Deep Learning

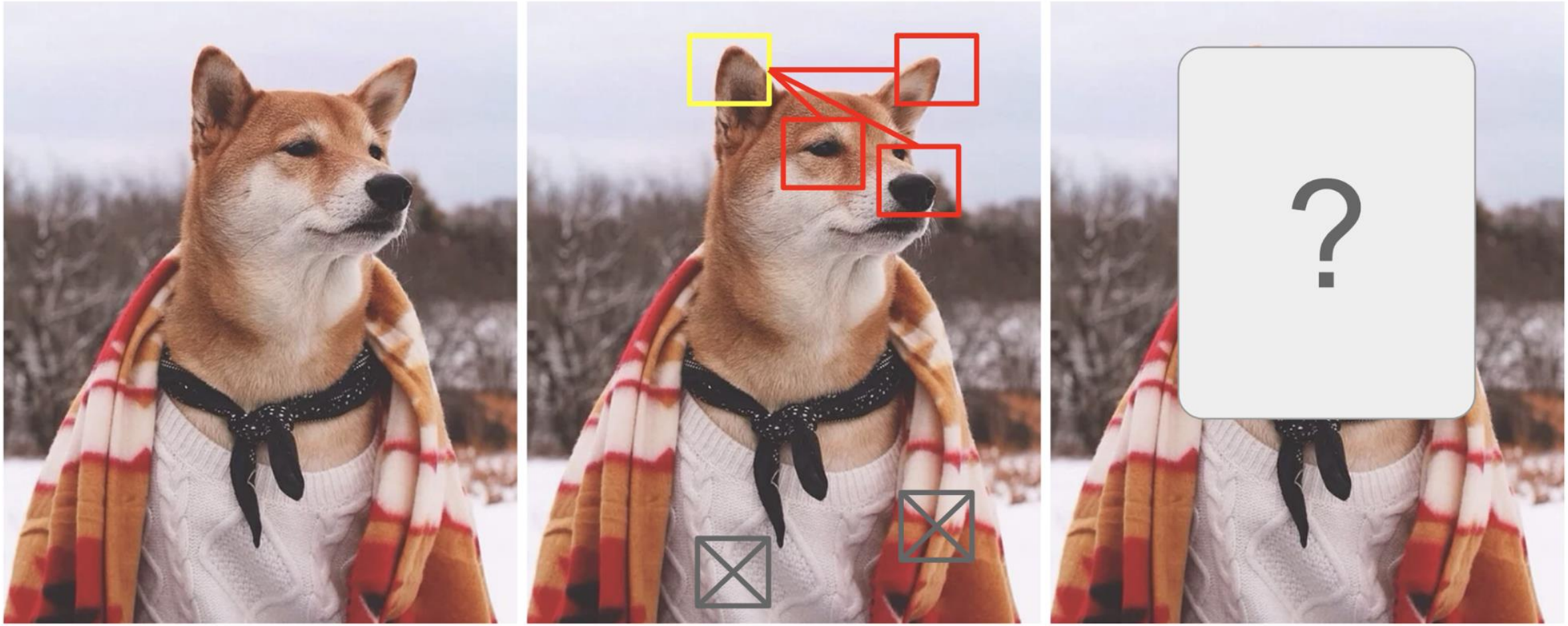
745.

Neural Attention and Self-Attention

Issues with RNN/LSTM seq2seq

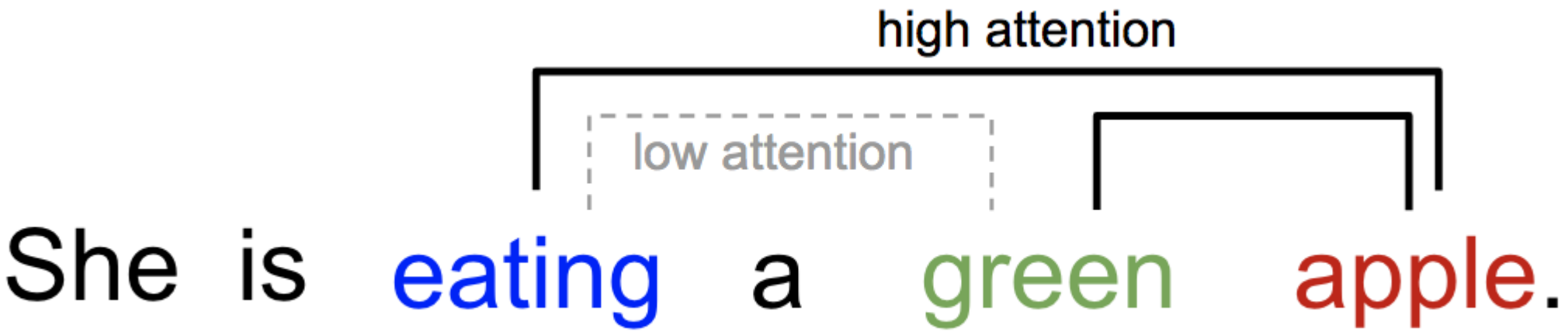
- Repeating text in the output
- Forgetting words at a distance
- Forgetting what words were already translated
- Dealing with unknown words

Visual attention



https://www.instagram.com/mensweardog/?hl=en

Textual Attention



Awesome tutorial by Jay Alammar

Jay Alammar

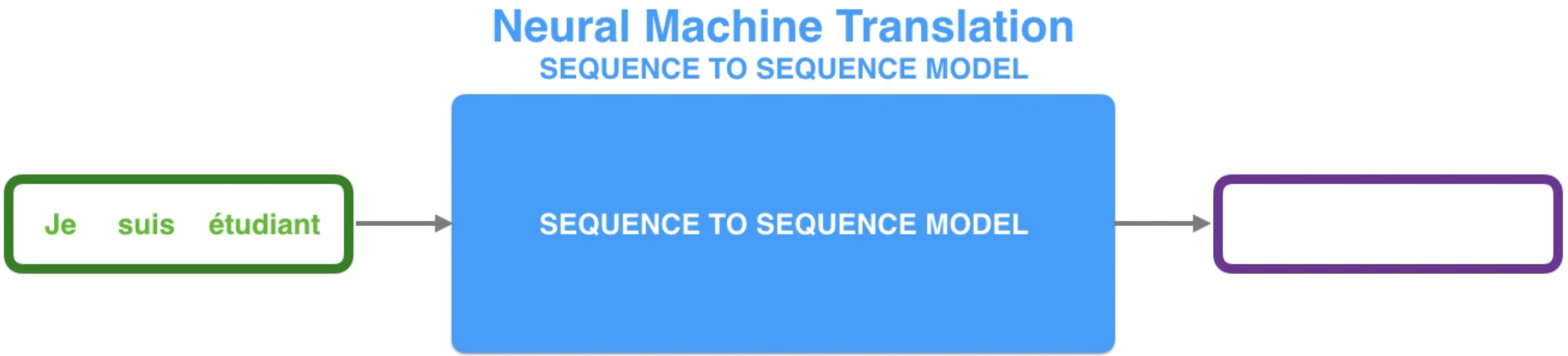
Visualizing machine learning one concept at a time

Blog About

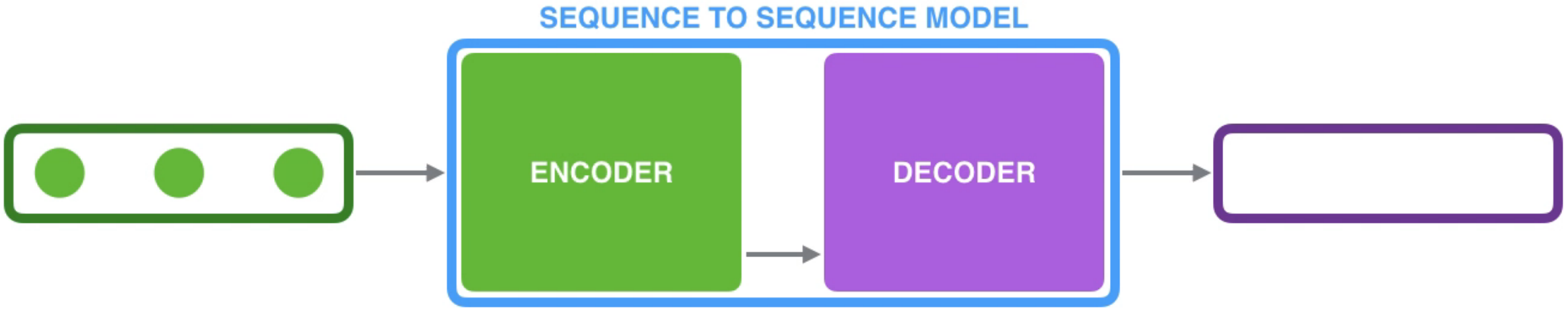
Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)

Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

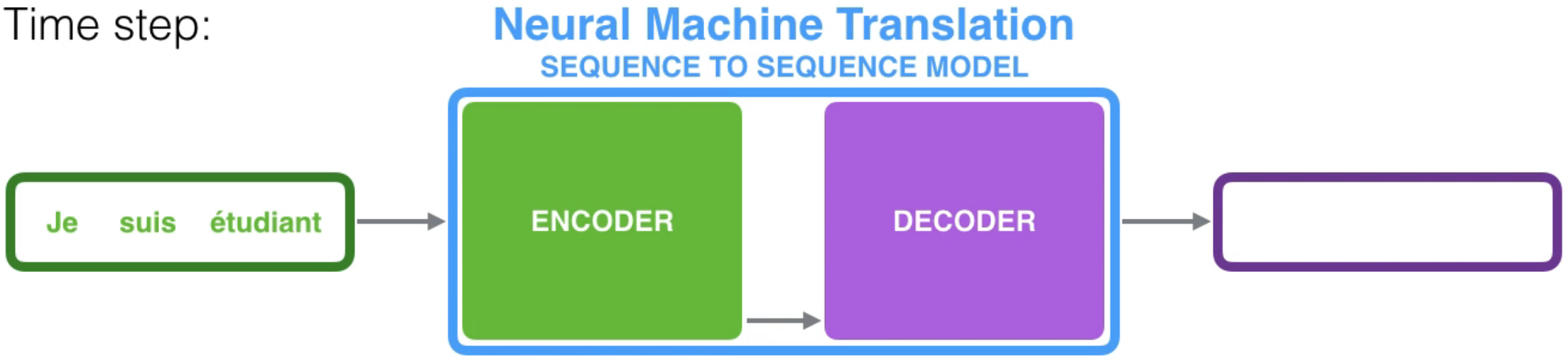
Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/



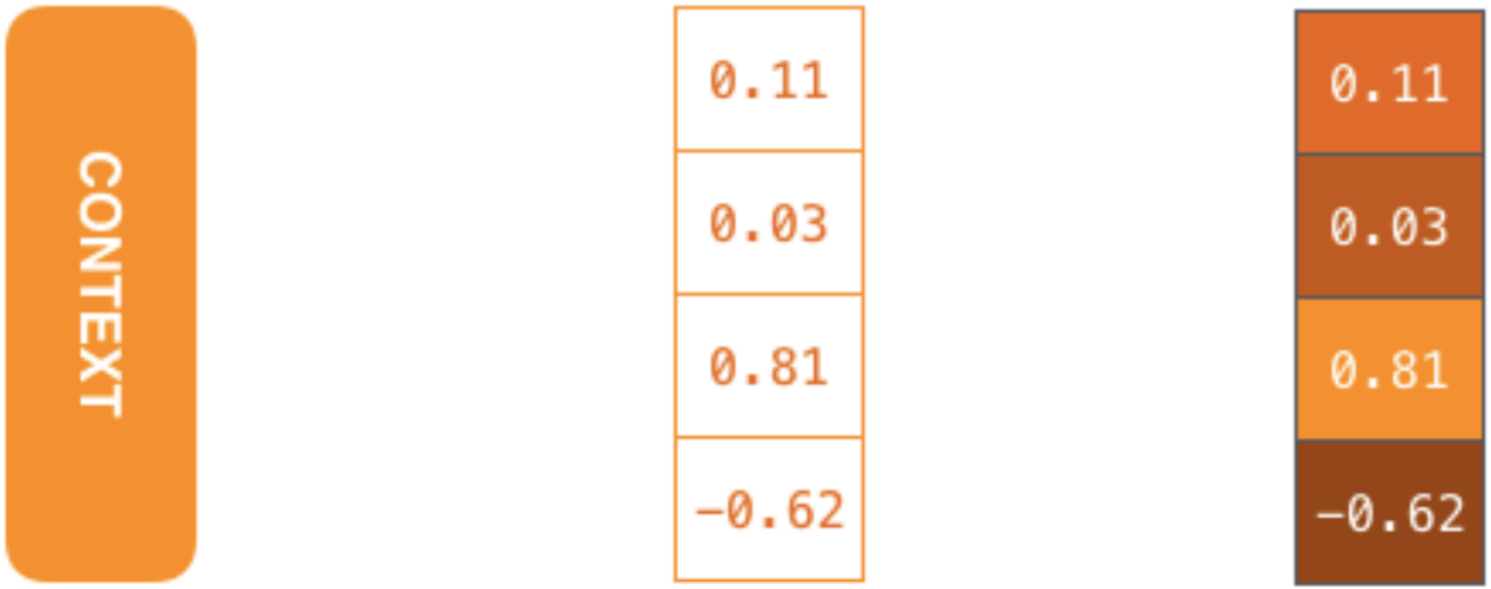
Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/



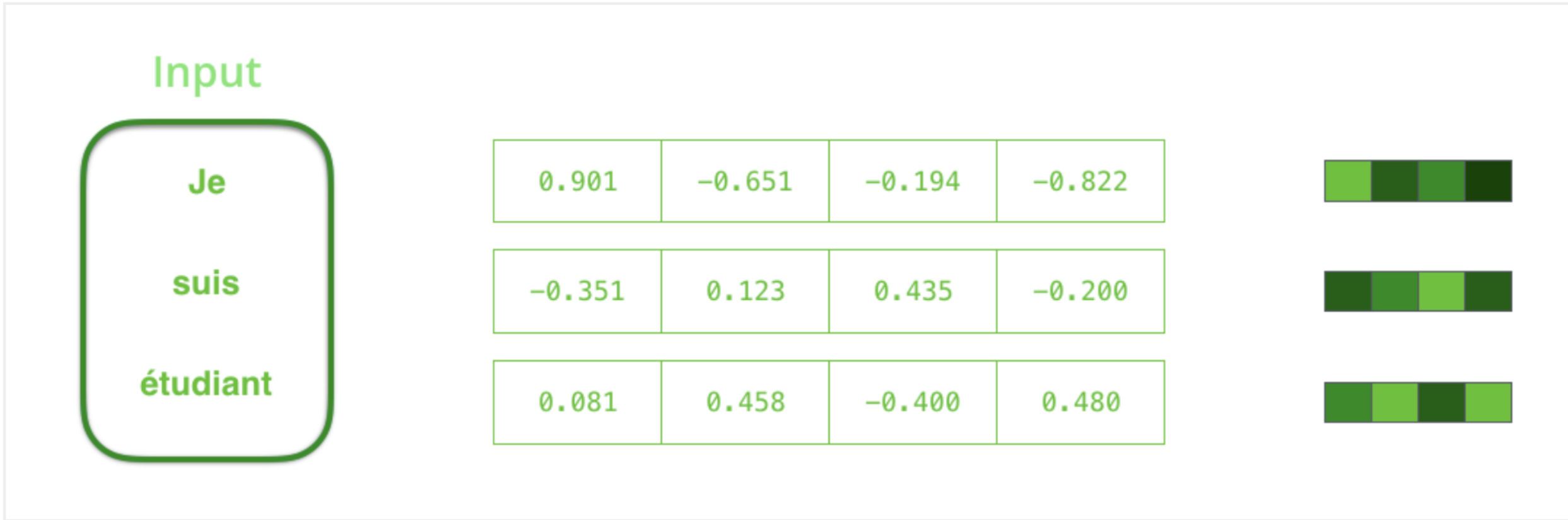
Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/



Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/



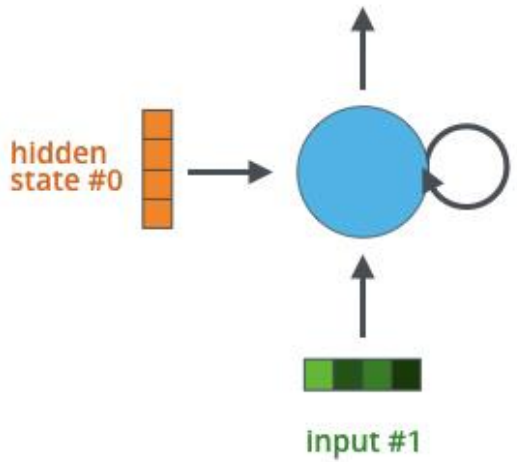
The context is a vector of floats. Later in this post we will visualize vectors in color by assigning brighter colors to the cells with higher values.



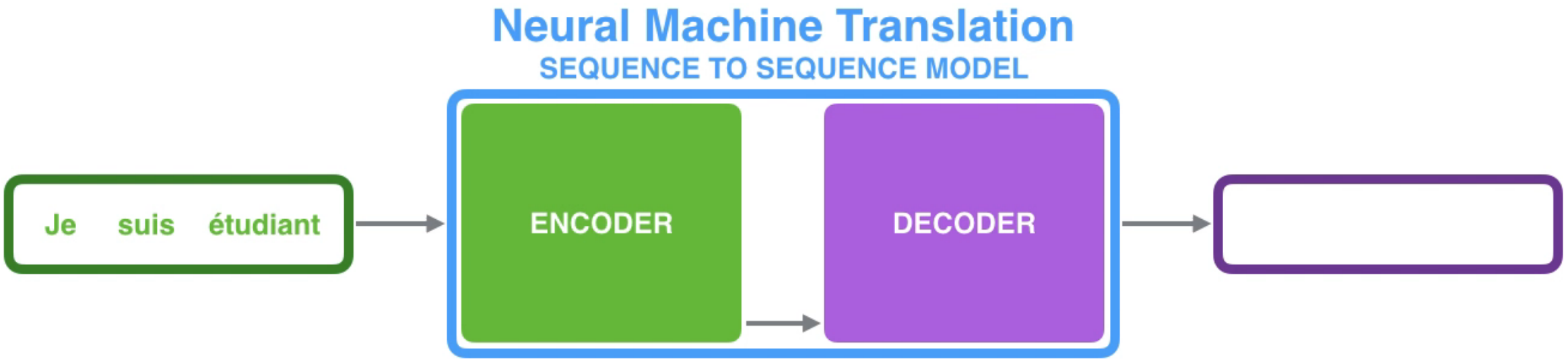
We need to turn the input words into vectors before processing them. That transformation is done using a word embedding algorithm. We can use pre-trained embeddings or train our own embedding on our dataset. Embedding vectors of size 200 or 300 are typical, we're showing a vector of size four for simplicity.

