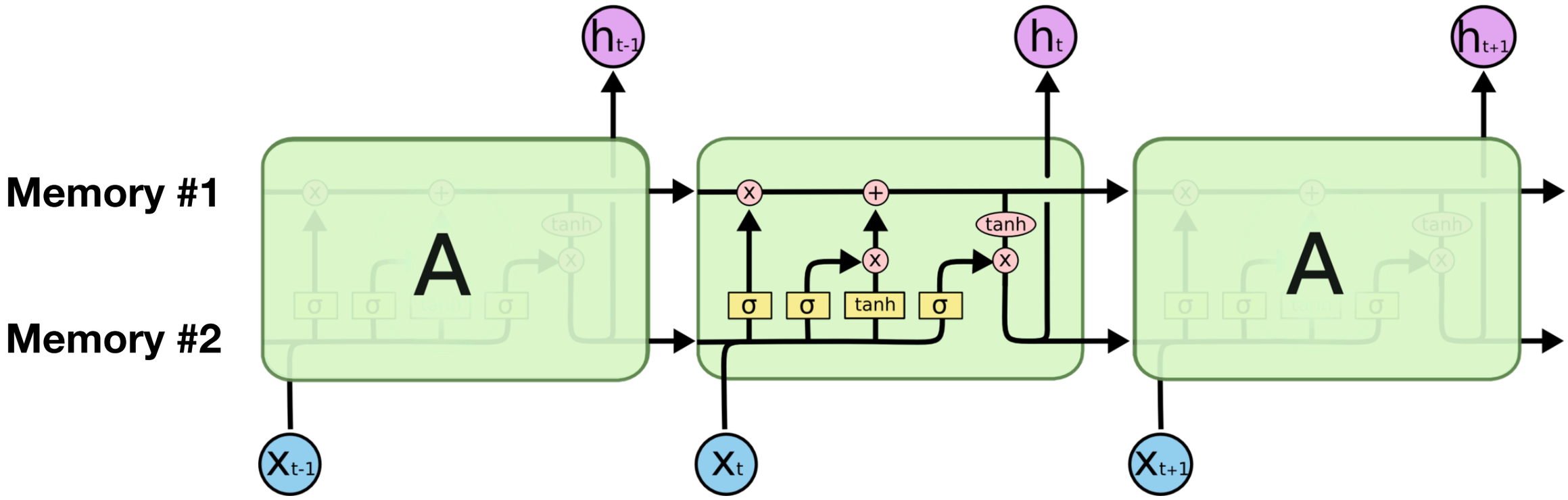
Transformer

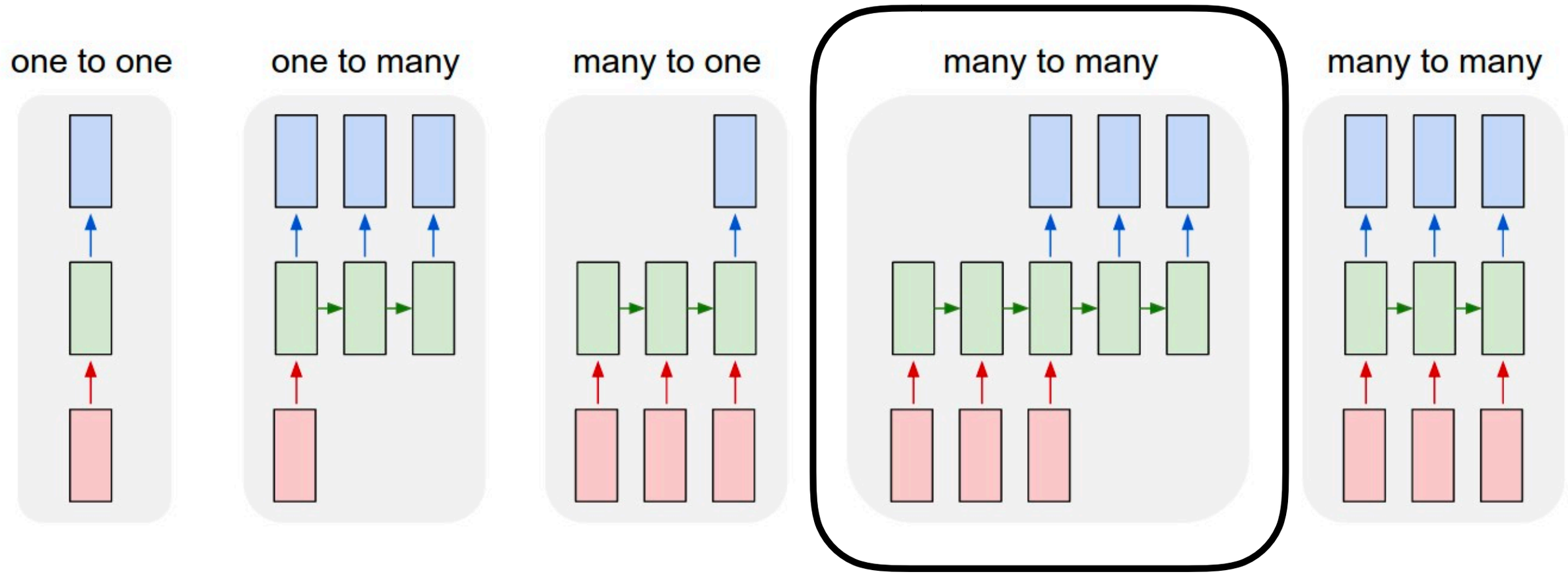
Boris Zubarev

@bobazooba

LSTM



Sequence to sequence



Sequence to sequence

Les pauvres sont démunis

Source sentence

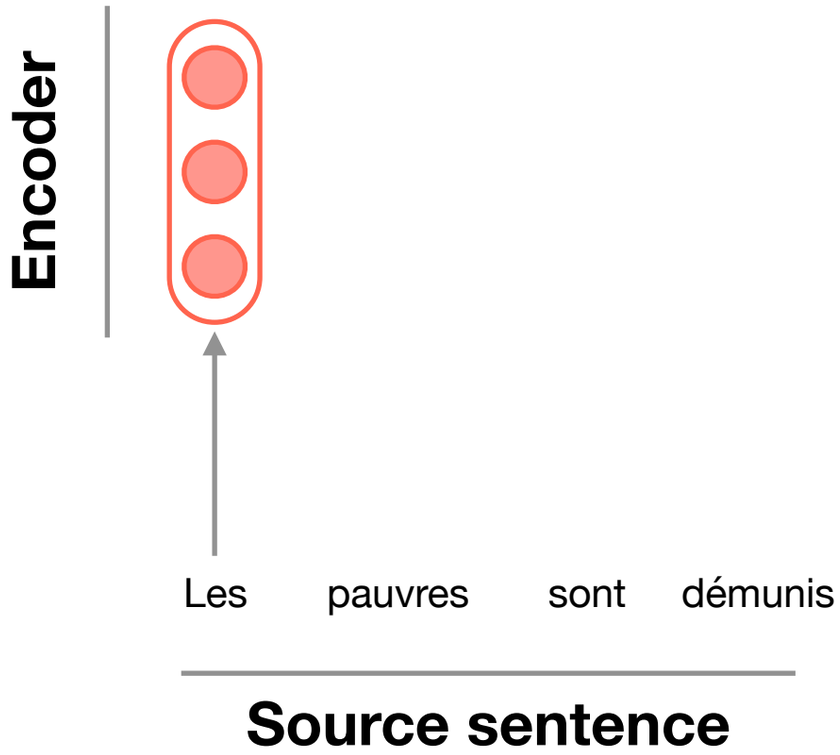
Sequence to sequence

Encoder

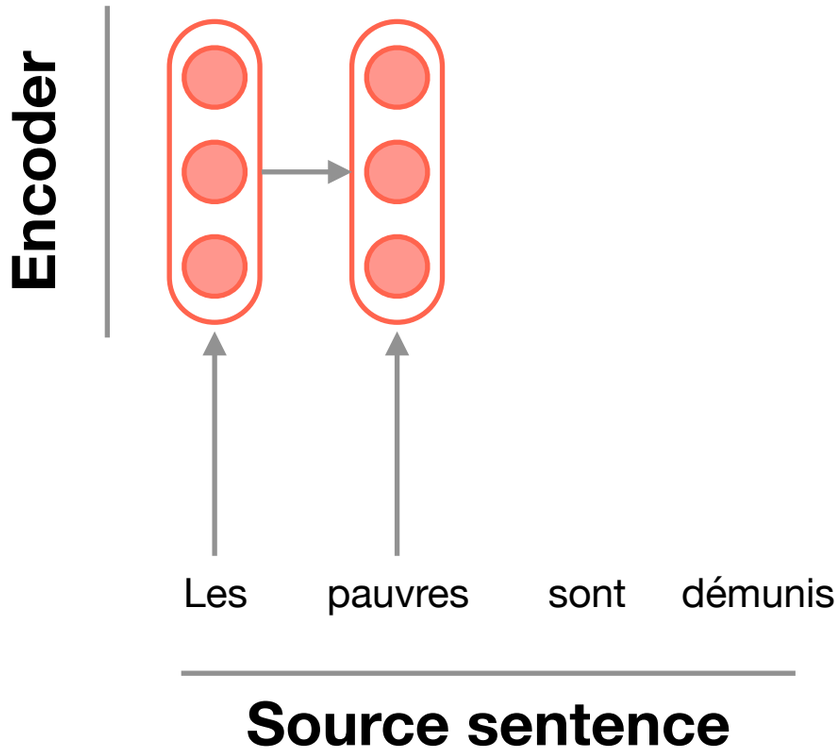
Les pauvres sont démunis

Source sentence

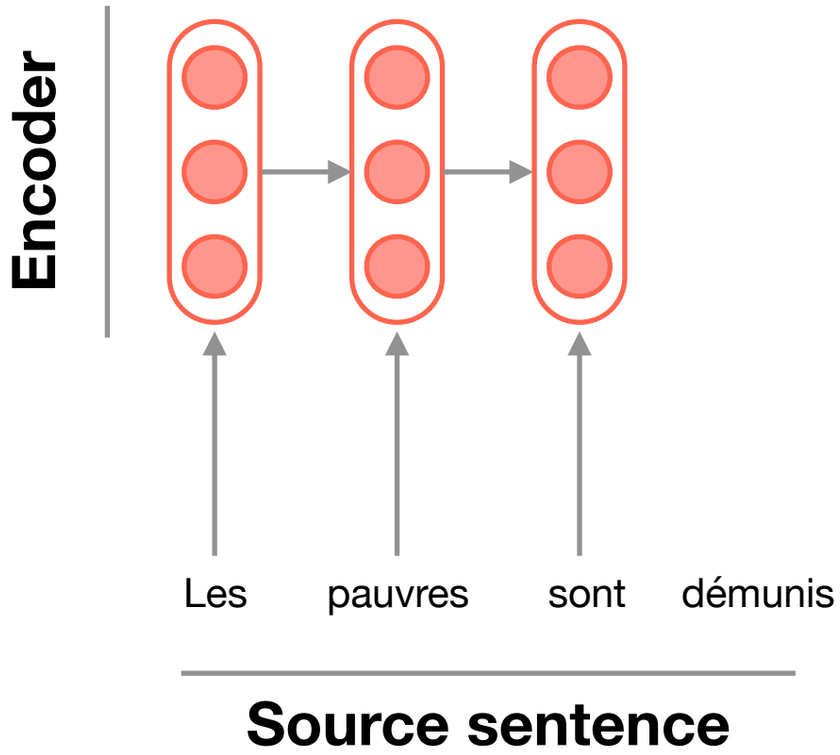
Sequence to sequence



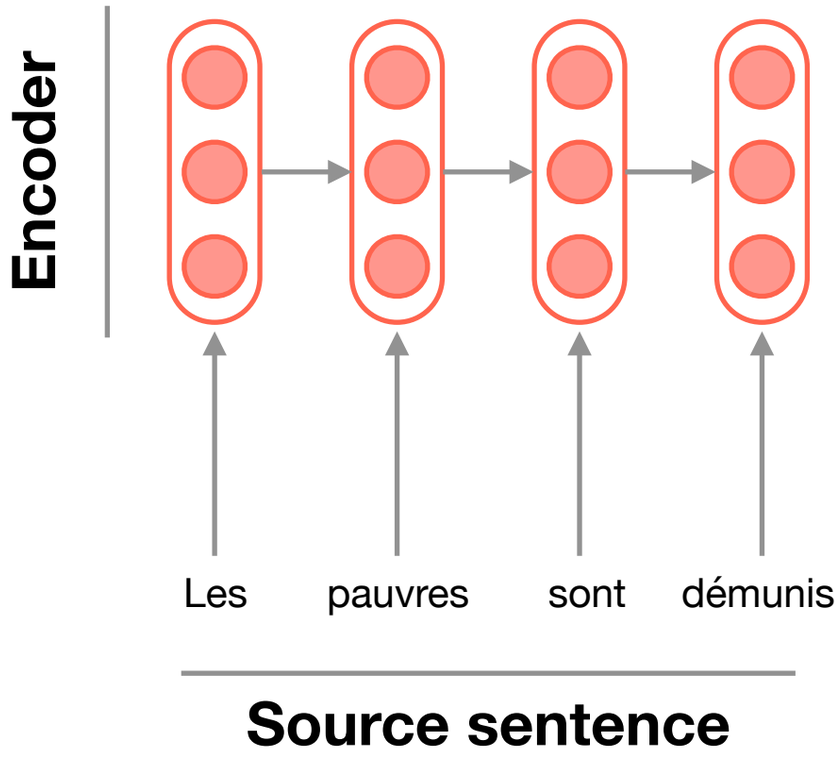
Sequence to sequence



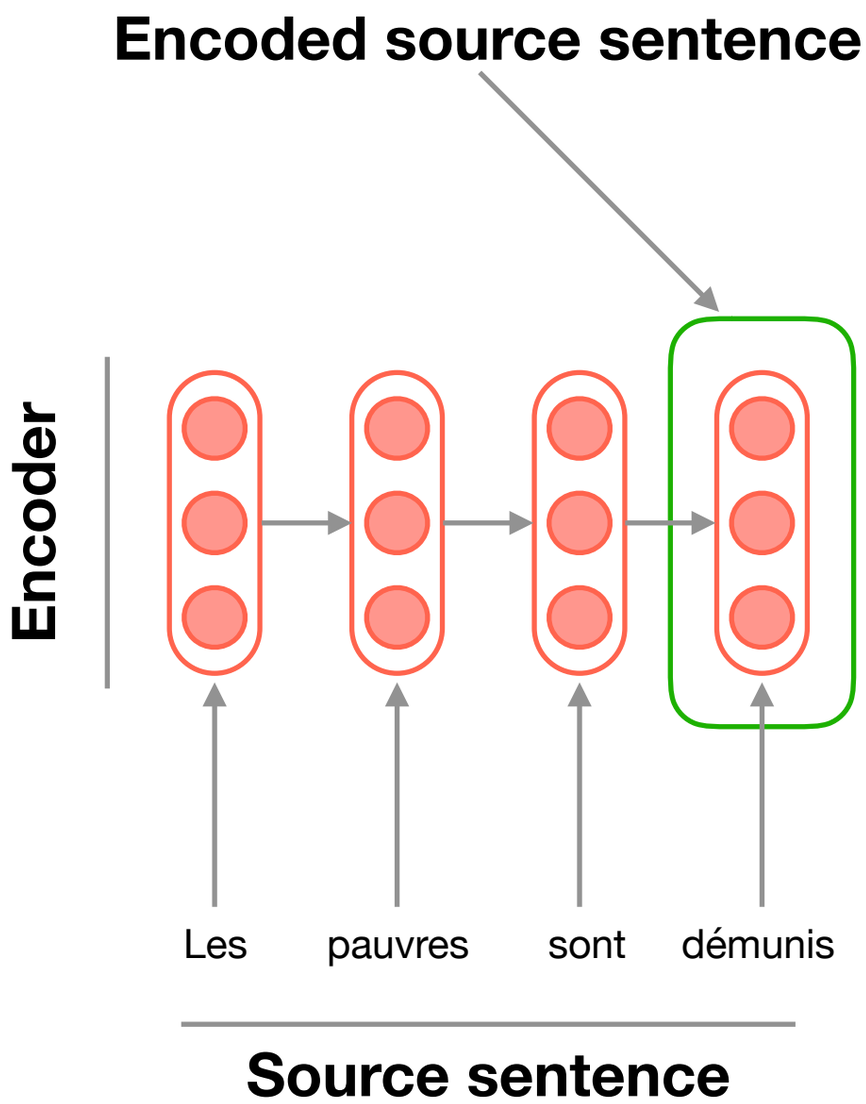
Sequence to sequence



Sequence to sequence

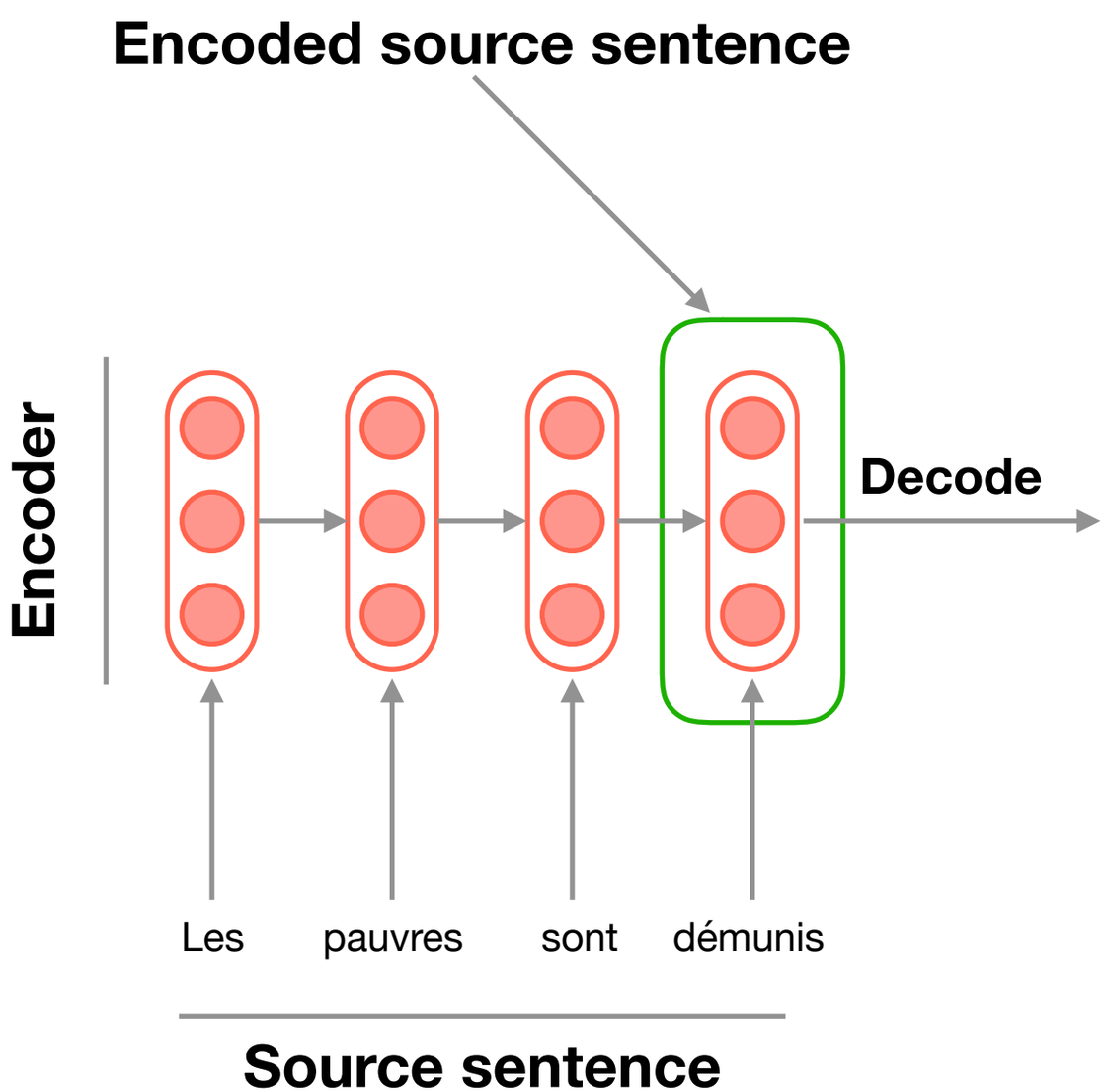


Sequence to sequence



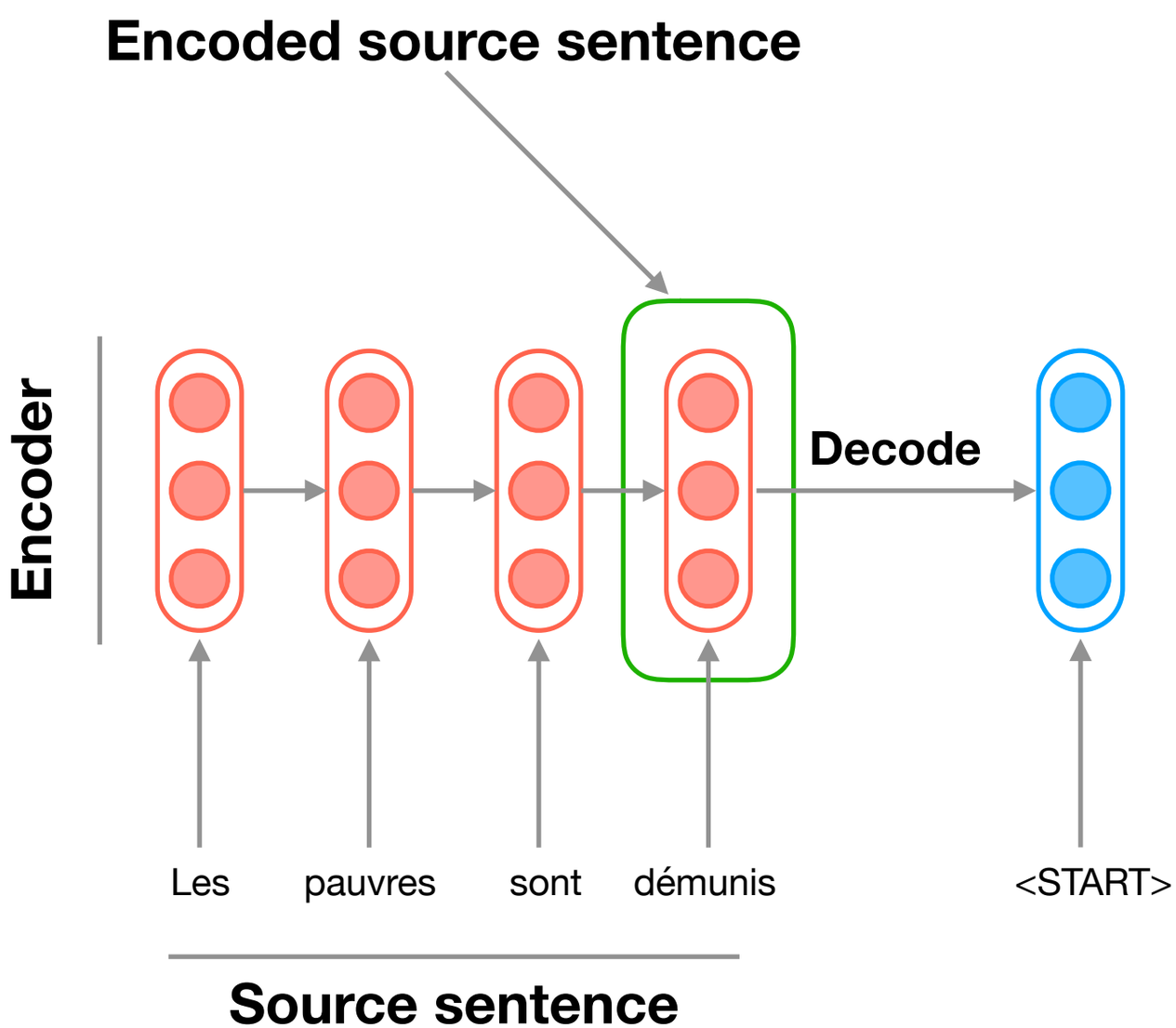
Sequence to sequence

Inference



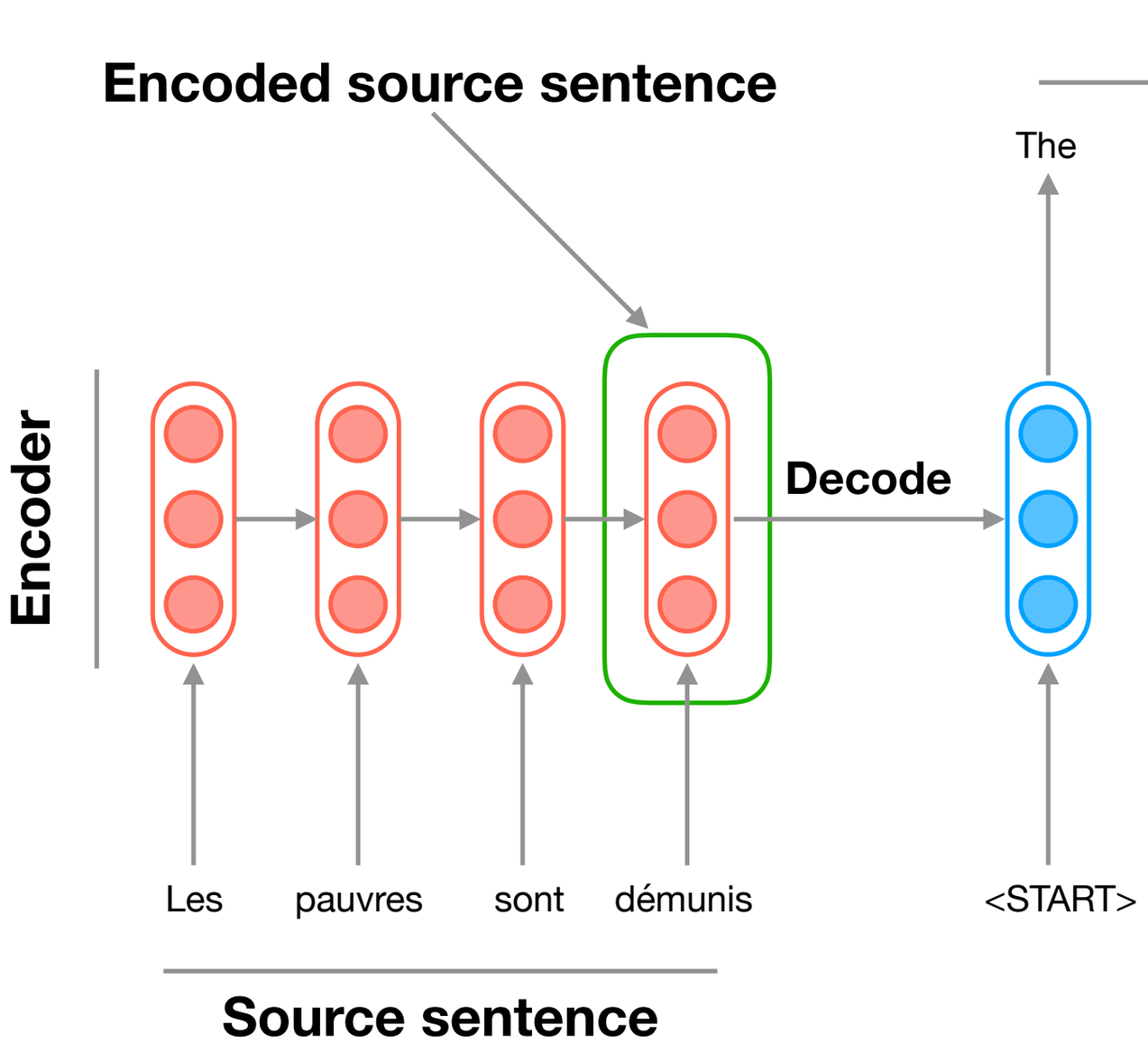
Sequence to sequence

Inference



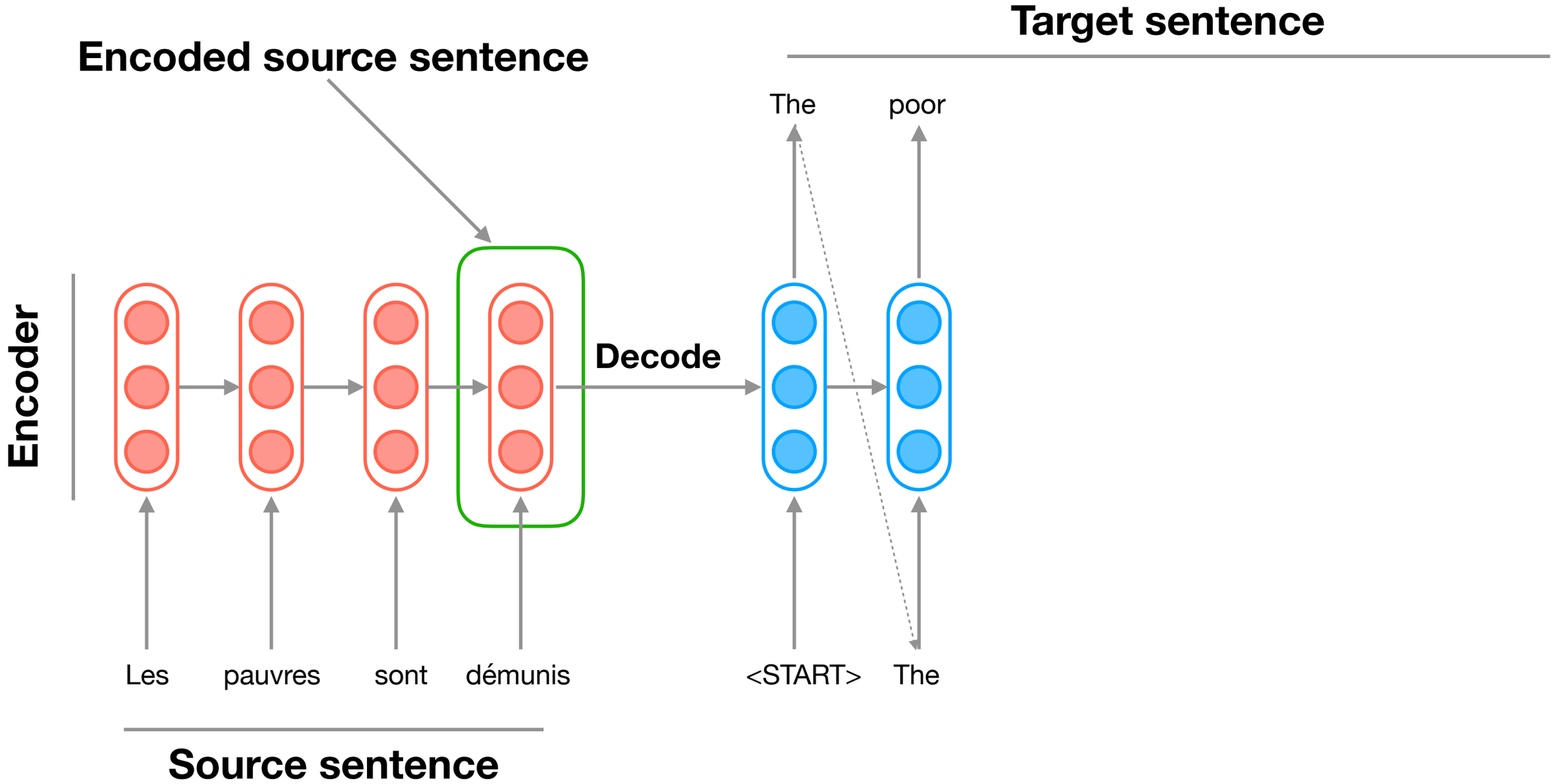
Sequence to sequence

Inference



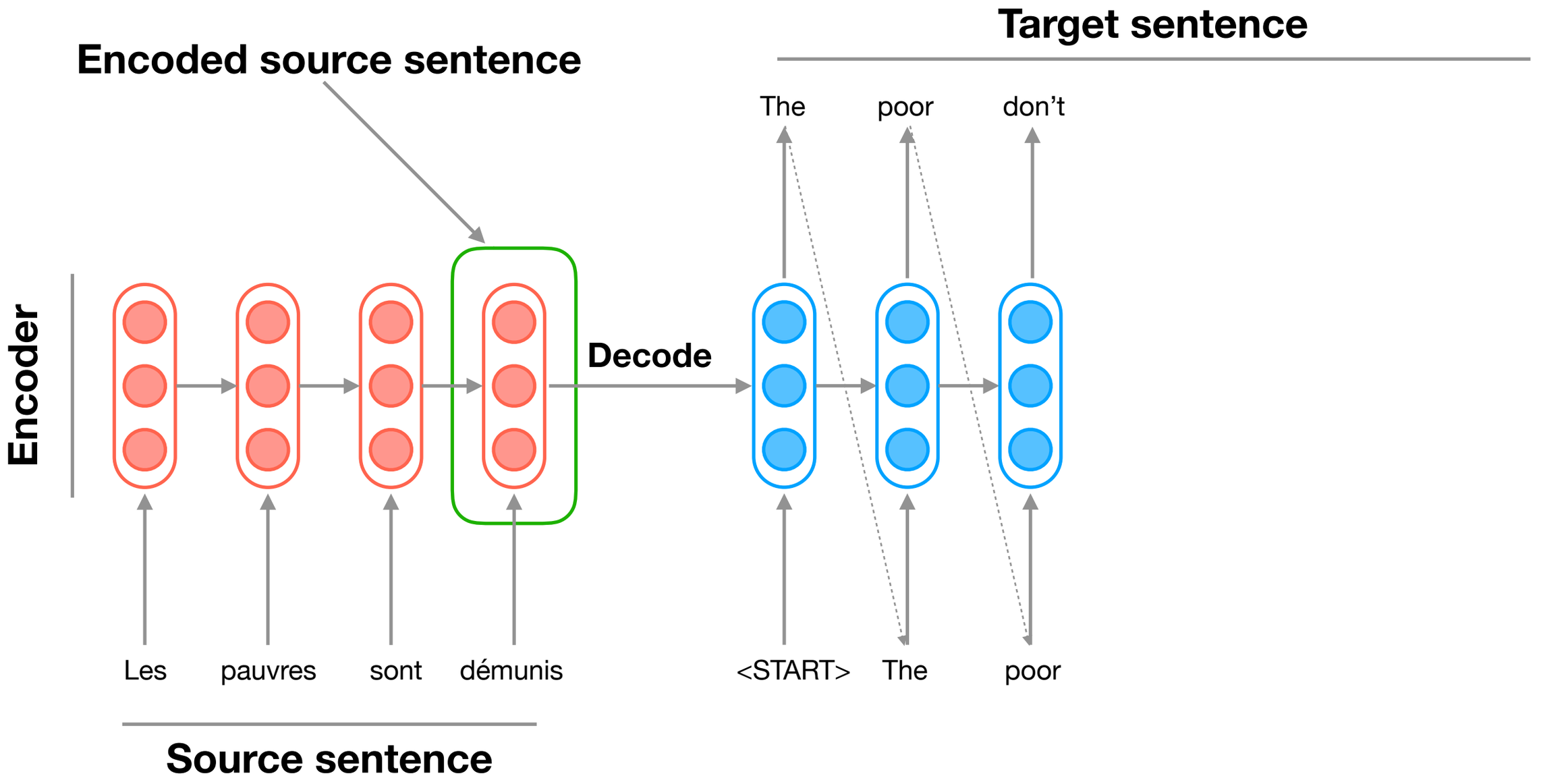
Sequence to sequence

Inference



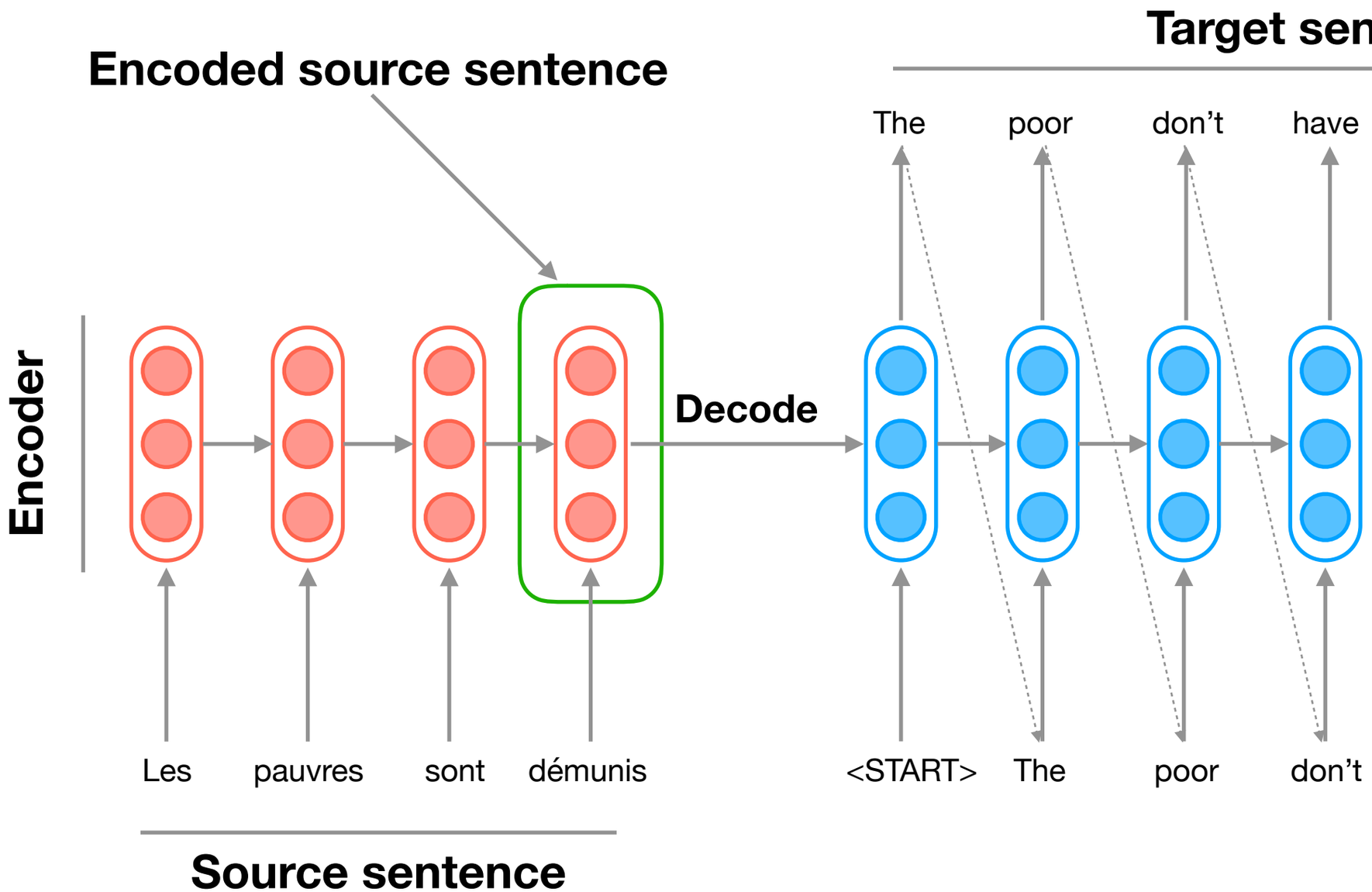
Sequence to sequence

Inference



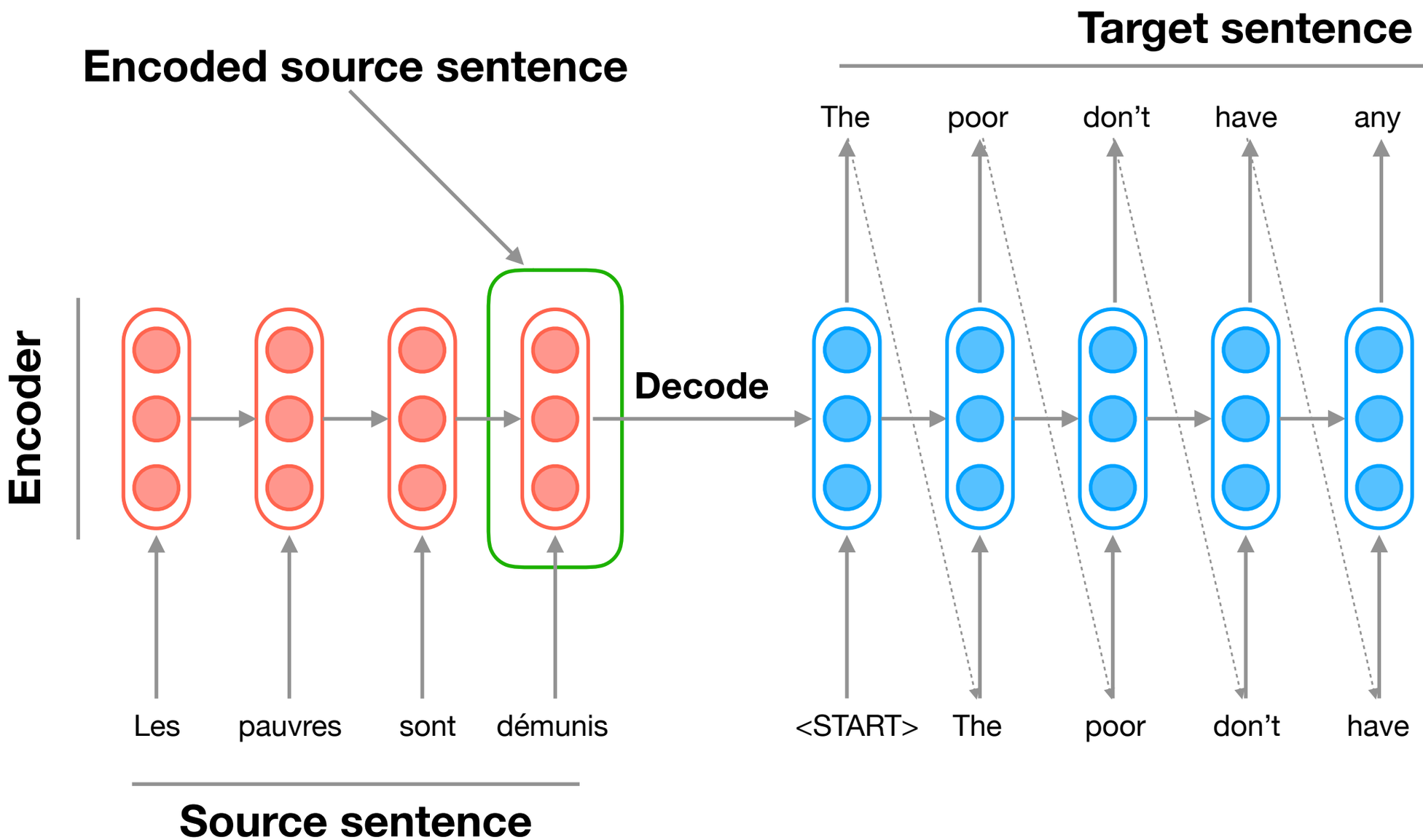
Sequence to sequence

Inference



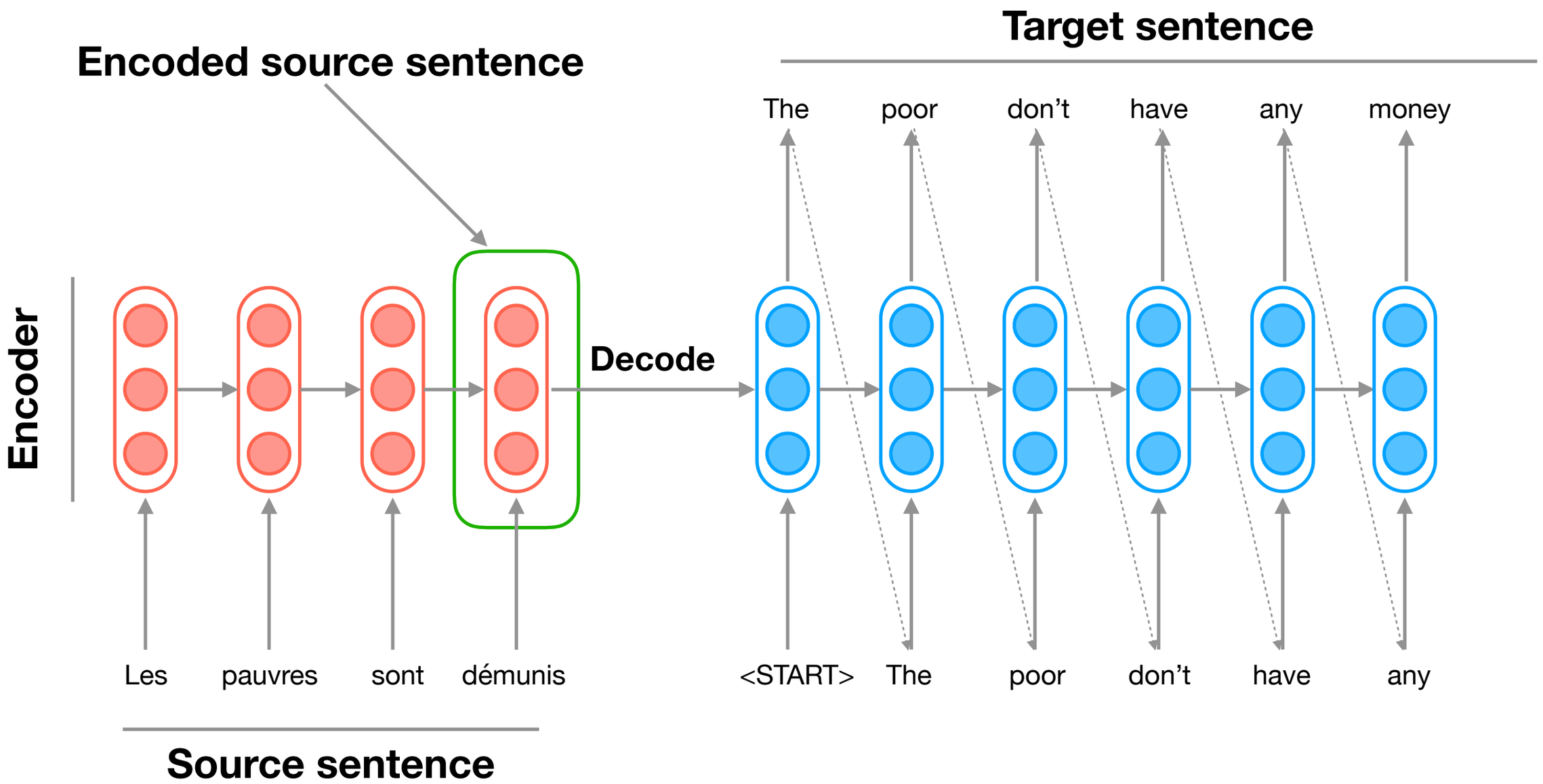
Sequence to sequence

Inference



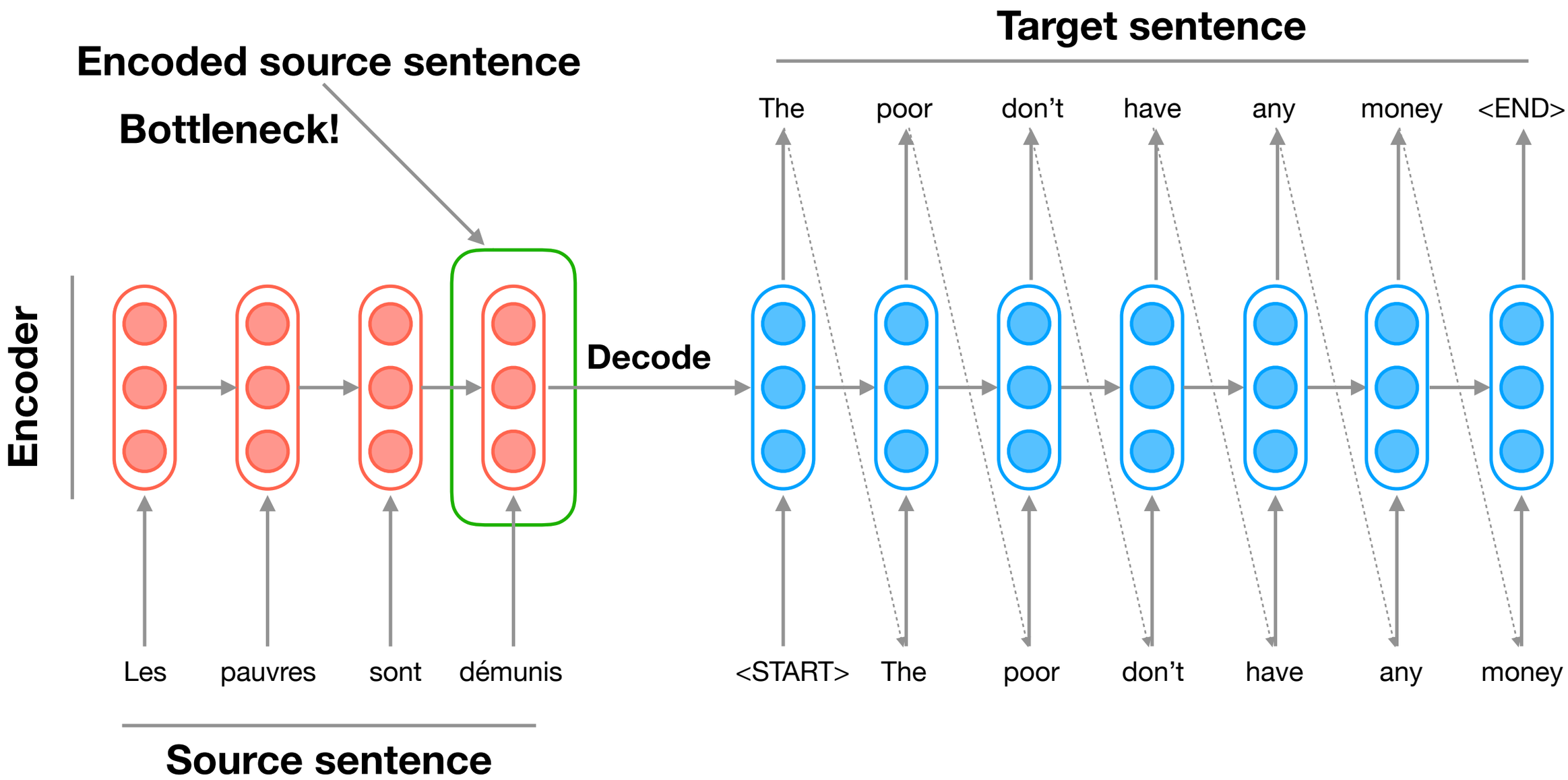
Sequence to sequence

Inference



Sequence to sequence

Inference



Contextualized Word Vectors

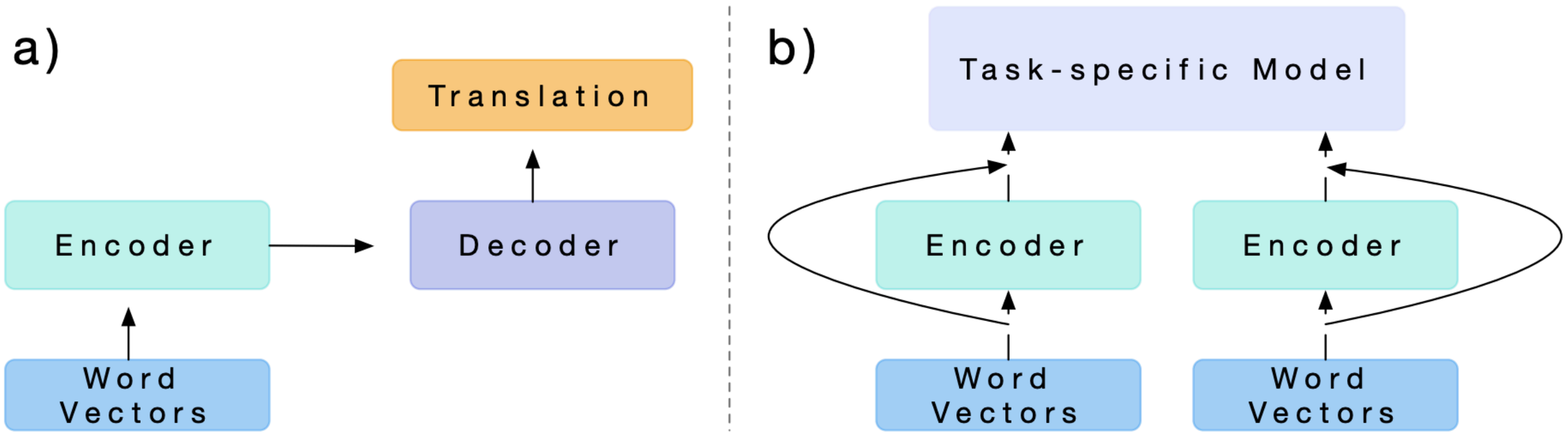


Figure 1: We a) train a two-layer, bidirectional LSTM as the encoder of an attentional sequence-to-sequence model for machine translation and b) use it to provide context for other NLP models.

Recurrent Neural Network

+

- Words relationship
- Variable length

—

- Forgot information
- Recurrent dependence

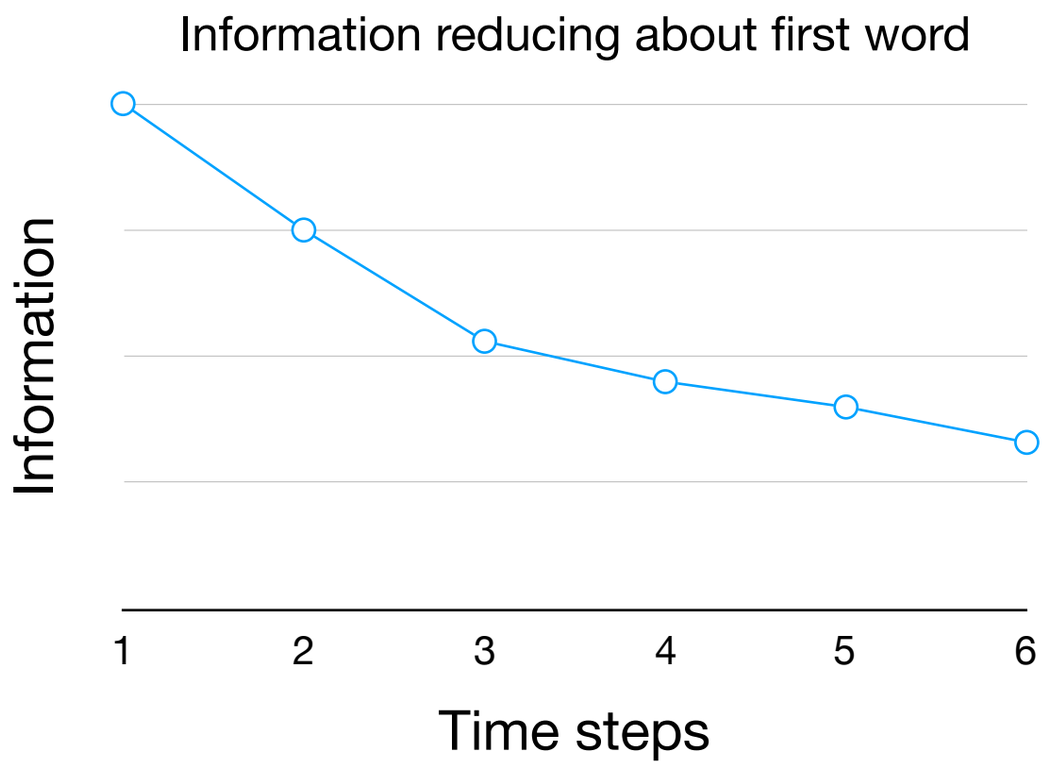
Recurrent Neural Network

+

- Words relationship
- Variable length

—

- Forgot information
- Recurrent dependence



Dot product

- Cosine similarity (just divide by norms) become measure between 0 and 1

Dot product

- Cosine similarity (just divide by norms) become measure between 0 and 1

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}},$$

Dot product

- Cosine similarity (just divide by norms) become measure between 0 and 1

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}},$$

```
x1 = np.random.rand(300)
x2 = np.random.rand(300)
```

```
np.dot(x1, x2)
```

```
69.77192646564589
```

```
np.dot(x1, x2) / (np.linalg.norm(x1) * np.linalg.norm(x2))
```

```
0.7349837217006946
```

```
1 - distance.cosine(x1, x2)
```

```
0.7349837217006939
```

Dot product

- Cosine similarity (just divide by norms) become measure between 0 and 1

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}},$$

Dot product

```
x1 = np.random.rand(300)
x2 = np.random.rand(300)
```

```
np.dot(x1, x2)
```

```
69.77192646564589
```

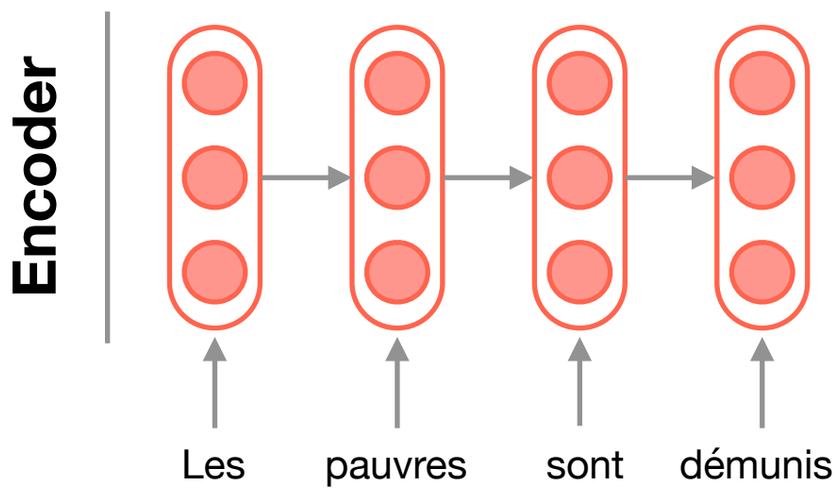
```
np.dot(x1, x2) / (np.linalg.norm(x1) * np.linalg.norm(x2))
```

```
0.7349837217006946
```

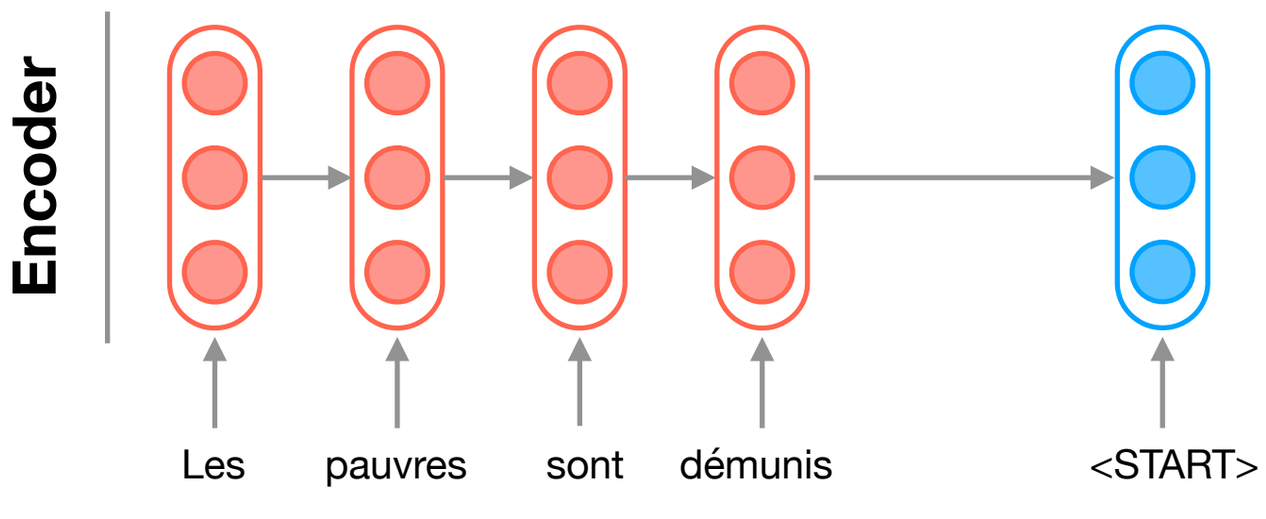
```
1 - distance.cosine(x1, x2)
```

```
0.7349837217006939
```

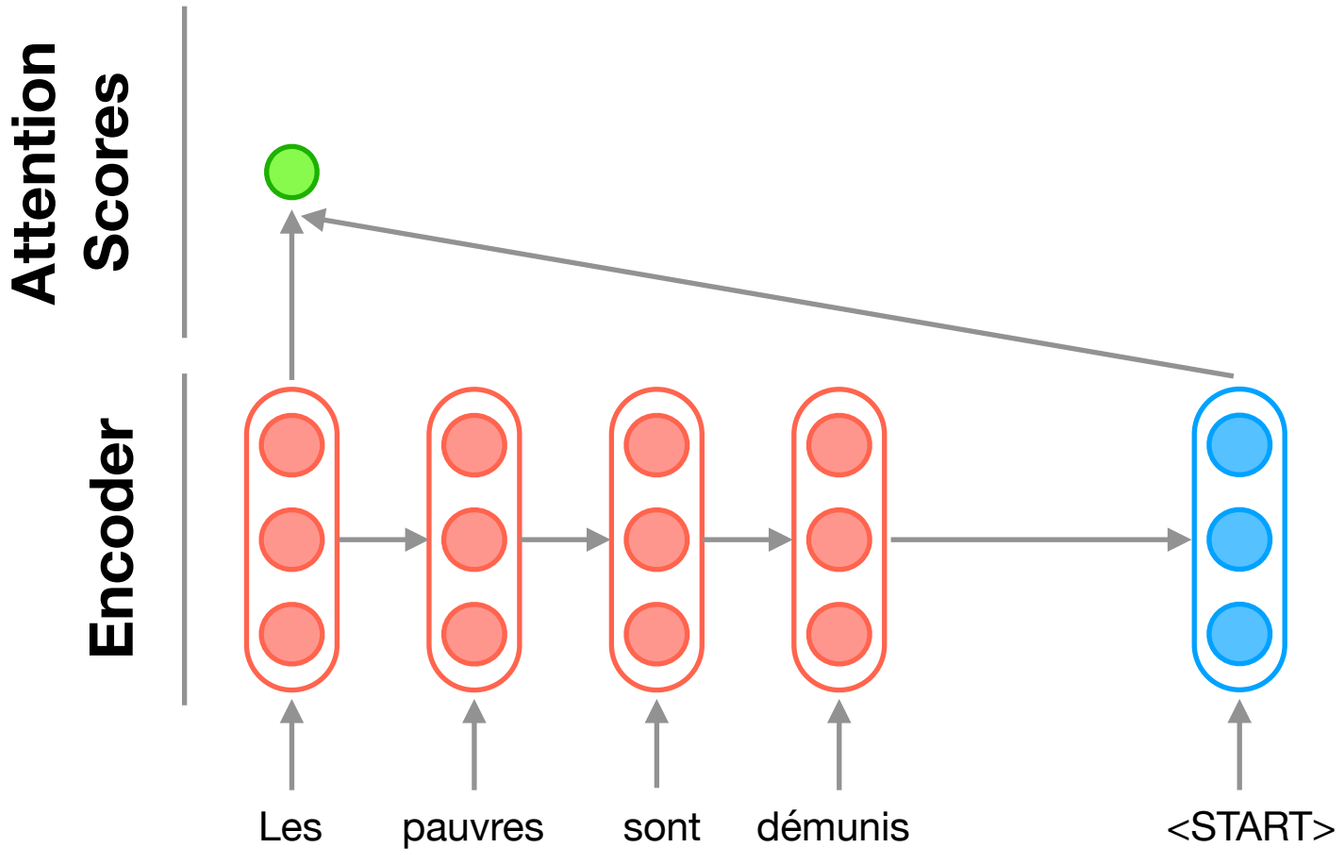
Attention



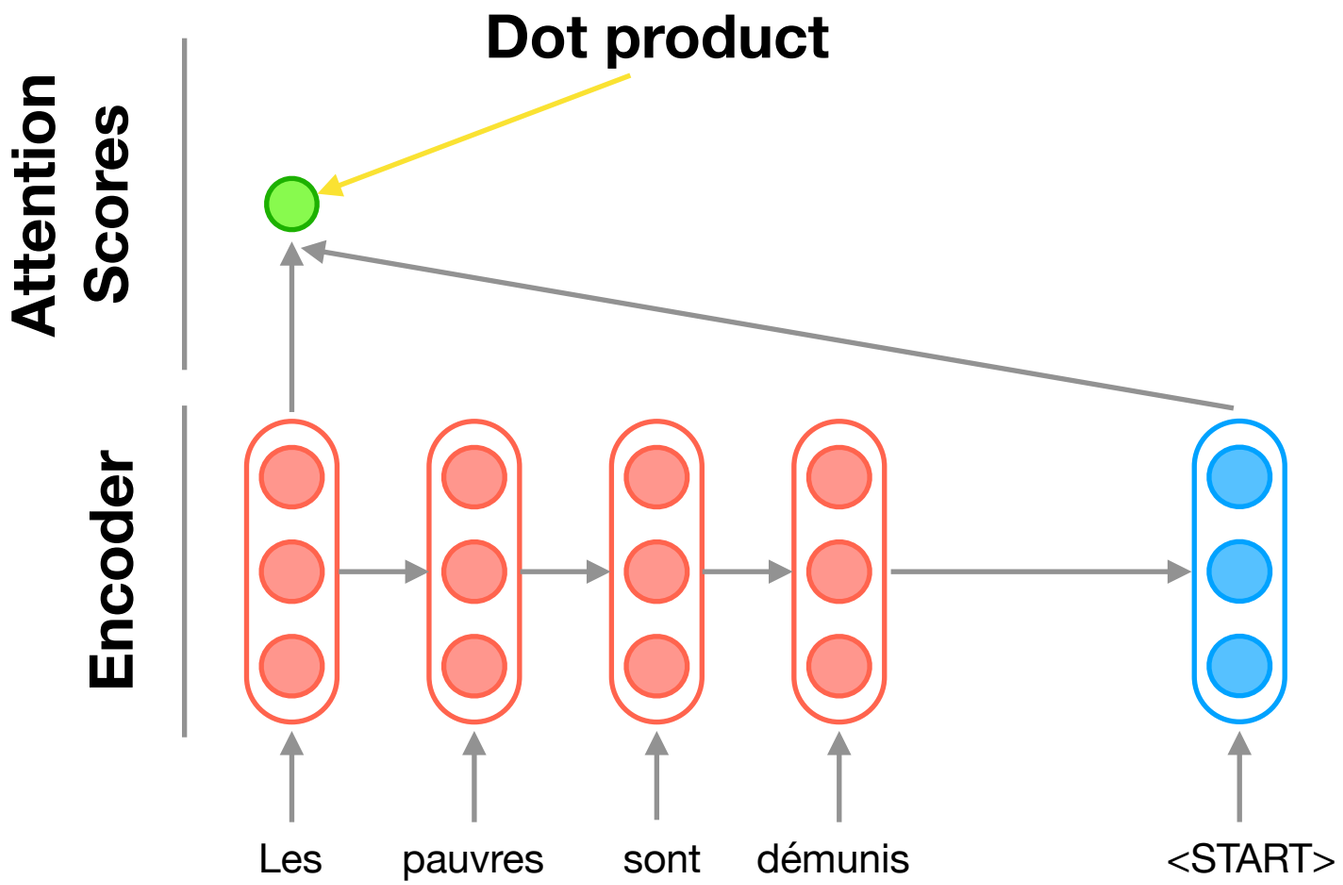
Attention



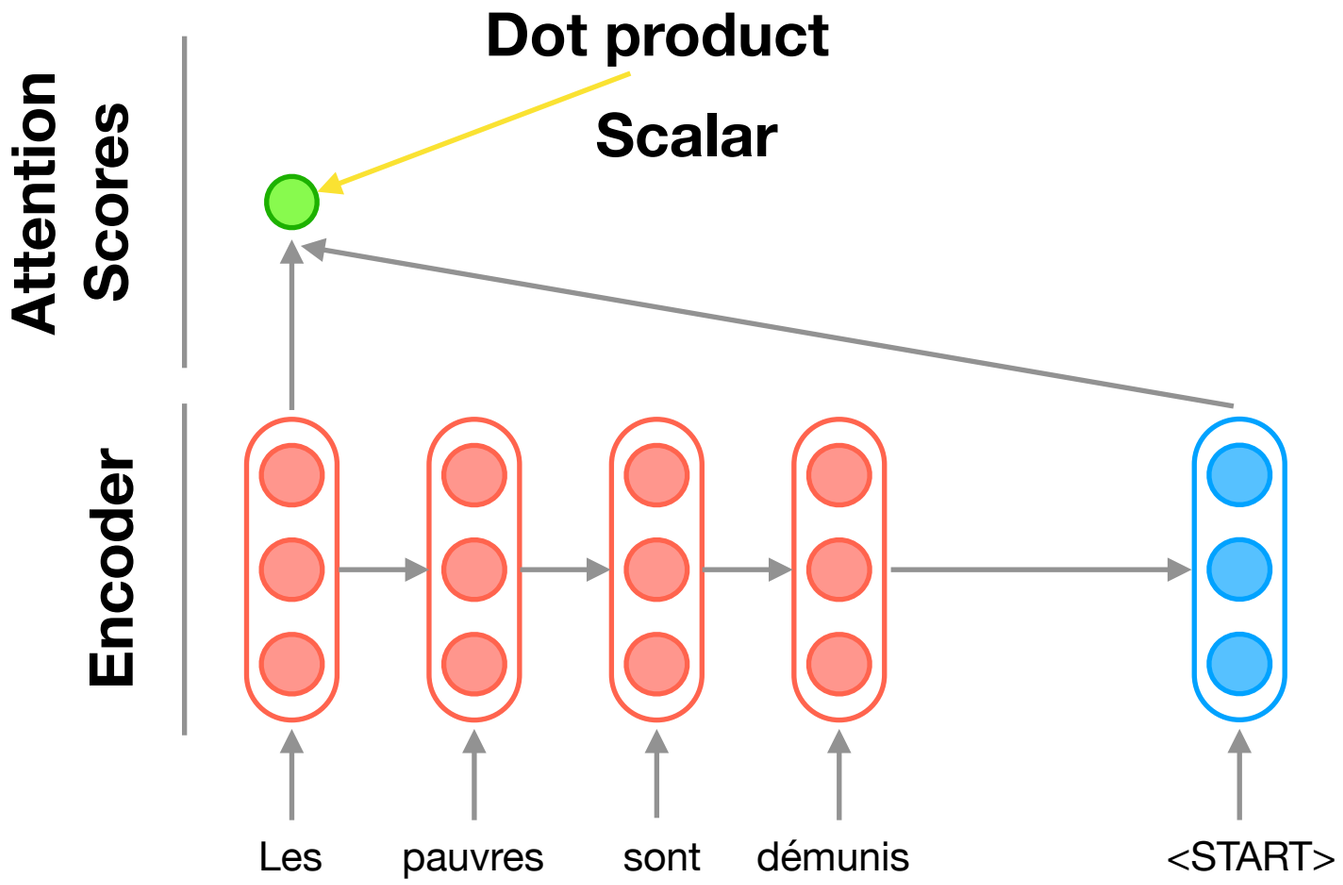
Attention



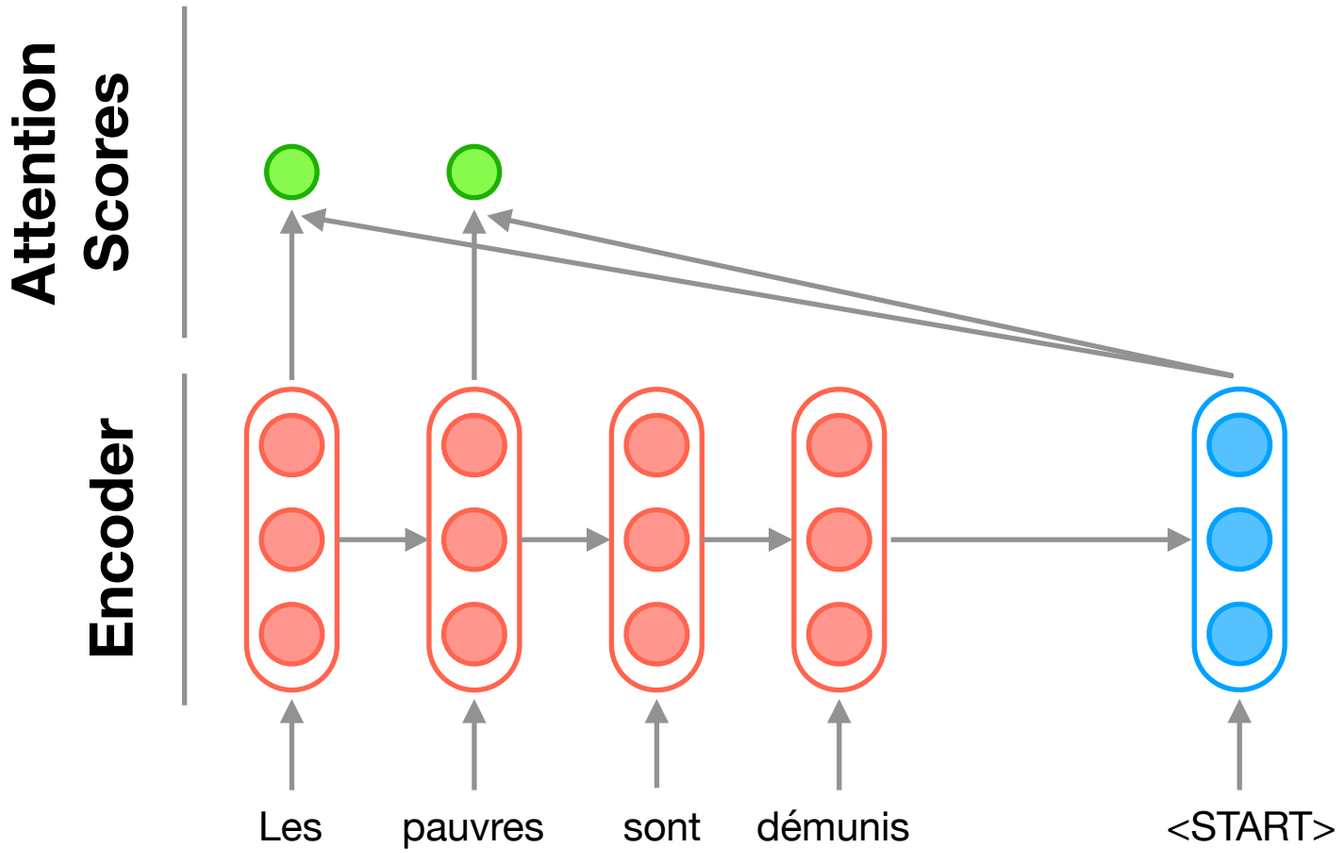
Attention


Dot product
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>

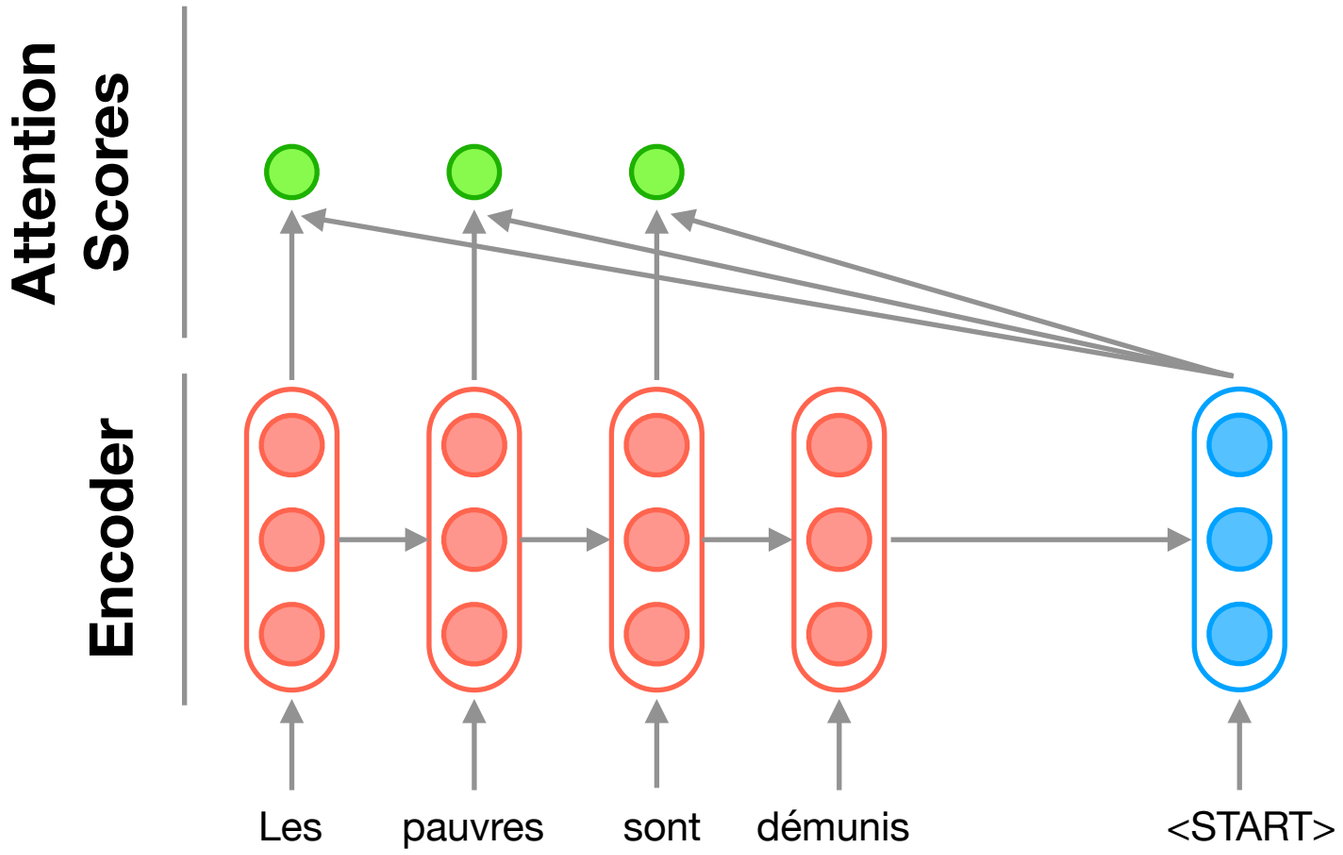
Attention


Dot product
Scalar
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>

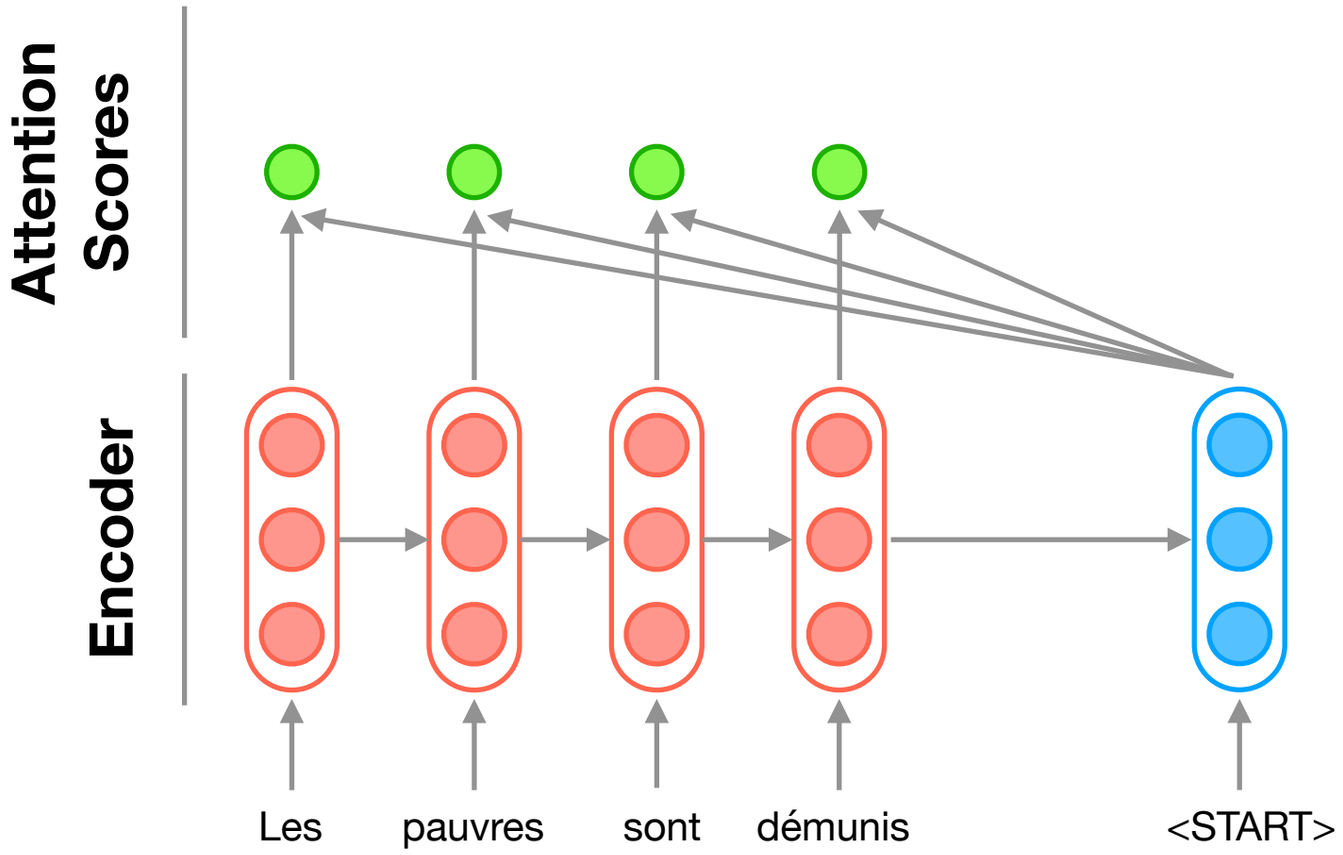
Attention



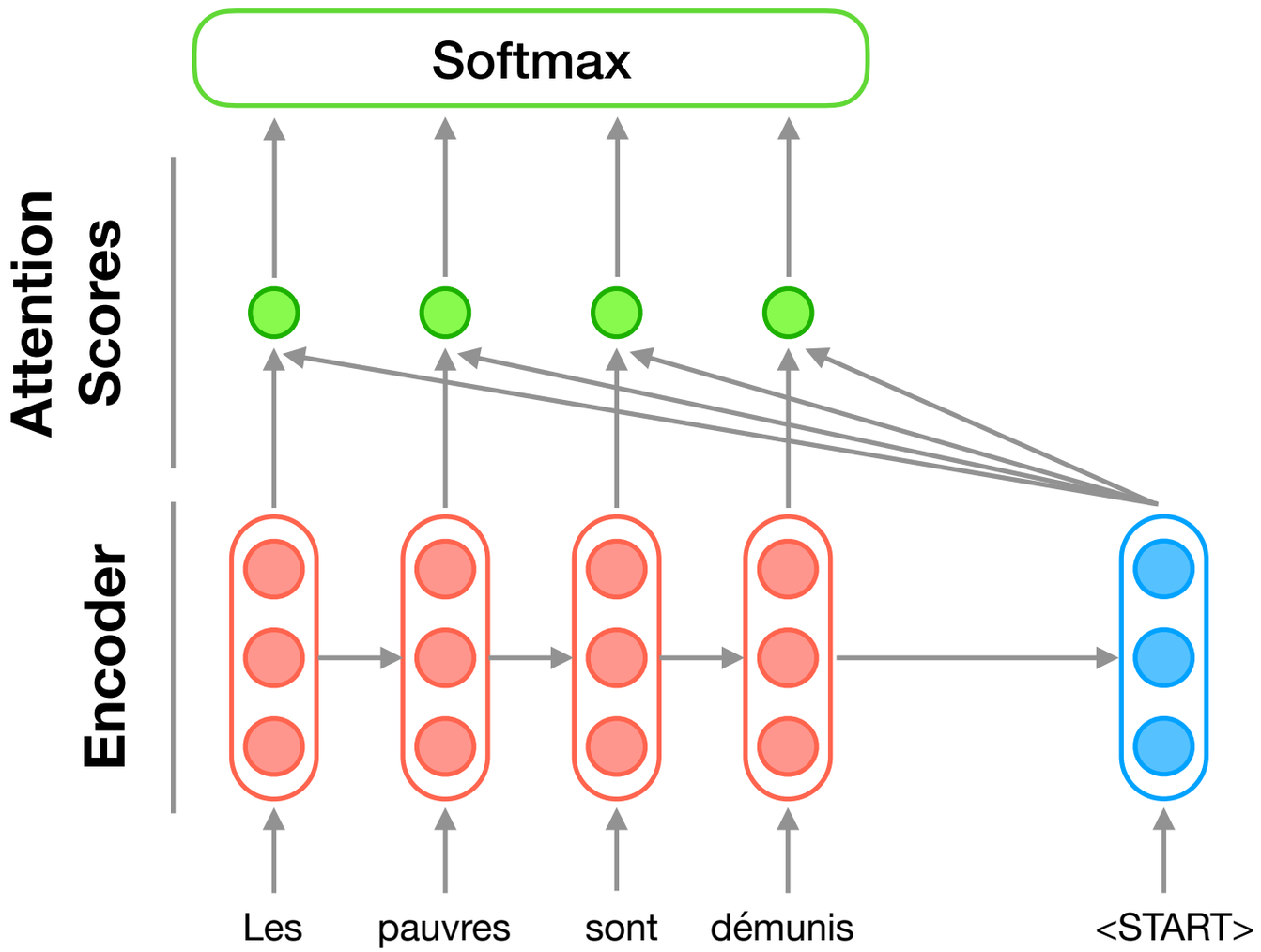
Attention



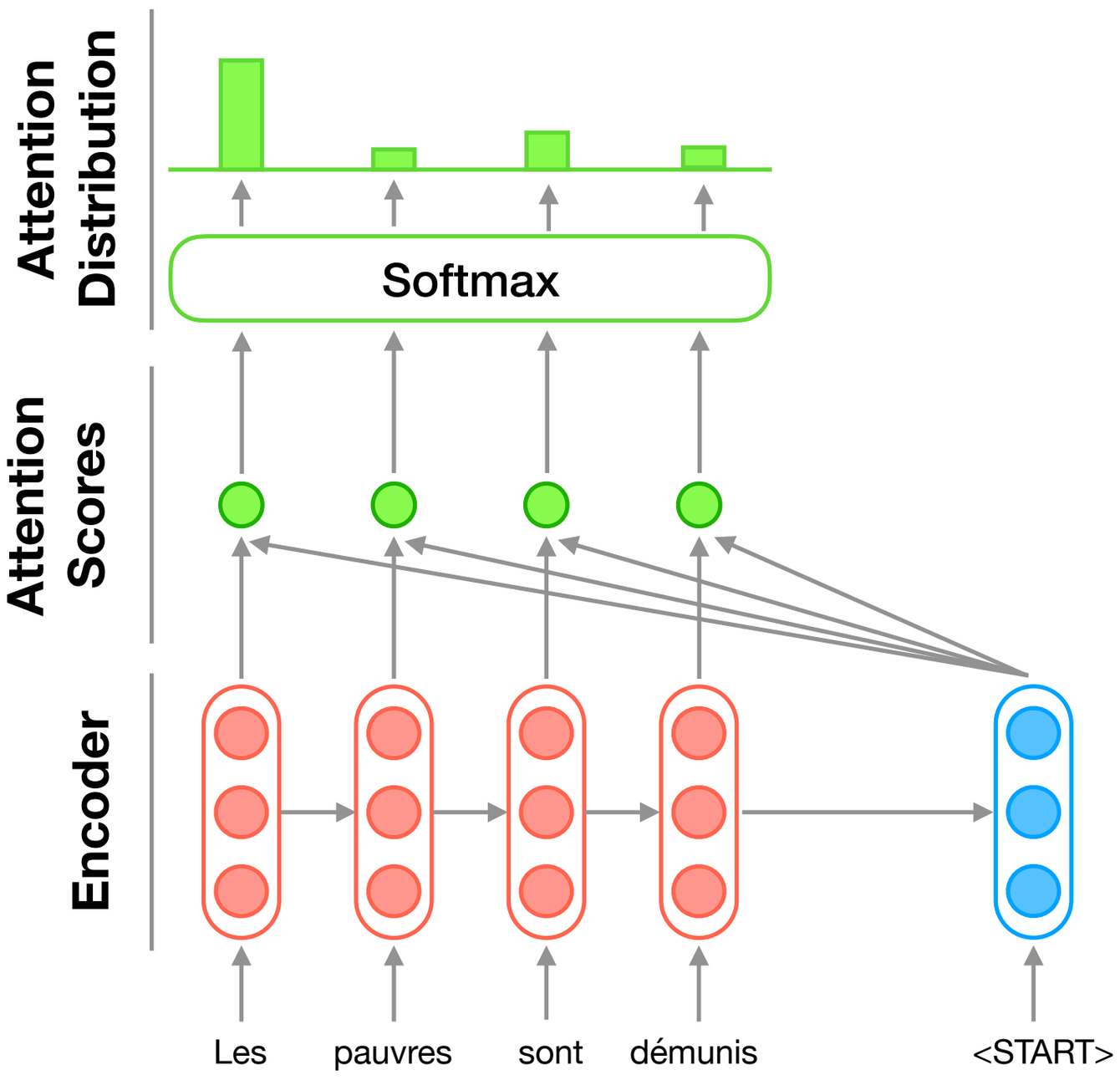
Attention



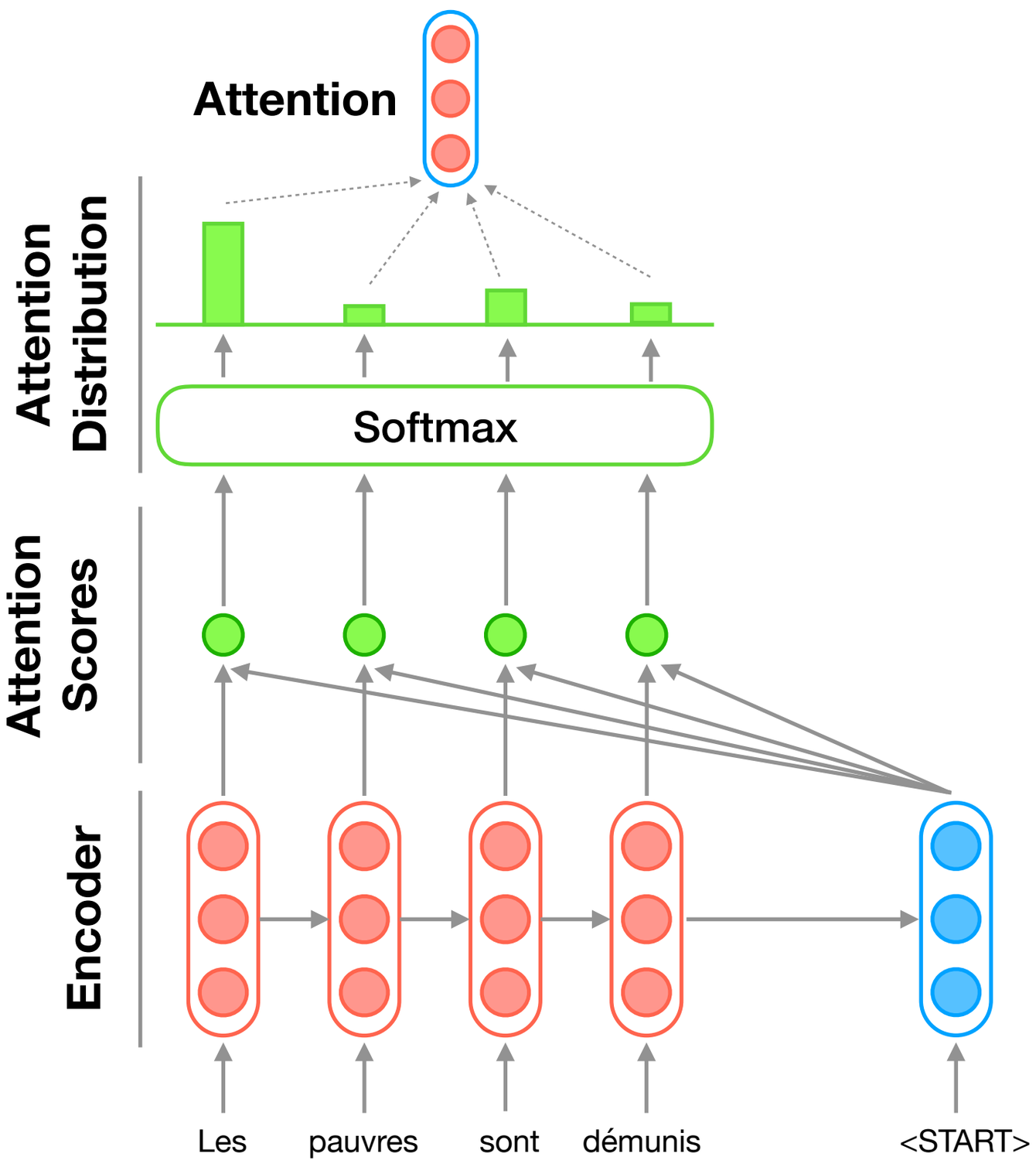
Attention


Softmax
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>

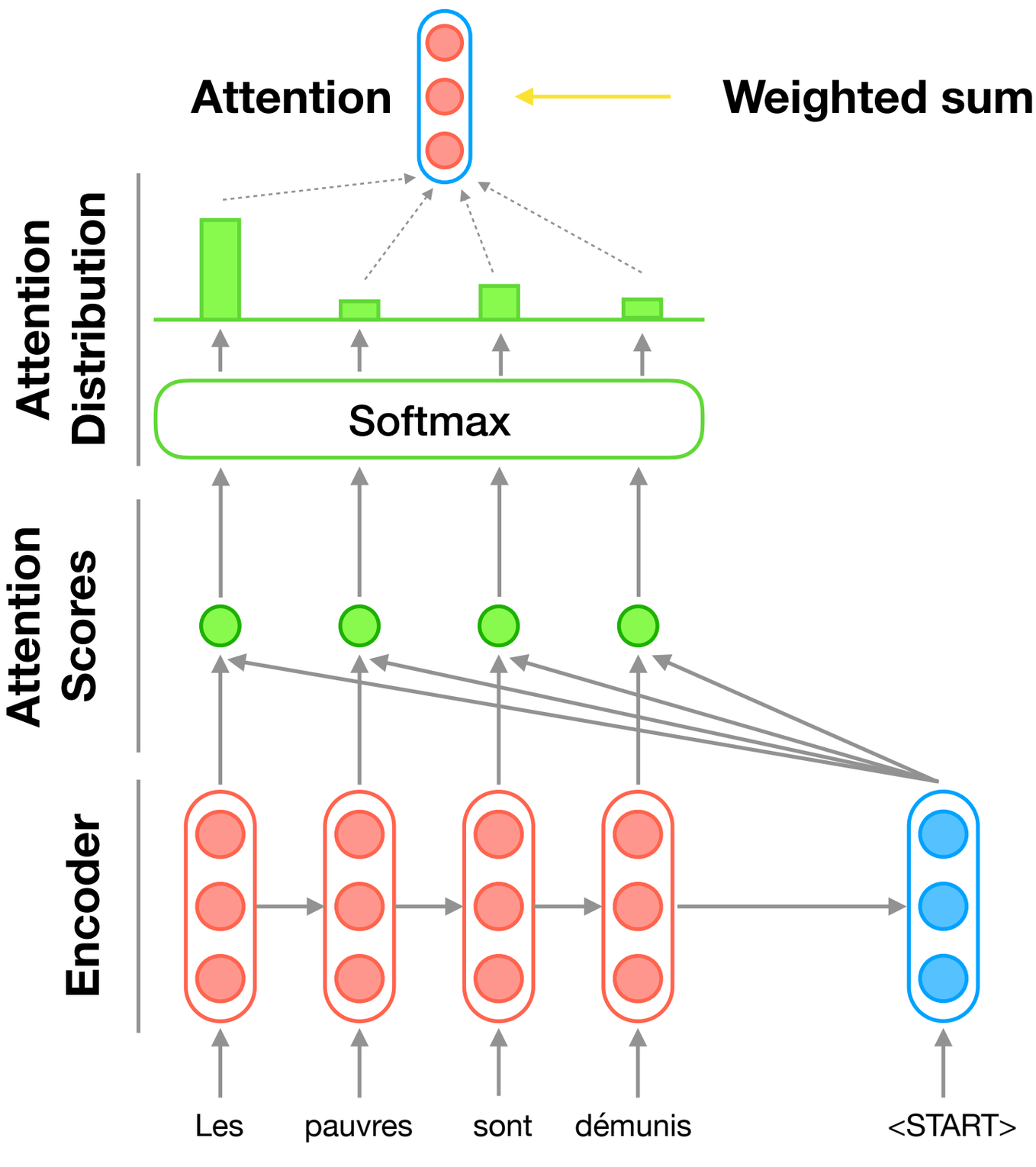
Attention


Attention Distribution
Softmax
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>

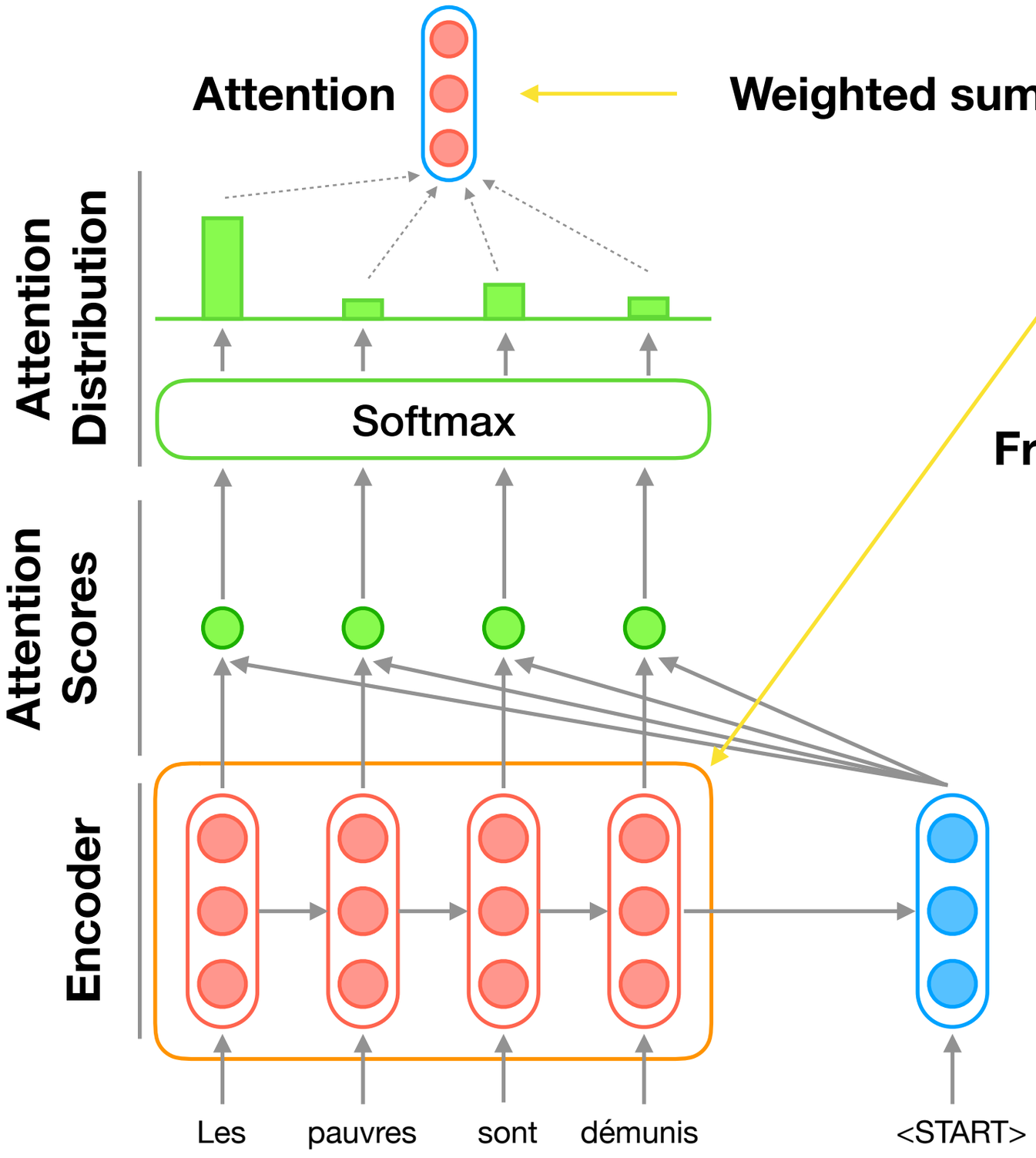
Attention


Attention
Attention Distribution
Softmax
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>

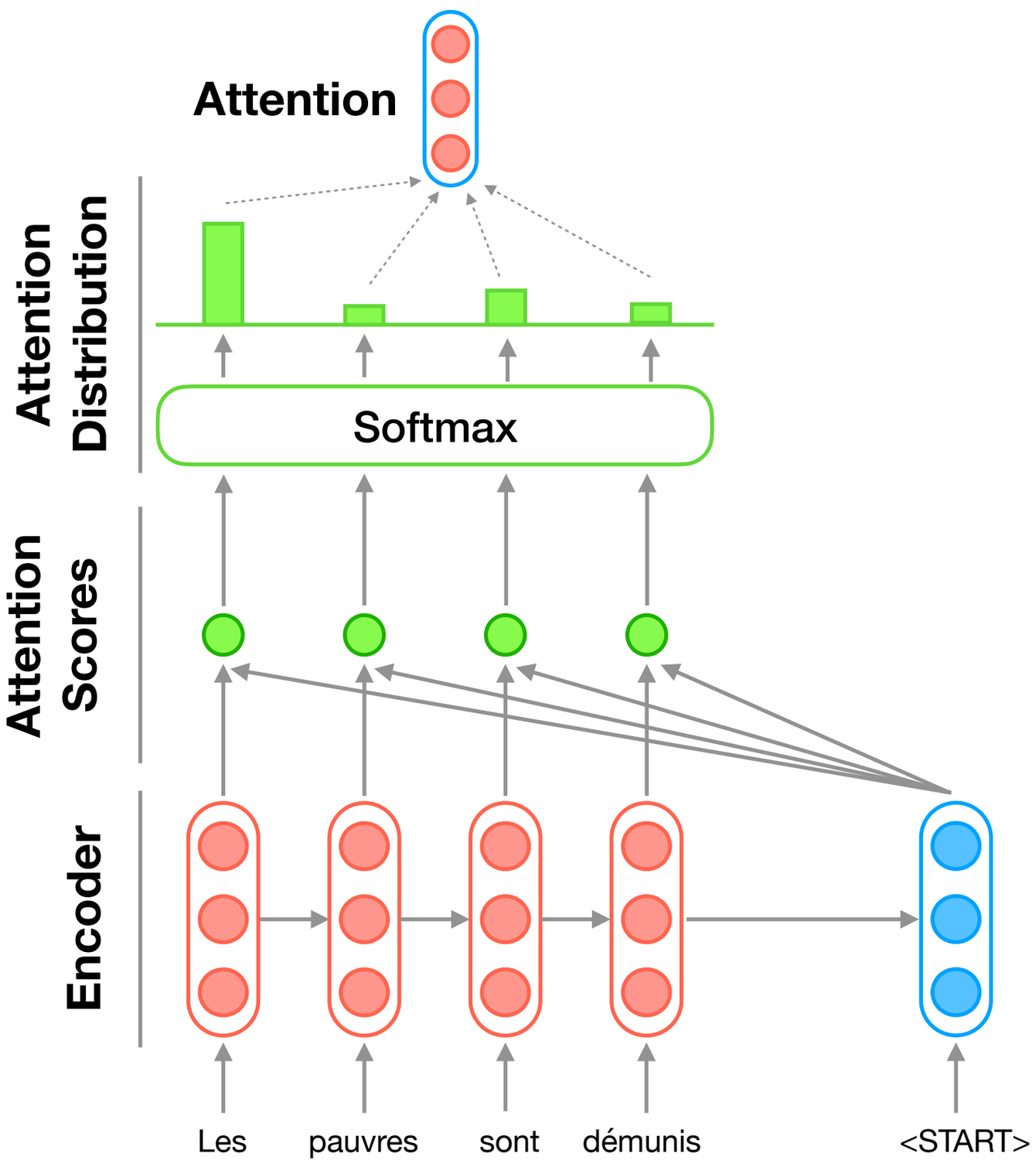
Attention


Attention
Weighted sum of encoder hidden states
Attention Distribution
Softmax
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>

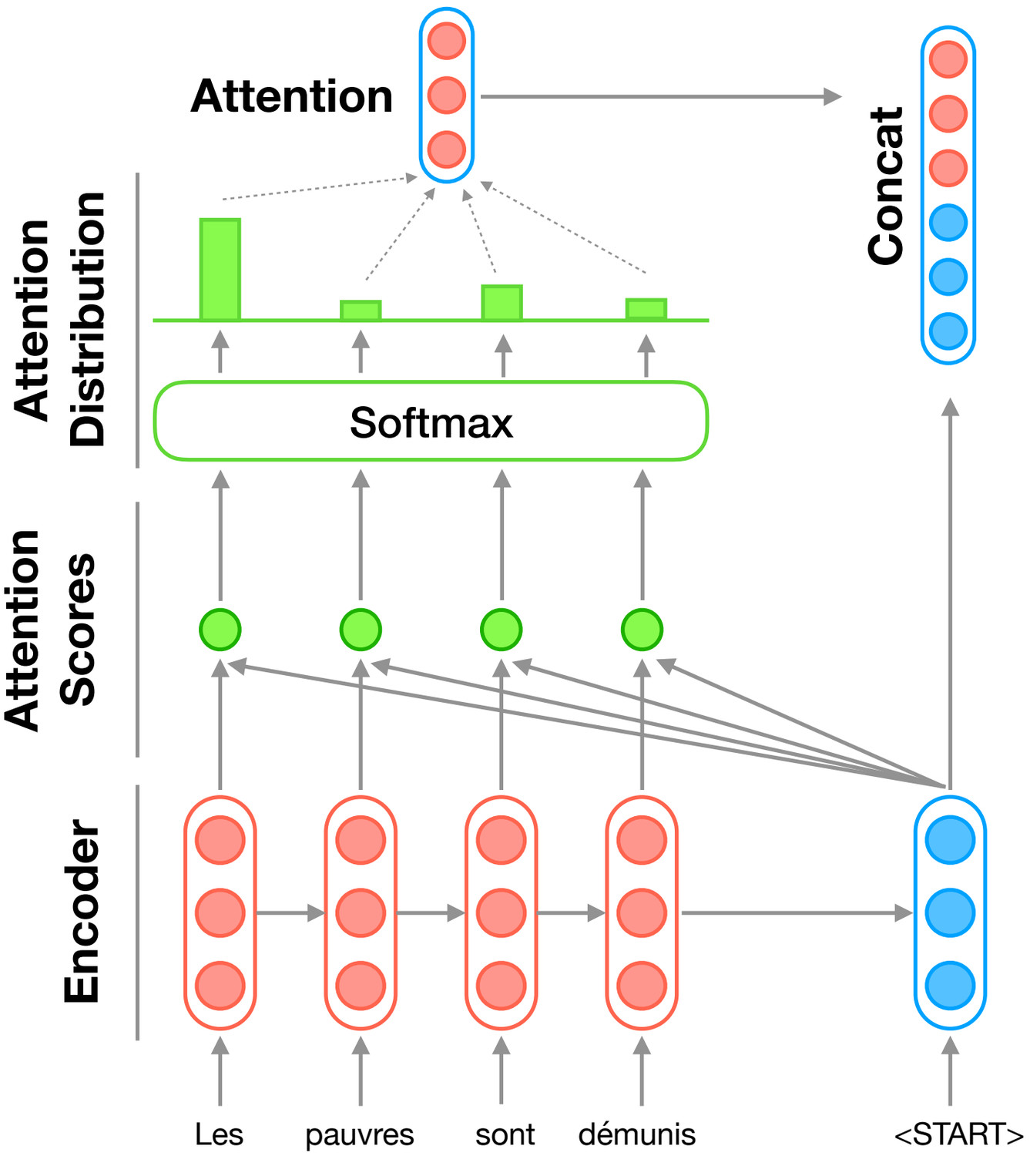
Attention


Attention
Weighted sum of encoder hidden states
Attention Distribution
Softmax
From this
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>

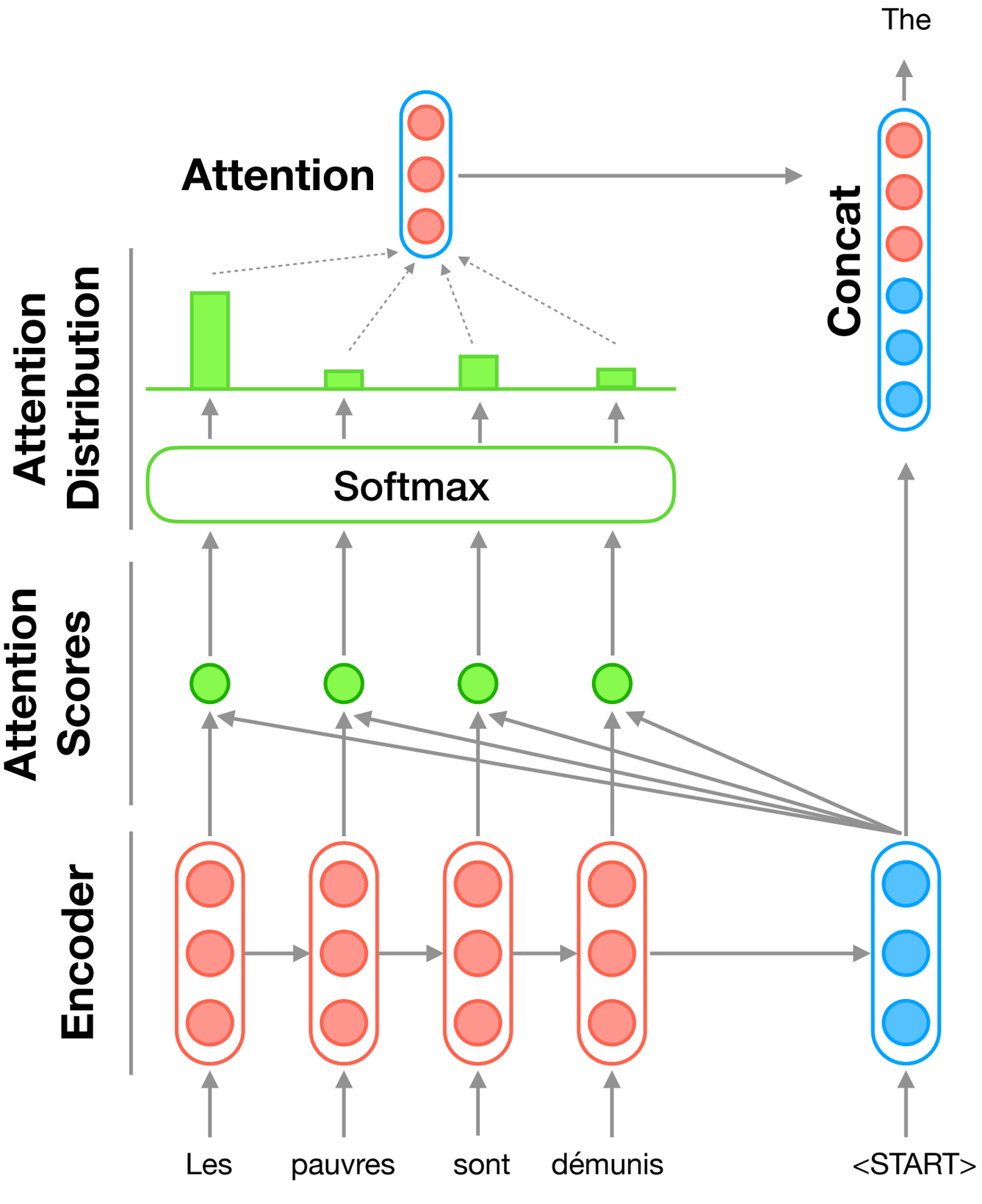
Attention


Attention
Attention Distribution
Softmax
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>

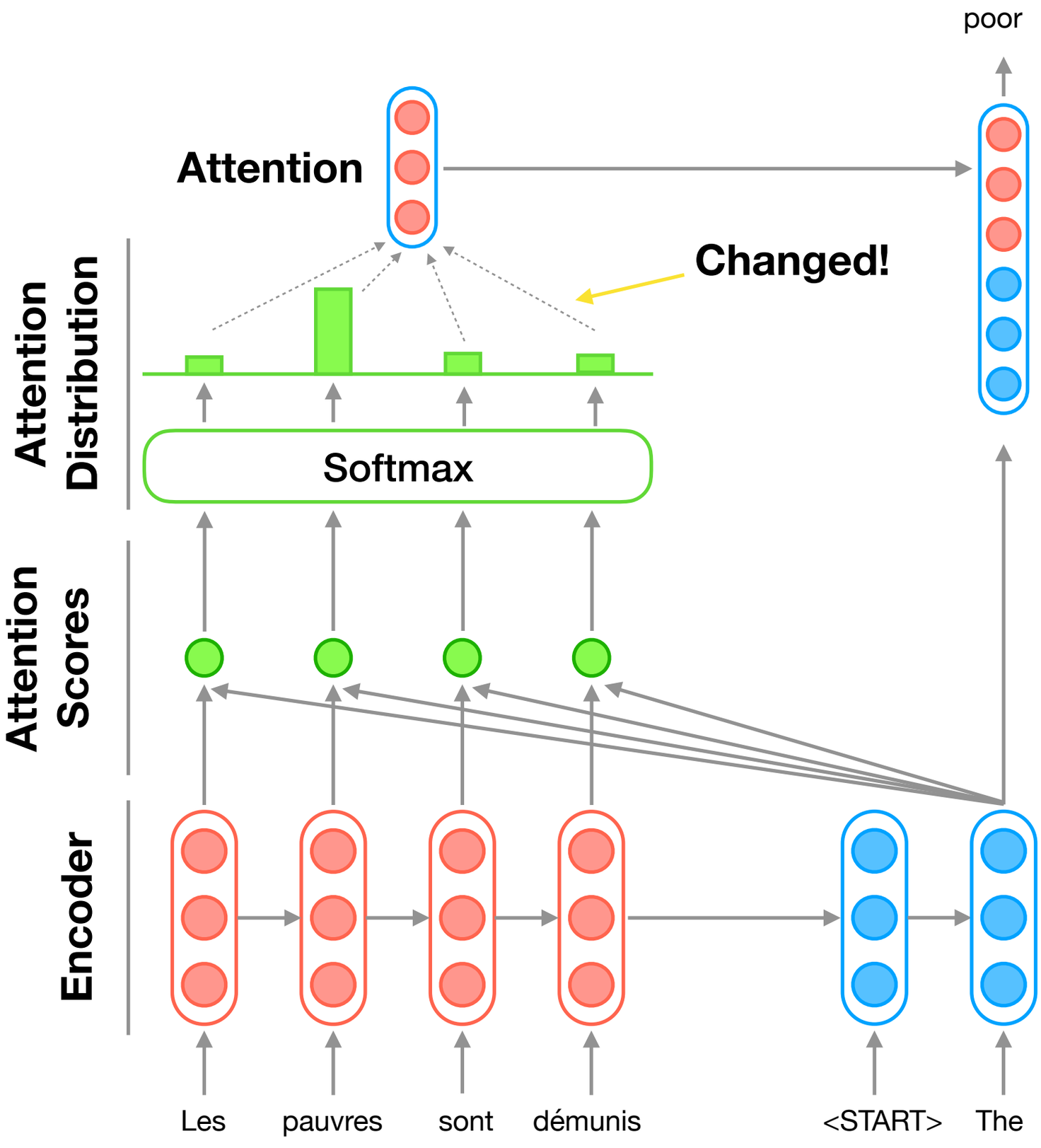
Attention


Attention
Concat
Attention Distribution
Softmax
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>

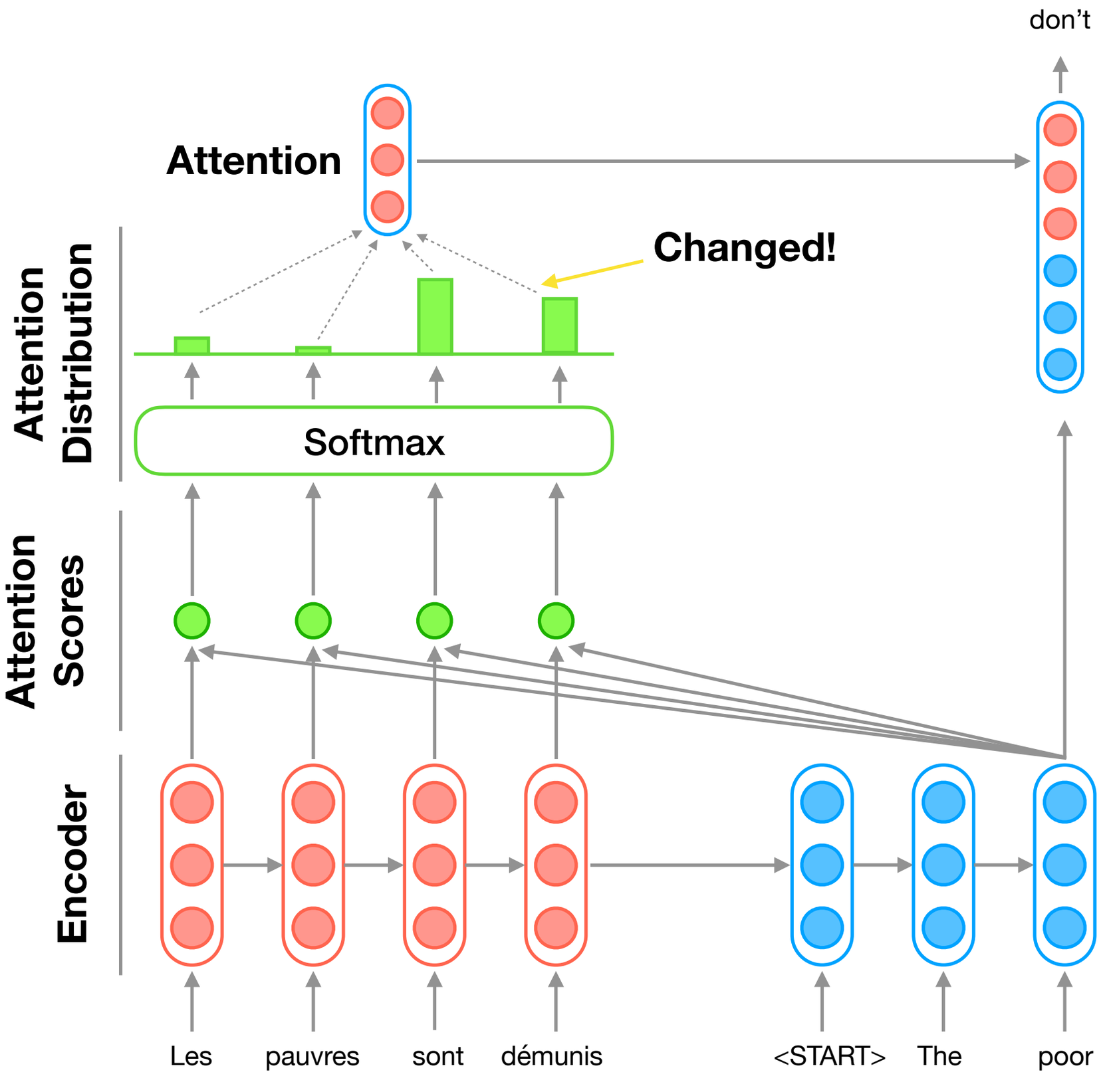
Attention


The
Attention
Concat
Attention Distribution
Softmax
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>

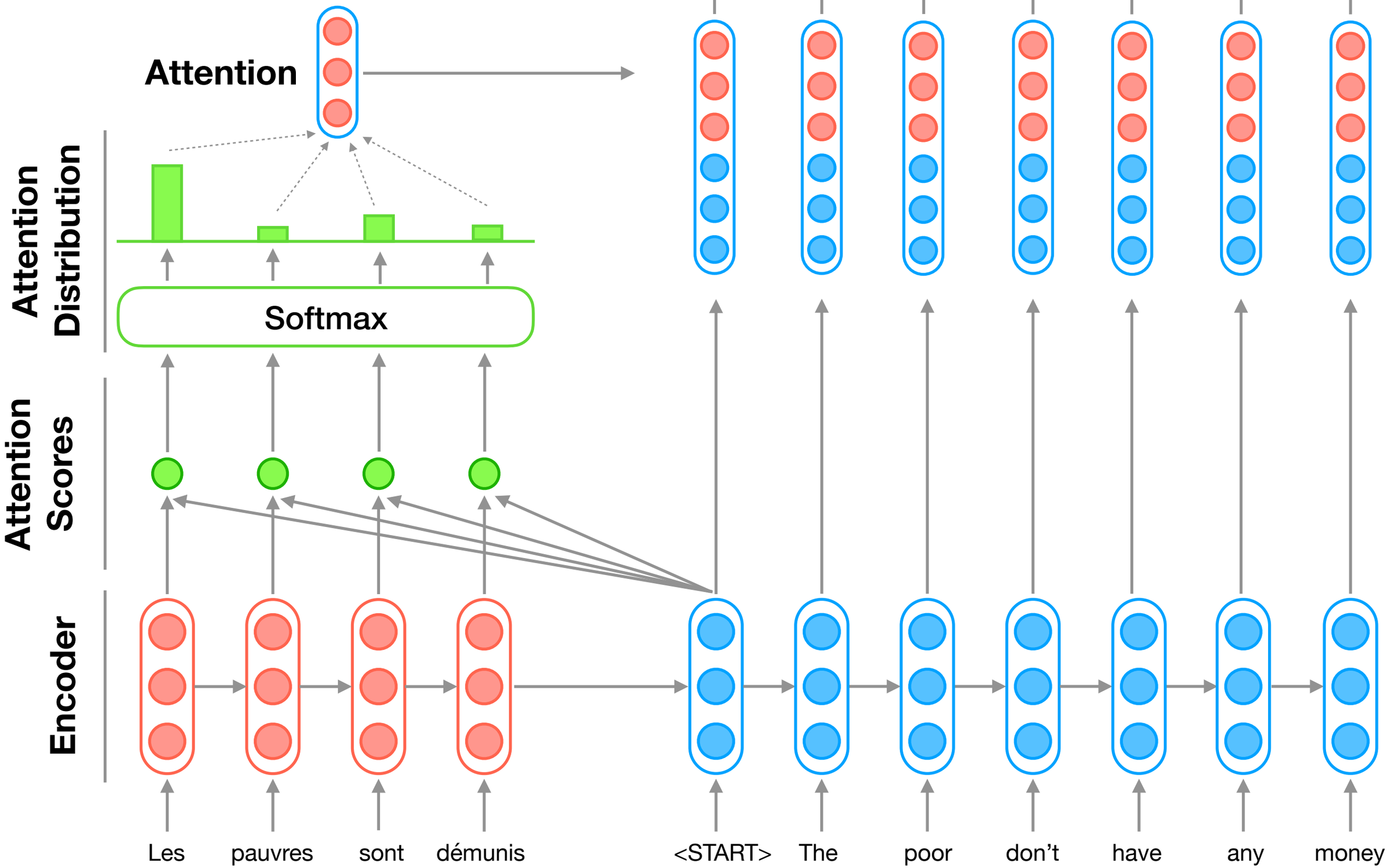
Attention


poor
Attention
Changed!
Attention Distribution
Softmax
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>
The

Attention


don’t
Attention
Changed!
Attention Distribution
Softmax
Attention Scores
Encoder
Les
pauvres
sont
démunis
<START>
The
poor

Attention


Attention
Softmax
Attention Distribution
Attention Scores
Encoder
The
poor
don’t
have
any
money
<END>
Les
pauvres
sont
démunis
<START>
The
poor
don’t
have
any
money

