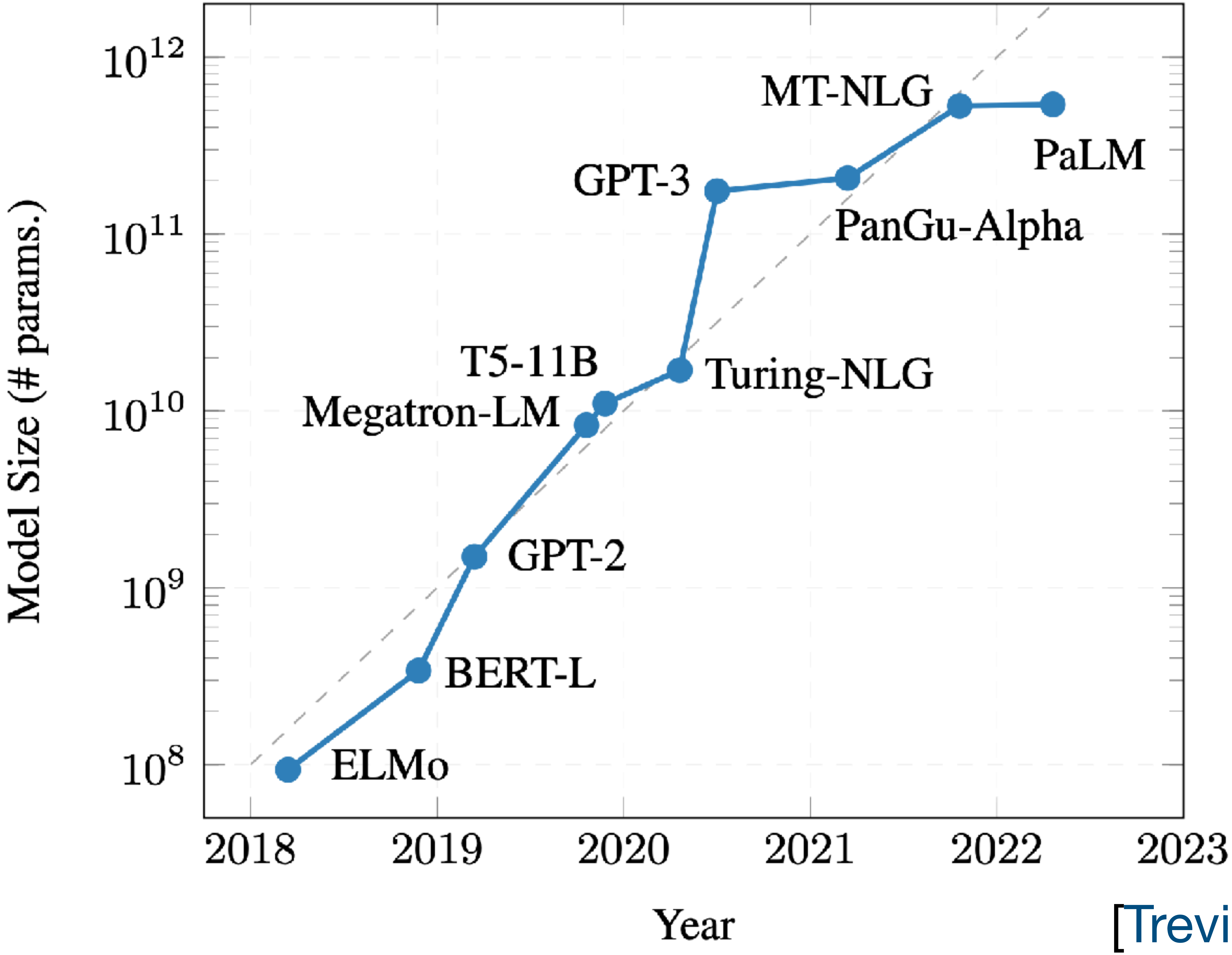
Natural Language Understanding, Generation, and Machine Translation

Lecture 27: Parameter-Efficient Fine-Tuning

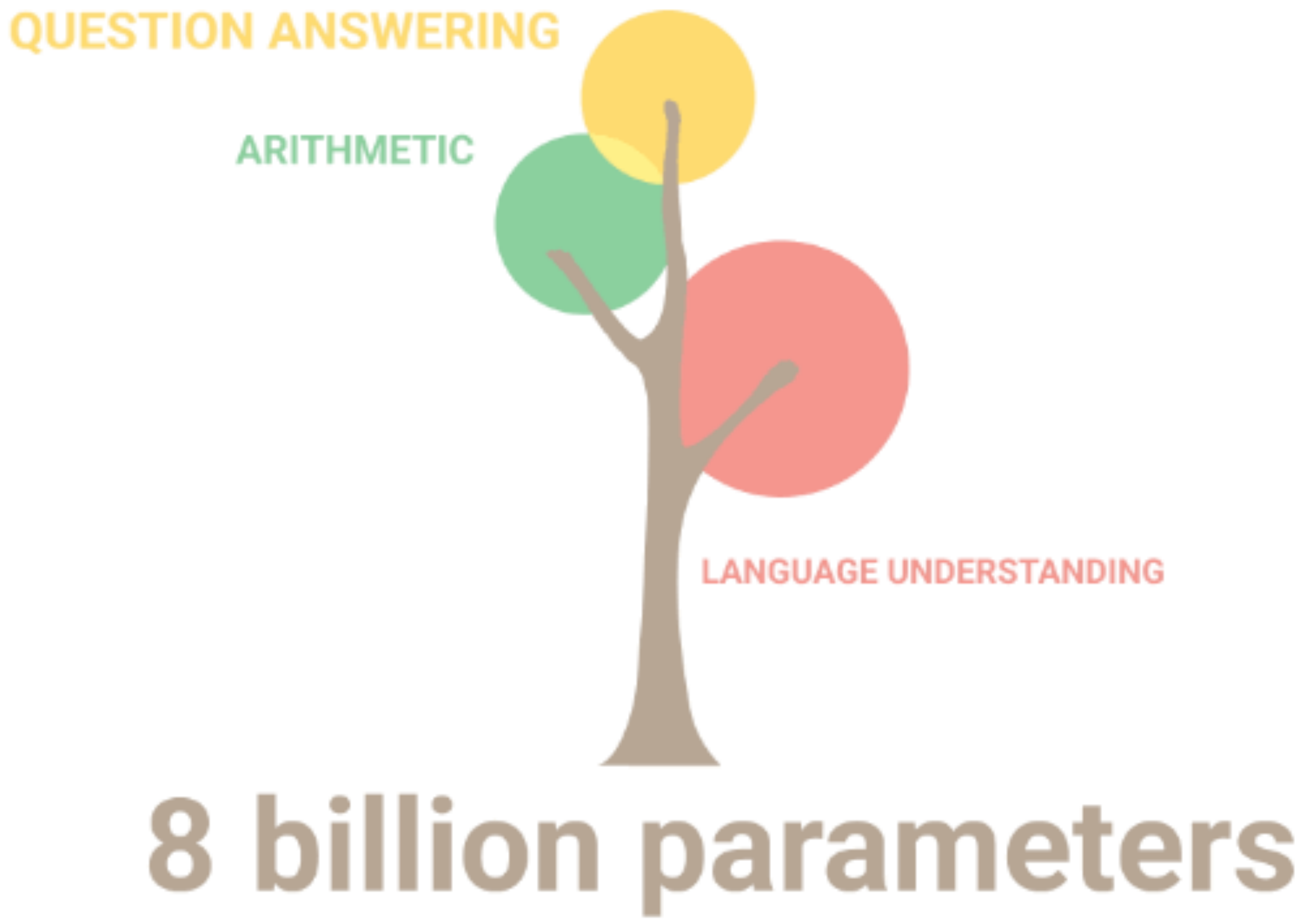
Pasquale Minervini
p.minervini@ed.ac.uk
March 22nd, 2024

Evolution of Pre-Trained Language Models



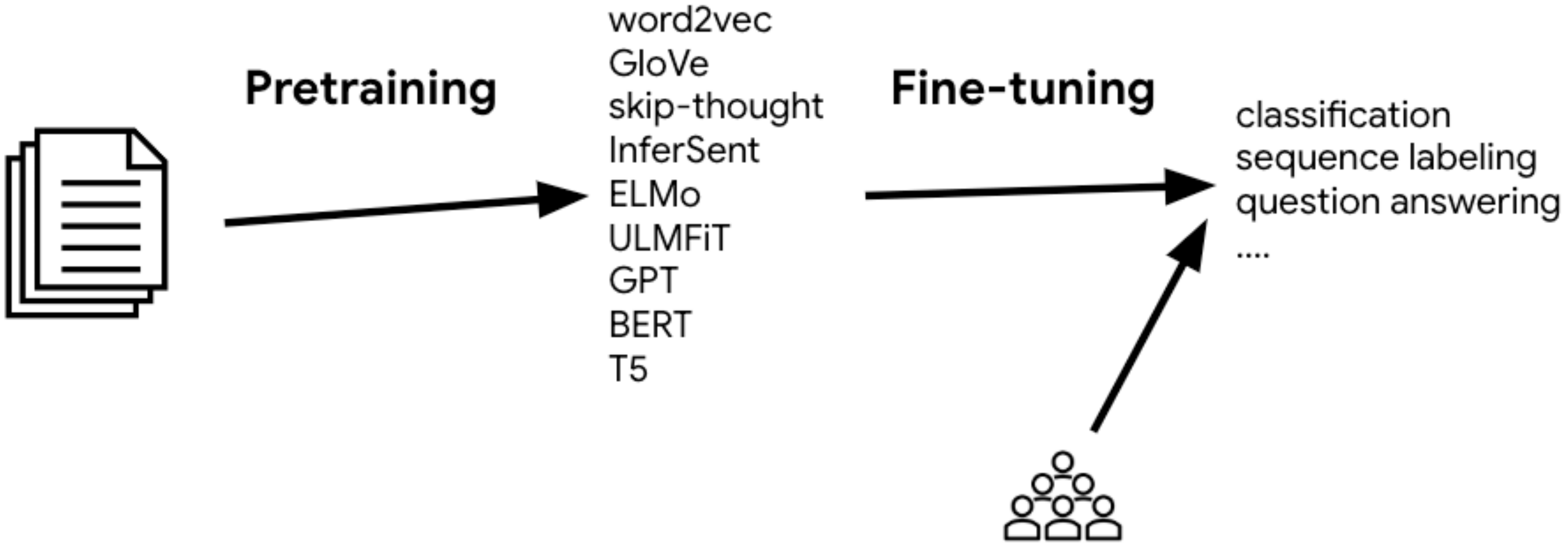
[Treviso et al., 2022]

Evolution of Pre-Trained Language Models



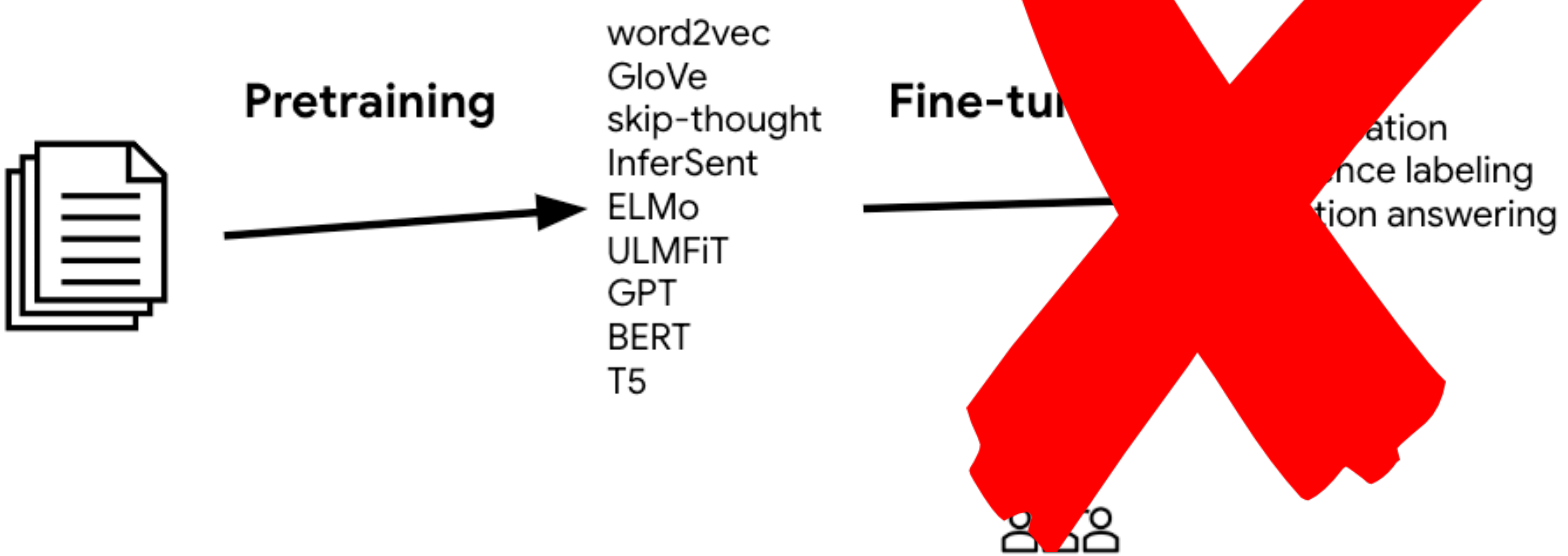
Transfer Learning in the Era of LLMs

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



Transfer Learning in the Era of LLMs

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



In-Context Learning

In-Context Learning — Downsides

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

In-Context Learning — Downsides

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]

In-Context Learning — Downsides

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]

Sensitivity to the wording of the prompt [Webson & Pavlick, 2022], order of examples — e.g., see [Zhao et al., 2021; Lu et al., 2022]

In-Context Learning — Downsides

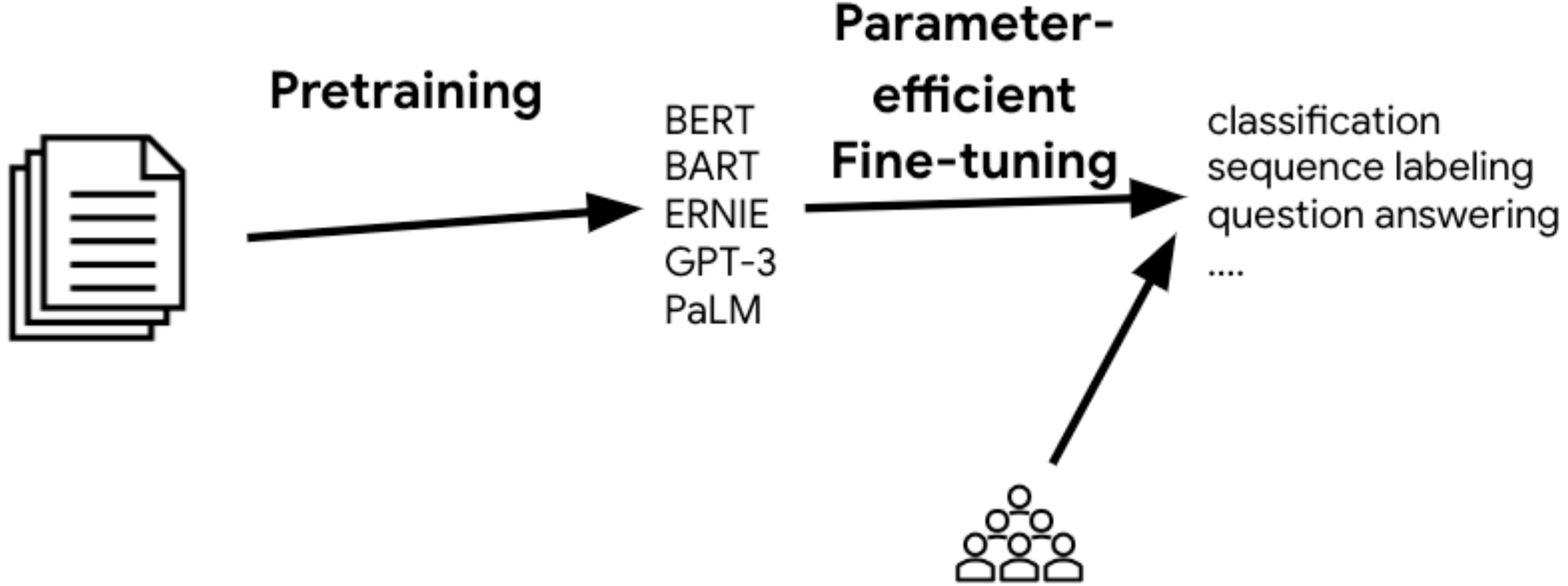
Inefficiency: the prompt needs to be processed *every time* the model makes a prediction

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

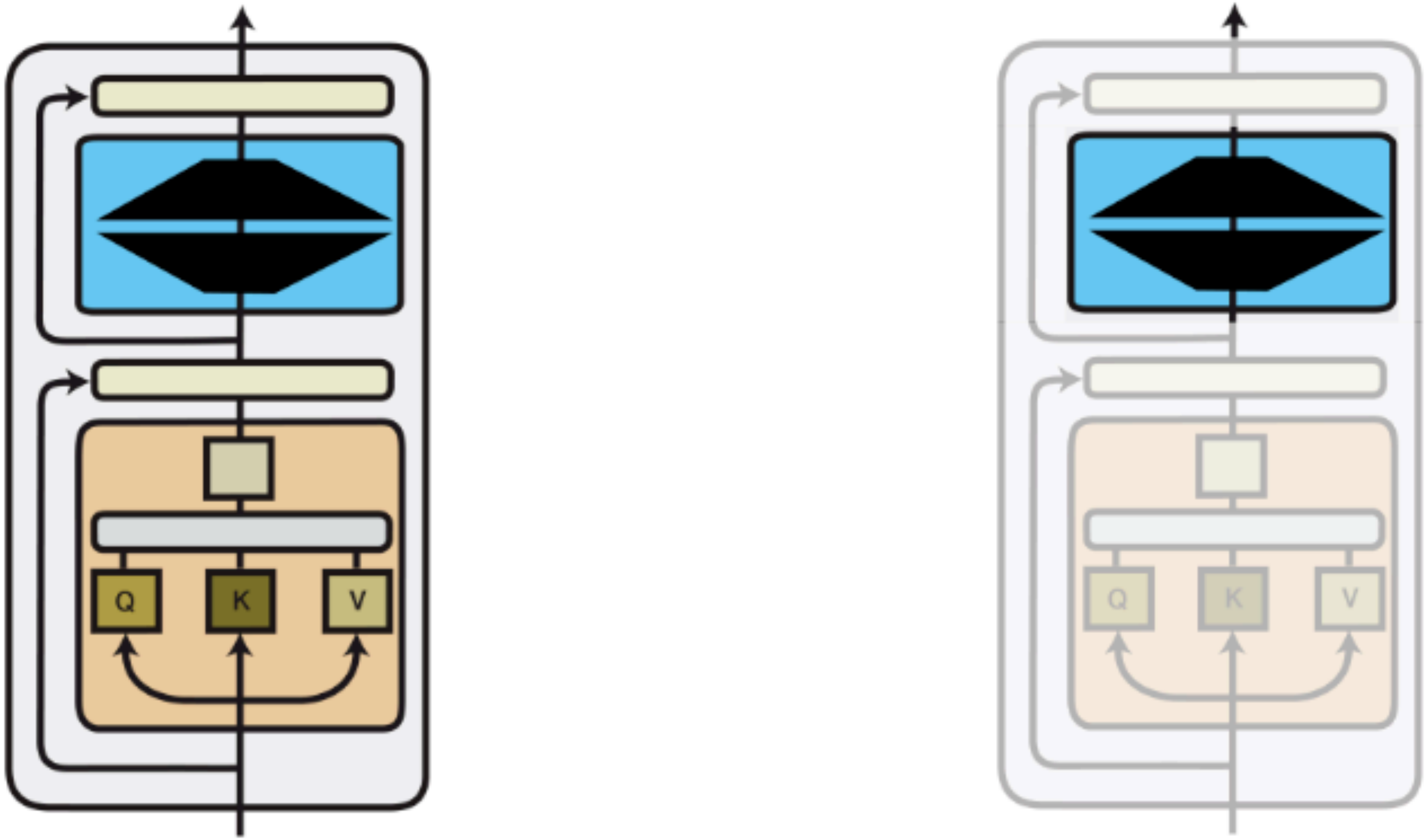
Sensitivity to the wording of the prompt [Webson & Pavlick, 2022], order of examples — e.g., see [Zhao et al., 2021; Lu et al., 2022]

Lack of clarity regarding what the model learns from the prompt — even random label can provide non-trivial results [Min et al., 2022]!

Fine-Tuning → Parameter-Efficient Fine-Tuning



Fine-Tuning → Parameter-Efficient Fine-Tuning



Full Fine-tuning
Update **all model parameters**

Parameter-efficient Fine-tuning
Update a **small subset** of model parameters

Fine-Tuning → Parameter-Efficient Fine-Tuning

Parameter-Efficient Fine-Tuning (PEFT) is not really a new idea!

- Updating the last layer of the model was common in computer vision [Donahue et al., 2014]. In NLP, people experimented with static (frozen) and non-static (trainable) [Kim, 2014]
- ELMo did not fine-tune word embeddings [Peters et al., 2018]

Fine-Tuning → Parameter-Efficient Fine-Tuning

Parameter-Efficient Fine-Tuning (PEFT) is not really a new idea!

- Updating the last layer of the model was common in computer vision [Donahue et al., 2014]. In NLP, people experimented with static (frozen) and non-static (trainable) [Kim, 2014]
- ELMo did not fine-tune word embeddings [Peters et al., 2018]

In practice, **fine-tuning everything** seems to work better in practice — why go back o fine-tuning **only** some parameters?

- Fine-tuning everything is **impractical** with large models
- LLMs nowadays are massively over-parameterised — PEFT matches full fine-tuning in downstream accuracy

Some Notation

Let $f_\theta : \mathscr{X} \mapsto \mathscr{Y}$ be a neural network, which can be decomposed into a **composition of functions** $f_{\theta_1} \odot f_{\theta_2} \odot \ldots \odot f_{\theta_n}$, where each function has parameters θ_i with $i \in \{1,\ldots,n\}$.

A module with parameters ϕ can modify a function f_{θ_i} as follows:

Some Notation

Let $f_\theta : \mathscr{X} \mapsto \mathscr{Y}$ be a neural network, which can be decomposed into a **composition of functions** $f_{\theta_1} \odot f_{\theta_2} \odot \ldots \odot f_{\theta_n}$, where each function has parameters θ_i with $i \in \{1,\ldots,n\}$.

A module with parameters ϕ can modify a function f_{θ_i} as follows:

- **Parameter composition**: $g_i = f_{\theta_i \oplus \phi}(x)$

Interpolation — e.g., element-wise addition

Some Notation

Let $f_\theta : \mathcal{X} \mapsto \mathcal{Y}$ be a neural network, which can be decomposed into a **composition of functions** $f_{\theta_1} \odot f_{\theta_2} \odot \ldots \odot f_{\theta_n}$, where each function has parameters θ_i with $i \in \{1,\ldots,n\}$.

A module with parameters ϕ can modify a function f_{θ_i} as follows:

- **Parameter composition**: $g_i = f_{\theta_i \oplus \phi}(x)$
- **Input composition**: $g_i(x) = f_{\theta_i}\left(\left[x, \phi\right]\right)$

Concatenation

Interpolation — e.g.,
element-wise addition

Some Notation

Let $f_\theta : \mathscr{X} \mapsto \mathscr{Y}$ be a neural network, which can be decomposed into a **composition of functions** $f_{\theta_1} \odot f_{\theta_2} \odot \ldots \odot f_{\theta_n}$, where each function has parameters θ_i with $i \in \{1,\ldots,n\}$.

A module with parameters ϕ can modify a function f_{θ_i} as follows:

- **Parameter composition**: $g_i = f_{\theta_i \oplus \phi}(x)$

  Interpolation — e.g., element-wise addition

- **Input composition**: $g_i(x) = f_{\theta_i}\Big(\big[x, \phi\big]\Big)$

  Concatenation

- **Function composition**: $g_i(x) = f_{\theta_i} \odot f_\phi(x)$

  Composition

Some Notation

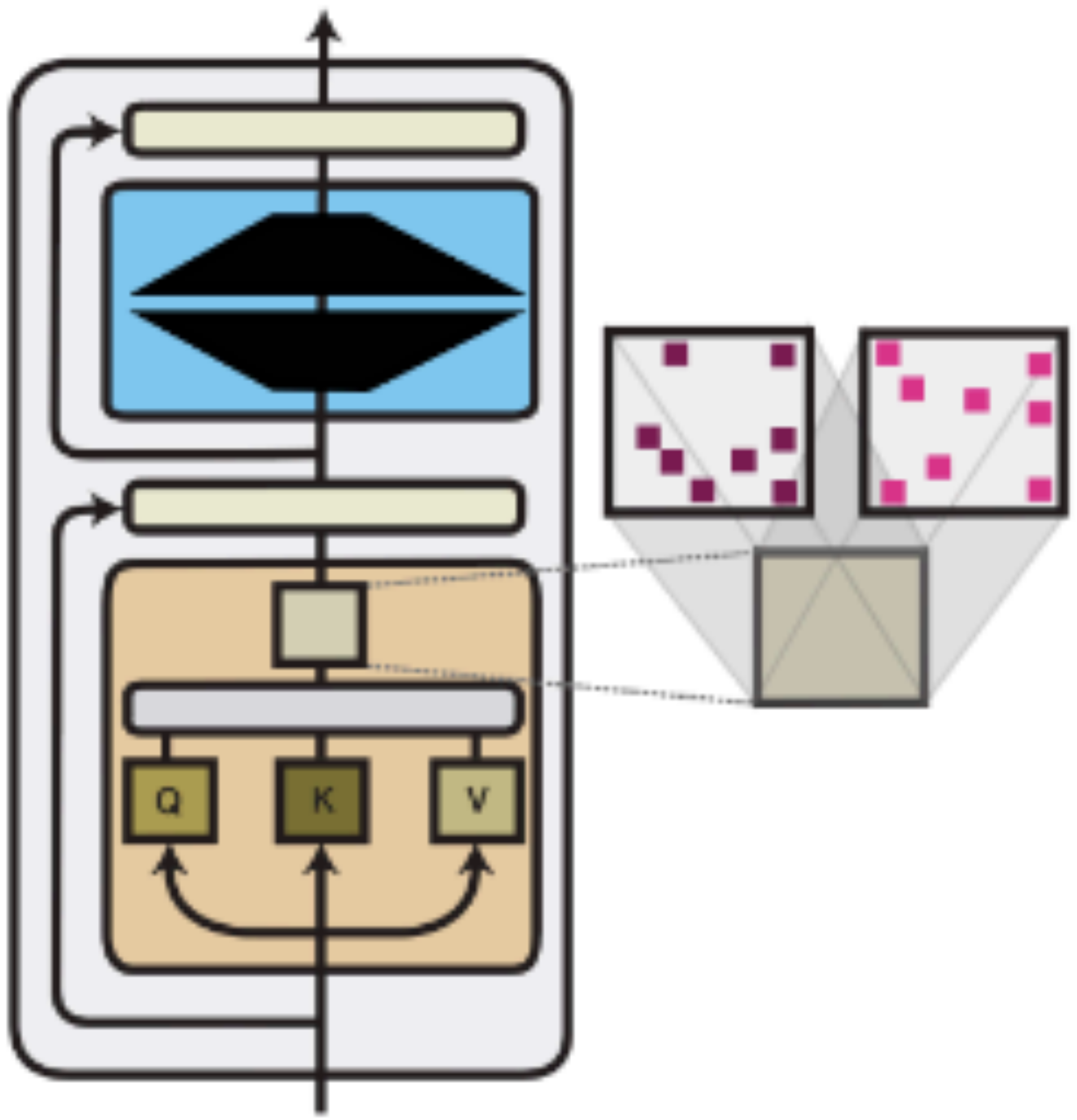
Let $f_\theta : \mathscr{X} \mapsto \mathscr{Y}$ be a neural network, which can be decomposed into a **composition of functions** $f_{\theta_1} \odot f_{\theta_2} \odot \ldots \odot f_{\theta_n}$, where each function has parameters θ_i with $i \in \{1,\ldots,n\}$.

A module with parameters ϕ can modify a function f_{θ_i} as follows:

- **Parameter composition**: $g_i = f_{\theta_i \oplus \phi}(x)$ Interpolation — e.g., element-wise addition
- **Input composition**: $g_i(x) = f_{\theta_i}\left(\underbrace{[x, \phi]}_{\text{Concatenation}}\right)$
- **Function composition**: $g_i(x) = f_{\theta_i} \odot f_\phi(x)$

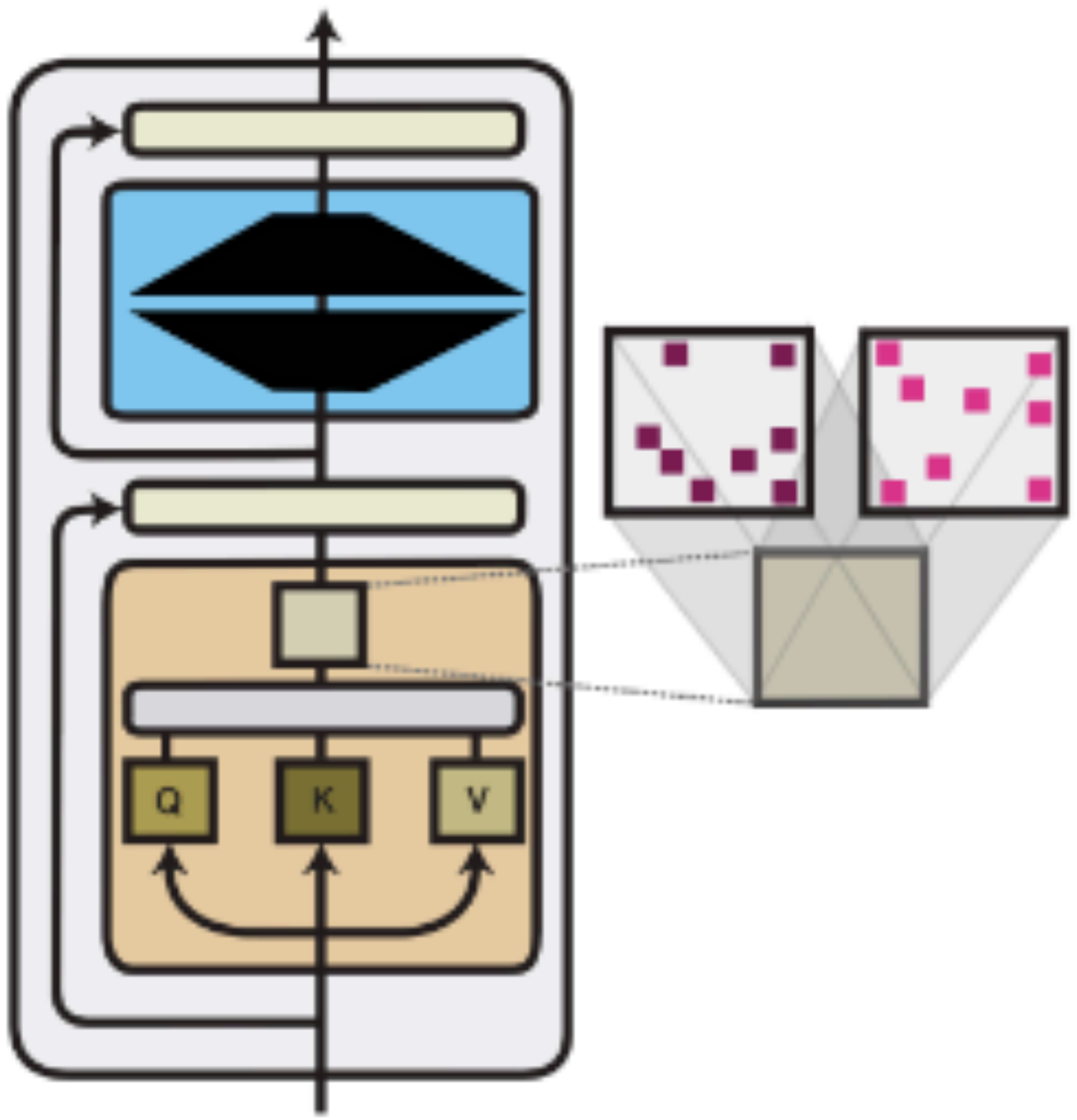
Typically, only module parameters ϕ are updated while θ is fixed

