



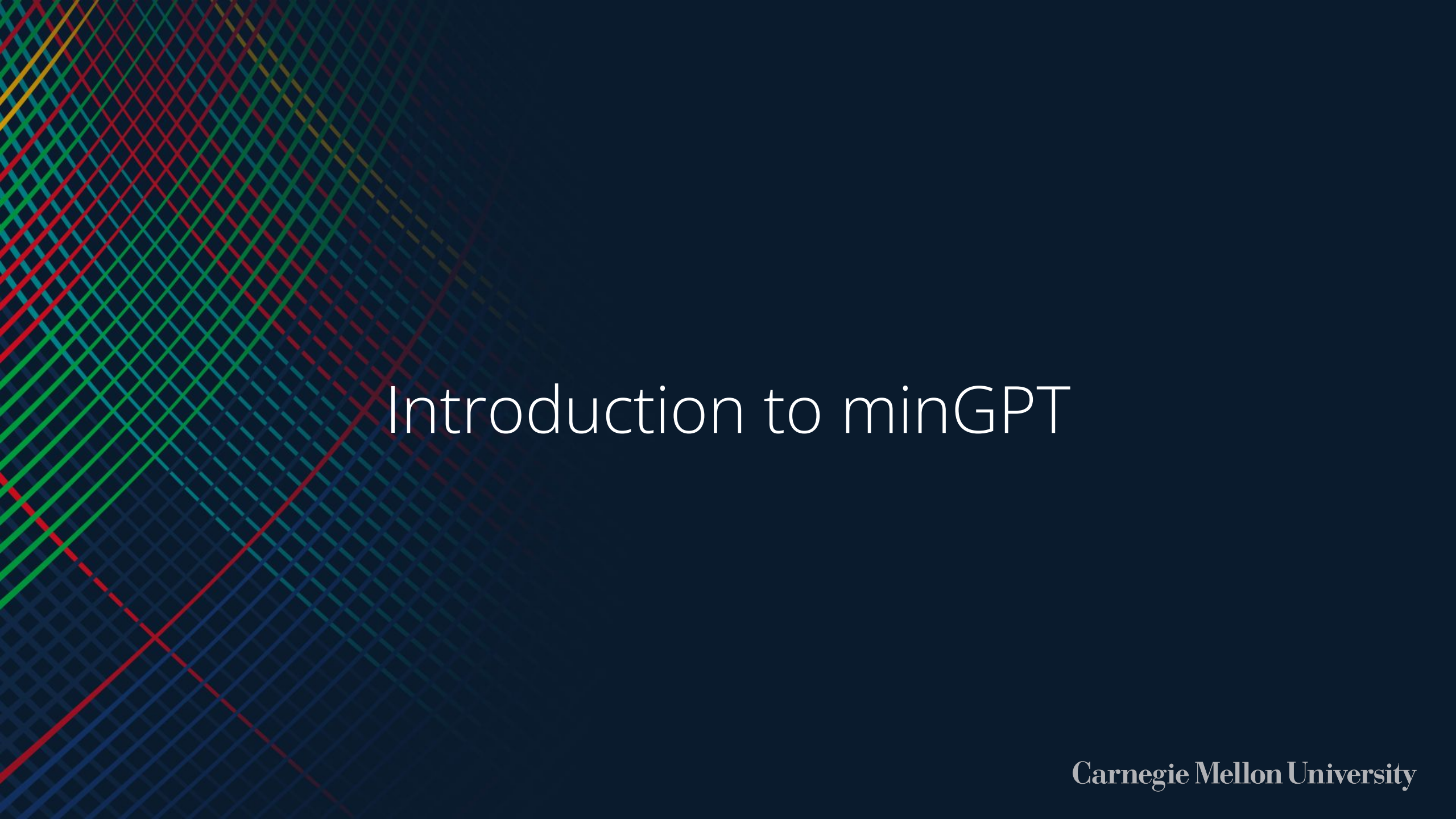
Recitation: HW1

10-423/10-623 Generative AI

September 12, 2025

Agenda

- 1.** Introduction to minGPT
- 2.** RoPE (Rotary Positional Embeddings)
- 3.** GQA (Grouped Query Attention)
- 4.** Einops
- 5.** Sliding Window Attention



Introduction to minGPT

minGPT - your very own transformer

- Thanks to Andrej Karpathy!
- Fully functioning transformer implementation, modified for this course
- Training data: the complete works of Shakespeare
- [Github repo link](#)

available GPT implementations



minGPT



You'll be upgrading minGPT with the latest research

- ✓ Rotary Positional Embeddings
- ✓ Grouped Query Attention
- ✓ Sliding window attention (only written, no programming)

available GPT implementations

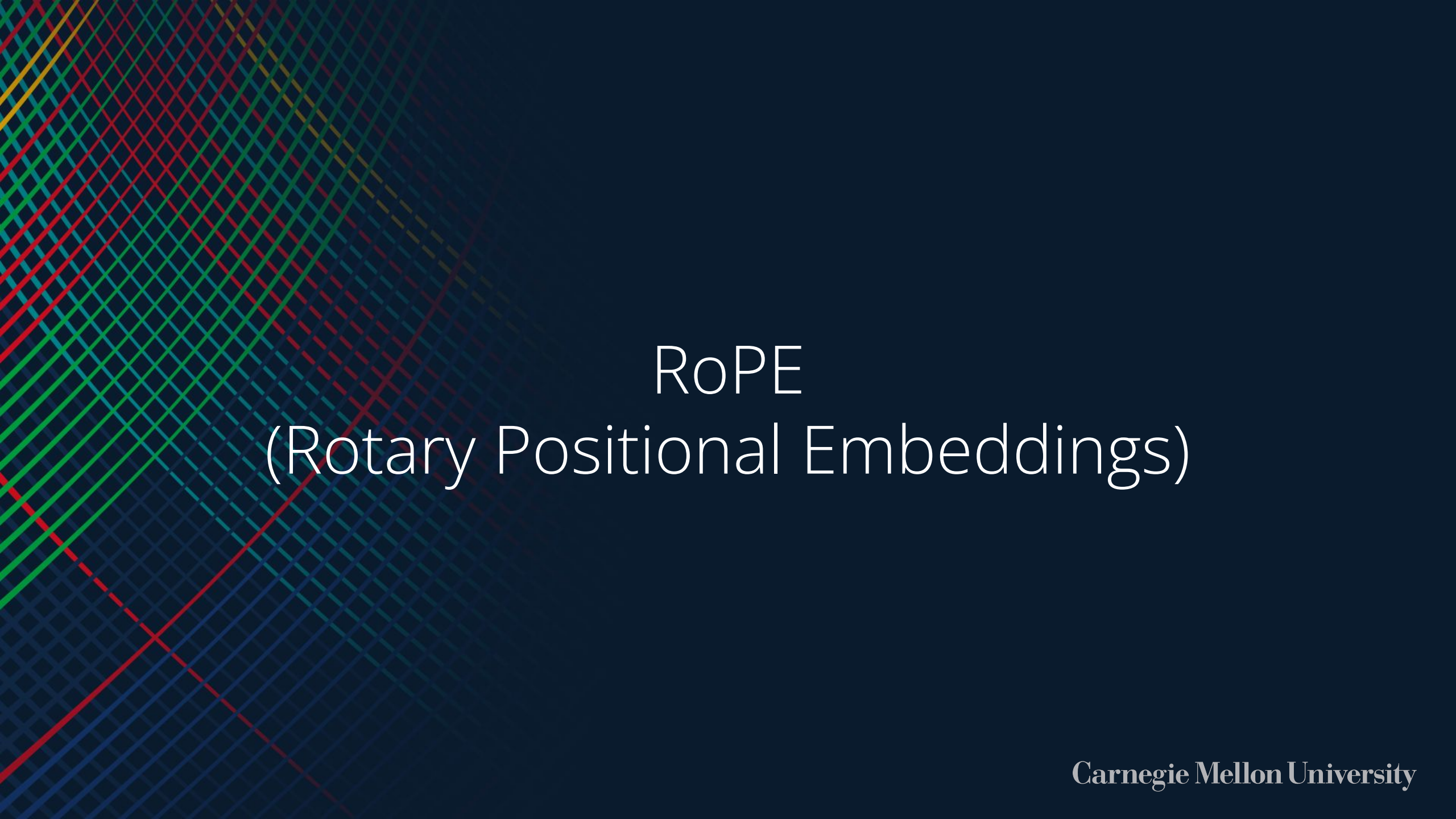


minGPT



YourGPT





RoPE

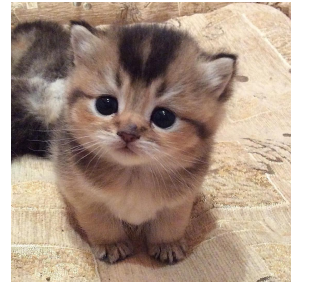
(Rotary Positional Embeddings)

Rotary Positional Embeddings (RoPE)

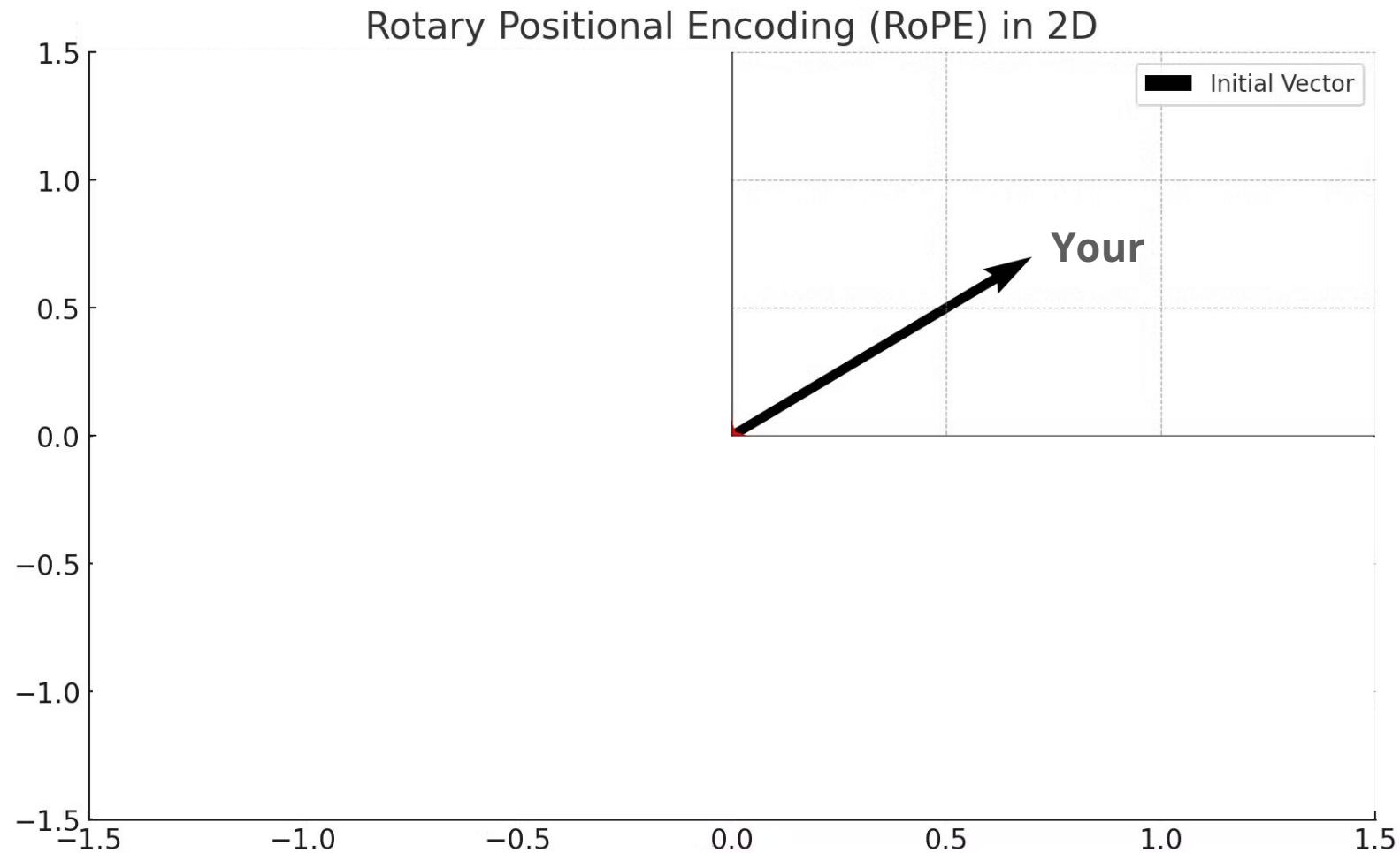
Sentence: **Your cat is a lovely cat**



Rotary Positional Embeddings (RoPE)



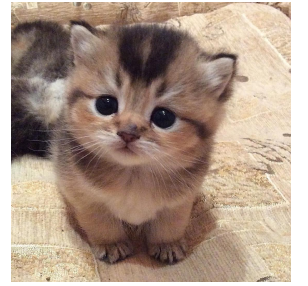
Sentence: **Your cat is a lovely cat**



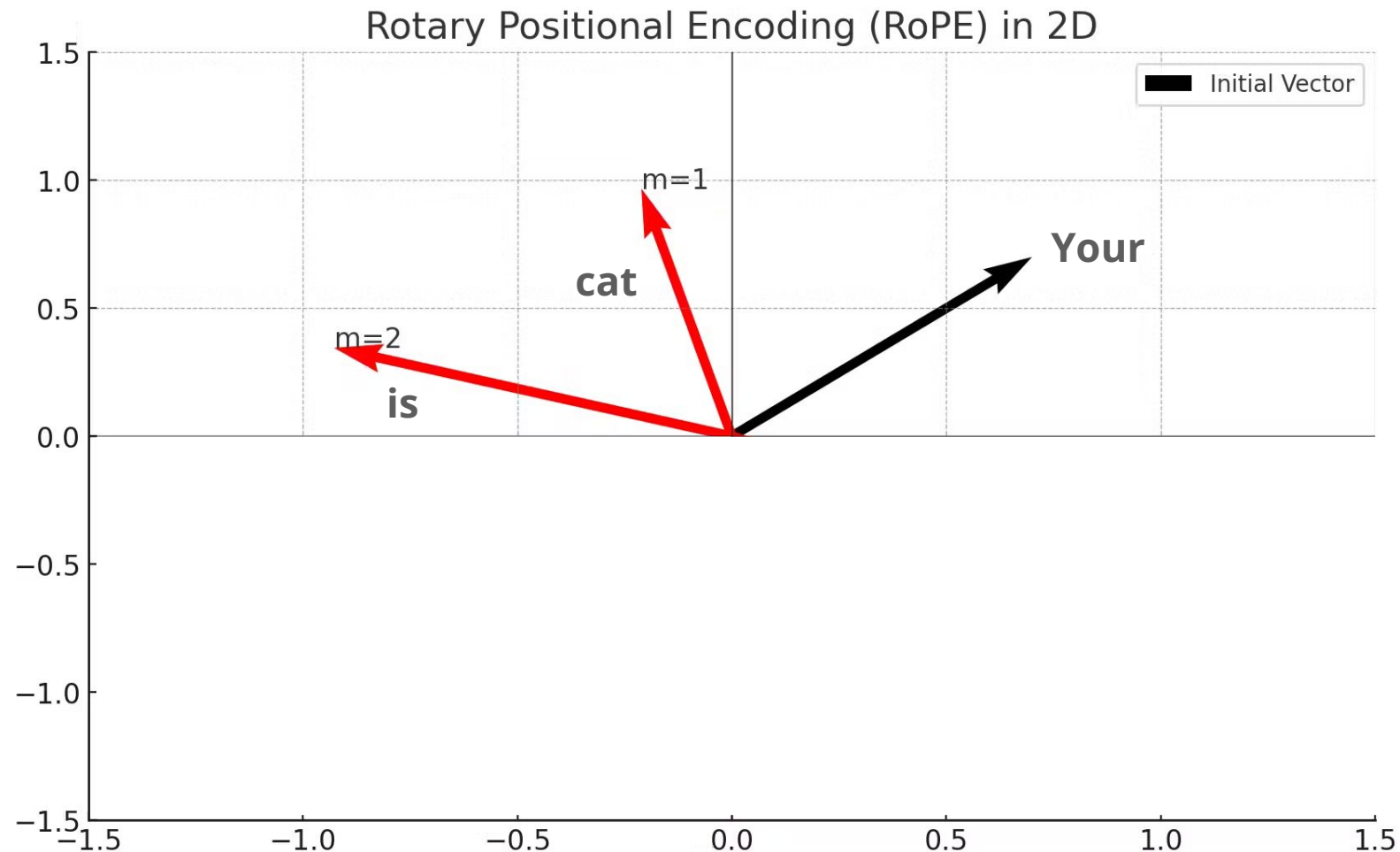
Your



Rotary Positional Embeddings (RoPE)



Sentence: **Your cat is a lovely cat**



Your



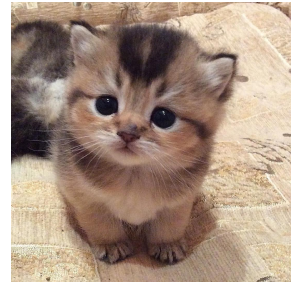
cat



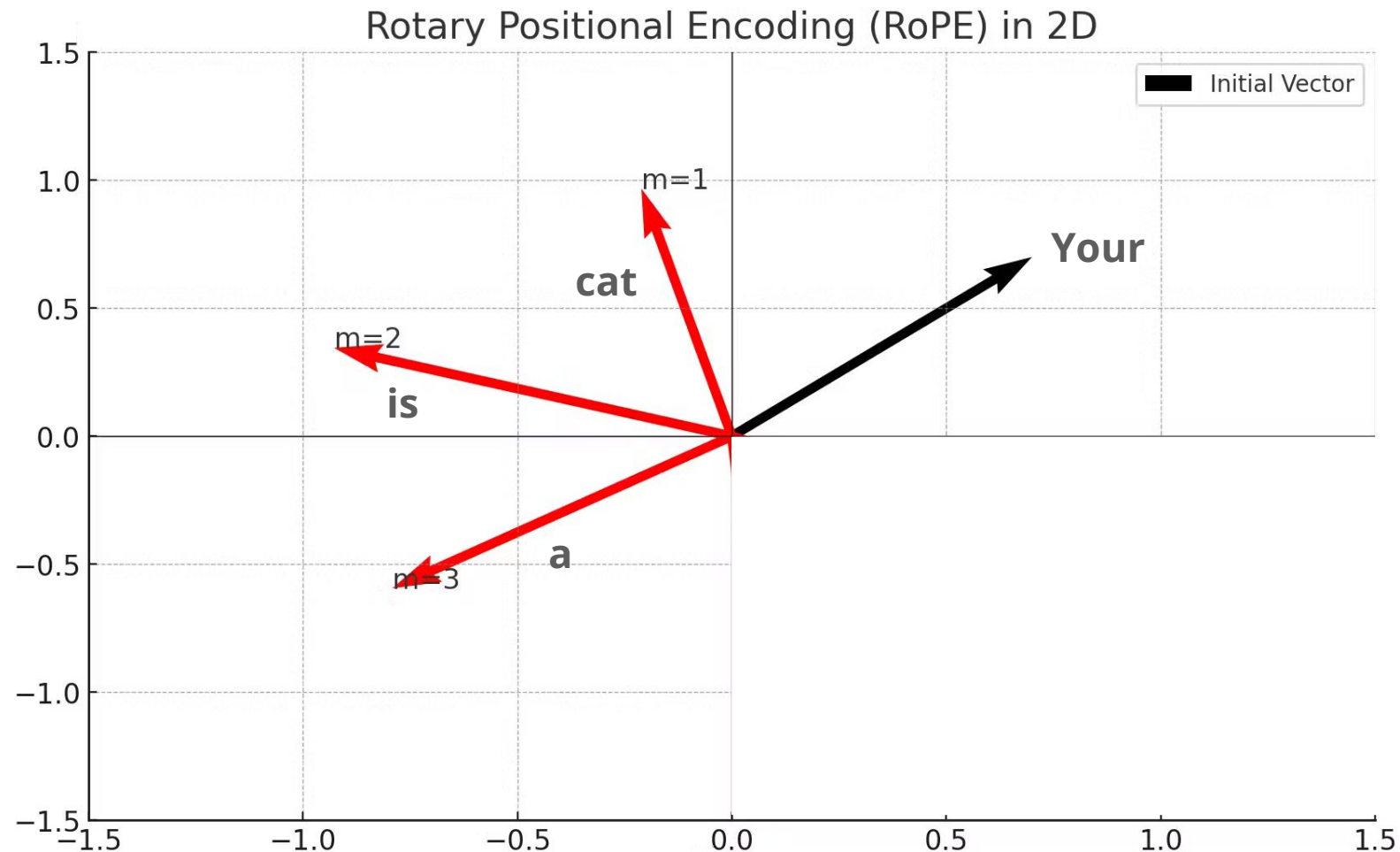
is



Rotary Positional Embeddings (RoPE)



