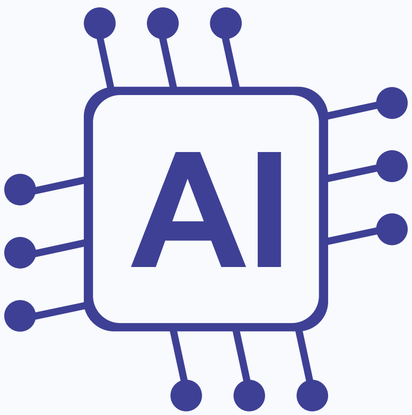
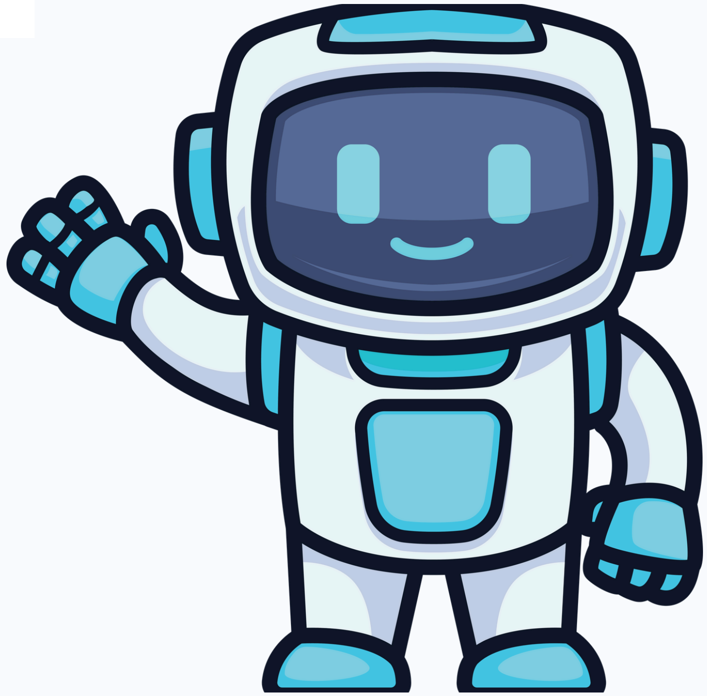
LiveAI

Self-Attention

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

Unlocking
Your Potential,
Unleashing
Your Success





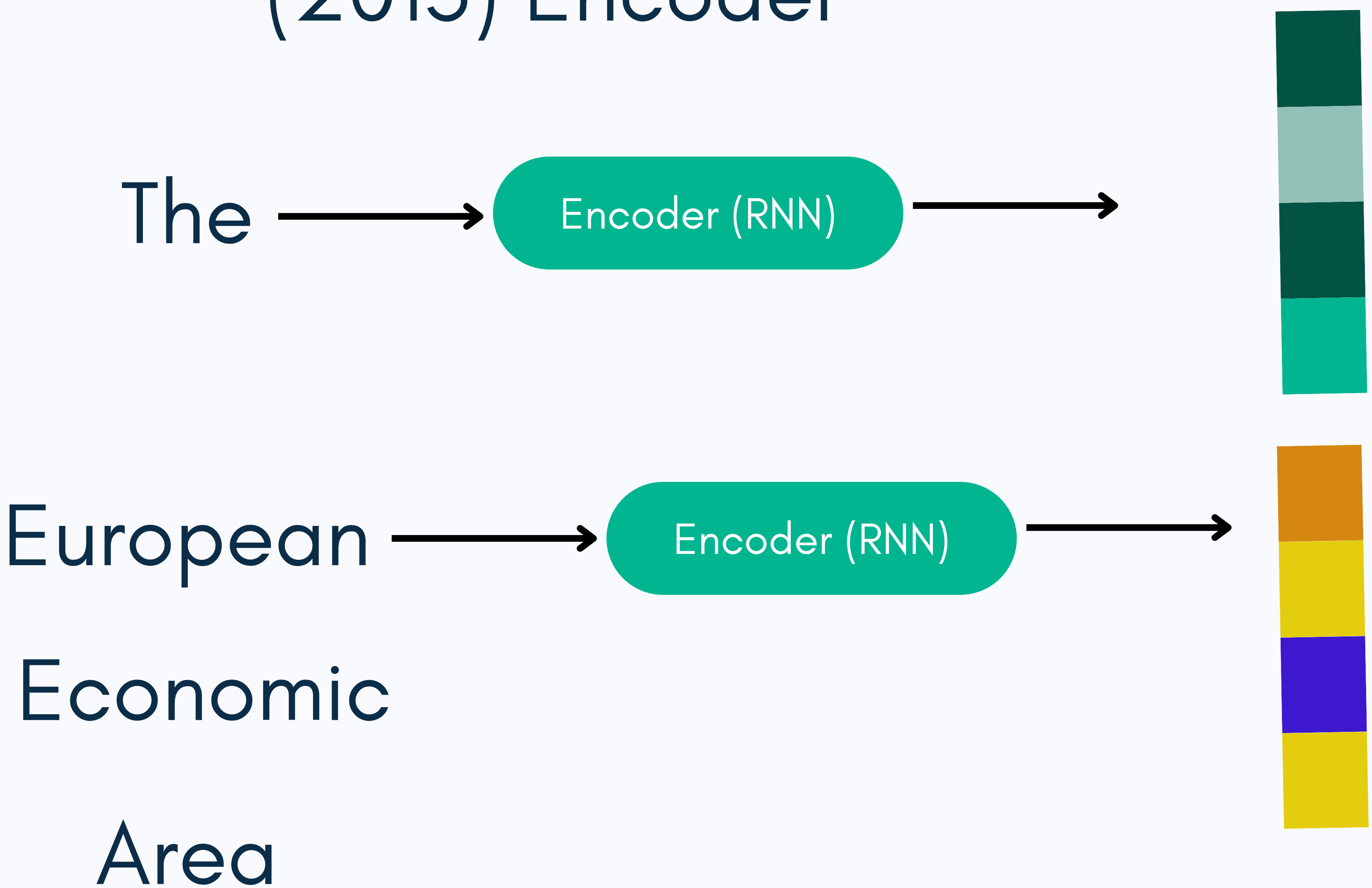
Order

English	French
The	La
European	Zone
Economic	Economique
Area	Europeene

Size Mismatch

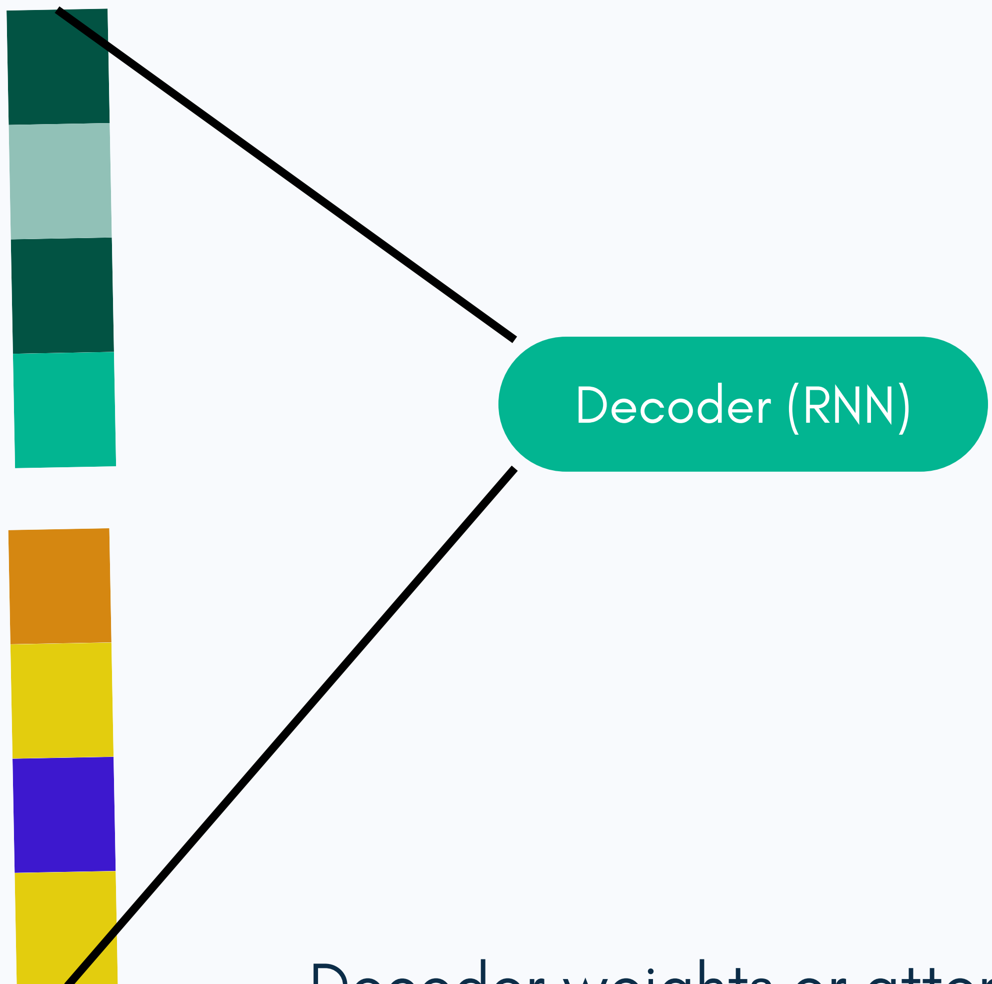
English	French
The	La
European Economic Area	Europeene Economique Zone

Machine Translation (2015) Encoder



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

Machine Translation (2015) Decoder



La attend to 1st e -word

Zone attend to 4th e -word

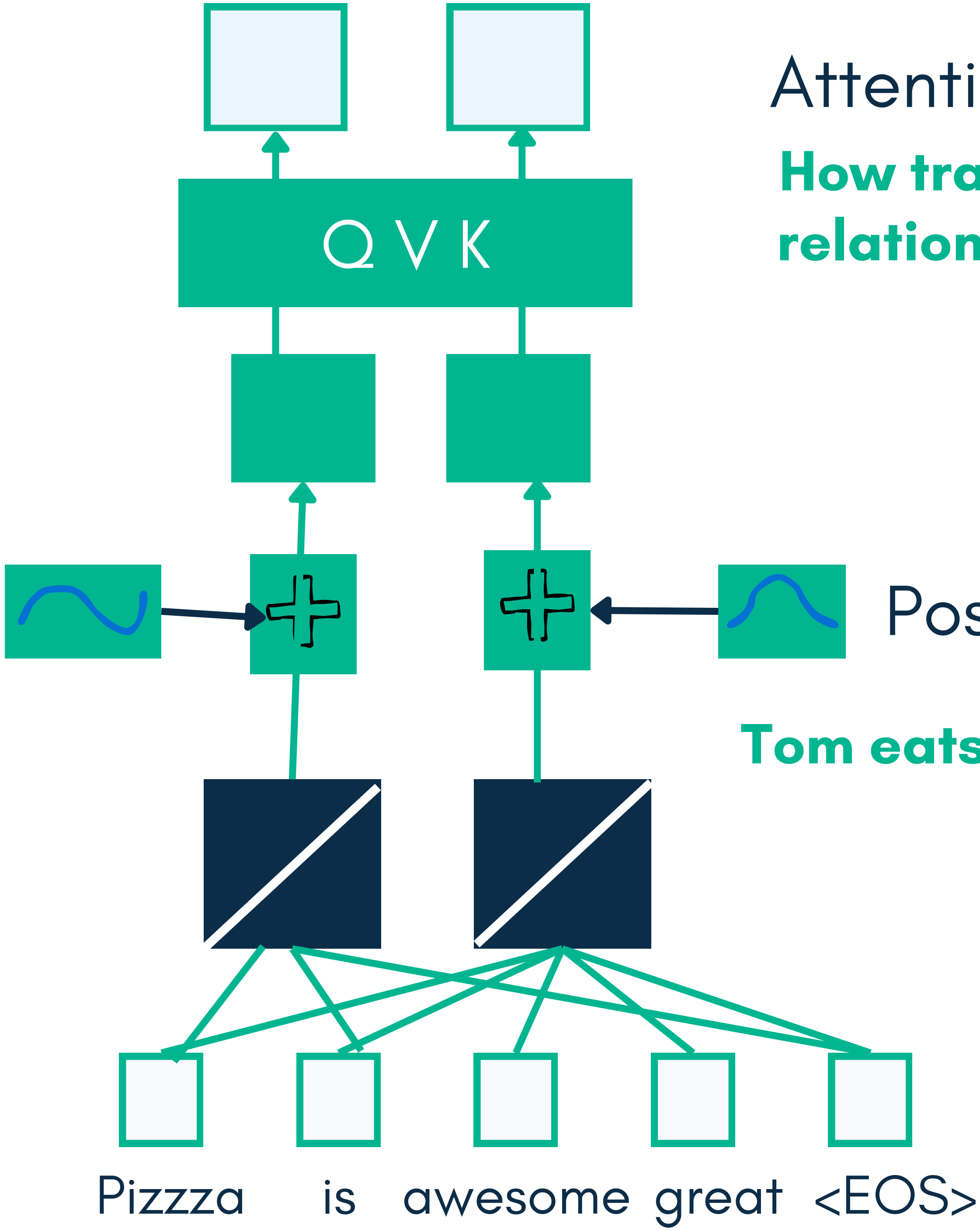
Economique attend to 3rd e -word

Europeene attend to 4th e -word

Decoder weights or attends to the inputs based on the previous and current words being generated .

Attention

How transformers establish relations among words with attention?



Positional Embedding

Tom eats Pizza **Pizza eats Tom**

Word Embedding

[0.1, 2.13, 3.14,5.0]

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

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 similiarity score between pizza, it and taste will be more

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

Key

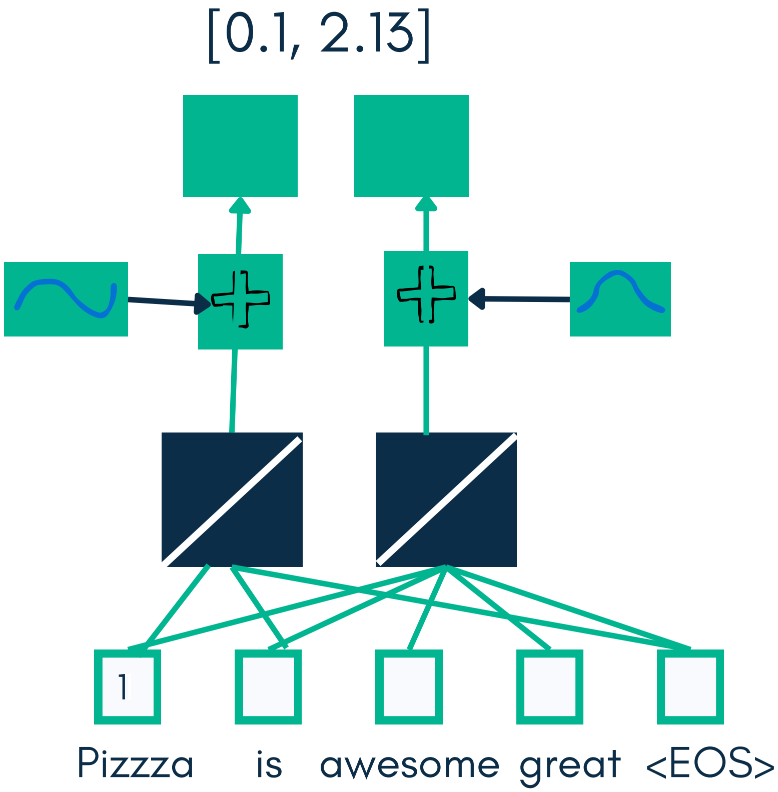
Last Name	Room Number
Smith	200
Summer	201
Smeeth	202

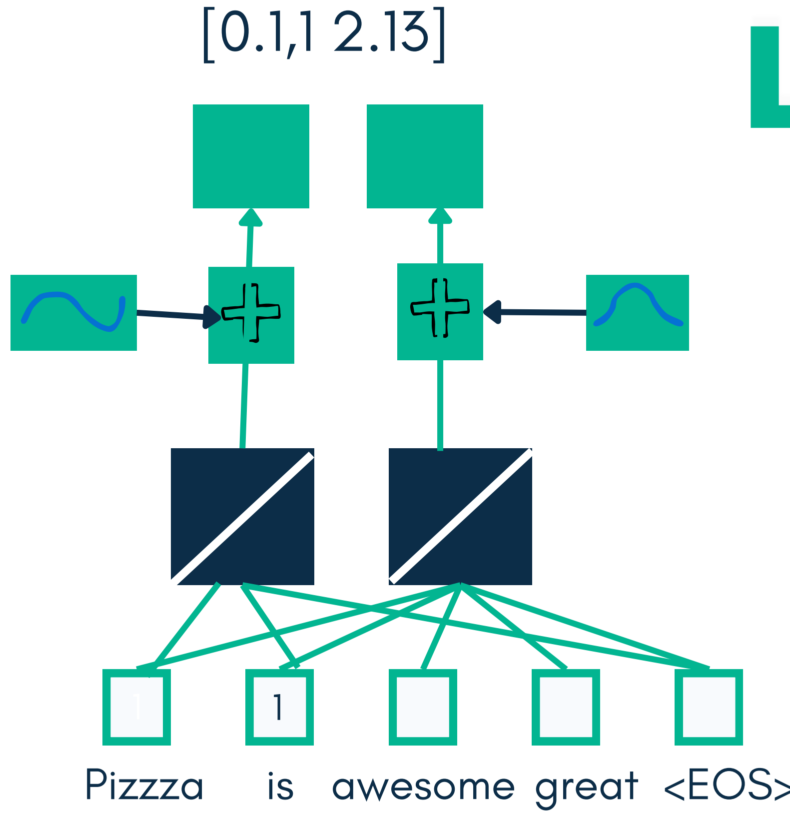
Summeth

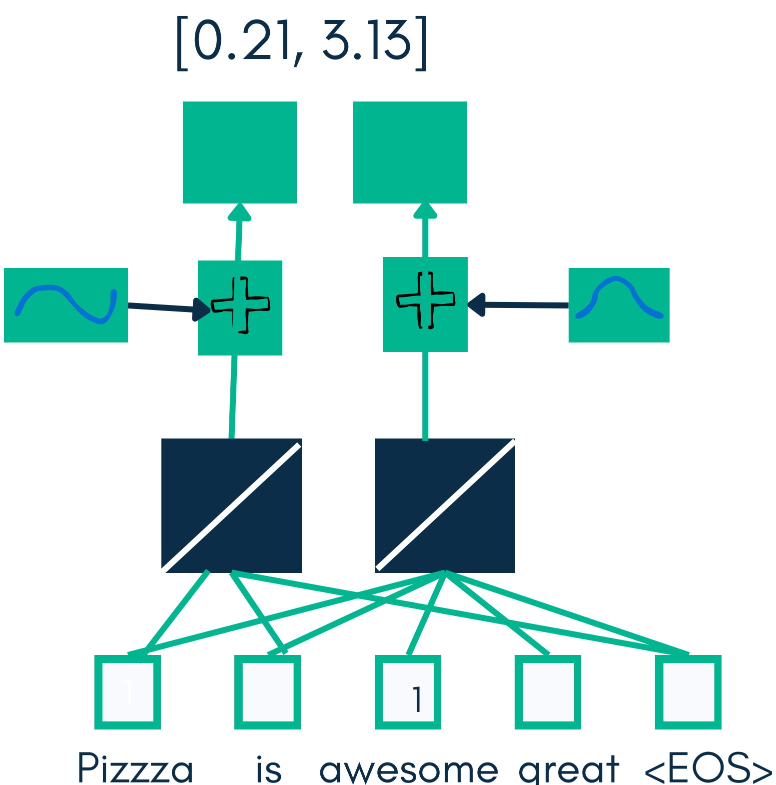
Query

200

Value







Pizza
is
awesome
<EOS>

0.1	**2.13**
0.11	2.13
0.21	3.13
0.8	0.9

Value

Value

0.1	**2.13**
0.11	2.13
0.21	3.13
0.8	0.9

Query Weights T

0.78	**2.0**
0.9	1.7

=

Query

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

Pizza
is
awesome
<EOS>

Because we stareted 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

0.1	**2.13**
0.11	2.13
0.21	3.13
0.8	0.9

Key Weights T

0.78	**2.0**
0.9	1.7

=

Key

..	..	Pizza
..	..	is
0.18	0.81	awesome
..	..	<EOS>

Because we stareted 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

0.1	**2.13**
0.11	2.13
0.21	3.13
0.8	0.9

Value Weights T

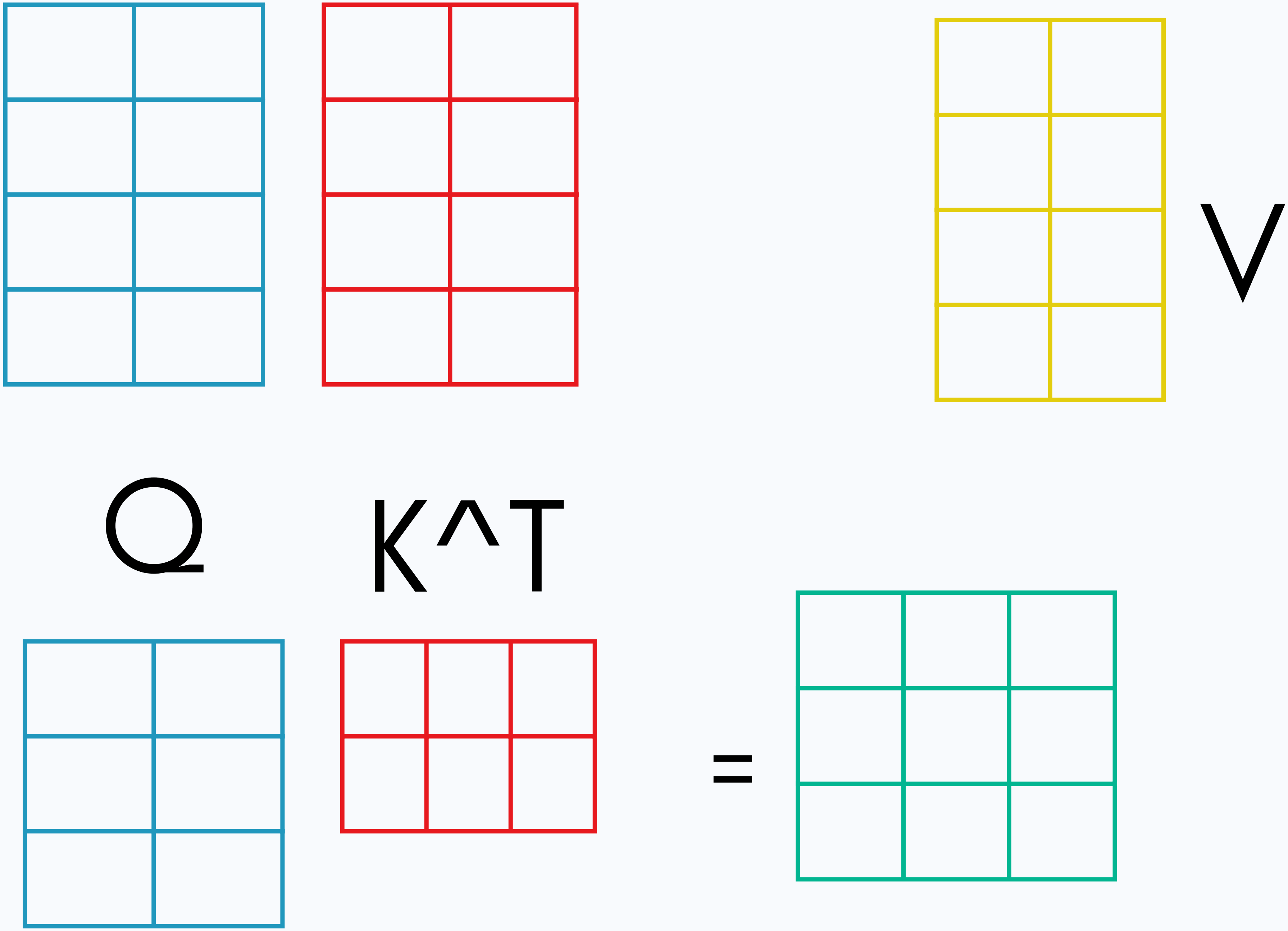
0.78	**2.0**
0.9	1.7

=

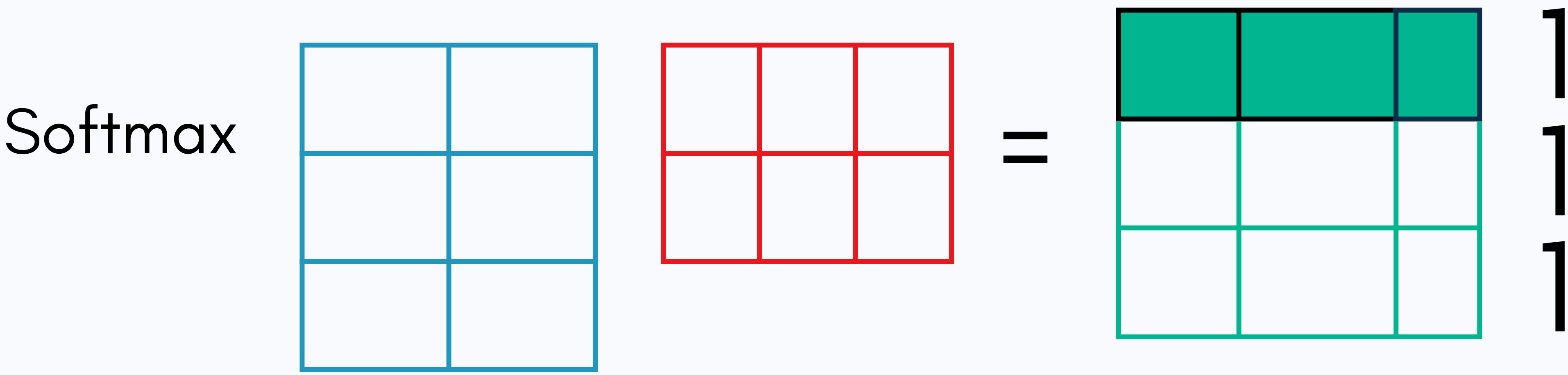
Value

..	..	Pizza
..	..	is
0.18	0.81	awesome
..	..	<EOS>

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



unscaled dot product and scale each dot product similarity by sqrt(2) -- encoded word dimnesion size