Composition Functions

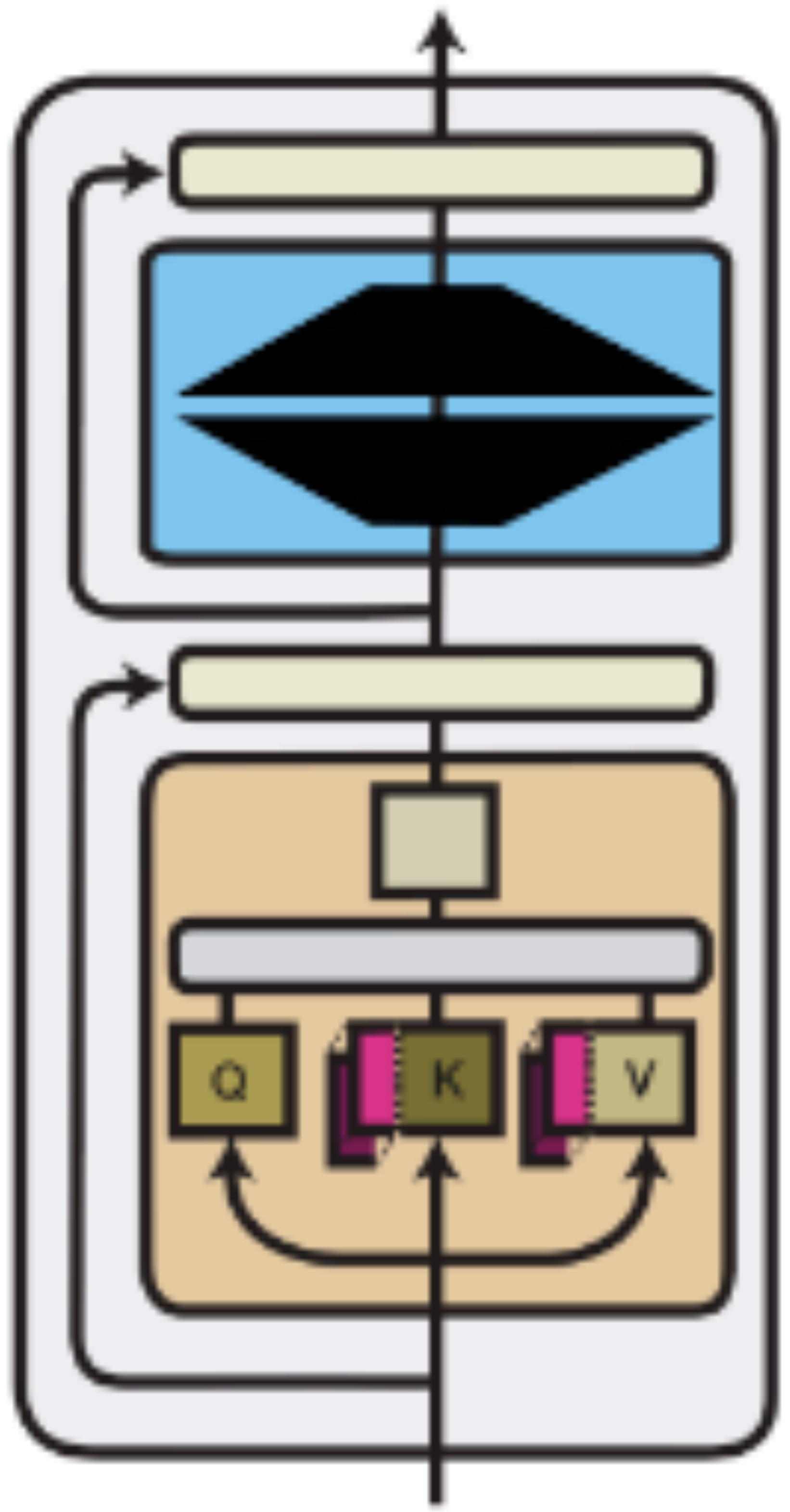


Parameter Composition

Composition Functions

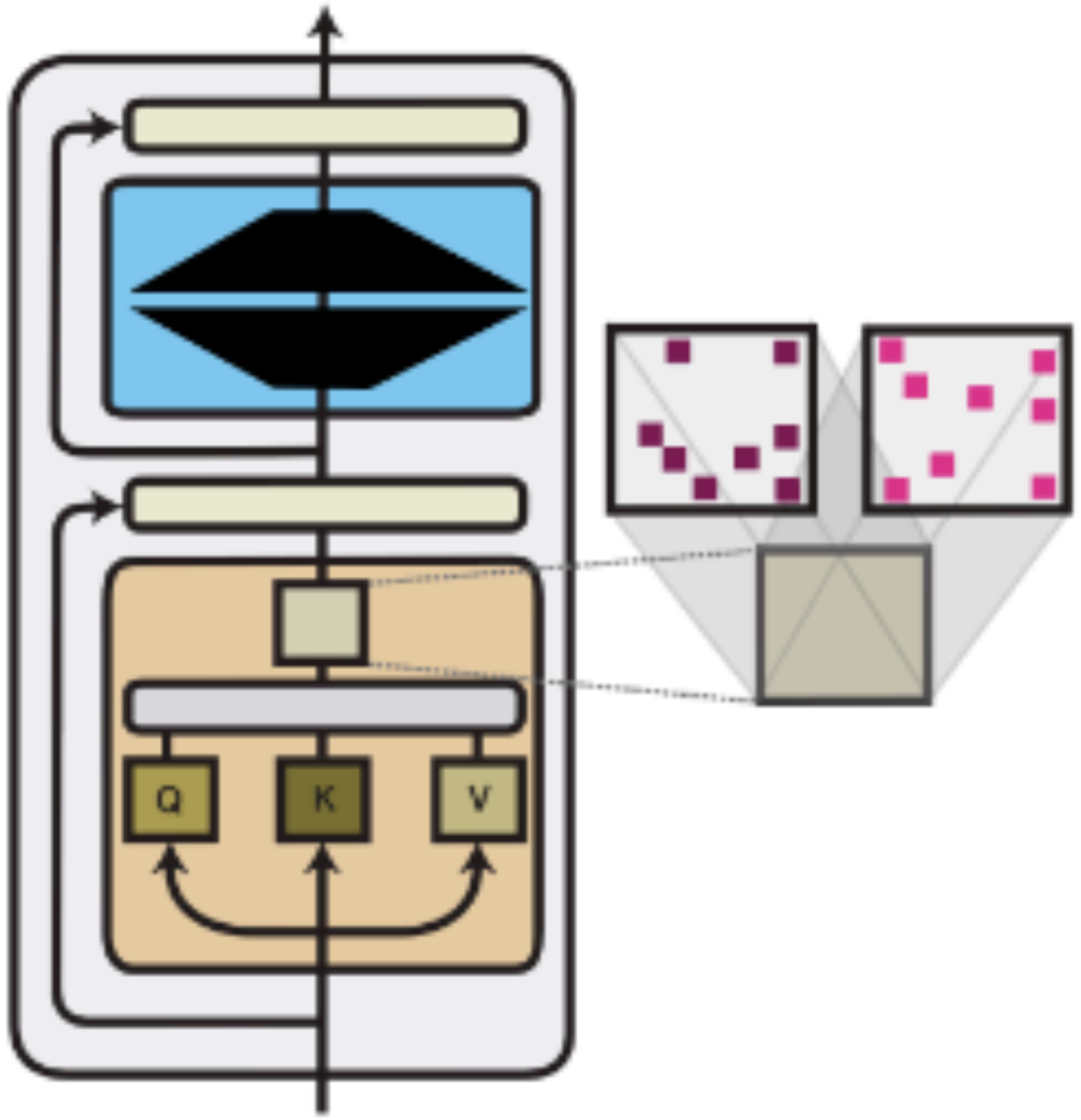


Parameter Composition

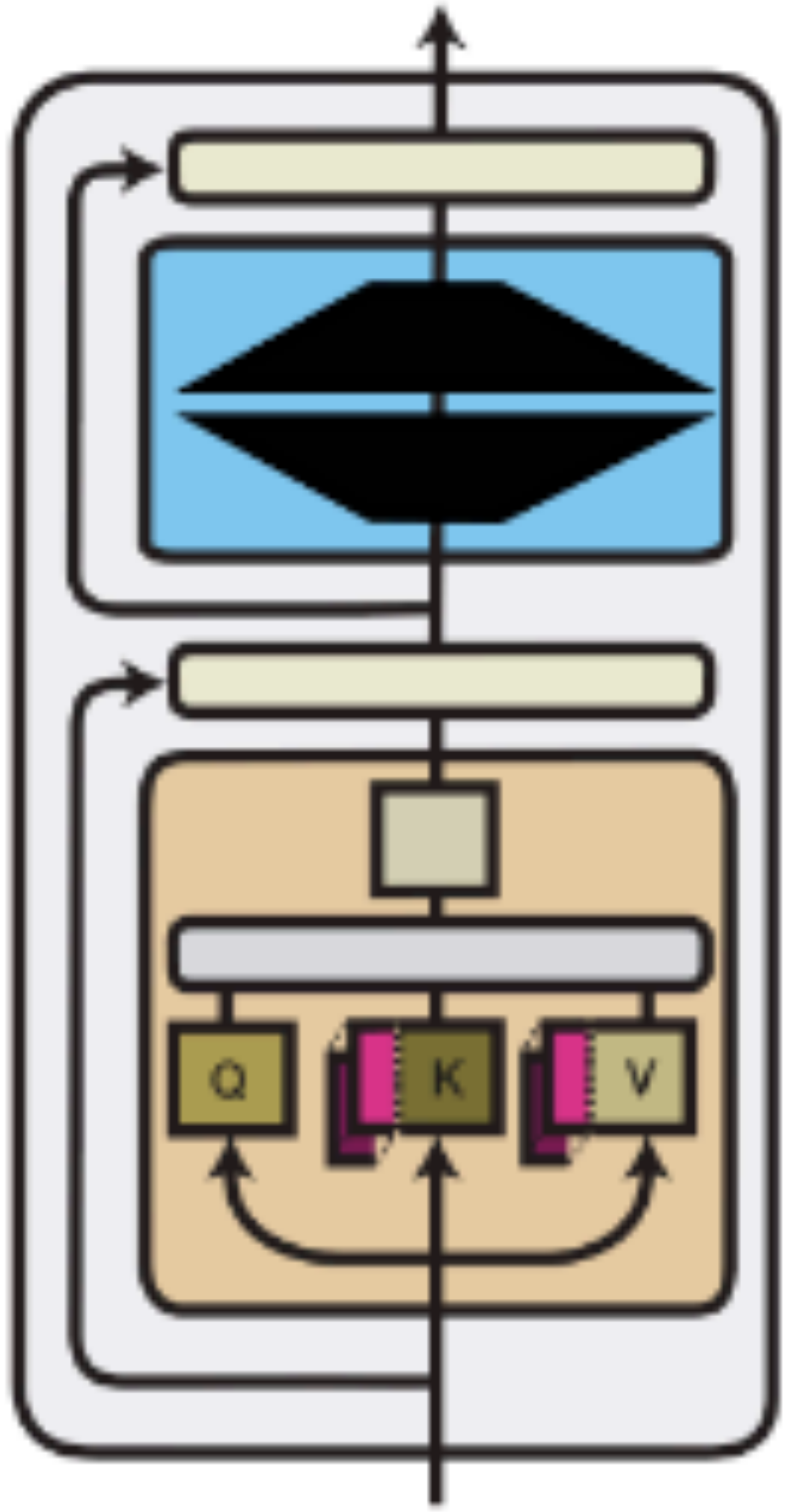


Input Composition

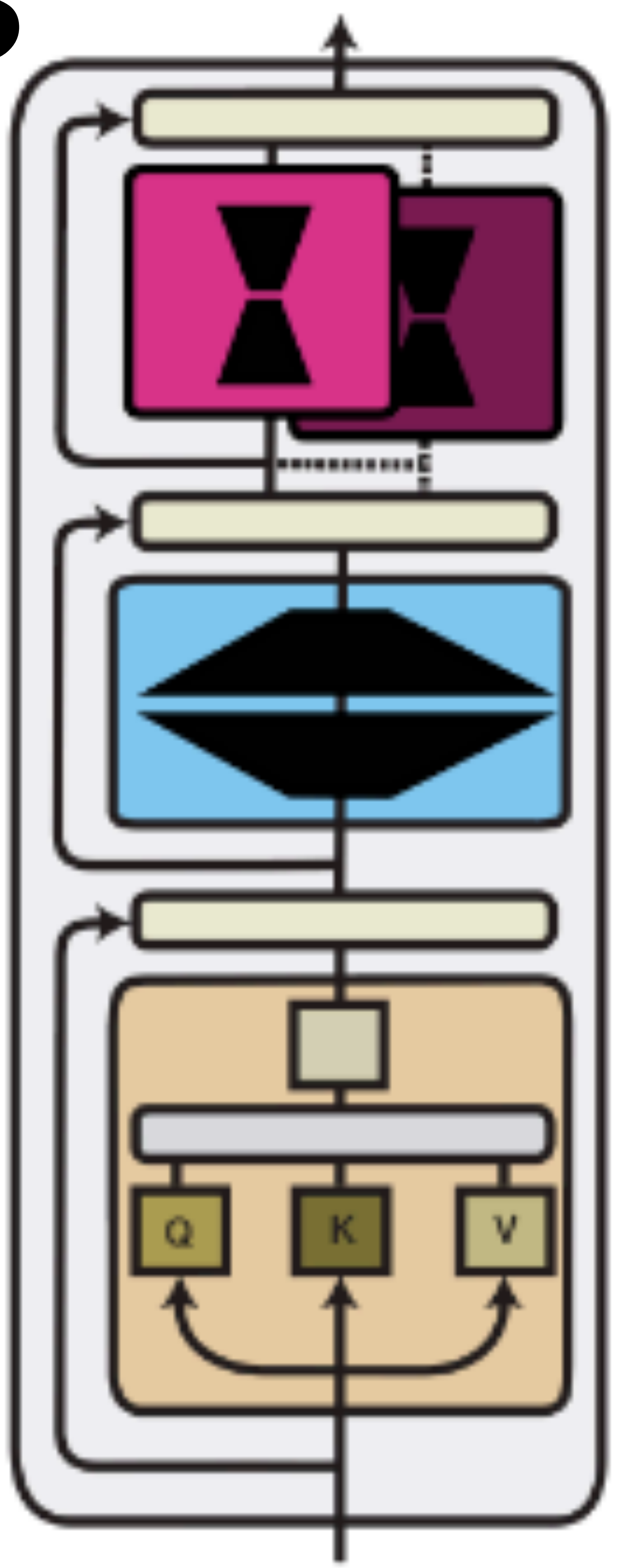
Composition Functions



Parameter Composition

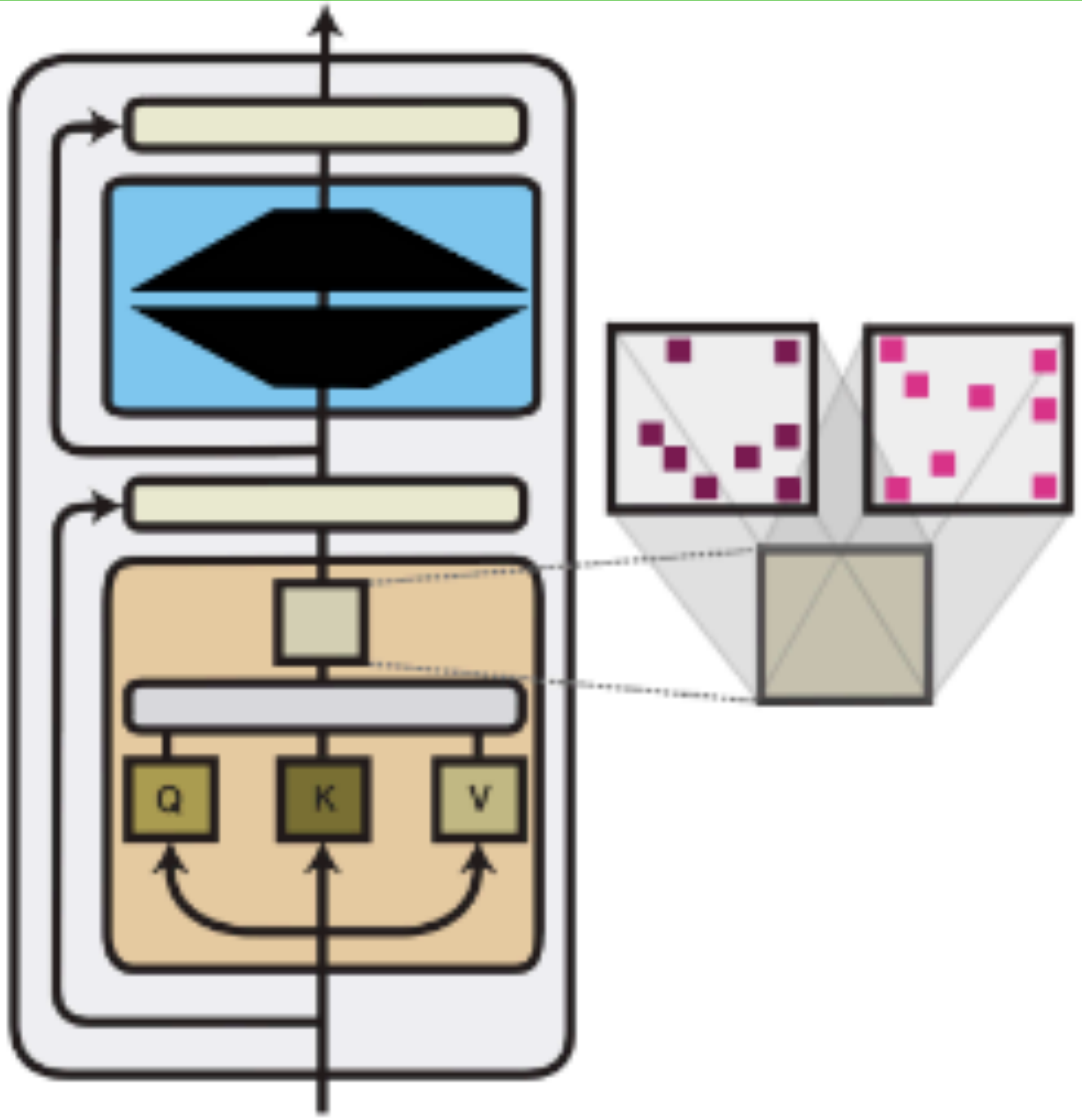


Input Composition

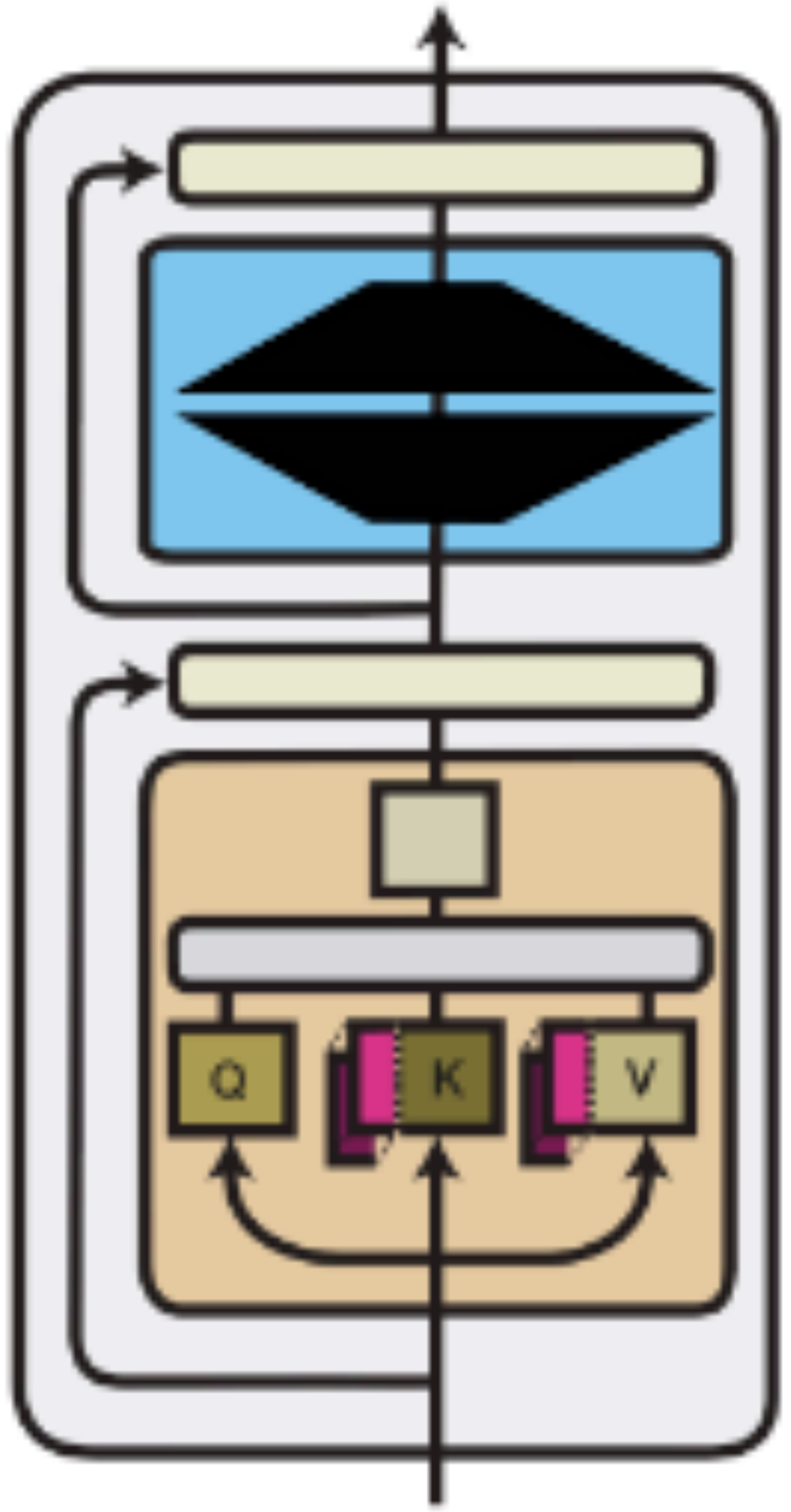


Function Composition

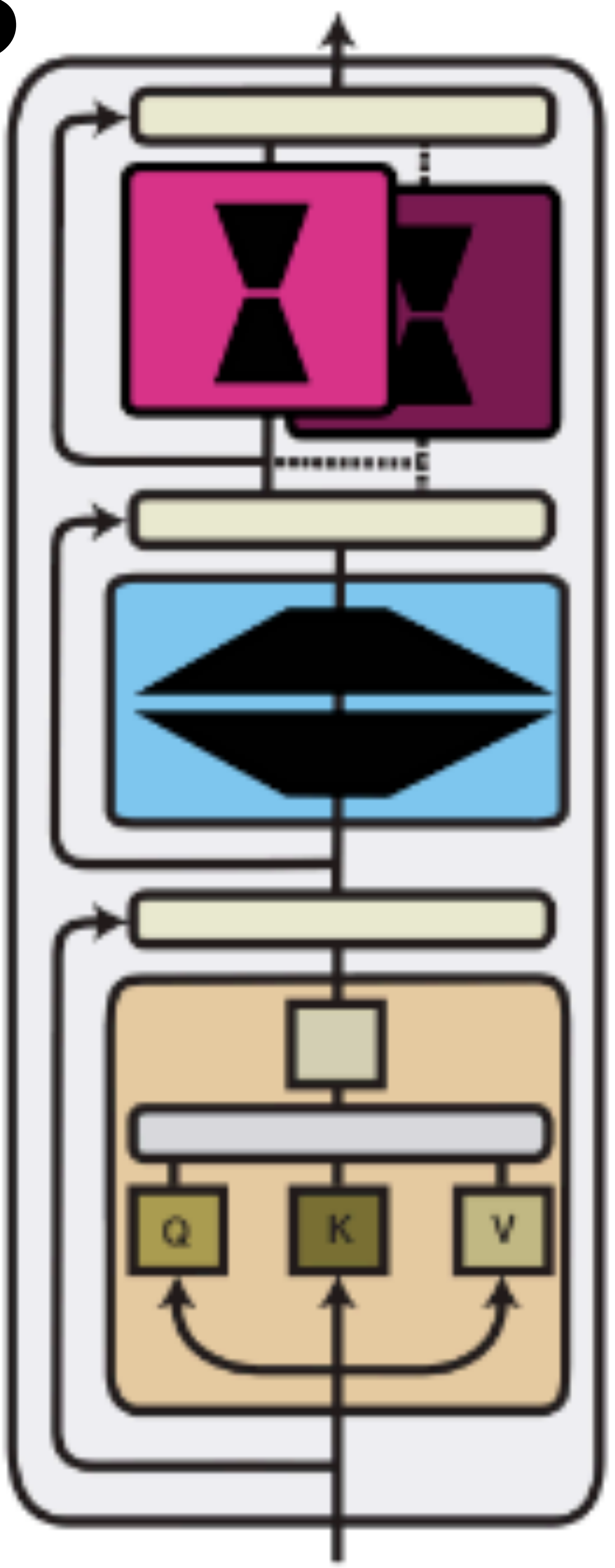
Composition Functions



Parameter Composition



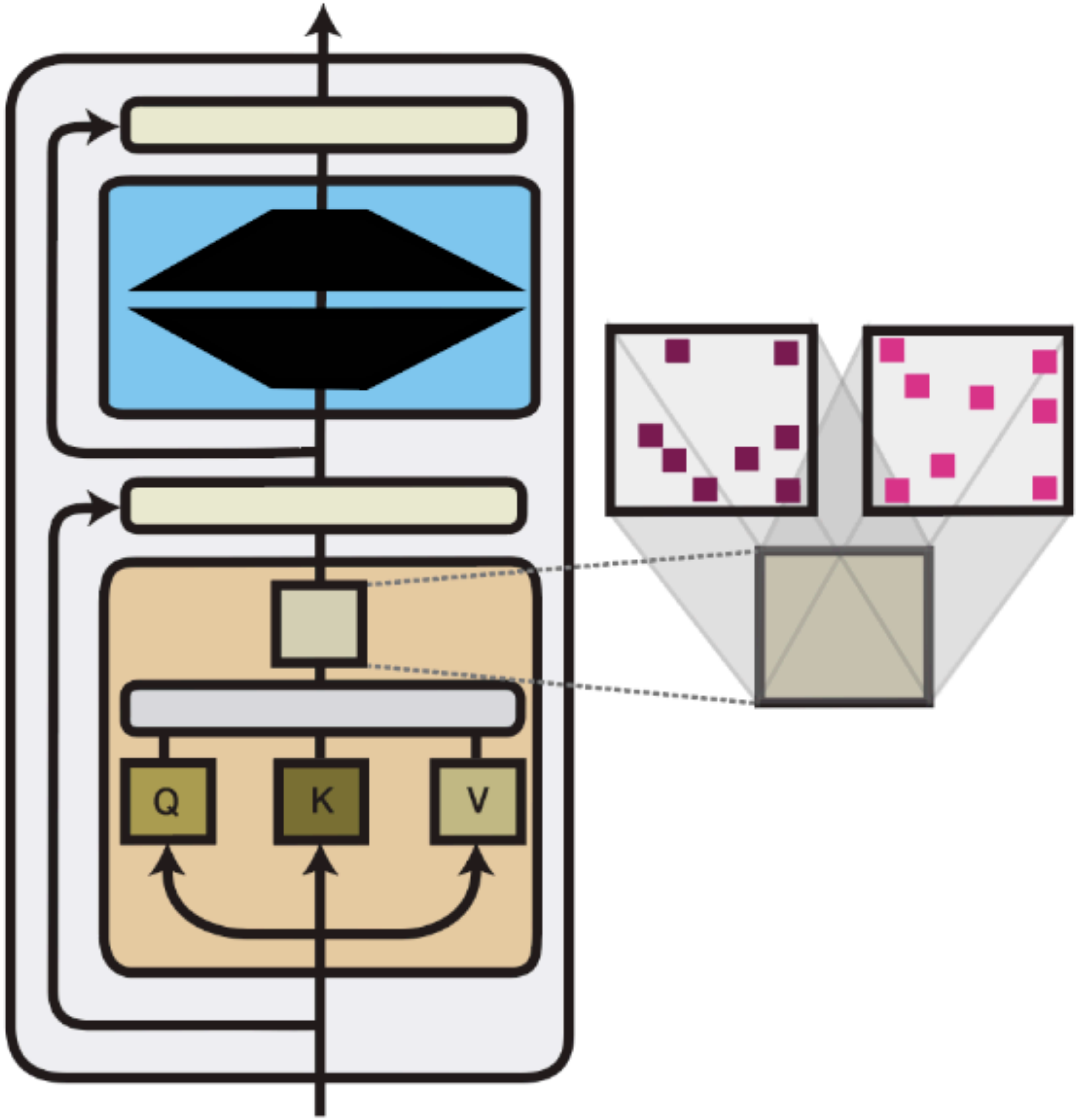
Input Composition



Function Composition

Parameter Composition

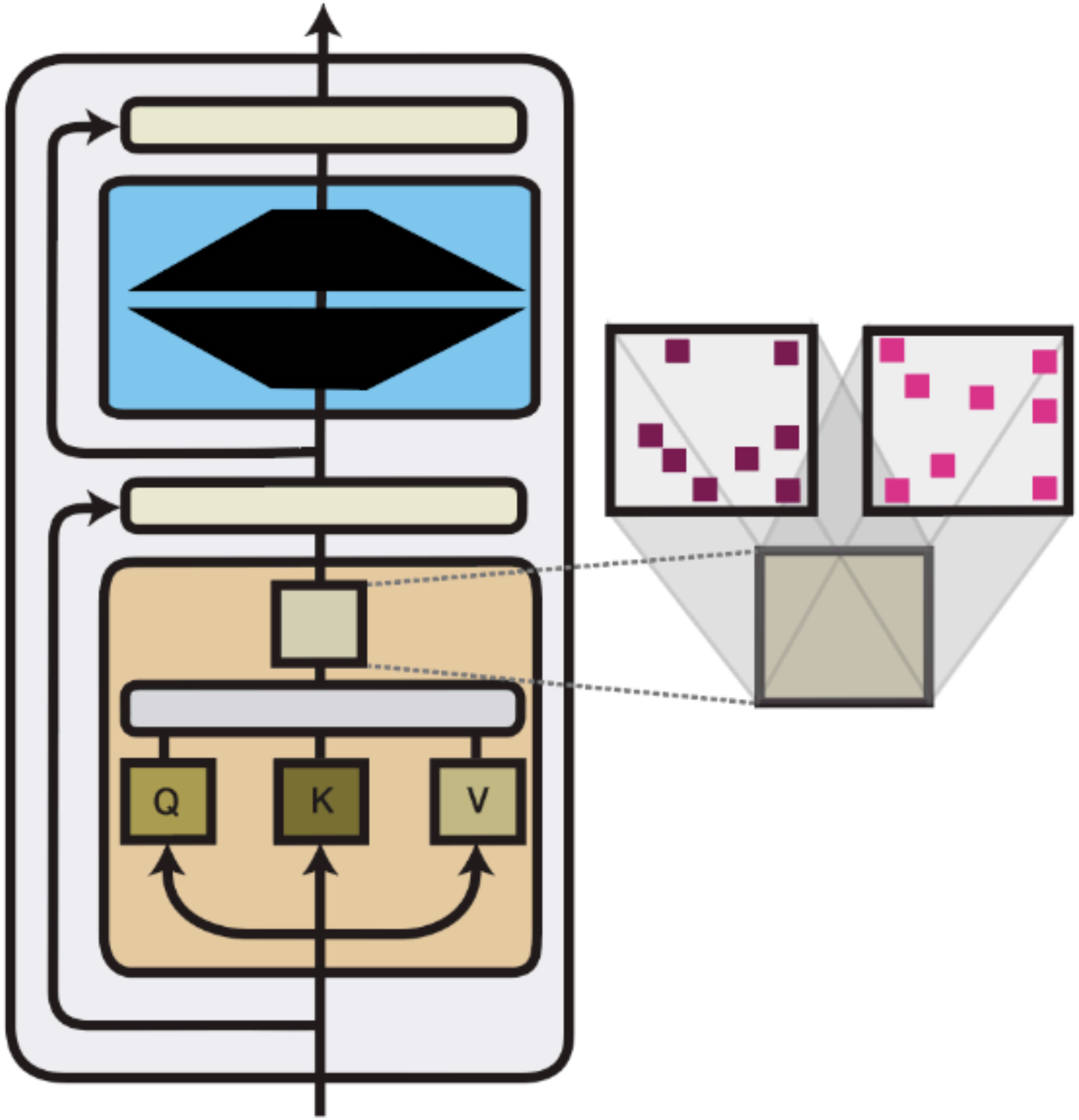
Sparse Subnetworks, where module parameters ϕ are enforced to be *sparse*



Parameter Composition

Sparse Subnetworks, where module parameters ϕ are enforced to be *sparse*

Structured Composition, where we impose a structure on the weights θ_i that we select — e.g., we update the weights belonging to a *pre-defined group*

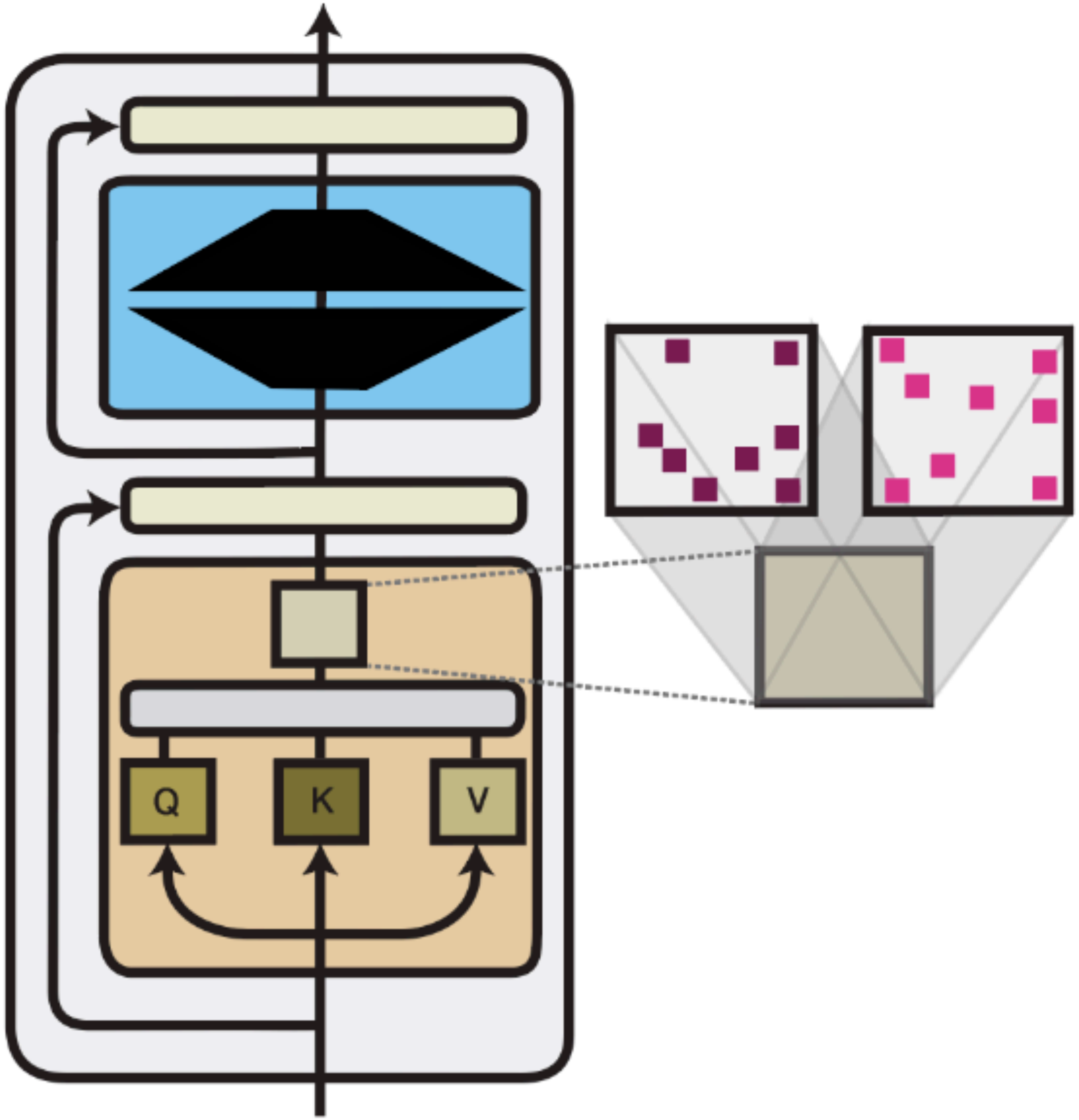


Parameter Composition

Sparse Subnetworks, where module parameters ϕ are enforced to be *sparse*

Structured Composition, where we impose a structure on the weights θ_i that we select — e.g., we update the weights belonging to a *pre-defined group*

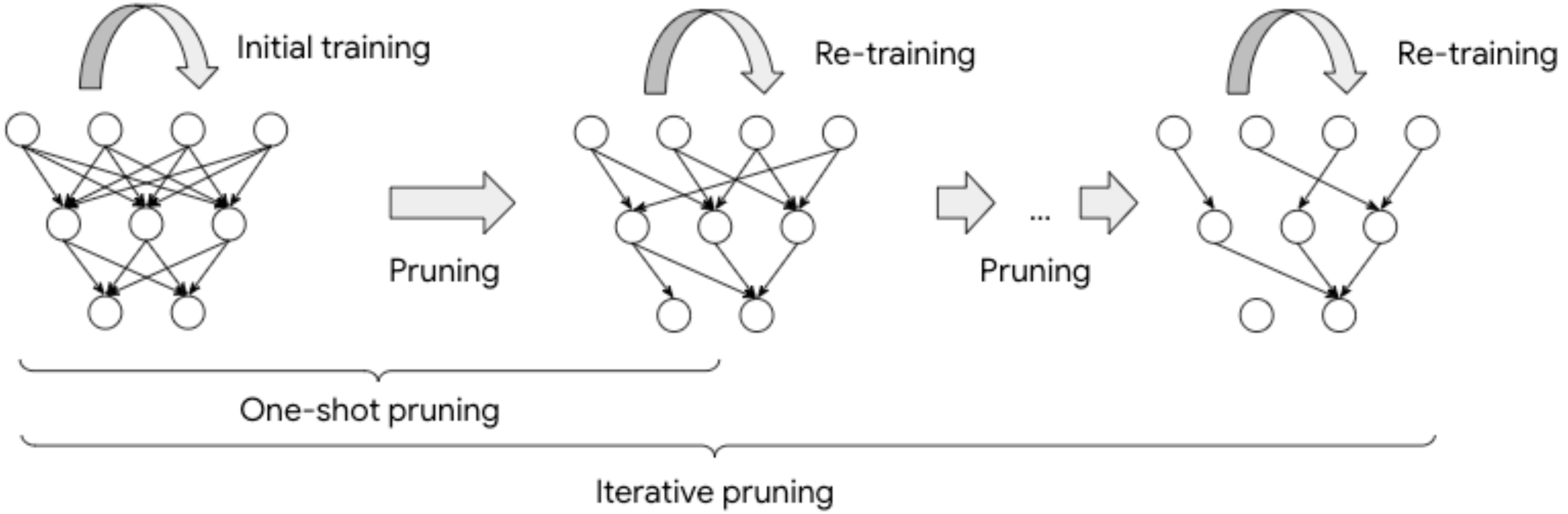
Low-Rank Composition, where the module parameters ϕ lie in a *low-dimensional space*



Parameter Composition — Sparse Subnetworks

A common inductive bias on module parameters ϕ is **sparsity**: when we do $g_i = f_{\theta_i \cdot \phi}(x)$ (element-wise product), we mask part of the neural network f

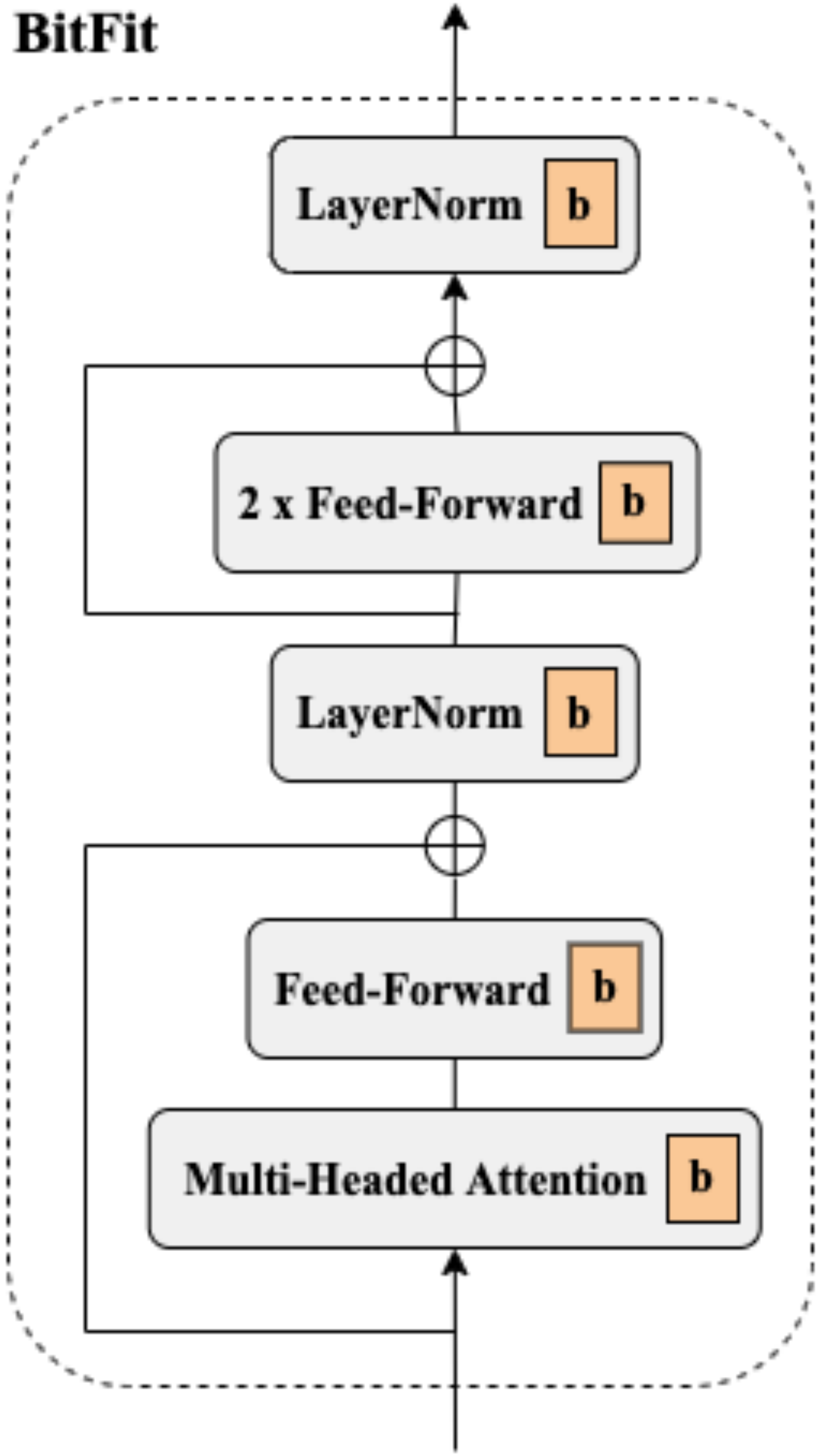
Most common sparsity method: **pruning** — e.g., see [Han et al., 2017]



Parameter Composition — Structured Composition

We can impose a structure on the weights that we select: we only modify the weights that are **associated in a pre-defined group** $\mathscr{G}$, for example, a layer, a group of layers, or more fine-grained components

Example: only update bias vectors — BitFit [Ben-Zaken et al., 2022]



Parameter Composition — Low-Rank Composition

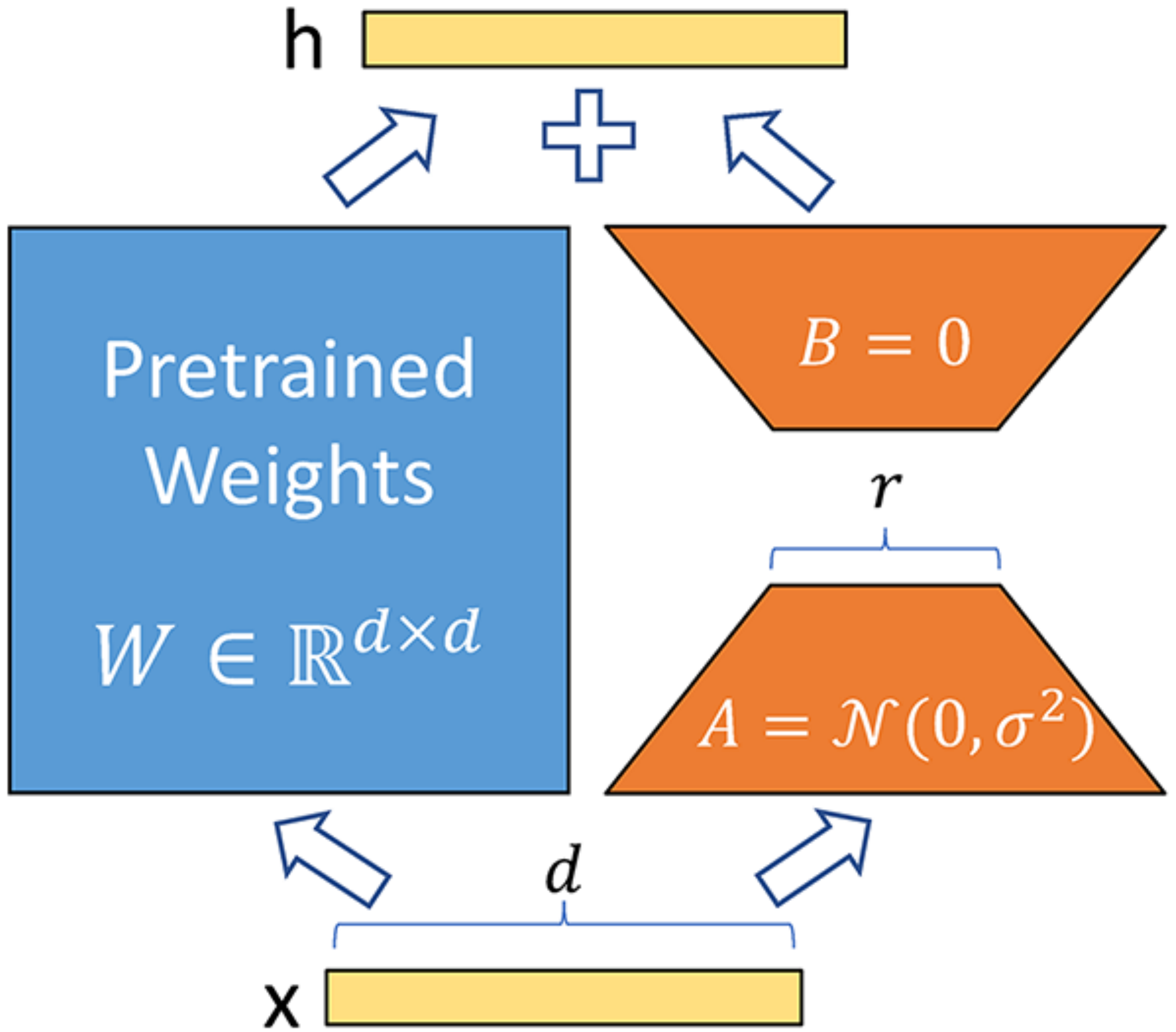
Another useful inductive bias: module parameters ϕ should **lie in a low-dimensional space.** Li et al., 2018 show that models can be optimised in a low-dimensional, randomly oriented subspace rather than the full parameter space

Low-rank fine-tuning takes the form $g = f_{\theta + P\phi}$ where $P \in \mathbb{R}^{D \times d}$ — with a dense matrix of shape $D \times d$, this scales as $\mathcal{O}(Dd)$ in time and storage

Parameter Composition — LoRA

Low-Rank Adaptation — instead of learning a low-rank factorisation via a random matrix P, we can learn the projection matrix directly

LoRA [Hu et al., 2022] learns two matrices $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$ that are applied to the self-attention weights:

