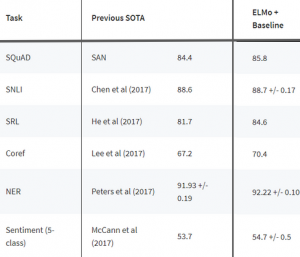
Types of Word embeddings:

1. ELMO
2. BERT
3. Fast Text
4. Word2Vec
5. GloVe

**ELMO:**

ELMO is a novel way to represent words in vectors or embeddings. These word embeddings are helpful in achieving state-of-the-art (SOTA) results in several NLP tasks:

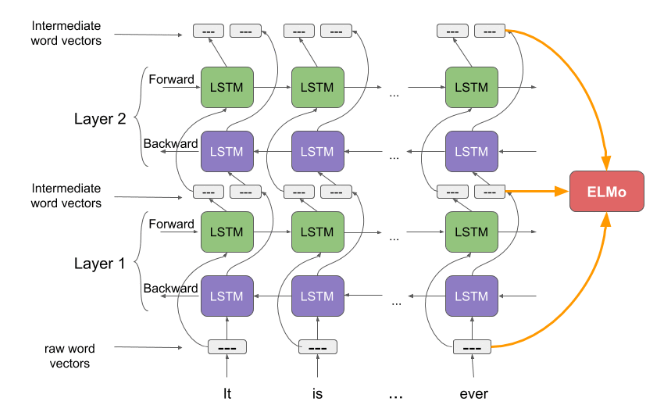


## Understanding how ELMo works

Let’s get an intuition of how ELMo works underneath before we implement it in Python. Why is this important?

Now, let’s come back to how ELMo works.

As I mentioned earlier, ELMo word vectors are computed on top of a two-layer bidirectional language model (biLM). This biLM model has two layers stacked together. Each layer has 2 passes — forward pass and backward pass:



The architecture above uses a character-level convolutional neural network (CNN) to represent words of a text string into raw word vectors

These raw word vectors act as inputs to the first layer of biLM

The forward pass contains information about a certain word and the context (other words) before that word

The backward pass contains information about the word and the context after it

This pair of information, from the forward and backward pass, forms the intermediate word vectors

These intermediate word vectors are fed into the next layer of biLM

The final representation (ELMo) is the weighted sum of the raw word vectors and the 2 intermediate word vectors.

**How is ELMo different from other word embeddings?**

Unlike traditional word embeddings such as [word2vec](https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/?utm_source=ELMoNLParticle&utm_medium=blog)and [GLoVe](https://nlp.stanford.edu/projects/glove/), the ELMo vector assigned to a token or word is actually a function of the entire sentence containing that word. Therefore, the same word can have different word vectors under different contexts.

I can imagine you asking – how does knowing that help me deal with NLP problems? Let me explain this using an example.

Suppose we have a couple of sentences:

I *read* the book yesterday.

Can you *read* the letter now?

Take a moment to ponder the difference between these two. The verb “read” in the first sentence is in the past tense. And the same verb transforms into present tense in the second sentence. This is a case of Polysemy wherein a word could have multiple meanings or senses.

*Language is such a wonderfully complex thing.*

Traditional word embeddings come up with the same vector for the word “read” in both the sentences. Hence, the system would fail to distinguish between the polysemous words. These word embeddings just cannot grasp the context in which the word was used.

ELMo word vectors successfully address this issue. ELMo word representations take the entire input sentence into equation for calculating the word embeddings. Hence, the term “read” would have different ELMo vectors under different context.

**Pseudo code:**

Import tensorflow\_hub as hub

import tensorflow as tf

elmo = hub.Module("https://tfhub.dev/google/elmo/2", trainable=True)

References:

<https://www.analyticsvidhya.com/blog/2019/03/learn-to-use-elmo-to-extract-features-from-text/>

**BERT:**

First, it’s easy to get that BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. Each word here has a meaning to it and we will encounter that one by one in this article. For now, the key take away from this line is – **BERT is based on the Transformer architecture.**

Second, BERT is pre-trained on a large corpus of unlabelled text including the entire Wikipedia(that’s 2,500 million words!) and Book Corpus (800 million words).

Third, BERT is a **“deeply bidirectional”** model. Bidirectional means that BERT learns information from both the left and the right side of a token’s context during the training phase.

