Assignment 3 – Feed-forward Neural Networks and Word Embeddings

CSCI-LING 5832 – Spring 2025

Due 03/09/2025

Disclaimer: All code turned in is expected to be python. You are given the choice of making a script file (*.py) or a jupyter notebook (*.ipynb). Whichever route you choose, please ensure your final submission is clean and well-organized – enough so as to be quickly evaluated by your TAs.

Allowed packages: *These assignments ask you to implement certain things for the sake of learning. As a result, there may be some packages and imports we do not want you to use, as it would be against the spirit of the assignment. Please limit the import statements you use to the ones used in the scaffolding code. We will also list those statements at the end of this pdf. If there is any other package you want to use, ask on Piazza first, otherwise you may be deducted points on your assignment.*

Word Embeddings

Word embeddings are a powerful concept in natural language processing (NLP) that involve representing words as vectors in a continuous vector space. Word embeddings capture semantic information about words. This means that words with similar meanings are represented by vectors that are close to each other in the vector space. The basic idea for training word embeddings is that we can describe a word based on the company it keeps: words that appear in similar contexts are similar to one another.

This can be captured in a sparse term-document matrix, in which columns (representing each document in a corpus) keep track of the counts of the terms in that document in each row. So, you can find the number of times a specific term w_i occurred in document d_j by looking at the term-document matrix $T_{i,j}$. Words that occur in the same document can be clustered together using an algorithm like TF-IDF or LSA, yielding dense vectors that represent each word (or document) in a continuous vector space.

Word embeddings can also be learned from context via unsupervised machine learning algorithms, such as skip-gram Word2vec. And, last but not least, word embeddings can be learned concurrently with another machine learning task. For example, we can extract the embedding layer of BERT, which was trained using masked language modeling and next sentence prediction objectives (see Section 5.3 of the original paper). In this assignment, we'll do something similar: create a feed-forward neural network with an embedding layer for a simple task. After we train the model for our task, we'll extract the embedding layer and visualize it.

Part 1: Train a Network with an Embedding Layer

We're going to do sentiment classification again, but this time on slightly different data: movie reviews from Rotten Tomatoes. We're using this data because the sequences are very short, and we want to be able to conduct all these experiments on a CPU.

Data preprocessing

The input to our model is going to be sequences of integers. Each integer represents the index of the word in our vocabulary. Scaffolding code for the data preprocessing can be found in `a3-data-scaffolding.py`. The embedding layer of our network is simply a lookup table that will convert the index into its embedding.

1. Load, preprocess, and tokenize the data. Preprocessing steps should include lemmatizing, removing punctuation, special characters, digits, and extra spaces. Feel free to use a package like `nltk` or `spacy`.

2. Write a function that truncates sequences to a certain length. For the sequences that are less than that length, add a special `[PAD]` token to pad it to the same length as all other sequences. Let's start with a length of 40, and then you can decrease if you have trouble running the model on sequences that large.

```
In: pad_sequence[sequence=["the", "cat", "sat"], max_length=4]
Out: ["the", "cat", "sat", "[PAD]"]
In: pad_sequence[sequence=["the", "cat", "sat", "on", "the", "mat"], max_length=4]
Out: ["the", "cat", "sat", "on"]
```

3. Convert the string tokens into indices with dictionaries keeping track of the vocabulary, i.e.:

```
In: token2index["the"]
Out: 1
In: index2token[1]
Out: "the"
```

    - *Note: keep track of the index of the `[PAD]` token; our embedding layer can be instructed not to update the vector for `[PAD]` during training.*

4. Finally, convert the data into X and y tensors for ingestion into our PyTorch model. We've included all of the code necessary for this task, but it's beneficial to be very familiar with the dimensions of your tensors and what they mean, so play around with the tensors, look at their shapes, and make sure you know why they are shaped the way that they are. This will help in debugging your neural network if you need to.

Training the network

Now, we're ready to define our network and train our model on the data. For this assignment, let's run some experiments training our own embedding layer, and some using pretrained GloVe embeddings.

5. Create a neural network in PyTorch according to the following specifications:

    - An embedding layer (please use `nn.Embedding` here) which takes a sequence of s indices as input and outputs an embedding matrix sized $s \times d$, where d is the dimensionality of the word embeddings.
    - The embedded sequence is then *pooled* so that the sequence is condensed into a single d-dimension representation: for our purposes, let's do a simple average over all embeddings in the sequence, no learnable parameters.
    - The d-dimension representation is fed into a linear layer with input size d and output size h, then a ReLU activation, and then finally a second linear layer which has an input size h and output size 1.
    - Finally, perform sigmoid on the single logit to get a class probability.
    - **You should not need any other packages besides those already present in the scaffolding code. Limit your packages to those.**

6. Train and evaluate the network on F1 and accuracy. Feel free to use a package such as sklearn for metrics.

    - *Note: Training should not take long at all, even on a CPU, to achieve an F1 score above .70. Try a batch size of 16 for 10 epochs with an embedding layer of size 50, and a hidden layer of size 50: if you're not getting at least .70 F1, with the validation loss decreasing at least modestly, and the training loss decreasing significantly throughout the training, there is probably a bug somewhere in your code.*

7. Modify your network so that it accepts pre-trained embeddings for the embedding layer, and train another network on GloVe embeddings of the same dimensionality you used for #6.

