CSCI 544 – Applied Natural Language Processing

Homework 3 – Vu Truong Si – 6031936649

Python version used: 3.8.8

PyTorch version used: 1.13.0+cpu

.ipynb file run time: Approximately 145 minutes:

- Data preparation and Word2Vec loading/training: 10 minutes
- SVM and Perceptron: 5 minutes
- FNN: 10 minutes (5 minutes each part)
- RNN, GRU, LSTM: 40 minutes each part (48 seconds per epoch – 50 epochs)

1. Dataset Generation

Similar to Homework 1, I read the dataset into a dataframe using Pandas. During data reading, I skipped bad lines by setting the parameter **error_bad_line** to False.

To keep the Reviews and Ratings, I only selected the two fields **review_body** and **star_rating**. I converted the ratings into readable numerical values, then encoded them as instructed using the following map {1:1, 2:1, 3:2, 4:3, 5:3}.

I implemented 6 different data cleaning techniques to prepare a high-quality dataset for Word2Vec model training:

- Convert reviews into lowercase: I used **.lower()** to achieve this.
- Remove HTML and URLs from the reviews: I used regular expression to eliminate the patterns of HTML tags and URLs. For example, “**<[^<]+?>**” will target all the HTML tags and I can remove them from the text.
- Expand contractions: I used the library **contractions**, specifically **contractions.fix()** to perform this technique.
- Remove non-alphabetical characters: I also used regular expression to achieve this. I kept all the characters between a-z and A-Z.
- Remove extra spaces: I combined regular expression with **.strip()** to remove extra spaces in the reviews.
- Remove review without content.

Afterward, I sampled 60000 reviews, 20000 from each class, from the cleaned dataset.

2. Word Embedding

a. I used **downloader.load('word2vec-google-news-300')** to load Google’s model, then performed similarity checks on some words like **okay** and **alright**.

b. To train my own Word2Vec model, I turned the reviews into a list of sentences and feed that into the Word2Vec module. I set the parameters as required: window = 13, vector_size = 300, min_count = 9.

3. Simple models

In order to get the average vectors, I initialize an empty array of length 300 then add the Word2Vec vectors of each word in a review to it. In the end, I divide that array by the number of vectors added.

For TF-IDF, I extracted the features using TfidfVectorizer, similar to Homework 1.

I used Perceptron and LinearSVC for the models. Below are the accuracies:

```
Accuracy for Perceptron using Word2Vec vectors: 0.59375
Accuracy for SVM using Word2Vec vectors: 0.6615833333333333
Accuracy for Perceptron using TF-IDF vectors: 0.6898333333333333
Accuracy for SVM using TF-IDF vectors: 0.7235
```

4. Feedforward Neural Networks (FNN)

a. I initialize a feedforward multilayer perceptron with 2 hidden layers (100 and 10 nodes). I used cross entropy loss and SGD optimizer. After that, I feed the data from question 3 into it. My parameters are: batch_size = 20, learning_rate = 0.01, n_epochs = 50.

b. The architecture is similar to part a, but the dataset is different. I concatenate the first 10 Word2Vec vectors of every review, with padding and use them as the input for the model.

Below are the accuracies of these 2 models:

```
Accuracy for FNN using average Word2Vec vectors: 0.6564166666666666
Accuracy for FNN using 10 first Word2Vec vectors: 0.5546666666666666
```

5. Recurrent Neural Networks (RNN)

a. I used the vectors for the first 20 words in a review for the input of this question. I initialize an RNN model with hidden_state = 20, batch_size = 24, learning_rate = 0.001, n_epochs = 50 then feed the data into it.

b. Similar to part a, except I replaced the RNN with a Gated Recurrent Unit.

c. Similar to part c, except I replace the RNN with a Long Short-Term Memory unit cell.

Below are the accuracies of these 3 models:

```
Accuracy for RNN using 20 first Word2Vec vectors: 0.61175
Accuracy for GRU using 20 first Word2Vec vectors: 0.6403333333333333
Accuracy for LSTM using 20 first Word2Vec vectors: 0.6334166666666666
```

In [1]:
```
import pandas as pd
from gensim import corpora, models, similarities, downloader
from gensim.models import Word2Vec
import re
import contractions
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
import torch
from torch.utils.data import DataLoader, Dataset
import torch.nn as nn
import torch.nn.functional as F
from numpy import dot
from numpy.linalg import norm
```

1. Dataset Generation

We will use the Amazon reviews dataset used in HW1. Load the dataset and build a balanced dataset of 60K reviews along with their ratings to create labels through random selection similar to HW1. You can store your dataset after generation and reuse it to reduce the computational load. For your experiments consider a 80%/20% training/testing split.

