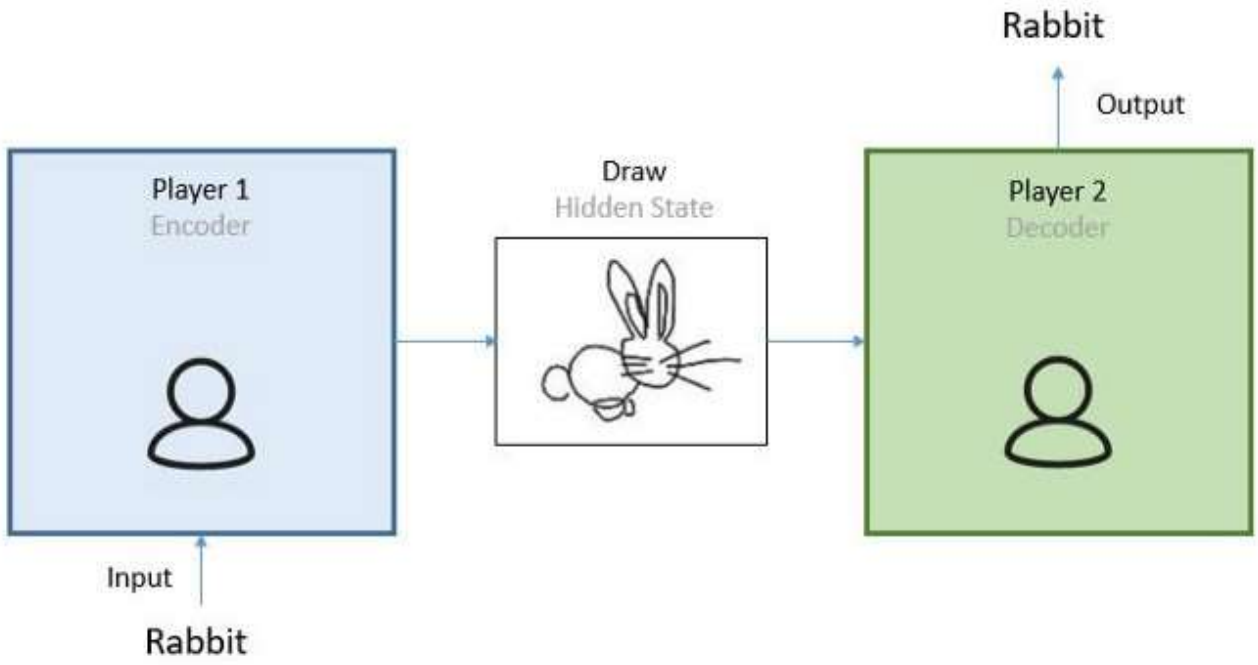
Assignment 9:Transformer

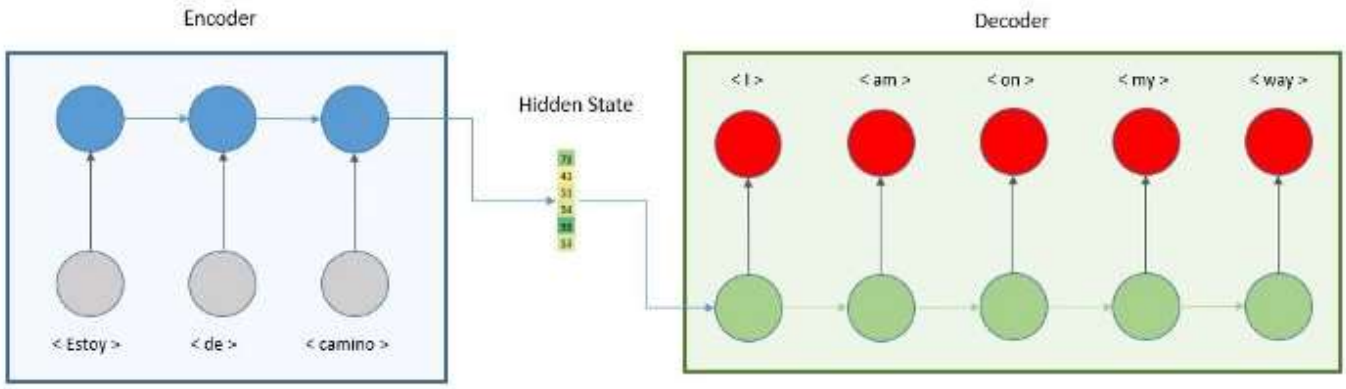
Table of content:

Enoder and Decoder in NLP:

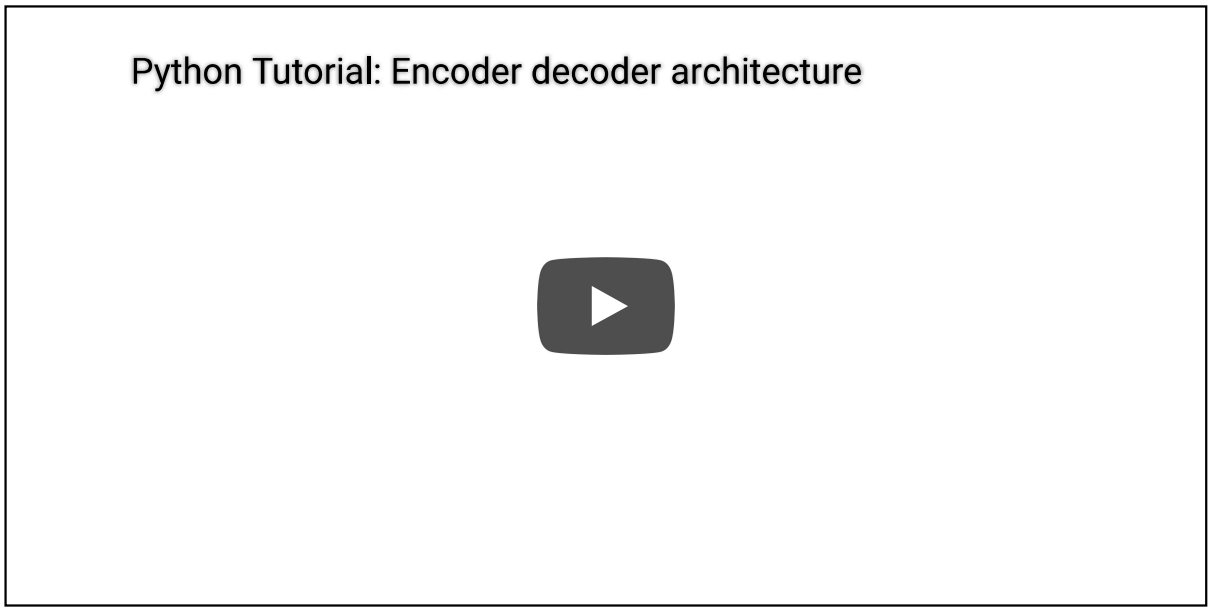
The best way to understand the concept of the encoder-decoder model is by running Pictionary. The rules of the game are very simple, the player chooses 1 word from a list randomly and needs to draw the meaning in the drawing. The role of the second player on the team is to analyze the drawing and select the word it describes. In this example we have three important elements, player 1 (the person who turns the word into a drawing), the drawing (the rabbit) and the person who guesses the word represented by the drawing (player 2). That's all we need to understand the encoder-decoder model, below we'll build a comparison between the game Pictionary and the encoder-decoder model for Spanish to English translation.



We translate the above graph into machine learning concepts, we would see the below one.

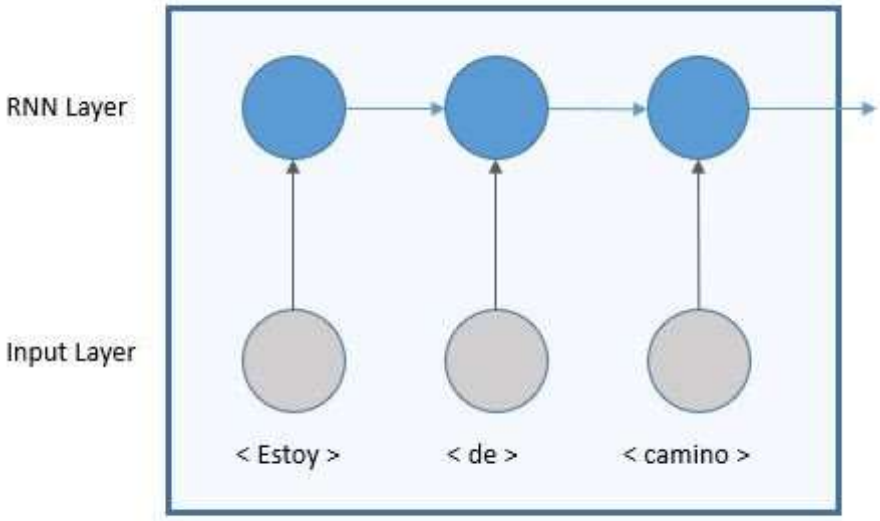



```
###Refer Video
from IPython.display import YouTubeVideo
YouTubeVideo('DDYpp1Pu-zE', width=600, height=300)
```



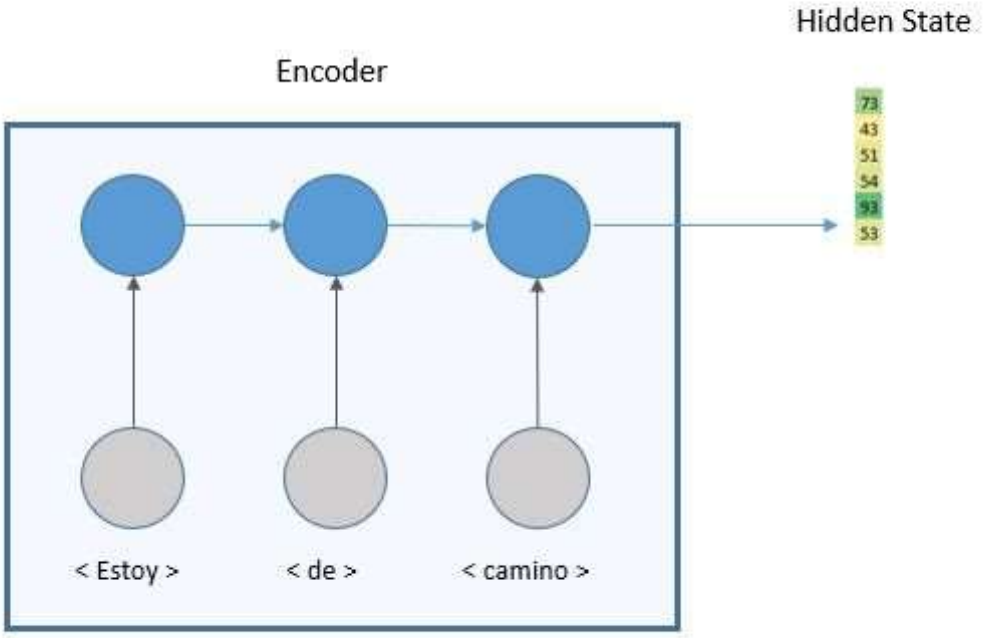
1.Encoder

Encoding means to convert data into a required format. In the Pictionary example we convert a word (text) into a drawing (image). In the machine learning context, we convert a sequence of words in Spanish into a two-dimensional vector, this two-dimensional vector is also known as hidden state. The encoder is built by stacking recurrent neural network. We use this type of layer because its structure allows the model to understand context and temporal dependencies of the sequences. The output of the encoder, the hidden state, is the state of the last RNN timestep.



2.Hidden State

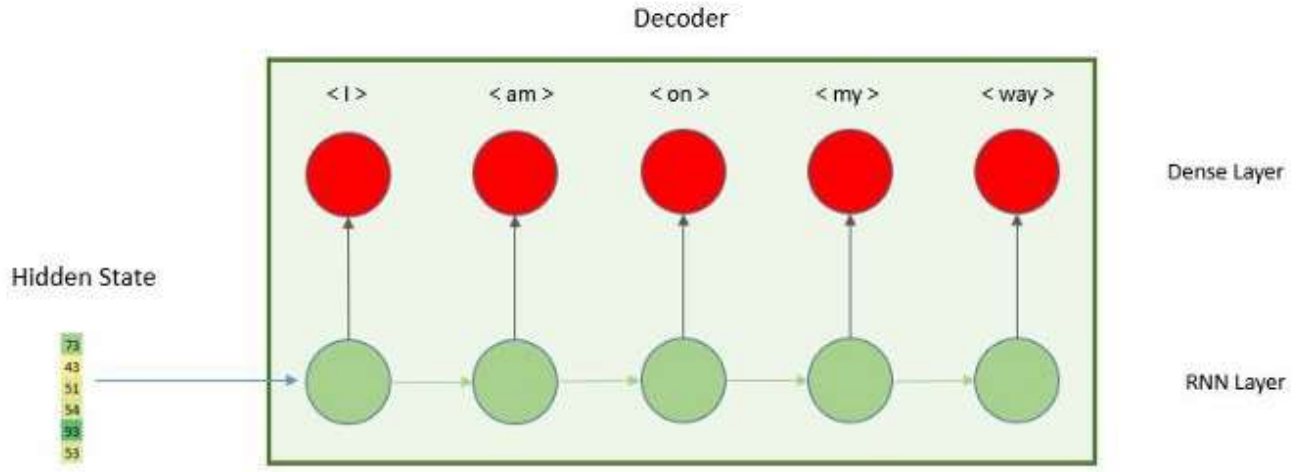
The output of the encoder, a two-dimensional vector that encapsulates the whole meaning of the input sequence. The length of the vector depends on the number of cells in the RNN.



