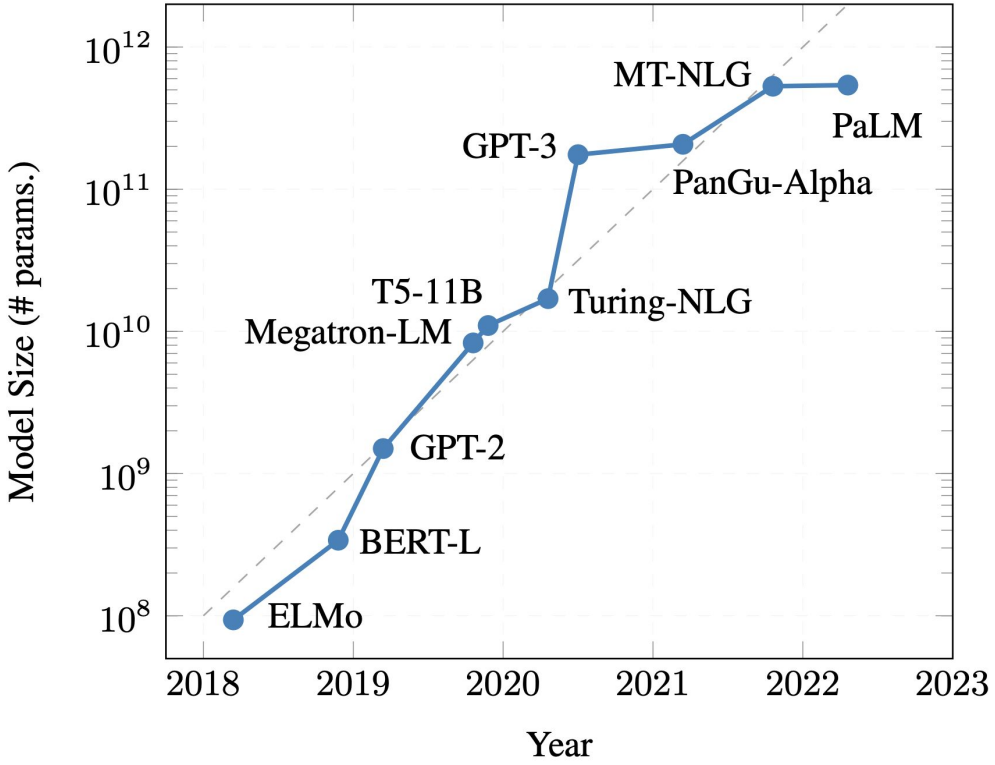
Lecture 13. PEFT.

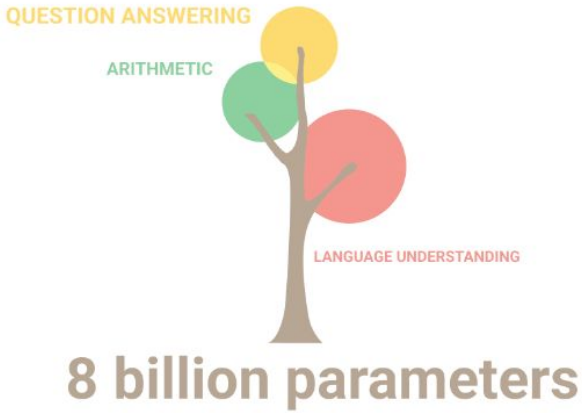
Nikolay
Karpachev
6.05.2024

State-of-the-art NLP Models Are Getting Ever Larger



Evolution of the size of large pre-trained models [Treviso et al., 2022]

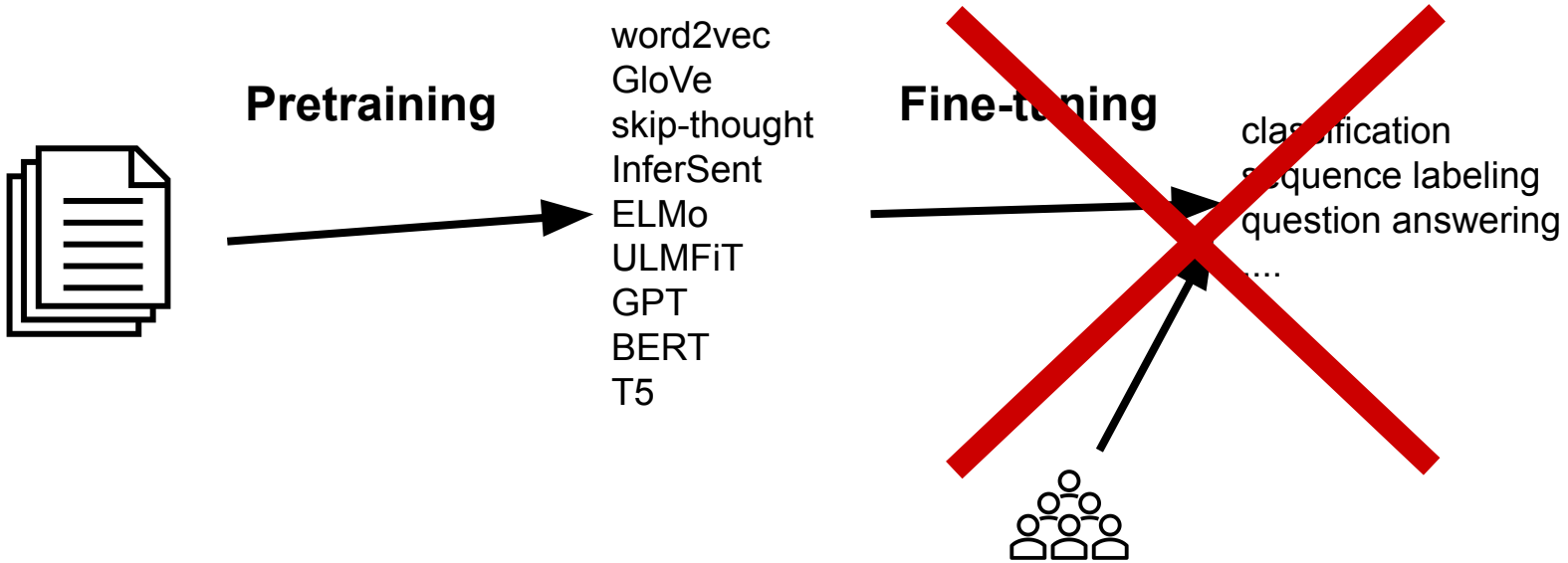
Large Models Develop Increasing NLU Capabilities



Credit: Google AI Blog

Transfer Learning in the Era of Large Models

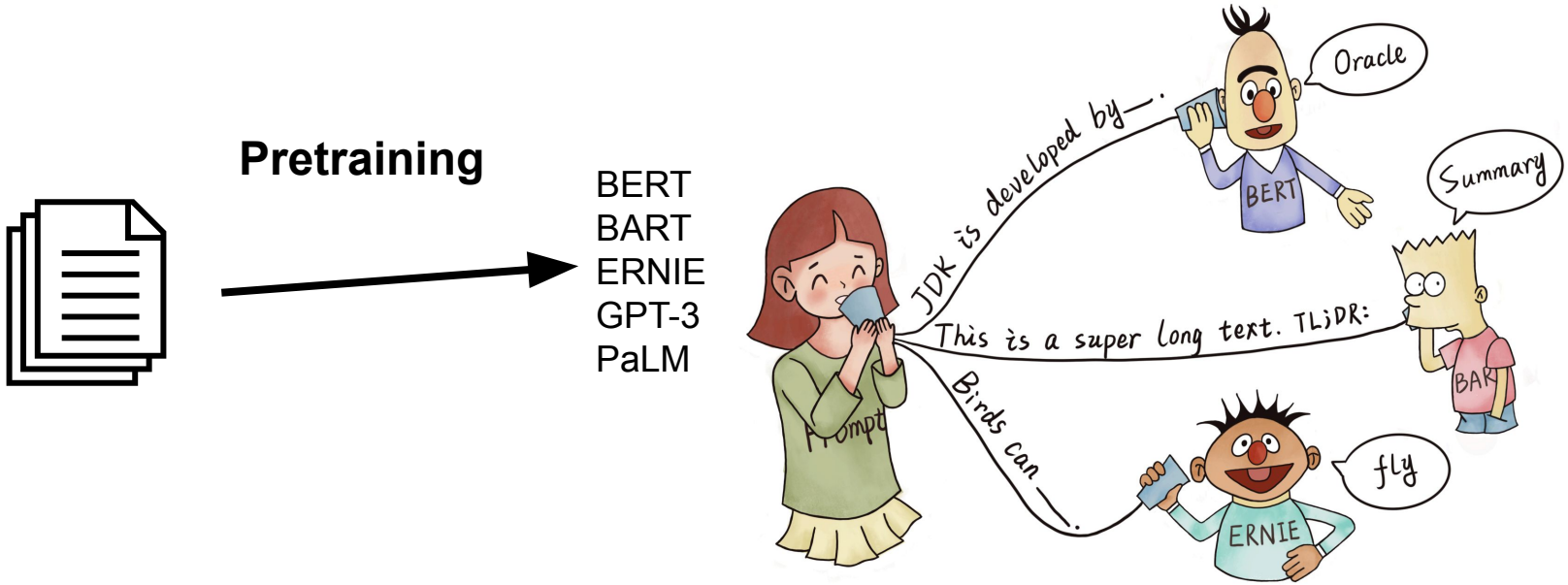
- With increasing model size, fine-tuning becomes increasingly expensive
- The standard transfer learning formula breaks down



Credit: NAACL 2019 Transfer learning tutorial

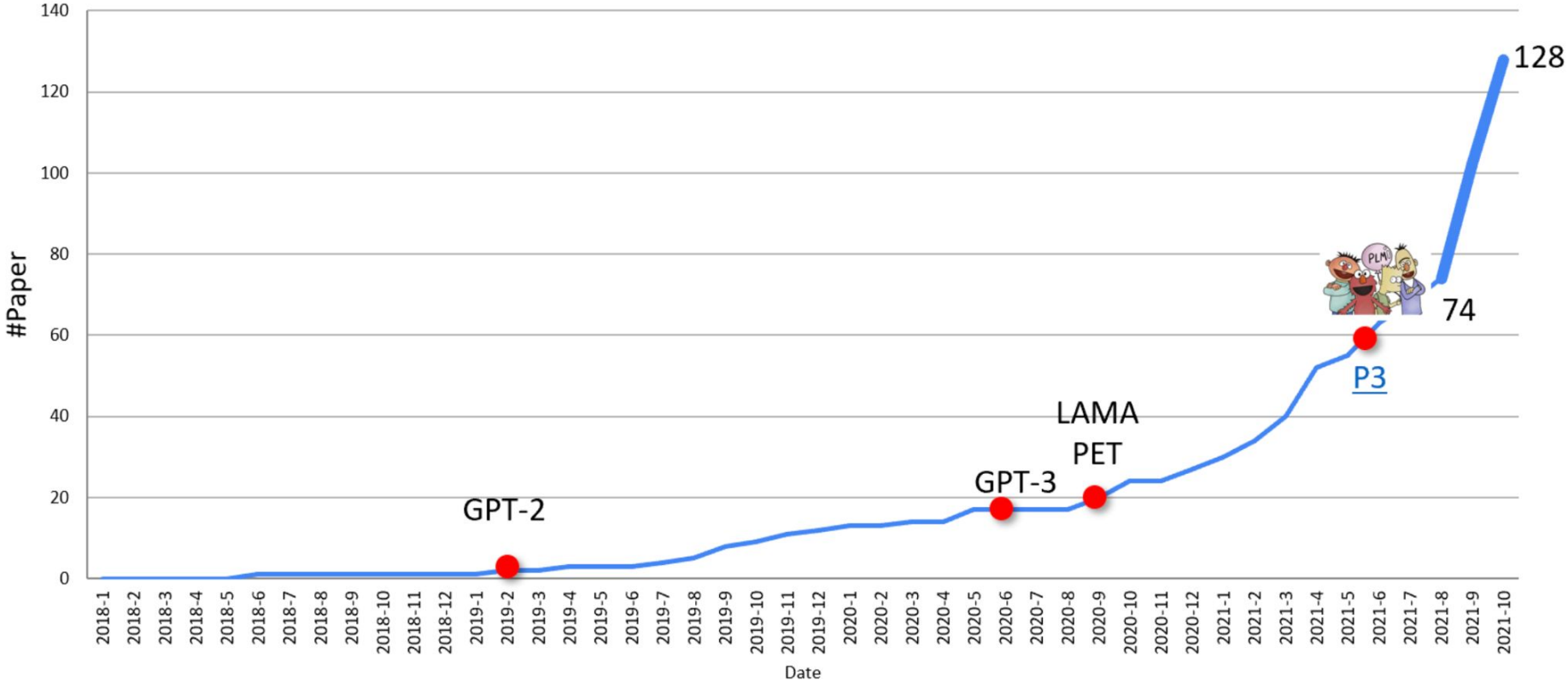
Transfer Learning in the Era of Large Models

- In-context learning has mostly replaced fine-tuning for large models



Credit: Liu et al. (2021)

Prompt-based Learning has Taken NLP by Storm



Credit: http://pretrain.nlpedia.ai/

Downsides of Prompt-based Learning

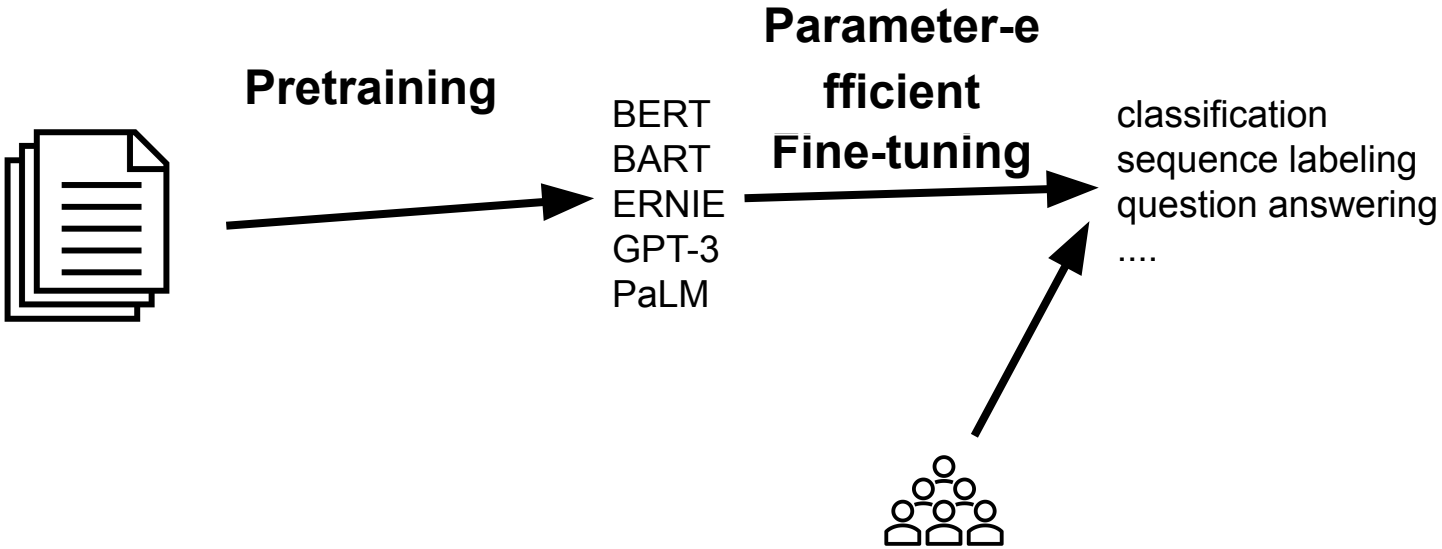
1. **Inefficiency:** The prompt needs to be processed *every time* the model makes a prediction.

2. **Poor performance**: Prompting generally performs worse than fine-tuning [Brown et al., 2020].

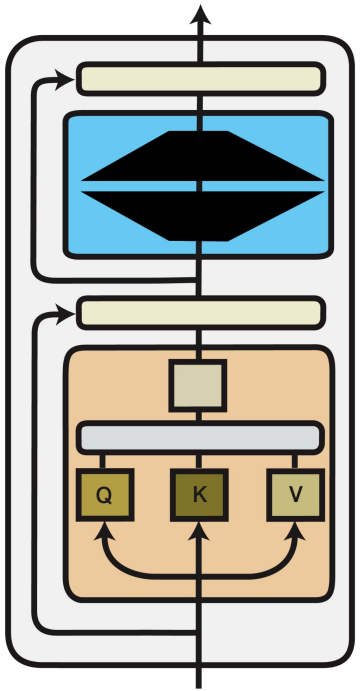
3. **Sensitivity** to the wording of the prompt [Webson & Pavlick, 2022], order of examples [Zhao et al., 2021; Lu et al., 2022], etc.

4. **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

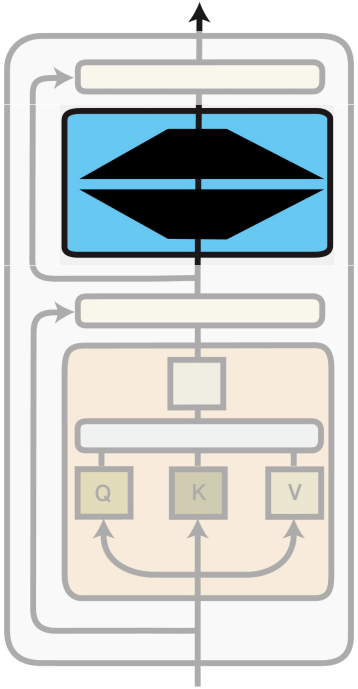


From Fine-tuning to Parameter-efficient Fine-tuning



Full Fine-tuning

Update **all model parameters**



Parameter-efficient Fine-tuning

Update a **small subset** of model parameters

Haven't We Seen This Before?

- Updating the last layer was common in computer vision [Donahue et al., 2014]. 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].

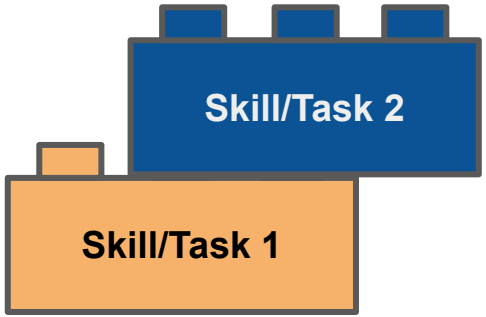
→ Fine-tuning all representations performed generally better in practice

Why go back to fine-tuning *only some* parameters?

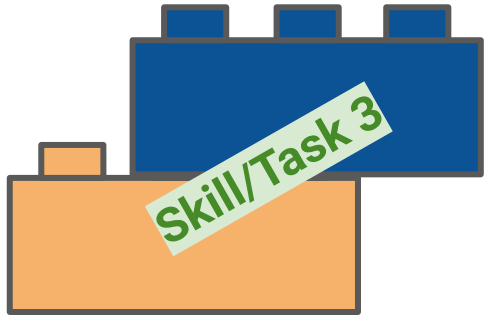
1. Fine-tuning all parameters is impractical with large models
2. State-of-the-art models are massively over-parameterized
   → Parameter-efficient fine-tuning matches performance of full fine-tuning
3. **Modular and compositional** representations

Modularity and Compositionality?

Modularity and Compositionality?

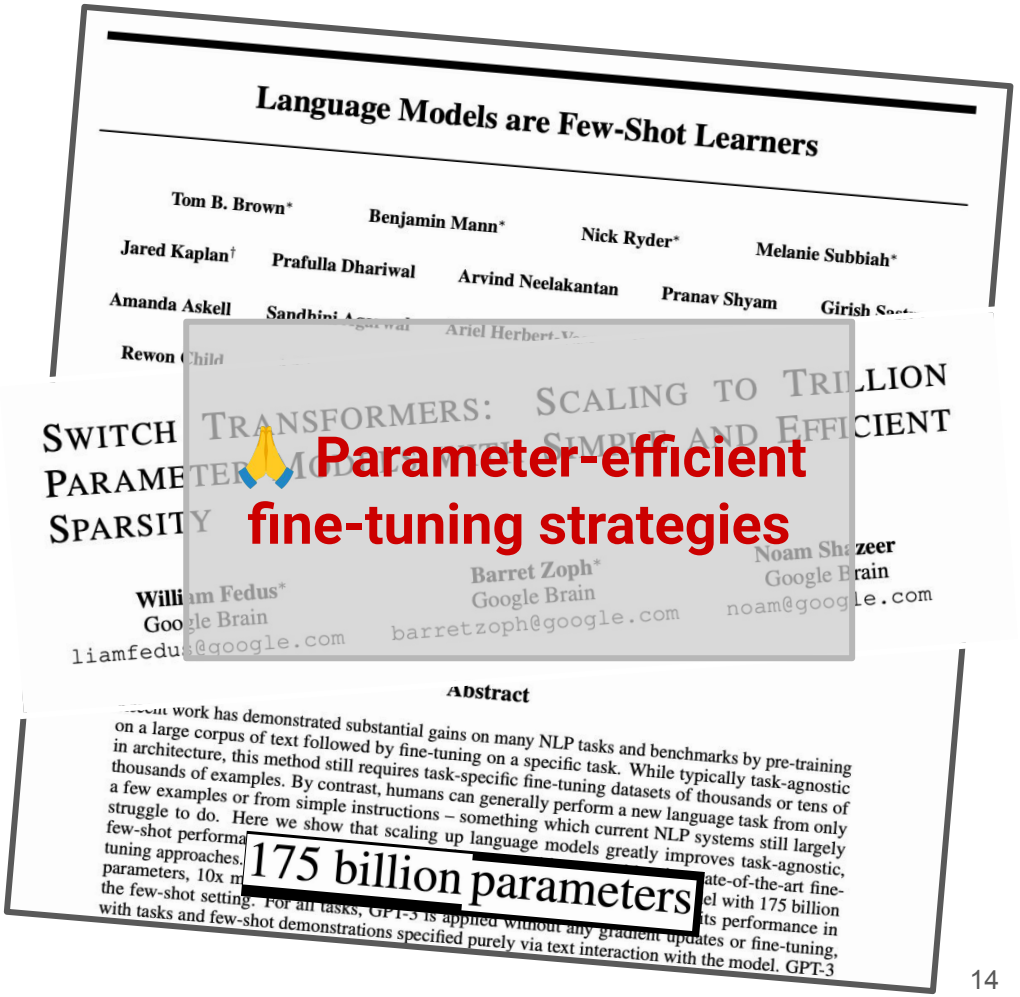


Modularity and Compositionality?



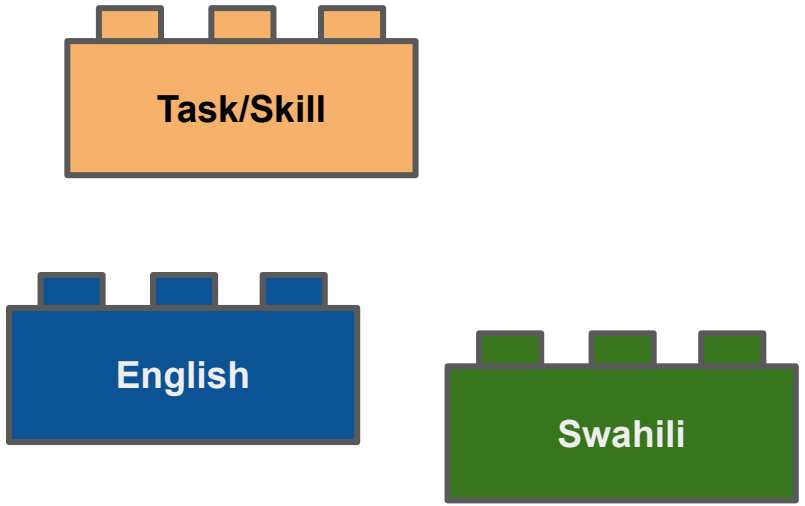
Why Modularity?

1. Models are **increasing** in **size**



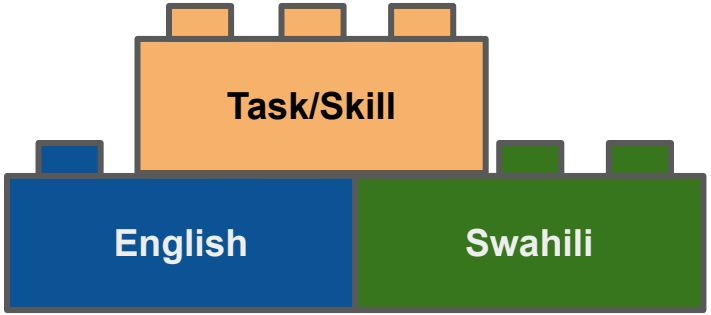
Why Modularity?

1. Models are **increasing** in **size**
→ Parameter-efficient fine-tuning strategies
2. **Unseen** Scenarios



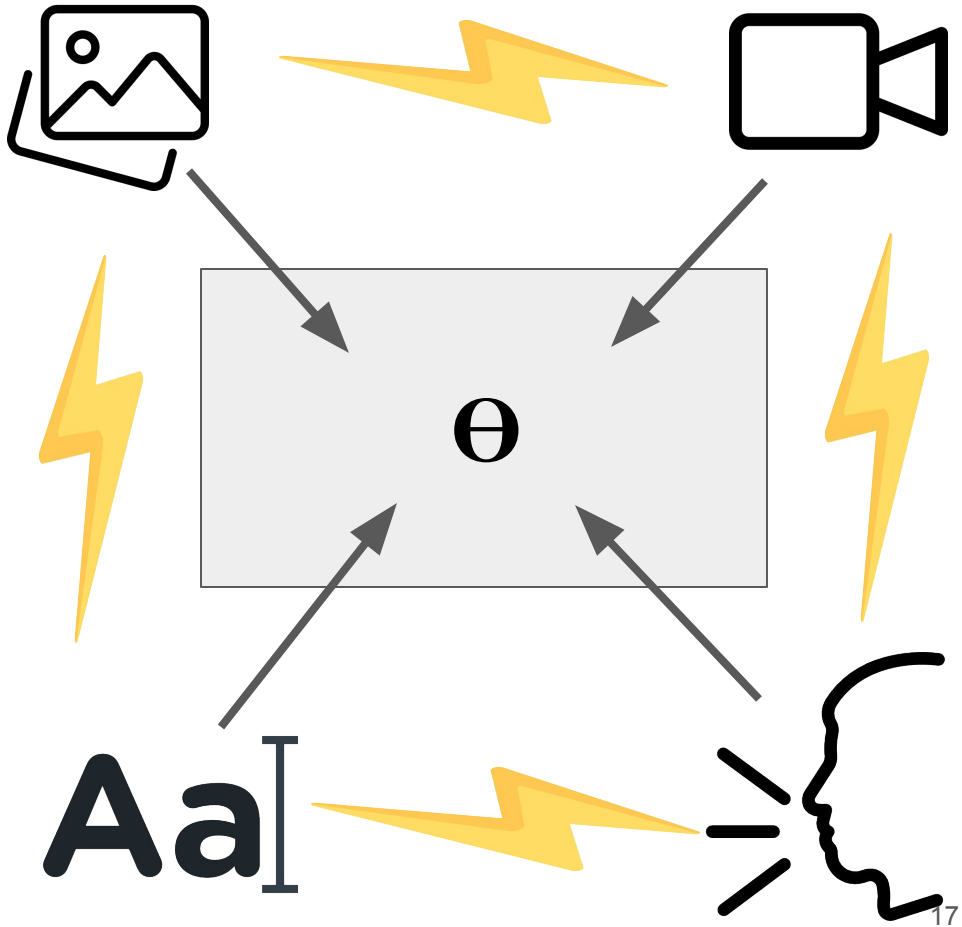
Why Modularity?

1. Models are **increasing** in **size**
→ Parameter-efficient fine-tuning strategies
2. **Unseen** Scenarios
→ Out-of-distribution generalization through **module composition**



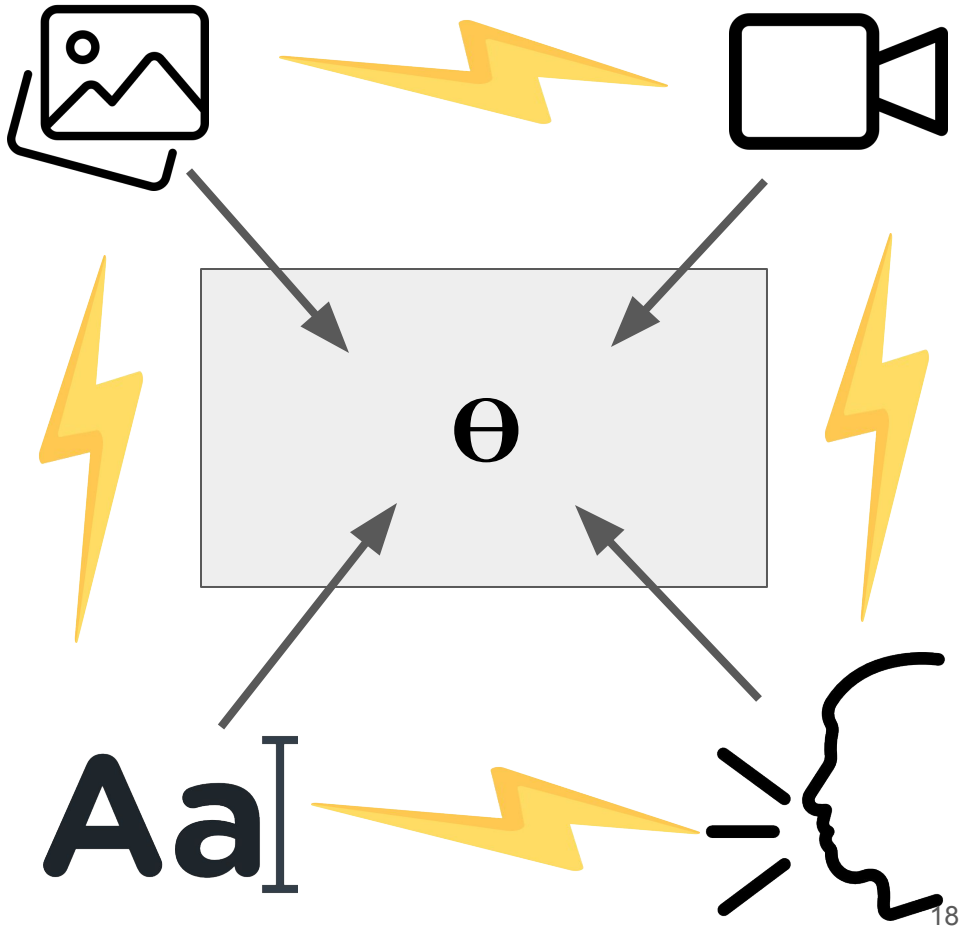
Why Modularity?

1. Models are **increasing** in **size**
→ Parameter-efficient fine-tuning strategies
2. **Unseen** Scenarios
→ Out-of-distribution generalization through **module composition**
3. **Catastrophic interference**



Why Modularity?

1. Models are **increasing** in **size**
→ Parameter-efficient fine-tuning strategies
2. **Unseen** Scenarios
→ Out-of-distribution generalization through **module composition**
3. **Catastrophic interference**
→ Modularity as inductive bias



Notation

Let a neural network $f_\theta : \mathcal{X} \to \mathcal{Y}$ be decomposed into a composition of functions $f_{\theta_1} \odot f_{\theta_2} \odot \cdots \odot f_{\theta_l}$. Each has parameters $\theta_i, i = 1, \ldots, l$.

Function composition

A module with parameters ϕ can modify the *i*-th subfunction as follows:

1. Parameter composition: $f'_i(\boldsymbol{x}) = f_{\theta_i \oplus \phi}(\boldsymbol{x})$

   Interpolation, e.g., element-wise addition

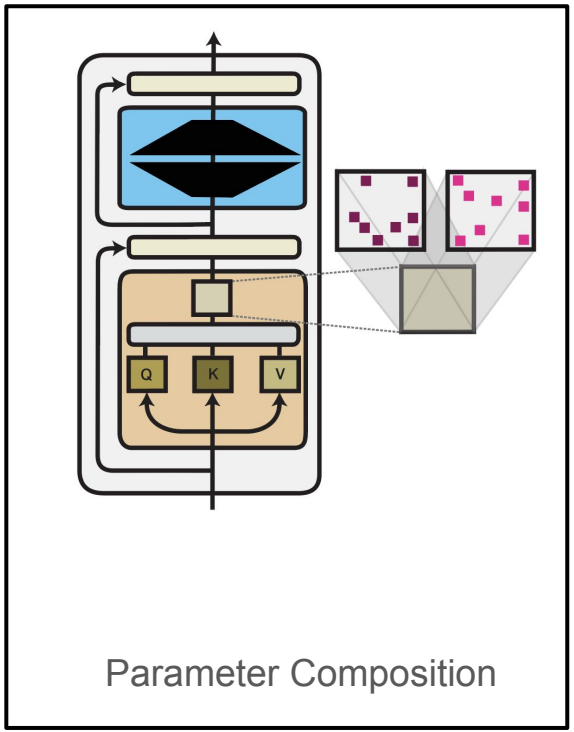
2. Input composition: $f'_i(\boldsymbol{x}) = f_{\theta_i}([\boldsymbol{x}, \phi])$

   Concatenation

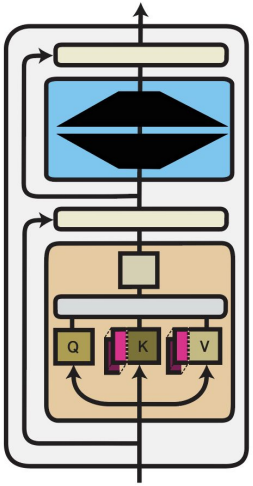
3. Function composition: $f'_i(\boldsymbol{x}) = f_{\theta_i} \odot f_\phi(\boldsymbol{x})$

In practice, typically only the module parameters ϕ are updated while θ is fixed.

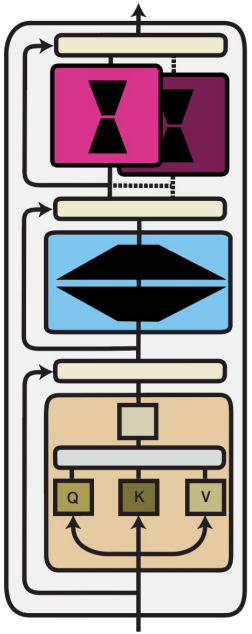
Three Computation Functions



Parameter Composition



Input Composition



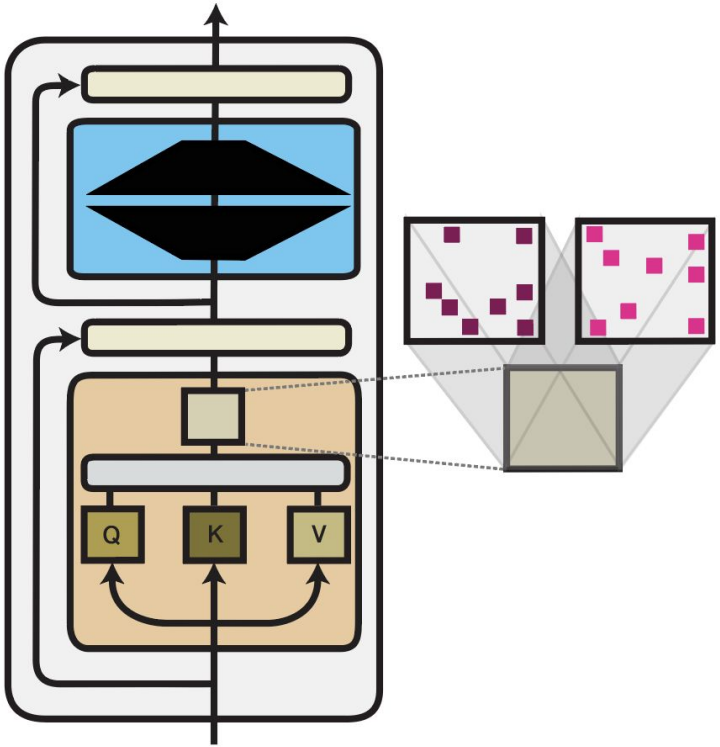
Function Composition

Parameter Composition

1. Sparse Subnetworks

2. Structured Composition

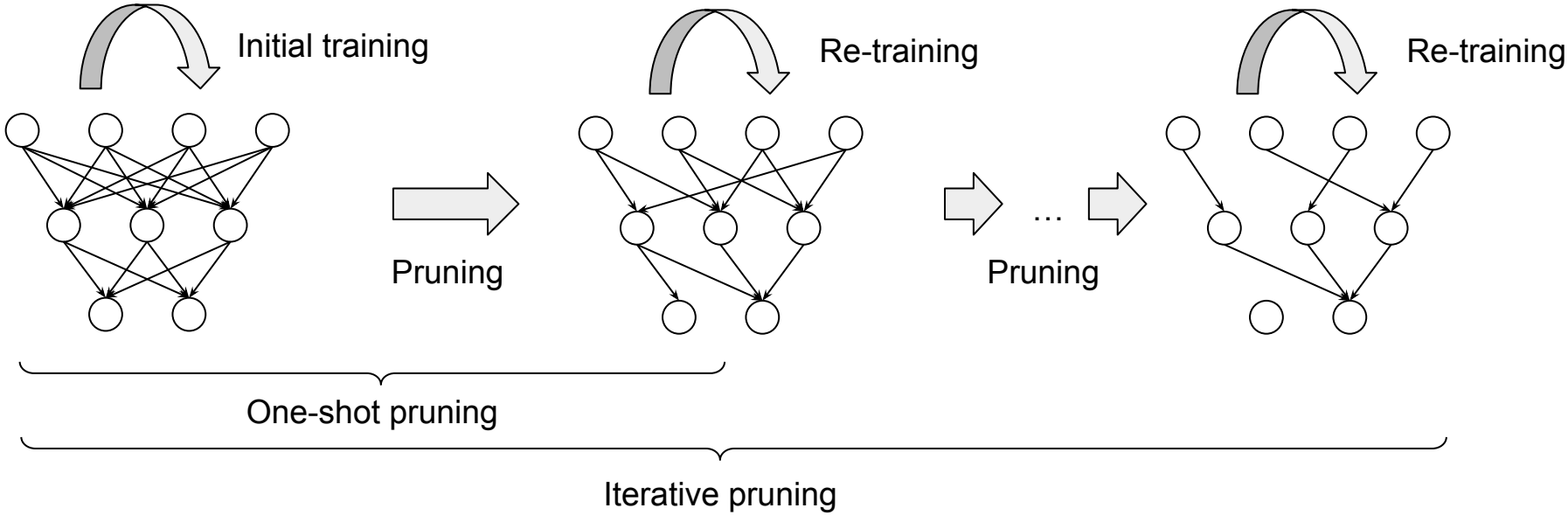
3. Low-rank Composition



Sparse Subnetworks

- A common inductive bias on the module parameters ϕ is **sparsity**
- 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



- During pruning, a fraction of the lowest-magnitude weights are removed
- The non-pruned weights are re-trained
- Pruning for multiple iterations is more common [Frankle & Carbin, 2019]

Another Perspective on Pruning

- We can also view pruning as adding a task-specific vector ϕ to the parameters of an existing model: $f'_\theta = f_{\theta+\phi}$ where $\phi_i = 0$ if $b_i = 0$.

- If the final model should be sparse, we can multiply the existing weights with the binary mask to set the pruned weights to 0: $f'_\theta = f_{\theta \circ \boldsymbol{b}+\phi}$

Element-wise product (Hadamard product)

- The main benefit is that these weight values were moving to 0 anyway [Zhou et al., 2019]

The Lottery Ticket Hypothesis

- Dense, randomly-initialized models **contain subnetworks** (“winning tickets”) that—when trained in isolation—**reach test accuracy comparable to the original network** in a similar number of iterations [Frankle & Carbin, 2019]

- Has also been verified in RL and NLP [Yu et al., 2020] and for larger models in computer vision [Frankle et al., 2020]

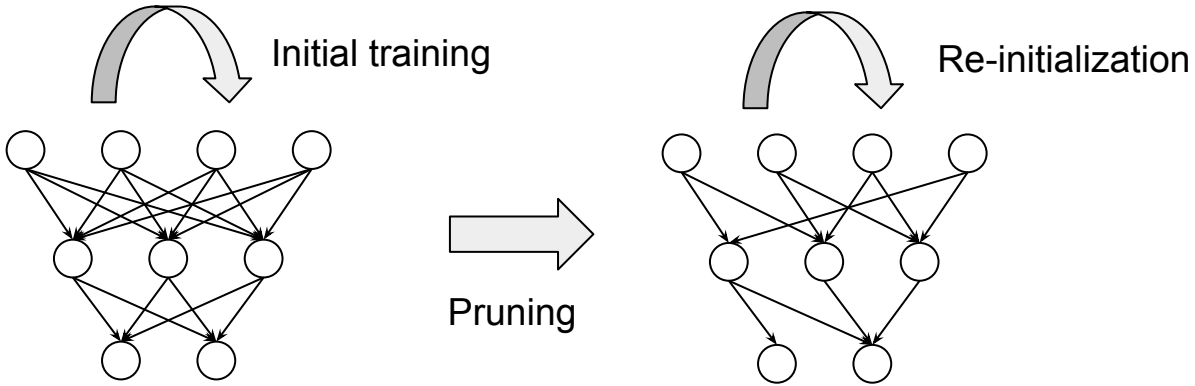
Beyond Lottery Tickets: Supermasks

- The number of possible subnetworks grows combinatorially with the number of model parameters

→ There are masks—*supermasks*—that achieve non-random performance even for randomly initialized, fixed models [Zhou et al., 2019]

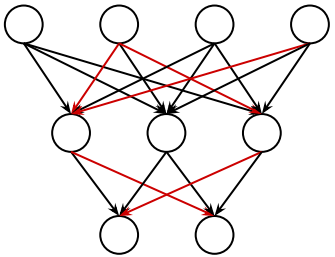
- In this case, the module parameters only consist of the binary mask:

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

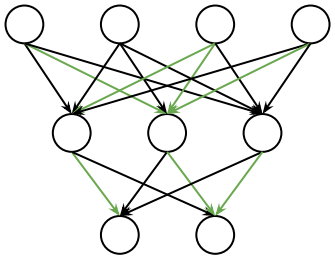


Beyond Lottery Tickets: Supermasks

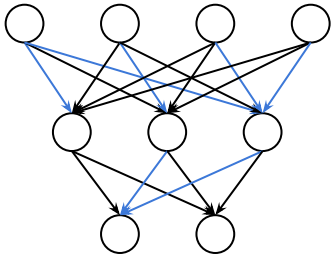
- A fixed model can accommodate a potentially unlimited number of task-specific binary masks [Wortsmann et al., 2020]



Supermask for Task A



Supermask for Task B



Supermask for Task C

Supermasks in Pre-trained Models

- Such supermasks have similarly been found useful in pre-trained models like BERT [Zhao et al., 2020]
- Similar to earlier work [Mallya et al., 2018], they learn a real-valued mask $\boldsymbol{b}^{\text{real}}$ that is then binarized via a thresholding function:

$$\boldsymbol{b}_i = \begin{cases} 1 & \text{if } \boldsymbol{b}_i^{\text{real}} \geq \tau \\ 0 & \text{otherwise} \end{cases}$$

- For back-propagation: $\nabla \boldsymbol{b}$ can be used as a noisy estimate of $\nabla \boldsymbol{b}^{\text{real}}$ (also known as straight-through estimator [Bengio et al., 2013])

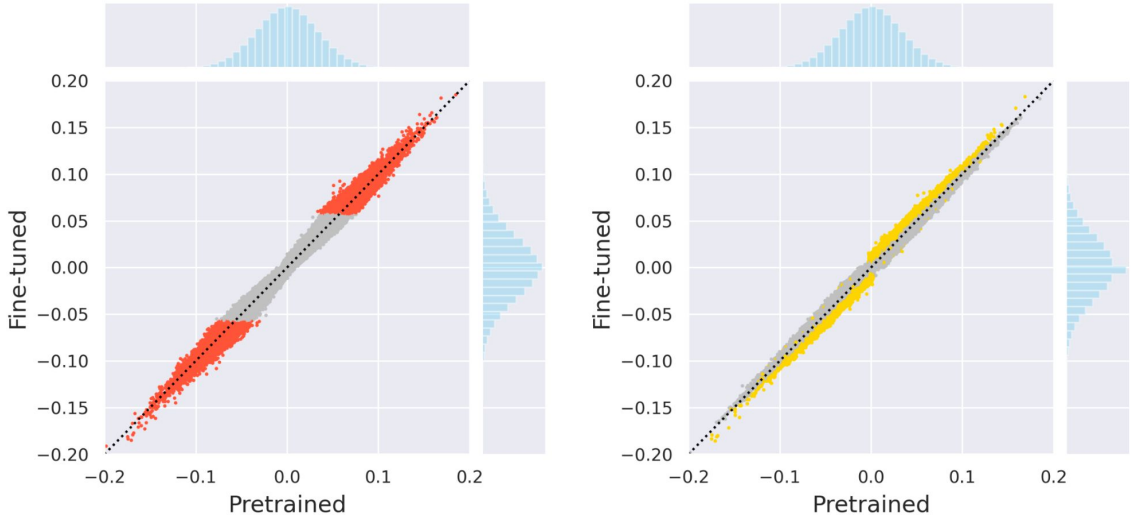
Masking as an Efficient Alternative to Finetuning for Pretrained Language Models | ACL Anthology

- The embedding layer is not masked

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

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.

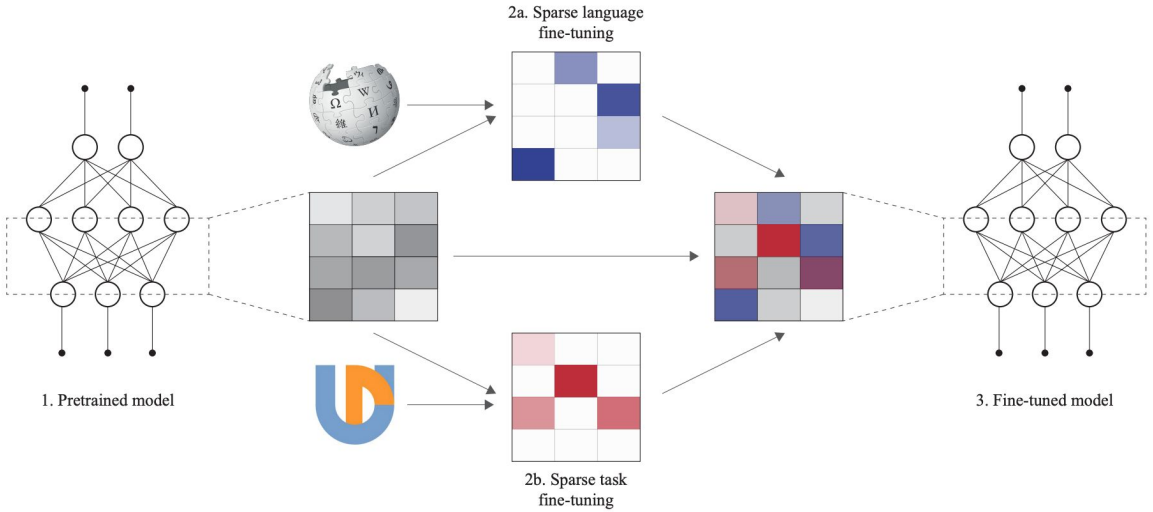
Lottery Ticket Sparse Fine-tuning

- Lottery Ticket Sparse Fine-tuning [Ansell et al., 2022] learns sparse subnetworks based on magnitude pruning of the module parameters ϕ

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

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

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

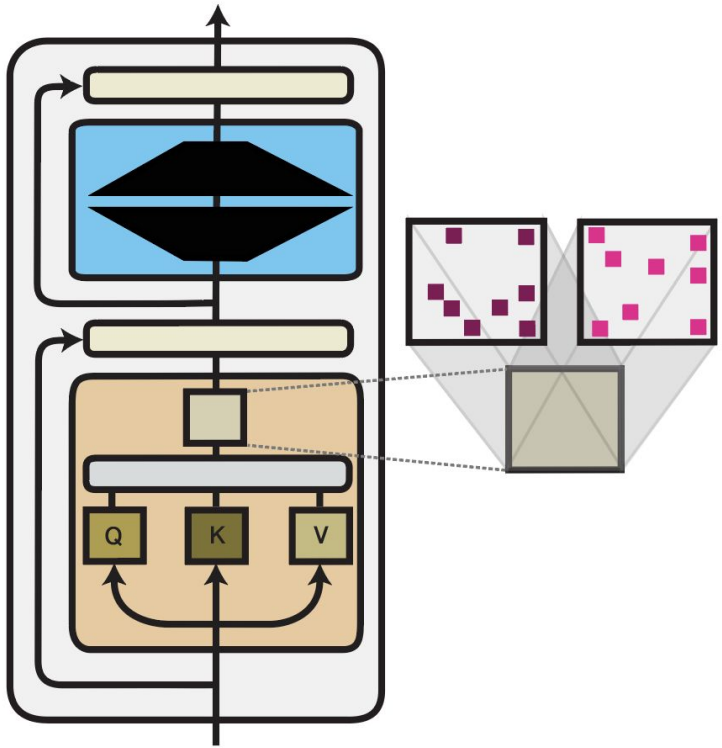


Parameter Composition

1. Sparse Subnetworks

2. Structured Composition

3. Low-rank Composition



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 $\mathcal{G}$: $f_i' = f_{\theta_i + \phi_i} \; \forall f_i' \in \mathcal{G}$

Group-based Fine-tuning

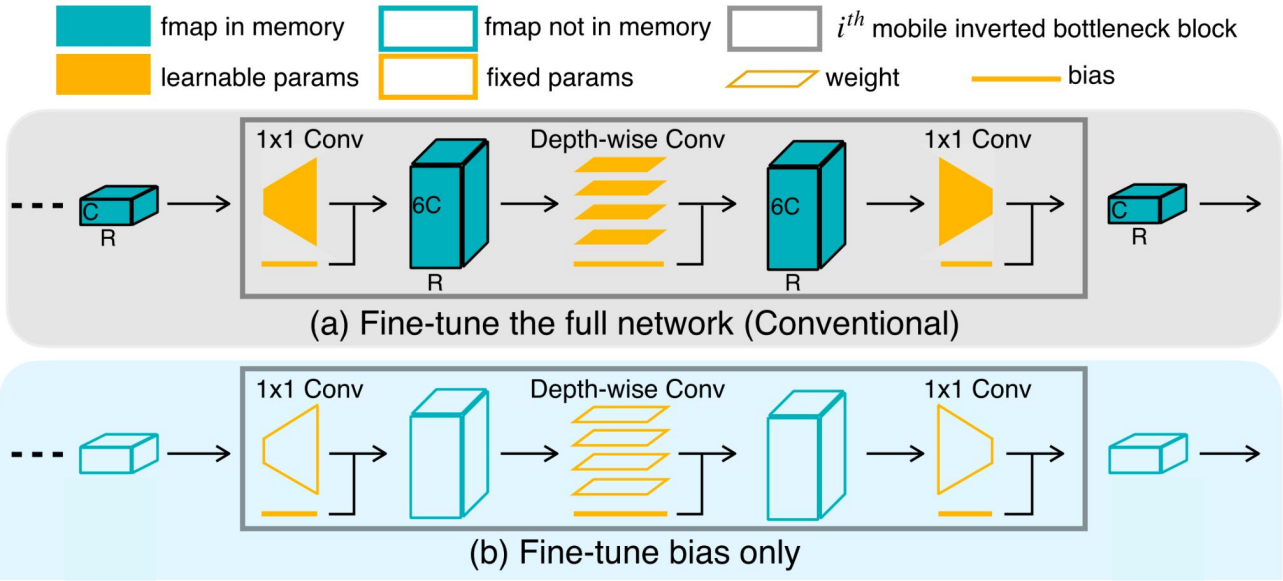
- Most common setting: each group $\mathcal{G}$ corresponds to a layer; only update the parameters associated with certain layers

- Groups can also relate to more fine-grained components

Bias-only Fine-tuning

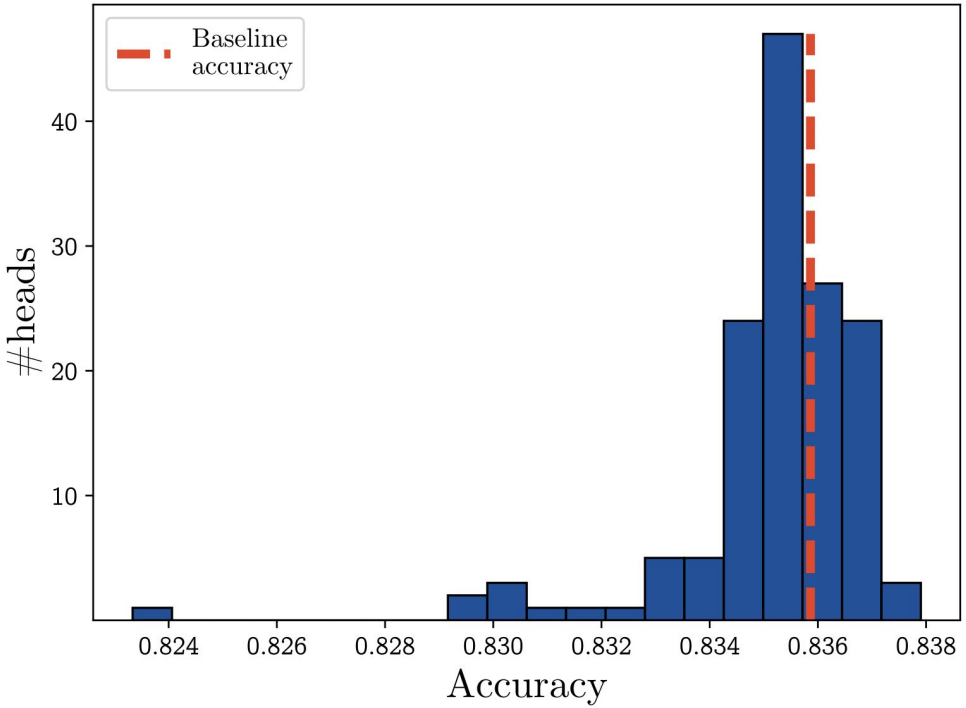
- A practical choice: updating only biases $\boldsymbol{b}$
- Computing $\nabla \boldsymbol{b}$ does not require storing activations [Cai et al., 2020]!

- In NLP, BitFit [Ben-Zaken et al., 2022] implements the same approach
- Query and second MLP layer biases are most important!



Structured Sparsity

- Structure can also be combined with sparse methods
- We can prune entire groups such as certain filters in a CNN [Anwar et al., 2015]
- In NLP, attention heads in pre-trained models have been pruned [Voita et al., 2019; Michel et al., 2019]



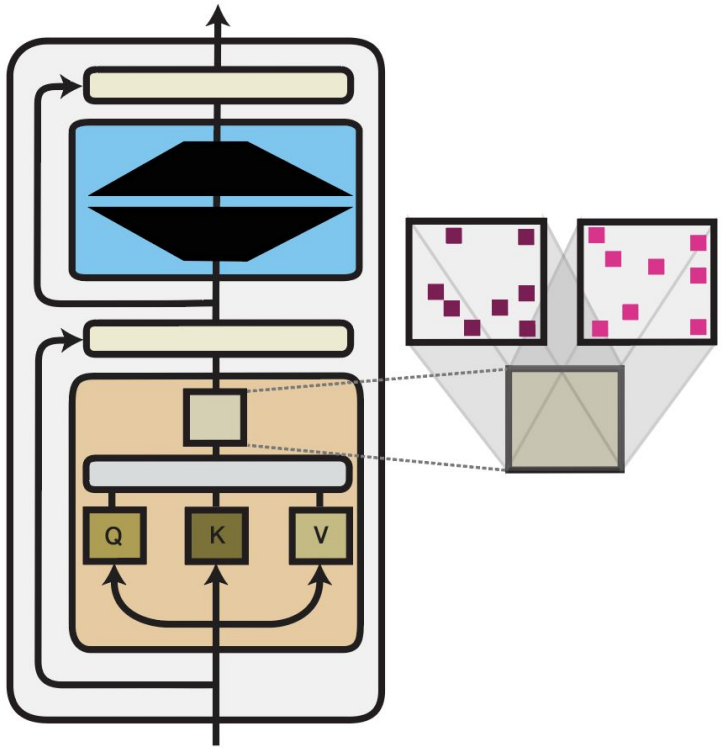
Distribution of BERT attention heads after pruning a single head based on MNLI performance [Michel et al., 2019]

Parameter Composition

1. Sparse Subnetworks

2. Structured Composition

3. Low-rank Composition

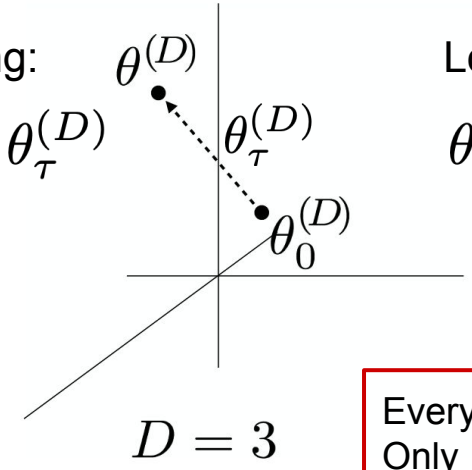


Low-rank Composition

- Another useful inductive bias: module parameters ϕ should lie in a low-dimensional space
- Li et al. [2018] show that models can be optimized in a low-dimensional, randomly oriented subspace rather than the full parameter space

Standard fine-tuning:

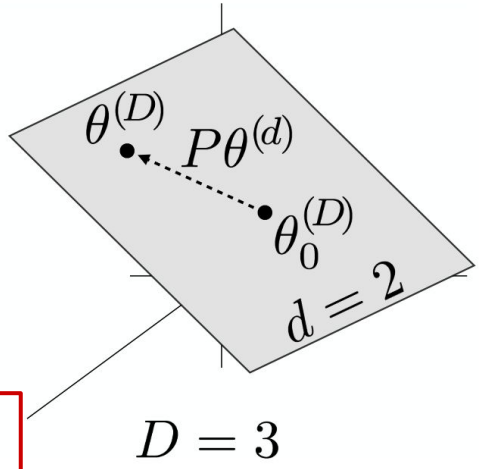
$$\theta^{(D)} = \theta_0^{(D)} + \theta_\tau^{(D)}$$



Low-rank fine-tuning:

$$\theta^{(D)} = \theta_0^{(D)} + P\theta^{(d)}$$

A random $D \times d$ projection matrix



Everything but $\theta^{(d)}$ is fixed. Only d dimensions are optimized.

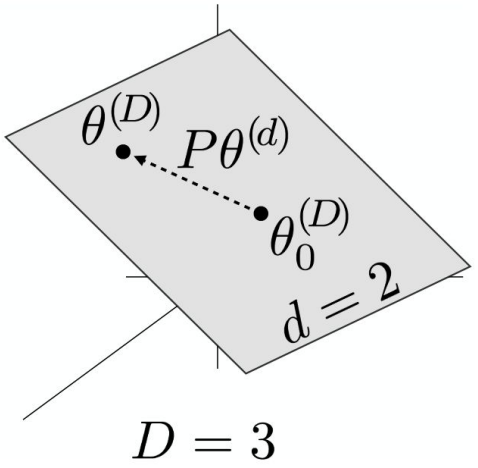
Low-rank Fine-tuning

- In our notation, low-rank fine-tuning takes the form: $f'_\theta = f_{\theta + P\phi}$ where $P \in \mathbb{R}^{D \times d}$.
- With a dense matrix $D \times d$, this scales as $\mathcal{O}(Dd)$ in matrix-vector multiply time and storage space
- For large pre-trained models, we need a more efficient solution

Low-rank fine-tuning:

$$\theta^{(D)} = \theta_0^{(D)} + P\theta^{(d)}$$

A random $D \times d$ projection matrix



Low-rank Fine-tuning via Matrix Factorization

- For the multiplication, we do not need to explicitly store P.

- Instead, we can decompose the matrix.
- A classic choice: Fastfood transform [Le et al., 2013], which requires only $\mathcal{O}(D)$ space and $\mathcal{O}(D \log d)$ time.

- Factorizes P via a sequence of random matrices with special properties:

$$P = HG\Pi HB$$

B: Random diagonal matrix with equal probability ± 1 entries

Π: Random permutation matrix

G: Random diagonal matrix with independent standard normal entries

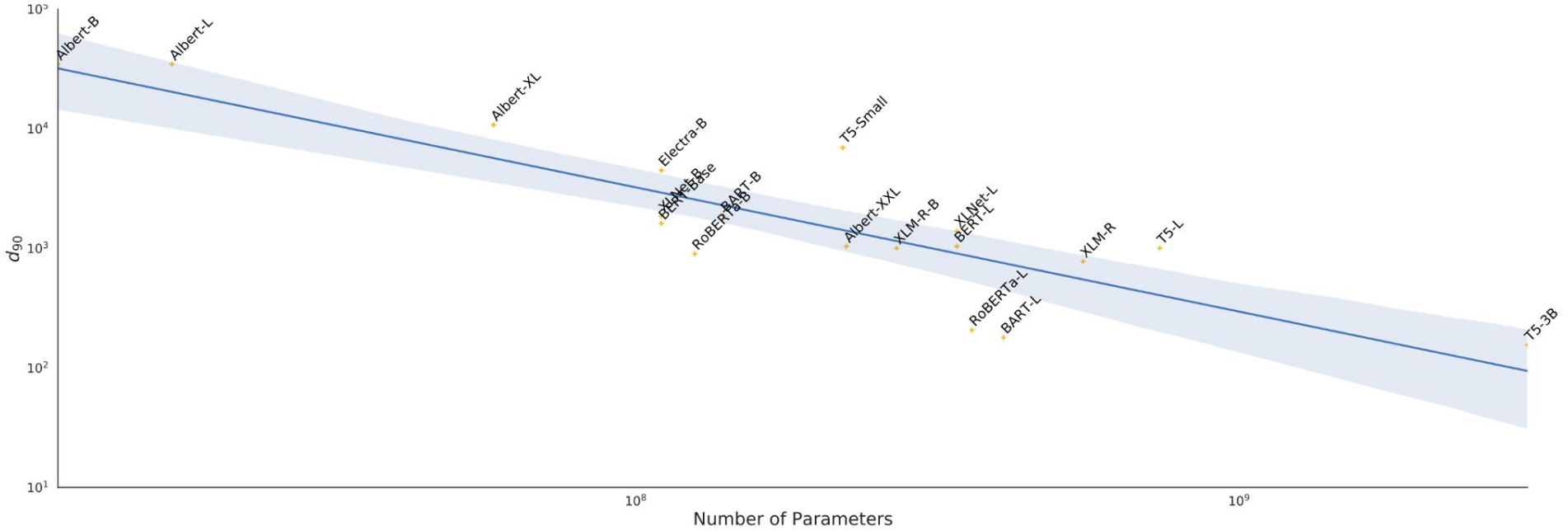
H: Hadamard matrix

Intrinsic Dimensionality

- Li et al. [2018] refer to the minimum d where a model achieves within 90% of the full-parameter model performance, d_{90} as the intrinsic dimensionality of a task
- Aghajanyan et al. [2021] investigate the intrinsic dimensionality of different NLP tasks and pre-trained models

- Observations:
  - Intrinsic dimensionality decreases during pre-training
  - Larger models have lower intrinsic dimensionality

Intrinsic Dimensionality



Intrinsic dimension d_{90} on the MRPC dataset for models of different sizes [Aghajanyan et al., 2021]

Low-rank Adaptation (LoRA)

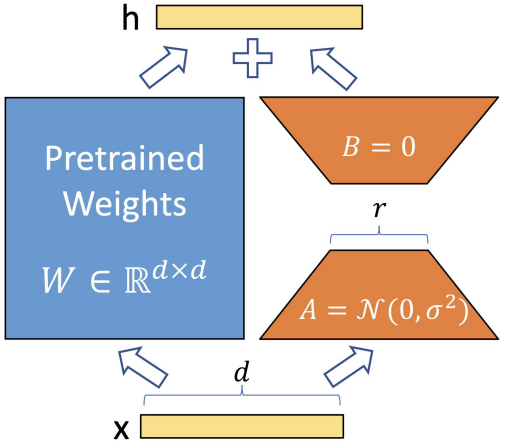
- In addition, instead of learning a low-rank factorization via a random matrix P, we can learn the projection matrix directly (if it is small enough)
- LoRA [Hu et al., 2022] learns two low-rank matrices $A \in \mathbb{R}^{r \times k}$ and $B \in \mathbb{R}^{d \times r}$ that are applied to the self-attention weights:

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

Matrix rank

Hidden dimension

Input dimension



- In our notation, this looks like the following:

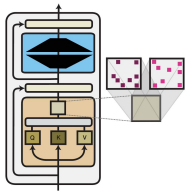
$$f_i' = f_{\theta_i + \mathrm{vec}(B_i A_i)} \; \forall f_i' \in \mathcal{G}$$

Vectorization

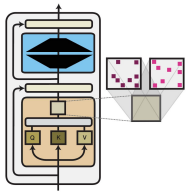
Parameter Composition Comparison

	Computation function
Sparse Subnetworks	$f'_\theta = f_{\theta+\phi}$ $f'_\theta = f_{\theta \circ \boldsymbol{b}+\phi}$ where $\phi_i = 0$ if $\boldsymbol{b}_i = 0$ $f'_\theta = f_{\theta \circ \boldsymbol{b}}$
Structured Composition	$f'_i = f_{\theta_i+\phi_i} \; \forall f'_i \in \mathcal{G}$
Low-rank Composition	$f'_\theta = f_{\theta+P\phi}$ where $P \in \mathbb{R}^{D \times d}$

Computation Functions Comparison

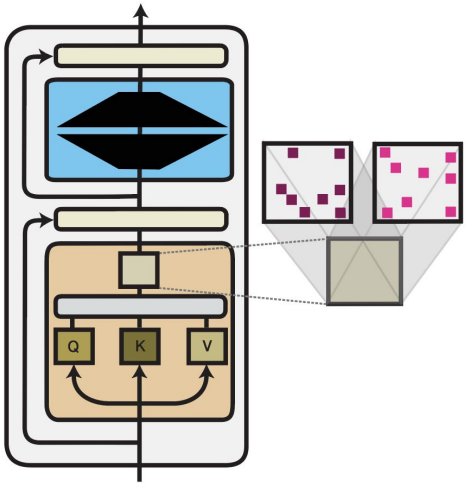
		Parameter efficiency	Training efficiency	Inference efficiency	Performance	Composition-ality
Parameter composition		Methods such as diff pruning require < 0.5% of parameters	Pruning requires re-training iterations	Does not increase the model size	E.g., LoRA achieves strong performance	Subnetworks can be composed

Computation Functions Comparison

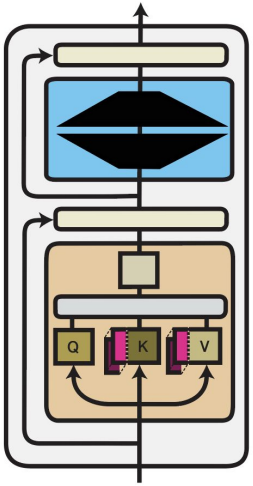
		Parameter efficiency	Training efficiency	Inference efficiency	Performance	Composition-ality
Parameter composition		+	-	++	+	+

This is mainly meant as high-level guidelines. Individual methods may have different trade-offs and mitigate certain weaknesses.

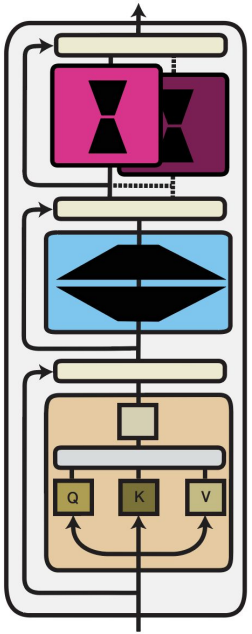
Three Computation Functions



Parameter Composition



Input Composition

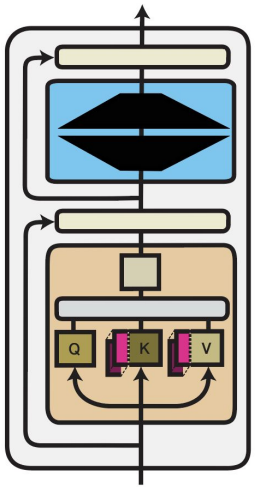


Function Composition

Input Composition

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

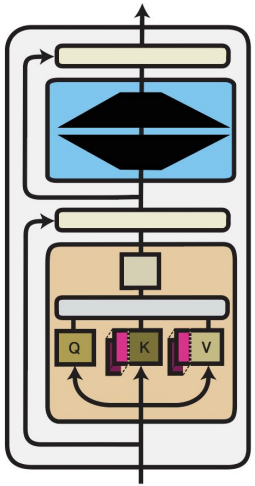
$$f_i'(\boldsymbol{x}) = f_{\theta_i}([\phi_i, \boldsymbol{x}])$$



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

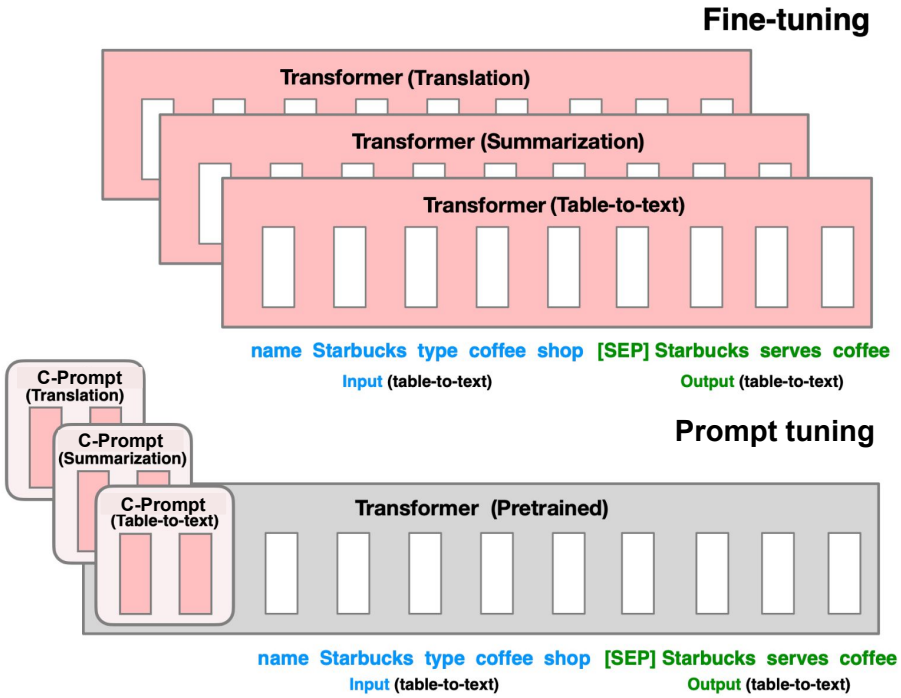
- However, models are sensitive to the formulation of the prompt [Webson & Pavlick, 2022] and to the order of examples [Zhao et al., 2021; Lu et al., 2022]



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]

- ϕ is typically a matrix consisting of a sequence of continuous prompt embeddings



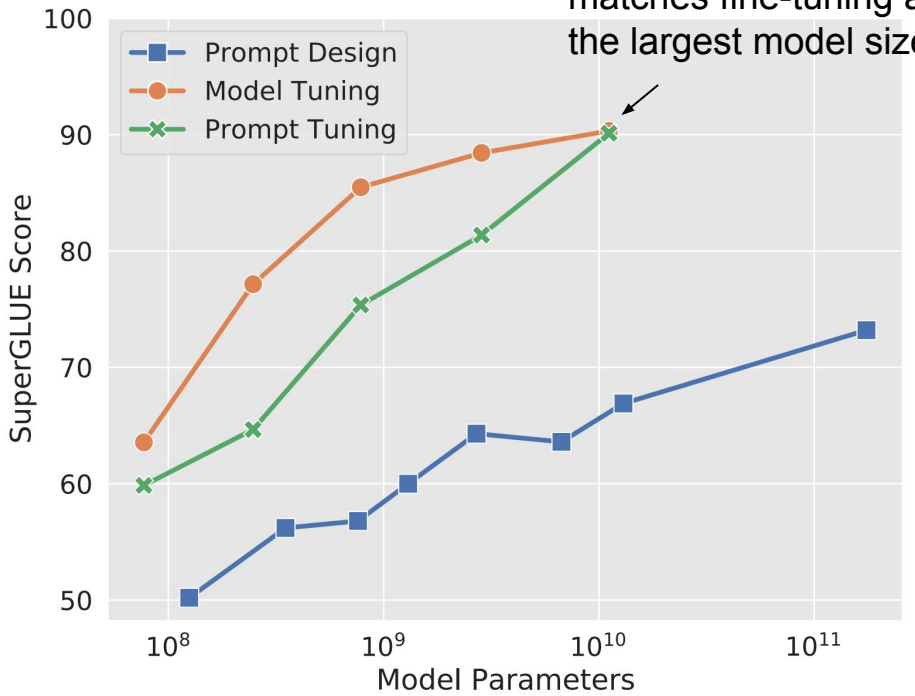
Fine-tuning vs Prompt tuning
(adapted from [Li & Liang, 2021])

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

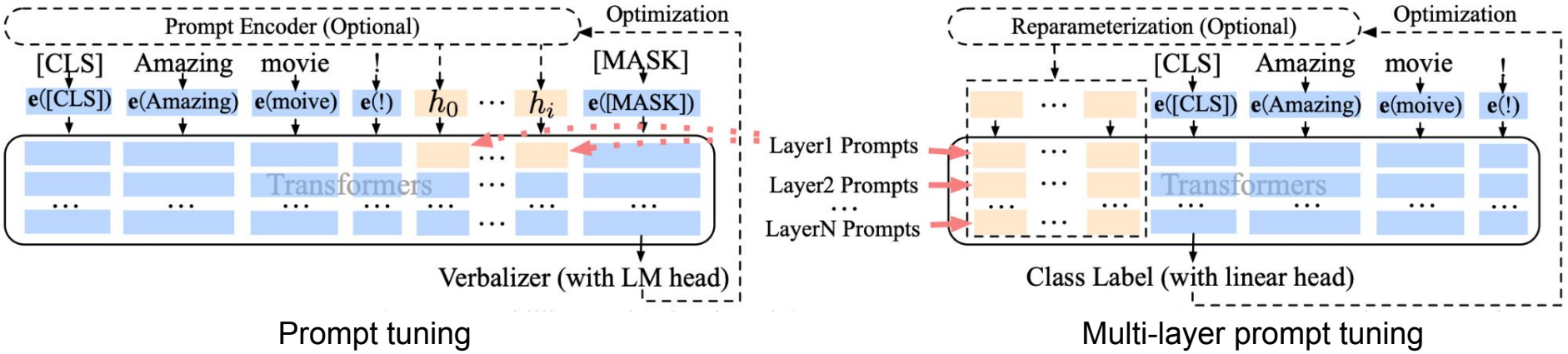


Prompt tuning vs standard fine-tuning and prompt design across T5 models of different sizes [Lester et al., 2021]

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 [Li & Liang, 2021; Liu et al., 2022]

- Continuous prompts in later layers are more important

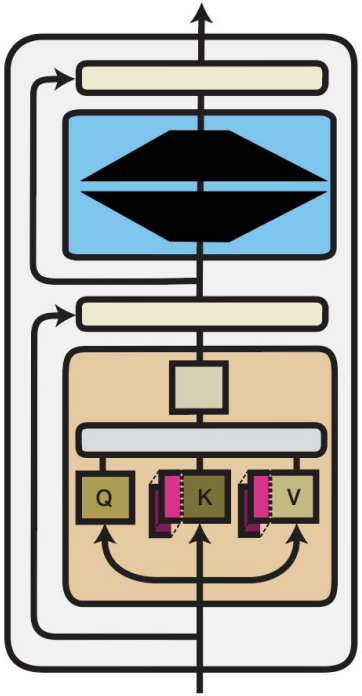


Prompt tuning

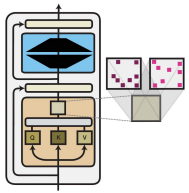
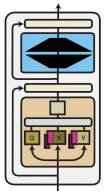
Multi-layer prompt tuning

Multi-Layer Prompt Tuning

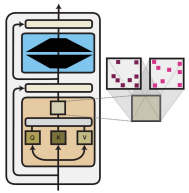
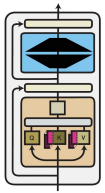
- In practice, continuous prompts ϕ_i are concatenated with the keys and values in the self-attention layer [Li & Liang, 2021]



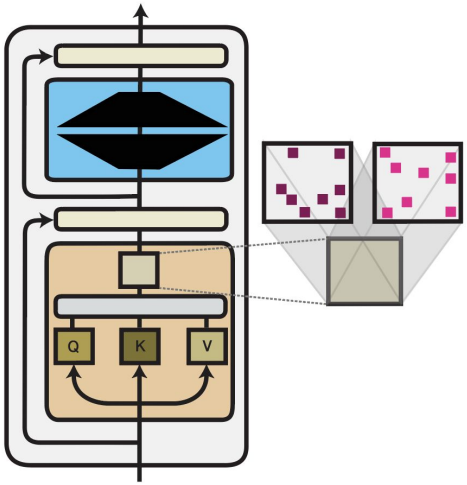
Computation Functions Comparison

		Parameter efficiency	Training efficiency	Inference efficiency	Performance	Composition-ality
Parameter composition		+	-	++	+	+
Input composition		Only add a small number of parameters	Extend the model's context window		Requires large models to perform well	Continuous prompts have been composed

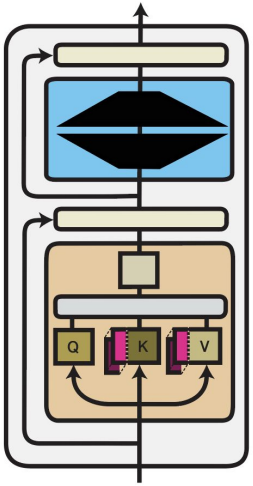
Computation Functions Comparison

		Parameter efficiency	Training efficiency	Inference efficiency	Performance	Composition-ality
Parameter composition		+	-	++	+	+
Input composition		++	--	--	-	+

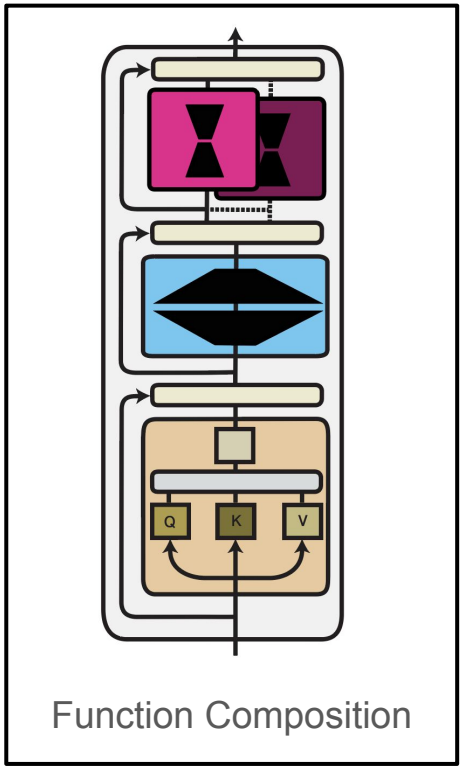
Three Computation Functions



Parameter Composition



Input Composition



Function Composition

Function Composition

- Function composition augments a model's functions with new task-specific functions:

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

- Most commonly used in multi-task learning where modules of different tasks are composed.
- Relevant surveys: [Ruder, 2017; Crawshaw, 2020]

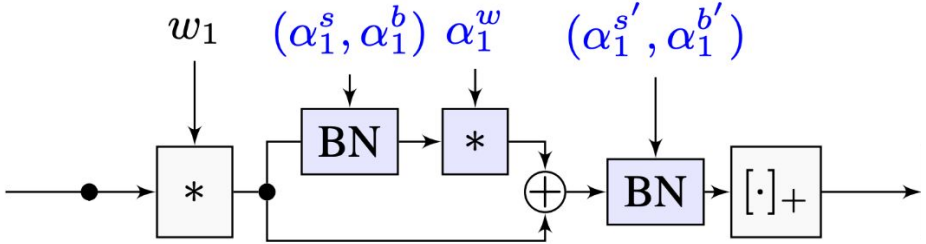
→ Focus in this tutorial is on functions that can be added to a pre-trained model

Adapters

- Main purpose of functions f_{ϕ_i} added to a pre-trained model is to adapt it

→ Functions are also known as ‘adapters’

- Design of adapters is model-specific
- ResNet adapter in CV: batch normalization and 1x1 convolution [Rebuffi et al., 2017]



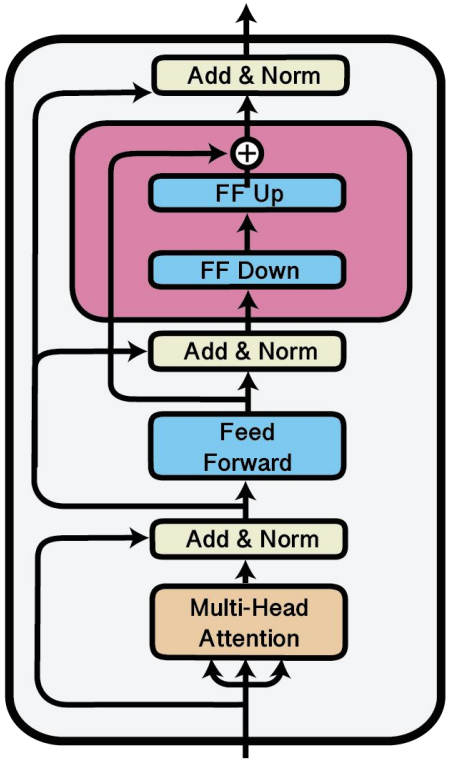
Residual adapter in a ResNet (adapter parameters are in blue) [Rebuffi et al., 2017]

Adapters in Transformer Models

- In NLP, an adapter in a Transformer layer typically consists of a feed-forward down-projection $W^D \in \mathbb{R}^{k \times d}$, a feed-forward up-projection $W^U \in \mathbb{R}^{d \times k}$ and an activation function σ[Houlsby et al,. 2019]:

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

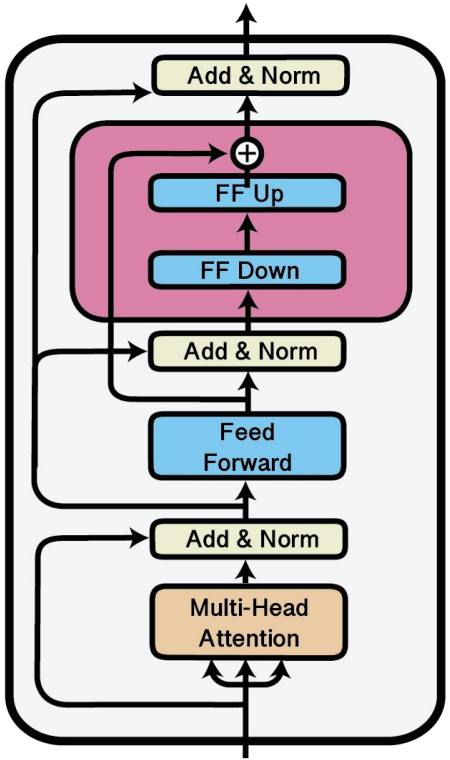
- σ is commonly a ReLU but other variants have been explored [Stickland & Murray, 2019]



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



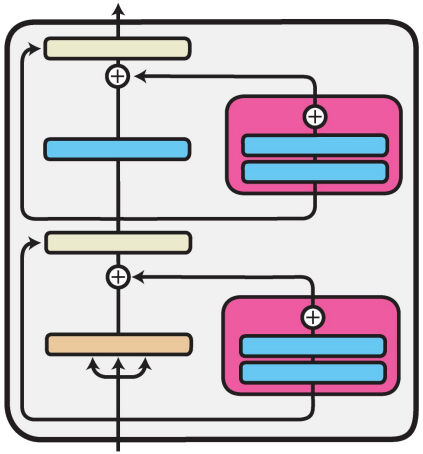
Sequential and Parallel Adapters

- Adapters can be routed sequentially or in parallel
- Sequential adapters are inserted between functions:

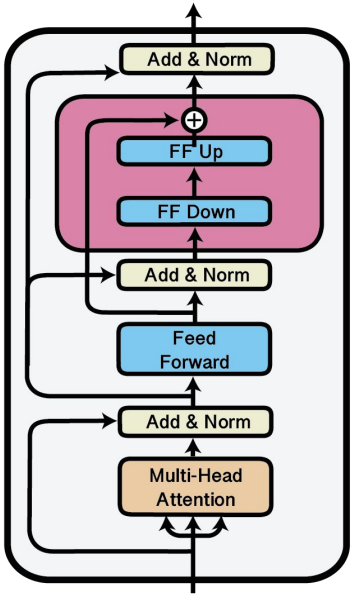
$$f_i'(\boldsymbol{x}) = f_{\phi_i}(f_{\theta_i}(\boldsymbol{x}))$$

- Parallel adapters are applied in parallel:

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



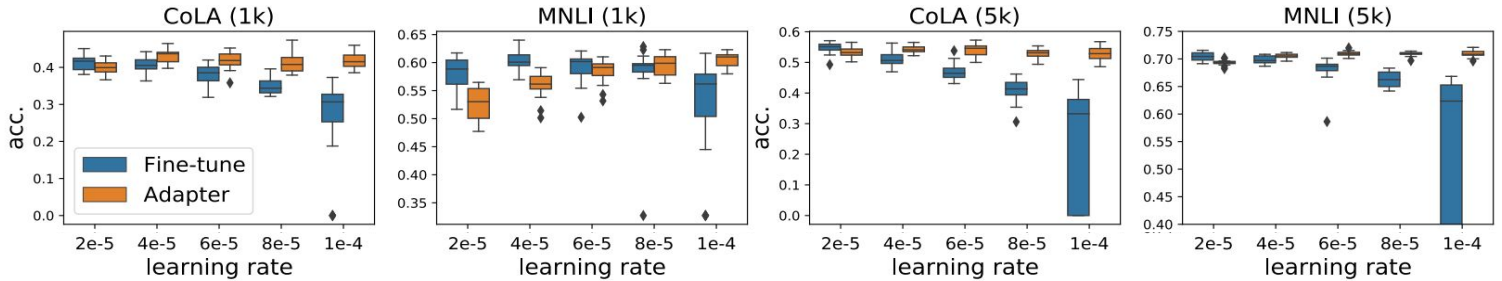
Two parallel adapters [Stickland & Murray, 2019]



A sequential adapter [Houlsby et al., 2019]

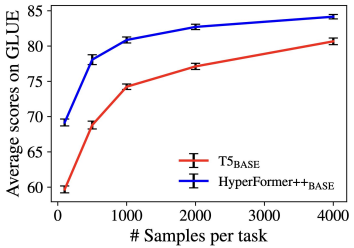
Benefits of Adapters

- Increased robustness [He et al., 2021; Han et al., 2021]



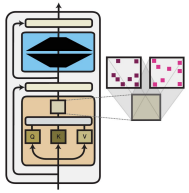
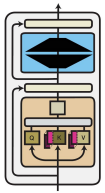
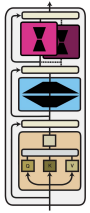
BERT test performance distributions over 20 runs with different learning rates [He et al., 2021]

- Increased sample efficiency

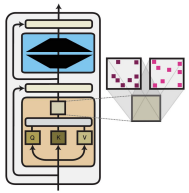
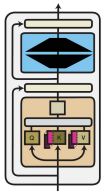
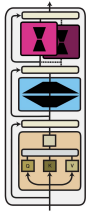


Results on GLUE with different numbers of training samples per task [Mahabadi et al., 2021]

Computation Functions Comparison

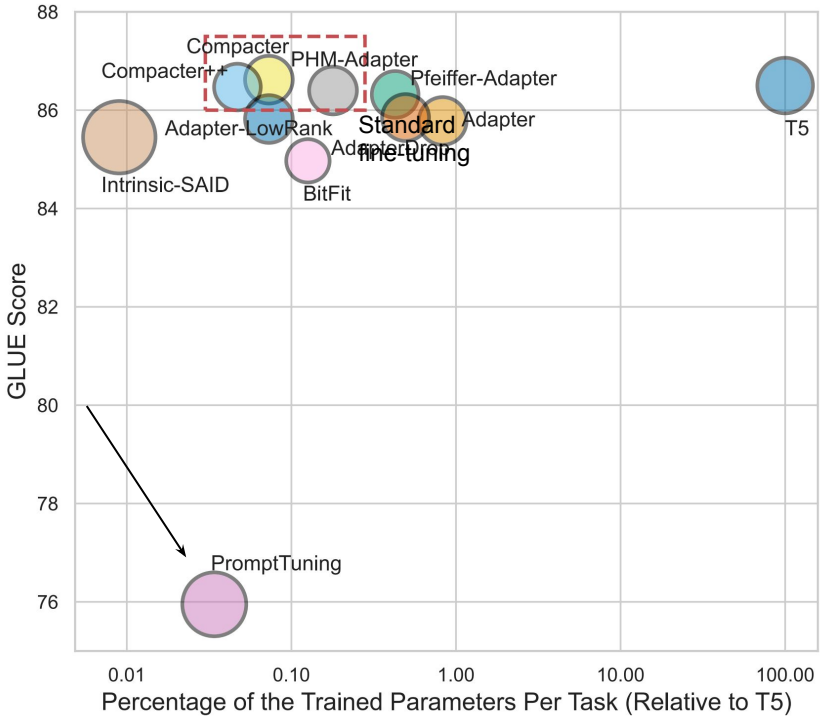
		Parameter efficiency	Training efficiency	Inference efficiency	Performance	Composition-ality
Parameter composition		+	-	++	+	+
Input composition		++	--	--	-	+
Function Composition		Adapters depend on the hidden size	Does not require gradients of frozen params	New functions increase # of operations	Match or outperform standard fine-tuning	Adapters can be composed

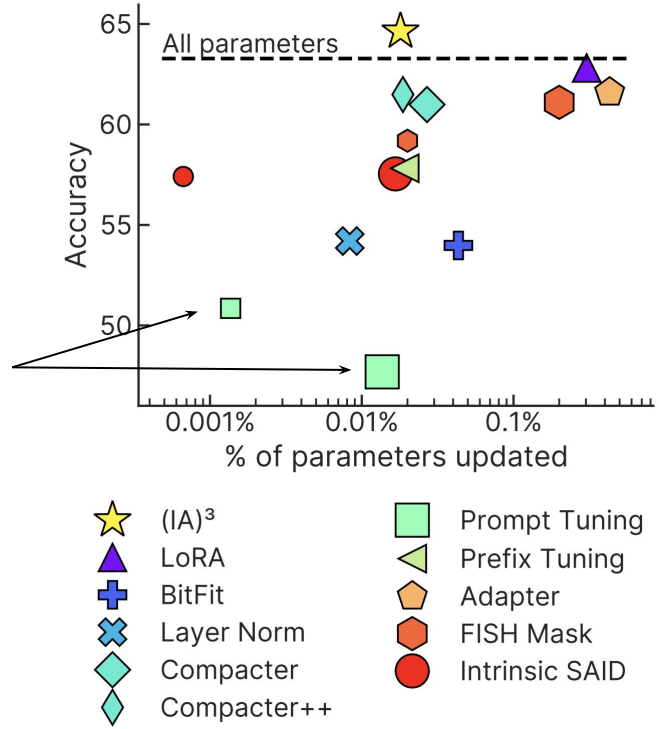
Computation Functions Comparison

		Parameter efficiency	Training efficiency	Inference efficiency	Performance	Composition-ality
Parameter composition		+	-	++	+	+
Input composition		++	--	--	-	+
Function Composition		-	+	-	++	+

Performance Comparison

Prompt tuning underperforms the other methods due to limited capacity

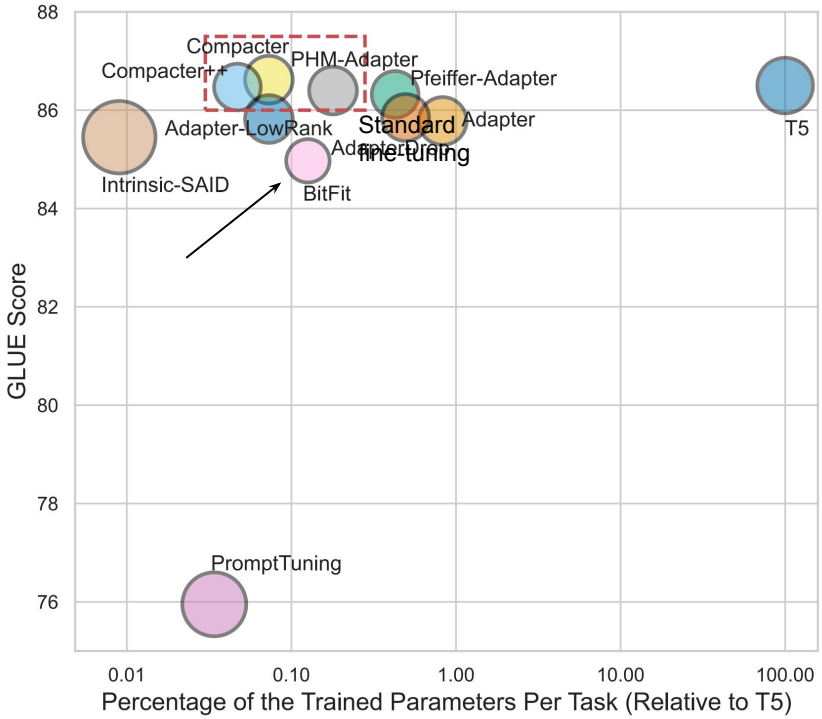


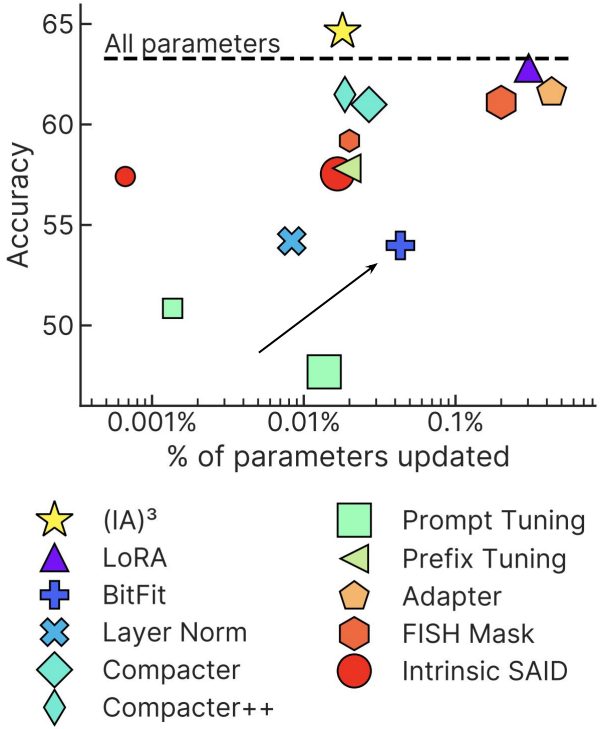


Average performance, % of trained parameters per task, and (left) memory footprint (circle size) of different methods on T5-Base (222B parameters; left) [Mahabadi et al., 2021] and T3-3B (right) [Liu et al., 2022]

Performance Comparison

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

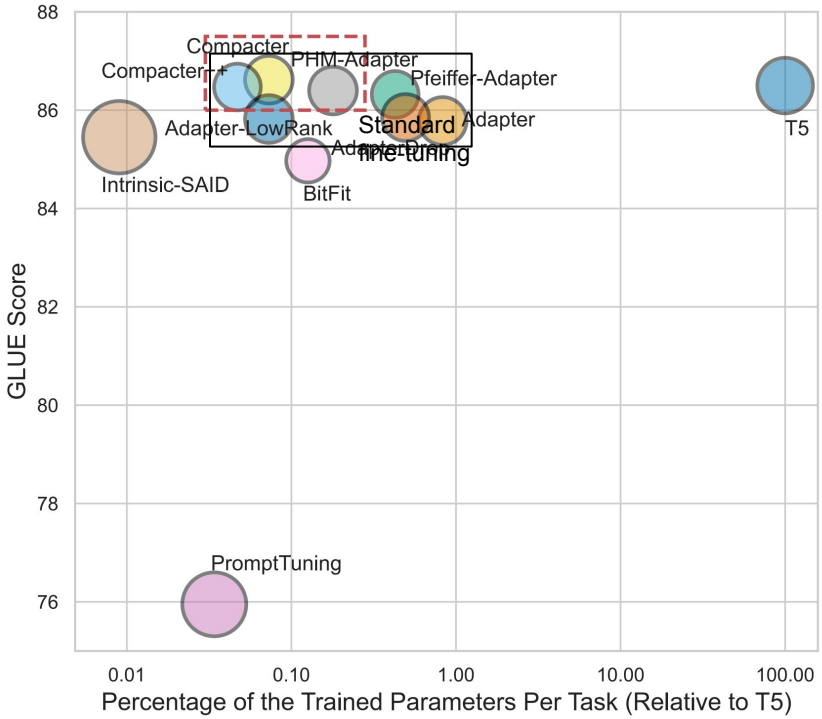


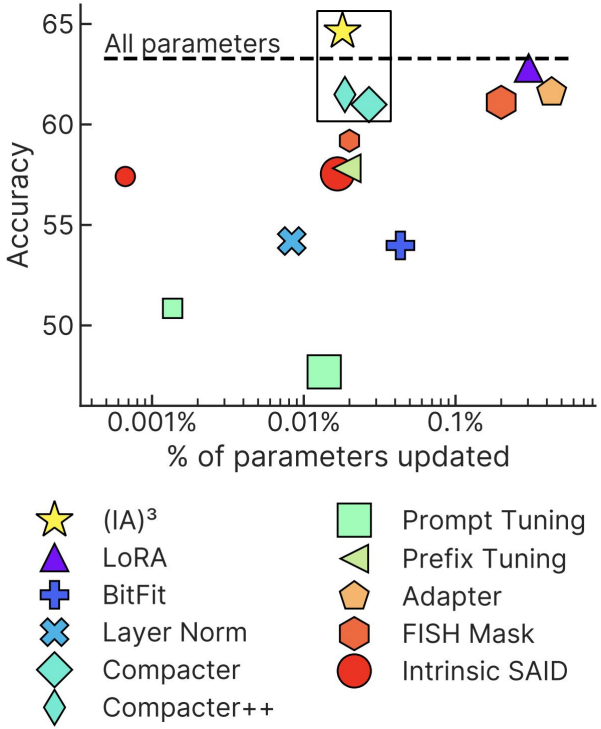


Average performance, % of trained parameters per task, and (left) memory footprint (circle size) of different methods on T5-Base (222B parameters; left) [Mahabadi et al., 2021] and T3-3B (right) [Liu et al., 2022]

Performance Comparison

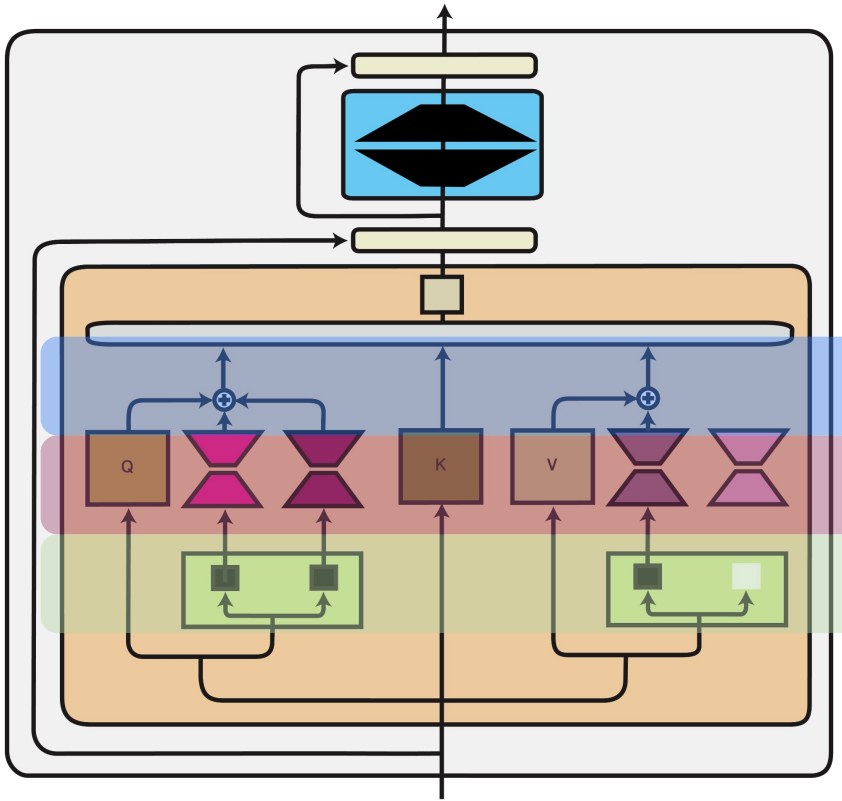
Function composition methods such as adapter, compacter, and IA^3 achieve the best performance but add more parameters





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]

Agenda



Parameter-efficient Models

1) Computation Function

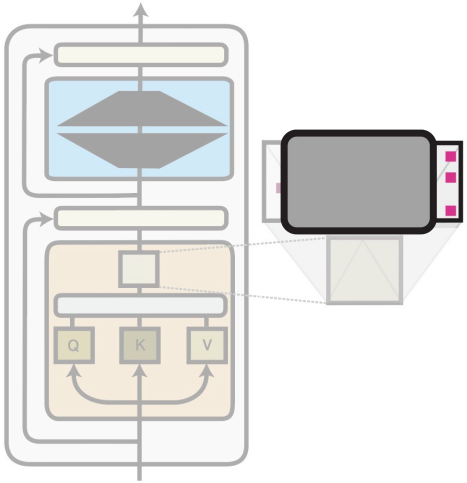
Modular Models

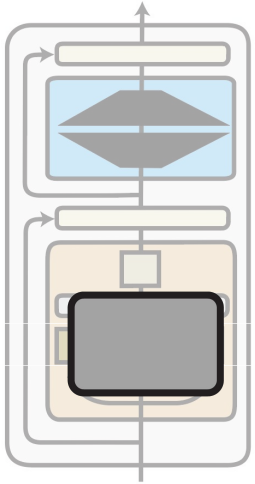
3) Aggregation

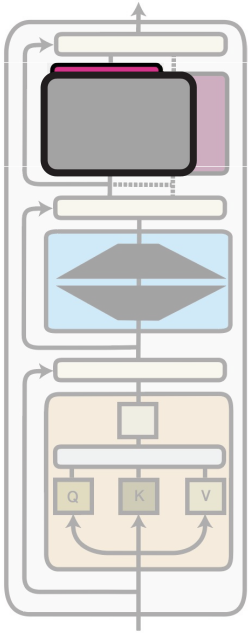
2) Routing

Applications

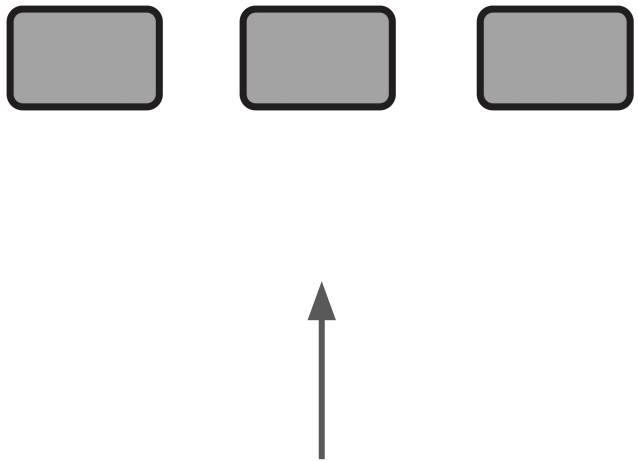
Introduction to Routing



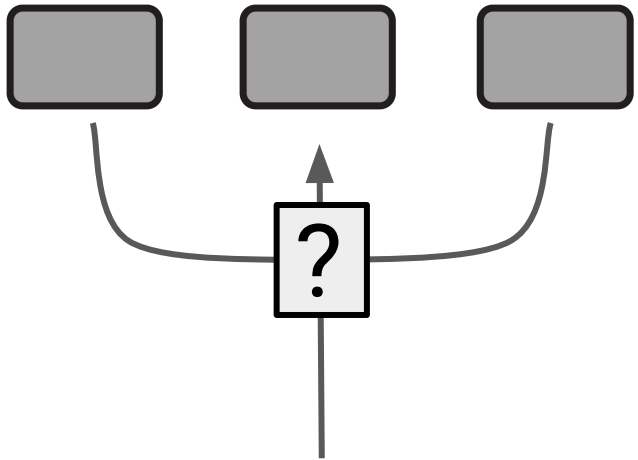




Introduction to Routing



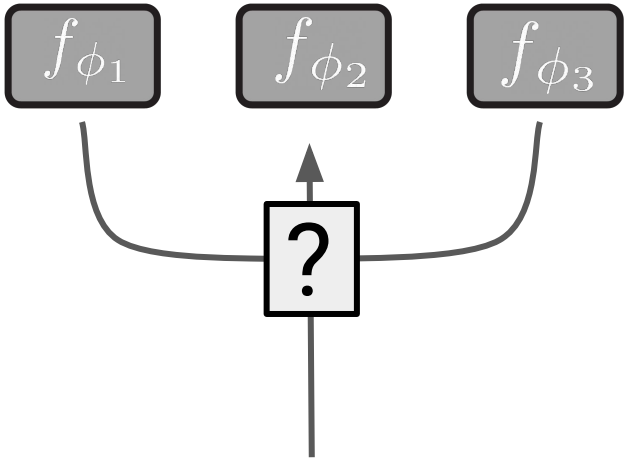
Introduction to Routing



Introduction to Routing

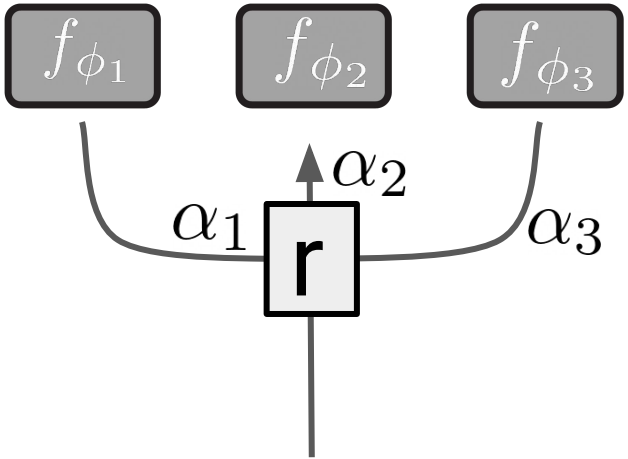
$$\mathcal{F} = f_{\phi_1}, f_{\phi_2}, \ldots, f_{\phi_{|\mathcal{F}|}}$$

Introduction to Routing



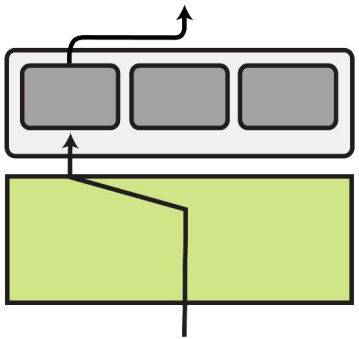
Introduction to Routing

$$r(\cdot) \rightarrow \alpha_i$$



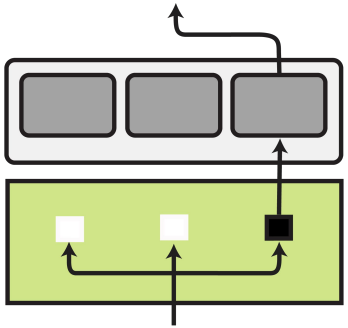
Routing

Fixed Routing

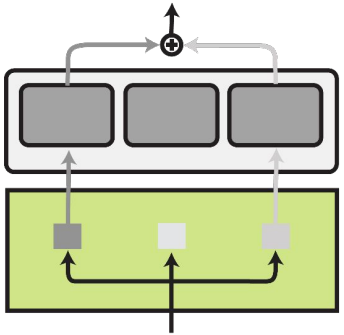


Learned Routing

Hard Learned Routing

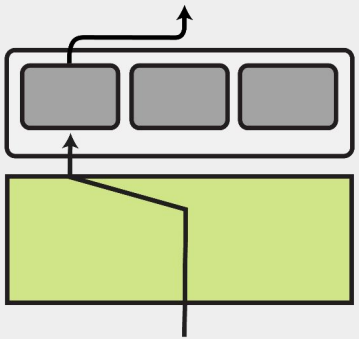


Soft Learned Routing



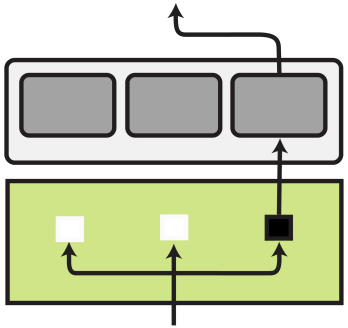
Routing

Fixed Routing

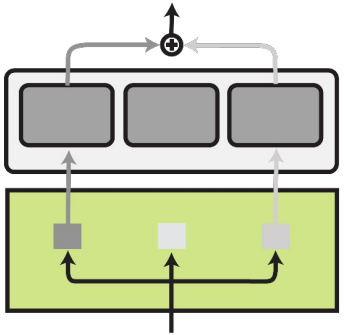


Learned Routing

Hard Learned Routing



Soft Learned Routing

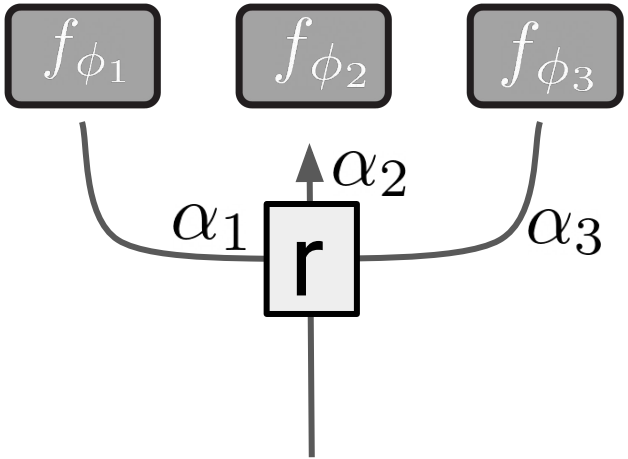


Fixed Routing

Routing decision is made a-priori.

$$r(\phi_i) = \begin{cases} 1 & \text{if } i \in \mathcal{K} \\ 0 & \text{otherwise} \end{cases}$$

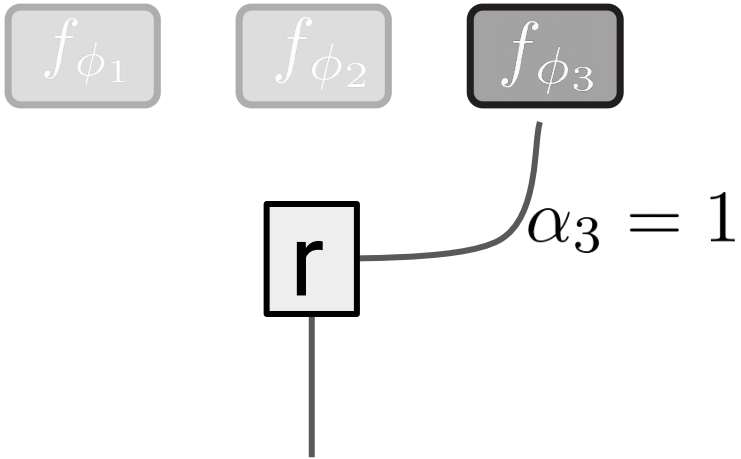
$$\mathcal{K} \subseteq \mathcal{F}$$



Fixed Routing

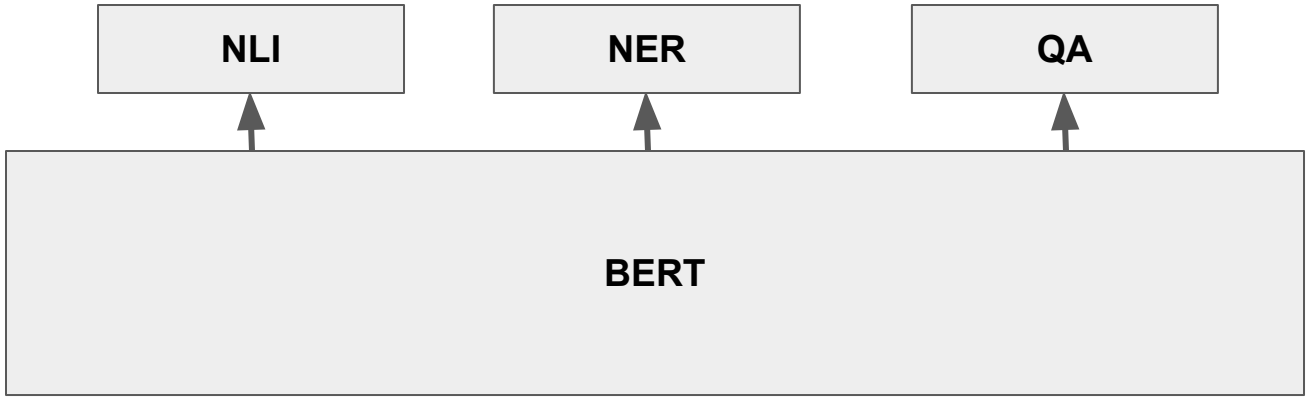
Routing decision is made a-priori.

$$r(\phi_i) = \begin{cases} 1 & \text{if } i \in \mathcal{K} \\ 0 & \text{otherwise} \end{cases}$$



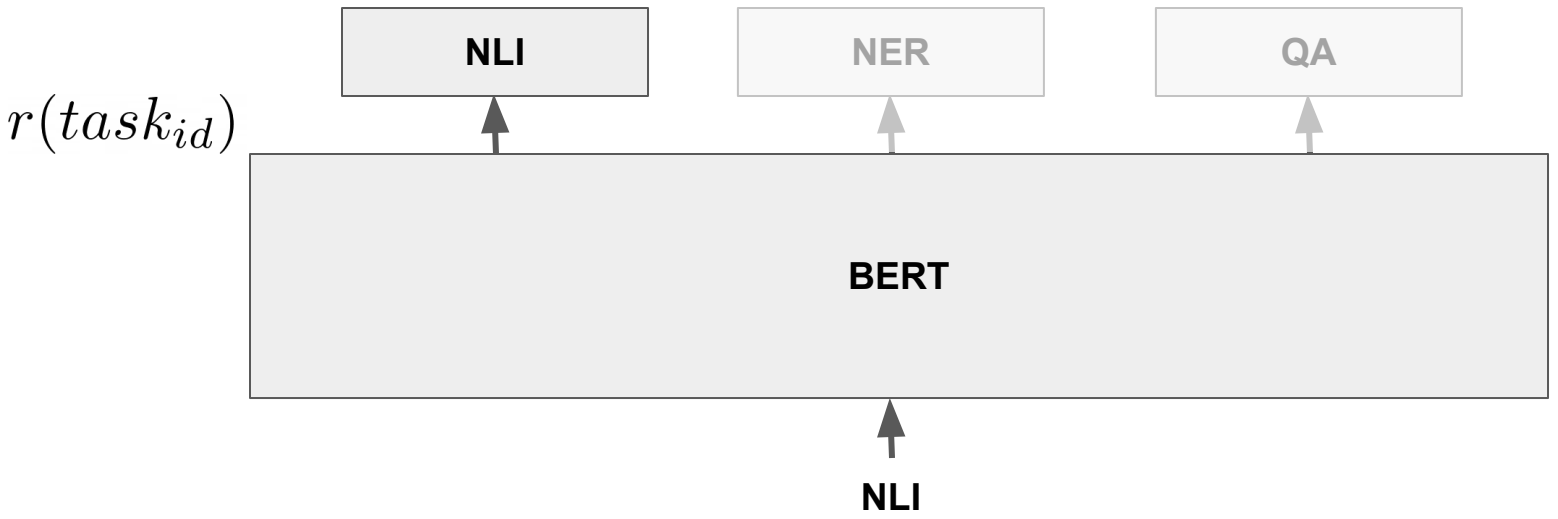
Fixed Routing

Multi Task Learning



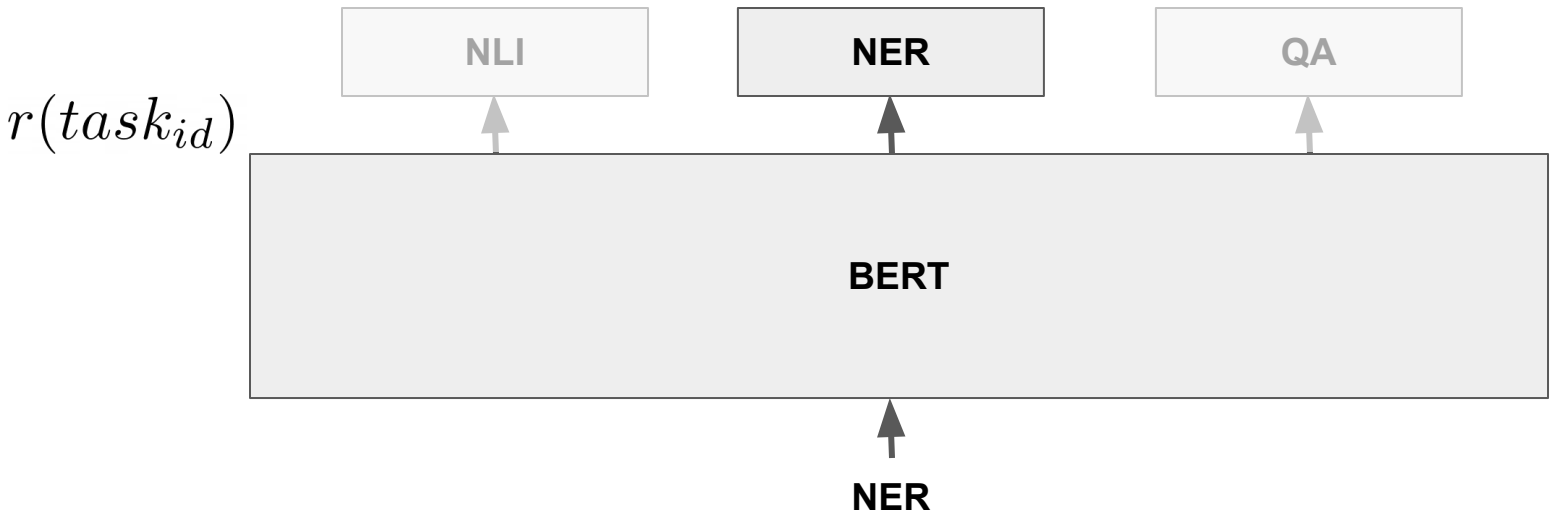
Fixed Routing

Multi Task Learning



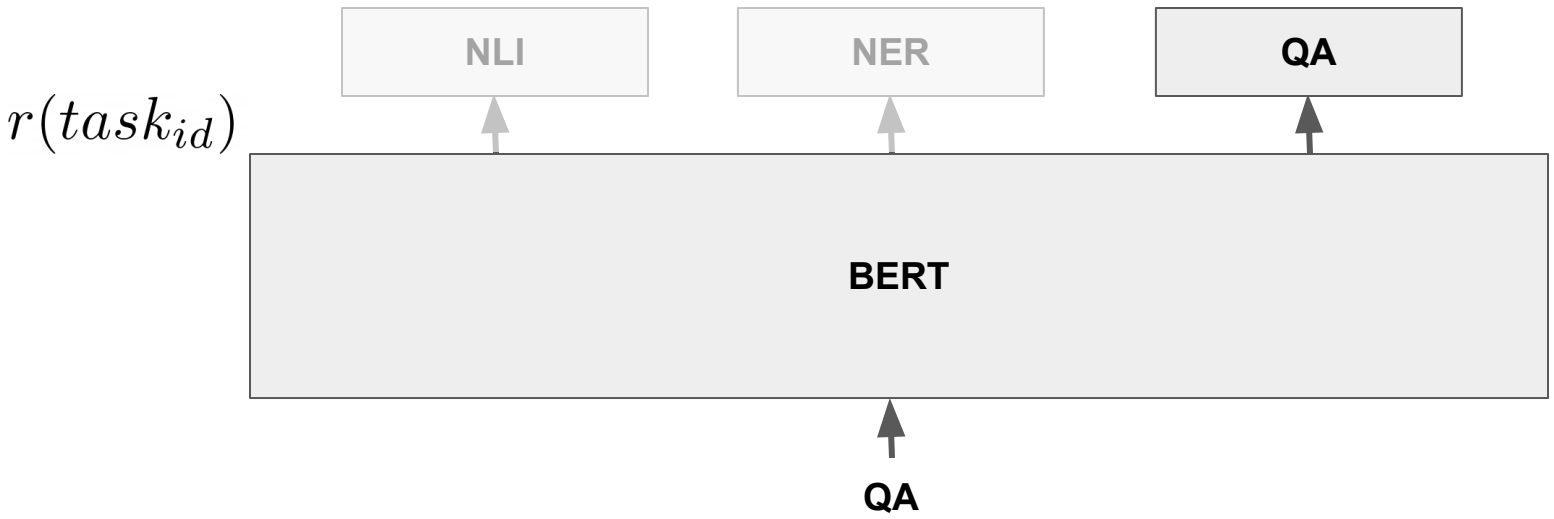
Fixed Routing

Multi Task Learning

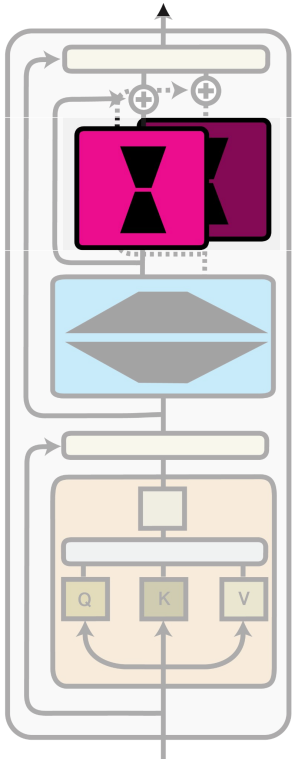


Fixed Routing

Multi Task Learning



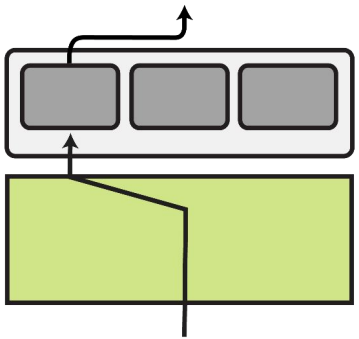
Fixed Routing



$$r(task_{id}) \rightarrow \text{task adapter}$$

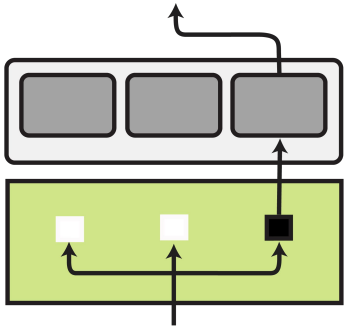
Routing

Fixed Routing

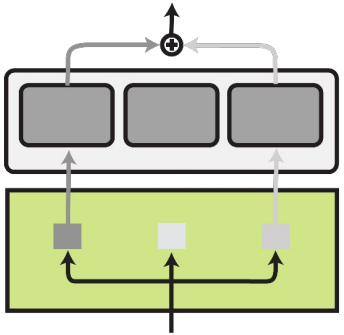


Learned Routing

Hard Learned Routing

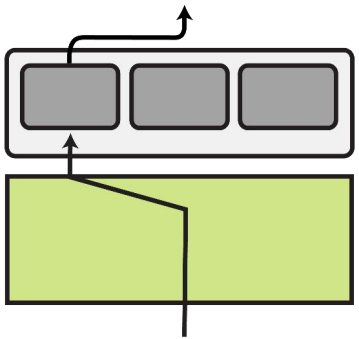


Soft Learned Routing



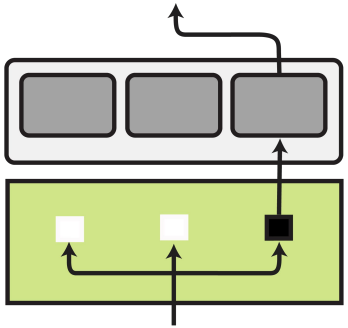
Routing

Fixed Routing

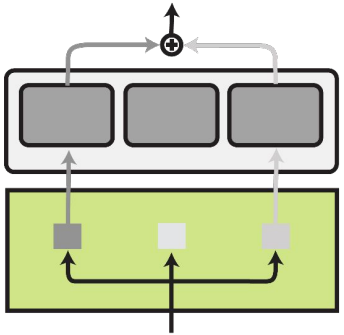


Learned Routing

Hard Learned Routing

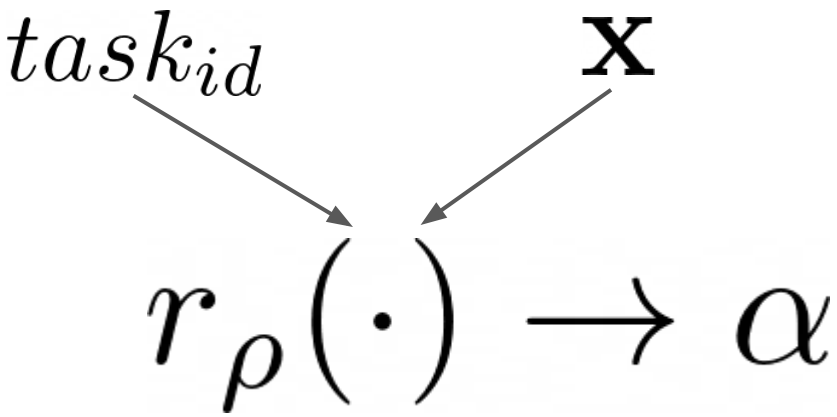


Soft Learned Routing



Learned Routing

Parametrize the routing function $r_\rho(\cdot)$ when routing decisions cannot be made a priori .

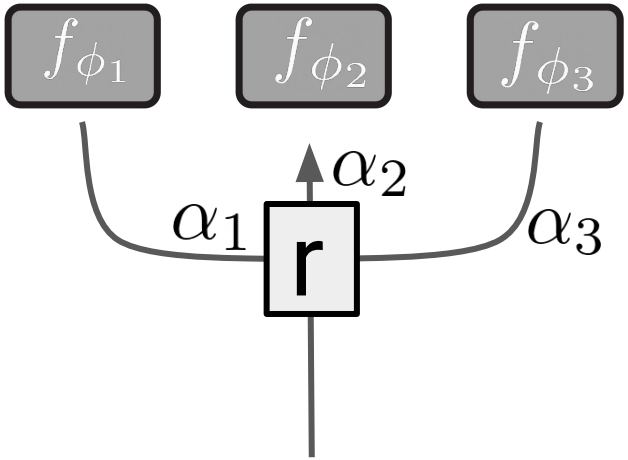


Learned Routing - Challenges

Rosenbaum et al. (2017)

Learned Routing - Challenges

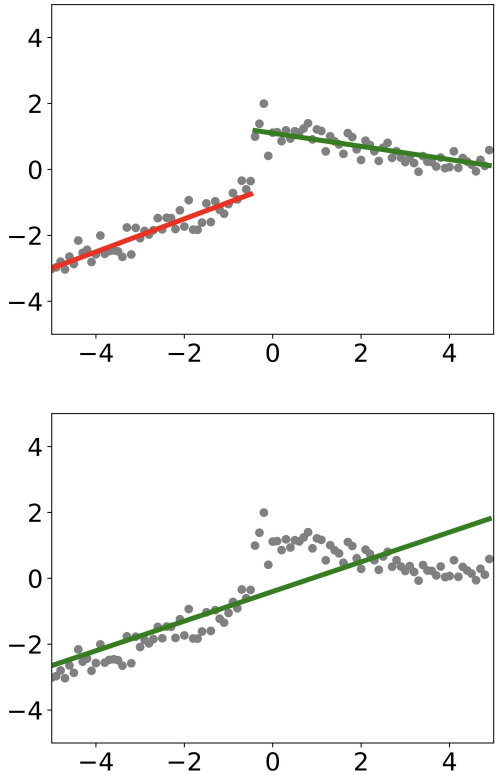
- **Training Instability**
  - Router and Modules are untrained => routing dynamics might never stabilize.



Rosenbaum et al. (2017)

Learned Routing - Challenges

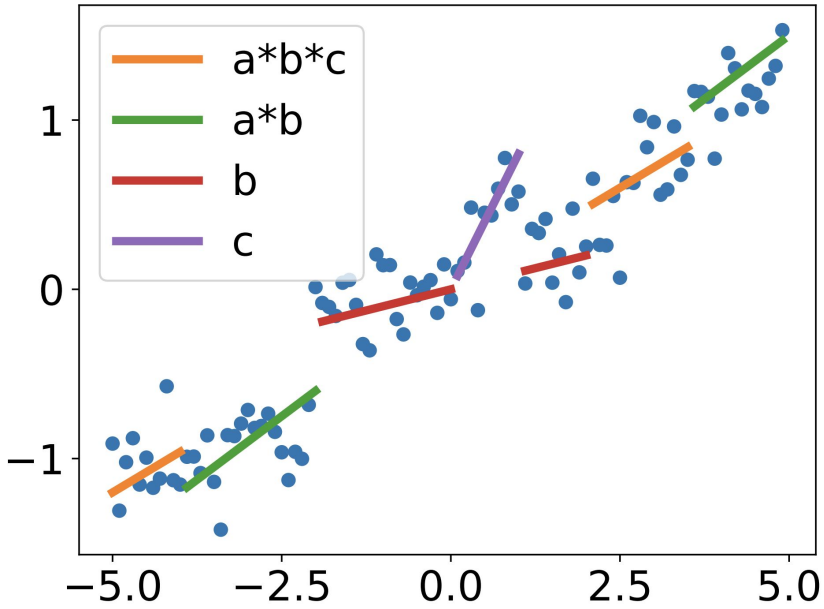
- **Training Instability**
  - Router and Modules are untrained => routing dynamics might never stabilize.
- **Module Collapse**
  - The router falls into a local optimum, choosing one or two modules exclusively.



Plots courtesy of [Rosenbaum et al. (2017)]

Learned Routing - Challenges

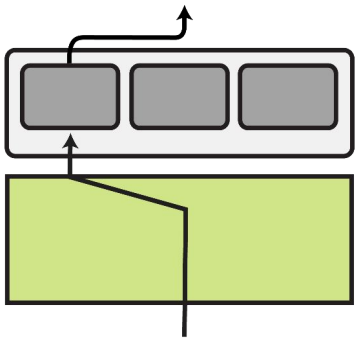
- **Training Instability**
  - Router and Modules are untrained => routing dynamics might never stabilize.
- **Module Collapse**
  - The router falls into a local optimum, choosing one or two modules exclusively.
- **Overfitting**
  - Deep modular networks risk overfitting to the noise.



Plot courtesy of Rosenbaum et al. (2017)

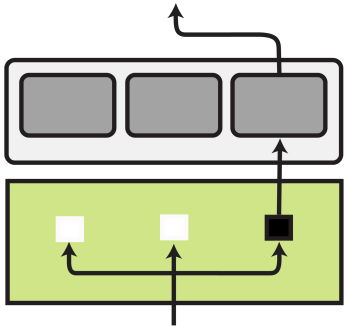
Routing

Fixed Routing

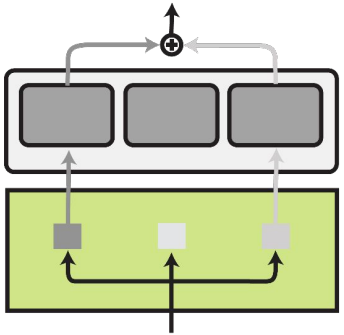


Learned Routing

Hard Learned Routing

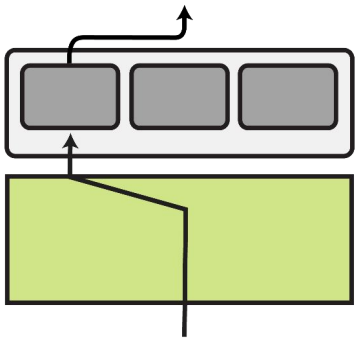


Soft Learned Routing



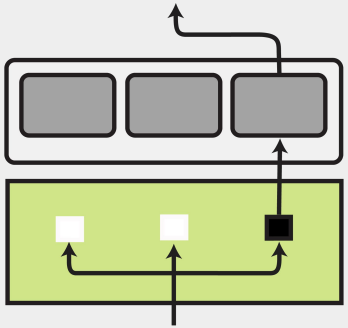
Routing

Fixed Routing

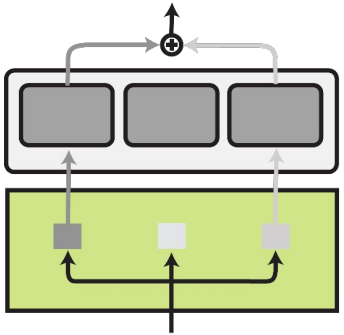


Learned Routing

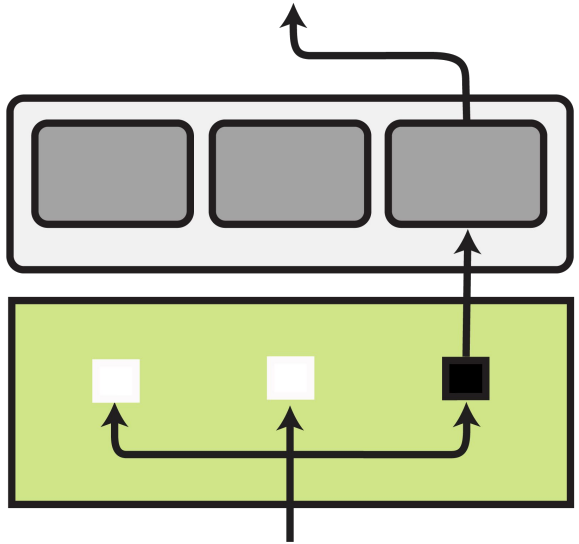
Hard Learned Routing



Soft Learned Routing

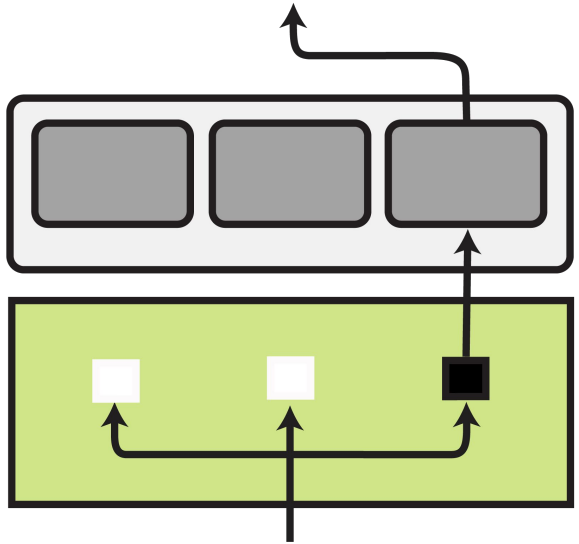


Hard Learned Routing



Hard Learned Routing

Discrete decisions are not amenable to be learned through vanilla gradient descent.

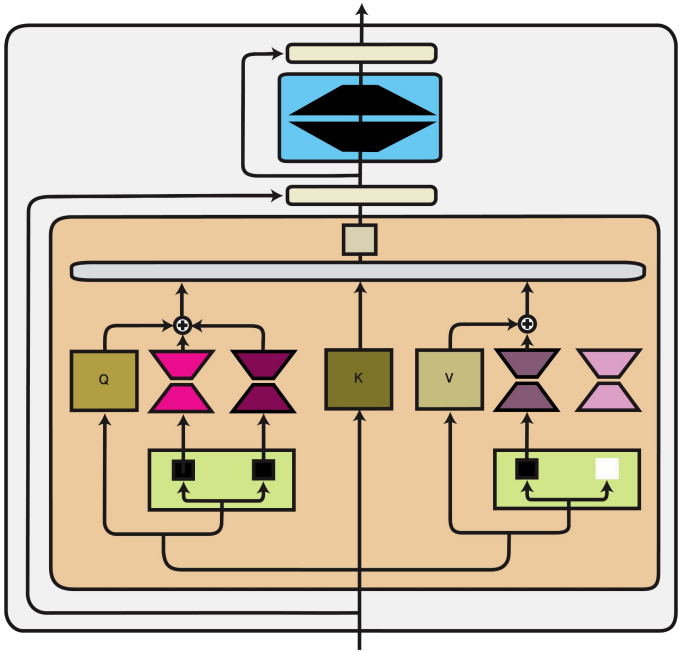


Hard Learned Routing - Stochastic Reparametrization

Gumbel Softmax (Jang et al.,2017)

$$\hat{\alpha}_{i,j} = \text{sigmoid}\left[\log \frac{\text{sigmoid}(\alpha_{i,j})\, u}{(1 - \text{sigmoid}(\alpha_{i,j}))\,(1-u)}^{1/\tau}\right]$$

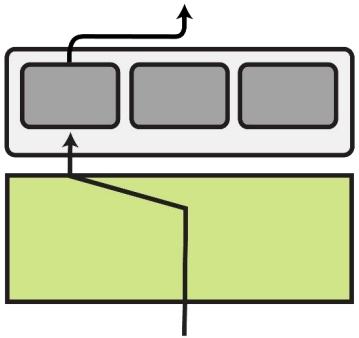
$$u \sim \text{Uniform}(0, 1).$$



Polytropon; Ponti et al. (2022)

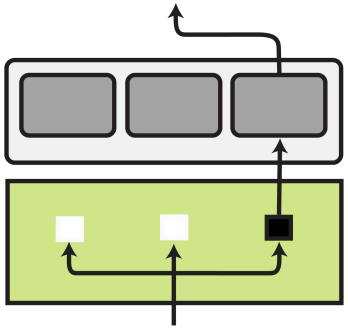
Routing

Fixed Routing

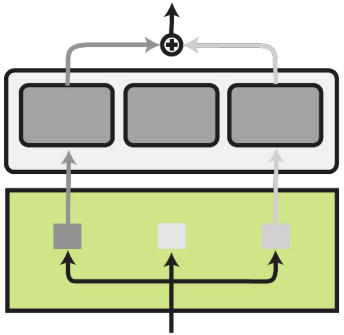


Learned Routing

Hard Learned Routing

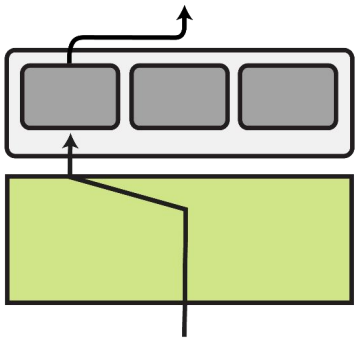


Soft Learned Routing



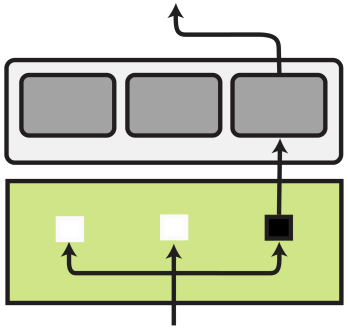
Routing

Fixed Routing

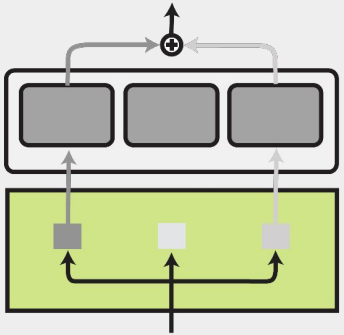


Learned Routing

Hard Learned Routing



Soft Learned Routing

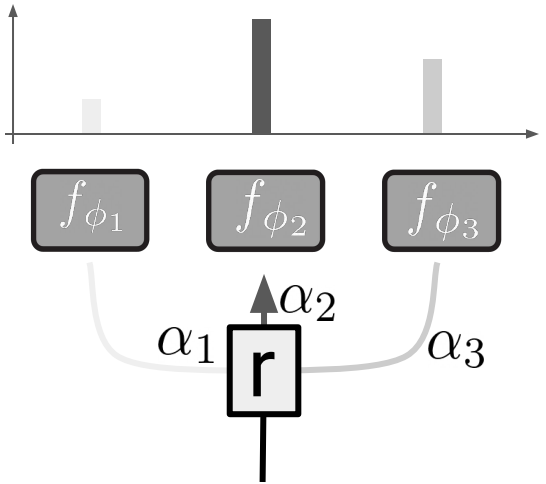


Soft Learned Routing

To sidestep ***discrete*** selections of modules, several works propose ***soft*** routing methods.

The router learns a probability distribution over the available modules:

$$p(\mathcal{F}) = r_\rho(\cdot)$$

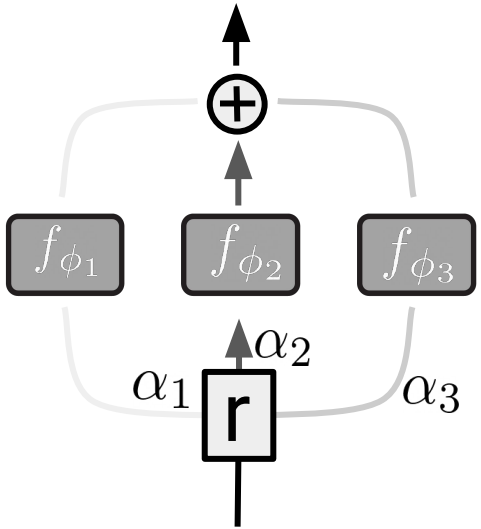


Jacobs et al. (1991); Jordan & Jacobs (1994)

Soft Learned Routing

Mixture of Experts (MoE)

$$f'_i(\mathbf{x}) = \sum_{\phi_j \in F} r(\phi_j)\, f(\mathbf{x}; \theta_i, \phi_j)$$



=> sum over all modules (aka “experts”)

Problem: All **modules** are **always activated**

=> Significant increase of **computational cost**.

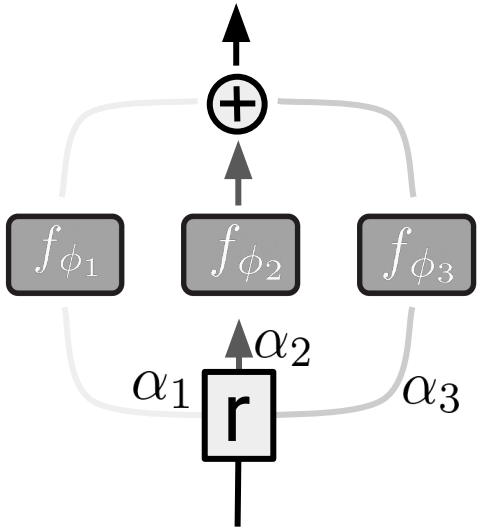
Eigen et al. (2014); Meyerson & Miikkulainen (2018); Wortsman et al. (2020)

Soft Learned Routing

Top-k routing.

Only select the top-k modules

$$f_i'(\mathbf{x}) = \sum_{\phi_j \in \text{top}_k[r(\boldsymbol{\phi})]} \frac{r(\phi_j)}{\sum r(\boldsymbol{\phi})} f(\mathbf{x}; \theta_i, \phi_j)$$



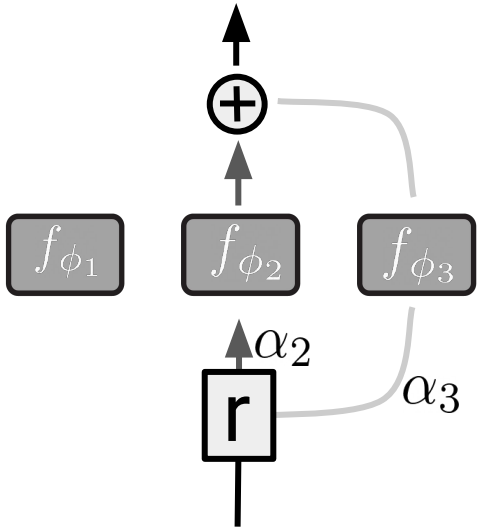
Shazeer et al. (2017); Lepikhin et al. (2021)

Soft Learned Routing

Top-k routing.

Only select the top-k modules

$$f_i'(\mathbf{x}) = \sum_{\phi_j \in \text{top}_k[r(\boldsymbol{\phi})]} \frac{r(\phi_j)}{\sum r(\boldsymbol{\phi})} f(\mathbf{x}; \theta_i, \phi_j)$$



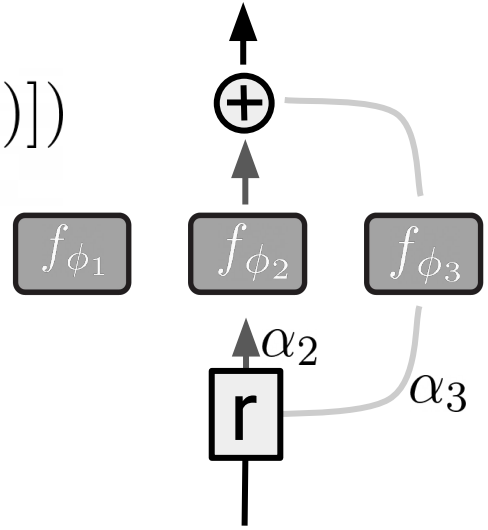
Shazeer et al. (2017); Lepikhin et al. (2021)

Soft Learned Routing

Top-1 routing.

Only select the top module

$$f_i'(\mathbf{x}) = \sum_{\phi_j \in \mathcal{F}} f(\mathbf{x}; \theta_i, \phi_j)\, 1_j(\text{argmax}[r(\phi)])$$



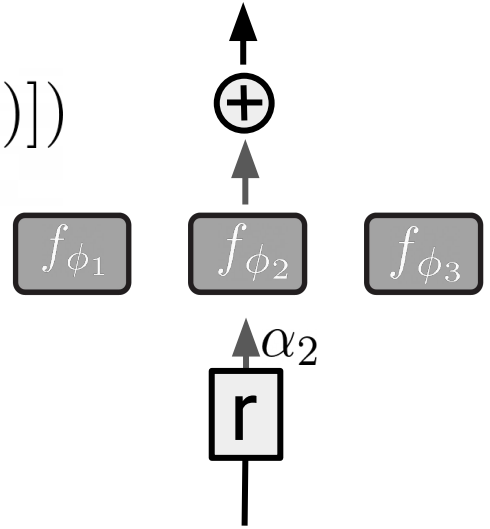
Fedus et al. (2021); Clark et al. (2022)

Soft Learned Routing

Top-1 routing.

Only select the top module

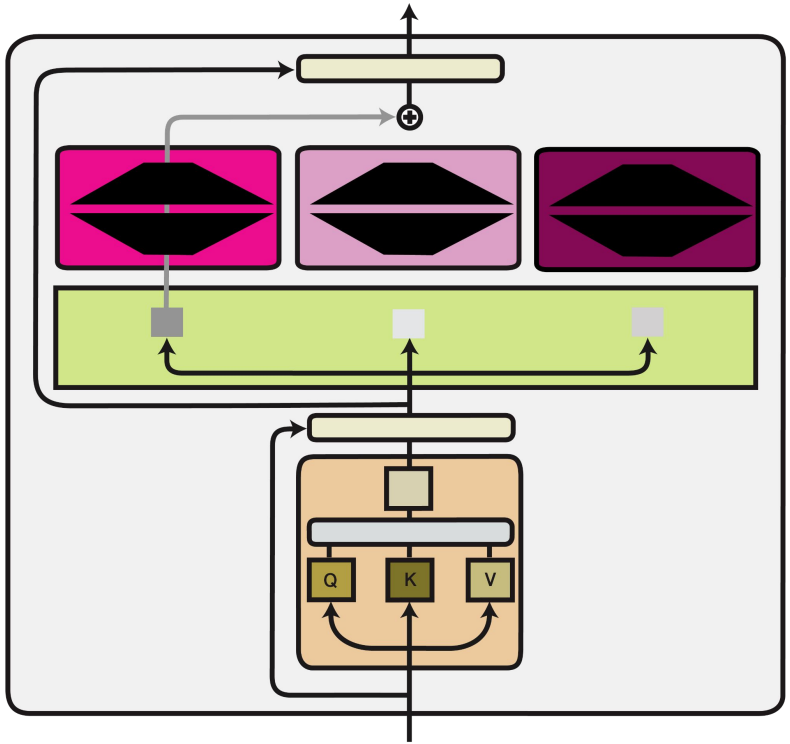
$$f_i'(\mathbf{x}) = \sum_{\phi_j \in \mathcal{F}} f(\mathbf{x}; \theta_i, \phi_j)\, 1_j(\text{argmax}[r(\phi)])$$



Fedus et al. (2021); Clark et al. (2022)

Token Level Routing - MoE

Each **FFN** component of the transformer is considered a **module** (aka “Expert”).

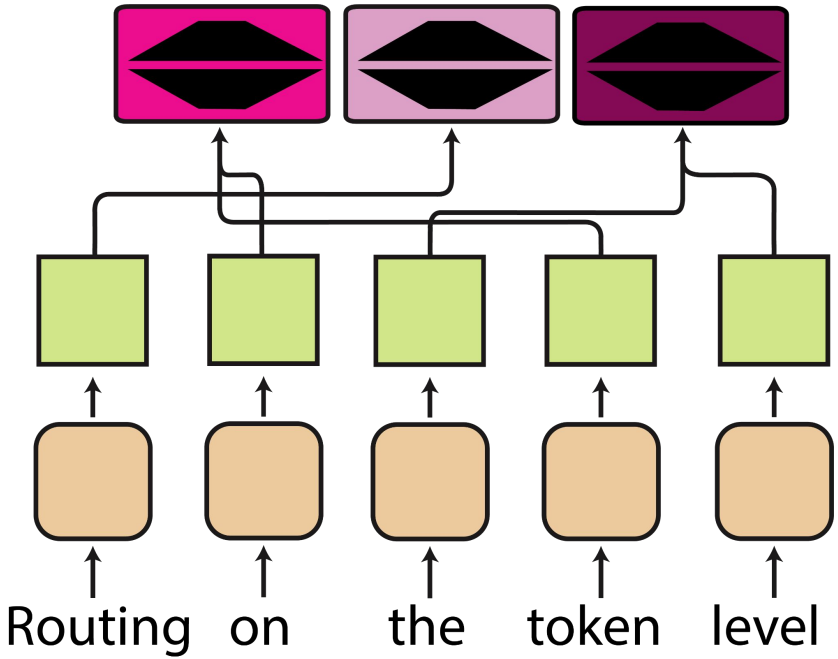


Shazeer et al. (2017); Lepikhin et al. (2021); Zhou et al. (2022); Lewis et al. (2021); Fedus et al. (202

Token Level Routing - MoE

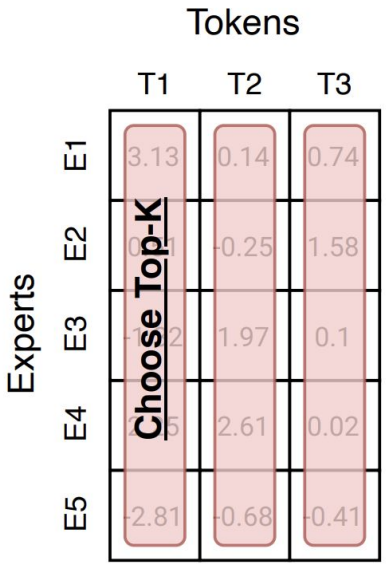
Each **FFN** component of the transformer is considered a **module** (aka “Expert”).

Routing is performed on the **token** level.

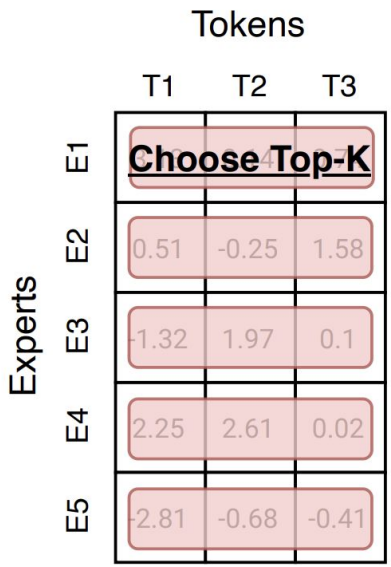


Shazeer et al. (2017); Lepikhin et al. (2021); Zhou et al. (2022); Lewis et al. (2021); Fedus et al. (202

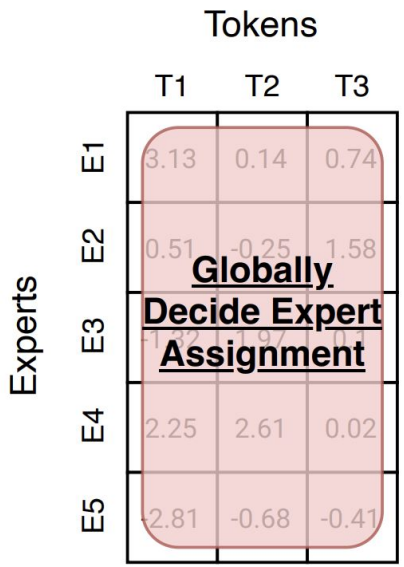
Token Level Routing - Load Balancing



[Shazeer et al. (2017)](); [Lepikhin et al. (2021)]()



[Zhou et al. (2022)]()



[Lewis et al. (2021)]()

Illustration in courtesy of [Fedus et al. (2022)]()

Token Level Routing

Problem: Load balancing restricts the system from routing an entire example to a single module.

Syntactically and semantically similar **words** (in contrast to sentences or phrases) are routed to to the same modules.

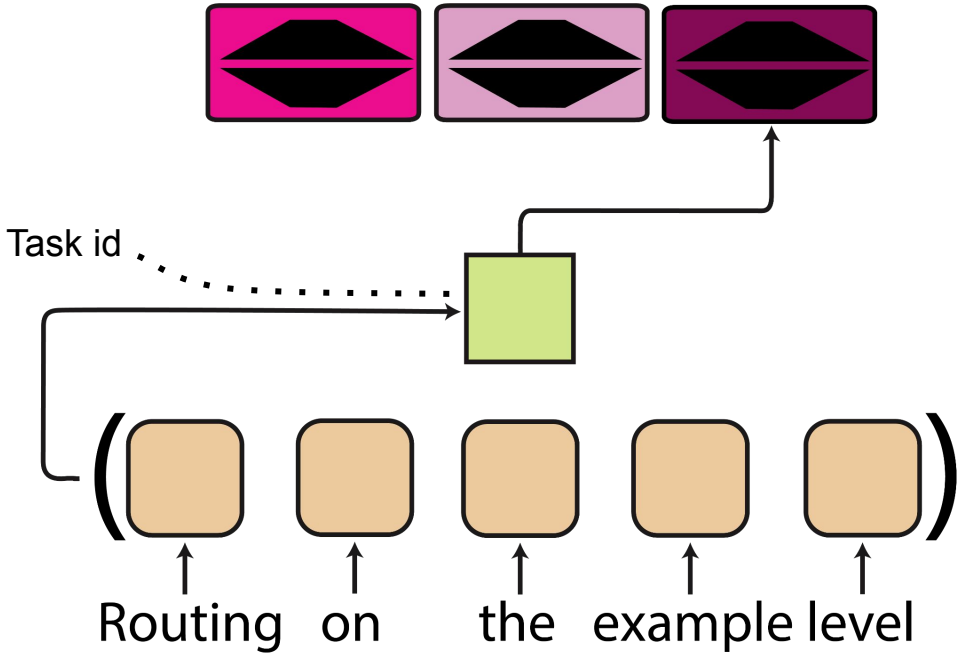
Lewis et al. (2021)

Expert	Top 5 Proceeding Tokens
5	*year, years, billion, million, tonnes*
8	*people, who, Man, everyone, one*
9	*electronic, local, public, national, outdoor*
23	*funding, budget, benefits, pressure, price*
27	*Mustang, State, Center, ation, Grande*
34	*to, will, should, it, may*
36	*business, bank, financial, science, school*
42	*two, 50, 1, 80, 000*
43	*Bank, Development, ., Construction, Plant*
62	*work, started, involved, working, launched*
72	*is, was, be, been, were*
74	*going, go, come, back, return*
76	*painting, case, song, statement, discussion*
81	*new, major, bad, larger, grand*
84	*Ret, Inspect, Pl, Pos, Architect*
96	*US, UNESCO, government, state, UN*
98	*waiver, procedures, warrant, status, loans*
101	*B, T, W, H, k*
105	*app, Windows, Microsoft, board, 10*
125	*his, 's, its, their, our*
126	*said, says, means, noting, out*

Example Level Routing

Instead of token level routing, each token of a **sentence** can be routed to the same module.

Routing can be achieved based on metadata, such as **language** or **task id**, or on the **pooled token representations**.



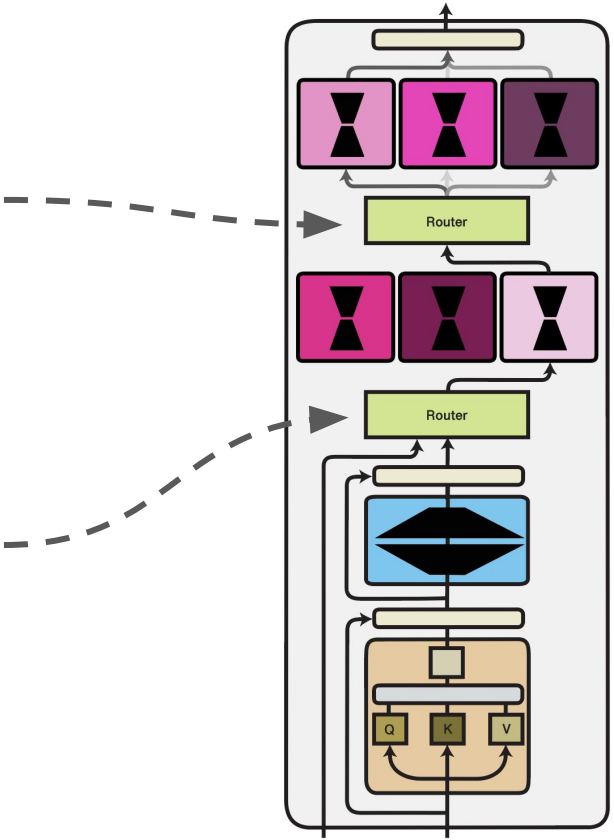
Kudugunta et al. (2021); Gupta et al. (2022); Xi et al. (2022)

Routing - Hybrids

It is possible to combine the concepts of **fixed** and **learned routing.**

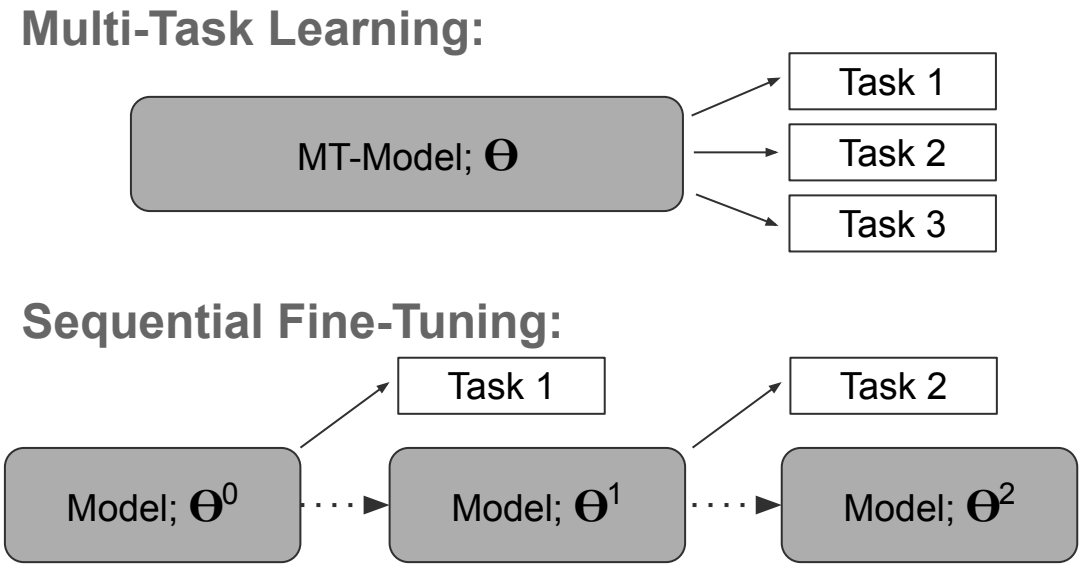
Learned Routing (i.e. MoE)
e.g. heterogeneous data

Fixed Routing
e.g. Language_id



Properties of Modular and Parameter-Efficient Tuning

- **Avoiding negative consequences** of full-model fine-tuning:
  - Bypassing catastrophic forgetting and interference



Properties of Modular and Parameter-Efficient Tuning

- **Avoiding negative consequences** of full-model fine-tuning:
  - Bypassing catastrophic forgetting and interference
  - Bypassing the creation of large full-model copies; boosting **efficiency** and **compactness**

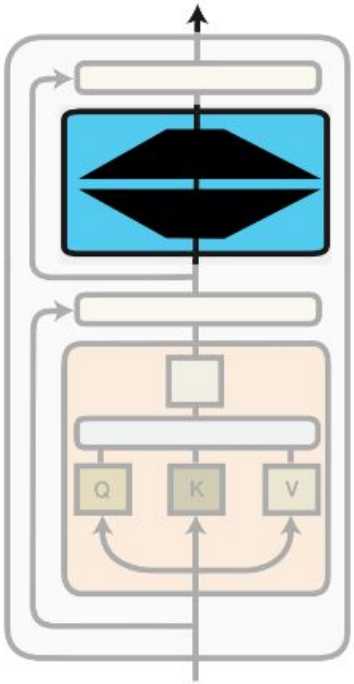
- **Positive aspects:**
  - **Positive transfer**
  - Compositionality and reusability of modules: **systematic generalisation**
  - **Local, asynchronous updates**: **parameter efficiency**
  - **Scaling** (e.g., through MoE) and **specialisation**

We will illustrate most of these properties via applications in (low-resource) multilingual NLP as a case study.

Parameter-Efficient Fine-Tuning

A basic scenario, seen many times across many applications

- Fix/freeze the base model
- Initialise a **task-specific module**
- All fine-tuning updates **are forced into** the task-specific module
- Compare against full-model fine-tuning and preceding PEFT approaches (on GLUE?)
- Choose **which benefit** you want to stress
  - *Improve performance with the same parameter budget*
  - *Maintain performance with a smaller parameter budget*



Acknowledgements

*This lecture is heavily based on EMNLP2022 tutorial “Modular and Parameter-Efficient Fine-Tuning for NLP Models” by **Sebastian Ruder**, **Jonas Pfeiffer** and **Ivan Vulic***

Sources: [1], [2]

Questions?