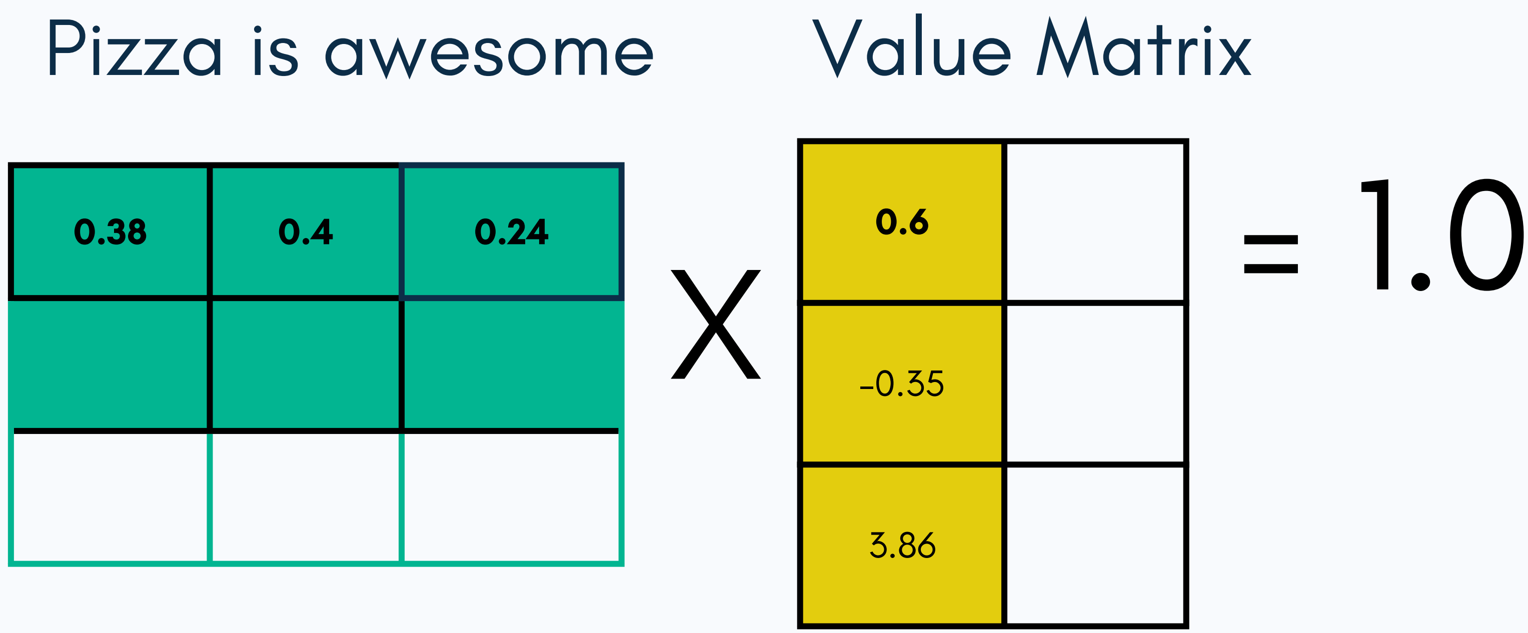
	Pizza	is	awesome
Pizza	0.38	0.4	0.9
is			
awesome			

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

Pizza is awesome   Value Matrix

$$\begin{bmatrix} 0.38 & 0.4 & 0.24 \\ & & \\ & & \end{bmatrix} \times \begin{bmatrix} 0.6 & \\ -0.35 & \\ 3.86 & \end{bmatrix} = 1.0$$

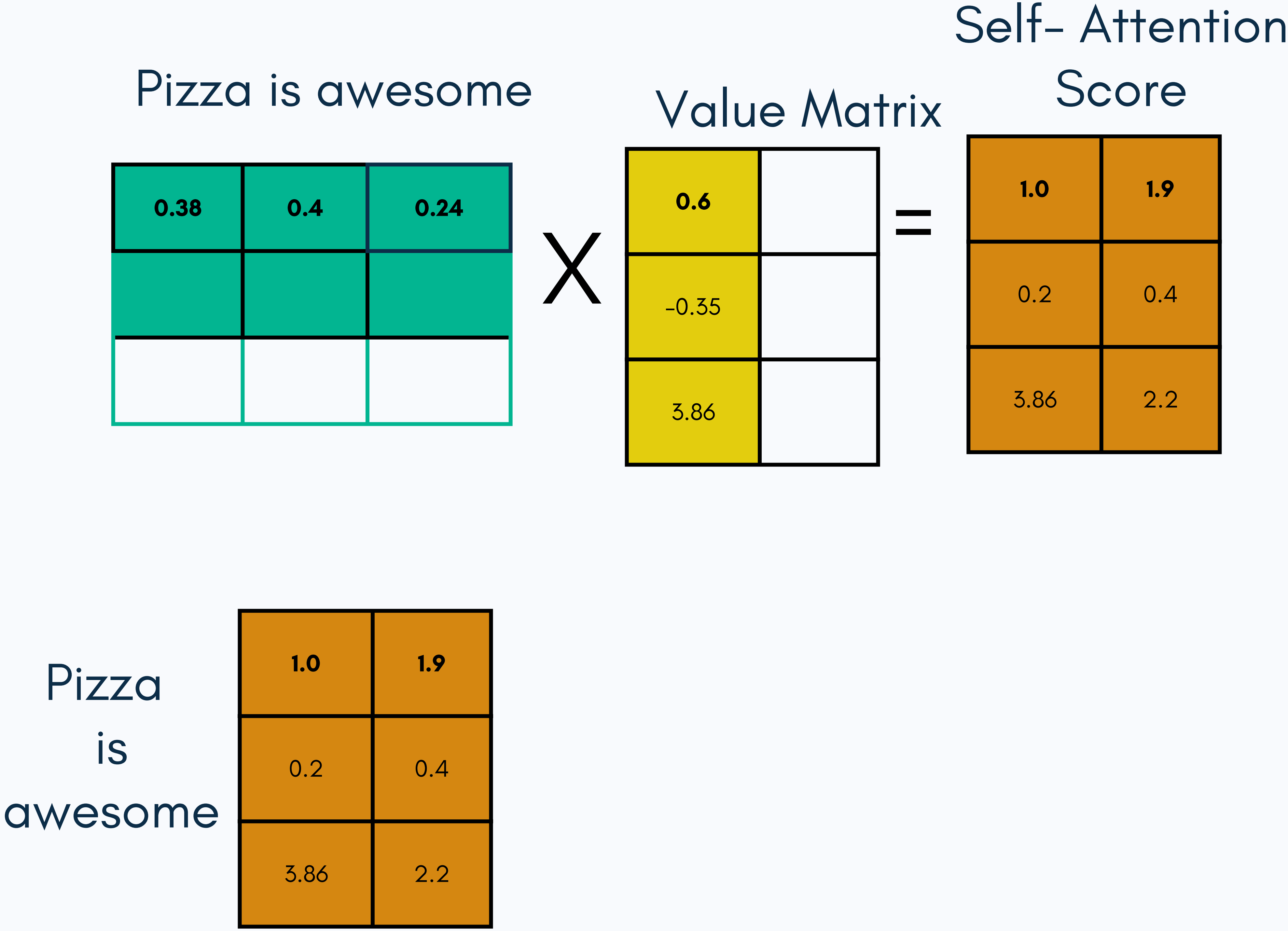
In other words the percentages that comes out of the softmax .... tells us how much influence each word should have on the final encoding for a given word



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


Pizza is awesome
0.38
0.4
0.24
X
Value Matrix
0.6
-0.35
3.86
=
Self- Attention Score
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):
```

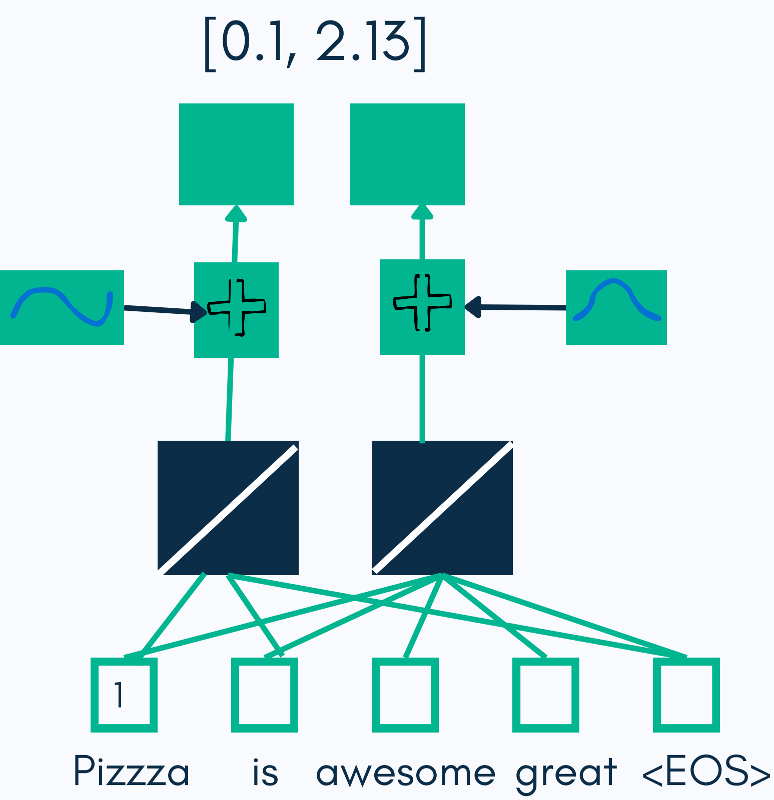
class selfAttention inherits from nn.Module

Its the base class of all neural netwrok modules

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

__init()__ method

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



Pizza
is
awesome
<EOS>

0.1	**2.13**
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

convinience 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)
```

Encoded Value

Pizza	**0.1**	**2.13**
is	0.11	2.13
awesome	0.21	3.13
<EOS>	0.8	0.9

X

Query Weights

-0.54	**0.17**
0.59	0.65

=

Query

0.71	**-0.05**
1.11	0.79
1.11	-2.15
0.8	0.9

nn.Linear creates the weight matrix

```
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)
```

Encoded Value

Pizza	**0.1**	**2.13**
is	0.11	2.13
awesome	0.21	3.13
<EOS>	0.8	0.9

X

Query Weights T

-0.54	**0.17**
0.59	0.65

=

Query

0.71	**-0.05**
1.11	0.79
1.11	-2.15
0.8	0.9

in_feature number of rows and out_feature number of columns

```
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)
```

Encoded Value

Pizza	**0.1**	**2.13**
is	0.11	2.13
awesome	0.21	3.13
<EOS>	0.8	0.9

X

Key Weights T

-0.54	**0.17**
0.59	0.65

in_feature number of rows and out_feature number of columns

=

Key

0.71	**-0.05**
1.11	0.79
1.11	-2.15
0.8	0.9

```
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)
```

Encoded Value

Pizza	**0.1**	**2.13**
is	0.11	2.13
awesome	0.21	3.13
<EOS>	0.8	0.9

X

Value Weights T

-0.54	**0.17**
0.59	0.65

in_feature number of rows and out_feature number of columns

=

Value

0.71	**-0.05**
1.11	0.79
1.11	-2.15
0.8	0.9

```
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
```

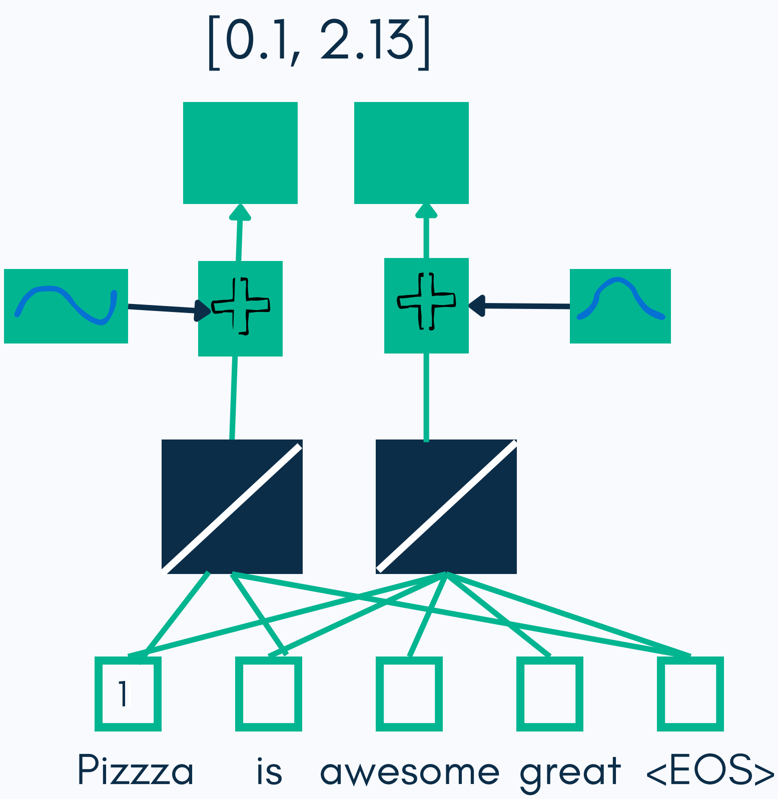
```
class SelfAttention(nn.Module):
    def __init__(self, d_model=2, row_dim, column_dim )
        super().__init()

        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



```
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)
```

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

Similiaries 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)
```

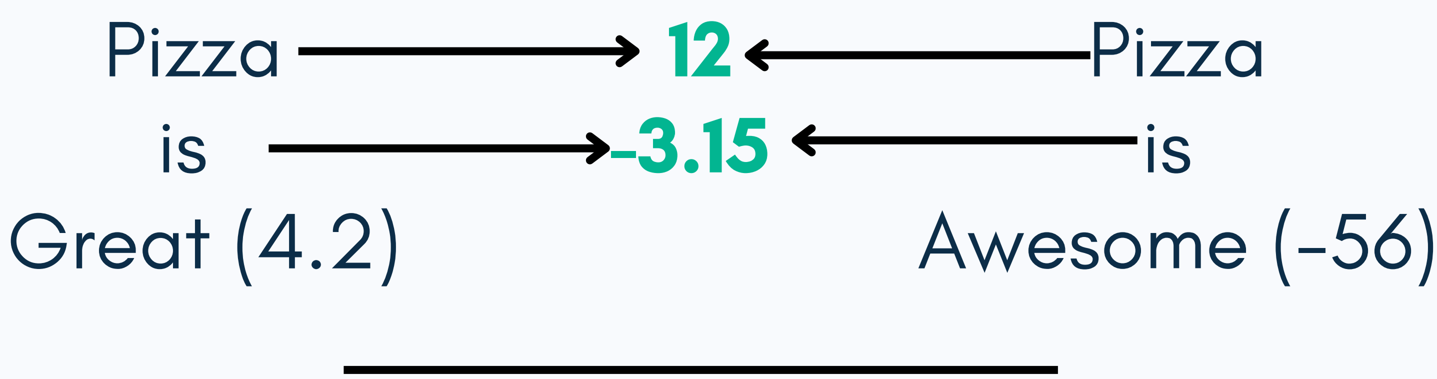
$$Attention\,(Q,K,V) = Softmax\,(\frac{QK^T}{\sqrt{d_k}})V$$

```
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 similaries
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)
```

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



Also the same word can be used in different contexts or made plural of or used in some other way, it might be nice to assign each word more than one number, so that the NN can easily adjust to the different context.

Lets think for a bit and assign some random numbers to sentences .

The word great can be used in positive way and negative way.

Pizza is **Great**

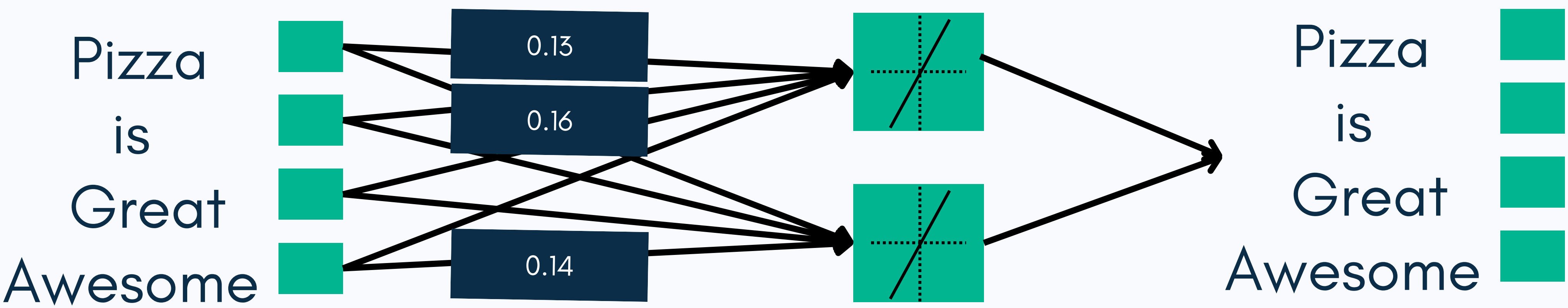
My phone broke, **Great**

So, it would be good to keep track of positive context and of negative context ...

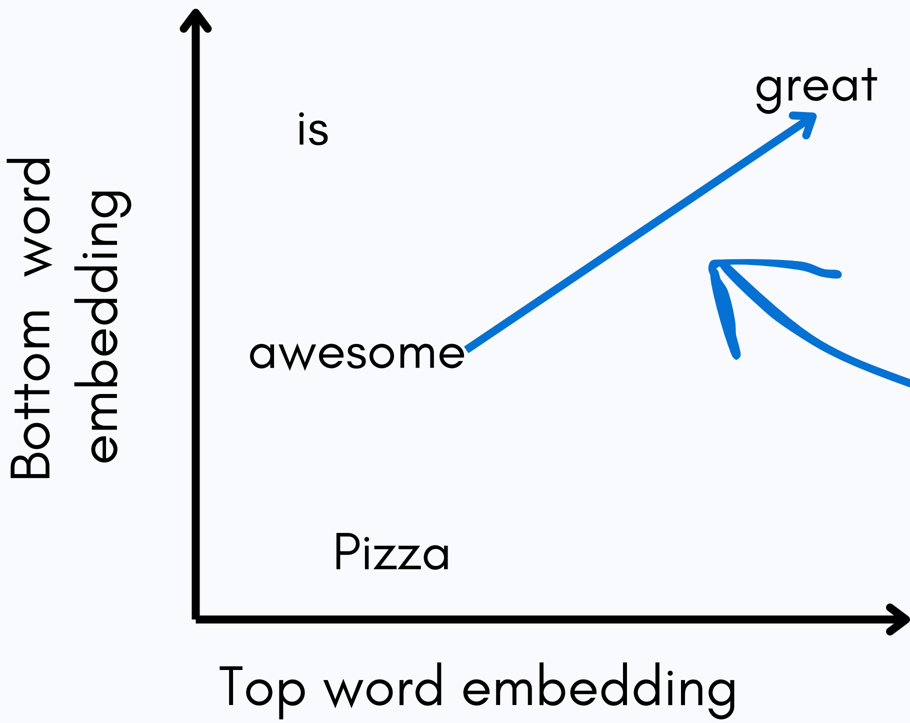
So, lets build a simple **word embedding** network for these 2 phrases

Pizza is Great	Pizza is Awesome

1. Create input for a simple NN.
2. Create output
3. Connect all the inputs to atleast one activation function.
4. Add weights , numbers with which the inputs are multiplied
5. Finally connect activation to output



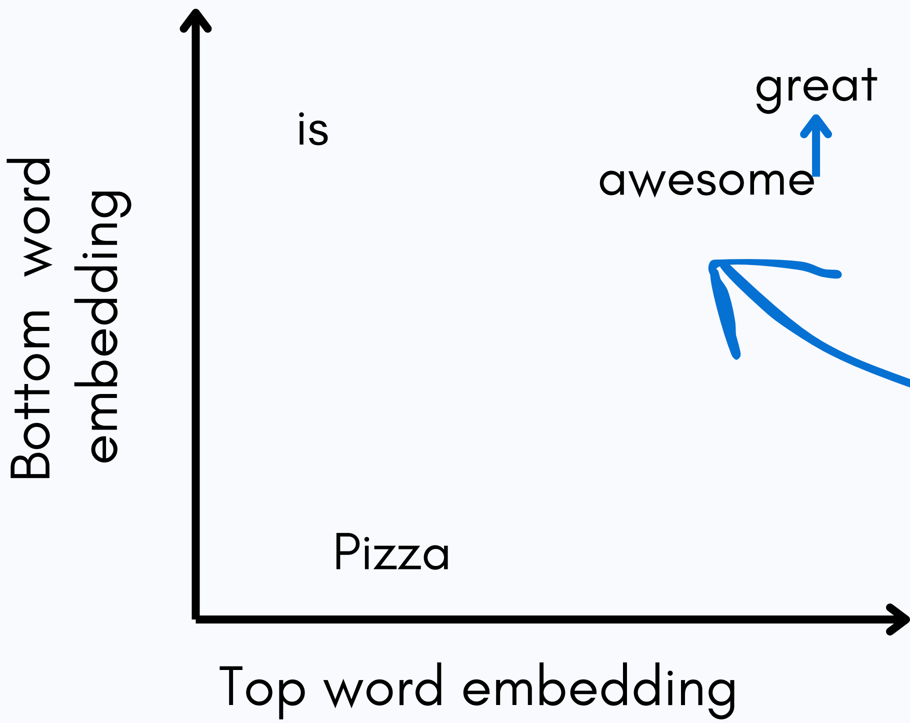
These weights are word embedding values and are randomly assigned ... but the plan is to change them using the training data



