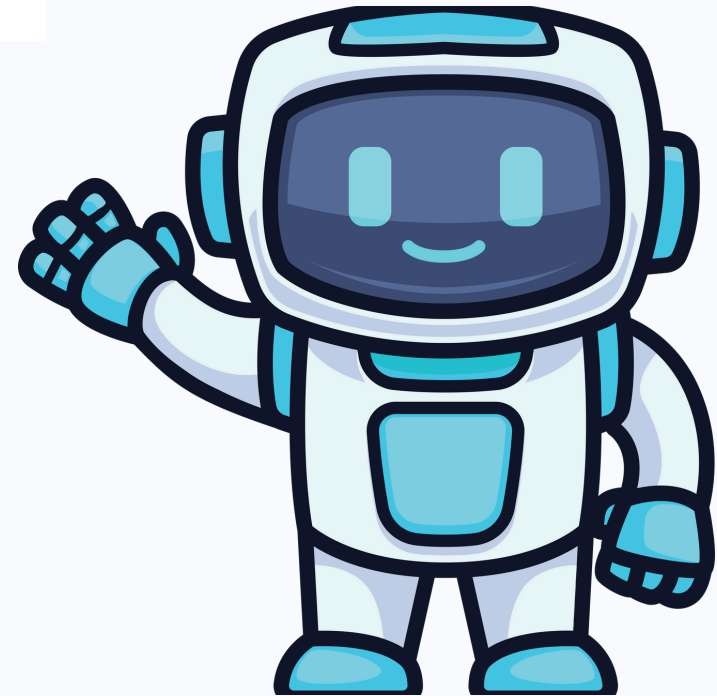
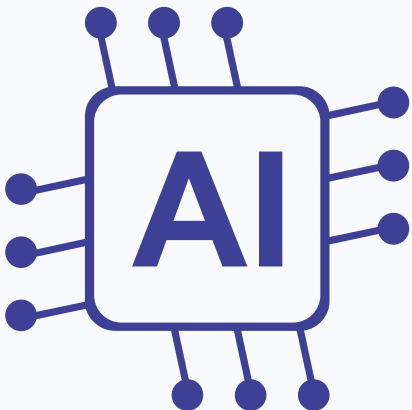


