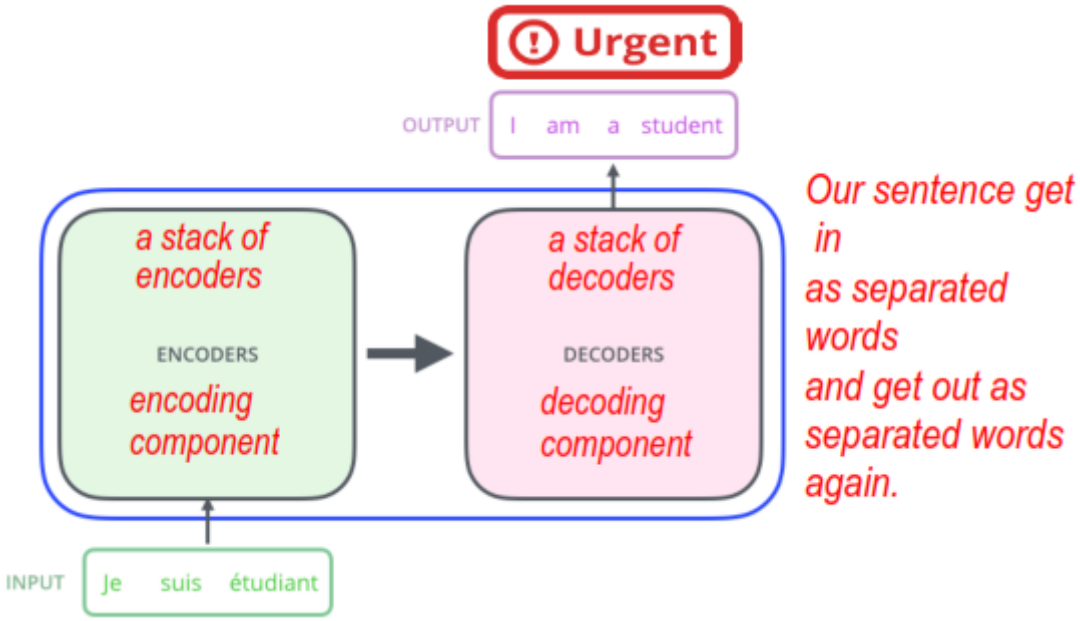
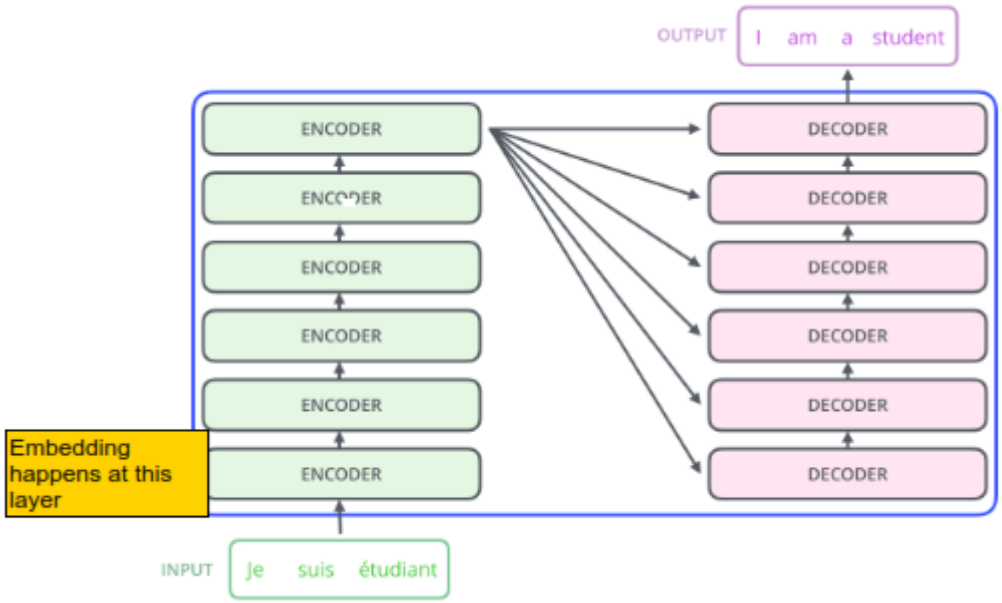
The Illustrated Transformer

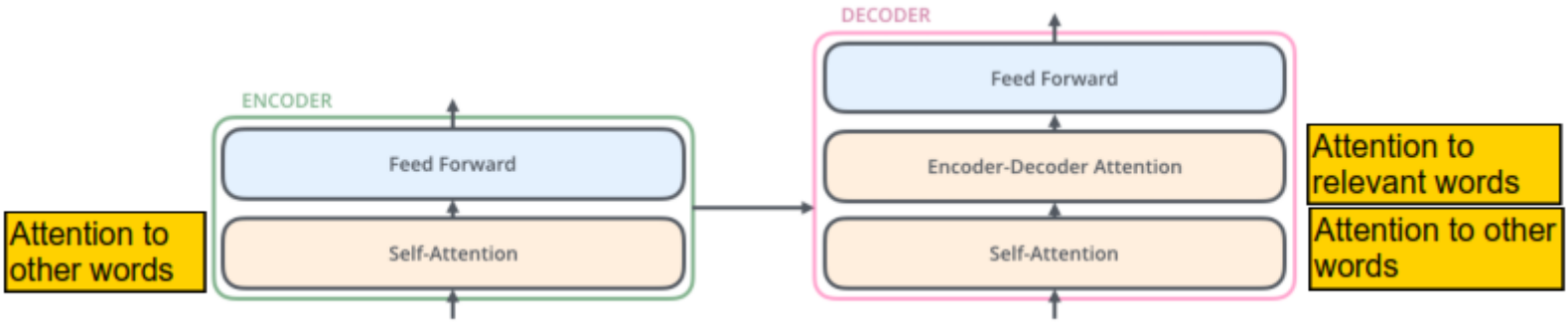
Transformer: a model that uses attention to boost the speed with which these models can be trained.

Attention: is a concept that helped improve the performance.

TensorFlow implementation of it is available as a part of the Tensor2Tensor package.

