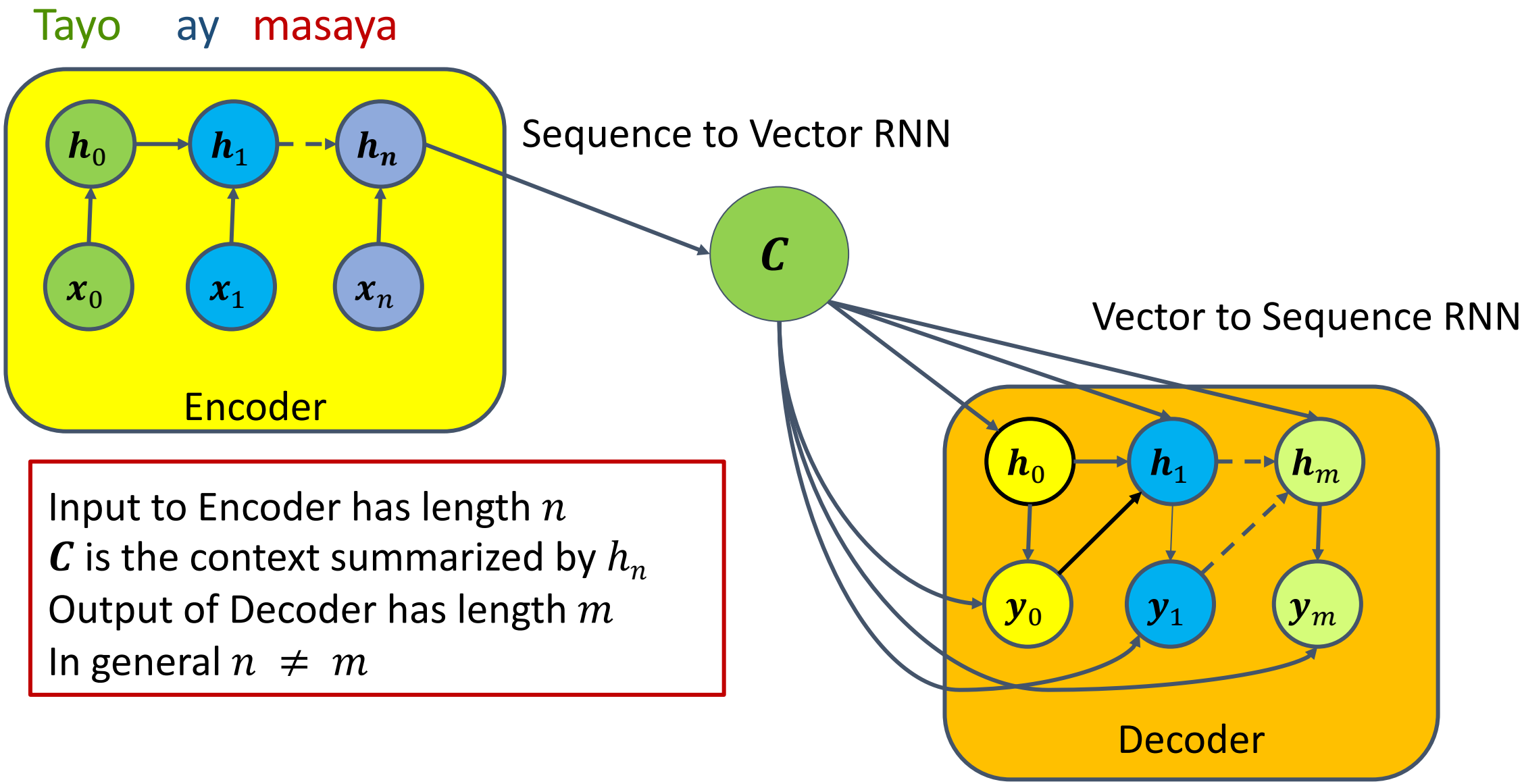
Transformers

CoE197Z/EE298Z (Deep Learning)

Rowel Atienza, Ph.D.

rowel@eee.upd.edu.ph

Encoder-Decoder Sequence-to-Sequence



Input to Encoder has length n
C is the context summarized by h_n
Output of Decoder has length m
In general $n \neq m$

