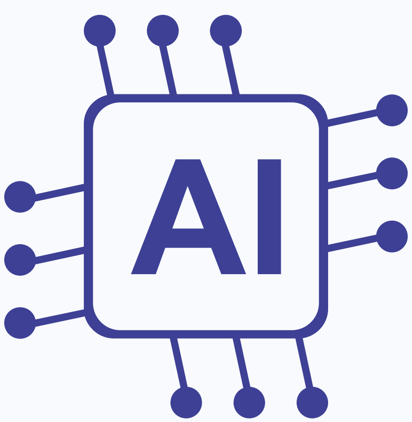
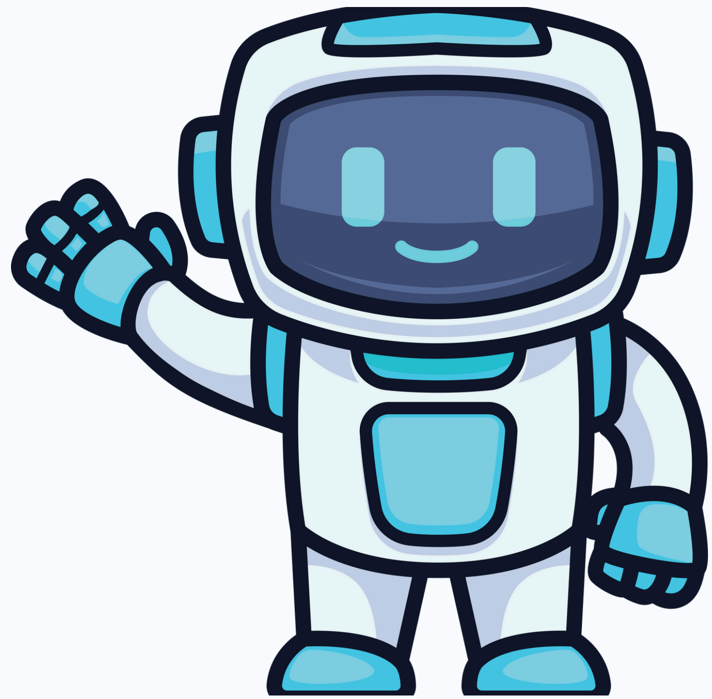
Transformer Chatbot

Unlocking
Your Potential,
Unleashing
Your Success





DataSet Identification

Cornell Movie-Dialogs Corpus

Each line in the movie_lines.txt file of the dataset follows this structure:

lineID +++$+++ characterID +++$+++ movieID +++$+++ character name +++$+++ text of the utterance

L1044 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ They do to!

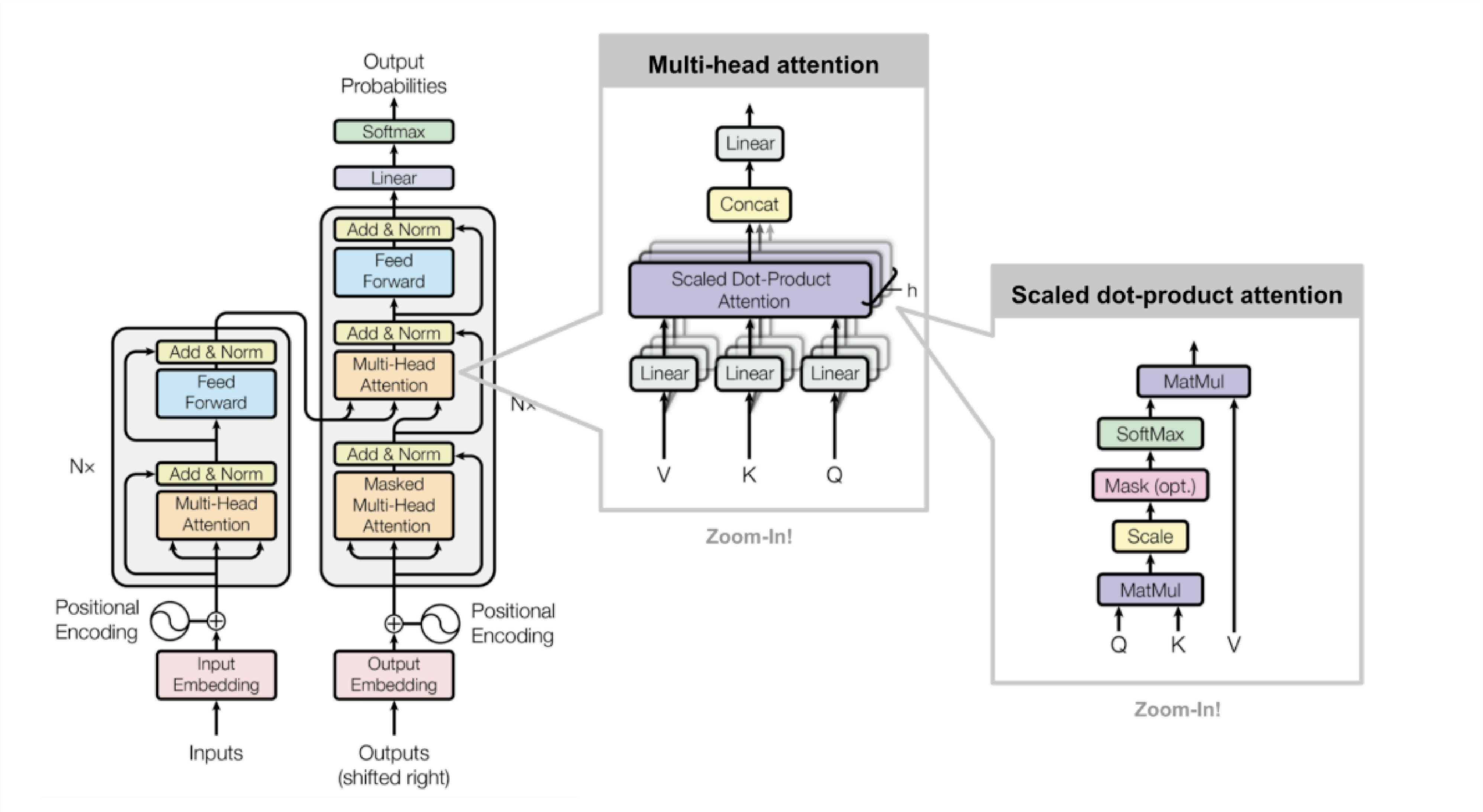
L1045 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ They do not!

movie_conversations.txt links line IDs from movie_lines.txt to reconstruct full conversations

u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L194', 'L195', 'L196', 'L197']

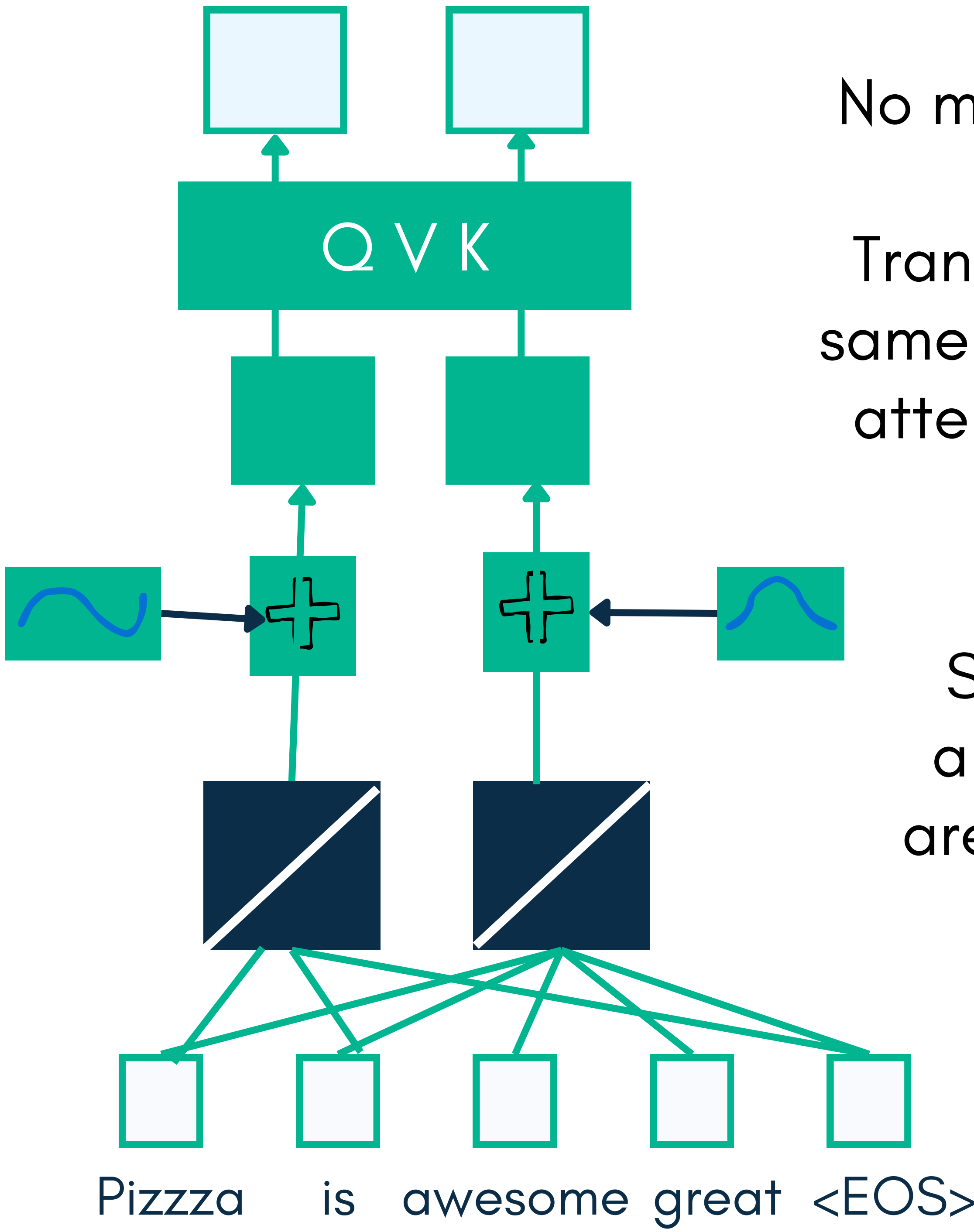
L194 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Can we make this quick? Roxanne Korrine and Andrew Barrett are having an incredibly horrendous public break-up on the quad. Again.
L195 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ Well, I thought we'd start with pronunciation, if that's okay with you.
L196 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Not the hacking and gagging and spitting part. Please.
L197 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ Okay... then how 'bout we try out some French cuisine. Saturday? Night?

Transformer Architecture



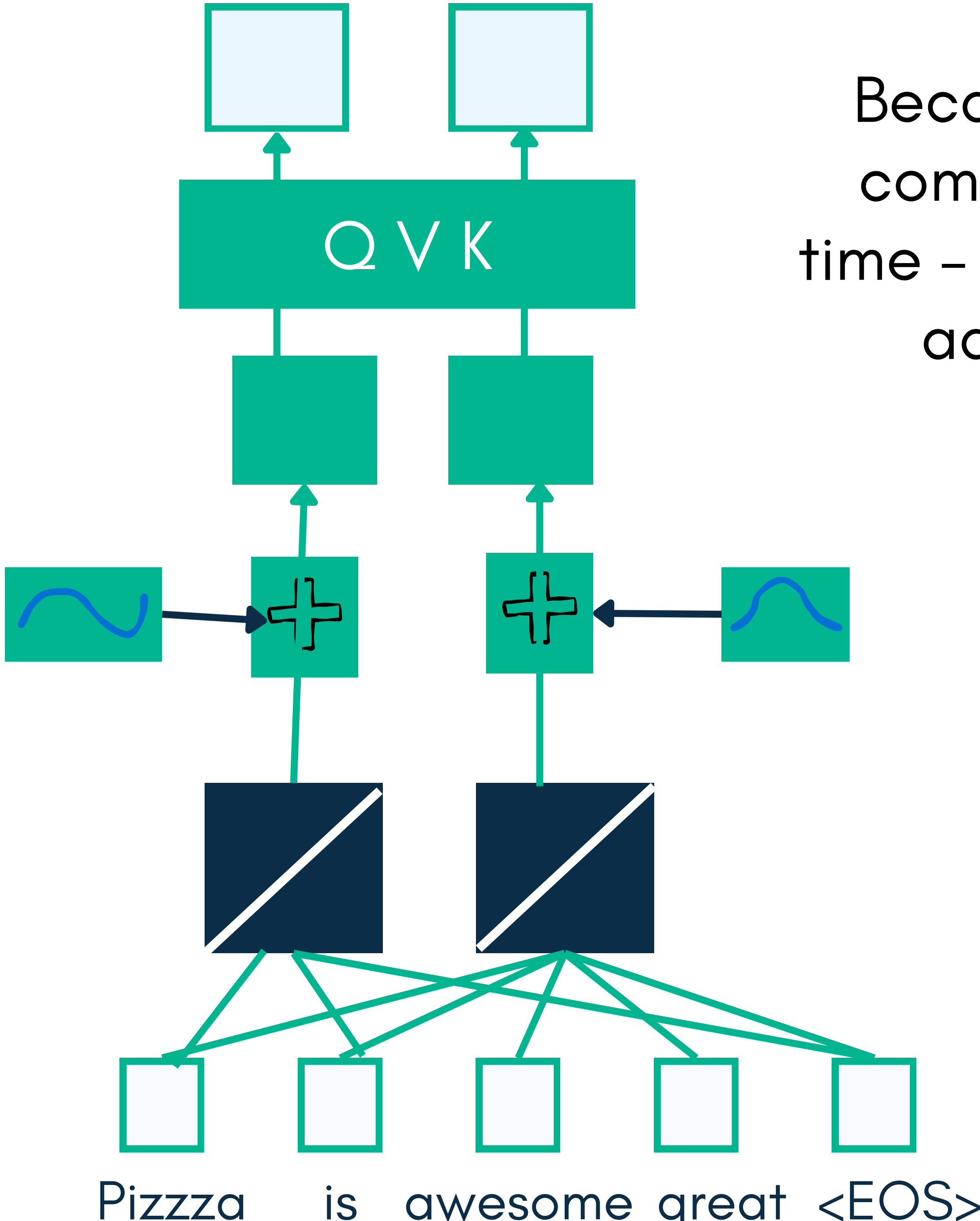
Transformers use **word-embeddings** to convert words into numbers...
Positional encdoding to keep track of words ....
Transformers have **self-attention** that associate word similiarity within input and output sequence....
Encoder-decoder attention to keep track of things between the input and the output phrases.... and are not lost in translation ....

