Economics 2355: Attention is All You Need

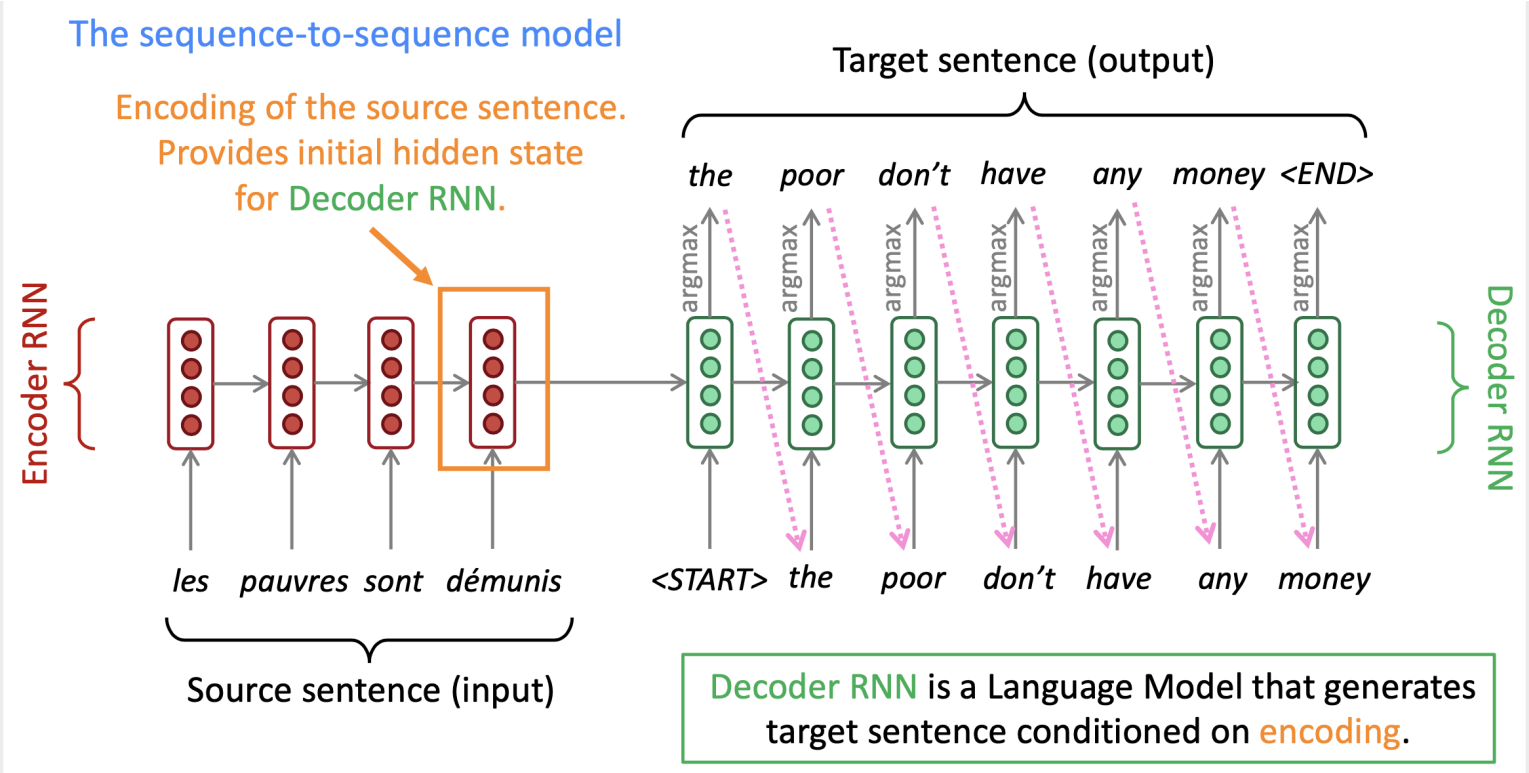
Melissa Dell

Harvard University

March 2021

Attention

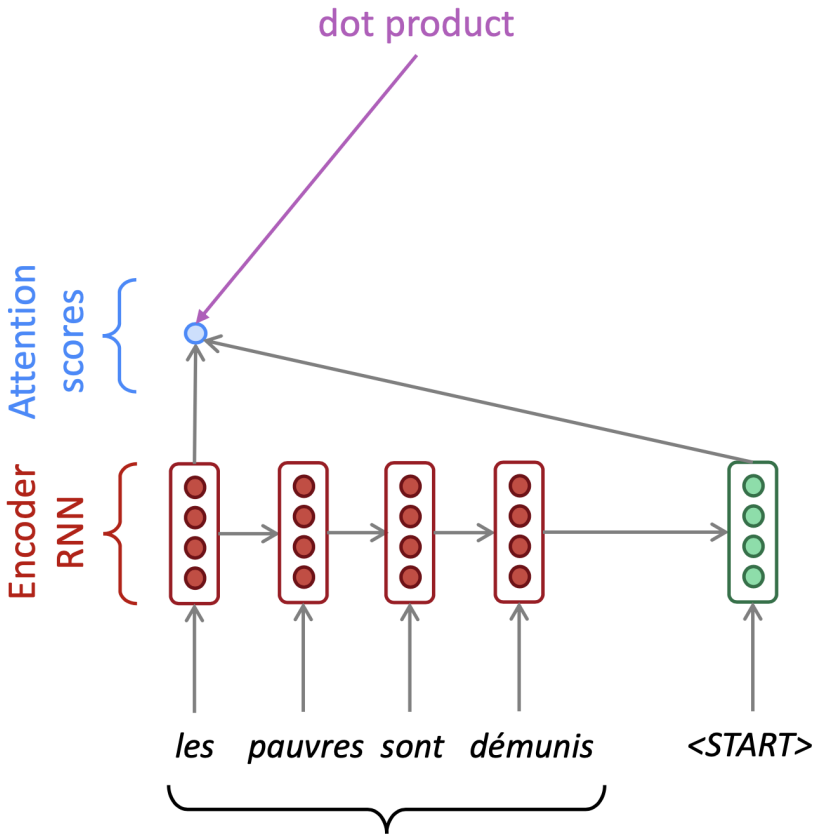
Recall that last class we introduced attention, which was developed for machine translation with seq2seq



Attention

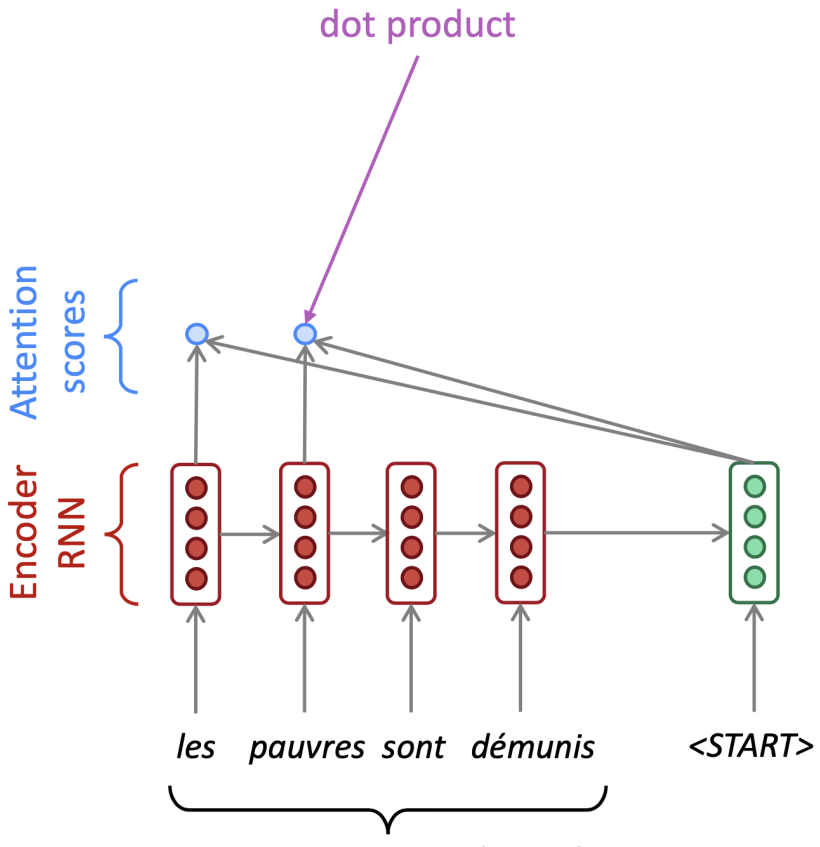
- At each part of the decoder, use a direct connection to the encoder to focus on a specific part of the source sentence
- Effectively allows us to pass information from a variably sized encoder model to the decoder, in a learnable way
- Attention has transformed NLP

Attention



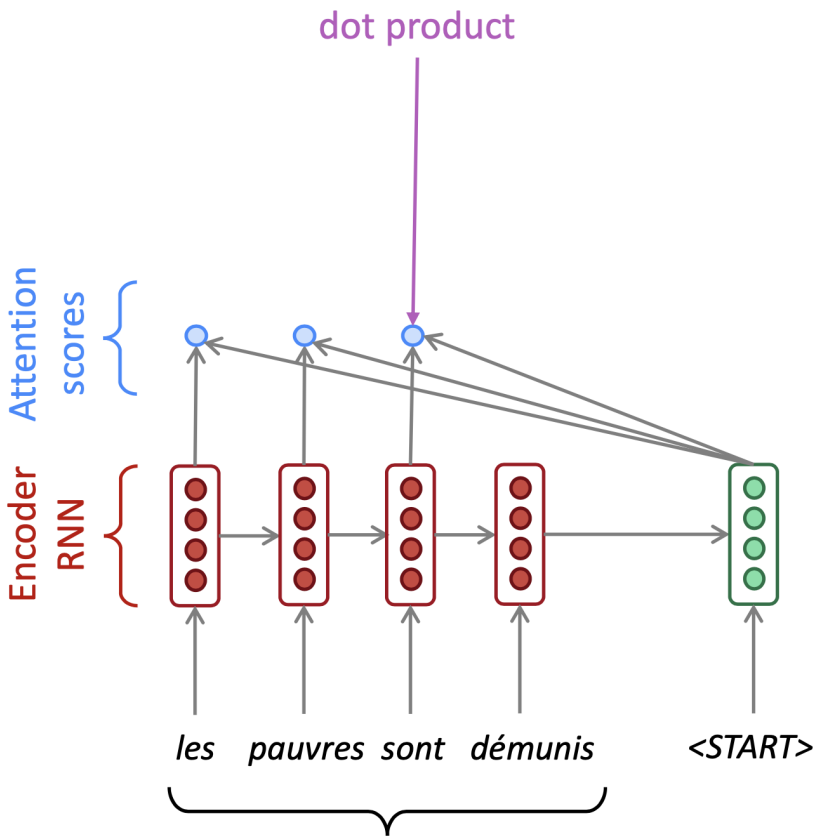
Stanford Linguistics 284

Attention



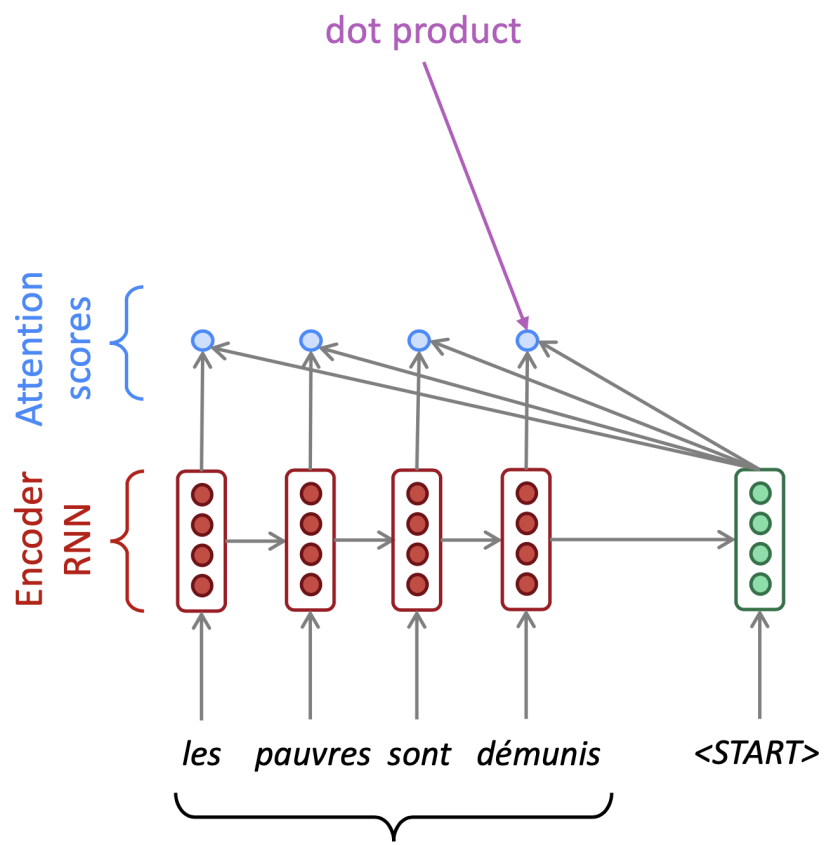
Stanford Linguistics 284

Attention



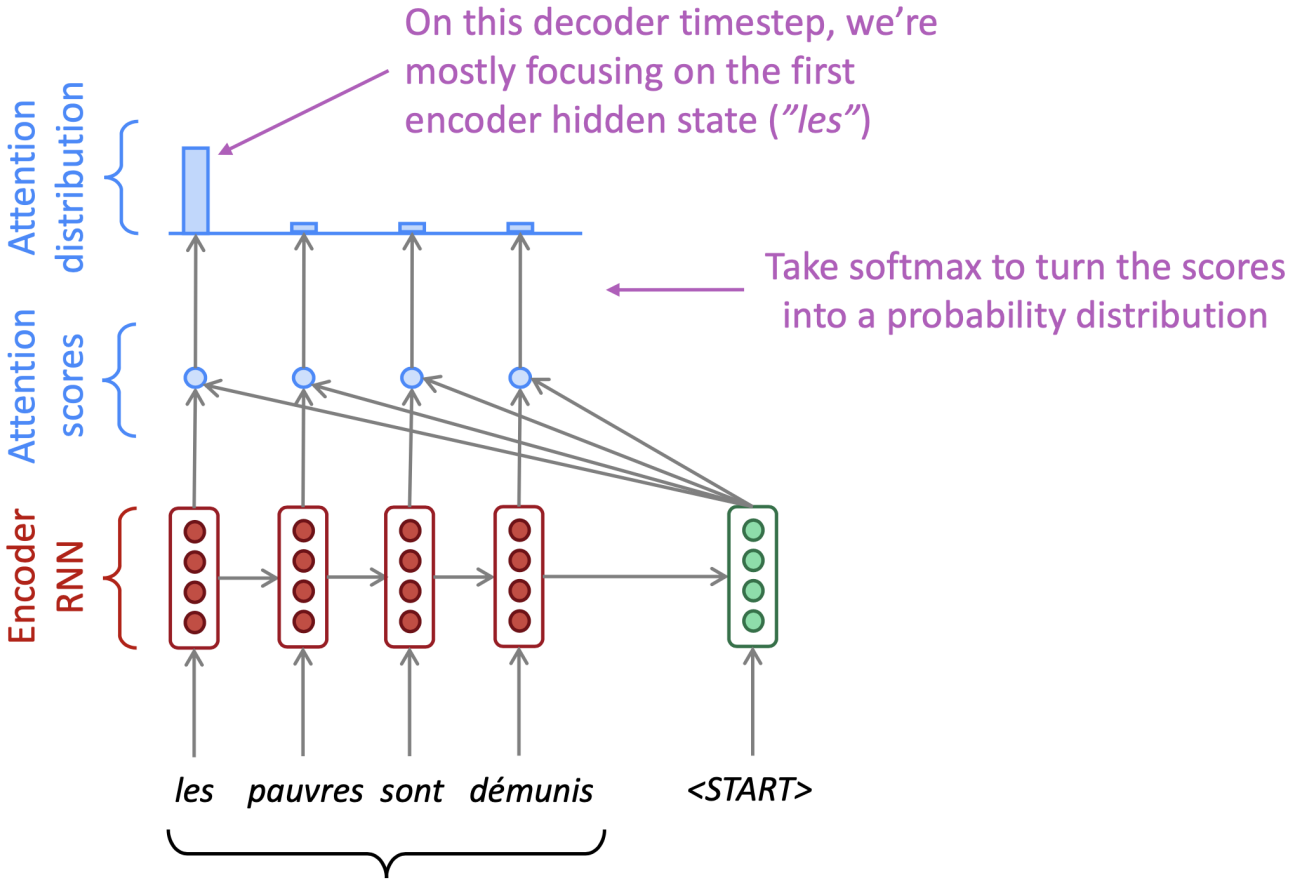
Stanford Linguistics 284

Attention



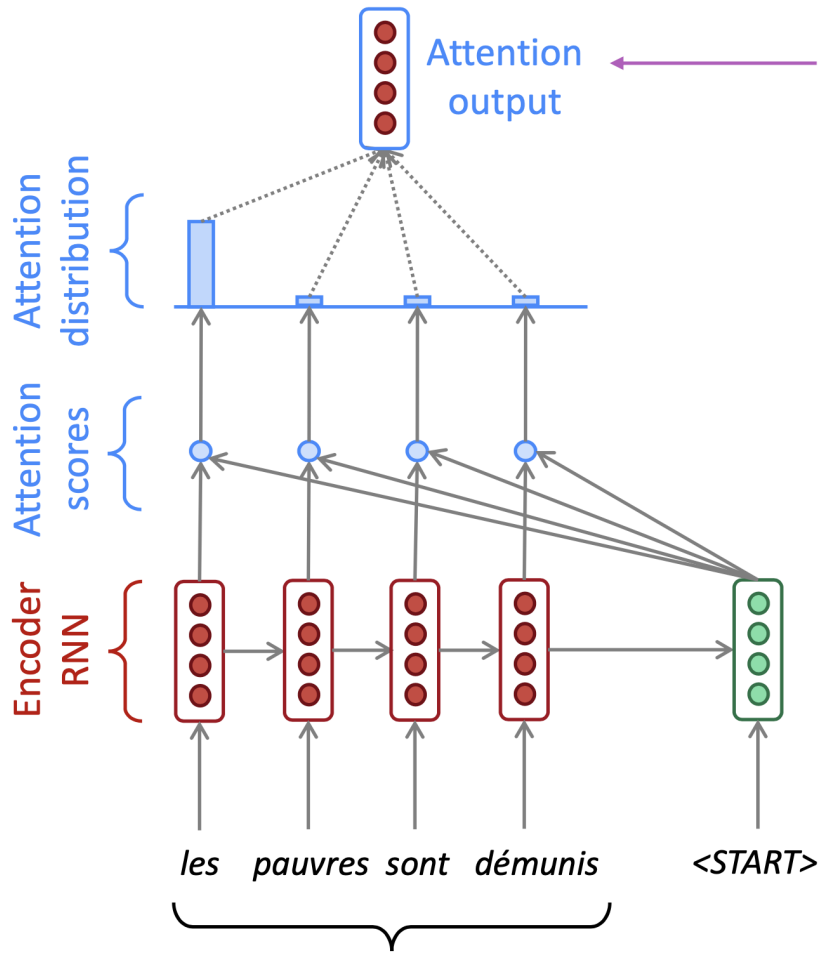
Stanford Linguistics 284

Attention



Stanford Linguistics 284

Attention

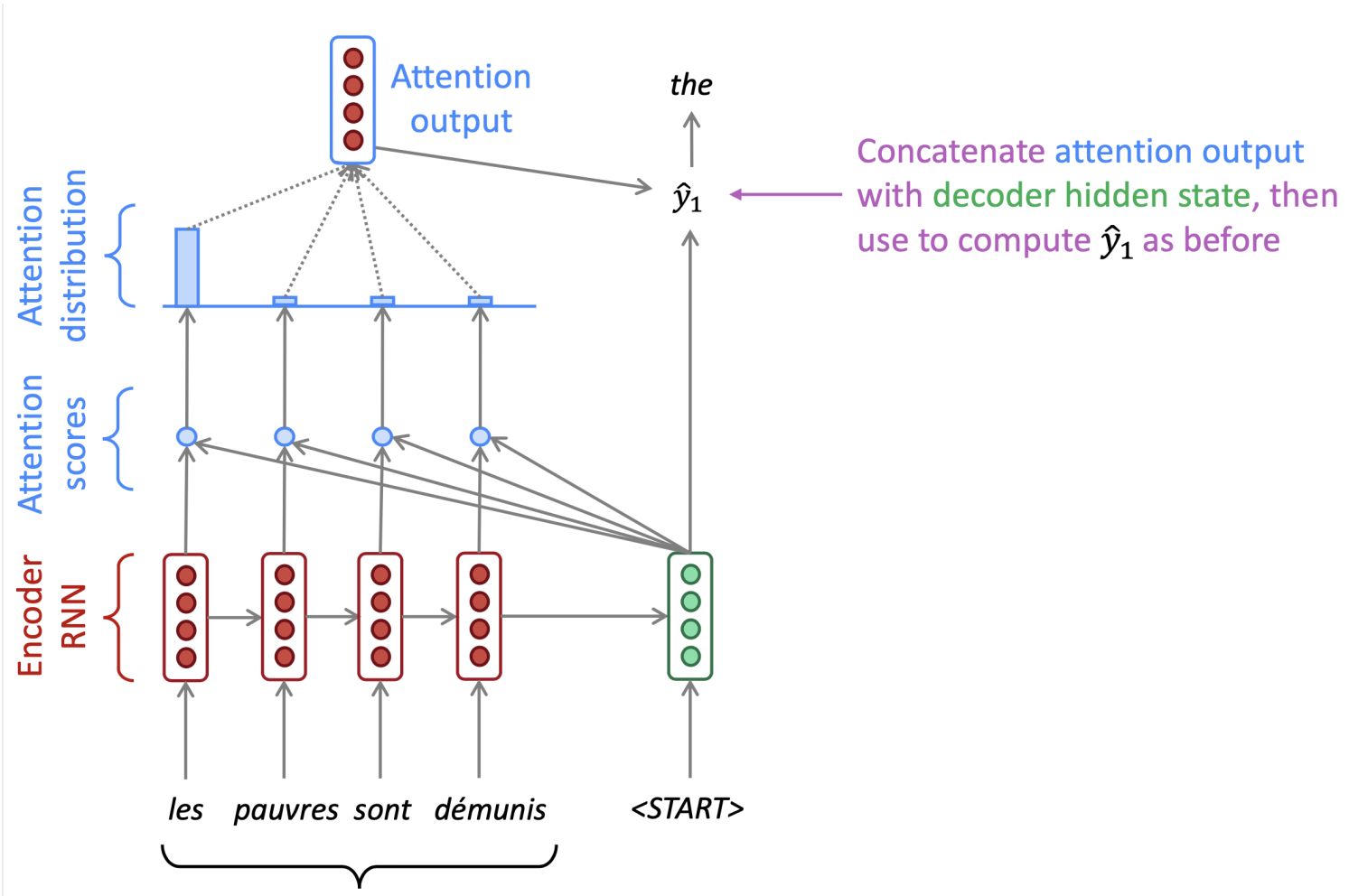


Use the attention distribution to take a **weighted sum** of the encoder hidden states.

The attention output mostly contains information the hidden states that received high attention.

