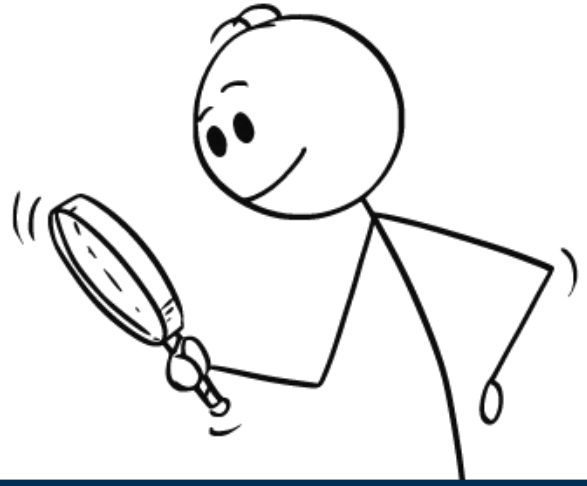


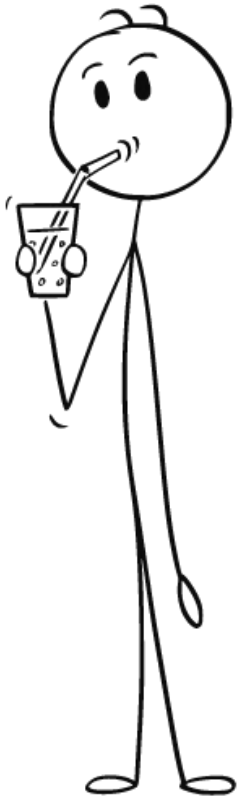
CSE 4/535
Information Retrieval

Sayantan Pal
PhD Student, Department of CSE
338Z Davis Hall

University at Buffalo
Department of CSE

Before we start

1. Please keep your VMs running for P2
2. It is expected that you have already started working on P3
3. No class next to next Wednesday (Nov 30th) Timeline will be updated
4. We discussed word Embeddings
    a. how text is converted to numbers



What are we learning today?

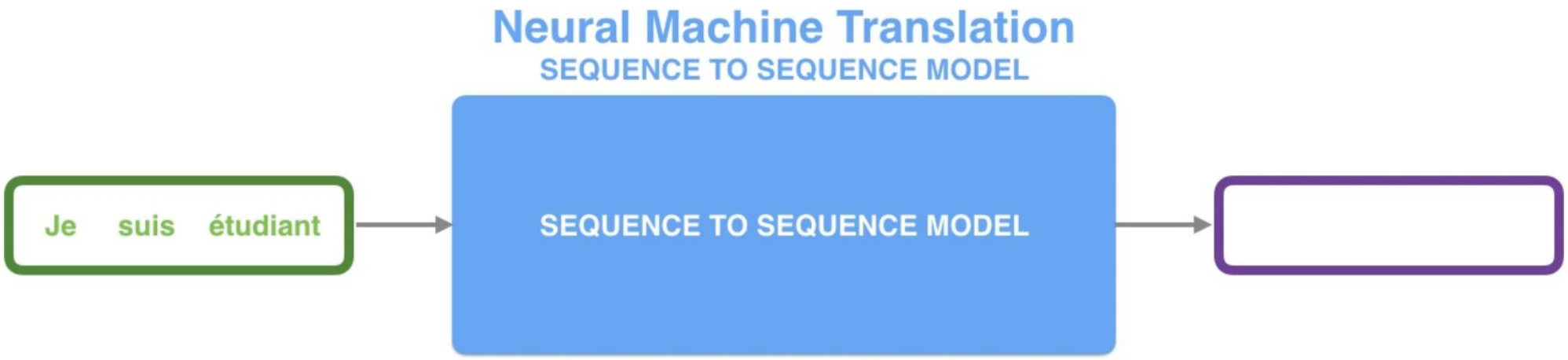
1. Seq2seq and Transformers - Credits to Jay Alammar
    a. What is RNN? Why LSTM?
    b. What is seq2seq models?
    c. What is Transformer (high level overview of self-attention)?
    d. Hugging Face for P3
        i. Classification Models
        ii. Generation Models

Sequence-to-sequence (Seq2seq)

A sequence-to-sequence model is a model that takes a sequence of items (words, letters, features of an images…etc) and outputs another sequence of items. A trained model would work like this:

Neural machine translation (NMT)

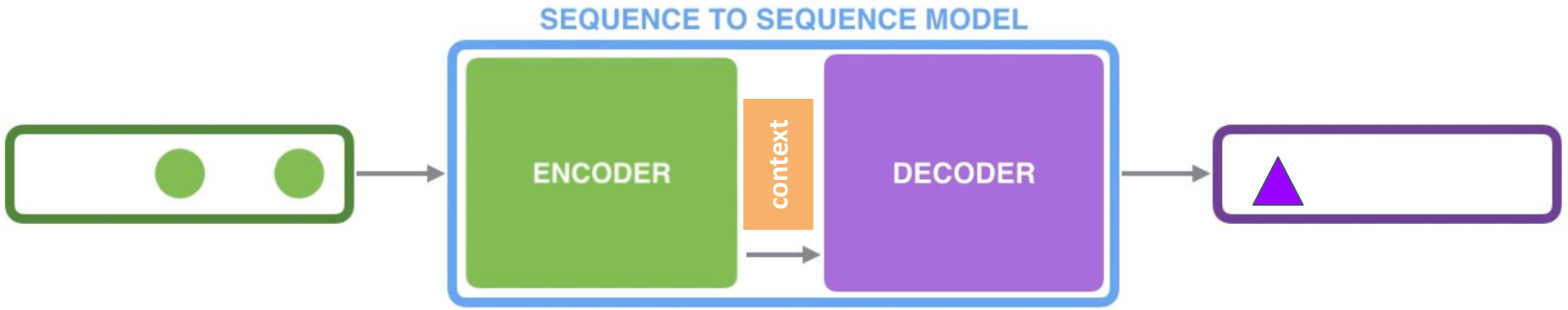
In neural machine translation, a sequence is a series of words, processed one after another. The output is, likewise, a series of words:



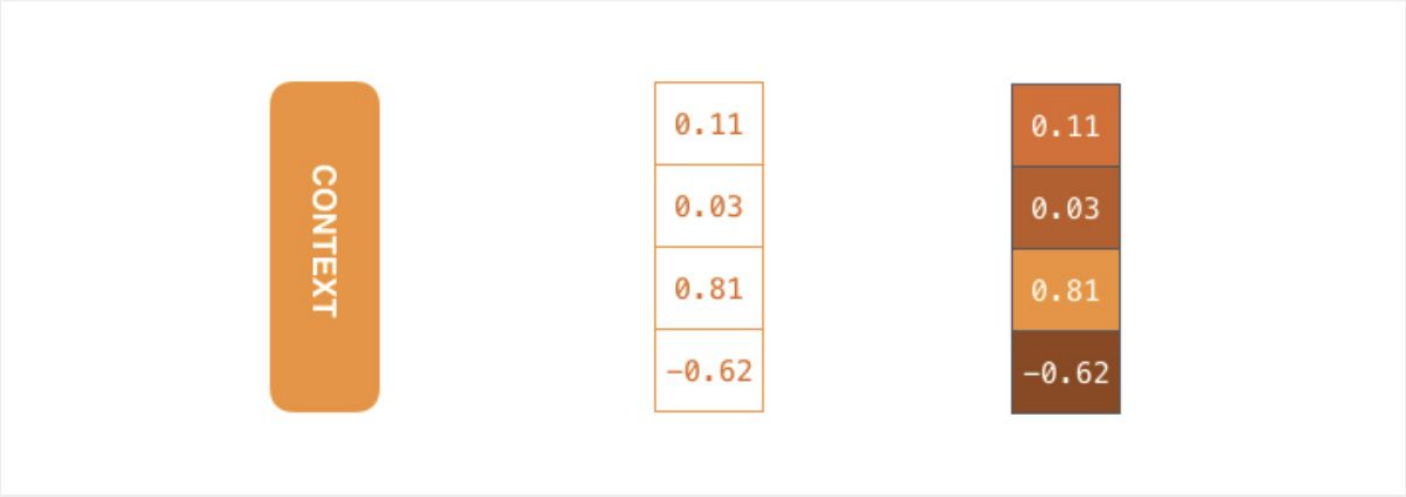
Looking under the hood

Under the hood, the model is composed of an encoder and a decoder.

The encoder processes each item in the input sequence, it compiles the information it captures into a vector (called the context). After processing the entire input sequence, the encoder sends the context over to the decoder, which begins producing the output sequence item by item.



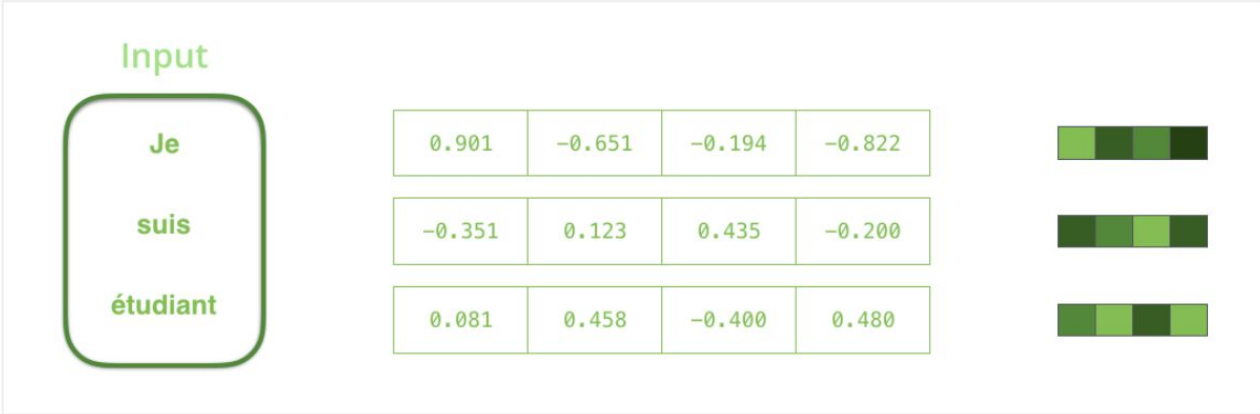
The context is a vector (an array of numbers, basically) in the case of machine translation. The encoder and decoder tend to both be recurrent neural networks (Be sure to check out Luis Serrano's A friendly introduction to Recurrent Neural Networks for an intro to RNNs).



The context is a vector of floats. Later in this post we will visualize vectors in color by assigning brighter colors to the cells with higher values.

You can set the size of the context vector when you set up your model. It is basically the number of hidden units in the encoder RNN. These visualizations show a vector of size 4, but in real world applications the context vector would be of a size like 256, 512, or 1024.

By design, a RNN takes two inputs at each time step: an input (in the case of the encoder, one word from the input sentence), and a hidden state. The word, however, needs to be represented by a vector. To transform a word into a vector, we turn to the class of methods called “word embedding” algorithms. These turn words into vector spaces that capture a lot of the meaning/semantic information of the words (e.g. king - man + woman = queen).



We need to turn the input words into vectors before processing them. That transformation is done using a word embedding algorithm. We can use pre-trained embeddings or train our own embedding on our dataset. Embedding vectors of size 200 or 300 are typical, we're showing a vector of size four for simplicity.

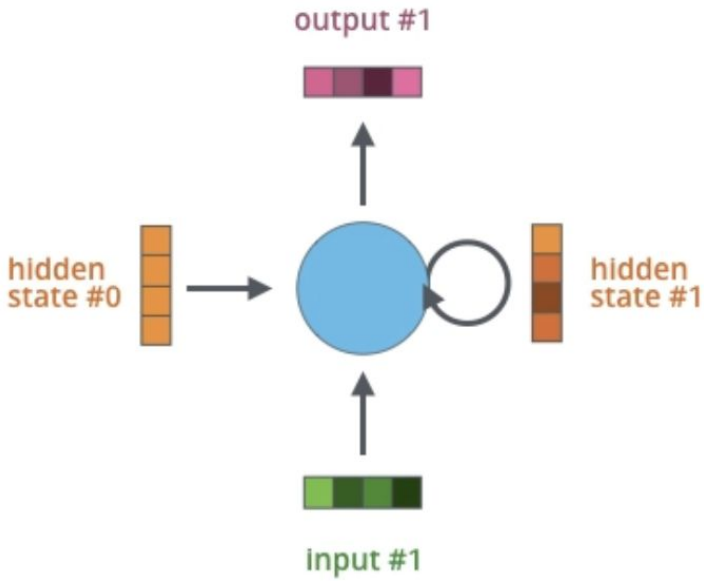
Now that we’ve introduced our main vectors/tensors, let’s recap the mechanics of an RNN and establish a visual language to describe these models:

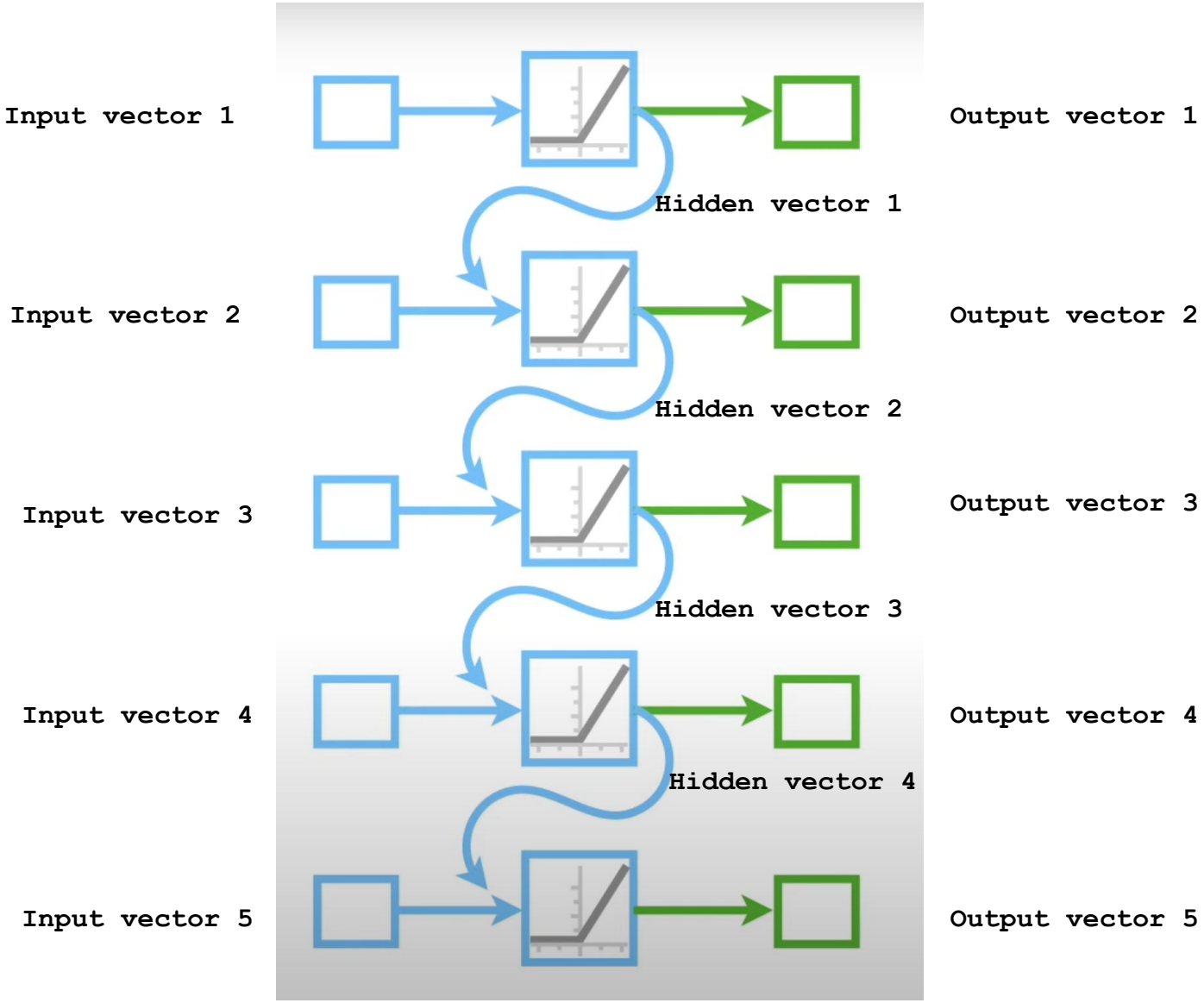
Recurrent Neural Network

Time step #1:
An RNN takes two input vectors:

Processes them

Then produces two output vectors:




Input vector 1
Output vector 1
Hidden vector 1
Input vector 2
Output vector 2
Hidden vector 2
Input vector 3
Output vector 3
Hidden vector 3
Input vector 4
Output vector 4
Hidden vector 4
Input vector 5
Output vector 5

Problems?

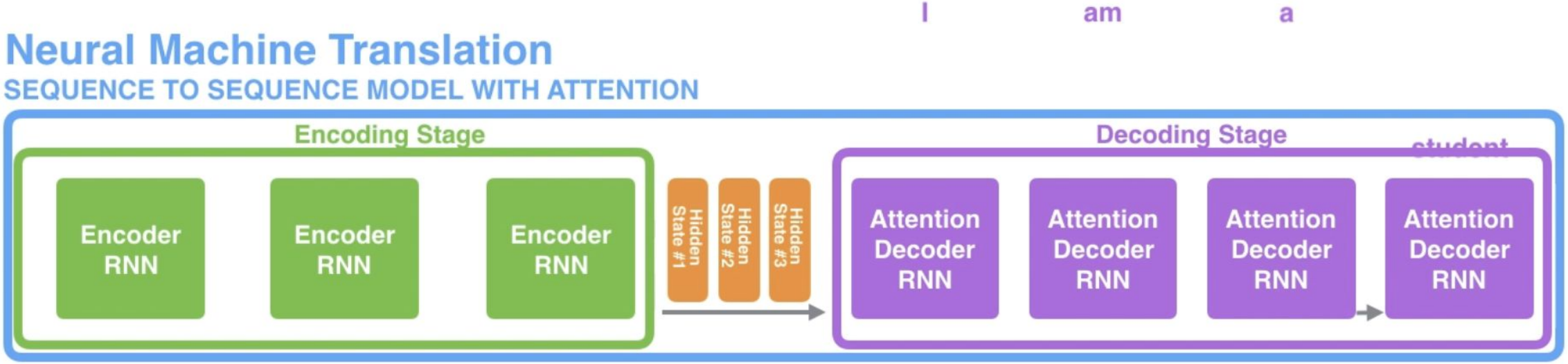
1. Vanishing Gradient
2. Exploding Gradient

Let's Pay Attention Now

The context vector turned out to be a bottleneck for these types of models. It made it challenging for the models to deal with long sentences. A solution was proposed in Bahdanau et al., 2014 and Luong et al., 2015. These papers introduced and refined a technique called "Attention", which highly improved the quality of machine translation systems. Attention allows the model to focus on the relevant parts of the input sequence as needed.

Let's continue looking at attention models at this high level of abstraction. An attention model differs from a classic sequence-to-sequence model in two main ways:

First, the encoder passes a lot more data to the decoder. Instead of passing the last hidden state of the encoding stage, the encoder passes *all* the hidden states to the decoder:



Second, an attention decoder does an extra step before producing its output. In order to focus on the parts of the input that are relevant to this decoding time step, the decoder does the following:

1. Look at the set of encoder hidden states it received – each encoder hidden state is most associated with a certain word in the input sentence
2. Give each hidden state a score (let's ignore how the scoring is done for now)
3. Multiply each hidden state by its softmaxed score, thus amplifying hidden states with high scores, and drowning out hidden states with low scores

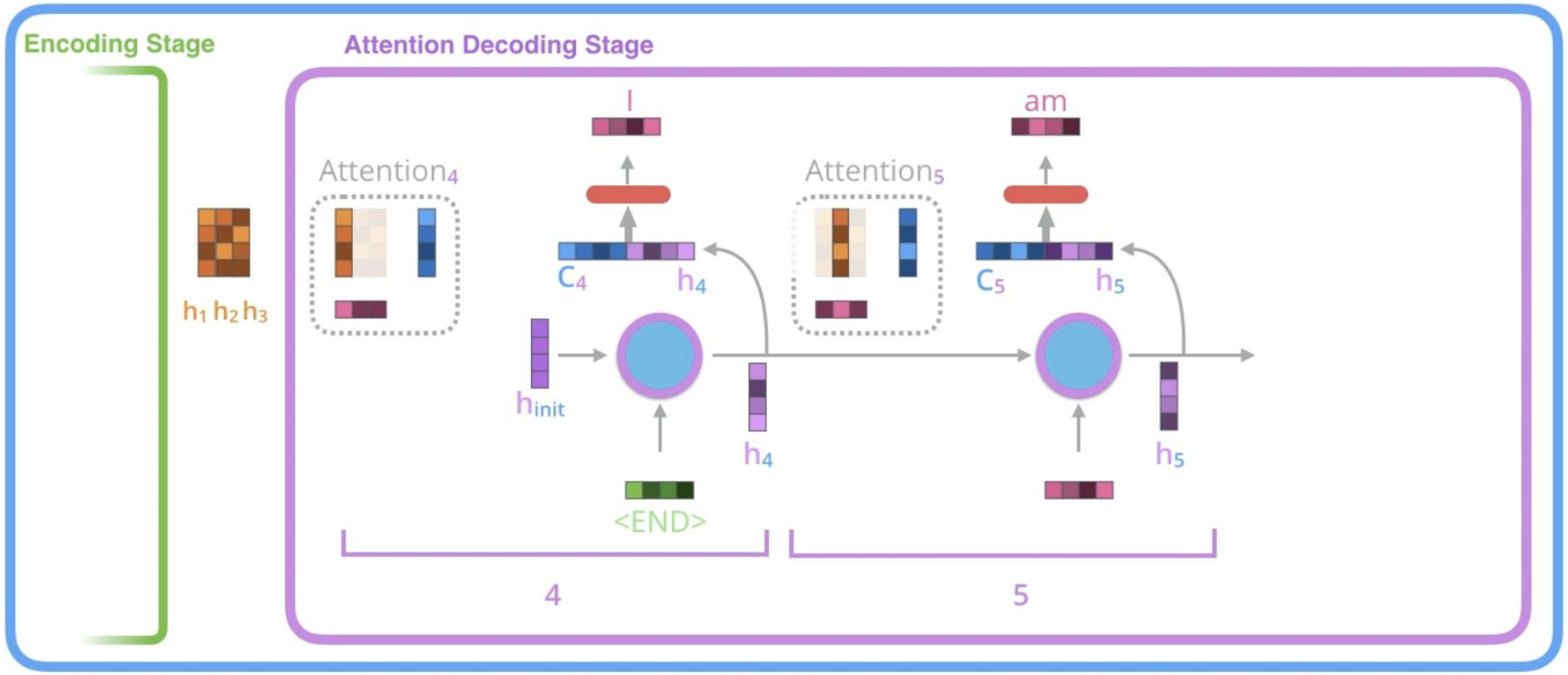

Attention at time step 4
1. Prepare inputs
h1
h2
h3
Encoder hidden states
Decoder hidden state at time step 4
2. Score each hidden state
13
9
9
scores
Attention weights for decoder time step #4
3. Softmax the scores
0.96
0.02
0.02
softmax scores
4. Multiply each vector by its softmaxed score
+
+
=
5. Sum up the weighted vectors
Context vector for decoder time step #4

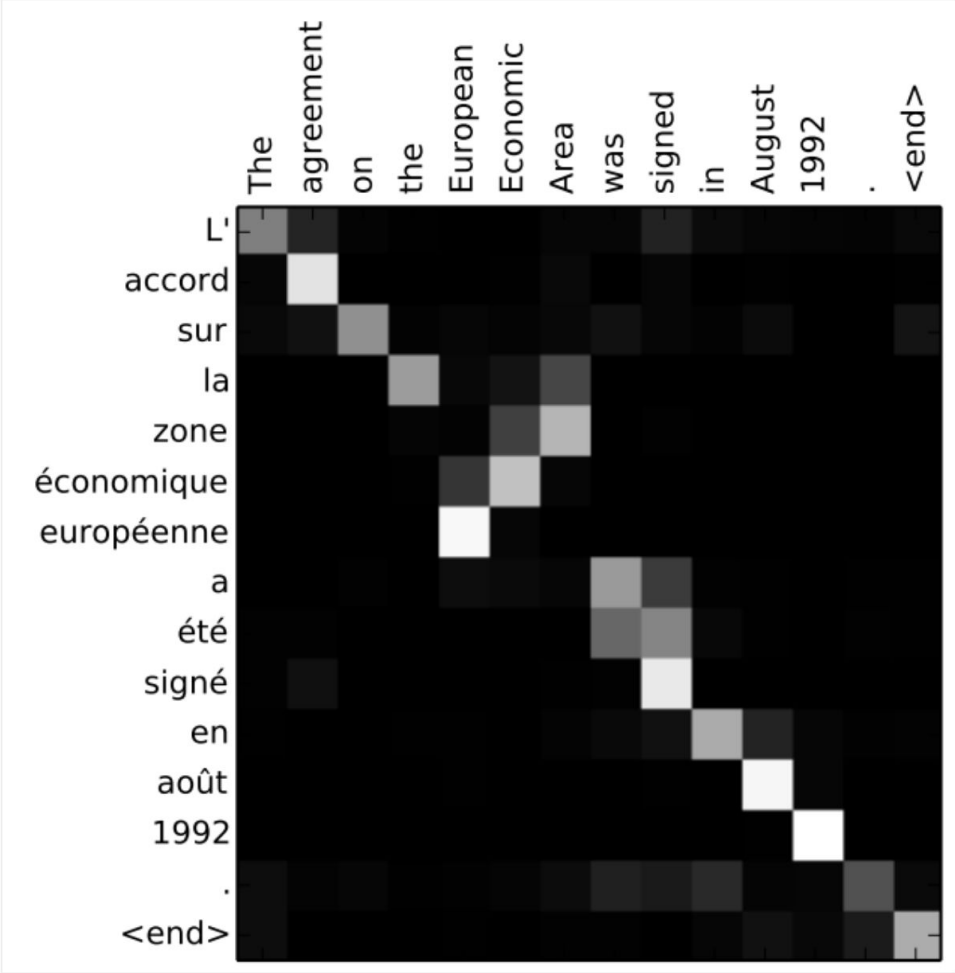
Let us now bring the whole thing together in the following visualization and look at how the attention process works:

1. The attention decoder RNN takes in the embedding of the <END> token, and an initial decoder hidden state.
2. The RNN processes its inputs, producing an output and a new hidden state vector (h4). The output is discarded.
3. Attention Step: We use the encoder hidden states and the h4 vector to calculate a context vector (C4) for this time step.
4. We concatenate h4 and C4 into one vector.
5. We pass this vector through a feedforward neural network (one trained jointly with the model).
6. The output of the feedforward neural networks indicates the output word of this time step.
7. Repeat for the next time steps

Neural Machine Translation

SEQUENCE TO SEQUENCE MODEL WITH ATTENTION





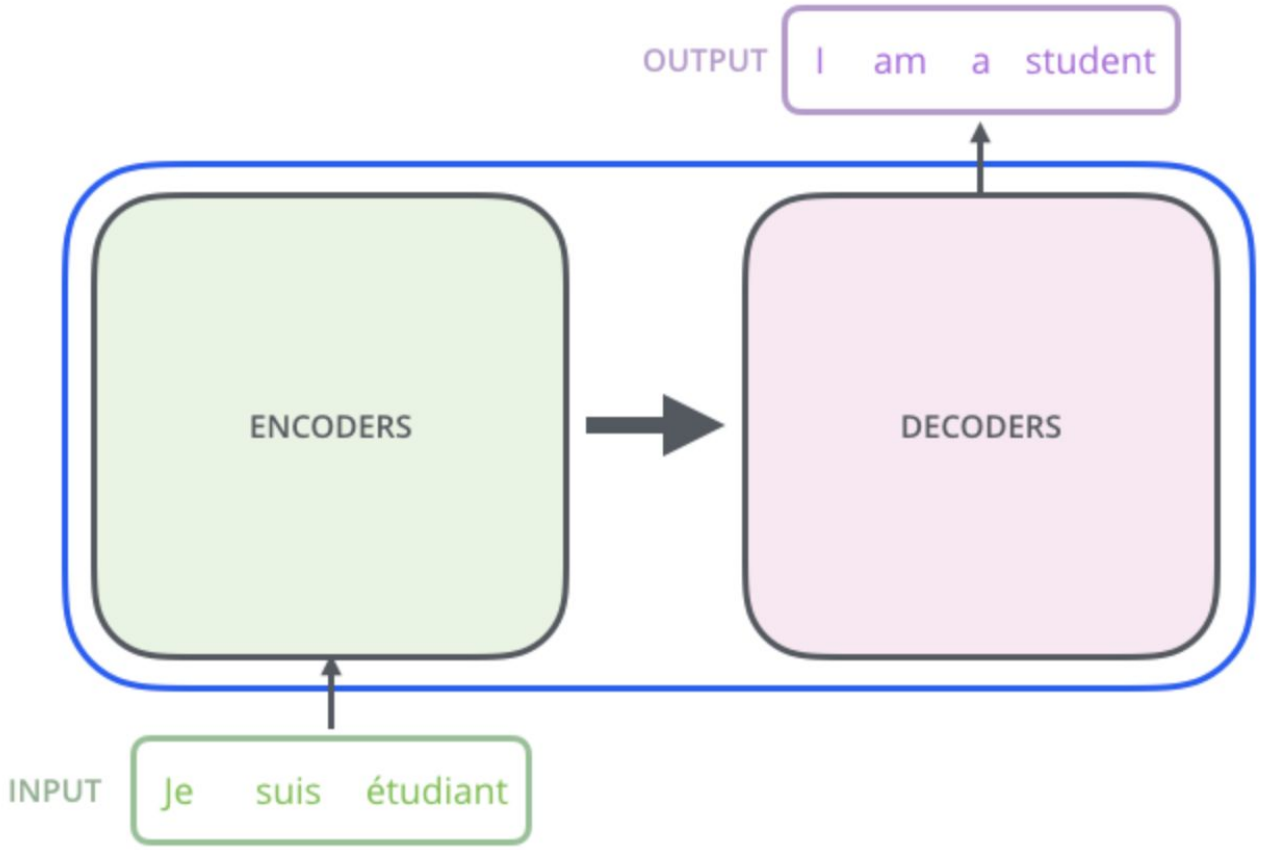
You can see how the model paid attention correctly when outputing "European Economic Area". In French, the order of these words is reversed ("européenne économique zone") as compared to English. Every other word in the sentence is in similar order.

Transformers

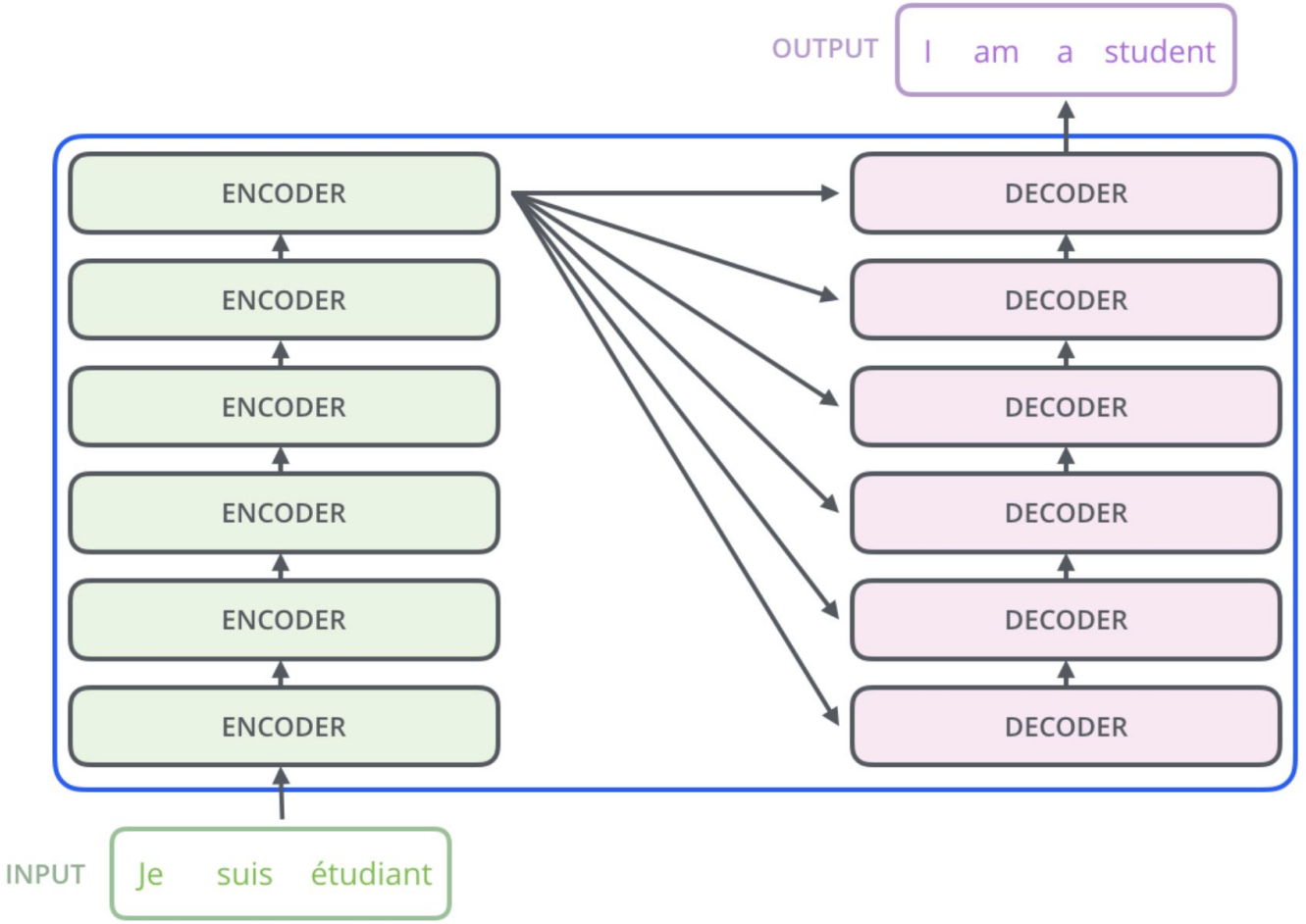
A High-Level Look

Let's begin by looking at the model as a single black box. In a machine translation application, it would take a sentence in one language, and output its translation in another.

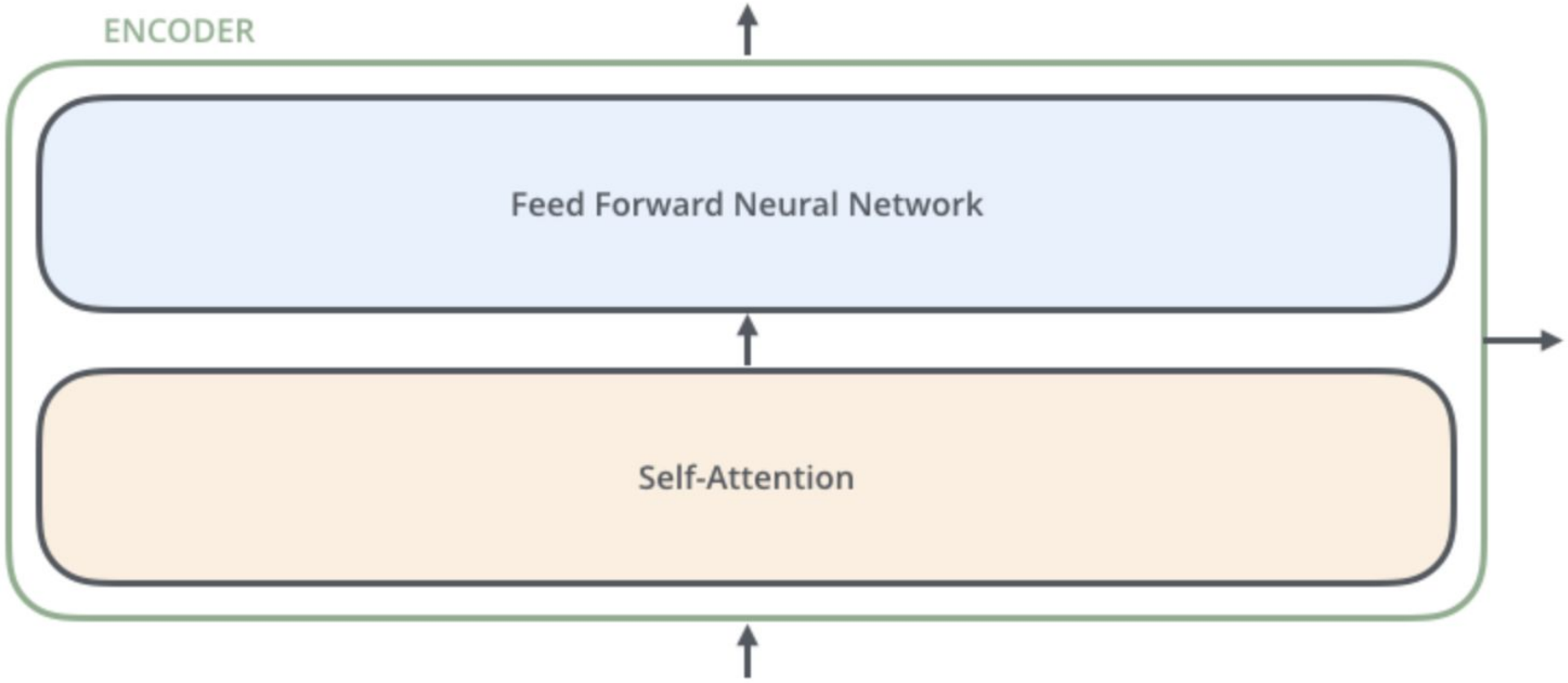
Popping open that Optimus Prime goodness, we see an encoding component, a decoding component, and connections between them.



The encoding component is a stack of encoders (the paper stacks six of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements). The decoding component is a stack of decoders of the same number.



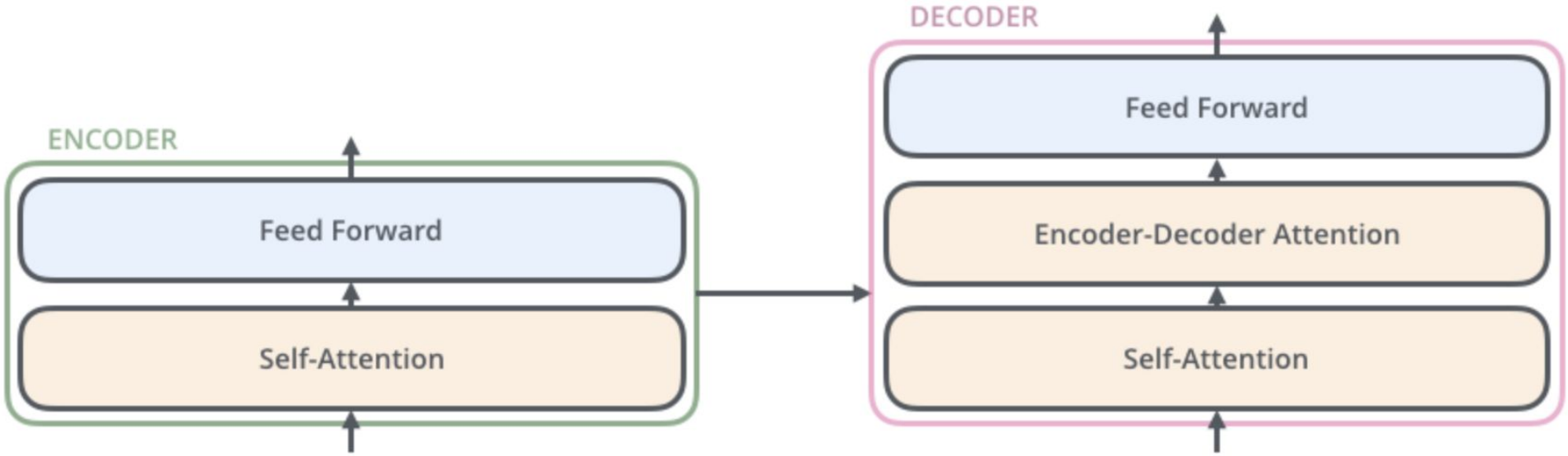
The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:



The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. We'll look closer at self-attention later in the post.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

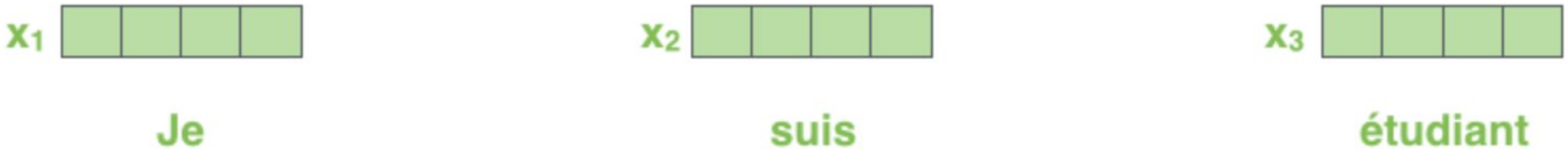
The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in seq2seq models).



Bringing The Tensors Into The Picture

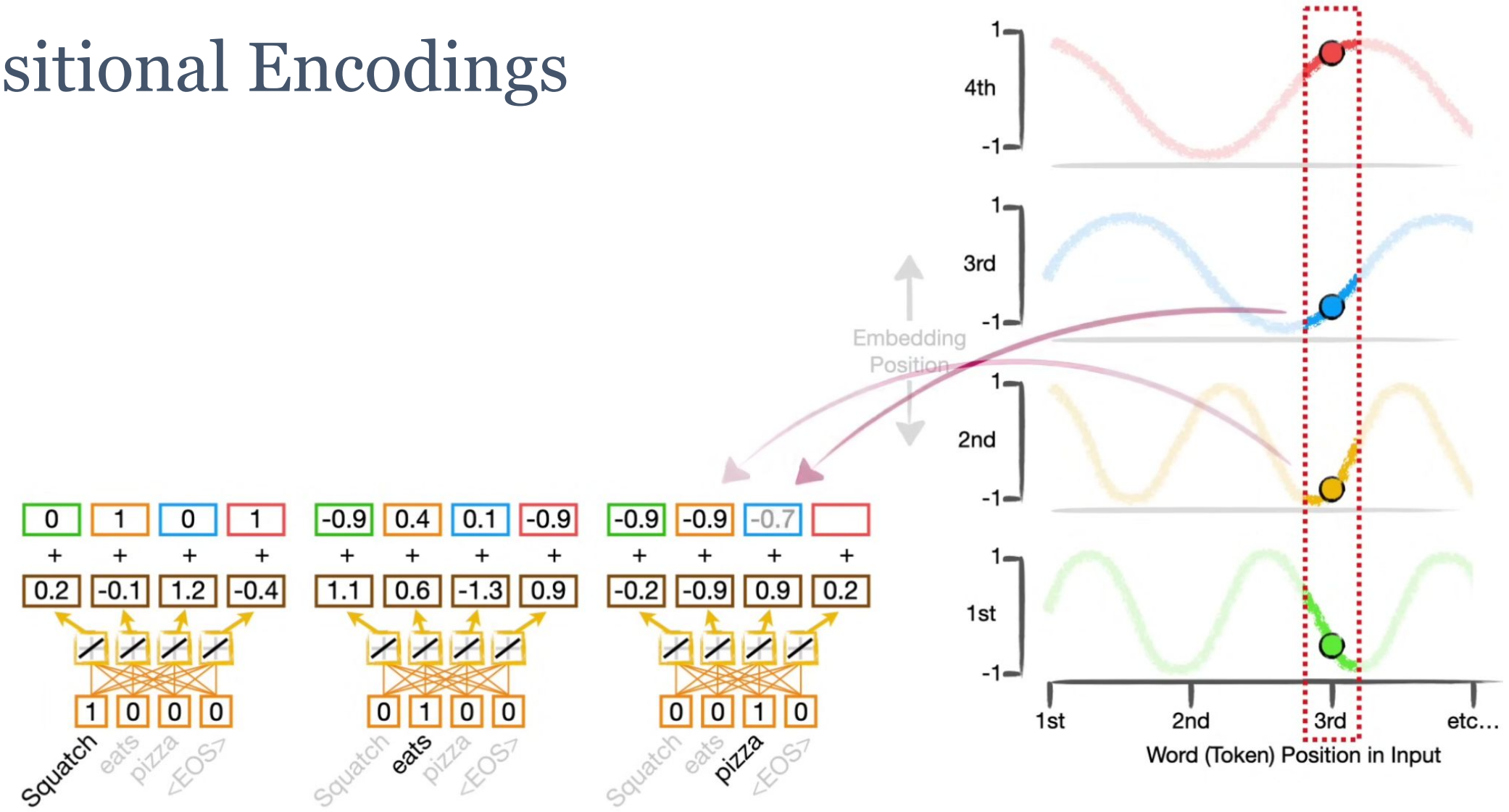
Now that we've seen the major components of the model, let's start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output.

As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm.

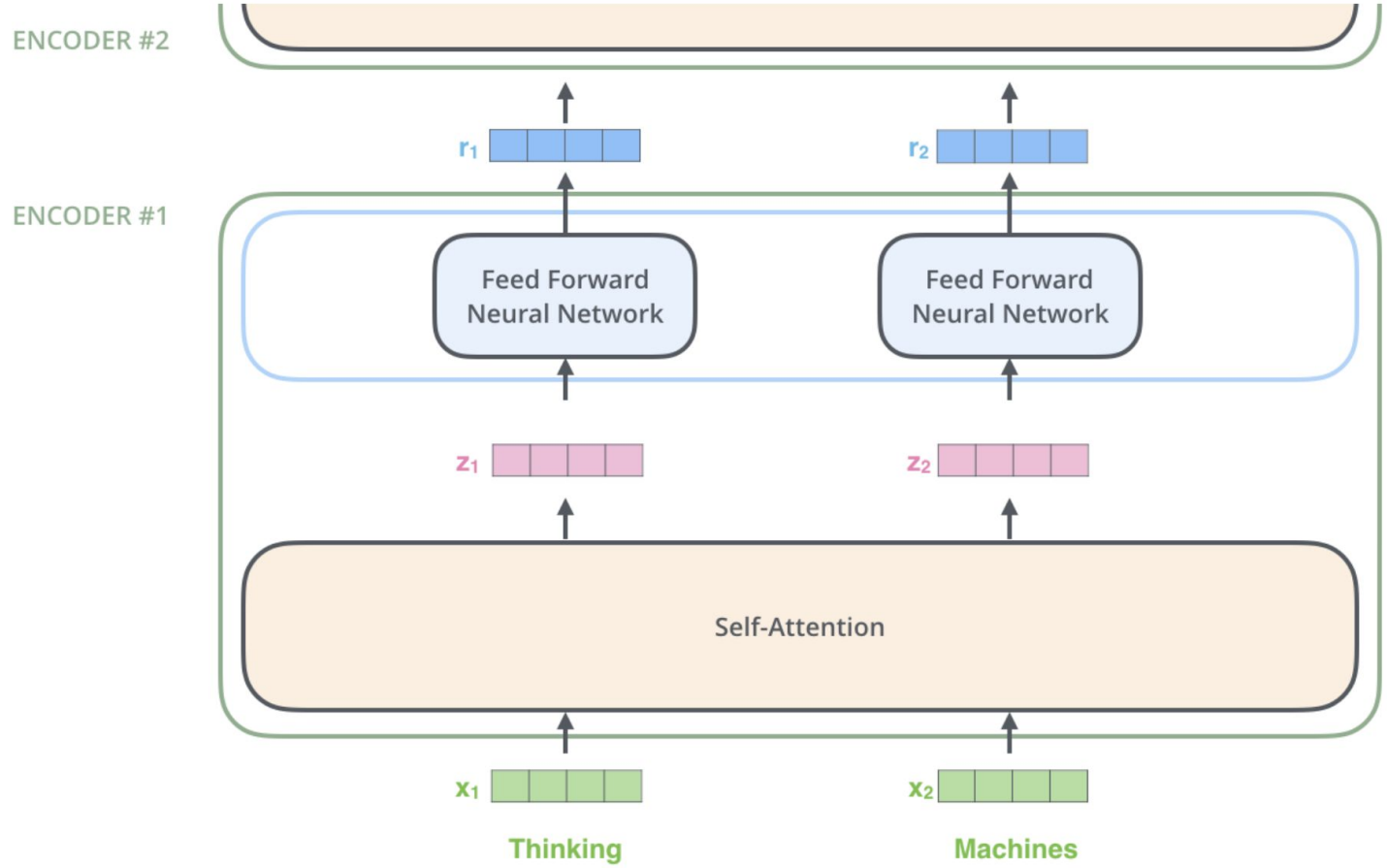


Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

Positional Encodings



As we've mentioned already, an encoder receives a list of vectors as input. It processes this list by passing these vectors into a 'self-attention' layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.



The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.

Self-Attention at a High Level

Don't be fooled by me throwing around the word "self-attention" like it's a concept everyone should be familiar with. I had personally never came across the concept until reading the Attention is All You Need paper. Let us distill how it works.

Say the following sentence is an input sentence we want to translate:

"`The animal didn't cross the street because it was too tired`"

What does "it" in this sentence refer to? Is it referring to the street or to the animal? It's a simple question to a human, but not as simple to an algorithm.

When the model is processing the word "it", self-attention allows it to associate "it" with "animal".

As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

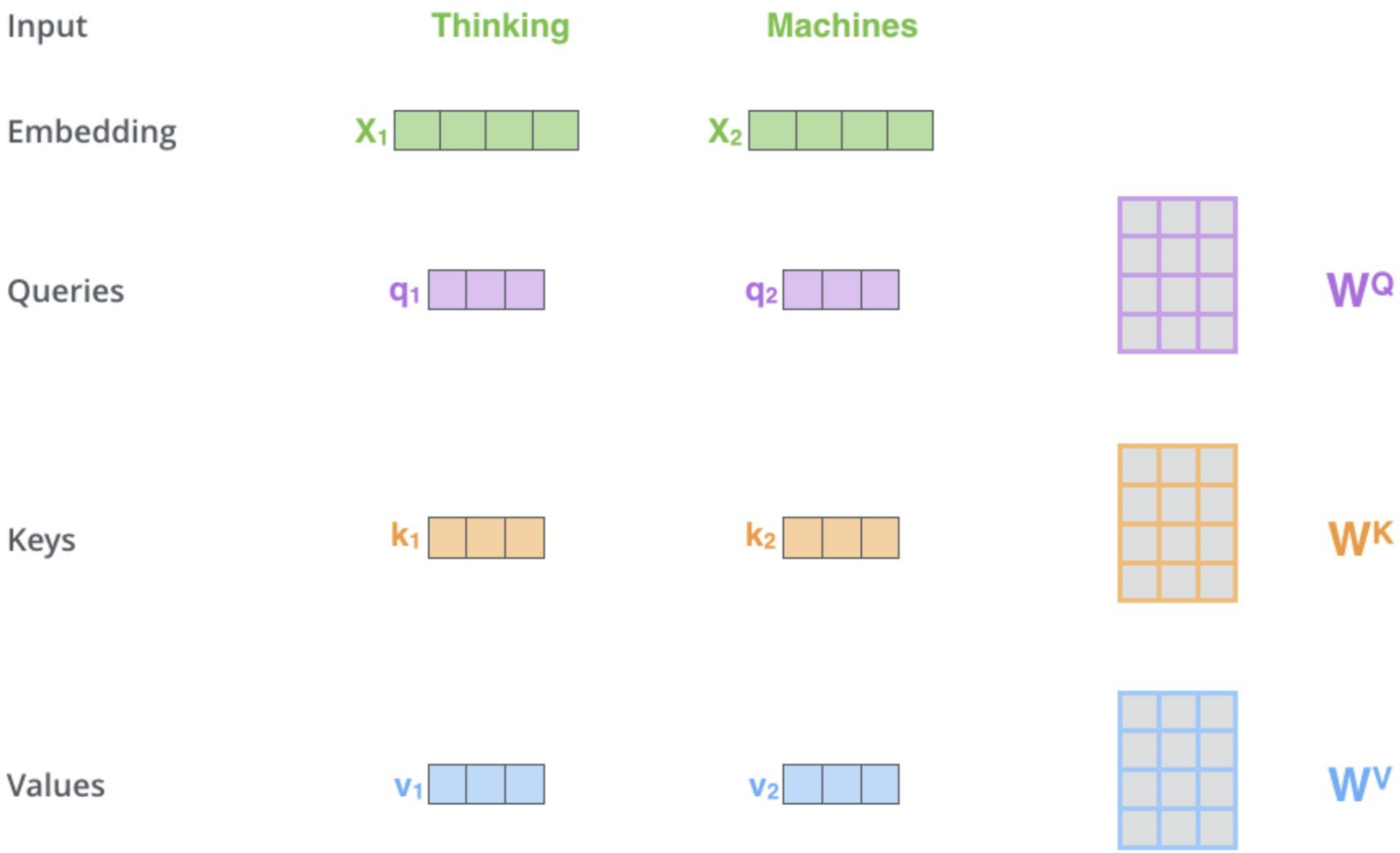
If you're familiar with RNNs, think of how maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it's processing. Self-attention is the method the Transformer uses to bake the "understanding" of other relevant words into the one we're currently processing.

Self-Attention in Detail

Let's first look at how to calculate self-attention using vectors, then proceed to look at how it's actually implemented – using matrices.

The **first step** in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512. They don't HAVE to be smaller, this is an architecture choice to make the computation of multiheaded attention (mostly) constant.

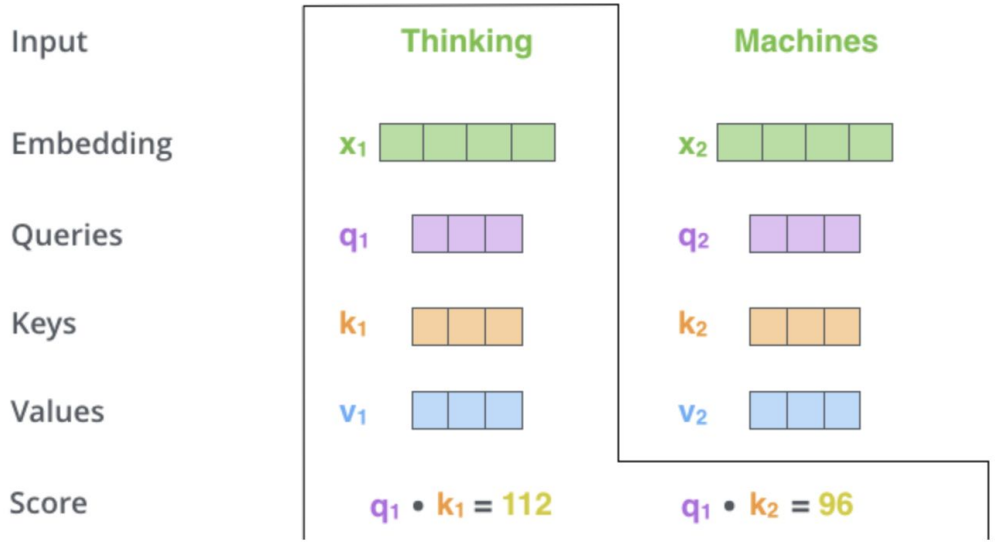

Input
Thinking
Machines
Embedding
X_1
X_2
Queries
q_1
q_2
W^Q
Keys
k_1
k_2
W^K
Values
v_1
v_2
W^V

What are the “query”, “key”, and “value” vectors?

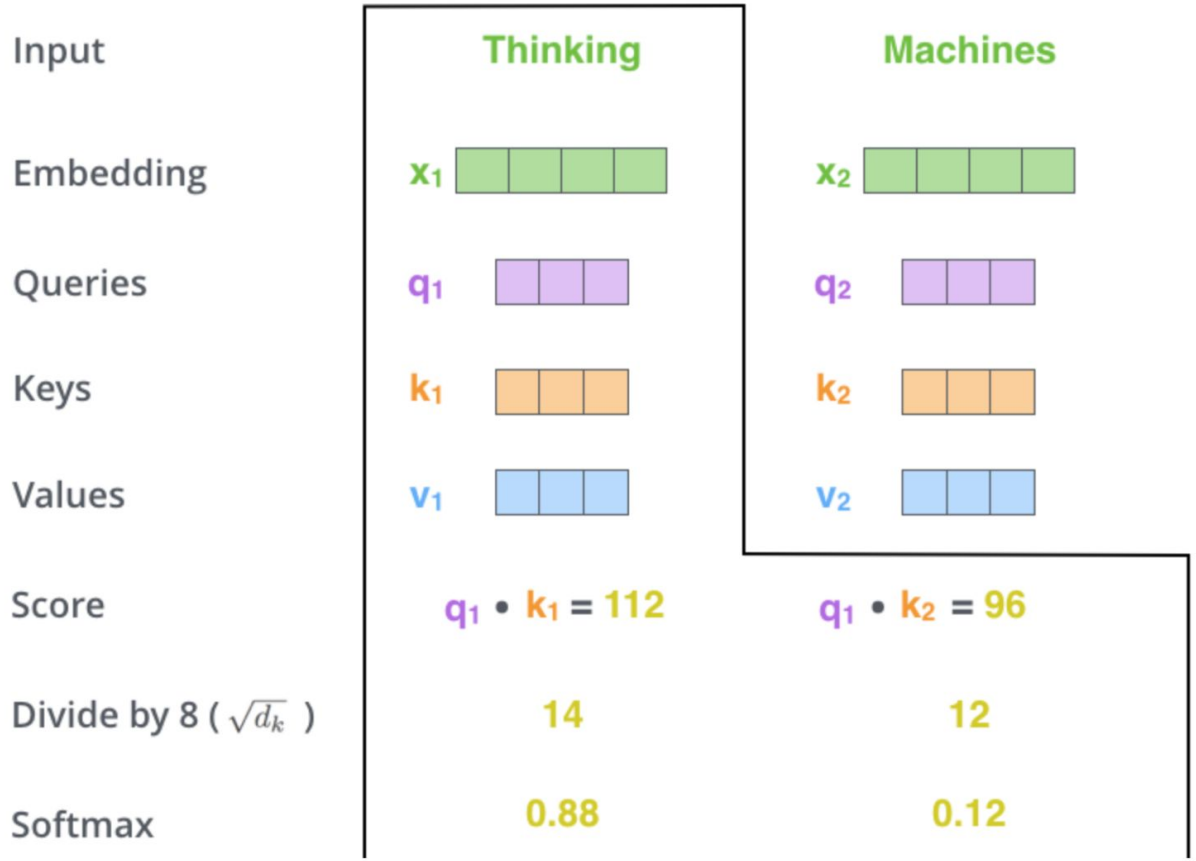
They’re abstractions that are useful for calculating and thinking about attention. Once you proceed with reading how attention is calculated below, you’ll know pretty much all you need to know about the role each of these vectors plays.

The **second step** in calculating self-attention is to calculate a score. Say we’re calculating the self-attention for the first word in this example, “Thinking”. We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we’re scoring. So if we’re processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.

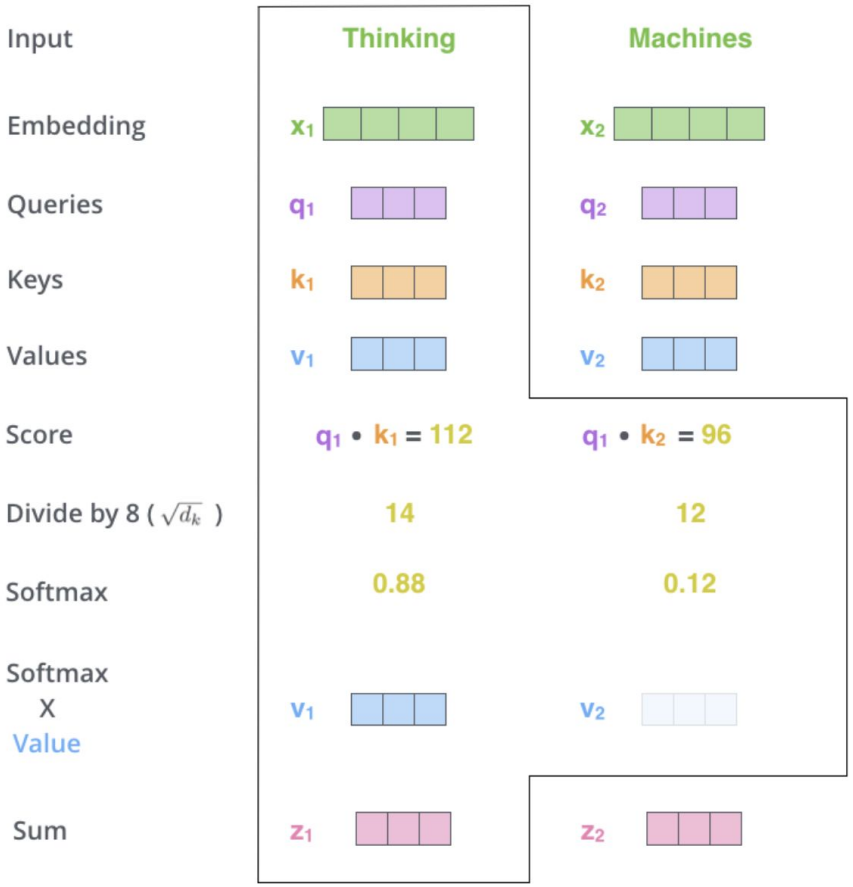


The **third and fourth steps** are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.



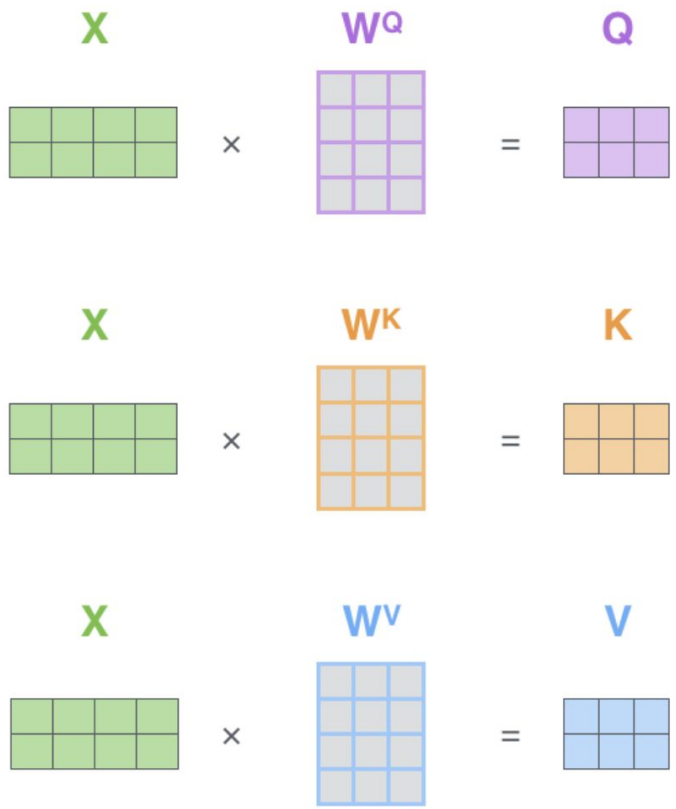
The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

The **sixth step** is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).



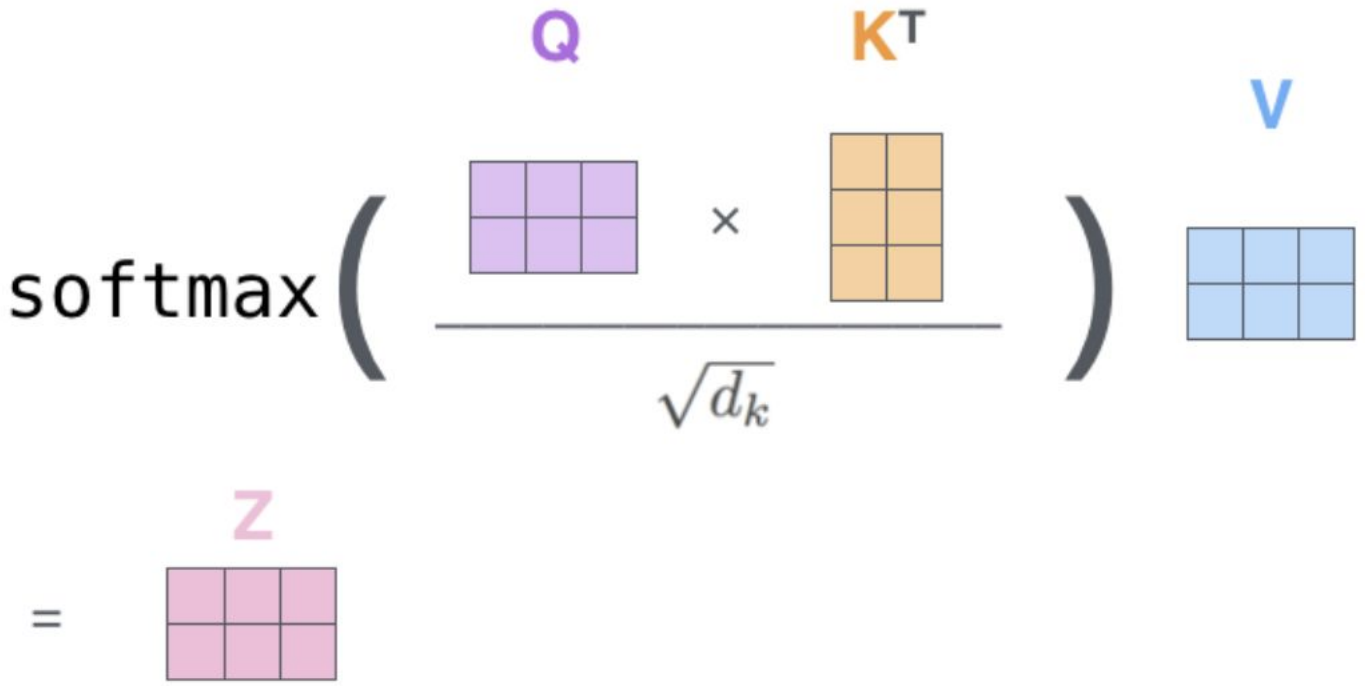
Matrix Calculation of Self-Attention

The first step is to calculate the Query, Key, and Value matrices. We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (WQ, WK, WV).



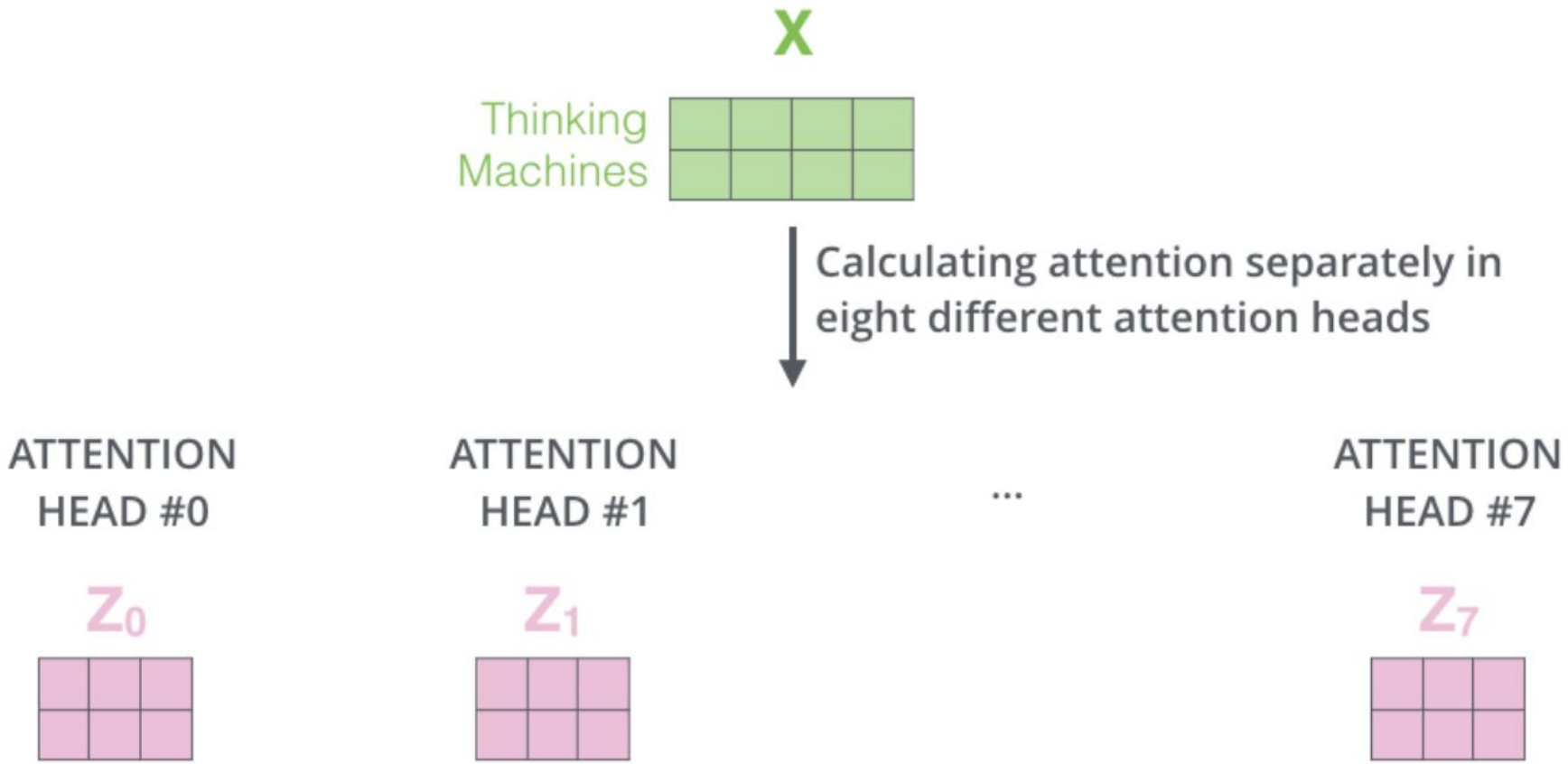
Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.



The self-attention calculation in matrix form

If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices



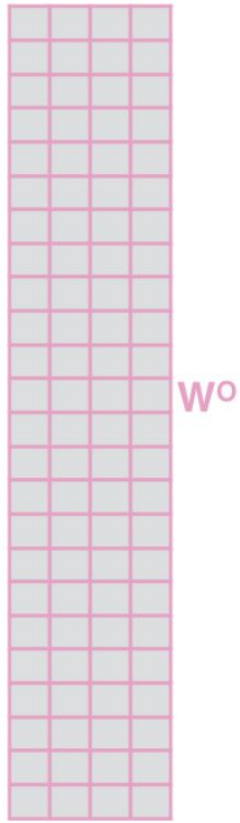
This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.

How do we do that? We concat the matrices then multiply them by an additional weights matrix WO.

1) Concatenate all the attention heads

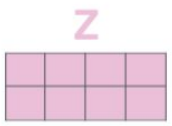
2) Multiply with a weight matrix W^O that was trained jointly with the model

X



3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

=



Why Multi-head Attention?

Example: The bank clerk helped me when the river bank was flooding.

Single Head Attention: The model processes this sentence with a single focus. It might learn to pay attention to the immediate context of each word. For example, it might associate "clerk" with the first "bank" and "river" with the second "bank". However, this single focus might not capture all the nuances. It might only focus on immediate word pairs and miss broader context or different kinds of relationships between words.

Multi-Head Attention Advantage

- Head 1 might focus on word meanings: It pays attention to "clerk" when processing the first "bank", understanding it as a financial institution.
- Head 2 might focus on sentence structure: It sees that "the river bank" is a phrase and understands this second "bank" as part of a geographical feature.
- Head 3 might focus on verb-object relationships: It connects "helped" more strongly with "me" and "was flooding" with "river bank".
- Other Heads: They might focus on other aspects, like past vs. present tense, noun phrases, etc.

References

1. https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/
2. https://jalammar.github.io/illustrated-transformer/
3. https://colab.research.google.com/github/jalammar/jalammar.github.io/blob/master/notebooks/Simple_Transformer_Language_Model.ipynb
4. RNN: https://www.youtube.com/watch?v=AsNTP8Kwu80
5. LSTM: https://www.youtube.com/watch?v=YCzL96nL7j0
6. Transformers: https://www.youtube.com/watch?v=zxQyTK8quyY