Everything Word Embeddings

Conceptual Overview

What is a word embedding?

A type of word representation that allows words with similar meaning to have similar vector representations. They map words or phrases from a vocabulary into a continuous vector space, usually in high dimensions.

The idea is to capture semantic meaning by embedding words into a dense, lower-dimensional space where words with similar meanings are closer together.

How do they work?

Context-based learning: Word embeddings are typically learned based on the contexts in which words appear. The core idea is that a word's meaning is defined by the words surrounding it (the distributional hypothesis).

Embeddings are generated by training a model to predict either the context of a word given the word itself or vice versa, as in models like Word2Vec or GloVe.

Why are they important?

Before embeddings, traditional methods like one-hot encoding or bag of words (BoW) were used to represent words. However, these methods suffer from a few limitations:

- High dimensionality: One-hot vectors are as long as the vocabulary, leading to very sparse representations.

- No semantic relationships: In one-hot encoding, words are treated as independent entities, meaning "cat" and "dog" would have no similarity despite being related concepts.

Word embeddings solve this by encoding semantic relationships between words. For example:

- Words that appear in similar contexts (like "king" and "queen") will have similar vector representations.

- Word embeddings allow mathematical operations like analogies, where the relationship between words can be captured (e.g., "king - man + woman = queen").

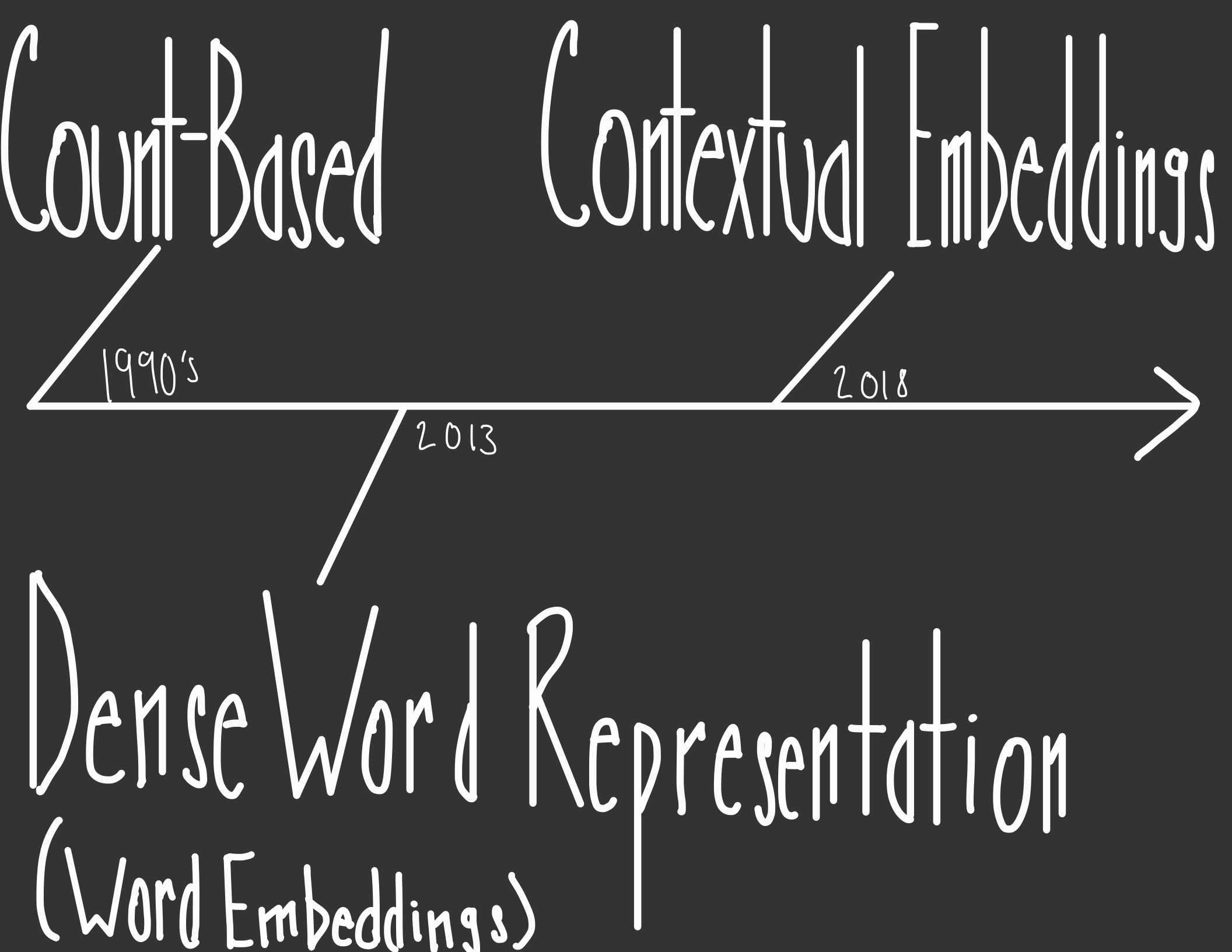
Their role in NLP

- **Semantic Similarity: Word embeddings allow models to understand the semantic relationships between words, making them key for tasks like search engines, machine translation, and sentiment analysis.**