# Self-Attention

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

Unlocking  
Your Potential,  
Unleashing  
Your Success



Order

English

French

The



La

European

Economic

Area



Zone

Economique

Europeene

# Size Mismatch



English

French

The



La

European

Economic

Area



Europeene

Economique

Zone

# Machine Translation (2015) Encoder

The → Encoder (RNN) →



European → Encoder (RNN) →

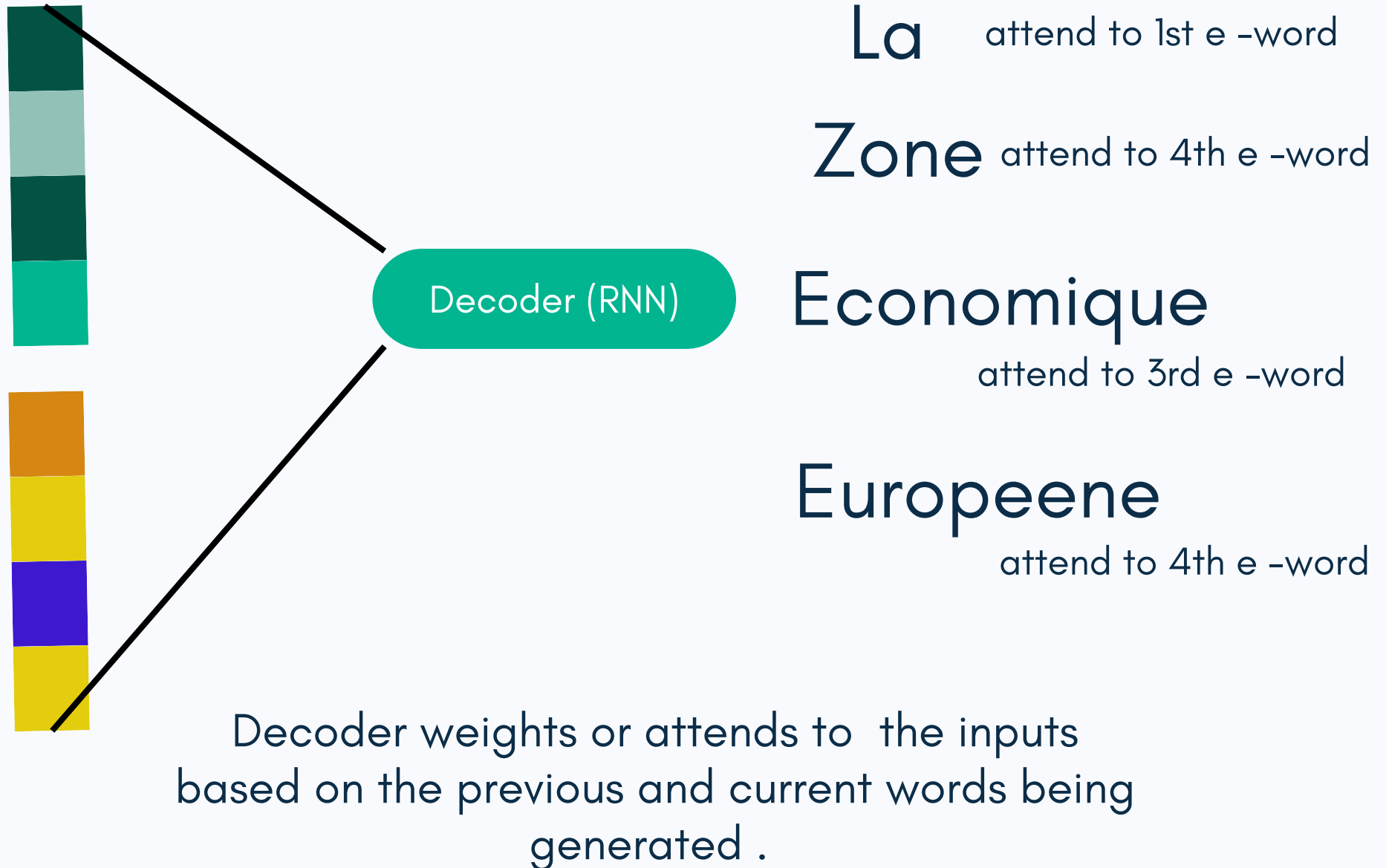


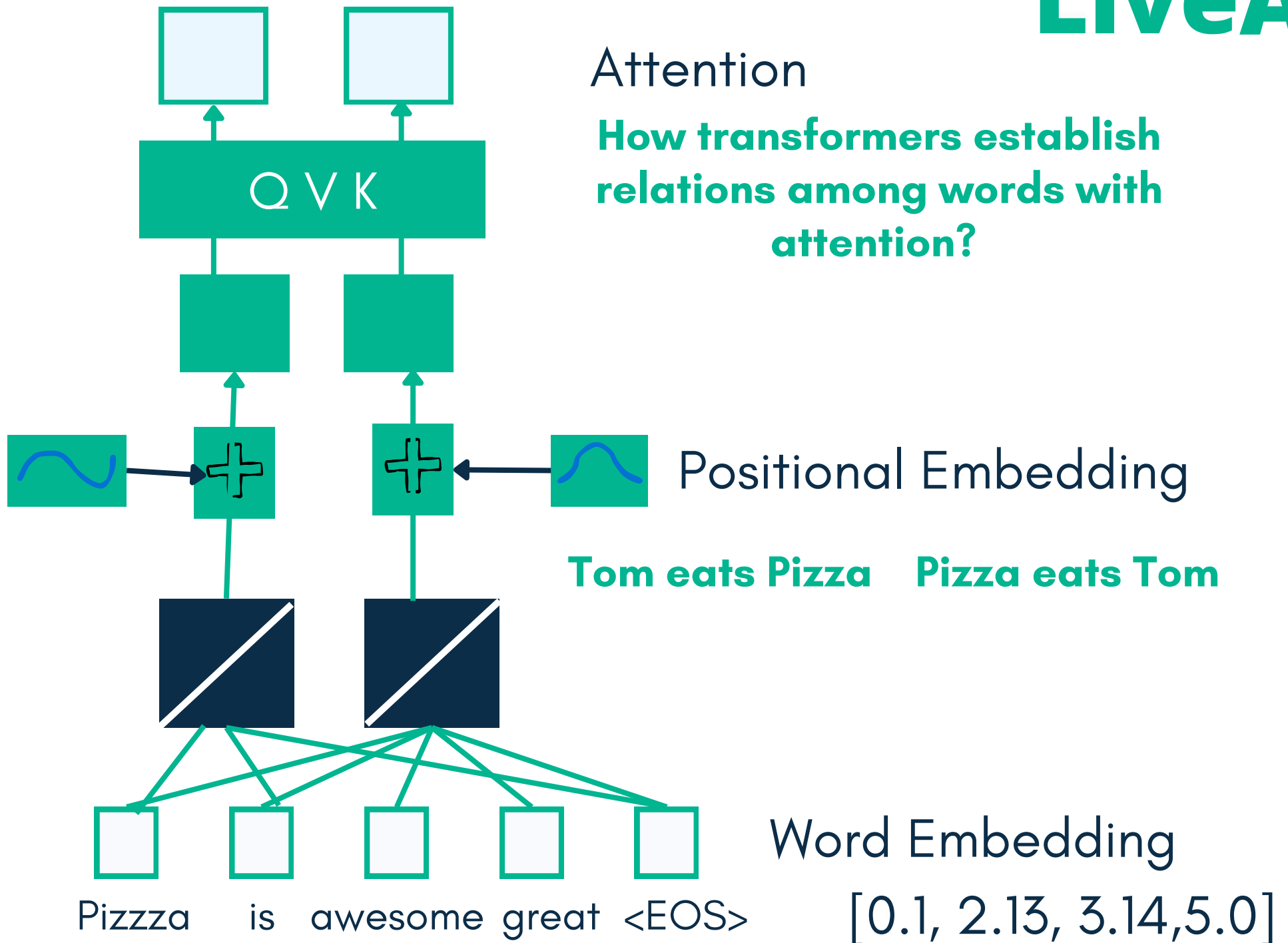
Economic

Area

Vectors which represent meaning of word or words in the context of sentences

# Machine Translation (2015) Decoder





The pizza came out of the oven and **it** tasted  
good



A diagram illustrating the concept of attention in a Transformer model. It features two red arrows. One arrow starts at the word 'it' (highlighted in teal) and points to the word 'pizza'. The other arrow starts at 'it' and points to the word 'oven'. Additionally, a long, curved red arrow starts at the end of the sentence and points back to 'pizza', representing a global or long-range attention mechanism.

Transformers have **attentions** to correctly  
associate the word it to pizza

The pizza came out of the oven and it tasted  
good

Self attention calculates the similarity  
between **The** and all the words in the  
sentence.



The **pizza** came out of the oven and **it**  
**tasted** good

If you have a lot of examples where the word  
**pizza** is related to **it** and **taste**

