Name – Mrunal Wagh (NLP Notes)

NLP

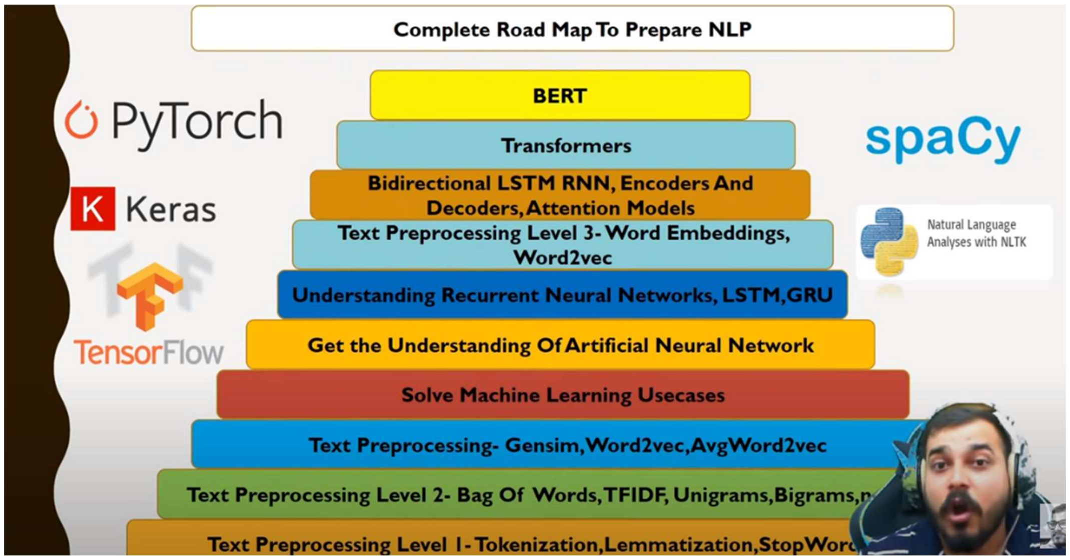
NLP

Natural language processing (NLP) is a branch of artificial intelligence that helps computers understand, interpret and manipulate human language. NLP draws from many disciplines, including computer science and computational linguistics, in its pursuit to fill the gap between human communication and computer understanding.

Stemming = use for word stamp word like base word. Word is positive or negative.

Lemmatization = do the meaningful Word

Stopword = Stopword lib is used for remove and, is, are like words, Stopword mostly use for sentimental Analysis.



Complete Road Map To Prepare NLP

BERT

Transformers

Bidirectional LSTM RNN, Encoders And Decoders, Attention Models

Text Preprocessing Level 3- Word Embeddings, Word2vec

Understanding Recurrent Neural Networks, LSTM,GRU

Get the Understanding Of Artificial Neural Network

Solve Machine Learning Usecases

Text Preprocessing- Gensim, Word2vec, AvgWord2vec

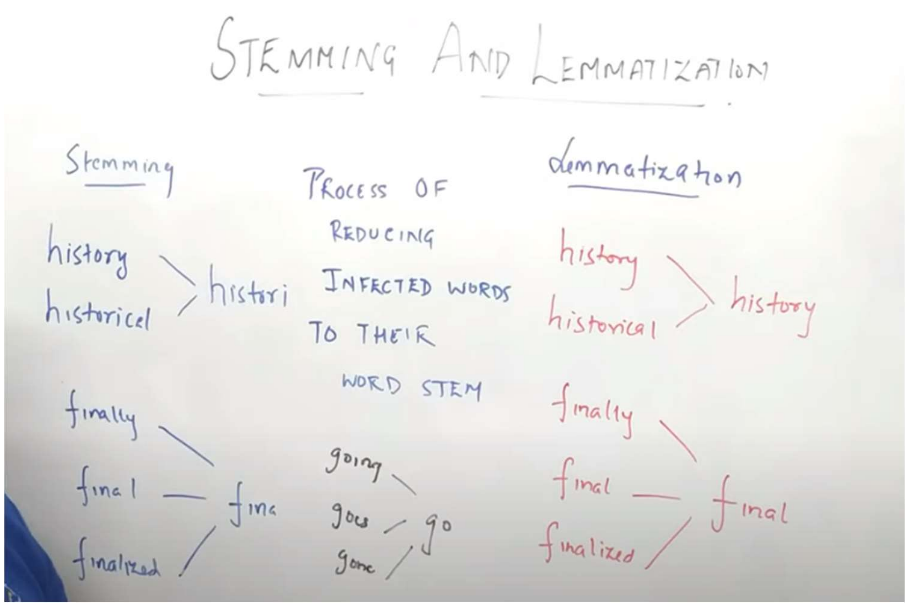
Text Preprocessing Level 2- Bag Of Words, TFIDF, Unigrams, Bigrams, n

Text Preprocessing Level 1- Tokenization, Lemmatization, StopWor

Tokenization of paragraphs/sentences

```
import nltk
nltk.download()

paragraph = """I have three visions for India. In 3000 years of our history, people from all over
        the world have come and invaded us, captured our lands, conquered our minds.
        From Alexander onwards, the Greeks, the Turks, the Moguls, the Portuguese, the British,
        the French, the Dutch, all of them came and looted us, took over what was ours.
        Yet we have not done this to any other nation. We have not conquered anyone.
        We have not grabbed their land, their culture,
        their history and tried to enforce our way of life on them.
        Why? Because we respect the freedom of others.That is why my
        first vision is that of freedom. I believe that India got its first vision of
        this in 1857, when we started the War of Independence. It is this freedom that
        we must protect and nurture and build on. If we are not free, no one will respect us.
        My second vision for India’s development. For fifty years we have been a developing nation.
        It is time we see ourselves as a developed nation. We are among the top 5 nations of the world
        in terms of GDP. We have a 10 percent growth rate in most areas. Our poverty levels are falling.
        Our achievements are being globally recognised today. Yet we lack the self-confidence to
        see ourselves as a developed nation, self-reliant and self-assured. Isn’t this incorrect?
        I have a third vision. India must stand up to the world. Because I believe that unless India
        stands up to the world, no one will respect us. Only strength respects strength. We must be
        strong not only as a military power but also as an economic power. Both must go hand-in-hand.
        My good fortune was to have worked with three great minds. Dr. Vikram Sarabhai of the Dept. of
        space, Professor Satish Dhawan, who succeeded him and Dr. Brahm Prakash, father of nuclear material.
        I was lucky to have worked with all three of them closely and consider this the great opportunity of my life.
        I see four milestones in my career"""

# Tokenizing sentences
sentences = nltk.sent_tokenize(paragraph)
# Tokenizing words
words = nltk.word_tokenize(paragraph)
```

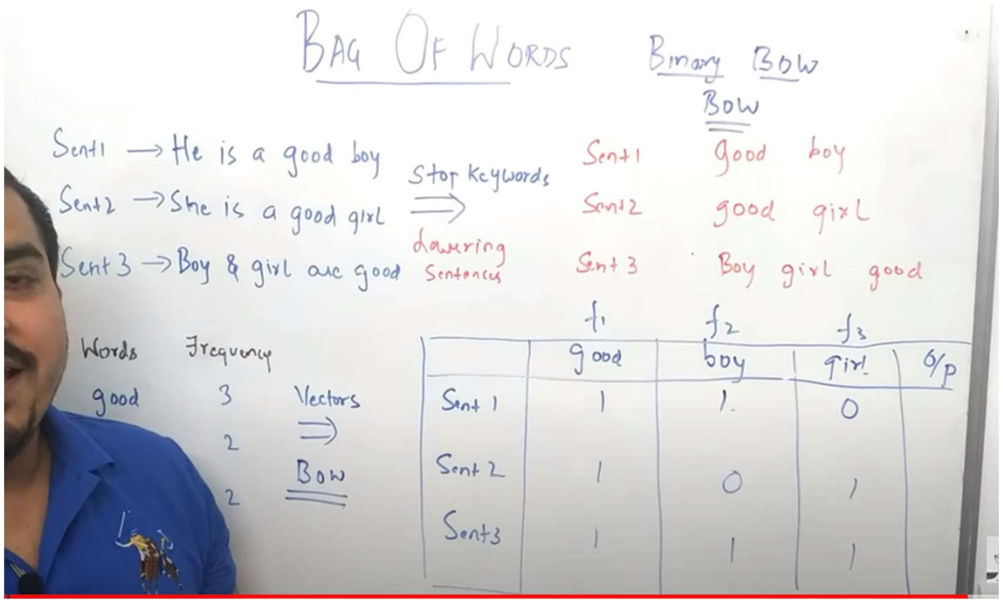
Stemming & Lemmatization



```
import nltk
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords

paragraph = """I have three visions for India. In 3000 years of our history, people from all over
        the world have come and invaded us, captured our lands, conquered our minds.
        From Alexander onwards, the Greeks, the Turks, the Moguls, the Portuguese, the British,
        the French, the Dutch, all of them came and looted us, took over what was ours.
        Yet we have not done this to any other nation. We have not conquered anyone.
        We have not grabbed their land, their culture,
        their history and tried to enforce our way of life on them.
        Why? Because we respect the freedom of others.That is why my
        first vision is that of freedom. I believe that India got its first vision of
        this in 1857, when we started the War of Independence. It is this freedom that
        we must protect and nurture and build on. If we are not free, no one will respect us.
        My second vision for India’s development. For fifty years we have been a developing nation.
        It is time we see ourselves as a developed nation. We are among the top 5 nations of the world
        in terms of GDP. We have a 10 percent growth rate in most areas. Our poverty levels are falling.
        Our achievements are being globally recognised today. Yet we lack the self-confidence to
        see ourselves as a developed nation, self-reliant and self-assured. Isn’t this incorrect?
        I have a third vision. India must stand up to the world. Because I believe that unless India
        stands up to the world, no one will respect us. Only strength respects strength. We must be
        strong not only as a military power but also as an economic power. Both must go hand-in-hand.
        My good fortune was to have worked with three great minds. Dr. Vikram Sarabhai of the Dept. of
        space, Professor Satish Dhawan, who succeeded him and Dr. Brahm Prakash, father of nuclear material.
        I was lucky to have worked with all three of them closely and consider this the great opportunity of my life.
        I see four milestones in my career"""

sentences = nltk.sent_tokenize(paragraph)
stemmer = PorterStemmer()

# Stemming
for i in range(len(sentences)):
    words = nltk.word_tokenize(sentences[i])
    words = [stemmer.stem(word) for word in words if word not in set(stopwords.words('english'))]
    sentences[i] = ' ‘. join(words)
```

```
#limitation

NLTK import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords

paragraph =  """I have three visions for India. In 3000 years of our history, people from all over
            the world have come and invaded us, captured our lands, conquered our minds.
            From Alexander onwards, the Greeks, the Turks, the Moguls, the Portuguese, the British,
            the French, the Dutch, all of them came and looted us, took over what was ours.
            Yet we have not done this to any other nation. We have not conquered anyone.
            We have not grabbed their land, their culture,
            their history and tried to enforce our way of life on them.
            Why? Because we respect the freedom of others.That is why my
            first vision is that of freedom. I believe that India got its first vision of
            this in 1857, when we started the War of Independence. It is this freedom that
            we must protect and nurture and build on. If we are not free, no one will respect us.
            My second vision for India's development. For fifty years we have been a developing nation.
            It is time we see ourselves as a developed nation. We are among the top 5 nations of the world
            in terms of GDP. We have a 10 percent growth rate in most areas. Our poverty levels are falling.
            Our achievements are being globally recognised today. Yet we lack the self-confidence to
            see ourselves as a developed nation, self-reliant and self-assured. Isn't this incorrect?
            I have a third vision. India must stand up to the world. Because I believe that unless India
            stands up to the world, no one will respect us. Only strength respects strength. We must be
            strong not only as a military power but also as an economic power. Both must go hand-in-hand.
            My good fortune was to have worked with three great minds. Dr. Vikram Sarabhai of the Dept. of
            space, Professor Satish Dhawan, who succeeded him and Dr. Brahm Prakash, father of nuclear material.
            I was lucky to have worked with all three of them closely and consider this the great opportunity of my life.
            I see four milestones in my career"""

sentences = nltk.sent_tokenize(paragraph)
lemmatizer = WordNetLemmatizer()
# Lemmatization
for i in range(len(sentences)):
  words = nltk.word_tokenize(sentences[i])
  words = [lemmatizer.lemmatize(word) for word in words if word not in set(stopwords.words('english'))]
  sentences[i] = ' '.join(words)
```

Bag of Words



BAG OF WORDS   Binary BOW

BOW

Sent1 → He is a good boy
Sent2 → She is a good girl
Sent3 → Boy & girl are good

Stop keywords ⇒ lowering Sentences

Sent1 good boy
Sent2 good girl
Sent 3 Boy girl good

Words	Frequency
good	3
	2
	2

Vectors ⇒ BOW

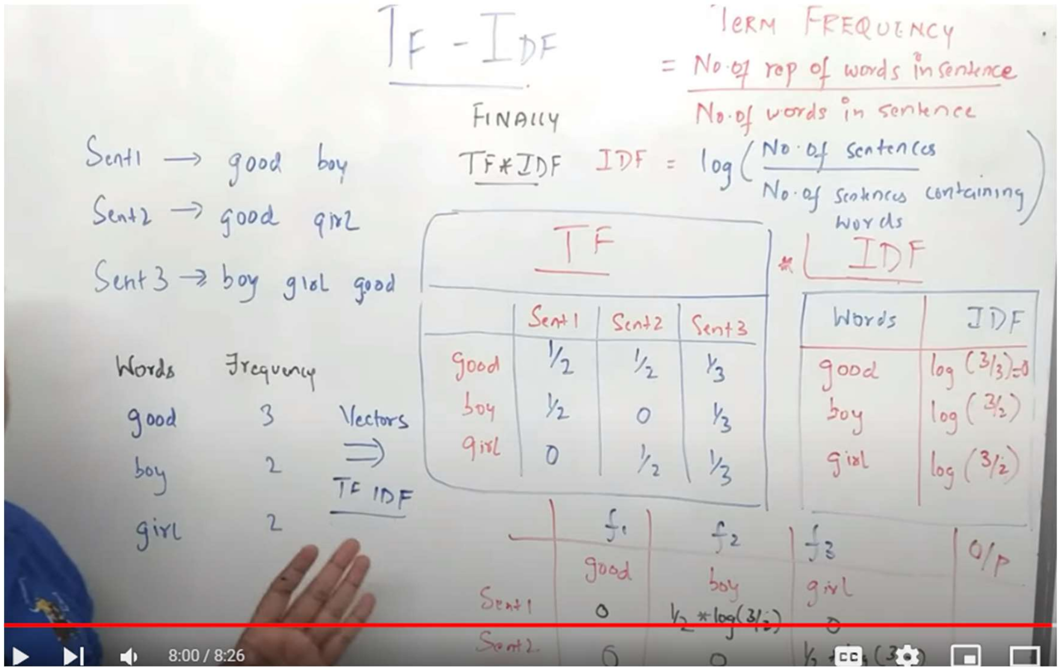
	f_1 good	f_2 boy	f_3 girl	O/P
Sent 1	1	1	0	
Sent 2	1	0	1	
Sent3	1	1	1	

Import re // re use for regular expression remove comma, punctuation , other special character
From nltk.corpus Import Stopword // is used for
from nltk.steam Import porterStemmer //is used for stemming purpose
from nltk.steam Import wordNetLeamatizer // is used for lemmatization
sentences = nltk.sent_tokenize(paragraph) // create sentences for paragraph
WordNetLimatizer = for limitization

From sklearn.feature_extraction.text import CountVectorizer // siketleran is use for vectorization bagOf Word
CountVectiorizer //responsible createing BagOfWord
fit_transform //is use for create matrix for bagof word

```
import nltk

paragraph =  """I have three visions for India. In 3000 years of our history, people from all over
        the world have come and invaded us, captured our lands, conquered our minds.
        From Alexander onwards, the Greeks, the Turks, the Moguls, the Portuguese, the British,
        the French, the Dutch, all of them came and looted us, took over what was ours.
        Yet we have not done this to any other nation. We have not conquered anyone.
        We have not grabbed their land, their culture,
        their history and tried to enforce our way of life on them.
        Why? Because we respect the freedom of others.That is why my
        first vision is that of freedom. I believe that India got its first vision of
        this in 1857, when we started the War of Independence. It is this freedom that
        we must protect and nurture and build on. If we are not free, no one will respect us.
        My second vision for India's development. For fifty years we have been a developing nation.
        It is time we see ourselves as a developed nation. We are among the top 5 nations of the world
        in terms of GDP. We have a 10 percent growth rate in most areas. Our poverty levels are falling.
        Our achievements are being globally recognised today. Yet we lack the self-confidence to
        see ourselves as a developed nation, self-reliant and self-assured. Isn't this incorrect?
        I have a third vision. India must stand up to the world. Because I believe that unless India
        stands up to the world, no one will respect us. Only strength respects strength. We must be
        strong not only as a military power but also as an economic power. Both must go hand-in-hand.
        My good fortune was to have worked with three great minds. Dr. Vikram Sarabhai of the Dept. of
        space, Professor Satish Dhawan, who succeeded him and Dr. Brahm Prakash, father of nuclear material.
        I was lucky to have worked with all three of them closely and consider this the great opportunity of my life.
        I see four milestones in my career"""
# Cleaning the texts
import re
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from nltk.stem import WordNetLemmatizer

ps = PorterStemmer()
wordnet=WordNetLemmatizer()
sentences = nltk.sent_tokenize(paragraph)
corpus = []
for i in range(len(sentences)):
    review = re.sub('[^a-zA-Z]', ' ', sentences[i])
    review = review.lower()
    review = review.split()
    review = [ps.stem(word) for word in review if not word in set(stopwords.words('english'))]
    review = ' '.join(review)
    corpus.append(review)

# Creating the Bag of Words model
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features = 1500)
X = cv.fit_transform(corpus).toarray()
```

TF-IDF



TF - IDF

Term Frequency $= \frac{\text{No. of rep of words in sentence}}{\text{No. of words in sentence}}$

FINALLY TF*IDF

IDF $= \log\left(\frac{\text{No. of sentences}}{\text{No. of sentences containing words}}\right)$

Sent1 → good boy
Sent2 → good girl
Sent3 → boy girl good

Words	Frequency
good	3
boy	2
girl	2

Vectors ⇒ TF IDF

TF

	Sent1	Sent2	Sent3
good	1/2	1/2	1/3
boy	1/2	0	1/3
girl	0	1/2	1/3

* IDF

Words	IDF
good	log(3/3)=0
boy	log(3/2)
girl	log(3/2)

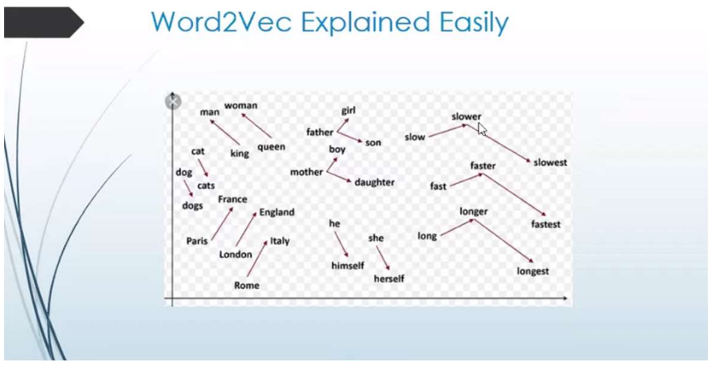
	f1	f2	f3	O/P
	good	boy	girl	
Sent1	0	1/2 * log(3/2)	0	
Sent2	0	0	1/2 * log(3/2)	

TF -IDF term frequency
TF = No. of reptation word in sentence /No. of word in sentence
IDF = Invert Document frequency
IDF = log(No. of sentence/ No. of sentences containing word)

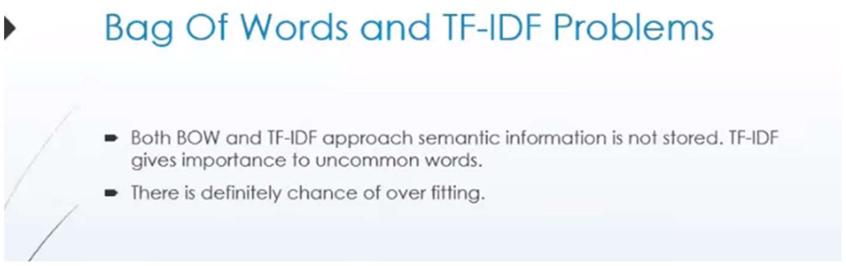
```
import nltk
paragraph =  """I have three visions for India. In 3000 years of our history, people from all over
          the world have come and invaded us, captured our lands, conquered our minds.
          From Alexander onwards, the Greeks, the Turks, the Moguls, the Portuguese, the British,
          the French, the Dutch, all of them came and looted us, took over what was ours.
          Yet we have not done this to any other nation. We have not conquered anyone.
          We have not grabbed their land, their culture,
          their history and tried to enforce our way of life on them.
          Why? Because we respect the freedom of others.That is why my
          first vision is that of freedom. I believe that India got its first vision of
          this in 1857, when we started the War of Independence. It is this freedom that
          we must protect and nurture and build on. If we are not free, no one will respect us.
          My second vision for India's development. For fifty years we have been a developing nation.
          It is time we see ourselves as a developed nation. We are among the top 5 nations of the world
          in terms of GDP. We have a 10 percent growth rate in most areas. Our poverty levels are falling.
          Our achievements are being globally recognised today. Yet we lack the self-confidence to
          see ourselves as a developed nation, self-reliant and self-assured. Isn't this incorrect?
          I have a third vision. India must stand up to the world. Because I believe that unless India
          stands up to the world, no one will respect us. Only strength respects strength. We must be
          strong not only as a military power but also as an economic power. Both must go hand-in-hand.
          My good fortune was to have worked with three great minds. Dr. Vikram Sarabhai of the Dept. of
          space, Professor Satish Dhawan, who succeeded him and Dr. Brahm Prakash, father of nuclear material.
          I was lucky to have worked with all three of them closely and consider this the great opportunity of my life.
          I see four milestones in my career"""


# Cleaning the texts
import re
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from nltk.stem import WordNetLemmatizer

ps = PorterStemmer()
wordnet=WordNetLemmatizer()
```

```
sentences = nltk.sent_tokenize(paragraph)
corpus = []
for i in range(len(sentences)):
    review = re.sub('[^a-zA-Z]', ' ', sentences[i])
    review = review.lower()
    review = review.split()
    review = [wordnet.lemmatize(word) for word in review if not word in set(stopwords.words('english'))]
    review = ' '.join(review)
    corpus.append(review)

# Creating the TF-IDF model
from sklearn.feature_extraction.text import TfidfVectorizer
cv = TfidfVectorizer()
X = cv.fit_transform(corpus).toarray()
```

Word2Vec

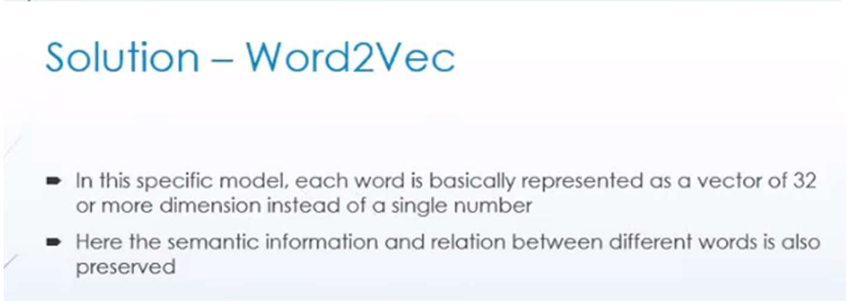


Word2Vec Explained Easily



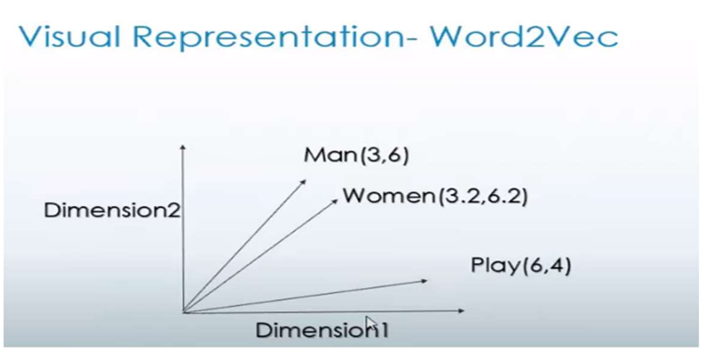
Bag Of Words and TF-IDF Problems

- Both BOW and TF-IDF approach semantic information is not stored. TF-IDF gives importance to uncommon words.
- There is definitely chance of over fitting.