- **Dimensionality Reduction: Instead of representing each word with thousands of features (as in one-hot vectors), embeddings reduce this down to hundreds of features while preserving meaning.**

- **Transfer Learning: Once learned, embeddings can be transferred and used in different NLP tasks without retraining the entire model, which saves time and computational resources.**

Model Exploration


Count-Based
Contextual Embeddings
1990's
2013
2018
Dense Word Representation
(Word Embeddings)

Early Models

Count-based models

Bag of Words (BoW):

- **Words are represented based on their frequency of occurrence in a document. Each word is treated independently, and there's no notion of word order or semantic relationships.**

- **Issues: High-dimensional (as many dimensions as there are words in the vocabulary) and lacks context or meaning across different words.**

TF-IDF (Term Frequency-Inverse Document Frequency):

- **Words are weighted by their frequency in a document and how unique they are in the entire corpus.**

- **Issues: Like BoW, TF-IDF treats words independently and doesn't capture word meaning or semantic relationships.**

Dense Word Representations

(Word Embeddings)

1. Word2Vec

- Developed by Google, Word2Vec was one of the first models to produce dense, low-dimensional vectors for words.

- Word2Vec uses predictive methods (Skip-gram and CBOW) to learn word embeddings by training on large corpora, capturing semantic relationships based on the context in which words appear.

2. GloVe (Global Vectors For Word Representation)

- Developed by Stanford, GloVe uses a count-based approach combined with matrix factorization techniques. It builds a co-occurrence matrix from the entire corpus and uses this to learn embeddings.

- GloVe aims to capture global statistical information about the corpus, focusing on word co-occurrence patterns.

3. FastText

- Developed by Facebook, fastText improves upon Word2Vec by taking into account subword information. Instead of learning vectors for entire words, it breaks words down into character-level n-grams (e.g., prefixes, suffixes).

- This allows fastText to handle rare words and out-of-vocabulary words better than Word2Vec and GloVe.

Word 2 Vec

Model deep dive

What is Word 2 Vec ?

A shallow, two-layer neural network that learns to represent words as dense vectors in a continuous vector space. The primary goal is to capture the semantic relationships between words based on the contexts in which they appear in the training data.

* Network Representation Img *

Smaller

Provides 2 key algorithms:

Skip-gram : Predicts the context words given a target word.

Continuous Bag Of Words (CBOW) : Predicts the target word given the context words.

Both methods result in word vectors (embeddings)

Skip - gram

Word prediction model used in Word2Vec that learns to represent words as vectors by predicting the context words surrounding a given target word. The central idea is that words appearing in similar contexts should have similar vector representations, thus capturing semantic relationships.

Objective

- Given a target word, the Skip-gram model predicts the words that appear in its context (i.e., the surrounding words).

- The goal is to maximize the probability of predicting the correct context words for a given target word.

How it works

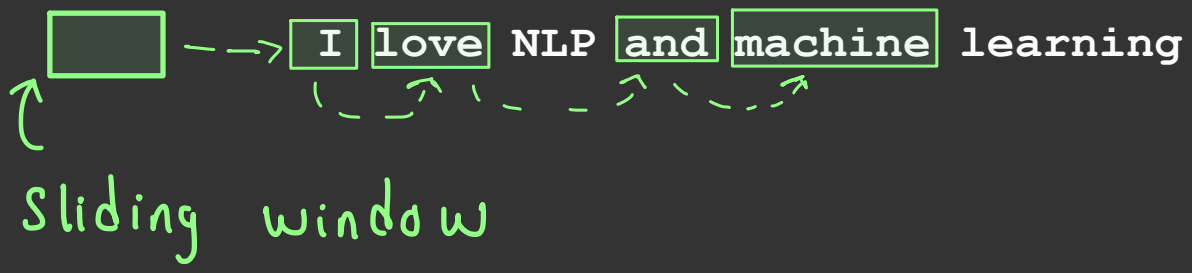
The model operates on a **sliding window** over a sequence of words in a text. The window defines the number of words before and after the target word that will be used as context.

Window size = 2

target word = 'NLP'

Obtained context words:

'I', 'love', 'and', 'machine'



Training Example

For each word in the corpus, Skip-gram creates a training example where the word in the middle is the input (the target word), and the words in its window are the output (the context words).

