Assignment 4: Sentiment Analysis using Feedforward Neural Networks

Ellie Larence, Matthew Lloyd Macias, Jameson Tenorio, Leo Wang

April 25, 2025

2 Base FNN Model

The base FNN model constructed by Matthew Macias consists of 3 hidden layers, where the first two hidden layers have half the number of the initial input layer, and the final hidden layer with a fourth of the initial input layer. The FNN applies a linear transformation, on each layer, using ReLU as the activation function. The dataset provided to the FNN was first split into a 70-30 training-testing data split, and then vectorized using tfidf vectorization utilizing the preprocessing and Porter tokenization functions shown in the textbook and during class on the slides. The testing and training features were then initially normalized before being passed into a tensor dataset that would afterwards be utilized by the FNN. The labels were untouched, but split in the same manner as the features. The optimizer and loss function our FNN uses is the SGD optimizer, and the Cross Entropy Loss function. No L2 Regularization parameter or learning rate was specified at this time, as to be able to tune these in Task 3. Our Base FNN also only trains over 5 epochs.

The split was modified by Leo Wang to ensure correct splitting of training and testing data. Leo Wang also additionally added the total and average loss calculations, as well as time measurement to the training loop. This Base FNN is found in the `generate_data.py` script provided in the Assignment zip file, and outlined here in the following code snippet.

```
import pyprind
import pandas as pd
import numpy as np
import os
import sys
import re
import time
from sklearn.metrics import accuracy_score
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# Don’t think these will be used for me (Matt)
# But maybe for others
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torch.utils.data import TensorDataset

import nltk
nltk.download(’stopwords’)

from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer

# Global seeds
torch.manual_seed(42)
np.random.seed(42)

# Global variables
csv_file_name = ’./movie_data.csv’
stop = stopwords.words(’english’)
porter = PorterStemmer()

```

```python
# Generates a dataframe object from the movie reviews.
# returns
#   - df - A dataframe object containing all reviews and their sentiments
def generate_df_from_reviews():
    # If the csv file has not been generated yet,
    # generate from the aclImdb directory.
    if not (os.path.exists(csv_file_name)):
        current_dir = os.path.dirname(__file__)

        data_dir_name = "aclImdb"
        dataset_dir = os.path.join(current_dir, data_dir_name)
        labels = {'pos':1, 'neg':0}
        # pbar can be commented out
        pbar = pyprind.ProgBar(50000, stream=sys.stdout)
        df = pd.DataFrame()
        for s in ('test', 'train'):
            for l in ('pos', 'neg'):
                path = os.path.join(dataset_dir, s, l)
                for file in sorted(os.listdir(path)):
                    with open(os.path.join(path, file), 'r', encoding='utf-8') as infile:
                        txt = infile.read()
                    df_extension = pd.DataFrame([[txt, labels[l]]])
                    df = pd.concat([df, df_extension], ignore_index=True)
                    # pbar can be commented out
                    pbar.update()

        # Label columns
        df.columns = ['review',  'sentiment']
        # Shuffle
        df = df.reindex(np.random.permutation(df.index))
        # Store to csv
        df.to_csv(csv_file_name, index=False, encoding='utf-8')
    # Else, data has already been written, so read in from existing csv
    else:
        df = pd.read_csv(csv_file_name, encoding='utf-8')
        df = df.rename(columns={"0": "review", "1": "sentiment"})

    # Return resulting dataframe. Generated from seed 42
    return df


# Preprocessor to clean the data (From Textbook/slides)
def preprocessor(text):
    text = re.sub('<[^>]*>', '', text)
    # emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)
    # text = (re.sub('[\W]+', ' ', text.lower()) + ' '.join(emoticons).replace('-', ''))
    # adding r for raw string to circumvent SyntaxWarning
    emoticons = re.findall(r'(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)
    text = (re.sub(r'[\W]+', ' ', text.lower()) + ' '.join(emoticons).replace('-', ''))
    return text


# Tokenizer (From Textbook/slides)
def tokenizer_porter(text):
    return [porter.stem(word) for word in text.split()]
```

```python


# Process the dataframe generated from the reviews
def process_data():
    start_time = time.time()

    # Generate DataFrame Object where data is stored
    print("\n1. Loading dataset...")
    df = generate_df_from_reviews()
    print(f"Total number of reviews in dataset: {len(df)}")

    # 70-30 split
    # X_train = df.loc[:35000, 'review'].values
    # X_test = df.loc[15000:, 'review'].values
    # y_train = df.loc[:35000, 'sentiment'].values
    # y_test = df.loc[15000:, 'sentiment'].values
    # ^ Old, bad split, caused very bad accuracy ^
    # data leakage due to overlap from row 15000 through 35000 between training and testing data!

    # New, proper 70-30 split from Leo Wang
    split_idx = 35000
    X_train = df.loc[:split_idx-1, 'review'].values
    X_test = df.loc[split_idx:, 'review'].values
    y_train = df.loc[:split_idx-1, 'sentiment'].values
    y_test = df.loc[split_idx:, 'sentiment'].values

    print(f"Training set size: {len(X_train)}")
    print(f"Test set size: {len(X_test)}")

    # Limit max features to roughly half of what it
    # would originally generate with this vectorizer
    # Change max_features to include more features for
    # processing. 50000 features went to about 2-3 hours
    # Original size from a very initial test was around
    # 100000 features. If you change this, change the scale
    # of the FNN and change the input in net.forward()
    # to also match this number.
    tfidf = TfidfVectorizer(max_features=10000, strip_accents=None, lowercase=False, preprocessor=preprocessor, tokenizer=tokenizer_porter, stop_words='english')

    # Vectorize the review data and turn it into something
    # that the TensorDataset can ingest
    # Also vectorize test data to test on finished model
    tfidf_reviews = tfidf.fit_transform(X_train).toarray()

    # tfidf_testing = tfidf.fit_transform(X_test).toarray()
    tfidf_testing = tfidf.transform(X_test).toarray()
    # doesn't need to fit the vectorizer again, test data should be vectorized using the same vocab and transform learned from the training data

    tfidf_reviews_norm = (tfidf_reviews - np.mean(tfidf_reviews)) / np.std(tfidf_reviews)
    tfidf_testing_norm = (tfidf_testing - np.mean(tfidf_testing)) / np.std(tfidf_testing)

    # Float for the reviews, as the array is likely to have floats rather than ints
```

```python
    # long for the sentiments to properly gauge positive or negative
    tfidf_reviews_norm = torch.from_numpy(tfidf_reviews_norm).float()
    tfidf_testing_norm = torch.from_numpy(tfidf_testing_norm).float()
    y_train = torch.from_numpy(y_train).long()


    # Initialize the dataset and load it into the dataloader
    train_ds = TensorDataset(tfidf_reviews_norm, y_train)
    review_data = DataLoader(train_ds, batch_size=100, shuffle=True)


    # initialize FNN. Model based off of Linear ReLU from class
    print("\n4. Initializing neural network...")
    net = torch.nn.Sequential(
        torch.nn.Linear(10000, 5000),
        torch.nn.ReLU(),
        torch.nn.Linear(5000, 5000),
        torch.nn.ReLU(),
        torch.nn.Linear(5000, 5000),
        torch.nn.ReLU(),
        torch.nn.Linear(5000, 2500),
        torch.nn.ReLU(),
        torch.nn.Linear(2500, 2),
        torch.nn.Softmax(dim=1))

    optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
    L = torch.nn.CrossEntropyLoss()

    # Train. It's doing something and it takes a while. I got to take a shower and it barely finished the
    # Second epoch. So to finish 5 epochs, likely 1 hour and 30 minutes.
    print("\n5. starting training...")
    print("Epoch | Training Loss | Test Accuracy | Time")
    print("-" * 40)

    for epoch in range(5):
        epoch_start = time.time()
        net.train()
        total_loss = 0
        batch_count = 0

        for (x, y) in review_data:
            output = net.forward(x.view(-1,10000))
            loss = L(output, y)
            total_loss += loss.item()
            batch_count += 1
            loss.backward()
            optimizer.step()
            net.zero_grad()

        avg_loss = total_loss / batch_count

        net.eval()
        with torch.no_grad():
            y_pred = net(tfidf_testing_norm)
```

```
            y_pred = torch.argmax(y_pred, dim=1)
            acc = accuracy_score(y_test, y_pred)
            epoch_time = time.time() - epoch_start
            print(f"{epoch+1:5d} | {avg_loss:.6f} | {acc:.6f} | {epoch_time:.2f}s")

    total_time = time.time() - start_time
    print(f"\n Training completed in {total_time/60:.2f} minutes")
    print(f"Final test accuracy: {acc:.4f}")


if __name__ == "__main__":
    process_data()
```