seq2seq

RNN

Serial

Difficult to parallelize

Uni-directional

Bi-directional version is much slower

Susceptible to catastrophic forgetting

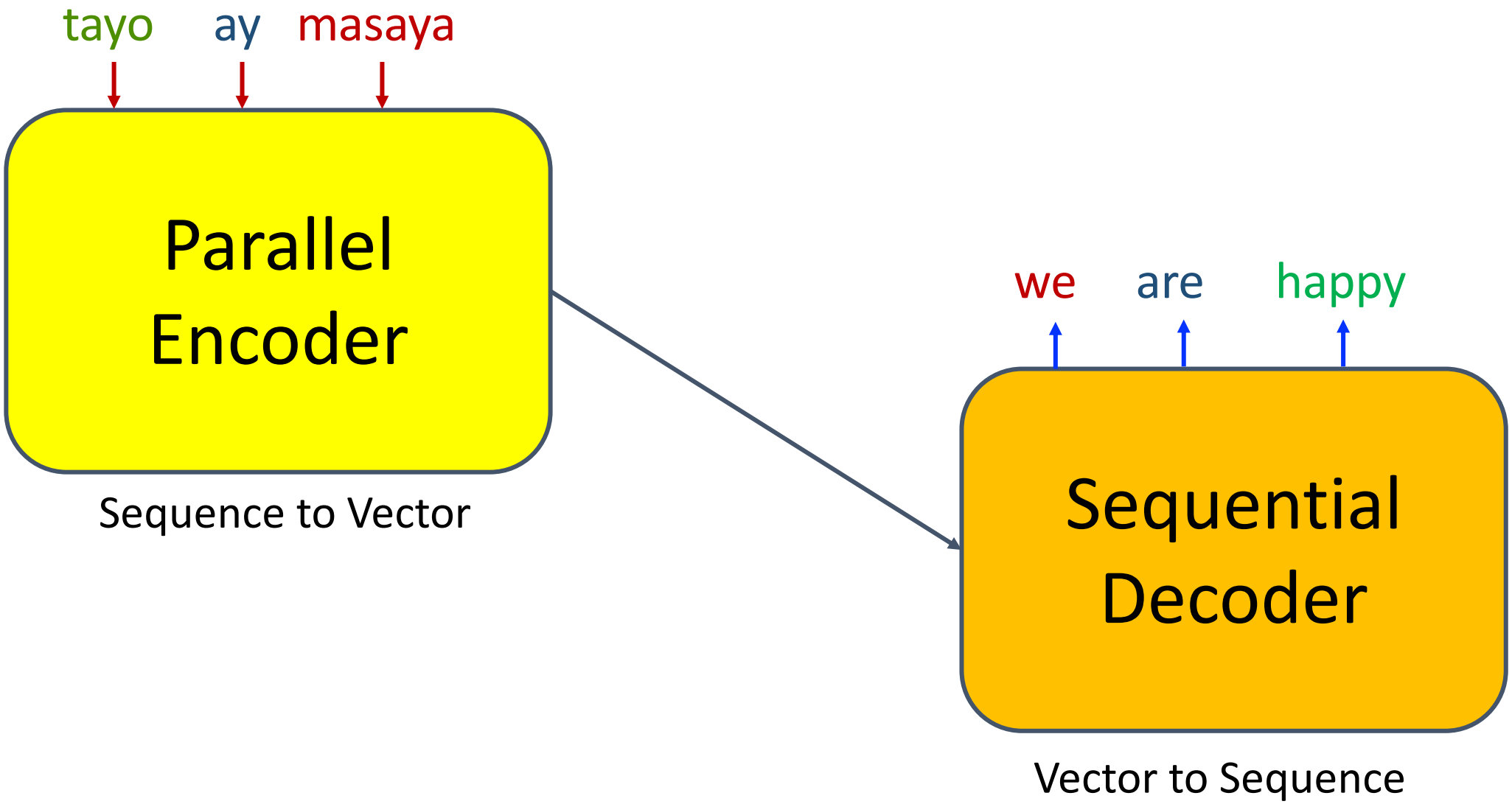
Transformer

Parallel

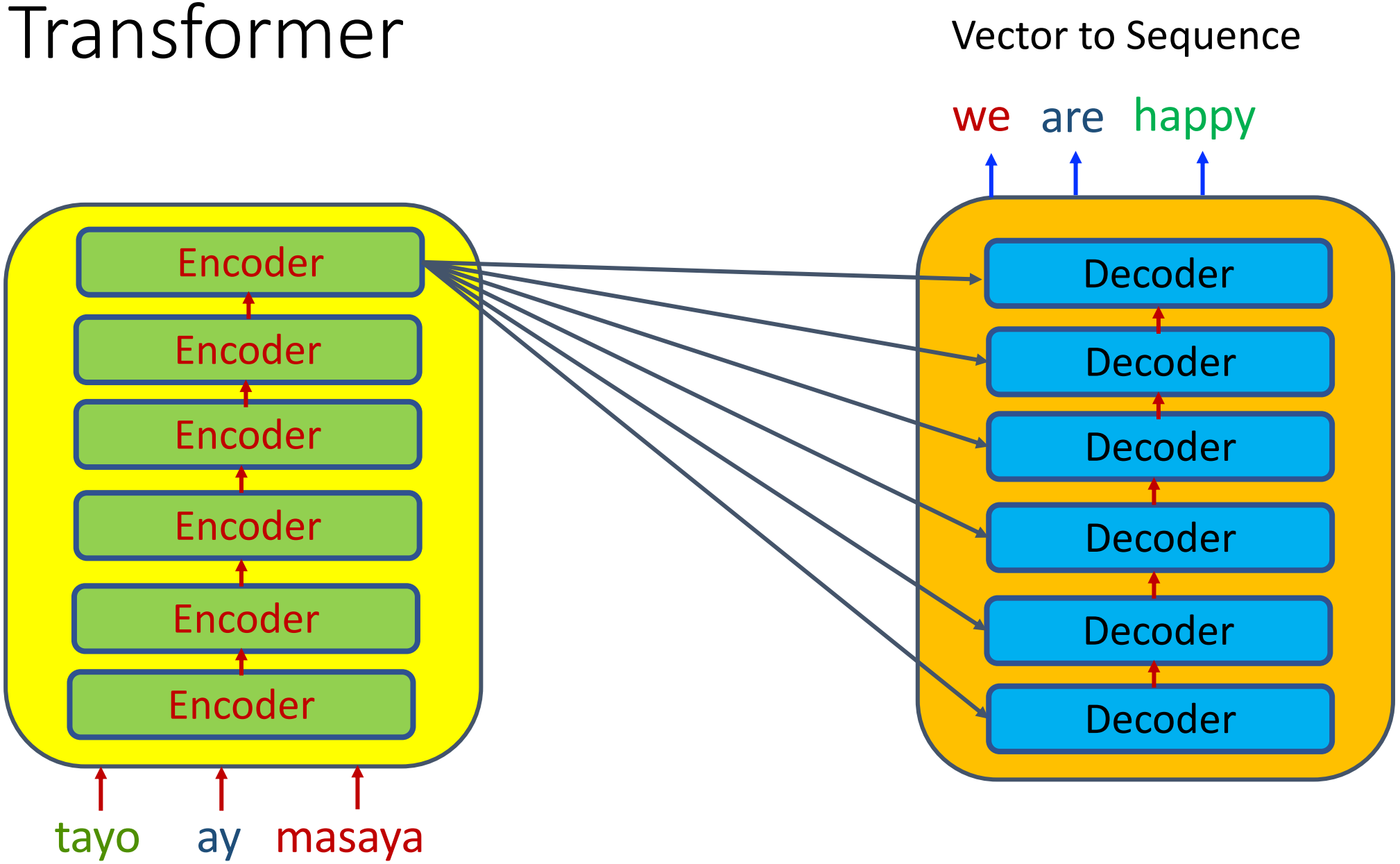
Bidirectional

Not susceptible to catastrophic forgetting

Transformer



Transformer

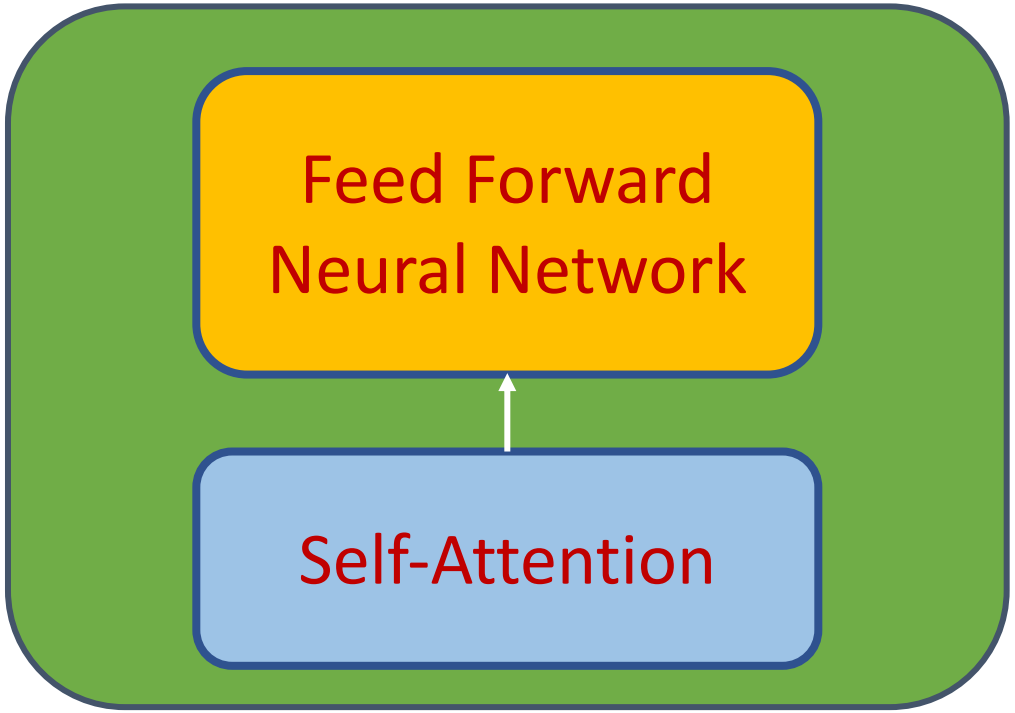


Transformer Encoder Unit Details

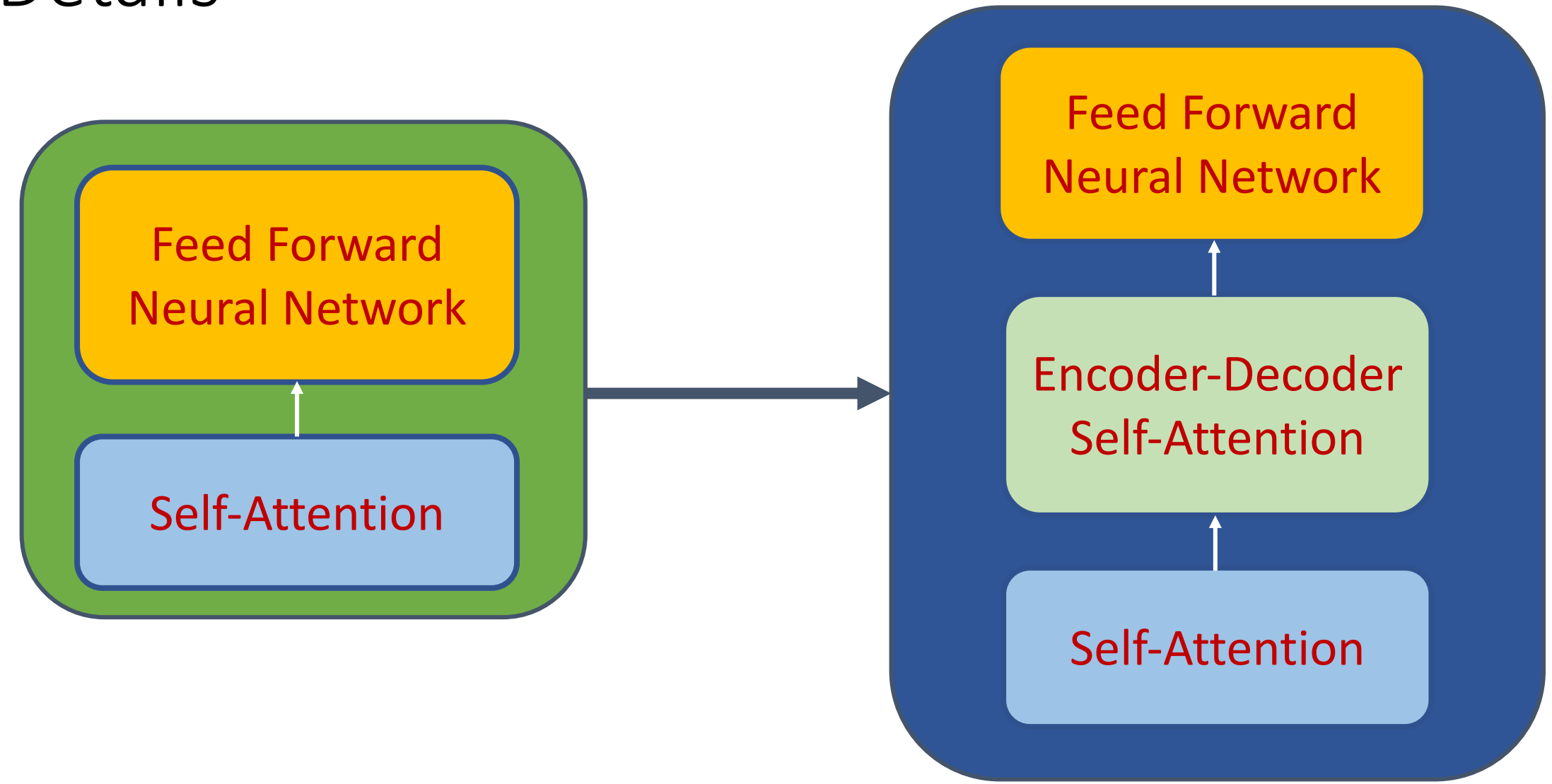
No recurrence (No RNN)

No CNN

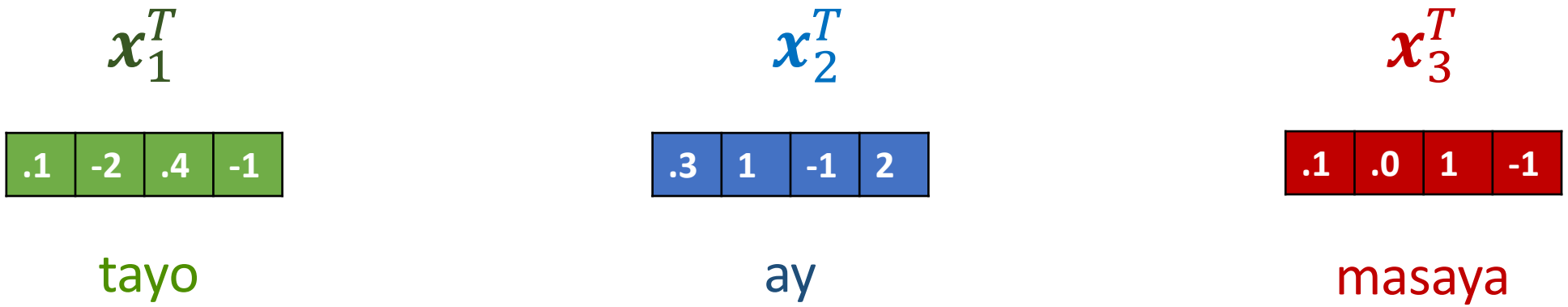
Operations: Linear, Norm, Matrix Multiply, Dot Product, Softmax



Transformer Encoder and Decoder Unit Details

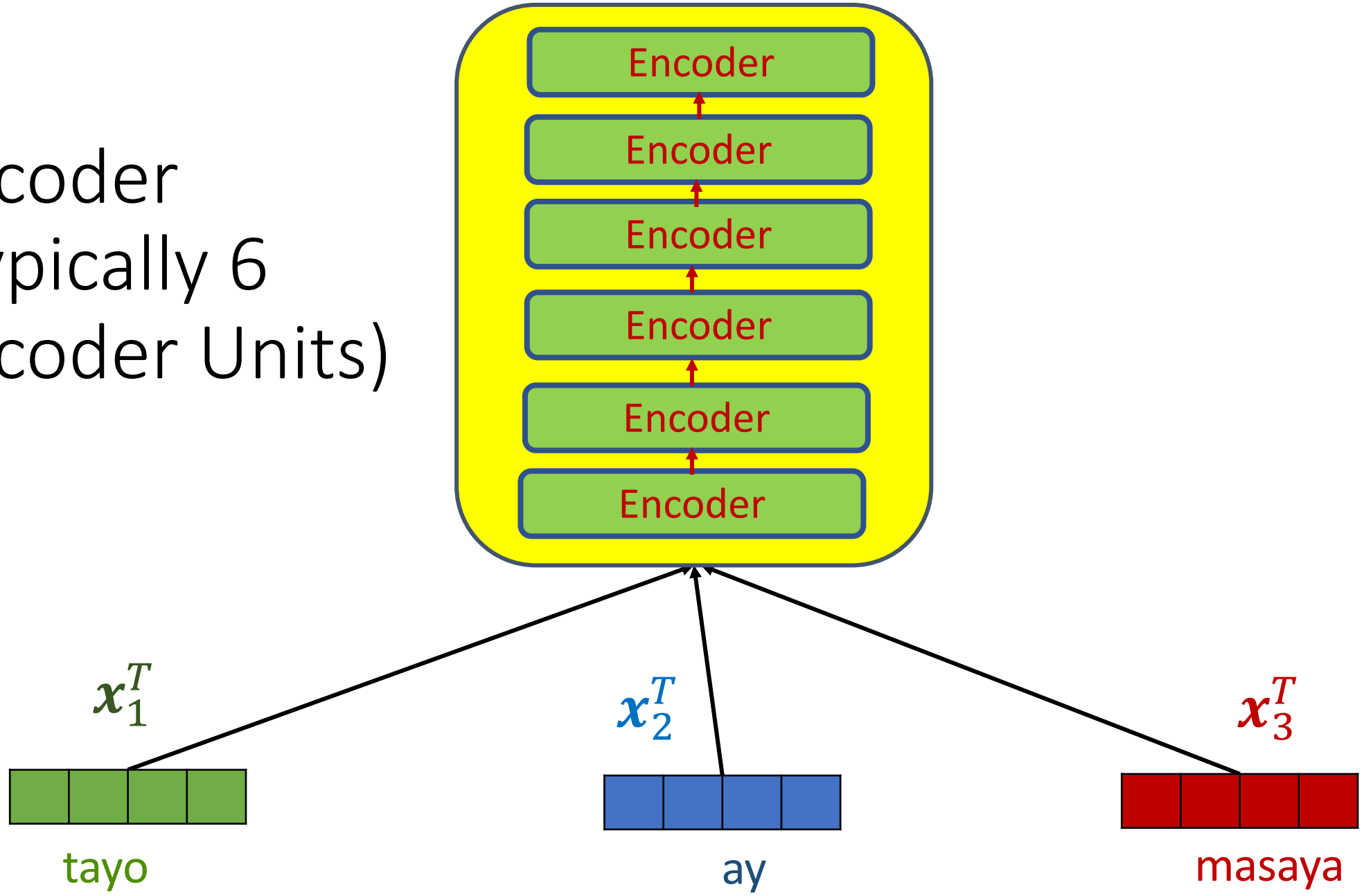


Input Embedding is an $n - dim$ vector

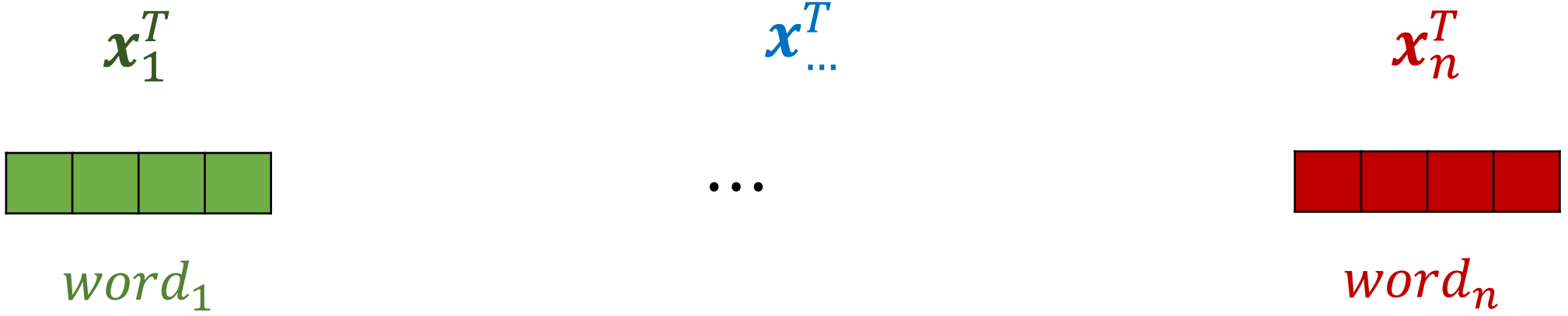


Example: Each word is converted into a 512-dim embedding vector.
In the simple example above, it is 4-dim.

Encoder (Typically 6 Encoder Units)

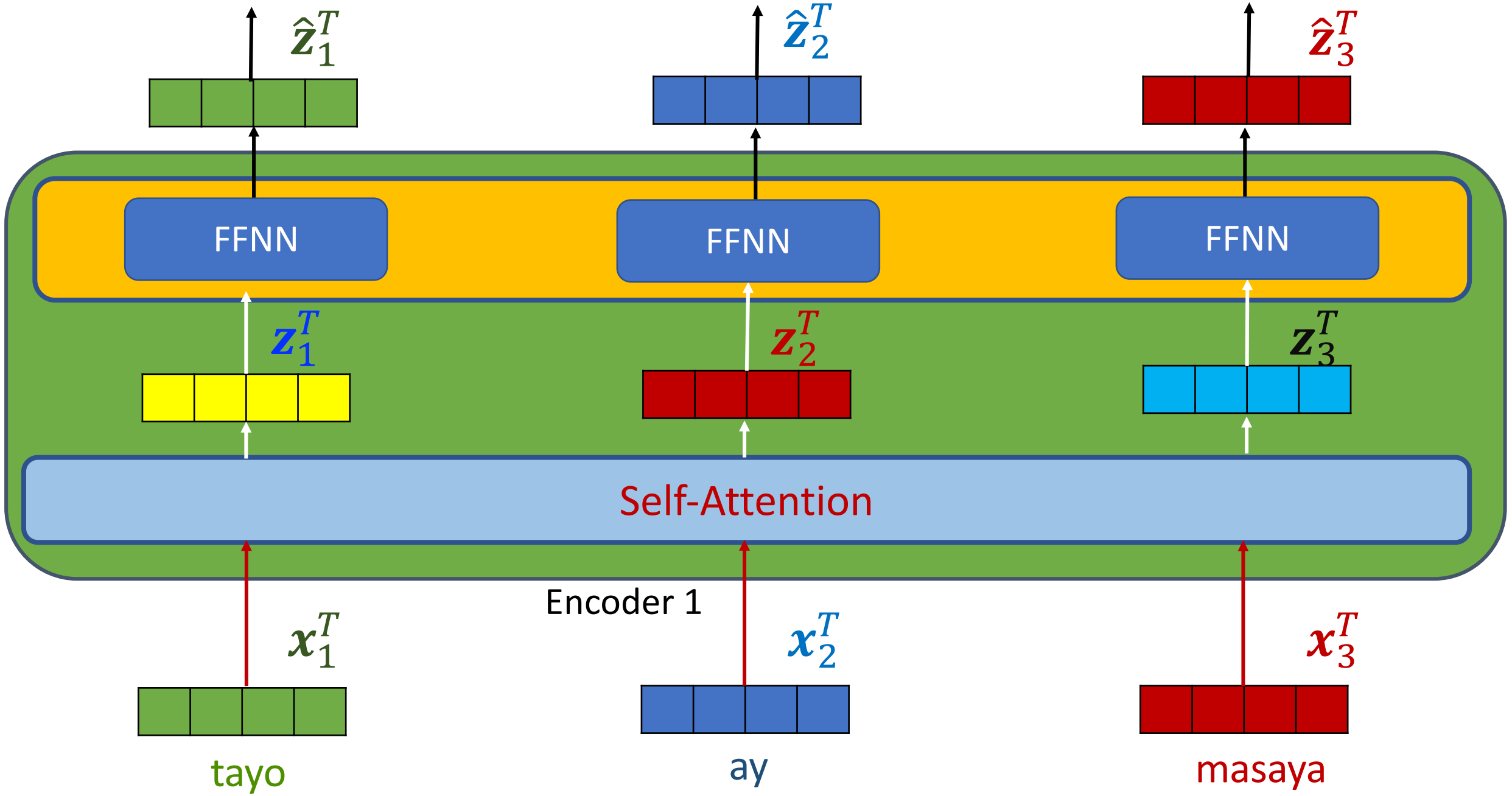


The Length of the Input is n

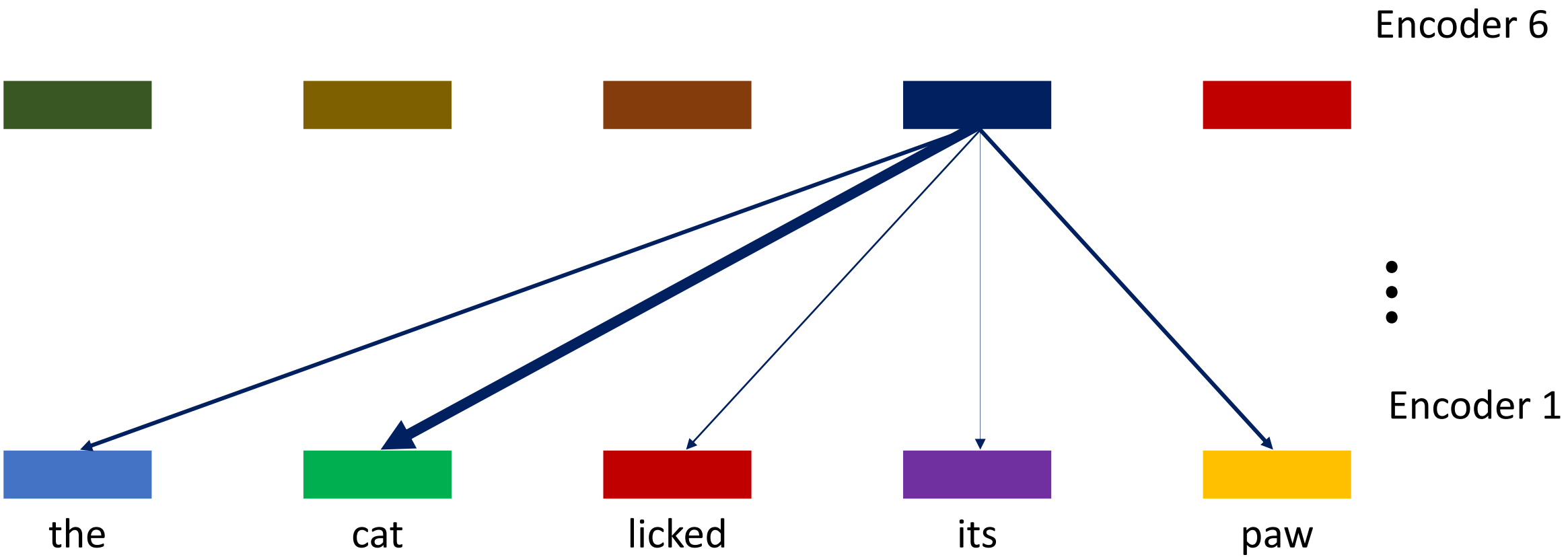


Example: n could be the maximum possible length of a sentence.

Encoder with Latent Variables z_i



Attention between 2 words



Self-Attention

Embedding

tayo x_1^T ay x_2^T masaya x_3^T

$$X = \begin{bmatrix} x_1^T \\ x_2^T \\ x_3^T \end{bmatrix}$$ Encoder 1 Inputs

Attention Layer 1 Learnable Parameters

W^Q

$q_1^T = x_1^T W^Q$ $q_2^T = x_2^T W^Q$ $q_3^T = x_3^T W^Q$

Queries Q

$Q = XW^Q$

W^K

$k_1^T = x_1^T W^K$ $k_2^T = x_2^T W^K$ $k_3^T = x_3^T W^K$

Keys

$K = XW^K$

W^V

$v_1^T = x_1^T W^V$ $v_2^T = x_2^T W^V$ $v_3^T = x_3^T W^V$

Values V

$V = XW^V$

tayo

x_1^T

ay

x_2^T

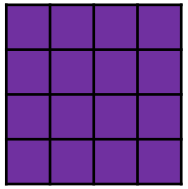
masaya

x_3^T

$$X = \begin{bmatrix} x_1^T \\ x_2^T \\ x_3^T \end{bmatrix}$$

Encoder 1 Inputs

W^Q

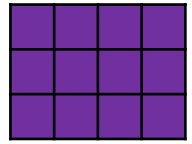


$q_1^T = x_1^T W^Q$

$q_2^T = x_2^T W^Q$

$q_3^T = x_3^T W^Q$

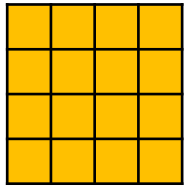
Queries



Q

$$Q = XW^Q$$

W^K

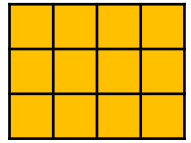


$k_1^T = x_1^T W^K$

$k_2^T = x_2^T W^K$

$k_3^T = x_3^T W^K$

Keys



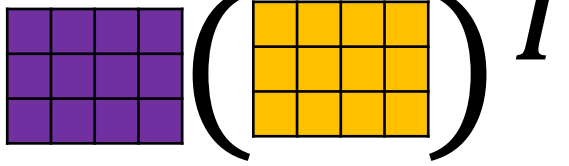
$$K = XW^K$$

Scores

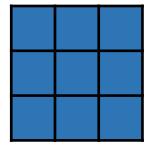
$$\begin{bmatrix} s_{11} = q_1^T k_1 \\ s_{12} = q_1^T k_2 \\ s_{13} = q_1^T k_3 \end{bmatrix}^T$$

$$\begin{bmatrix} s_{21} = q_2^T k_1 \\ s_{22} = q_2^T k_2 \\ s_{23} = q_2^T k_3 \end{bmatrix}^T$$

$$\begin{bmatrix} s_{31} = q_3^T k_1 \\ s_{32} = q_3^T k_2 \\ s_{33} = q_3^T k_3 \end{bmatrix}^T$$



$$= \quad = S$$



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

d_k is keys/queries dim (e.g. 4)

$$Attention = softmax\left(\frac{\blacksquare\blacksquare^T}{\sqrt{d_k}}\right)\blacksquare$$

$$Attention(Q, K, V) = Z = \blacksquare$$

Multi-Head
(eg 8-head)

tayo x_1^T

ay x_2^T

masaya x_3^T

$$X = \begin{bmatrix} x_1^T \\ x_2^T \\ x_3^T \end{bmatrix}$$

Encoder 1 Inputs

W_1^Q

Queries

$Q_1 = XW_1^Q$

W_8^Q

Queries

$Q_8 = XW_8^Q$

W_1^K

Keys

$K_1 = XW_1^K$

W_8^K

Keys

$K_8 = XW_8^K$

W_1^V

Values

$V_1 = XW_1^V$

Head 1: Z_1

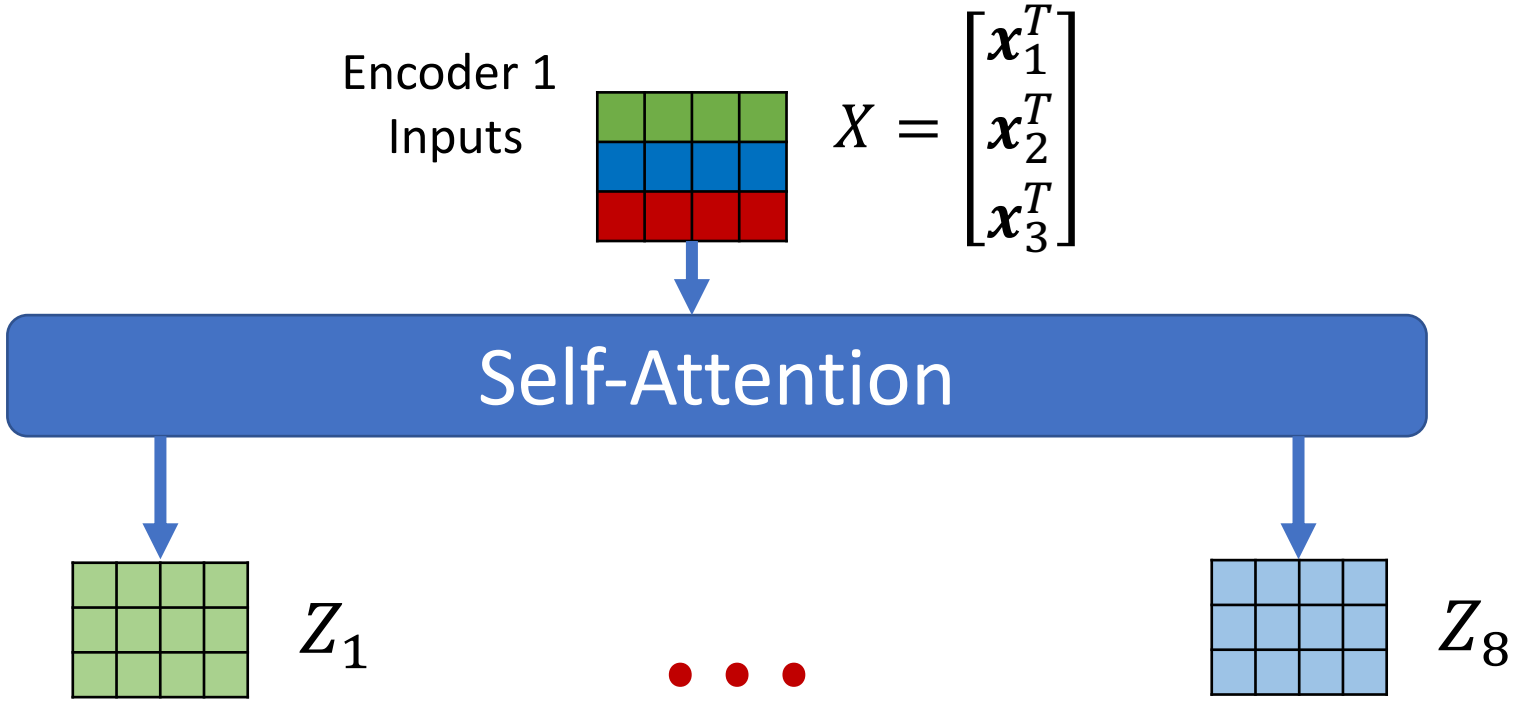
W_8^V

Values

$V_8 = XW_8^V$

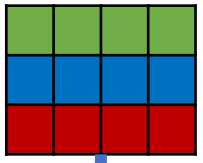
Head 8: Z_8

Multi-Head
(eg 8-head)



