Attention is All You Need

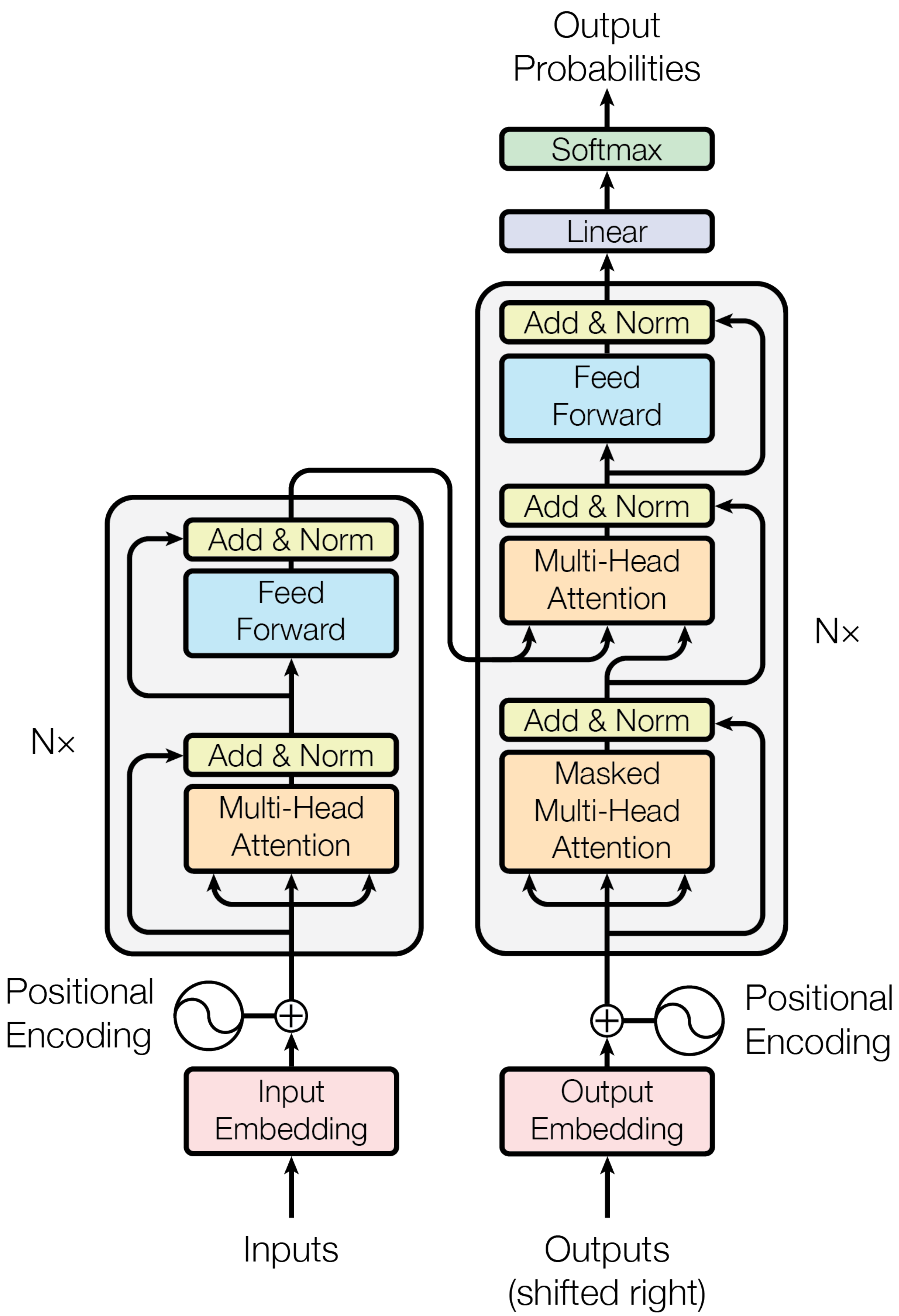
Transformer in details

Encoder Part

Guangyi Liu
CUHK-Shenzhen

Outlines

- Notations
- Embedding Layer
- Positional Encoding
- Multi-Head Attention
- Position-wise Feed Forward
- Residual Connection
- Encoder Block
- Encoder



Notations

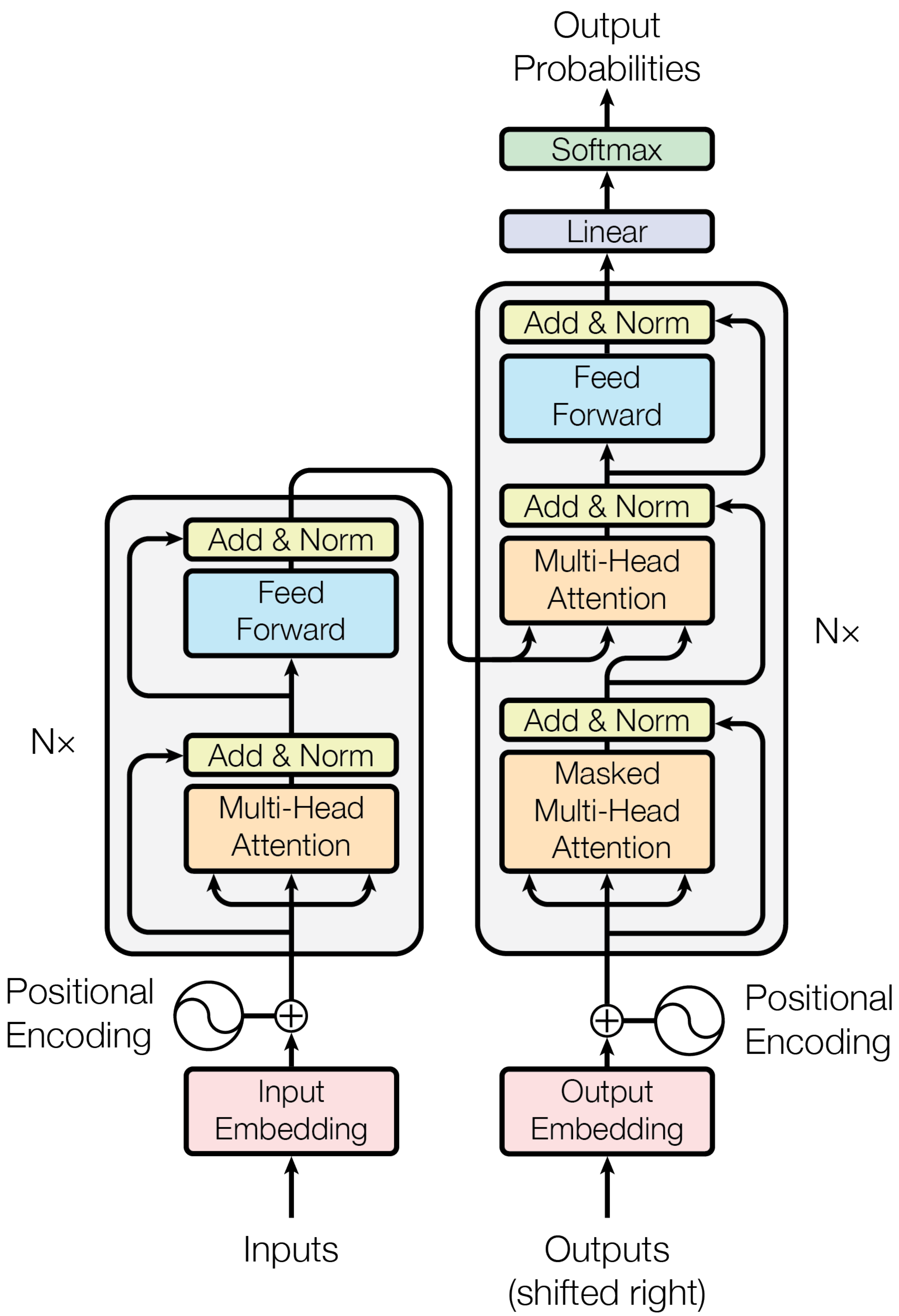
V : **V**ocabulary Size

L : Sequence **L**ength

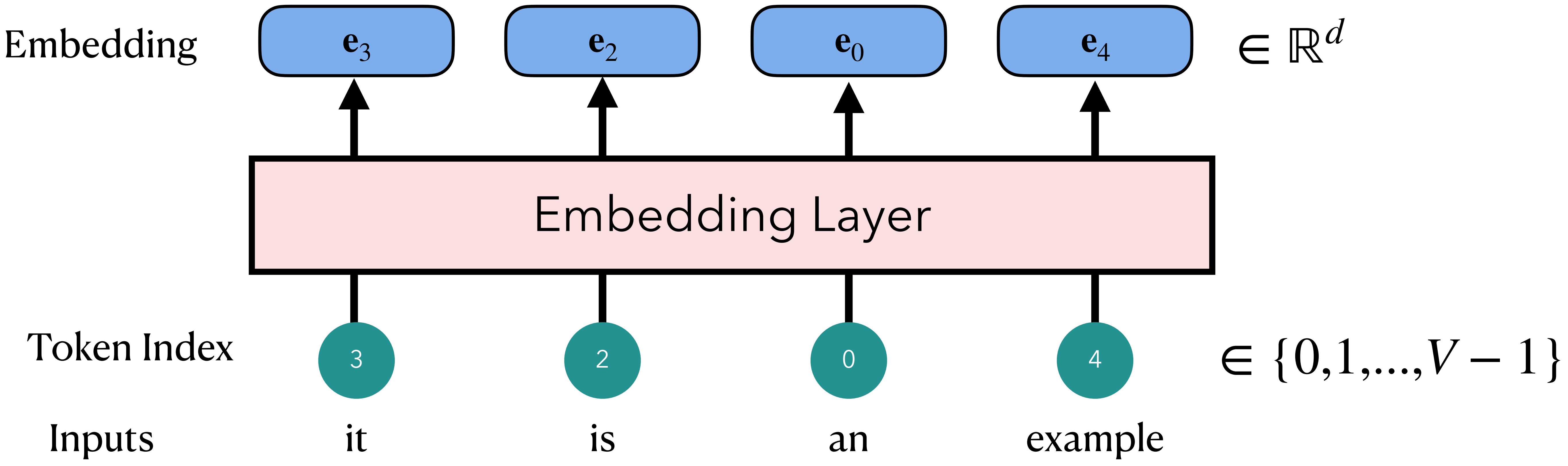
B: **B**atch Size

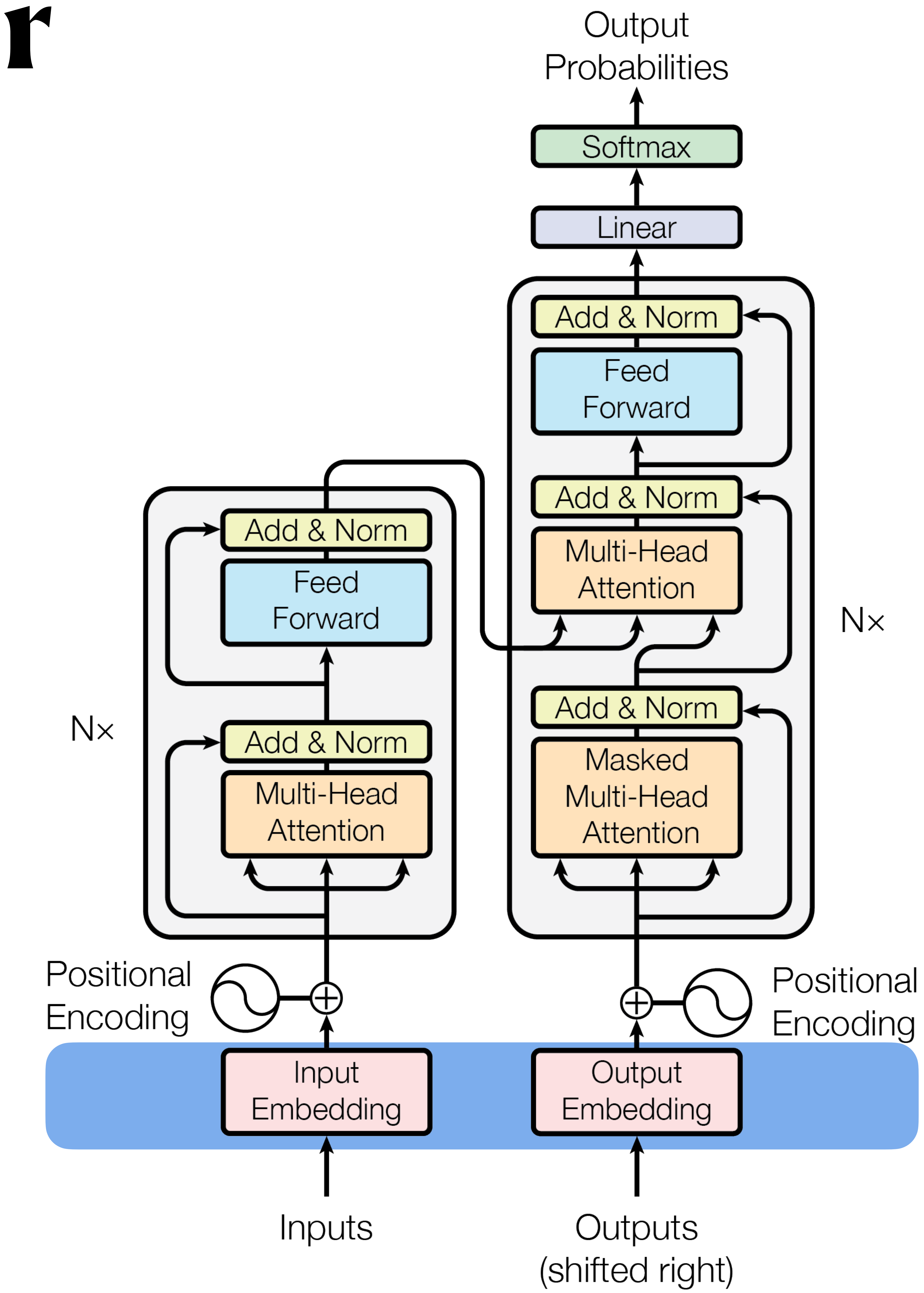
N: Number of Encoder Blocks

d : **D**imension of Hidden/Embedding



Embedding Layer





Goal: Convert the token indexes to vectors of dimension d

A lookup table that is equivalent to a linear layer.

Embedding matrix: $\mathbf{E} \in \mathbb{R}^{V \times d}$

Embedding of the i-th token:

i-th row of $\mathbf{E}$: $\mathbf{e}_i = \mathbf{E}[i, \cdot\,]^T \in \mathbb{R}^d$

Reference : Transformer Coding Details – A Simple Implementation, https://kikaben.com/transformers-coding-details

Vocabulary	Index
an	0
a	1
is	2
it	3
example	4

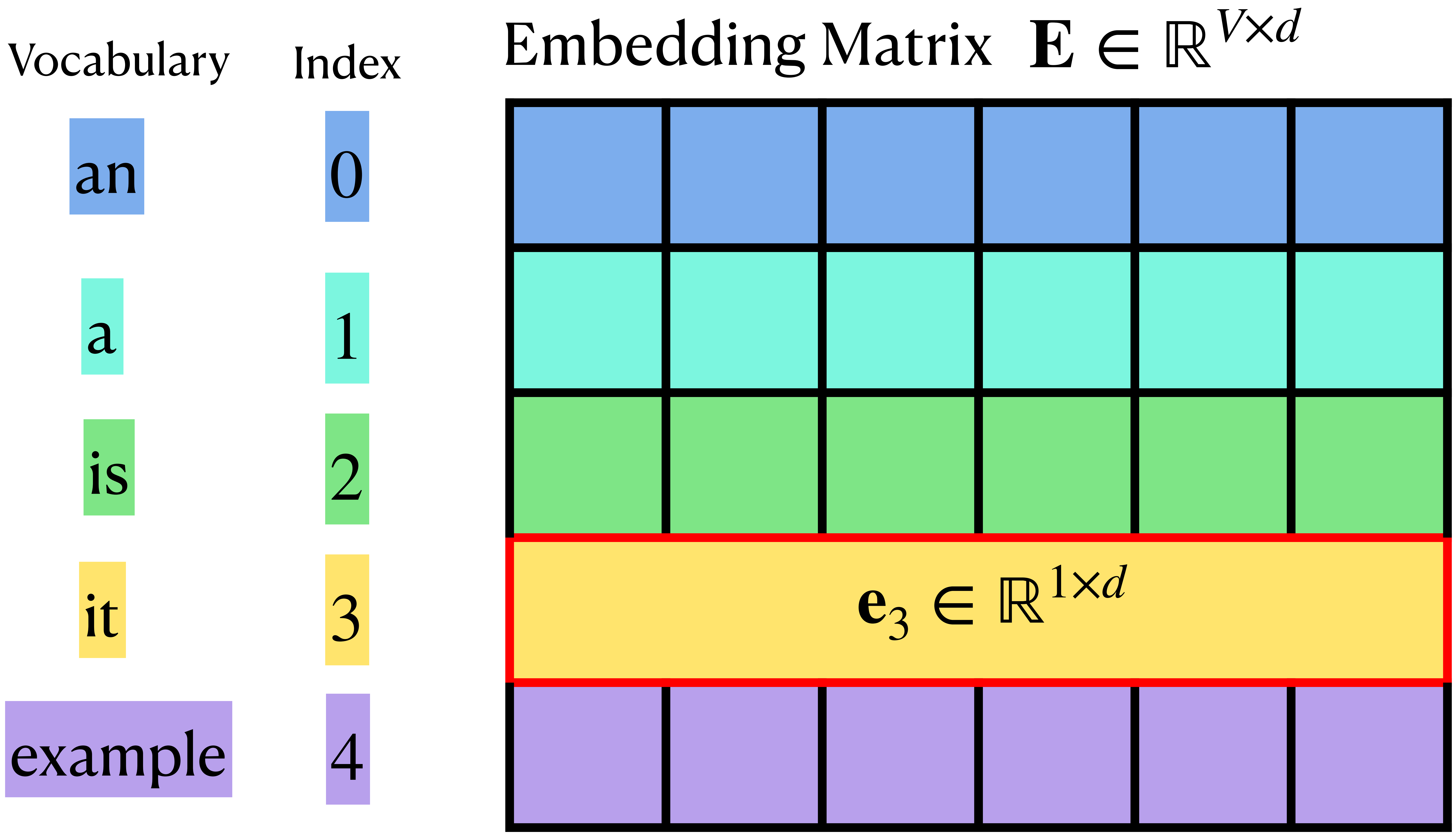
Embedding Matrix $\mathbf{E} \in \mathbb{R}^{V \times d}$

$\mathbf{e}_3 \in \mathbb{R}^{1 \times d}$

V: Vocabulary Size

d: Hidden Dimension

Embedding Matrix $\mathbf{E} \in \mathbb{R}^{V\times d}$



Vocabulary	Index
an	0
a	1
is	2
it	3
example	4

[it, is, an, example]

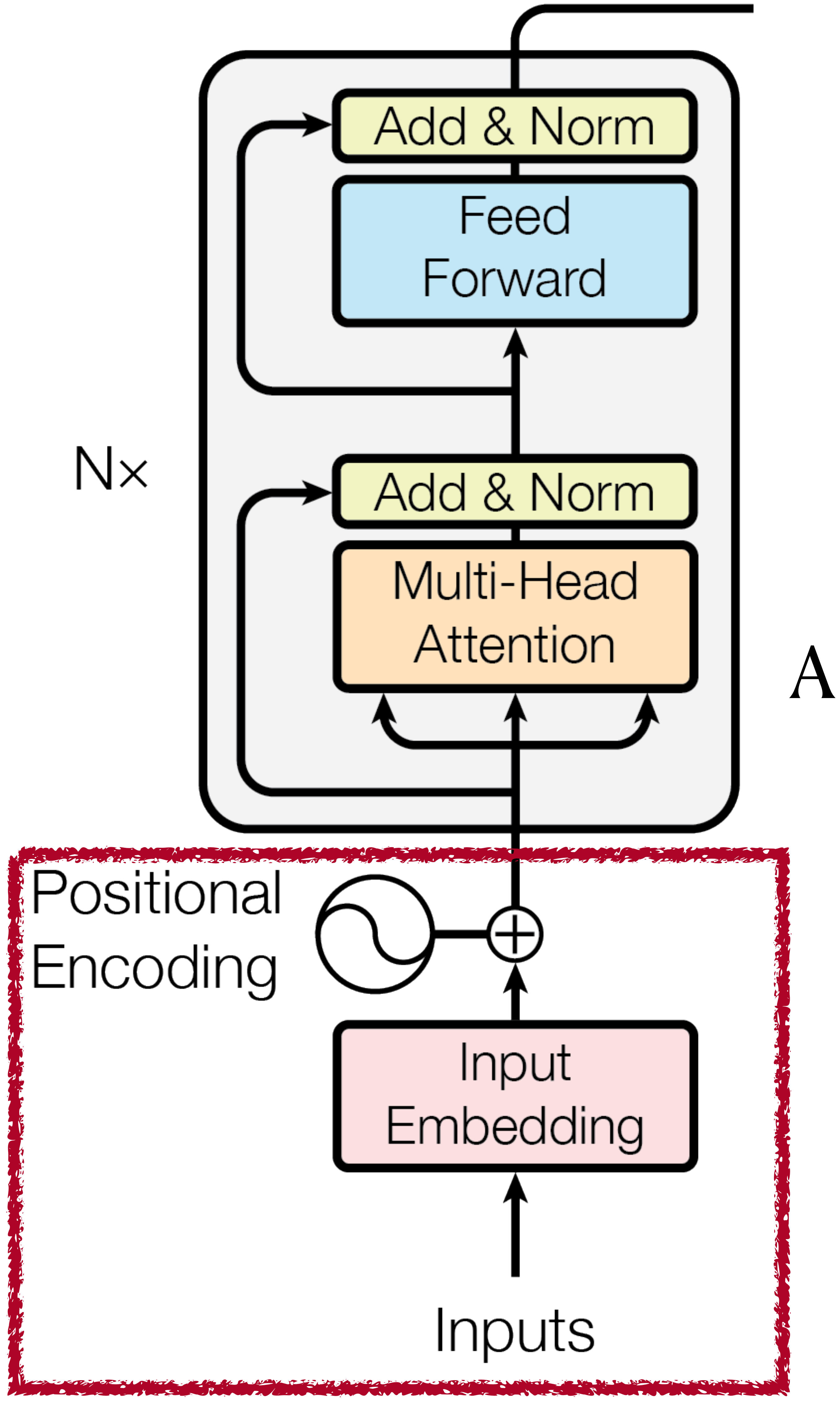
Input list: [3,2,0,4]

$$\mathbf{e}_i = \mathbf{E}[i], i \in \{3,2,0,4\}$$

$$\mathbf{X} = \begin{bmatrix} \mathbf{e}_3 \\ \mathbf{e}_2 \\ \mathbf{e}_0 \\ \mathbf{e}_4 \end{bmatrix} \in \mathbb{R}^{L\times d}$$

Batch: $\mathbf{X} \in \mathbb{R}^{B\times L\times d}$

$$\mathbf{X} = \sqrt{d} \cdot \mathbf{X}$$



$\sqrt{d}$ for ***Rescaling***

Initialization: $\mathcal{N}(\mathbf{0}, \mathbf{I})$

The Same Scale as PE

(PE deterministic)

Assign larger weight on word embedding

If PE is **trainable**

No need rescaling

(e.g., BERT, GPT2)

But: Max Length

[it, is, an, example]

Input list: [3,2,0,4]

