Assignment 1: MLPs and Backpropagation

Due on November 7th, 2024 (23:59:59)

Welcome to Comp541: Deep Learning Course!

Before you start, make sure you read the README.txt in the same directory as this notebook for important setup information. A lot of code is provided in this notebook, and we highly encourage you to read and understand it as part of the learning.

Assignment Notes: Please make sure to save the notebook as you go along. Submission Instructions are located at the bottom of the notebook.

In [109]:
```
# All Import Statements Defined Here
# Note: Do not add to this list.
# ----------------

import sys
assert sys.version_info[0]==3
assert sys.version_info[1] >= 5

from gensim.models import KeyedVectors
from gensim.test.utils import datapath
import pprint
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 5]
import nltk
nltk.download('reuters')
from nltk.corpus import reuters
import numpy as np
import random
import scipy as sp
from sklearn.decomposition import TruncatedSVD
from sklearn.decomposition import PCA

START_TOKEN = '<START>'
END_TOKEN = '<END>'

np.random.seed(0)
random.seed(0)
# ----------------
```

```
[nltk_data] Downloading package reuters to
[nltk_data]     C:\Users\KPAYCI21\AppData\Roaming\nltk_data...
[nltk_data]   Package reuters is already up-to-date!
```

Word Vectors

Word Vectors are often used as a fundamental component for downstream NLP tasks, e.g. question answering, text generation, translation, etc., so it is important to build some intuitions as to their strengths and weaknesses. Here, you will explore two types of word vectors: those derived from *co-occurrence matrices*, and those derived via *GloVe*.

Note on Terminology: The terms "word vectors" and "word embeddings" are often used interchangeably. The term "embedding" refers to the fact that we are encoding aspects of a word's meaning in a lower dimensional space. As Wikipedia (https://en.wikipedia.org/wiki/Word_embedding) states, "*conceptually it involves a mathematical embedding from a space with one dimension per word to a continuous vector space with a much lower dimension*".

Part 1: Count-Based Word Vectors (10 points)

Most word vector models start from the following idea:

You shall know a word by the company it keeps (Firth, J. R. 1957:11 (https://en.wikipedia.org/wiki/John_Rupert_Firth))

Many word vector implementations are driven by the idea that similar words, i.e., (near) synonyms, will be used in similar contexts. As a result, similar words will often be spoken or written along with a shared subset of words, i.e., contexts. By examining these contexts, we can try to develop embeddings for our words. With this intuition in mind, many "old school" approaches to constructing word vectors relied on word counts. Here we elaborate upon one of those strategies, *co-occurrence matrices* (for more information, see here (http://web.stanford.edu/class/cs124/lec/vectorsemantics.video.pdf) or here (https://medium.com/data-science-group-iitr/word-embedding-2d05d270b285)).

Co-Occurrence

A co-occurrence matrix counts how often things co-occur in some environment. Given some word w_i occurring in the document, we consider the *context window* surrounding w_i. Supposing our fixed window size is n, then this is the n preceding and n subsequent words in that document, i.e. words $w_{i-n} \dots w_{i-1}$ and $w_{i+1} \dots w_{i+n}$. We build a *co-occurrence matrix* M, which is a symmetric word-by-word matrix in which M_{ij} is the number of times w_j appears inside w_i's window among all documents.

Example: Co-Occurrence with Fixed Window of n=1:

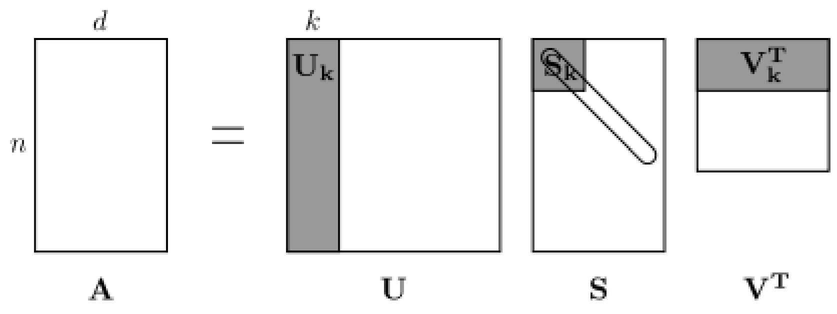
Document 1: "all that glitters is not gold"

Document 2: "all is well that ends well"

*	<START>	all	that	glitters	is	not	gold	well	ends	<END>
<START>	0	2	0	0	0	0	0	0	0	0
all	2	0	1	0	1	0	0	0	0	0
that	0	1	0	1	0	0	0	1	1	0
glitters	0	0	1	0	1	0	0	0	0	0
is	0	1	0	1	0	1	0	1	0	0
not	0	0	0	0	1	0	1	0	0	0
gold	0	0	0	0	0	1	0	0	0	1
well	0	0	1	0	1	0	0	0	1	1
ends	0	0	1	0	0	0	0	1	0	0
<END>	0	0	0	0	0	0	1	1	0	0

Note: In NLP, we often add `<START>` and `<END>` tokens to represent the beginning and end of sentences, paragraphs or documents. In thise case we imagine `<START>` and `<END>` tokens encapsulating each document, e.g., " `<START>` All that glitters is not gold `<END>` ", and include these tokens in our co-occurrence counts.

The rows (or columns) of this matrix provide one type of word vectors (those based on word-word co-occurrence), but the vectors will be large in general (linear in the number of distinct words in a corpus). Thus, our next step is to run *dimensionality reduction*. In particular, we will run *SVD (Singular Value Decomposition)*, which is a kind of generalized *PCA (Principal Components Analysis)* to select the top k principal components. Here's a visualization of dimensionality reduction with SVD. In this picture our co-occurrence matrix is A with n rows corresponding to n words. We obtain a full matrix decomposition, with the singular values ordered in the diagonal S matrix, and our new, shorter length-k word vectors in U_k.



This reduced-dimensionality co-occurrence representation preserves semantic relationships between words, e.g. *doctor* and *hospital* will be closer than *doctor* and *dog*.

Notes: If you can barely remember what an eigenvalue is, here's [a slow, friendly introduction to SVD (https://davetang.org/file/Singular_Value_Decomposition_Tutorial.pdf)](https://davetang.org/file/Singular_Value_Decomposition_Tutorial.pdf). If you want to learn more thoroughly about PCA or SVD, feel free to check out lectures [7 (https://web.stanford.edu/class/cs168/l/l7.pdf)](https://web.stanford.edu/class/cs168/l/l7.pdf), [8 (http://theory.stanford.edu/~tim/s15/l/l8.pdf)](http://theory.stanford.edu/~tim/s15/l/l8.pdf), and [9 (https://web.stanford.edu/class/cs168/l/l9.pdf)](https://web.stanford.edu/class/cs168/l/l9.pdf) of CS168. These course notes provide a great high-level treatment of these general purpose algorithms. Though, for the purpose of this class, you only need to know how to extract the k-dimensional embeddings by utilizing pre-programmed implementations of these algorithms from the numpy, scipy, or sklearn python packages. In practice, it is challenging to apply full SVD to large corpora because of the memory needed to perform PCA or SVD. However, if you only want the top k vector components for relatively small k — known as [Truncated SVD (https://en.wikipedia.org/wiki/Singular_value_decomposition#Truncated_SVD)](https://en.wikipedia.org/wiki/Singular_value_decomposition#Truncated_SVD) — then there are reasonably scalable techniques to compute those iteratively.

