**Advanced Word Embeddings: Word2Vec, GloVe, and FastText**

The Complete NLP Guide: Text to Context #6

[[Merve Bayram Durna](https://medium.com/@mervebdurna?source=post_page-----26e546ffedbd--------------------------------)](https://medium.com/@mervebdurna?source=post_page-----26e546ffedbd--------------------------------)

[Merve Bayram Durna](https://medium.com/@mervebdurna?source=post_page-----26e546ffedbd--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F9b7719e7f3c8&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40mervebdurna%2Fadvanced-word-embeddings-word2vec-glove-and-fasttext-26e546ffedbd&user=Merve+Bayram+Durna&userId=9b7719e7f3c8&source=post_page-9b7719e7f3c8----26e546ffedbd---------------------post_header-----------)

9 min read

·

Jan 15

65

Welcome to the 6th series of ‘The Complete NLP Guide: Text to Context’ blog. In our journey so far, we’ve explored the basics, applications, and challenges of Natural Language Processing. We delved into tokenization, text cleaning, stop words, stemming, lemmatization, part-of-speech tagging, and named entity recognition. Our exploration included text representation techniques like Bag-of-Words, TF-IDF, and an introduction to word embeddings. We then bridged NLP with Machine Learning, covering supervised and unsupervised learning, sentiment analysis, and the basics of classification and regression. Recently, we ventured into deep learning, discussing neural networks, RNNs, and LSTMs. Now, we’re set to dive deeper into word embeddings within the realm of deep learning.

Here’s what to expect in the 6th blog post:

1. **Word2Vec:** Delve into the world of Word2Vec, exploring its architecture, working principles, and how it revolutionizes the understanding of semantic relationships within the text. We’ll examine its two primary training algorithms: Continuous Bag of Words (CBOW) and Skip-gram, to understand their roles in capturing contextual word meanings.
2. **GloVe (Global Vectors for Word Representation):** Unpack the intricacies of the GloVe model. We’ll explore how it differs from Word2Vec by leveraging global word-word co-occurrence statistics, offering a unique approach to embedding words based on their collective context in a corpus.
3. **FastText:** Investigate the capabilities of FastText, focusing on its innovative approach to handling out-of-vocabulary words. Understand how FastText breaks down words into smaller units (n-grams) and how this method enhances the representation of words, especially in languages with rich morphology.
4. **Choosing the Right Embedding Model:** Dive into the critical factors to consider when selecting an embedding model for your NLP project. We’ll discuss the nuances of each model, helping you identify which one aligns best with your specific needs in terms of linguistic richness, computational efficiency, and scope of application.
5. **Compare Word Embeddings Code Example:** Bring theory into practice with a hands-on code example. This section will provide a practical demonstration, comparing the performance of Word2Vec, GloVe, and FastText in a common NLP task, giving you a tangible sense of their strengths and weaknesses in a real-world application.

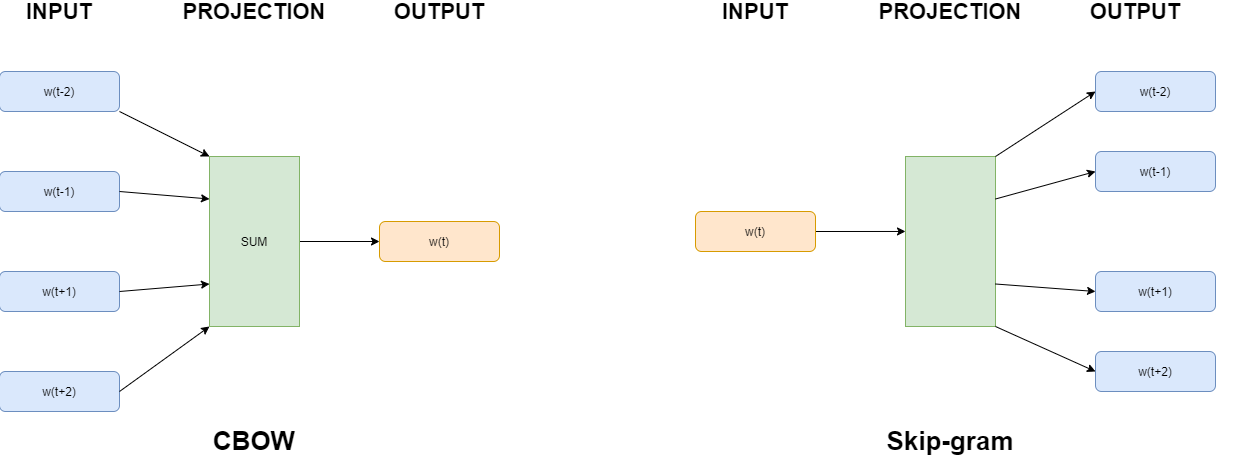
This blog post aims to not only educate you about these advanced embedding techniques but also to empower you with the knowledge to make informed decisions when it comes to implementing them in your NLP projects.

**Word2Vec**

Word2Vec is a popular word embedding technique that aims to represent words as continuous vectors in a high-dimensional space. It introduces two models: Continuous Bag of Words (CBOW) and Skip-gram, each contributing to the learning of vector representations.

**1. Model Architecture:**

* **Continuous Bag of Words (CBOW):** In CBOW, the model predicts a target word based on its context. The context words are used as input, and the target word is the output. The model is trained to minimize the difference between the predicted and actual target words.
* **Skip-gram:**Conversely, the Skip-gram model predicts context words given a target word. The target word serves as input, and the model aims to predict the words that are likely to appear in its context. Like CBOW, the goal is to minimize the difference between the predicted and actual context words.



**2. Neural Network Training:**

Both CBOW and Skip-gram models leverage neural networks to learn vector representations. The neural network is trained on a large text corpus, adjusting the weights of connections to minimize the prediction error. This process places similar words closer together in the resulting vector space.

**3. Vector Representations:**

Once trained, Word2Vec assigns each word a unique vector in the high-dimensional space. These vectors capture semantic relationships between words. Words with similar meanings or those that often appear in similar contexts have vectors that are close to each other, indicating their semantic similarity.

**4. Advantages and Disadvantages:**

**Advantages:**

* Captures semantic relationships effectively.
* Efficient for large datasets.
* Provides meaningful word representations.

**Disadvantages:**

* May struggle with rare words.
* Ignores word order.

**5. Code Example with Toy Dataset:**

The provided code example demonstrates the training of a Word2Vec model using the Gensim library on a toy dataset. Tokenization of sentences, model training, and access to word embeddings are showcased.

# Code Example with Toy Dataset  
from gensim.models import Word2Vec  
from nltk.tokenize import word\_tokenize  
  
# Toy dataset  
sentences = ["I love natural language processing.",   
 "Word embeddings are powerful."]  
  
# Tokenize sentences  
tokenized\_sentences = [word\_tokenize(sentence.lower()) for sentence in sentences]  
  
# Train Word2Vec model  
model = Word2Vec(sentences=tokenized\_sentences, vector\_size=100, window=5, min\_count=1, workers=4)  
  
# Access embeddings  
word\_embeddings = model.wv  
print(word\_embeddings['natural'])

In summary, Word2Vec’s mechanics involve training neural network models (CBOW and Skip-gram) to learn vector representations that effectively capture semantic relationships between words. The resulting vectors provide meaningful and efficient word representations in the vector space.

**GloVe (Global Vectors for Word Representation)**

Global Vectors for Word Representation (GloVe) is a powerful word embedding technique that captures the semantic relationships between words by considering their **co-occurrence probabilities** within a corpus. The key to GloVe’s effectiveness lies in the construction of a word-context matrix and the subsequent factorization process.

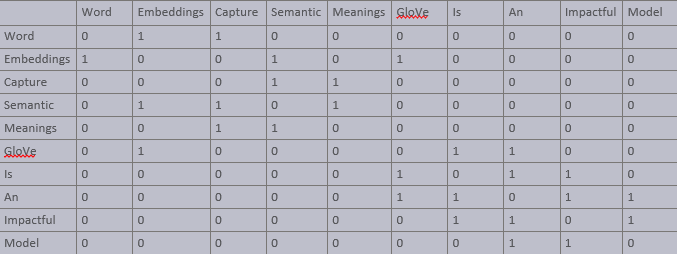
**1. Word-Context Matrix Formation:**

The first step in GloVe’s mechanics involves creating a word-context matrix. This matrix is designed to represent the likelihood of a given word appearing near another across the entire corpus. Each cell in the matrix holds the co-occurrence count of how often words appear together in a certain context window.

Let’s consider a simplified example. Assume we have the following sentences in our corpus:

* “Word embeddings capture semantic meanings.”
* “GloVe is an impactful word embedding model.”

The word-context matrix might look like this:



Here, each row and column corresponds to a unique word in the corpus, and the values in the cells represent how often these words appear together within a certain context window.

