NLP: Word Embedding Techniques you should know

> Explanation of different Embedding techniques with Hands on, applications and pros and cons

In our previous blog , we discussed Text Pre-processing techniques that is the first step we usually do for any NLP use case, second to Pre-processing we have Word Embeddings that we will discuss further in this blog.

In natural language processing (NLP), an embedding technique is a way of representing words or text data as vectors of numbers that capture the underlying semantic meaning of the words. The idea is to transform the textual data into a numerical representation that can be used as input to machine learning algorithms.

The need for embedding techniques in NLP arises from the fact that machine learning algorithms typically work with numerical data, and therefore require a numerical representation of text data to be able to learn from it. By converting text data into numerical embeddings, we can train machine learning models to perform a variety of NLP tasks, such as sentiment analysis, text classification, and machine translation. Embeddings also allow us to capture the semantic meaning of words and sentences, which is important for many NLP tasks that involve understanding the meaning of natural language text.

Categories of Embedding Techniques

1. Frequency or Count-based embedding techniques: These techniques use the frequency of words or their co-occurrence with other words in a large corpus to generate word embeddings. Techniques such Bag of Words(BOW) and TF-IDF comes under this category.
2. Prediction-based embedding techniques: These techniques use neural networks to predict the context of a word based on its neighbouring words. These embeddings can be generated using methods such as Word2Vec, Continuous Bag-of-Words (CBOW), and Skip-gram. Prediction-based embeddings are often used for tasks such as language modelling and machine translation.

We will discuss briefly on all these techniques with their code implementation, applications, pros and cons.

1. -The bag-of-words technique is a fundamental and simple method used in Natural Language Processing (NLP) for extracting features from text data. The main idea behind this technique is to represent a text document as a bag (multiset) of its words, disregarding grammar and word order but keeping track of word frequency.

Input:

```
from sklearn.feature_extraction.text import CountVectorizer

# Define the documents
doc1 = "The quick brown fox jumps over the lazy dog."
doc2 = "Early bird catches the worm."
doc3 = "A stitch in time saves nine."

# Preprocess the documents
docs = [doc1, doc2, doc3]

# Create the vectorizer and fit it to the documents
vectorizer = CountVectorizer(stop_words='english')
```

```
bow_vectors = vectorizer.fit_transform(docs)

# Get the vocabulary
vocab = vectorizer.get_feature_names_out()

(, vocab) i, doc  (docs):    (, i+, , bow_vectors[i].toarray())
```

Output:

```
Vocabulary:  [, , , , , , , , , , , , , ]BOW vector  document  :  [[
]]BOW vector  document  :  [[             ]]BOW vector  document  :  [[
]]
```

Advantages:

1. Simple and Intuitive
2. Language agnostic, i.e., it can be applied to any language with some preprocessing

Disadvantages:

1. Can result in a high-dimensional and sparse feature vector representation, which can be computationally expensive and require a lot of memory.
2. Vulnerable to noise in the data, such as spelling errors and grammatical mistakes, which can negatively impact the accuracy of the model.
3. Ignores the order of words in the text, leading to a loss of contextual information.
4. Not able to capture semantic meaning of the words.

Applications: Sentiment Analysis, Topic Modeling, Text Classification, Information Retrieval, Machine Translation.

Concept of N-grams:

In NLP, n-grams are a contiguous sequence of n items, typically words or characters, that occur in a piece of text. An n-gram model is a statistical language model that predicts the probability of the next word in a sequence based on the n-grams of the previous words. For example, in the sentence "The quick brown fox jumps over the lazy dog," some examples of n-grams are:

- Unigrams (n=1): The, quick, brown, fox, jumps, over, the, lazy, dog
- Bigrams (n=2): The quick, quick brown, brown fox, fox jumps, jumps over, over the, the lazy, lazy dog
- Trigrams (n=3): The quick brown, quick brown fox, brown fox jumps, fox jumps over, jumps over the, over the lazy, the lazy dog

The choice of the value of n depends on the specific task and the size of the text corpus. N-grams are widely used in NLP for tasks such as text classification, language modeling, and machine translation.

