

Seminar 2: Efficiency of LLM

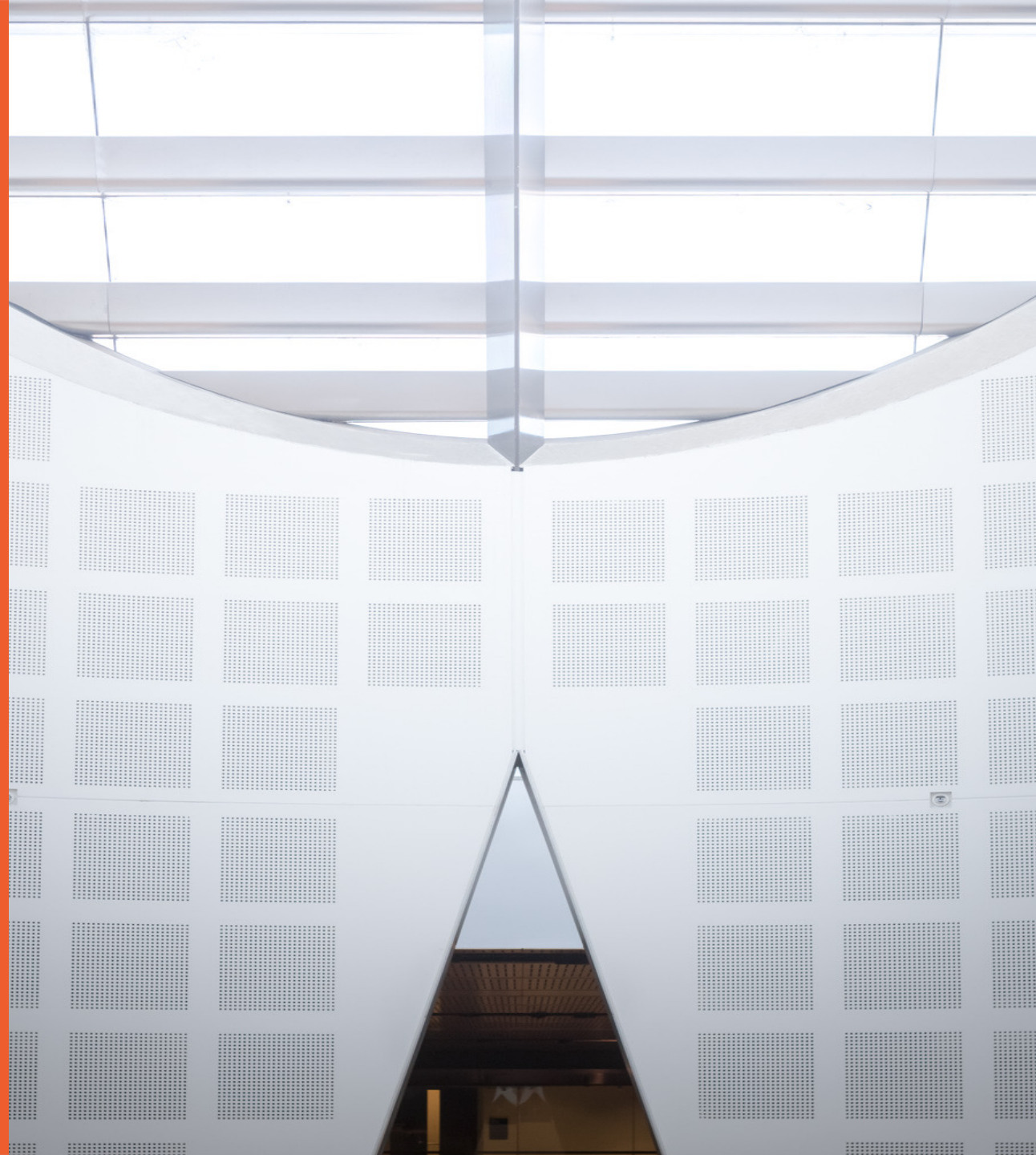
Xiaohuan Pei

xiaohuan.pei@sydney.edu.au

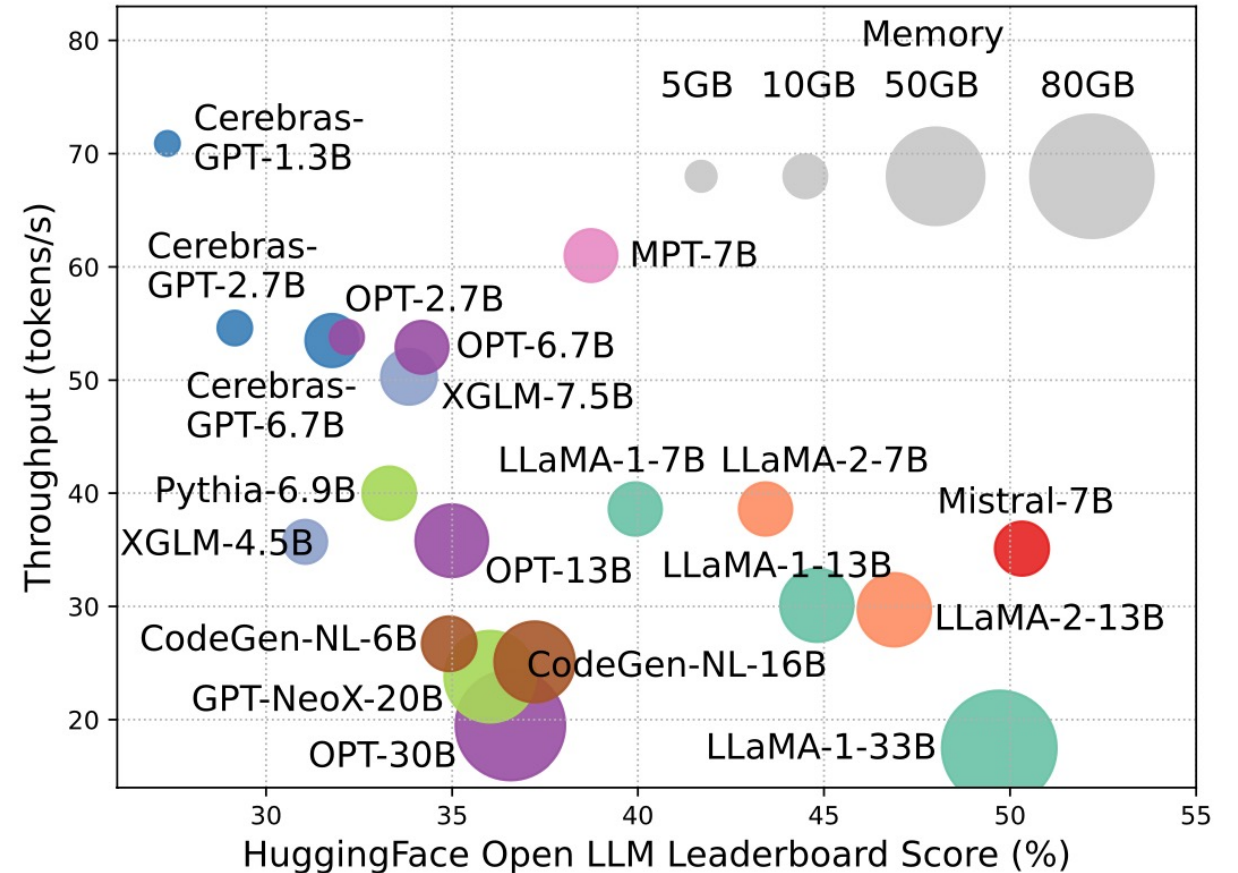
