Mini-project 3: NLP an Neural Networks

Author: Mckayla Ashley

Reference

Precept_word2vec_nn.ipynb

Word2Vec: https://radimrehurek.com/gensim/models/word2vec.html

https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html#sphx-glr-auto-examples-tutorials-run-word2vec-py
https://rare-technologies.com/word2vec-tutorial/#app

LDA: https://radimrehurek.com/gensim/auto_examples/tutorials/run_lda.html

NN: https://keras.io/guides/training_with_built_in_methods/

In [3]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import gensim.downloader
```

1. NLP and Word2Vec

a. Please describe what is meant by a "vector embedding" of words in Word2Vec

A vector embedding of words in Word2Vec means words are represented numerically as vectors in n-dimensional space, where n is the chosen embedding size of the vector. A higher dimension can tell us more about the words and complex relationships, whereas a lower dimension is more compact.

Perceptrons, used for making binary decisions, take in numerical inputs, multiplies them by corresponding weights, adds them up, and returns a prediction of either 1 or 0. Then, it adjusts the weights based on whether the prediction was correct or not. Word2Vec often uses multiple perceptrons stacked together, a neural network, to predict words and learn word relationships. Word2Vec turns words into vectors by looking at how words often show up in sentences, and quantifying their similarities through vector representation. The "nearby," or most similar vectors, are computed using cosine similarity-- where vectors that share the smallest angle are most similar.

b. Please use the pre-trained gensim 'glove-wiki-gigaword-50’ Word2Vec model to determine reasonable synonyms for the following words:

In [5]:
```
## Download the "glove-wiki-gigaword-50" embeddings
glove_vectors = gensim.downloader.load('glove-wiki-gigaword-50')
```

i. Tiger

In [8]:
```
glove_vectors.most_similar('tiger', topn=5)
```

Out[8]:
```
[('tigers', 0.723923921585083),
 ('woods', 0.6852725148200989),
 ('warrior', 0.6822083592414856),
 ('ltte', 0.6664599776268005),
 ('wild', 0.6495701670646667)]
```

The top 5 most similar words to "Tiger" are "tigers", "woods", "warrior", "ltte", and "wild". It is reasonable to consider "tigers" and "woods" as synonyms based on the model. However, the term "woods" being the second-most similar is likely due to the professional golfer Tiger Woods.

ii. Awesome

In [9]:
```
glove_vectors.most_similar('awesome', topn=5)
```

Out[9]:
```
[('unbelievable', 0.8638607859611511),
 ('amazing', 0.8620665669441223),
 ('incredible', 0.8470884561538696),
 ('fantastic', 0.8059698343276978),
 ('marvelous', 0.7899898886680603)]
```

The top five words most similar to "awesome" are "unbelievable", "amazing", "incredible", "fantastic", and "marvelous". These are reasonable synonyms as they have the highest cosine similarity and they are all adjectives.

iv. Data

In [10]:
```
glove_vectors.most_similar('data', topn=5)
```
Out[10]:
```
[('information', 0.8329989314079285),
 ('tracking', 0.81246018409729),
 ('database', 0.8122304677963257),
 ('analysis', 0.7966611981391907),
 ('applications', 0.7923666834831238)]
```
The top three words most similar to "data" are "information", "tracking", and "database". "Information" is the most reasonable synoynm.

iii. Song

In [11]:
```
glove_vectors.most_similar('song', topn=5)
```
Out[11]:
```
[('album', 0.9297007918357849),
 ('songs', 0.9009864330291748),
 ('soundtrack', 0.8414769172668457),
 ('albums', 0.8228148221969604),
 ('pop', 0.8219154477119446)]
```
The top three most similar words to "song" are "album", "songs", and "soundtrack". These are reasonable synonyms and all make sense together, as multiple songs make up an album, songs is the plural form, and soundtrack is an album for a movie.

c. Please use the pre-trained gensim ‘glove-wiki-gigawmodels to determine reasonable answers for the following analogies:

i. puppy : kitten :: dog : ?

The top answer is "cat".

In [25]:
```
## Compute an analogy:   puppy : kitten :: dog : ?

## "(dog - puppy) + kitten"

result = glove_vectors.most_similar(negative=['puppy'], positive=['kitten', 'dog'], topn=1)

print(f"The analogy is puppy : kitten :: dog : {result[0][0]}" )
```
```
The analogy is puppy : kitten :: dog : cat
```

ii. freshman: sophomore :: junior: ?

The top answer is "basketball".

In [26]:
```
## Compute an analogy:   freshman: sophomore :: junior: ?

## "(junior - freshman) + sophomore"

result = glove_vectors.most_similar(negative=['freshman'], positive=['sophomore', 'junior'], topn=1)

print(f"The analogy is freshman: sophomore :: junior: {result[0][0]}" )
```
```
The analogy is freshman: sophomore :: junior: basketball
```

iii. brother : sister :: grandson : ?

The top answer is "granddaughter".

In [27]:
```
## Compute an analogy:   brother : sister :: grandson : ?

## "(grandson - brother) + sister"

result = glove_vectors.most_similar(negative=['brother'], positive=['sister', 'grandson'], topn=1)

print(f"The analogy is brother : sister :: grandson : {result[0][0]}" )
```
```
The analogy is brother : sister :: grandson : granddaughter
```

1. NLP and Topic Modelling

a. Please prepare the built-in "fake-news" corpus of text.

In [95]:
```
import gensim.downloader as api

corpus_data = api.load('fake-news')
docs = [x["text"] for x in corpus_data]
```

b. Please use Gensim to preprocess these documents by tokenizing and lemmatizing them and removing other small text/strings that you decide are not meaningful for NLP, as well as rare and scarce words.

We remove rare and scarce words in Part C using the dictionary.

In [96]:
```
# Tokenize the documents.
from nltk.tokenize import RegexpTokenizer
from nltk.stem.wordnet import WordNetLemmatizer
from gensim.parsing.preprocessing import STOPWORDS

# Split the documents into tokens.
tokenizer = RegexpTokenizer(r'\w+')
for idx in range(len(docs)):
    docs[idx] = docs[idx].lower()  # Convert to lowercase.
    docs[idx] = tokenizer.tokenize(docs[idx])  # Split into words.

# Remove numbers, but not words that contain numbers.
docs = [[token for token in doc if not token.isnumeric()] for doc in docs]

# Remove words that are only one character.
docs = [[token for token in doc if len(token) > 2] for doc in docs]

# Remove stopwords
docs = [[token for token in doc if token not in STOPWORDS] for doc in docs]

# Lemmatize the documents
lemmatizer = WordNetLemmatizer()
docs = [[lemmatizer.lemmatize(token) for token in doc] for doc in docs]
```

c. Please use the Gensim Dictionary and dictionary.doc2bow to create a dictionary and a bag of words representation of your tokenized corpus.

In [97]:
```
# Remove rare and common tokens.
from gensim.corpora import Dictionary

# Create a dictionary representation of the documents.
dictionary = Dictionary(docs)

# Filter out words that occur less than 10 documents, or more than 60% of the documents.
dictionary.filter_extremes(no_below=10, no_above=0.6)

# Bag-of-words representation of the documents.
corpus = [dictionary.doc2bow(doc) for doc in docs]
```
In [7]:
```
# Check whether we capture meaningful words
# print(dictionary.token2id)
```

d. Please use the Gensim LDAModel to perform topic modelling of the corpus into 3 topics.

In [99]:

```
from gensim import models

lda = models.LdaModel(
    corpus = corpus,
    id2word = dictionary,
    num_topics = 3,
    random_state = 10,
    passes = 15,
    alpha='auto'
)
```

e. For each topic show the main words and use these to give a rough name to each topic.

Topic 1 is Presidential Election Campaigns, based on the words "clinton", "trump", "hillary", and "election". Topic 2 is U.S Government and Political Affairs, based on the words "state", "war", "american", and "government". Topic 3 is Society and Global Perspectives, based on the words "people", "time", "year", "like". and "world".

In [102]:
```
for idx, topic in lda.print_topics(num_topics=3, num_words=5):
    print(f"Topic {idx + 1}: {topic}")
```
```
Topic 1: 0.027*"clinton" + 0.018*"trump" + 0.016*"hillary" + 0.011*"election" + 0.010*"email"
Topic 2: 0.008*"state" + 0.007*"war" + 0.006*"american" + 0.006*"government" + 0.006*"trump"
Topic 3: 0.007*"people" + 0.006*"time" + 0.005*"year" + 0.005*"like" + 0.003*"world"
```

3. Data Classification

a. Please take the MNIST training dataset and split off the last 10,000 images as a validation data set. Then assign the other training images to a new training set.

In [83]:
```
## Import Keras and Tensorflow
import numpy as np
import tensorflow as tf
print("tensorFlow version:", tf.__version__)
print("keras version:", keras.__version__)

from tensorflow import keras
from keras import layers
```
```
tensorFlow version: 2.17.0
keras version: 3.4.1
```
In [104]:
```
## Load the builtin MNIST data
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Reserve 10,000 samples for validation
x_val = x_train[-10000:]
y_val = y_train[-10000:]
x_train = x_train[:-10000]
y_train = y_train[:-10000]
```

b. Please create a Neural Network with three dense hidden layers each having 32 nodes, and train it to classify the MNIST Data set over five epochs. For this please use your new training set and validation set respectively for model training and model performance reporting

In [105]:

```
## Defines the Neural Network model
inputs = keras.Input(shape=(784,))
x = layers.Dense(32, activation="relu")(inputs)
x = layers.Dense(32, activation="relu")(x)
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(10, activation="softmax")(x)

model = keras.Model(inputs=inputs, outputs=outputs)

# Preprocess the data (these are NumPy arrays)
x_train = x_train.reshape(-1, 784).astype("float32") / 255
x_test = x_test.reshape(-1, 784).astype("float32") / 255
x_val = x_val.reshape(-1, 784).astype("float32") / 255

## Prepare the model with additional information about how to train it!  (i.e. "compile" it)
model.compile(
    optimizer='adam', #keras.optimizers.RMSprop(),  Optimizer Adam
    # Loss function to minimize
    loss=keras.losses.SparseCategoricalCrossentropy(),
    # List of metrics to monitor
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
)
```

In [106]:

```
## Fit our model with the initial training data, and check it with the "validation" training data!
print("Fit model on training data")
history = model.fit(
    x_train,
    y_train,
    batch_size=64,
    epochs=5, # 5 epochs
    # We pass some validation for
    # monitoring validation loss and metrics
    # at the end of each epoch
    validation_data=(x_val, y_val),
)
```

```
Fit model on training data
Epoch 1/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 2s 987us/step - loss: 0.7972 - sparse_categorical_accuracy: 0.7467 - val_loss: 0.2117 - val_sparse_categorica
l_accuracy: 0.9396
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 901us/step - loss: 0.2174 - sparse_categorical_accuracy: 0.9360 - val_loss: 0.1689 - val_sparse_categorica
l_accuracy: 0.9514
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 851us/step - loss: 0.1690 - sparse_categorical_accuracy: 0.9492 - val_loss: 0.1562 - val_sparse_categorica
l_accuracy: 0.9554
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.1448 - sparse_categorical_accuracy: 0.9567 - val_loss: 0.1386 - val_sparse_categorical_
accuracy: 0.9601
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.1216 - sparse_categorical_accuracy: 0.9634 - val_loss: 0.1282 - val_sparse_categorical_
accuracy: 0.9625
```

In [110]:

```
model.summary()
```

```
Model: "functional_4"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer_4 (InputLayer)      │ (None, 784)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_15 (Dense)                │ (None, 32)             │        25,120 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_16 (Dense)                │ (None, 32)             │         1,056 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_17 (Dense)                │ (None, 32)             │         1,056 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_18 (Dense)                │ (None, 10)             │           330 │
└─────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 82,688 (323.00 KB)
 Trainable params: 27,562 (107.66 KB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 55,126 (215.34 KB)
```

In [112]:

```
## Learn about the model training history!
history.history
```

Out[112]:

```
{'loss': [0.4425293505191803,
  0.20567937195301056,
  0.16604284942150116,
  0.1427481770515442,
  0.12371402233839035],
 'sparse_categorical_accuracy': [0.8668000102043152,
  0.9393799901008606,
  0.9498199820518494,
  0.9574800133705139,
  0.9629600048065186],
 'val_loss': [0.21168047189712524,
  0.16890335083007812,
  0.15615299344062805,
  0.13860121369361877,
  0.12821520864963531],
 'val_sparse_categorical_accuracy': [0.9395999908447266,
  0.9513999819755554,
  0.9553999900817871,
  0.960099995136261,
  0.9624999761581421]}
```

c. What is the performance of the model on the training and validation sets over the 5 epochs? Which of these do we expect is characteristic of model performance in new data?

In [113]:

```
# Evaluate the model on the training data using `evaluate`
print("Evaluate on training data")
results = model.evaluate(x_train, y_train, batch_size=64)
print(f"\ntraining loss: {results[0]}, \ntest acc: {results[1]}")

# Evaluate the model on the validation data using `evaluate`
print("\nEvaluate on validation data")
results = model.evaluate(x_val, y_val, batch_size=64)
print(f"\nvalidation loss: {results[0]}, \nvalidation acc: {results[1]}")
```

```
Evaluate on training data
782/782 ━━━━━━━━━━━━━━━━━━━━ 0s 413us/step - loss: 0.1001 - sparse_categorical_accuracy: 0.9694

training loss: 0.1065206527709961,
test acc: 0.9674199819564819

Evaluate on validation data
157/157 ━━━━━━━━━━━━━━━━━━━━ 0s 445us/step - loss: 0.1407 - sparse_categorical_accuracy: 0.9610

validation loss: 0.12821520864963531,
validation acc: 0.9624999761581421
```

The model performs with 0.1065 loss and 96.74% accuracy on training data. This is good loss, because 0 is the best possible (perfect prediction). The loss decreases over time. The model performs with 0.1282 loss and 96.25% for the validation data. This is slightly less accurate than the training data, to be expected, but still quite high. The validation accuracy is characteristic of new, unseen data, so it is a good indicator for how the model will perform on testing data. Since the training and validation accuracy are very similar, with loss decreasing and accuracy increasing over each epoch for both, we expect that the model is likely to perform well on new testing data.

d. Check your model performance on the test set, and compare with your expectations in part c.

In [114]:
```
# Evaluate the model on the test data using `evaluate`
print("Evaluate on test data")
results = model.evaluate(x_test, y_test, batch_size=64)
print(f"\ntest loss: {results[0]}, \ntest acc: {results[1]}")
```

```
Evaluate on test data
157/157 ━━━━━━━━━━━━━━━━━━━━ 0s 451us/step - loss: 0.1476 - sparse_categorical_accuracy: 0.9535

test loss: 0.13321974873542786,
test acc: 0.9580000042915344
```

The model performs with 0.1332 loss and 95.80% accuracy for testing data. Compared to the validation data performance, the testing data has slightly higher loss and slightly lower accuracy, with a difference of +0.015 loss and -0.45% accuracy. This aligns with our expectation in part c because we expected the model to perform similarly to the validation data.