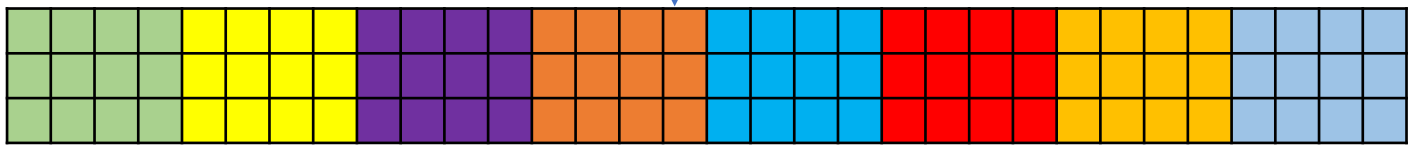
Multi-Head
(eg 8-head)
Merge Outputs
Apply Weights

Encoder 1
Inputs



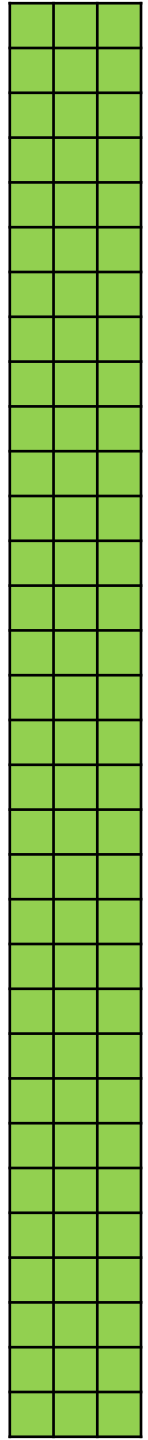
$$X = \begin{bmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \boldsymbol{x}_3^T \end{bmatrix}$$

Self-Attention



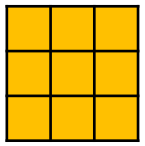
$$cat(Z_1, \ldots, Z_8)$$

$$\times$$

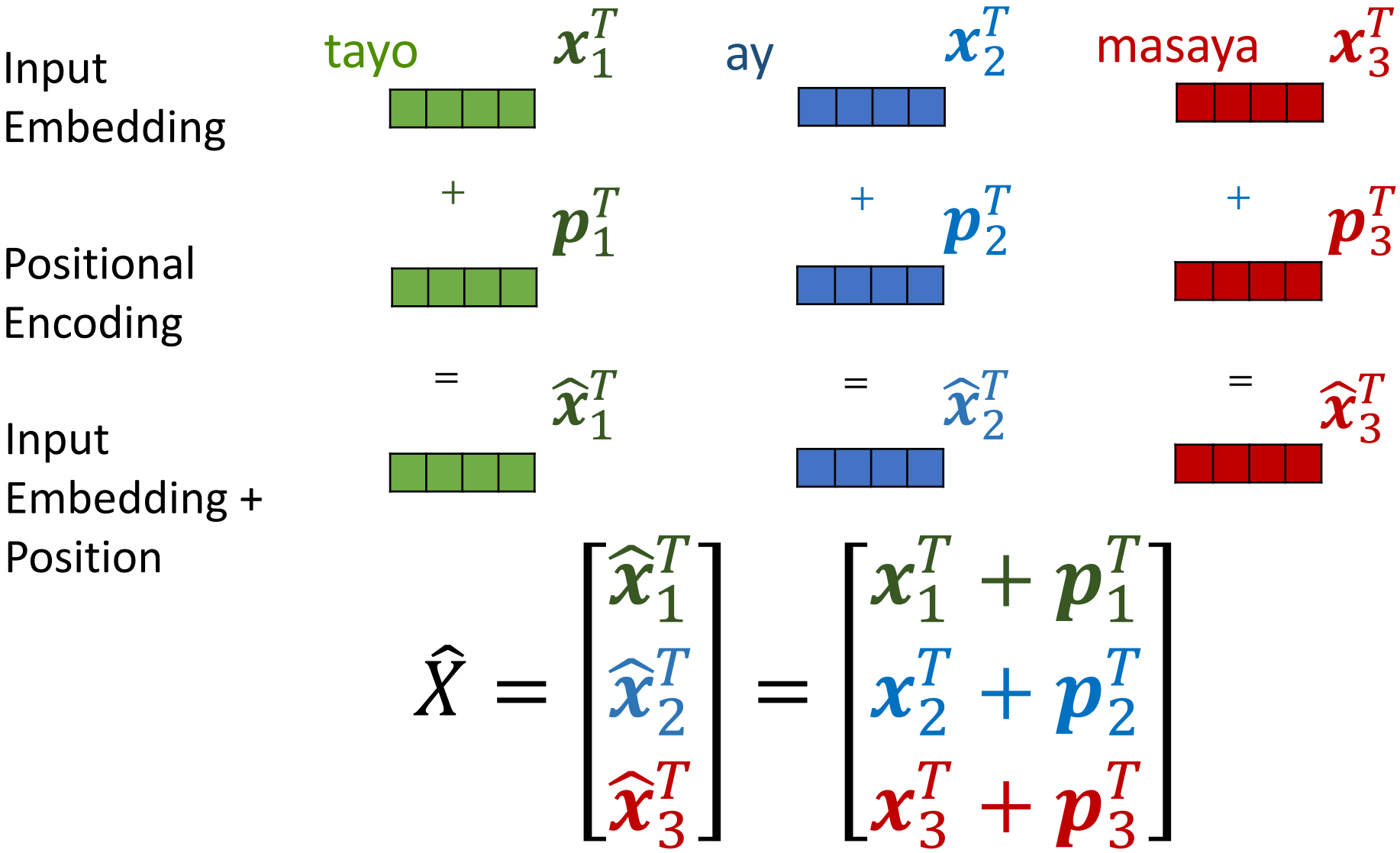


$$W^O$$

$$= Z$$



Adding Position Info to Inputs



$$\hat{X} = \begin{bmatrix} \hat{x}_1^T \\ \hat{x}_2^T \\ \hat{x}_3^T \end{bmatrix} = \begin{bmatrix} x_1^T + p_1^T \\ x_2^T + p_2^T \\ x_3^T + p_3^T \end{bmatrix}$$

Positional Encoding

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_k}}}\right) \quad dim = 2i \ \ is\ even$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_k}}}\right) \quad dim = 2i + 1 \ \ is\ odd$$

$pos = 0,1, \ldots n_{pos-1}$

$dim = 0,1, \ldots n_{dim-1}$

Other positional encoding methods: learnable

Assuming $n_{pos-1} = 2$ and $n_{dim-1} = 3$

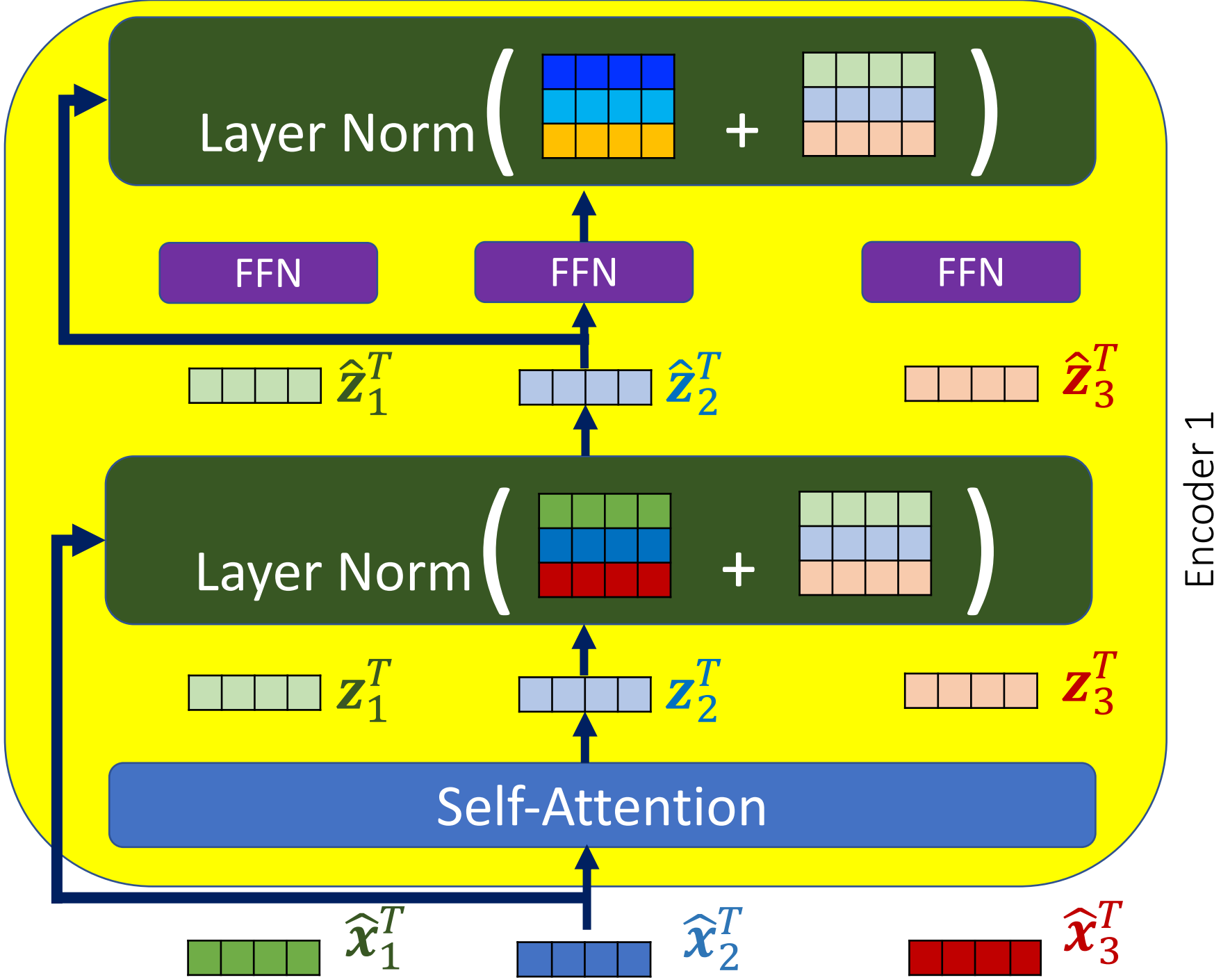
pos	dim			
	0	1	2	3
0	$\sin\left(\frac{0}{10000^{0/4}}\right)$	$\cos\left(\frac{0}{10000^{0/4}}\right)$	$\sin\left(\frac{0}{10000^{2/4}}\right)$	$\cos\left(\frac{0}{10000^{2/4}}\right)$
1	$\sin\left(\frac{1}{10000^{0/4}}\right)$	$\cos\left(\frac{1}{10000^{0/4}}\right)$	$\sin\left(\frac{1}{10000^{2/4}}\right)$	$\cos\left(\frac{1}{10000^{2/4}}\right)$
2	$\sin\left(\frac{2}{10000^{0/4}}\right)$	$\cos\left(\frac{2}{10000^{0/4}}\right)$	$\sin\left(\frac{2}{10000^{2/4}}\right)$	$\cos\left(\frac{2}{10000^{2/4}}\right)$

Improvements:

Residual Connections

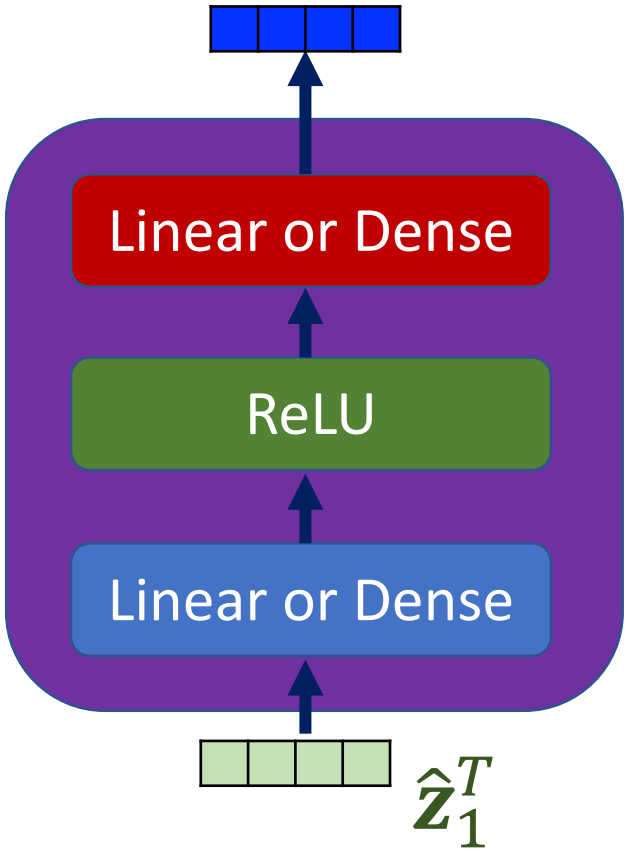
Layer Norm

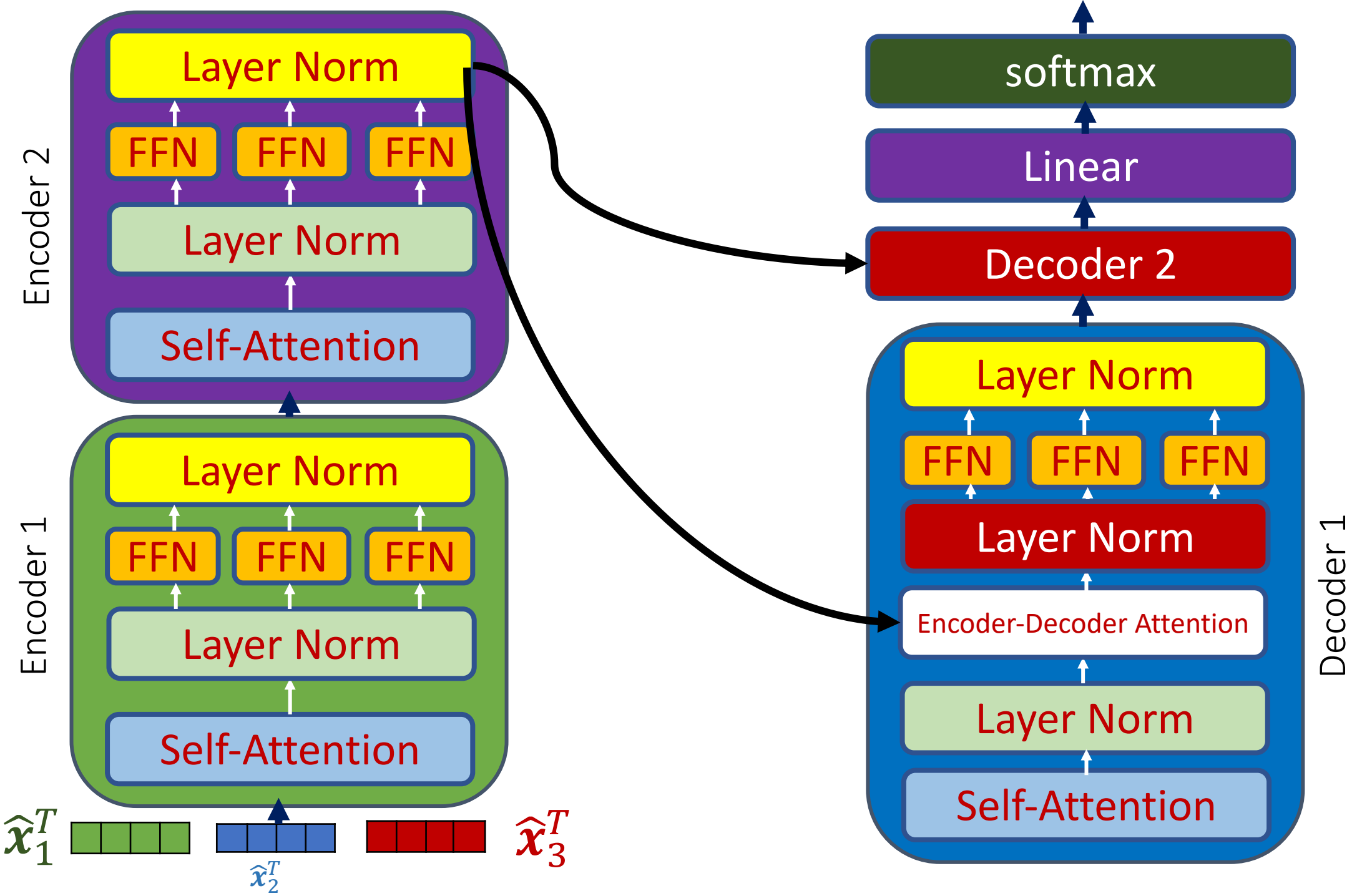
FFN (MLP)

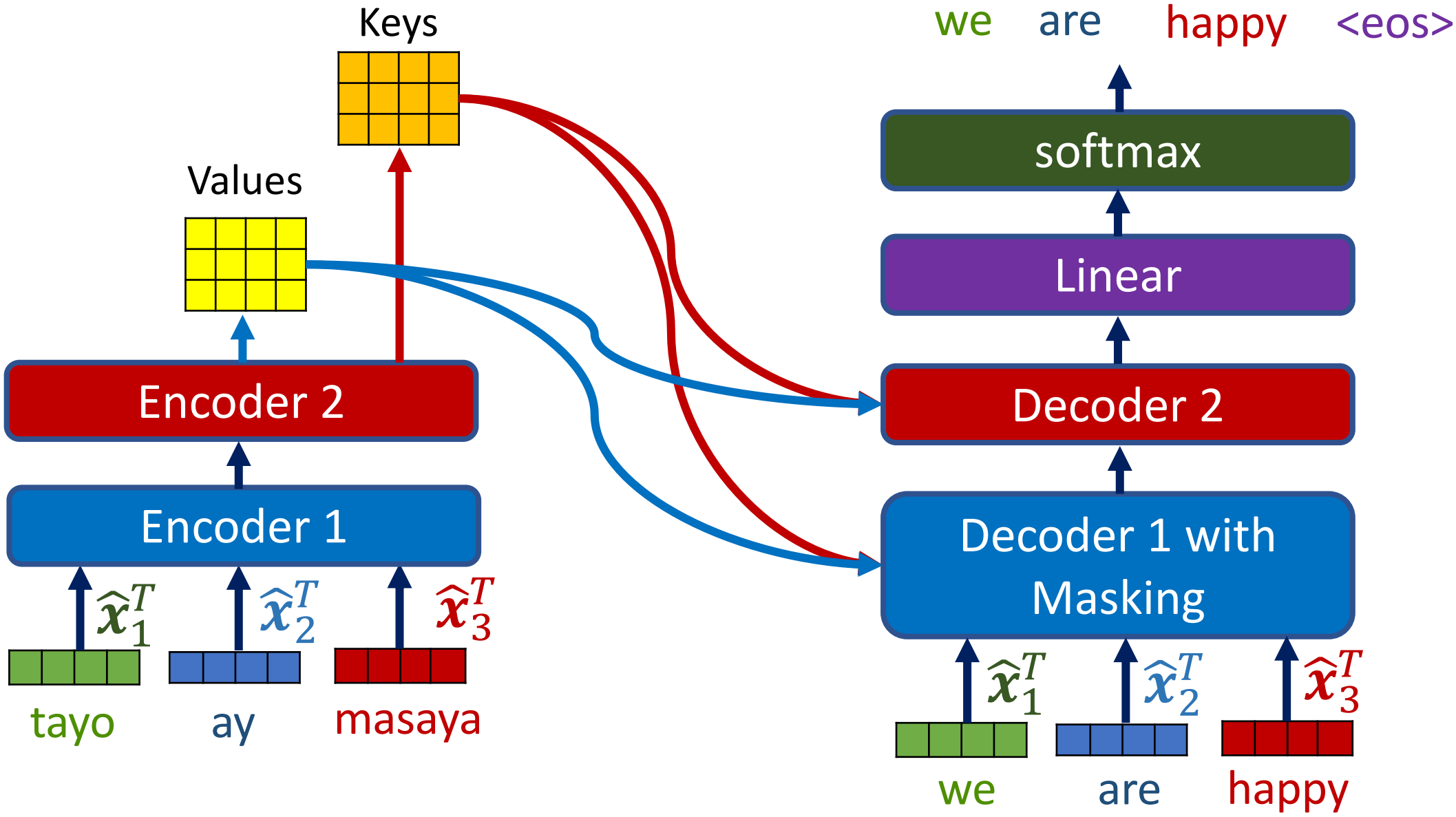


FFN: Feed Forward Neural Network

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




Encoder 2
Layer Norm
FFN
FFN
FFN
Layer Norm
Self-Attention
Encoder 1
Layer Norm
FFN
FFN
FFN
Layer Norm
Self-Attention
$\hat{x}_1^T$
$\hat{x}_2^T$
$\hat{x}_3^T$
softmax
Linear
Decoder 2
Decoder 1
Layer Norm
FFN
FFN
FFN
Layer Norm
Encoder-Decoder Attention
Layer Norm
Self-Attention


Keys
Values
Encoder 2
Encoder 1
$\hat{x}_1^T$
$\hat{x}_2^T$
$\hat{x}_3^T$
tayo
ay
masaya
we
are
happy
<eos>
softmax
Linear
Decoder 2
Decoder 1 with Masking
$\hat{x}_1^T$
$\hat{x}_2^T$
$\hat{x}_3^T$
we
are
happy

Vision Transformer

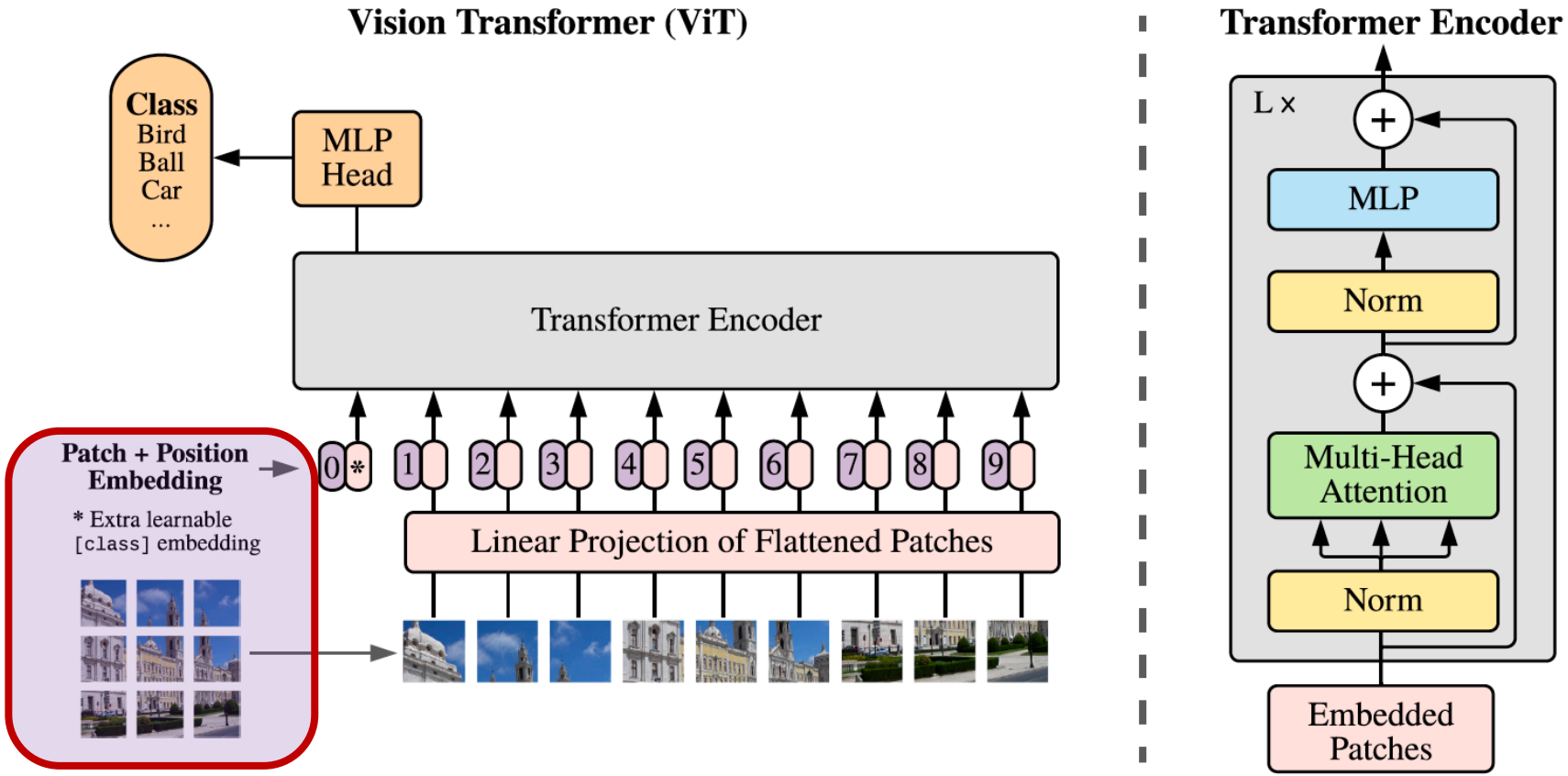
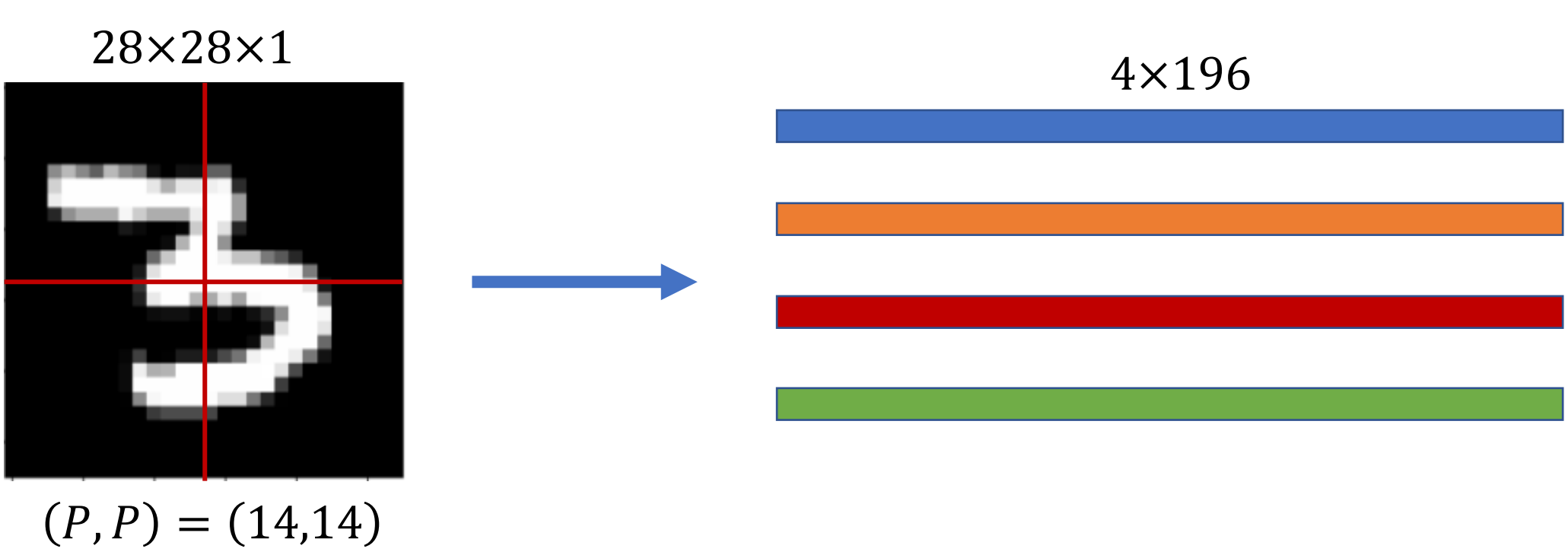


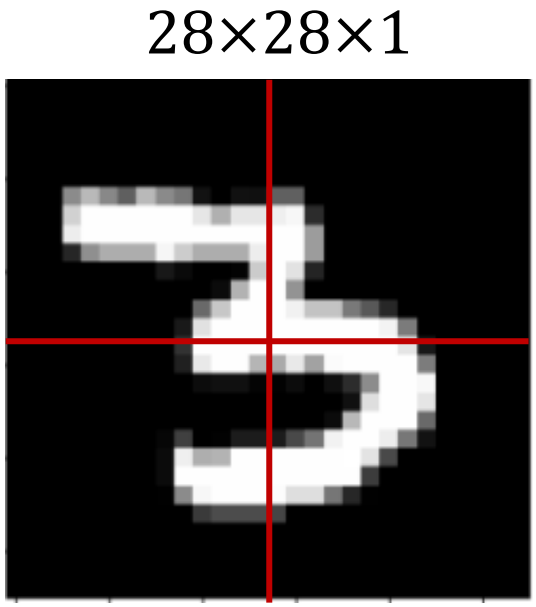
Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings to the resulting sequence of vectors, and feed the patches to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

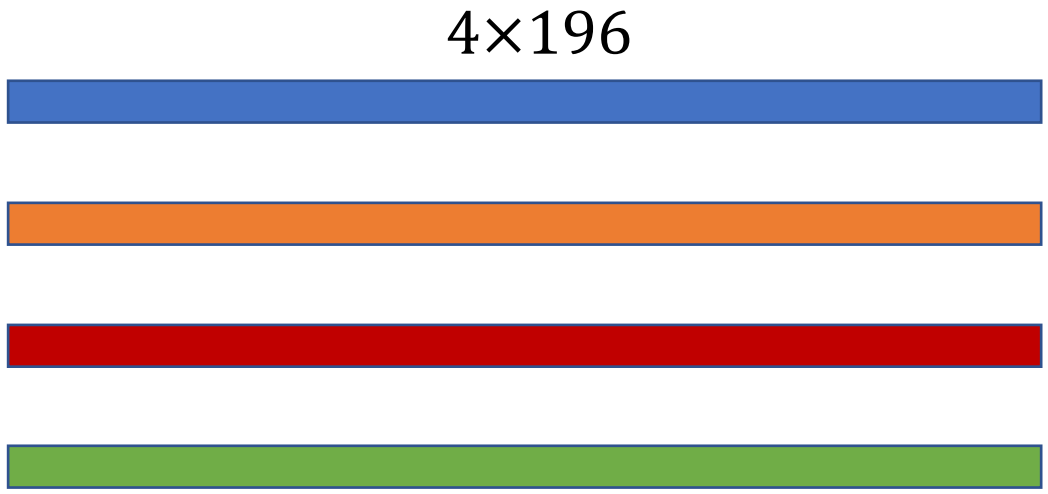
```python
class ViT(nn.Module):
    def __init__(self, *, image_size, patch_size, num_classes, dim, depth, heads, mlp_dim, channels = 3):
        super().__init__()
        assert image_size % patch_size == 0, 'image dimensions must be divisible by the patch size'
        num_patches = (image_size // patch_size) ** 2
        patch_dim = channels * patch_size ** 2

        self.patch_size = patch_size

        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.patch_to_embedding = nn.Linear(patch_dim, dim)
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.transformer = Transformer(dim, depth, heads, mlp_dim)

        self.to_cls_token = nn.Identity()
```



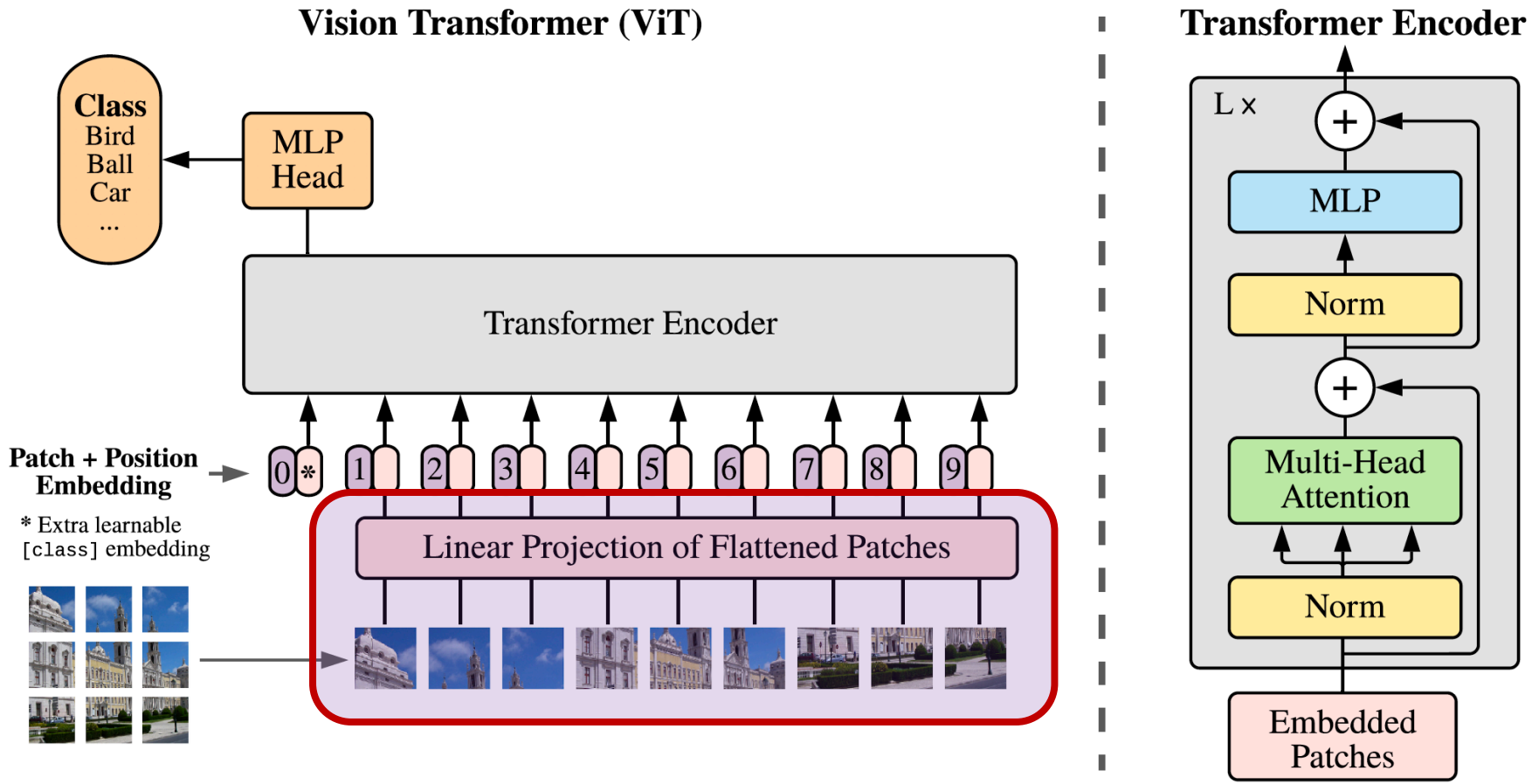
3.1 VISION TRANSFORMER (VIT)

Our Transformer for images follows the architecture designed for NLP. Figure 1 depicts the setup. The standard Transformer receives as input a 1D sequence of token embeddings. To handle 2D images, we reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$. (H, W) is the resolution of the original image and (P, P) is the resolution of each image patch. $N = HW/P^2$ is then the effective sequence length for the Transformer. The Transformer uses constant widths through all of its layers, so a trainable linear projection maps each vectorized patch to the model dimension D (Eq. 1), the output of which we refer to as our patch embeddings.

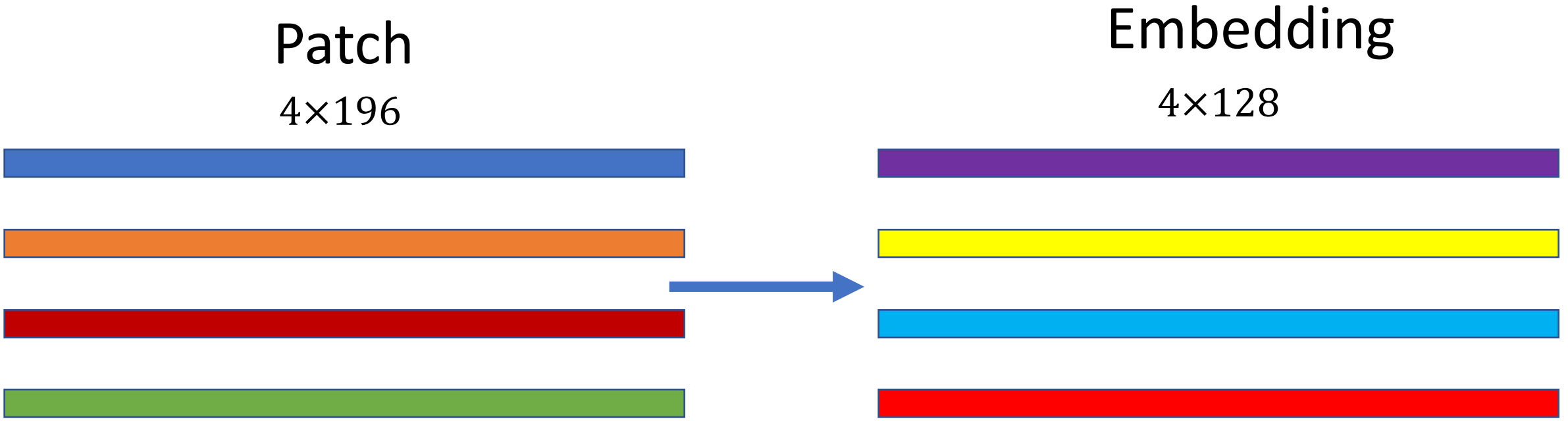





```
def forward(self, img, mask = None):
    p = self.patch_size

    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = p, p2 = p)
```




```
    self.patch_to_embedding = nn.Linear(patch_dim, dim)
def forward(self, img, mask = None):
    p = self.patch_size

    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = p, p2 = p)
    x = self.patch_to_embedding(x)
```




```
self.patch_to_embedding = nn.Linear(patch_dim, dim)
```

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \mathbf{x}_p^2\mathbf{E};\ \cdots\ ;\ \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$$

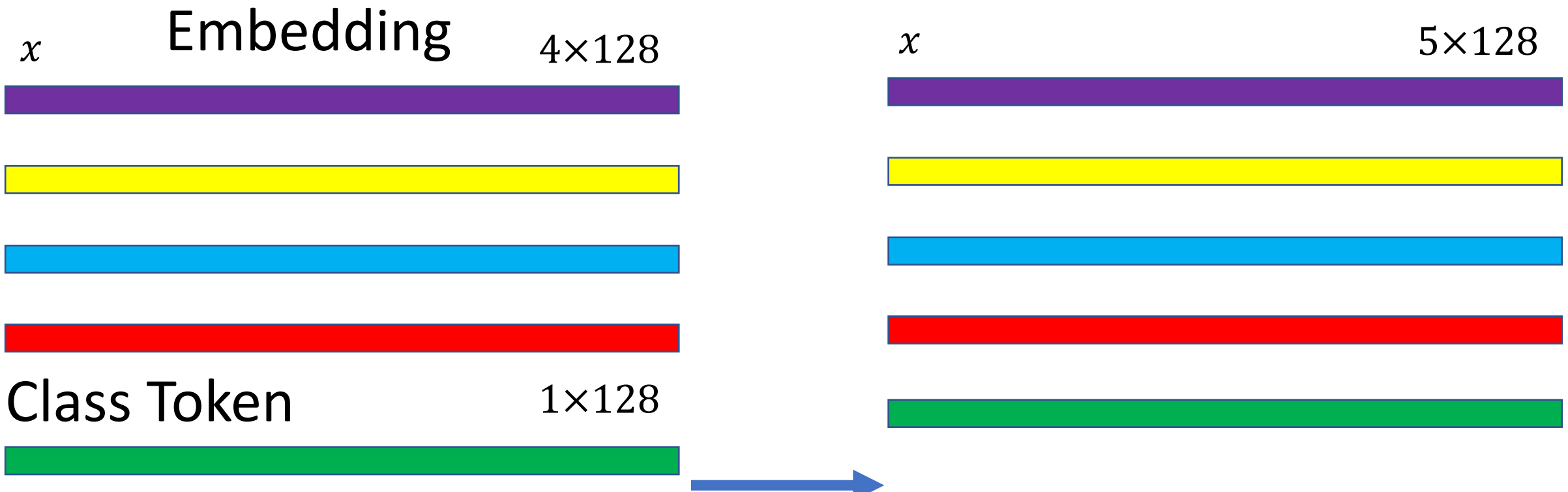
Class Token

1×128

Similar to BERT's `[class]` token, we prepend a learnable embedding to the sequence of embedded patches ($\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$), whose state at the output of the Transformer encoder ($\mathbf{z}_0^L$) serves as the

```
    self.cls_token = nn.Parameter(torch.randn(1, 1, dim))

def forward(self, img, mask = None):
    p = self.patch_size

    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = p, p2 = p
    x = self.patch_to_embedding(x)

    cls_tokens = self.cls_token.expand(img.shape[0], -1, -1)
```

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$$



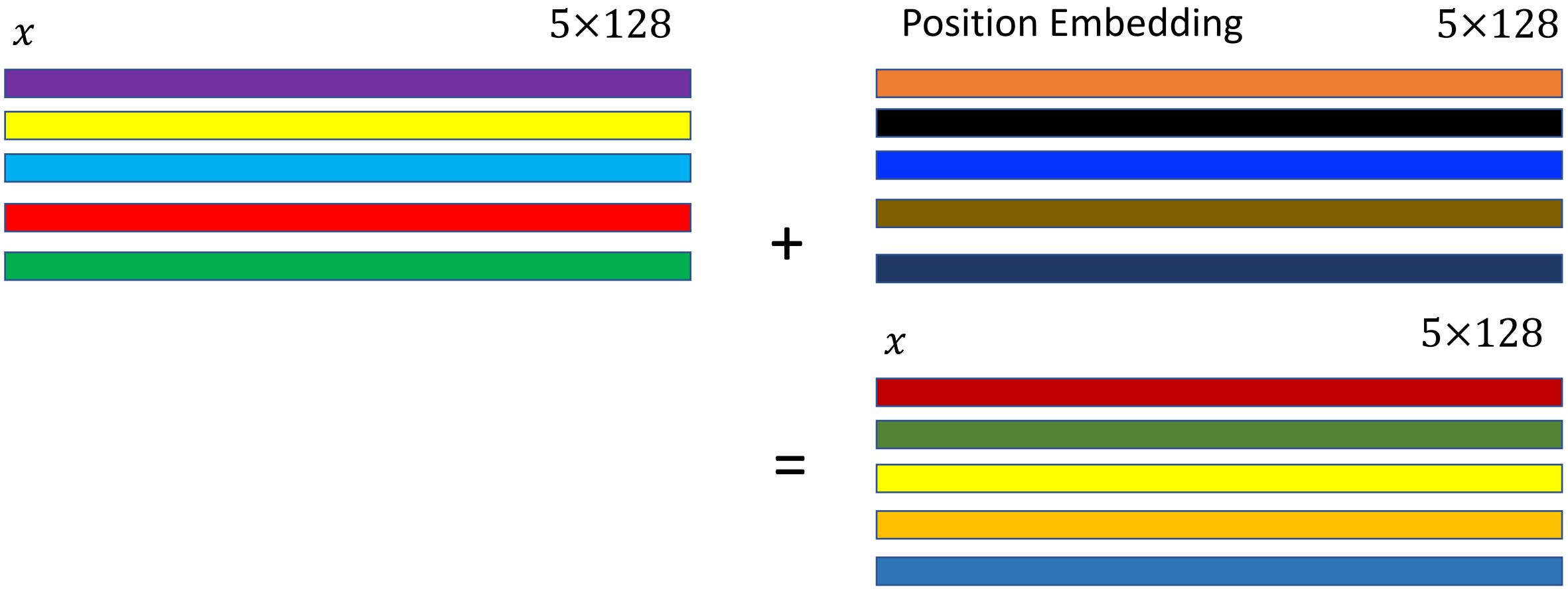

```
x = torch.cat((cls_tokens, x), dim=1)
```

```
def forward(self, img, mask = None):
    p = self.patch_size

    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = p, p2 = p)
    x = self.patch_to_embedding(x)

    cls_tokens = self.cls_token.expand(img.shape[0], -1, -1)
    x = torch.cat((cls_tokens, x), dim=1)
```

Position Embedding

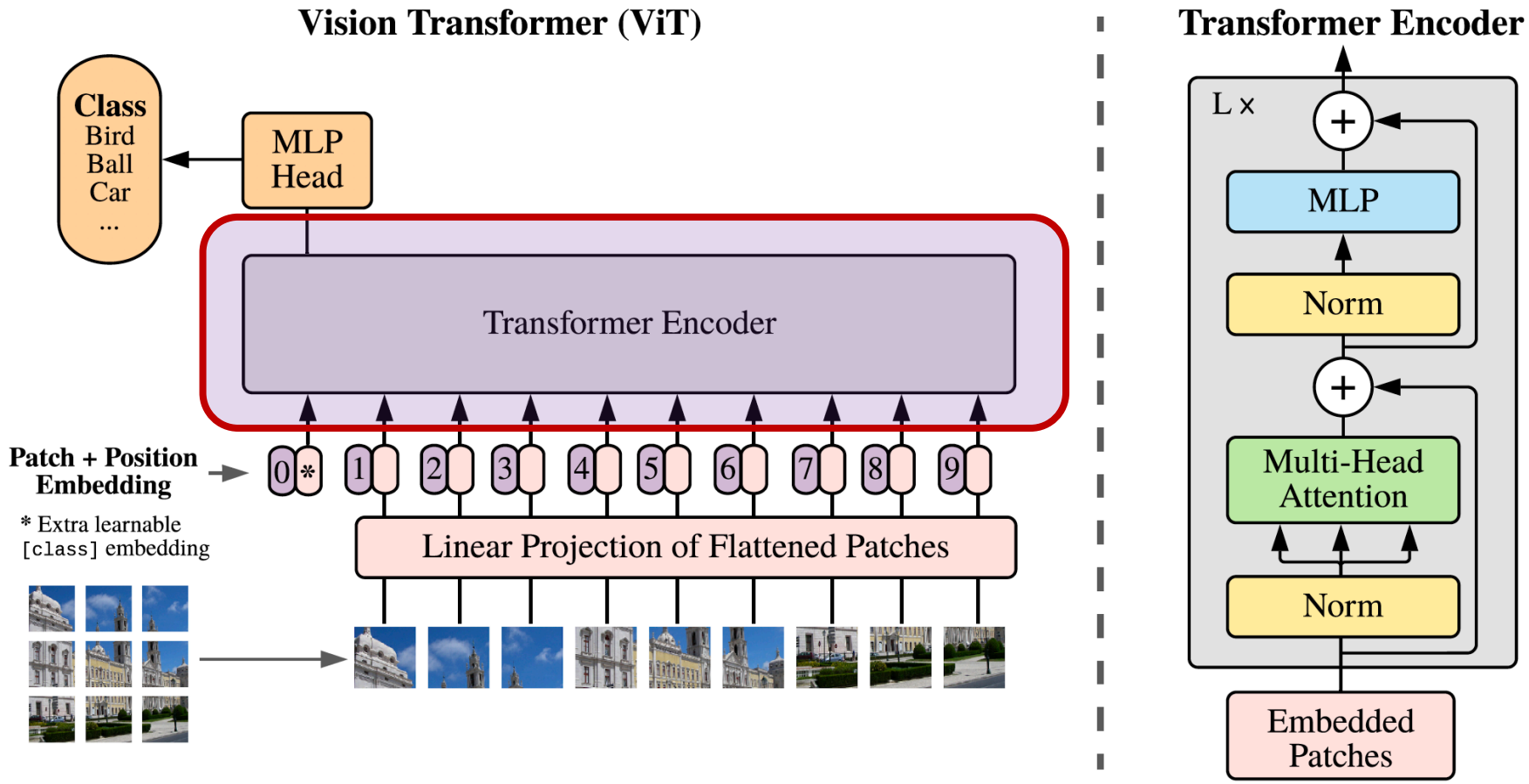
Position embeddings are added to the patch embeddings to retain positional information. We explore different 2D-aware variants of position embeddings (Appendix C.3) without any significant gains over standard 1D position embeddings. The joint embedding serves as input to the encoder.

```
self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
```

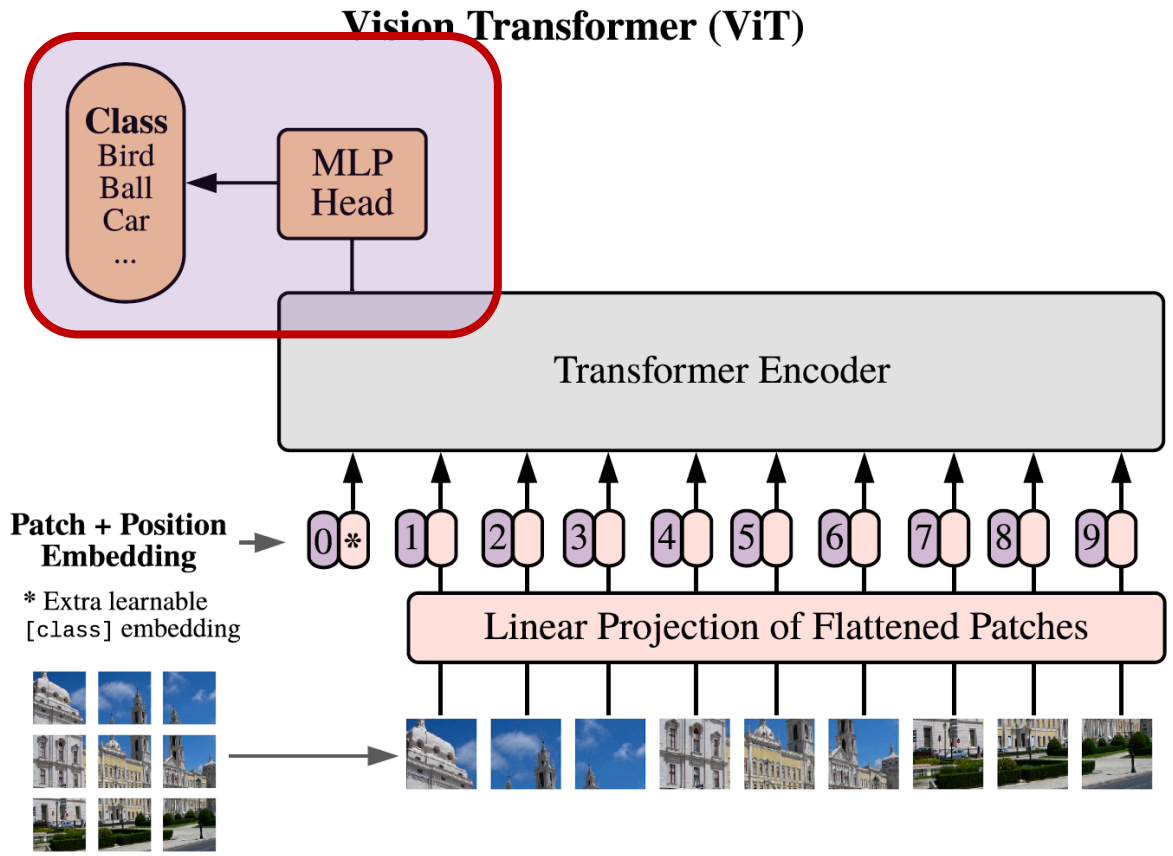


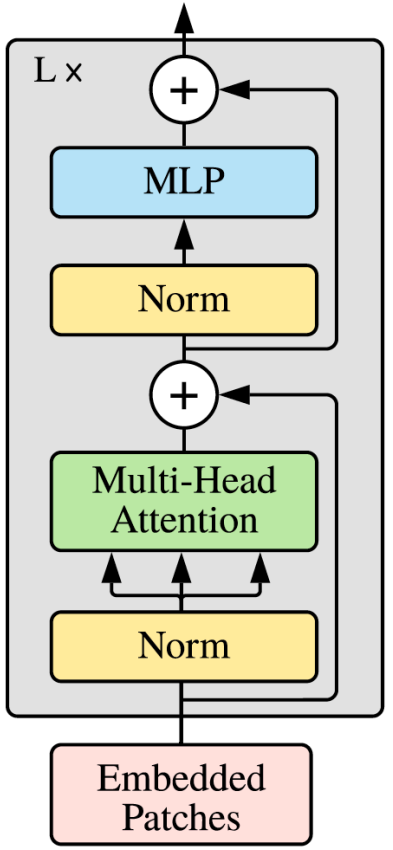
$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \mathbf{x}_p^2\mathbf{E};\ \cdots\ ;\ \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$$

```
x += self.pos_embedding
```