The bidirectionality of a model is important for truly understanding the meaning of a language. Let’s see an example to illustrate this. There are two sentences in this example and both of them involve the word “bank”:

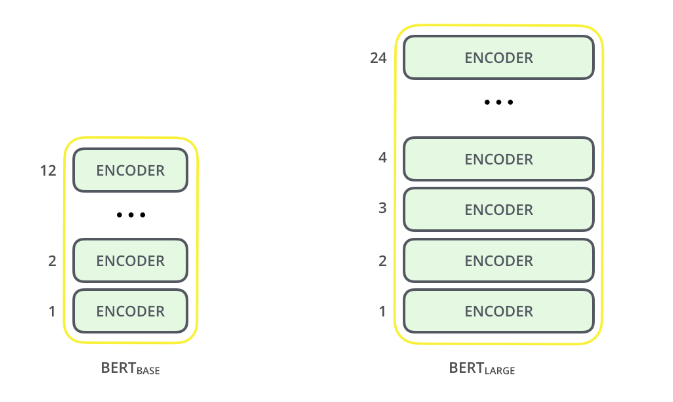
If we try to predict the nature of the word “bank” by only taking either the left or the right context, then we will be making an error in at least one of the two given examples.

One way to deal with this is to consider both the left and the right context before making a prediction. That’s exactly what BERT does! We will see later in the article how this is achieved.

And finally, the most impressive aspect of BERT. We can fine-tune it by adding just a couple of additional output layers to create state-of-the-art models for a variety of NLP tasks.

he BERT architecture builds on top of Transformer. We currently have two variants available:

* BERT Base: 12 layers (transformer blocks), 12 attention heads, and 110 million parameters
* BERT Large: 24 layers (transformer blocks), 16 attention heads and, 340 million parameters

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/09/bert_encoder.png)

The BERT Base architecture has the same model size as OpenAI’s GPT for comparison purposes. All of these Transformer layers are **Encoder**-only blocks.

**Pseudo code:**

Installing BERT-As-Service

pip install bert-serving-server # server

pip install bert-serving-client # client

From bert\_serving.client import BertClient

Bc = BertClient(ip=”server\_ip\_here”)

Embedding = bc.encode(“input text”)

|  |
| --- |
|  |
|  |  |
|  |  |
|  |  |

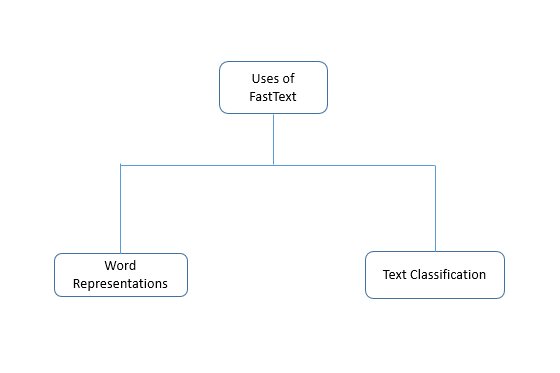
References:

<https://www.analyticsvidhya.com/blog/2019/09/demystifying-bert-groundbreaking-nlp-framework/>

**Fast Text:**

What is FastText?

FastText is a library created by the Facebook Research Team(FLAIR) for efficient learning of word representations and sentence classification.



**How is FastText different from gensim Word Vectors?**

FastText differs in the sense that word vectors a.k.aword2vec treats every single word as the smallest unit whose vector representation is to be found but FastText assumes a word to be formed by a n-grams of character, for example, sunny is composed of [sun, sunn,sunny],[sunny,unny,nny]  etc, where n could range from 1 to the length of the word. This new representation of word by fastText provides the following benefits over word2vec or glove.

Helpful to find rare words.

It can give the vector representations for the words not present in the dictionary (OOV words) since these can also be broken down into character n-grams. word2vec and glove both fail to provide any vector representations for words not in the dictionary.

character n-grams embeddings tend to perform superior to word2vec and glove on smaller datasets.

**Pseudo Code:**

From gensim.models import FastText

Model\_ted = FastText(sentences\_ted, size=100, window=5, min\_count=5, workers=4, sg=1)

Mdoel\_ted.wv.most\_similar(“input text”)

References:

<https://www.tutorialkart.com/fasttext/train-and-test-supervised-text-classifier-using-fasttext/>

<https://medium.com/@ageitgey/text-classification-is-your-new-secret-weapon-7ca4fad15788>

<https://arxiv.org/pdf/1807.02892.pdf>

**Word2Vec:**

Word2Vec is one of the most popular pretrained word embeddings developed by Google. Word2Vec is trained on the Google News dataset (about 100 billion words).

The architecture of Word2Vec is really simple. It’s a feed-forward neural network with just one hidden layer. Hence, it is sometimes referred to as a **Shallow Neural Network architecture**.

Word2Vec is classified into two approaches:

* Continuous Bag-of-Words (CBOW)
* Skip-gram model

Continuous Bag-of-Words (CBOW) model learns the focus word given the neighbouring words whereas the Skip-gram model learns the neighbouring words given the focus word. That’s why:

Continuous Bag Of Words and Skip-gram are inverses of each other.

For example, consider the sentence: “I have failed at times but I never stopped trying”.  Let’s say we want to learn the embedding of the word “failed”. So, here the focus word is “failed”.

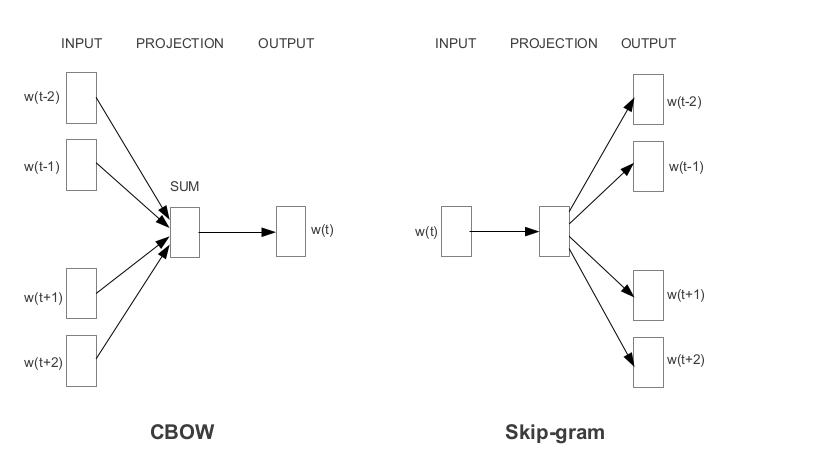
The first step is to define a context window. A context window refers to the number of words appearing on the left and right of a focus word. The words appearing in the context window are known as neighbouring words (or context). Let’s fix the context window to 2 and then input and output pairs for both approaches:

Continuous Bag-of-Words: Input = [ I, have, at, times ],  Output = failed

Skip-gram: Input = failed, Output = [I, have, at, times ]

As you can see here, CBOW accepts multiple words as input and produces a single word as output whereas Skip-gram accepts a single word as input and produces multiple words as output.

So, let us define the architecture according to the input and output. But keep in mind that each word is fed into a model as a one-hot vector:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/03/Screenshot-from-2020-03-12-13-05-42.png)

**Advantages of CBOW:**

Being probabilistic is nature, it is supposed to perform superior to deterministic methods(generally).

It is low on memory. It does not need to have huge RAM requirements like that of co-occurrence matrix where it needs to store three huge matrices.

Advantages of Skip-Gram Model

Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit.

Skip-gram with negative sub-sampling outperforms every other method generally

**Pseudo code:**

from gensim.models import Word2Vec  
model = Word2Vec.load\_word2vec\_format('GoogleNews-vectors-negative300.bin', binary=True, norm\_only=True)

model = gensim.models.Word2Vec(sentence, min\_count=1,size=300,workers=4)  
print(model.most\_similar(positive=['input text'], negative=['input text']))  
print(model.doesnt\_match("input text".split()))  
print(model.similarity('input text', 'man'))

Parameters:

min\_count=1 -the threshold value for the words. Word with frequency greater than this only are going to be included into the model.  
size=300 – the number of dimensions in which we wish to represent our word. This is the size of the word vector.  
workers=4 – used for parallelization

**GloVe:**

The basic idea behind the GloVe word embedding is to derive the relationship between the words from Global Statistics.

One of the simplest ways is to look at the co-occurrence matrix. A co-occurrence matrix tells us how often a particular pair of words occur together. Each value in a co-occurrence matrix is a count of a pair of words occurring together.

For example, consider a corpus: “I play cricket, I love cricket and I love football”. The co-occurrence matrix for the corpus looks like this:



One of the simplest ways is to look at the co-occurrence matrix. A co-occurrence matrix tells us how often a particular pair of words occur together. Each value in a co-occurrence matrix is a count of a pair of words occurring together.

For example, consider a corpus: “I play cricket, I love cricket and I love football”. The co-occurrence matrix for the corpus looks like this:



Now, we can easily compute the probabilities of a pair of words. Just to keep it simple, let’s focus on the word “cricket”:

p(cricket/play)=1

p(cricket/love)=0.5

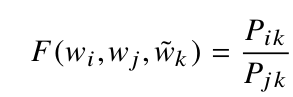
Next, let’s compute the ratio of probabilities:

p(cricket/play) / p(cricket/love) = 2

As the ratio > 1, we can infer that the most relevant word to cricket is “play” as compared to “love”. Similarly, if the ratio is close to 1, then both words are relevant to cricket.

We are able to derive the relationship between the words using simple statistics. This the idea behind the GloVe pretrained word embedding.

GloVe learns to encode the information of the probability ratio in the form of word vectors. The most general form of the model is given by:



**Pseudo code:**

embeddings\_index=dict()

f =open('../input/glove6b/glove.6B.300d.txt')

model=Sequential()

model.add(Embedding(size\_of\_vocabulary,300,weights=[embedding\_matrix],input\_length=100,trainable=False))

|  |
| --- |
| model.add(GlobalMaxPooling1D()) |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |