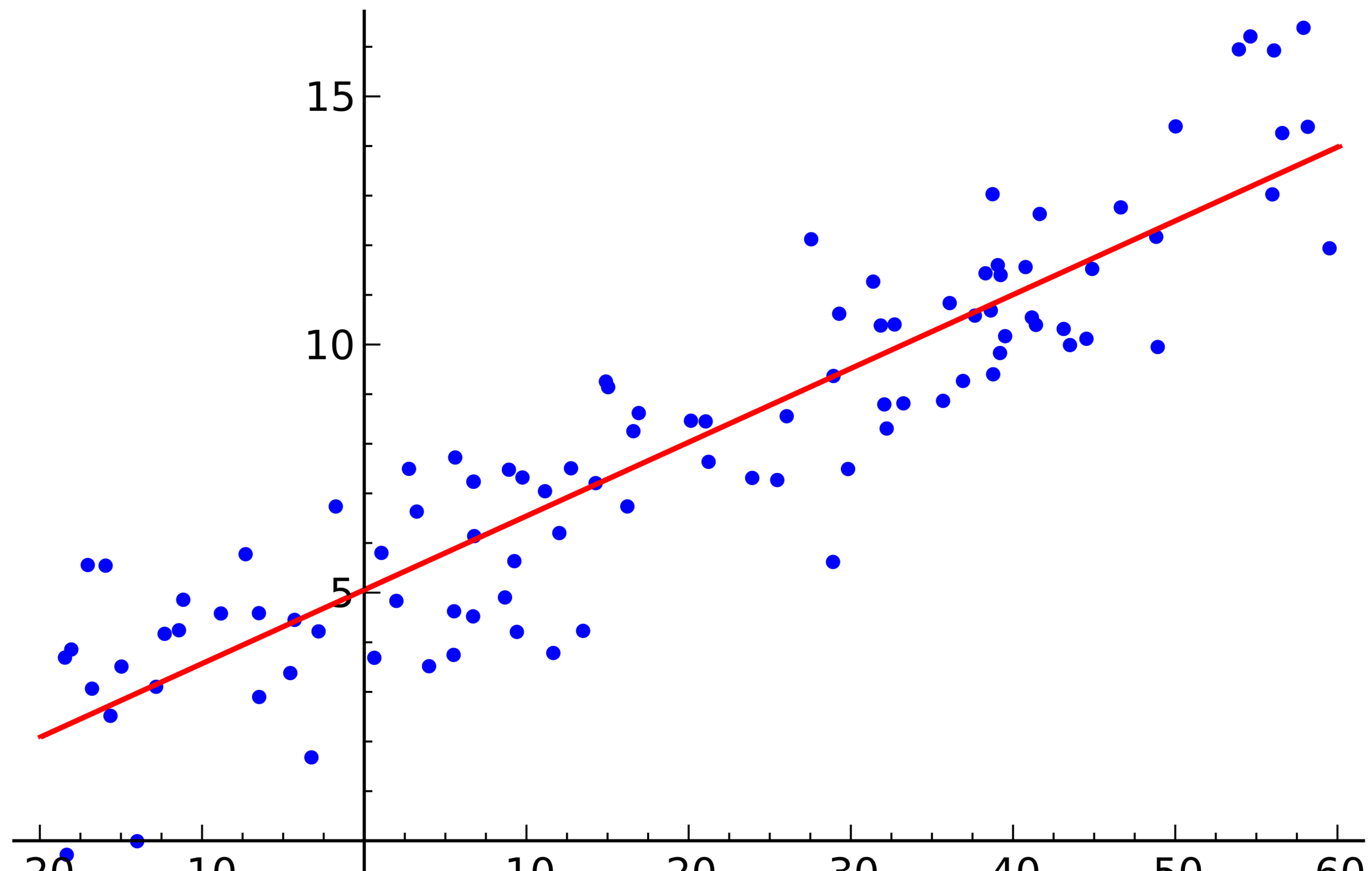
EECE695D: Efficient ML Systems

Semester in Review

1. Linear Models



We first started with **"how to count memory / compute"** of both machine learning inference and training.

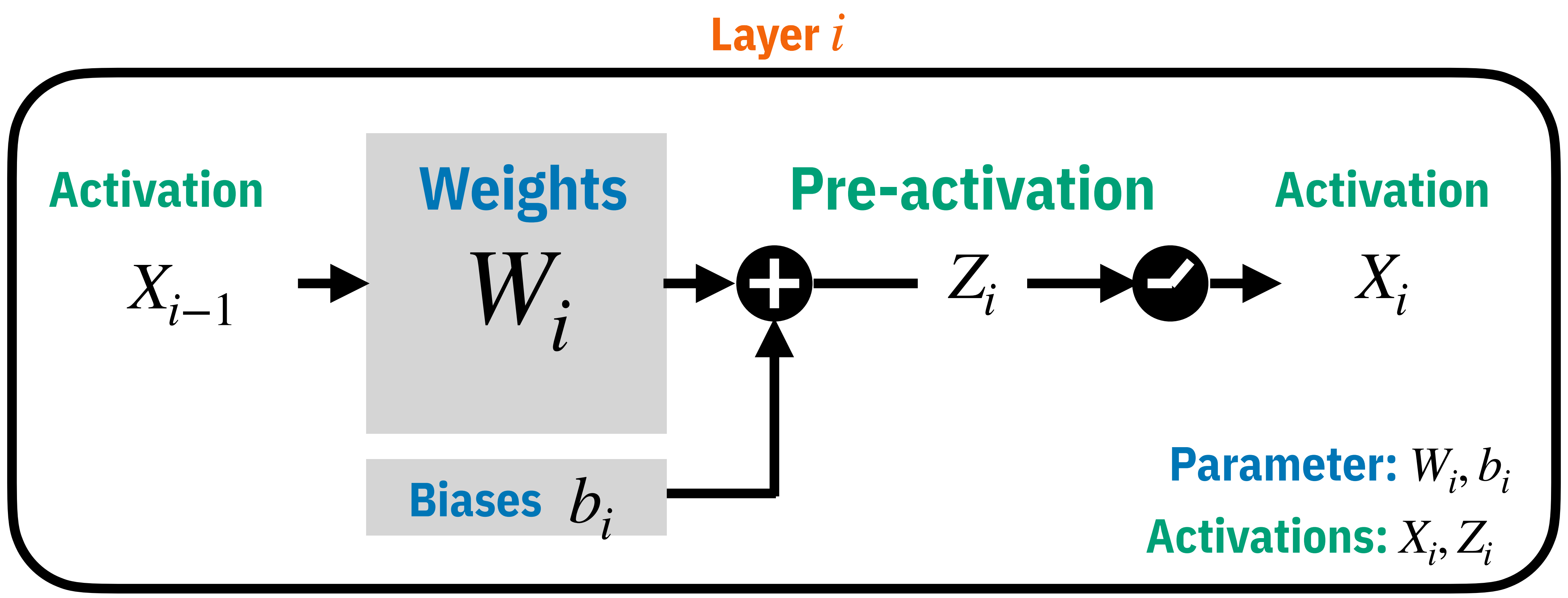
- **Compute.** FLOPs and MACs...
  (rounding issues of MAC, tradeoff with parallel)
- **Memory.** So-called von Neumann bottleneck
  (very basic ideas of how CPU works, idle time)
- **Training cost.** Depends on your optimization algorithm!
  (Explicit solution vs. approximate solution)
- **Matmuls.** Modern ML is all about matrix multiplications!
  (some parts can be precomputed!)