Then the similarity score between pizza, it  
and taste will be more

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

Key

Last Name	Room Number
Smith	200
Summer	201
Smeeth	202

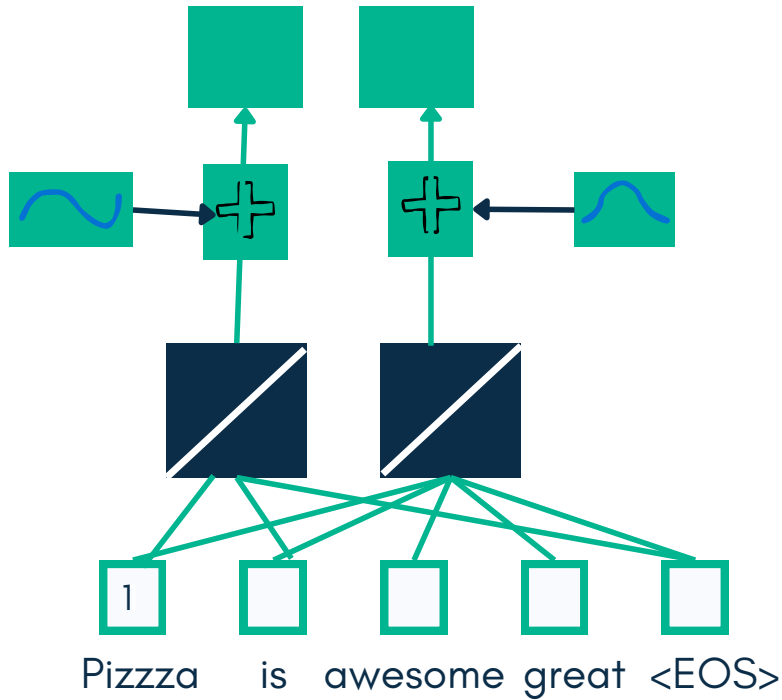
Summeth

Query

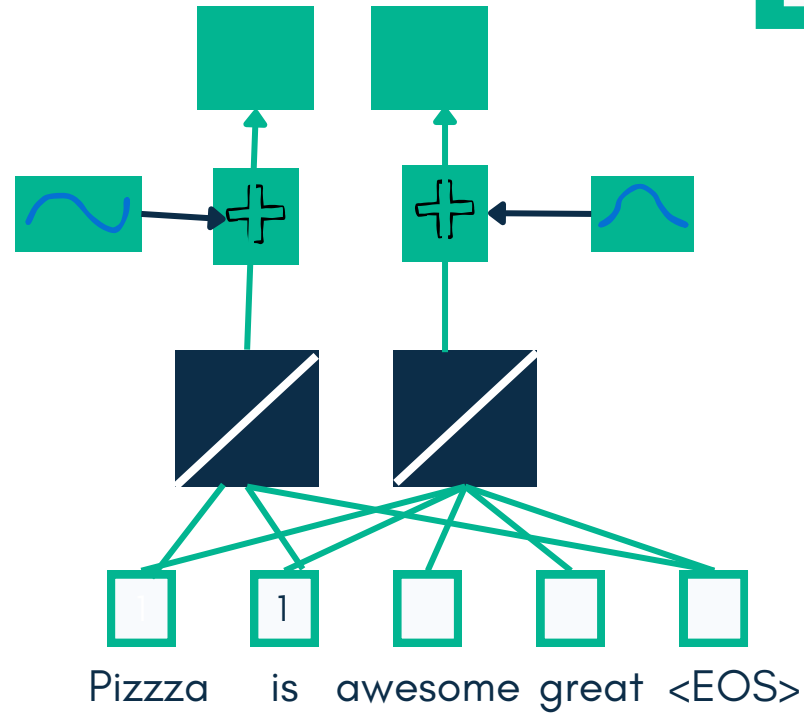
200

Value

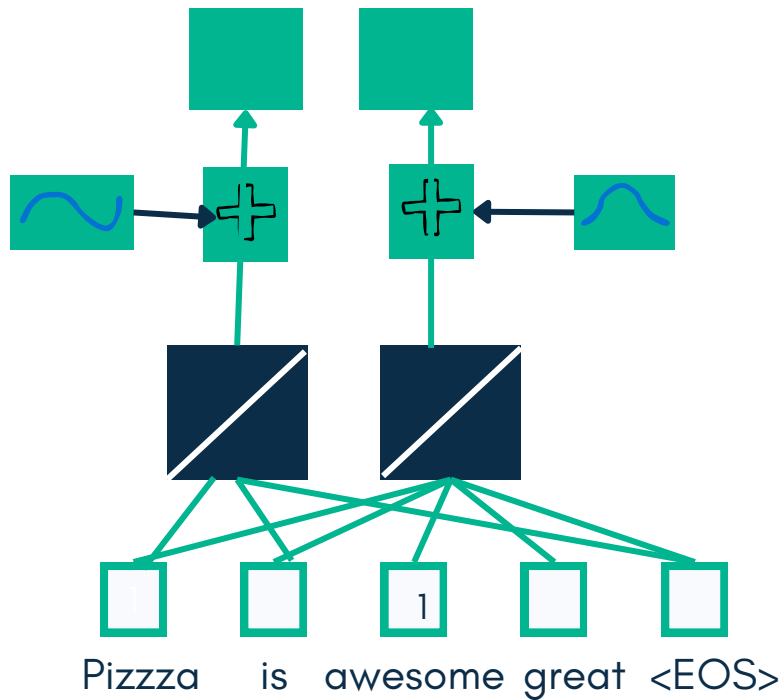
[0.1, 2.13]



[0.1, 2.13]



[0.21, 3.13]



Pizza  
is  
awesome  
<EOS>

0.1	2.13
0.11	2.13
0.21	3.13
0.8	0.9

Value

Value

<b>0.1</b>	<b>2.13</b>
0.11	2.13
0.21	3.13
0.8	0.9

Query Weights T

<b>0.78</b>	<b>2.0</b>
0.9	1.7

=

Query

..	..
..	..
0.8	0.8
..	..

Pizza  
is  
awesome  
<EOS>

Because we started with 2 encoded values we multiplies with 2-D weight matrix . If we start with **512-encoded** value we will have a **512X512** weight

# Value

<b>0.1</b>	<b>2.13</b>
0.11	2.13
0.21	3.13
0.8	0.9

# Key Weights T

<b>0.78</b>	<b>2.0</b>
0.9	1.7

=

# Key

..	..
..	..
0.18	0.81
..	..

