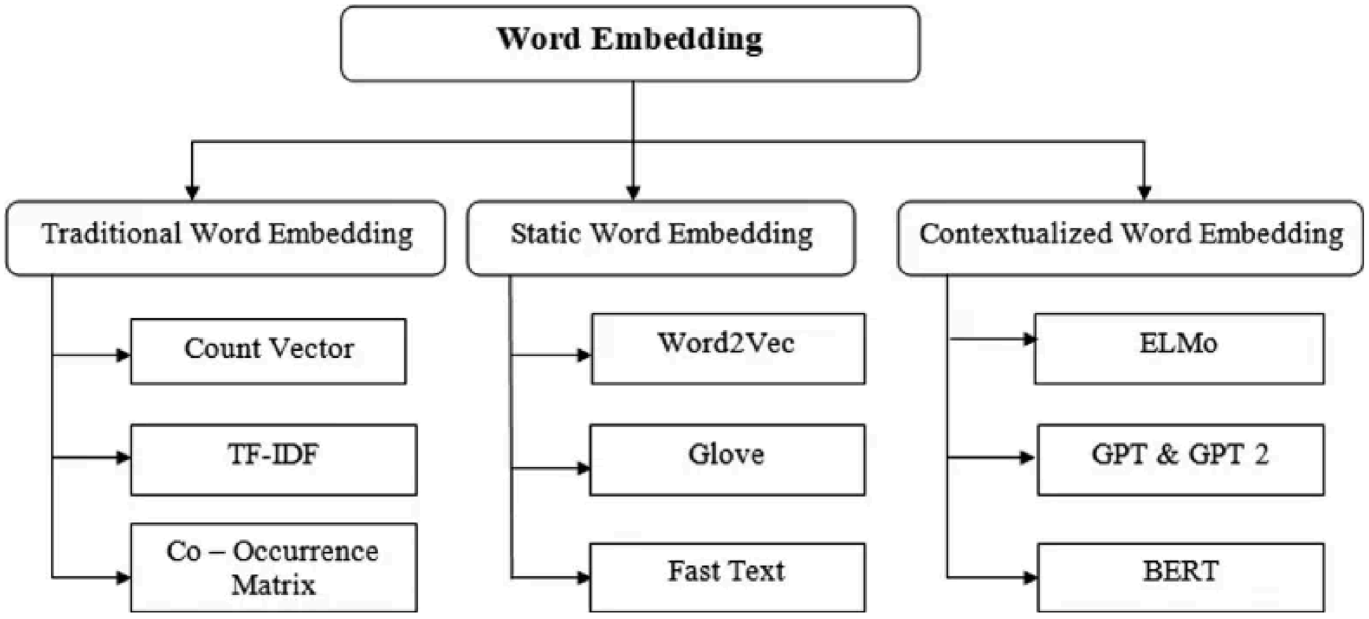
A Comprehensive Guide to Word Embeddings in NLP

Harsh Vardhan · Follow
10 min read · Jul 11, 2024



In the realm of Natural Language Processing (NLP), converting words into vectors — commonly referred to as word embeddings — is fundamental. These embeddings serve as the foundation for numerous NLP applications, enabling computers to understand and interpret human language. In this blog, rather than understanding why we use word embeddings, we will explore various techniques used to convert words into vectors, providing detailed explanations and code implementations for each.

Table of Contents

1. One Hot Encoding

It can be identified as one of the simplest forms of representing words numerically. Each word is represented as a binary vector with the length of the entire vocabulary. The vector has a “1” in the position corresponding to the word’s index and “0” elsewhere.
Pro’s: Simple, interpretable.
Con’s: High dimensionality, doesn’t capture semantic relationships.

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Sample data
corpus = ['dog', 'cat', 'dog', 'fish']

# Reshape data to fit the model
corpus = np.array(corpus).reshape(-1, 1)

# One-hot encode the data
onehot_encoder = OneHotEncoder(sparse=False)
onehot_encoded = onehot_encoder.fit_transform(corpus)

print(onehot_encoded)

#output
[[0. 1. 0.]
 [1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

2. Bag of Words(BoW)

Bag-of-Words (BOW) is a simple technique in Natural Language Processing (NLP) for representing text documents as numerical vectors. The idea is to treat each document as a bag, or a collection, of words, and then count the frequency of each word in the document. It does not consider the order of words but provides a straightforward way to convert text into vectors.
Pros: Simple, capture word importance based on frequency
Cons: Ignores word order and context.

```
from sklearn.feature_extraction.text import CountVectorizer

# Sample data
corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]

# Initialize the CountVectorizer
vectorizer = CountVectorizer()

# Fit and transform the corpus
X = vectorizer.fit_transform(corpus)

print(X.toarray())
print(vectorizer.get_feature_names_out())

#output of the above code
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
['and' 'document' 'first' 'is' 'one' 'second' 'the' 'third' 'this']
```

