NLP Roadmap

shbahmanyar98@gmail.com

Step1) Libraries

spaCy, Core NLP, Text Blob, PyNLPI, Gensim, Pattern, NLTK

1. SpaCy

spaCy is a free, open-source library for advanced Natural Language Processing (NLP) in Python. It will be used to build information extraction, natural language understanding systems, and to pre-process text for deep learning. Spacy is used in NLP projects, such as Tokenization, Lemmatisation, Part-of-speech(POS) tagging, Entity recognition, Dependency parsing, Sentence recognition, Word-to-vector transformations, and other cleaning and normalization text methods.

2. CoreNLP

Stanford CoreNLP provides a set of natural language analysis tools that can give the base forms of words, their parts of speech, whether they are names of companies, people, etc. CoreNLP enables users to derive linguistic annotations for text, including token and sentence boundaries, parts of speech, named entities, numeric and time values, dependency and constituency parses, coreference, sentiment, quote attributions, and relations. People use CoreNLP while writing their own code in Javascript, Python, or some other language.

3.TextBlob

TextBlob is a python library and offers a simple API to access its methods and perform basic NLP tasks. With TextBlob, you spend less time struggling with the intricacies of Pattern and NLTK and more time getting results. TextBlob smooths the way by leveraging native Python objects and syntax. TextBlob, which is built on the shoulders of NLTK and Pattern. A big advantage of this is, it is easy to learn and offers a lot of features like sentiment analysis, pos-tagging, noun phrase extraction, etc.

4. PyNLPI

PyNLPI is a Python library for Natural Language Processing that contains various modules useful for common, and less common, NLP tasks. PyNLPI can be used for basic tasks such as the extraction of n-grams and frequency lists, and to build simple language model. PyNLPI can read and process GIZA, Moses++, SoNaR, Taggerdata, and TiMBL data formats, and devotes an entire module to working with FoLiA, the XML document format used to annotate language resources like corpora.

5. Gensim

Gensim is a free open-source Python library for representing documents as semantic vectors, as efficiently (computer-wise) and painlessly (human-wise) as possible. Gensim is implemented in Python and Cython for performance. Gensim is designed to handle large text collections using data streaming and incremental online algorithms, which differentiates it from most other machine learning software packages that target only in-memory processing. Gensim depends on scipy and numpy. You must have them installed prior to installing gensim.

6. Pattern

Pattern is an open-source python library and performs different NLP tasks. Pattern Library is used for NLP by performing tasks such as tokenization, stemming and sentiment analysis. We will also see how the Pattern library can be used for web mining. Pattern comes with built-ins for scraping a number of popular web services and sources (Google, Wikipedia, Twitter, Facebook, generic RSS, etc.), all of which are available as Python modules. Pattern exposes some of its lower-level functionality, allowing you to to use NLP functions, n-gram search, vectors, and graphs directly if you like.

7. NLTK

NLTK is a leading platform for building Python programs to work with human language data. NLTK is available for Windows, Mac OS X, and Linux. Best of all, NLTK is a free, open source, community-driven project. It has text processing libraries for classification, tokenization, stemming, POS tagging, parsing, and semantic reasoning. NLTK is a Python package that you can use for NLP.

Step2) Text Preprocessing Level-1

In NLP, we have the text data, which our Machine Learning algorithms cannot directly use, so we have first to preprocess it and then feed the preprocessed data to our Machine Learning algorithms. So, In this step, we will try to learn the same basic processing steps which we have to perform in almost every NLP problem.

1) Tokenization,

2) Lemmatization,

3) Stemming,

4) Parts of Speech (POS),

5) Stopwords removal,

6) Punctuation removal, etc.

1. Tokenization

Splitting the text into individual words or subwords (tokens).

Here is how to implement tokenization in NLTK:

```
import nltk

# input text
text = "Natural language processing is a field of artificial intelligence that deals with the interaction between computers and human (natural) language."

# tokenize the text
tokens = nltk.word_tokenize(text)

print("Tokens:", tokens)
```

This will output the following list of tokens:

```
['Natural', 'language', 'processing', 'is', 'a', 'field', 'of', 'artificial', 'intelligence',
'that', 'deals', 'with', 'the', 'interaction', 'between', 'computers', 'and', 'human',
'(', 'natural', ')', 'language', '.']
```

The nltk.word_tokenize() function uses the Punkt tokenization algorithm, which is a widely used method for tokenizing text in multiple languages. You can also use other tokenization methods, such as splitting the text on whitespace or punctuation, but these may not be as reliable for handling complex text structures and languages.

2. Lemmatization

A more complicated and accurate method of reducing words to their base form than stemming.

To perform lemmatization on a list of tokens using NLTK, you can use the nltk.stem.WordNetLemmatizer() function to create a lemmatizer object and the lemmatize() method to lemmatize each token. Here is an example of how to do this:

```
import nltk


# input text

text = "Natural language processing is a field of artificial intelligence that deals with the interaction between computers and human (natural) language."
```

```
# tokenize the text

tokens = nltk.word_tokenize(text)


# create lemmatizer object

lemmatizer = nltk.stem.WordNetLemmatizer()


# lemmatize each token

lemmatized_tokens = [lemmatizer.lemmatize(token) for token in tokens]



print("Lemmatized tokens:", lemmatized_tokens)
```

This will output the following list of lemmatized tokens:

```
Lemmatized tokens: ['Natural', 'language', 'processing', 'is', 'a', 'field', 'of', 'artificial', 'intelligence', 'that', 'deals', 'with', 'the', 'interaction', 'between', 'computer', 'and', 'human', '(', 'natural', ')', 'language', '.']
```

The WordNet lemmatizer uses the WordNet database of English words to lemmatize the tokens, taking into account the part of speech and the context in which the word is used. You can specify the part of speech of the token using the pos argument of the lemmatize() method (e.g., "n" for nouns, "v" for verbs, etc.).

3. Stemming

Reducing words to their base form, such as converting “jumping” to “jump.”

To perform stemming on a list of tokens using NLTK, you can use the nltk.stem.PorterStemmer() function to create a stemmer object and the stem() method to stem each token. Here is an example of how to do this:

```
import nltk

# input text

text = "Natural language processing is a field of artificial intelligence that deals with the interaction between computers and human (natural) language."


# tokenize the text

tokens = nltk.word_tokenize(text)


# create stemmer object

stemmer = nltk.stem.PorterStemmer()


# stem each token

stemmed_tokens = [stemmer.stem(token) for token in tokens]
```

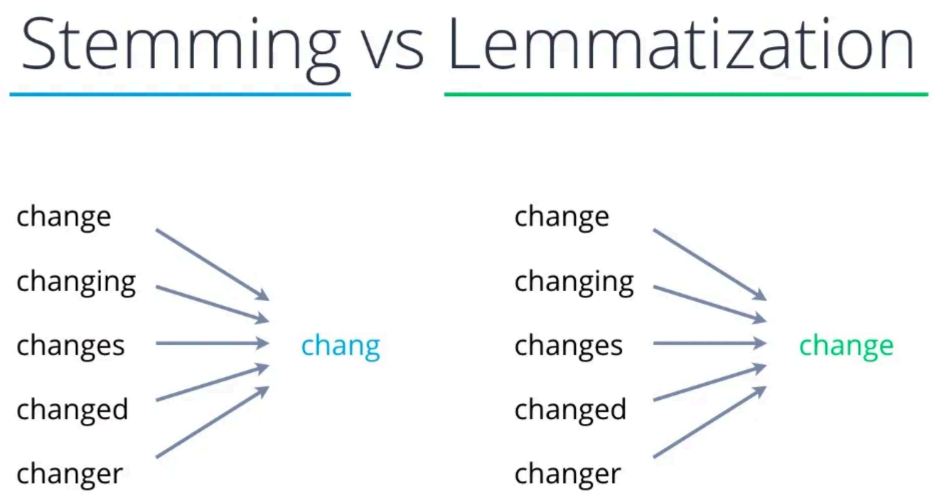
print("Stemmed tokens:", stemmed_tokens)

This will output the following list of stemmed tokens:

Stemmed tokens: ['natur', 'languag', 'process', 'is', 'a', 'field', 'of', 'artifici', 'intellig', 'that', 'deal', 'with', 'the', 'interact', 'between', 'comput', 'and', 'human', '(', 'natur', ')', 'languag', '.']

The Porter stemmer is a widely used algorithm that removes common morphological affixes from words in order to obtain their base form or root. Other stemmers are also available in the nltk library, such as the Snowball stemmer, which supports multiple languages.

The difference between Stemmning and Lemmatization can be shown in the following figure:



4. Part-of-speech tagging

Identifying the part of speech of each word in the text, such as noun, verb, or adjective.

To perform part of speech (POS) tagging on a list of tokens using NLTK, you can use the nltk.pos_tag() function to tag the tokens with their corresponding POS tags. Here is an example of how to do this:

```
import nltk

# input text

text = "Natural language processing is a field of artificial intelligence that deals with the interaction between computers and human (natural) language."


# tokenize the text

tokens = nltk.word_tokenize(text)


# tag the tokens with their POS tags

tagged_tokens = nltk.pos_tag(tokens)


print("Tagged tokens:", tagged_tokens)
```

This will output the following list of tuples with the tokens and their corresponding POS tags:

```
Tagged tokens: [('Natural', 'NNP'), ('language', 'NN'), ('processing', 'NN'), ('is', 'VBZ'), ('a', 'DT'), ('field', 'NN'), ('of', 'IN'), ('artificial', 'JJ'), ('intelligence', 'NN'), ('that', 'WDT'), ('deals', 'NNS'), ('with', 'IN'), ('the', 'DT'), ('interaction', 'NN'), ('between', 'IN'), ('computers', 'NNS'), ('and', 'CC'), ('human', 'NNS'), ('natural', 'NNP'), ('language', 'NN')]
```

5. Remove stop words

Removing common words that do not add significant meaning to the text, such as “a,” “an,” and “the.”

To remove common stop words from a list of tokens using NLTK, you can use the nltk.corpus.stopwords.words() function to get a list of stopwords in a specific language and filter the tokens using this list. Here is an example of how to do this:

```
import nltk


# input text

text = "Natural language processing is a field of artificial intelligence that deals with the interaction between computers and human (natural) language."
```

```
# tokenize the text

tokens = nltk.word_tokenize(text)


# get list of stopwords in English

stopwords = nltk.corpus.stopwords.words("english")


# remove stopwords

filtered_tokens = [token for token in tokens if token.lower() not in stopwords]


print("Tokens without stopwords:", filtered_tokens)
```

This will output the following list of tokens without stopwords:

```
Tokens without stopwords: ['Natural', 'language', 'processing', 'field', 'artificial', 'intelligence', 'deals', 'interaction', 'computers', 'human', '(', 'natural', ')', 'language', '.']
```

6. Remove punctuation

Removing punctuation marks simplifies the text and make it easier to process.

To remove punctuation from a list of tokens using NLTK, you can use the string module to check if each token is a punctuation character. Here is an example of how to do this:

```
import nltk
import string

# input text
text = "Natural language processing is a field of artificial intelligence that deals with the interaction between computers and human (natural) language."

# tokenize the text
tokens = nltk.word_tokenize(text)

# remove punctuation
filtered_tokens = [token for token in tokens if token not in string.punctuation]

print("Tokens without punctuation:", filtered_tokens)
```

This will output the following list of tokens without punctuation:

```
Tokens without punctuation: ['Natural', 'language', 'processing', 'is', 'a', 'field', 'of', 'artificial', 'intelligence', 'that', 'deals', 'with', 'the', 'interaction', 'between', 'computers', 'and', 'human', 'natural', 'language']
```

Step3) Text preprocessing Level-2

All these are the primary methods to convert our Text data into numerical data (Vectors) to apply a Machine Learning algorithm to it.

1) Bag of words (BOW),

2) Term frequency Inverse Document Frequency (TFIDF),

3) Unigram, Bigram, and Ngrams.

1. Bag of words (BOW)

We will consider few reviews to explain BOW and how the BOW matrix is created.

review1 (r1): this coffee is very bad and is expensive

review1 (r2): this coffee is not bad and is cheap

review1 (r3): this coffee is not amazing and is not affordable lets clarify few more terms before moving further:

Document: a document is a text item. In this case it is each single review r1, r2 and r3

Corpus: Corpus is the collection of all the documents

Dictionary: a set of all unique words in all the documents

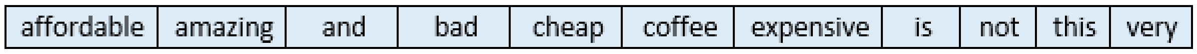
Sparse Vector: A vector having most of the dimensions in it with a 0 value

Dense Vector: Opposite of Sparse, having more non-zero values than 0 values

Steps in BOW:

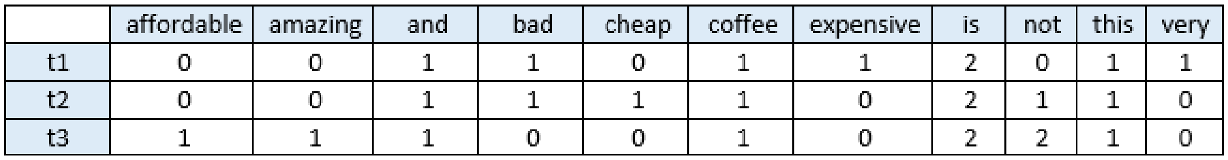
Step 1) Construct a dictionary with all the unique words in all the documents.

Thus, the dictionary constructed out of the above three reviews r1, r2 and r3 will be:



affordable	amazing	and	bad	cheap	coffee	expensive	is	not	this	very

Step 2) Construct a vector for every text by counting number of time each word in each text occurred out of the dictionary



	affordable	amazing	and	bad	cheap	coffee	expensive	is	not	this	very
t1	0	0	1	1	0	1	1	2	0	1	1
t2	0	0	1	1	1	1	0	2	1	1	0
t3	1	1	1	0	0	1	0	2	2	1	0

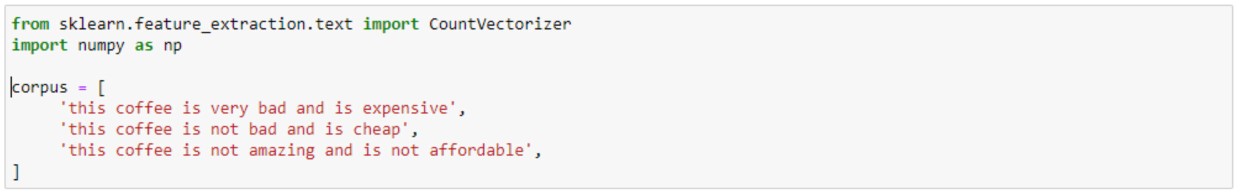
The resultant Bag of Words matrix above, will always be a Sparse Vector.

We can use the above Sparse matrix to either calculate various types of distances between vectors/points or for other feature engineering techniques.

Implementing BOW in Python:

BOW is created from a text corpus using Scikit Learn's CountVectorizer function

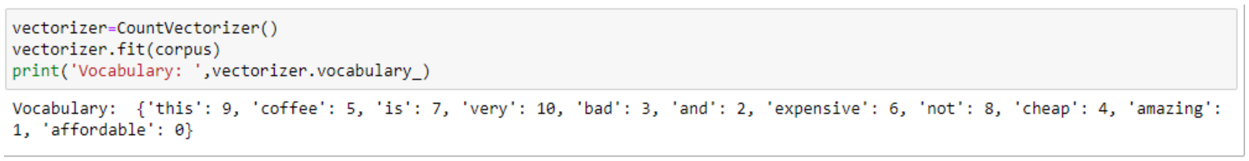
Step1) Import packages and corpus



```
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np

corpus = [
     'this coffee is very bad and is expensive',
     'this coffee is not bad and is cheap',
     'this coffee is not amazing and is not affordable',
]
```

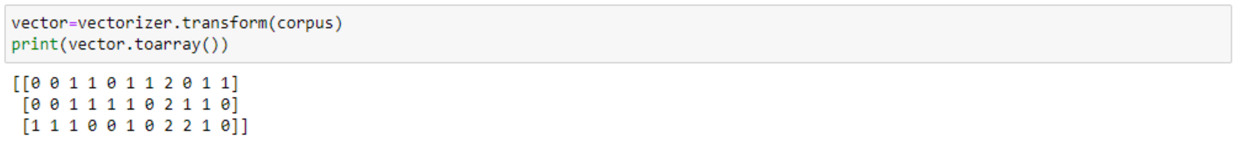
Step2) Load CountVectorizer() function and check the corpus by using "vectorizer.vocabulary_)



```
vectorizer=CountVectorizer()
vectorizer.fit(corpus)
print('Vocabulary: ',vectorizer.vocabulary_)
```

```
Vocabulary:  {'this': 9, 'coffee': 5, 'is': 7, 'very': 10, 'bad': 3, 'and': 2, 'expensive': 6, 'not': 8, 'cheap': 4, 'amazing':
1, 'affordable': 0}
```

Step3) Transform all docs in the corpus and check the array



```
vector=vectorizer.transform(corpus)
print(vector.toarray())
```

```
[[0 0 1 1 0 1 1 2 0 1 1]
 [0 0 1 1 1 1 0 2 1 1 0]
 [1 1 1 0 0 1 0 2 2 1 0]]
```

2. Term frequency Inverse Document Frequency (TFIDF)

You will see that the output array from CountVectorizer() matches with the one calculated manually.

TF-IDF is a numerical statistic that indicates **how important a word is** to a document in a corpus.

First lets understand the two terms separately.

Term Frequency (TF)

Term Frequency of a word indicates the frequency of occurrence of a word in a document.

Lets assume a corpus having words (w_i) and reviews (r_j) and “N” be the total documents.

Then TF of a word word “w_i” in review “r_j” is denoted by”

$$TF\,(w_i,\, r_j) = \frac{\textit{Number of times "}w_i\textit{" occurs in "}r_j\textit{"}}{\textit{Total words in "}r_j\textit{"}}$$

For Example:

Let us assume to have below reviews:

r1: w1 w2 w3 w2 w4

r2: w1 w3 w2 w5 w6 w5

TF (w2, r1) = 2/5= 0.4

TF (w3, r2)= 1/6=0.166

*The term frequency of any word in general lies in between 0 and 1 (inclusive). Thus, we can interpret it as probability and hence TF(w_i , r□) can also be called as probability of occurrence of the word ‘w_i ’ in ‘r□ ’.

Inverse Document Frequency (IDF)

Inverse Document Frequency(IDF)of a word indicates the frequency of occurrence of a word across the documents (corpus “D”)

Thus the IDF of a word “w_i” in the corpus (D) having “N” documents is denoted by:

$$IDF\ (w_i, D) = \log_e \frac{\text{Total documents "N"}}{\text{Document in which "}w_i\text{" occurs}}$$

Here the denominator is also denoted by “n_i” (total documents in which w_i occurs)

Calculating TF-IDF:

The value of TF-IDF for a word is the product of its TF and the IDF values.

$$TF\text{-}IDF = TF * IDF$$

$$= \frac{\text{Number of times "}w_i\text{" occurs in "}r_j\text{"}}{\text{Total words in "}r_j\text{"}} \text{ X } \log_e \frac{\text{Total documents "N"}}{\text{Document in which "}w_i\text{" occurs}}$$

Note:

If “n_i” decreases, (N/n_i) decreases and ultimately $\log_e(N/n_i)$ will also decreases

Since “n_i” depends on “w_i”, the more the occurrence of “w_i” across the reviews in the corpus, the lesser is the IDF score, which means:

If **“w_i” is more frequent, IDF(w_i ,D) will be low**

If **“w_i” is rare, IDF(w_i ,D) will be more**

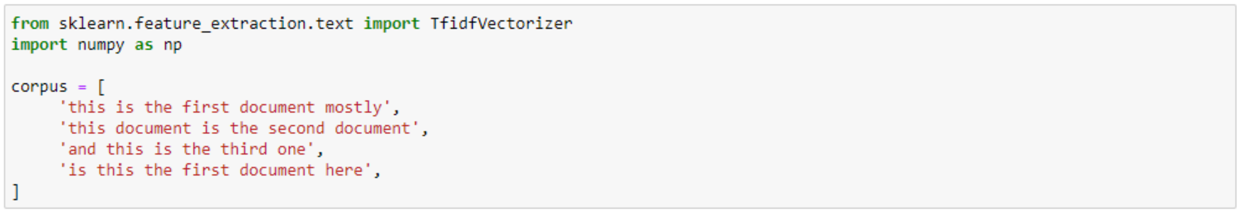
How TF-IDF is different from BOW?

Although in both BOW and TF-IDF we convert each review text into a d-dimensional vector, the key difference between the two is that BOW creates a set of vectors containing the count of the word occurrences in the document (reviews), while the TF-IDF model contains information about the word like, which words are more important and which words are less important.

Implementing TF-IDF in Python

TF-IDF matrix is created from a text corpus using Scikit Learn’s TfidfTransformer function.

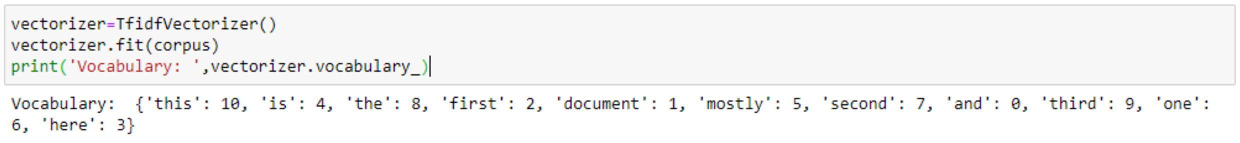
Step1) Import packages and corpus



```python
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np

corpus = [
     'this is the first document mostly',
     'this document is the second document',
     'and this is the third one',
     'is this the first document here',
]
```

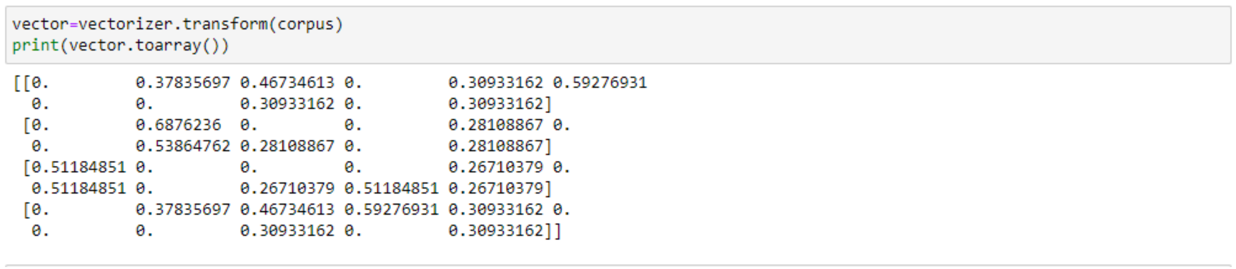
Step2) Load TfidfTransformer() function and check the corpus by using "vectorizer.vocabulary_"



```python
vectorizer=TfidfVectorizer()
vectorizer.fit(corpus)
print('Vocabulary: ',vectorizer.vocabulary_)
```

```
Vocabulary:  {'this': 10, 'is': 4, 'the': 8, 'first': 2, 'document': 1, 'mostly': 5, 'second': 7, 'and': 0, 'third': 9, 'one':
6, 'here': 3}
```

Step3) Transform all docs in the corpus and check the array



```python
vector=vectorizer.transform(corpus)
print(vector.toarray())
```

```
[[0.         0.37835697 0.46734613 0.         0.30933162 0.59276931
  0.         0.         0.30933162 0.         0.30933162]
 [0.         0.6876236  0.         0.         0.28108867 0.
  0.         0.53864762 0.28108867 0.         0.28108867]
 [0.51184851 0.         0.         0.         0.26710379 0.
  0.51184851 0.         0.26710379 0.51184851 0.26710379]
 [0.         0.37835697 0.46734613 0.59276931 0.30933162 0.
  0.         0.         0.30933162 0.         0.30933162]]
```

3. Unigram, Bigram, and Ngrams

These techniques involve considering single words (unigrams), pairs of consecutive words (bigrams), or groups of N consecutive words (N-grams) as features for vectorization. They capture different levels of context and can be useful for various NLP tasks.

Step 4) Text preprocessing Level-3

All these are advanced techniques to convert words into vectors.

1) Word2vec,

2) GloVe,

3) FastText

What are Word Embeddings in NLP?

Word embeddings form the backbone of many NLP applications by representing words as continuous vectors in a high-dimensional space. These embeddings capture semantic relationships between words, enabling algorithms to process and understand language more effectively.

fastText, like its predecessors, excels in generating these embeddings but stands out due to its approach at the subword level.

1. Word2Vec

A statistical technique called Word2Vec can effectively learn a standalone word embedding from a text corpus. It was created by Tomas Mikolov and colleagues at Google in 2013 to improve the effectiveness of embedding training using neural networks. It has since taken over as the industry norm. The work also included investigating how vector math applied to word representations and analysing the learned vectors.

A typical example used to explain word vectors is the phrase, “the king is to the queen as a man is to a woman.” If we take the male gender out of the word “king” and add the female gender, we would arrive at the word “queen.” In this way, we can start to reason with words through the relationships that they hold in regard to other words.

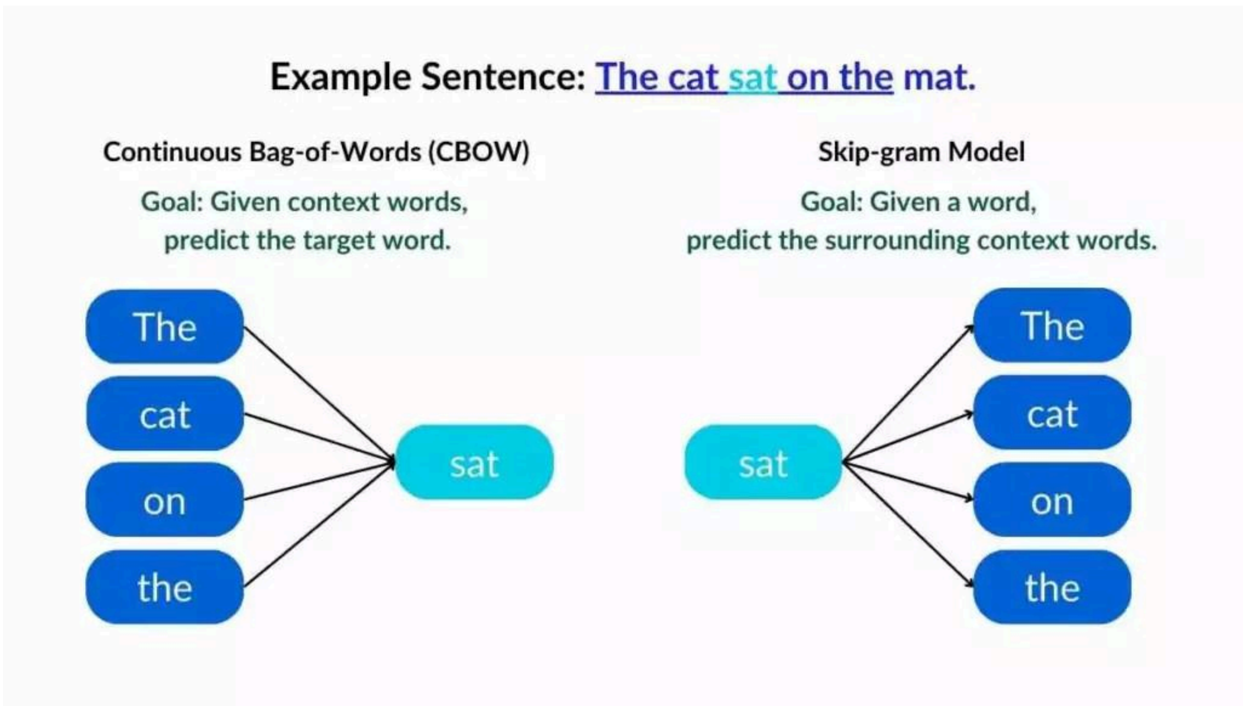
“The king is to the queen as a man is to a woman.” – Word2Vec understands the context.

Two new learning models were presented to learn the word embedding using the word2vec method.

- Continuous Bag-of-Words (CBOW) model
- Continuous Skip-Gram Model

The CBOW model learns the embedding by predicting the next word based on the current word’s context. On the other hand, the continuous skip-gram model learns the embedding for the current word by predicting the words that will be around it.

Both models emphasise learning words based on their context, so the words are close by. A window of nearby words, therefore, determines the context of a word. This window is a model parameter that can be adjusted according to a given use case.


Example Sentence: The cat sat on the mat.
Continuous Bag-of-Words (CBOW)
Goal: Given context words, predict the target word.
The
cat
on
the
sat
Skip-gram Model
Goal: Given a word, predict the surrounding context words.
sat
The
cat
on
the

The method's main advantage is its ability to learn high-quality word embeddings quickly, enabling the learning of more significant embeddings from high-dimensional data. As a result, much larger corpora of text with billions of words can be easily represented.

2. GloVe

GloVe, or Global Vectors for Word Representation, is an unsupervised learning algorithm that obtains vector word representations by analyzing the co-occurrence statistics of words in a text corpus. These word vectors capture the semantic meaning and relationships between words.

The key idea behind GloVe is to learn word embeddings by examining the probability of word co-occurrences across the entire corpus. It constructs a global word-word co-occurrence matrix and then factorizes it to derive word vectors representing words in a continuous vector space.

These word vectors have gained popularity in natural language processing (NLP) tasks due to their ability to capture semantic relationships between words. They are used in various applications such as machine translation, sentiment analysis, text classification, and more, where understanding the meaning and context of words is crucial.

The word2vec algorithm has been extended to create the Global Vectors for Word Representation (GloVe) algorithm. GloVe is based on word-context matrix factorisation techniques. It first creates a sizable matrix of (words x context) co-occurrence data, in which you count the number of times a word appears in a particular "context" (the columns) for each "word" (the rows).

There are many "contexts" because their size is essentially combinatorial. When this matrix is factorised, a lower-dimensional (words x features) matrix is produced, with each row creating a vector representation for the corresponding word. Typically, this is accomplished by reducing a “reconstruction loss.” This loss looks for lower-dimensional models that can account for the majority of the variance in the high-dimensional data.

GloVe creates an explicit word context or word co-occurrence matrix using statistics across the entire text corpus rather than using a window to define local context, like in Word2Vec. The outcome is a learning model that might lead to more effective word embeddings.

3. FastText

FastText, essentially a word2vec model extension, treats each word as being made up of character n-grams. Thus, the sum of these character n-grams constitutes the vector for a word. For instance, the word vector “orange” is the sum of the n-gram vectors:

"<or", "ora", "oran", "orang", "orange" "orange>", "ran", "rang", "range" "range>", "ang", "ange", "ange>", "nge","nge>", "ge", "ge>"

The use of n-grams is the primary distinction between FastText and Word2Vec.

Word2Vec uses only complete words found in the training corpus to learn vectors. In contrast, FastText learns vectors for individual words and the n-grams found within them. The mean of the target word vector and its n-gram component vectors are used for training at each stage of the FastText process.

Each combined vector creates the target, which is then uniformly updated using the adjustment calculated from the error. These calculations significantly increase the amount of computation in the training phase. A word must add up and average its n-gram parts at each point.

Through various metrics, it has been demonstrated that these vectors are more accurate than Word2Vec vectors.

The most notable enhancement to FastText is the N-gram feature, which addresses the OOV (out-of-vocabulary) problem. For instance, the word "aquarium" can be broken down into "aq/aqu/qua/uar/ari/riu/ium/um>," where "<" and ">" denote the beginning and end of the word, respectively. Though the word embedder may not immediately recognise the word "Aquarius," it can infer its meaning. This can be done because the words "aquarium" and "Aquarius" share a common root.

A Simple Example of fastText

Imagine fastText as a tool that learns to understand words by breaking them down into smaller parts.

For instance, let's take the word "apple." Instead of treating it as a single entity, fastText dissects it into smaller components called character n-grams, such as 'ap,' 'pp,' 'pl,' 'le,' etc. These character-level fragments or subword units help fastText understand the word's structure and meaning.

Learning from a large dataset of words and their contexts, fastText grasps relationships between these subword units and how they combine to form words. This knowledge enables fastText to represent known words and unseen or rare words by considering their constituent subword components.

This approach allows fastText to create embeddings and numerical representations of words based on their subword information. These embeddings capture similarities and relationships between words, making them an efficient tool for language identification, text classification, and handling morphologically rich languages or specialized vocabularies.

Applications of fastText

1. Text Classification and Categorisation

fastText excels in text classification tasks, efficiently categorising texts into predefined classes or categories. Its ability to capture subword information allows for more nuanced Understanding, enabling accurate classification even with limited training data. This capability finds applications in spam filtering, topic categorisation, and content tagging across various domains.

2. Language Identification and Translation

The subword-level embeddings in fastText empower it to discern and work with languages even in cases where only fragments or limited text samples are available. This proves beneficial in language identification tasks, aiding multilingual applications and facilitating language-specific processing. Additionally, fastText’s embeddings have been utilised to enhance machine translation systems, improving the accuracy and performance of translation models.

3. Sentiment Analysis and Opinion Mining

In sentiment analysis, fastText’s robustness in capturing subtle linguistic nuances allows for more accurate sentiment classification. Its ability to understand and represent words based on their subword units enables a more profound comprehension of sentiment-laden expressions, contributing to more nuanced opinion mining in social media analysis, product reviews, and customer feedback.

4. Entity Recognition and Tagging

Entity recognition involves identifying and classifying entities within a text, such as names of persons, organisations, locations, and more. fastText’s subword embeddings contribute to better handling of unseen or rare entities, improving the accuracy of entity recognition systems. This capability finds applications in information extraction, search engines, and content analysis.

fastText’s versatility across these applications stems from its unique ability to handle subword information effectively, enabling a deeper understanding of language nuances. Its prowess in tasks like text classification, language identification, sentiment analysis, and entity recognition underlines its significance in diverse NLP applications, contributing to more accurate and efficient processing of textual data.

What is the Difference Between fastText and Word2Vec?

fastText and Word2Vec are two popular algorithms for generating word embeddings, but they differ significantly in their approaches and capabilities. Here are the critical differences between fastText and Word2Vec:

1. Handling of Out-of-Vocabulary (OOV) Words

- **Word2Vec:** Word2Vec operates at the word level, generating embeddings for individual words. It struggles with out-of-vocabulary words as it cannot represent words it hasn't seen during training.
- **fastText:** In contrast, fastText introduces subword embeddings by considering words to be composed of character n-grams. This enables it to handle out-of-vocabulary words effectively by breaking terms into subword units and generating embeddings for these units, even for unseen words. This capability makes fastText more robust in dealing with rare or morphologically complex expressions.

2. Representation of Words

- **Word2Vec:** Word2Vec generates word embeddings based solely on the words without considering internal structure or morphological information.
- **fastText:** fastText captures subword information, allowing it to understand word meanings based on their constituent character n-grams. This enables fastText to represent words by considering their morphological makeup, providing a richer representation, especially for morphologically rich languages or domains with specialised jargon.

3. **Training Efficiency**

- **Word2Vec:** The training process in Word2Vec is relatively faster than older methods but might be slower than fastText due to its word-level approach.
- **fastText:** fastText is known for its exceptional speed and scalability, especially when dealing with large datasets, as it operates efficiently at the subword level.

4. **Use Cases**

- **Word2Vec:** Word2Vec's word-level embeddings are well-suited for tasks like finding similar words, understanding relationships between words, and capturing semantic similarities.
- **fastText:** fastText's subword embeddings make it more adaptable in scenarios involving out-of-vocabulary words, sentiment analysis, language identification, and tasks requiring a deeper understanding of morphology.

How To Use fastText In Python: Practical Tutorial With Examples

1. Installation and Setup Guide for fastText

To start with fastText, the installation process is relatively straightforward. It's an open-source library and can be installed on various platforms. Here's a step-by-step guide:

Installation:

- For Python, using pip: **pip install fasttext**
- For building from source, clone the GitHub repository and follow the provided instructions for compilation.

Training Models:

Pre-trained models are available for download, but training custom models using your dataset is recommended for specific tasks or domain-specific applications. Use **fasttext.train_supervised** for text classification and **fasttext.train_unsupervised** for unsupervised tasks.

2. Example Use Cases and Code Snippets

Text Classification: Implementing a text classification task using a pre-existing dataset or your data involves:

```
import fasttext
```

```
# Training data file format: __label__<class_name> <text>

train_data = [

    "__label__positive This movie is fantastic!",

    "__label__negative I didn't like the ending of this book.",

    "__label__neutral The weather today is quite pleasant."

]


# Saving training data to a file

with open('train.txt', 'w') as f:

    for line in train_data:

        f.write(line + '\n')


# Training the model

model = fasttext.train_supervised(input="train.txt", epoch=25, lr=1.0)


# Testing the model

text_to_predict = "This restaurant serves delicious food!"

predicted_label = model.predict(text_to_predict)

print(predicted_label)
```

This example demonstrates how to train a simple text classification model using fastText. It starts by preparing training data in the format **__label__<class_name> <text>** and saves it to a file (‘train.txt’ in this case). Then, the model is trained using **fasttext.train_supervised()** with specified parameters like the input file, number of epochs, and learning rate.

After training, the model can be used to predict the label of new text samples using **model.predict()**. The output will display the provided text sample’s predicted label(s).

Word Embeddings: Generating word embeddings:

```
import fasttext


model = fasttext.train_unsupervised('corpus.txt', model='skipgram')

print(model.get_word_vector('example'))
```

Step5) Deep Learning Models

RNN is mainly used when we have the data sequence in hand, and we have to analyze that data. We will understand LSTM and GRU, conceptually succeeding topics after RNN.

1) Recurrent Neural Networks (RNN),

2) Long Short Term Memory (LSTM),

3) Gated Recurrent Unit (GRU).

1. recurrent neural network (RNN)

A recurrent neural network (RNN) is an artificial neural network that works well with data that comes in a certain order. RNNs are useful for tasks like translating languages, recognising speech, and adding captions to images. This is because they can process sequences of inputs and turn them into sequences of outputs. One thing that makes RNNs different is that they have “memory.” This lets them keep data from previous inputs in the current processing step.

To do this, hidden states are used. They are changed at each time step as the input sequence is processed and stored in memory. RNNs can unroll a sequence of inputs over time to show how they dealt with them step by step.

In NLP, RNNs are frequently used in machine translation to process a sequence of words in one language and generate a corresponding series of words in a different language as the output.

Language modelling, which involves predicting the following word in a sequence based on the preceding terms, is another application for RNNs. This can be used, for instance, to create text that appears to have been written by a person.

One thing that makes RNNs different is that they have “memory”.

One thing that makes RNNs different is that they have “memory”.

RNNs can also classify text by determining whether a passage is positive or negative. Or identifying named entities, such as people, organisations, and places mentioned in a passage.

RNNs can capture the relationships between words in a sequence and use this knowledge to predict the next word in the series. This makes them an effective tool for NLP tasks in general.

2. Long short-term memory (LSTM) networks

Recurrent neural networks (RNNs) of the type known as long short-term memory (LSTM) networks can recognise long-term dependencies in sequential data. They are beneficial in language translation, speech recognition, and image captioning. The input sequence can be very long, and the elements' dependencies can extend over numerous time steps.

"Memory cells," which can store data for a long time, and "gates," which regulate the information flow into and out of the memory cells, make up LSTM networks. LSTMs are especially good at finding long-term dependencies because they can choose what to remember and what to forget.

Elman RNNs and gated recurrent units (GRUs) are two examples of other RNNs that are typically simpler and easier to train than LSTM networks. However, LSTM networks are generally more powerful and perform better across various tasks.

3. Gated recurrent units (GRUs)

Long short-term memory (LSTM) networks and gated recurrent units (GRUs) are two types of recurrent neural networks (RNNs), but GRUs have fewer parameters and are typically simpler to train.

Like LSTMs, GRUs are effective for speech recognition, image captioning, and language translation because they can identify long-term dependencies in sequential data.

Update gates and reset gates are the two different types of gates found in GRUs. The reset gate decides what information should be forgotten, and the update gate decides what information should be kept from the previous time step. As with LSTMs, this enables GRUs to remember or omit information selectively.

GRUs are an excellent option for many NLP tasks, even though they are typically less effective than LSTMs due to their simplicity and ease of training. Also, they use less energy to run, which can be crucial in places where resources are scarce.

An implementation of an RNN in NLP using Keras

Here's an example of how to use the Python Keras library to set up a simple recurrent neural network (RNN) for natural language processing (NLP):

```
from keras.preprocessing.text import Tokenizer

from keras.preprocessing.sequence import pad_sequences

from keras.utils import to_categorical

from keras.layers import Embedding, LSTM, Dense, Dropout

from keras.models import Sequential

# Preprocess the data
```

```
texts = ['This is the first document', 'This document is the second document', 'And this is the third one', 'Is this the first document?']

max_words = 20000

max_len = 100

# Tokenize the texts

tokenizer = Tokenizer(num_words=max_words)

sequences = tokenizer.texts_to_sequences(texts)

# Pad the sequences to a fixed length

padded_sequences = pad_sequences(sequences, maxlen=max_len)

# Convert the labels to categorical variables

labels = to_categorical([0, 0, 1, 1])

# Build the model

model = Sequential()

model.add(Embedding(max_words, 128, input_length=max_len))
model.add(LSTM(64))

model.add(Dropout(0.5))

model.add(Dense(2, activation='sigmoid'))

# Compile the model

model.compile(loss='binary_crossentropy',                    optimizer='adam',

metrics=['accuracy'])

# Fit the model

model.fit(padded_sequences, labels, epochs=5, batch_size=32)
```

This example uses an LSTM layer to create a straightforward binary classification model. First, a list of texts is tokenized and then padded to a predetermined length. This is provided as input to the model.

After that, the labels are changed into categorical variables. The model has an embedding layer, an LSTM layer, a dropout layer, and a dense output layer.

The Adam optimisation algorithm and a binary cross-entropy loss function are used to construct the model. The model is then fitted to the padded sequences and labels for five epochs.

This is just a simple example. You will need more complicated preprocessing and model architectures for more complicated models and tasks. But this should give you a general idea of using Keras to implement an RNN for NLP.

Step 6) Bidirectional LSTM

Bi-LSTM (Bidirectional Long Short-Term Memory) is a type of recurrent neural network (RNN) that processes sequential data in both forward and backward directions. It combines the power of LSTM with bidirectional processing, allowing the model to capture both past and future context of the input sequence.

To understand Bi-LSTM, let's break down its components and functionality:

1. **LSTM (Long Short-Term Memory):** LSTM is a type of RNN designed to overcome the limitations of traditional RNNs in capturing long-term dependencies in sequential data. It introduces memory cells and gating mechanisms to selectively retain and forget information over time. LSTMs have an internal memory state that can store information for long durations,

allowing them to capture dependencies that may span across many time steps.

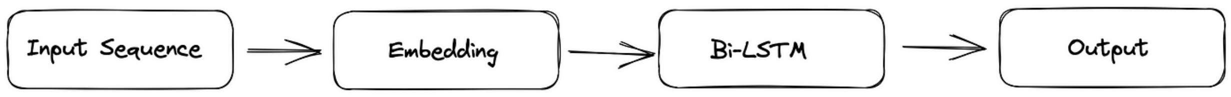
2. **Bidirectional Processing:** Unlike traditional RNNs that process input sequences in only one direction (either forward or backward), Bi-LSTM processes the sequence in both directions simultaneously. It consists of two LSTM layers: one processing the sequence in the forward direction and the other in the backward direction. Each layer maintains its own hidden states and memory cells.
3. **Forward Pass:** During the forward pass, the input sequence is fed into the forward LSTM layer from the first time step to the last. At each time step, the forward LSTM computes its hidden state and updates its memory cell based on the current input and the previous hidden state and memory cell.
4. **Backward Pass:** Simultaneously, the input sequence is also fed into the backward LSTM layer in reverse order, from the last time step to the first. Similar to the forward pass, the backward LSTM

computes its hidden state and updates its memory cell based on the current input and the previous hidden state and memory cell.

5. **Combining Forward and Backward States:** Once the forward and backward passes are complete, the hidden states from both LSTM layers are combined at each time step. This combination can be as simple as concatenating the hidden states or applying some other transformation.

The benefit of Bi-LSTM is that it captures not only the context that comes before a specific time step (as in traditional RNNs) but also the context that follows. By considering both past and future information, Bi-LSTM can capture richer dependencies in the input sequence.

The architecture of Bi-LSTM can be shown in the following figure:



Let’s break down each component of the architecture:

1. **Input Sequence:** The input sequence is a sequence of data points, such as words in a sentence or characters in a text. Each data point is typically represented as a vector or embedded representation.
2. **Embedding:** The input sequence is often transformed into dense vector representations called embeddings. Embeddings capture the semantic meaning of the data points and provide a more compact and meaningful representation for the subsequent layers.
3. **Bi-LSTM:** The Bi-LSTM layer is the core component of the architecture. It consists of two LSTM layers: one processing the input sequence in the forward direction and the other in the backward direction. Each LSTM layer has its own set of parameters.
4. **Output:** The output of the Bi-LSTM layer is the combination of the hidden states from both the forward and backward LSTM layers at each time step. The specific combination method can vary,

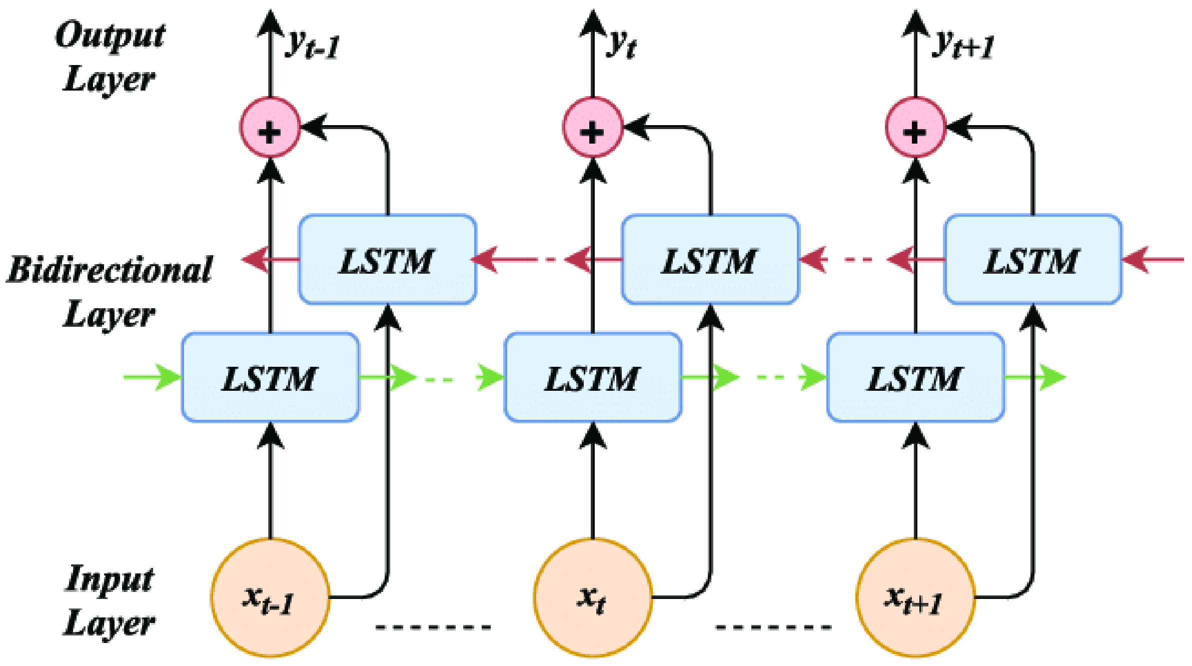
such as concatenating the hidden states or applying a different transformation.

The Bi-LSTM layer processes the input sequence in both forward and backward directions simultaneously. During the forward pass, the LSTM layer captures information from the past (previous time steps), while during the backward pass, it captures information from the future (following time steps). This bidirectional processing allows the model to effectively capture long-term dependencies in the input sequence.

The output of the Bi-LSTM layer can be used for various purposes depending on the specific task. For example, in text classification, the output may be passed through a fully connected layer followed by a softmax activation to obtain class probabilities. In sequence labeling tasks like named entity recognition, the output may be directly used to predict the label for each input token.