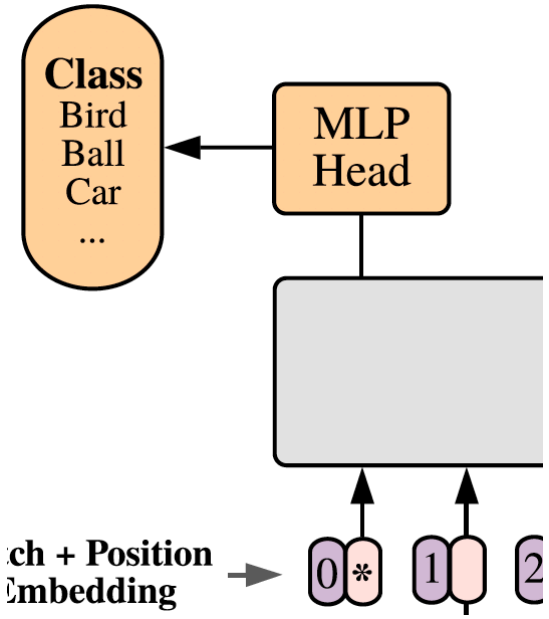



```
x = self.transformer(x, mask)
```






```
x = self.to_cls_token(x[:, 0])
return self.mlp_head(x)
```




```
self.mlp_head = nn.Sequential(
    nn.Linear(dim, mlp_dim),
    nn.GELU(),
    nn.Linear(mlp_dim, num_classes)
)
```

image representation $\mathbf{y}$ (Eq. 4). Both during pre-training and fine-tuning, the classification head is attached to $\mathbf{z}_L^0$.

```
x = self.to_cls_token(x[:, 0])
return self.mlp_head(x)
```

```
def forward(self, img, mask = None):
    p = self.patch_size

    x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = p, p2 = p)
    x = self.patch_to_embedding(x)

    cls_tokens = self.cls_token.expand(img.shape[0], -1, -1)
    x = torch.cat((cls_tokens, x), dim=1)
    x += self.pos_embedding
    x = self.transformer(x, mask)

    x = self.to_cls_token(x[:, 0])
    return self.mlp_head(x)
```

Transformer

$$\mathbf{z}'_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L$$

$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \qquad \ell = 1 \ldots L$$

$$\mathbf{y} = \mathrm{LN}(\mathbf{z}_L^0)$$

```
class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, mlp_dim):
        super().__init__()
        self.layers = nn.ModuleList([])
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                Residual(PreNorm(dim, Attention(dim, heads = heads))),
                Residual(PreNorm(dim, FeedForward(dim, mlp_dim)))
            ]))
    def forward(self, x, mask = None):
        for attn, ff in self.layers:
            x = attn(x, mask = mask)
            x = ff(x)
        return x
```

Residual

```
class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(x, **kwargs) + x
```

Layer Norm

```
class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)
```

Feed Forward (MLP)

```
class FeedForward(nn.Module):
    def __init__(self, dim, hidden_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.GELU(),
            nn.Linear(hidden_dim, dim)
        )
    def forward(self, x):
        return self.net(x)
```

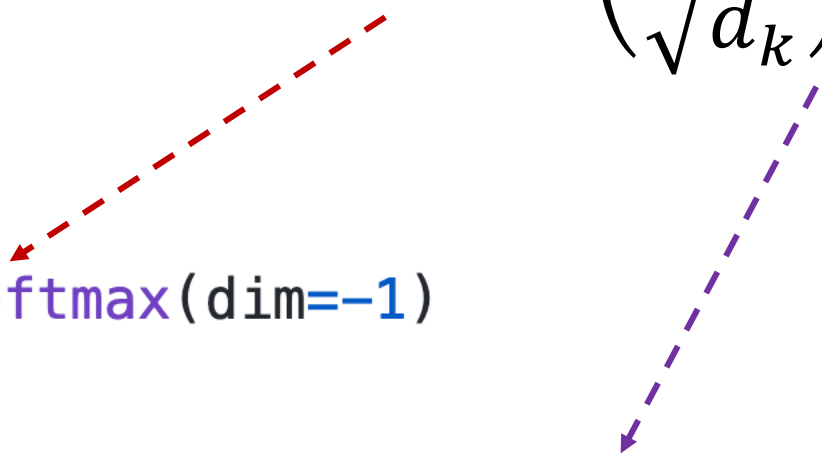
Attention

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

```
class Attention(nn.Module):
    def __init__(self, dim, heads = 8):
        super().__init__()
        self.heads = heads
        self.scale = dim ** -0.5

        self.to_qkv = nn.Linear(dim, dim * 3, bias = False)
        self.to_out = nn.Linear(dim, dim)
    def forward(self, x, mask = None):
        b, n, _, h = *x.shape, self.heads
        qkv = self.to_qkv(x)
        q, k, v = rearrange(qkv, 'b n (qkv h d) -> qkv b h n d', qkv = 3, h = h)

        dots = torch.einsum('bhid,bhjd->bhij', q, k) * self.scale
```

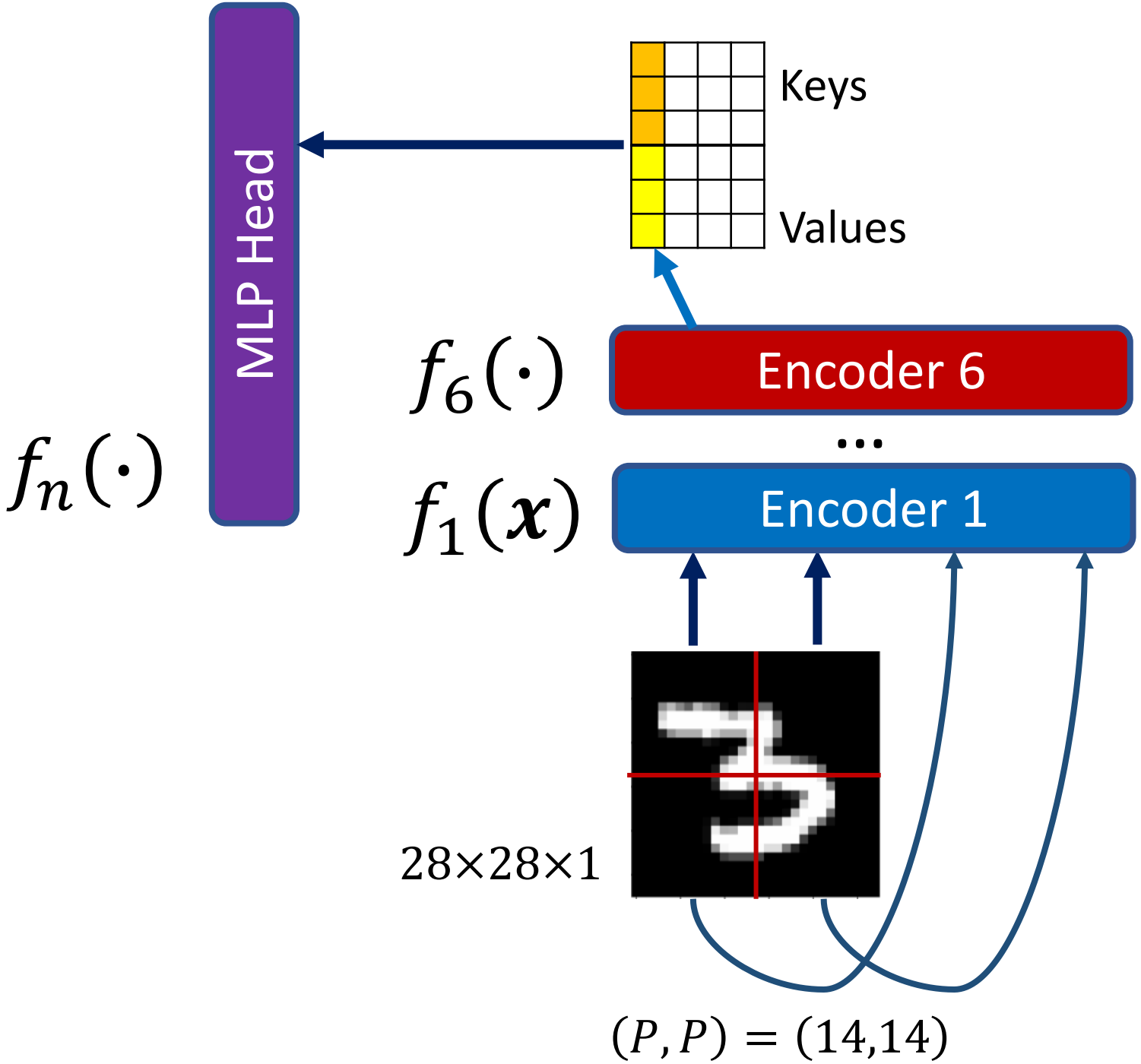
$$\frac{QK^T}{\sqrt{d_k}}$$

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



```
attn = dots.softmax(dim=-1)

out = torch.einsum('bhij,bhjd->bhid', attn, v)
out = rearrange(out, 'b h n d -> b n (h d)')
out =  self.to_out(out)
return out
```

Function composition



References

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems*. 2017.

Illustrated Transformer, http://jalammar.github.io/illustrated-transformer/

Transformers from Scratch, http://peterbloem.nl/blog/transformers

Transformer Family, https://lilianweng.github.io/lil-log/2020/04/07/the-transformer-family.html

In Summary

Transformer could be the most important breakthrough in the recent history of deep learning

Transformer has been used to produce state-of-the-art performances in NLP and vision

Expect more development in this field in the near future

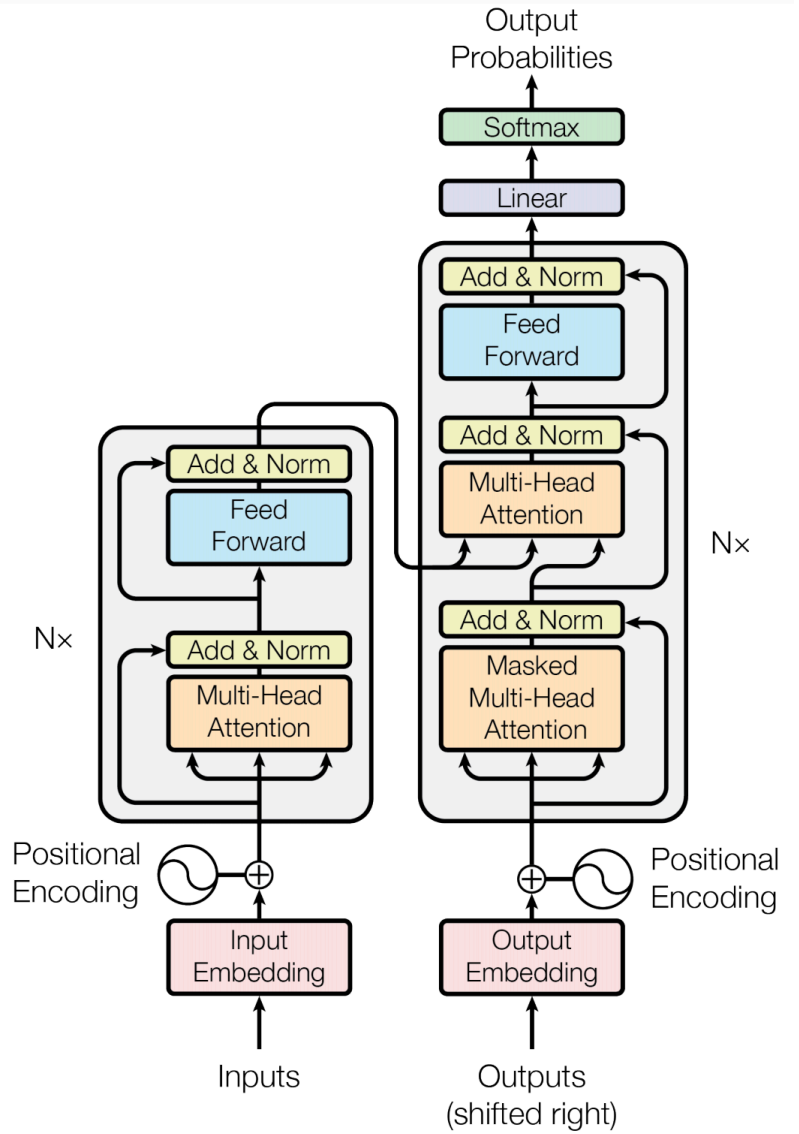


Figure 1: The Transformer - model architecture.

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems*. 2017.