Deep Learning
CSC-Elective

Instructor: Dr. Muhammad Ismail

Slides prepared by Dr. M Asif Khan

ismail@iba-suk.edu.pk

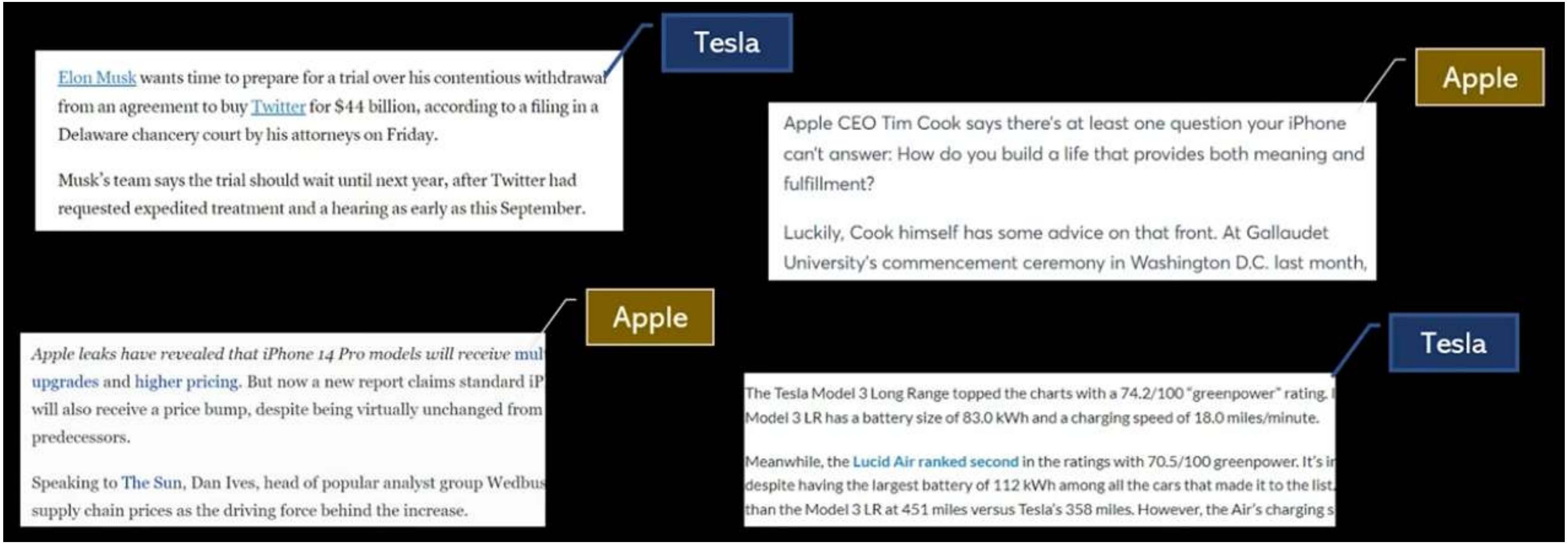
Unit 02 NLP Week 2

Contents

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

- TF-IDF is a statistical measure used to evaluate the importance of a word in a document relative to a collection of documents (corpus).
- Helps identify words that are important within specific documents, reducing the impact of common words across the entire dataset.
- Unlike simple word counts, TF-IDF accounts for the uniqueness of terms across documents, improving the performance of NLP models.

Text Classification



	musk	that	price	market	investor	iphone	itunes	gigafactory	...
Apple article 1	[0	32	45	48	26	7	3	0	...]
Apple article 2	[0	4	3	7	8	6	3	0	...]
Tesla article 3	[15	31	44	43	25	0	0	0	...]
Tesla article 4	[3	0	0	0	0	0	0	1	...]

Bage of Word

	musk	that	price	market	investor	iphone	itunes	gigafactory	...
Apple article 1	[0	0.05	0.01	0.05	0.05	0.9	0.8	0	...]
Apple article 2	[0	0.002	0.008	0.01	0.02	0.9	0.8	0	...]
Tesla article 3	[0.99	0.05	0.01	0.05	0.05	0	0	0	...]
Tesla article 4	[0.95	0	0	0	0	0	0	0.87	...]

TF-IDF

TF-IDF

- Term Frequency (TF):
  - Measures the frequency of a term in a document.
  - If a word is more present in in the sentence the it gets more importance.
  - Formula:

$$TF = \frac{No\ of\ rep\ of\ word\ in\ sentence}{No\ of\ words\ in\ sentence}$$

- Inverse Document Frequency (IDF):
  - Measures how important a term is across all documents.
  - If a word is present in all sentences then less importance is given to it.
  - Formula:

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

- TF-IDF Score:
  - Computed by multiplying TF and IDF.
  - Formula: $TF - IDF = TF \times IDF$

TF-IDF: Example

- **Document 1**: "apple banana apple"
- **Document 2**: "banana orange banana"
- **Document 3**: "apple orange"
- **Document 4**: "apple apple banana"
- **Document 5**: "banana banana orange"

Now, calculate TF IDF for the terms "apple," "banana," and "orange."

$$TF = \frac{No\ of\ rep\ of\ word\ in\ sentence}{No\ of\ words\ in\ sentence}$$

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

$$TF - IDF = TF \times IDF$$

TF-IDF

- How to calculate TF-IDF:
- Corpus: ["The cat sat on the mat", "The dog sat on the log"]
- Step 1: Compute TF:
  - For "sat" in Document 1: TF = 1/6
- Step 2: Compute IDF:
  - For "sat": Appears in 2 documents, so IDF = log(2/2) = 0
  - For "cat": Appears in 1 document, so IDF = log(2/1) = 0.301
- Step 3: Calculate TF-IDF:
  - Multiply TF by IDF for each term in each document.

TF-IDF

- S1 -> good girl
- S2 -> bad work boy
- S3 -> boy girl good
- Vocab: good, bad, boy, girl, work

Term Frequency

	S1	S2	S3
good	1/2 = 0.5	0/3 = 0	1/3 = 0.33
bad	0/2 = 0	1/3 = 0.33	0/3 = 0
boy	0/2 = 0	1/3 = 0.33	1/3 = 0.33
girl	½ = 0.5	0/3 = 0	1/3 = 0.33
work	0/2 = 0	1/3 = 0.33	0/3 = 0

$$TF = \frac{No\ of\ rep\ of\ word\ in\ sentence}{No\ of\ words\ in\ sentence}$$

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

Inverse Document Frequency

good	ln(3/2) = 0.405
bad	ln(3/1) = 1.099
boy	ln(3/2) = 0.405
girl	ln(3/2) = 0.405
work	ln(3/1) = 1.099

TF-IDF

	S1	S2	S3
good	0.5x0.405= 0.2	0x0.405 = 0	0.33x0.405= 0
Bad	0x1.099= 0	0.33x1.099=0.36	0x1.099= 0
Boy	0x0.405= 0	0.33x0.405=0.13	0.33x0.405=0.13
Girl	0.5x0.405= 0.20	0x0.405= 0	0.33x0.405=0.13
work	0x1.099= 0	0.33x1.099=0.36	0x1.099= 0

You are working for a startup that builds search engines. The startup has a small database of documents, and users want to find the most relevant documents based on their search queries. To implement this, you decide to use **TF-IDF** to rank documents by their relevance to a given query.

Corpus:

Document 1: "Data science is amazing."

Document 2: "Data science and machine learning."

Document 3: "Machine learning is a part of data science."

Search Query: "machine learning science"

Preprocessing: Perform tokenization and convert the corpus and query to lowercase.

TF Calculation:

- Compute the term frequency (TF) for each word in the query in **Document 1**.

IDF Calculation:

- Compute the IDF for the words "machine," "learning," and "science" across the corpus.

TF-IDF Score:

- Compute the TF-IDF score for the words "machine," "learning," and "science" in **Document 1**.

TF-IDF

- **Top Features by TF-IDF Score:** Prioritizes terms that are both frequent in individual documents and unique across the corpus.
- **Using max_df or min_df Parameters:** These parameters in Tfidf Vectorizer allow further control over feature selection:
- max_df can remove terms that appear in a high percentage of documents (too common).
- min_df ensures that only terms appearing in a minimum number of documents are included, excluding very rare words.
- Advantages
  - Intuitive
  - Fixed Size - > Vocab size
  - Word importance is being captured
- Disadvantages
  - Sparse matrix still exist

TF-IDF

```
[1] from sklearn.feature_extraction.text import TfidfVectorizer

    # Sample corpus of text documents
    corpus = [
        "The quick brown fox jumps over the lazy dog.",
        "Never jump over the lazy dog quickly.",
        "The fox was quick to jump."
    ]

    # Create the TF-IDF Vectorizer
    vectorizer = TfidfVectorizer()

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

    # Display the TF-IDF matrix
    print("TF-IDF Matrix:\n", X.toarray())

    # Display the feature names (words in the vocabulary)
    print("\nVocabulary:\n", vectorizer.get_feature_names_out())
```

```
TF-IDF Matrix:
 [[0.3988115  0.30330642 0.30330642 0.         0.3988115  0.30330642
  0.         0.30330642 0.30330642 0.         0.47108899 0.
  0.        ]
 [0.         0.35221512 0.         0.35221512 0.         0.35221512
  0.46312056 0.35221512 0.         0.46312056 0.27352646 0.
  0.        ]
 [0.         0.         0.37633075 0.37633075 0.         0.
  0.         0.         0.37633075 0.         0.2922544  0.49482971
  0.49482971]]

Vocabulary:
 ['brown' 'dog' 'fox' 'jump' 'jumps' 'lazy' 'never' 'over' 'quick'
 'quickly' 'the' 'to' 'was']
```

TF-IDF

Displaying TF-IDF Scores for Each Word in Each Document

```
[9] import pandas as pd

    # Convert TF-IDF matrix to a DataFrame
    tfidf_df = pd.DataFrame(X.toarray(), columns=vectorizer.get_feature_names_out())

    # Display the DataFrame
    print("TF-IDF Scores for Each Word in Each Document:\n", tfidf_df)
```

```
TF-IDF Scores for Each Word in Each Document:
       brown       dog       fox      jump     jumps      lazy     never  \
0  0.398811  0.303306  0.303306  0.000000  0.398811  0.303306  0.000000
1  0.000000  0.352215  0.000000  0.352215  0.000000  0.352215  0.463121
2  0.000000  0.000000  0.376331  0.376331  0.000000  0.000000  0.000000

       over     quick   quickly       the       to      was
0  0.303306  0.303306  0.000000  0.471089  0.00000  0.00000
1  0.352215  0.000000  0.463121  0.273526  0.00000  0.00000
2  0.000000  0.376331  0.000000  0.292254  0.49483  0.49483
```

Word Embeddings

- Word embedding is a term used for representation of words for text analysis, typically in the **form of a real-valued vector**.
- That **real-valued vector encodes the meaning of the word** such that the words that are **closer in the vector space** are expected to be similar in the meaning.