Pizza
is
great
awesome

-0.11	0.10
-0.12	0.46
0.38	0.42
-0.24	0.29

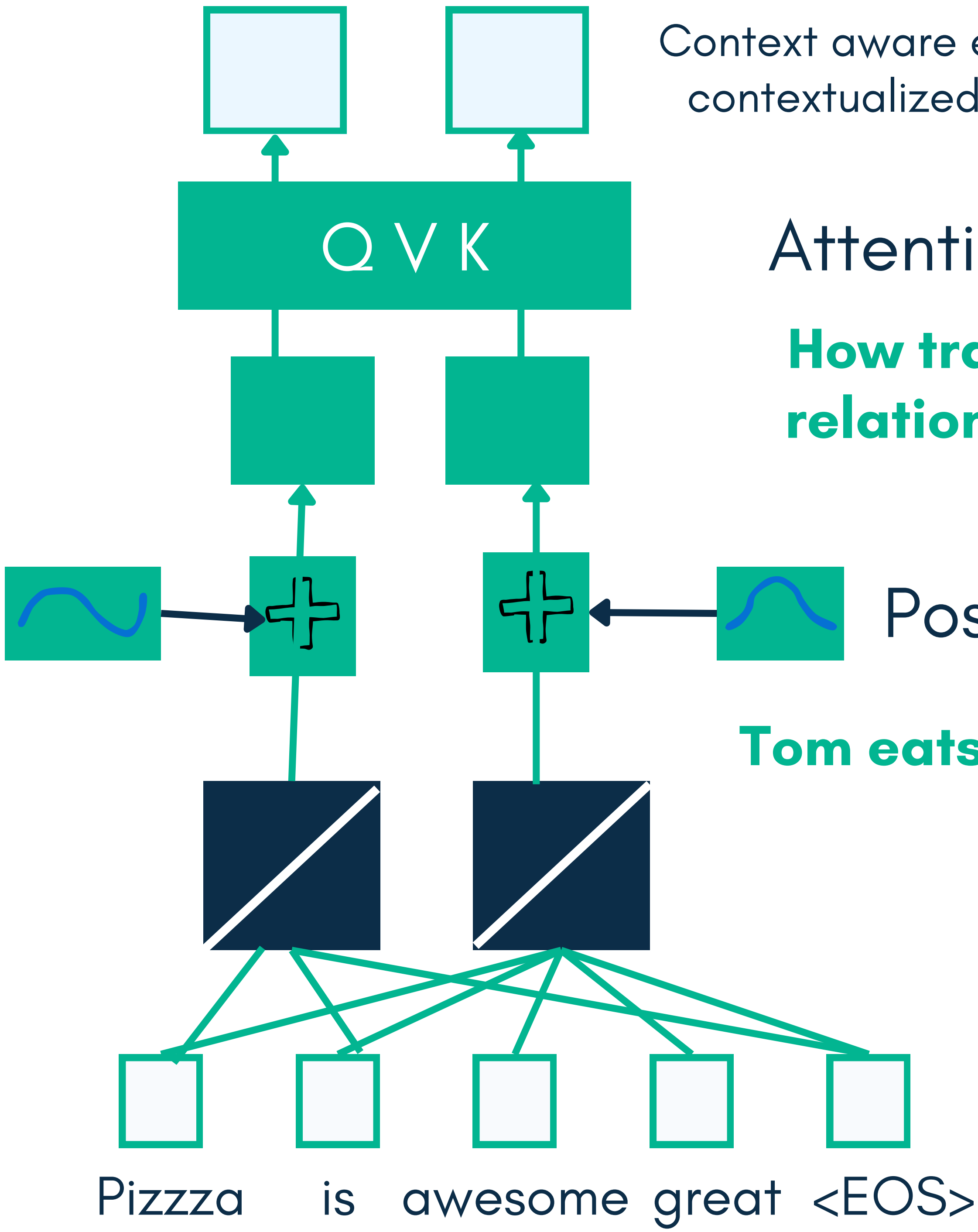
There is no similarity between awesome and great with the random values but with training the word embedding will become more similar.



Pizza
is
great
awesome

-0.11	0.10
-0.12	0.46
0.38	0.42
-0.24	0.29

After **training great and awesome** end up with similar **Word Embeddings**



Context aware embedding or
contextualized embedding

Attention

How transformers establish relations among words with attention?

Positional Embedding

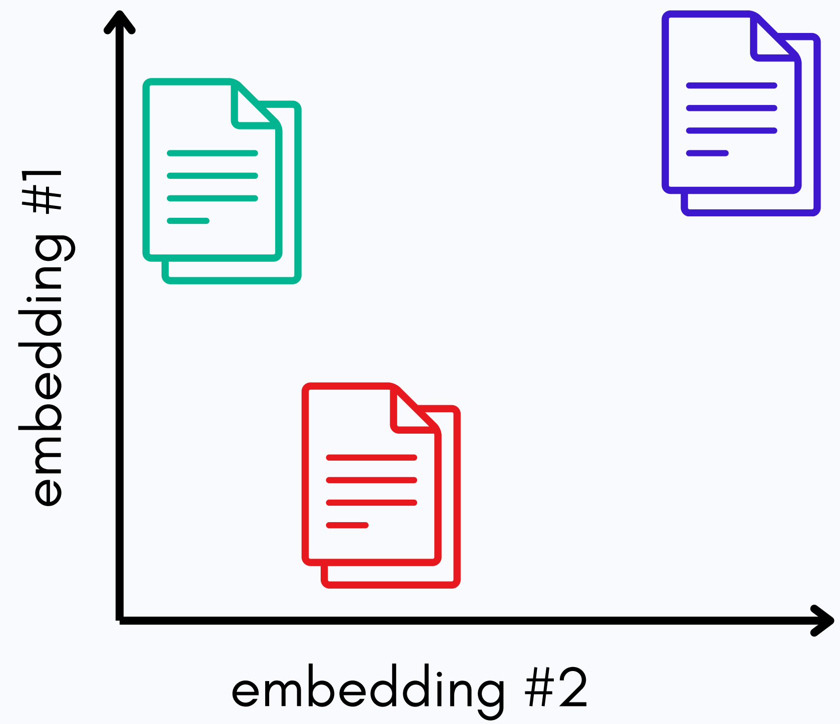
Tom eats Pizza   **Pizza eats Tom**

Word Embedding

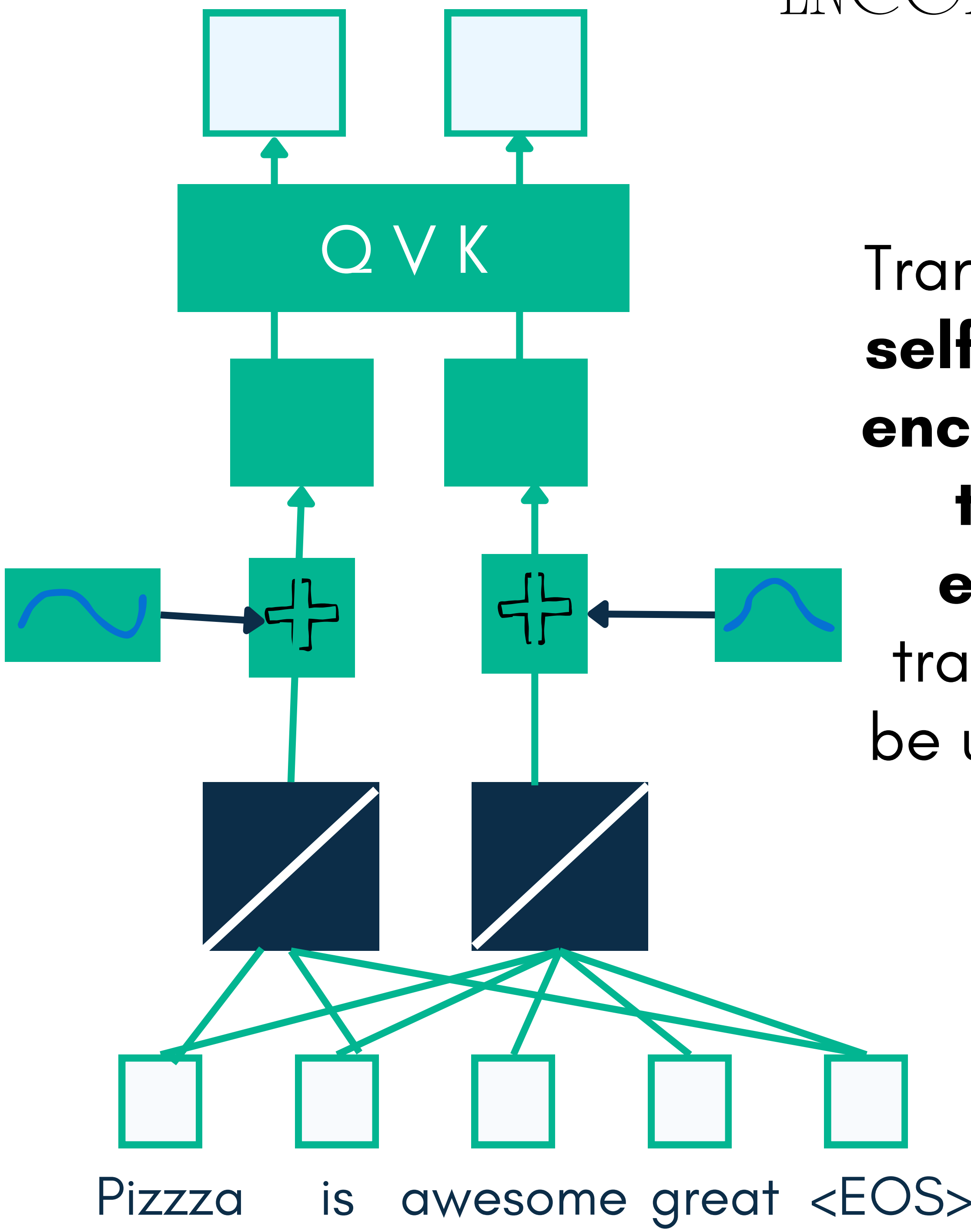
[0.1, 2.13, 3.14,5.0]

Word Embedding cluster similar words

Context Embedding cluster similar sentences and even cluster similar documents

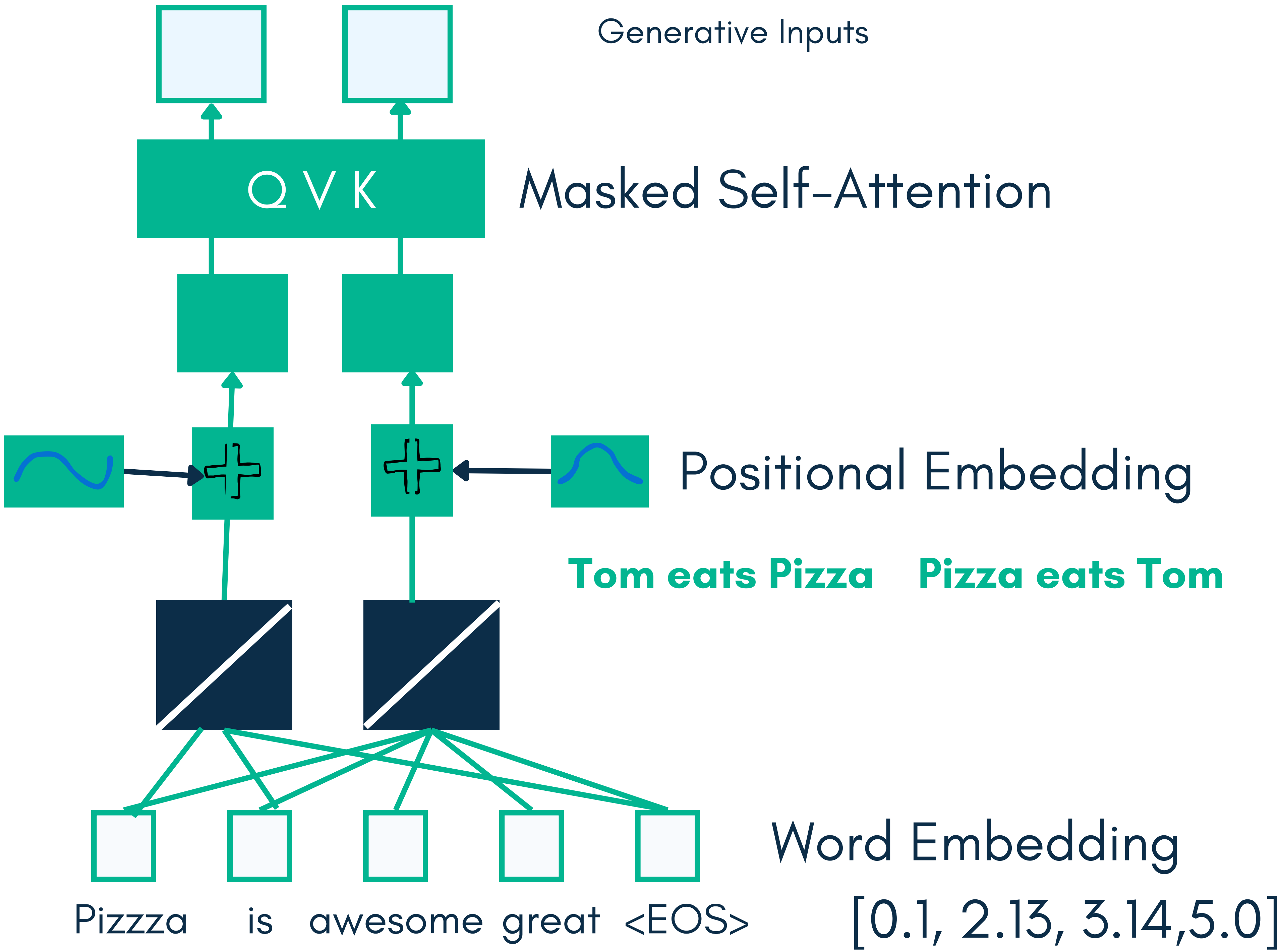


ENCODER ONLY TRANFORMER



Transformers that only use **self- attention** are called **encoder only attention... the context aware embedding** that the transformers create can be used in wide variety of setting

DECODER ONLY TRANFORMER



Self-attention looks at word before and after the word of interest
The **pizza** came out of the **oven** and **it** tasted good
The **pizza** came out of the **oven** and **it** tasted good
Mask-attention looks at word before the word of interest

What will happen to the word The?