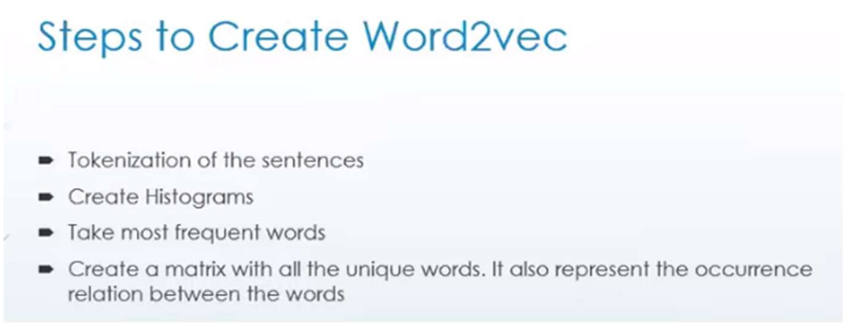


Solution – Word2Vec

- In this specific model, each word is basically represented as a vector of 32 or more dimension instead of a single number
- Here the semantic information and relation between different words is also preserved



Visual Representation- Word2Vec



Steps to Create Word2vec

- Tokenization of the sentences
- Create Histograms
- Take most frequent words
- Create a matrix with all the unique words. It also represent the occurrence relation between the words

```
import nltk
from gensim.models import Word2Vec
from nltk.corpus import stopwords
import re

paragraph = """I have three visions for India. In 3000 years of our history, people from all over
        the world have come and invaded us, captured our lands, conquered our minds.
        From Alexander onwards, the Greeks, the Turks, the Moguls, the Portuguese, the British,
        the French, the Dutch, all of them came and looted us, took over what was ours.
        Yet we have not done this to any other nation. We have not conquered anyone.
        We have not grabbed their land, their culture,
        their history and tried to enforce our way of life on them.
        Why? Because we respect the freedom of others.That is why my
        first vision is that of freedom. I believe that India got its first vision of
        this in 1857, when we started the War of Independence. It is this freedom that
        we must protect and nurture and build on. If we are not free, no one will respect us.
        My second vision for India's development. For fifty years we have been a developing nation.
        It is time we see ourselves as a developed nation. We are among the top 5 nations of the world
        in terms of GDP. We have a 10 percent growth rate in most areas. Our poverty levels are falling.
        Our achievements are being globally recognised today. Yet we lack the self-confidence to
        see ourselves as a developed nation, self-reliant and self-assured. Isn't this incorrect?
        I have a third vision. India must stand up to the world. Because I believe that unless India
        stands up to the world, no one will respect us. Only strength respects strength. We must be
        strong not only as a military power but also as an economic power. Both must go hand-in-hand.
        My good fortune was to have worked with three great minds. Dr. Vikram Sarabhai of the Dept. of
        space, Professor Satish Dhawan, who succeeded him and Dr. Brahm Prakash, father of nuclear material.
        I was lucky to have worked with all three of them closely and consider this the great opportunity of my life.
        I see four milestones in my career"""

# Preprocessing the data
text = re.sub(r'\[[0-9]*\]',' ',paragraph)
text = re.sub(r'\s+',' ',text)
text = text.lower()
text = re.sub(r'\d',' ',text)
text = re.sub(r'\s+',' ',text)

# Preparing the dataset
sentences = nltk.sent_tokenize(text)
sentences = [nltk.word_tokenize(sentence) for sentence in sentences]

for i in range(len(sentences)):
    sentences[i] = [word for word in sentences[i] if word not in stopwords.words('english')]

# Training the Word2Vec model
model = Word2Vec(sentences, min_count=1)

words = model.wv.vocab
# Finding Word Vectors
vector = model.wv['war']

# Most similar words
similar = model.wv.most_similar('vikram')
```

ML Project: - Spam SMS Classifier

1st step download code from this github link (link = https://github.com/krishnaik06/SpamClassifier/tree/master)
2nd download dataset(sms spam collection) from same git hub link **or** UCL SMS Spam Collection Data set

Code

```
# importing the Dataset
import pandas as pd   //pandas use for reading particular dataset.
messages = pd.read_csv('smsspamcollection/SMSSpamCollection', sep='\t',
                       names=["label", "message"])

#Data cleaning and preprocessing
import re
import nltk
nltk.download('stopwords')

from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
ps = PorterStemmer()
corpus = []
for i in range(0, len(messages)):
    review = re.sub('[^a-zA-Z]', ' ', messages['message'][i])
    review = review.lower()
    review = review.split()
    review = [ps.stem(word) for word in review if not word in stopwords.words('english')]
    review = ' '.join(review)
    corpus.append(review)

# Creating the Bag of Words model
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features=2500)
X = cv.fit_transform(corpus).toarray()

y=pd.get_dummies(messages['label'])
y=y.iloc[:,1].values

# Train Test Split

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20, random_state = 0)

# Training model using Naive bayes classifier

from sklearn.naive_bayes import MultinomialNB
spam_detect_model = MultinomialNB().fit(X_train, y_train)
y_pred=spam_detect_model.predict(X_test)
```

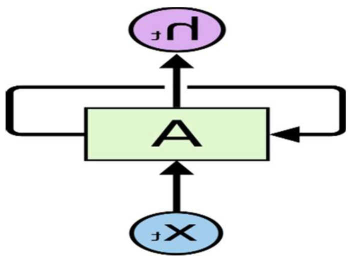
sequence-to-sequence learning in Keras

Recurrent Neural Networks

Humans don’t start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don’t throw everything away and start thinking from scratch again. Your thoughts have persistence.

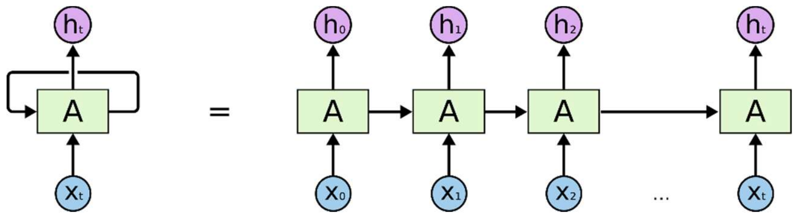
Traditional neural networks can’t do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It’s unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.



Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, A�, looks at some input xt�� and outputs a value htℎ�. A loop allows information to be passed from one step of the network to the next.
These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren’t all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



An unrolled recurrent neural network.

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They’re the natural architecture of neural network to use for such data.

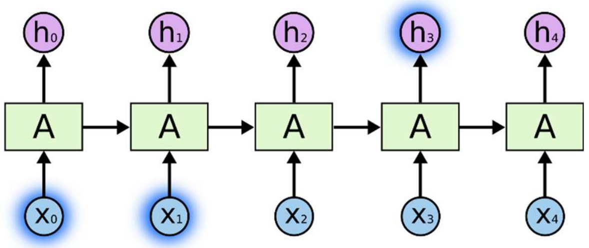
And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning… The list goes on. I’ll leave discussion of the amazing feats one can achieve with RNNs to Andrej Karpathy’s excellent blog post, The Unreasonable Effectiveness of Recurrent Neural Networks. But they really are pretty amazing.

Essential to these successes is the use of “LSTMs,” a very special kind of recurrent neural network which works, for many tasks, much much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them. It’s these LSTMs that this essay will explore.

The Problem of Long-Term Dependencies

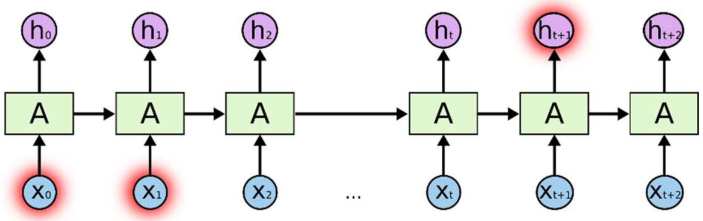
One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they’d be extremely useful. But can they? It depends.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in “the clouds are in the *sky*,” we don’t need any further context – it’s pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it’s needed is small, RNNs can learn to use the past information.



But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France… I speak fluent *French*." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) [German] and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.

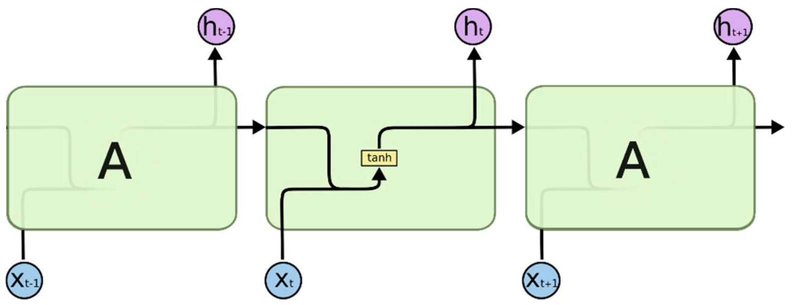
Thankfully, LSTMs don't have this problem!

LSTM Networks

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work.[1] They work tremendously well on a large variety of problems, and are now widely used.

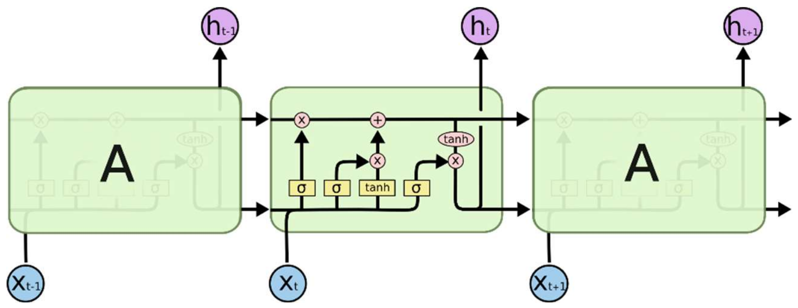
LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.



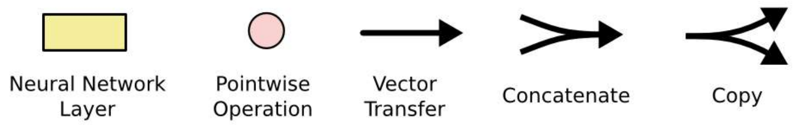
The repeating module in a standard RNN contains a single layer.

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



The repeating module in an LSTM contains four interacting layers.

Don't worry about the details of what's going on. We'll walk through the LSTM diagram step by step later. For now, let's just try to get comfortable with the notation we'll be using.

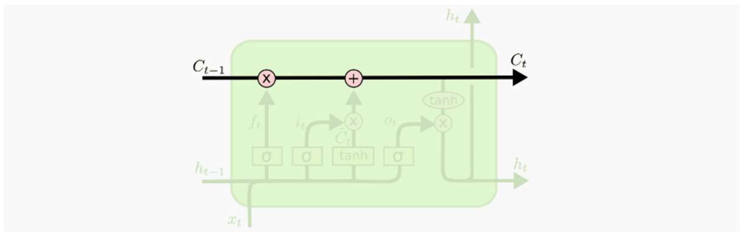


In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

The Core Idea Behind LSTMs

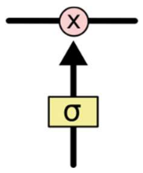
The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



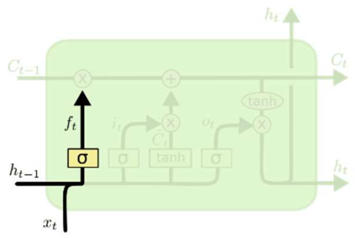
The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!"

An LSTM has three of these gates, to protect and control the cell state.

Step-by-Step LSTM Walk Through

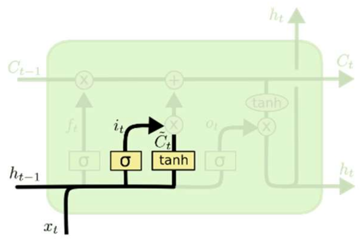
The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at ht−1ℎ�−1 and xt��, and outputs a number between 00 and 11 for each number in the cell state Ct−1��−1. A 11 represents "completely keep this" while a 00 represents "completely get rid of this."

Let’s go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



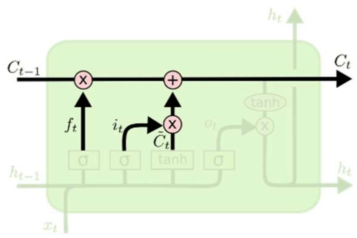
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

The next step is to decide what new information we’re going to store in the cell state. This has two parts. First, a sigmoid layer called the “input gate layer” decides which values we’ll update. Next, a tanh layer creates a vector of new candidate values, C~t�~�, that could be added to the state. In the next step, we’ll combine these two to create an update to the state.
In the example of our language model, we’d want to add the gender of the new subject to the cell state, to replace the old one we’re forgetting.



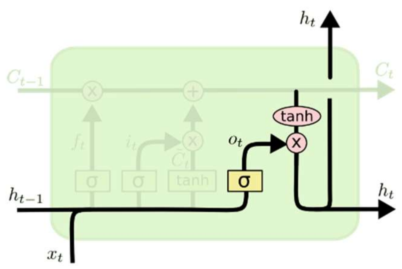
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

It’s now time to update the old cell state, Ct−1��−1, into the new cell state Ct��. The previous steps already decided what to do, we just need to actually do it.
We multiply the old state by ft��, forgetting the things we decided to forget earlier. Then we add it∗C~t��∗�~�. This is the new candidate values, scaled by how much we decided to update each state value.
In the case of the language model, this is where we’d actually drop the information about the old subject’s gender and add the new information, as we decided in the previous steps.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finally, we need to decide what we’re going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we’re going to output. Then, we put the cell state through tanhtanh (to push the values to be between −1−1 and 11) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.
For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that’s what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that’s what follows next.

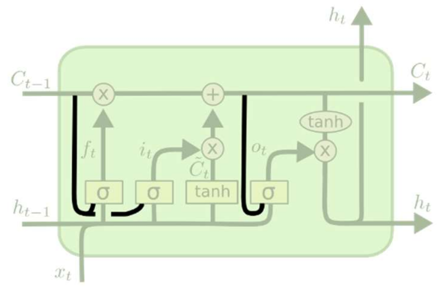


$$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Variants on Long Short Term Memory

What I’ve described so far is a pretty normal LSTM. But not all LSTMs are the same as the above. In fact, it seems like almost every paper involving LSTMs uses a slightly different version. The differences are minor, but it’s worth mentioning some of them.

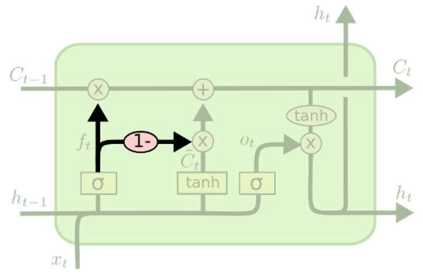
One popular LSTM variant, introduced by Gers & Schmidhuber (2000), is adding “peephole connections.” This means that we let the gate layers look at the cell state.



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

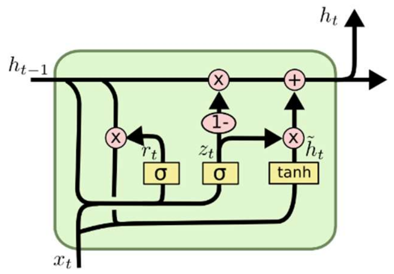
The above diagram adds peepholes to all the gates, but many papers will give some peepholes and not others.

Another variation is to use coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we’re going to input something in its place. We only input new values to the state when we forget something older.



$$C_t = f_t * C_{t-1} + \boldsymbol{(1 - f_t)} * \tilde{C}_t$$

A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit, or GRU, introduced by Cho, et al. (2014). It combines the forget and input gates into a single “update gate.” It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

These are only a few of the most notable LSTM variants. There are lots of others, like Depth Gated RNNs by Yao, et al. (2015). There’s also some completely different approach to tackling long-term dependencies, like Clockwork RNNs by Koutnik, et al. (2014).

Which of these variants is best? Do the differences matter? Greff, et al. (2015) do a nice comparison of popular variants, finding that they’re all about the same. Jozefowicz, et al. (2015) tested more than ten thousand RNN architectures, finding some that worked better than LSTMs on certain tasks.

Conclusion

Earlier, I mentioned the remarkable results people are achieving with RNNs. Essentially all of these are achieved using LSTMs. They really work a lot better for most tasks!

Written down as a set of equations, LSTMs look pretty intimidating. Hopefully, walking through them step by step in this essay has made them a bit more approachable.

LSTMs were a big step in what we can accomplish with RNNs. It’s natural to wonder: is there another big step? A common opinion among researchers is: “Yes! There is a next step and it’s attention!” The idea is to let every step of an RNN pick information to look at from some larger collection of information. For example, if you are using an RNN to create a caption describing an image, it might pick a part of the image to look at for every word it outputs. In fact, Xu, *et al.* (2015) do exactly this – it might be a fun starting point if you want to explore attention! There’s been a number of really exciting results using attention, and it seems like a lot more are around the corner…

Attention isn't the only exciting thread in RNN research. For example, Grid LSTMs by Kalchbrenner, *et al.* (2015) seem extremely promising. Work using RNNs in generative models – such as Gregor, *et al.* (2015), Chung, *et al.* (2015), or Bayer & Osendorfer (2015) – also seems very interesting. The last few years have been an exciting time for recurrent neural networks, and the coming ones promise to only be more so!

How to Use Word Embedding Layers for Deep Learning with Keras

Word embeddings provide a dense representation of words and their relative meanings.

They are an improvement over sparse representations used in simpler bag of word model representations.

Word embeddings can be learned from text data and reused among projects. They can also be learned as part of fitting a neural network on text data.

In this tutorial, you will discover how to use word embeddings for deep learning in Python with Keras.

After completing this tutorial, you will know:

- About word embeddings and that Keras supports word embeddings via the Embedding layer.
- How to learn a word embedding while fitting a neural network.
- How to use a pre-trained word embedding in a neural network.

Overview

This is divided into 3 parts; they are:

1. Word Embedding
2. Keras Embedding Layer
3. Example of Learning an Embedding
4. Example of Using Pre-Trained GloVe Embedding

1. Word Embedding

A word embedding is a class of approaches for representing words and documents using a dense vector representation.

It is an improvement over more the traditional bag-of-word model encoding schemes where large sparse vectors were used to represent each word or to score each word within a vector to represent an entire vocabulary. These representations were sparse because the vocabularies were vast and a given word or document would be represented by a large vector comprised mostly of zero values.

Instead, in an embedding, words are represented by dense vectors where a vector represents the projection of the word into a continuous vector space.

The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used.

The position of a word in the learned vector space is referred to as its embedding.

Two popular examples of methods of learning word embeddings from text include:

- Word2Vec.
- GloVe.

In addition to these carefully designed methods, a word embedding can be learned as part of a deep learning model. This can be a slower approach, but tailors the model to a specific training dataset.

2. Keras Embedding Layer

Keras offers an Embedding layer that can be used for neural networks on text data.

It requires that the input data be integer encoded, so that each word is represented by a unique integer. This data preparation step can be performed using the Tokenizer API also provided with Keras.

The Embedding layer is initialized with random weights and will learn an embedding for all of the words in the training dataset.

It is a flexible layer that can be used in a variety of ways, such as:

- It can be used alone to learn a word embedding that can be saved and used in another model later.
- It can be used as part of a deep learning model where the embedding is learned along with the model itself.
- It can be used to load a pre-trained word embedding model, a type of transfer learning.

The Embedding layer is defined as the first hidden layer of a network. It must specify 3 arguments:

It must specify 3 arguments:

- **input_dim**: This is the size of the vocabulary in the text data. For example, if your data is integer encoded to values between 0-10, then the size of the vocabulary would be 11 words.
- **output_dim**: This is the size of the vector space in which words will be embedded. It defines the size of the output vectors from this layer for each word. For example, it could be 32 or 100 or even larger. Test different values for your problem.
- **input_length**: This is the length of input sequences, as you would define for any input layer of a Keras model. For example, if all of your input documents are comprised of 1000 words, this would be 1000.

For example, below we define an Embedding layer with a vocabulary of 200 (e.g. integer encoded words from 0 to 199, inclusive), a vector space of 32 dimensions in which words will be embedded, and input documents that have 50 words each.

```
e = Embedding(200, 32, input_length=50)
```

The Embedding layer has weights that are learned. If you save your model to file, this will include weights for the Embedding layer.

The output of the *Embedding* layer is a 2D vector with one embedding for each word in the input sequence of words (input document).

If you wish to connect a *Dense* layer directly to an Embedding layer, you must first flatten the 2D output matrix to a 1D vector using the *Flatten* layer.

Now, let’s see how we can use an Embedding layer in practice.

3. Example of Learning an Embedding

In this section, we will look at how we can learn a word embedding while fitting a neural network on a text classification problem.

We will define a small problem where we have 10 text documents, each with a comment about a piece of work a student submitted. Each text document is classified as positive “1” or negative “0”. This is a simple sentiment analysis problem.

First, we will define the documents and their class labels.

```
# define documents
docs = ['Well done!',
 'Good work',
 'Great effort',
 'nice work',
 'Excellent!',
 'Weak',
 'Poor effort!',
 'not good',
 'poor work',
 'Could have done better.']
# define class labels
labels = array([1,1,1,1,1,0,0,0,0,0])
```

Next, we can integer encode each document. This means that as input the Embedding layer will have sequences of integers. We could experiment with other more sophisticated bag of word model encoding like counts or TF-IDF.

Keras provides the one_hot() function that creates a hash of each word as an efficient integer encoding. We will estimate the vocabulary size of 50, which is much larger than needed to reduce the probability of collisions from the hash function.

```
# integer encode the documents
vocab_size = 50
encoded_docs = [one_hot(d, vocab_size) for d in docs]
print(encoded_docs)
```

The sequences have different lengths and Keras prefers inputs to be vectorized and all inputs to have the same length. We will pad all input sequences to have the length of 4. Again, we can do this with a built in Keras function, in this case the pad_sequences() function.

```
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
```

We are now ready to define our *Embedding* layer as part of our neural network model.

The *Embedding* has a vocabulary of 50 and an input length of 4. We will choose a small embedding space of 8 dimensions.

The model is a simple binary classification model. Importantly, the output from the *Embedding* layer will be 4 vectors of 8 dimensions each, one for each word. We flatten this to a one 32-element vector to pass on to the *Dense* output layer.

```
# define the model
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# summarize the model
print(model.summary())
```

Finally, we can fit and evaluate the classification model.

```
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

The complete code listing is provided below.

```
from numpy import array
from keras.preprocessing.text import one_hot
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
# define documents
docs = ['Well done!',
'Good work',
'Great effort',
'nice work',
'Excellent!',
'Weak',
'Poor effort!',
'not good',
'poor work',
'Could have done better.']
# define class labels
labels = array([1,1,1,1,1,0,0,0,0,0])
# integer encode the documents
vocab_size = 50
encoded_docs = [one_hot(d, vocab_size) for d in docs]
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
```

```
print(padded_docs)
# define the model
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# summarize the model
print(model.summary())
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

Running the example first prints the integer encoded documents.

```
[[6, 16], [42, 24], [2, 17], [42, 24], [18], [17], [22, 17], [27, 42], [22, 24], [49, 46, 16, 34]]
```

Then the padded versions of each document are printed, making them all uniform length.

```
[[ 6 16  0  0]
 [42 24  0  0]
 [ 2 17  0  0]
 [42 24  0  0]
 [18  0  0  0]
 [17  0  0  0]
 [22 17  0  0]
 [27 42  0  0]
 [22 24  0  0]
 [49 46 16 34]]
```

After the network is defined, a summary of the layers is printed. We can see that as expected, the output of the Embedding layer is a 4×8 matrix and this is squashed to a 32-element vector by the Flatten layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 4, 8)              400
_________________________________________________________________
flatten_1 (Flatten)          (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 433
Trainable params: 433
Non-trainable params: 0
_________________________________________________________________
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Finally, the accuracy of the trained model is printed, showing that it learned the training dataset perfectly (which is not surprising).

```
Accuracy: 100.000000
```

You could save the learned weights from the Embedding layer to file for later use in other models.

You could also use this model generally to classify other documents that have the same kind vocabulary seen in the test dataset.

Next, let's look at loading a pre-trained word embedding in Keras.

4. Example of Using Pre-Trained GloVe Embedding

The Keras Embedding layer can also use a word embedding learned elsewhere.

It is common in the field of Natural Language Processing to learn, save, and make freely available word embeddings.

For example, the researchers behind GloVe method provide a suite of pre-trained word embeddings on their website released under a public domain license. See:

- GloVe: Global Vectors for Word Representation

The smallest package of embeddings is 822Mb, called "*glove.6B.zip*". It was trained on a dataset of one billion tokens (words) with a vocabulary of 400 thousand words. There are a few different embedding vector sizes, including 50, 100, 200 and 300 dimensions.

You can download this collection of embeddings and we can seed the Keras *Embedding* layer with weights from the pre-trained embedding for the words in your training dataset.

This example is inspired by an example in the Keras project: pretrained_word_embeddings.py.

After downloading and unzipping, you will see a few files, one of which is "*glove.6B.100d.txt*", which contains a 100-dimensional version of the embedding.

If you peek inside the file, you will see a token (word) followed by the weights (100 numbers) on each line. For example, below are the first line of the embedding ASCII text file showing the embedding for "*the*".

```
the -0.038194 -0.24487 0.72812 -0.39961 0.083172 0.043953 -0.39141 0.3344 -0.57545 0.087459 0.28787 -0.06731 0.30906 -0.26384 -0.13231 -0.20757 0.33395 -0.33848 -0.31743 -0.48336 0.1464 -0.37304 0.34577 0.052041 0.44946 -0.46971 0.02628 -0.54155 -0.15518 -0.14107 -0.039722 0.28277 0.14393 0.23464 -0.31021 0.086173 0.20397 0.52624 0.17164 -0.082378 -0.71787 -0.41531 0.20335 -0.12763 0.41367 0.55187 0.57908 -0.33477 -0.36559 -0.54857 -0.062892 0.26584 0.30205 0.99775 -0.80481 -3.0243 0.01254 -0.36942 2.2167 0.72201 -0.24978 0.92136 0.034514 0.46745 1.1079 -0.19358 -0.074575 0.23353 -0.052062 -0.22044 0.057162 -0.15806 -0.30798 -0.41625 0.37972 0.15006 -0.53212 -0.2055 -1.2526 0.071624 0.70565 0.49744 -0.42063 0.26148 -1.538 -0.30223 -0.073438 -0.28312 0.37104 -0.25217 0.016215 -0.017099 -0.38984 0.87424 -0.72569 -0.51058 -0.52028 -0.1459 0.8278 0.27062
```

As in the previous section, the first step is to define the examples, encode them as integers, then pad the sequences to be the same length.

In this case, we need to be able to map words to integers as well as integers to words.

Keras provides a Tokenizer class that can be fit on the training data, can convert text to sequences consistently by calling the *texts_to_sequences()* method on the *Tokenizer* class, and provides access to the dictionary mapping of words to integers in a *word_index* attribute.

```
# define documents
docs = ['Well done!',
 'Good work',
 'Great effort',
 'nice work',
 'Excellent!',
 'Weak',
 'Poor effort!',
 'not good',
 'poor work',
 'Could have done better.']
# define class labels
labels = array([1,1,1,1,1,0,0,0,0,0])
# prepare tokenizer
t = Tokenizer()
t.fit_on_texts(docs)
vocab_size = len(t.word_index) + 1
# integer encode the documents
encoded_docs = t.texts_to_sequences(docs)
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
```

Next, we need to load the entire GloVe word embedding file into memory as a dictionary of word to embedding array.

```
# load the whole embedding into memory
embeddings_index = dict()
f = open('glove.6B.100d.txt')
for line in f:
 values = line.split()
 word = values[0]
 coefs = asarray(values[1:], dtype='float32')
 embeddings_index[word] = coefs
f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
```

This is pretty slow. It might be better to filter the embedding for the unique words in your training data.

Next, we need to create a matrix of one embedding for each word in the training dataset. We can do that by enumerating all unique words in the *Tokenizer.word_index* and locating the embedding weight vector from the loaded GloVe embedding.

The result is a matrix of weights only for words we will see during training.

```
# create a weight matrix for words in training docs
embedding_matrix = zeros((vocab_size, 100))
for word, i in t.word_index.items():
 embedding_vector = embeddings_index.get(word)
 if embedding_vector is not None:
 embedding_matrix[i] = embedding_vector
```

Now we can define our model, fit, and evaluate it as before.

The key difference is that the embedding layer can be seeded with the GloVe word embedding weights. We chose the 100-dimensional version, therefore the Embedding layer must be defined with *output_dim* set to 100. Finally, we do not want to update the learned word weights in this model, therefore we will set the *trainable* attribute for the model to be *False*.

```
e = Embedding(vocab_size, 100, weights=[embedding_matrix], input_length=4, trainable=False)
```

The complete worked example is listed below.

```
from numpy import array
from numpy import asarray
from numpy import zeros
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Embedding
# define documents
docs = ['Well done!',
 'Good work',
 'Great effort',
 'nice work',
 'Excellent!',
 'Weak',
 'Poor effort!',
 'not good',
 'poor work',
 'Could have done better.']
# define class labels
labels = array([1,1,1,1,1,0,0,0,0,0])
# prepare tokenizer
t = Tokenizer()
t.fit_on_texts(docs)
vocab_size = len(t.word_index) + 1
# integer encode the documents
encoded_docs = t.texts_to_sequences(docs)
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
```

```
print(padded_docs)
# load the whole embedding into memory
embeddings_index = dict()
f = open('../glove_data/glove.6B/glove.6B.100d.txt')
for line in f:
 values = line.split()
 word = values[0]
 coefs = asarray(values[1:], dtype='float32')
 embeddings_index[word] = coefs
f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
# create a weight matrix for words in training docs
embedding_matrix = zeros((vocab_size, 100))
for word, i in t.word_index.items():
 embedding_vector = embeddings_index.get(word)
 if embedding_vector is not None:
 embedding_matrix[i] = embedding_vector
# define model
model = Sequential()
e = Embedding(vocab_size, 100, weights=[embedding_matrix], input_length=4, trainable=False)
model.add(e)
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# summarize the model
print(model.summary())
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example may take a bit longer, but then demonstrates that it is just as capable of fitting this simple problem.

```
[[6, 2], [3, 1], [7, 4], [8, 1], [9], [10], [5, 4], [11, 3], [5, 1], [12, 13, 2, 14]]

[[ 6  2  0  0]
 [ 3  1  0  0]
 [ 7  4  0  0]
 [ 8  1  0  0]
 [ 9  0  0  0]
 [10  0  0  0]
 [ 5  4  0  0]
 [11  3  0  0]
 [ 5  1  0  0]
 [12 13  2 14]]

Loaded 400000 word vectors.

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 4, 100)            1500
_________________________________________________________________
flatten_1 (Flatten)          (None, 400)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 401
=================================================================
Total params: 1,901
Trainable params: 401
```

```
Non-trainable params: 1,500
_________________________________________________________________


Accuracy: 100.000000
```

In practice, I would encourage you to experiment with learning a word embedding using a pre-trained embedding that is fixed and trying to perform learning on top of a pre-trained embedding.

See what works best for your specific problem.

Transformer

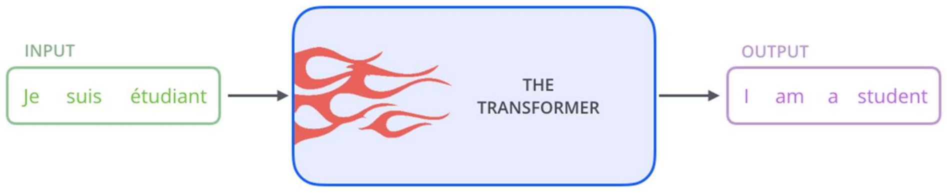
The Illustrated Transformer

The Transformer – a model that uses attention to boost the speed with which these models can be trained. The Transformer outperforms the Google Neural Machine Translation model in specific tasks. The biggest benefit, however, comes from how The Transformer lends itself to parallelization. It is in fact Google Cloud's recommendation to use The Transformer as a reference model to use their Cloud TPU offering. So let's try to break the model apart and look at how it functions.

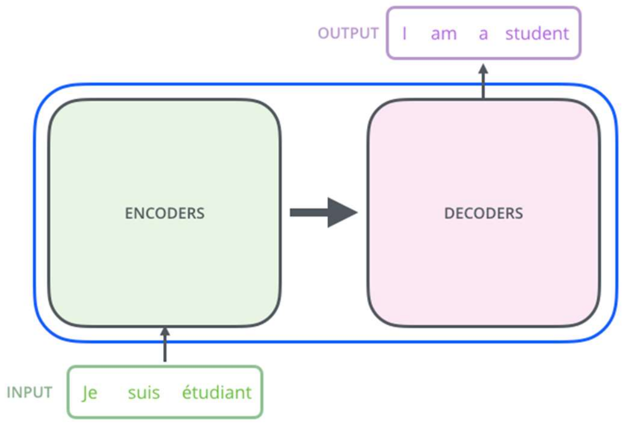
The Transformer was proposed in the paper Attention is All You Need. A TensorFlow implementation of it is available as a part of the Tensor2Tensor package. Harvard's NLP group created a guide annotating the paper with PyTorch implementation. In this post, we will attempt to oversimplify things a bit and introduce the concepts one by one to hopefully make it easier to understand to people without in-depth knowledge of the subject matter.

A High-Level Look

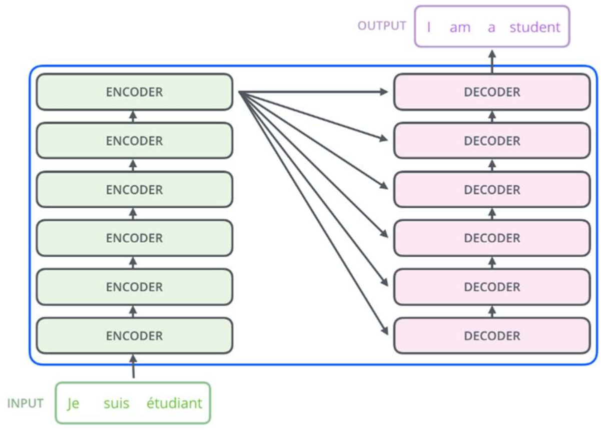
Let's begin by looking at the model as a single black box. In a machine translation application, it would take a sentence in one language, and output its translation in another.



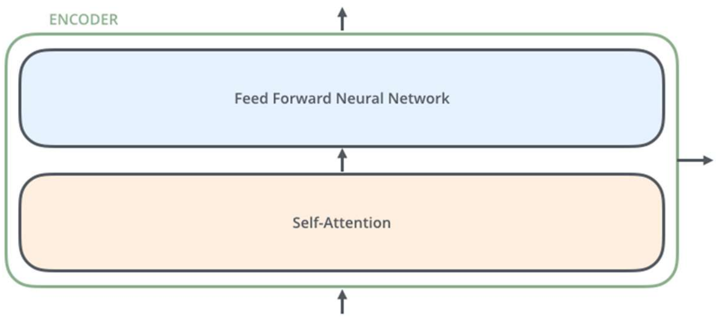
Popping open that Optimus Prime goodness, we see an encoding component, a decoding component, and connections between them.



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

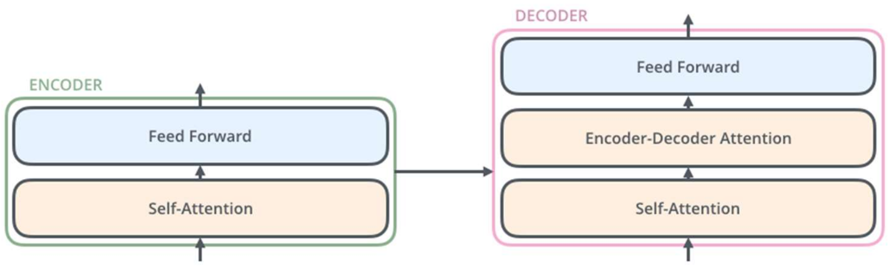


The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:



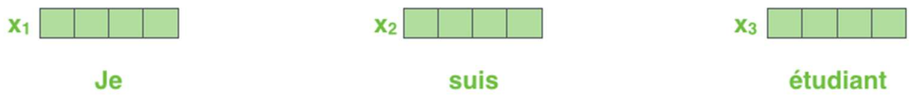
The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. We'll look closer at self-attention later in the post.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.
The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in seq2seq models).



Bringing The Tensors Into The Picture

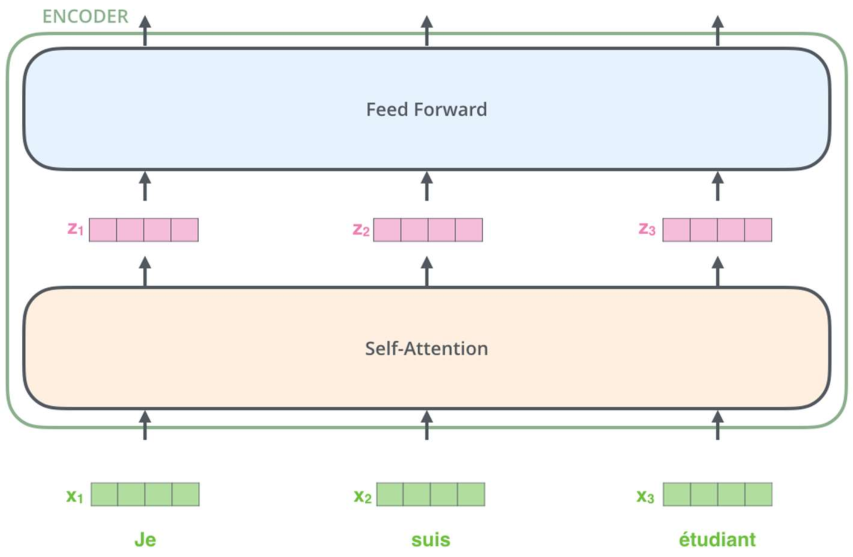
Now that we've seen the major components of the model, let's start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output.
As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm.



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

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.

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

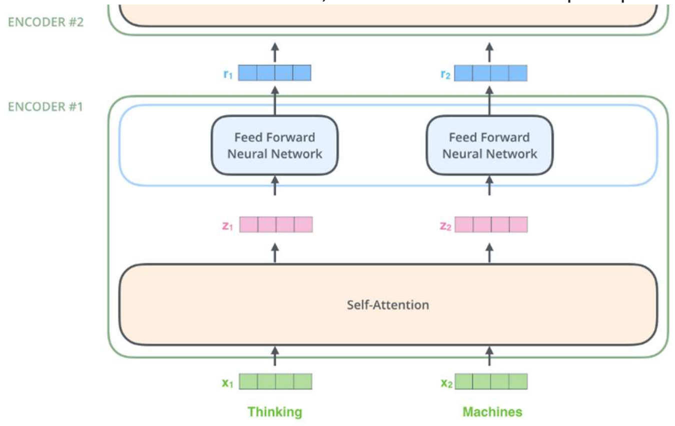


Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer. The feed-forward layer does not have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.

Next, we'll switch up the example to a shorter sentence and we'll look at what happens in each sub-layer of the encoder.

Now We're Encoding!

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



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

Self-Attention at a High Level

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

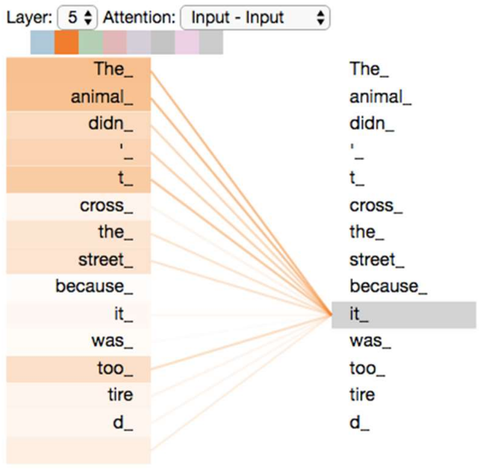
Say the following sentence is an input sentence we want to translate:
"The animal didn't cross the street because it was too tired"

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

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

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

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



As we are encoding the word "it" in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of "it".

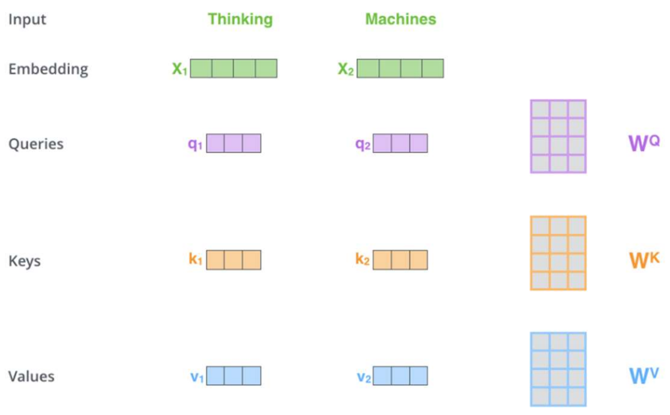
Be sure to check out the Tensor2Tensor notebook where you can load a Transformer model, and examine it using this interactive visualization.

Self-Attention in Detail

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

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

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

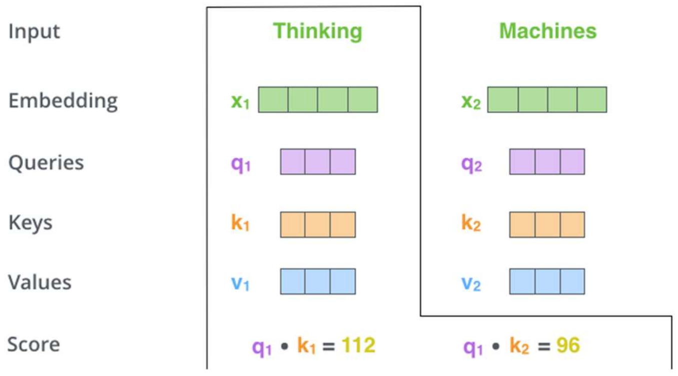


Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

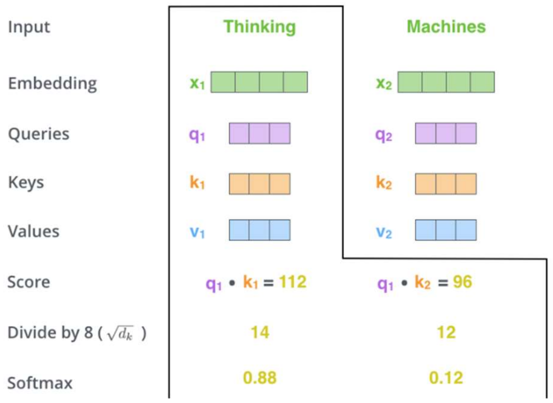
What are the "query", "key", and "value" vectors?

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

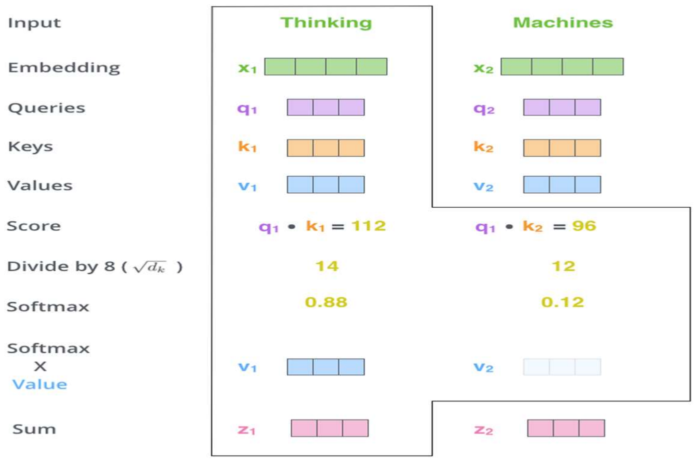
The **second step** in calculating self-attention is to calculate a score. Say we're calculating the self-attention for the first word in this example, "Thinking". We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.
The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring. So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.



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



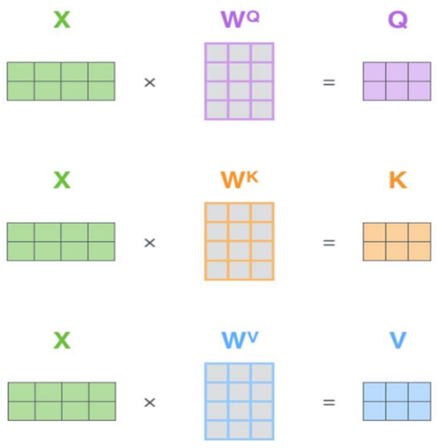
This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.
The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).
The **sixth step** is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).



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

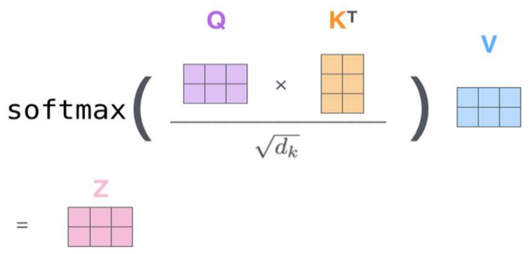
Matrix Calculation of Self-Attention

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



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

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



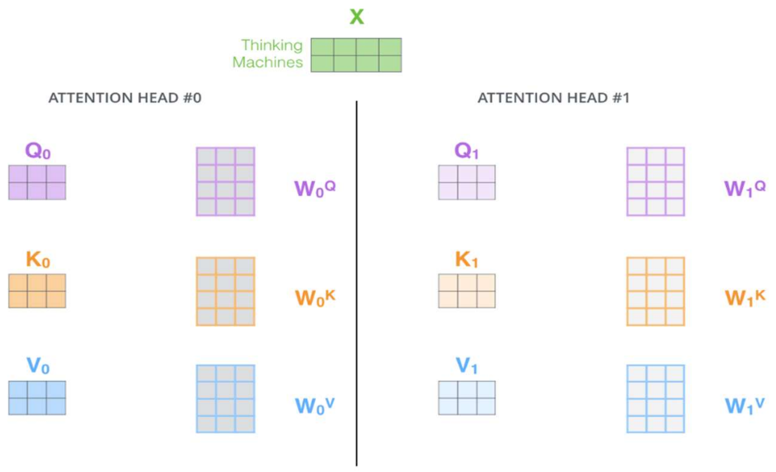
The self-attention calculation in matrix form

The Beast With Many Heads

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

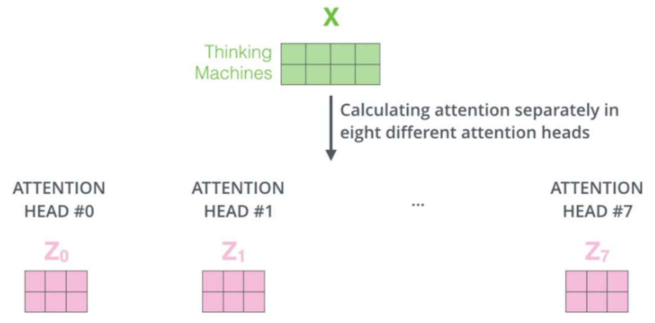
1. It expands the model's ability to focus on different positions. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the actual word itself. If we're translating a sentence like "The animal didn't cross the street because it was too tired", it would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



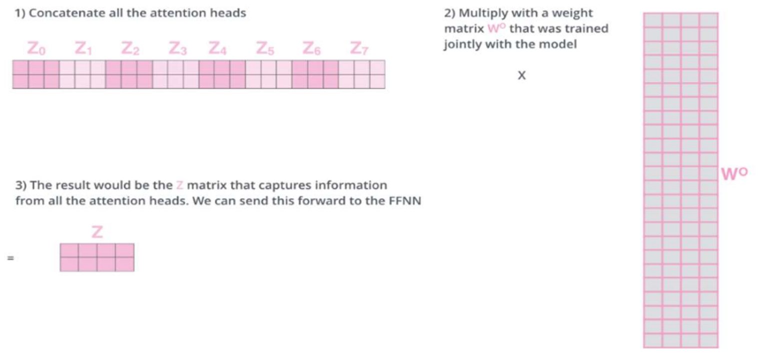
With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

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

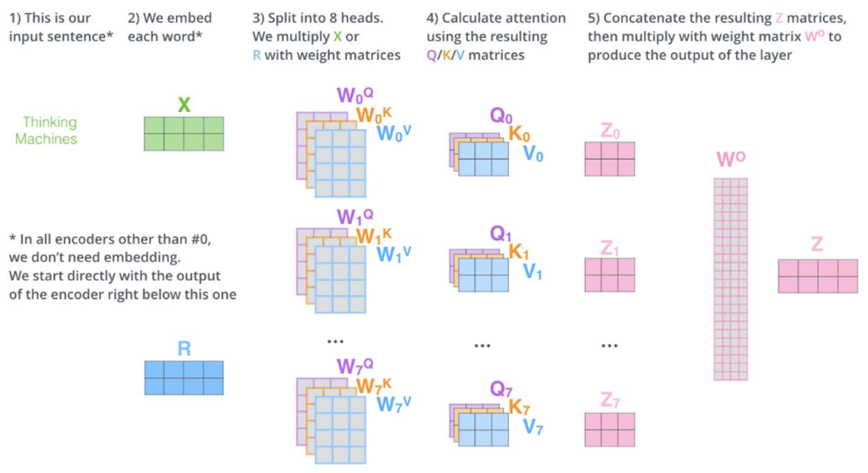


This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.

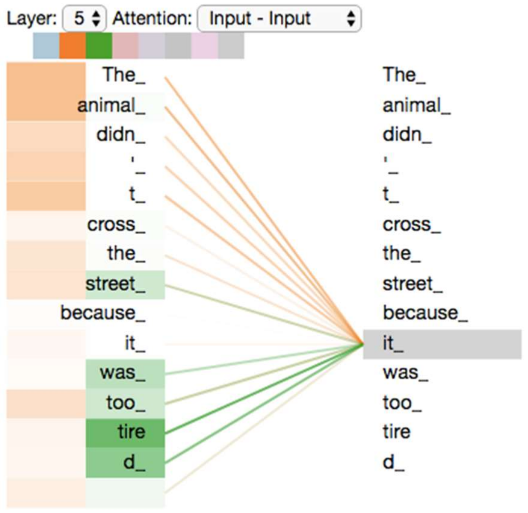
How do we do that? We concat the matrices then multiply them by an additional weights matrix WO.


1) Concatenate all the attention heads
Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7
2) Multiply with a weight matrix WO that was trained jointly with the model
x
WO
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN
=
Z

That's pretty much all there is to multi-headed self-attention. 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

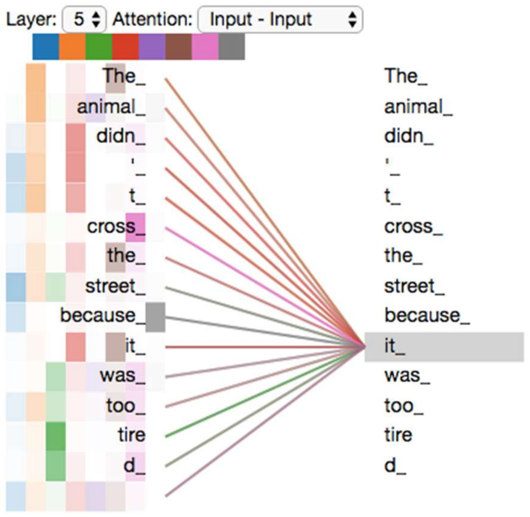

1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer
Thinking Machines
X
R
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
W0Q W0K W0V
W1Q W1K W1V
W7Q W7K W7V
Q0 K0 V0
Q1 K1 V1
Q7 K7 V7
Z0
Z1
Z7
WO
Z

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



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

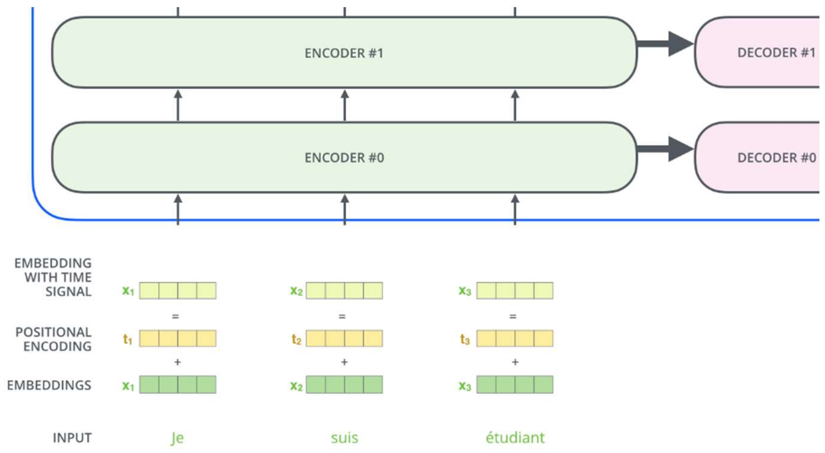
If we add all the attention heads to the picture, however, things can be harder to interpret:



Representing The Order of The Sequence Using Positional Encoding

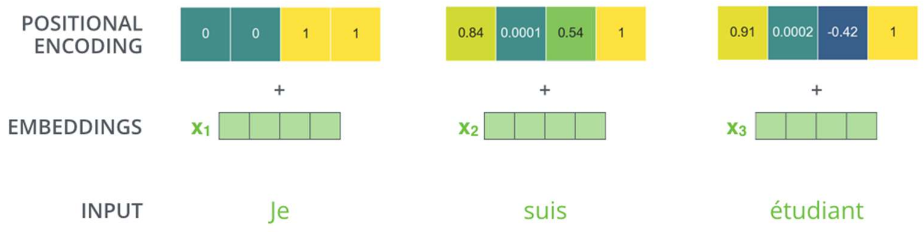
One thing that's missing from the model as we have described it so far is a way to account for the order of the words in the input sequence.

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



To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.

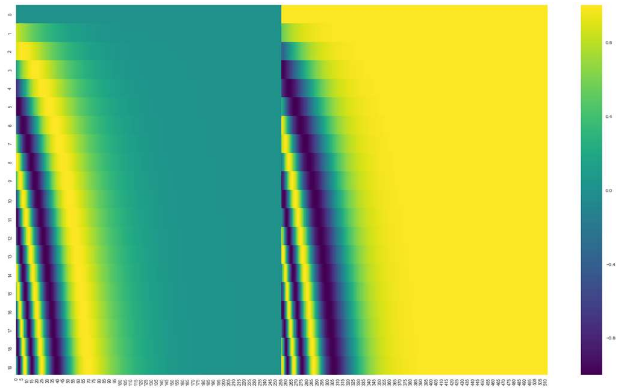
If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like this:



A real example of positional encoding with a toy embedding size of 4

What might this pattern look like?

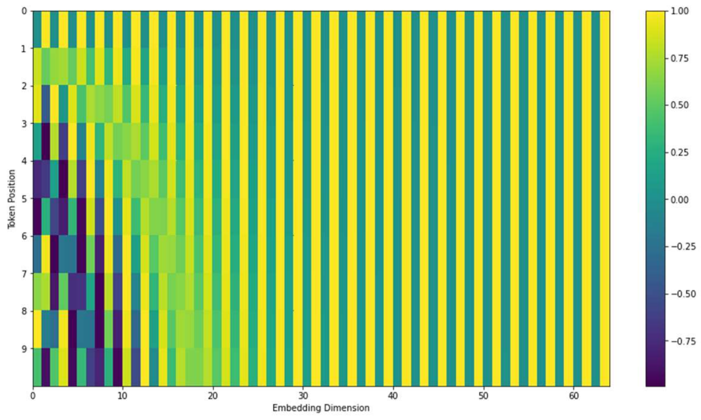
In the following figure, each row corresponds to a positional encoding of a vector. So the first row would be the vector we'd add to the embedding of the first word in an input sequence. Each row contains 512 values – each with a value between 1 and -1. We've color-coded them so the pattern is visible.



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

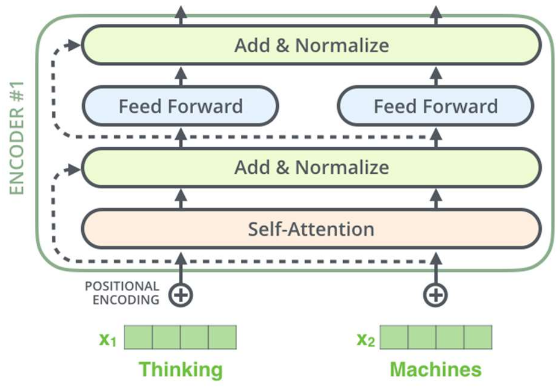
The formula for positional encoding is described in the paper (section 3.5). You can see the code for generating positional encodings in get_timing_signal_1d(). This is not the only possible method for positional encoding. It, however, gives the advantage of being able to scale to unseen lengths of sequences (e.g. if our trained model is asked to translate a sentence longer than any of those in our training set).

July 2020 Update: The positional encoding shown above is from the Tensor2Tensor implementation of the Transformer. The method shown in the paper is slightly different in that it doesn't directly concatenate, but interweaves the two signals. The following figure shows what that looks like. Here's the code to generate it:

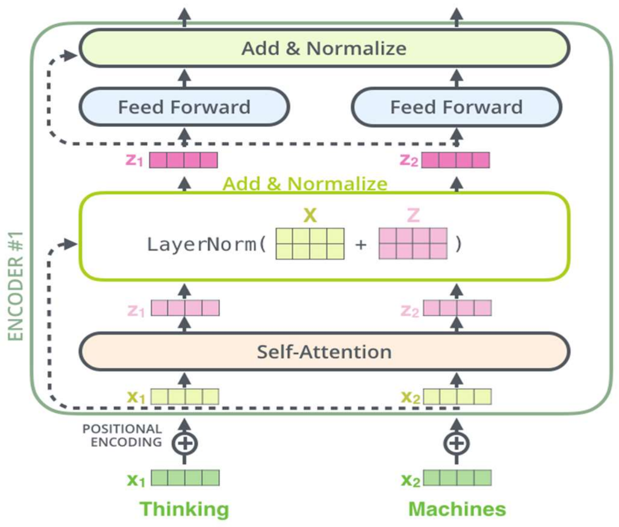


The Residuals

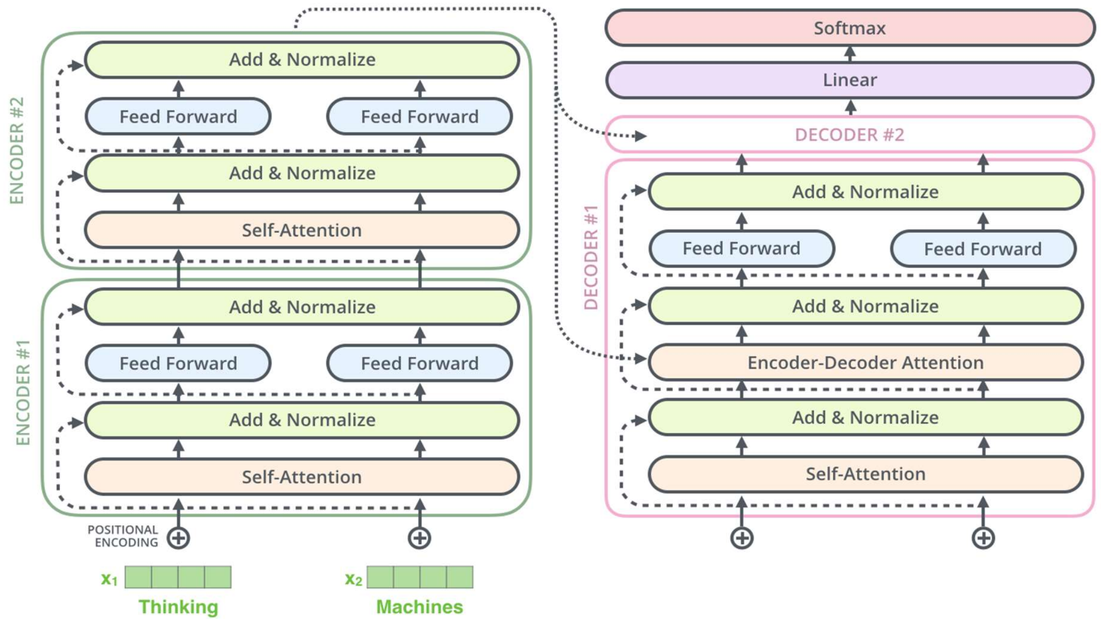
One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization step.



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



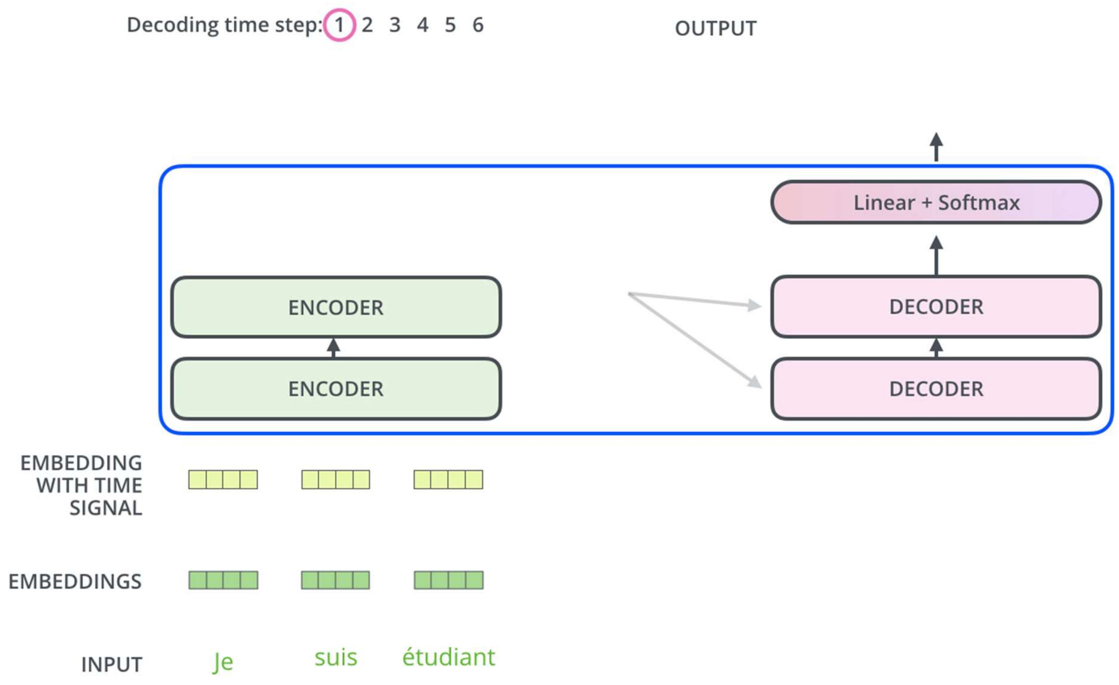
This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:



The Decoder Side

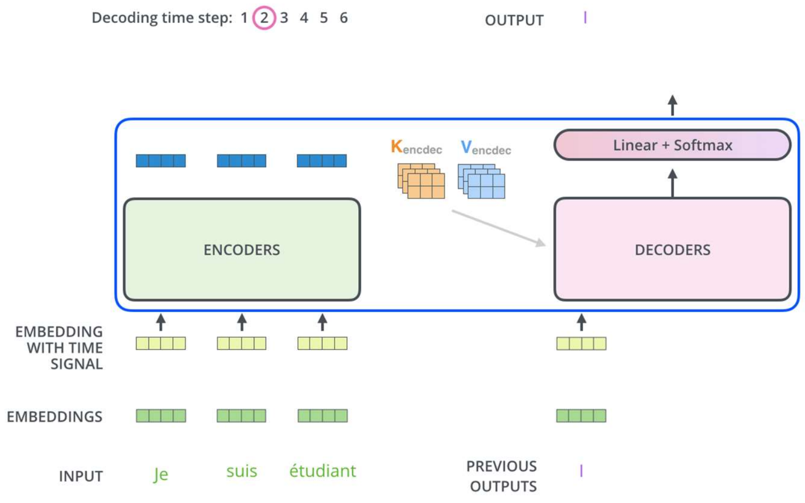
Now that we've covered most of the concepts on the encoder side, we basically know how the components of decoders work as well. But let's take a look at how they work together.

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



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

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



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.

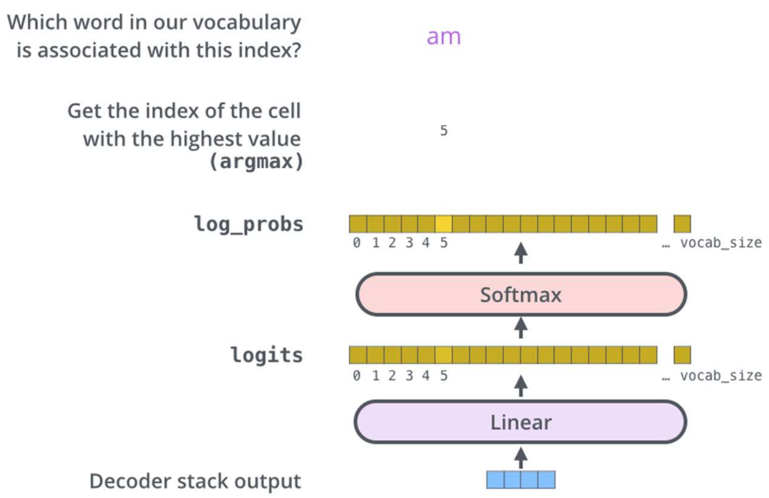
The Final Linear and Softmax Layer

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.

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

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

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


Which word in our vocabulary is associated with this index?
am
Get the index of the cell with the highest value (argmax)
5
log_probs
0 1 2 3 4 5
… vocab_size
Softmax
logits
0 1 2 3 4 5
… vocab_size
Linear
Decoder stack output

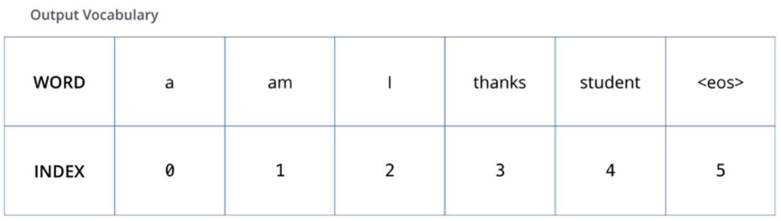
This figure starts from the bottom with the vector produced as the output of the decoder stack. It is then turned into an output word.

Recap Of Training

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

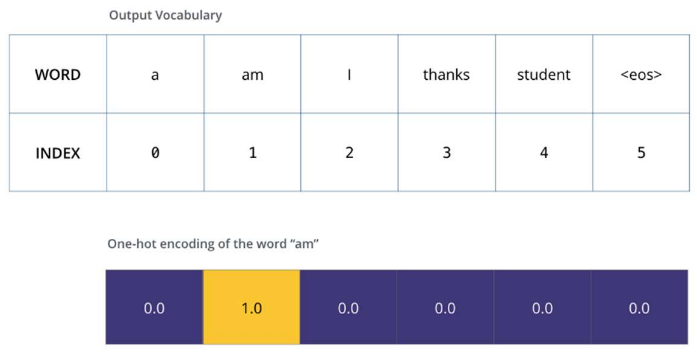
During training, an untrained model would go through the exact same forward pass. But since we are training it on a labeled training dataset, we can compare its output with the actual correct output.

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


Output Vocabulary

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

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

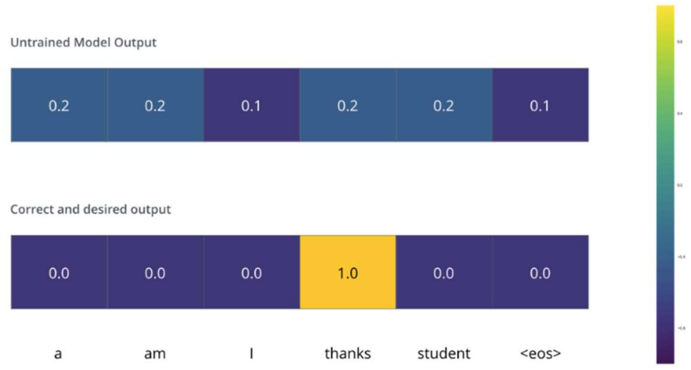
Example: one-hot encoding of our output vocabulary

Following this recap, let's discuss the model's loss function – the metric we are optimizing during the training phase to lead up to a trained and hopefully amazingly accurate model.

The Loss Function

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

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


Untrained Model Output

a	am	I	thanks	student	<eos>
0.2	0.2	0.1	0.2	0.2	0.1

Correct and desired output

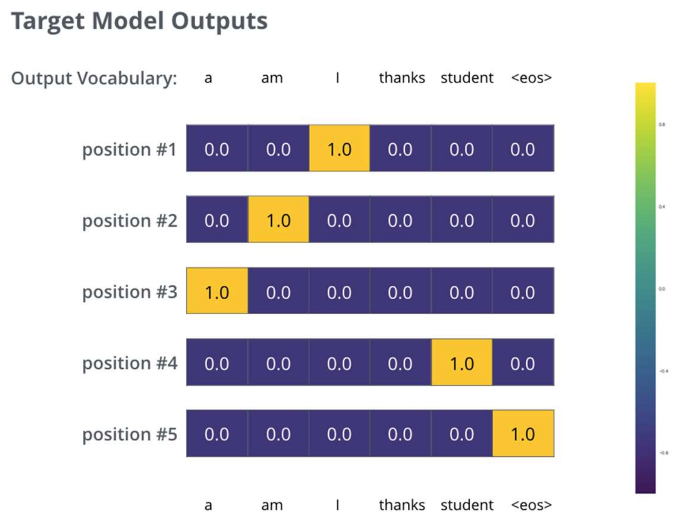
a	am	I	thanks	student	<eos>
0.0	0.0	0.0	1.0	0.0	0.0

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.

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

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

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

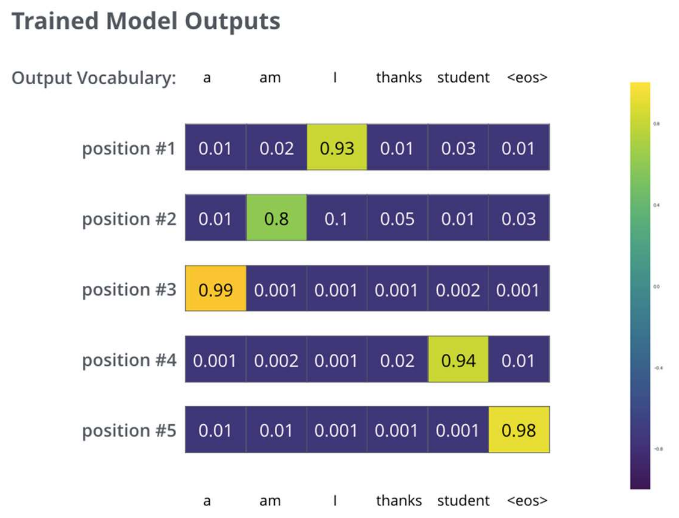


Target Model Outputs

Output Vocabulary:	a	am	I	thanks	student	<eos>
position #1	0.0	0.0	1.0	0.0	0.0	0.0
position #2	0.0	1.0	0.0	0.0	0.0	0.0
position #3	1.0	0.0	0.0	0.0	0.0	0.0
position #4	0.0	0.0	0.0	0.0	1.0	0.0
position #5	0.0	0.0	0.0	0.0	0.0	1.0
	a	am	I	thanks	student	<eos>

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

After training the model for enough time on a large enough dataset, we would hope the produced probability distributions would look like this:



Trained Model Outputs

Output Vocabulary:	a	am	I	thanks	student	<eos>
position #1	0.01	0.02	0.93	0.01	0.03	0.01
position #2	0.01	0.8	0.1	0.05	0.01	0.03
position #3	0.99	0.001	0.001	0.001	0.002	0.001
position #4	0.001	0.002	0.001	0.02	0.94	0.01
position #5	0.01	0.01	0.001	0.001	0.001	0.98
	a	am	I	thanks	student	<eos>

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

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