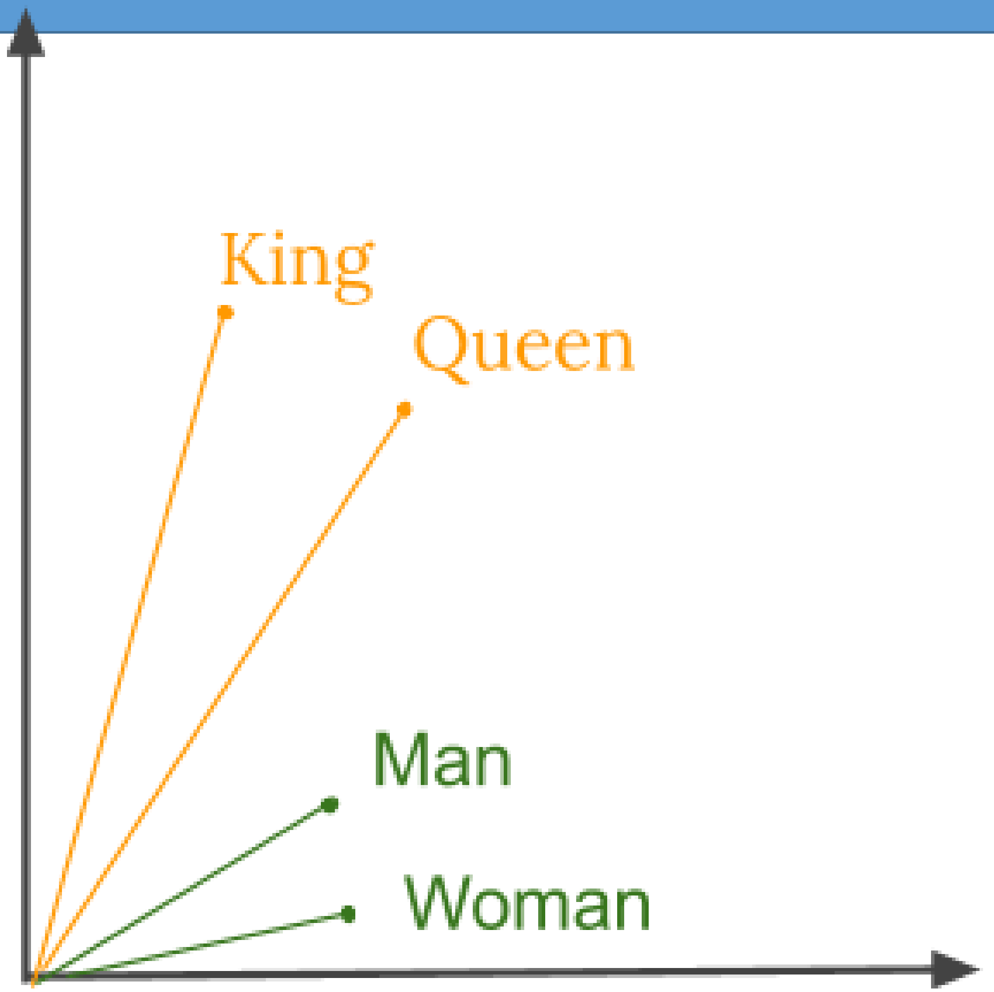
- **Capture meaning:** Words with similar meaning → similar vectors
- **Handle context:** Words are closer in vector space if they appear in similar contexts
- **Efficient:** Reduces the number of features compared to one-hot encoding
- **Improves model performance:** Helps neural networks and ML models understand text better

Word Embedding

- Word Ebmeddings learned representations of text in an n-dimensional space

- words that have the same meaning have a similar representation.

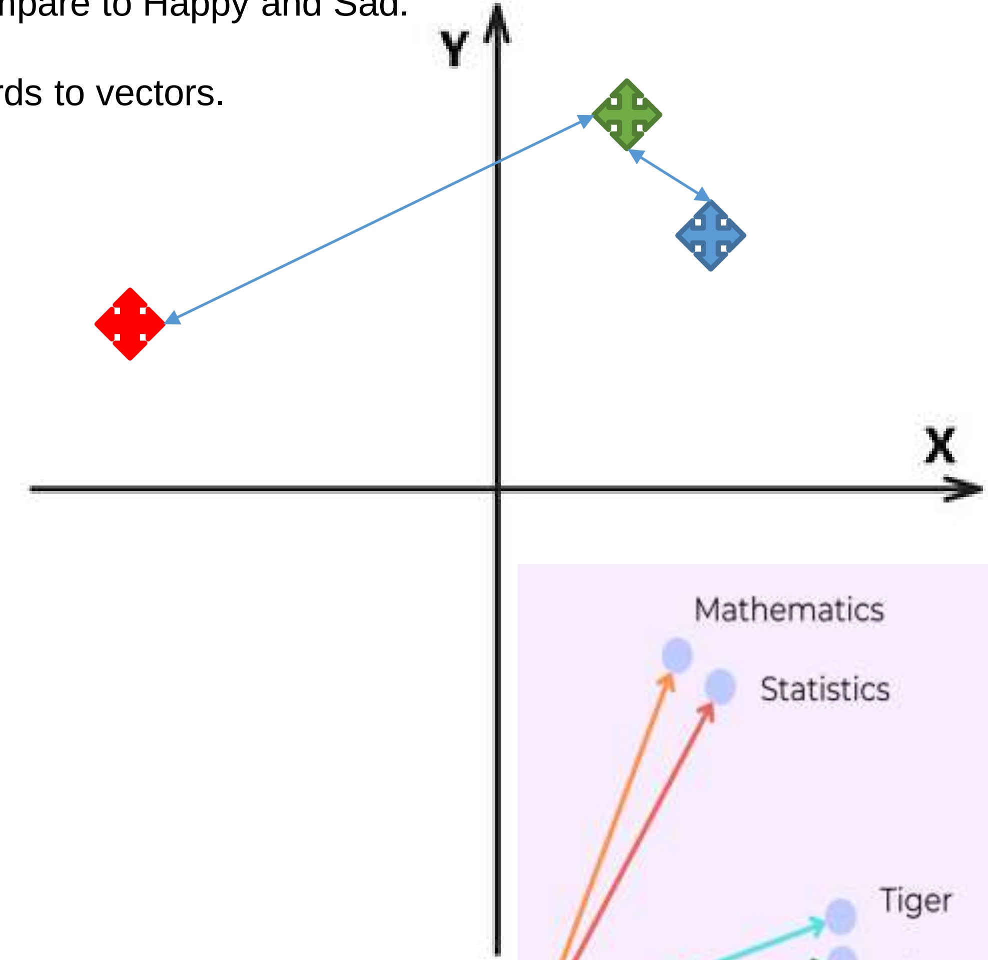
Word	One-Hot Encoding (simplified)	Word Embedding (example)
king	[0, 0, 0, 1, 0, 0]	[0.82, 0.51, -0.10, 0.33]
queen	[0, 0, 1, 0, 0, 0]	[0.79, 0.48, -0.12, 0.36]
apple	[1, 0, 0, 0, 0, 0]	[0.12, 0.93, -0.45, 0.05]

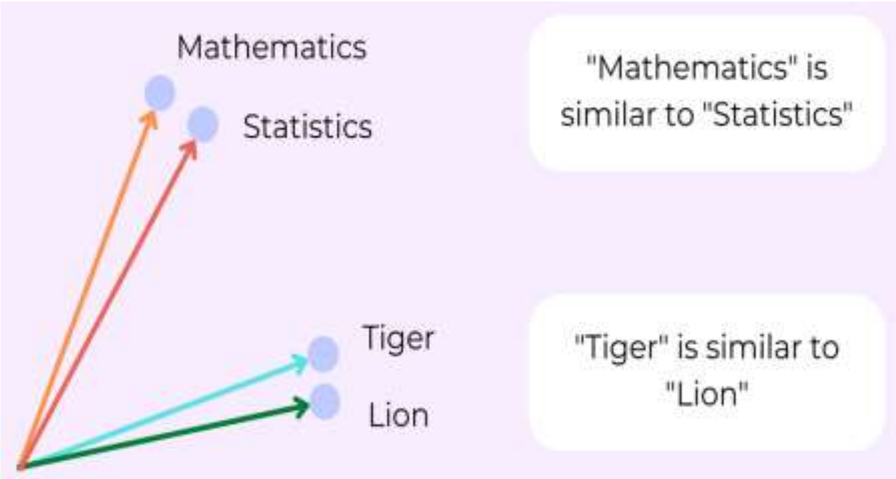


Word Embeddings

- If **Happy** (), **Thrilled** () and **Sad** () are encoded with word embeddings then their distance should look like following graph.
- **Happy** and **Thrilled should be closer** in vector space compare to Happy and Sad.
- This all will be possible due to **efficient conversion** of words to vectors.

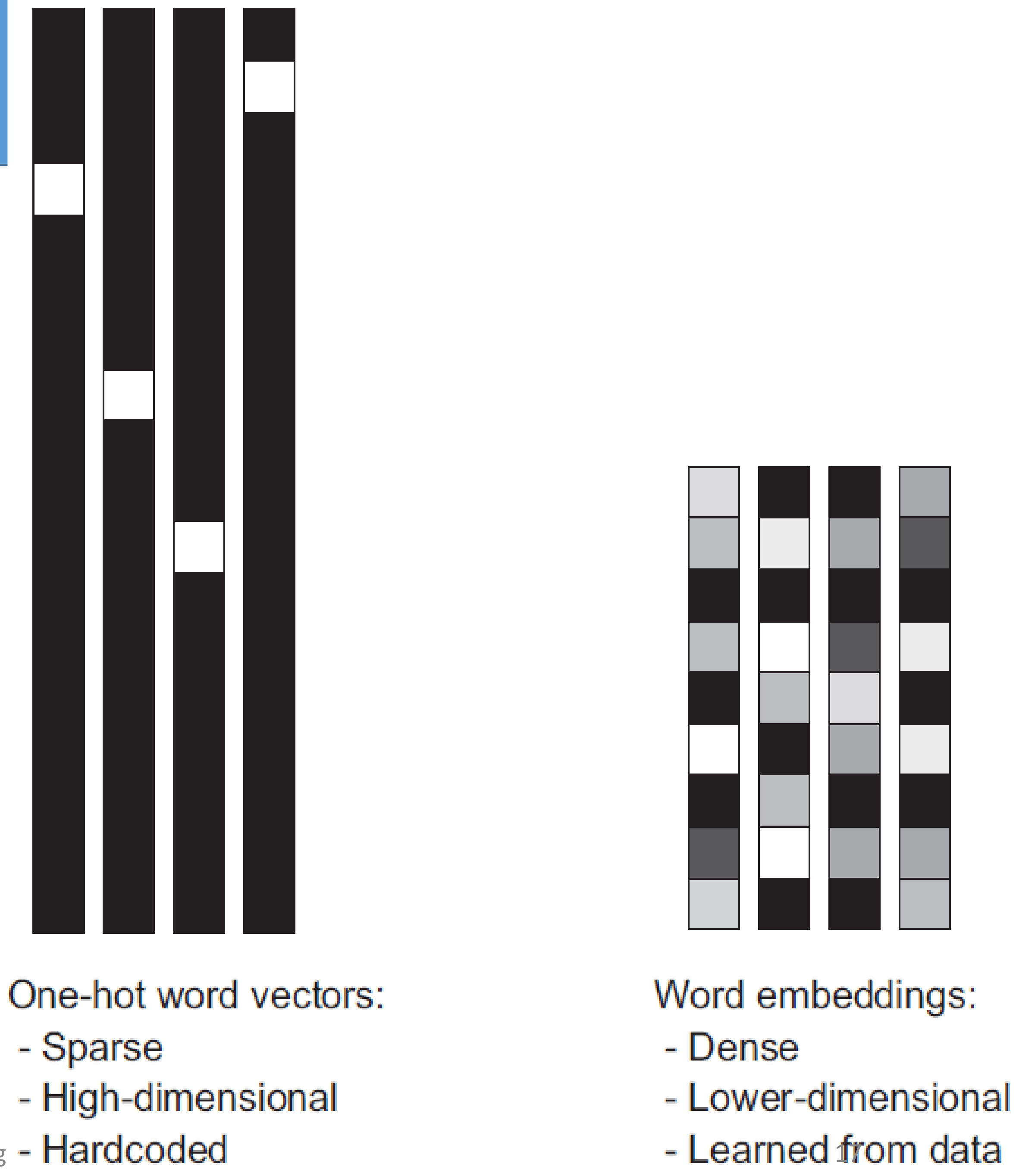
$$\begin{array}{ccc} \text{Sad} & \text{Happy} & \text{Thrilled} \\ \begin{bmatrix} 0.25 \\ 1 \\ 0.18 \\ 0.24 \\ 0.28 \\ 0.05 \\ . \\ . \\ . \\ 0.63 \end{bmatrix} & \begin{bmatrix} 1 \\ 0.5 \\ 0.8 \\ 0.02 \\ 0.8 \\ 0.95 \\ . \\ . \\ . \\ 0.26 \end{bmatrix} & \begin{bmatrix} 0.9 \\ 0.75 \\ 0.28 \\ 0.2 \\ 0.7 \\ 0.5 \\ . \\ . \\ . \\ 0.32 \end{bmatrix} \end{array}$$





Using Word Embedding

- One-hot-encoding
  - Binary
  - Sparse
  - High dim (same as num of unique words)
- Word Embedding
  - Low dimensional (256, 512, 1024 dim)
  - Floating point
  - Learned from data



Concept of Cosine Similarity

Cosine similarity measures **how similar two vectors are in direction**, **not magnitude**.
It tells us **how close in meaning** two words are in the embedding space.
Formula reminder:

$$\text{Cosine } \mathit{Similarity} = \frac{A \cdot B}{|| A || \times || B ||}$$

•Range: **–1 to +1**

- **+1 → perfectly similar (same direction)**
- **0 → unrelated (orthogonal)**
- **–1 → completely opposite meanings**

Applications in NLP

- **Document similarity** (e.g., finding similar news articles)
- **Semantic search** (retrieving related results)
- **Chatbot response matching**
- **Plagiarism detection**
- **Duplicate question detection** (e.g., Quora)

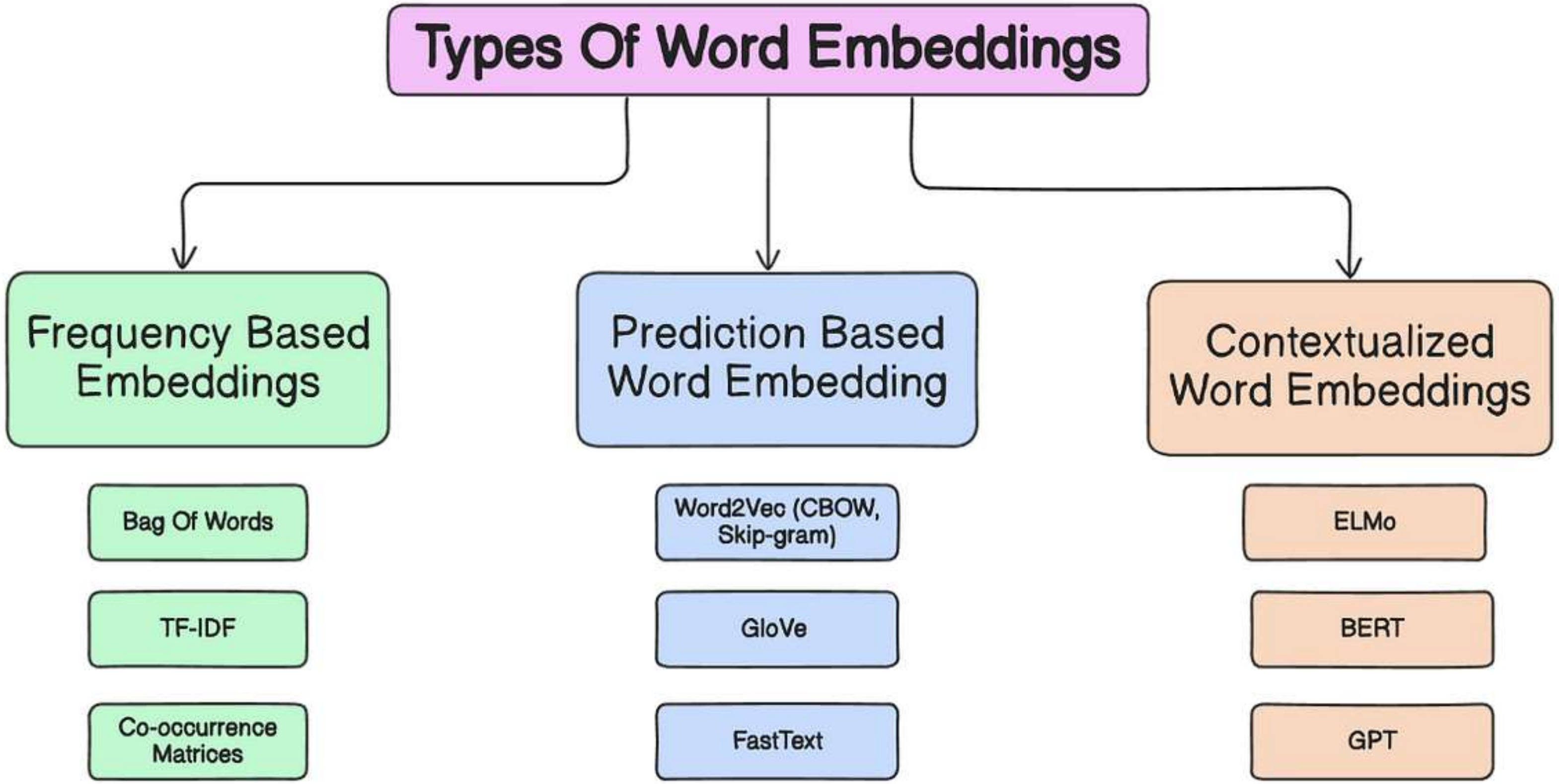
Concept of Cosine Similarity

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Sample sentences
sentence1 = "I love machine learning and Artificial Intelligence"
sentence2 = "I like artificial intelligence"

# Convert text to TF-IDF vectors
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform([sentence1, sentence2])

# Compute cosine similarity
similarity = cosine_similarity(tfidf_matrix[0:1], tfidf_matrix[1:2])

print("Cosine Similarity:", similarity[0][0])
```

```
... Cosine Similarity: 0.3187840217537793
```

Types of Word Embeddings

- Many of the disadvantages are overcome by deep learning trained models.



Word2Vec

- Word2Vec is a popular **algorithm for creating word embeddings**.
- Developed by a team led by **Tomas Mikolov** at **Google** in 2013.
- Converts words into **dense vector representations** in a continuous vector space.
- Uses a **neural network** model to learn word associations from a larger corpus of text.
- Once the model is trained it can **detect synonymous** words or **suggest additional words** for a partial sentence.
- It represents each distinct word with a particular **list of numbers called** vector.
- Types are
  - **Skip-Gram Model**: Predicts context words given a target word. Good for infrequent words.
  - **Continuous Bag of Words (CBOW)**: Predicts a target word based on surrounding context words. Efficient for frequent words.

Word2Vec

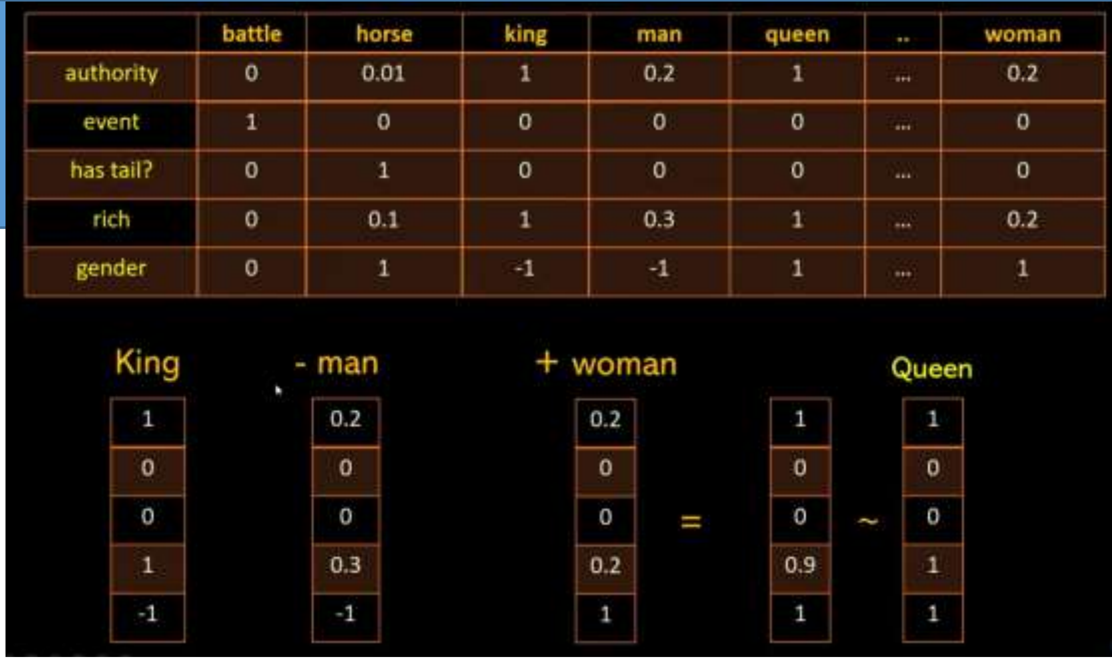
- Benefits:
  - Captures **semantic** relationships (e.g., "king" - "man" + "woman" ≈ "queen").
  - **Fast** and **scalable**, suitable for large datasets.
- Advantages:
  - **Semantic Similarity**: Enables better understanding of context and meaning.
  - **Dimensionality Reduction**: Efficiently represents words in lower dimensions compared to traditional methods.
  - **Flexibility**: Can be trained on specific datasets, tailoring embeddings to domain-specific language.

Word2Vec

- Corpus -> Unique words -> Vocabulary
- All these values (-1, +1. 0.01, 0.02) comes from DNN trained model.
- Because of these vectors similar words will be closed to each other.

Feature Representation	Boy	Girl	King	Queen	Apple	Mango	Organ
Gender	-1	+1	-0.92	+0.94	0.02	0.01	0.2
Royal	0.01	0.02	0.96	0.94	0.05	0.08	0.01
Age	0.03	0.01	0.78	0.69	0.56	0.68	0.45
Food	0.001	0.02	0.02	**0.01**	0.98	0.9	0.91
.	.	.	.	**.**	.	.	.
.	.	.	.	**.**	.	.	.
300th.	.	.	.	**.**	.	.	.

Word2Vec



	battle	horse	king	man	queen	..	woman
authority	0	0.01	1	0.2	1	...	0.2
event	1	0	0	0	0	...	0
has tail?	0	1	0	0	0	...	0
rich	0	0.1	1	0.3	1	...	0.2
gender	0	1	-1	-1	1	...	1

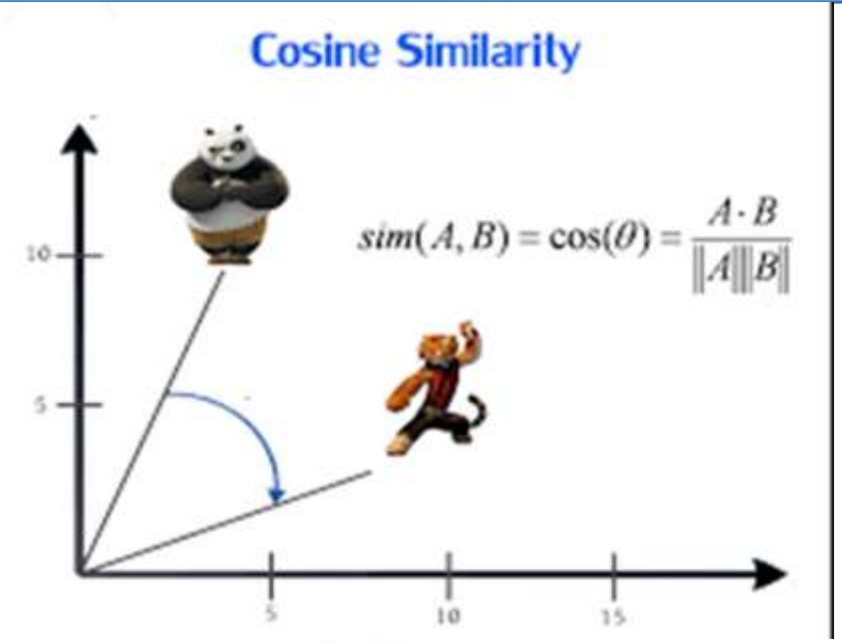
- **[King – Boy + Queen] ≈ Girl**
- King features are subtracted from Queen’s
- King’s gender features are subtracted by boy
- The remaining features are gender features of Queen
- Google wrod2vec is trained on 3B words

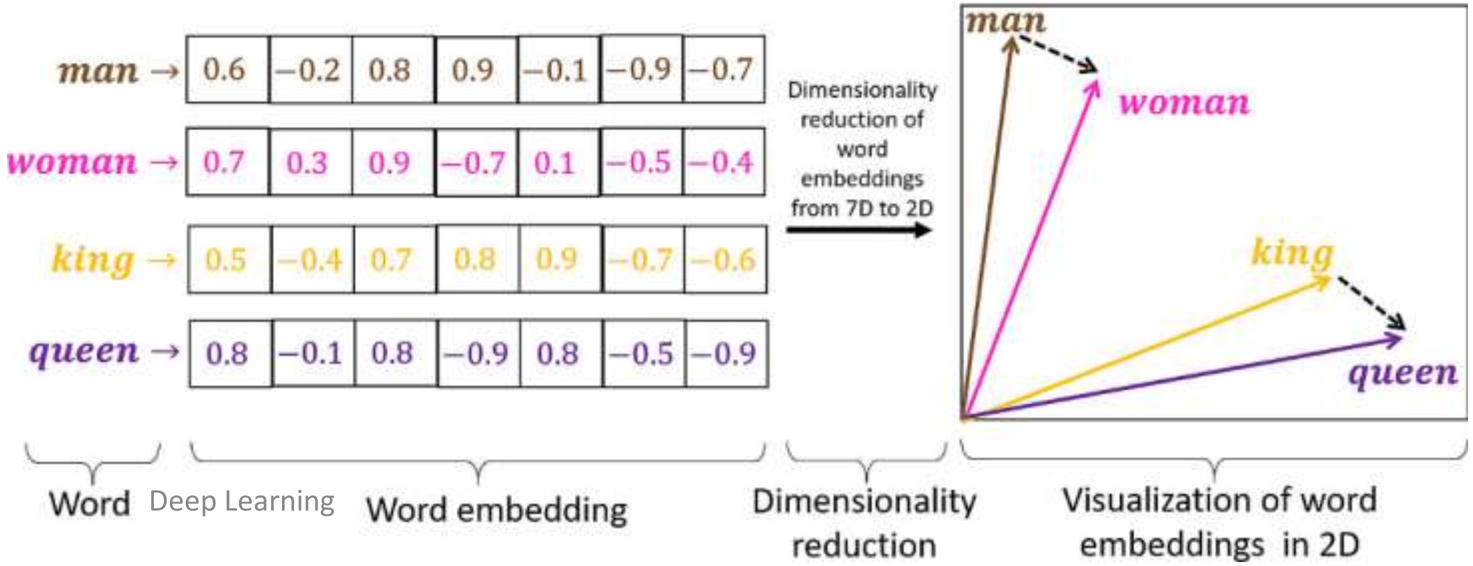
Feature Representation	Boy	Girl	King	Queen	Apple	Mango	Organ
Gender	-1	+1	-0.92	+0.94	0.02	0.01	0.2
Royal	0.01	0.02	0.96	0.94	0.05	0.08	0.01
Age	0.03	0.01	0.78	0.69	0.56	0.68	0.45
Food	0.001	0.02	0.02	**0.01**	0.98	0.9	0.91
.	.	.	.	**.**	.	.	.
.	.	.	.	**.**	.	.	.
300th.	.	.	.	**.**	.	.	.

How does it work

In simple words Word2vec is just vector representation of words in n dimension space. It is also called embedding.

Now why we use cosine similarity - **To get similarity between two words.**
How does it work Cosine similarity = 1 - cosine distance.
Cosine distance is nothing but getting distance between two vectors in n dimension space. Distance represent how words are related to each other.

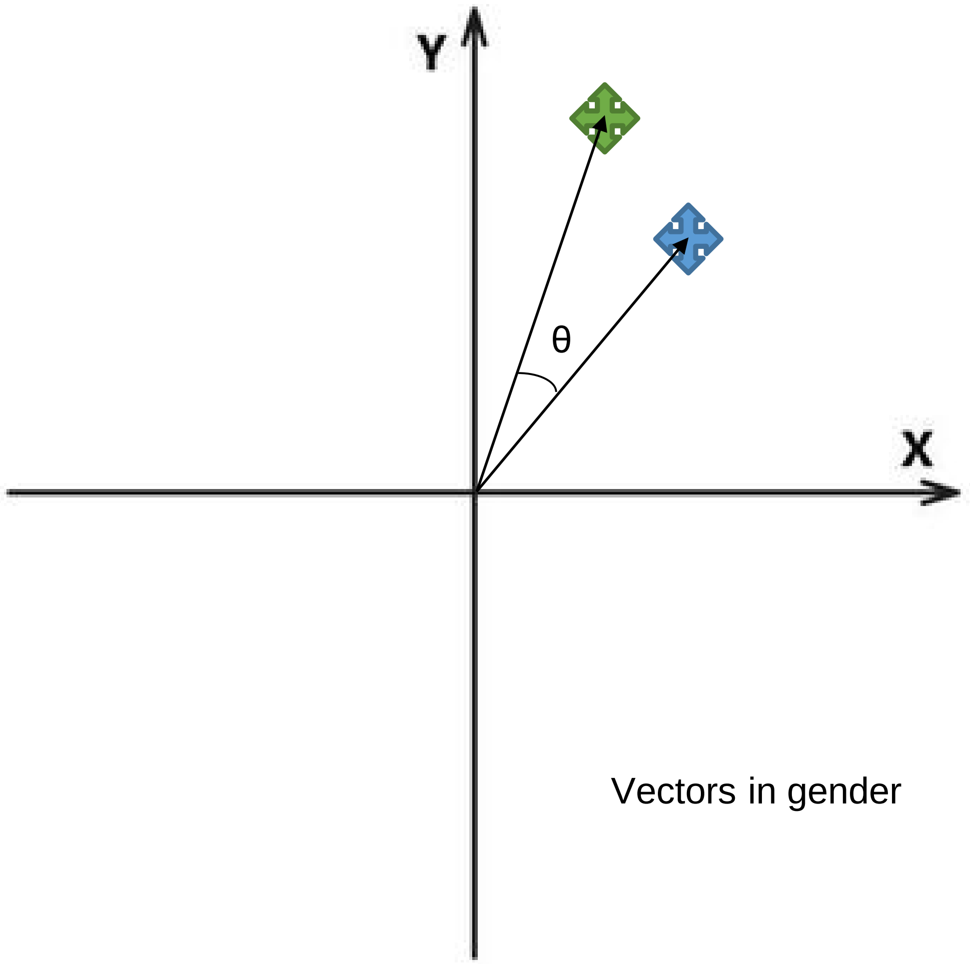




Word2Vec

- Assume following are vector representation after **PCA**:
- King = [0.95, 0.96]
- Man = [0.95, 0.98]
- Queen = [-0.96, 0.95]
- Women = [-0.94, -0.96]
- King – Man + Queen = Women

- **Cosine distance/similarity metric** = 1 - cos(angle θ b/w two vectors)
- What is distance if angle is 90 degree
- What is distance if angle is 180 degree
- What is distance if angle is 0 degree



Vectors in gender

Word2Vec

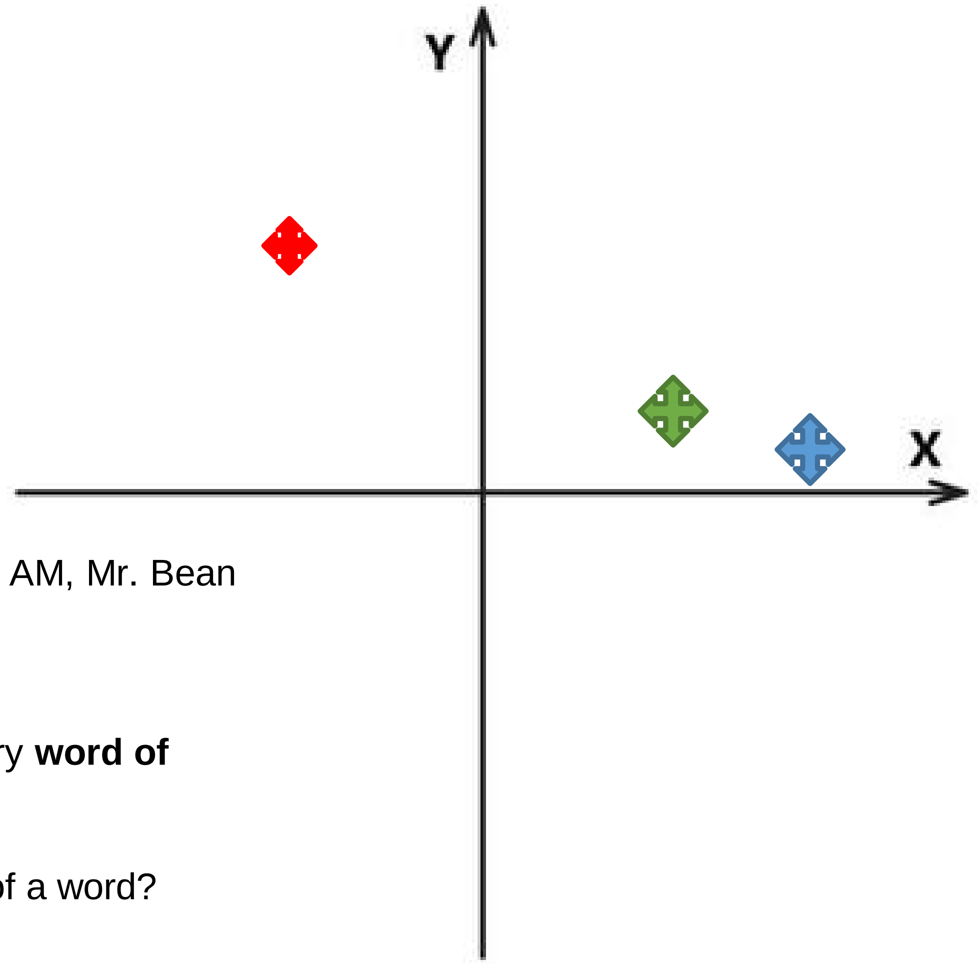
- Assume following are vector representation after PCA:
- King = [0.95, 0.96]
- Man = [0.95, 0.98]
- Queen = [-0.96, 0.95]
- Women = [-0.94, -0.96]
- King – Man + Queen = Womenθ

- Cosine distance/similarity metric = 1 - Cos(angle b/w two vectors)
- What is distance if angle is 90 degree
- What is distance if angle is 180 degree
- What is distance if angle is 0 degree

- Let take **scenario of movies**
- The movies are Avengers , Ironman , Hangover , Titanic, Bruce AM, Mr. Bean
- What are the vocabulary?
- What could be feature representations?

- At the end of word2vec we are creating **a feature representation** of every **word of vocabulary**.

- Now we need to know how feature representation is created and vector of a word?



Continuous Bag of Words (CBoW)

- A model used for generating word embeddings in NLP.
- **Part of the Word2Vec** framework developed by Google.
- Predicts a target word **based on its surrounding context words**.
- Key Characteristics:
  - **Contextual Focus**: Takes a set of context words (surrounding words) as input to predict the target word.
  - **Word Representation**: Each word is represented as a vector, enabling semantic understanding and relationships.
- Training Objective:
  - Minimize the loss function that measures the difference between the predicted word and the actual target word.
  - word2Vec also have **pretrained model** e.g., a model from Google trained on 3 Billion words, or you can train a model from scratch.

Continuous Bag of Words (CBoW)

- How Does CBOW Work?
  - **Input:** A set of context words around the target word.
  - **Output**: The target word predicted based on the context.
  - How CBOW Works in Practice:
  - **Example Sentence**: "The cat sat on the mat.“
    - Context (Window size = 2): "The", "cat", "on", "the“
    - Target word: "sat“
    - CBOW learns to predict the target word "sat" using the surrounding context words ("The", "cat", "on", "the").

CBoW

Corpus = [iCreative company $\boxed{\text{is}}$ $\boxed{\text{related}}$ $\boxed{\text{to}}$ SEO optimization]

Window size = 5

I/P	O/P
iCreative, company, related, to	is
Company, is, to, SEO	Related
Related, to, SEO, optimization	to

Vocabulary is = [iCreative, company, is, related, to, SEO, optimization]
One hot encoding of following words:
iCreative = [1, 0, 0, 0, 0, 0, 0]
company = [0, 1, 0, 0, 0, 0, 0]
related = [0, 0, 1, 0, 0, 0, 0]
to = [0, 0, 0, 1, 0, 0, 0]
It mean if I want to send iCreative, company or other word then I have to send its above one hot encoded vector.

CBoW

Corpus = [iCreative company is related to SEO optimization]

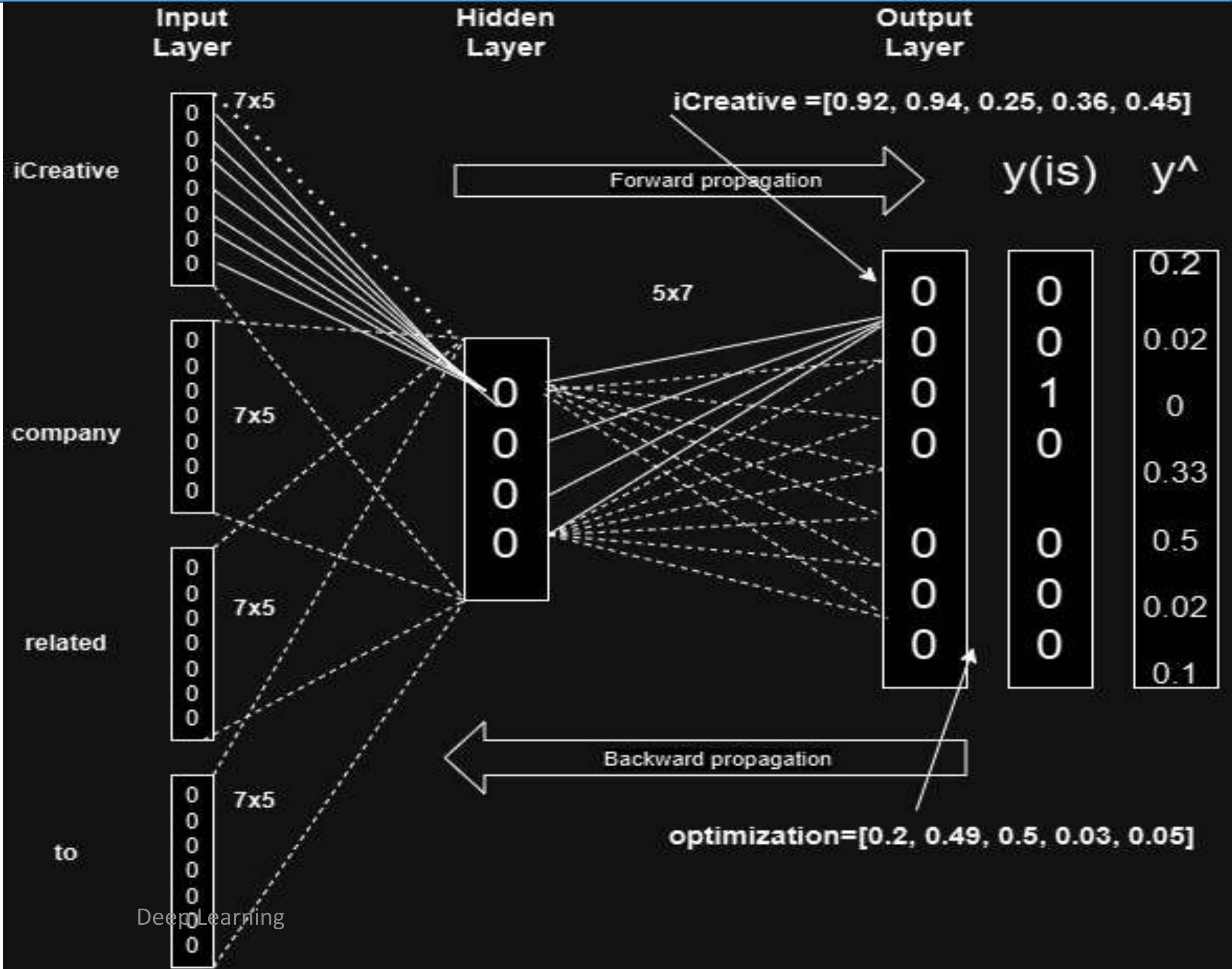
Window size = 5

I/P	O/P
iCreative, company, related, to	is
Company, is, to, SEO	Related
Related, to, SEO, optimization	to

Vocabulary is = [iCreative, company, is, related, to, SEO, optimization]
One hot encoding of following words:

iCreative = [1, 0, 0, 0, 0, 0, 0]
company = [0, 1, 0, 0, 0, 0, 0]
related = [0, 0, 1, 0, 0, 0, 0]
to = [0, 0, 0, 1, 0, 0, 0]

It mean if I want to send iCreative, company or other word then
I have to send its above one hot encoded vector.

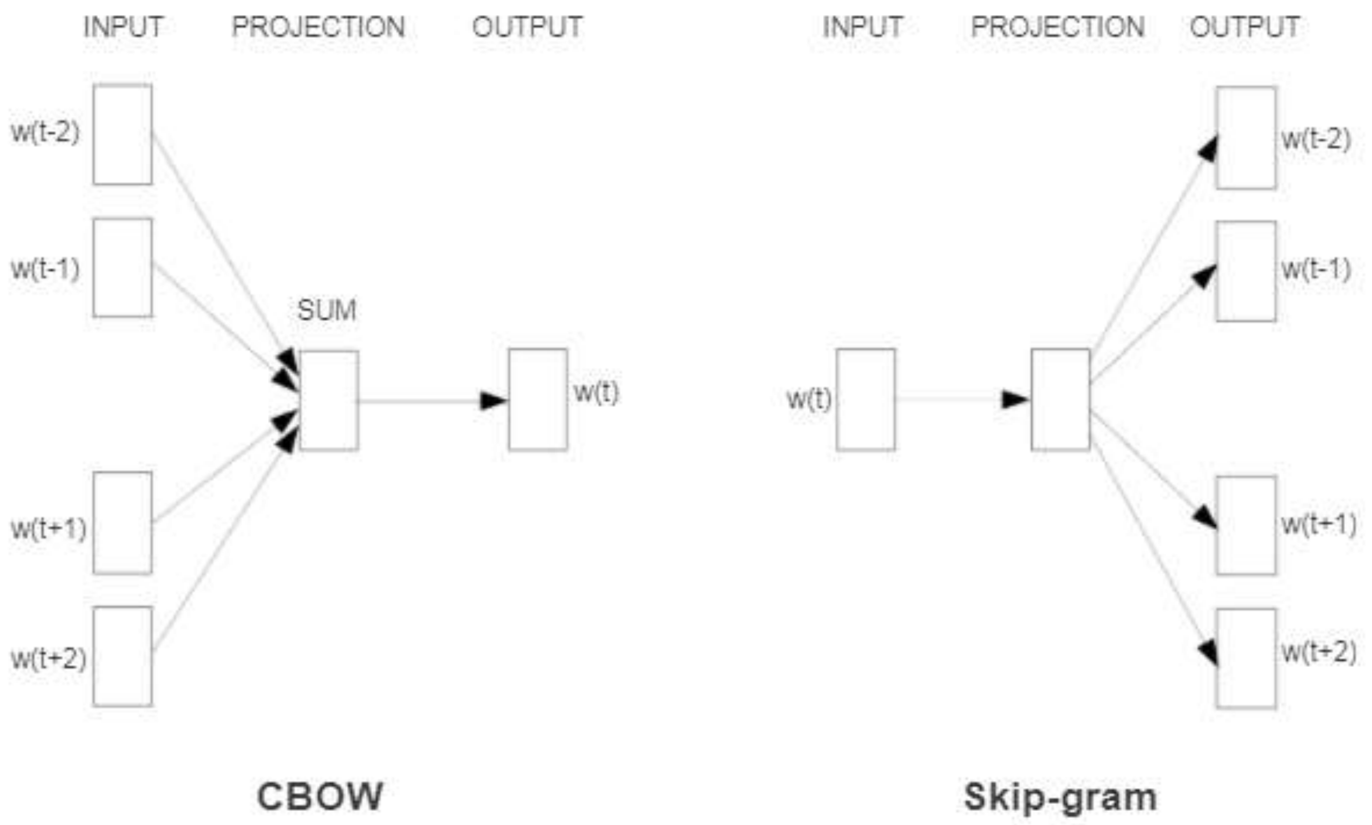


Skip-Gram

- The Skip-Gram model, **part of the Word2Vec framework**, works by **predicting context words from a target word**. While CBOW predicts the **target word from context words**.
- Skip-Gram **works in reverse**: it takes a single target word and tries to predict the surrounding context words.
- Key Functionality of Skip-Gram:
  - **Goal**: Given a target word, Skip-Gram tries to predict the words that appear around it (the context words).
  - **Training Objective**: Learn vector representations of words such that the target word's vector can predict the surrounding words' vectors with high accuracy.

Skip-Gram

- There are two architectures used by Word2vec:



Skip-Gram

- How Skip-Gram Works:
  - **Input**: The model takes a single target word as input.
  - **Output**: It predicts the context words within a fixed-size window around the target word.
  - **Context Window**: The window size determines how many surrounding words are considered as context. For example, in the sentence "The cat sat on the mat," if "sat" is the target word and the window size is 2, the context words would be "The," "cat," "on," and "the."

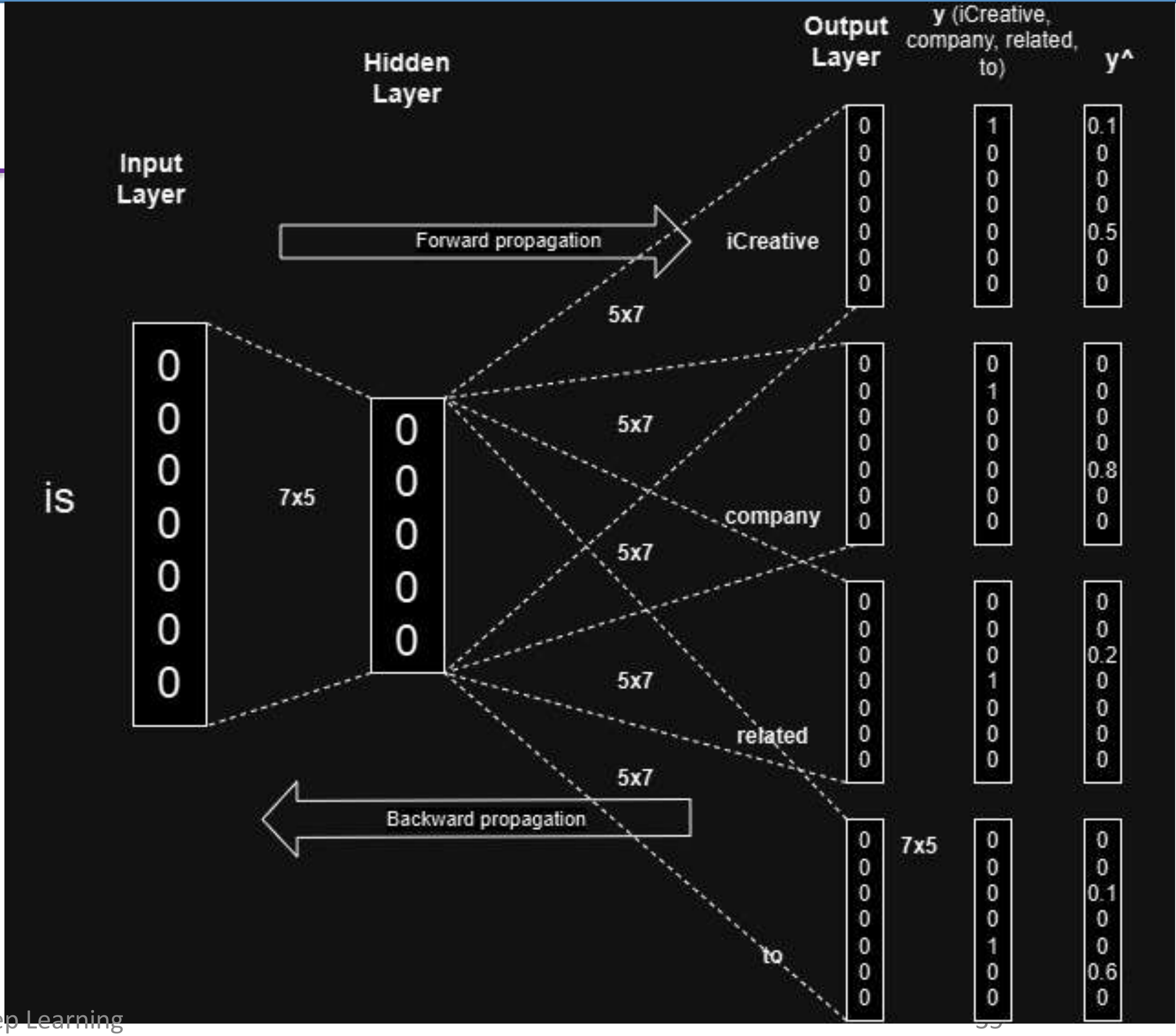
Skip-Gram

Corpus = [iCreative company is related to SEO optimization]

Window size = 5

I/P	O/P
is	iCreative, company, related, to
related	Company, is, to, SEO
to	Related, to, SEO, optimization

Vocabulary is = [iCreative, company, is, related, to, SEO, optimization]
One hot encoding of following words:
iCreative = [1, 0, 0, 0, 0, 0, 0]
company = [0, 1, 0, 0, 0, 0, 0]
related = [0, 0, 1, 0, 0, 0, 0]
to = [0, 0, 0, 1, 0, 0, 0]
It mean if I want to send iCreative, company or other word then
I have to send its above one hot encoded vector.



Skip-Gram Example

```python
from gensim.models import Word2Vec

# Sample corpus
sentences = [["king", "queen", "man", "woman", "prince", "princess"]]

# Train Word2Vec model
# sg=0 – use CBOW architecture (Continuous Bag of Words)
model = Word2Vec(sentences, vector_size=50, window=2, min_count=1, sg=0)

# Get vector for 'king'
print(model.wv["king"])

# Find similar words
print(model.wv.most_similar("king"))
```

```python
model.wv.most_similar("king")
```

```
[('queen', 0.10232102125883102),
 ('prince', 0.018277142196893692),
 ('princess', 0.012442175298929214),
 ('woman', -0.027841335162520441),
 ('man', -0.2091004103422165)]
```

Skip-gram Vs CBoW

- **Small dataset** -> CBoW (As it takes a set of context words and tries to predict a single word)
- **Large dataset** -> Skipgram (As it tries to predict the surrounding context words given a target word.)
- How to **improve the accuracy** for both
  - Increase the dataset size
  - Increase the window size or increasing vector dimension
- Google word2vec is trained on 3 billion words
- Feature representation of 300 dimension
- E.g., a word "**boy**" in Google model = [1st feature, ………, 300th feature]

Skip-gram vs CBOW

- Both Skip-gram and CBOW (Continuous Bag of Words) are Word2Vec architectures
  - Used to learn word embeddings from text.
- **CBOW:**
  - Predicts the target word from the context words.
- **Skip-gram:**
  - Predicts the context words from the target word.

Model	Input	Output
CBOW	Context words	Target word
Skip-gram	Target word	Context words

Avg Word2Vec

- **Average Word2Vec** is a simple and effective method to **combine individual word embeddings** into a single vector representation for a document, sentence, or paragraph.
- Instead of relying on traditional Word2Vec models that generate vectors for individual words, Avg Word2Vec **averages the embeddings of all words** in a given text.
- How Does Avg Word2Vec Work?
  - Step 1: Use a pretrained Word2Vec model (or train a new one) to generate vector representations for each word.
  - Step 2: For a given text (sentence, paragraph, etc.), compute the average of the word embeddings for all words in the text.
  - Step 3: The resulting vector is the text **representation for the entire document**.

Avg Word2Vec

- Example of Avg Word2Vec:
- Sentence: "The dog barks loudly.“
- Get word vectors for "The", "dog", "barks", "loudly".
- Avg Word2Vec vector = (vector for "The" + vector for "dog" + vector for "barks" + vector for "loudly") / 4
- Doc1 = The day is good.

$$\begin{array}{ccccccccc} \text{The} & & \text{day} & & \text{is} & & \text{good} & = & \text{Doc1} \\ \begin{bmatrix} 0.25 \\ 1 \\ 0.18 \\ 0.24 \\ 0.28 \\ 0.05 \\ . \\ . \\ . \\ 0.63 \end{bmatrix} & + & \begin{bmatrix} 1 \\ 0.5 \\ 0.8 \\ 0.02 \\ 0.8 \\ 0.95 \\ . \\ . \\ . \\ 0.26 \end{bmatrix} & + & \begin{bmatrix} 0.9 \\ 0.75 \\ 0.28 \\ 0.2 \\ 0.7 \\ 0.5 \\ . \\ . \\ . \\ 0.32 \end{bmatrix} & + & \begin{bmatrix} 0.25 \\ 1 \\ 0.18 \\ 0.24 \\ 0.28 \\ 0.05 \\ . \\ . \\ . \\ 0.63 \end{bmatrix} & = & \begin{bmatrix} 0.6 \\ 0.81 \\ 0.36 \\ 0.17 \\ 0.51 \\ 0.38 \\ . \\ . \\ . \\ 0.46 \end{bmatrix} \end{array}$$

Google Word2Vec pretrained model

```
!pip install gensim
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: gensim in /usr/local/lib/python3.7/dist-packages (3.6.0)
Requirement already satisfied: numpy>=1.11.3 in /usr/local/lib/python3.7/dist-packages (from gensim) (1.21.6)
Requirement already satisfied: smart-open>=1.2.1 in /usr/local/lib/python3.7/dist-packages (from gensim) (5.2.1)
Requirement already satisfied: six>=1.5.0 in /usr/local/lib/python3.7/dist-packages (from gensim) (1.15.0)
Requirement already satisfied: scipy>=0.18.1 in /usr/local/lib/python3.7/dist-packages (from gensim) (1.7.3)
```

```
[ ] import gensim
```

```
[ ] from gensim.models import Word2Vec, KeyedVectors
```

```
[ ] ## References: https://stackoverflow.com/questions/46433778/import-googlenews-vectors-negative300-bin
```

```
[ ] import gensim.downloader as api

    wv = api.load('word2vec-google-news-300')

    vec_king = wv['king']
```

```
[==================================================] 100.0% 1662.8/1662.8MB downloaded
```

Google Word2Vec pretrained model

```
[ ]  vec_king
```

```
array([ 1.25976562e-01,  2.97851562e-02,  8.60595703e-03,  1.39648438e-01,
       -2.56347656e-02, -3.61328125e-02,  1.11816406e-01, -1.98242188e-01,
        5.12695312e-02,  3.63281250e-01, -2.42187500e-01, -3.02734375e-01,
       -1.77734375e-01, -2.49023438e-02, -1.67968750e-01, -1.69921875e-01,
        3.46679688e-02,  5.21850586e-03,  4.63867188e-02,  1.28906250e-01,
        1.36718750e-01,  1.12792969e-01,  5.95703125e-02,  1.36718750e-01,
        1.01074219e-01, -1.76757812e-01, -2.51953125e-01,  5.98144531e-02,
        3.41796875e-01, -3.11279297e-02,  1.04492188e-01,  6.17675781e-02,
        1.24511719e-01,  4.00390625e-01, -3.22265625e-01,  8.39843750e-02,
        3.90625000e-02,  5.85937500e-03,  7.03125000e-02,  1.72851562e-01,
        1.38671875e-01, -2.31445312e-01,  2.83203125e-01,  1.42578125e-01,
        3.41796875e-01, -2.39257812e-02, -1.09863281e-01,  3.32031250e-02,
       -5.46875000e-02,  1.53198242e-02, -1.62109375e-01,  1.58203125e-01,
       -2.59765625e-01,  2.01416016e-02, -1.63085938e-01,  1.35803223e-03,
       -1.44531250e-01, -5.68847656e-02,  4.29687500e-02, -2.46582031e-02,
        1.85546875e-01,  4.47265625e-01,  9.58251953e-03,  1.31835938e-01,
        9.86328125e-02, -1.85546875e-01, -1.00097656e-01, -1.33789062e-01,
       -1.25000000e-01,  2.83203125e-01,  1.23046875e-01,  5.32226562e-02,
       -1.77734375e-01,  8.59375000e-02, -2.18505859e-02,  2.05078125e-02,
       -1.39648438e-01,  2.51464844e-02,  1.38671875e-01, -1.05468750e-01,
        1.38671875e-01,  8.88671875e-02, -7.51953125e-02, -2.13623047e-02,
        1.72851562e-01,  4.63867188e-02, -2.65625000e-01,  8.91113281e-03,
        1.49414062e-01,  3.78417969e-02,  2.38281250e-01, -1.24511719e-01,
       -2.17773438e-01, -1.81640625e-01,  2.97851562e-02,  5.71289062e-02,
       -2.89306641e-02,  1.24511719e-02,  9.66796875e-02, -2.31445312e-01,
        5.81054688e-02,  6.68945312e-02,  7.08007812e-02, -3.08593750e-01,
       -2.14843750e-01,  1.45507812e-01, -4.27734375e-01, -9.39941406e-03,
        1.54296875e-01, -7.66601562e-02,  2.89062500e-01,  2.77343750e-01,
       -4.86373901e-04, -1.36718750e-01,  3.24218750e-01, -2.46093750e-01,
       -3.03649902e-03, -2.11914062e-01,  1.25000000e-01,  2.69531250e-01,
        2.04101562e-01,  8.25195312e-02, -2.01171875e-01, -1.60156250e-01,
       -3.78417969e-02, -1.20117188e-01,  1.15234375e-01, -4.10156250e-02,
       -3.95507812e-02. -8.98437500e-02.  6.34765625e-03.  2.03125000e-01.
```

Google Word2Vec pretrained model

```
[ ] vec_king.shape
```

```
(300,)
```

```
[ ] wv['cricket']
```

```
 4.49218750e-01, -1.36718750e-01, -2.34375000e-01,  4.12597656e-02,
-2.15820312e-01,  1.69921875e-01,  2.56347656e-02,  1.50146484e-02,
-3.75976562e-02,  6.95800781e-03,  4.00390625e-01,  2.09960938e-01,
 1.17675781e-01, -4.19921875e-02,  2.34375000e-01,  2.03125000e-01,
-1.86523438e-01, -2.46093750e-01,  3.12500000e-01, -2.59765625e-01,
-1.06933594e-01,  1.04003906e-01, -1.79687500e-01,  5.71289062e-02,
-7.41577148e-03, -5.59082031e-02,  7.61718750e-02, -4.14062500e-01,
-3.65234375e-01, -3.35937500e-01, -1.54296875e-01, -2.39257812e-01,
-3.73046875e-01,  2.27355957e-03, -3.51562500e-01,  8.64257812e-02,
 1.26953125e-01,  2.21679688e-01, -9.86328125e-02,  1.08886719e-01,
 3.65234375e-01, -5.66406250e-02,  5.66406250e-02, -1.09375000e-01,
-1.66992188e-01, -4.54101562e-02, -2.00195312e-01, -1.22558594e-01,
 1.31835938e-01, -1.31835938e-01,  1.03027344e-01, -3.41796875e-01,
-1.57226562e-01,  2.04101562e-01,  4.39453125e-02,  2.44140625e-01,
-3.19824219e-02,  3.20312500e-01, -4.41894531e-02,  1.08398438e-01,
-4.98046875e-02, -9.52148438e-03,  2.46093750e-01, -5.59082031e-02,
 4.07714844e-02, -1.78222656e-02, -2.95410156e-02,  1.65039062e-01,
 5.03906250e-01, -2.81250000e-01,  9.81445312e-02,  1.80664062e-02,
-1.83593750e-01,  2.53906250e-01,  2.25585938e-01,  1.63574219e-02,
 1.81640625e-01,  1.38671875e-01,  3.33984375e-01,  1.39648438e-01,
 1.45874023e-02, -2.89306641e-02, -8.39843750e-02,  1.50390625e-01,
 1.67968750e-01,  2.28515625e-01,  3.59375000e-01,  1.22558594e-01,
-3.28125000e-01, -1.56250000e-01,  2.77343750e-01,  1.77001953e-02,
-1.46484375e-01, -4.51660156e-03, -4.46777344e-02,  1.75781250e-01,
-3.75000000e-01,  1.16699219e-01, -1.39648438e-01,  2.55859375e-01,
-1.96289062e-01, -2.57568359e-02, -5.41992188e-02, -2.51464844e-02,
-1.93359375e-01, -3.17382812e-02, -8.74023438e-02, -1.32812500e-01,
```

```
[ ] wv.most_similar('cricket')
```

```
[('cricketing', 0.8372225165367126),
 ('cricketers', 0.8165745735168457),
 ('Test_cricket', 0.8094818592071533),
 ('Twenty##_cricket', 0.8068488240242004),
 ('Twenty##', 0.7624266147613525),
 ('Cricket', 0.7541396617889404),
 ('cricketer', 0.7372579574584961),
 ('twenty##', 0.7316356897354126),
 ('T##_cricket', 0.7304614782333374),
 ('West_Indies_cricket', 0.6987986564636230)]
```

```
[ ] wv.most_similar('happy')
```

```
[('glad', 0.7408890128135681),
 ('pleased', 0.6632171273231506),
 ('ecstatic', 0.6626912355422974),
 ('overjoyed', 0.6599286794662476),
 ('thrilled', 0.6514049768447876),
 ('satisfied', 0.6437950134277344),
 ('proud', 0.636042058467865),
 ('delighted', 0.627237856388092),
 ('disappointed', 0.6269949674606323),
 ('excited', 0.6247666478157043)]
```

```
[ ] wv.similarity("hockey","sports")
```

```
0.53541523
```

Google Word2Vec pretrained model

```
[ ] vec=wv['king']-wv['man']+wv['woman']
```

```
[ ] vec
```

```
array([ 4.29687500e-02, -1.78222656e-01, -1.29089355e-01,  1.15234375e-01,
        2.68554688e-03, -1.02294922e-01,  1.95800781e-01, -1.79504395e-01,
        1.95312500e-02,  4.09919739e-01, -3.68164062e-01, -3.96484375e-01,
       -1.56738281e-01,  1.46484375e-03, -9.30175781e-02, -1.16455078e-01,
       -5.51757812e-02, -1.07574463e-01,  7.91015625e-02,  1.98974609e-01,
        2.38525391e-01,  6.34002686e-02, -2.17285156e-02,  0.00000000e+00,
        4.72412109e-02, -2.17773438e-01, -3.44726562e-01,  6.37207031e-02,
        3.16406250e-01, -1.97631836e-01,  8.59375000e-02, -8.11767578e-02,
       -3.71093750e-02,  3.15551758e-01, -3.41796875e-01, -4.68750000e-02,
        9.76562500e-02,  8.39843750e-02, -9.71679688e-02,  5.17578125e-02,
       -5.00488281e-02, -2.20947266e-01,  2.29492188e-01,  1.26403809e-01,
        2.49023438e-01,  2.09960938e-02, -1.09863281e-01,  5.81054688e-02,
       -3.35693359e-02,  1.29577637e-01,  2.41699219e-02,  3.48129272e-02,
       -2.60009766e-01,  2.42309570e-01, -3.21777344e-01,  1.45416260e-02,
       -1.59179688e-01, -8.37402344e-02,  1.65039062e-01,  1.58691406e-03,
        3.09570312e-01,  3.16406250e-01,  7.38525391e-03,  2.41210938e-01,
        4.90722656e-02, -9.86328125e-02,  2.90527344e-02,  1.49414062e-01,
       -4.83398438e-02,  2.35595703e-01,  2.21191406e-01,  1.25488281e-01,
       -1.38671875e-01,  1.54296875e-01,  7.18994141e-02,  1.29882812e-01,
       -1.05712891e-01,  6.00585938e-02,  3.14697266e-01,  1.09619141e-01,
        8.49609375e-02,  7.71484375e-02, -2.17285156e-02,  6.11572266e-02,
       -1.89941406e-01,  2.07519531e-01, -1.63085938e-01,  1.13525391e-01,
        2.01171875e-01,  6.06689453e-02,  1.27929688e-01, -3.11279297e-01,
       -2.80151367e-01, -1.55883789e-01,  4.15039062e-02,  9.87854004e-02,
        1.69555664e-01, -3.49121094e-02,  2.08496094e-01, -9.89990234e-02,
        4.39453125e-03, -7.27539062e-02, -4.24804688e-02, -4.09179688e-01,
       -2.76367188e-01,  1.64062500e-01, -5.57617188e-01, -2.02199936e-01,
```

```
[ ] wv.most_similar([vec])
```

```
[('king', 0.8449392318725586),
 ('queen', 0.7300517559051514),
 ('monarch', 0.645466089248657 2),
 ('princess', 0.6156251430511475),
 ('crown_prince', 0.5818676948547363),
 ('prince', 0.5777117609977722),
 ('kings', 0.5613663792610168),
 ('sultan', 0.5376776456832886),
 ('Queen_Consort', 0.5344247817993164),
 ('queens', 0.5289887189865112)]
```

Homework and practice tasks

1. Given the following simplified embeddings:
   - king = [0.9, 0.8, 0.7]
   - queen = [0.8, 0.7, 0.9]
   - man = [0.7, 0.9, 0.6]
   - woman = [0.6, 0.8, 0.8]

   Compute **cosine similarity** between:
   - (king, queen)
   - (man, woman)
2. If the embedding of “cat” = [0.2, 0.4, 0.6] and “dog” = [0.3, 0.5, 0.7], find the **Euclidean distance** between them.
3. The analogy task “King – Man + Woman = ?” is often used in Word2Vec. Explain the concept behind this analogy and what result it aims to produce.
4. Suppose we use one-hot encoding for a vocabulary of size 10,000.
   - What will be the vector size for each word?
   - How does word embedding improve efficiency compared to this?
5. You are using Word2Vec with a **window size of 3**. Explain what the window size means and how it affects training.