Plotting Co-Occurrence Word Embeddings

Here, we will be using the Reuters (business and financial news) corpus. If you haven't run the import cell at the top of this page, please run it now (click it and press SHIFT-RETURN). The corpus consists of 10,788 news documents totaling 1.3 million words. These documents span 90 categories and are split into train and test. For more details, please see [https://www.nltk.org/book/ch02.html (https://www.nltk.org/book/ch02.html)](https://www.nltk.org/book/ch02.html). We provide a `read_corpus` function below that pulls out only articles from the "crude" (i.e. news articles about oil, gas, etc.) category. The function also adds `<START>` and `<END>` tokens to each

In [110]:
```
def read_corpus(category="crude"):
    """ Read files from the specified Reuter's category.
        Params:
            category (string): category name
        Return:
            list of lists, with words from each of the processed files
    """
    files = reuters.fileids(category)
    return [[START_TOKEN] + [w.lower() for w in list(reuters.words(f))] + [
```

Let's have a look what these documents are like….

In [111]:
```
reuters_corpus = read_corpus()
pprint.pprint(reuters_corpus[:3], compact=True, width=100)
```

```
 ', 'a', '54', 'pct', 'leap',
  'from', '1985', '.', 'the', 'turn', 'in', 'the', 'fortunes', 'of', 't
he', 'once', '-', 'sickly',
  'chemical', 'industry', 'has', 'been', 'brought', 'about', 'by', 'a',
'combination', 'of', 'luck',
  'and', 'planning', ',', 'said', 'pace', "'", 's', 'john', 'dosher',
'.', 'dosher', 'said', 'last',
  'year', "'", 's', 'fall', 'in', 'oil', 'prices', 'made', 'feedstock
s', 'dramatically', 'cheaper',
  'and', 'at', 'the', 'same', 'time', 'the', 'american', 'dollar', 'wa
s', 'weakening', 'against',
  'foreign', 'currencies', '.', 'that', 'helped', 'boost', 'u', '.',
's', '.', 'chemical',
  'exports', '.', 'also', 'helping', 'to', 'bring', 'supply', 'and', 'd
emand', 'into', 'balance',
  'has', 'been', 'the', 'gradual', 'market', 'absorption', 'of', 'the',
'extra', 'chemical',
  'manufacturing', 'capacity', 'created', 'by', 'middle', 'eastern', 'o
il', 'producers', 'in',
  'the', 'early', '1980s', '.', 'finally', ',', 'virtually', 'all', 'ma
```

Question 1.1: Implement `distinct_words` [code] (2 points)

Write a method to work out the distinct words (word types) that occur in the corpus. You can do this with `for` loops, but it's more efficient to do it with Python list comprehensions. In particular, this (https://coderwall.com/p/rcmaea/flatten-a-list-of-lists-in-one-line-in-python) may be useful to flatten a list of lists. If you're not familiar with Python list comprehensions in general, here's more information (https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html).

Your returned `corpus_words` should be sorted. You can use python's `sorted` function for this.

You may find it useful to use Python sets (https://www.w3schools.com/python/python_sets.asp) to remove duplicate words.

In [112]:
```
def distinct_words(corpus):
    """ Determine a list of distinct words for the corpus.
        Params:
            corpus (list of list of strings): corpus of documents
        Return:
            corpus_words (list of strings): sorted list of distinct words a
            num_corpus_words (integer): number of distinct words across the
    """
    corpus_words = []
    num_corpus_words = -1

    # ------------------
    # Write your implementation here.
    corpus_words = sorted(set(word for document in corpus for word in docum
    num_corpus_words = len(corpus_words)

    # ------------------

    return corpus_words, num_corpus_words
```

In [113]:
```
# ---------------------
# Run this sanity check
# Note that this not an exhaustive check for correctness.
# ---------------------

# Define toy corpus
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN, END
test_corpus_words, num_corpus_words = distinct_words(test_corpus)

# Correct answers
ans_test_corpus_words = sorted([START_TOKEN, "All", "ends", "that", "gold",
ans_num_corpus_words = len(ans_test_corpus_words)

# Test correct number of words
assert(num_corpus_words == ans_num_corpus_words), "Incorrect number of dist

# Test correct words
assert (test_corpus_words == ans_test_corpus_words), "Incorrect corpus_word

# Print Success
print ("-" * 80)
print("Passed All Tests!")
print ("-" * 80)
```

```
--------------------------------------------------------------------------
------
Passed All Tests!
--------------------------------------------------------------------------
------
```

Question 1.2: Implement `compute_co_occurrence_matrix` [code] (3 points)

Write a method that constructs a co-occurrence matrix for a certain window-size n (with a default of 4), considering words n before and n after the word in the center of the window. Here, we start to use `numpy (np)` to represent vectors, matrices, and tensors.

In [114]:
```
def compute_co_occurrence_matrix(corpus, window_size=4):
    """ Compute co-occurrence matrix for the given corpus and window_size (

        Note: Each word in a document should be at the center of a window.
              number of co-occurring words.

              For example, if we take the document "<START> All that glitte
              "All" will co-occur with "<START>", "that", "glitters", "is",

        Params:
            corpus (list of list of strings): corpus of documents
            window_size (int): size of context window
        Return:
            M (a symmetric numpy matrix of shape (number of unique words in
                Co-occurence matrix of word counts.
                The ordering of the words in the rows/columns should be the
            word2ind (dict): dictionary that maps word to index (i.e. row/c
    """
    words, num_words = distinct_words(corpus)
    M = None
    word2ind = {}

    # ------------------
    # Write your implementation here.
    corpus_words, num_words = distinct_words(corpus)
    word2ind = {word: idx for idx, word in enumerate(corpus_words)}

    # Initialize the co-occurrence matrix with zeros
    M = np.zeros((num_words, num_words))

    # Populate the co-occurrence matrix
    for document in corpus:
        for i, word in enumerate(document):
            word_idx = word2ind[word]

            # Define the window bounds within the document
            start = max(i - window_size, 0)
            end = min(i + window_size + 1, len(document))

            # Iterate over each word within the window and update the matri
            for j in range(start, end):
                if i != j:  # Exclude the word itself
                    neighbor_word_idx = word2ind[document[j]]
                    M[word_idx, neighbor_word_idx] += 1

    # ------------------

    return M, word2ind
```

In [115]:
```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
# ---------------------

# Define toy corpus and get student's co-occurrence matrix
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN, END
M_test, word2ind_test = compute_co_occurrence_matrix(test_corpus, window_si

# Correct M and word2ind
M_test_ans = np.array(
    [[0., 0., 0., 0., 0., 0., 1., 0., 0., 1.,],
     [0., 0., 1., 1., 0., 0., 0., 0., 0., 0.,],
     [0., 1., 0., 0., 0., 0., 0., 0., 1., 0.,],
     [0., 1., 0., 0., 0., 0., 0., 0., 0., 1.,],
     [0., 0., 0., 0., 0., 0., 0., 0., 1., 1.,],
     [0., 0., 0., 0., 0., 0., 0., 1., 1., 0.,],
     [1., 0., 0., 0., 0., 0., 0., 1., 0., 0.,],
     [0., 0., 0., 0., 0., 1., 1., 0., 0., 0.,],
     [0., 0., 1., 0., 1., 1., 0., 0., 0., 1.,],
     [1., 0., 0., 1., 1., 0., 0., 0., 1., 0.,]]
)
ans_test_corpus_words = sorted([START_TOKEN, "All", "ends", "that", "gold",
word2ind_ans = dict(zip(ans_test_corpus_words, range(len(ans_test_corpus_wo

# Test correct word2ind
assert (word2ind_ans == word2ind_test), "Your word2ind is incorrect:\nCorre

# Test correct M shape
assert (M_test.shape == M_test_ans.shape), "M matrix has incorrect shape.\n

# Test correct M values
for w1 in word2ind_ans.keys():
    idx1 = word2ind_ans[w1]
    for w2 in word2ind_ans.keys():
        idx2 = word2ind_ans[w2]
        student = M_test[idx1, idx2]
        correct = M_test_ans[idx1, idx2]
        if student != correct:
            print("Correct M:")
            print(M_test_ans)
            print("Your M: ")
            print(M_test)
            raise AssertionError("Incorrect count at index ({}, {})=({}, {}

# Print Success
print ("-" * 80)
print("Passed All Tests!")
print ("-" * 80)
```

```
--------------------------------------------------------------------------
------
Passed All Tests!
--------------------------------------------------------------------------
------
```

Question 1.3: Implement `reduce_to_k_dim` [code] (1 point)

Construct a method that performs dimensionality reduction on the matrix to produce k-dimensional embeddings. Use SVD to take the top k components and produce a new matrix of k-dimensional embeddings.

Note: All of numpy, scipy, and scikit-learn (`sklearn`) provide *some* implementation of SVD, but only scipy and sklearn provide an implementation of Truncated SVD, and only sklearn provides an efficient randomized algorithm for calculating large-scale Truncated SVD. So please use sklearn.decomposition.TruncatedSVD (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html).

In [116]:
```
def reduce_to_k_dim(M, k=2):
    """ Reduce a co-occurence count matrix of dimensionality (num_corpus_wo
        to a matrix of dimensionality (num_corpus_words, k) using the follo
            - http://scikit-learn.org/stable/modules/generated/sklearn.deco

        Params:
            M (numpy matrix of shape (number of unique words in the corpus
            k (int): embedding size of each word after dimension reduction
        Return:
            M_reduced (numpy matrix of shape (number of corpus words, k)):
                    In terms of the SVD from math class, this actually retu
    """
    n_iters = 10     # Use this parameter in your call to `TruncatedSVD`
    M_reduced = None
    print("Running Truncated SVD over %i words..." % (M.shape[0]))

    # ------------------
    # Write your implementation here.

    svd = TruncatedSVD(n_components=k, n_iter=n_iters, random_state=42)
    M_reduced = svd.fit_transform(M)  # This gives us the reduced dimension
    # ------------------

    print("Done.")
    return M_reduced
```

In [117]:
```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness
# In fact we only check that your M_reduced has the right dimensions.
# ---------------------

# Define toy corpus and run student code
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN, END
M_test, word2ind_test = compute_co_occurrence_matrix(test_corpus, window_si
M_test_reduced = reduce_to_k_dim(M_test, k=2)

# Test proper dimensions
assert (M_test_reduced.shape[0] == 10), "M_reduced has {} rows; should have
assert (M_test_reduced.shape[1] == 2), "M_reduced has {} columns; should ha

# Print Success
print ("-" * 80)
print("Passed All Tests!")
print ("-" * 80)
```

```
Running Truncated SVD over 10 words...
Done.
--------------------------------------------------------------------------
------
Passed All Tests!
--------------------------------------------------------------------------
------
```

Question 1.4: Implement `plot_embeddings` [code] (1 point)

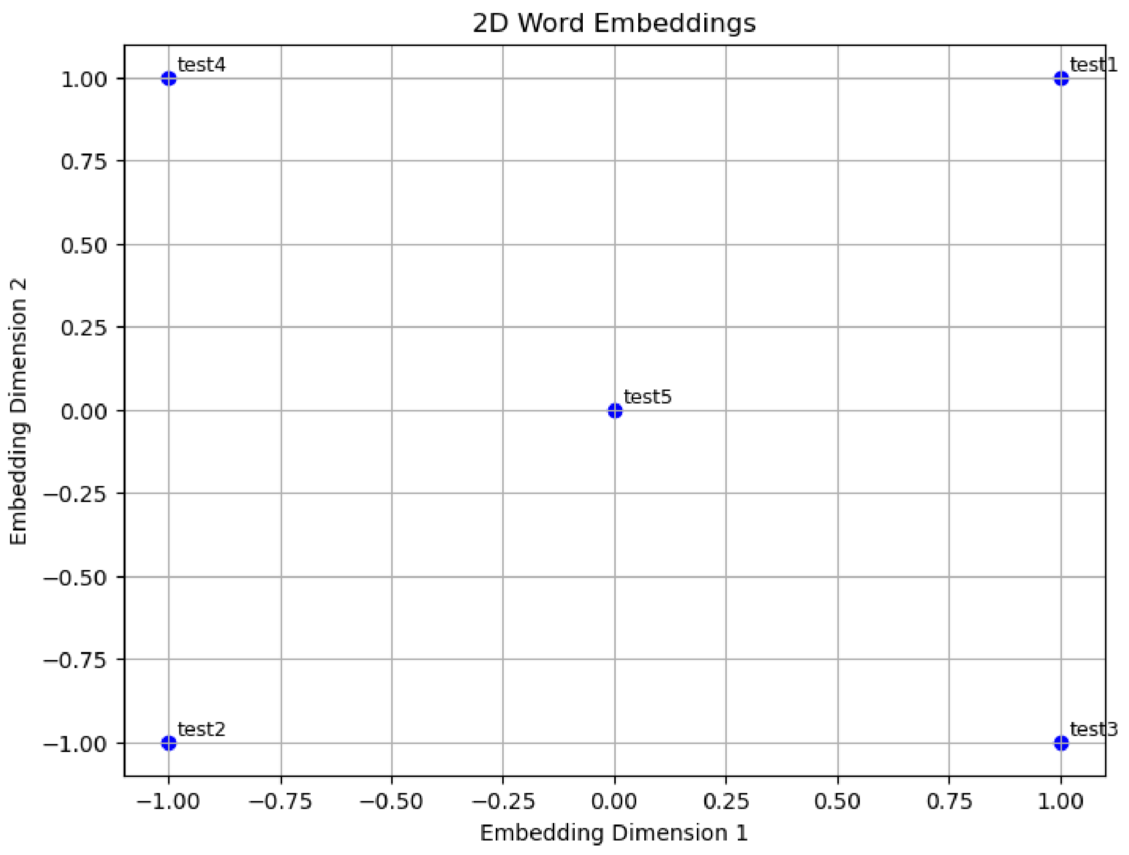
Here you will write a function to plot a set of 2D vectors in 2D space. For graphs, we will use Matplotlib (`plt`).

For this example, you may find it useful to adapt this code (http://web.archive.org/web/20190924160434/https://www.pythonmembers.club/2018/05/08/m scatter-plot-annotate-set-text-at-label-each-point/). In the future, a good way to make a plot is to look at the Matplotlib gallery (https://matplotlib.org/gallery/index.html), find a plot that looks somewhat like what you want, and adapt the code they give.

In [118]:
```python
def plot_embeddings(M_reduced, word2ind, words):
    """ Plot in a scatterplot the embeddings of the words specified in the
        NOTE: do not plot all the words listed in M_reduced / word2ind.
        Include a label next to each point.

        Params:
            M_reduced (numpy matrix of shape (number of unique words in the
            word2ind (dict): dictionary that maps word to indices for matri
            words (list of strings): words whose embeddings we want to visu
    """

    # ------------------
    # Write your implementation here.
    plt.figure(figsize=(8, 6))
    for word in words:
        # Get the index of the word and its corresponding 2D embedding
        idx = word2ind[word]
        x, y = M_reduced[idx, 0], M_reduced[idx, 1]

        # Plot the point
        plt.scatter(x, y, marker='o', color='blue')

        # Annotate the point with the word label
        plt.text(x + 0.02, y + 0.02, word, fontsize=9)


    plt.title("2D Word Embeddings")
    plt.xlabel("Embedding Dimension 1")
    plt.ylabel("Embedding Dimension 2")
    plt.grid(True)
    plt.show()

    # ------------------
```

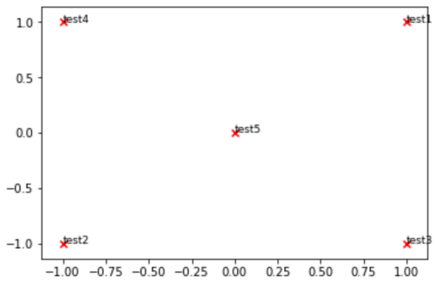
In [119]:
```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
# The plot produced should look like the "test solution plot" depicted belo
# ---------------------

print ("-" * 80)
print ("Outputted Plot:")

M_reduced_plot_test = np.array([[1, 1], [-1, -1], [1, -1], [-1, 1], [0, 0]]
word2ind_plot_test = {'test1': 0, 'test2': 1, 'test3': 2, 'test4': 3, 'test
words = ['test1', 'test2', 'test3', 'test4', 'test5']
plot_embeddings(M_reduced_plot_test, word2ind_plot_test, words)

print ("-" * 80)
```

```
--------------------------------------------------------------------------
------
Outputted Plot:
```



```
--------------------------------------------------------------------------
------
```

Test Plot Solution



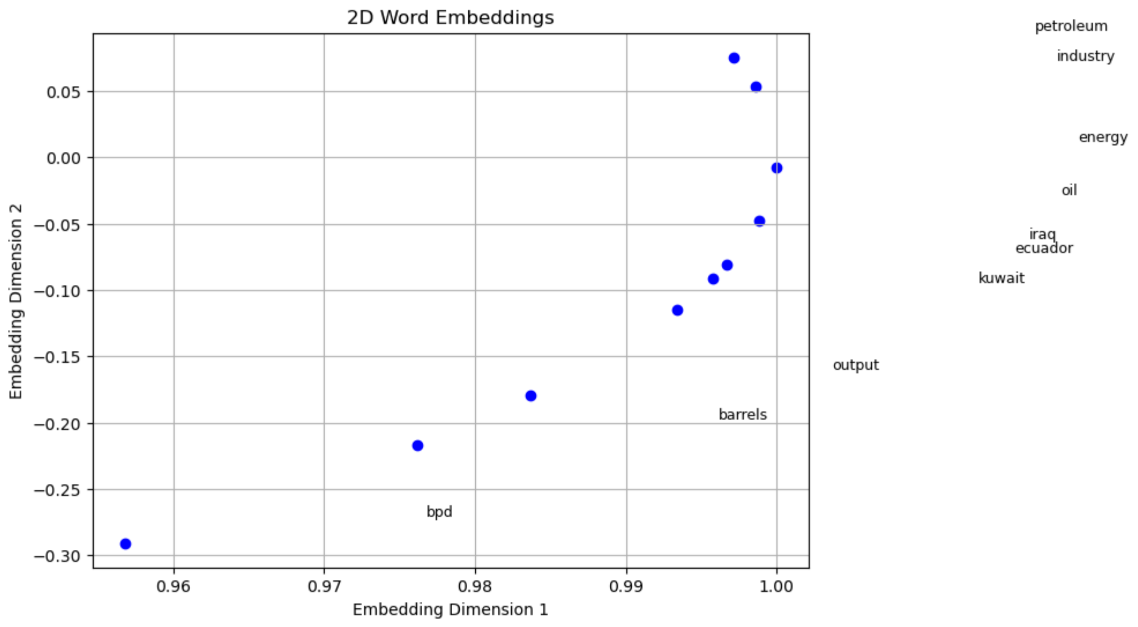
Question 1.5: Co-Occurrence Plot Analysis [written] (3 points)

Now we will put together all the parts you have written! We will compute the co-occurrence matrix with fixed window of 4 (the default window size), over the Reuters "crude" (oil) corpus. Then we will use TruncatedSVD to compute 2-dimensional embeddings of each word. TruncatedSVD returns U*S, so we need to normalize the returned vectors, so that all the vectors will appear around the unit circle (therefore closeness is directional closeness). **Note**: The line of code below that does the normalizing uses the NumPy concept of *broadcasting*. If you don't know about broadcasting, check out Computation on Arrays: Broadcasting by Jake VanderPlas (https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html).

Run the below cell to produce the plot. It'll probably take a few seconds to run. What clusters together in 2-dimensional embedding space? What doesn't cluster together that you might think should have? **Note:** "bpd" stands for "barrels per day" and is a commonly used abbreviation in crude oil topic articles.