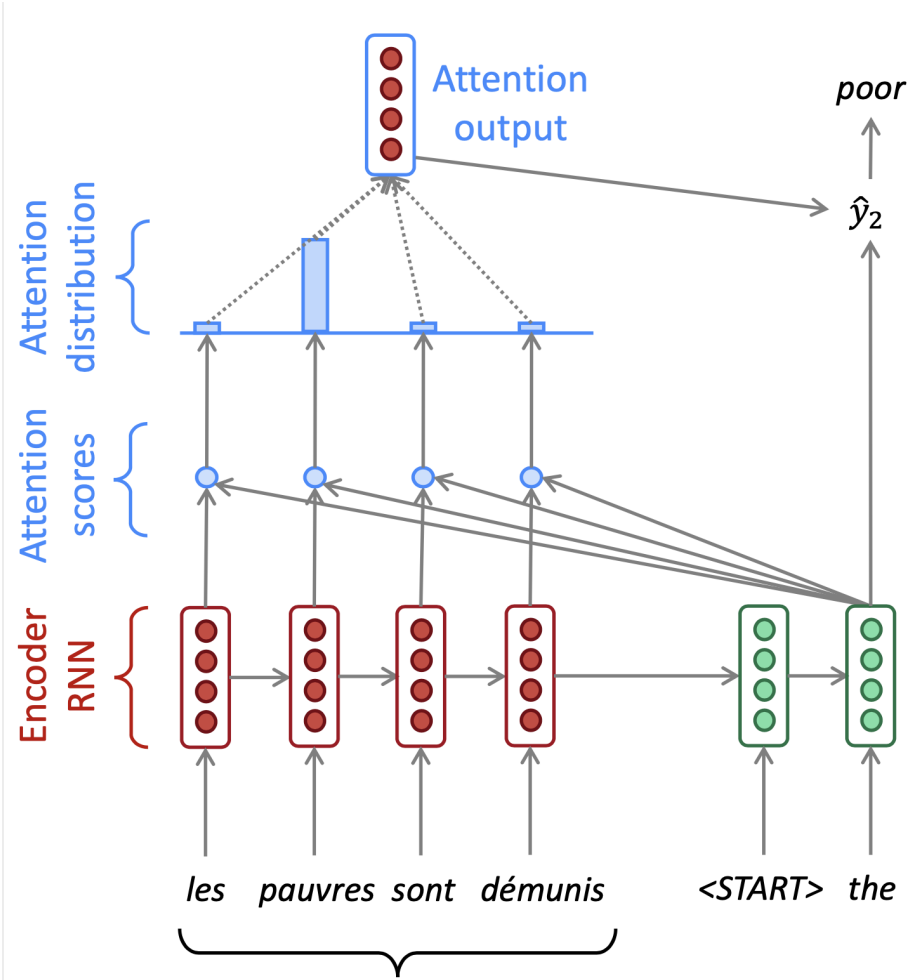
Stanford Linguistics 284

Attention



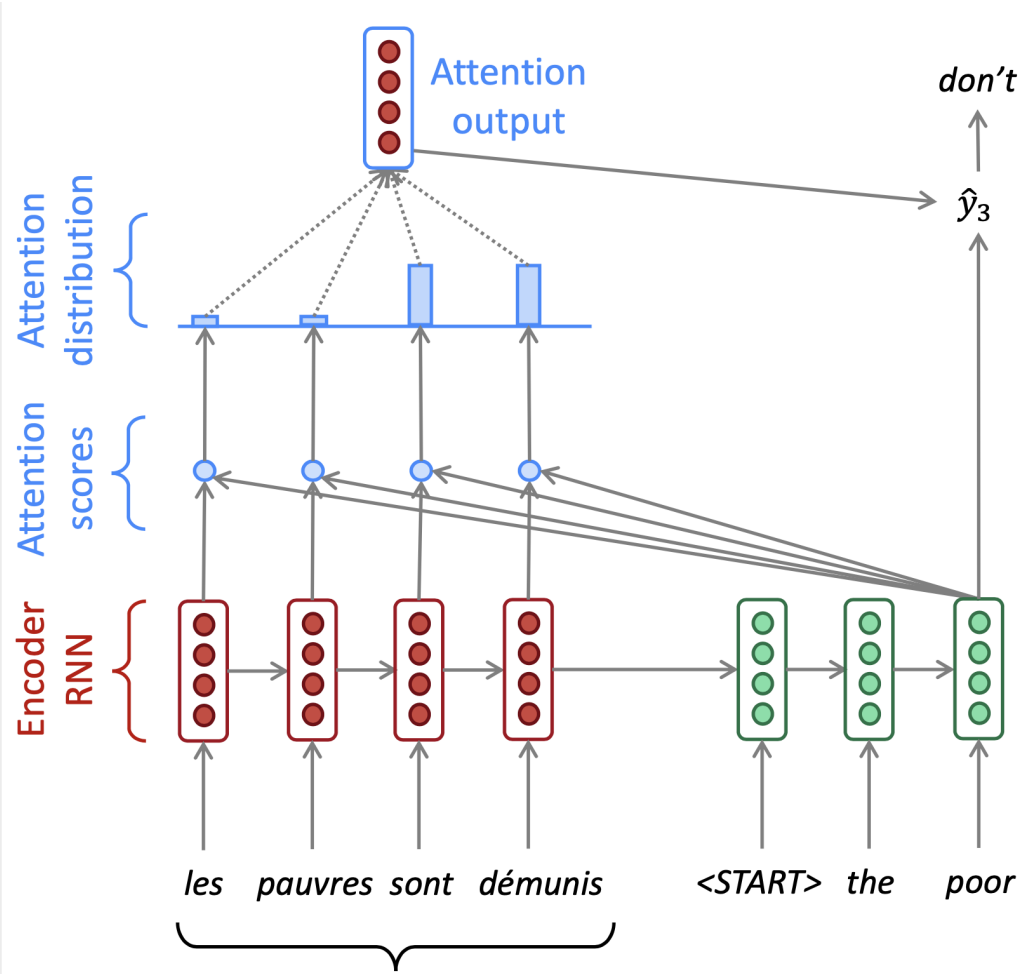
Stanford Linguistics 284

Attention



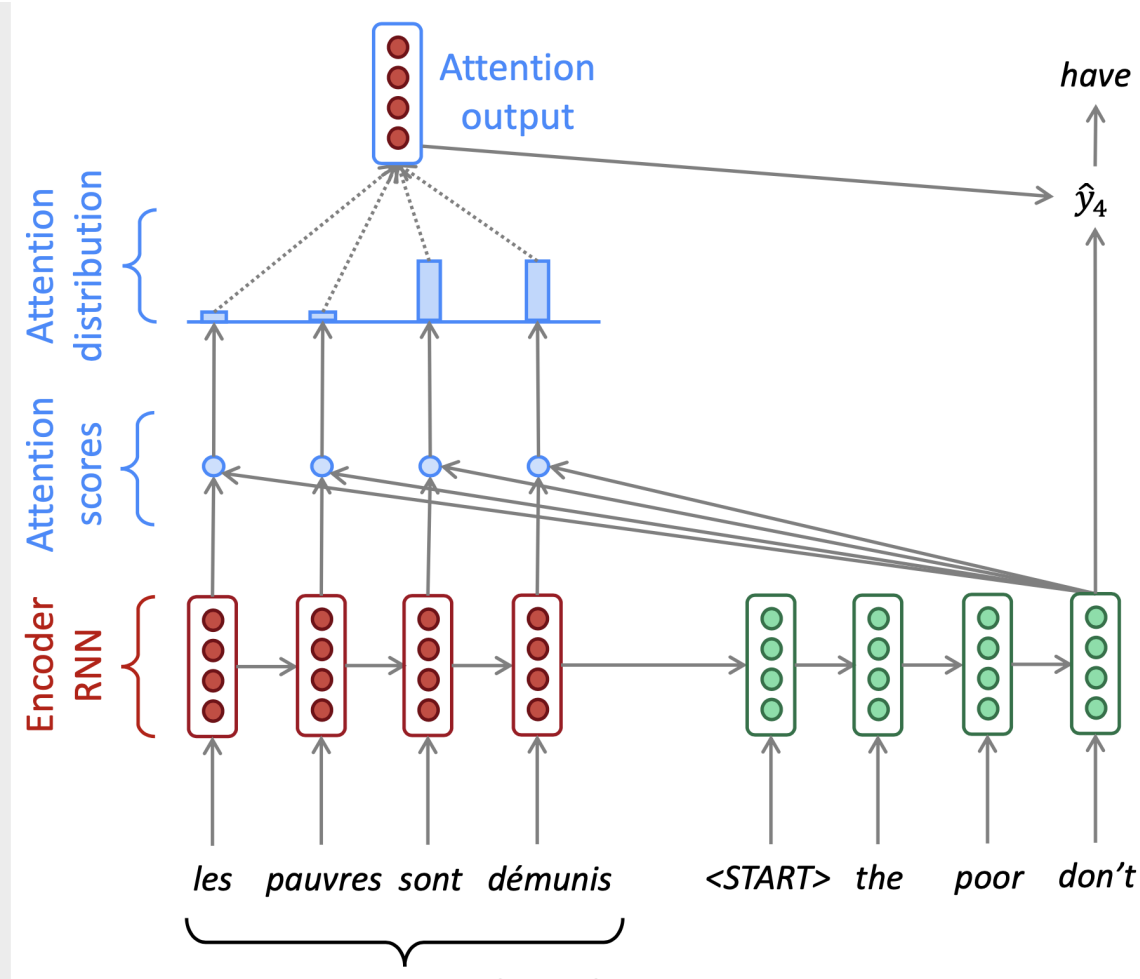
Stanford Linguistics 284

Attention



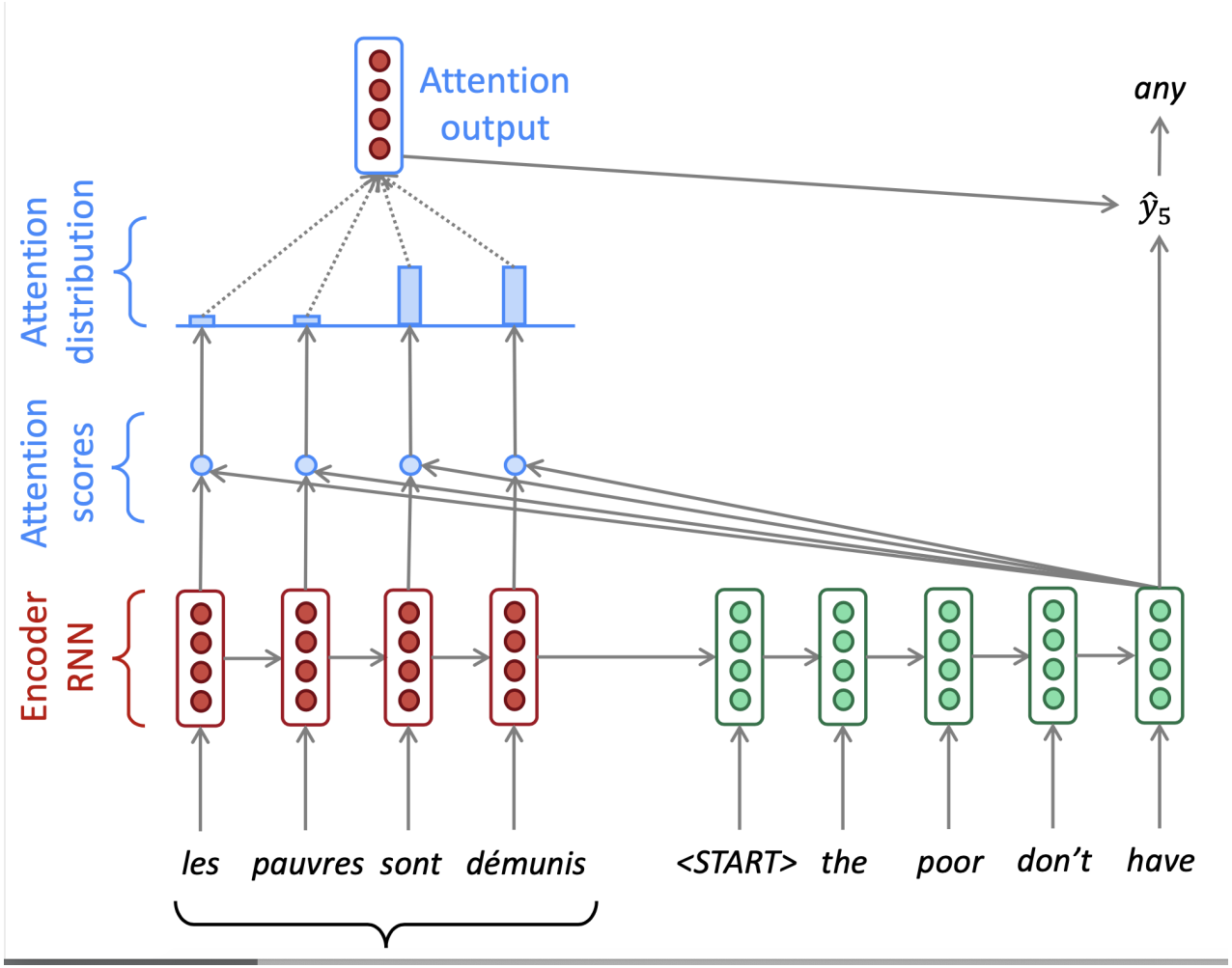
Stanford Linguistics 284

Attention



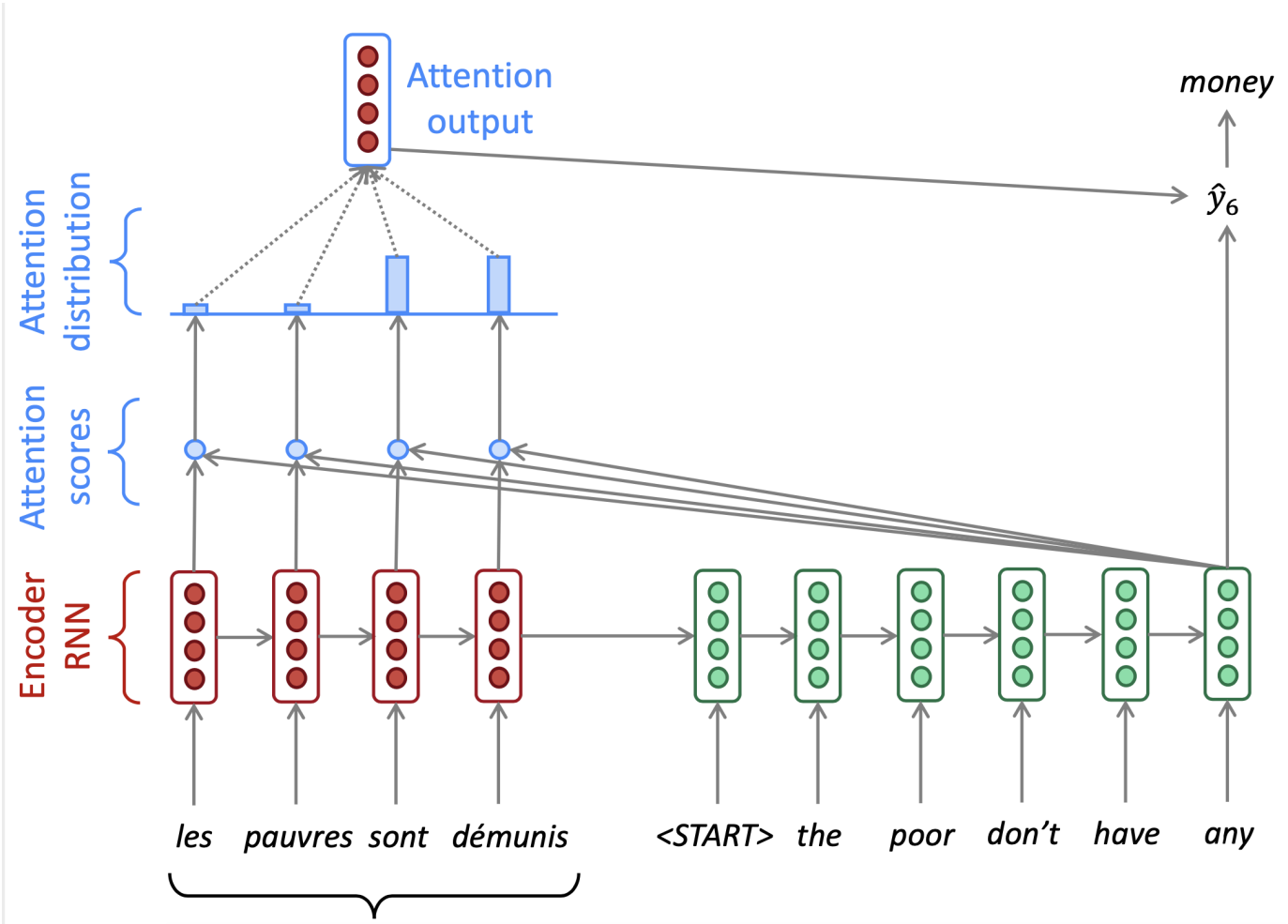
Stanford Linguistics 284

Attention



Stanford Linguistics 284

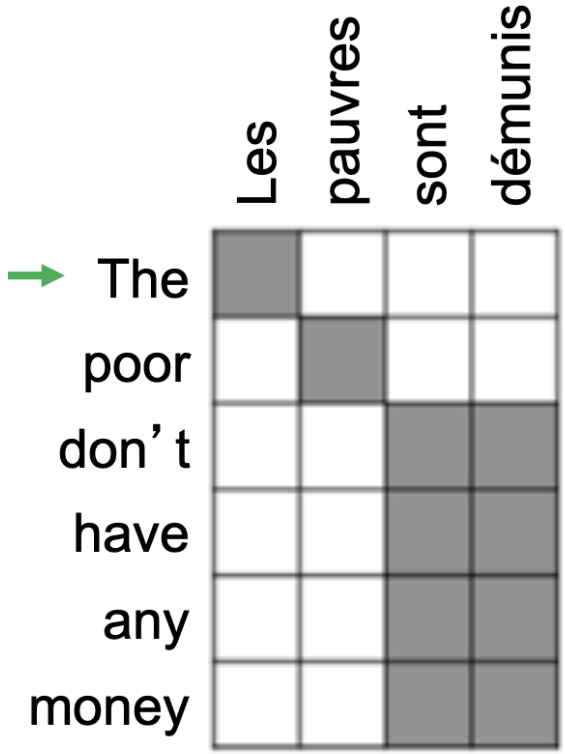
Attention



Stanford Linguistics 284

Attention and Alignment

What does the attention distribution look like?



Stanford Linguistics 284

We're not training the model to give us alignment, it learns this through attention. Don't need labels for alignment, which are very costly to produce.

Other Advantages of Attention

- Solves the bottleneck problem
- Helps with vanishing gradients by providing a connection to far away states (another **recurring theme** of this course, when it comes to discussing major advances in the literature; goes back to the fact that at its core, DL is linear algebra and MV calculus)

Attention and Deep Learning

- **Attention definition:** Given a set of vector *values* and a vector *query*, attention computes a weighted sum of the values that is dependent on the query
  - The query tells us where to focus in creating a weighted sum of the values
- In short, attention creates a *fixed-size* representation of an arbitrarily sized set of representations, given other representations (the query)

Attention Variants

Suppose $v_1 \ldots v_n$ (i.e. our encoder states) are values and q is the query (i.e. our first decoder hidden state)

Attention always requires computing an attention output a from the attention scores e:

$$\alpha = softmax(e)$$

$$a = \sum_{i=1}^{N} \alpha_i v_i$$

Several ways to compute e

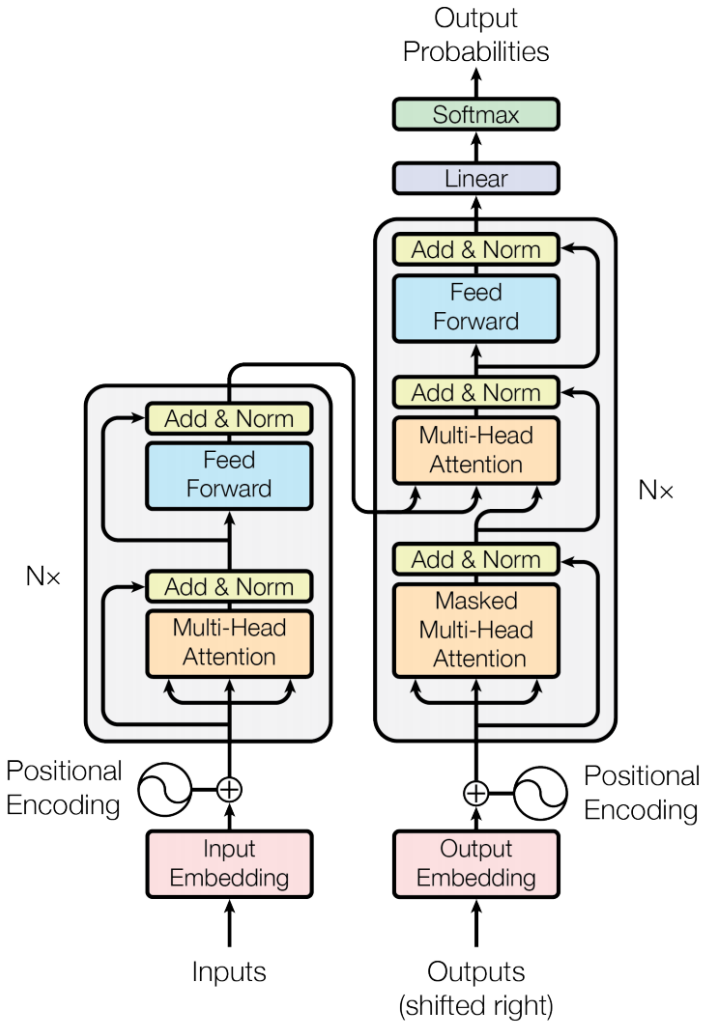
Attention Variants

1. *Dot product attention*: $e_i = q^T v_i$
   - Need dimensions of q and v to be the same
2. *Multiplicative attention*: $e_i = q^T W v_i$
   - W is a weight matrix, dimensions reflect the dimensionality of q and v
3. *Additive attention*: $e_i = u^T tanh(W_1 v_i + W_2 q)$
   - W_1 and W_2 are weight matrices and u is a vector of weights

Outline

Outline

The Transformer



Vaswani et al., 2017

The Transformer is a seq2seq model based entirely on attention ("Attention is All You Need")

The Transformer

- The paper has been enormously influential in NLP, sometimes referred to as the ImageNet/AlexNet moment of NLP
- The original transformer is about machine translation, but we will see subsequently that it can form the basis for many NLP model architectures
- One advantage of being such an influential paper is that there are many efforts to explain it, some of which are very good. I will use visualizations today from the Illustrated Transformer (`http://jalammar.github.io/illustrated-transformer/`)
- I'll also draw attention to parts that are important for understanding how the transformer has been integrated into language modeling

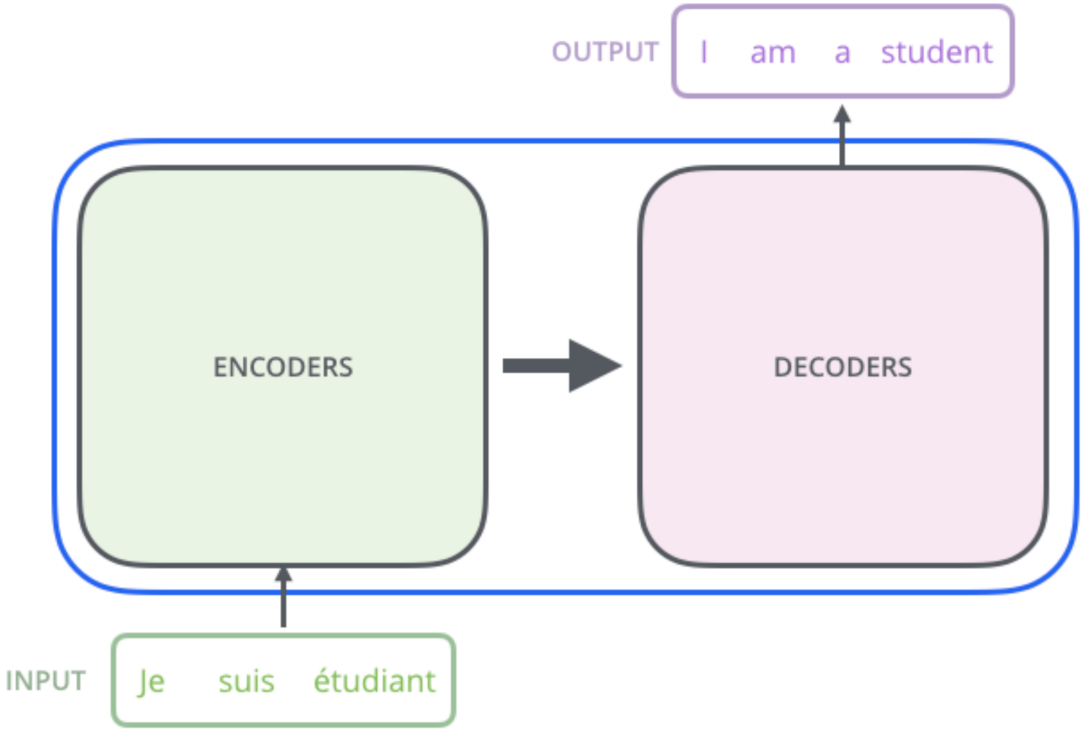
Attention Tricks

The paper made several powerful modifications to attention, which we will talk about in turn:

1. Self-attention
2. Multi-headed attention
3. Positional encodings
4. Normalized dot product attention

High Level Overview

The transformer was developed for machine translation, and hence it has an encoder component and a decoder component



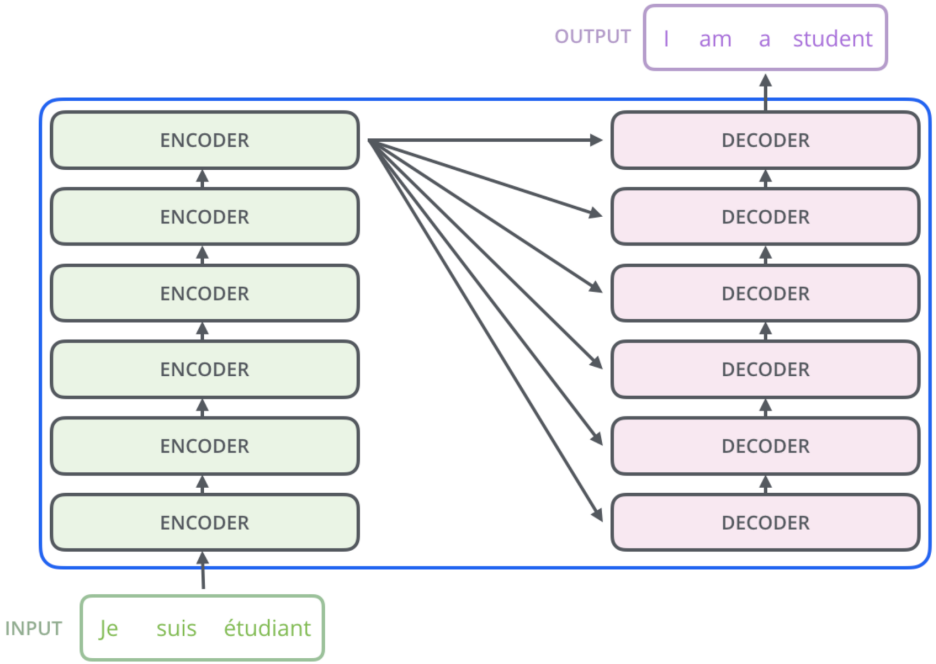
http://jalammar.github.io/illustrated-transformer/

High Level Overview

- Having an encoder and a decoder is not essential to the transformer architecture
- We will see subsequently that many of the most powerful applications of the transformer architecture use only the encoder or decoder blocks
- This goes back to the idea of neural networks as legos that can be stacked in various ways (subject to solving the vanishing gradient problem), one of the most powerful ideas in this course

High Level Overview

The encoder and decoder each stack six transformer blocks



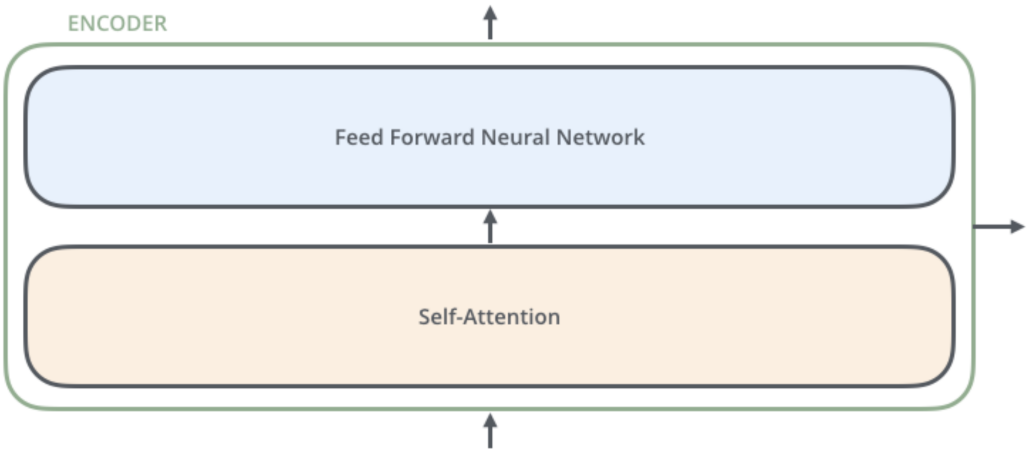
http://jalammar.github.io/illustrated-transformer/

There is nothing inherent about stacking 6 transformer blocks, it's an architectural choice. We'll see a language model (GPT-3) that stacks 96 transformer blocks (and has 175 billion parameters/cost $4.6m to train)

The Encoder Blocks

The encoder/decoder layers differ in important ways, that we'll discuss. This is going to be important moving forward, because when we see transformer-based language models, they will just use either encoder or decoder blocks.

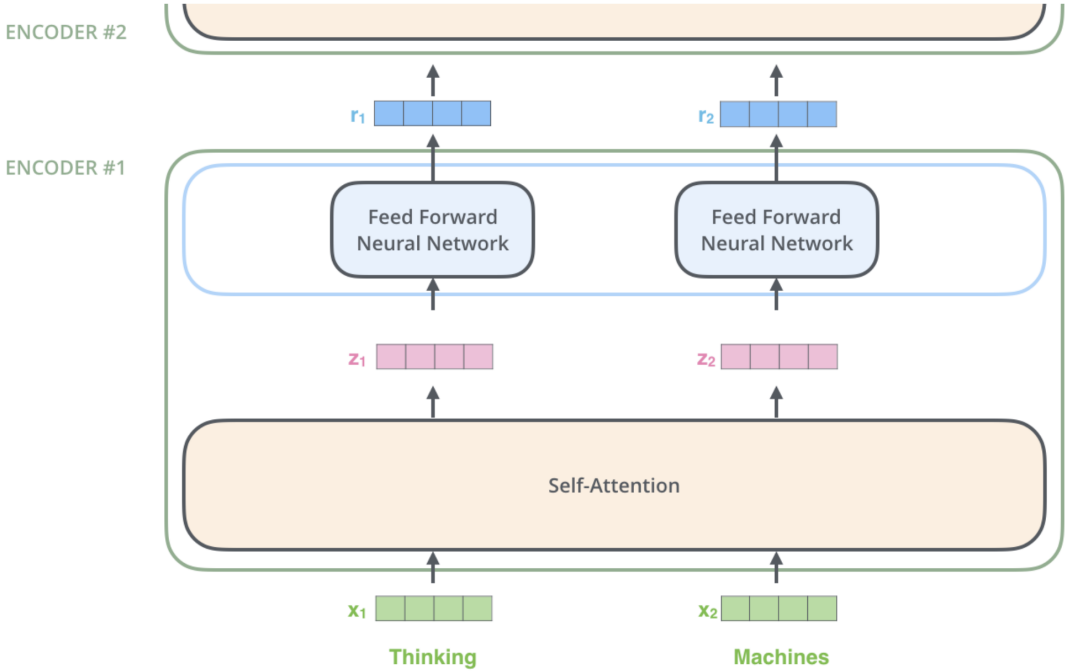
Each of the six encoder layers have the same structure; they do not share weights



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

Encoder Blocks

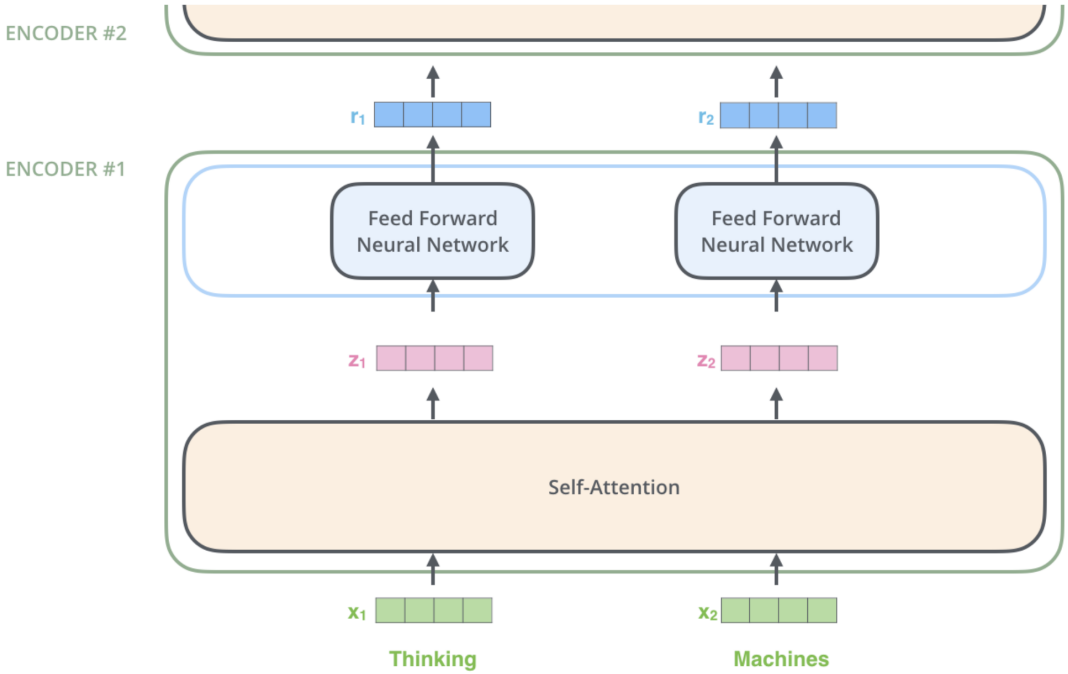
Actually, we should draw the encoder blocks like this



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

A sequence of words is fed into the transformer at the same time. There are dependencies in the self-attention layer (as we'll see in a minute, this is the whole point of self-attention)

Encoder Blocks



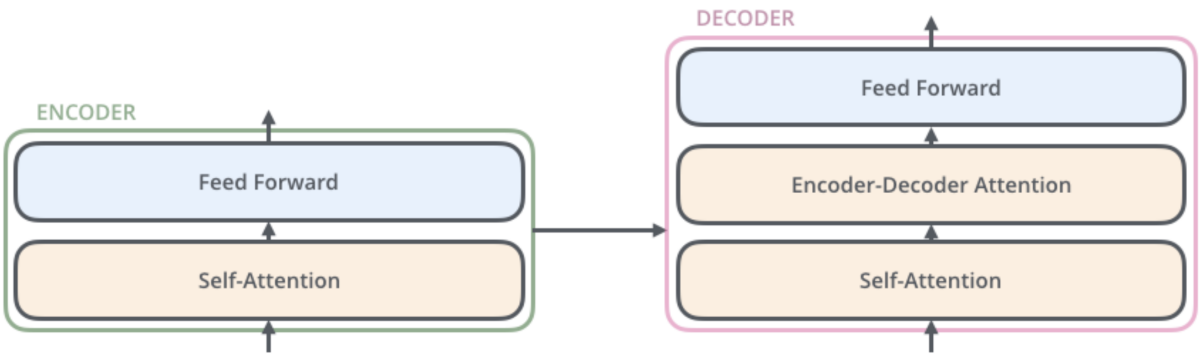
http://jalammar.github.io/illustrated-transformer/

The parallel aspect of this - feeding all words in a sequence into the transformer block together - is central to the success of this model. It makes it possible to compute things much faster - and hence train a much larger model - than architectures like an RNN that are computed sequentially.

The Decoder Blocks

The decoder blocks also have a self-attention layer and a feed forward layer, but include an additional attention layer that attends to the final encoder layer (like the attention in seq2seq that we saw earlier)

Self-attention works differently in the encoder versus decoder. This is going to be relevant subsequently, when we see that some language models stack decoder blocks and others encoder blocks



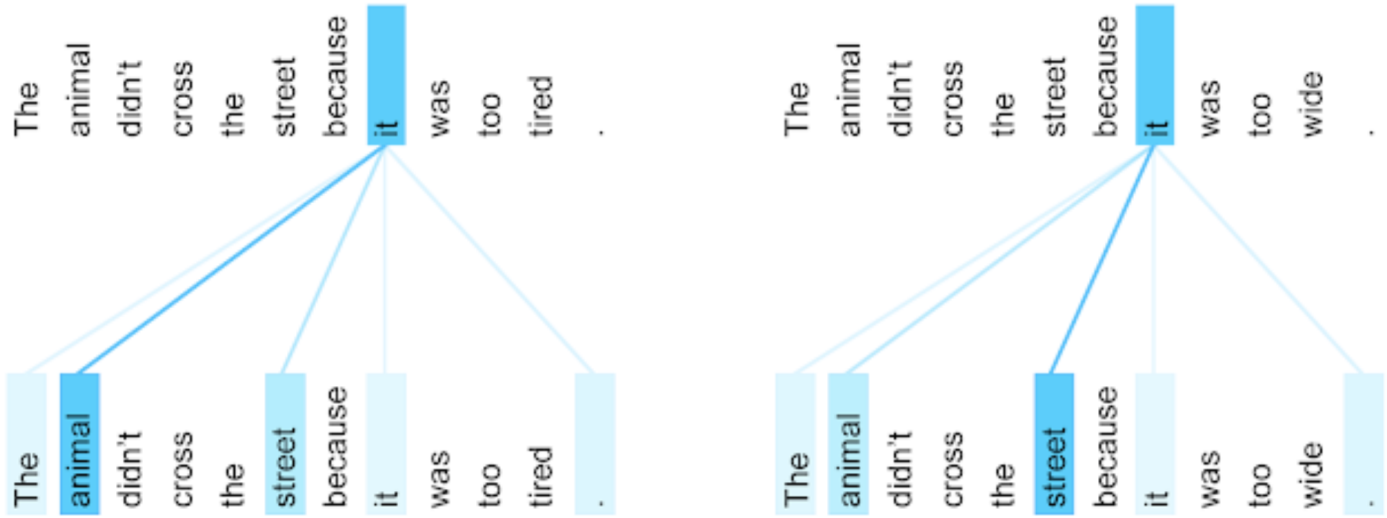
http://jalammar.github.io/illustrated-transformer/

Self-Attention

- Previously, attention allowed each step of the decoder to attend to different portions of the encoder
- A key insight of the transformer is that we also want a sentence or sequence of words to be able to attend to different portions of itself while being embedded
- To use the example from the paper: “The animal didn’t cross the road because it was too tired” versus “The animal didn’t cross the road because it was too wide.” Does “it” refer to animal or road? Relevant when translating to a language with gendered pronouns.

Self-Attention

The transformer gets this right, whereas previous models did not. You can see this in the self-attention distributions

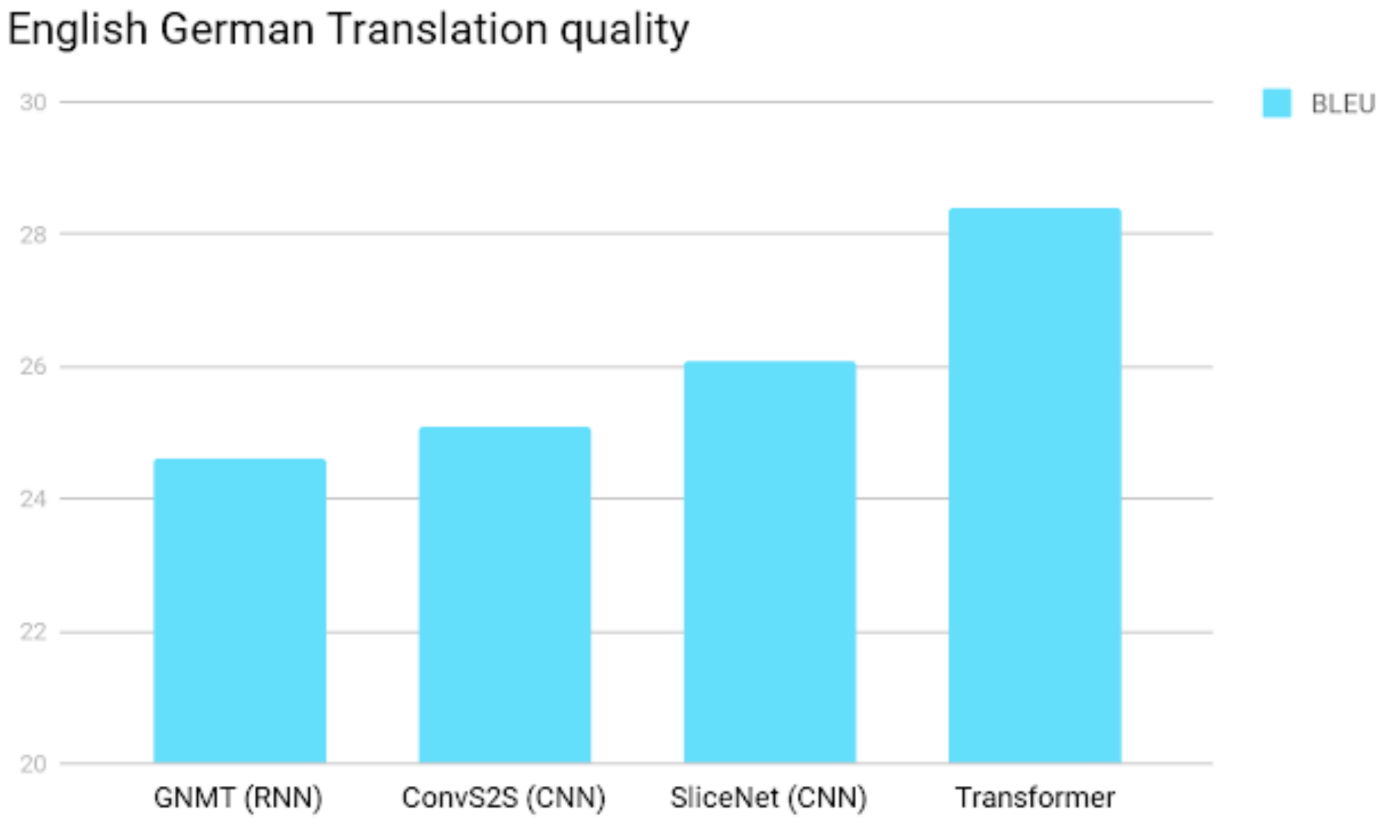


The encoder self-attention distribution for the word "it" from the 5th to the 6th layer of a Transformer trained on English to French translation (one of eight attention heads).

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

Recall in an RNN the hidden state allows it to incorporate past representations into the current representation. Self-attention serves this role in the transformer.

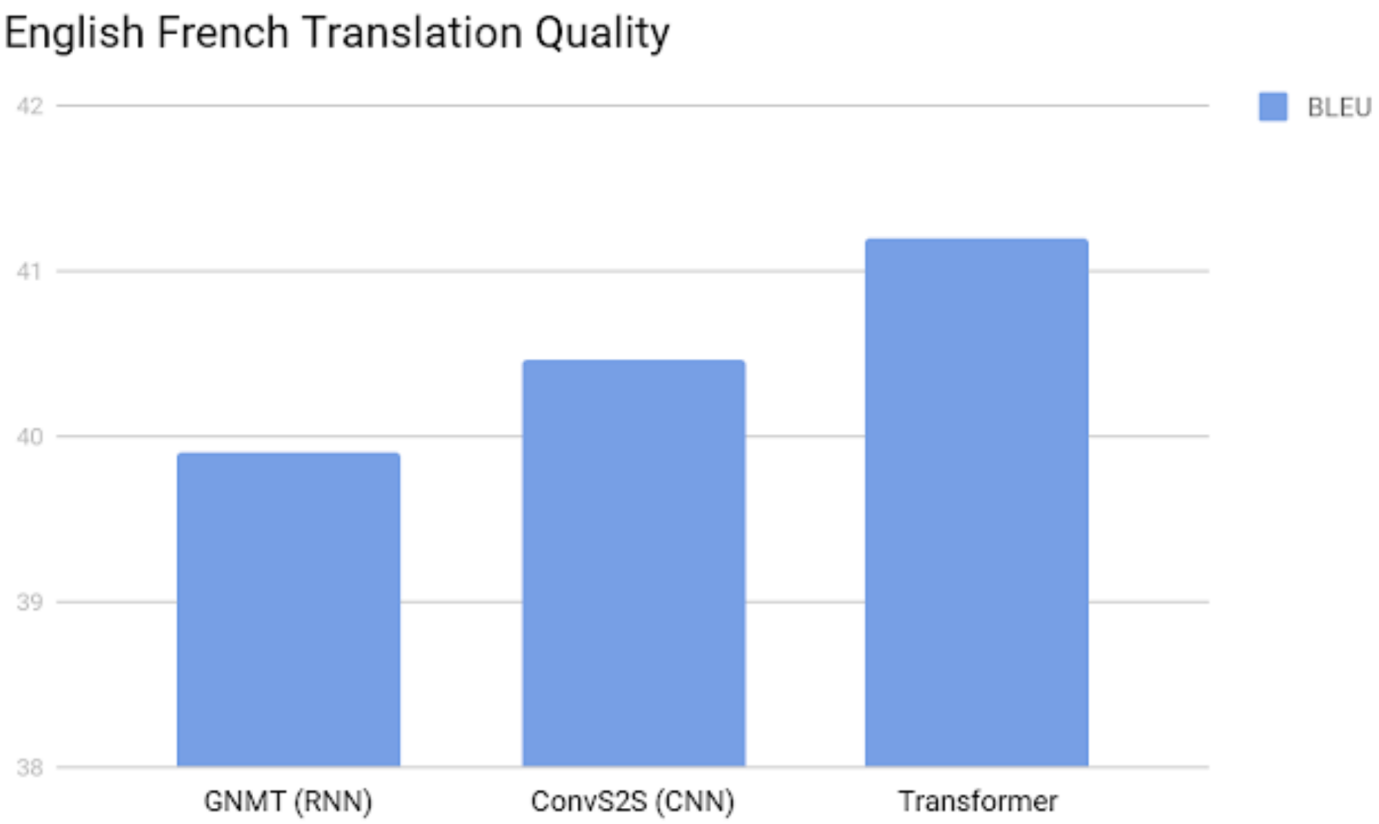
Impressive Results



BLEU scores (higher is better) of single models on the standard WMT newstest2014 English to German translation benchmark.

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

Impressive Results



BLEU scores (higher is better) of single models on the standard WMT newstest2014 English to French translation benchmark.

