Application of Deep Learning to Text and Images

Module 2, Lab 3: GloVe Word Vectors

This notebook supports the topics presented on on the **Word Embeddings** lecture.

In this lab you will learn how to use word embeddings. Word embeddings, or word vectors, are a way of representing words as numeric vectors in a high-dimensional space. These embeddings capture the meaning of the words, the relationships between them, and can be used as inputs to machine learning models for a variety of natural language processing tasks.

The term **Word vectors** refers to a family of related techniques, first gaining popularity via `Word2Vec` which associates an n-dimensional vector to every word in the target language.

- **Note**: Normally n is in the range of 50 to 500. In this lab, you will set it to 50

You will learn:

- What GloVe word vectors are
- How to load GloVe word vectors
- How to use GloVe to produce word vectors
- What cosine Similarity is
- How to use cosine similarity to compare words

You will be presented with two kinds of exercises throughout the notebook: activities and challenges.

 **Activity**	 **Challenge**
No coding is needed for an activity. You try to understand a concept, answer questions, or run a code cell.	Challenges are where you can practice your coding skills.

Index

1. GloVe Word Vectors
2. Cosine Similarity

First, install the latest versions of the libraries.

In [1]:
```
# installing libraries
!pip install -U -q -r requirements.txt
```

```
ERROR: pip's dependency resolver does not currently take into account all t
he packages that are installed. This behaviour is the source of the followi
ng dependency conflicts.
autovizwidget 0.21.0 requires pandas<2.0.0,>=0.20.1, but you have pandas 2.
0.3 which is incompatible.
hdijupyterutils 0.21.0 requires pandas<2.0.0,>=0.17.1, but you have pandas
2.0.3 which is incompatible.
sparkmagic 0.21.0 requires pandas<2.0.0,>=0.17.1, but you have pandas 2.0.3
which is incompatible.
```

In [2]:
```
from torchtext.vocab import GloVe

#from torchtext.vocab import GloVe
GloVe.url['6B'] = 'https://huggingface.co/stanfordnlp/glove/resolve/main/glc

from sklearn.decomposition import PCA
from sklearn.metrics.pairwise import cosine_similarity

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

```
Matplotlib is building the font cache; this may take a moment.
```

GloVe Word Vectors

You learned about **Word2Vec** and **FastText** as word embedding techniques. Now you will use a set of pre-trained word embeddings. Pre-trained embeddings are created by someone else who took the time and computational power to train. This reduces your cost by not having to train the model yourself. One popular word embedding is **GloVe** embeddings. GloVe is a variation of a Word2Vec model. To learn more about GloVe, read the Project GloVe website.

In this exercise, you will discover relationships between word vectors using the GloVe embeddings.

You can easily import GloVe embeddings from the Torchtext library. Here, you will get vectors with 50 dimensions.

The `name` parameter refers to the particular pre-trained model that should be loaded:

- Wikipedia 2014 + Gigaword 5

    - 6B tokens, 400K vocab, uncased, 50d, 100d, 200d, & 300d vectors, 822 MB download: `"6B"`
    - This is the model that you will load.
- Common Crawl
    - 42B tokens, 1.9M vocab, uncased, 300d vectors, 1.75 GB download: `"42B"`
- Common Crawl
    - 840B tokens, 2.2M vocab, cased, 300d vectors, 2.03 GB download: `"840B"`
- Etc
    - See documentation in Stanford link above

Try it Yourself!

Run the cell below to load the GloVe embedding model and select the dimension.

In [3]:
```
# Load the model. You can change dim to 50, 100, 300
glove = GloVe(name="6B", dim=50)
```

```
.vector_cache/glove.6B.zip: 862MB [00:04, 195MB/s]
100%|| 400000/400001 [00:14<00:00, 27702.91it/s]
```

Now that the data is loaded, you can access it and print example word embeddings.

In [4]:
```
print(f"cat -> {glove['cat']}\n")
```

```
cat -> tensor([ 0.4528, -0.5011, -0.5371, -0.0157,  0.2219,  0.5460, -0.6730, -0.6891,
         0.6349, -0.1973,  0.3368,  0.7735,  0.9009,  0.3849,  0.3837,  0.2657,
        -0.0806,  0.6109, -1.2894, -0.2231, -0.6158,  0.2170,  0.3561,  0.4450,
         0.6089, -1.1633, -1.1579,  0.3612,  0.1047, -0.7832,  1.4352,  0.1863,
        -0.2611,  0.8328, -0.2312,  0.3248,  0.1449, -0.4455,  0.3350, -0.9595,
        -0.0975,  0.4814, -0.4335,  0.6945,  0.9104, -0.2817,  0.4164, -1.2609,
         0.7128,  0.2378])
```

What do these numbers mean?

You might notice that the tensor has 50 values in it. This is related to the dimension flag (`dim=50`) you set when you loaded the GloVe model. You can generate word embeddings for several words and use them to determine how closely related words are. This is a task that machine learning is really good at.

Try it Yourself!

 Challenge

In the code block below, generate word embeddings for the words "computer" and "human" using pre-trained GloVe embedding.

In [12]:
```
############### CODE HERE ###############
import torch
from torchtext.vocab import GloVe

# Load the pre-trained GloVe model (50-dimensional vectors)
glove = GloVe(name="6B", dim=50)

# Get word embeddings for "computer" and "human"
computer_embedding = glove["computer"]
human_embedding = glove["human"]

# Print the word embeddings
print(f"Embedding for 'computer':\n{computer_embedding}\n")
print(f"Embedding for 'human':\n{human_embedding}\n")


