Welcome Here!!

NATURAL LANGUAGE PROCESSING (NLP)

NUMERICAL REPRESENTATION (WORD EMBEDDING) & RECURRENT NEURAL NETWORK

DSN LEKKI-AJAH

BY ABEREJO HABEEBLAH O.

X: @HABEREJO

Day 3

Tuesday, 18th March, 2025

Why are we here

To equip learners with the skills to understand how Artificial Intelligence models are built, explore Machine Learning and Deep Learning, and focus on Natural Language Processing (NLP).

We should build some project right?

Yes!!

If you are in with me,

we should build some project.

Classes Structure

Every Tuesday throughout the month of March.

4 classes in total. This is the third class

Each class for about 1:45 minutes

Days are NOT likely to change, although, my schedule can be OUCH, I wil reachout earlier before. Understand me bikoooo

Who should be here

- You're curious and ready to learn something new
- You dream of becoming a Data Scientist.
- You're passionate about building the future as a Machine Learning Engineer.
- You have a basic understanding of Python programming.
- You've worked with data and want to take your skills to the next level.
- You're a researcher exploring the exciting world of AI.

which of these are you?

Who should be here

- You're familiar with the fundamentals of Machine Learning and Deep Learning.
- You love problem solving and are excited to tackle real world problems with AI.
- You want to understand the technology that is shaping the future.
- You're driven by a desire to learn and grow in the field of AI.
- You enjoy collaborating and learning from others.

Which of these are you?????
You ticked any of these boxes?
Let's Goooooooo

Before we begin

Who I am?

- A Machine Learning Engineer
- A young MAN eager to master AI

FUN FACT:
I've taught over 100 people Python, but
I'm still learning new things every day.
(Especially how to avoid typos when
coding late at night!)

Scan to visit my portfolio
or
https://bheez.netlify.app

Before we begin

Quick Recap (First Class)

1️⃣ **Understanding AI, ML, DL, and NLP**

2️⃣ **Introduction to NLP**

3️⃣ **NLP Use Cases:** Translation Apps 🌍, Spam Filters 📧, Search Engines 🔍, Sentiment Analysis 😀 😡, Voice Assistants 🎙️, Chatbots 💬, Autocorrect & Predictive Text 📱.

4️⃣ **How Computer Understands Text:** 1. Text Preprocessing 2. Tokenization, 3. Numerical Representation (Word Embedding)

5️⃣ **Text Preprocessing Techniques:** Lowercasing, Removing Punctuation Marks, Removing Stopwords (i.e. like A, of, in, etc), Stemming / Lemmatization.

Before we begin

Quick Recap (Second Class)

1 **How Computer Understands Text:** 1. Text Preprocessing

2 **Text Preprocessing Techniques:** Lowercasing, Removing Punctuation Marks, Removing Stopwords (i.e. like A, of, in, etc), Stemming / Lemmatization.

3 **How Computer Understands Text:** 2. Tokenization

4 Tokenization: **Sentence Tokenization** (Sentence-Level Tokenization)

Word Tokenization (Word-Level Tokenization)

Subword Tokenization

Character-level Tokenization

5 **How Computer Understands Text:** 3. Numerical Representation

6 **Numerical Representation:** Vocabulary and Integer Encoding

7 **Numerical Representation:** One Hot Encoding

Course Structure

1. Numerical Representation: Word Embedding
2. Working with **Word2Vec**
3. Working with **Glove**
4. Discussing **t-distributed stochastic neighbor embedding, t-SNE (pronounced tee-snee)**
5. **FastText**
6. **BERT** (bi-directional encoder representations from transformers)

And more

Step 3: Numerical Representation

There are several ways to do this:

a) Word Embeddings (aka Word Vectors) which uses techniques like **Word2Vec, GloVe, FastText, BERT, or by creating an embedding layer and training** (i.e creating your model to be used for embedding).

b) One-Hot Encoding

c) Vocabulary and Integer Encoding

Step 3: Numerical Representation: Word Embedding

Word Embeddings (aka Word Vectors): a specific and powerful type of numerical representation designed to capture **semantic meaning.**

Vector representations of words are the information-dense alternative to one-hot encodings of words. Whereas one-hot representations capture information about word location only, word vectors **(also known as word embeddings or vector-space embeddings)** capture information about word meaning as well as location.

Step 3: Numerical Representation: Word Embedding

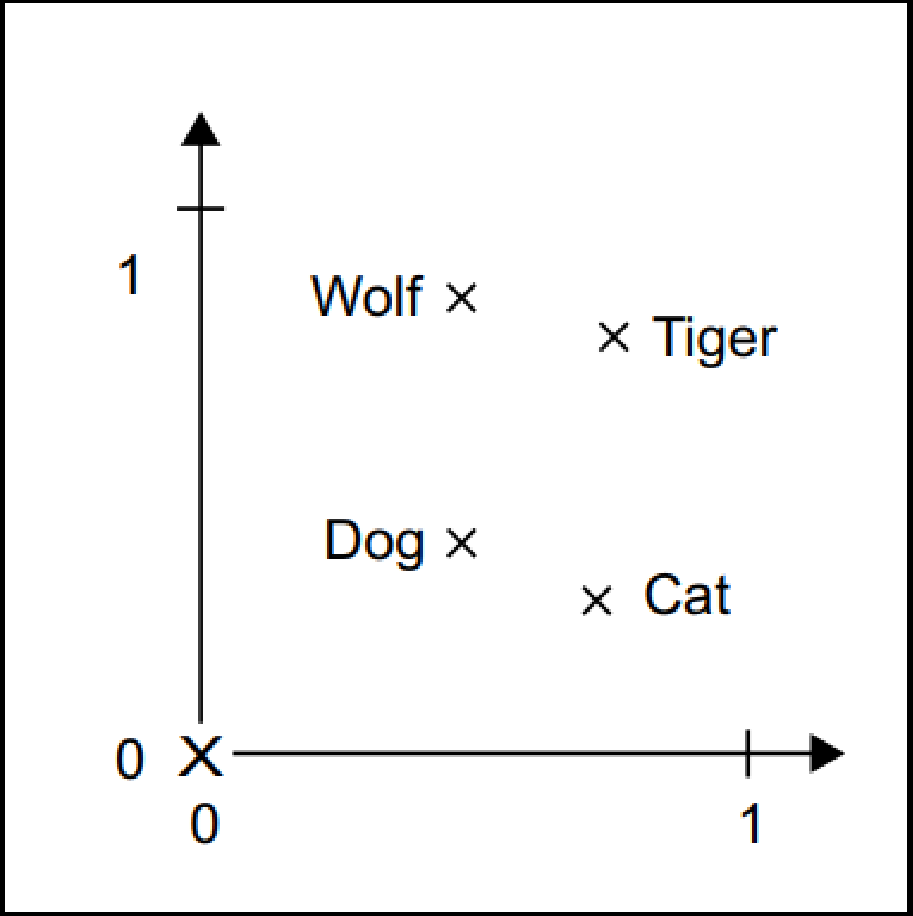
canine== dog like, with pointy tooth btw incisors. FELINE == cat like,

The key advantage, word vectors enable deep learning NLP models to automatically learn linguistic features.

For instance, four words are embedded on a 2D plane: cat, dog, wolf, and tiger.

The same vector allows us to go from **cat** to **tiger** and from **dog** to **wolf.** This vector could be interpreted as the **"from pet to wild animal"** vector.

Similarly, another vector lets us go from **dog** to **cat** and from **wolf** to **tiger**, which could be interpreted as a **"from canine to feline"** vector.



Step 3: Numerical Representation: Word Embedding

In real-world word-embedding spaces, common examples of meaningful geometric transformations are “**gender**” vectors and “**plural**” vectors.

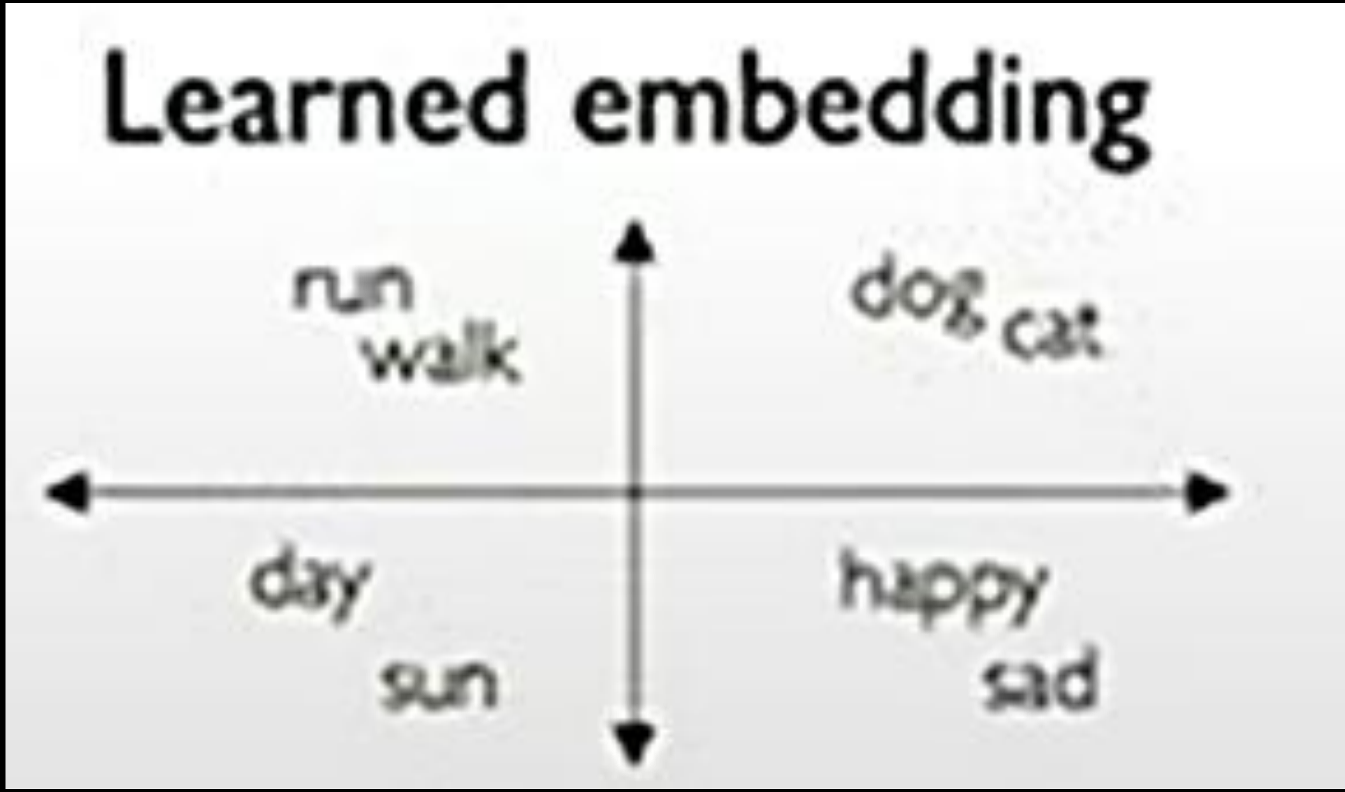
For instance, by adding a “**female**” vector to the vector “**king,**” we obtain the vector “**queen.**”

By adding a “**plural**” vector, we obtain “**kings.**”

Remember one-hot encoding? It tells us where a word is in a list, but it doesn't tell us anything about its meaning. It's like having a list of cities but no map to show how they're related.

Step 3: Numerical Representation: Word Embedding

So basically, Word Embedding converts the words into number such that, things that are related to each other in language, should numerically be similar and close to each other in the space and things that are very dissimilar should numerically dissimilar and far from away in the space.



Step 3: Numerical Representation: Word Embedding

There are two ways to obtain word embeddings:

1. Learn word embeddings jointly with the main task you care about **(such as document classification or sentiment prediction)**. In this setup, **you start with random word vectors** and then learn word vectors in the same way you learn **the weights of a neural network.**

   weights of a neural neural??? Should I explain???

2. Load into your model word embeddings that were precomputed using a different machine learning task than the one you're trying to solve. These are called **pretrained word embeddings**. They include Word2Vec, GloVe, FastText, BERT. **Word2Vec, GloVe, are most popular.**

Step 3: Numerical Representation: Word Embedding

Keras is a framework like Tensorflow but lightweight

Why should we learn (or create) a new embedding space with every new task create out word embedding?

- Because, what makes a good word-embedding space depends heavily on your task
- The perfect word-embedding space for an **English-language movie-review sentiment-analysis model** may look different from the perfect embedding space for an **English-language legal-document classification model**, because the importance of certain semantic relationships varies from task
- It's thus reasonable to learn a new embedding space with every new task.
- Fortunately, backpropagation makes this easy, and **Keras** makes it even easier. It's about learning the weights of a layer.

Step 3: Numerical Representation: Word Embedding

Keras is a framework like Tensorflow but lightweight

Learning (or creating) a new embedding space.

- Instantiating an Embedding layer

```
embedding_layer = layers.Embedding(input_dim=max_tokens,
output_dim=256)
```

```
# Embedding layer: Converts integer sequences to dense vectors (e
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)
```

The Embedding layer takes at least two arguments, max_token & output_dim.

Parameters: a) input_dim: The number of unique tokens in your vocabulary (the size of your "word list").

b) output_dim: The dimensionality of the embeddings (how many numbers to represent each word).

Step 3: Numerical Representation: Word Embedding

```python
import tensorflow as tf  # Import TensorFlow library
from tensorflow import keras  # Import Keras module from TensorFlow
from tensorflow.keras import layers  # Import layers module from Keras

# --- Data Preparation (Dummy Data for Demonstration) --- # Define example parameters for the data
max_tokens = 10000  # Example vocabulary size (number of unique words/tokens)
input_length = 50   # Example sequence length (maximum length of input sequences)
batch_size = 32     # Batch size for training

# Create dummy training dataset using tf.data.Dataset
int_train_ds = tf.data.Dataset.from_tensor_slices(
    (tf.random.uniform(shape=(1000, input_length), minval=0, maxval=max_tokens, dtype=tf.int64),  # Input sequences (random integers)
     tf.random.uniform(shape=(1000,), minval=0, maxval=2, dtype=tf.int64))  # Target labels (random 0 or 1)
).batch(batch_size)  # Batch the dataset into batches of size batch_size

# Create dummy validation dataset using tf.data.Dataset
int_val_ds = tf.data.Dataset.from_tensor_slices(
    (tf.random.uniform(shape=(200, input_length), minval=0, maxval=max_tokens, dtype=tf.int64),  # Input sequences (random integers)
     tf.random.uniform(shape=(200,), minval=0, maxval=2, dtype=tf.int64))  # Target labels (random 0 or 1)
).batch(batch_size)  # Batch the dataset into batches of size batch_size

# --- Model Definition ---
# Define the input layer, taking integer sequences as input
inputs = keras.Input(shape=(None,), dtype="int64")  # shape=(None,) allows variable sequence lengths

# Embedding layer: Converts integer sequences to dense vectors (embeddings)
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)  # 256-dimensional embeddings

# Bidirectional LSTM layer: Processes the embedded sequences in both forward and backward directions
x = layers.Bidirectional(layers.LSTM(32))(embedded)  # 32 units in each LSTM layer
```

In summary, we created a dummy training dataset and dummy validation dataset, understand that the dummy generated dataset are random integers which serves as tokenized dataset, therefore, I could move directly into the embedding layer for word embedding

Step 3: Numerical Representation: Word Embedding

Why should be use pretrained word embeddings?

- They are trained on massive datasets.
- Capture a lot of semantic information.
- Save training time.

Pretrained word embedding techniques include: Word2Vec, Global Vectors for Word Representation (GloVe), FastText, BERT and others.

Word2Vec, GloVe, are most popular.

NB

Step 3: Numerical Representation: Word Embedding

Word2Vec developed by **Tomas Mikolov at Google in 2013.**

Word2Vec learns by looking at the words surrounding each target word. It tries to predict the context.

Word2Vec dimensions capture specific semantic properties, such as gender.

NB

Step 3: Numerical Representation: Word Embedding

To Use Word2Vec:

NB: Loading the data should take about 2gb of data.

Load the model

```
import gensim.downloader as api  # Import the Gensim downloader for pre-trained models
from sklearn.manifold import TSNE  # Import t-SNE for dimensionality reduction (visualization)
import matplotlib.pyplot as plt  # Import Matplotlib for plotting

# --- 1. Load Pre-trained Word2Vec Model ---

# Load the pre-trained Word2Vec model trained on Google News dataset (300-dimensional vectors)
# 'word2vec-google-news-300' is the model identifier in Gensim's downloader
wv = api.load('word2vec-google-news-300')  # wv stands for word vectors
```

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

Step 3: Numerical Representation: Word Embedding

To Use Word2Vec:

As seen, similarity btw king and car is LOW

Apply the model to some text:

```
# --- 2. Example 1: Semantic Similarity ---

# Calculate and print the cosine similarity between word vectors
# Cosine similarity measures how similar two vectors are (between -1 and 1)
print(wv.similarity('king', 'queen'))  # Similarity between 'king' and 'queen' (should be high)
print(wv.similarity('man', 'woman'))  # Similarity between 'man' and 'woman' (should be high)
print(wv.similarity('king', 'man'))  # Similarity between 'king' and 'man' (should be high)
print(wv.similarity('king', 'car'))  # Similarity between 'king' and 'car' (should be low)
```

```
0.6510957
0.76640123
0.22942673
0.061895393
```

Step 3: Numerical Representation: Word Embedding

As seen, similarity btw king and car is LOW

To Use Word2Vec:

Let's give an analogy, Find words most similar to 'king' + 'woman' - 'man'. (This should be close to 'queen')

```
result = wv.most_similar(positive=['king', 'woman'], negative=['man'])
print(result)
```

RESULT: `[('queen', 0.7118193507194519), ('monarch', 0.6189674139022827), ('princess', 0.5902431011199951), ('crown_prince', 0.5499460697174072), ('prince', 0.5377321839332581), ('kings', 0.5236844420433044), ('Queen_Consort', 0.5235945582389832), ('queens', 0.5181134343147278), ('sultan', 0.5098593831062317), ('monarchy', 0.5087411999702454)]`

This demonstrates the ability of Word2Vec to solve analogy problems

Step 3: Numerical Representation: Word Embedding

To Use Word2Vec:

Let’s give another analogy,

Find words most similar to 'king’ + girl’ + ‘young’ - 'man’ – ‘adult’.

What could this be??

```
result = wv.most_similar(positive=['king', 'girl', 'young'], negative=['man', 'adult'])
print(result)
```

Prince? King child? Princess?
Prince_Paras???

Step 3: Numerical Representation: Word Embedding

A Princess!!

Because, A princess relates to a king, is a girl, is young, is not a man, and not an adult.

```
result = wv.most_similar(positive=['king', 'girl', 'young'], negative=['man', 'adult'])
print(result)
```

```
[('princess', 0.4750033020973205б), ('prince', 0.470218300819397), ('Prince_Paras', 0.4611
```

As seen, princess has the highest.

Step 3: Numerical Representation: Word Embedding

To Use Word2Vec:

To get the vector for a word:

```
# --- 4. Example 3: Getting the Vector for a Word ---

# Retrieve the vector representation of the word 'king'
vector_king = wv['king']
print(vector_king)  # Print the vector (300 numbers)
```

The Vector for the word KING is:

```python
# Retrieve the vector representation of the word 'king'
vector_king = wv['king']
print(vector_king)  # Print the vector (300 numbers)
```

```
[ 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
  1.86523438e-01  2.73437500e-01  6.29882812e-02  1.41601562e-01
 -9.81445312e-02  1.38671875e-01  1.82617188e-01  1.73828125e-01
  1.73828125e-01 -2.37304688e-01  1.78710938e-01  6.34765625e-02
  2.36328125e-01 -2.08984375e-01  8.74023438e-02 -1.66015625e-01
 -7.91015625e-02  2.43164062e-01 -8.88671875e-02  1.26953125e-01
 -2.16796875e-01 -1.73828125e-01 -3.59375000e-01 -8.25195312e-02
 -6.49414062e-02  5.07812500e-02  1.35742188e-01 -7.47070312e-02
 -1.64062500e-01  1.15356445e-02  4.45312500e-01 -2.15820312e-01
 -1.11328125e-01 -1.92382812e-01  1.70898438e-01 -1.25000000e-01
  2.65502930e-03  1.92382812e-01 -1.74804688e-01  1.39648438e-01
  2.92968750e-01  1.13281250e-01  5.95703125e-02 -6.39648438e-02
  9.96093750e-02 -2.72216797e-02  1.96533203e-02  4.27246094e-02
 -2.46093750e-01  6.39648438e-02 -2.25585938e-01 -1.68945312e-01
  2.89916992e-03  8.20312500e-02  3.41796875e-01  4.32128906e-02
  1.32812500e-01  1.42578125e-01  7.61718750e-02  5.98144531e-02
 -1.19140625e-01  2.74658203e-03 -6.29882812e-02 -2.72216797e-02
 -4.82177734e-03 -8.20312500e-02 -2.49023438e-02 -4.00390625e-01
 -1.06933594e-01  4.24804688e-02  7.76367188e-02 -1.16699219e-01
  7.37304688e-02 -9.22851562e-02  1.07910156e-01  1.58203125e-01
  4.24804688e-02  1.26953125e-01  3.61328125e-02  2.67578125e-01
 -1.01074219e-01 -3.02734375e-01 -5.76171875e-02  5.05371094e-02
  5.26428223e-04 -2.07031250e-01 -1.38671875e-01 -8.97216797e-03
 -2.78320312e-02 -1.41601562e-01  2.07031250e-01 -1.58203125e-01
  1.27929688e-01  1.49414062e-01 -2.24609375e-02 -8.44726562e-02
```

Step 3: Numerical Representation: Word Embedding

The Vector for the word KING is:

```
[ 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 1.86523438e-01 2.73437500e-01
6.29882812e-02 1.41601562e-01 -9.81445312e-02 1.38671875e-01 1.82617188e-01 1.73828125e-01 1.73828125e-01 -2.37304688e-01 1.78710938e-01 6.34765625e-02
2.36328125e-01 -2.08984375e-01 8.74023438e-02 -1.66015625e-01 -7.91015625e-02 2.43164062e-01 -8.88671875e-02 1.26953125e-01 -2.16796875e-01 -
1.73828125e-01 -3.59375000e-01 -8.25195312e-02 -6.49414062e-02 5.07812500e-02 1.35742188e-01 -7.47070312e-02 -1.64062500e-01 1.15356445e-02 4.45312500e-
01 -2.15820312e-01 -1.11328125e-01 -1.92382812e-01 1.70898438e-01 -1.25000000e-01 2.65502930e-03 1.92382812e-01 -1.74804688e-01 1.39648438e-01
2.92968750e-01 1.13281250e-01 5.95703125e-02 -6.39648438e-02 9.96093750e-02 -2.72216797e-02 1.96533203e-02 4.27246094e-02 -2.46093750e-01 6.39648438e-02
-2.25585938e-01 -1.68945312e-01 2.89916992e-03 8.20312500e-02 3.41796875e-01 4.32128906e-02 1.32812500e-01 1.42578125e-01 7.61718750e-02 5.98144531e-02
-1.19140625e-01 2.74658203e-03 -6.29882812e-02 -2.72216797e-02 -4.82177734e-03 -8.20312500e-02 -2.49023438e-02 -4.00390625e-01 -1.06933594e-01
4.24804688e-02 7.76367188e-02 -1.16699219e-01 7.37304688e-02 -9.22851562e-02 1.07910156e-01 1.58203125e-01 4.24804688e-02 1.26953125e-01 3.61328125e-02
2.67578125e-01 -1.01074219e-01 -3.02734375e-01 -5.76171875e-02 5.05371094e-02 5.26428223e-04 -2.07031250e-01 -1.38671875e-01 -8.97216797e-03 -
2.78320312e-02 -1.41601562e-01 2.07031250e-01 -1.58203125e-01 1.27929688e-01 1.49414062e-01 -2.24609375e-02 -8.44726562e-02 1.22558594e-01 2.15820312e-
01 -2.13867188e-01 -3.12500000e-01 -3.73046875e-01 4.08935547e-03 1.07421875e-01 1.06933594e-01 7.32421875e-02 8.97216797e-03 -3.88183594e-02 -
1.29882812e-01 1.49414062e-01 -2.14843750e-01 -1.83868408e-03 9.91210938e-02 1.57226562e-01 -1.14257812e-01 -2.05078125e-01 9.91210938e-02 3.69140625e-
01 -1.97265625e-01 3.54003906e-02 1.09375000e-01 1.31835938e-01 1.66992188e-01 2.35351562e-01 1.04980469e-01 -4.96093750e-01 -1.64062500e-01 -
1.56250000e-01 -5.22460938e-02 1.03027344e-01 2.43164062e-01 -1.88476562e-01 5.07812500e-02 -9.37500000e-02 -6.68945312e-02 2.27050781e-02 7.61718750e-
02 2.89062500e-01 3.10546875e-01 -5.37109375e-02 2.28515625e-01 2.51464844e-02 6.78710938e-02 -1.21093750e-01 -2.15820312e-01 -2.73437500e-01 -
3.07617188e-02 -3.37890625e-01 1.53320312e-01 2.33398438e-01 -2.08007812e-01 3.73046875e-01 8.20312500e-02 2.51953125e-01 -7.61718750e-02 -4.66308594e-
02 -2.23388672e-02 2.99072266e-02 -5.93261719e-02 -4.66918945e-03 -2.44140625e-01 -2.09960938e-01 -2.87109375e-01 -4.54101562e-02 -1.77734375e-01 -
2.79296875e-01 -8.59375000e-02 9.13085938e-02 2.51953125e-01]
```

Step 3: Numerical Representation: Word Embedding

To Use Word2Vec:

To check if a word exists in the vocabulary:

```
# Check if the word 'cat' exists in the Word2Vec vocabulary
if 'cat' in wv:
    print("Cat is in the vocabulary")
else:
    print("Cat is not in the vocabulary")
```

```
Cat is in the vocabulary
```

Step 3: Numerical Representation: Word Embedding

To Use Word2Vec: To Visualize Embeddings(using t-SNE):

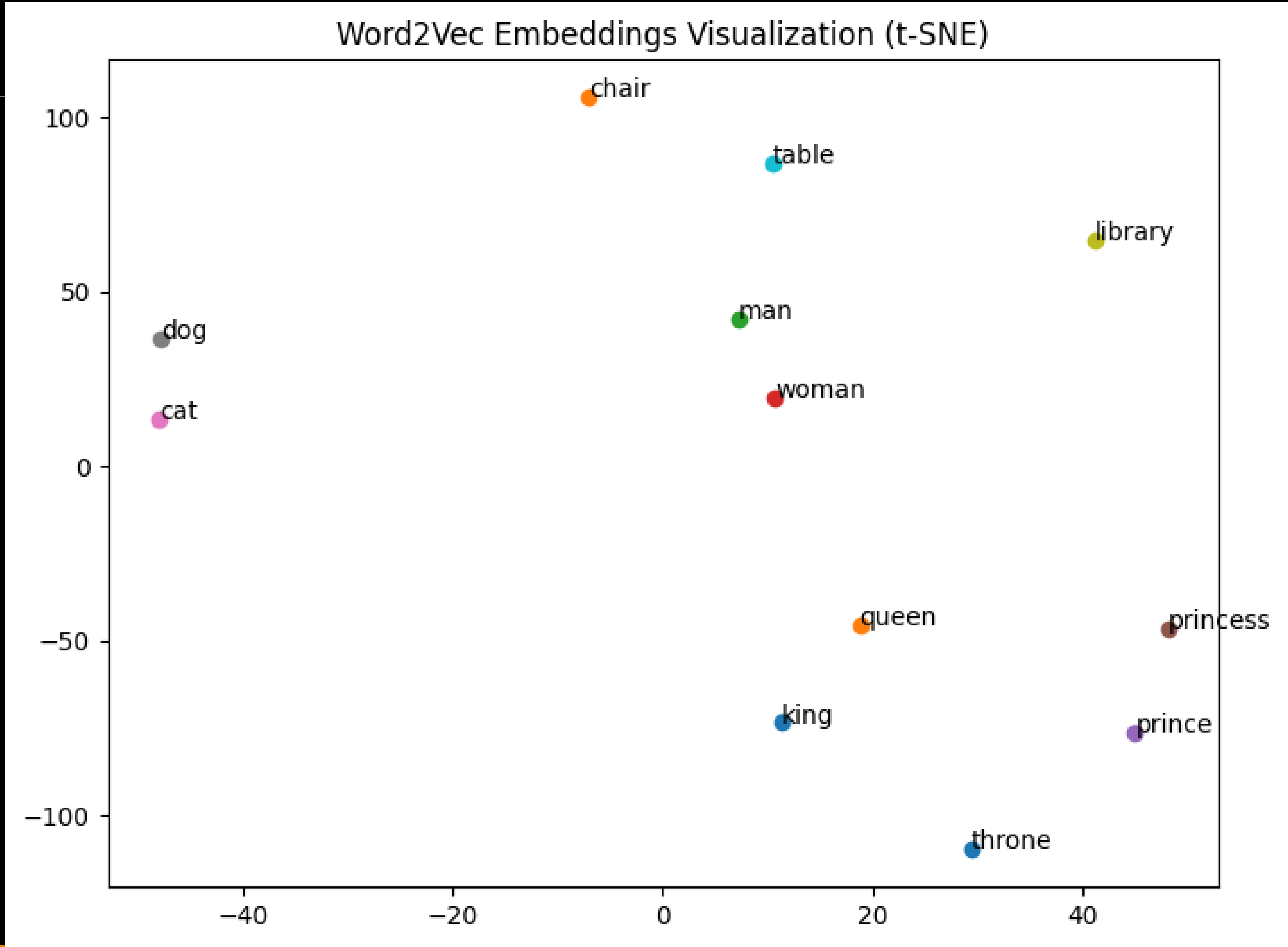
```
# List of words to visualize
words = ['king', 'queen', 'man', 'woman', 'prince', 'princess', 'cat', 'dog', 'library', 'table', 'throne', 'chair']

# Get the vector representations for the words (only if they are in the vocabulary)
embeddings = [wv[word] for word in words if word in wv]

# Convert the list of embeddings to a 2D NumPy array
import numpy as np  # Import NumPy for array manipulation
embeddings = np.array(embeddings) # Convert the list of embeddings to a NumPy array

# Reduce the dimensionality of the vectors to 2D for visualization using t-SNE
# t-SNE (t-Distributed Stochastic Neighbor Embedding) is a technique for visualizing high-dimensional data
# Set perplexity to a value less than the number of samples (8 in this case)
tsne = TSNE(n_components=2, perplexity=5, random_state=42)  # Reduce to 2 dimensions, perplexity=5
embeddings_2d = tsne.fit_transform(embeddings)

# Create a scatter plot of the 2D embeddings
plt.figure(figsize=(8, 6))  # Set the figure size
for i, word in enumerate(words):
    if word in wv:  # Only plot if the word is in the vocabulary
        plt.scatter(embeddings_2d[i, 0], embeddings_2d[i, 1])  # Plot the point
        plt.annotate(word, (embeddings_2d[i, 0], embeddings_2d[i, 1]))  # Add the word label
plt.title('Word2Vec Embeddings Visualization (t-SNE)')  # Set the plot title
plt.show()  # Display the plot
```

Visualizing embedding means how close words relates,

Can we guess??

Word2Vec Visualization

words = ['king', 'queen', 'man', 'woman', 'prince', 'princess', 'cat', 'dog', 'library', 'table', 'throne', 'chair']



Step 3: Numerical Representation: Word Embedding

To Use Word2Vec: To Visualize Embeddings(using t-SNE):

List of words to visualize

['king', 'queen', 'man', 'woman', 'prince', 'princess', 'cat', 'dog', 'library', 'table', 'throne', 'chair']

Let's get this done…..

Can we guess??

Step 3: Numerical Representation: Word Embedding

Waiting!!!!!!!!!

Let's get this done.....

Can we give a trial??

Step 3: Numerical Representation: Word Embedding

List of words to visualize ['king', 'queen', 'man', 'woman', 'prince', 'princess', 'cat', 'dog', 'library', 'table', 'throne', 'chair’]

Say: Royalty: king, queen, prince, princess, throne

Gender: man, woman

Pets/Animals: cat, dog

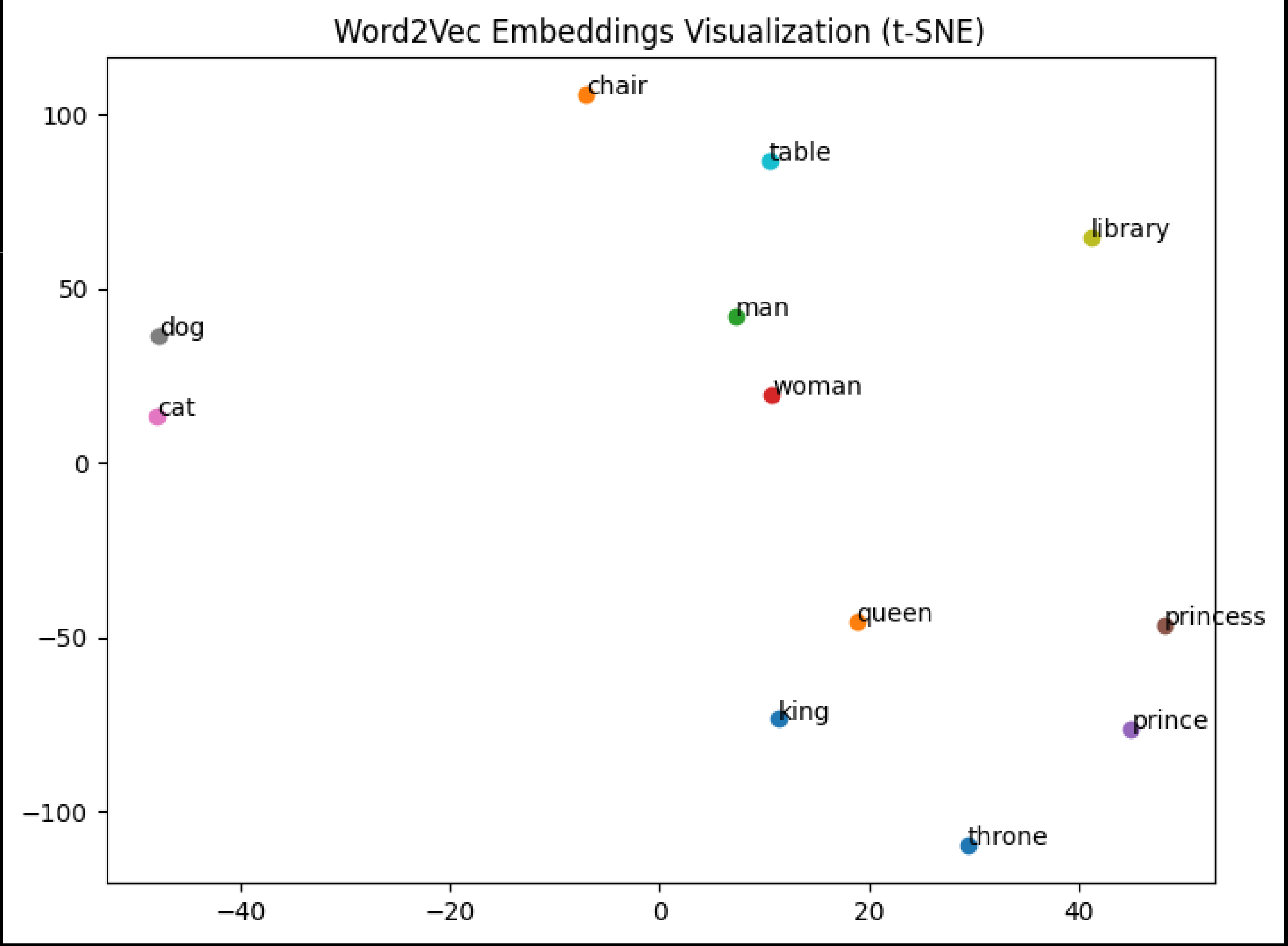
Books/Reading: library

Furniture: chair, table,

Right????

List of words to visualize ['king', 'queen', 'man', 'woman', 'prince', 'princess', 'cat', 'dog', 'library', 'table', 'throne', 'chair']

So, we have....



Interesting right!!

Step 3: Numerical Representation: Word Embedding

Using Word2Vec, Plot the Embedding od the words below.

```
words = ['king', 'queen', 'prince', 'princess', 'man', 'woman',
'cat', 'dog', 'kitten', 'puppy', 'book', 'library', 'page',
'author', 'chair', 'table', 'furniture', 'sofa']
```

Let's get this done…..

This time more accurate response??

Step 3: Numerical Representation: Word Embedding

Waiting!!!!!!!!!

Let's get this done…..

Can we give a trial??

Step 3: Numerical Representation: Word Embedding

List of words to visualize: `['king', 'queen', 'prince', 'princess', 'man', 'woman', 'cat', 'dog', 'kitten', 'puppy', 'book', 'library', 'page', 'author', 'chair', 'table', 'furniture', 'sofa']`

Say: Royalty: king, queen, prince, princess

Gender: man, woman

Pets/Animals: cat, dog, kitten, puppy

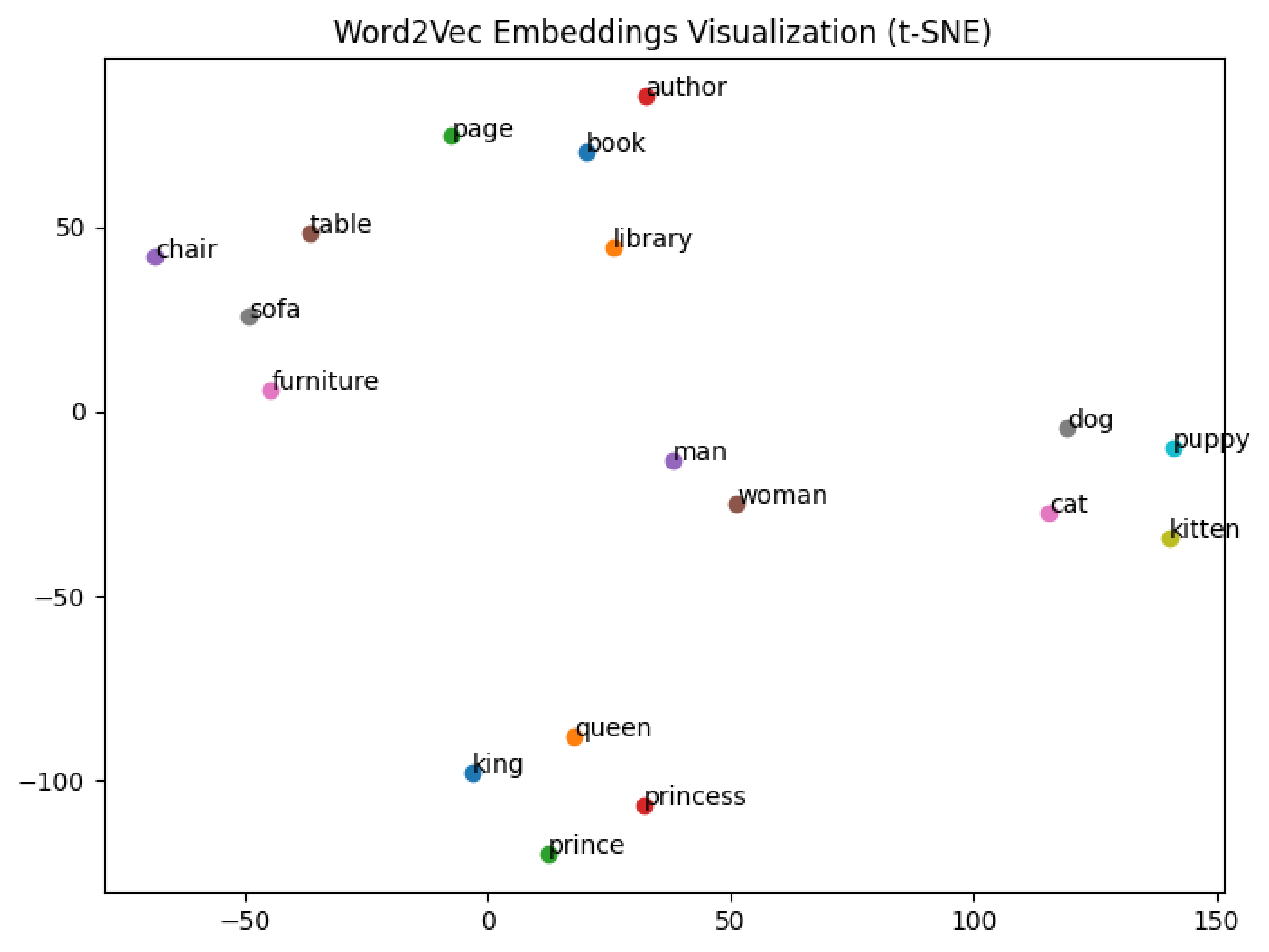
Books/Reading: book, library, page, author

Furniture: chair, table, furniture, sofa

Right????

List of words to visualize:

['king', 'queen', 'prince', 'princess', 'man', 'woman', 'cat', 'dog', 'kitten', 'puppy', 'book', 'library', 'page', 'author', 'chair', 'table', 'furniture', 'sofa']



So, we have…*interesting right!!*

Step 3: Numerical Representation: Word Embedding

GloVe developed by **Stanford researchers in 2014.**

This embedding technique is based on factorizing a matrix of word co-occurrence statistics.

Its developers have made available precomputed embeddings for millions of English tokens, obtained from Wikipedia data and Common Crawl data.

NB

GloVe == Global vectors

Step 3: Numerical Representation: Word Embedding

To use the Glove:

First, let's download the GloVe word embeddings precomputed on the 2014 English Wikipedia dataset. It's an 822 MB zip file containing 100-dimensional embedding vectors for 400,000 words (or non-word tokens).

Type the code:

!wget http:/ /nlp.stanford.edu/data/glove.6B.zip

!unzip -q glove.6B.zip

NB

GloVe

To use the Glove:

Downloading GloVe

This about 900mb and another 400mb…..

NOW LETS USE GLOVE TECHNIQUE

```
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip glove.6B.zip
```

```
--2025-02-24 14:25:18--  http://nlp.stanford.edu/data/glove.6B.zip
Resolving nlp.stanford.edu (nlp.stanford.edu)... 171.64.67.140
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://nlp.stanford.edu/data/glove.6B.zip [following]
--2025-02-24 14:25:18--  https://nlp.stanford.edu/data/glove.6B.zip
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip [following]
--2025-02-24 14:25:18--  https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)... 171.64.64.22
Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64.64.22|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 862182613 (822M) [application/zip]
Saving to: ‘glove.6B.zip’

glove.6B.zip        100%[===================>] 822.24M  5.00MB/s    in 2m 39s

2025-02-24 14:27:57 (5.19 MB/s) - ‘glove.6B.zip’ saved [862182613/862182613]

Archive:  glove.6B.zip
  inflating: glove.6B.50d.txt
  inflating: glove.6B.100d.txt
  inflating: glove.6B.200d.txt
  inflating: glove.6B.300d.txt
```

GloVe

Setup some long process which include

Preparing Training and validation sentence

Text Vectorization, embedding, and building the model.

You may need to define some functions you can simply call and pass in it's parameter/arguments to use the built GloVe Model.

NB: This code snippet by the side isn't the full code, we can check that out in a colab

```
# 5. Create the embedding matrix using the consistent vocabulary
embedding_dim = 100
vocabulary = text_vectorization.get_vocabulary()
word_index = dict(zip(vocabulary, range(len(vocabulary))))
embedding_matrix = np.zeros((max_tokens, embedding_dim))
for word, i in word_index.items():
    if i < max_tokens:
        embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector

# 6. Create the Embedding layer
embedding_layer = layers.Embedding(
    max_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,
    mask_zero=True,
)
# 7. Vectorize your training and validation data
int_train_ds = tf.data.Dataset.from_tensor_slices(
    (text_vectorization(sentences_train), labels_train)  # Use text_vectorizat
).batch(2)

int_val_ds = tf.data.Dataset.from_tensor_slices(
    (text_vectorization(sentences_val), labels_val)  # Use text_vectorization
).batch(2)
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = embedding_layer(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("glove_embeddings_sequence_model.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("glove_embeddings_sequence_model.keras")
# Now 'int_train_ds' and 'int_val_ds' have indices matching the embedding layer
```

Step 3: Numerical Representation: Word Embedding

To Use GloVe:

To check for semantic similarities:

You see king and car and very little semantic similarities.

```
# Examples of Semantic Similarity
print(f"Similarity ('king', 'queen'): {glove_similarity('king', 'queen', embeddings_index)}")
print(f"Similarity ('man', 'woman'): {glove_similarity('man', 'woman', embeddings_index)}")
print(f"Similarity ('king', 'man'): {glove_similarity('king', 'man', embeddings_index)}")
print(f"Similarity ('king', 'car'): {glove_similarity('king', 'car', embeddings_index)}")
print(f"Similarity ('cat', 'dog'): {glove_similarity('cat', 'dog', embeddings_index)}")
```

```
Similarity ('king', 'queen'): 0.7507690787315369
Similarity ('man', 'woman'): 0.8323494791984558
Similarity ('king', 'man'): 0.5118681192398071
Similarity ('king', 'car'): 0.28304237127304077
Similarity ('cat', 'dog'): 0.8798074722290039
```

Step 3: Numerical Representation: Word Embedding

To Use GloVe:

To find similar word:

```
# Examples of Finding Similar Words (Continued)
print(f"Most similar to 'cat': {find_most_similar('cat', embeddings_index)}")
print(f"Most similar to 'book': {find_most_similar('book', embeddings_index)}")
```

```
Most similar to 'cat': [('dog', 0.8798075), ('rabbit', 0.74244267), ('cats', 0.7323004), ('monkey', 0.728871), ('pet', 0.71901405)]
Most similar to 'book': [('books', 0.84764856), ('novel', 0.81811666), ('published', 0.8023924), ('story', 0.7941391), ('author', 0.
```

Dog has the most similar word

Step 3: Numerical Representation: Word Embedding

To Use GloVe:

To find similar word:

```
# Examples of Finding Similar Words (Continued)
print(f"Most similar to 'cat': {find_most_similar('cat', embeddings_index)}")
print(f"Most similar to 'book': {find_most_similar('book', embeddings_index)}")
```

```
Most similar to 'cat': [('dog', 0.8798075), ('rabbit', 0.74244267), ('cats', 0.7323004), ('monkey', 0.728871), ('pet', 0.71901405)]
Most similar to 'book': [('books', 0.84764856), ('novel', 0.81811666), ('published', 0.8023924), ('story', 0.7941391), ('author', 0.
```

Dog has the most similar word to Cat

Step 3: Numerical Representation: Word Embedding

To Use GloVe:

King exists, randomword doesn't!
That's expected.

To check if specific Word Vector exists:

```
# Examples of Checking Word Vector Existence
check_word_vector("king", embeddings_index)
check_word_vector("randomword", embeddings_index)
```

```
'king' has a GloVe vector.
'randomword' does not have a GloVe vector.
```

Step 3: Numerical Representation: Word Embedding

To Use GloVe:

To check if specific Word Vector exists:

As seen, Vector for "**queen**" exists, but no vector for the word "**anotherword**'

```
# Examples of Getting the Vector for a Word
get_word_vector("queen", embeddings_index)
get_word_vector("anotherword", embeddings_index)
```

```
Vector for 'queen': [-0.50045  -0.70826   0.55388   0.673     0.22486   0.60281  -0.26194
  0.73872  -0.65383  -0.21606  -0.33806   0.24498  -0.51497   0.8568
 -0.37199  -0.58824   0.30637  -0.30668  -0.2187    0.78369  -0.61944
 -0.54925   0.43067  -0.027348  0.97574   0.46169   0.11486  -0.99842
  1.0661   -0.20819   0.53158   0.40922   1.0406    0.24943   0.18709
  0.41528  -0.95408   0.36822  -0.37948  -0.6802   -0.14578  -0.20113
  0.17113  -0.55705   0.7191    0.070014 -0.23637   0.49534   1.1576
 -0.05078   0.25731  -0.091052  1.2663    1.1047   -0.51584  -2.0033
 -0.64821   0.16417   0.32935   0.048484  0.18997   0.66116   0.080882
  0.3364    0.22758   0.1462   -0.51005   0.63777   0.47299  -0.3282
  0.083899 -0.78547   0.099148  0.039176  0.27893   0.11747   0.57862
  0.043639 -0.15965  -0.35304  -0.048965 -0.32461   1.4981    0.58138
 -1.132    -0.60673  -0.37505  -1.1813    0.80117  -0.50014  -0.16574
 -0.70584   0.43012   0.51051  -0.8033   -0.66572  -0.63717  -0.36032
  0.13347  -0.56075  ]
'anotherword' has no vector.
```

Again, the vectors are in numbers just like it was when we used word2vec

Step 3: Numerical Representation: Word Embedding

To Use GloVe:

To visualize Embeddings (using t-SNE):

The word_to_visualize variable stores the list with the words

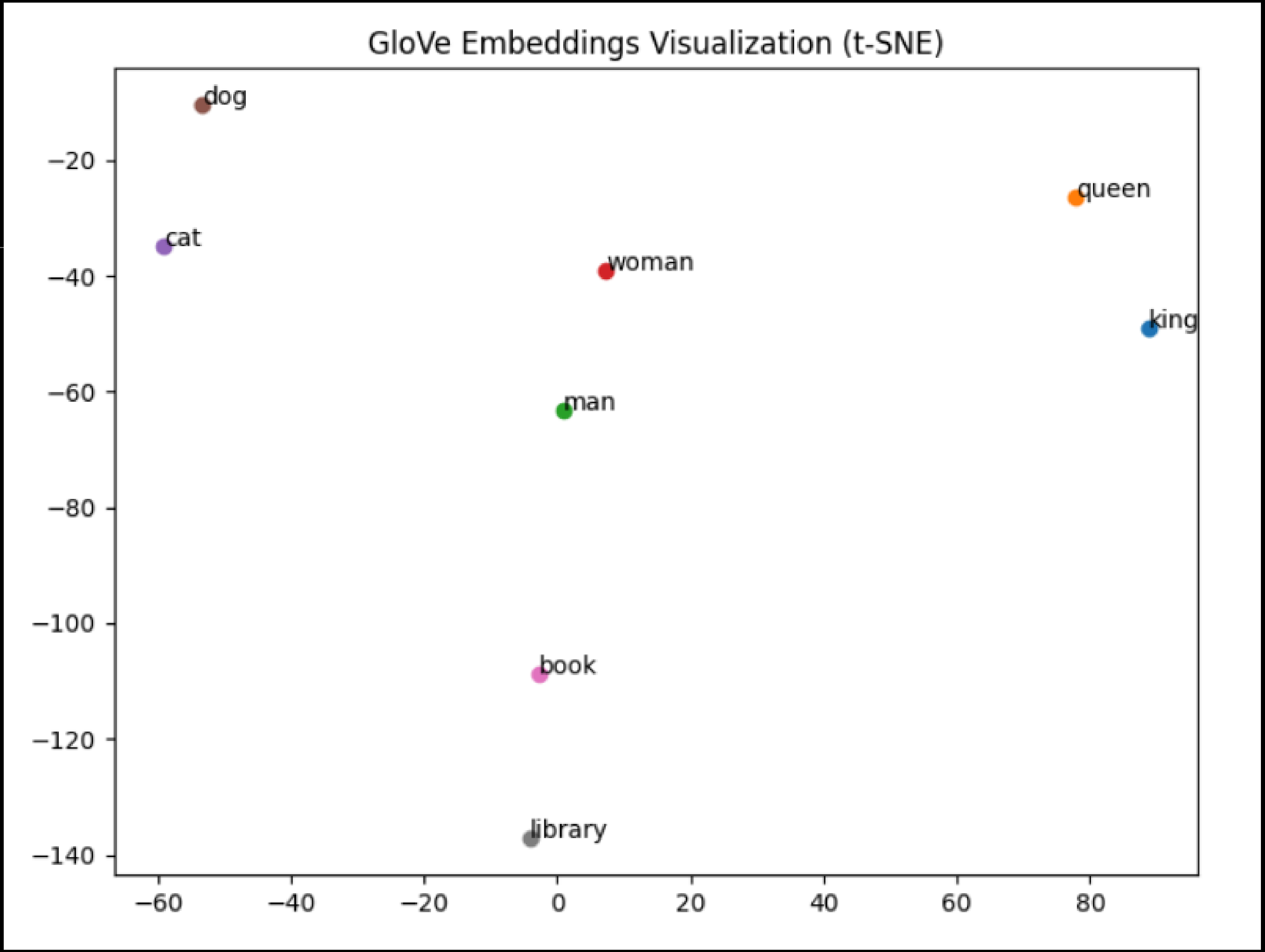
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

def visualize_embeddings(words, embeddings_index):
    """Visualizes GloVe embeddings using t-SNE."""
    embeddings = [get_glove_vector(word, embeddings_index) for word in words if get_glove_vector(word, embeddings_index) is not None]
    words_filtered = [word for word in words if get_glove_vector(word, embeddings_index) is not None]

    if not embeddings:
        print("No embeddings to visualize.")
        return

    # Convert the list of embeddings to a NumPy array
    embeddings = np.array(embeddings)

    # Lower the perplexity to be significantly less than the number of samples
    tsne = TSNE(n_components=2, perplexity=3, random_state=42)  # Reduced perplexity to 3
    embeddings_2d = tsne.fit_transform(embeddings)

    plt.figure(figsize=(8, 6))
    for i, word in enumerate(words_filtered):
        plt.scatter(embeddings_2d[i, 0], embeddings_2d[i, 1])
        plt.annotate(word, (embeddings_2d[i, 0], embeddings_2d[i, 1]))
    plt.title('GloVe Embeddings Visualization (t-SNE)')
    plt.show()

words_to_visualize = ['king', 'queen', 'man', 'woman', 'cat', 'dog', 'book', 'library']
visualize_embeddings(words_to_visualize, embeddings_index)
```

GloVe

To Use GloVe:

To visualize Embeddings (using t-SNE):

words_to_visualize = ['king', 'queen', 'man', 'woman', 'cat', 'dog', 'book', 'library']



Step 3: Numerical Representation: Word Embedding

To Use GloVe:

To visualize Embeddings (using t-SNE):

words_to_visualize = ['king', 'queen', 'prince', 'princess', 'man', 'woman’, 'cat', 'dog', 'kitten', 'puppy’, 'book', 'library', 'page', 'author’, 'chair', 'table', 'furniture', 'sofa']

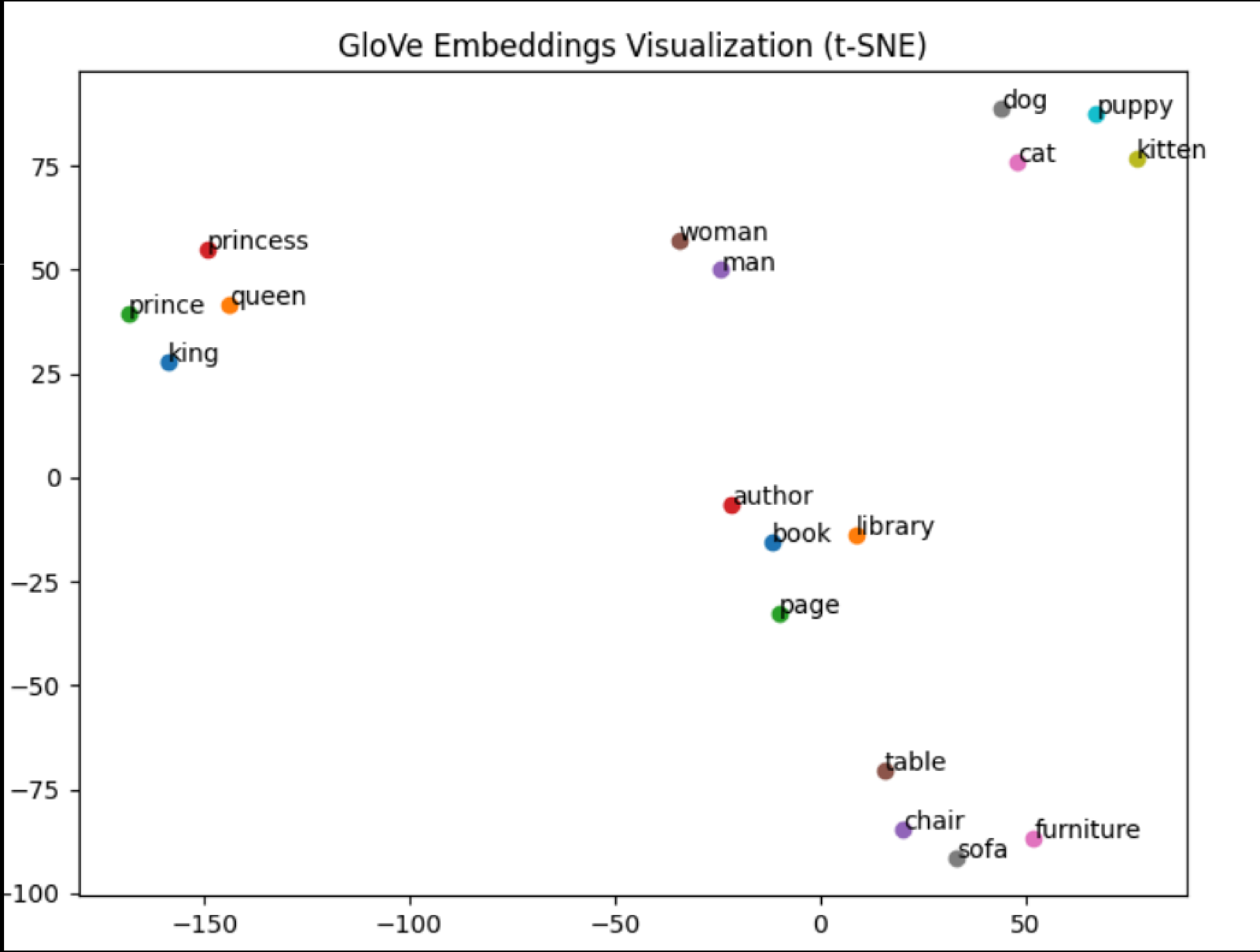
```
words_to_visualize = ['king', 'queen', 'prince', 'princess', 'man', 'woman',
          'cat', 'dog', 'kitten', 'puppy',
          'book', 'library', 'page', 'author',
          'chair', 'table', 'furniture', 'sofa']

visualize_embeddings(words_to_visualize, embeddings_index)
```

GloVe

To Use GloVe:

To visualize Embeddings (using t-SNE):

words_to_visualize = ['king', 'queen', 'prince', 'princess', 'man', 'woman', 'cat', 'dog', 'kitten', 'puppy', 'book', 'library', 'page', 'author', 'chair', 'table', 'furniture', 'sofa']



What is t-SNE (pronounced tee-snee)

t-distributed stochastic neighbor embedding, t-SNE (pronounced tee-snee), simply used for plotting word vector.

It was developed by **Laurens van der Maaten** in collaboration with **Geoff Hinton.**

t-SNE allow us to use the **dimensionality reduction** to approximately map the locations of words from high dimensional word-vector space down to two or three dimensions.