Sentence: **Your cat is a lovely cat**



Your



cat



is



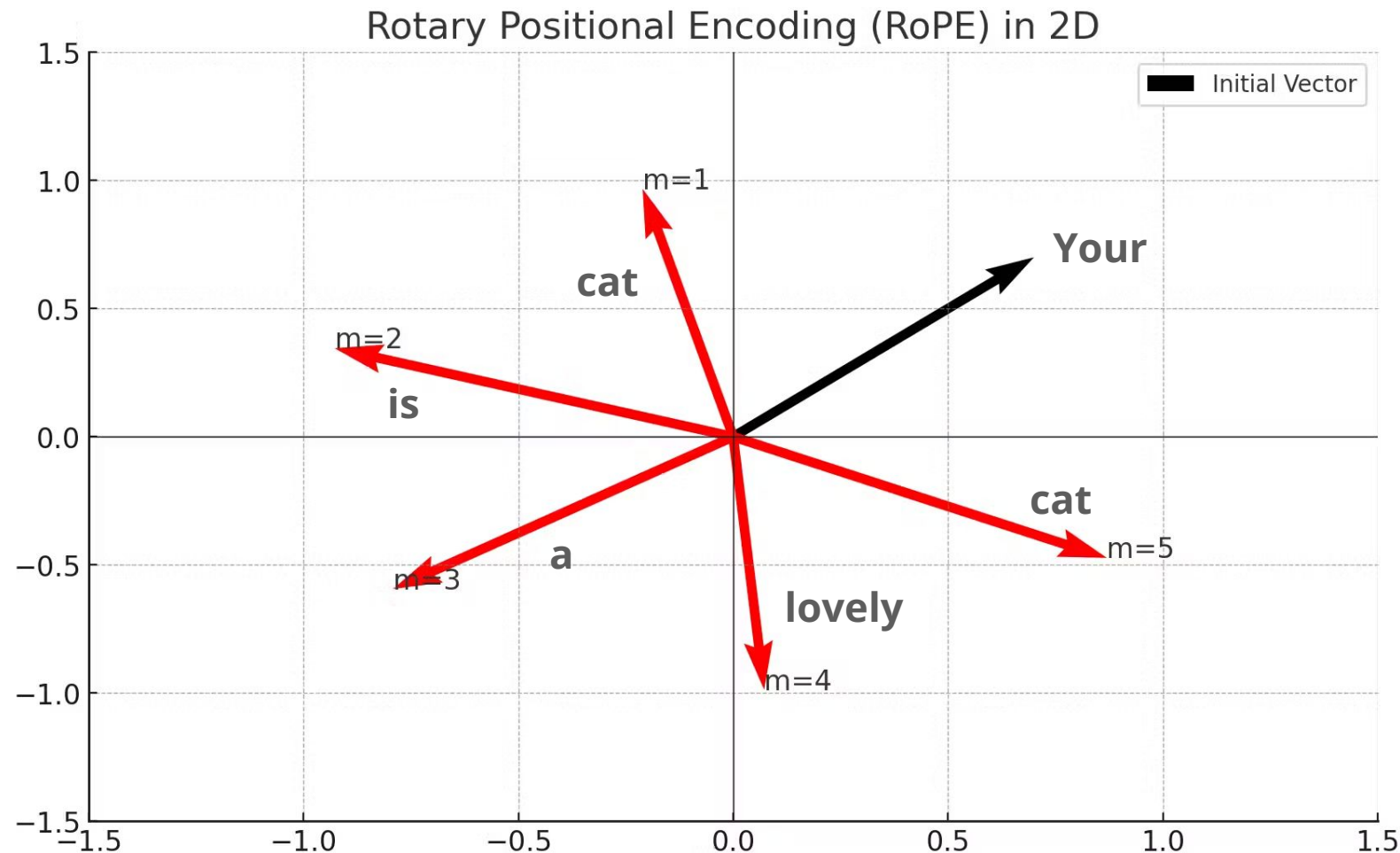
a



Rotary Positional Embeddings (RoPE)



Sentence: **Your cat is a lovely cat**



Your



cat



is



a



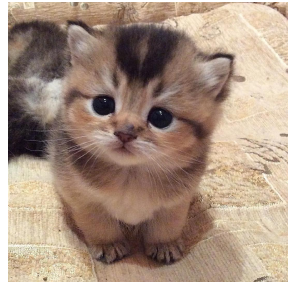
lovely



cat

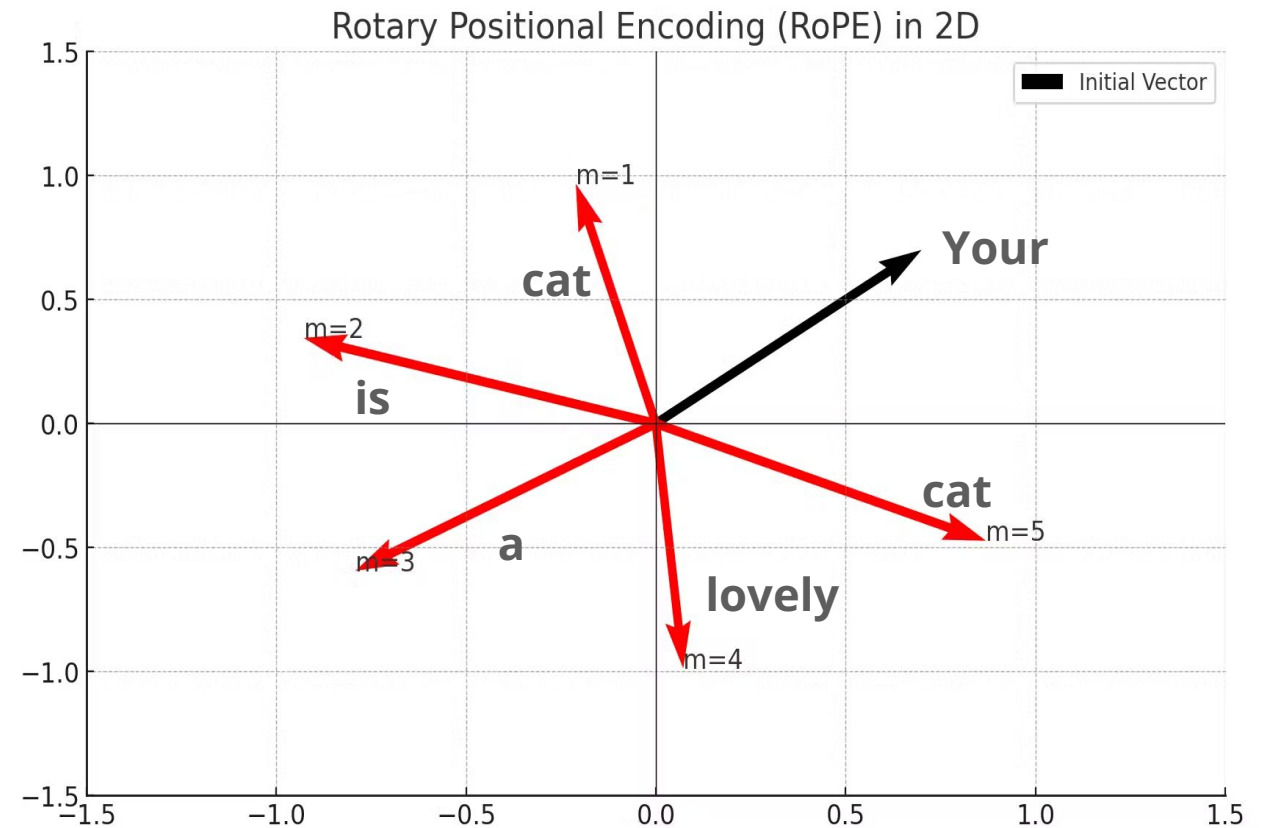


Rotary Positional Embeddings (RoPE)



Encodes **both** the absolute and relative positional information

- **absolute position**
 - each token receives a unique encoding based on its position
- **relative position**
 - distance between any 2 tokens should be accurately represented
 - ex. positions 1 & 2 should be more closely related than positions 2 & 5



Recall: Rotation Matrices

In $\mathbb{R}^2$, consider the matrix that rotates a given vector $\mathbf{v}_0$ by a counterclockwise angle θ in a fixed coordinate system.

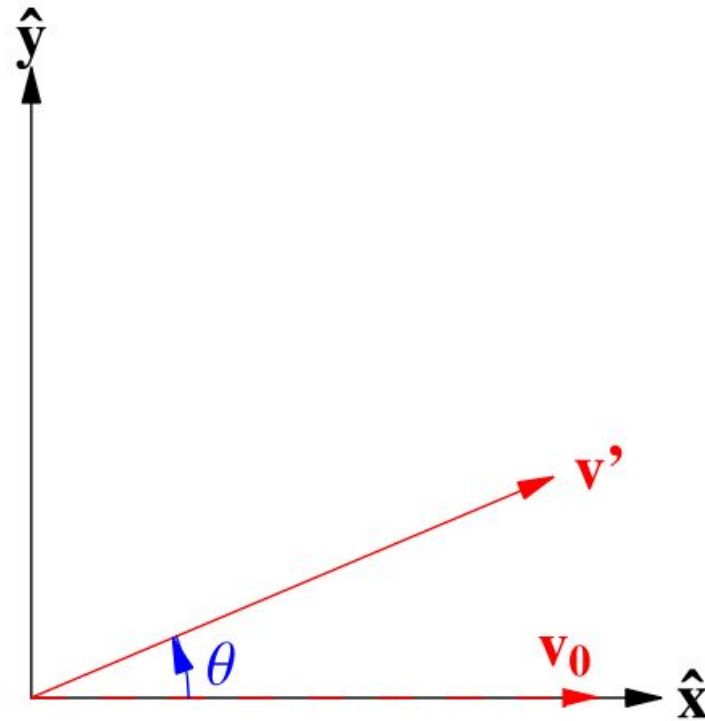
$$\mathbf{R}_\theta = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix},$$

so

$$\mathbf{v}' = \mathbf{R}_\theta \mathbf{v}_0.$$

Diagram illustrating the components of the equation $\mathbf{v}' = \mathbf{R}_\theta \mathbf{v}_0$:

- $\mathbf{v}'$: vector rotated θ degrees
- $\mathbf{R}_\theta$: Rotation matrix
- $\mathbf{v}_0$: original vector



RoPE in Attention

Standard attention:

$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j, \forall j$$

$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j, \forall j$$

$$s_{t,j} = \mathbf{k}_j^T \mathbf{q}_t / \sqrt{|\mathbf{k}|}, \forall j, t$$

$$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t), \forall t$$

RoPE attention:

$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j, \forall j$$

$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j, \forall j$$

$$\tilde{\mathbf{q}}_j = \mathbf{R}_{\Theta,j} \mathbf{q}_j$$

$$\tilde{\mathbf{k}}_j = \mathbf{R}_{\Theta,j} \mathbf{k}_j$$

$$s_{t,j} = \tilde{\mathbf{k}}_j^T \tilde{\mathbf{q}}_t / \sqrt{d_k}, \forall j, t$$

$$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t), \forall t$$

$$\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, \dots, d/2]\}$$

where d=embedding dimension

RoPE beyond 2D (What happens if $d > 2$?)

2D rotary matrix ($d = 2$):

$$R_{\Theta, m}^d = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix}$$

$$\begin{bmatrix} q_1^{(1)} & q_1^{(2)} \end{bmatrix}$$

if query represented by
embedding of size $d=2$

4D rotary matrix ($d = 4$):

$$R_{\Theta, m}^d = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 \end{pmatrix}$$

Block 1

Block 2

$$\begin{bmatrix} q_1^{(1)} & q_1^{(2)} & q_1^{(3)} & q_1^{(4)} \end{bmatrix}$$

if query represented by
embedding of size $d=4$

Rotary Matrix: General Form

d-dimensional rotary matrix:

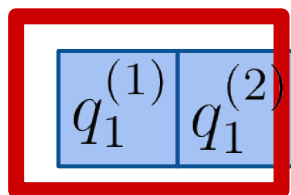
$$\mathbf{R}_{\Theta, m}^d = \begin{pmatrix} \boxed{\cos m\theta_1} & \boxed{-\sin m\theta_1} & 0 & 0 & \cdots & 0 & 0 \\ \boxed{\sin m\theta_1} & \boxed{\cos m\theta_1} & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \boxed{\cos m\theta_2} & \boxed{-\sin m\theta_2} & \cdots & 0 & 0 \\ 0 & 0 & \boxed{\sin m\theta_2} & \boxed{\cos m\theta_2} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \boxed{\cos m\theta_{d/2}} & \boxed{-\sin m\theta_{d/2}} \\ 0 & 0 & 0 & 0 & \cdots & \boxed{\sin m\theta_{d/2}} & \boxed{\cos m\theta_{d/2}} \end{pmatrix}$$

Block 1

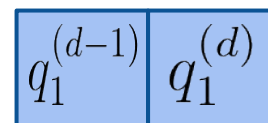
Block 2

Block d/2

if query represented by
embedding of size d



...



rotate by:

$m\theta_1$

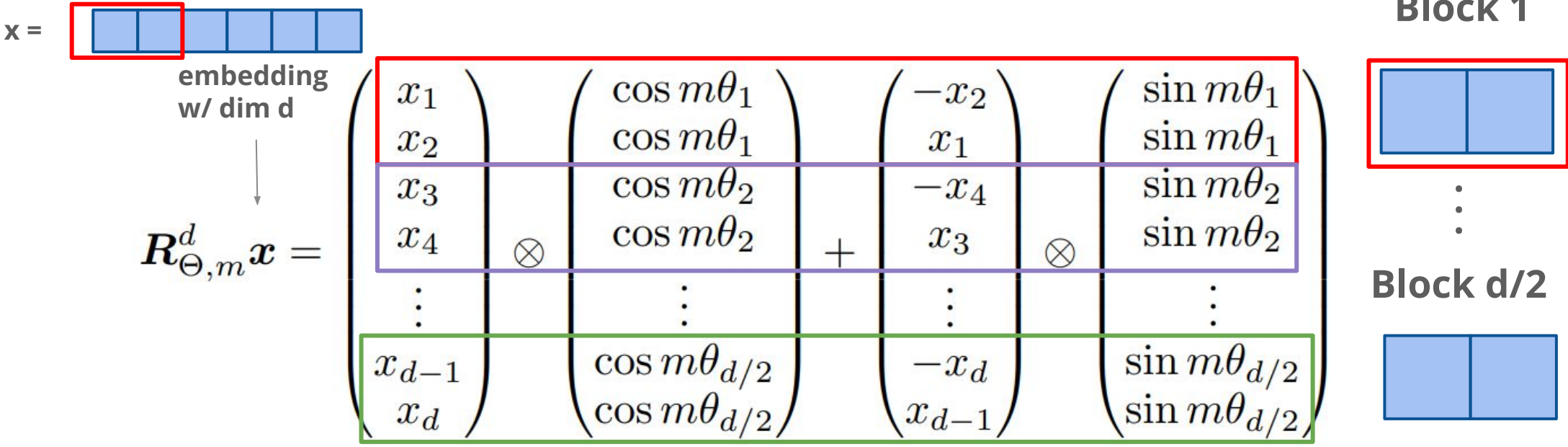
$m\theta_2$

$m\theta_{d/2}$

Rotary Matrix: Computationally Efficient Form

Since the matrix is sparse, it is not computationally efficient to compute.

$$R_{\Theta, m}^d = \begin{pmatrix} \boxed{\cos m\theta_1} & \boxed{-\sin m\theta_1} & 0 & 0 & \cdots & 0 & 0 \\ \boxed{\sin m\theta_1} & \boxed{\cos m\theta_1} & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \boxed{\cos m\theta_2} & \boxed{-\sin m\theta_2} & \cdots & 0 & 0 \\ 0 & 0 & \boxed{\sin m\theta_2} & \boxed{\cos m\theta_2} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \boxed{\cos m\theta_{d/2}} & \boxed{-\sin m\theta_{d/2}} \\ 0 & 0 & 0 & 0 & \cdots & \boxed{\sin m\theta_{d/2}} & \boxed{\cos m\theta_{d/2}} \end{pmatrix}$$



Matrix Version of RoPE

Given matrix $\mathbf{Y}$ (which we want to apply RoPE to):

$\mathbf{Y} =$

		embedding dim			
		1	2	...	d
sequence length	1	Your	cat	is	a
	2	cat	is	a	lovely
	...				
	N	cat			

$$\tilde{\mathbf{Y}} = g(\mathbf{Y}; \Theta)$$

$$= \left[\begin{array}{ccc|ccc} Y_{1,1} & \cdots & Y_{1,\frac{d}{2}} & Y_{1,\frac{d}{2}+1} & \cdots & Y_{1,d} \\ \vdots & & \vdots & \vdots & & \vdots \\ Y_{N,1} & \cdots & Y_{N,\frac{d}{2}} & Y_{N,\frac{d}{2}+1} & \cdots & Y_{N,d} \end{array} \right] \odot \left[\begin{array}{ccc|ccc} \cos 1\theta_1 & \cdots & \cos 1\theta_{\frac{d}{2}} & \cos 1\theta_1 & \cdots & \cos 1\theta_{\frac{d}{2}} \\ \vdots & & \vdots & \vdots & & \vdots \\ \cos N\theta_1 & \cdots & \cos N\theta_{\frac{d}{2}} & \cos N\theta_1 & \cdots & \cos N\theta_{\frac{d}{2}} \end{array} \right]$$

$$+ \left[\begin{array}{ccc|ccc} -Y_{1,\frac{d}{2}+1} & \cdots & -Y_{1,d} & Y_{1,1} & \cdots & Y_{1,\frac{d}{2}} \\ \vdots & & \vdots & \vdots & & \vdots \\ -Y_{N,\frac{d}{2}+1} & \cdots & -Y_{N,d} & Y_{N,1} & \cdots & Y_{N,\frac{d}{2}} \end{array} \right] \odot \left[\begin{array}{ccc|ccc} \sin 1\theta_1 & \cdots & \sin 1\theta_{\frac{d}{2}} & \sin 1\theta_1 & \cdots & \sin 1\theta_{\frac{d}{2}} \\ \vdots & & \vdots & \vdots & & \vdots \\ \sin N\theta_1 & \cdots & \sin N\theta_{\frac{d}{2}} & \sin N\theta_1 & \cdots & \sin N\theta_{\frac{d}{2}} \end{array} \right]$$

(equivalently, and more compactly)

Use this to implement for Hw1!

$$\mathbf{C} = \left[\begin{array}{ccc|ccc} 1\theta_1 & \cdots & 1\theta_{\frac{d}{2}} & 1\theta_1 & \cdots & 1\theta_{\frac{d}{2}} \\ \vdots & & \vdots & \vdots & & \vdots \\ N\theta_1 & \cdots & N\theta_{\frac{d}{2}} & N\theta_1 & \cdots & N\theta_{\frac{d}{2}} \end{array} \right]$$

$$\tilde{\mathbf{Y}} = g(\mathbf{Y}; \Theta)$$

$$\begin{aligned} &= \left[\mathbf{Y}_{\cdot, 1:d/2} \mid \mathbf{Y}_{\cdot, d/2+1:d} \right] \odot \cos(\mathbf{C}) \\ &\quad + \left[-\mathbf{Y}_{\cdot, d/2+1:d} \mid \mathbf{Y}_{\cdot, 1:d/2} \right] \odot \sin(\mathbf{C}) \end{aligned}$$

Q: In our minGPT, what is Y? i.e. what do we apply RoPE to?

A: The keys and queries (k and q)

What RoPE Promises

Used in models including: Llama 3.1, Falcon, PaLM

1. Faster convergence of training and validation curves, lower overall validation loss

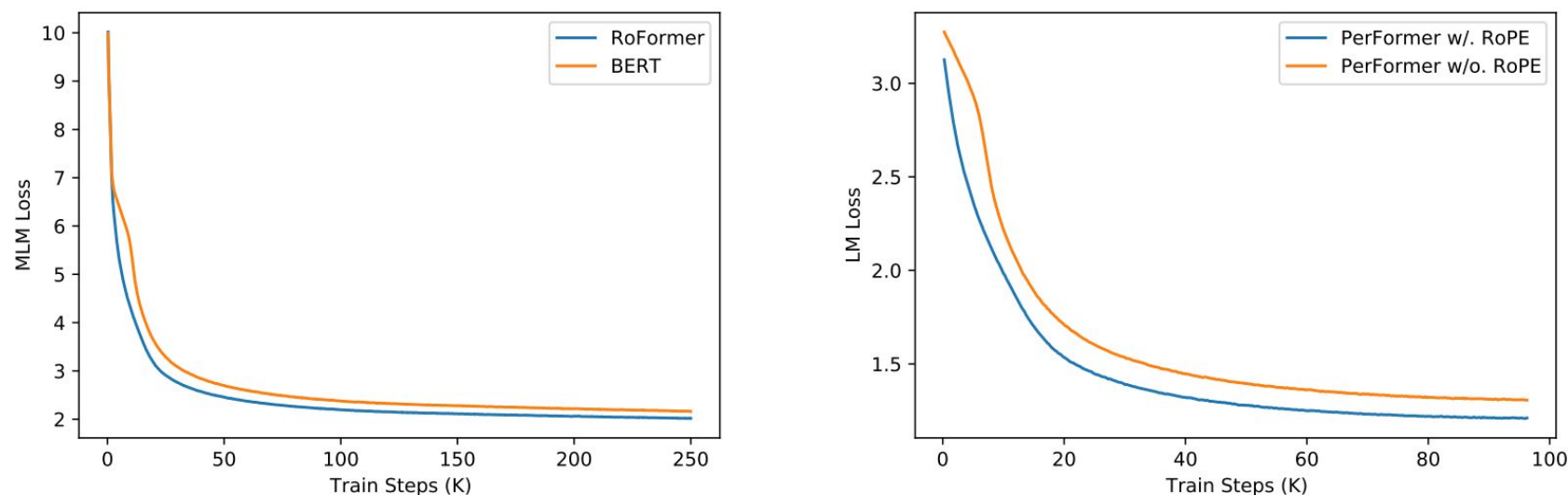


Figure 3: Evaluation of RoPE in language modeling pre-training. **Left:** training loss for BERT and RoFormer. **Right:** training loss for PerFormer with and without RoPE.

What RoPE Promises

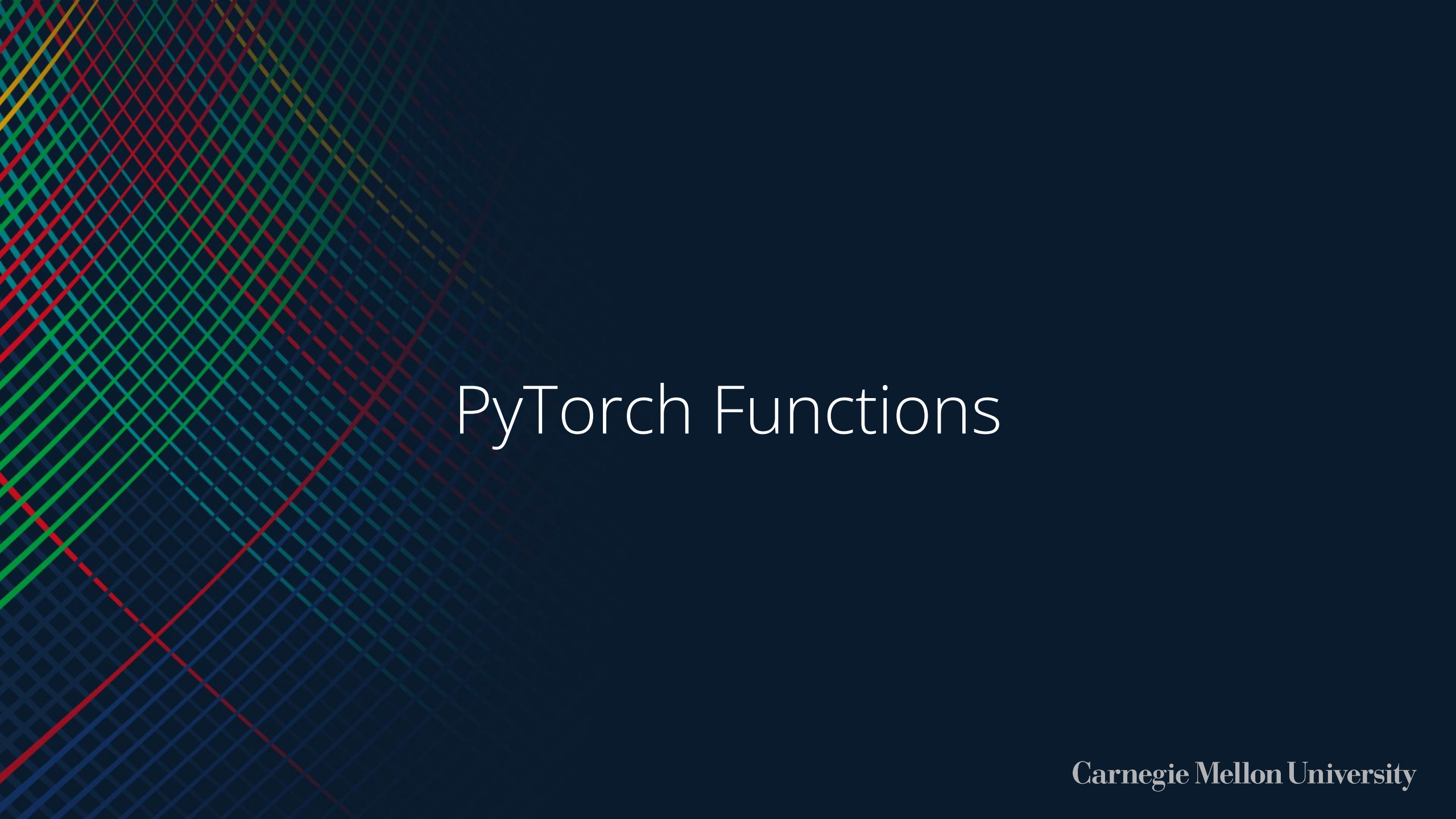
2. Better generalizability to different sequence lengths as opposed to absolute positional embeddings

Absolute Positional Embeddings

- assigns fixed position IDs to tokens
- limited to the sequence lengths seen during training

RoPE

- encodes position as rotations in a continuous space
- preserves relative distances between tokens
- extends naturally to longer sequences without retraining

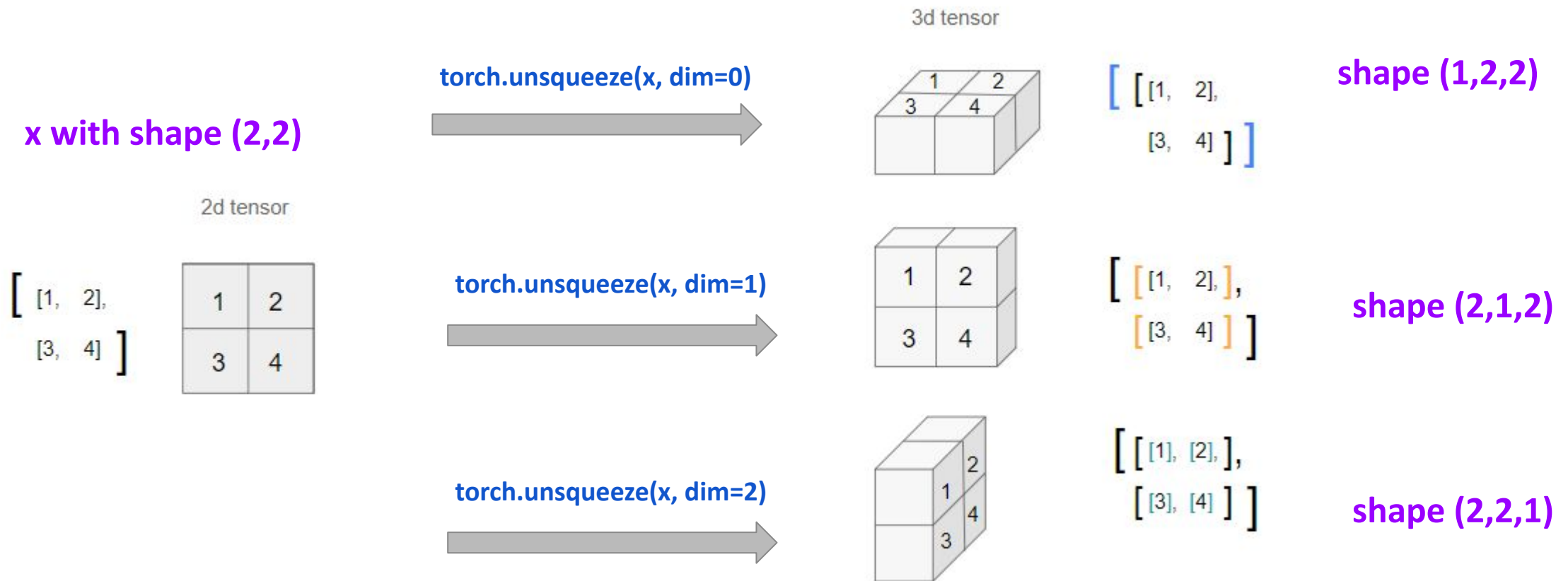


PyTorch Functions

PyTorch functions

[torch.unsqueeze — PyTorch 2.1 documentation](#)

- **`torch.unsqueeze(tensor, dim)`**: returns a new tensor with a dimension of size 1 inserted at `dim`



PyTorch functions

- **`torch.cat(tensors, dim=0, *, out=None)`**: concatenates a sequence of tensors along a specified dimension.

x w/ shape (2,3)



y w/ shape (2,3)



`torch.cat((x,y), dim=0)`



shape (4,3)

`torch.cat((x,y), dim=1)`



shape (2,6)

PyTorch functions

- **torch.arange(start=0, end, step=1)**: returns tensor of values [start, end) with step

```
>>> torch.arange(5)
tensor([ 0,  1,  2,  3,  4])
>>> torch.arange(1, 4)
tensor([ 1,  2,  3])
>>> torch.arange(1, 2.5, 0.5)
tensor([ 1.0000,  1.5000,  2.0000])
```

PyTorch functions

- **`torch.sin(input)`**: returns a new tensor with the sine of the elements of input.

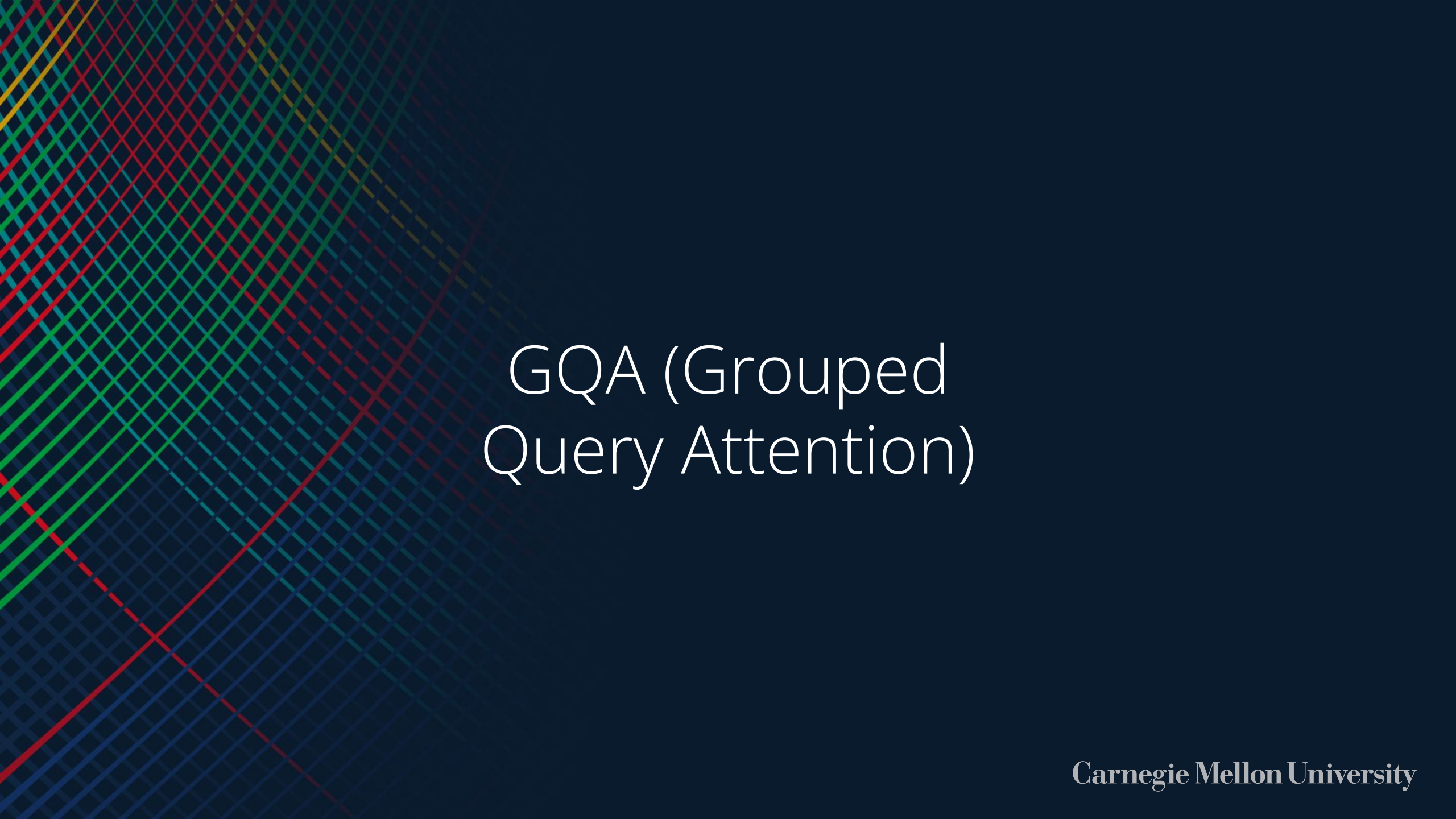
$$\text{out}_i = \sin(\text{input}_i)$$

- **`torch.cos(input)`**: returns a new tensor with the cosine of the elements of input.

$$\text{out}_i = \cos(\text{input}_i)$$

[torch.sin — PyTorch 2.1 documentation](#)

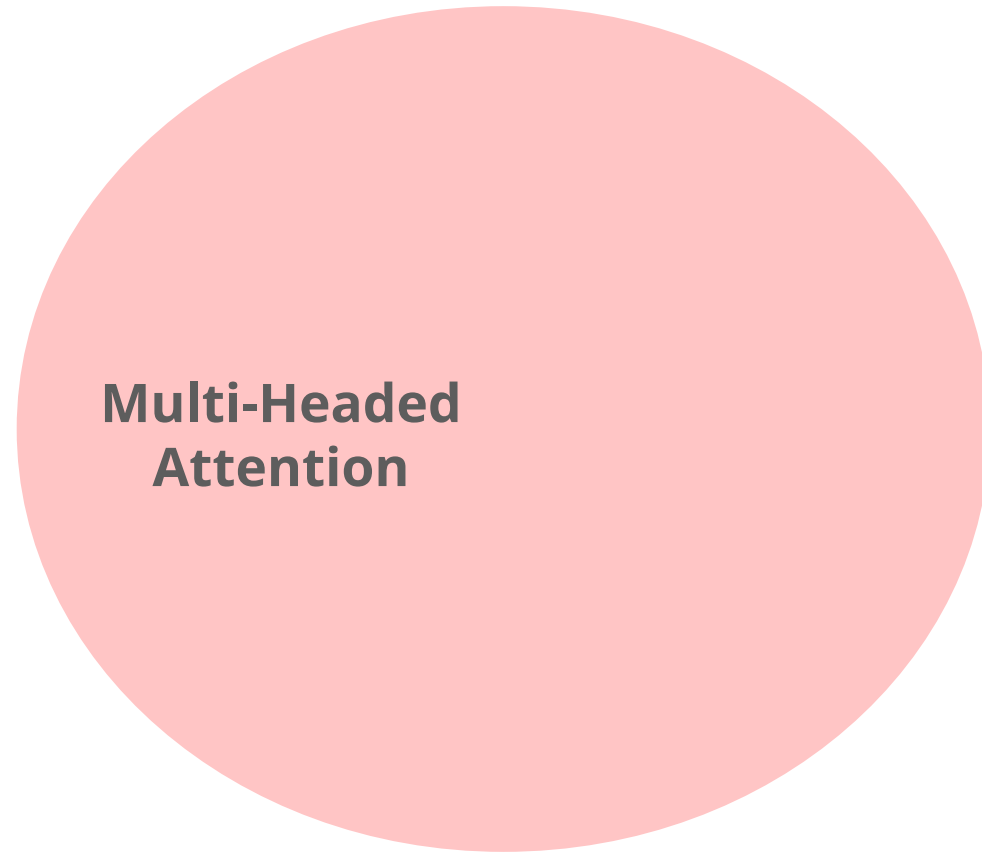
[torch.cos — PyTorch 2.1 documentation](#)



GQA (Grouped Query Attention)

What do ML people even want?

What do ML people want? **Good performance!**

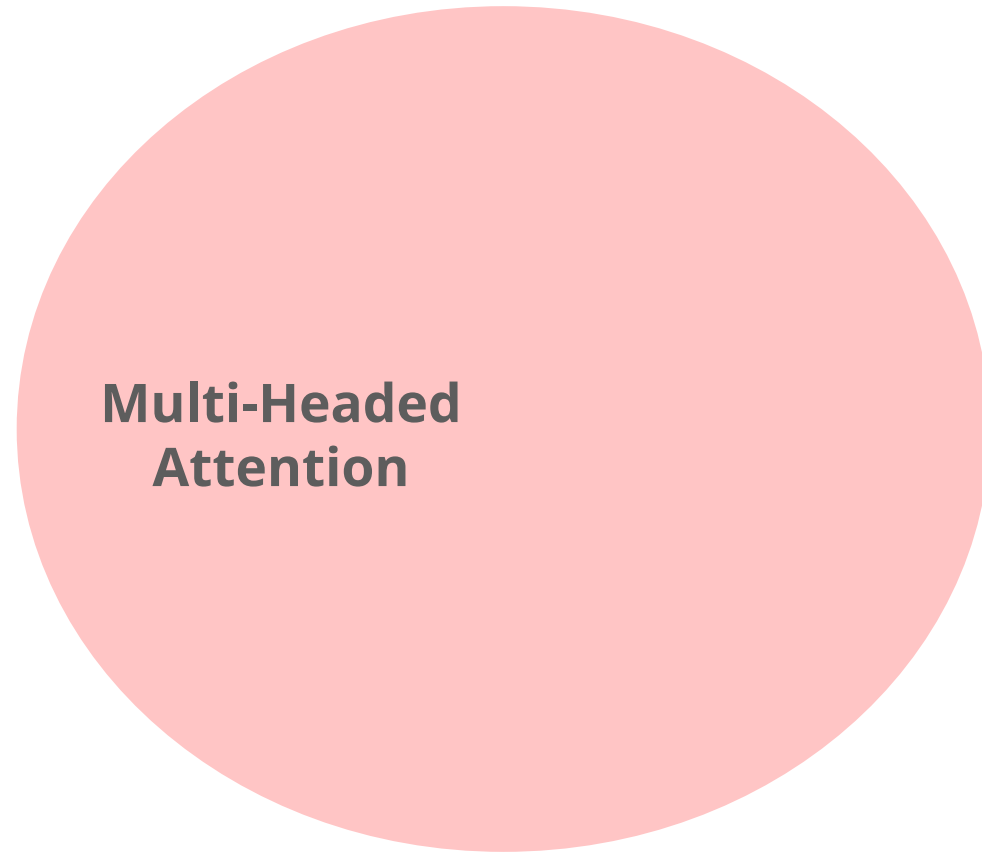


**Multi-Headed
Attention**

Performance

What do ML people want? **Good performance!**

When do they want it?

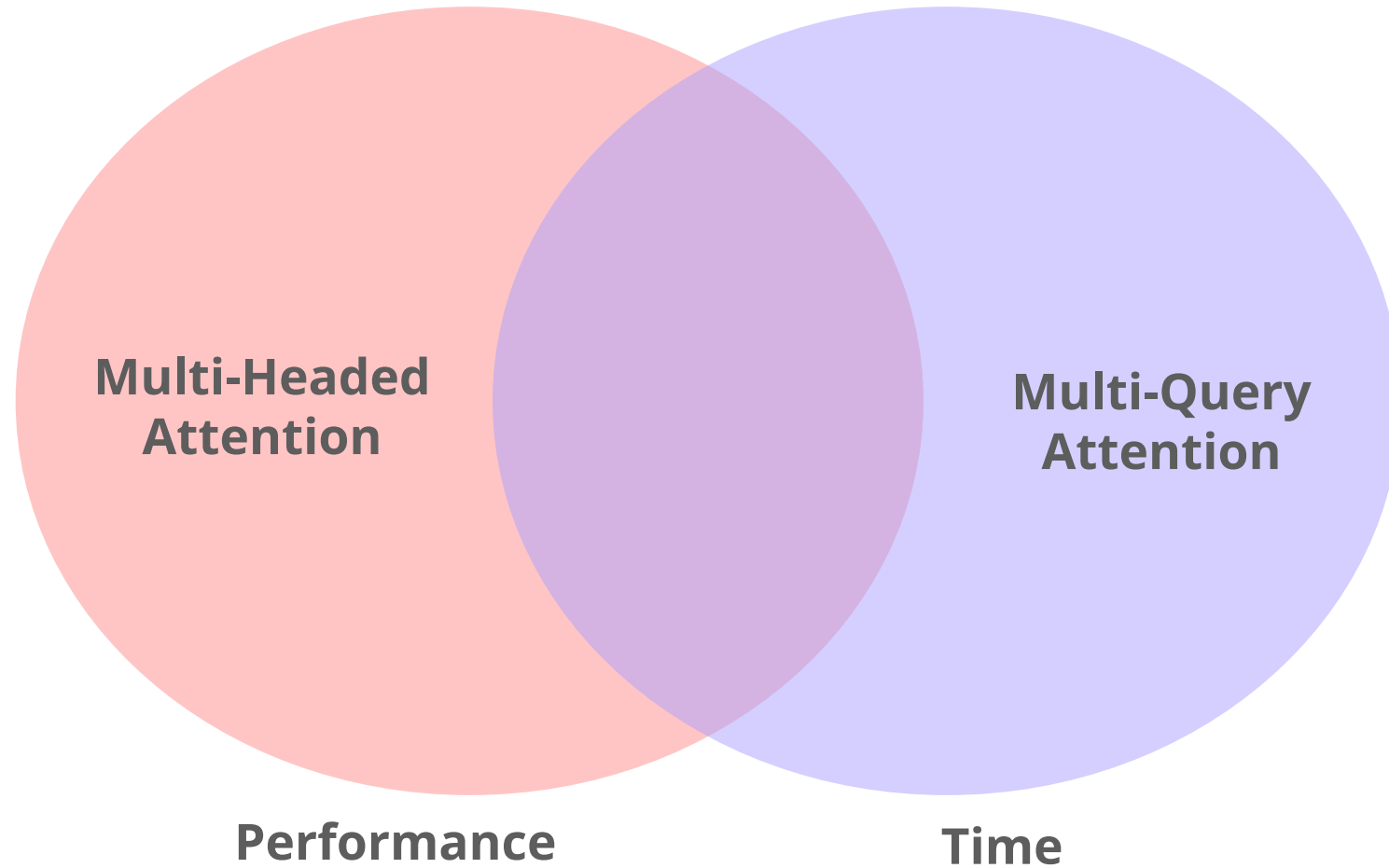


**Multi-Headed
Attention**

Performance

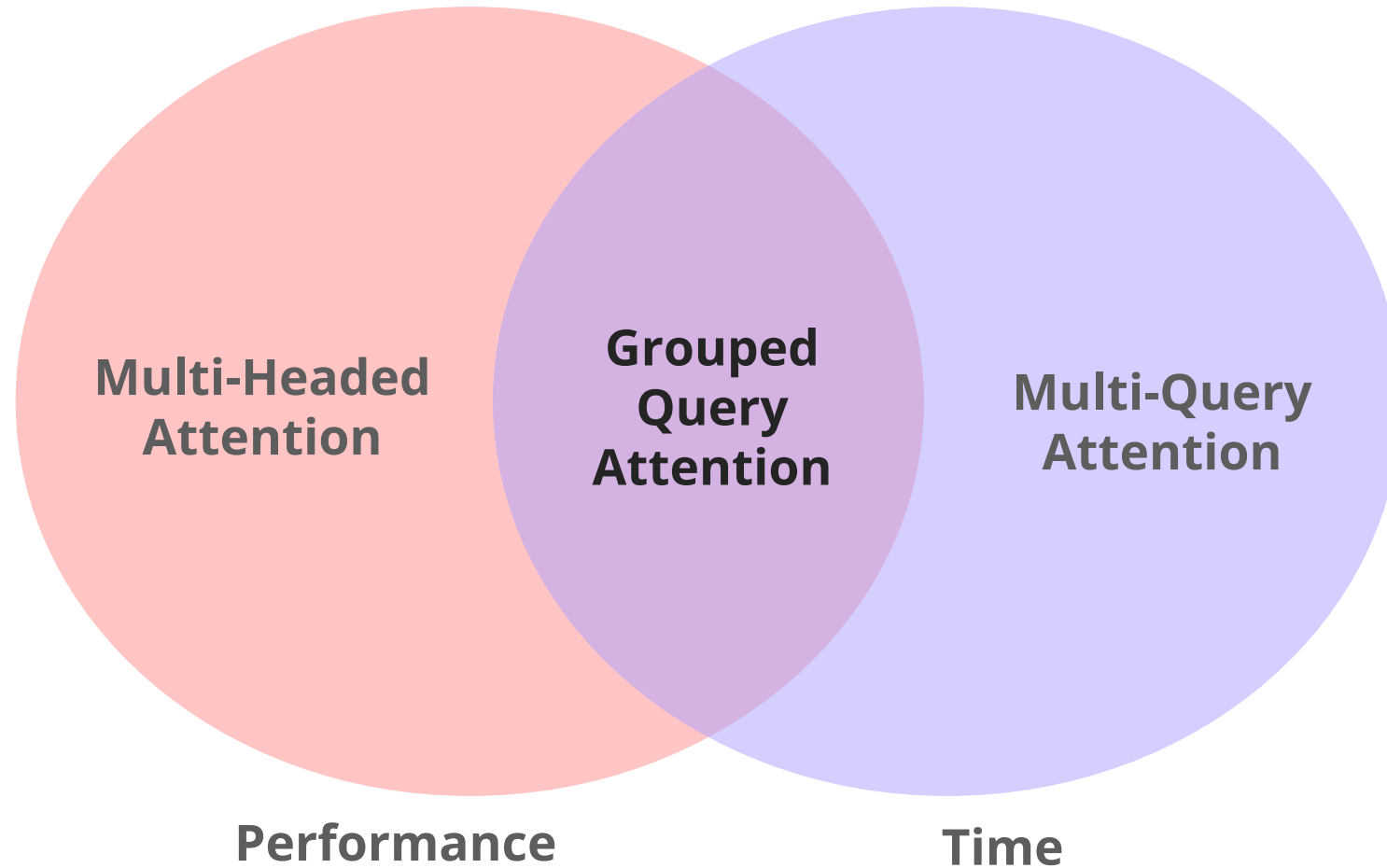
What do ML people want? **Good performance!**

When do they want it? **As fast as humanly possible.**

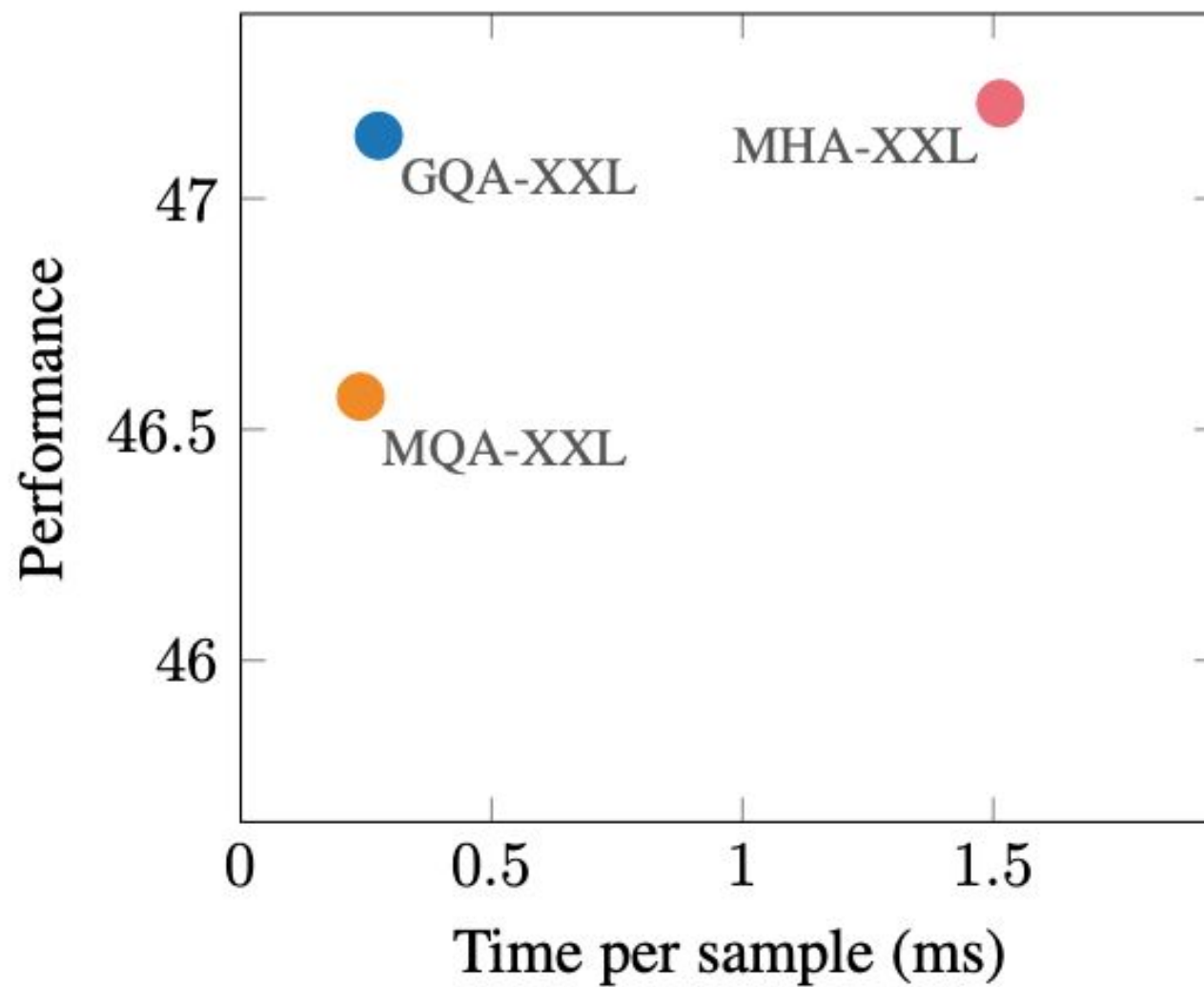


What do ML people want? **Good performance!**

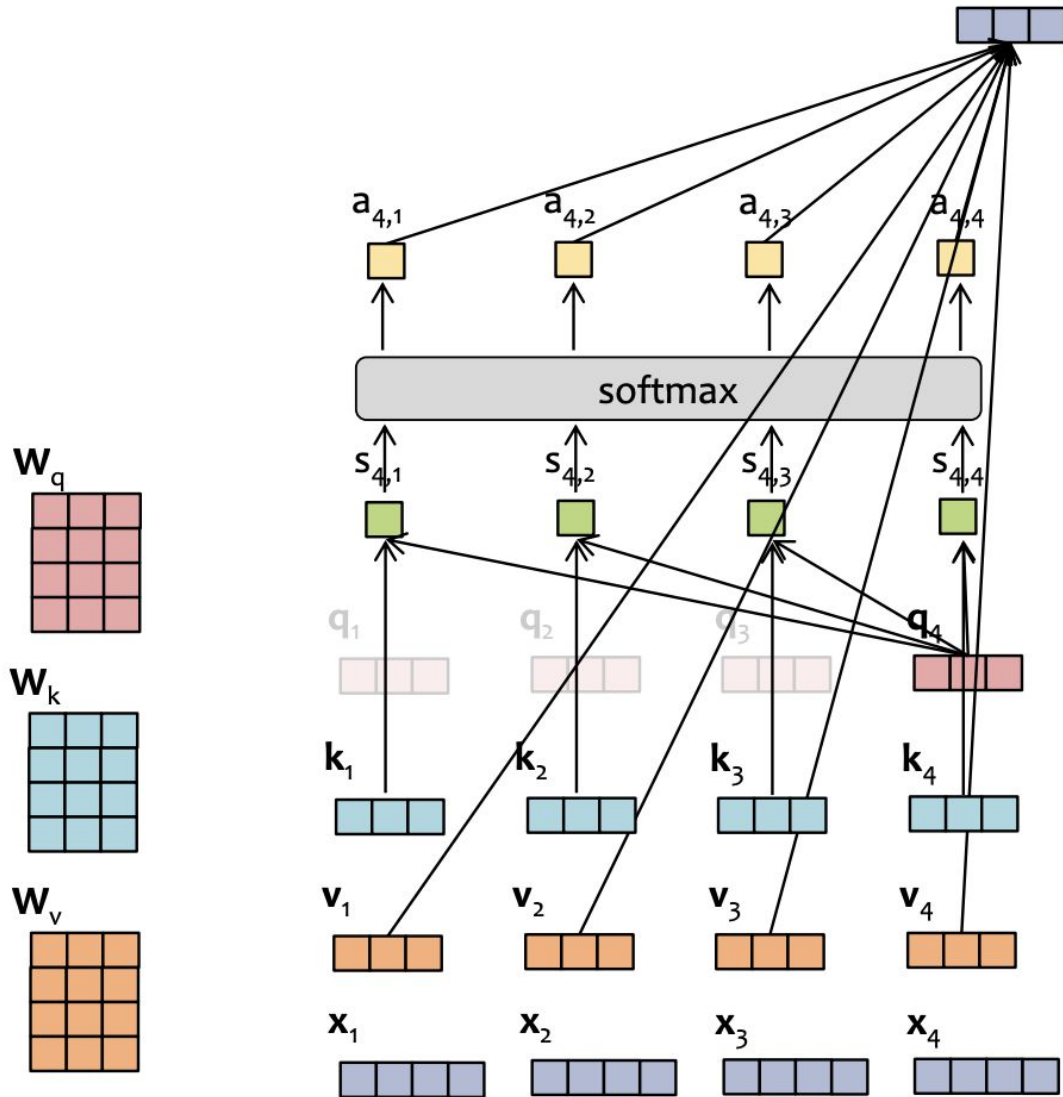
When do they want it? **As fast as humanly possible.**



MHA vs. GQA vs. MQA



Recap: One (1) Scaled Dot-Product Attention Head



$$\mathbf{x}'_t = \sum_{j=1}^t a_{t,j} \mathbf{v}_j$$

$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t)$ attention weights

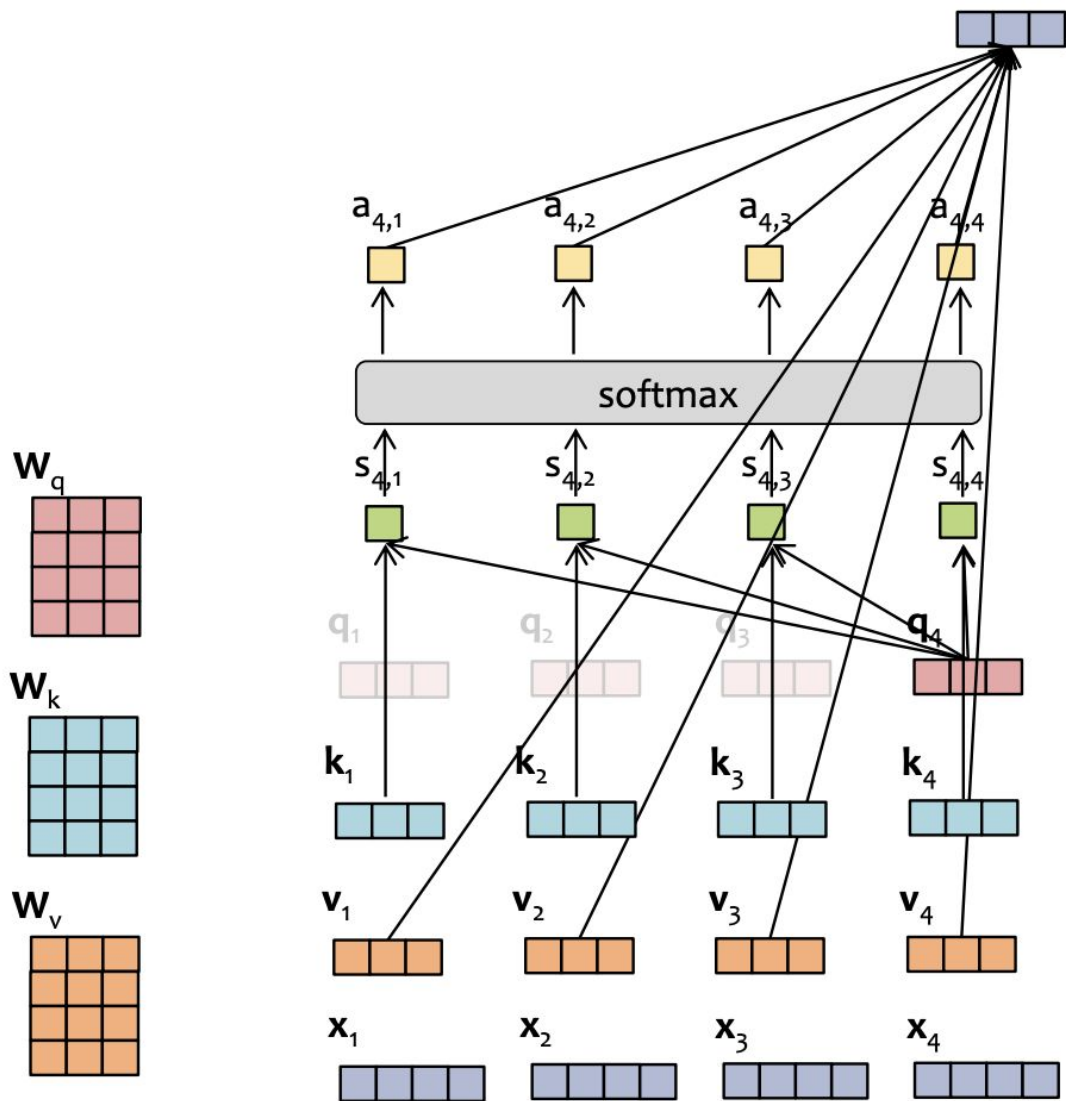
$s_{t,j} = \mathbf{k}_j^T \mathbf{q}_t / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Recap: Scaled Dot-Product Attention



