

LoRA

**Low-Rank Adaptation
Of Large Language
Models**

Key Idea

- Fine-tuning for different task
- More Efficient and lowers the hardware
- Approach: Merge the trainable matrices with the frozen weights and finetuning matrices for downstream tasks
- Can combine with another fintuning method (prefix tuning, ...)

Problems of Transfer Learning in LLMs

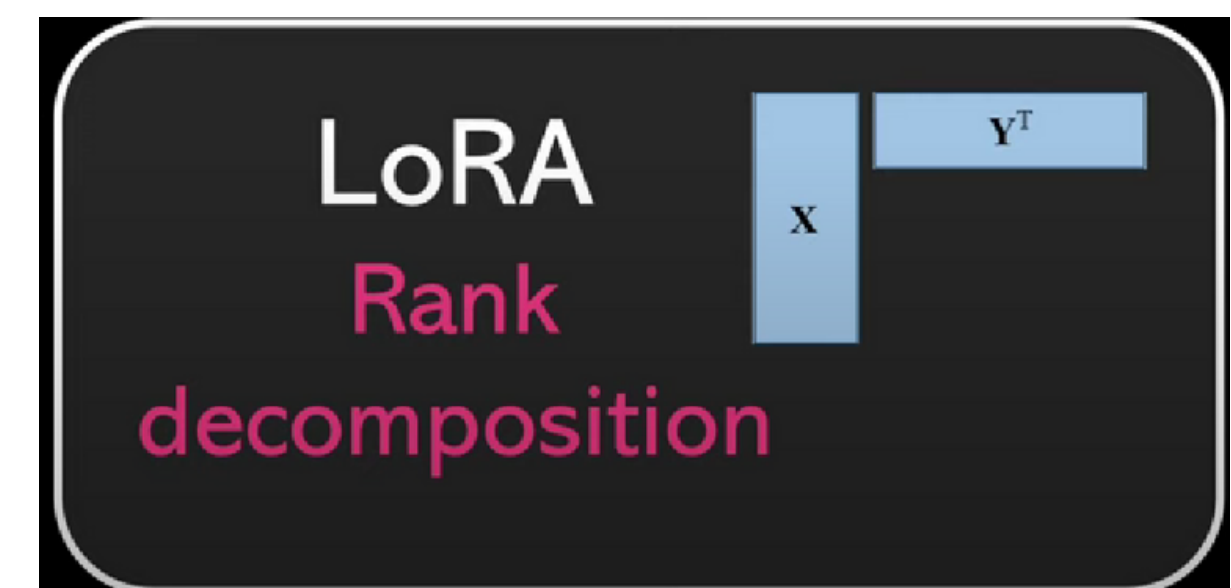
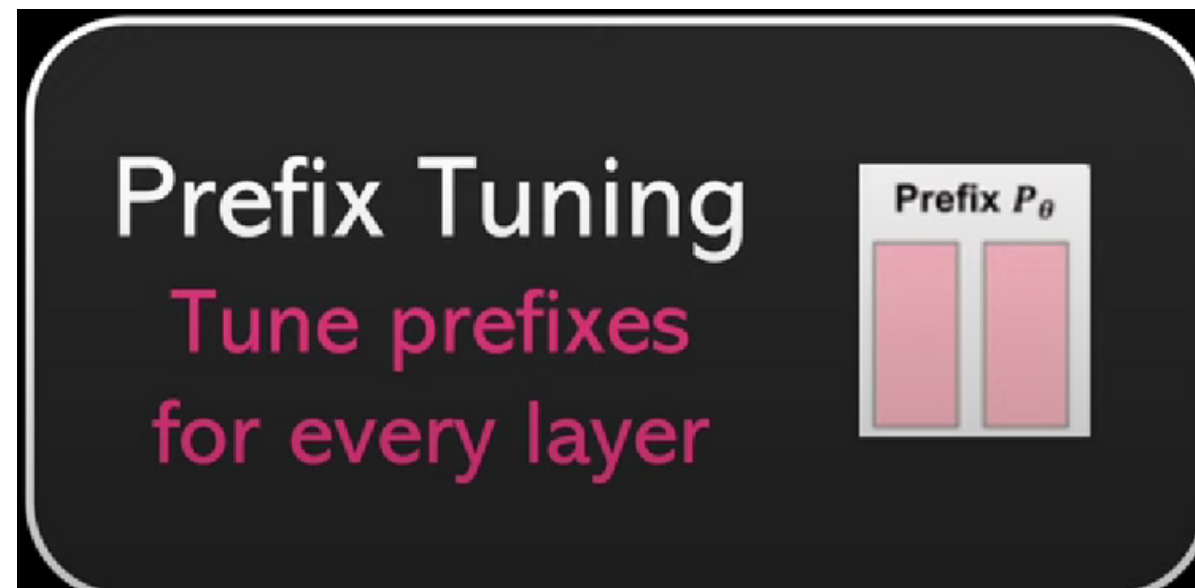
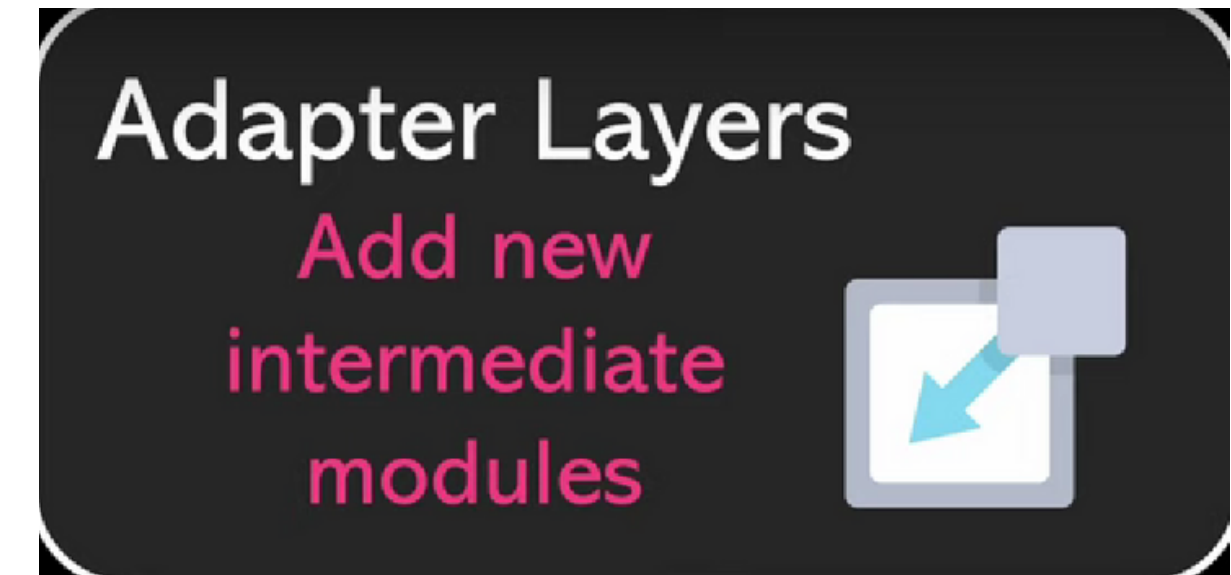
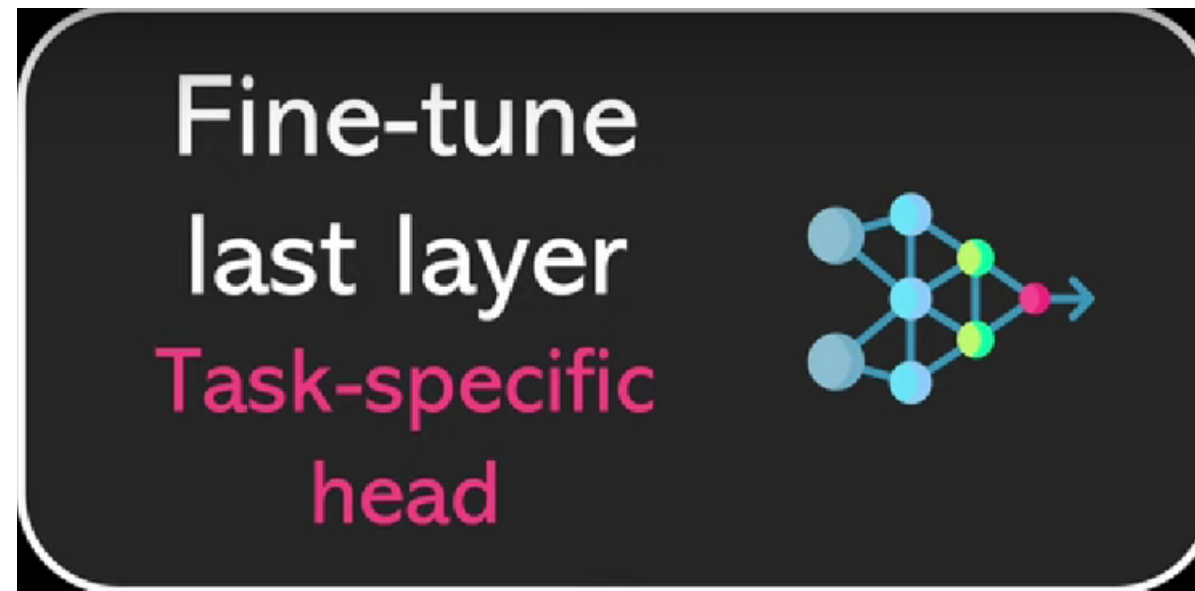
- Much much parameters
 - Get a lot of time and resources for training
-

$$\max_{\Phi} \sum_{(x,y) \in \mathcal{Z}} \sum_{t=1}^{|y|} \log (P_{\Phi}(y_t|x, y_{<t}))$$

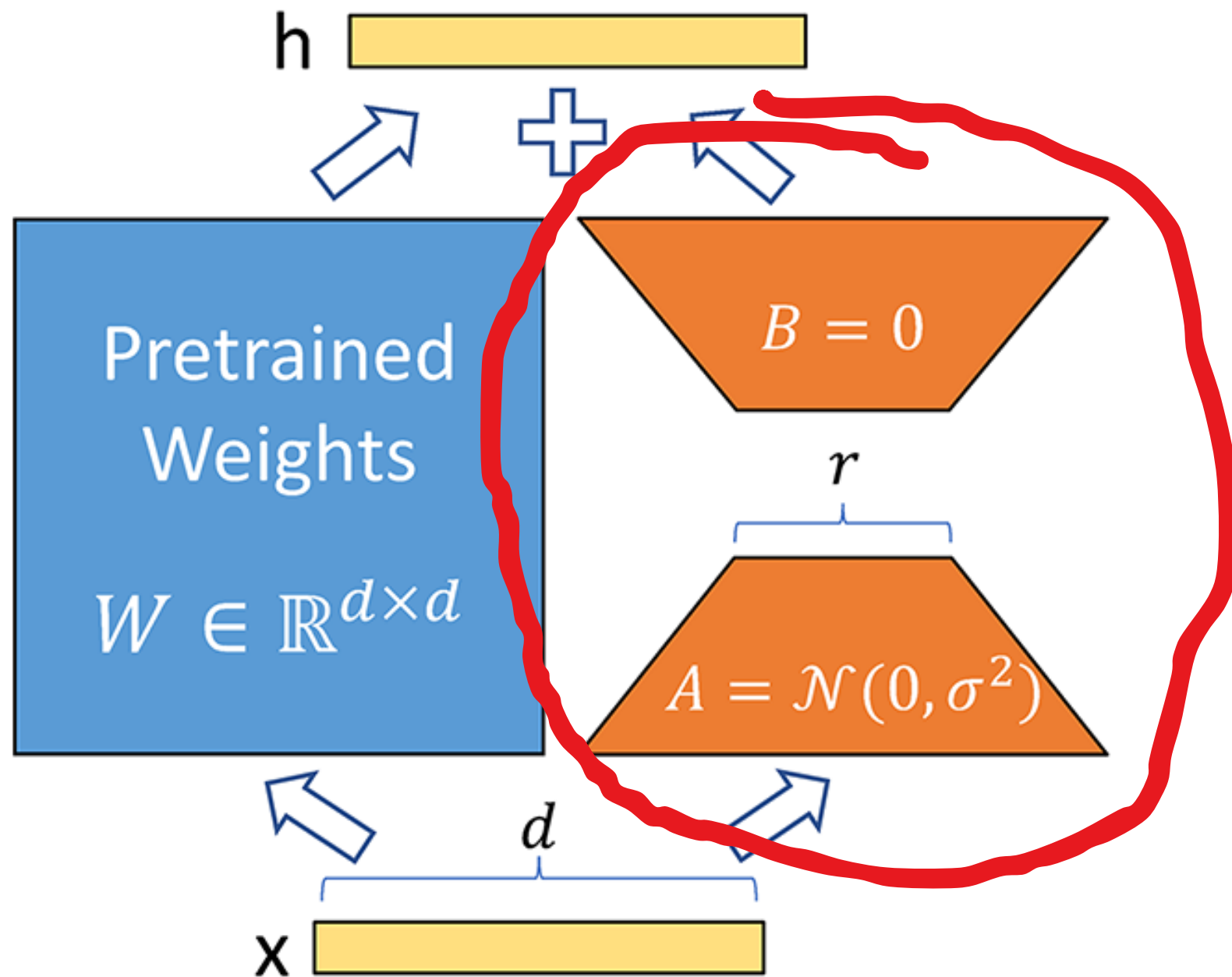
- Adopt a more parameter-efficient approach