The architecture of a Bi-LSTM can be further extended or modified based on the specific requirements of the task. Additional layers, such as fully connected layers or attention mechanisms, can be added on top of the Bi-LSTM layer to further enhance the model's capabilities and performance.

The internal Architecture of Bi-LSTM can be shown in the following figure:



Python Implementation of Bi-LSTM

Here's an example of a Python implementation of a Bi-LSTM using the Keras library:

```
from keras.models import Sequential

from keras.layers import Embedding, Bidirectional, LSTM, Dense
```

```
# Define the model architecture

model = Sequential()

# Add an embedding layer to convert input sequences to dense vectors

model.add(Embedding(input_dim=vocab_size,

output_dim=embedding_dim, input_length=max_sequence_length))

# Add a Bidirectional LSTM layer

model.add(Bidirectional(LSTM(units=lstm_units, return_sequences=True)))

# Add a dense output layer

model.add(Dense(units=num_classes, activation='softmax'))

# Compile the model

model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['

accuracy'])

# Print the model summary

model.summary()
```

In this code, you’ll need to replace vocab_size with the size of your vocabulary, embedding_dim with the desired dimensionality of the embedding space, max_sequence_length with the maximum length of your input sequences, lstm_units with the number of LSTM units, and num_classes with the number of classes in your classification task.

The model starts with an embedding layer that converts the input sequences into dense vectors. The Bidirectional LSTM layer processes the embedded sequences in both forward and backward directions. Finally, a dense output layer with softmax activation is added for classification tasks.

After defining the model, you can compile it by specifying the loss function, optimizer, and metrics. Then, you can train the model on your labeled data using the fit() function.

Keep in mind that this is a simplified example, and you may need to adapt it to your specific use case by adding additional layers, adjusting hyperparameters, and preprocessing your data accordingly.

Pros and Cons of using Bidirectional- LSTM

Here are some potential pros and cons of using a bidirectional LSTM for a particular task:

Pros:

1. **Captures long-term dependencies:** Bidirectional LSTMs can capture long-term dependencies in sequential data by processing input sequences in both forward and backward directions. This makes them well-suited for tasks that require modeling context over a long period of time, such as speech recognition or natural language processing.
2. **Improved performance:** Bidirectional LSTMs often perform better than traditional LSTMs on tasks such as speech recognition, machine translation, and sentiment analysis.
3. **Flexible architecture:** The architecture of bidirectional LSTMs is flexible and can be customized by adding additional layers, such as convolutional or attention layers, to improve performance.

Cons:

1. **High computational cost:** Bidirectional LSTMs can be computationally expensive due to the need to process the input

sequence in both directions. This can make them impractical for use in resource-constrained environments.

2. **Requires large amounts of data:** Bidirectional LSTMs require large amounts of training data to learn meaningful representations of the input sequence. Without sufficient training data, the model may overfit to the training set or fail to generalize to new data.
3. **Difficult to interpret:** Bidirectional LSTMs are often seen as “black boxes,” making it difficult to interpret how the model is making predictions. This can be problematic in applications where interpretability is important, such as medical diagnosis or financial analysis.

Overall, the decision to use a bidirectional LSTM should depend on the specific task at hand and the available resources, as well as the trade-offs between performance and interpretability.

Step7) Sequence-to-sequence models, Self-attention models

1. Sequence-to-sequence models

Sequence-to-sequence (Seq2Seq) is a deep learning architecture used in natural language processing (NLP) and other sequence modelling tasks. It is designed to handle input sequences of variable length and generate output sequences of varying length, making it suitable for tasks like machine translation, text summarization, speech recognition, and more.

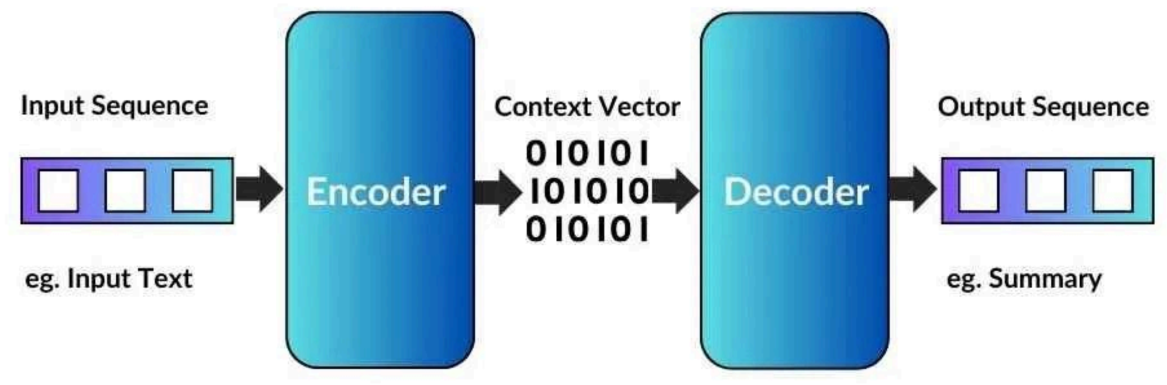
To appreciate Seq2Seq models, let's begin by acknowledging the ubiquity of sequences in our world. Sequences are everywhere, from natural language sentences and speech signals to financial time series and genomic data. A sequence is simply an ordered list of elements and the order matters. Understanding, predicting, and generating sequences pose unique challenges compared to traditional fixed-size data.

Traditional machine learning models, like feedforward neural networks, are unsuited for handling sequences because they expect inputs and outputs of fixed dimensions. Seq2Seq models, on the other hand, are specifically designed to tackle variable-length sequences. This flexibility allows Seq2Seq models to excel in many tasks with pivotal sequences.

The Encoder-Decoder Architecture

At the heart of Seq2Seq models lies the encoder-decoder architecture. This architectural paradigm mirrors the human thought process: first, we gather information, and then we use that information to generate a response or make a decision.

- **Encoder**: The encoder comprehends the input sequence and captures its essential information. It processes the sequence step by step, often using recurrent neural networks (RNNs), long short-term memory networks (LSTMs), or, more recently, transformer-based architectures. As it does so, it accumulates knowledge in a fixed-size context vector, sometimes called the “thought” vector.
- **Decoder**: The decoder takes over once the input sequence is encoded into a context vector. It generates the output sequence based on the information stored in the context vector. Like the encoder, the decoder can also employ RNNs, LSTMs, transformers, or other sequential modelling techniques to produce the output sequence. This generation process is typically performed one step at a time, with each step considering the previous output and the context vector.



Sequence-to-sequence architecture

What are sequence-to-sequence models used for?

Seq2Seq models are exceptionally versatile. They can be applied to various problems involving sequences, making them indispensable in natural language processing, speech recognition, and more. For instance:

- **Machine Translation**: Seq2Seq models have revolutionized machine translation by enabling systems to translate text from one language to another, capturing language structure and context nuances.
- **Text Summarization**: They are adept at summarizing lengthy texts into concise and coherent summaries, a valuable skill in information retrieval and content extraction.

- **Speech Recognition**: Seq2Seq models can convert spoken language into written text, driving the development of voice assistants and transcription services.

In the following sections, we'll dive deeper into the inner workings of Seq2Seq models, explore the building blocks that make them tick, and demonstrate how to build your own Seq2Seq model for various applications. By the end of this blog post, you'll have a solid grasp of Seq2Seq models and the tools to leverage their power in your projects.

The Encoder-Decoder Architecture

In the previous section, we introduced the concept of sequence-to-sequence (Seq2Seq) models and their importance in handling data sequences. Now, let's dive deeper into the fundamental architecture of Seq2Seq models, known as the encoder-decoder framework. Understanding how these components work together is essential for harnessing the full potential of Seq2Seq models.

The Encoder: Capturing Context

The encoder is the first half of the Seq2Seq model, responsible for processing the input sequence and summarizing its essential information into a fixed-size context vector. This context vector is a "thought" vector, encapsulating the knowledge extracted from the input sequence. Here’s how the encoder typically functions:

- **Embedding Layer**: The input sequence, often consisting of discrete elements like words or tokens, is passed through an embedding layer. This layer converts each element into a continuous vector representation, making it suitable for neural network processing.
- **Recurrent Layers (or Transformers)**: Following the embedding layer, recurrent layers, such as RNNs or LSTMs, process the embedded input sequence step by step. Modern Seq2Seq models may use transformer-based encoders for parallelism and improved performance.
- **Context Vector**: As the encoder processes the input sequence, it accumulates information as hidden states. Once the input sequence is processed, the final hidden state is used as the context vector. This context vector encapsulates the essence of the input sequence and serves as the starting point for the decoder.

The Decoder: Generating Output Sequences

The decoder, the second half of the Seq2Seq model, takes the context vector produced by the encoder and generates the output sequence. Its primary role is to produce a sequence of elements based on the context vector and, optionally, some initial input. Here’s how the decoder typically operates:

- **Embedding Layer**: Similar to the encoder, the decoder's first step is to use an embedding layer to transform the input sequence (or individual elements) into continuous vector representations.
- **Recurrent Layers (or Transformers)**: Like the encoder, the decoder can employ recurrent layers or transformers to process the embedded input. These layers generate output sequences one step at a time, considering both the context vector and the previously generated elements of the output sequence.
- **Output Layer**: The decoder produces a probability distribution over the possible output elements at each step. This distribution is then used to sample the next component in the output sequence. The output layer is typically a softmax layer that converts the model's predictions into probabilities.
- **Teacher Forcing (Optional):** "Teacher forcing" is often used during training. It involves feeding the true target sequence elements as inputs to the decoder rather than using its predictions. This helps stabilize training but may lead to a discrepancy between training and inference.

Cooperation of Encoder and Decoder

The synergy between the encoder and decoder enables Seq2Seq models to shine in various tasks. The encoder summarizes the input sequence into a context vector, which serves as a compact representation of the input's information. The decoder then uses this context vector to generate the output sequence, one step at a time.

The encoder-decoder architecture is a powerful paradigm for handling sequences of varying lengths. However, it’s not without challenges. Seq2Seq models must learn to capture relevant information in the context vector and effectively decode it into the output sequence. To address these challenges, techniques like attention mechanisms, which allow the model to focus on specific parts of the input sequence, have been introduced and have greatly improved the capabilities of Seq2Seq models.

The following section will explore the building blocks that makeup Seq2Seq models, including embeddings, recurrent layers, and attention mechanisms. These components are the key to unlocking the full potential of Seq2Seq models in various applications.

Building Blocks of Sequence-to-Sequence Models

In the previous section, we explored the fundamental architecture of sequence-to-sequence (Seq2Seq) models, emphasizing the crucial roles played by the encoder and decoder. Now, let’s look at the building blocks that constitute Seq2Seq models and enable them to excel in various sequence-related tasks.

The Role of Embeddings

One essential building block in Seq2Seq models is embeddings. Embeddings are vector representations of discrete elements such as words or tokens. They serve as the bridge between the discrete world of sequences and the continuous space of neural networks. Here’s how embeddings work in Seq2Seq models:

- **Input Embeddings**: At the beginning of the encoder and decoder, each element of the input and target sequences is embedded into continuous vectors. These embeddings capture semantic information about the elements, allowing the model to work with them in a continuous space. Training the embeddings is part of the model’s learning process, which means they adapt to the task.
- **Shared Embeddings**: In many Seq2Seq models, the same embedding layer is used for input and target sequences. This shared embedding space enables the model to represent input and output elements, facilitating better learning consistently.

Recurrent Layers vs. Transformers

Seq2Seq models require layers that can handle sequences, and two popular choices are recurrent layers (RNNs and LSTMs) and transformer-based architectures. Here’s a brief comparison:

- **Recurrent Layers (RNNs and LSTMs)**: These layers process sequences sequentially, one element at a time. They maintain hidden states that capture information from previous steps, allowing them to capture sequential dependencies effectively. However, they may struggle with long sequences and can be computationally intensive.
- **Transformers**: Transformers, introduced in the famous “Attention Is All You Need” paper, have revolutionized Seq2Seq models. They use

self-attention mechanisms to capture dependencies between all elements in a sequence in parallel. This parallelism makes transformers highly efficient and suitable for long sequences. They have become the go-to choice for many Seq2Seq applications.