**2. Factorization for Word Vectors:**

With the word-context matrix in place, GloVe turns to matrix factorization. The objective here is to decompose this high-dimensional matrix into two smaller matrices — one representing words and the other contexts. Let’s denote these as *W* for words and *C* for contexts. The ideal scenario is when the dot product of *W* and *CT* (transpose of *C*) approximates the original matrix:

X≈*W*⋅*CT*

Through iterative optimization, GloVe adjusts *W* and *C* to minimize the difference between *X* and *W*⋅*CT*. This process yields refined vector representations for each word, capturing the nuances of their co-occurrence patterns.

**3. Vector Representations:**

Once trained, GloVe provides each word with a dense vector that captures not just local context but global word usage patterns. These vectors encode semantic and syntactic information, revealing similarities and differences between words based on their overall usage in the corpus.

**4. Advantages and Disadvantages:**

**Advantages:**

* Efficiently captures global statistics of the corpus.
* Good at representing both semantic and syntactic relationships.
* Effective in capturing word analogies.

**Disadvantages:**

* Requires more memory for storing co-occurrence matrices.
* Less effective with very small corpora.

**5. Code Example with Toy Dataset:**

The following code snippet demonstrates the basic usage of the GloVe model using the GloVe Python package on a toy dataset. The example covers the creation of co-occurrence matrix, training of the GloVe model, and retrieval of word embeddings.

from glove import Corpus, Glove  
from nltk.tokenize import word\_tokenize  
  
# Toy dataset  
sentences = ["Word embeddings capture semantic meanings.",  
 "GloVe is an impactful word embedding model."]  
  
# Tokenize sentences  
tokenized\_sentences = [word\_tokenize(sentence.lower()) for sentence in sentences]  
  
# Creating a corpus object  
corpus = Corpus()   
  
# Training the corpus to generate the co-occurrence matrix  
corpus.fit(tokenized\_sentences, window=10)  
  
# Training the GloVe model  
glove = Glove(no\_components=100, learning\_rate=0.05)  
glove.fit(corpus.matrix, epochs=30, no\_threads=4, verbose=True)  
glove.add\_dictionary(corpus.dictionary)  
  
# Retrieve and display word embeddings  
word = "glove"  
embedding = glove.word\_vectors[glove.dictionary[word]]  
print(f"Embedding for '{word}': {embedding}")

In conclusion, GloVe’s approach to word embeddings focuses on capturing global word co-occurrence patterns within a corpus, providing rich and meaningful vector representations. This method effectively encodes both semantic and syntactic relationships, offering a comprehensive view of word meanings based on their broad usage patterns. The above code example illustrates how to implement GloVe embeddings on a basic dataset.

**FastText**

FastText is an advanced word embedding technique developed by Facebook AI Research (FAIR) that extends the Word2Vec model. Unlike Word2Vec, FastText not only considers whole words but also incorporates subword information — parts of words like n-grams. This approach enables the handling of morphologically rich languages and captures information about word structure more effectively.

**1. Subword Information:**

FastText represents each word as a bag of character n-grams in addition to the whole word itself. This means that the word “apple” is represented by the word itself and its constituent n-grams like “ap”, “pp”, “pl”, “le”, etc. This approach helps capture the meanings of shorter words and affords a better understanding of suffixes and prefixes.

**2. Model Training:**

Similar to Word2Vec, FastText can use either the CBOW or Skip-gram architecture. However, it incorporates the subword information during training. The neural network in FastText is trained to predict words (in CBOW) or context (in Skip-gram) not just based on the target words but also based on these n-grams.

**3. Handling Rare and Unknown Words:**

A significant advantage of FastText is its ability to generate better word representations for rare words or even words not seen during training. By breaking down words into n-grams, FastText can construct meaningful representations for these words based on their subword units.

**4. Advantages and Disadvantages:**

**Advantages:**

* Better representation of rare words.
* Capable of handling out-of-vocabulary words.
* Richer word representations due to subword information.

**Disadvantages:**

* Increased model size due to n-gram information.
* Longer training times compared to Word2Vec.

**5. Code Example with Toy Dataset:**

The following code demonstrates how to use FastText with the Gensim library on a toy dataset. It highlights model training and accessing word embeddings.

from gensim.models import FastText  
from nltk.tokenize import word\_tokenize  
  
# Toy dataset  
sentences = ["FastText embeddings handle subword information.",  
 "It is effective for various languages."]  
# Tokenize sentences  
tokenized\_sentences = [word\_tokenize(sentence.lower()) for sentence in sentences]  
  
# Train FastText model  
model = FastText(sentences=tokenized\_sentences, vector\_size=100, window=5, min\_count=1, workers=4)  
  
# Access embeddings  
word\_embeddings = model.wv  
print(word\_embeddings['subword'])

In summary, FastText enriches the word embedding landscape by incorporating subword information, making it highly effective for capturing intricate details in language and handling rare or unseen words.