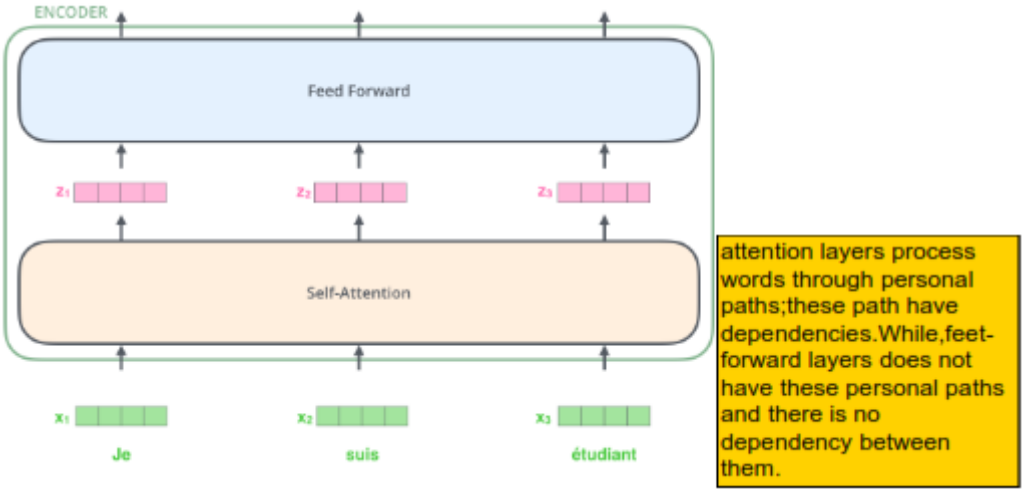






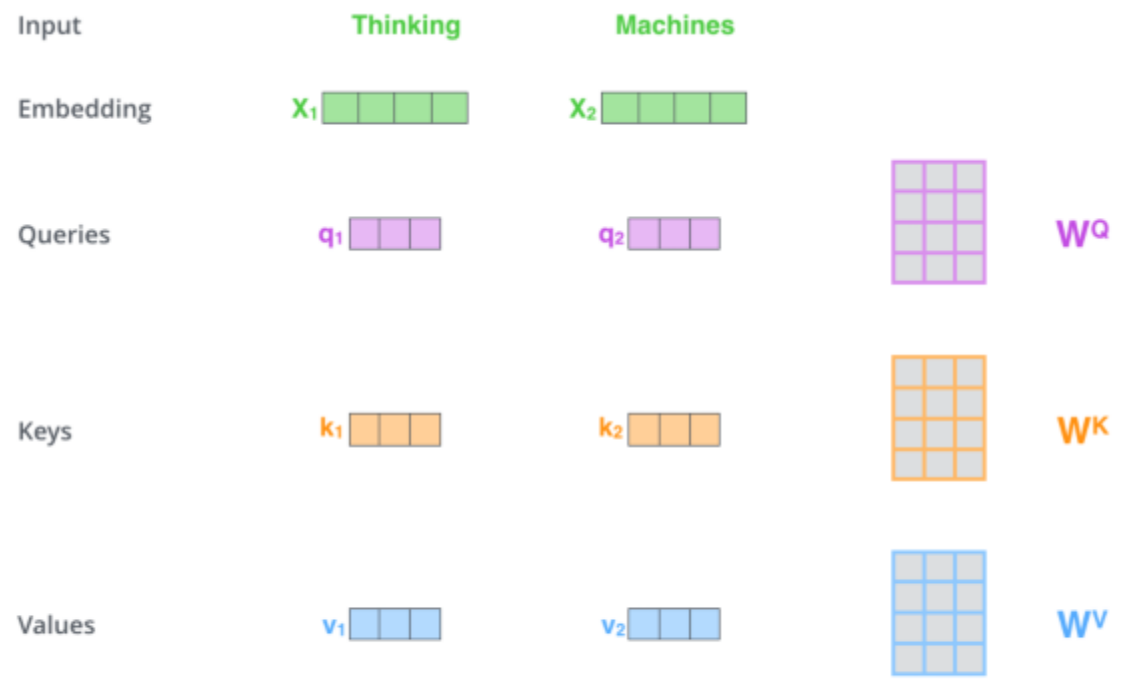
Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes. The embedding only happens in the **bottom-most encoder**. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512 – In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. The size of this list is hyperparameter we can set basically it would be the length of the longest sentence in our training dataset.



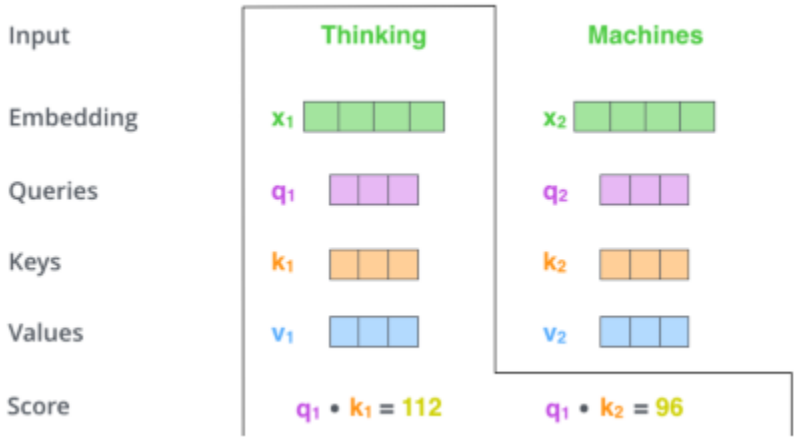
Self-Attention in Detail

Step 1: Embedding Vector * 3 Matrices trained during Training(Weight Matrix) = 3 New Vectors (Query, Key and Value); accordingly, from each word we have three vectors (Abstract Values for Attention Calculation).

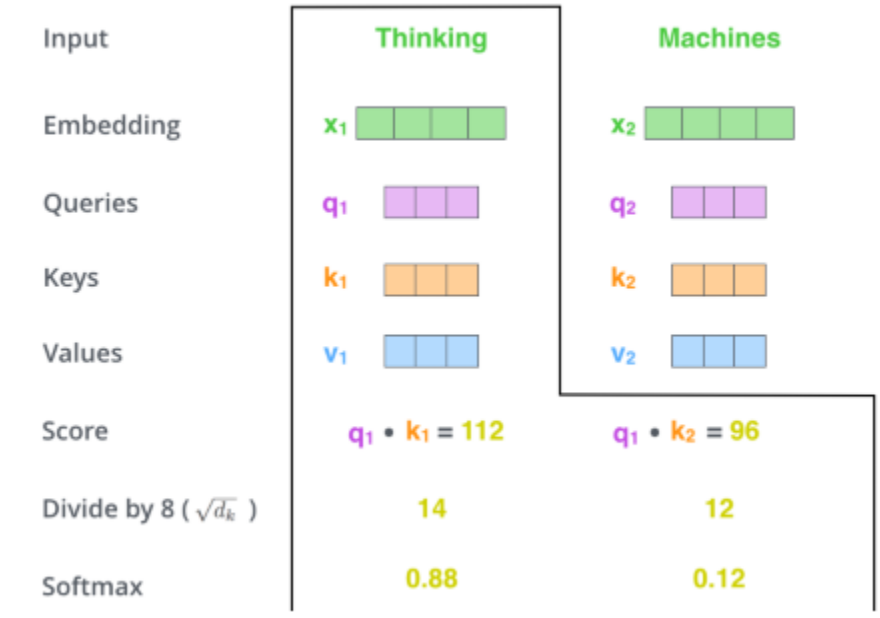


Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

Step 2: (self-attention is calculated for each word by a score; Dot product of the query vector and the key vector) 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. 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.



Step 3 and 4: The third and fourth steps are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default, then pass the result through a SoftMax operation. SoftMax normalizes the scores so they're all positive and add up to 1.