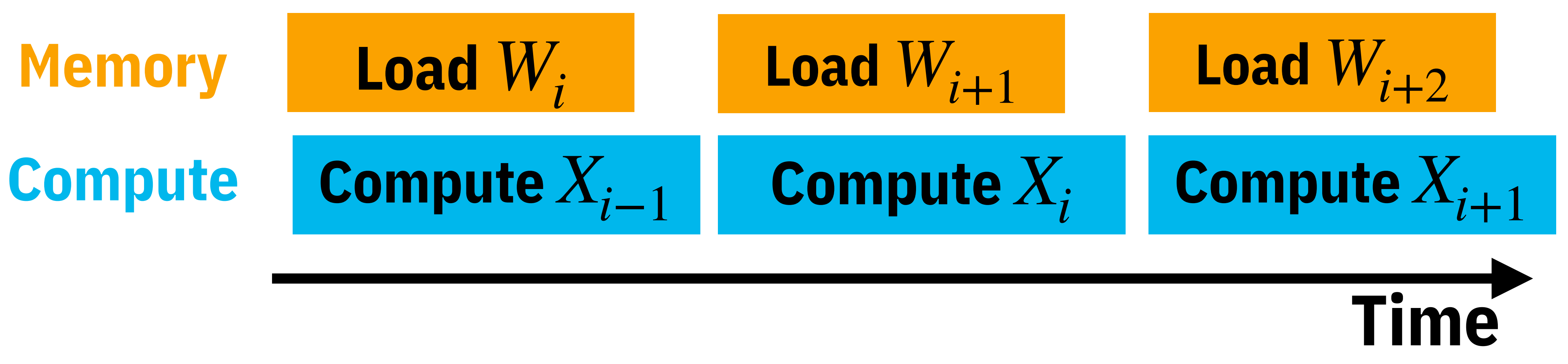
$$\mathbf{w}^{\text{new}} = \mathbf{w} - 2\epsilon \cdot (\mathbf{X}^\top \mathbf{X}\mathbf{w} - \mathbf{X}^\top \mathbf{y})$$

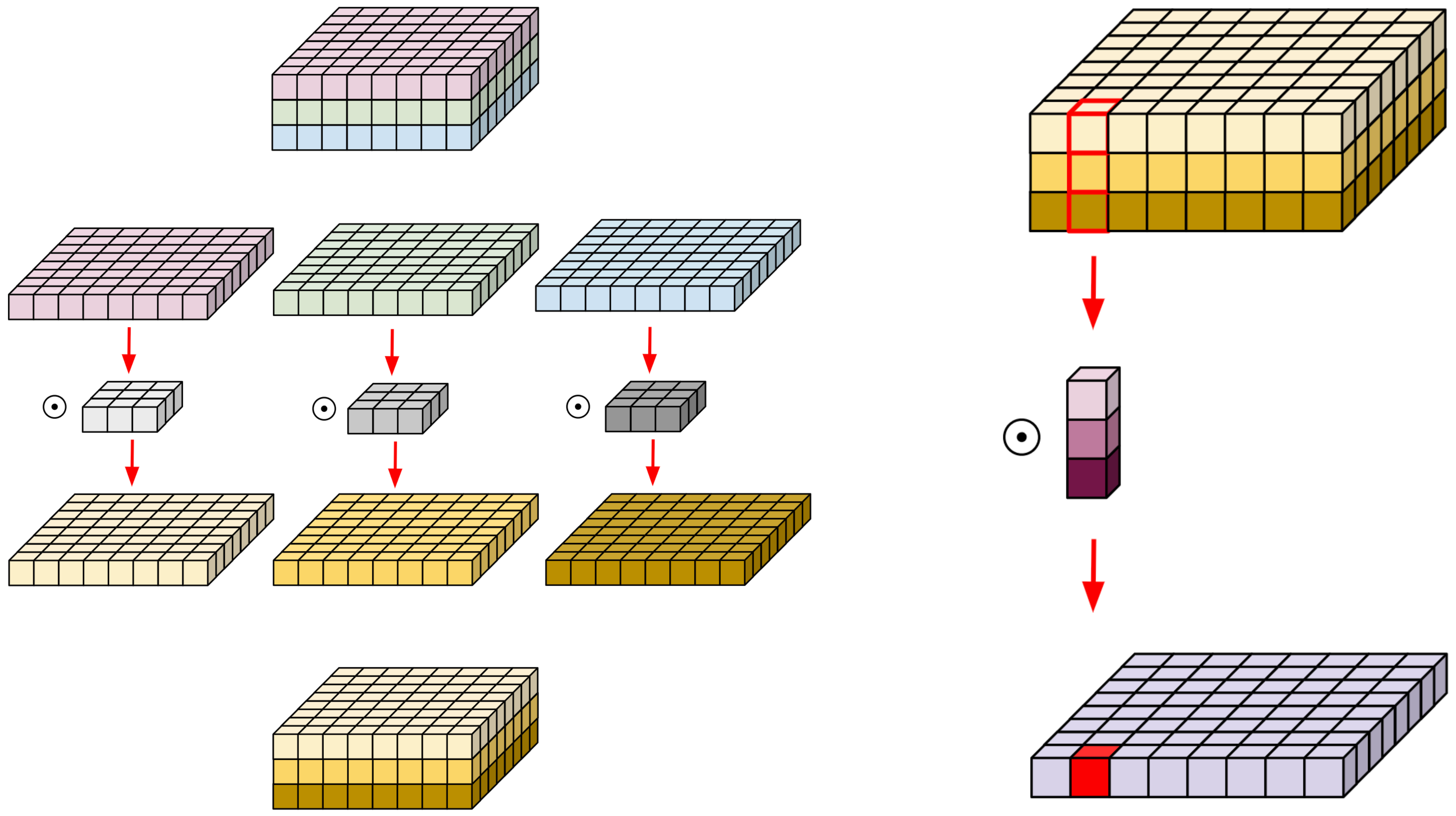
Option 1. Precompute $\mathbf{X}^\top \mathbf{X}$ and reuse.
Option 2. Compute $\mathbf{X}\mathbf{w}$ first
(better with less #steps).

Note. Only requires computing once!
Can be reused for every iteration.

2. Deep Models

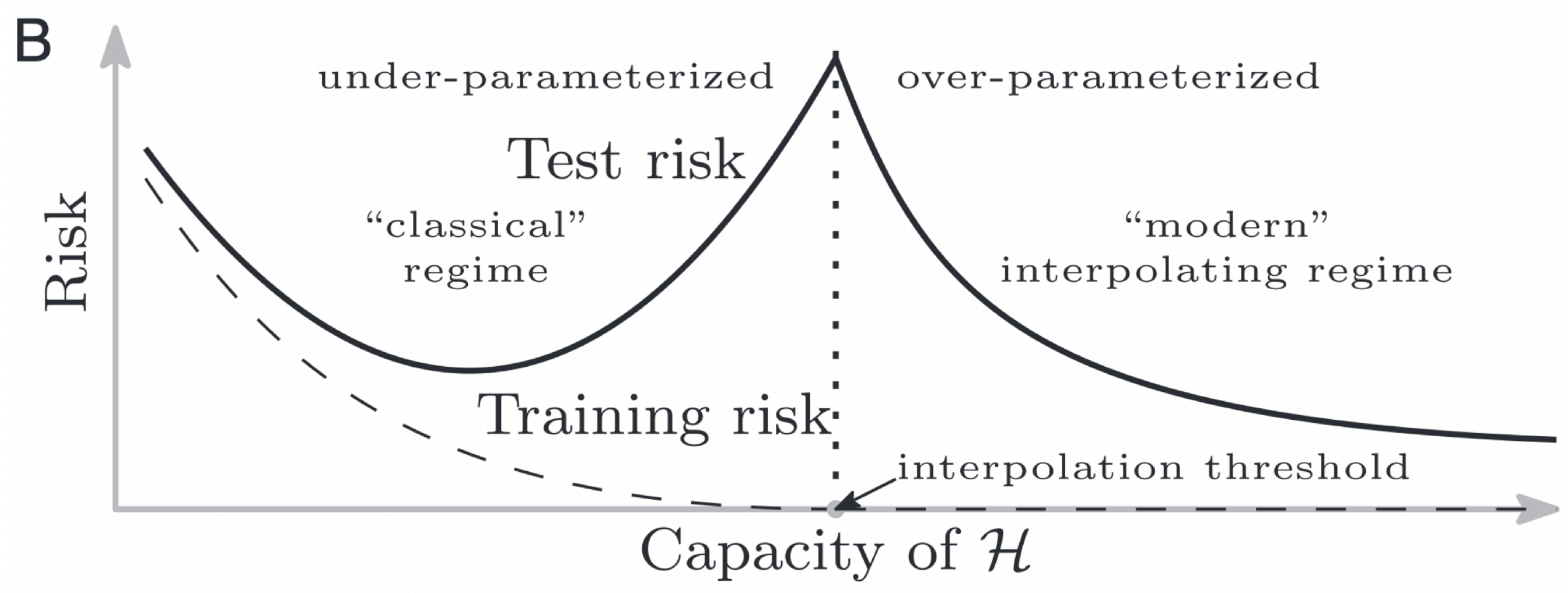


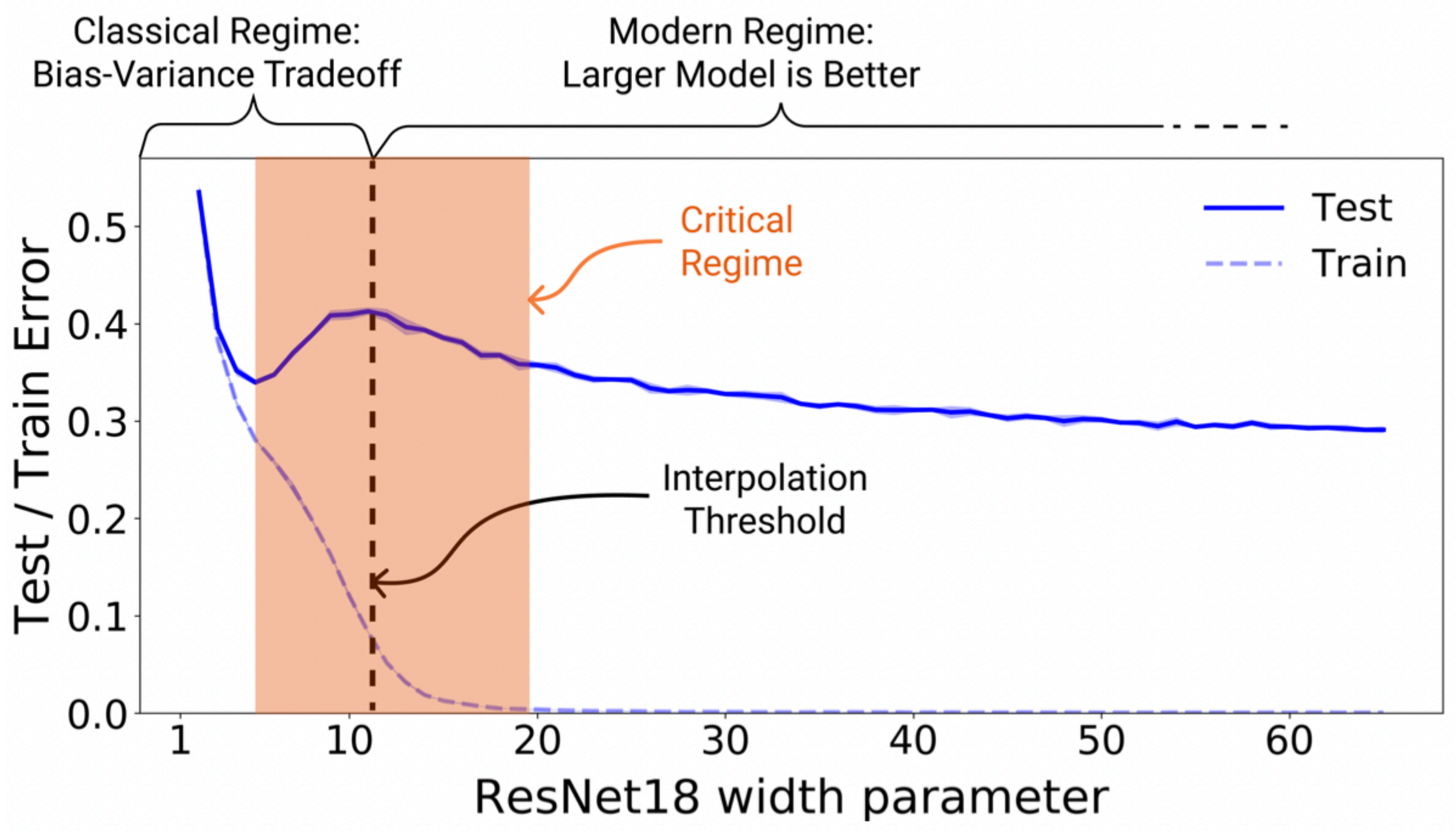




- Repetition of linear models + activation functions
- "Loading all weights" does not work—
  Scheduling of weight loading and computing!
  (overlapping is a common practice)
- The exact solution does not work—backpropagation!
  2 * Forward FLOPs $\approx$ Backward FLOPs
  (Requires us to keep track of the activations and gradients of other layers... memory checkpointing...)
- Modules with smaller compute/memory footprint, e.g., depthwise convolutions
- Efficiency metrics: Time, #Param, Compute, Energy

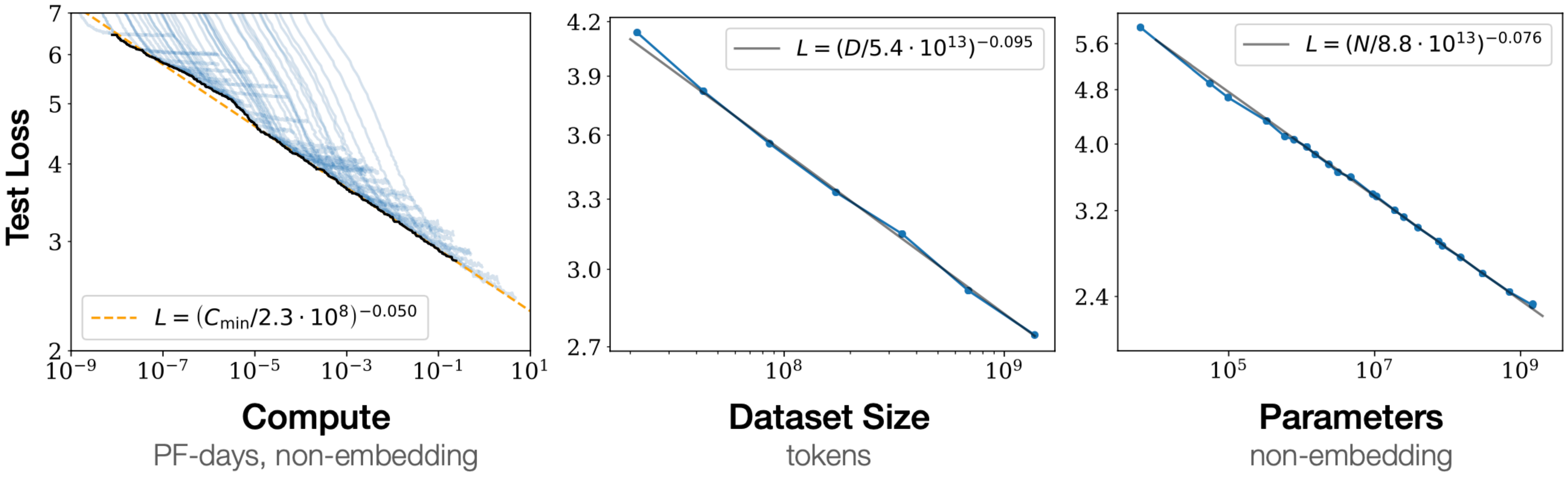
3. Bias-Variance Tradeoff...?





- In the past, people believed that large models (larger than dataset) do not generalize well.
- In these days, people are more sure that larger models generalize better
  (if they are large enough)
- There are some theories to explain why this happens...
  (slightly out of our scope though)

4. Neural Scaling Laws

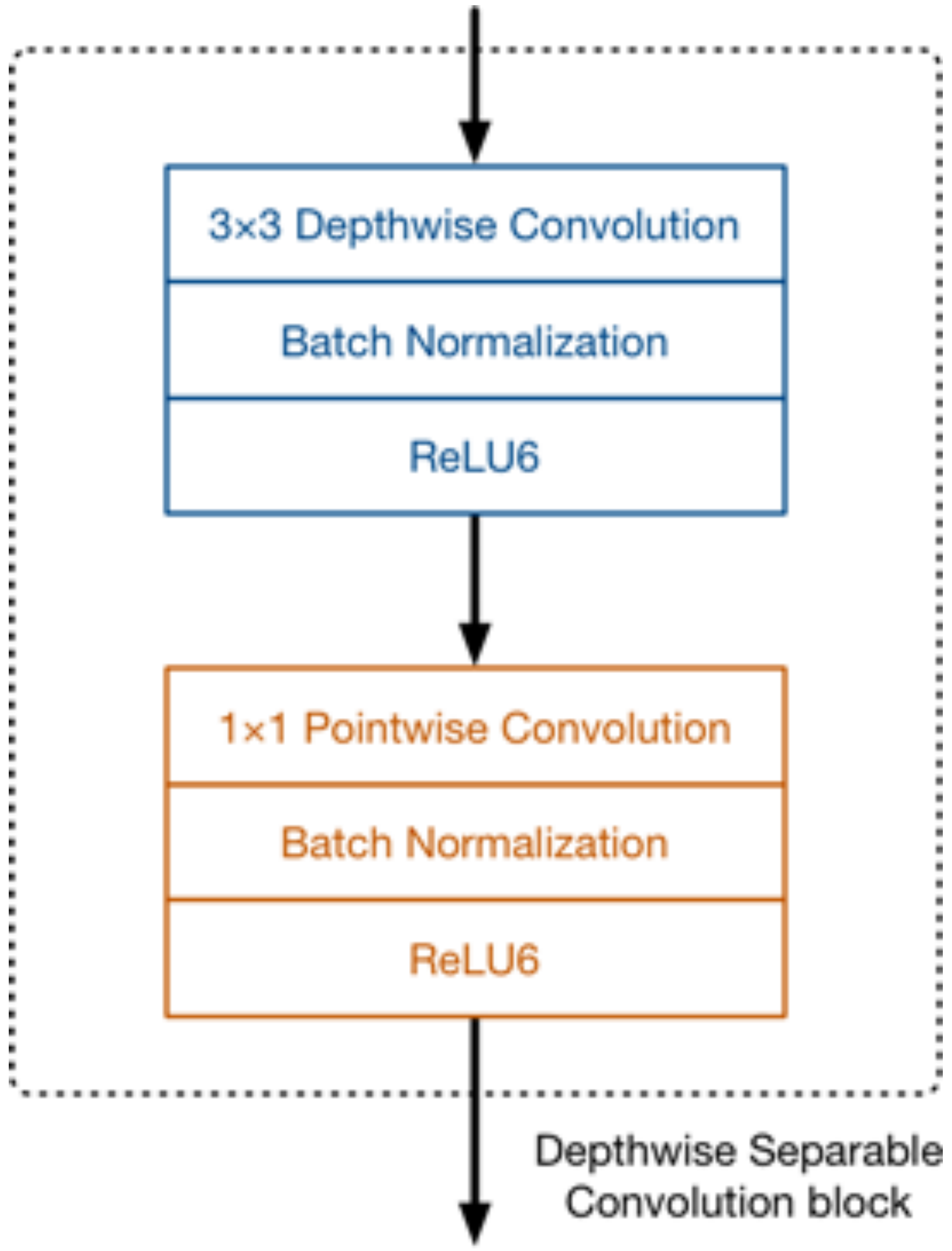


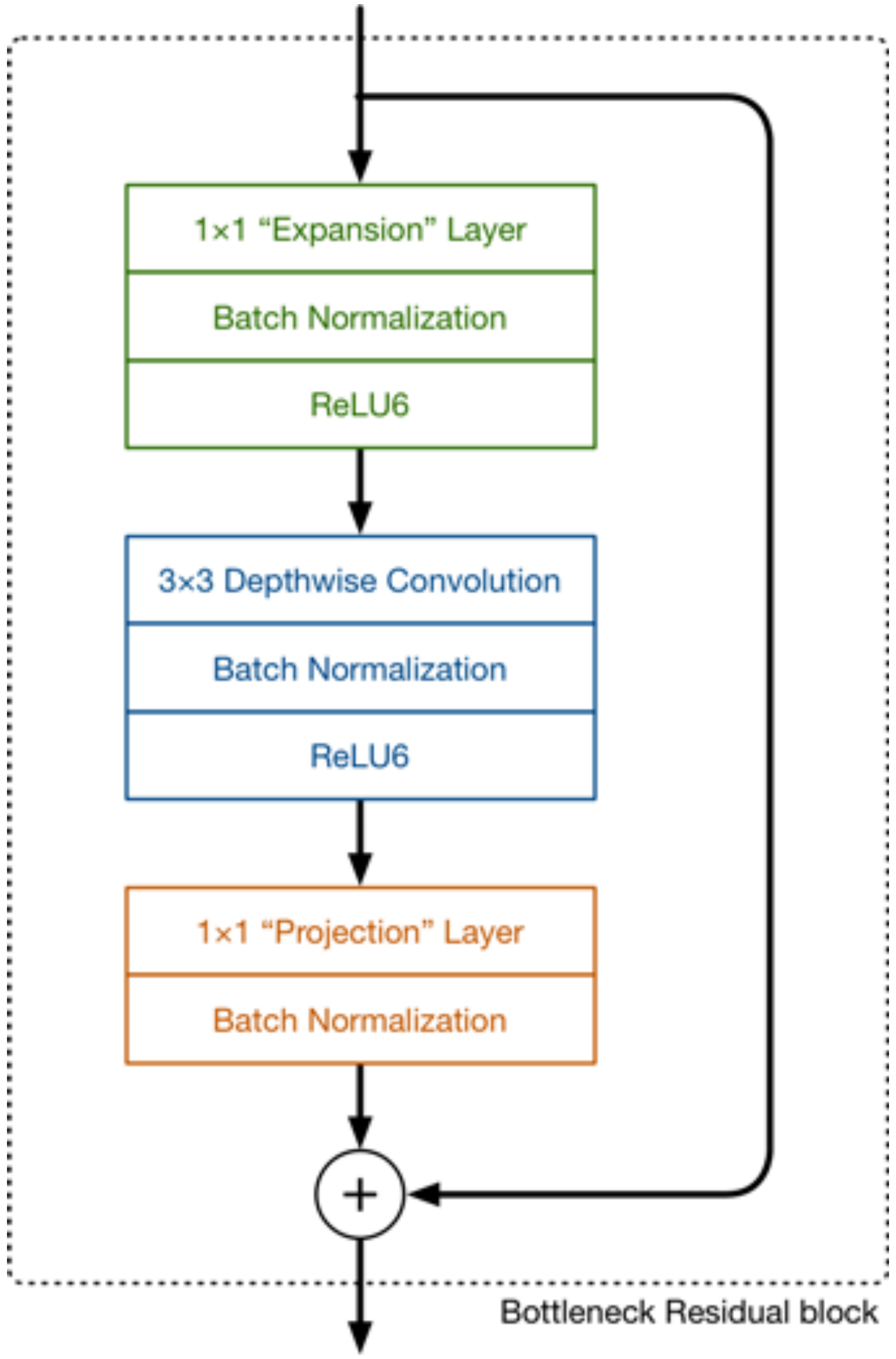
- People find that, neural network scaling follows some **"power law"**
  (happens for architectures that scale well, e.g., transformers)

- The compute, #parameters, and dataset should grow simultaneously—the performance saturates otherwise.

- Recent works use this model to generate compute-efficient models (e.g., Chinchilla)

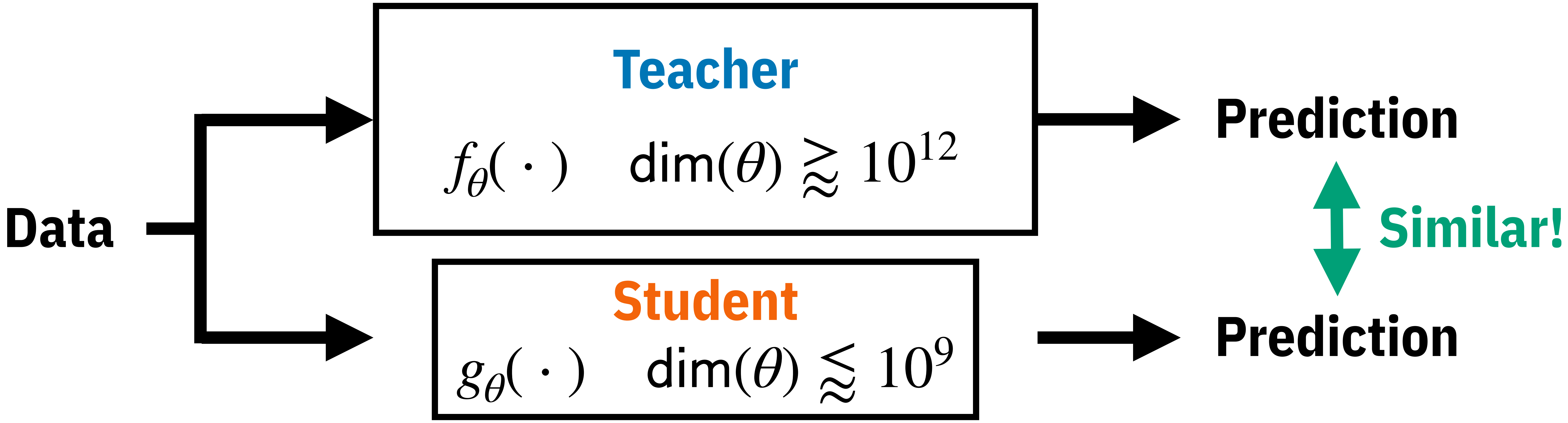
5. Model Architectures





- To design compute/memory-efficient models, people used to design new modules.
  - Fire module, with 1x1 conv (e.g., SqueezeNet)
  - Inverted Residual, with depthwise separable conv (e.g., MBNetv2)
  - ReLU6 (e.g., MBNetv1)
- More recently, people used NAS to build new models based on these modules (e.g., MBNetv3)
- The focus is moving toward trainability—
  via scaling laws (e.g., EfficientNet)
  via memory-efficiency (e.g., MCUNets)

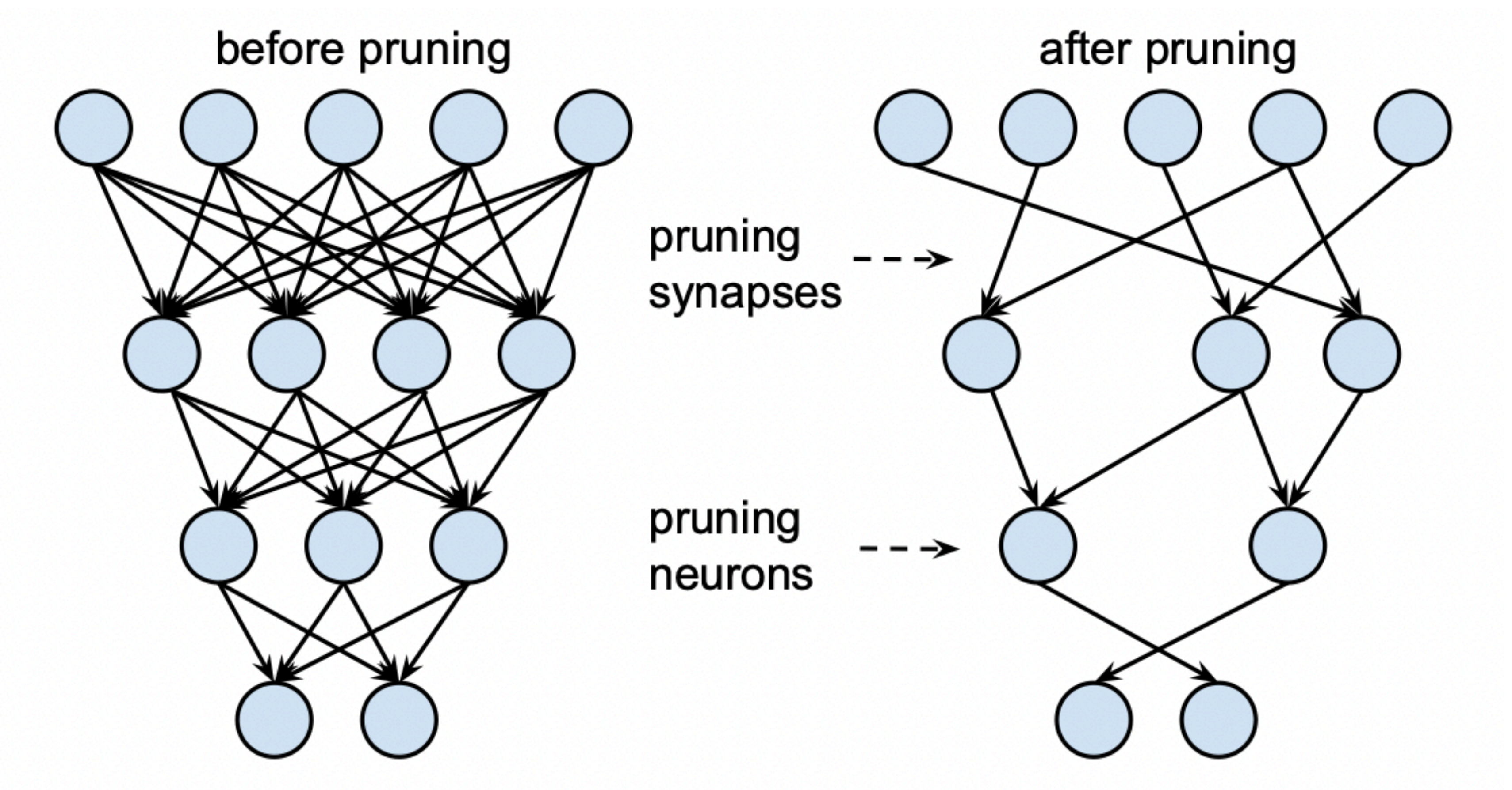
6. Knowledge Distillation



- Enhance the training of a small model, by training it to approximate the output of a large model (or itself)—Benefits in terms of generalization, rather than approximation.
  - Mimic the softmax outputs
  - Mimic the intermediate activations
  - Mimic the relationship between data samples

7. Pruning

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 0 & \tilde{a}_3 & \tilde{a}_4 \\ \tilde{a}_5 & 0 & \tilde{a}_7 & 0 \\ \tilde{a}_9 & 0 & 0 & \tilde{a}_{12} \\ \tilde{a}_{13} & 0 & \tilde{a}_{15} & \tilde{a}_{16} \end{bmatrix}$$



- Add zeros to the weights of a neural network, so that the associated compute/memory is no longer necessary.
- Key questions are **what to prune**, **when to prune**, **how much to prune**.
  - A popular method is to prune by magnitude, gradually by cubic schedule, with heuristic layerwise sparsity (e.g., Gale et al., (2019))
    Pruning at initialization, or using Hessian-based criteria is also popular.
- Difficult to exploit the benefit if there is no structure to the zeros—prune whole filters or force patterns.
- Note: There is a recent work that argues “for some tasks, you can’t do pruning” (I don’t agree though)

8. Activation Sparsity

$$\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \end{bmatrix} \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \\ x_5 & x_6 \end{bmatrix}$$

Dense

$$\begin{bmatrix} w_1 & 0 & w_3 \\ 0 & w_5 & w_6 \end{bmatrix} \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \\ x_5 & x_6 \end{bmatrix}$$

Sparse Weight

$$\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \end{bmatrix} \begin{bmatrix} x_1 & 0 \\ x_3 & x_4 \\ 0 & x_6 \end{bmatrix}$$

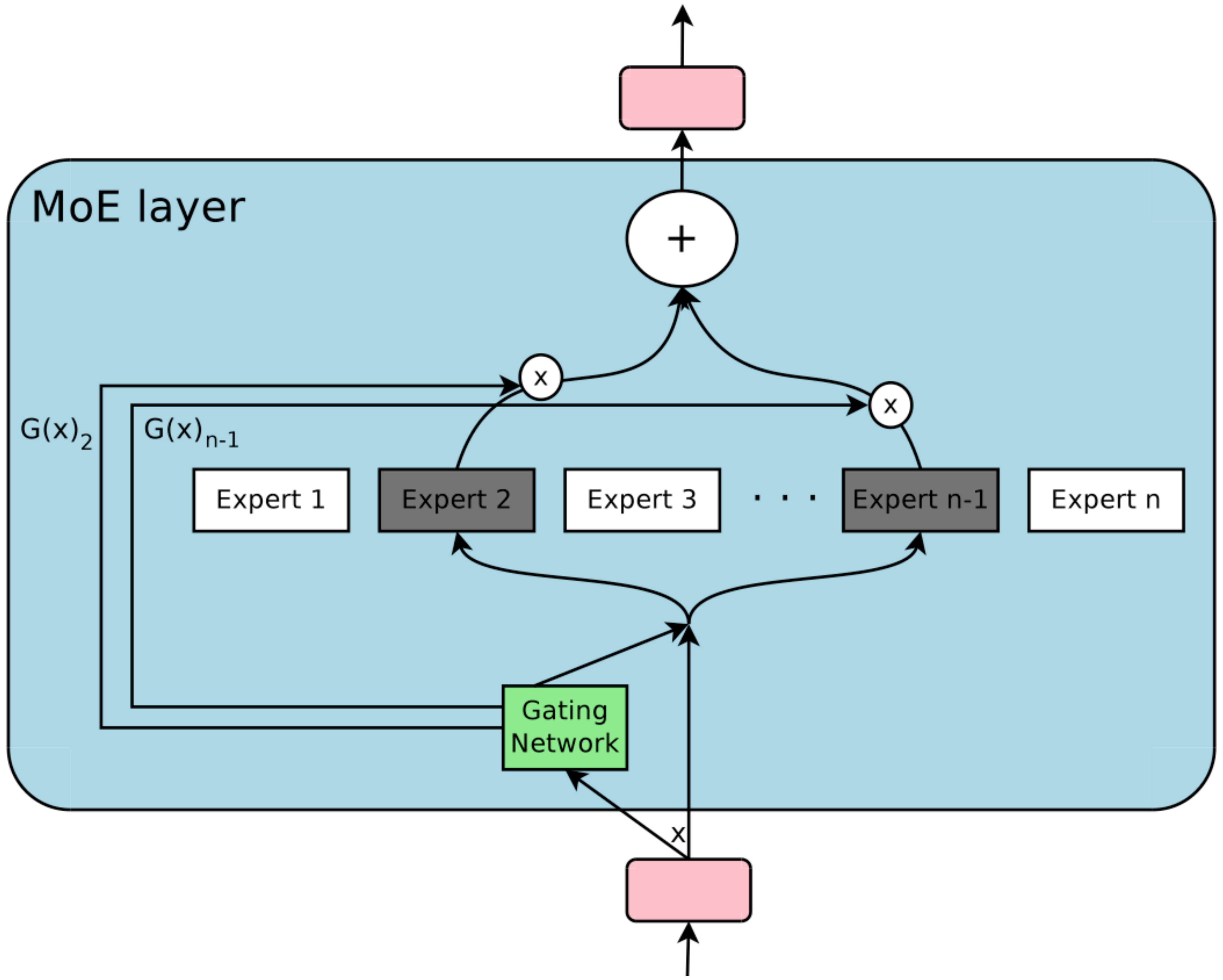
Sparse Activation

- Sometimes we sparsify the **activations** instead of weight tensors.
  (Often, there are naturally arising sparsity)