$$\max_{\Theta} \sum_{(x,y) \in \mathcal{Z}} \sum_{t=1}^{|y|} \log (p_{\Phi_0 + \Delta \Phi(\Theta)}(y_t|x, y_{<t}))$$

Some Another Approach



Method

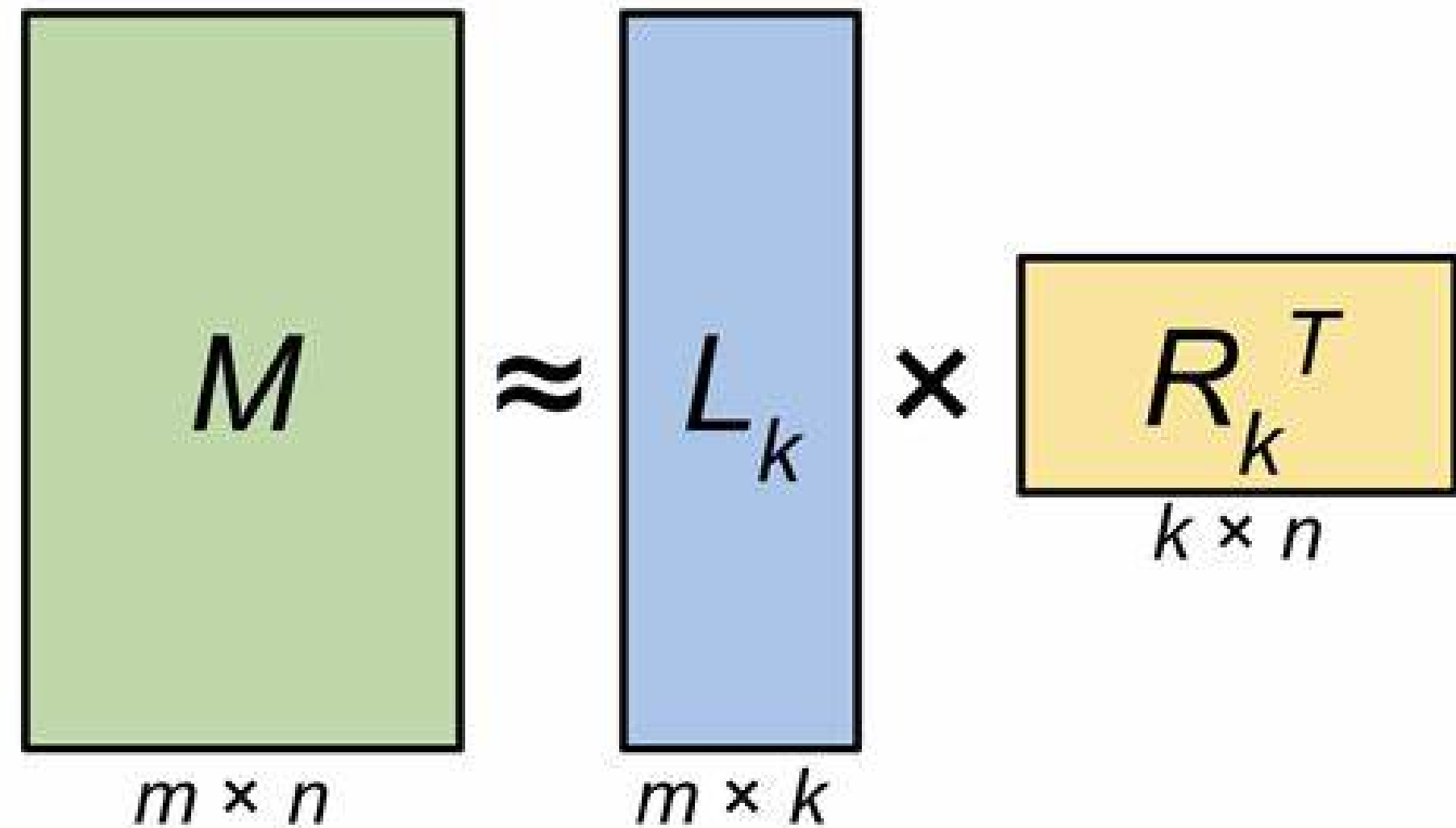


using low “insstrinsic dimension”

$$h = W_0 x + \Delta W x = W_0 x + B A x$$

Rank decomposition matrices

$$h = W_0 x + \Delta W x = W_0 x + B A x$$



Scaling Factor Alpha

We illustrate our reparametrization in Figure 1. We use a random Gaussian initialization for A and zero for B , so $\Delta W = BA$ is zero at the beginning of training. We then scale ΔW by $\frac{\alpha}{r}$, where α is a constant in r . When optimizing with Adam, tuning α is roughly the same as tuning the learning rate if we scale the initialization appropriately. As a result, we simply set α to the first r we try and do not tune it. This scaling helps to reduce the need to retune hyperparameters when we vary r (Yang & Hu, 2021).

constant α .

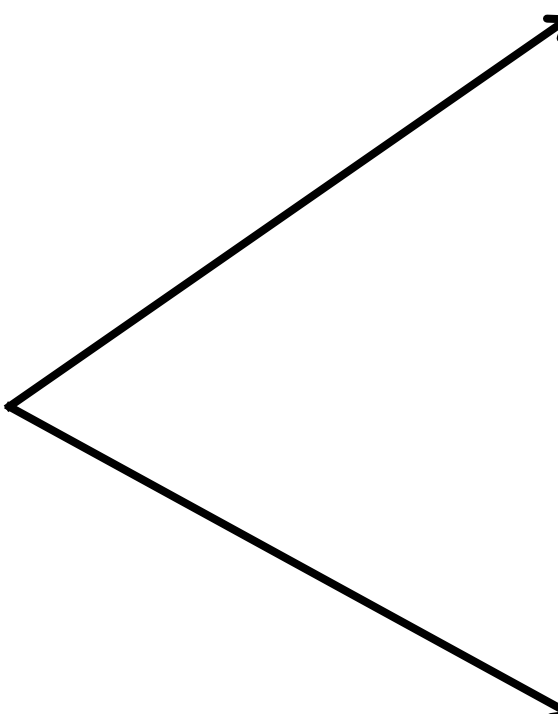


$$W_0 + \Delta W = W_0 + \underbrace{BA}_{\frac{\alpha}{r}}$$

$B \in \mathbb{R}^{d \times r}$
 $A \in \mathbb{R}^{r \times k}$
 $\text{rank } r \ll \min(d, k)$