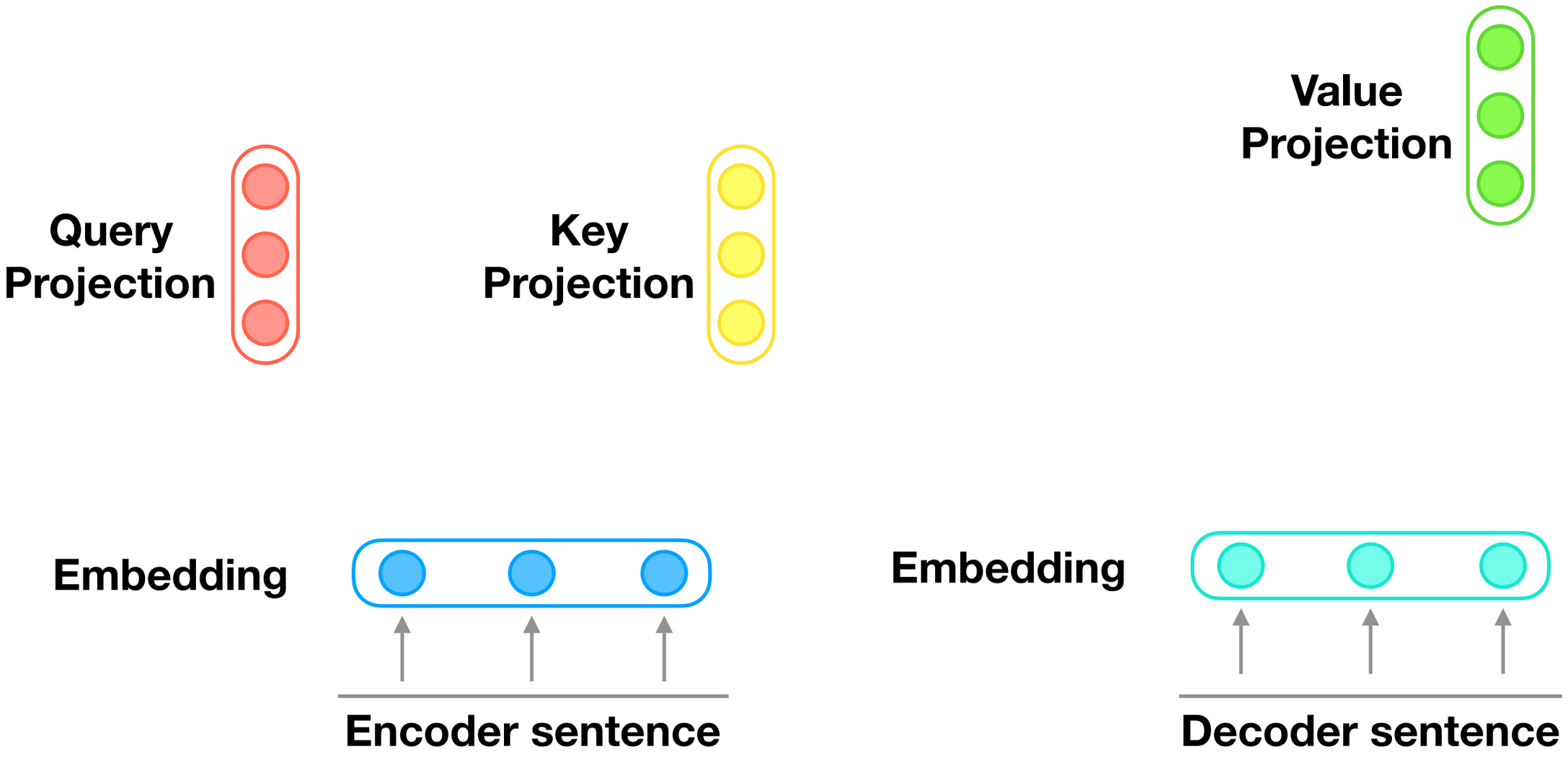
Attention

Encoder sentence

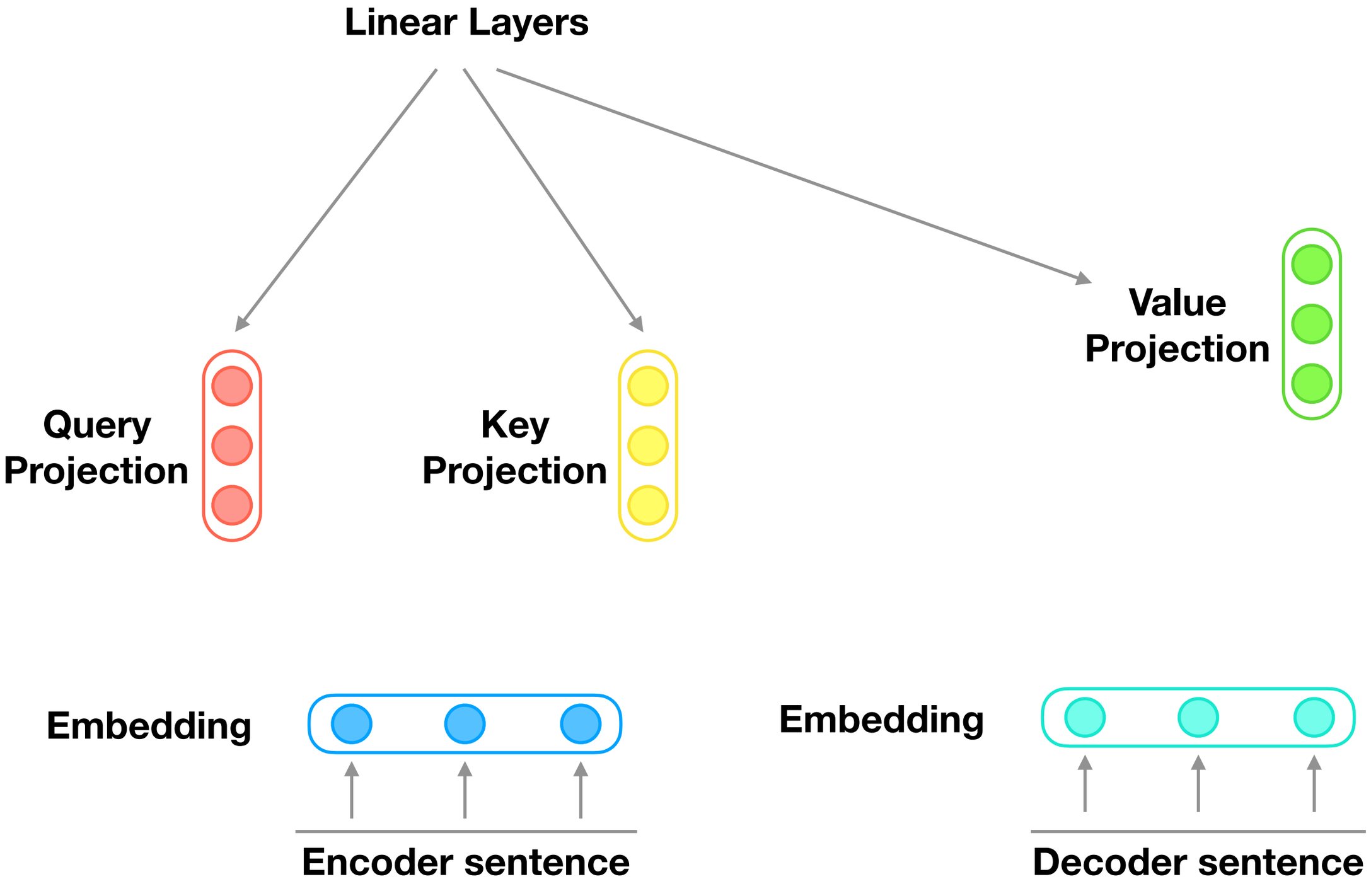
Decoder sentence

Attention

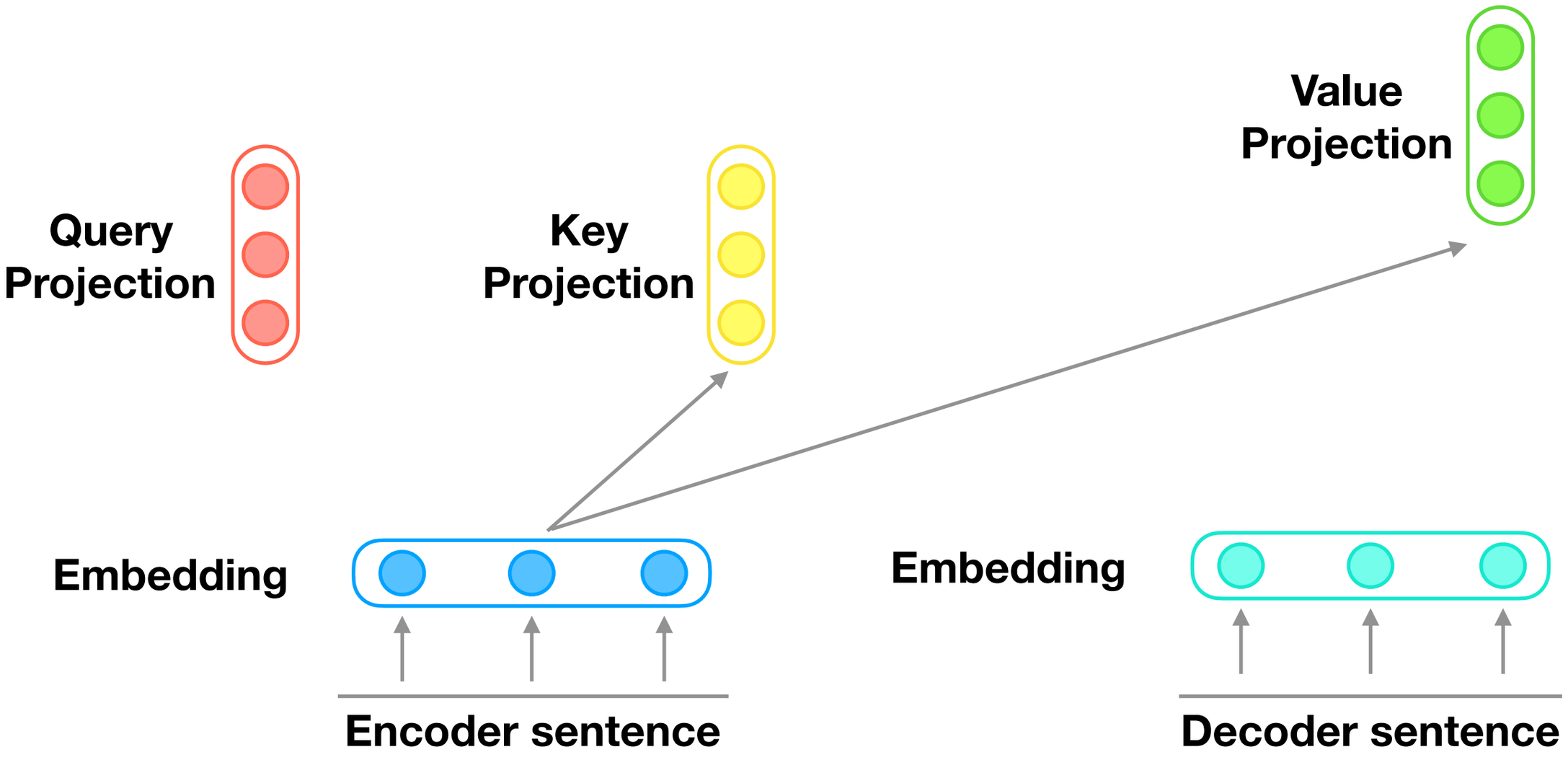
Attention


Value
Projection
Query
Projection
Key
Projection
Embedding
Encoder sentence
Embedding
Decoder sentence

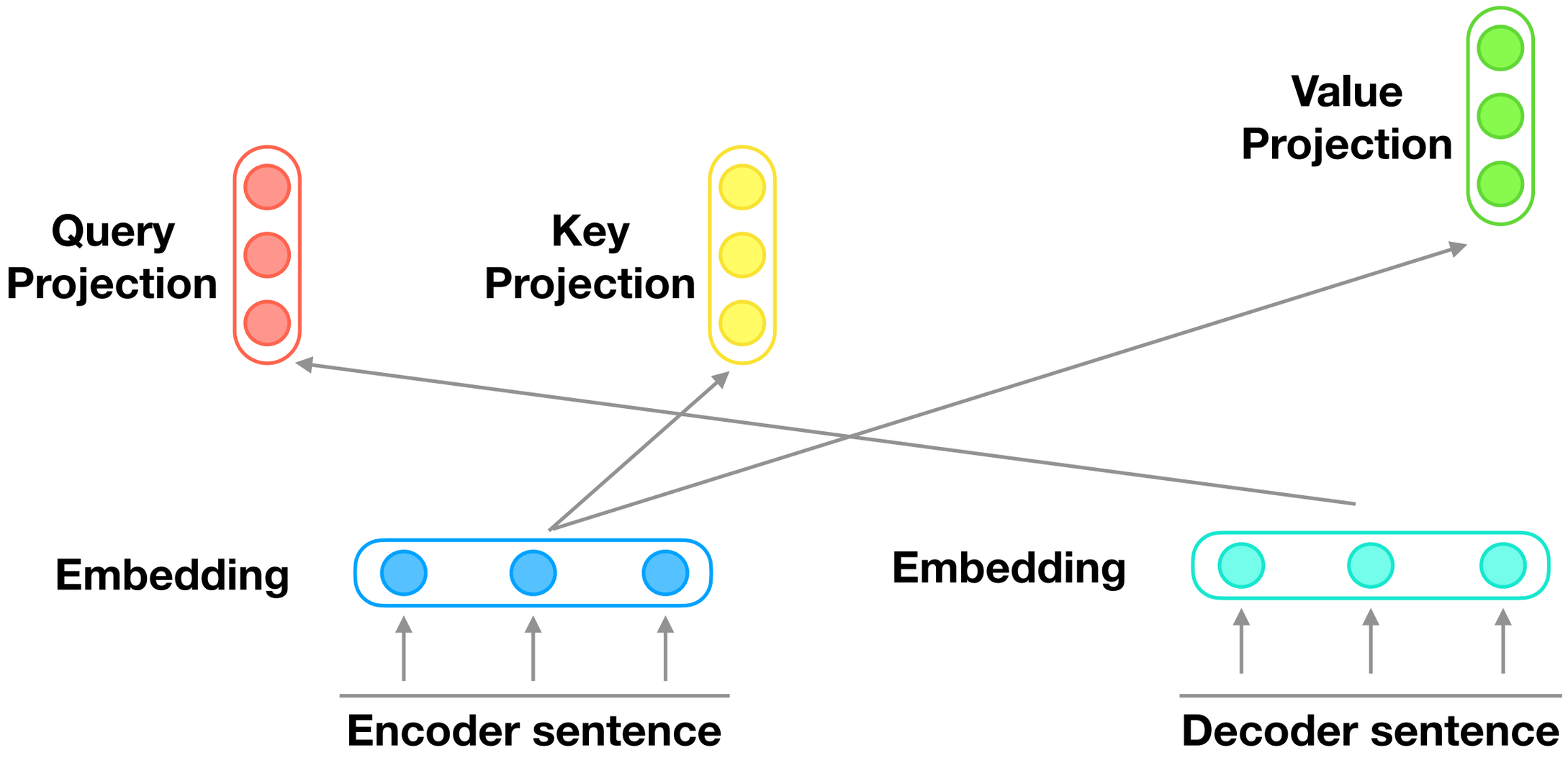
Attention


Linear Layers
Query Projection
Key Projection
Value Projection
Embedding
Encoder sentence
Embedding
Decoder sentence

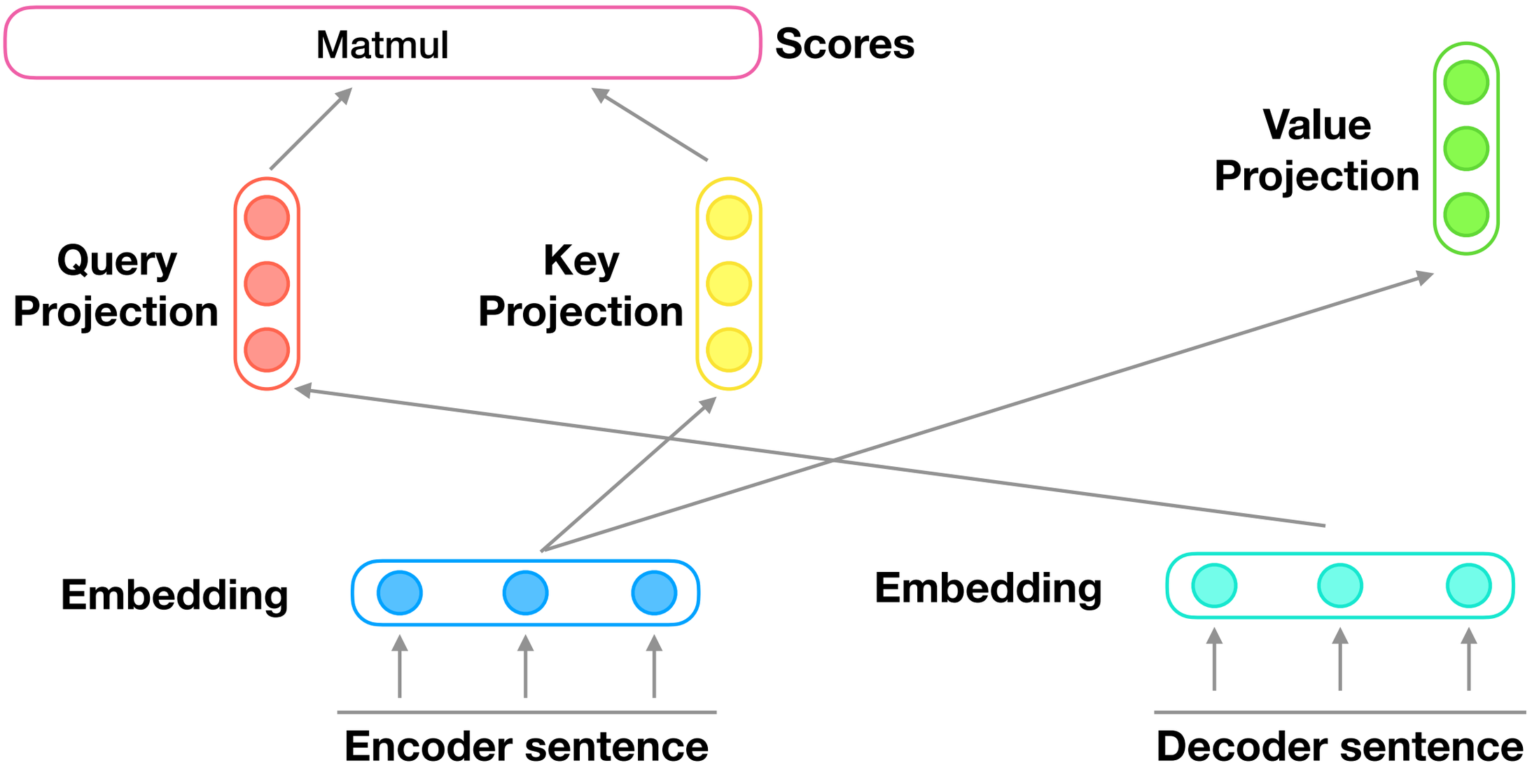
Attention


Value
Projection
Query
Projection
Key
Projection
Embedding
Embedding
Encoder sentence
Decoder sentence

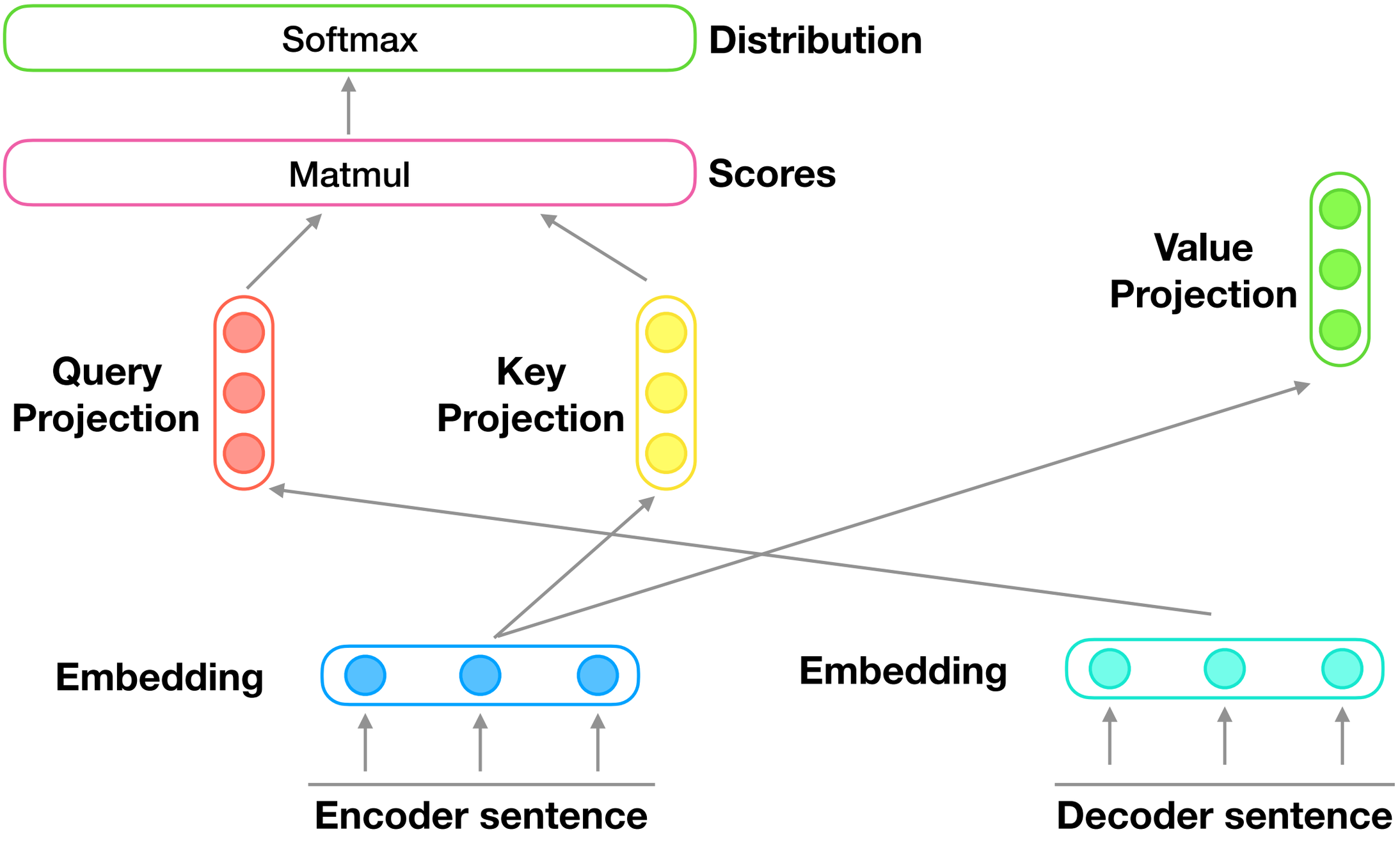
Attention


Value
Projection
Query
Projection
Key
Projection
Embedding
Embedding
Encoder sentence
Decoder sentence

Attention

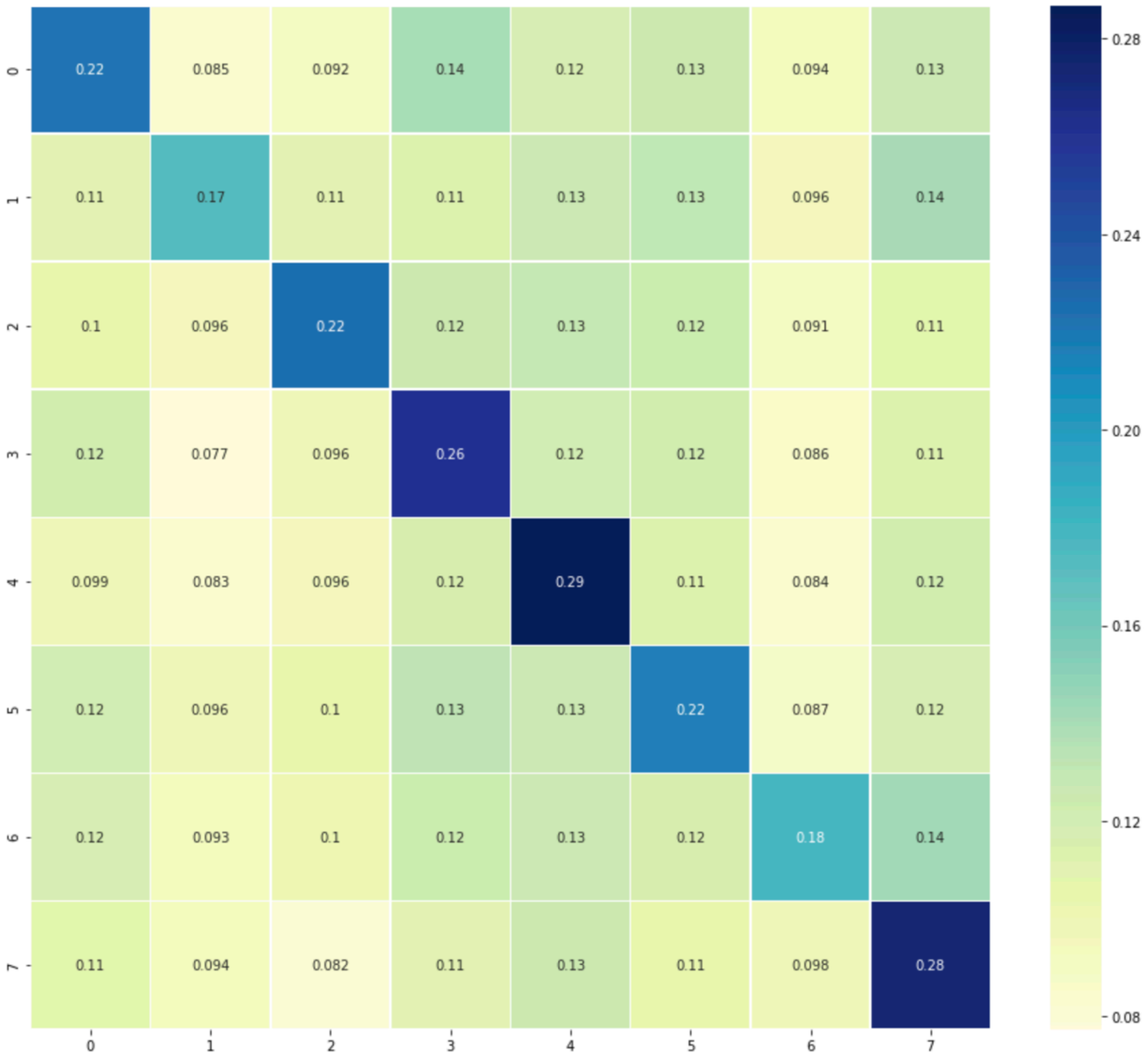

Matmul
Scores
Query Projection
Key Projection
Value Projection
Embedding
Embedding
Encoder sentence
Decoder sentence

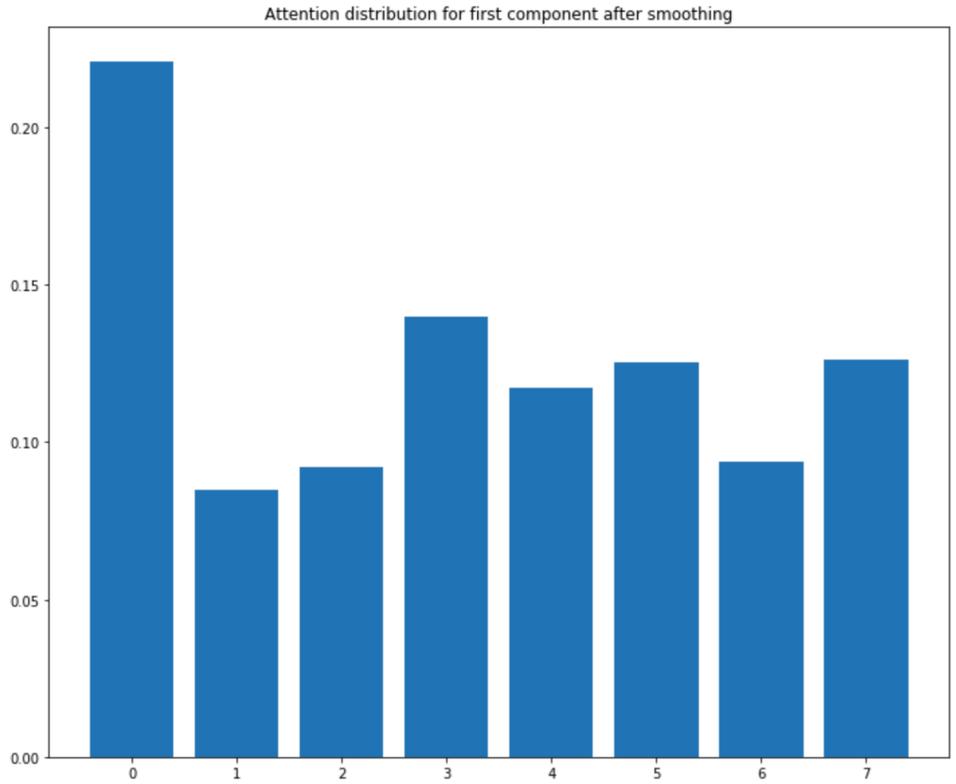
Attention


Softmax
Distribution
Matmul
Scores
Query Projection
Key Projection
Value Projection
Embedding
Embedding
Encoder sentence
Decoder sentence

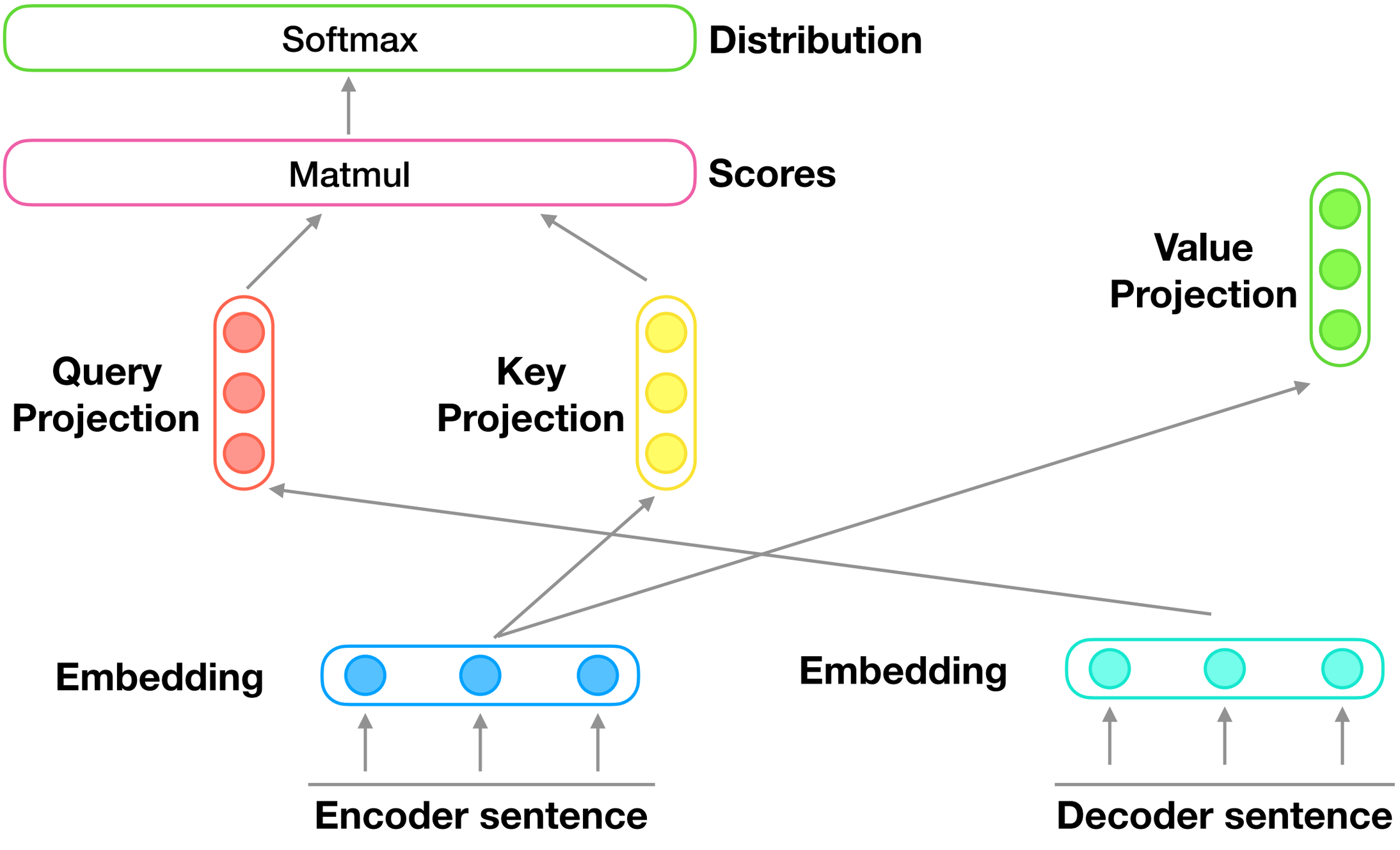
Attention

Attention Distribution
Sequence length = 8

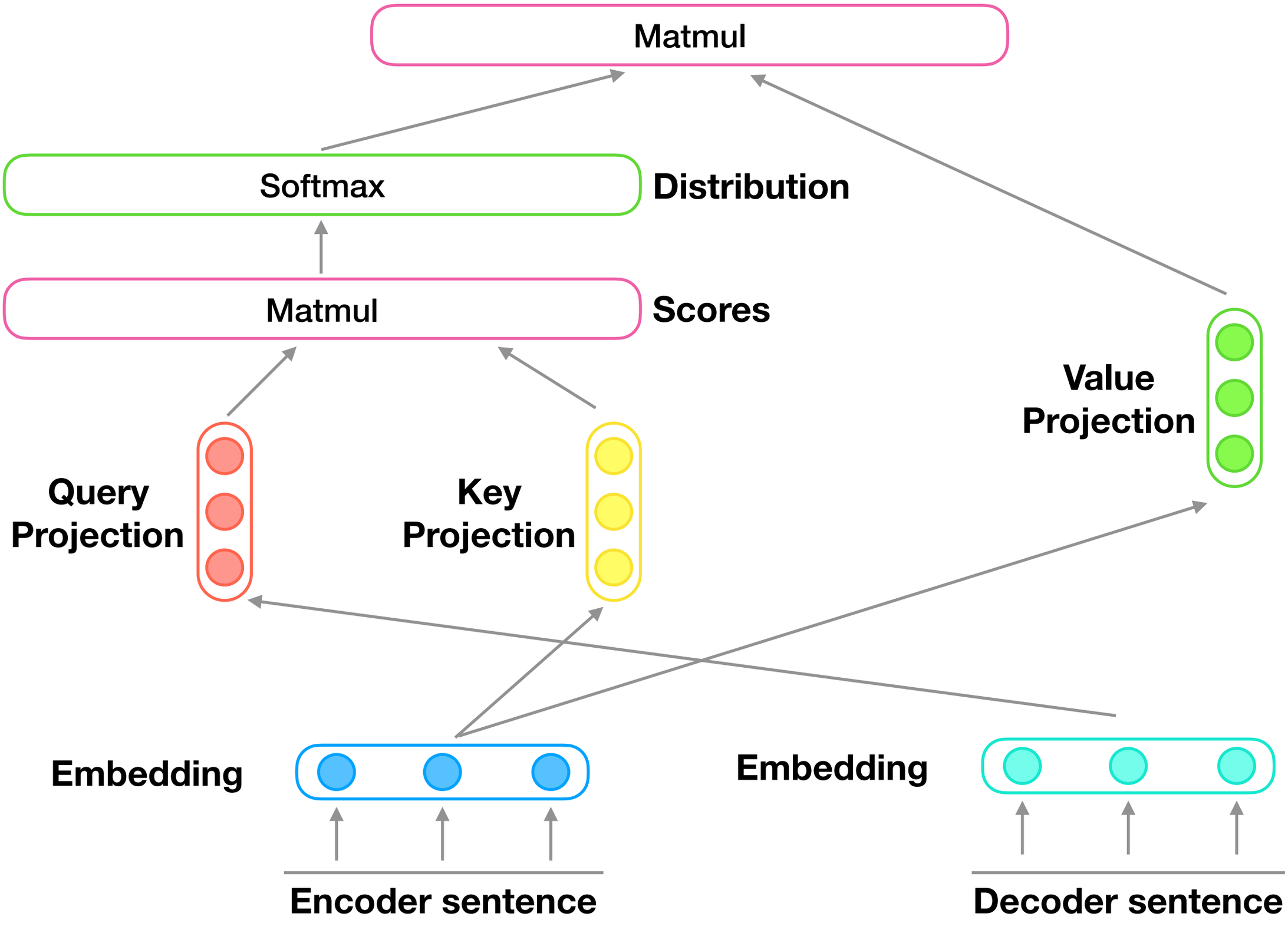




Attention


Softmax
Distribution
Matmul
Scores
Query Projection
Key Projection
Value Projection
Embedding
Embedding
Encoder sentence
Decoder sentence

Attention


Matmul
Softmax
Distribution
Matmul
Scores
Value
Projection
Query
Projection
Key
Projection
Embedding
Embedding
Encoder sentence
Decoder sentence

Attention

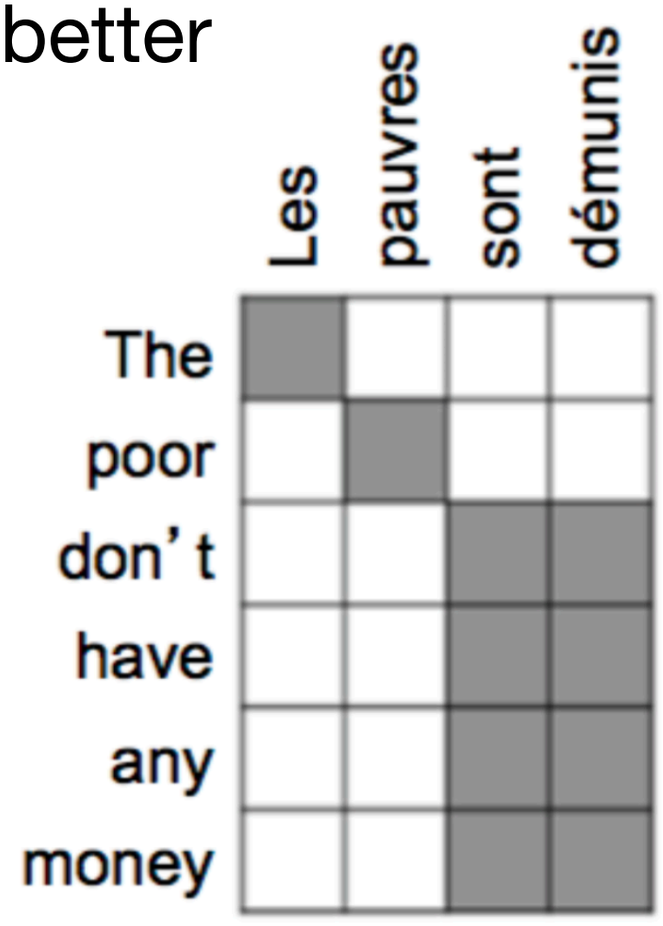
- Work with long dependencies
- Significantly improves NMT performance
- Solves bottleneck problem

Attention

- Work with long dependencies
- Significantly improves NMT performance
- Solves bottleneck problem
- Attention helps to do your language task better

Attention

- Work with long dependencies
- Significantly improves NMT performance
- Solves bottleneck problem
- Attention helps to do your language task better



Self-Attention

What if we apply attention to yourself?

Self-Attention

What if we apply attention to yourself?

```
attention_scores = torch.matmul(x, x.t())
```

Self-Attention

What if we apply attention to yourself?

```
attention_scores = torch.matmul(x, x.t())
```

x.shape == (sequence_length, embedding_dim)

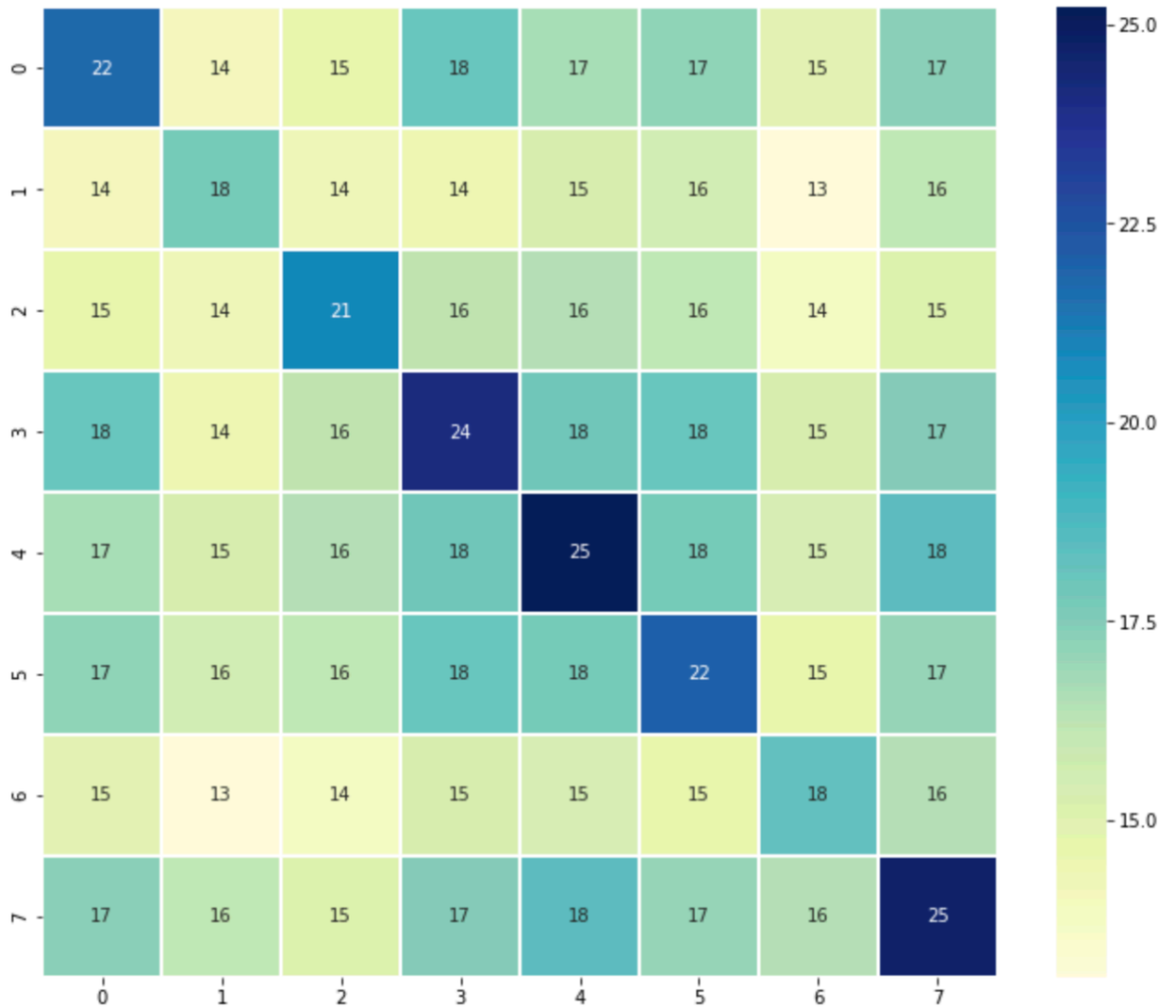
Self-Attention

What if we apply attention to yourself?

```
attention_scores = torch.matmul(x, x.t())
```

x.shape == (sequence_length, embedding_dim)

softmax(



)

Self-Attention

What if we apply attention to yourself?

```
attention_distribution = torch.nn.functional.softmax(attention_scores, dim=1)
```

x.shape == (sequence_length, embedding_dim)

Self-Attention

What if we apply attention to yourself?

```
attention_distribution = torch.nn.functional.softmax(attention_scores, dim=1)
```

x.shape == (sequence_length, embedding_dim)

→ **Apply weighted sum**

Self-Attention

What if we apply attention to yourself?

```
x_pred = torch.matmul(attention_distribution, x)
```

```
x.mean(), x.std()
```

```
(tensor(0.5047), tensor(0.2943))
```

```
x_pred.mean(), x_pred.std()
```

```
(tensor(0.5099), tensor(0.2739))
```

```
(x_pred - x).mean(), (x_pred - x).std()
```

```
(tensor(0.0052), tensor(0.0467))
```

Self-Attention

What if we apply attention to yourself?

```
x_pred = torch.matmul(attention_distribution, x)
```

```
x.mean(), x.std()
```

```
(tensor(0.5047), tensor(0.2943))
```

```
x_pred.mean(), x_pred.std()
```

```
(tensor(0.5099), tensor(0.2739))
```

```
(x_pred - x).mean(), (x_pred - x).std()
```

```
(tensor(0.0052), tensor(0.0467))
```

How we can do attention learnable?

Self-Attention

What if we apply attention to yourself?

```
x_pred = torch.matmul(attention_distribution, x)
```

```
x.mean(), x.std()
```

```
(tensor(0.5047), tensor(0.2943))
```

```
x_pred.mean(), x_pred.std()
```

```
(tensor(0.5099), tensor(0.2739))
```

```
(x_pred - x).mean(), (x_pred - x).std()
```

```
(tensor(0.0052), tensor(0.0467))
```

How we can do attention learnable?

Do linear projections!

Self-Attention

How we can do attention learnable?

Do linear projections!

```
query_projection = torch.nn.Linear(in_features=x.shape[-1], out_features=x.shape[-1])
key_projection = torch.nn.Linear(in_features=x.shape[-1], out_features=x.shape[-1])
value_projection = torch.nn.Linear(in_features=x.shape[-1], out_features=x.shape[-1])

query = query_projection(x)
key = key_projection(x)
value = value_projection(x)

attention_scores = torch.matmul(query, key.t())
attention_distribution = torch.nn.functional.softmax(attention_scores, dim=1)

x_pred = torch.matmul(attention_distribution, value)
```

```
x.mean(), x.std()
```

```
(tensor(0.5047), tensor(0.2943))
```

```
x_pred.mean(), x_pred.std()
```

```
(tensor(-0.0542, grad_fn=<MeanBackward1>),
 tensor(0.2679, grad_fn=<StdBackward0>))
```

```
(x_pred - x).mean(), (x_pred - x).std()
```

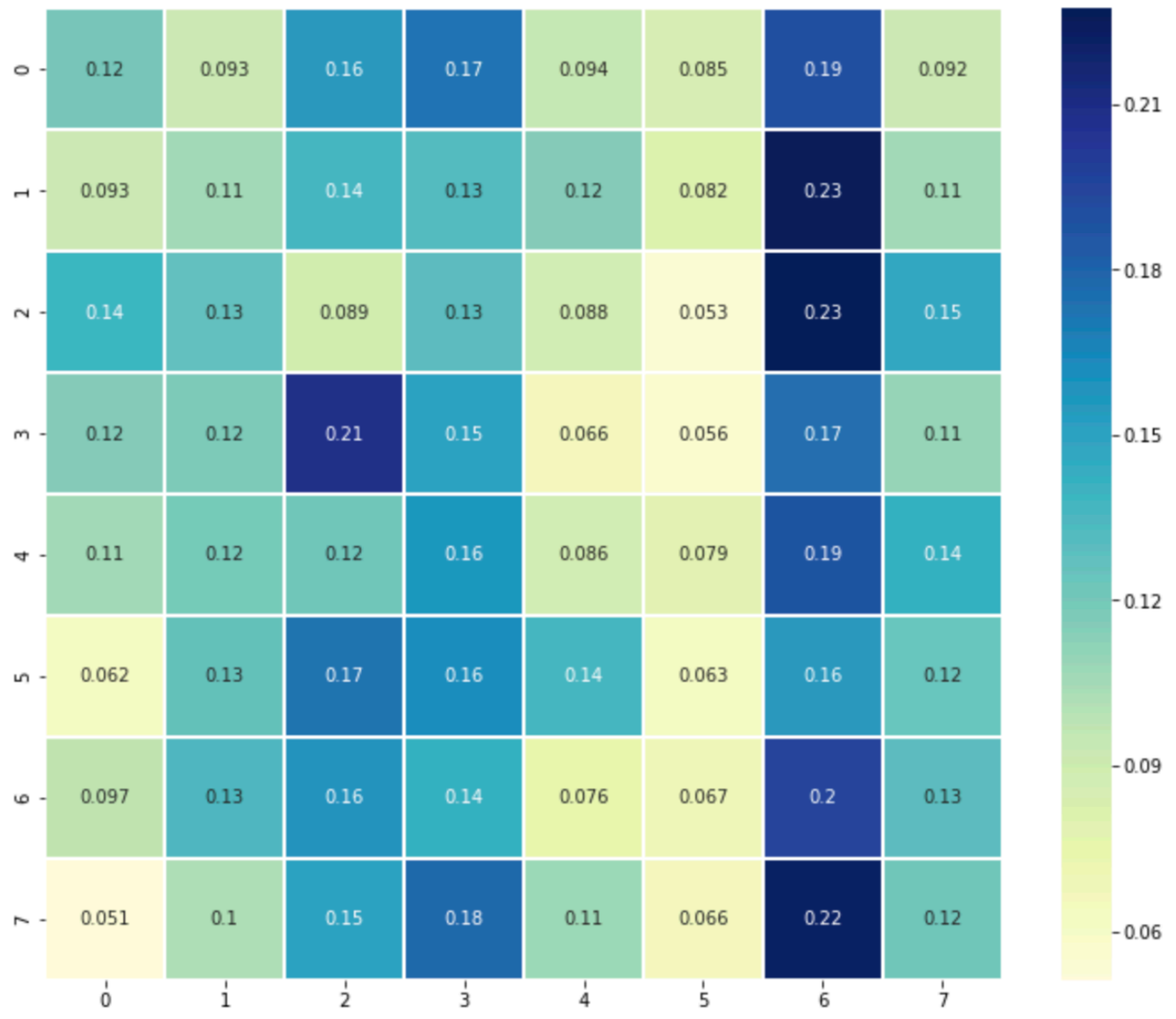
```
(tensor(-0.5589, grad_fn=<MeanBackward1>),
 tensor(0.4147, grad_fn=<StdBackward0>))
```

Self-Attention

How we can do attention learnable?

Do linear projections!

Attention Distribution



Without training

Scaled Dot-Product Attention

Scaling

Scaled Dot-Product Attention

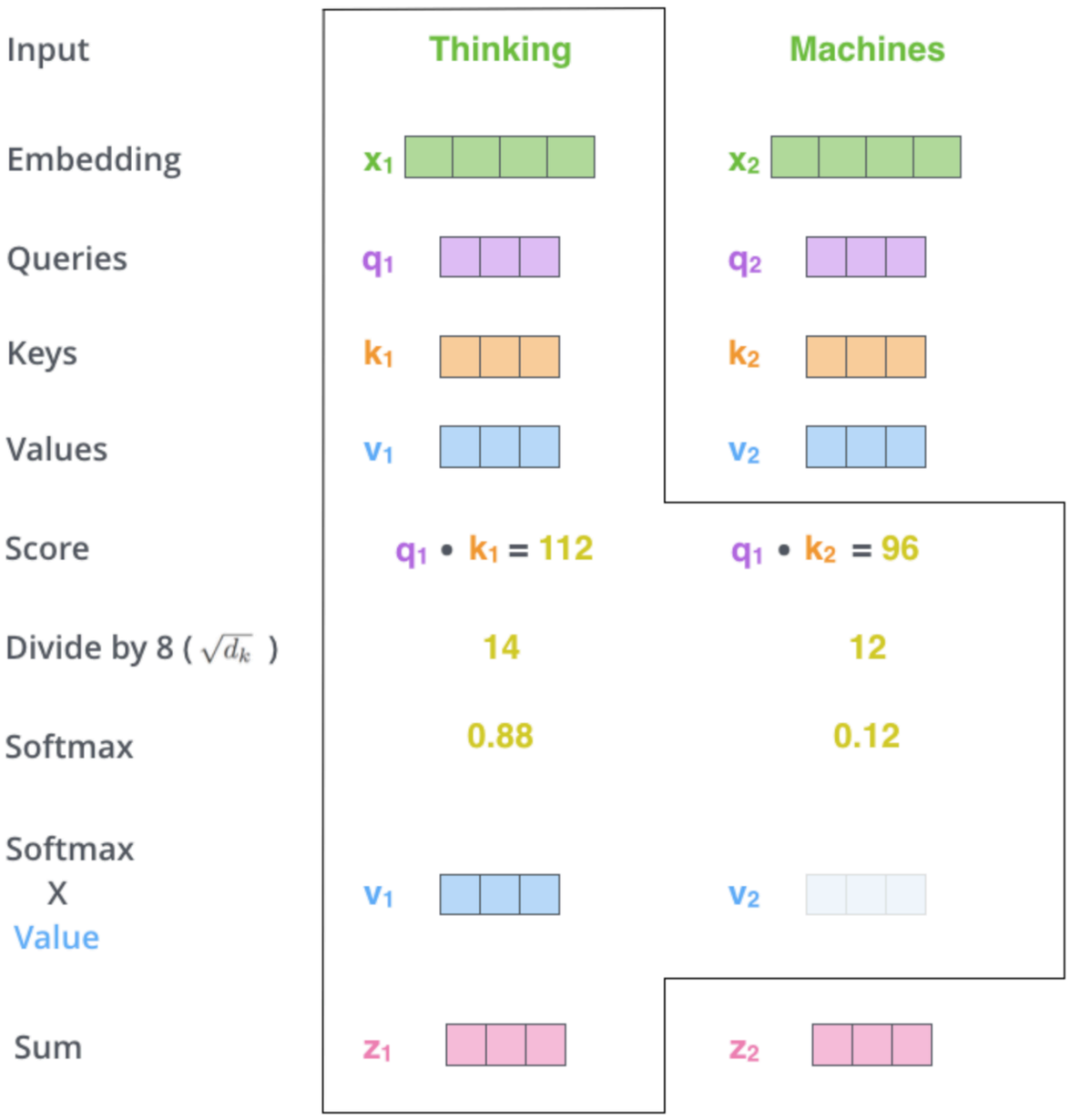
Scaling

Smoothing for faster convergence

Scaled Dot-Product Attention

Scaling

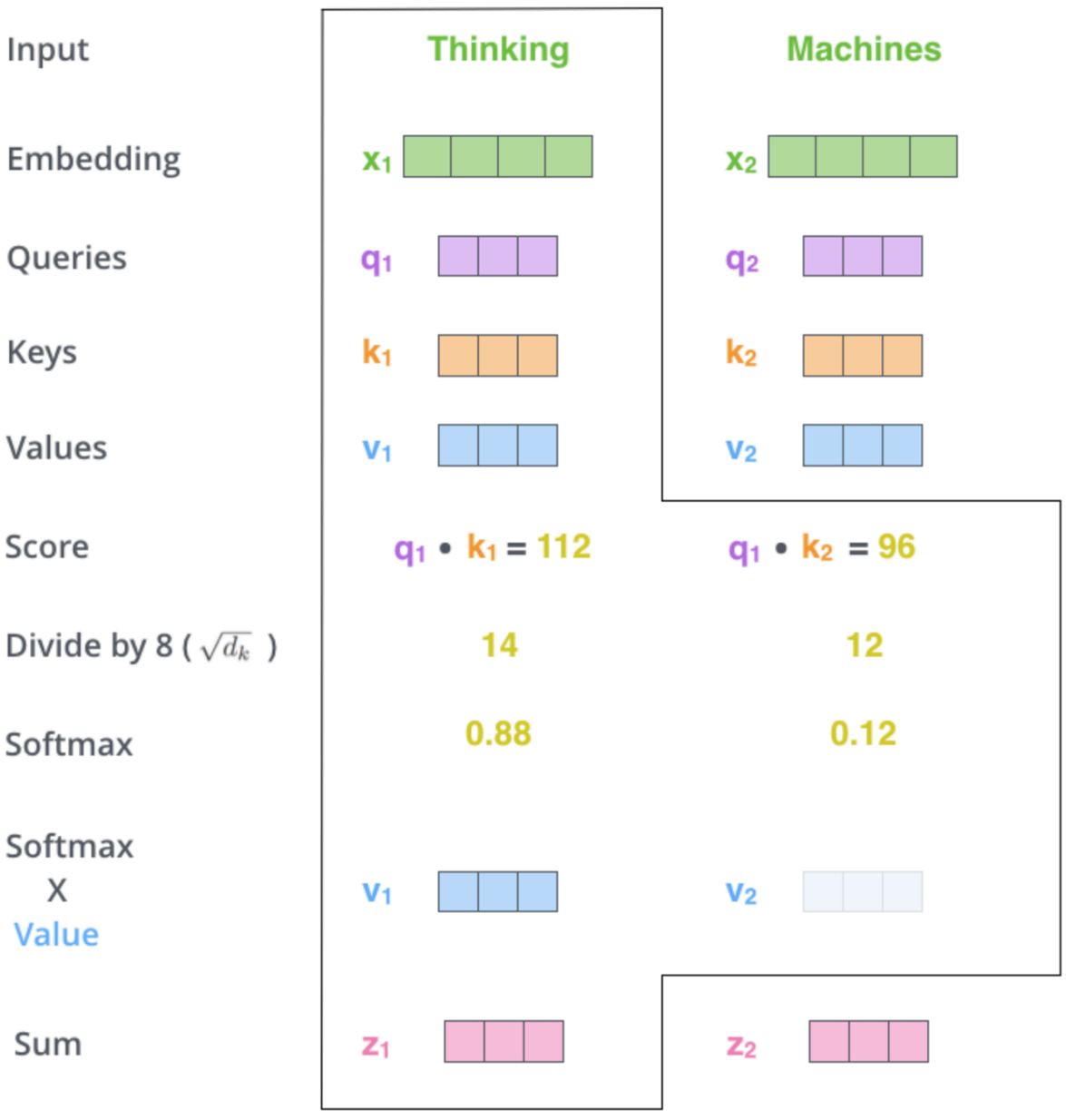
Smoothing for faster convergence

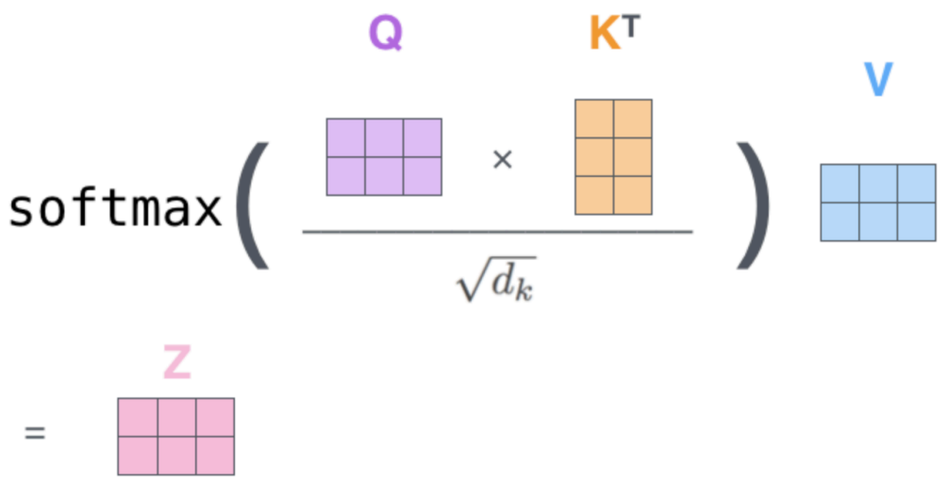


Scaled Dot-Product Attention

Scaling

Smoothing for faster convergence





Scaled Dot-Product Attention

Scaling

Smoothing for faster convergence

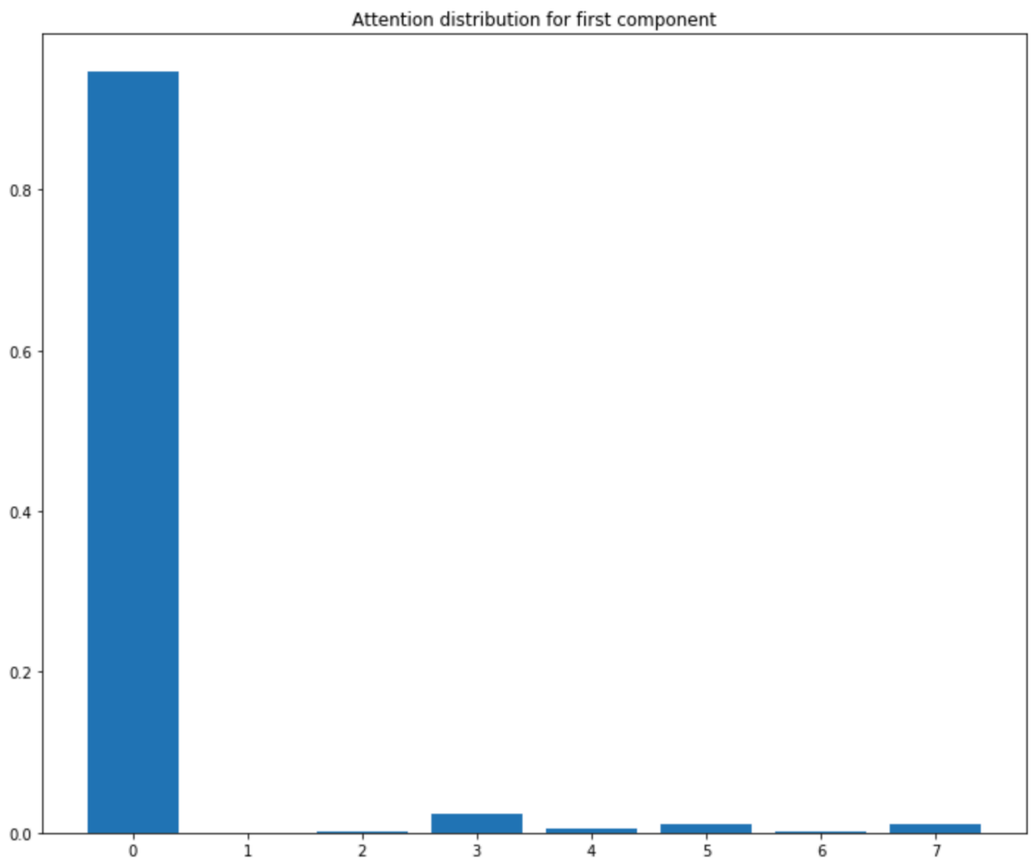
Attention distribution

Scaled Dot-Product Attention

Scaling

Smoothing for faster convergence

Attention distribution



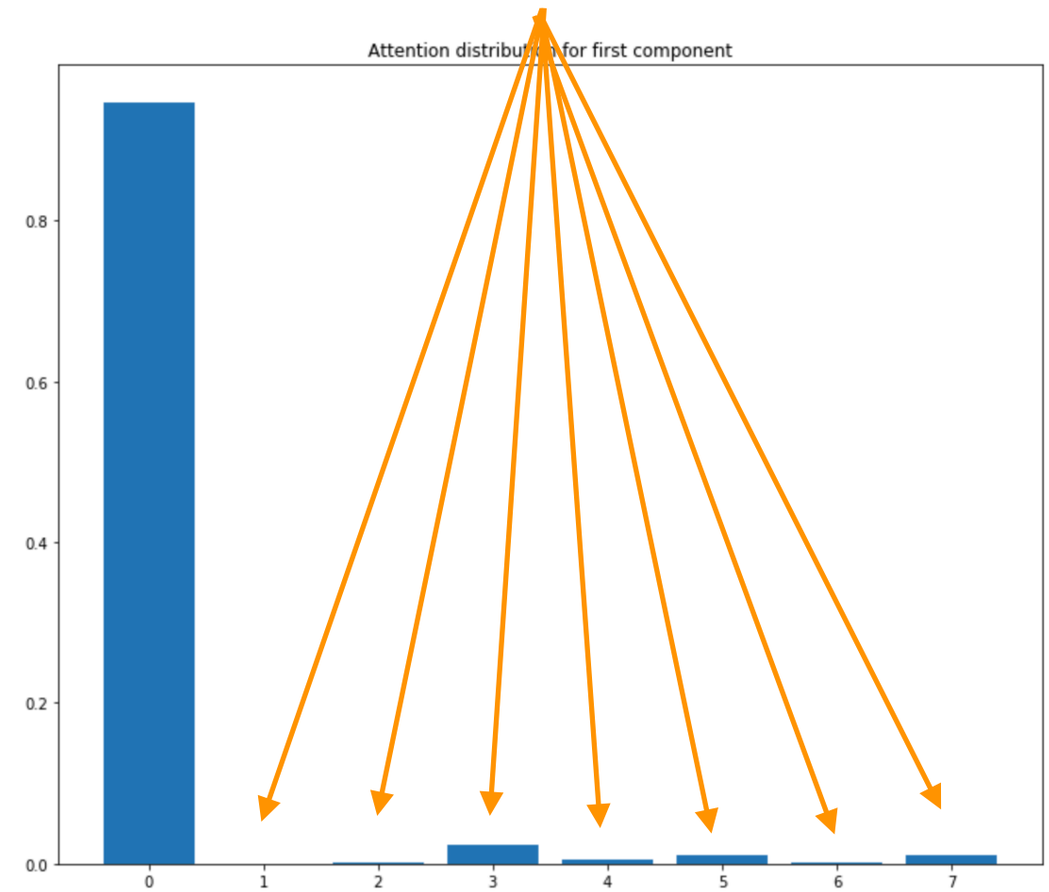
Scaled Dot-Product Attention

Scaling

Smoothing for faster convergence

Attention distribution

Extremely small gradients



Scaled Dot-Product Attention

Scaling

Smoothing for faster convergence

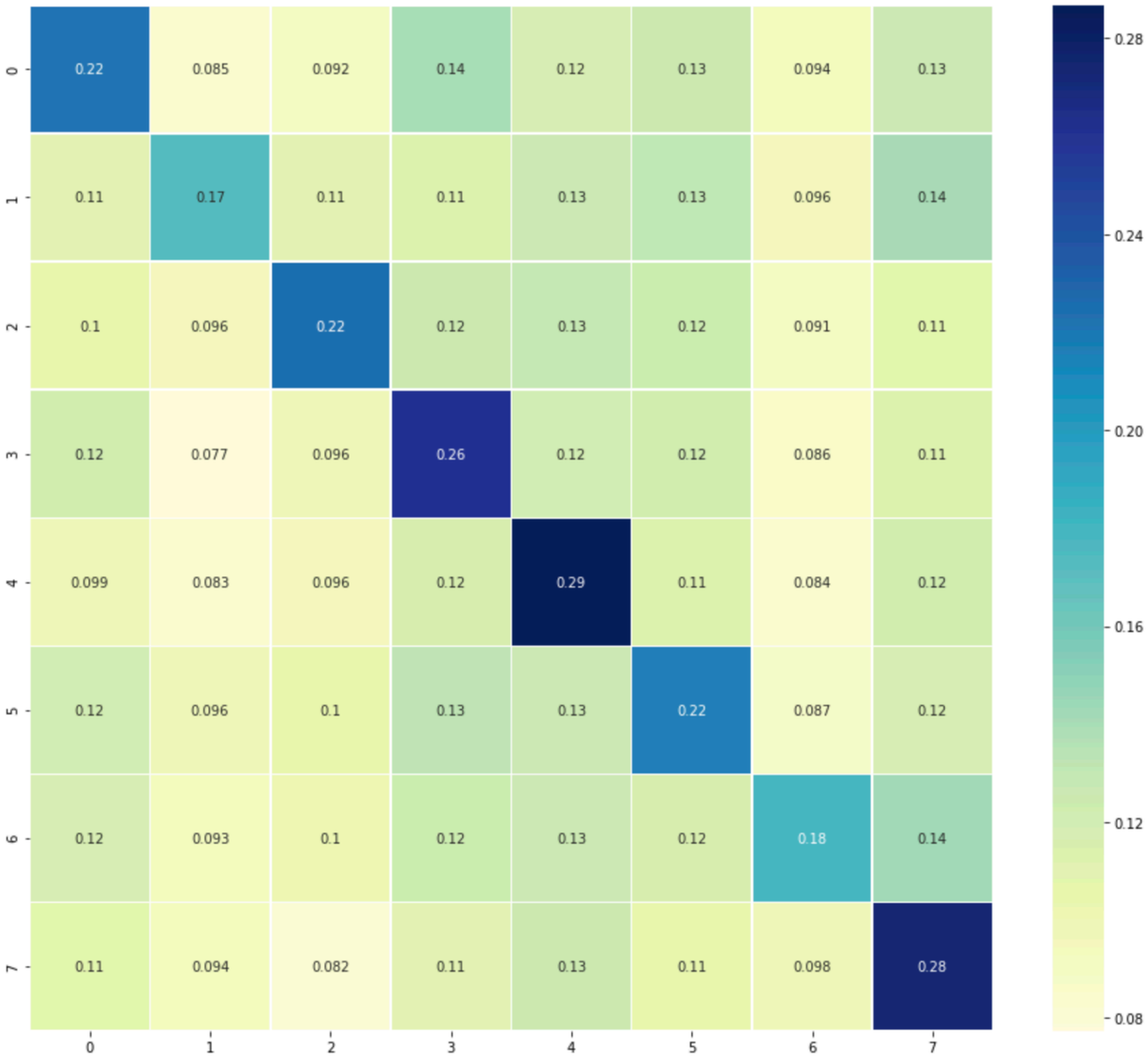
Attention distribution after scaling

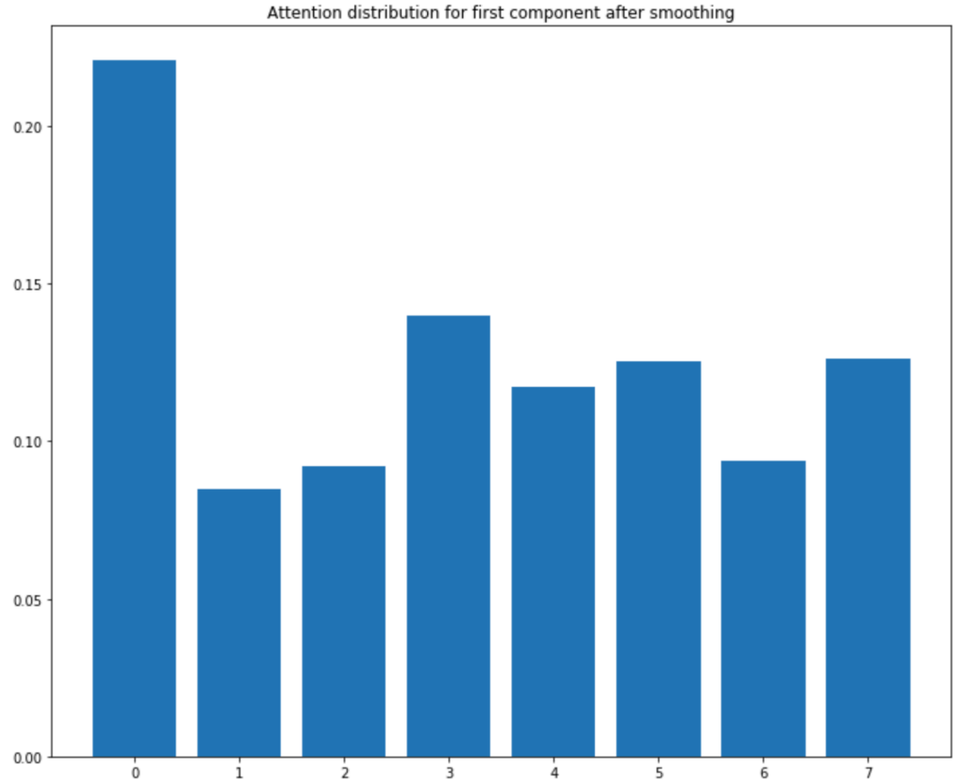
Scaled Dot-Product Attention

Scaling

Smoothing for faster convergence

Attention distribution after scaling

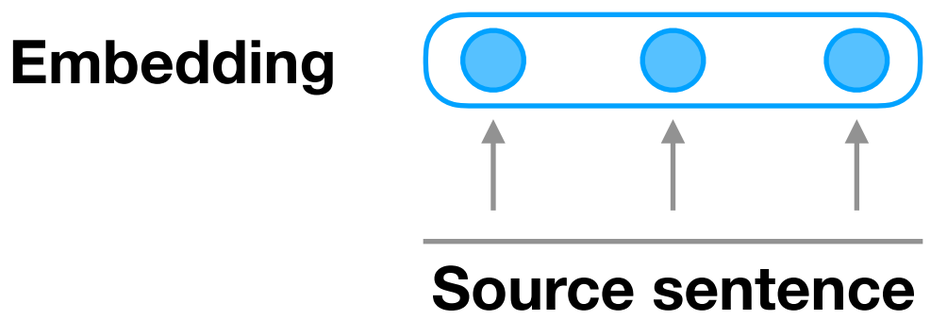




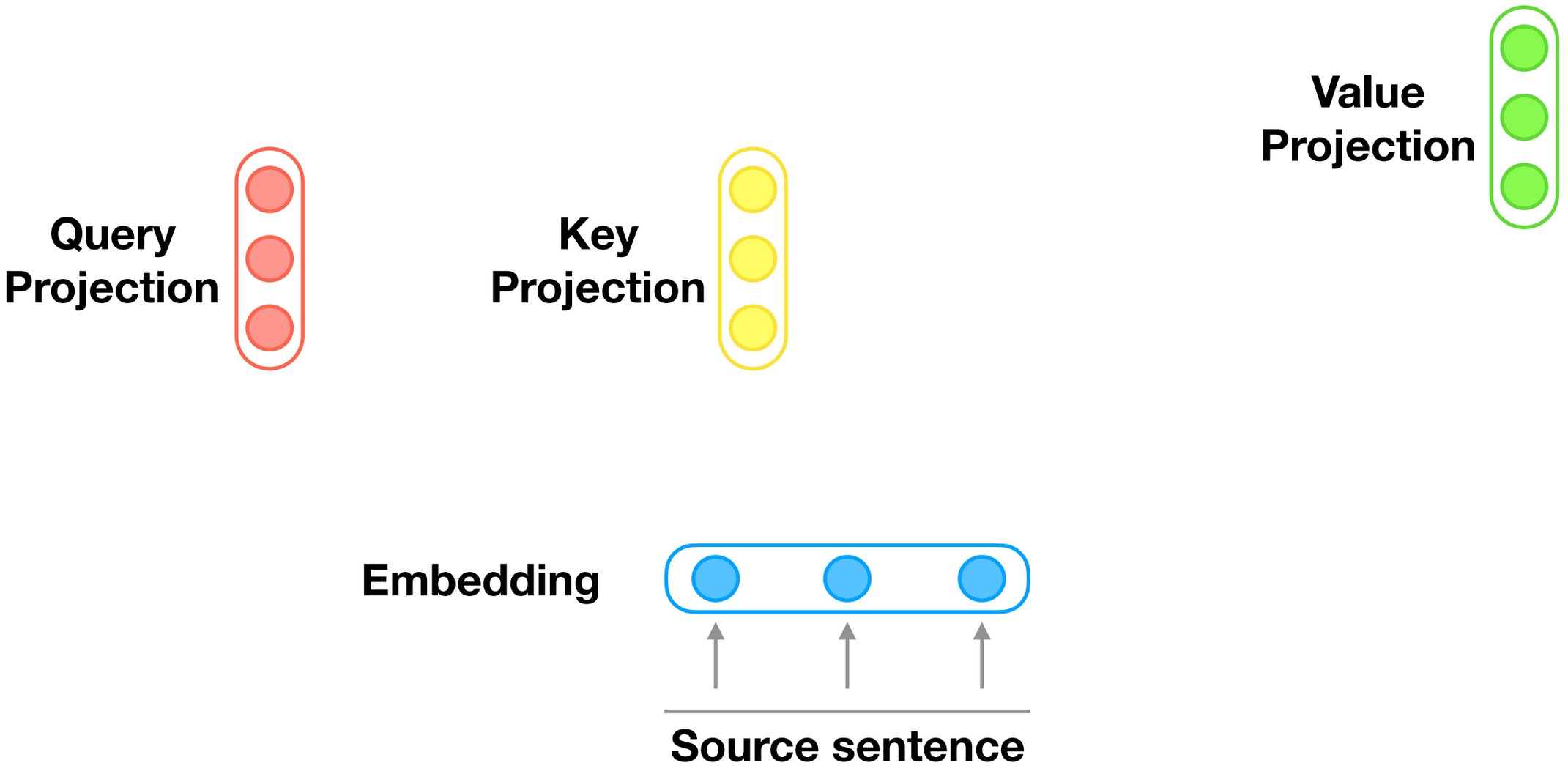
Scaled Dot-Product Attention

Source sentence

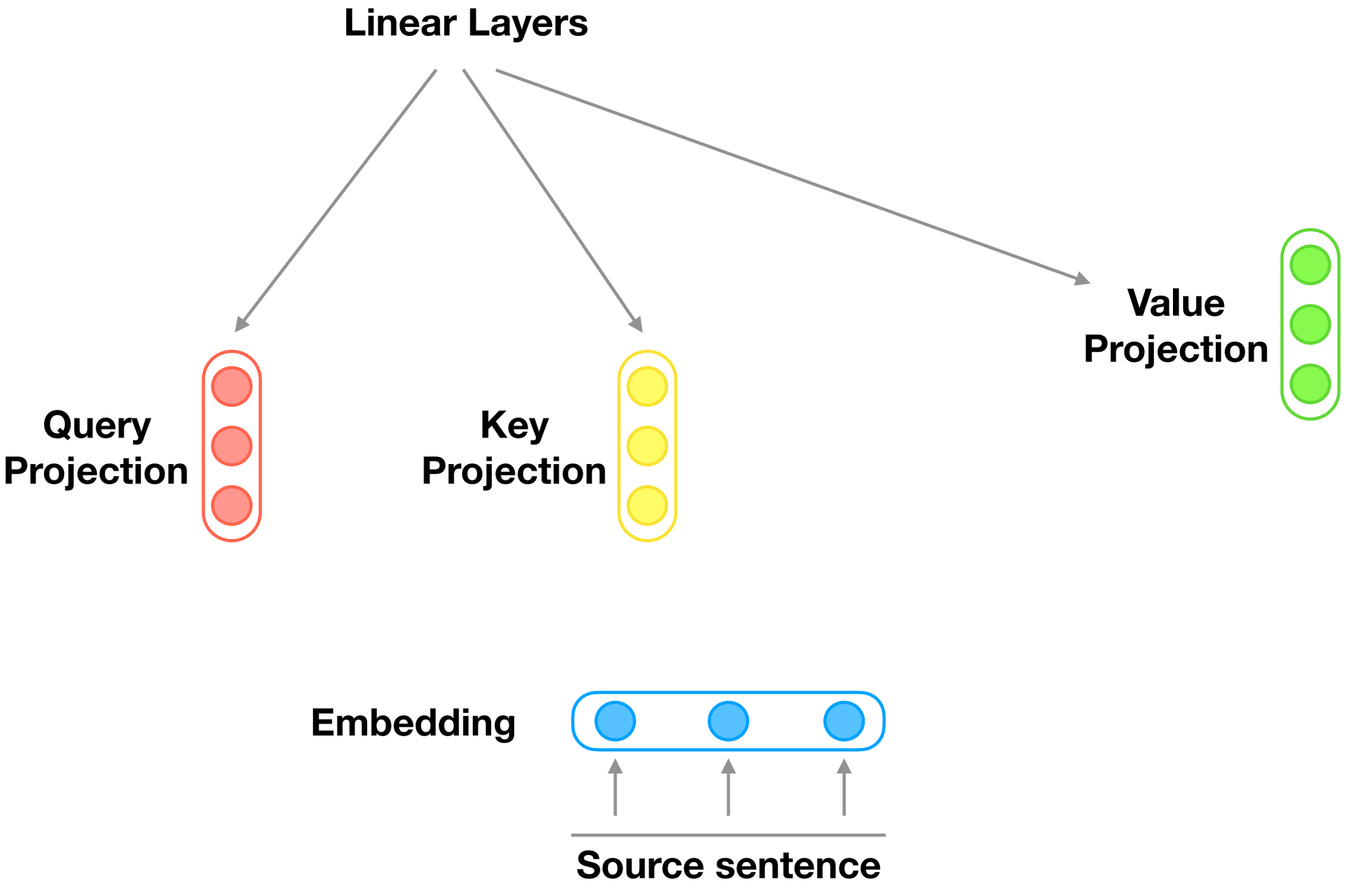
Scaled Dot-Product Attention



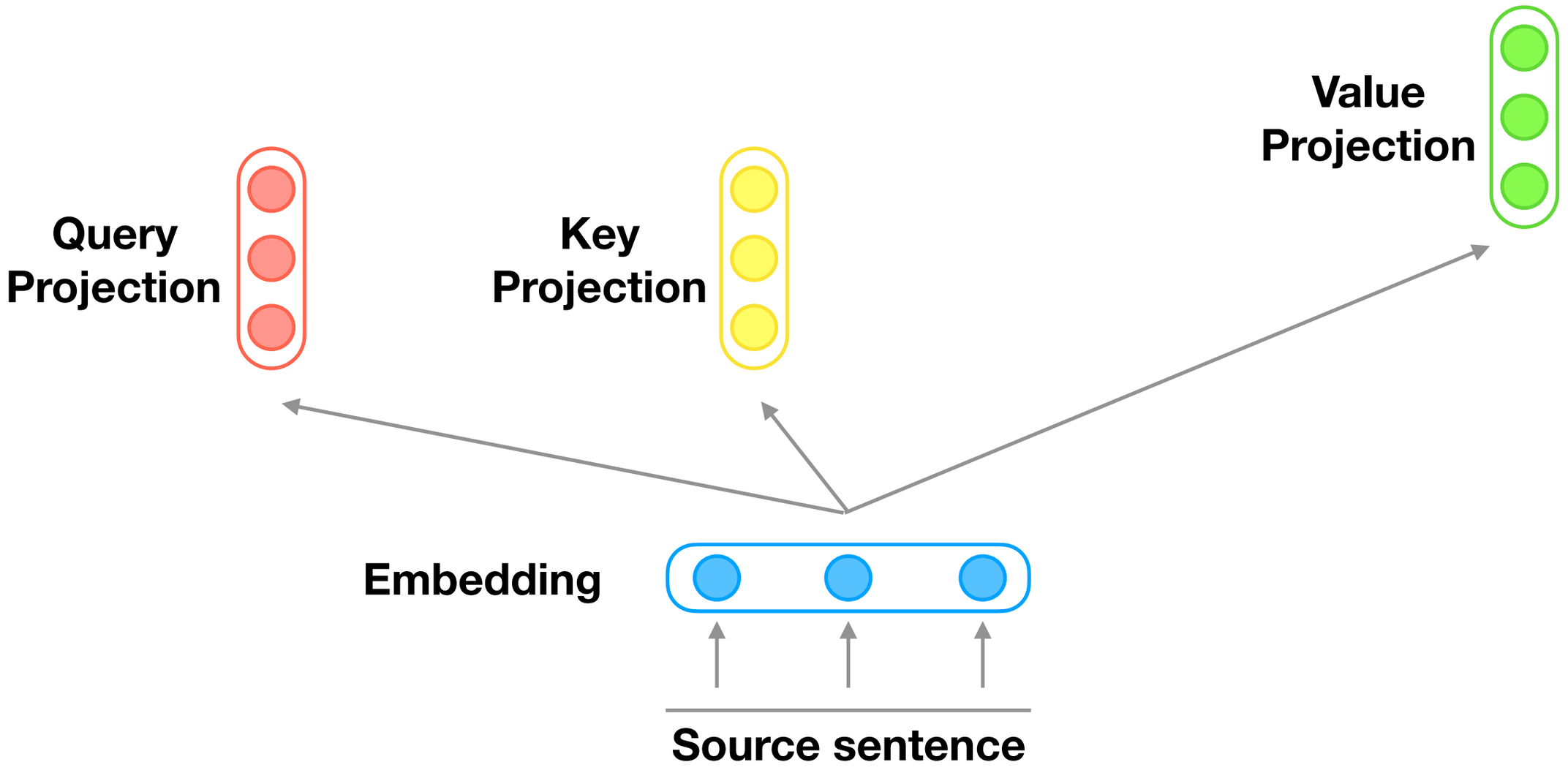
Scaled Dot-Product Attention



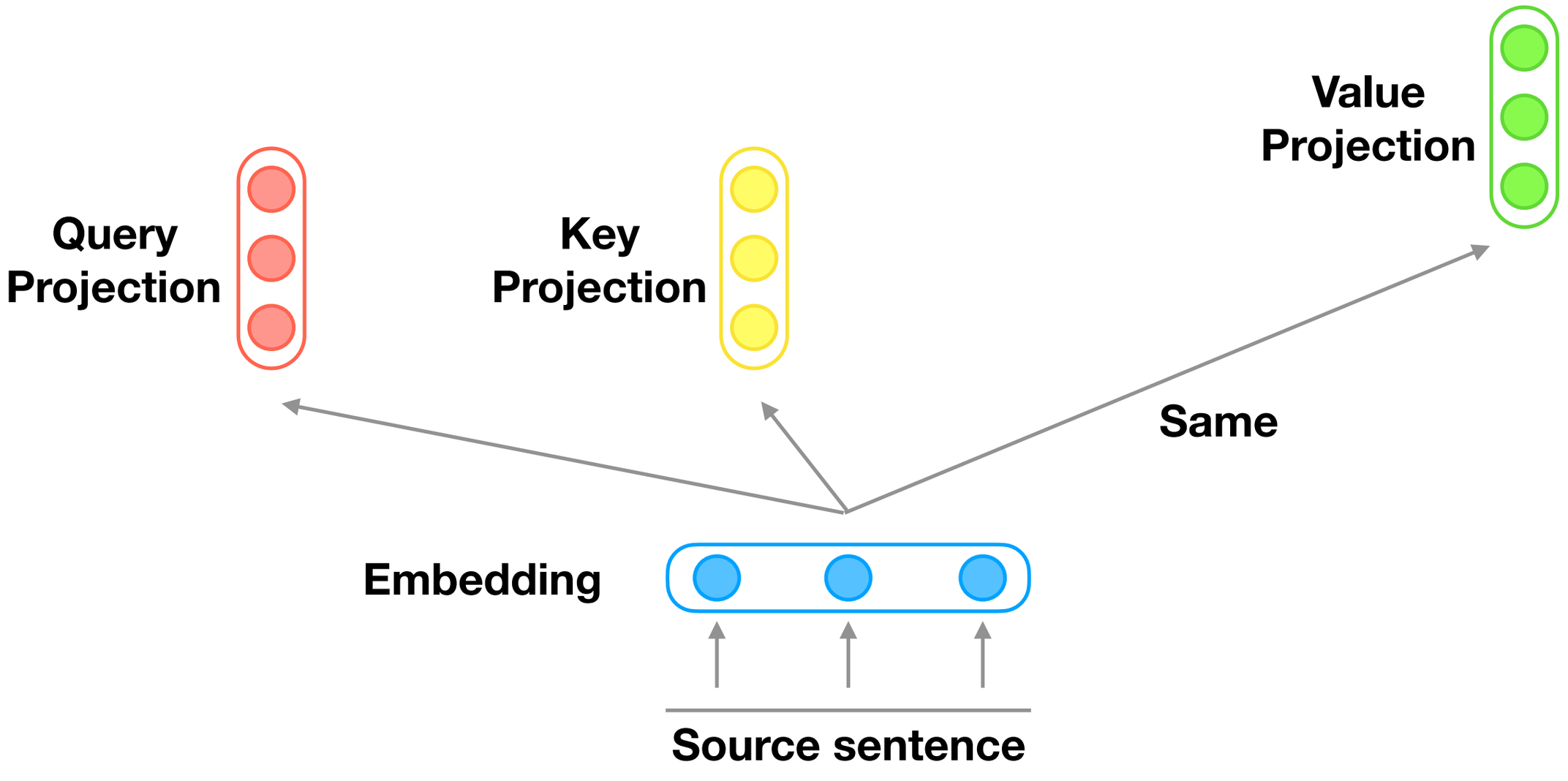
Scaled Dot-Product Attention



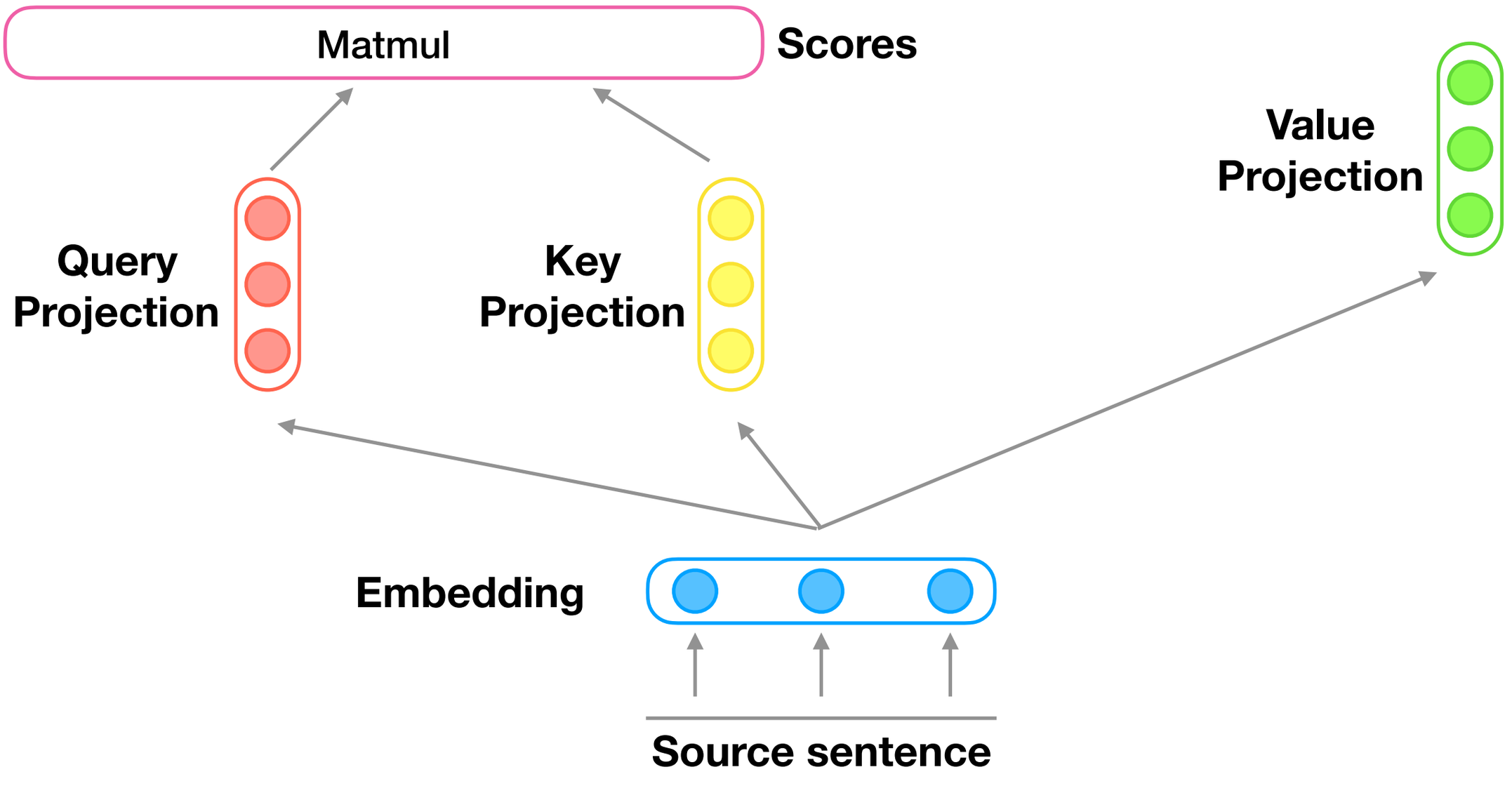
Scaled Dot-Product Attention



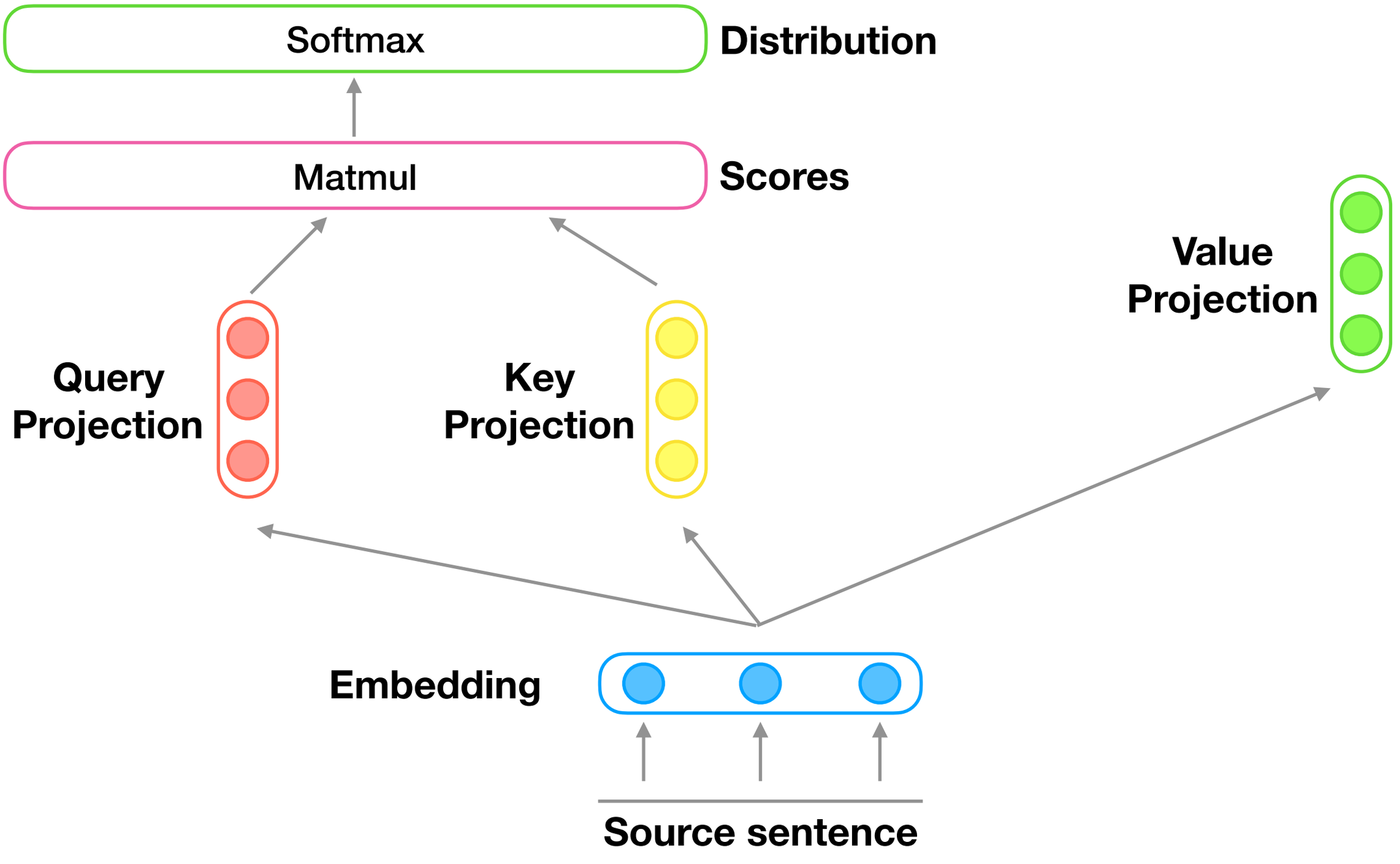
Scaled Dot-Product Attention



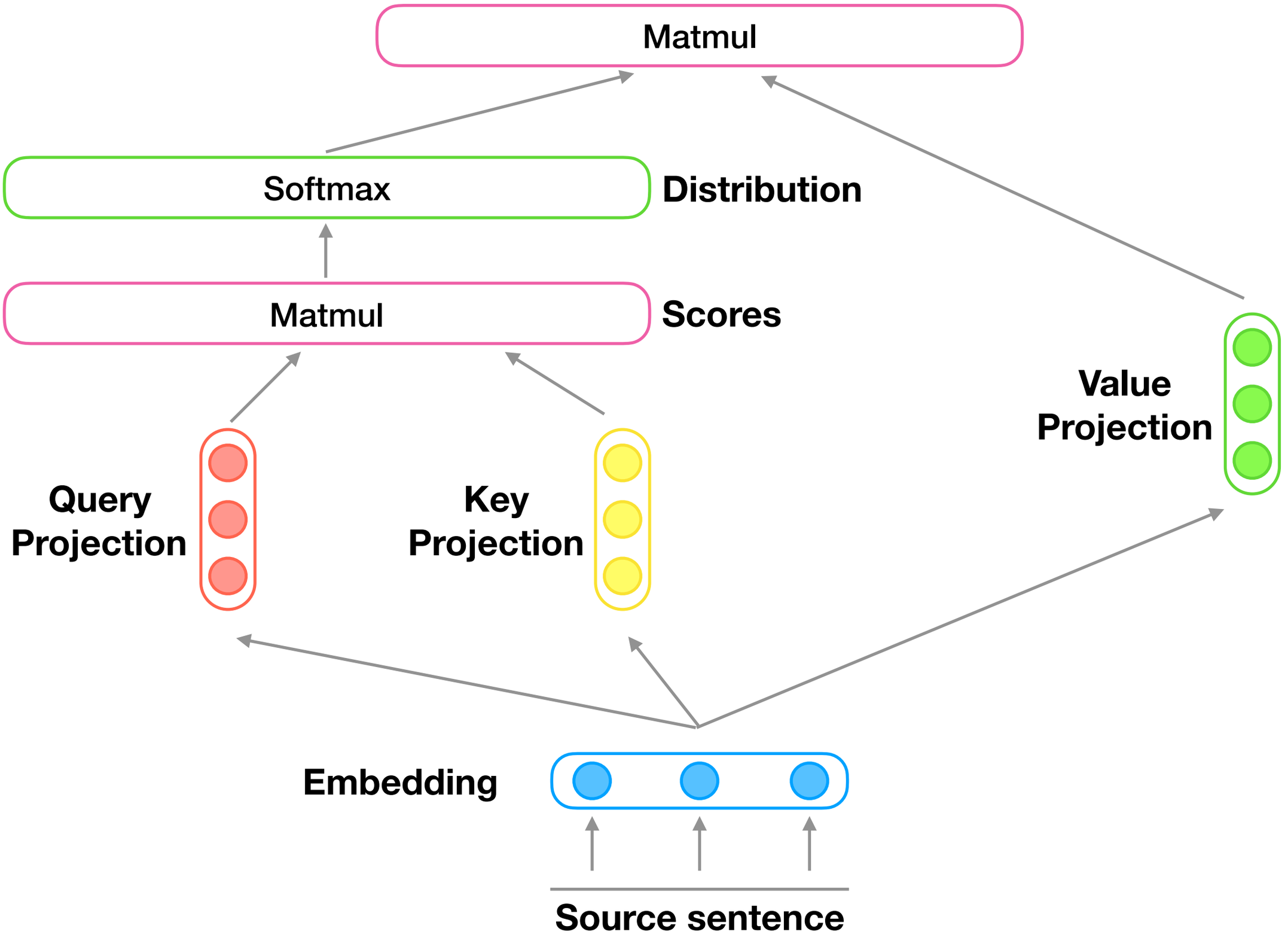
Scaled Dot-Product Attention



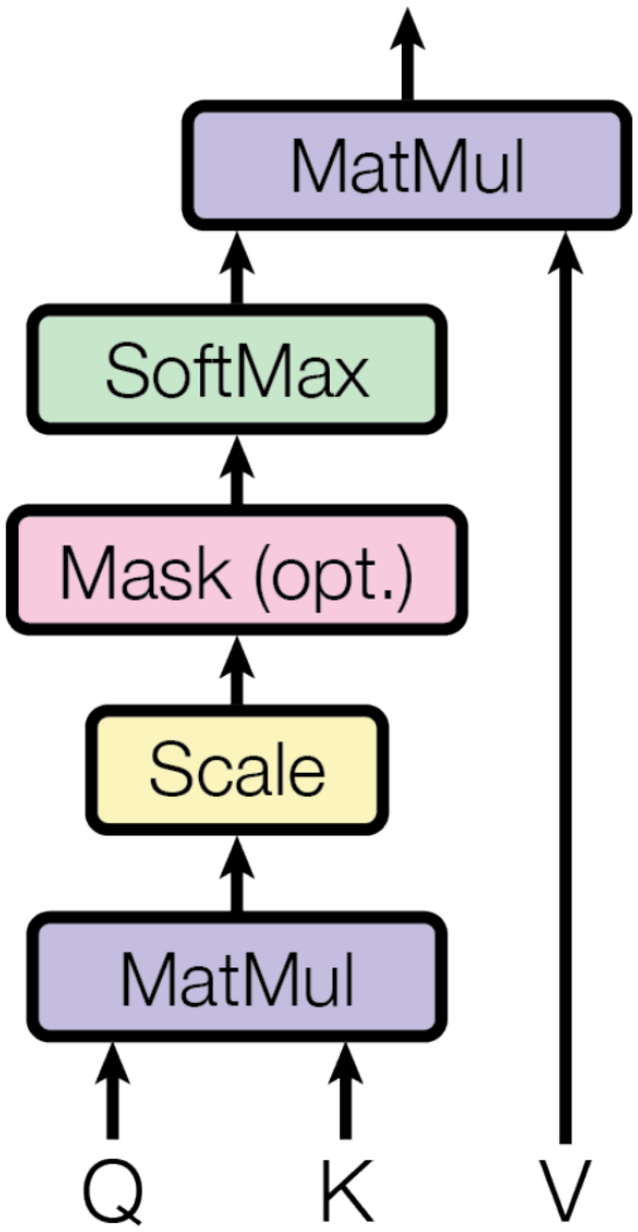
Scaled Dot-Product Attention



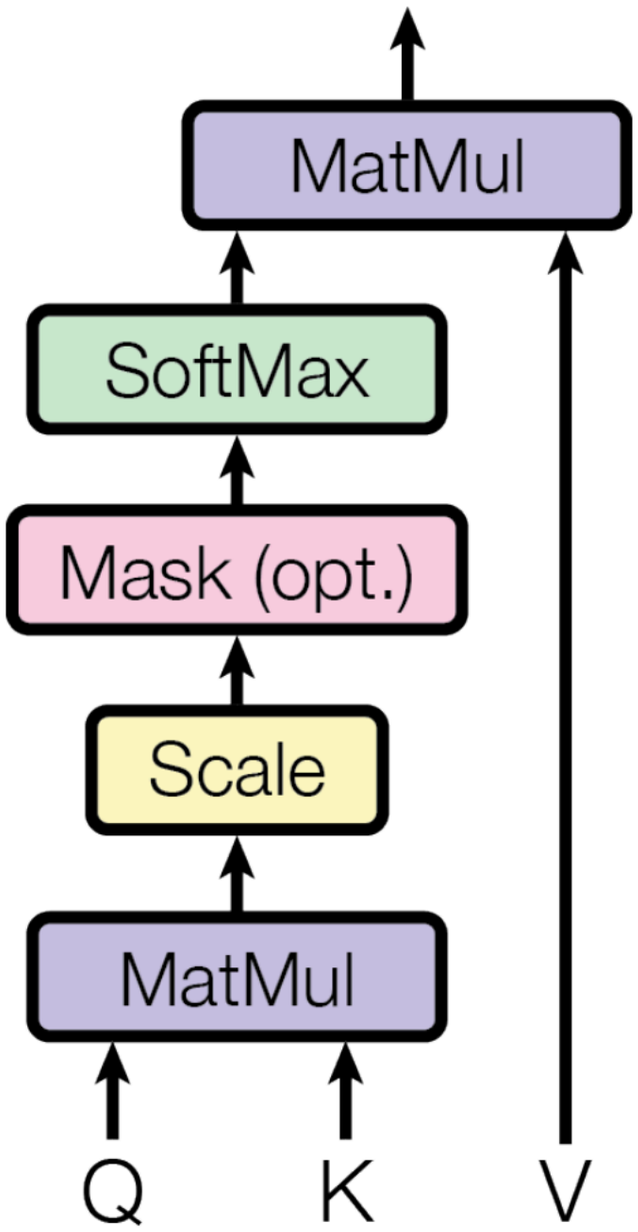
Scaled Dot-Product Attention



