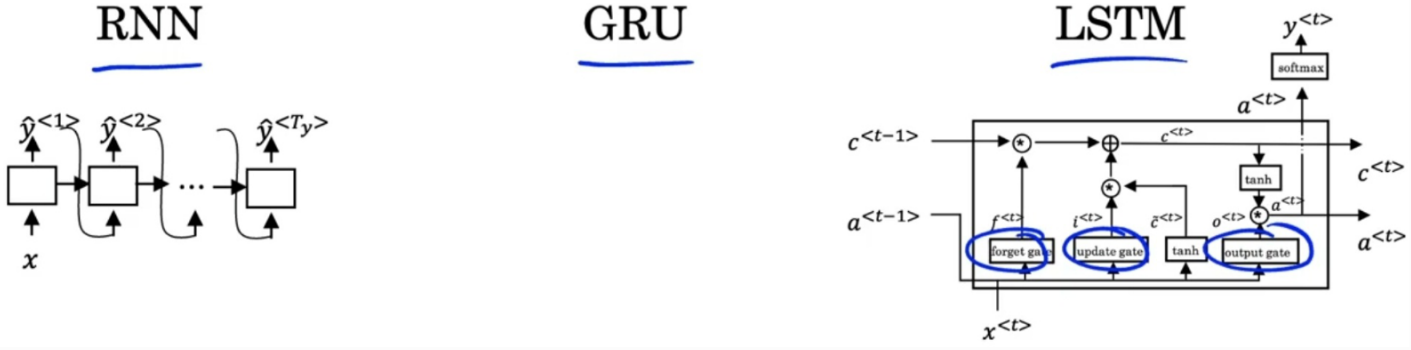
Transformer Network Motivation

increased complexity
sequential →

RNN | GRU | LSTM

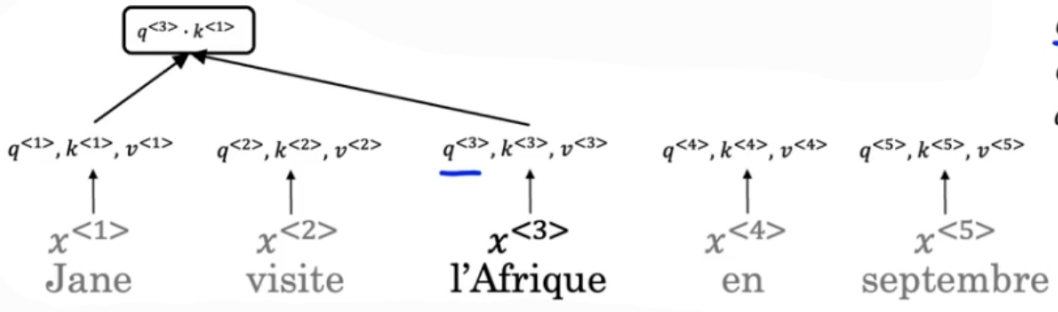


Transformers can be computed in parallel!!
the intuition behind, is to combine attention + CNN.
Specifically, self-attention and multi-head attention.

Self-attention:

A(q, K, v) ⇒ like a database, we search for a query, obtain the best Key an get its value.
query Key value

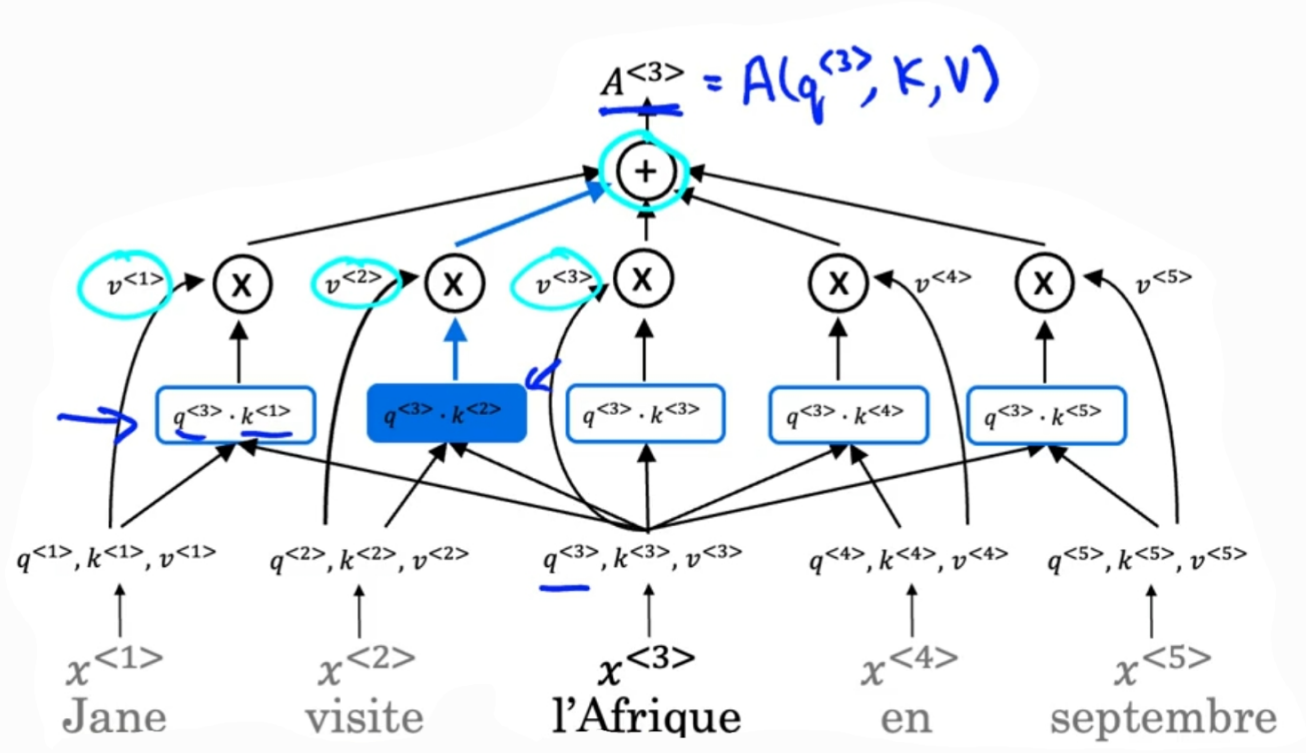
Query (Q)	Key (K)	Value (V)
$q^{<1>}$	$k^{<1>}$ person	$v^{<1>}$
$q^{<2>}$	$k^{<2>}$ action	$v^{<2>}$
$q^{<3>}$ what's happening there	$k^{<3>}$	$v^{<3>}$
$q^{<4>}$	$k^{<4>}$	$v^{<4>}$
$q^{<5>}$	$k^{<5>}$	$v^{<5>}$



$q^{<3>} = W^Q x^{<3>}$
$k^{<3>} = W^K x^{<3>}$
$v^{<2>} = W^V x^{<3>}$

So, if our query for $X^{<3>}$ is "what is happening there?", we will get the best K with $K^{<2>}$, as it is the verb (inner product $q^{<i>} K^{<j>}$).

We then, take all the inner products and compute a softmax, and multiply to each the corresponding $v^{<i>}$. Finally, sum them all, and there it is $A^{<i>}$!!



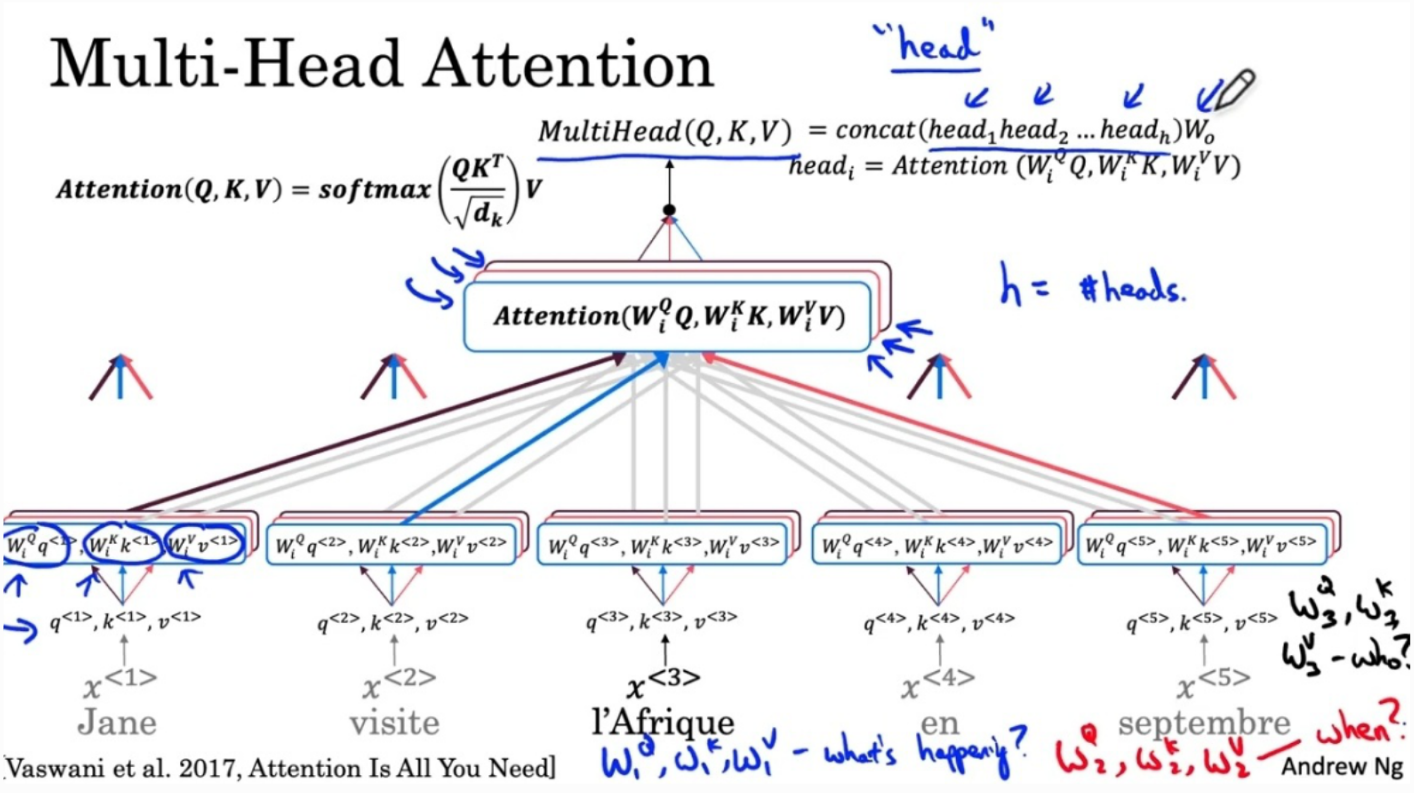
Consequently, this process can be computed in parallel for each $x^{<i>}$.

$$A(q,K,V) = \sum_i \frac{\exp(q \cdot k^{<i>})}{\sum_j \exp(q \cdot k^{<j>})} v^{<i>}$$

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

The Key advantage, is that each word does not have a fixed embedding, instead, it realizes what it really means (action, adverb, location, ...).
Therefore, it allows a much richer representation.

Multi-head attention: we ask many questions, compute self-attention for each, stack the results and then, pass them to a NNet.



Vaswani et al. 2017, Attention Is All You Need]

TRANSFORMER NETWORK

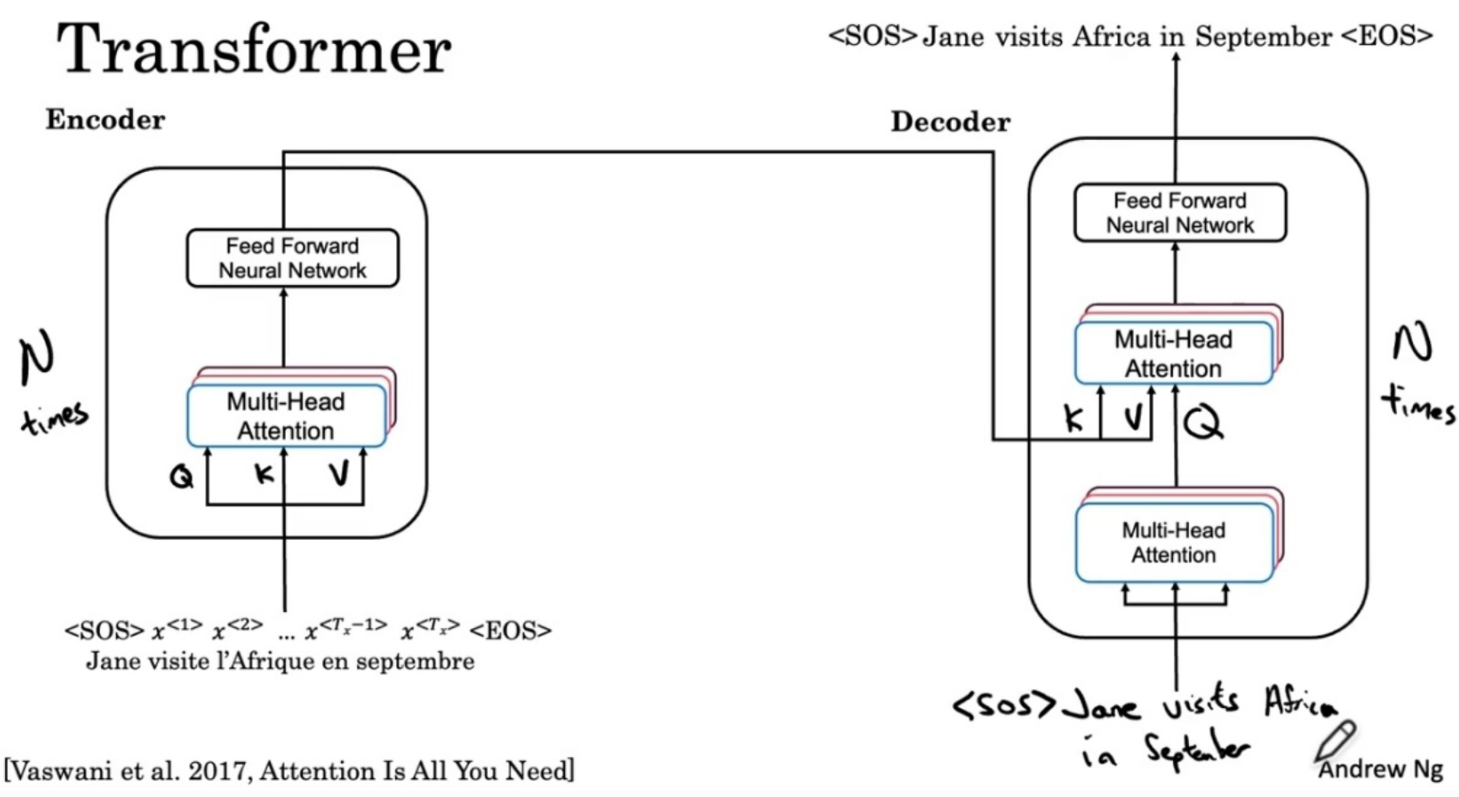
1st: Encoder block:

Feed Q, K and V into a multi-head attention layer. Then, the resulting matrix is passed into a feed forward NNet. We repeat it N times.

2nd: Decoder block:

The multi-head attention, at the beginning receives as input a <SOS> (start of sentence). With that, it generates a Q, which with the output of the encoder, pass to other multi-head attention block. The results are fed to a NNet. Repeat N times.

The goal of the decoder, is to predict which is the next sentence given the input (for ex. <sos>). Once we get the prediction, we concatenate and feed the current predicted sentence to the decoder.



[Vaswani et al. 2017, Attention Is All You Need]

There are some additional details, which improve their performance.

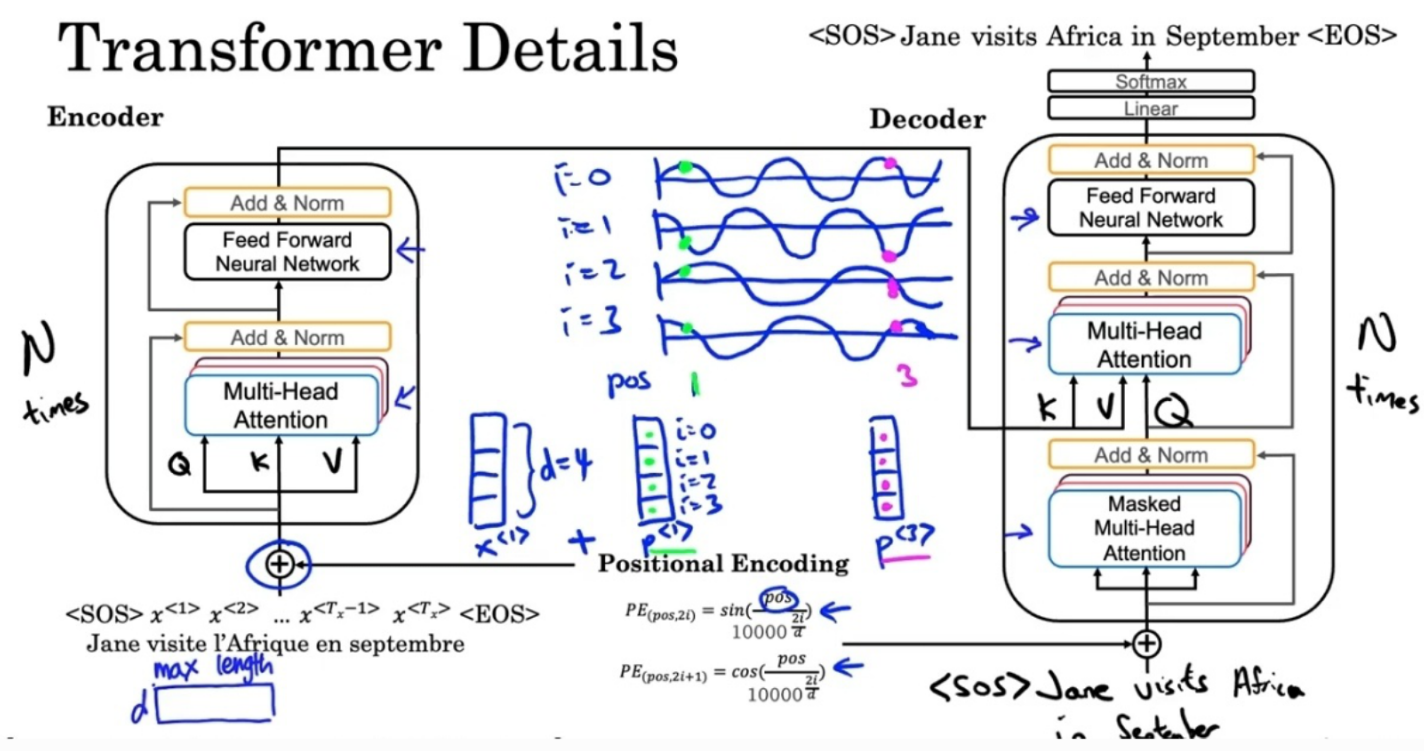
- Positional encoding: create an array for each word, of the same length of the embedding, and compute their sum.

Positional Encoding

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{\frac{2i}{d}}})$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{\frac{2i}{d}}})$$

- Add & Norm layer: speeds up learning. Kinda similar to batch-norm.
- For the output of the decoder, linear & softmax.
- The 1st MHAt layer of the encoder usually has a mask (important only during training). It hides the last x words, to see how it behaves in real-time.



What you should remember:

- The combination of self-attention and convolutional network layers allows of parallization of training and *faster training*.
- Self-attention is calculated using the generated query Q, key K, and value V matrices.
- Adding positional encoding to word embeddings is an effective way of include sequence information in self-attention calculations.
- Multi-head attention can help detect multiple features in your sentence.
- Masking stops the model from 'looking ahead' during training, or weighting zeroes too much when processing cropped sentences.

What you should remember:

- Positional encodings can be expressed as linear functions of each other, which allow the model to learn according to the relative positions of words.
- Positional encodings can affect the word embeddings, but if the relative weight of the positional encoding is small, the sum will retain the semantic meaning of the words.

3 - Self-Attention

As the authors of the Transformers paper state, "Attention is All You Need".

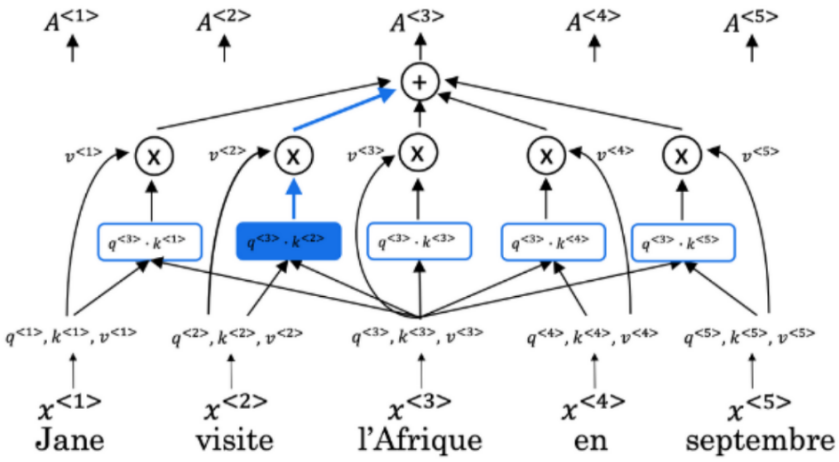


Figure 1: Self-Attention calculation visualization

The use of self-attention paired with traditional convolutional networks allows for the parallization which speeds up training. You will implement **scaled dot product attention** which takes in a query, key, value, and a mask as inputs to returns rich, attention-based vector representations of the words in your sequence. This type of self-attention can be mathematically expressed as:

$$\text{Attention}\ (Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V \tag{4}$$

- Q is the matrix of queries
- K is the matrix of keys
- V is the matrix of values
- M is the optional mask you choose to apply
- d_k is the dimension of the keys, which is used to scale everything down so the softmax doesn't explode

4 - Encoder

The Transformer Encoder layer pairs self-attention and convolutional neural network style of processing to improve the speed of training and passes K and V matrices to the Decoder, which you'll build later in the assignment. In this section of the assignment, you will implement the Encoder by pairing multi-head attention and a feed forward neural network (Figure 2a).

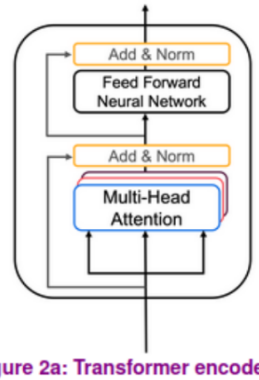


Figure 2a: Transformer encoder layer

4.2 - Full Encoder

Awesome job! You have now successfully implemented positional encoding, self-attention, and an encoder layer - give yourself a pat on the back. Now you're ready to build the full Transformer Encoder (Figure 2b), where you will embedd your input and add the positional encodings you calculated. You will then feed your encoded embeddings to a stack of Encoder layers.

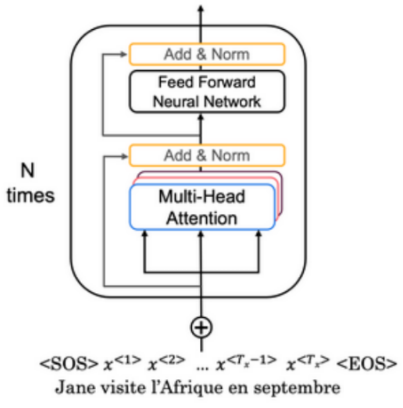


Figure 2b: Transformer Encoder

5 - Decoder

The Decoder layer takes the K and V matrices generated by the Encoder and in computes the second multi-head attention layer with the Q matrix from the output (Figure 3a).

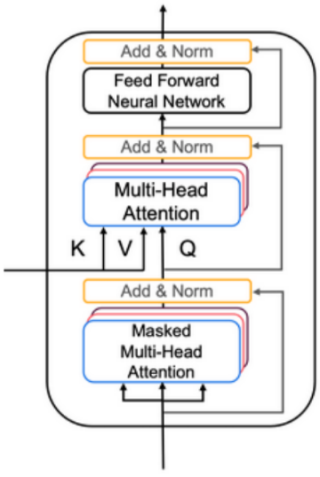


Figure 3a: Transformer Decoder layer

5.1 - Decoder Layer

Again, you'll pair multi-head attention with a feed forward neural network, but this time you'll implement two multi-head attention layers. You will also use residual connections and layer normalization to help speed up training (Figure 3a).

5.2 - Full Decoder

You're almost there! Time to use your Decoder layer to build a full Transformer Decoder (Figure 3b). You will embedd your output and add positional encodings. You will then feed your encoded embeddings to a stack of Decoder layers.

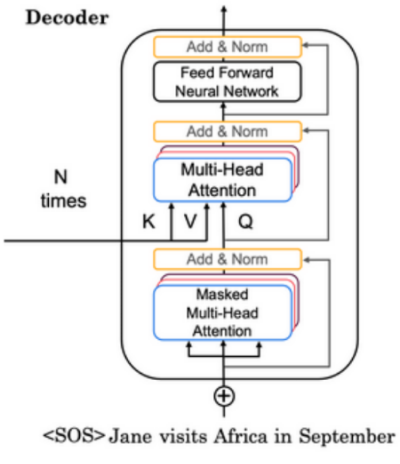


Figure 3b: Transformer Decoder

6 - Transformer

Phew! This has been quite the assignment, and now you've made it to your last exercise of the Deep Learning Specialization. Congratulations! You've done all the hard work, now it's time to put it all together.

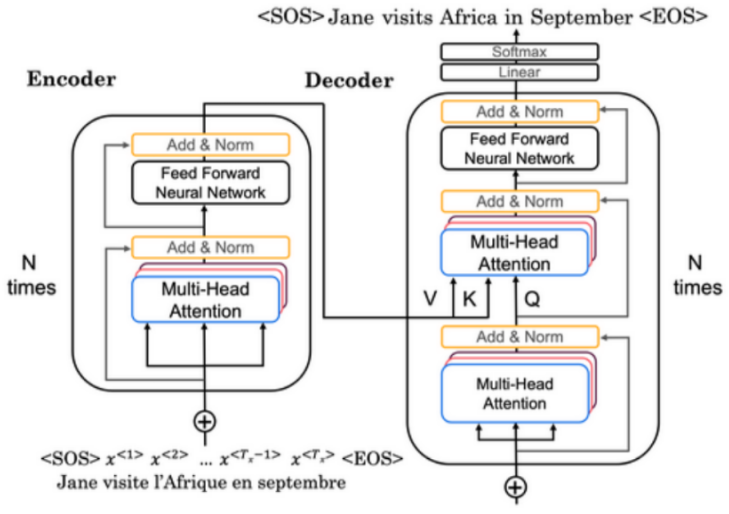


Figure 4: Transformer

The flow of data through the Transformer Architecture is as follows:

- First your input passes through an Encoder, which is just repeated Encoder layers that you implemented:
  - embedding and positional encoding of your input
  - multi-head attention on your input
  - feed forward neural network to help detect features
- Then the predicted output passes through a Decoder, consisting of the decoder layers that you implemented:
  - embedding and positional encoding of the output
  - multi-head attention on your generated output
  - multi-head attention with the Q from the first multi-head attention layer and the K and V from the Encoder
  - a feed forward neural network to help detect features
- Finally, after the Nth Decoder layer, two dense layers and a softmax are applied to generate prediction for the next output in your sequence.

1 - Positional Encoding

In sequence to sequence tasks, the relative order of your data is extremely important to its meaning. When you were training sequential neural networks such as RNNs, you fed your inputs into the network in order. Information about the order of your data was automatically fed into your model. However, when you train a Transformer network using multi-head attention, you feed your data into the model all at once. While this dramatically reduces training time, there is no information about the order of your data. This is where positional encoding is useful - you can specifically encode the positions of your inputs and pass them into the network using these sine and cosine formulas:

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{1}$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{2}$$

- d is the dimension of the word embedding and positional encoding
- pos is the position of the word.
- i refers to each of the different dimensions of the positional encoding.

To develop some intuition about positional encodings, you can think of them broadly as a feature that contains the information about the relative positions of words. The sum of the positional encoding and word embedding is ultimately what is fed into the model. If you just hard code the positions in, say by adding a matrix of 1's or whole numbers to the word embedding, the semantic meaning is distorted. Conversely, the values of the sine and cosine equations are small enough (between -1 and 1) that when you add the positional encoding to a word embedding, the word embedding is not significantly distorted, and is instead enriched with positional information. Using a combination of these two equations helps your Transformer network attend to the relative positions of your input data. This was a short discussion on positional encodings, but develop further intuition, check out the *Positional Encoding Ungraded Lab*.

Note: In the lectures Andrew uses vertical vectors, but in this assignment all vectors are horizontal. All matrix multiplications should be adjusted accordingly.

1.1 - Sine and Cosine Angles

Notice that even though the sine and cosine positional encoding equations take in different arguments (`2i` versus `2i+1`, or even versus odd numbers) the inner terms for both equations are the same:

$$\theta(pos, i, d) = \frac{pos}{10000^{\frac{2i}{d}}} \tag{3}$$

2 - Masking

There are two types of masks that are useful when building your Transformer network: the *padding mask* and the *look-ahead mask*. Both help the softmax computation give the appropriate weights to the words in your input sentence.

2.1 - Padding Mask

Oftentimes your input sequence will exceed the maximum length of a sequence your network can process. Let's say the maximum length of your model is five, it is fed the following sequences:

```
[["Do", "you", "know", "when", "Jane", "is", "going", "to", "visit", "Africa"],
 ["Jane", "visits", "Africa", "in", "September" ],
 ["Exciting", "!"]
]
```

which might get vectorized as:

```
[[ 71, 121, 4, 56, 99, 2344, 345, 1284, 15],
 [ 56, 1285, 15, 181, 545],
 [ 87, 600]
]
```

When passing sequences into a transformer model, it is important that they are of uniform length. You can achieve this by padding the sequence with zeros, and truncating sentences that exceed the maximum length of your model:

```
[[ 71, 121, 4, 56, 99],
 [ 2344, 345, 1284, 15, 0],
 [ 56, 1285, 15, 181, 545],
 [ 87, 600, 0, 0, 0],
]
```

Sequences longer than the maximum length of five will be truncated, and zeros will be added to the truncated sequence to achieve uniform length. Similarly, for sequences shorter than the maximum length, they zeros will also be added for padding. However, these zeros will affect the softmax calculation - this is when a padding mask comes in handy! You will need to define a boolean mask that specifies which elements you must attend(1) and which elements you must ignore(0). Later you will use that mask to set all the zeros in the sequence to a value close to negative infinity (-1e9). We'll implement this for you so you can get to the fun of building the Transformer network! 😇 Just make sure you go through the code so you can correctly implement padding when building your model.

2.2 - Look-ahead Mask

The look-ahead mask follows similar intuition. In training, you will have access to the complete correct output of your training example. The look-ahead mask helps your model pretend that it correctly predicted a part of the output and see if, *without looking ahead*, it can correctly predict the next output.

For example, if the expected correct output is `[1, 2, 3]` and you wanted to see if given that the model correctly predicted the first value it could predict the second value, you would mask out the second and third values. So you would input the masked sequence `[1, -1e9, -1e9]` and see if it could generate `[1, 2, -1e9]`.