In BOW , We use Count Vectorizer model to convert text to vectors , we can tune this model with a parameter n-gram depending upon the use case to capture more semantic meaning of the words.

2. **TF-IDF** — TF-IDF stands for *Term Frequency-Inverse Document Frequency*, and it is a technique used in natural language processing (NLP) to measure the importance of words in a document or corpus.

In simple terms, TF-IDF gives a score to each word in a document based on how often it appears in the document (term frequency) and how often it appears in the corpus (inverse document frequency). The intuition behind this is that words that appear frequently in a document but rarely in the corpus are more important than words that appear frequently in both.

First, let's define some terms:

- `t`: a term (word) in the document
- `d`: a document in the corpus
- `N`: the total number of documents in the corpus
- `n_t`: the number of documents in the corpus that contain the term `t`

The formula for TF-IDF is:

```
TF-IDF(t, d) = TF(t, d) * IDF(t)
```

- `TF(t, d)` is the term frequency of `t` in `d`. It is calculated as the number of times `t` appears in `d`, divided by the total number of terms in `d`. This gives us a measure of how often the term `t` appears in the document `d`.

```
TF(t, d) = (number of occurrences of t in d) / (total number of terms in d)
```

- `IDF(t)` is the inverse document frequency of `t`. It is calculated as the logarithm of the total number of documents in the corpus `N`, divided by the number of documents in which `t` appears. This gives us a measure of how important the term `t` is in the corpus as a whole.

```
IDF(t) = log(N / n_t)
```

The logarithm is used to dampen the effect of very common words, which would otherwise dominate the score.

Multiplying the term frequency `TF(t, d)` and the inverse document frequency `IDF(t)` gives us the TF-IDF score for the term `t` in the document `d`. This score is higher when the term `t` appears frequently in the document `d`, and when it appears rarely in the corpus as a whole.

By calculating the TF-IDF score for each term in each document, we can compare documents and identify those that are most relevant to a given query. We can also cluster similar documents together based on their TF-IDF scores.

Overall, TF-IDF is a powerful technique for measuring the importance of words in a document or corpus, and it has many applications in natural language processing and text mining.

Input:

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Define the documents
doc1 = "The quick brown fox jumped over the lazy dog"
doc2 = "The dog and the fox were good friends"
doc3 = "The quick brown cat jumped over the lazy dog"
```

```
# Create a list of the documents
documents = [doc1, doc2, doc3]

# Create an instance of the TfidfVectorizer class
vectorizer = TfidfVectorizer()

# Fit and transform the documents using the vectorizer
tfidf_matrix = vectorizer.fit_transform(documents)

# Print the vocabulary
vocab = vectorizer.get_feature_names_out()

 i, doc  (documents):    ()    (tfidf_matrix.toarray()[i])
```

The `get_feature_names` method of the `TfidfVectorizer` class returns a list of the unique terms that appear in the documents. By default, the vectorizer converts all terms to lowercase and removes punctuation and stop words.

Output:

```
Vocabulary: ['and', 'brown', 'cat', 'dog', 'fox', 'friends', 'good', 'jumped',
'lazy', 'over', 'quick', 'the', 'were']

Document 1: The quick brown fox jumped over the lazy dog
[0.394248620.0.0.394248620.394248620.
0.294377090.0.0.0.0.
0.        ]



Document 2: The dog and the fox were good friends
[0.0.0.449436420.0.0.3595293
0.0.0.449436420.0.0.
0.        ]


