AI VIETNAM
All-in-One Course

Extra Class

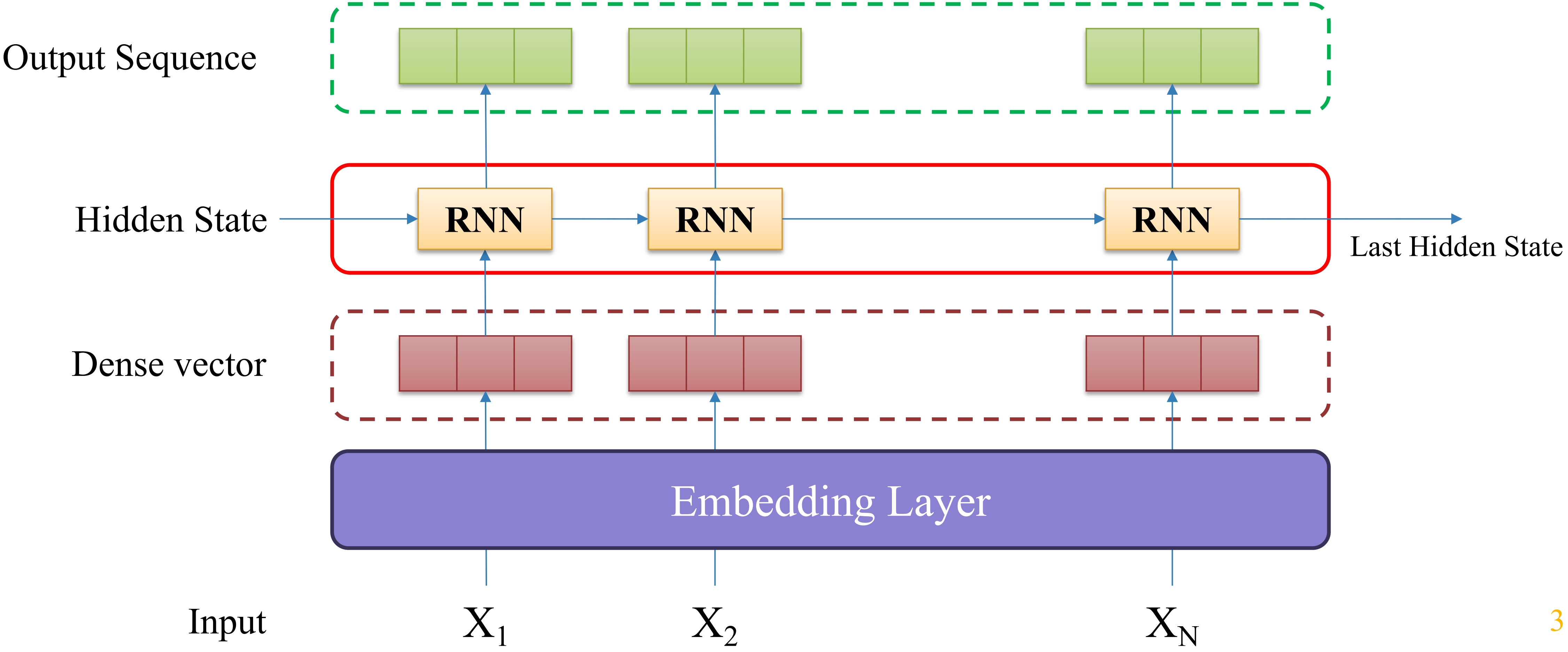
Introduction to Transformer

Nguyen Quoc Thai

CONTENT

1 – Attention

RNNs Model

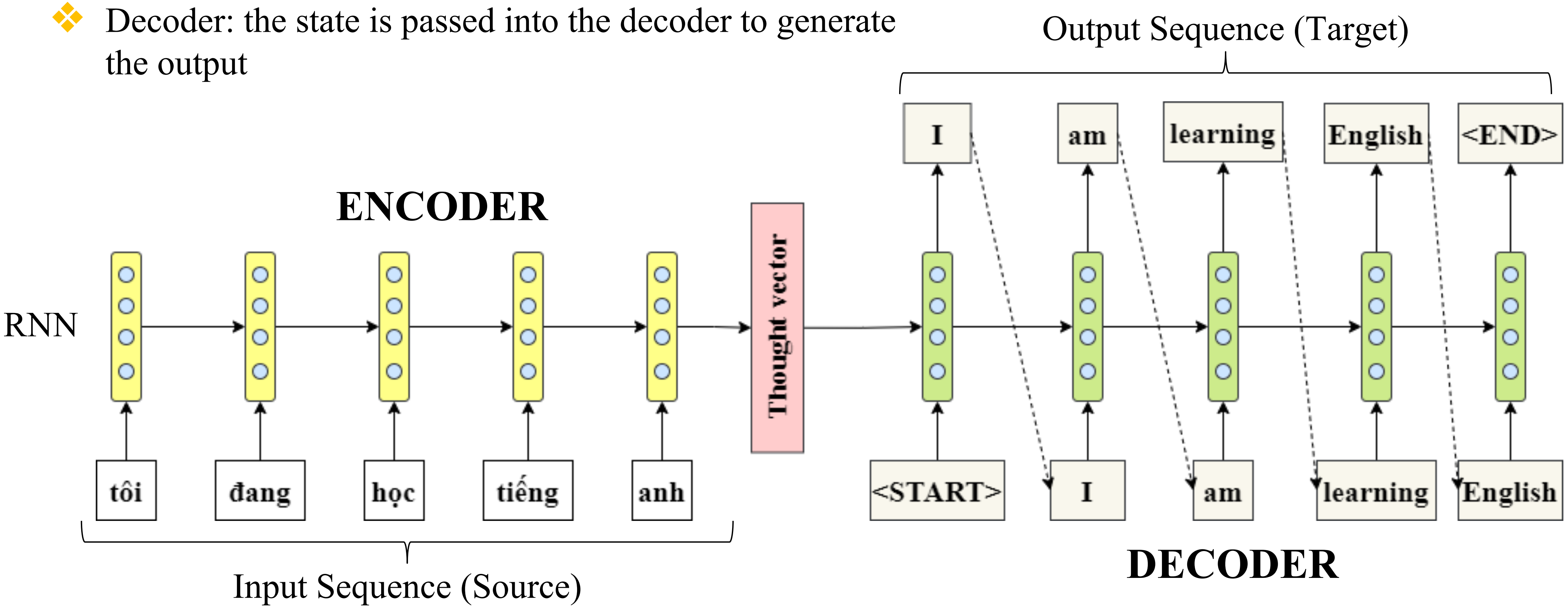


AI VIET NAM
@aivietnam.edu.vn

1 – Attention

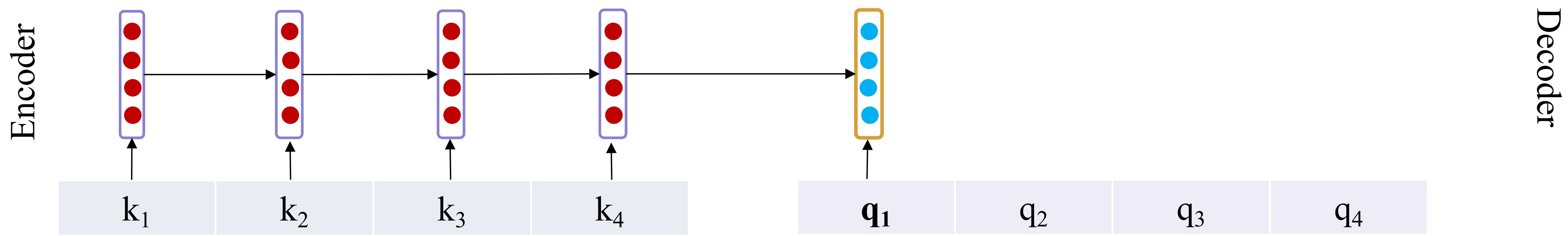
Sequence-to-Sequence Architecture

- Encoder: encoding the inputs into state (thought vector)
- Decoder: the state is passed into the decoder to generate the output



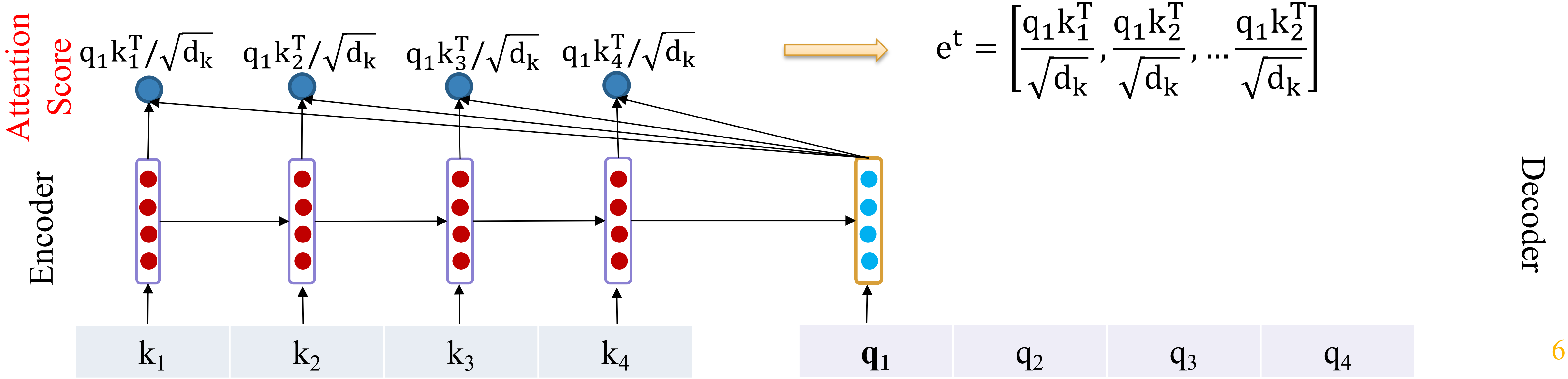
1 – Attention

Scaled Dot-Product Attention



1 – Attention

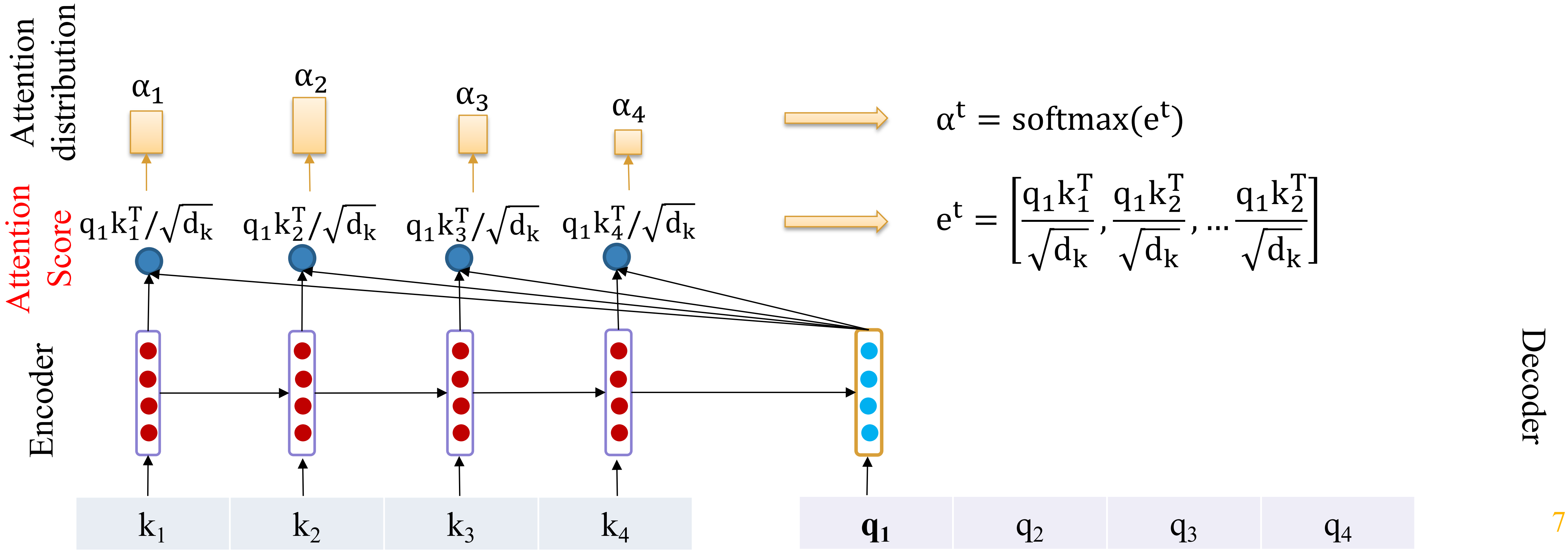
Scaled Dot-Product Attention



$$e^t = \left[\frac{q_1 k_1^T}{\sqrt{d_k}}, \frac{q_1 k_2^T}{\sqrt{d_k}}, \ldots \frac{q_1 k_2^T}{\sqrt{d_k}}\right]$$

1 – Attention

Scaled Dot-Product Attention

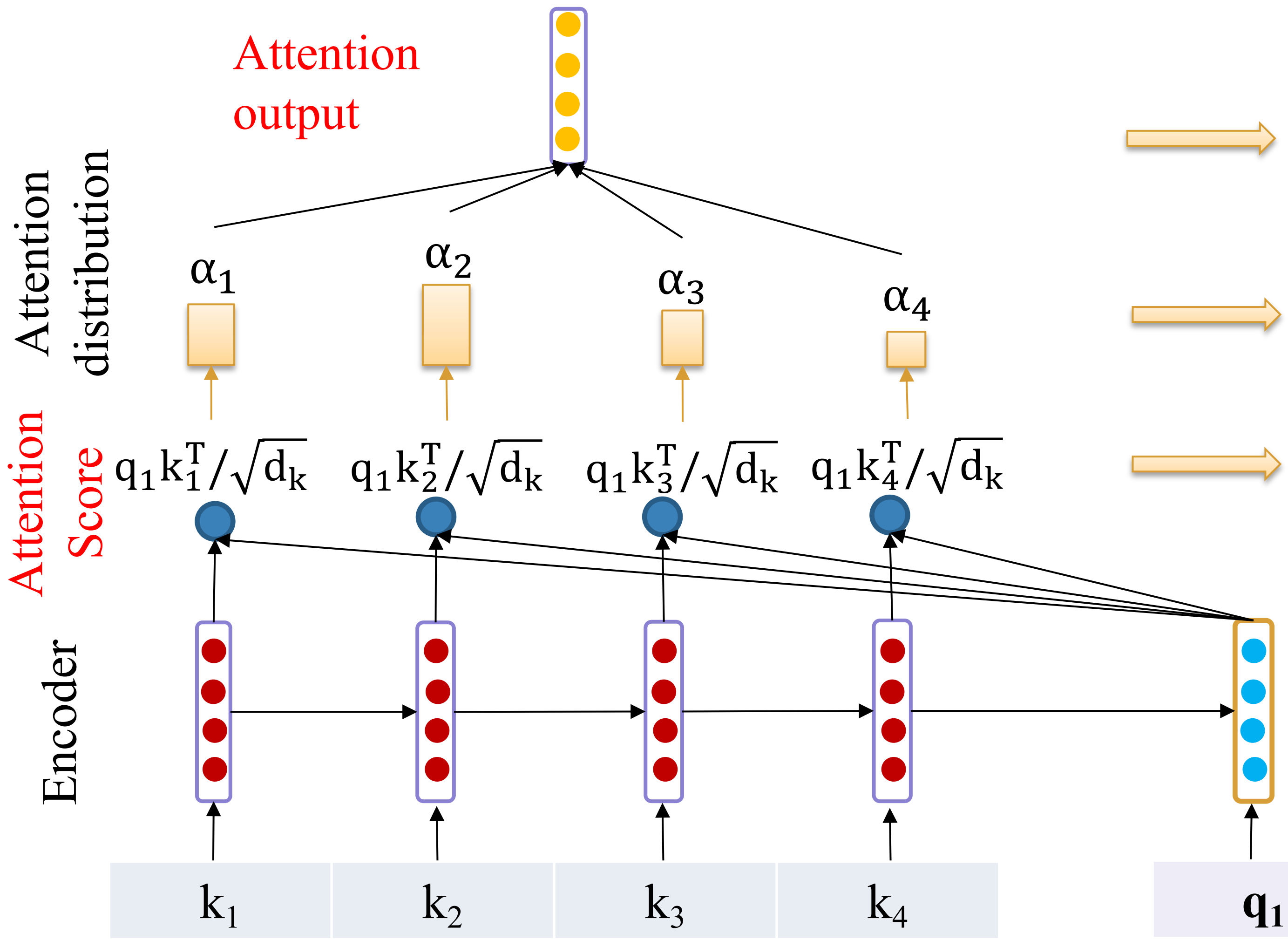


$$\alpha^t = \mathrm{softmax}(e^t)$$

$$e^t = \left[\frac{q_1k_1^T}{\sqrt{d_k}}, \frac{q_1k_2^T}{\sqrt{d_k}}, \dots \frac{q_1k_2^T}{\sqrt{d_k}}\right]$$

1 – Attention

Scaled Dot-Product Attention



$$a^t = \sum_{1}^{N} \alpha_i^t v_i \,;\; v_i = k_i$$

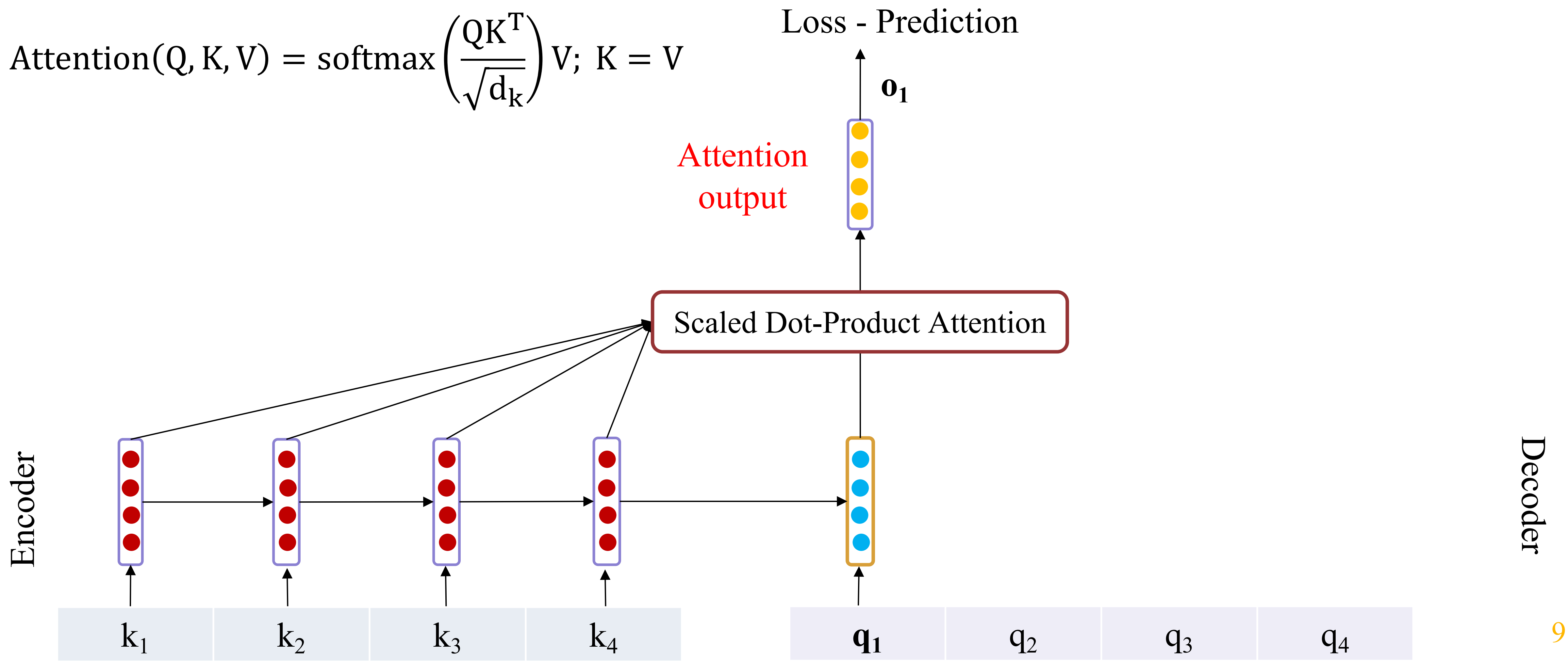
$$\alpha^t = \mathrm{softmax}(e^t)$$

$$e^t = \left[\frac{q_1k_1^T}{\sqrt{d_k}}, \frac{q_1k_2^T}{\sqrt{d_k}}, \ldots \frac{q_1k_2^T}{\sqrt{d_k}}\right]$$

1 – Attention

Scaled Dot-Product Attention

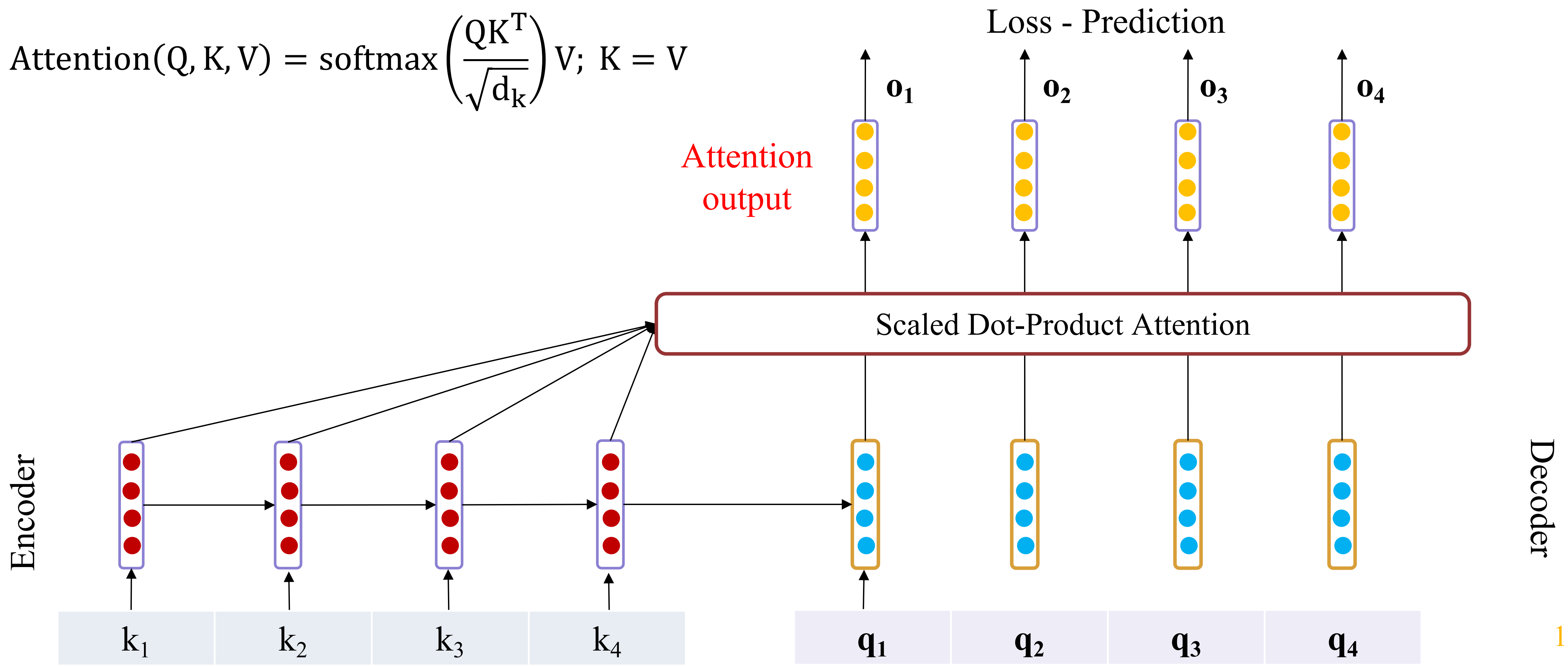
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V;\ K = V$$



1 – Attention

Scaled Dot-Product Attention

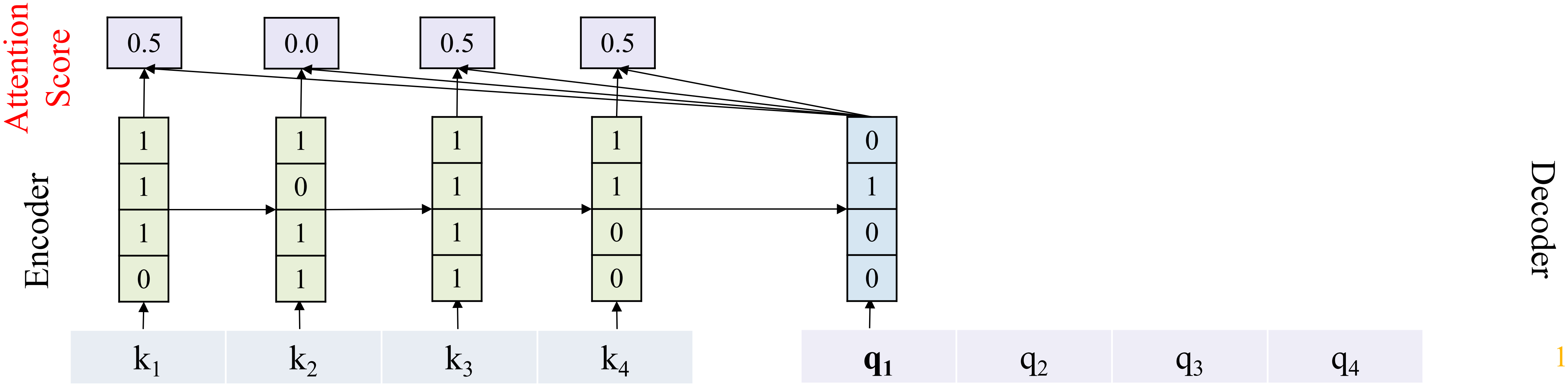
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V; \; K = V$$



1 – Attention

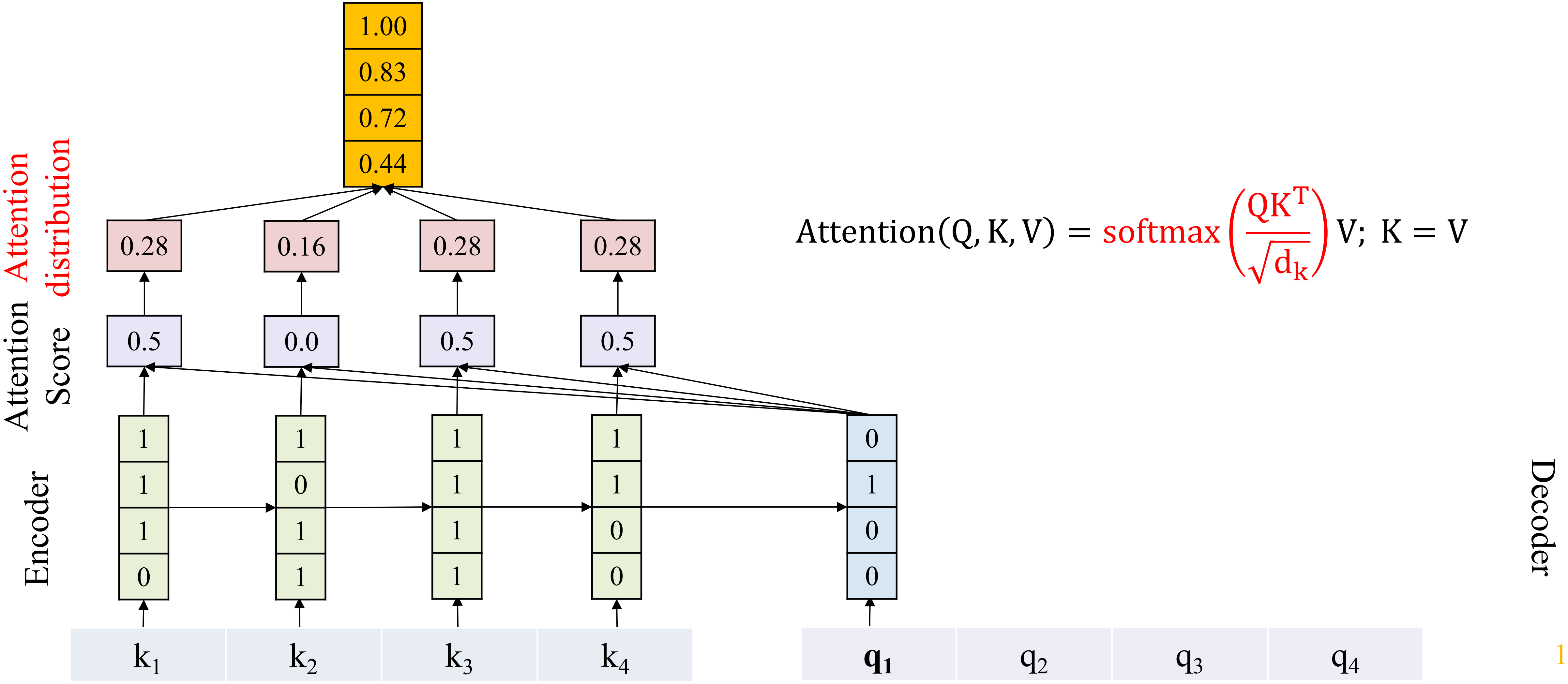
Scaled Dot-Product Attention - Example

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



1 – Attention

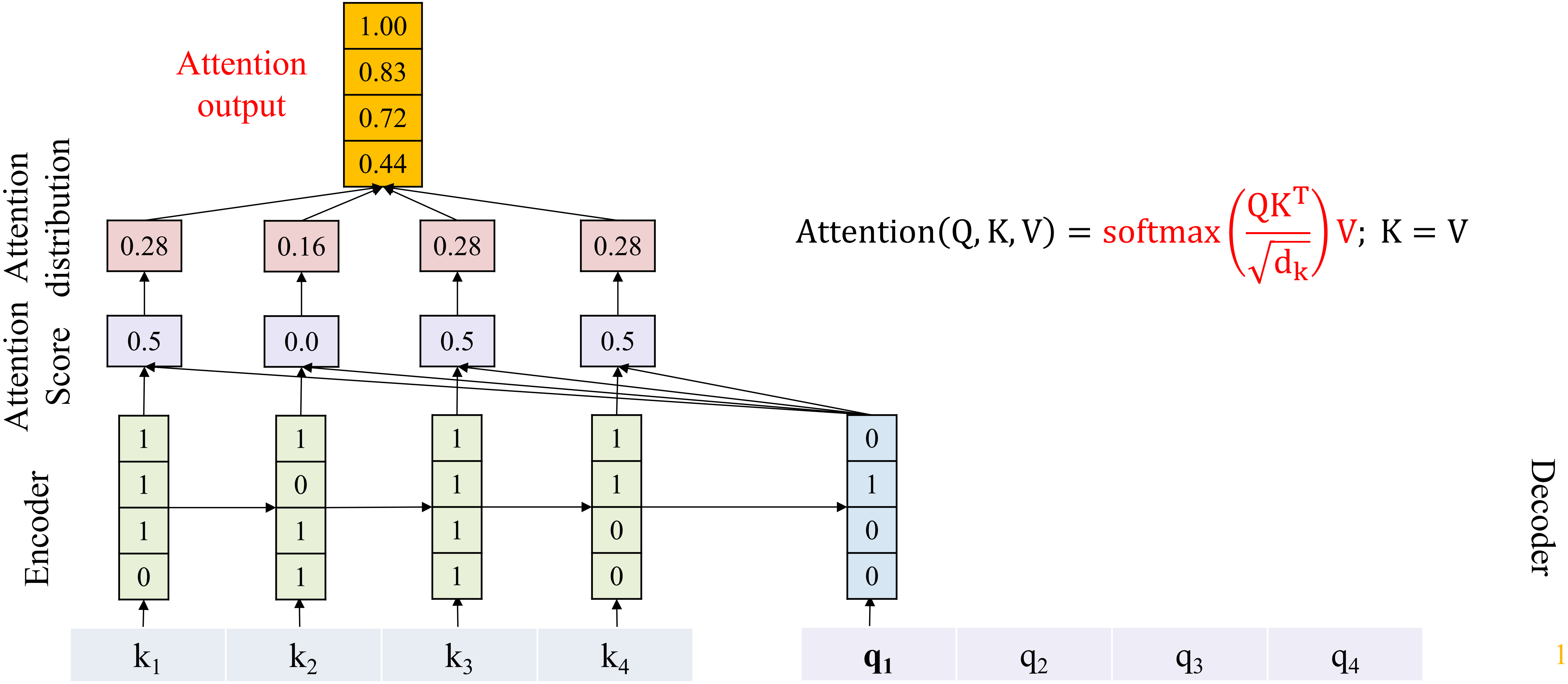
Scaled Dot-Product Attention - Example



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

1 – Attention

Scaled Dot-Product Attention - Example

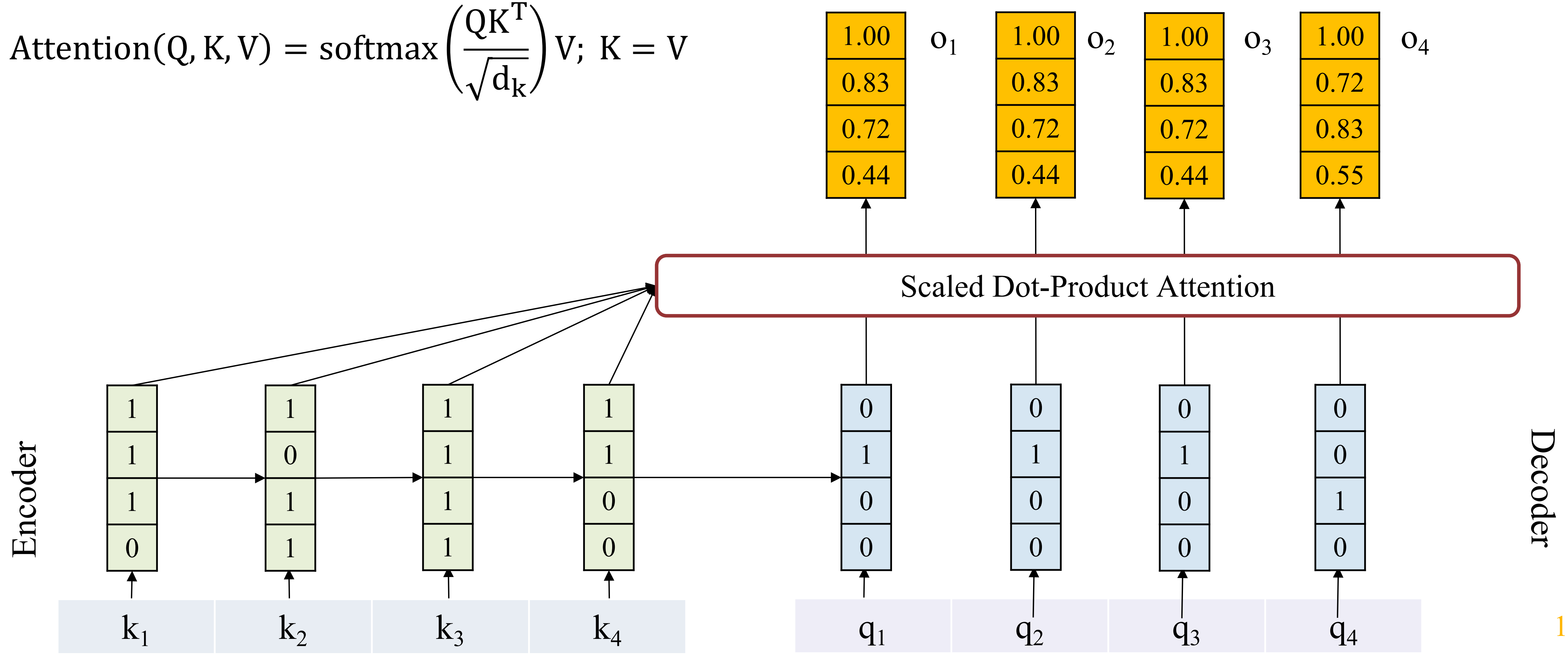


$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V; \; K = V$$

1 – Attention

Scaled Dot-Product Attention - Example

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



1 – Attention

Scaled Dot-Product Attention - Demo

```
query = torch.randint(
    high=2,
    size=(1, 4, 4), # batch_size x seq_len x embedding_dim
    dtype=torch.float32
)
query
```