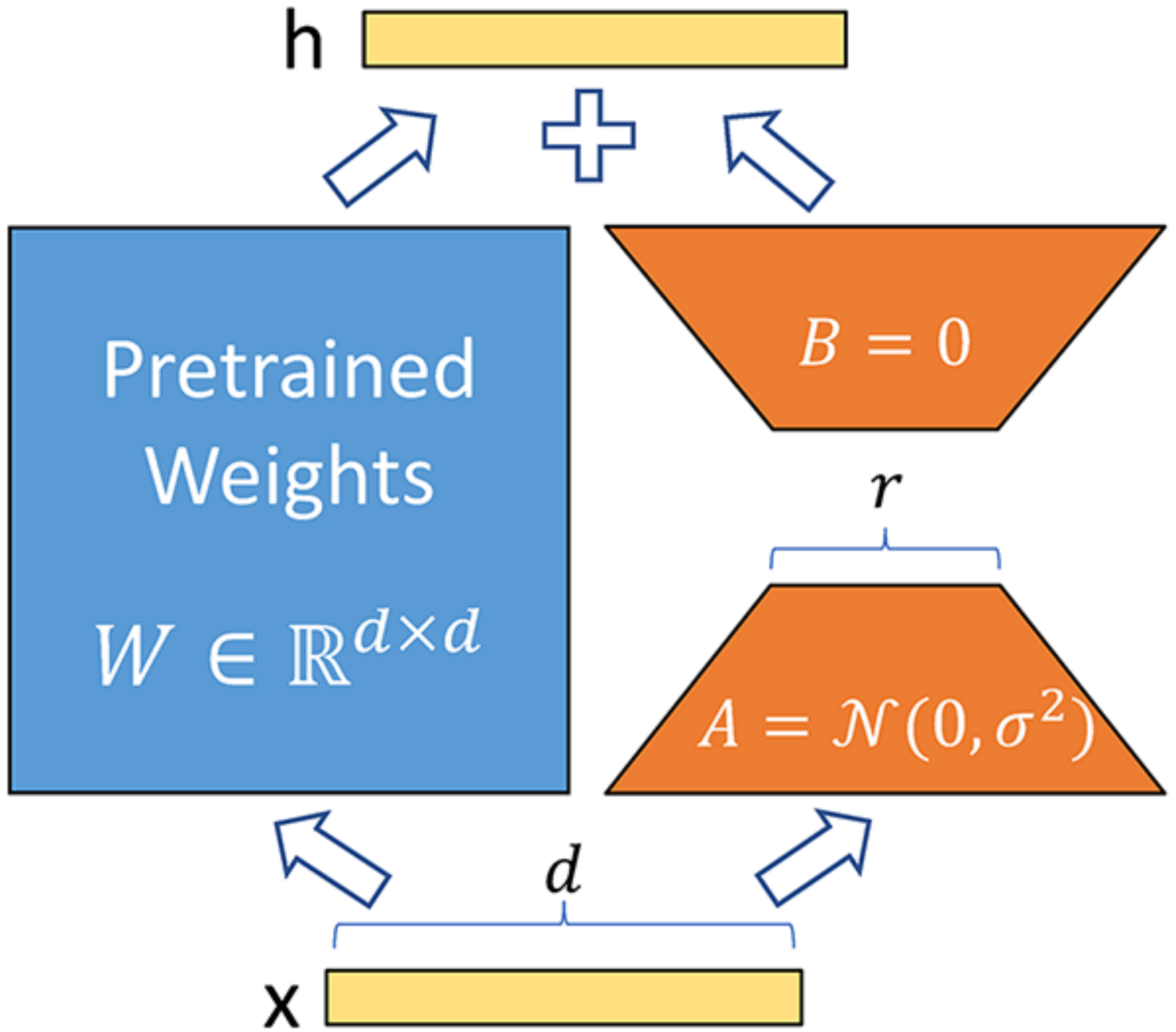


Parameter Composition — LoRA

Low-Rank Adaptation — instead of learning a low-rank factorisation via a random matrix P, we can learn the projection matrix directly

LoRA [Hu et al., 2022] learns two matrices $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$ that are applied to the self-attention weights:

$$h = \left[W_0 + \underbrace{\Delta W}_{\text{Update to parameters } W_0}\right] x = \left[W_0 + BA\right] x$$

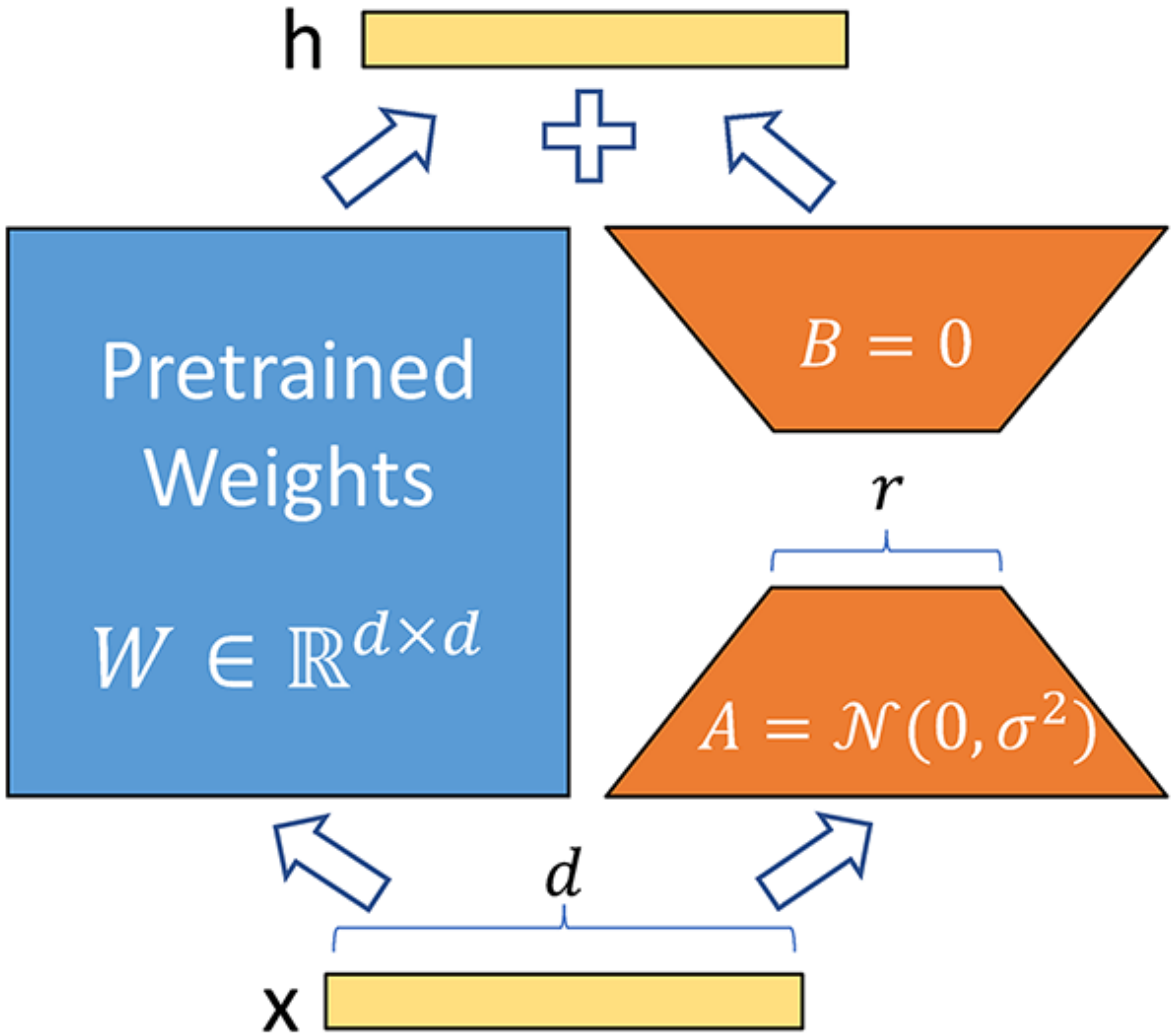


Parameter Composition — LoRA

Low-Rank Adaptation — instead of learning a low-rank factorisation via a random matrix P, we can learn the projection matrix directly

LoRA [Hu et al., 2022] learns two matrices $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$ that are applied to the self-attention weights:

$$h = \left[W_0 + \underbrace{\Delta W}_{\text{Update to parameters } W_0}\right] x = \left[W_0 + BA\right] x$$



In our notation:

$$g_i = f_{\theta_i + B_i A_i}, \forall f_i \in \mathcal{G}$$

Parameter Composition — LoRA

Applying LoRA to a Transformer layer — remember how Transformers work:

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

$$\text{with } \text{Attention}(Q\boxed{W_{Q,i}}, K\boxed{W_{K,i}}, V\boxed{W_{V,i}}) \text{ and } \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)$$

Parameter Composition — LoRA

Applying LoRA to a Transformer layer — remember how Transformers work:

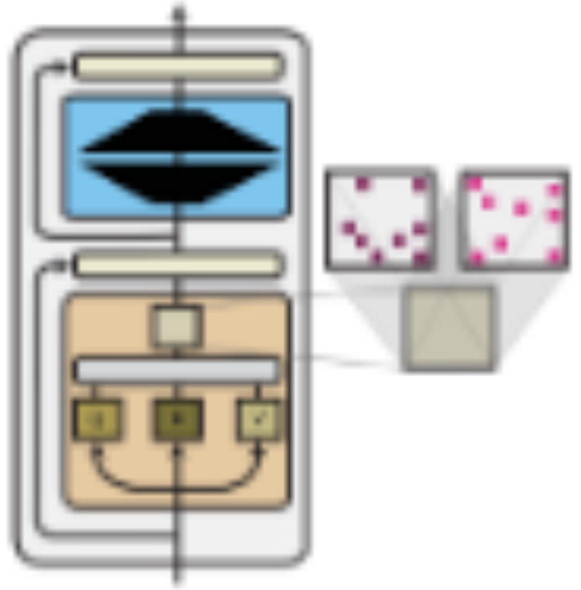
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_n)W_O$$

$$\text{with } \text{Attention}(Q\boxed{W_{Q,i}}, K\boxed{W_{K,i}}, V\boxed{W_{V,i}}) \text{ and } \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)$$

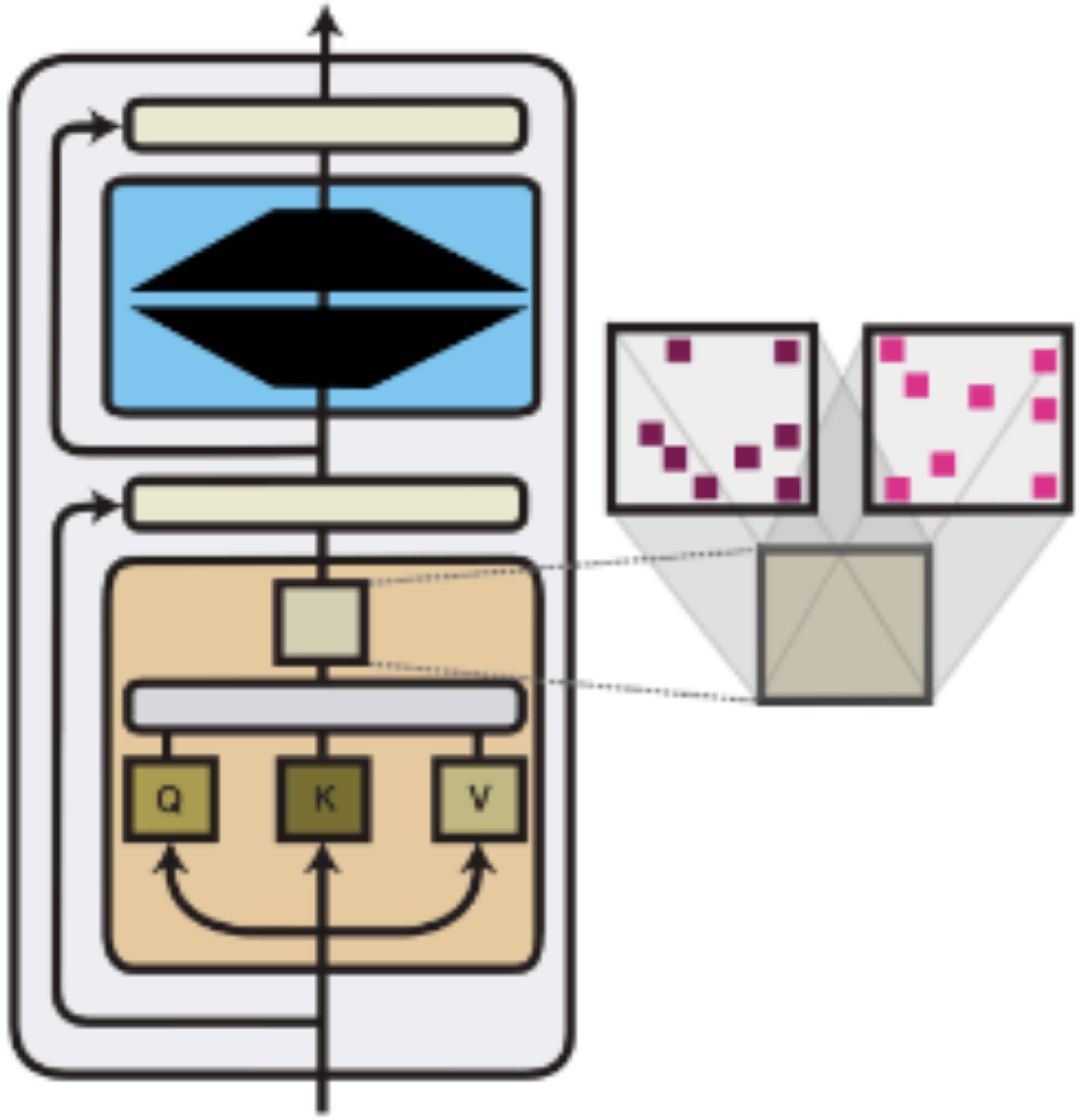
We can use LoRA to adapt the weights $W_{Q,i}$, $W_{K,i}$, and/or $W_{V,i}$ — in the case of $W_{Q,i}$, the updated weights will be:

$$\begin{aligned}\widetilde{W_{Q,i}} &= W_{Q,i} + \Delta W_{Q,i} \\ &= W_{Q,i} + B_{Q,i}A_{Q,i} \text{ with } B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}\end{aligned}$$

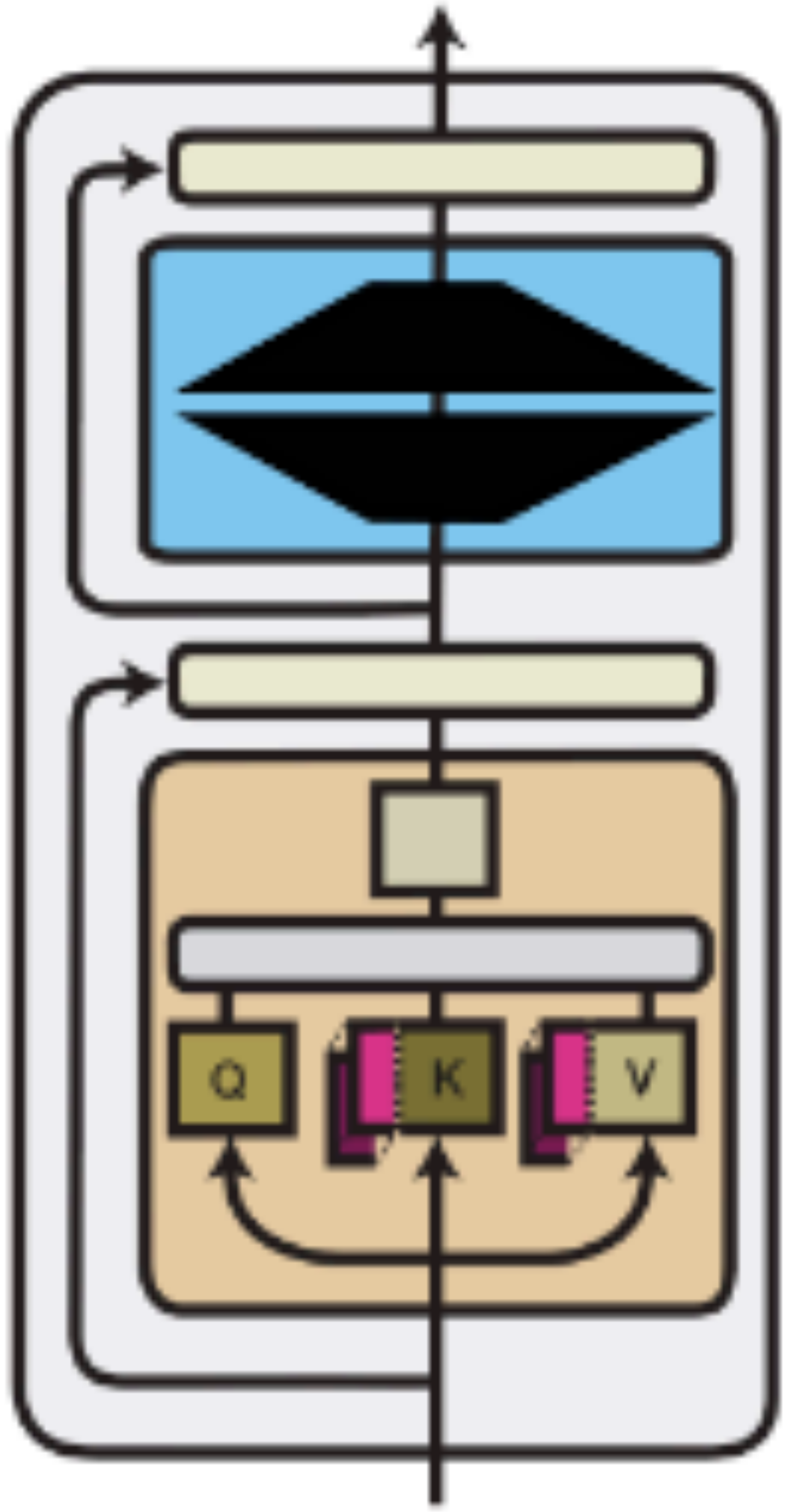
Computation Functions — Comparison

	Parameter efficiency	Training efficiency	Inference efficiency	Performance
Parameter composition 	Methods such as LoRA require < 3% of parameters	Pruning requires re-training iterations	Does not increase the model size	E.g., LoRA achieves strong performance
	+	-	++	+