Document : The quick brown cat jumped over the lazy dog[
]
```

The first line of output shows the vocabulary of the vectorizer, which consists of 13 unique terms after removing stop words and punctuation. The output then shows the corresponding text for each row of the TF-IDF matrix, along with the TF-IDF values for each term in that document.

Advantages:

1. Accounts for word importance: TF-IDF considers the importance of a word in a document by weighting it based on its frequency in the document and rarity in the corpus.
2. Reduces noise: Words that appear frequently across all documents (e.g., “the”, “and”) have a lower score and thus are less likely to contribute to the model’s predictions. This helps to reduce noise and improve the model’s accuracy.

Disadvantages:

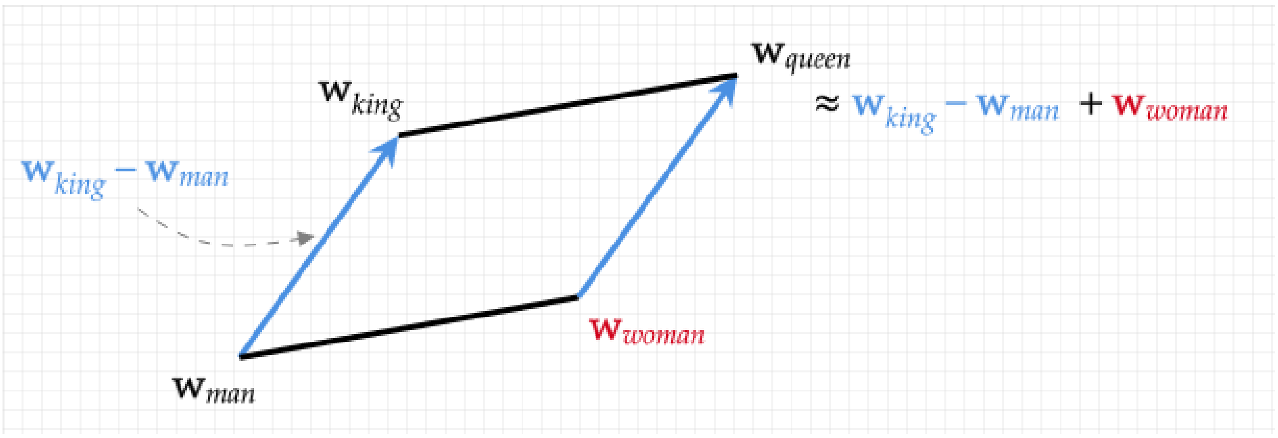
1. Does not capture word order
2. Sparsity
3. Out of Vocabulary

Applications: Information retrieval, Text classification, Keyword extraction, Recommender systems, Document clustering.

3. **Word2Vec-**Word2Vec is a widely used technique in natural language processing (NLP) that is used to represent words as vectors in a high-dimensional space. It is a type of neural network that can learn to predict the context of a word, and it does so by training on large amounts of text data.

The key idea behind Word2Vec is that words that are used in similar contexts tend to have similar meanings. By training a neural network to predict the context of a word, the model can learn to group together words that are semantically similar.

Example: *If King and Queen are related , Man and Women Are related*


$\mathbf{w}_{queen}$
$\approx \mathbf{w}_{king} - \mathbf{w}_{man} + \mathbf{w}_{woman}$
$\mathbf{w}_{king}$
$\mathbf{w}_{king} - \mathbf{w}_{man}$
$\mathbf{w}_{woman}$
$\mathbf{w}_{man}$

[]

King — Man + Woman= Queen(Can show similar representations hence, Semantic Meaning is captured)