3. TF-IDF (Term Frequency-Inverse Document Frequency)

TF-IDF is an improvement over BoW, which considers the importance of words by reducing the weight of common words while increasing the weight of rare ones. The idea behind TF-IDF is to calculate the importance of a word in a document by considering two factors:

1. **Term Frequency (TF):** This measures how frequently a term appears in a document. The higher the frequency, the more important the term is assumed to be to that document.
2. **Inverse Document Frequency (IDF):** It is a measure of how important a term is across all documents in the corpus. It's based on the intuition that terms that appear in many documents are less informative than terms that appear in fewer documents.

TF-IDF = TF * IDF

The TF-IDF score is used to rank the importance of words in a document, with higher scores indicating more important words.

Pros: Considers both term frequency and rarity.

Cons: Still doesn't capture semantic relationships.

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample data
corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]

# Initialize the TfidfVectorizer
vectorizer = TfidfVectorizer()

# Fit and transform the corpus
X = vectorizer.fit_transform(corpus)

print(X.toarray())
print(vectorizer.get_feature_names_out())

#output
[[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]
 [0.         0.6876236  0.         0.28108867 0.         0.53864762
  0.28108867 0.         0.28108867]
 [0.51184851 0.         0.         0.26710379 0.51184851 0.
  0.26710379 0.51184851 0.26710379]
 [0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
['and' 'document' 'first' 'is' 'one' 'second' 'the' 'third' 'this']
```

4. Word2Vec

Word2Vec is a neural network-based model that generates dense vector representations of words. The basic idea behind Word2Vec is to train a neural network to predict the context words given a target word, and then use the resulting vector representations to capture the semantic meaning of the words. It captures semantic relationships between words using two main approaches: Continuous Bag of Words (CBOW) and Skip-gram.

1. **Continuous Bag-of-Words (CBOW):** Predicts the target word given its surrounding context words.
2. **Skip-Gram:** Predicts the surrounding context words given a target word.

Pros: Captures semantic relationships between words, lower dimensionality than classical methods.
Cons: Requires training on a large corpus, doesn't handle out-of-vocabulary (OOV) words well.

```python
from gensim.models import Word2Vec

# Sample data
sentences = [
    ['this', 'is', 'the', 'first', 'document'],
    ['this', 'document', 'is', 'the', 'second', 'document'],
    ['and', 'this', 'is', 'the', 'third', 'one'],
    ['is', 'this', 'the', 'first', 'document']
]

# Initialize the Word2Vec model
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)

# Train the model
model.train(sentences, total_examples=len(sentences), epochs=10)

# Get vector for a word
print(model.wv['document'])
```

5. GloVe (Global Vectors for Word Representation)

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. It aims to capture both global word co-occurrence statistics and local context window-based information (like Word2Vec) to create high-quality word vectors.
The core idea is that the ratios of word-word co-occurrence probabilities encode meaningful semantic relationships. For example, consider the words “ice” and “steam” and their co-occurrence probabilities with other words like “solid” and “gas.” The ratio `P(solid | ice) / P(solid | steam)` would be high, indicating a stronger association between "ice" and "solid" compared to "steam" and "solid." GloVe leverages these ratios to learn word vectors.

Pros: Captures both global and local context, faster training than Word2Vec.
Cons: Similar to Word2Vec in terms of OOV words.

```
import gensim.downloader as api

# Download pre-trained GloVe model (choose the size you need - 50, 100, 200, or
glove_vectors = api.load("glove-wiki-gigaword-100")  # Example: 100-dimensional

# Get word vectors (embeddings)
word1 = "king"
word2 = "queen"
vector1 = glove_vectors[word1]
vector2 = glove_vectors[word2]

# Compute cosine similarity between the two word vectors
similarity = glove_vectors.similarity(word1, word2)

print(f"Word vectors for '{word1}': {vector1}")
print(f"Word vectors for '{word2}': {vector2}")
print(f"Cosine similarity between '{word1}' and '{word2}': {similarity}")
```

6. FastText

Subword Information: Unlike traditional word embedding models (e.g., Word2Vec, Glove) that treat each word as a distinct unit, FastText considers words as a bag of character n-grams (subwords). An n-gram is a contiguous sequence of *n* characters.
Efficient Handling of Out-of-Vocabulary (OOV) Words: By utilizing subword information, FastText can generate meaningful representations for rare words or words not seen during training. This is particularly useful for morphologically rich languages (e.g., agglutinative languages) and in situations where the vocabulary is constantly evolving.
Fast Training and Inference: FastText employs two key techniques to achieve high speed:

- **Hierarchical Softmax:** A hierarchical tree structure is used to represent the output labels, significantly reducing the computational complexity of the softmax function during training and prediction.
- **Negative Sampling:** Only a small subset of the output labels (negative samples) are updated during training, making updates much faster.

Pros: Handles OOV words, captures morphological information.
Cons: Higher dimensionality than Word2Vec.

```
from gensim.models import FastText

# Sample data
sentences = [
    ['this', 'is', 'the', 'first', 'document'],
    ['this', 'document', 'is', 'the', 'second', 'document'],
    ['and', 'this', 'is', 'the', 'third', 'one'],
    ['is', 'this', 'the', 'first', 'document']
]

# Initialize the FastText model
model = FastText(sentences, vector_size=100, window=5, min_count=1, workers=4)
```

```
# Train the model
model.train(sentences, total_examples=len(sentences), epochs=10)

# Get vector for a word
print(model.wv['document'])
```

7. ELMo (Embeddings from Language Models)

ELMo is a deep contextualized word representation that models both complex characteristics of word use (e.g., syntax and semantics) and how these uses vary across linguistic contexts (e.g., to model polysemy). ELMo generates context-sensitive embeddings using deep bi-directional LSTM models. It provides different embeddings for words based on their context.

Key Features of ELMo:

1. **Contextualized Representations:** ELMo embeddings vary depending on the context in which a word appears. This is different from traditional embeddings like Word2Vec or GloVe, which provide a single representation for each word regardless of context.
2. **Deep Representations:** ELMo uses deep bidirectional LSTM networks to generate embeddings, capturing complex syntactic and semantic information.
3. **Layer-wise Representations:** ELMo combines representations from all the layers of the biLSTM, which allows it to capture different types of information at different layers.

Pros: Contextualized Embeddings, Bi-directional Context.
Cons: Complexity and Size, Fixed context window.

```
import tensorflow as tf
import tensorflow_hub as hub

# Load pre-trained ELMo model from TensorFlow Hub
elmo = hub.load("https://tfhub.dev/google/elmo/3")

# Sample data
sentences = ["This is the first document.", "This document is the second documen

def elmo_vectors(sentences):
    embeddings = elmo.signatures['default'](tf.constant(sentences))['elmo']
    return embeddings

# Get ELMo embeddings
elmo_embeddings = elmo_vectors(sentences)
print(elmo_embeddings)
```

8. BERT (Bidirectional Encoder Representations from Transformers)

BERT is a transformer-based model that generates context-aware embeddings by considering the entire sentence bidirectionally (i.e., both left-to-right and right-to-left). Unlike traditional word embeddings like Word2Vec or GloVe, which generate a single representation for each word, BERT produces different embeddings for each word based on its context. Key Features of BERT:

1. **Encoder:** The encoder is the main component of the BERT architecture, consisting of multiple transformer encoder layers. Each encoder layer has a multi-head self-attention mechanism, which allows the model to attend to different positions in the input sequence when encoding a specific token. The self-attention mechanism is bidirectional, meaning that the encoding of a token can attend to both left and right context in the sequence. Each encoder layer also has a position-wise feed-forward

network, which applies non-linear transformations to the output of the self-attention mechanism.

2. **Pre-trained on Large Corpus:** BERT is pre-trained on a large corpus of unlabeled text using two main objectives: a. **Masked Language Modeling (MLM):** In this task, some tokens in the input sequence are randomly masked (replaced with a [MASK] token), and the model is trained to predict the masked tokens based on the context. b. **Next Sentence Prediction (NSP):** In this task, the model is trained to predict whether two input sentences are consecutive in the original text or not.
3. **Fine-tuning:** After pre-training, BERT can be fine-tuned for specific NLP tasks by adding a task-specific output layer on top of the encoder. The parameters of the pre-trained BERT model are fine-tuned using labeled data for the target task, while the output layer is trained from scratch. This transfer learning approach allows BERT to leverage its pre-trained knowledge and adapt to new tasks with relatively small amounts of labeled data.

Pros: Pre-training and Fine-tuning, Contextualized Embeddings, Bi-directional Context.
Cons: Complexity and Size, Complexity, Inference Time.