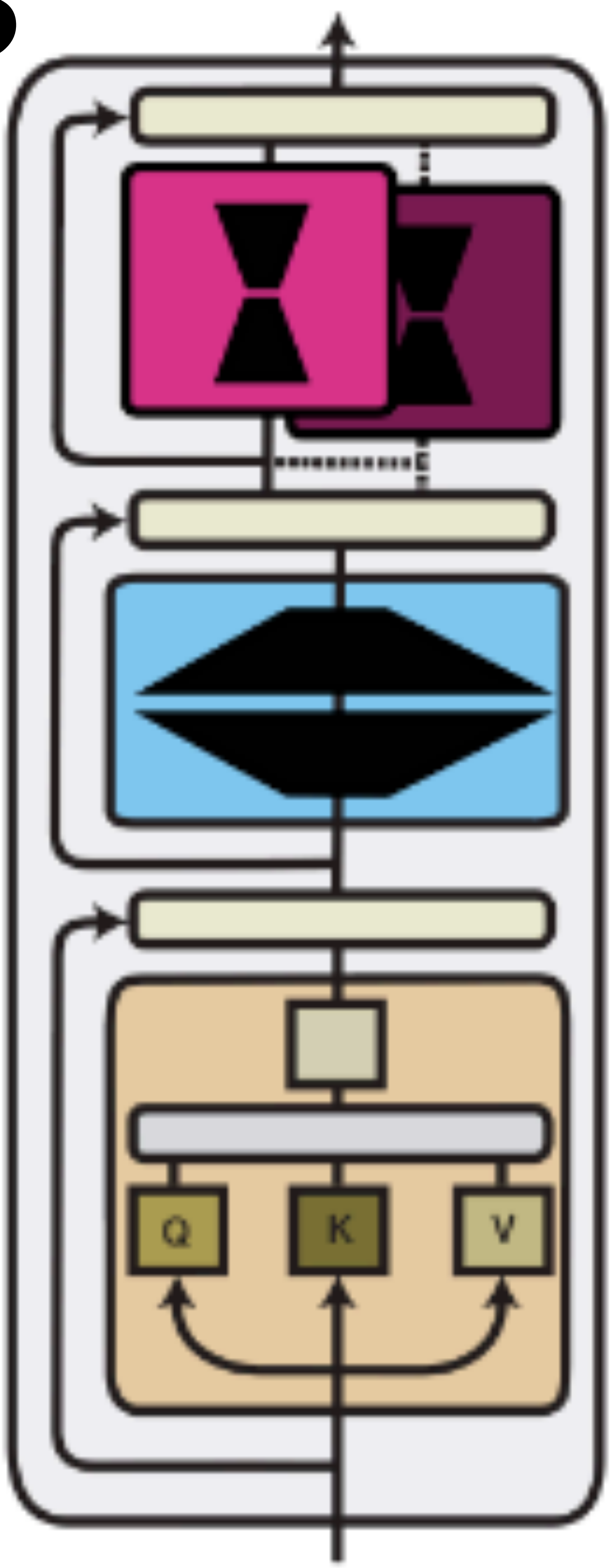
Composition Functions



Parameter Composition



Input Composition

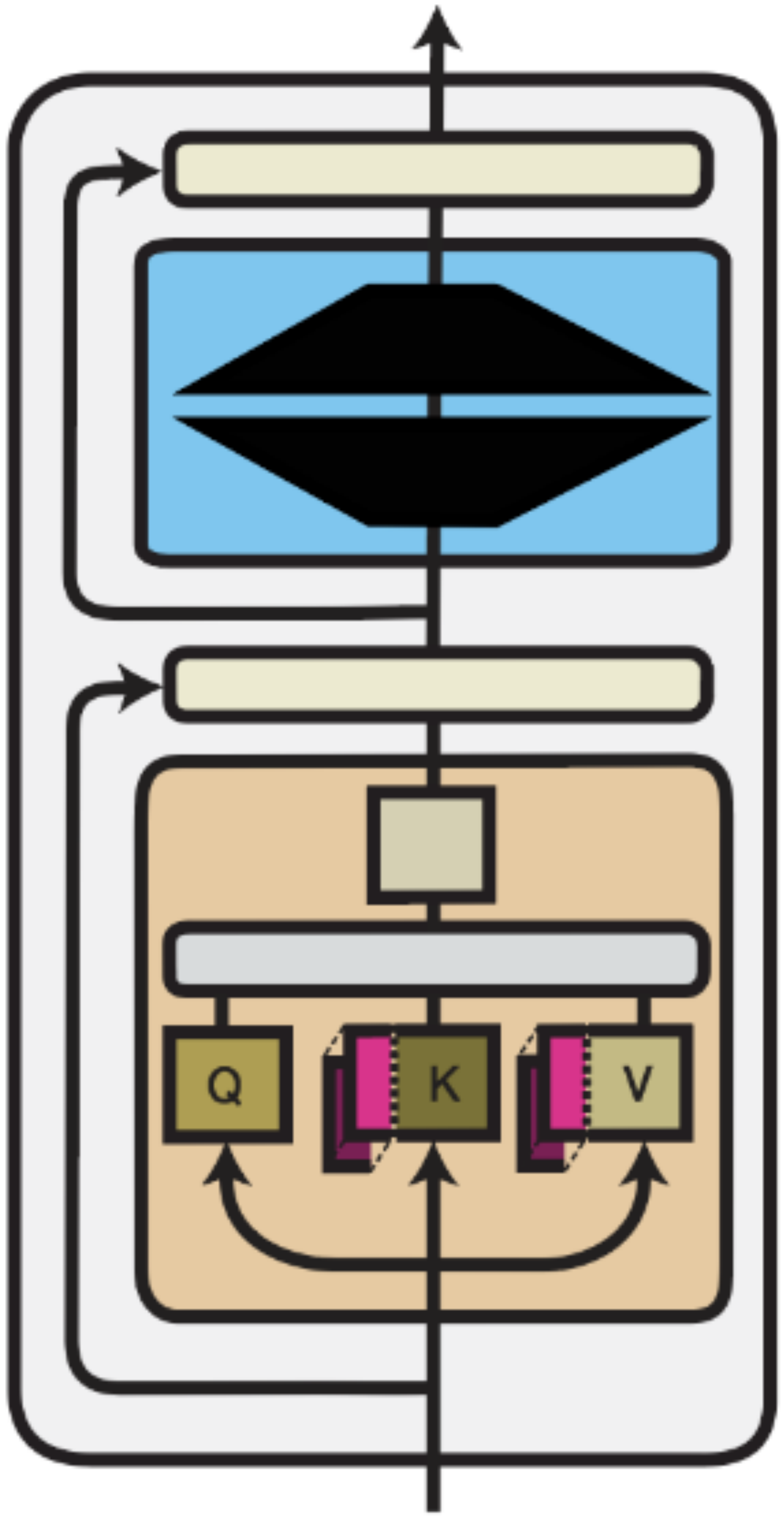


Function Composition

Input Composition

Idea — augment the input of the model with a learnable vector ϕ:

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

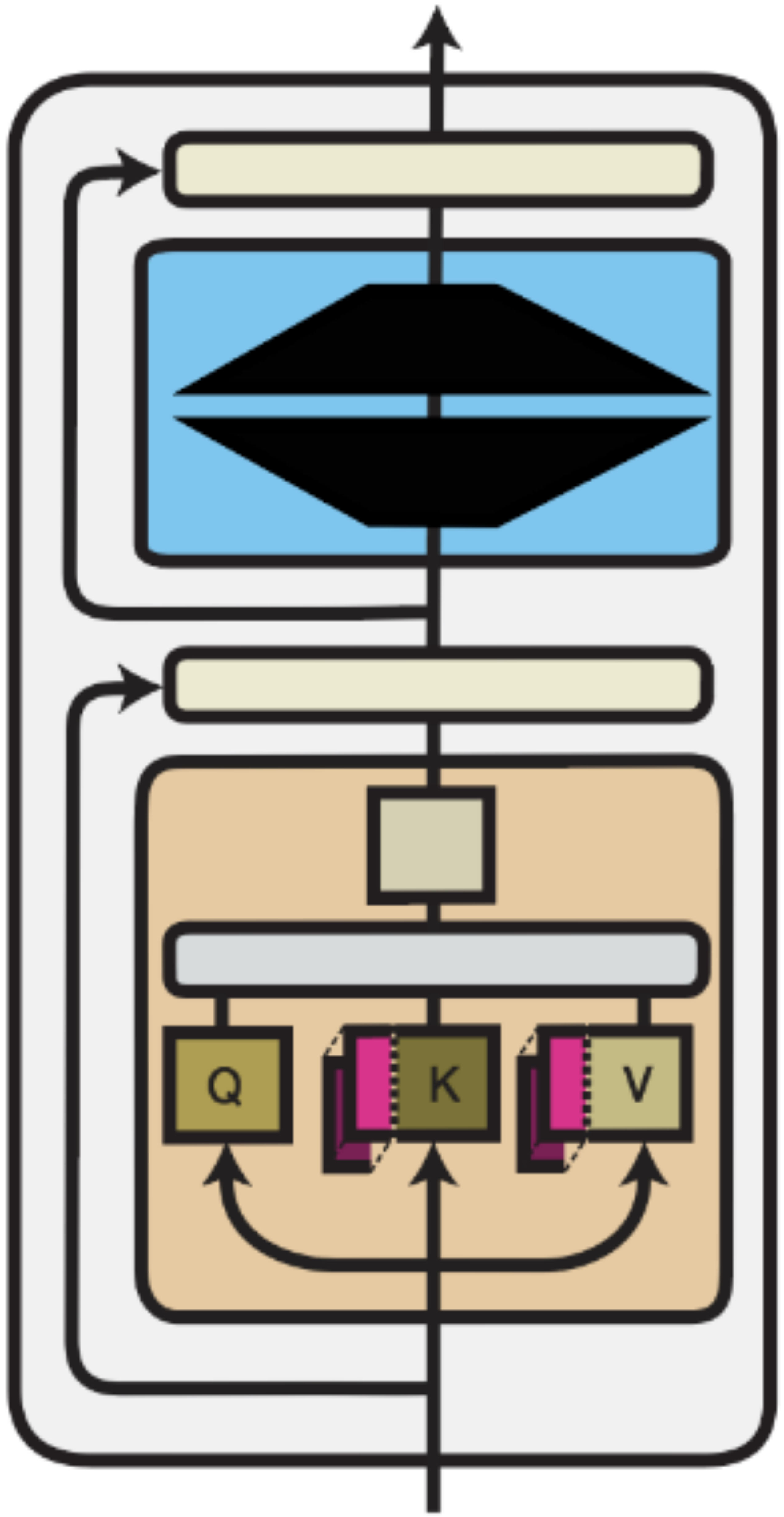


Input Composition

Idea — augment the input of the model with a learnable vector ϕ:

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

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



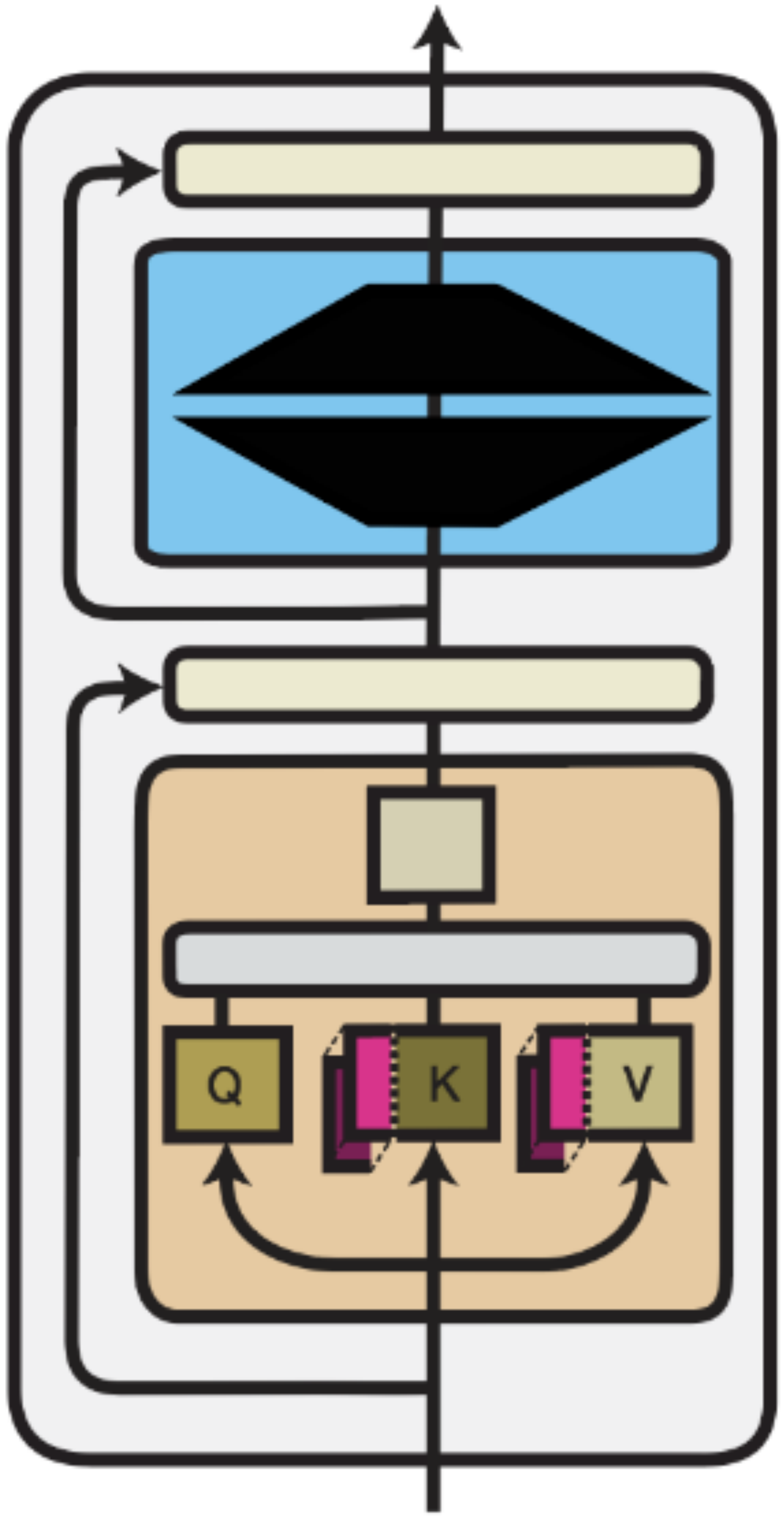
Input Composition

Idea — augment the input of the model with a learnable vector ϕ:

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

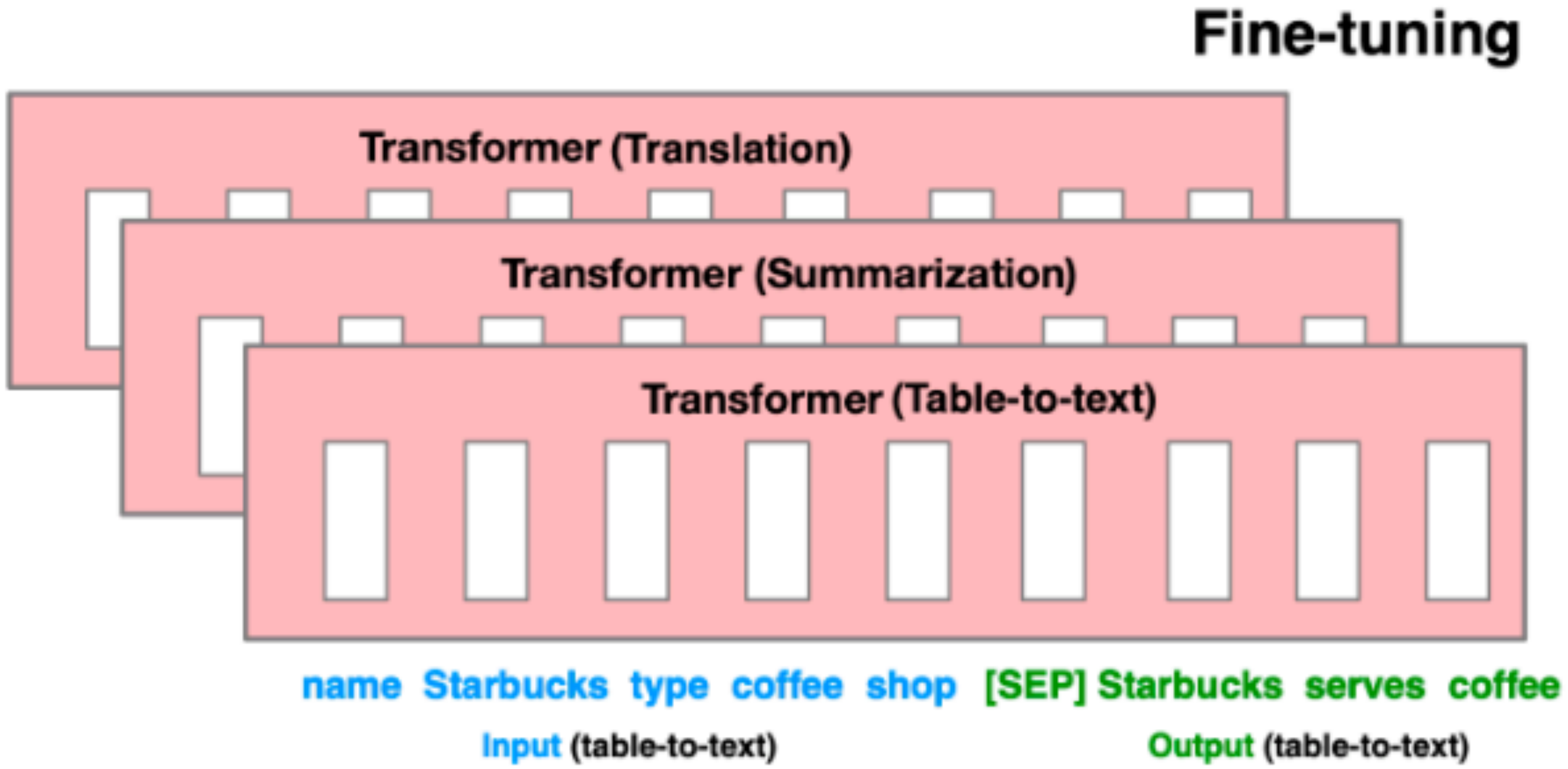
Input Composition and Prompting — standard prompting can be seen as finding a discrete text prompt that, when embedded using the model's embedding layer, yields ϕ_i

However, models tend to be sensitive to the choice of the prompt [Webson and Pavlick, 2022] and the order of examples [Zhao et al., 2021; Lu et al., 2022]



Prompt Tuning

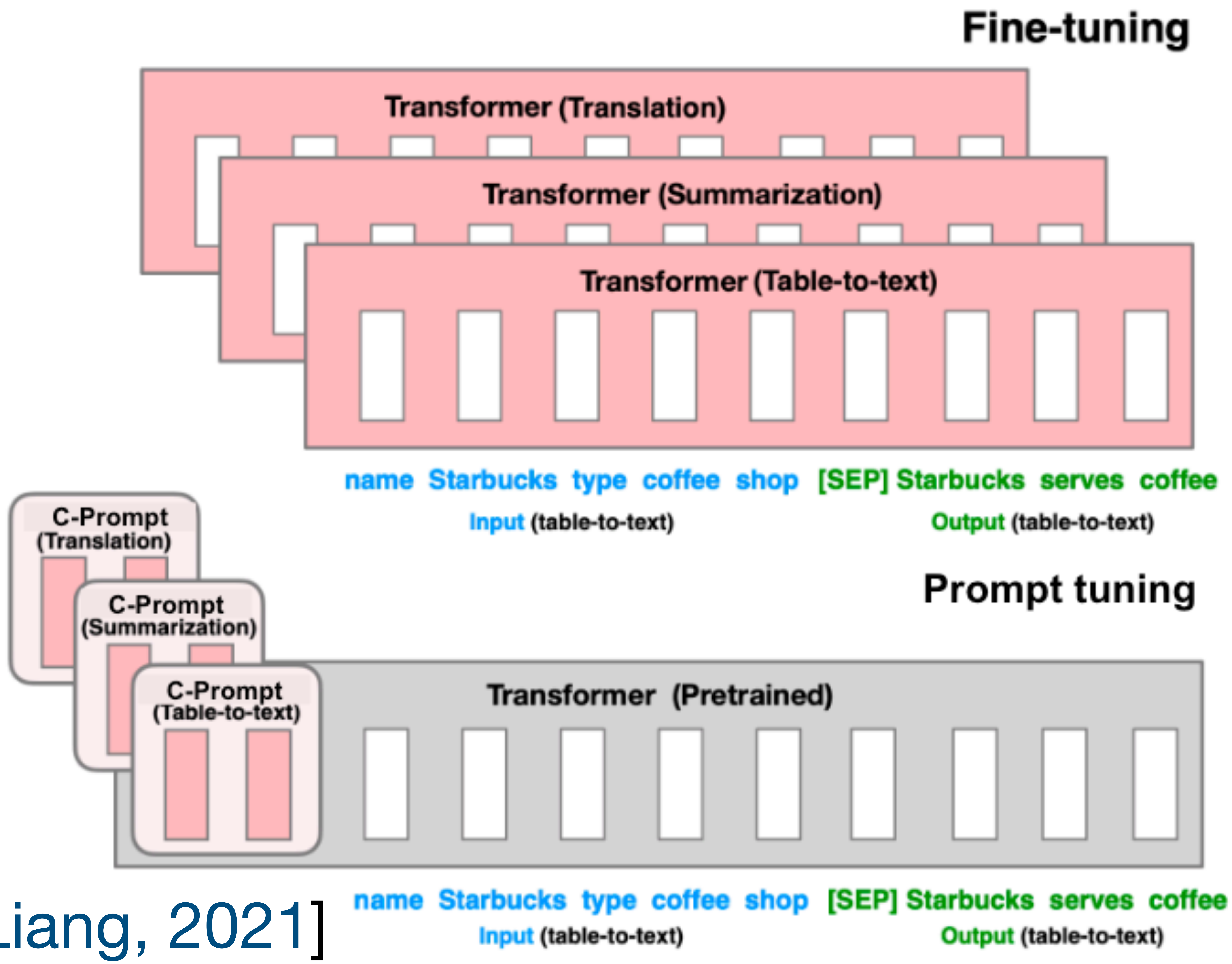
Idea — we can directly learn a *continuous prompt* ϕ which is pre-pended to the input [Liu et al., 2021; Hambardzumyan et al., 2021; Lester et al., 2021]



Prompt Tuning

Idea — we can directly learn a *continuous prompt* ϕ which is pre-pended to the input [Liu et al., 2021; Hambardzumyan et al., 2021; Lester et al., 2021]