Word2vec can utilize either of two model architectures to produce these distributed representations of words: continuous-bag-of-words (CBOW) or continuous skip-gram. CBOW (Continuous Bag of Words) and Skip-gram. In CBOW, the model is trained to predict a word given its context, while in Skip-gram, the model is trained to predict the context given a word. In both architectures, word2vec considers both individual words and a sliding window of context words surrounding individual words as it iterates over the entire corpus. Lets understand both of them Briefly.

1.

The objective function of CBOW model is to predict the middle word as target when given past N/2 history words and N/2 future words. Eg-"*Pineapples are spikey and yellow*". In this case "*spikey*" is the middle or target word that need to predicted and our context words to be ["*Pineapples*", "*are*", "*and*", "*yellow*"].

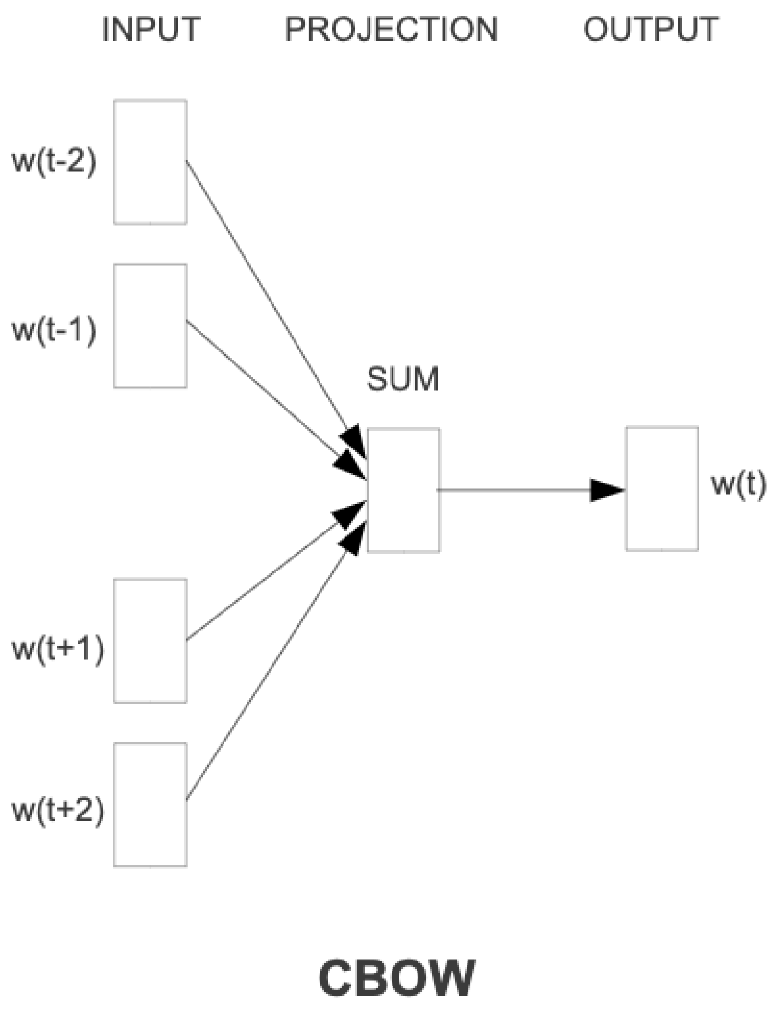
Note: The input selected as an window size from a corpus of text

During training, the input layer feeds the one-hot encoded vectors for the context words into the hidden layer(s)(projection layer). In the projection layer, word vectors of context words are simply averaged as a compressed representation. The output layer takes this compressed representation and produces the embedding vector for the target word. We denote this model further as CBOW, as unlike standard bag-of-words model, it uses continuous distributed representation of the context.

The number of hidden layers in projection layer and the number of neurons in each layer are hyperparameters that can be tuned during training to improve the performance of the model.

The embedding vectors are learned by minimizing the loss function, which measures the discrepancy between the predicted and actual target words. The weights of the network are updated using backpropagation and stochastic gradient descent.

