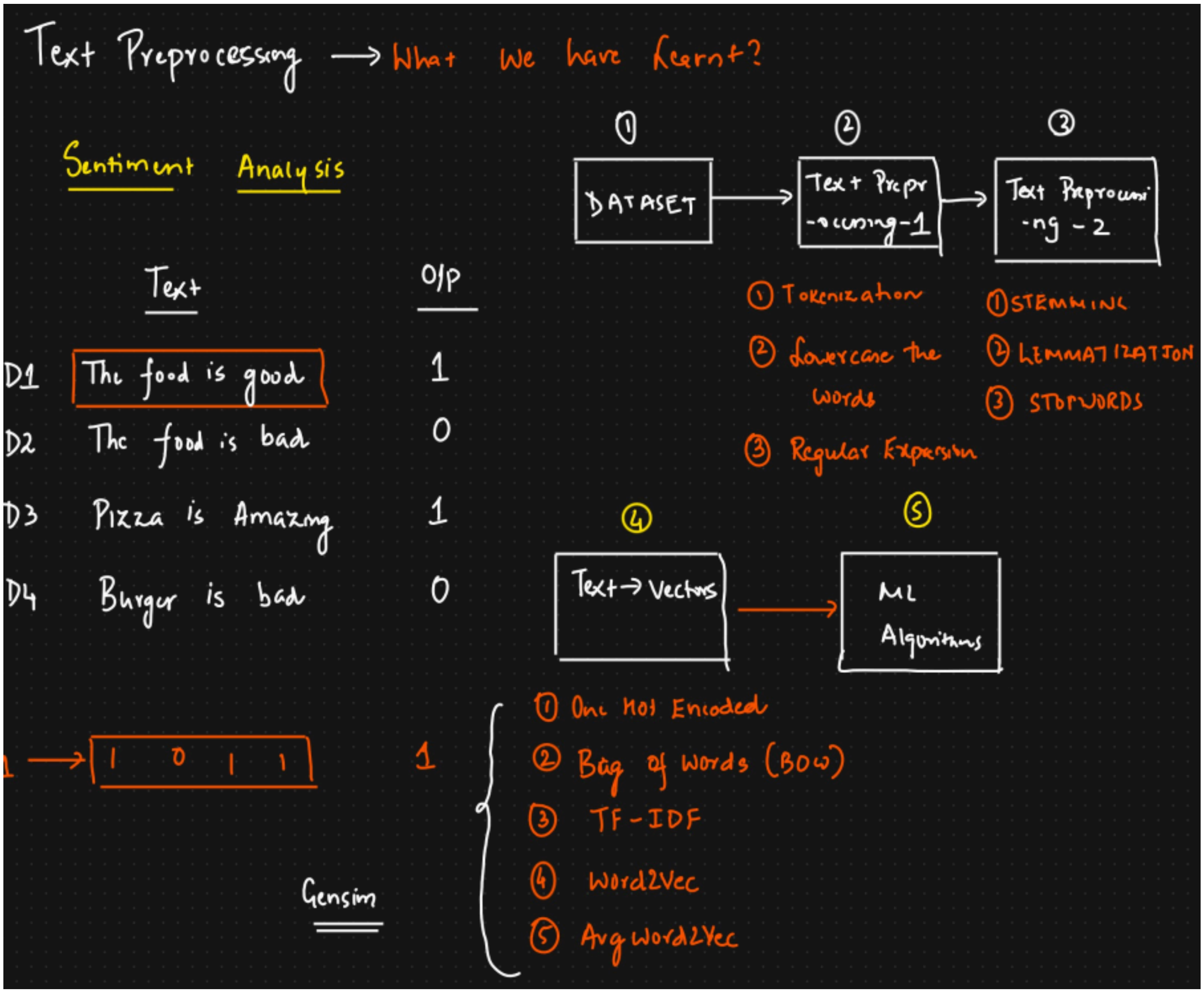


🧠 One-Hot Encoding (OHE)

📌 Definition:

One-Hot Encoding is a technique to convert categorical variables (like words or labels) into a numerical format that can be fed into machine learning models. In NLP, it is used to represent words as vectors.

🔢 How It Works:

- Given a vocabulary of size `V`, each word is represented as a `V`-dimensional vector.
- The vector has:
  - `1` at the index of the word
  - `0` everywhere else

📘 Example:

Vocabulary = `["apple", "banana", "cherry"]`
Each word is assigned an index:

```
"apple"  → [1, 0, 0]
"banana" → [0, 1, 0]
"cherry" → [0, 0, 1]
```

So, the word **"banana"** is represented as:
`[0, 1, 0]`

📈 Visual Representation:

```
Vocabulary:          One-Hot Vector:
----------------     -----------------
apple                [1, 0, 0, 0, 0]
banana               [0, 1, 0, 0, 0]
cherry               [0, 0, 1, 0, 0]
date                 [0, 0, 0, 1, 0]
elderberry           [0, 0, 0, 0, 1]
```

⚙️ Use Case in NLP:

- Represent words for **Bag-of-Words models**
- Input format for simple models (e.g., Naive Bayes, Logistic Regression)
- Useful in small vocabularies and when training simple models

✅ Advantages:

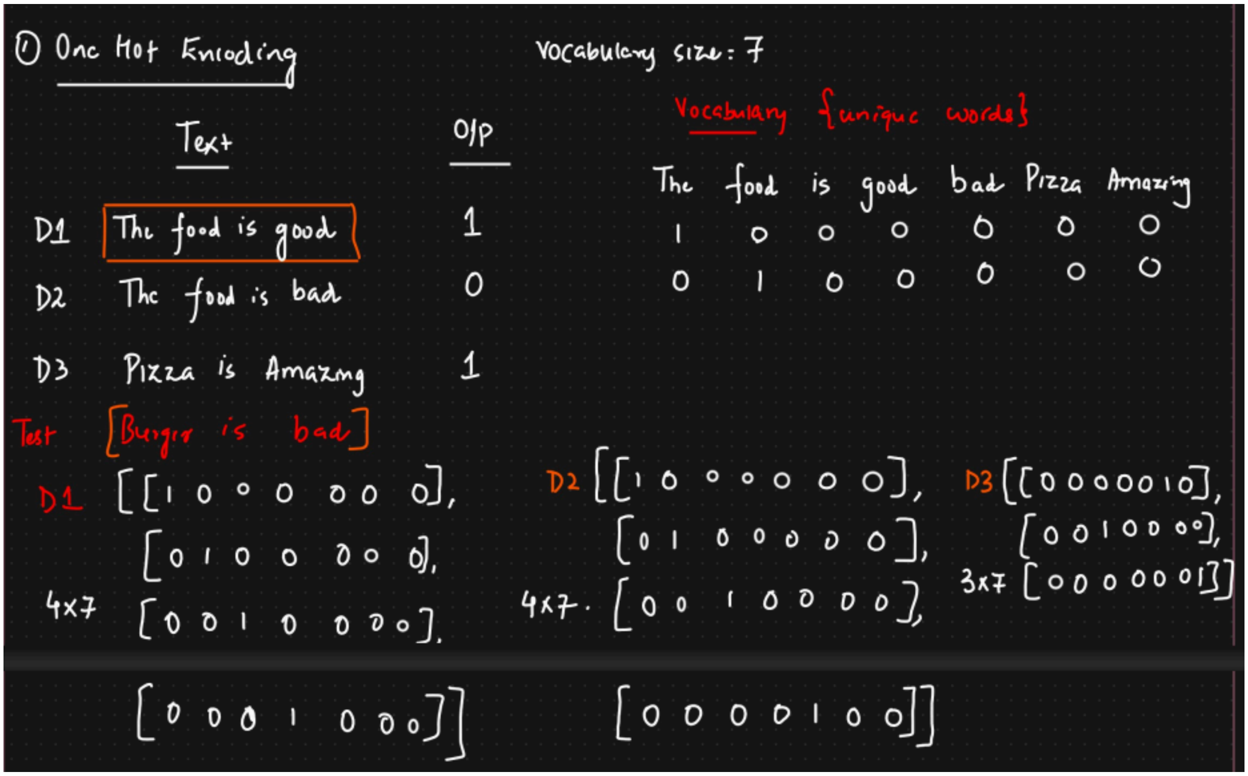
- Simple and intuitive
- Works well for small vocabularies or labels