In [3]:
```
# Read the data while skipping bad lines.

dataframe = pd.read_table("amazon_reviews_us_Beauty_v1_00.tsv", error_bad_lines = False, warn_bad_lines=False
```

In [4]:
```
reviews_and_ratings = dataframe[["review_body", "star_rating"]]
```

In [5]:
```
# Convert string to numerical values ("1.0" -> 1.0) and mark as None if the value is invalid.

reviews_and_ratings.loc[: , "star_rating"] = pd.to_numeric(reviews_and_ratings["star_rating"], errors = 'coer
```

In [6]:
```
# Drop null values.

reviews_and_ratings = reviews_and_ratings.dropna(how = "any")
```

In [7]:
```
reviews_and_ratings
```

Out[7]:

	review_body	star_rating
0	Love this, excellent sun block!!	5.0
1	The great thing about this cream is that it do...	5.0
2	Great Product, I'm 65 years old and this is al...	5.0
3	I use them as shower caps & conditioning caps....	5.0
4	This is my go-to daily sunblock. It leaves no ...	5.0
...	...	...
5094302	After watching my Dad struggle with his scisso...	5.0
5094303	Like most sound machines, the sounds choices a...	3.0
5094304	I bought this product because it indicated 30 ...	5.0
5094305	We have used Oral-B products for 15 years; thi...	5.0
5094306	I love this toothbrush. It's easy to use, and ...	5.0

5093907 rows × 2 columns

In [8]:
```
ratings_dict = {1:1, 2:1, 3:2, 4:3, 5:3}
```

In [9]:
```
# Map the ratings to appropriate classes.

reviews_and_ratings.loc[:,"class"] = reviews_and_ratings["star_rating"].map(ratings_dict)
```

In [10]:
```
reviews_and_ratings
```

Out[10]:

	review_body	star_rating	class
0	Love this, excellent sun block!!	5.0	3
1	The great thing about this cream is that it do...	5.0	3
2	Great Product, I'm 65 years old and this is al...	5.0	3
3	I use them as shower caps & conditioning caps....	5.0	3
4	This is my go-to daily sunblock. It leaves no ...	5.0	3
...	...	...	...
5094302	After watching my Dad struggle with his scisso...	5.0	3
5094303	Like most sound machines, the sounds choices a...	3.0	2
5094304	I bought this product because it indicated 30 ...	5.0	3
5094305	We have used Oral-B products for 15 years; thi...	5.0	3
5094306	I love this toothbrush. It's easy to use, and ...	5.0	3

5093907 rows × 3 columns

In [11]:
```
# Preprocesisng.

# Convert text to lowercase.
reviews_and_ratings["review_body"] = reviews_and_ratings["review_body"].str.lower()

# Remove HTML tags.
reviews_and_ratings["review_body"] = [re.sub('<[^<]+?>', '', str(x)) for x in reviews_and_ratings["review_bod

# Remove URLs.
reviews_and_ratings["review_body"] =  [re.sub(r"http\S+","", str(x)) for x in reviews_and_ratings["review_bod

# Expand contractions.
reviews_and_ratings["review_body"] = [contractions.fix(str(x)) for x in reviews_and_ratings["review_body"]]

# Remove non-alphabetical characters.
reviews_and_ratings["review_body"] = [re.sub(r"[^a-zA-Z ]", "", str(x)) for x in reviews_and_ratings["review_

# Remove excess spaces.
reviews_and_ratings["review_body"] = reviews_and_ratings["review_body"].replace("\s+", " ", regex = True).str
```

In [12]:
```
# Compute the length of each review.

reviews_and_ratings["review_length"] = [len(str(x)) for x in reviews_and_ratings["review_body"]]
```

In [13]:
```
# Drop reviews without content.

reviews_and_ratings = reviews_and_ratings[reviews_and_ratings["review_length"] > 0]
```

In [14]:
```
# Randomly sample 20000 rows from each class.

class_1 = reviews_and_ratings[reviews_and_ratings["class"] == 1].sample(20000)
class_2 = reviews_and_ratings[reviews_and_ratings["class"] == 2].sample(20000)
class_3 = reviews_and_ratings[reviews_and_ratings["class"] == 3].sample(20000)
```

In [15]:
```
balanced_dataset = pd.concat([class_1, class_2, class_3], axis = 0)
```

In [16]:
```
balanced_dataset
```

Out[16]:

	review_body	star_rating	class	review_length
3055819	i have purchased these before and i was always...	2.0	1	454
1662879	if this works you would not know it from my sk...	2.0	1	498
1443692	very flimsy not great	2.0	1	21
4395615	this shampoo is very runny i runs through my f...	1.0	1	161
206392	did not work for me at all left flakes in my h...	2.0	1	122
...	...	...	...	...
5042414	i am sure this nightguard is much better than ...	4.0	3	380
4535600	love this productit does exactly what it promi...	5.0	3	122
2114896	works exactly as advertised and perfect size f...	5.0	3	101
1890940	great thank you	5.0	3	15
2504270	this is my favorite curling tool i get really ...	5.0	3	145

60000 rows × 4 columns

In [17]:
```
# Export to csv for local processing.

# balanced_dataset.to_csv("balanced_dataset.csv", index = False)
```

In [18]:
```
# balanced_dataset = pd.read_csv("balanced_dataset.csv")
```

2. Word Embedding

In this part the of the assignment, you will generate Word2Vec features for the dataset you generated. You can use Gensim library for this purpose. A helpful tutorial is available in the following link: [https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html (https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html)](https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html)

(a) Load the pretrained “word2vec-google-news-300” Word2Vec model and learn how to extract word embeddings for your dataset. Try to check semantic similarities of the generated vectors using three examples of your own, e.g., King −Man +Woman = Queen or excellent ~ outstanding.

In [19]:
```
# Import Google's Word2Vec model.

wv = downloader.load('word2vec-google-news-300')
```

Example 1: okay is most similar to **alright**

In [20]:
```
# okay ~ alright
wv.similarity("okay", "alright")
```

Out[20]:
```
0.8877537
```

In [21]:
```
# okay ~ alright
wv.most_similar("okay")
```

Out[21]:
```
[('alright', 0.8877537250518799),
 ('ok', 0.8567795753479004),
 ('OK', 0.7831767201423645),
 ('yeah', 0.6638472676277161),
 ('allright', 0.6537948250770569),
 ('hey', 0.603424608707428),
 ('say_Hey_feller', 0.5872976779937744),
 ('anyway', 0.5856852531433105),
 ('anyways', 0.5801851153373718),
 ('maybe', 0.5797765851020813)]
```

Example 2: terrible is most similar to **horrible**

In [22]:
```
# terrible ~ horrible
wv.similarity("terrible", "horrible")
```

Out[22]:
```
0.92439204
```

In [23]:
```
# terrible ~ horrible
wv.most_similar("terrible")
```

Out[23]:
```
[('horrible', 0.9243921041488647),
 ('horrendous', 0.8467271327972412),
 ('dreadful', 0.8022766709327698),
 ('awful', 0.7478912472724915),
 ('horrid', 0.7179027199745178),
 ('atrocious', 0.689181387424469),
 ('horrific', 0.6830835342407227),
 ('bad', 0.6828612089157104),
 ('appalling', 0.6752808690071106),
 ('horrible_horrible', 0.6672273278236389)]
```

Example 3: hate is most similar to **despise**

In [24]:
```
# hate ~ despise
wv.similarity("hate", "despise")
```

Out[24]:
```
0.6712518
```

In [25]:
```
# hate ~ despise
wv.most_similar("hate")
```

Out[25]:
```
[('despise', 0.6712517142295837),
 ('Hate', 0.6400399804115295),
 ('detest', 0.6179037094116211),
 ('hatred', 0.6156139969825745),
 ('hating', 0.6103581786155701),
 ('hates', 0.6091769933700562),
 ('HATE', 0.6020098328590393),
 ('dislike', 0.6013234853744507),
 ('love', 0.600395679473877),
 ('hated', 0.5922116637229919)]
```

(b) Train a Word2Vec model using your own dataset. You will use these extracted features in the subsequent questions of this assignment. Set the embedding size to be 300 and the window size to be 13. You can also consider a minimum word count of 9. Check the semantic similarities for the same two examples in part (a). What do you conclude from comparing

vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better? For the rest of this assignment, use the pretrained “word2vec-googlenews-300” Word2Ve features.

In [26]:
```
# Get reviews.

reviews = balanced_dataset["review_body"].values
```

In [27]:
```
# Create a list of sentences to train the custom Word2Vec model.

sentences = []
for review in reviews:
    temp = []
    word_list = review.split(" ")
    for word in word_list:
        temp.append(word)
    sentences.append(temp)
```

In [28]:
```
# Train the Word2Vec model using the sentences.

w2v_model = Word2Vec(sentences = sentences, window = 13, vector_size = 300, min_count = 9)
```

Example 1: okay vs **alright**

Here we can see that **okay** is most similar to **ok** in the new w2v model, this is because the training data of the two models are different, but the next closest word is still **alright**, so our model seems to work well with the first example. We can also see other candidates for okay such as **awesome**, **fantastic**, etc.

In [29]:
```
# okay test
w2v_model.wv.most_similar("okay")
```

Out[29]:
```
[('ok', 0.9183574914932251),
 ('alright', 0.7863861322402954),
 ('fine', 0.6159654855728149),
 ('good', 0.5993782877922058),
 ('awesome', 0.572403609752655),
 ('great', 0.5683795809745789),
 ('amazing', 0.5301942229270935),
 ('awful', 0.49639642238616943),
 ('decent', 0.49194318056106567),
 ('fantastic', 0.48155900835990906)]
```

Example 2: terrible vs **horrible**

Here we can see that **terrible** is most similar to **horrible** in our newly trained model. Some other candidates are **awful**, **weird**, etc.

In [30]:
```
# good test
w2v_model.wv.most_similar("terrible")
```

Out[30]:
```
[('horrible', 0.8330451250076294),
 ('awful', 0.7492370009422302),
 ('wonderful', 0.6612504124641418),
 ('fantastic', 0.6514009237289429),
 ('awesome', 0.6283658742904663),
 ('amazing', 0.624359667301178),
 ('strange', 0.6114692091941833),
 ('funny', 0.5741326212882996),
 ('nasty', 0.5681740045547485),
 ('disgusting', 0.5679157972335815)]
```

King − Man + Woman = Queen and excellent ~ **outstanding.**

We can see that **excellent** is close to **outstanding**, however, the similarity between **king** and **queen** is very different from that of Google model.

In [31]:
```
w2v_model.wv.most_similar("excellent")
```

Out[31]:
```
[('outstanding', 0.7779552936553955),
 ('awesome', 0.6842052936553955),
 ('terrific', 0.6772099137306213),
 ('affordable', 0.6662511229515076),
 ('fabulous', 0.6584552526473999),
 ('fantastic', 0.639548659324646),
 ('amazing', 0.6251711249351501),
 ('attractive', 0.5944375991821289),
 ('inexpensive', 0.5640001893043518),
 ('great', 0.5562418103218079)]
```

In [32]:
```
king_man_woman_custom = w2v_model.wv["king"] - w2v_model.wv["man"] + w2v_model.wv["woman"]
king_man_woman_google = wv["king"] - wv["man"] + wv["woman"]
```

In [33]:
```
# queen vs king - man + woman using custom w2v.

dot(king_man_woman_custom, w2v_model.wv["queen"]) / (norm(king_man_woman_custom) * norm(w2v_model.wv["queen"]
```

Out[33]:
```
0.06652864
```

In [34]:
```
# queen vs king - man + woman using Google's w2v.
dot(king_man_woman_google, wv["queen"]) / (norm(king_man_woman_google) * norm(wv["queen"]))
```

Out[34]:
```
0.73005176
```

Conclusion

We can conclude that our model captured the meanings of adjectives nicely because reviews are usually written using lots of adjectives to express opinions. However, when it comes to nouns, our dataset did not have enough instances to cover them. That is why our results for words like **okay** and **terrible** are quite similar to those of the Google model, but results for **king** and **queen** are different. To sum up, Google's Word2Vec was still better at capturing semantic similarities due to their high quality training dataset.