3 Hyperparameter Tuning Results and Analysis

Utilizing the FNN provided by the previous task we attempted to find the learning rate and L2 regularization that produced the best accuracy. The learning rates tested were [0.0001, 0.001, 0.01, 0.1] with L2 regularization's of [0, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1]. Due time and hardware limitations feature sizes of 2500, 5000, 10,000 were run using these learning rates and L2 regularization's for 10 epochs. Once the best learning rate was identified and top two L2 regularization's considered, runs using 20,0000 and 50,000 features were made to see if we could gain a better accuracy using larger feature sets. We were able to determine that a learning rate of 0.01 with an L2 regularization of 0.01 performed best for our FNN. Below is a table with accuracies and runtime per epoch for each feature size. We decided to compare accuracies at epoch 8 as it took 396.15 minutes for the 50,000 feature run on our FNN and capped that feature set there.

Table 1: Final Hyperparameter Tuning for FNN: Learning Rate: 0.01, L2-Regularization 0.01

Feature Size	Accuracy (%) at Epoch = 8	Time Range per Epoch (s)
2500	0.8713	11-12 (s)
5000	0.8847	33-38 (s)
10000	0.8899	145-190 (s)
20000	0.8949	560-590 (s)
50000	0.8950	2980-3100 (s)

In Table 1 we can see that as we increase the number of features with the tuned hyperparameter's our accuracy increases. However, the cost per epoch also increase drastically. With enough time and hardware to use the full feature set it may be possible to get over 90% accuracy using our FNN. However, by the time we reached 50,000 features we were at or near hardware capacity for the computers used.
A slight modification of the textbooks logistic regression to use PyTorch and tensors for the dataset was made but the same structure was maintained from the textbook implementation. For the logistic regression it was found that it had the best accuracy at 500 iterations (iterations tested were [10, 100, 500, 1000, 2000]). So using the same learning rate of 0.01 at 500 iterations we have the following table for logistic regression.

Table 2: Logistic Regression: Learning Rate: 0.01, Iterations = 500

Feature Size	Accuracy (%)	Total Time Cost (s)
2500	0.8830	5.4 (s)
5000	0.8891	12.6 (s)
10000	0.8891	25.8 (s)
20000	0.8889	46.2 (s)
50000	0.8816	103.8 (s)

The run time of the logistic regression is several times faster that the FNN. However, the accuracy does not seem to be impacted by the number of features add. It seems to swing in the range of 88 to 89 percent and would infer that the accuracy is capped at 89% for the logistic regression model of this dataset.

4 K-Fold Cross Validation Analysis

4.0.1 Impact of k-Fold Cross Validation

For the baseline configuration ($K = 1$), the network attains 88.28 % accuracy in 3.20 s. Introducing k-fold cross validation improves accuracy only marginally—at $K = 5$, the model rises to 88.25 % and at $K = 10$, it converges near 88.65 %. However, wall-clock time increases almost linearly with K, as shown in Table 3. This test case demonstrates that although cross validation yields a slightly more reliable estimate of performance, it imposes an immense overhead that is seldom justified once the architecture and hyper-parameters have been fixed.

