NLP3: A comprehensive guide for converting Words into Vectors

Saket Kumar

22 min read · 22 hours ago

In natural language processing (NLP), machine learning models often require numerical inputs. To bridge the gap between text and numbers, we use various techniques to convert words into a numerical format that algorithms can process.

One-Hot Encoding (OHE)

> *One-Hot Encoding (OHE) is a technique used to convert categorical data into a numerical format that machine learning algorithms can process. Each unique category (or word, in the case of text) is represented as a binary vector. Here's a step-by-step explanation of how OHE works using the example sentences:*

Example Sentences

1. "Dogs chase cats in the garden."
2. "Dogs bark loudly at night."

Steps to Apply One-Hot Encoding

1. **Vocabulary Creation :** First, we need to create a vocabulary of all unique words present in the dataset. From our example sentences, the unique words are:

- Dogs
- chase
- cats
- in
- the
- garden
- bark
- loudly
- at
- night

This gives us the following vocabulary list:

```
["Dogs", "chase", "cats", "in", "the", "garden", "bark", "loudly", "at", "night"
```

Binary Vector Representation

Each word in the vocabulary will be assigned a unique index. For example:

- “Dogs” → index 0
- “chase” → index 1

- “cats” → index 2
- “in” → index 3
- “the” → index 4
- “garden” → index 5
- “bark” → index 6
- “loudly” → index 7
- “at” → index 8
- “night” → index 9

For each sentence, create a binary vector of length equal to the vocabulary size, where the element at the index corresponding to each word is set to 1.

Sentence 1: “Dogs chase cats in the garden.”

- “Dogs”: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
- “chase”: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
- “cats”: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
- “in”: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
- “the”: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
- “garden”: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]

The final one-hot encoded vector for this sentence:

```
[
  [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
]
```

Advantages and Disadvantages of One-Hot Encoding (OHE)

Advantages:

1. **Simplicity**: One-Hot Encoding is straightforward to understand and implement. Each word is represented as a distinct binary vector, making it easy to interpret.
2. **No Assumptions About Order**: OHE does not make any assumptions about the order or frequency of words. Each word is treated independently, which can be beneficial when the sequence of words is not important for the model.
3. **Compatibility**: OHE produces features that are suitable for algorithms that require numerical inputs, such as logistic regression, support vector machines, and neural networks.
4. **No Information Loss**: Unlike some methods, OHE preserves all the information about the presence or absence of each word. Every word is represented explicitly.

Disadvantages:

1. **High Dimensionality**: The size of the feature vector is equal to the size of the vocabulary. For large vocabularies, this can result in very high-dimensional vectors, leading to increased memory usage and computational costs.

2. **Sparsity**: The vectors generated by OHE are sparse, meaning that most elements are zero. Sparse vectors can be inefficient to store and process, especially when the vocabulary is large.
3. **Lack of Context**: OHE does not capture any information about the relationships or context between words. It treats each word as an independent feature, ignoring the order and semantics of the words in a sentence.
4. **Scalability Issues**: For very large text corpora, the vocabulary size can become enormous, making OHE impractical. The number of features grows linearly with the size of the vocabulary.

Bag of Words (BoW)

> *The Bag of Words (BoW) model is a common technique used in natural language processing (NLP) for text representation. It is a simple yet powerful approach to convert text data into numerical features that can be used by machine learning algorithms.*

Text Preprocessing: For the given sentences:

- “Dogs chase cats in the garden.”
- “Dogs bark loudly at night.”

After preprocessing (assuming lowercasing, removing punctuation, and stopword removal):

- “dogs chase cats garden”
- “dogs bark loudly night”

Vocabulary Creation: The BoW model creates a vocabulary, which is a list of all unique words (tokens) present in the entire text corpus. From the above sentences, the vocabulary would be:

- `['dogs', 'chase', 'cats', 'garden', 'bark', 'loudly', 'night']`

Vectorization: Each sentence (or document) is represented as a vector, where each element of the vector corresponds to a word in the vocabulary. The value of each element is the frequency (or sometimes the presence/absence) of the corresponding word in that sentence.