$$\mathbf{x}'_t = \sum_{j=1}^t a_{t,j} \mathbf{v}_j$$

$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t)$ attention weights

$$s_{t,j} = \mathbf{k}_j^T \mathbf{q}_t / \sqrt{d_k} \text{ scores}$$

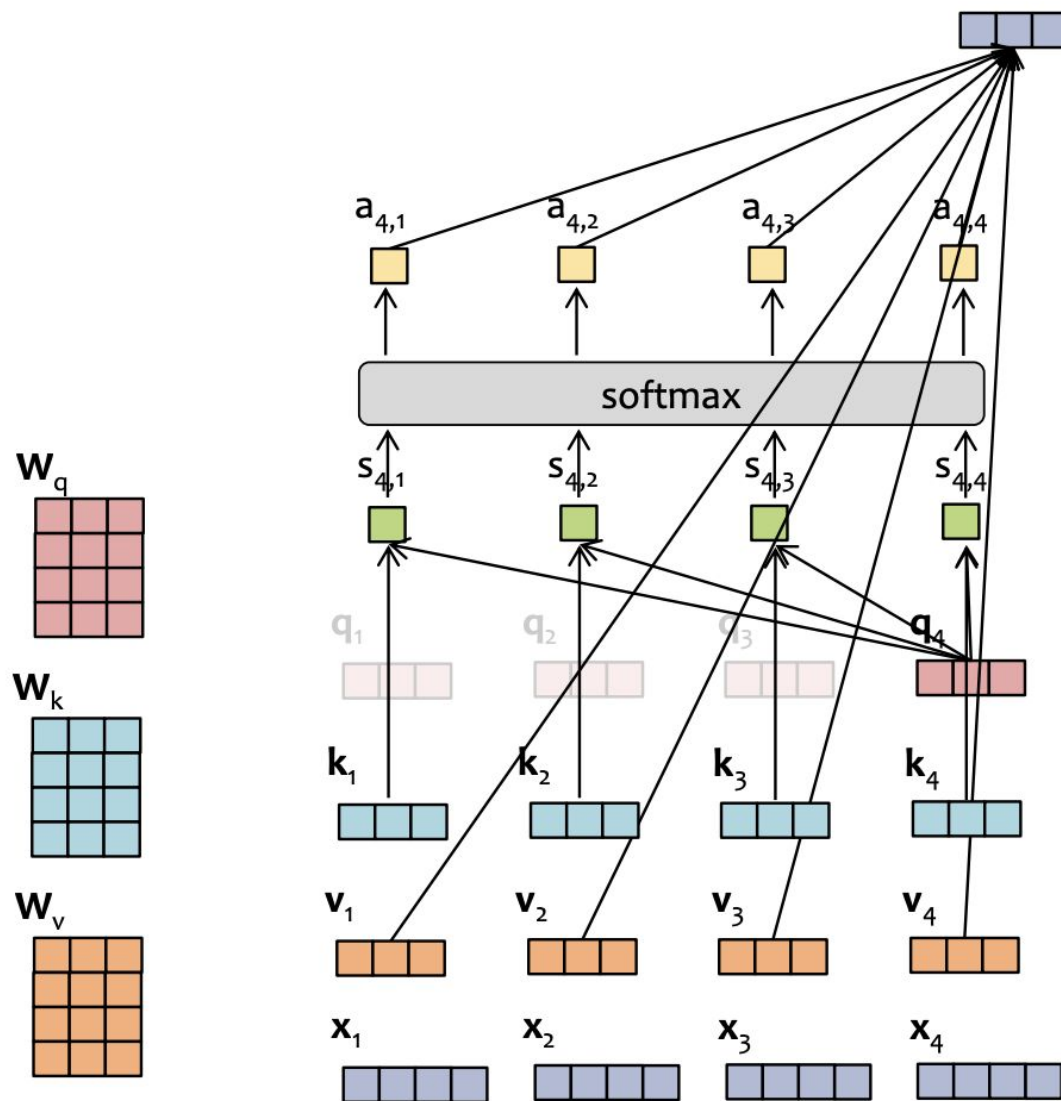
$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j \text{ queries}$$

$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j \text{ keys}$$

$$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j \text{ values}$$

This is used **first**
(KQ)

Recap: Scaled Dot-Product Attention



$$\mathbf{x}'_t = \sum_{j=1}^t a_{t,j} \mathbf{v}_j$$

This is used
second (V)

$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t)$ attention weights

$$s_{t,j} = \mathbf{k}_j^T \mathbf{q}_t / \sqrt{d_k} \text{ scores}$$

This is used **first**
(KQ)

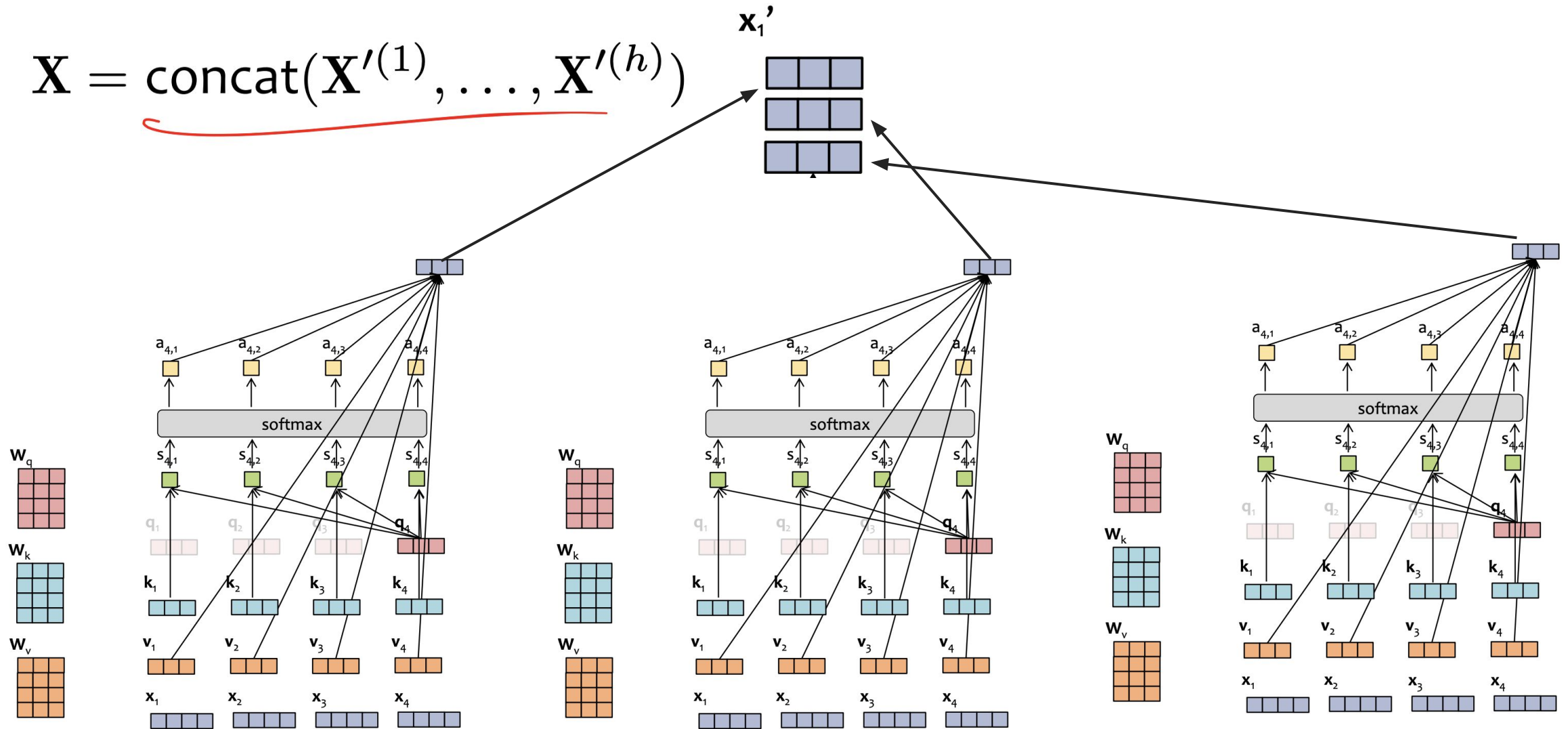
$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j \text{ queries}$$

$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j \text{ keys}$$

$$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j \text{ values}$$

Multi-Headed Attention (**MHA**)

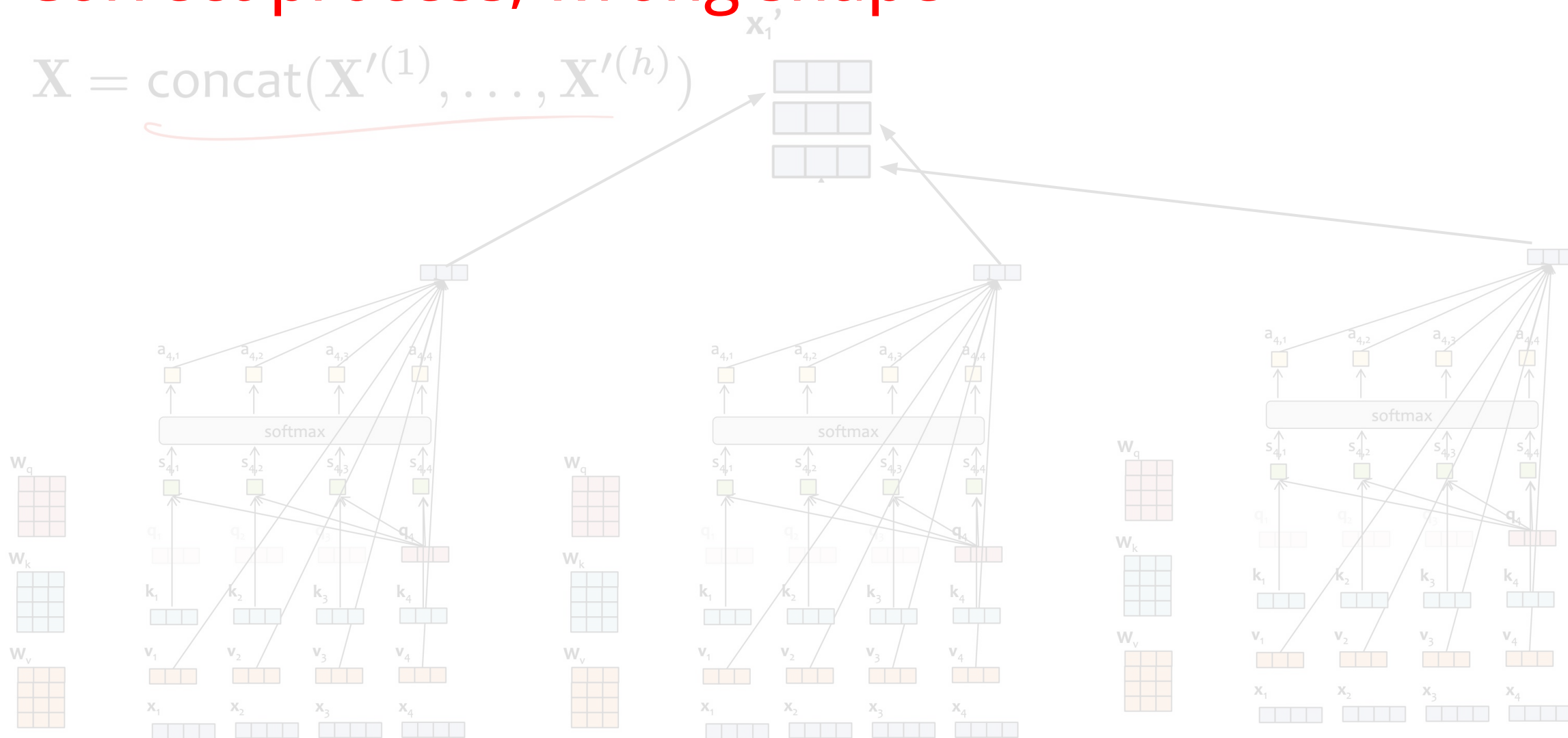
$$\mathbf{X} = \text{concat}(\mathbf{X}'^{(1)}, \dots, \mathbf{X}'^{(h)})$$



Recap: Multi-Headed Attention (**MHA**)

Correct process, wrong shape

$$X = \text{concat}(X'^{(1)}, \dots, X'^{(h)})$$



Recap: Multi-Headed Attention (**MHA**)

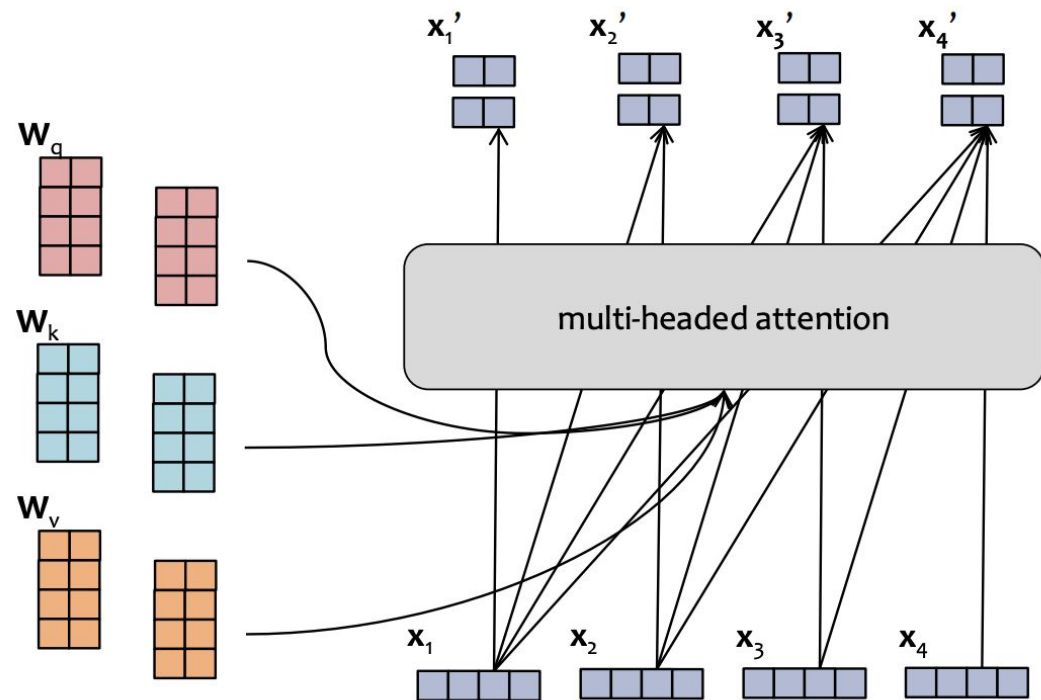
Correct process, wrong shape

1. Independent Attention:

Every head computes its own attention scores with scaled-down query, key, and value matrices.

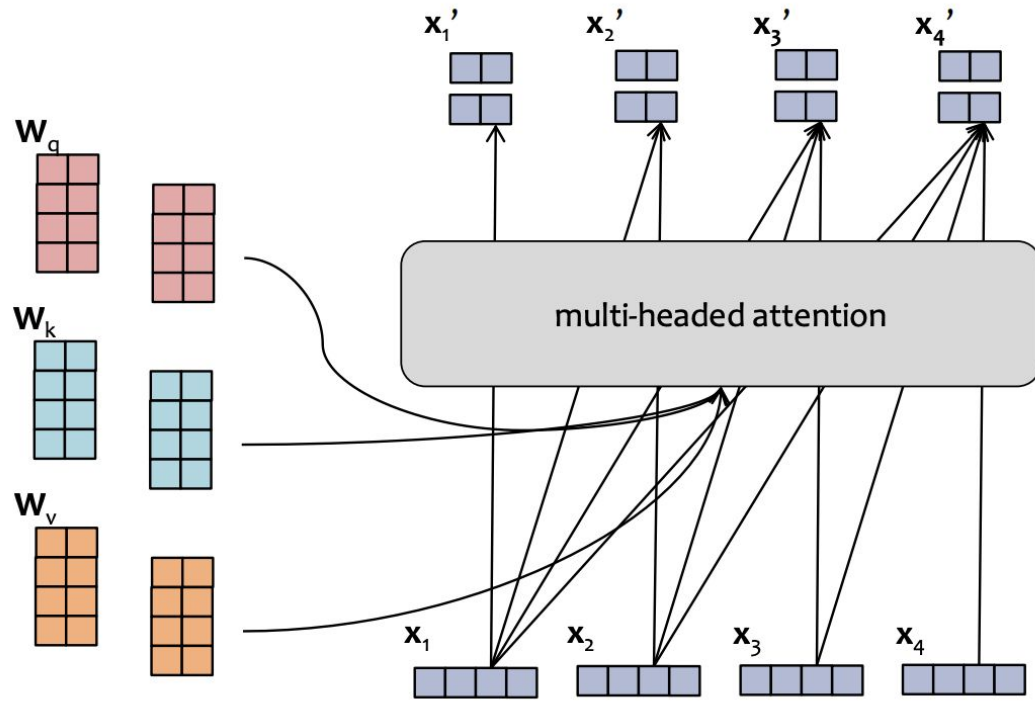
2. But when we concatenate, our **input size must match our output size**

Correct: Multi-Headed Attention (**MHA**)



- In order to make input size = output size, we actually **divide** the size of the QKV weights to make multiple heads, so it matches up when we concatenate it
- Each head's dimension is
$$\frac{d_{\text{model}}}{\text{num_heads}}$$
$$512 \text{ d_model} / 8 \text{ num_heads} = 64 \text{ d_k}$$

Correct: Multi-Headed Attention (**MHA**)

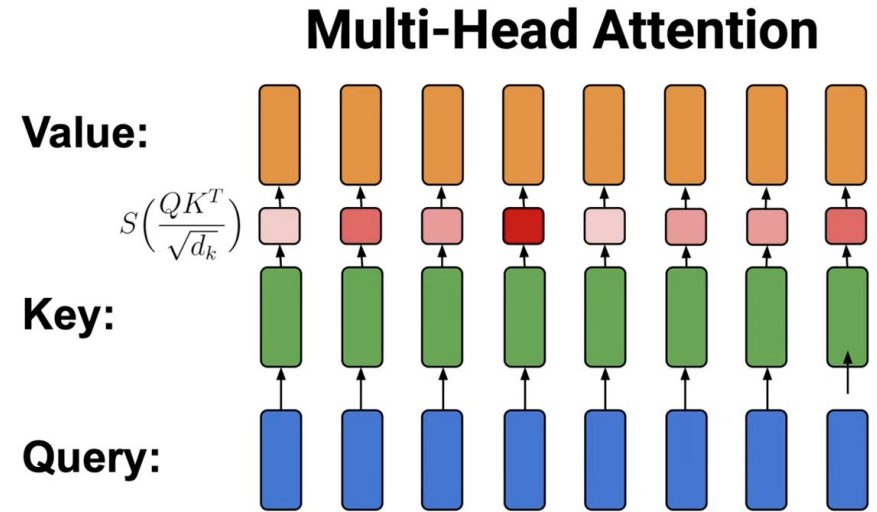


Note: we sometimes refer to d_{model} as **n_embed**

- In order to make input size = output size, we actually **divide** the size of the QKV weights to make multiple heads, so it matches up when we concatenate it
- Each head's dimension is $d_{\text{model}} / \text{num_heads}$
 $512 d_{\text{model}} / 8 \text{ num_heads} = 64 d_k$
- QKV vectors are $[d_k \times 1]$
- QKV weights are $[d_{\text{model}} \times d_k]$

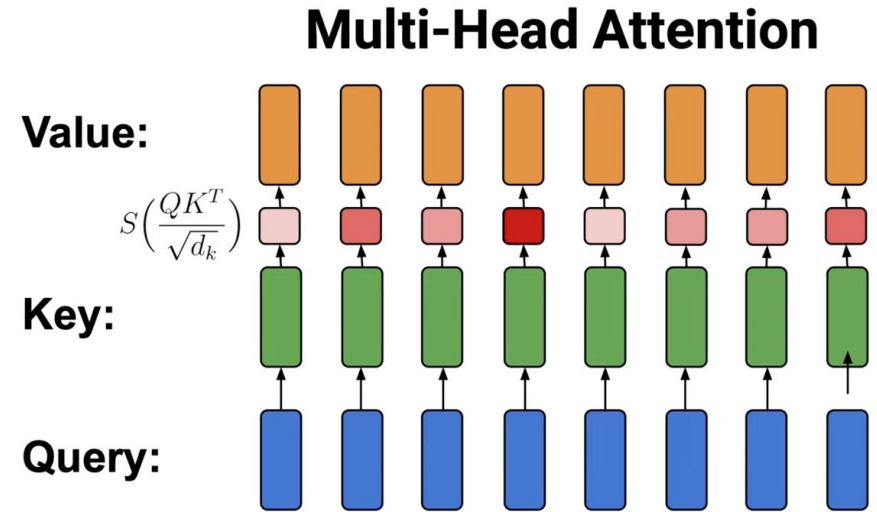
Multi-Headed Attention (**MHA**)

- **Independent Attention:**
Every head computes its own attention scores with scaled-down query, key, and value matrices.
- **Attention Matrix Size:**
Each head generates a (seq_len, seq_len) attention matrix, representing pairwise attention scores in the sequence.



Multi-Headed Attention (**MHA**)

- **Incremental Generation:**
Utilizes a step-by-step approach during inference, feeding single tokens into the network progressively.
- **Sequence Length ($S = 1$):**
Each step involves adding only one token at a time.
- **Dynamic K and V:**
Key (K) and Value (V) matrices expand with each new token, encompassing all tokens seen so far.
- **Issues:**
Requires repeatedly loading large K and V tensors, Memory and computational issues arise.



Multi-Query Attention (**MQA**)

- **Dimensions:**

For each head, the dimension of Q is $(seq_len, d_model / num_heads)$ where seq_len is the sequence length.

Both K and V will have dimensions of $(seq_len, d_model / num_heads)$.

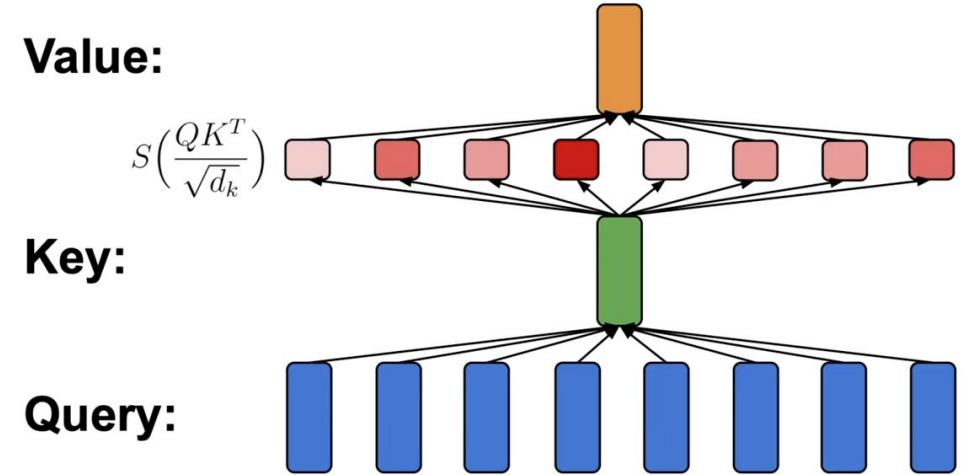
- **Computation:**

The amount of computation in incremental multi-query attention (MQA) is comparable to that in multi-head attention (MHA).

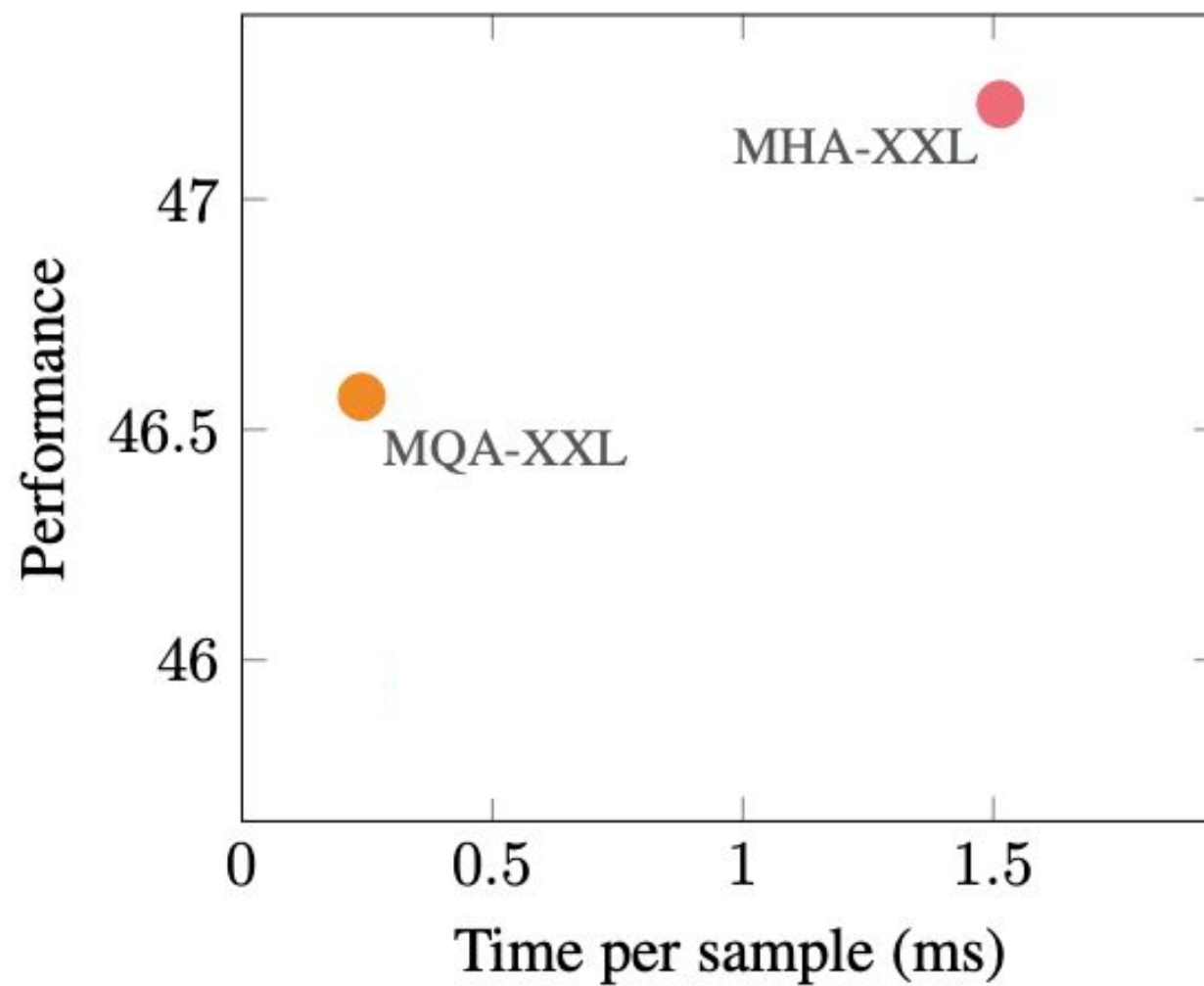
- **Memory Efficiency:**

MQA reduces the amount of data read/written from memory, decreasing the memory footprint.

- **Issues:** Performance & Information Loss

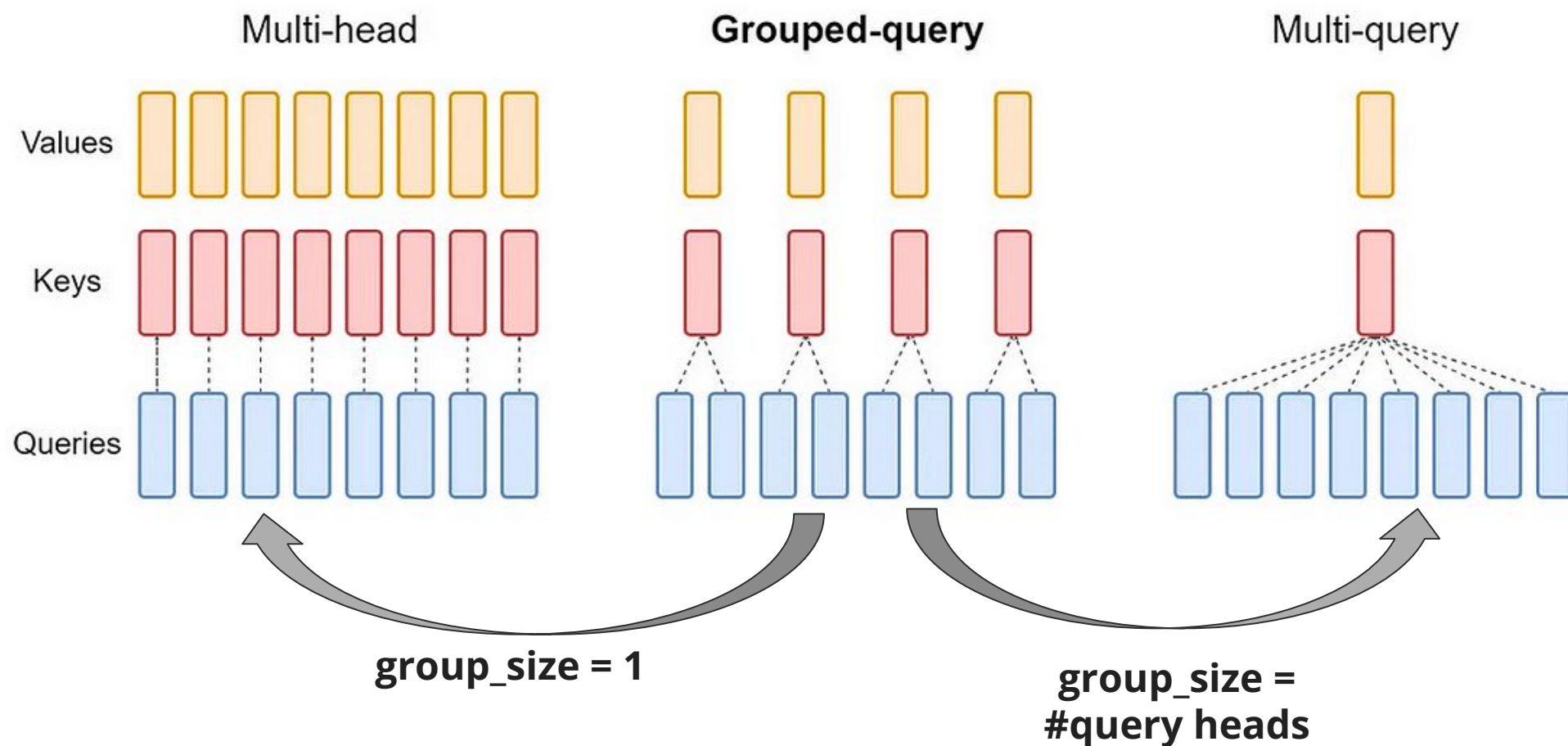


MHA vs MQA Comparison



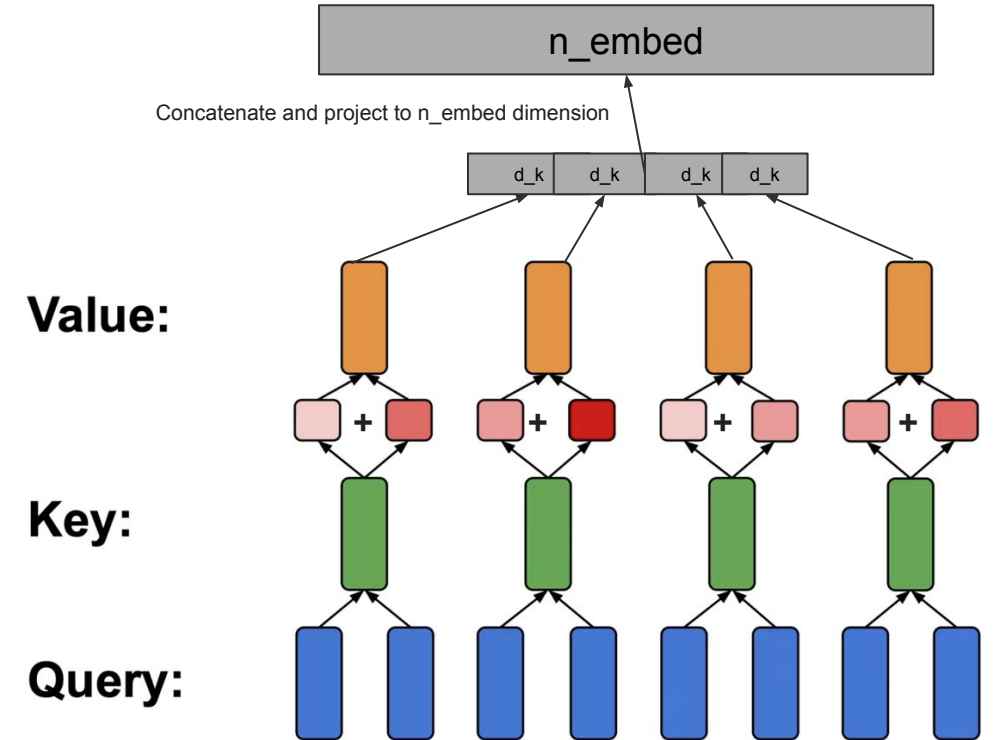
Grouped Query Attention (**GQA**)

and this notion of **g = group size**



Grouped Query Attention (**GQA**)

- Same number of **query** heads as MHA or MQA
- $\text{\#KV heads (MQA)} < \text{\#KV heads (GQA)} < \text{\#KV heads (MHA)}$
- **Attention Computation:** Same as MQA or MHA
- **Total Computation:** $\text{MQA} < \text{GQA} < \text{MHA}$
- **Memory Efficiency:** $\text{MQA} < \text{GQA} < \text{MHA}$



Grouped Query Attention (**GQA**) - Steps

1. Each query matrix within a group is multiplied by the key matrix of the group

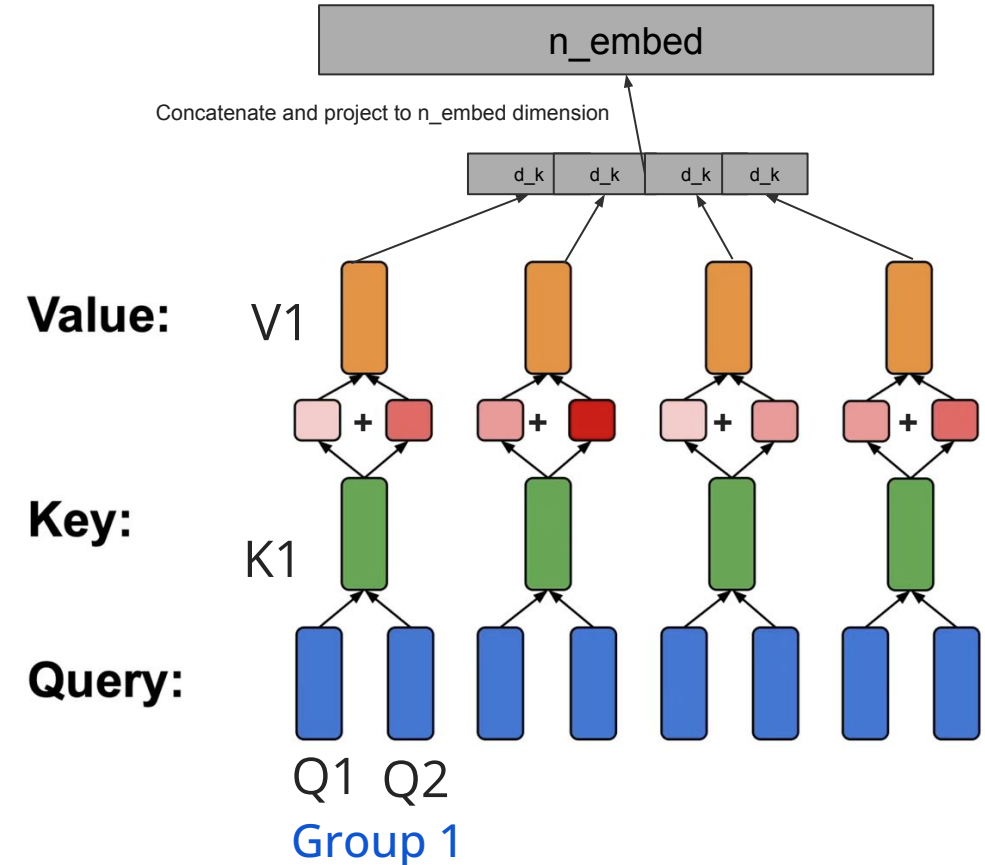
$$\mathbf{S}^{(i,j)} = \mathbf{Q}^{(i,j)} (\mathbf{K}^{(i)})^T / \sqrt{d_k},$$

2. Softmax on the score matrices + mask
3. The attention matrices within a group are summed up (element-wise) to form one attention matrix per group ($\mathbf{A}(i)$)

4. One attention matrix per group is multiplied with the value matrix of the group

$$\mathbf{X}'^{(i,j)} = \mathbf{A}^{(i,j)} \mathbf{V}^{(i)}$$

5. Output per group is of dimension d_k
6. Outputs from each group are concatenated and then scaled back to the n_embed dimension.



Grouped Query Attention (**GQA**) - Equations

Implement this:

$$\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_T]^T$$

$$\mathbf{V}^{(i)} = \mathbf{X}\mathbf{W}_v^{(i)}, \forall i \in \{1, \dots, h_{kv}\}$$

$$\mathbf{K}^{(i)} = \mathbf{X}\mathbf{W}_k^{(i)}, \forall i \in \{1, \dots, h_{kv}\}$$

$$\mathbf{Q}^{(i,j)} = \mathbf{X}\mathbf{W}_q^{(i,j)}, \forall i \in \{1, \dots, h_{kv}\}, \forall j \in \{1, \dots, g\}$$

$$\mathbf{S}^{(i,j)} = \mathbf{Q}^{(i,j)}(\mathbf{K}^{(i)})^T / \sqrt{d_k}, \quad \forall i \in \{1, \dots, h_{kv}\}, \forall j \in \{1, \dots, g\}$$

$$\mathbf{A}^{(i,j)} = \text{softmax}(\mathbf{S}^{(i,j)}), \quad \forall i \in \{1, \dots, h_{kv}\}, \forall j \in \{1, \dots, g\}$$

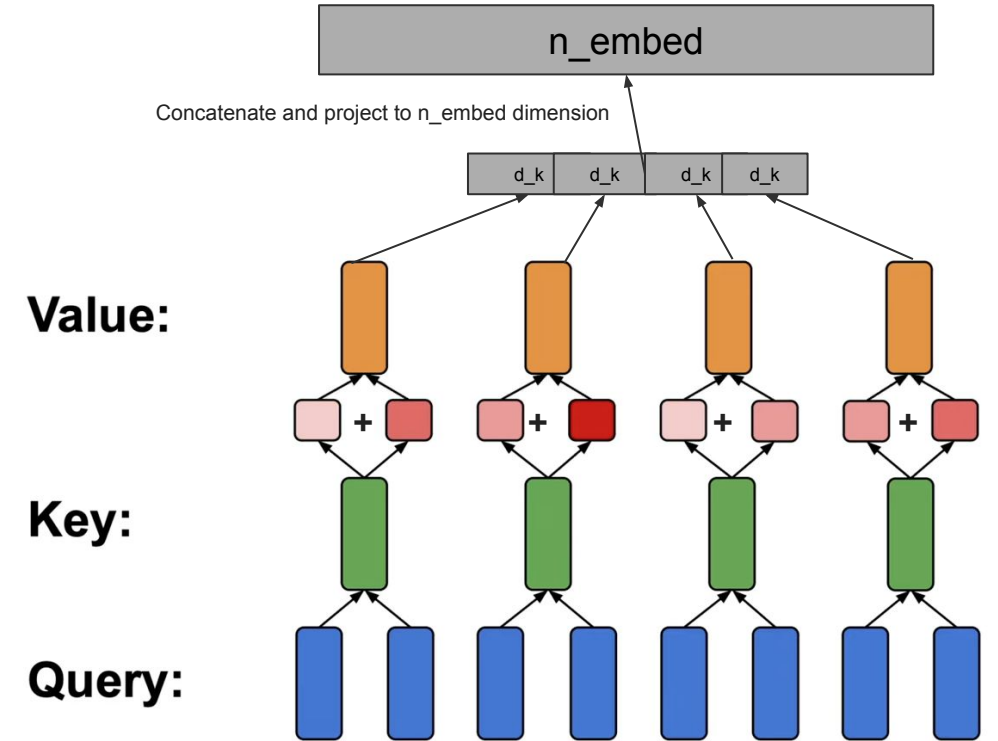
$$\mathbf{X}'^{(i,j)} = \mathbf{A}^{(i,j)}\mathbf{V}^{(i)}, \quad \forall i \in \{1, \dots, h_{kv}\}, \forall j \in \{1, \dots, g\}$$

$$\mathbf{X}' = \text{concat}(\mathbf{X}'^{(i,j)}), \quad \forall i \in \{1, \dots, h_{kv}\}, \forall j \in \{1, \dots, g\}$$

$$\mathbf{X} = \mathbf{X}'\mathbf{W}_o \quad (\text{where } \mathbf{W}_o \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}})$$

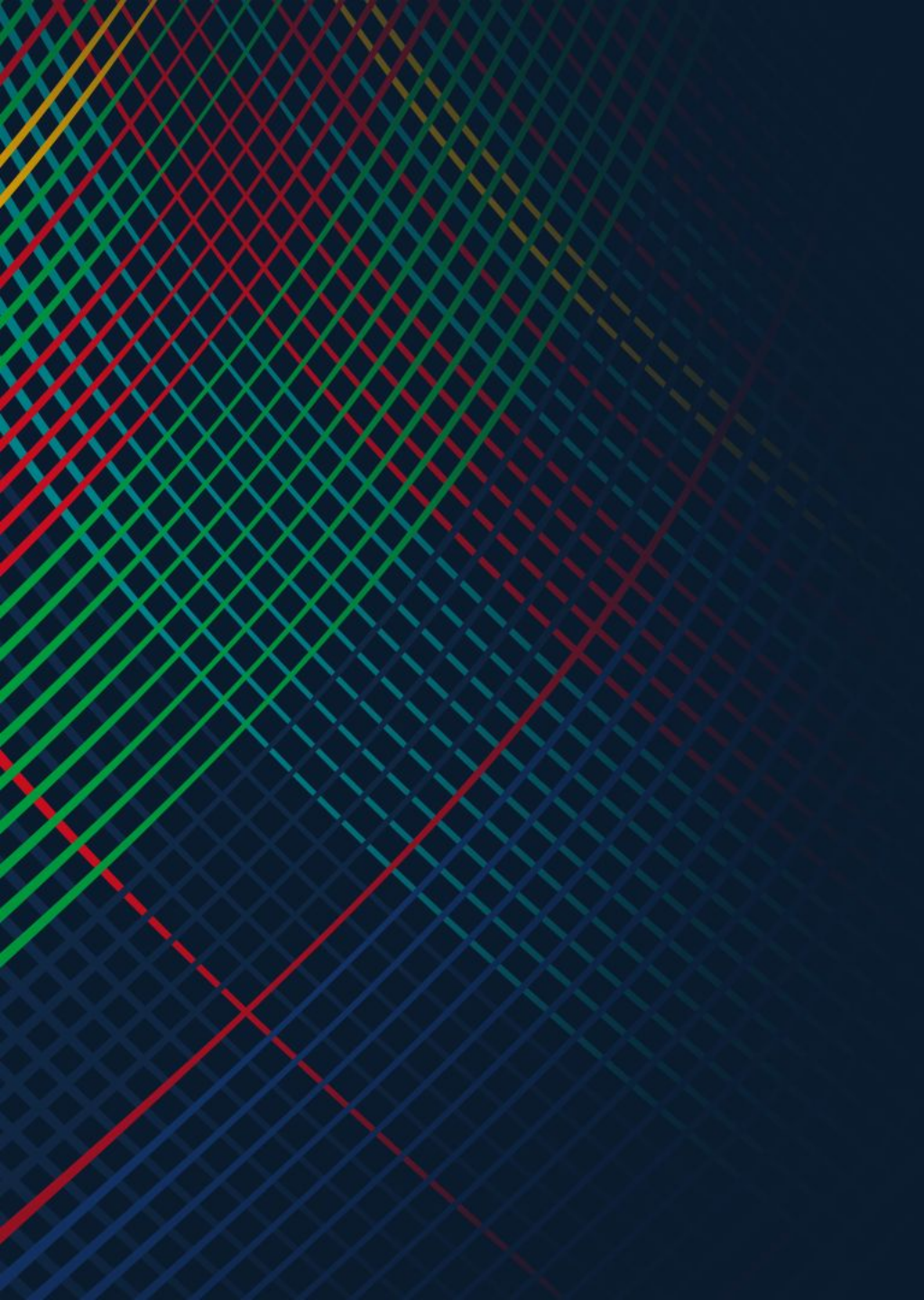
Steps:

1. Each query matrix within a group is multiplied by the key matrix of the group
2. Each attention matrix is multiplied with the value matrix of the group
3. Output per group is of dimension $d_k * h_{kv}$
4. Outputs from each group are concatenated and then projected back to the n_embed dimension.



Grouped Query Attention (**GQA**) - **Matrices**

- X actually starts as [batch size x seq_len x n_embd]
- QKV will need to be split along n_embed accordingly



Einops

Einops Library

Saves you a few lines of code

```
import einops
import torch

# Example tensor with shape: batch, channels, height, width
x = torch.rand(10, 3, 32, 32)

# Applying the einops rearrange operation
x = einops.rearrange(x, 'b c h w -> (b w) c h')

# New shape of x (after rearrangement): (320, 3, 32)
print(x.shape)
```

`einops.rearrange()`

```
import torch

# Example tensor with shape: batch, channels, height, width
x = torch.rand(10, 3, 32, 32)

# Permute the dimensions of x
x = x.permute(0, 3, 1, 2)

# Unpack the shape of the permuted tensor
N, W, C, H = x.shape

# Reshape the tensor
x = x.contiguous().view(N * W, C, -1)
```

torch equivalent code

Context

```
1  import torch
2  from einops import rearrange, reduce, repeat
3
4  # b = batch size
5  # c = number of channels
6  # h = height
7  # w = width
```

1. einops.rearrange()

```
9      # 1. REARRANGE
10
11     # a. Flattening/Reducing Dimensions
12
13     x = torch.randn(8, 3, 32, 32)
14     # shape of x is 8x3x32x32
15
16     y = rearrange(x, 'b c h w -> b c (h w)')
17     # shape of y is 8x3x1024
18
19     z = rearrange(x, 'b c h w -> b (c h w)')
20     # shape of z is 8x3072
```

1. einops.rearrange()

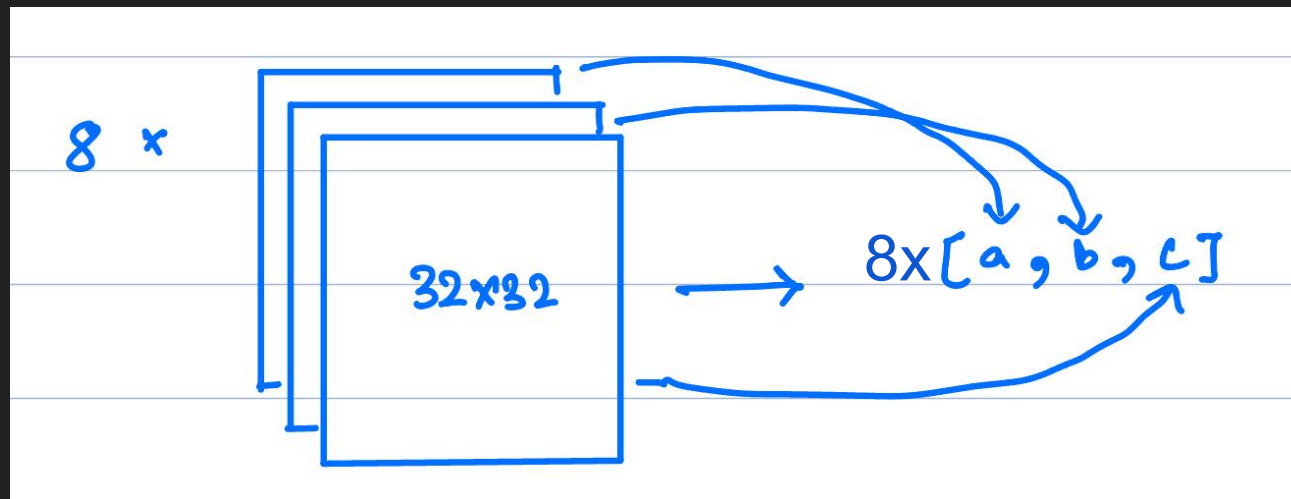
```
22     # b. Expanding Dimensions
23
24     z = torch.randn(8, 3072)
25     # shape of z is 8x3072
26
27     y = rearrange(z, 'b (c hw) -> b c hw', c=3)
28     # shape of y is 8x3x1024
29
30     x = rearrange(y, 'b c (h w) -> b c h w', h=32)
31     # shape of x is 8x3x32x32
32
33     #Alternatively, we can go directly from z to x
34     x = rearrange(z, 'b (c h w) -> b c h w', c=3, h=32)
35     # shape of x is 8x3x32x32 again
```

1. einops.rearrange()

```
37     # c. Permuting Dimensions
38
39     x = torch.randn(8,32,32,3)
40     # shape of x is 8x32x32x3
41
42     y = rearrange(x, 'b h w c -> b c h w')
43     # shape of y is 8x3x32x32
```

2. einops.reduce()

```
45 # 2. REDUCE
46
47 # a. Summing over Dimensions
48
49 x = torch.randn(8, 3, 32, 32)
50 # shape of x is 8x3x32x32
51
52 y = reduce(x, 'b c h w -> b c', 'sum')
53 # shape of y is 8x3 and each element is the sum of all elements in the corresponding 32x32 matrix
54
55 # Similarly, we can average, or take the maximum or minimum over dimensions
56 # by replacing 'sum' with 'mean', 'max', or 'min' respectively
```



3. einops.repeat()

```
58     # 3. REPEAT
59
60     x = torch.randn(8, 3, 32)
61     # shape of x is 8x3x32
62
63     y = repeat(x, 'b c h -> b c h w', w=32)
64     # shape of y is 8x3x32x32
```

rearrange Example (CausalSelfAttention)

```
# split the embedding dimension (n_embd) across number of heads with new 'h' dim
# reshape the query, key, value tensors to increase efficiency of matrix mult
# b = batch size, t = sequence length, h = # of heads, d = n_embd / # of heads
q = rearrange(q, 'b t (h d) -> b h t d', h=self.n_head)
k = rearrange(k, 'b t (h d) -> b h t d', h=self.n_head)
v = rearrange(v, 'b t (h d) -> b h t d', h=self.n_head)
```

```
# rearrange the output tensor to (batch size, sequence length, n_embd)
# re-assemble all head outputs side by side
y = rearrange(y, 'b h q d -> b q (h d)')
```

Einops Library - einsum()

```
A = torch.randn(2, 3, 4, 5) # Shape: (B, T, D, Qh)
B = torch.randn(2, 3, 5, 6) # Shape: (B, T, Qh, Vh)

# 4D matrix multiplication using einops.einsum with spaces for clarity
result = einsum('B T D Qh, B T Qh Vh -> B T D Vh', A, B)

print(result.shape) # Expected shape: (2, 3, 4, 6)
```

`einops.einsum()`

```
# Example 4D tensors
A = torch.randn(2, 3, 4, 5) # Shape: (a, b, c, d)
B = torch.randn(2, 3, 5, 6) # Shape: (a, b, d, e)

# 4D matrix multiplication using torch.einsum
# Multiplying across dimensions a, b, and d, and summing over d
result = torch.einsum('abcd,abde->abce', A, B)

print(result.shape) # Expected shape: (2, 3, 4, 6)
```

`torch.einsum()`

1. Summation over axes

```
import numpy as np

# 1. Summation (Reduce Over an Axis)
x = np.array([1, 2, 3])
print(np.einsum('i ->', x)) # Sum over all elements (Output: 6)

A = np.array([[1, 2], [3, 4]])
print(np.einsum('ij -> i', A)) # Sum along columns (Output: [3, 7])
```

2 and 3. Transpose and Dot Product

```
# 2. Transposition (Reordering Dimensions)
```

```
print(np.einsum('ij -> ji', A)) # Transpose matrix (Output: [[1, 3], [2, 4]])
```

```
# 3. Dot Product (Vector Inner Product)
```

```
a = np.array([1, 2, 3])
```

```
b = np.array([4, 5, 6])
```

```
print(np.einsum('i, i ->', a, b)) # Dot product (Output: 32)
```

4, 5 and 6. MV multiplication, MM multiplication, Outer product

```
# 4. Matrix-Vector Multiplication
x = np.array([5, 6])
print(np.einsum('ij, j -> i', A, x)) # Matrix-vector multiplication (Output: [17, 39])

# 5. Matrix Multiplication
B = np.array([[5, 6], [7, 8]])
print(np.einsum('ik, kj -> ij', A, B)) # Matrix multiplication (Output: [[19, 22], [43, 50]])

# 6. Outer Product (Expanding Dimensions)
c = np.array([3, 4])
print(np.einsum('i, j -> ij', a, c)) # Outer product (Output: [[3, 4], [6, 8], [9, 12]])
```

7. Matrix multiplication in batches

```
# 7. Batch Matrix Multiplication (Common in Deep Learning)
A_batch = np.random.randn(5, 3, 4) # (batch=5, 3x4 matrices)
B_batch = np.random.randn(5, 4, 2) # (batch=5, 4x2 matrices)
result = np.einsum('bij, bjk -> bik', A_batch, B_batch)
print(result.shape) # Output: (5, 3, 2)
```

EXAMPLES

```
# EXAMPLE
```

```
## Self-Attention in Transformers
```

```
Q = np.random.randn(10, 8, 64) # (batch, heads, features)
```

```
K = np.random.randn(10, 8, 64)
```

```
V = np.random.randn(10, 8, 64)
```

```
attention_scores = np.einsum('bhd, bhd -> bh', Q, K) # Compute attention scores
```

```
print(attention_scores.shape) # Output: (10, 8)
```

```
# EXAMPLE
```

```
## Convolution Operation (Simplified Example)
```

```
image = np.random.randn(1, 3, 32, 32) # (batch, channels, height, width)
```

```
kernel = np.random.randn(3, 3, 3, 3) # (out_channels, in_channels, kernel_h, kernel_w)
```

```
output = np.einsum('bchw, oihw -> boh', image, kernel)
```

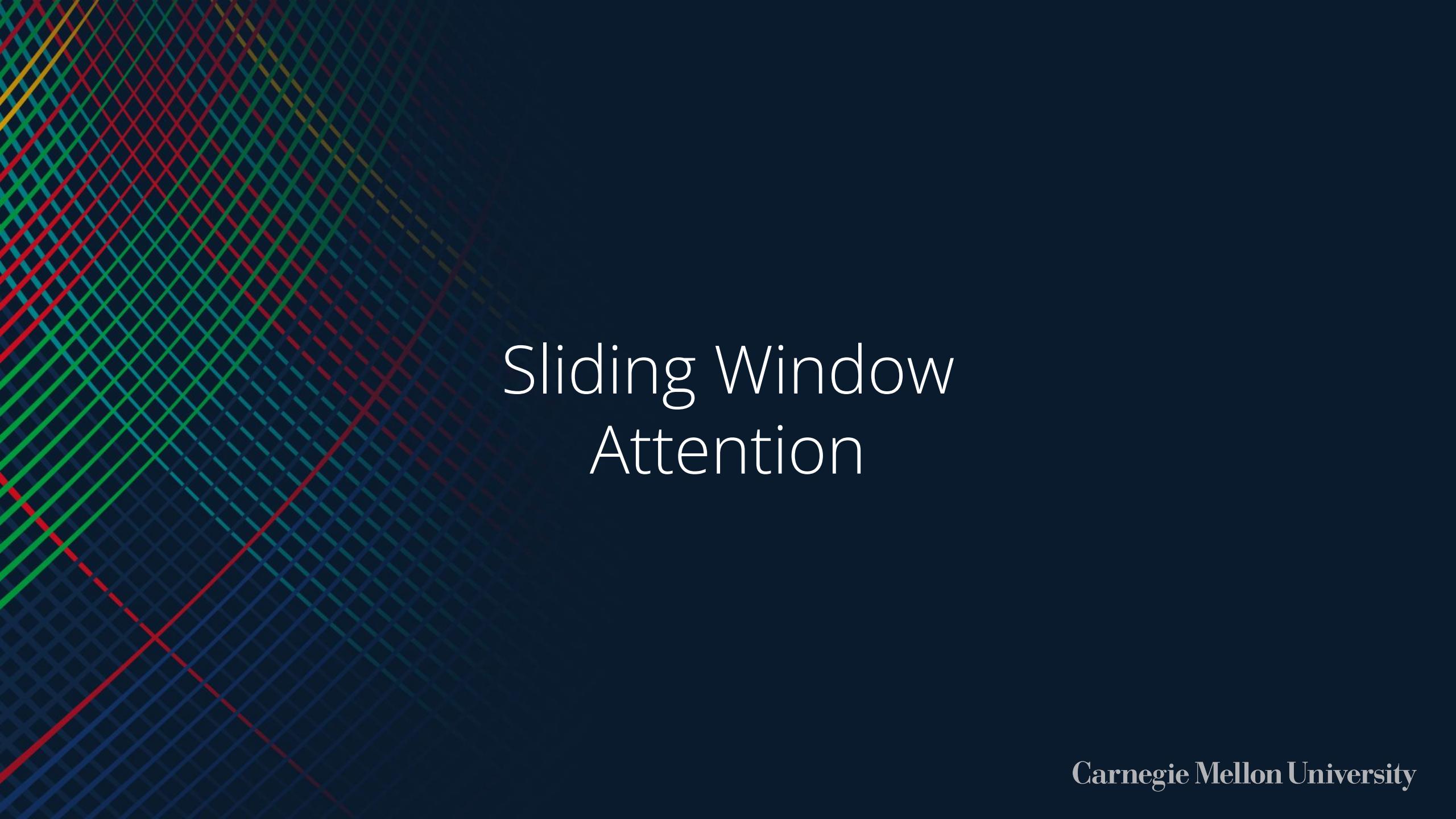
```
print(output.shape) # Output shape after convolution
```

einsum Example (CausalSelfAttention)

```
# calculate the attention scores with the query and key
# b = batch size, t = sequence length, h = # of heads, d = n_embd / # of heads
att = einsum(q, k, 'b h q d, b h k d -> b h q k') / scale
```

Summary Table

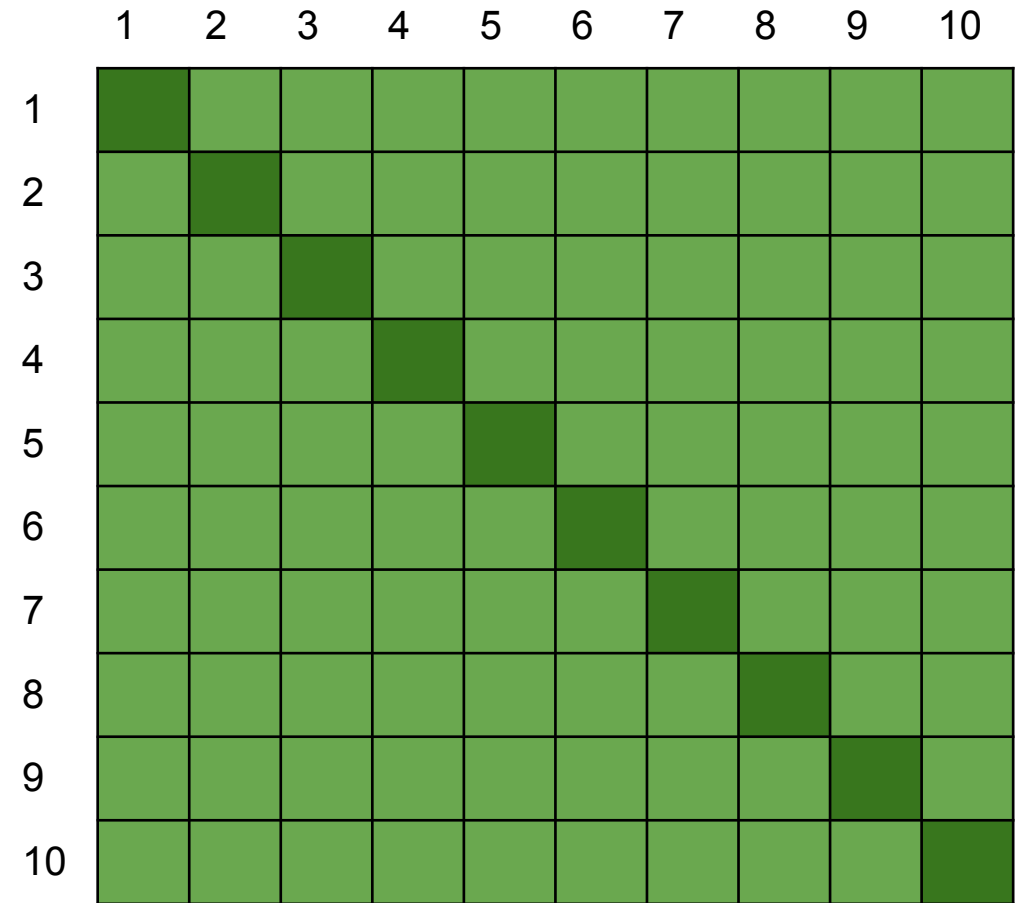
Operation	Equivalent NumPy	Einsum Expression
Sum	<code>np.sum(A, axis=0)</code>	<code>'ij -> j'</code>
Transpose	<code>A.T</code>	<code>'ij -> ji'</code>
Dot Product	<code>np.dot(a, b)</code>	<code>'i, i ->'</code>
Matrix-Vector	<code>np.matmul(A, x)</code>	<code>'ij, j -> i'</code>
Matrix Multiply	<code>np.matmul(A, B)</code>	<code>'ik, kj -> ij'</code>
Outer Product	<code>np.outer(a, b)</code>	<code>'i, j -> ij'</code>
Batch MatMul	<code>torch.bmm(A, B)</code>	<code>'bij, bjk -> bik'</code>



Sliding Window Attention

Full Self-Attention

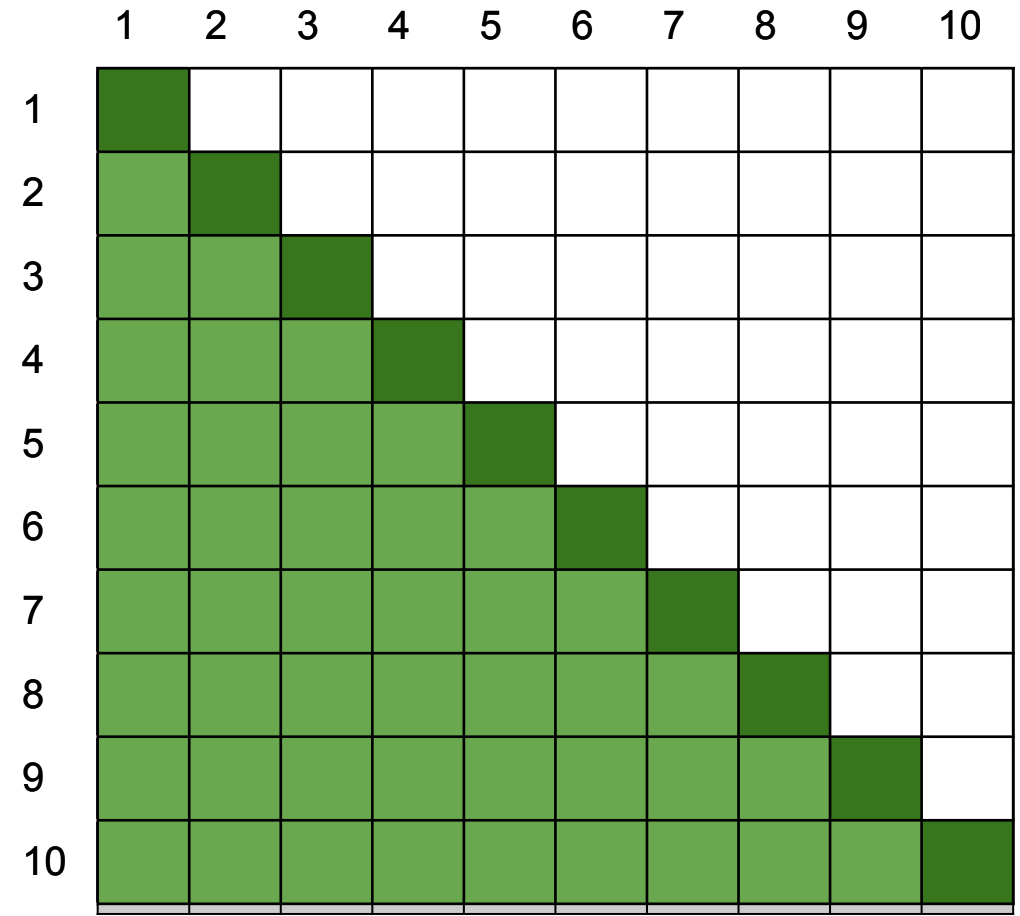
- Memory = $O(n^2)$
- Time Complexity = $O(n^2)$
- Infeasible (or very expensive) to process long sequences



10X10 Full Self - Attention matrix for sequence of length 10

Causal Self-Attention

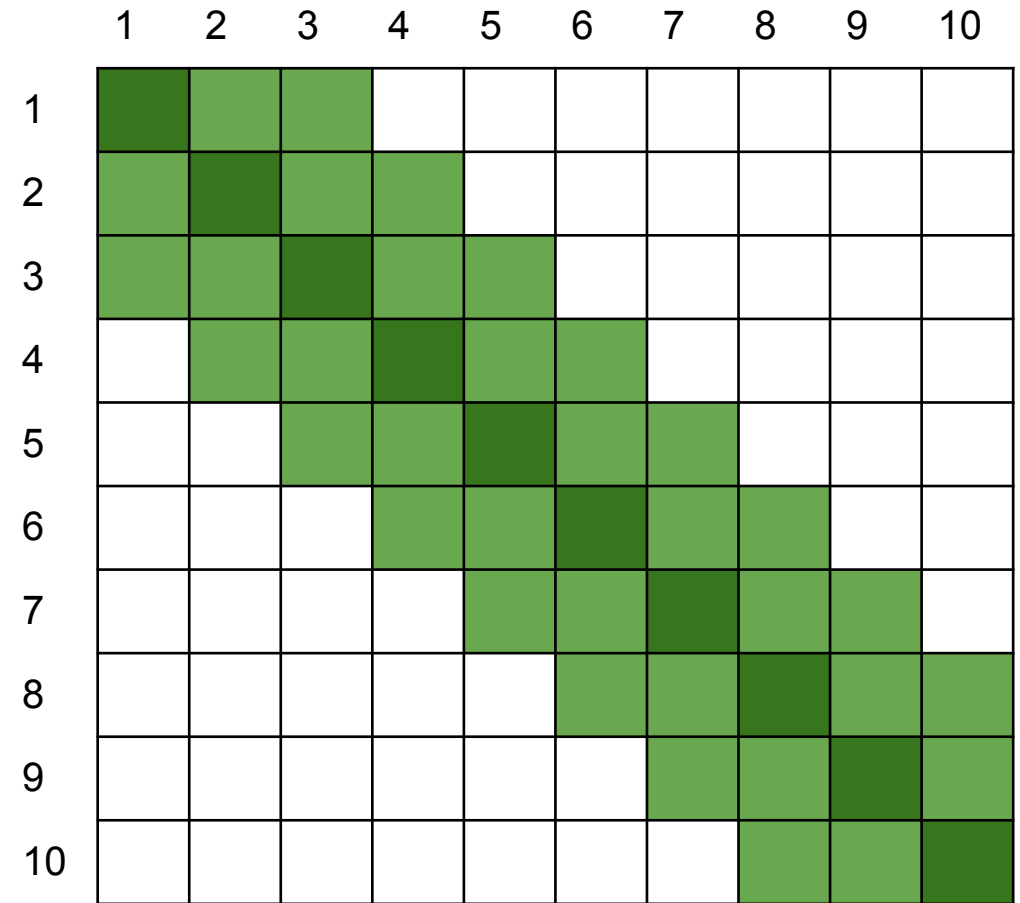
- Memory = $O(n^2)$
- Time Complexity = $O(n^2)$
- Infeasible (or very expensive) to process long sequences



10X10 Causal Self-Attention matrix for sequence of length 10

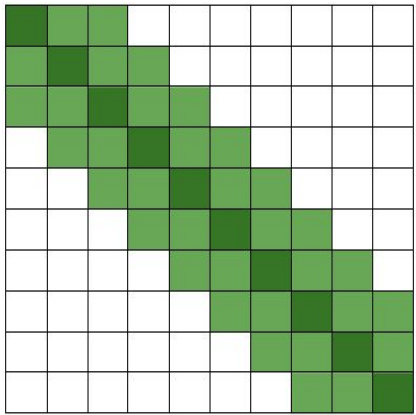
Sliding Window Attention

- Window Size = w
- Attend $w/2$ tokens on each side
- Time Complexity = $O(n \times w)$
 - Linear wrt seq_len (n) instead of quadratic!
- Memory = ?

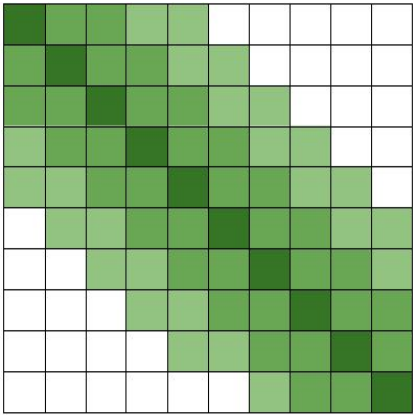


10X10 Sliding Window Attention matrix for sequence of length 10 and window size=4

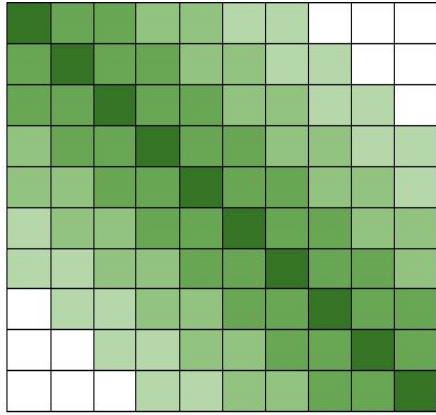
Sliding Window Attention



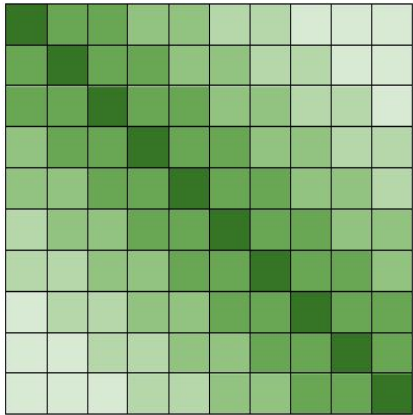
Layer1



Layer2



Layer3



Layer4

Attention Implementation

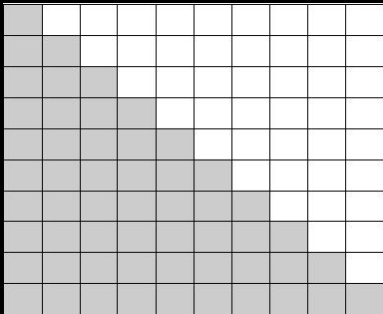
```
similarity = similarity.masked_fill(mask == 0, float("-1e20"))
attention = torch.softmax(similarity / (self.embed_size ** (1 / 2)), dim=3)
```

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where $Q \in \mathbb{R}^{h \times S \times d_k}$ and $K, V \in \mathbb{R}^{h \times L \times d_k}$

Causal Self-Attention

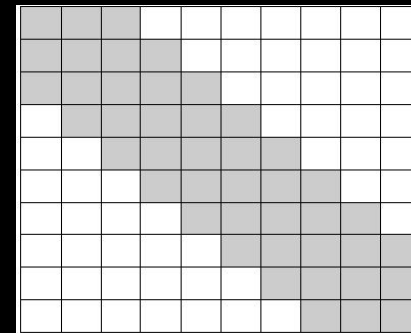
```
mask = torch.triu(torch.ones(T, T), diagonal=1).bool()
# If mask = True, we mask out that index
```



Memory = $O(n^2)$

Sliding Window Attention

```
mask = torch.ones(attn.size(), dtype=torch.bool)
for i in range(T):
    for j in range(max(0, i-w//2), min(T, i+w//2 + 1)):
        mask[:, :, i, j] = False
```



Memory = Still $O(n^2)$:(

But you
can do
better :)
See HW



Appendix

(look over this on your own)

Flags you'll find useful in the code

Feature	Example usage (pass after <code>python chargpt.py</code>)
Change model sequence length	<code>python chargpt.py --data.block_size=128</code>
Change directory where model is stored	<code>python chargpt.py --system.work_dir=out/new_chargpt</code>
Change number of query heads	<code>python chargpt.py --model.n_query_head=6</code>
Change number of key-value heads	<code>python chargpt.py --model.n_kv_head=3</code>
Load a model trained in a previous run	<code>python chargpt.py --model.pretrained_folder=out/new_chargpt</code>
Enable RoPE embeddings	<code>python chargpt.py --model.rope=True</code>
Number of iterations to run the model for	<code>python chargpt.py --model.max_iters=201</code>
Change device type (useful for debugging)	<code>python chargpt.py --trainer.device=cpu</code>

Example: Converting Math to PyTorch Code

Sinusoidal Position Embeddings ($n == 10000$):

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k, 2i + 1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

k = token's position index in sequence, $k \in [0, \text{seq_len})$

i = dim index in embedding vector, $i \in [0, d//2)$

d = dimensionality of embedding



Converting Math to PyTorch Code

```
1 def get_positional_embedding(seq_len, d, n=10000):
2     assert(d % 2 == 0)
3
4     # create position indices k = 0, 1, ..., seq_len - 1
5     positions = torch.arange(0, seq_len).unsqueeze_(1) # (seq_len, 1)
6
7     # initialize the embeddings as a zeros matrix
8     embeddings = torch.zeros(seq_len, d) # (seq_len, d)
9
10    # compute the denominators in the formula = 10000^(2i/d)
11    denominators = torch.pow(n, 2*torch.arange(0, d//2)/d) # (d//2)
12
13    # apply sine to every even position = sin(k/10000^(2i/d))
14    embeddings[:, 0::2] = torch.sin(positions/denominators)
15
16    # apply cosine to every odd position = cos(k/10000^(2i/d))
17    embeddings[:, 1::2] = torch.cos(positions/denominators)
18
19    return embeddings
```

Example: Converting Math to PyTorch Code



Converting Math to PyTorch Code

Notice embeddings is fixed for a given seq_len, d, n, and will remain constant throughout training
→ **so we should cache** it to prevent recomputing it every single time

```
1 def get_positional_embedding(seq_len, d, n=10000):
2     assert(d % 2 == 0)
3
4     # create position indices k = 0, 1, ..., seq_len - 1
5     positions = torch.arange(0, seq_len).unsqueeze_(1) # (seq_len, 1)
6
7     # initialize the embeddings as a zeros matrix
8     embeddings = torch.zeros(seq_len, d) # (seq_len, d)
9
10    # compute the denominators in the formula = 10000^(2i/d)
11    denominators = torch.pow(n, 2*torch.arange(0, d//2)/d) # (d//2)
12
13    # apply sine to every even position = sin(k/10000^(2i/d))
14    embeddings[:, 0::2] = torch.sin(positions/denominators)
15
16    # apply cosine to every odd position = cos(k/10000^(2i/d))
17    embeddings[:, 1::2] = torch.cos(positions/denominators)
18
19    return embeddings
```

PyTorch functions

- **`torch.cat(tensors, dim=0, *, out=None)`**: concatenates the given sequence of tensors in the given dimension. All tensors must either have the same shape (except in the concatenating dimension) or be empty.

```
[>>> x = torch.randn(2,3)
[>>> x
tensor([[ -0.1592, -0.9748, -1.0052],
        [-0.1304, -0.4735,  0.6236]])
[>>> torch.cat((x, x, x), 0)
tensor([[ -0.1592, -0.9748, -1.0052],
        [-0.1304, -0.4735,  0.6236],
        [-0.1592, -0.9748, -1.0052],
        [-0.1304, -0.4735,  0.6236],
        [-0.1592, -0.9748, -1.0052],
        [-0.1304, -0.4735,  0.6236]])
[>>> torch.cat((x, x, x), 1)
tensor([[ -0.1592, -0.9748, -1.0052, -0.1592, -0.9748, -1.0052, -0.1592, -0.9748,
         -1.0052],
        [-0.1304, -0.4735,  0.6236, -0.1304, -0.4735,  0.6236, -0.1304, -0.4735,
         0.6236]])
```

PyTorch functions

- **`torch.unsqueeze(tensor, dim)`**: returns a new tensor with a dimension of size 1 inserted at `dim`. `dim` should be in `[-input.dim()-1, input.dim+1)`

[torch.unsqueeze — PyTorch 2.1 documentation](#)

```
[>>> x = torch.arange(1, 5)
[>>> x.shape
torch.Size([4])
[>>> y = torch.unsqueeze(x, dim = 0)
[>>> y.shape
torch.Size([1, 4])
[>>> z = torch.unsqueeze(x, dim = 1)
[>>> z.shape
torch.Size([4, 1])
```