Scaling factor
Rank

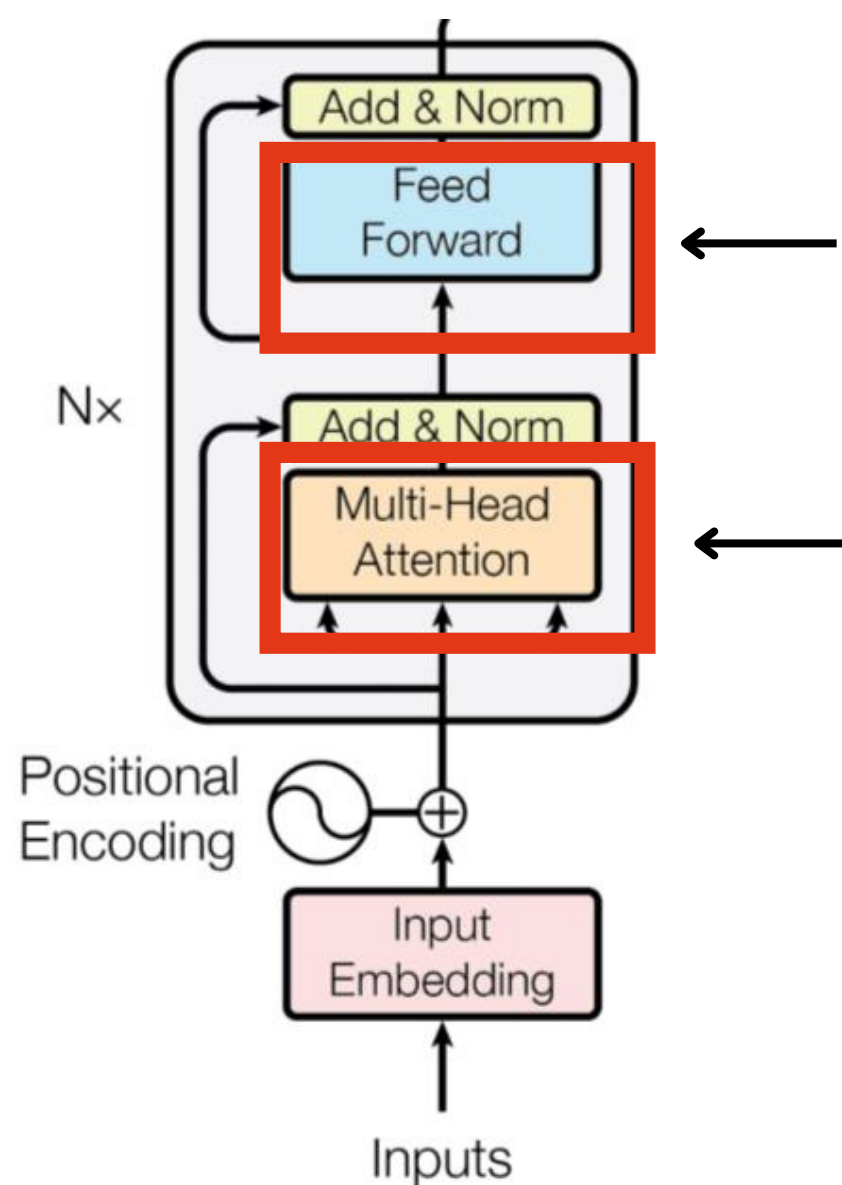
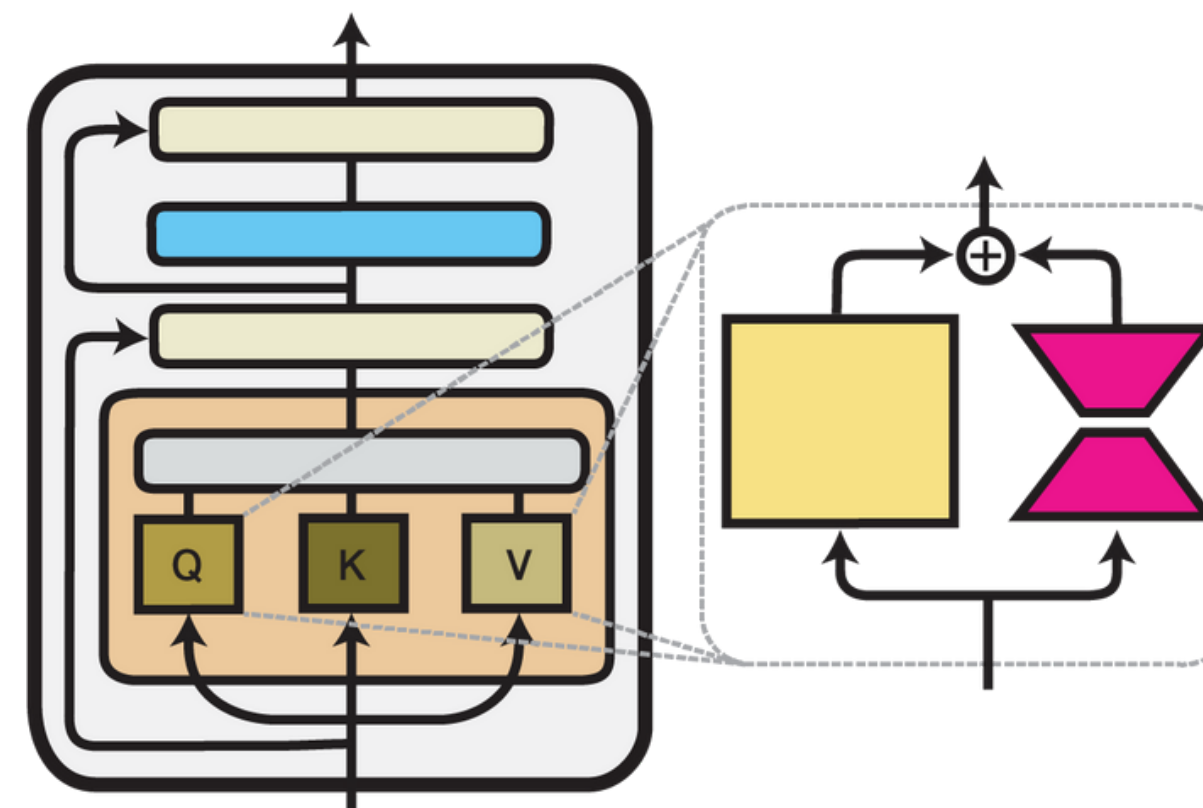
LoRA Inference



A Generalization of Full Fine-tuning. A more general form of fine-tuning allows the training of a subset of the pre-trained parameters. LoRA takes a step further and does not require the accumulated gradient update to weight matrices to have full-rank during adaptation. This means that when applying LoRA to all weight matrices and training all biases², we roughly recover the expressiveness of full fine-tuning by setting the LoRA rank r to the rank of the pre-trained weight matrices. In other words, as we increase the number of trainable parameters³, training LoRA roughly converges to training the original model, while adapter-based methods converges to an MLP and prefix-based methods to a model that cannot take long input sequences.

No Additional Inference Latency. When deployed in production, we can explicitly compute and store $W = W_0 + BA$ and perform inference as usual. Note that both W_0 and BA are in $\mathbb{R}^{d \times k}$. When we need to switch to another downstream task, we can recover W_0 by subtracting BA and then adding a different $B'A'$, a quick operation with very little memory overhead. Critically, this

LoRA In Transformer



In principle, we can apply LoRA to any subset of weight matrices in a neural network to reduce the number of trainable parameters. In the Transformer architecture, there are four weight matrices in the self-attention module (W_q, W_k, W_v, W_o) and two in the MLP module. We treat W_q (or W_k, W_v) as a single matrix of dimension $d_{model} \times d_{model}$, even though the output dimension is usually sliced into attention heads. We limit our study to **only adapting the attention weights** for downstream tasks and freeze the MLP modules (so they are not trained in downstream tasks) both for simplicity and parameter-efficiency. We further study the effect on adapting different types of attention weight matrices in a Transformer in Section 7.1. We leave the empirical investigation of adapting the MLP layers, LayerNorm layers, and biases to a future work.

Reference

- <https://www.youtube.com/watch?v=t509sv5MT0w&t=882s>
- <https://arxiv.org/pdf/2106.09685.pdf>
- https://www.youtube.com/watch?v=X4VvO3G6_vw