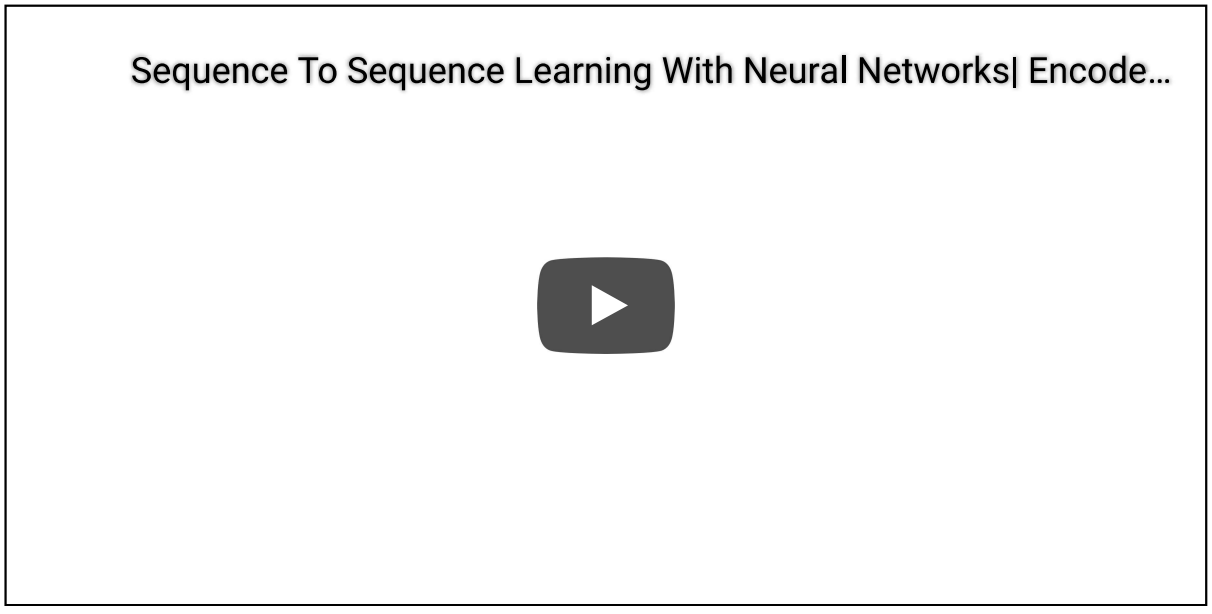
3.Decoder

To decode means to convert a coded message into intelligible language. The second person in the team playing Pictionary will convert the drawing into a word. In the machine learning model, the role of the decoder will be to convert the two-dimensional vector into the output sequence,

the English sentence. It is also built with RNN layers and a dense layer to predict the English word.




```
###Refer Video
YouTubeVideo('jCrgzJlxTKg', width=600, height=300)
```

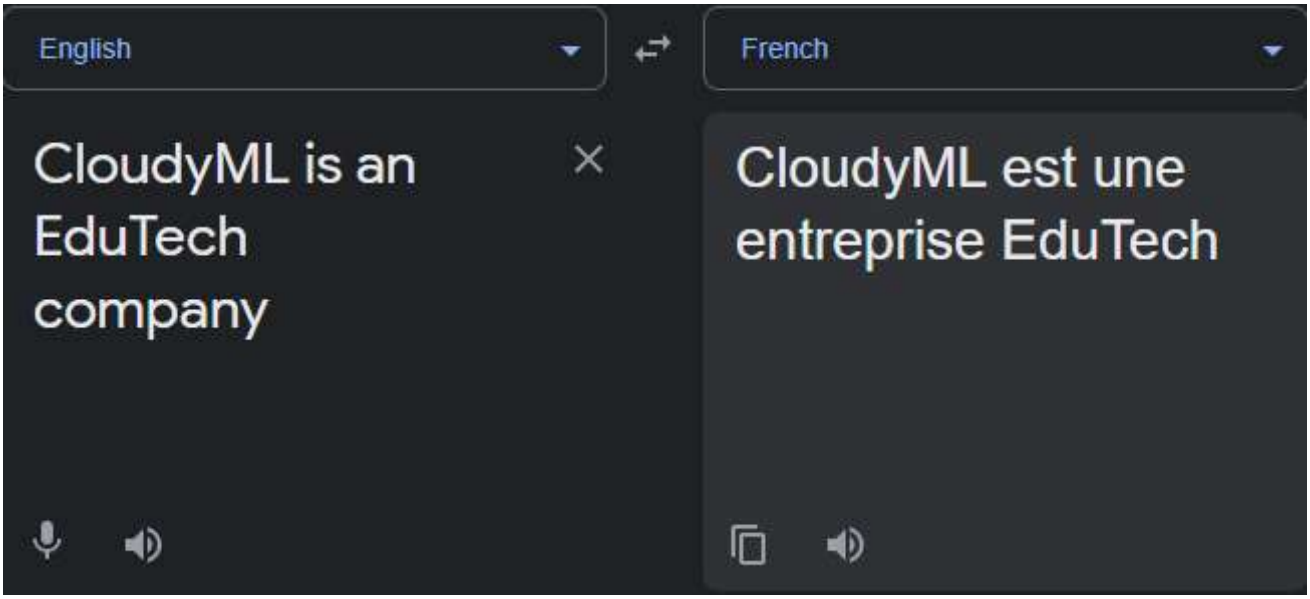


Where we use an encoder decoder model?

1. Image Captioning

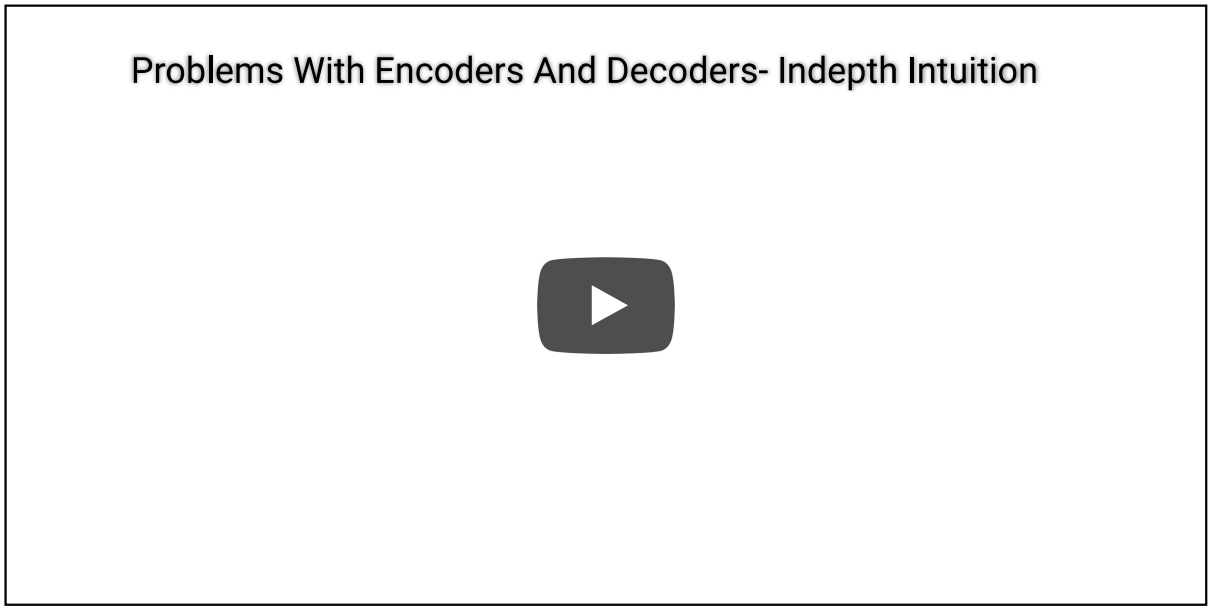
2. Sentiment Analysis

3. Translation



Problems with Encoder and Decoder:

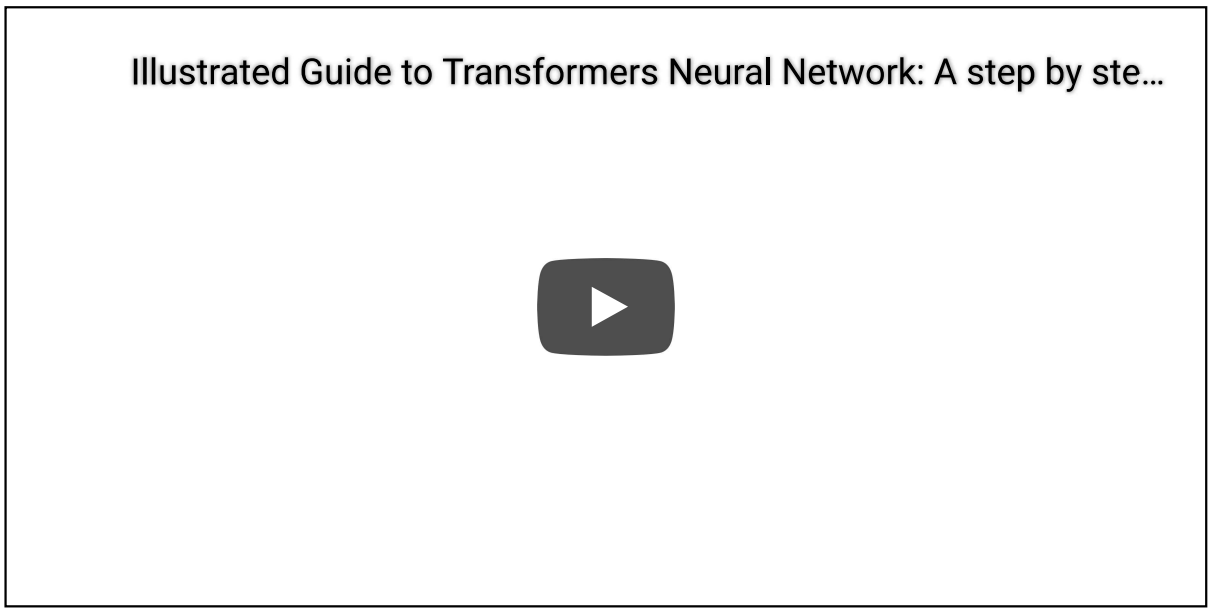
```
###Refer Video
YouTubeVideo('tHf4CmTH1QE', width=600, height=300)
```



The rise of the Transformer: Attention Is All You Need

In December 2017, Vaswani et al. published their seminal paper, Attention Is All You Need. They performed their work at Google Research and Google Brain. I will refer to the model described in Attention Is All You Need as the "original Transformer model". In this section, we will look at the Transformer model they built from the outside. In the following sections, we will explore what is inside each component of the model.

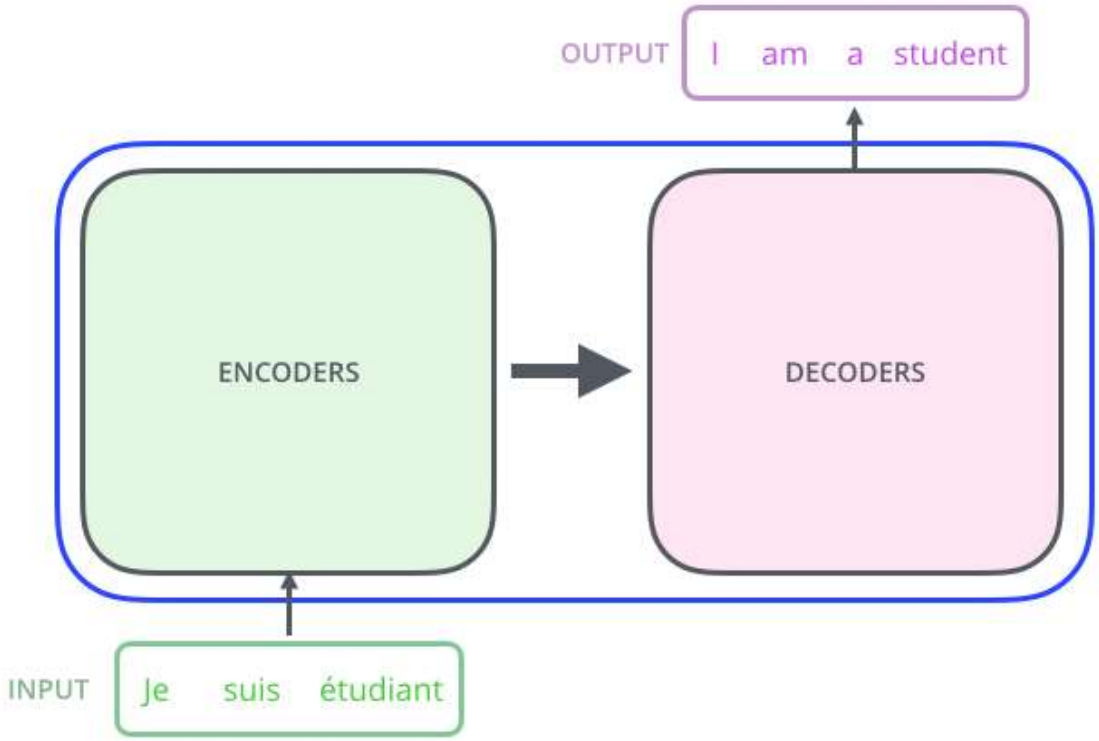
```
###Refer Video
YouTubeVideo('4Bdc55j80l8', width=600, height=300)
```



A Look at Transformer

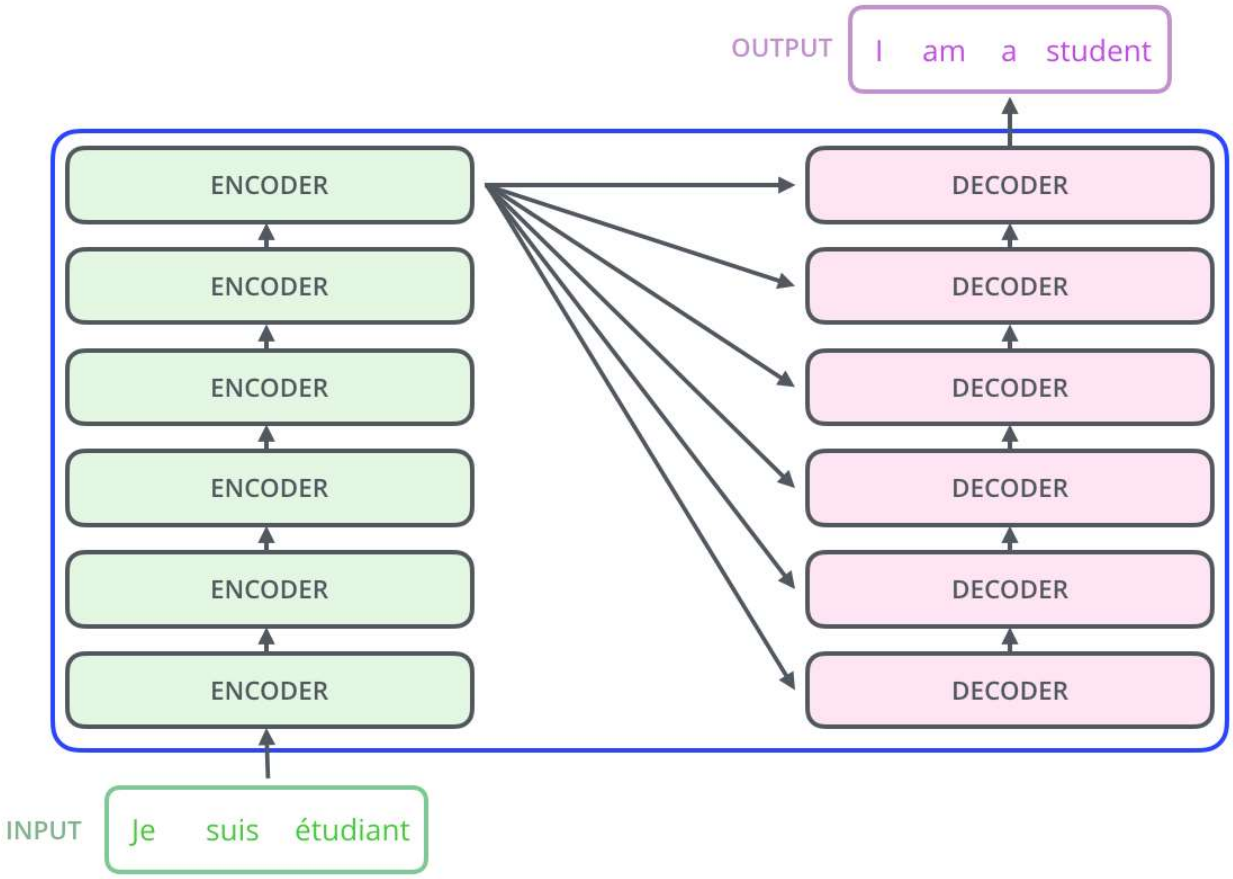
In a translation, it would take a sentence in one language, and output its translation in another.

Futher dividing into Encoder and Decoder



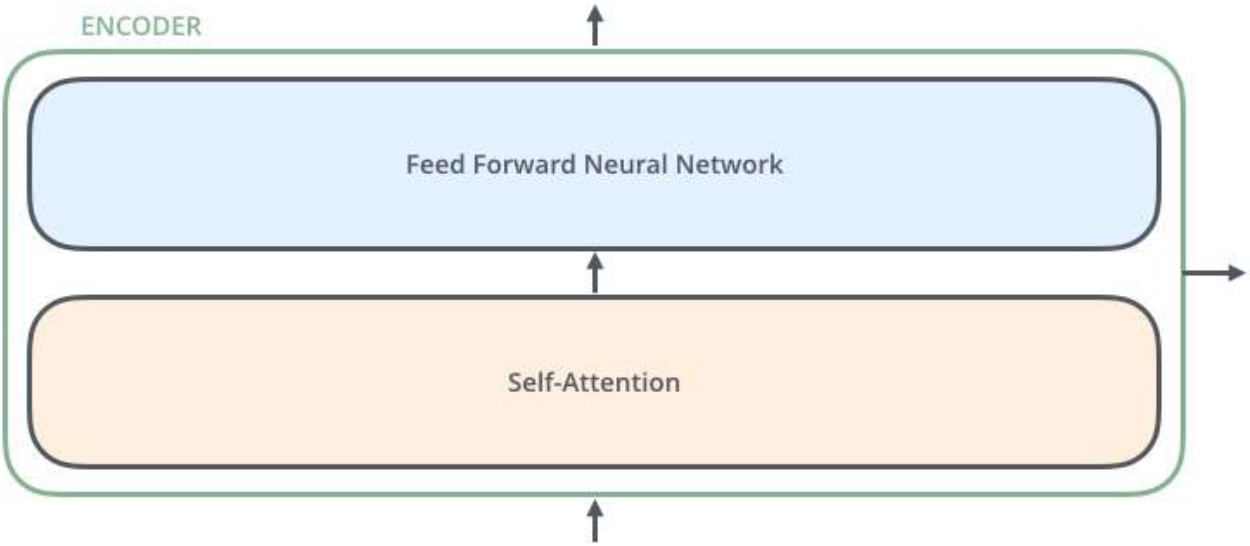
The original Transformer model is a stack of 6 layers.

The encoding component is a stack of encoders.The decoding component is a stack of decoders of the same number.



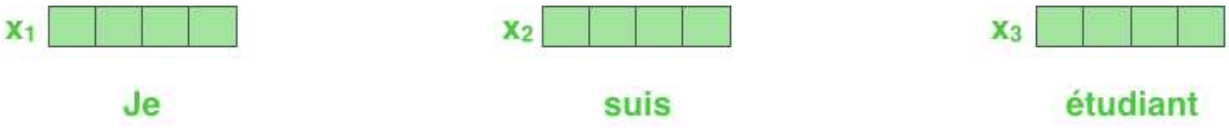
Encoder

The original encoder layer structure remains the same for all of the N=6 layers of the Transformer model. Each layer contains two main sub-layers: a multi-headed attention mechanism and a fully connected position-wise feedforward network.



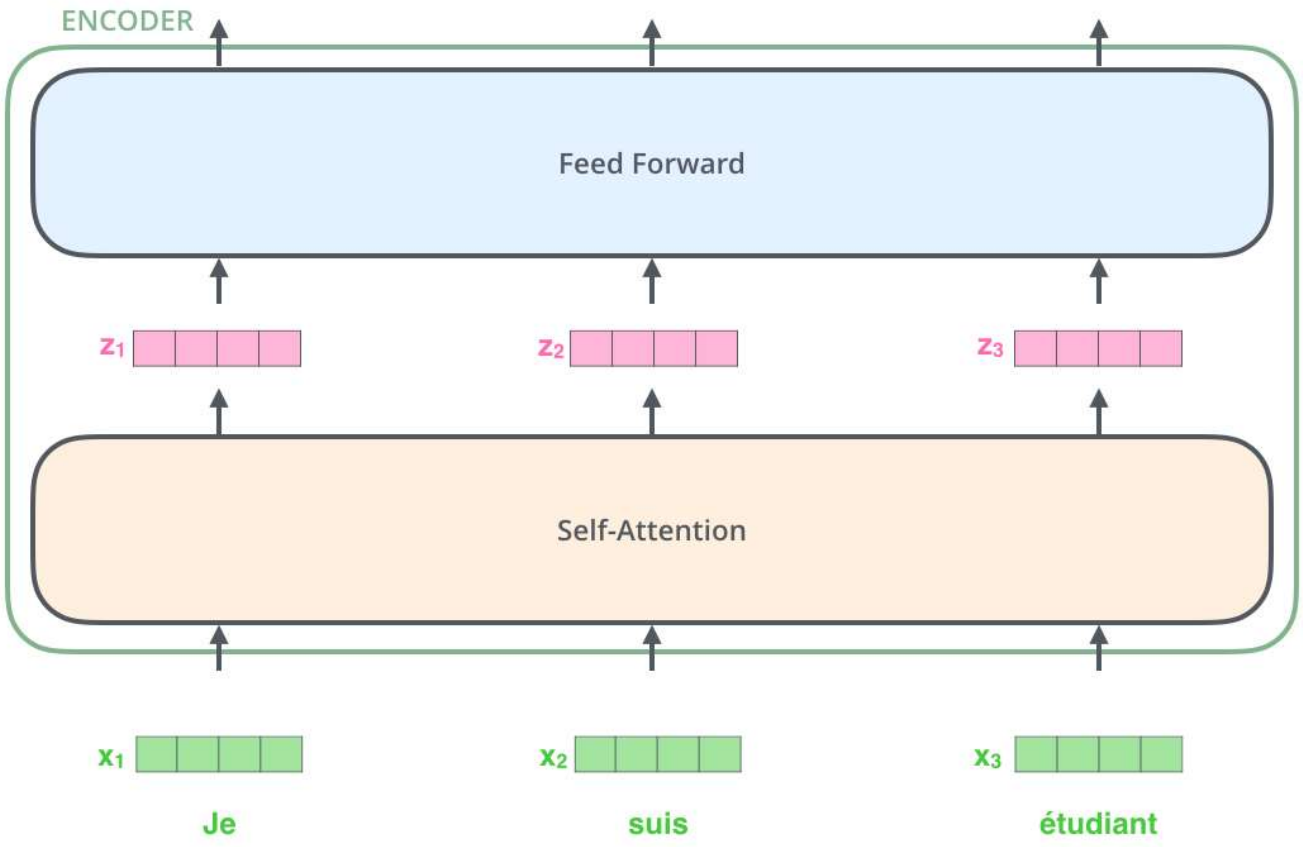
The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

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



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

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.



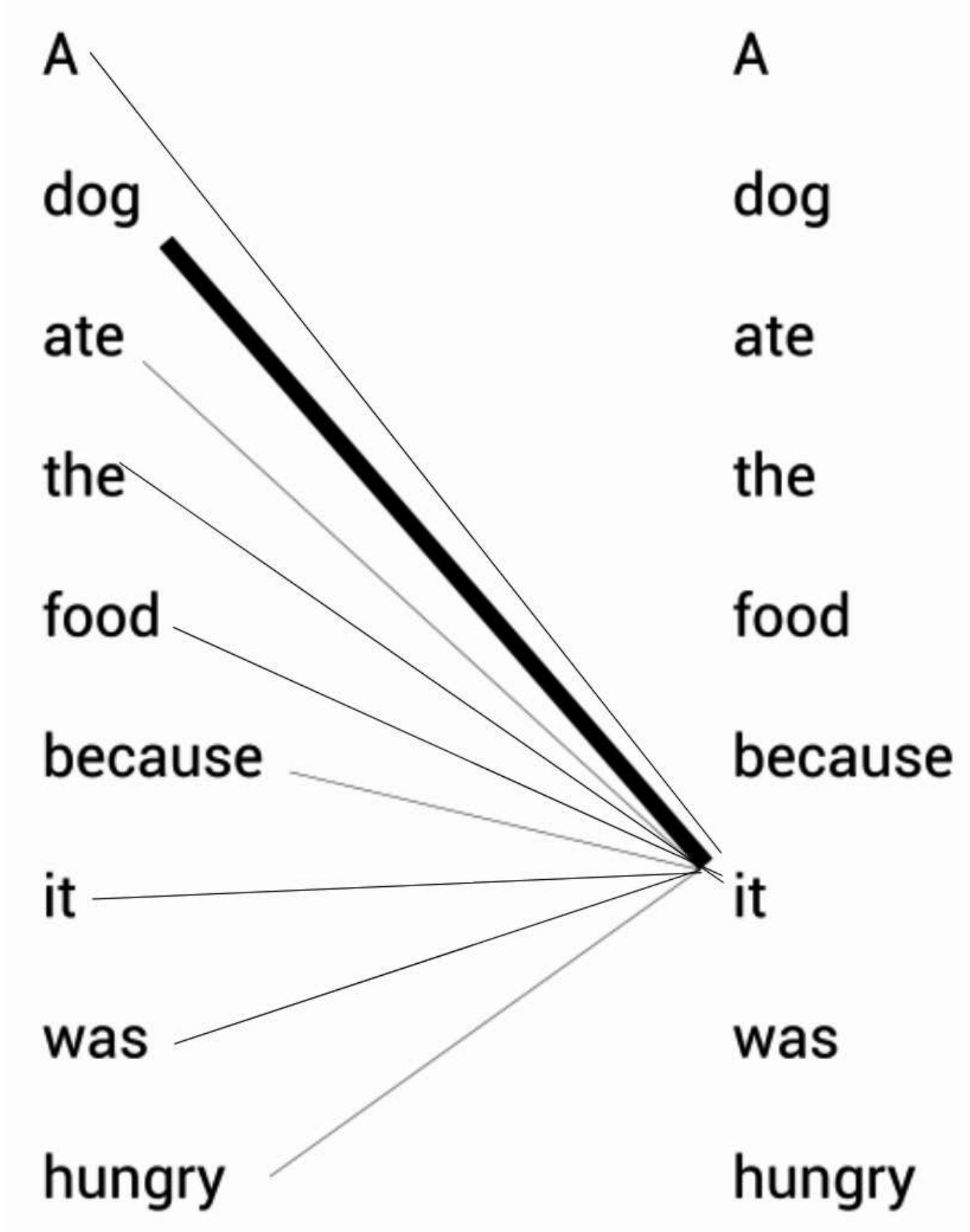
After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.

Self-Attention Mechanism

Let's understand the self-attention mechanism with an example. Consider the following sentence:

```
A dog ate the food because it was hungry
```

In the preceding sentence, the pronoun it could mean either dog or food. By reading thesentence, we can easily understand that the pronoun it implies the dog and not food. But how does our model understand that in the given sentence, the pronoun it implies the dog and not food? Here is where the self-attention mechanism helps us.
In the given sentence, A dog ate the food because it was hungry, first, our model computes the representation of the word A, next it computes the representation of the word dog, then it computes the representation of the word ate, and so on. While computing the representation of each word, it relates each word to all other words in the sentence to understand more about the word.

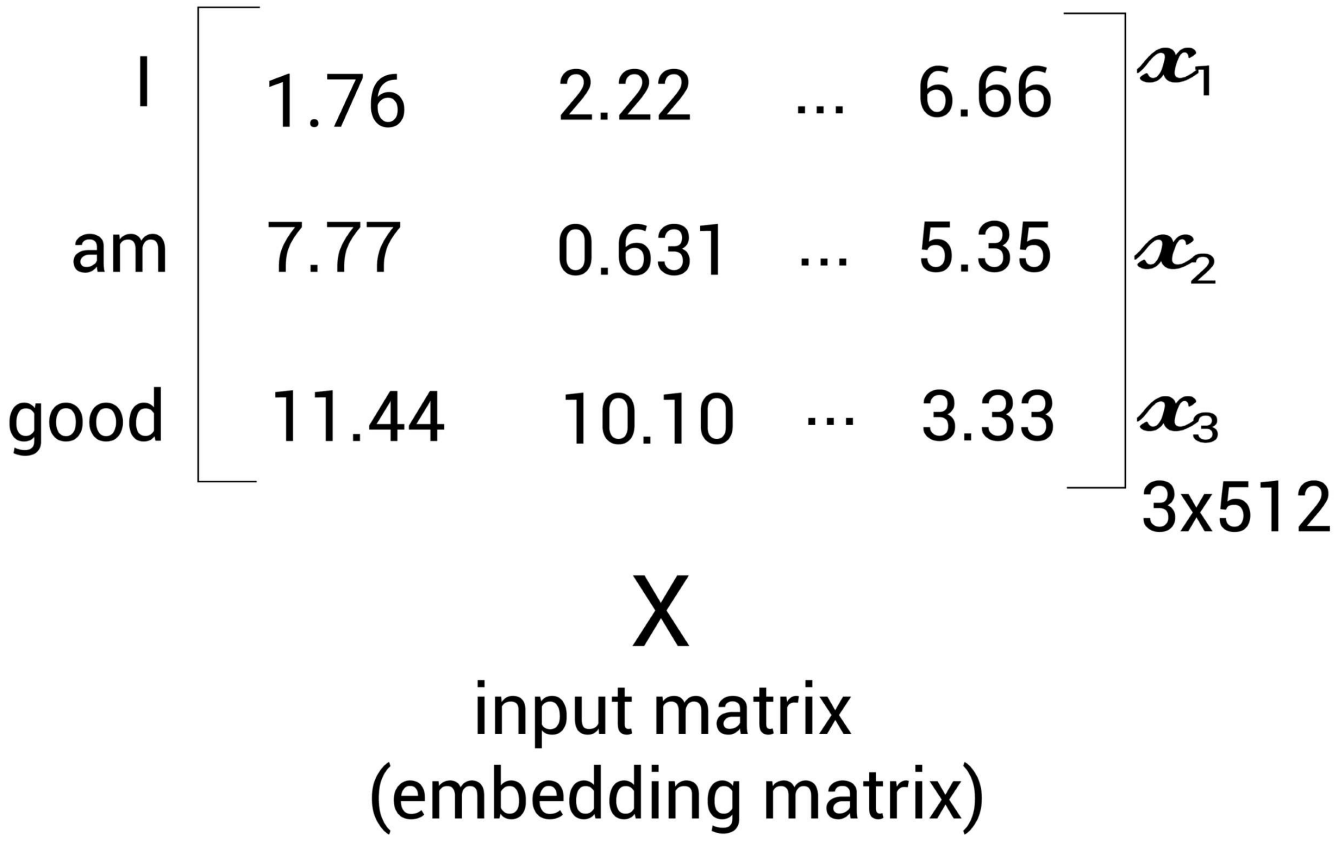


As shown in the following figure, in order to compute the representation of the word it, our model relates the word it to all the words in the sentence. By relating the word it to all the words in the sentence, our model can understand that the word it is related to the word dog and not food. As we can observe, the line connecting the word it to dog is thicker compared to the other lines,

which indicates that the word it is related to the word dog and not food in the given sentence

Okay, but how exactly does this work? Now that we have a basic idea of what the selfattention mechanism is, let's understand more about it in detail.

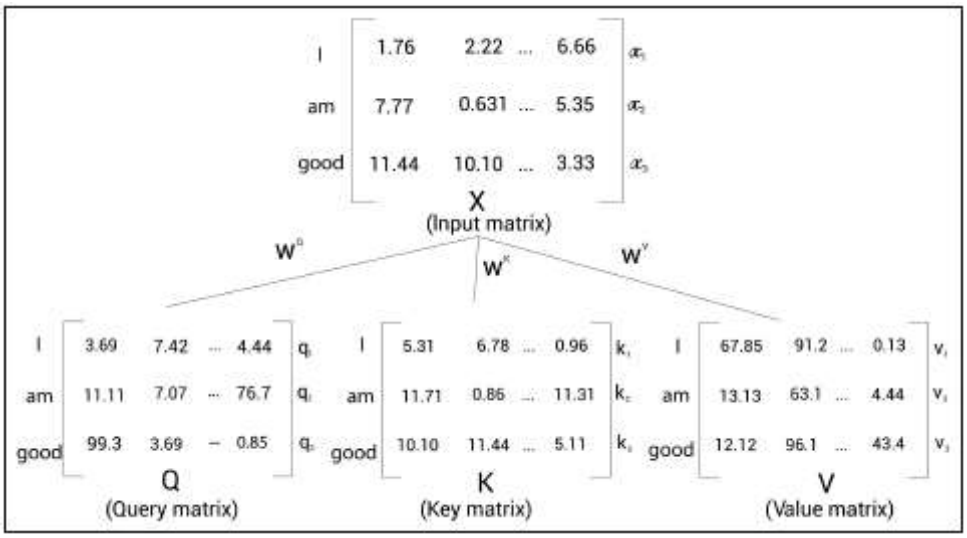
Suppose our input sentence (source sentence) is I am good. First, we get the embeddings for each word in our sentence. Note that the embeddings are just the vector representation of the word and the values of the embeddings will be learned during training. we can represent our input sentence I am good using the input matrix (embedding matrix or input embedding) as shown here:



Thus, the dimension of the input matrix will be [sentence length x embedding dimension].
The number of words in our sentence (sentence length) is 3. Let the embedding dimension be 512; then, our input matrix(input embedding) dimension will be [3 x 512].

Now, from the input matrix, , we create three new matrices: a query matrix, , key matrix, , and value matrix, . Wait. What are these three new matrices? And why do we need them? They are used in the self-attention mechanism. We will see how exactly these three matrices are used in a while.
Okay, how we can create the query, key, and value matrices? To create these, we introduce three new weight matrices, called WQ,Wk and Wv. We create the query Q,key K and value V matrices by multiplying the input matrix X by WQ, WK,Wv respectively. Note that the weight matrices WQ,

WK,Wv are randomly initialized and their optimal values will be learned during training. As we learn the optimal weights, we will obtain more accurate query, key, and value matrices.



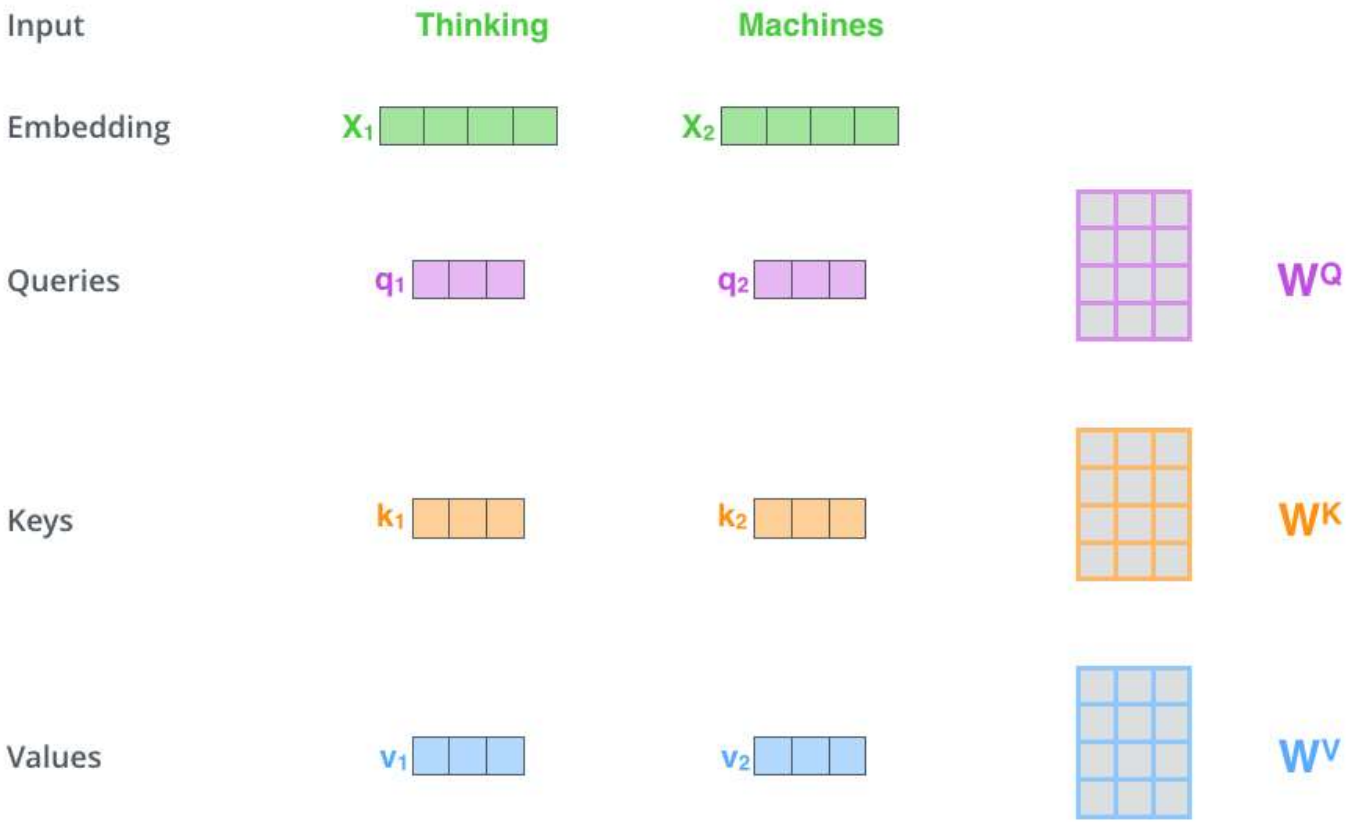
From the preceding figure, we can understand the following:

1. The first row in the query, key, and value matrices q1,k1 and v1 implies the query, key, and value vectors of the word I.
2. The second row in the query, key, and value matrices q2,k2 and v2 implies the query, key, and value vectors of the word am.
3. The third row in the query, key, and value matrices q3,k3 and v3 implies the query, key, and value vectors of the word good.

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. 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.

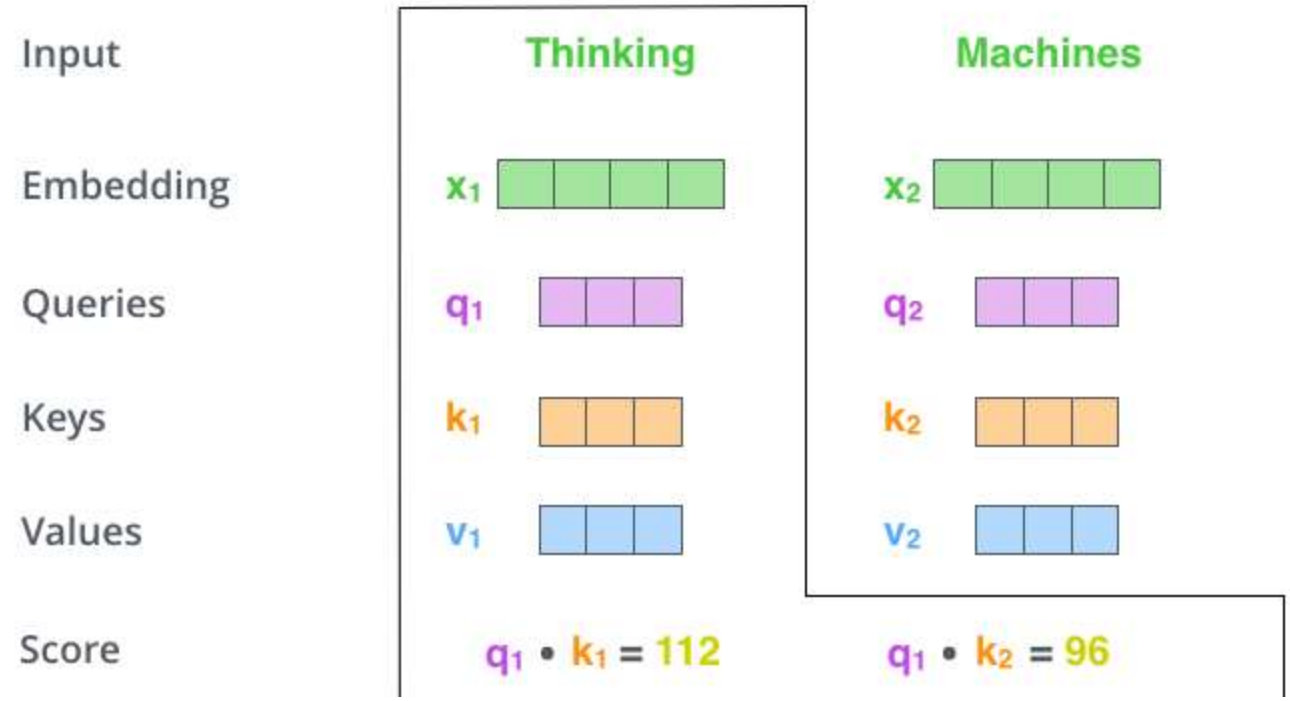
Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512.



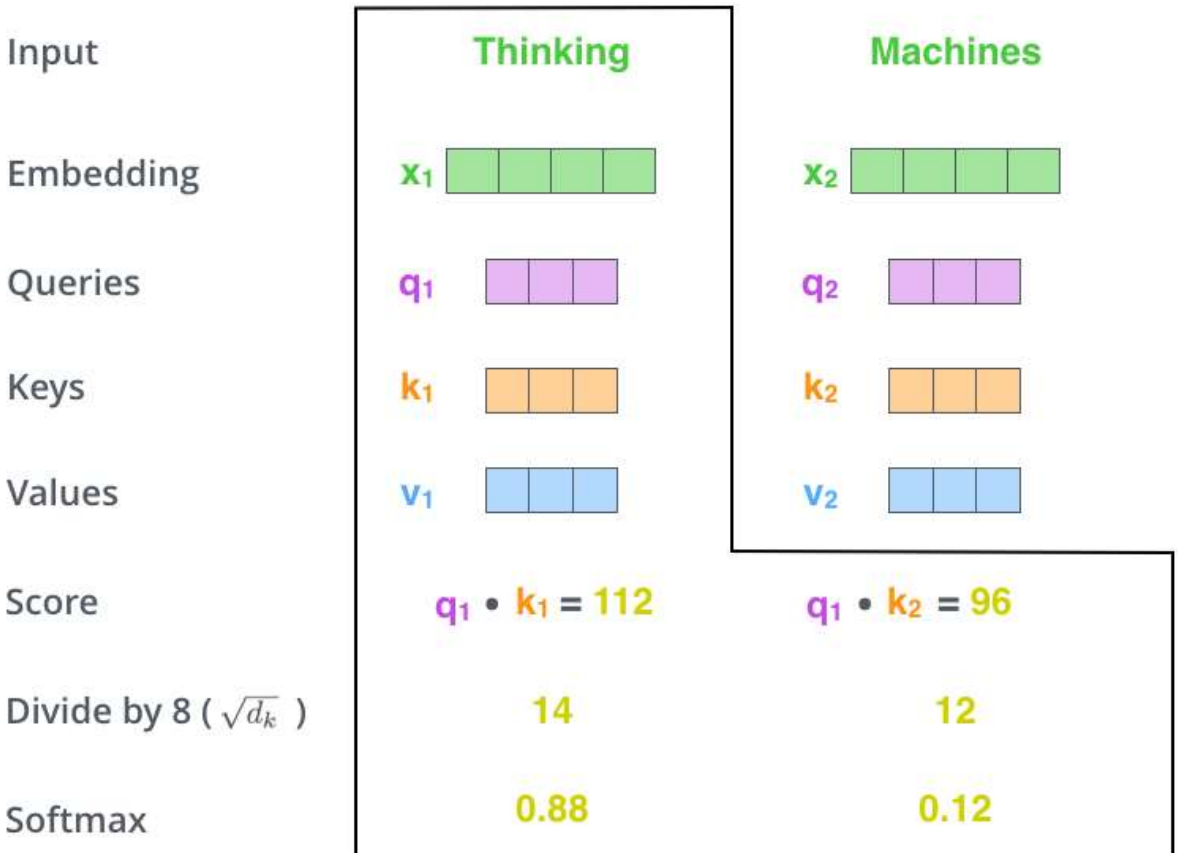
Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

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

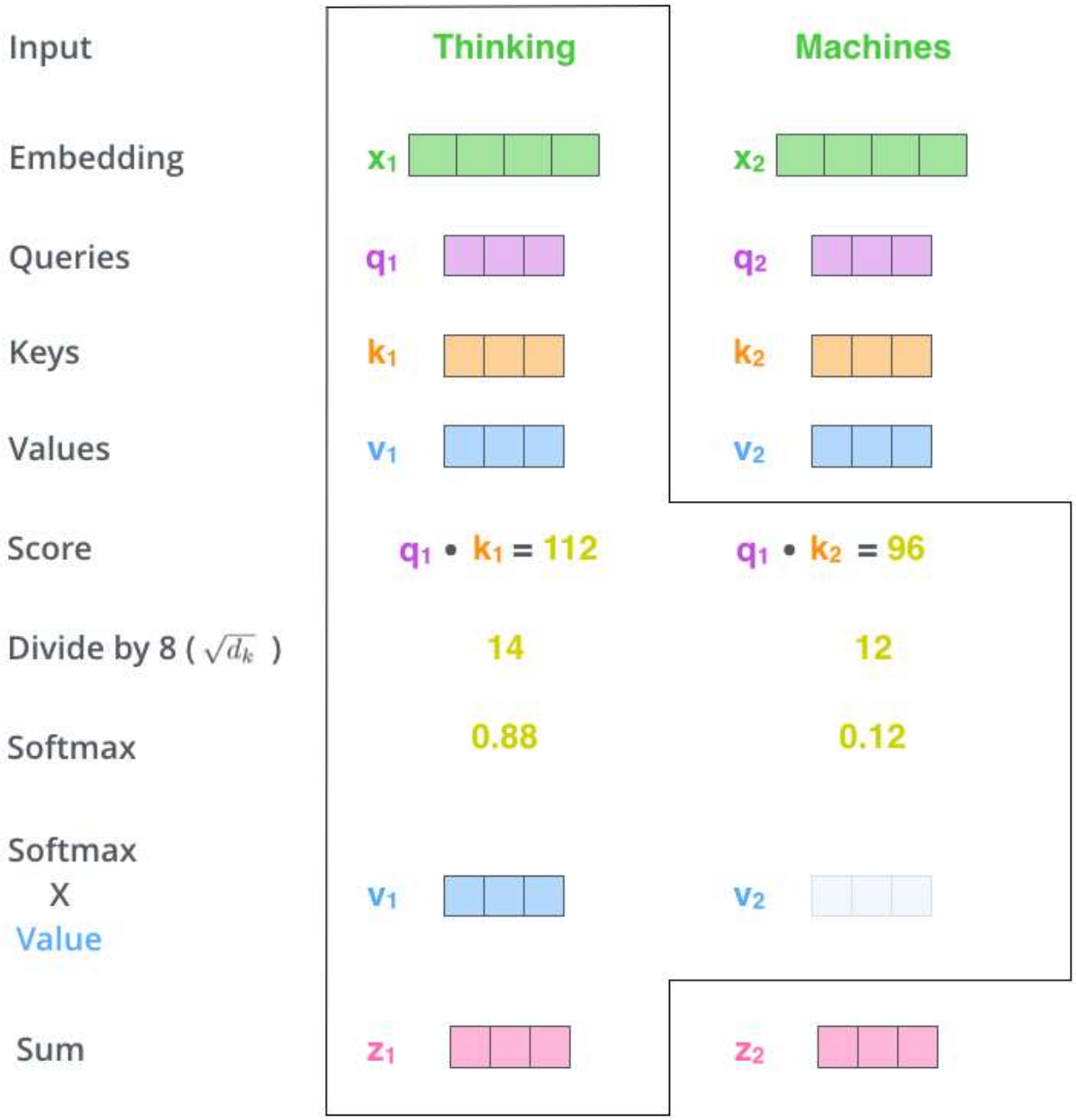
The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring. So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1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



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



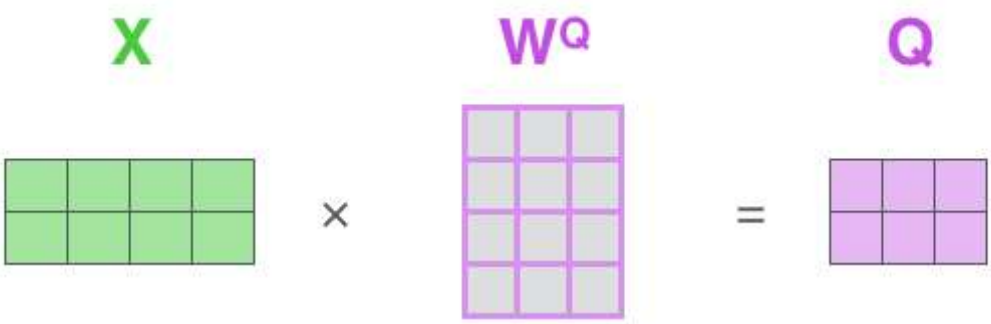
The **fifth** step is to multiply each value vector by the softmax score. The intuition here is to keep intact the values of the word we want to focus on, and drown-out irrelevant words. The **sixth** step is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).



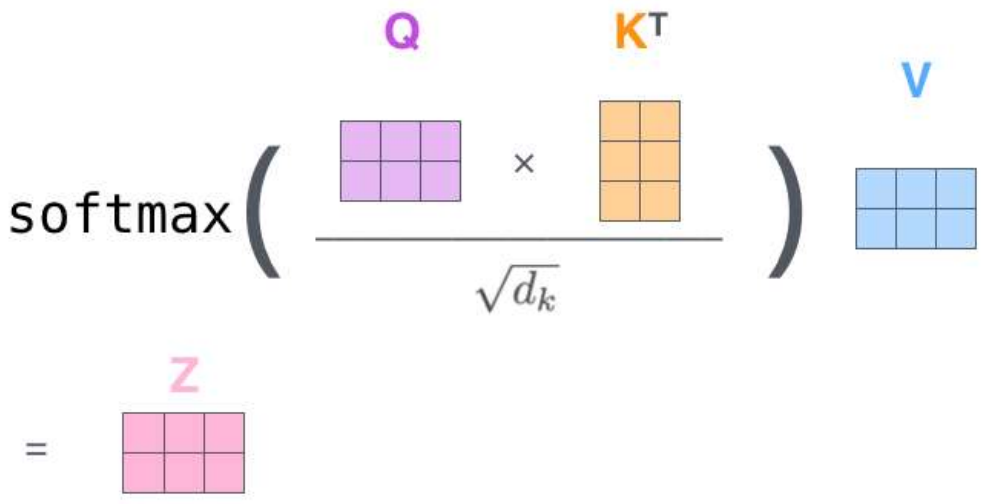
That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is done in matrix form for faster processing. So let's look at that now that we've seen the intuition of the calculation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

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).



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.

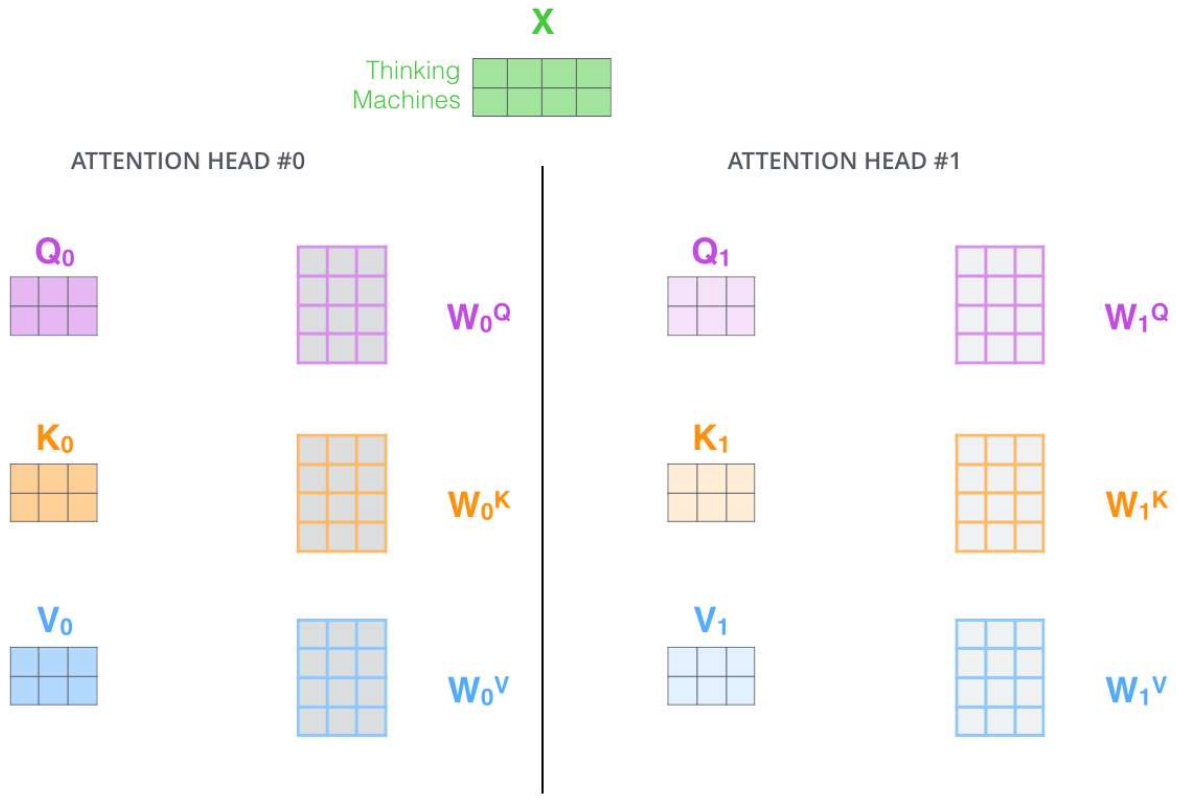


```
###Refer Video
YouTubeVideo('phOc25QfNS0', width=600, height=300)
```

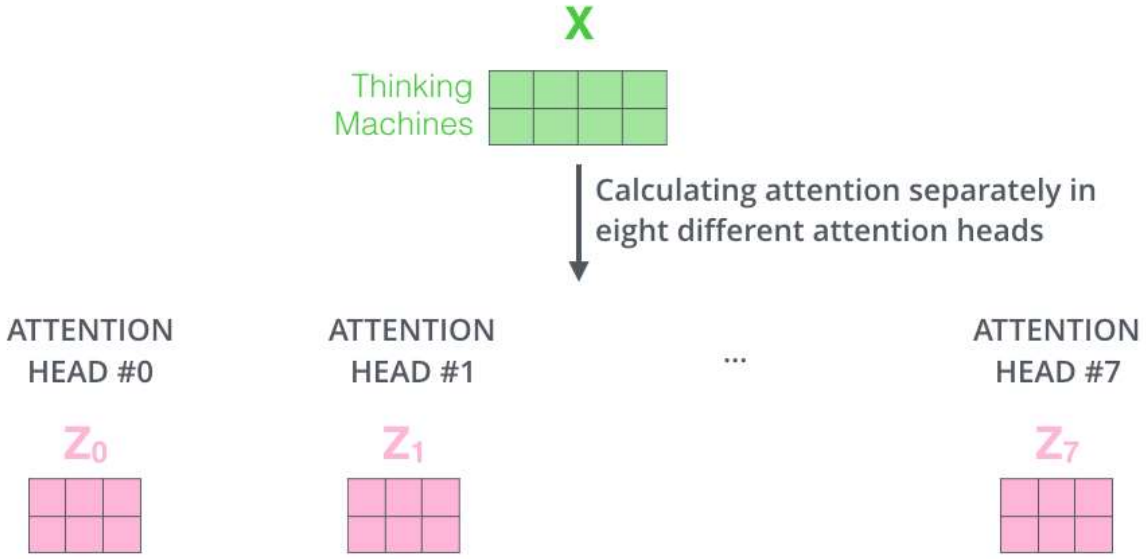
Multi-head attention

his improves the performance of the attention layer in two ways:

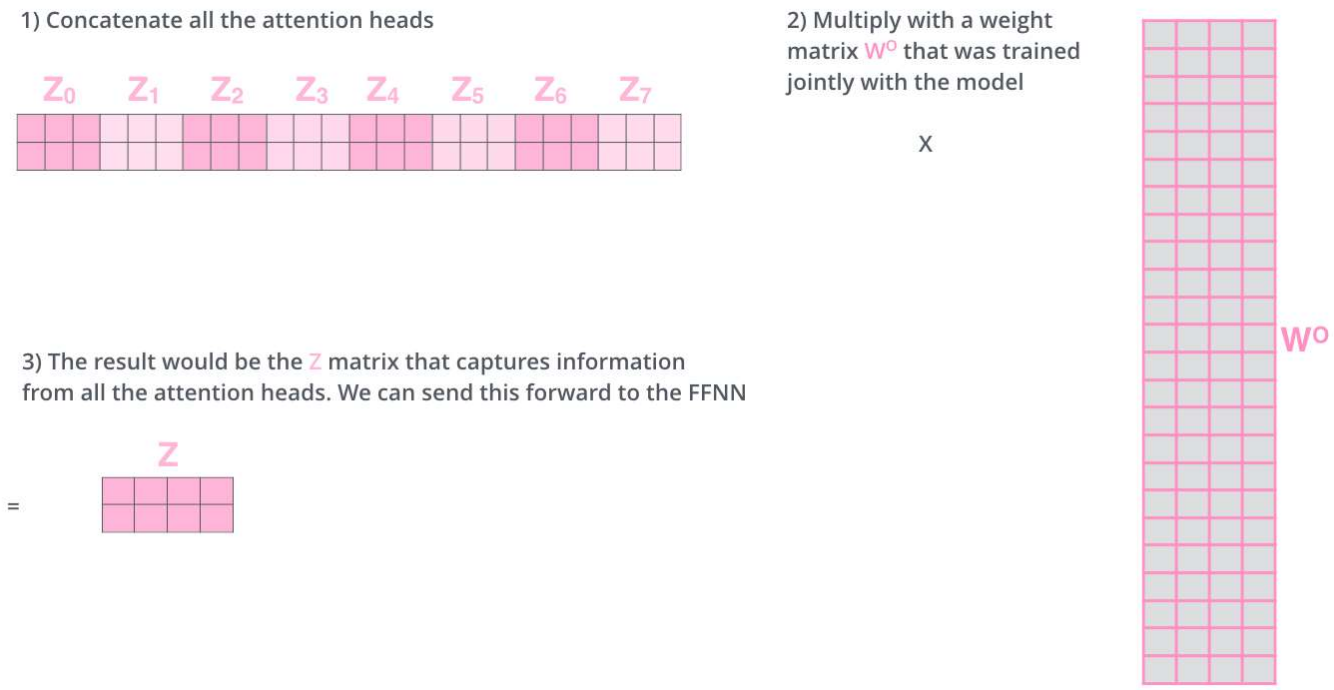
1. It expands the model's ability to focus on different positions. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the the actual word itself. It would be useful if we're translating a sentence like "The animal didn't cross the street because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



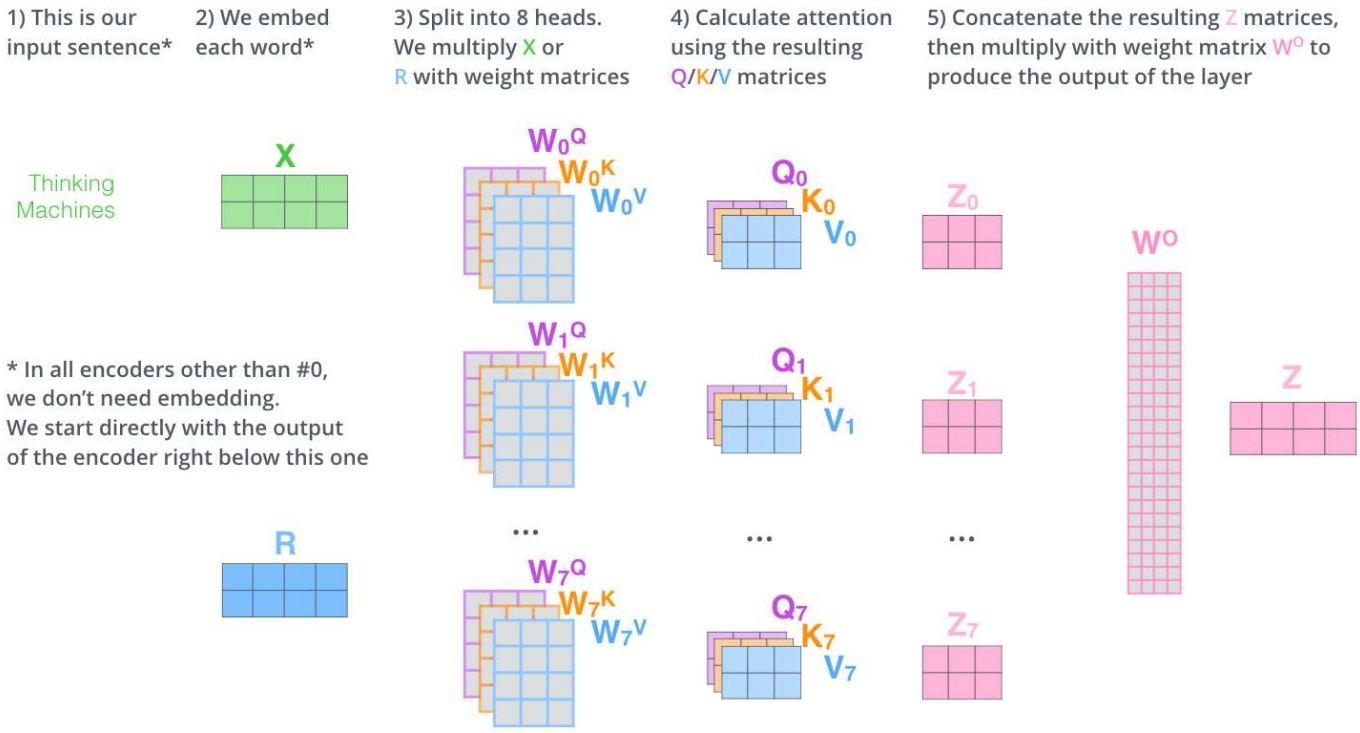
If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices.



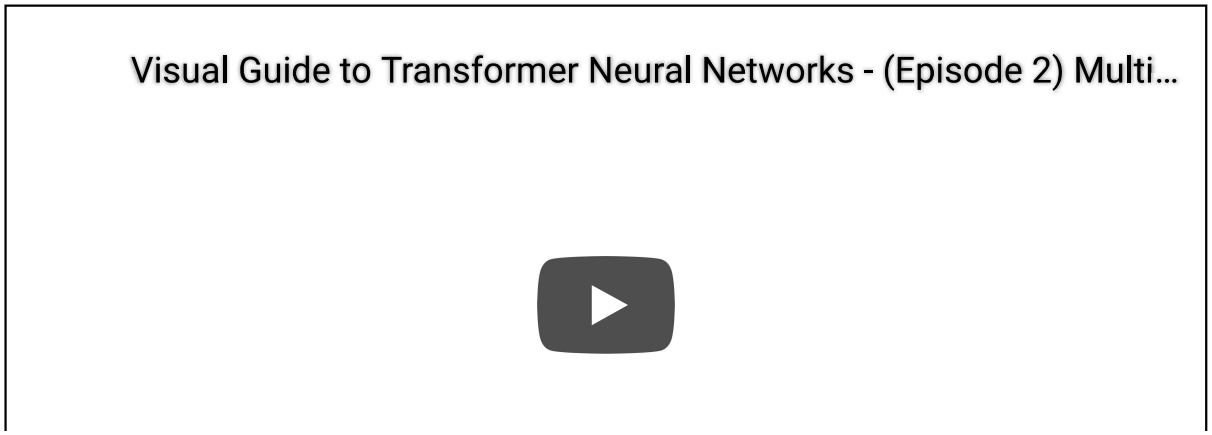
This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.



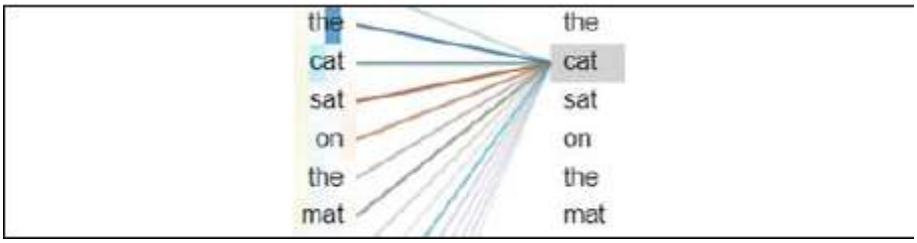
It's quite a handful of matrices, I realize. Let me try to put them all in one visual so we can look at them in one place.




```
###Refer Video
YouTubeVideo('mMa2PmYJlCo', width=600, height=300)
```

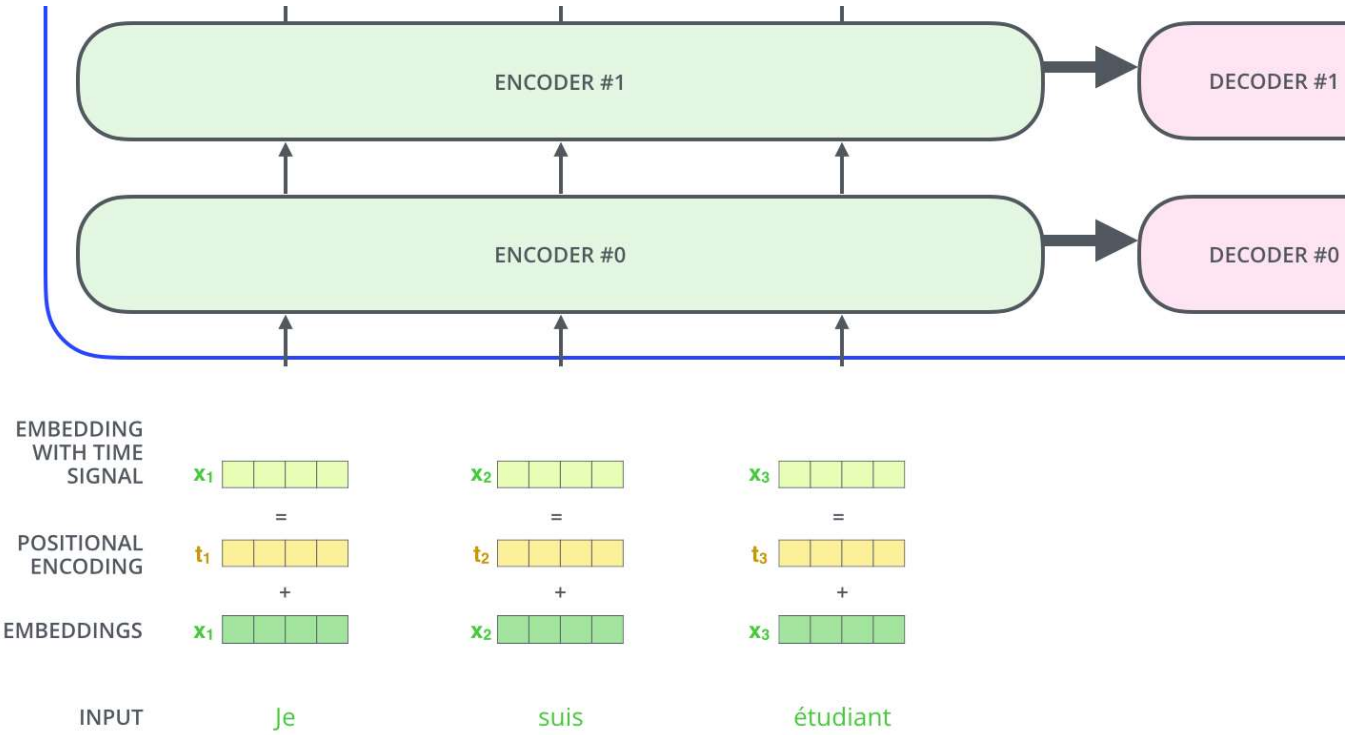


Attention will run dot products between word vectors and determine the strongest relationships of a word among all the other words, including itself ("cat" and "cat").
The attention mechanism will provide a deeper relationship between words and produce better results.
For each attention sub-layer, the original Transformer model runs not one but eight attention mechanisms in parallel to speed up the calculations.



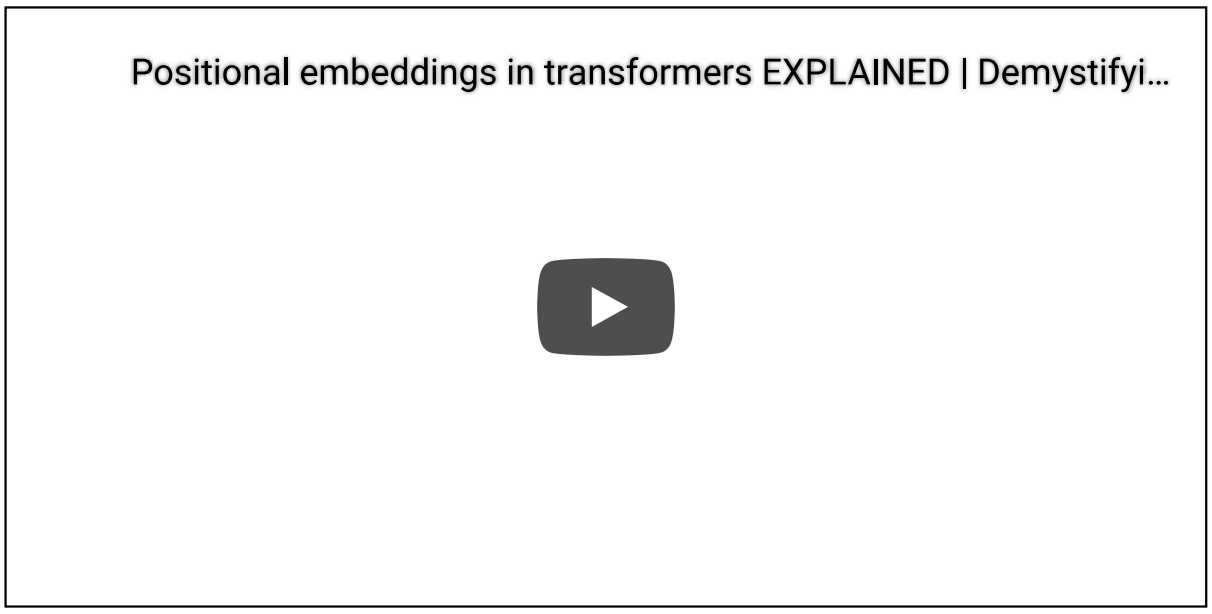
Positional encoding

We enter this positional encoding function of the Transformer with no idea of the position of a word in a sequence:



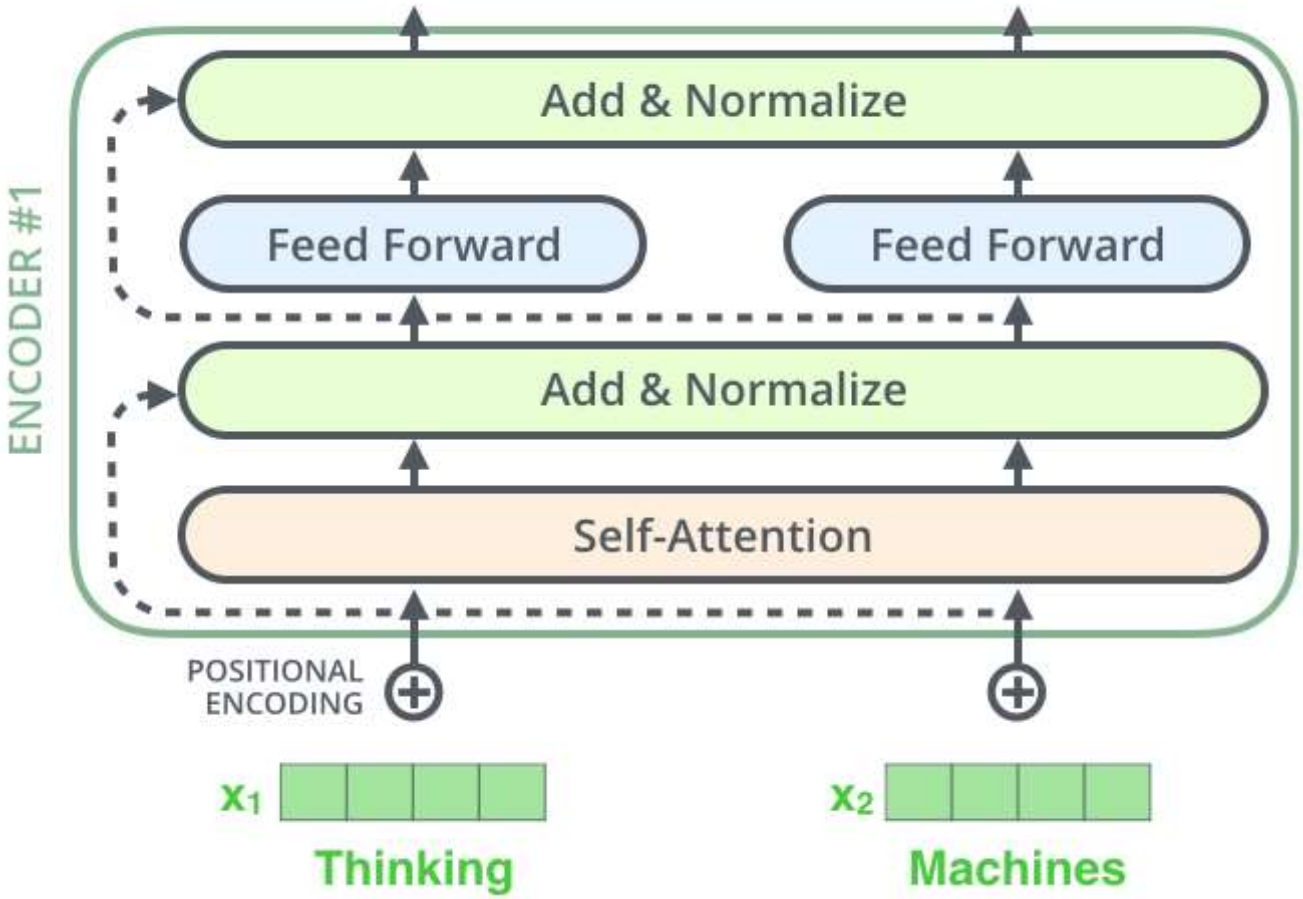
The transformer adds a vector to each input embedding. These vectors follow a specific pattern

```
###Refer Video
YouTubeVideo('1biZfFLPRSY', width=600, height=300)
```

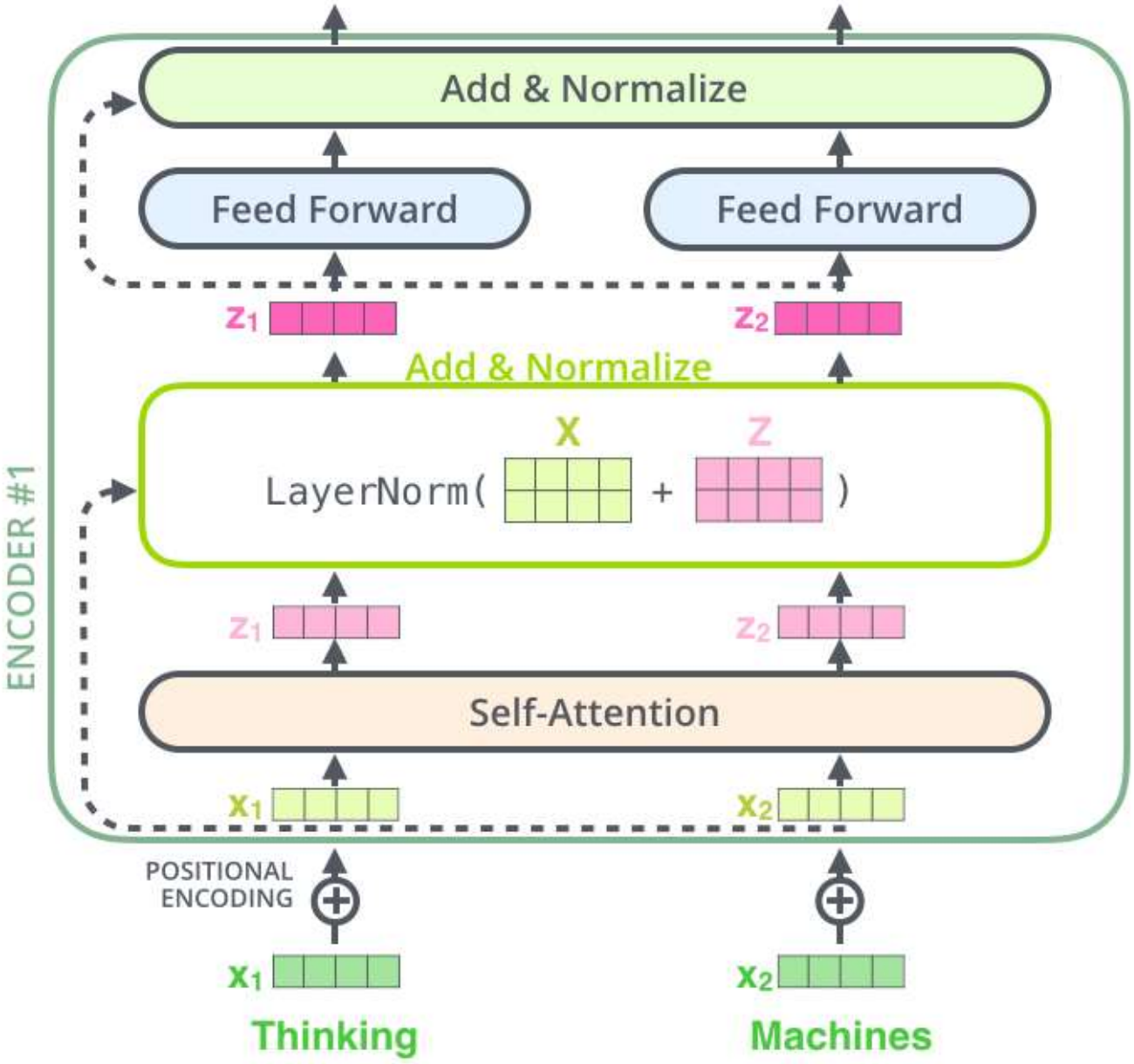


Residuals

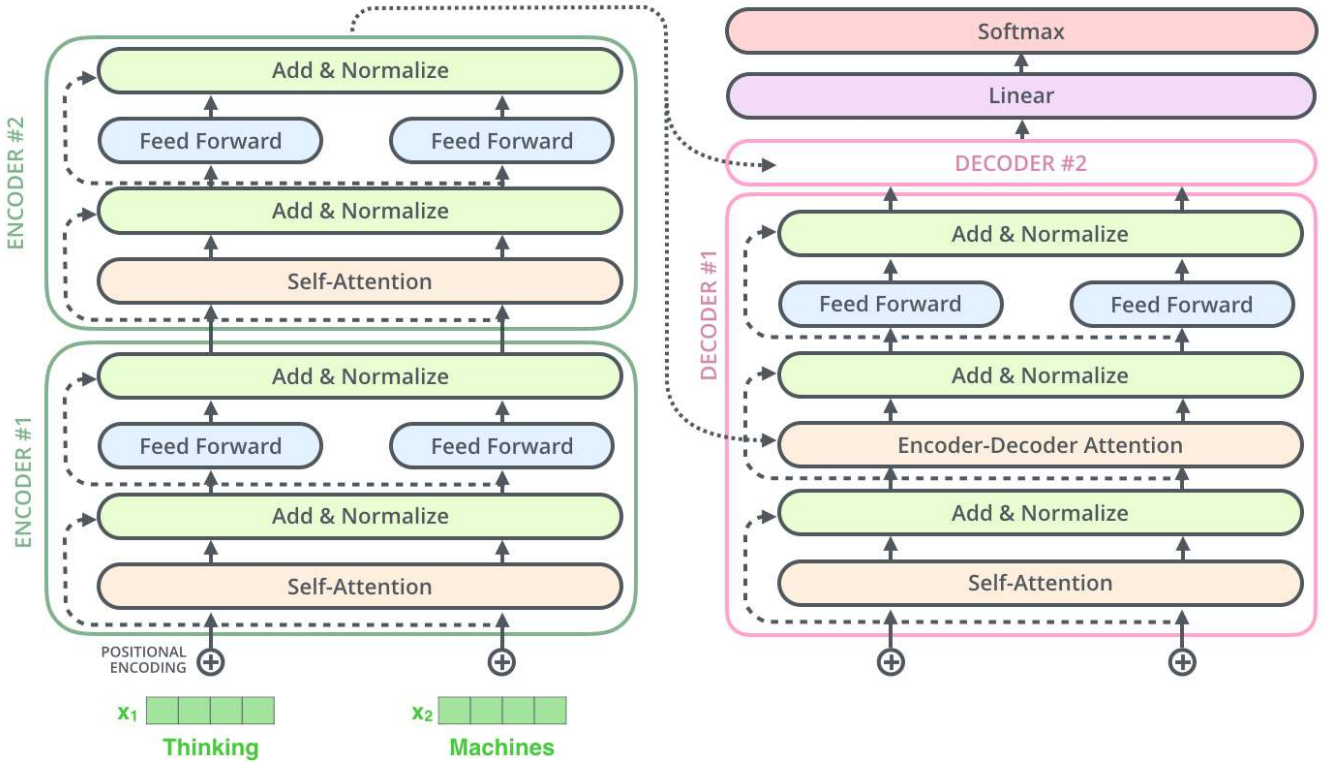
One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization step.



If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:



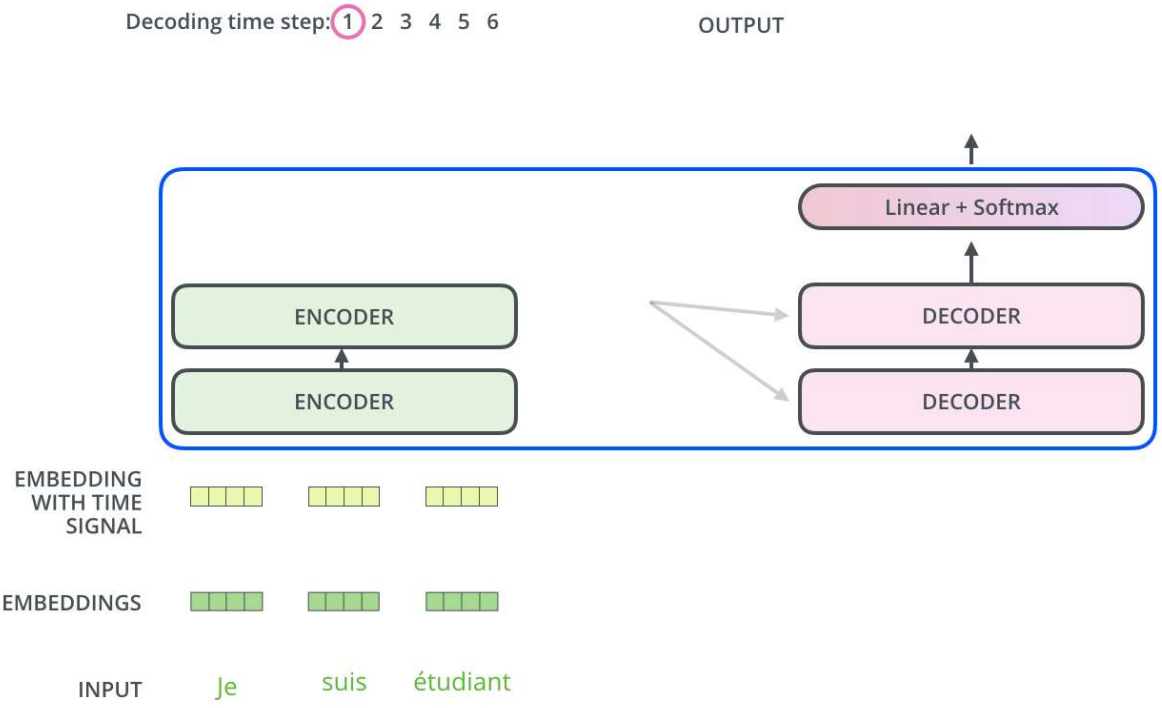
The sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:



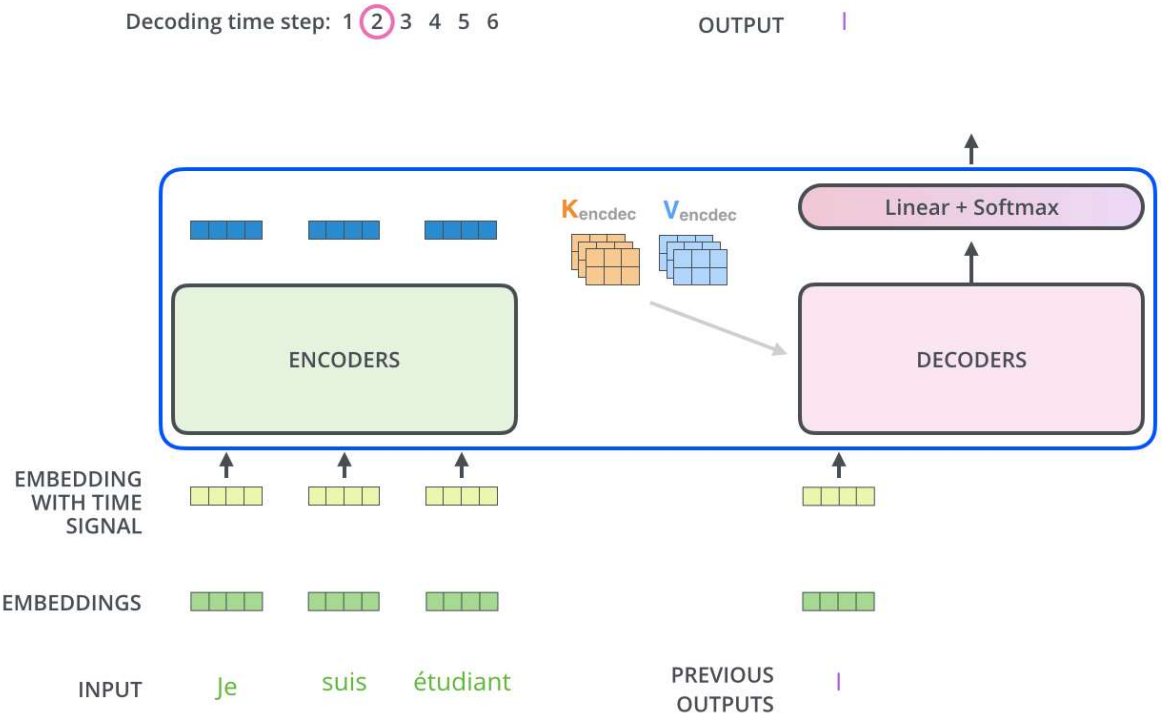
Decoder Side:

The structure of the decoder layer remains the same as the encoder for all the N=6 layers of the Transformer model. Each layer contains three sub-layers: a multiheaded masked attention mechanism, a multi-headed attention mechanism, and a fully connected position-wise feedforward network.
The decoder has a third main sub-layer, which is the masked multi-head attention mechanism. In this sub-layer output, at a given position, the following words are masked so that the Transformer bases its assumptions on its inferences without seeing the rest of the sequence. That way, in this model, it cannot see future parts of the sequence.

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:



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.



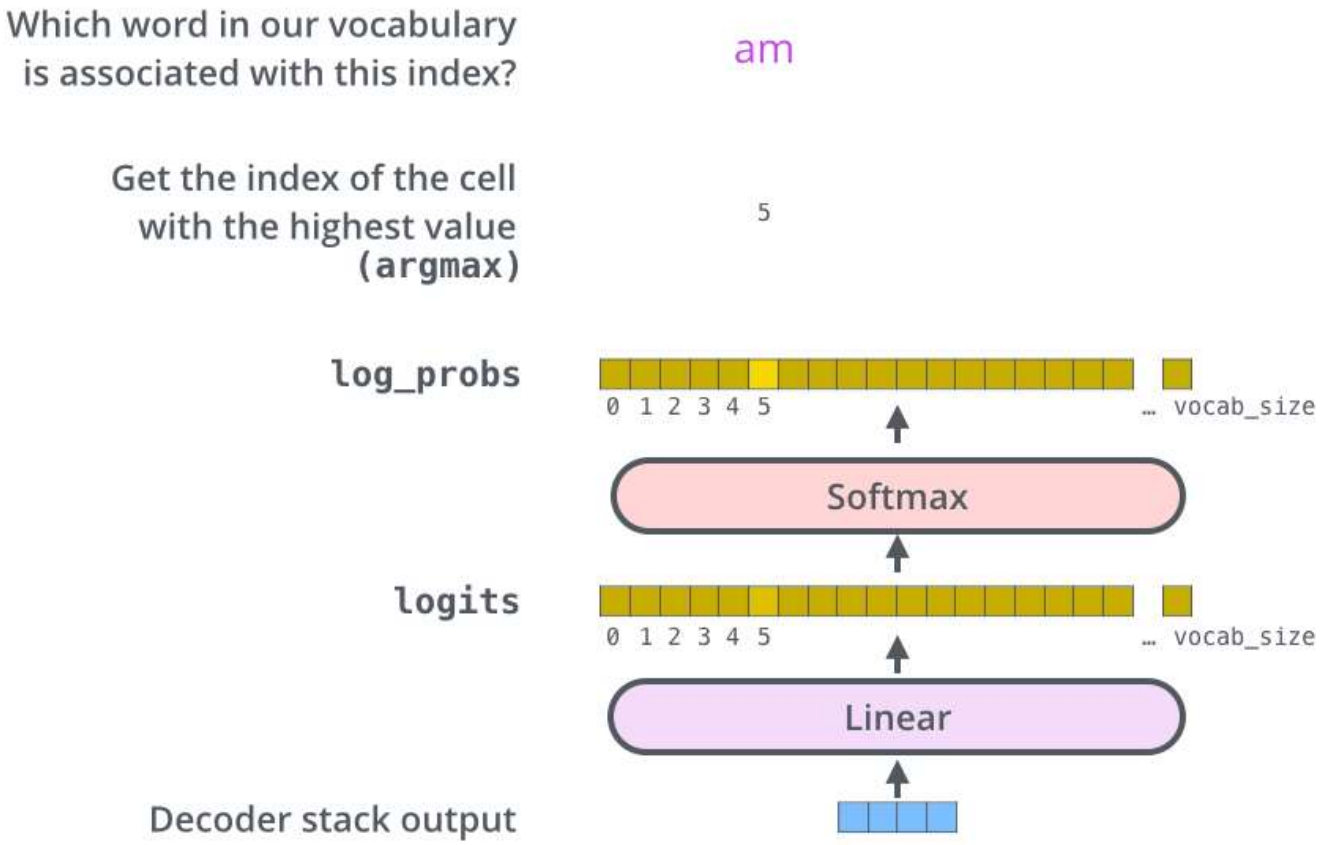
The self attention layers in the decoder operate in a slightly different way than the one in the encoder:

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.

Final Linear and 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.

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.



This figure starts from the bottom with the vector produced as the output of the decoder stack. It is then turned into an output word.

Hugging Face

Hugging Face has a large open-source community, with Transformers library among its top attractions. Transformers Library is backed by deep learning libraries– PyTorch and TensorFlow. It provides thousands of pretrained models to perform text classification, information retrieval, question and answer, translation, text generation, and summarisation. Transformers provide APIs that can be quickly downloaded and use pretrained models on a text to fine-tune users’ datasets.

The startup is putting efforts into growing the open-source community for the development of language models. The company said there is a disconnect between the research and the engineering teams in NLP. Big tech companies do not completely embrace the open-source approach, and even in a few cases where they have, the open-sourced repositories are hard to use and not well-maintained.

Hugging Face aims to become GitHub for machine learning. Hugging Face is one of the leading startups in the NLP space. Big tech companies such as Apple, Monzo, and Bing use its library in production.

Blog: https://huggingface.co/

```
###Refer Video
YouTubeVideo('00GKzGyWFEs', width=600, height=300)
```

Transformers Pipeline

Pipelines are the abstraction for the complex code behind the transformers library; It is easiest to use the pre-trained models for inference. It provides easy-to-use pipeline functions for a variety of tasks, including but not limited to, Named Entity Recognition, Masked Language Modeling, Sentiment Analysis, Feature Extraction, and Question Answering.

For the machine learning/deep learning experiment, we need to preprocess the data, train the model and write an inference script; in contrast with Pipeline functions, we need to import it and pass our raw data. The Pipeline will preprocess our data in the backend, including tokenization and padding and all the relevant processing steps for the algorithm's input, and return the output with just a call to it.

Refer:https://github.com/huggingface/transformers#quick-tour-of-pipelines

```
###Refer Video
```

```
YouTubeVideo('x8gdOPO35HA', width=600, height=300)
```

We need to install the Transformers library to use these fantastic pipeline functions.

Install the library using pip transformers

```
#install transformers using pip
```

```
Collecting transformers
  Downloading transformers-4.15.0-py3-none-any.whl (3.4 MB)
     |████████████████████████████████| 3.4 MB 4.2 MB/s
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-packages (fr
Collecting pyyaml>=5.1
  Downloading PyYAML-6.0-cp37-cp37m-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_
     |████████████████████████████████| 596 kB 57.9 MB/s
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (fr
Collecting sacremoses
  Downloading sacremoses-0.0.47-py2.py3-none-any.whl (895 kB)
     |████████████████████████████████| 895 kB 57.4 MB/s
Collecting tokenizers<0.11,>=0.10.1
  Downloading tokenizers-0.10.3-cp37-cp37m-manylinux_2_5_x86_64.manylinux1_x86_64.mar
     |████████████████████████████████| 3.3 MB 41.5 MB/s
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.7/dist-packages
Collecting huggingface-hub<1.0,>=0.1.0
  Downloading huggingface_hub-0.4.0-py3-none-any.whl (67 kB)
     |████████████████████████████████| 67 kB 5.9 MB/s
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.7
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.7/d
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-pa
```

```
Requirement already satisfied: click in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: joblib in /usr/local/lib/python3.7/dist-packages (fro
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from sa
Installing collected packages: pyyaml, tokenizers, sacremoses, huggingface-hub, trans
  Attempting uninstall: pyyaml
    Found existing installation: PyYAML 3.13
    Uninstalling PyYAML-3.13:
      Successfully uninstalled PyYAML-3.13
Successfully installed huggingface-hub-0.4.0 pyyaml-6.0 sacremoses-0.0.47 tokenizers-
```

Let's unwrap the magic box and see how it surprise us. First import the Pipeline from transformers library :)

Refer:https://huggingface.co/docs/transformers/main_classes/pipelines

```
#import pipeline from transformers
```

Sentiment Analysis

Sentiment analysis is used to predict the sentiment of the text, whether the text is positive or negative.

To perform sentiment analysis using Pipeline, we need to initialize the Pipeline with the 'sentiment-analysis' task as follows.

```
#initialize pipeline with sentiment analysis
```

```
No model was supplied, defaulted to distilbert-base-uncased-finetuned-sst-2-english (
Downloading: 100%  629/629 [00:00<00:00, 14.9kB/s]
Downloading: 100%  255M/255M [00:05<00:00, 49.3MB/s]
Downloading: 100%  48.0/48.0 [00:00<00:00, 692B/s]
Downloading: 100%  226k/226k [00:00<00:00, 365kB/s]
```

```
#input a sentence to perform sentiment analysis.
test= 'CloudyML is very Nice EduTech Company'
#print the result
```

```
[{'label': 'POSITIVE', 'score': 0.9994739890098572}]
```

Now we import **matplotlib** for visualize the data and **pandas** to import the dataset or analysis of data.

```
#Importing matplotlib library to plot pie chart.
```

```
#importing pandas library
```

After importing the libraries we load the data by using pandas.

```
#Loading the data to the variable
```

We print the dataset with help of head function.

```
#partial view of dataset from top
```

	Review	Liked
0	Wow... Loved this place.	1
1	Crust is not good.	0
2	Not tasty and the texture was just nasty.	0
3	Stopped by during the late May bank holiday of...	1
4	The selection on the menu was great and so wer...	1

In the next part, we segregate the data based on the positive and negative labels. Text with positive label is stored in 'pos' variable and text with negative label is stored in 'neg' variable.

```
#finding the positive and negative text in Data
```

```
Positive text
                                              Review  Liked
0                           Wow... Loved this place.      1
3  Stopped by during the late May bank holiday of...      1
4  The selection on the menu was great and so wer...      1
8                          The fries were great too.      1
9                                    A great touch.      1

Negative text
                                              Review  Liked
1                                Crust is not good.      0
2         Not tasty and the texture was just nasty.      0
5    Now I am getting angry and I want my damn pho.      0
6             Honeslty it didn't taste THAT fresh.)      0
7  The potatoes were like rubber and you could te...      0
```

Now we will check Null values present in the dataset. Null values effect the model accuary. So we are checking for the null values in dataset.

```
#checking the for null values
```

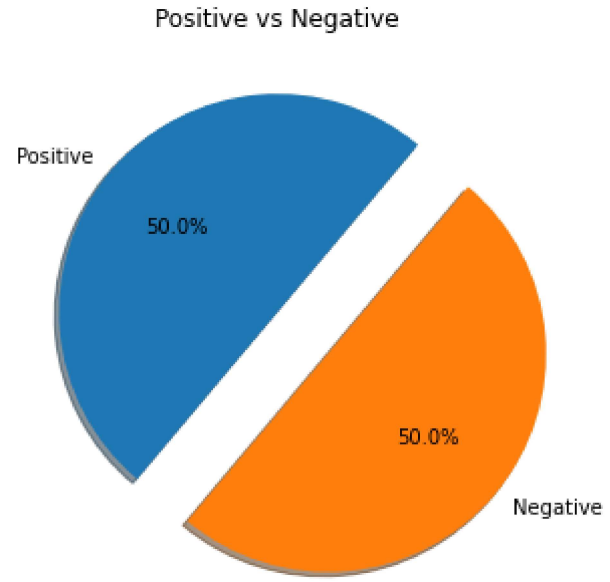
```
Review    0
Liked     0
dtype: int64
```

Here we can see that we have both postive and negative text in the data or dataset. So we can visulize the data by using matplotlib.

Now we plot a pie chart of the positive and negative data to understand the ratio of the data.
Matplotlib library is used to plot the pie chart using various parameters.
Do refer to the given documentation to know more about the used parameters:
https://matplotlib.org/stable/gallery/pie_and_polar_charts/pie_features.html

```
#Plotting the Postive vs Negative in piechart.
```

```
Text(0.5, 1.0, 'Positive vs Negative')
```



After visualization we will print the first 5 reviews for sentiment analysis.

```
#print first 5 reviews for sentiment analysis
```

```
0                           Wow... Loved this place.
1                                 Crust is not good.
2          Not tasty and the texture was just nasty.
3    Stopped by during the late May bank holiday of...
4    The selection on the menu was great and so wer...
5       Now I am getting angry and I want my damn pho.
Name: Review, dtype: object
```

We can see that the percentage of positive and negative data is equal i.e. 50% each. So we will apply sentiment analysis to first 5 reviews of the dataset.

```
#initialize a variable to 0

#run a for loop iterating through review column

  #check if the variable is less than equal to 5

    #apply the sentiment analysis function to current review

    #increment the variable

  #else break
```

```
[{'label': 'POSITIVE', 'score': 0.9998804330825806}]
[{'label': 'NEGATIVE', 'score': 0.9997690320014954}]
[{'label': 'NEGATIVE', 'score': 0.9996275901794434}]
[{'label': 'POSITIVE', 'score': 0.9995303153991699}]
[{'label': 'POSITIVE', 'score': 0.9996086955070496}]
[{'label': 'NEGATIVE', 'score': 0.992295503616333}]
```

You try with your nos of input.For understanding we have taken 5 you can change the inputs :-)

We can easily use other pipelines, including text summarization, named entity recognition, language translation, and many more. With this powerful transformers functionality, we can create excellent applications without even going into the coding ground. One of the advantages of using these pre-trained models is that we don't have to train our models from scratch, which sometimes takes days to prepare on a large volume of data, reducing our resource consumption and ultimately reducing our running cost.

TASK:

Text Summarization using Hugging Face Transformer, Hugging Face Transformer uses the Abstractive Summarization approach where the model develops new sentences in a new form, exactly like people do, and produces a whole distinct text that is shorter than the original.

You need to build text summarizarion model with the help of above reference code[Sentiment analysis code] and try to built it -__-

Hint: Initialize the HuggingFace summarization pipeline("summarization")

```
#import pipeline from transformers
```

```
# using pipeline  for summarization task
```

```
No model was supplied, defaulted to sshleifer/distilbart-cnn-12-6 (https://huggingfa
Downloading: 100%  1.76k/1.76k [00:00<00:00, 30.1kB/s]
Downloading: 100%  1.14G/1.14G [00:37<00:00, 31.4MB/s]
Downloading: 100%  26.0/26.0 [00:00<00:00, 559B/s]
Downloading: 100%  878k/878k [00:00<00:00, 1.79MB/s]
Downloading: 100%  446k/446k [00:00<00:00, 791kB/s]
```

```
#here we have given the input passage,You change the input as per your needs:)
text = """
Paul Walker is hardly the first actor to die during a production.
But Walker's death in November 2013 at the age of 40 after a car crash was especially eeri
The release of "Furious 7" on Friday offers the opportunity for fans to remember -- and po
"He was a person of humility, integrity, and compassion," military veteran Kyle Upham said
Walker secretly paid for the engagement ring Upham shopped for with his bride.
"We didn't know him personally but this was apparent in the short time we spent with him.
I know that we will never forget him and he will always be someone very special to us," sa
The actor was on break from filming "Furious 7" at the time of the fiery accident, which a
Producers said early on that they would not kill off Walker's character, Brian O'Connor, a
There are scenes that will resonate with the audience -- including the ending, in which th
Social media has also been paying homage to the late actor. A week after Walker's death, a
"""
```

```
#apply the summarization function to paragraph
```

```
Summary: [{'summary_text': ' Paul Walker died in November 2013 after a car crash in l
```

Great job!! You have come to the end of this assignment. Treat yourself for this :))

Do fill this feedback form

You may head on to the next assignment.