In [120]:
```
# ------------------------------
# Run This Cell to Produce Your Plot
# ------------------------------
reuters_corpus = read_corpus()
M_co_occurrence, word2ind_co_occurrence = compute_co_occurrence_matrix(reut
M_reduced_co_occurrence = reduce_to_k_dim(M_co_occurrence, k=2)

# Rescale (normalize) the rows to make them each of unit-length
M_lengths = np.linalg.norm(M_reduced_co_occurrence, axis=1)
M_normalized = M_reduced_co_occurrence / M_lengths[:, np.newaxis] # broadca

words = ['barrels', 'bpd', 'ecuador', 'energy', 'industry', 'kuwait', 'oil'

plot_embeddings(M_normalized, word2ind_co_occurrence, words)
```

```
Running Truncated SVD over 8185 words...
Done.
```



Write your answer here.

I think "energy" and "oil" should be more close. Also "Kuwait", "ecuador", "iraq" are respectively close and these are the countries that produce oil. Also "petroleum" and "industry" make a cluster. These words are more used in similar contexts.

Part 2: Prediction-Based Word Vectors (15 points)

As discussed in class, more recently prediction-based word vectors have demonstrated better performance, such as word2vec and GloVe (which also utilizes the benefit of counts). Here, we shall explore the embeddings produced by GloVe. Please revisit the class notes and lecture slides for more details on the word2vec and GloVe algorithms. If you're feeling adventurous, challenge yourself and try reading GloVe's original paper (https://nlp.stanford.edu/pubs/glove.pdf).

Then run the following cells to load the GloVe vectors into memory. **Note**: If this is your first time to run these cells, i.e. download the embedding model, it will take a couple minutes to run. If you've run these cells before, rerunning them will load the model without redownloading it, which will take about 1 to 2 minutes.

In [121]:
```
def load_embedding_model():
    """ Load GloVe Vectors
        Return:
            wv_from_bin: All 400000 embeddings, each lengh 200
    """
    import gensim.downloader as api
    wv_from_bin = api.load("glove-wiki-gigaword-200")
    print("Loaded vocab size %i" % len(wv_from_bin.vocab.keys()))
    return wv_from_bin
```

In [122]:
```
# -----------------------------------
# Run Cell to Load Word Vectors
# Note: This will take a couple minutes
# -----------------------------------
wv_from_bin = load_embedding_model()
```

```
Loaded vocab size 400000
```

Note: If you are receiving a "reset by peer" error, rerun the cell to restart the download.

Reducing dimensionality of Word Embeddings

Let's directly compare the GloVe embeddings to those of the co-occurrence matrix. In order to avoid running out of memory, we will work with a sample of 10000 GloVe vectors instead. Run the following cells to:

1. Put 10000 Glove vectors into a matrix M
2. Run `reduce_to_k_dim` (your Truncated SVD function) to reduce the vectors from 200-dimensional to 2-dimensional.

In [123]:
```
def get_matrix_of_vectors(wv_from_bin, required_words=['barrels', 'bpd', 'e
    """ Put the GloVe vectors into a matrix M.
        Param:
            wv_from_bin: KeyedVectors object; the 400000 GloVe vectors load
        Return:
            M: numpy matrix shape (num words, 200) containing the vectors
            word2ind: dictionary mapping each word to its row number in M
    """
    import random
    words = list(wv_from_bin.vocab.keys())
    print("Shuffling words ...")
    random.seed(224)
    random.shuffle(words)
    words = words[:10000]
    print("Putting %i words into word2ind and matrix M..." % len(words))
    word2ind = {}
    M = []
    curInd = 0
    for w in words:
        try:
            M.append(wv_from_bin.word_vec(w))
            word2ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    for w in required_words:
        if w in words:
            continue
        try:
            M.append(wv_from_bin.word_vec(w))
            word2ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    M = np.stack(M)
    print("Done.")
    return M, word2ind
```

In [124]:
```
# ------------------------------------------------------------------
# Run Cell to Reduce 200-Dimensional Word Embeddings to k Dimensions
# Note: This should be quick to run
# ------------------------------------------------------------------
M, word2ind = get_matrix_of_vectors(wv_from_bin)
M_reduced = reduce_to_k_dim(M, k=2)

# Rescale (normalize) the rows to make them each of unit-length
M_lengths = np.linalg.norm(M_reduced, axis=1)
M_reduced_normalized = M_reduced / M_lengths[:, np.newaxis] # broadcasting
```

```
Shuffling words ...
Putting 10000 words into word2ind and matrix M...
Done.
Running Truncated SVD over 10010 words...
Done.
```

Note: If you are receiving out of memory issues on your local machine, try closing other applications to free more memory on your device. You may want to try restarting your machine so that you can free up extra memory. Then immediately run

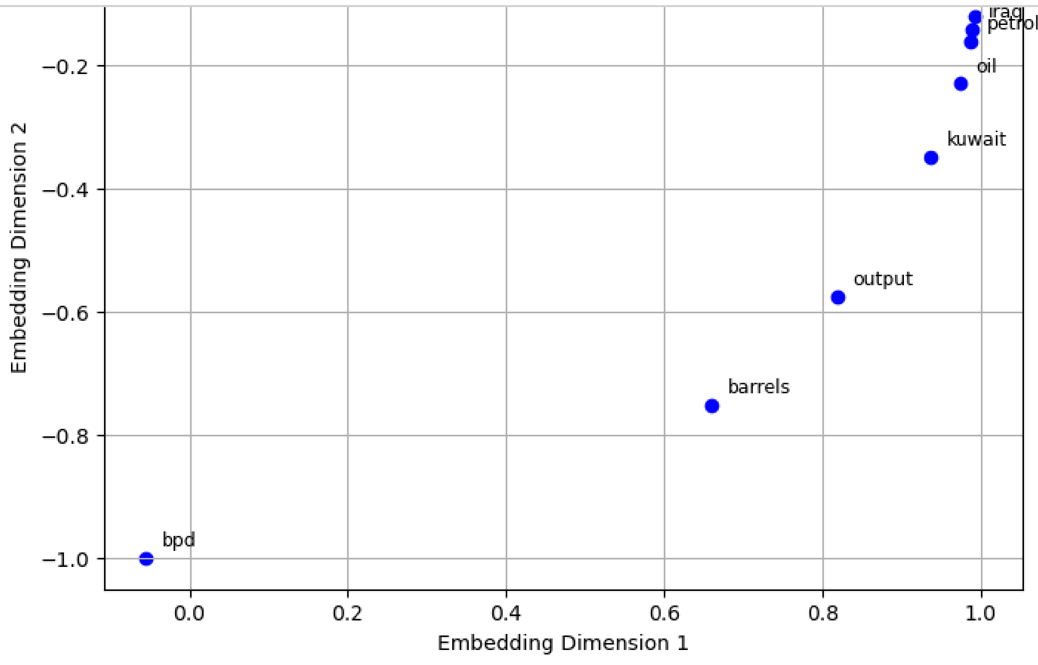
the jupyter notebook and see if you can load the word vectors properly. If you still have problems with loading the embeddings onto your local machine after this, please go to office hours or contact course staff.

Question 2.1: GloVe Plot Analysis [written] (3 points)

Run the cell below to plot the 2D GloVe embeddings for `['barrels', 'bpd', 'ecuador', 'energy', 'industry', 'kuwait', 'oil', 'output', 'petroleum', 'iraq']`.

What clusters together in 2-dimensional embedding space? What doesn't cluster together that you think should have? How is the plot different from the one generated earlier from the co-occurrence matrix? What is a possible cause for the difference?

In [125]:
```
words = ['barrels', 'bpd', 'ecuador', 'energy', 'industry', 'kuwait', 'oil'
plot_embeddings(M_reduced_normalized, word2ind, words)
```



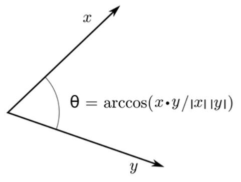
Write your answer here.

Countries should be a cluster but kuwait is a anomalia here. Also Industry and energy are more close than co-occurence version. The reason is same: These words are more used in similar contexts.

Cosine Similarity

Now that we have word vectors, we need a way to quantify the similarity between individual words, according to these vectors. One such metric is cosine-similarity. We will be using this to find words that are "close" and "far" from one another.

We can think of n-dimensional vectors as points in n-dimensional space. If we take this perspective [L1 (http://mathworld.wolfram.com/L1-Norm.html)](http://mathworld.wolfram.com/L1-Norm.html) and [L2 (http://mathworld.wolfram.com/L2-Norm.html)](http://mathworld.wolfram.com/L2-Norm.html) Distances help quantify the amount of space "we must travel" to get between these two points. Another approach is to examine the angle between two vectors. From trigonometry we know that:



Instead of computing the actual angle, we can leave the similarity in terms of $similarity = cos(\Theta)$. Formally the Cosine Similarity (https://en.wikipedia.org/wiki/Cosine_similarity) s between two vectors p and q is defined as:

$$s = \frac{p \cdot q}{||p||||q||}, \text{ where } s \in [-1, 1]$$

Question 2.2: Words with Multiple Meanings (1.5 points) [code + written]

Polysemes and homonyms are words that have more than one meaning (see this wiki page (https://en.wikipedia.org/wiki/Polysemy) to learn more about the difference between polysemes and homonyms). Find a word with *at least two different meanings* such that the top-10 most similar words (according to cosine similarity) contain related words from *both* meanings. For example, "leaves" has both "go_away" and "a_structure_of_a_plant" meaning in the top 10, and "scoop" has both "handed_waffle_cone" and "lowdown". You will probably need to try several polysemous or homonymic words before you find one.

Please state the word you discover and the multiple meanings that occur in the top 10. Why do you think many of the polysemous or homonymic words you tried didn't work (i.e. the top-10 most similar words only contain **one** of the meanings of the words)?

Note: You should use the `wv_from_bin.most_similar(word)` function to get the top 10 similar words. This function ranks all other words in the vocabulary with respect to their cosine similarity to the given word. For further assistance, please check the **GenSim documentation (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedve**

In [126]:
```
# ------------------
# Write your implementation here.
polysemes = ["suit"]
results = {}
for word in polysemes:
    similar_words = wv_from_bin.most_similar(word, topn=10)
    print(f"Word: {word}\nTop-10 similar words:")
    for similar_word, similarity in similar_words:
        print(f"  {similar_word}: {similarity}")
    print("\n")

# ------------------
```

```
Word: suit
Top-10 similar words:
  suits: 0.8774810433387756
  lawsuit: 0.7398070693016052
  filed: 0.6544123291969299
  complaint: 0.5840126872062683
  filing: 0.575992226600647
  wearing: 0.5652857422828674
  plaintiffs: 0.5600565671920776
  sued: 0.5595393180847168
  case: 0.5564867854118347
  jacket: 0.5559611916542053
```

Write your answer here.

Question 2.3: Synonyms & Antonyms (2 points) [code + written]

When considering Cosine Similarity, it's often more convenient to think of Cosine Distance, which is simply 1 - Cosine Similarity.

Find three words (w_1, w_2, w_3) where w_1 and w_2 are synonyms and w_1 and w_3 are antonyms, but Cosine Distance $(w_1, w_3) <$ Cosine Distance (w_1, w_2).

As an example, w_1="happy" is closer to w_3="sad" than to w_2="cheerful". Please find a different example that satisfies the above. Once you have found your example, please give a possible explanation for why this counter-intuitive result may have happened.

You should use the the `wv_from_bin.distance(w1, w2)` function here in order to compute the cosine distance between two words. Please see the GenSim documentation (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedve for further assistance.

In [127]:
```
# ------------------
# Write your implementation here.
w1, w2, w3 = "happy", "cheerful", "sad" #this was for seeing the result, il

w1_w2_distance = wv_from_bin.distance(w1, w2)
w1_w3_distance = wv_from_bin.distance(w1, w3)

print(f"{w1_w3_distance} < {w1_w2_distance}" if w1_w3_distance < w1_w2_dist

# ------------------
```

```
0.4040136933326721 < 0.5172466933727264
```

Write your answer here.

Because some antonym words are tend to be used in similar context.

Question 2.4: Analogies with Word Vectors [written] (1.5 points)

Word vectors have been shown to *sometimes* exhibit the ability to solve analogies.

As an example, for the analogy "man : king :: woman : x" (read: man is to king as woman is to x), what is x?

In the cell below, we show you how to use word vectors to find x using the `most_similar` function from the [GenSim documentation (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedve](https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedve) The function finds words that are most similar to the words in the `positive` list and most dissimilar from the words in the `negative` list (while omitting the input words, which are often the most similar; see [this paper (https://www.aclweb.org/anthology/N18-2039.pdf)](https://www.aclweb.org/anthology/N18-2039.pdf)). The answer to the analogy will have the highest cosine similarity (largest returned numerical value).

In [128]:
```
# Run this cell to answer the analogy -- man : king :: woman : x
pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'king'], negative
```

```
[('queen', 0.6978679299354553),
 ('princess', 0.6081743836402893),
 ('monarch', 0.5889754891395569),
 ('throne', 0.5775110125541687),
 ('prince', 0.5750998258590698),
 ('elizabeth', 0.5463595986366272),
 ('daughter', 0.5399126410484314),
 ('kingdom', 0.5318052768707275),
 ('mother', 0.5168544054031372),
 ('crown', 0.5164472460746765)]
```

Let m, k, w, and x denote the word vectors for `man`, `king`, `woman`, and the answer, respectively. Using **only** vectors m, k, w, and the vector arithmetic operators $+$ and $-$ in your answer, what is the expression in which we are maximizing cosine similarity with x?

Hint: Recall that word vectors are simply multi-dimensional vectors that represent a word. It

Write your answer here.

```
EXPRESSION: minimize => [(w - m) - (x - k)] for x;
this implements 0 (approximately) = [(w - m) - (x - k)]
=> w - m = x - k => x = w - m + k => x is the most similar word wi
th the word: w - m + k
```

Question 2.5: Finding Analogies [code + written] (1.5 points)

Find an example of analogy that holds according to these vectors (i.e. the intended word is ranked top). In your solution please state the full analogy in the form x:y :: a:b. If you believe the analogy is complicated, explain why the analogy holds in one or two sentences.

Note: You may have to try many analogies to find one that works!

In [129]:
```
# ------------------
# Write your implementation here.
# man : groom :: woman : x
pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'bride'], negativ
# ------------------
```

```
[('groom', 0.6067668795585632),
 ('wedding', 0.6011814475059509),
 ('brides', 0.5978908538818359),
 ('bridegroom', 0.5542412400245667),
 ('marry', 0.5450136065483093),
 ('pregnant', 0.5319290161132812),
 ('girl', 0.5257630348205566),
 ('daughter', 0.5251718759536743),
 ('mother', 0.5247462391853333),
 ('princess', 0.5237212777137756)]
```

Write your answer here.

because bride - woman = marriage => man + marriage = groom

Question 2.6: Incorrect Analogy [code + written] (1.5 points)

Find an example of analogy that does *not* hold according to these vectors. In your solution, state the intended analogy in the form x:y :: a:b, and state the (incorrect) value of b according to the word vectors.

In [130]:
```
# ------------------
# Write your implementation here.


# ------------------
```

Write your answer here.

Question 2.7: Guided Analysis of Bias in Word Vectors [written] (1 point)

It's important to be cognizant of the biases (gender, race, sexual orientation etc.) implicit in our word embeddings. Bias can be dangerous because it can reinforce stereotypes through applications that employ these models.

Run the cell below, to examine (a) which terms are most similar to "woman" and "worker" and most dissimilar to "man", and (b) which terms are most similar to "man" and "worker" and most dissimilar to "woman". Point out the difference between the list of female-associated words and the list of male-associated words, and explain how it is reflecting gender bias.

In [131]:
```
# Run this cell
# Here `positive` indicates the list of words to be similar to and `negativ
# most dissimilar from.
pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'worker'], negati
print()
pprint.pprint(wv_from_bin.most_similar(positive=['man', 'worker'], negative
```

```
[('employee', 0.6375863552093506),
 ('workers', 0.6068919897079468),
 ('nurse', 0.5837946534156799),
 ('pregnant', 0.536388635635376),
 ('mother', 0.5321309566497803),
 ('employer', 0.5127025842666626),
 ('teacher', 0.5099576711654663),
 ('child', 0.5096741914749146),
 ('homemaker', 0.5019454956054688),
 ('nurses', 0.4970572292804718)]

[('workers', 0.611325740814209),
 ('employee', 0.5983108282089233),
 ('working', 0.5615329146385193),
 ('laborer', 0.5442320108413696),
 ('unemployed', 0.5368516445159912),
 ('job', 0.5278826951980591),
 ('work', 0.5223962664604187),
 ('mechanic', 0.5088937282562256),
 ('worked', 0.5054520964622498),
 ('factory', 0.4940453767776489)]
```

Write your answer here.

woman = Things that require more attention from others man = Things that require more physical power

Question 2.8: Independent Analysis of Bias in Word Vectors [code + written] (1 point)

Use the `most_similar` function to find another case where some bias is exhibited by the vectors. Please briefly explain the example of bias that you discover.

```
In [132]: # ------------------
          # Write your implementation here.
          pprint.pprint(wv_from_bin.most_similar(positive=['man', 'physicist'], negat
          # ------------------
```

```
[('mathematician', 0.6312294006347656),
 ('scientist', 0.6067357063293457),
 ('chemist', 0.6054997444152832),
 ('physics', 0.5739578604698181),
 ('physicists', 0.56578528881073),
 ('inventor', 0.5634065866470337),
 ('astrophysicist', 0.530306875705719),
 ('astronomer', 0.5110200047492981),
 ('astrophysics', 0.5000457167625427),
 ('philosopher', 0.4885155260562897)]
```

Write your answer here.

according to this vectors close to man and physicist and far from woman are generally scientific jobs.

Question 2.9: Thinking About Bias [written] (2 points)

Give one explanation of how bias gets into the word vectors. What is an experiment that you could do to test for or to measure this source of bias?

Write your answer here.

This bias gets into the word vectors because of the dataset that used for training. And these datasets are naturally extracted contexts. This shows the gender thoughts in people.

Part 3: Sentiment Analysis (15 points)

Lastly, you will implement a simple sentiment classifier **from scratch** by using the Deep Averaging Network (DAN) proposed in the paper (https://aclanthology.org/P15-1162.pdf). The model is based on the following three steps:

- Take the vector average of the embeddings associated with the words in the inputs
- Pass that average vector through one or more feed-forward layers
- Perform linear classification on the final layer's representation

Here, you will use Stanford Sentiment Treebank (SST) dataset but note that in this dataset, the sentiment levels are originally represented with real values. Hence, you need to discretize these values into the following five classes:

- 0: "very negative" (≤ 0.2),
- 1: "negative" (≤ 0.4),
- 2: "neutral" (≤ 0.6),
- 3: "positive" (≤ 0.8),
- 4: "very positive" (> 0.8)

Download the Dataset

You can download the dataset here (https://nlp.stanford.edu/sentiment/) (Download the "**Main zip file with readme (6mb)**" version). Please read `README.txt` in details, that comes with the .zip folder.

Create a /data directory to store your SST data and unzip your downloaded folder there. Your data path should be like following:

```
./comp541-441/assignment1/data
                                    └── stanfordSentimentTreebank
                                        ├── README.txt
                                        ├── SOStr.txt
                                        ├── STree.txt
                                        ├── datasetSentences.txt
                                        ├── datasetSplit.txt
                                        ├── dictionary.txt
                                        ├── original_rt_snippets.txt
                                        └── sentiment_labels.txt
```

Or, you can simply use Huggingface's **datasets** library if you are familiar.

What to show

In your work, perform the following experiments and explain your findings:

- Provide your loss curves by plotting them clearly,
- Play with the number of layers,
- Try with embeddings trained on different corpuses
- Test with the GloVe embeddings and the embeddings formed through the word co-occurrence matrix. Report your results on the test set for both types of embeddings (make sure to use the same test set for both, to ensure a fair comparison).

Submission Instructions

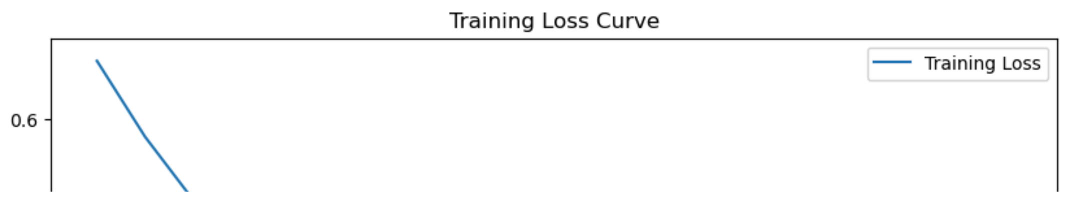
1. Click the Save button at the top of the Jupyter Notebook.
2. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
3. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
4. Once you've rerun everything, select File -> Download as -> PDF via LaTeX (If you have trouble using "PDF via LaTex", you can also save the webpage as pdf. Make sure all your solutions especially the coding parts are displayed in the pdf, it's okay if the provided codes get cut off because lines are not wrapped in code cells).
5. Look at the PDF file and make sure all your solutions are there, displayed correctly.
6. Download a .ipynb version of your notebook
7. Please name your files as username_assignment1.ipynb and username_assignment1.pdf.
8. Submit your work to Blackboard by the deadline.

In [133]:
```python
# ------------------
# Start your implementation here.
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from datasets import load_dataset
import gensim.downloader as api

# huggingface
dataset = load_dataset("sst", "default")

def discretize_sentiment(value):
    if value <= 0.2:
        return 0
    elif value <= 0.4:
        return 1
    elif value <= 0.6:
        return 2
    elif value <= 0.8:
        return 3
    else:
        return 4

def preprocess_data(data_split):
    texts = []
    labels = []
    for example in data_split:
        texts.append(example['sentence'])
        labels.append(discretize_sentiment(example['label']))
    return texts, labels

train_texts, train_labels = preprocess_data(dataset['train'])
test_texts, test_labels = preprocess_data(dataset['test'])


def load_embedding_model():
    """ Load GloVe Vectors
        Return:
            wv_from_bin: All 400000 embeddings, each lengh 200
    """
    import gensim.downloader as api
    wv_from_bin = api.load("glove-wiki-gigaword-200")
    print("Loaded vocab size %i" % len(wv_from_bin.vocab.keys()))
    return wv_from_bin


embedding_lookup = load_embedding_model()
embedding_dim = 200


class SSTDataset(Dataset):
    # I TOOK THIS IMPLEMENTATION FROM CHATGPT
    def __init__(self, texts, labels, embedding_lookup):
        self.texts = texts
        self.labels = labels
        self.embedding_lookup = embedding_lookup

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        words = self.texts[idx].split()
        embeddings = [self.embedding_lookup[word] for word in words if word
        if len(embeddings) > 0:
            avg_embedding = np.mean(embeddings, axis=0)
        else:
            avg_embedding = np.zeros(300)
        return torch.tensor(avg_embedding, dtype=torch.float32), torch.tens


class DANModel:
    def __init__(self, input_dim, hidden_dims, output_dim, learning_rate=0.
        torch.manual_seed(456)
        self.weights = [torch.randn(input_dim, hidden_dims[0], requires_gra
                       [torch.randn(hidden_dims[i], hidden_dims[i+1], requi
                       [torch.randn(hidden_dims[-1], output_dim, requires_g
        self.biases = [torch.randn(h, requires_grad=True) for h in hidden_d
        self.learning_rate = learning_rate
        self.hidden_dims = hidden_dims
        prev_dim = input_dim


        for dim in hidden_dims:
            self.weights.append(torch.randn(prev_dim, dim, requires_grad=Tr
            self.biases.append(torch.zeros(dim, requires_grad=True))
            prev_dim = dim


        self.weights.append(torch.randn(prev_dim, output_dim, requires_grad
        self.biases.append(torch.zeros(output_dim, requires_grad=True))

    def forward(self, x):
        activations = [x]
        for i in range(len(self.hidden_dims)):
            z = torch.matmul(activations[-1], self.weights[i]) + self.biase
            a = self.relu(z)
            activations.append(a)


        output = torch.matmul(activations[-1], self.weights[-1]) + self.bia
        return output

    def relu(self, x):
        return torch.maximum(x, torch.tensor(0.0))




learning_rate = 0.1

def cross_entropy_loss(predictions, labels):
    one_hot_labels = torch.zeros(predictions.shape[0], predictions.shape[1]

    log_probs = torch.log_softmax(predictions, dim=1)
    loss = -torch.sum(one_hot_labels * log_probs) / predictions.shape[0]
    return loss


train_dataset = SSTDataset(train_texts, train_labels, embedding_lookup)
```

```
test_dataset = SSTDataset(test_texts, test_labels, embedding_lookup)


train_loader = DataLoader(train_dataset, batch_size=2, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=2, shuffle=False)




input_dim = 200
hidden_dims = [100, 50]
output_dim = 5





model = DANModel(input_dim=input_dim, hidden_dims=hidden_dims, output_dim=o

train_losses = []

num_epochs = 20
for epoch in range(num_epochs):
    running_loss = 0.0
    for i in range(1):
        predictions = model.forward(inputs)

        loss = cross_entropy_loss(predictions, labels)

        print(f'Epoch {epoch + 1}, Batch {i + 1}, Loss: {loss.item()}')

        loss.backward(retain_graph=True)

        # Check the gradients
        print(f"Gradients for weights[0]: {model.weights[0].grad}")

        with torch.no_grad():
            for j in range(len(model.weights)):
                if model.weights[j].grad is not None:
                    model.weights[j] -= model.learning_rate * model.weights
                    model.biases[j] -= model.learning_rate * model.biases[j

                    # Zero gradients after update
                    model.weights[j].grad.zero_()
                    model.biases[j].grad.zero_()

        running_loss += loss.item()

    avg_loss = running_loss / 3
    train_losses.append(avg_loss)

    print(f'Epoch {epoch + 1}/{num_epochs}, Avg Loss: {avg_loss:.4f}')


plt.plot(range(1, num_epochs + 1), train_losses, label="Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training Loss Curve")
plt.legend()
plt.show()
```

```
all_preds = []
all_labels = []

with torch.no_grad():
    for inputs, labels in test_loader:
        predictions = model.forward(inputs)
        probs = torch.softmax(predictions, dim=1)
        preds = torch.argmax(probs, dim=1)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())


accuracy = accuracy_score(all_labels, all_preds)
print(f'Test Accuracy: %{(accuracy*100):.2f}')


# ------------------
```

```
        ...,
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0006,  0.0000,  0.0002],
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0013,  0.0000, -0.0034],
        [ 0.0000,  0.0000,  0.0000,  ..., -0.0002,  0.0000,  0.0014]])
Epoch 20/20, Avg Loss: 0.1294

C:\Users\KPAYCI21\AppData\Local\anaconda3\envs\comp541\lib\site-package
s\ipykernel_launcher.py:159: UserWarning: The .grad attribute of a Tens
or that is not a leaf Tensor is being accessed. Its .grad attribute wo
n't be populated during autograd.backward(). If you indeed want the gra
dient for a non-leaf Tensor, use .retain_grad() on the non-leaf Tensor.
If you access the non-leaf Tensor by mistake, make sure you access the
leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more
information.
```



This assignment is adapted from Stanford [CS224n (http://web.stanford.edu/class/cs224n/)](http://web.stanford.edu/class/cs224n/)