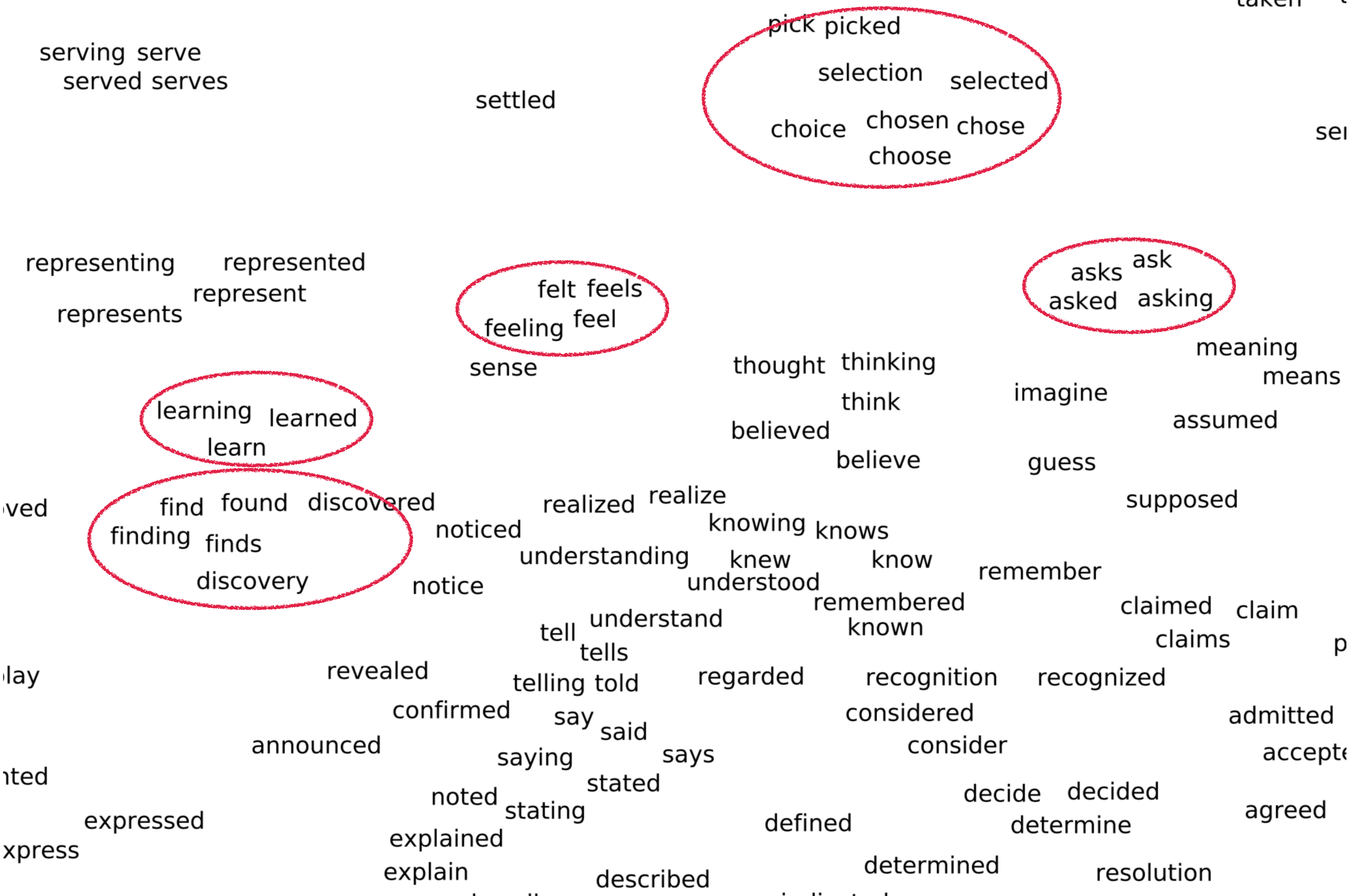
$$\mathbf{e}_i = \mathbf{E}[i], i \in \{3,2,0,4\}$$

$$\mathbf{X} = \begin{bmatrix} \mathbf{e}_3 \\ \mathbf{e}_2 \\ \mathbf{e}_0 \\ \mathbf{e}_4 \end{bmatrix} \in \mathbb{R}^{L \times d}$$

Batch: $\mathbf{X} \in \mathbb{R}^{B \times L \times d}$

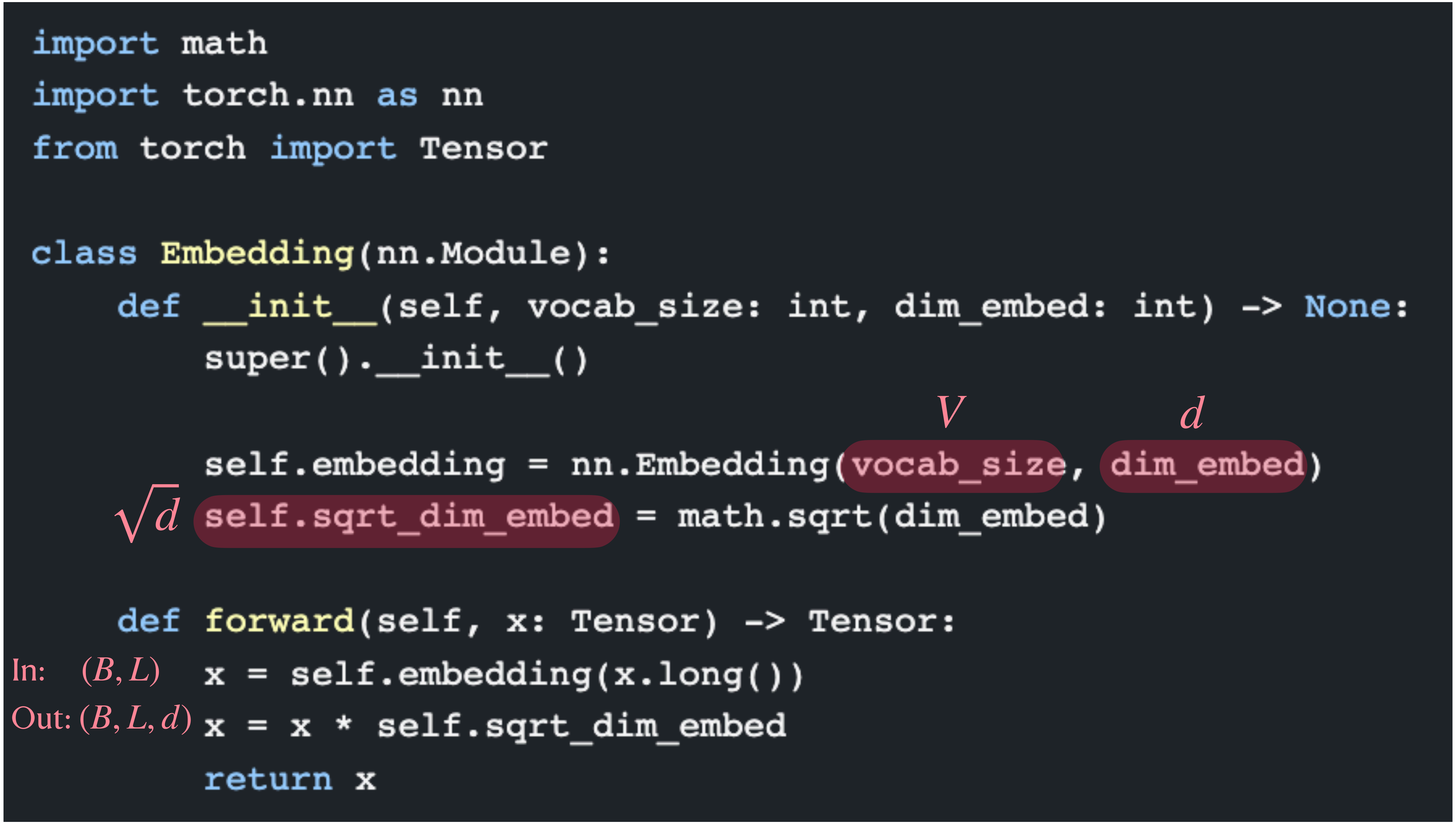
$$\mathbf{X} = \sqrt{d} \cdot \mathbf{X}$$

t-SNE



taken
serving serve
served serves
settled
pick picked
selection selected
choice chosen chose
choose
ser
representing represented
represent
represents
felt feels
feeling feel
sense
asks ask
asked asking
meaning
means
thought thinking
think
imagine
believed
believe
assumed
guess
learning learned
learn
ved
find found discovered
finding finds
discovery
noticed
notice
realized realize
knowing knows
understanding
knew know
understood
remember
remembered
known
supposed
claimed claim
claims
p
tell understand
tells
lay
revealed
telling told
regarded
recognition
recognized
confirmed
say said
announced
saying
says
considered
consider
admitted
accepte
nted
stated
noted
stating
decide decided
determine
agreed
defined
expressed
xpress
explained
explain
described
determined
resolution

Embedding Layer



```
import math
import torch.nn as nn
from torch import Tensor

class Embedding(nn.Module):
    def __init__(self, vocab_size: int, dim_embed: int) -> None:
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, dim_embed)
        self.sqrt_dim_embed = math.sqrt(dim_embed)

    def forward(self, x: Tensor) -> Tensor:
        x = self.embedding(x.long())
        x = x * self.sqrt_dim_embed
        return x
```

Positional Encoding

Why: No sense of position/order for each word

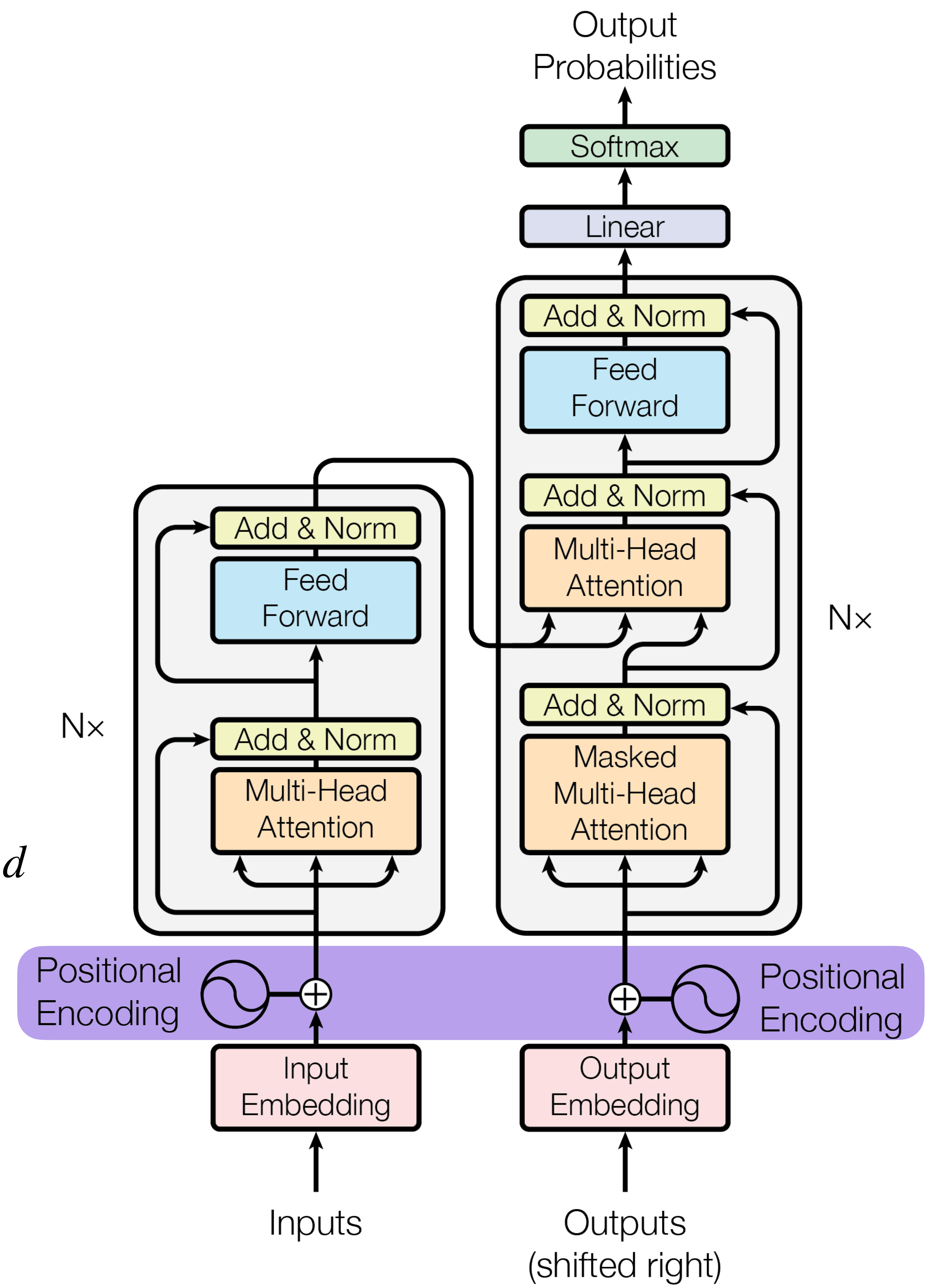
Goal: Insert position information into embedding

Desired Properties:

1. **Unique encoding** for each position
2. **Relative position** should be **consistent**
3. Handle longer sentences without any efforts

Positional vector at position t

$$\mathbf{p}_t = \begin{bmatrix} p_t^{(0)} & p_t^{(1)} & \cdots & p_t^{(i)} & p_t^{(i+1)} & \cdots & p_t^{(d-2)} & p_t^{(d-1)} \end{bmatrix} \in \mathbb{R}^d$$

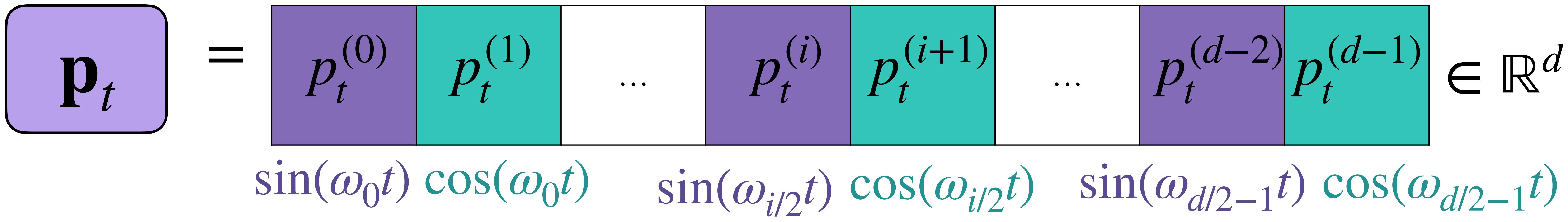


Reference : Transformer Architecture: The Positional Encoding, https://kazemnejad.com/blog/transformer_architecture_positional_encoding

Positional Encoding

Formulation:

$$p_t^{(i)} := \begin{cases} \sin(\omega_k \,.\, t), \text{ if } i = 2k \\ \cos(\omega_k \,.\, t), \text{ if } i = 2k+1 \end{cases} \qquad \omega_k = \left(\frac{1}{10000}\right)^{2k/d}$$

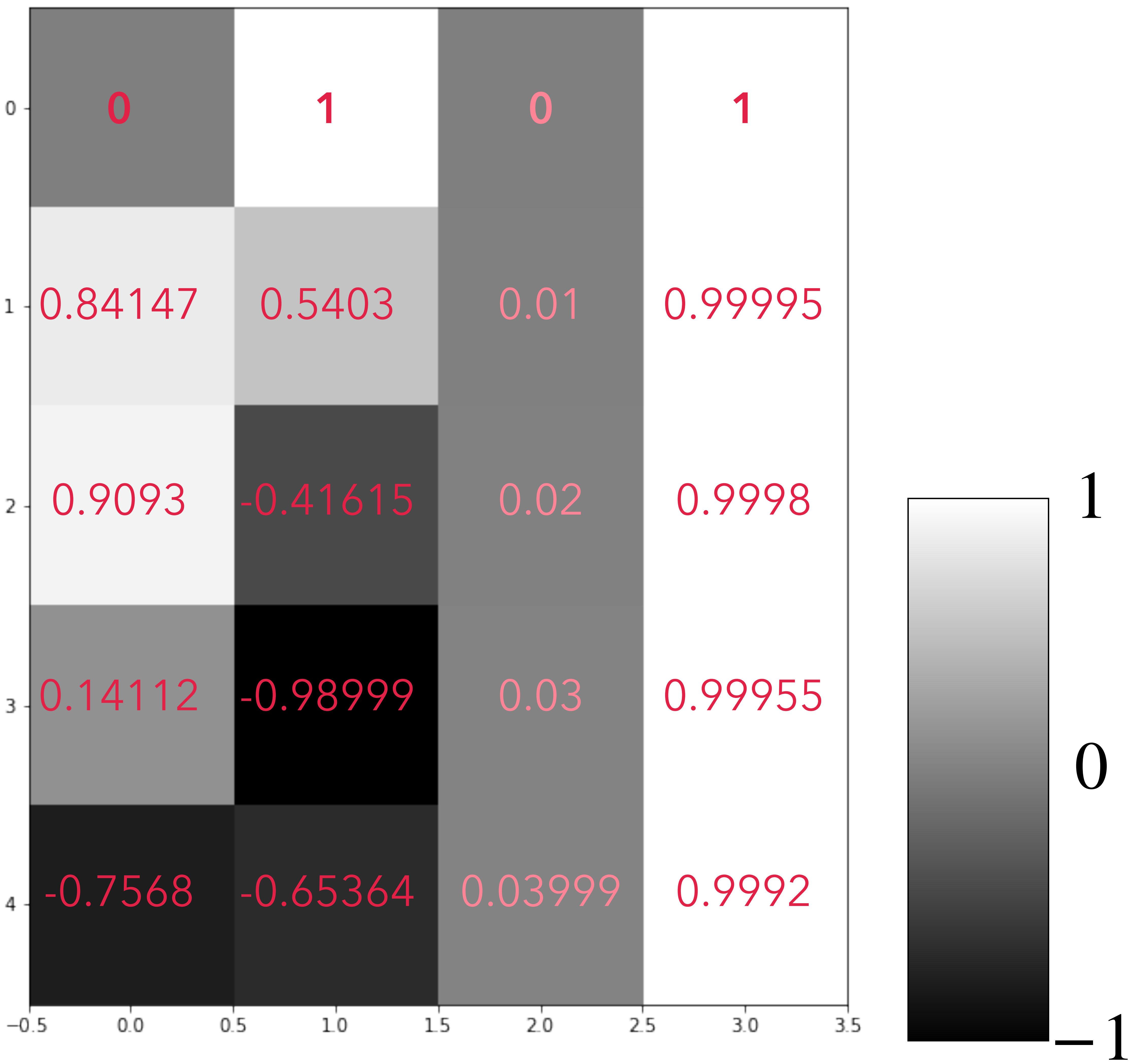


$$\mathbf{p}_t = \begin{bmatrix} p_t^{(0)} & p_t^{(1)} & \dots & p_t^{(i)} & p_t^{(i+1)} & \dots & p_t^{(d-2)} & p_t^{(d-1)} \end{bmatrix} \in \mathbb{R}^d$$

$\sin(\omega_0 t)$ $\cos(\omega_0 t)$ $\sin(\omega_{i/2} t)$ $\cos(\omega_{i/2} t)$ $\sin(\omega_{d/2-1} t)$ $\cos(\omega_{d/2-1} t)$

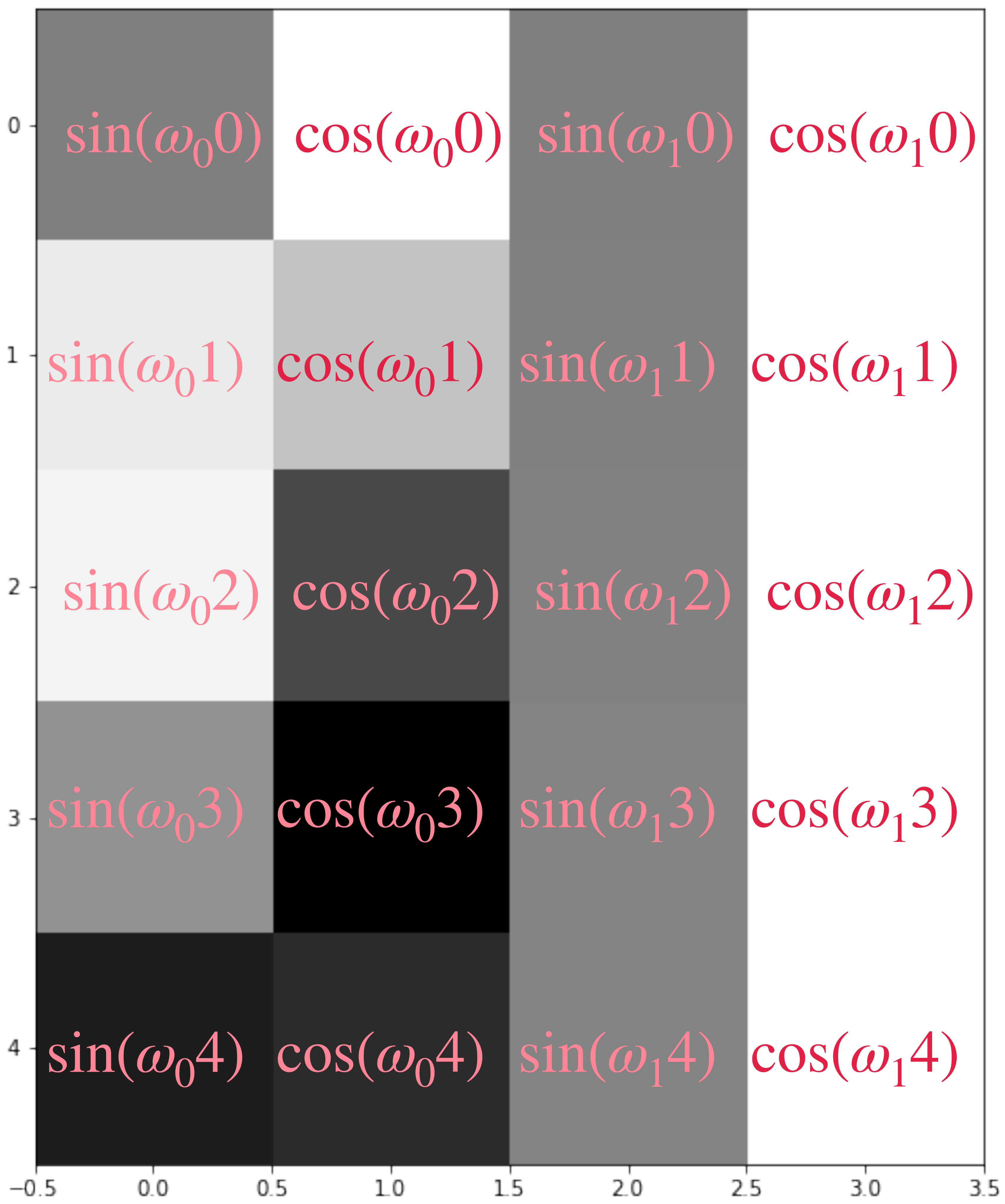
$$\omega_0 = \left(\frac{1}{10000}\right)^{0} = 1 \qquad \omega_{i/2} = \left(\frac{1}{10000}\right)^{i/d} \qquad \omega_{d/2-1} = \left(\frac{1}{10000}\right)^{1-2/d}$$

$d = 4$
$L = 5$



Position \ Dimension:	$i = 0$	$i = 1$	$i = 2$	$i = 3$
$t = 0$	0	1	0	1
$t = 1$	0.84147	0.5403	0.01	0.99995
$t = 2$	0.9093	-0.41615	0.02	0.9998
$t = 3$	0.14112	-0.98999	0.03	0.99955
$t = 4$	-0.7568	-0.65364	0.03999	0.9992
	$k = 0$	$k = 0$	$k = 1$	$k = 1$

