

Deep learning

Unpacking Transformers, LLMs and image generation

Session 4

Syllabus

Session	Lecture	TP
1	Intro to DL Gradient descent and backprop	Intro to micrograd (*) fit the x^2 function
2	DL fundamentals I <ul style="list-style-type: none">• Backprop• Loss functions• Neural Probabilistic Language Model (Bengio 2003)	Bigram model and MLP for next-character prediction (*) extend to tri-gram
3	DL fundamentals II <ul style="list-style-type: none">• Activation function• Regularization• Initialization• Residual networks• Normalization Recurrent Neural Networks	Backprop ninja MLP in pytorch (*) add batchnorm to TP2
4	Attention and Transformers	GPT from scratch
5	DL for computer vision: convnets, <u>unets</u>	Convnets for CIFAR-10
6	VAE & Diffusion models	1D diffusion model 2D diffusion model (*) train on GPU Quiz

Attention and Transformers

The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and the service.

Attention and Transformers

Query vector q_1

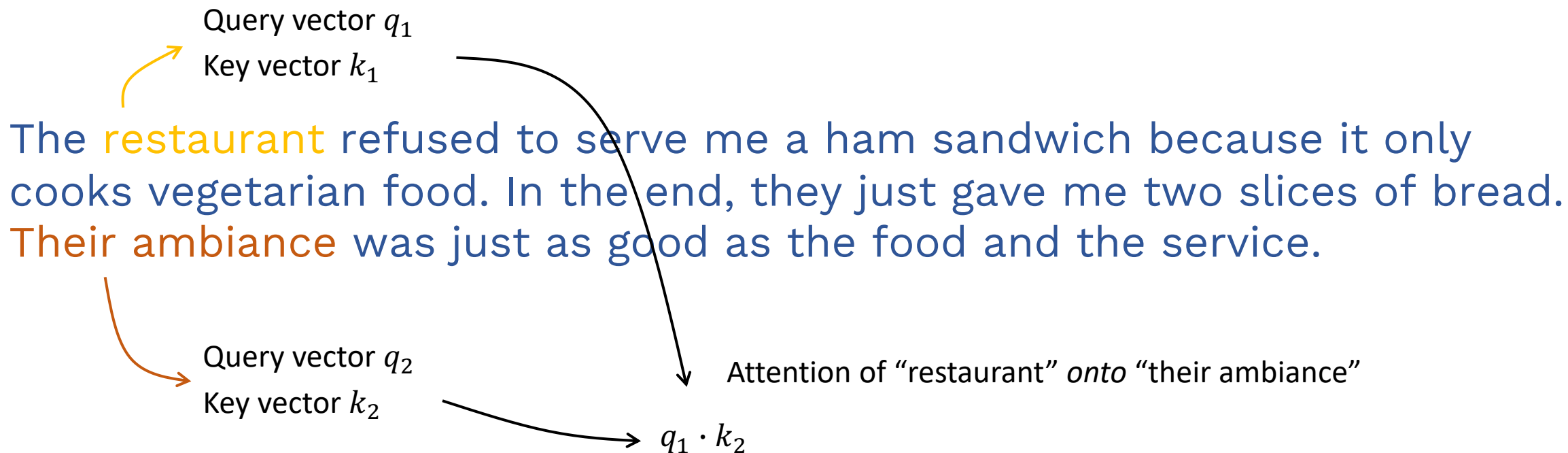
Key vector k_1

The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and the service.