Scaled Dot-Product Attention

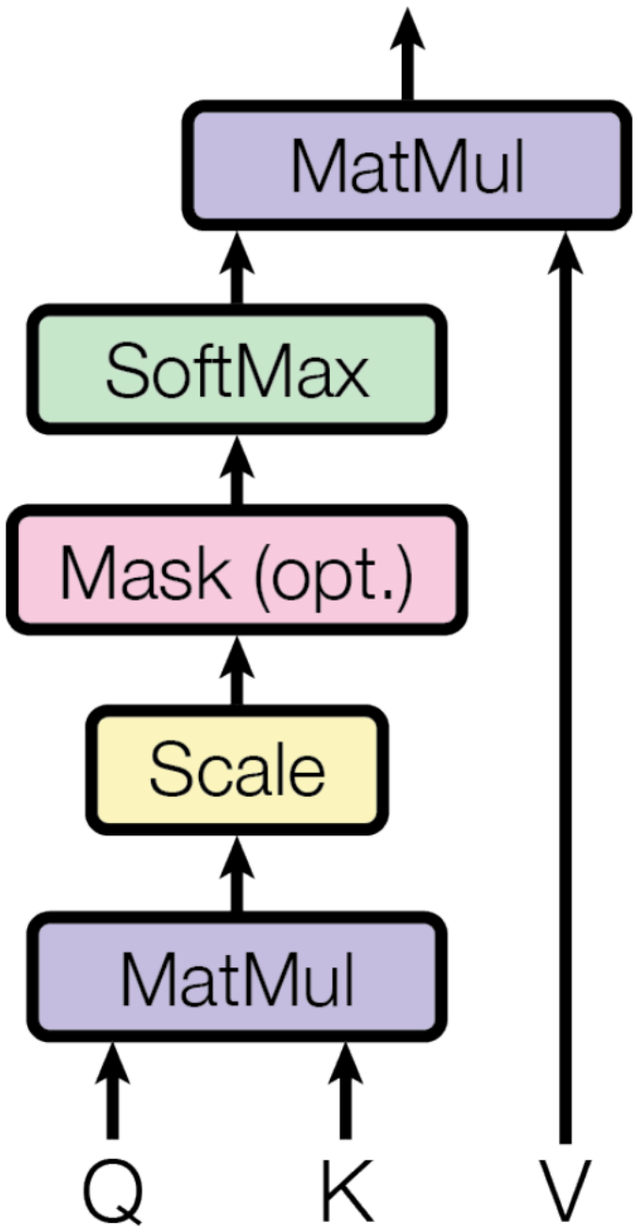


Scaled Dot-Product Attention



- Q — query
- K — key
- V — value

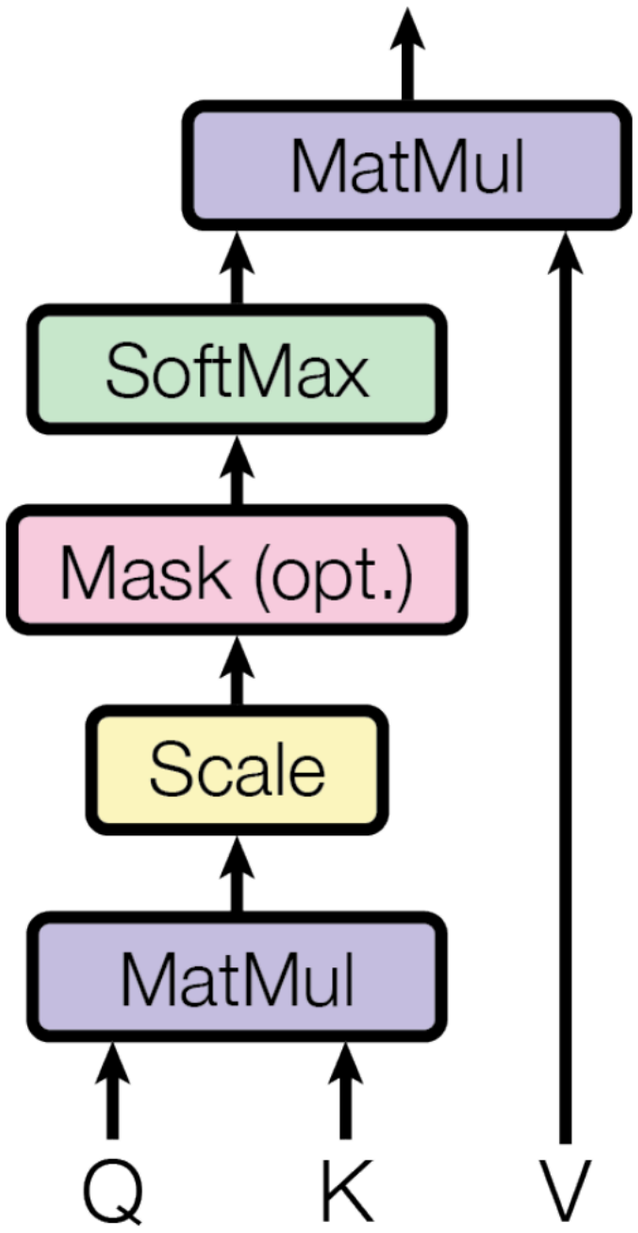
Scaled Dot-Product Attention



- Q — query
- K — key
- V — value

- Scaling by square root of key dimension for faster convergence, make smoother

Scaled Dot-Product Attention

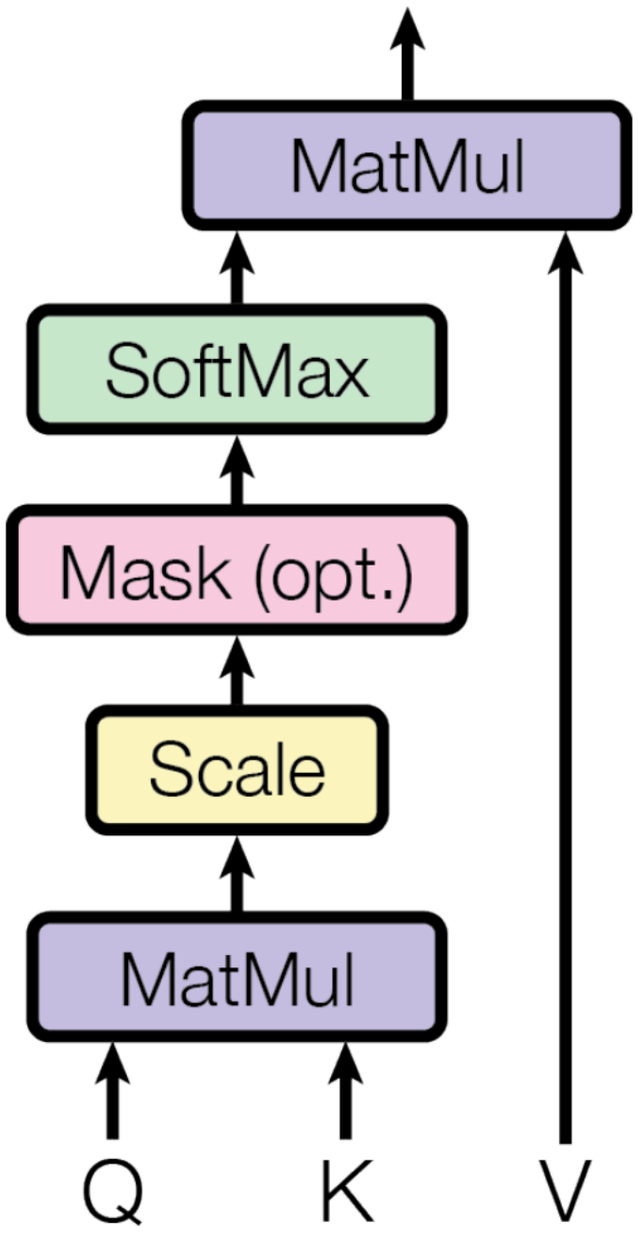


- Q — query
- K — key
- V — value

- Scaling by square root of key dimension for faster convergence, make smoother

- Masking for future steps (for seq2seq learning), «opt.» means optional

Scaled Dot-Product Attention



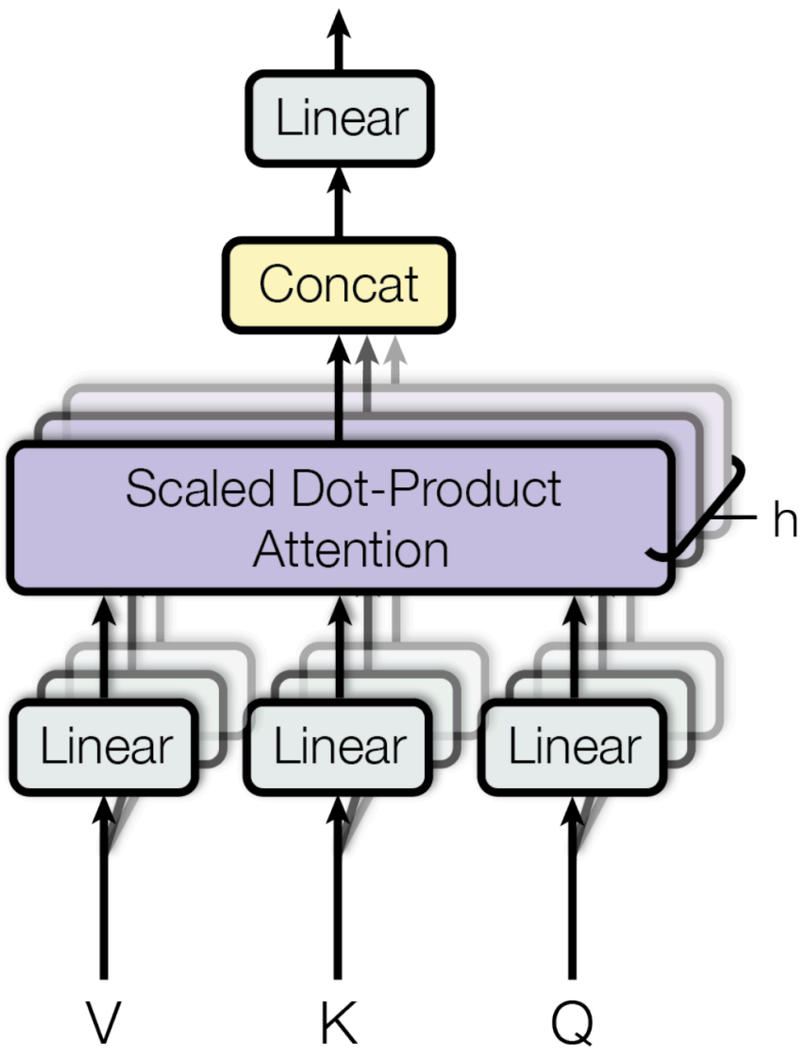
- Q — query
- K — key
- V — value

- Scaling by square root of key dimension for faster convergence, make smoother

- Masking for future steps (for seq2seq learning), «opt.» means optional

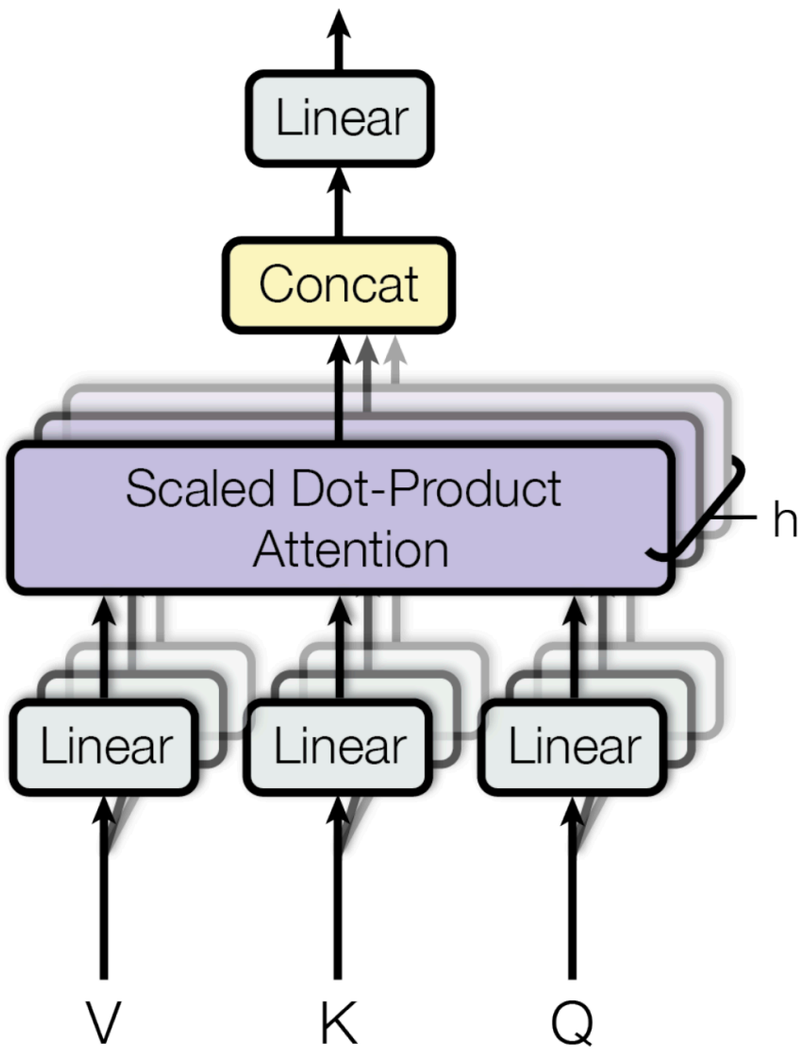
- Self-attention give for every embedded token information about whole sentence

Multi-Head Attention

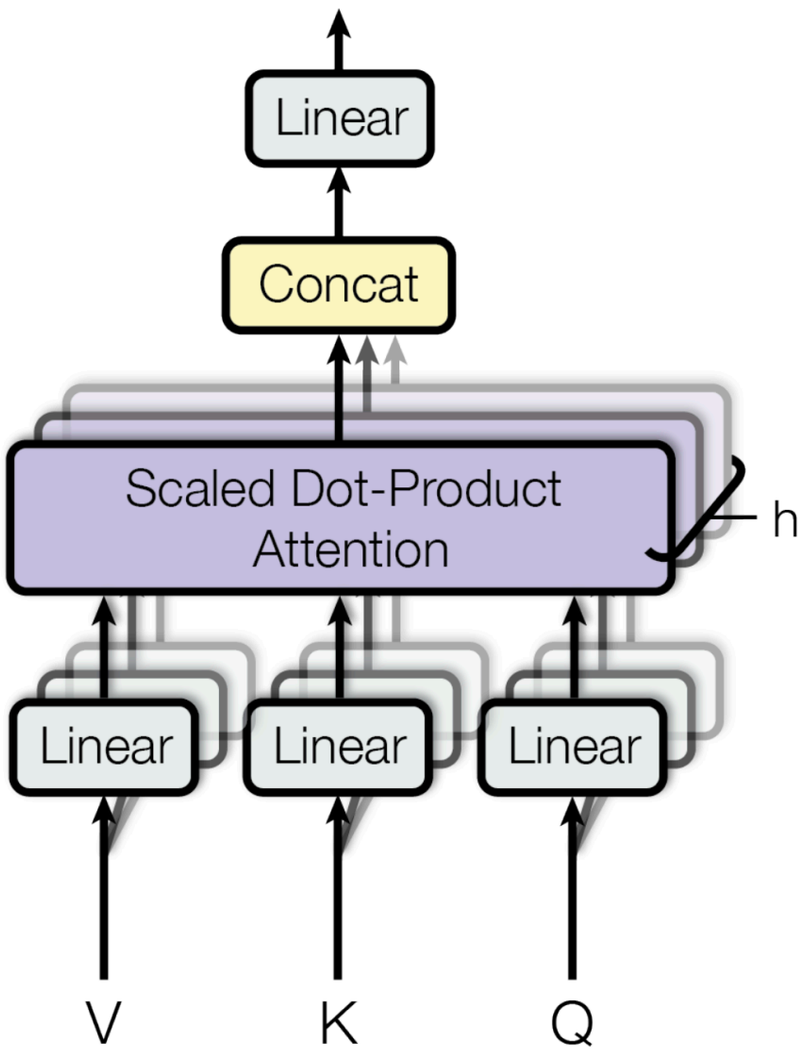


Multi-Head Attention

- Same things look different and we learning to look differently

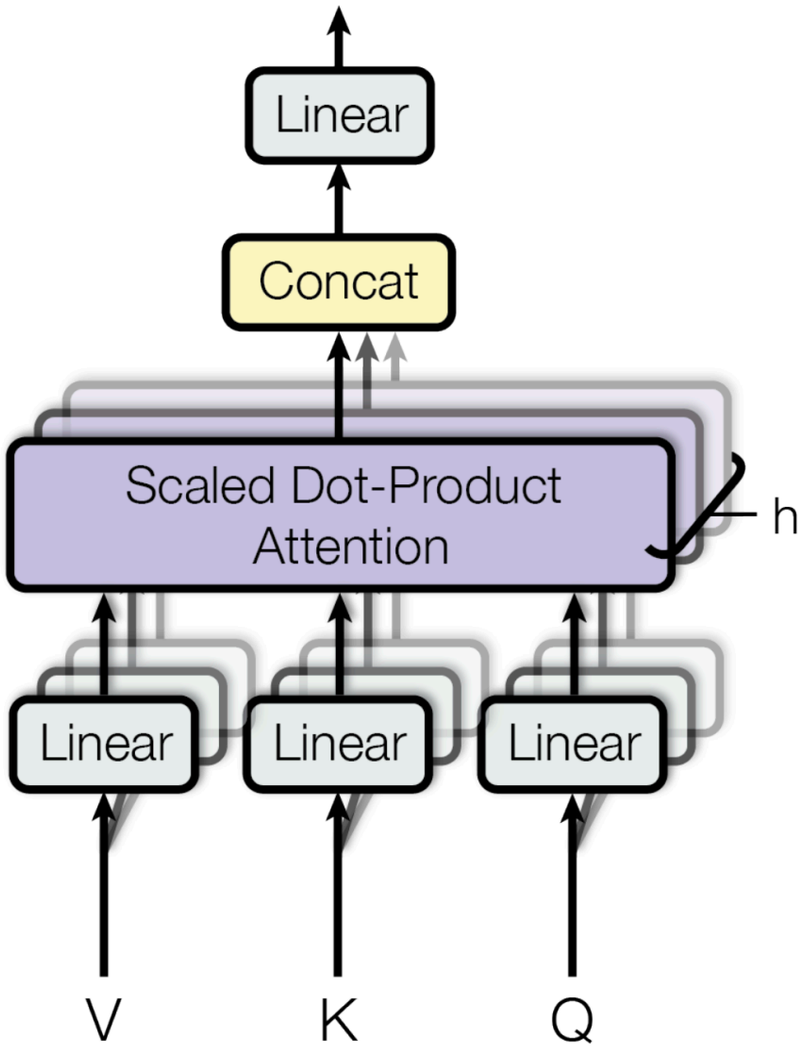


Multi-Head Attention



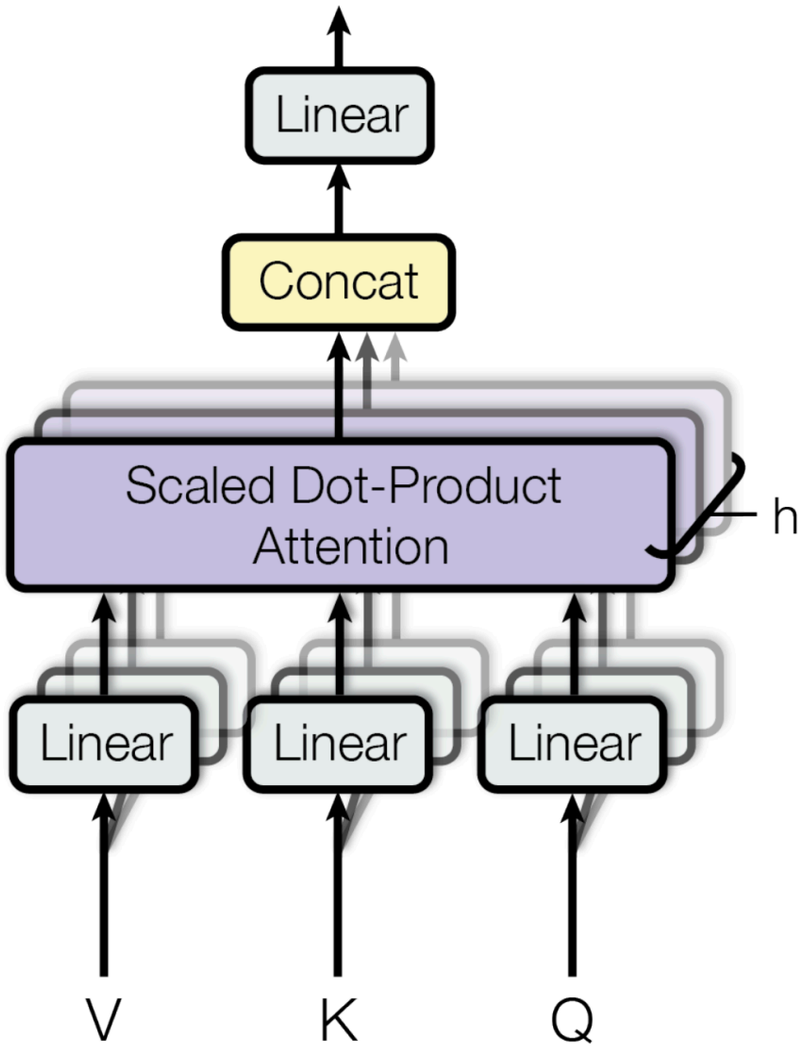
- Same things look different and we learning to look differently
- It’s like convolution filters

Multi-Head Attention



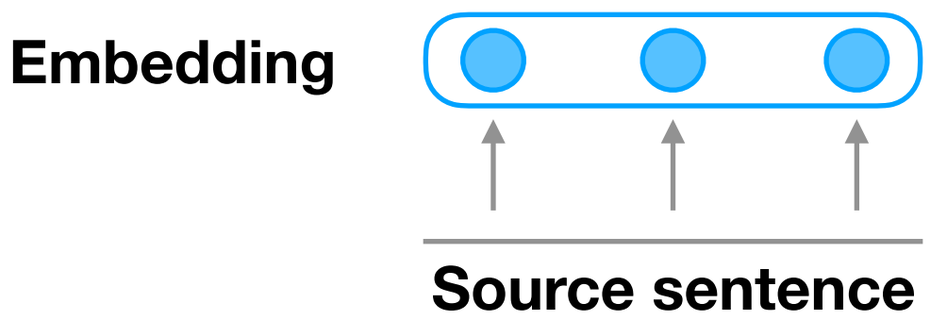
- Same things look different and we learning to look differently
- It’s like convolution filters
- After split to different heads and learning something we need concatenate it

Multi-Head Attention

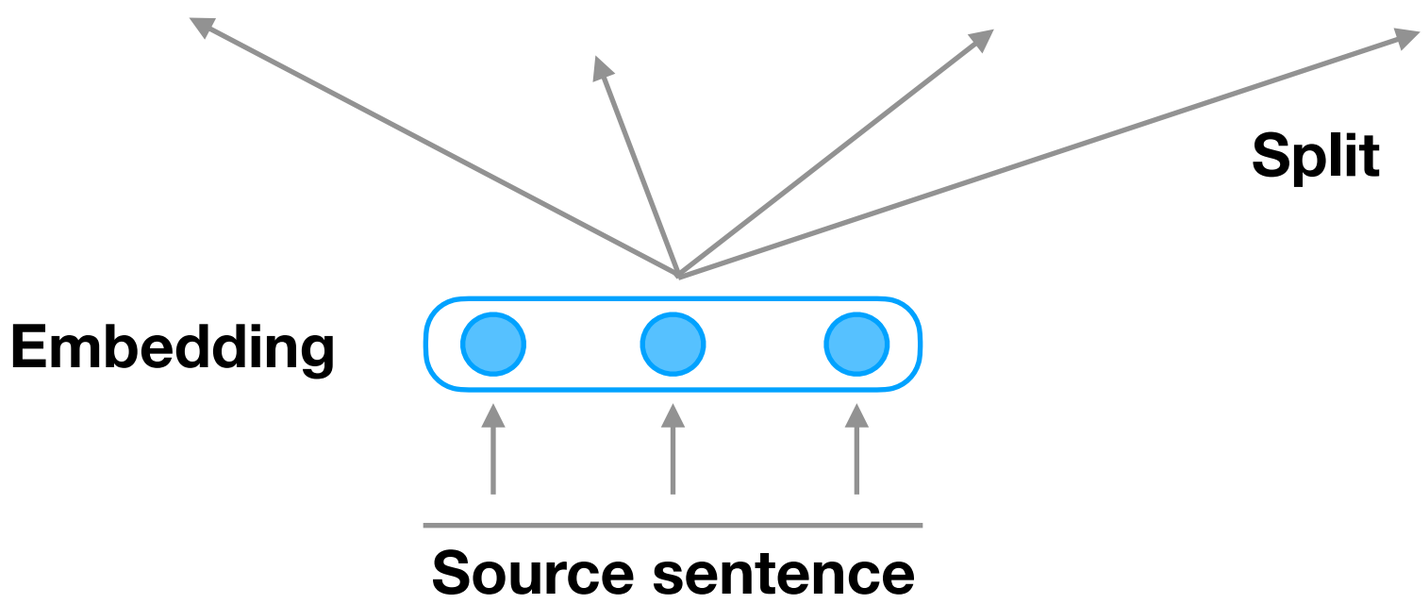


- Same things look different and we learning to look differently
- It’s like convolution filters
- After split to different heads and learning something we need concatenate it
- And mixed up heads by another linear projection

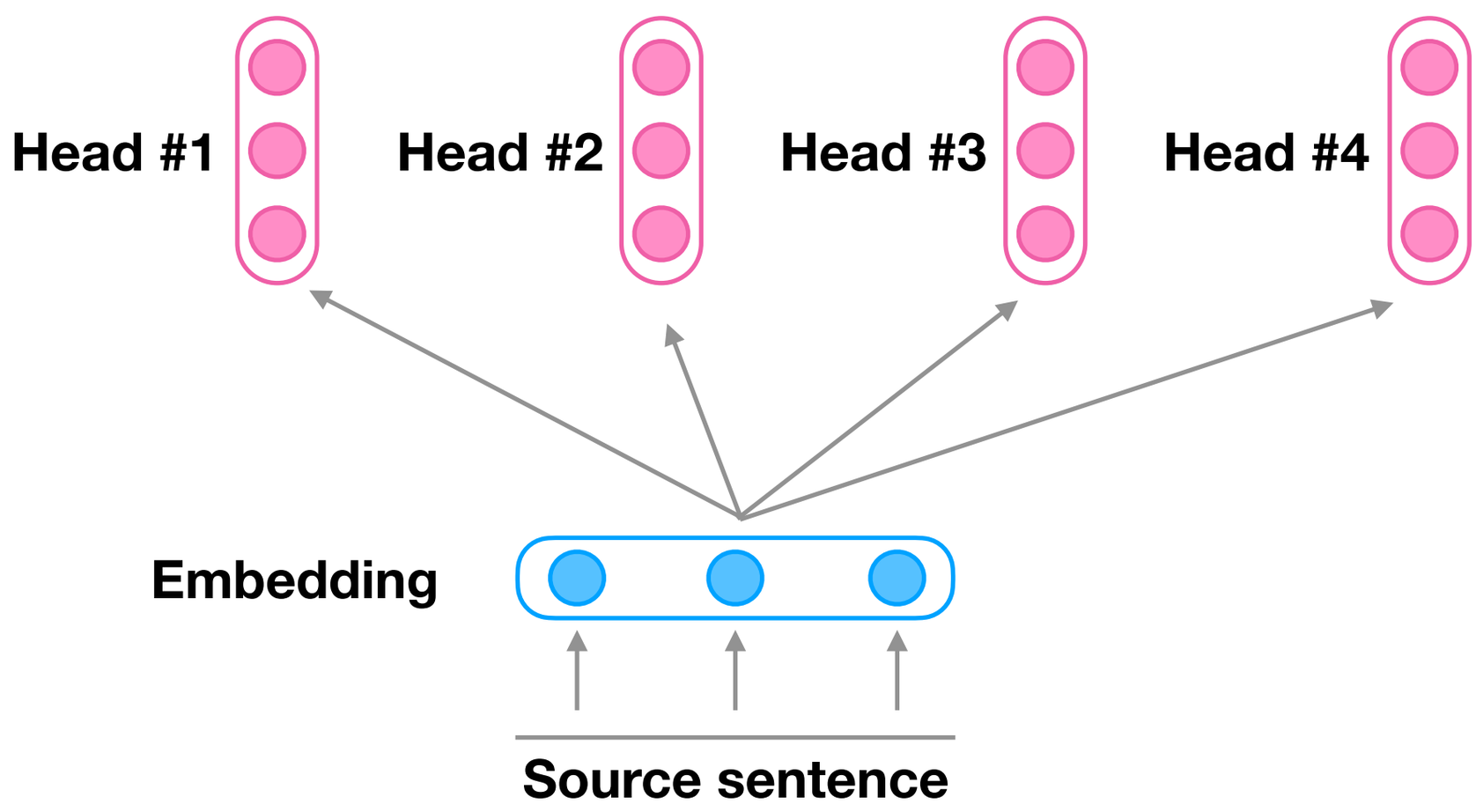
Multi-Head Attention



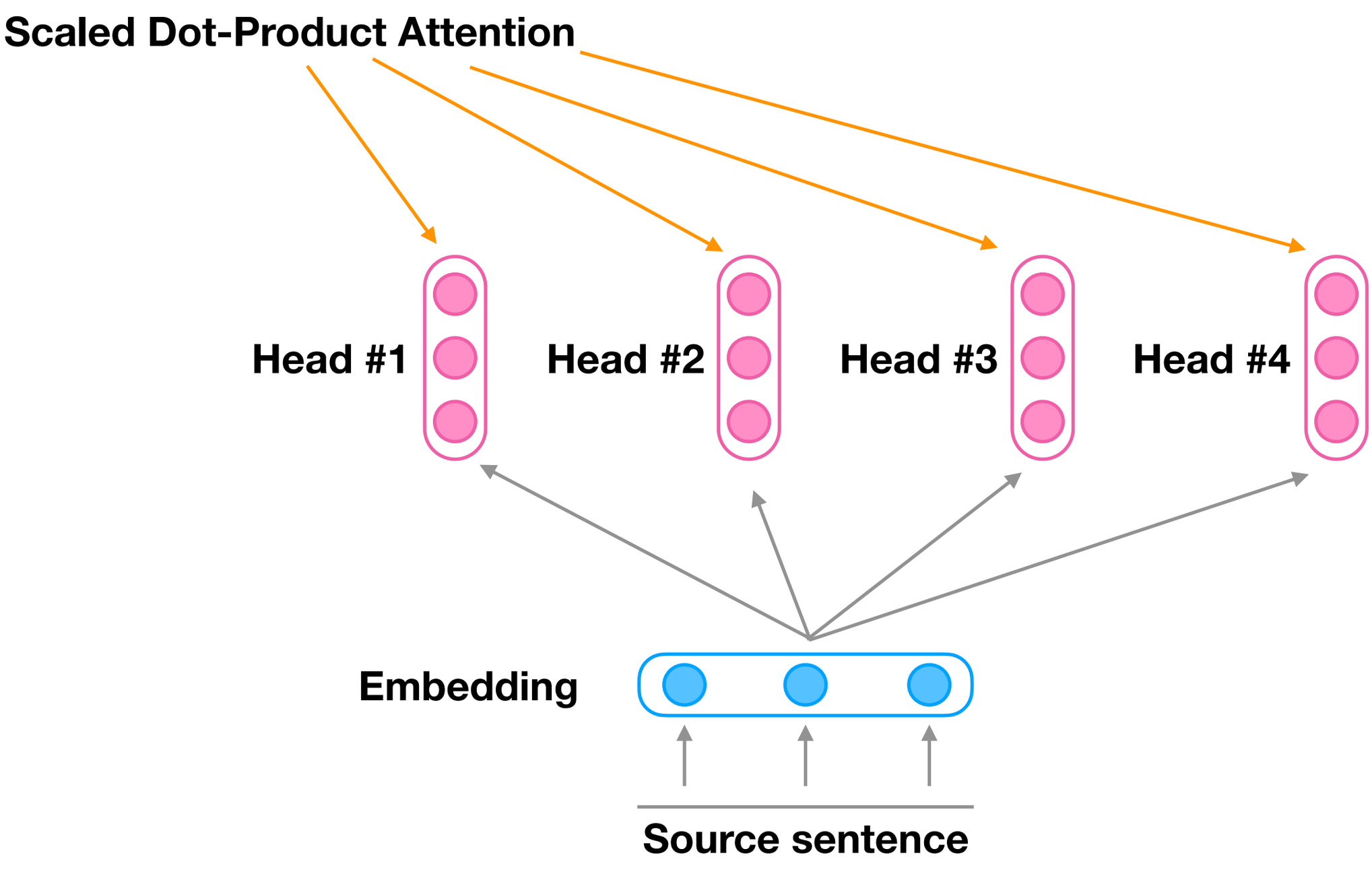
Multi-Head Attention



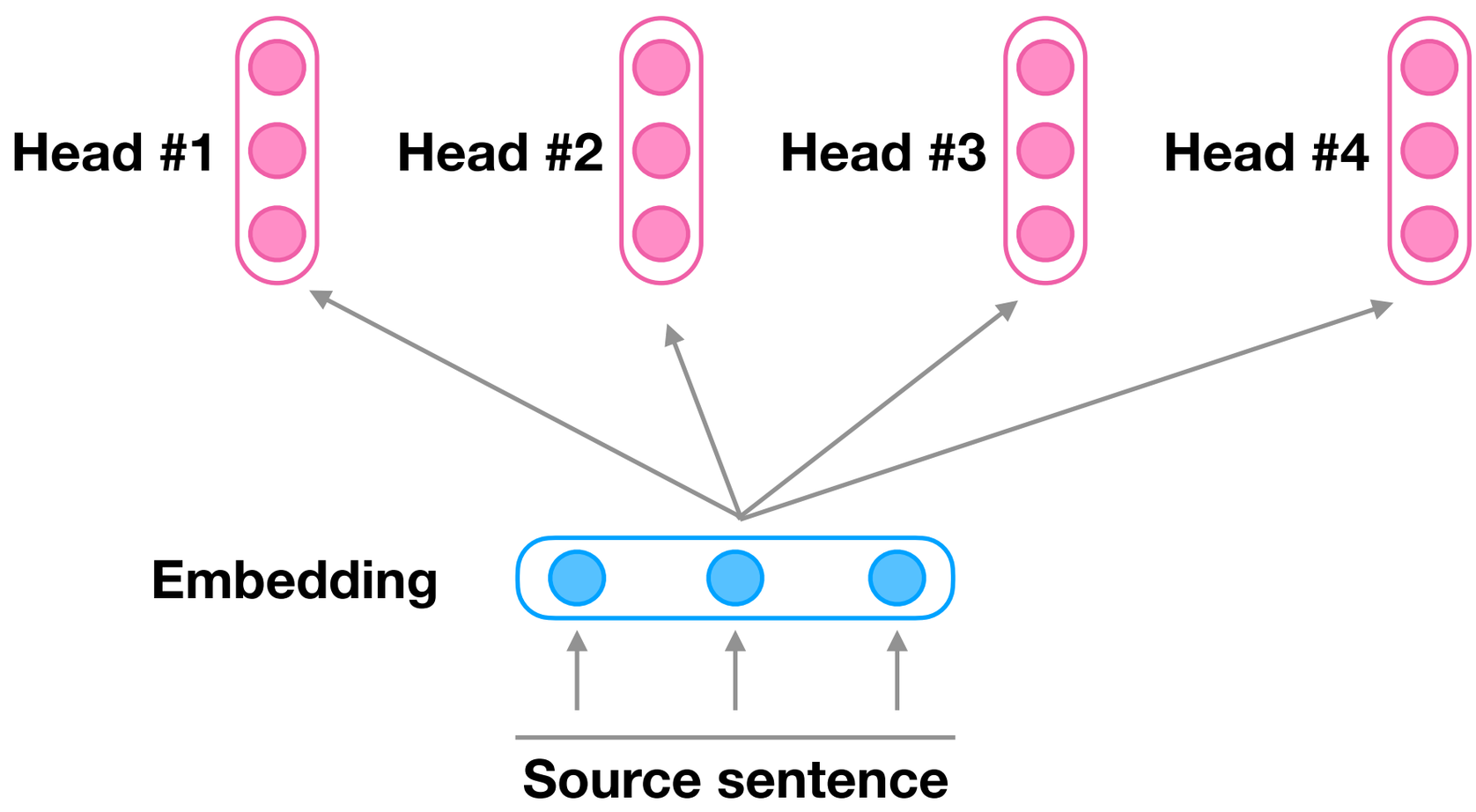
Multi-Head Attention



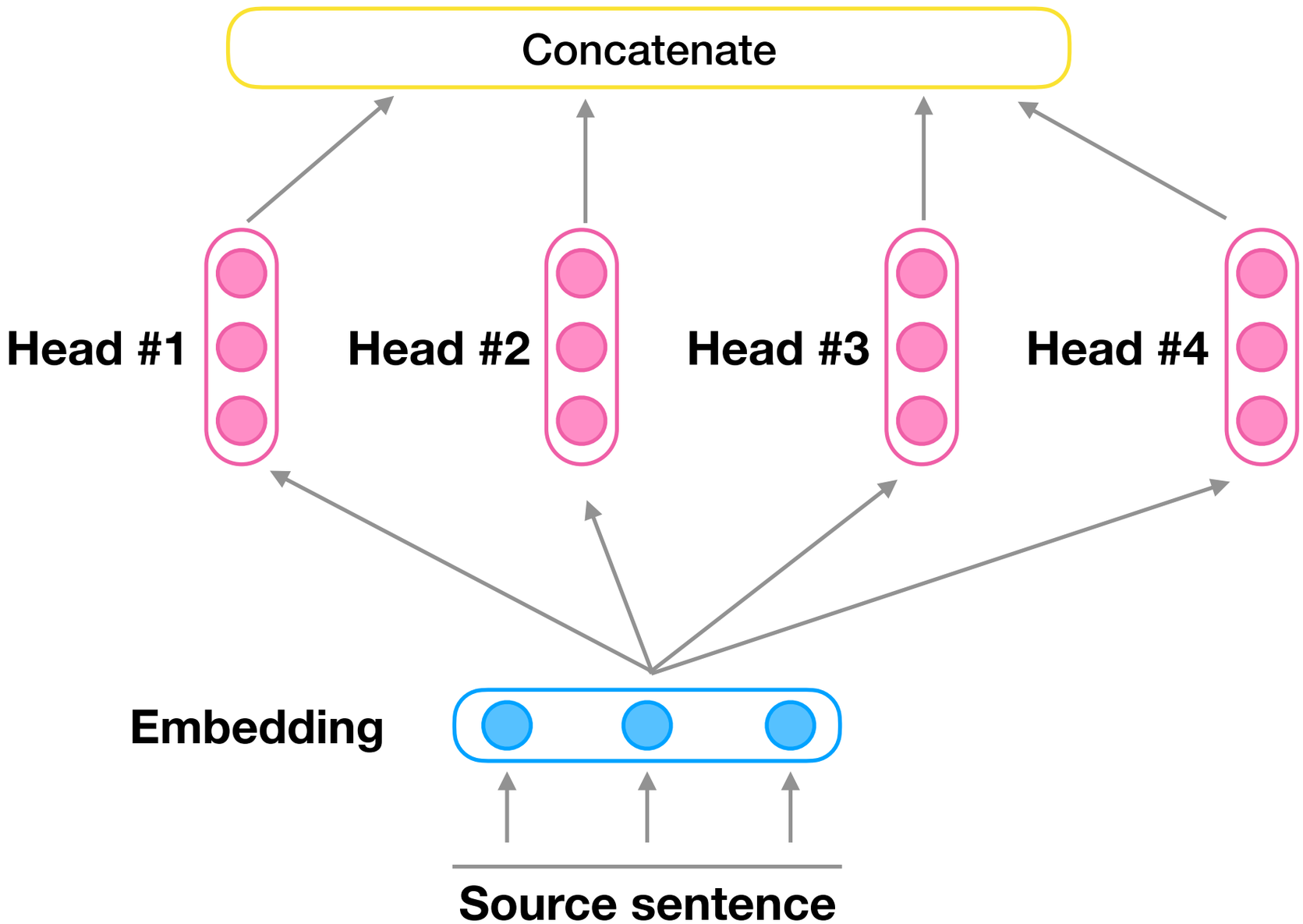
Multi-Head Attention



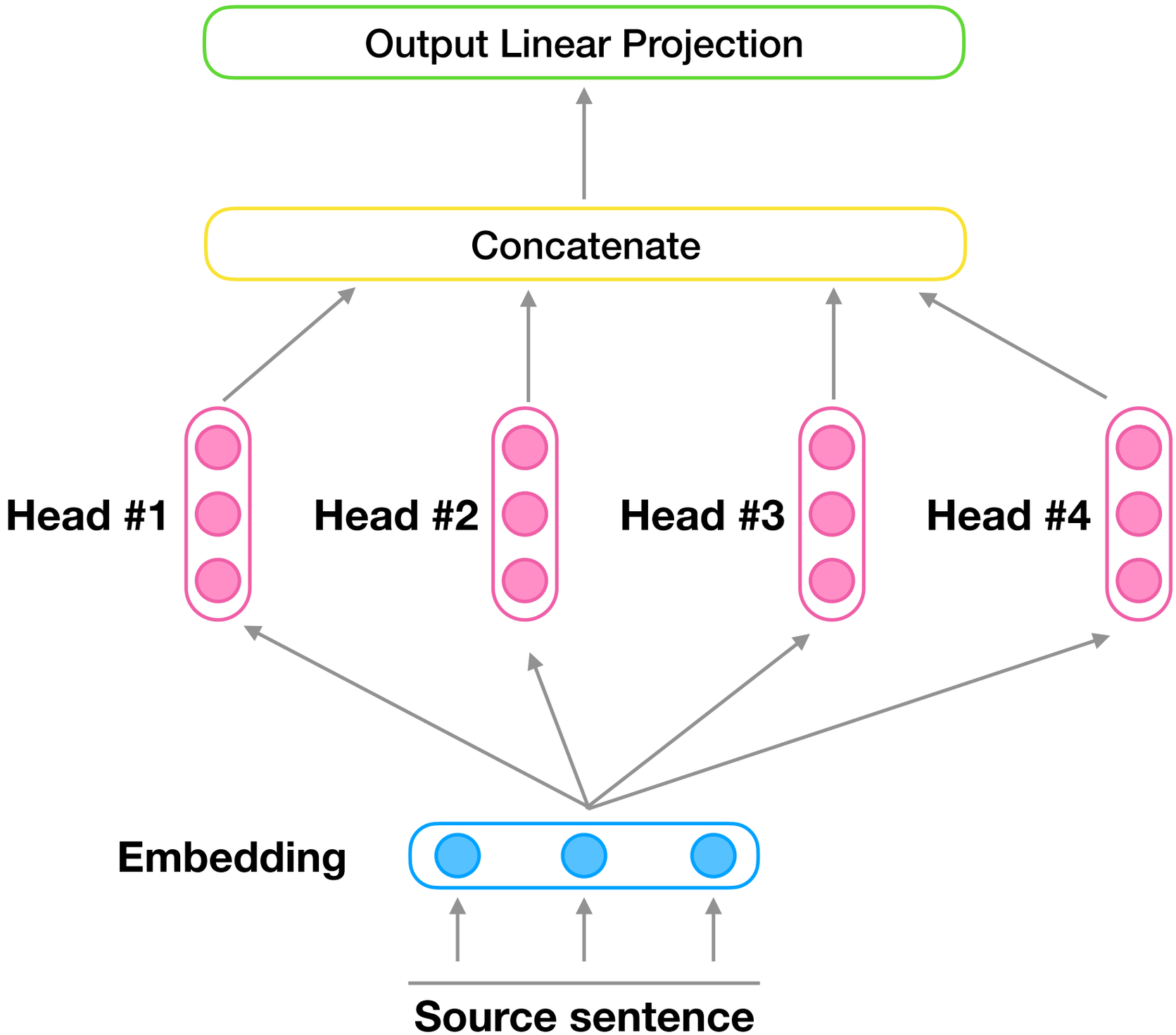
Multi-Head Attention



Multi-Head Attention

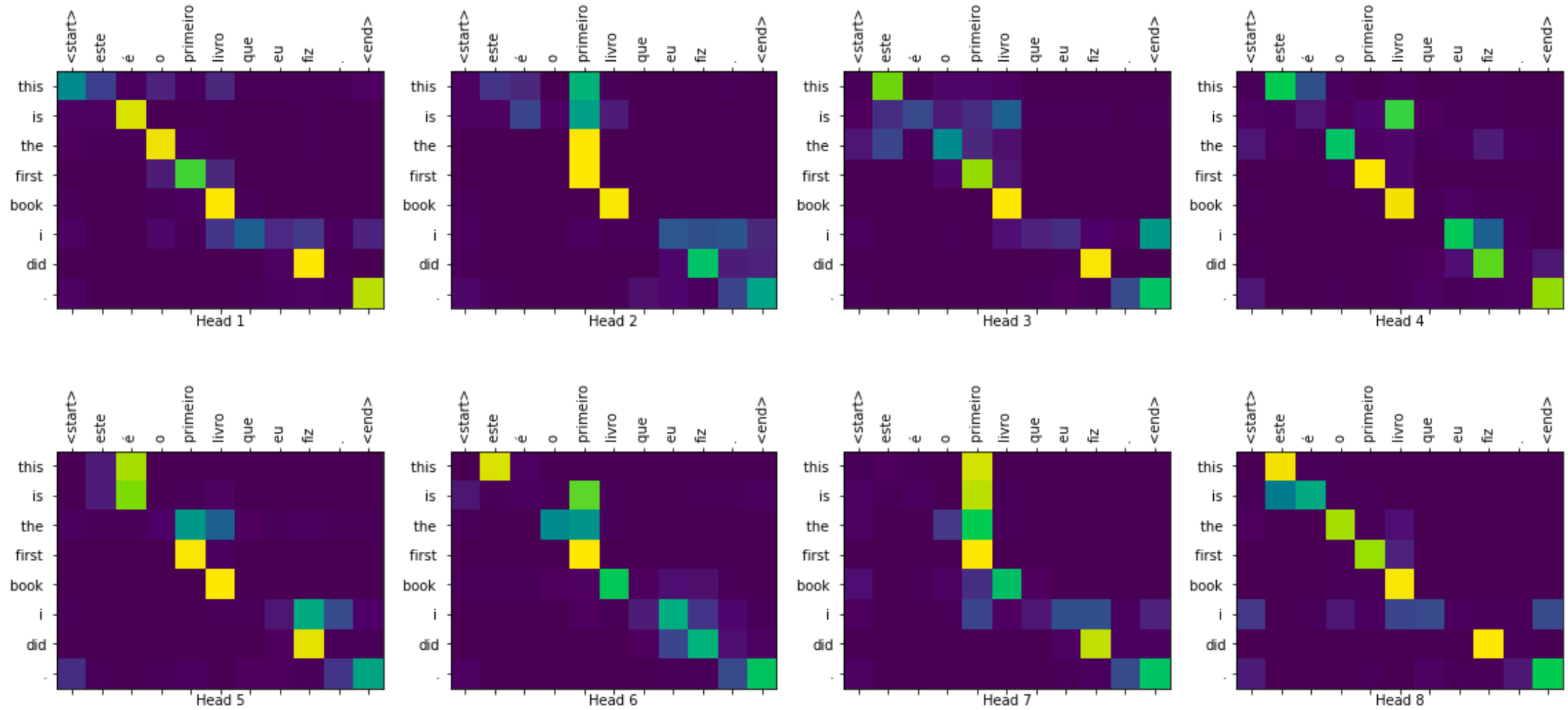


Multi-Head Attention



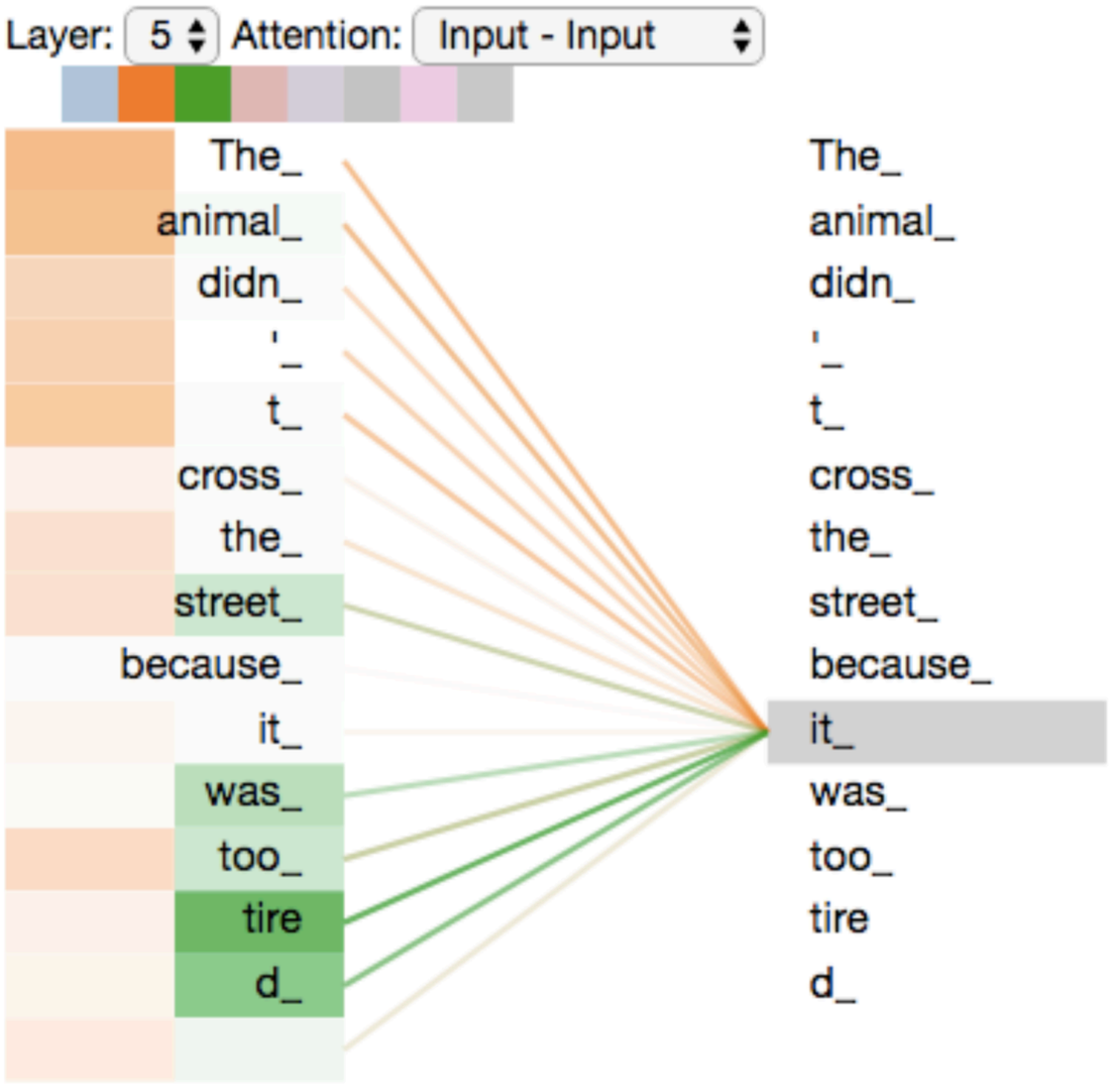
Multi-Head Attention

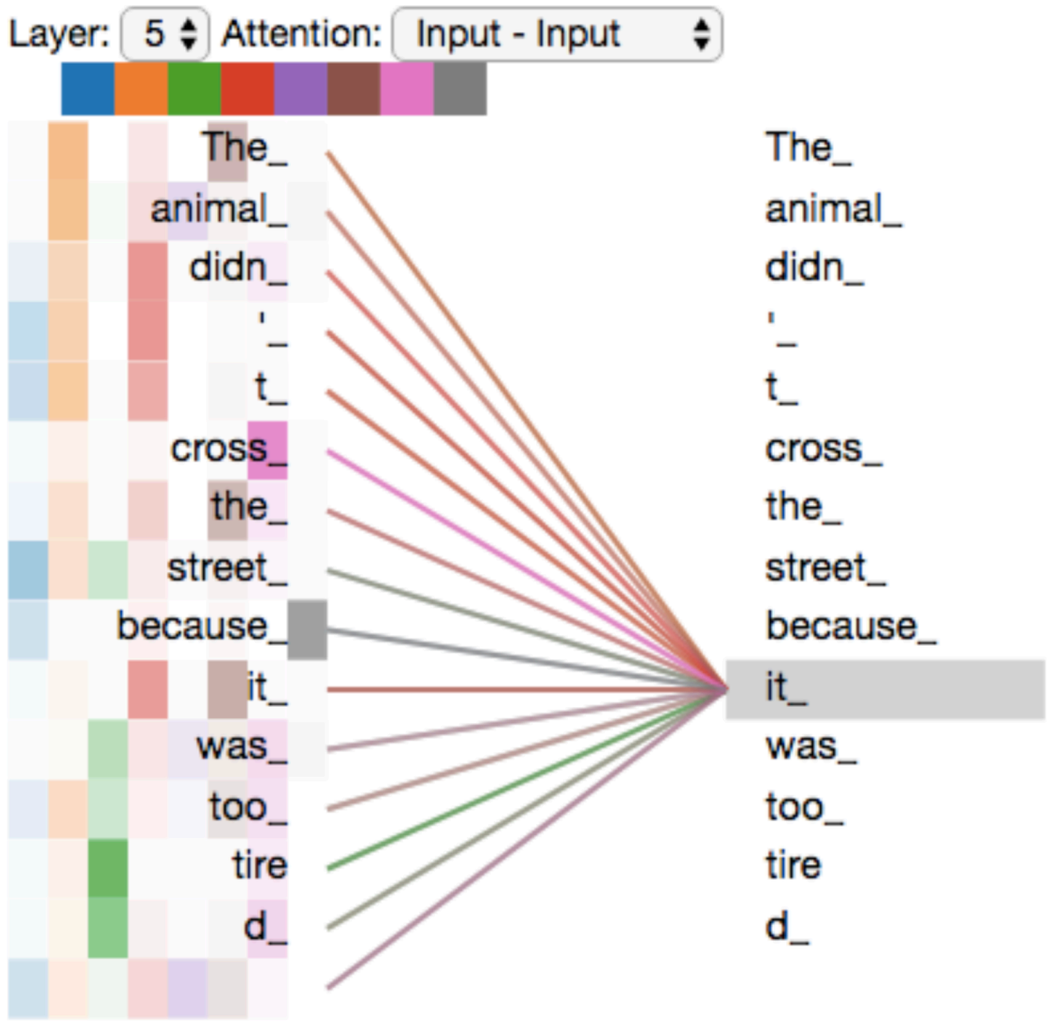
Attention Heads Visualization



Multi-Head Attention

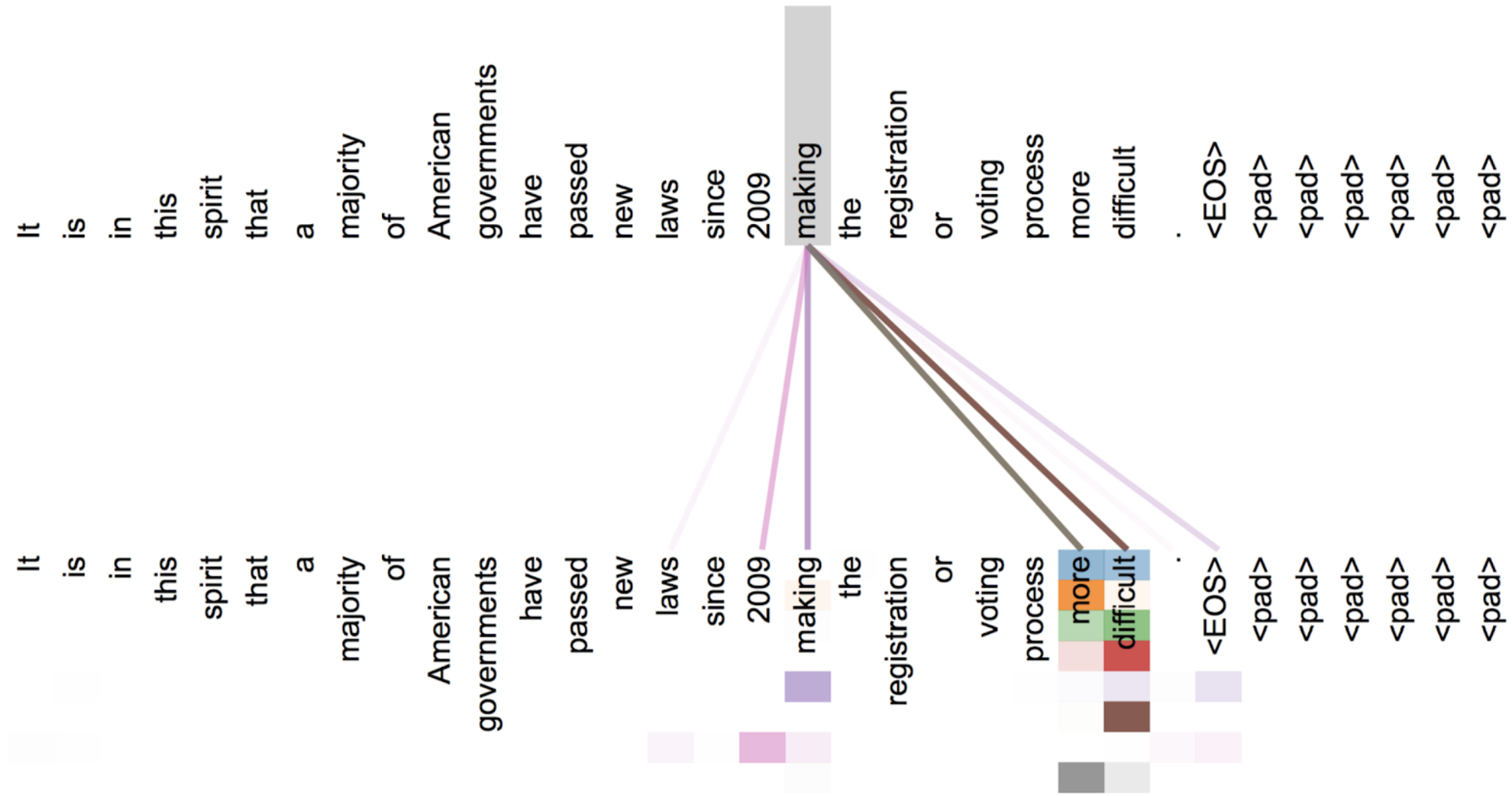
Attention Heads Visualization





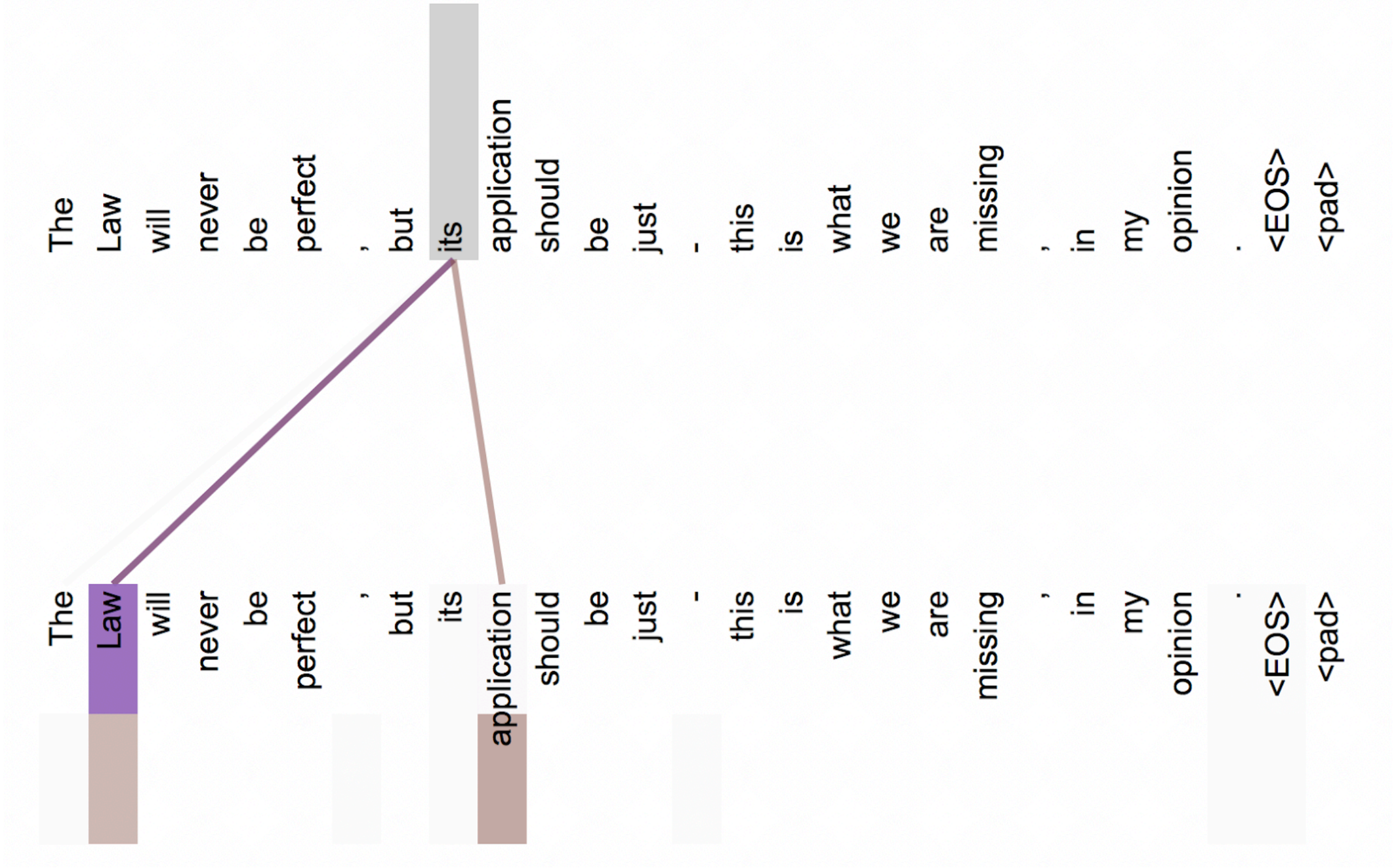
Multi-Head Attention

Attention Heads Visualization

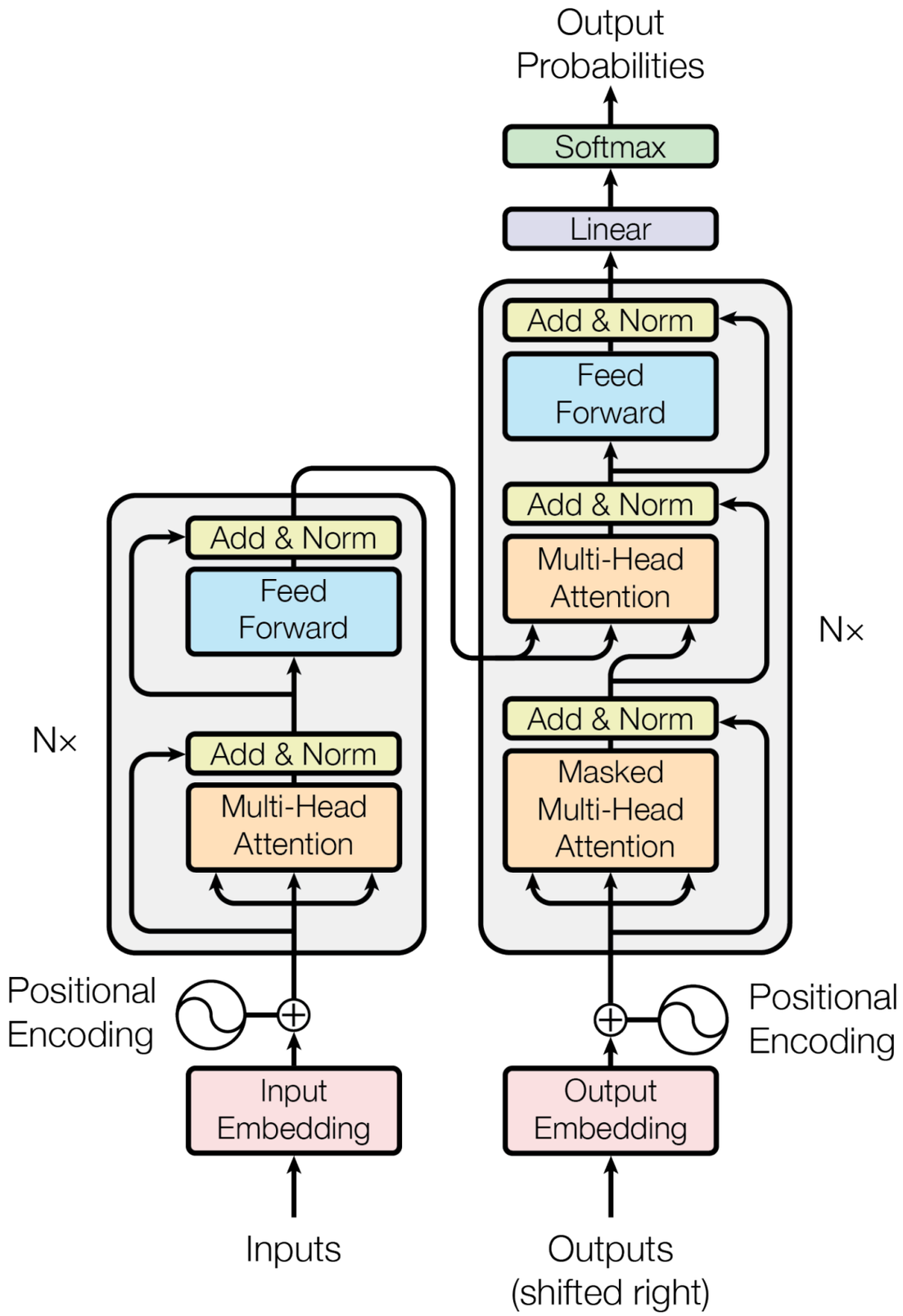


Multi-Head Attention

Attention Heads Visualization

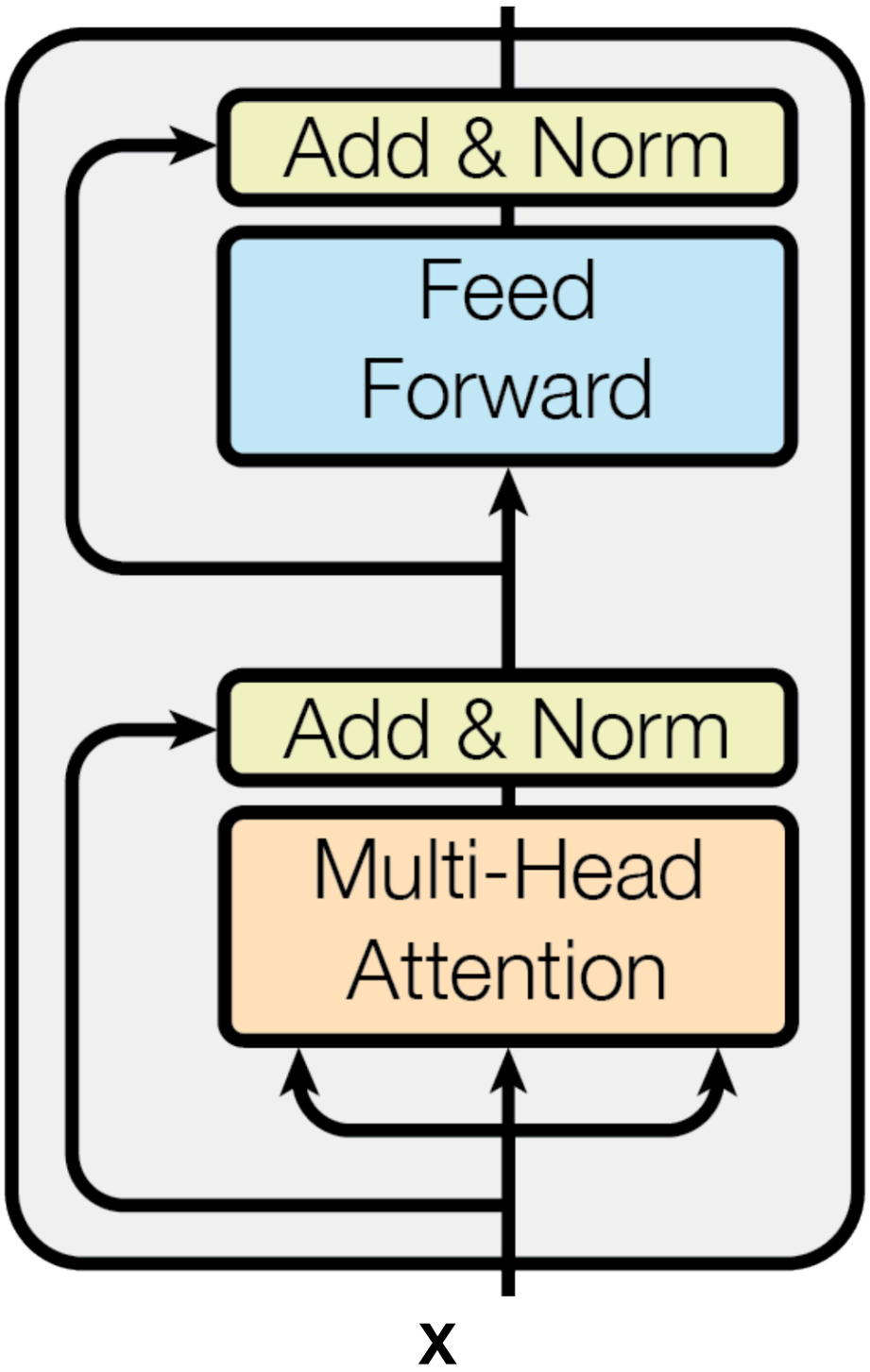


Transformer



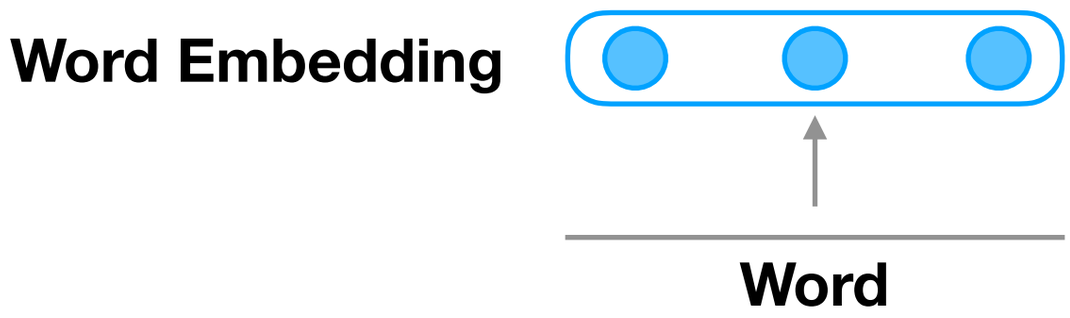
- Masking to ensure that we do not look into the future
- Masks setting to ***-inf*** then we apply softmax and this word becomes zero
- Linear and softmax is just word prediction

Transformer Encoder

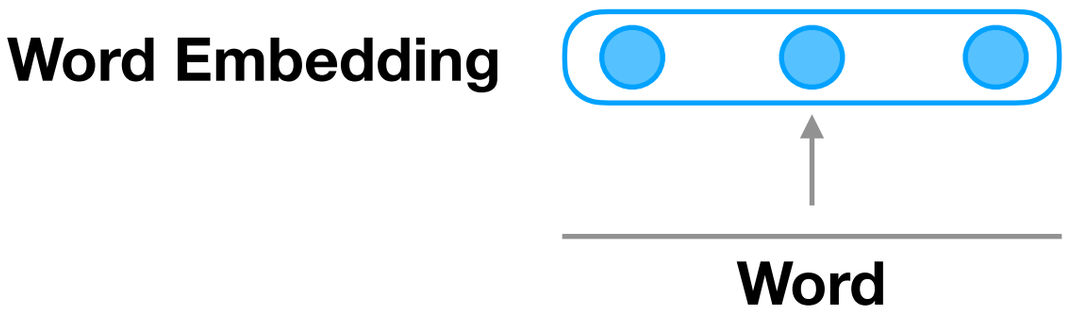


- Add & Norm = LayerNorm(x + sublayer(x))
- LayerNorm changes input to have mean 0 and variance 1 for faster convergence
- Residual connections
- Dropout with small probability (0.1)

Feed Forward



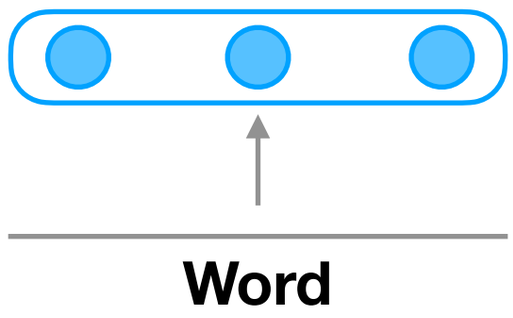
Feed Forward



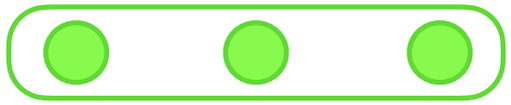
Feed Forward

Usually 4X of previous dimension

Word Embedding

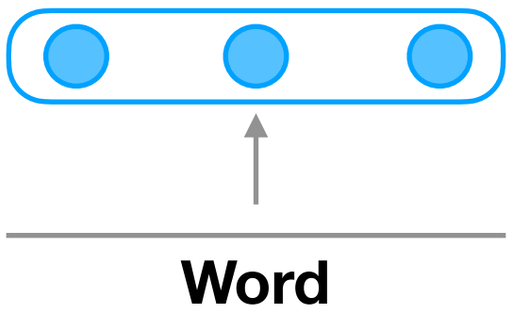


Feed Forward



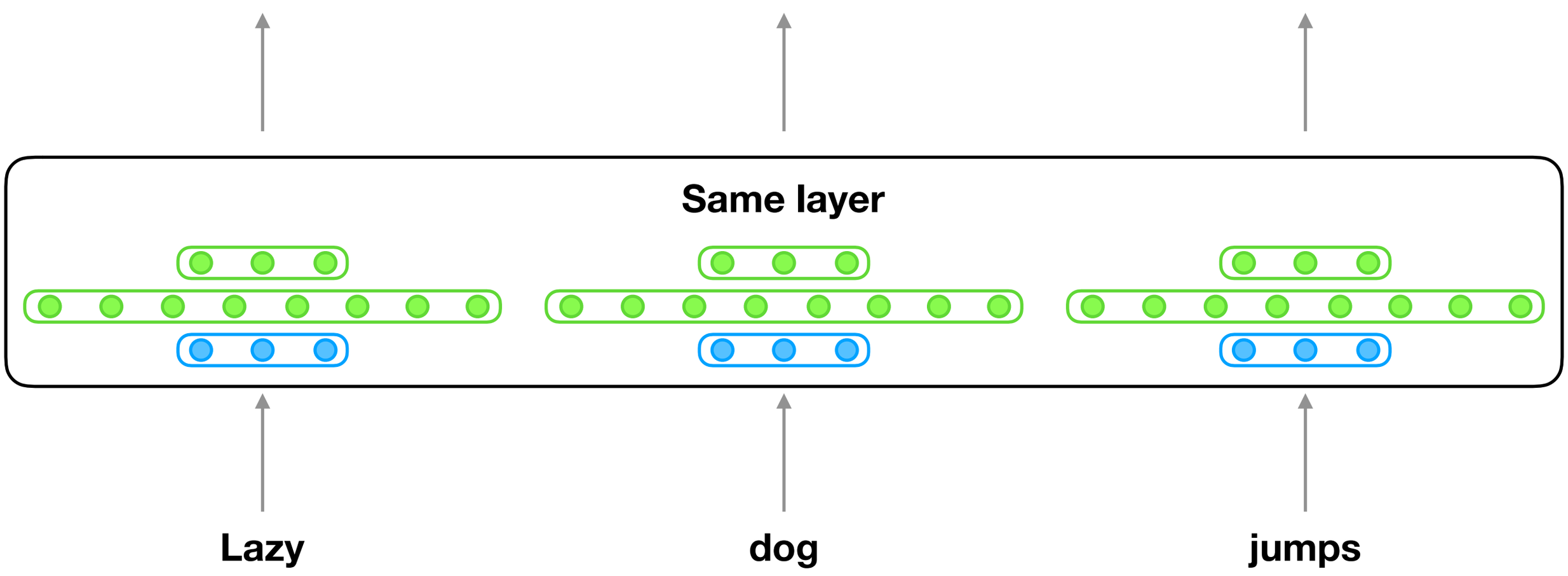
Usually 4X of previous dimension

Word Embedding

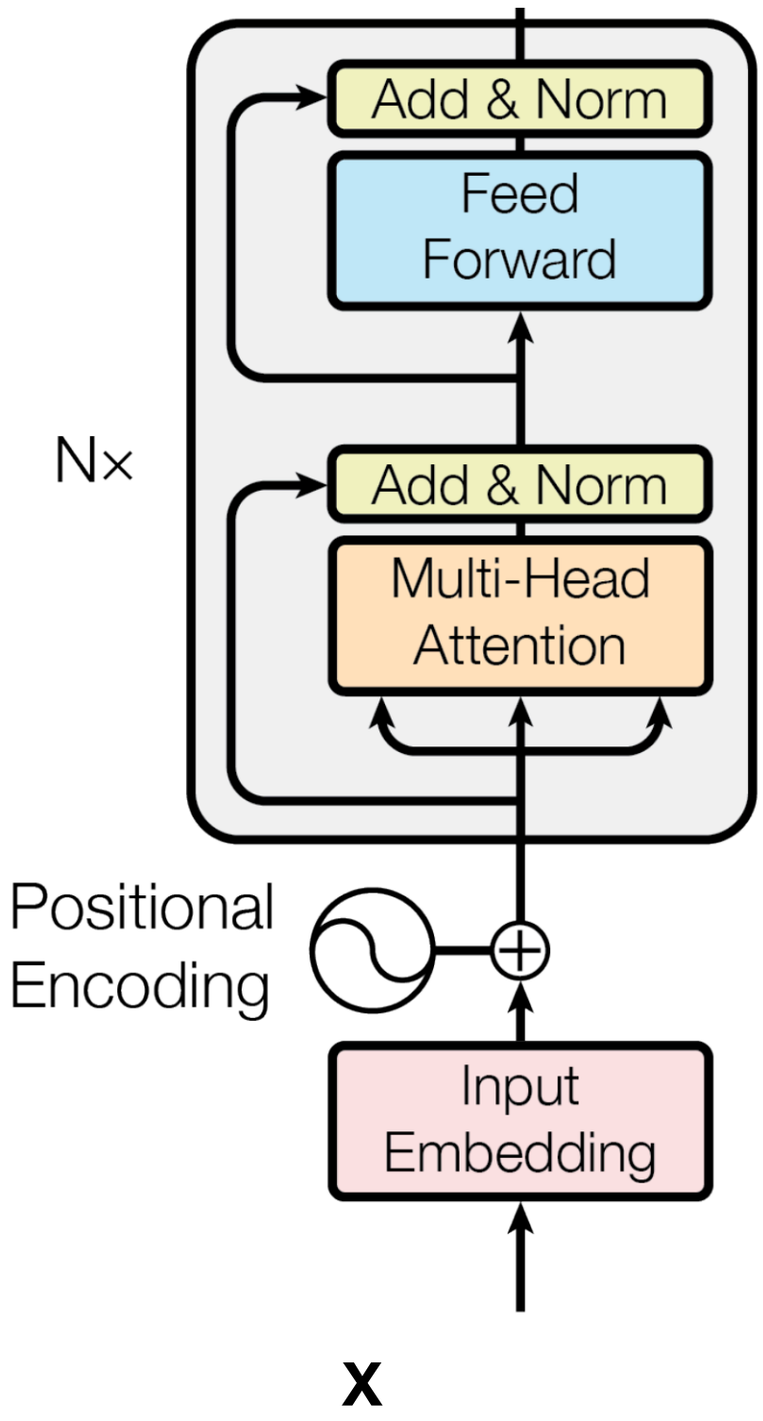


Word

Position-Wise Feed Forward

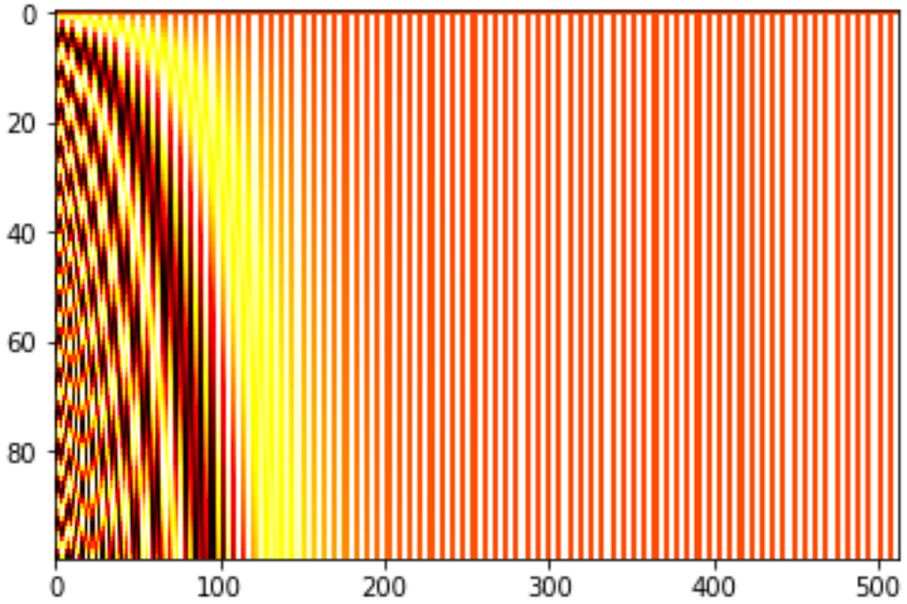


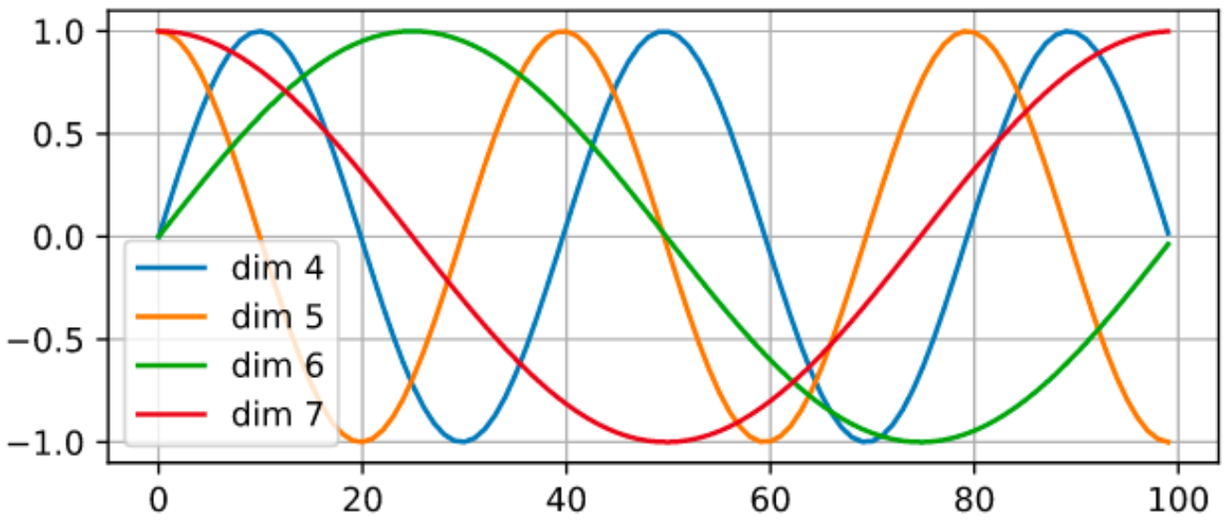
Transformer Encoder



- Nx — just stacked layers for more capacity
- Positional Encoding give information about position of word in text (can be heuristic or learnable)
- Input Embedding just common lookup table

Positional Encoding



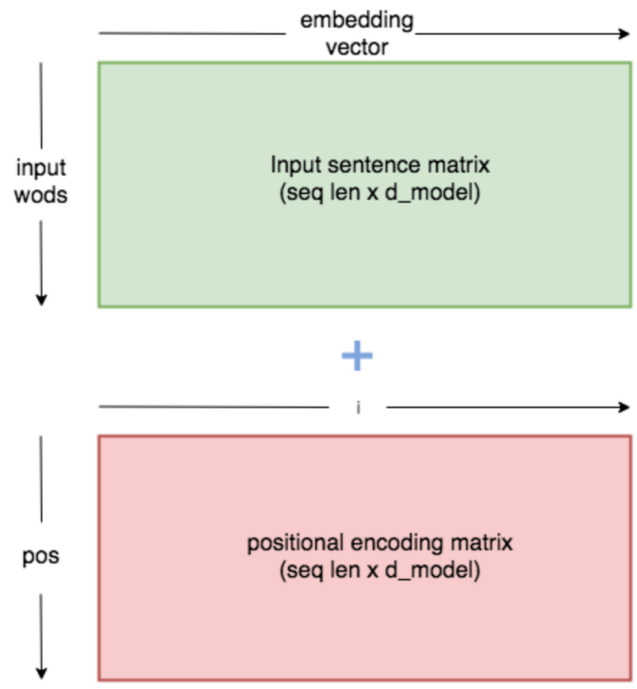


$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

i — index in embedding

- We can calculate positional encoding for every length
- It’s hypothesis and heuristic
- Same results compare to learnable positional embeddings



Positional Embedding

```
positional_embedding = torch.nn.Embedding(num_embeddings=x.shape[0], embedding_dim=x.shape[-1])
```

```
positional_embedding
```

```
Embedding(8, 64)
```

```
positions = torch.arange(start=0, end=x.shape[0])
positions
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7])
```

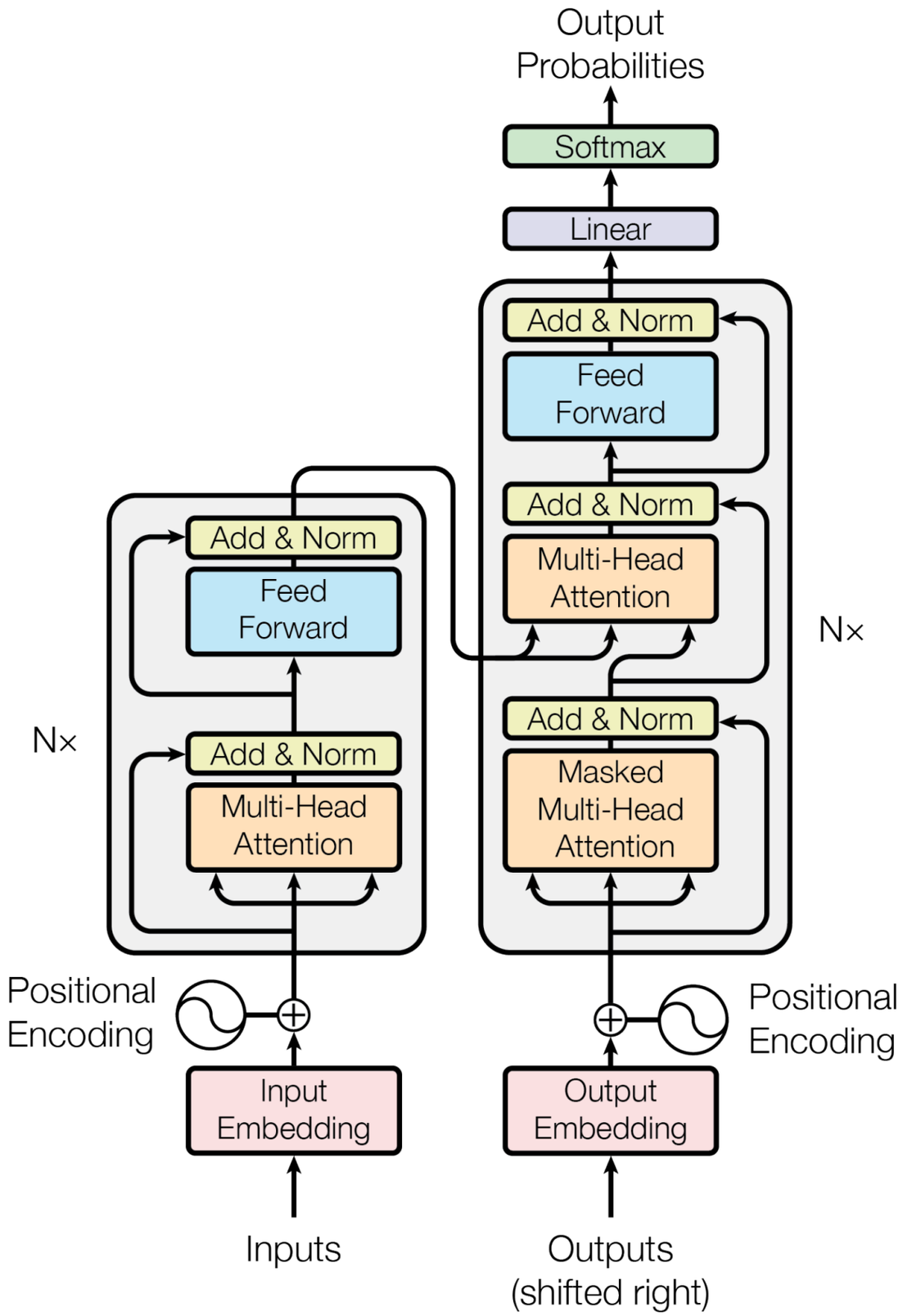
```
pos_emb = positional_embedding(positions)
```

```
x.shape == pos_emb.shape
```

```
True
```

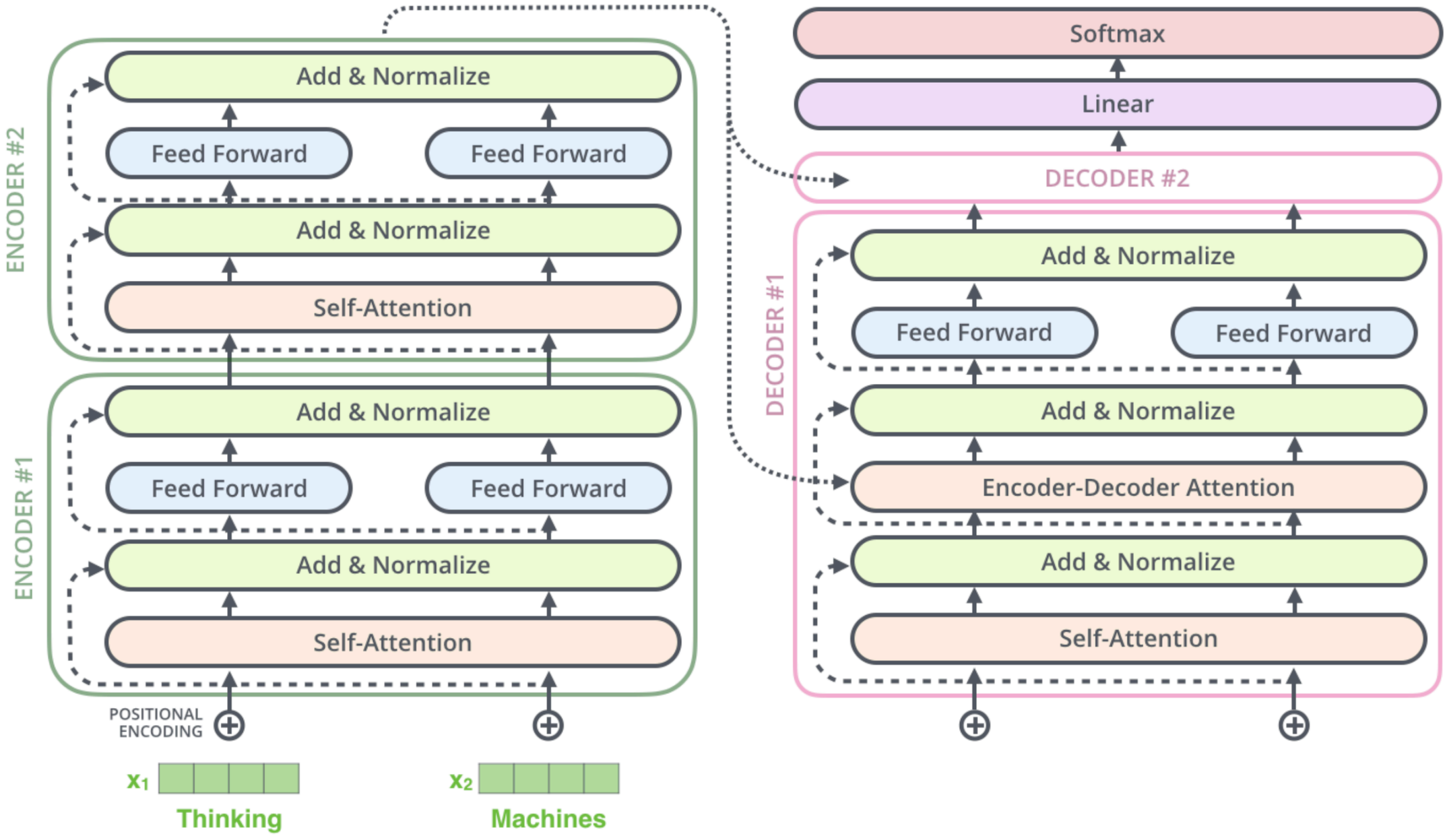
```
x = x + pos_emb
```

Transformer



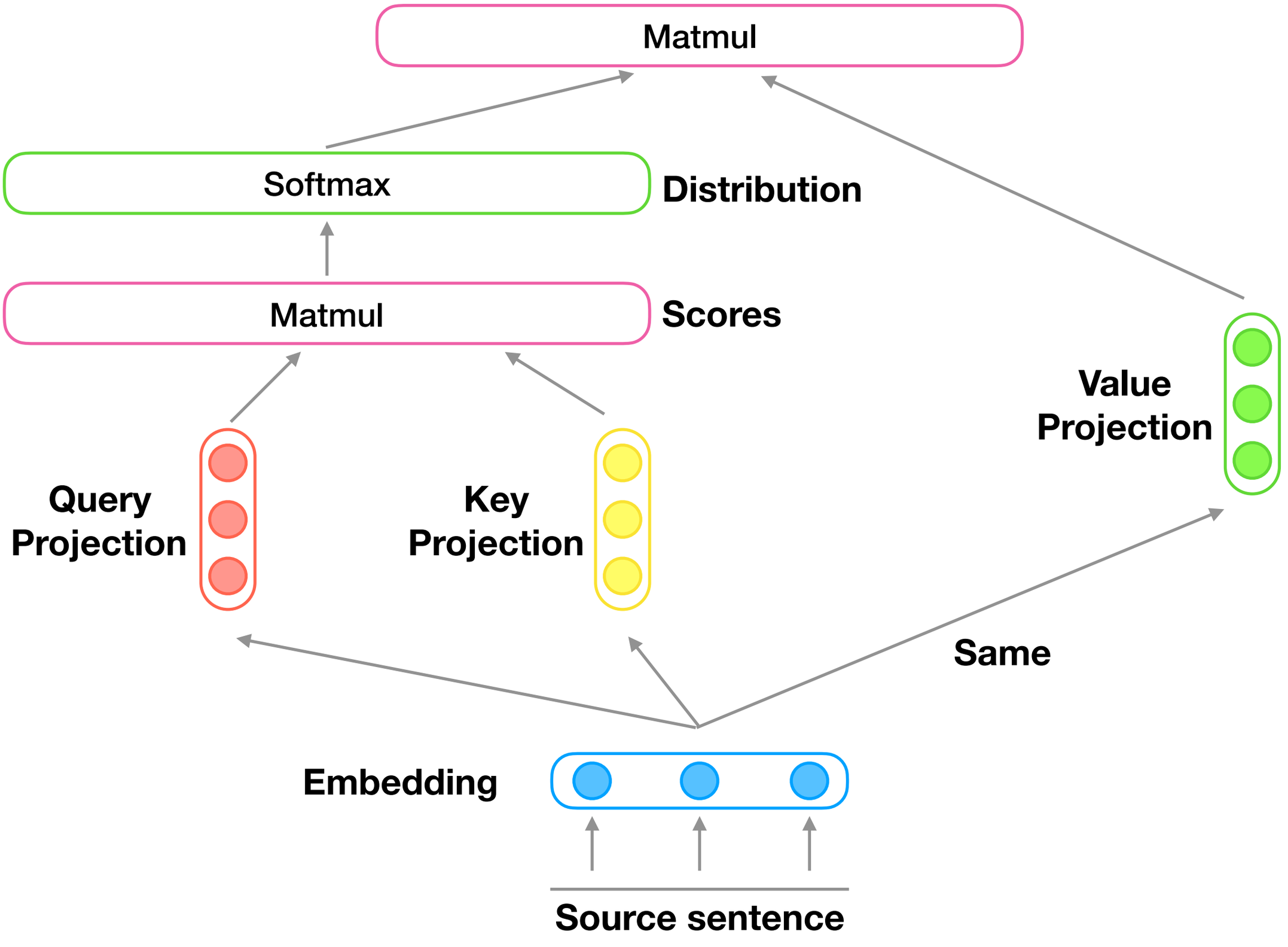
- Masking to ensure that we do not look into the future
- Masks setting to ***-inf*** then we apply softmax and this word becomes zero
- Linear and softmax is just word prediction

Transformer

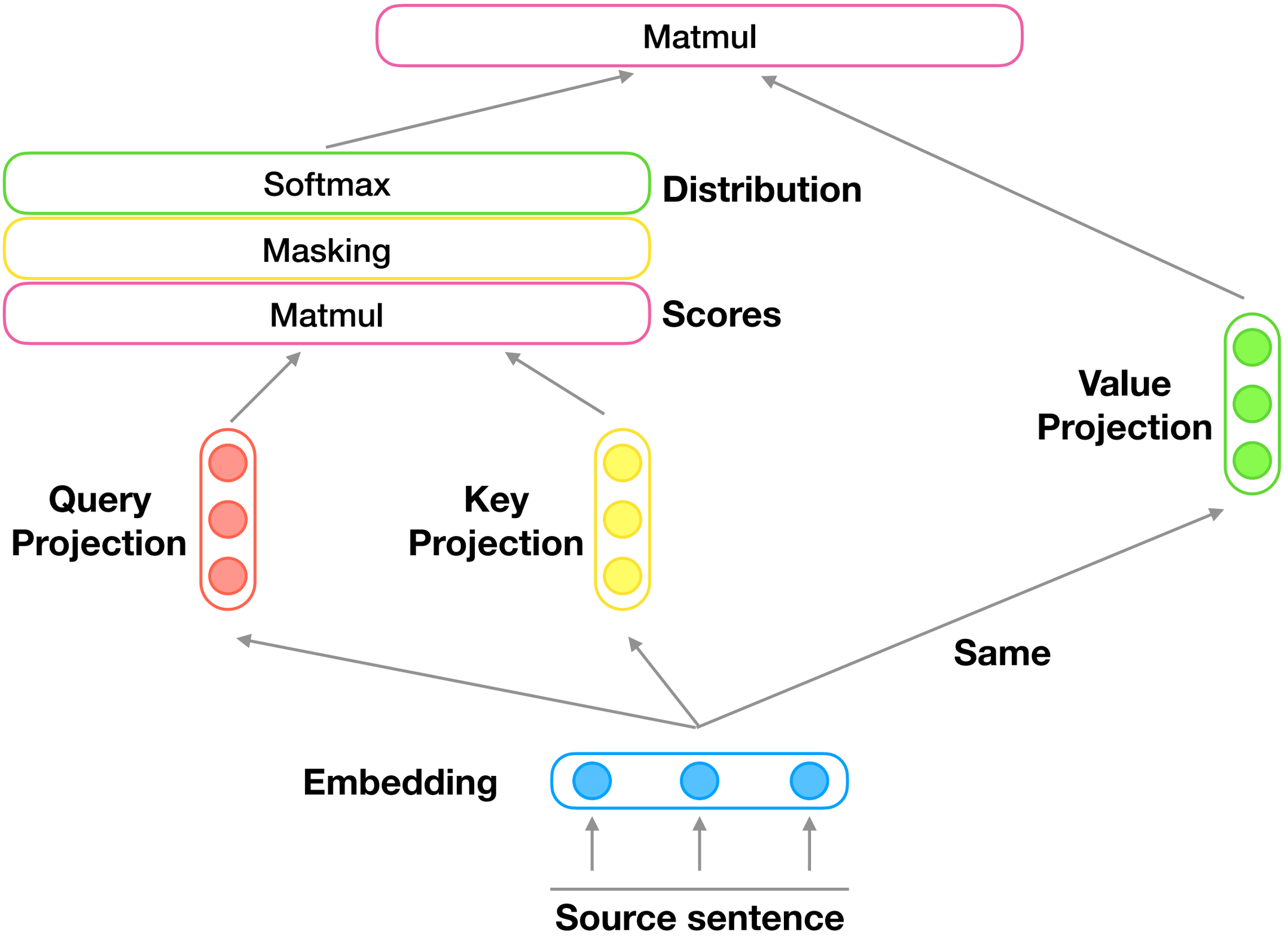


Masking

Self-Attention



Masked Self-Attention



Masking

Source text	I	am	space	invader

Masking

Source text

I am space invader

Attention Scores

0.11	0.04	0.05	0.3
0.19	0.53	0.42	0.37
0.81	0.21	0.05	0.09
0.51	0.43	0.12	0.03

Masking

Source text

I am space invader

Attention Scores

0.11	0.04	0.05	0.3
0.19	0.53	0.42	0.37
0.81	0.21	0.05	0.09
0.51	0.43	0.12	0.03

Masking

Future

Masked Attention Scores

0.11	-inf	-inf	-inf
0.19	0.53	-inf	-inf
0.81	0.21	0.05	-inf
0.51	0.43	0.12	0.03

Masking

Source text

I am space invader

Attention Scores

0.11	0.04	0.05	0.3
0.19	0.53	0.42	0.37
0.81	0.21	0.05	0.09
0.51	0.43	0.12	0.03

Masking

Future

Masked Attention Scores

Time

0.11	-inf	-inf	-inf
0.19	0.53	-inf	-inf
0.81	0.21	0.05	-inf
0.51	0.43	0.12	0.03

Masking

Source text

I am space invader

Attention Scores

0.11	0.04	0.05	0.3
0.19	0.53	0.42	0.37
0.81	0.21	0.05	0.09
0.51	0.43	0.12	0.03

Masking

→

Future

Masked Attention Scores

Time →

0.11	-inf	-inf	-inf
0.19	0.53	-inf	-inf
0.81	0.21	0.05	-inf
0.51	0.43	0.12	0.03

Masking

Source text I am space invader

Attention Scores

0.11	0.04	0.05	0.3
0.19	0.53	0.42	0.37
0.81	0.21	0.05	0.09
0.51	0.43	0.12	0.03

Masking

Future

Masked Attention Scores

Time

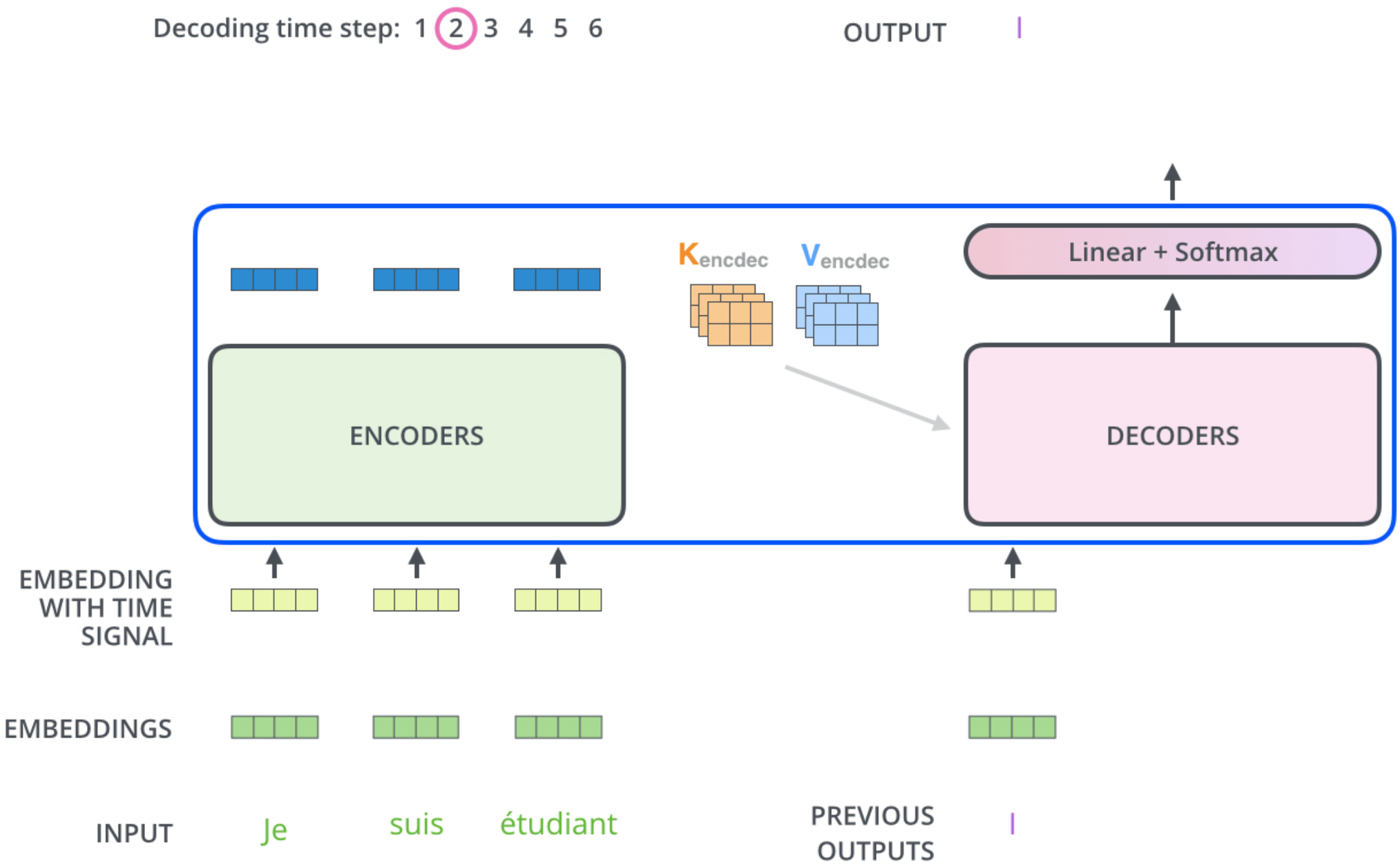
0.11	-inf	-inf	-inf
0.19	0.53	-inf	-inf
0.81	0.21	0.05	-inf
0.51	0.43	0.12	0.03

Softmax

Attention Distribution

1	0	0	0
0.48	0.52	0	0
0.45	0.21	0.34	0
0.25	0.16	0.33	0.26

Decoding


Decoding time step: 1 2 3 4 5 6
OUTPUT
I
Linear + Softmax
K encdec
V encdec
ENCODERS
DECODERS
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant
PREVIOUS OUTPUTS
I

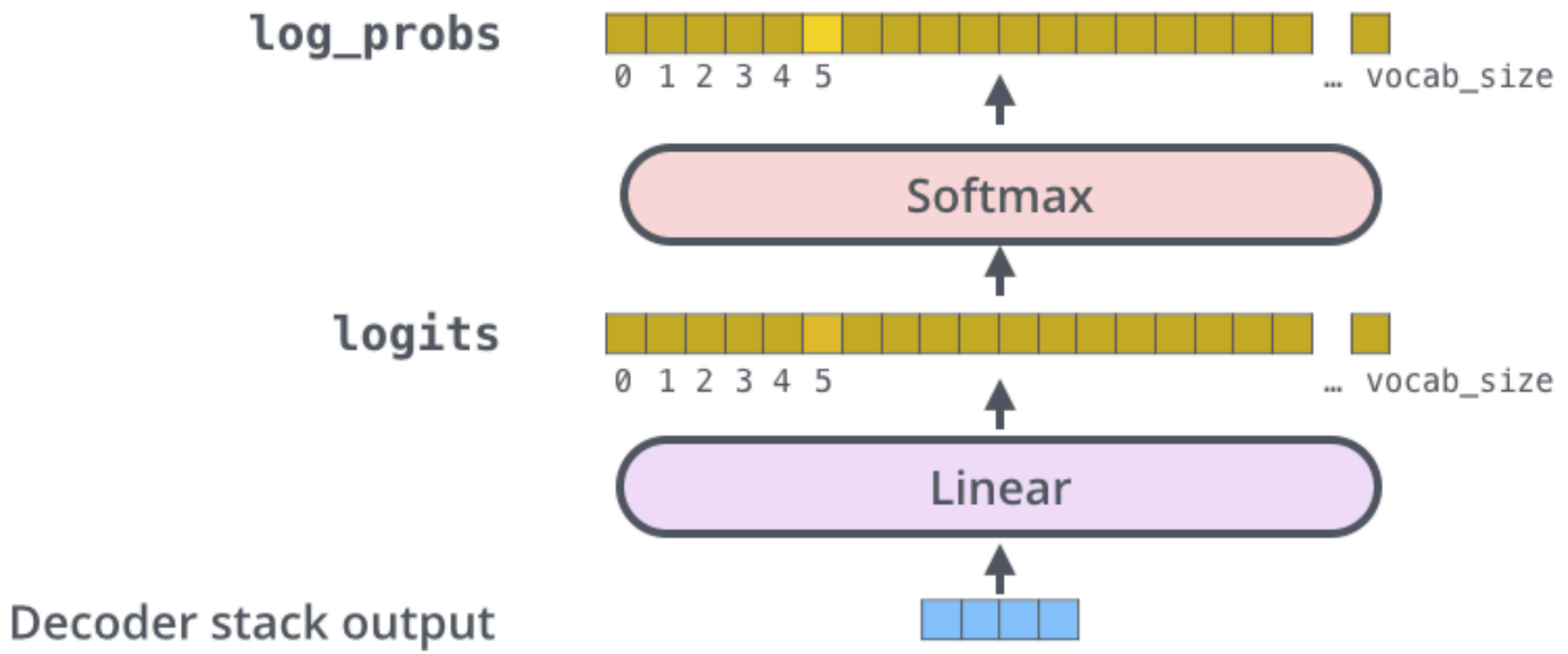
Decoding

Which word in our vocabulary is associated with this index?

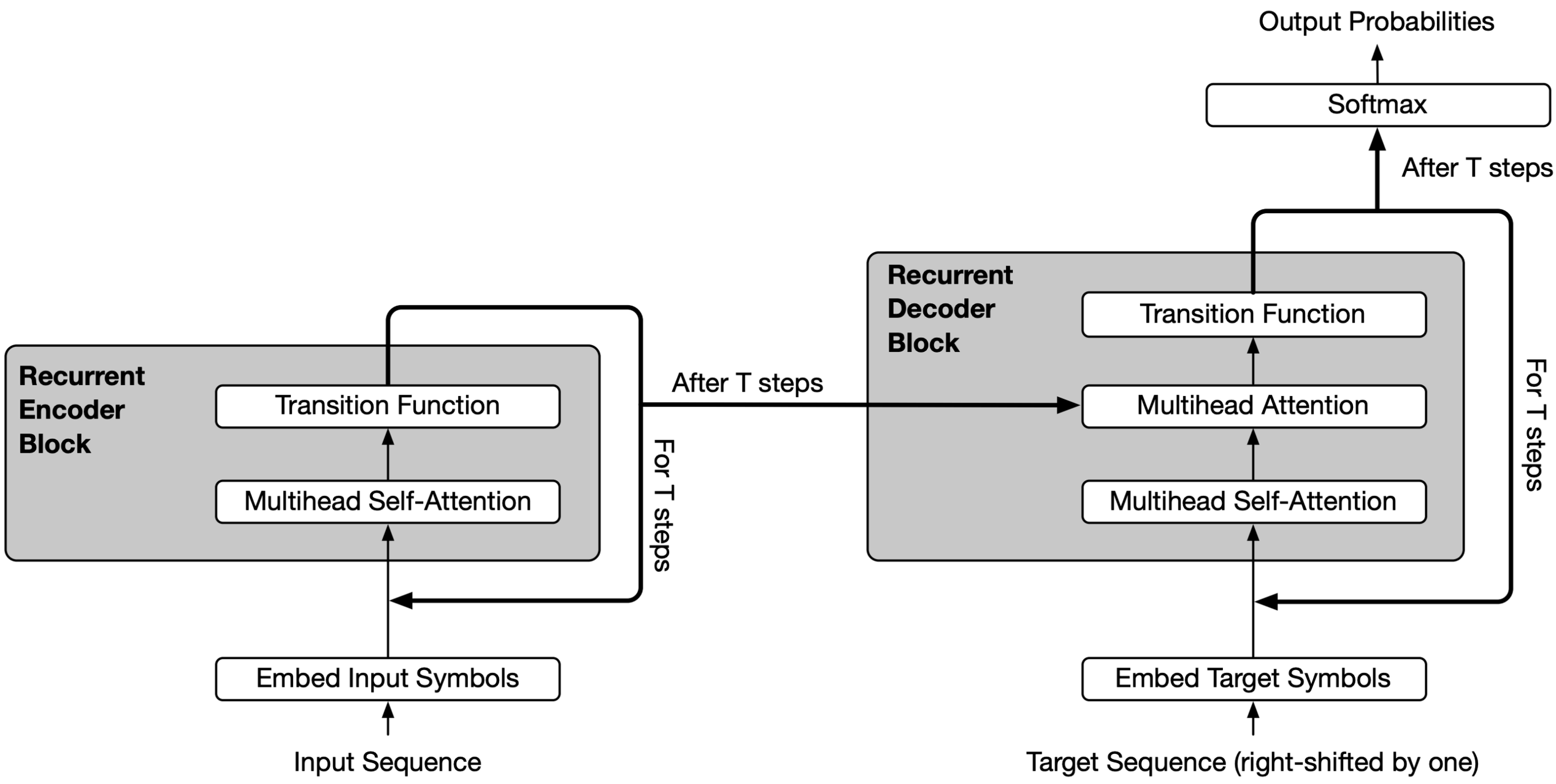
am

Get the index of the cell with the highest value (`argmax`)

5



Universal Transformer



Thanks for your Attention!

Boris Zubarev

@bobazooba