Here the module parameters ϕ is typically a matrix consisting of a sequence of continuous prompt embeddings

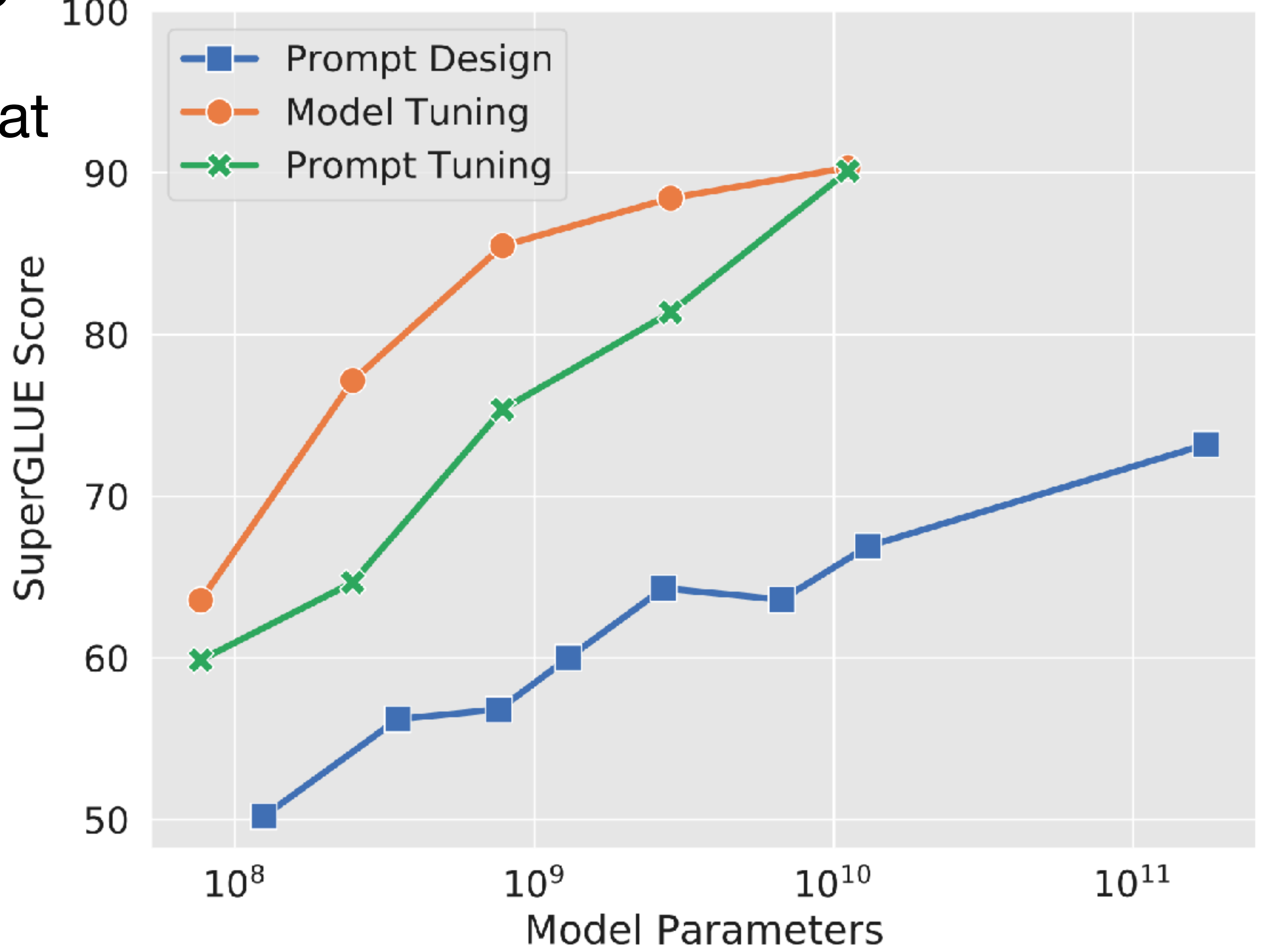


[Li and Liang, 2021]

Prompt Tuning Works Well at Scale

Only using trainable parameters at the input layer limits its 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 vs. Standard fine-tuning and prompt design across T5 models of different sizes [Lester et al., 2021]

Multi-Layer Prompt Tuning

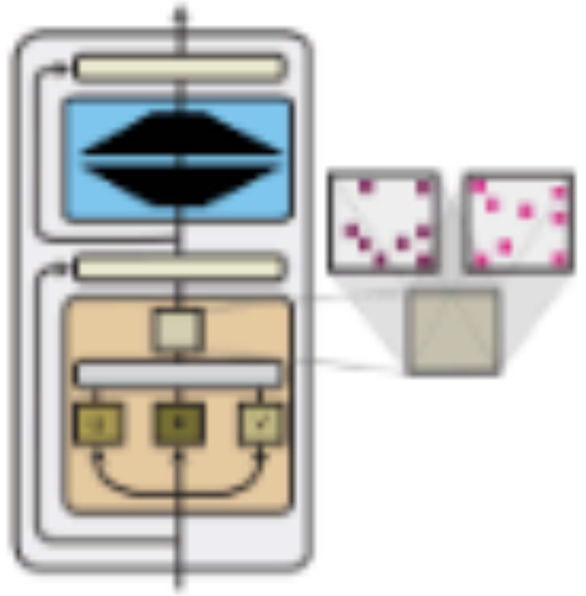
Instead of learning the module parameters ϕ_i only at the input layer, we can learn them at *every layer of the model* [Li and Jiang, 2021; Liu et al., 2022]