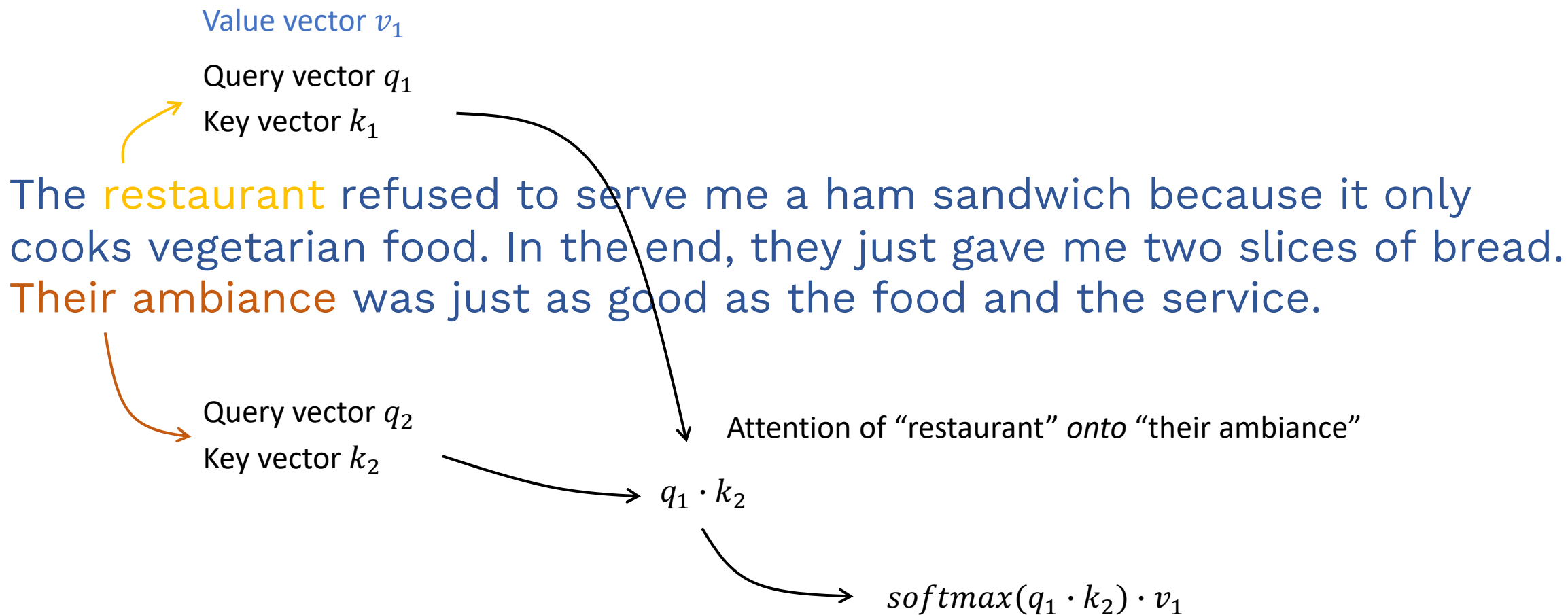
Query vector q_2

Key vector k_2

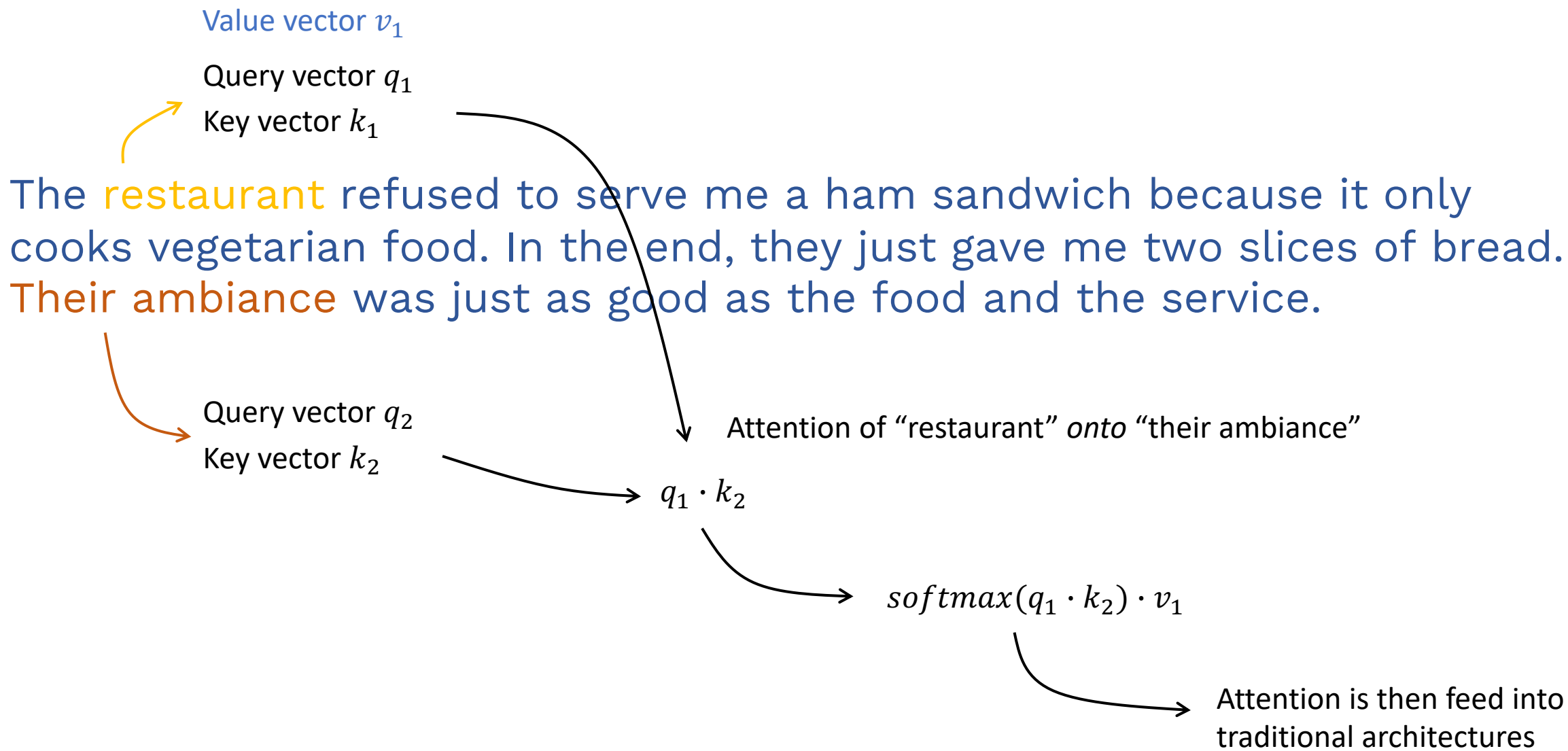
Attention and Transformers



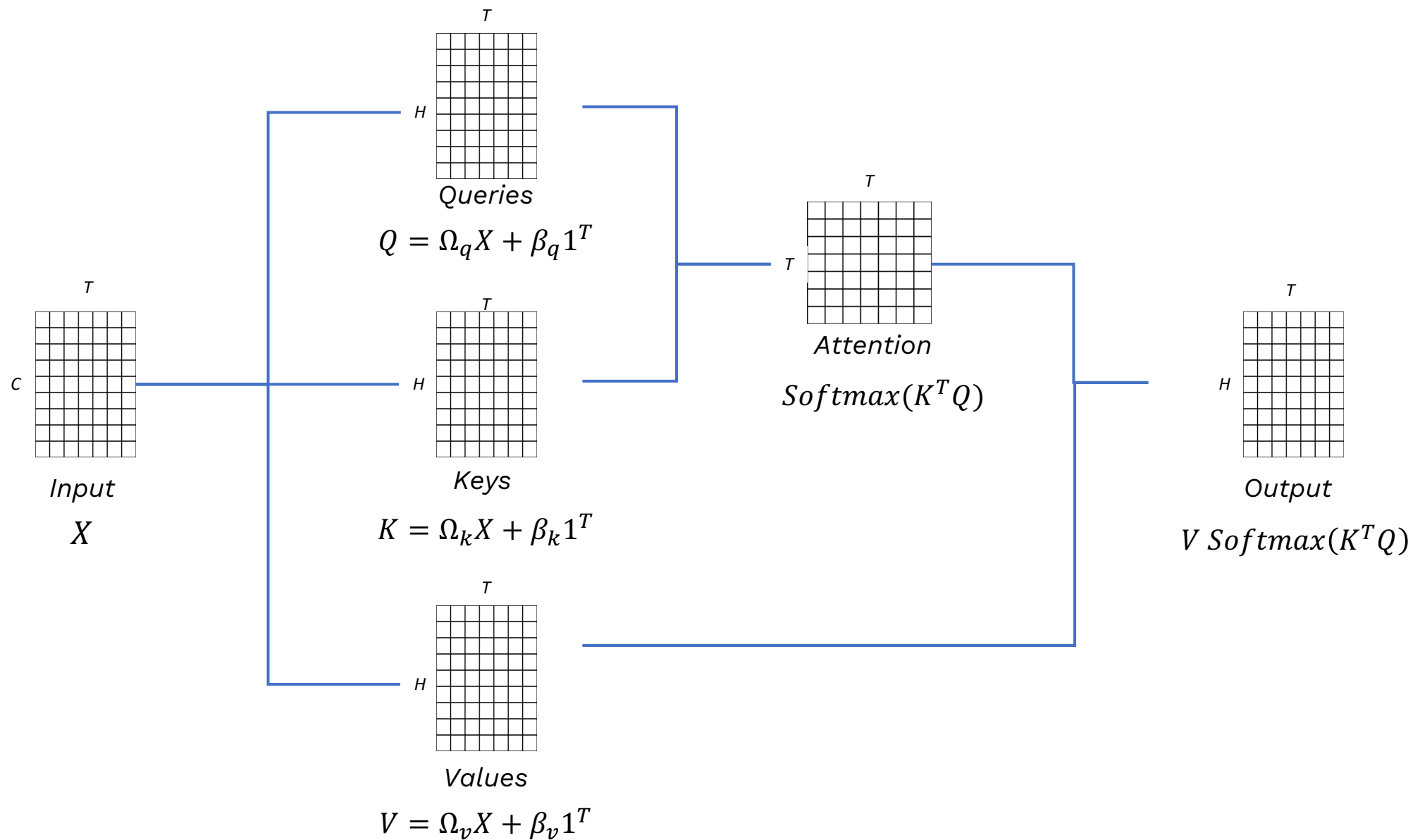
Attention and Transformers



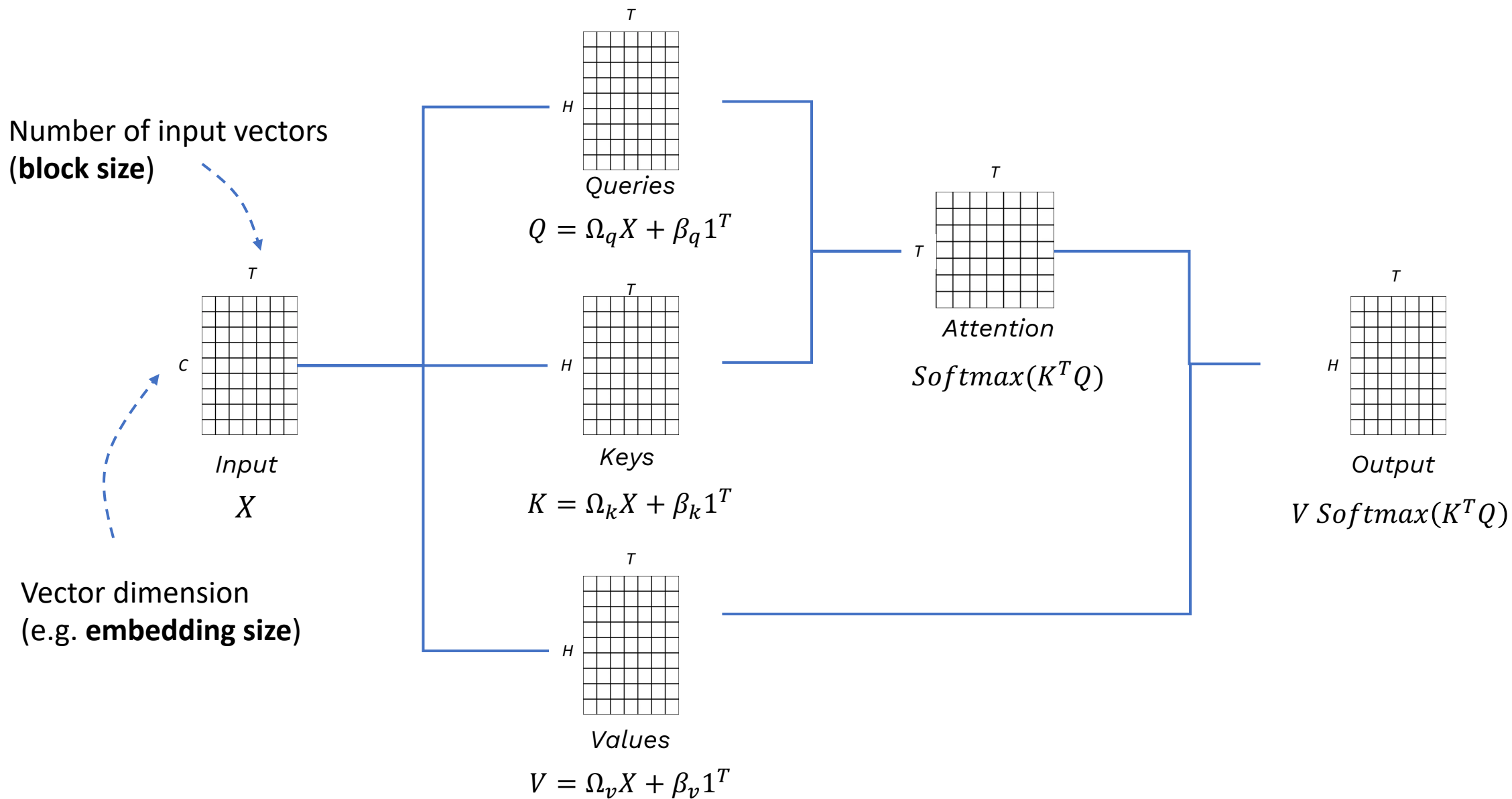
Attention and Transformers



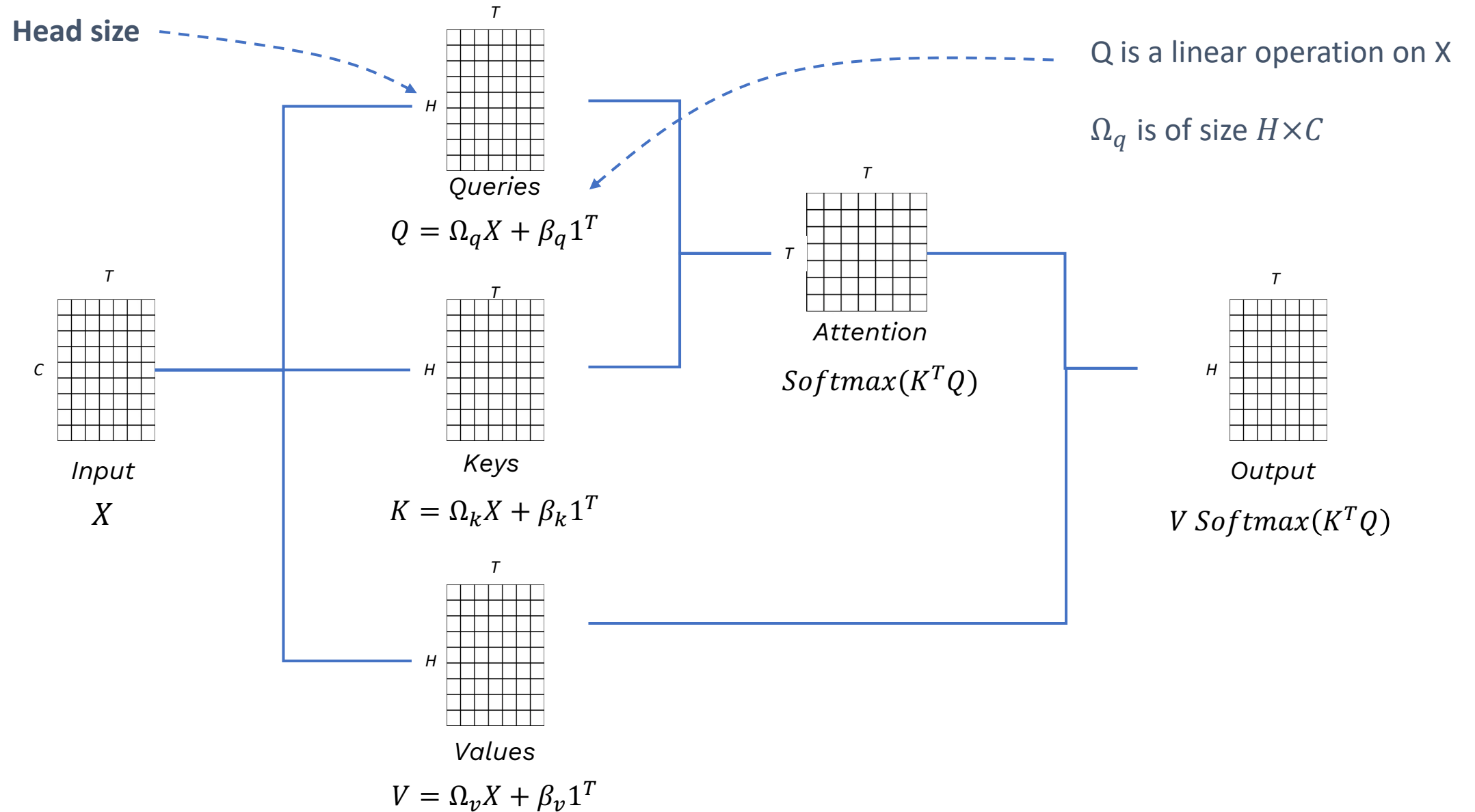
Attention and Transformers



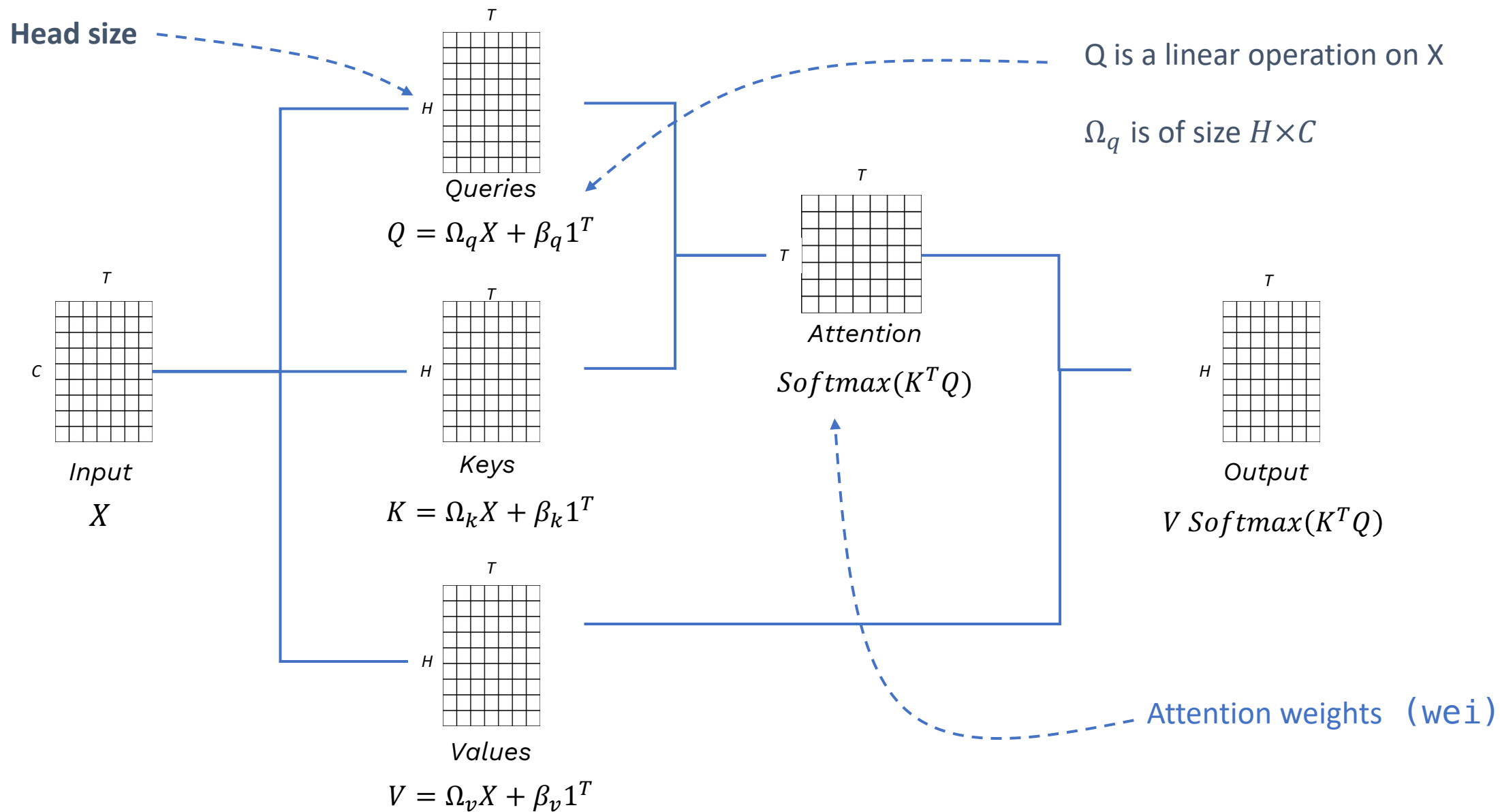
Attention and Transformers



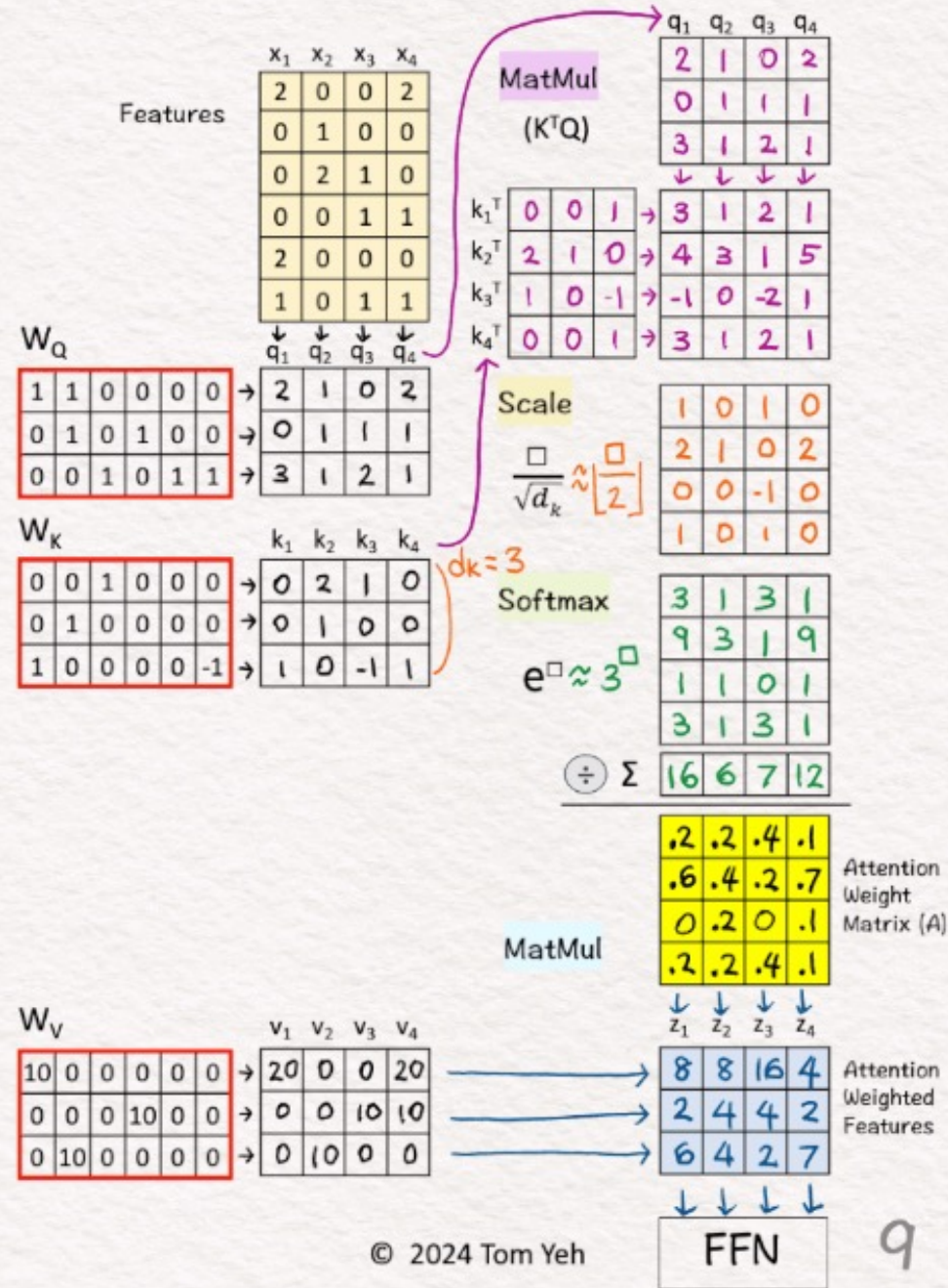
Attention and Transformers



Attention and Transformers



Self Attention



Attention and Transformers

Scaled Dot-Product Self-Attention

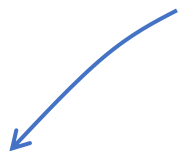
$$Sa[X] = V \cdot Softmax \left[\frac{K^T Q}{\sqrt{D_q}} \right]$$