Table 3: K-Fold Cross-Validation with the Original Model: Time Cost and Mean Accuracy

K	Accuracy (%)	Time Cost (s)
1	0.8828	3.20
2	0.8758	5.10
3	0.8790	9.25
4	0.8804	14.90
5	0.8825	19.78
6	0.8815	23.04
7	0.8803	27.34
8	0.8828	32.73
9	0.8837	37.86
10	0.8865	43.92
11	0.8866	49.12
12	0.8867	54.56
13	0.8864	59.20

```
def create_model():
    return nn.Sequential(
        nn.Linear(INPUT_SIZE, 1024),
        nn.ReLU(),
        nn.Linear(1024, 512),
        nn.ReLU(),
        nn.Linear(512, 128),
        nn.ReLU(),
        nn.Linear(128, 2)
    )
```

4.0.2 Smaller Model vs. Original Model

After experimenting with different builds, The tuned architecture delivers comparable or better accuracies while roughly halving execution time at every K (e.g., 21.76 s vs. 43.92 s at $K = 10$). This behavior could be due to fewer parameters shortening forward/backward passe, directly cutting runtime. In addition, the lower capacity mitigates over-fitting on the features, allowing validation accuracy to match or even slightly exceed that of the deeper original network.

Table 4: K-Fold Cross-Validation with a Tuned Model: Time Cost and Mean Accuracy

K	Accuracy (%)	Time Cost (s)
1	0.8830	2.28
2	0.8758	3.32
3	0.8759	5.50
4	0.8848	7.84
5	0.8840	10.16
6	0.8855	12.49
7	0.8783	14.79
8	0.8857	17.17
9	0.8841	19.42
10	0.8865	21.76
11	0.8868	24.05
12	0.8869	26.14
13	0.8865	31.52

5 Training using Dropout Regularization

To add dropout to our FNN, we used PyTorch's Dropout function to dropout following the last ReLU layer. The dropout probability was set to 0.5 for all models trained. Below is the function that generated the FNN with dropout.

```
def gen_single_dropout():
    print("\n4.␣Initializing␣neural␣network...")
    net_single_dropout = torch.nn.Sequential(
        torch.nn.Linear(10000, 5000),
        torch.nn.ReLU(),
        torch.nn.Linear(5000, 5000),
        torch.nn.ReLU(),
        torch.nn.Linear(5000, 5000),
        torch.nn.ReLU(),
        torch.nn.Linear(5000, 2500),
        torch.nn.ReLU(),
        torch.nn.Dropout(p=0.5),
        torch.nn.Linear(2500, 2),
        torch.nn.Softmax(dim=1))

    return net_single_dropout
```

The same training code used to build the baseline model was used to train the single dropout model. For the set of bagged dropout models, the script below was used to train five models and then ensemble them.

```
print("===␣Bagging␣5␣Dropout␣Models␣Training␣===")
bag_accs = []
bag_times = []
for i in range(5):
    print(f"--␣Training␣model␣{i+1}␣--")
    model = gen_single_dropout()
    acc, t = train(model, train_loader, X_test, y_test)
    bag_accs.append(acc)
    bag_times.append(t)
```

```


bag_acc = [sum(epoch_accs) / len(bag_accs) for epoch_accs in zip(*bag_accs)]
total_bag_time = sum(bag_times)
print(f"Bagging total time: {total_bag_time/60:.2f} minutes, Final Ensemble Avg Acc: {bag_acc[-1]:.4f}\n")
```

Model Version	Training Time (minutes)
Baseline	13.83
Single Dropout	12.91
Bagging (5 Dropout Models)	52.32

Table 5: Comparison of training times for different model configurations to 12 epochs. Max feature input was set to 10000.