You can find significant scaffolding for this code in `a3-net-scaffolding.py`.

Part 2: Analyze the Word Embeddings

Before analyzing our trained word embeddings, ensure that you're getting the following results:

- The training loss decreases significantly (i.e. from 0.65 to 0.03) throughout the training.
- The validation loss decreases at least modestly (i.e. from 0.65 to 0.60) throughout the training.
- The trained model achieves at least 0.70 F1 on a held-out test set.

If you are achieving all of those things, then you can be pretty sure your network is (1) learning *something* (the loss decreases) and (2) whatever it's learning can generalize, at least a little bit, to unseen data (the test F1 is significantly higher than random chance).

We can try to get a better idea about exactly *what* it's learning by analyzing the word embeddings it's learning during training time. We can extract these embeddings directly from the trained model:

```
embeddings = model.embedding.weight.data
```

We can look at the embedding for a single word, for example, the word "good":

```
In: embeddings[token2index['good']]
Out: tensor([-1.1137,  0.3538, -0.1122,  0.8031, -0.5461, ...])
```

But clearly, it's not exactly easy to interpret a word embedding by itself. We're only going write a little bit more code, and then you can use it to explore your embeddings freely, and use what you learn for your written report.

8. Implement a k-nearest-neighbors function that takes an embedding matrix, a token, a token-to-index dict, and an integer k, and returns the k nearest neighbors in the embedding space, using cosine similarity as the distance function:

   ```
   In: k_nearest_neighbors(embeddings, token2index, 'good', k=5)
   Out: ['good', 'nonconformist', 'ambitious', 'gnashing', 'blunt']
   ```

   Do not import a package to calculate the k nearest neighbors. Write your own implementation. You may use a package to calculate cosine distance.

9. Experiment with plotting your trained embeddings, versus the pre-trained GloVe embeddings. We've provided the plotting code for you in `a3-explore-scaffolding.py`; just ensure that you are able to run it with your embeddings so that you can use the plots in your report.

Packages

These are the only import statements you are explicitly allowed to use in this assignment. If you want to use another package, please seek permission on Piazza first.

```
# Preprocessing
import pandas as pd
import re
from nltk.stem import WordNetLemmatizer
import nltk
nltk.download('wordnet')
nltk.download('punkt')

# Converting to tensors and dataloaders
import torch
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader
from sklearn.preprocessing import LabelEncoder
```

```
from torch.utils.data import TensorDataset, DataLoader

# Neural net design and training
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score
import gensim.downloader as api # Only for GloVe embeddings

# Word embedding exploration
from sklearn.metrics.pairwise import cosine_similarity
# Besides cosine_similarity, DO NOT IMPORT ANY FUNCTION TO FIND THE K
# NEAREST NEIGHBORS, YOU ARE REQUIRED TO WRITE YOUR OWN
# The statements below are in the provided plotting functions that
# you shouldn't need to change:
from nltk.corpus import stopwords
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
```

Written Report

The last item you will be required to turn in is a written report (minimum 1 page) answering the following questions about this assignment and the concepts it is meant to cover.

1. Run a few experiments with different embedding dimensions, batch sizes, and number of epochs run. Try with and without the pre-trained GloVe embeddings. Across your experiments, did the pre-trained embeddings work better, or was it better to train your own embedding layer for this task? Did you notice any trends with certain parameters? What happens if you freeze the GloVe embeddings during training time?

2. Compare your own embeddings, trained on the Rotten Tomatoes task, against the more general-purpose GloVe embeddings. Choose some words you think might be interesting: for example, since this is a dataset of movie reviews, the words "good" and "bad" might be interesting, since what is considered "good" in the movie domain may differ significantly from the word "good" more generally. Find the nearest neighbors for the words for both your own embeddings, and the GloVe embeddings. What do you notice about the differences in the vector space between your own embeddings and the pre-trained embeddings?

3. See if you can make any interesting or illuminating visualizations of your own embeddings with the two `plot_embeddings` functions in the `a3-explore-scaffolding.py` file. If none of the visualizations make any sense to you, explain why you're surprised by what you see, and what you would've expected instead. Speculate whether it's because of the idiosyncrasy of the data, or if it's because the embeddings haven't been trained to optimally represent language, or something else entirely. Then, compare and contrast with visualizations of the GloVe embeddings.

Rubric

Base Points	Item
20	Data preprocessing
20	Training the network
20	Analyzing the word embeddings
20	Written report
Point Multipliers	**Criteria**
1	All expected functionality is present. Report answers questions in a near-perfect manner.
0.75	Minor errors not impacting general functionality. Output may slightly deviate from what is expected. Report answers questions well with minor errors.
0.5	Errors demonstrating a lack of understanding. Some functionality missing. Report demonstrates lack of understanding.
0.25	Significant errors affecting the functionality of the item. Report has significant errors and does not answer questions in an acceptable manner.
0	No work present or code does not run. No report.