Our example includes a chat conversation
Actor 1: Pizza is awesome great <EOS> --- encoder
Actor 2. That's really good<EOS> - decoder

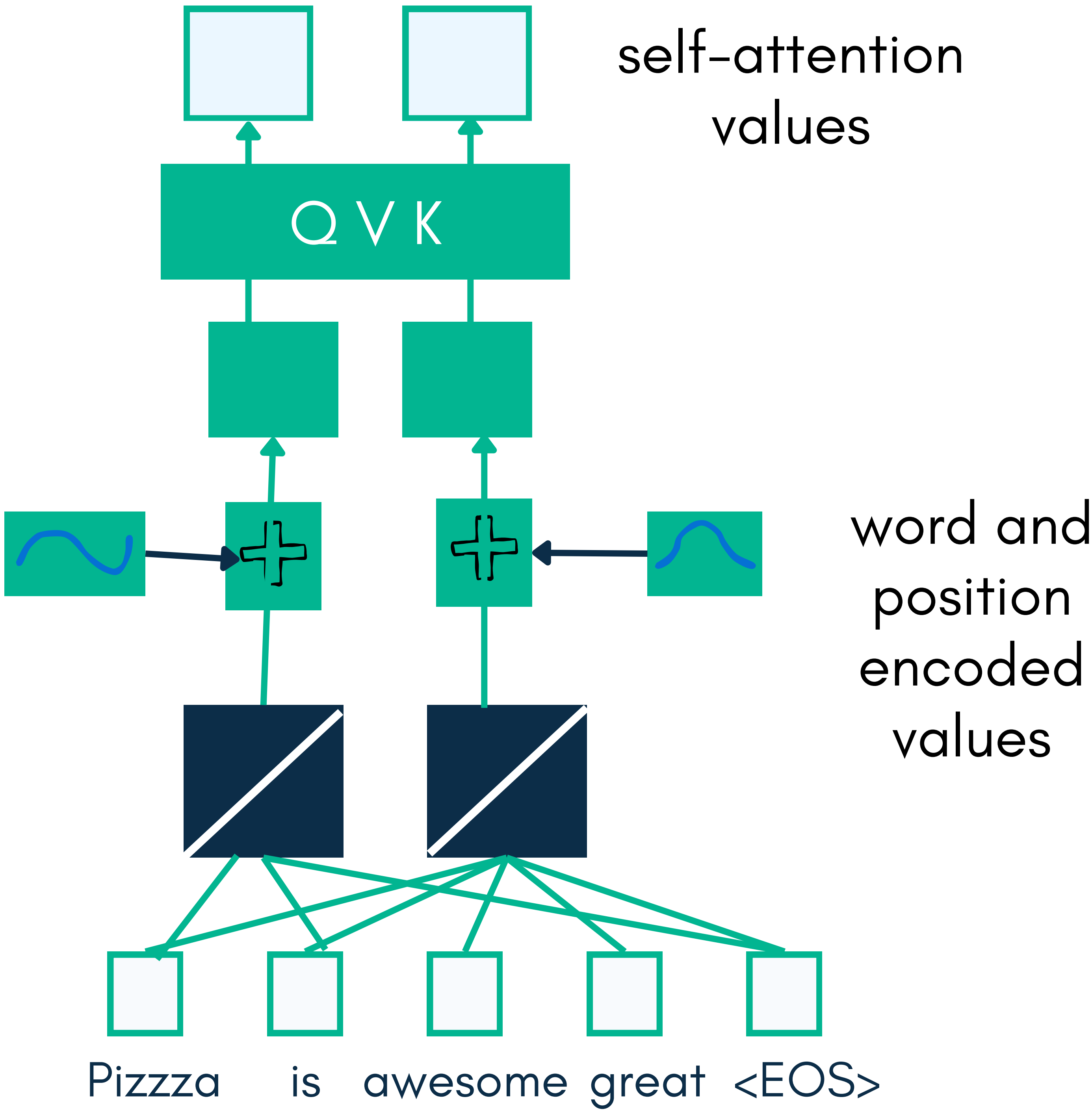


No matter how many words are input into the Transformers we reuse the same set of weights for self-attention queries,keys and values

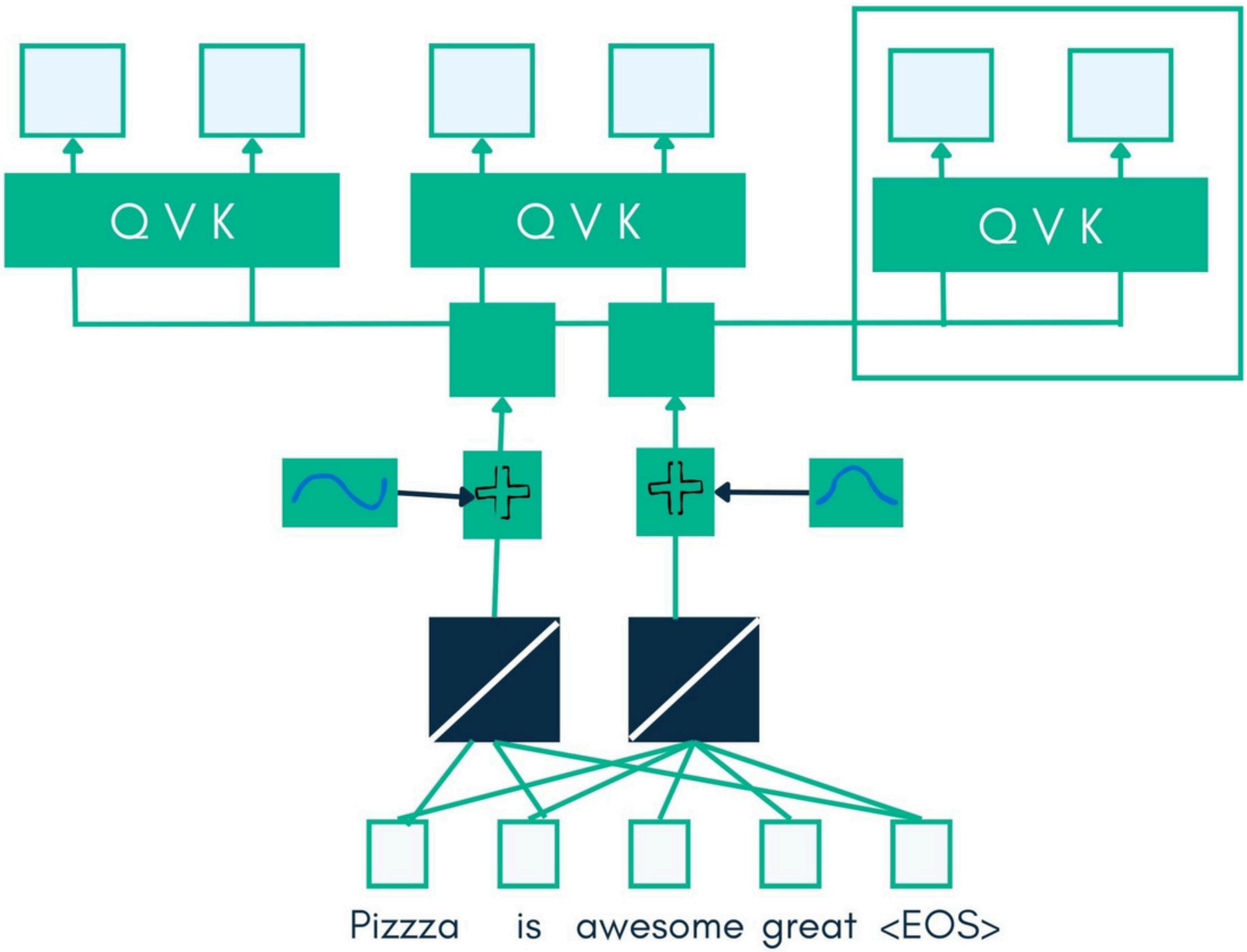
Self-attention queries,keys and values for all the words are calculated simultaneously

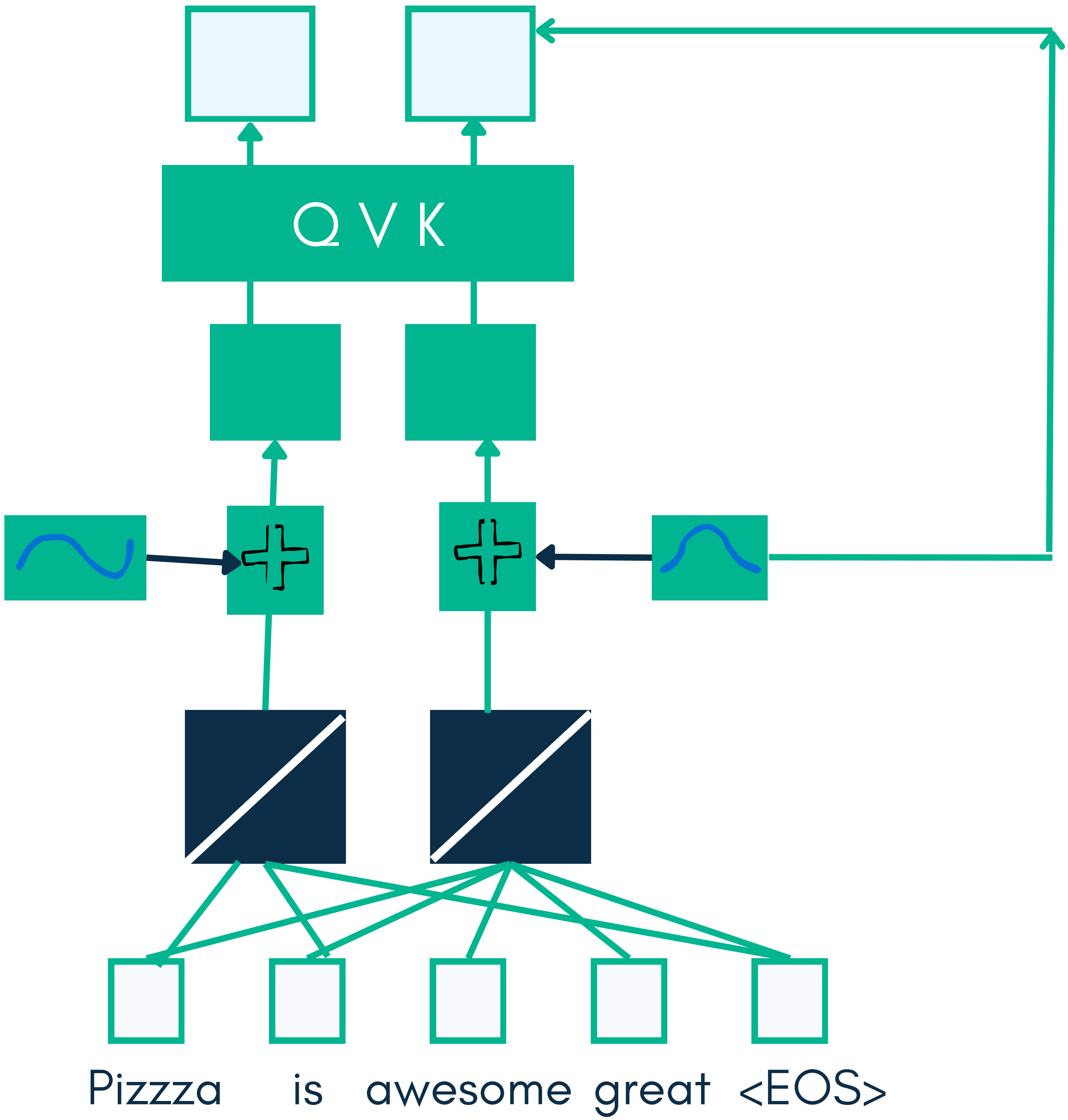


Because we can do all the computations at the same time – **Transformers** can take advantage of parallel computation


self-attention
values
Q V K
word and
position
encoded
values
Pizzza
is
awesome
great
<EOS>

Finally, if you have a number of sellf-attention units together you will get multi-head attentions

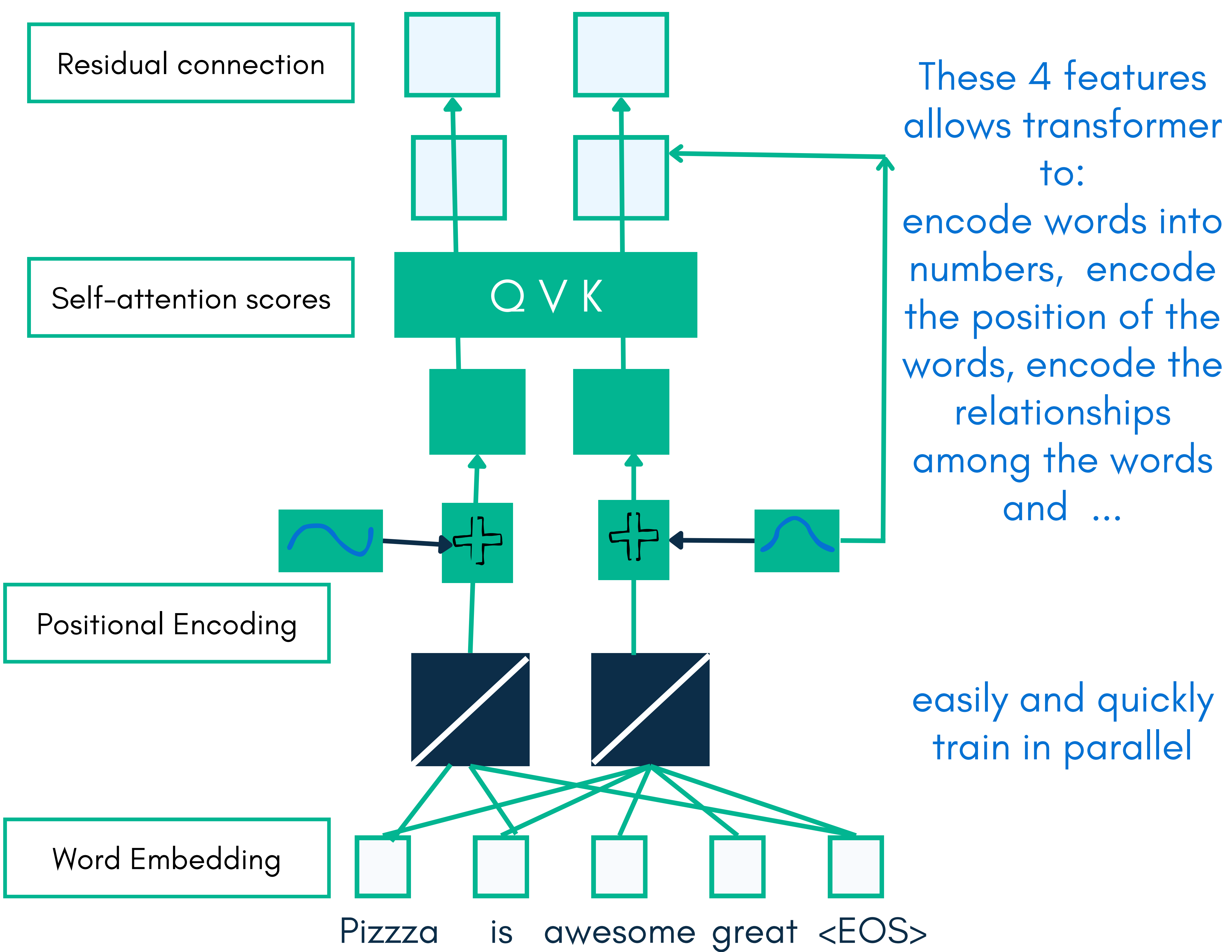


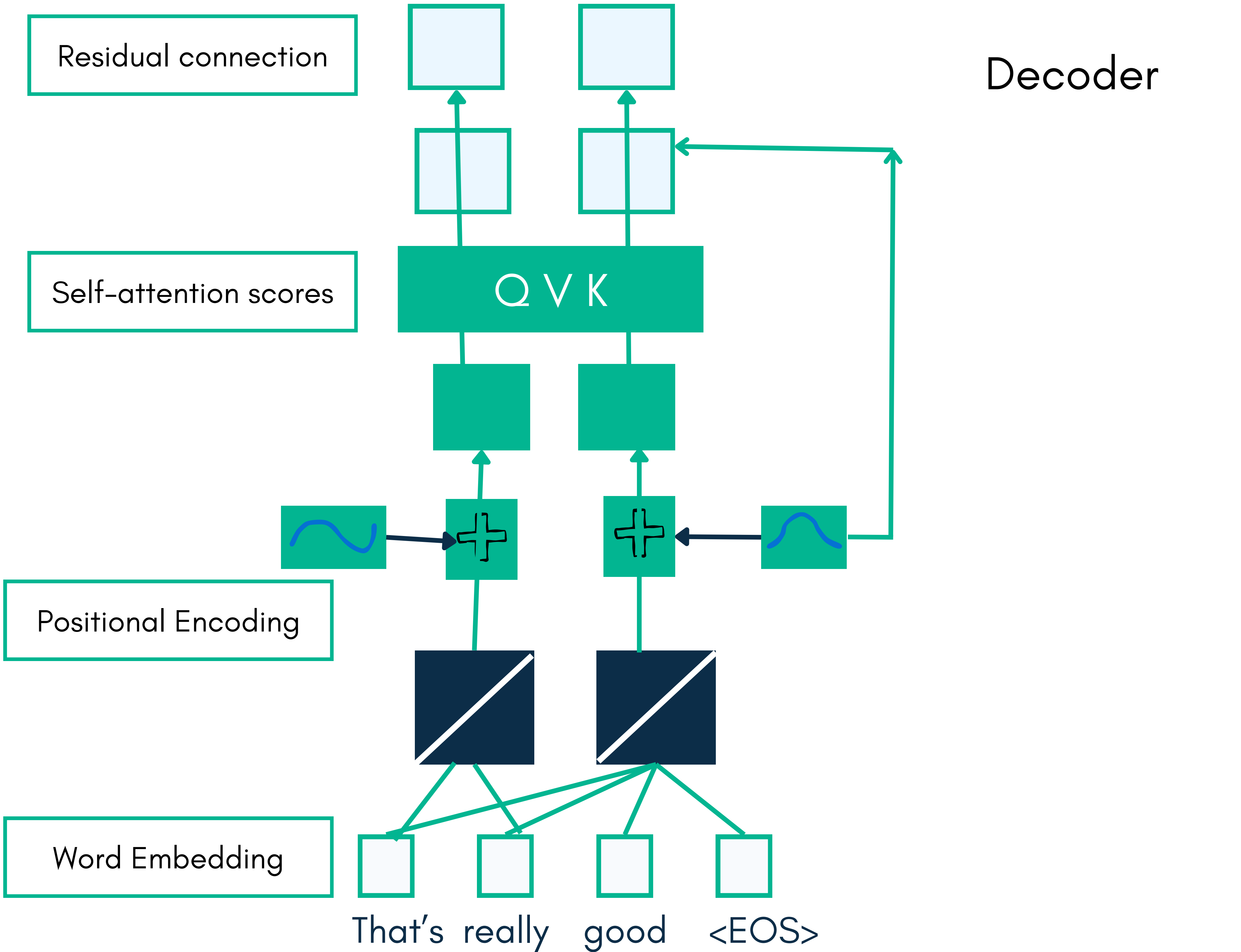


We take the position encoded values and them to the self-attention values ..

These by-passes are called Residual Connections and they make it easier to train complex neural networks.....

It allows the self-attention layer to establish relation among the words without preserving word embedding and postional encoding information.


Residual connection
Self-attention scores
Positional Encoding
Word Embedding
Q V K
These 4 features allows transformer to:
encode words into numbers, encode the position of the words, encode the relationships among the words and ...
easily and quickly train in parallel
Pizzza
is
awesome
great
<EOS>


Residual connection
Self-attention scores
Positional Encoding
Word Embedding
Decoder
Q V K
That's
really
good
<EOS>

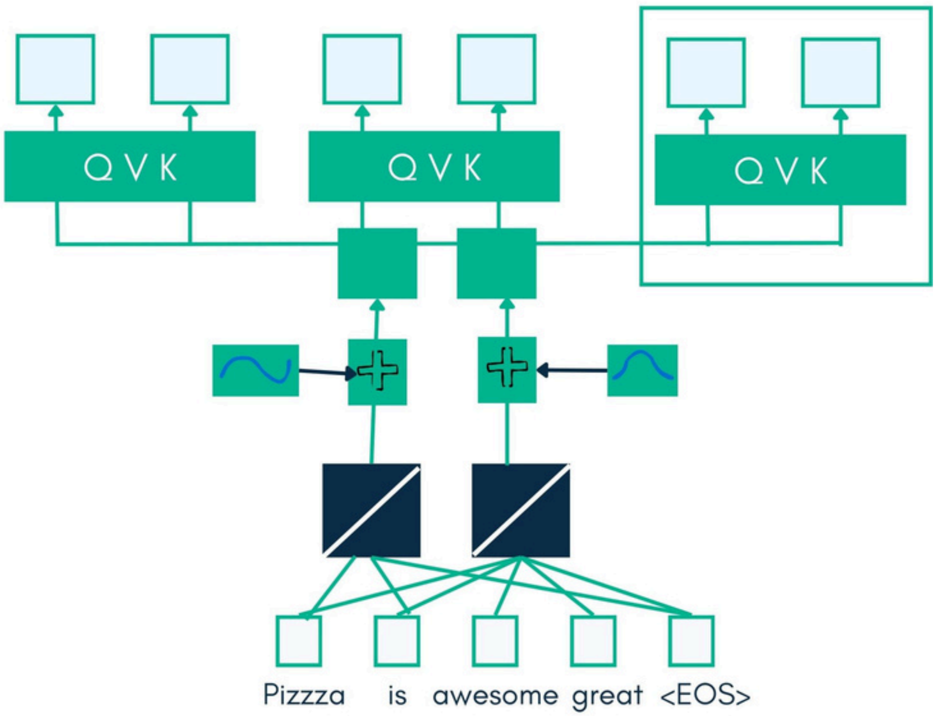
So far we talked about how self-attention keeps track of how words are related with a sentence ... However, since we are creating a conversational chatbot .... we need to keep track between the input sentence and the output sentence.

Its important for the **Decoder** to keep track of the significant words in the input.

So, the main idea of **encoder-decoder attention** is to keep track of the significant words in the input.

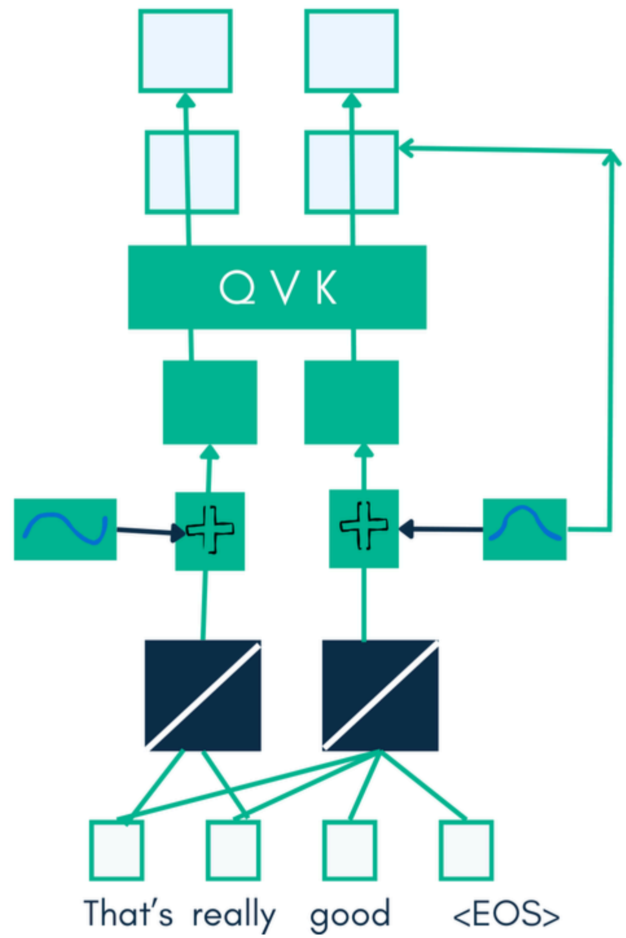
we create 2 new values to represent the Query of the <EOS> token in the Decoder

Q: [-0.9, 2.6]



Then we create keys for each word in the Encoder

K: [-4.5, 2.1, -0.5, 4.1]



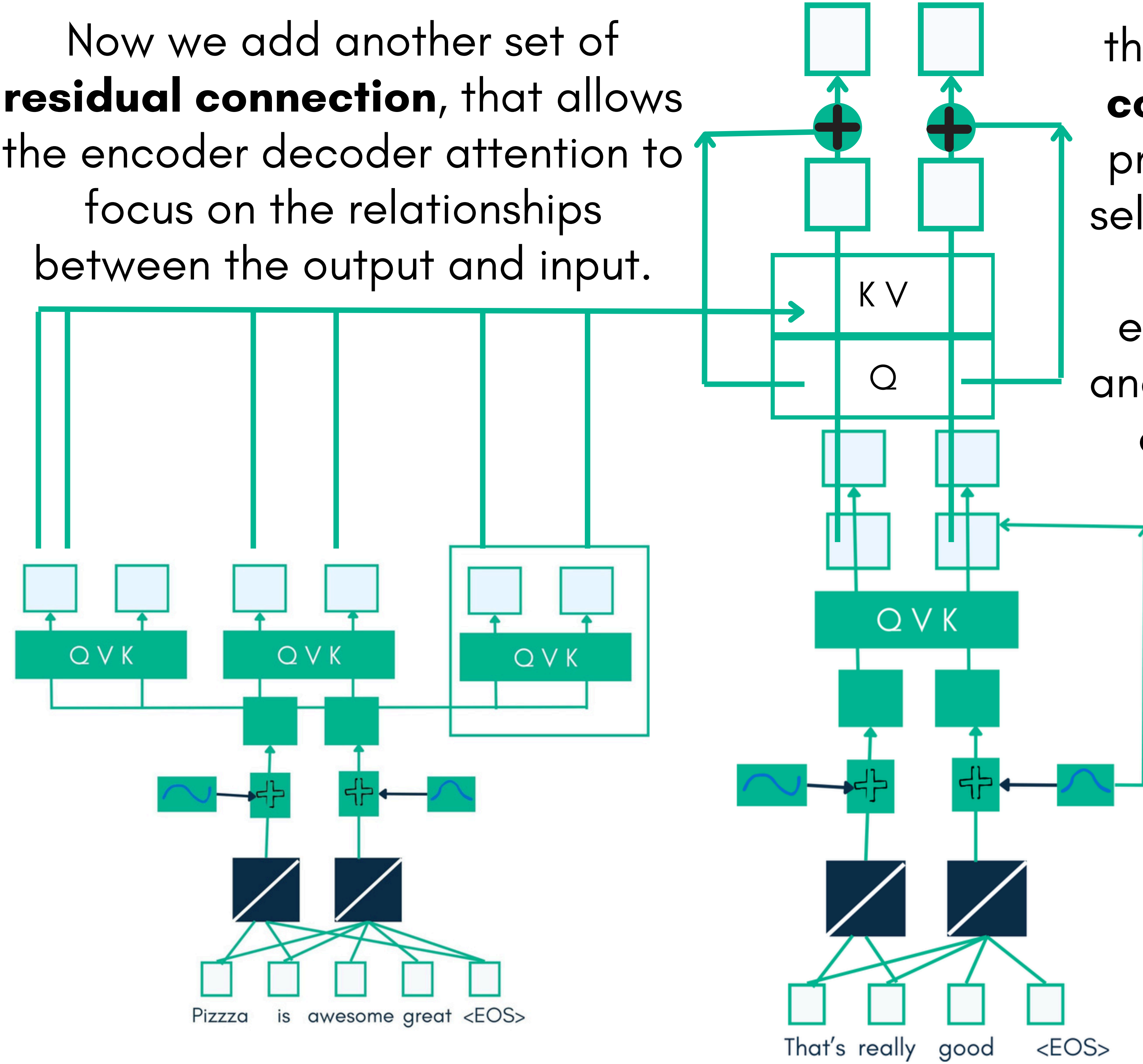
We calculate the similarities between <EOS> token in the decoder and each word in the encoder ...

By calculating the dot products.

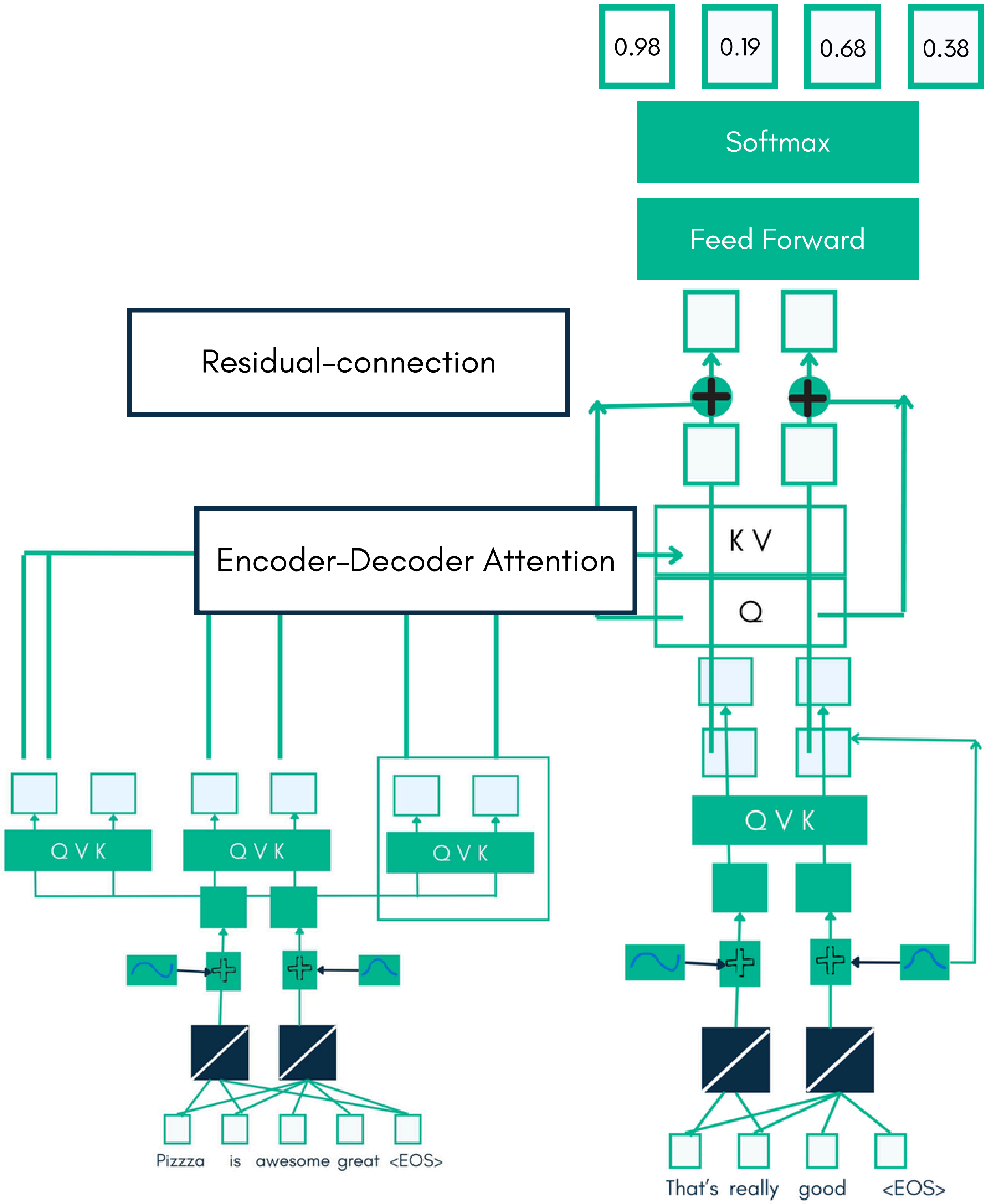
We then run the similarities through a softmax function.

We then calculate values of each input word and scale those values by softmax percentages. Finally, add the pair of scaled values together to get Encoder-Decoder attention values.

Now we add another set of **residual connection**, that allows the encoder decoder attention to focus on the relationships between the output and input.



the **residual connection**, preserve the self-attention, word embedding and positional encoding



Finally, we need to connect a fully connected layer to calculate the probabilities of the tokens "That's", "really", "good", "<EOS>". The fully connected layer has one input for each value of the current token, so in this case **2** ...

and one output for each token in the output vocabulary wihich is **4** in this case

We run the final Softmax function to select the sentence "That's really good <EOS>"

```python
# Updated vocabulary with <start> and <end>
token_to_id = {
  'what': 0,
  'is': 1,
  'LiveAI': 2,
  'awesome': 3,
  '<start>': 4,
  '<end>': 5
}
id_to_token = dict(map(reversed, token_to_id.items()))

# Encoder inputs: questions only (no special tokens)
encoder_inputs = torch.tensor([
  [token_to_id["what"], token_to_id["is"], token_to_id["LiveAI"]],
  [token_to_id["LiveAI"], token_to_id["is"], token_to_id["what"]]
])
# Decoder inputs: start with <start> token
decoder_inputs = torch.tensor([
  [token_to_id["<start>"], token_to_id["awesome"]],
  [token_to_id["<start>"], token_to_id["awesome"]]
])
# Target outputs: shifted right by one position to predict next token
decoder_targets = torch.tensor([
  [token_to_id["awesome"], token_to_id["<end>"]],
  [token_to_id["awesome"], token_to_id["<end>"]]
])
```

- This is what you feed into the decoder at each time step during training.
- It starts with <start> token, followed by the partial/generated output so far (awesome).
- Purpose: allows the decoder to learn to generate the next token given prior tokens.

```python
# Updated vocabulary with <start> and <end>
token_to_id = {
  'what': 0,
  'is': 1,
  'LiveAI': 2,
  'awesome': 3,
  '<start>': 4,
  '<end>': 5
}
id_to_token = dict(map(reversed, token_to_id.items()))

# Encoder inputs: questions only (no special tokens)
encoder_inputs = torch.tensor([
  [token_to_id["what"], token_to_id["is"], token_to_id["LiveAI"]],
  [token_to_id["LiveAI"], token_to_id["is"], token_to_id["what"]]
])
# Decoder inputs: start with <start> token
decoder_inputs = torch.tensor([
  [token_to_id["<start>"], token_to_id["awesome"]],
  [token_to_id["<start>"], token_to_id["awesome"]]
])
# Target outputs: shifted right by one position to predict next token
decoder_targets = torch.tensor([
  [token_to_id["awesome"], token_to_id["<end>"]],
  [token_to_id["awesome"], token_to_id["<end>"]]
])
```

- This is the correct output we expect the decoder to predict at each position.
- It’s shifted by one position compared to decoder_inputs → so the model predicts awesome after <start>, and <end> after awesome.
- Purpose: serves as the ground truth labels for loss calculation.

```
class Encoder(nn.Module):

  def __init__(self, num_tokens=4, d_model=2,
max_len=6):

    super().__init__()


    self.we =
nn.Embedding(num_embeddings=num_tokens,
                embedding_dim=d_model)

    self.pe = PositionEncoding(d_model=d_model,
                 max_len=max_len)

    self.self_attention = Attention(d_model=d_model)
    self.layernorm = nn.LayerNorm(d_model)

    self.fc_layer = nn.Linear(in_features=d_model,
out_features=num_tokens)
```

A Encoder Transformer simply brings together...

- Word Embedding
- Position Encoding
- Self-Attention
- Residual Connections + Normalization
- A fully connected layer

```
class Encoder(nn.Module):

  def __init__(self, num_tokens=4, d_model=2,
max_len=6):

    super().__init__()


    self.we =
nn.Embedding(num_embeddings=num_tokens,
                embedding_dim=d_model)

    self.pe = PositionEncoding(d_model=d_model,
                  max_len=max_len)

    self.self_attention = Attention(d_model=d_model)
    self.layernorm = nn.LayerNorm(d_model)

    self.fc_layer = nn.Linear(in_features=d_model,
out_features=num_tokens)
```

A Encoder Transformer simply brings together...

- Word Embedding
- Position Encoding
- Self-Attention
- Residual Connections + Normalization
- A fully connected layer

This is an encoder block that applies embedding → positional encoding → self-attention → add & norm → linear projection → normalized output

```python
class Decoder(nn.Module):

  def __init__(self, num_tokens=4, d_model=2,
max_len=6):

    super().__init__()


    self.we =
nn.Embedding(num_embeddings=num_tokens,
                 embedding_dim=d_model)

    self.pe = PositionEncoding(d_model=d_model,
                  max_len=max_len)

    self.self_attention = Attention(d_model=d_model)
    self.cross_attention =
Attention(d_model=d_model)
    self.layernorm1 = nn.LayerNorm(d_model)
    self.layernorm2 = nn.LayerNorm(d_model)


    self.fc_layer = nn.Linear(in_features=d_model,
out_features=num_tokens)
```

A Decoder Transformer simply brings together...

- Word Embedding
- Position Encoding
- Masked-Attention
- Residual Connections + Normalization
- Encoder-Decoder Attention
- Residual Connections + Normalization
- A fully connected layer

```
def forward(self, token_ids,encoder_k,
encoder_v):
    device = token_ids.device
    word_embeddings = self.we(token_ids)

    position_encoded = self.pe(word_embeddings)


    mask =
torch.tril(torch.ones((token_ids.size(dim=0),
token_ids.size(dim=0))))

    mask = mask == 0
```

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

- For the decoder-only transformer, we need to use "masked self-attention" so that
- when we are training we can't cheat and look ahead at
- what words come after the current word.
- To create the mask we are creating a matrix where the lower triangle is filled with 1, and everything above the diagonal is filled with 0s.

```
mask_self_attention_values =
self.self_attention(position_encoded,
                    position_encoded,
                    position_encoded,
                    mask=mask)

   residual_connection_values =
self.layernorm1(position_encoded +
mask_self_attention_values)

x_cross_att =
self.cross_attention(residual_connection_values,
encoder_k, encoder_v, mask=None)

x = self.layernorm2(residual_connection_values +
x_cross_att)

fc_layer_output = self.fc_layer(x)


   return fc_layer_output
```

- Computes self-attention over decoder input, applying a mask to prevent attending to future tokens.
- Adds original input to self-attention output (residual connection) and normalizes.
- Decoder attends over encoder's key/value outputs to gather relevant information from the encoder.
- Adds result of cross-attention to earlier output and normalizes.
- Applies a linear transformation to the normalized output.

```
class Transformer(nn.Module):
  def __init__(self, num_tokens, d_model,
max_len):
    super().__init__()

    self.encoder =
Encoder(num_tokens=num_tokens,
d_model=d_model, max_len=max_len)
    self.decoder =
Decoder(num_tokens=num_tokens,
d_model=d_model, max_len=max_len)


    self.output_linear = nn.Linear(num_tokens,
num_tokens)

def forward(self, src_tokens, tgt_tokens):

    encoder_output, encoder_hidden =
self.encoder(src_tokens)
    decoder_output = self.decoder(tgt_tokens,
encoder_output, encoder_hidden)

    return decoder_output
```

- Builds an Encoder and Decoder with specified token size, model dimension, and max sequence length.
- Adds an output linear layer to map final outputs.
- Forward pass:
- Encode: Passes src_tokens through the encoder to get encoder_output and encoder_hidden.
- Decode: Feeds tgt_tokens along with encoder outputs into the decoder.
- Return decoder output as the model’s output.