Attention and Transformers

Scaled Dot-Product Self-Attention

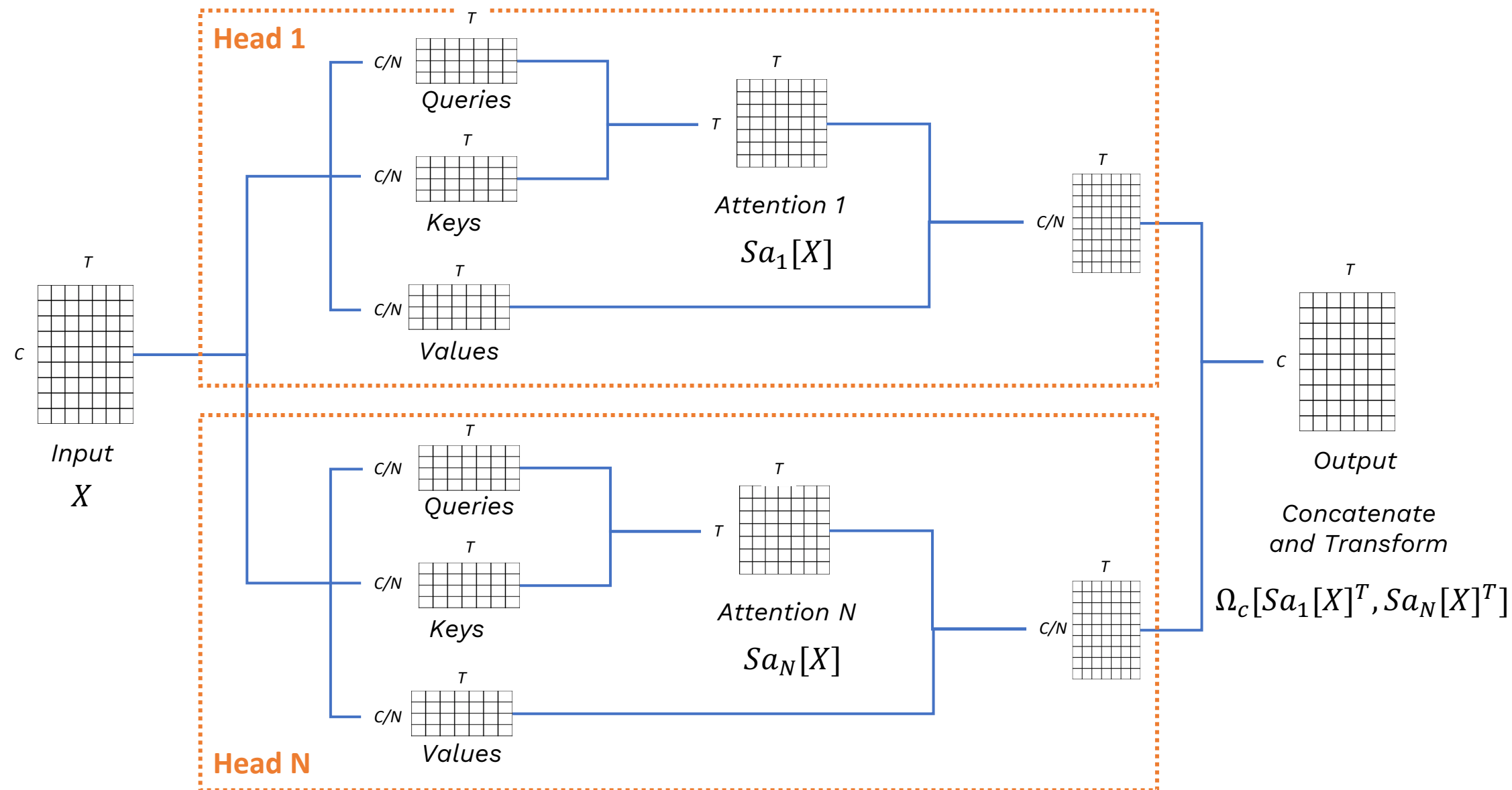
$$Sa[X] = V \cdot Softmax \left[\frac{K^T Q}{\sqrt{D_q}} \right]$$

Quadratic in X!

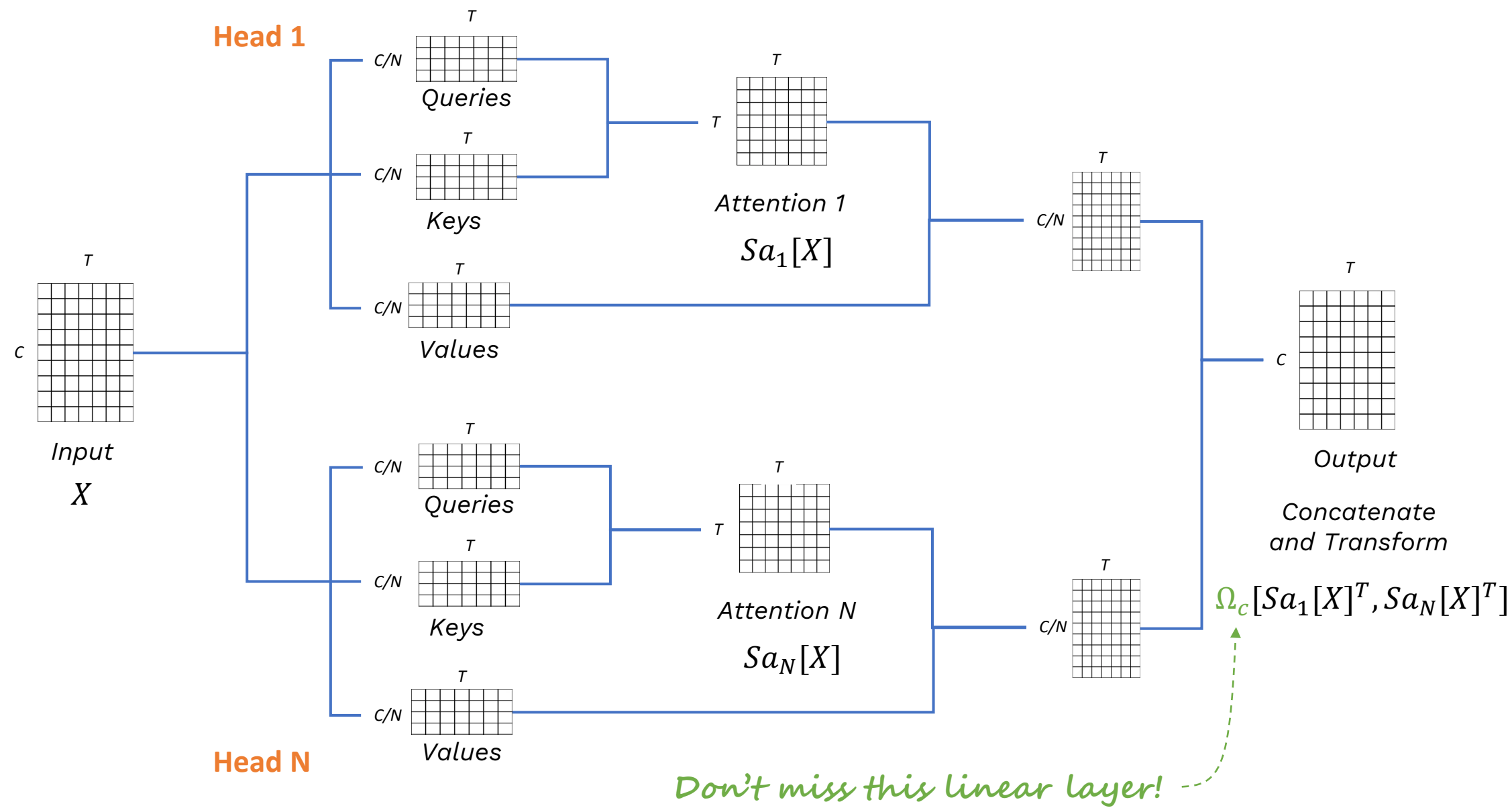


$$Sa[X] = V \cdot Softmax \left[\frac{X^T \Omega_K^T \Omega_Q X}{\sqrt{D_q}} \right]$$

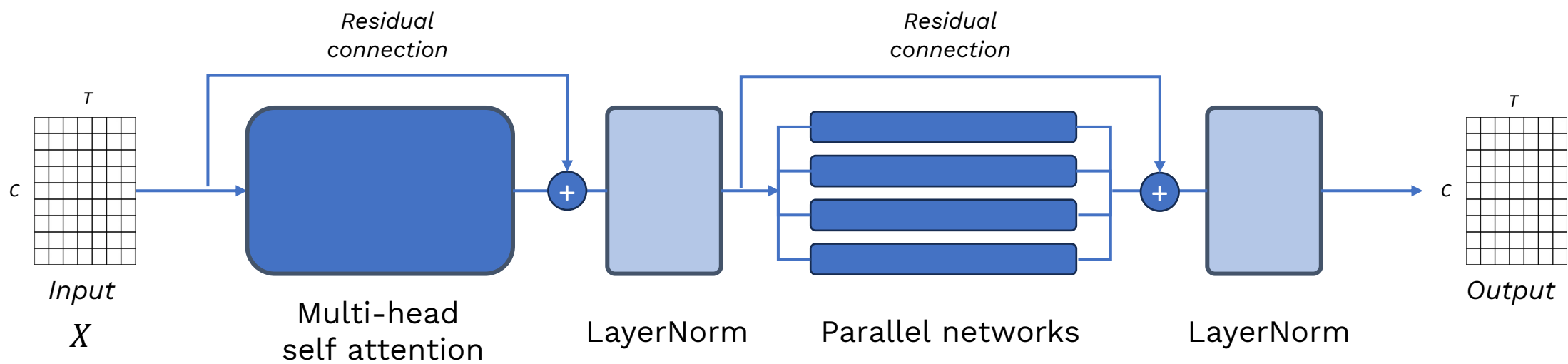
Multi-head attention (for N heads)



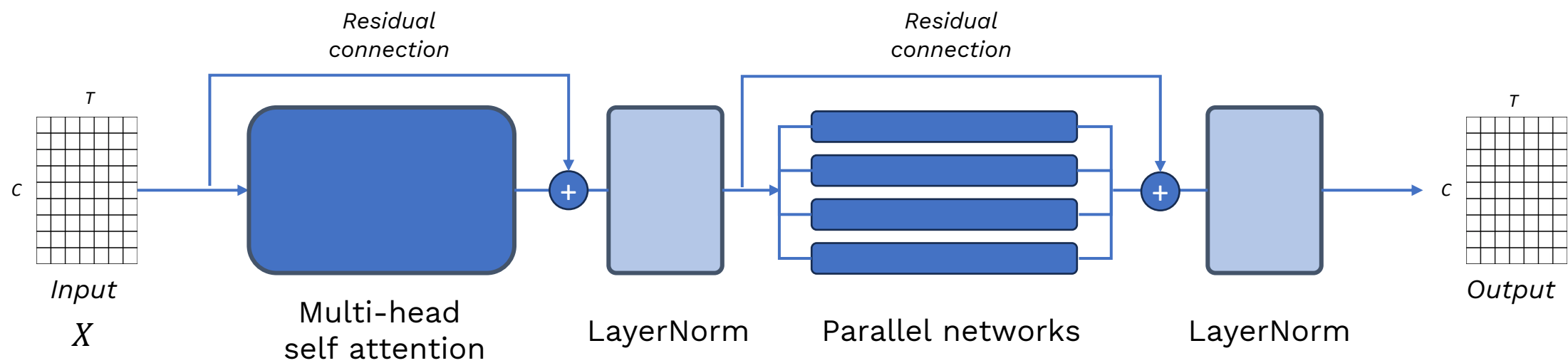
Multi-head attention (for N heads)



Attention and Transformers

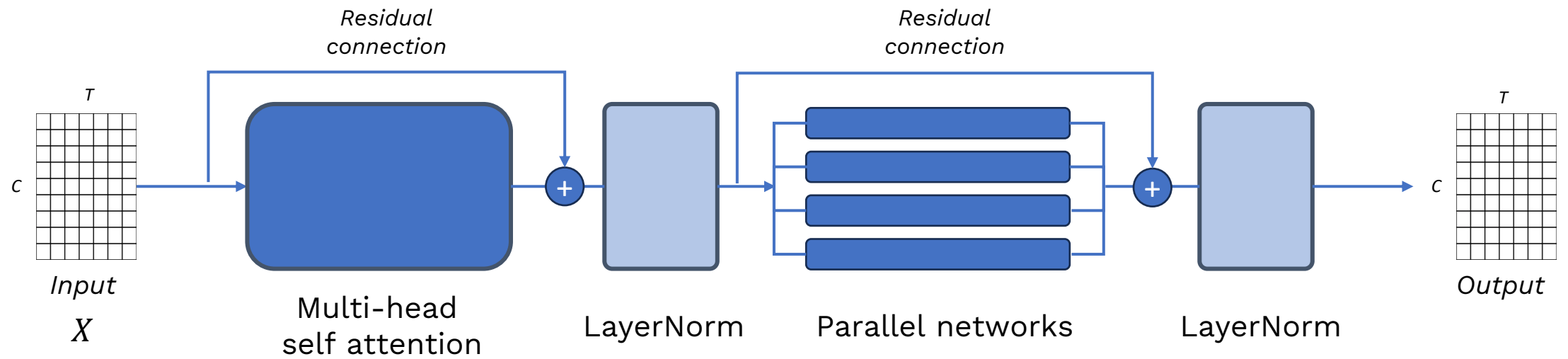


Attention and Transformers



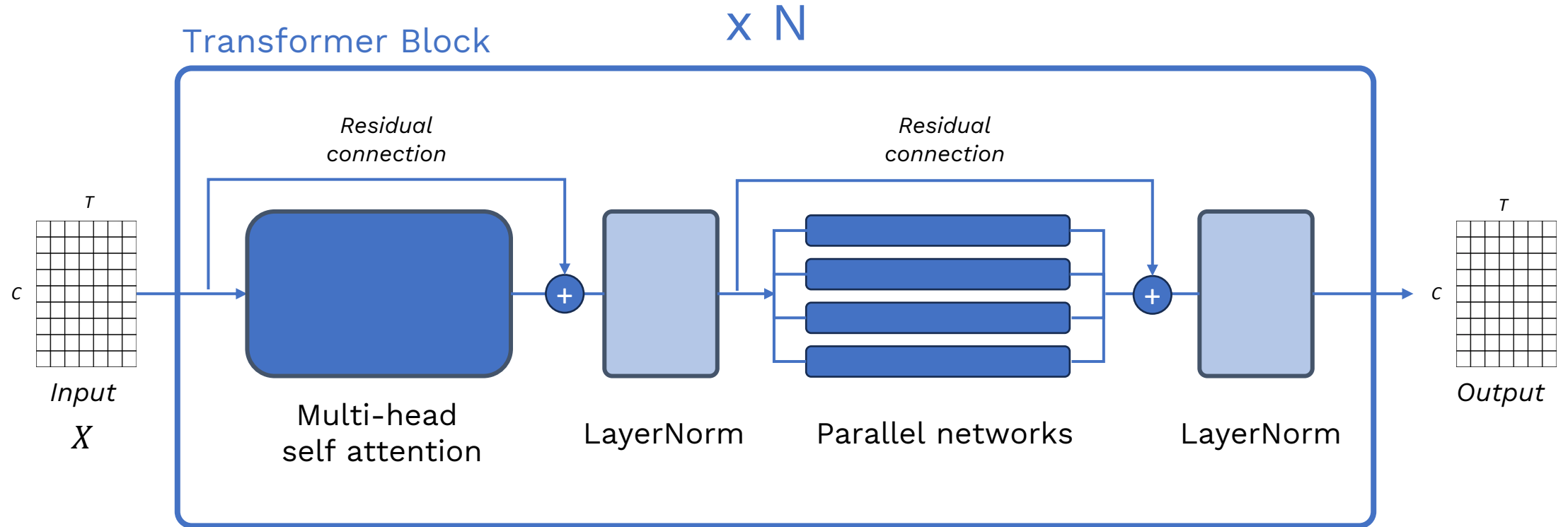
The same network is applied to all the vectors in parallel

Attention and Transformers



This is a feedforward with Relu and a 4X inner layer

Attention and Transformers



Tokenizing the input

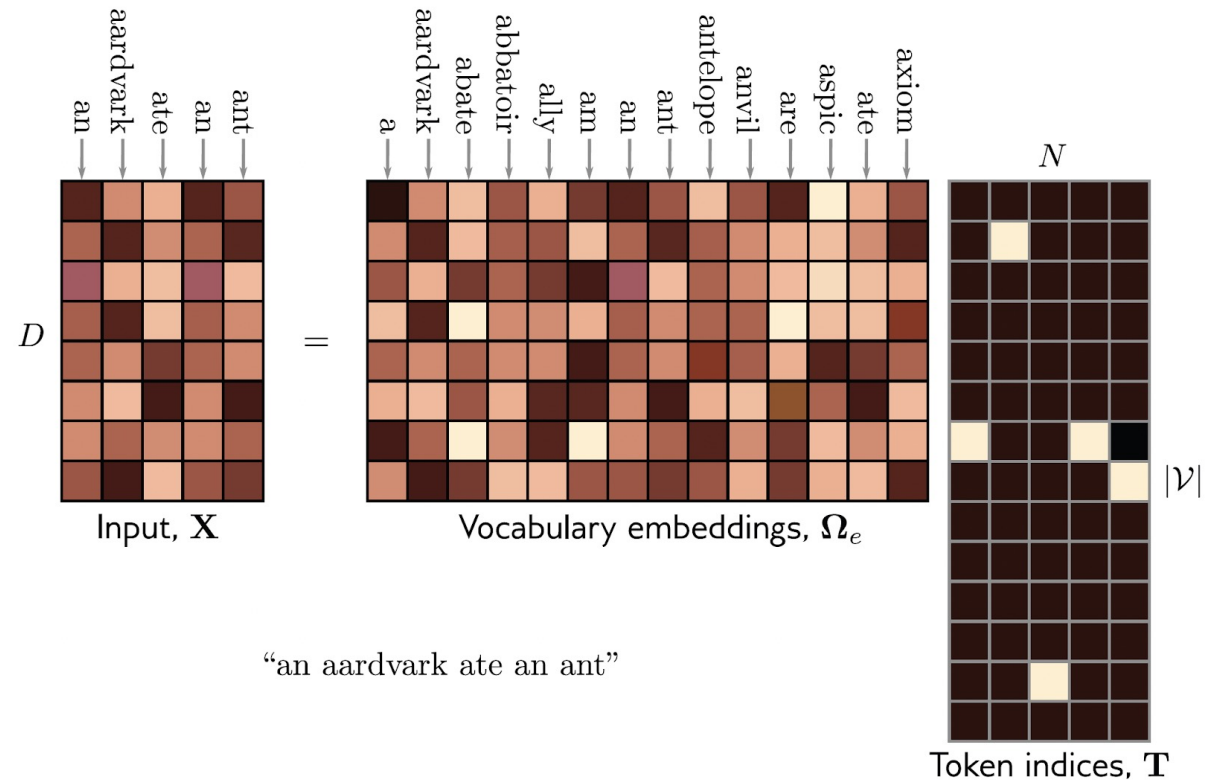
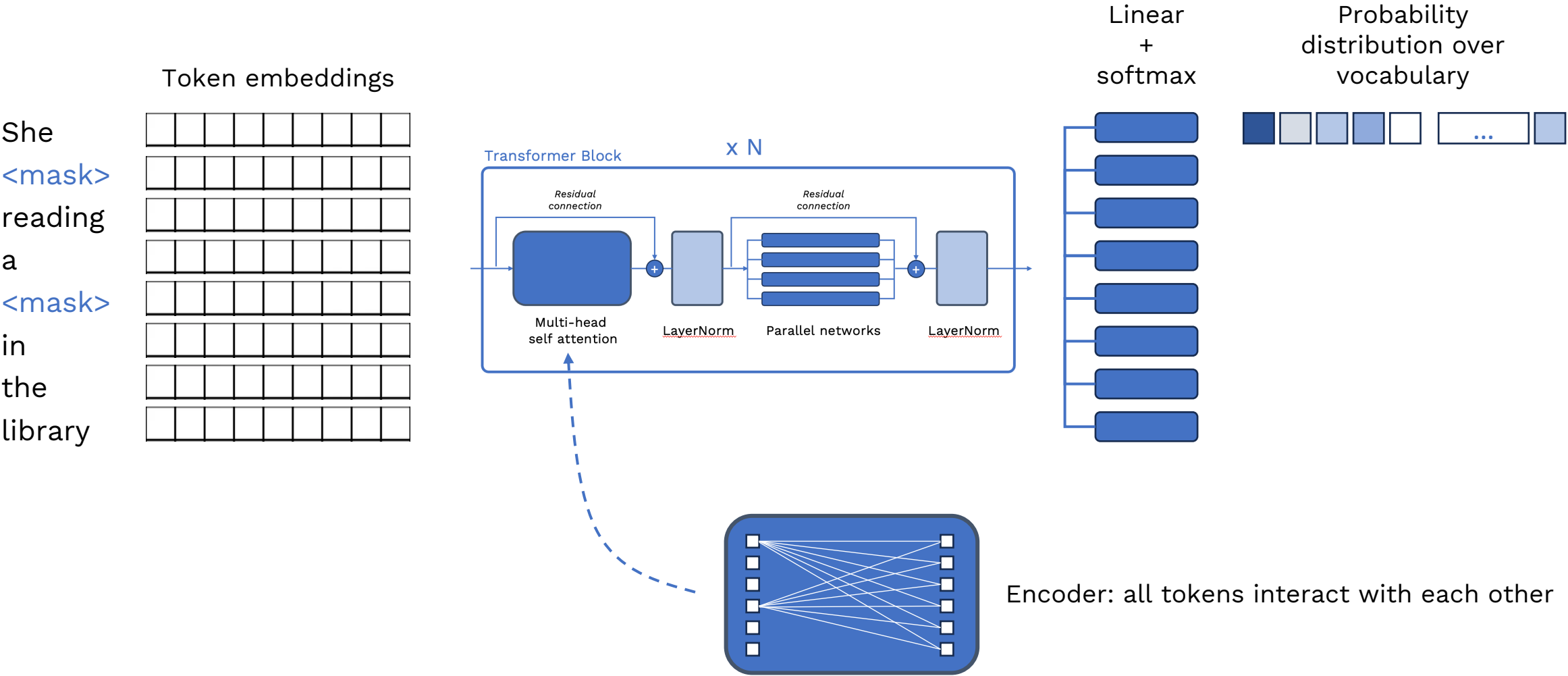


Figure 12.9 The input embedding matrix $\mathbf{X} \in \mathbb{R}^{D \times N}$ contains N embeddings of length D and is created by multiplying a matrix $\mathbf{\Omega}_e$ containing the embeddings for the entire vocabulary with a matrix containing one-hot vectors in its columns that correspond to the word or sub-word indices. The vocabulary matrix $\mathbf{\Omega}_e$ is considered a parameter of the model and is learned along with the other parameters. Note that the two embeddings for the word **an** in \mathbf{X} are the same.

Pretraining for BERT-like encoder

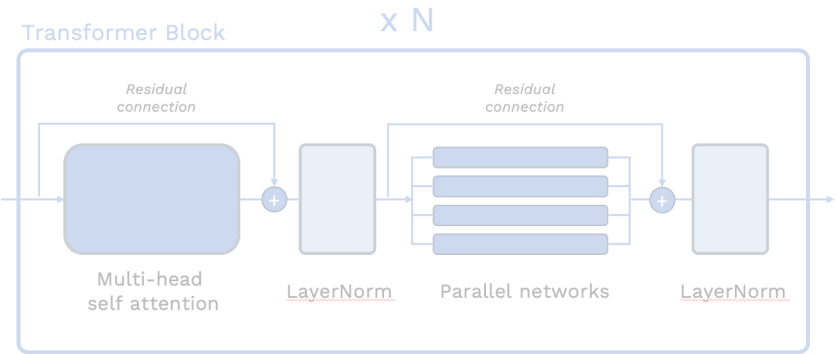
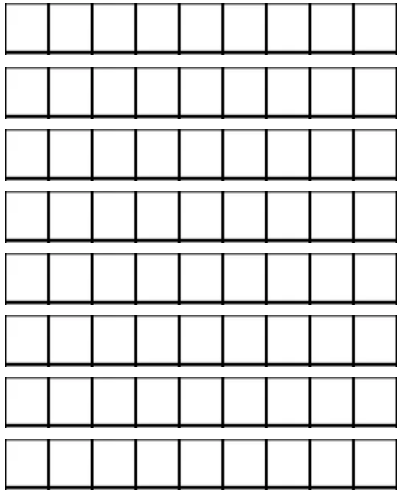


Fine-tuning to specific tasks: review prediction

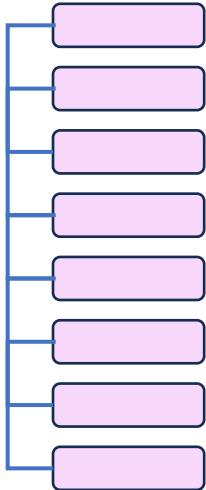
<mask>

The
soup
had
a
terrible
taste
and

Token embeddings



Linear
+
softmax



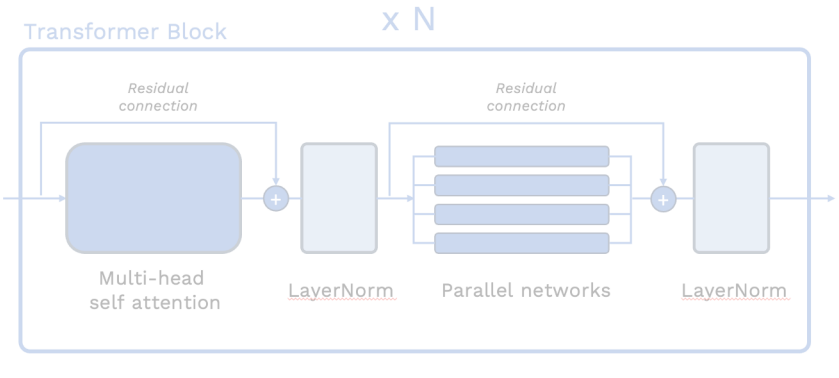
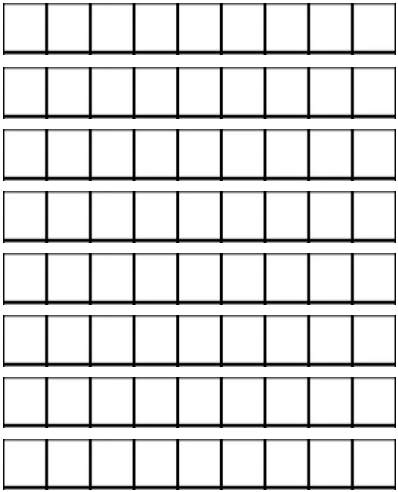
Probability of a
positive review



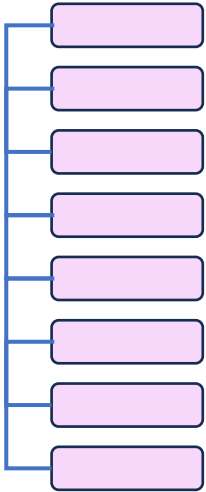
Fine-tuning to specific tasks: text classification

<mask>
We
arrived
in
Venice
with
Jennie
on

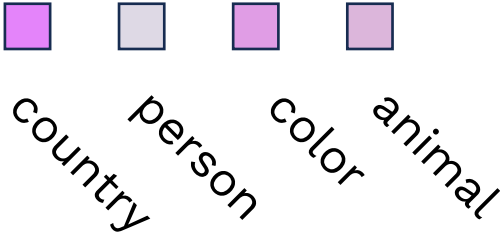
Token embeddings



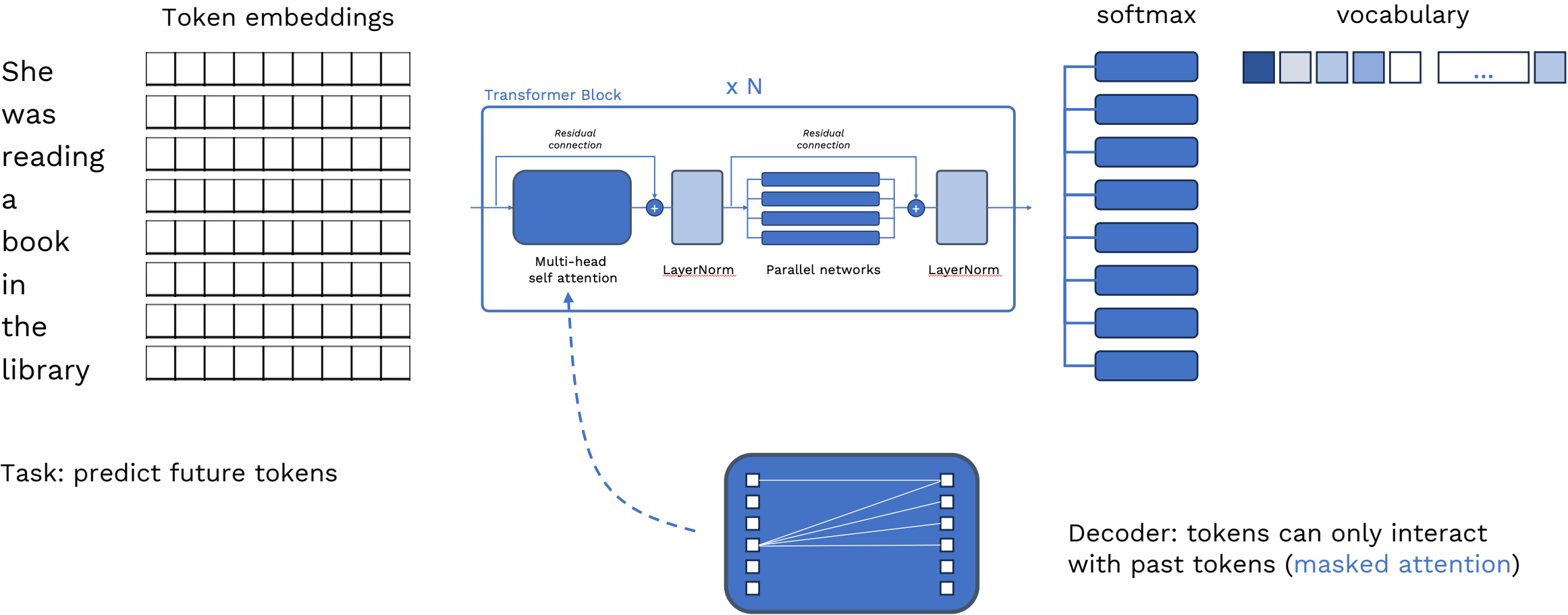
Linear + softmax



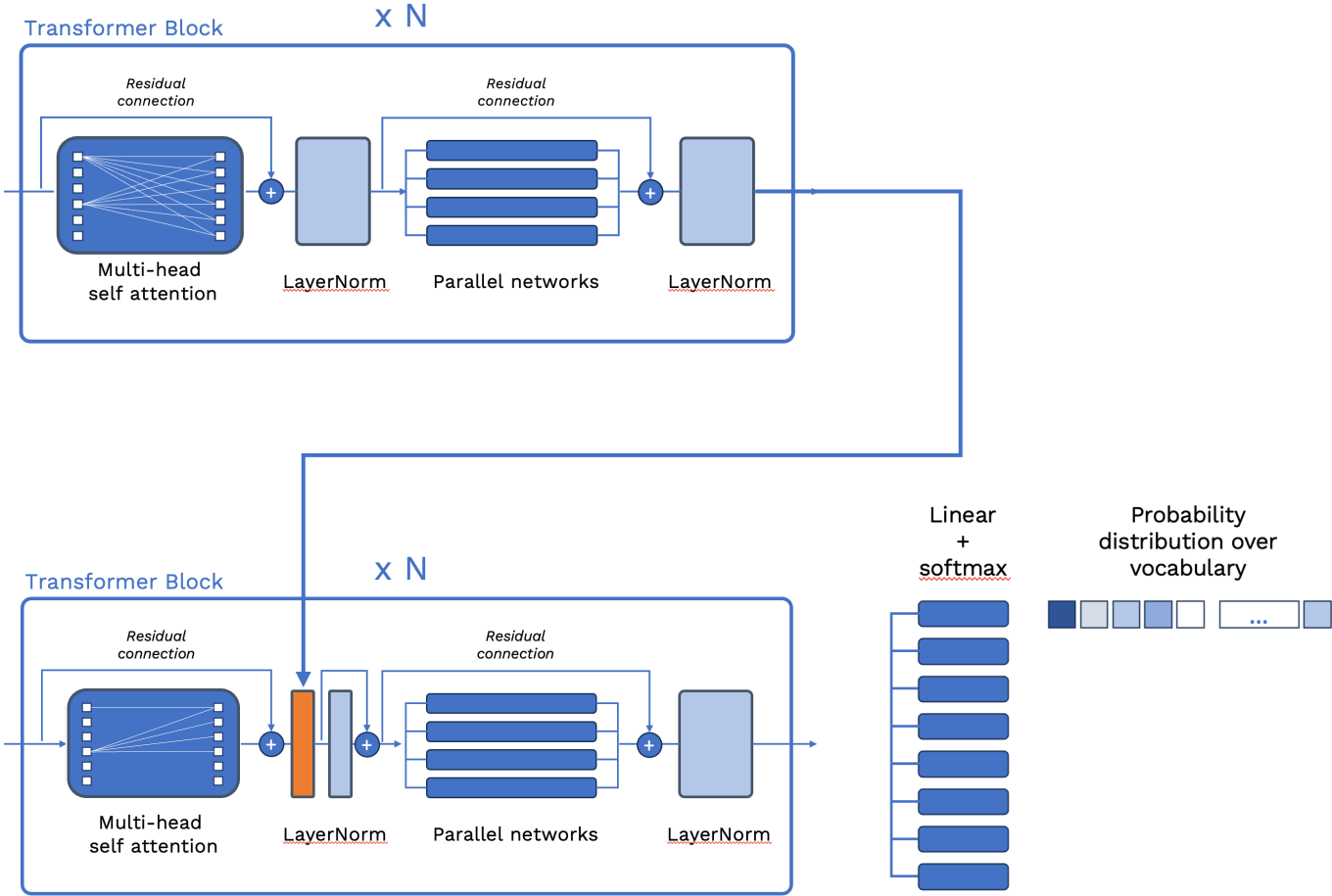
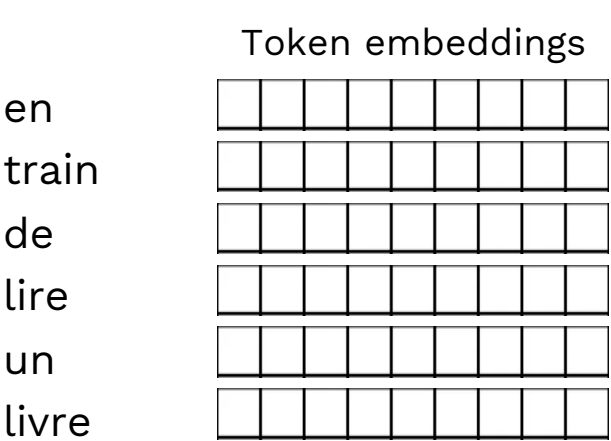
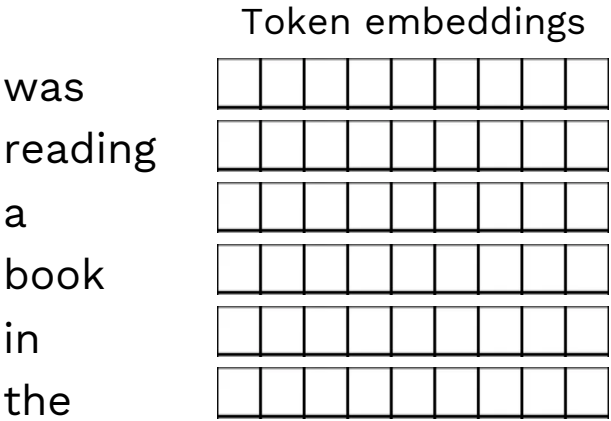
Distribution over classes



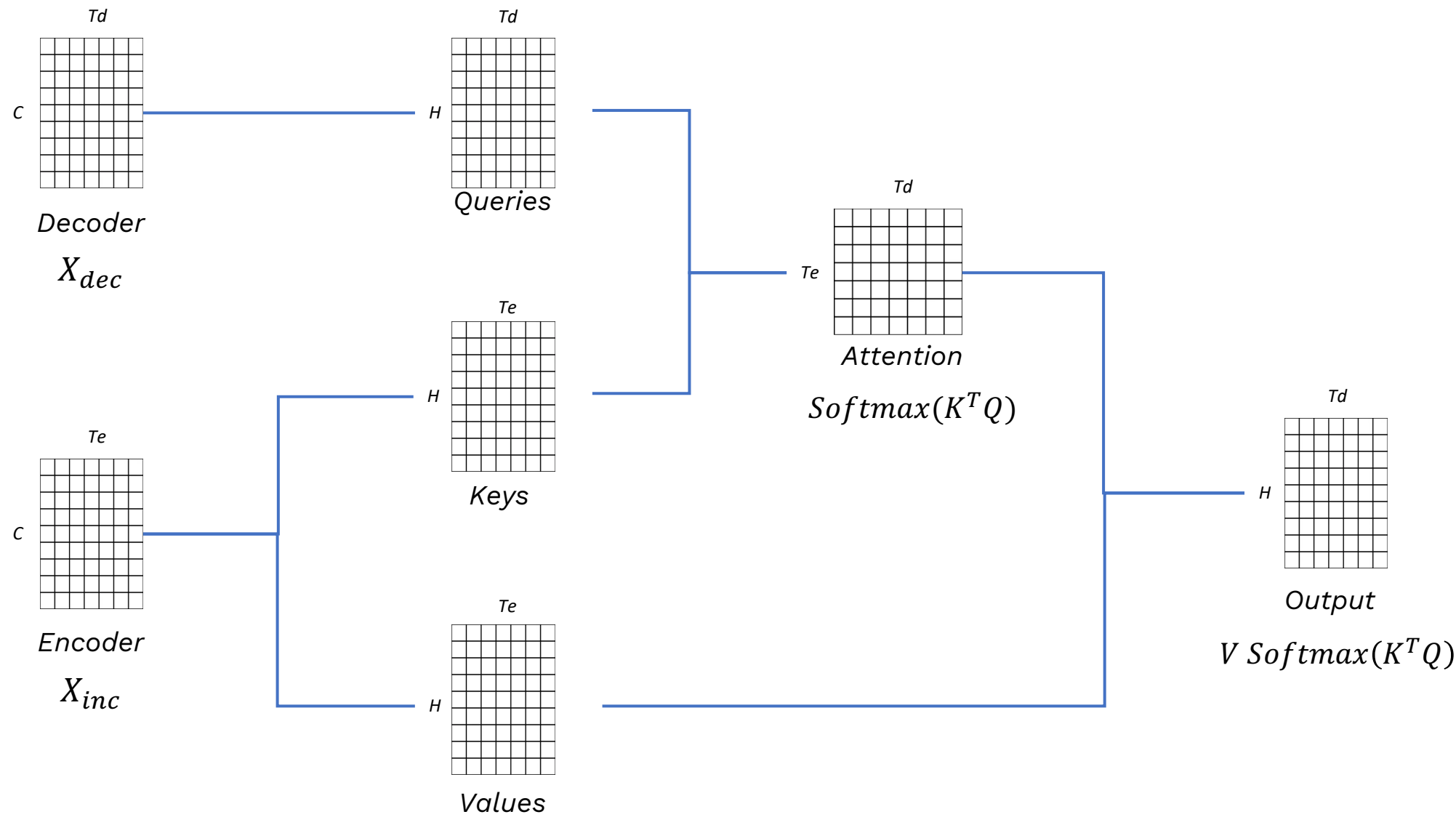
Pretraining for GPT-like decoder



Encoder-decoder architecture for translation with cross-attention



Cross-attention



The original Transformer architecture

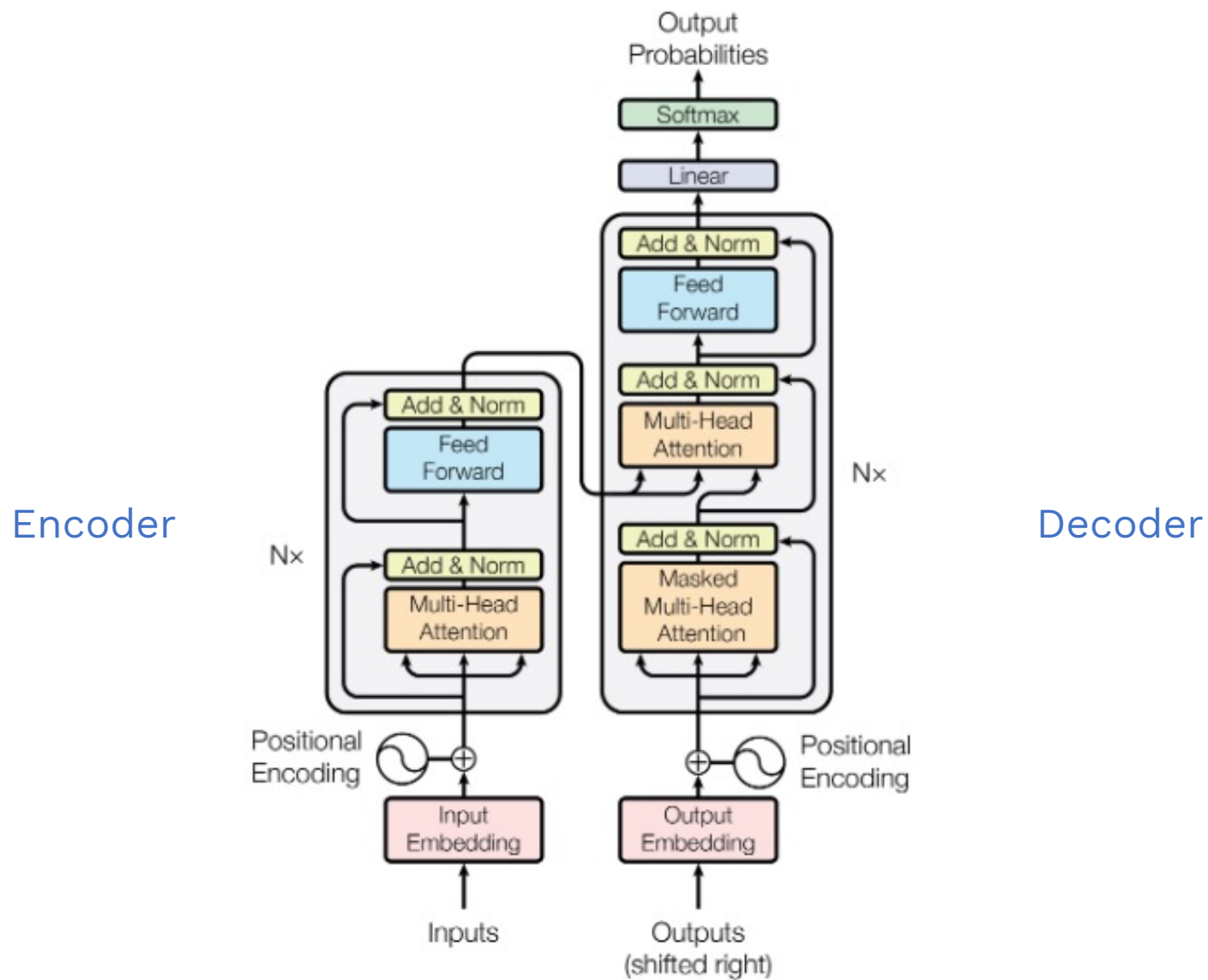
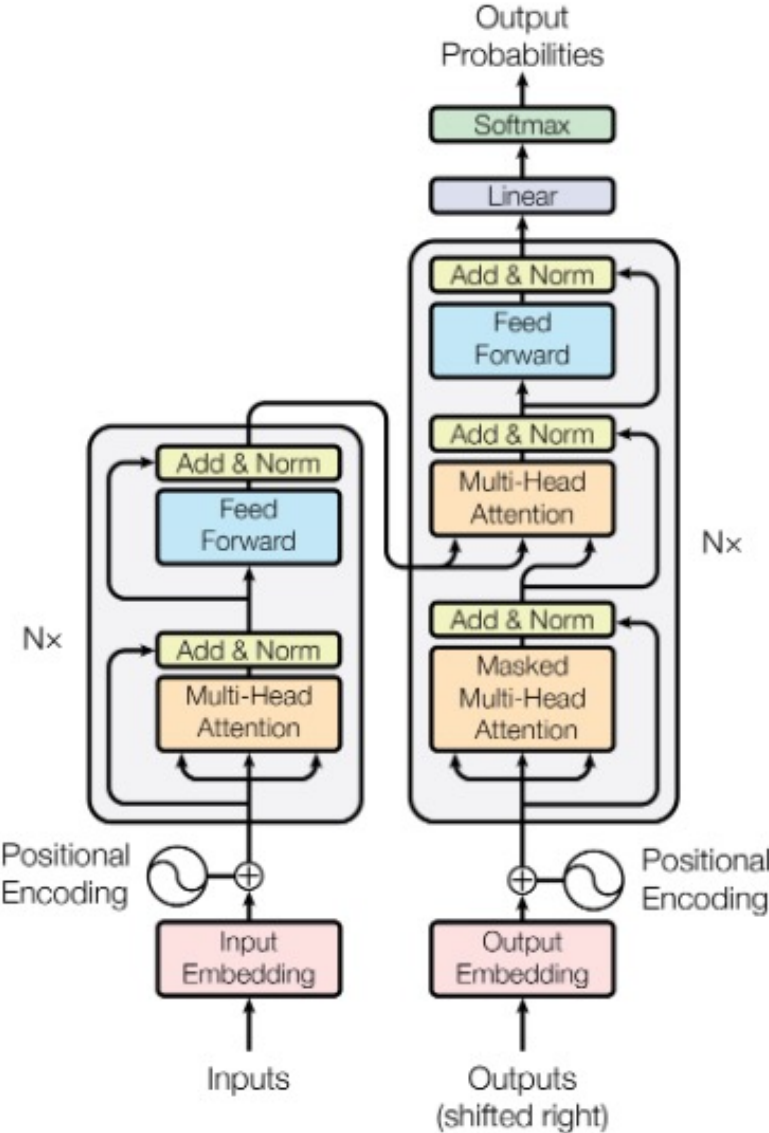


Figure 1: The Transformer - model architecture.

Positional encoding

Encodes the relative position of tokens in the block



Vocabulary embedding table

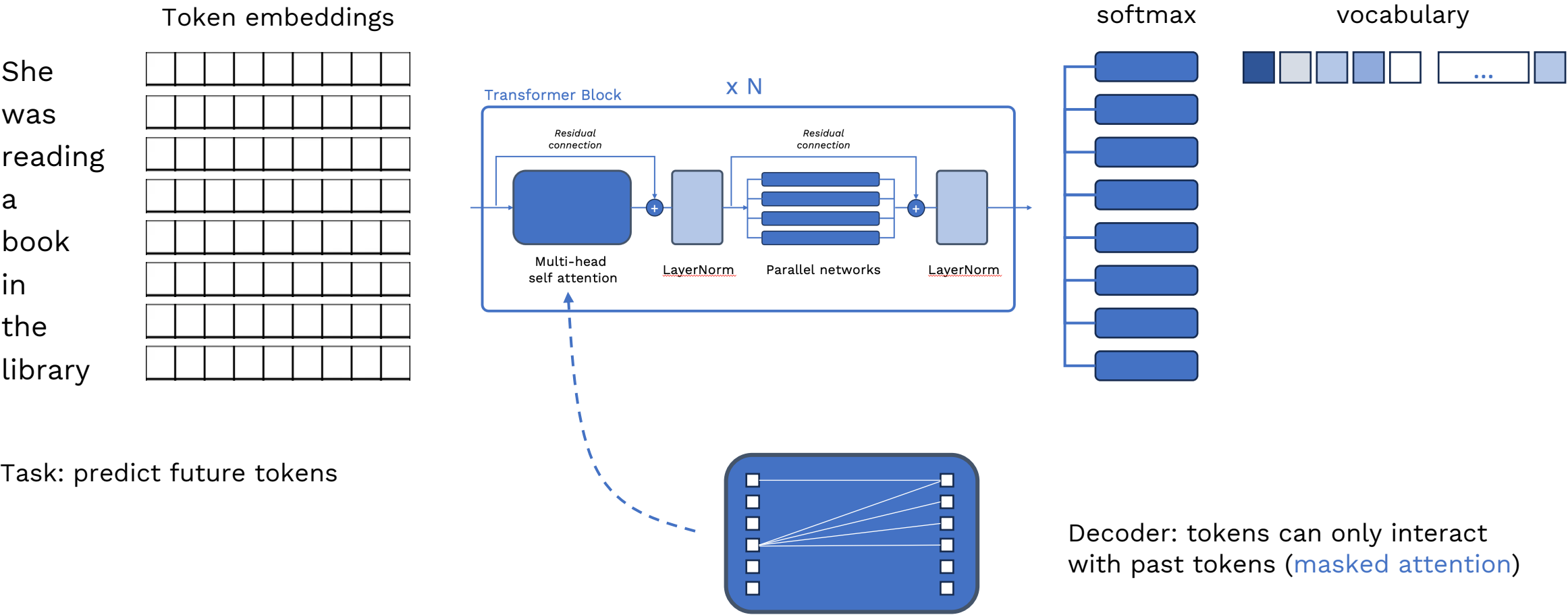
embedding size x vocab size

Position embedding table

embedding size x block size

Figure 1: The Transformer - model architecture.

Practical 4: Let's build a GPT-like encoder!



Dataset generation

First, you know Caius Marcius is chief enemy to the people.

18 47 56 57 22 13

Dataset generation

Batch	18	47	56	57	22	13
	22	31	82	16	46	81
	46	36	32	82	10	11
	74	59	82	91	60	38
	27	21	37	26	42	18

Dataset generation

X							Y					
Batch	18	47	56	57	22	13	47	56	57	22	13	42
	22	31	82	16	46	81	31	82	16	46	81	32
	46	36	32	82	10	11	36	32	82	10	11	69
	74	59	82	91	60	38	59	82	91	60	38	41
	27	21	37	26	42	18	21	37	26	42	18	77

Dataset generation

Example 1

	X							Y					
	18	47	56	57	22	13	47	56	57	22	13	42	
	22	31	82	16	46	81	31	82	16	46	81	32	
Batch	46	36	32	82	10	11	36	32	82	10	11	69	
	74	59	82	91	60	38	59	82	91	60	38	41	
	27	21	37	26	42	18	21	37	26	42	18	77	

Dataset generation

Example 2

X

Y

	18	47	56	57	22	13		47	56	57	22	13	42
	22	31	82	16	46	81		31	82	16	46	81	32
Batch	46	36	32	82	10	11		36	32	82	10	11	69
	74	59	82	91	60	38		59	82	91	60	38	41
	27	21	37	26	42	18		21	37	26	42	18	77

Dataset generation

Example 3

	X							Y					
	18	47	56	57	22	13		47	56	57	22	13	42
	22	31	82	16	46	81		31	82	16	46	81	32
Batch	46	36	32	82	10	11		36	32	82	10	11	69
	74	59	82	91	60	38		59	82	91	60	38	41
	27	21	37	26	42	18		21	37	26	42	18	77

Dataset generation

	X						Y					
	18	47	56	57	22	13	47	56	57	22	13	42
	22	31	82	16	46	81	31	82	16	46	81	32
Batch	46	36	32	82	10	11	36	32	82	10	11	69
	74	59	82	91	60	38	59	82	91	60	38	41
	27	21	37	26	42	18	21	37	26	42	18	77

Dataset generation

	X						Y					
Batch	18	47	56	57	22	13	47	56	57	22	13	42
	22	31	82	16	46	81	31	82	16	46	81	32
	46	36	32	82	10	11	36	32	82	10	11	69
	74	59	82	91	60	38	59	82	91	60	38	41
	27	21	37	26	42	18	21	37	26	42	18	77

Example 30

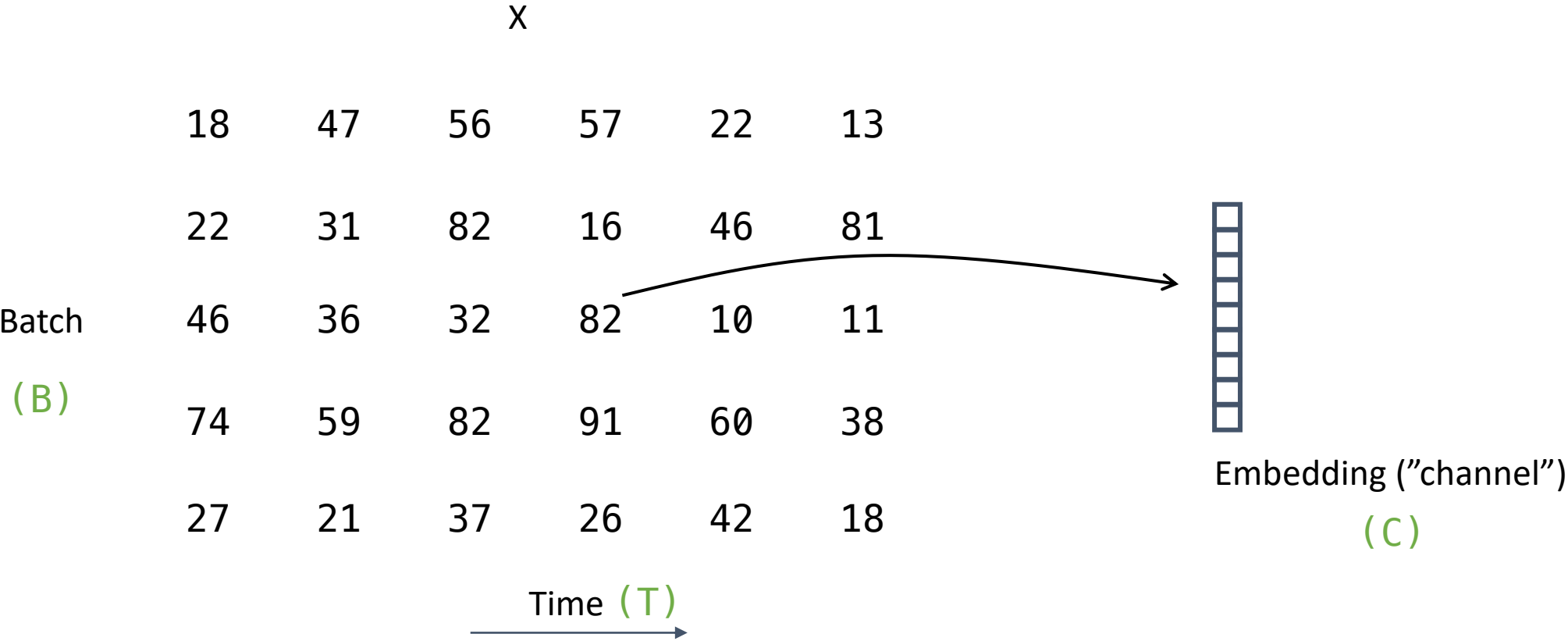
Dataset generation

x

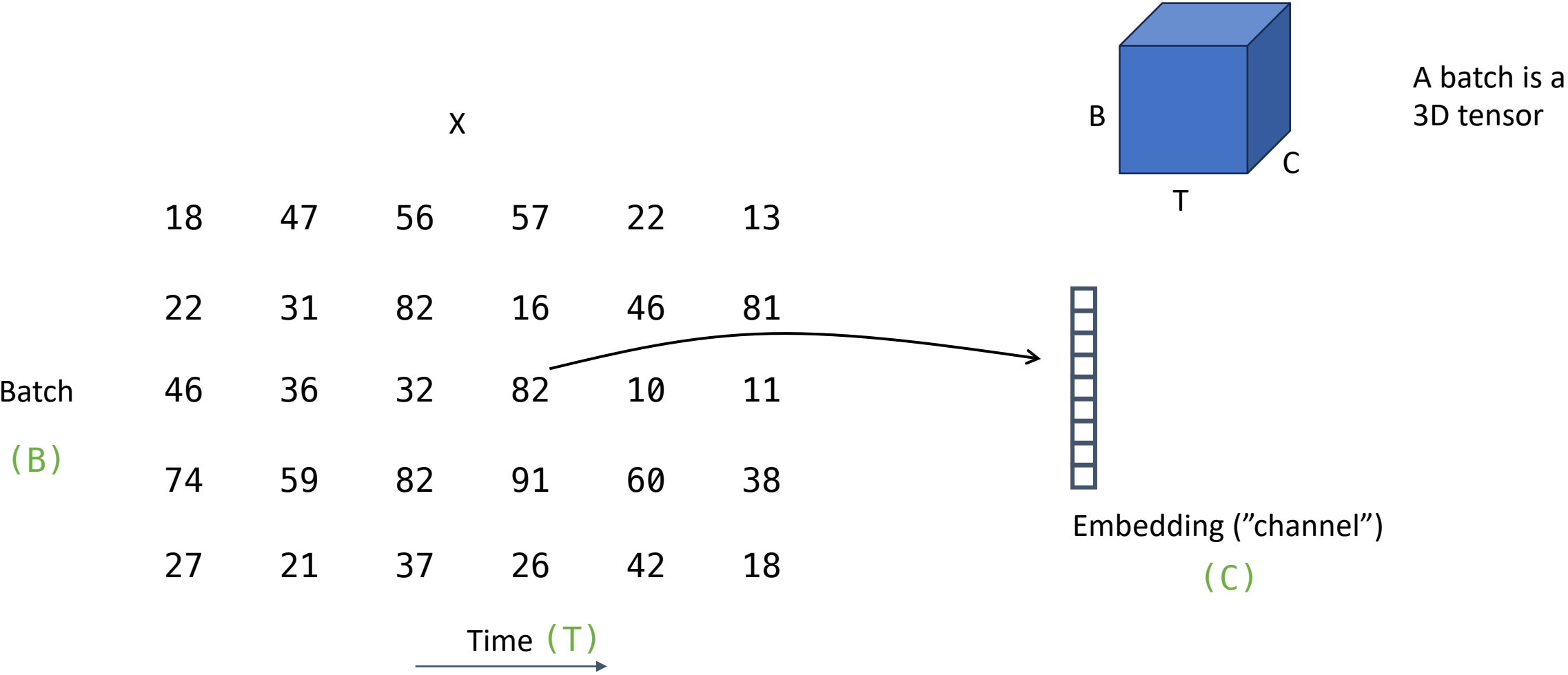
Batch

18	47	56	57	22	13
22	31	82	16	46	81
46	36	32	82	10	11
74	59	82	91	60	38
27	21	37	26	42	18