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

Outline

The Encoder

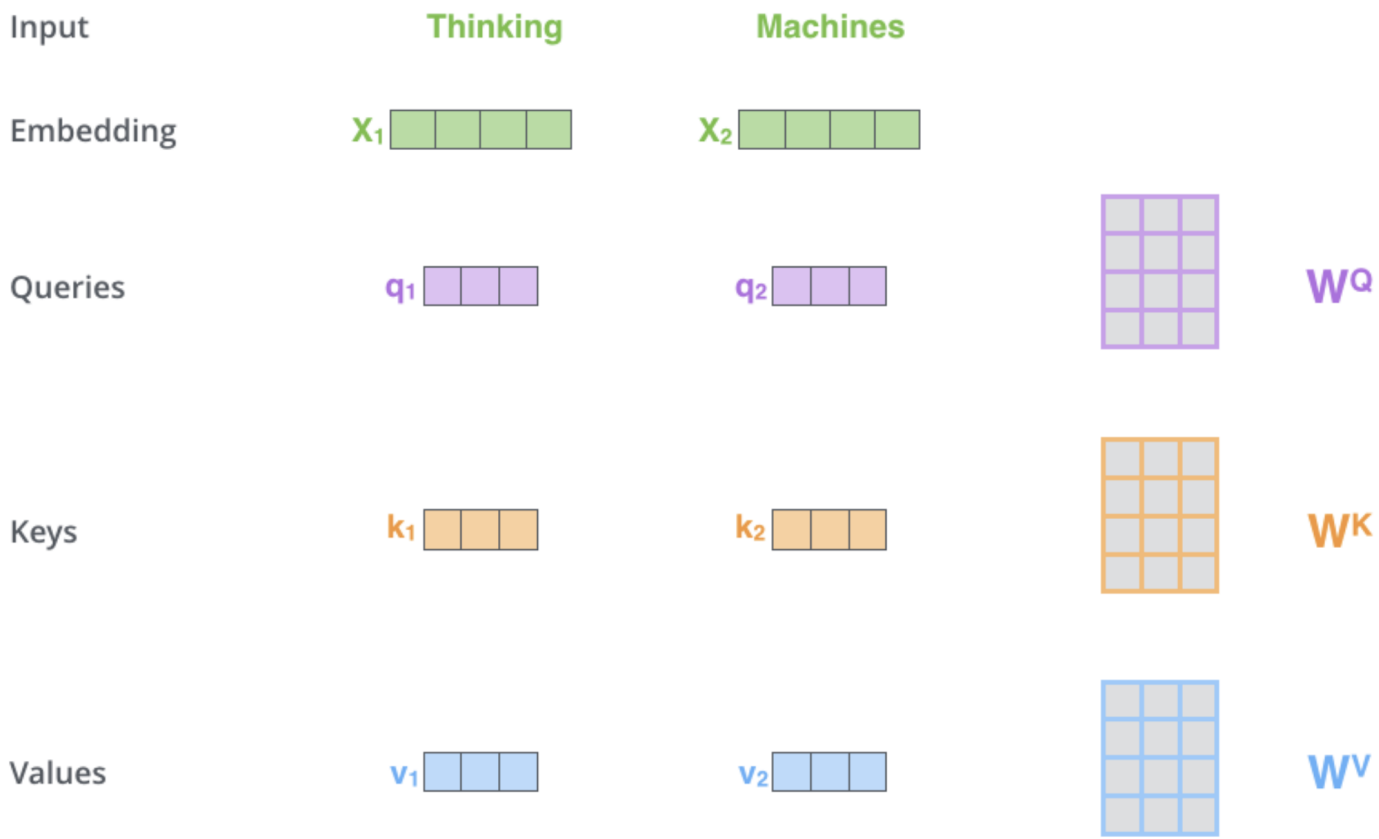
We need to understand the key features of the encoder blocks:

- Encoder self-attention
- Positional encodings
- Add and normalize

Computing Encoder Self-Attention

- As in previous models we've seen, the initial inputs to the transformer are word vectors from a lookup table (that are themselves parameters to be estimated). They are fed into the model at the same time (versus sequentially, as in an RNN)
- The first step to calculating self-attention is to compute three vectors: a query vector, a key vector, and a value vector
- These are calculated by multiplying the embeddings fed into the model by three matrices
- Note that these vectors are smaller (i.e. dimensionality 64, versus 512 for the input embeddings). As we'll see, the transformer has multiple attention heads and this is done for computational purposes

Computing Encoder Self-Attention



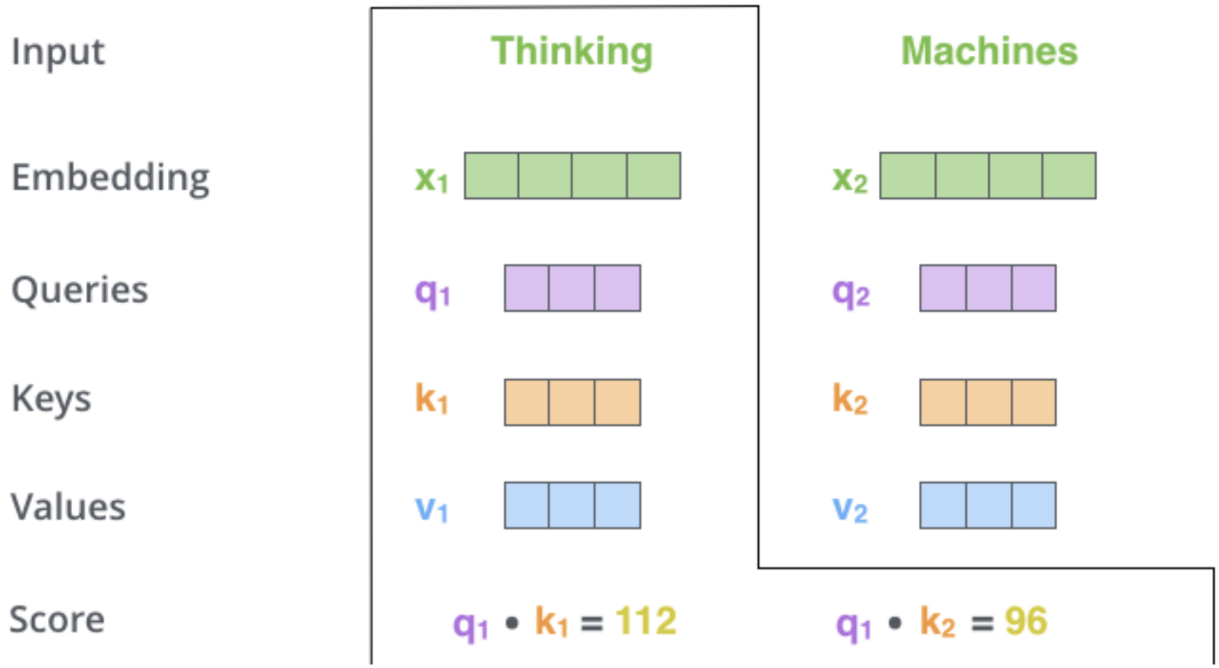
Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

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

The weight matrices for the query, key, and value vectors are estimated parameters

Computing Self-Attention

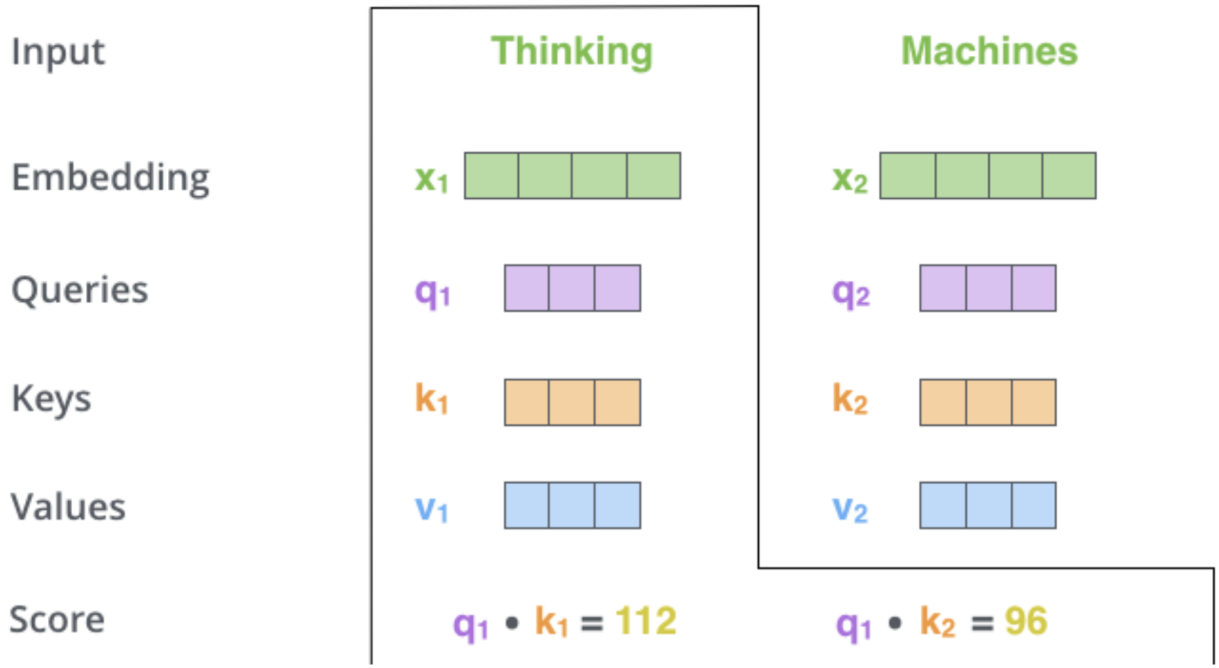
Just as with regular attention, we use the query and the key to compute the attention score



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

Computing Self-Attention

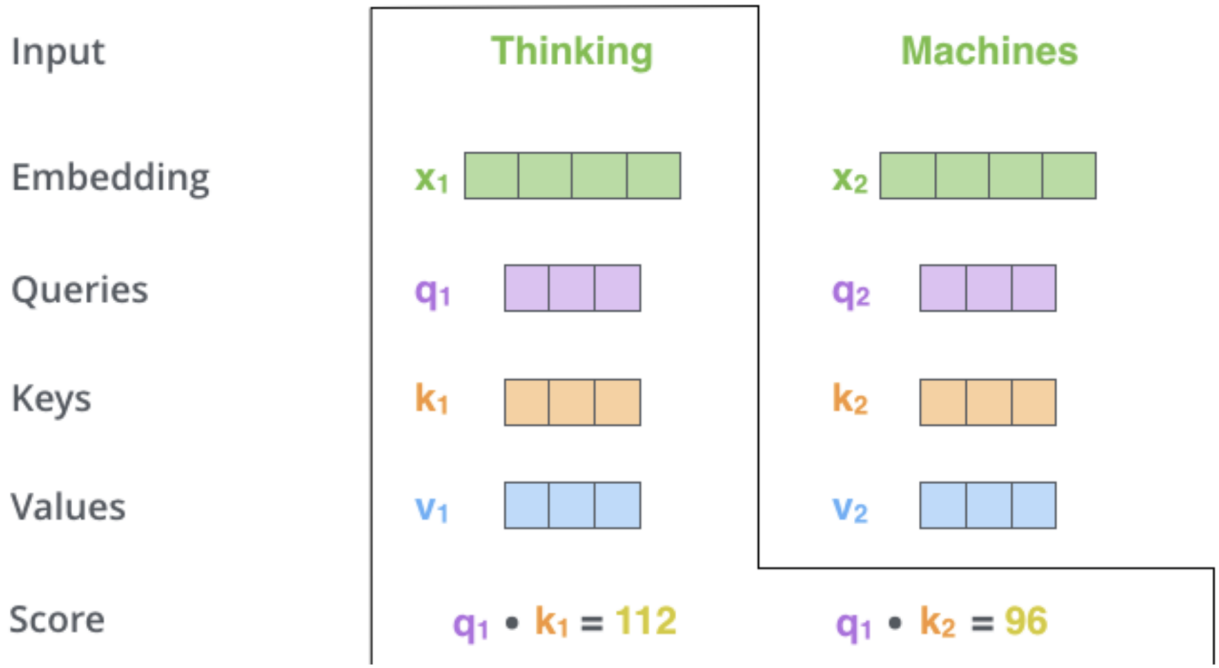
For a given word, we use the query to compute attention to all words in the input



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

Computing Self-Attention

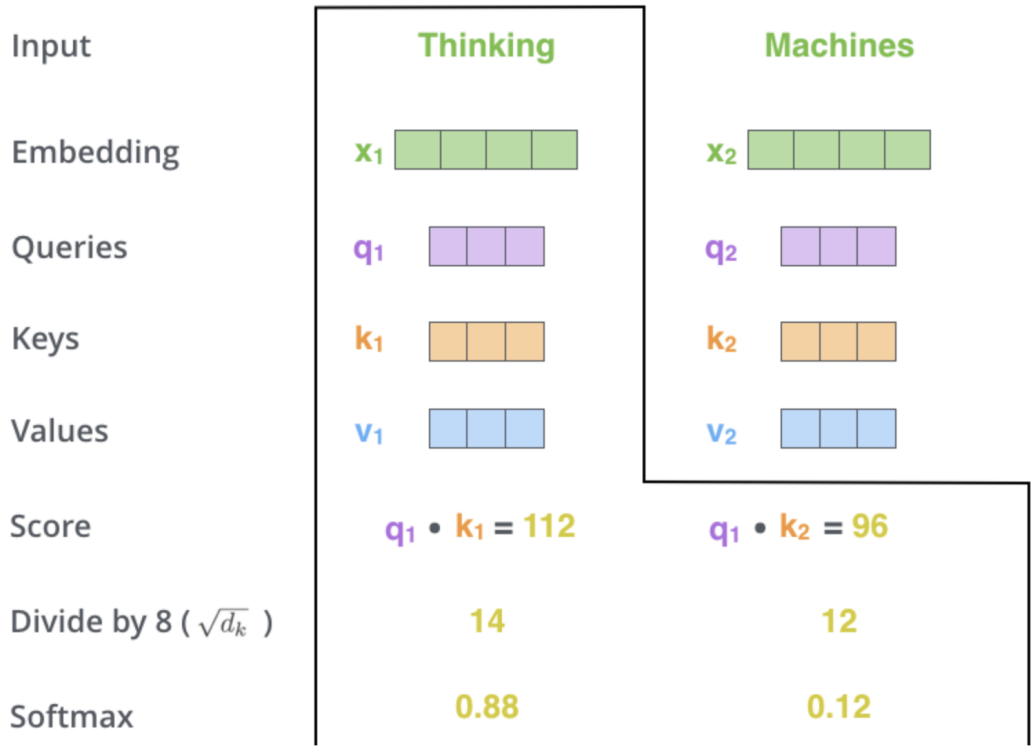
We compute the attention score by taking the dot product between the query and the key for each word in the sequence



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

Computing Self-Attention

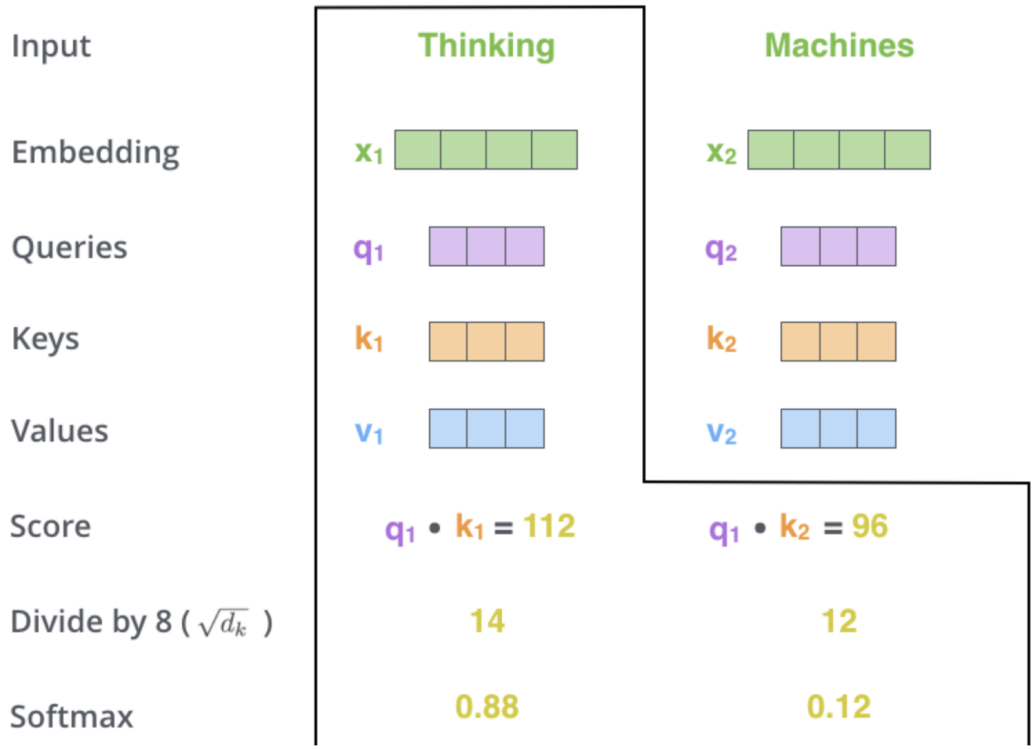
Next we divide the scores by the square root of the key vector dimensionality, which in the original transformers is 8. This produces more stable gradients



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

Computing Self-Attention

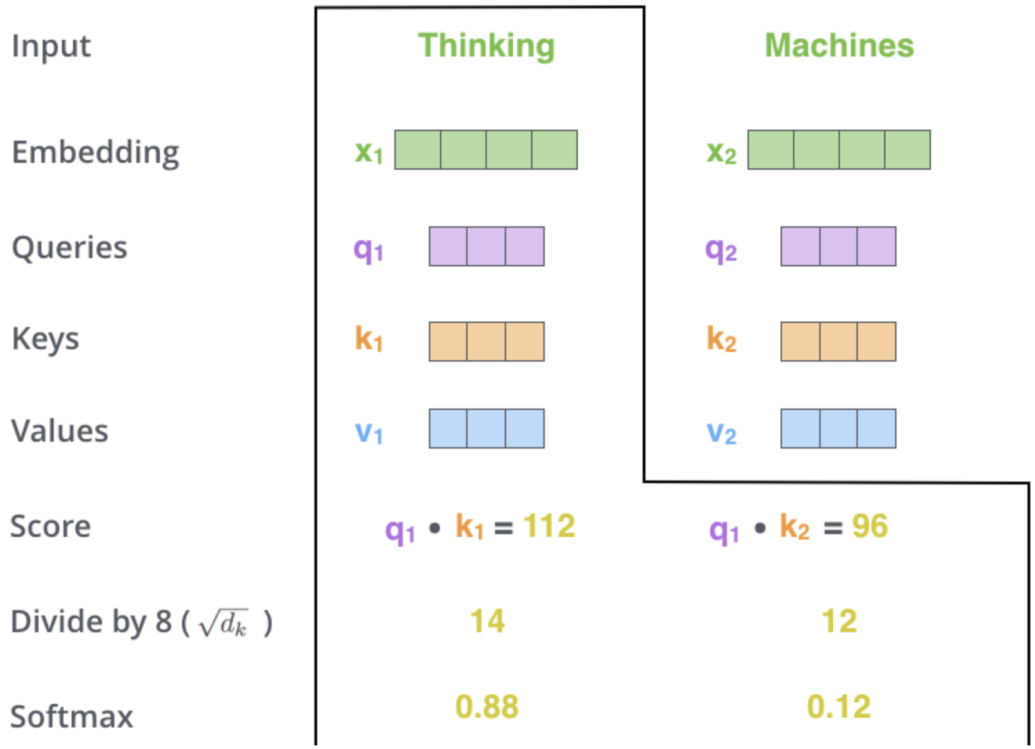
Then we take the softmax



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

Computing Self-Attention

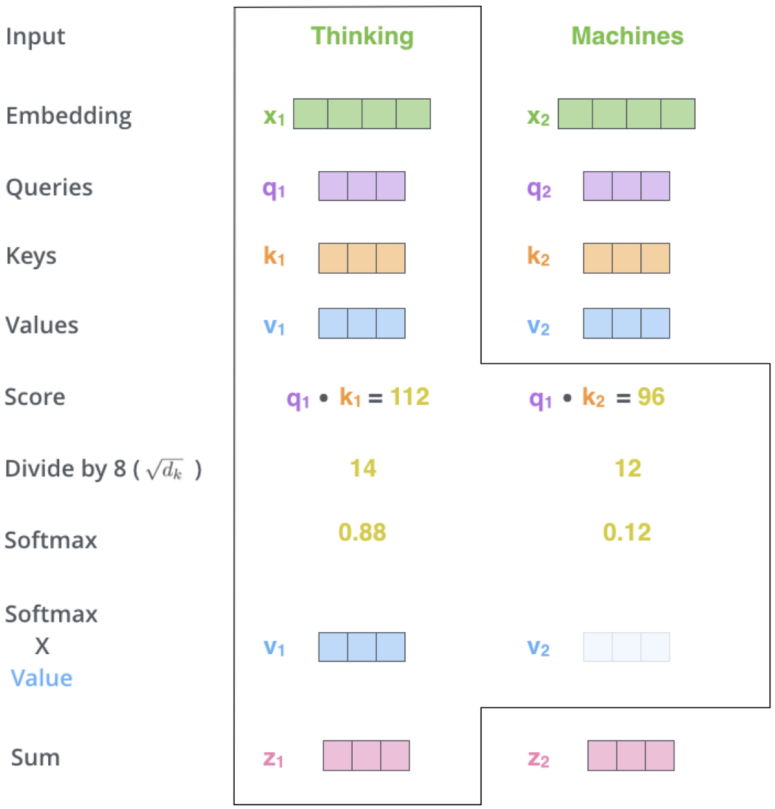
As we might expect, the word at the position under consideration has the highest score, but other words may also be attended to



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

Computing Self-Attention

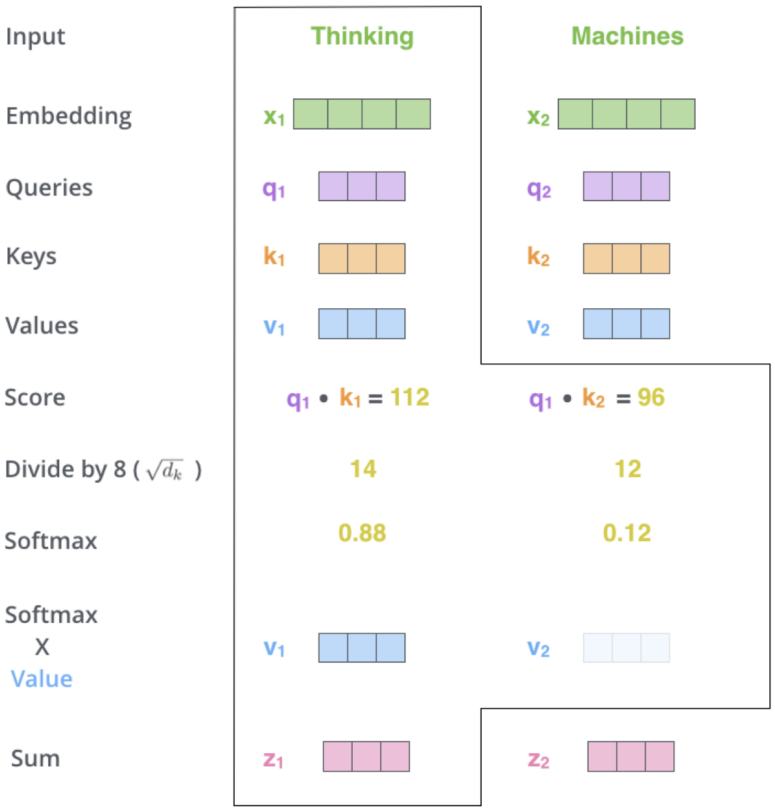
Next, we multiply the softmax score times the value vector for each position



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

Computing Self-Attention

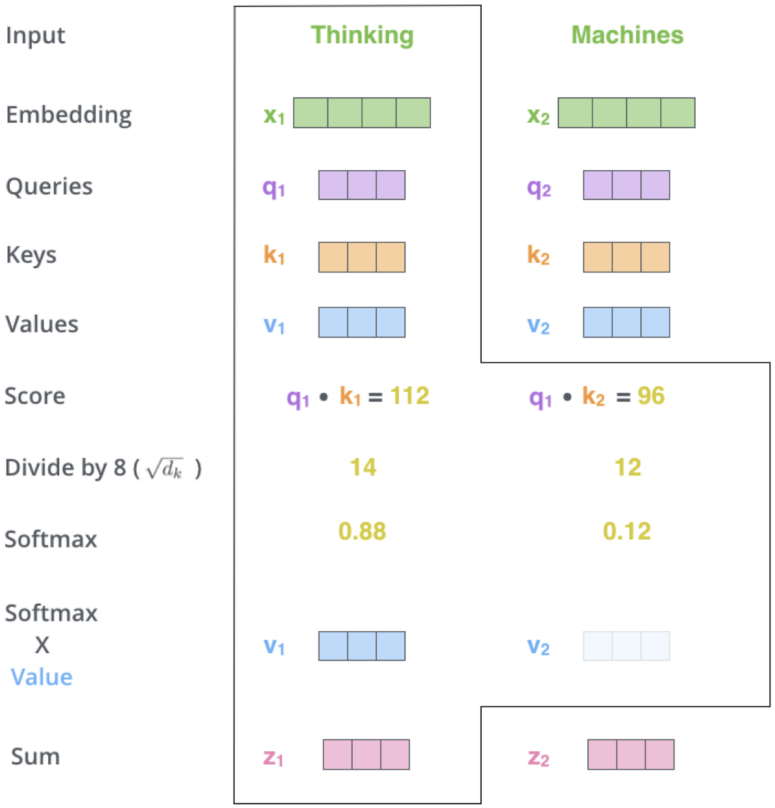
This will preserve the values of words we want to focus on and send the values of other words to zero by multiplying by a small number



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

Computing Self-Attention

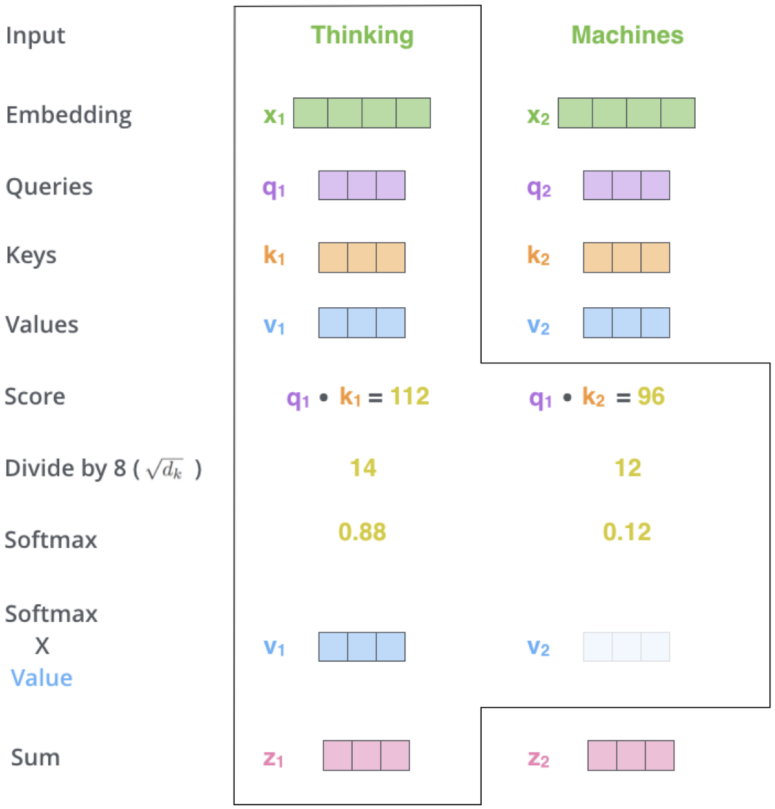
Finally, we add up the weighted value vectors



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

Computing Self-Attention

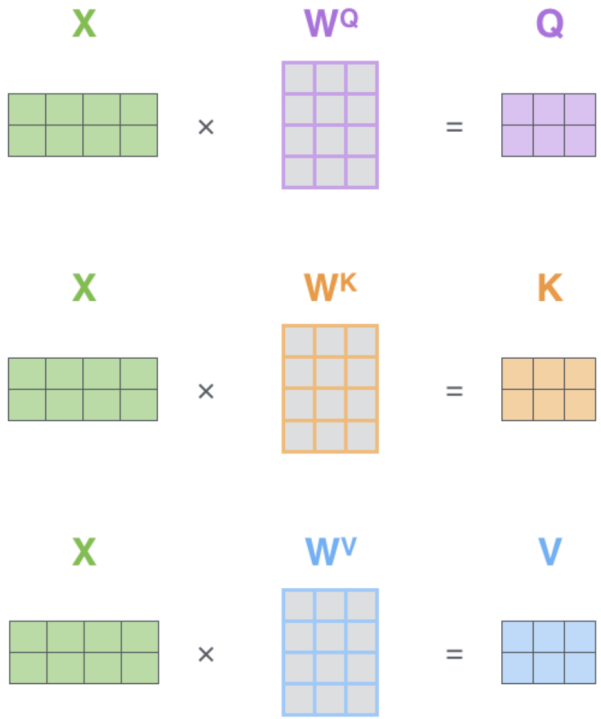
This produces the output of self-attention for that position, which we will call z



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

Writing this in matrix format

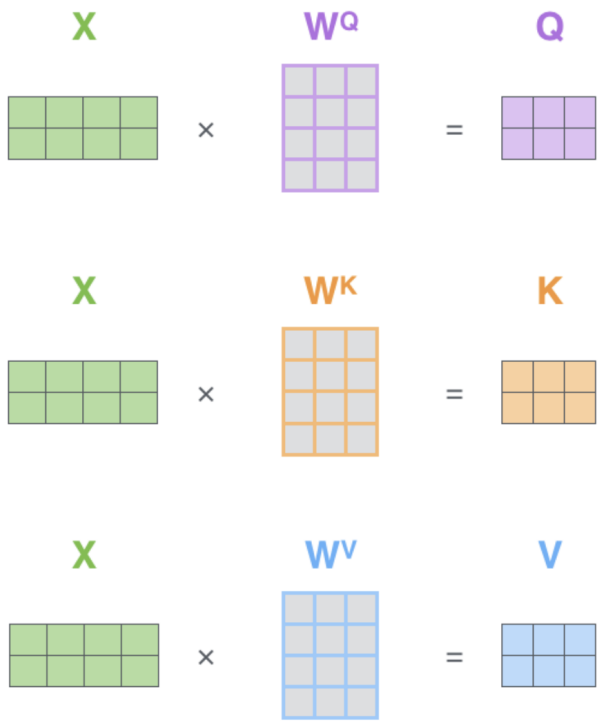
Take a simplified example where we are only feeding in two words



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

Writing this in matrix format

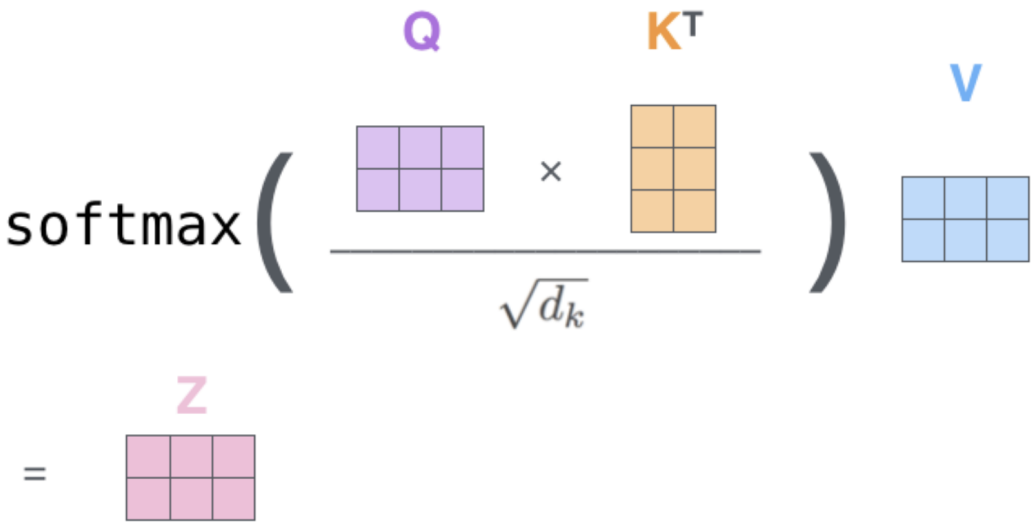
Compute the query, key, and value vectors for each input by matrix multiplication



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

Writing this in matrix format

Calculate the attention scores

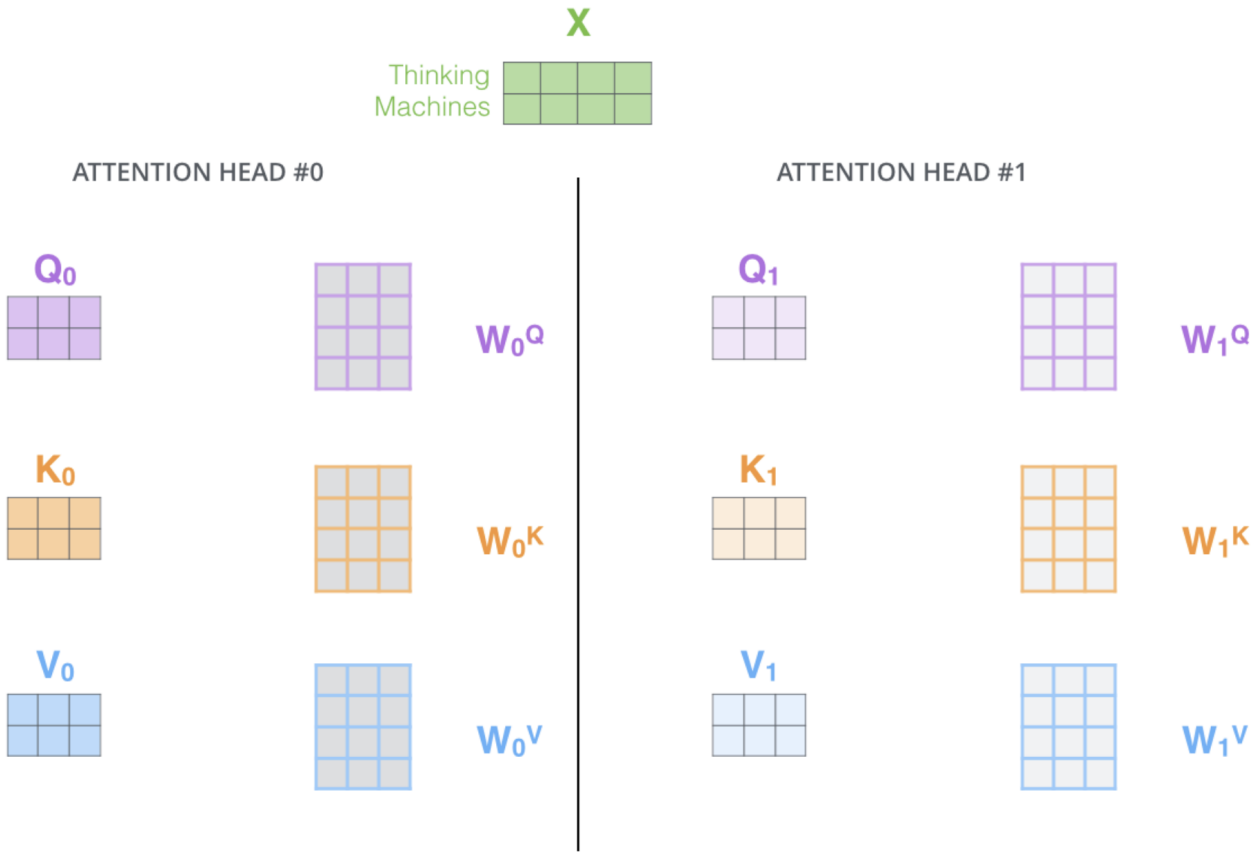


http://jalammar.github.io/illustrated-transformer/

Multi-headed attention

- The paper uses eight attention heads
- This allows for multiple representation subspaces, which leads to better results, i.e. one attention head can mostly pay attention to itself, another can pay attention to the noun that the pronoun refers to, etc.

Multiple Attention Heads

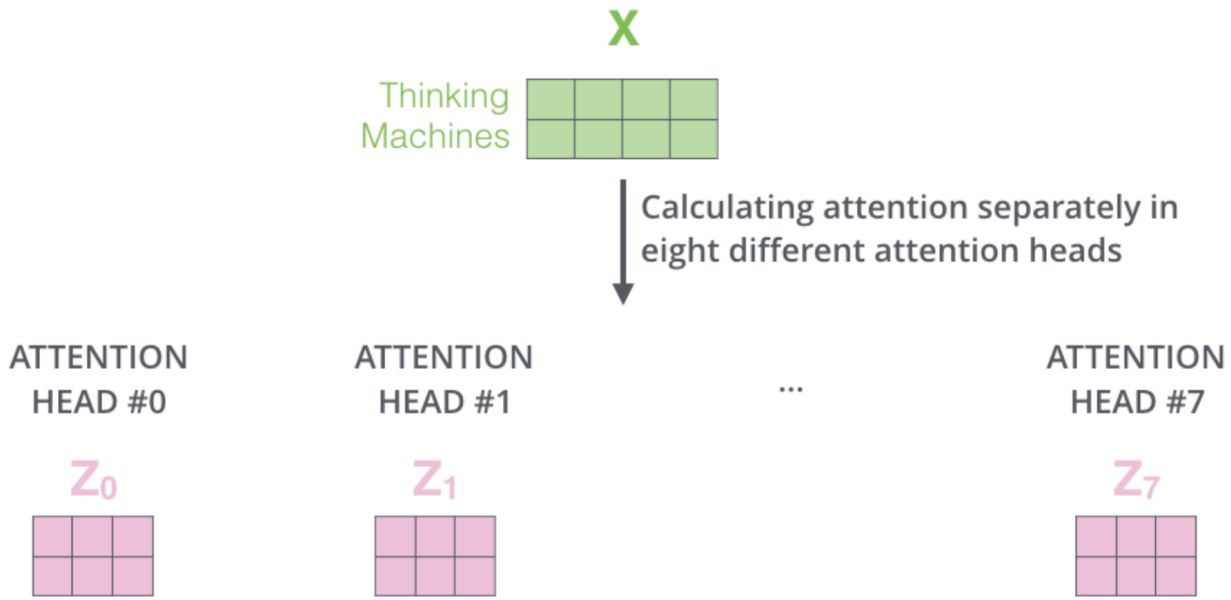


With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

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

Multiple Attention Heads

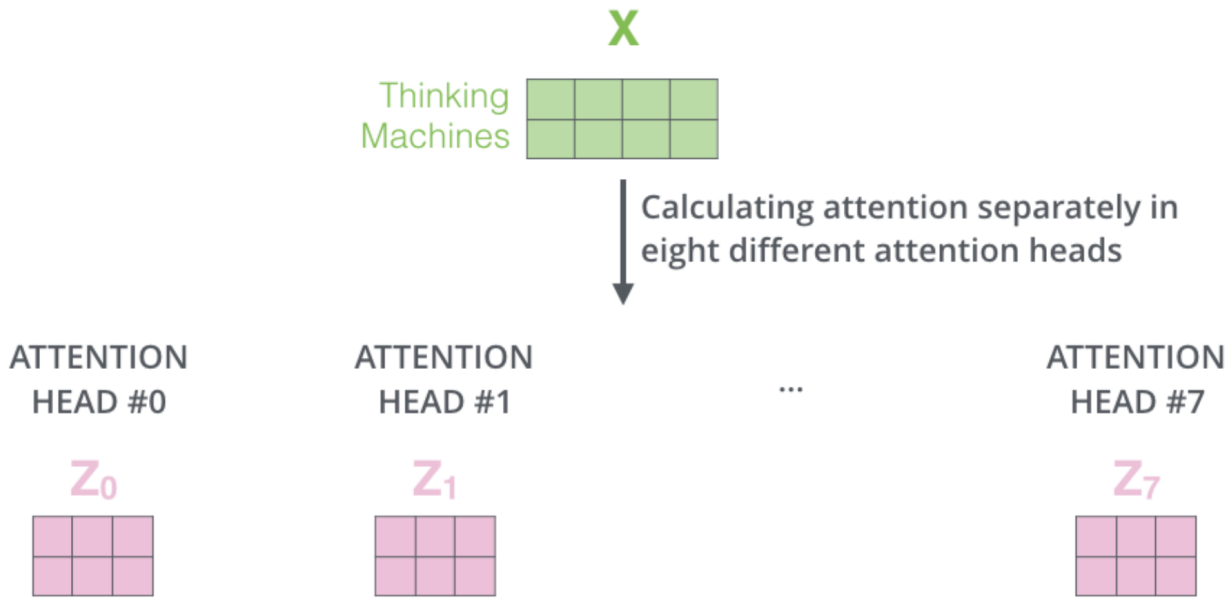
This yields 8 different output vectors



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

Multiple Attention Heads

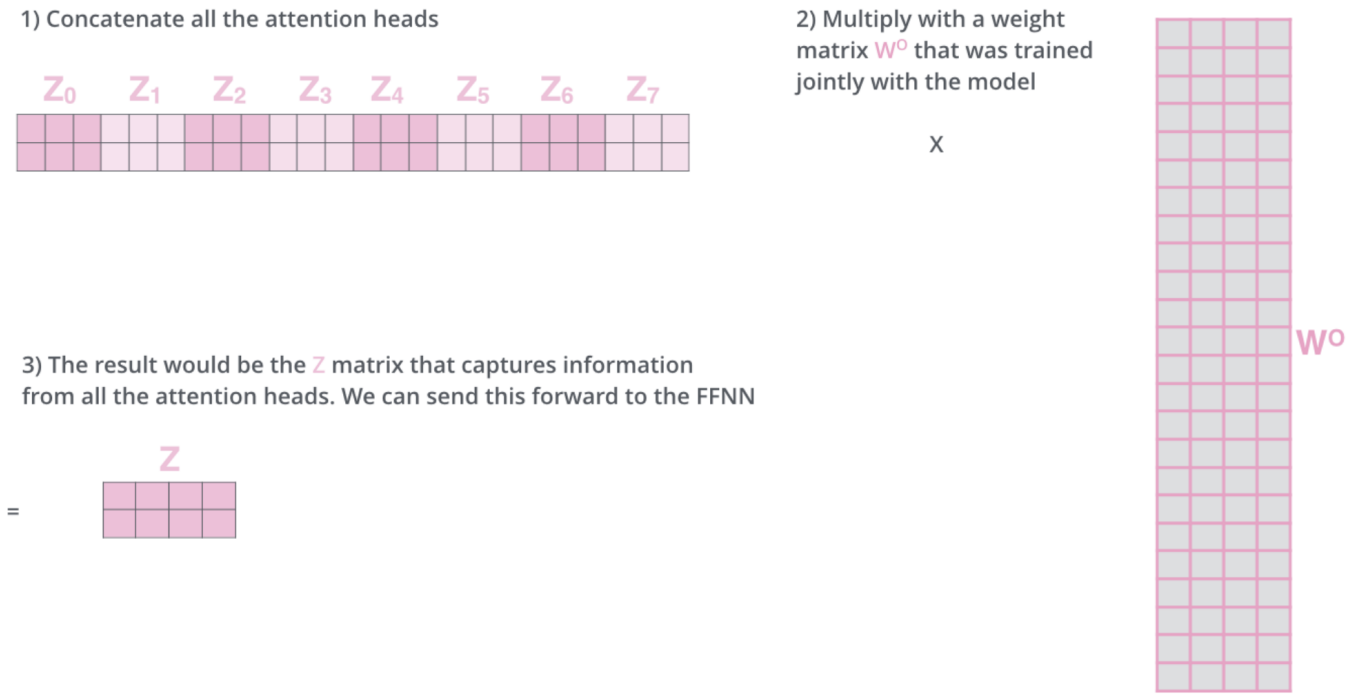
Recall that these representation are different dimensionality than the inputs (64 vs. 512)



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

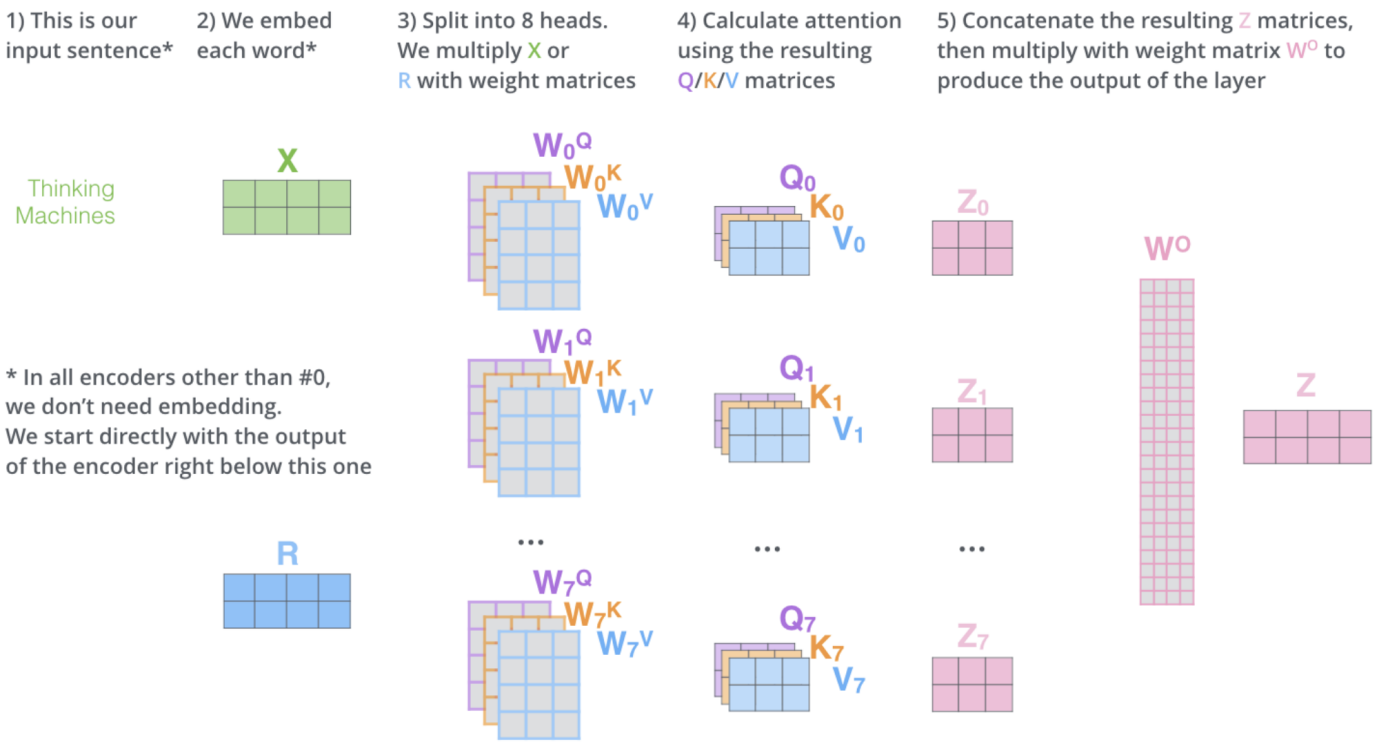
Multiple Attention Heads

We condense this into a single vector for each input, of size 512, by multiplying by yet another matrix



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

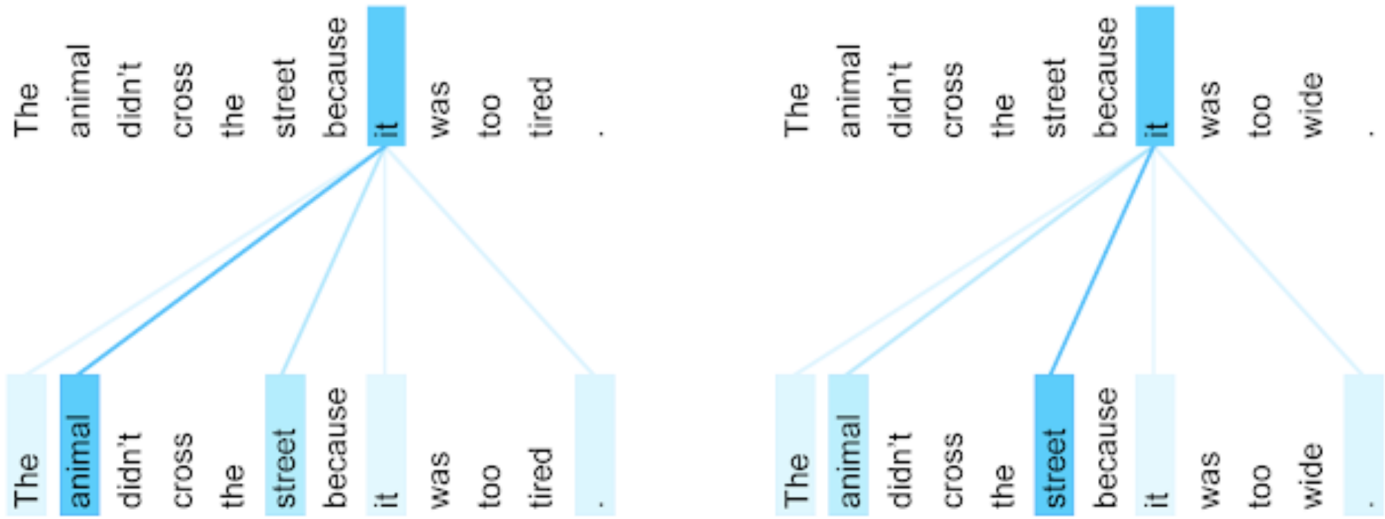
Putting It All Together



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

Self-Attention

This is how we get these self-attention distributions. There are also some other important aspects of the encoder architecture



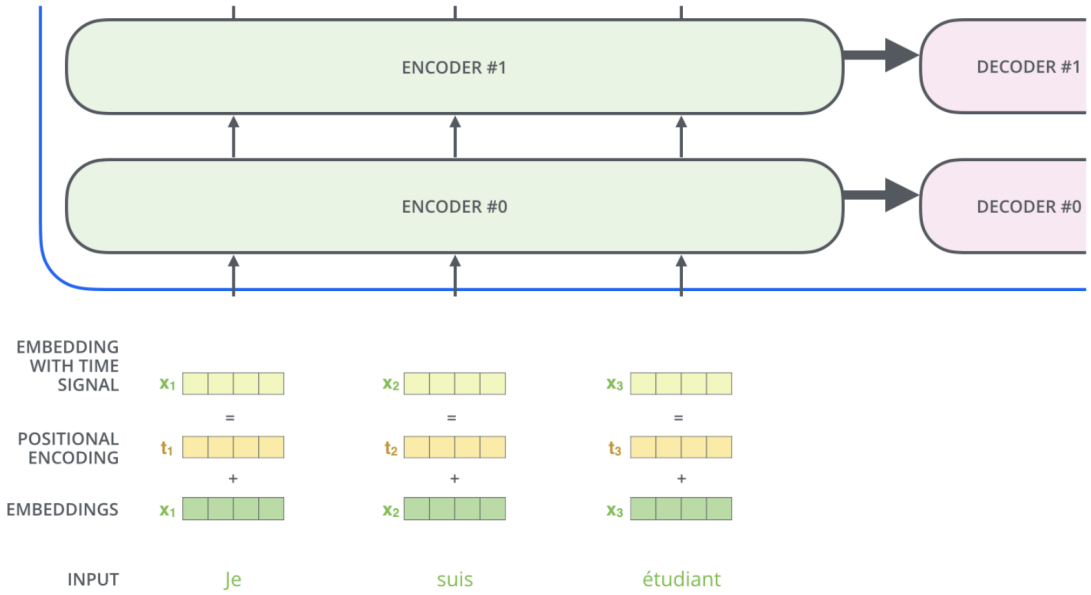
The encoder self-attention distribution for the word “it” from the 5th to the 6th layer of a Transformer trained on English to French translation (one of eight attention heads).

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

Positional Encodings

- In an RNN where we feed in input words one at a time, the model bakes in a way to keep track of the word order. This is important - for example, a word is more likely to need to attend to a word that is closer - and missing from our description of the transformer thus far because we are feeding all tokens in at once
- The model addresses this by adding a vector to each input that captures the position
- These follow a specific pattern, which the model can learn in order to be able to incorporate positional information

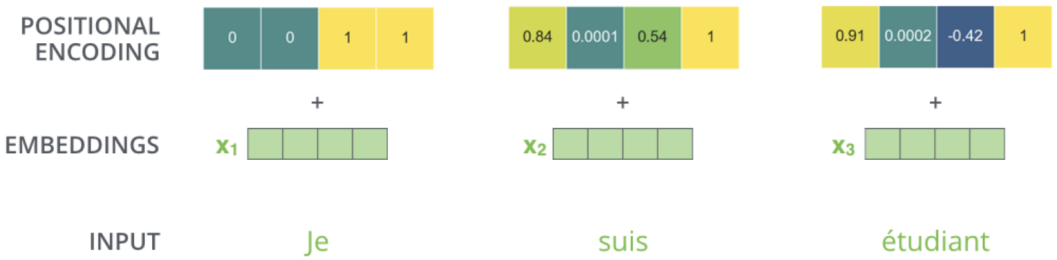
Positional Encodings



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

Positional Encodings

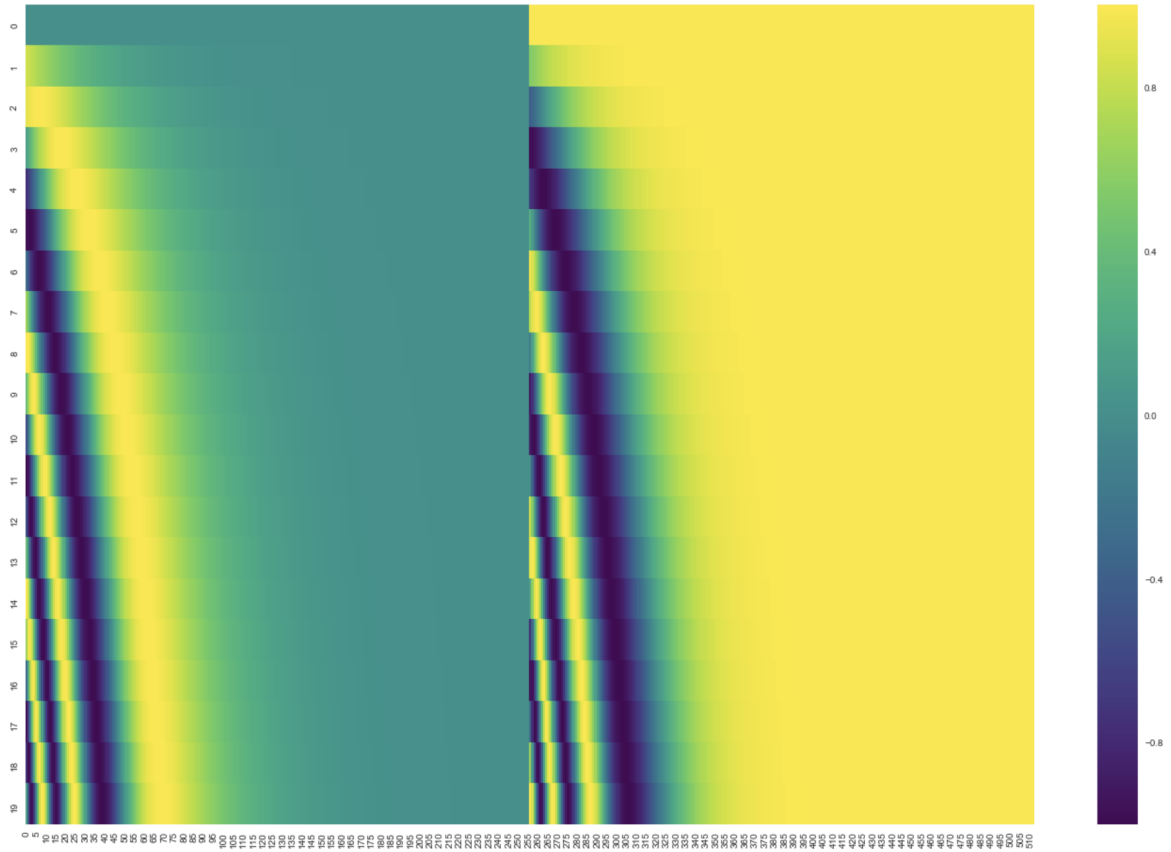
Here's an example for a four word input:



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

Positional Encodings

Plot from the transformer to transformer implementation package

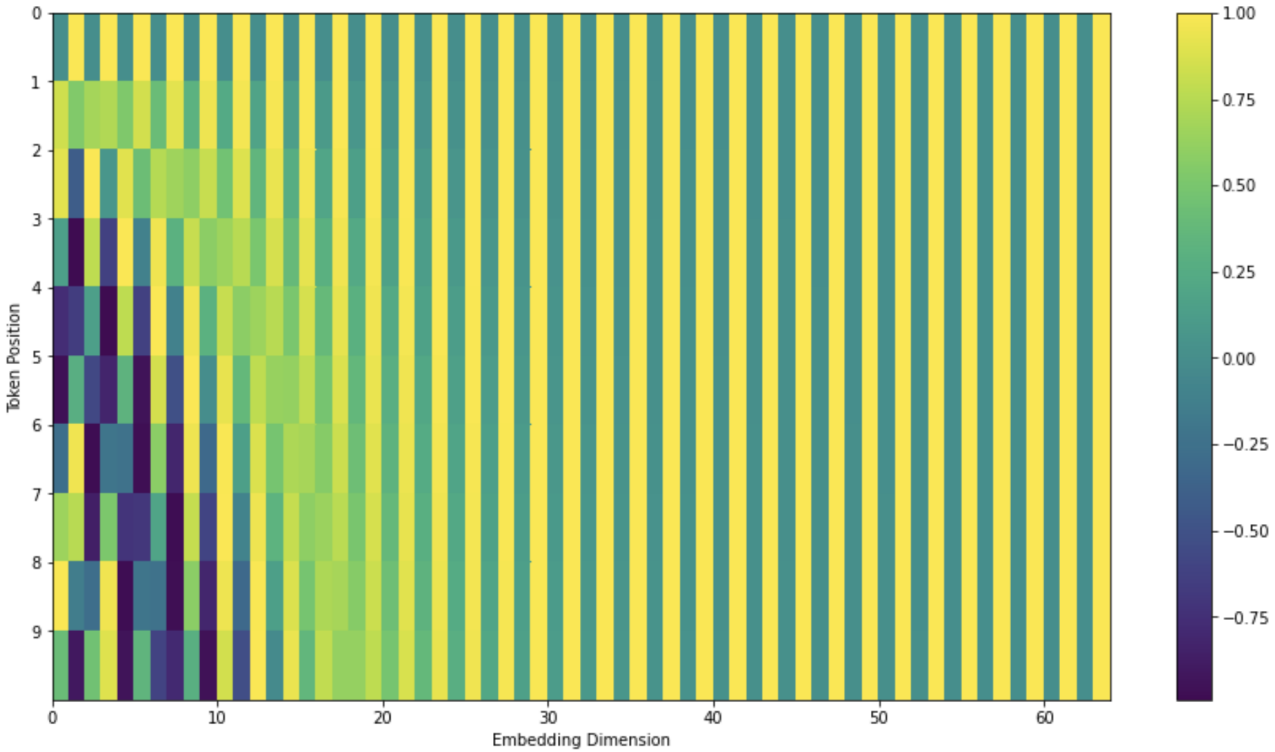


A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in half down the center. That's because the values of the left half are generated by one function (which uses sine), and the right half is generated by another function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

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

Positional Encodings

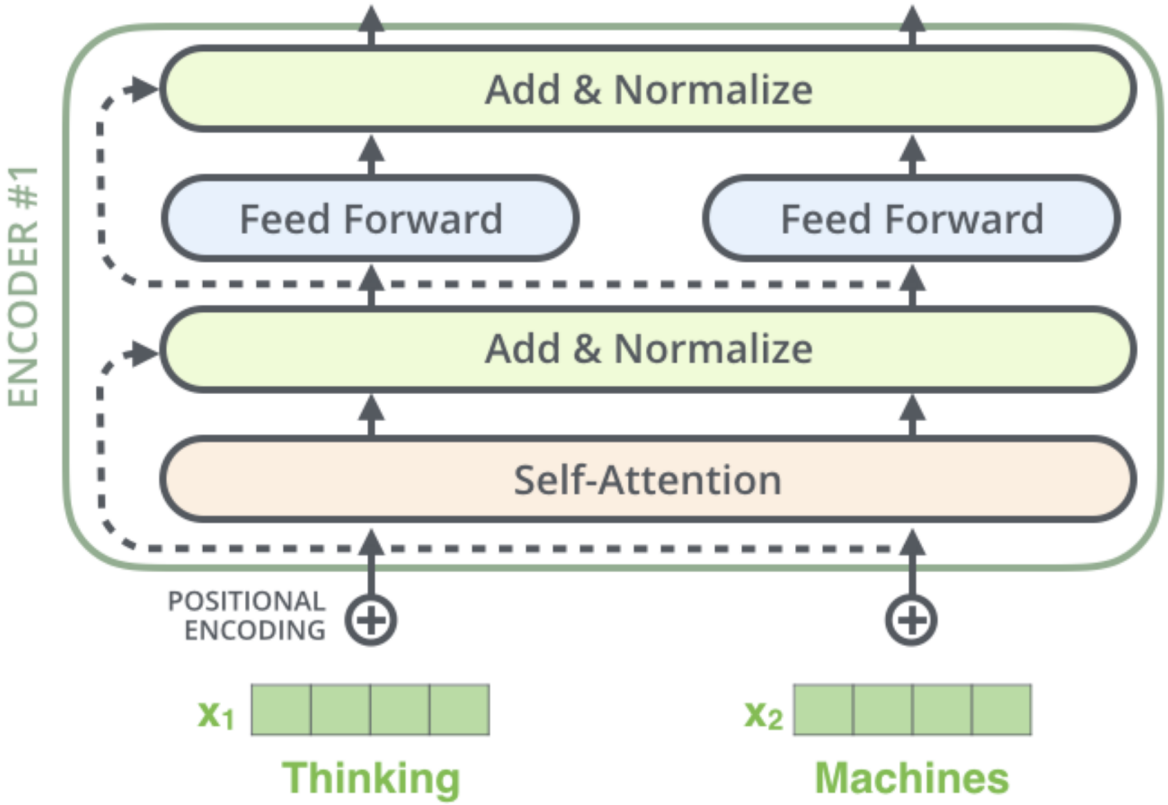
Plot from method used in the original paper, which intersperses the sin and cos signals



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

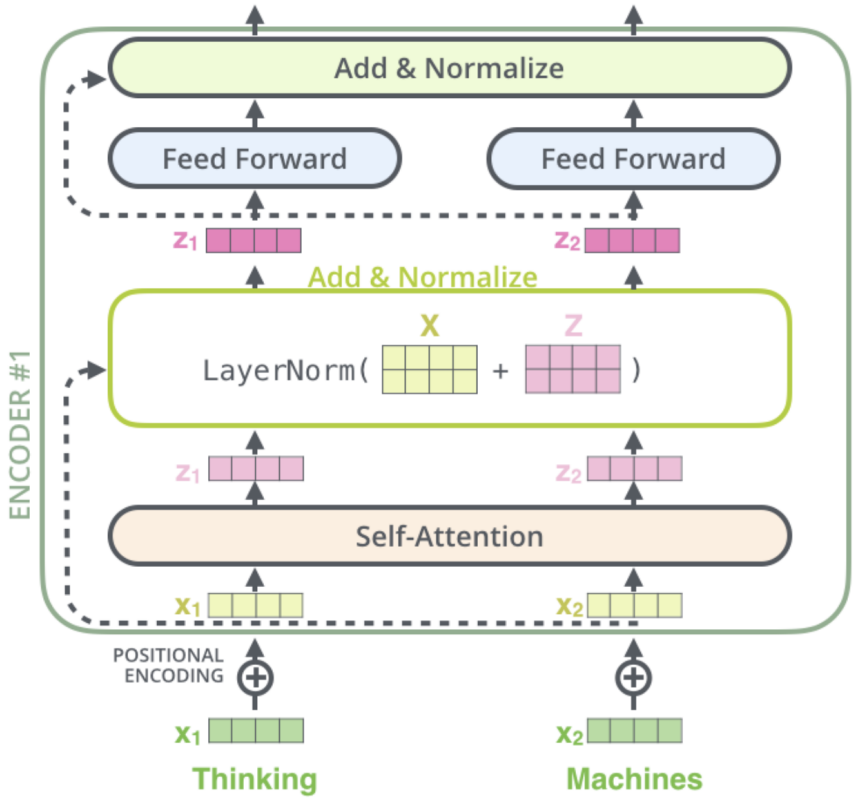
Add and Normalize

Another key component of the architecture is the residual connections, as well as normalization at each layer. We've seen both of these before (i.e. they are integral to ResNet's success)



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

Add and Normalize

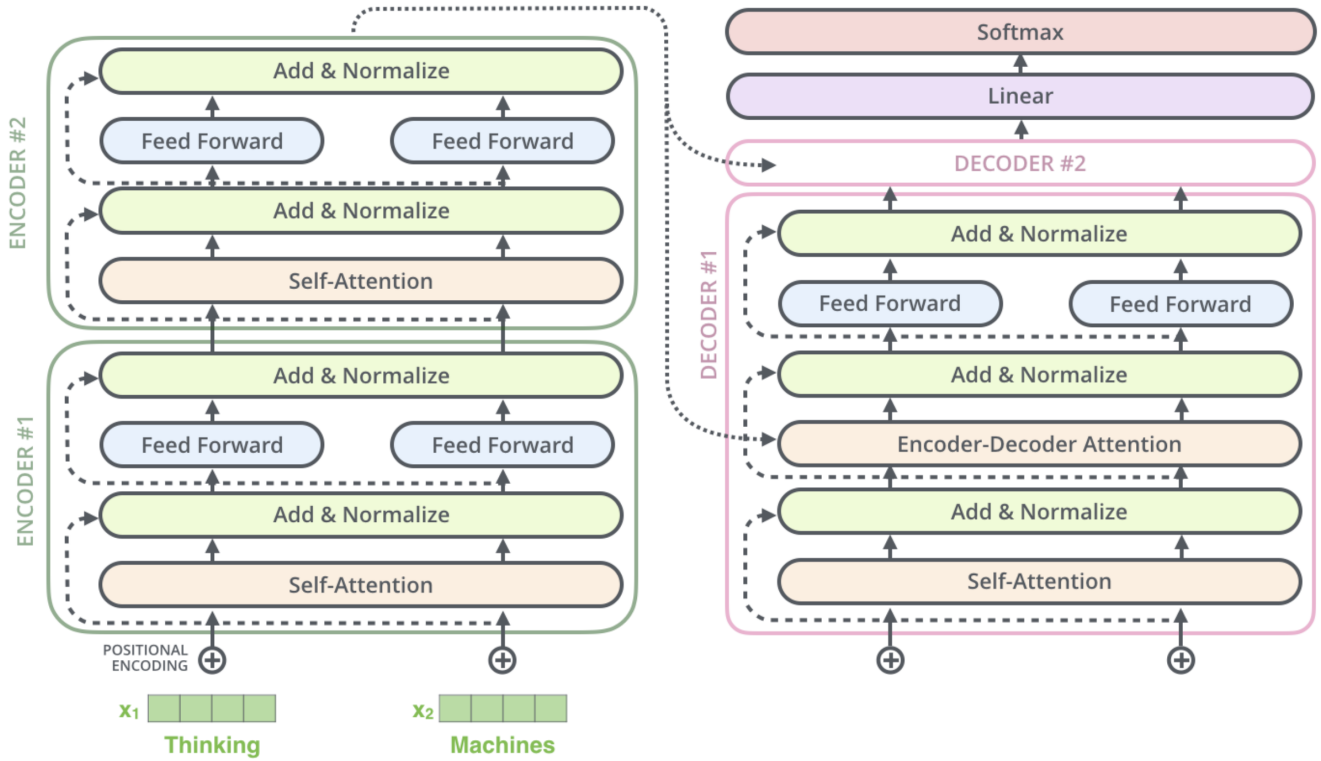


http://jalammar.github.io/illustrated-transformer/

Outline

The Decoder

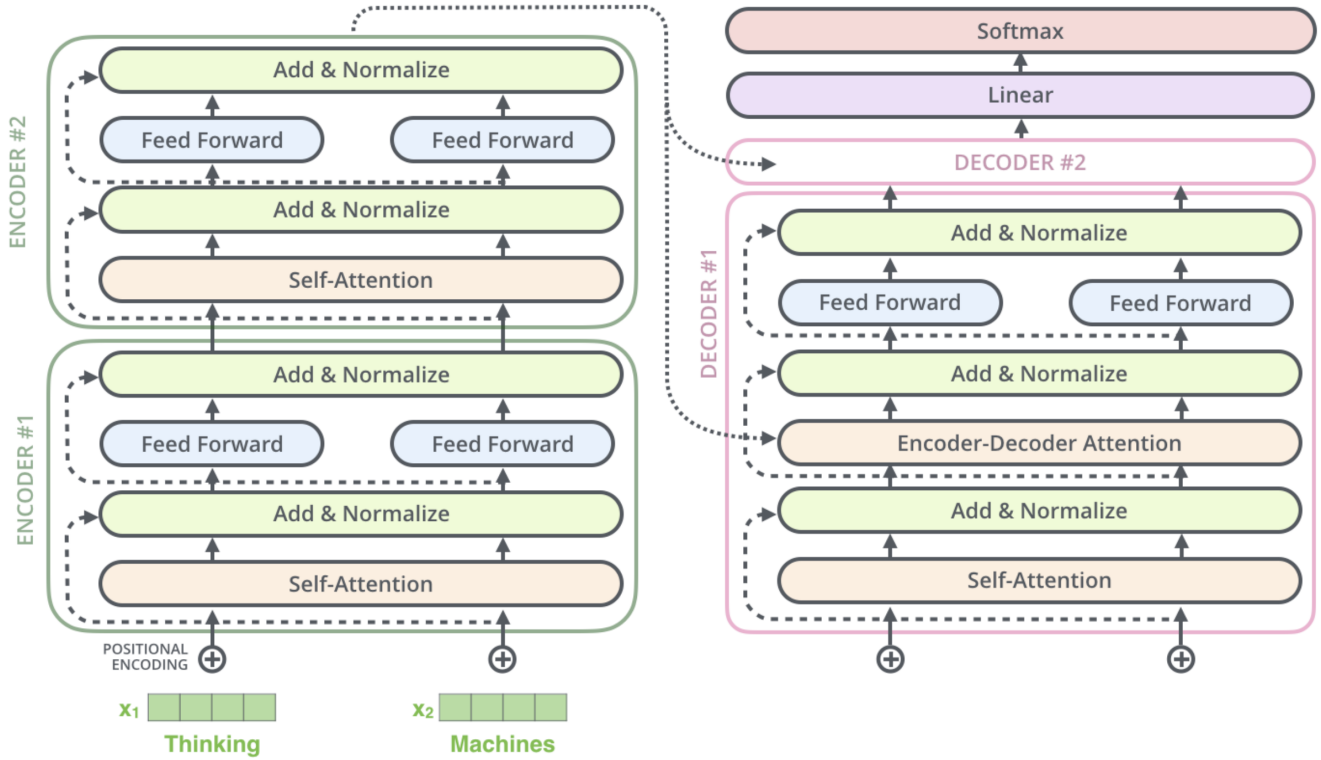
In many ways, the decoder is similar to the encoder



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

The Decoder

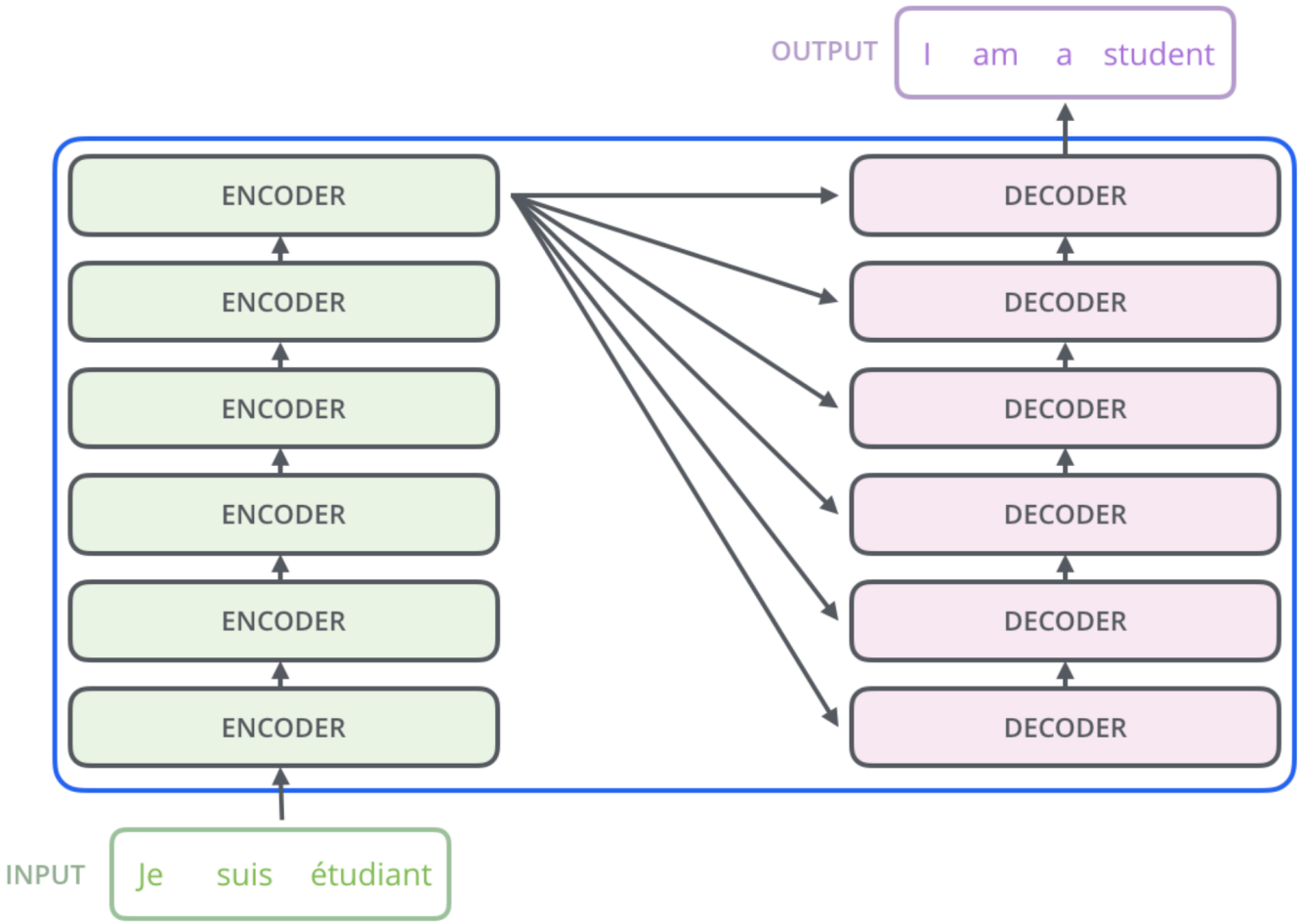
It adds an additional block to each transformer, called Encoder-Decoder Attention, and the self-attention blocks are somewhat different than in the encoder



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

Encoder-Decoder Attention

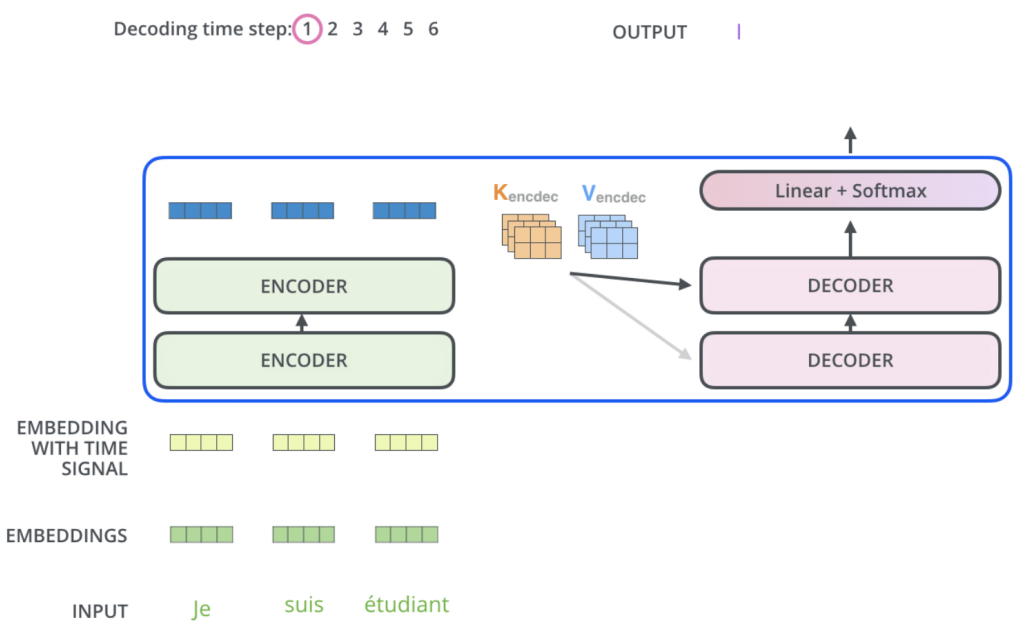
Encoder-decoder attention allows the decoder to attend to the hidden states from the last layer of the encoder, much as we saw with attention in seq2seq



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

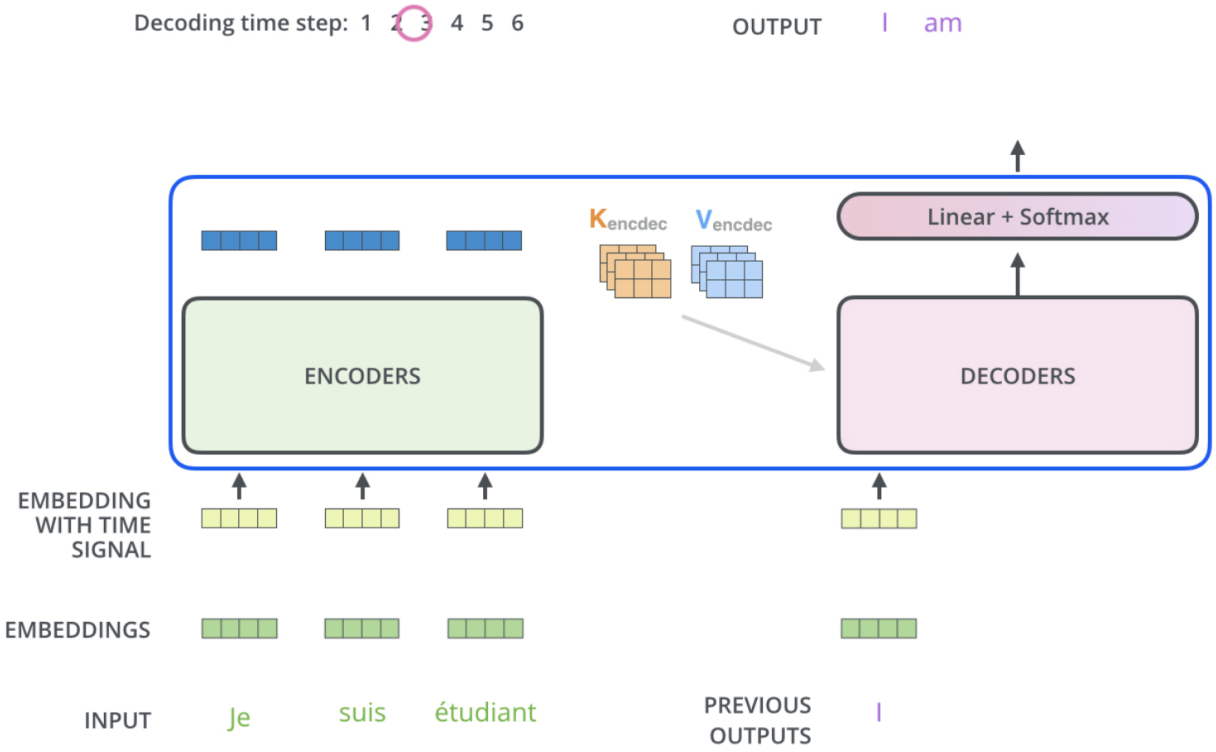
Encoder-Decoder Attention

Specifically, key and value outputs from the top encoder are used by each encoder-decoder attention block to help the decoder focus on appropriate positions in the input sequence



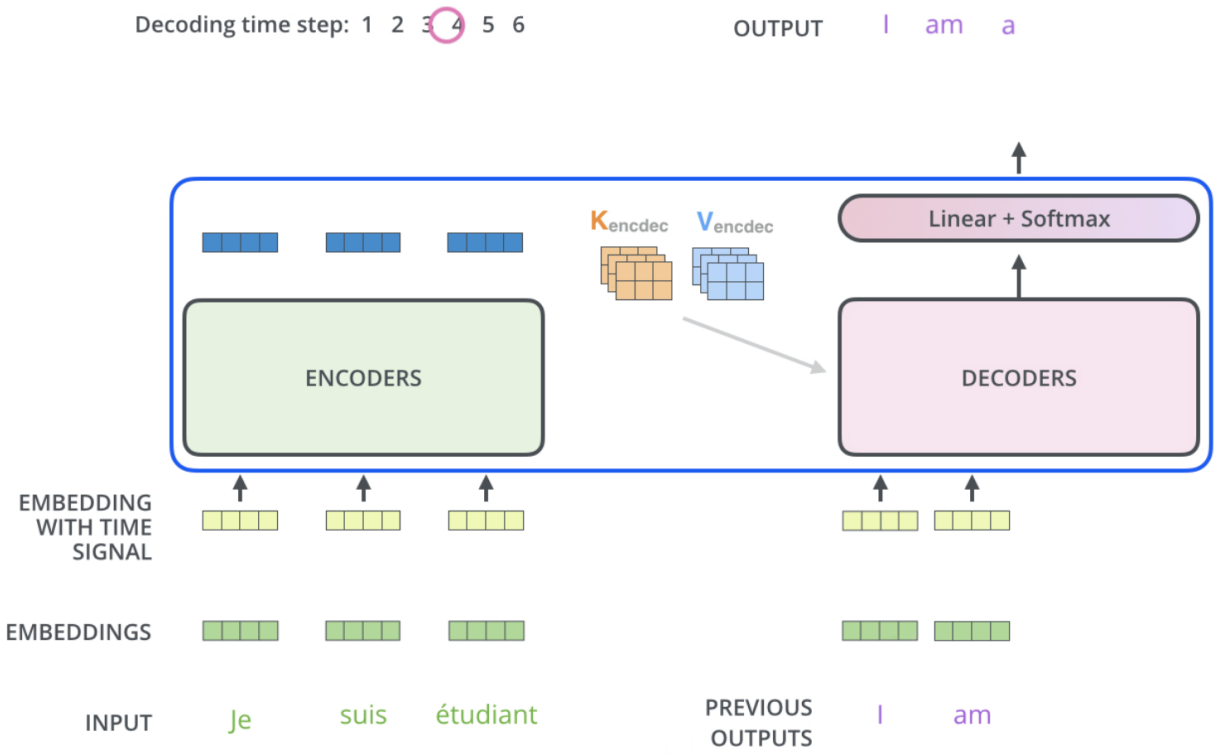
http://jalammar.github.io/illustrated-transformer/

Encoder-Decoder Attention



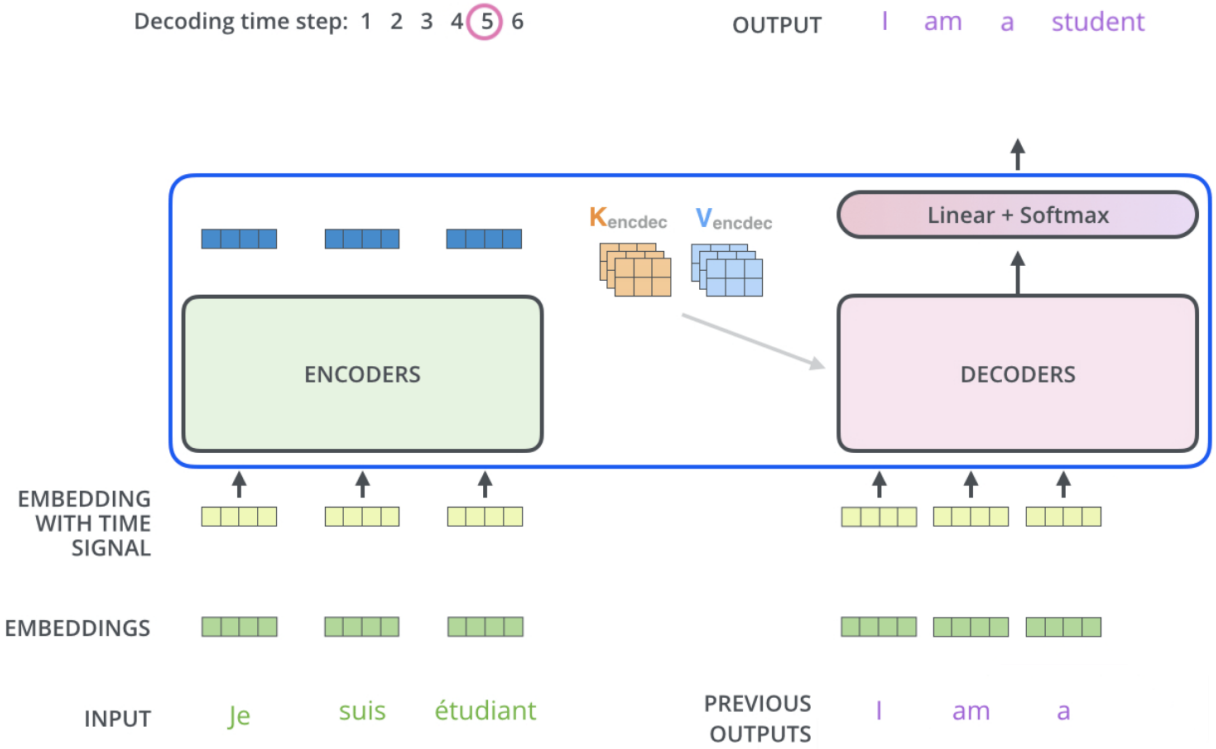
http://jalammar.github.io/illustrated-transformer/

Encoder-Decoder Attention



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

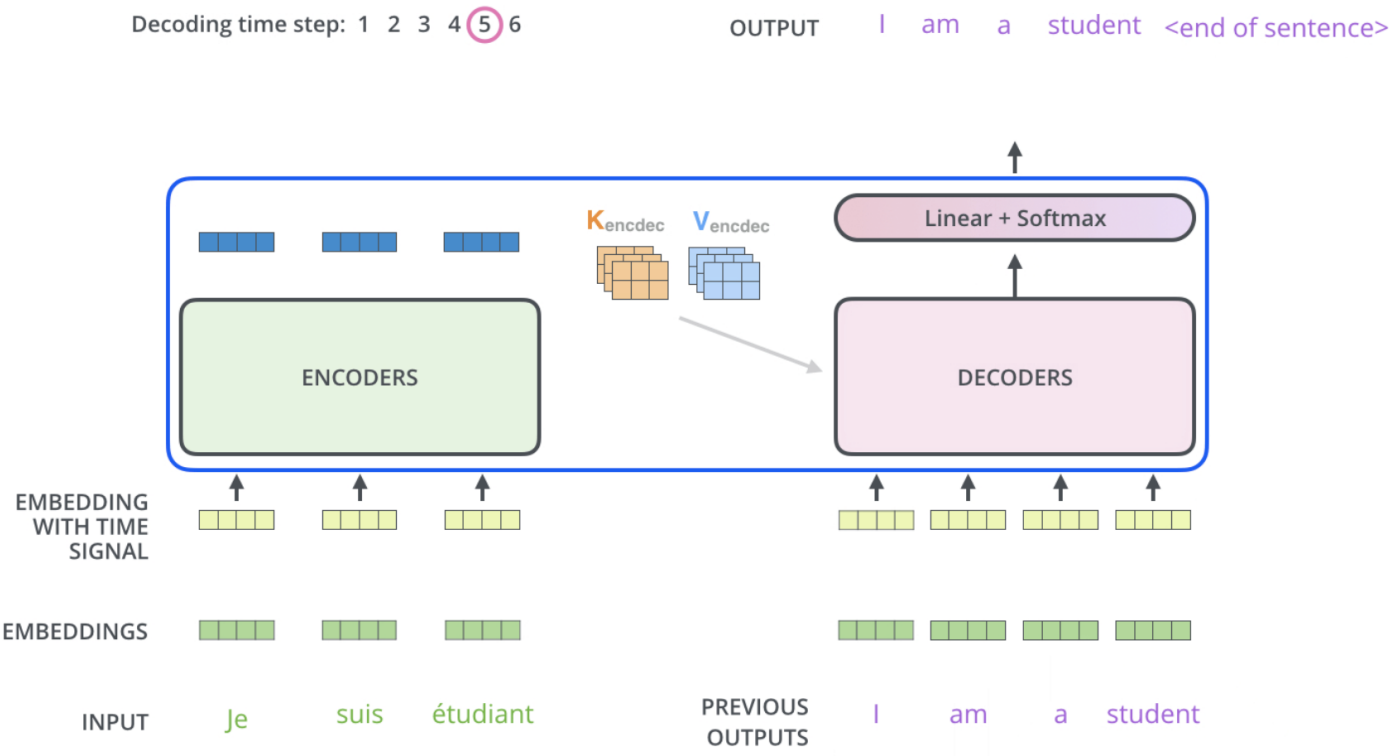
Encoder-Decoder Attention



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

Encoder-Decoder Attention

This considers sequentially, until an $< end >$ token is reached

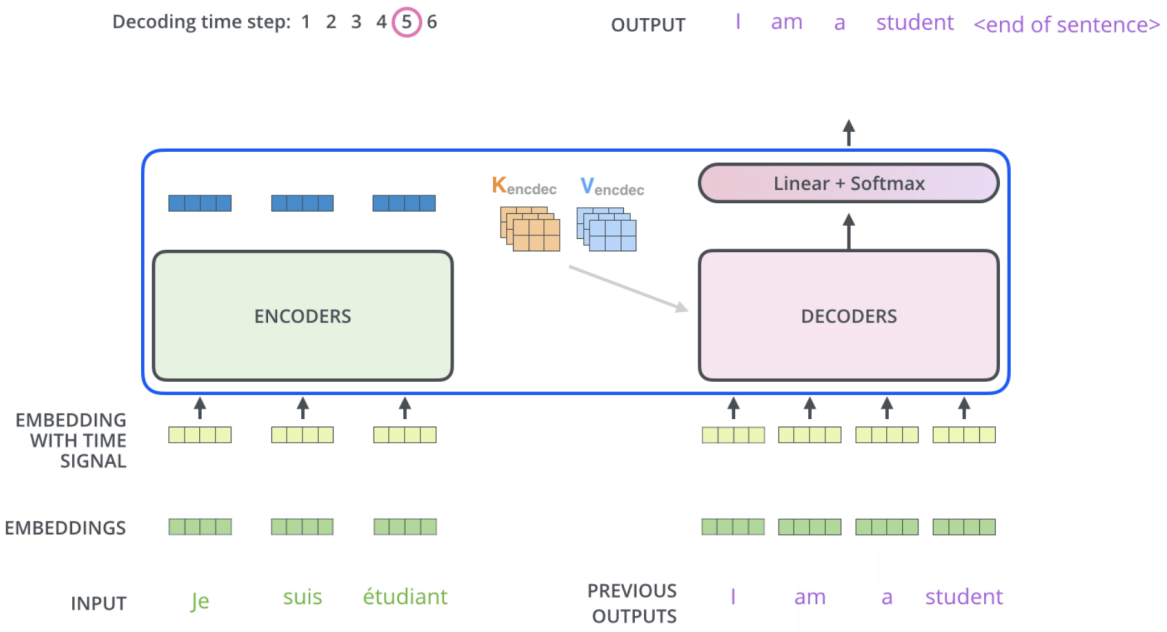


http://jalammar.github.io/illustrated-transformer/

It is analogous to how attention worked in vanilla seq2seq

Encoder-Decoder Attention

It continues sequentially, until an $< end >$ token is reached

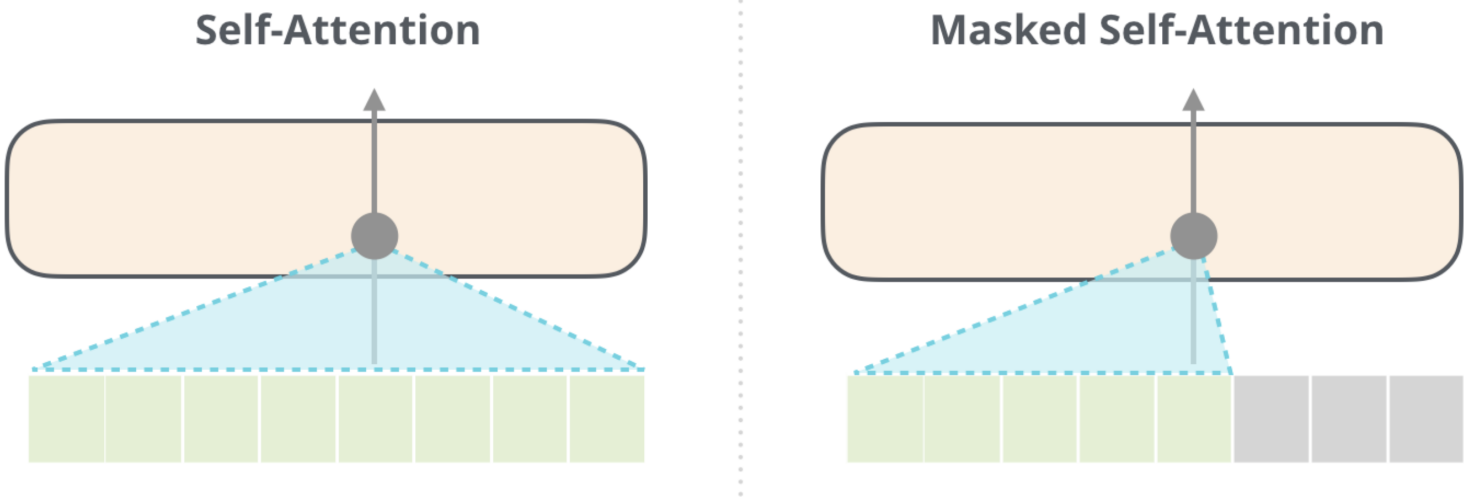


http://jalammar.github.io/illustrated-transformer/

It is also analogous to how the self-attention blocks work, except the key and value vectors come from the last encoder layer and the queries from the layer below (or the input, for the first layer)

The Decoder Uses Masked Self-Attention

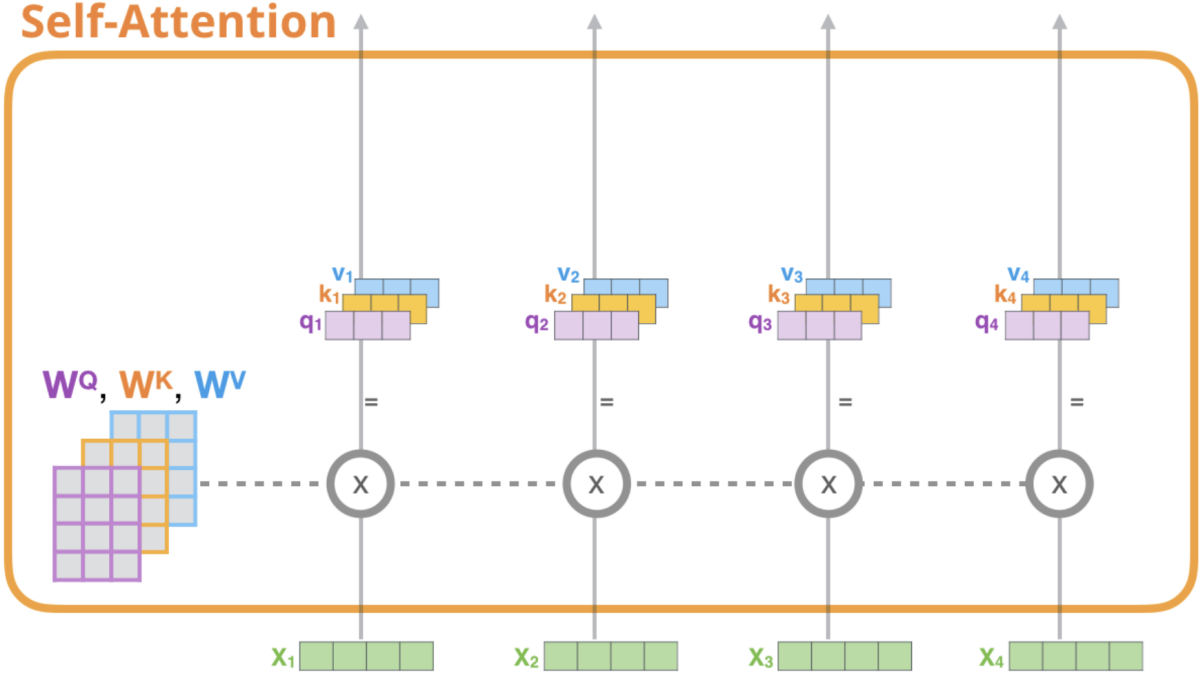
In the encoder self-attention blocks, a position can see tokens from the right. This is not the case for the decoder blocks. The idea here is akin to predicting the next word, given the previous contexts.



https://jalammar.github.io/illustrated-gpt2/

A Recap of Self-Attention without Masking: Step 1

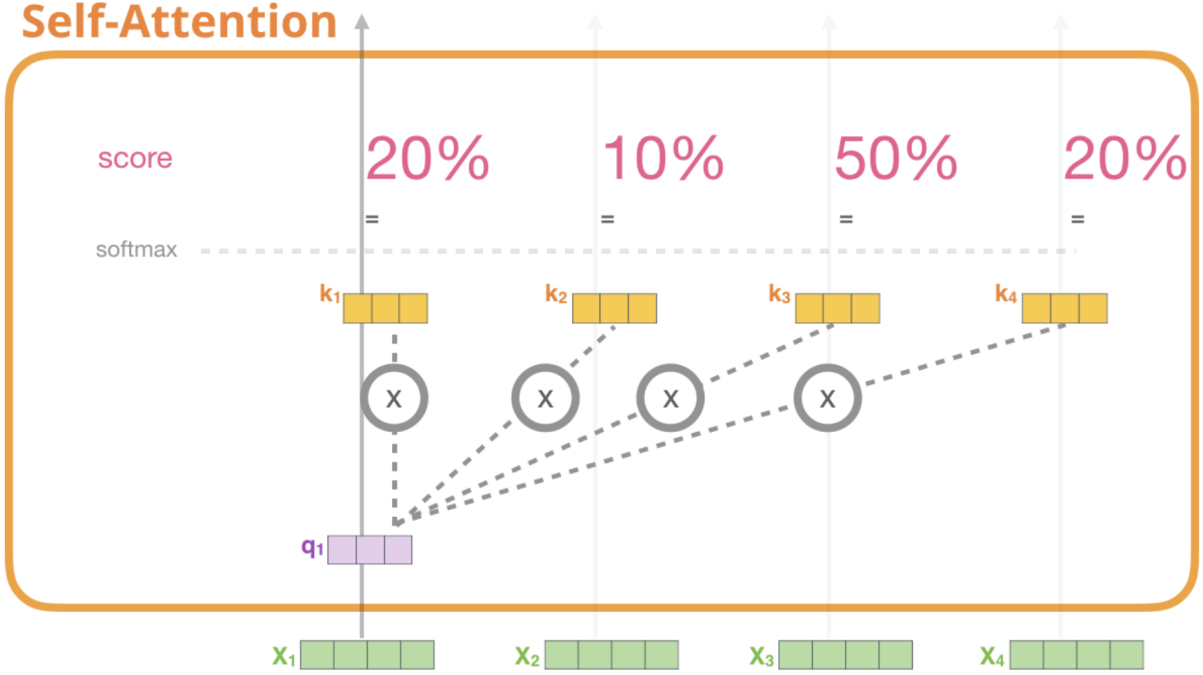
1) For each input token, create a query vector, a key vector, and a value vector by multiplying by weight Matrices W^Q, W^K, W^V



https://jalammar.github.io/illustrated-gpt2/

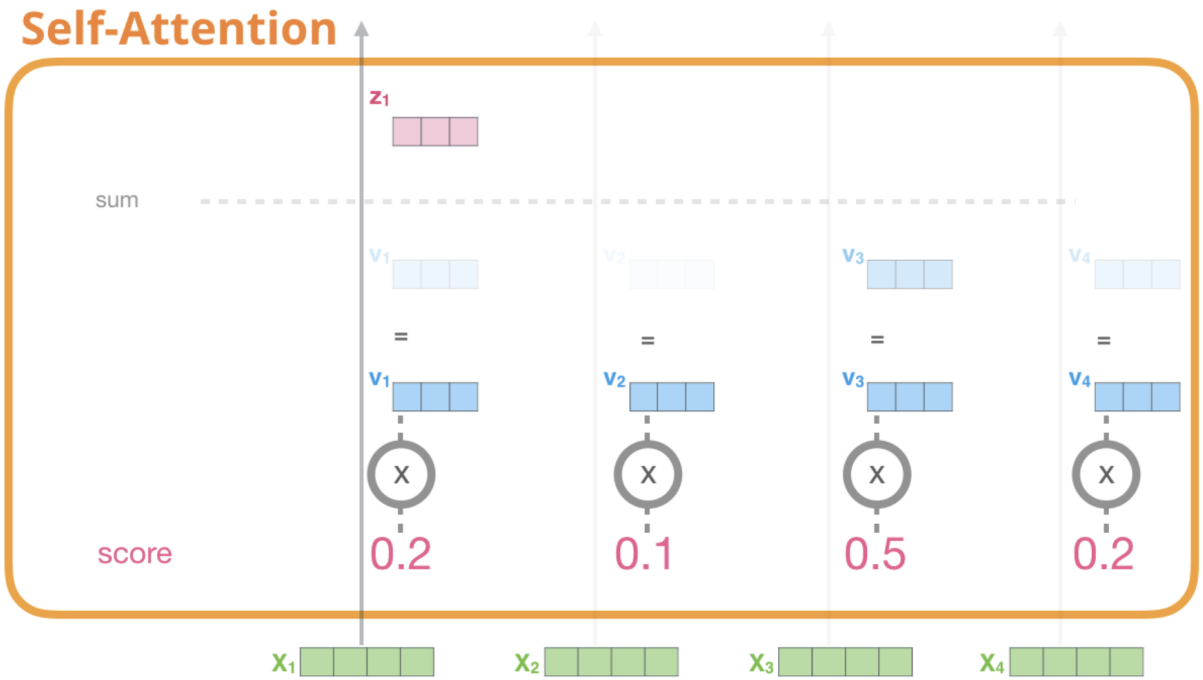
A Recap of Self-Attention without Masking: Step 2

2) Multiply (dot product) the current query vector, by all the key vectors, to get a score of how well they match



https://jalammar.github.io/illustrated-gpt2/

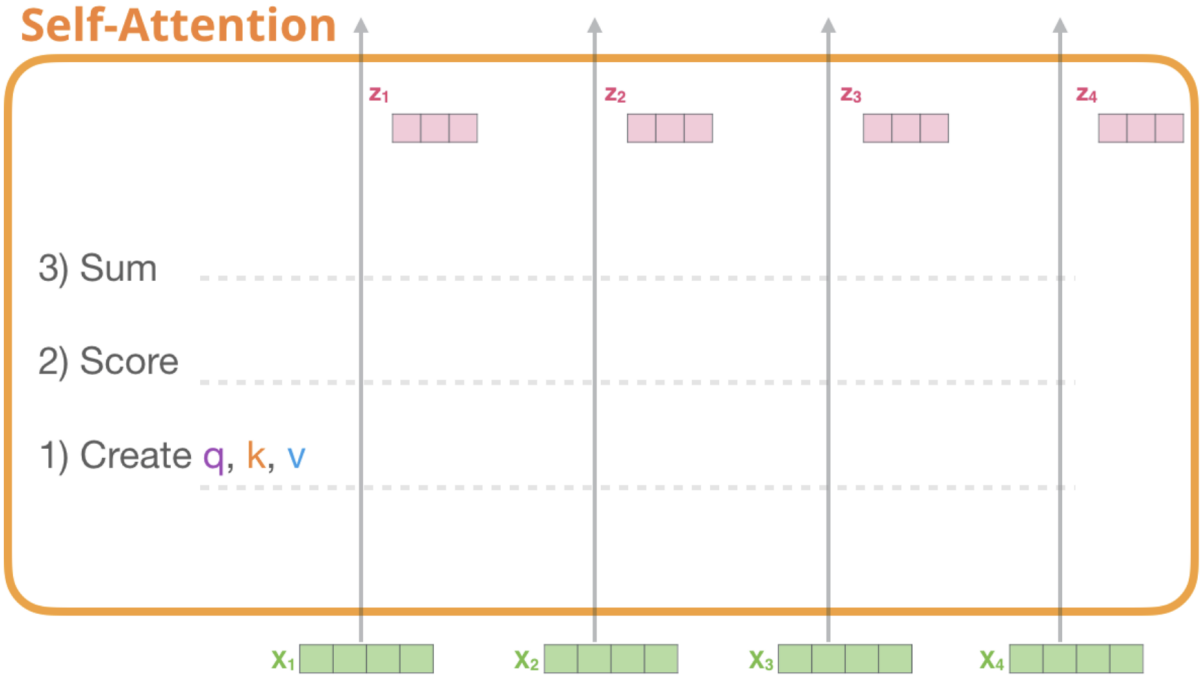
A Recap of Self-Attention without Masking: Step 3



The lower the score, the more transparent we're showing the value vector. That's to indicate how multiplying by a small number dilutes the values of the vector.

https://jalammar.github.io/illustrated-gpt2/

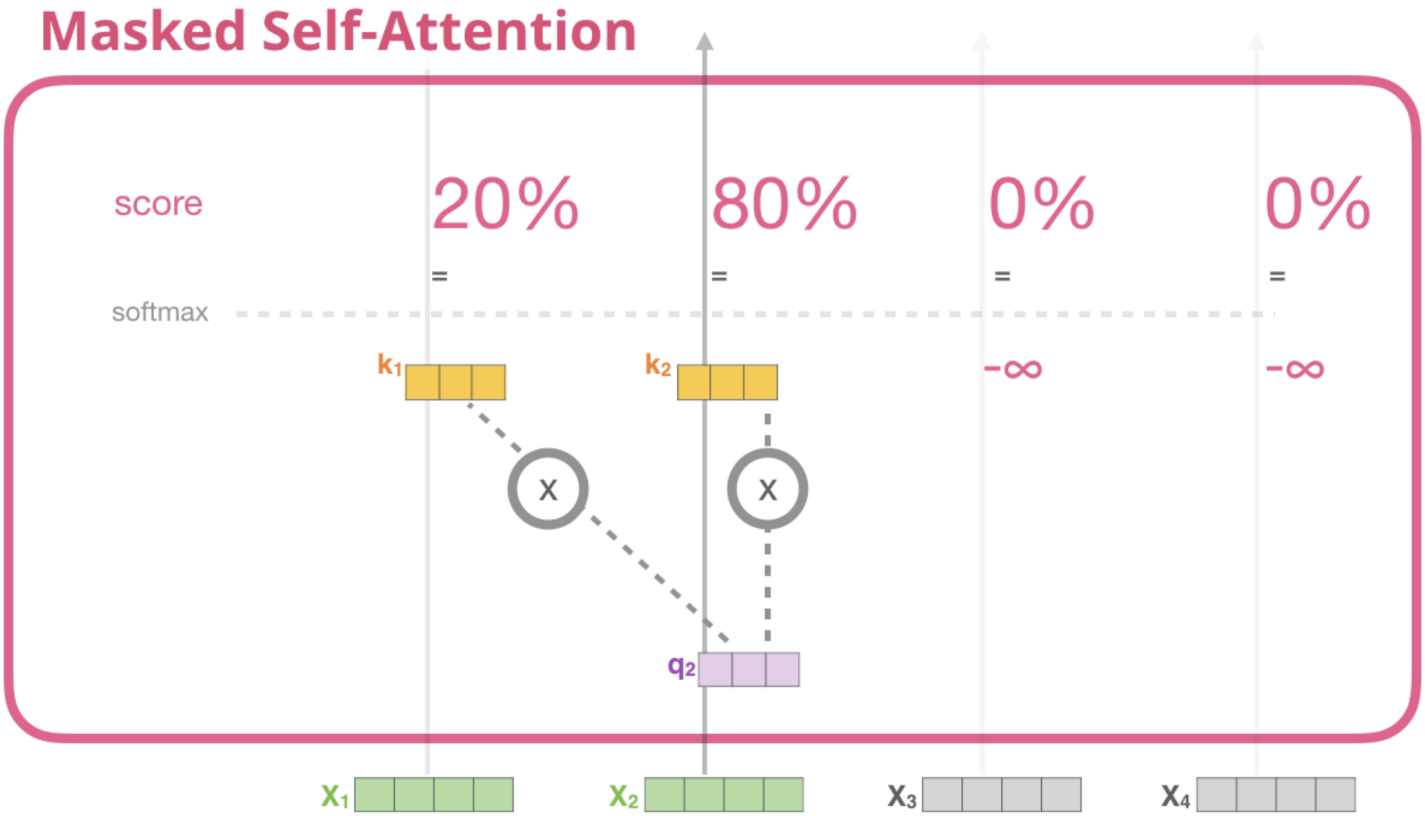
A Recap of Self-Attention without Masking: Step 4



https://jalammar.github.io/illustrated-gpt2/

Masked Self-Attention

Masked self-attention is identical to self-attention without masking, except at step 2.



https://jalammar.github.io/illustrated-gpt2/

Masked Self-Attention

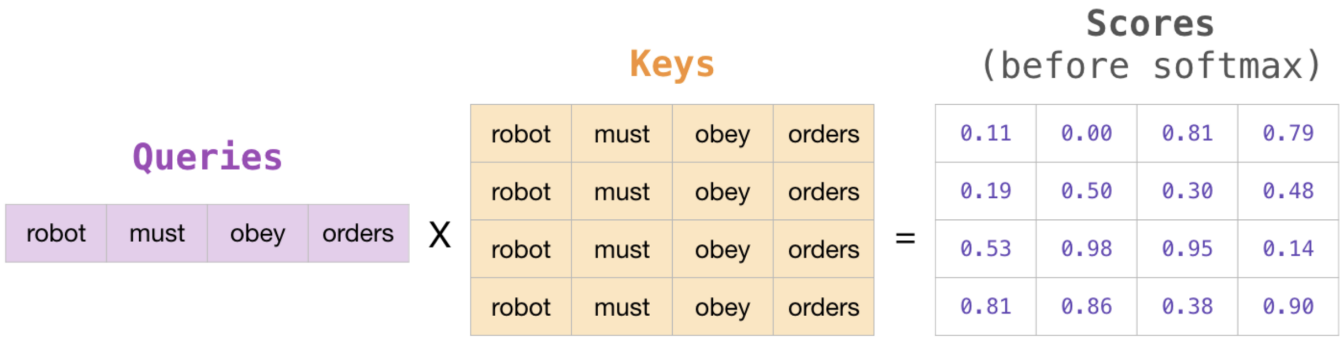
Consider a sentence with four words:

	Features				Labels
position:	1	2	3	4	
Example: 1	robot	must	obey	orders	must
2	robot	must	obey	orders	obey
3	robot	must	obey	orders	orders
4	robot	must	obey	orders	<eos>

https://jalammar.github.io/illustrated-gpt2/

Masked Self-Attention

Calculate the attention scores by multiplying the query matrix by the key matrix:



https://jalammar.github.io/illustrated-gpt2/

In practice, each word is associated with a vector

Masked Self-Attention

After creating the scores, multiply by an attention mask, which will be a triangle

Scores (before softmax)

0.11	0.00	0.81	0.79
0.19	0.50	0.30	0.48
0.53	0.98	0.95	0.14
0.81	0.86	0.38	0.90

Apply Attention Mask →

Masked Scores (before softmax)

0.11	-inf	-inf	-inf
0.19	0.50	-inf	-inf
0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90

https://jalammar.github.io/illustrated-gpt2/

Masked Self-Attention

Then apply softmax

Masked Scores
(before softmax)

0.11	-inf	-inf	-inf
0.19	0.50	-inf	-inf
0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90

Softmax
(along rows)

Scores

1	0	0	0
0.48	0.52	0	0
0.31	0.35	0.34	0
0.25	0.26	0.23	0.26

https://jalammar.github.io/illustrated-gpt2/

Outline

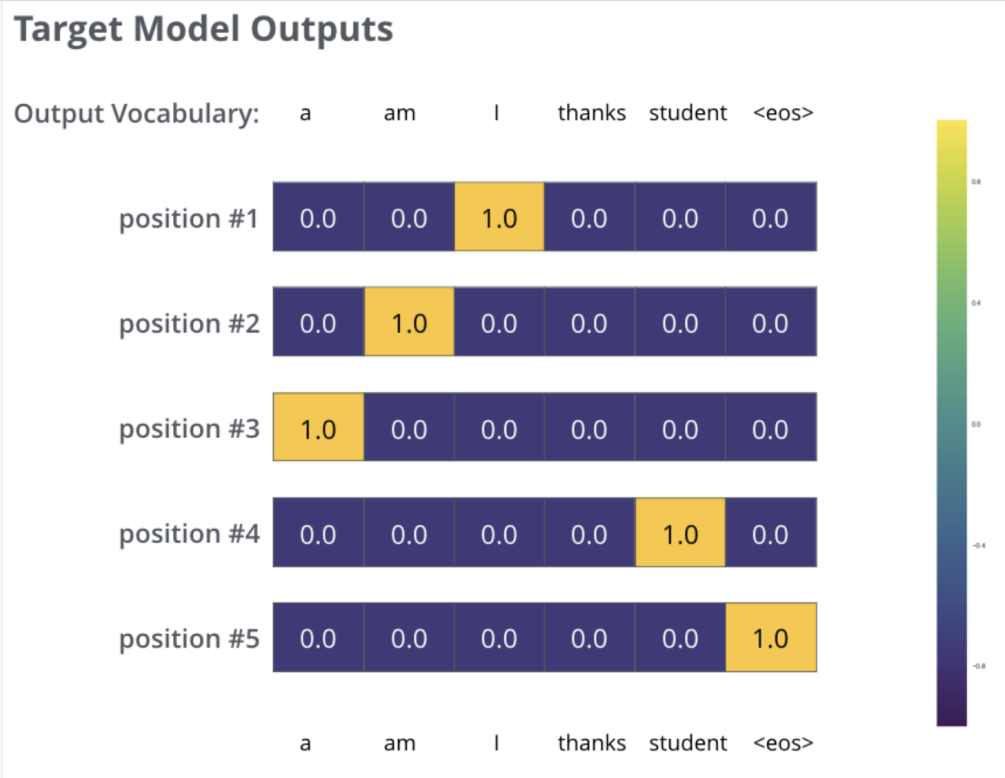
Linear and Softmax Layers

- How do we turn the vector outputs from the decoder into a word?
- The linear layer is a vanilla fully connected neural network that projects the vector produced by the decoder stack into a much larger vector, called the logits vector
- For example, if our vocabulary has 10K words, the logits vector would have dimensionality 10K, with each value corresponding to the score of a unique word
- Softmax turns these scores into probabilities
- The cell with the highest probability is chosen (if doing greedy decoding; or we may keep track of more than one, if doing beam search

Outline

Training

The training objective is just a cross-entropy loss

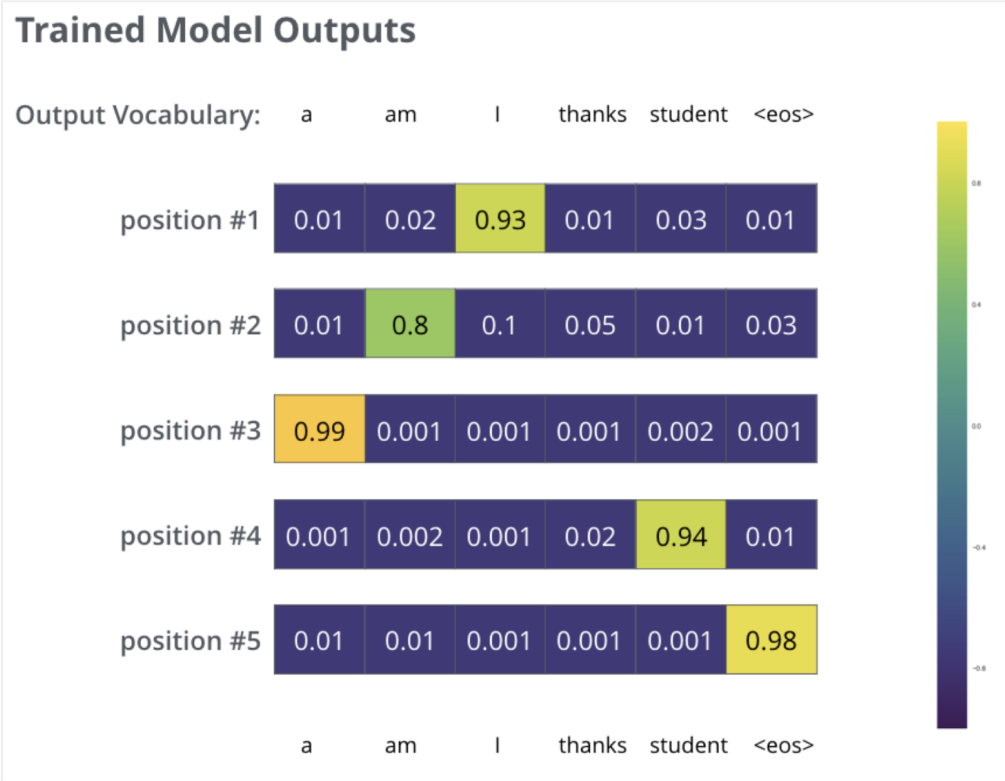


The targeted probability distributions we'll train our model against in the training example for one sample sentence.

https://jalammar.github.io/illustrated-gpt2/

Training

The training objective is just a cross-entropy loss



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.

https://jalammar.github.io/illustrated-gpt2/

Beam Search Decoding

- At each step, keep track of the k most plausible partial translations (hypotheses)
- k is the beam size
- The plausibility score is just the probability of the sequence:

$$logP(y_1 ... y_t|x) = \sum_{i=1}^{t} logP(y_i|y_1 ... y_{i-1}, x)$$

Beam Search

- When a hypothesis produces the $< End >$ token, put it aside
- Continue exploring other hypotheses
- Stop either once reach time step t (hyperparameter) or once collected at least N (hyperparameter) completed hypotheses
- Normalize by score length when selecting the final hypothesis