Once the model has been trained, the embedding vectors can be used to represent words in a continuous vector space that captures their semantic relationships



[]

2.

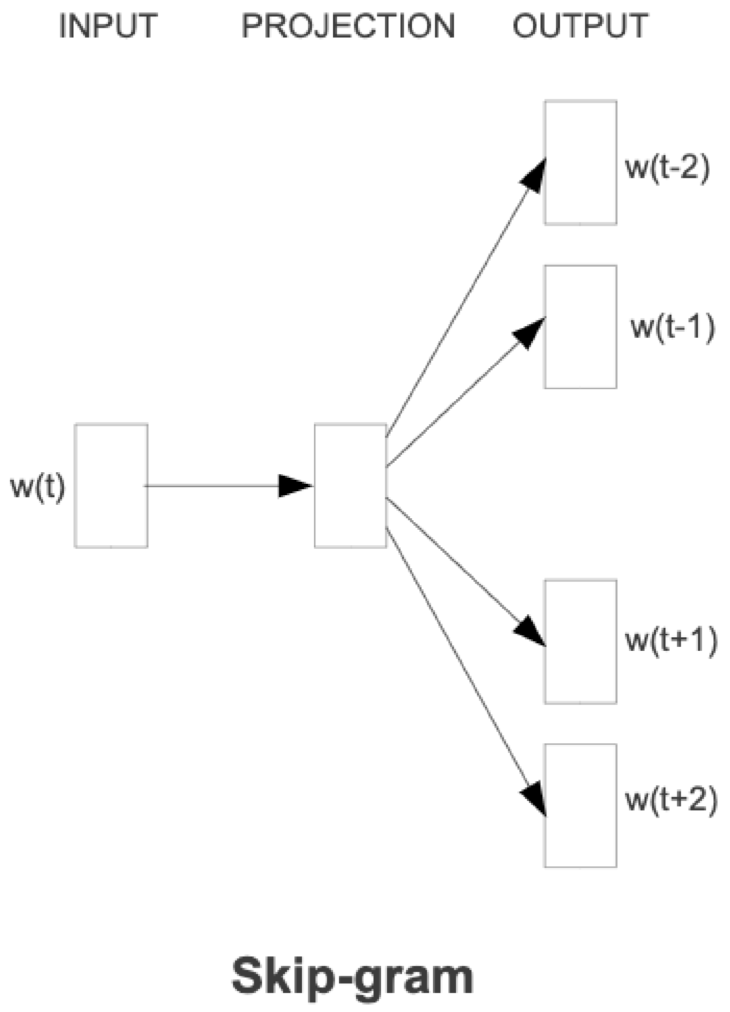
This model reverses an objective of CBOW model. Given the current word as an input(“spikey”), it predicts the nearby context words within a certain range both in history and future([“Pineapples”, “are”, “and”, “yellow”]).We found that increasing the range improves quality of the resulting word vectors, but it also increases the computational complexity. Since the more distant words are usually less related to the current word than those close to it, we give less weight to the distant words by sampling less from those words in our training examples.

During training, the input layer feeds the one-hot encoded vector for the target word into the hidden layer(s) (Projection Layer). The Projection layer then transform the input vector into a compressed representation(simple average) that captures the context of the target word. The output layer takes this compressed representation and produces the one-hot encoded vectors for the context words.

The number of hidden layers and the number of neurons in each layer are hyperparameters that can be tuned during training to improve the performance of the model.

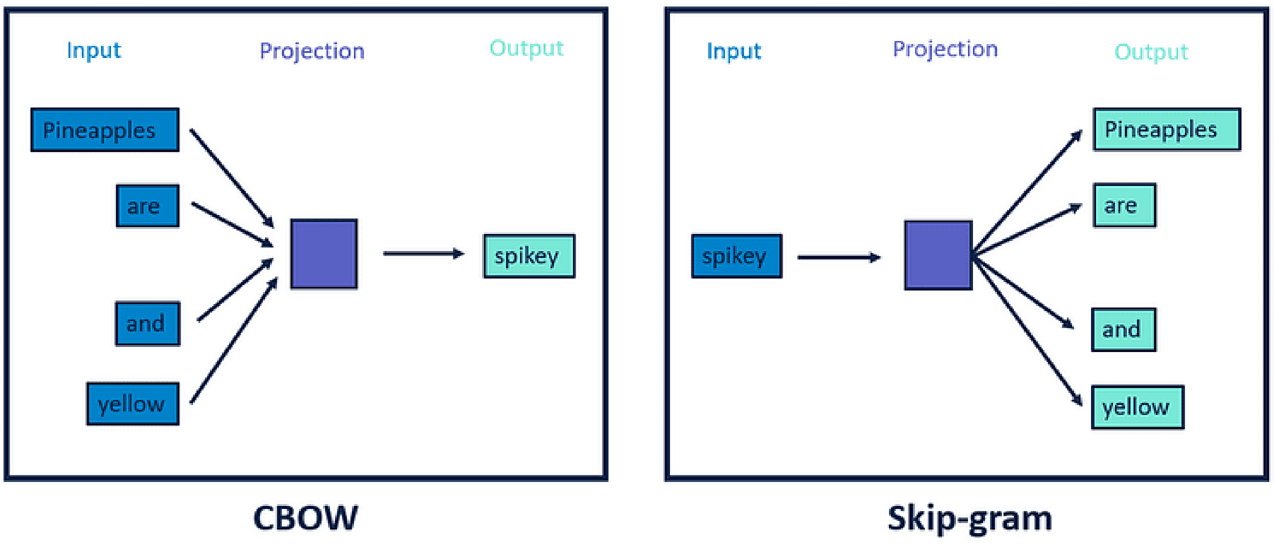
The weights of the network are learned by minimizing the loss function, which measures the discrepancy between the predicted and actual context words. The weights are updated using backpropagation and stochastic gradient descent.

Once the model has been trained, the embedding vectors can be used to represent words in a continuous vector space that captures their semantic relationships.



[]

Example representation of Both CBOW and Skip-Gram.



[]

Input:

```
from gensim.models import Word2Vec
import nltk
nltk.download('punkt')

# Sample input text
sentences = [
"The quick brown fox jumps over the lazy dog",
"I have a cat named Bob",
"Mary had a little lamb",
"The cat chased the mouse"
]



# Tokenize the sentences
tokenized_sentences = [nltk.word_tokenize(sentence.lower()) for sentence in
sentences]

# Define the CBOW model
model_cbow = Word2Vec(sentences=tokenized_sentences, vector_size=10, window=2,
sg=0, min_count=1, workers=4, epochs=100)

# Get the vocabulary and corresponding vectors
vocabulary = model_cbow.wv.index_to_key
vectors = model_cbow.wv[vocabulary]

(, vocabulary)(, vectors)
```

Output:

```
Vocabulary: [, , , , , , , , , , , , , , , , , , ]Vectors: [[-       - -       -
-] [  - -   - -         - - ] [               -  -   - -] [- -   - - -    - - -]
[-   - -   -  -       ] [-   -     -    - -  ] [-   - - -    -    -  ] [    - -
-  - -] [-       -      - -  ] [             - -    ] [-        -      - -] [-
- -            - -] [  -   -       -   -  ] [  -  - -            ] [  - - - -
-    ] [  - -  -        -  ] [-   -  -        -    ] [- -     -  -  - -    - ]
[-      -   -     -   -] [  -   - -      - -  ]]
```

Advantages:

1. Vectors created are of Limited Dimensions.
2. Sparsity will be reduced.
3. Semantic meaning between vectors is maintained.

Disadvantage:

1. Word2Vec models require large amounts of data to train effectively, which may not be available in some cases.
2. Word2Vec models may not perform as well on domain-specific or specialized vocabularies, as they are typically trained on large and generic corpora.
3. Lack of interpretability
4. Word2Vec is limited by the context window size, meaning that it may not be able to capture relationships between words that are separated by a large number of words.

Applications: Text Classification, Recommendation Systems, Search Engines, Question Answering Systems, Chatbots, Machine Translation, Named Entity Recognition.

3. **Average Word2Vec**

Average word2vec is another way to represent words in a vector space, similar to word2vec. However, instead of using a neural network to predict the context words given a target word or vice versa, it uses a simpler approach that averages the word vectors of all the words in a sentence or document to get the overall vector representation.

The advantage of using average word2vec over word2vec is that it doesn't suffer from the limitation of being able to represent only individual words and their relationships with other words. Instead, it can capture the overall meaning of a sentence or document by taking into account all the words present in it. Additionally, it doesn't require training a neural network, making it a simpler and faster approach.

Input:

```
import gensim
import numpy as np

# Define sentences
sentences = [["apple", "banana", "orange"], ["orange", "grape", "apple"],
["banana", "grape", "cherry"]]

# Train word2vec model
model = gensim.models.Word2Vec(sentences, min_count=1)

# Define function to get vector representation of a sentence
defget_sentence_vector(model, sentence):
    vectors = []
```

```
for word in sentence:
if word in model.wv.index_to_key:
            vectors.append(model.wv[word])
iflen(vectors) > 0:
return np.mean(vectors, axis=0)
else:
return np.zeros((model.vector_size,), dtype=np.float32)



# Get vector representation for each sentence
vectors = []
for sentence in sentences:
    vectors.append(get_sentence_vector(model, sentence))
```

```
(, (model.wv.index_to_key))(, np.array(vectors))
```

Output:

```
Vocabulary: [, , , , ]Vectors: [[-   -      -      - -   -        -      -    - -
-           -     -  - - -             - -          -  - -   -  - - -        -
-  -     -           -     -      - -  -   - -  -      -    -   -    - -      -
- - -] [-         -      - -   -  -            - -    -              -     -
-        -          - -               - -           -   - -
-   -        -  -   - -  -        - -   -  - - - -    -        - -  ] [-   -
-   - -  -   - -      - -    -        -      -     -    - - -        -  - - -
- -        -    - - - -  - - - -   -   -  -        -        - -            -  -
-      - -   -  -          -        -    ]]
```

Advantages:

1. It captures the semantic relationship between words.
2. It produces dense and continuous word embeddings that are more useful for machine learning models.
3. It can handle out-of-vocabulary words by averaging the vectors of the words in the sentence.
4. It requires less computational power compared to other deep learning techniques.

Disadvantages:

1. It may not capture the nuances of the context as effectively as other techniques like LSTM or Transformer models.
2. It requires a large amount of text data to produce accurate embeddings.
3. It does not capture the order of words in the sentence, which can be important in some NLP tasks.
4. It may not work well for languages with complex grammatical structures or with a large number of inflections.

Applications: Text classification, Search and recommendation engines, Machine translation, Named entity recognition, Question answering, Chatbots.

Post-padding and Pre-padding in NLP

In NLP, padding refers to the process of adding extra tokens to the beginning or end of a sentence or sequence to make it a fixed length. Pre-padding refers to adding tokens to the beginning of a sequence, while post-padding refers to adding tokens to the end of a sequence.

For example, suppose we have a dataset of sentences with varying lengths, and we want to use them as input to a neural network that requires fixed-length input. We can pad the shorter sentences with zeros at the end to make them the same length as the longest sentence in the dataset.

Pre-padding would involve adding zeros to the beginning of the shorter sentences, while post-padding would involve adding zeros to the end of the shorter sentences.

Padding is important in NLP because it allows us to handle variable-length inputs in a consistent manner and enables the use of batch processing for training and inference.

Here's an example code snippet that demonstrates how to use post and pre padding in NLP using the Keras library in Python:

Input:

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Define sequences
sequences = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]

# Pad sequences to a maximum length of 5
padded_sequences_post = pad_sequences(sequences, maxlen=5, padding='post')
padded_sequences_pre = pad_sequences(sequences, maxlen=5, padding='pre')

(, padded_sequences_post)(, padded_sequences_pre)
```

In this example, we define a list of sequences and then use the `pad_sequences` function from the Keras library to pad the sequences with zeros to a maximum length of 5. We use the `padding` parameter to specify whether to add the padding at the beginning (pre) or end (post) of the sequences.

Output:

```
Post-padded sequences:
 [[12300]
 [45000]
 [67890]]

            Pre-padded sequences: [[    ] [    ] [    ]]
```

As we can see, the `pad_sequences` function has added zeros to the end of the sequences in the post-padded version, and to the beginning of the sequences in the pre-padded version, to bring them up to the desired length of 5.

Facts

The above embedding techniques are being used since long in the industries but Recently, there has been a growing trend towards using ELMO and transformer-based models such as BERT (Bidirectional Encoder

Representations from Transformers) and GPT (Generative Pretrained Transformer) for generating contextualized word embeddings. These models use self-attention mechanisms to generate embeddings that capture the meaning of a word in the context of the entire sentence or document. These embeddings have been shown to be very effective for a wide range of NLP tasks, including sentiment analysis, question answering, and named entity recognition.

We will further discuss about trending ELMO and transformer based Embedding technique in our future blogs.

Summary: The blog will take you through some of most common Embedding techniques that have been explained in well structure way with respective images for better understanding along with their pros, cons and applications with a hands on stuff which will be help beginners in better understanding.

Resources:

[1] Original Research Paper-https://arxiv.org/pdf/1301.3781.pdf

Thanks for taking out time to read the post. I hope it was helpful. Please let me know what are your thoughts/ comments. Feel free to reach out to me if you have any queries and suggestions.