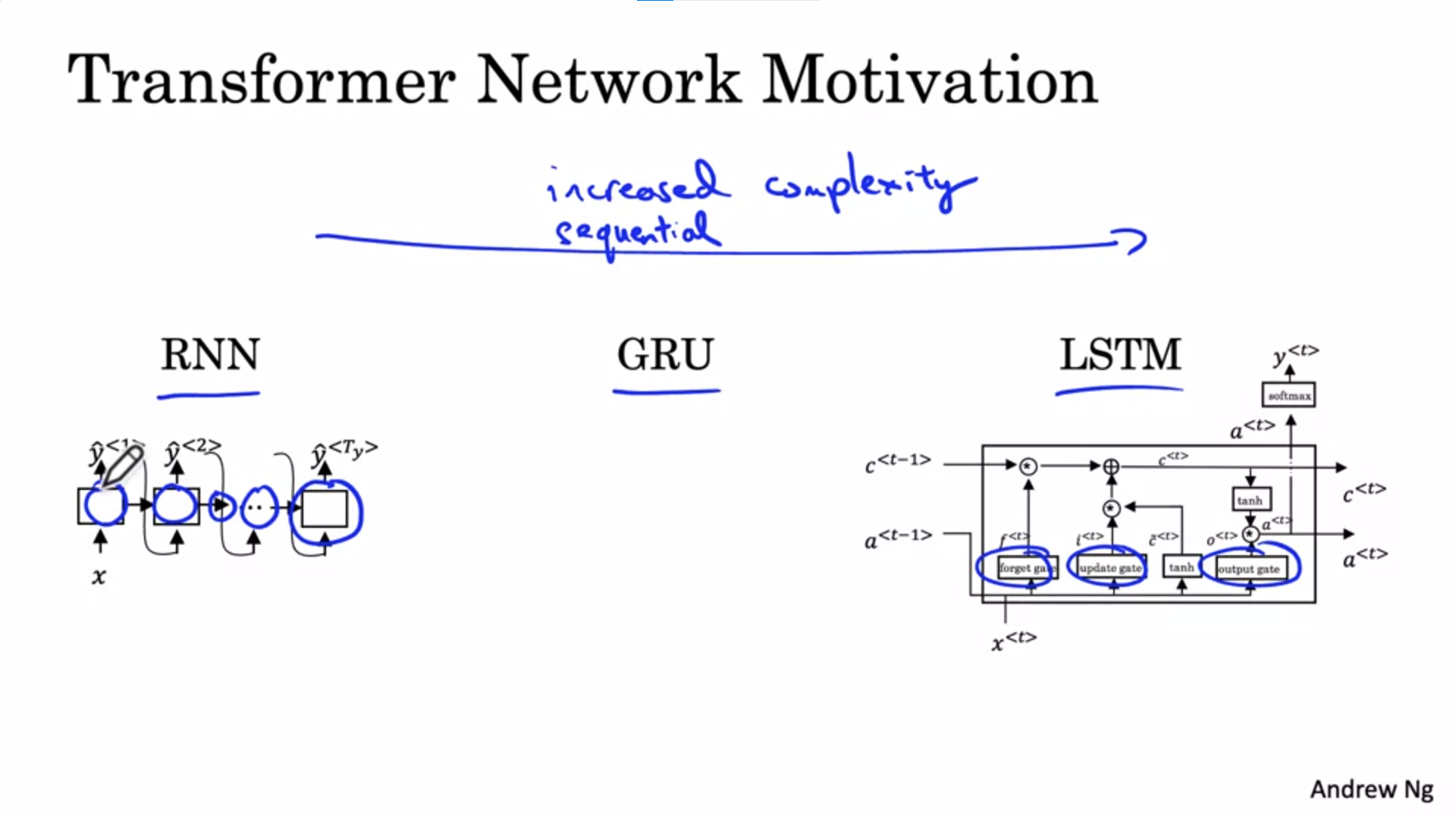


One of the most exciting developments in deep learning has been the transformer network, sometimes called transformers. This is an architecture that has completely taken the NLP word by storm. Many of the most effective algorithms in NLP today are based on the transformer architecture.

It is a relatively complex network architecture. We will go through it now.