For the given sentences:

- **Sentence 1:** “dogs chase cats garden”
- **Sentence 2:** “dogs bark loudly night”

The vectors would be:

Vocabulary	Dogs	Chase	Cats	Garden	Bark	Loudly	Night
Sentence 1 ("dogs chase cats garden")	1	1	1	1	0	0	0
Sentence 2 ("dogs bark loudly night")	1	0	0	0	1	1	1

- **Sentence 1 Vector:** [1, 1, 1, 1, 0, 0, 0]
- **Sentence 2 Vector:** [1, 0, 0, 0, 1, 1, 1]

Explanation:

The vector for Sentence 1:

- `1` for "dogs" (appears once)

- `1` for "chase" (appears once)
- `1` for "cats" (appears once)
- `1` for "garden" (appears once)
- `0` for "bark" (does not appear)
- `0` for "loudly" (does not appear)
- `0` for "night" (does not appear)

The vector for Sentence 2:

- `1` for "dogs" (appears once)
- `0` for "chase" (does not appear)
- `0` for "cats" (does not appear)
- `0` for "garden" (does not appear)
- `1` for "bark" (appears once)
- `1` for "loudly" (appears once)
- `1` for "night" (appears once)

If a word is present more than 1 we can make it 1 by using binary bag of words.

Advantages of Bag of Words:

- **Simplicity**: BoW is easy to understand and implement.
- **Efficiency**: It is computationally efficient, especially for small text datasets.

Limitations of Bag of Words:

- **Loss of Semantic Meaning**: BoW ignores the order of words, which means it loses the context and semantic meaning of the text.
- **Sparsity**: The resulting vectors can be large and sparse, especially if the vocabulary is extensive.
- **Handling of Synonyms**: BoW does not account for synonyms or words with similar meanings, treating them as entirely separate features.

TF-IDF (Term Frequency-Inverse Document Frequency)

> *TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure used in text mining and information retrieval to evaluate the importance of a word in a document relative to a collection of documents (corpus). Unlike the Bag of Words model, which only considers raw word counts, TF-IDF also takes into account how common or rare a word is across multiple documents. This helps in identifying words that are particularly important or unique to a specific document.*

How TF-IDF Works:

1. Term Frequency (TF):

- TF measures how frequently a word appears in a document. It's calculated as the number of times a word appears in a document divided by the total number of words in the document.

Formula:

$$\mathrm{TF}(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

For example:

In Sentence 1: “Dogs chase cats in the garden.”

- TF(“dogs”) = 1/6 (since “dogs” appears once, and there are 6 words in total)
- TF(“chase”) = 1/6
- TF(“cats”) = 1/6
- TF(“in”) = 1/6
- TF(“the”) = 1/6
- TF(“garden”) = 1/6

In Sentence 2: “Dogs bark loudly at night.”

- TF(“dogs”) = 1/6
- TF(“bark”) = 1/6
- TF(“loudly”) = 1/6
- TF(“at”) = 1/6
- TF(“night”) = 1/6

2. Inverse Document Frequency (IDF):

- IDF measures how important a word is across the entire corpus. Words that are common across many documents have a low IDF, while rare words have a high IDF.

Formula:

$$\text{IDF}(t, D) = \log\left(\frac{\text{Total number of documents } (N)}{\text{Number of documents containing term } t}\right)$$

- Here, N is the total number of documents, and the denominator is the number of documents in which the term appears.

For our two sentences:

- "dogs" appears in both documents, so:

$$\text{IDF}(\text{"dogs"}) = \log\left(\frac{2}{2}\right) = \log(1) = 0$$

(The IDF is 0, meaning "dogs" is not particularly informative as it appears in all documents.)

- "chase", "cats", "garden" appear only in Sentence 1:

$$\text{IDF}(\text{"chase"}) = \log\left(\frac{2}{1}\right) = \log(2) \approx 0.693$$

- "bark", "loudly", "night" appear only in Sentence 2:

$$\text{IDF}(\text{"bark"}) = \log\left(\frac{2}{1}\right) = \log(2) \approx 0.693$$

TF-IDF Calculation:

- Finally, the TF-IDF value for each word in a document is calculated by multiplying the TF and IDF values:

$$\text{TF-IDF}(t,d,D) = \text{TF}(t,d) \times \text{IDF}(t,D)$$

For example:

In Sentence 1:

- TF-IDF(“dogs”) = 1/6 × 0 = 0
- TF-IDF(“chase”) = 1/6 × 0.693 ≈ 0.1155

In Sentence 2:

- TF-IDF(“dogs”) = 1/6 × 0 = 0
- TF-IDF(“bark”) = 1/6 × 0.693 ≈ 0.1155

Excel Table Representation:

Vocabulary	Dogs	Chase	Cats	Garden	Bark	Loudly	Night
TF Sentence 1	1/6	1/6	1/6	1/6	0	0	0
TF Sentence 2	1/6	0	0	0	1/6	1/6	1/6
IDF	0	0.693	0.693	0.693	0.693	0.693	0.693
TF-IDF Sentence 1	0	0.1155	0.1155	0.1155	0	0	0
TF-IDF Sentence 2	0	0	0	0	0.1155	0.1155	0.1155

Advantages of TF-IDF:

- **Relevance**: TF-IDF helps identify words that are important to a specific document but not common across other documents.
- **Handling Common Words**: Unlike BoW, TF-IDF down weights common words that appear in many documents, reducing their impact.

Limitations of TF-IDF:

- **No Semantic Understanding**: TF-IDF still doesn't capture the meaning of words or their context.
- **Sparse Vectors**: For large corpora, the resulting vectors can still be large and sparse.
- **Ignores Word Order**: Like BoW, TF-IDF ignores the order of words in a document.

Word2Vec

> *Word2Vec is a neural network-based model that learns vector representations of words. These vectors, often referred to as word embeddings, capture the semantic meaning of words in a continuous vector space. Unlike traditional one-hot encoding, where words are represented as sparse vectors with a single '1' in a large*

> *vector of zeros, Word2Vec maps words to dense vectors of real numbers, where semantically similar words are close to each other in this space.*

How Does Word2Vec Work?

Word2Vec has two main approaches: **Continuous Bag of Words (CBOW)** and **Skip-Gram**.

1. **CBOW** predicts a target word given its surrounding context words.
2. **Skip-Gram** does the opposite: it predicts the context words given a target word.

In both approaches, the model is trained on large corpora of text to adjust the word vectors such that words appearing in similar contexts have similar vectors.

To explain Continuous Bag of Words (CBOW) in detail using the examples “Dogs chase cats in the garden.” and “Dogs bark loudly at night.,” let’s break down how CBOW works and how it applies to these sentences.

CBOW

The Continuous Bag of Words (CBOW) model is one of the two approaches used in Word2Vec. It predicts a target word based on its surrounding context words. The model takes a set of context words (or a “window” of words around the target word) and tries to predict the word that is most likely to occur in that context.

Key Concepts in CBOW

- **Context Words:** These are the words surrounding the target word within a certain window size.

- **Target Word:** The word that the model is trying to predict based on the context words.

Example Sentences

Consider the two sentences:

1. “Dogs chase cats in the garden.”
2. “Dogs bark loudly at night.”

How CBOW Works (Step-by-Step)

Let’s go through how CBOW would operate on these sentences.

1. Defining the Context Window

First, we need to define the size of the context window. For simplicity, let’s assume a context window size of 2 (i.e., we look at 2 words before and 2 words after the target word).

2. Processing Each Word

For each word in the sentence, CBOW will use the surrounding context words to predict the target word.

Example 1: “Dogs chase cats in the garden.”

Target Word: “chase”

Context Words: [“Dogs”, “cats”]

- The model will take the words “Dogs” and “cats” and predict that the most likely target word in this context is “chase.”

Target Word: “cats”

Context Words: [“chase”, “in”]

- The model will use “chase” and “in” to predict that the word “cats” is most likely to occur in this context.

Target Word: “in”

Context Words: [“cats”, “the”]

- The context words “cats” and “the” will be used to predict “in.”

Example 2: “Dogs bark loudly at night.”

Target Word: “bark”

Context Words: [“Dogs”, “loudly”]

- The words “Dogs” and “loudly” are used to predict “bark.”

Target Word: “loudly”

Context Words: [“bark”, “at”]

- The context words “bark” and “at” will help predict that “loudly” is the correct target word.

3. Training the Model

- **Input Layer:** The context words are input into the model as one-hot encoded vectors.

- **Hidden Layer:** These input vectors are multiplied by a weight matrix to produce a hidden layer representation.
- **Output Layer:** The hidden layer representation is then multiplied by another weight matrix to produce a score for each possible word in the vocabulary. The word with the highest score is the predicted target word.

During training, the CBOW model adjusts the weights in the matrices so that the predicted target words become as close as possible to the actual target words. This process happens across all words in the training corpus, leading to the learning of word vectors that encode semantic relationships.