$\sin(\omega_k t)$ $\cos(\omega_k t)$

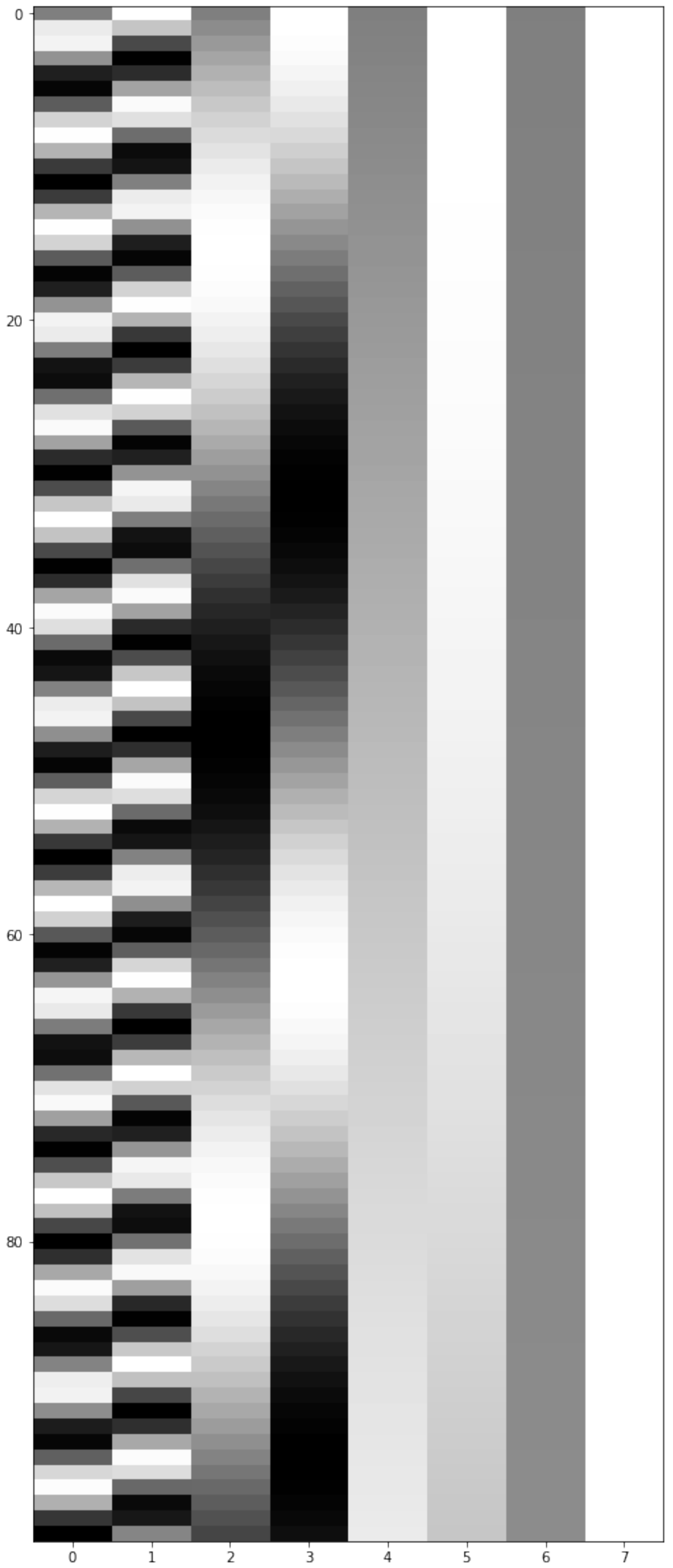


$t = 0$	$\sin(\omega_0 0)$	$\cos(\omega_0 0)$	$\sin(\omega_1 0)$	$\cos(\omega_1 0)$
$t = 1$	$\sin(\omega_0 1)$	$\cos(\omega_0 1)$	$\sin(\omega_1 1)$	$\cos(\omega_1 1)$
$t = 2$	$\sin(\omega_0 2)$	$\cos(\omega_0 2)$	$\sin(\omega_1 2)$	$\cos(\omega_1 2)$
$t = 3$	$\sin(\omega_0 3)$	$\cos(\omega_0 3)$	$\sin(\omega_1 3)$	$\cos(\omega_1 3)$
$t = 4$	$\sin(\omega_0 4)$	$\cos(\omega_0 4)$	$\sin(\omega_1 4)$	$\cos(\omega_1 4)$
	$k = 0$	$k = 0$	$k = 1$	$k = 1$

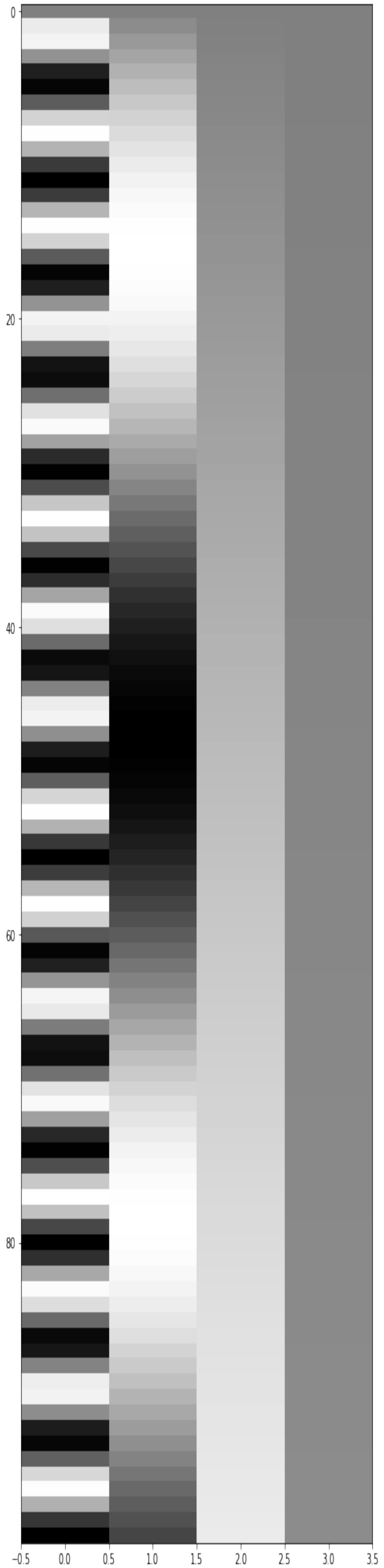
$k \uparrow$ $\omega_k \downarrow$ **Frequency** $\downarrow$

$d = 8$

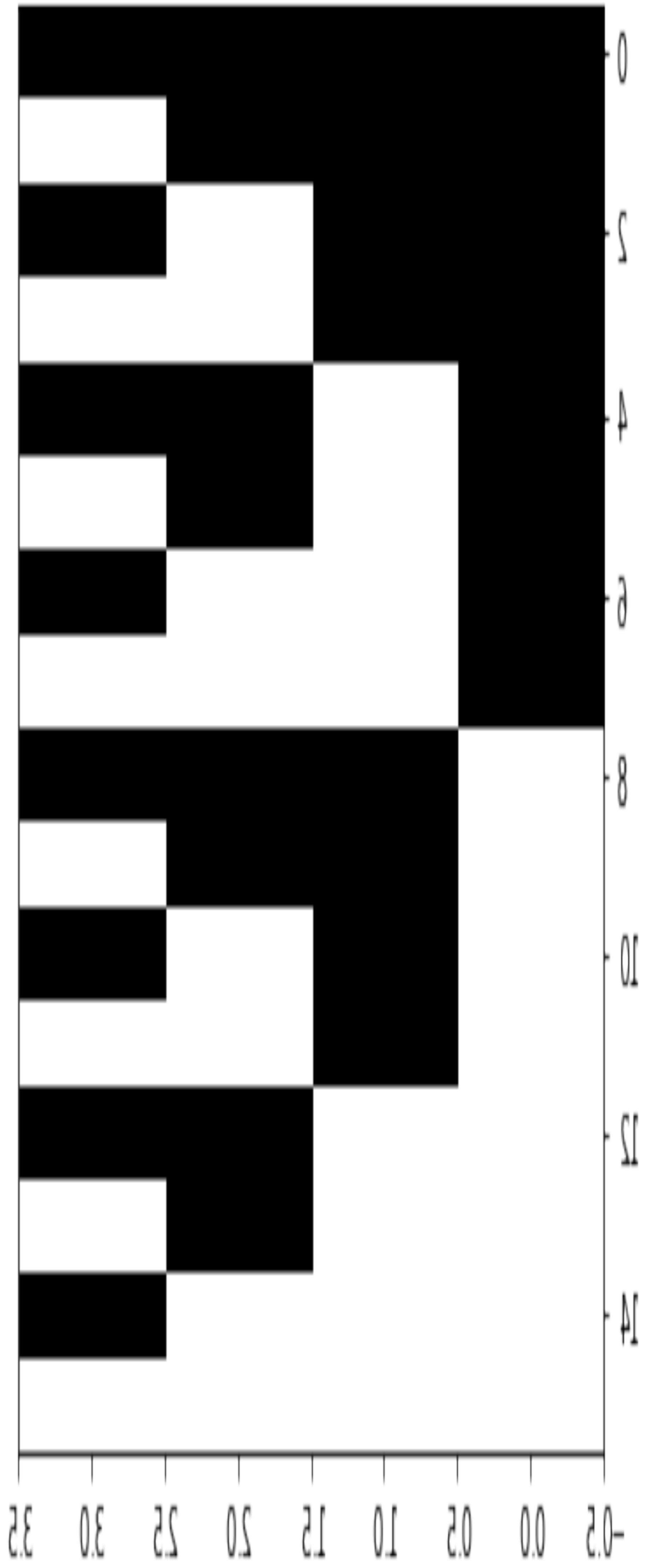
$L = 100$



$k \uparrow \omega_k \downarrow$ **Frequency ↓**



sin part



0 :	0	0	0	0
1 :	0	0	0	1
2 :	0	0	1	0
3 :	0	0	1	1
4 :	0	1	0	0
5 :	0	1	0	1
6 :	0	1	1	0
7 :	0	1	1	1
8 :	1	0	0	0
9 :	1	0	0	1
10 :	1	0	1	0
11 :	1	0	1	1
12 :	1	1	0	0
13 :	1	1	0	1
14 :	1	1	1	0
15 :	1	1	1	1

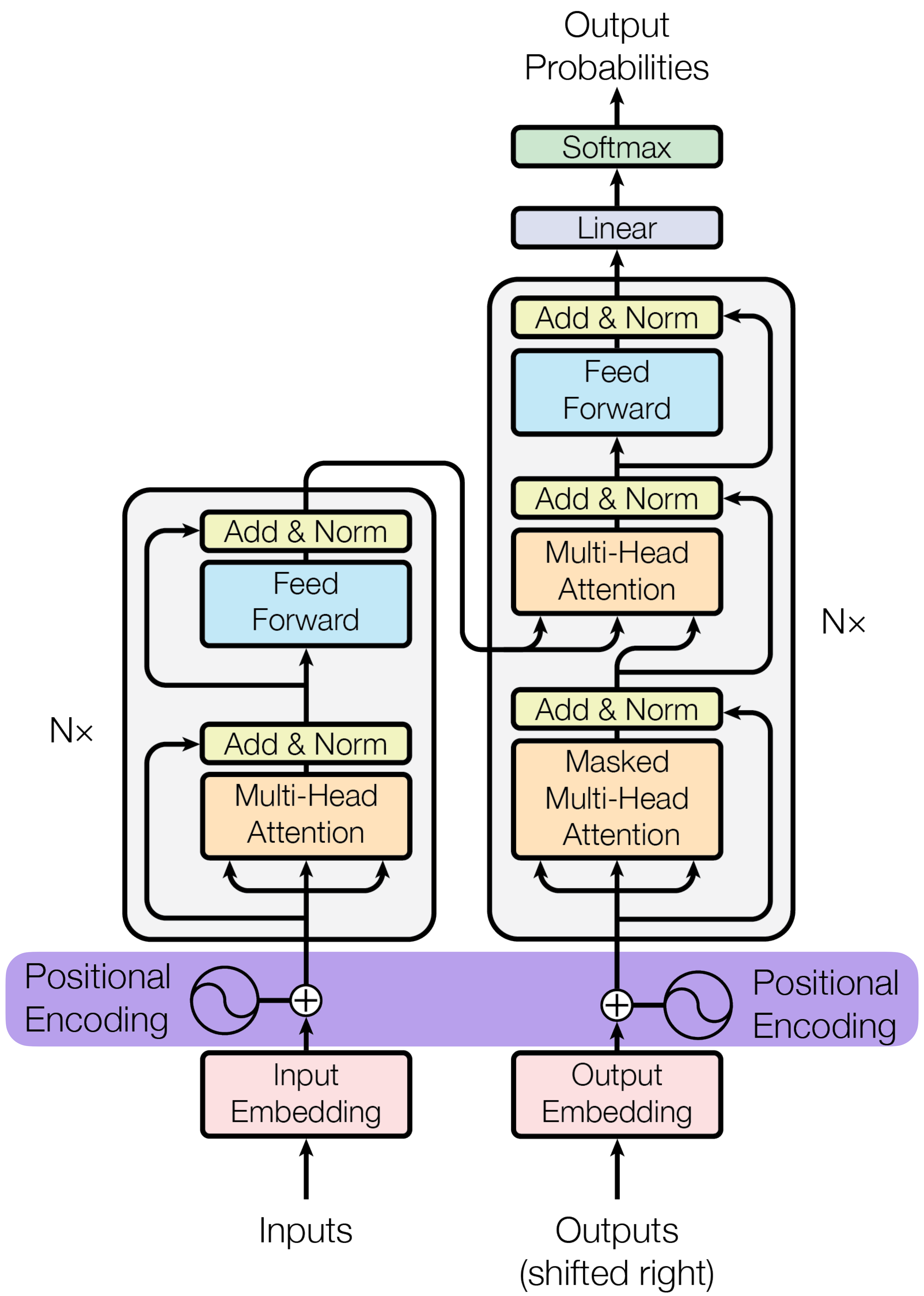
Positional Encoding

$$p_t^{(i)} := \begin{cases} \sin(\omega_k \, . \, t), \text{ if } i = 2k \\ \cos(\omega_k \, . \, t), \text{if } i = 2k + 1 \end{cases}$$

$$\omega_k = \left(\frac{1}{10000} \right)^{\frac{2k}{d}}$$

Why do we use both *sin* and *cos*?

We chose this function because we hypothesized it would allow the model to easily learn to attend by ***relative positions, since for any fixed offset*** ϕ***,*** $\mathbf{p}_{t+\phi}$ ***can be represented as a*** ***<u>linear function</u>*** ***of*** $\mathbf{p}_t$



For every sine-cosine pair corresponding to ω_k , there is a linear transformation $M \in \mathbb{R}^{2\times 2}$ (independent of t) where the following equation holds:

$$M(k, \phi) \cdot \begin{bmatrix} \sin(\omega_k \,.\, t) \\ \cos(\omega_k \,.\, t) \end{bmatrix} = \begin{bmatrix} \sin(\omega_k \,.\, (t + \phi)) \\ \cos(\omega_k \,.\, (t + \phi)) \end{bmatrix}$$

Easy **Proof:**

$$\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix} \cdot \begin{bmatrix} \sin(\omega_k \,.\, t) \\ \cos(\omega_k \,.\, t) \end{bmatrix} = \begin{bmatrix} \sin(\omega_k \,.\, (t + \phi)) \\ \cos(\omega_k \,.\, (t + \phi)) \end{bmatrix}$$

RHS becomes:

$$= \begin{bmatrix} \sin(\omega_k \,.\, t)\cos(\omega_k \,.\, \phi) + \cos(\omega_k \,.\, t)\sin(\omega_k \,.\, \phi) \\ \cos(\omega_k \,.\, t)\cos(\omega_k \,.\, \phi) - \sin(\omega_k \,.\, t)\sin(\omega_k \,.\, \phi) \end{bmatrix}$$

By *addition theorem*, we get:

$$u_1 \sin(\omega_k \,.\, t) + v_1 \cos(\omega_k \,.\, t) = \cos(\omega_k \,.\, \phi)\sin(\omega_k \,.\, t) + \sin(\omega_k \,.\, \phi)\cos(\omega_k \,.\, t)$$
$$u_2 \sin(\omega_k \,.\, t) + v_2 \cos(\omega_k \,.\, t) = -\sin(\omega_k \,.\, \phi)\sin(\omega_k \,.\, t) + \cos(\omega_k \,.\, \phi)\cos(\omega_k \,.\, t)$$

One solution is:

$$u_1 = \cos(\omega_k \,.\, \phi) \quad v_1 = \sin(\omega_k \,.\, \phi)$$
$$u_2 = -\sin(\omega_k \,.\, \phi) \quad v_2 = \cos(\omega_k \,.\, \phi)$$

The final transformation **does not depend on position *t*.**

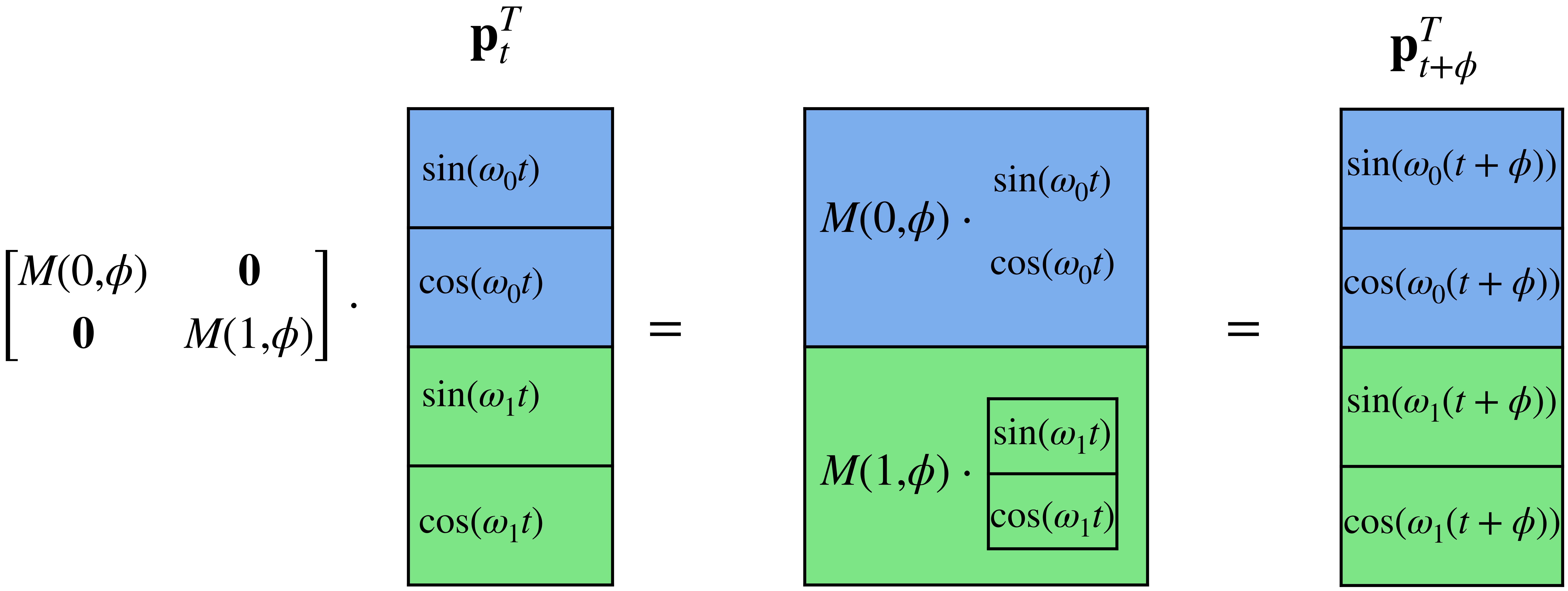
If we only use *sin* or *cos*, **the linear transformation will not hold** (addition theorem).

$$M(k, \phi) = \begin{bmatrix} \cos(\omega_k \,.\, \phi) & \sin(\omega_k \,.\, \phi) \\ -\sin(\omega_k \,.\, \phi) & \cos(\omega_k \,.\, \phi) \end{bmatrix}$$

Positional Encoding

$$M(k,\phi)\cdot\begin{bmatrix}\sin(\omega_k\,.\,t)\\ \cos(\omega_k\,.\,t)\end{bmatrix}=\begin{bmatrix}\sin(\omega_k\,.\,(t+\phi))\\ \cos(\omega_k\,.\,(t+\phi))\end{bmatrix}$$

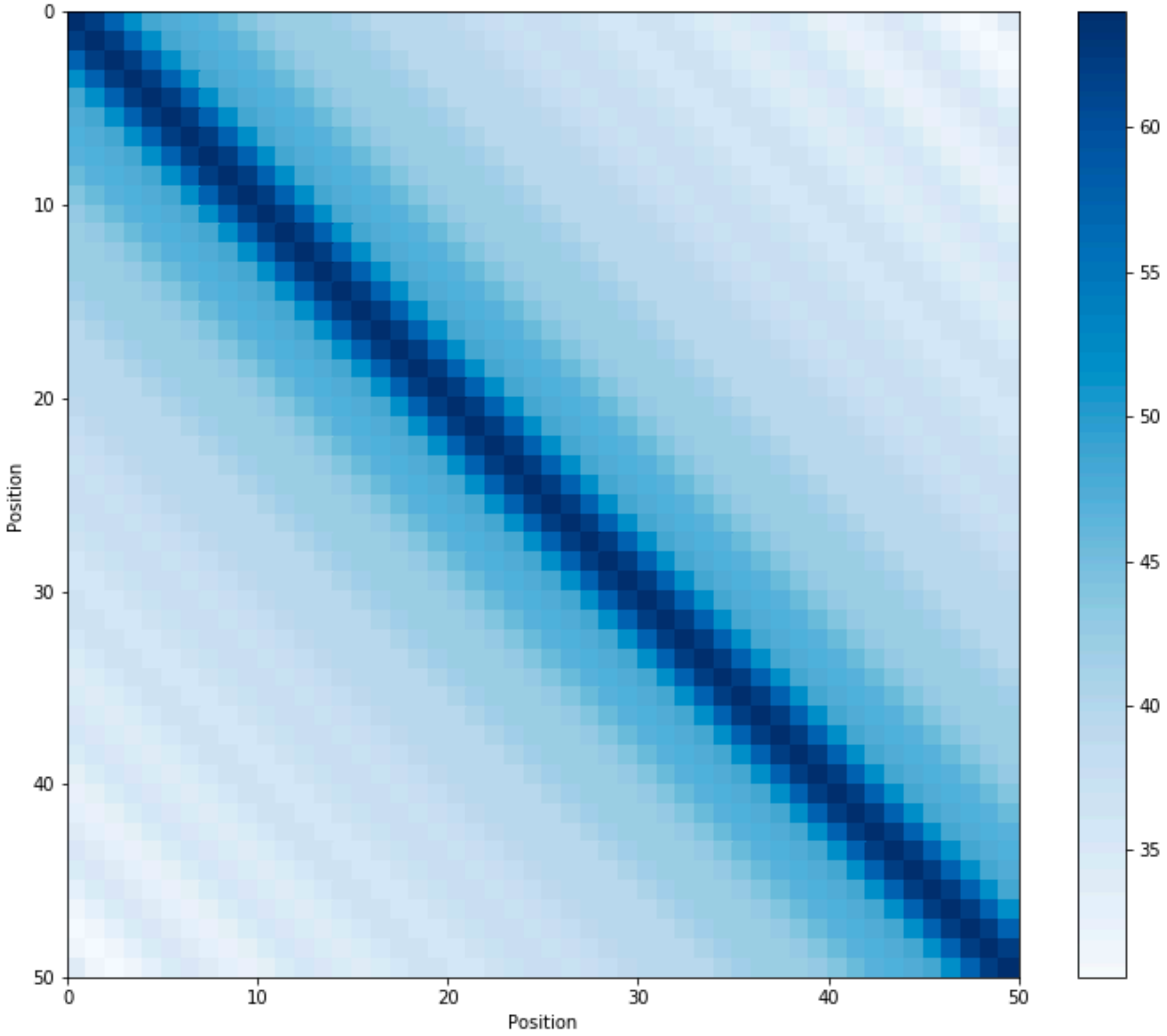
$$M(k,\phi)=\begin{bmatrix}\cos(\omega_k\,.\,\phi) & \sin(\omega_k\,.\,\phi)\\ -\sin(\omega_k\,.\,\phi) & \cos(\omega_k\,.\,\phi)\end{bmatrix}$$



$$\begin{bmatrix}M(0,\phi) & \mathbf{0}\\ \mathbf{0} & M(1,\phi)\end{bmatrix}\cdot\underbrace{\begin{bmatrix}\sin(\omega_0 t)\\ \cos(\omega_0 t)\\ \sin(\omega_1 t)\\ \cos(\omega_1 t)\end{bmatrix}}_{\mathbf{p}_t^T}=\begin{bmatrix}M(0,\phi)\cdot\begin{matrix}\sin(\omega_0 t)\\ \cos(\omega_0 t)\end{matrix}\\ M(1,\phi)\cdot\boxed{\begin{matrix}\sin(\omega_1 t)\\ \cos(\omega_1 t)\end{matrix}}\end{bmatrix}=\underbrace{\begin{bmatrix}\sin(\omega_0(t+\phi))\\ \cos(\omega_0(t+\phi))\\ \sin(\omega_1(t+\phi))\\ \cos(\omega_1(t+\phi))\end{bmatrix}}_{\mathbf{p}_{t+\phi}^T}$$

Positional Encoding

Distance between neighboring positions are **symmetrical** and **decays nicely** with positions.