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

```
key = torch.randint(
    high=2,
    size=(1, 4, 4),
    dtype=torch.float32
)
key
```

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

```
value = key
value
```

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

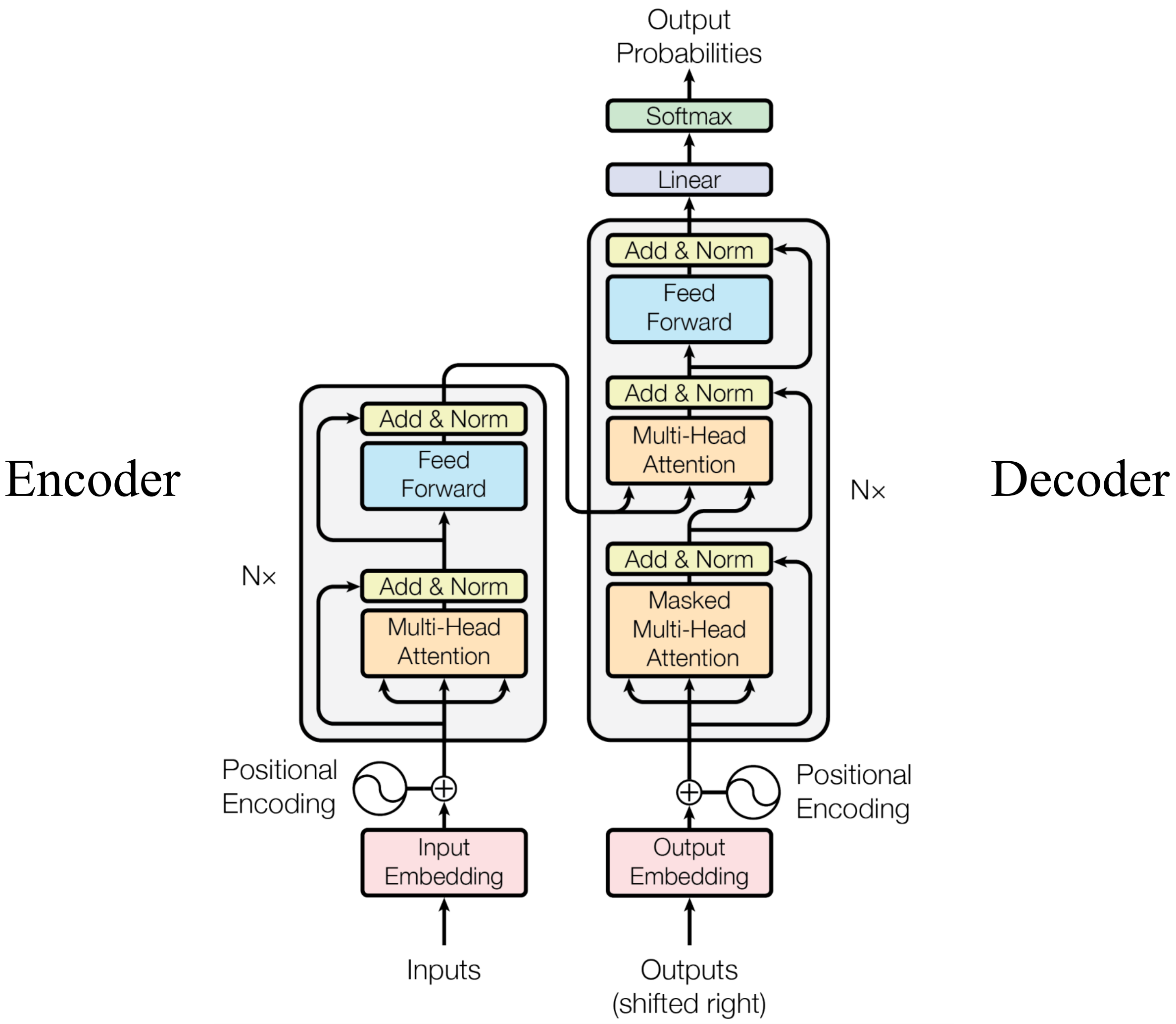
```
attentionn_weight = F.scaled_dot_product_attention(
    query=query,
    key=key,
    value=value
)
attentionn_weight
```

```
tensor([[[1.0000, 0.8318, 0.7227, 0.4455],
         [1.0000, 0.8318, 0.7227, 0.4455],
         [1.0000, 0.8318, 0.7227, 0.4455],
         [1.0000, 0.7227, 0.8318, 0.5545]]])
```

2 – Transformer-Encoder

Transformer

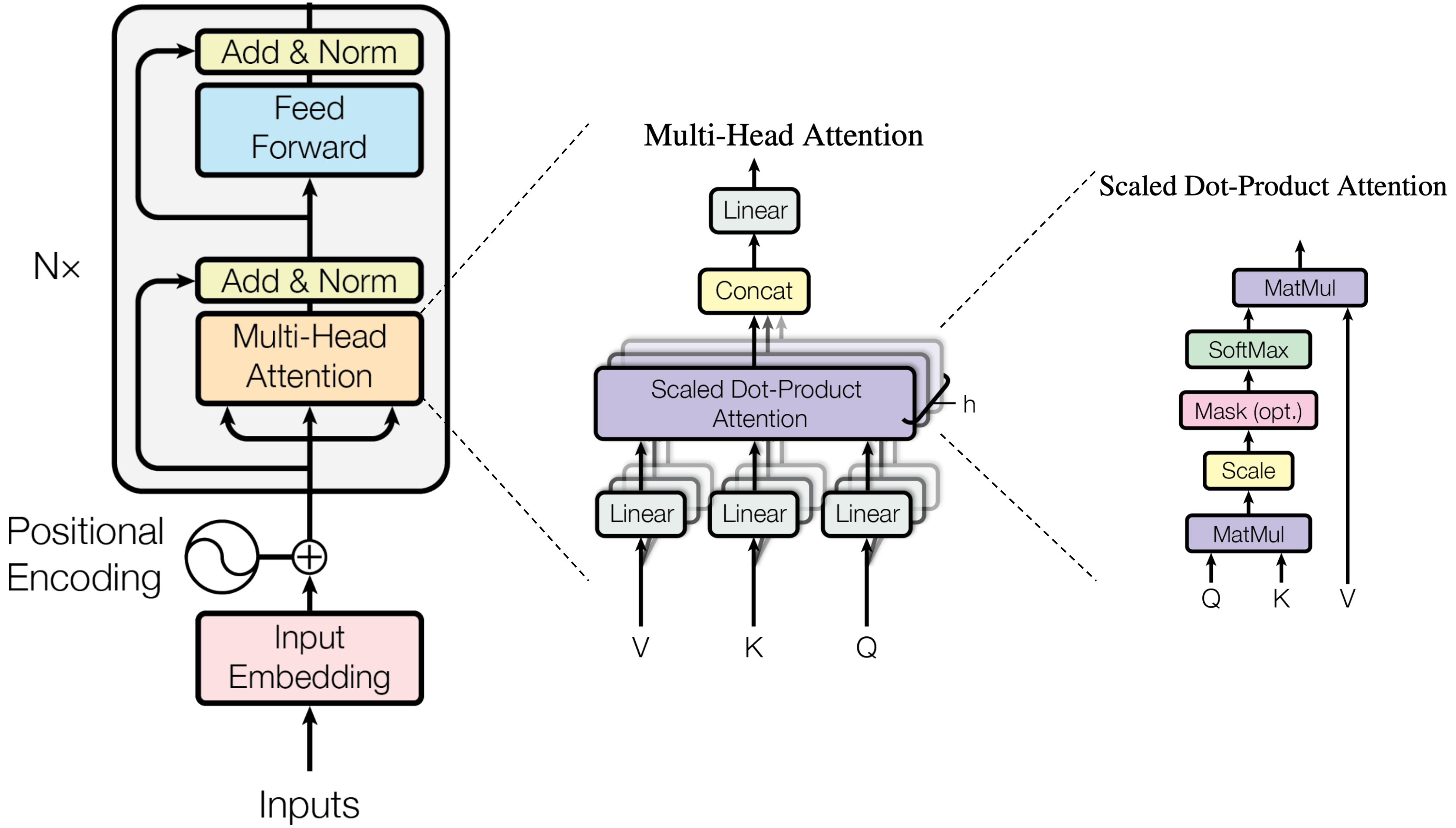
- Architecture:
  - N Encoder Layer
  - N Decoder Layer
- Core technique: attention
- Loss function: cross-entropy



2 – Transformer-Encoder

Transformer-Encoder

- Input Embedding
- Positional Encoding
- Multi-Head Attention
- Feed Forward
- Add & Norm

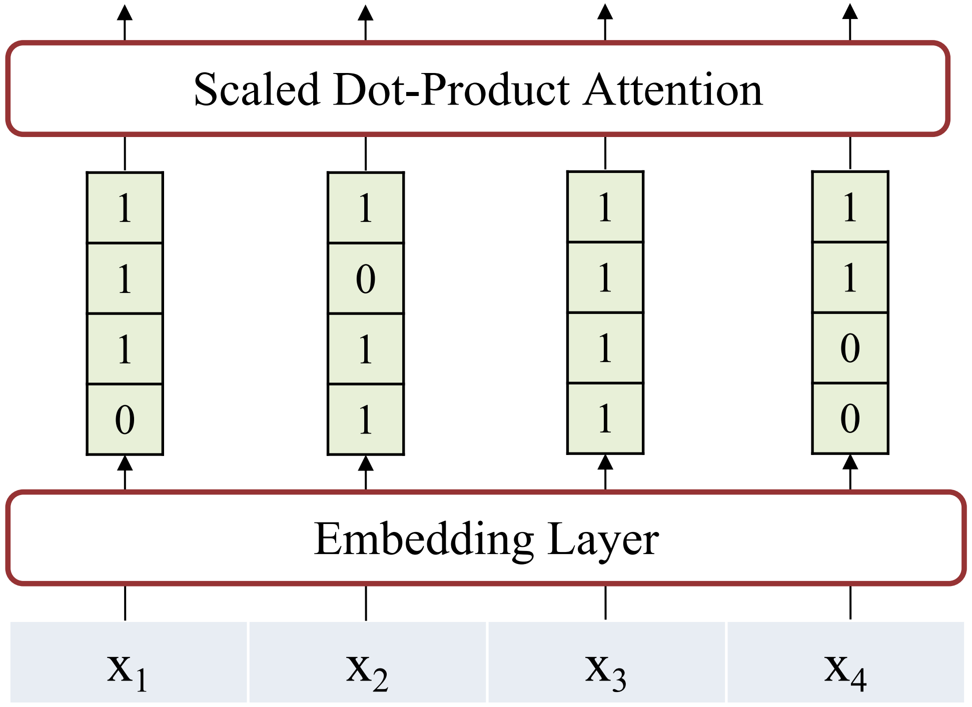


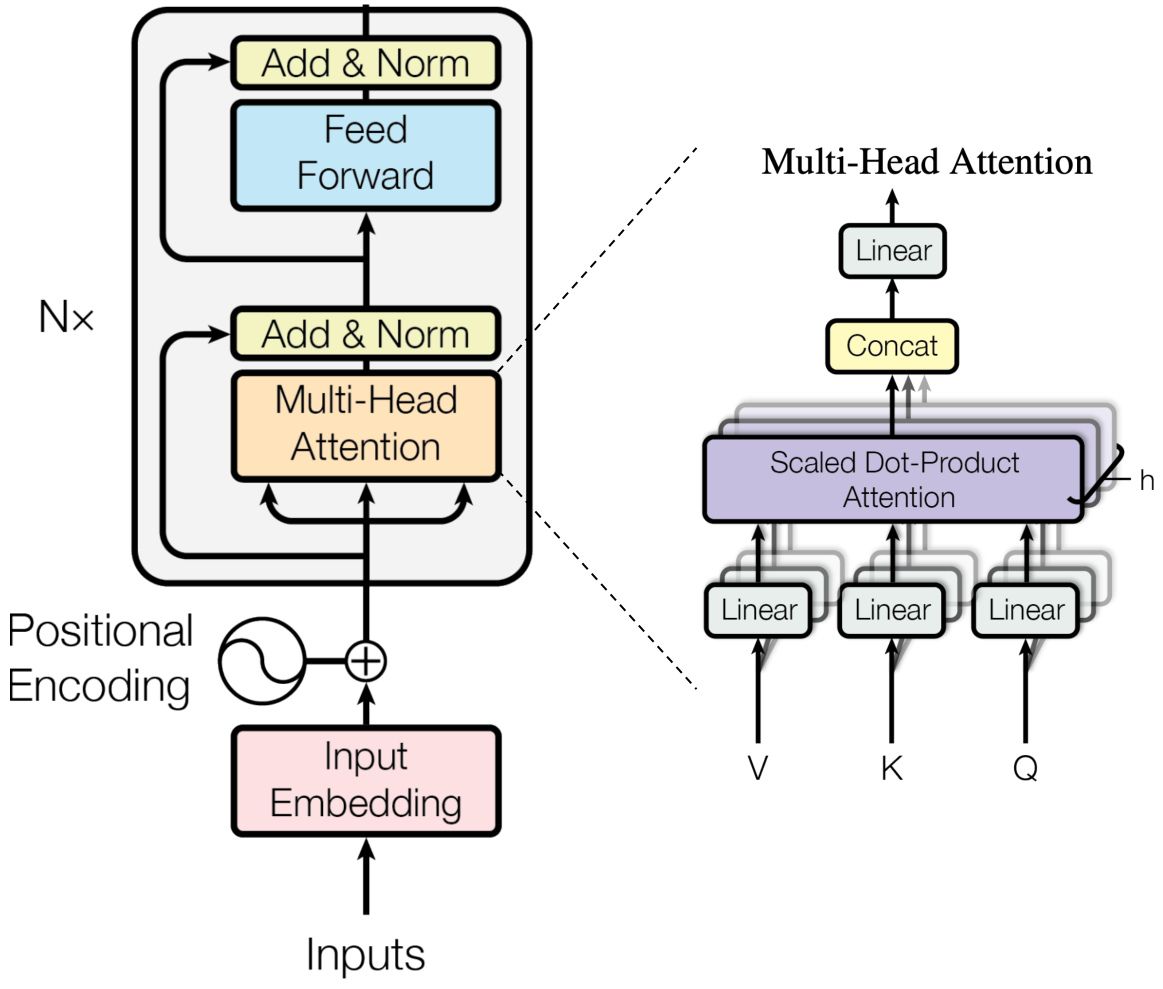
2 – Transformer-Encoder

Input Embedding

❖ Input Embedding: Embedding Layer

QUERY – KEY – VALUE ?