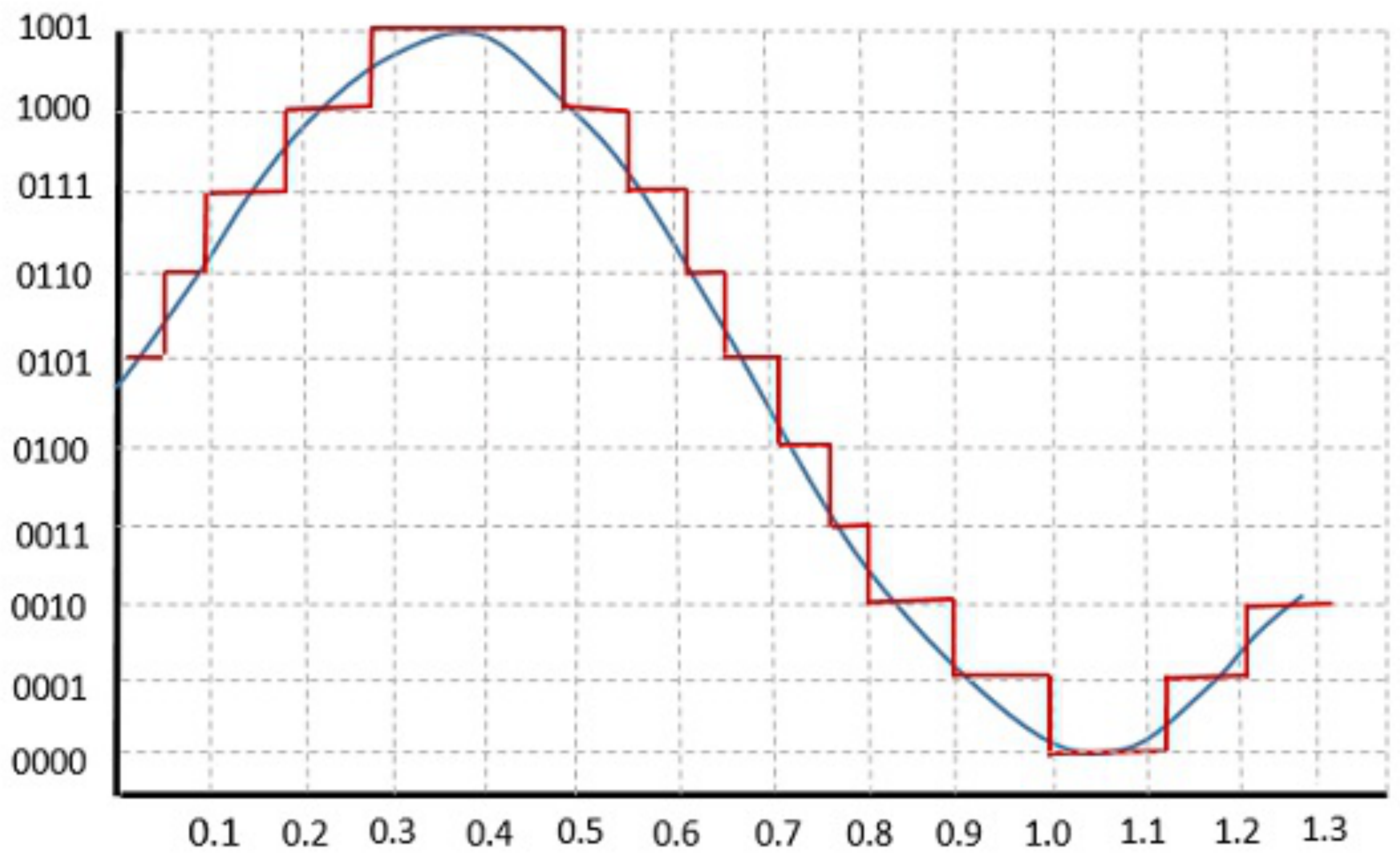
- We modify activation functions (FATReLU) or regularizers (Hoyer regularization) to encourage sparsity

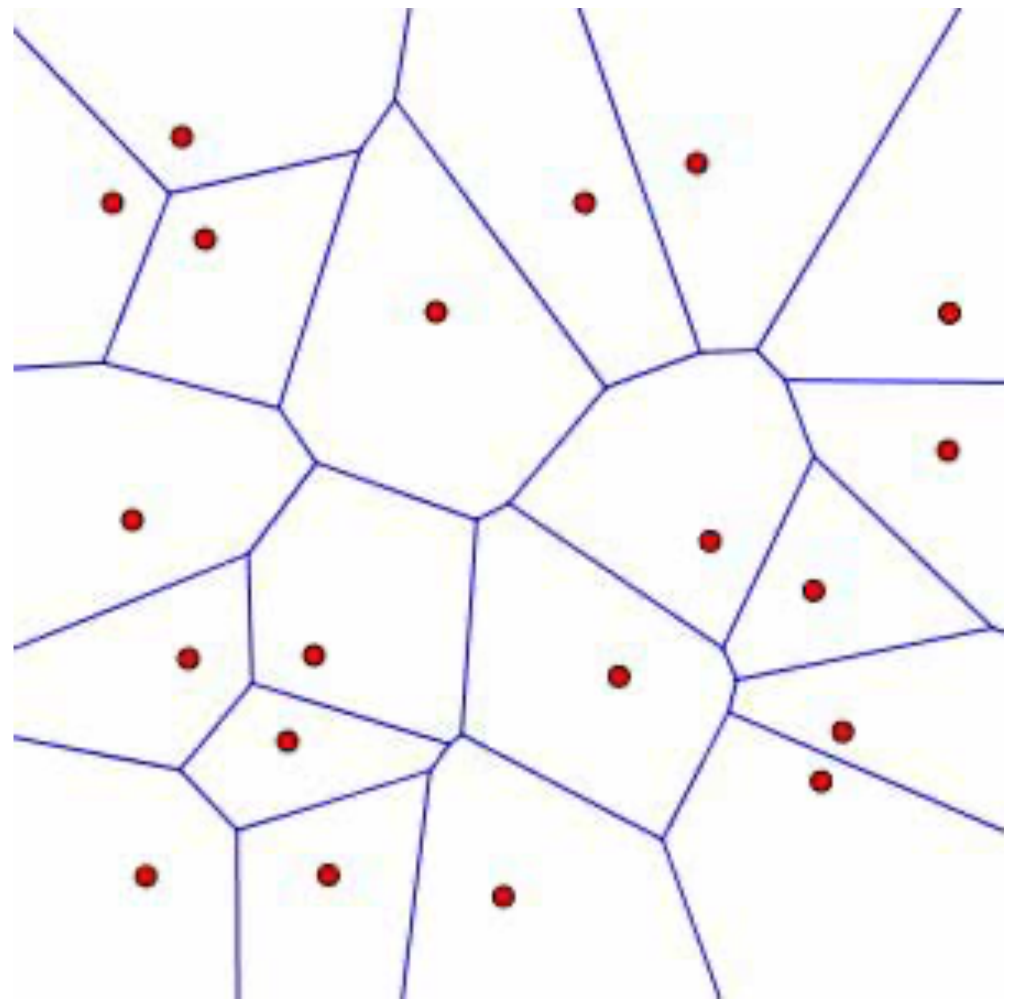
- Sometimes we force the activation sparsity, by top-k operations or external routers.

  - The latter variant, called Mixture-of-Experts, are known to be training-efficient.