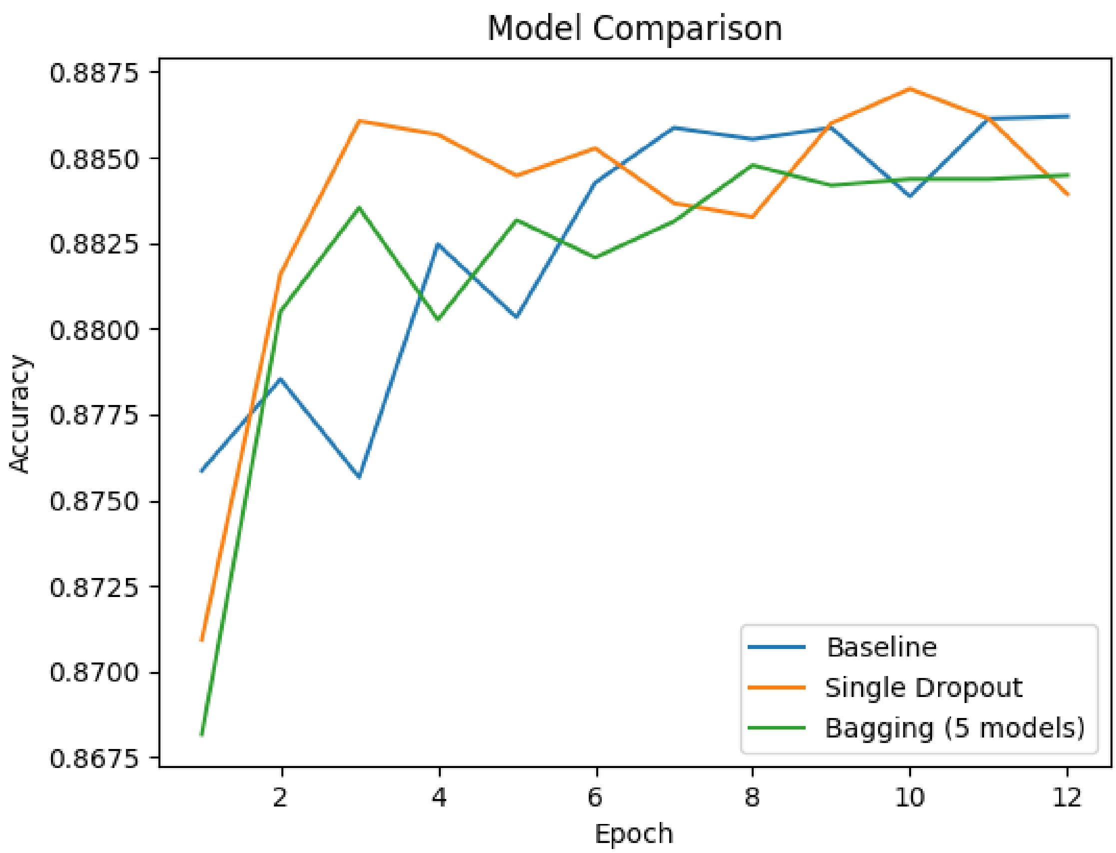

Figure 1: Accuracy over epochs for model variants. Max feature input was set to 10000.

6 Team Member Contributions

Below are the contributions of each group member.

Ellie Larence: Worked on Task 5, and helped give clarity on Tasks 2, 3, and 4.

Matthew Lloyd Macias: Worked on tasks 1, 2, and ran a few supplementary experimental tests for Task 3.

Jamerson Tenorio: Worked on Task 3, reformatting the original `generate_data.py` script to be more modular and be able to experiment with variable defined hyperparameters for tuning.

Leo Wang: Worked on Task 4, as well as provided some needed fixes to the original `generate_data.py` script.

References

[1] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL: `http://www.aclweb.org/anthology/P11-1015`.

[1]