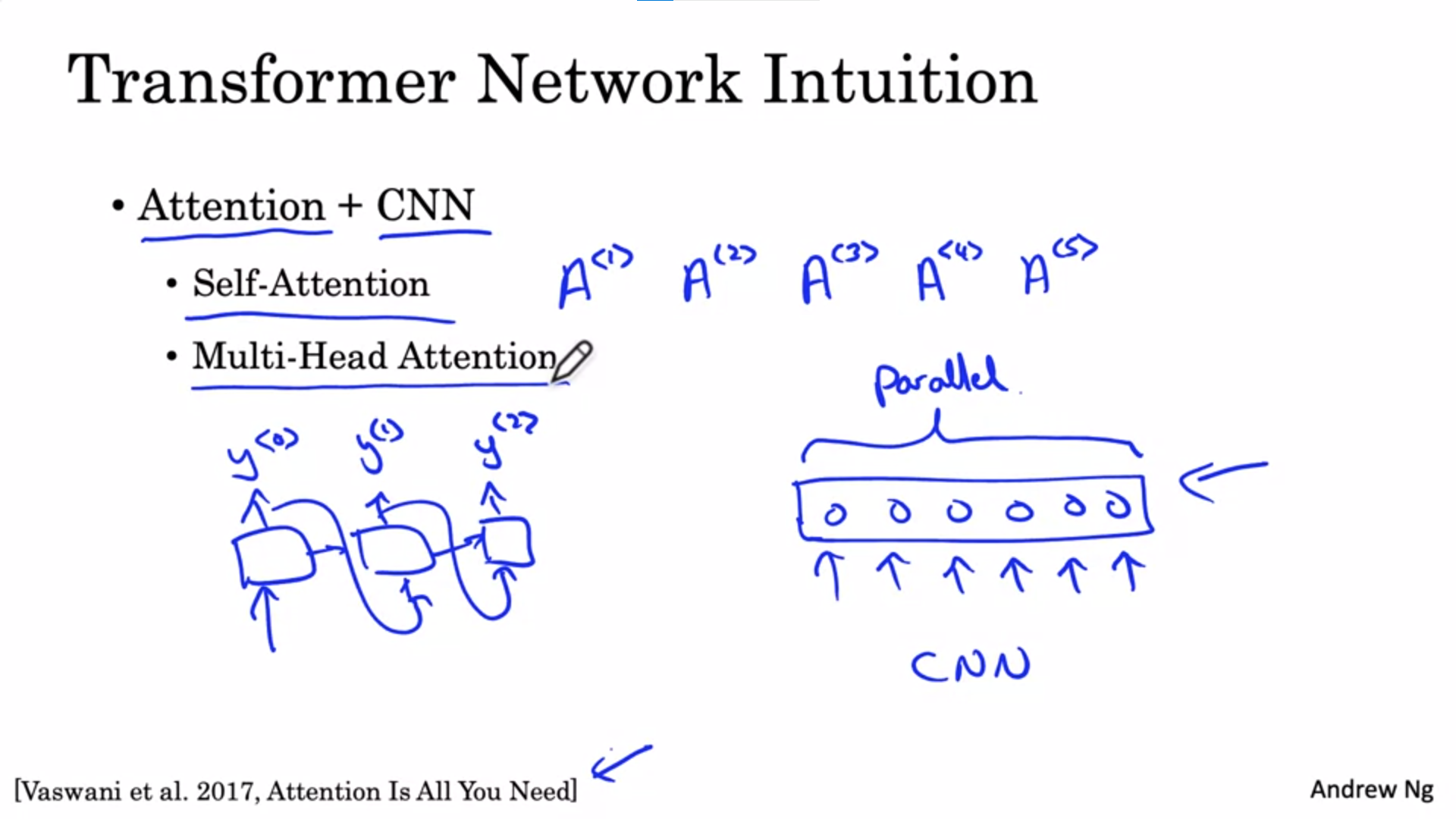
As the complexity of the sequence to sequence task increases so does the complexity of the model.

We had started with the RNN where we found problems with vanishing gradients and thus problems with capturing long-range dependencies. We then looked at the GRU and LSTM models as a way to resolve these problems of the RNN.

LSTM: makes use of gates to control the flow of information, but comes with more computations and more complexity.

All of these models are still sequential models in that they ingested the input sentence one word or one token at a time. Each unit is a bottleneck to the flow of information. E.g. to compute the final output we first had to calculate the output of all units that come before.

The transformer architecture allows us to run a lot more of these computations in parallel. We can ingest a whole sentence all at the same time instead of processing it one word at a time from the left to the right.



The transformer network was published in a seminal paper mentioned on the slide here.

One of the inventors of the transformer network is co-instructor of the NLP specialization from deeplearning.ai.

The major innovation of the transformer architecture is combining the use of attention-based representation and a CNN style of processing.

RNN processes one output at a time -> very sequential way of processing tokens. Contrast that to a CNN which can take as input a lot of pixels (or words) and can compute representations for them in parallel.

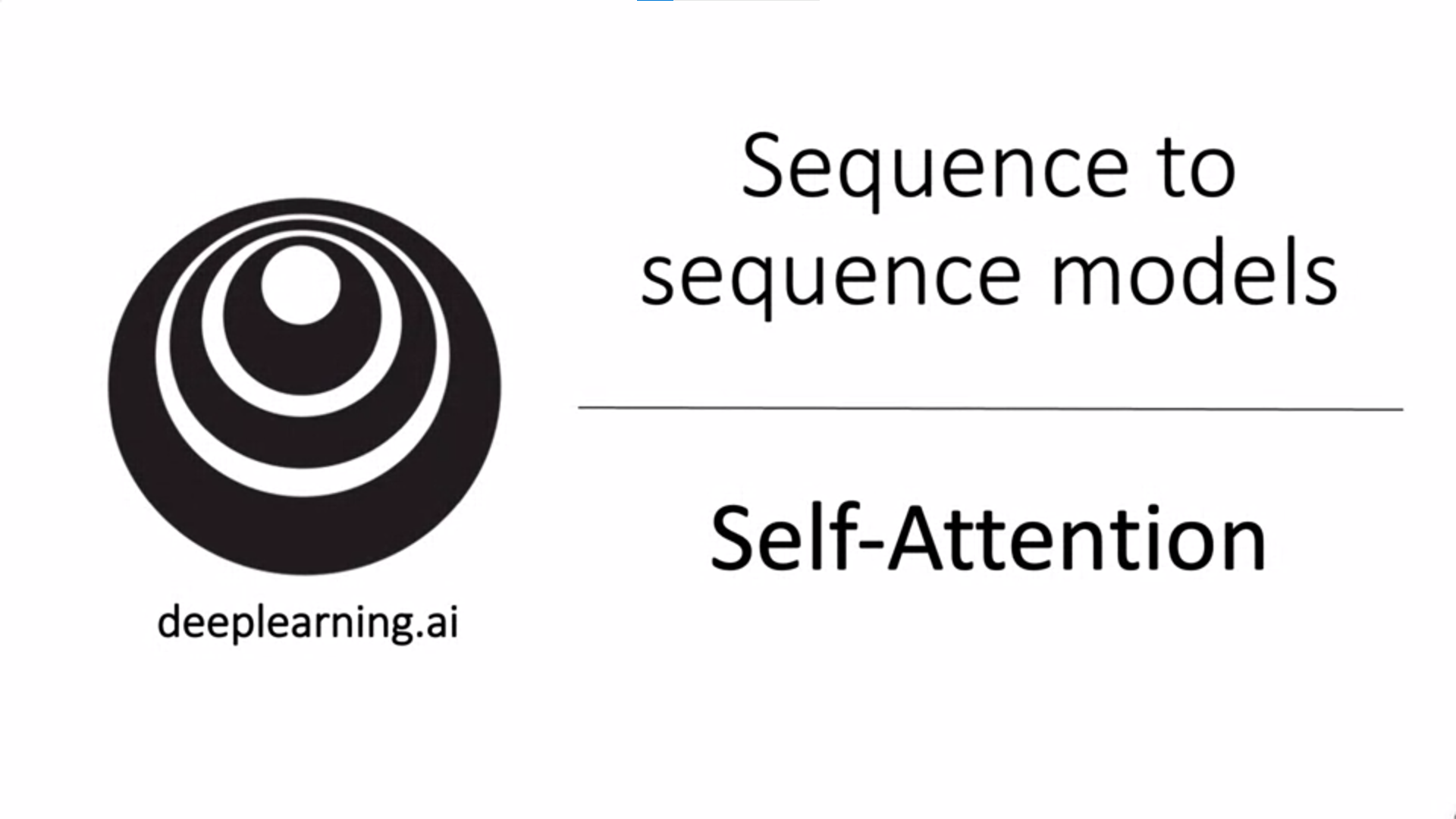
The transformer network computes very rich very useful representations but more with a CNN style of processing. There are two key ideas: self-attention and multi-head attention.

The goal of self-attention is, given a sentence with 5 words, to compute 5 representations, one for each word in parallel.

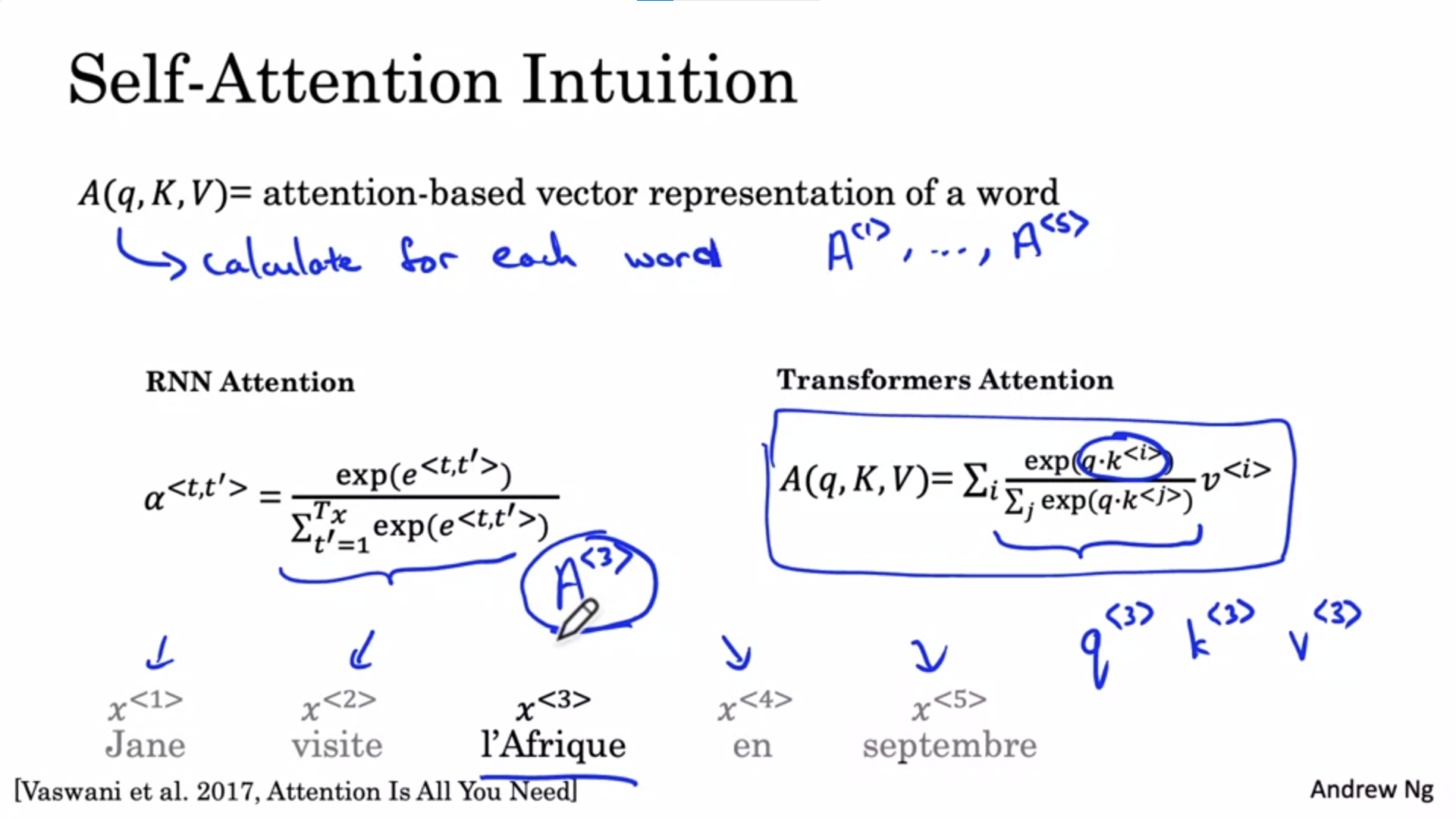
Multi-head attention is basically a for loop over the self-attention process. We end up with multiple versions of these representations.

It turns out that these representations will be very rich representations that can be used for machine translation or other NLP tasks very effectively.

In the next section let’s jump in to learn about self-attention to compute these rich representations. After that we will talk about multi-head attention. Finally we will put everything together to understand the entire transformer architecture.



Self-attention is the most important core idea behind what makes transformer networks work.



We have seen previously how attention is used with sequential neural networks such as RNNs.

To use attention with a style more like CNNs we need to calculate self-attention where we create an attention-based representation for each word in the input sentence.

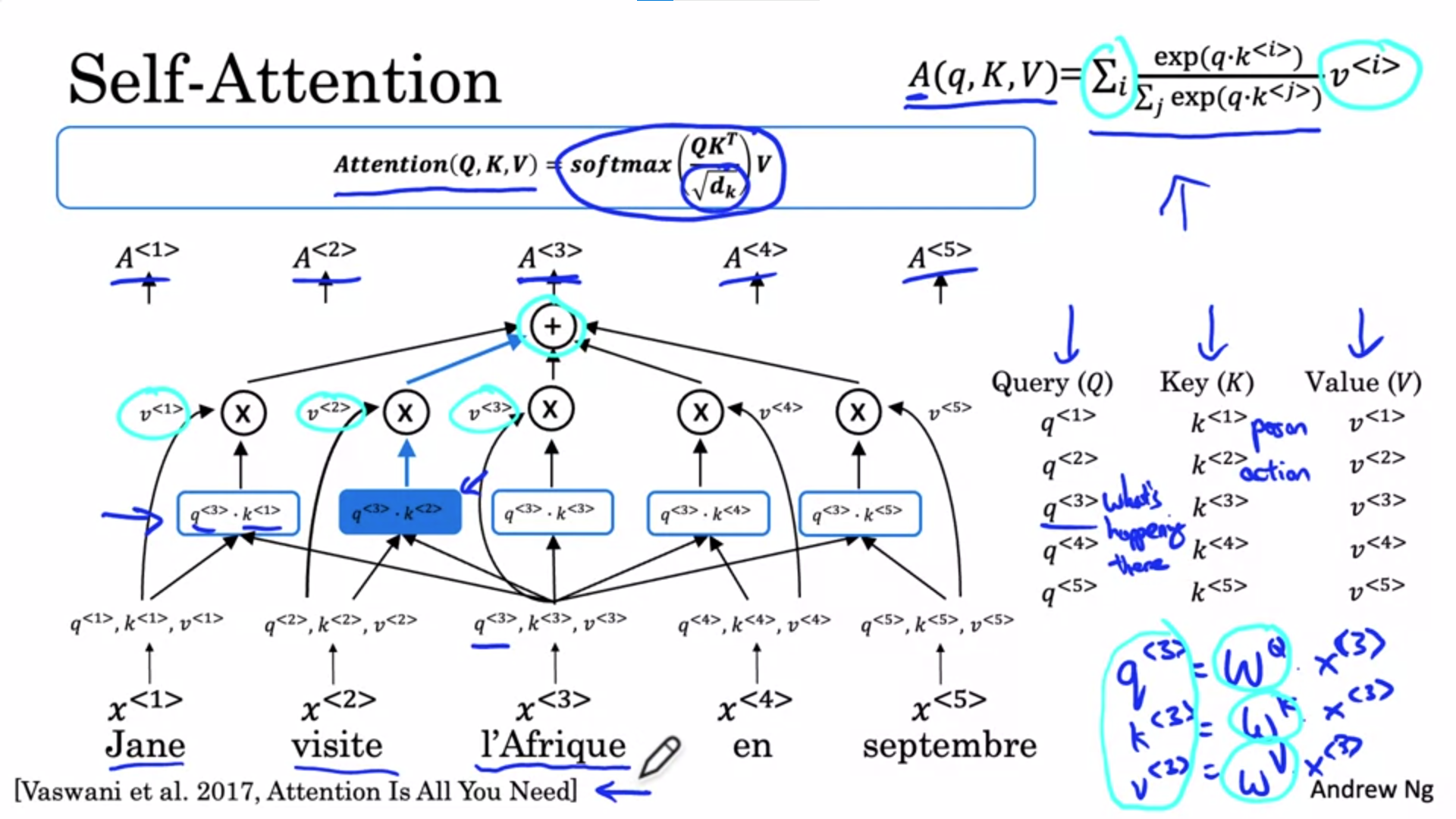
q, K, V will be explained in a later slide.

We will focus on the word l’Afrique. Calculate a vector representation of this word. Previously we have seen word embeddings. One way to represent l’Afrique would be to just look up the word embedding. But depending on the context we might think about different things (e.g. Africa as a site of historical interest, or as a site of tourist destination or as the second largest continent). Depending on how we think about it we might want to represent it differently.

A<3> will do exactly this. It will look at the surrounding words to try to figure out what is actually going on in the sentence and how we try to talk about Africa. The calculation will not be very different from what we have seen previously as applying attention in the context of RNNs. The difference is that we will do it for all 5 words in parallel.

On the slide on the left we see the RNN attention. The attention equation is instead going to look like on the right. The equations have some similarity. The inner term also involves a softmax. We can think of exponent terms as being akin to attention values. Exactly how these terms are going to be worked out we will see on the next slide.

For every word, e.g. l’Afrique we have 3 values: the query q, the key k and the value v. These vectors are the key inputs to computing the attention values for each word.



Let’s walk through the steps needed to actually calculate A<3>.

Associate each of the words with 3 values, the query, key, value pairs.

q<3> is a learned matrix. Similar for the key, value pairs.

The matrices are parameters of the learning algorithm. They allow us to pull up the query, key and value vectors for each word.

Query, key, value vectors were named using a lose analogy to a concept in databases where we can have a query and key value pairs.

q<3> is a question that we get to ask about l’Afrique. E.g. what is happening there?

Compute the inner product between query 3 and key 1. This will tell us how good as an answer word 1 is to the question of what is happening in Africa. Then compute the inner product between q<3> and k<2>. This is intended to tell us how good is “visite” as an answer to the question of what is happening in Africa. And so on for the other words in the sequence.

The goal of this operation is to pull up the most information as needed to compute the most useful representation A<3>.

For intuition: if k<1> represents that “Jane” is a person, and k<2> represent that “visite” is an action, then we may find that q<3>’s inner product with k<2> has the largest value and this might suggest that “visite” gives us the most relevant context for what is happening in Africa (a destination for a visit).

Then we take the 5 values in the blue row and compute a softmax over them. The blue shaded indicates the largest value.

We are gonna take the softmax values and multiply them with v<1> (the value for word 1), v<2> (the value for word 2) and so on. Then we sum them all up and this gives us A<3>.

They key advantage of this representation is that the vector of l’Afrique is not some sort of fixed word embedding but it lets the self-attention mechanism realize that l’Afrique is the destination of a visit, thus producing richer representations of the word.

We do the same for all 5 words.

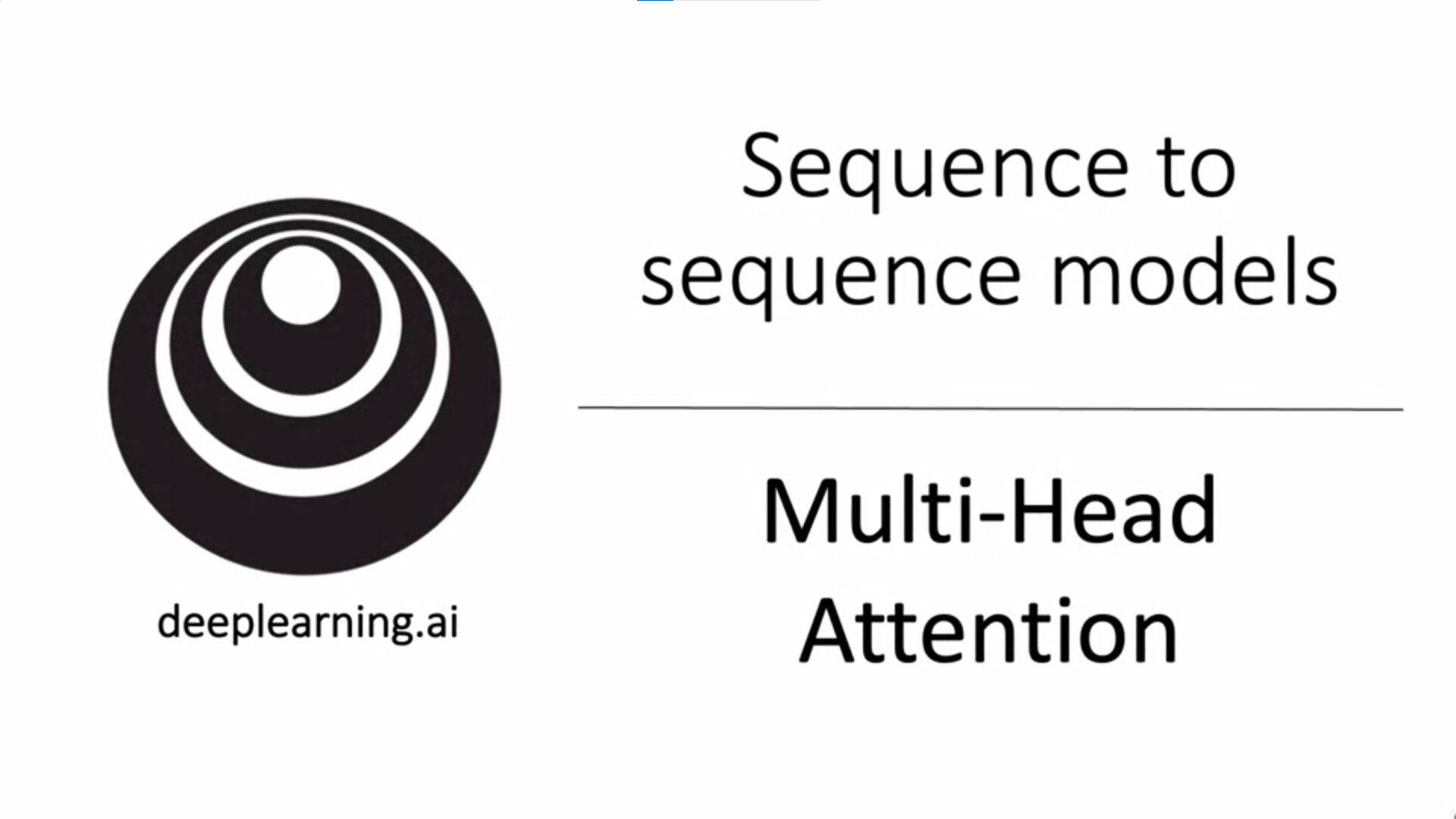
The notation used in literature looks like displayed in the blue box at the top. This is a compressed or vectorized representation of the equation in the top right.

The term in the denominator is just to scale the dot product so that it does not explode.

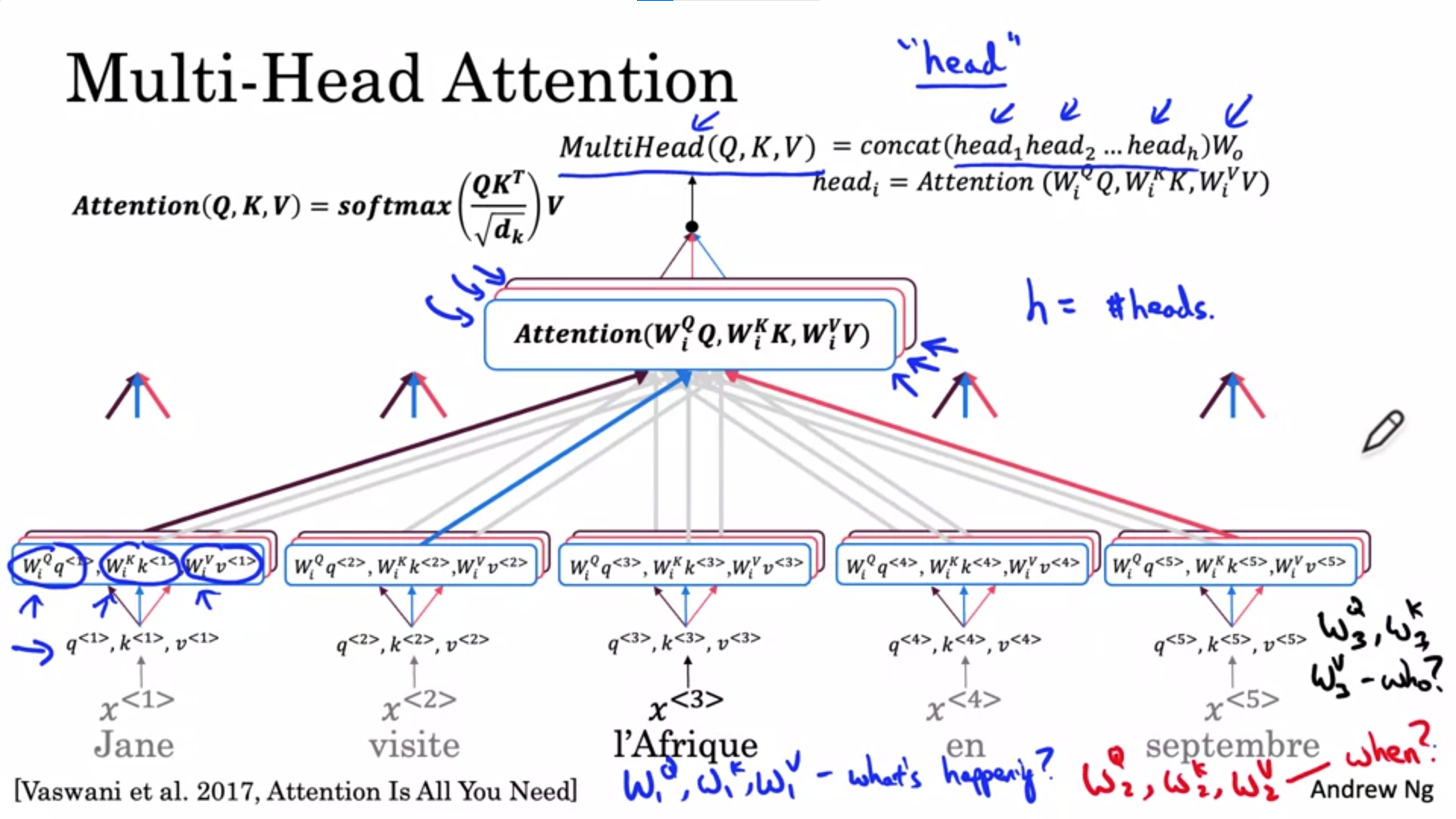
Another term for this kind of attention is the scaled dot product attention.

To recap: associated with each of the five words we end up with a query, key, value. The query lets us ask a question about that word. The key’s similarity to the query helps us figure out which word gives the most relevant answer to that question. Finally, the value allows a representation to plug in e.g. how “visite” should be represented within the representation of Africa (A<3>). This helps us come up with a representation which not only tells us that this is Africa but that someone is visiting Africa. This is a much more nuanced richer representation for the word than if we just had the same fixed word embedding no matter the context.

Now we will put a big for loop over this whole thing and that will be the multi-head attention mechanism.



Let’s learn about the multi-head attention mechanism. The notation gets a bit complicated but keep in mind that it is a for loop over the self-attention mechanism.



Each time we calculate self-attention for a sequence it is called a “head”. MultiHead -> do it a bunch of times.

We have the same set of vectors, q, k, v. We calculate multiple self-attentions with different weight matrices. We do this for all the words.

For the sake of intuition think of W1q, W1k and W1v as asking the question “what is happening there”. This is more or less the self-attention mechanism shown earlier. For this question the word “visite” gives the best answer (highlighted with blue arrow). The inner product for the key of “visite” has the highest value for the “query” of l’Africa. We do the same for all words.

This is the first head of the multihead attention.

Now we do this a couple of times so that we have e.g. eight heads rather than 1 head (this would mean doing the computation 8x).

Now we do the computation for the second head. The second head will have new matrices W2q, W2k and W2v. Maybe the second question now is “When is something happening?”. The computation is exactly the same but with a new set of matrices. Now the inner product between the Septembre word key and the l’Afrique query will have the highest value. So this word will play a huge role for the second part of the representation of l’Afrique.

We now ask the third question as represented by W3q, W3k, W3v: “Who has something to do with Africa?”. Now the inner product between Jane and l’Afrique will be the highest and the value of Jane will play the biggest role in the third piece of the representation of l’Afrique.

In the literature the number of heads is usually represented as h.

In this example we have 3 heads, more typically you may use 8 heads.

At the end we will use the concatenation of all the heads to compute the output value of the multi-head attention (multiply the concatenation with W0).

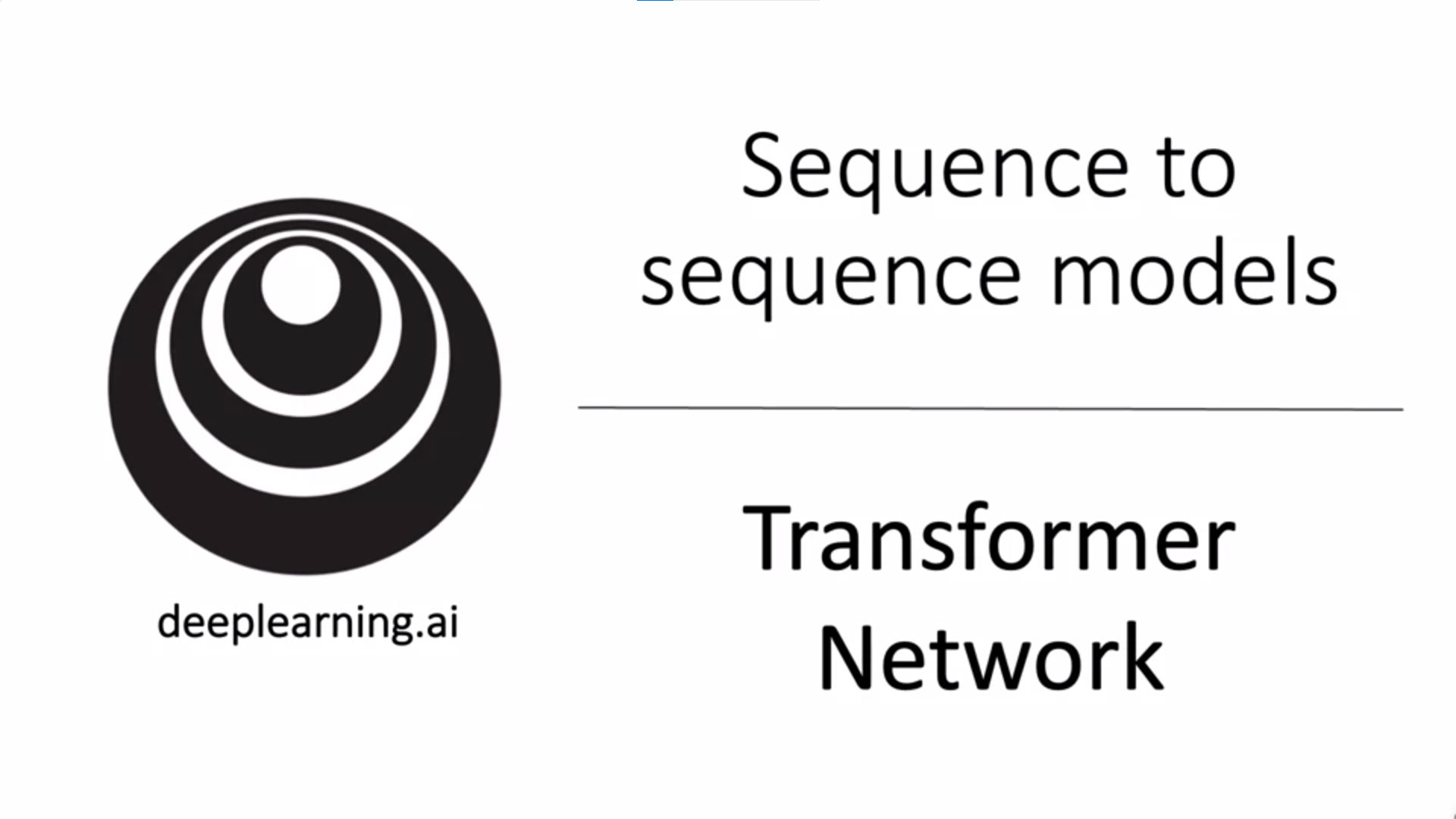
One detail that is worth keeping in mind is that in the description of multi-head attention Andrew has described computing different values for different heads in a big for loop. Conceptually that is ok but in practice these values are computed in parallel because no one head value depends on the values of other heads.

In the next video we will use a simplified icon as shown on the slide to denote this multi-head attention in the full transformer network. The little icon will denote all of these computations shown on the slide.

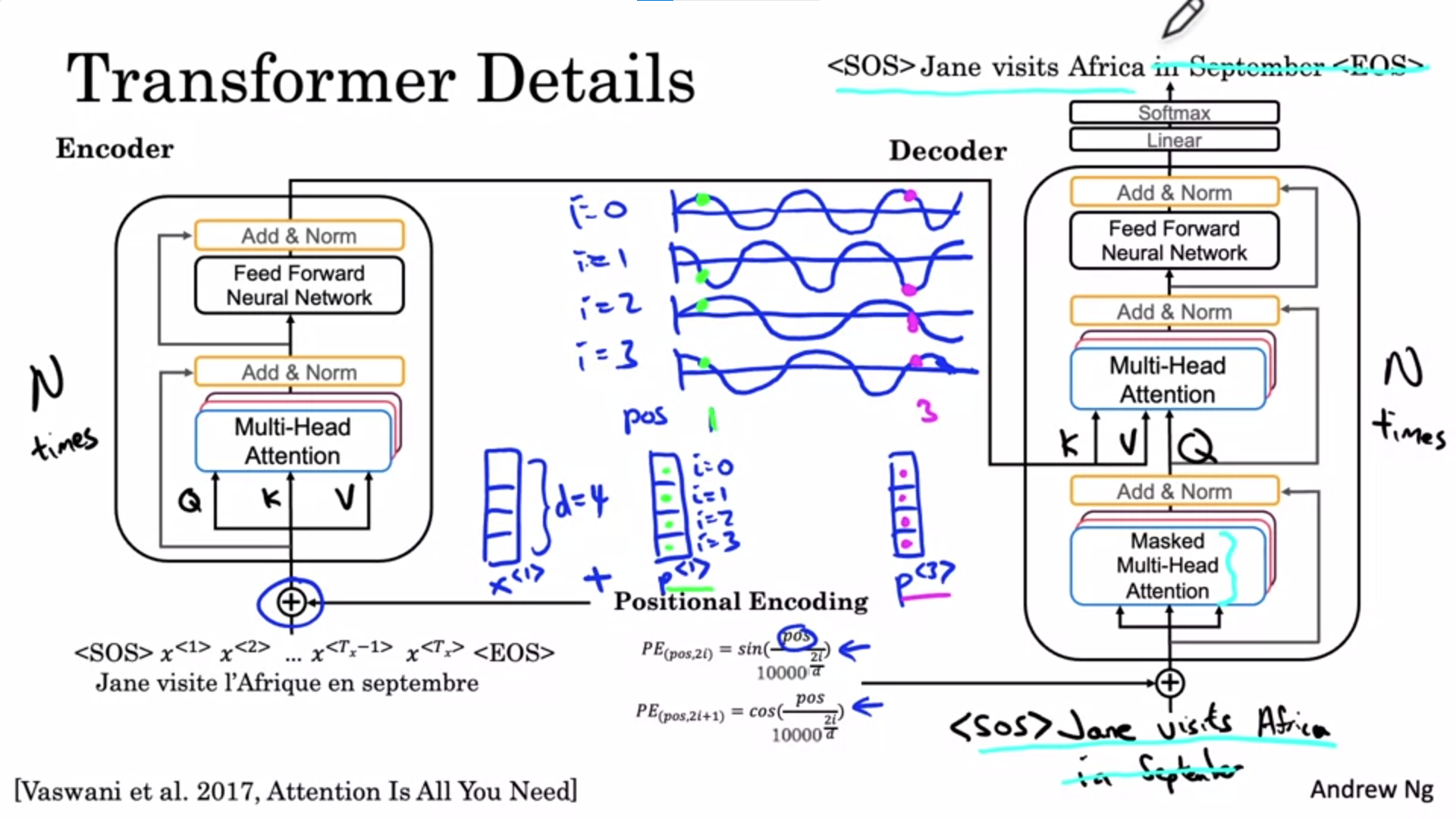
Recap: Multi-head attention mechanism -> doing self-attention a couple of times.

We can ask multiple questions for the same word and learn a much better richer representation for every word.

Now let’s put everything together to build the transformer network.



We learned about self-attention and multi-headed attention. Now we put it together to build the transformer network.



We start with the same sentence that we want to translate from French to English.

Up to this point for simplicity we have only talked about the embeddings for the words in the sentence. But in many sequence to sequence translation task it will be useful to add <SOS> and <EOS> token.

First part of the transformer is that the input sequence gets fed into an encoder block which has multi-head attention. Feed in the values Q, K, V computed from the embeddings and weight matrices W. This layer computes a matrix that is fed into a feed forward neural network. This helps to determine which interesting features there are in the sentence.

In the transformer paper this encoder block is repeated n times and a typical value for n is 6.

After 6 blocks the output of the encoder block is fed into a decoder block.

The decoder block’s goal is to output the translation.

The decoder block will input the first few words of the output (whatever we have already outputted as translation.

When we get started we know that the translation will start with a <SOS> token. So the <SOS> token gets fed into the multi-head attention block. This is the only component that will be used to calculate q,k and v for the decoder multi-head attention block.

The first block’s output will generate the Q matrix for the next multi-head attention block. The output of the encoder is used to generate k and v.

Why is it structured this way? Intuition: the input for the first multi-head attention block is whatever we have translated so far. So this will ask a query about the <SOS> and it will then pull context from k,v which are translated from the French version of the sentence to then decide what is the next word in that sequence to generate.

To finish, the second multi-head attention block’s output will be fed into a feed forward neural network. This decoder block is also going to be repeated n times. The job of this network is to predict the next word in the sentence. Hopefully it will decide that the first word in the English translation is “Jane”. We will then feed “Jane” as input into the decoder and so the next query will be asked by “<SOS> Jane” to find the most appropriate next word (hopefully “visits”). Then we run the neural network again and feed “visits”, which will hopefully generate “Africa” and so on.

These encoder and decoder blocks and how they are combined to perform a sequence to sequence translation task are the main ideas of the transformer architecture.

In this case we saw how we can translate an input sentence into a sentence in another language to gain intuition about how attention in neural networks can be combined to allow simultaneous computation.

Beyond these main ideas there are some more bills and whistles which make the transformer network even better. The first of which is positional encoding.

So far the position of the word in the sentence is not in the encoding but the position of the word can be extremely important for the translation. The way we encode the position of elements in the input is that we use a combination of sin and cos equations.

Let’s say for example that our word embedding x<1> is a vector with 4 values. The dimension d of the word embedding is 4. We then create a positional embedding vector of the same dimension p<1>.

In the equation of the positional encoding “pos” denotes the numerical position of the word. For the word “Jane” pos is equal to 1. i refers to the different dimensions of the encoding.

The positional encoding equation with the sin and cos creates a unique positional encoding vector for each word. p<3> will be different from p<1>. Sin and cos curves are shown on the slide.

i=1: matching cosine for i=0 sin -> 90 degrees out of phase.

i = 2: lower frequency sin.

I=3: matches cosine curve to i=2 sin.

Although some values might be the same or similar between the p<1> and p<3> vector, looking at all 4 values will make it unique.

The positional encoding p<1> is added directly to x<1> to reflect the position in the feature representation. Also pass them to the network with residual connections. The purpose of these is to pass along the position into the entire architecture.

In addition to positional encoding, the transformer also uses a layer called add & norm (playing a role similar to batch norm what we have seen before). It helps speed up training.

For the output of the decoder block there is also a linear and a softmax layer to predict the next word one word at a time.

When reading about the transformer literature we will also hear the term “masked multi-head attention”. Masked multi-head attention is only important for the training set. Previously we have stepped through to see how the transformer performs prediction one word at a time. But how does it train? Let’s say we have the correct French to English translation in our training set. So during training we have access to the entire correct English translation. We don’t have to generate the words one at a time during training. Instead what masking does is it blocks out the last part of the sentence to mimic what the network will have to do at test time / during prediction. So it repeatedly pretends that the network had perfectly translated the first few words and hides the remaining words (marked in teal).

Recap: Now we have a good sense of all major blocks of the transformer network.