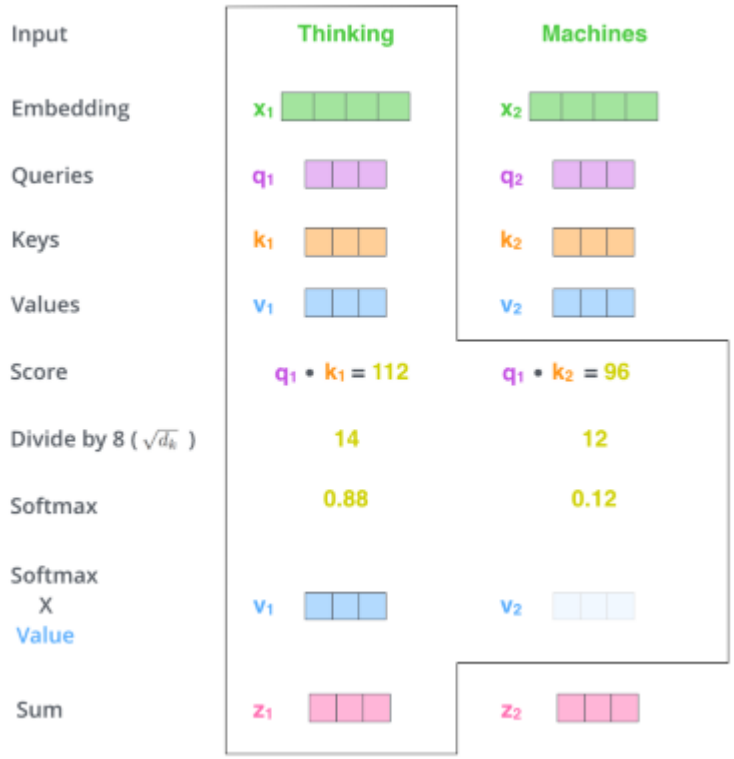


This SoftMax score determines how much each word will be expressed at this position.

Step 5 and 6: 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-

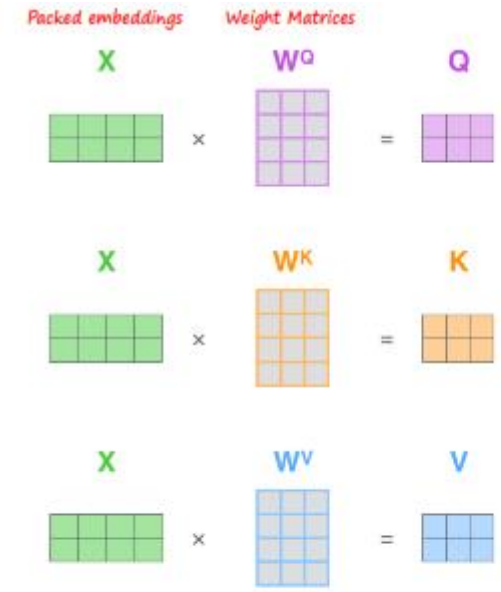
attention layer at this position (for the first word).



In the actual implementation, however, this calculation is done in matrix form for faster processing

Matrix Calculation of Self-Attention

Create a Matrix of Vectors and multiply them by the weight matrices trained before.



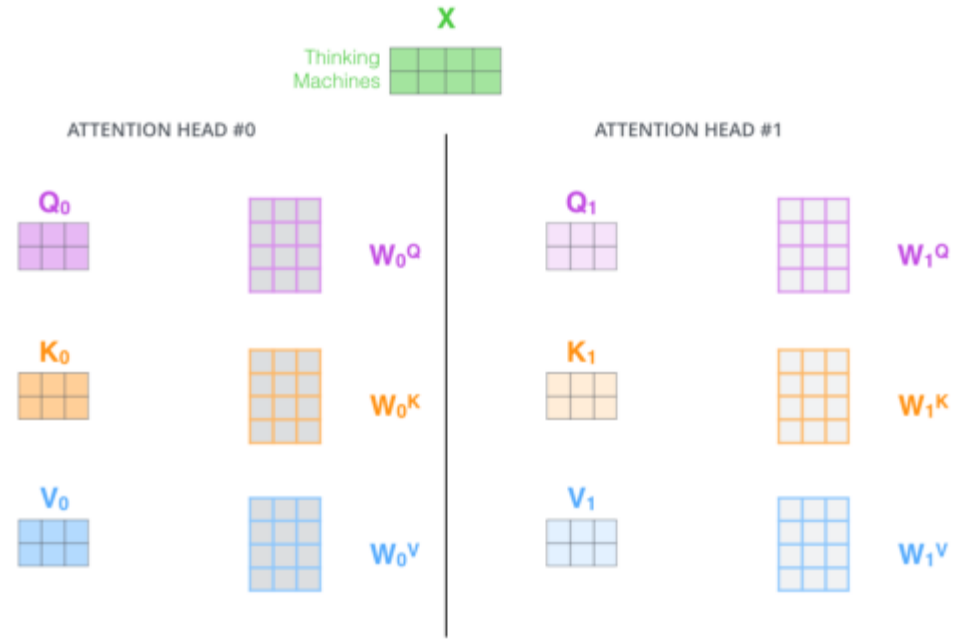
Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

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

The self-attention calculation in matrix form

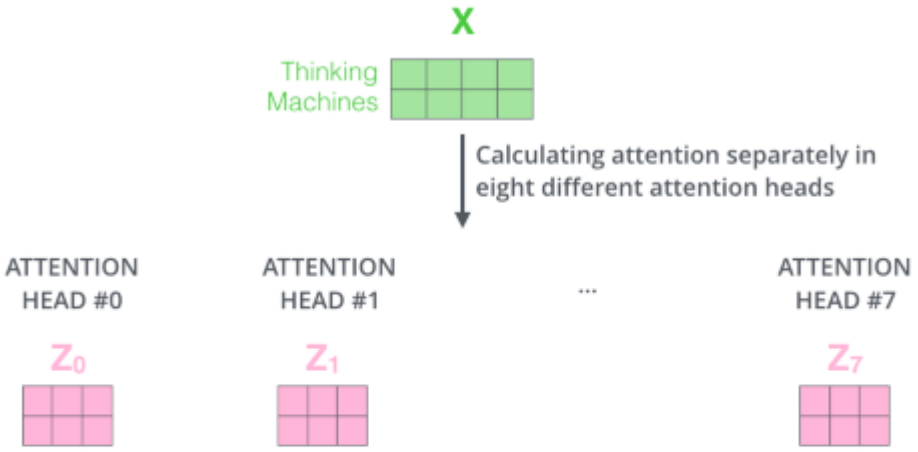
The Beast with Many Heads (Multi-Headed Attention)

1. **It expands the model's ability to focus on different positions.**
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).**

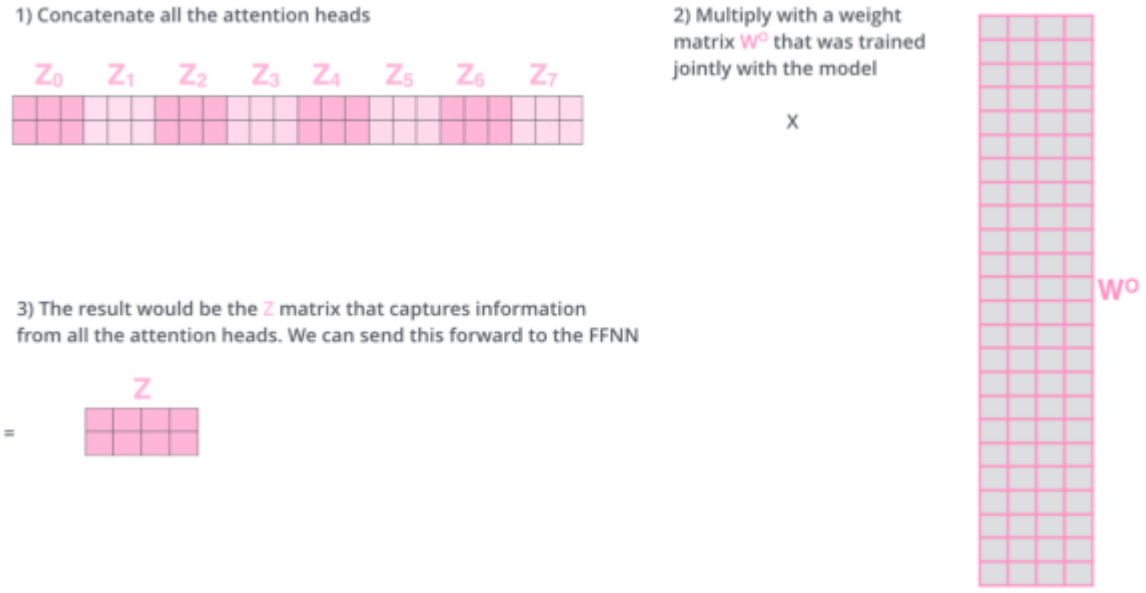


With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

*Embedding Vector * 3 Weight Matrices trained during Training = 3 New Vectors(Query, Key and Value)*

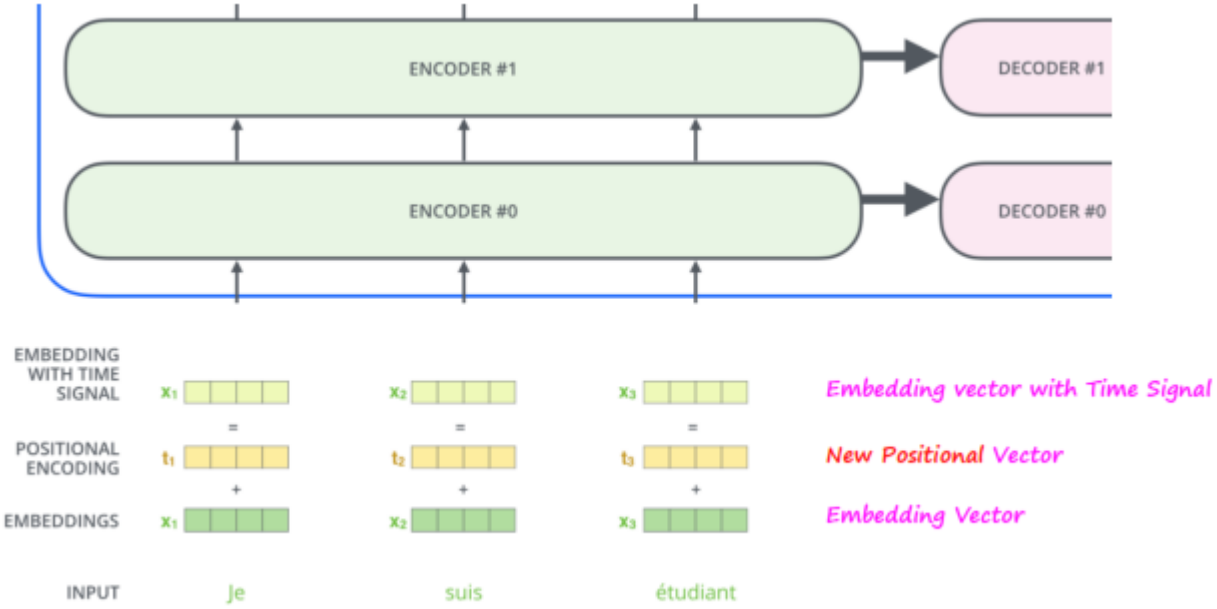


This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.



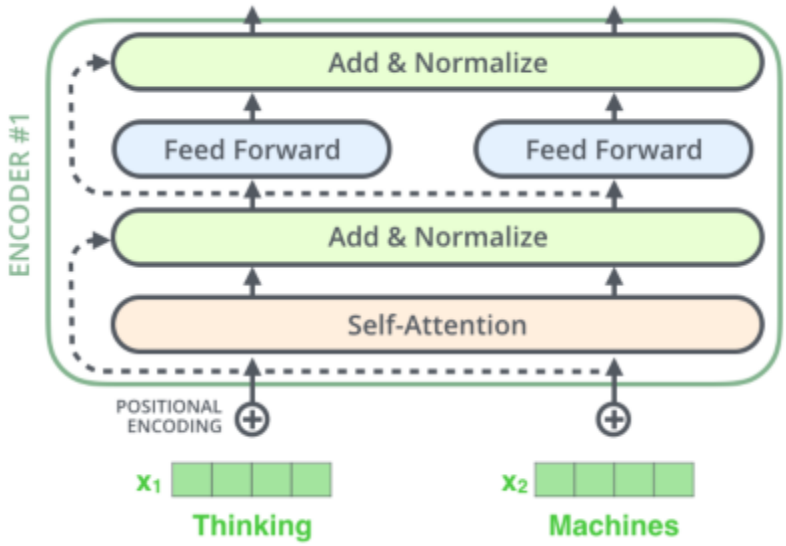
Representing the Order of The Sequence Using Positional Encoding

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.

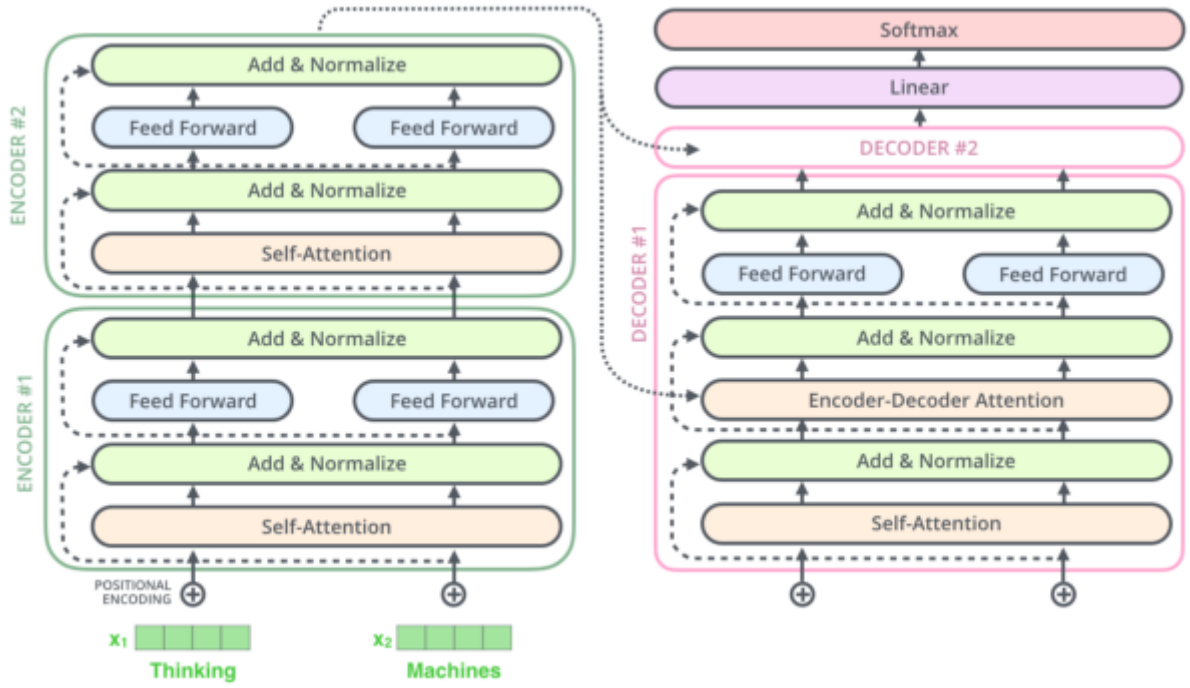


The Residuals (Residual Connection, Layer Normalization (Add a new Layer and normalize the vector))

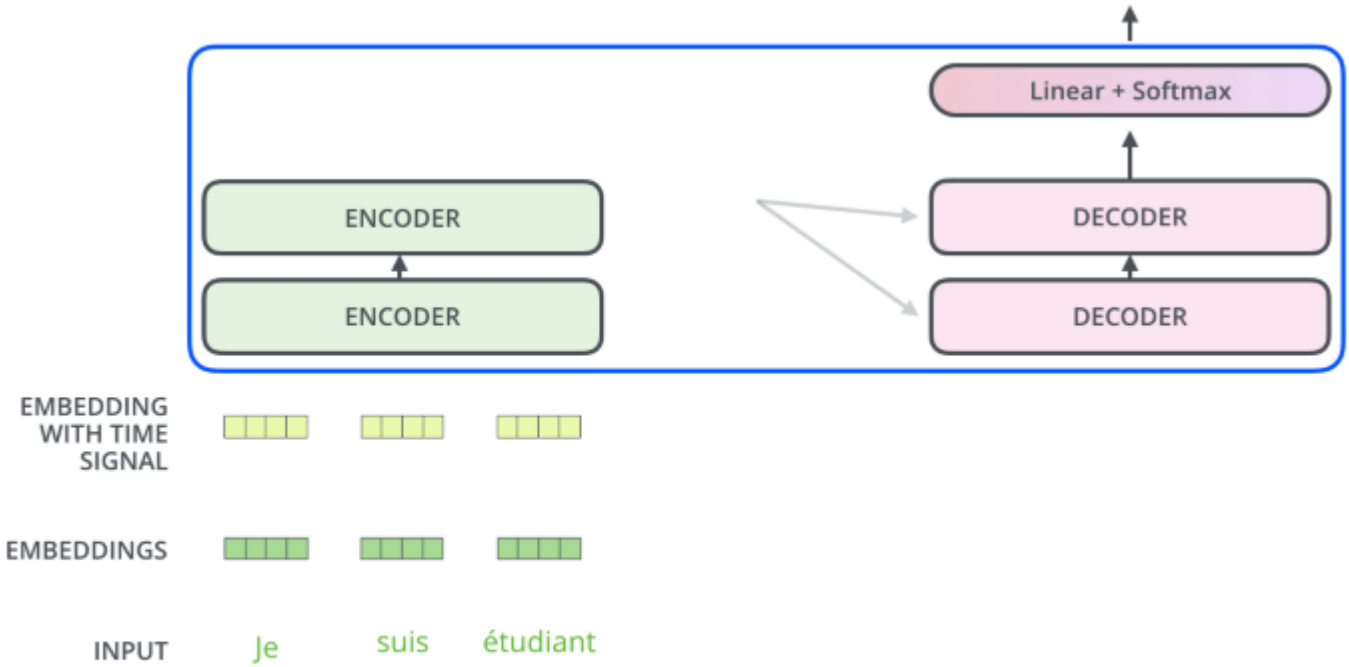
One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer normalization step.



Each Layer has a Normalizer Layer; accordingly, Encoders have two Normalizer Layers while Decoders have three normalizers.

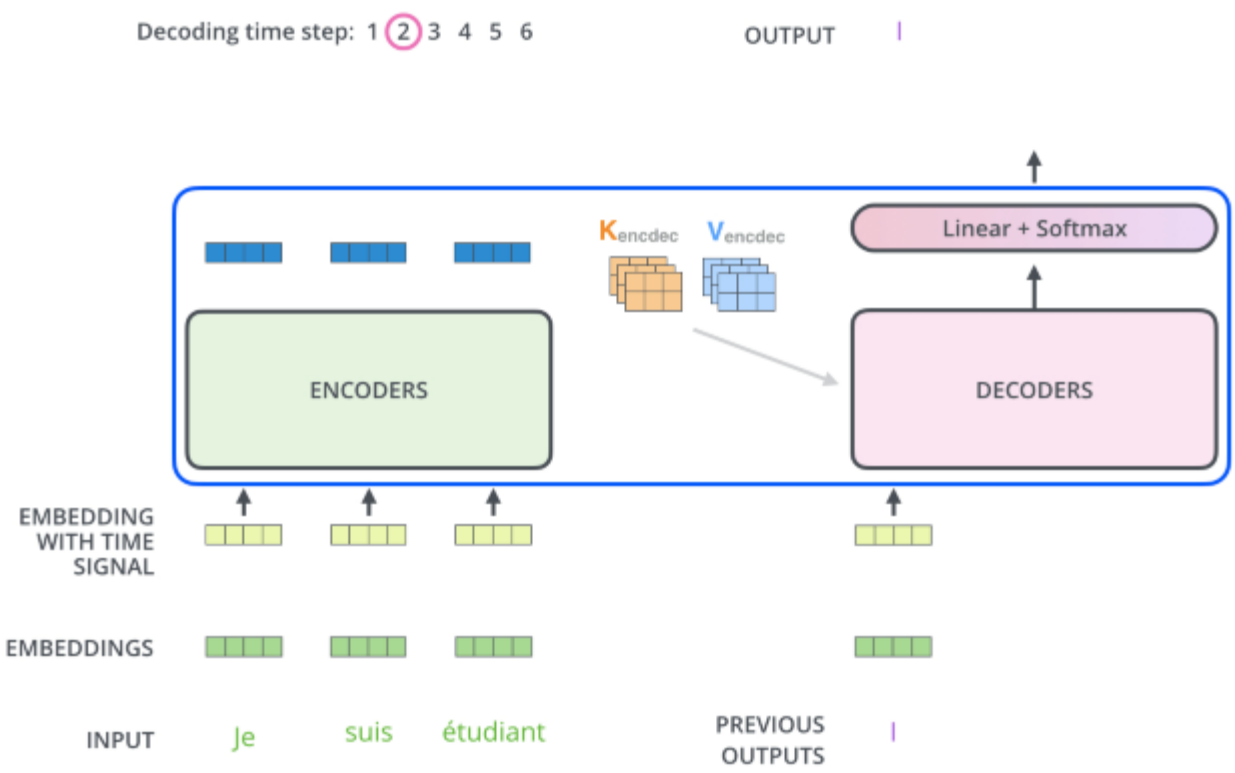


The Decoder Side



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).

Highly Important



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.

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.

Output Vocabulary

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

The output vocabulary of our model is created in the preprocessing phase before we even begin training.

Once we define our output vocabulary, we can use a vector of the same width to indicate each word in our vocabulary. This also known as one-hot encoding. So for example, we can indicate the word "am" using the following vector:

Output Vocabulary

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

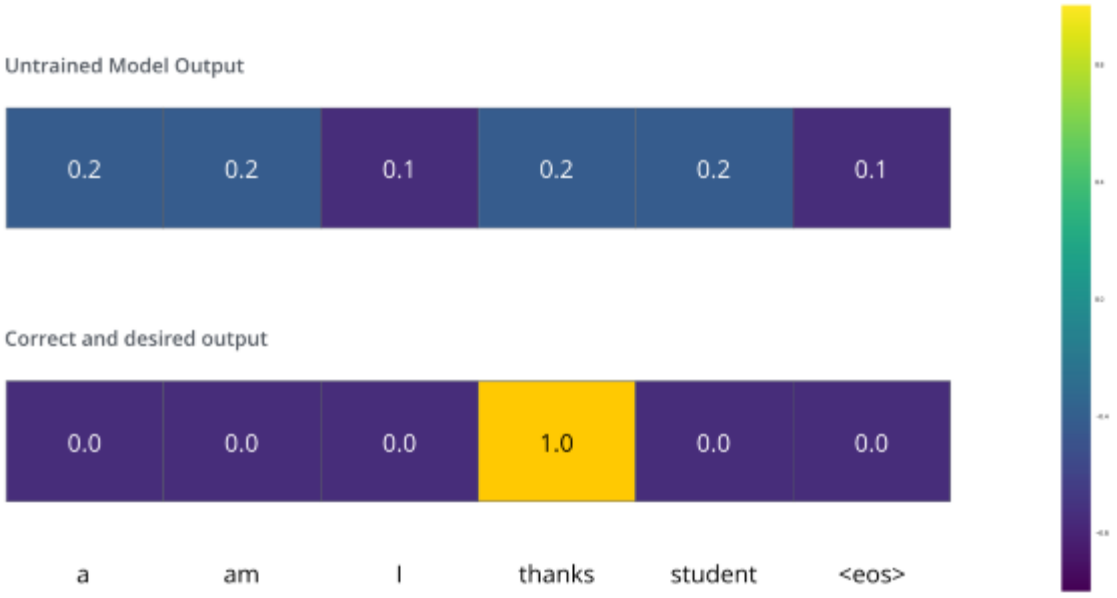
One-hot encoding of the word "am"

0.0	1.0	0.0	0.0	0.0	0.0

Example: one-hot encoding of our output vocabulary

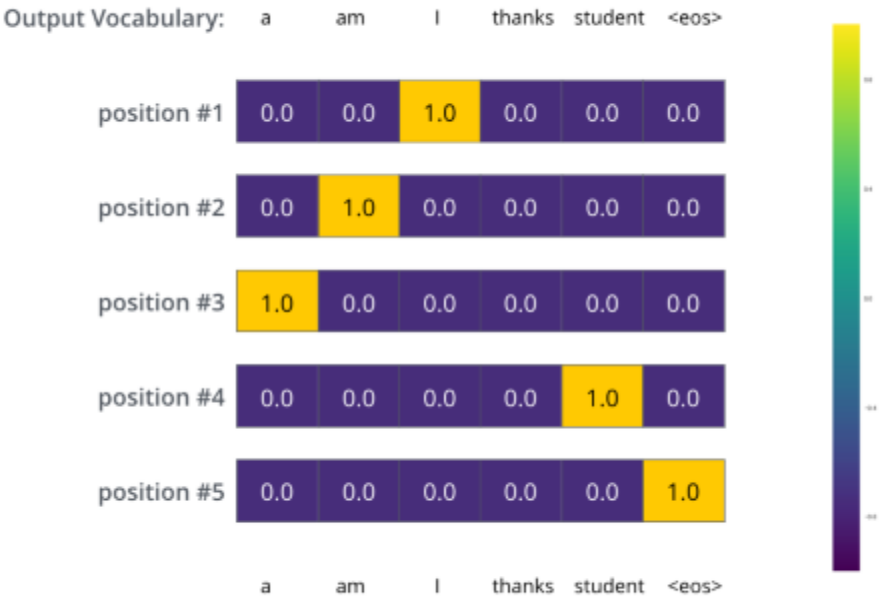
The Loss Function

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".

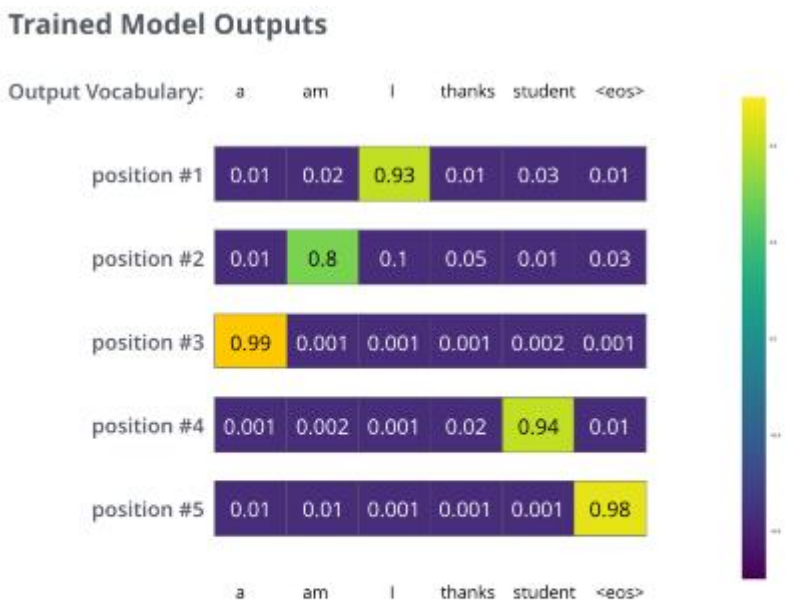


Since the model's parameters (weights) are all initialized randomly, the (untrained) model produces a probability distribution with arbitrary values for each cell/word. We can compare it with the actual output, then tweak all the model's weights using backpropagation to make the output closer to the desired output.

Target Model Outputs



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



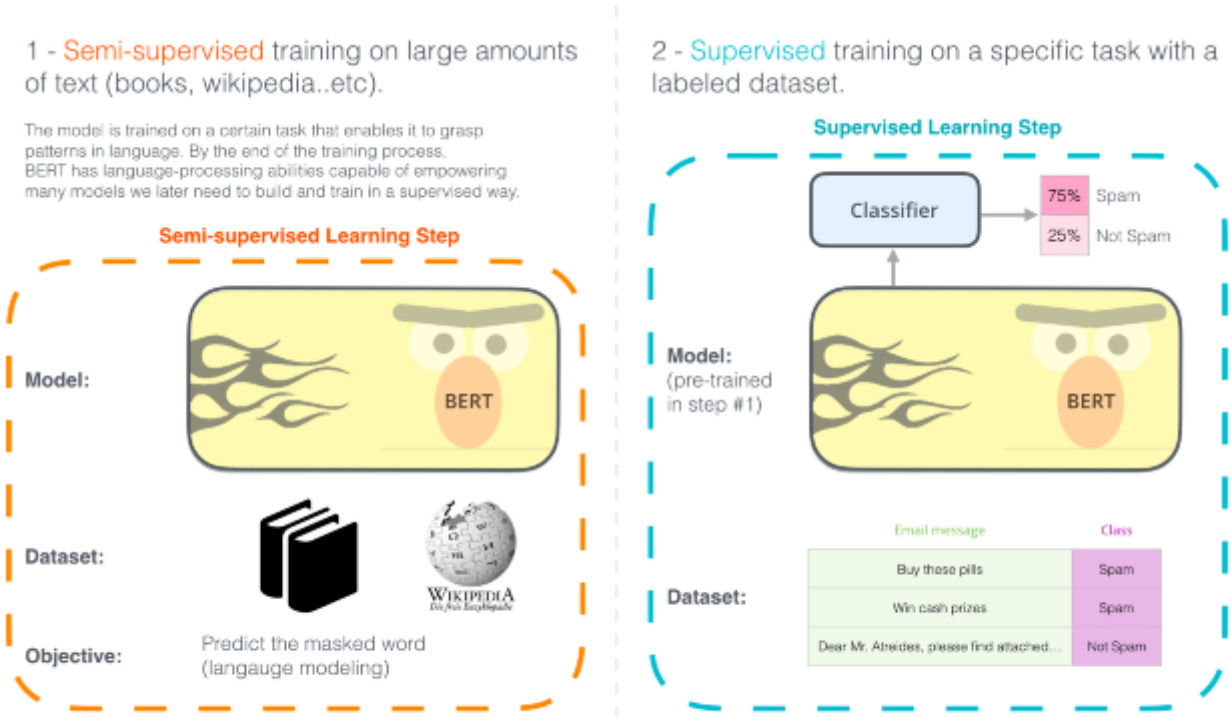
Hopefully upon training, the model would output the right translation we expect. Of course it's no real indication if this phrase was part of the training dataset (see: cross validation). Notice that every position gets a little bit of probability even if it's unlikely to be the output of that time step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

Bert

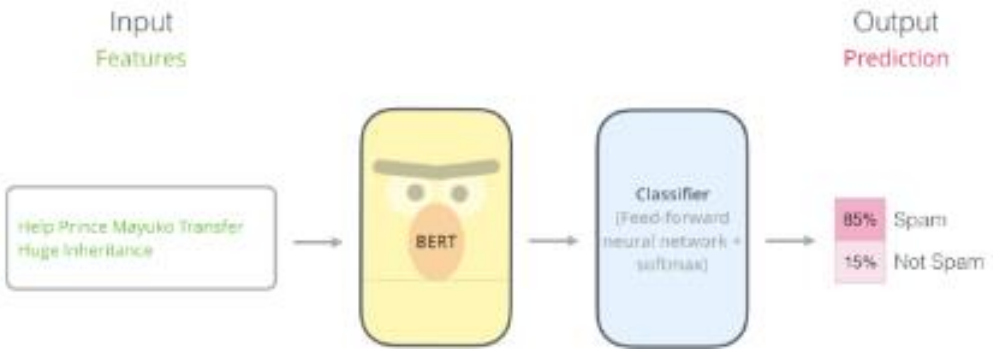
Our conceptual understanding of how best to represent words and sentences in a way that best

captures underlying meanings and relationships is rapidly evolving.



The two steps of how BERT is developed. You can download the model pre-trained in step 1 (trained on un-annotated data), and only worry about fine-tuning it for step 2. [Source for book icon].

Wanna use Bert? Train your Classifier;minimal changes that happen to Bert due to this classification training will helps you in sentence classification.



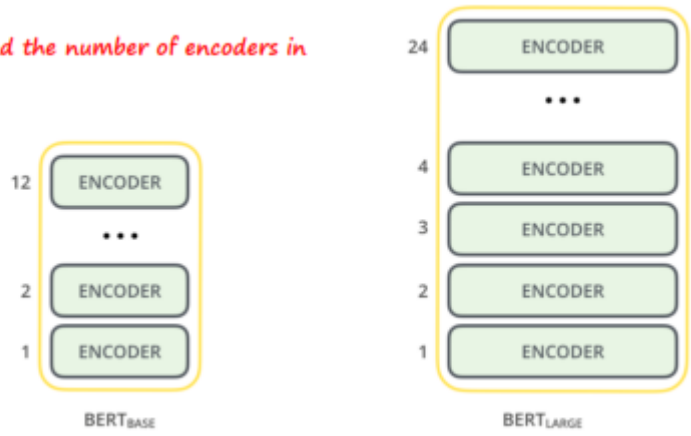
The paper presents two model sizes for BERT:

Open AI
- BERT BASE – Comparable in size to the OpenAI Transformer in order to compare performance

State of the Art
- BERT LARGE – A ridiculously huge model which achieved the state of the art results reported in the paper

Attention: Bert base is comparable to Open AI Transformer in size and not anything else (Bear it in mind that Open AI Transformers are not a type of Bert; but also they are sack of Decoders and not sack of Encoders).

Bert is an Encoder Stack;Mind the number of encoders in different types of the Bert.



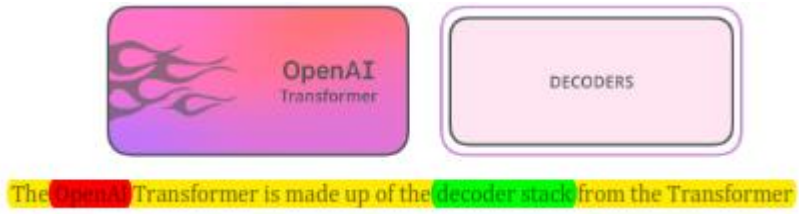
Both BERT model sizes have a large number of encoder layers (which the paper calls Transformer Blocks) – twelve for the Base version, and twenty-four for the Large version. These also have larger feedforward-networks (768 and 1024 hidden units respectively), and more attention heads (12 and 16 respectively) than the default configuration in the reference implementation of the Transformer in the initial paper (6 encoder layers, 512 hidden units, and 8 attention heads). Bert is just a sack of Encoders; accordingly, it receives sequences, turn them into embedded sequence and pass it to another encoder; finally, this embedded sequence gets into a decoder.

Word Embedding Recap

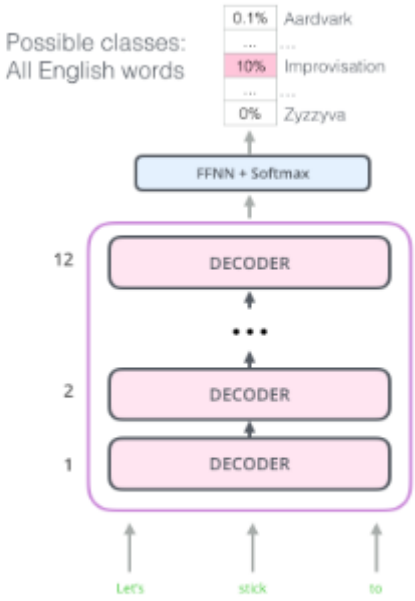
1. **Word2Vec or GloVe: want to use embeddings that are pre-trained on very big data? Use Word2Vec or GloVe.**
2. **ELMo: Contextualized word-embeddings: Instead of using a fixed embedding for each word, ELMo looks at the entire sentence before assigning each word in it and embedding. What's ELMo's secret? ELMo gained its language understanding from being trained to predict the next word in a sequence of words - a task called Language Modeling. This is convenient because we have vast amounts of text data that such a model can learn from without needing labels.**
3. **Directional LSTM: Contextual Word Embeddings use Bi-directional LSTM. ELMo actually goes a step further and trains a bi-directional LSTM – so that its language model doesn't only have a sense of the next word, but also the previous word.**

Some know Transformers as a valid substitute for LSTM; because they deal with long term dependencies much better.

Open AI Transformer: Pre-training a Transformer Decoder for Language Modeling It turns out we don't need an entire Transformer to adopt transfer learning and a fine-tunable language model for NLP tasks. We can do with just the decoder of the transformer. The decoder is a good choice because it's a natural choice for language modeling (predicting the next word) since it's built to mask future tokens – a valuable feature when it's generating a translation word by word. Language Modeling: Predicting the next word (Decoder Transformer Task).



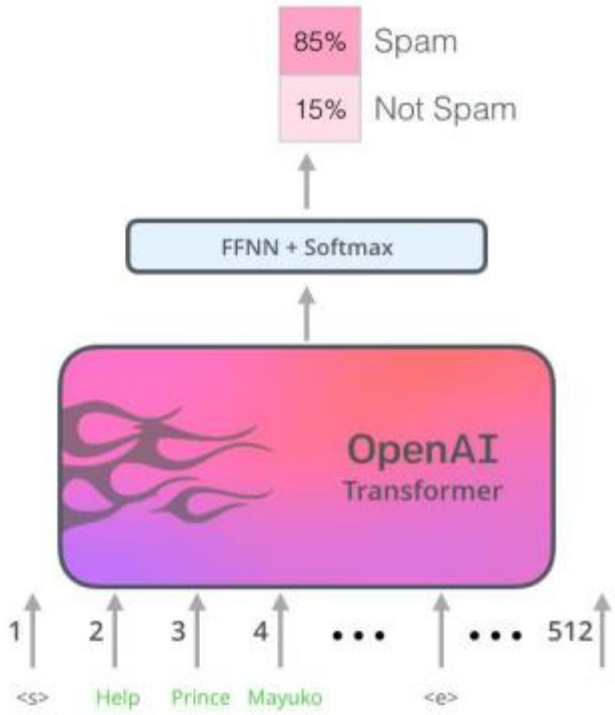
The model stacked twelve decoder layers. Since there is no encoder in this set up, these decoder layers would not have the encoder-decoder attention sublayer that vanilla transformer decoder layers have. It would still have the self-attention layer. With this structure, we can proceed to train the model on the same language modeling task: predict the next word using massive (unlabeled) datasets. Just, throw the text of 7,000 books at it and have it learned! Books are great for this sort of task since it allows the model to learn to associate related information even if they're separated by a lot of text – something you don't get for example, when you're training with tweets, or articles.



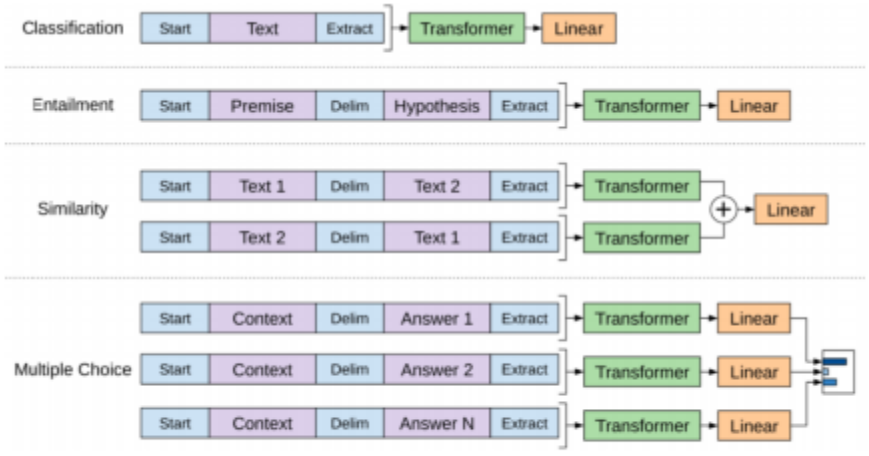
The OpenAI Transformer is now ready to be trained to predict the next word on a dataset made up of 7,000 books.

Transfer Learning to Downstream Tasks

Now that the Open AI transformer is pre-trained and its layers have been tuned to reasonably handle language, we can start using it for downstream tasks. Let's first look at sentence classification (classify an email message as "spam" or "not spam"): Down Stream Tasks are those supervised-learning tasks that utilize a pre-trained model or component.



How to use a pre-trained OpenAI transformer to do sentence clasification

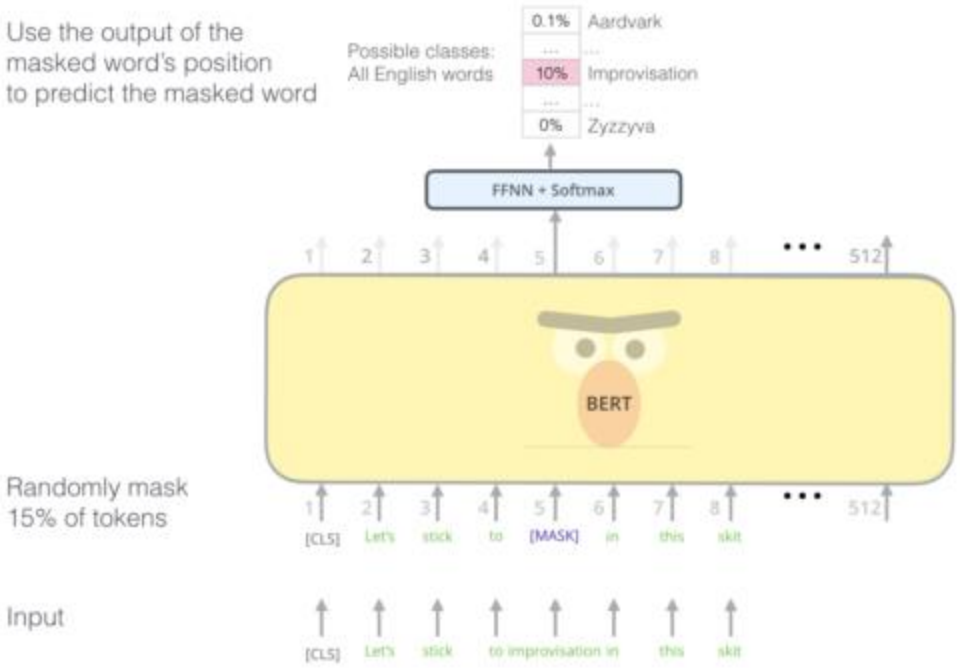


Isn't that clever?

BERT: From Decoders to Encoders

The open AI transformer (The same as Bert in Size only) gave us a fine-tunable pre-trained model based on the Transformer. But something went missing in this transition from LSTMs to Transformers. ELMo's language model was bi-directional, but the open AI transformer only trains a forward language model. LSTM can be bidirectional while Transformers are not bidirectional (they are not conditioned on both left and right context).

"We'll use masks", said BERT confidently.



BERT's clever language modeling task masks 15% of words in the input and asks the model to predict the missing word.

Finding the right task to train a Transformer stack of encoders is a complex hurdle that BERT resolves by adopting a "masked language model" concept from earlier literature (where it's called a Cloze task). Beyond masking 15% of the input, BERT also mixes things a bit in order to improve how the model later fine-tunes. Sometimes it randomly replaces a word with another word and asks the model to predict the correct word in that position.

Viable Bert Application:

1. **Two-sentence Tasks**
2. **Task specific-Models**
3. **feature extraction**
4. **Sentence Classification**
5. **Sentiment analysis**
6. **Fact-checking**

hugging face(Bert library)