Recurrent Neural Network

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



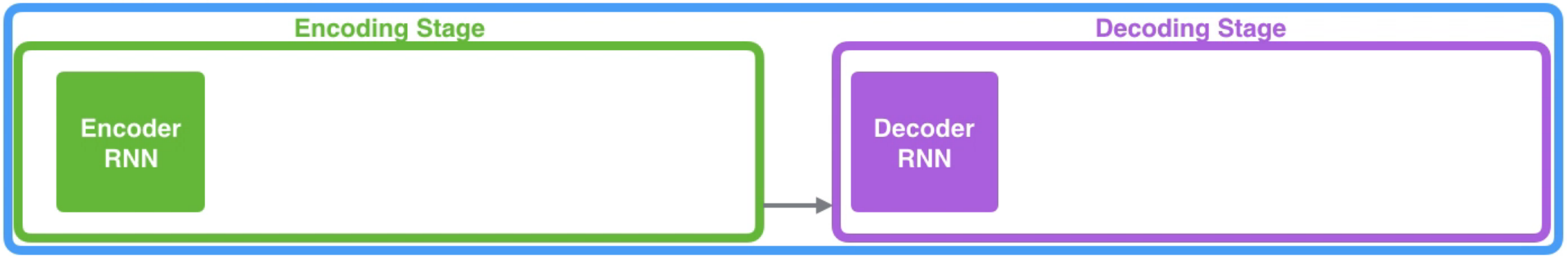
Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/



Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Neural Machine Translation

SEQUENCE TO SEQUENCE MODEL



Je suis étudiant

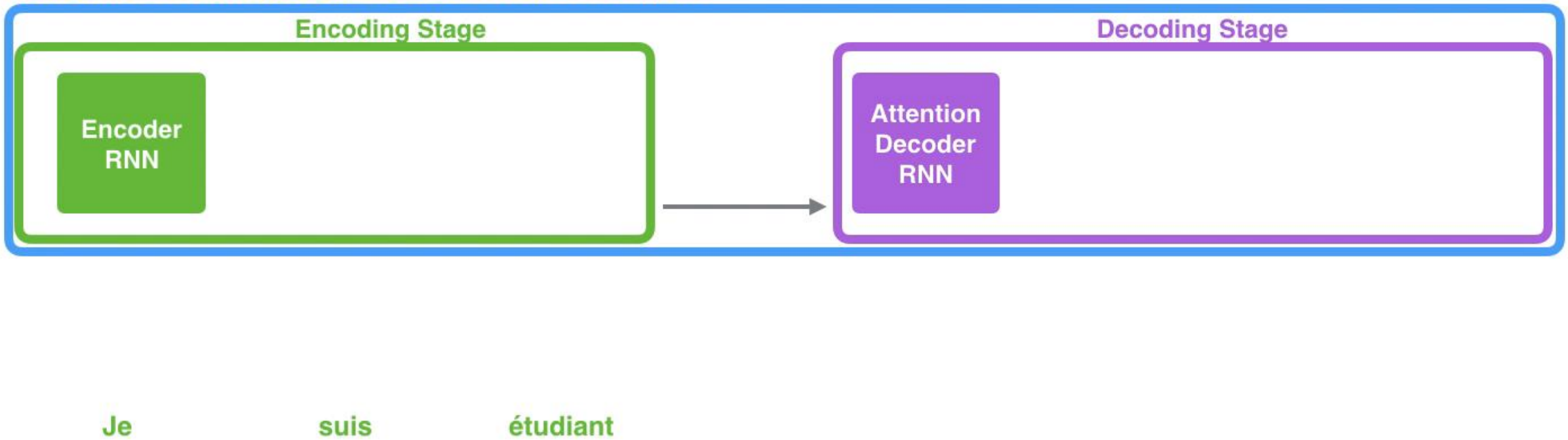
Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Time step: 7

At time step 7, the attention mechanism enables the decoder to focus on the word "étudiant" ("student" in french) before it generates the English translation. This ability to amplify the signal from the relevant part of the input sequence makes attention models produce better results than models without attention.

Neural Machine Translation

SEQUENCE TO SEQUENCE MODEL WITH ATTENTION



Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

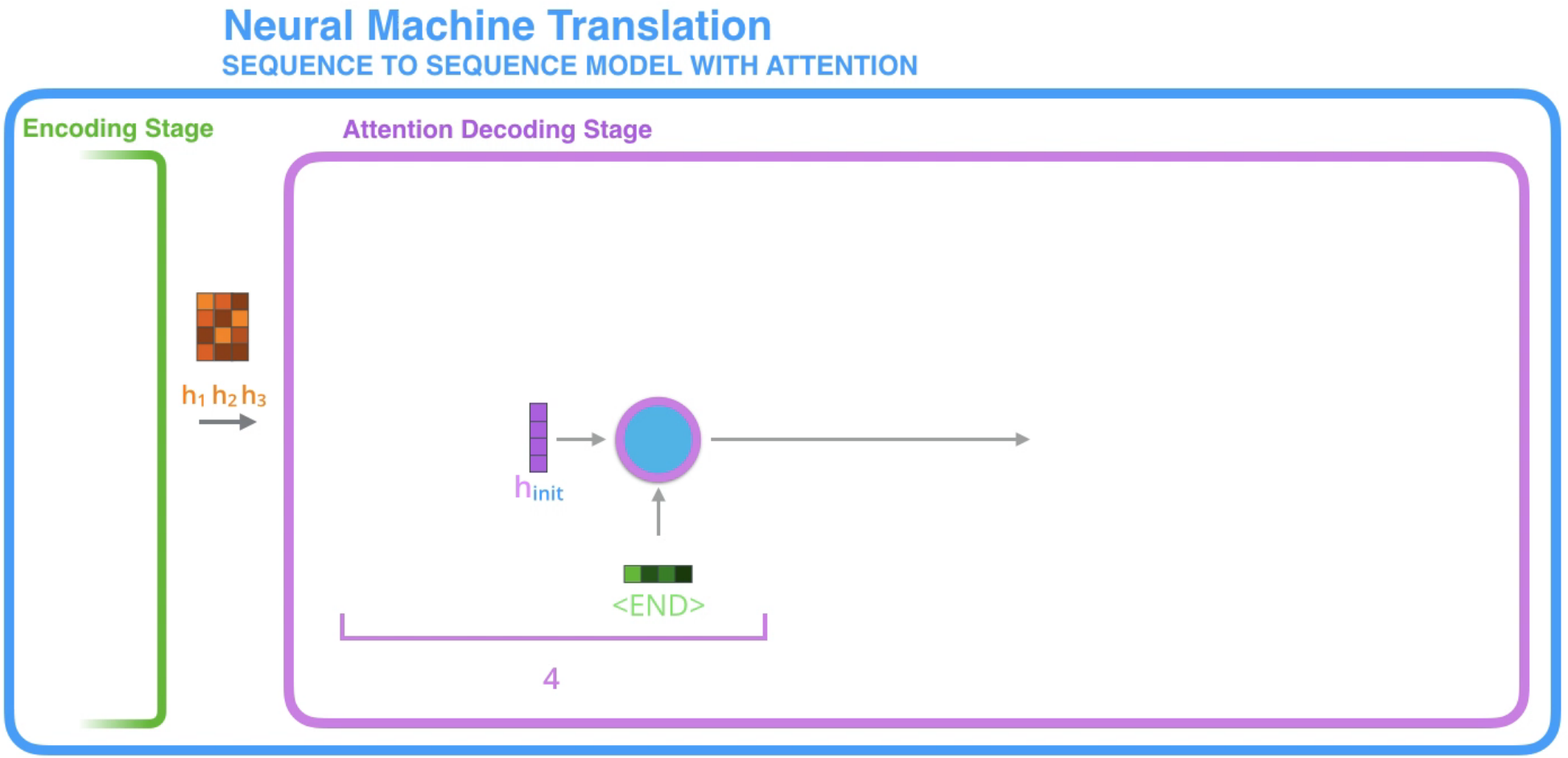
Attention at time step 4

Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

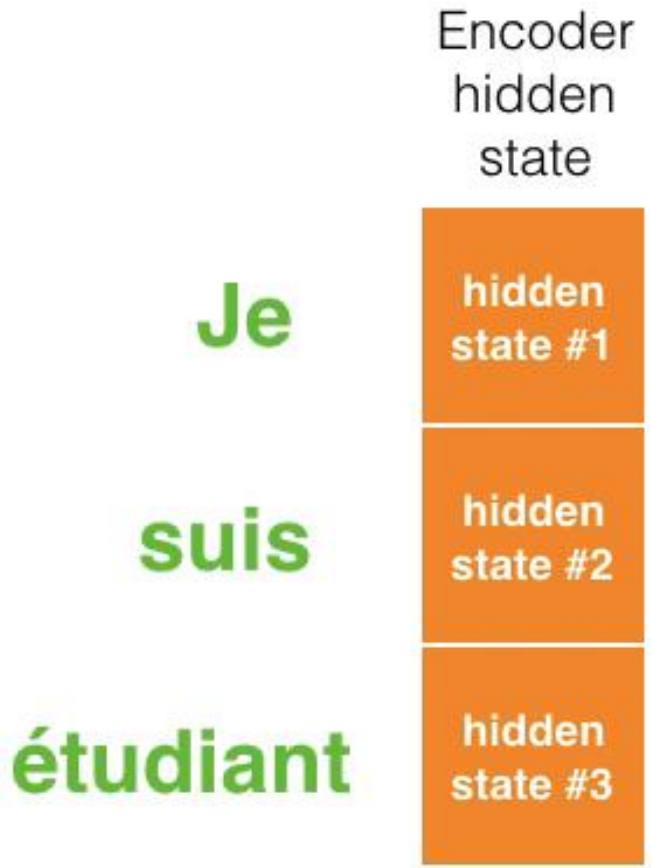
This scoring exercise is done at each time step on the decoder side.

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

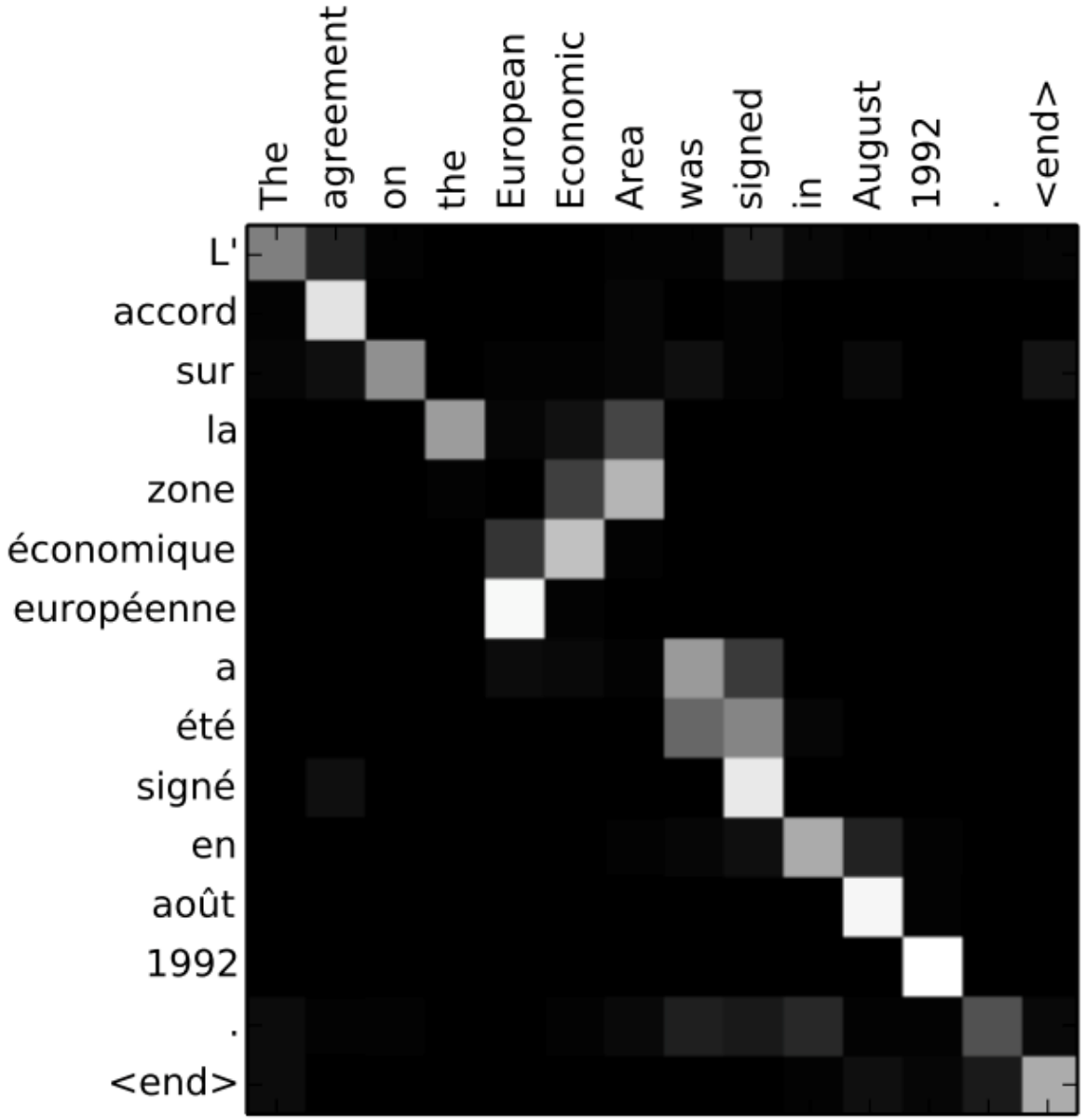
1. The attention decoder RNN takes in the embedding of the <END> token, and an initial decoder hidden state.
2. The RNN processes its inputs, producing an output and a new hidden state vector (h4). The output is discarded.
3. Attention Step: We use the encoder hidden states and the h4 vector to calculate 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



Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/



Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/



Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Neural Attention

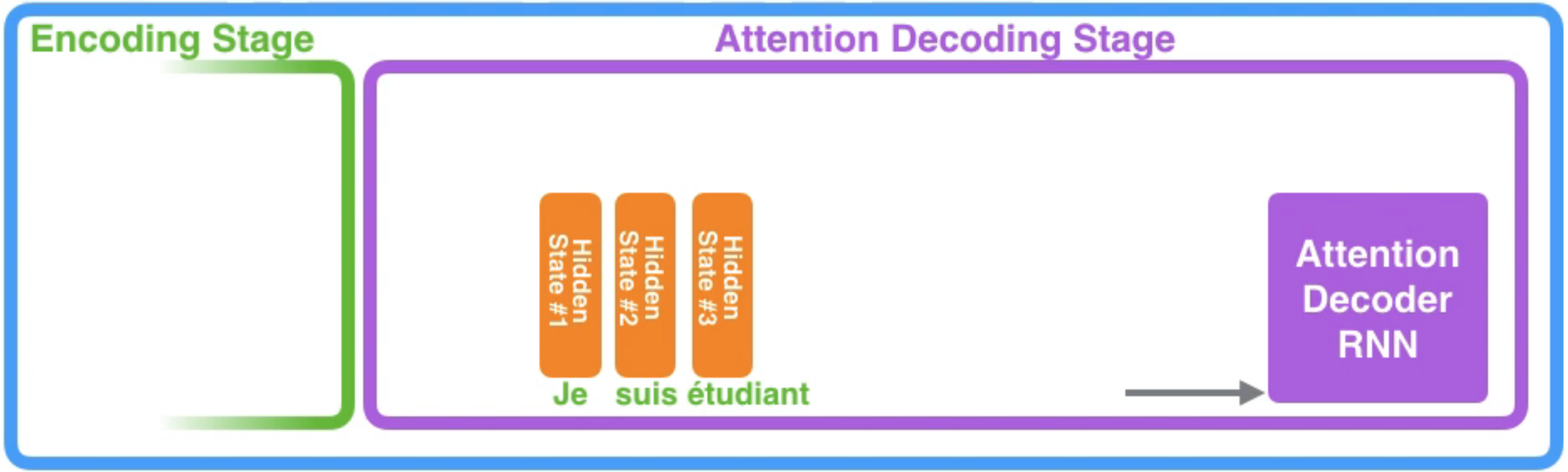
"You can't cram the meaning of a whole %&!$ing sentence into a single $&!*ing vector!"

Ray Mooney

- Representing sentences?
  - Use a sequence of vectors
- Neural Attention
  - Introduced by Bahdanau et al. 2014
  - Encode each word as a vector
  - When decoding, use a linear combination of these vectors
  - Using attention vectors
  - Pick the next word based on the output

Neural Machine Translation

SEQUENCE TO SEQUENCE MODEL WITH ATTENTION



Jay Alammar. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Useful tutorials

- https://www.tensorflow.org/versions/master/tutorials/seq2seq
- https://github.com/tensorflow/nmt

Advantages of Neural Attention

- It improves performance
- It solves the bottleneck problem
- It provides shortcuts to remote words, thus helping with vanishing gradients
- It provides some interpretability
- It directly leads to the next big thing, the Transformer

[Example from Chris Manning]

Attention Score Functions

Name	Alignment score function	Citation
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves2014
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

(*) Referred to as "concat" in Luong, et al., 2015 and as "additive attention" in Vaswani, et al., 2017.

(^) It adds a scaling factor $1/\sqrt{n}$, motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning.

https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

What to Attend to?

- Input Sentence
  - Copying mechanism
- Previously generated things
  - In LM – previous words (Merrity et al. 2016)
  - In MT – either input or previous output (Vaswani et al. 2017)
- Other modalities
  - Images (Xu et al. 2015)
  - Speech (Chan et al. 2015)
- Multiple sources
  - Multiple sentences (Zoph et al. 2015)
  - Both a sentence and an image (Huang et al. 2016)

[Examples from Graham Neubig]

Attention for abstractive summarization

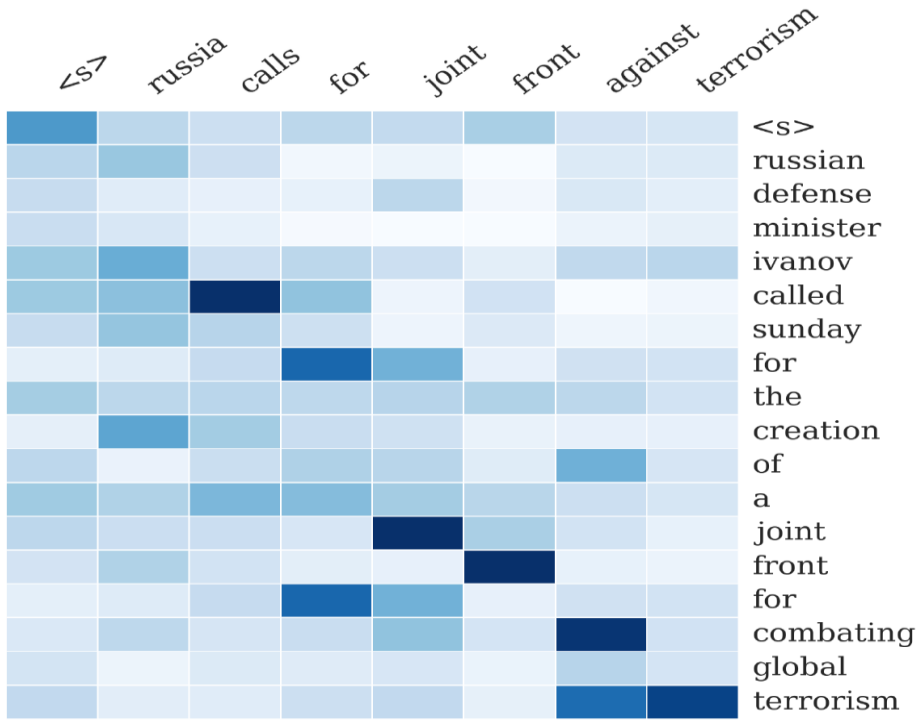


Figure 1: Example output of the attention-based summarization (ABS) system. The heatmap represents a soft alignment between the input (right) and the generated summary (top). The columns represent the distribution over the input after generating each word.