```
from transformers import BertTokenizer, BertModel
import torch

# Load pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Sample data
sentence = "This is the first document."

# Tokenize input
```

```
inputs = tokenizer(sentence, return_tensors='pt')

# Get embeddings
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state

print(embeddings)
```

9. GPT (Generative Pre-trained Transformer)

GPT is a transformer-based model that generates context-aware embeddings by processing text in a unidirectional manner (left-to-right).
GPT uses the decoder part of the transformer, to process the input text. The model consists of multiple layers of self-attention mechanisms, which allow it to capture dependencies between words in a sentence. This allows GPT to excel in tasks that involve language generation, such as text completion, dialogue generation, and more. The model is pre-trained on a large corpus of text and can be fine-tuned for specific tasks.

Pros: Effective for language generation, Fine-tuning capability.
Cons: Unidirectional context, Resource Intensive, Context Length Limitation.

```
from transformers import GPT2Tokenizer, GPT2Model
import torch

# Load pre-trained GPT-2 model and tokenizer
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model = GPT2Model.from_pretrained('gpt2')

# Sample data
sentence = "This is the first document."

# Tokenize input
inputs = tokenizer(sentence, return_tensors='pt')
```

```
# Get embeddings
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state

print(embeddings)
```

10. Other Transformer-based Models (RoBERTa, XLNet, ALBERT(A lite BERT), T5 etc.)

Several variants of the original BERT model, such as RoBERTa, XLNet, and ALBERT, have been developed to improve performance and efficiency. These models generate state-of-the-art embeddings for various NLP tasks.

```
#RoBERTa Example
from transformers import RobertaTokenizer, RobertaModel
import torch

# Load pre-trained RoBERTa model and tokenizer
tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
model = RobertaModel.from_pretrained('roberta-base')

# Sample data
sentence = "This is the first document."

# Tokenize input
inputs = tokenizer(sentence, return_tensors='pt')

# Get embeddings
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state

print(embeddings)
```

T5 is a transformer model that frames all NLP tasks as a text-to-text problem. It is versatile and can handle tasks like translation, summarization, and question answering.

```
#T5 example
from transformers import T5Tokenizer, T5Model
import torch

# Load pre-trained T5 model and tokenizer
tokenizer = T5Tokenizer.from_pretrained('t5-small')
model = T5Model.from_pretrained('t5-small')

# Sample data
sentence = "This is the first document."

# Tokenize input
inputs = tokenizer(sentence, return_tensors='pt')

# Get embeddings
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state

print(embeddings)
```

XLNet is a generalized autoregressive pretraining method that enables learning bidirectional contexts by maximizing the expected likelihood over all permutations of the factorization order.

```
from transformers import XLNetTokenizer, XLNetModel
import torch

# Load pre-trained XLNet model and tokenizer
tokenizer = XLNetTokenizer.from_pretrained('xlnet-base-cased')
model = XLNetModel.from_pretrained('xlnet-base-cased')

# Sample data
```

```
sentence = "This is the first document."

# Tokenize input
inputs = tokenizer(sentence, return_tensors='pt')

# Get embeddings
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state

print(embeddings)
```

ALBERT is a lighter and faster variant of BERT that uses factorized embedding parameterization and cross-layer parameter sharing to reduce the number of parameters while maintaining performance.

```
from transformers import AlbertTokenizer, AlbertModel
import torch

# Load pre-trained ALBERT model and tokenizer
tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
model = AlbertModel.from_pretrained('albert-base-v2')

# Sample data
sentence = "This is the first document."

# Tokenize input
inputs = tokenizer(sentence, return_tensors='pt')

# Get embeddings
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state

print(embeddings)
```

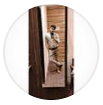
Conclusion

In this blog, we've explored a wide range of techniques for converting words to vectors, from simple methods like One-Hot Encoding and Bag of Words to advanced models like BERT and GPT. Each technique has its strengths and applications, making it essential to choose the right one based on the specific requirements of your NLP task.

Whether you're working on text classification, sentiment analysis, or any other NLP application, understanding and leveraging these word embedding techniques will significantly enhance your models' performance and capabilities.

Feel free to explore and experiment with these techniques, and don't hesitate to reach out with any questions or feedback. Happy learning!

NLP AI Word Embeddings Data Science Deep Learning



Written by Harsh Vardhan

85 Followers · 9 Following

Generative-AI Engineer

Follow

Responses (2)