Dot product of position embeddings for all positions

$$\mathbf{p}_t \cdot \mathbf{p}_{t+\phi} = \sum_k \left(\sin \omega_k t \cdot \sin \omega_k (t+\phi) + \cos \omega_k t \cdot \cos \omega_k (t+\phi)\right)$$

$$\mathbf{p}_t \cdot \mathbf{p}_{t-\phi} = \sum_k \left(\sin \omega_k t \cdot \sin \omega_k (t-\phi) + \cos \omega_k t \cdot \cos \omega_k (t-\phi)\right)$$

$$\begin{aligned}\sin \omega_k t \cdot \sin \omega_k (t+\phi) &= \sin \omega_k t \cdot (\sin \omega_k t \cos \omega_k \phi + \cos \omega_k t \sin \omega_k \phi) \\ &= \sin^2 \omega_k t \cos \omega_k \phi + \sin \omega_k t \cos \omega_k t \sin \omega_k \phi\end{aligned}$$

$$\begin{aligned}\cos \omega_k t \cdot \cos \omega_k (t+\phi) &= \cos \omega_k t \cdot (\cos \omega_k t \cos \omega_k \phi - \sin \omega_k t \sin \omega_k \phi) \\ &= \cos^2 \omega_k t \cos \omega_k \phi - \cos \omega_k t \sin \omega_k t \sin \omega_k \phi\end{aligned}$$

$$\begin{aligned}&\sin \omega_k t \cdot \sin \omega_k (t+\phi) + \cos \omega_k t \cdot \cos \omega_k (t+\phi) \\ &\quad = \sin^2 \omega_k t \cos \omega_k \phi + \cos^2 \omega_k t \sin \omega_k \phi\end{aligned}$$

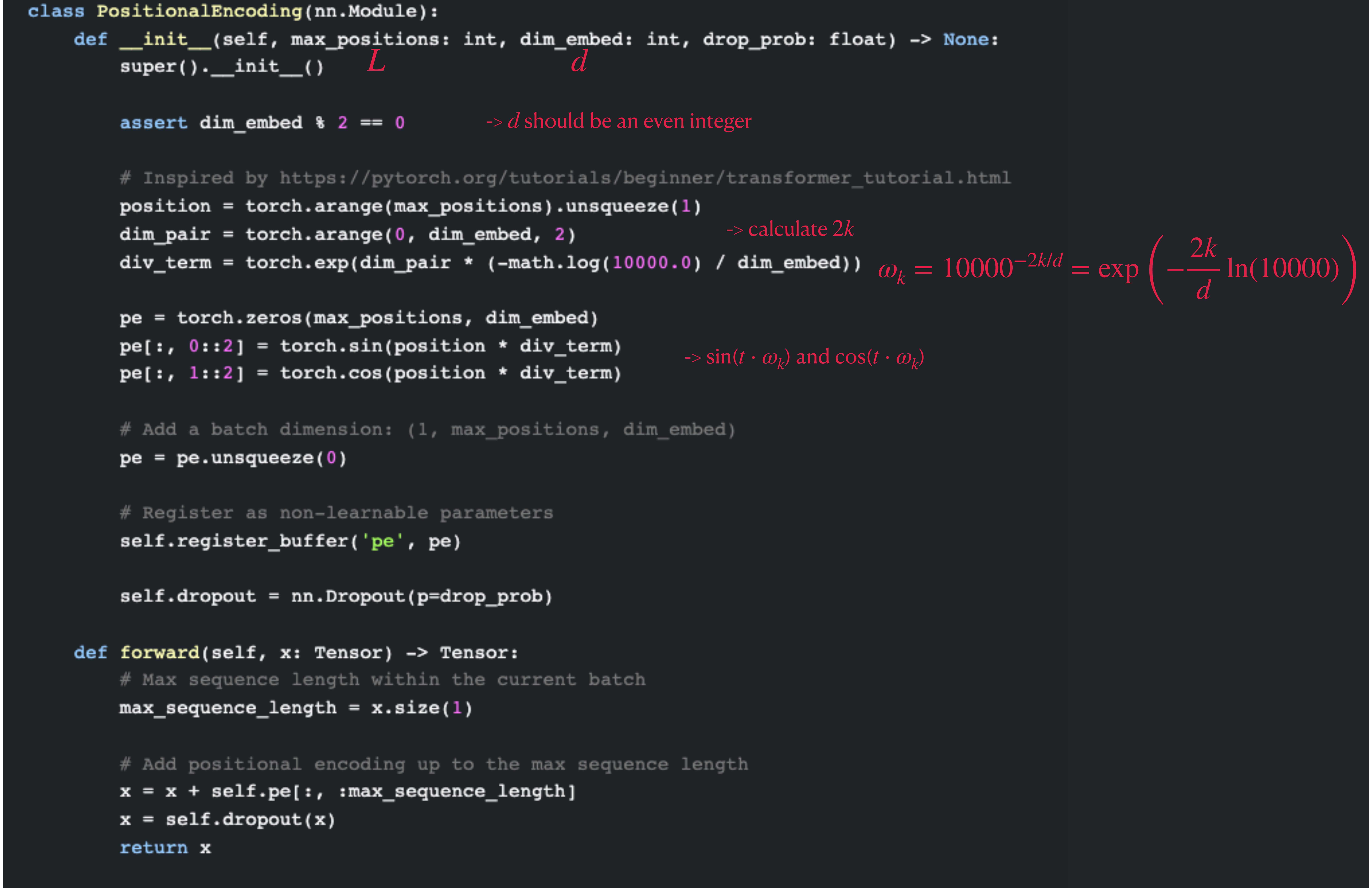
$$\begin{aligned}&\sin \omega_k t \cdot \sin \omega_k (t-\phi) + \cos \omega_k t \cdot \cos \omega_k (t-\phi) \\ &\quad = \sin^2 \omega_k t \cos \omega_k \phi + \cos^2 \omega_k t \sin \omega_k \phi\end{aligned}$$

Positional Encoding

$$\mathbf{x} \in \mathbb{R}^{L}, \mathbf{X}, \mathbf{P} \in \mathbb{R}^{L\times d}, \mathbf{W} \in \mathbb{R}^{2d\times d}$$

$$\mathbf{X}' = \mathbf{X} + \mathbf{P} = \underset{\mathbb{R}^{L\times d}}{\text{one-hot}(\mathbf{x})} \cdot \mathbf{E} + \underset{\textbf{Fixed}}{\mathbf{P}}$$

$$\mathbf{X}' = \text{Concat}[\mathbf{X}, \mathbf{P}] \cdot \mathbf{W} = \mathbf{X} \cdot \mathbf{W}_1 + \mathbf{P} \cdot \mathbf{W}_2 = \text{one-hot}(\mathbf{x}) \cdot \mathbf{E} \cdot \mathbf{W}_1 + \mathbf{P} \cdot \underset{\textbf{Learnable}}{\mathbf{W}_2}$$



```python
class PositionalEncoding(nn.Module):
    def __init__(self, max_positions: int, dim_embed: int, drop_prob: float) -> None:
        super().__init__()

        assert dim_embed % 2 == 0

        # Inspired by https://pytorch.org/tutorials/beginner/transformer_tutorial.html
        position = torch.arange(max_positions).unsqueeze(1)
        dim_pair = torch.arange(0, dim_embed, 2)
        div_term = torch.exp(dim_pair * (-math.log(10000.0) / dim_embed))

        pe = torch.zeros(max_positions, dim_embed)
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        # Add a batch dimension: (1, max_positions, dim_embed)
        pe = pe.unsqueeze(0)

        # Register as non-learnable parameters
        self.register_buffer('pe', pe)

        self.dropout = nn.Dropout(p=drop_prob)

    def forward(self, x: Tensor) -> Tensor:
        # Max sequence length within the current batch
        max_sequence_length = x.size(1)

        # Add positional encoding up to the max sequence length
        x = x + self.pe[:, :max_sequence_length]
        x = self.dropout(x)
        return x
```

L d

-> d should be an even integer

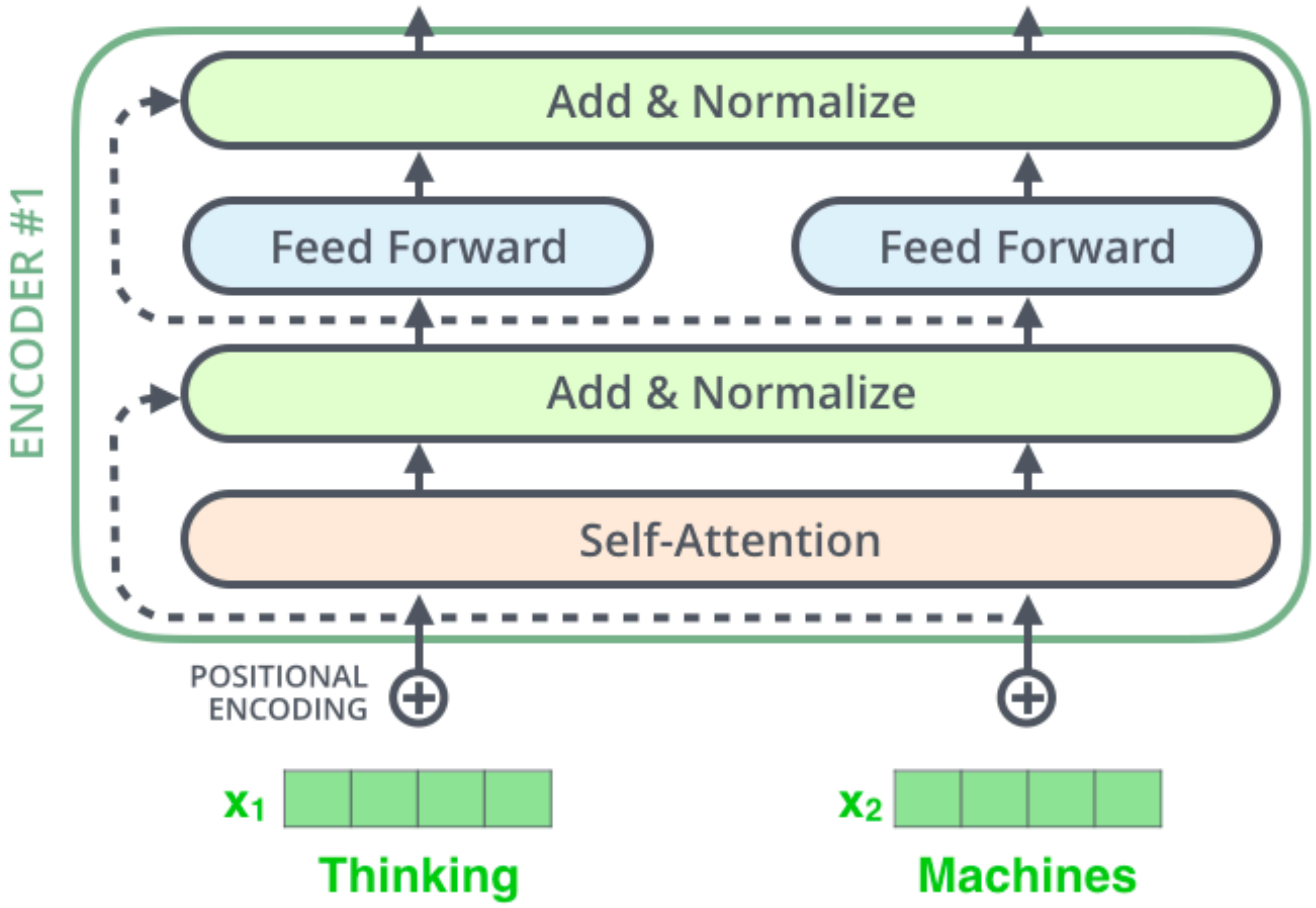
-> calculate $2k$

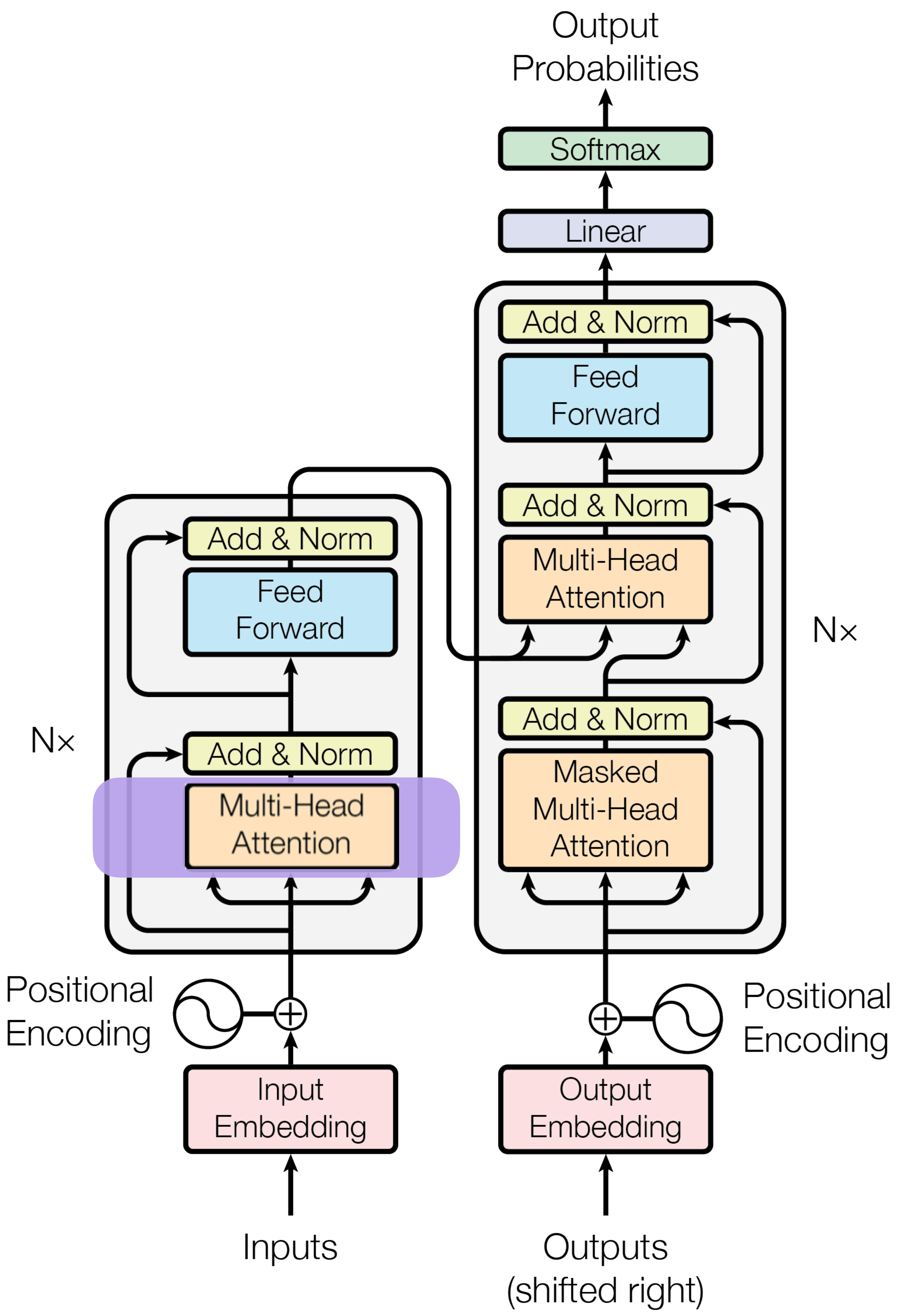
$$\omega_k = 10000^{-2k/d} = \exp\left(-\frac{2k}{d}\ln(10000)\right)$$

-> $\sin(t \cdot \omega_k)$ and $\cos(t \cdot \omega_k)$

Multi-Head Attention

Self-Attention

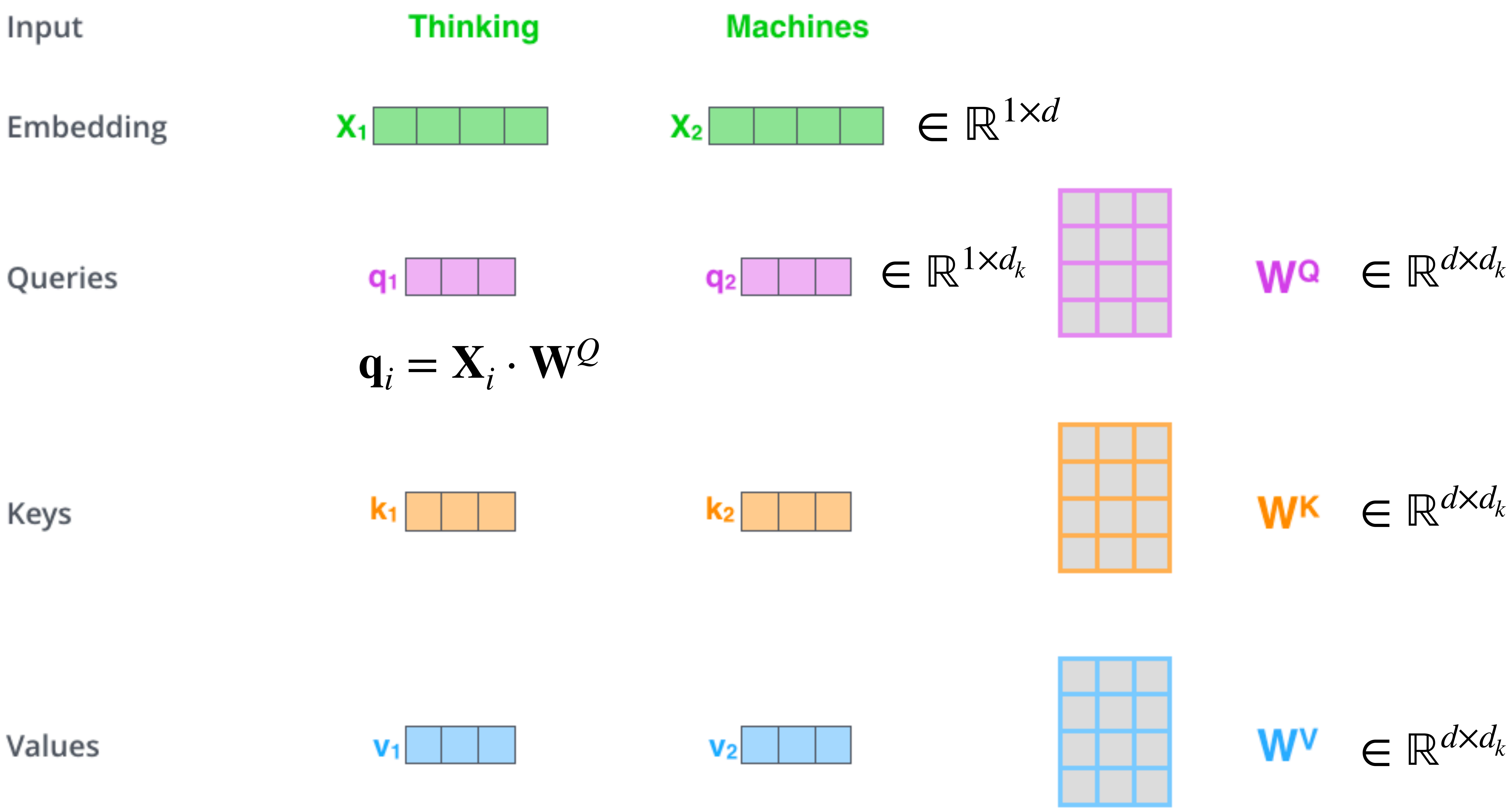




Reference : The Illustrated Transformer, https://jalammar.github.io/illustrated-transformer/

Multi-Head Attention

Self-Attention



Input — Thinking — Machines

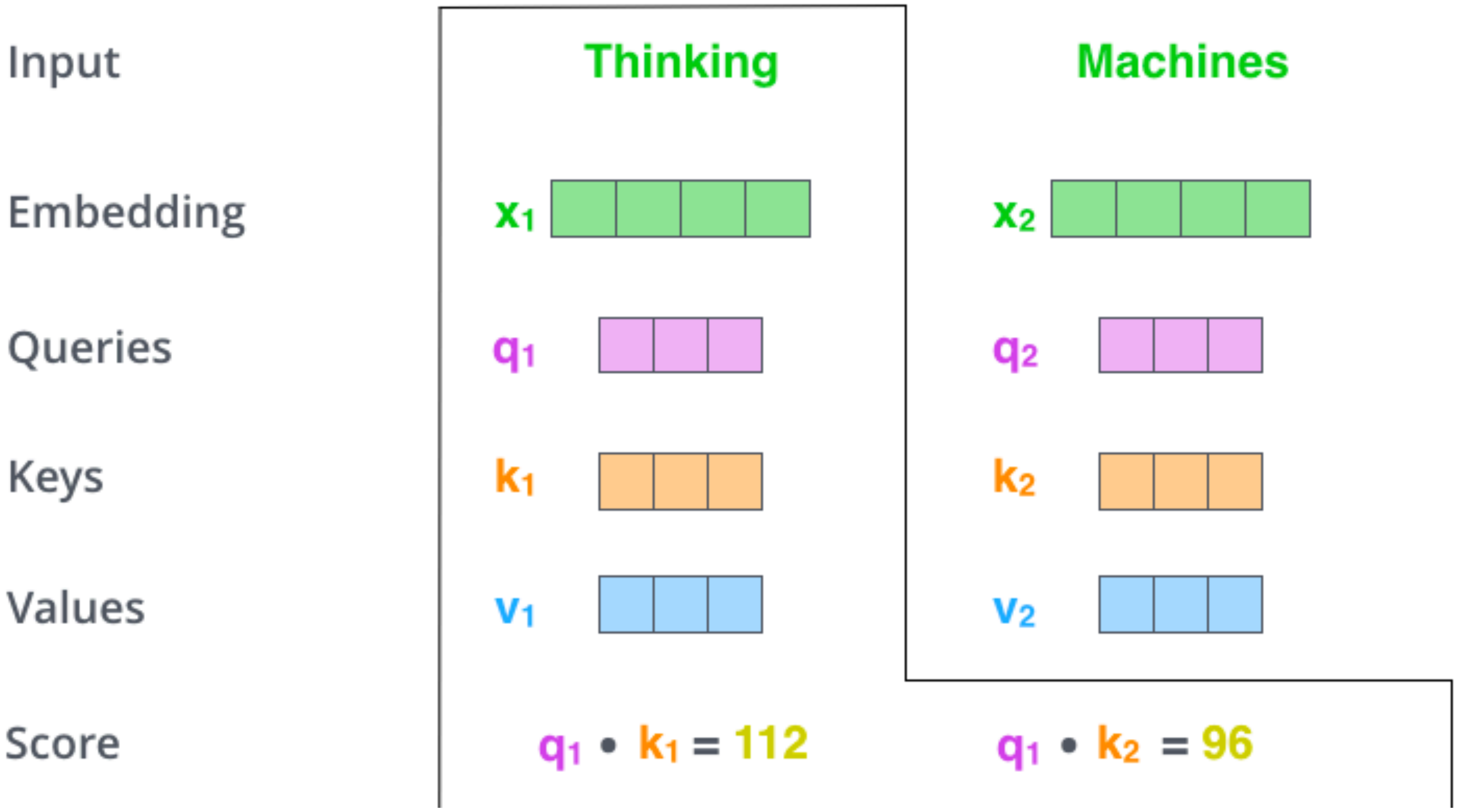
Embedding — X_1 — X_2 — $\in \mathbb{R}^{1\times d}$

Queries — q_1 — q_2 — $\in \mathbb{R}^{1\times d_k}$ — W^Q $\in \mathbb{R}^{d\times d_k}$

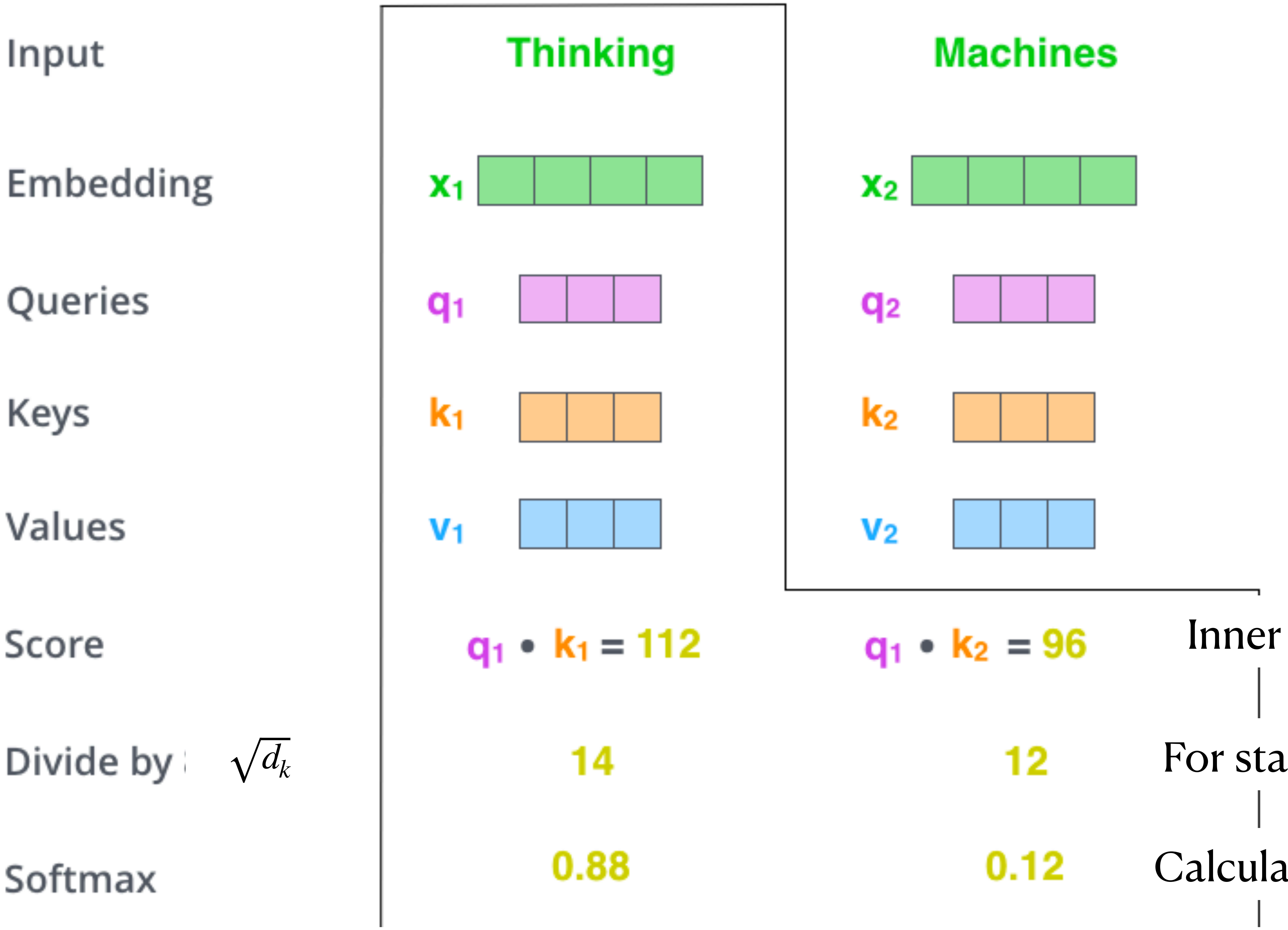
$$\mathbf{q}_i = \mathbf{X}_i \cdot \mathbf{W}^Q$$

Keys — k_1 — k_2 — W^K $\in \mathbb{R}^{d\times d_k}$

Values — v_1 — v_2 — W^V $\in \mathbb{R}^{d\times d_k}$



Input	Thinking	Machines
Embedding	x_1	x_2
Queries	q_1	q_2
Keys	k_1	k_2
Values	v_1	v_2
Score	$q_1 \bullet k_1 = 112$	$q_1 \bullet k_2 = 96$

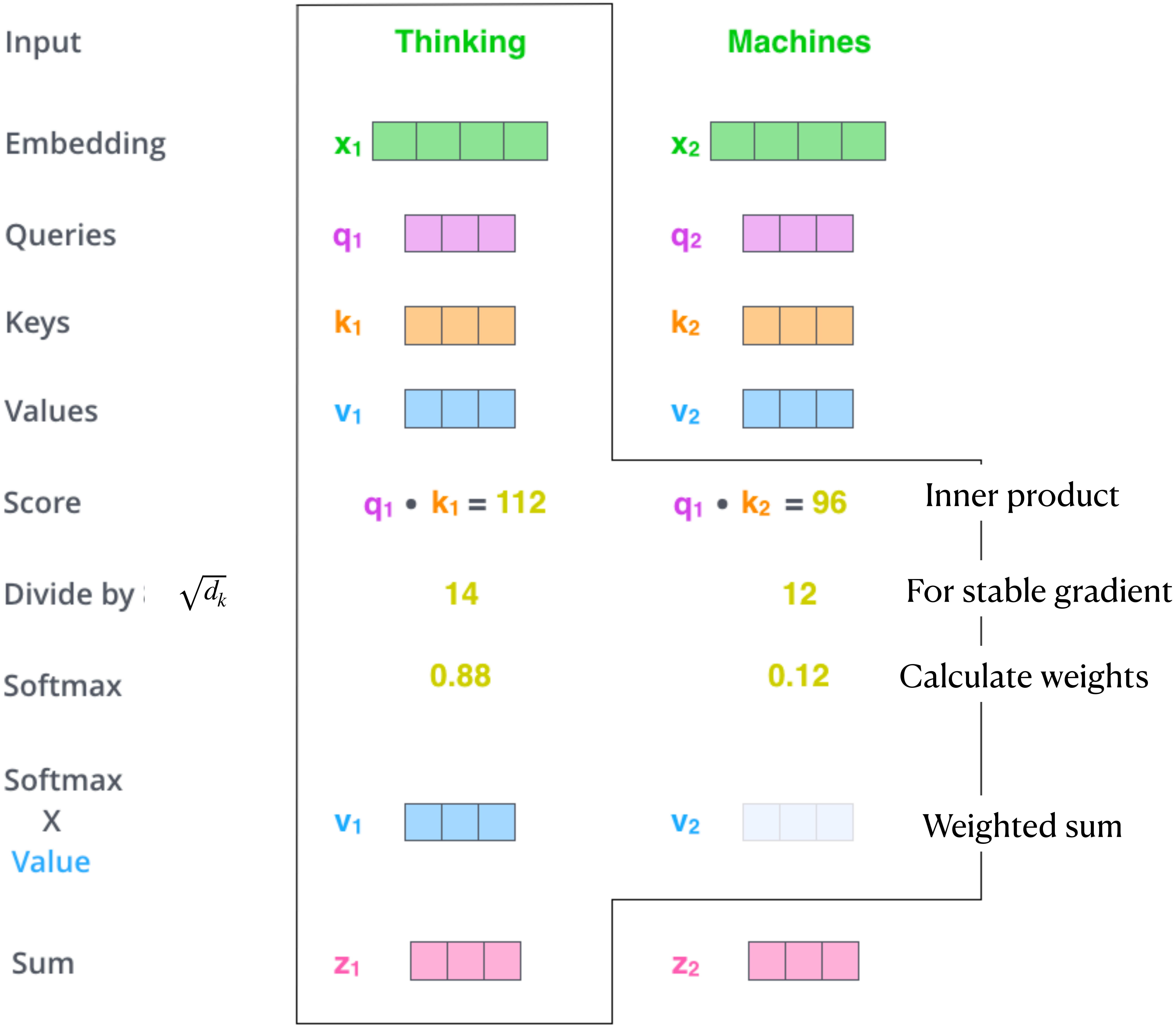


Inner product

For stable gradient

Calculate weights

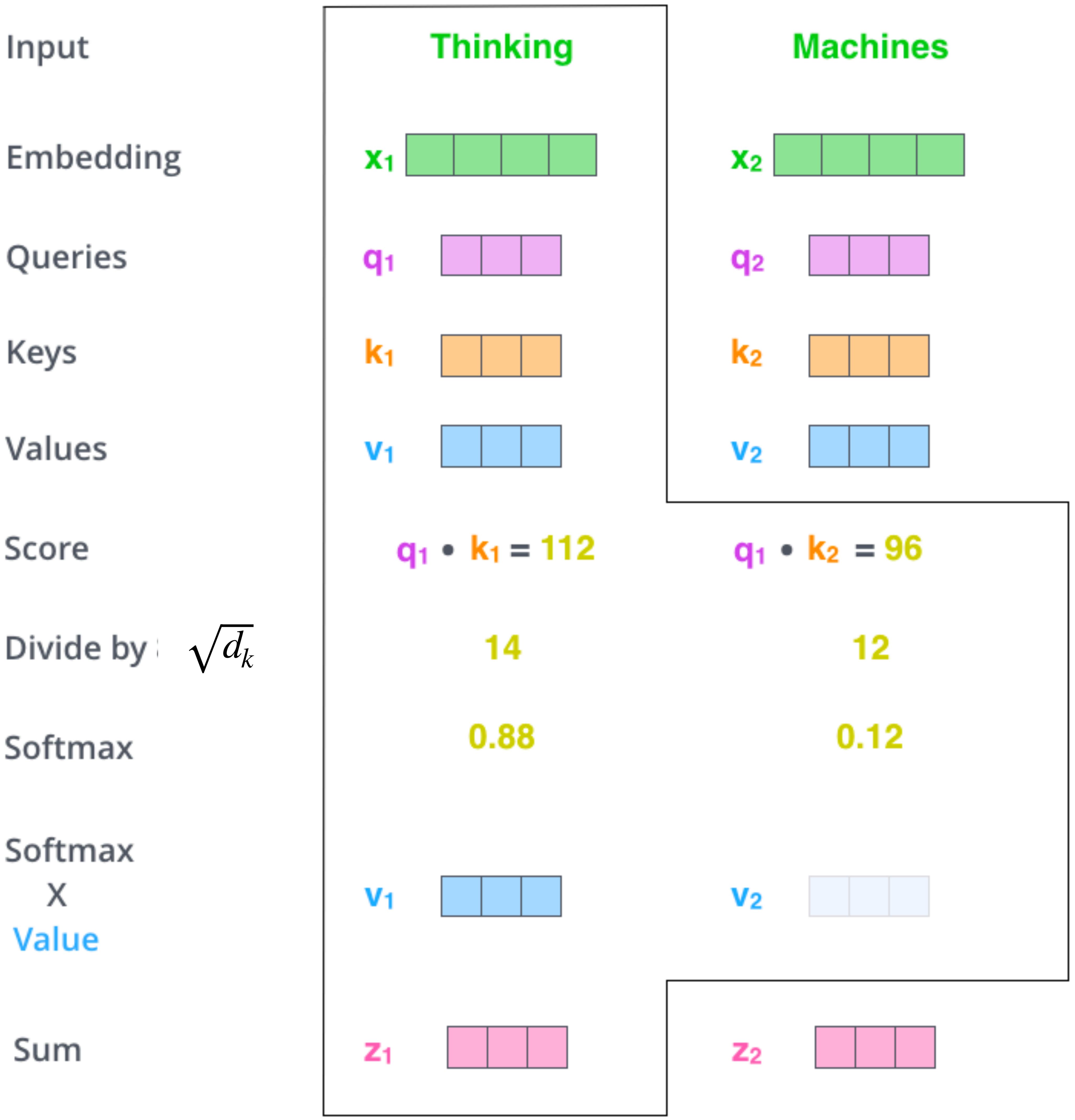
"We suspect that **for large values of d_k** , the **dot products grow large in magnitude**, pushing the softmax function into regions where it has **extremely small gradients**. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$. " ——Sec 3.2.1

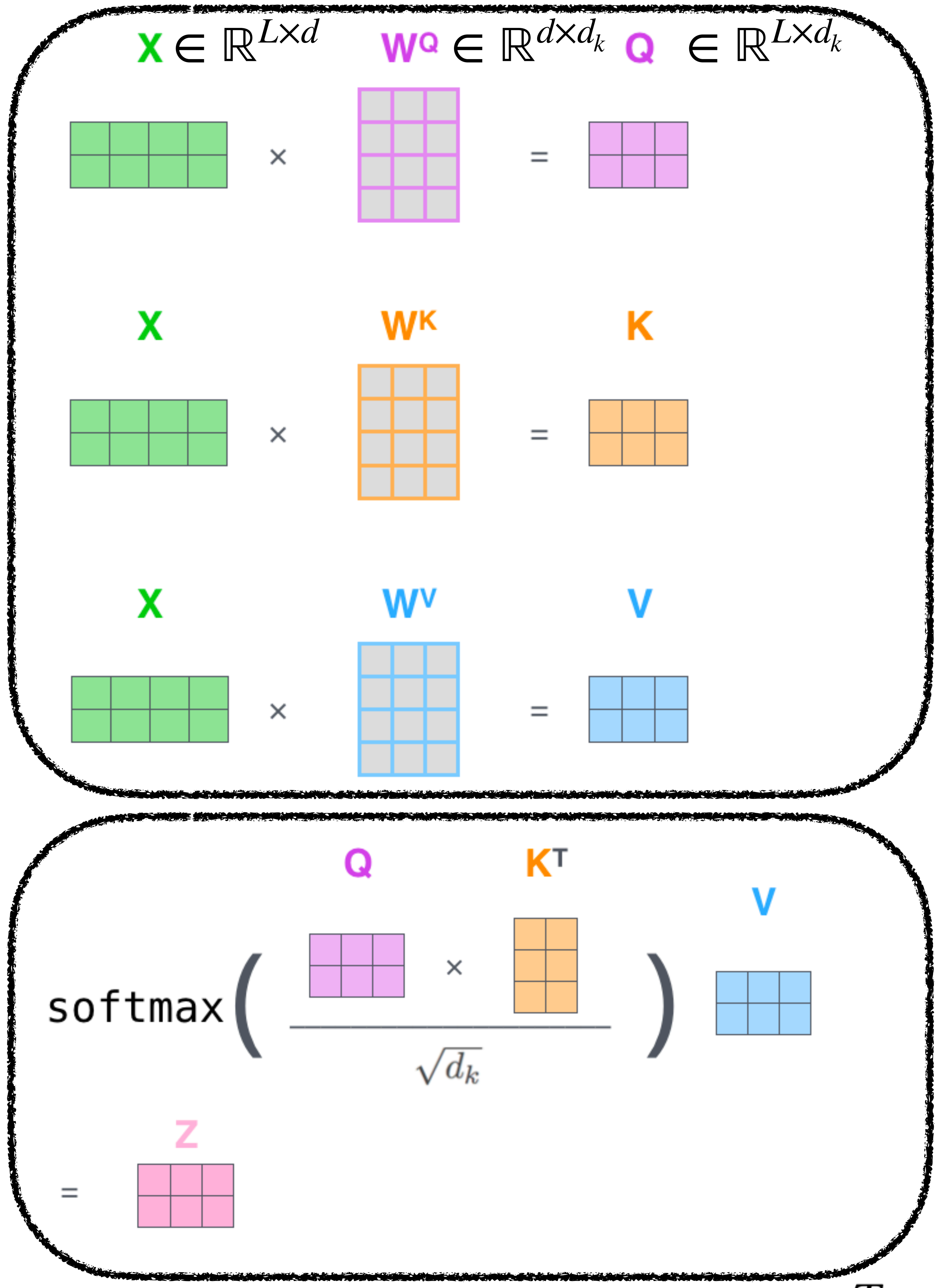


Input	Thinking	Machines	
Embedding	x_1	x_2	
Queries	q_1	q_2	
Keys	k_1	k_2	
Values	v_1	v_2	
Score	$q_1 \bullet k_1 = 112$	$q_1 \bullet k_2 = 96$	Inner product
Divide by $\sqrt{d_k}$	14	12	For stable gradient
Softmax	0.88	0.12	Calculate weights
Softmax X Value	v_1	v_2	Weighted sum
Sum	z_1	z_2	

Multi-Head Attention

Matrix Calculation of Self-Attention

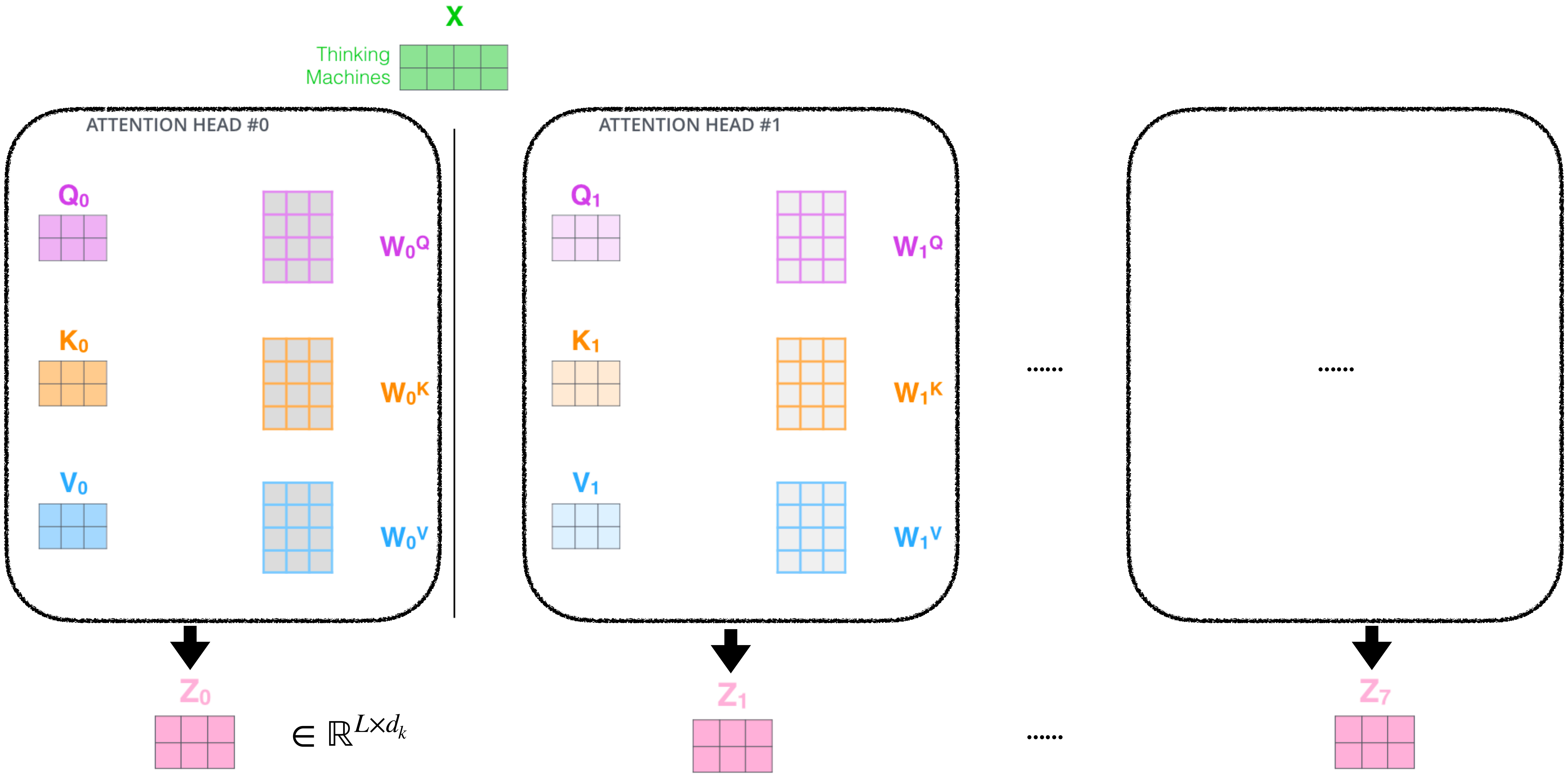



$X \in \mathbb{R}^{L \times d}$
$W^Q \in \mathbb{R}^{d \times d_k}$
$Q \in \mathbb{R}^{L \times d_k}$
×
=
X
W^K
K
×
=
X
W^V
V
×
=
softmax
Q
K^T
×
$\sqrt{d_k}$
V
=
Z

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

Multi-Head Attention

Multi-Head



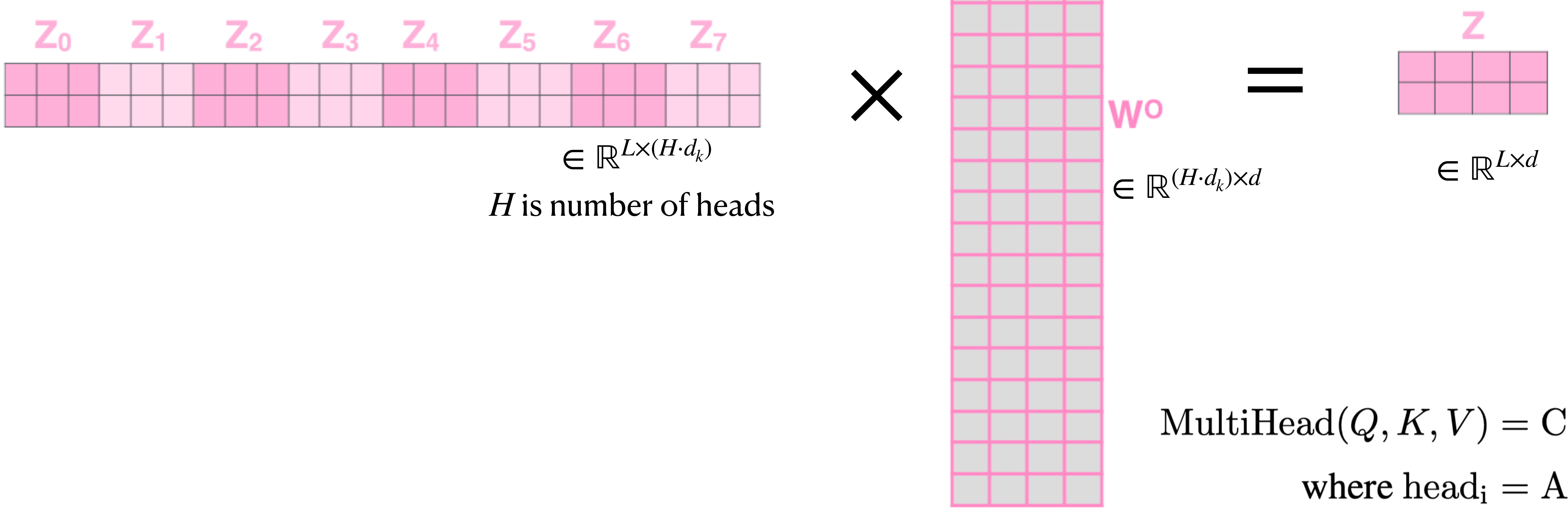
Multi-Head Attention

Multi-Head

1) Concatenate all the attention heads

2) Multiply with a weight matrix W^O that was trained jointly with the model

3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN



$\in \mathbb{R}^{L\times(H\cdot d_k)}$

H is number of heads

$\in \mathbb{R}^{(H\cdot d_k)\times d}$

$\in \mathbb{R}^{L\times d}$

$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1,\dots,\text{head}_H)W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Multi-Head Attention

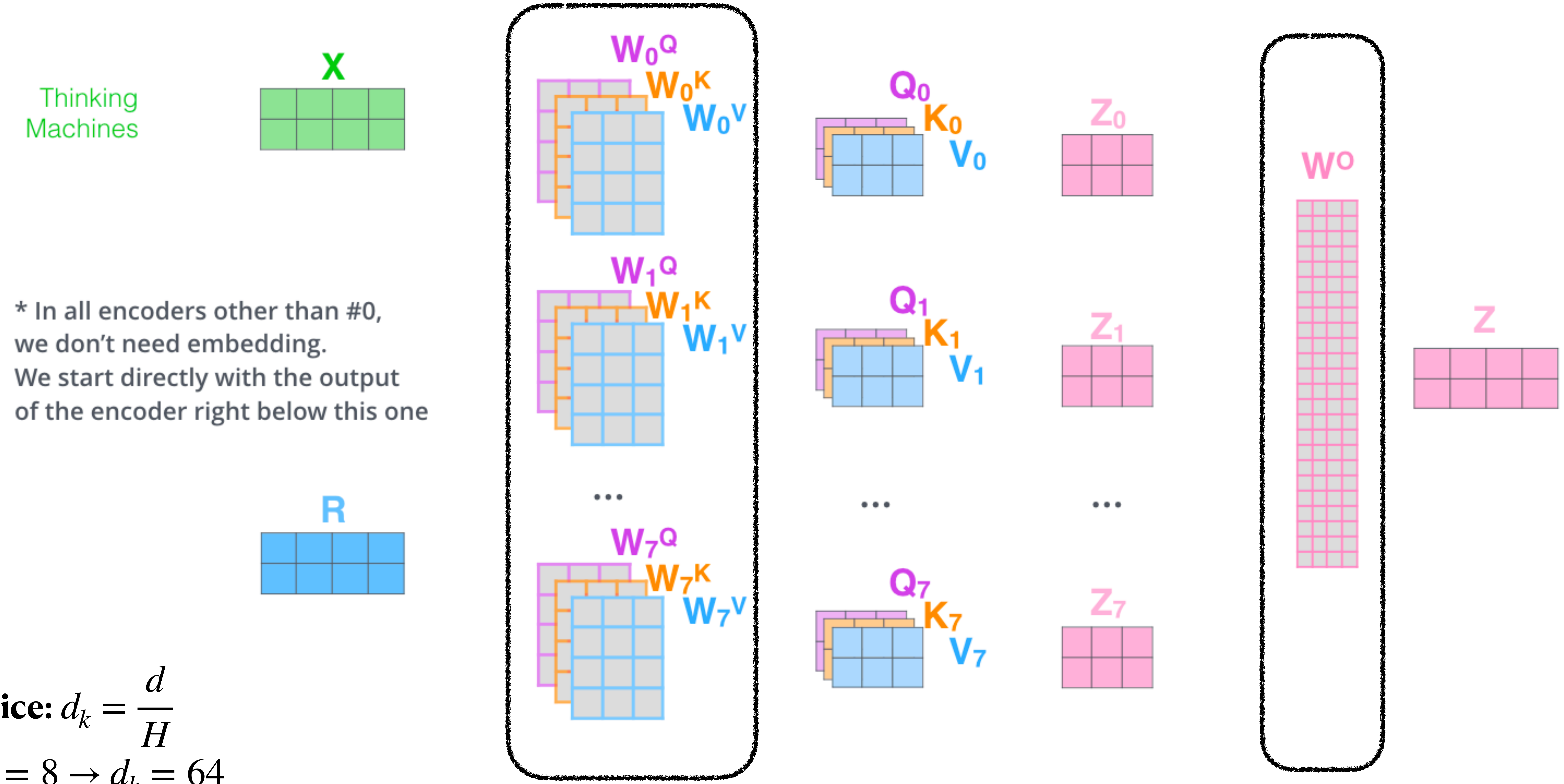
1) This is our input sentence*

2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

4) Calculate attention using the resulting Q/K/V matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer



* In all encoders other than #0, we don’t need embedding. We start directly with the output of the encoder right below this one

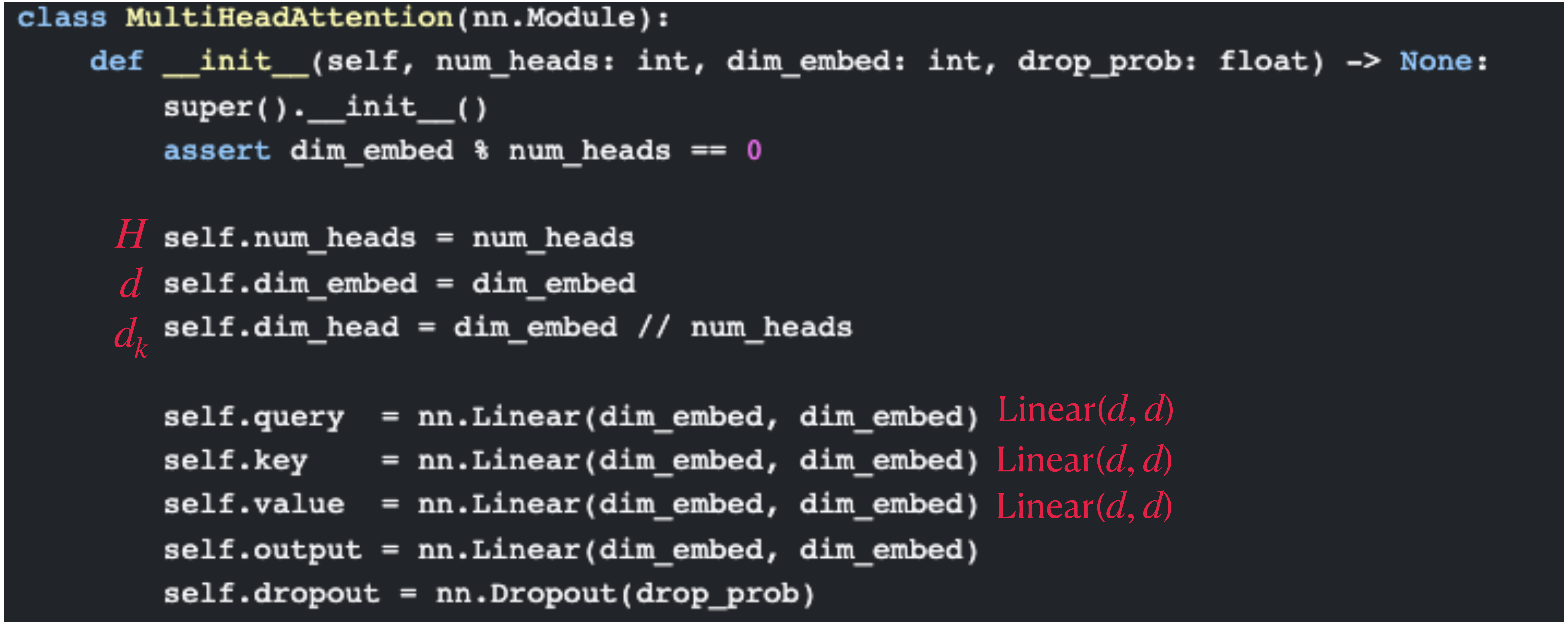
Default choice: $d_k = \dfrac{d}{H}$

e.g., $d = 512, H = 8 \rightarrow d_k = 64$

$H \times \text{Linear}(d, d_k)$
-> $\text{Linear}(d, Hd_k)$
-> $\text{Linear}(d, d)$

$\text{Linear}(Hd_k, d)$
->$\text{Linear}(d, d)$

Multi-Head Attention



```
class MultiHeadAttention(nn.Module):
    def __init__(self, num_heads: int, dim_embed: int, drop_prob: float) -> None:
        super().__init__()
        assert dim_embed % num_heads == 0

        self.num_heads = num_heads
        self.dim_embed = dim_embed
        self.dim_head = dim_embed // num_heads

        self.query  = nn.Linear(dim_embed, dim_embed)
        self.key    = nn.Linear(dim_embed, dim_embed)
        self.value  = nn.Linear(dim_embed, dim_embed)
        self.output = nn.Linear(dim_embed, dim_embed)
        self.dropout = nn.Dropout(drop_prob)
```

H

d

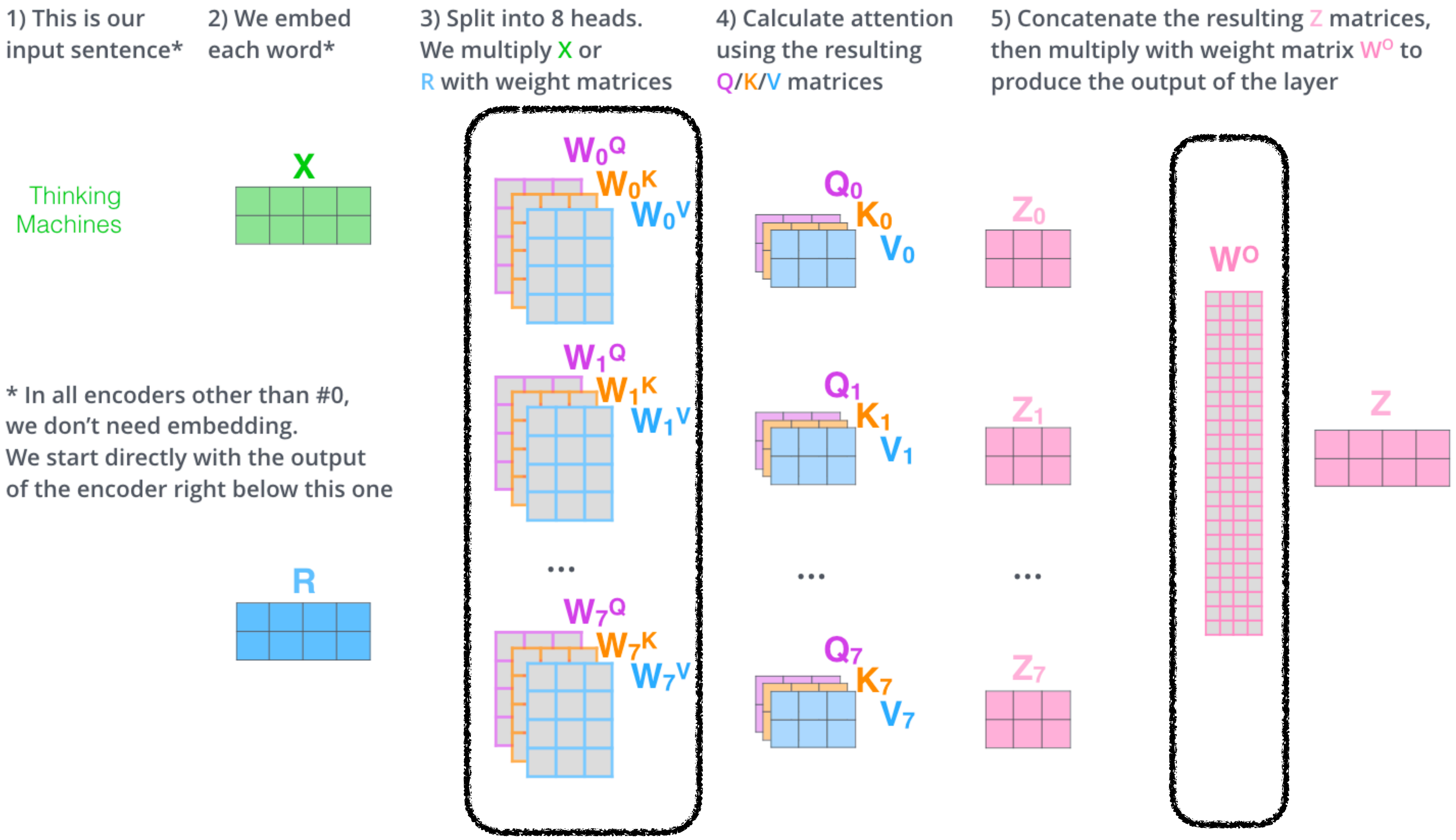
d_k

Linear(d, d)

Linear(d, d)

Linear(d, d)

Multi-Head Attention



1) This is our input sentence*

2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

4) Calculate attention using the resulting Q/K/V matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

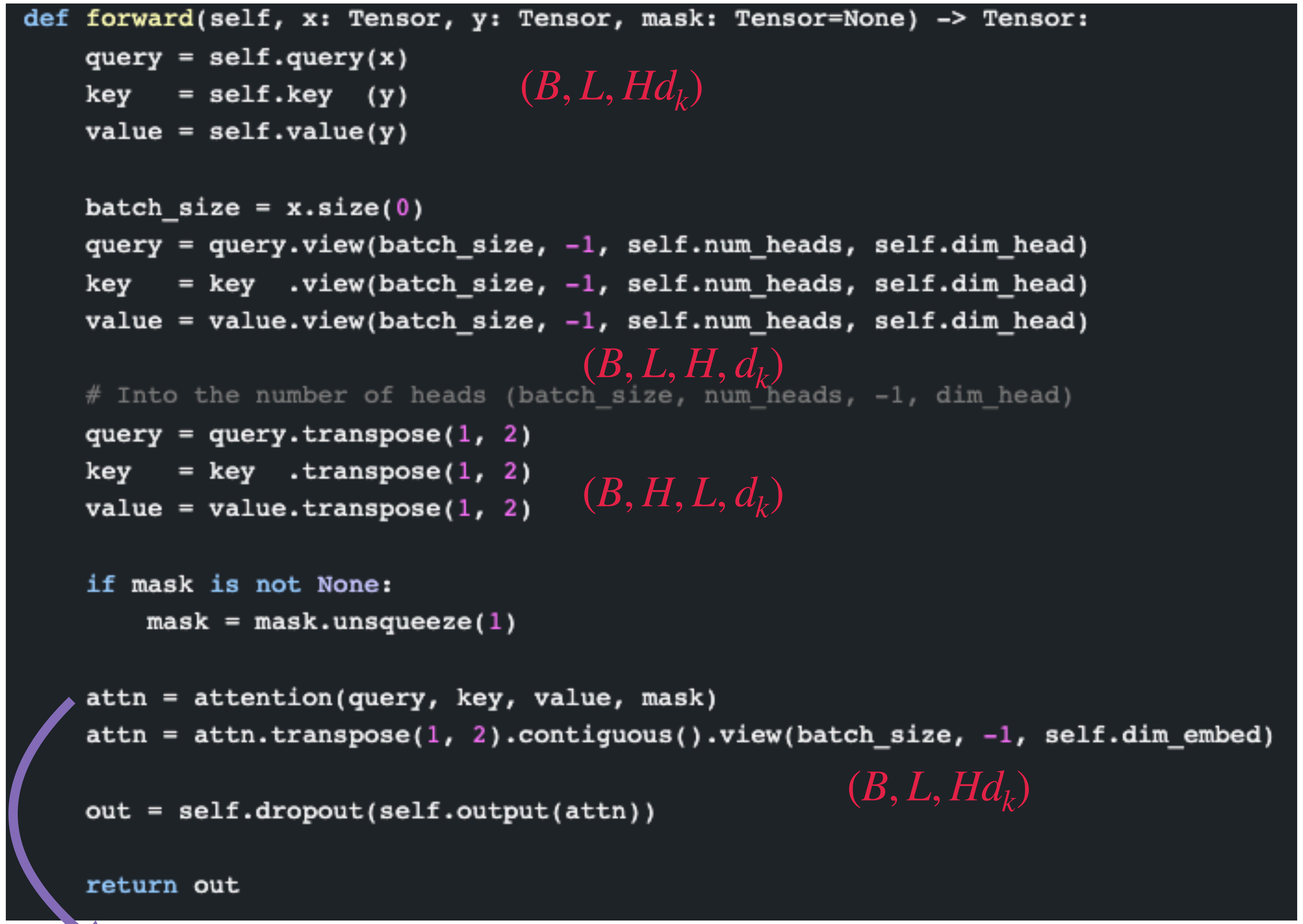
$H \times \text{Linear}(d, d_k)$

-> $\text{Linear}(d, Hd_k)$

-> $\text{Linear}(d, d)$

$\text{Linear}(Hd_k, d)$

->$\text{Linear}(d, d)$



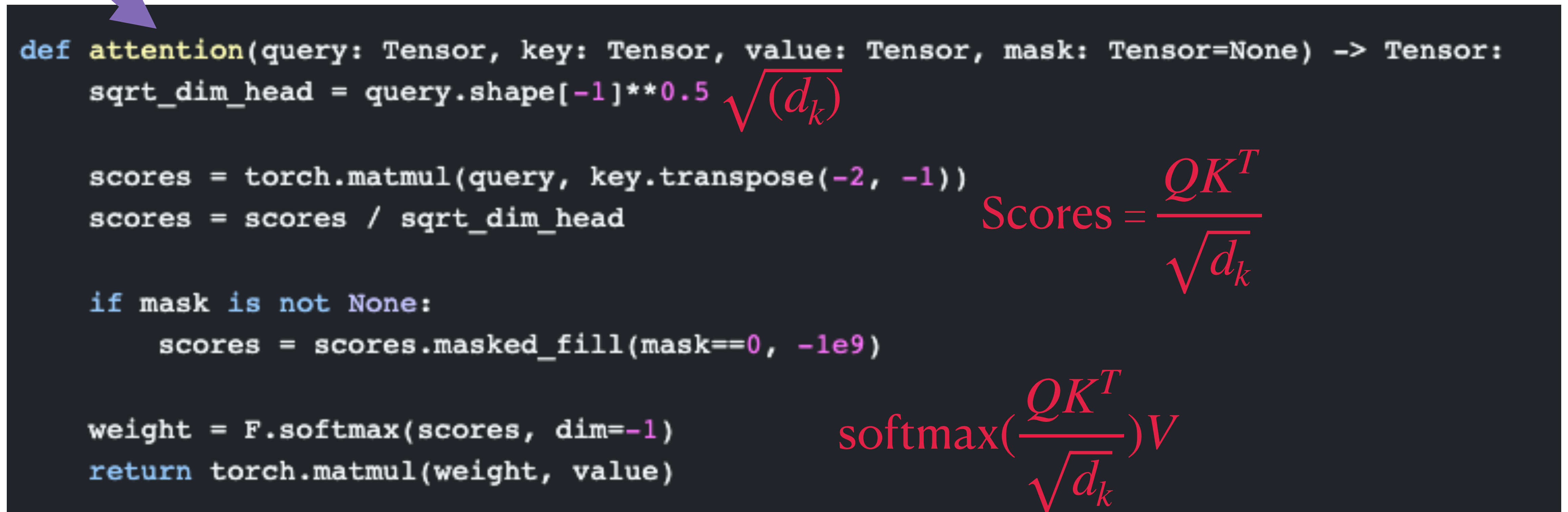
```
def forward(self, x: Tensor, y: Tensor, mask: Tensor=None) -> Tensor:
    query = self.query(x)
    key   = self.key  (y)
    value = self.value(y)

    batch_size = x.size(0)
    query = query.view(batch_size, -1, self.num_heads, self.dim_head)
    key   = key  .view(batch_size, -1, self.num_heads, self.dim_head)
    value = value.view(batch_size, -1, self.num_heads, self.dim_head)

    # Into the number of heads (batch_size, num_heads, -1, dim_head)
    query = query.transpose(1, 2)
    key   = key  .transpose(1, 2)
    value = value.transpose(1, 2)

    if mask is not None:
        mask = mask.unsqueeze(1)

    attn = attention(query, key, value, mask)
    attn = attn.transpose(1, 2).contiguous().view(batch_size, -1, self.dim_embed)

    out = self.dropout(self.output(attn))

    return out
```

(B, L, Hd_k)

(B, L, H, d_k)

(B, H, L, d_k)

(B, L, Hd_k)



```
def attention(query: Tensor, key: Tensor, value: Tensor, mask: Tensor=None) -> Tensor:
    sqrt_dim_head = query.shape[-1]**0.5

    scores = torch.matmul(query, key.transpose(-2, -1))
    scores = scores / sqrt_dim_head

    if mask is not None:
        scores = scores.masked_fill(mask==0, -1e9)

    weight = F.softmax(scores, dim=-1)
    return torch.matmul(weight, value)
```

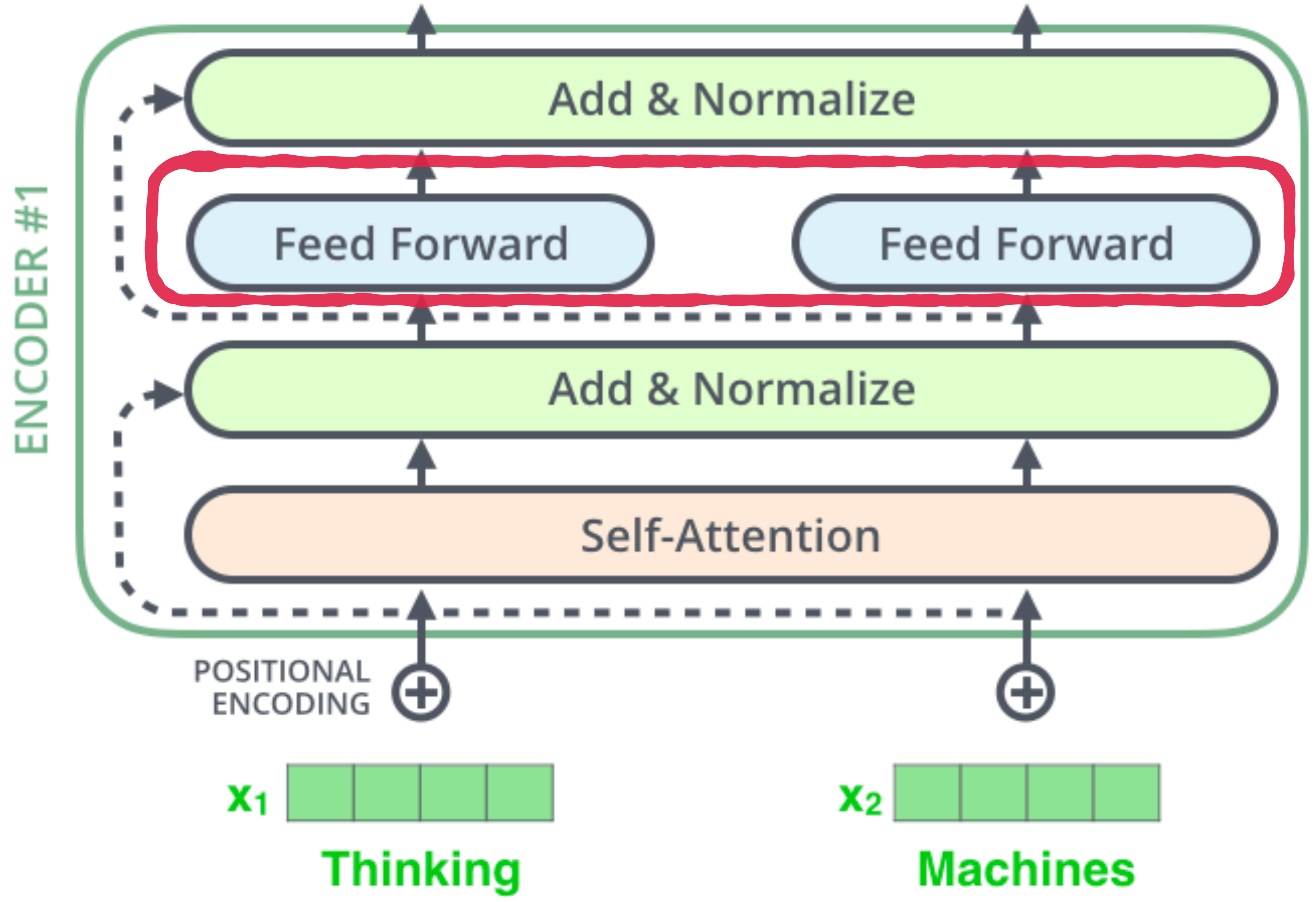
$\sqrt{(d_k)}$

$\text{Scores} = \frac{QK^T}{\sqrt{d_k}}$

$\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$

Position-Wise Feed-Forward

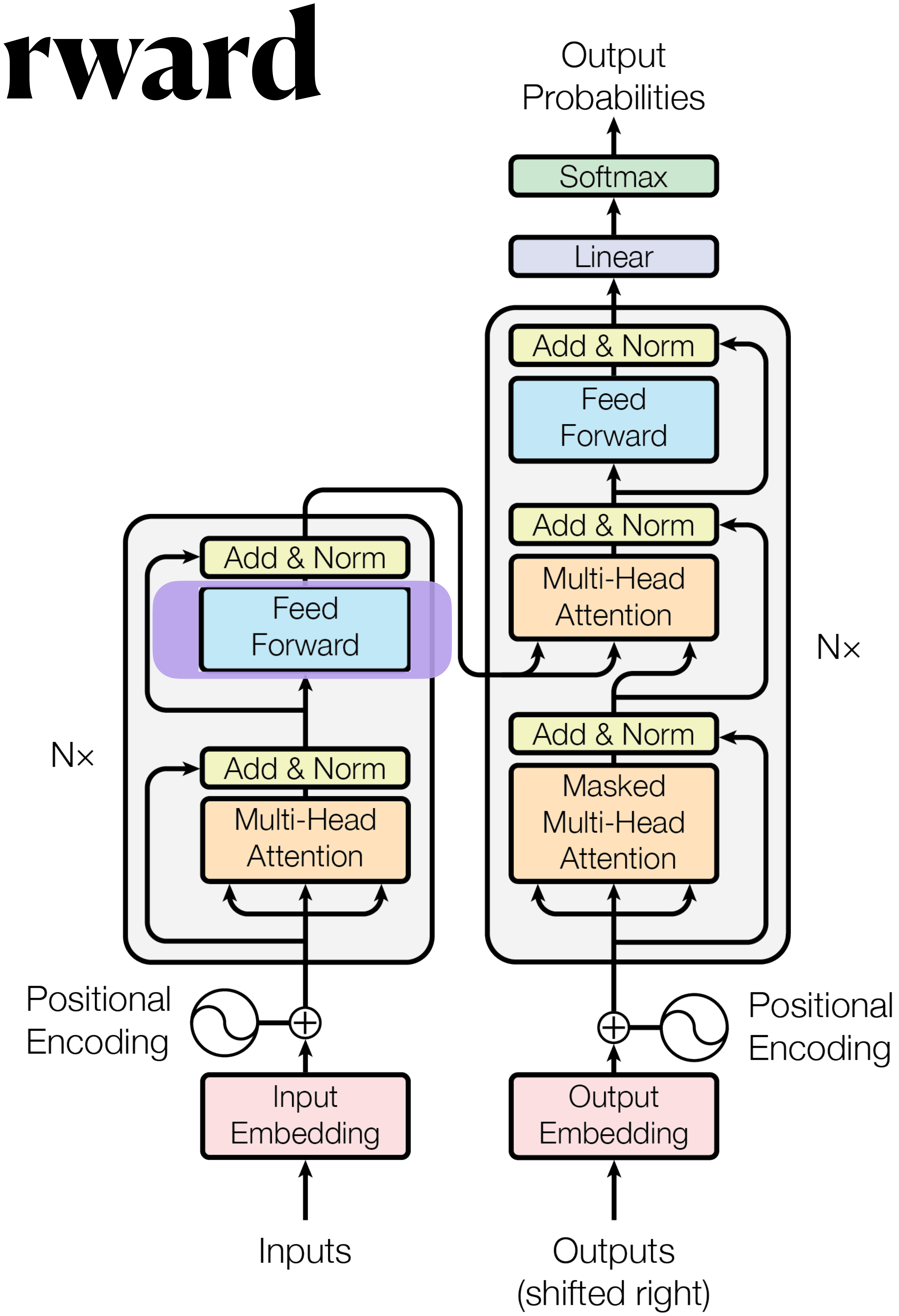
Goal: Inject non-linearity into embedding vectors.
Linear operations are applied to each position independently and identically.



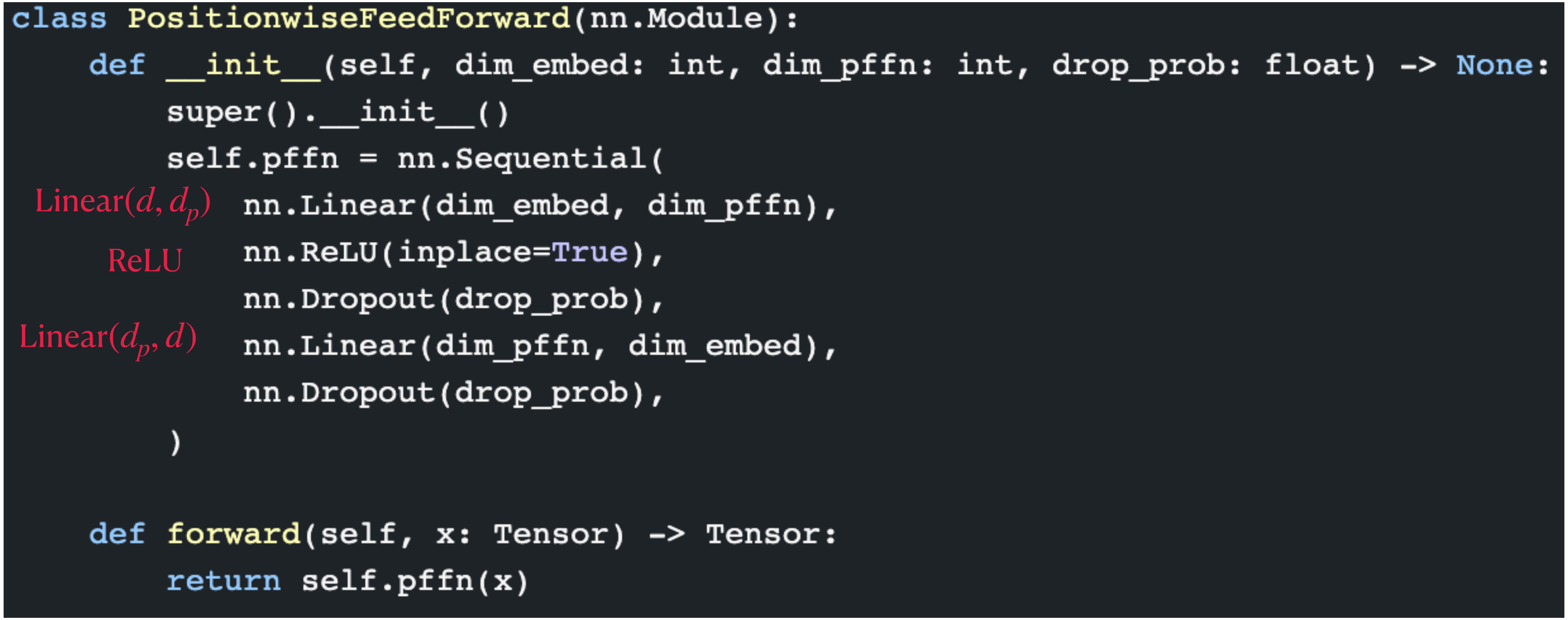
$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

$$W_1 \in \mathbb{R}^{d \times d_p}, W_2 \in \mathbb{R}^{d_p \times d}$$

Two Linear layers with ReLU activation



Position-Wise Feed-Forward



```
class PositionwiseFeedForward(nn.Module):
    def __init__(self, dim_embed: int, dim_pffn: int, drop_prob: float) -> None:
        super().__init__()
        self.pffn = nn.Sequential(
            nn.Linear(dim_embed, dim_pffn),
            nn.ReLU(inplace=True),
            nn.Dropout(drop_prob),
            nn.Linear(dim_pffn, dim_embed),
            nn.Dropout(drop_prob),
        )

    def forward(self, x: Tensor) -> Tensor:
        return self.pffn(x)
```

Linear(d, d_p)

ReLU

Linear(d_p, d)

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

$$W_1 \in \mathbb{R}^{d \times d_p}, W_2 \in \mathbb{R}^{d_p \times d}$$

Two Linear layers with ReLU activation

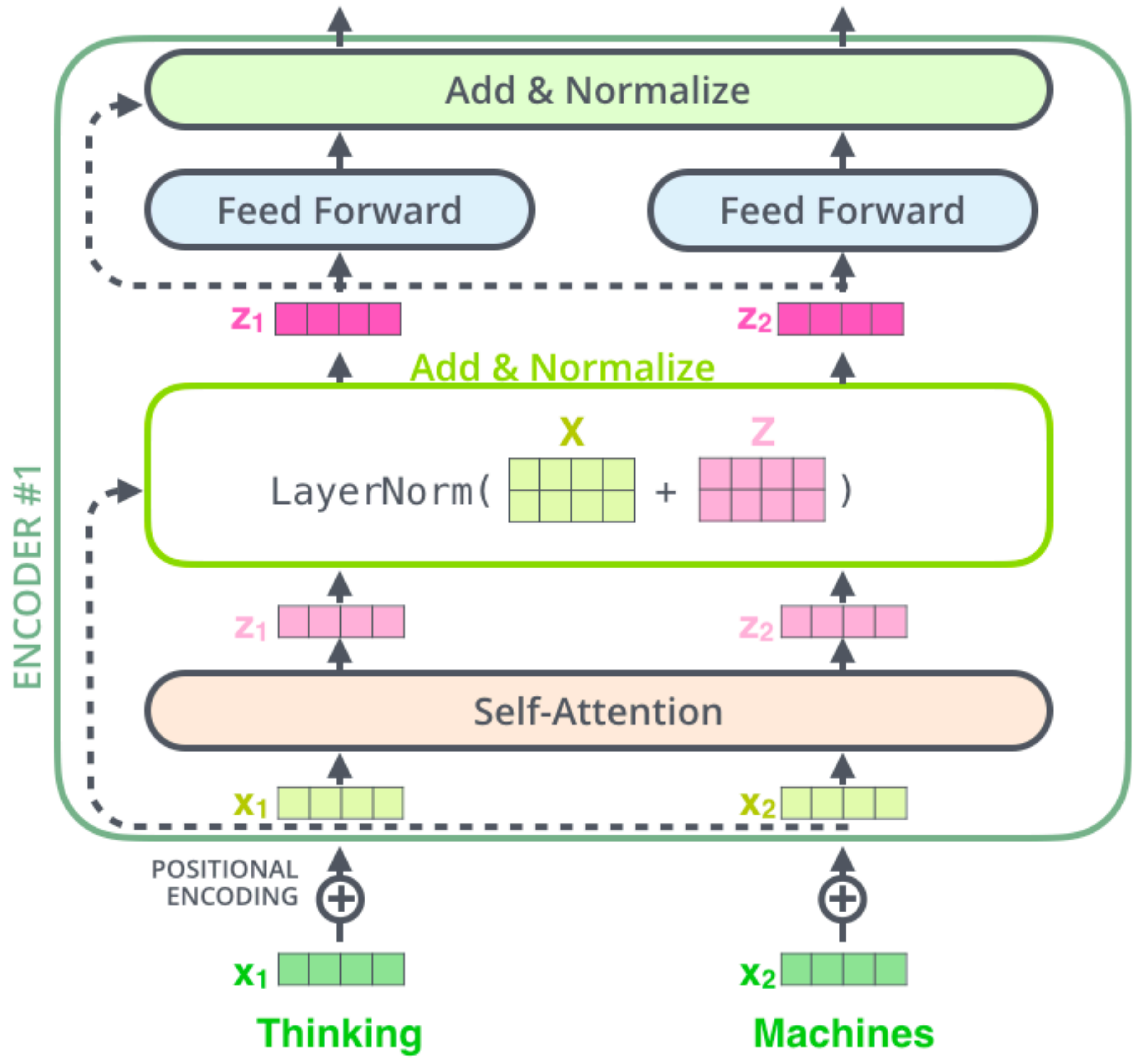
Residual Connection

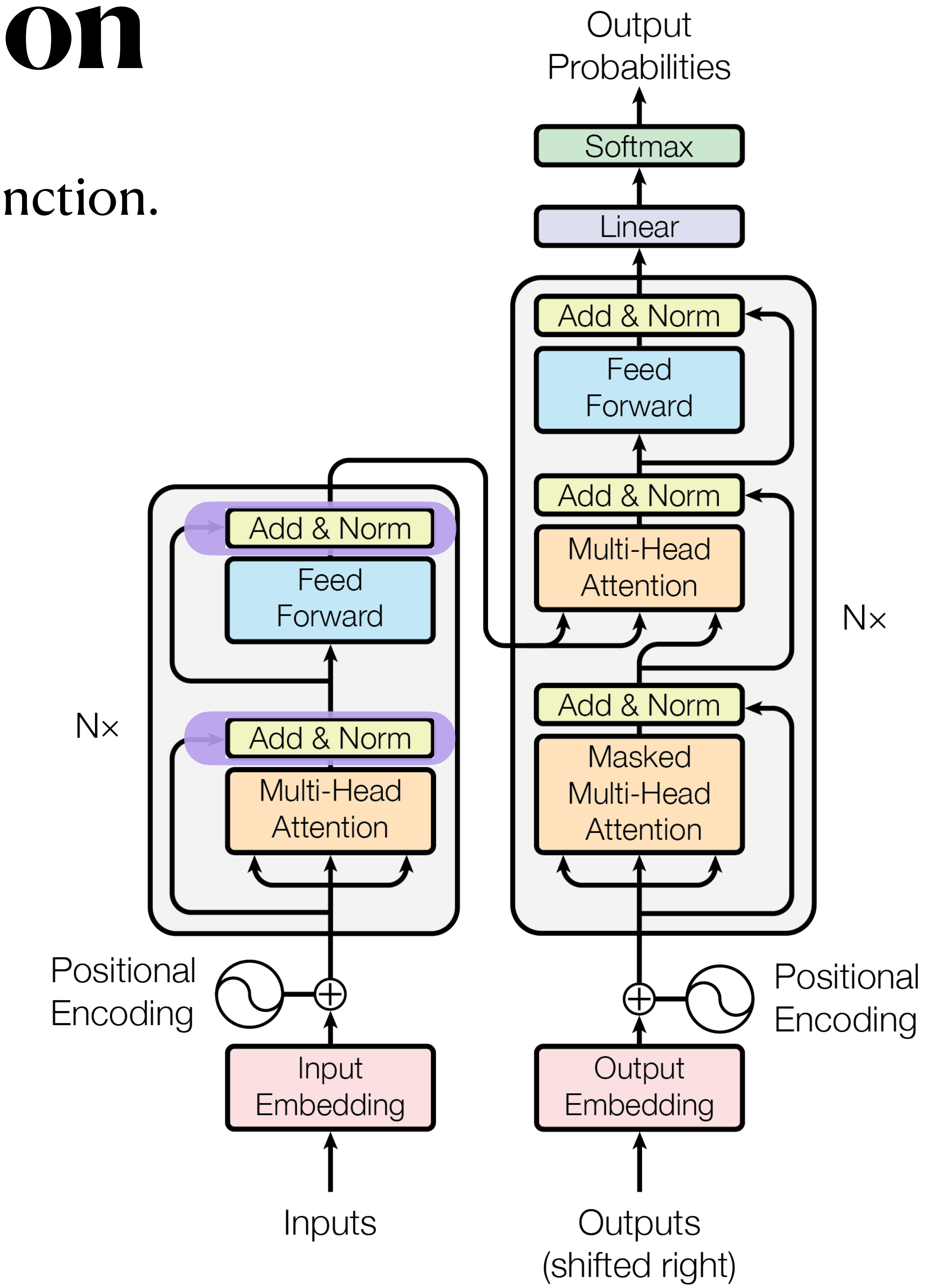
Goal: Mitigate the vanishing gradient problem.

During BP, the signal gets multiplied by the derivative of the activation function.

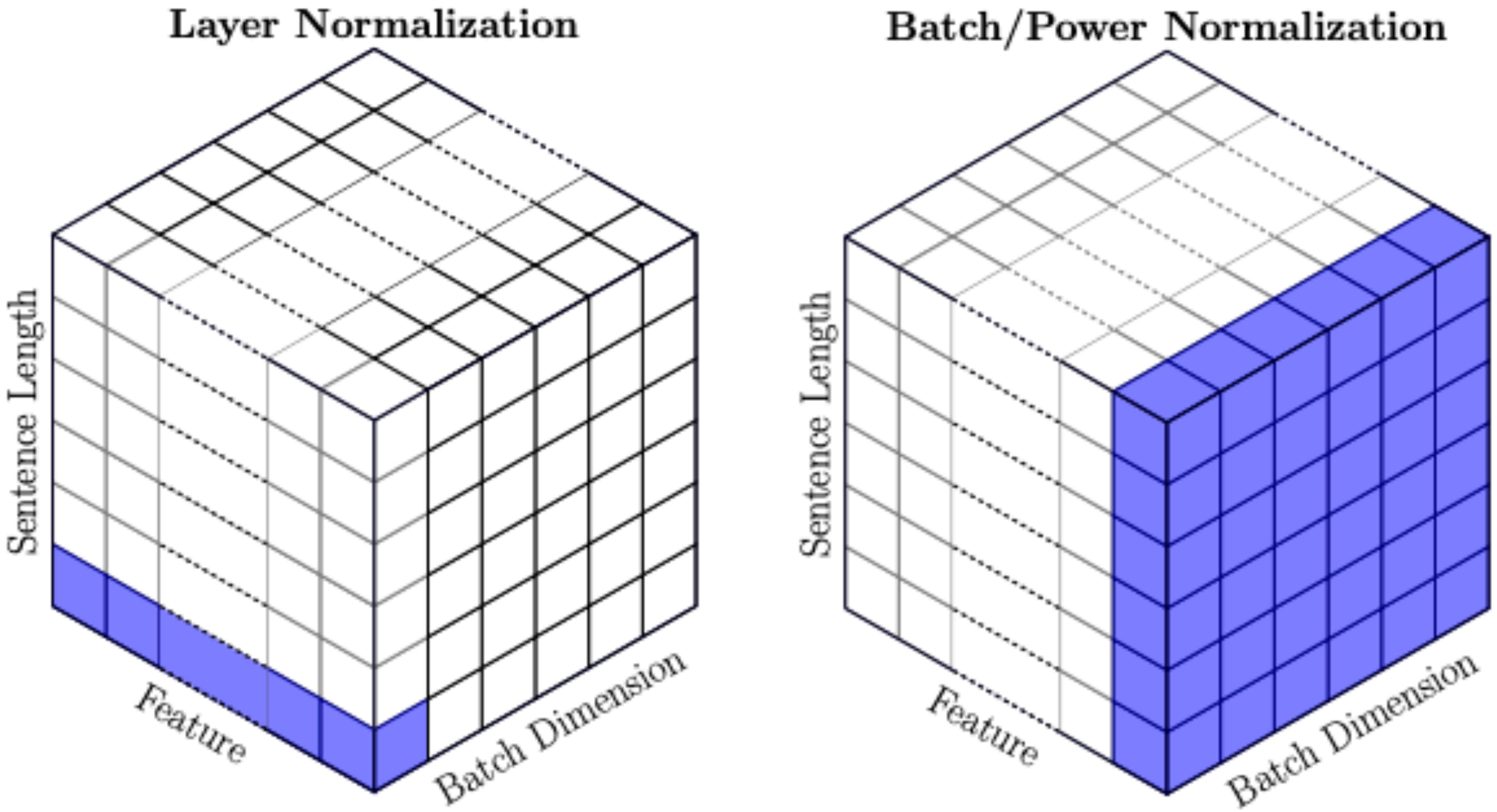
E.g., ReLU, about the half of the cases, the gradient is zero.

Without residual-> large part of training gets lost.





LN and BN



Normalization helps to stabilize the gradient in BP.
BN doesn’t perform good on RNN
Then people proposed LN-> default choice for NLP

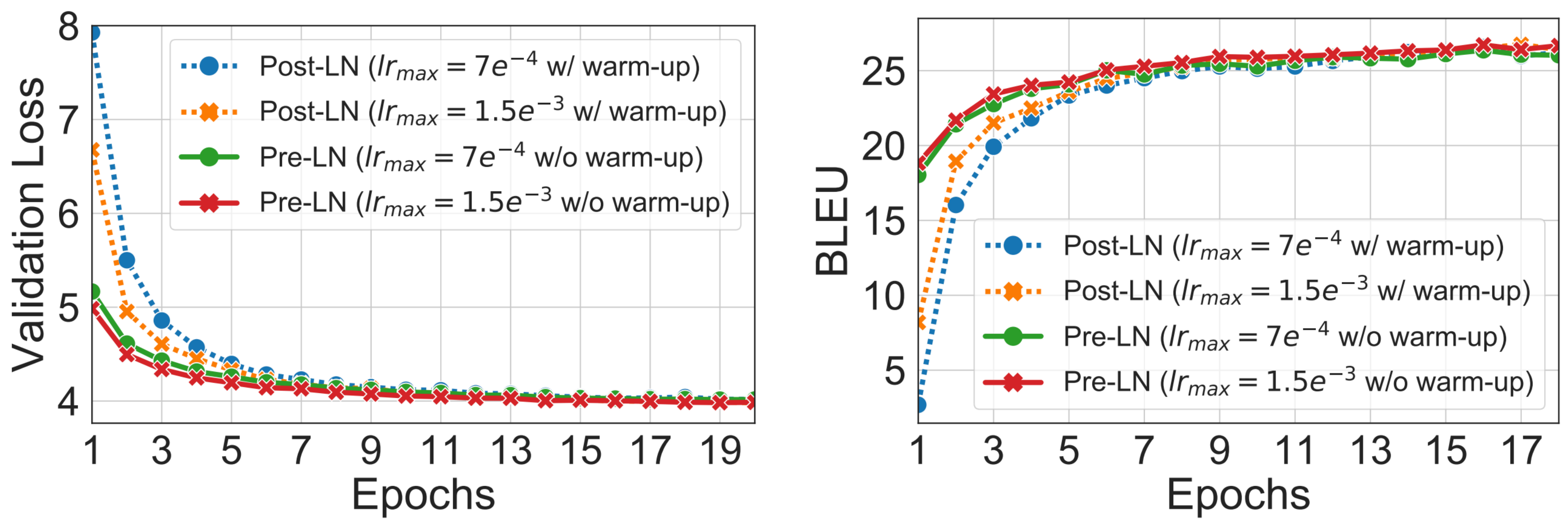
LN fits sequence data with unfixed length (operating on feature)

Related Papers:

- Leveraging Batch Normalization for Vision Transformers
- PowerNorm: Rethinking Batch Normalization in Transformers
- Understanding and Improving Layer Normalization

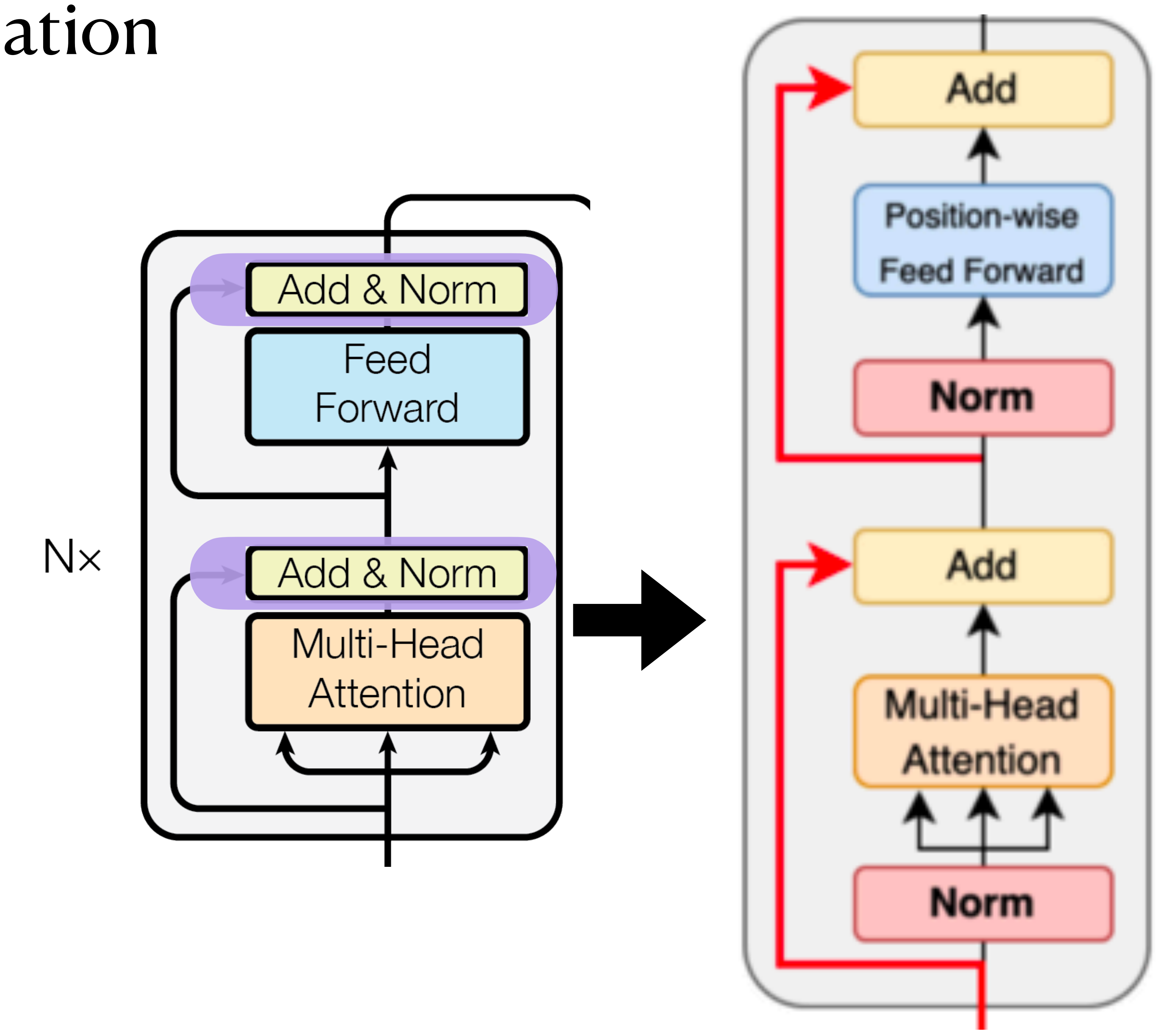
Layer Normalization

Post-LN and Pre-LN

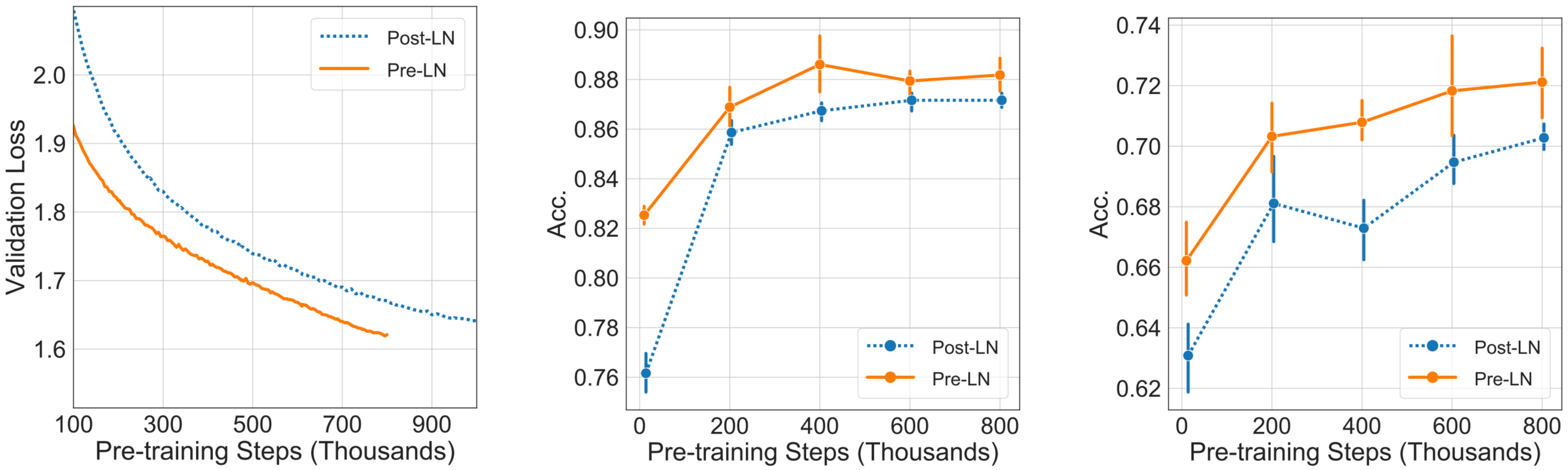


(c) Validation Loss (WMT) (d) BLEU (WMT)

Better Initialization



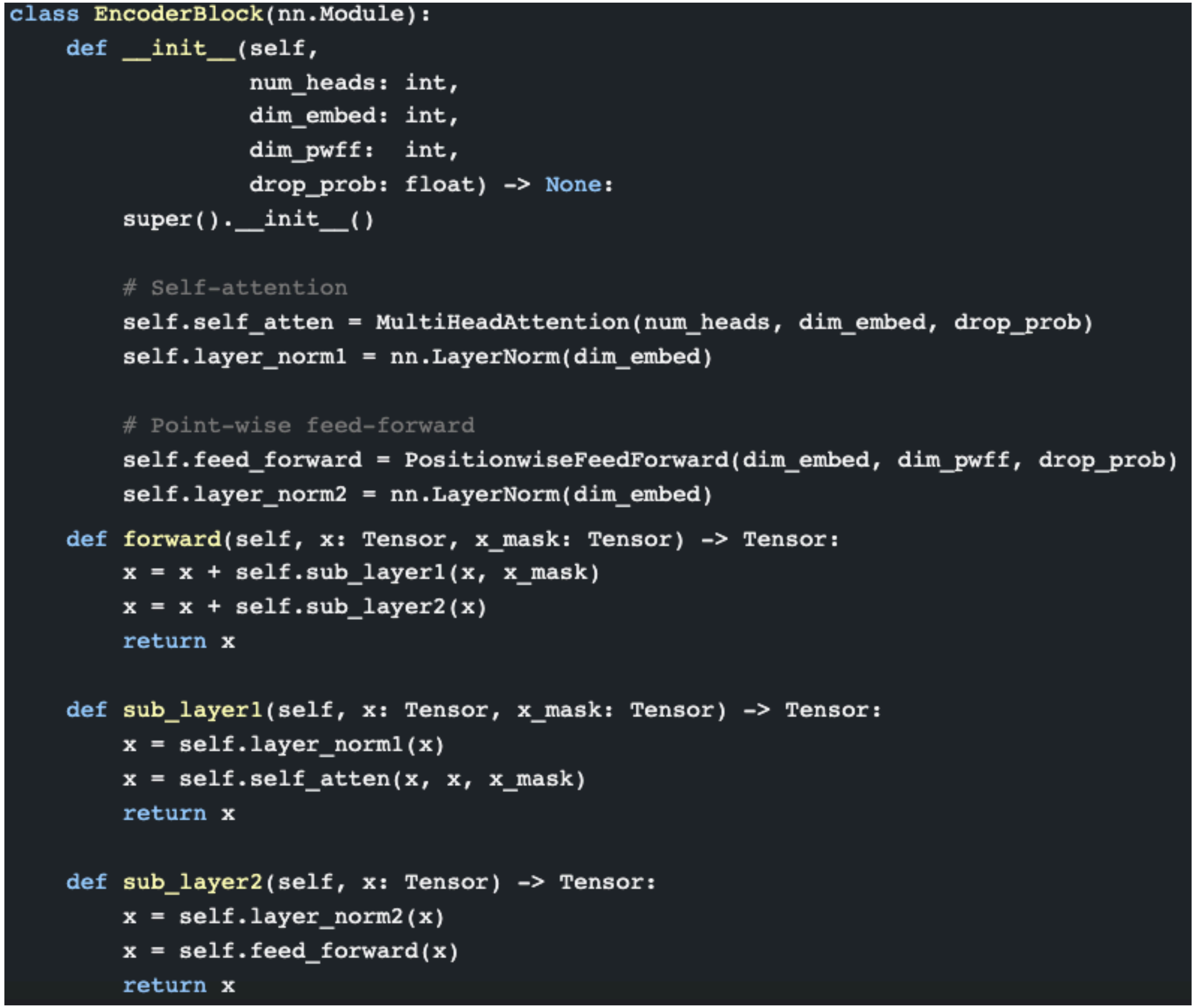
Train much faster



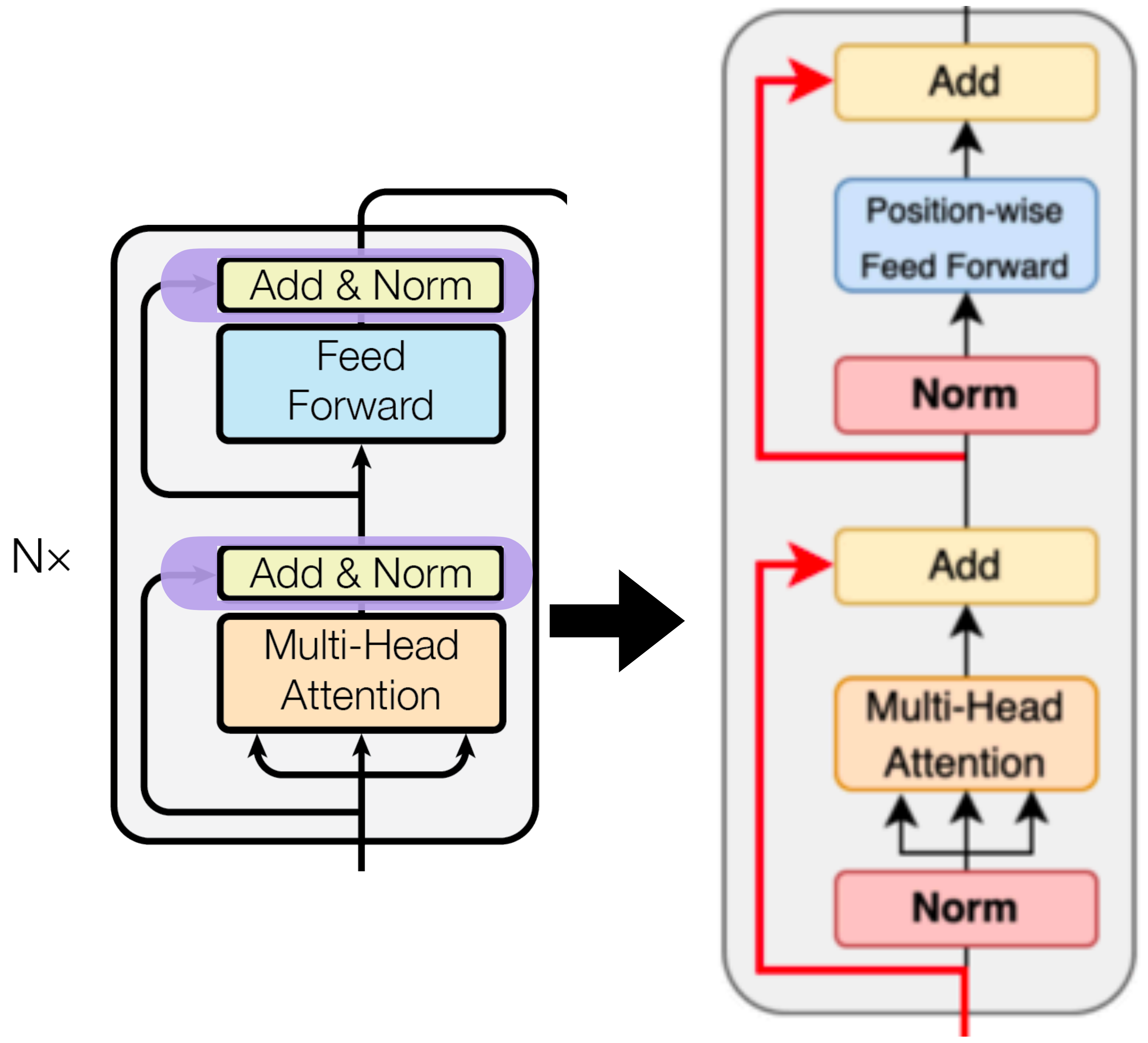
(a) Validation Loss on BERT (b) Accuracy on MRPC (c) Accuracy on RTE

Xiong, Ruibin, et al. "On layer normalization in the transformer architecture." *International Conference on Machine Learning*. PMLR, 2020

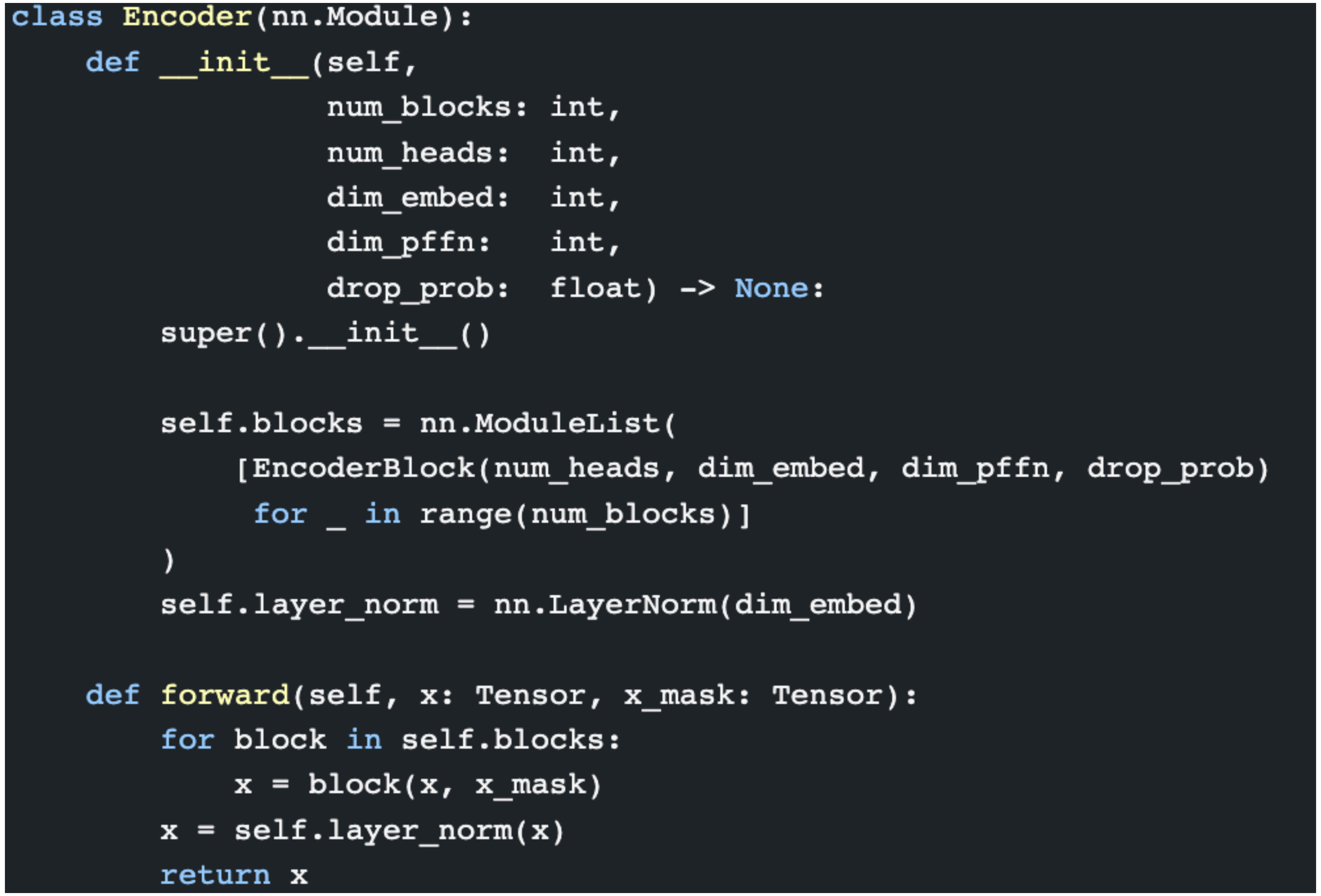
Encoder Block



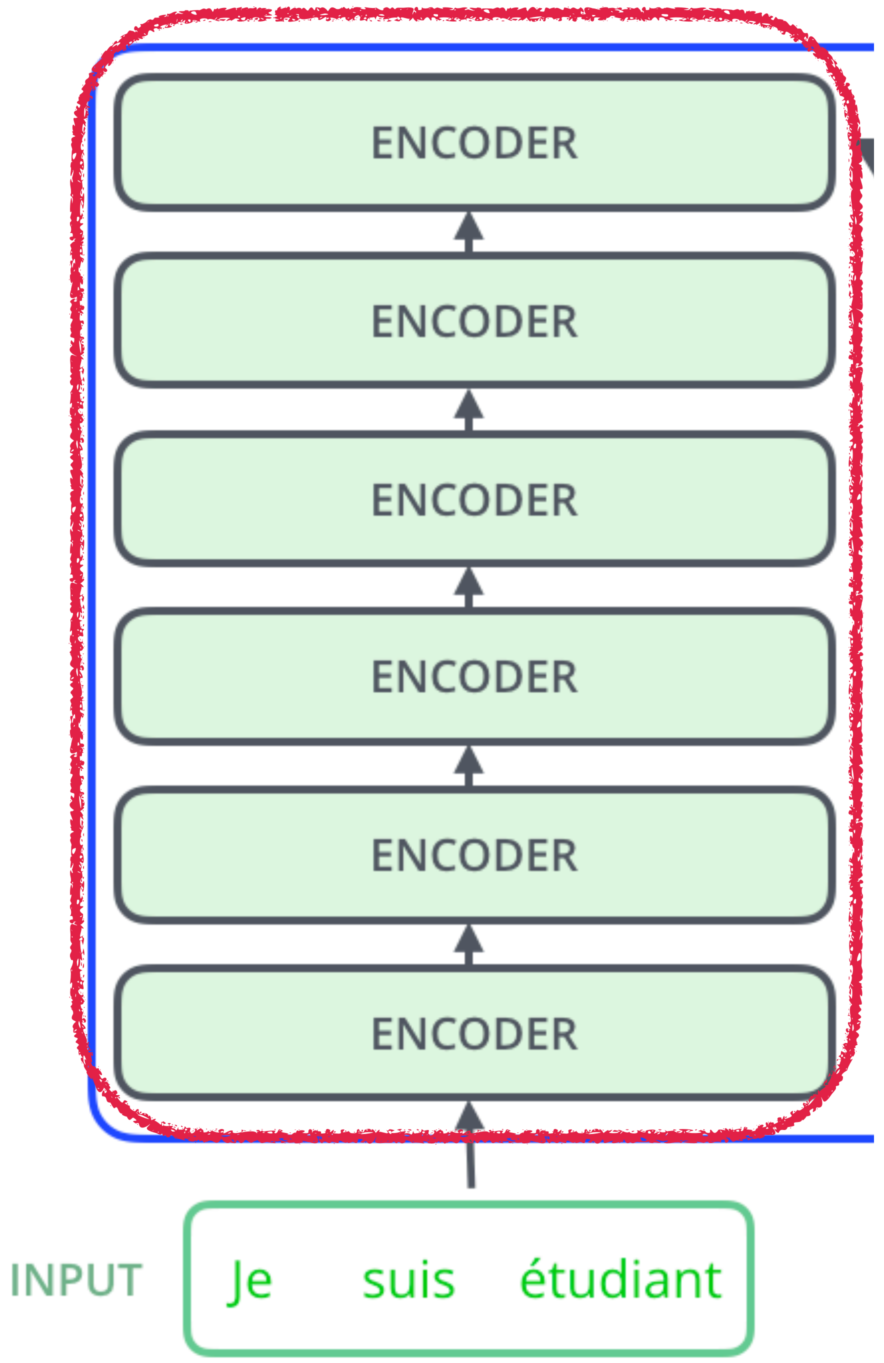
```python
class EncoderBlock(nn.Module):
    def __init__(self,
                 num_heads: int,
                 dim_embed: int,
                 dim_pwff:  int,
                 drop_prob: float) -> None:
        super().__init__()

        # Self-attention
        self.self_atten = MultiHeadAttention(num_heads, dim_embed, drop_prob)
        self.layer_norm1 = nn.LayerNorm(dim_embed)

        # Point-wise feed-forward
        self.feed_forward = PositionwiseFeedForward(dim_embed, dim_pwff, drop_prob)
        self.layer_norm2 = nn.LayerNorm(dim_embed)

    def forward(self, x: Tensor, x_mask: Tensor) -> Tensor:
        x = x + self.sub_layer1(x, x_mask)
        x = x + self.sub_layer2(x)
        return x

    def sub_layer1(self, x: Tensor, x_mask: Tensor) -> Tensor:
        x = self.layer_norm1(x)
        x = self.self_atten(x, x, x_mask)
        return x

    def sub_layer2(self, x: Tensor) -> Tensor:
        x = self.layer_norm2(x)
        x = self.feed_forward(x)
        return x
```



Encoder



```
class Encoder(nn.Module):
    def __init__(self,
                 num_blocks: int,
                 num_heads:  int,
                 dim_embed:  int,
                 dim_pffn:   int,
                 drop_prob:  float) -> None:
        super().__init__()

        self.blocks = nn.ModuleList(
            [EncoderBlock(num_heads, dim_embed, dim_pffn, drop_prob)
             for _ in range(num_blocks)]
        )
        self.layer_norm = nn.LayerNorm(dim_embed)

    def forward(self, x: Tensor, x_mask: Tensor):
        for block in self.blocks:
            x = block(x, x_mask)
        x = self.layer_norm(x)
        return x
```



References

[1] Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

[2] Transformer Coding Details – A Simple Implementation, https://kikaben.com/transformers-coding-details

[3] Transformer Architecture: The Positional Encoding, https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

[4] The Illustrated Transformer, https://jalammar.github.io/illustrated-transformer/

[5] Xiong, Ruibin, et al. "On layer normalization in the transformer architecture." *International Conference on Machine Learning*. PMLR, 2020.

Thanks