############### END OF CODE ##################
```

```
Embedding for 'computer':
tensor([ 0.0791, -0.8150,  1.7901,  0.9165,  0.1080, -0.5563, -0.8443, -1.4
951,
         0.1342,  0.6363,  0.3515,  0.2581, -0.5503,  0.5106,  0.3741,  0.1
209,
        -1.6166,  0.8365,  0.1420, -0.5235,  0.7345,  0.1221, -0.4908,  0.3
253,
         0.4531, -1.5850, -0.6385, -1.0053,  0.1045, -0.4298,  3.1810, -0.6
219,
         0.1682, -1.0139,  0.0641,  0.5784, -0.4556,  0.7378,  0.3720, -0.5
772,
         0.6644,  0.0551,  0.0379,  1.3275,  0.3099,  0.5070,  1.2357,  0.1
274,
        -0.1143,  0.2071])

Embedding for 'human':
tensor([ 0.6185,  0.1191, -0.4679,  0.3137,  1.0334,  0.9596,  0.8780, -1.0
346,
         1.6322,  0.2935,  0.8084, -0.0589,  0.0213,  0.4099,  0.5444, -0.3
331,
         0.5371, -0.3582,  0.2937,  0.0902, -0.9205,  0.6939,  0.3910, -0.6
439,
         0.7783, -1.7215, -0.4839, -0.5033, -0.2251,  0.0992,  3.2095, -0.3
155,
        -0.7175, -1.6752, -1.3537,  0.1520,  0.0546, -0.1633, -0.0280,  0.3
917,
        -0.5501, -0.0792,  0.6339,  0.5145,  0.7012,  0.2764, -0.5344,  0.0
648,
        -0.2197, -0.5205])
```

Cosine Similarity

You learned about cosine similarity in class, now let's look at an example. Use the cosine_similarity() function from scikit-learn to easily calculate cosine similarity between word vectors.

Try it Yourself!

 Activity

Run the cell below to calculate cosine similarity between word vectors.

In [ ]:
```
# define the similarity between two words
def similarity(w1, w2):
    return cosine_similarity([glove[w1].tolist()], [glove[w2].tolist()])


# Say if w1 is closer to w2 than w3
def simCompare(w1, w2, w3):
```

```
    s1 = similarity(w1, w2)
    s2 = similarity(w1, w3)
    if s1 > s2:
        print(f"'{w1}'\tis closer to\t'{w2}'\tthan\t'{w3}'\n")
    else:
        print(f"'{w1}'\tis closer to\t'{w3}'\tthan\t'{w2}'\n")
```

In [ ]:
```
simCompare("actor", "pen", "film")
simCompare("cat", "dog", "sea")
```

Try it Yourself!

 Challenge

Write code to determine if "car" is closer to "truck" than "bike".

In [13]:
```
############### CODE HERE ###############

from sklearn.metrics.pairwise import cosine_similarity

# Define the similarity function between two words
def similarity(w1, w2):
    return cosine_similarity([glove[w1].tolist()], [glove[w2].tolist()])

# Compare if w1 is closer to w2 or w3
def simCompare(w1, w2, w3):
    s1 = similarity(w1, w2)
    s2 = similarity(w1, w3)

    if s1 > s2:
        print(f"'{w1}'\tis closer to\t'{w2}'\tthan\t'{w3}'\n")
    else:
        print(f"'{w1}'\tis closer to\t'{w3}'\tthan\t'{w2}'\n")

# Challenge: Check if "car" is closer to "truck" than "bike"
simCompare("car", "truck", "bike")


############### END OF CODE ##################
```

```
'car'   is closer to    'truck' than    'bike'
```

Conclusion

You have now seen how to use word embeddings and determine relationships between word vectors using the GloVe embeddings.

Word embeddings capture semantic meaning – The GloVe model effectively represents words as numerical vectors while preserving relationships between their meanings.

Cosine similarity measures word closeness – Using cosine similarity, we can determine how semantically related two words are based on their vector representations.

Similar words have higher cosine similarity – Words that are closely related in meaning, like "car" and "truck," will have a higher similarity score compared to unrelated words like "car" and "bike."

Word vectors encode relationships – The way word embeddings are structured allows them to capture relationships such as synonyms, antonyms, and domain-specific similarities.

GloVe embeddings generalize well – Since GloVe is trained on large text corpora (e.g., Wikipedia, Common Crawl), its pre-trained embeddings work well across different NLP tasks.

Vector dimensions impact performance – Using higher-dimensional embeddings (e.g., 300D instead of 50D) can improve accuracy but requires more computational resources.

Word relationships are context-dependent – Although word embeddings capture meaning, their similarity scores can vary based on the dataset they were trained on.

Word analogies can be solved using embeddings – GloVe embeddings allow for operations like "king - man + woman ≈ queen", demonstrating their ability to capture complex relationships.

Pre-trained embeddings save training time – Instead of training word embeddings from scratch, pre-trained models like GloVe provide immediate access to high-quality word vectors.

NLP models benefit from embeddings – Machine learning models for text classification, sentiment analysis, and chatbot development perform better when using word embeddings instead of raw text.

Next Lab: Word Embeddings

In the next lab of this module you will learn how to build a recurrent neural network (RNN) with PyTorch. It will also show you how to implement a simple RNN-based model for natural language processing.

End Of Lab

In [ ]: