoRA?

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

Parameter Reduction from LoRA

- Original matrix size: $d \times d$
- Original number of parameters: d^2
- LoRA introduces:
  - Down-projection matrix $\boldsymbol{A}_{\text{down}} \in \mathbb{R}^{d \times r}$ with $d \times r$ parameters
  - Up-projection matrix $\boldsymbol{B}_{\text{up}} \in \mathbb{R}^{r \times d}$ with $r \times d$ parameters
- Total parameters in LoRA: $2dr$
- Parameter reduction:

$$\text{Reduction} = d^2 - 2dr$$

- Percentage reduction:

$$\left(1 - \frac{2r}{d}\right) \times 100$$

Parameter Reduction from LoRA with GPT-3 Example

Example with $r = 16$, $d = 12288$ (GPT-3):

- Original matrix size: 12288×12288
- Original number of parameters: $12288^2 = 151,000,064$
- LoRA introduces:
  - Down-projection matrix $\boldsymbol{A}_{\text{down}} \in \mathbb{R}^{12288 \times 16}$ with $196,608$ parameters
  - Up-projection matrix $\boldsymbol{B}_{\text{up}} \in \mathbb{R}^{16 \times 12288}$ with $196,608$ parameters
- Total parameters in LoRA: $393,216$
- Parameter reduction: $151,000,064 - 393,216 = 150,606,848$
- Percentage reduction: 99.7402%

Low-rank Adaptation (LoRA)

LoRA is better than FitBit, even better than Full fine-tuning on GLUE!

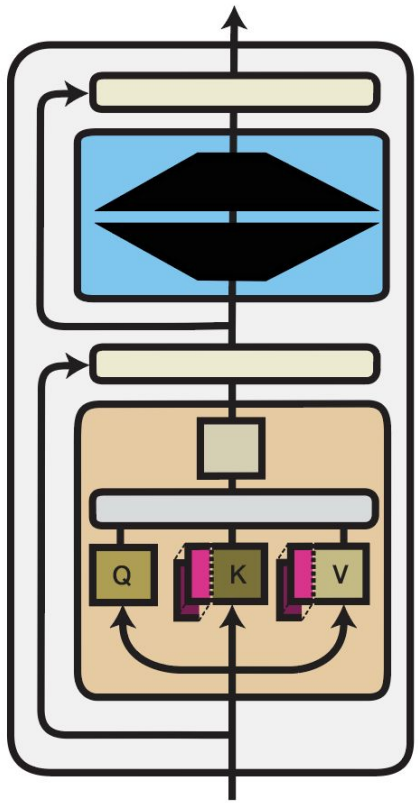
Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB_{base} (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB_{base} (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB_{base} (Adpt^{D})*	0.3M	$87.1_{\pm.0}$	$94.2_{\pm.1}$	$88.5_{\pm1.1}$	$60.8_{\pm.4}$	$93.1_{\pm.1}$	$90.2_{\pm.0}$	$71.5_{\pm2.7}$	$89.7_{\pm.3}$	84.4
RoB_{base} (Adpt^{D})*	0.9M	$87.3_{\pm.1}$	$94.7_{\pm.3}$	$88.4_{\pm.1}$	$62.6_{\pm.9}$	$93.0_{\pm.2}$	$90.6_{\pm.0}$	$75.9_{\pm2.2}$	$90.3_{\pm.1}$	85.4
RoB_{base} (LoRA)	0.3M	$87.5_{\pm.3}$	$\mathbf{95.1}_{\pm.2}$	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm.3}$	$90.8_{\pm.1}$	$\mathbf{86.6}_{\pm.7}$	$\mathbf{91.5}_{\pm.2}$	**87.2**
$\text{RoB}_{\text{large}}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
$\text{RoB}_{\text{large}}$ (LoRA)	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.9}_{\pm1.2}$	$\mathbf{68.2}_{\pm1.9}$	$\mathbf{94.9}_{\pm.3}$	$91.6_{\pm.1}$	$\mathbf{87.4}_{\pm2.5}$	$\mathbf{92.6}_{\pm.2}$	**89.0**
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	3.0M	$90.2_{\pm.3}$	$96.1_{\pm.3}$	$90.2_{\pm.7}$	$\mathbf{68.3}_{\pm1.0}$	$\mathbf{94.8}_{\pm.2}$	$\mathbf{91.9}_{\pm.1}$	$83.8_{\pm2.9}$	$92.1_{\pm.7}$	88.4
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	0.8M	$\mathbf{90.5}_{\pm.3}$	$\mathbf{96.6}_{\pm.2}$	$89.7_{\pm1.2}$	$67.8_{\pm2.5}$	$\mathbf{94.8}_{\pm.3}$	$91.7_{\pm.2}$	$80.1_{\pm2.9}$	$91.9_{\pm.4}$	87.9
$\text{RoB}_{\text{large}}$ (Adpt^{H})†	6.0M	$89.9_{\pm.5}$	$96.2_{\pm.3}$	$88.7_{\pm2.9}$	$66.5_{\pm4.4}$	$94.7_{\pm.2}$	$92.1_{\pm.1}$	$83.4_{\pm1.1}$	$91.0_{\pm1.7}$	87.8
$\text{RoB}_{\text{large}}$ (Adpt^{H})†	0.8M	$90.3_{\pm.3}$	$96.3_{\pm.5}$	$87.7_{\pm1.7}$	$66.3_{\pm2.0}$	$94.7_{\pm.2}$	$91.5_{\pm.1}$	$72.9_{\pm2.9}$	$91.5_{\pm.5}$	86.4
$\text{RoB}_{\text{large}}$ (LoRA)†	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.2}_{\pm1.0}$	$68.2_{\pm1.9}$	$\mathbf{94.8}_{\pm.3}$	$91.6_{\pm.2}$	$\mathbf{85.2}_{\pm1.1}$	$\mathbf{92.3}_{\pm.5}$	**88.6**
DeB_{XXL} (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeB_{XXL} (LoRA)	4.7M	$\mathbf{91.9}_{\pm.2}$	$96.9_{\pm.2}$	$\mathbf{92.6}_{\pm.6}$	$\mathbf{72.4}_{\pm1.1}$	$\mathbf{96.0}_{\pm.1}$	$\mathbf{92.9}_{\pm.1}$	$\mathbf{94.9}_{\pm.4}$	$\mathbf{93.0}_{\pm.2}$	**91.3**

Input Composition

Input Composition

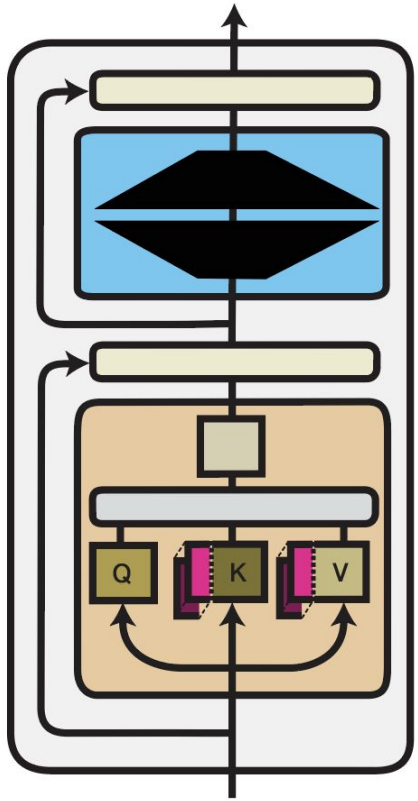
- Augment model's input with a learnable parameter vector ϕ_i :

$$f_i'(\boldsymbol{x}) = f_{\theta_i}\left([\phi_i, \boldsymbol{x}]\right)$$



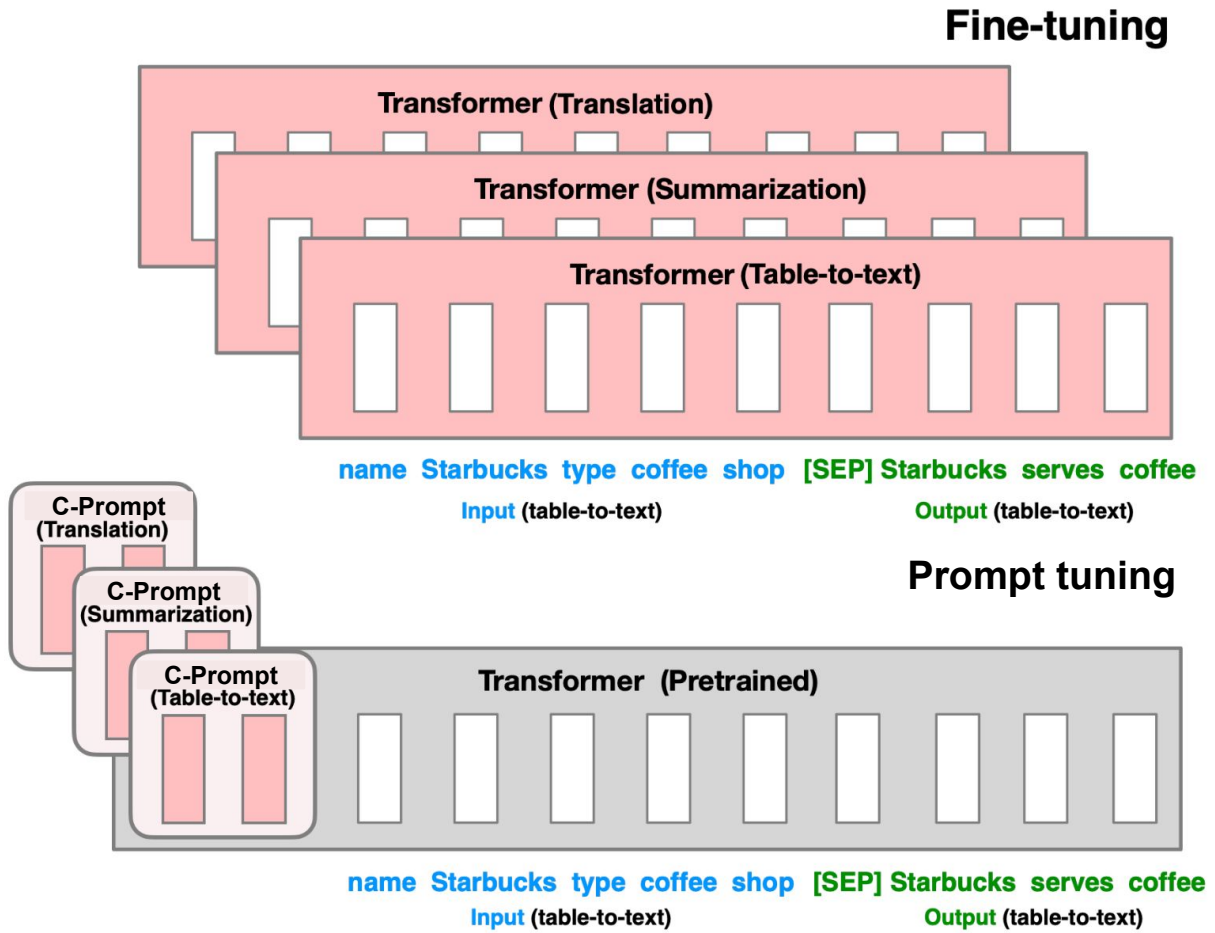
Input Composition and Prompting

- Standard prompting can be seen as finding a discrete text prompt that-when embedded using the model's embedding layer-yields ϕ_i
- However, models are sensitive to the formulation of the prompt [Webson & Pavlick, 2022] and to the order of examples [Zhao et al., 2021; Lu et al., 2022]



Prompt Tuning

- Instead, we can directly learn a continuous prompt ϕ that is prepended to the input [Liu et al.. 2021; Hambardzumyan et al., 2021; Lester et al., 2021]



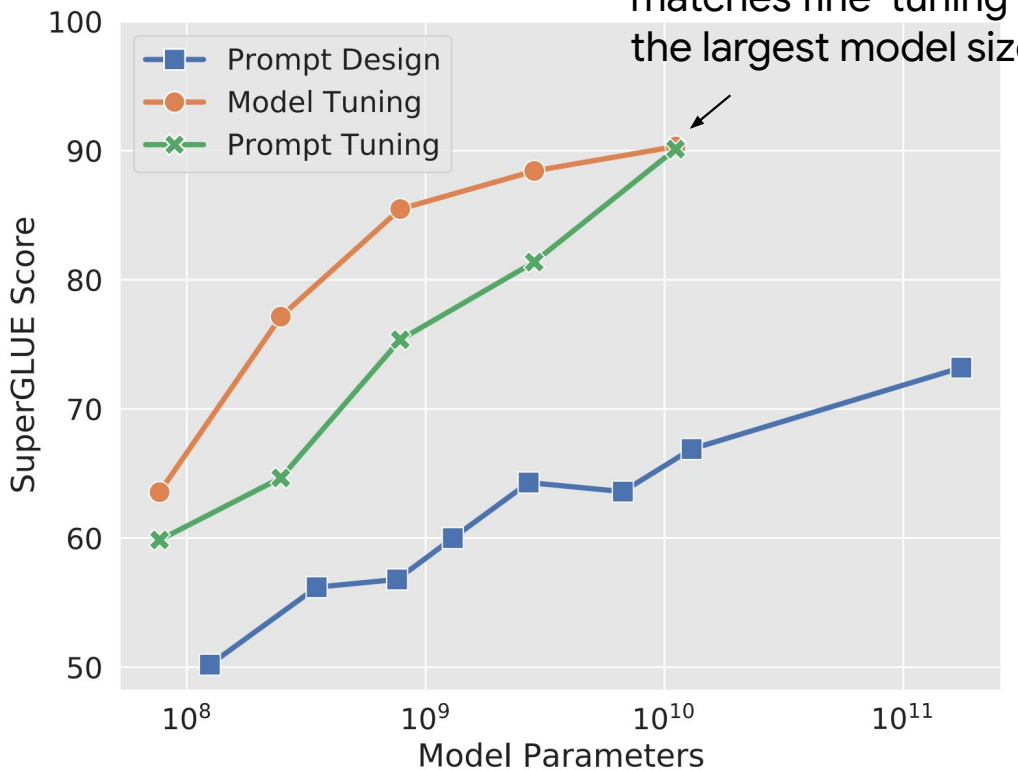
Fine-tuning vs Prompt tuning
(adapted from [Li & Liang, 2021])

Prompt Tuning Only Works Well at Scale

- Only using trainable parameters at the input layer limits capacity for adaptation

→ Prompt tuning performs poorly at smaller model sizes and on harder tasks [Mahabadi et al., 2021; Liu et al., 2022]

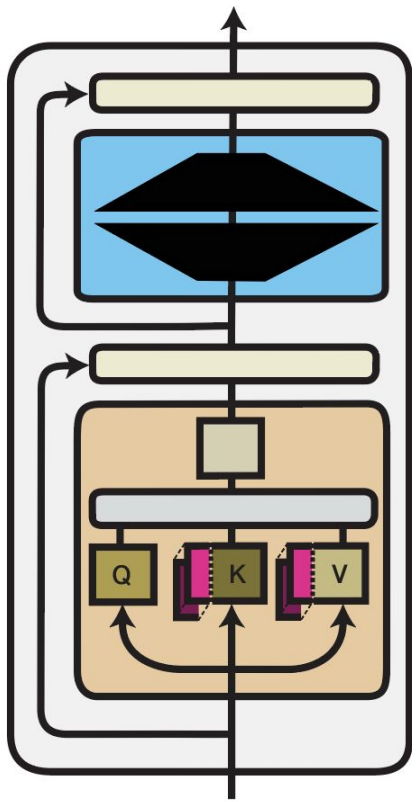
Prompt tuning only matches fine-tuning at the largest model size



Prompt tuning vs standard fine-tuning and prompt design across T5 models of different sizes [Lester et al., 2021]

Multi-Layer Prompt Tuning

- Instead of learning ϕ_i parameters only at the input layer, we can learn them at every layer of the model
- Continuous prompts ϕ_i are concatenated with the keys and values in the self-attention layer [Li& Liang, 2021]



Function Composition: Adapters

Adapters

- Main purpose of functions f_{ϕ_i} added to a pre-trained model is to adapt it
- Functions are also known as "adapters" [Houlsby et al., 2019]
- Design of adapters is model-specific

Adapter Functions

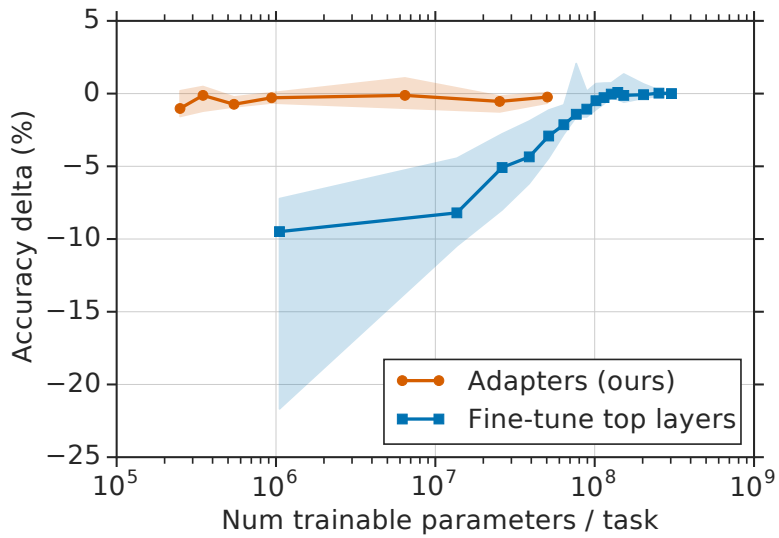
Adapters are small blocks inserted into the network to **adapt pretrained models to specific tasks without modifying the original model weights extensively**

Parameter-Efficient Transfer Learning for NLP

Neil Houlsby [1] **Andrei Giurgiu** [1 *] **Stanisław Jastrzębski** [2 *] **Bruna Morrone** [1] **Quentin de Laroussilhe** [1]
Andrea Gesmundo [1] **Mona Attariyan** [1] **Sylvain Gelly** [1]

Abstract

Fine-tuning large pre-trained models is an effective transfer mechanism in NLP. However, in the presence of many downstream tasks, fine-tuning is parameter inefficient: an entire new model is required for every task. As an alternative, we propose transfer with adapter modules. Adapter modules yield a compact and extensible model; they add only a few trainable parameters per task, and new tasks can be added without revisiting previous ones. The parameters of the original network remain fixed, yielding a high degree of

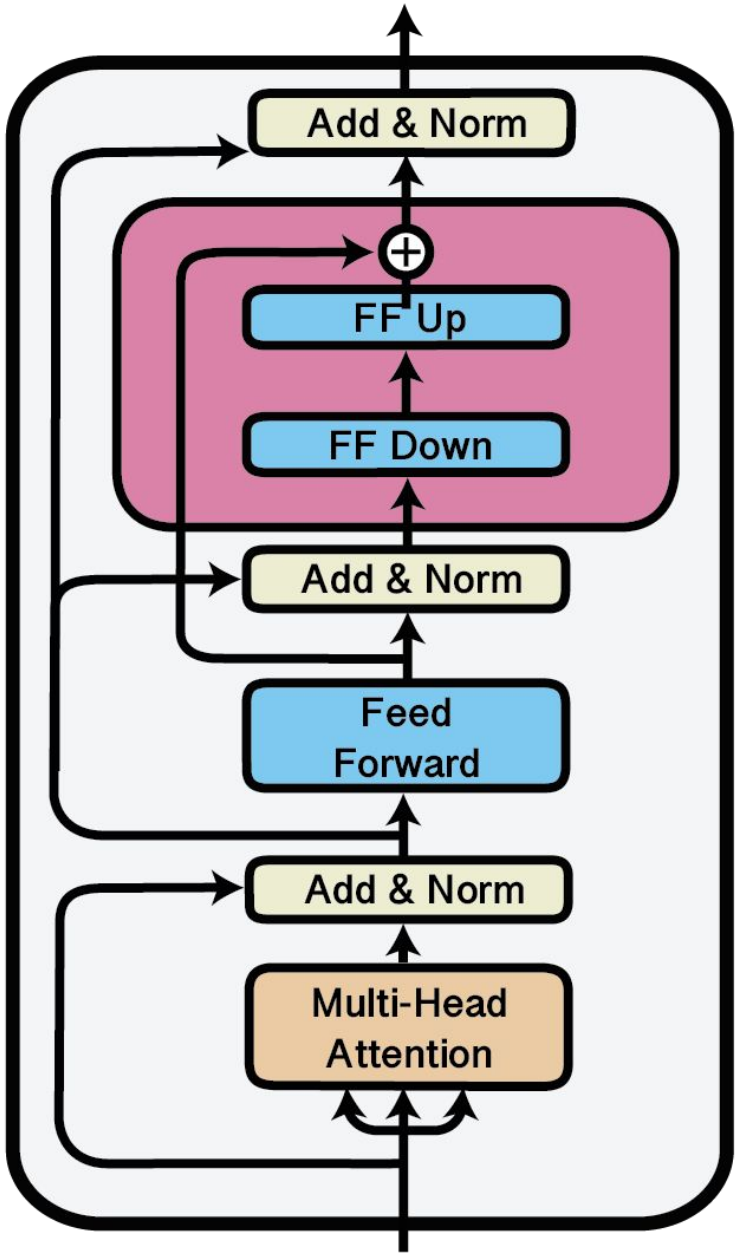


Adapters in Transformer Models

- An adapter in a Transformer layer typically consists of a feed-forward down-projection $W^D \in \mathbb{R}^{d \times r}$, a feed-forward up-projection $W^U \in \mathbb{R}^{r \times d}$ and an activation function σ:

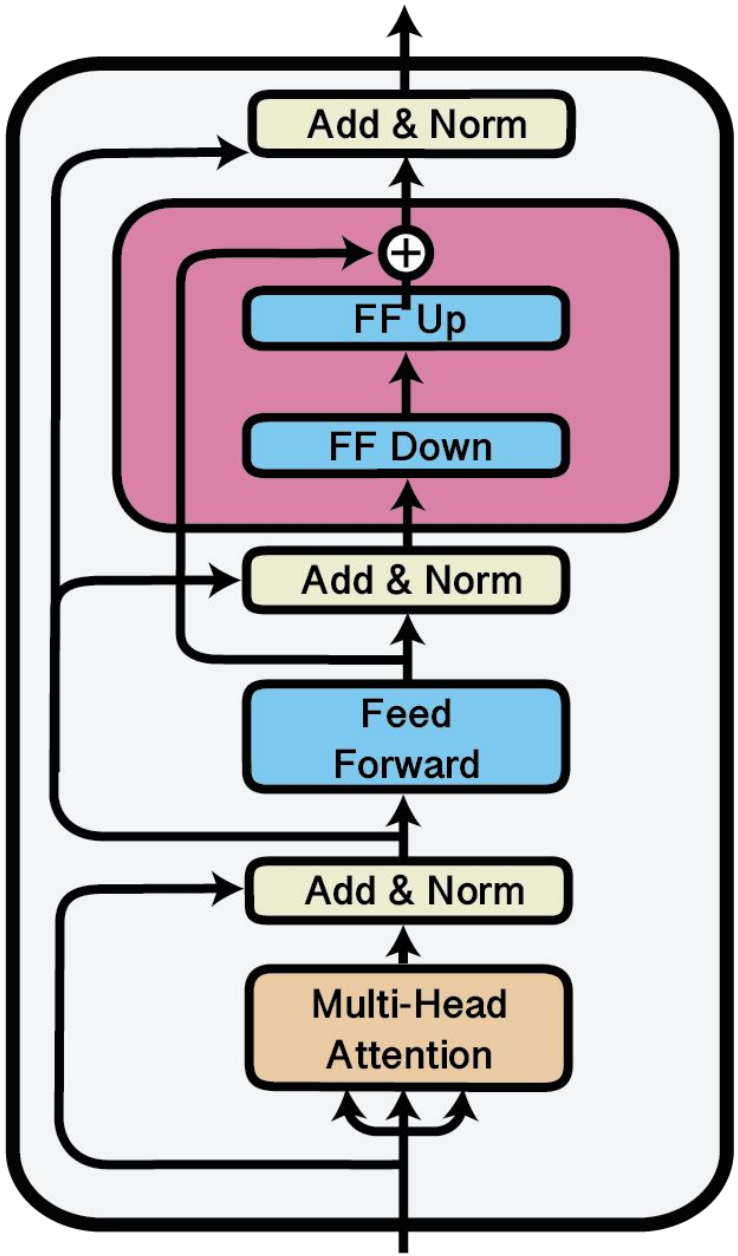
$$f_{\phi_i}(\boldsymbol{x}) = W^D \left(\sigma \left(W^U \boldsymbol{x} \right) \right)$$

- σ is commonly a ReLU



Adapters in Transformer Models

- The adapter is usually placed after the multi-head attention and/or after the feed-forward layer
- Most approaches have used this bottleneck design with linear layers

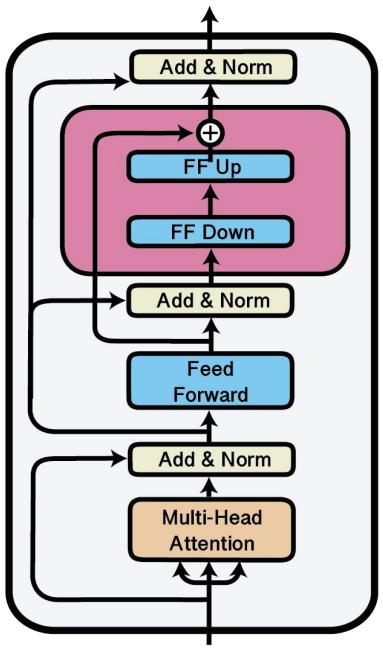


Sequential vs Parallel Adapters

Adapters can be routed sequentially or in parallel

Sequential adapters are inserted between functions:

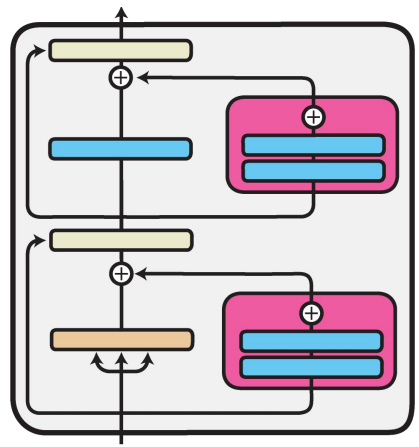
$$f_i'(\boldsymbol{x}) = f_{\phi_i}\left(f_{\theta_i}(\boldsymbol{x})\right)$$



A sequential adapter [Houlsby et al., 2019]

Parallel adapters are applied in parallel:

$$f_i'(\boldsymbol{x}) = f_{\theta_i}(\boldsymbol{x}) + f_{\phi_i}(\boldsymbol{x})$$



Two parallel adapters [Stickland & Murray, 2019]

Sequential vs Parallel Adapters

Adapters can be routed sequentially or in parallel

Sequential Adapters are inserted between functions:

$$f_i'(\boldsymbol{x}) = f_{\phi_i}\left(f_{\theta_i}(\boldsymbol{x})\right)$$

- More **deeply integrate** into the model
- Can **significantly modify the model's processing**
- Better for tasks requiring substantial changes.

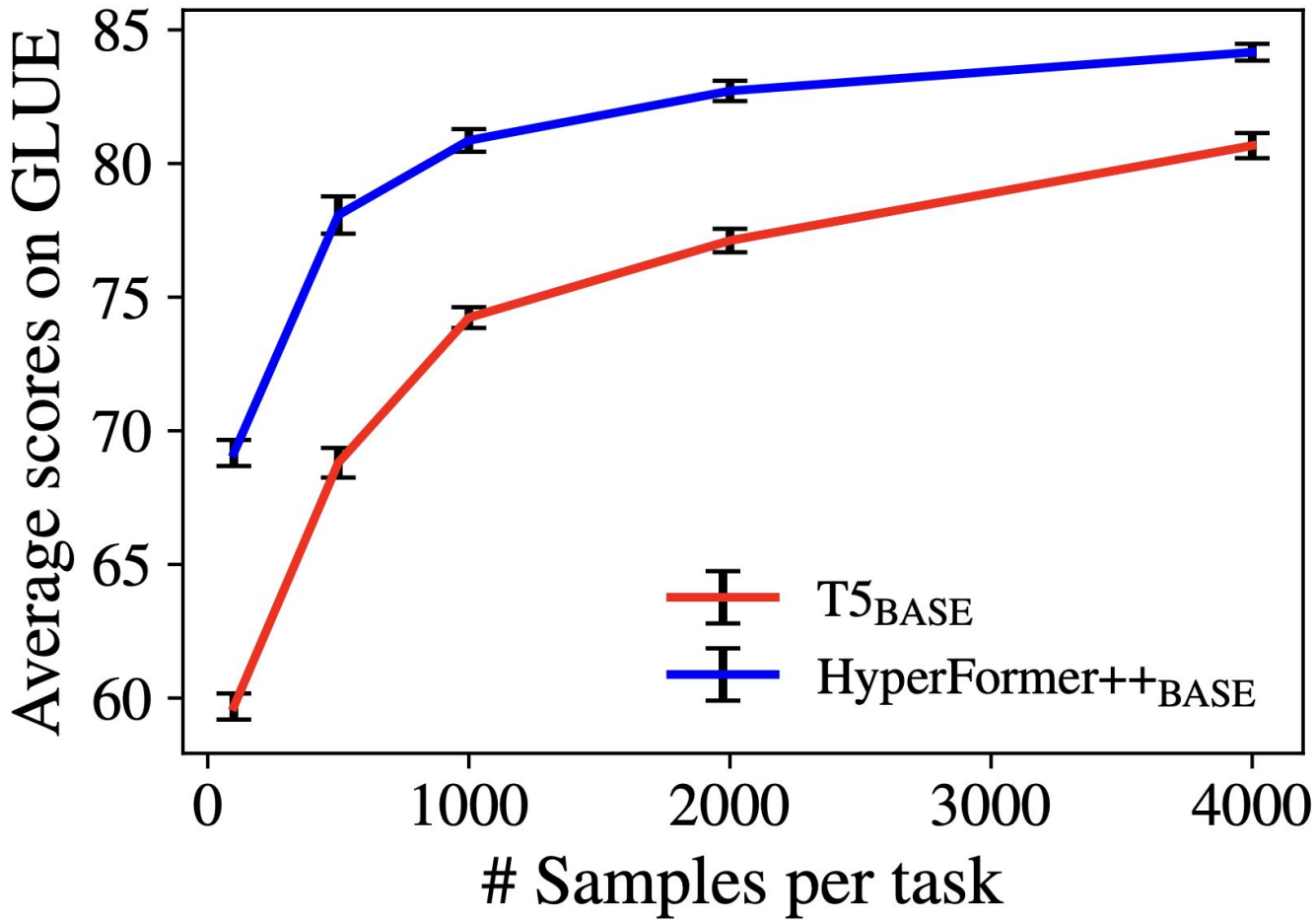
Parallel Adapters are applied in parallel:

$$f_i'(\boldsymbol{x}) = f_{\theta_i}(\boldsymbol{x}) + f_{\phi_i}(\boldsymbol{x})$$

- **Less intrusive** to the model's architecture
- Maintain the model's original behavior more closely
- Suitable for tasks requiring only **minor adjustments**

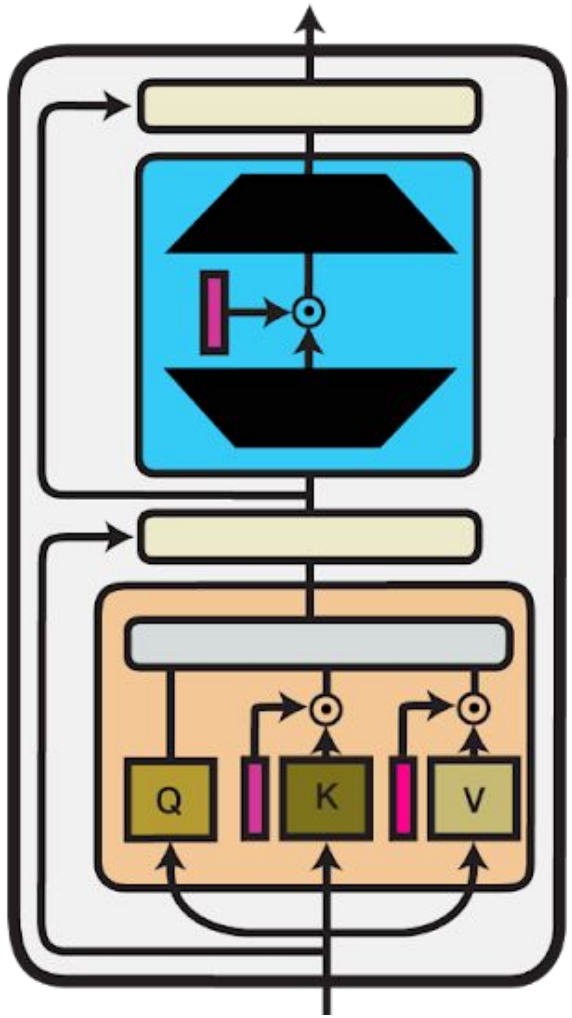
Benefits of Adapters

- Increased sample efficiency
- Results on GLUE with different numbers of training samples per task [Mahabadi et al., 2021]



Rescaling

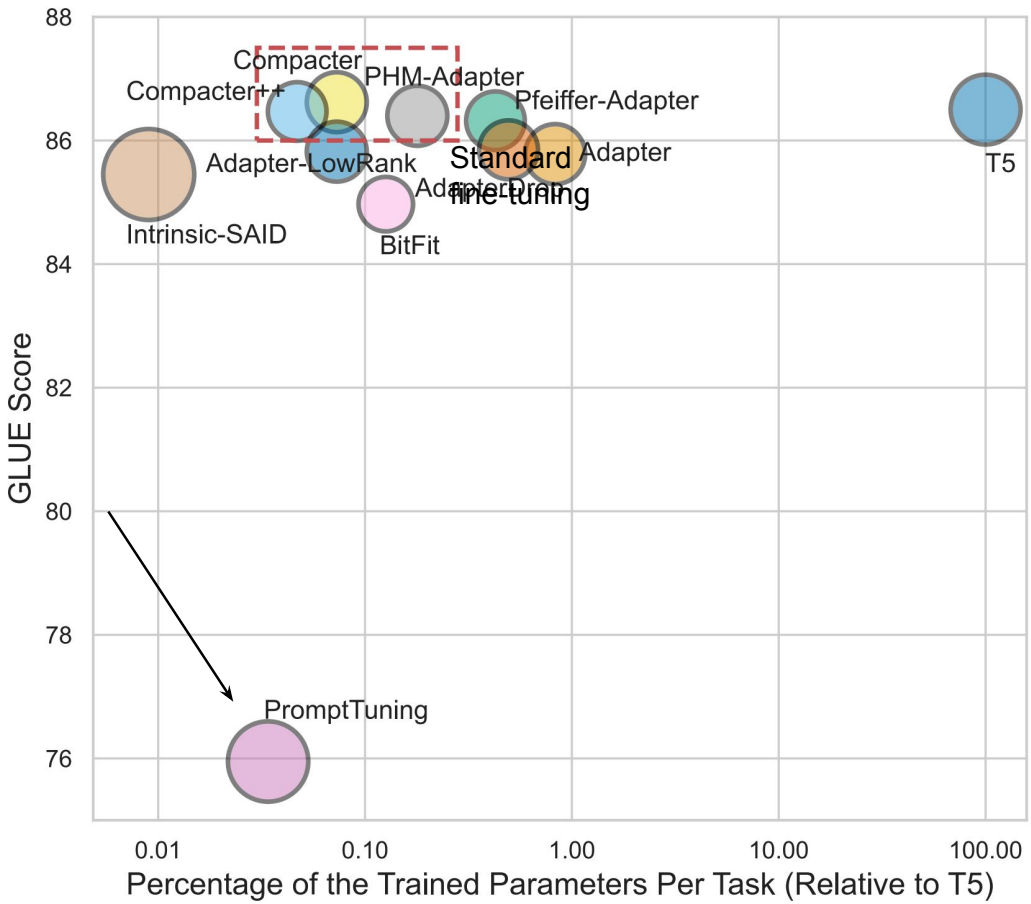
- Instead of learning a function, even rescaling via element-wise multiplication can be effective: $f_i'(\boldsymbol{x}) = f_{\theta_i}(\boldsymbol{x}) \circ \phi_i$
- Allows the model to select parameters that are more and less important for a qiven task
- IA^3[Liu et al., 2022] multiplies **learned vectors** with the **keys** and **values** in self-attention and the intermediate activations in the feed-forward layer
- Compatible with other methods, e.g., LoRA, which has a tunable scalar parameter
  - $W_{\text{adapted}} = W_0 + \alpha \cdot (\mathbf{A}_{\text{down}}\ \mathbf{B}_{\text{up}})$

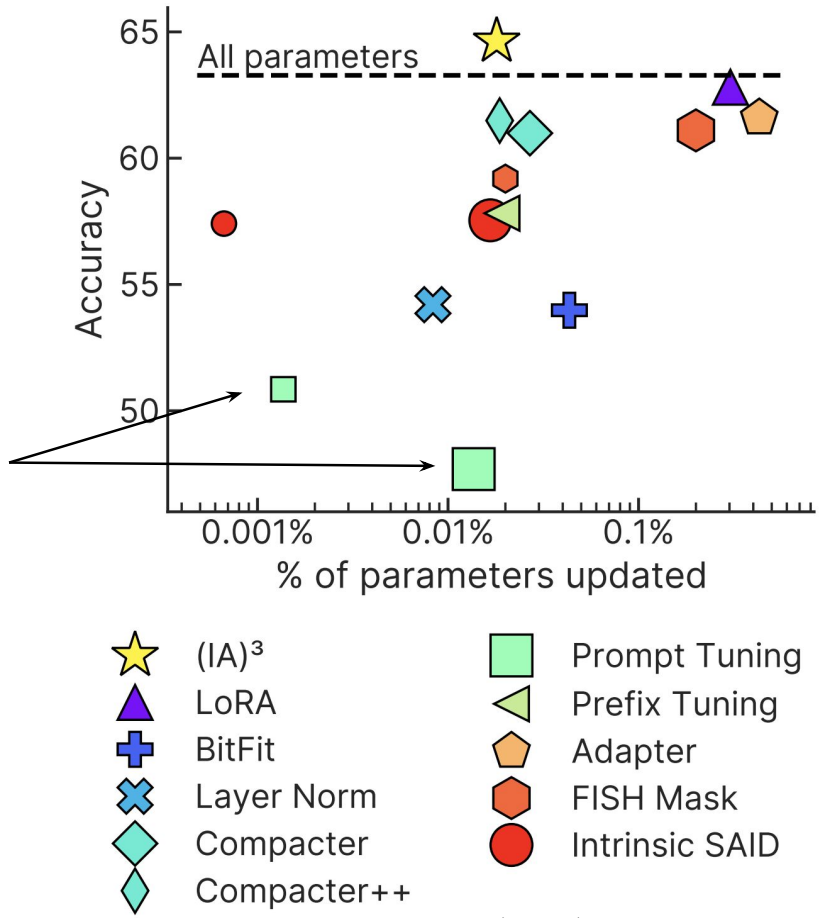


IA^3 [Liu et al., 2022] in the Transformer model

Performance Comparison

Prompt tuning underperform the other methods due to limited capacity

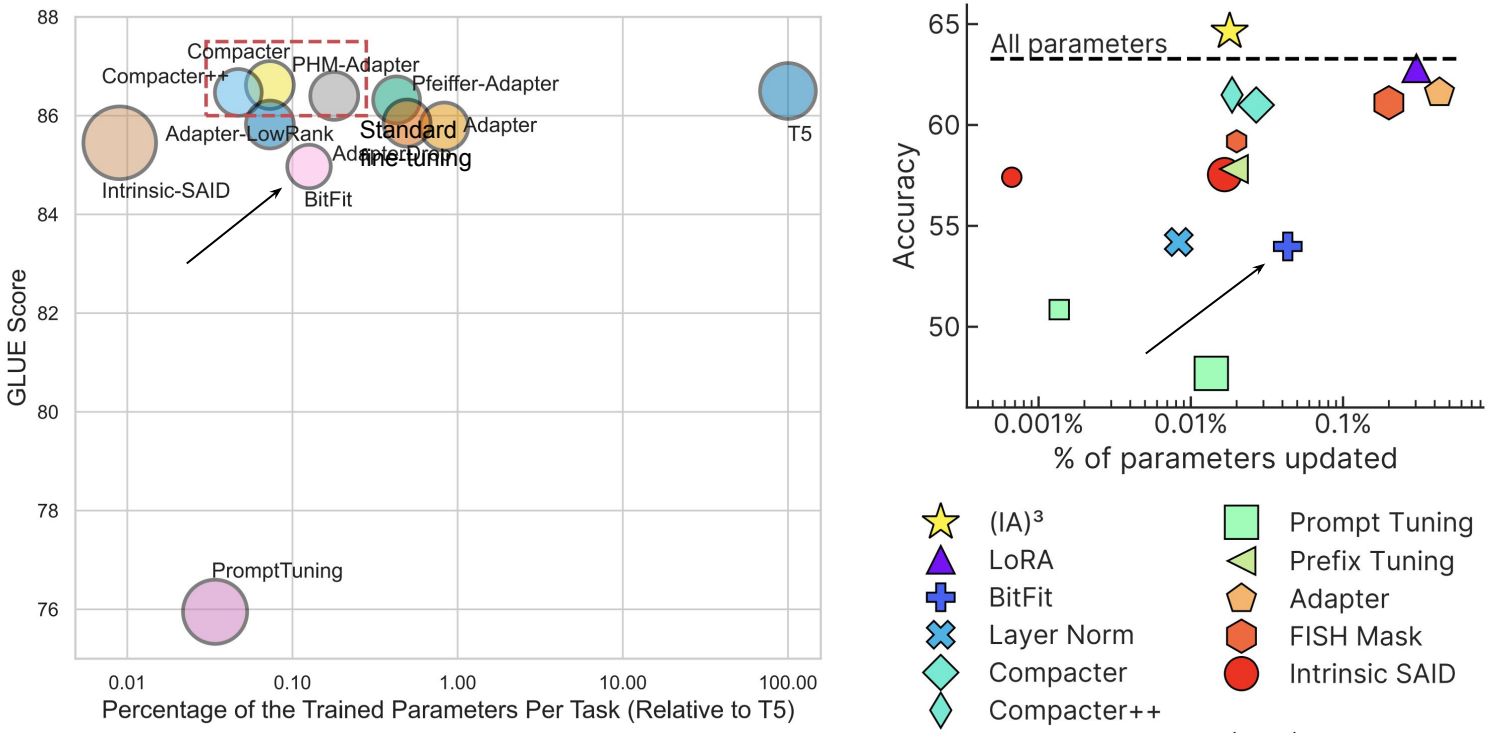




Average performance, % of trained parameters per task, and (left) memory footprint (circle size) of different methods on T5-Base (222B parameters; left) [Mahabadi et al., 2021] and T3-3B (right) [Liu et al., 2022]

Performance Comparison

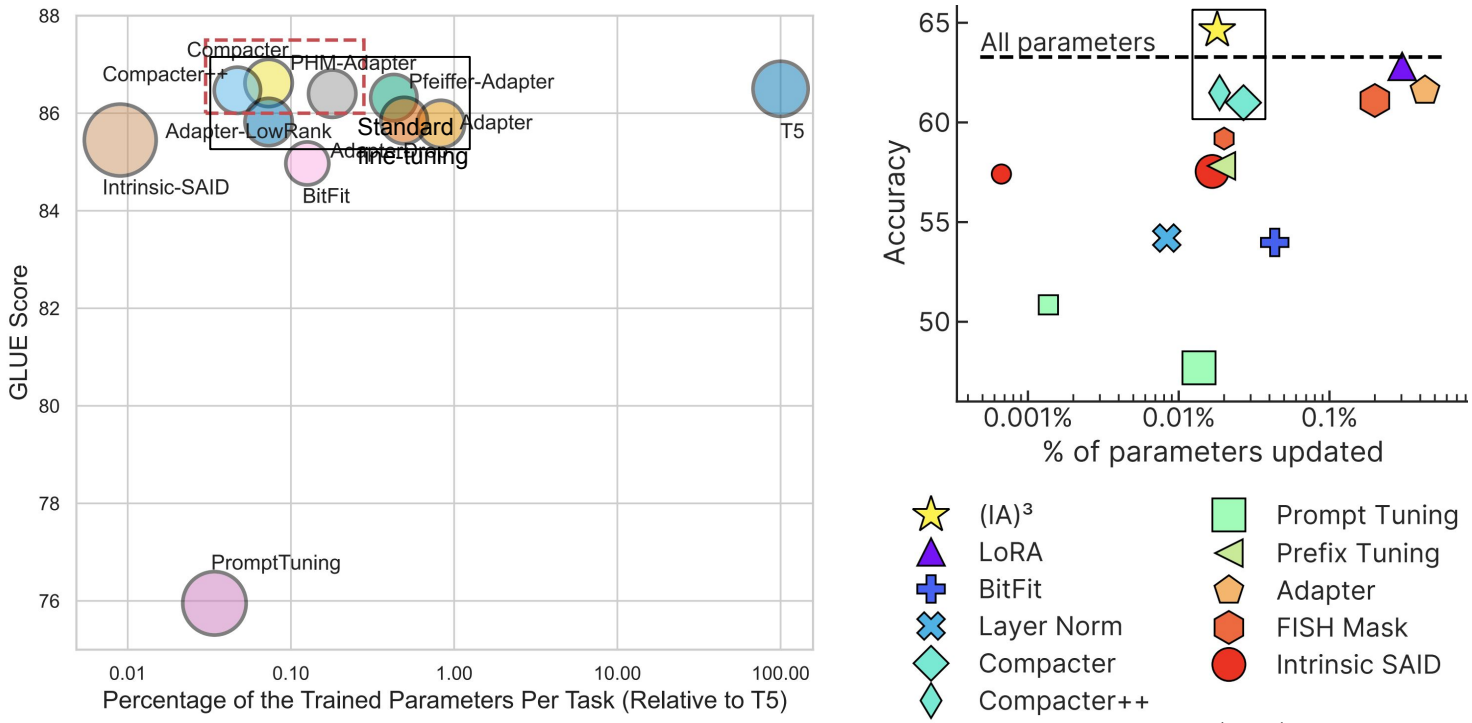
Fine-tuning biases (BitFit) only has a small memory footprint but achieves lower performance



Average performance, % of trained parameters per task, and (left) memory footprint (circle size) of different methods on T5-Base (222B parameters; left) [Mahabadi et al., 2021] and T3-3B (right) [Liu et al., 2022]

Performance Comparison

Function composition methods such as adapter, compacter, and IA^3 achieve the best performance but add more parameters



Average performance, % of trained parameters per task, and (left) memory footprint (circle size) of different methods on T5-Base (222B parameters; left) [Mahabadi et al., 2021] and T3-3B (right) [Liu et al., 2022]

PEFT Summary

- Many ways to do PEFT: sparse fine-tuning, BitFit, LoRA, Prompt Tuning, Adapters, etc.
- There is no 'one-size-fits-all' design and model choice
- Understand your task and your data
- Understand your constraints and requirements:
  - **Parameter** budget?
  - **Compute** budget?
  - **Storage** budget?
  - **Training** speed?
  - **Inference** speed?

That's all for today

- Some content based on slides by Sebastian Ruder and others