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

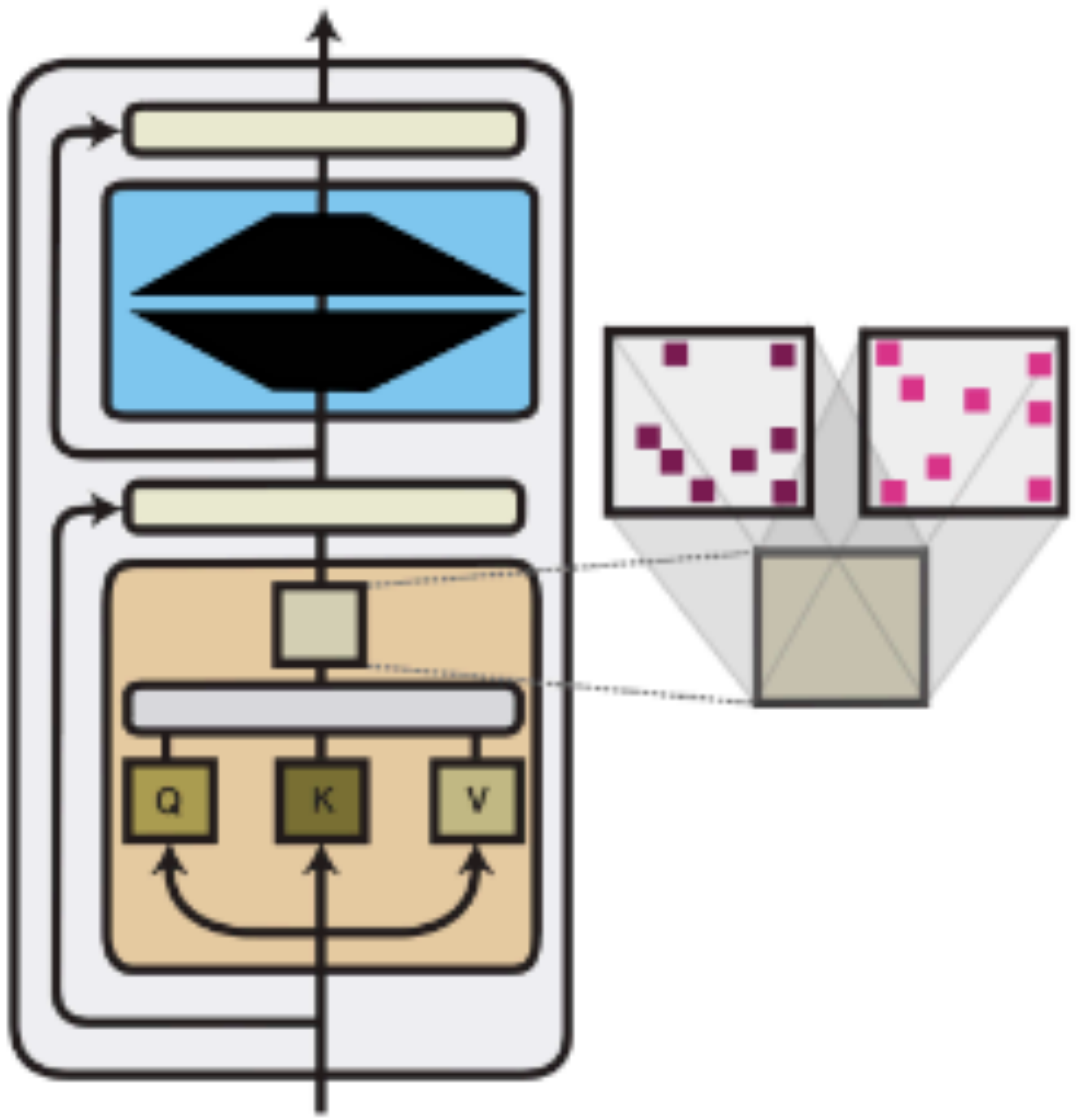
Prompt tuning

Multi-layer prompt tuning

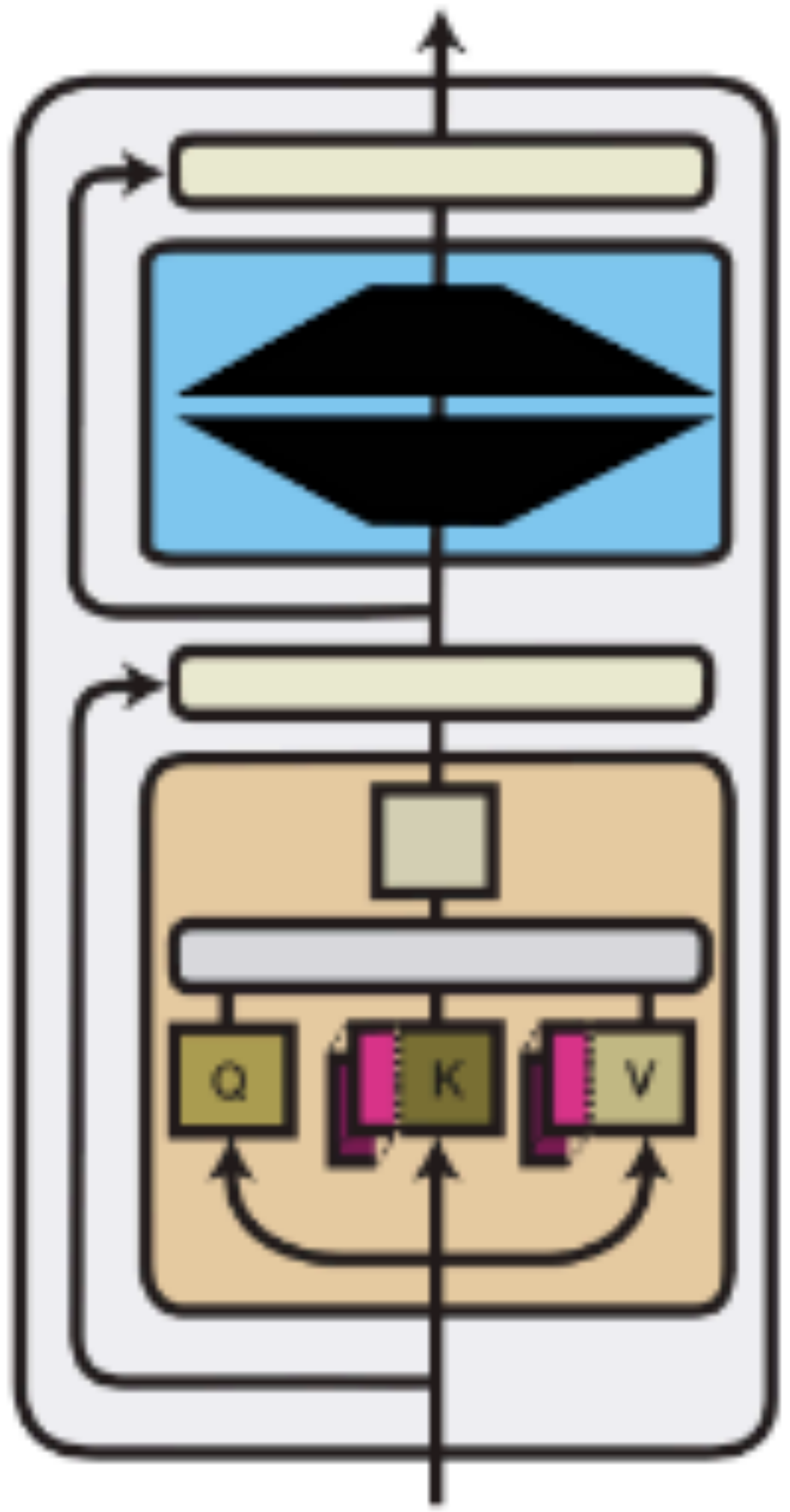
Computation Functions — Comparison

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

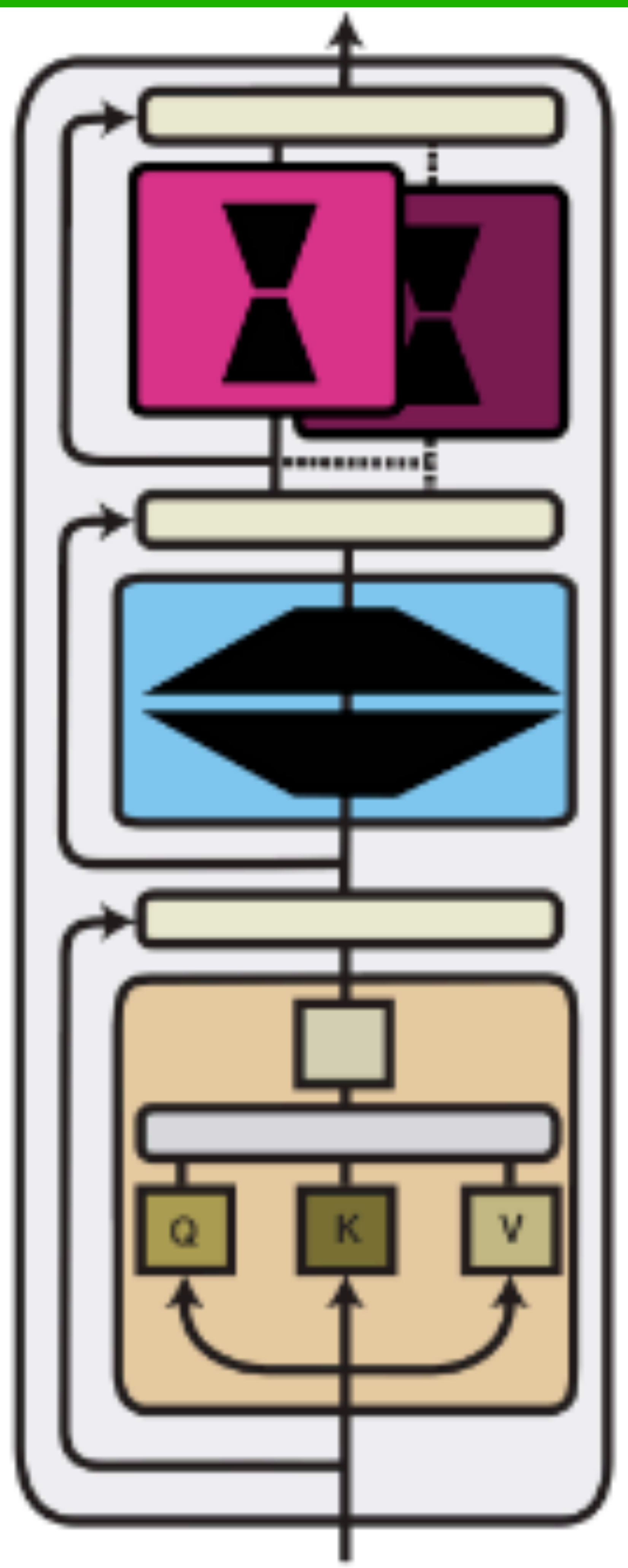
Composition Functions



Parameter Composition



Input Composition



Function Composition

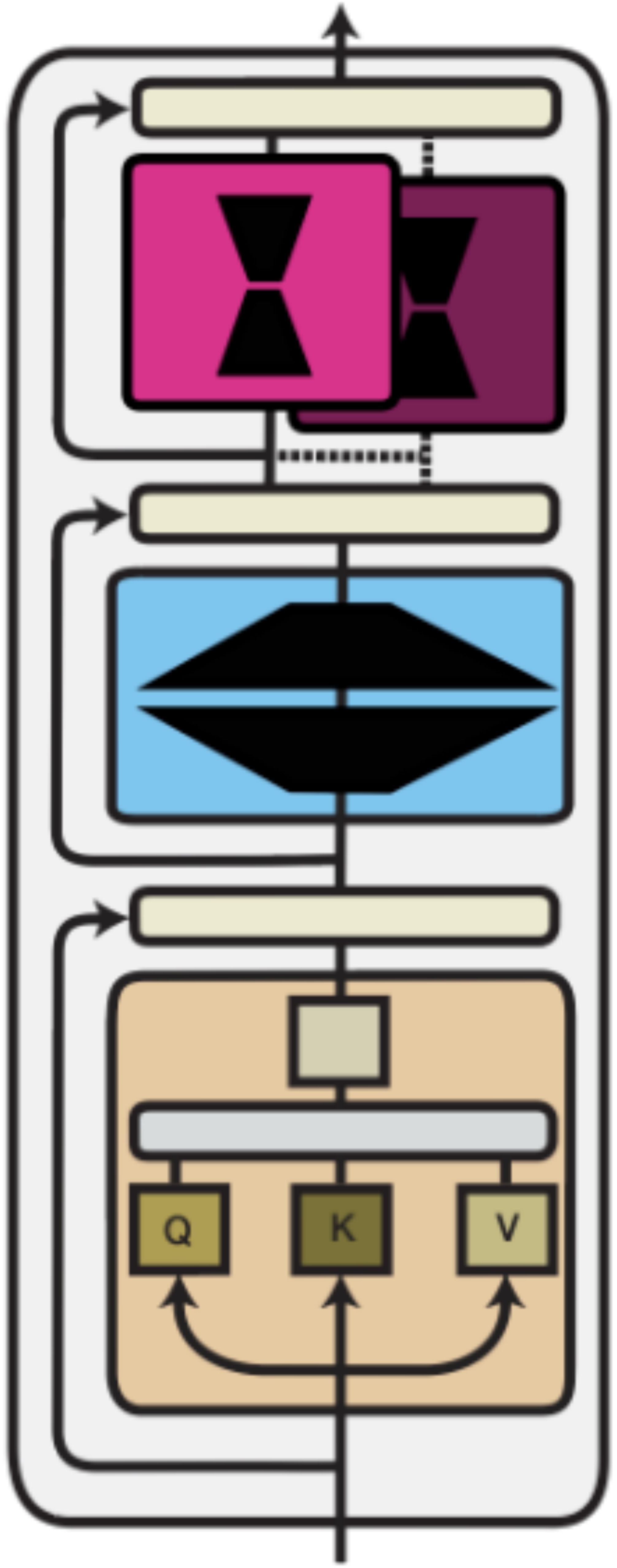
Function Composition

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

$$g_i(x) = f_{\theta_i} \odot f_\phi(x)$$

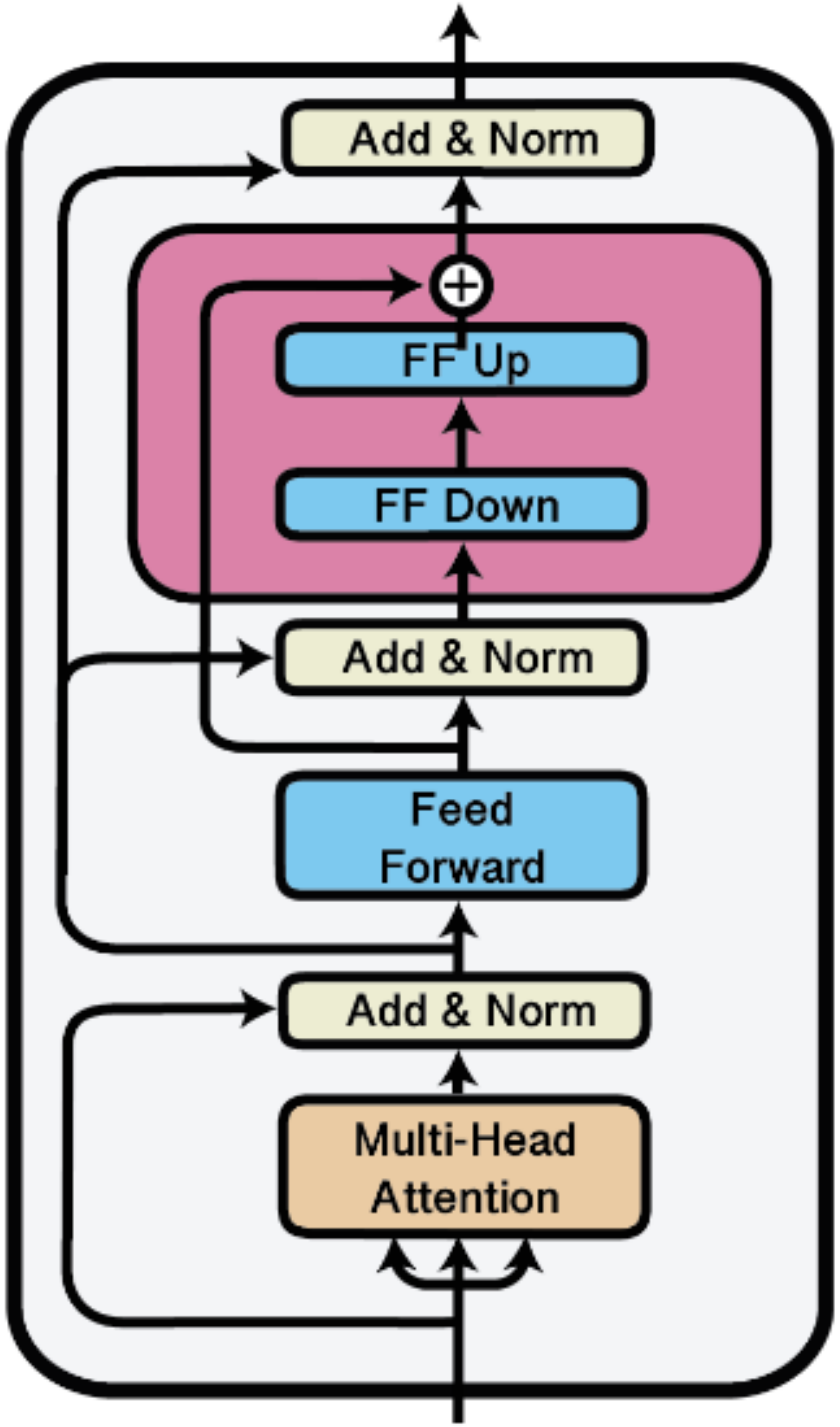
Commonly used in multi-task learning, where we have multiple task-specific models composed together — e.g., see the surveys in [Ruder, 2017; Crawshaw, 2020]

However, here we focus on functions that can be added to pre-trained models like LLMs 🙂



Adapters

The main purpose of functions f_{ϕ_i} added to a pre-trained model is to adapt it to a new task — these functions are also known as **adapters**

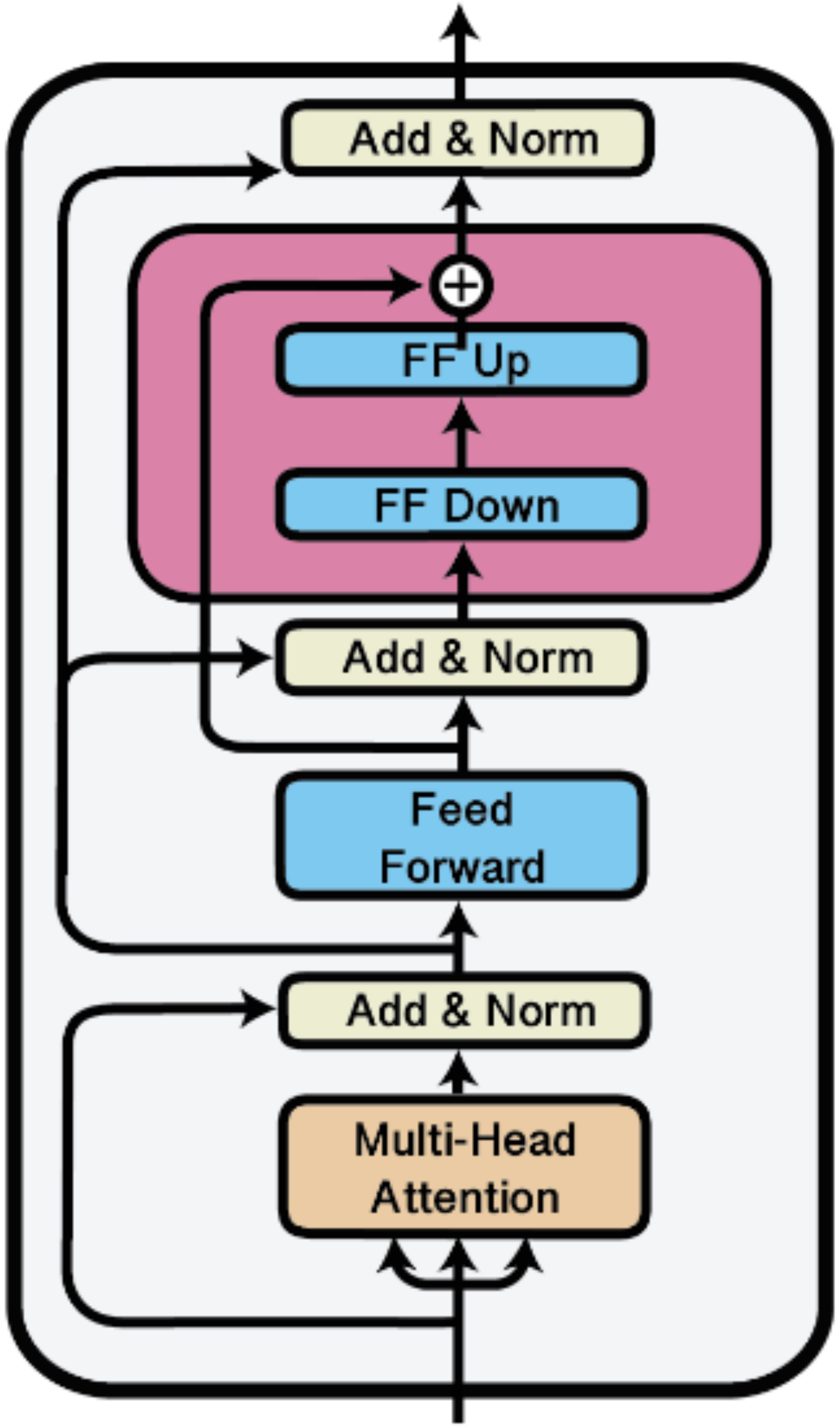


Adapters

The main purpose of functions f_{ϕ_i} added to a pre-trained model is to adapt it to a new task — these functions are also known as **adapters**

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]

$$f_{\phi_i}(x) = W_D \left[\sigma \left(W_U x \right) \right]$$



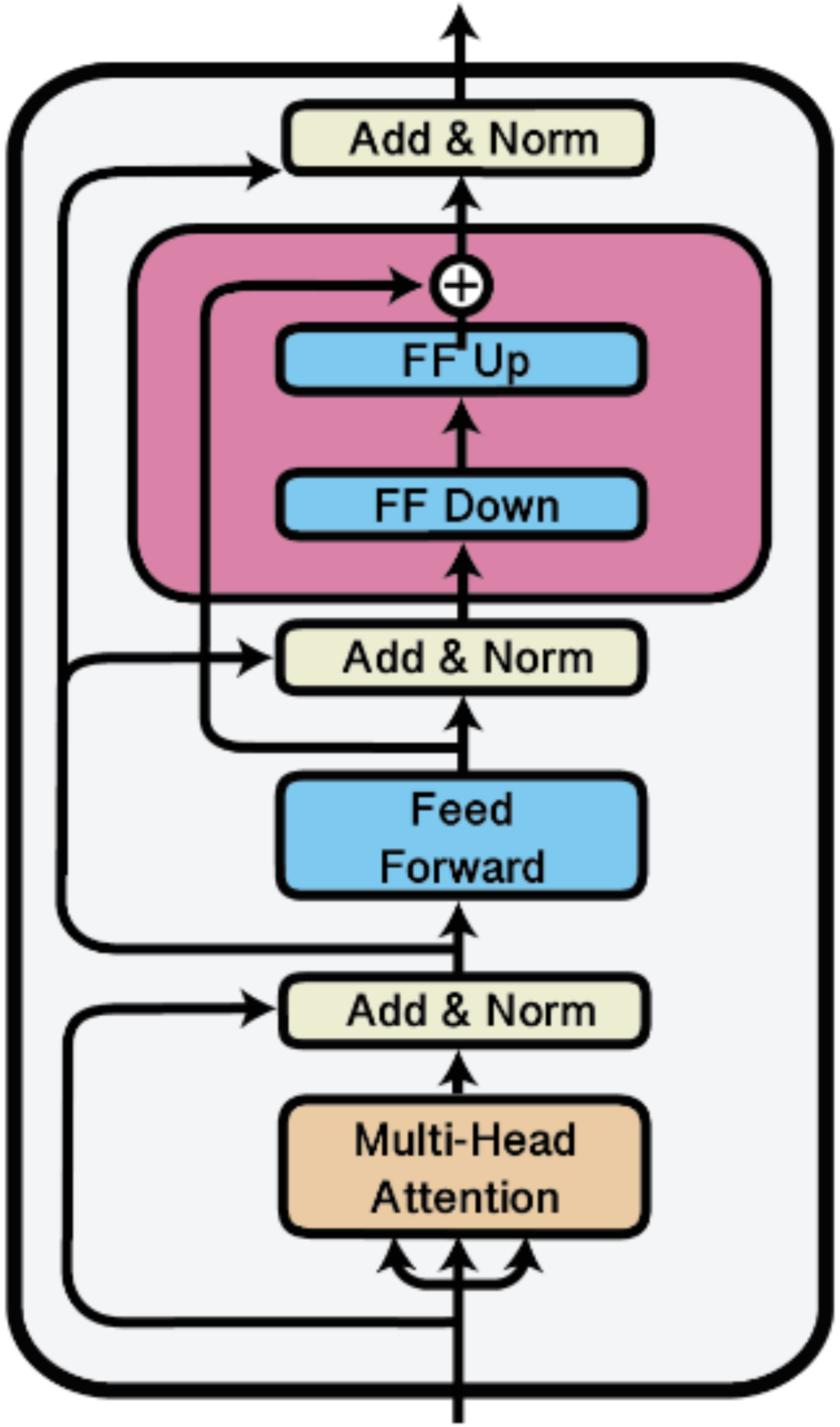
Adapters

The main purpose of functions f_{ϕ_i} added to a pre-trained model is to adapt it to a new task — these functions are also known as **adapters**

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]

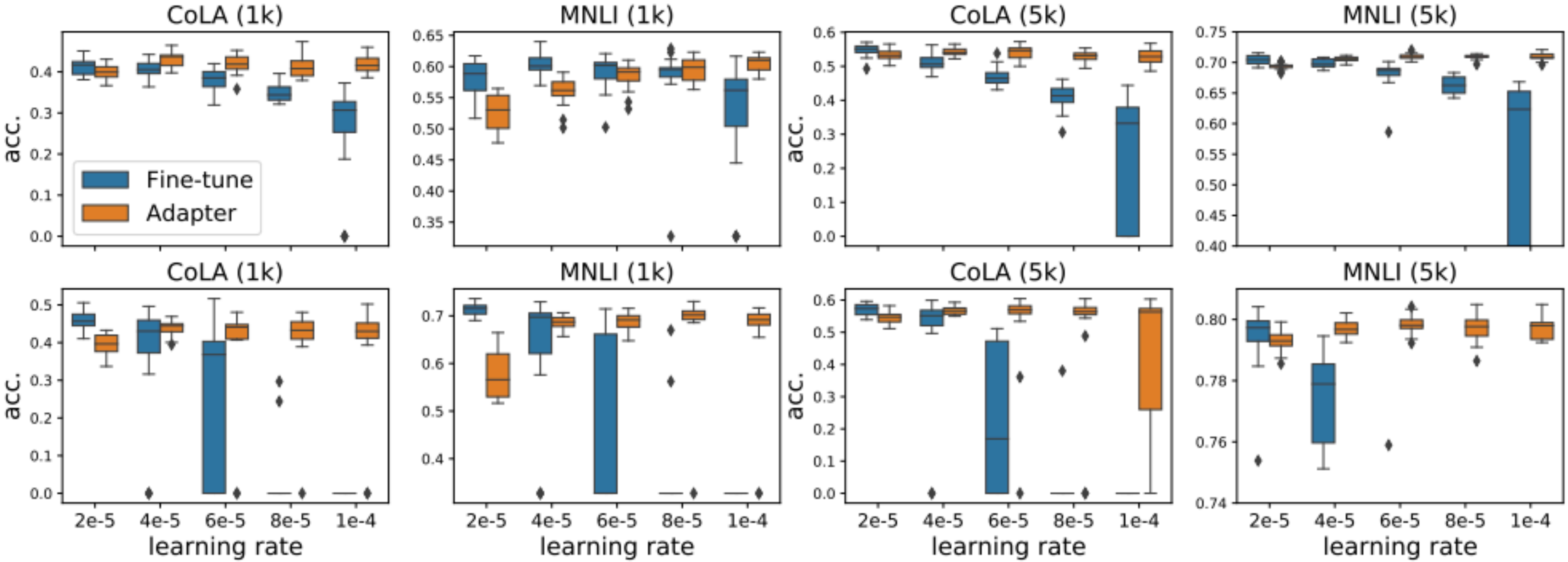
$$f_{\phi_i}(x) = W_D \left[\sigma \left(W_U x \right) \right]$$

Adapter usually placed after multi-head attention and/or after the feed-forward layer



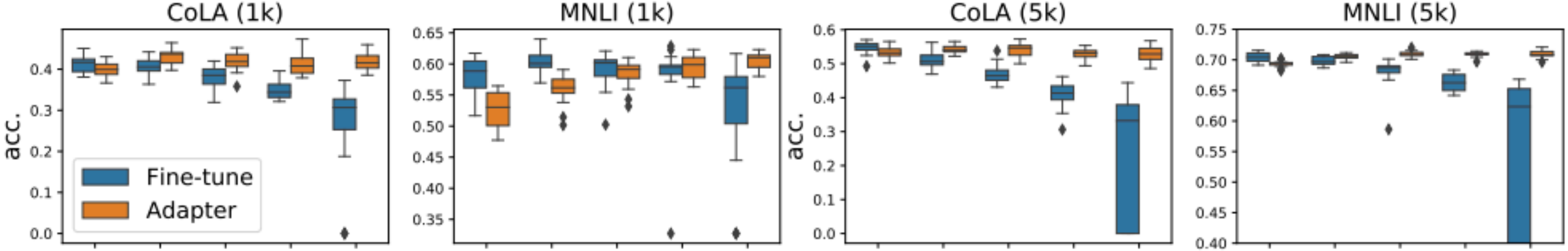
Benefits of Adapters

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



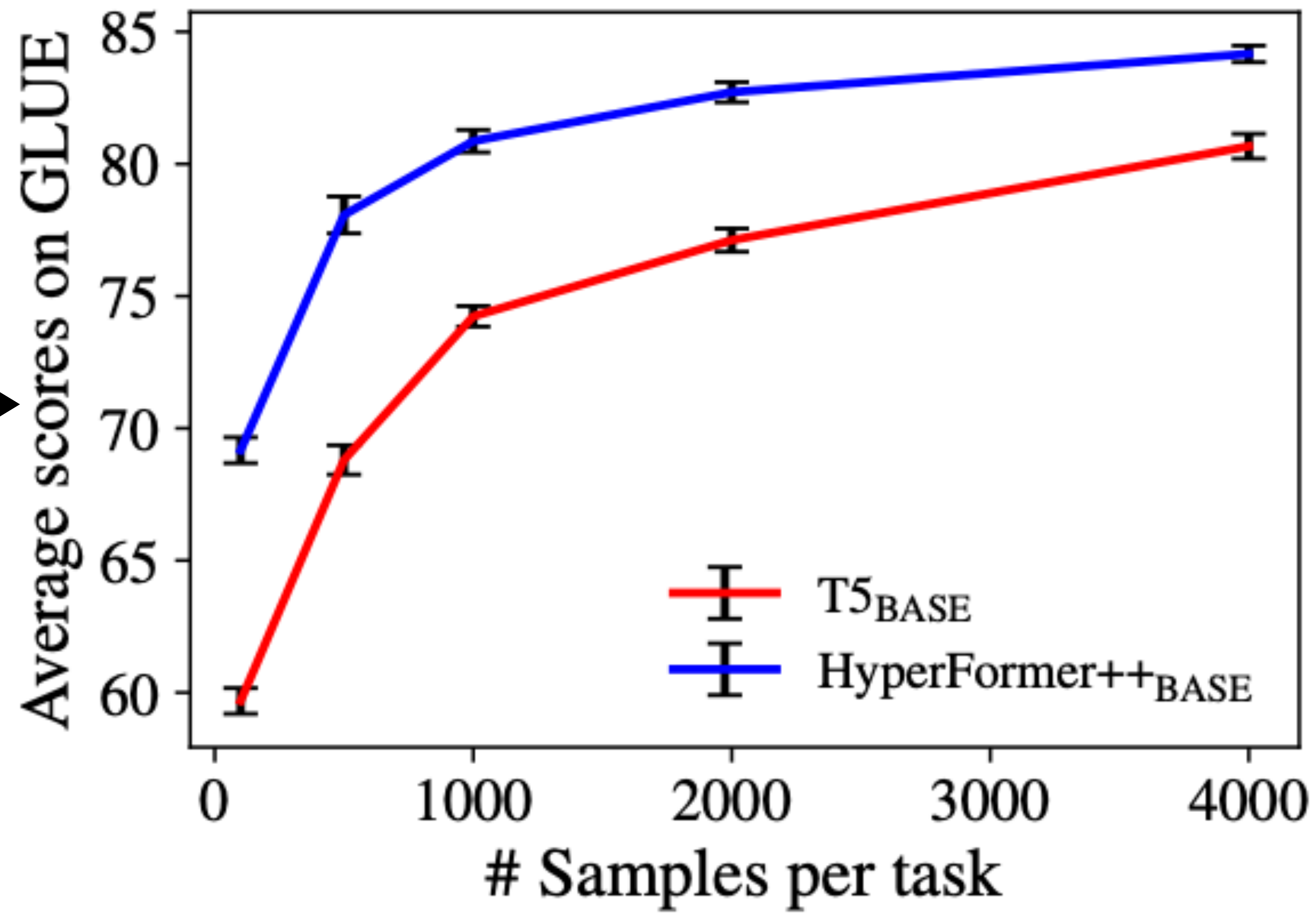
Benefits of Adapters

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

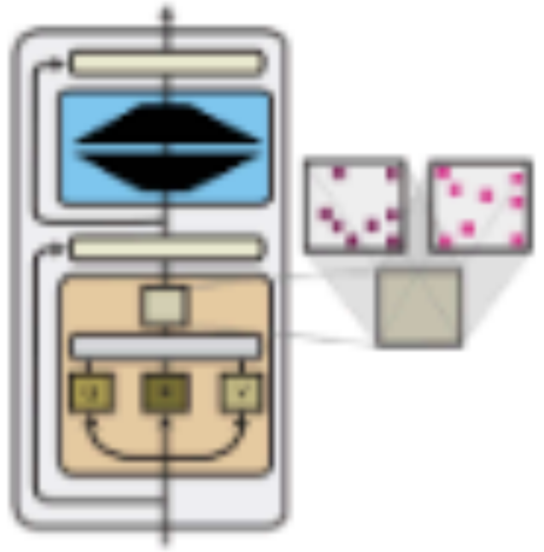


Increased sample efficiency [Mahabadi et al., 2021]

Results on GLUE with different numbers of training examples per task



Computation Functions — Comparison

	Parameter efficiency	Training efficiency	Inference efficiency	Performance
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
	−	+	−	++