AI VIETNAM
All-in-One Course

Exercise Class

Transformer Applications

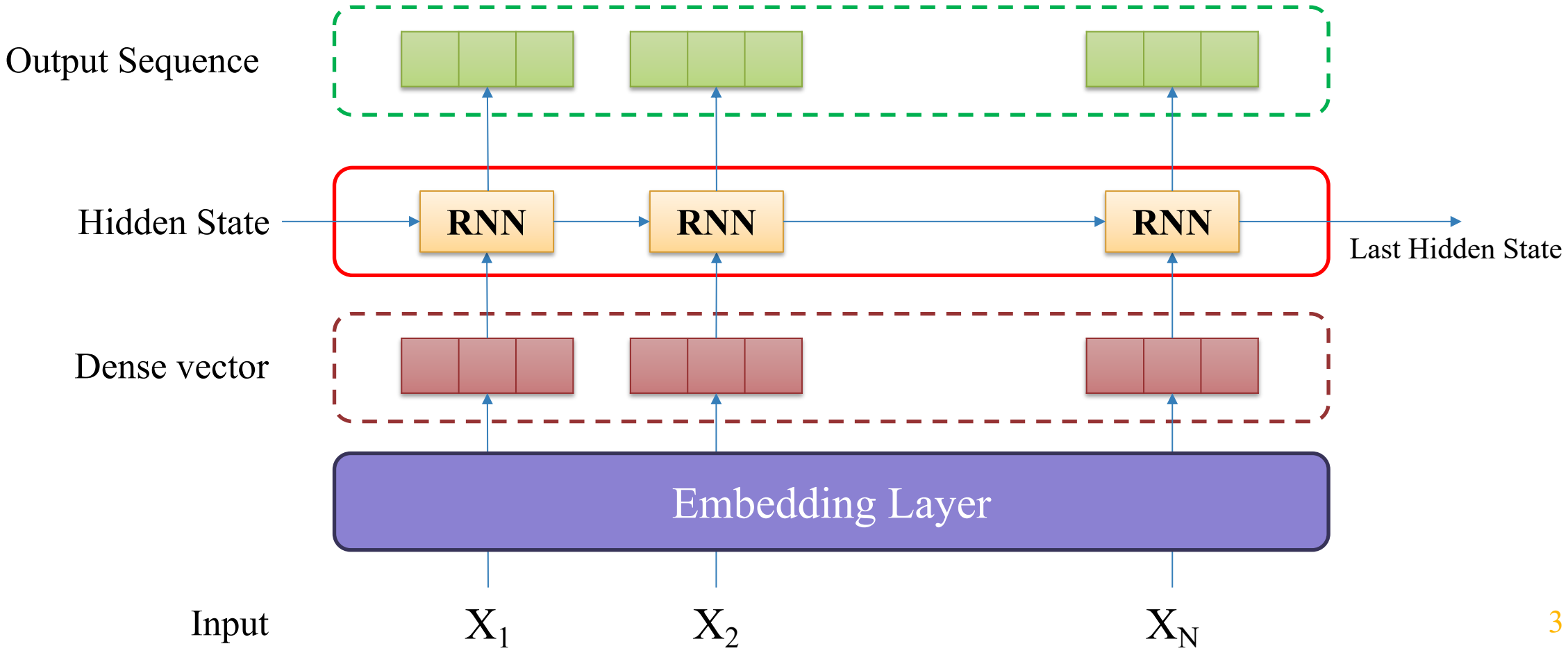
Nguyen Quoc Thai

Year 2023

CONTENT

1 – Sequence-to-Sequence

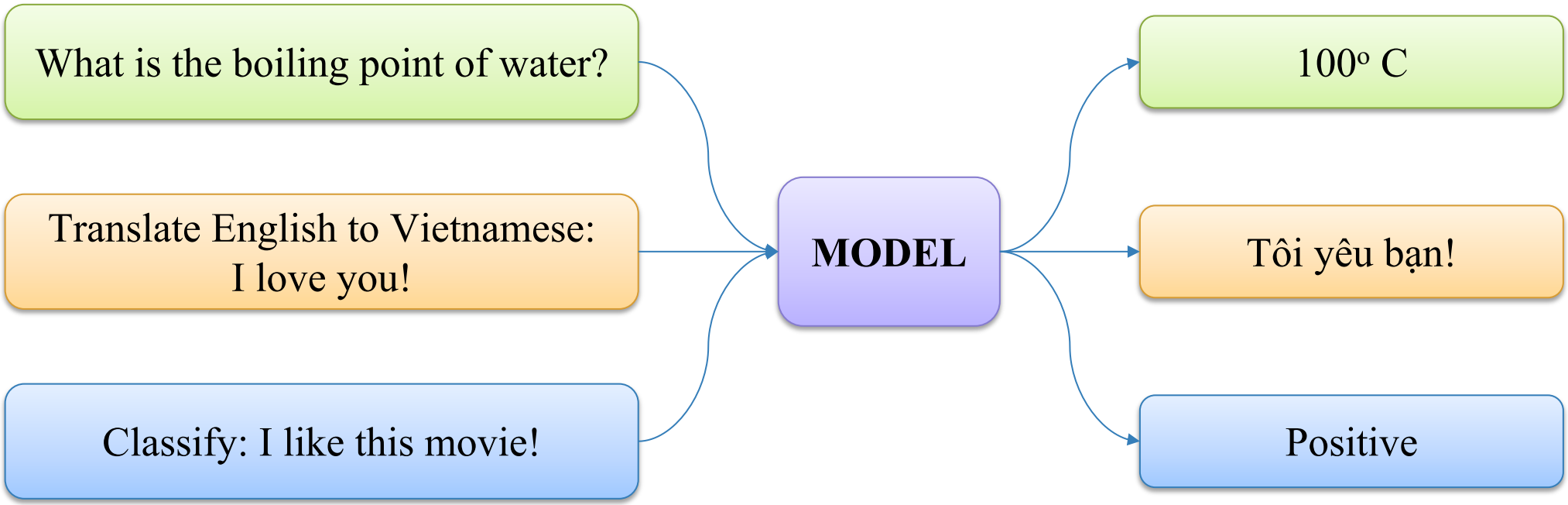
RNNs Model



1 – Sequence-to-Sequence

Text Generation

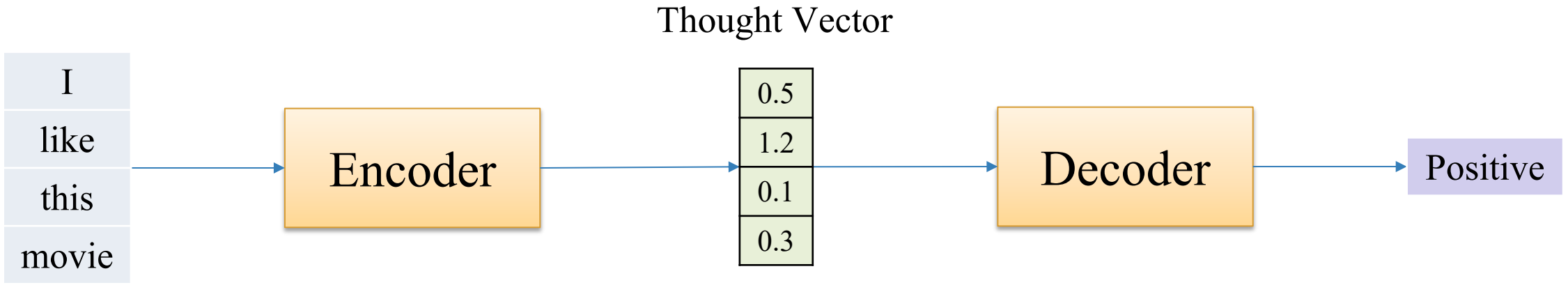
- Take a sequence as input, predict a sequence as output



1 – Sequence-to-Sequence

Sequence-to-Sequence Architecture

- Take a sequence as input, predict a sequence as output
- Encoder: encoding the inputs into state (thought vector)
- Decoder: the state is passed into the decoder to generate the output

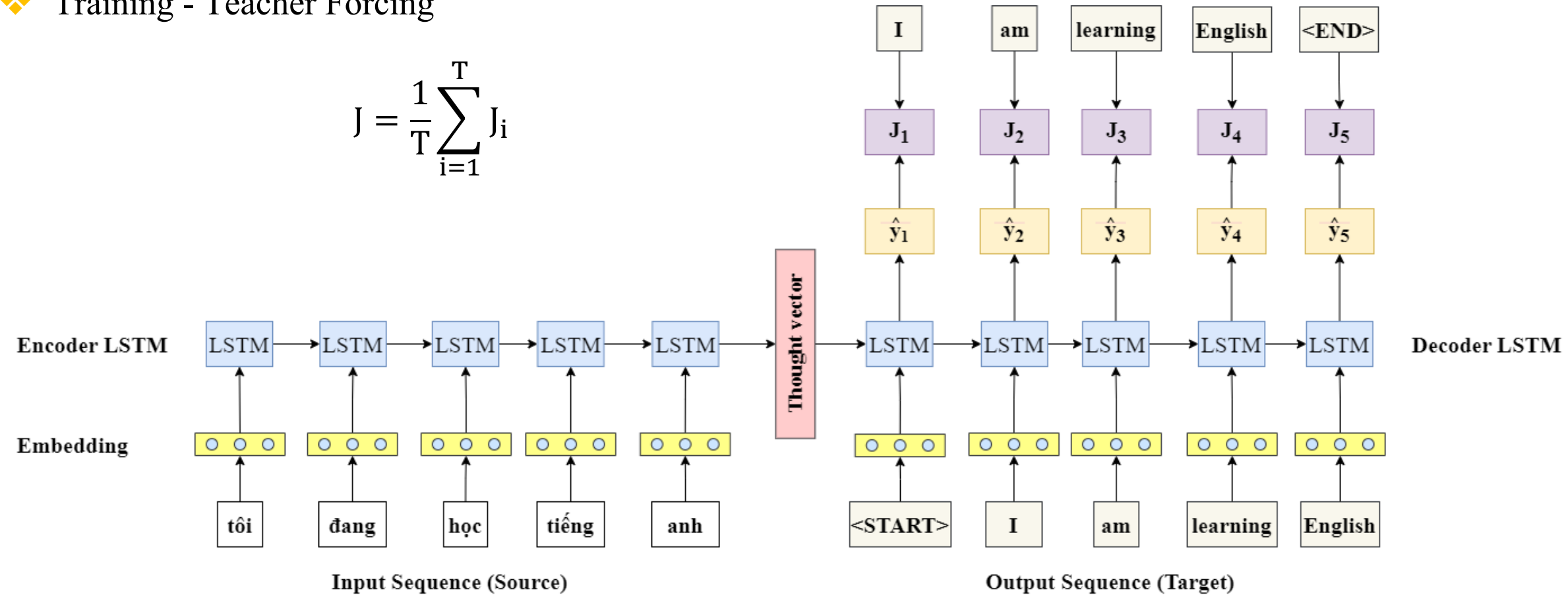


1 – Sequence-to-Sequence

Sequence-to-Sequence Architecture

- Training - Teacher Forcing

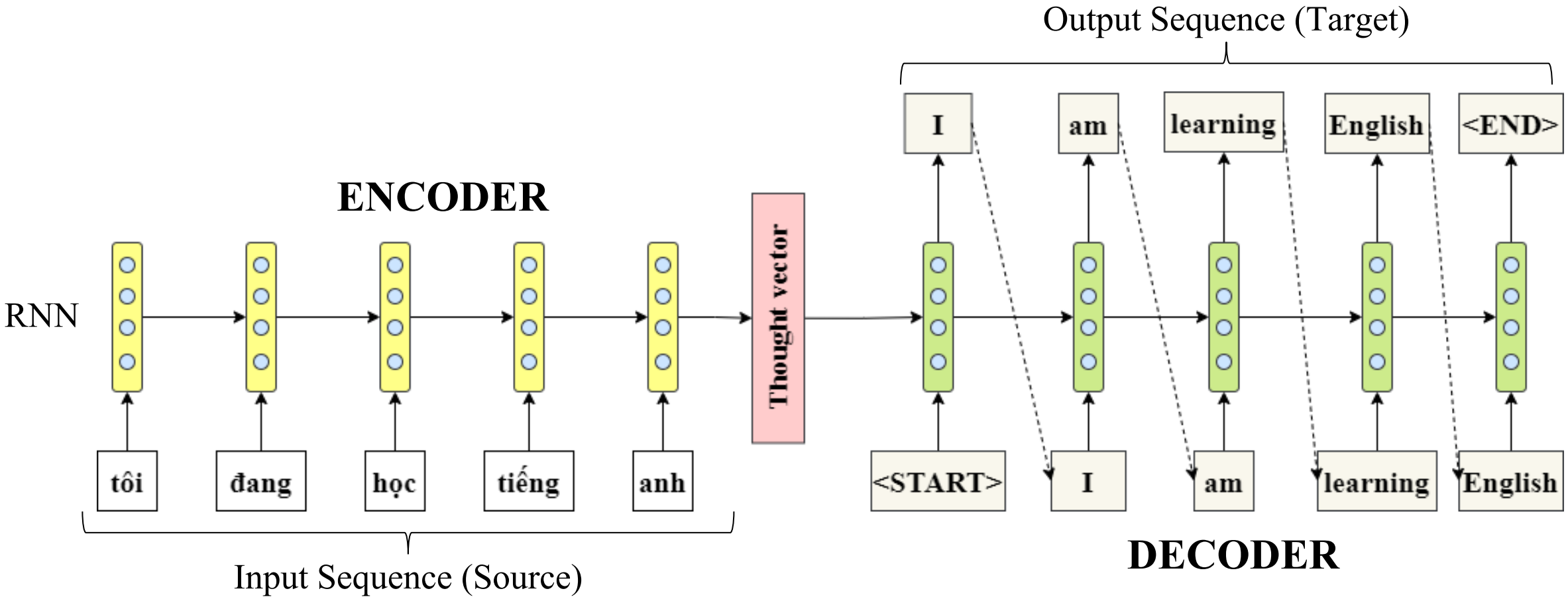
$$J = \frac{1}{T}\sum_{i=1}^{T} J_i$$



1 – Sequence-to-Sequence

Sequence-to-Sequence Architecture

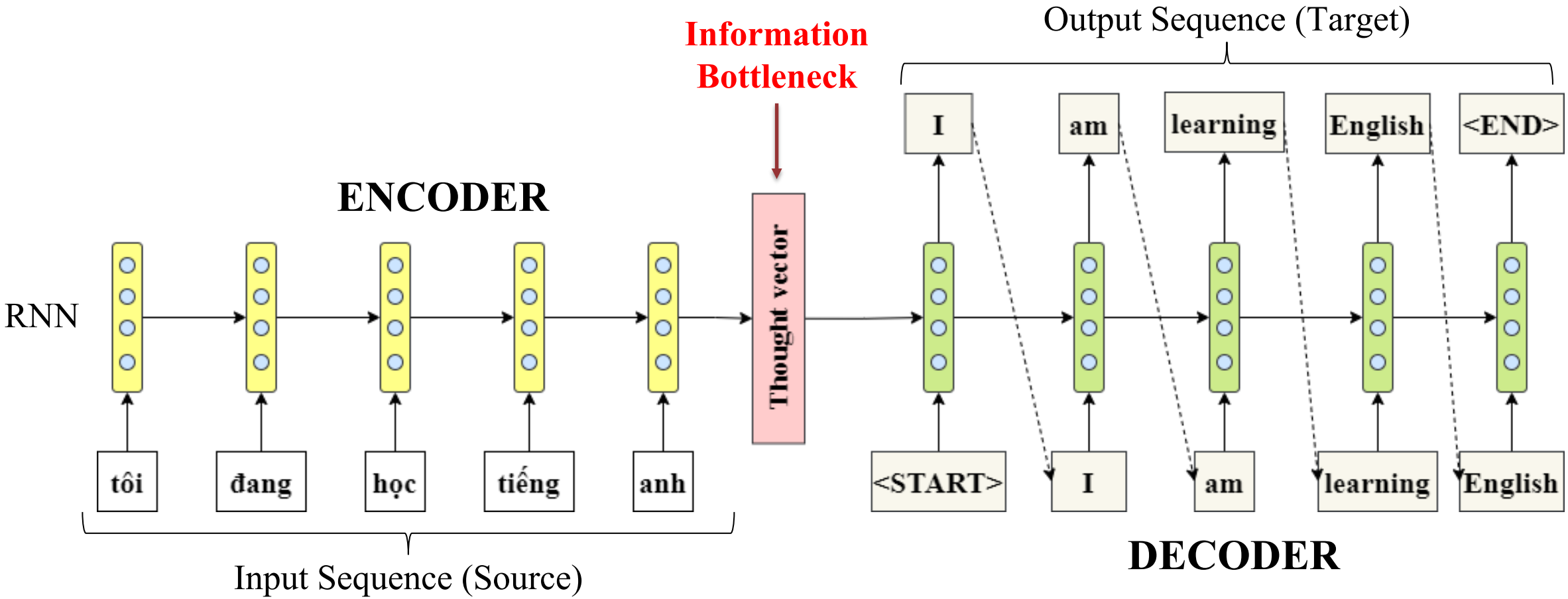
- Inference – Decoding – Greedy Search



1 – Sequence-to-Sequence

Information Bottleneck

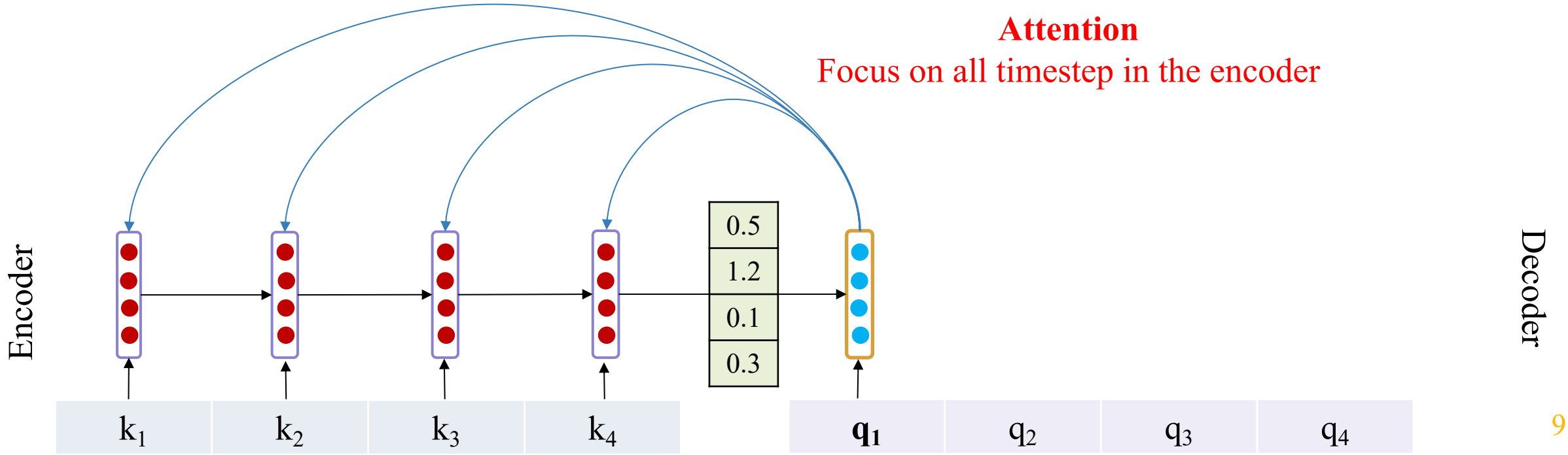
- Thought vector: capture all information of input sentence



1 – Sequence-to-Sequence

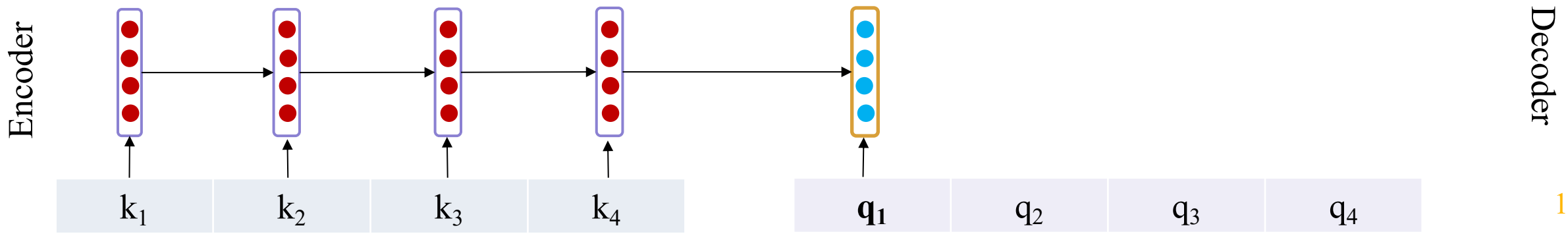
Information Bottleneck

- Thought vector: capture all information of input sentence



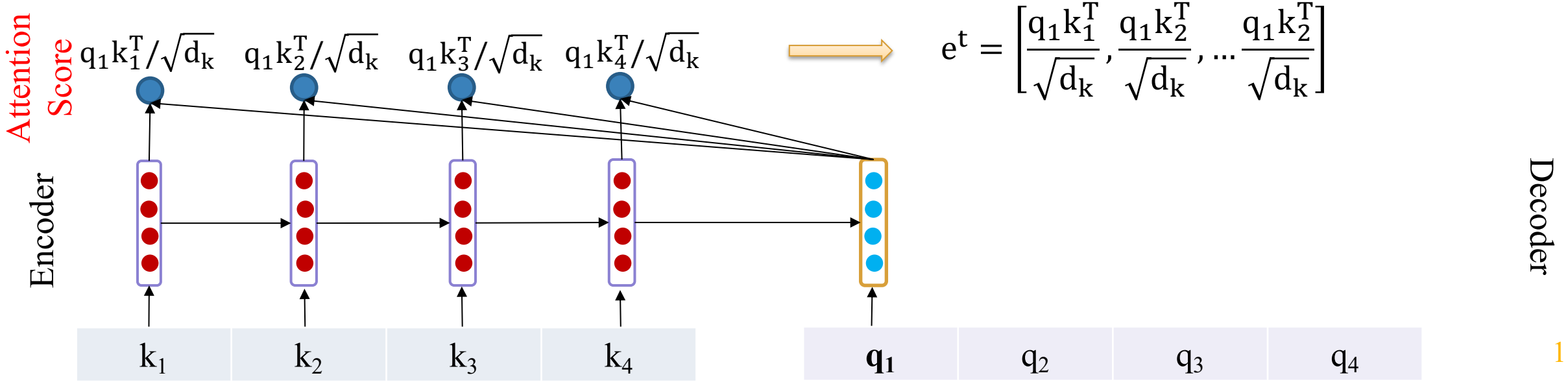
1 – Sequence-to-Sequence

Scaled Dot-Product Attention



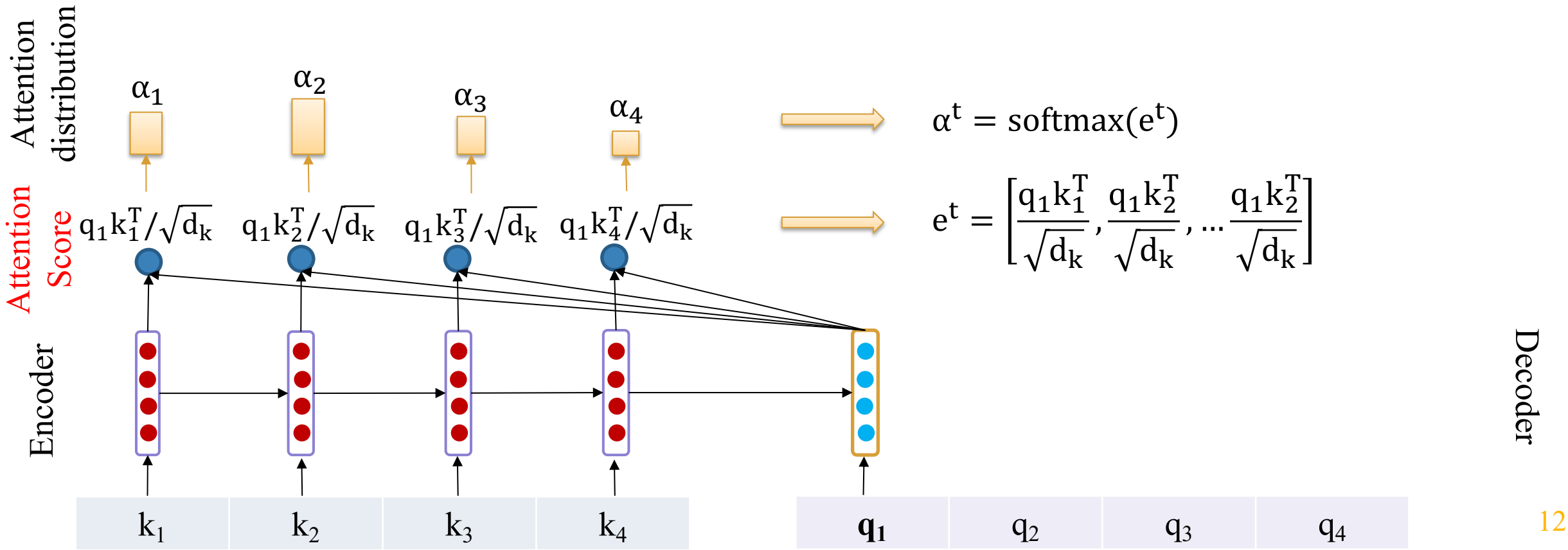
1 – Sequence-to-Sequence

Scaled Dot-Product Attention



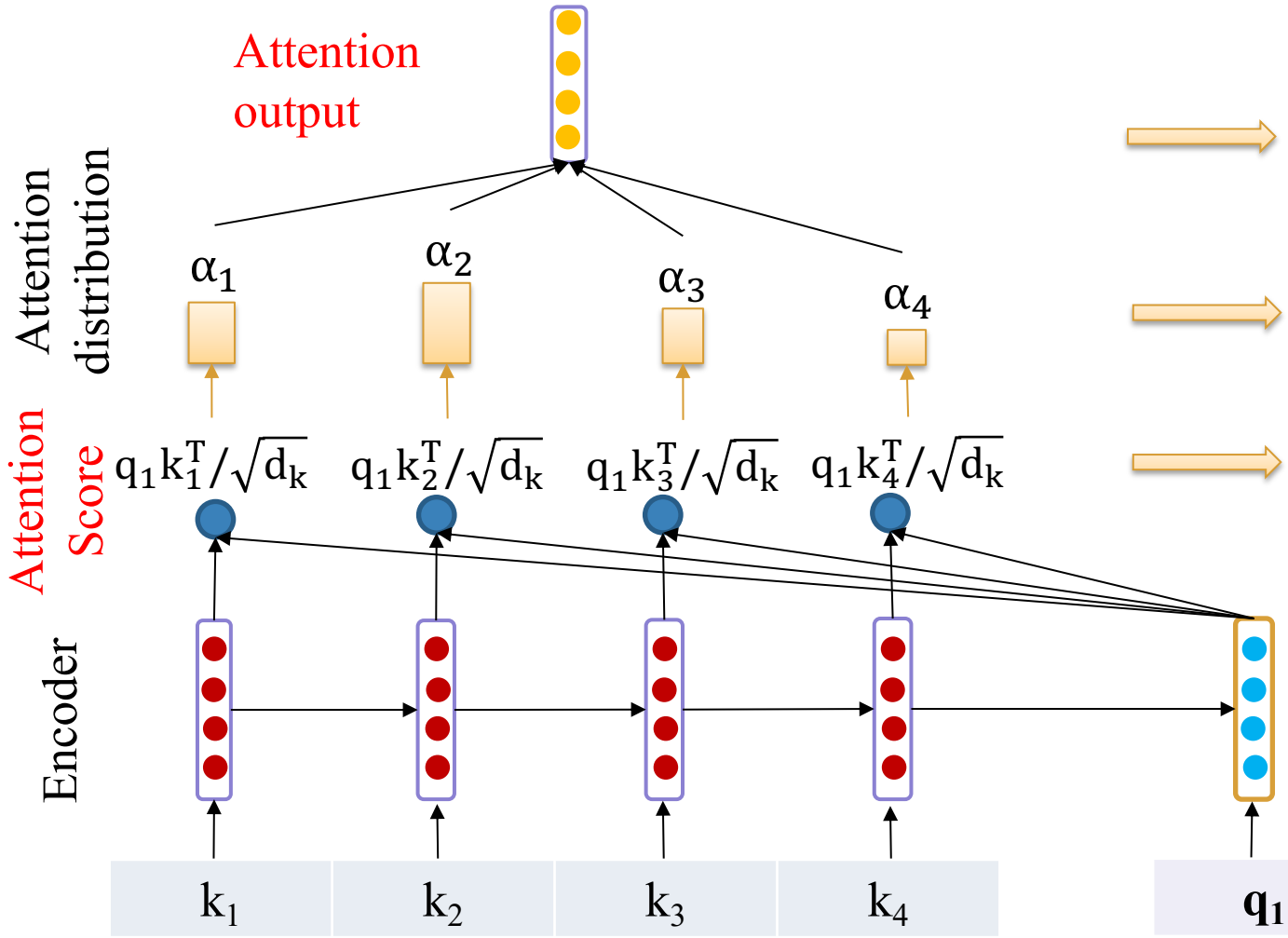
1 – Sequence-to-Sequence

Scaled Dot-Product Attention



1 – Sequence-to-Sequence

Scaled Dot-Product Attention



$$a^t = \sum_1^N \alpha_i^t v_i \; ; \; v_i = k_i$$

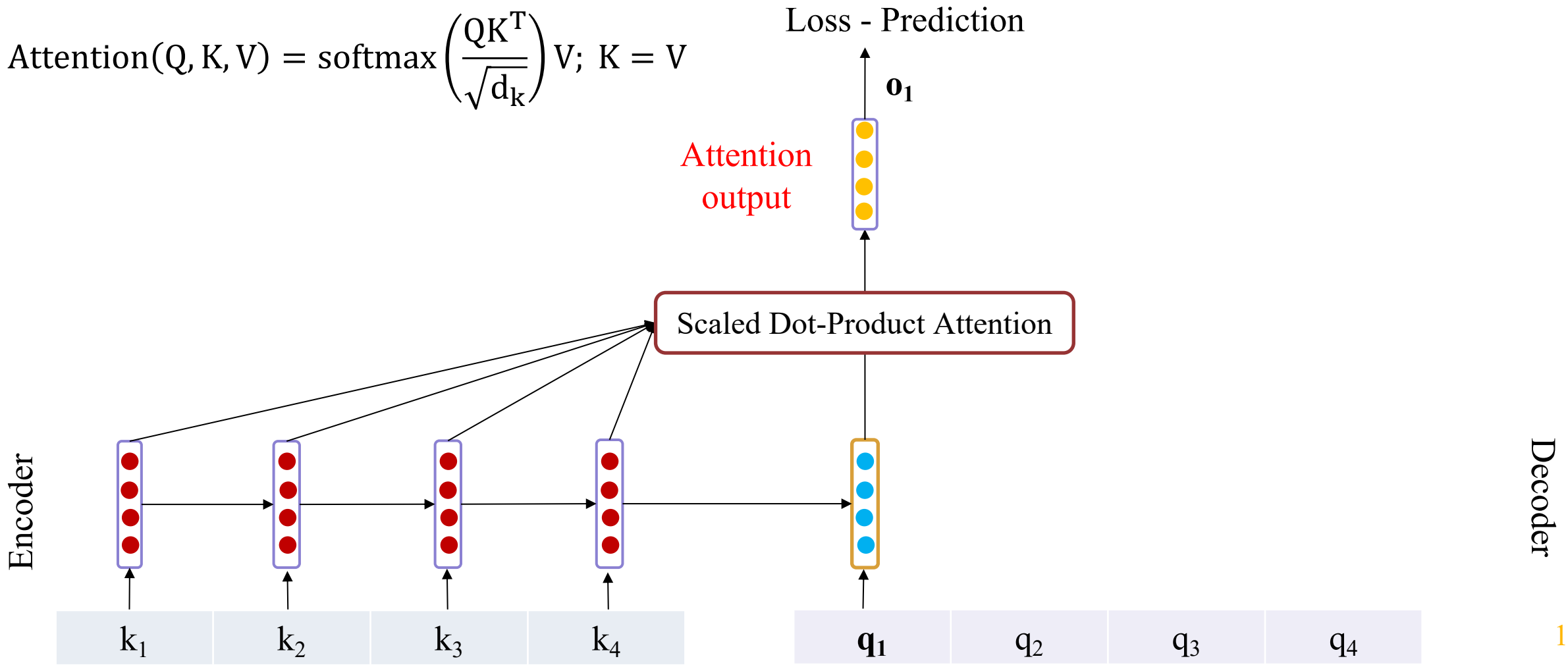
$$\alpha^t = \text{softmax}(e^t)$$

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

1 – Sequence-to-Sequence

Scaled Dot-Product Attention

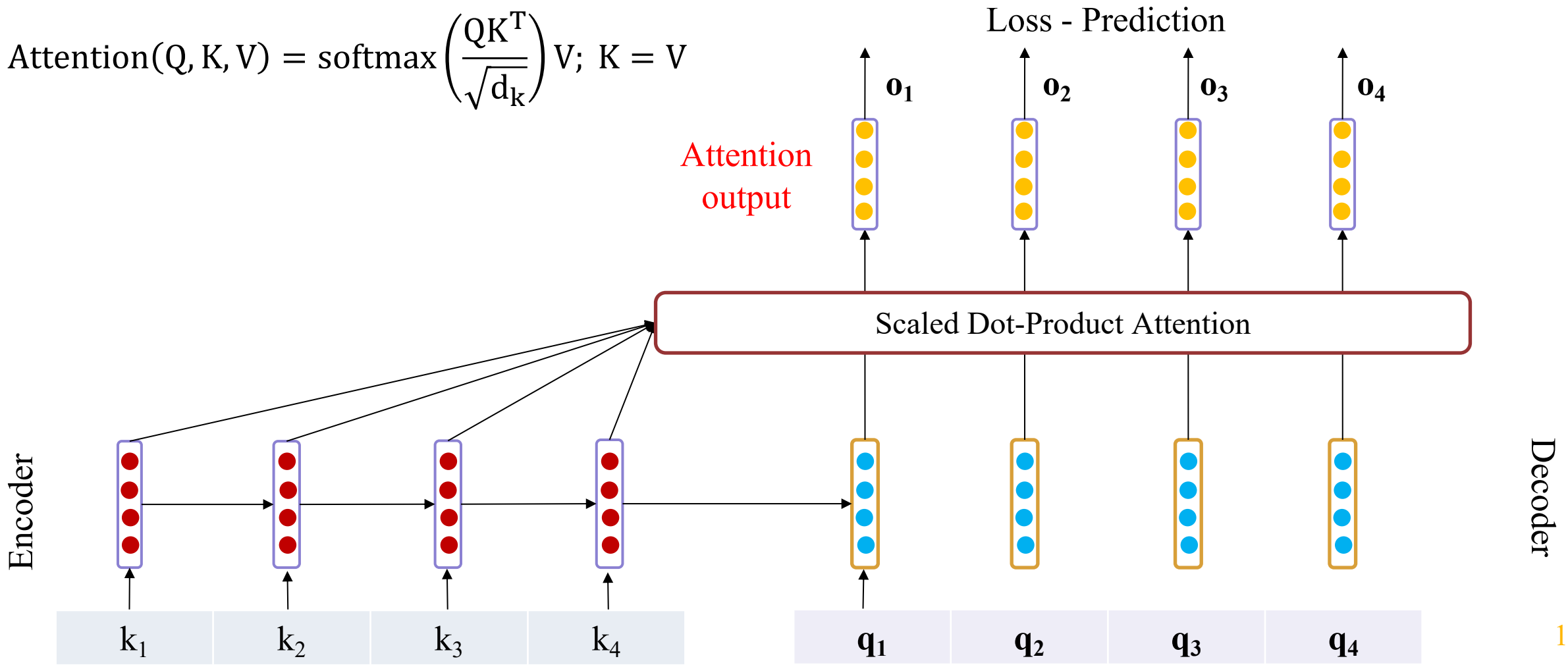
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V;\ K = V$$



1 – Sequence-to-Sequence

Scaled Dot-Product Attention

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



1 – Sequence-to-Sequence

Attention Variants

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

Multiplicative Attention

$$\text{softmax}(QWK^T)V$$

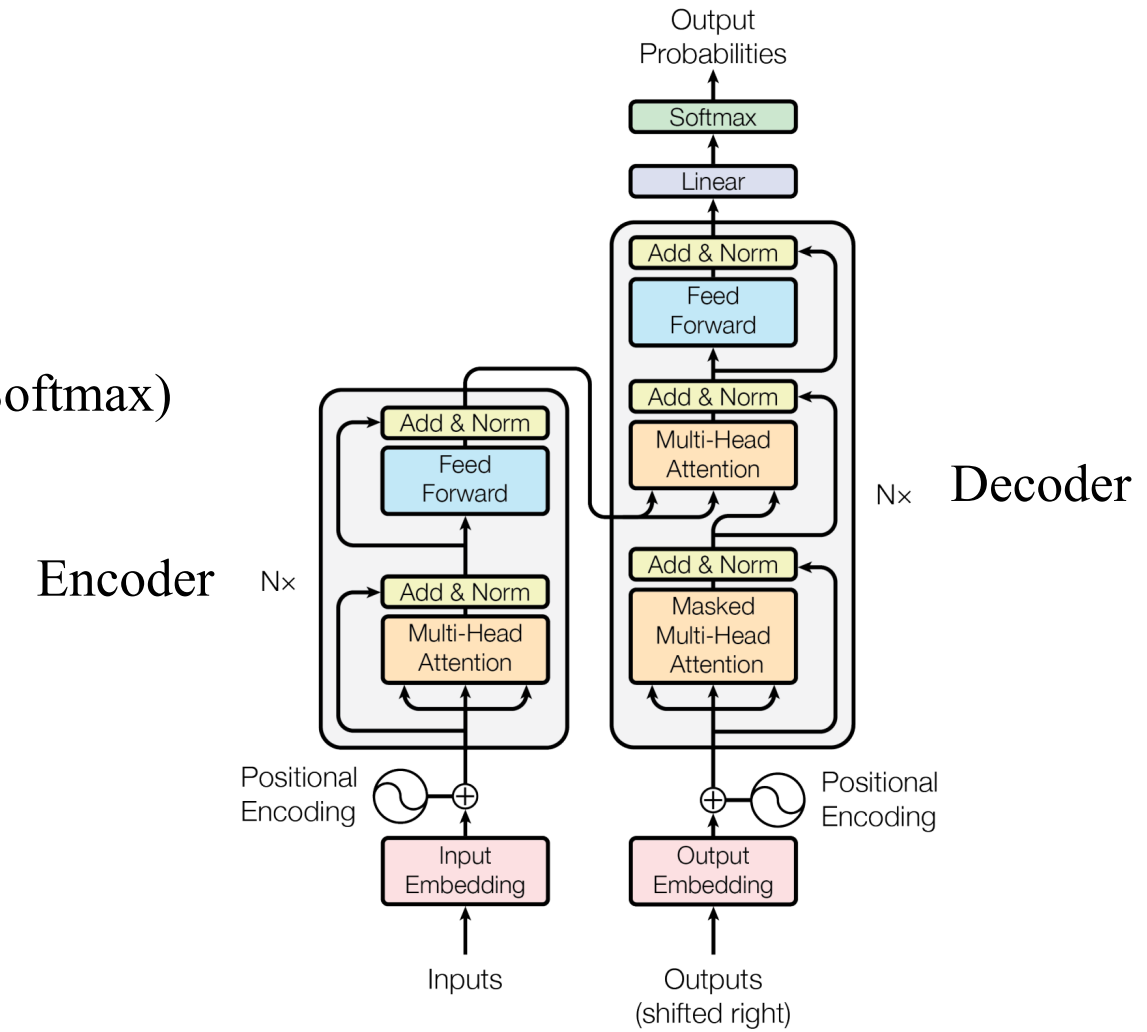
Additive Attention

$$\tanh(W_1 Q + W_2 K)\, V$$

2 – Transformer

Transformer

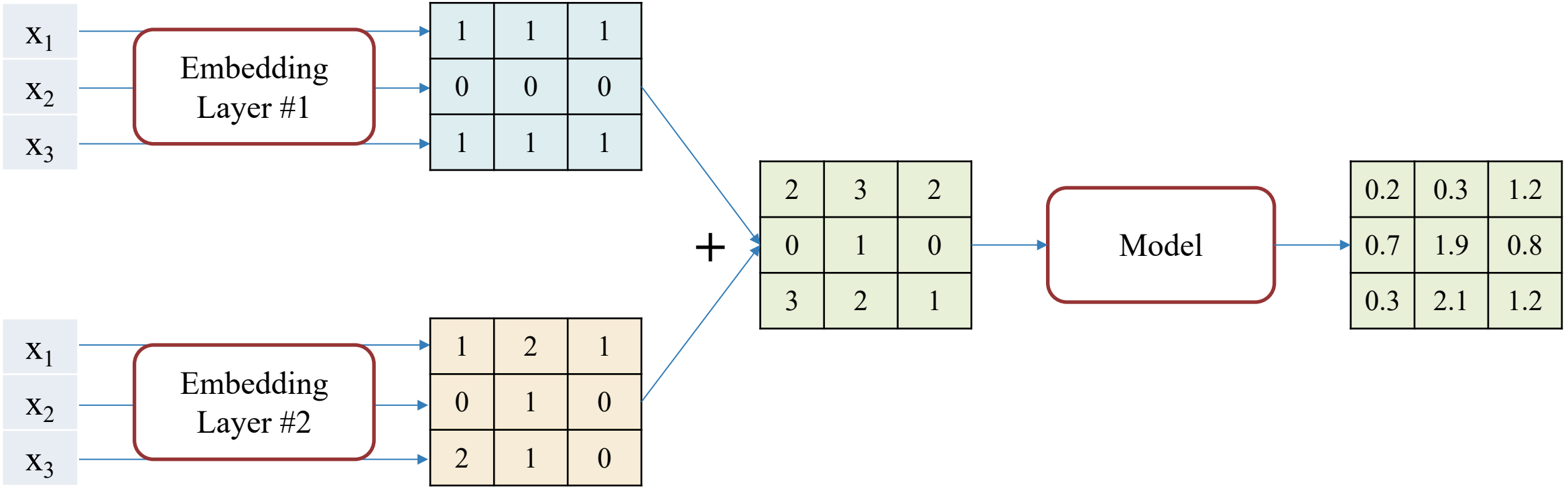
- Architecture:
  - Input Embedding
  - Positional Encoding
  - N Encoder Layer
  - N Decoder Layer
  - Language Model Head (Projection + Softmax)
- Core technique: attention
- Loss function: cross-entropy



2 – Transformer

Input Embedding - Positional Encoding

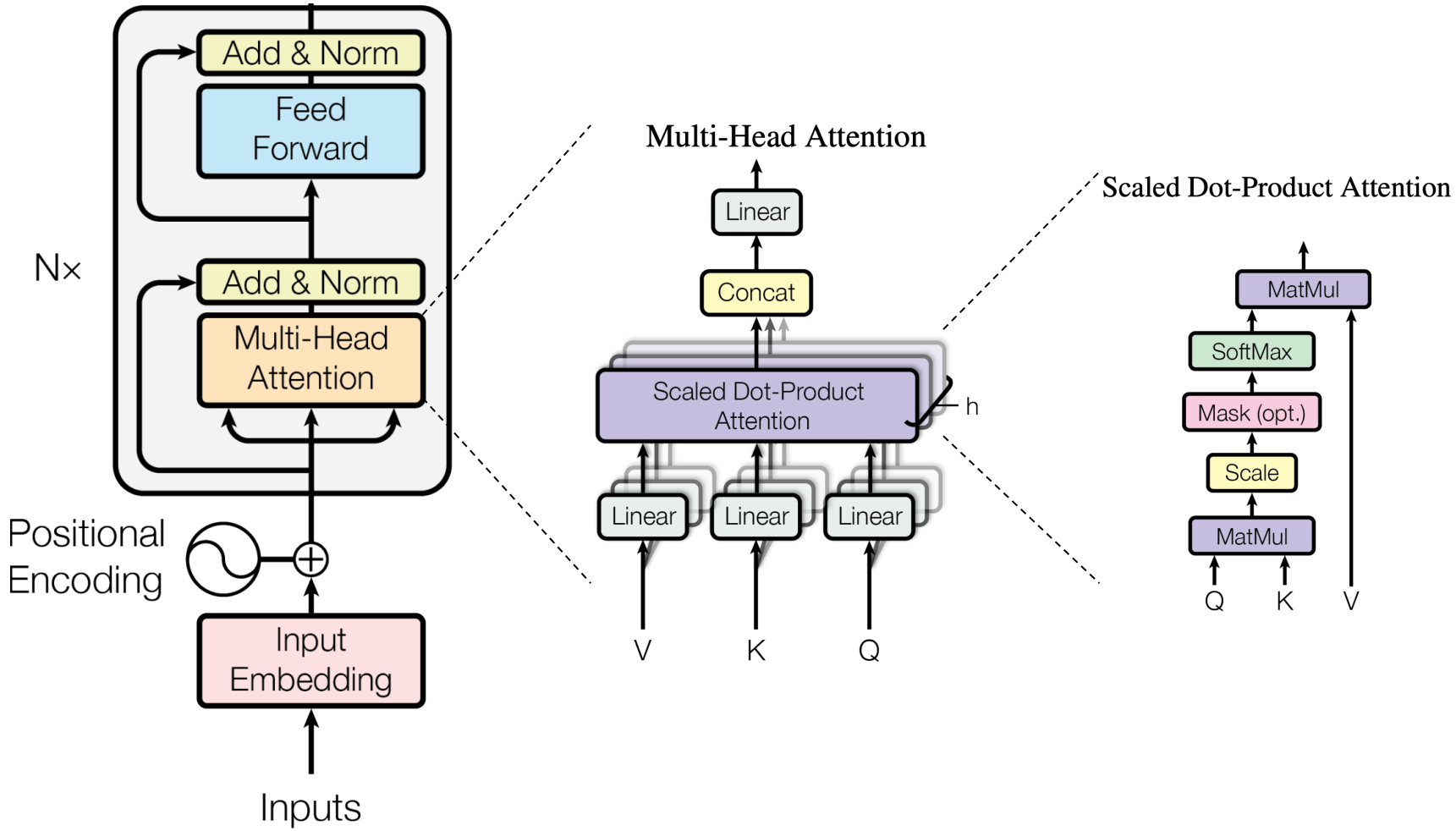
- The position of a token in a sentence as unique representation – each position is mapped to a vector
- Methods: Sinusoid; Learned positional embedding (as learned input embedding)



2 – Transformer

Transformer-Encoder

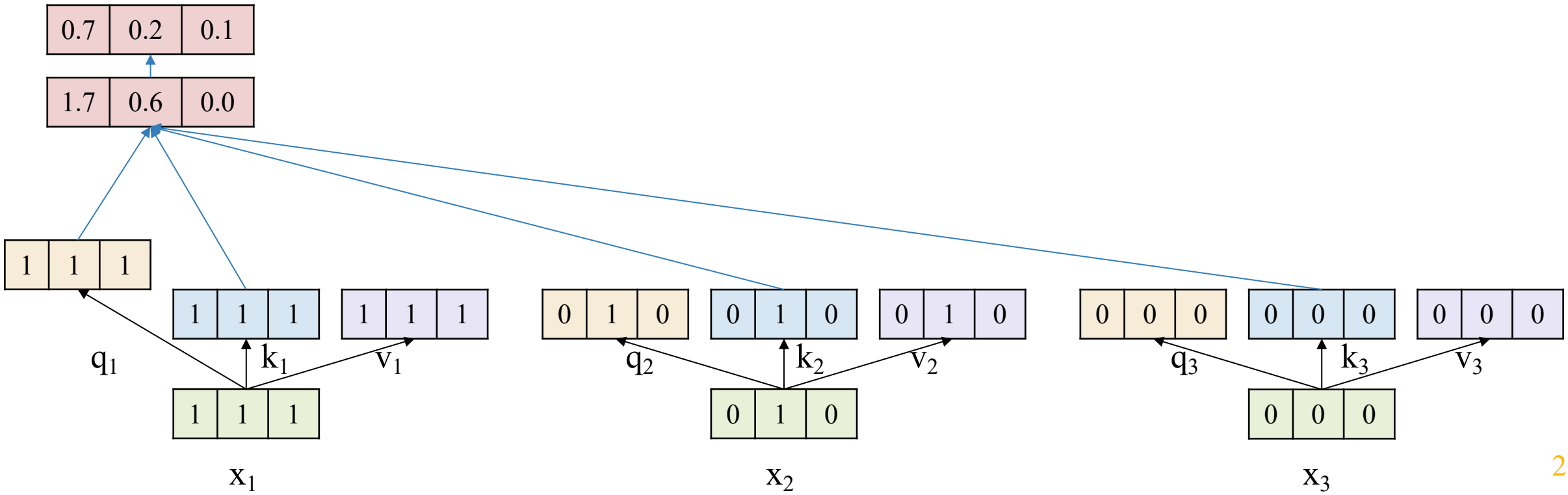
- Multi-Head Attention
- Feed Forward
- Add & Norm



2 – Transformer

Transformer-Encoder

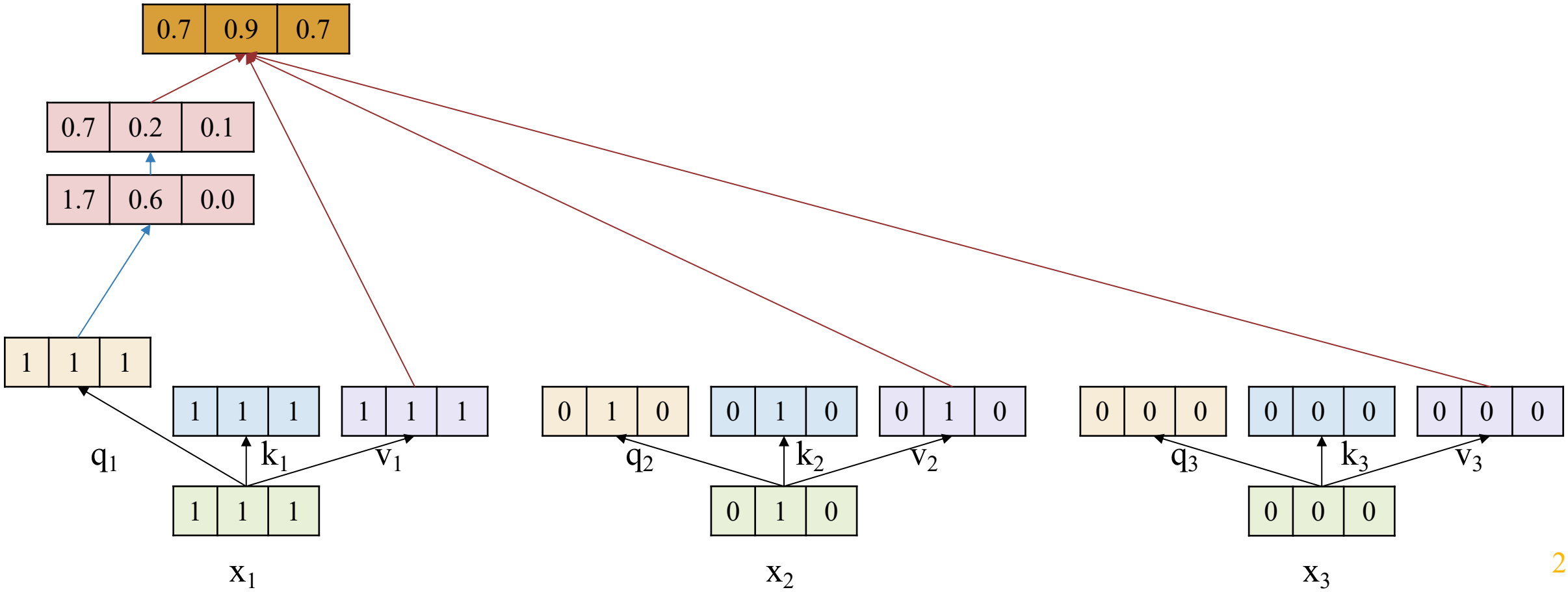
❖ Self-Attention



2 – Transformer

Transformer-Encoder

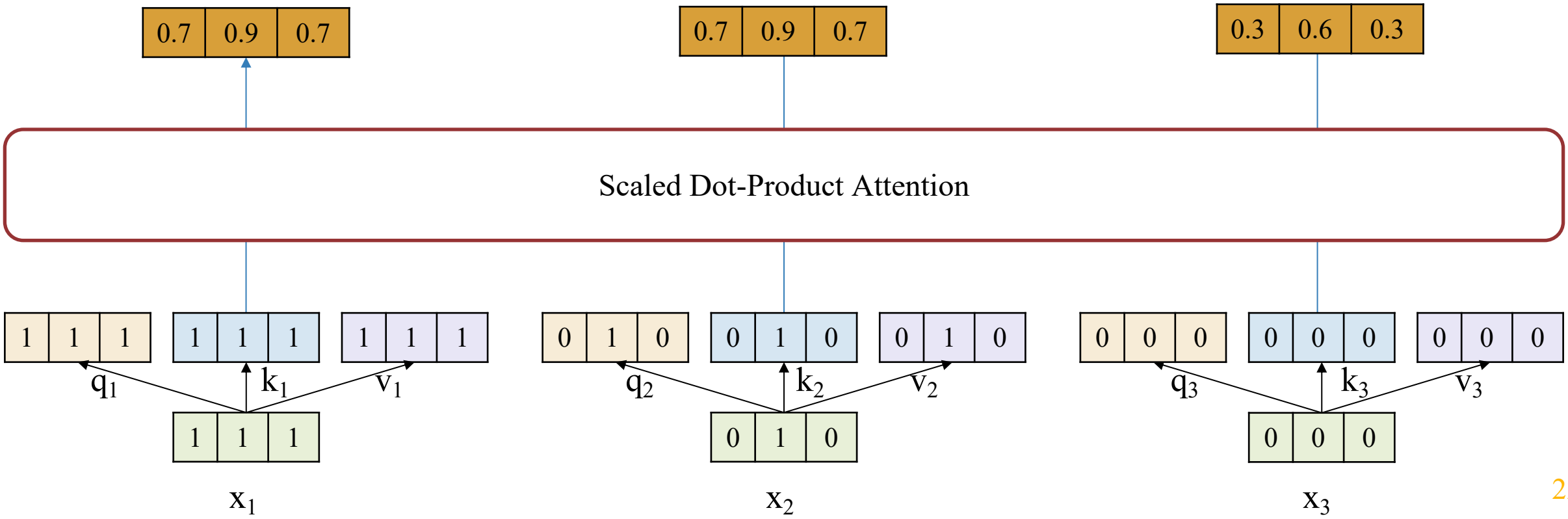
Self-Attention



2 – Transformer

Transformer-Encoder

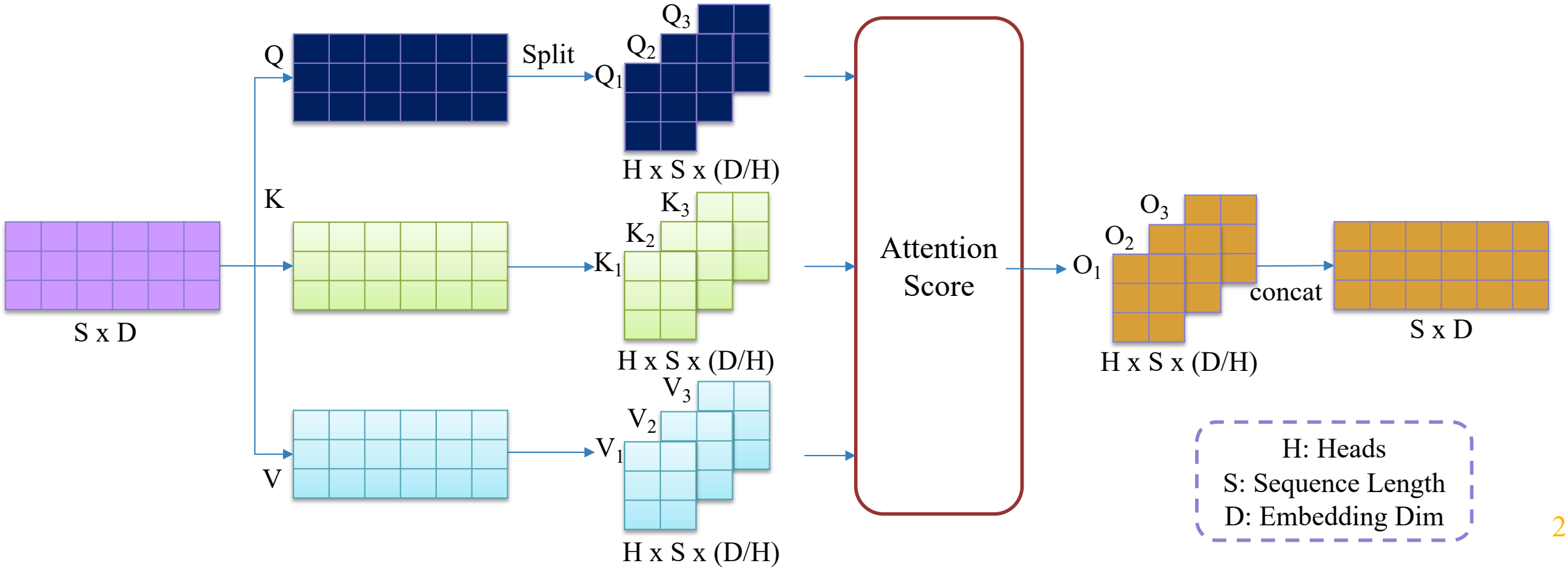
❖ Self-Attention



2 – Transformer

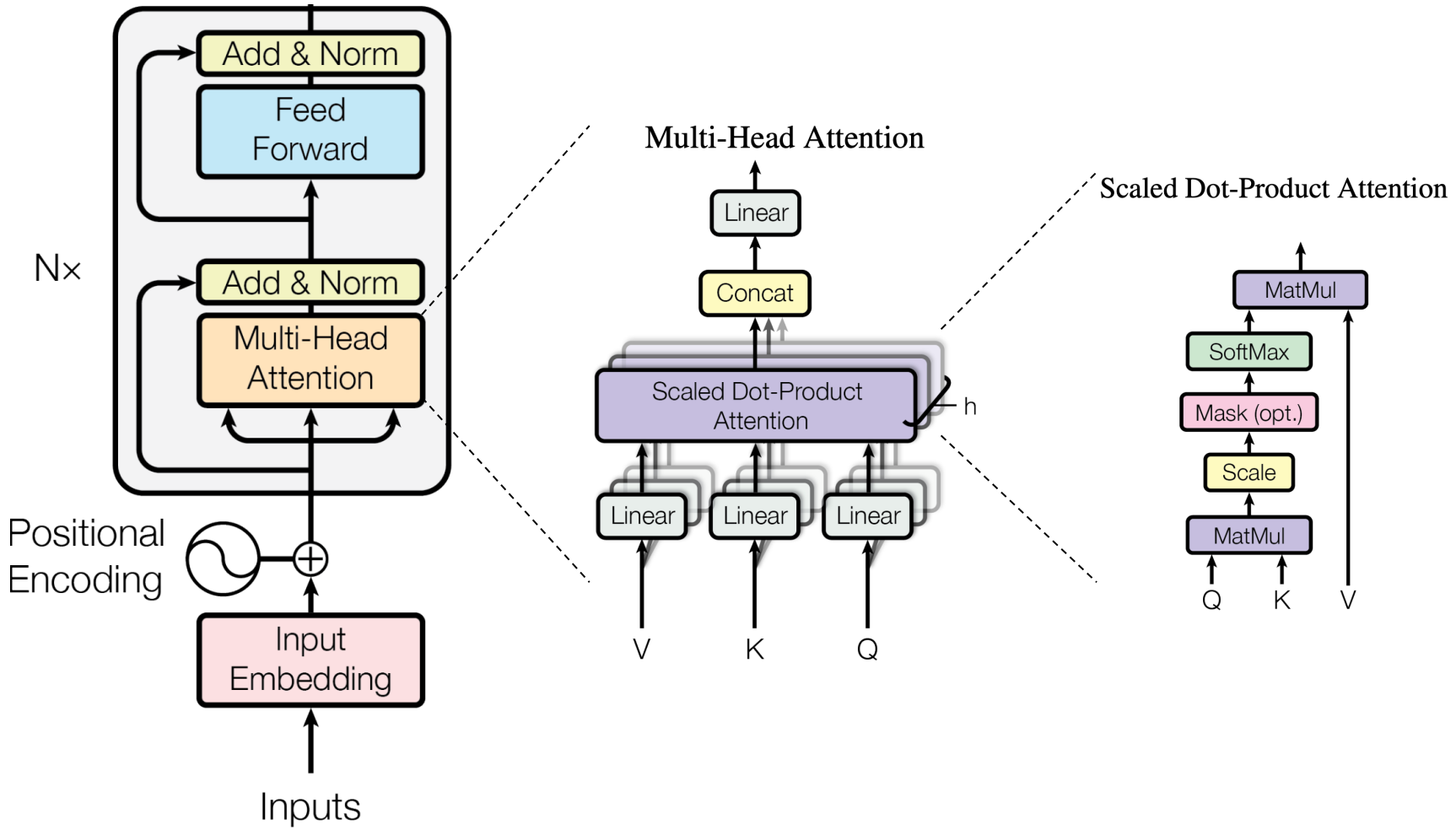
Transformer-Encoder

- **Multi-Head Attention**
- Split into the multiple attention heads (process independently) => self-attention => concat



2 – Transformer

Transformer-Encoder



2 – Transformer

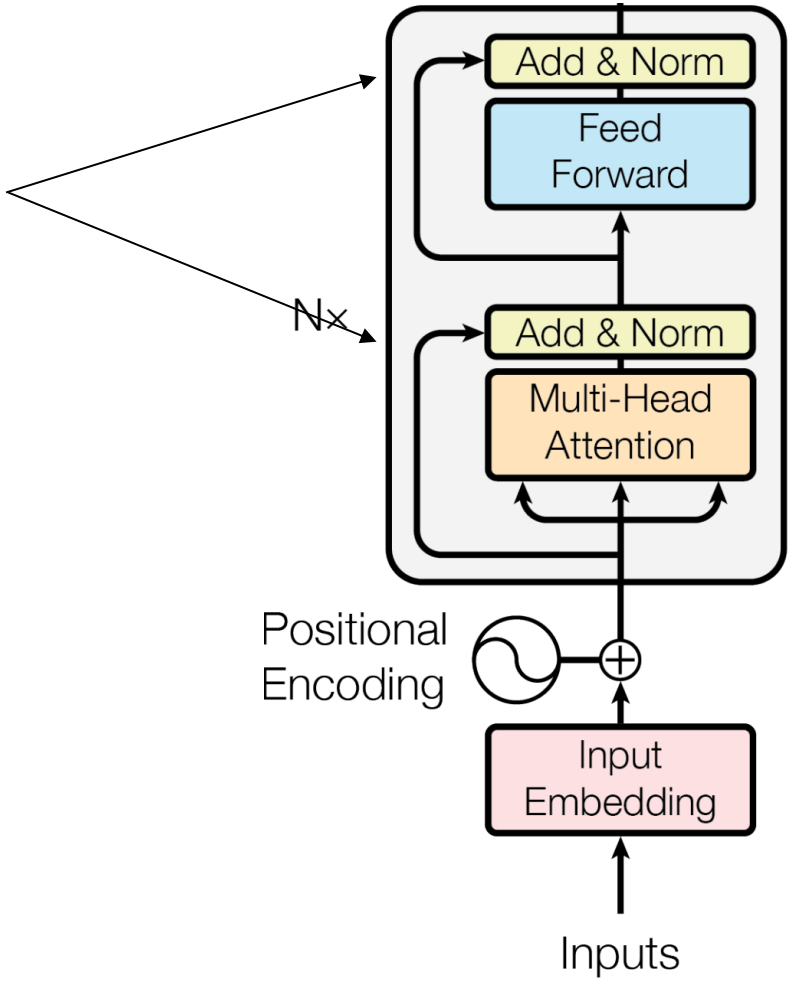
Transformer-Encoder

Layer Normalization

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

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

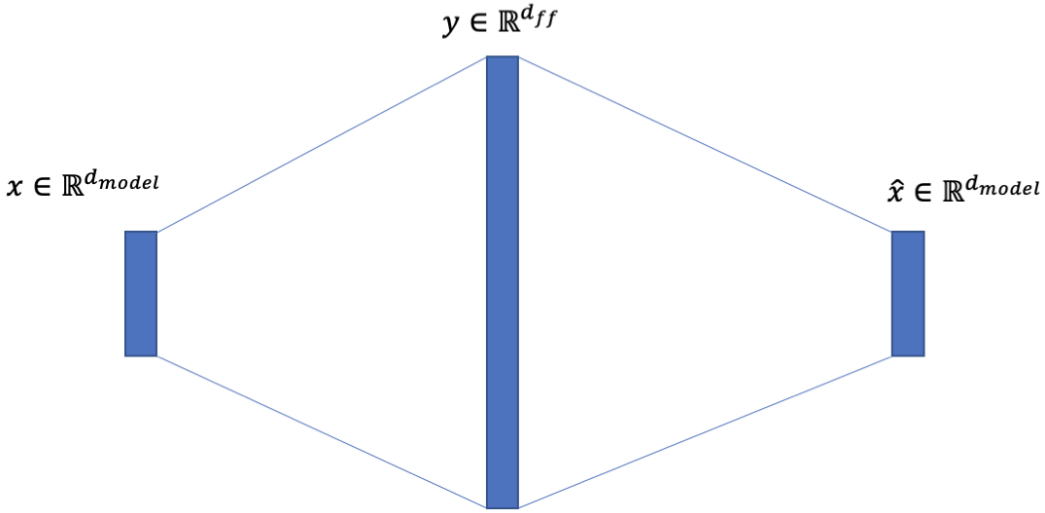
$$\hat{x}_{ij} = \frac{(x_{ij} - \mu_i)}{\sqrt{\sigma_i^2 + \epsilon}}$$

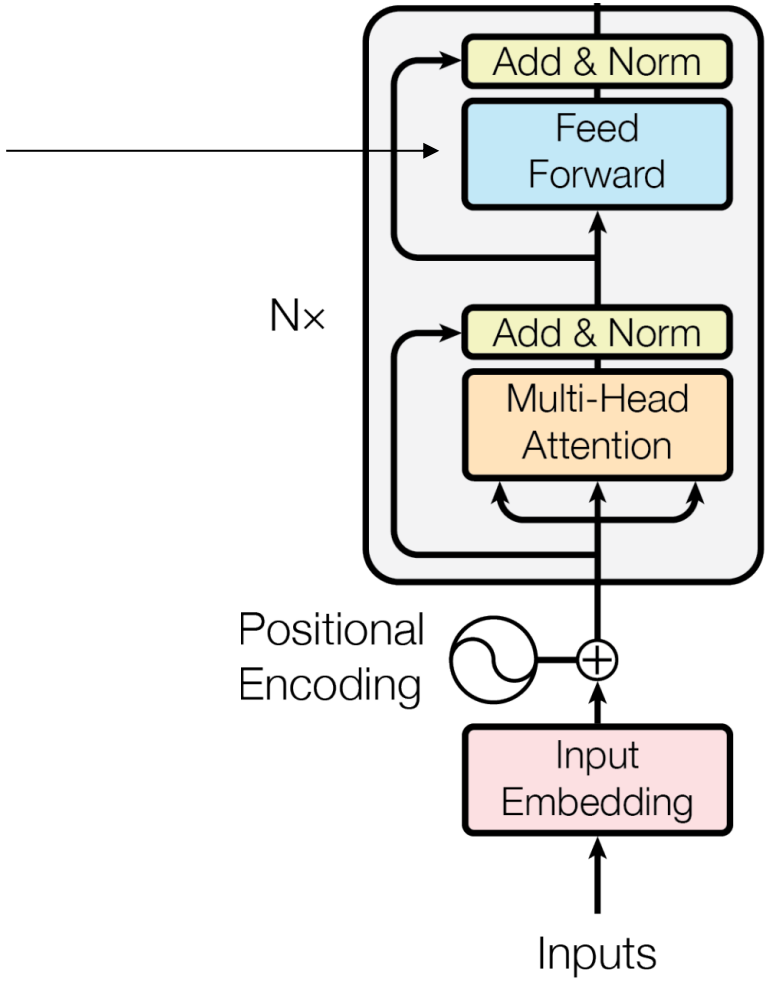


2 – Transformer

Transformer-Encoder

- **Feed Forward**
- 2 FC Layer





2 – Transformer

Transformer-Encoder – Demo

```python
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
```

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

```python
embedded.shape
```

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

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

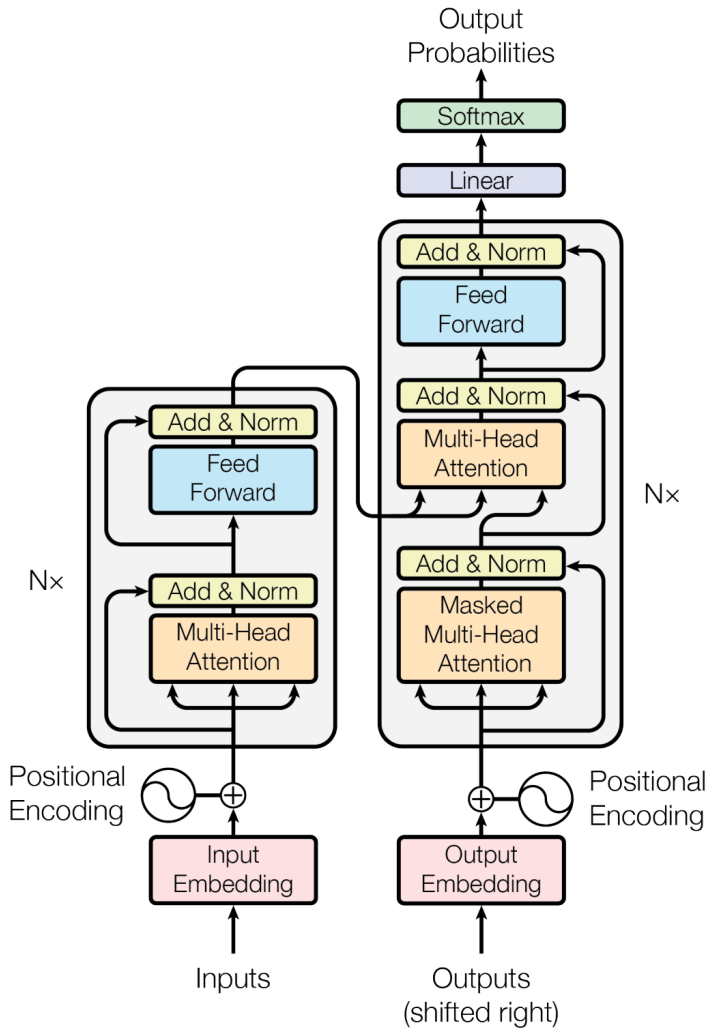
```python
encoded.shape
```

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

2 – Transformer

Transformer-Decoder

- Masked Multi-Head Attention
- Multi-Head Attention
- Layer Normalization
- Feed Forward



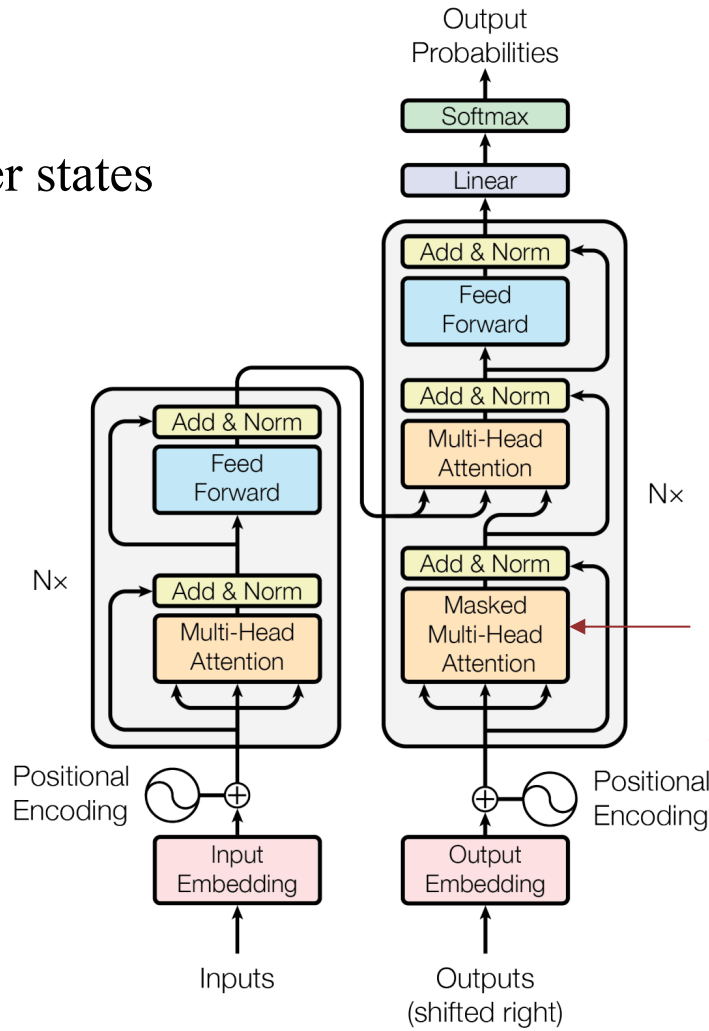
2 – Transformer

Transformer-Decoder

- **Masked Multi-Head Attention**
- Predict a next token based on the history token
- Queries, keys, values are computed from decoder states

Not Seen

	I	am	learn	#ing	nlp
I	1.2	2.3	2.5	1.2	3.5
am	2.3	1.5	0.5	0.6	1.2
learn	0.5	1.4	1.6	0.3	4.8
#ing	0.6	1.8	2.4	0.3	1.2
nlp	2.1	2.3	0.2	2.0	2.5



Decoder self-attention (masked)
Tokens look at the previous tokens

2 – Transformer

Transformer-Decoder

- **Masked Multi-Head Attention**
- Predict a next token based on the history token
- Queries, keys, values are computed from decoder states

Not Seen

	I	am	learn	#ing	nlp
I	1.2	2.3	2.5	1.2	3.5
am	2.3	1.5	0.5	0.6	1.2
learn	0.5	1.4	1.6	0.3	4.8
#ing	0.6	1.8	2.4	0.3	1.2
nlp	2.1	2.3	0.2	2.0	2.5

1.2	-inf	-inf	-inf	-inf
2.3	1.5	-inf	-inf	-inf
0.5	1.4	1.6	-inf	-inf
0.6	1.8	2.4	0.3	-inf
2.1	2.3	0.2	2.0	2.5

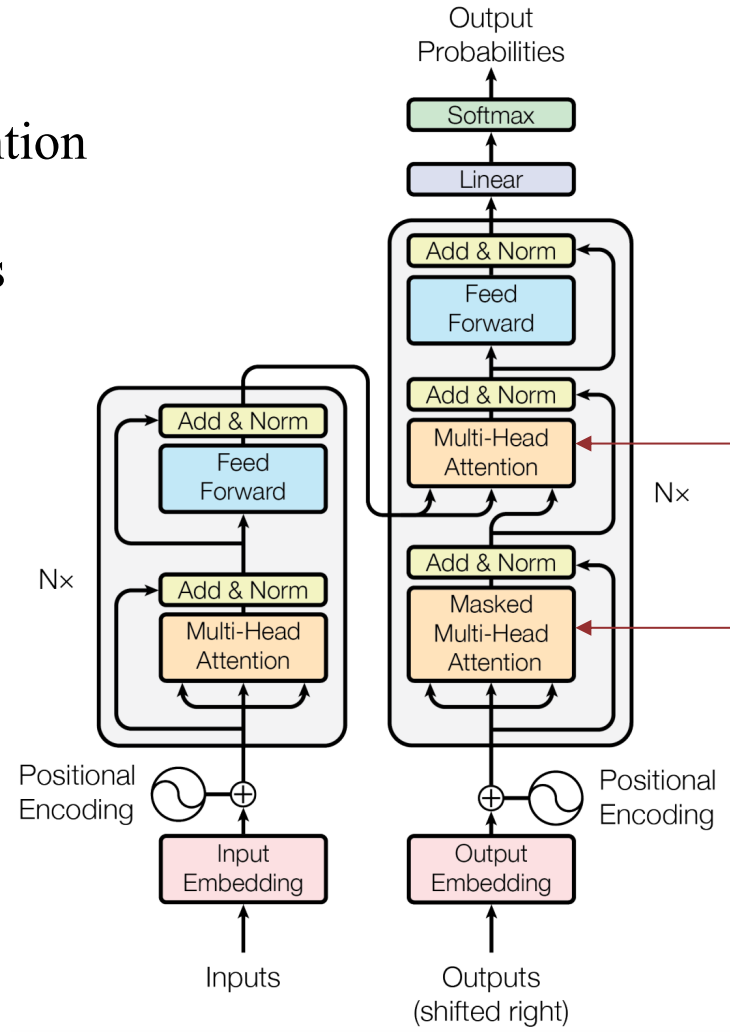
Softmax

1.0	0.0	0.0	0.0	0.0
0.69	0.31	0.0	0.0	0.0
0.15	0.38	0.46	0.0	0.0
0.6	1.8	2.4	0.3	0.0
2.1	2.3	0.2	2.0	2.5

2 – Transformer

Transformer-Decoder

- **Multi-Head Attention**
- Sequence-to-sequence with attention
- Queries – from decoder states
- Keys, values from encoder states



Decoder – encoder attention
Target token look at the source

Decoder self-attention (masked)
Tokens look at the previous tokens

2 – Transformer

Transformer-Decoder – Demo

```python
class TransformerDecoderBlock(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.cross_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.layernorm_3 = nn.LayerNorm(normalized_shape=embed_dim, eps=1e-6)
        self.dropout_1 = nn.Dropout(p=dropout)
        self.dropout_2 = nn.Dropout(p=dropout)
        self.dropout_3 = nn.Dropout(p=dropout)
```

```python
    def forward(self, x, enc_output, src_mask, tgt_mask):
        attn_output, _ = self.attn(x, x, x, attn_mask=tgt_mask)
        attn_output = self.dropout_1(attn_output)
        out_1 = self.layernorm_1(x + attn_output)

        attn_output, _ = self.cross_attn(
            out_1, enc_output, enc_output, attn_mask=src_mask
        )
        attn_output = self.dropout_2(attn_output)
        out_2 = self.layernorm_2(out_1 + attn_output)

        ffn_output = self.ffn(out_2)
        ffn_output = self.dropout_2(ffn_output)
        out_3 = self.layernorm_2(out_2 + ffn_output)
        return out_3
```

2 – Transformer

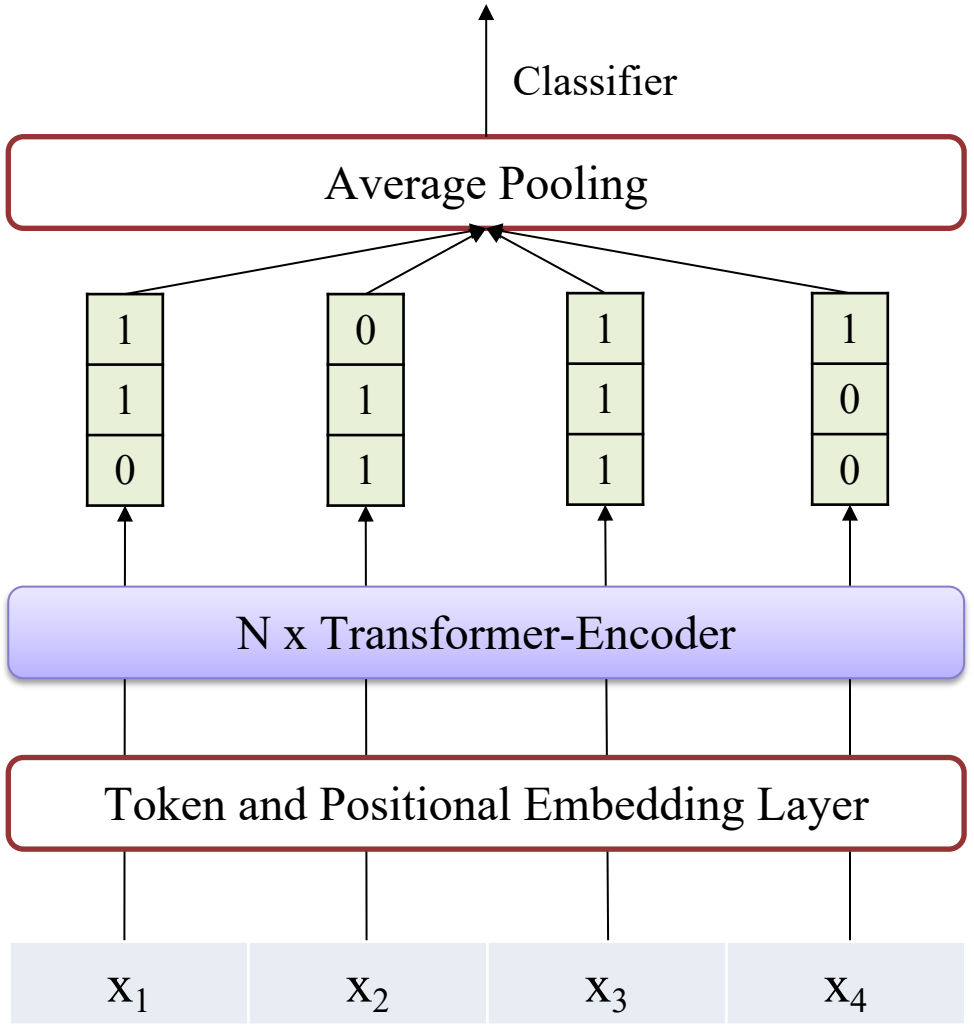
Transformer – Demo

```python
class Transformer(nn.Module):
    def __init__(self,
            src_vocab_size, tgt_vocab_size,
            embed_dim, max_length, num_layers, num_heads, ff_dim,
            dropout=0.1, device='cpu'
        ):
        super().__init__()
        self.device = device
        self.encoder = TransformerEncoder(
            src_vocab_size, embed_dim, max_length, num_layers, num_heads, ff_dim
        )
        self.decoder = TransformerDecoder(
            tgt_vocab_size, embed_dim, max_length, num_layers, num_heads, ff_dim
        )
        self.fc = nn.Linear(embed_dim, tgt_vocab_size)
```

```python
def generate_mask(self, src, tgt):
    src_seq_len = src.shape[1]
    tgt_seq_len = tgt.shape[1]

    src_mask = torch.zeros(
        (src_seq_len, src_seq_len),
        device=self.device).type(torch.bool)

    tgt_mask = (torch.triu(torch.ones(
        (tgt_seq_len, tgt_seq_len),
        device=self.device)
    ) == 1).transpose(0, 1)
    tgt_mask = tgt_mask.float().masked_fill(
        tgt_mask == 0, float('-inf')).masked_fill(tgt_mask == 1, float(0.0))
    return src_mask, tgt_mask

def forward(self, src, tgt):
    src_mask, tgt_mask = self.generate_mask(src, tgt)
    enc_output = self.encoder(src)
    dec_output = self.decoder(tgt, enc_output, src_mask, tgt_mask)
    output = self.fc(dec_output)
    return output
```

2 – Transformer

Text Classification using Transformer-Encoder – Demo



2 – Transformer

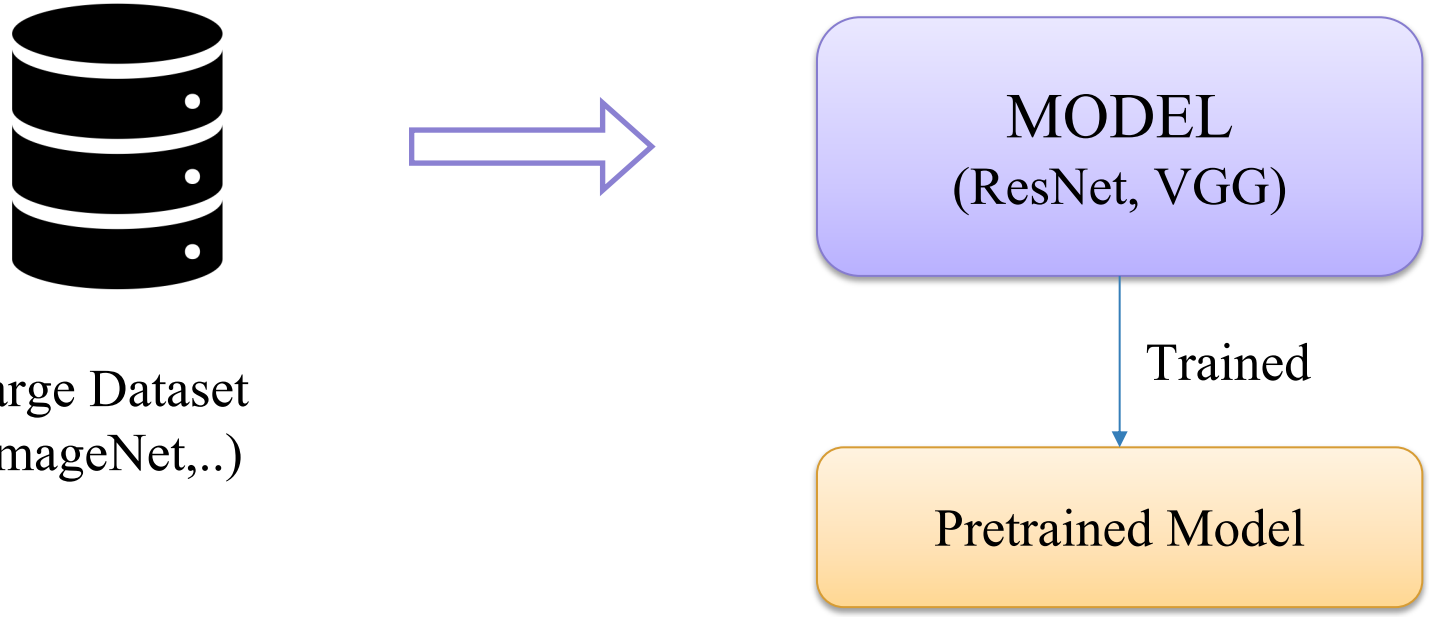
Text Classification using Transformer-Encoder

```python
class TransformerEncoderCls(nn.Module):
    def __init__(self,
                 vocab_size, max_length, num_layers, embed_dim, num_heads, ff_dim,
                 dropout=0.1, device='cpu'
        ):
        super().__init__()
        self.encoder = TransformerEncoder(
            vocab_size, embed_dim, max_length, num_layers, num_heads, ff_dim, dropout, device
        )
        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=2)
        self.dropout = nn.Dropout(p=dropout)
        self.relu = nn.ReLU()
    def forward(self, x):
        output = self.encoder(x)
        output = self.pooling(output.permute(0,2,1)).squeeze()
        output = self.dropout(output)
        output = self.fc1(output)
        output = self.dropout(output)
        output = self.fc2(output)
        return output
```

3 – BERT

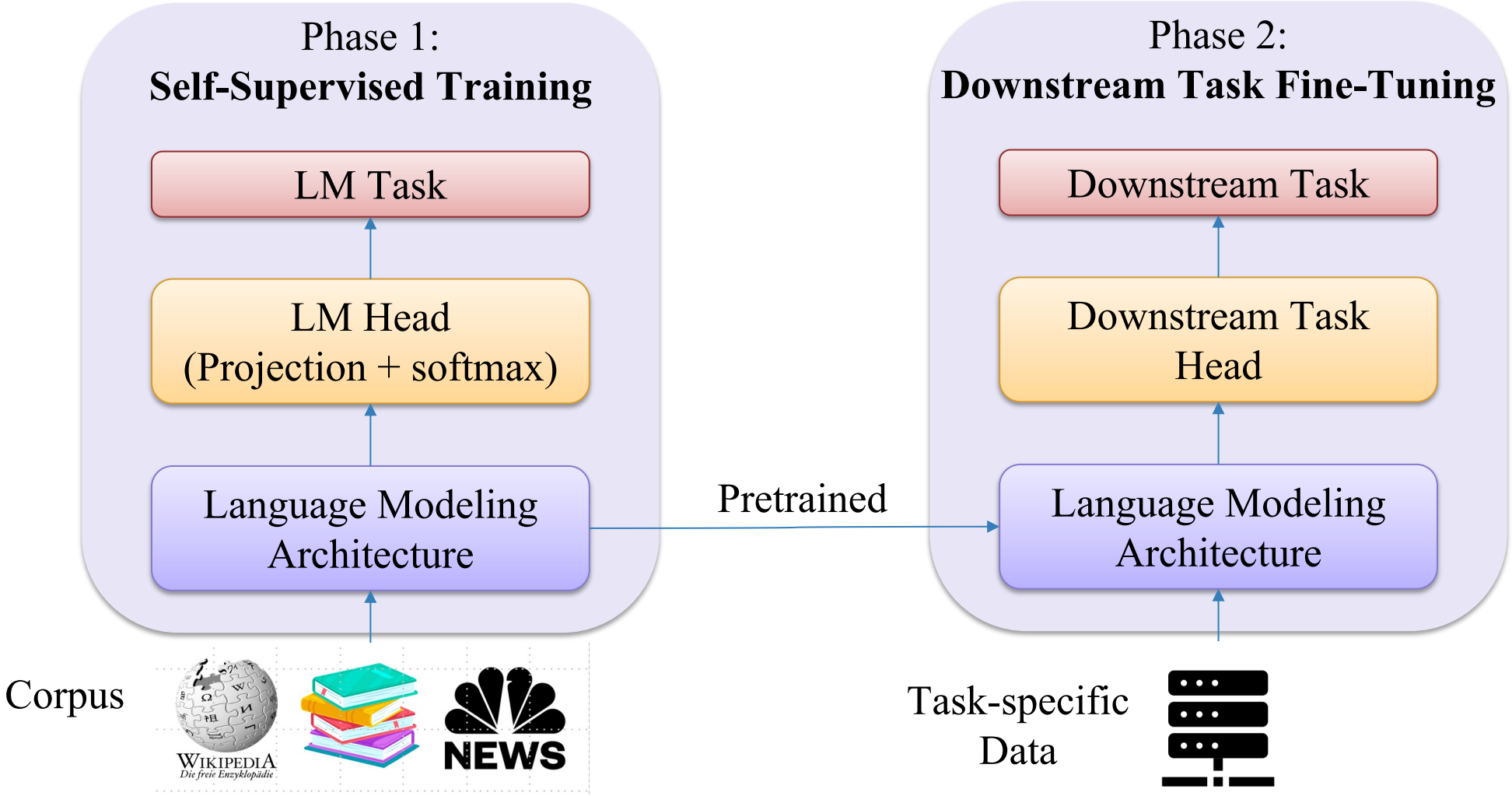
Pretrained Model for Image

- **ImageNet**
- Training: 1,281,167 images. Validation: 50,000 images. Testing: 100,000 images
- Object classes: 1,000



3 – BERT

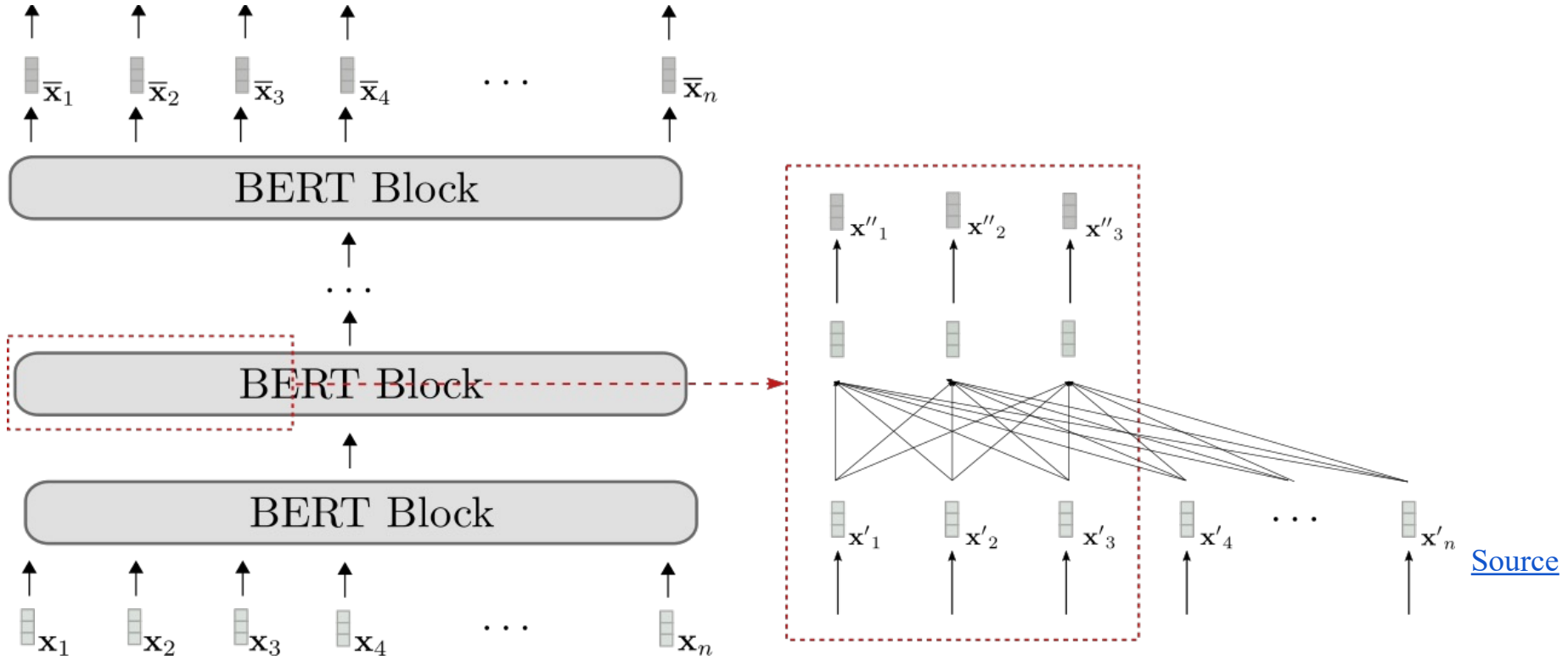
Pretrained Model for Text



3 – BERT

BERT

- BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
- BERT: An encoder-only model
- Maps an input sequence to a contextualized sequence: $f_{\theta_{BERT}}: X_{1:n} \longrightarrow \overline{X}_{1:n}$

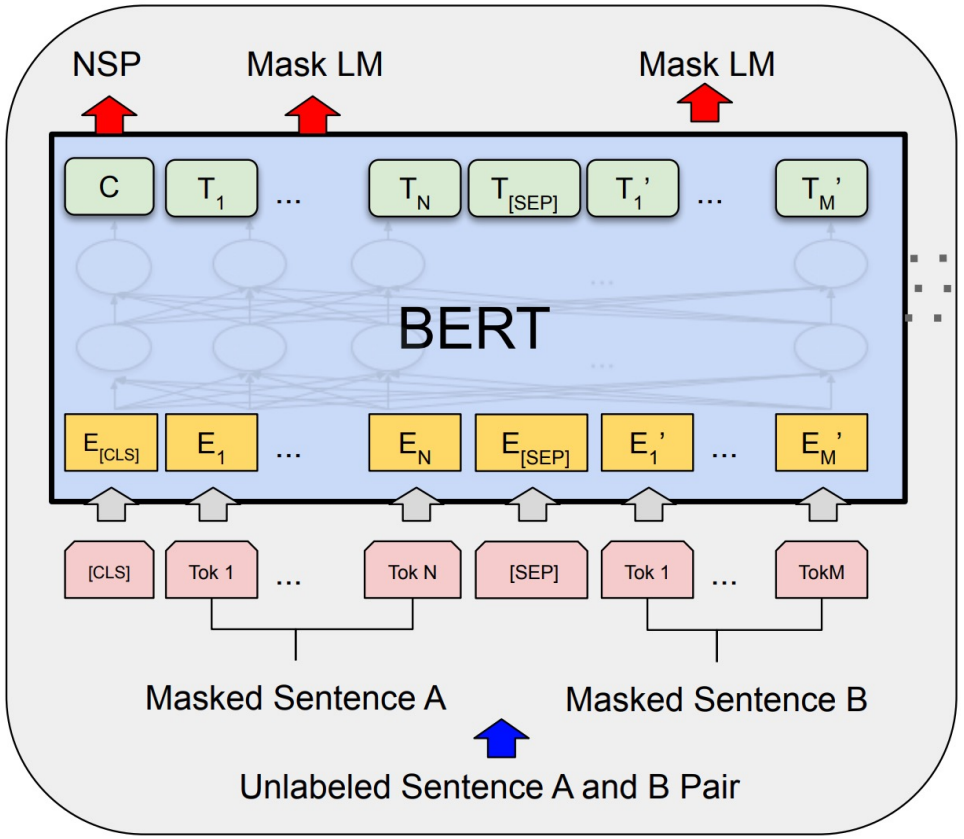


Source

3 – BERT

Objective Functions

- Masked LM (15% token):
  - 80%: replace with [MASK]
  - 10%: replace with a random word
  - 10%: keep unchanged
- Next Sentence Prediction (NSP)
  - Classification Task
  - 2 Labels: IsNext and NotNext
  - Use [SEP] [CLS] token



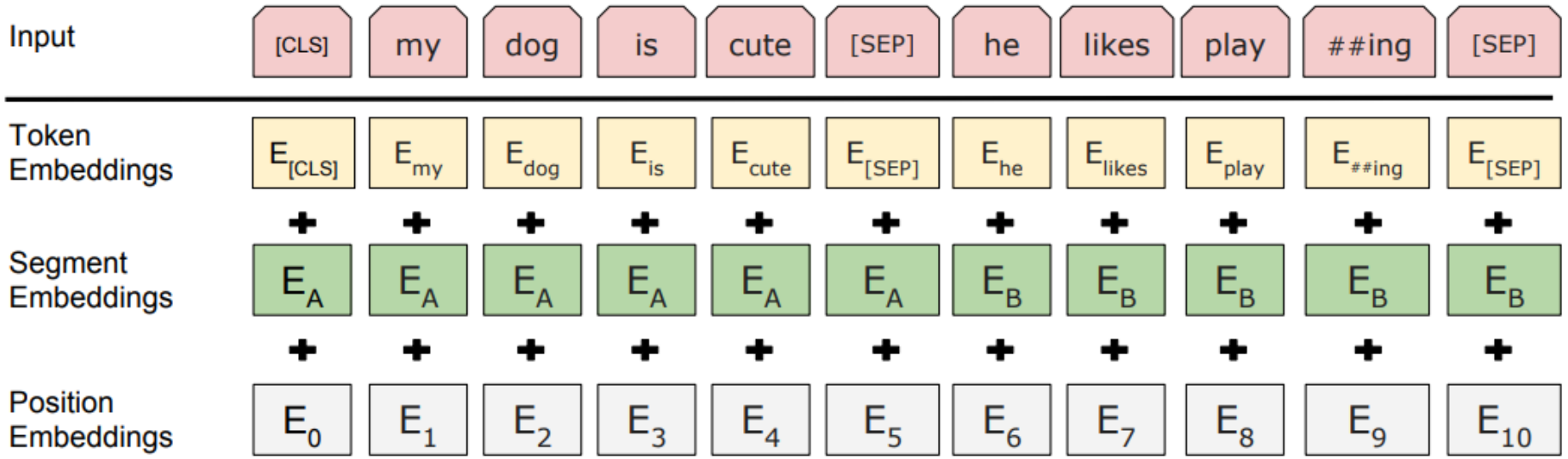
Pre-training

Source

3 – BERT

Input Representation

- BERT Input Representation



Source

3 – BERT

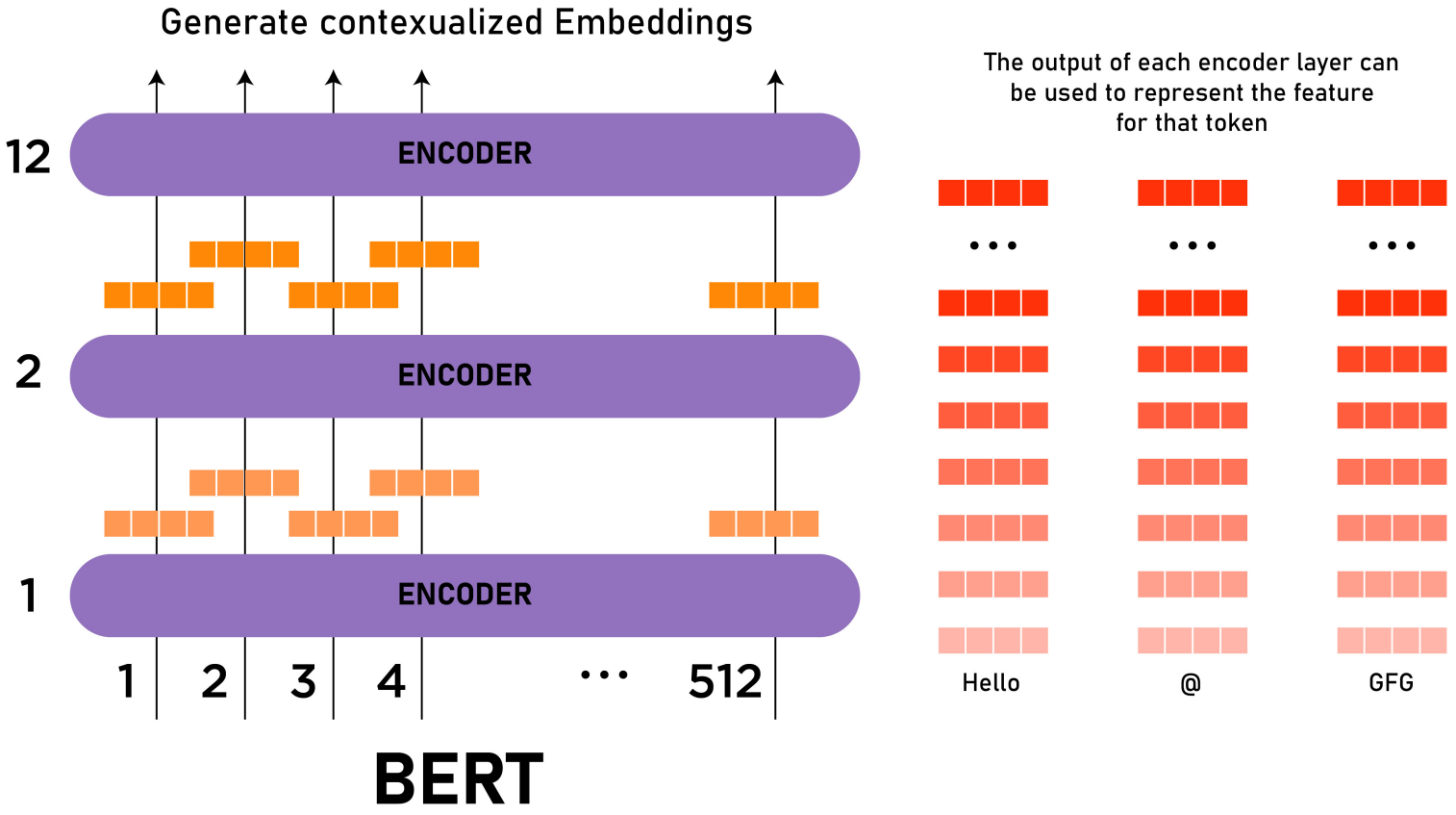
BERT Training

- BERT base:

L=12, H=768, A=12, P=110M

- BERT large:

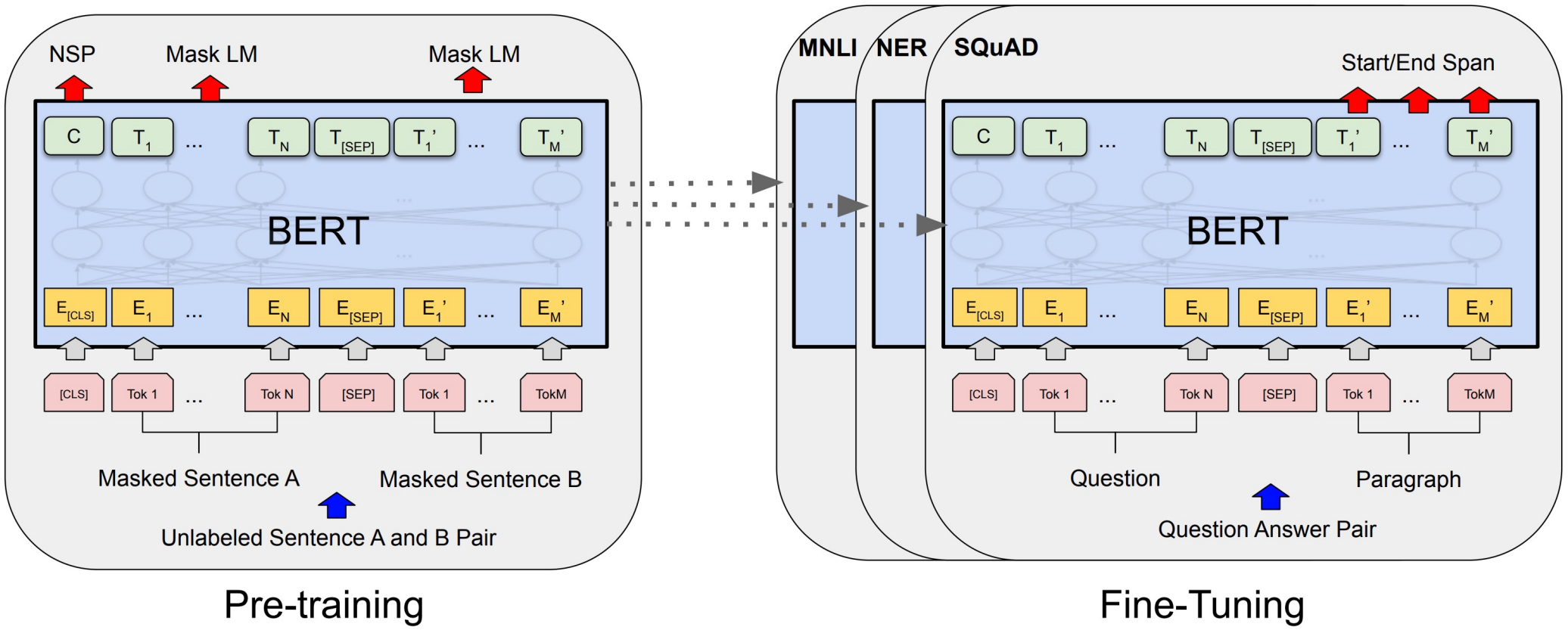
L=24, H=1024, A=16, P=340M



Source

3 – BERT

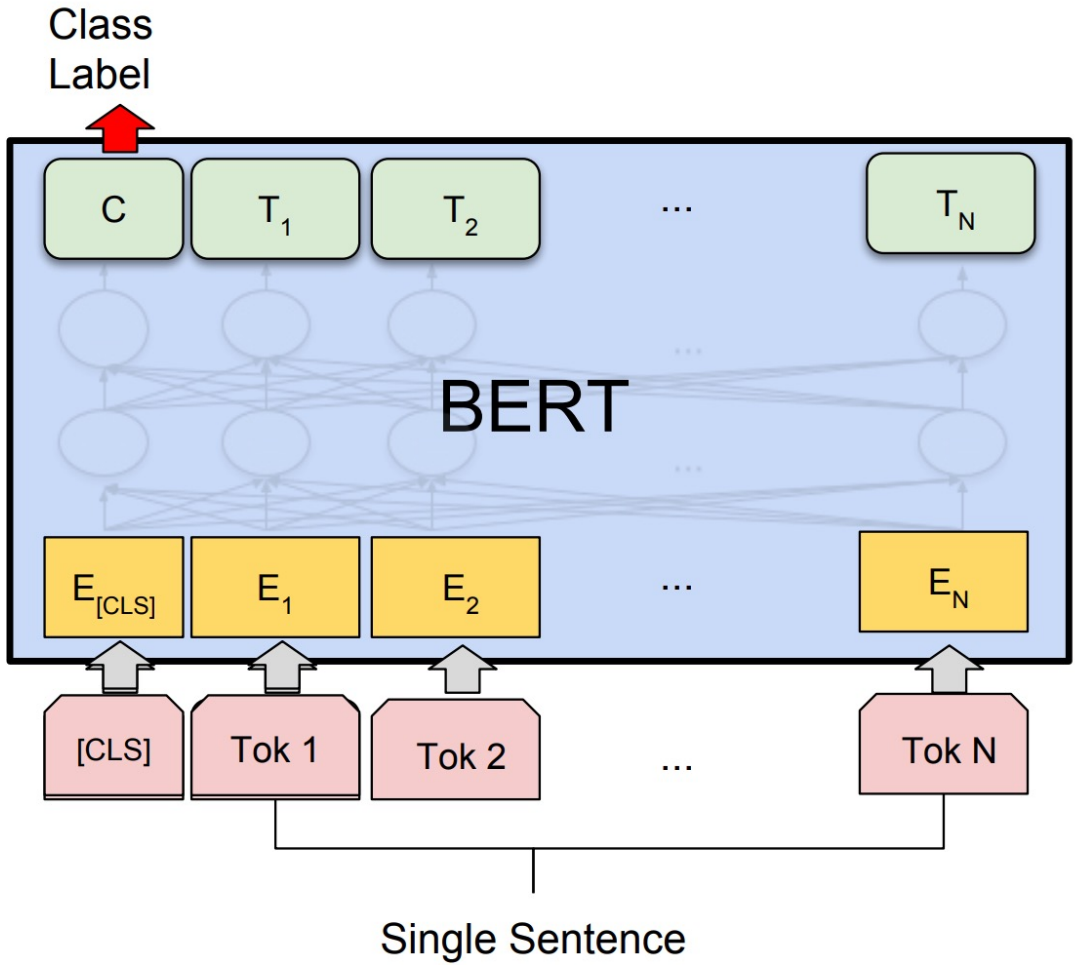
Fine-Tuning



Source

3 – BERT

Text Classification using BERT – Demo



Source

3 – BERT

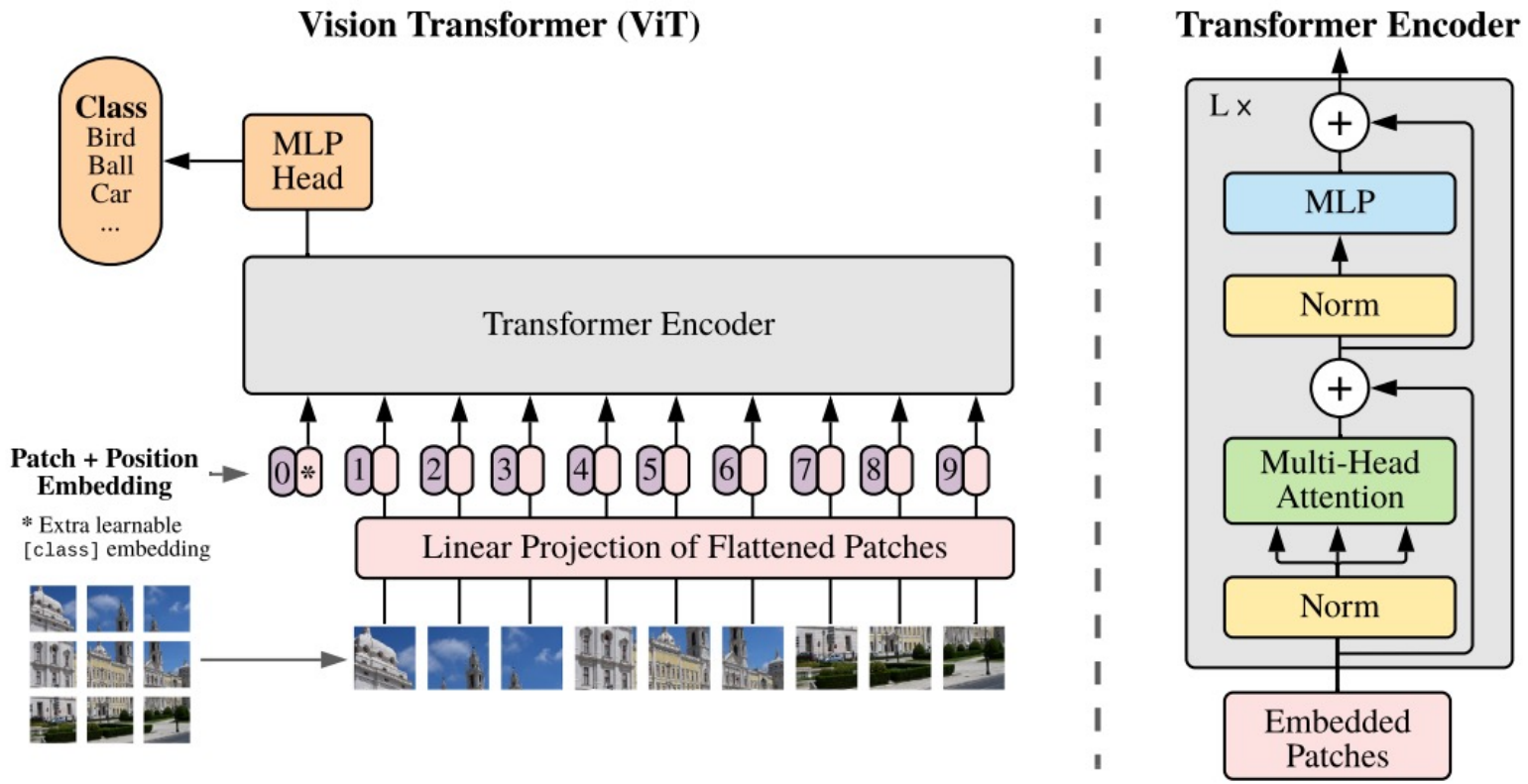
Text Classification using BERT

- NTC-SCV Dataset

Model	Accuracy
Transformer-Encoder	82.55
BERT	85.52
RNN	80.92
TextCNN	88.78

4 – Vision Transformer

Architecture



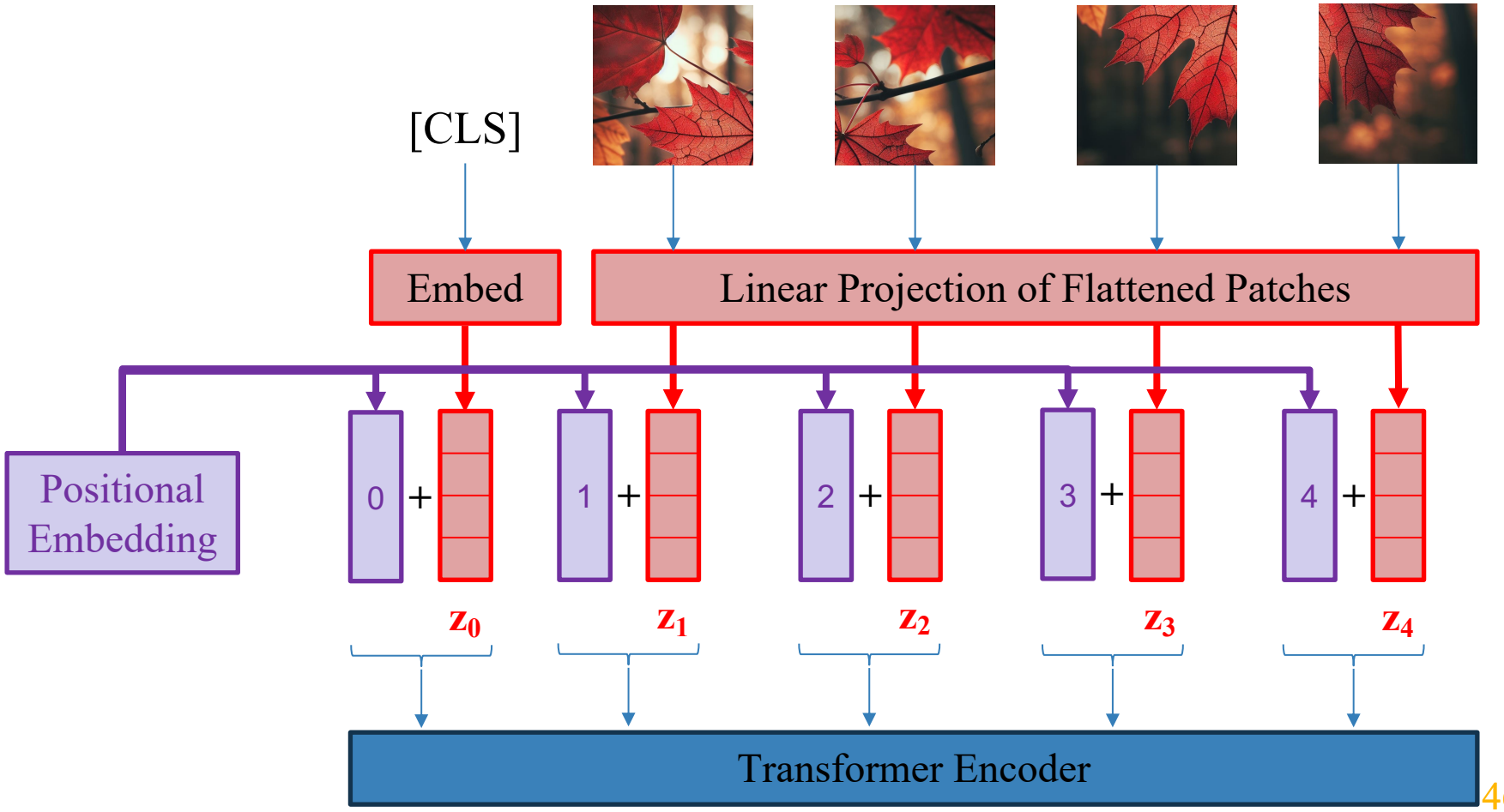
4 – Vision Transformer

Embedded Patches

Patches

Original

Vectorization



4 – Vision Transformer

Flower Dataset

4 – Vision Transformer

Implementation (Scratch) - Linear Projection – Demo

```python
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
```

4 – Vision Transformer

Implementation (Scratch) - Linear Projection – Demo

```python
class PatchPositionEmbedding(nn.Module):
    def __init__(self, image_size=224, embed_dim=512, patch_size=16, device='cpu'):
        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))
        self.device = device

    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(self.device)
        return x
```

add positional embedding to patch embedding

initialize positional embedding

4 – Vision Transformer

Implementation (Scratch) - Linear Projection – Demo

```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, device=device
        )
        self.transformer_layer = TransformerEncoder(
            embed_dim, num_heads, ff_dim, dropout
        )
        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

Implementation (Pretrained) – Demo

```
from transformers import ViTForImageClassification

id2label = {id:label for id, label in enumerate(classes)}
label2id = {label:id for id,label in id2label.items()}

model = ViTForImageClassification.from_pretrained(
    'google/vit-base-patch16-224-in21k',
    num_labels=num_classes,
    id2label=id2label,
    label2id=label2id
)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
```

4 – Vision Transformer

Pretrained - google/vit-base-patch16-224-in21k

Dataset	ImageNet-21k
Num of Images	14 millions
Num of Classes	21,000
Resized	224 x 224
Normalized Mean	(0.5, 0.5, 0.5)
Normalized Standard Deviation	(0.5, 0.5, 0.5)
Hardware	TPUv3 (8 cores)
Batch Size	4096

AI VIET NAM

@aivietnam.edu.vn

Thanks!

Any questions?