**Choosing the Right Embedding Model**

* Word2Vec: Use when semantic relationships are crucial, and you have a large dataset.
* GloVe: Suitable for diverse datasets and when capturing global context is important.
* FastText: Opt for morphologically rich languages or when handling out-of-vocabulary words is vital.

**Compare Word Embeddings Code Example**

# Import necessary libraries  
from gensim.models import Word2Vec  
from gensim.models import FastText  
from glove import Corpus, Glove  
from sklearn.manifold import TSNE  
import matplotlib.pyplot as plt  
  
# Toy dataset  
toy\_data = [  
 "word embeddings are fascinating",  
 "word2vec captures semantic relationships",  
 "GloVe considers global context",  
 "FastText extends Word2Vec with subword information"  
]  
  
# Function to train Word2Vec model  
def train\_word2vec(data):  
 model = Word2Vec([sentence.split() for sentence in data], vector\_size=100, window=5, min\_count=1, workers=4)  
 return model  
  
# Function to train GloVe model  
def train\_glove(data):  
 corpus = Corpus()  
 corpus.fit(data, window=5)  
 glove = Glove(no\_components=100, learning\_rate=0.05)  
 glove.fit(corpus.matrix, epochs=30, no\_threads=4, verbose=True)  
 return glove  
  
# Function to train FastText model  
def train\_fasttext(data):  
 model = FastText(sentences=[sentence.split() for sentence in data], vector\_size=100, window=5, min\_count=1, workers=4)  
 return model  
  
# Function to plot embeddings  
def plot\_embeddings(model, title):  
 labels = model.wv.index\_to\_key  
 vectors = [model.wv[word] for word in labels]  
   
 tsne\_model = TSNE(perplexity=40, n\_components=2, init='pca', n\_iter=2500, random\_state=23)  
 new\_values = tsne\_model.fit\_transform(vectors)  
  
 x, y = [], []  
 for value in new\_values:  
 x.append(value[0])  
 y.append(value[1])  
  
 plt.figure(figsize=(10, 8))   
 for i in range(len(x)):  
 plt.scatter(x[i],y[i])  
 plt.annotate(labels[i],  
 xy=(x[i], y[i]),  
 xytext=(5, 2),  
 textcoords='offset points',  
 ha='right',  
 va='bottom')  
 plt.title(title)  
 plt.show()  
  
# Train models  
word2vec\_model = train\_word2vec(toy\_data)  
glove\_model = train\_glove(toy\_data)  
fasttext\_model = train\_fasttext(toy\_data)  
  
# Plot embeddings  
plot\_embeddings(word2vec\_model, 'Word2Vec Embeddings')  
plot\_embeddings(glove\_model, 'GloVe Embeddings')  
plot\_embeddings(fasttext\_model, 'FastText Embeddings')

**Conclusion**

As we conclude our exploration of advanced word embeddings, the next stop on our NLP journey will be Sequence-to-Sequence models, Attention mechanisms, and Encoder-Decoder architectures. These advanced techniques are instrumental in tasks like machine translation and summarization, allowing models to focus on specific parts of input sequences.

Stay tuned for the next installment as we unravel the complexities of Sequence-to-Sequence models, shining a light on the power of attention mechanisms and encoder-decoder architectures.

**Explore the Series on GitHub**

For a comprehensive hands-on experience, visit our GitHub repository. It houses all the code samples from this article and the entire “The Complete NLP Guide: Text to Context” blog series. Dive in to experiment with the codes and enhance your understanding of NLP. Check it out here: <https://github.com/mervebdurna/10-days-NLP-blog-series>

Feel free to clone the repository, experiment with the code, and even contribute to it if you have suggestions or improvements. This is a collaborative effort, and your input is highly valued!

Happy exploring and coding!

[NLP](https://medium.com/tag/nlp?source=post_page-----26e546ffedbd---------------nlp-----------------)

[Llm](https://medium.com/tag/llm?source=post_page-----26e546ffedbd---------------llm-----------------)

[Deep Learning](https://medium.com/tag/deep-learning?source=post_page-----26e546ffedbd---------------deep_learning-----------------)

[Machine Learning](https://medium.com/tag/machine-learning?source=post_page-----26e546ffedbd---------------machine_learning-----------------)

[Mathematics](https://medium.com/tag/mathematics?source=post_page-----26e546ffedbd---------------mathematics-----------------)