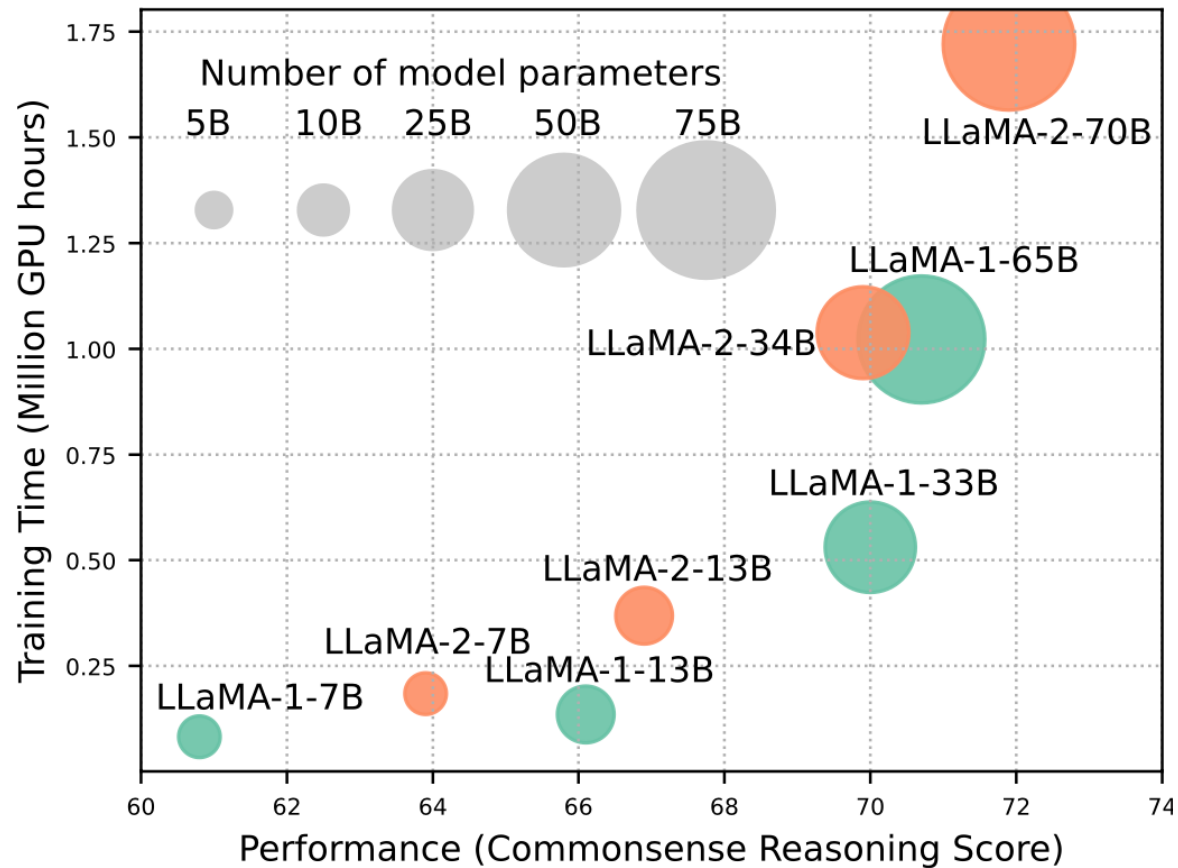
School of Computer Science



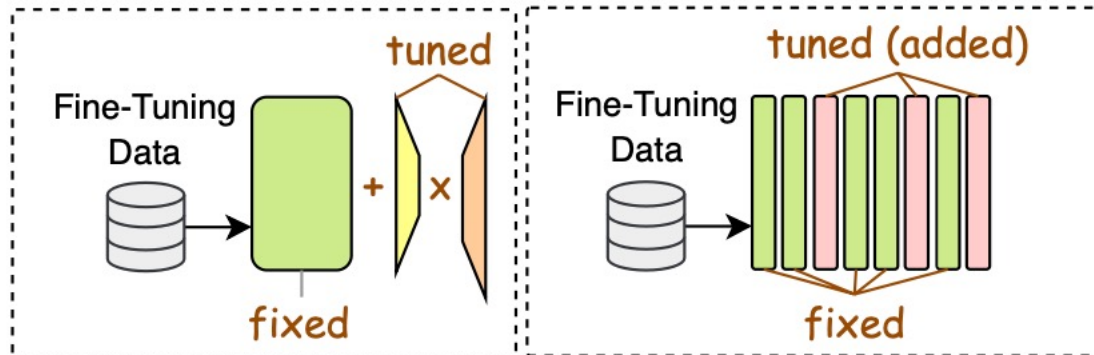
THE UNIVERSITY OF
SYDNEY



How important of efficiency ?

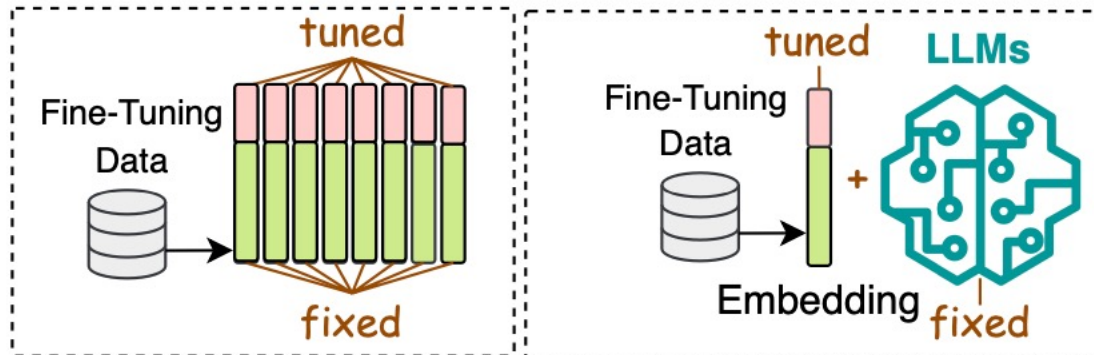


Overview of Efficiency in LLM



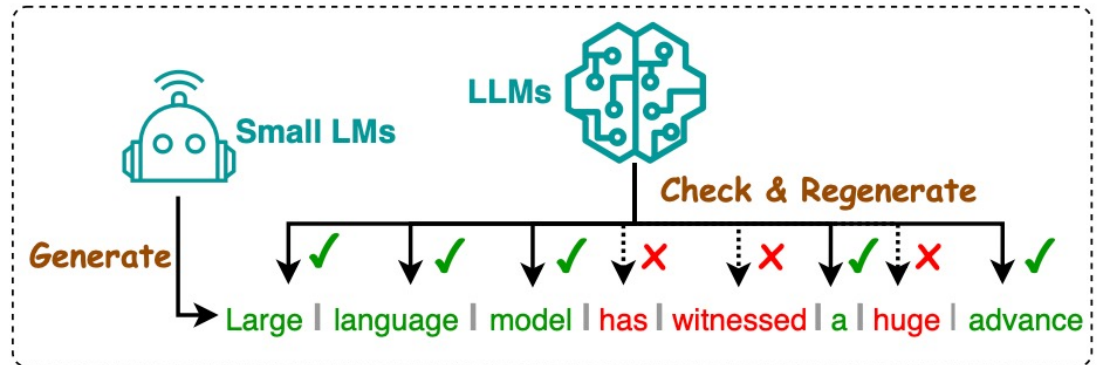
(a) Low-Rank Adaptation

(b) Adapter-based Tuning

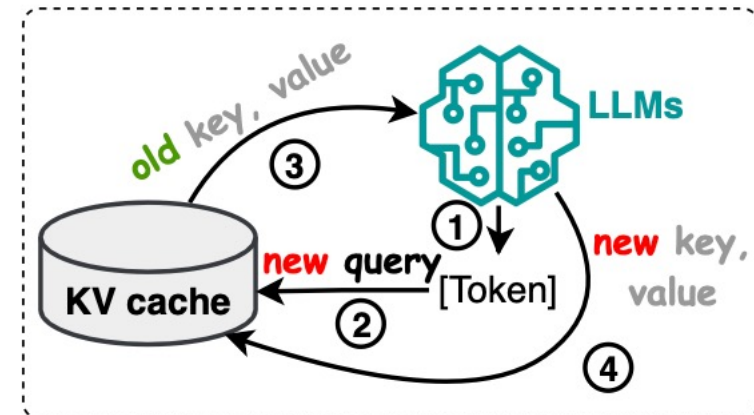


(c) Prefix Tuning

(d) Prompt Tuning



(a) Speculative Decoding

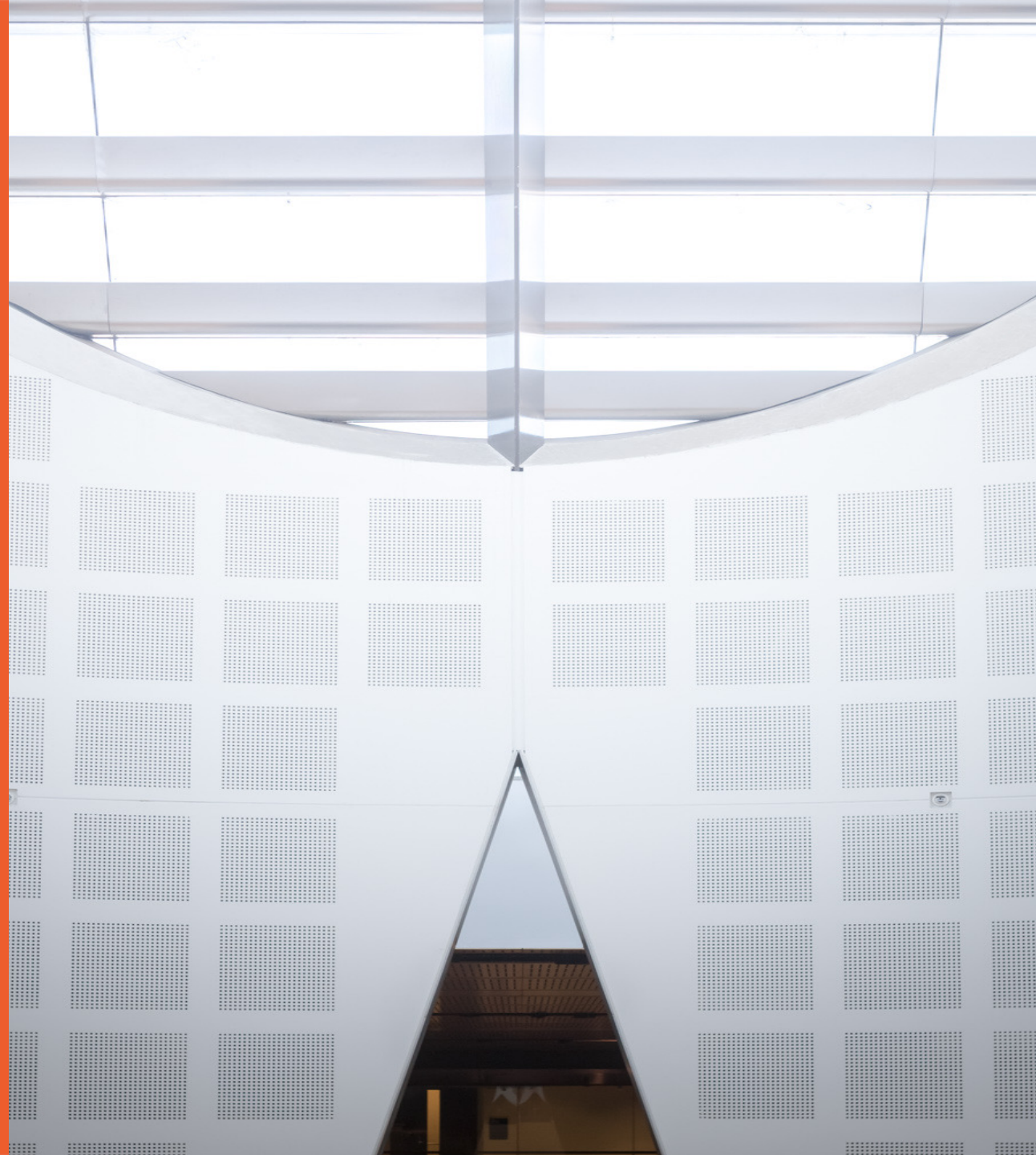


(b) KV-Cache Optimization

Seminar 2, Part 1.1: Efficiency of Fune-Tuning

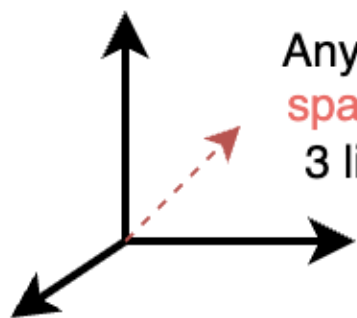


THE UNIVERSITY OF
SYDNEY

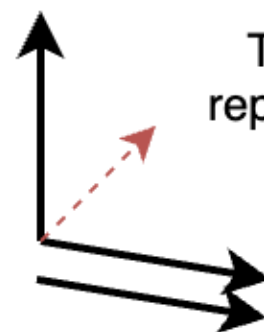


Low-Rank Adaption: LoRA

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix}, \quad \text{Rank}(\mathbf{A}) = 3. \quad \mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 2 & 4 & 6 \end{bmatrix}, \quad \text{Rank}(\mathbf{A}) = 2.$$



Any **vectors** in 3-dimensional space could be represent by 3 linear dependent vectors

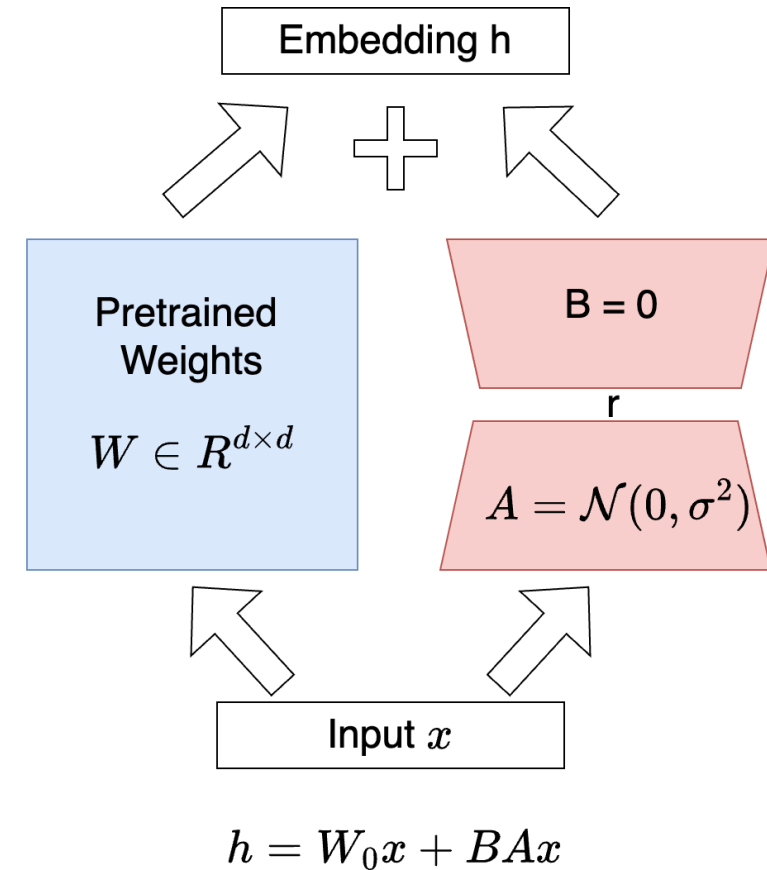
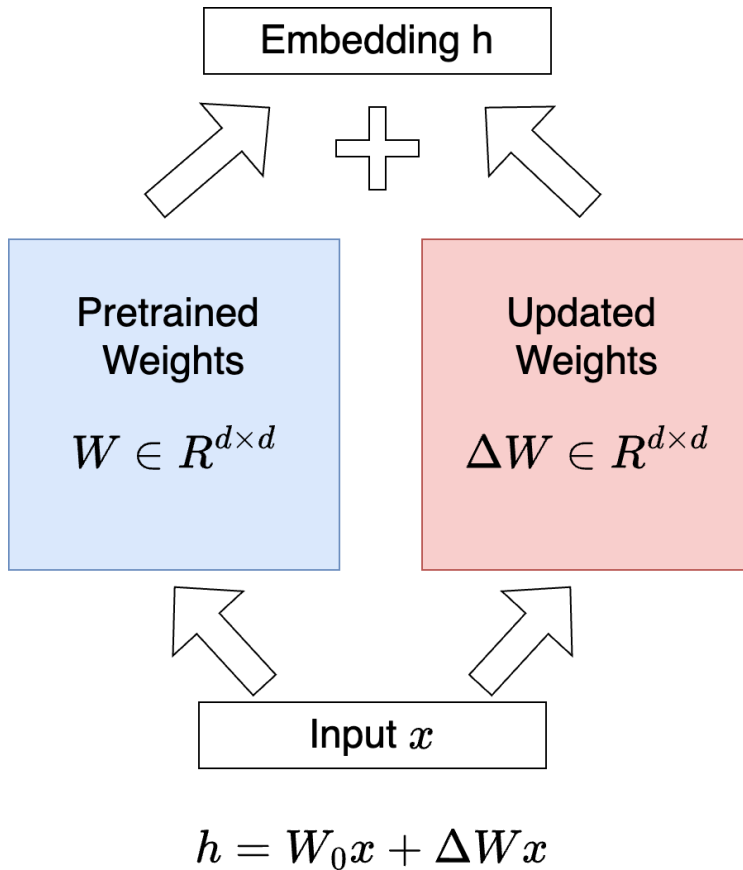


The **vector** could **not** be represented by only 2 linear dependent vectors

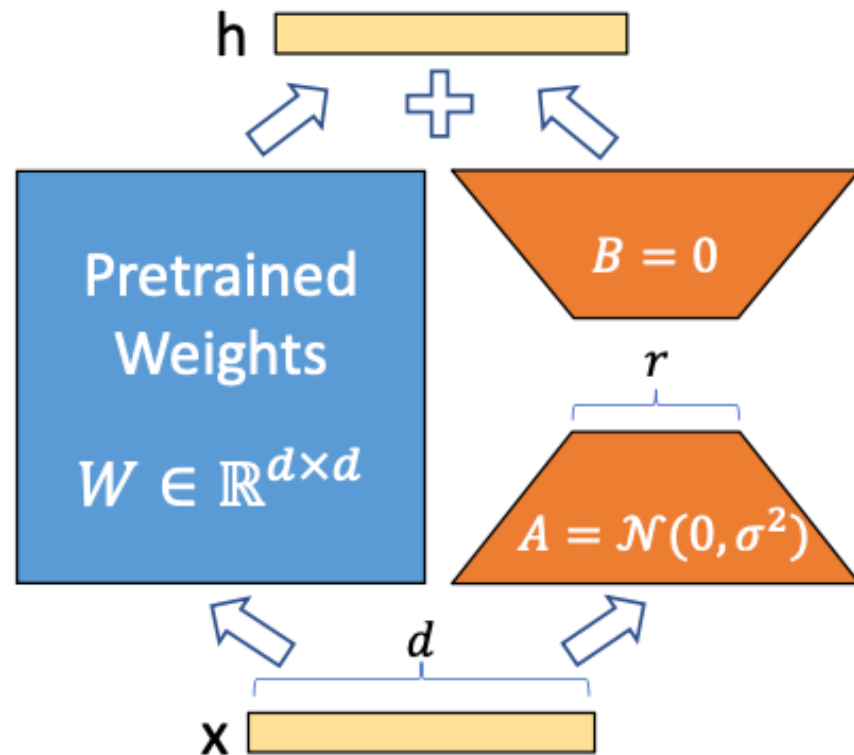
We hope the rank is large in weights as it can represent more dimentional latent features

The weights pretrained only need specfic base vectors for specific tasks

Low-Rank Adaption: LoRA



Low-Rank Adaption: LoRA



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

Why

$B = 0$

Tuning starts from original W

Why

$A = \mathcal{N}(0, \sigma^2)$

Gradient updating process is stable

Implement for each layer:

1) This is a simple example module where we want add LoRA modules to each Linear layer

```
class FeedForwardNetwork(torch.nn.Module):
    def __init__(self, embedding_dim):
        super().__init__()
        self.network = nn.Sequential(
            torch.nn.Linear(embedding_dim, 4*embedding_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(4*embedding_dim, embedding_dim),
            torch.nn.Dropout(0.2)
        )

    def forward(self, x):
        return self.network(x)
```

2) The original Linear layer is provided as input to the LinearWithLoRA layer

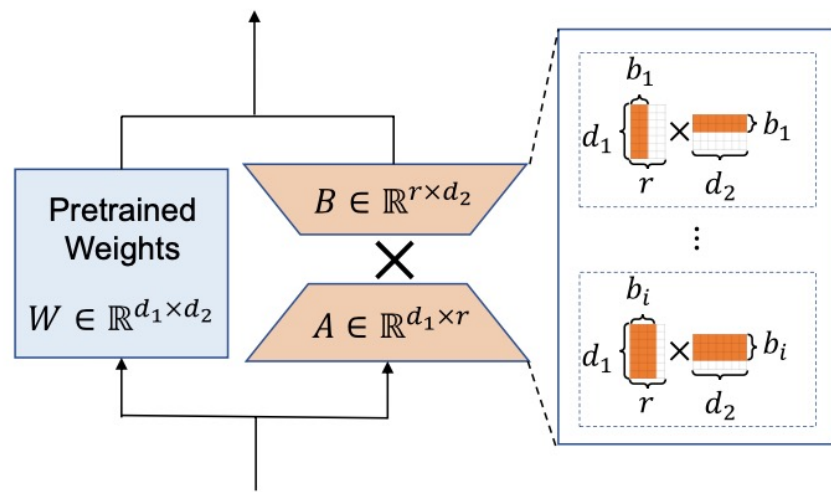
```
class LinearWithLoRA(torch.nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(
            linear.in_features, linear.out_features, rank, alpha
        )

    def forward(self, x):
        return self.linear(x) + self.lora(x)
```

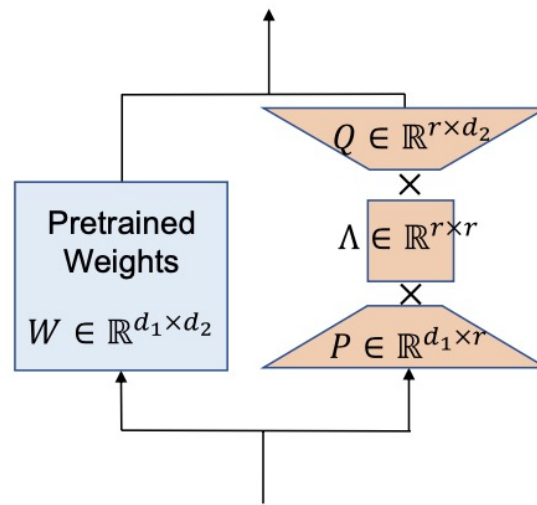
3) The LinearWithLoRA layer consists of the original Linear layer plus a LoRALayer containing the LoRA weight matrices

4) In the new LinearWithLoRA layer, we add the linear and LoRA layer outputs during the forward pass.

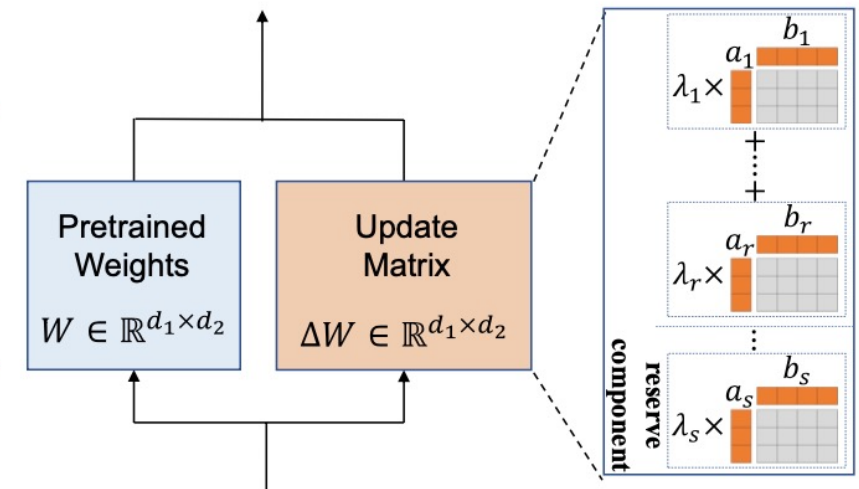
Other Variants



(a) DyLoRA



(b) AdaLoRA

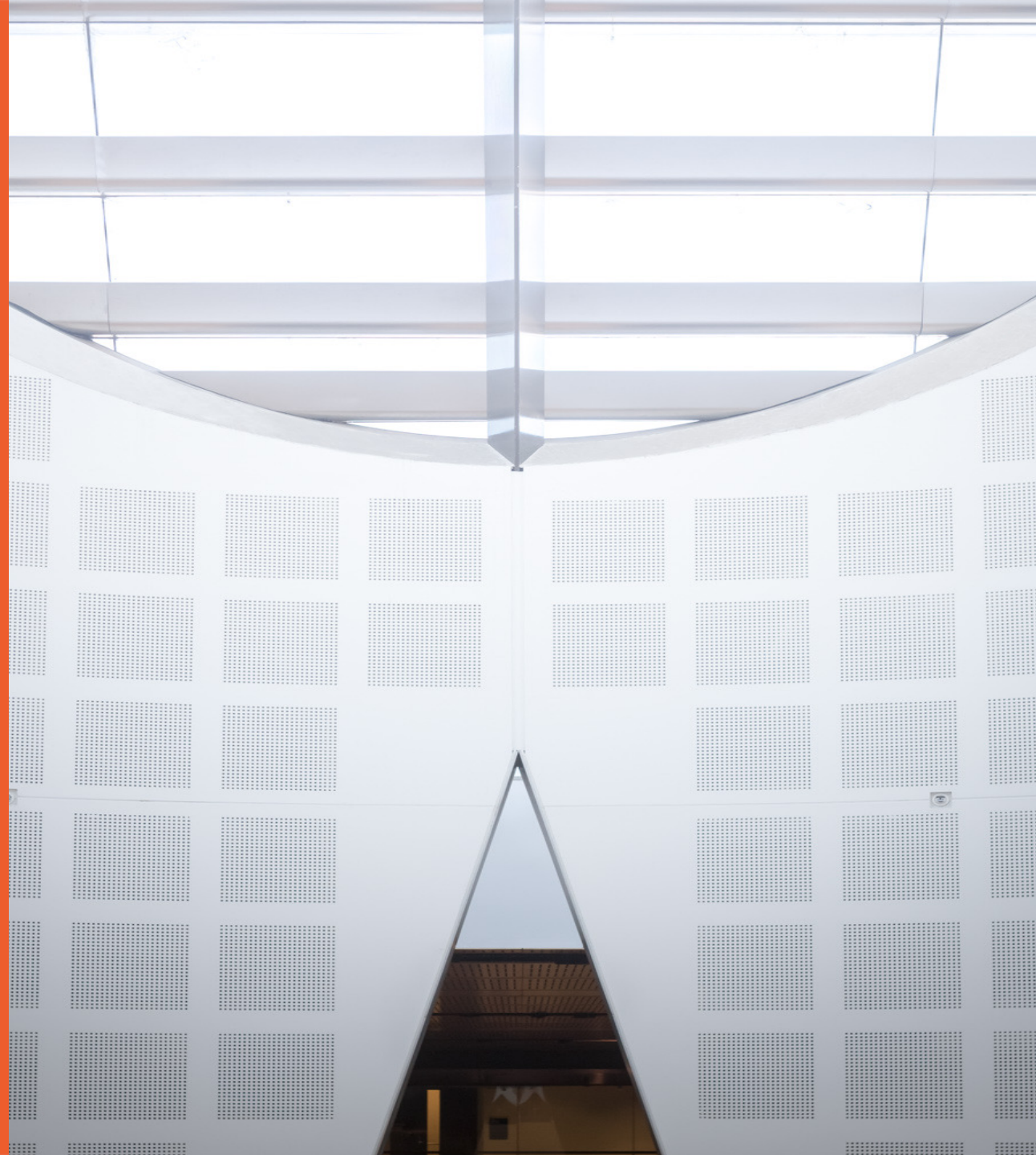


(c) IncreLoRA

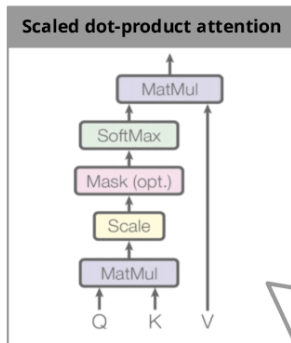
Seminar 1, Part 1.2: Efficiency of Inference



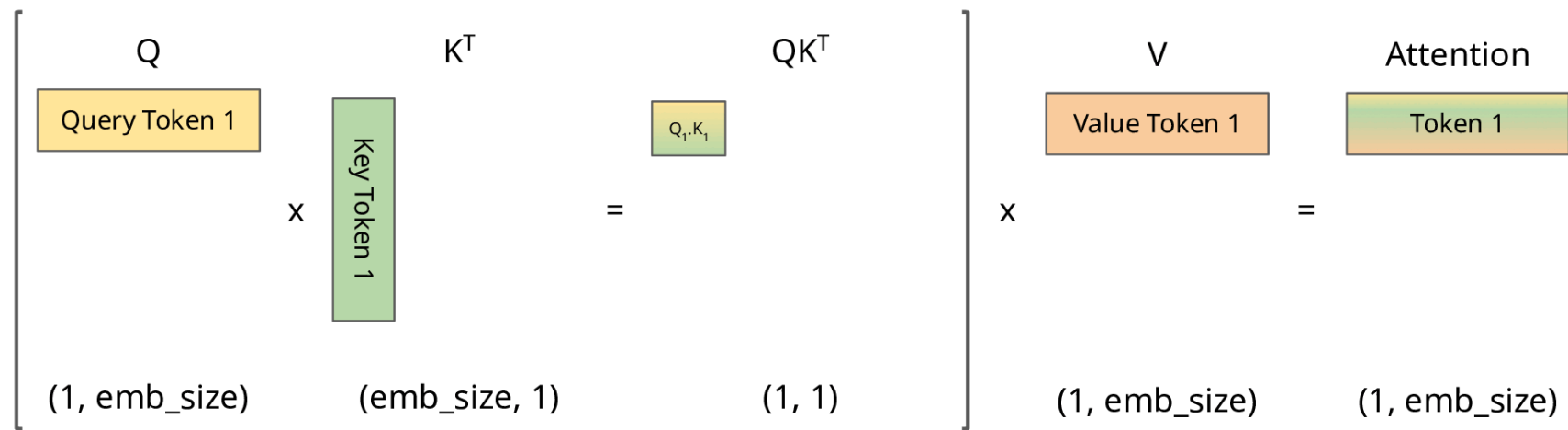
THE UNIVERSITY OF
SYDNEY



KV Cache Inference



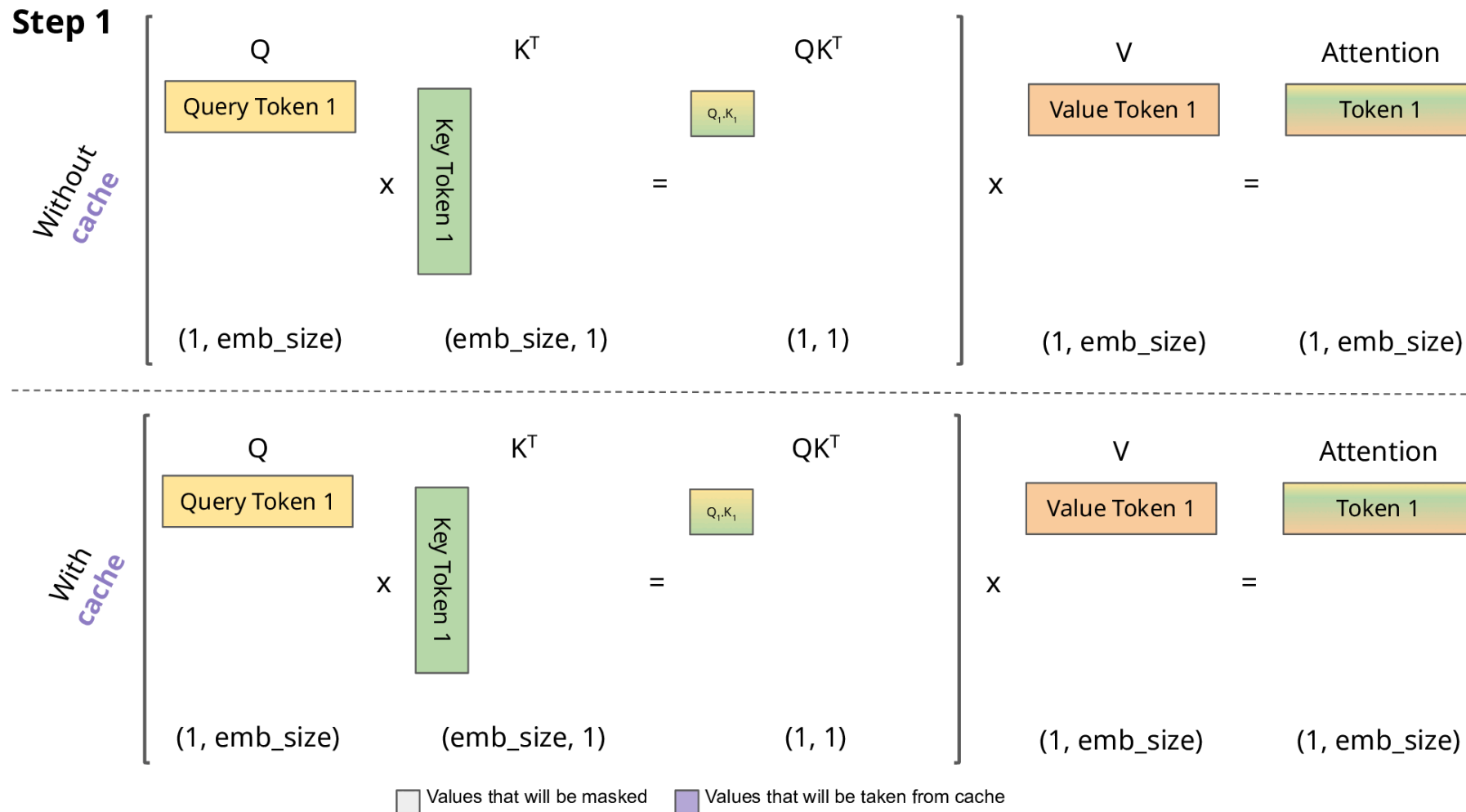
Step 1



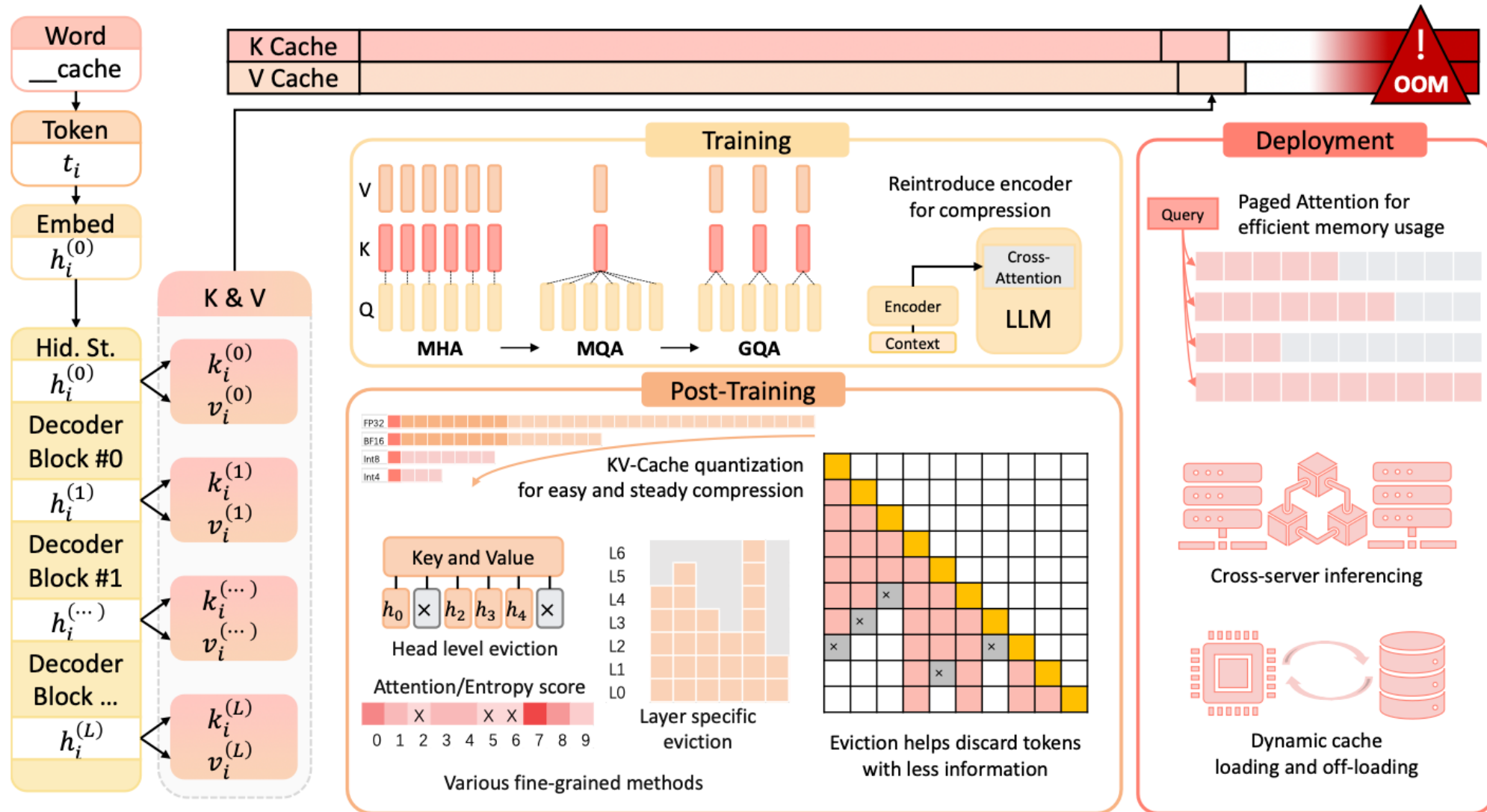
Values that will be masked

Zoom-in! (simplified without Scale and Softmax)

KV Cache Inference

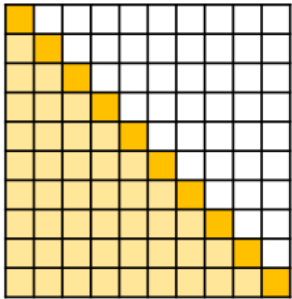


KV Cache Inference



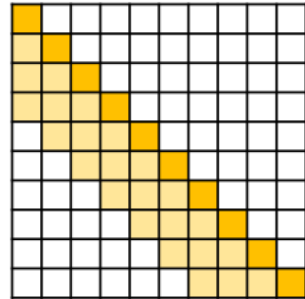
KV Cache Inference

Traditional Policies

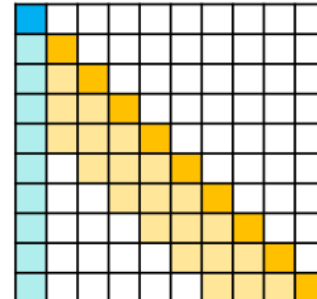


(a) Dense Attention

Static Policies

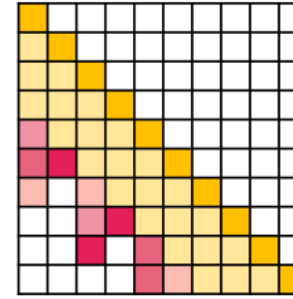


(b) Window Attention



(c) Initial Token Attention

Dynamic Policies

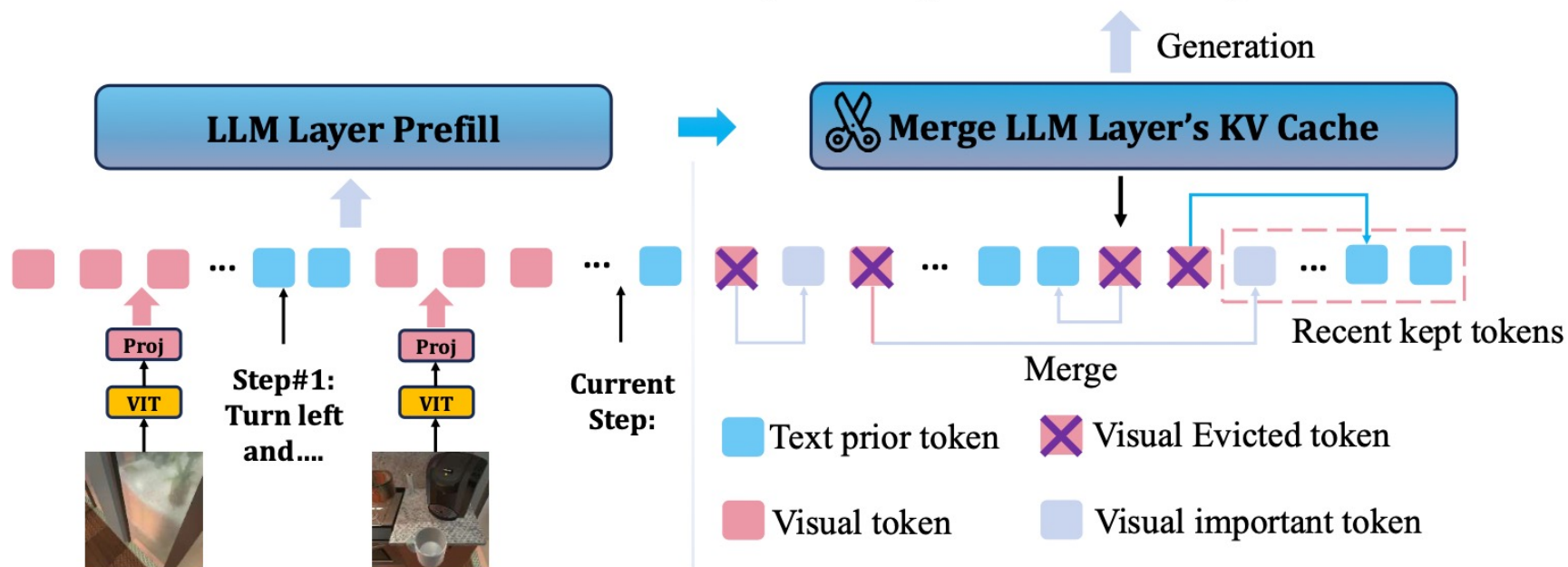


(e) Attention Based

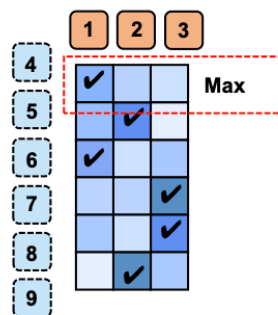


VLM Inference

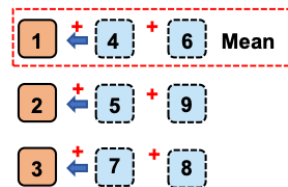
Pick up the mug that's in front of you at the coffee maker.



Conserved Token (orange square), Evicted Token (blue square), Max Similarity Value (blue checkmark)



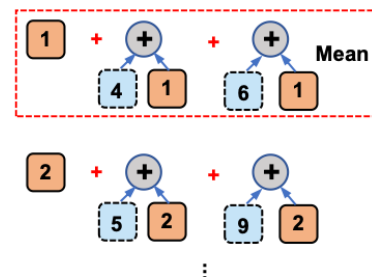
(a) Similarity Matrix



(b) Averaged Merging

K_c 1 2 3

K_e 4 5 6 7 8 9

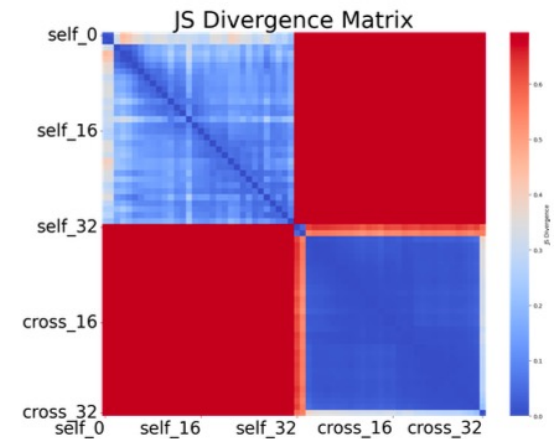
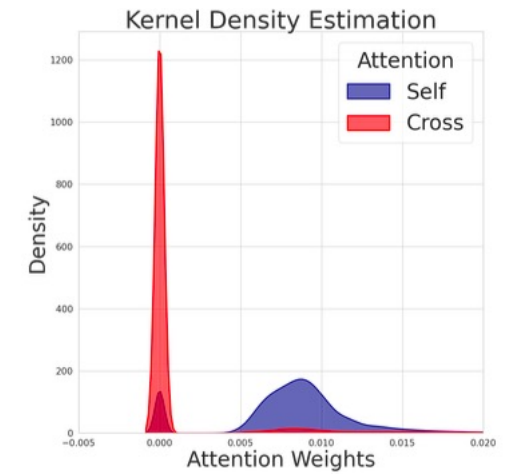
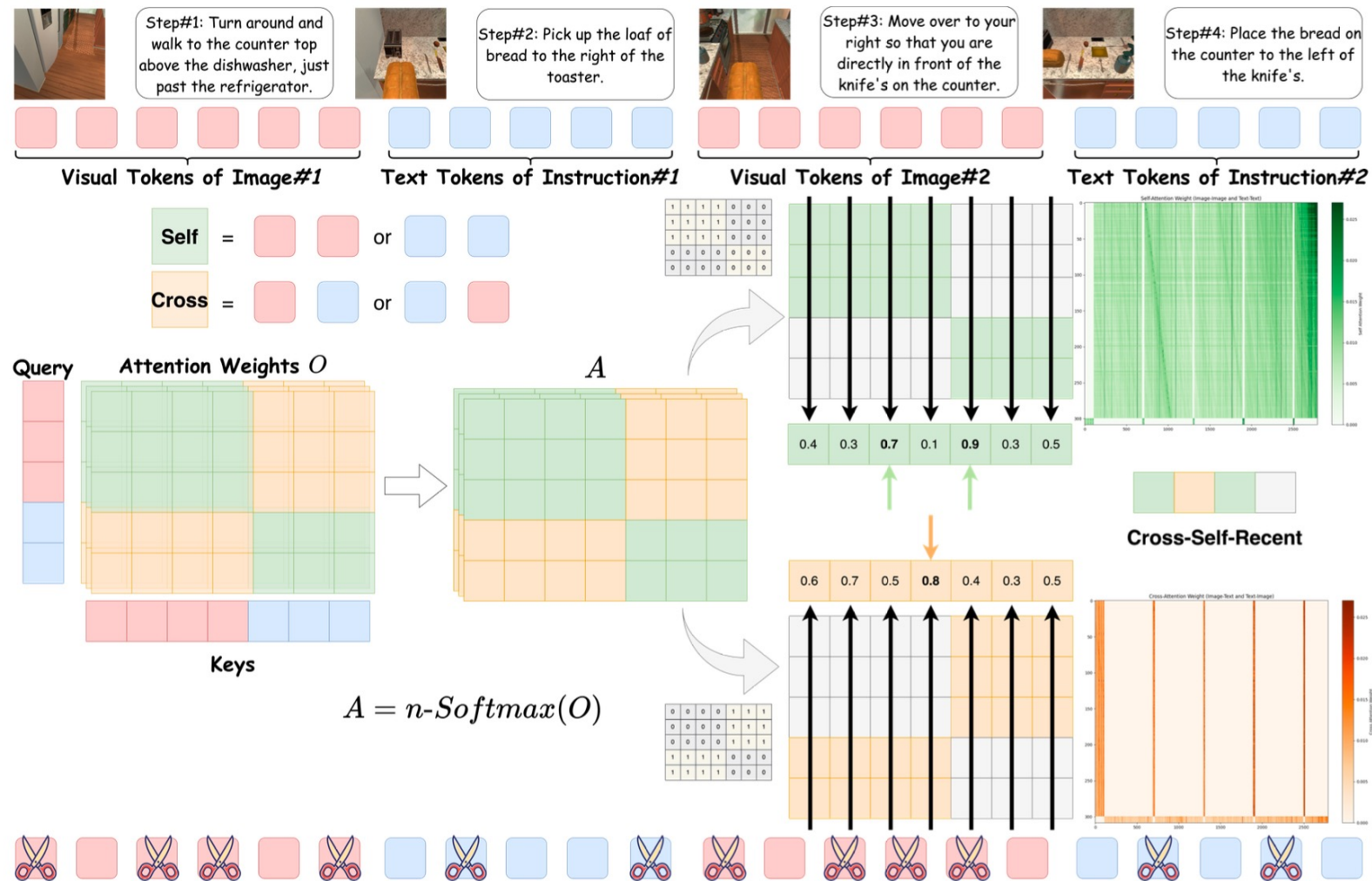


(c) Pivotal Merging



(d) Weighted Merging

VLM Inference



How about another approach ?

Thank you!
Q&A