Self-Attention

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

Mask Attention

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

What will happen to the word The?

Self-Attention

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

Mask Attention

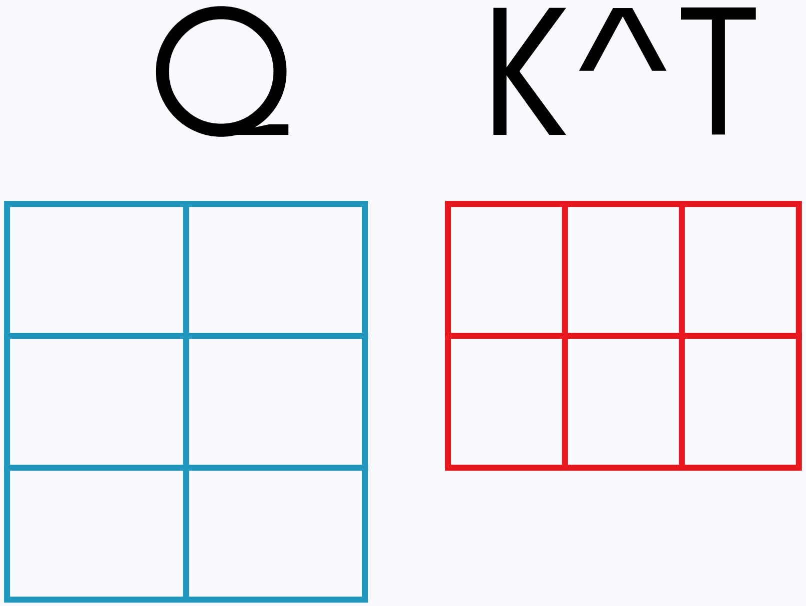
The **pizza** came out of the **oven** and **it** tasted good

The decoder only transformers will do a good job at generating prompt responses .. because it can not look ahead

This is why chatGPT which is decoder only transformer is called Generative model .. because its specifically trained to generate the text that comes next ....

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

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



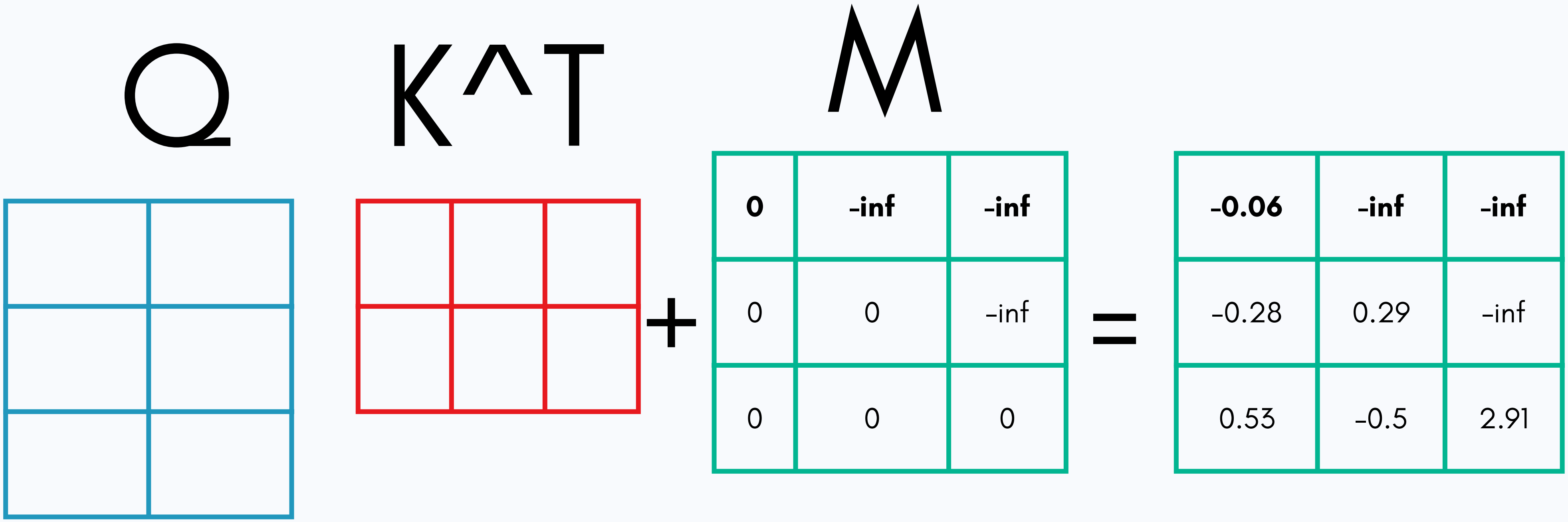
+

M

0	**-inf**	**-inf**
0	0	-inf
0	0	0

unscaled dot product and scale each dot product similarity by sqrt(2) -- encoded word dimnesion size

add zeros to values we want to include and -inf to mask out



unscaled dot product and scale each dot product similarity by sqrt(2) -- encoded word dimnesion size

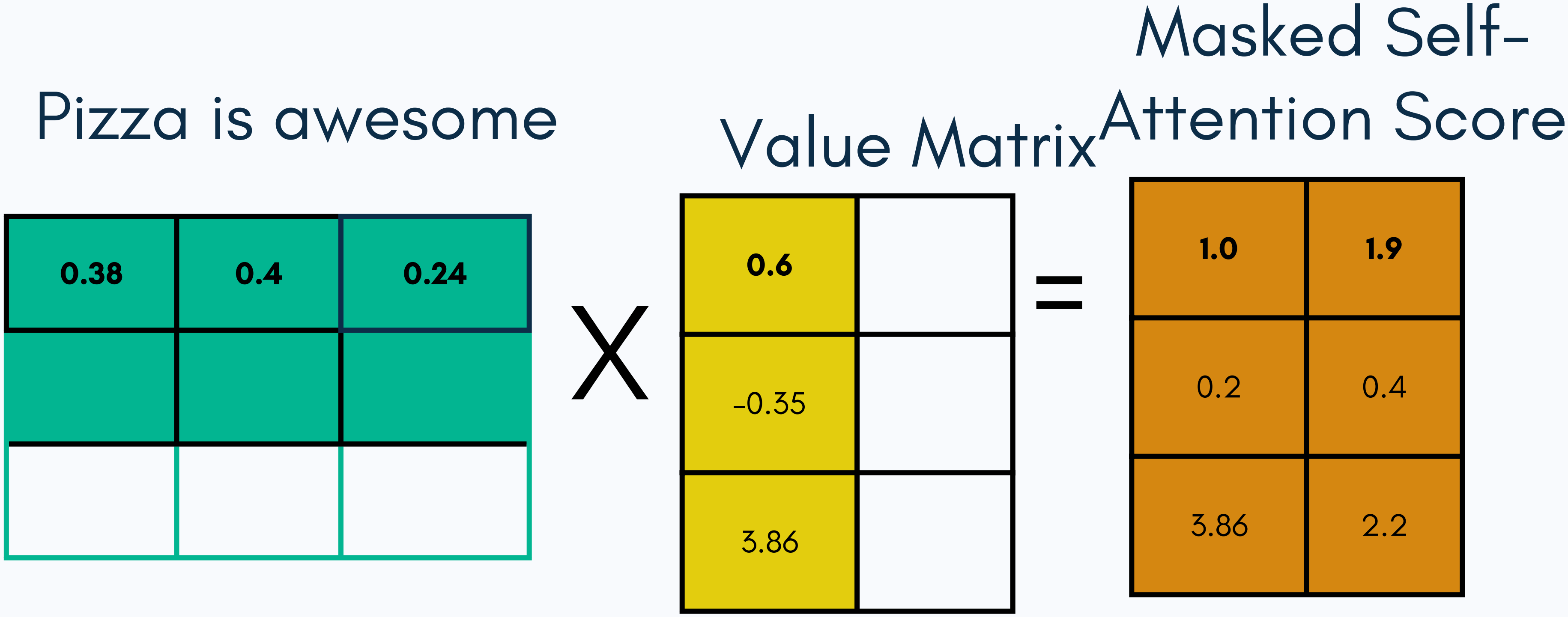
add zeros to values we want to include and -inf to mask out

	Pizza	is	awesome
Softmax	**-0.06**	**-inf**	**-inf**
Softmax	-0.28	0.29	-inf
Softmax	0.53	-0.5	2.91

=

Pizza	is	awesome
1	**0**	**0**
0.36	0.24	0
0.53	0.03	0.91

Pizza has 100% similiarity to Pizza and 0% to others
is has 0% similiarity to awesome
awesome 91% similirity to awesome 3% similiarity to is
etc...



The masked self-attention value doesn't include anything that came after it Pizza does not include is and awesome

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

__init()__ method

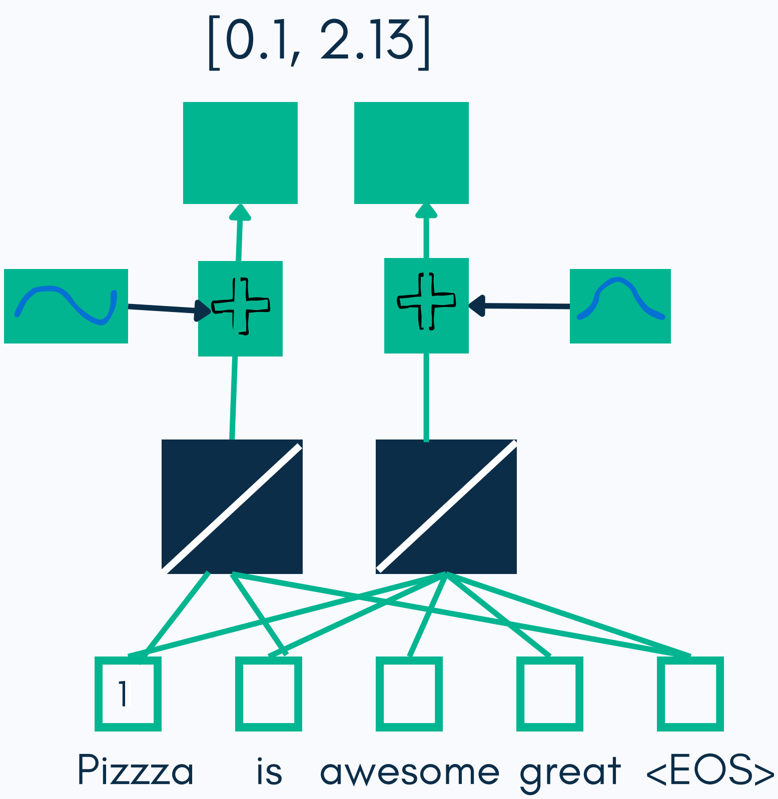
size of word embedding

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

```
class MaskedAttention(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



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

Parent class init method

```
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
```

Query weights, Key Weights and Value Weights

```
class MaskedAttention(nn.Module):
def forward(self, token_encoding, mask=None ):
      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)
```

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

Similiaries between all possible combinations of Queries and keys

```
class MaskedAttention(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)
     if Mask is not None:
                    scaled_sims= scaled_sims.Masked_fill(mask=mask, value=-1e9)
```

True corrosponds to attention values that we want to mask out

```
tensor([[False, True,True],
        [False,False,True],
        [False,False,False]])
```