The Power of Attention Mechanisms

Attention mechanisms are a game-changer in Seq2Seq modelling. They allow the model to selectively focus on specific parts of the input sequence when generating the output sequence. The attention mechanism consists of three main components:

- **Query**: The decoder generates the element at a given time step.
- **Key-Value Pairs**: Elements of the input sequence, serving as keys and values.
- **Attention Scores**: Calculated based on the compatibility between the query and the keys.

The attention scores determine how much attention the model should pay to each input sequence element when generating the current output element. This dynamic attention mechanism enables Seq2Seq models to handle long-range dependencies and accurately align input elements with output elements.

What are some variations of sequence-to-sequence models?

Seq2Seq models are highly adaptable and have seen numerous variations and enhancements. Here are some notable techniques and variations:

- **Teacher Forcing**: During training, Seq2Seq models can use the true target sequence as inputs to the decoder instead of using its predictions. This technique, known as “teacher forcing,” helps stabilize training but can lead to discrepancies between training and inference.
- **Beam Search**: Beam search is often employed for generating sequences during inference. It explores multiple potential output sequences in parallel and selects the one with the highest probability. This technique often leads to more coherent and accurate outputs.

In the upcoming sections, we’ll explore real-world applications of Seq2Seq models, including machine translation, text summarization, and more. Additionally, we’ll provide practical examples and demonstrate how to build Seq2Seq models for specific tasks using popular deep learning frameworks.

What are common applications of sequence-to-sequence models?

Now that we’ve explored the foundational components of sequence-to-sequence (Seq2Seq) models, it’s time to delve into the exciting real-world applications, where these models have significantly impacted. Seq2Seq models have proven their versatility by addressing various sequence-related tasks, offering state-of-the-art solutions across multiple domains.

1. **Machine Translation Machine** translation, the task of automatically translating text from one language to another, has been revolutionized by Seq2Seq models. Here’s how Seq2Seq models excel in this application:

- **Bidirectional Understanding**: Seq2Seq models can encode input sentences in the source language and generate equivalent sentences in the target language. This bidirectional understanding allows them to capture language structure and context nuances.
- **Transformer-Based Models**: Transformer-based Seq2Seq models, such as the famous “Transformer” architecture, have become the backbone of modern machine translation systems. They use self-attention mechanisms to handle long-range dependencies effectively.
- **Multilingual Translation**: Seq2Seq models can handle multiple languages, enabling the development of multilingual translation systems that can translate between numerous language pairs.

2. **Text Summarization**

Seq2Seq models have also made significant strides in abstractive text summarization. Abstractive summarization involves generating a concise and coherent summary of a longer text instead of extractive summarization, which selects and combines existing sentences. Here’s how Seq2Seq models shine in this task:

- **Content Understanding**: Seq2Seq models effectively understand the content of the input text and generate summaries that capture the critical information, making them highly suitable for news summarization, document summarization, and content extraction.
- **Advanced Architectures**: Variations of Seq2Seq models, often incorporating attention mechanisms and transformer-based architectures, have improved the quality and coherence of generated summaries.

3. **Speech Recognition**

Seq2Seq models play a pivotal role in converting spoken language into written text, a task known as automatic speech recognition (ASR). Here’s why Seq2Seq models are instrumental in this application:

- **Audio Processing**: ASR involves processing audio waveforms as input sequences, and Seq2Seq models, particularly those with attention mechanisms, excel in capturing spoken language patterns.
- **Accuracy Improvement**: Using Seq2Seq models has led to significant improvements in ASR accuracy, making them essential for voice assistants, transcription services, and more.
- **Multilingual ASR**: Seq2Seq models can also be adapted for multilingual ASR, where they accurately transcribe speech in multiple languages.

4. **Image Captioning**

Seq2Seq models are not limited to processing text data; they can also handle image data effectively. In image captioning, Seq2Seq models generate natural language descriptions of images. Here’s why Seq2Seq models are a natural fit for this task:

- **Multimodal Understanding**: Seq2Seq models combine image features extracted by convolutional neural networks (CNNs) with textual generation capabilities, enabling them to describe images in a human-readable format.
- **Creative Descriptions**: Seq2Seq models can generate creative and contextually relevant image captions, which is valuable in applications like content generation and accessibility for the visually impaired.
- **Visual question Answering**: Seq2Seq models are used in tasks like visible question answering (VQA), where they generate answers to questions about images, demonstrating their versatility in multimodal applications.

These are just a few examples of the many applications where Seq2Seq models have proven their worth. From machine translation to summarization, speech recognition, and image captioning, Seq2Seq models continue to drive innovation and advance the capabilities of machine learning in handling sequential data.

In the next section, we’ll take a more hands-on approach and guide you through building your own Seq2Seq model for a specific task using popular deep learning frameworks.

How to build your sequence-to-sequence model

Now that we've explored the fundamental concepts and applications of sequence-to-sequence (Seq2Seq) models, it's time to get hands-on and guide you through building your own Seq2Seq model. Whether you're interested in machine translation, text summarization, or another sequence-related task, this section will provide the essential steps to get started.

Step 1: Data Preparation

The first crucial step is data preparation. You'll need a dataset that suits your task. You'll need parallel text data in both source and target languages for machine translation. You'll require a corpus of text documents and their corresponding summaries for text summarization. You might need to tokenize and preprocess the data depending on your task.

Step 2: Define Your Model Architecture

The choice of model architecture depends on your task and data. You can create a Seq2Seq model using popular deep learning frameworks like TensorFlow or PyTorch. Here's a high-level overview of the steps involved:

- **Encoder**: Define the encoder component to process the input sequence and generate a context vector. You can choose from various architectures, including LSTM, GRU, or transformer-based encoders.

- **Decoder**: Create the decoder component responsible for generating the output sequence based on the context vector and, optionally, some initial input. You can choose from different recurrent layers or transformer-based architectures like the encoder.
- **Embeddings**: Implement embeddings for the input and output sequences. These embeddings allow the model to work with discrete elements, such as words or tokens, in a continuous vector space.
- **Attention Mechanisms (Optional)**: Depending on your task and model architecture, you might want to incorporate attention mechanisms to improve the model's ability to focus on relevant parts of the input sequence.
- **Loss Function and Optimization**: Define an appropriate loss function for your task, such as categorical cross-entropy for text generation tasks. Choose an optimization algorithm like Adam or SGD to train your model.

Step 3: Training and Evaluation

Once your model architecture is defined, it's time to train and evaluate the model:

- **Training**: Use your prepared dataset to train the model. During training, you'll input source sequences and their corresponding target sequences and update the model's parameters to minimize the chosen loss function. Consider techniques like teacher forcing to improve training stability.

- **Validation**: Monitor the model's performance on a validation dataset to prevent overfitting. Adjust hyperparameters and model architecture as needed.
- **Testing**: Evaluate the model on a separate test dataset to assess its generalization to unseen data.

Step 4: Inference

After training, you can use your Seq2Seq model for inference on new data:

- **Inference Pipeline**: Set up an inference pipeline that takes input sequences, preprocesses them (e.g., tokenization), passes them through the encoder, and generates the output sequences using the decoder.
- **Beam Search (Optional)**: For tasks like text generation, consider using beam search during inference to improve the quality of generated sequences.

Step 5: Fine-Tuning and Iteration

Seq2Seq model building is an iterative process. You may need to fine-tune your model's hyperparameters, adjust the architecture, or experiment with different techniques to improve performance on your task.

Step 6: Deployment (Optional)

If your Seq2Seq model is intended for production use, consider deploying it as a service or integrating it into your application. Frameworks like TensorFlow Serving and Flask can help you deploy models in a production environment.

Step 7: Experiment and Explore

Don’t be afraid to experiment and explore! Seq2Seq models offer a wide range of possibilities, and you can adapt them to various tasks beyond the ones mentioned in this blog post. Consider tackling challenges like code generation, conversational agents, or creative content generation.

Remember that building Seq2Seq models requires a combination of domain expertise, data preparation, and experimentation. It’s a rewarding journey that allows you to leverage the power of deep learning for sequence-related tasks.

How to implement sequence-to-sequence models in PyTorch

In PyTorch, you can implement a sequence-to-sequence (Seq2Seq) model for various tasks such as machine translation, text summarization, and speech recognition. Here, we will provide a high-level overview of how to build a Seq2Seq model using PyTorch. Note that this is a simplified example, and actual implementations can vary depending on the specific task and model architecture.

Assuming you have PyTorch installed, you can create a basic Seq2Seq model as follows:

```
import torch

import torch.nn as nn

import torch.optim as optim



# Define the Encoder

class Encoder(nn.Module):

    def __init__(self, input_dim, emb_dim, hidden_dim, n_layers, dropout):

        super().__init__()



        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.rnn = nn.LSTM(emb_dim, hidden_dim, n_layers, dropout = 
dropout)
        self.dropout = nn.Dropout(dropout)


    def forward(self, src):
        embedded = self.dropout(self.embedding(src))
        outputs, (hidden, cell) = self.rnn(embedded)
        return hidden, cell


# Define the Decoder
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hidden_dim, n_layers, dropout):
        super().__init__()
        self.output_dim = output_dim

        self.embedding = nn.Embedding(output_dim, emb_dim)

        self.rnn = nn.LSTM(emb_dim, hidden_dim, n_layers, dropout = dropout)

        self.fc_out = nn.Linear(hidden_dim, output_dim)

        self.dropout = nn.Dropout(dropout)



    def forward(self, input, hidden, cell):

        input = input.unsqueeze(0)

        embedded = self.dropout(self.embedding(input))

        output, (hidden, cell) = self.rnn(embedded, (hidden, cell))

        prediction = self.fc_out(output.squeeze(0))

        return prediction, hidden, cell
```

```
# Define the Seq2Seq model

class Seq2Seq(nn.Module):

    def __init__(self, encoder, decoder, device):

        super().__init__()



        self.encoder = encoder

        self.decoder = decoder

        self.device = device



    def forward(self, src, trg, teacher_forcing_ratio = 0.5):

        # src: source sequence, trg: target sequence

        trg_len = trg.shape[0]

        batch_size = trg.shape[1]

        trg_vocab_size = self.decoder.output_dim
```

```
                    outputs    =    torch.zeros(trg_len,    batch_size,
trg_vocab_size).to(self.device)

        # Initialize the hidden and cell states of the encoder

        hidden, cell = self.encoder(src)

        # Take the <sos> token as the first input to the decoder

        input = trg[0,:]

        for t in range(1, trg_len):

            output, hidden, cell = self.decoder(input, hidden, cell)

            outputs[t] = output

            teacher_force = random.random() < teacher_forcing_ratio

            top1 = output.argmax(1)

            input = trg[t] if teacher_force else top1

        return outputs
```

How to implement sequence-to-sequence in Tensorflow

Creating a sequence-to-sequence (Seq2Seq) model using TensorFlow involves defining an encoder-decoder architecture. Here, I'll provide a simplified example using TensorFlow. Remember that real-world implementations can vary significantly based on the specific task and model architecture.

Assuming you have TensorFlow installed, here's an outline of how to build a Seq2Seq model using TensorFlow:

```
import tensorflow as tf

from tensorflow.keras.layers import Embedding, LSTM, Dense

from tensorflow.keras.models import Model



# Define the Encoder

class Encoder(tf.keras.Model):

    def __init__(self, vocab_size, embedding_dim, enc_units):

        super(Encoder, self).__init__()

        self.embedding = Embedding(vocab_size, embedding_dim)
        self.lstm = LSTM(enc_units, return_sequences=True,
return_state=True)

    def call(self, x):
        x = self.embedding(x)
        output, state_h, state_c = self.lstm(x)
        return output, state_h, state_c

# Define the Decoder
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units):
        super(Decoder, self).__init__()
        self.embedding = Embedding(vocab_size, embedding_dim)
        self.lstm = LSTM(dec_units, return_sequences=True,
return_state=True)
        self.dense = Dense(vocab_size, activation='softmax')


    def call(self, x, initial_state):
        x = self.embedding(x)
        output, _, _ = self.lstm(x, initial_state=initial_state)
        prediction = self.dense(output)
        return prediction


# Define the Seq2Seq Model
class Seq2SeqModel(tf.keras.Model):
    def __init__(self, encoder, decoder):
        super(Seq2SeqModel, self).__init__()
        self.encoder = encoder

        self.decoder = decoder



    def call(self, inputs):

        source, target = inputs

        enc_output, enc_state_h, enc_state_c = self.encoder(source)

            dec_output = self.decoder(target, initial_state=[enc_state_h,
enc_state_c])

        return dec_output



# Define the hyperparameters and instantiate the model

vocab_size = 10000  # Example vocabulary size

embedding_dim = 256

enc_units = 512
```

```
dec_units = 512



encoder = Encoder(vocab_size, embedding_dim, enc_units)

decoder = Decoder(vocab_size, embedding_dim, dec_units)

seq2seq_model = Seq2SeqModel(encoder, decoder)



# Compile the model (you may choose an appropriate optimizer and loss function)

seq2seq_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

In this example, we define an encoder-decoder architecture using LSTM layers. You must adapt this code to your specific task, prepare data, and define training and evaluation loops. Additionally, consider adding mechanisms like attention to improve performance for more complex tasks.

What are the challenges in sequence-to-sequence modelling?

1. **Handling Long Sequences**: Seq2Seq models can struggle with long sequences because they rely on recurrent layers or self-attention mechanisms. Addressing this challenge is a crucial area of ongoing research.
2. **Training Instability**: Training Seq2Seq models can sometimes be unstable, leading to issues like vanishing gradients or mode collapse. Techniques like gradient clipping and curriculum learning are employed to mitigate these problems.
3. **Inference Speed**: While transformer-based Seq2Seq models are highly effective, they can be computationally intensive during inference, particularly for real-time applications. Optimizing model inference speed is a priority.
4. **Multimodal Seq2Seq**: Extending Seq2Seq models to handle multiple modalities (e.g., text and images) remains a challenging research area, with applications in areas like visual question answering (VQA) and more.

What does the future hold for sequence-to-sequence modelling?

1. **Advanced Architectures**: Seq2Seq models will continue to benefit from advances in architecture design. Researchers are exploring novel model architectures and ways to make them more efficient and capable.
2. **Few-Shot and Zero-Shot Learning**: Seq2Seq models are being extended to tackle few-shot and zero-shot learning scenarios, where the model can generalize from minimal examples or adapt to entirely new tasks.

3. **Multilingual and Cross-Lingual Models**: Developing multilingual and cross-lingual Seq2Seq models is ongoing, enabling seamless translation and understanding across multiple languages.
4. **Continual Learning**: Seq2Seq models that can continuously adapt and learn from new data are promising, particularly for applications that require lifelong learning.
5. **Interpretable Seq2Seq**: Researchers are making Seq2Seq models more interpretable, allowing users to understand why the model makes specific predictions or generates particular sequences.
6. **Ethical Considerations**: As Seq2Seq models become more capable, ethical considerations around their use and potential biases in the data they learn from are increasingly important. Research in this area will continue to evolve.
7. **Integration with Real-World Systems**: Seq2Seq models will become more tightly integrated into real-world systems and applications, impacting fields such as healthcare, finance, and education.

2. Self-attention

Self-attention is the reason transformers are so successful at many NLP tasks. Learn how they work, the different types, and how to implement them with PyTorch in Python.

Self-attention is a type of attention mechanism used in deep learning models, also known as the self-attention mechanism. It lets a model decide how important each part of an input sequence is, which makes it possible to find dependencies and connections in the data.

Self-attention is used extensively in deep learning architectures, especially in natural language processing (NLP). For example, tasks like machine translation, sentiment analysis, and question-answering depend significantly on self-attention.

In self-attention, a model calculates the attention weights between each element in the input sequence, allowing it to focus on the relevant factors for a given task. This mechanism works very well because it lets the model take into account long-term dependencies and relationships in the data, which improves performance on many jobs. Self-attention looks for relationships in the data.

Self-attention example

An example of self-attention in deep learning is its use in machine translation. In this task, a model takes a source sentence in one language as input and produces a translated sentence in another.

Using self-attention, the model can focus on different parts of the source sentence, assigning weights to each piece to determine its importance in the translation.

For example, in a sentence like “I will go to the park with my friends,” the model may give more weight to the word “park” because it is an essential aspect of the sentence that needs to be translated correctly. The self-attention mechanism allows the model to make these dynamic, context-specific decisions, improving the accuracy of the translation.

Types of self-attention

There are several types of self-attention mechanisms used in deep learning, including:

1. **Dot-product attention**: The attention scores are calculated as the dot product of the queries and keys. This type of self-attention is used in the Transformer architecture.
2. **Scaled dot-product attention:** is similar to dot-product attention, but the attention scores are divided by the square root of the number of dimensions of the queries and keys to ensure they are stable.
3. **Multi-head attention**: Multiple attention heads capture different aspects of the input sequence. Each head calculates its own set of attention scores, and the results are concatenated and transformed to produce the final attention weights.
4. **Local attention**: The attention mechanism is only used on several elements in the input sequence. This lets the model focus on dependencies close to the sequence’s beginning or end.

5. **Additive attention**: The attention scores are calculated as a function of the similarity between the queries and keys rather than just their dot product.
6. **Cosine attention:** scores are calculated as the cosine similarity between the queries and keys.

These are some of the most commonly used self-attention mechanisms in deep learning. The choice of a self-attention mechanism depends on the specific task and the desired properties of the model.

Self-attention vs attention

Self-attention and attention are similar mechanisms in deep learning, but there is a critical difference between the two.

Attention refers to a mechanism in which a model calculates attention scores between different parts of an input and another part of the input or external memory. For example, in machine translation, the attention mechanism calculates attention scores between the source sentence and the target sentence, allowing the model to weigh the importance of each part of the source sentence in the target translation.

On the other hand, self-attention is a mechanism by which the model calculates attention scores between different parts of the input sequence without using external memory. Self-attention lets the model figure out how important each part of the series is, determine how the parts depend on each other and make predictions based on that.

In short, attention is a mechanism in which a model calculates attention scores between different parts of an input and another part of the input or external memory. On the other hand, self-attention is a mechanism in which the model only calculates attention scores between different parts of the input sequence.

Step 8) Transformers

A transformer is a kind of neural network architecture used in natural language processing (NLP) to process sequential data. It was first discussed in the 2017 paper “Attention is All You Need” by Vaswani et al. It has since been widely applied to language modelling, summarisation, and machine translation tasks.

Instead of using a fixed window of adjacent elements as in conventional RNNs or CNNs, the transformer architecture is based on self-attention, which enables the model to compare any two input elements with one another directly. This makes it easier to train and run the transformer than models based on RNNs or CNNs, and it also makes it better to find long-range dependencies in the input data.

The transformer architecture has generally been very effective in NLP and has resulted in appreciable performance improvements on various tasks. It has also been used in other fields, like computer vision, and has many potential applications.

Transformers have led to increased performance in various NLP tasks.

What is an encoder in a neural network?

Known as a latent representation or embedding, an encoder is a part of a neural network that processes the input data and transforms it into a compact model. This latent representation, which usually has fewer dimensions than the original data, tries to capture the original data's most important parts or traits more concisely.

A sequence of words or tokens in a sentence or document is processed by an encoder in a natural language processing (NLP) context and transformed into a continuous, fixed-length vector representation. This vector representation can then be fed into a decoder or a classifier, among other model parts.

Encoders are frequently employed in tasks like machine translation. For example, the encoder analyses the input sentence in the source language. It produces a latent representation, which is then sent to a decoder, which does the translation in the target language. Additionally, they are employed in language modelling, where the encoder analyses a run of words and forecasts the following term.

What is a decoder in a neural network?

A decoder is a part of a neural network that takes an embedding-like compact representation of the input data and changes it into a more helpful format for the task at hand. For example, in natural language processing (NLP), a decoder is often used to make text from an input embedding that shows the context or meaning of the text.

For instance, in machine translation, the decoder uses the latent representation created by the encoder as input to produce the translation in the target language. In language modelling, the decoder predicts the following word in the sequence using the embedding produced by the encoder as input.

Recurrent neural networks (RNNs) or transformers are frequently used in the implementation of decoders, allowing them to process sequential input data and produce output sequences of varying lengths. They can also be used in conjunction with attention mechanisms, which let the decoder generate the output while selectively focusing on various elements of the input embedding.

What is an encoder-decoder architecture in a neural network?

A typical neural network architecture for tasks involving changing a sequence of data from one form to another is an encoder-decoder. It is made up of an encoder and a decoder.

The encoder processes the input sequence into an embedding—a condensed, fixed-length representation. The embedding, which usually has fewer dimensions than the original data, tries to capture better the most important parts or characteristics of the original data.

The output sequence, which should be a modified version of the input sequence, is then produced by the decoder after processing the embedding.

For example, in machine translation, an input sequence is a sentence in the source language, an embedding is a hidden representation of the sentence's meaning, and an output sequence is the translation of the sentence into the target language.

Generally speaking, the encoder-decoder architecture is famous for tasks involving sequential data and has proven particularly effective in natural language processing (NLP) applications. However, it has also been used in other fields, like computer vision, and has many potential applications.

What is the difference between encoder, decoder and encoder-decoder in a neural network?

The input data is processed by an encoder transformer that embeds it in a condensed, fixed-length representation. The embedding, which usually has fewer dimensions than the original data, tries to capture better the most important parts or characteristics of the original data.

An embedding is processed by a decoder transformer, which then transforms it back into a format better suited for the task. In machine translation, for example, the latent representation made by the encoder is used by the decoder as input to do the translation in the target language.

An encoder-decoder transformer is frequently used when converting a sequence of data from one form to another, as is the case when performing machine translation, summarisation, or language modelling. The encoder processes the input sequence, which also creates an embedding. This embedding is then given to the decoder, which makes the output sequence.

In general, you would use an encoder when you wanted to compress the input data, a decoder when you needed to produce some output based on the input, and an encoder-decoder when you needed to change the format of a sequence of data.

Applications of encoder and decoder architecture in a neural network

Encoder

- **Image classification:** An encoder can analyse an image and produce a compact embedding that encapsulates its features or characteristics. A classifier can be given the embedding to determine the label or class of the picture.
- **Speech recognition:** By analysing a waveform, an encoder can produce an embedding that depicts the features or traits of the speech signal. The embedding can then be given to a classifier to predict the transcription of the speech.

Decoder

- **Text generation:** Given an input embedding that represents the context or meaning of the text, a decoder can produce a string of words or tokens. For example, a decoder could make a document summary based on an embedding that shows what the document is about.
- **Image creation:** A decoder can be used to create an image if an input embedding represents the features or characteristics of the picture. An embedding that means the pose and appearance of the

person, for instance, could be used by a decoder to create a photo-realistic image of the person.

Encoder-decoder

- **Text translation:** A sentence in one language can be translated into a sentence in another language using an encoder-decoder (also known as machine translation). The decoder produces a translation using an embedding made by the encoder after processing the input sentence.