9. Quantization





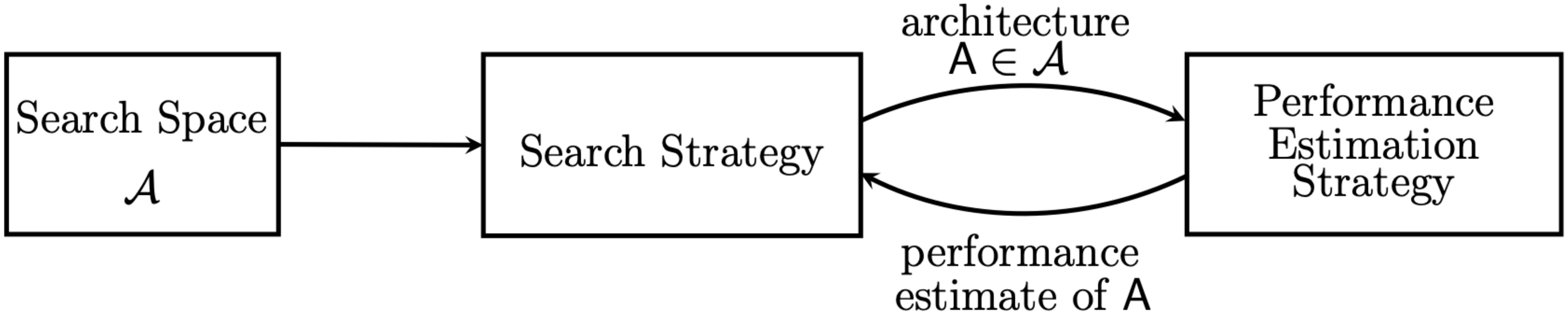
- Dropping the precision of weights/activations to lower bits (e.g., 4/8 bits) for faster compute & smaller size
  (As we use floating points, this gives rise to many different formats by #bits for exponent, e.g., bfloat16)

- A popular way is **“linear quantization,”** which quantizes weights using uniform grids—
  key parameters are the scaling and shifting factors.

  - Post-Training Quantization: No special training before/after quantization

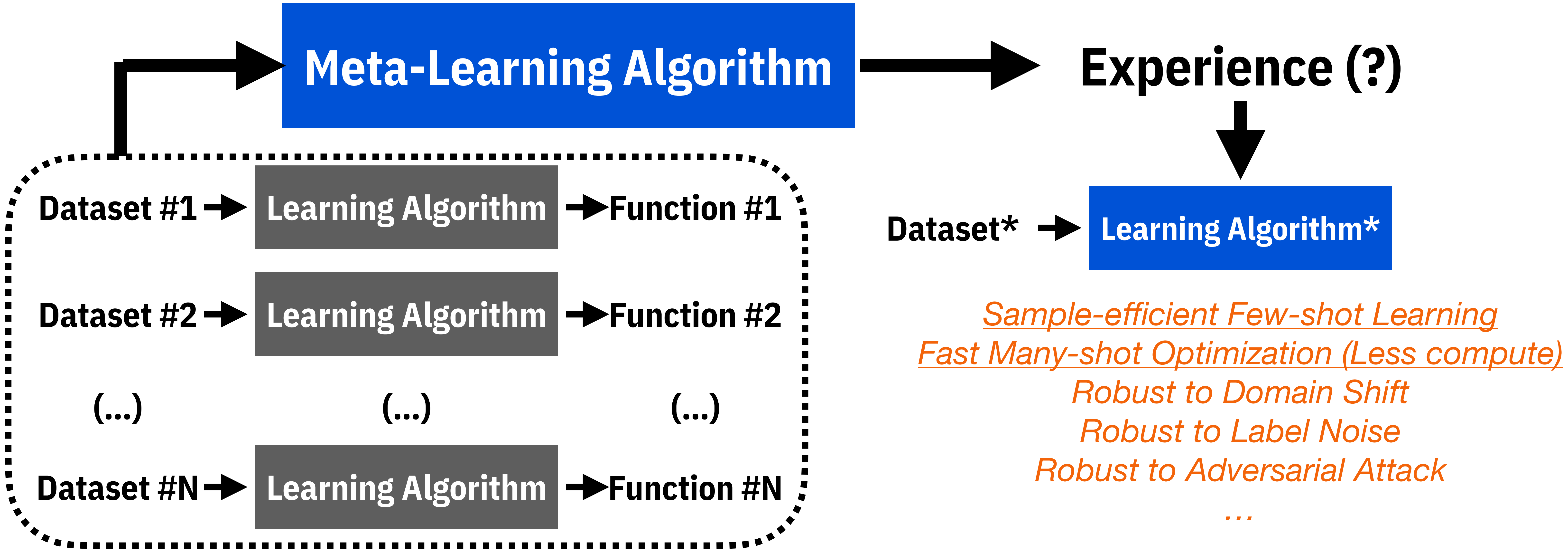
  - Quantization-aware training: Additional training, with simulating the noise from quantization.

10. Neural Architecture Search



- Look for the best-performing neural network architecture, within some search space
- Key elements are: Search Space, Search Strategy, and Performance estimation.
  - **Search space:** Networks as a repetition of blocks, where blocks are composed of repeated layers. For each layer, search one specific configuration of efficient modules.
  - **Search strategy:** The decisions are discrete—use RL / evolutionary strategy.
  - **Performance estimation:** Full training is difficult—shorter training or share weights (or zero-cost proxy)

11. Meta-Learning



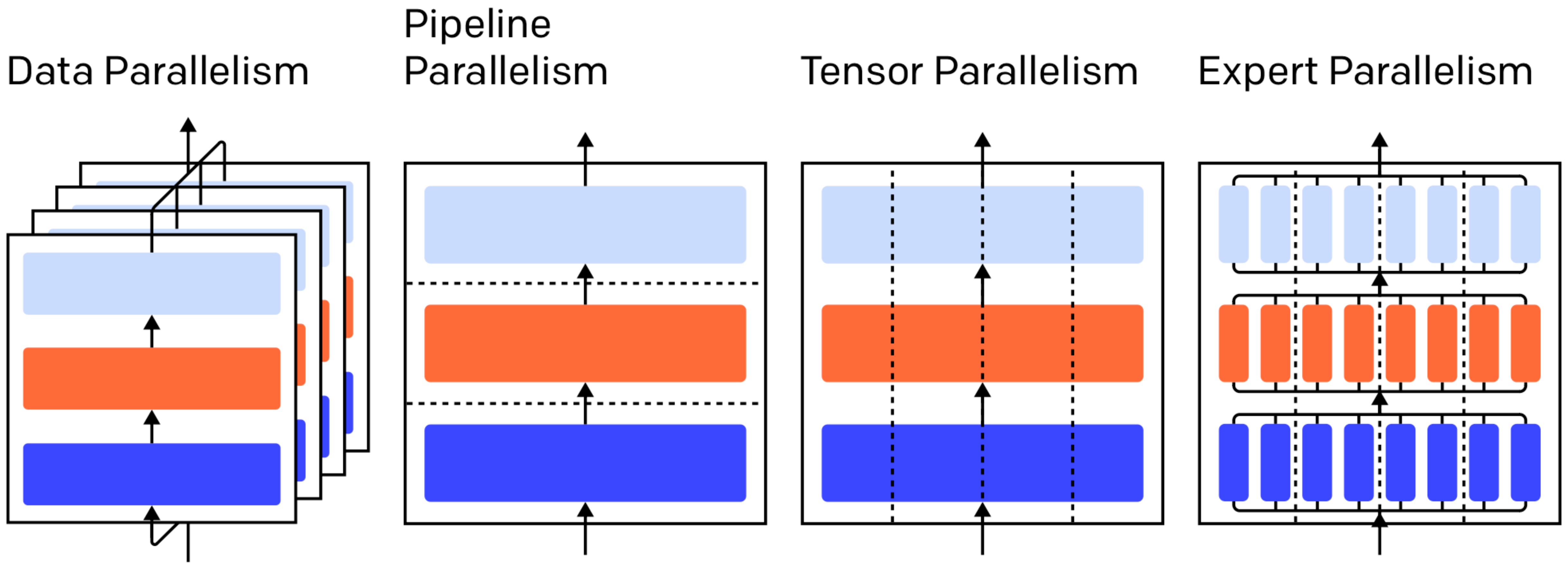
- For the training-efficiency, transfer the knowledge from task 1, ..., task N to task N+1.

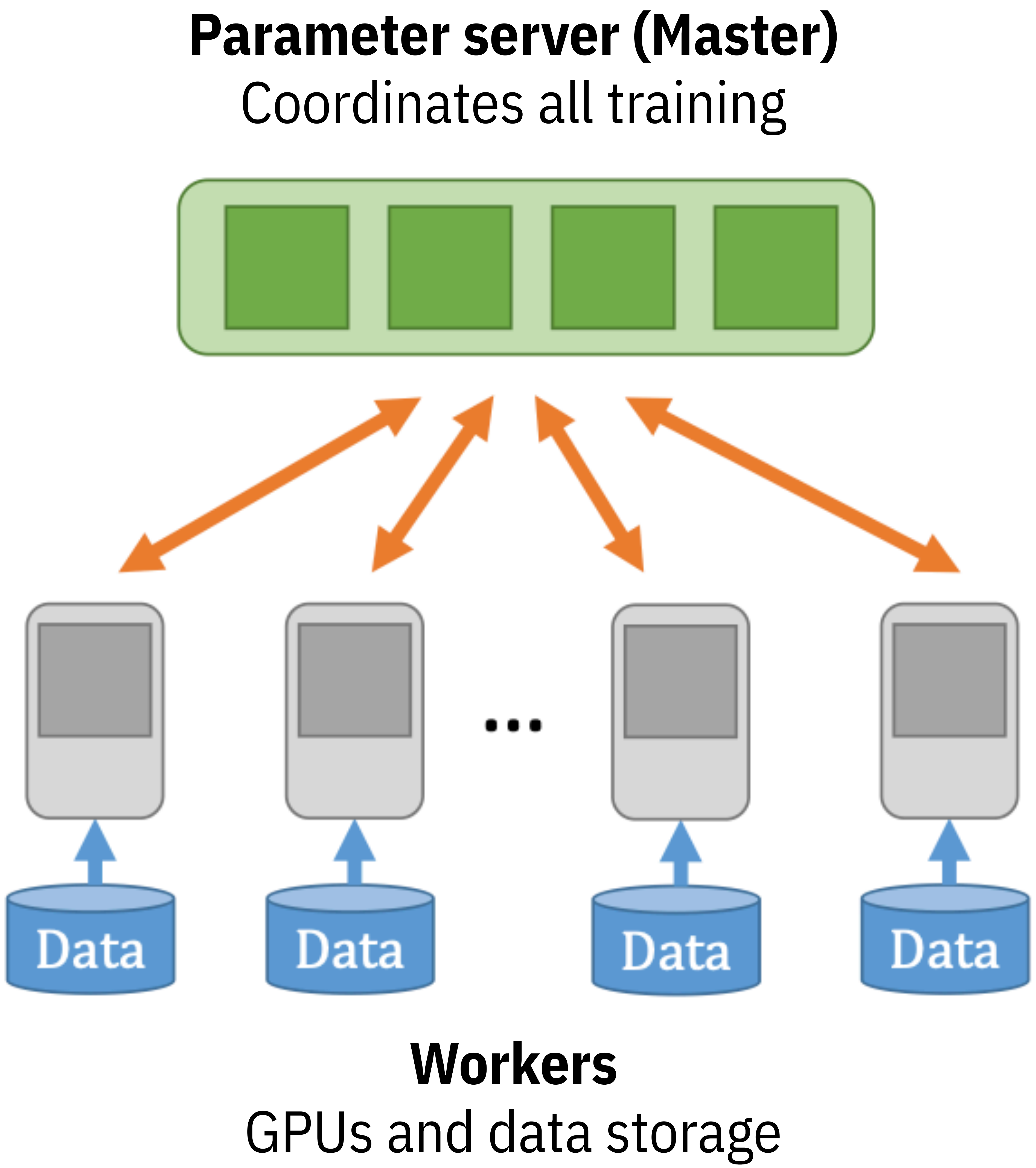
- A popular form is **MAML**—we transfer the weights of a model to a new task, where we train the weights so that they can adapt to a new task within a small number of steps.

- Another interesting application is training the "optimizer" (like Adam).

12. Distributed Training

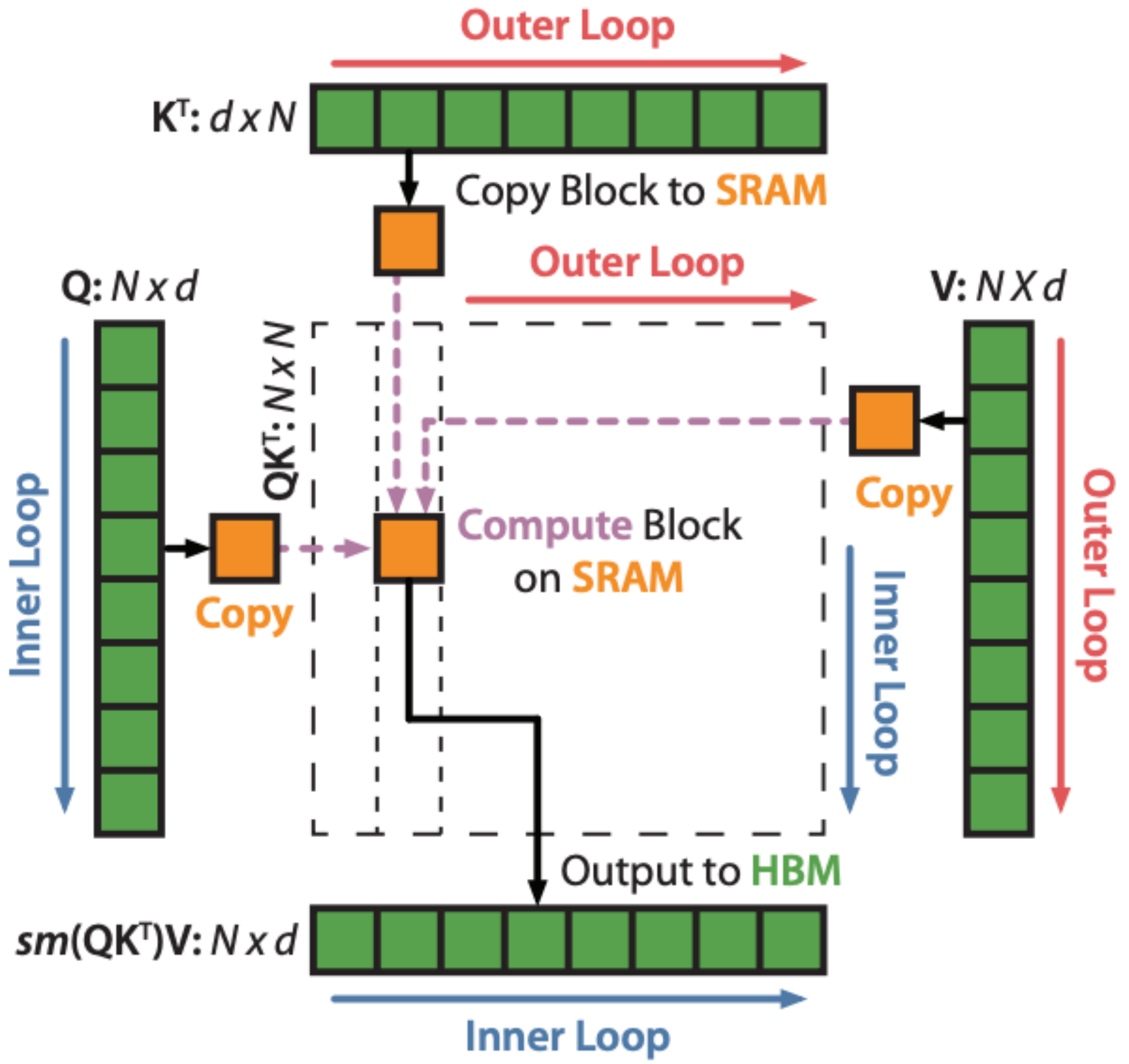
- For training large-scale models, we exploit many parallelisms.
  - **Data parallelism:** Data are distributed among workers,
    Model parameters are shared,
    Training synchronized by communication.
  - **Model parallelism:** Model is distributed—each GPU gets
    (1) some layers (pipeline)
    (2) some part of each layer (tensor)
    (3) an expert of MoE (expert)

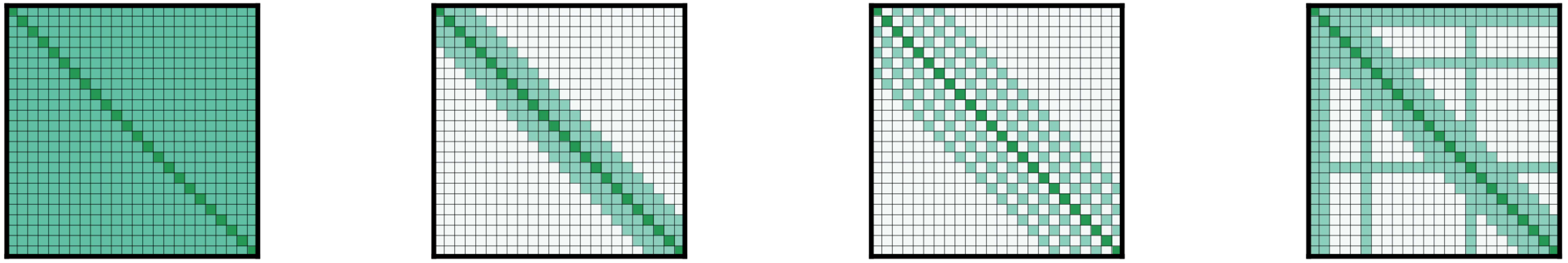




13. Efficient Transformers

- Transformer architectures view any data as a sequence– Compute grows quadratically w.r.t. sequence length!
- **Efficient attention** mechanisms use sparse attention among each token inside a sequence
- **Flashattention** focuses on optimizing the data movement
- **Token pruning/merging** reduce the number of intermediate tokens for a faster inference and training.





(a) Full n^2 attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window