```
tensor([[0, -1e9,-1e9],[0,0,-1e9],[0,0,0]])
```

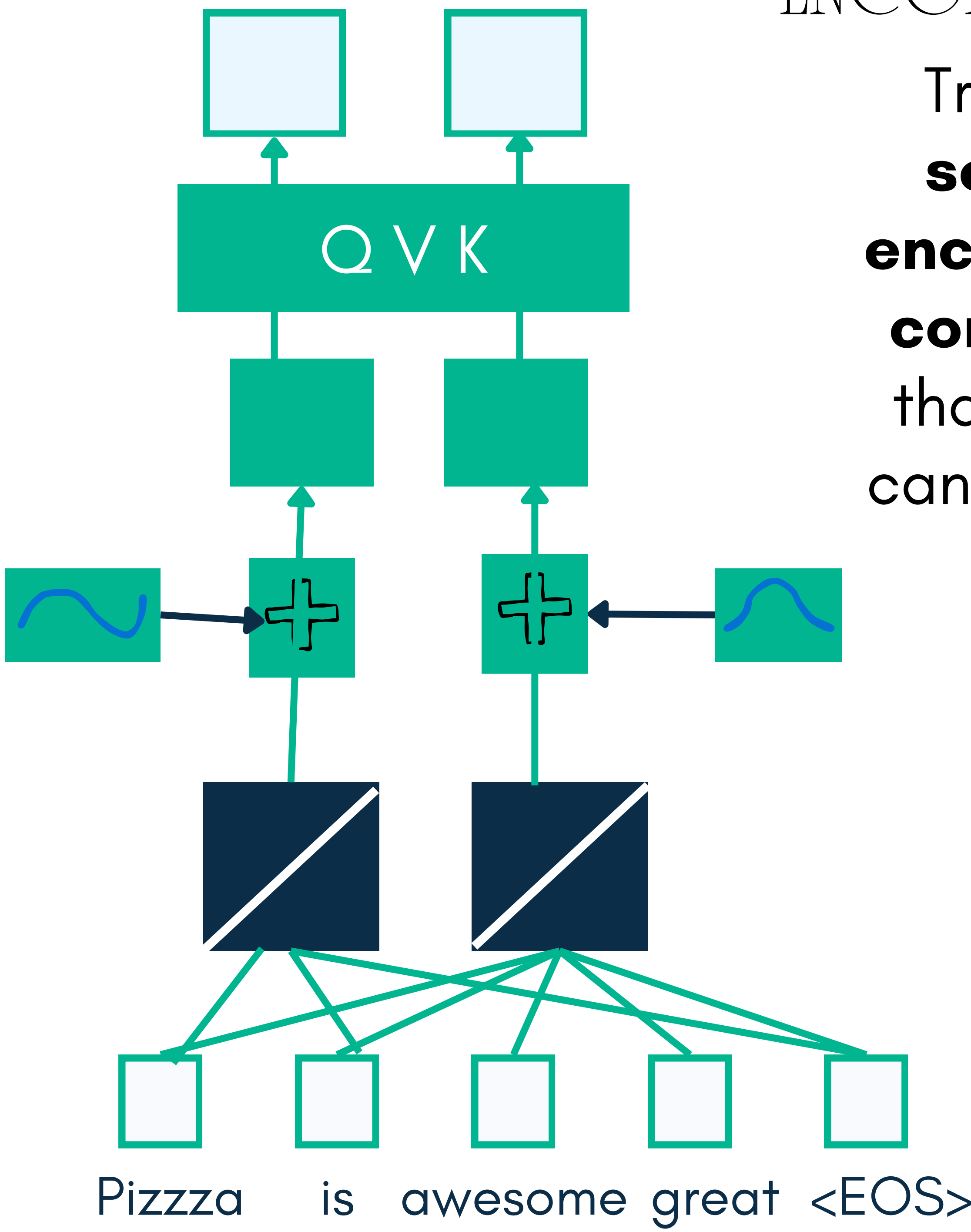
$$Attention(K,Q,V) = Softmax\left(\frac{QK^T}{\sqrt{d_K}} + M\right)V$$