```
# t-SNE (t-Distributed Stochastic Neighbor Embedding) is a technique for visualizing high-dimensional
# Set perplexity to a value less than the number of samples (8 in this case)
tsne = TSNE(n_components=2, perplexity=5, random_state=42)  # Reduce to 2 dimensions, perplexity=5
embeddings_2d = tsne.fit_transform(embeddings)
```

Word2Vec vs GloVe

FEATURE	WORD2VEC	GLOVE
Learning Method	"Guess the Neighbors" (Predictive)	"Count the Co-occurrences" (Statistical)
How It Works	Tries to predict nearby words in a sentence.	Analyzes how often words appear together in the whole text.
Focus	Local context (words close together)	Global statistics (overall word relationships)
Analogy	"Gossip Network" (learns from nearby "friends")	"Census Taker" (counts co-occurrences like neighbors)
Strength	Captures subtle semantic relationships.	Captures general word similarity and relatedness.
Example Use	Understanding "run" in different contexts (exercise vs. program).	Knowing "cat" and "dog" are generally similar animals.
Key Idea	Learns by predicting what words go together in a sentence.	Learns by counting how often words appear together in a large text.
Think of It As	Learning from the immediate surrounding words.	Learning from the entire collection of words.

Numerical Representation: Word Embedding

FastText: Developed by facebook’s AI Research lab (meta) in 2015

- It’s the contemporary leading alternative to both word2vec and GloVe, Designed to address their limitations, particularly in handling rare words and morphologically rich languages (as in word2vec).
- FastText can understand that 'unbreakable' is related to 'break' even if it hasn't seen 'unbreakable' before.
- It considers subword information, making it effective for morphologically rich languages and handling out-of-vocabulary words.
- This enables fastText to work around some of the issues related to rare words and out-of-vocabulary words addressed in the preprocessing section at the outset of this chapter.

Step 3: Numerical Representation: Word Embedding

BERT (bi-directional encoder representations from transformers):

- BERT was developed by Google and introduced in 2018.
- It revolutionized NLP with its ability to understand context deeply.
- It's based on the **Transformer** architecture, which uses **attention mechanisms.** What's Attention Mechanism??? What are transformers???
- Unlike Word2Vec and GloVe, which are unidirectional or consider context in a limited way, BERT is trained bidirectionally.
- This means it considers the context of a word from both the left and right sides, leading to a much richer understanding.

Step 3: Numerical Representation: Word Embedding

Because BERT is Transformer Based, we need to understand a new topic called **Attention Mechanism** and **Transformers** first.

For now, understand that, considering Architecture, Word2Vec and GloVe are simpler models. BERT is a deep **Transformer-based model.**

Also, Unlike Word2Vec and GloVe, which are unidirectional or consider context in a limited way, **BERT** is trained **bidirectionally.**

This means it considers the context of a word from both the left and right sides, leading to a much richer understanding.

Yaa, Let's wait till then,

Meanwhile, before then, DYOR

Pre-Trained Model for Word Embedding history

Recap:

Word2Vec = 2013

Glove = 2014

FastText = 2015

BERT = 2018

What's Next

Maybe I don't know also!!!

If Yes,

Let's move back a bit!!

Understand Human Language

There are arguably 3 steps to help computer understand human texts,

1. Text Preprocessing
2. Tokenization
3. Numerical Representation (Word Embedding)

Now, Lets discuss each further!!

Remember this? We've explained the 3 steps..... Whats next in NLP? Can we now build our ChatGPT?

What's Next 😀😀

I remembered something

How about

Transformers ??

and

Attention ??

Transformers: Attention is All You Need

A transformer is a type of artificial intelligence model that learns to understand and generate human-like text by analyzing patterns in large amounts of text data

Transformers are a revolutionary type of **neural network** that rely entirely on something called **attention.**

They process all the words at the same time, making them much faster than **RNNs (Recurrent Neural Networks).**

We Move!!!!

What do you notice 😃 😃

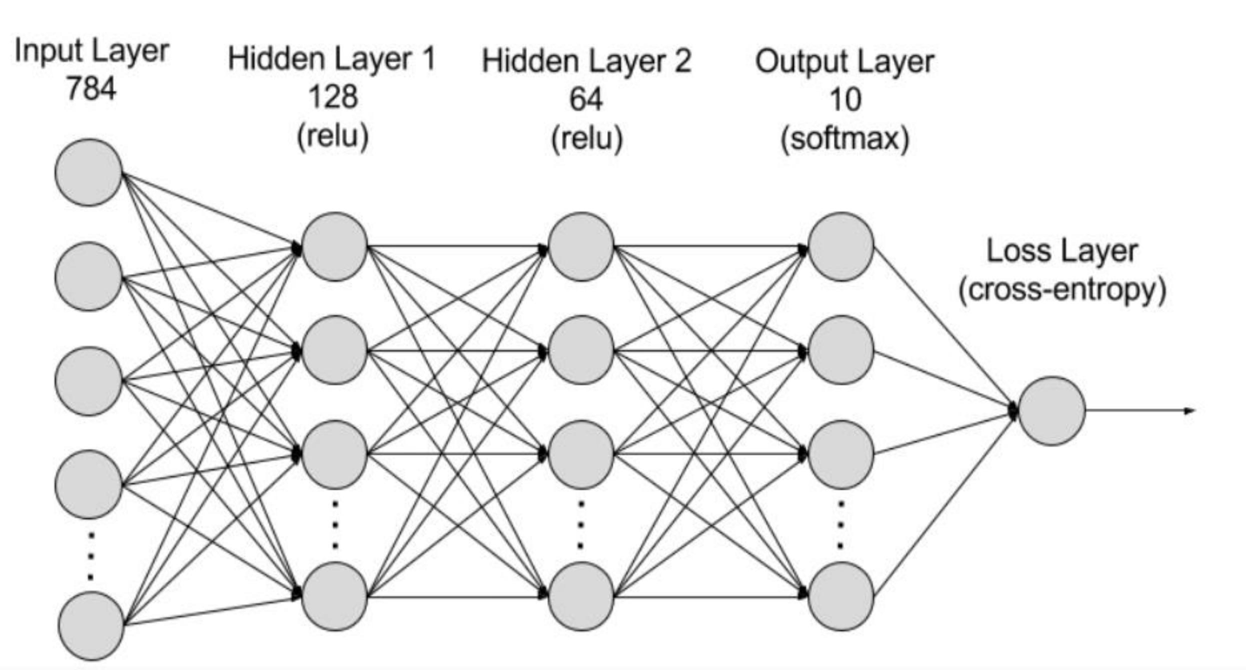
We have mentioned RNNs (Recurrent Neural Network) severally with minimal explanation

RNN, let's discuss briefly

Recurrent Neural Networks (RNNs)

Recurrent neural networks (RNNs) is a family of deep learning models that can handle long sequences of data.

The family include specialized layer types like **long short-term memory units (LSTMs)** and **gated recurrent units (GRUs).**

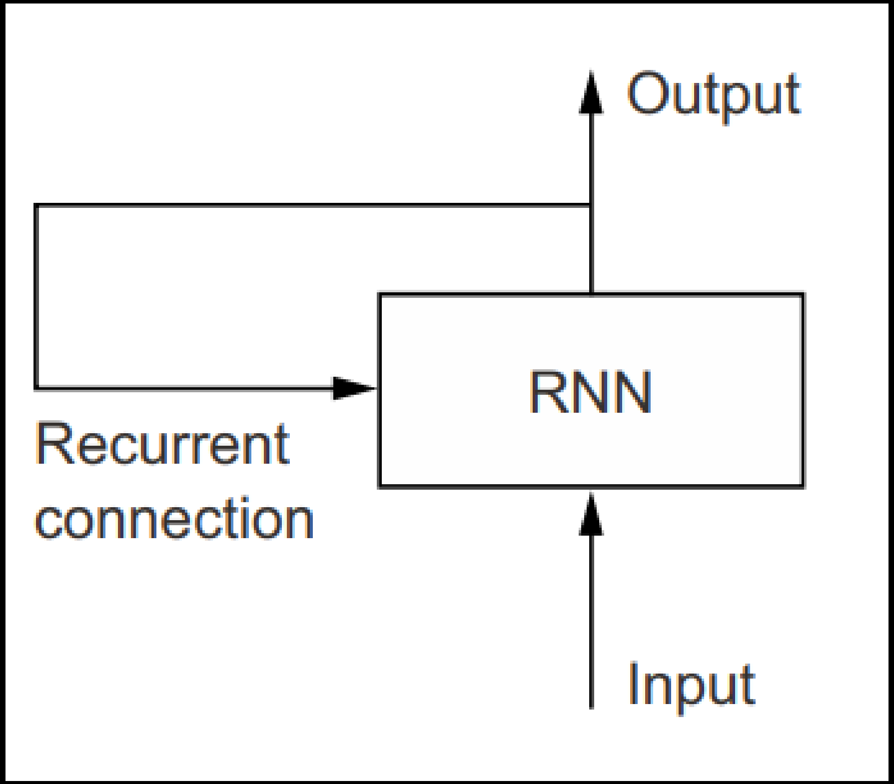


Sequential data is data, such as words, sentences, or time-series data, where sequential components interrelate based on complex semantics and syntax rules.

Recurrent Neural Networks (RNNs)

In effect, an RNN is a type of neural network that has an internal loop, hence, **Recurrent Neural Network**.

In simple terms; an RNN is a **for loop** that reuses quantities computed during the **previous iteration of the loop**.



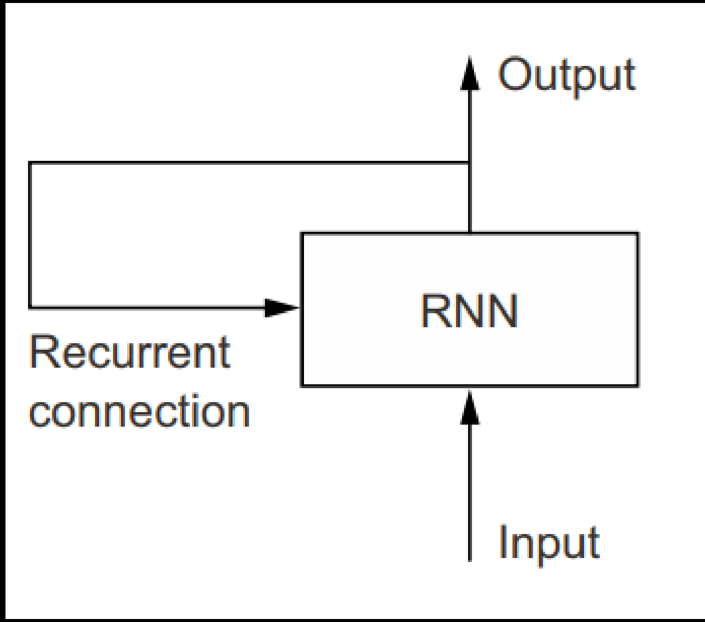
A recurrent network: a network with a loop

Recurrent Neural Networks (RNNs)

RNNs maintain a **hidden state** that acts as a form of memory.

At each time step, the hidden state is updated based on the current input and the previous hidden state.

This allows the network to capture dependencies and patterns across the sequence, though forgetting captured information from the very beginning of a long Sentence.



Types of RNN Architectures:

Simple RNNs (Vanilla RNNs): The basic form of RNNs. Prone to vanishing and exploding gradient problems.

Long Short-Term Memory (LSTM) Networks: Designed to address the vanishing gradient problem. Use **"gates"** to control the flow of information in the hidden state. Better at capturing long-range dependencies.

Gated Recurrent Unit (GRU) Networks: A simplified version of LSTMs. Also uses gates to control information flow. Often performs comparably to LSTMs with fewer parameters.

Bidirectional RNNs: Process sequences in both forward and backward directions. Useful for tasks where context from both past and future is important.

Deep RNNs: Stack multiple RNN layers on top of each other. Allow the network to learn more complex representations.

Bottleneck of RNNs

RNNs have trouble dealing with very long sequences, as they tend to progressively forget about the past.

By the time you've reached the 100th token in either sequence, little information remains about the start of the sequence.

That means RNN-based models can't hold onto long-term context, which can be essential for translating long documents.

QUESTIONS AND ANSWER

The END Day 3

DSN LEKKI-AJAH

BY ABEREJO HABEEBLAH O.

X: @HABEREJO

Next Class schedule:

Tuesday, 25th March, by 5:00pm.