- **Summarisation:** Given the complete text of a document, an encoder-decoder can produce a summary of the document. So that the decoder can make the summary, the encoder has to process the document and make an embedding.
- **Language modelling:** By using the previous words in the sentence as input, an encoder-decoder can predict the following word in a sentence. The decoder uses the embedding to predict the next word after the encoder has processed the input sentence up to the current word.

State-of-the-art transformers

Since the original transformer model was introduced in the paper “Attention is All You Need” by Vaswani et al., there have been numerous developments in transformers (2017). Following are a few examples of contemporary transformer models created since then:

- BERT (Devlin et al., 2018): Among other natural language processing (NLP) tasks, BERT (Bidirectional Encoder Representations from Transformers) has attained state-of-the-art performance on a variety of NLP tasks, including language understanding, natural language generation, and machine translation. It can accurately capture the context and relationships between words in a sentence because it was trained using a combination of supervised and unsupervised learning.
- GPT-3 (Brown et al., 2020): A transformer-based model called GPT-3 (Generative Pre-training Transformer 3) has achieved cutting-edge performance on various language tasks, including translation, summarisation, question-answering, and text generation. It can write a text that sounds like a human wrote it because it was trained using both supervised and unsupervised learning.
- T5 (Raffel et al., 2020): Using a single, unified architecture, the transformer-based model T5 (Text-To-Text Transfer Transformer) can carry out various natural language processing tasks. It can learn new tasks quickly and effectively without requiring task-specific fine-tuning. It has attained state-of-the-art performance on tasks like language translation, summarisation, and question-answering.

Step9)Bidirectional Encoder Representations from Transformer (BERT)

BERT is an open source machine learning framework for natural language processing (NLP). BERT is designed to help computers understand the meaning of ambiguous language in text by using surrounding text to establish context. The BERT framework was pre-trained using text from Wikipedia and can be fine-tuned with question and answer datasets.

BERT, which stands for Bidirectional Encoder Representations from Transformers, is based on Transformers, a deep learning model in which every output element is connected to every input element, and the weightings between them are dynamically calculated based upon their connection. (In NLP, this process is called *attention*.)

Historically, language models could only read text input sequentially -- either left-to-right or right-to-left -- but couldn't do both at the same time. BERT is different because it is designed to read in both directions at once. This capability, enabled by the introduction of Transformers, is known as bidirectionality.

Using this bidirectional capability, BERT is pre-trained on two different, but related, NLP tasks: Masked Language Modeling and Next Sentence Prediction.

The objective of Masked Language Model (MLM) training is to hide a word in a sentence and then have the program predict what word has been hidden (masked) based on the hidden word's context. The objective of Next Sentence Prediction training is to have the program predict whether two given sentences have a logical, sequential connection or whether their relationship is simply random.

Transformers were first introduced by Google in 2017. At the time of their introduction, language models primarily used recurrent neural networks (RNN) and convolutional neural networks (CNN) to handle NLP tasks.

Although these models are competent, the Transformer is considered a significant improvement because it doesn't require sequences of data to be processed in any fixed order, whereas RNNs and CNNs do. Because Transformers can process data in any order, they enable training on larger amounts of data than ever was possible before their existence. This, in turn, facilitated the creation of pre-trained models like BERT, which was trained on massive amounts of language data prior to its release.

In 2018, Google introduced and open-sourced BERT. In its research stages, the framework achieved groundbreaking results in 11 natural language understanding tasks, including sentiment analysis, semantic role labeling, sentence classification and the disambiguation of polysemous words, or words with multiple meanings.

Completing these tasks distinguished BERT from previous language models such as word2vec and GloVe, which are limited when interpreting context and polysemous words. BERT effectively addresses ambiguity, which is the greatest challenge to natural language understanding according to research scientists in the field. It is capable of parsing language with a relatively human-like "common sense".

In October 2019, Google announced that they would begin applying BERT to their United States based production search algorithms.

BERT is expected to affect 10% of Google search queries. Organizations are recommended not to try and optimize content for BERT, as BERT aims to provide a natural-feeling search experience. Users are advised to keep queries and content focused on the natural subject matter and natural user experience.

In December 2019, BERT was applied to more than 70 different languages.

How BERT works?

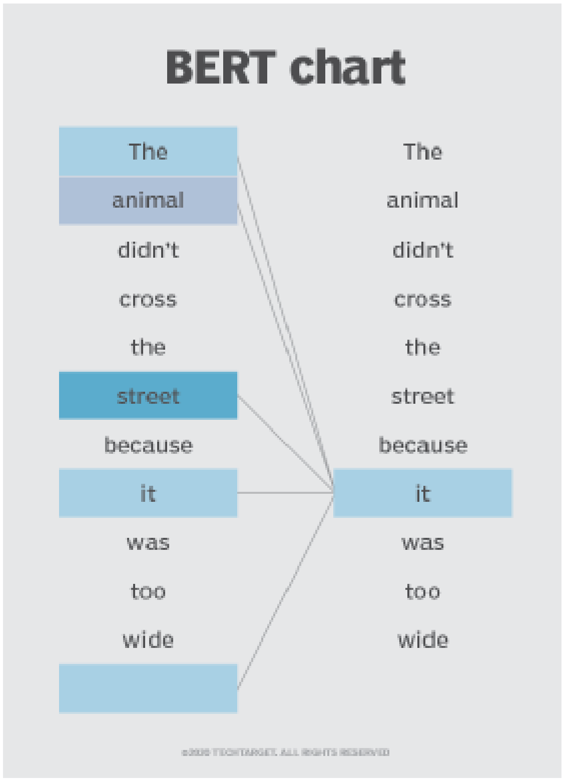
The goal of any given NLP technique is to understand human language as it is spoken naturally. In BERT's case, this typically means predicting a word in a blank. To do this, models typically need to train using a large repository of specialized, labeled training data. This necessitates laborious manual data labeling by teams of linguists.

BERT, however, was pre-trained using only an unlabeled, plain text corpus (namely the entirety of the English Wikipedia, and the Brown Corpus). It continues to learn unsupervised from the unlabeled text and improve even as its being used in practical applications (ie Google search). Its pre-training serves as a base layer of "knowledge" to build from. From there, BERT can adapt to the ever-growing body of searchable content and queries and be fine-tuned to a user's specifications. This process is known as transfer learning.

As mentioned above, BERT is made possible by Google's research on Transformers. The transformer is the part of the model that gives BERT its increased capacity for understanding context and ambiguity in language. The transformer does this by processing any given word in relation to all other words in a sentence, rather than processing them one at a time. By looking at all surrounding words, the Transformer allows the BERT model to understand the full context of the word, and therefore better understand searcher intent.

This is contrasted against the traditional method of language processing, known as word embedding, in which previous models like GloVe and word2vec would map every single word to a vector, which represents only one dimension, a sliver, of that word's meaning.

These word embedding models require large datasets of labeled data. While they are adept at many general NLP tasks, they fail at the context-heavy, predictive nature of question answering, because all words are in some sense fixed to a vector or meaning. BERT uses a method of masked language modeling to keep the word in focus from "seeing itself" -- that is, having a fixed meaning independent of its context. BERT is then forced to identify the masked word based on context alone. In BERT words are defined by their surroundings, not by a pre-fixed identity. In the words of English linguist John Rupert Firth, "You shall know a word by the company it keeps."



BERT is also the *first* NLP technique to rely solely on self-attention mechanism, which is made possible by the bidirectional Transformers at the center of BERT's design. This is significant because often, a word may change meaning as a sentence develops. Each word added augments the overall meaning of the word being focused on by the NLP algorithm. The more words that are present in total in each sentence or phrase, the more ambiguous the word in focus becomes. BERT accounts for the augmented meaning by reading bidirectionally, accounting for the effect of all other words in a sentence on the focus word and eliminating the left-to-right momentum that biases words towards a certain meaning as a sentence progresses.

For example, in the image above, BERT is determining which prior word in the sentence the word "is" referring to, and then using its attention mechanism to weigh the options. The word with the highest calculated score is deemed the correct association (i.e., "is" refers to "animal", not "he"). If this phrase was a search query, the results would reflect this subtler, more precise understanding the BERT reached.

What is BERT used for?

BERT is currently being used at Google to optimize the interpretation of user search queries. BERT excels at several functions that make this possible, including:

Sequence-to-sequence based language generation tasks such as:

- Question answering
- Abstract summarization
- Sentence prediction
- Conversational response generation

Natural language understanding tasks such as:

- Polysemy and Coreference (words that sound or look the same but have different meanings) resolution

    - Word sense disambiguation
    - Natural language inference
    - Sentiment classification

BERT is expected to have a large impact on voice search as well as text-based search, which has been error-prone with Google's NLP techniques to date. BERT is also expected to drastically improve international SEO, because its proficiency in understanding context helps it interpret patterns that different languages share without having to understand the language completely. More broadly, BERT has the potential to drastically improve artificial intelligence systems across the board.

BERT is open source, meaning anyone can use it. Google claims that users can train a state-of-the-art question and answer system in just 30 minutes on a cloud tensor processing unit (TPU), and in a few hours using a graphic processing unit (GPU). Many other organizations, research groups and separate factions of Google are fine-tuning the BERT model architecture with supervised training to either optimize it for efficiency (modifying the learning rate, for example) or specialize it for certain tasks by pre-training it with certain contextual representations. Some examples include:

- patentBERT - a BERT model fine-tuned to perform patent classification.

- docBERT - a BERT model fine-tuned for document classification.
- bioBERT - a pre-trained biomedical language representation model for biomedical text mining.
- VideoBERT - a joint visual-linguistic model for process unsupervised learning of an abundance of unlabeled data on Youtube.
- SciBERT - a pretrained BERT model for scientific text
- G-BERT - a BERT model pretrained using medical codes with hierarchical representations using graph neural networks (GNN) and then fine-tuned for making medical recommendations.
- TinyBERT by Huawei - a smaller, "student" BERT that learns from the original "teacher" BERT, performing transformer distillation to improve efficiency. TinyBERT produced promising results in comparison to BERT-base while being 7.5 times smaller and 9.4 times faster at inference.
- DistilBERT by HuggingFace - a supposedly smaller, faster, cheaper version of BERT that is trained from BERT, and then certain architectural aspects are removed for the sake of efficiency.

Step 10: Large Language Models (LLMs)

A large language model (LLM) is a deep learning algorithm that can perform a variety of natural language processing (NLP) tasks. Large language models use transformer models and are trained using massive datasets — hence, large. This enables them to recognize, translate, predict, or generate text or other content.

In addition to teaching human languages to artificial intelligence (AI) applications, large language models can also be trained to perform a variety of tasks like understanding protein structures, writing software code, and more. Like the human brain, large language models must be pre-trained and then fine-tuned so that they can solve text classification, question answering, document summarization, and text generation problems. Their problem-solving capabilities can be applied to fields like healthcare, finance, and entertainment where large language models serve a variety of NLP applications, such as translation, chatbots, AI assistants, and so on.

Large language models also have large numbers of parameters, which are akin to memories the model collects as it learns from training. Think of these parameters as the model’s knowledge bank.

Large language models like GPT-3 and GPT-4 have gained popularity in the field of NLP. These models are capable of generating human-like text and can be fine-tuned for various NLP tasks.

To work with LLMs, you can use libraries like OpenAI's GPT or Facebook's Llama 2. These libraries provide interfaces and utilities for working with LLMs and integrating them into your NLP applications.

Step 11: LLM Libraries

As Large Language Models (LLMs) like GPT variants continue to dominate the NLP space, the need for specialized libraries that can streamline the implementation of these models is ever-increasing. Two such libraries that have gained considerable traction are the LangChain and Llama Index. Here's what you need to know about these popular tools and how they can supercharge your NLP projects.

LangChain: Your One-Stop Shop for LLM Applications

LangChain is a framework for developing applications powered by language models. LangChain offers a comprehensive suite of utilities designed to simplify the building of applications around Large Language Models. Whether you're looking to chunk PDF files, interface with vector databases, or execute more complex tasks like Natural Language to SQL conversions, LangChain has got you covered.

It enables applications that:

- **Are context-aware**: connect a language model to sources of context (prompt instructions, few shot examples, content to ground its response in, etc.)
- **Reason**: rely on a language model to reason (about how to answer based on provided context, what actions to take, etc.)

This framework consists of several parts.

- LangChain Libraries: The Python and JavaScript libraries. Contains interfaces and integrations for a myriad of components, a basic run time for combining these components into chains and agents, and off-the-shelf implementations of chains and agents.
- LangChain Templates: A collection of easily deployable reference architectures for a wide variety of tasks.
- LangServe: A library for deploying LangChain chains as a REST API.
- LangSmith: A developer platform that lets you debug, test, evaluate, and monitor chains built on any LLM framework and seamlessly integrates with LangChain.

Features:

- PDF Chunking: Efficiently divide large PDF files into manageable pieces.
- Interface with LLMs: Seamless integration with popular Large Language Models like those from OpenAI.
- LangChain SQL Agent: A specialized component for Natural Language to SQL conversions.

Llama Index: A Flexible Library for Data Augmentation and Indexing

Llama Index serves a somewhat overlapping but distinct role compared to LangChain. Its core strength lies in connecting to various data sources and indexing documents to augment the capabilities of Large Language Models.

Features:

- Multiple Connectors: Easily connect to data sources like Google Docs, Notion, and PDF files.
- Advanced Indexing: Offers multiple methods to index documents, from simple list indexes to more complex tree structures and table keyword indexes.

Use-Cases:

Llama Index is ideal for projects that require advanced semantic search capabilities, thanks to its robust indexing features.