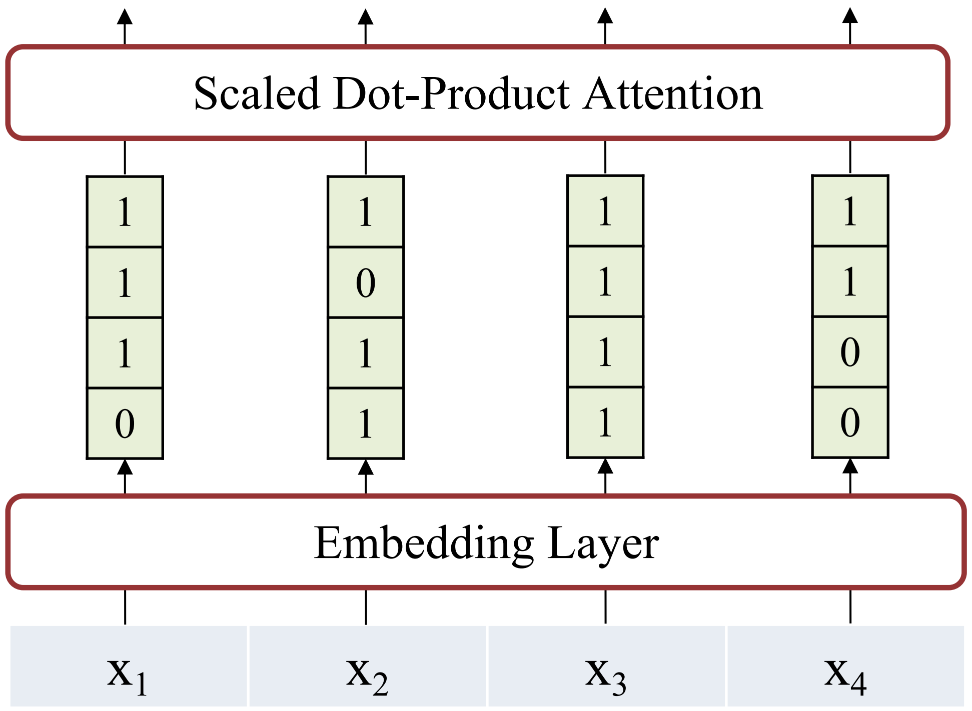


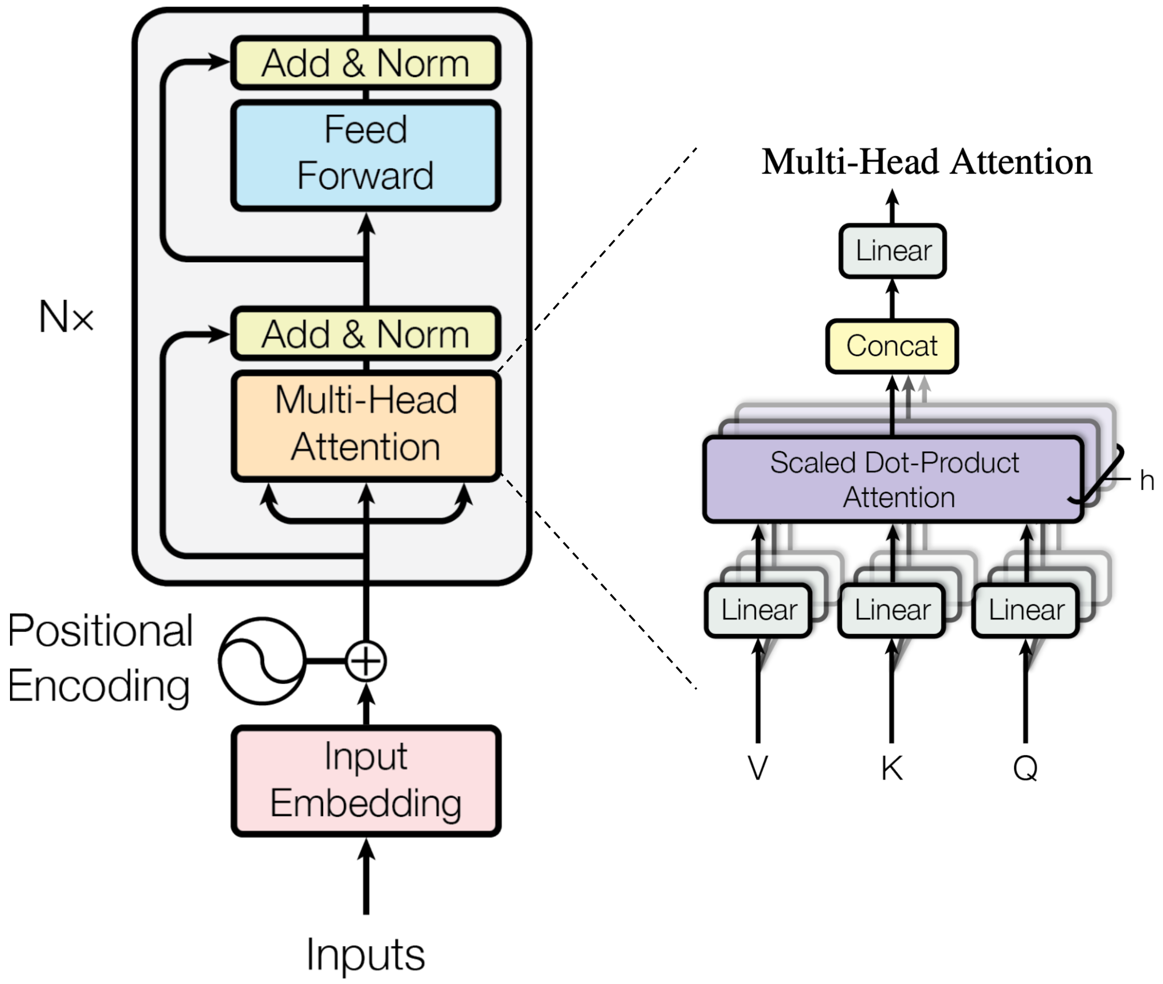


2 – Transformer-Encoder

Self-Attention

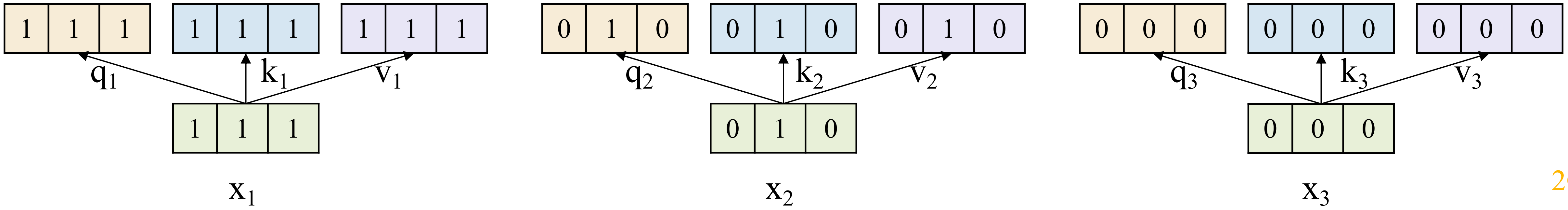
QUERY = KEY = VALUE = EMBEDDED





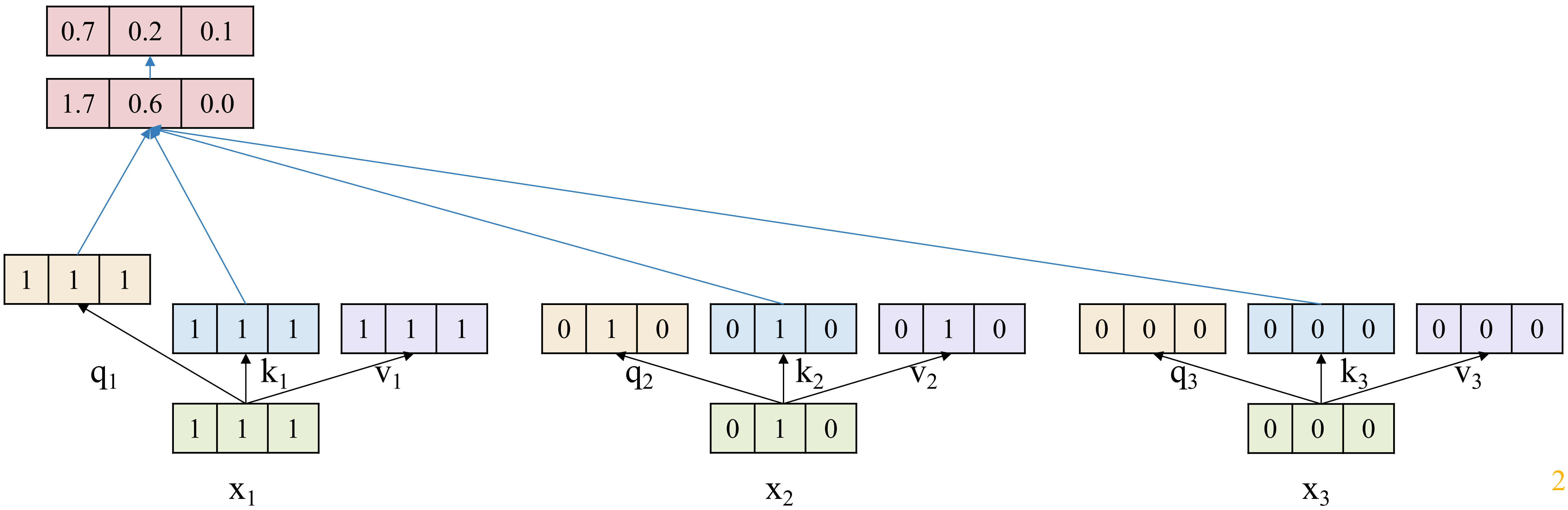
2 – Transformer-Encoder

Self-Attention



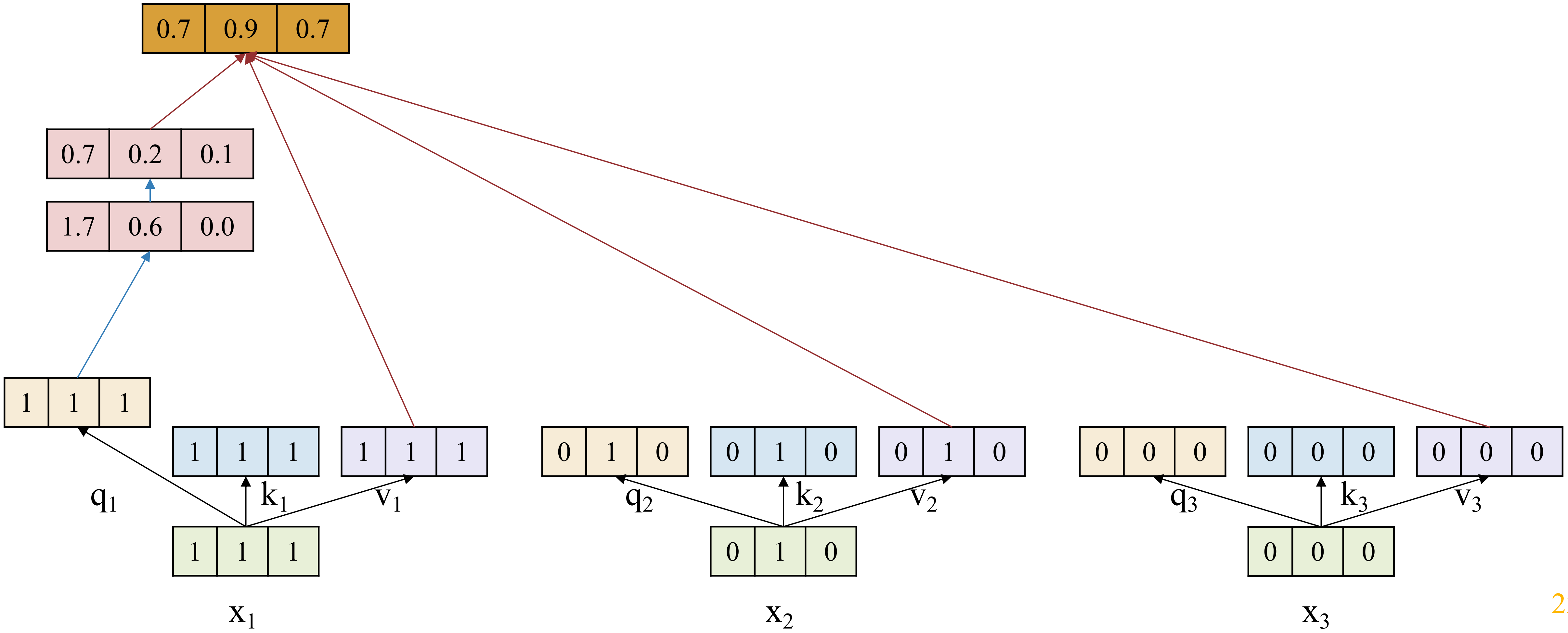
2 – Transformer-Encoder

Self-Attention



2 – Transformer-Encoder

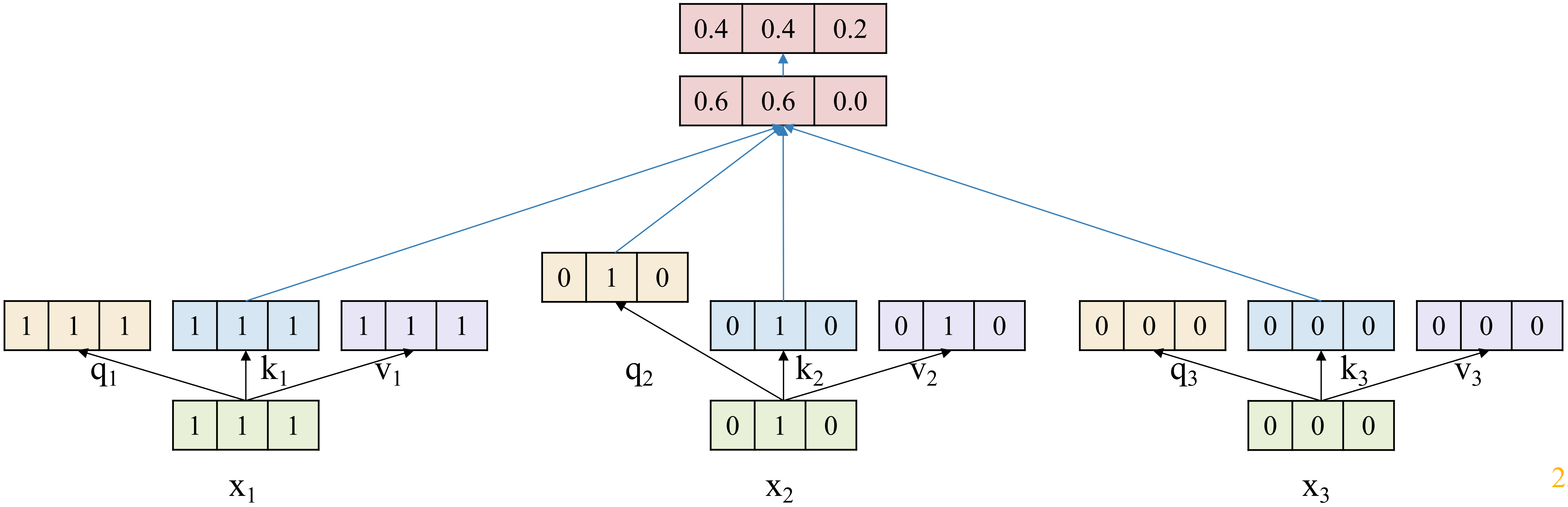
Self-Attention



2 – Transformer-Encoder

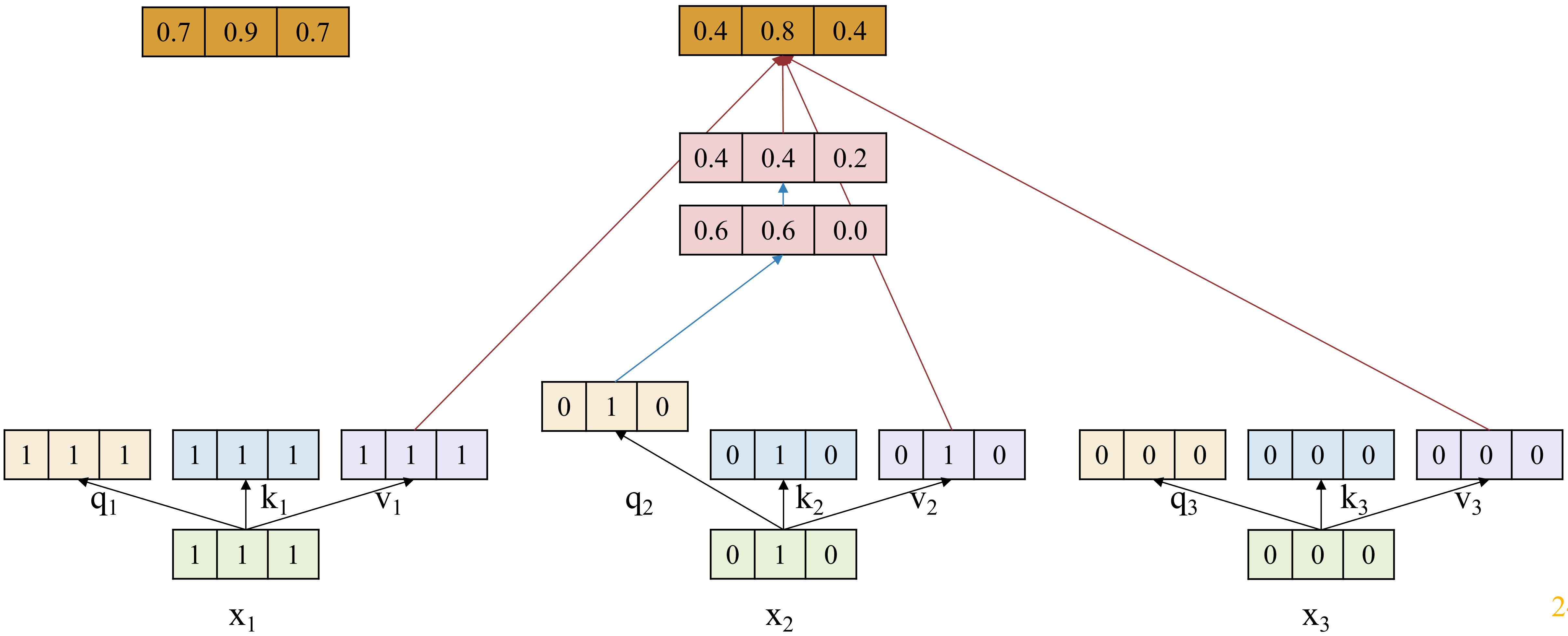
Self-Attention

0.7	0.9	0.7



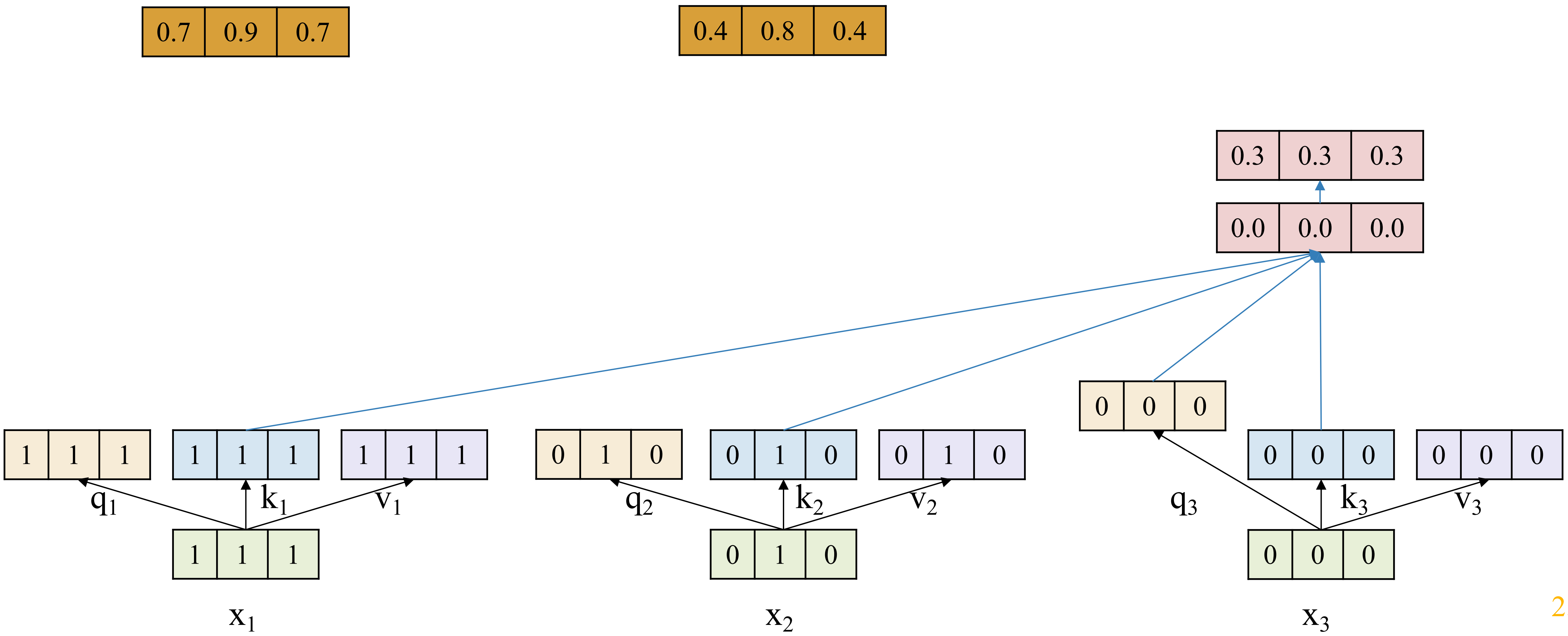
2 – Transformer-Encoder

Self-Attention



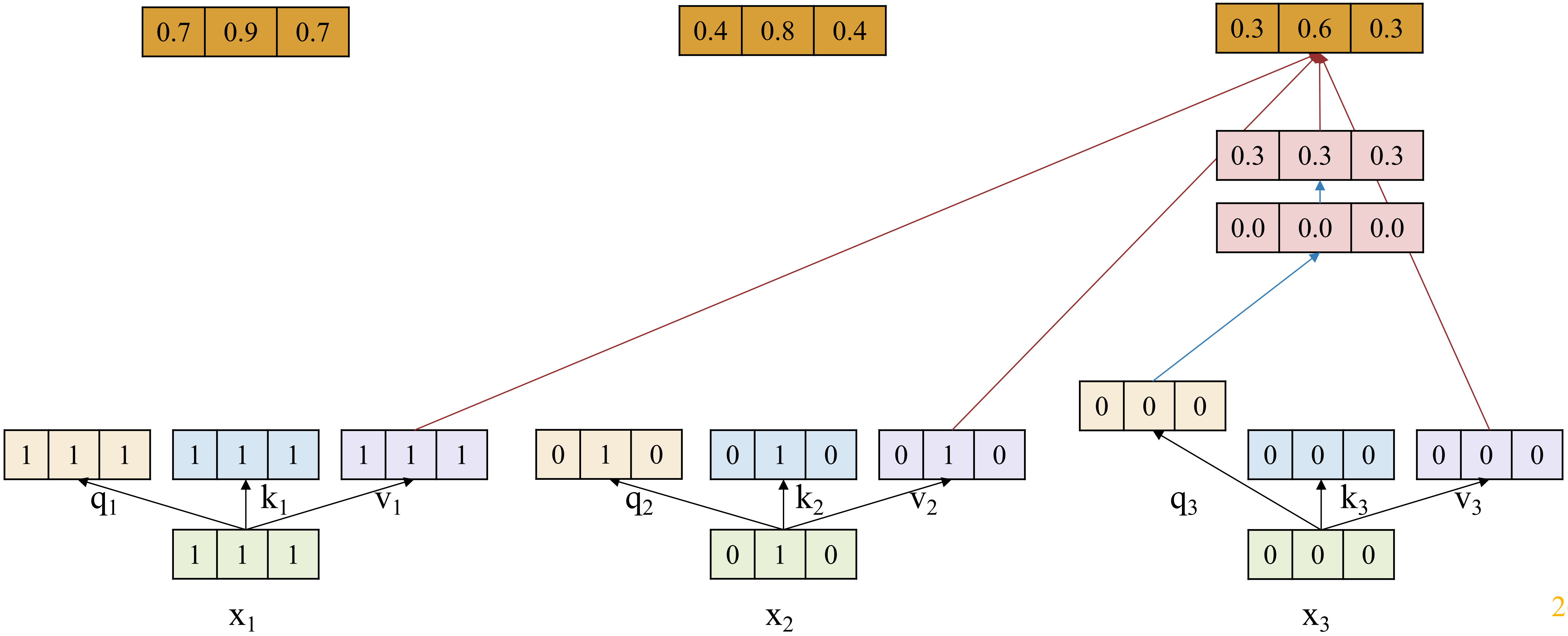
2 – Transformer-Encoder

Self-Attention



2 – Transformer-Encoder

Self-Attention

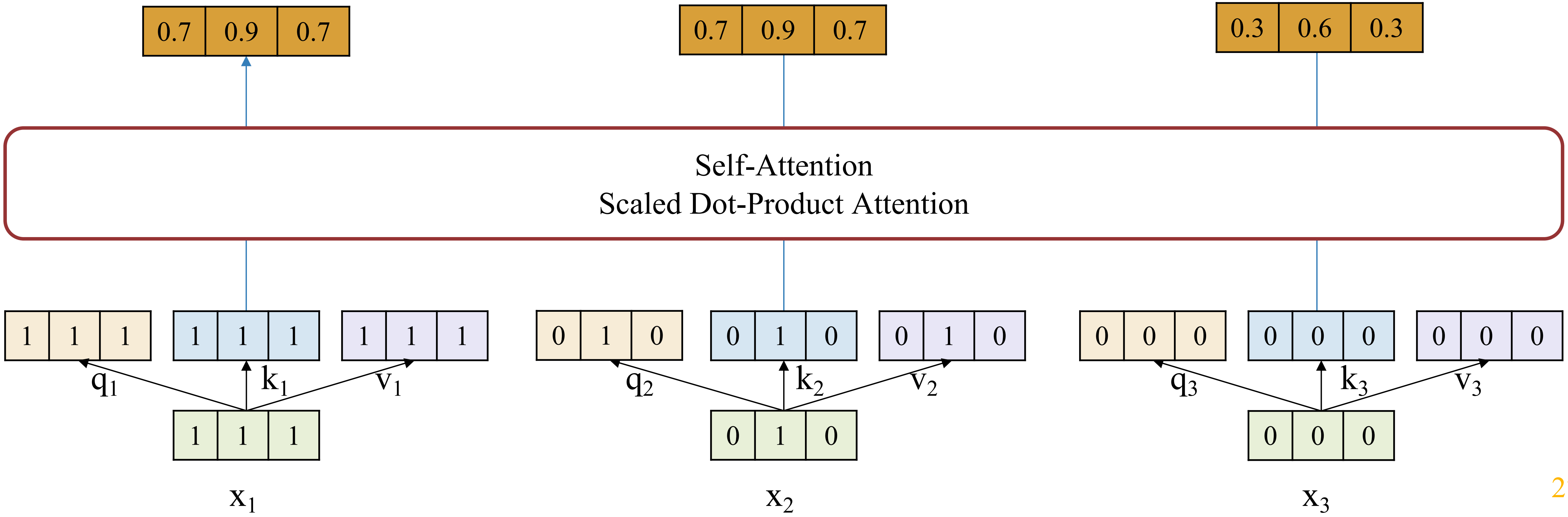


2 – Transformer-Encoder

Self-Attention

- To learn the relationship between word in the sentence

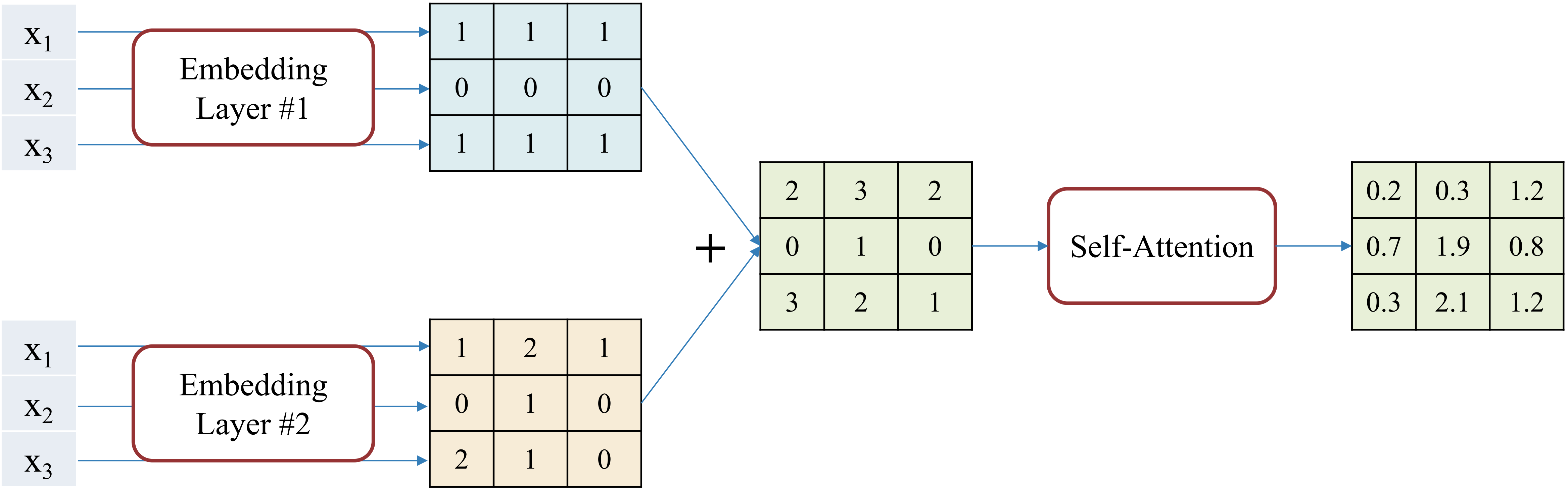
Ignore the order of words in the sentence ?



2 – Transformer-Encoder

Positional Encoding

- The position of a token in a sentence as unique representation – each position is mapped to a vector
- Methods: Sinusoid; Learned positional embedding (as learned input embedding)



2 – Transformer-Encoder

Positional Encoding – Demo

```python
class TokenAndPositionEmbedding(nn.Module):
    def __init__(self, vocab_size, embed_dim, max_length, device='cpu'):
        super().__init__()
        self.device = device
        self.word_emb = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embed_dim
        )
        self.pos_emb = nn.Embedding(
            num_embeddings=max_length,
            embedding_dim=embed_dim
        )

    def forward(self, x):
        N, seq_len = x.size()
        positions = torch.arange(0, seq_len).expand(N, seq_len).to(self.device)
        output1 = self.word_emb(x)
        output2 = self.pos_emb(positions)
        output = output1 + output2
        return output
```

```python
vocab_size = 10000
embed_dim = 200
max_length = 50
embedding = TokenAndPositionEmbedding(
    vocab_size,
    embed_dim,
    max_length
)
```

```python
batch_size = 32

input = torch.randint(
    high=2,
    size=(batch_size, max_length),
    dtype=torch.int64
)
```

```python
embedded = embedding(input)
```

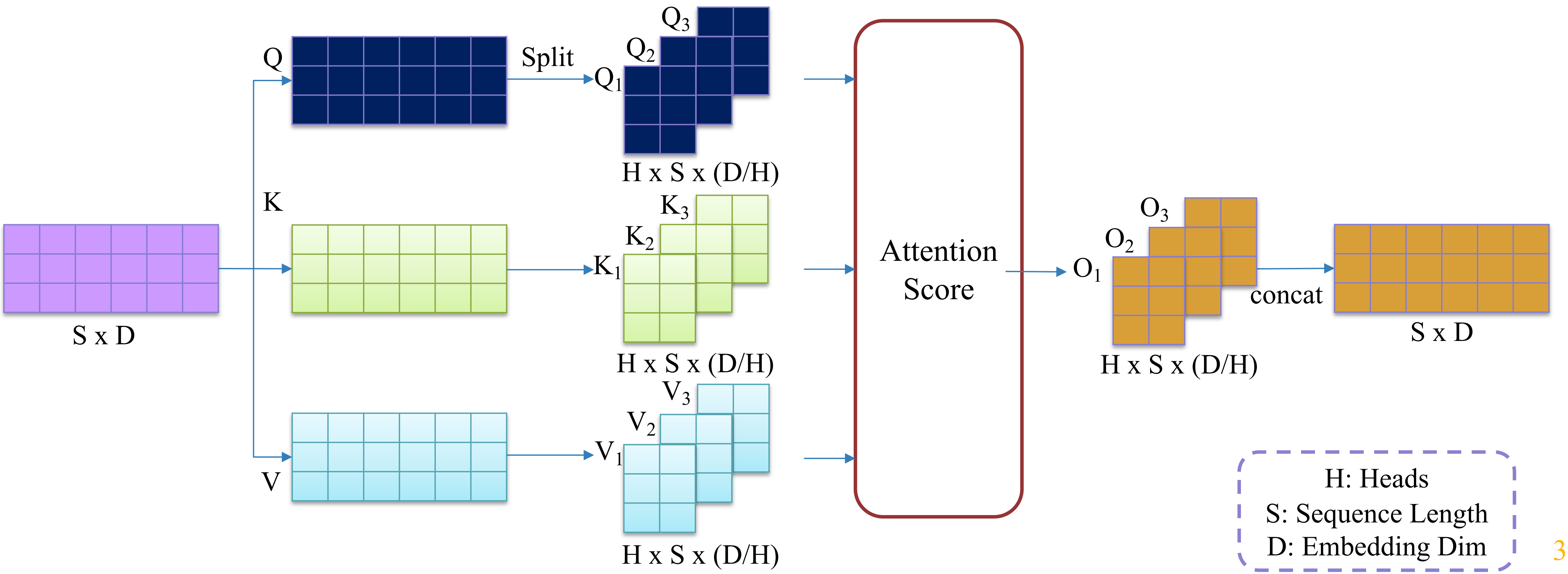
```python
embedded.shape
```

```
torch.Size([32, 50, 200])
```

2 – Transformer-Encoder

Multi-Head Attention

- Split into the multiple attention heads (process independently) => self-attention => concat



2 – Transformer-Encoder

Multi-Head Attention – Demo

```
batch_size = 1
seq_len = 50
embedding_dim = 200

input = torch.randint(
    high=2,
    size=(batch_size, seq_len, embedding_dim),
    dtype=torch.float32
)
input
```

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

```
embedding_dim = 200
num_heads = 5

att_layer = nn.MultiheadAttention(
    embed_dim=embedding_dim,
    num_heads=num_heads,
    batch_first=True
)
```

```
attn_output, attn_output_weights = att_layer(
    query=input,
    key=input,
    value=input
)
```

```
attn_output.shape
```

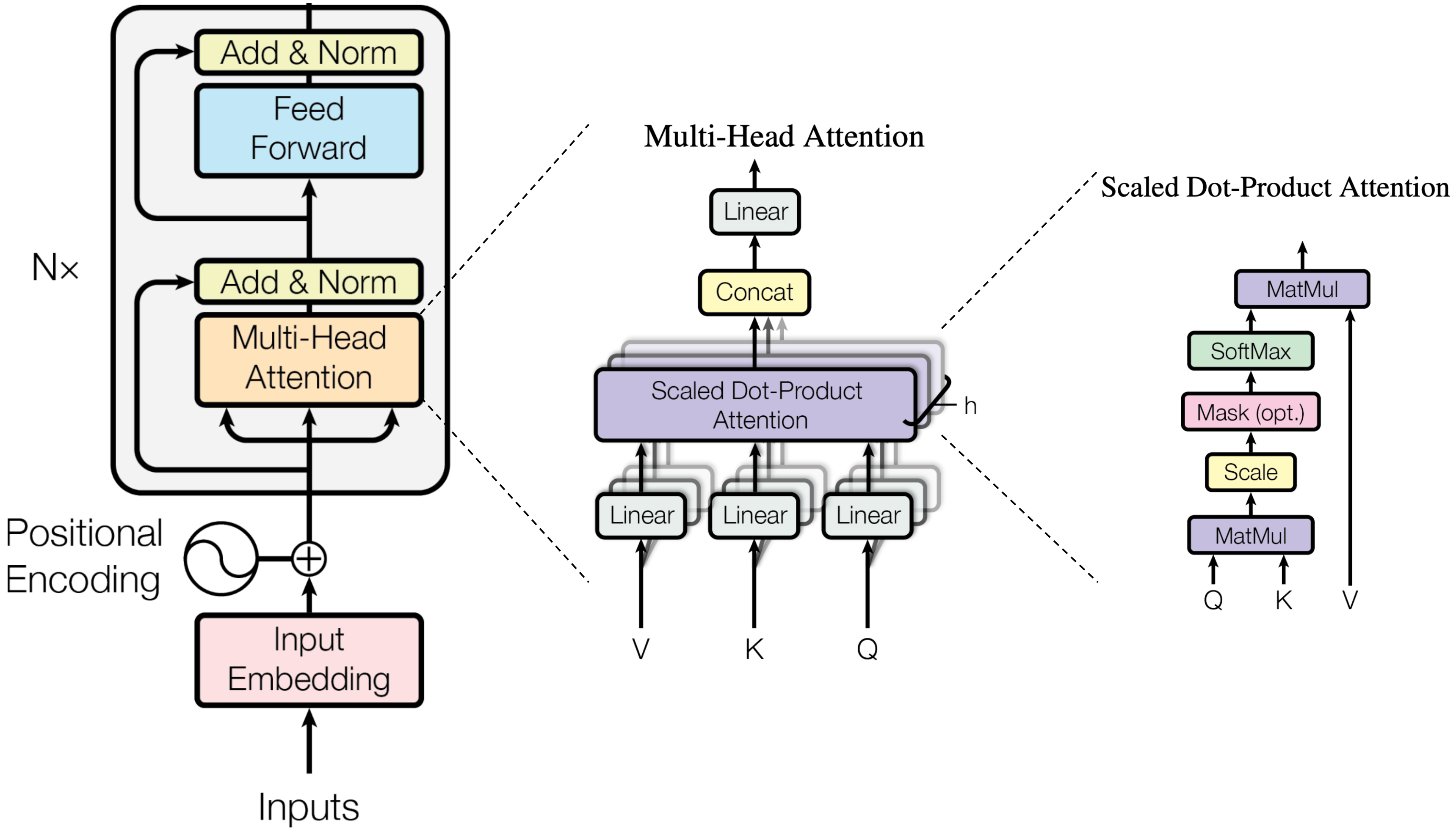
```
torch.Size([1, 50, 200])
```

```
attn_output_weights.shape
```

```
torch.Size([1, 50, 50])
```