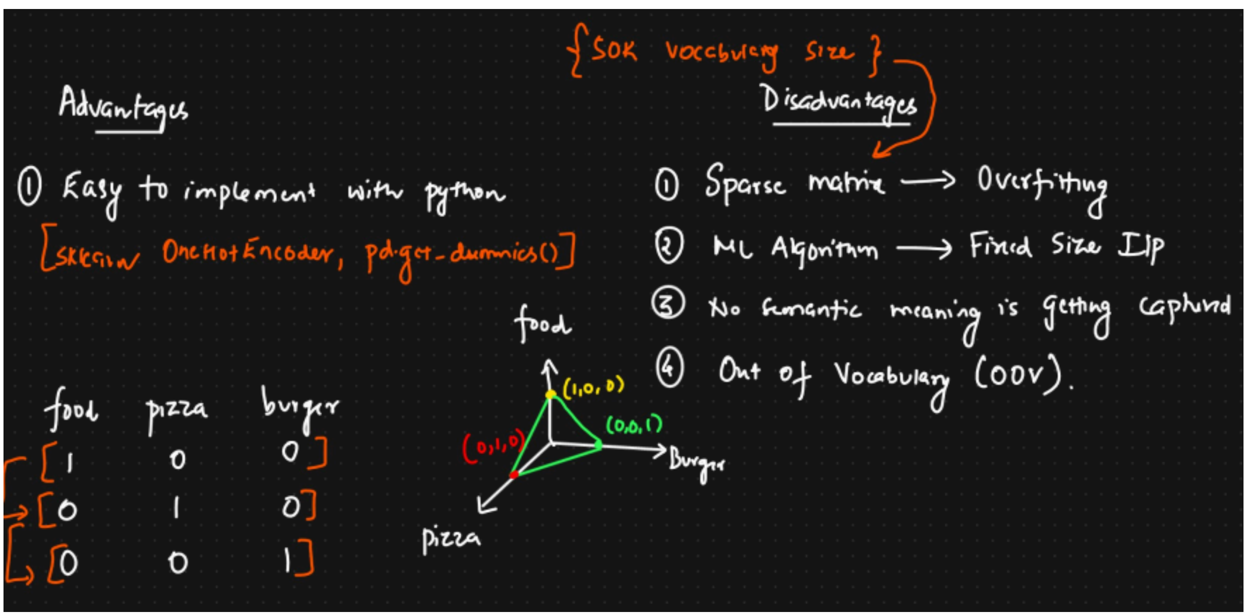
❌ Limitations:

1. **High Dimensionality**: For large vocabularies (like in NLP), the vector becomes very sparse and memory-inefficient.
2. **No Semantic Meaning**: It does not capture relationships between words (e.g., "king" and "queen" are just as different as "king" and "banana").
3. **Scalability Issues**: As vocabulary size increases, computation becomes inefficient.

💡 Tip:

Avoid one-hot encoding for large NLP tasks — prefer **word embeddings** or **pre-trained transformer embeddings** like BERT or Word2Vec.





🧠 Bag of Words (BoW)

📌 Definition:

Bag of Words is a simple and commonly used technique in NLP to represent text data (documents, sentences) as numerical vectors by counting word occurrences — ignoring grammar and word order.

🧱 Key Idea:

- Treat each document as a “bag” (multiset) of its words.
- Build a **vocabulary** (a set of unique words from the corpus).
- Represent each document as a vector of word frequencies.

📘 Example:

Corpus (3 sentences):

1. “I love NLP”
2. “NLP is fun”
3. “I love machine learning”

Vocabulary = [`I`, `love`, `NLP`, `is`, `fun`, `machine`, `learning`] (size = 7)

Sentence	Vector Representation (BoW)
“I love NLP”	`[1, 1, 1, 0, 0, 0, 0]`
“NLP is fun”	`[0, 0, 1, 1, 1, 0, 0]`
“I love machine learning”	`[1, 1, 0, 0, 0, 1, 1]`

Each position in the vector corresponds to a word in the vocabulary.
The values represent the **count of each word** in the sentence.

📊 Visual Overview:

```
Vocabulary Index:

["I", "love", "NLP", "is", "fun", "machine", "learning"]
```

```
       ↓        ↓      ↓      ↓      ↓        ↓          ↓

Sentence → [Count_I, Count_love, ..., Count_learning]
```

✅ Advantages:

- Simple to understand and implement.
- Converts text into a fixed-length numerical vector (good for ML models).

❌ Limitations:

1. **No word order**: Cannot understand phrases or context (e.g., "not good" ≠ "good").
2. **Sparse vectors**: High dimensionality for large vocabularies.
3. **No semantic meaning**: Cannot understand similarity between words.
4. **Vocabulary explosion**: Large corpus → large vocabulary → memory issues.

🧠 When to Use:

- For baseline NLP models
- When interpretability is important
- In combination with feature selection (e.g., removing stopwords, limiting vocabulary size)

💡 Tips:

- **Preprocess text** (lowercase, remove punctuation, stopwords, lemmatize) before building BoW.
- Limit vocabulary with `max_features` or `min_df`/`max_df` in tools like Scikit-learn.

② Bag of Words

Dataset

Text	O/P
He is a good boy	1
She is a good girl	1
Boy and girl are good	1

lower all the words case

⟹

Stopwords

S1 → good boy

S2 → good girl

S3 → Boy girl good

good

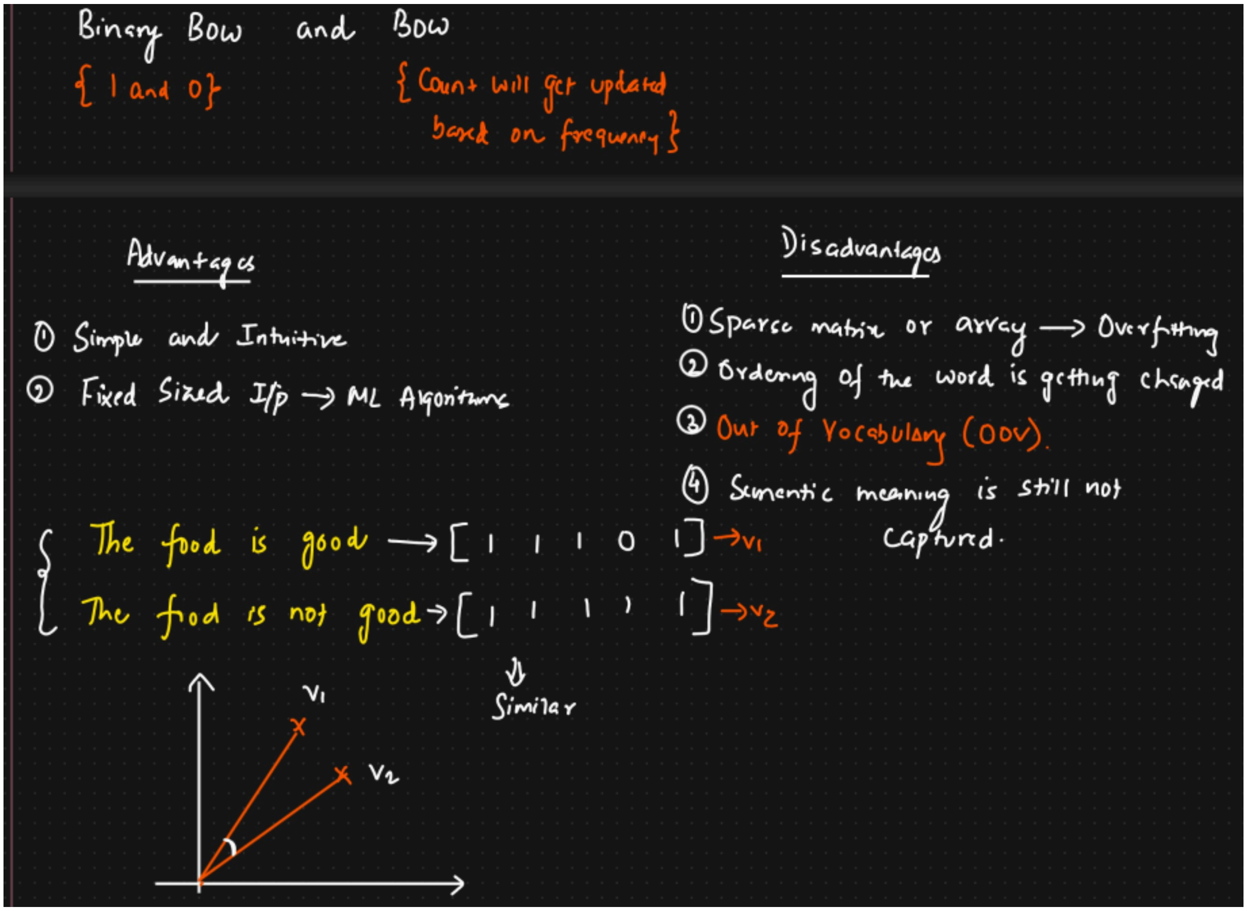
~~School~~

Test

Vocabulary	frequency
good	3
boy	2
girl	2

⟹

	good	boy	girl	O/P
S1	1	1	0	1
S2	1	0	1	1
S3	1	1	1	1

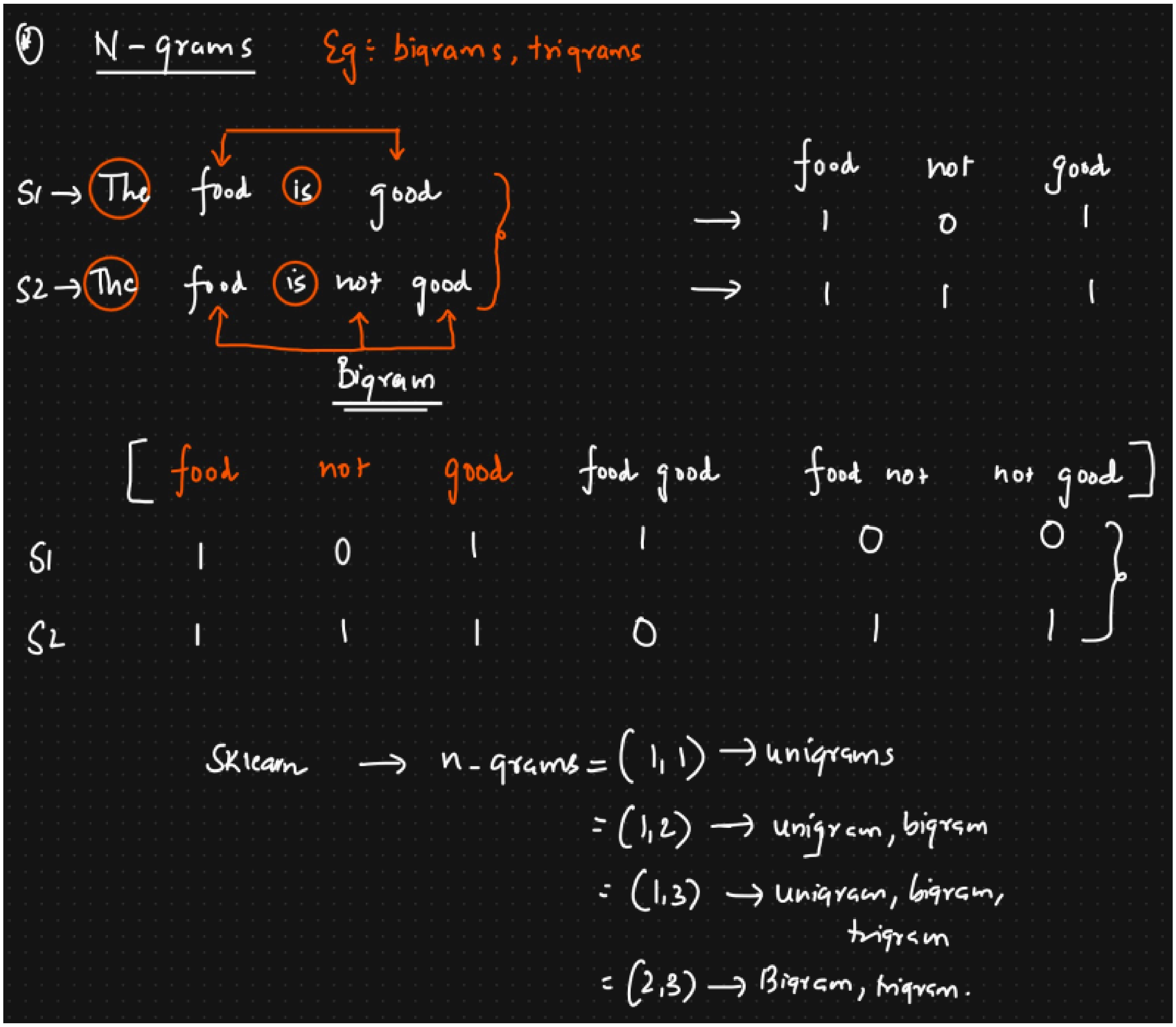


An **N-gram** is a **contiguous sequence of N items (usually words or characters)** from a given text. It helps capture **local context and word order**, unlike one-hot or simple BoW.

🔢 What is "N"?

- **Unigram (1-gram)**: Single word
  → "I love NLP" → `["I", "love", "NLP"]`
- **Bigram (2-gram)**: Sequence of 2 words
  → "I love NLP" → `["I love", "love NLP"]`
- **Trigram (3-gram)**: Sequence of 3 words
  → "I love NLP" → `["I love NLP"]`

General form: `n-gram = sequence of n tokens (words/chars)`



📊 TF-IDF in NLP

📌 Definition:

TF-IDF stands for **Term Frequency–Inverse Document Frequency**.
It is a numerical statistic that reflects how **important a word is** to a **document** in a **corpus**.

Unlike simple word counts, TF-IDF **downweights common words** and **upweights rare but meaningful ones**.

📐 Formula:

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

Where:

- **TF (Term Frequency)** = Frequency of term *t* in document *d*

$$\text{TF}(t, d) = \frac{\text{Count of t in d}}{\text{Total terms in d}}$$

- **IDF (Inverse Document Frequency)** = Log measure of how rare term *t* is across the corpus

$$\text{IDF}(t) = \log\left(\frac{N}{1 + \text{DF}(t)}\right)$$

> `N` = Total number of documents
> `DF(t)` = Number of documents containing the term `t`

📘 Simple Example:

Corpus:

- D1: "NLP is fun"
- D2: "Learning NLP is cool"
- D3: "I love NLP and learning"

Vocabulary: [“NLP”, “learning”, “fun”, “cool”, “love”, “and”, “is”, “I”]

Term	DF (Docs it appears in)	IDF
NLP	3	log(3/4) ≈ 0.00 (common)
learning	2	log(3/3) = 0.00
fun	1	log(3/2) ≈ 0.18
cool	1	log(3/2) ≈ 0.18

→ So, **“fun”** and **“cool”** have **higher TF-IDF** than “NLP” or “learning”.

📊 Intuition:

- **TF** shows how frequent a term is in the doc
- **IDF** penalizes commonly seen words in the corpus
- Together, they highlight **important & unique terms**

🧠 Why Use TF-IDF?

- Improves over Bag-of-Words by **removing noise** from common words.
- Better for tasks like:
  - Text classification
  - Document similarity
  - Keyword extraction
  - Information retrieval

✅ Advantages:

- Simple to implement
- Effective for many real-world NLP problems
- Interpretable and intuitive

❌ Limitations:

- Still creates **sparse high-dimensional vectors**
- **No semantic meaning** or word relationships
- Sensitive to exact word forms (e.g., “run” ≠ “running”)

④ TF-IDF [Term Frequency - Inverse Document Frequency]

S1 → good boy

S2 → good girl

S3 → boy girl good

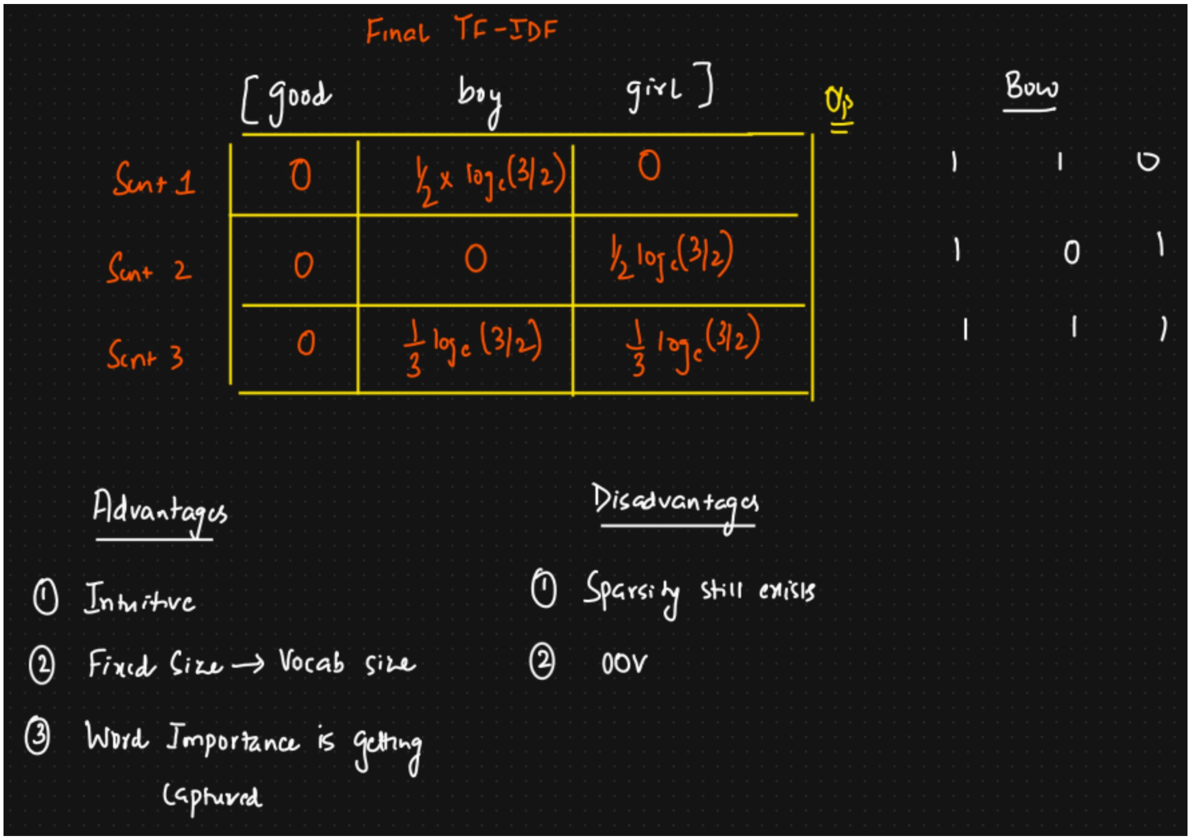
$$\text{Term Freq (TF)} = \frac{\text{No. of rep of words in sentence}}{\text{No. of words in sentence}}$$

$$\text{IDF} = \log_e\left(\frac{\text{No. of sentences}}{\text{No. of sentences containing the word}}\right)$$

Term Frequency * **IDF**

	S1	S2	S3
good	$1/2$	$1/2$	$1/3$
boy	$1/2$	0	$1/3$
girl	0	$1/2$	$1/3$

Words	IDF
good	$\log_e(3/3) = 0$
boy	$\log_e(3/2)$
girl	$\log_e(3/2)$



🧠 Word Embeddings

📌 Definition:

Word Embeddings are **dense vector representations** of words in a continuous vector space, where **semantically similar words** are placed closer together.

Unlike one-hot encoding or BoW, which are **sparse and context-agnostic**, embeddings capture **semantic relationships** between words.

🎯 Core Idea:

Words that appear in **similar contexts** have **similar meanings** → they should have **similar vectors**.

> "King" and "Queen" might differ in gender but are related in royalty → captured in vector relationships.

📊 Visual Representation:

Let's say each word is mapped to a **300-dimensional vector**:

Word	Vector (simplified view)
king	[0.25, 0.10, -0.12, ..., 0.05]
queen	[0.27, 0.11, -0.14, ..., 0.06]
banana	[-0.45, 0.90, 0.31, ..., -0.22]

🔍 Vectors of **king** and **queen** will be close
🛑 Vectors of **banana** and **queen** will be far apart

🧮 Famous Relationship:

A popular example:

$$\text{vector}(\text{"King"}) - \text{vector}(\text{"Man"}) + \text{vector}(\text{"Woman"}) \approx \text{vector}(\text{"Queen"})$$

This illustrates that **word embeddings can encode analogies and relationships**.

🧠 Why Use Word Embeddings?

Traditional Methods	Word Embeddings
Sparse	Dense
No context	Capture semantics
Equal distance	Semantic closeness

High dimensional	Lower dimensional

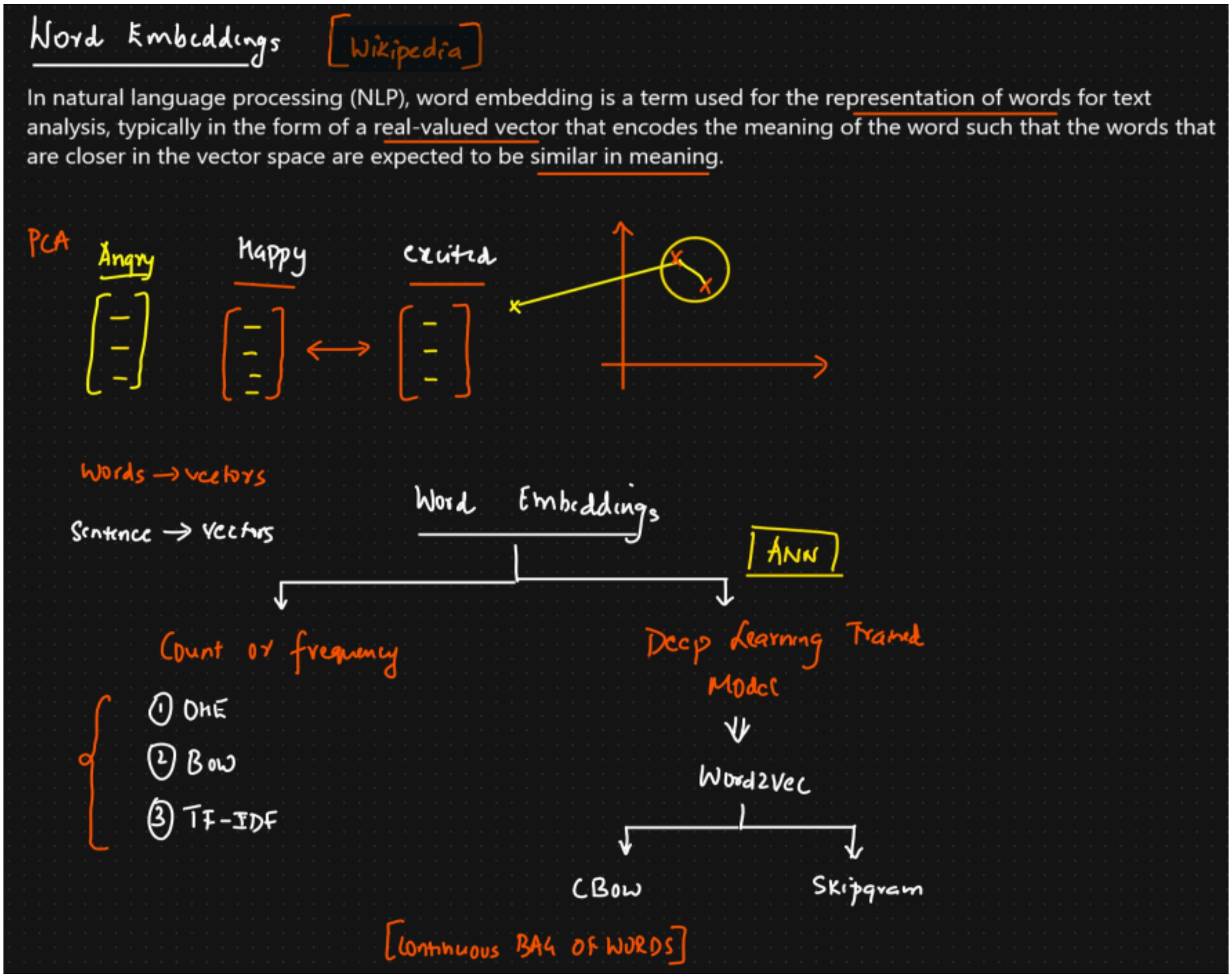
📘 Common Word Embedding Models:

Model	Description
Word2Vec	Predicts context words (Skip-Gram) or target word (CBOW)
GloVe	Global matrix factorization with local context
FastText	Adds subword (character n-gram) info for rare words
ELMo	Deep contextualized embeddings (from entire sentence)
BERT	Contextual, bidirectional embeddings using transformers

🔍 Word2Vec Overview:

Two architectures:

- **CBOW (Continuous Bag of Words)**: Predict word from context
- **Skip-Gram**: Predict context from word (better for rare words)



🧠 Word2Vec in NLP

📌 Definition:

Word2Vec is a shallow neural network model developed by **Google (2013)** that learns **dense vector representations** (embeddings) of words based on the context in which they appear.

> "You shall know a word by the company it keeps." – Firth (1957)
> Word2Vec applies this principle using surrounding words to learn a word's meaning.

🎯 Goal:

Map each word to a **low-dimensional continuous vector** such that **similar words have similar vectors**.

Neural Network Architecture: Word2Vec employs shallow, two-layer neural networks to learn these word embeddings. The network is trained to reconstruct the linguistic contexts of words. The learned weights from the hidden layer of this network form the word vectors.

How Word2Vec Works (Simplified):

1. **Input:** A large corpus of text.
2. **Process:**
   - The text is tokenized into words.
   - A vocabulary of unique words is created.
   - For each word, the model looks at its neighboring words within a defined "context window."
   - The neural network (either CBOW or Skip-gram) is trained. The goal is to adjust the weights (which become the word embeddings) to either predict a target word from context (CBOW) or predict context words from a target word (Skip-gram).
3. **Output:** A **mapping of each unique word** in the corpus to a vector in a high-dimensional space (typically several hundred dimensions).

Important Training Parameters:

- **Training Algorithm:**
  - **Hierarchical Softmax:** Uses a Huffman tree for efficient probability calculation, often better for infrequent words.
  - **Negative Sampling:** Modifies the objective by training the model to distinguish true context words from randomly sampled "negative" words, more effective for frequent words and lower-dimensional vectors.
- **Sub-sampling:** High-frequency words (like "the," "a") often provide less information. They can be subsampled to speed up training and improve representations of rarer words.
- **Dimensionality:** The size of the word vectors (embedding dimension). Higher dimensions can capture more information but require more data and computational power. Typical values range from 100 to 1,000.
- **Context Window:** The number of words to consider before and after a given target word as its context. Recommended values often differ for CBOW (e.g., 5) and Skip-gram (e.g., 10).

Word2Vec → Feature Representation

Google

Word2vec is a technique for natural language processing published in 2013. The word2vec algorithm uses a neural network model to learn word associations from a large corpus of text. Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence. As the name implies, word2vec represents each distinct word with a particular list of numbers called a vector.

Vocabulary → Unique words → Corpus.

Feature Representation

	Boy	Girl	KING	QUEEN	Apple	Mango
Gender	-1	1	-0.92	+0.93	0.01	0.05
Royal	0.01	0.02	0.95	0.96	-0.02	0.02
Age	0.03	0.02	0.75	0.68	0.95	0.96
Food	-	-	-	-	0.91	0.92
				-	-	-
300 dimension !	-	-	-		-	-
nth	-	-	-	-	-	-

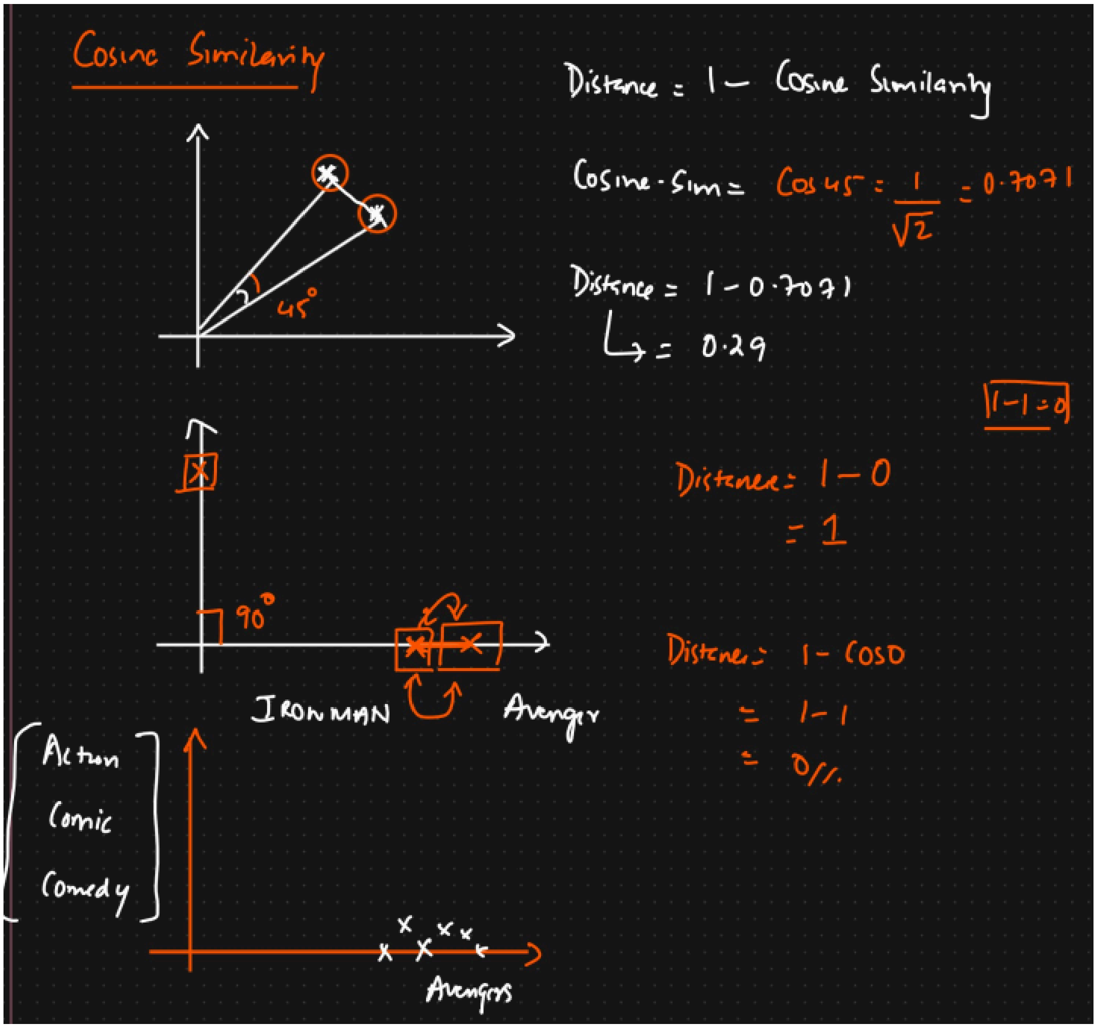
Above tells how much the word is related to the feature representation. Google uses 300 features.

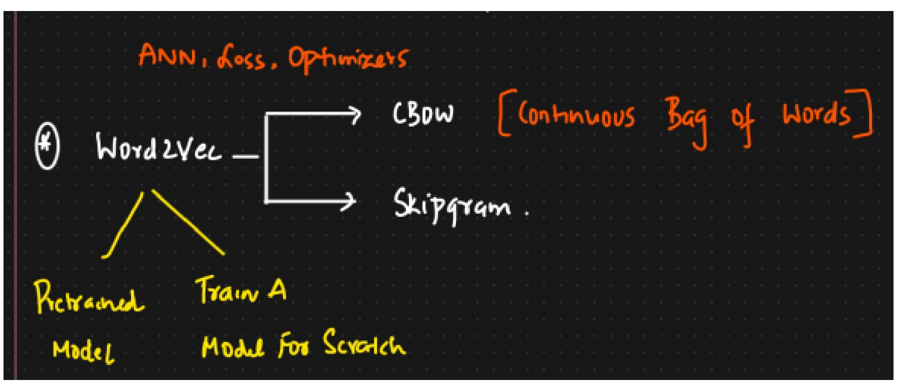
[KING - BOY + QUEEN = GIRL]

KING [0.95, 0.96] Man [0.95 0.98]

QUEEN [-0.96, 0.95] Women [-0.94, -0.96]

KING - MAN + QUEEN = WOMEN





🔧 Architectures of Word2Vec

1. CBOW (Continuous Bag of Words)

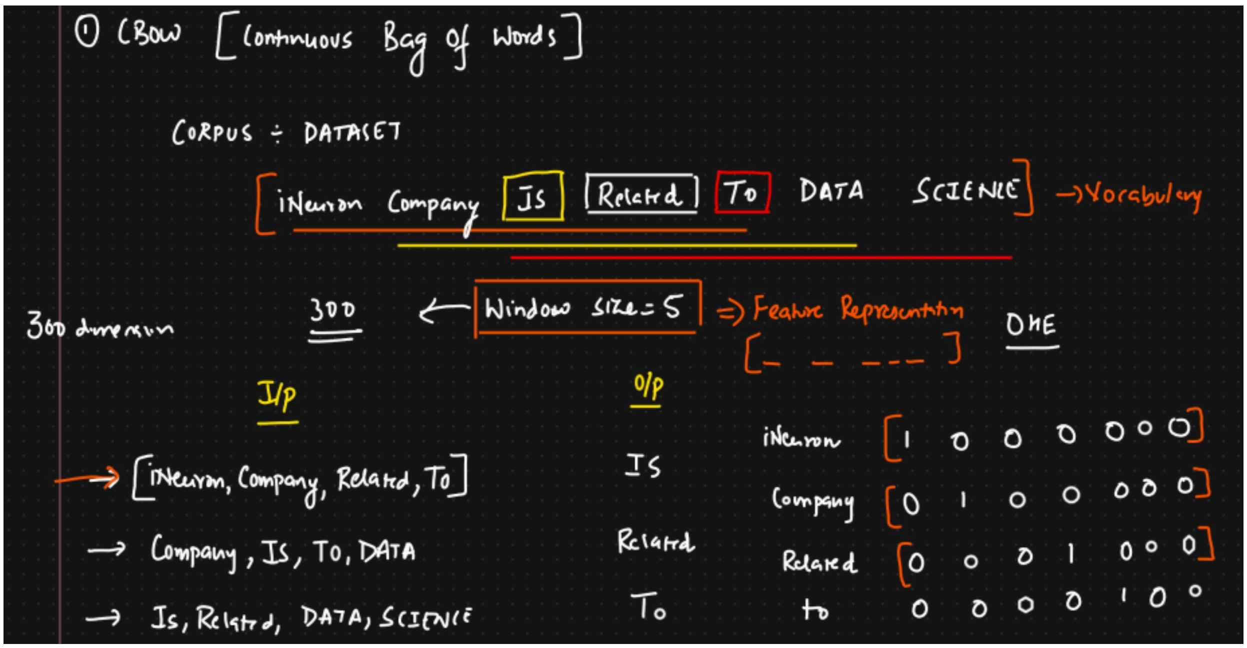
- **Predicts the target word** using surrounding context.

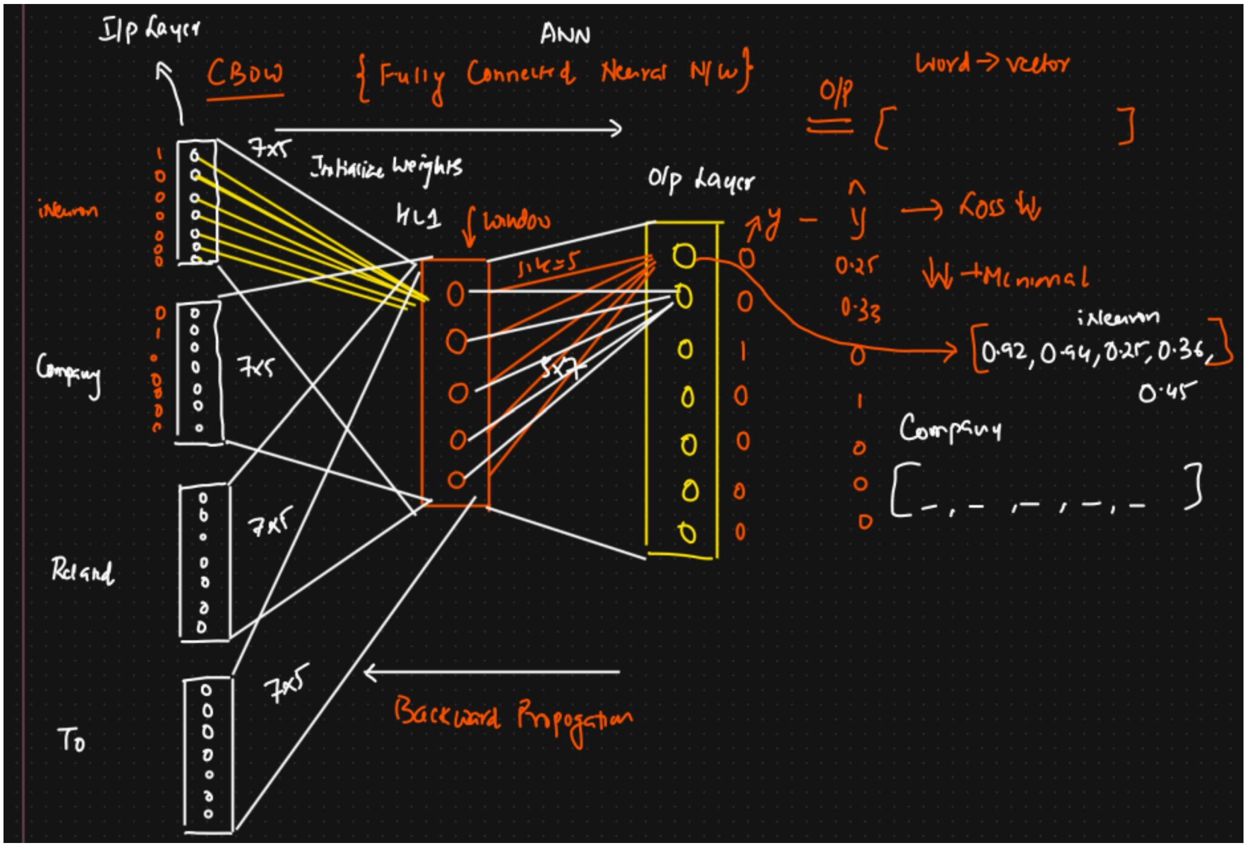
- Good for **frequent words**.

Example:

Sentence: "The cat sat on the mat"
Context (window=2): ["The", "cat", "on", "the"] → predict "sat"

Strengths: Generally faster to train and performs well with frequent words. It tends to smooth over distributional information.





Fully connected means every element or feature is connected to the next.

2. Skip-Gram

- **Predicts context words** given a target word.
- Better for **rare words**.

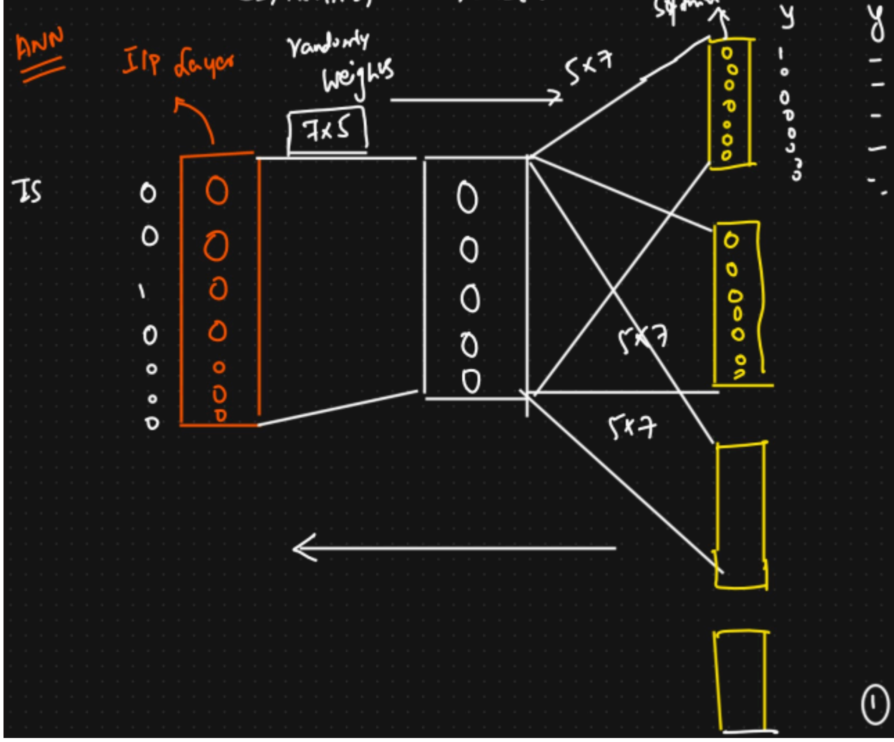
Example:

> Target = "sat" → Predict: ["cat", "on"]

② Skipgram – Word2Vec

Window size=5

O/p	I/p
→ [iNeuron, Company, Related, To]	Is
→ Company, Is, To, DATA	Related
→ Is, Related, DATA, SCIENCE	To

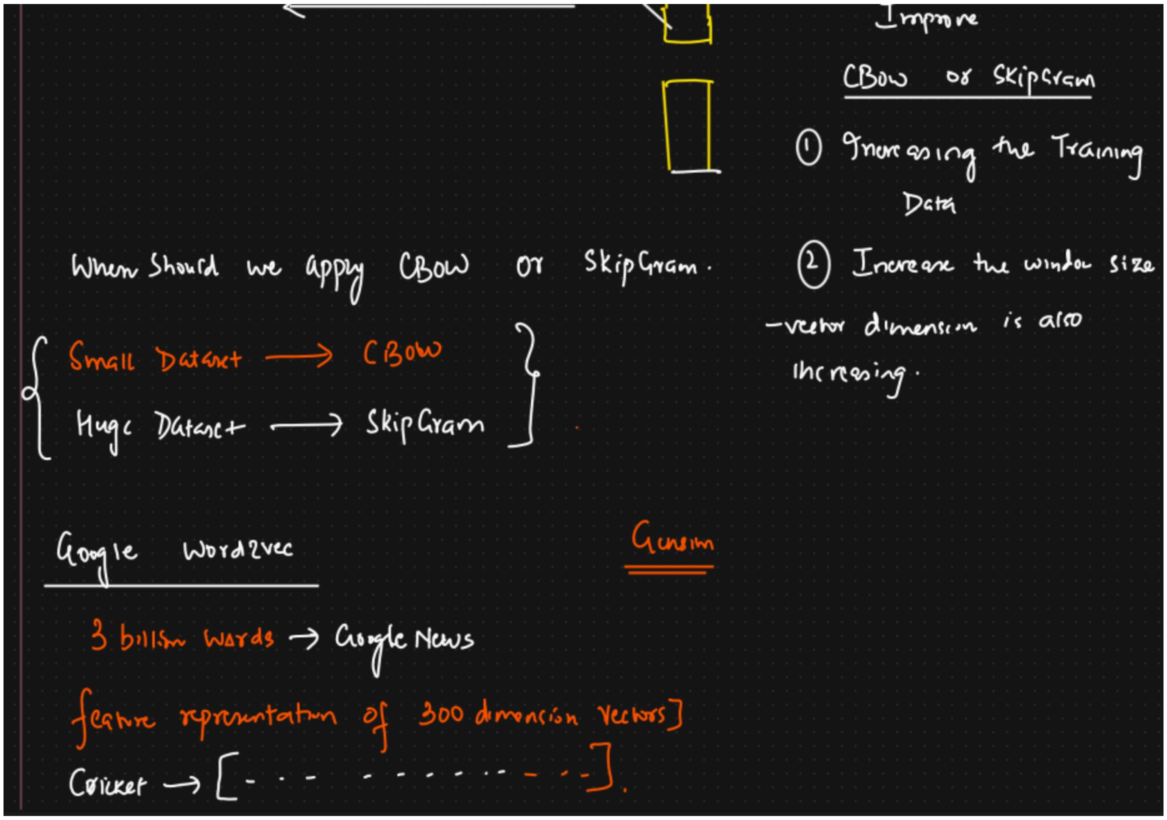


Loss function ↓

Improve

CBow or Skipgra

① Increasing the T



🧮 Training Objective:

CBOW: Maximize

$$P(\text{target}|\text{context})$$

Skip-Gram: Maximize

$$\sum_{\text{context word}} P(\text{context word}|\text{target})$$

Softmax-based Prediction:

$$P(w_O|w_I) = \frac{e^{v_{w_O} \cdot v_{w_I}}}{\sum_{w=1}^{V} e^{v_w \cdot v_{w_I}}}$$

⚙️ Optimization Techniques

Due to the **huge vocabulary**, full softmax is expensive. Word2Vec uses:

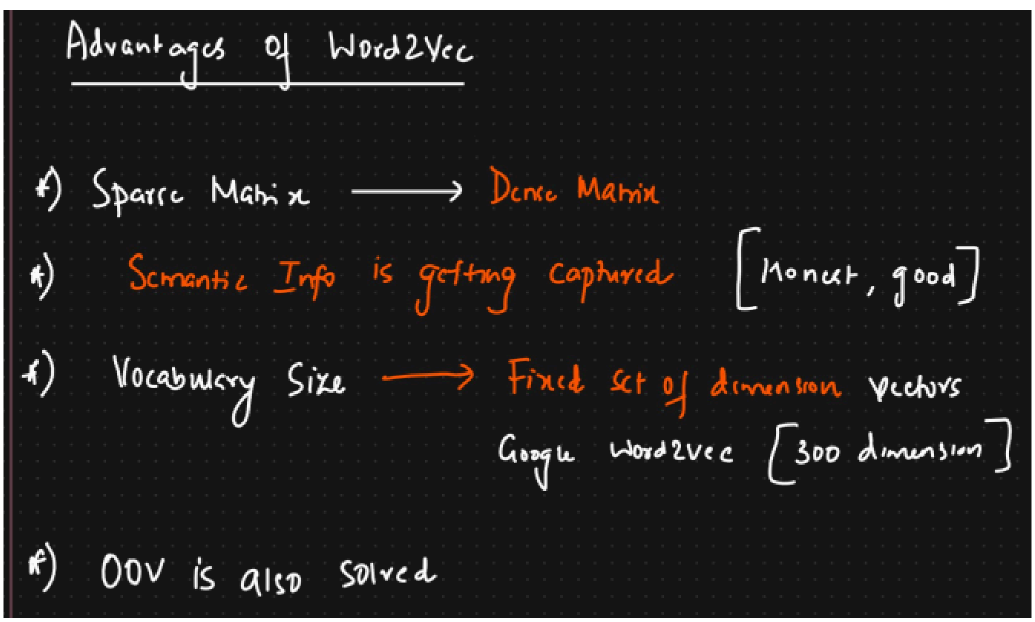
Technique	Description
Negative Sampling	Only update weights for a few “negative” samples
Hierarchical Softmax	Tree-based representation to reduce computation

📘 Example:

Sentence: `"The king loves his queen"`

After training, vectors learn relationships like:

- `king` ≈ `queen`
- `king` - `man` + `woman` ≈ `queen`



📌 What is Avg Word2Vec?

Average Word2Vec is a technique to generate **sentence or document vectors** by **averaging the Word2Vec embeddings of all words** in the sentence.

It’s a **baseline** method that works surprisingly well for many NLP tasks like classification, similarity, and clustering.

🎯 Why?

Word2Vec gives vectors for **individual words**, but ML models need fixed-size **input vectors** for:

- Sentences
- Documents

Avg Word2Vec solves this by computing:

$$\text{Vector(Sentence)} = \frac{1}{N} \sum_{i=1}^{N} \text{Word2Vec}(w_i)$$

Where:

- w_i = word in the sentence
- N = number of words (excluding stopwords, if filtered)

🔍 Example:

Sentence: `"NLP is fun"`

Let’s say:

- Word2Vec("NLP") = [0.2, 0.4, -0.1]
- Word2Vec("is") = [0.0, 0.1, 0.0]
- Word2Vec("fun") = [0.5, 0.3, -0.2]

Avg Vector =

$$\frac{1}{3}([0.2, 0.4, -0.1] + [0.0, 0.1, 0.0] + [0.5, 0.3, -0.2]) = [0.233, 0.267, -0.1]$$

🧠 Advantages

✅ Simple and fast
✅ Captures some semantic meaning
✅ Works with pretrained embeddings
✅ Great as a baseline for classification or clustering

❌ Limitations

🚫 Loses word order and syntax
🚫 Gives equal weight to all words (including stopwords unless removed)
🚫 Not context-aware (polysemy isn't handled)