A Neural Attention Model for Abstractive Sentence Summarization

Alexander M. Rush **Sumit Chopra** **Jason Weston**

Self-Attention

Self-attention, also known as **intra-attention**, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the same sequence. It has been shown to be very useful in machine reading, abstractive summarization, or image description generation.

The long short-term memory network paper used self-attention to do machine reading. In the example below, the self-attention mechanism enables us to learn the correlation between the current words and the previous part of the sentence.

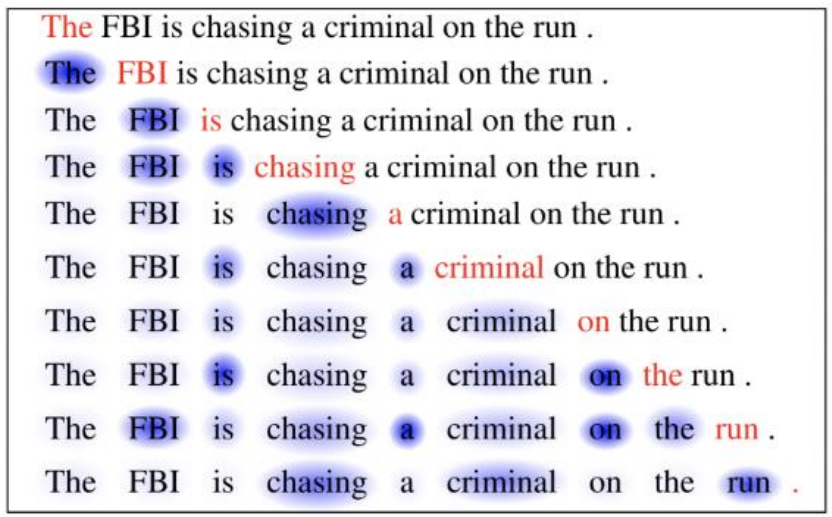


Fig. 6. The current word is in red and the size of the blue shade indicates the activation level. (Image source: Cheng et al., 2016)

Long Short-Term Memory-Networks for Machine Reading

Jianpeng Cheng, Li Dong and **Mirella Lapata**
School of Informatics, University of Edinburgh
10 Crichton Street, Edinburgh EH8 9AB
{jianpeng.cheng,li.dong}@ed.ac.uk, mlap@inf.ed.ac.uk

https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Self-Attention

- Each word serves as a “query”
- The query is then used to compute attention to other words

Pointer Network

In problems like sorting or travelling salesman, both input and output are sequential data. Unfortunately, they cannot be easily solved by classic seq-2-seq or NMT models, given that the discrete categories of output elements are not determined in advance, but depends on the variable input size. The **Pointer Net** (**Ptr-Net**; Vinyals, et al. 2015) is proposed to resolve this type of problems: When the output elements correspond to *positions* in an input sequence. Rather than using attention to blend hidden units of an encoder into a context vector (See Fig. 8), the Pointer Net applies attention over the input elements to pick one as the output at each decoder step.

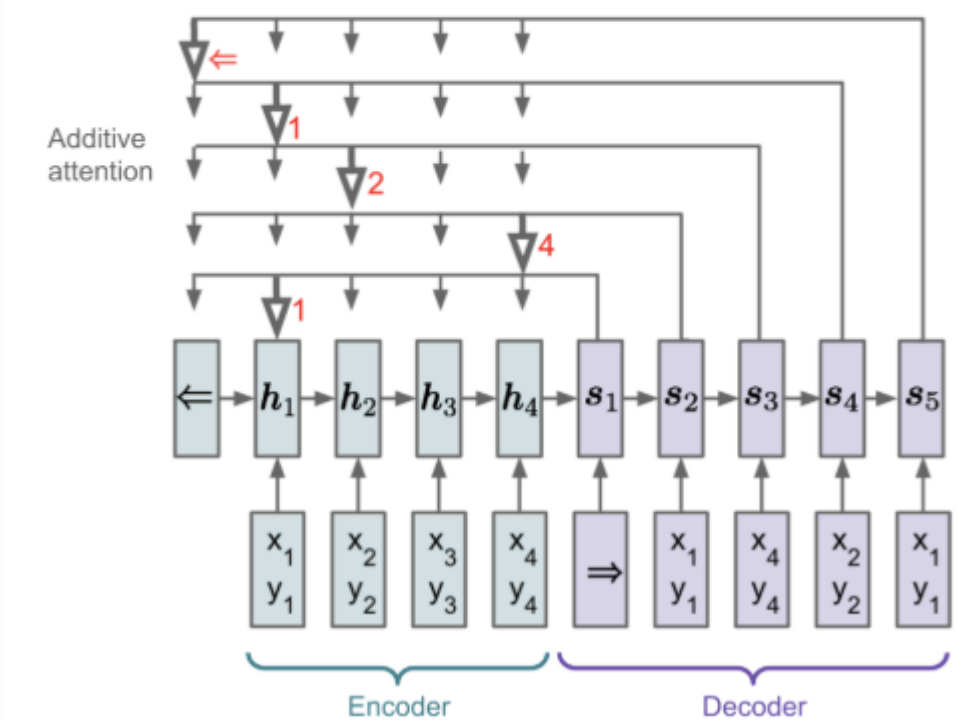


Fig. 13. The architecture of a Pointer Network model. (Image source: Vinyals, et al. 2015)

https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Introduction to NLP

758.

Transformers

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

1 Introduction

Attention is all you need
A Vaswani, N Shazeer, N Parmar, J Uszkoreit, L Jones, AN Gomez, ...
Advances in neural information processing systems, 5998-6008

7168 2017

The Transformer (Vaswani et al. 2017)

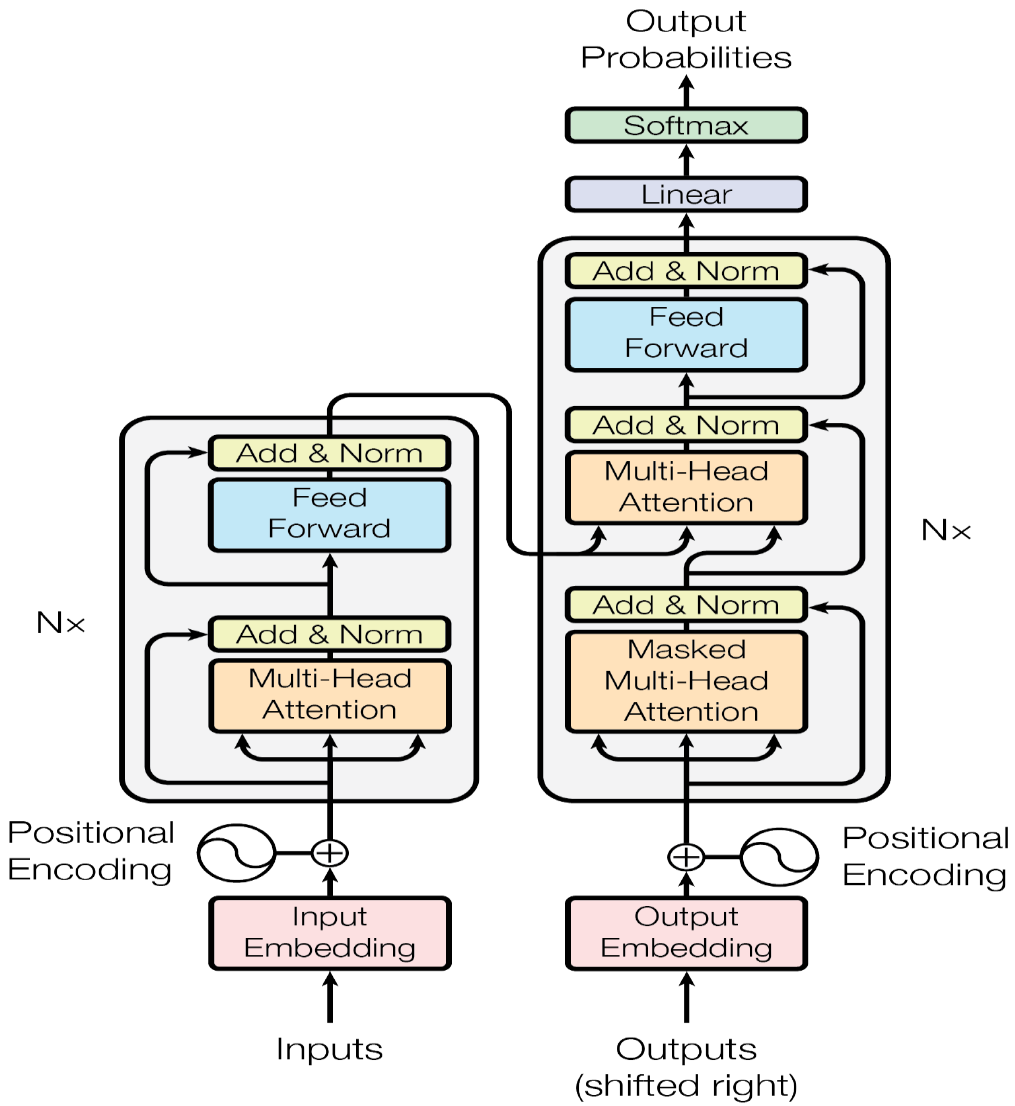
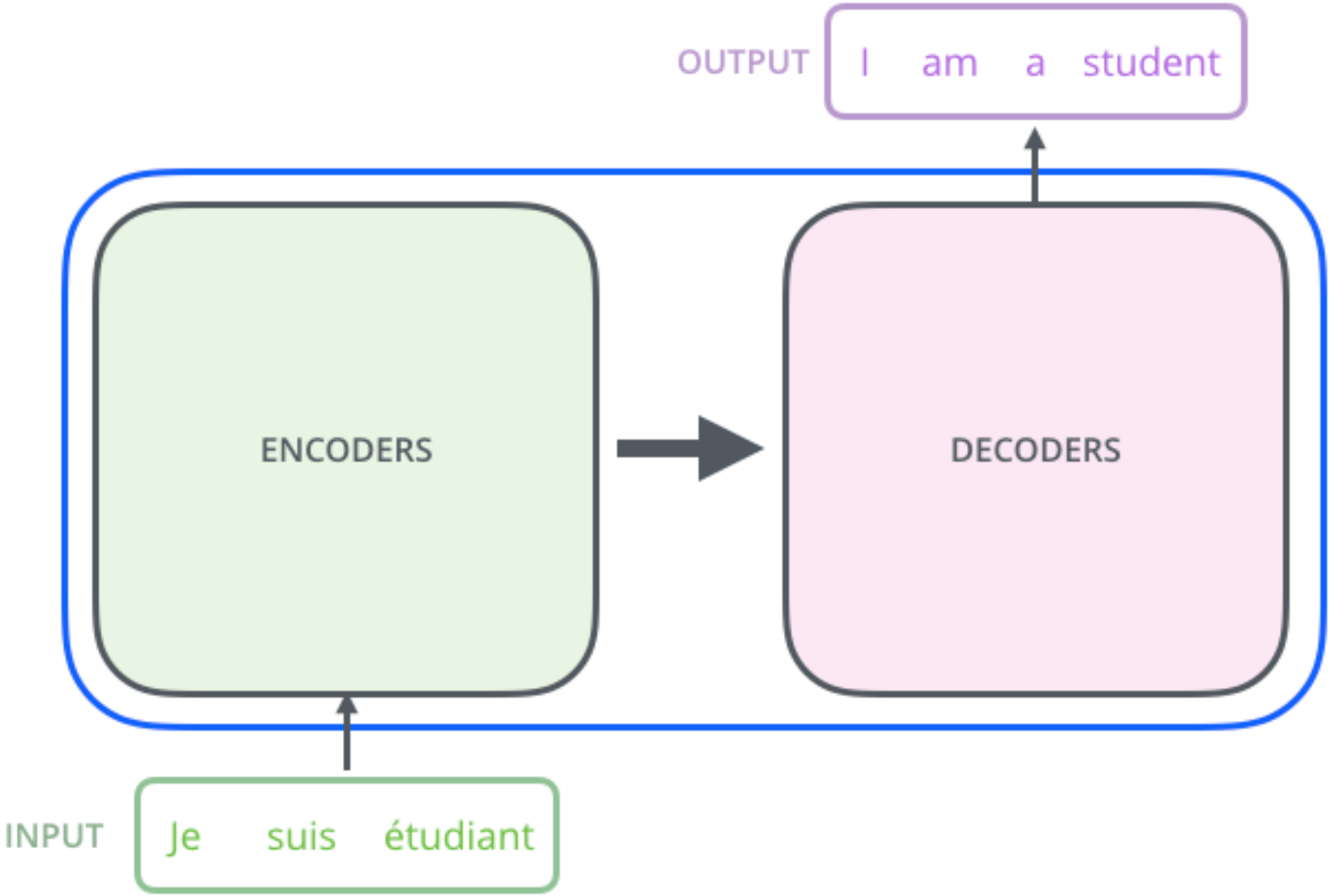


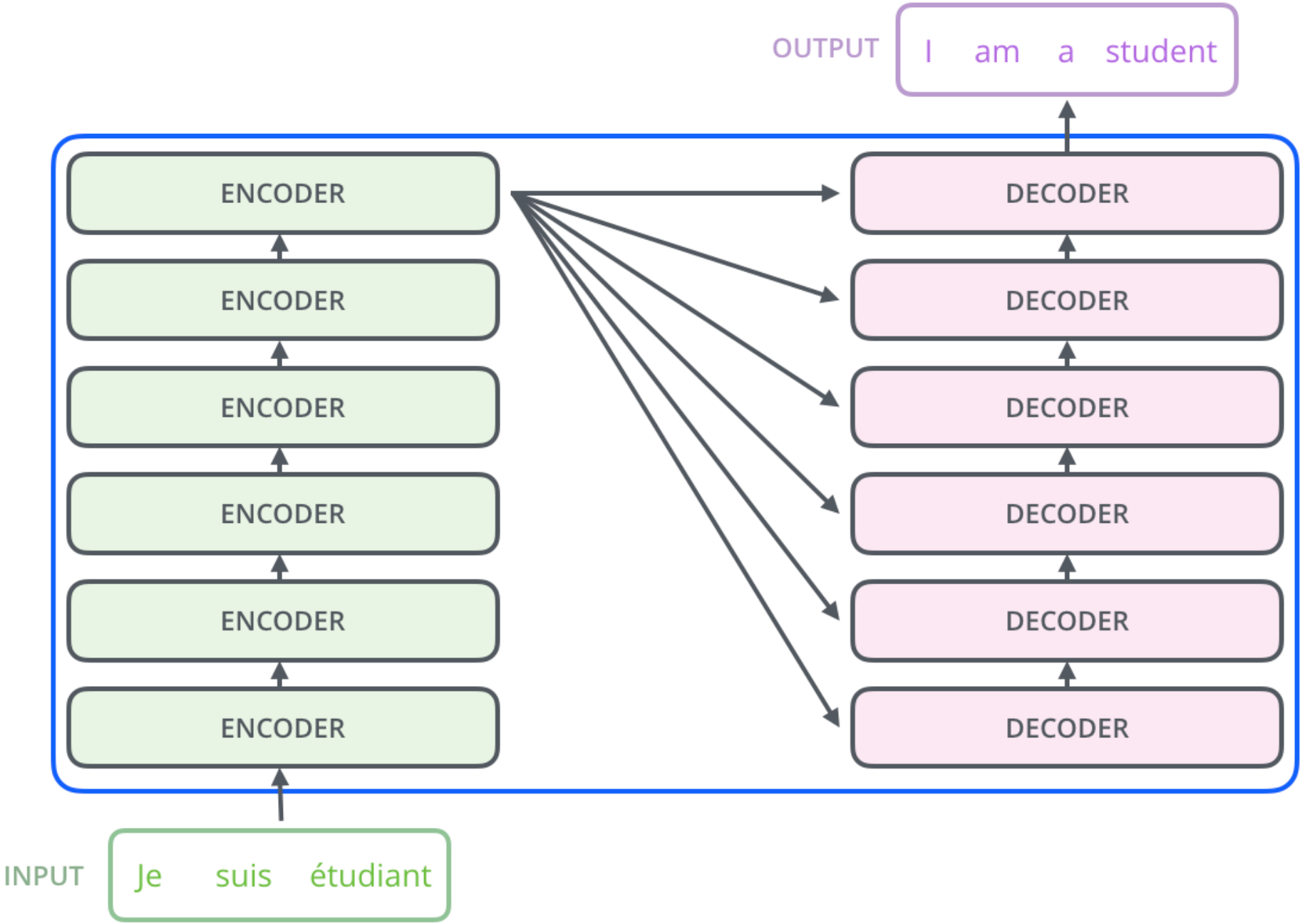
Figure 1: The Transformer - model architecture.

Another awesome tutorial by Jay Alammar

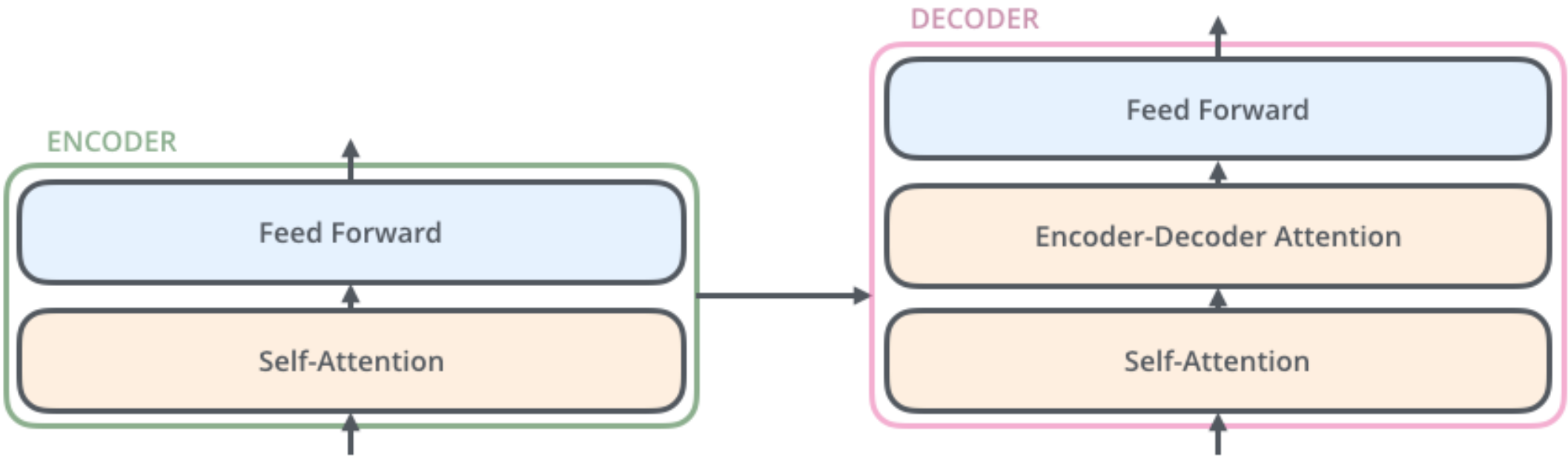
http://jalammar.github.io/illustrated-transformer/



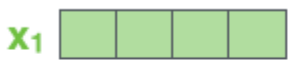
http://jalammar.github.io/illustrated-transformer/



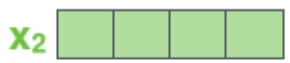
http://jalammar.github.io/illustrated-transformer/



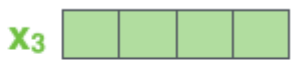
http://jalammar.github.io/illustrated-transformer/



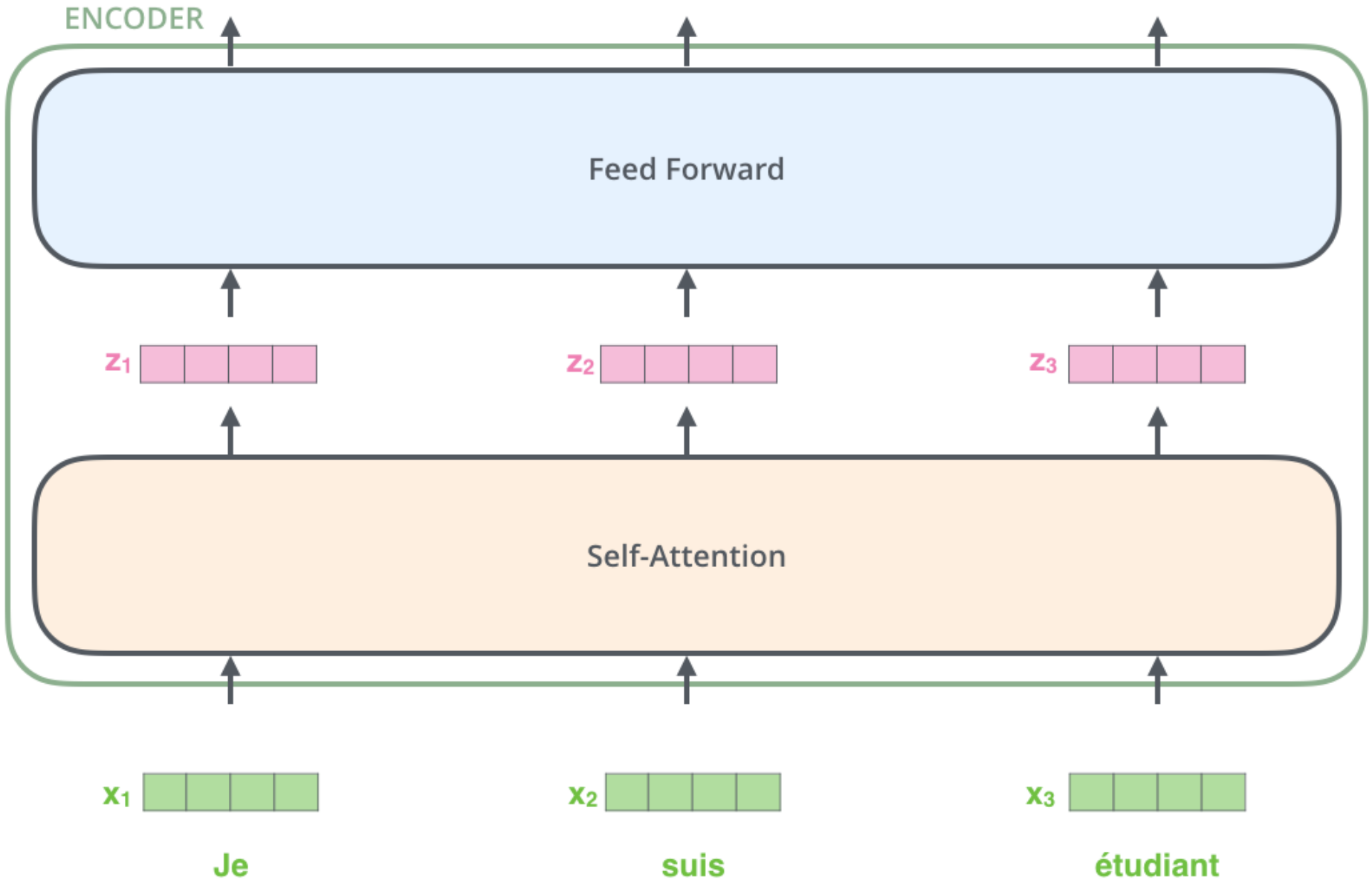
Je



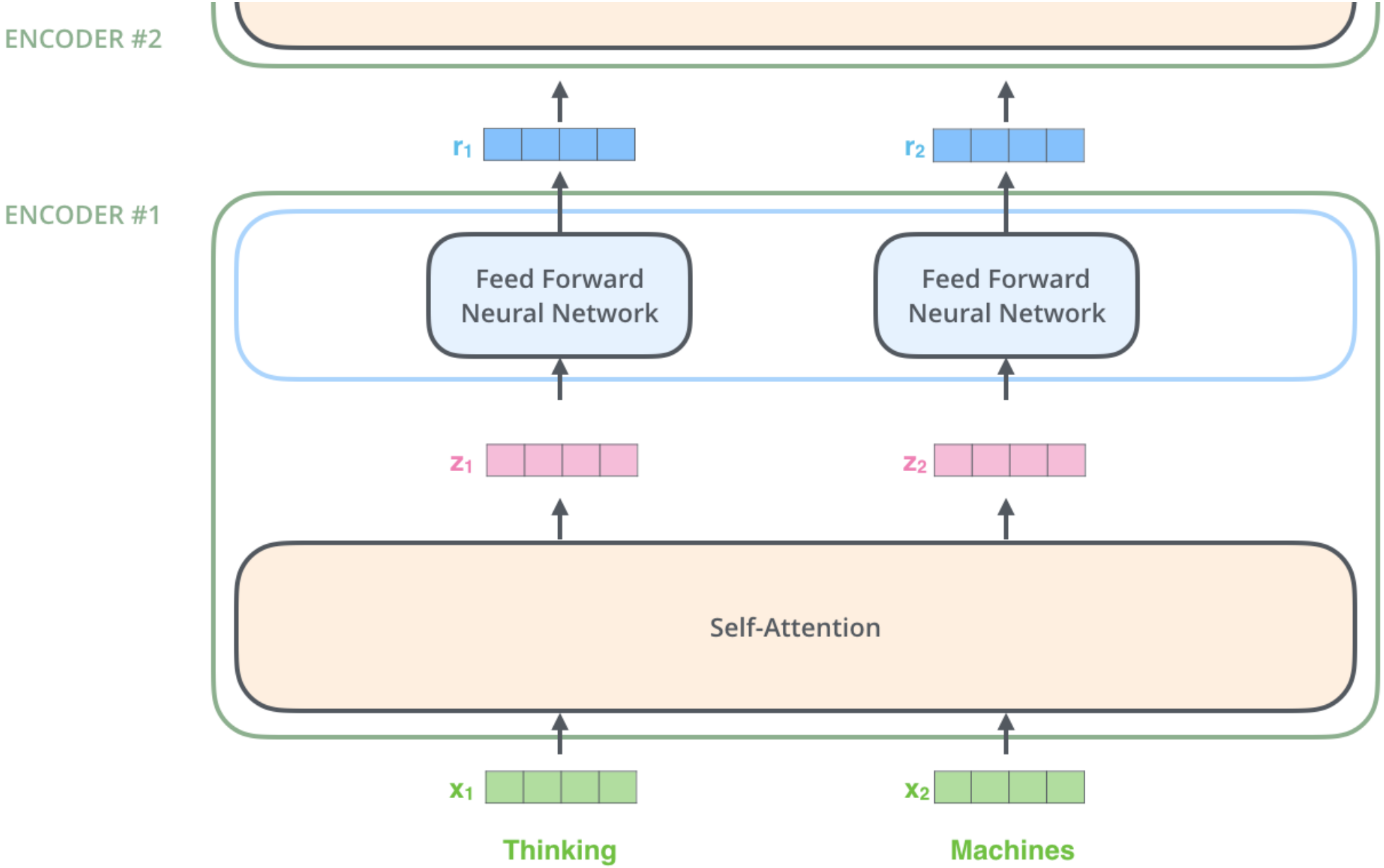
suis



étudiant



http://jalammar.github.io/illustrated-transformer/



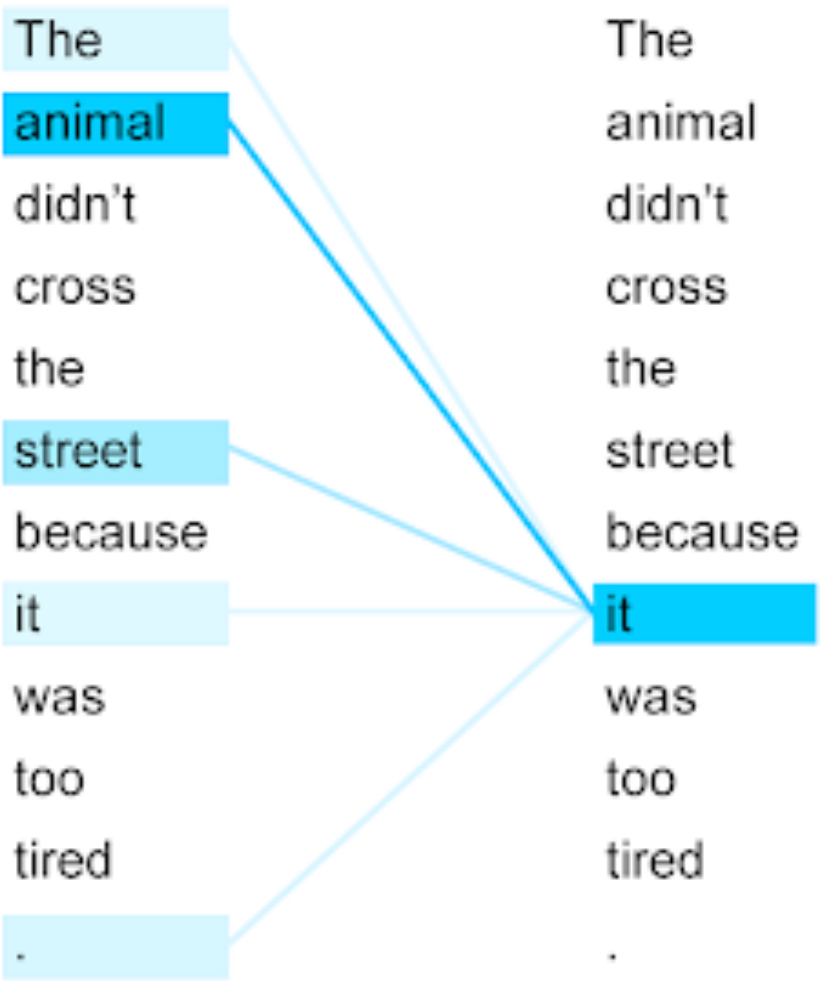
http://jalammar.github.io/illustrated-transformer/

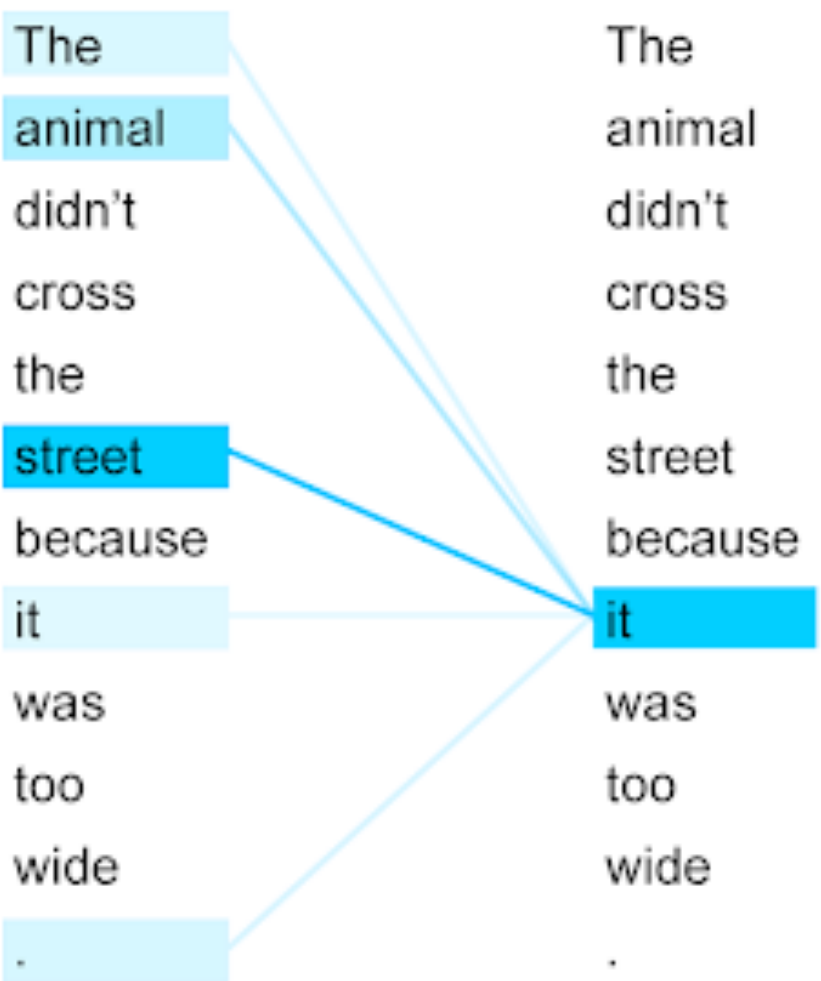
Self-attention

The animal didn't cross the street because it was too tired.
L'animal n'a pas traversé la rue parce qu'il était trop fatigué.

The animal didn't cross the street because it was too wide.
L'animal n'a pas traversé la rue parce qu'elle était trop large.

https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html





https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

Self-attention in detail

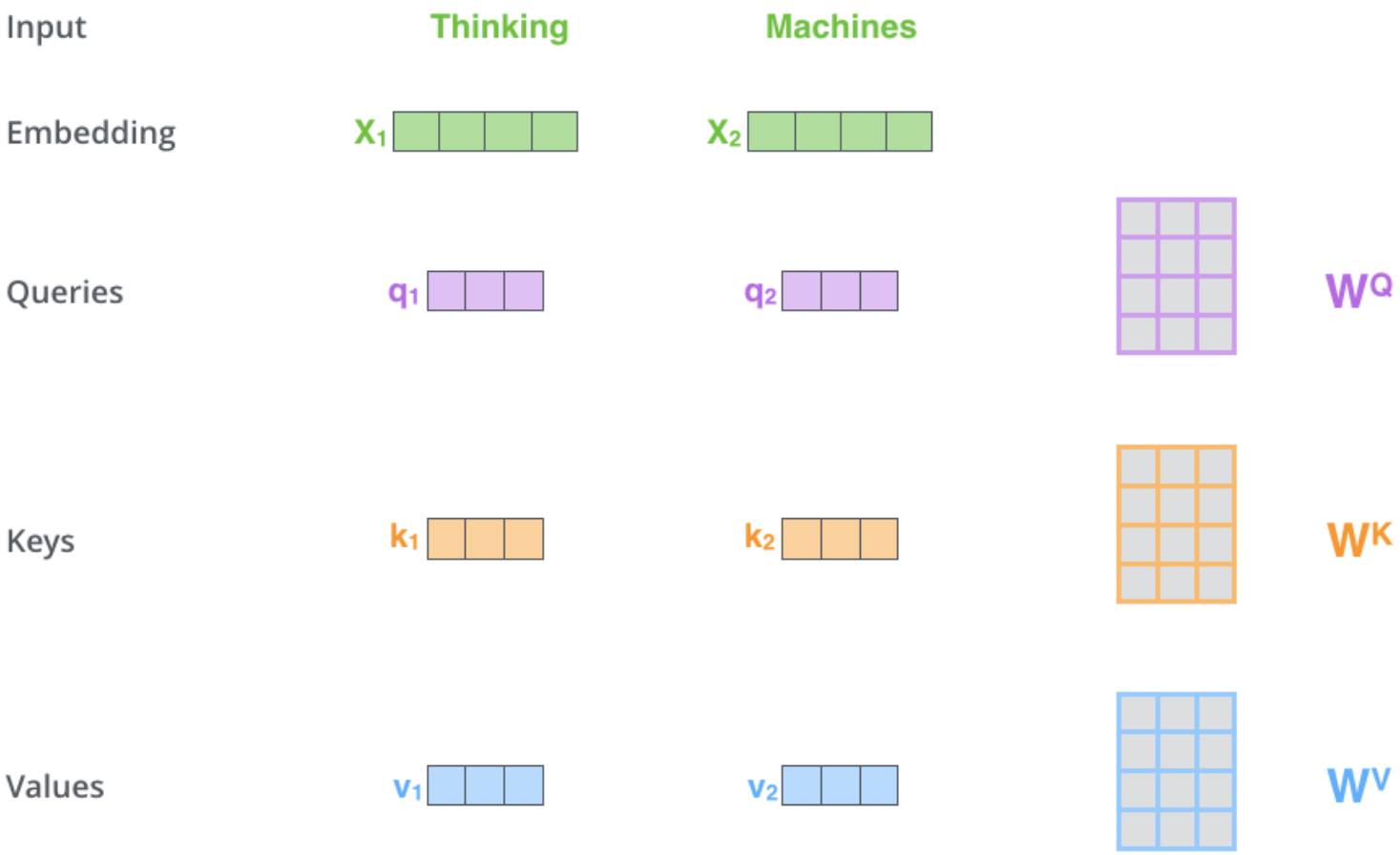
Let's first look at how to calculate self-attention using vectors, then proceed to look at how it's actually implemented – using matrices.

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.

http://jalammar.github.io/illustrated-transformer/

Self-attention in detail



Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

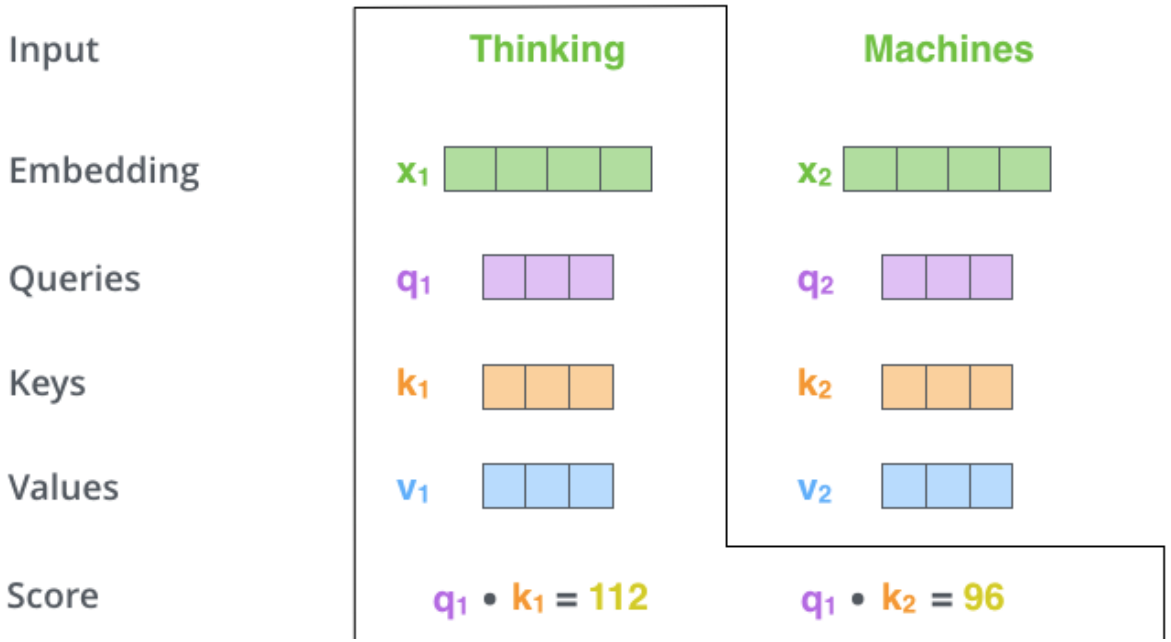
http://jalammar.github.io/illustrated-transformer/

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.

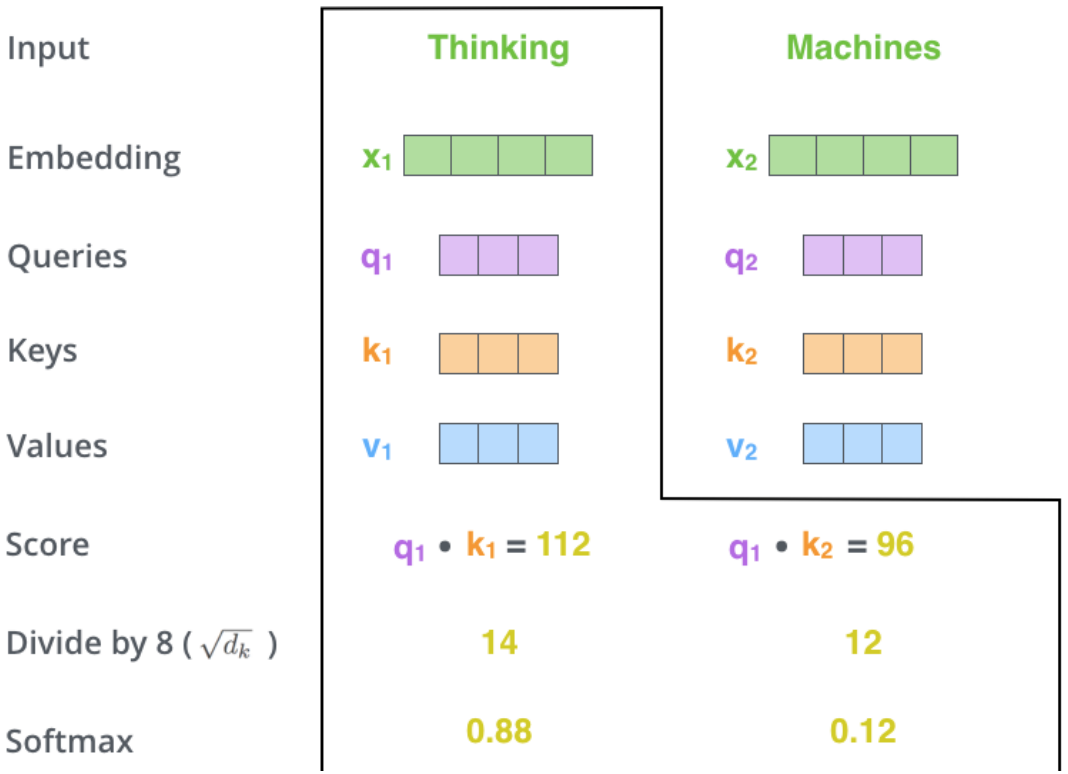
The **second step** in calculating self-attention is to calculate a score. Say we’re calculating the self-attention for the first word in this example, “Thinking”. We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input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



http://jalammar.github.io/illustrated-transformer/

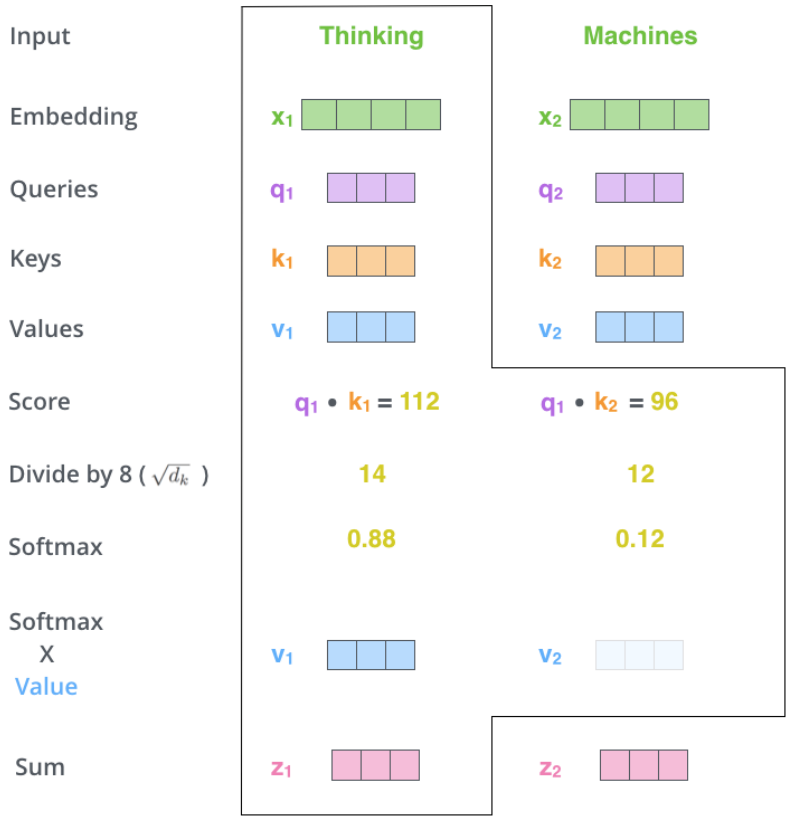
The **third and forth steps** are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax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



http://jalammar.github.io/illustrated-transformer/

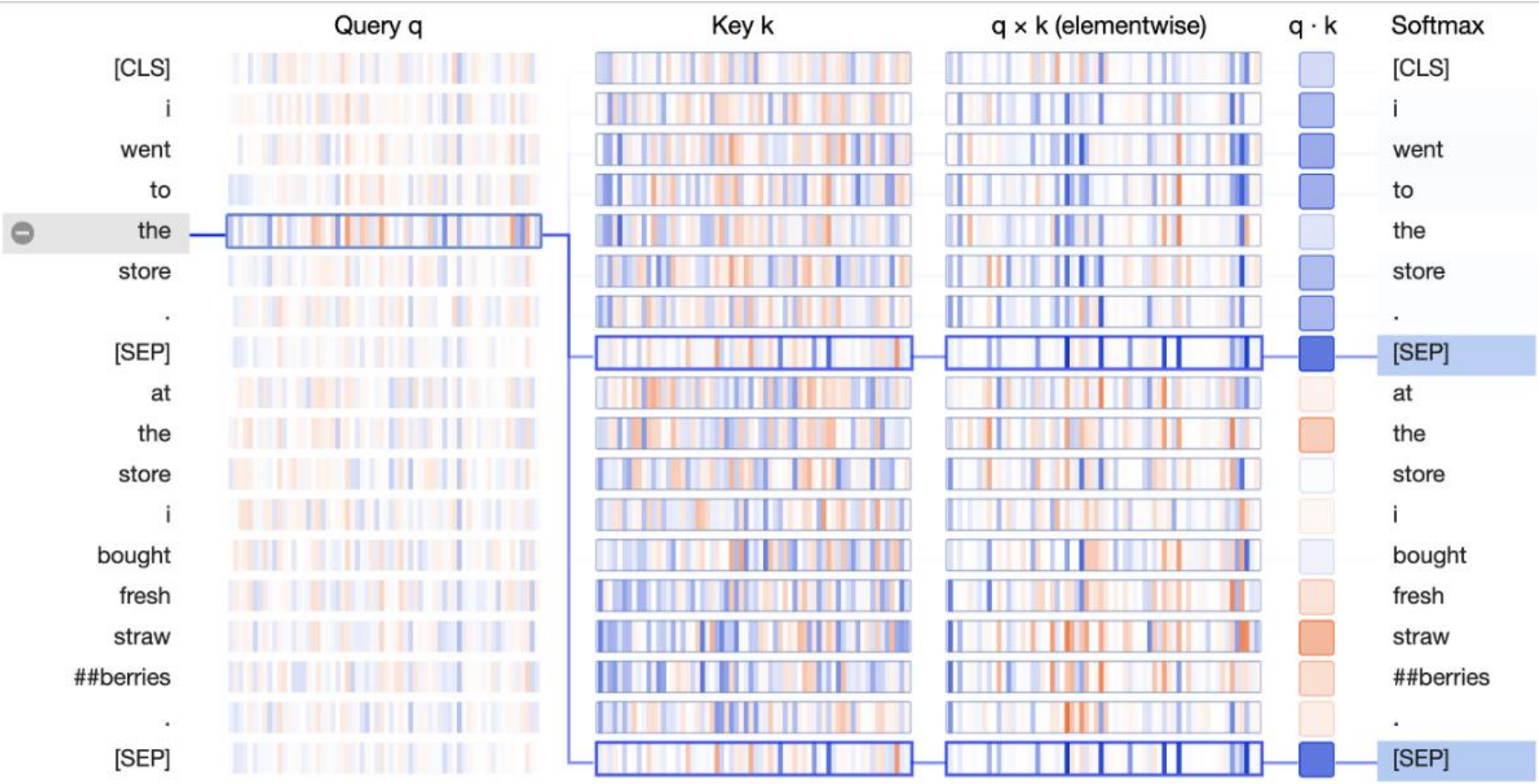
The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

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).



That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is done in matrix form for faster processing. So let's look at that now that we've seen the intuition of the calculation on the word level.

http://jalammar.github.io/illustrated-transformer/



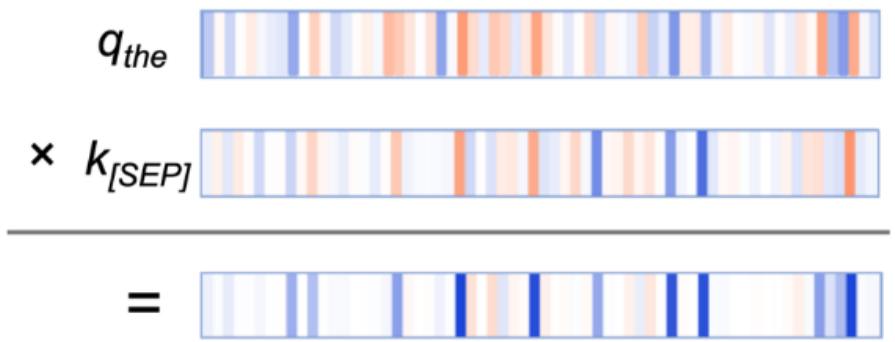
In the **Key** column, the key vectors for the two occurrences of *[SEP]* carry a distinctive signature: they both have a small number of active neurons with strongly positive (blue) or negative (orange) values, and a larger number of neurons with values close to zero (light blue/orange or white):

https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

In the **Key** column, the key vectors for the two occurrences of *[SEP]* carry a distinctive signature: they both have a small number of active neurons with strongly positive (blue) or negative (orange) values, and a larger number of neurons with values close to zero (light blue/orange or white):

Key vector for first [SEP] token.

The query vectors tend to match the *[SEP]* key vectors along those active neurons, resulting in high values for the elementwise product $q \times k$, as in this example:



Query vector for first occurrence of "the", key vector for first occurrence of [SEP], and elementwise product

https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

Let's go through the columns in the neuron view one at a time, and revisit some of the concepts discussed earlier:

Query q: the query vector *q* encodes the word on the left that is paying attention, i.e. the one that is "querying" the other words. In the example above, the query vector for "on" (the selected word) is highlighted.

Key k: the key vector *k* encodes the word on the right to which attention is being paid. The key vector and the query vector together determine a compatibility score between the two words.

https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

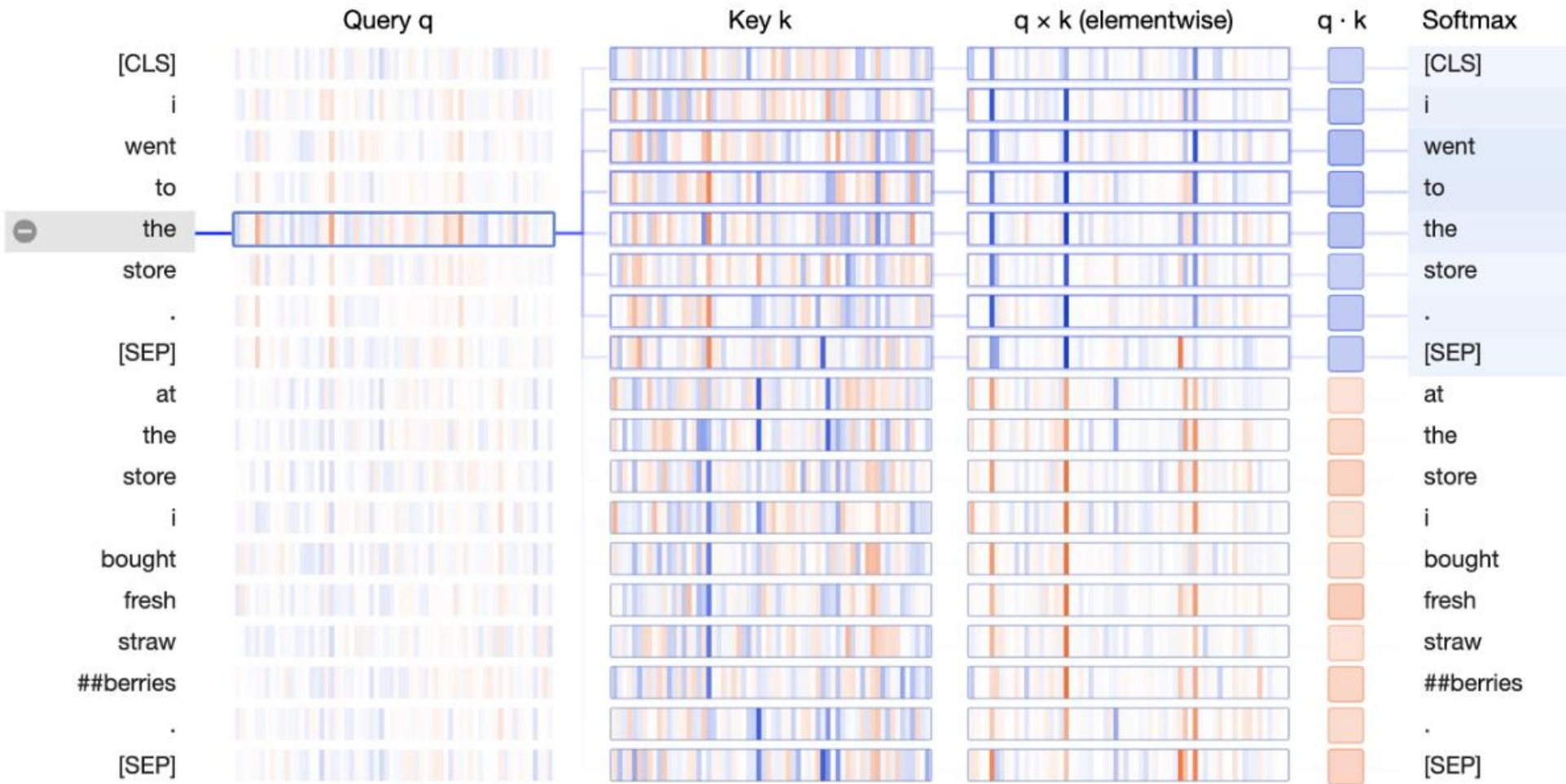
q×k (elementwise): the elementwise product between the query vector of the selected word and each of the key vectors. This is a precursor to the dot product (the sum of the elementwise product) and is included for visualization purposes because it shows how individual elements in the query and key vectors contribute to the dot product.

q·k: the scaled dot product (see above) of the selected query vector and each of the key vectors. This is the unnormalized attention score.

Softmax: the softmax of the scaled dot product. This normalizes the attention scores to be positive and sum to one.

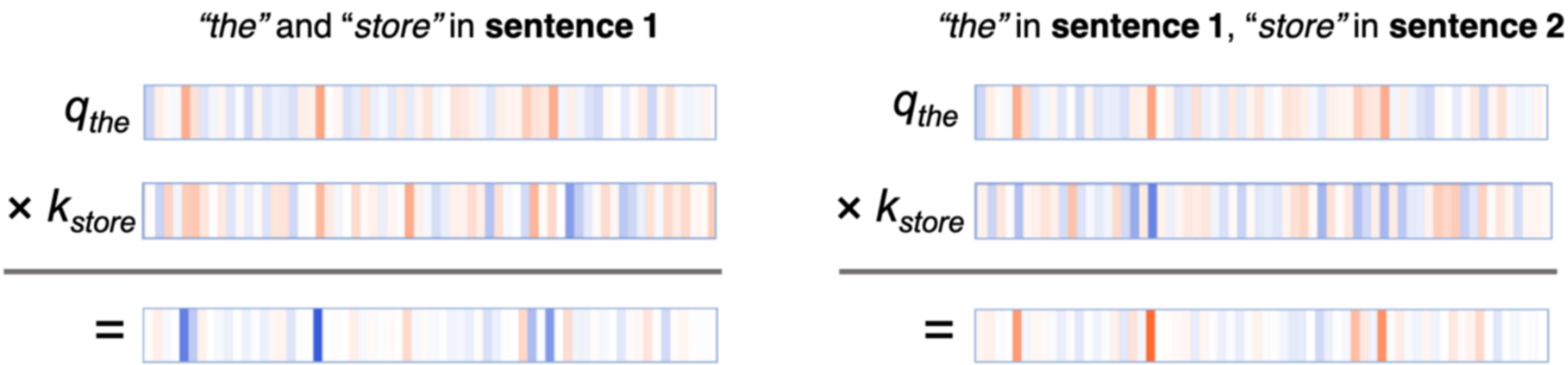
https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

So how does BERT finesse the queries and keys to form this attention pattern? Let's again turn to the neuron view:



https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

In the **q×k** column, we see a clear pattern: a small number of neurons (2–4) dominate the calculation of the attention scores. When query and key vector are in the same sentence (the first sentence, in this case), the product shows high values (blue) at these neurons. When query and key vector are in different sentences, the product is strongly negative (orange) at these same positions, as in this example:



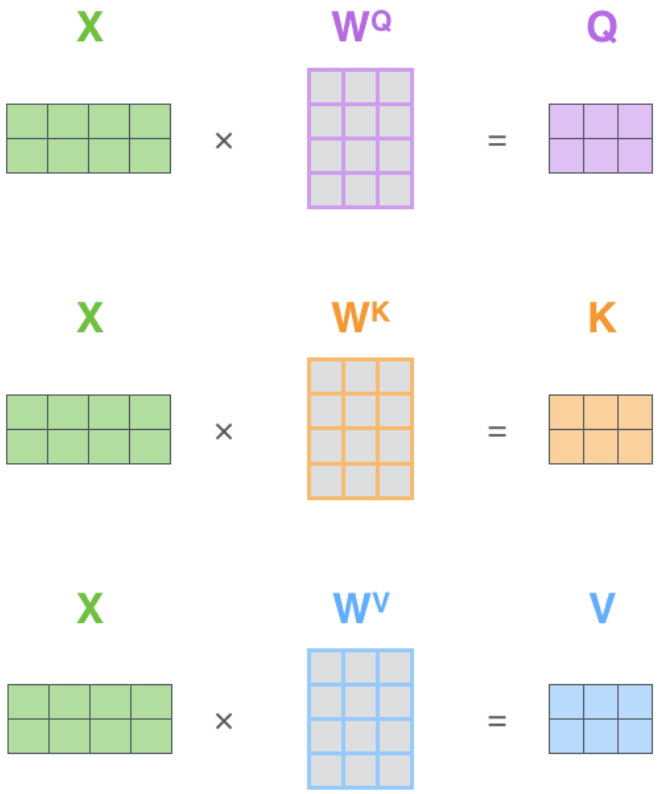
The query-key product tends to be positive when query and key are in the same sentence (left), and negative when query and key are in different sentences (right).

https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

Matrix Calculation of Self-Attention

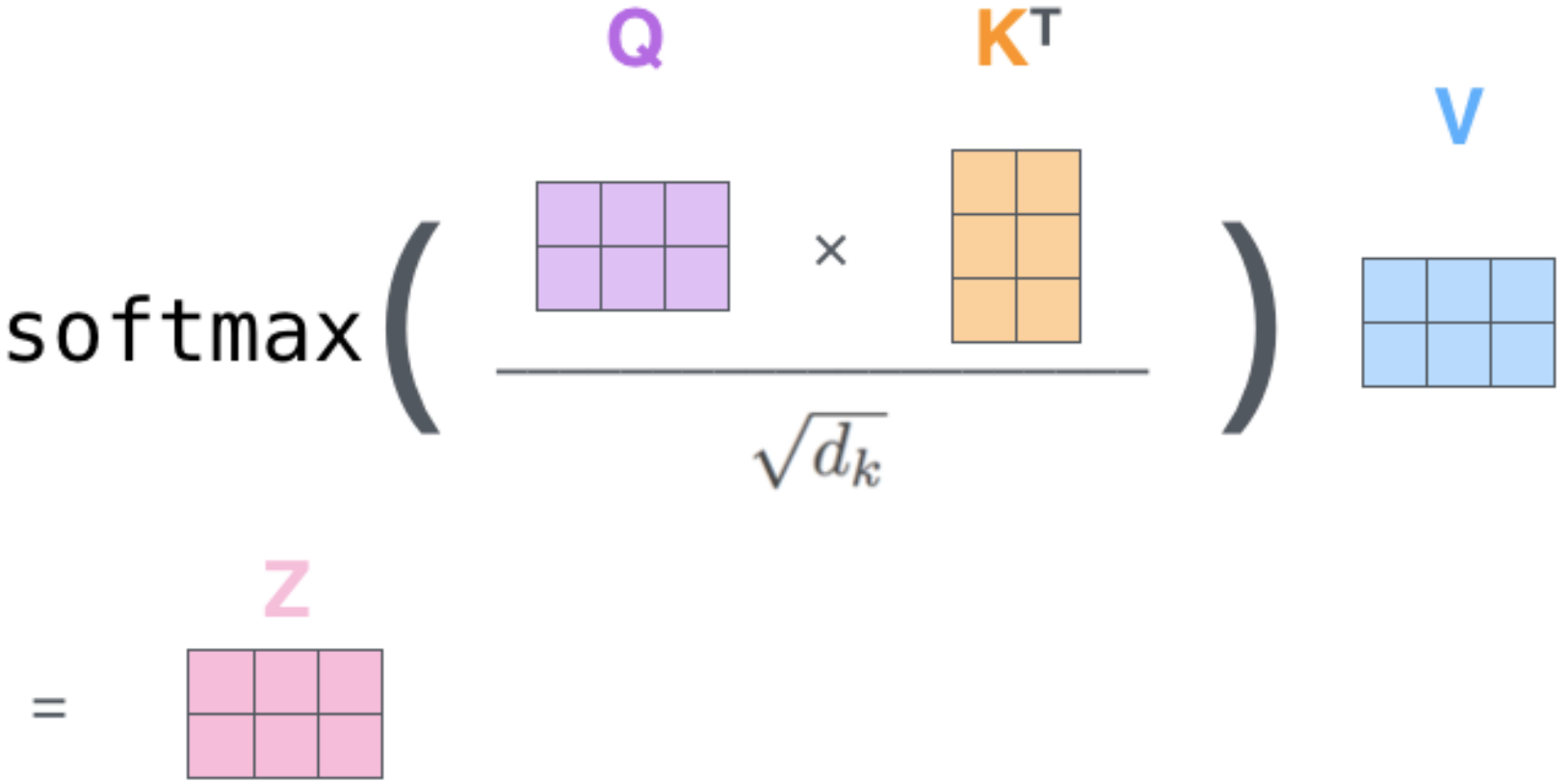
The first step is to calculate the Query, Key, and Value matrices. We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (WQ, WK, WV).



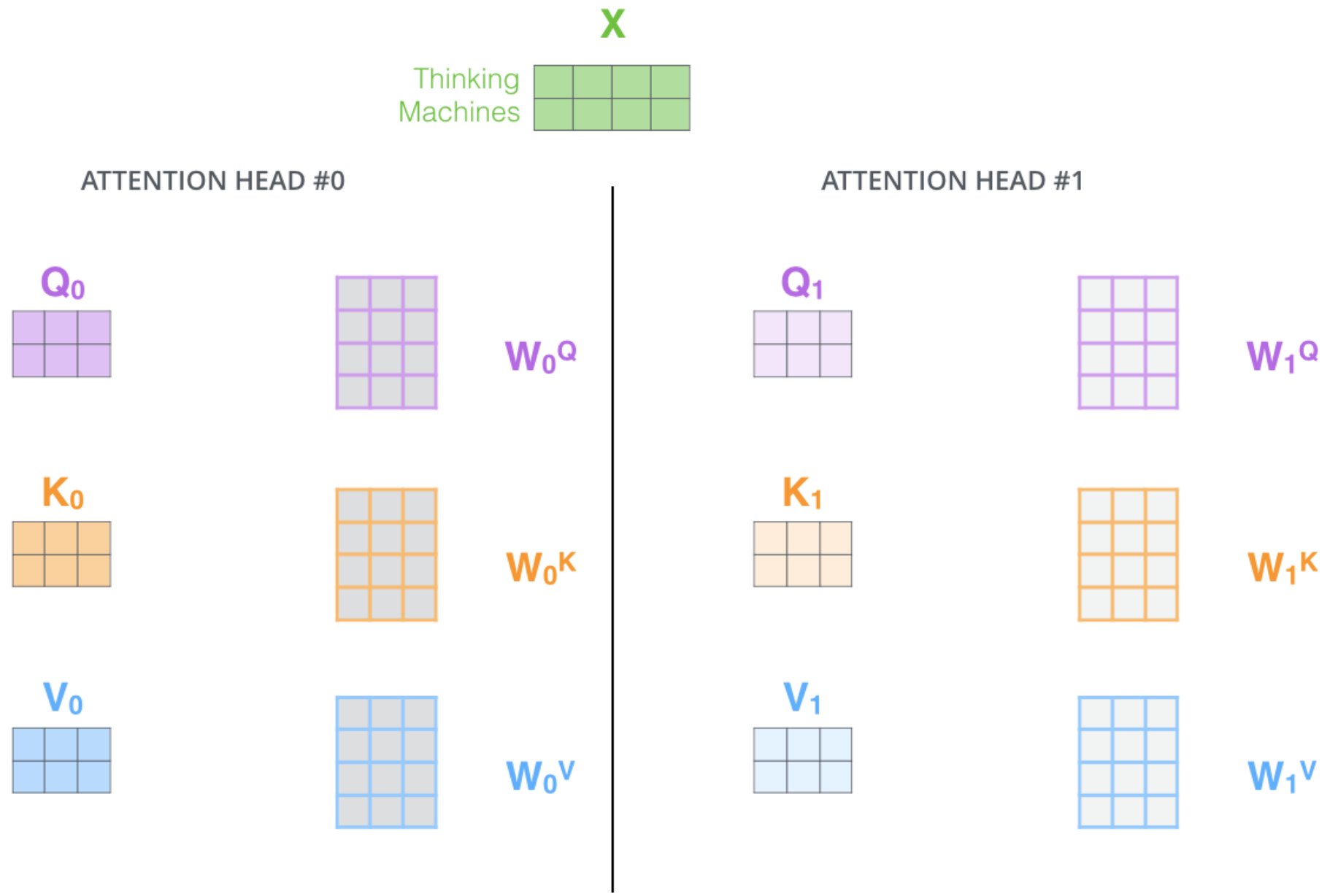
Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

http://jalammar.github.io/illustrated-transformer/

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.



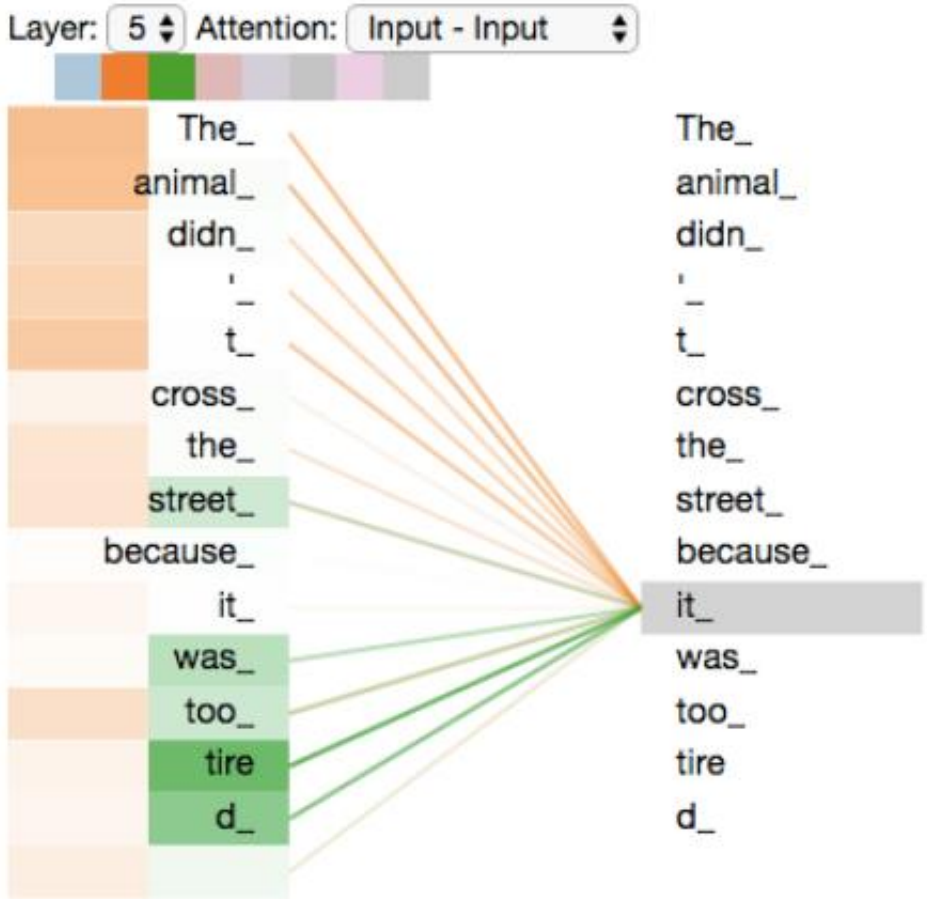
http://jalammar.github.io/illustrated-transformer/



http://jalammar.github.io/illustrated-transformer/

“The animal didn't cross the street because it was too tired”

Now that we have touched upon attention heads, let’s revisit our example from before to see where the different attention heads are focusing as we encode the word “it” in our example sentence:



As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

http://jalammar.github.io/illustrated-transformer/

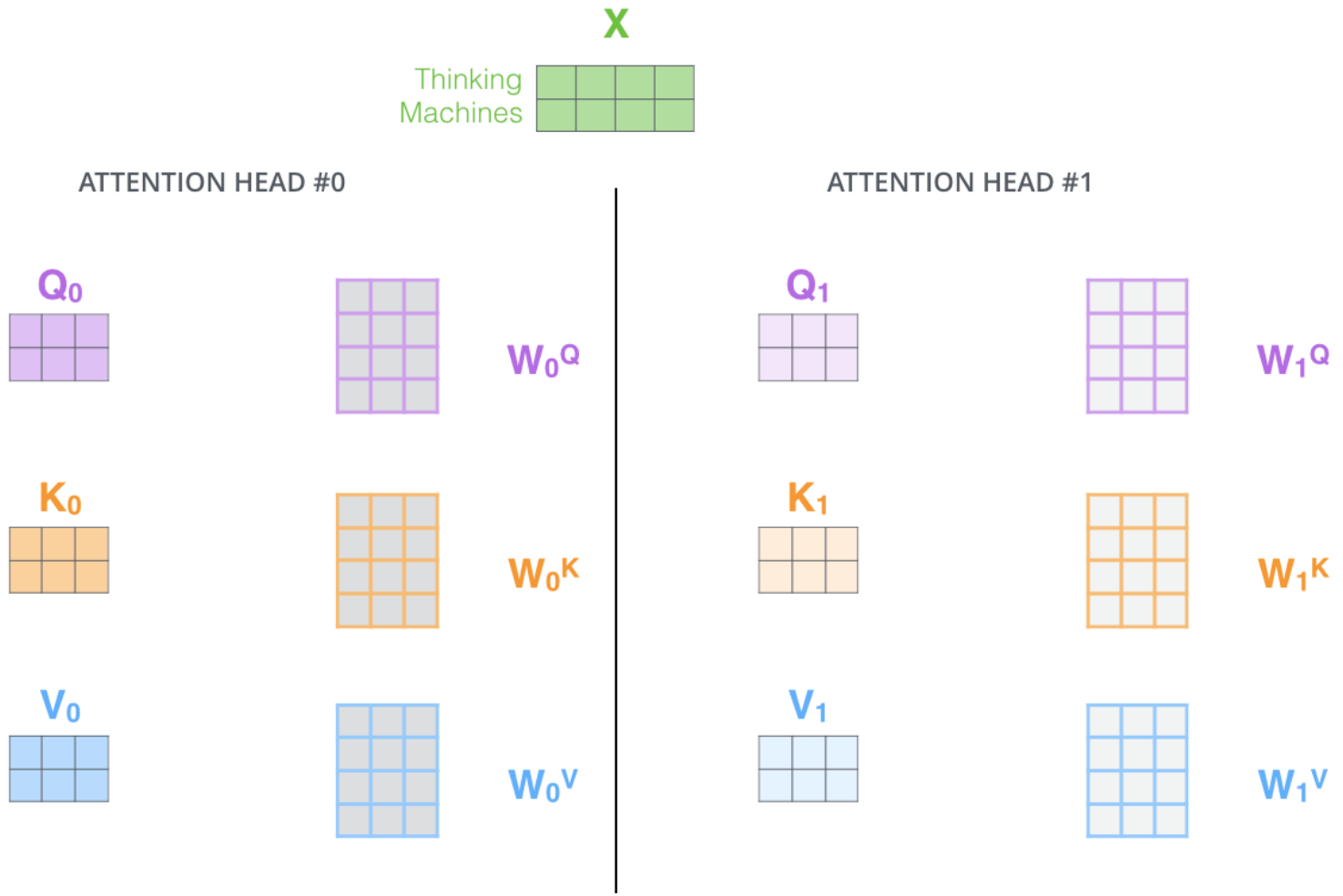
The Beast With Many Heads

The paper further refined the self-attention layer by adding a mechanism called “multi-headed” attention. This improves the performance of the attention layer in two ways:

1. It expands the model’s ability to focus on different positions. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the the actual word itself. It would be useful if we’re translating a sentence like “The animal didn’t cross the street because it was too tired”, we would want to know which word “it” refers to.

2. It gives the attention layer multiple “representation subspaces”. As we’ll see next, with multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.

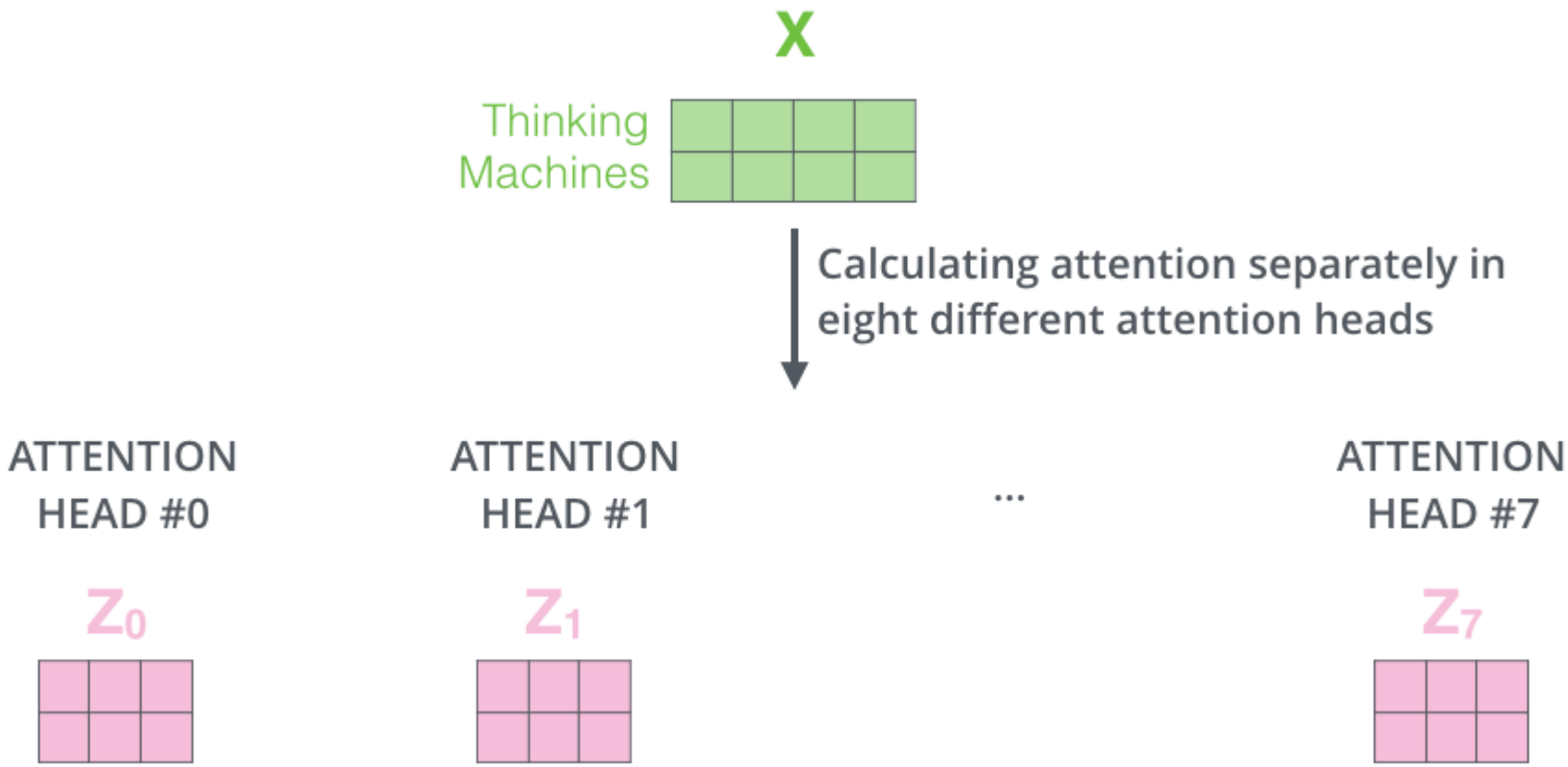
http://jalammar.github.io/illustrated-transformer/



With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

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

http://jalammar.github.io/illustrated-transformer/

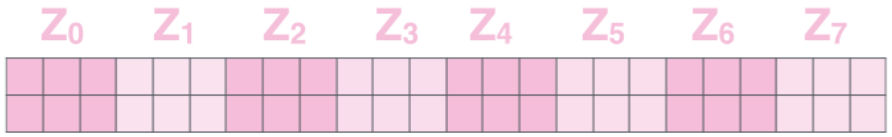


http://jalammar.github.io/illustrated-transformer/

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.

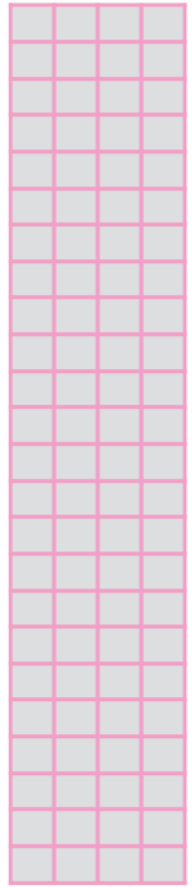
How do we do that? We concat the matrices then multiple them by an additional weights matrix WO.

1) Concatenate all the attention heads



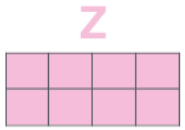
2) Multiply with a weight matrix W^O that was trained jointly with the model

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

=



http://jalammar.github.io/illustrated-transformer/

1) This is our input sentence*

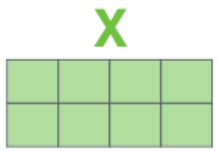
2) We embed each word*

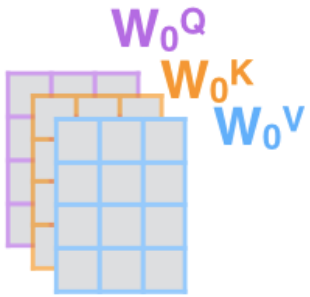
3) Split into 8 heads. We multiply X or R with weight matrices

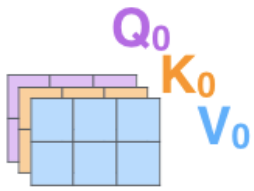
4) Calculate attention using the resulting Q/K/V matrices

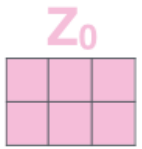
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

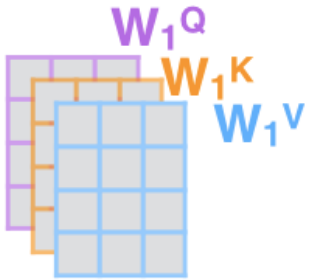
Thinking Machines

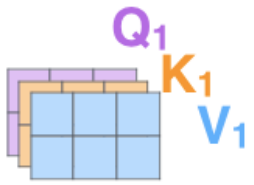


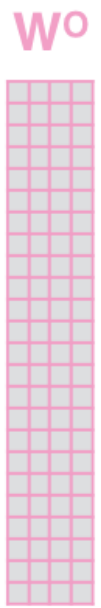


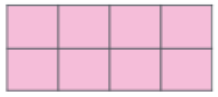




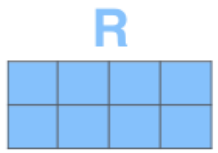


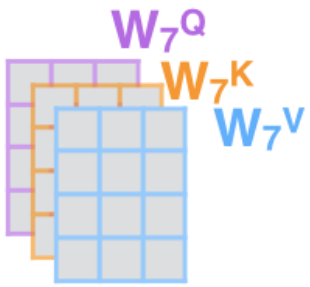


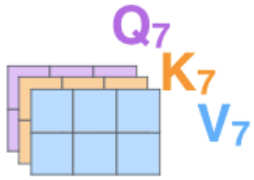


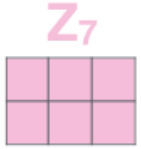


* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

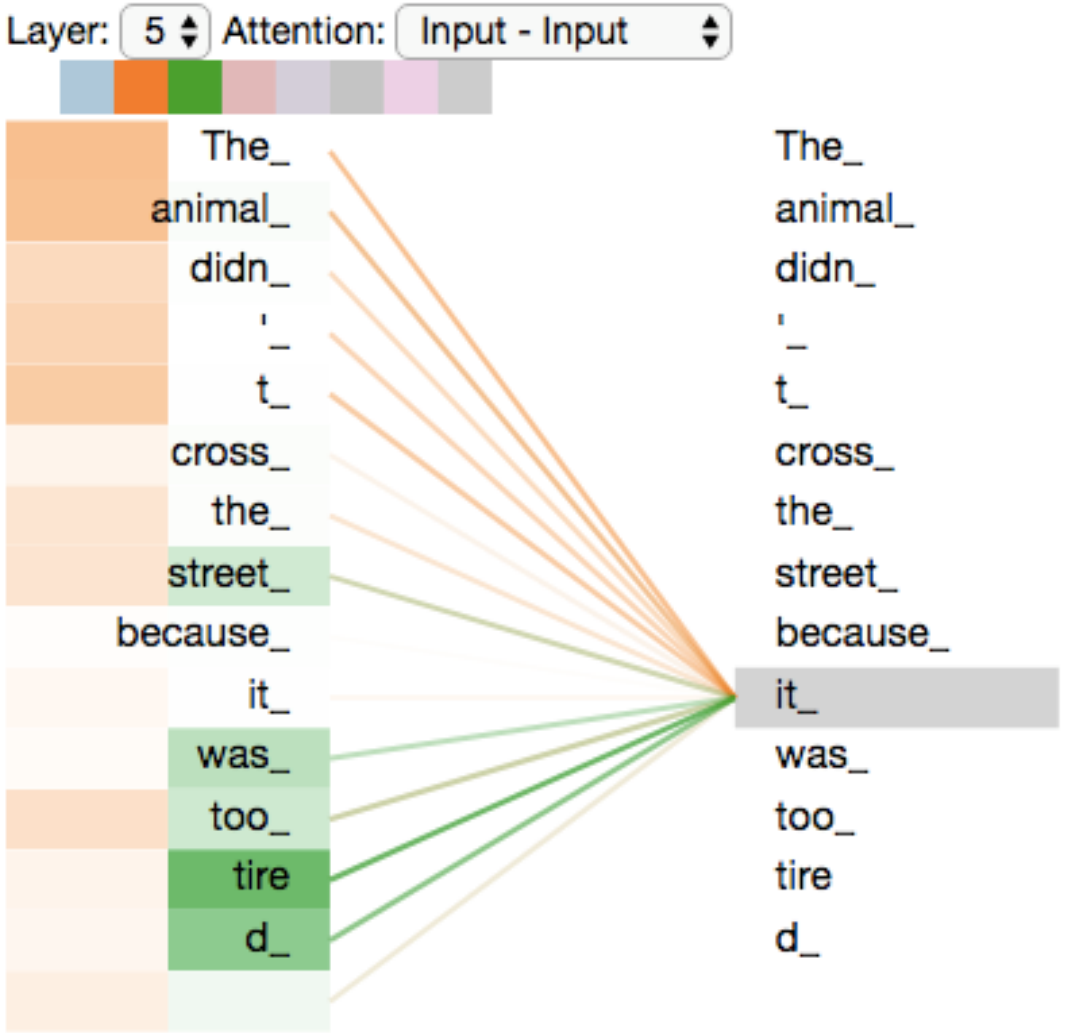








http://jalammar.github.io/illustrated-transformer/



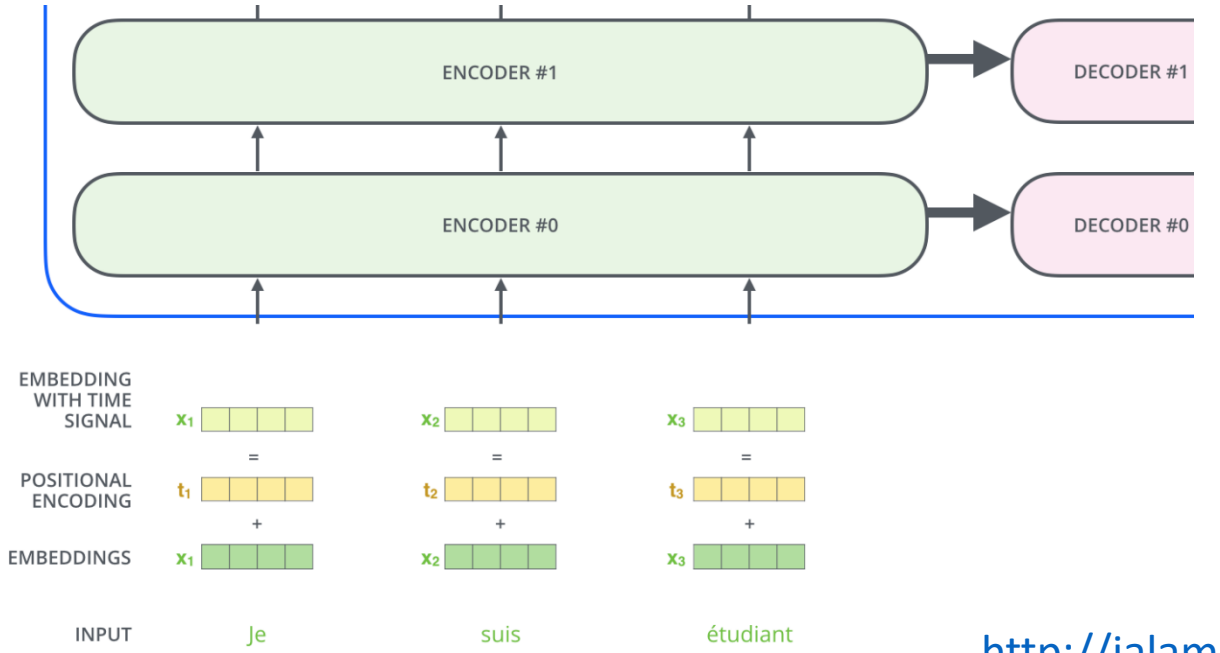
As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

http://jalammar.github.io/illustrated-transformer/

Representing The Order of The Sequence Using Positional Encoding

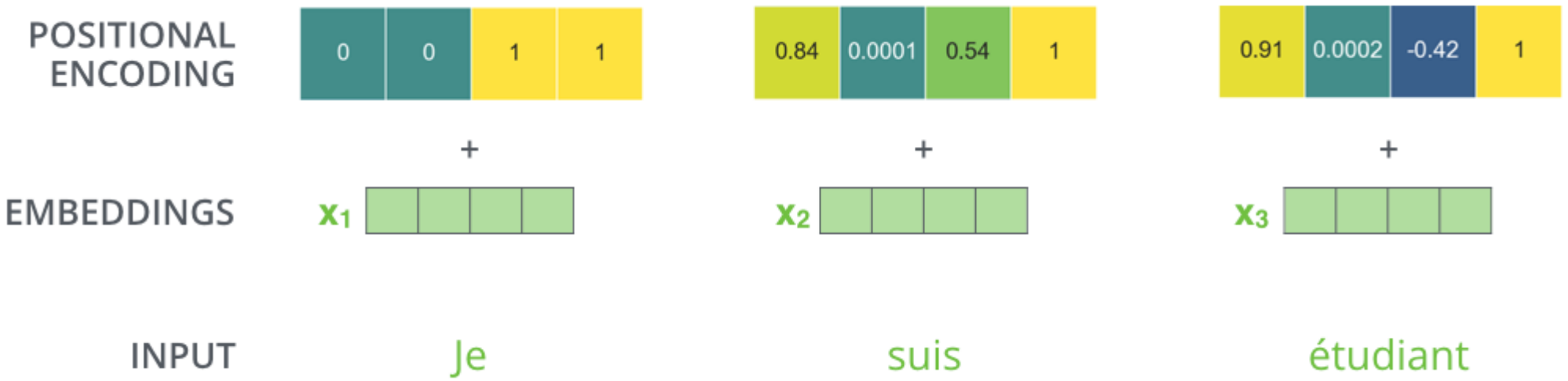
One thing that's missing from the model as we have described it so far is a way to account for the order of the words in the input sequence.

To address this, the transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence. The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.



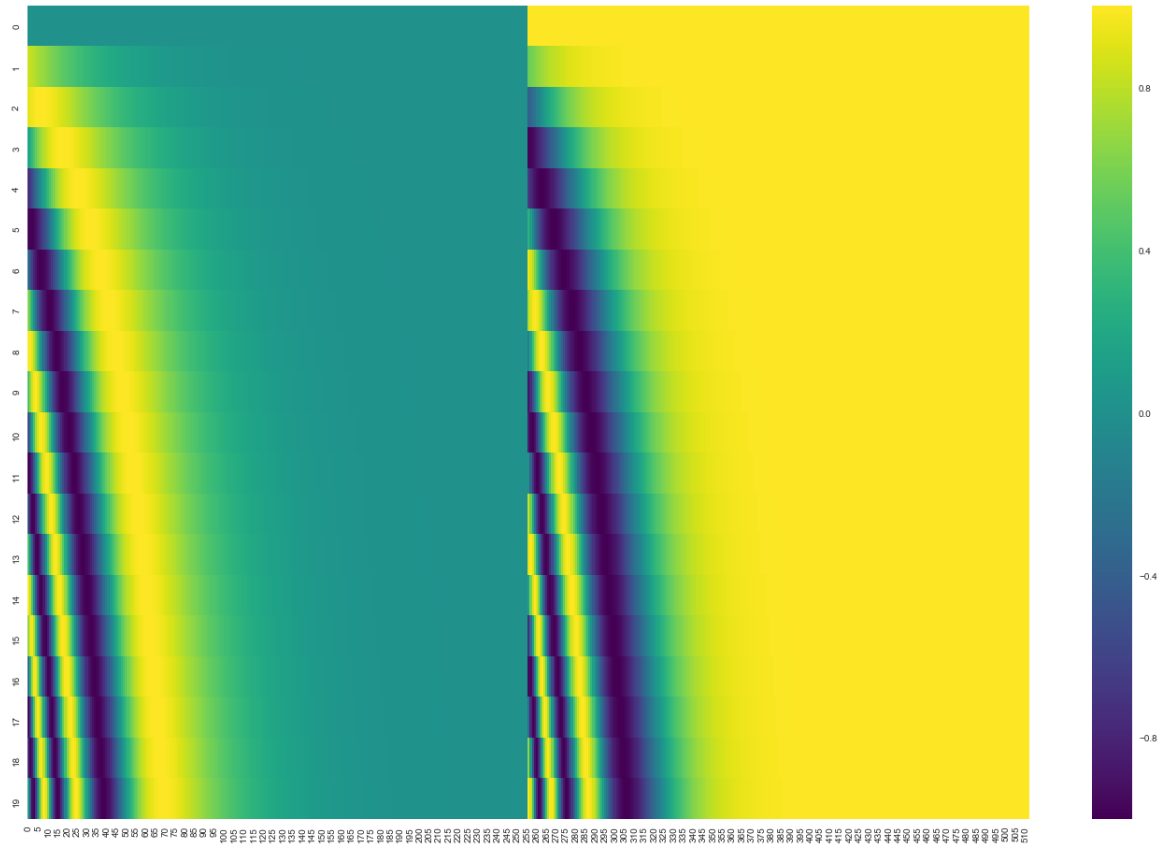
http://jalammar.github.io/illustrated-transformer/

If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like this:



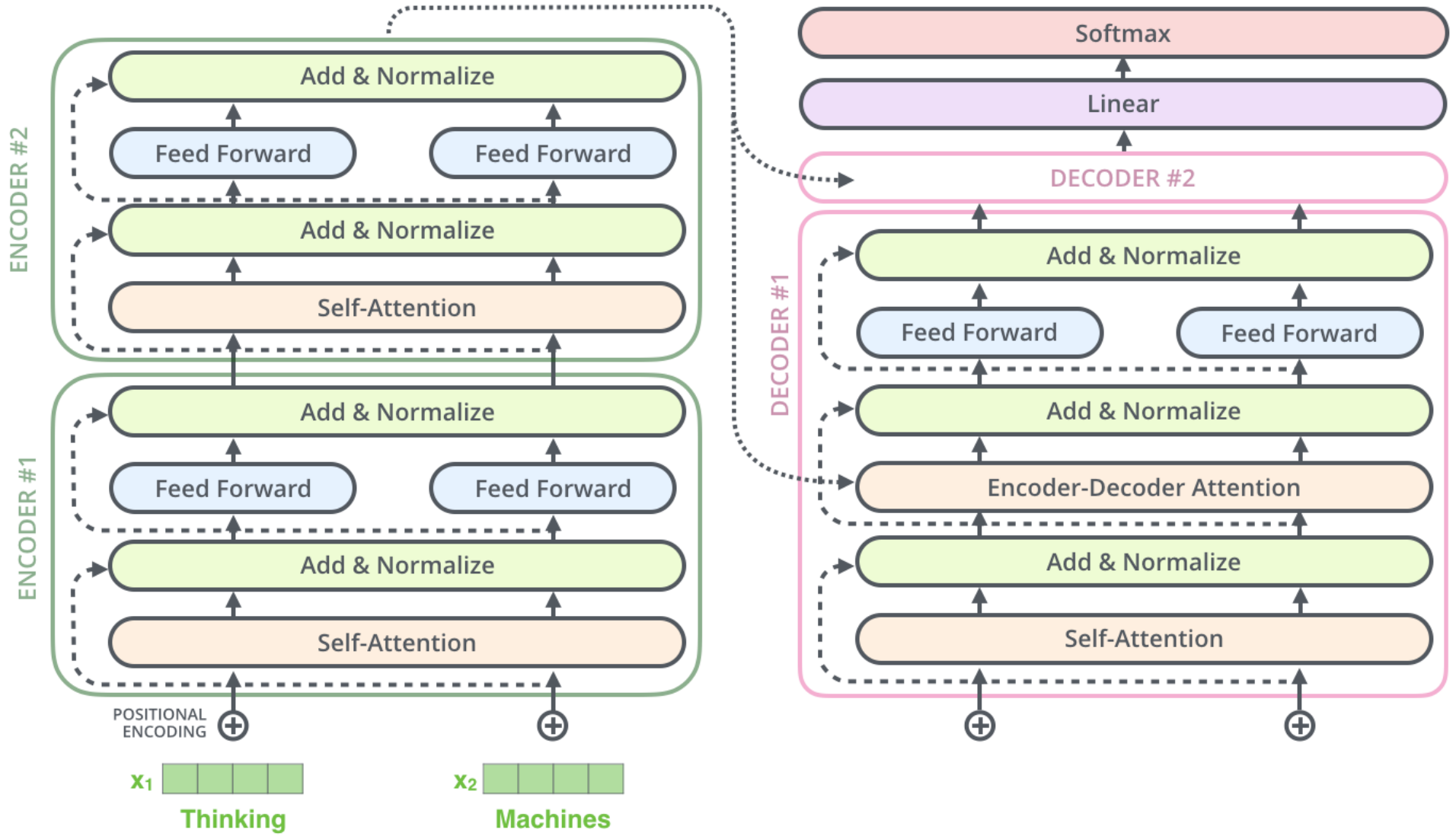
http://jalammar.github.io/illustrated-transformer/

In the following figure, each row corresponds the a positional encoding of a vector. So the first row would be the vector we'd add to the embedding of the first word in an input sequence. Each row contains 512 values – each with a value between 1 and -1. We've color-coded them so the pattern is visible.



A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in half down the center. That's because the values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). They're then concatenated to form each of the positional encoding vectors.

http://jalammar.github.io/illustrated-transformer/

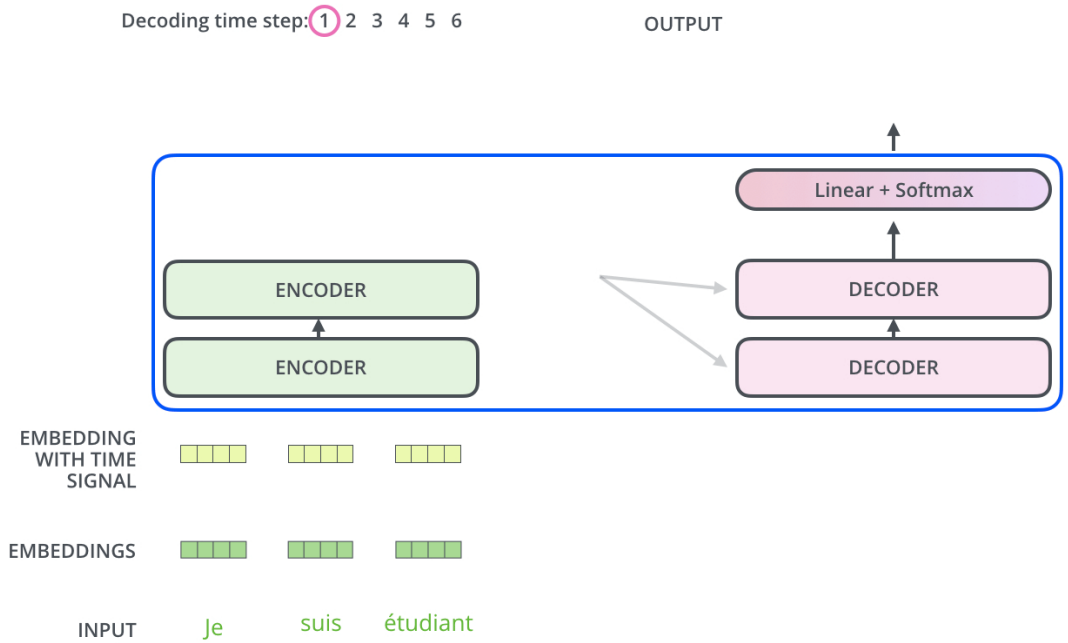


http://jalammar.github.io/illustrated-transformer/

The Decoder Side

Now that we've covered most of the concepts on the encoder side, we basically know how the components of decoders work as well. But let's take a look at how they work together.

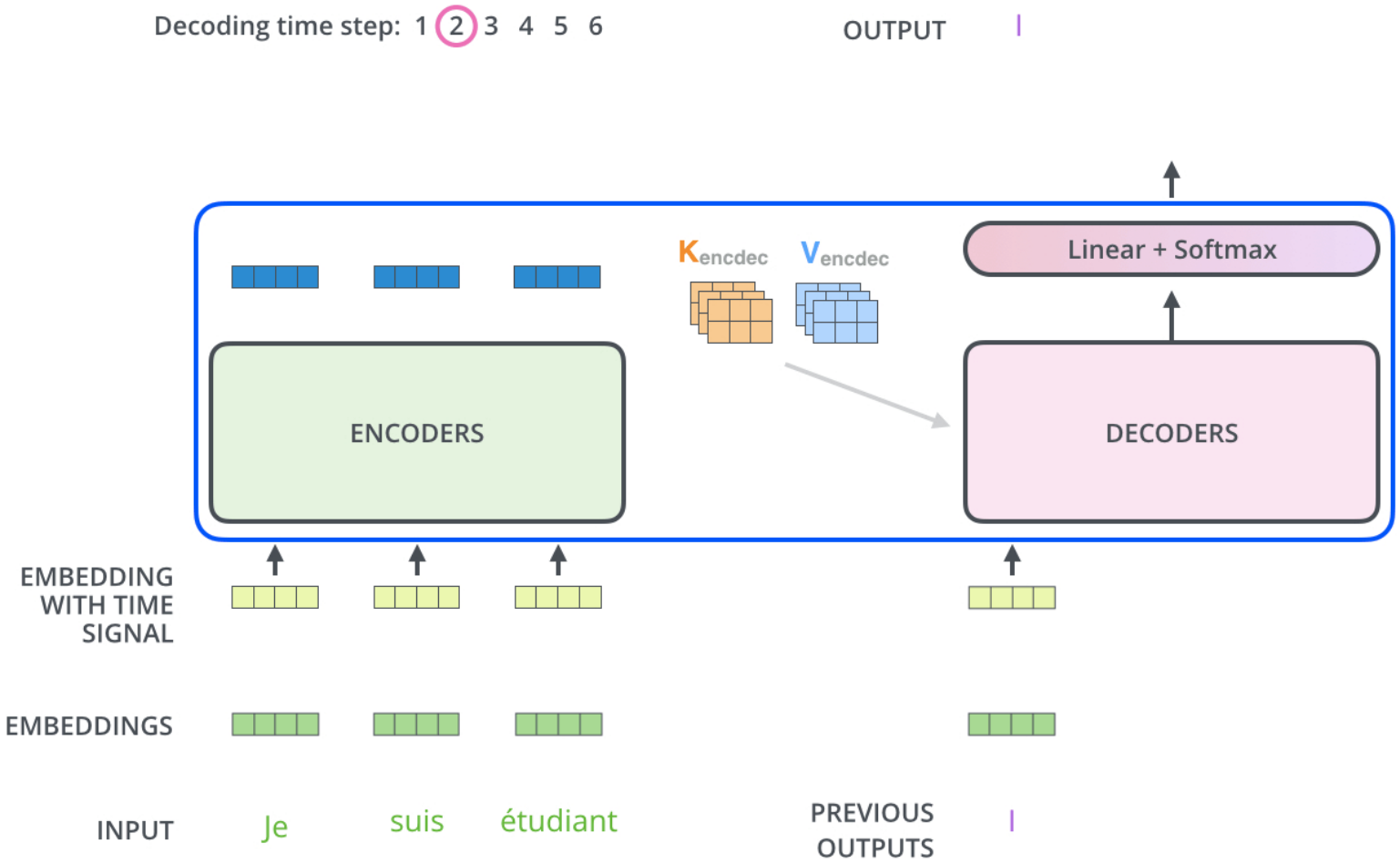
The encoder start by processing the input sequence. The output of the top encoder is then transformed into a set of attention vectors K and V. These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence:



After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

http://jalammar.github.io/illustrated-transformer/

The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.



http://jalammar.github.io/illustrated-transformer/

The self attention layers in the decoder operate in a slightly different way than the one in the encoder:

In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to `-inf`) before the softmax step in the self-attention calculation.

The “Encoder-Decoder Attention” layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

http://jalammar.github.io/illustrated-transformer/

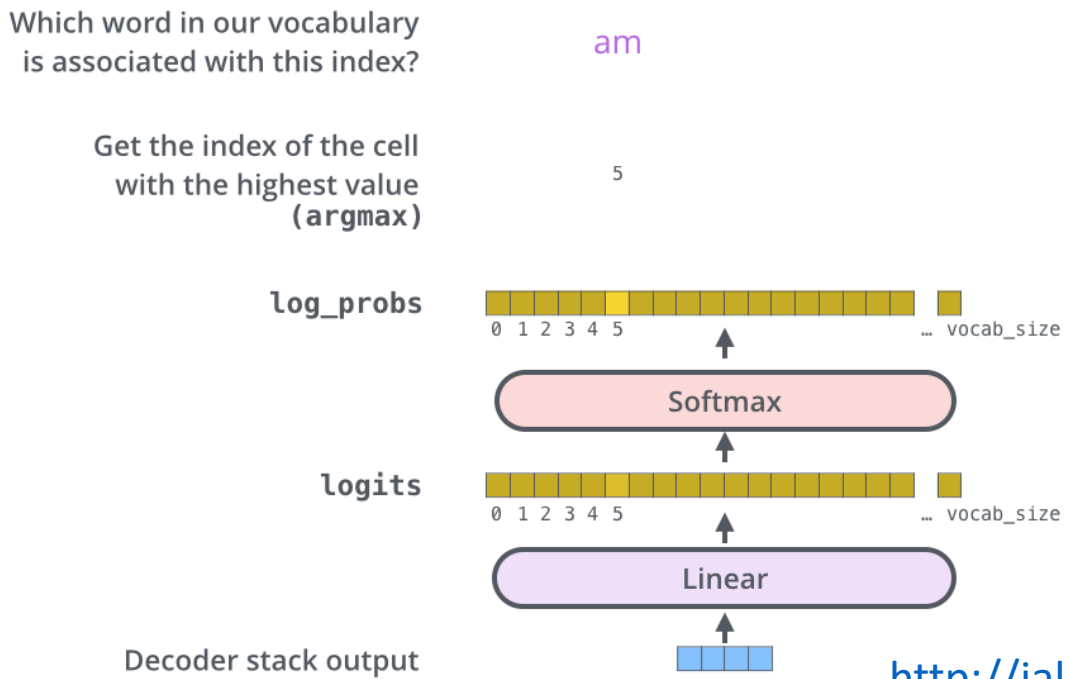
The Final Linear and Softmax Layer

The decoder stack outputs a vector of floats. How do we turn that into a word? That's the job of the final Linear layer which is followed by a Softmax Layer.

The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

Let's assume that our model knows 10,000 unique English words (our model's "output vocabulary") that it's learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.



http://jalammar.github.io/illustrated-transformer/

Training a transformer

Recap Of Training

Now that we've covered the entire forward-pass process through a trained Transformer, it would be useful to glance at the intuition of training the model.

During training, an untrained model would go through the exact same forward pass. But since we are training it on a labeled training dataset, we can compare its output with the actual correct output.

To visualize this, let's assume our output vocabulary only contains six words("a", "am", "i", "thanks", "student", and "<eos>" (short for 'end of sentence')).

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word "am"

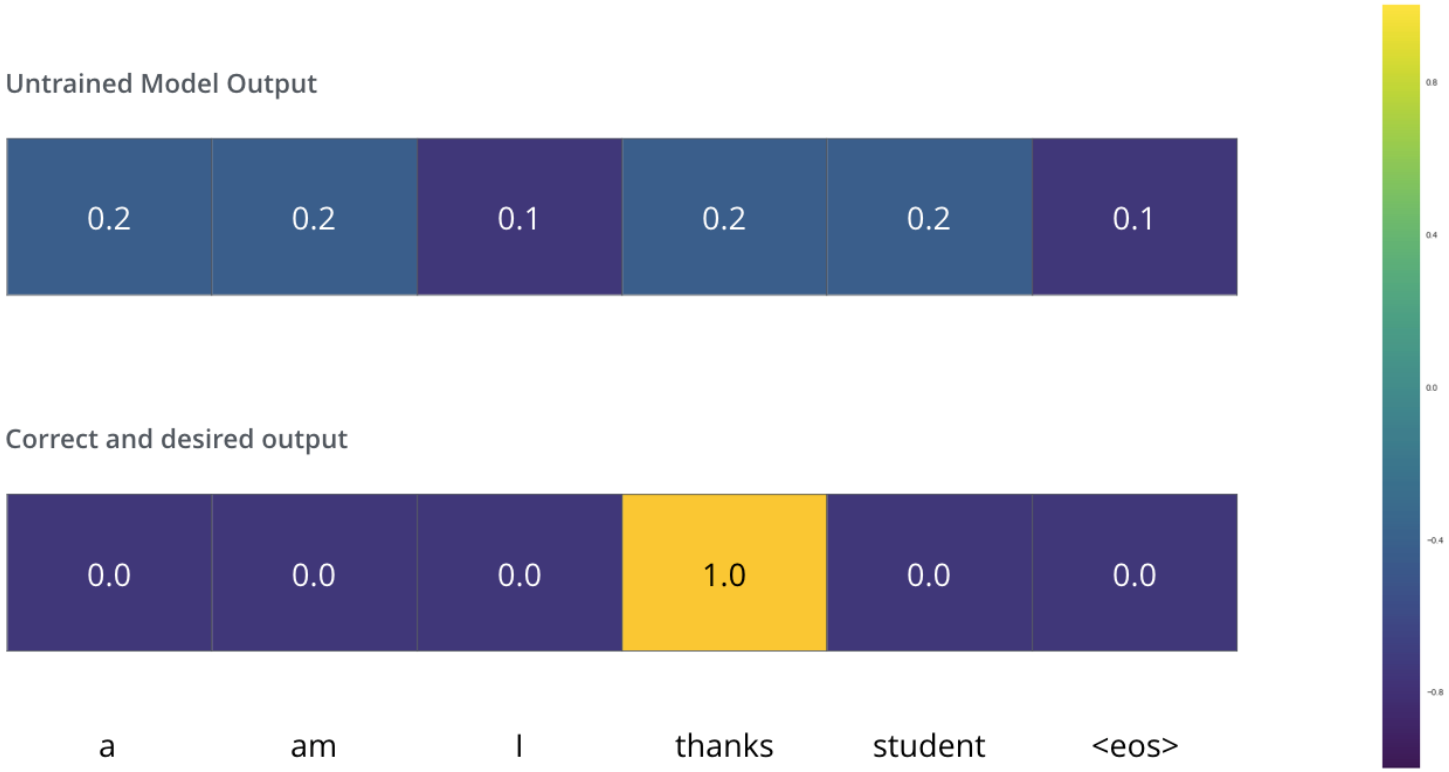
0.0	1.0	0.0	0.0	0.0	0.0

http://jalammar.github.io/illustrated-transformer/

The Loss Function

Say we are training our model. Say it's our first step in the training phase, and we're training it on a simple example – translating "merci" into "thanks".

What this means, is that we want the output to be a probability distribution indicating the word "thanks". But since this model is not yet trained, that's unlikely to happen just yet.

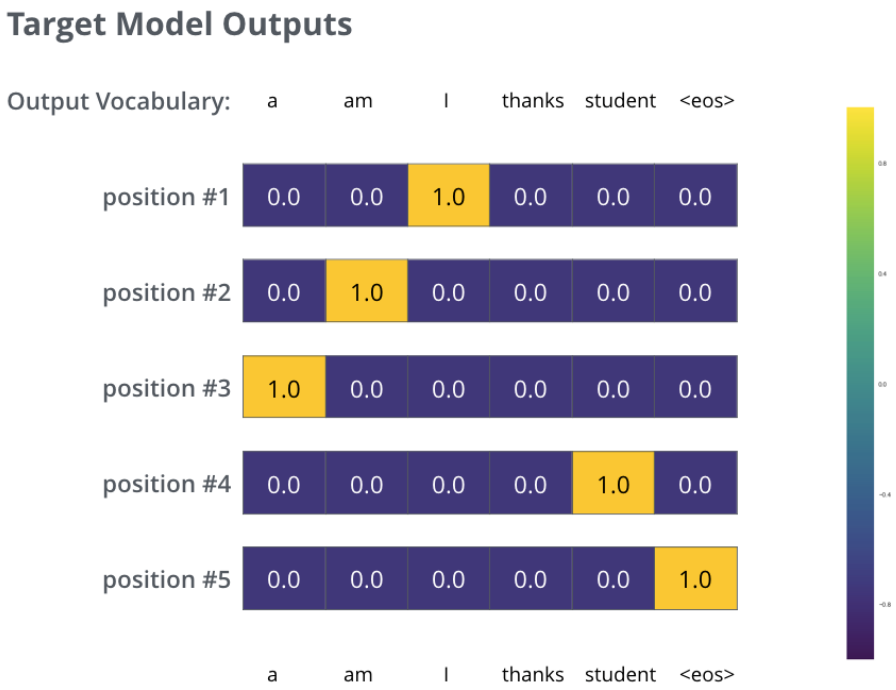


http://jalammar.github.io/illustrated-transformer/

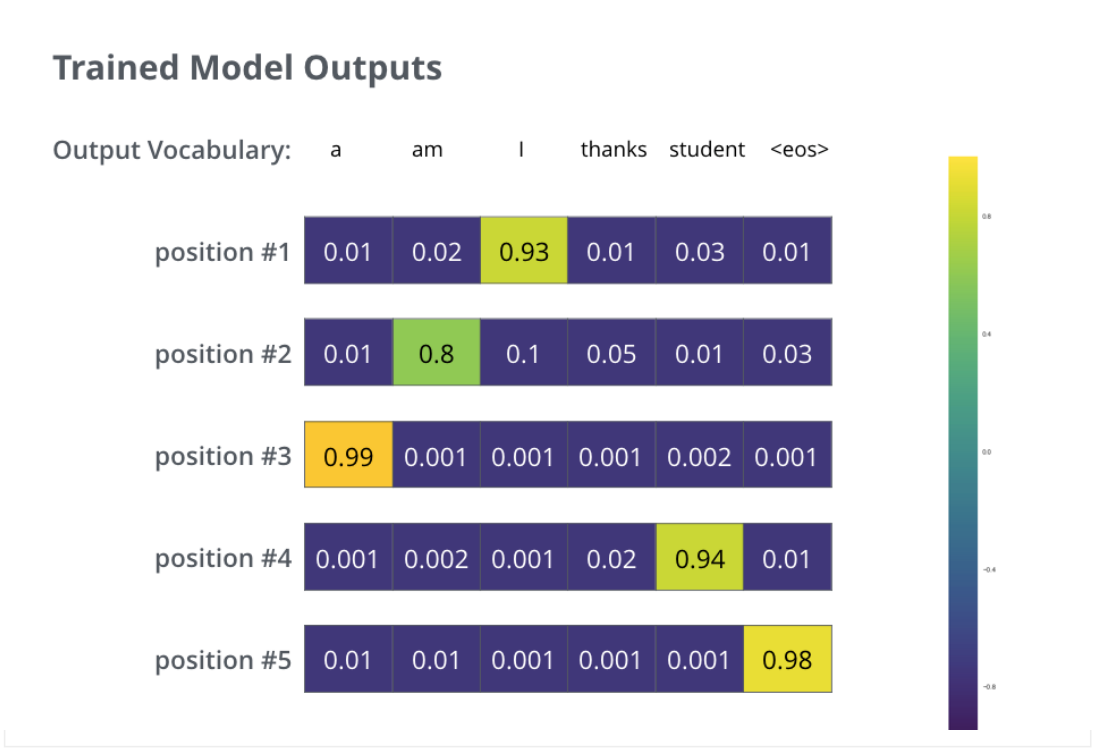
How do you compare two probability distributions? We simply subtract one from the other. For more details, look at cross-entropy and Kullback–Leibler divergence.

But note that this is an oversimplified example. More realistically, we'll use a sentence longer than one word. For example – input: "je suis étudiant" and expected output: "i am a student". What this really means, is that we want our model to successively output probability distributions where:

- Each probability distribution is represented by a vector of width vocab_size (6 in our toy example, but more realistically a number like 30,000 or 50,000)
- The first probability distribution has the highest probability at the cell associated with the word "i"
- The second probability distribution has the highest probability at the cell associated with the word "am"
- And so on, until the fifth output distribution indicates '`<end of sentence>`' symbol, which also has a cell associated with it from the 10,000 element vocabulary.



http://jalammar.github.io/illustrated-transformer/



Hopefully upon training, the model would output the right translation we expect. Of course it's no real indication if this phrase was part of the training dataset (see: cross validation). Notice that every position gets a little bit of probability even if it's unlikely to be the output of that time step -- that's a very useful property of softmax which helps the training process.

Now, because the model produces the outputs one at a time, we can assume that the model is selecting the word with the highest probability from that probability distribution and throwing away the rest. That's one way to do it (called greedy decoding). Another way to do it would be to hold on to, say, the top two words (say, 'I' and 'a' for example), then in the next step, run the model twice: once assuming the first output position was the word 'I', and another time assuming the first output position was the word 'a', and whichever version produced less error considering both positions #1 and #2 is kept. We repeat this for positions #2 and #3…etc. This method is called "beam search", where in our example, beam_size was two (meaning that at all times, two partial hypotheses (unfinished translations) are kept in memory), and top_beams is also two (meaning we'll return two translations). These are both hyperparameters that you can experiment with.

http://jalammar.github.io/illustrated-transformer/