```
class MaskedAttention(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)
    if Mask is not None:
                    scaled_sims= scaled_sims.Masked_fill(mask=mask, value=-1e9)
            attention_percentage = F.softmax(scaled_sim, dim= self.col_dim)
                attention_scores = torch.matmul(attention_percentage, v)
```

We run the scaled similarities through a **softmax()** to determine the percentage of influence that each token should have on the other.

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

ENCODER ONLY TRANFORMER

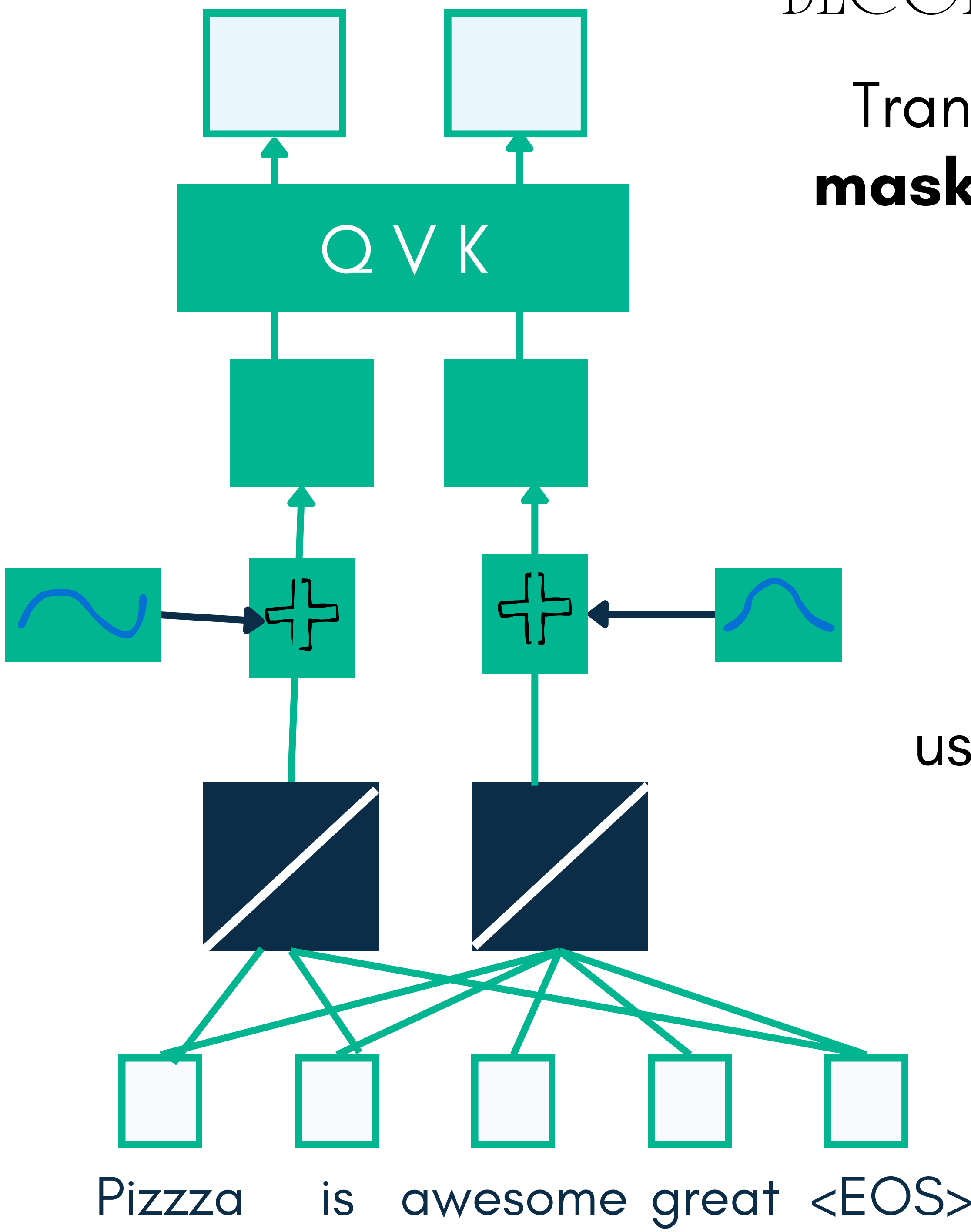


Transformers that only use **self- attention** are called **encoder only attention... the context aware embedding** that the transformers create can be used in wide variety of setting

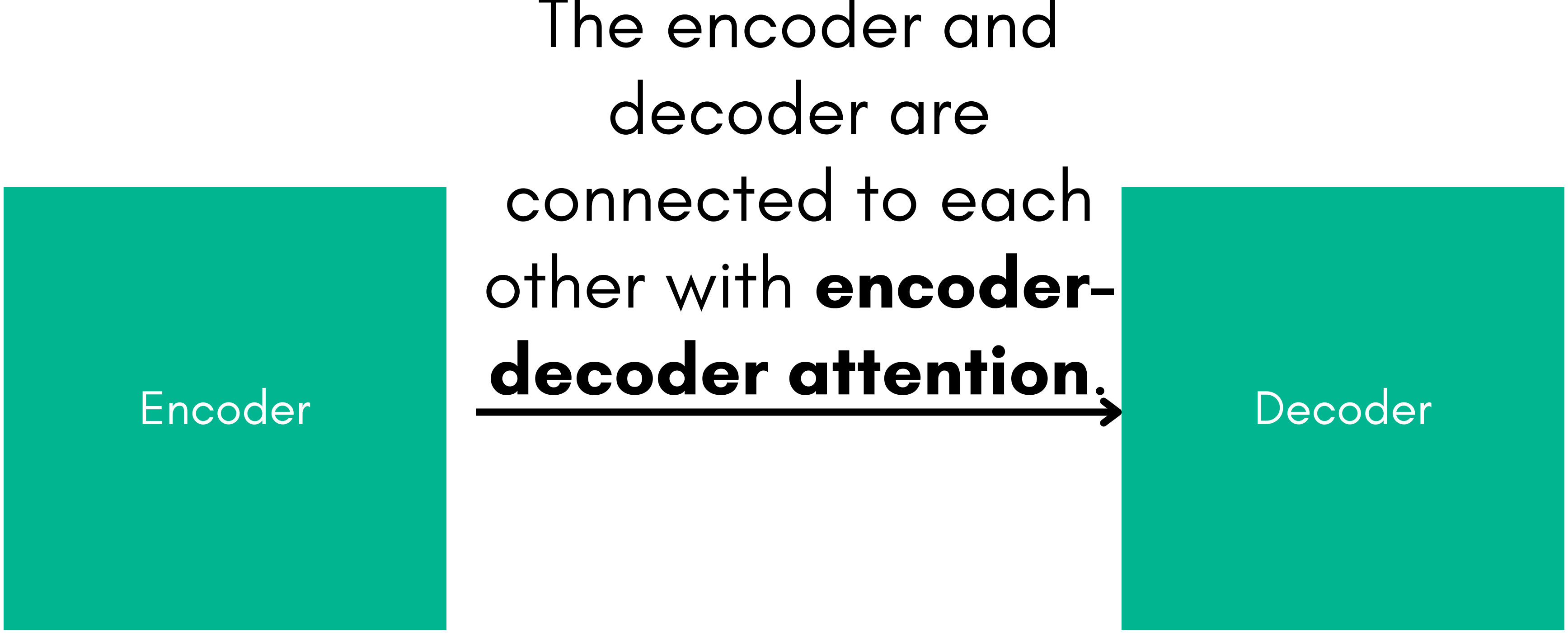
usecase: Input to classification model

DECODER ONLY TRANFORMER

Transformers that only use **masked self- attention** and are used for



usecase: generative texts



The **encoder-decoder attention** uses the output from **encoder** to calculate the **keys and values ...**
Queries are calculated from the output of masked self-attention generated by the decoder

This model is used for language translation.

The encoder-decoder attention is called cross-attention.

Where are these encoder-decoder models used in modern day?

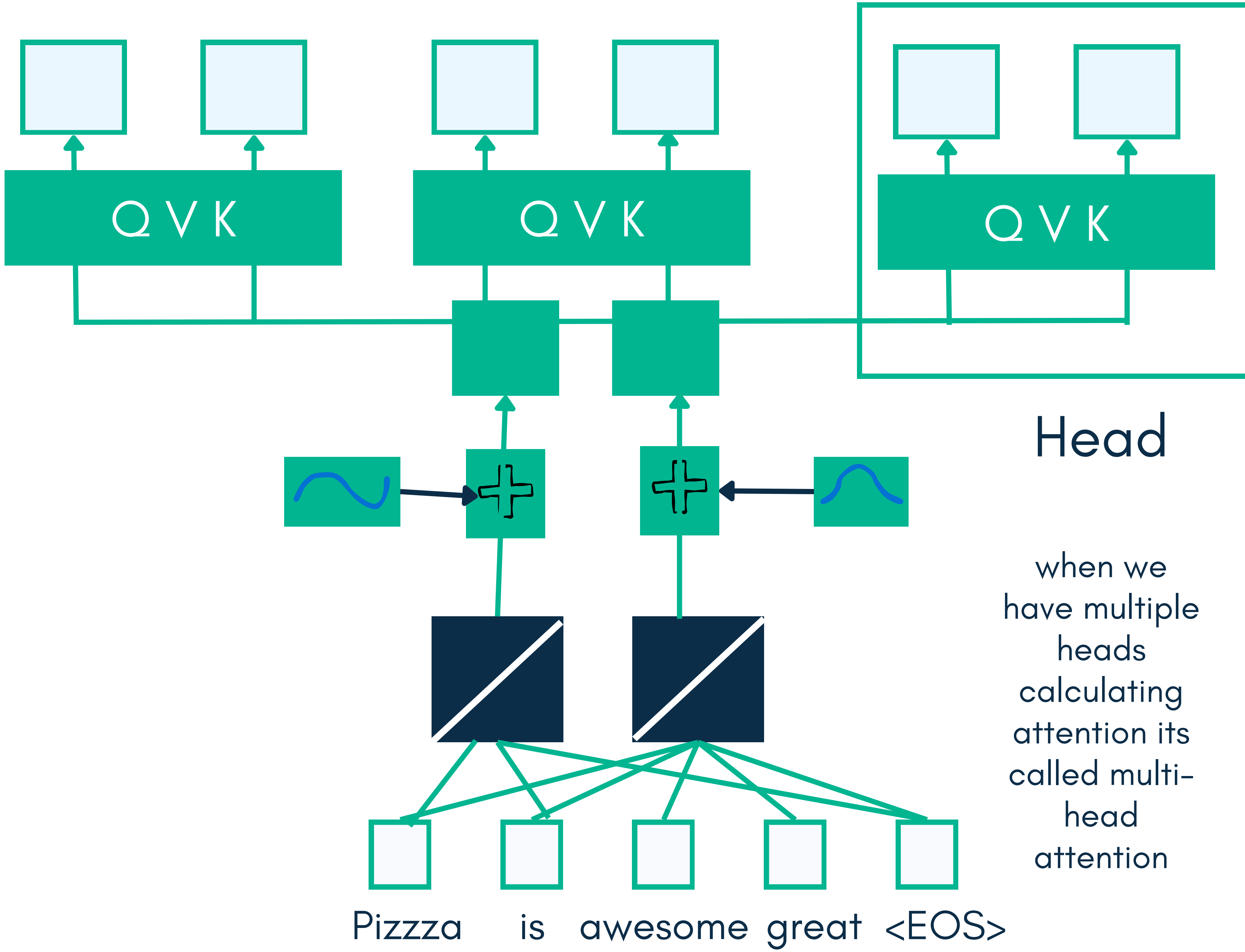
Multi-modal models we might have an encoder trainned on images or sound and use decoder to find its caption.

Useful repo: https://github.com/eonu/transformers-from-scratch(Will discuss later)

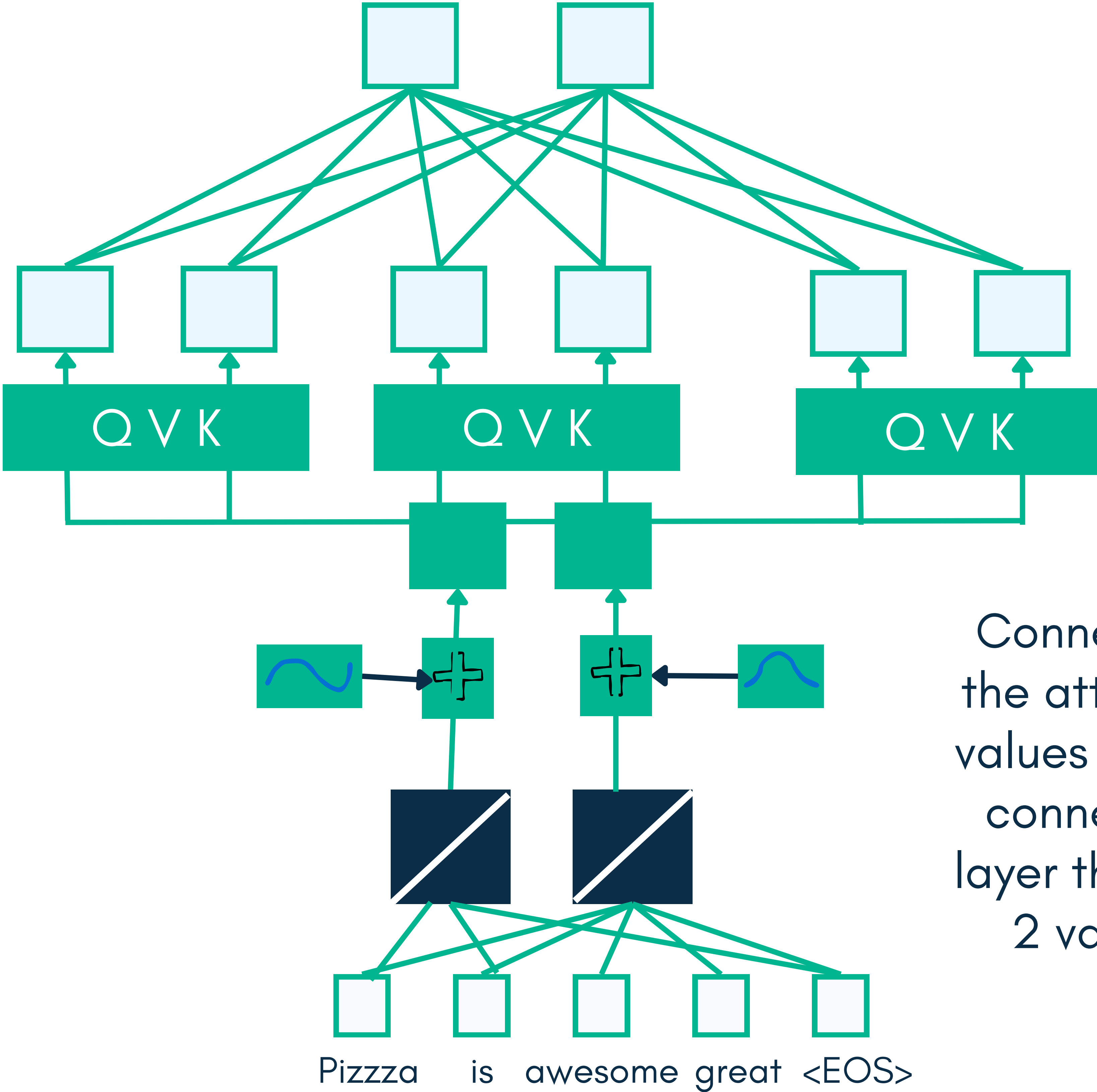
So far we have seen Attention helps establish how each word in the input is related to other.

However, in order to correctly establish how words arerelated in more complicated sentences and paragraphs ... we can apply Attention to encoded value multiple times simultaneouly.

Each attention unit is called a **head** and has own set of weights



Q V K
Q V K
Q V K
Head
when we have multiple heads calculating attention its called multi-head attention
Pizzza
is
awesome
great
<EOS>



Connect all the attention values to fully connected layer that has 2 values

Another common way to reduce the number of outputs is to modify the shape of the value weight matrix

In this example value weight has 2 columns which give value 2 columns ... and as a result attention head has 2 outputs

Encoded Value

Pizza	**0.1**	**2.13**
is	0.11	2.13
awesome	0.21	3.13
<EOS>	0.8	0.9

X

Value Weights T

-0.54	**0.17**
0.59	0.65

=

Value

0.71	**-0.05**
1.11	0.79
1.11	-2.15
0.8	0.9

Another common way to reduce the number of outputs is to modify the shape of the value weight matrix

In this example value weight has 2 columns which give value 2 columns ... and as a result attention head has 2 outputs

Encoded Value

Pizza	**0.1**	**2.13**
is	0.11	2.13
awesome	0.21	3.13
<EOS>	0.8	0.9

X

Value Weights T

-0.54	**0.17**
0.59	0.65

=

Value

0.71	**-0.05**
1.11	0.79
1.11	-2.15
0.8	0.9

In this example value weight has 1 columns which give value 1 columns ... and as a result eaxch attention will only output 1 value

Encoded Value

Pizza	**0.1**	**2.13**
is	0.11	2.13
awesome	0.21	3.13
<EOS>	0.8	0.9

X

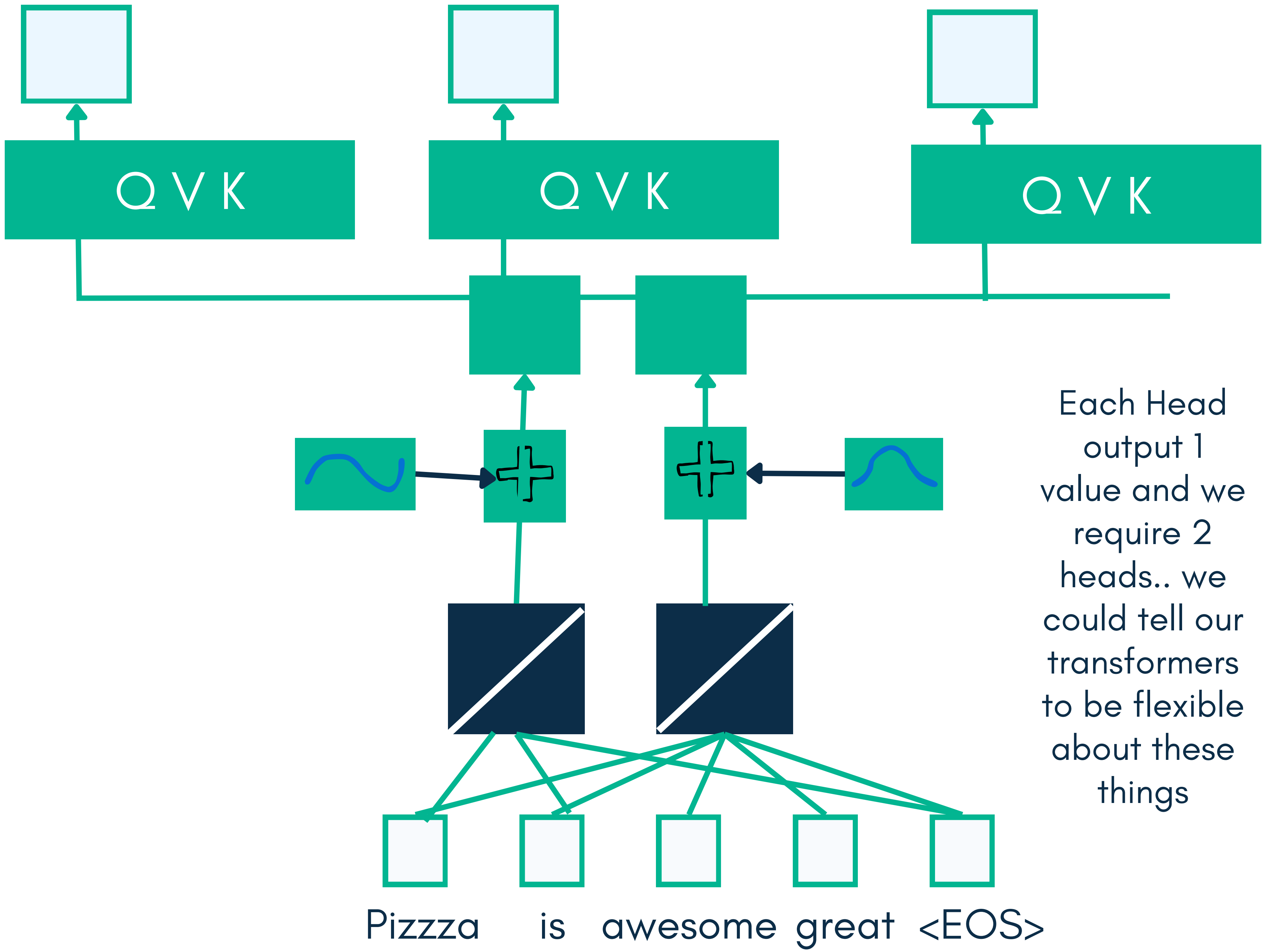
Value Weights T

-0.54
0.59

=

Value

0.71
1.11
1.11
0.8



Each Head output 1 value and we require 2 heads.. we could tell our transformers to be flexible about these things