2 – Transformer-Encoder

Transformer-Encoder



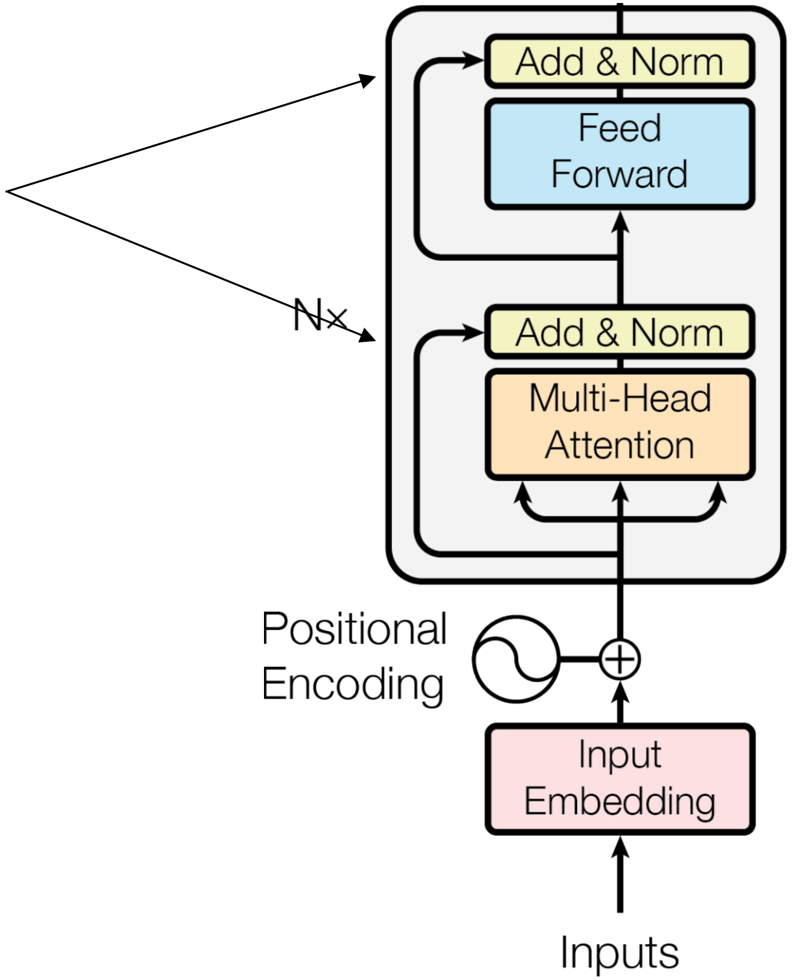
2 – Transformer-Encoder

Layer Normalization

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_{ij}$$

$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}(x_{ij} - \mu_i)^2$$

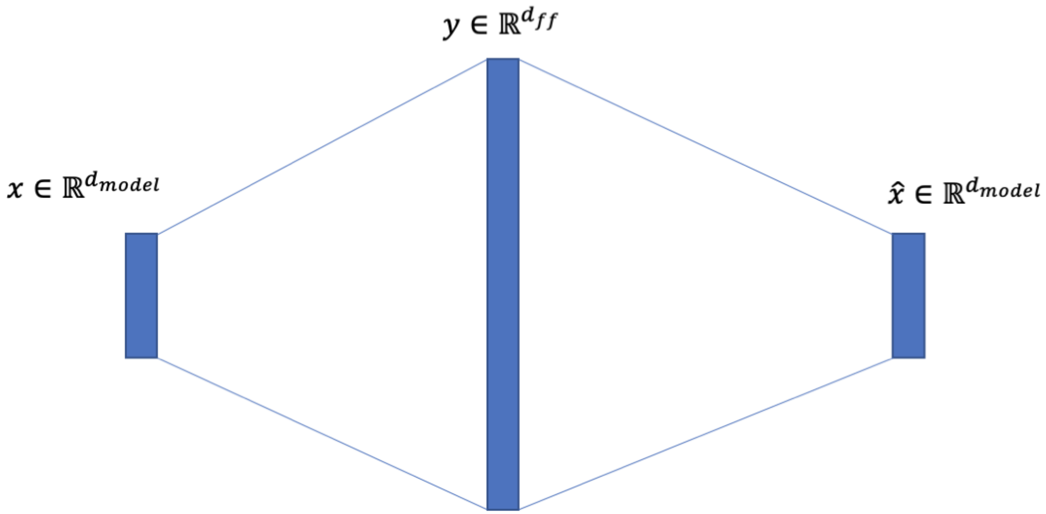
$$\hat{x}_{ij} = \frac{(x_{ij} - \mu_i)}{\sqrt{\sigma_i^2 + \epsilon}}$$

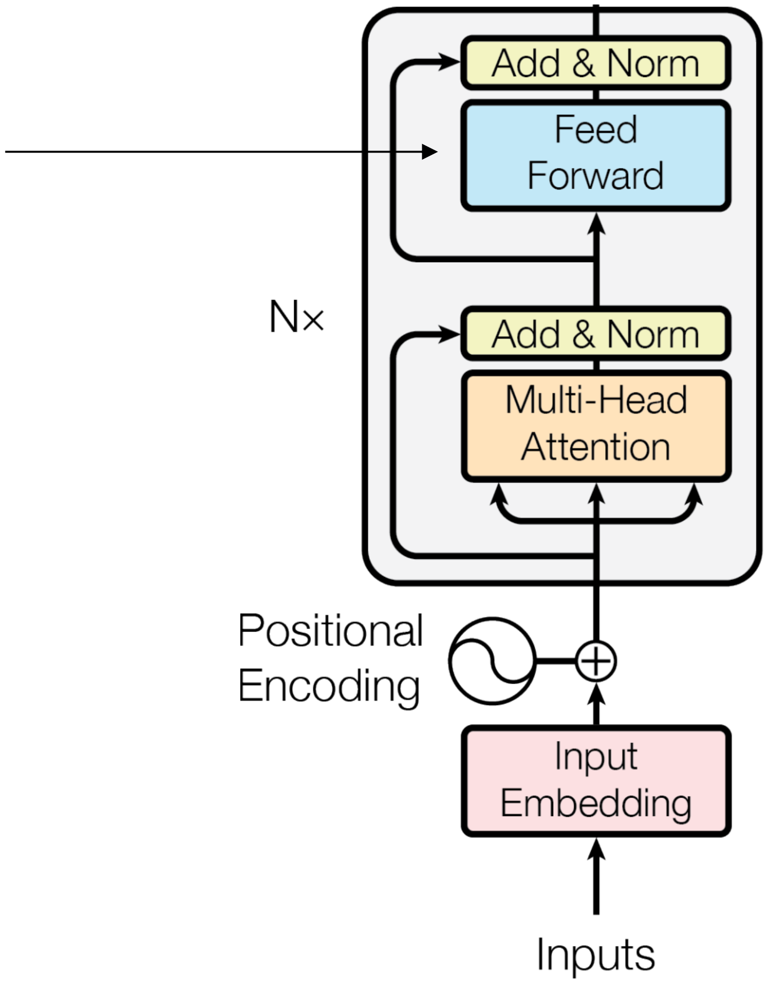


2 – Transformer-Encoder

Feed Forward

- 2 FC Layer





2 – Transformer-Encoder

Transformer-Encoder – Demo

```
class TransformerEncoder(nn.Module):
    def __init__(self, embed_dim, num_heads, ff_dim, dropout=0.1):
        super().__init__()
        self.attn = nn.MultiheadAttention(
            embed_dim=embed_dim,
            num_heads=num_heads,
            batch_first=True
        )
        self.ffn = nn.Sequential(
            nn.Linear(in_features=embed_dim, out_features=ff_dim, bias=True),
            nn.ReLU(),
            nn.Linear(in_features=ff_dim, out_features=embed_dim, bias=True)
        )
        self.layernorm_1 = nn.LayerNorm(normalized_shape=embed_dim, eps=1e-6)
        self.layernorm_2 = nn.LayerNorm(normalized_shape=embed_dim, eps=1e-6)
        self.dropout_1 = nn.Dropout(p=dropout)
        self.dropout_2 = nn.Dropout(p=dropout)

    def forward(self, query, key, value):
        attn_output, _ = self.attn(query, key, value)
        attn_output = self.dropout_1(attn_output)
        out_1 = self.layernorm_1(query + attn_output)
        ffn_output = self.ffn(out_1)
        ffn_output = self.dropout_2(ffn_output)
        out_2 = self.layernorm_2(out_1 + ffn_output)
        return out_2
```

```
encoder_layer = TransformerEncoder(
    embed_dim=200,
    num_heads=5,
    ff_dim=1024
)
```

```
embedded.shape
```

```
torch.Size([32, 50, 200])
```

```
encoded = encoder_layer(embedded, embedded, embedded)
```

```
encoded.shape
```

```
torch.Size([32, 50, 200])
```

3 – Text Classification

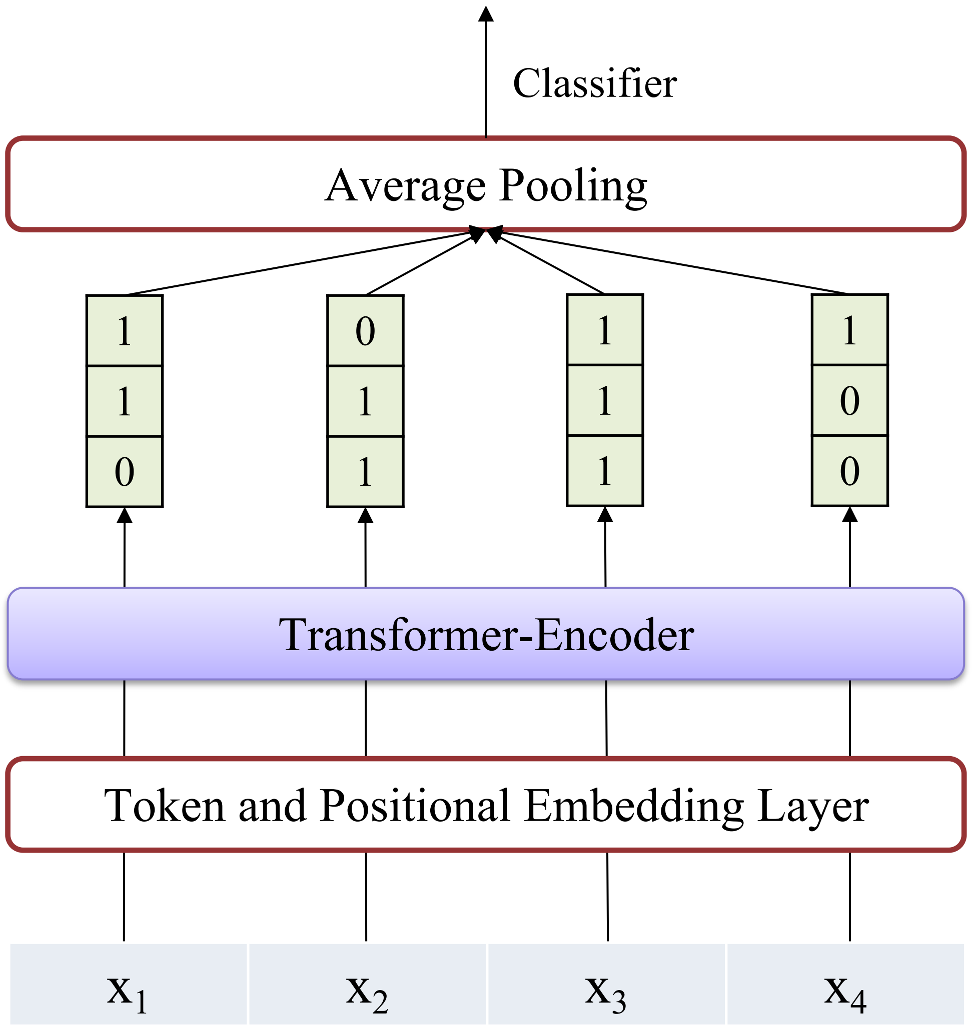
NTC-SCV Dataset

❖ **Sentiment Analysis**

Positive Example	Negative Example
Mình được 1 cô bạn giới_thiệu đến đây , tìm địa_chỉ khá dễ . Menu nước uống chất khỏi nói . Mình muốn cũng đc 8 loại nước ở đây , món nào cũng ngon và bổ_dưỡng cả .	Quán chế_biến đồ_ăn lâu , Cá_Sapa nướng uớp rất dở , sò Lông ko tươi , nước_chấm ko ngon\n Tóm_lại sẽ ko bao_giờ ghé nữa , ăn_dở mà uổng tiền
Mỗi lần thèm trà sữa là làm 1 ly . Quán dễ kiếm , không_gian lại rộng_rãi . Nhân_viên thì dễ_thương gần_gũi . Nói_chung thèm trà sữa là mình ghé Quán ở đây vì gần nhà .	Quán này thấy khá nhiều người bảo mình nên mình đã đi ăn thử , nhưng thực_sự ăn xong thấy không được như mong_đợi lắm .

3 – Text Classification

Modeling



3 – Text Classification

Modeling – Demo

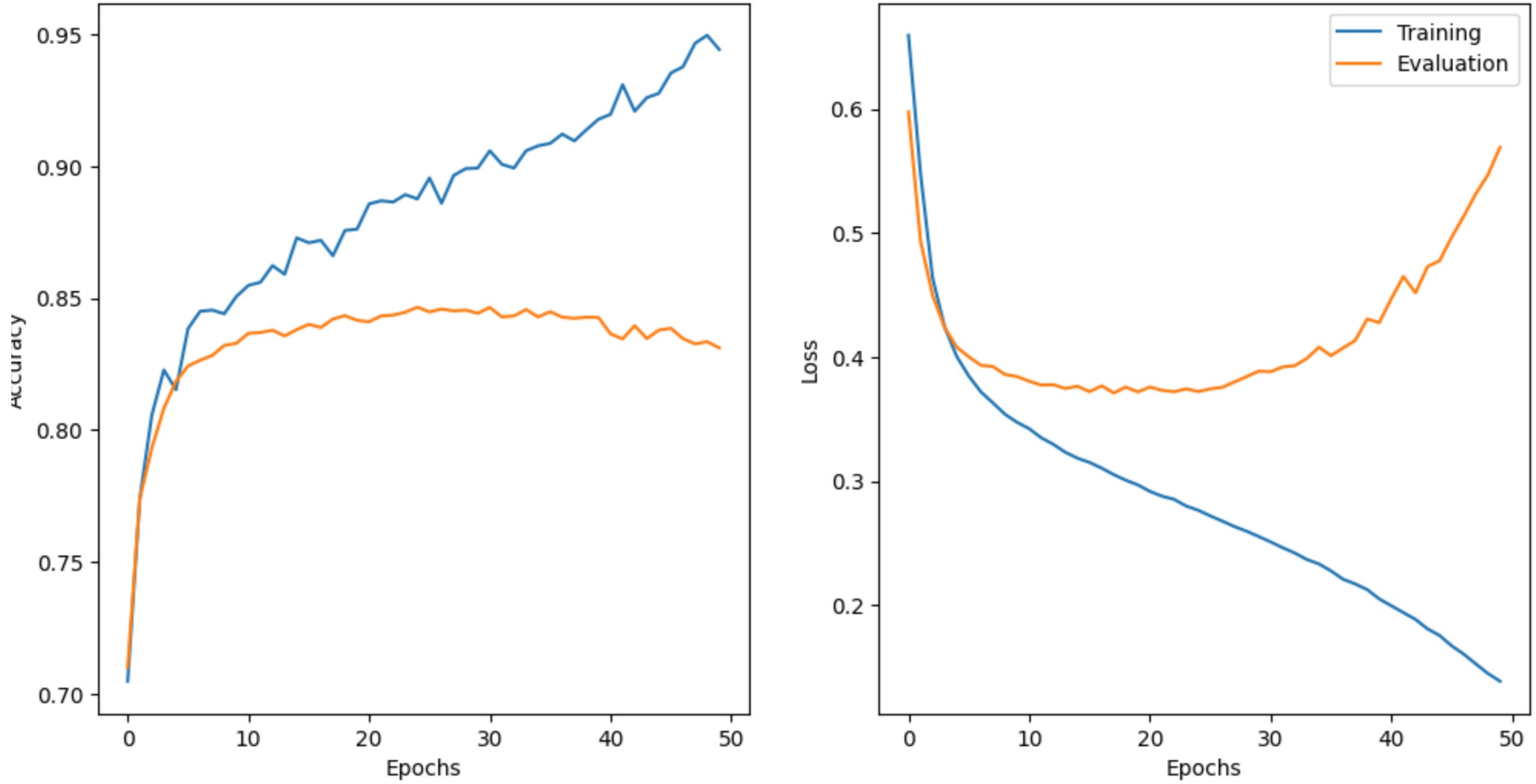
```
class TransformerEncoder(nn.Module):
    def __init__(self, embed_dim, num_heads, ff_dim, dropout=0.1):
        super().__init__()
        self.attn = nn.MultiheadAttention(
            embed_dim=embed_dim,
            num_heads=num_heads,
            batch_first=True
        )
        self.ffn = nn.Sequential(
            nn.Linear(in_features=embed_dim, out_features=ff_dim, bias=True),
            nn.ReLU(),
            nn.Linear(in_features=ff_dim, out_features=embed_dim, bias=True)
        )
        self.layernorm_1 = nn.LayerNorm(normalized_shape=embed_dim, eps=1e-6)
        self.layernorm_2 = nn.LayerNorm(normalized_shape=embed_dim, eps=1e-6)
        self.dropout_1 = nn.Dropout(p=dropout)
        self.dropout_2 = nn.Dropout(p=dropout)

    def forward(self, query, key, value):
        attn_output, _ = self.attn(query, key, value)
        attn_output = self.dropout_1(attn_output)
        out_1 = self.layernorm_1(query + attn_output)
        ffn_output = self.ffn(out_1)
        ffn_output = self.dropout_2(ffn_output)
        out_2 = self.layernorm_2(out_1 + ffn_output)
        return out_2
```

3 – Text Classification

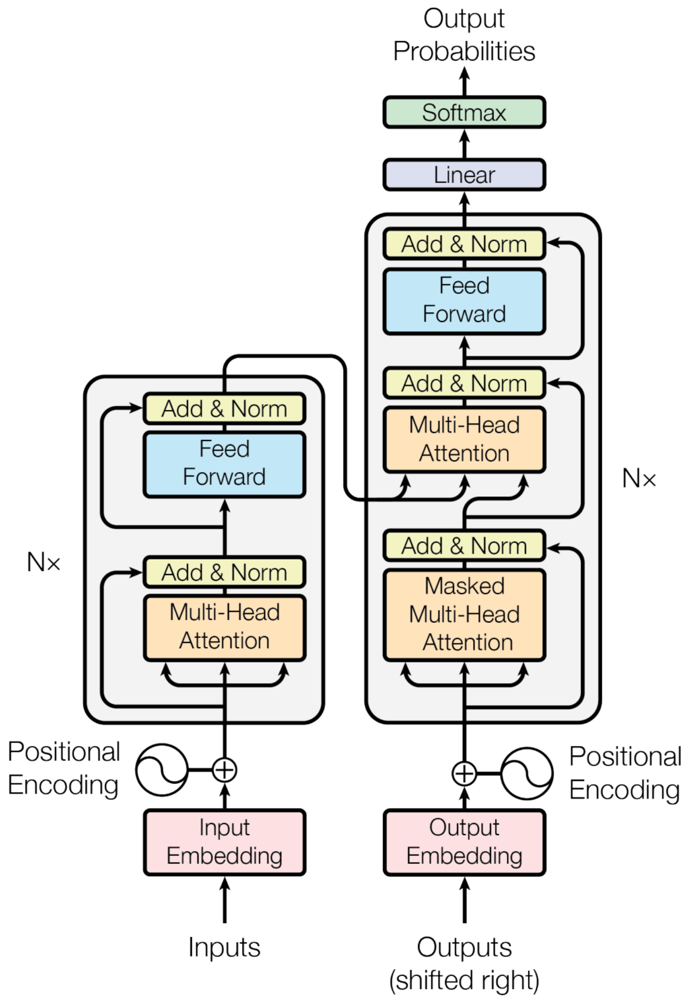
Training

❖ Testing: 83.66%



4 – Vision Transformer

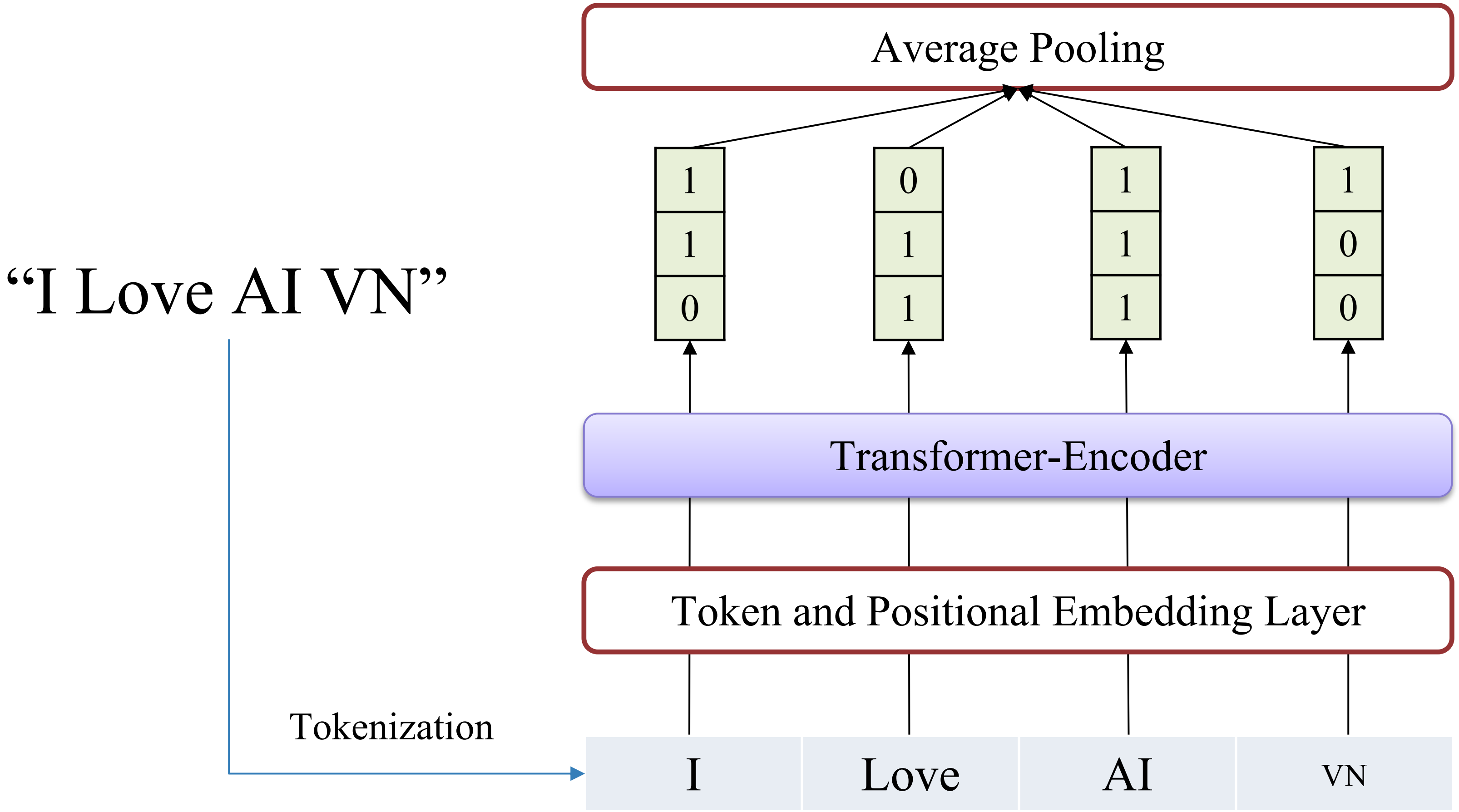
ViT



Transformers are so successful in NLP,
Can we use them for images?

4 – Vision Transformer

From text to image



4 – Vision Transformer

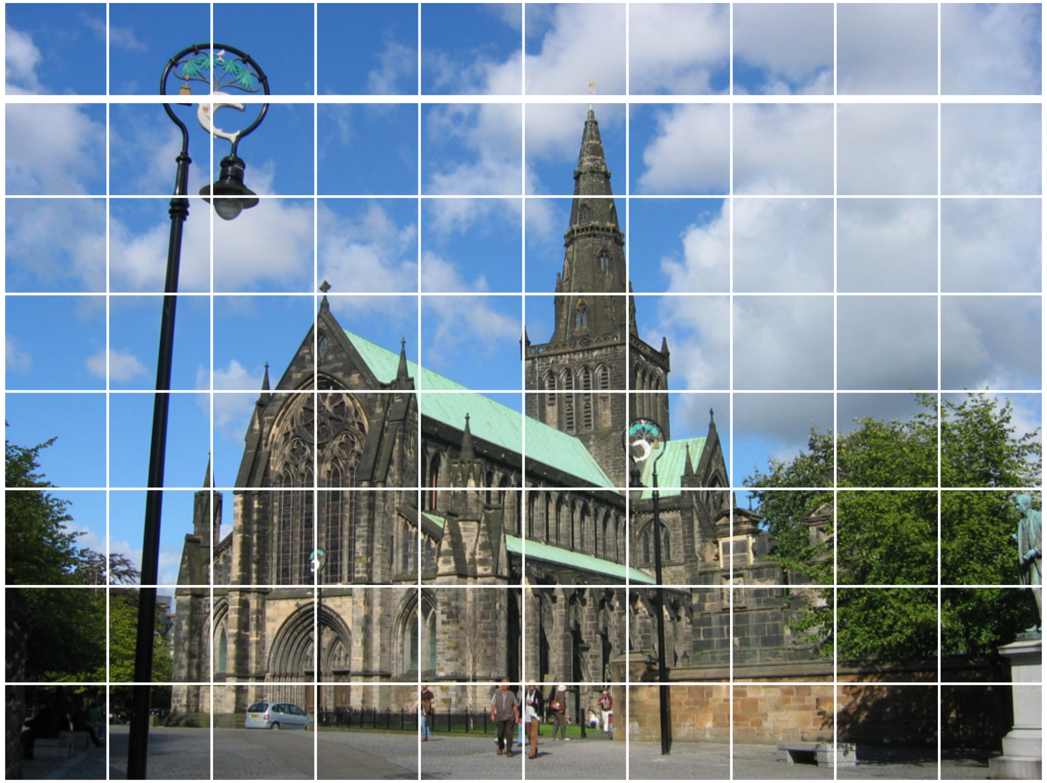
From text to image

Can we tokenize an image?

4 – Vision Transformer

From text to image

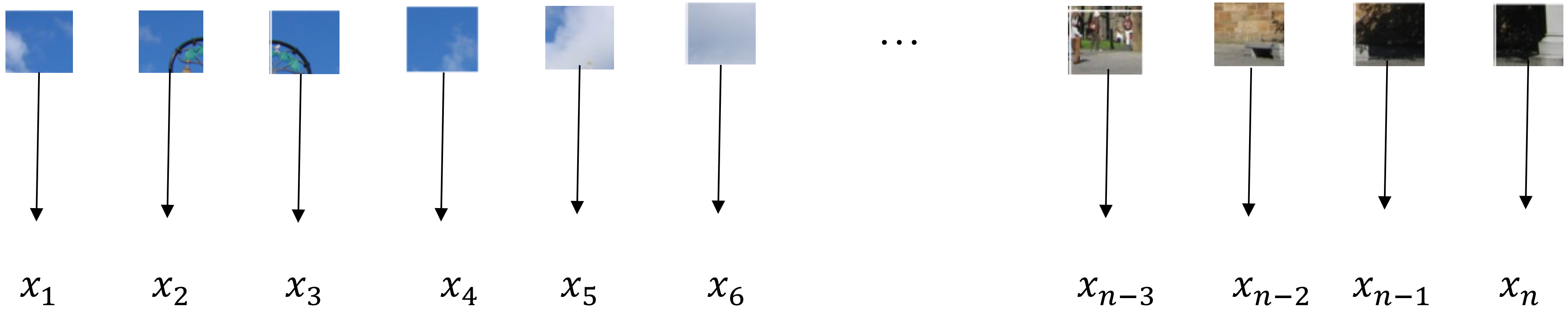
Can we tokenize an image?



4 – Vision Transformer

From text to image

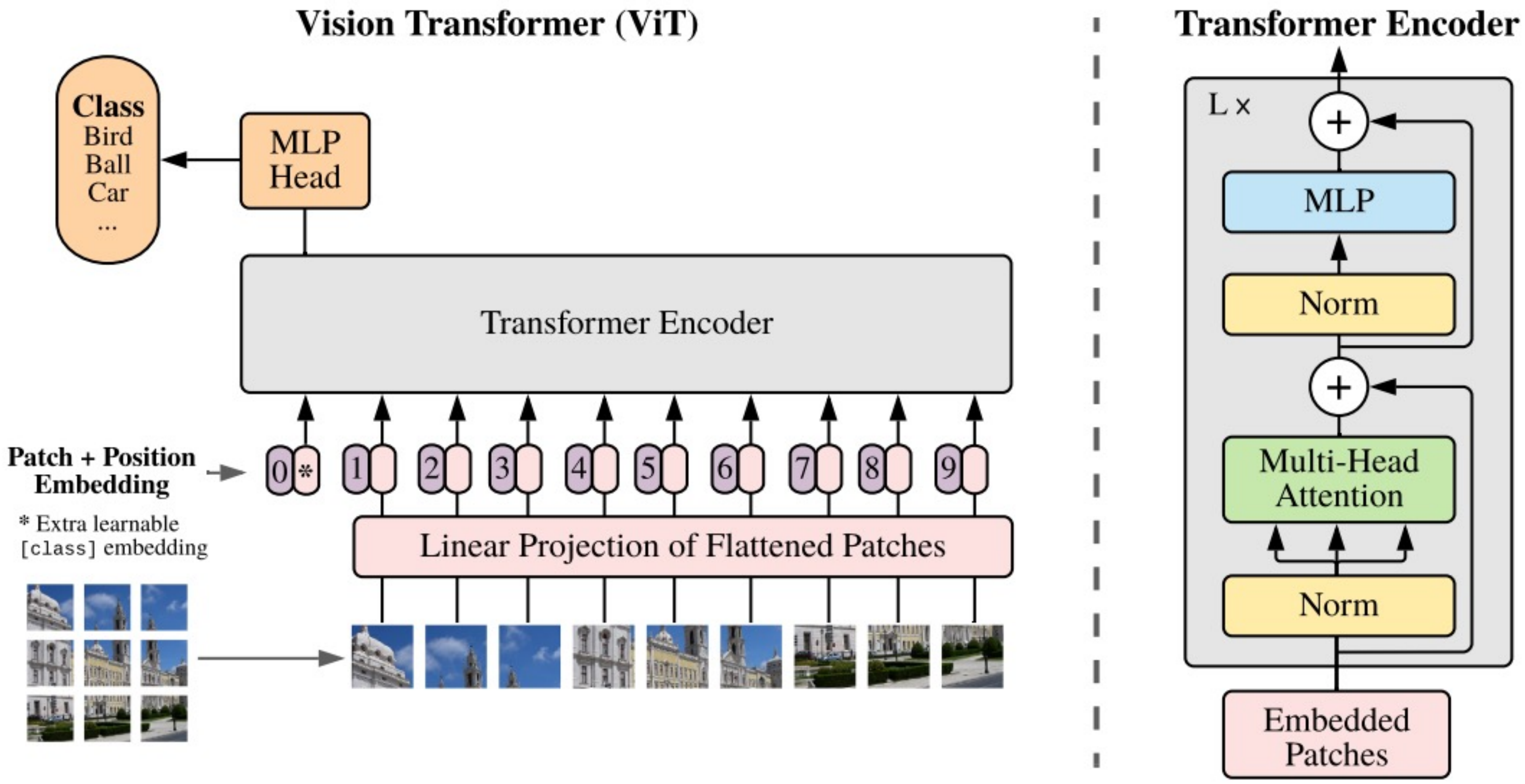
Can we tokenize an image?



Flattening

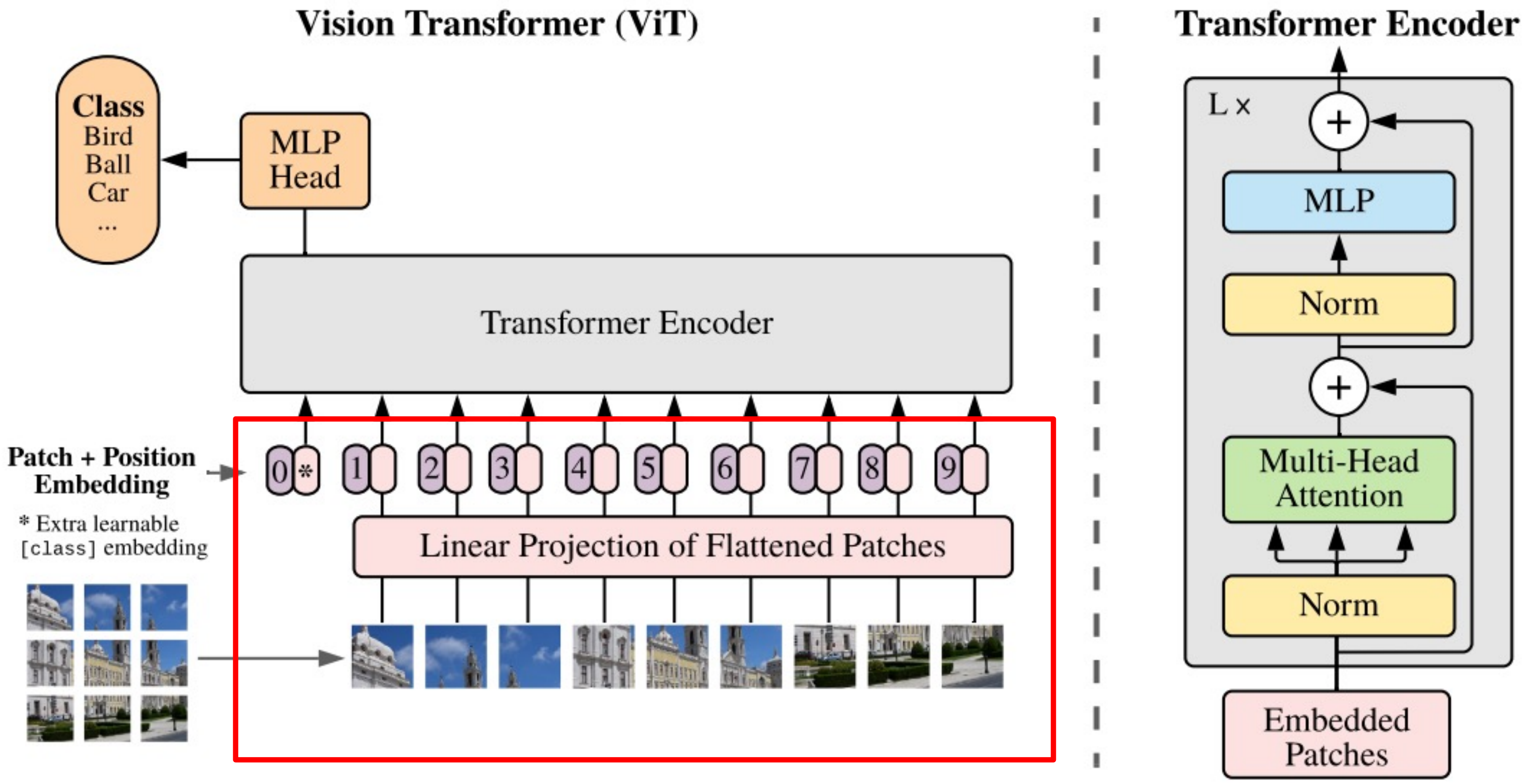
4 – Vision Transformer

ViT Architecture



4 – Vision Transformer

Patch embedding



AI VIET NAM
@aivietnam.edu.vn

4 – Vision Transformer

Patch embedding

```python
class PatchEmbedding(nn.Module):
    def __init__(self, embed_dim=512, patch_size=16, image_size=224):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=embed_dim, kernel_size=patch_size, stride=patch_size, bias=False)

    def forward(self, x):
        x = self.conv1(x) # shape = [*, width, grid, grid]
        x = x.reshape(x.shape[0], x.shape[1], -1)  # shape = [*, width, grid ** 2]
        x = x.permute(0, 2, 1)  # shape = [*, grid ** 2, width]
        return x
```

```python
patch_embedding = PatchEmbedding()
x = torch.randn(1, 3, 224, 224)

out = patch_embedding(x)
print(out.shape)
```

```
torch.Size([1, 196, 512])
```

4 – Vision Transformer

Patch embedding

1	2	4	2	2	3	3	2
1	0	2	1	2	1	1	1
2	2	3	4	3	4	1	3
2	1	3	0	0	2	3	0
3	3	4	0	2	0	2	2
1	4	4	3	4	0	4	0
1	2	0	0	0	3	2	3
4	1	4	1	0	0	0	0

Patch size: 4

1	0	0	1
0	1	1	0
1	0	0	1
1	1	0	1

1	1	0	0
0	0	1	0
0	1	0	0
1	1	0	0

→

14	10
14	12
18	17
11	9

4 – Vision Transformer

Positional embedding

Embed + add

4 – Vision Transformer

Positional embedding

```
class PatchPositionEmbedding(nn.Module):
    def __init__(self, embed_dim=512, patch_size=16, image_size=224):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=embed_dim, kernel_size=patch_size, stride=patch_size, bias=False)

        scale = embed_dim ** -0.5
        self.positional_embedding = nn.Parameter(scale * torch.randn((image_size // patch_size) ** 2, embed_dim))

    def forward(self, x):
        x = self.conv1(x) # shape = [*, width, grid, grid]
        x = x.reshape(x.shape[0], x.shape[1], -1)  # shape = [*, width, grid ** 2]
        x = x.permute(0, 2, 1)  # shape = [*, grid ** 2, width]

        x = x + self.positional_embedding.to(x.dtype)
        return x
```

```
[36]
patchpos_embedding = PatchPositionEmbedding()
x = torch.randn(1, 3, 224, 224)

out = patchpos_embedding(x)
print(out.shape)
```

```
torch.Size([1, 196, 512])
```

4 – Vision Transformer

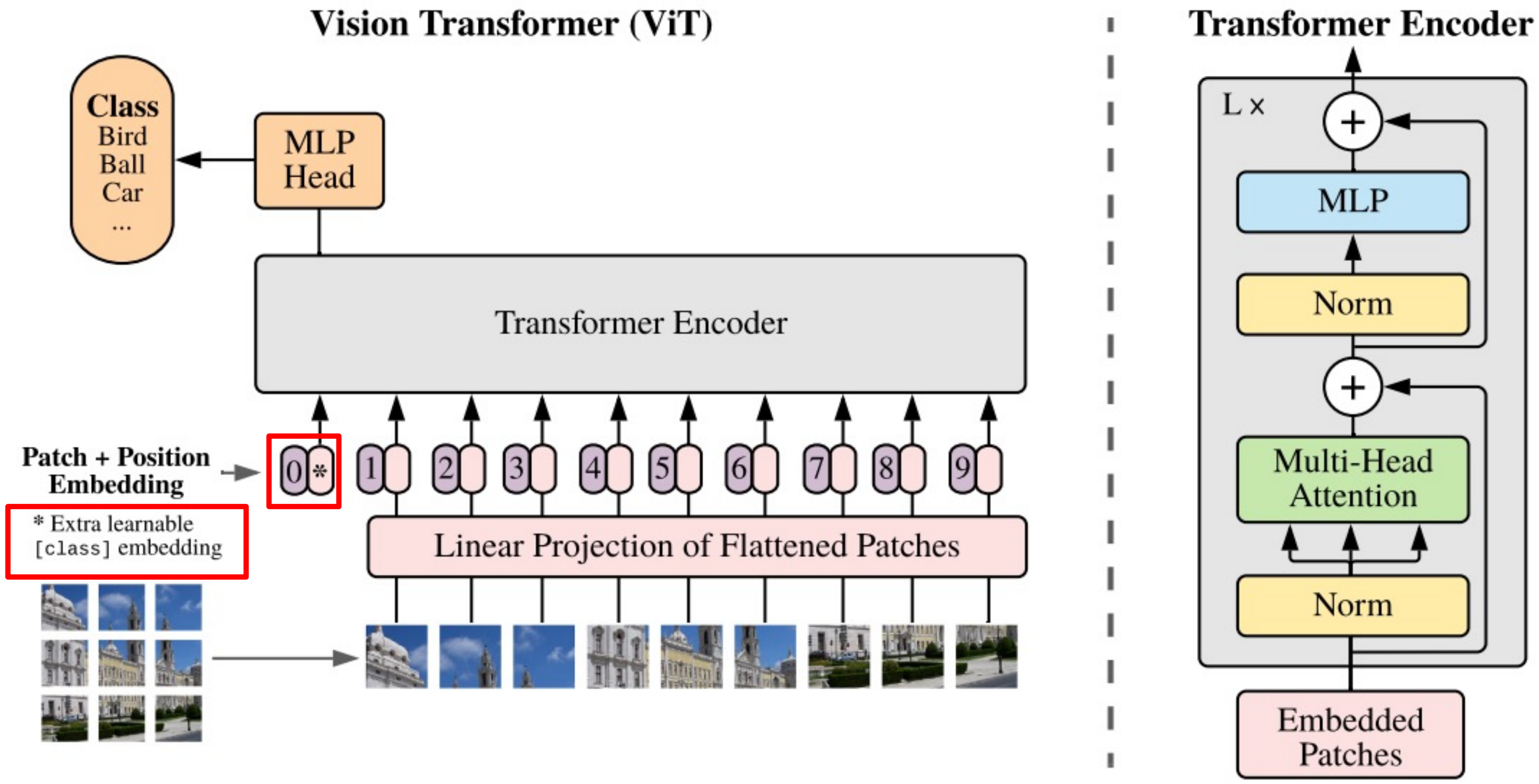
! Positional embedding

$$\begin{bmatrix} 14 & 10 \\ 14 & 12 \\ 18 & 17 \\ 11 & 9 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 14 & 10 \\ 15 & 13 \\ 19 & 18 \\ 11 & 9 \end{bmatrix}$$

Positional
Embedding

4 – Vision Transformer

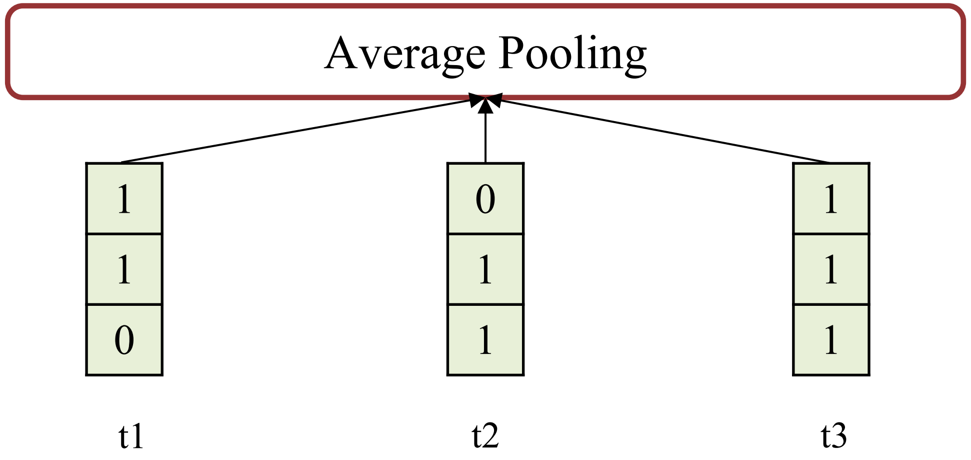
[CLS] Token

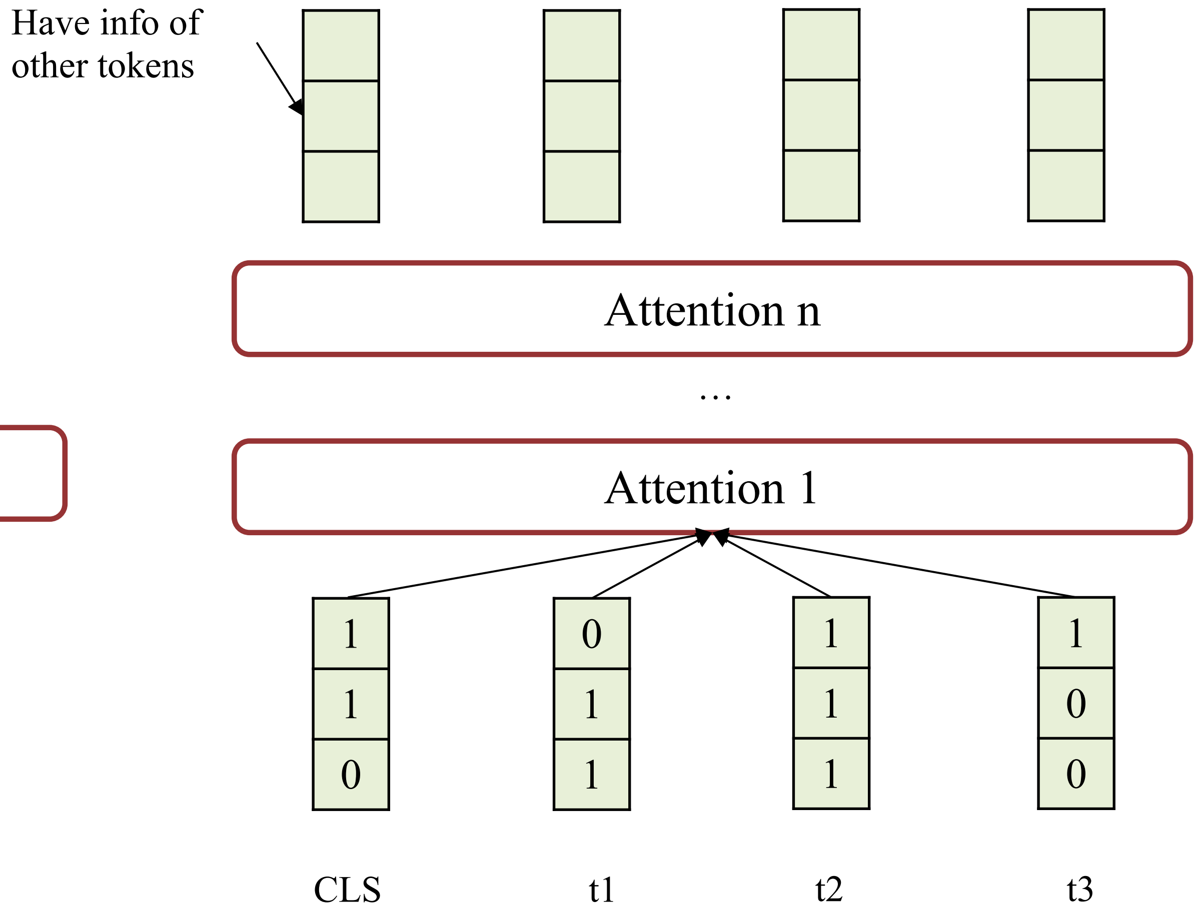


4 – Vision Transformer

[CLS] Token – Why?

Alternatives?
- Global Average Pooling
- Max Pooling
- …





4 – Vision Transformer

[CLS] Token – Demo

```python
class PatchPositionEmbedding(nn.Module):
    def __init__(self, embed_dim=512, patch_size=16, image_size=224):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=embed_dim, kernel_size=patch_size, stride=patch_size, bias=False)

        scale = embed_dim ** -0.5
        self.class_embedding = nn.Parameter(scale * torch.randn(embed_dim))
        self.positional_embedding = nn.Parameter(scale * torch.randn((image_size // patch_size) ** 2 + 1, embed_dim))

    def forward(self, x):
        x = self.conv1(x) # shape = [*, width, grid, grid]
        x = x.reshape(x.shape[0], x.shape[1], -1)  # shape = [*, width, grid ** 2]
        x = x.permute(0, 2, 1)  # shape = [*, grid ** 2, width]
        # expanding the CLS embedding
        cls_embs = self.class_embedding.to(x.dtype) + torch.zeros(x.shape[0], 1, x.shape[-1], dtype=x.dtype, device=x.device)
        x = torch.cat([cls_embs, x], dim=1)  # shape = [*, grid ** 2 + 1, width]

        x = x + self.positional_embedding.to(x.dtype)
        return x
```

4 – Vision Transformer

Modeling

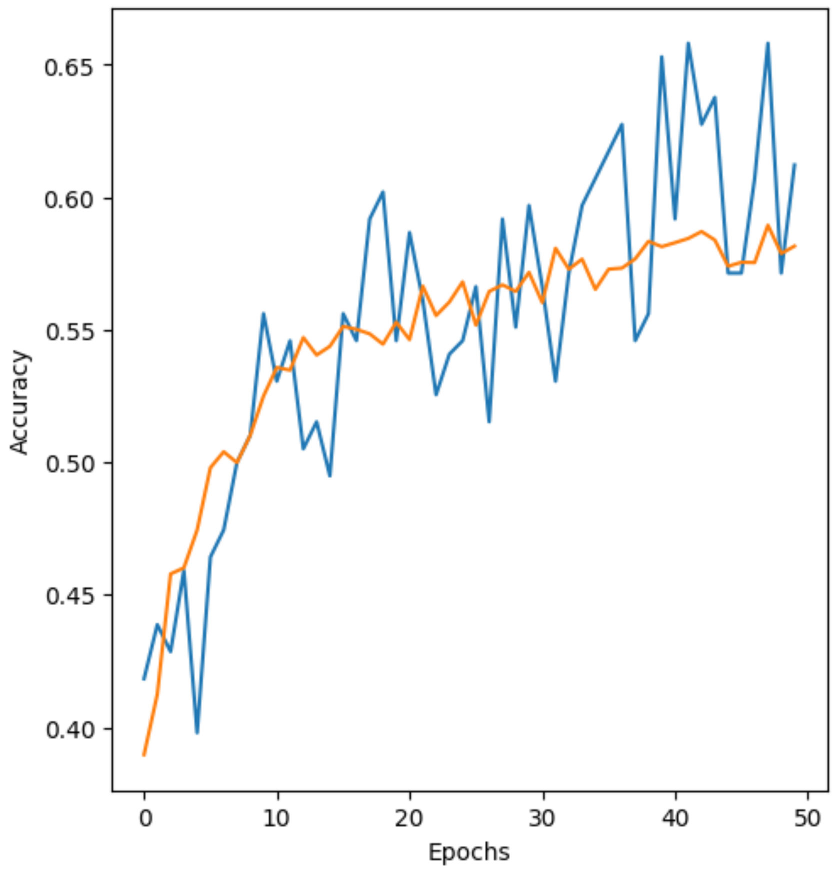
Change Token Embedding with Patch Embedding

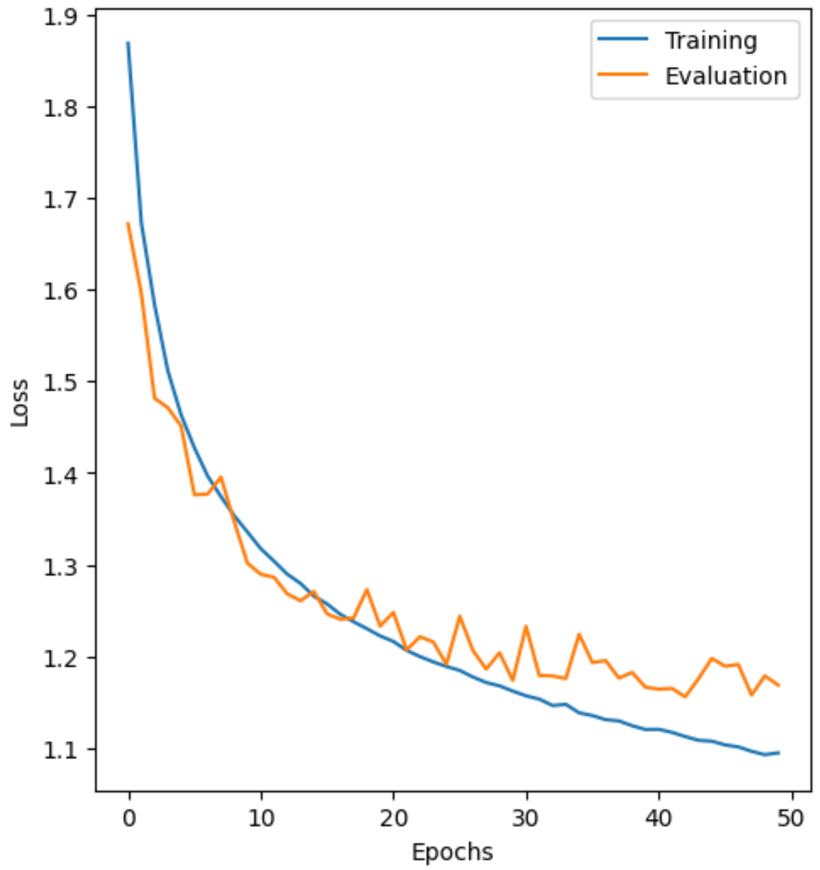
[CLS] token instead of pooling (can still use pooling)

```python
class VisionTransformerCls(nn.Module):
    def __init__(self,
                 image_size, embed_dim, num_heads, ff_dim,
                 dropout=0.1, device='cpu', num_classes = 10, patch_size=16
        ):
        super().__init__()
        self.embd_layer = PatchPositionEmbedding(
            image_size=image_size, embed_dim=embed_dim, patch_size=patch_size
        )
        self.transformer_layer = TransformerEncoder(
            embed_dim, num_heads, ff_dim, dropout
        )
        # self.pooling = nn.AvgPool1d(kernel_size=max_length)
        self.fc1 = nn.Linear(in_features=embed_dim, out_features=20)
        self.fc2 = nn.Linear(in_features=20, out_features=num_classes)
        self.dropout = nn.Dropout(p=dropout)
        self.relu = nn.ReLU()
    def forward(self, x):
        output = self.embd_layer(x)
        output = self.transformer_layer(output, output, output)
        output = output[:, 0, :]
        output = self.dropout(output)
        output = self.fc1(output)
        output = self.dropout(output)
        output = self.fc2(output)
        return output
```

4 – Vision Transformer

Training





Thanks!

Any questions?