ENSAE

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 • 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 • 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, unets	Convnets for CIFAR-10
6	VAE & Diffusion models	1D diffusion model 2D diffusion model (*) train on GPU **Quiz**

onfido

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.

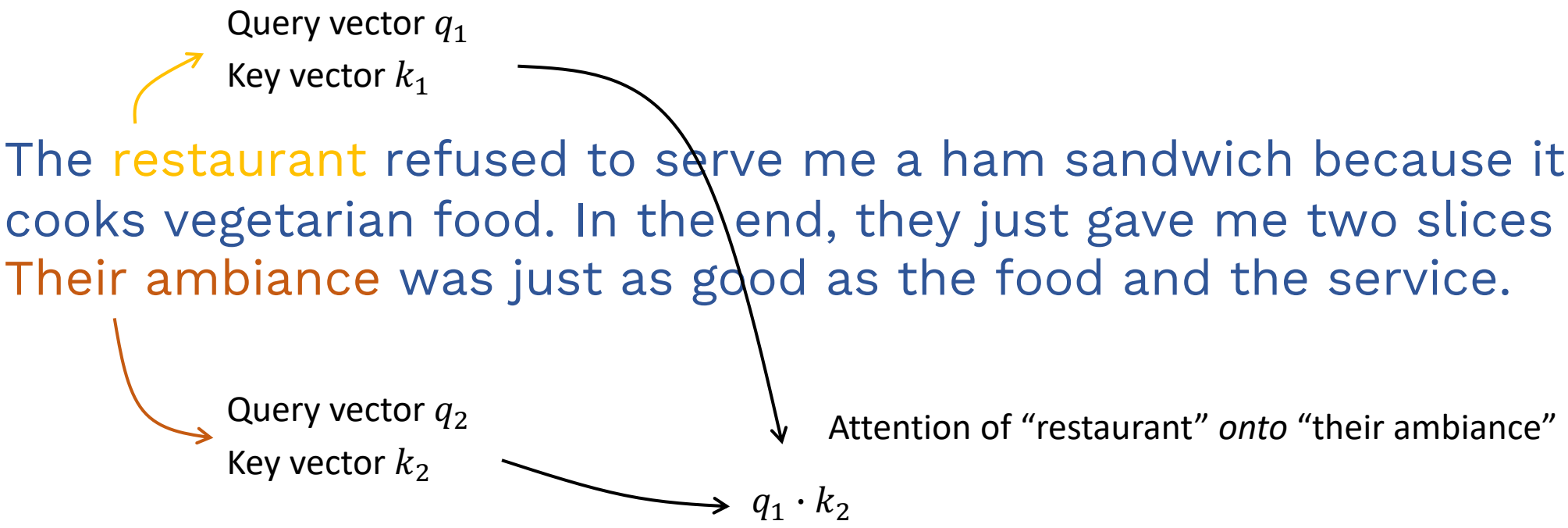
Query vector q_1

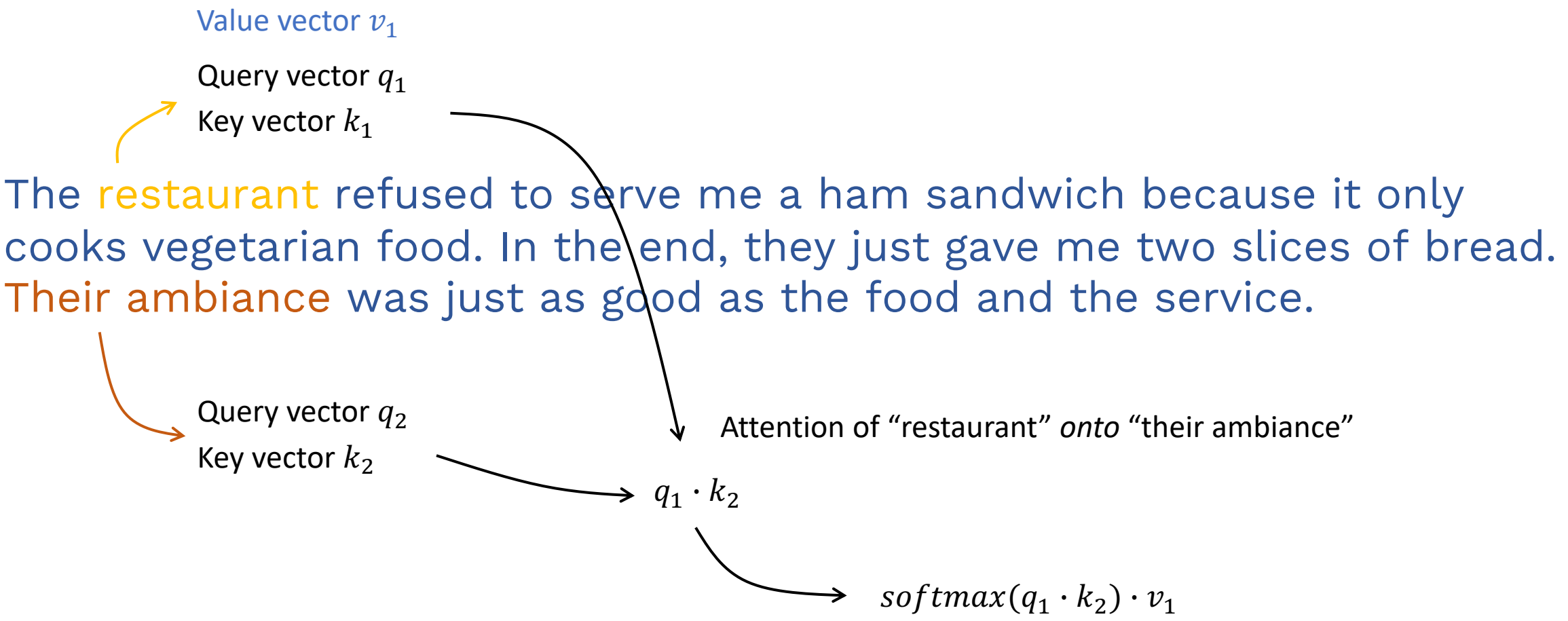
Key vector k_1

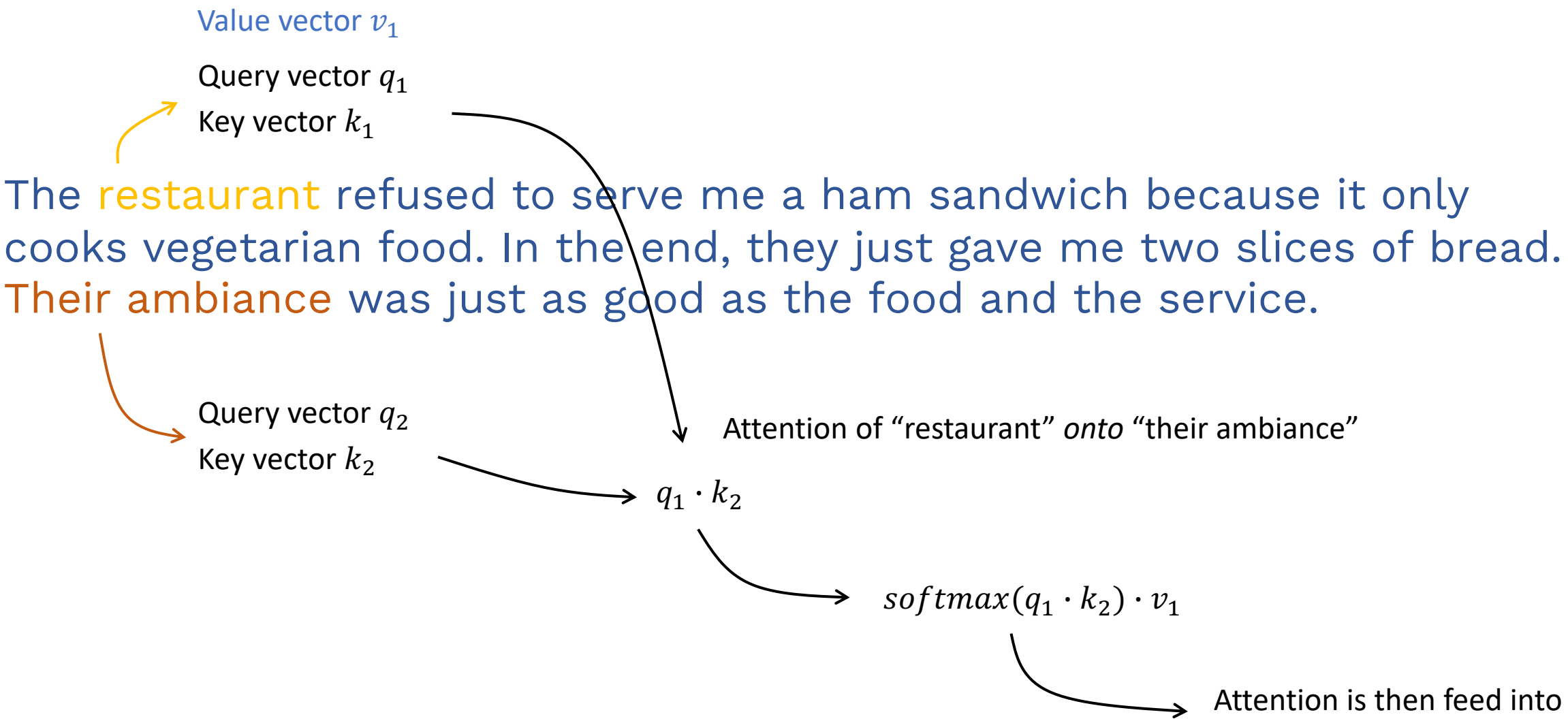
The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and the service.

Query vector q_2

Key vector k_2


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 of “restaurant” *onto* “their ambiance”
$q_1 \cdot k_2$


Value vector v_1
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 of "restaurant" *onto* "their ambiance"
$q_1 \cdot k_2$
$softmax(q_1 \cdot k_2) \cdot v_1$


Value vector v_1
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 of "restaurant" *onto* "their ambiance"
$q_1 \cdot k_2$
$softmax(q_1 \cdot k_2) \cdot v_1$
Attention is then feed into traditional architectures

Attention and Transformers

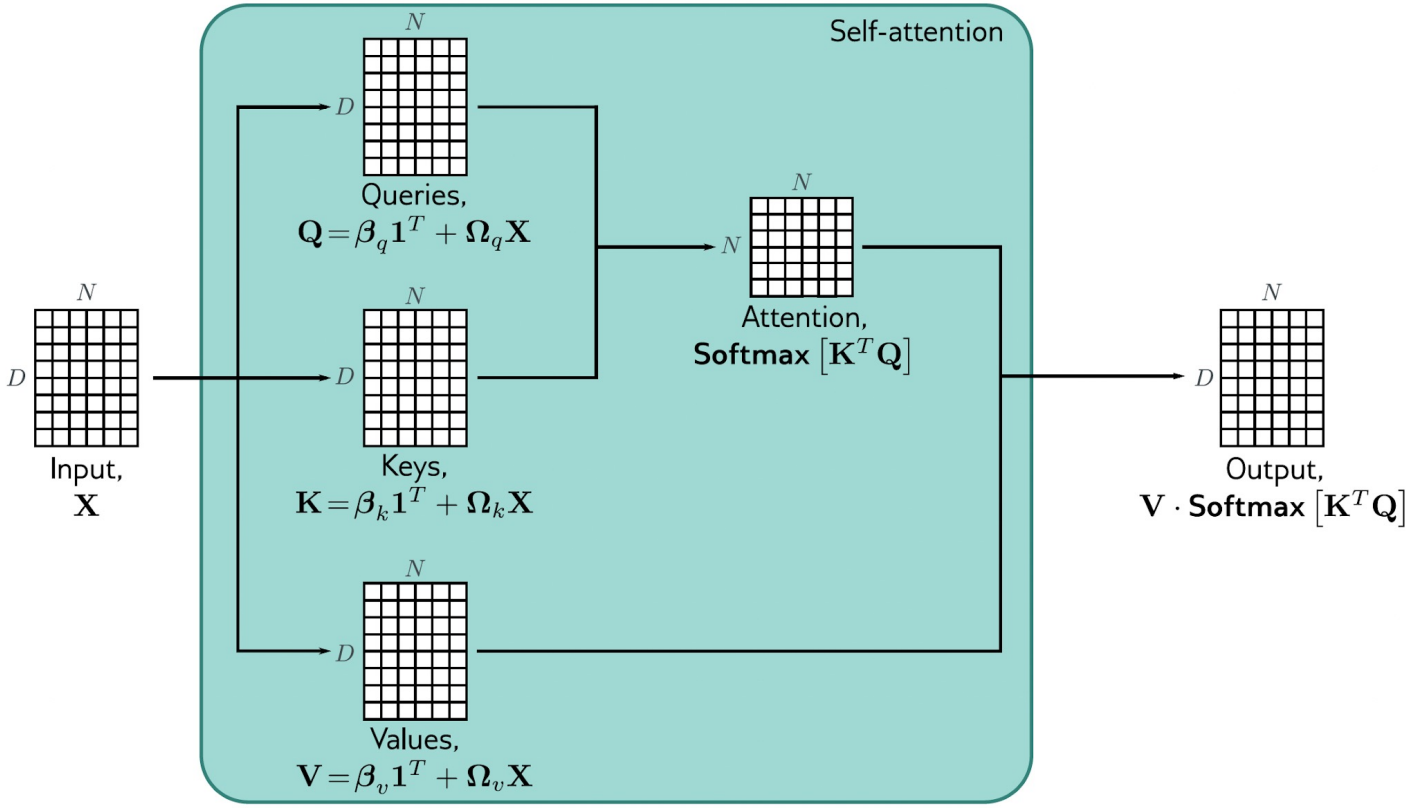


Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

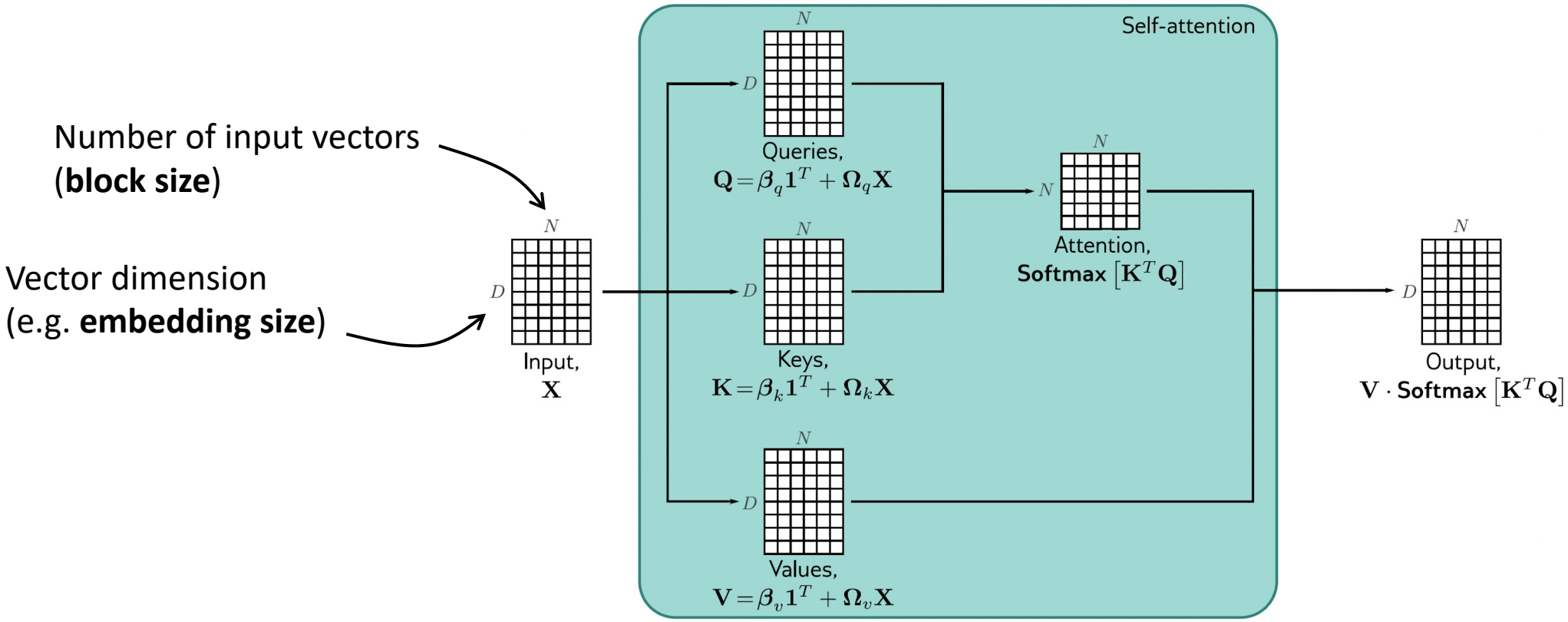


Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

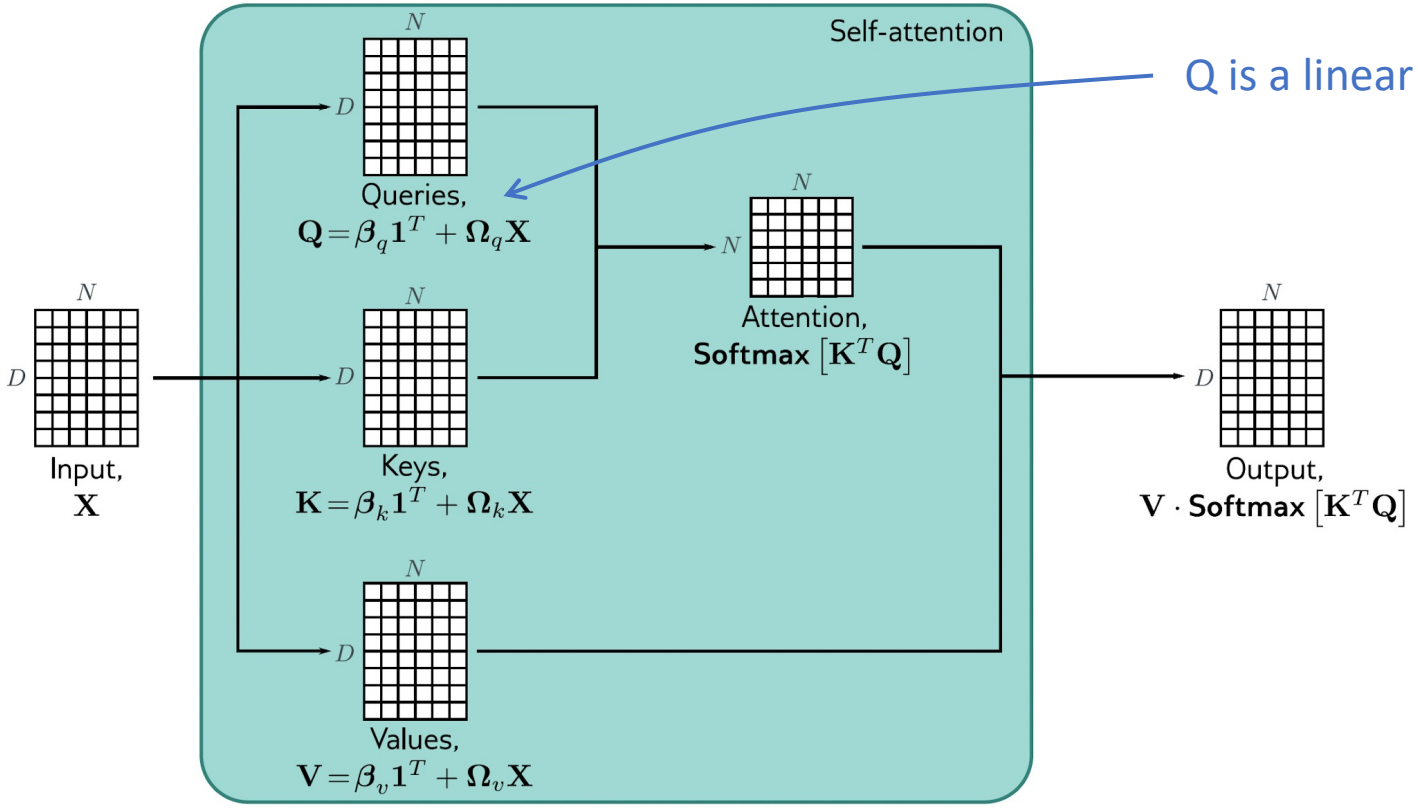


Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

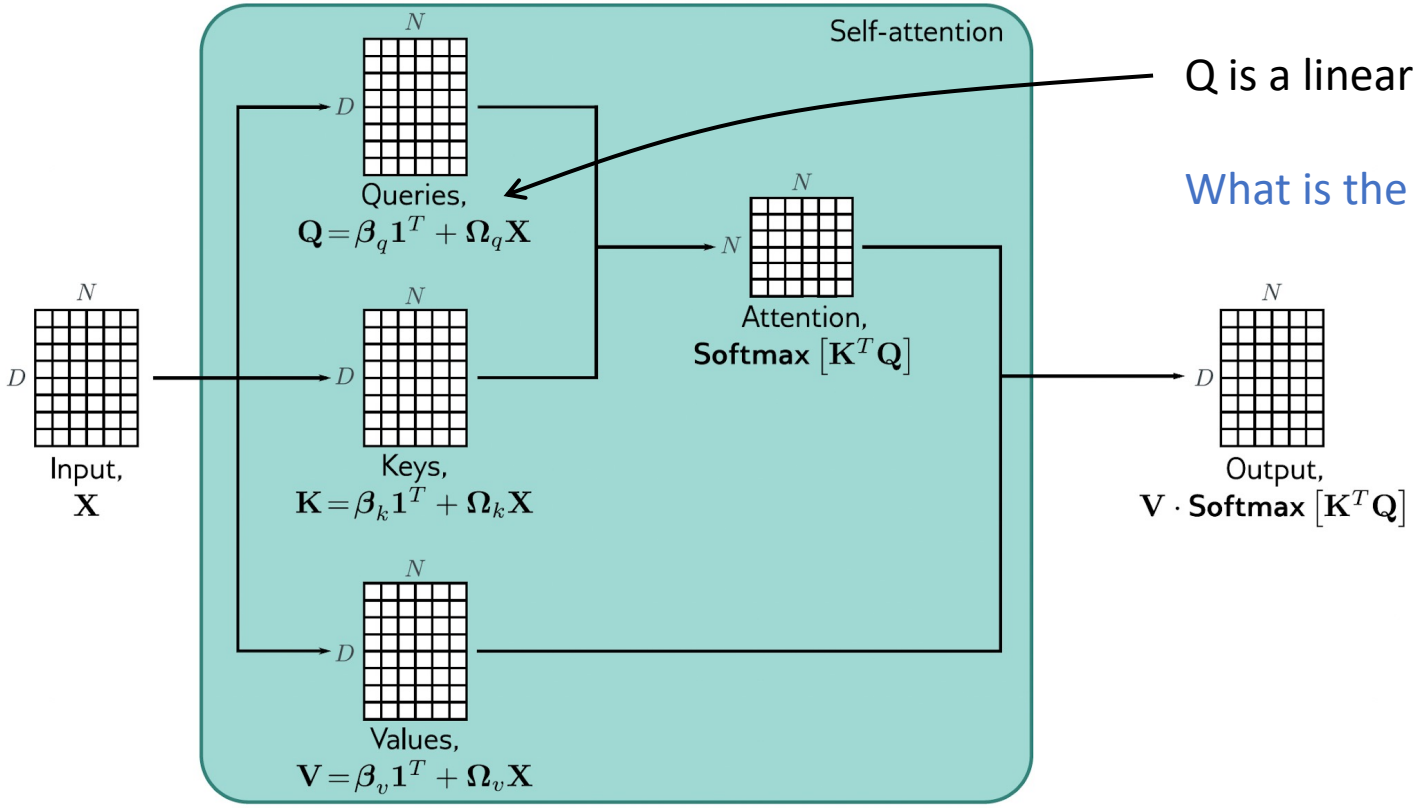
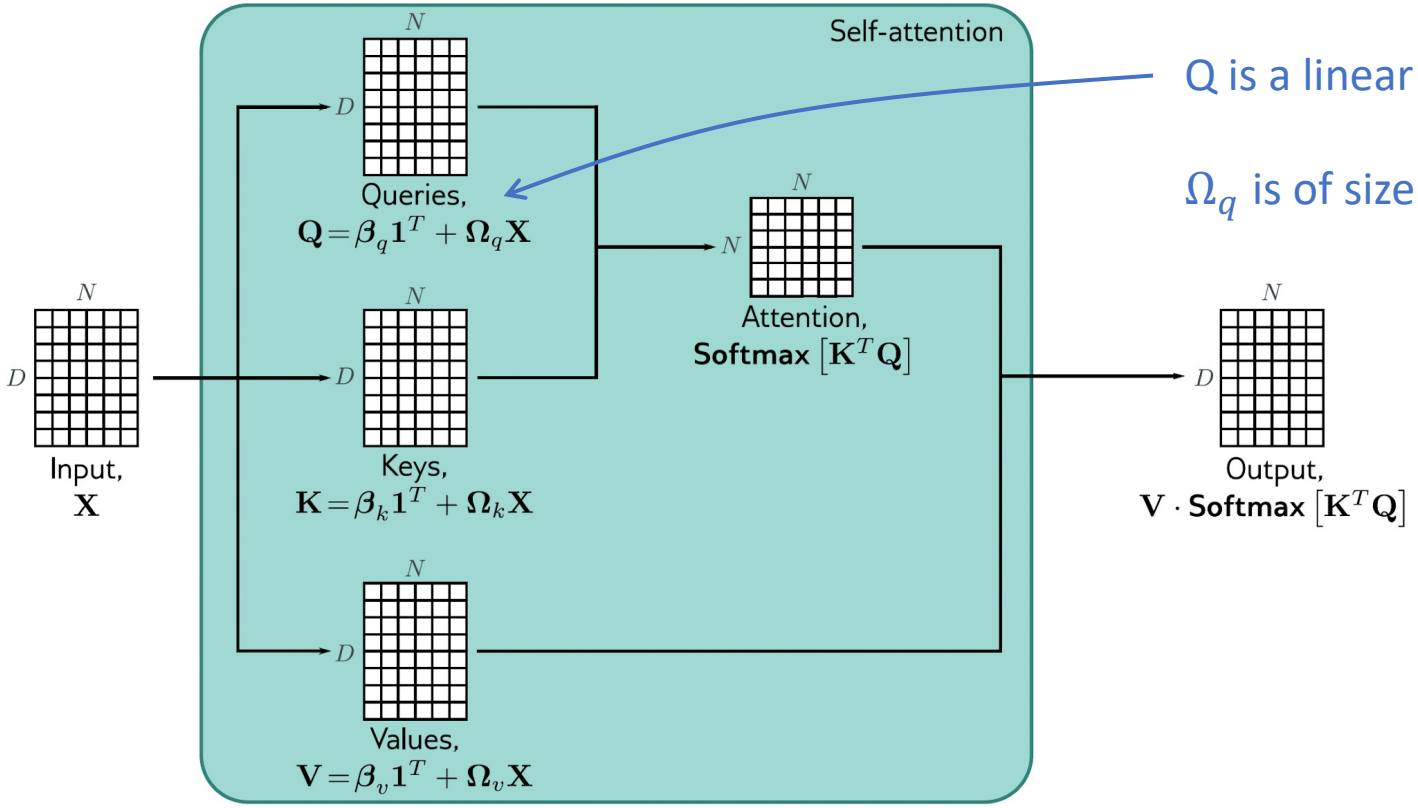


Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers



Q is a linear operation on X

Ω_q is of size $H \times D$

H is the **head size**

Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D \times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

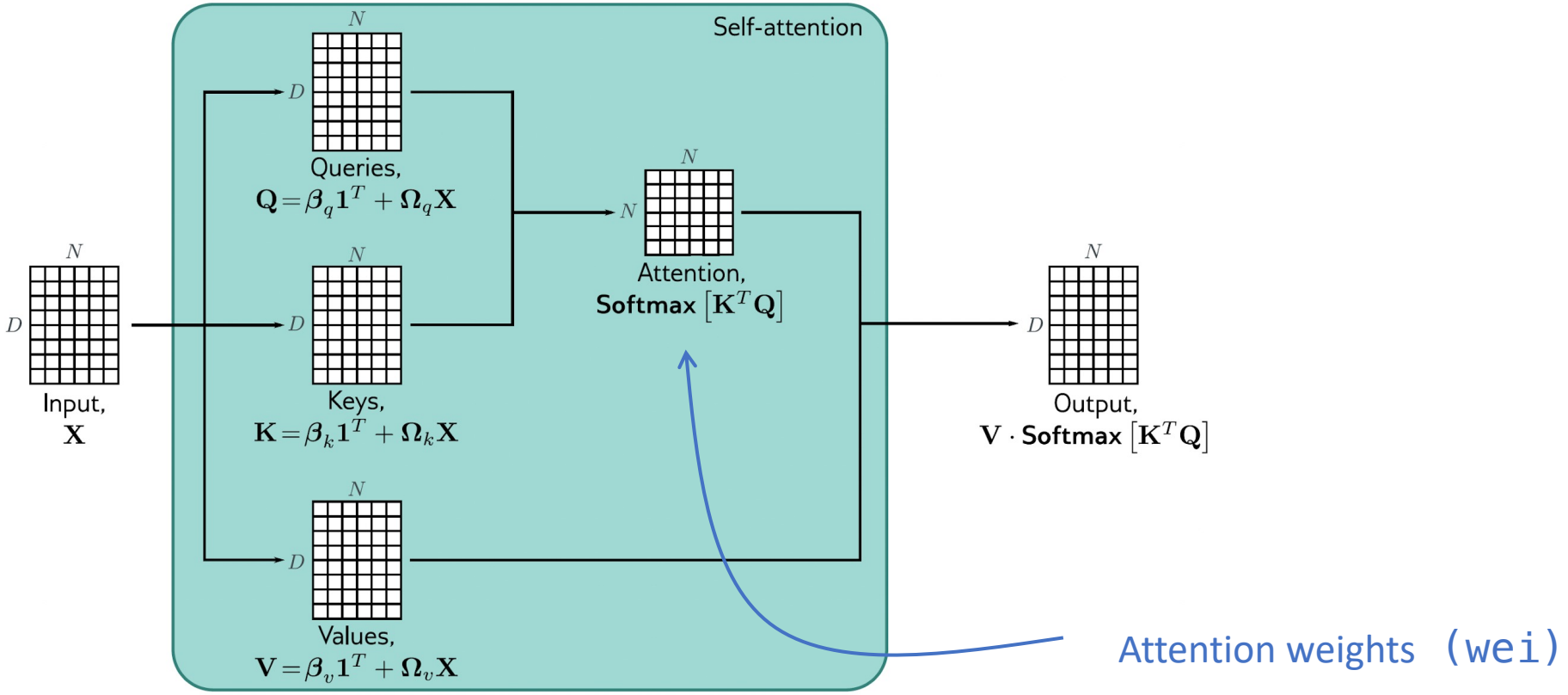
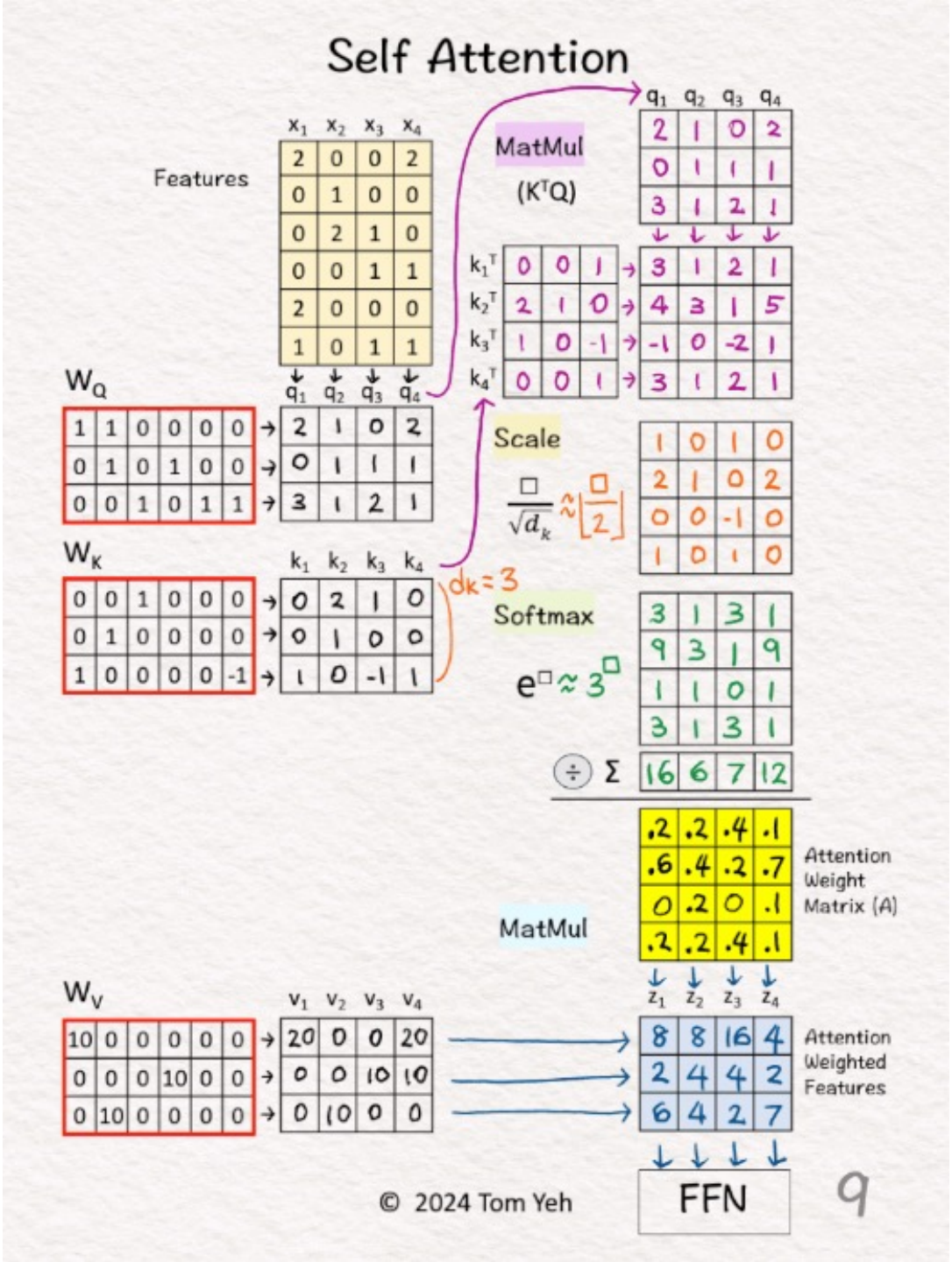


Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

Source: Understanding Deep Learning, S. Prince, 2023



Source: Tom Yeh, 2024

Scaled Dot-Product Self-Attention

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

Source: Understanding Deep Learning, S. Prince, 2023

Scaled Dot-Product Self-Attention

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

Quadratic in X!

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

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

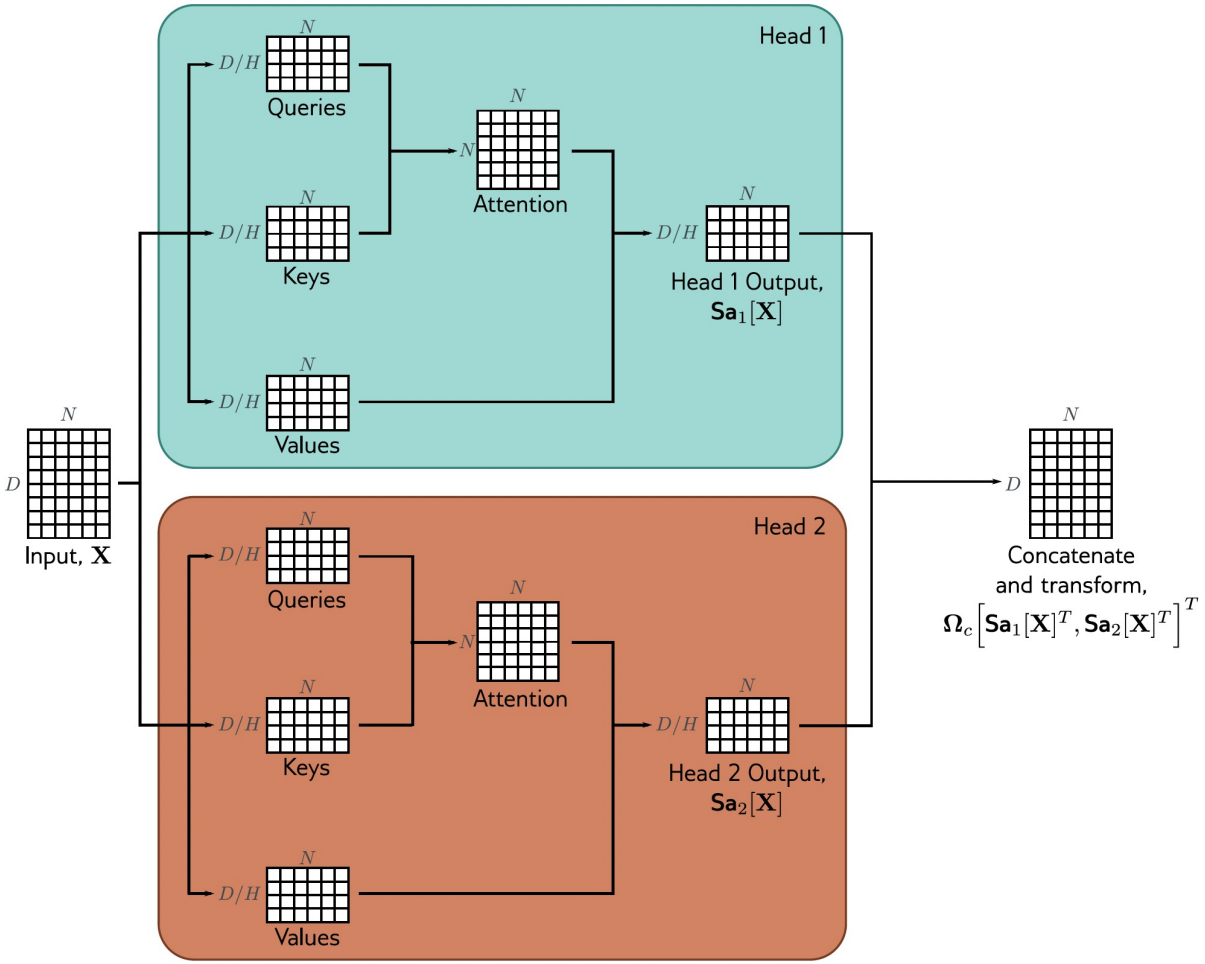


Figure 12.6 Multi-head self-attention. Self-attention occurs in parallel across multiple "heads." Each has its own queries, keys, and values. Here two heads are depicted, in the cyan and orange boxes, respectively. The outputs are vertically concatenated, and another linear transformation $\boldsymbol{\Omega}_c$ is used to recombine them.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

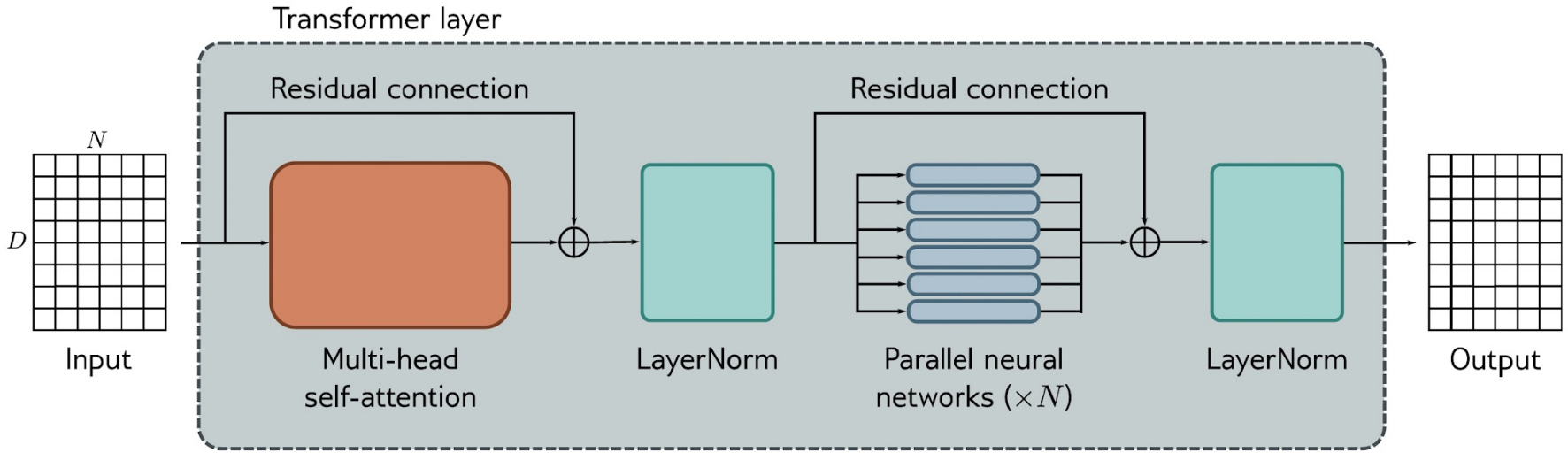


Figure 12.7 The transformer. The input consists of a $D \times N$ matrix containing the D-dimensional word embeddings for each of the N input tokens. The output is a matrix of the same size. The transformer consists of a series of operations. First, there is a multi-head attention block, allowing the word embeddings to interact with one another. This forms the processing of a residual block, so the inputs are added back to the output. Second, a LayerNorm operation is applied. Third, there is a second residual layer where the same fully connected neural network is applied separately to each of the N word representations (columns). Finally, LayerNorm is applied again.

Source: Understanding Deep Learning, S. Prince, 2023

Attention Is All You Need, A. Vaswani et al, NeurIPS 2017

Attention and Transformers

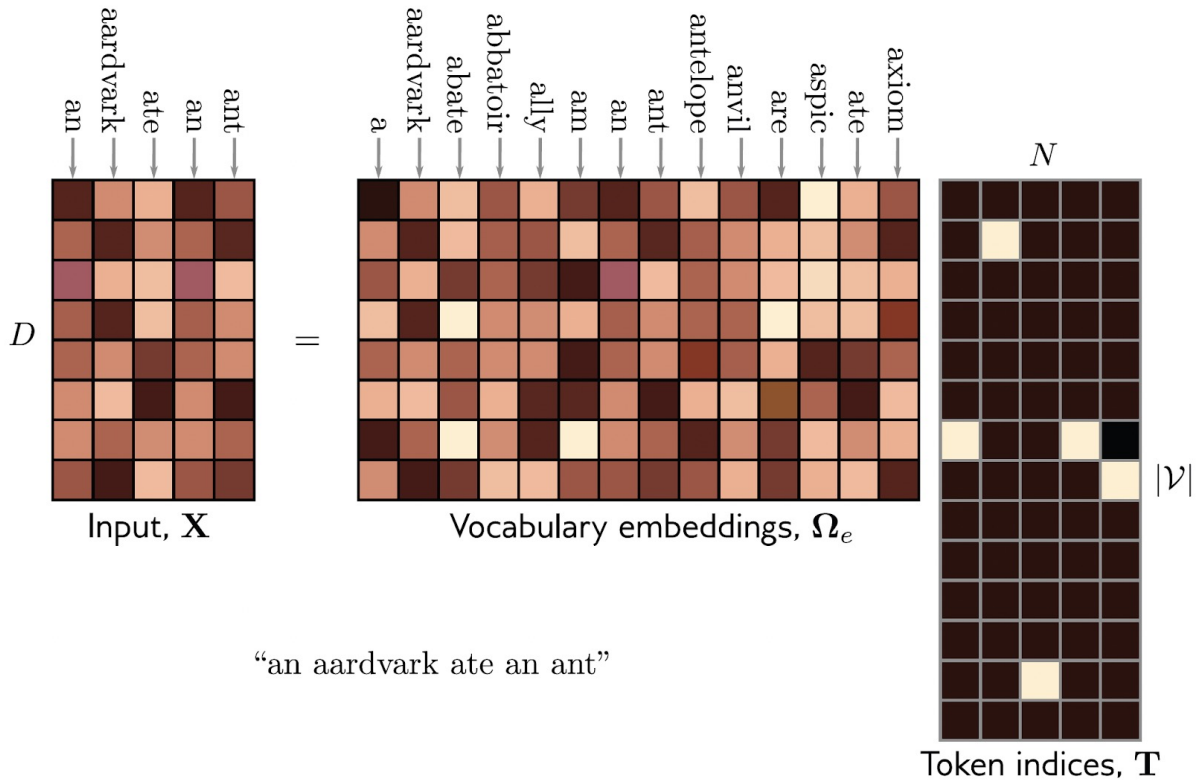


Figure 12.9 The input embedding matrix $\mathbf{X} \in \mathbb{R}^{D \times N}$ contains N embeddings of length D and is created by multiplying a matrix $\mathbf{\Omega}_e$ containing the embeddings for the entire vocabulary with a matrix containing one-hot vectors in its columns that correspond to the 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.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

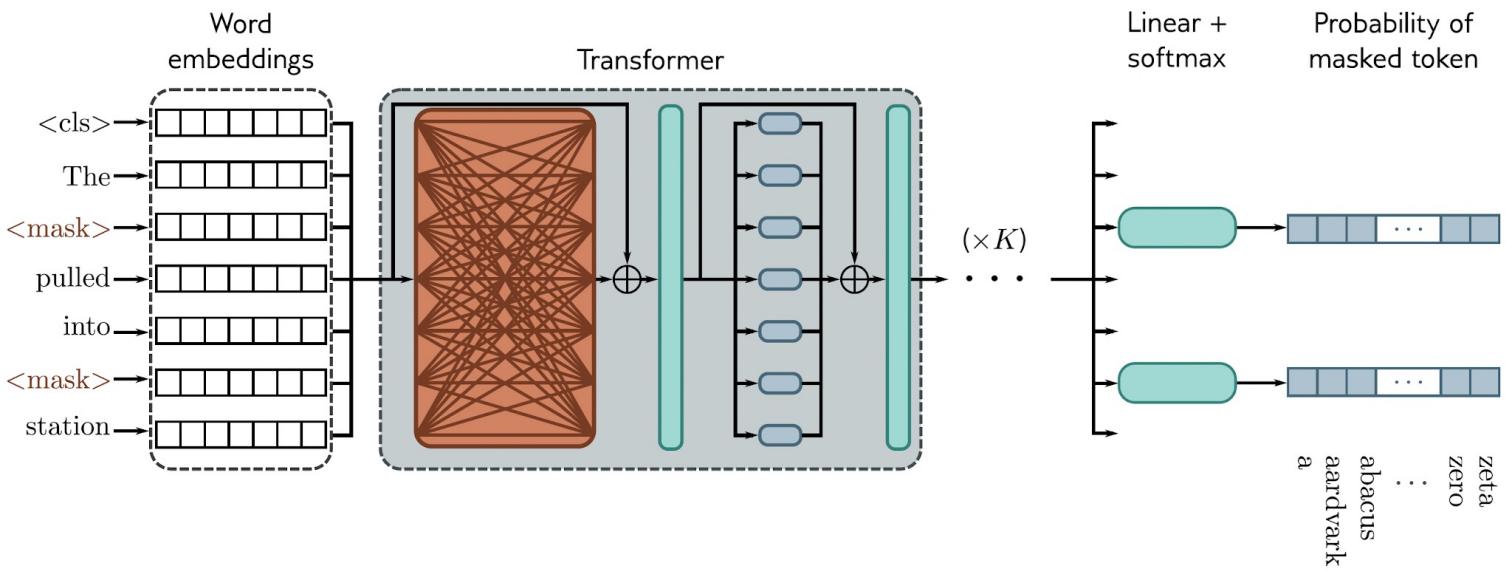


Figure 12.10 Pre-training for BERT-like encoder. The input tokens (and a special <cls> token denoting the start of the sequence) are converted to word embeddings. Here, these are represented as rows rather than columns, so the box labeled "word embeddings" is $\mathbf{X}^T$. These embeddings are passed through a series of transformers (orange connections indicate that every token attends to every other token in these layers) to create a set of output embeddings. A small fraction of the input tokens is randomly replaced with a generic <mask> token. In pre-training, the goal is to predict the missing word from the associated output embedding. As such, the output embeddings are passed through a softmax function, and the multiclass classification loss (section 5.24) is used. This task has the advantage that it uses both the left and right context to predict the missing word but has the disadvantage that it does not make efficient use of data; here, seven tokens need to be processed to add two terms to the loss function.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

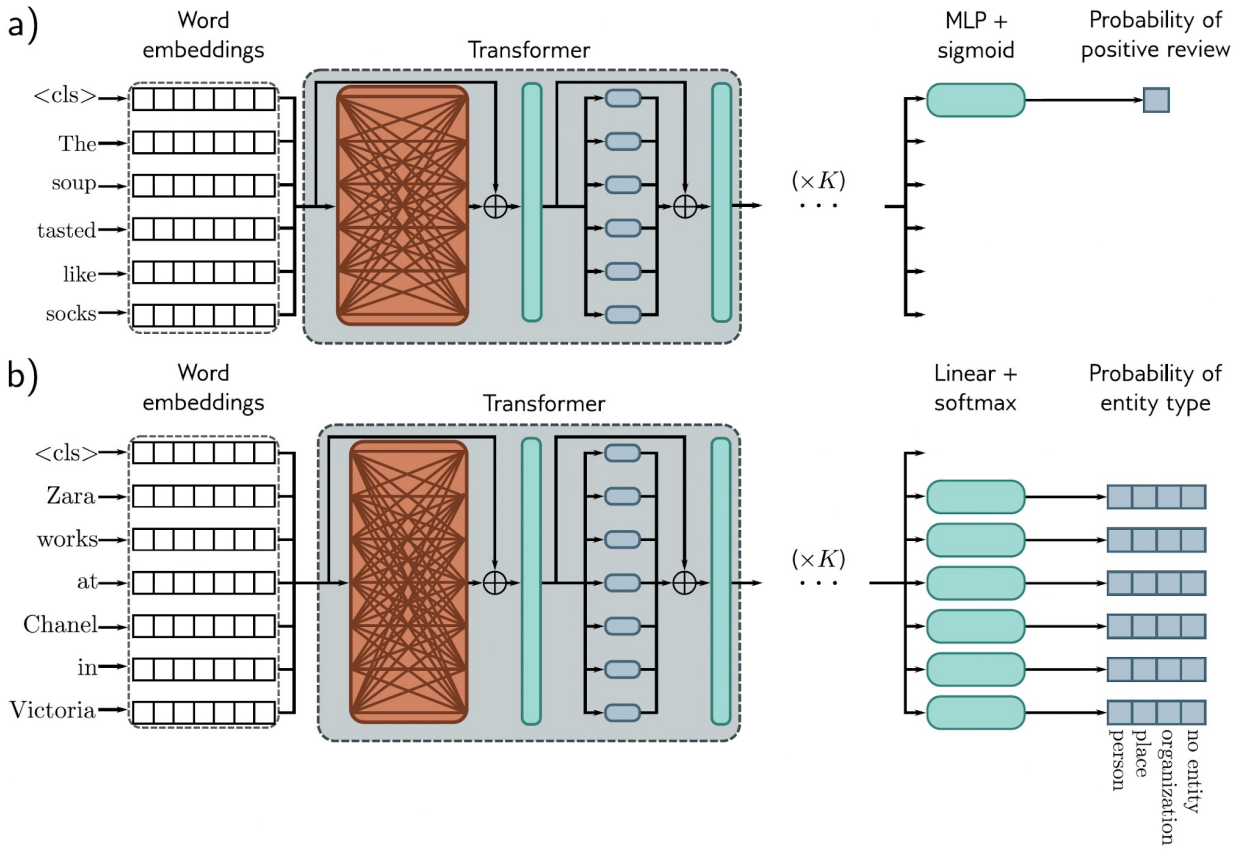


Figure 12.11 After pre-training, the encoder is fine-tuned using manually labeled data to solve a particular task. Usually, a linear transformation or a multi-layer perceptron (MLP) is appended to the encoder to produce whatever output is required. a) Example text classification task. In this sentiment classification task, the <cls> token embedding is used to predict the probability that the review is positive. b) Example word classification task. In this named entity recognition problem, the embedding for each word is used to predict whether the word corresponds to a person, place, or organization, or is not an entity.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

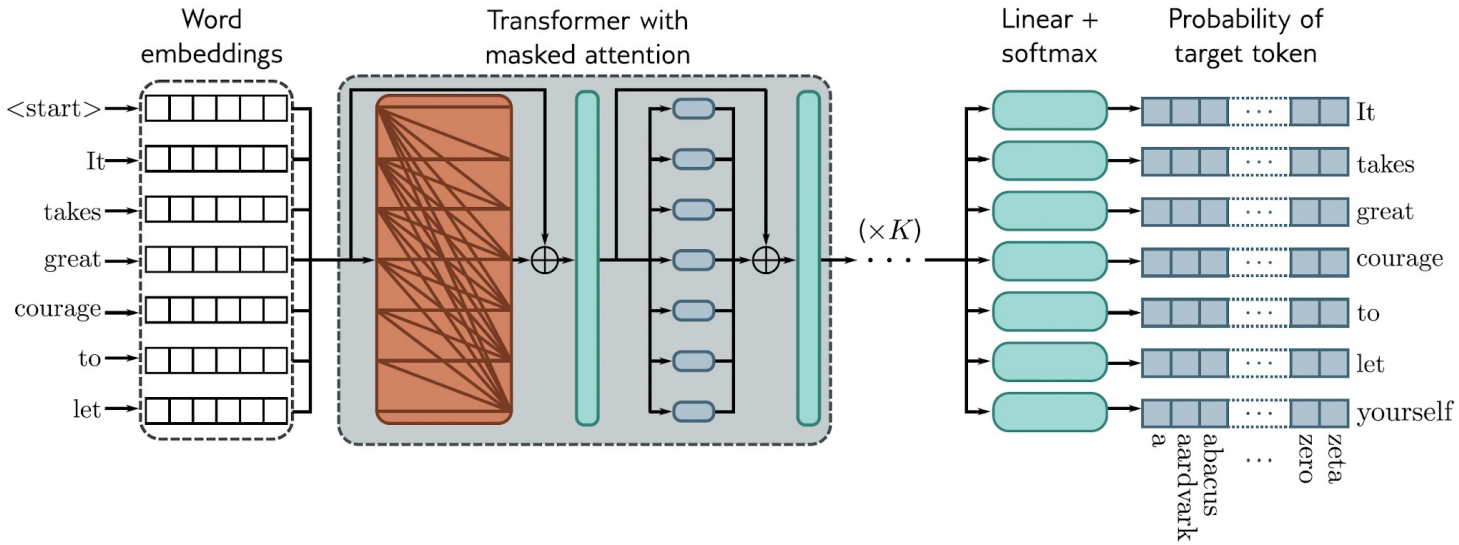


Figure 12.12 Training GPT3-type decoder network. The tokens are mapped to word embeddings with a special <start> token at the beginning of the sequence. The embeddings are passed through a series of transformers that use masked self-attention. Here, each position in the sentence can only attend to its own embedding and the embeddings of tokens earlier in the sequence (orange connections). The goal at each position is to maximize the probability of the following ground truth token in the sequence. In other words, at position one, we want to maximize the probability of the token It; at position two, we want to maximize the probability of the token takes; and so on. Masked self-attention ensures the system cannot cheat by looking at subsequent inputs. The autoregressive task has the advantage of making efficient use of the data since every word contributes a term to the loss function. However, it only exploits the left context of each word.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

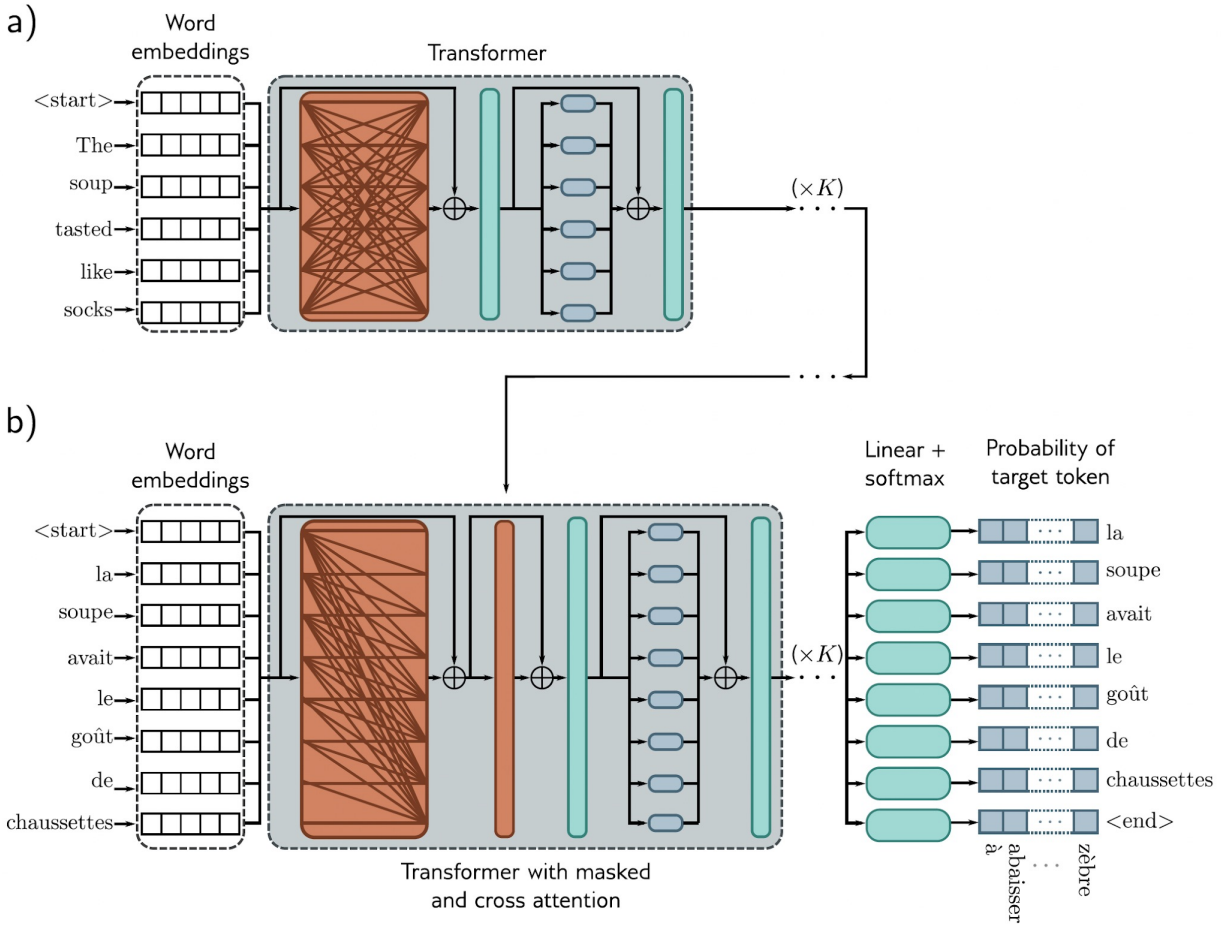


Figure 12.13 Encoder-decoder architecture. Two sentences are passed to the system with the goal of translating the first into the second. a) The first sentence is passed through a standard encoder. b) The second sentence is passed through a decoder that uses masked self-attention but also attends to the output embeddings of the encoder using cross-attention (orange rectangle). The loss function is the same as for the decoder model; we want to maximize the probability of the next word in the output sequence.

Source: Understanding Deep Learning, S. Prince, 2023

Attention and Transformers

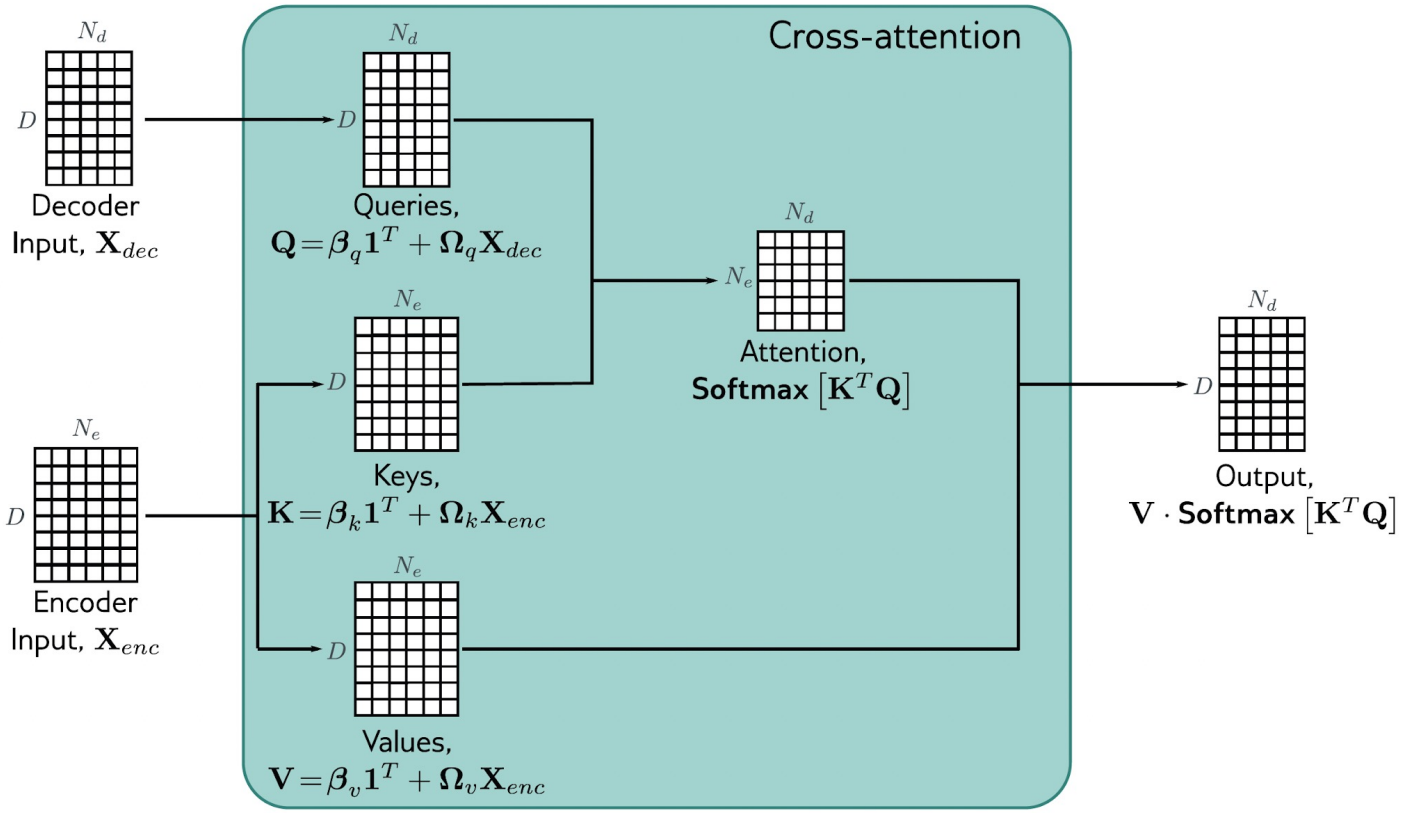


Figure 12.14 Cross-attention. The flow of computation is the same as in standard self-attention. However, the queries are now calculated from the decoder embeddings $\mathbf{X}_{dec}$, and the keys and values from the encoder embeddings $\mathbf{X}_{enc}$.

Source: Understanding Deep Learning, S. Prince, 2023

Tensor computation

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

 18    47  56    57     22    13
```

Tensor computation

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

Tensor computation

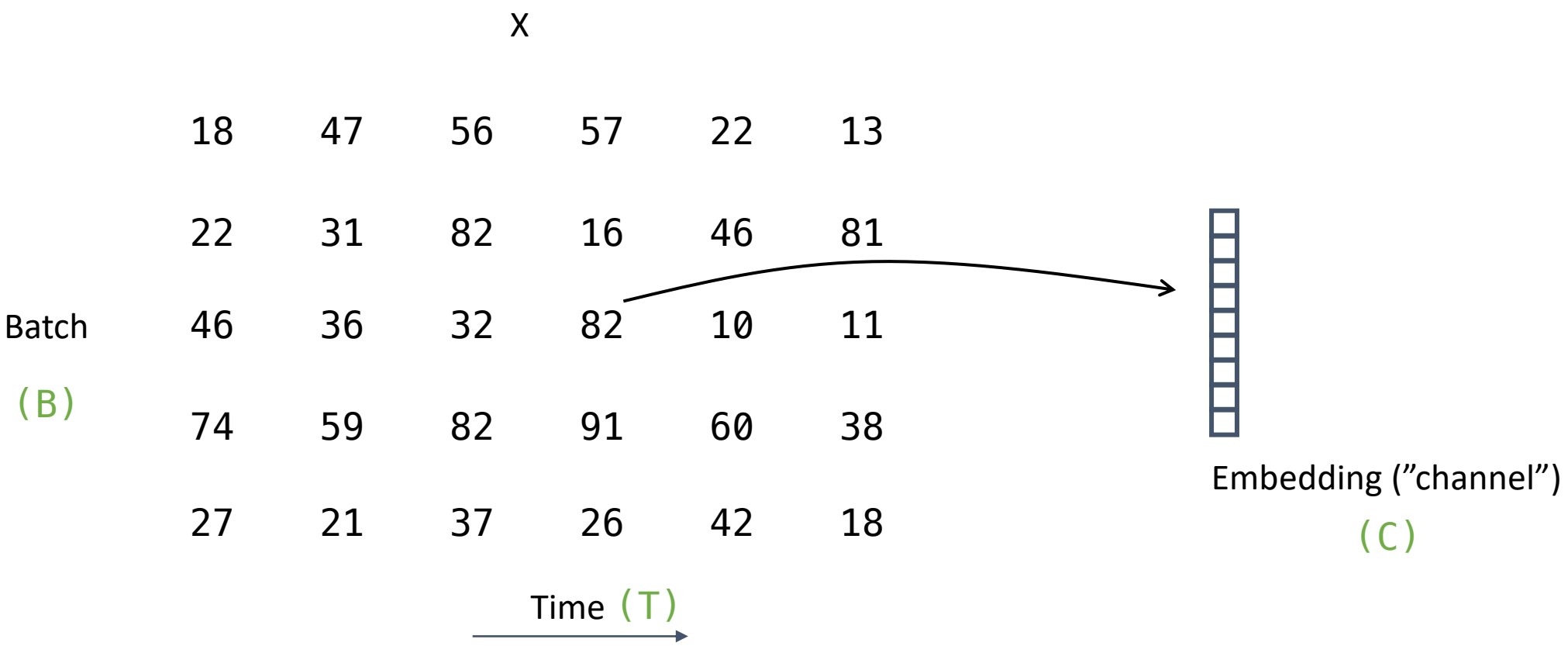
	X						Y					
	18	47	56	57	22	13	47	56	57	22	13	42
	22	31	82	16	46	81	31	82	16	46	81	32
Batch	46	36	32	82	10	11	36	32	82	10	11	69
	74	59	82	91	60	38	59	82	91	60	38	41
	27	21	37	26	42	18	21	37	26	42	18	77

Tensor computation

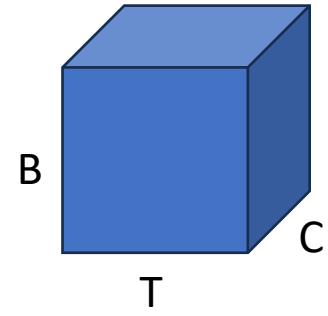
X

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

Tensor computation



Tensor computation



A batch is a 3D tensor

X

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

Time (T)

Embedding (”channel”)

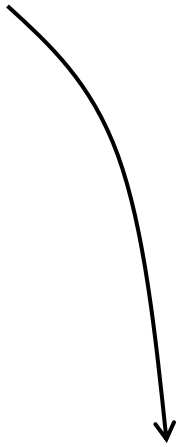
(C)

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

Brodcasting 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)
```

Masked self-attention

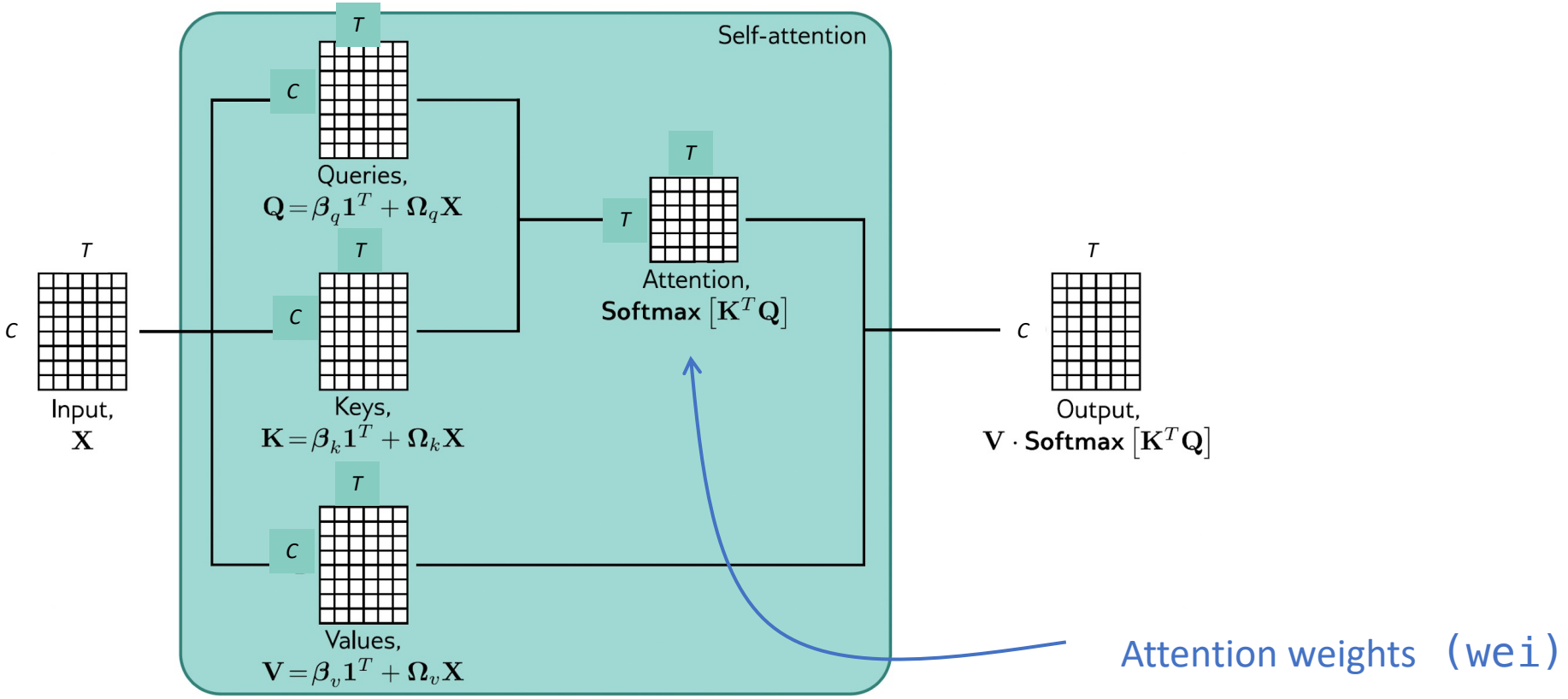


Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

Source: Understanding Deep Learning, S. Prince, 2023

Masked self-attention

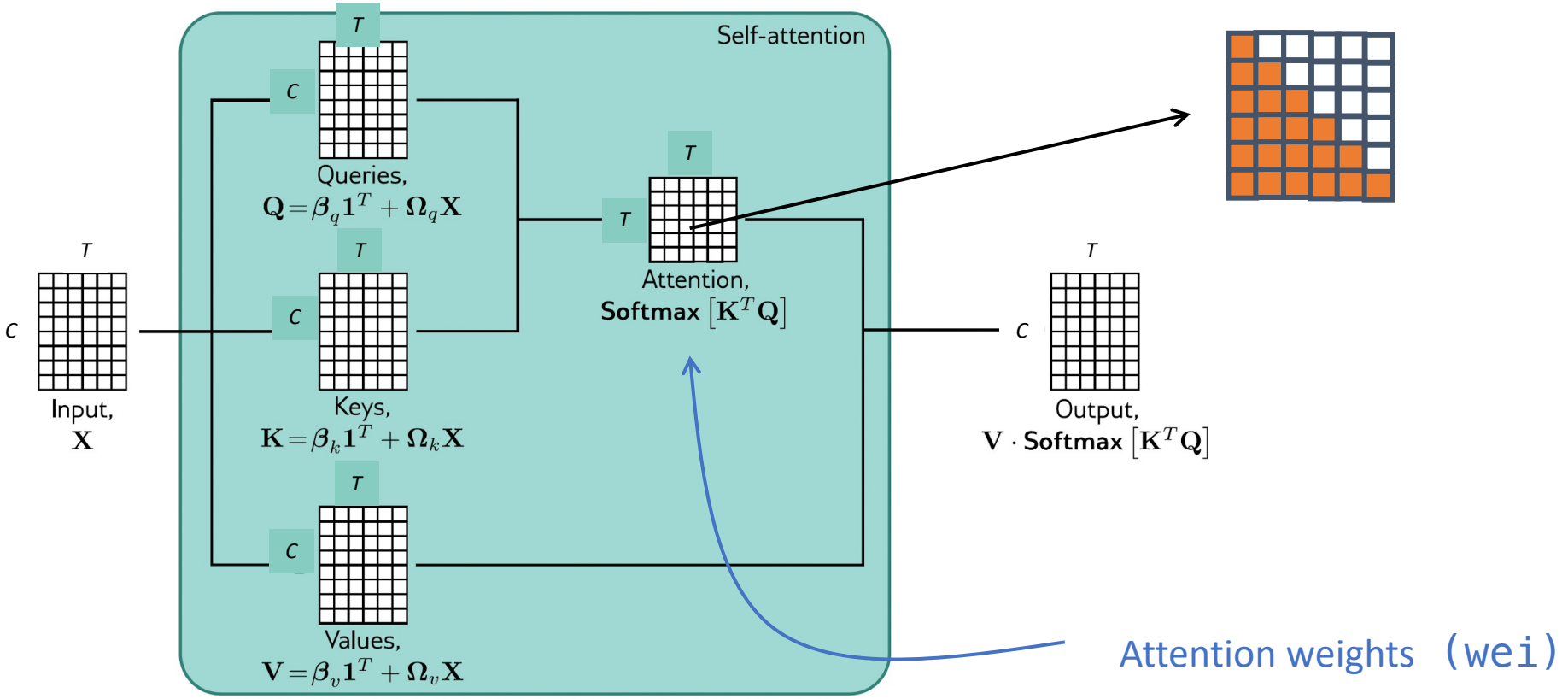


Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

Source: Understanding Deep Learning, S. Prince, 2023

Tensor computation

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

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

Tensor computation

```
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.]])
```

Tensor computation

```
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 computation

```
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]])
```

Tensor computation

When W_Q is a learnable parameter,

Q = W_Q @ X

is implemented as:

```
W_Q = nn.Linear(m, n, bias=False)

Q = W_Q(X)
```

Tensor computation

```
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)
        ...
```

Tensor computation

```
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)
        ...
```

Transformer architecture

Encoder

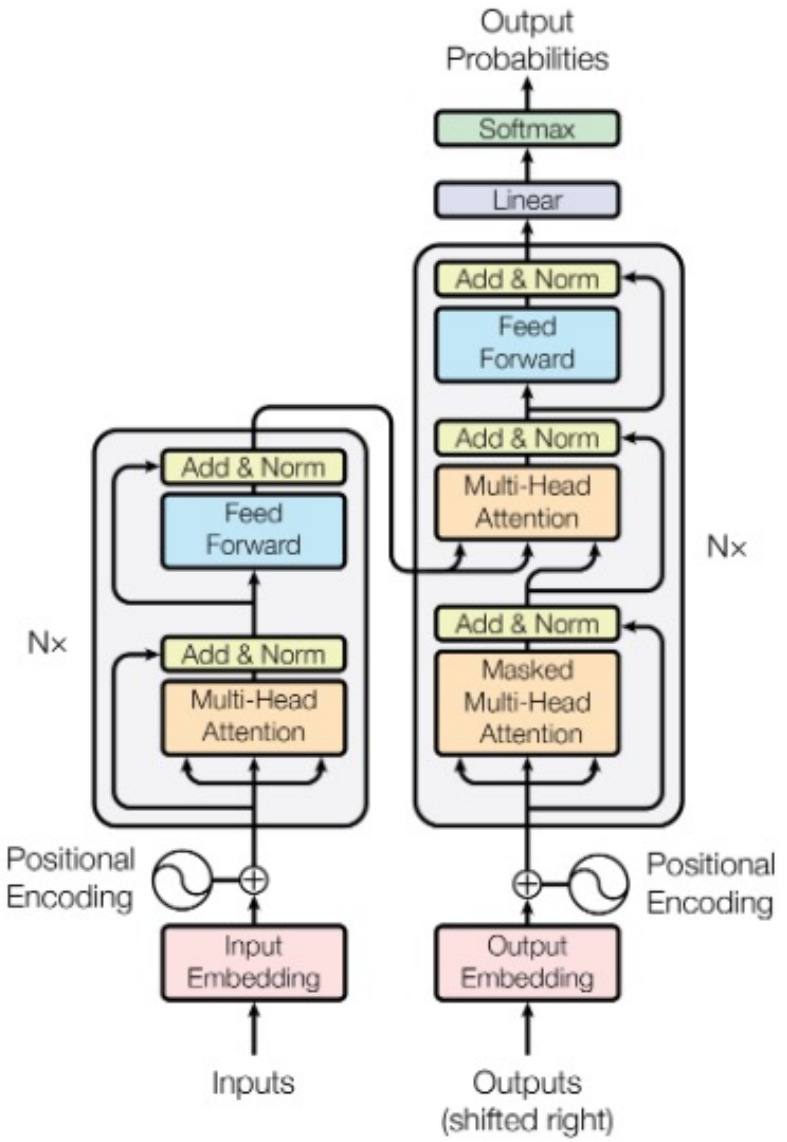


Figure 1: The Transformer - model architecture.

Decoder

Source: Attention is All you Need

TP4: build a mini GPT from scratch

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)