In our example sentence, if the target word is "NLP", the Skip-gram model will try to predict the words in its context:

- Input: "NLP"
- Output: ["love", "I", "and", "machine"]

The model learns to maximize the likelihood of predicting these context words given the input.

Training Process

1. Neural Network Structure
2. Softmax & Negative Sampling
3. SGD and Backprop

Neural Network Structure

Input layer

Input = target word (e.g., "NLP")
In practice, each word is represented as a one-hot encoded vector of size V, where V is the size of the vocabulary. If the vocabulary has 10,000 words, the vector is mostly zeros except for a 1 at the index of the target word.

- Example: If "NLP" is the 5000th word in the vocabulary, the one-hot vector for "NLP" will have a 1 at the 5000th position and 0s everywhere else.

Hidden layer

The one-hot encoded input is multiplied by the weight matrix to produce the word vector. The weight matrix has dimensions V x N, where V is the vocabulary size, and N is the dimensionality of the embeddings (e.g., 100 or 300 dimensions). This transforms the one-hot vector into a dense word vector (embedding).

- Example: If N=100, the one-hot vector for "NLP" is transformed into a 100-dimensional vector.

Output layer

The output layer is again a weight matrix (of size N x V), where the dense word vector is multiplied by the weights to predict the context words. The model tries to maximize the probability of predicting the correct context words using a softmax function, which converts the raw scores into probabilities.

Softmax & Negative Sampling

Softmax

Converts the output of the network into a probability distribution over all words in the vocabulary. It measures how likely each word in the vocabulary is to be one of the context words.

- For a large vocabulary, this step is computationally expensive because the softmax function involves normalizing over all words in the vocabulary.

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

- e = the base of the natural logarithm (Euler's number, approximately 2.718).
- z_i = the score (logit) for the i -th class.
- n = the total number of classes (in Word2Vec, this is the vocabulary size, V).
- p_i = the resulting probability for the i -th class.

Negative Sampling

To make the model efficient, Skip-gram uses negative sampling. Instead of computing probabilities for all words, negative sampling only updates a small sample of "negative" words (words that do not appear in the context).

- For each positive training example (correct word-context pair), a few negative examples (random words not in the context) are sampled and the model adjusts its weights for both positive and negative examples.
- Negative sampling is a key technique that speeds up training for large vocabularies.

Even more in depth softmax?

SGD (Stochastic Gradient Descent)

The Skip-gram model uses gradient descent to minimize the loss function, which measures how far the predicted context words are from the actual ones. After each training example (a word and its context), the model updates the weights to improve its predictions.

Backpropagation

Errors are propagated back through the network, and the weight matrices are adjusted accordingly. The word vectors (embeddings) are gradually updated so that words appearing in similar contexts end up with similar vectors.

Advantages

Rare word handling

better at learning meaningful representations for rare words because it treats each word-context pair as a separate example, allowing rare words to be trained individually.

Good for small datasets

works well on smaller datasets since it learns word-context pairs directly rather than averaging contexts (as CBOW does).

Rich semantic relationships

often captures richer semantic relationships between words, especially for less frequent words, because of the direct word-to-context prediction

Limitations

Computationally intensive

While Skip-gram captures fine-grained word relationships, it can be slow to train, especially with large vocabularies, due to the word-context predictions.

Sensitive to hyperparameters

The quality of the embeddings learned depends on the window size, dimensionality of vectors, and number of negative samples. Tuning these hyperparameters is crucial for good performance.

More in-depth example to tie it all together. Same word from the CROW example.

CBOW

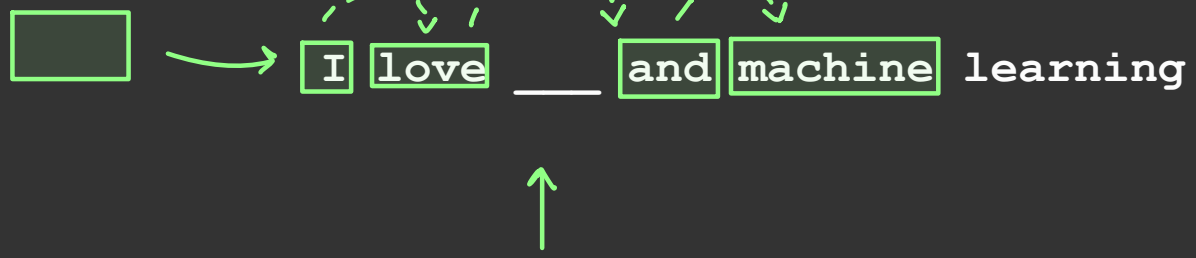
Word prediction model used in Word2Vec that aims to predict a target word given its surrounding context words.

Objective

It looks at a group of context words (words surrounding a missing word in a sentence) and tries to predict the word that should be in the center of those words. It uses the context of the surrounding words to infer the target word, learning embeddings for each word in the process.

How it works

Uses a sliding window over sentences. For each window, it uses the words before and after the target word as context, and the target word is the one in the center of the window.



Obtained context words:

'I', 'love', 'and', 'machine'

also input

what we want as output/ prediction ⟶ 'NLP'

Training Process

Neural Network Structure

Input layer

The input consists of the context words surrounding the target word. Each context word is represented as a one-hot encoded vector of size V, where V is the size of the vocabulary.

If the vocabulary contains 10,000 words, each word is represented as a vector of length 10,000, with a 1 in the position corresponding to the word and 0s everywhere else.

Hidden layer (Vector Aggregation)

The hidden layer aggregates the input vectors of the context words. This is typically done by averaging the word vectors of the context words.

- Example: If the context words are "I", "love", "machine", and "learning", their embeddings (dense vectors) are averaged to produce a single vector.

- This averaged vector is then passed through to the output layer, acting as the representation of the context as a whole.

Output layer (target word prediction)

The hidden layer aggregates the input vectors of the context words. This is typically done by averaging the word vectors of the context words.

- Example: If the context words are "I", "love", "machine", and "learning", their embeddings (dense vectors) are averaged to produce a single vector.

- This averaged vector is then passed through to the output layer, acting as the representation of the context as a whole.

Softmax

How probabilities are calculated

- In CBOW, once the context word embeddings have been aggregated (usually by averaging), the model uses a Softmax function in the output layer to predict the target word.
- The output is a probability distribution over all possible words in the vocabulary, and Softmax ensures that the sum of these probabilities is equal to 1.

Why Softmax is used

- Softmax transforms the raw scores (logits) into probabilities, allowing the model to rank the likelihood of each word being the target word.

- The model is trained to maximize the probability of predicting the correct target word given the surrounding context words.

Just like in Skip-gram, the Softmax function converts the logits into a probability distribution, making it easier for the model to learn the relationships between words and their contexts.

CBOW Advantages

Faster to train

CBOW tends to be faster to train than Skip-gram because it averages context word embeddings, rather than predicting each context word individually. This reduces the number of predictions the model has to make, making it computationally efficient.

Works well with large datasets

CBOW is particularly effective when training on large corpora of text, as it can quickly learn word embeddings due to its efficiency in processing multiple context words at once.

Robust to noisy data

Since CBOW averages the context words, it is less sensitive to noisy context words. This means that it can still produce reasonable predictions even if some of the surrounding words don't strongly relate to the target word.

Limitations

Less effective for rare words

CBOW tends to perform poorly for rare words because it predicts the target word based on the averaged context. Rare words may not appear frequently enough in the training data for the model to learn strong association

Loss of granular information

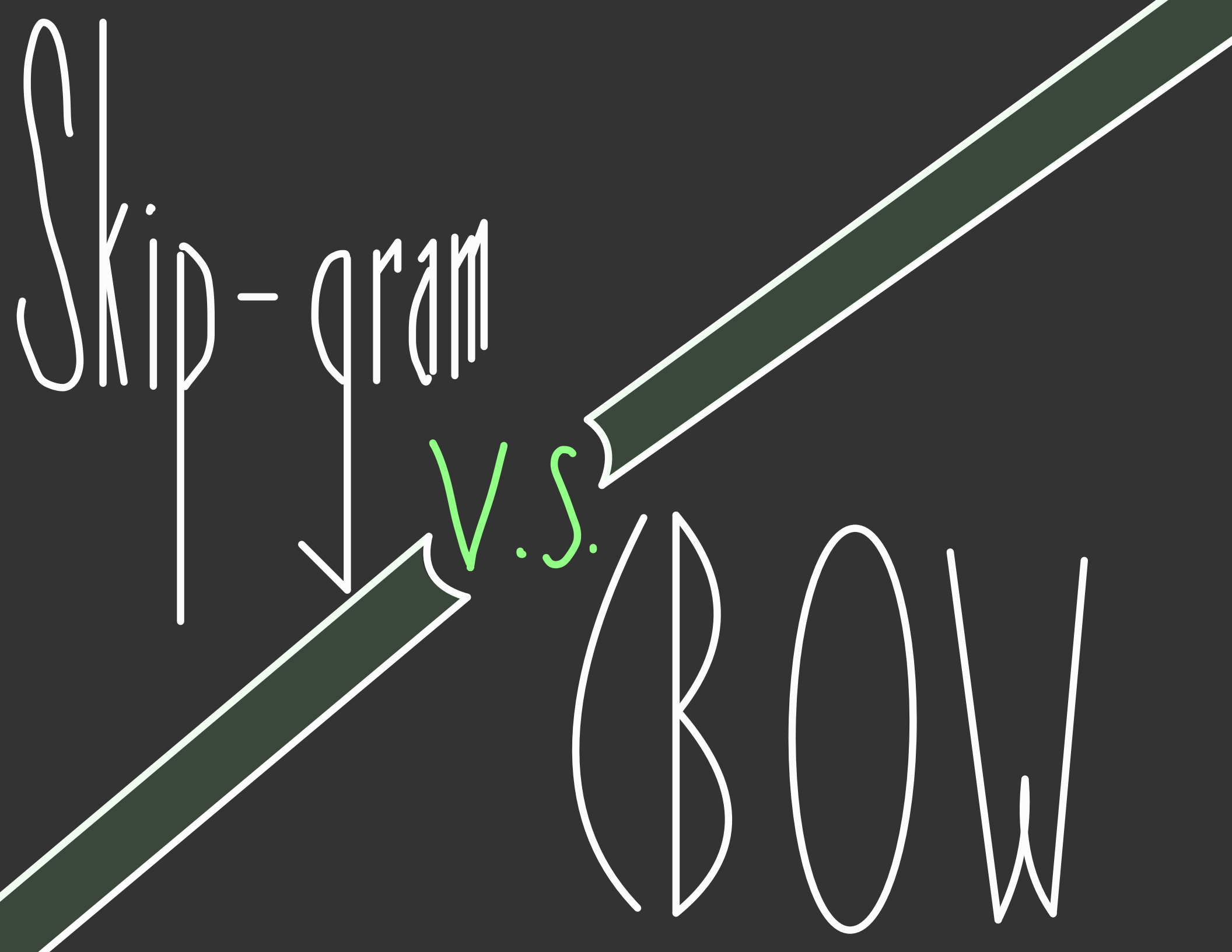
By averaging the context word vectors, CBOW can lose some fine-grained semantic information. In cases where different context words carry very different meanings, the model might fail to capture the subtle distinctions between them.

Context aggregation might dilute information

Since CBOW aggregates all context words into a single vector, it may dilute the importance of more informative context words. Each word in the context window contributes equally to the prediction, which can lead to suboptimal predictions when certain context words are more relevant than others.

✳ CBOW Example ✳

Example Continuation


Skip-gram
V.S.
CBOW

Similarities

Both algorithms are used in Word2Vec to learn embeddings.

Both are predictive they use context information to predict words in text.

Both ultimately produce dense, low- dimensional word vectors that capture semantic relationships between words.

Both use a simple, shallow neural network with an input, hidden, and out-it layer with weights corresponding to the learned word embeddings.

Both use softmax to convert raw scores into probabilities or employ negative sampling to speed training up.

Differences

Objective:

Skip-gram: Creates more training examples, as it generates one training example for each context word. This allows it to learn better representations for rare words.

CBOW: Generates fewer training examples by averaging context words, which can speed up training but is less precise when dealing with rare words.

Training Speed :

Skip-gram: Slower to train because it predicts each context word individually for a given target word.

CBOW: Faster to train as it predicts the target word by averaging context words in a single step.

Rare-word handling:

Skip-gram: Performs better with rare words because it treats each word-context pair independently.

CBOW: Struggles with rare words because it relies on averaging, and rare words might not appear in enough contexts to have strong representations.

Can they be used complementarily?

No, in a single Word2Vec training process, you use either Skip-gram or CBOW, not both simultaneously. However, they can be considered complementary in different situations based on the nature of the dataset and task:

Complementary Usage:

If you're training multiple models for different tasks, you could use Skip-gram where the focus is on learning fine-grained word relationships (like rare word representation), and use CBOW when speed and large-scale training are more important.

For different stages of an NLP pipeline, you might train a Skip-gram model when dealing with rare or domain-specific terms (e.g., medical terms) and a CBOW model when you need general-purpose embeddings for common words.

When to use Skip-gram:

Small Datasets: Skip-gram works well on smaller datasets because it learns directly from each word-context pair.

Rare Words: If your dataset contains many rare words, or words that are important but appear infrequently, Skip-gram can better learn their relationships by focusing on individual word-context predictions.

Need for Detailed Relationships: Skip-gram is better when capturing fine-grained relationships between words is crucial, such as in domains with technical vocabulary or for tasks where subtle word distinctions matter.

When to use CBOW:

Large Datasets: CBOW is better suited for large corpora because it trains faster by predicting the target word in a single step.

Noisy Data: If your dataset contains noisy data, where some context words might be irrelevant or erroneous, CBOW's averaging of context words makes it more robust to noise.

Common Words: If the task mainly involves common words, CBOW can efficiently learn their embeddings without needing the precision Skip-gram offers for rare words.

Speed and Scalability: When speed is a priority (e.g., in real-time applications or large-scale embedding learning), CBOW's faster training time makes it the better choice.

Word 2 Vec

Code Walkthrough

Sentences = list of lists

tokenized sentences

Context window size

# of CPU cores used for training.

```
model = Word2Vec(sentences, vector_size=10, window=3, min_count=1, workers=4)
```

Word2Vec instance

Each word is represented as a 10-dimensional vector

minimum # of ocurrences to be included in corpus

Window example, embedding for 'NLP'.

['I', 'love', 'NLP', 'and', 'machine', 'learning', ...]

Window is "slid" through the sentence, capturing n number of words before and after the target word.

```
word_embeddings = model.wv
```

- Accessing the trained word vectors from the Word2Vec model. The model.wv attribute contains the learned word embeddings (vectors) for every word in the vocabulary.

- It contains the mappings between words and their corresponding embeddings.

Dictionary

```
{
    'NLP': [-0.2345879,  0.1456735,  0.8764567, -0.3435482,
0.2673410,
            0.1578943, -0.9756432,  0.4567281, -0.2347890,
0.7895672],
    'machine': [0.4568721, -0.8975642,  0.3456871, 0.5673423,
-0.6789341,
                0.1257645,  0.3986723, -0.7861231, 0.2345671,
-0.9845678],
    'learning': [-0.2345879,  0.1456735,  0.8764567,
-0.3435482, 0.2673410,
            0.1578943, -0.9756432,  0.4567281, -0.2347890,
0.7895672],
    'word': [0.4568721, -0.8975642,  0.3456871, 0.5673423,
-0.6789341,
                0.1257645,  0.3986723, -0.7861231, 0.2345671,
-0.9845678],
}
```

```
word_vector = word_embeddings['NLP']
```

- Retrieving the word vector for the word "NLP" from the word_embeddings object (model.wv)

dense vector

```
[-0.2345879,  0.1456735,  0.8764567, -0.3435482, 0.2673410,
  0.1578943, -0.9756432,  0.4567281, -0.2347890, 0.7895672]
```

each value represents a feature the model learned based on the words appearance context

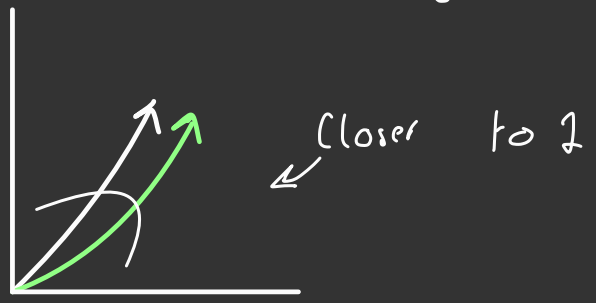
Semantically similar words will have vectors pointing in similar directions

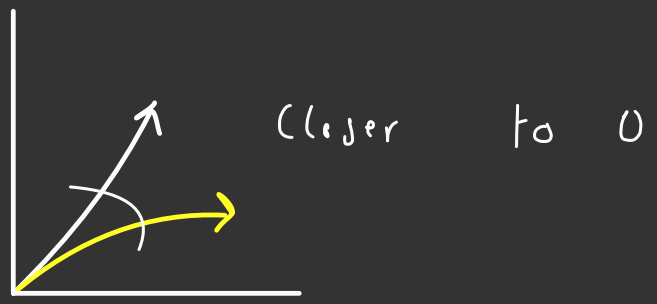
apple
king
man

```
similar_words = word_embeddings.most_similar('NLP')
```

- Uses the most_similar() function from Gensim's KeyedVectors object to find the words in the vocabulary that are closest to 'NLP' in the vector space. It does this by calculating cosine similarity between the vector for 'NLP' and all other word vectors in the model.

Cosine similarity: measures the angle between 2 vectors





The output tuple contains the words and their similarity score. The highest scores will be words whose vectors form the smallest angle with the target word.

Contextual Embeddings

As NLP tasks became more complex, it became clear that static embeddings (like Word2Vec, GloVe, and fastText) were limited because they assigned a single vector to each word, regardless of context. This gave rise to contextual embeddings, where the representation of a word changes depending on its sentence or context.

ELMo

- ELMo introduced contextualized word embeddings, meaning the embedding for a word depends on the entire sentence it's used in.

- It uses a bi-directional LSTM architecture, capturing context from both directions (left to right and right to left)

BERT

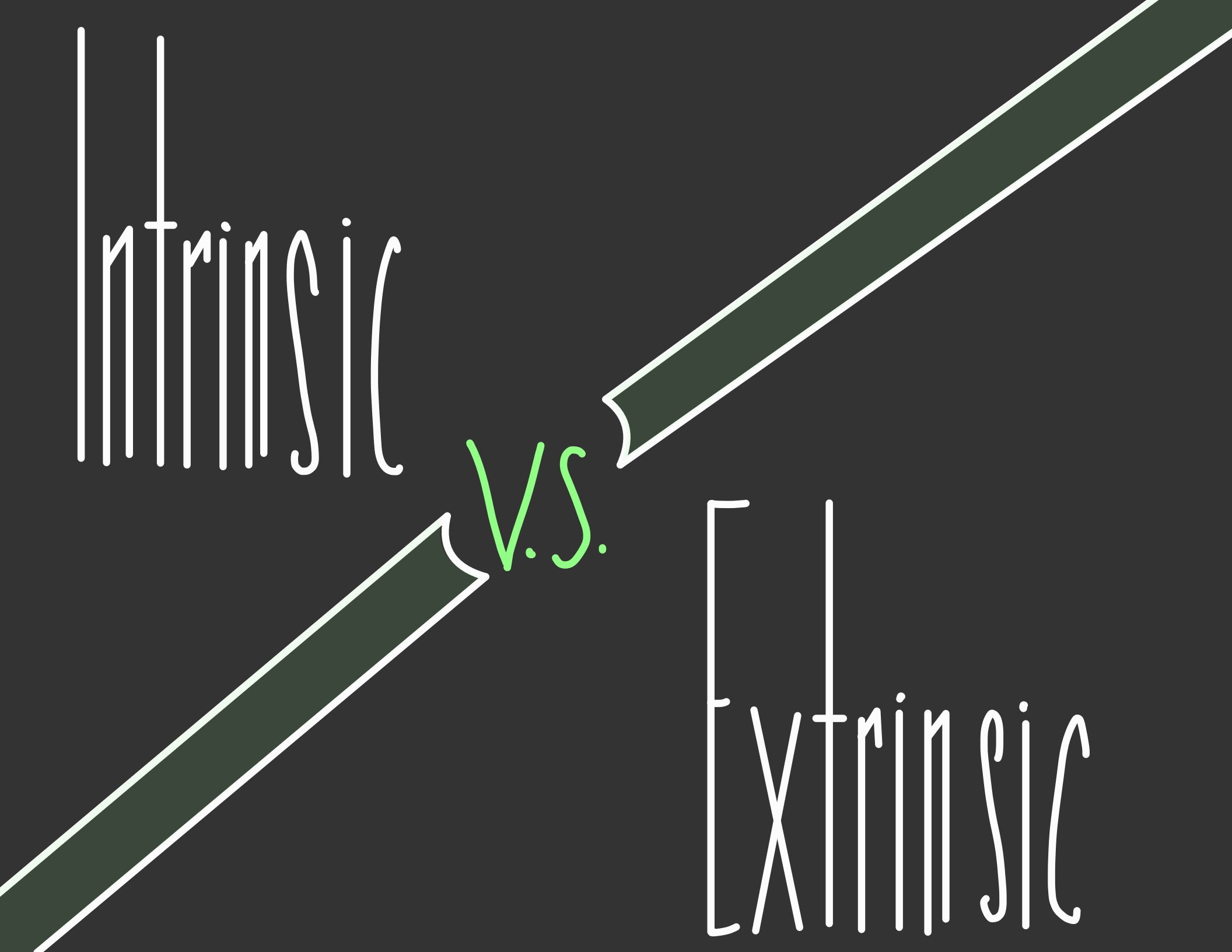
- BERT introduced the transformer architecture and is considered one of the most powerful contextual models.

- BERT generates embeddings that are context-dependent, where each occurrence of a word has a unique embedding depending on the surrounding context. This allows it to handle words with multiple meanings (e.g., "bank" in "river bank" vs. "money in the bank").

GPT

- GPT (and subsequent models like GPT-2 and GPT-3) builds on transformers, but it is designed primarily for text generation. It uses an autoregressive model to predict the next word in a sequence, generating embeddings based on context.

Quality Evaluation


Intrinsic
V.S.
Extrinsic

Intrinsic evaluation

Intrinsic evaluation measures how well word embeddings capture linguistic properties or semantic relationships within the embeddings themselves, without applying them to any specific task. Common intrinsic tasks include:

Word Similarity tasks

These tasks evaluate whether words that are semantically similar have similar vectors. For example, we expect the embeddings for "king" and "queen" to be closer in the vector space than "king" and "banana."

The evaluation is usually done by comparing word pair similarity scores (from embeddings) to human judgment scores.

Word Analogy tasks

These tasks test whether embeddings capture relationships between words, such as analogies. For example, using vector arithmetic, we expect the result of the equation:

$$king - man + woman \approx queen$$

The model is evaluated based on how well it predicts the correct word for given analogies.

Clustering

This evaluation checks whether similar words cluster together in the vector space. A good set of word embeddings will naturally create clusters of related words (e.g., “apple”, “orange”, and “banana” should cluster as fruits).

t-SNE Visualization

t-SNE (t-distributed stochastic neighbor embedding) is a popular technique used to visualize word embeddings in a 2D or 3D space. While this isn’t a quantitative measure, it allows researchers to visually inspect whether words with similar meanings cluster together.

Exntrinsic evaluation

Extrinsic evaluation tests the usefulness of word embeddings in downstream NLP tasks. This is the most practical way of evaluating how effective word embeddings are, as they are embedded into real applications. Common extrinsic evaluation tasks include:

Text classification

- Word embeddings are used as features in a text classification model, and the accuracy of the classifier is evaluated.

- Example Task: Sentiment analysis, where embeddings are used to classify whether a sentence expresses positive or negative sentiment.

Named Entity Recognition

- Word embeddings are evaluated based on their performance in recognizing and classifying proper nouns in text (e.g., names of people, organizations, places).

- Embeddings that capture richer semantic relationships will typically perform better in NER tasks.

Machine translation

Word embeddings are used in models that translate text from one language to another. Evaluation metrics like BLEU (Bilingual Evaluation Understudy) score measure how accurately the embeddings contribute to translation quality.

Question answering

Word embeddings are tested based on how well they help models answer questions based on a given text. For example, BERT embeddings excel in this area due to their contextual nature.

Evaluation Metrics

Cosine similarity

Cosine similarity is a commonly used metric to measure the similarity between word vectors. It ranges from -1 (completely dissimilar) to 1 (completely similar).

$$\text{Cosine similarity} = \frac{A \cdot B}{|A||B|}$$

Where A and B are word vectors, and the dot product is divided by the magnitudes (norms) of the vectors.

Accuracy

In extrinsic tasks (e.g., text classification, NER), embeddings are evaluated based on how well the overall model performs. Common metrics include accuracy, F1 score, and precision/recall.

Spearmans Rank Correlation

For word similarity tasks, the Spearman's rank correlation measures how well the predicted similarity scores (from embeddings) match human annotated scores.

BLEU Score

Used in machine translation to measure how well the generated text matches human-translated text.

Challenges & Limitations

Static nature of traditional word embeddings:

Single representation for each word

Traditional word embeddings like Word2Vec and GloVe assign a single vector to each word, regardless of the context in which it appears. This can lead to problems when dealing with polysemous words (words with multiple meanings).

lack of contextual information

Static embeddings cannot handle nuances in meaning that arise from different contexts, limiting their ability to accurately represent word meanings in complex sentences or documents.

Bias

Embedding bias:

Word embeddings are often trained on large corpora of text, such as news articles, books, or the web, which can contain societal biases. As a result, word embeddings may inadvertently learn and reflect these biases, leading to biased outcomes in downstream NLP tasks.

Rare and OOV words

Rare words

Traditional word embeddings often struggle with rare words or words that do not appear frequently in the training corpus. These words may not have enough training data to generate meaningful embeddings.

Out-of-Vocabulary Words

Static embeddings cannot generate vectors for words that were not present in the training data. This becomes a challenge when dealing with domain-specific terms, new slang, or words from different languages.

Computational and resource demand

Large-scale models

Contextual word embedding models like BERT and GPT are computationally expensive to train and fine-tune. They require substantial hardware resources (e.g., powerful GPUs) and large datasets to achieve state-of-the-art performance.

Inference time

Even after training, using models like BERT for inference can be slower than traditional methods due to their complex architecture and larger size.

Energy Consumption

Training large NLP models that rely on embeddings can be energy-intensive, contributing to concerns about the environmental impact of large-scale AI models.

World knowledge capturing

Lacks of real world understanding

Word embeddings capture the statistical relationships between words in text but do not have access to world knowledge or common sense reasoning. This can lead to errors in tasks that require a deeper understanding of context or factual correctness.

Training and fine-tuning

Domain adaptation

Word embeddings trained on general-purpose corpora (e.g., Wikipedia, news articles) may not perform well in specialized domains such as medicine, law, or finance. Adapting these embeddings to domain-specific tasks requires fine-tuning on specialized datasets, which may not always be available.

Catastrophic forgetting

When fine-tuning pre-trained embeddings on a new task or domain, the model may forget some of the general-purpose knowledge it previously learned, which can hurt performance on other tasks.

Sequence level tasks

Long sequences

Traditional word embeddings (like Word2Vec and GloVe) work well for representing individual words but struggle with understanding long sequences or sentences. Models like BERT and GPT have improved this by capturing dependencies across longer contexts, but even these models have limitations when it comes to handling extremely long documents or conversations.