4. Vector Representation

- After training, each word in the vocabulary has a learned vector representation. For instance, “Dogs,” “bark,” “chase,” “loudly,” etc., will all have vectors that capture their meanings based on how they appear in context.
- Words that often appear in similar contexts will have similar vectors, meaning “bark” might be close to “chase” in the vector space because both are actions commonly associated with “Dogs.”

Why CBOW is Useful

- **Efficient for Frequent Words:** CBOW tends to perform better on common words, as it learns to predict them based on the context they appear in.
- **Simpler Training:** CBOW is generally faster to train than the Skip-Gram model because it predicts a single target word from multiple context words, which results in fewer computations.

Skip-Gram

Let’s explore the Skip-Gram model using the sentences “Dogs chase cats in the garden.” and “Dogs bark loudly at night.” This explanation will help

illustrate how Skip-Gram works, its advantages, and how it applies to these examples.

What is the Skip-Gram Model?

The Skip-Gram model is another approach used in Word2Vec. Unlike CBOW, which predicts a target word based on its surrounding context words, Skip-Gram does the opposite: it predicts the context words given a target word. The goal of the Skip-Gram model is to find word representations that are useful for predicting the surrounding words in a sentence or document.

Key Concepts in Skip-Gram

- **Target Word:** The word from which the model predicts surrounding context words.
- **Context Words:** The words that surround the target word within a defined window size.

Example Sentences

Consider the sentences:

1. “Dogs chase cats in the garden.”
2. “Dogs bark loudly at night.”

How Skip-Gram Works (Step-by-Step)

1. Defining the Context Window

As with CBOW, we define a context window size. Let’s assume a context window of 2 words on each side.

2. Processing Each Word

For each word in the sentence, Skip-Gram will use the target word to predict its surrounding context words.

Example 1: “Dogs chase cats in the garden.”

Target Word: “Dogs”

Context Words: [“chase”, “cats”]

- The model will try to predict that the words “chase” and “cats” are likely to be found in the context of “Dogs.”

Target Word: “chase”

Context Words: [“Dogs”, “cats”, “in”]

- Here, the target word “chase” is used to predict that “Dogs,” “cats,” and “in” are in its context.

Target Word: “cats”

Context Words: [“chase”, “in”, “the”]

- The model will predict that the words “chase,” “in,” and “the” are in the context of “cats.”

Example 2: “Dogs bark loudly at night.”

Target Word: “Dogs”

Context Words: [“bark”, “loudly”]

- The model will predict that “bark” and “loudly” are in the context of “Dogs.”

Target Word: “bark”

Context Words: [“Dogs”, “loudly”, “at”]

- The model uses “bark” to predict “Dogs,” “loudly,” and “at.”

3. Training the Model

- **Input Layer:** The target word is input as a one-hot encoded vector.
- **Hidden Layer:** The one-hot encoded vector is multiplied by a weight matrix to create a hidden layer representation. This representation is the vector for the target word.
- **Output Layer:** The hidden layer is multiplied by another weight matrix to predict scores for all words in the vocabulary. The model tries to predict the context words by comparing these scores to the actual context words.

During training, the Skip-Gram model adjusts the weights to maximize the probability of predicting the correct context words given the target word.

4. Vector Representation

- After training, each word has a dense vector representation that reflects the contexts in which it appears.
- Words that frequently share similar contexts (e.g., “Dogs” with “bark” and “chase”) will have similar vector representations in the learned vector space.

Why Skip-Gram is Useful

- **Effective for Rare Words:** Skip-Gram tends to work well for infrequent words, as it focuses on learning a good representation for each target word based on the different contexts in which it appears.
- **Captures Complex Relationships:** By predicting multiple context words from a single target word, Skip-Gram can capture more nuanced relationships between words.

Comparison of CBOW and Skip-Gram

Training Speed:

- CBOW is faster to train because it averages context words to predict the target word.
- Skip-Gram is slower but more effective at capturing the nuances of less frequent words.

Accuracy:

- Skip-Gram is generally more accurate for smaller datasets or datasets with many rare words.
- CBOW is better suited for larger, balanced datasets.

Extensions and Variations of Word2Vec

Negative Sampling:

- An optimization technique used to reduce the computational complexity of Word2Vec, especially when dealing with a large vocabulary. Instead of updating the weights for all words in the vocabulary, negative sampling updates the weights for a small, random subset of words.

Hierarchical Softmax:

- Another technique to speed up the training process by replacing the standard softmax function with a binary tree representation of the output layer. This approach reduces the number of computations needed for each update.

Here are some popular Word2Vec pre-trained models :

1. **Google News Word2Vec:** Trained on 100 billion words from Google News, with 300-dimensional vectors.
2. **Wikipedia Word2Vec:** Trained on the entire English Wikipedia, useful for general knowledge tasks.
3. **Common Crawl Word2Vec:** Derived from web text data, ideal for understanding broad web-based content.
4. **Text8 Word2Vec:** A smaller model trained on a compressed Wikipedia subset, great for quick experiments.
5. **Google Code Word2Vec:** Trained on Google Code data, tailored for understanding programming-related text.
6. **Gensim Word2Vec Models:** Pre-trained on various corpora like Wikipedia and Common Crawl, easily integrated.
7. **Amazon Reviews Word2Vec:** Trained on Amazon product reviews, capturing customer sentiment and feedback language.

GloVe (Global Vectors for Word Representation)

> *GloVe is a word embedding technique developed by researchers at Stanford University. It combines the advantages of matrix factorization techniques and word context approaches like Word2Vec to capture the global statistical information of a corpus while also embedding words into a continuous vector space. Let's explore how GloVe works using the sentences:*

> *“Dogs chase cats in the garden.”*
>
> *“Dogs bark loudly at night.”*

Core Idea of GloVe

GloVe is based on the idea that the meaning of a word can be derived from the company it keeps, but more specifically, the co-occurrence statistics of words across a large corpus. The fundamental principle behind GloVe is to learn word vectors in such a way that the dot product of two word vectors is related to the logarithm of the words’ probability of co-occurrence.

Steps to Understand GloVe

1. **Build the Co-occurrence Matrix**

- **Context Window**: Define a context window size (e.g., 2 words before and 2 words after the target word). This window determines how many words around the target word are considered as context.
- **Co-occurrence Counting**: For each word in the vocabulary, count how often it appears in the context of every other word within the window.

For the example sentences:

- Vocabulary: [Dogs, chase, cats, in, the, garden, bark, loudly, at, night]
- We calculate the co-occurrence of each word with every other word within the defined context window.

For “Dogs chase cats in the garden.” with a window size of 2:

- “Dogs” co-occurs with “chase” and “cats.”
- “chase” co-occurs with “Dogs,” “cats,” and “in.”

- “cats” co-occurs with “chase,” “in,” and “the.”
- And so on.

The result is a co-occurrence matrix where each cell (i, j) contains the number of times word j appears in the context of word i.

2. Create the Probability Ratios

GloVe focuses on capturing the ratio of co-occurrence probabilities rather than the absolute values. The key insight is that certain word pairs should have specific relationships based on their co-occurrence probability.

For instance, the ratio of the probability that “garden” co-occurs with “dogs” to the probability that “garden” co-occurs with “bark” should provide meaningful information about the relationships between these words.

3. Objective Function

GloVe’s learning objective is to find word vectors such that their dot product equals the logarithm of the word’s co-occurrence probability. The objective function is as follows:

The goal is to minimize this objective function so that the word vectors capture meaningful semantic relationships

$$J = \sum_{i,j=1}^{V} f(X_{ij}) \left(w_i^T \cdot w_j + b_i + b_j - \log(X_{ij}) \right)^2$$

- X_{ij} is the co-occurrence count of words i and j.
- w_i and w_j are the word vectors for words i and j.
- b_i and b_j are biases for words i and j.
- $f(X_{ij})$ is a weighting function that reduces the impact of very frequent or very rare co-occurrences.

4. Learning Word Vectors

By solving the optimization problem defined by the objective function, GloVe learns the word vectors. Words that have similar contexts will have vectors that are close to each other in the vector space, allowing the model to capture semantic meaning and analogies.

Applying GloVe to Our Example Sentences

For simplicity, let's imagine we've applied GloVe to a large corpus that includes our example sentences. After training, GloVe would produce word vectors for each word in our sentences.

- **"Dogs"** might be represented by a vector like `[0.2, -0.3, 0.5, ...]`.
- **"chase"** might be represented by `[0.1, 0.6, -0.1, ...]`.
- **"cats"** might be represented by `[0.3, -0.2, 0.4, ...]`.

These vectors are positioned in such a way that the dot product of any two vectors gives us insight into their semantic relationship. For instance, the vectors for "Dogs" and "bark" would be closer together compared to "Dogs" and "garden," reflecting that "Dogs" and "bark" are more semantically related.

GloVe Vectors: Normally, you would load pre-trained GloVe vectors from a file (e.g., GloVe 6B).

Sentence Representation: The vectors for each word in a sentence are summed to create a vector representation of the entire sentence. This sentence vector captures the overall meaning of the sentence in the context of the words it contains.

When using GloVe vectors trained on the “GloVe 6B” corpus, the dimensionality of the word vectors can vary based on the specific version you choose. The GloVe 6B dataset is pre-trained on 6 billion tokens (words) from Wikipedia and Gigaword 5, and it offers several options for dimensionality. The available dimensions are:

- **50 dimensions:** Smaller vectors that are faster to train and use, but may capture less detailed semantic information.
- **100 dimensions:** A balanced choice, often used in many applications, providing a good trade-off between performance and resource requirements.
- **200 dimensions:** Provides more detailed word representations, capturing more nuanced relationships.
- **300 dimensions:** The most detailed among the options, useful for tasks requiring high precision in word embeddings.

Advantages and Disadvantages of GloVe

Advantages:

- **Captures Global Context:** GloVe captures both global and local context by focusing on word co-occurrence statistics across the entire corpus.

- **Semantic Relationships**: GloVe can capture complex semantic relationships, such as analogies (e.g., “king” is to “man” as “queen” is to “woman”).
- **Efficient Training**: Once trained, GloVe embeddings can be used for a variety of tasks, and training is generally faster and less resource-intensive than models like deep neural networks.

Disadvantages:

- **Requires Large Corpus**: GloVe requires a large and diverse corpus to effectively capture meaningful word relationships.
- **Fixed Vectors**: Once trained, GloVe vectors are static, which means they cannot adapt to new words or contexts not seen during training.
- **No Contextualization**: Unlike models like BERT, GloVe does not account for different meanings of a word in different contexts.

FastText

> *FastText, developed by Facebook’s AI Research (FAIR) lab, is an extension of Word2Vec that improves upon traditional word embeddings by incorporating subword (character n-gram) information. This approach allows FastText to create better embeddings for rare and out-of-vocabulary words and handle morphological variations.*

Steps in FastText

1. **Tokenization**: FastText first tokenizes the input text. For our sentences, the tokens would be:

- “Dogs chase cats in the garden.”

- Tokens: [‘Dogs’, ‘chase’, ‘cats’, ‘in’, ‘the’, ‘garden.’]
- “Dogs bark loudly at night.”
- Tokens: [‘Dogs’, ‘bark’, ‘loudly’, ‘at’, ‘night.’]

2. Subword Generation: FastText breaks down each word into character n-grams. For example, if n=3 (trigrams):

- **“Dogs”**: ‘Dog’, ‘ogs’, ‘Do’, ‘ogs’, ‘gs’
- **“chase”**: ‘cha’, ‘has’, ‘ase’, ‘ch’, ‘ase’
- **“cats”**: ‘cat’, ‘ats’, ‘ca’, ‘ts’
- **“garden.”**: ‘gar’, ‘ar’, ‘den’, ‘den.’, ‘ar’, ‘den’
- **“bark”**: ‘bar’, ‘ark’, ‘ba’, ‘rk’
- **“loudly”**: ‘lou’, ‘oud’, ‘udl’, ‘dly’, ‘l’, ‘ly’

3. Context Windows: FastText uses context windows to generate training examples. For a given window size, it extracts context around each target word. For instance, with a window size of 2:

- In “Dogs chase cats in the garden.”:
- Context of ‘chase’: (‘Dogs’, ‘cats’, ‘in’)
- Context of ‘cats’: (‘chase’, ‘in’, ‘the’)
- In “Dogs bark loudly at night.”:
- Context of ‘bark’: (‘Dogs’, ‘loudly’, ‘at’)
- Context of ‘loudly’: (‘bark’, ‘at’, ‘night’)

4. Embedding Representation:

- **Subword Embeddings**: Each subword is mapped to a vector in the embedding space. For example, the trigram ‘Dog’ and ‘ogs’ have their respective embeddings.
- **Word Embeddings**: The final word embedding is computed by aggregating the embeddings of its subwords. This can be done by averaging or summing the vectors of the subwords.

5. Training:

- FastText trains using a skip-gram or CBOW approach, similar to Word2Vec. The model learns to predict the context words from a target word (skip-gram) or predict the target word from context words (CBOW).
- It adjusts the vectors to minimize the prediction error, learning embeddings that capture semantic relationships.

Example Breakdown

For the sentence “Dogs chase cats in the garden.”:

Subword Embeddings:

- ‘Dogs’: [‘Dog’, ‘ogs’, ‘Do’, ‘gs’]
- ‘chase’: [‘cha’, ‘has’, ‘ase’]
- ‘cats’: [‘cat’, ‘ats’]
- ‘garden.’: [‘gar’, ‘den’, ‘den.’]
- Each subword has an embedding vector. For instance, ‘Dog’ might have a vector `[0.2, -0.1, 0.3, ...]` and 'ogs' might have `[0.1, 0.0, 0.2, ...]`.

Word Embeddings:

- The word embedding for ‘Dogs’ is derived from averaging or summing the embeddings of its subwords: `[mean(0.2, 0.1, ...), mean(-0.1, 0.0, ...), ...]`.

For the sentence “Dogs bark loudly at night.”:

Subword Embeddings:

- ‘bark’: [‘bar’, ‘ark’]
- ‘loudly’: [‘lou’, ‘oud’, ‘dly’]
- Similarly, embeddings for each subword are learned and aggregated.

Word Embeddings:

- The embedding for ‘bark’ is obtained by combining the embeddings of ‘bar’ and ‘ark’: `[mean(0.3, 0.2, ...), mean(-0.2, 0.1, ...), ...]`.

Advantages of FastText

1. **Handling Out-of-Vocabulary Words**: FastText can generate embeddings for words not seen during training by using subword information.
2. **Morphological Sensitivity**: It captures variations in word forms, making it suitable for languages with rich morphology.

By using FastText, you obtain embeddings that not only reflect the meanings of words but also consider their internal structure, providing richer representations that can improve performance in various NLP tasks.

ELMo

> *ELMo, developed by AllenNLP, provides context-dependent word embeddings. Unlike static embeddings (like Word2Vec and GloVe), which generate a single vector for each word regardless of context, ELMo produces embeddings based on the word's context within a sentence. This allows ELMo to capture more nuanced meanings.*

Key Concepts

1. **Bidirectional LSTM (BiLSTM):** ELMo uses a two-layer bidirectional LSTM to generate embeddings. This means it processes text in both forward and backward directions, capturing context from both sides of each word.
2. **Contextualized Embeddings**: The embeddings generated by ELMo are context-dependent. The same word can have different embeddings based on its surrounding words.

Example Sentences

1. **"The bank was crowded with people."**
2. **"She went to the bank to deposit some money."**

ELMo's Approach

Forward and Backward Passes:

1. **Forward Pass:** ELMo processes the sentence from left to right, capturing information from the words that follow the target word.

For Sentence 1 ("The bank was crowded with people."):

- After "bank," ELMo processes the words: "was," "crowded," "with," "people."

For Sentence 2 (“She went to the bank to deposit some money.”):

- After “bank,” ELMo processes the words: “to,” “deposit,” “some,” “money.”

2. Backward Pass: ELMo processes the sentence from right to left, capturing information from the words that precede the target word.

For Sentence 1:

- Before “bank,” ELMo considers: “The.”
- After “bank,” it considers: “was,” “crowded,” “with,” “people.”

For Sentence 2:

- Before “bank,” ELMo considers: “She,” “went,” “to,” “the.”
- After “bank,” it considers: “to,” “deposit,” “some,” “money.”

Hidden States and Concatenation:

Sentence 1:

- **Forward Hidden States:** ELMo captures the context of “bank” being related to people and a crowded place, potentially a riverbank or similar location.
- **Backward Hidden States:** ELMo incorporates the context that “bank” is part of a sentence discussing crowd and physical location.

Sentence 2:

- **Forward Hidden States:** ELMo understands “bank” in the context of financial transactions and depositing money.
- **Backward Hidden States:** ELMo incorporates the context that “bank” relates to financial operations, which changes its representation.

Concatenation and Projection:

- The hidden states from both forward and backward LSTMs are concatenated for each token.
- ELMo uses a final projection layer (a feed-forward neural network) to map these concatenated hidden states to the embedding space.

Contextualized Embeddings:

For Sentence 1:

- ELMo might generate an embedding for “bank” that reflects its role as a physical place or edge, perhaps including information about location or environment.

For Sentence 2:

- ELMo would generate a different embedding for “bank,” reflecting its role as a financial institution, including information about financial transactions and services.

For each token, ELMo produces a contextualized embedding by combining the information from both directions and projecting it into the final embedding space.

Key Points

1. **Contextualization**: ELMo's embeddings are dynamic and change depending on the surrounding words, providing more nuanced representations compared to static embeddings.
2. **Bidirectional Processing**: By incorporating context from both directions, ELMo captures richer semantic and syntactic information.
3. **Pre-trained Language Model**: ELMo is trained on a large corpus and can be fine-tuned for specific tasks, allowing it to provide embeddings that reflect broad linguistic knowledge.

By using ELMo, you obtain embeddings that encapsulate the meaning of words based on their specific contexts within sentences, enhancing performance on various NLP tasks.

Size of word vectors

> *The size (or dimensionality) of a word vector in models like Word2Vec, GloVe, or other word embedding techniques is a hyperparameter chosen during the training process. Here's how the size of the vector is determined and its implications:*

1. Choice of Dimensionality (d):

- **Hyperparameter Setting**: The dimensionality d of the word vectors is explicitly chosen by the person training the model. Common choices for d are 50, 100, 200, or 300 dimensions, but this can vary depending on the specific needs of the task or the computational resources available.

Trade-off Between Precision and Resources:

Higher Dimensionality (e.g., 300 or 400 dimensions):

- **Pros**: Captures more complex relationships between words, allowing the model to encode finer-grained semantic information.

- **Cons**: Requires more computational power and memory for both training and inference. It may also lead to overfitting if the training corpus is not large enough.
- **Lower Dimensionality (e.g., 50 or 100 dimensions)**:
- **Pros**: Faster to train and requires less memory. It's often sufficient for simpler tasks or when working with smaller datasets.
- **Cons**: May not capture as much semantic richness or complex relationships between words.

2. Corpus and Task Requirements:

Size and Diversity of the Training Corpus:

- A larger and more diverse corpus may benefit from higher-dimensional vectors to capture the varied contexts in which words appear.
- For smaller or more specific corpora, lower-dimensional vectors may suffice.

Downstream Task:

- Tasks requiring nuanced understanding, such as analogy reasoning or sophisticated text classification, may perform better with higher-dimensional vectors.
- Simpler tasks like basic sentiment analysis might not require as much dimensionality.

3. Training Algorithm and Model Complexity:

Model Type:

- Techniques like Word2Vec (Skip-Gram and CBOW) and GloVe allow for flexibility in choosing dimensionality.

- More complex models like transformer-based architectures (e.g., BERT) have inherent dimensionality that is typically higher and fixed.

Complexity of Relationships:

- The model's ability to learn complex semantic relationships is partly influenced by the vector's dimensionality. More dimensions allow for more nuanced representation of words' meanings.

Summary

> *In this blog, we delve into various methods for converting words into numerical vectors, essential for natural language processing (NLP) tasks. We start with* ***One-Hot Encoding (OHE)****, a basic technique that represents words as unique binary vectors, and* ***Bag of Words (BoW)****, which counts word occurrences without considering word order. We then explore* ***Term Frequency-Inverse Document Frequency (TF-IDF)****, which adjusts word frequency based on its importance across documents.*
>
> *Moving beyond these foundational methods, we examine* ***Word2Vec****, a popular model that learns word embeddings through continuous prediction tasks, and* ***GloVe (Global Vectors for Word Representation)****, which captures word relationships by factorizing the word co-occurrence matrix. We also cover* ***FastText****, an extension of Word2Vec that accounts for subword information, and* ***ELMo (Embeddings from Language Models)****, which provides context-dependent word embeddings by leveraging deep bidirectional language models.*
>
> *Each method is explained in detail, including their advantages, disadvantages, and applications, providing a comprehensive overview for choosing the appropriate technique based on specific NLP needs.*

NLP Word Embeddings Data Science Interview Word2vec

Written by Saket Kumar

Edit profile

10 Followers

Seasoned mechanical engineer looking to get into Data Scientist role.

More from Saket Kumar

Saket Kumar

How to become a Expert on Kaggle in 2 weeks!!

Before discussing how to become an expert on Kaggle, it's important to understand the...

Aug 11 5

Saket Kumar

NLP2: Mastering Text Handling: Types of Sentences and...

Types of Data in NLP

Sep 3 2