Dataset generation



Dataset generation



Tensor computation

```
ones = torch.zeros(2, 2) + 1
```

```
twos = torch.ones(2, 2) * 2
```

```
threes = (torch.ones(2, 2) * 7 - 1) / 2
```

```
fours = twos ** 2
```

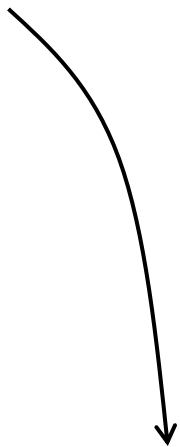
```
sqrt2s = twos ** 0.5
```

Tensor computation

```
powers2 = twos ** torch.tensor([[1, 2], [3, 4]])
```

```
fives = ones + fours
```

```
dozens = threes * fours
```



```
tensor([[ 2.,  4.],  
        [ 8., 16.]])
```

Tensor computation

```
a = torch.rand((2,4,3))
```

```
a.transpose()
```

Tensor computation

```
a = torch.rand((2,4,3))
```

```
a.transpose()
```

TypeError: transpose() received an invalid combination of arguments

Tensor computation

```
a = torch.rand((2,4,3))
```

```
a.transpose(-2, -1)
```

```
a.shape
```

```
torch.Size([2, 3, 4])
```

Tensor computation

```
a = torch.rand(2, 3)
b = torch.rand(3, 2)

print(a * b)
```

Tensor computation

```
a = torch.rand(2, 3)
b = torch.rand(3, 2)

print(a * b)
```

RuntimeError: The size of tensor a (3) must match the size of tensor b (2) at non-singleton dimension 1

Tensor computation

```
a = torch.rand(2, 3)
b = torch.rand(3, 2)

print(a @ b)
```

This **works!**

@ is for matrix multiplication

* is for element-wise multiplication

Tensor broadcasting

```
a = torch.rand(2, 3)  
b = torch.rand(1, 3)
```

```
print(a * b)
```

This works!

Tensor broadcasting

```
a = torch.tensor([[1, 2, 1], [2, 5, 1]])  
b = torch.ones(1, 3) + 1  
  
print(a * b)  
  
tensor([[ 2.,  4.,  2.],  
        [ 4., 10.,  2.]])
```

Tensor broadcasting

Broadcasting rules:

Comparing the dimension sizes of the two tensors, going from last to first:

- Each dimension must be equal, or

- One of the dimensions must be of size 1, or

- The dimension does not exist in one of the tensors

Tensor computation

```
a = torch.rand(5, 4, 3)
b = torch.rand(1, 3, 6)

print(a @ b)
```

Tensor computation

```
a = torch.rand(5, 4, 3)
b = torch.rand(1, 3, 6)

print(a @ b)
```

This **works!**

Tensor computation

```
a = torch.rand(1, 5, 4, 3)
b = torch.rand(3, 1, 3, 6)

print(a @ b)
```

Tensor computation

```
a = torch.rand(1, 5, 4, 3)
b = torch.rand(3, 1, 3, 6)

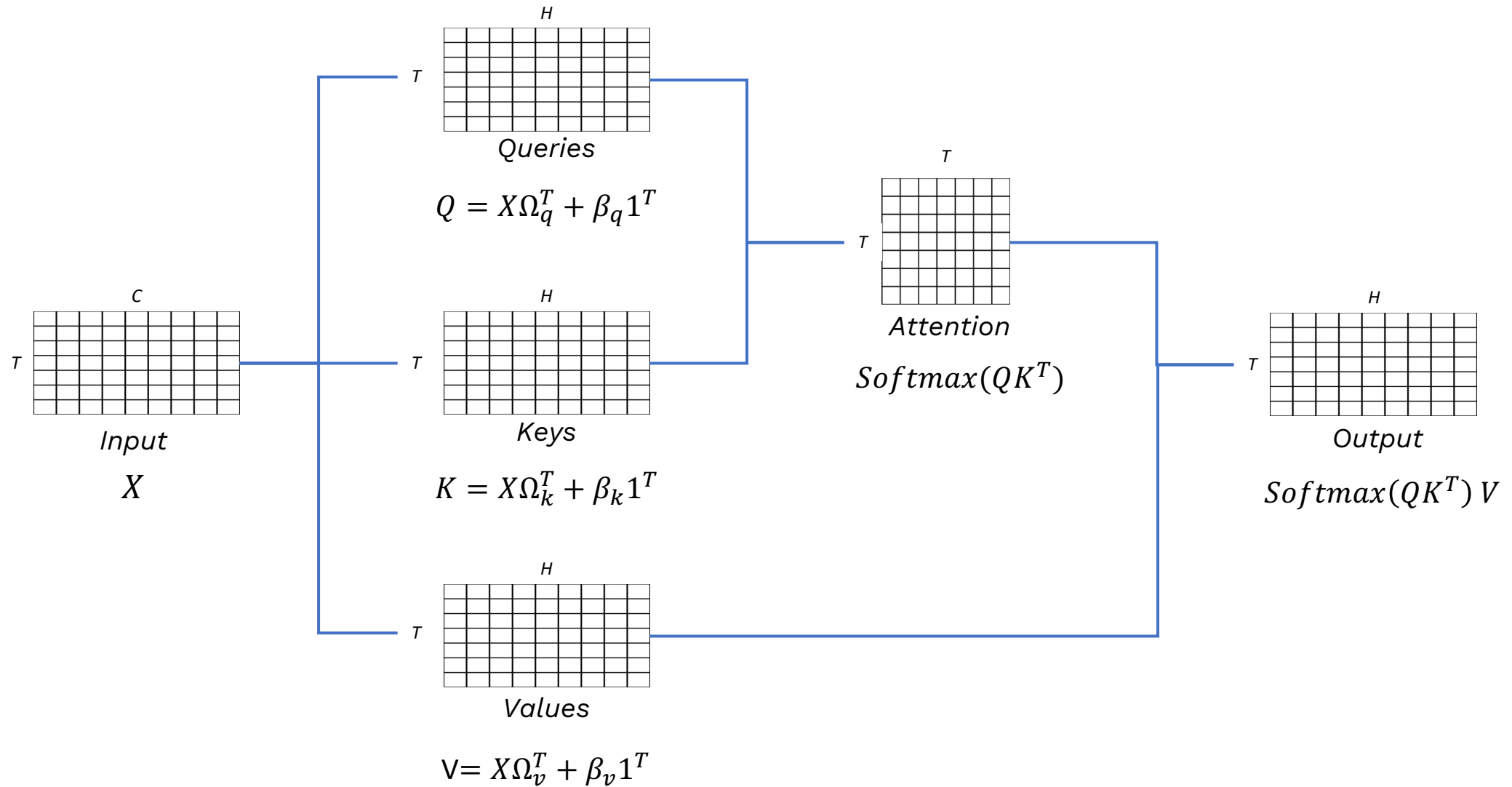
print(a @ b)
```

This **works!**

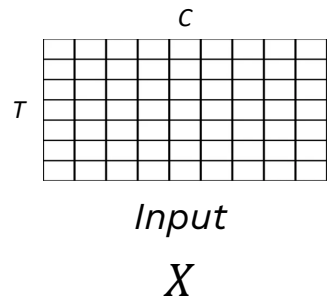
Tensor computation

`xbow = wei @ x # (B, T, T) x (B, T, C) --> (B, T, C)`

In a pytorch implementation, the last two dimensions are **inverted**!

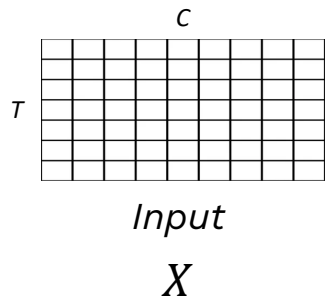


In a pytorch implementation, the last two dimensions are **inverted**!



This is because pytorch is [channel-last](#) for memory optimization.

In a pytorch implementation, the last two dimensions are **inverted**!



A linear layer `nn.Linear(in, out)` implements:

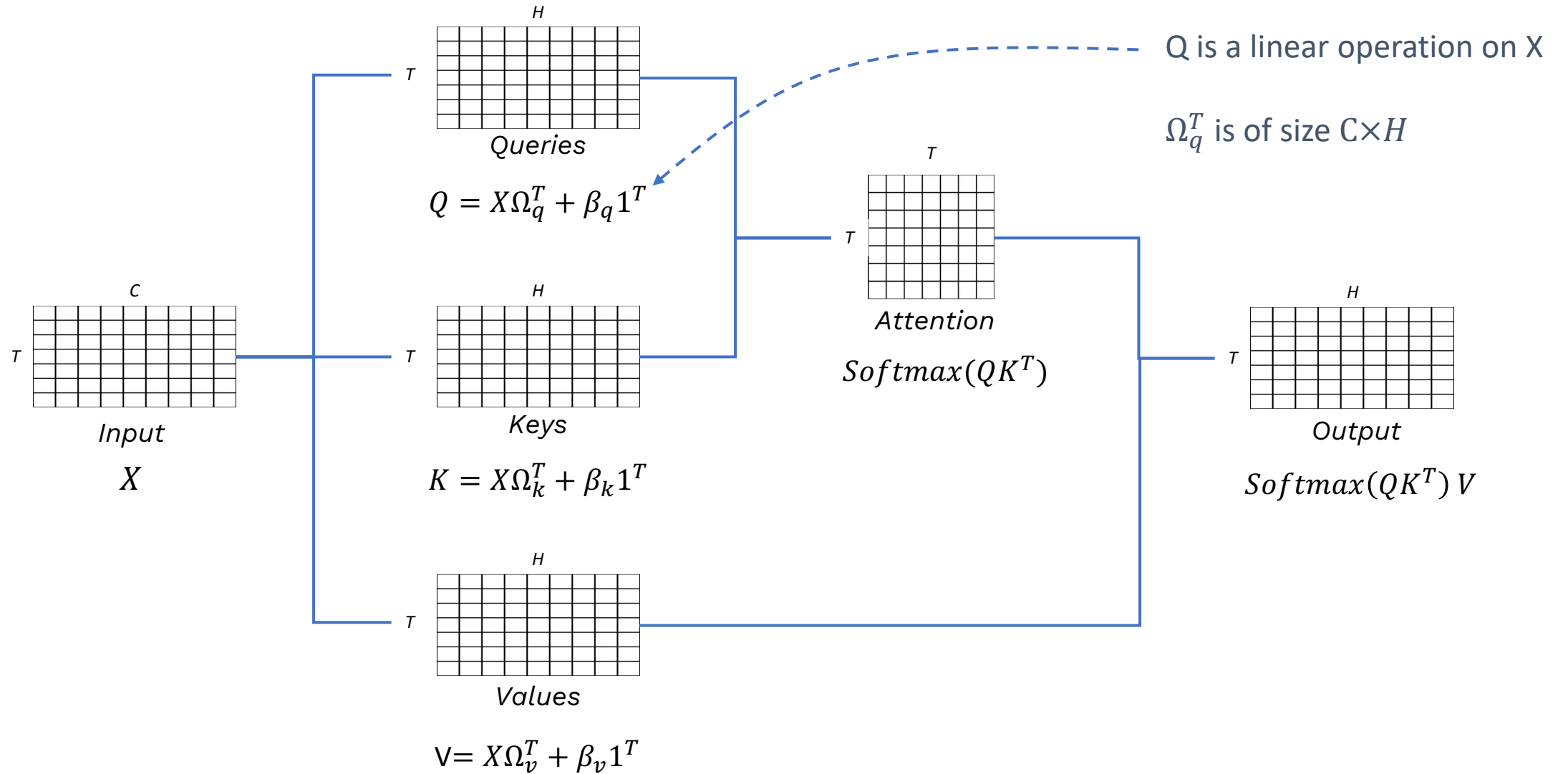
$$y = x.A^T + b$$

and not

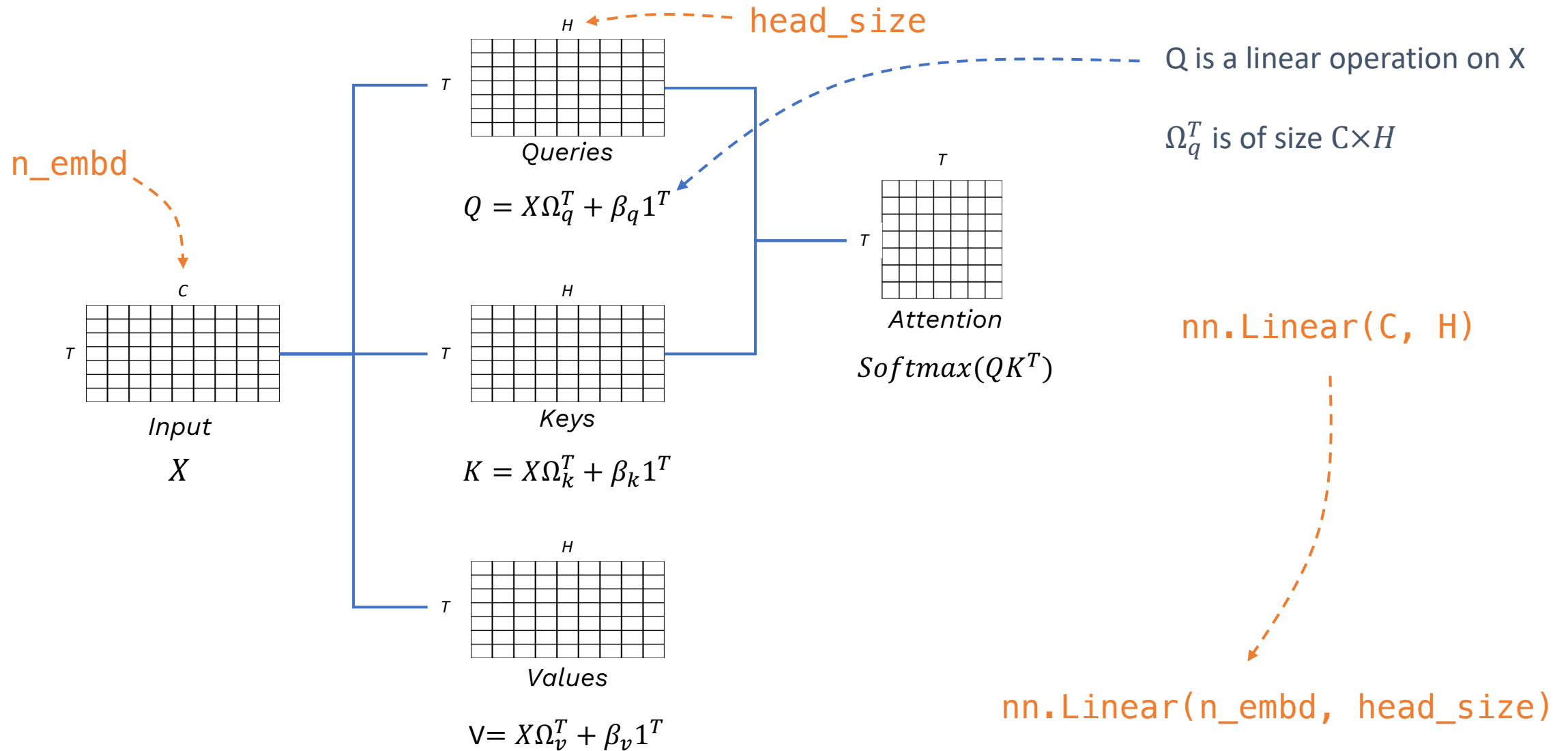
$$y = A.x + b$$

Therefore the shape of A is `(out, in)`

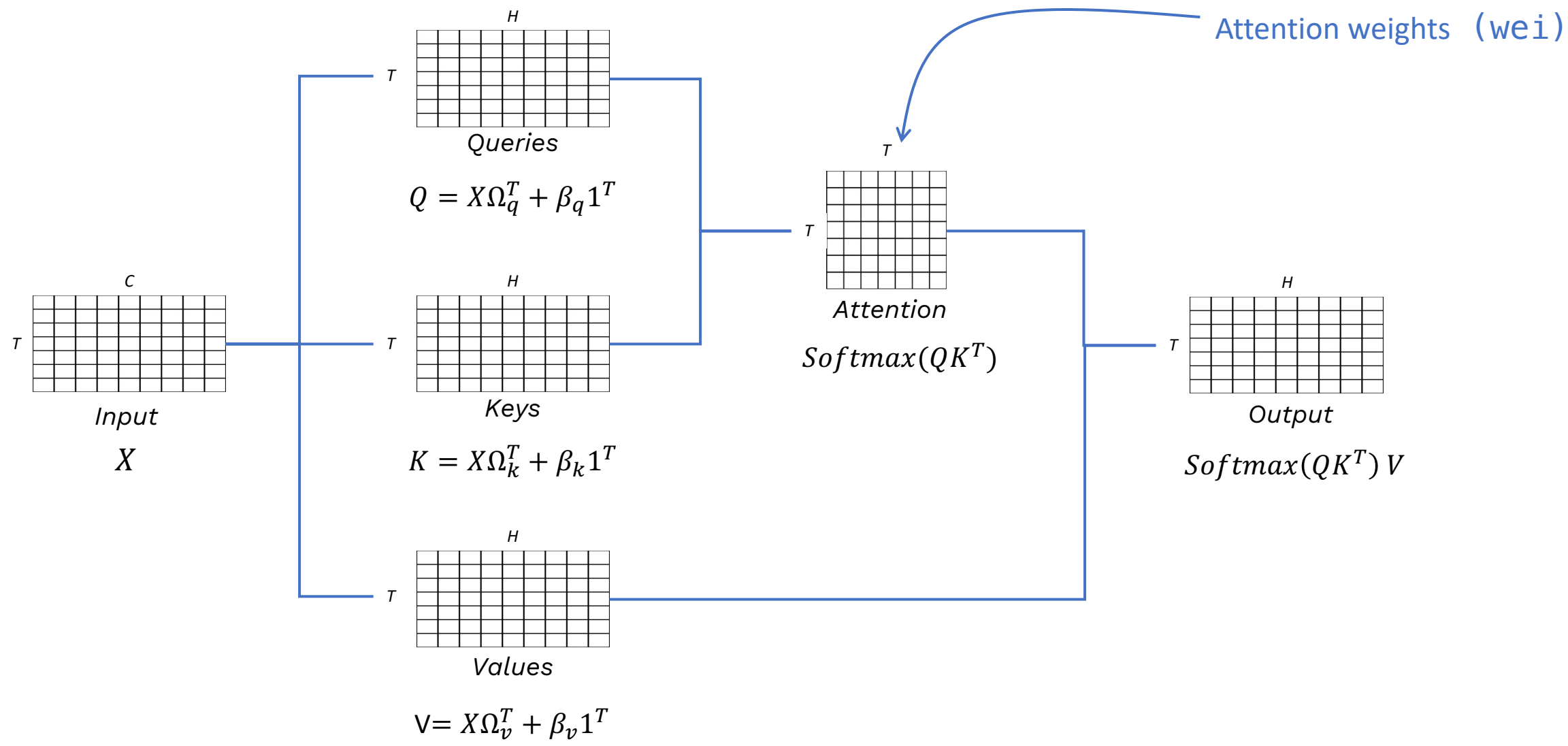
In a pytorch implementation, the last two dimensions are **inverted**!



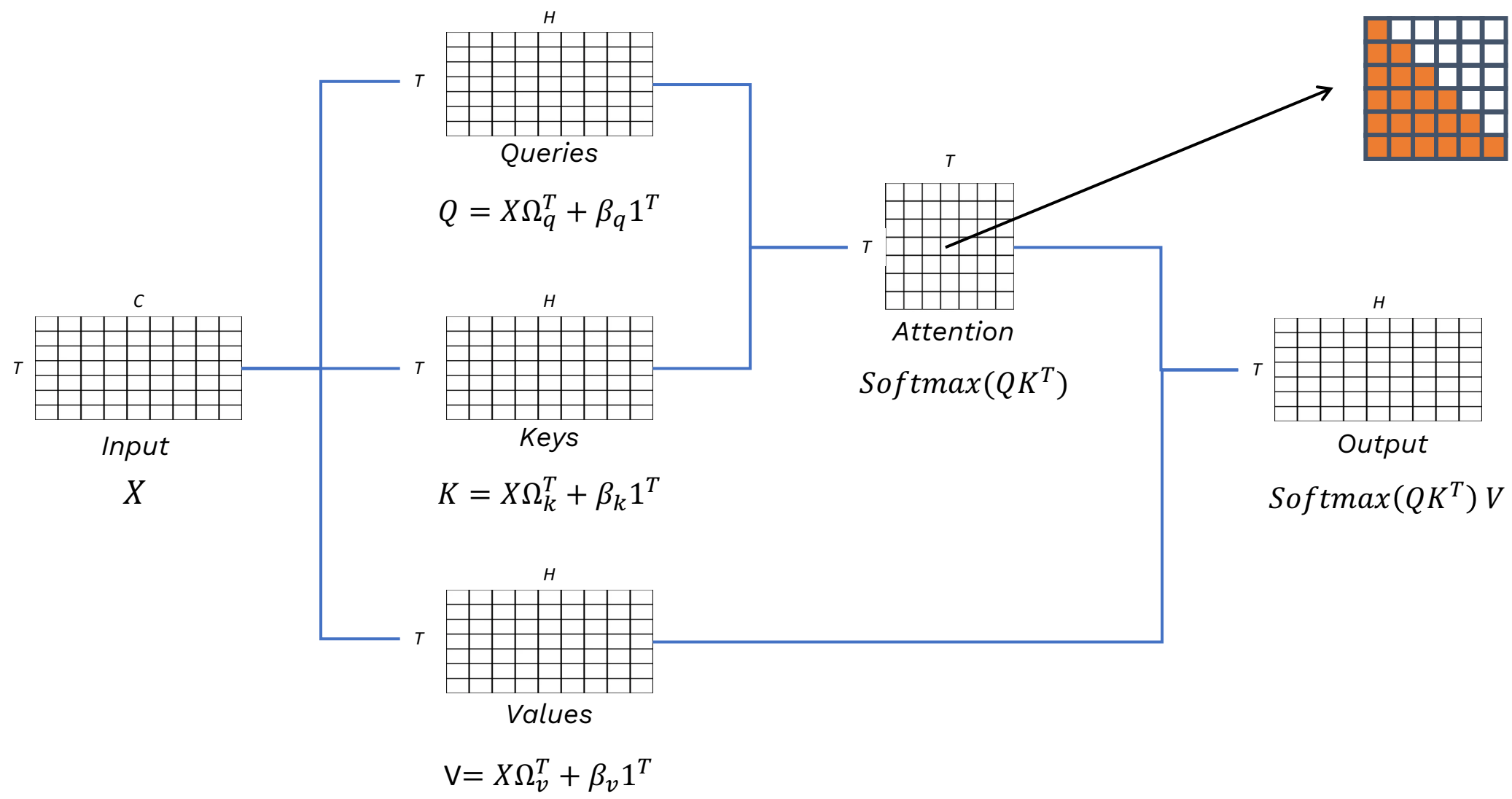
In a pytorch implementation, the last two dimensions are **inverted**!



Masked self-attention



Masked self-attention



Masked self-attention

```
tril = torch.tril(torch.ones(T,T))
```

```
tensor([[1., 0., 0.],  
        [1., 1., 0.],  
        [1., 1., 1.]])
```

Masked self-attention

```
tril = torch.tril(torch.ones(T,T))
```

```
wei = torch.zeros((T,T))
```

```
wei = wei.masked_fill(tril == 0, float('-inf'))
```

```
tensor([[0., -inf, -inf],  
        [0., 0., -inf],  
        [0., 0., 0.]])
```

Masked self-attention

```
tril = torch.tril(torch.ones(T,T))

wei = torch.zeros((T,T))

wei = wei.masked_fill(tril == 0, float('-inf'))

tensor([[0., -inf, -inf],
        [0., 0., -inf],
        [0., 0., 0.]])

wei = F.softmax(wei, dim=-1)
```

Masked self-attention

```
tril = torch.tril(torch.ones(T,T))
```

```
wei = torch.zeros((T,T))
```

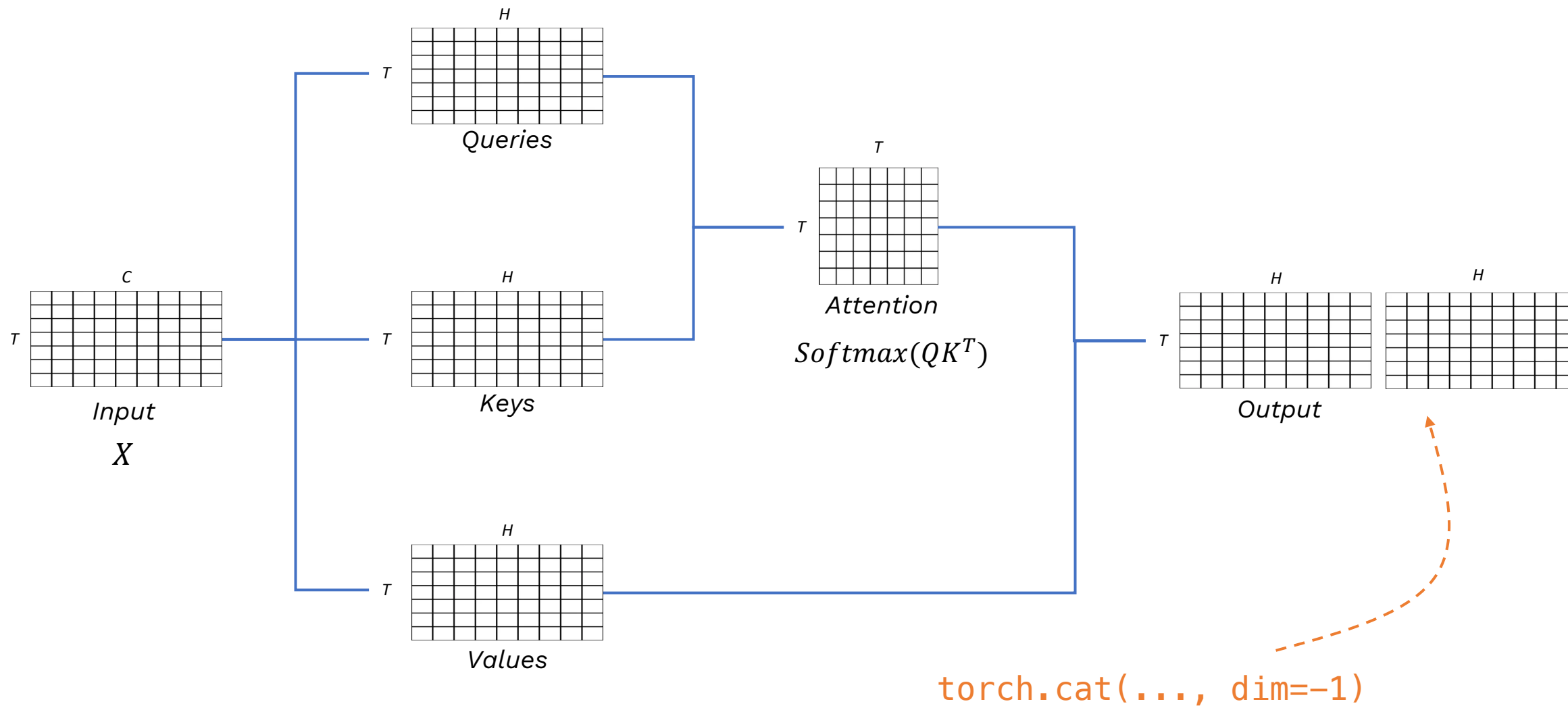
```
wei = wei.masked_fill(tril == 0, float('-inf'))
```

```
tensor([[0., -inf, -inf],  
        [0., 0., -inf],  
        [0., 0., 0.]])
```

```
wei = F.softmax(wei, dim=-1)
```

```
tensor([[1.0000, 0.0000, 0.0000],  
        [0.5000, 0.5000, 0.0000],  
        [0.3333, 0.3333, 0.3333]])
```

Beware of concatenation in multi-head



Getting started

```
class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(..., ..., bias=False)
        ...

    def forward (self, x):
        B, T, C = x.shape
        k = self.key(x) # (B,T,C)
        ...
```

Getting started

```
class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(..., ..., bias=False)
        ...

    def forward (self, x):
        B, T, C = x.shape
        k = self.key(x) # (B,T,C)
        q = ...
        # compute self attention scores (affinities)
        wei = ...
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
        wei = F.softmax(wei, dim=-1)
        ...
```

Practical 4 summary

1. Self-attention by hand
2. Self-attention in pytorch
3. GPT piece-by-piece
4. GPU goes rrr!

Dataset: Shakespeare's corpus (input.txt)