Pizza  
is  
awesome  
<EOS>

Because we started with 2 encoded values we multiplies with 2-D weight matrix . If we start with 512-encoded value we will have a 512X512 weight

Value

<b>0.1</b>	<b>2.13</b>
0.11	2.13
0.21	3.13
0.8	0.9

Value Weights T

<b>0.78</b>	<b>2.0</b>
0.9	1.7

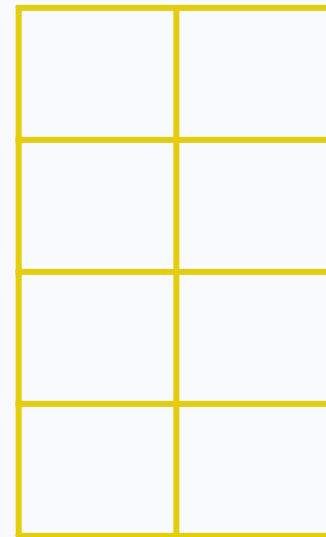
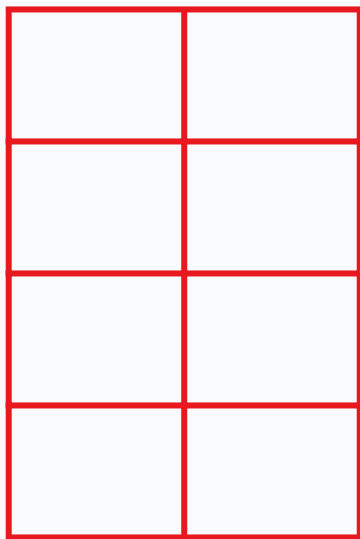
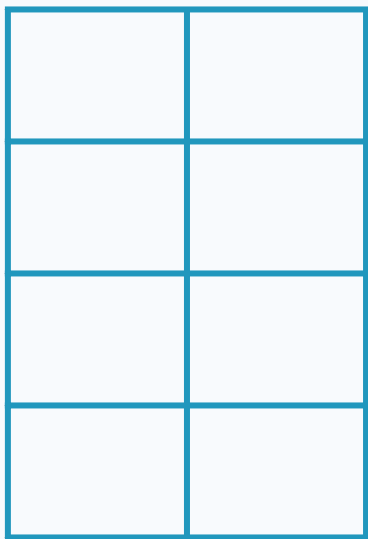
=

Value

..	..
..	..
0.18	0.81
..	..

Pizza  
is  
awesome  
<EOS>

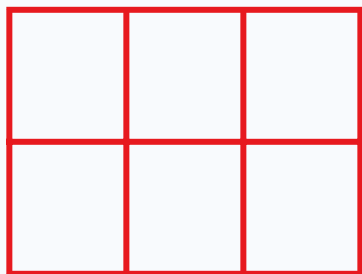
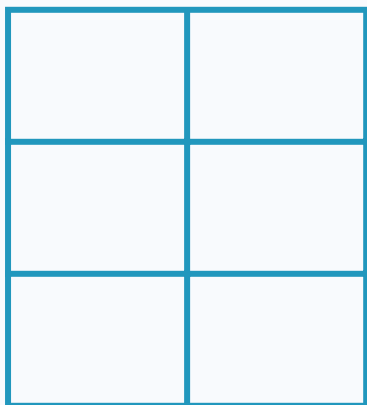
Because we started with 2 encoded values we multiply with 2-D weight matrix. If we start with **512-encoded** value we will have a **512X512** weight



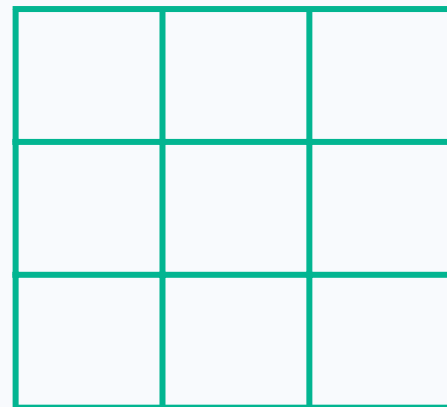
V

Q

$K^T$



=



unscaled dot product and scale each dot product  
similarity by  $\sqrt{2}$  -- encoded word dimension size

Softmax



=


1  
1  
1

Pizza is awesome

Pizza  
is  
awesome

<b>0.38</b>	<b>0.4</b>	<b>0.9</b>

Pizza is 0.38% similar to Pizza, 0.4% similar to is etc...



Pizza is awesome

Value Matrix

0.38	0.4	0.24

X

0.6	
-0.35	
3.86	

= 1.0

In other words the percentages that comes out of the softmax .... tells us how much influence each word should have on the final encoding for a given word

Pizza is awesome

Value Matrix

0.38	0.4	0.24

X

0.6	
-0.35	
3.86	

= 1.0

we calculate 36% of the first value for Pizza

we calculate 40% of the first value for is

we calculate 24% of the first value for  
awesome

# Self- Attention Score

Pizza is awesome

0.38	0.4	0.24

X

Value Matrix

0.6	
-0.35	
3.86	

=

1.0	1.9
0.2	0.4
3.86	2.2

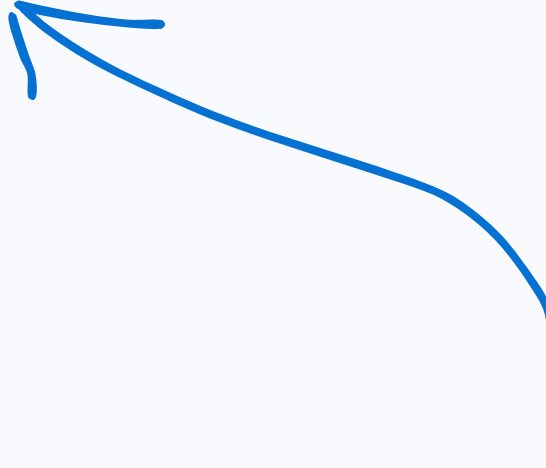
Pizza  
is  
awesome

1.0	1.9
0.2	0.4
3.86	2.2

```
class SelfAttention(nn.Module):
```




class selfAttention inherits  
from nn.Module




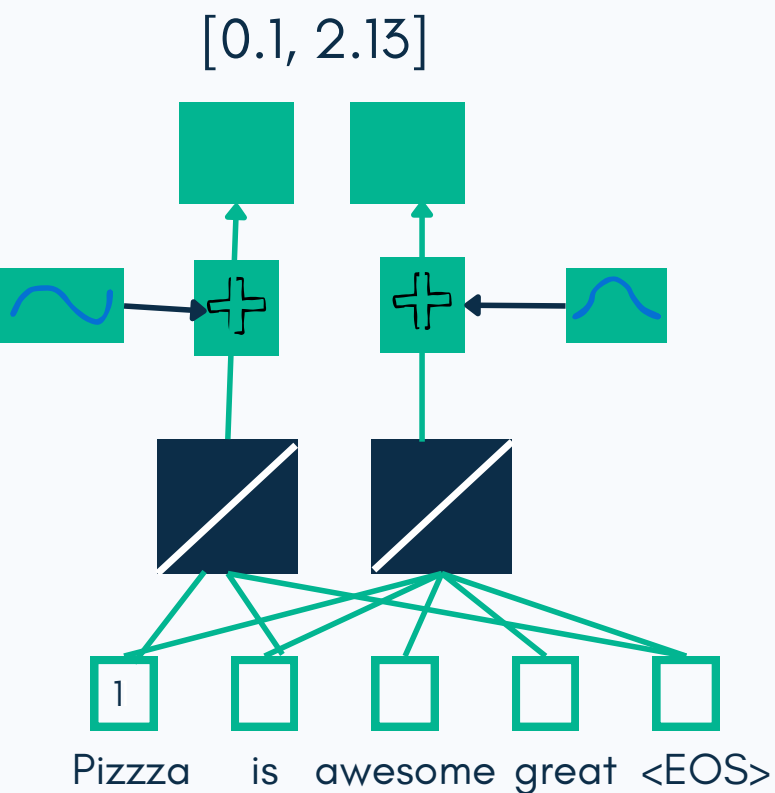
Its the base class of all  
neural network modules

```
class SelfAttention(nn.Module):
```

```
    def __init__(self, d_model=2, row_dim, column_dim )
```

 `__init()` method

 Embedding size of per token (ie 2 or 256 etc)




Pizza  
is  
awesome  
<EOS>

<b>0.1</b>	<b>2.13</b>
0.11	2.13
0.21	3.13
0.8	0.9


2 **encoded  
value** per  
token

```
class SelfAttention(nn.Module):
```

```
    def __init__(self, d_model=2, row_dim, column_dim )
```



\_\_\_init()\_\_\_ method



convenience parameters to easily modify row  
and column index of the data this could be  
batches of data

```
class SelfAttention(nn.Module):  
    def __init__(self, d_model=2, row_dim, column_dim )  
        super().__init__()
```

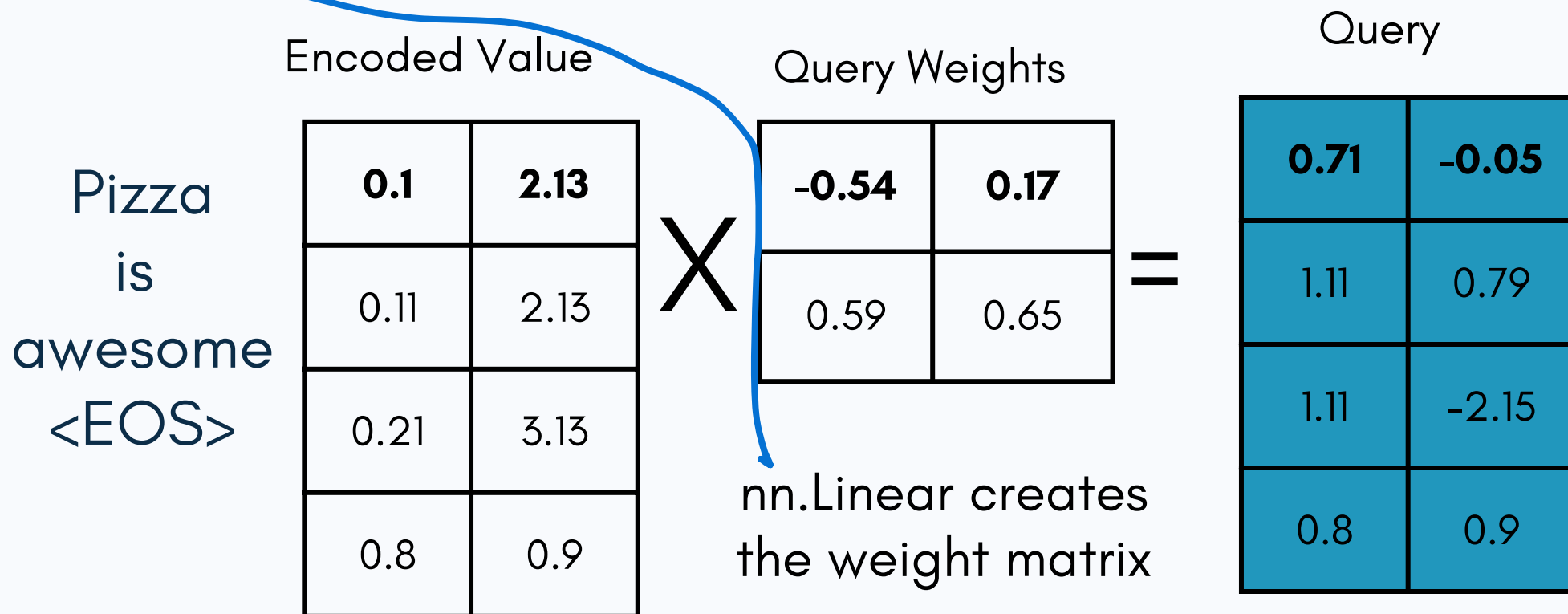


Parent class init method  
because we are inheriting the  
class nn.Modules

```
class SelfAttention(nn.Module):
```

```
    def __init__(self, d_model=2, row_dim, column_dim )  
        super().__init__()
```

```
        self.W_q = nn.Linear(in_feature=d_model,  
                               out_feature=d_model, bias=False)
```

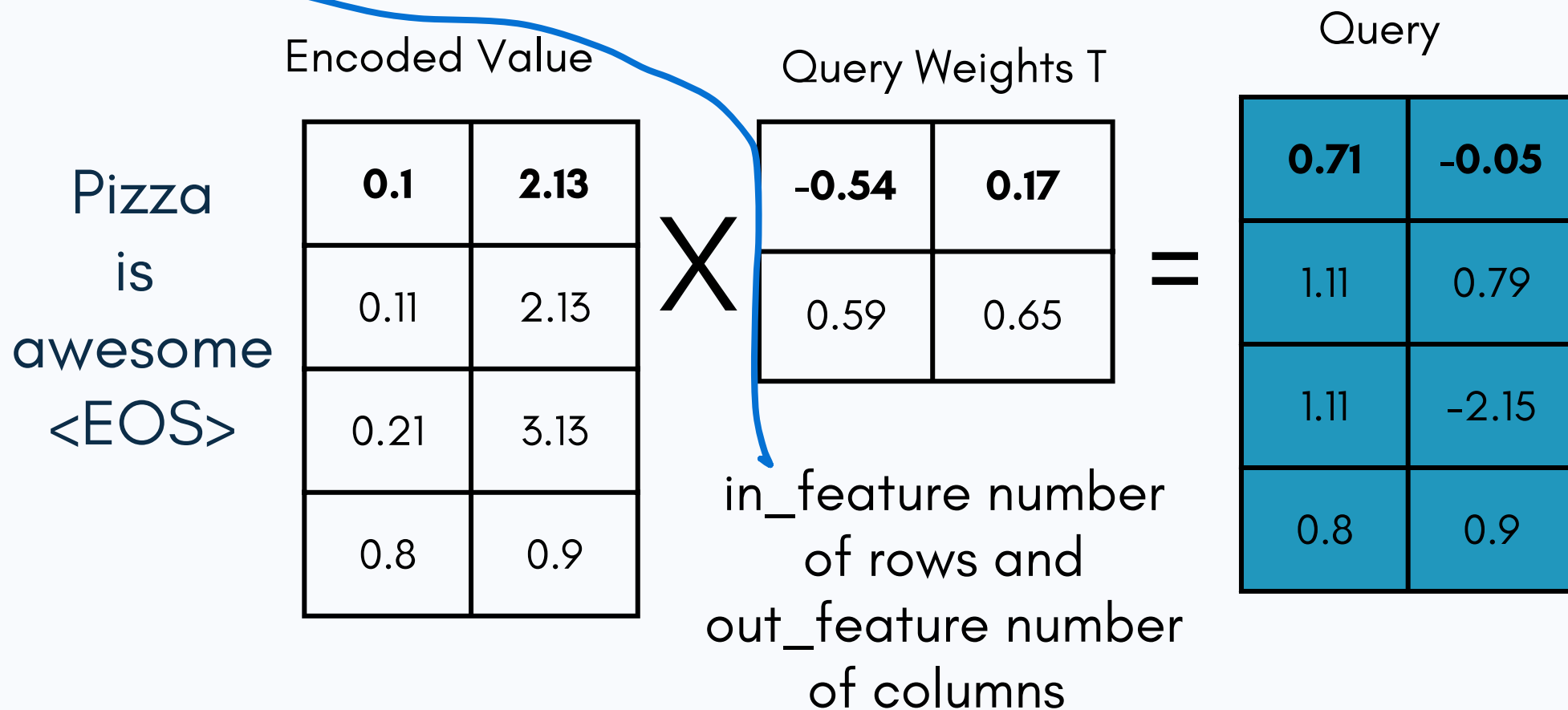




```
class SelfAttention(nn.Module):
```

```
    def __init__(self, d_model=2, row_dim, column_dim )  
        super().__init__()
```

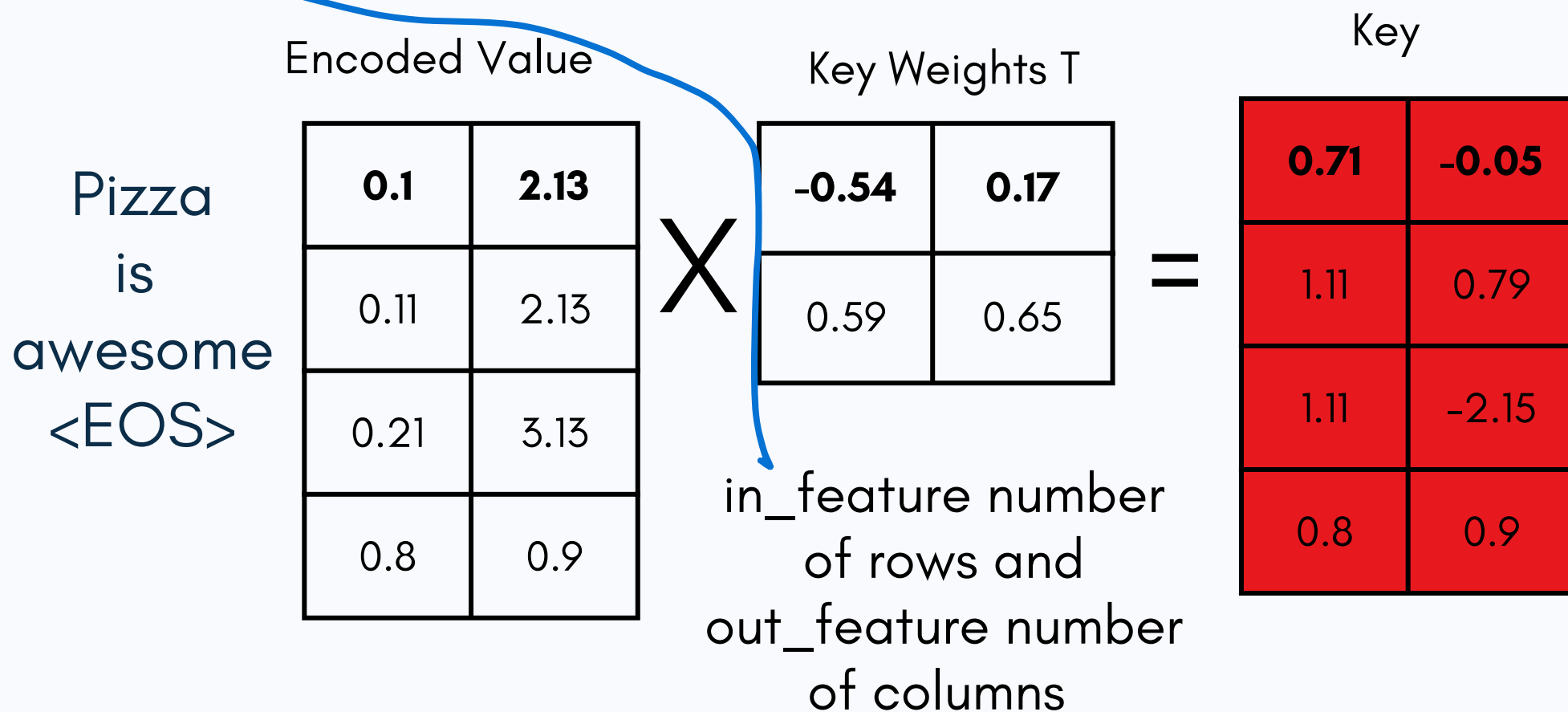
```
        self.W_q = nn.Linear(in_feature=d_model,  
                               out_feature=d_model, bias=False)
```



```
class SelfAttention(nn.Module):
```

```
    def __init__(self, d_model=2, row_dim, column_dim )  
        super().__init__()
```

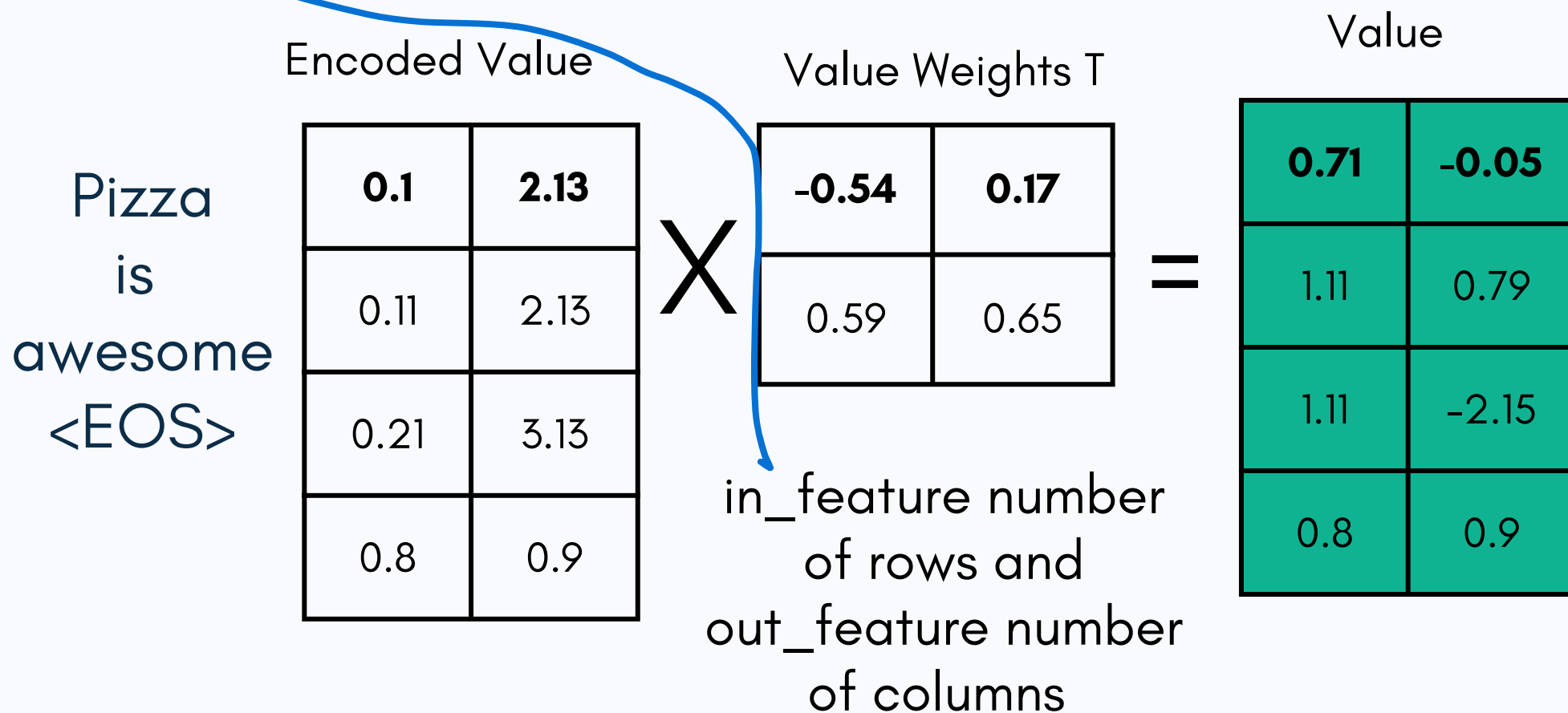
```
        self.W_k = nn.Linear(in_feature=d_model,  
                               out_feature=d_model, bias=False)
```



```
class SelfAttention(nn.Module):
```

```
    def __init__(self, d_model=2, row_dim, column_dim )  
        super().__init__()
```

```
        self.W_v = nn.Linear(in_feature=d_model,  
                               out_feature=d_model, bias=False)
```



```
class SelfAttention(nn.Module):  
    def __init__(self, d_model=2, row_dim, column_dim )  
        super().__init()  
  
        self.W_q = nn.Linear(in_feature=d_model,  
                               out_feature=d_model, bias=False)  
        self.W_k = nn.Linear(in_feature=d_model,  
                               out_feature=d_model, bias=False)  
        self.W_v = nn.Linear(in_feature=d_model,  
                               out_feature=d_model, bias=False)  
        self.row_dim = row_dim  
        self.column_dim = column_dim
```

```
class SelfAttention(nn.Module):  
    def __init__(self, d_model=2, row_dim, column_dim )  
        super().__init()  
  
        self.W_q = nn.Linear(in_feature=d_model,  
                               out_feature=d_model, bias=False)  
        self.W_k = nn.Linear(in_feature=d_model,  
                               out_feature=d_model, bias=False)  
        self.W_v = nn.Linear(in_feature=d_model,  
                               out_feature=d_model, bias=False)  
        self.row_dim = row_dim  
        self.column_dim = column_dim
```

```
class SelfAttention(nn.Module):
```

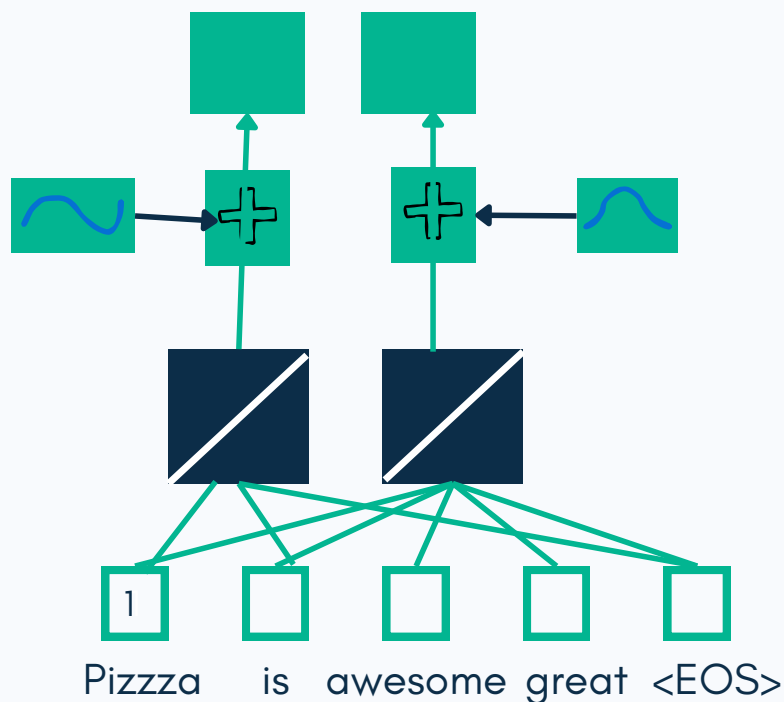
```
def forward(self, token_encoding ):
```

```
    q= self.W_q(token_encoding)
```

Word Embedding and  
positional encoding of each  
token

Matrix multiplication between  
the Weight matrix and returns  
the Query matrix

[0.1, 2.13]



```

class SelfAttention(nn.Module):
    def forward(self, token_encoding ):
        q= self.W_q(token_encoding)
        k= self.W_k(token_encoding)
        v= self.W_v(token_encoding)
        sims = torch.matmul(q,k.transpose(dim0= self.row_dim,
dim1= self.column_dim))

```

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

Similarities between all possible combinations of Queries and keys

```
class SelfAttention(nn.Module):  
    def forward(self, token_encoding ):  
        q= self.W_q(token_encoding)  
        k= self.W_k(token_encoding)  
        v= self.W_v(token_encoding)  
        sims = torch.matmul(q,k.transpose(dim0= self.row_dim,  
dim1= self.column_dim)  
scaled_sims = sims/torch.tensor(k.size(self.col_dim)**0.5)
```

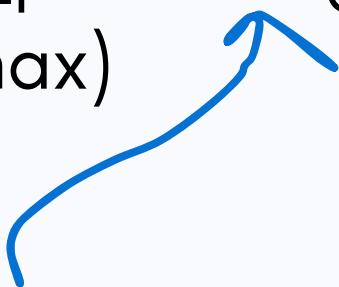
$$Attention(Q, K, V) = Softmax\left(\frac{QK^T}{\sqrt{d_K}}\right)V$$



```
class SelfAttention(nn.Module):  
    def forward(self, token_encoding ):  
        q= self.W_q(token_encoding)  
        k= self.W_k(token_encoding)  
        v= self.W_v(token_encoding)  
        sims = torch.matmul(q,k.transpose(dim0= self.row_dim,  
dim1= self.column_dim))  
        scaled_sims = sims/torch.tensor(k.size(self.col_dim)**0.5)  
        attention_percentages = F.softmax(scaled_sims, dim=  
self.col_max)
```

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

```
class SelfAttention(nn.Module):  
    def forward(self, token_encoding ):  
        q= self.W_q(token_encoding)  
        k= self.W_k(token_encoding)  
        v= self.W_v(token_encoding)  
        sims = torch.matmul(q,k.transpose(dim0= self.row_dim,  
dim1= self.column_dim))  
        scaled_sims = sims/torch.tensor(k.size(self.col_dim)**0.5)  
        attention_percentages = F.softmax(scaled_sims, dim=  
self.col_max)
```



Applying the softmax function to scaled similarities  
determines the percentage of influence of each  
token should have on the other

```
class SelfAttention(nn.Module):  
    def forward(self, token_encoding ):  
        q= self.W_q(token_encoding)  
        k= self.W_k(token_encoding)  
        v= self.W_v(token_encoding)  
        sims = torch.matmul(q,k.transpose(dim0= self.row_dim,  
dim1= self.column_dim))  
        scaled_sims = sims/torch.tensor(k.size(self.col_dim)**0.5)  
        attention_percentages = F.softmax(scaled_sims, dim=  
self.col_max)  
        attention_scores = torch.matmul(attention_percentages, v)
```

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