3. Simple models

Using the Google pre-trained Word2Vec features, train a single perceptron and an SVM model for the classification problem. For this purpose, use the average Word2Vec vectors for each review as the input feature (x =1NPN i=1Wi for a review with N words). Report your accuracy values on the testing split for these models similar to HW1, i.e., for each of perceptron and SVM models, report two accuracy values Word2Vec and TF-IDF features. What do you conclude from comparing performances for the models trained using the two different feature types (TF-IDF and your trained 2 Word2Vec features)?

In [35]:
```
# Extract average Word2Vec vector for each review.
# I initialize an empty array of length 300, then add the Word2Vec vectors of each word in the review to it.
# Finally, I divide the array by the number of vectors added to it to get the average vector.

w2v_features_3 = []
for review in reviews:
    array = np.zeros(300)
    count = 0
    words = review.split(" ")
    for word in words:
        try:
            array = array + wv[word]
            count = count + 1
        except:
            continue
    if count == 0:
        w2v_features_3.append(array)
    else:
        w2v_features_3.append(array/count)
```

In [36]:
```
# Extract TF-IDF features for each review, similar to Homework 1.

tfidf_vectorizer = TfidfVectorizer(min_df = 0.0001, max_df = 0.5, ngram_range = (1,3))
tfidf_features = tfidf_vectorizer.fit_transform(balanced_dataset["review_body"])
```

In [37]:
```
# Function to calculate accuracy.

def accuracy(predictions, true):
    n = len(true)
    right = 0
    for i in range(len(true)):
        if true[i] == predictions[i]:
            right += 1
    return right / n
```

In [38]:
```
# Prepare datasets.

X_w2v = w2v_features_3
X_tfidf = tfidf_features
y = balanced_dataset["class"]
```

In [45]:
```
# Train test split.

X_train_w2v, X_test_w2v, y_train_w2v, y_test_w2v = train_test_split(X_w2v, y, test_size = 0.2)
X_train_tfidf, X_test_tfidf, y_train_tfidf, y_test_tfidf = train_test_split(X_tfidf, y, test_size = 0.2)
```

In [48]:
```
# Perceptron using Word2Vec embeddings.

perceptron_w2v = Perceptron(tol=1e-5, alpha = 0.001)
perceptron_w2v.fit(X_train_w2v, y_train_w2v)
y_pred_perceptron_w2v = perceptron_w2v.predict(X_test_w2v)
```

In [49]:
```
# SVM using Word2Vec embeddings.

svm_w2v = LinearSVC(C = 1.0, tol = 1e-3)
svm_w2v.fit(X_train_w2v, y_train_w2v)
y_pred_svm_w2v = svm_w2v.predict(X_test_w2v)
```

In [50]:
```
# Perceptron using TF-IDF embeddings.

perceptron_tfidf = Perceptron(tol=1e-5, alpha = 0.0001)
perceptron_tfidf.fit(X_train_tfidf, y_train_tfidf)
y_pred_perceptron_tfidf = perceptron_tfidf.predict(X_test_tfidf)
```

In [51]:
```
# SVM using TF-IDF embeddings.

svm_tfidf = LinearSVC(C = 1.0, tol = 1e-3)
svm_tfidf.fit(X_train_tfidf, y_train_tfidf)
y_pred_svm_tfidf = svm_tfidf.predict(X_test_tfidf)
```

In [52]:
```
print("Accuracy for Perceptron using Word2Vec vectors:", accuracy(y_pred_perceptron_w2v, y_test_w2v.values))
print("Accuracy for SVM using Word2Vec vectors:", accuracy(y_pred_svm_w2v, y_test_w2v.values))
print("Accuracy for Perceptron using TF-IDF vectors:", accuracy(y_pred_perceptron_tfidf, y_test_tfidf.values)
print("Accuracy for SVM using TF-IDF vectors:", accuracy(y_pred_svm_tfidf, y_test_tfidf.values))
```

```
Accuracy for Perceptron using Word2Vec vectors: 0.59375
Accuracy for SVM using Word2Vec vectors: 0.6615833333333333
Accuracy for Perceptron using TF-IDF vectors: 0.6898333333333333
Accuracy for SVM using TF-IDF vectors: 0.7235
```

Conclusion

TF-IDF seems to be better than Word2Vec embeddings for this problem. Both Perceptron and SVM showed better accuracy when they were using TF-IDF features.

4. Feedforward Neural Networks

Using the Word2Vec features, train a feedforward multilayer perceptron network for classification. Consider a network with two hidden layers, each with 100 and 10 nodes, respectively. You can use cross entropy loss and your own choice for other hyperparamters, e.g., nonlinearity, number of epochs, etc. Part of getting good results is to select suitable values for these hyperparamters. You can also refer to the following tutorial to familiarize yourself: [https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist (https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist)](https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist) Although the above tutorial is for image data but the concept of training an MLP is very similar to what we want to do.

(a) To generate the input features, use the average Word2Vec vectors similar to the “Simple models” section and train the neural network. Report accuracy values on the testing split for your MLP.

In [53]:
```
# Function to prepare datasets.

class ReviewDataset(Dataset):

    def __init__(self, data, transform=None):
        self.features = data.feature
        self.labels = data.label

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

    def __getitem__(self, index):

        feature = self.features[index]
        label = self.labels[index]

        return feature, label
```

In [54]:
```
# FNN architecture.

class FNN(nn.Module):
    def __init__(self, input_size, hidden_1 = 100, hidden_2 = 10):
        super(FNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 3)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x
```

In [55]:
```
# Prepare datasets. We will use the average Word2Vec vectors from question 3.

X_fnn = w2v_features_3.copy()
y = balanced_dataset["class"] - 1
```

In [56]:
```
X_train_fnn, X_test_fnn, y_train_fnn, y_test_fnn = train_test_split(X_fnn, y, test_size = 0.2)
```

In [57]:
```
train_data_fnn_df = pd.DataFrame(data = {"feature": X_train_fnn, "label": y_train_fnn}).reset_index(drop = Tr
test_data_fnn_df = pd.DataFrame(data = {"feature": X_test_fnn, "label": y_test_fnn}).reset_index(drop = True)
```

In [58]:
```
train_data_fnn = ReviewDataset(train_data_fnn_df)
test_data_fnn = ReviewDataset(test_data_fnn_df)
```

In [59]:
```
# Create data loaders.

batch_size = 20

train_loader_fnn = torch.utils.data.DataLoader(train_data_fnn, batch_size = batch_size)
test_loader_fnn = torch.utils.data.DataLoader(test_data_fnn, batch_size = batch_size)
```

In [60]:
```
# Create FNN model for 4a.

model_4a = FNN(300,100,10)
print(model_4a)
```

```
FNN(
  (fc1): Linear(in_features=300, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [61]:
```
# Initialize loss function and optimizer.

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_4a.parameters(), lr = 0.01)
```

In [62]:
```
# Model training.

n_epochs = 50
for epoch in range(n_epochs):
    for data, target in train_loader_fnn:
        optimizer.zero_grad()
        data = data.to(torch.float32)
        output = model_4a(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    print('Epoch: {}/{} --- Loss: {:.4f}'.format(epoch+1, n_epochs, loss.item()))
```

```
Epoch: 1/50 --- Loss: 1.0961
Epoch: 2/50 --- Loss: 1.0869
Epoch: 3/50 --- Loss: 1.0288
Epoch: 4/50 --- Loss: 0.9164
Epoch: 5/50 --- Loss: 0.8287
Epoch: 6/50 --- Loss: 0.7701
Epoch: 7/50 --- Loss: 0.7741
Epoch: 8/50 --- Loss: 0.7767
Epoch: 9/50 --- Loss: 0.7408
Epoch: 10/50 --- Loss: 0.7850
Epoch: 11/50 --- Loss: 0.6860
Epoch: 12/50 --- Loss: 0.7953
Epoch: 13/50 --- Loss: 0.6870
Epoch: 14/50 --- Loss: 0.6639
Epoch: 15/50 --- Loss: 0.6513
Epoch: 16/50 --- Loss: 0.6745
Epoch: 17/50 --- Loss: 0.6266
Epoch: 18/50 --- Loss: 0.7288
Epoch: 19/50 --- Loss: 0.6388
Epoch: 20/50 --- Loss: 0.6918
Epoch: 21/50 --- Loss: 0.5951
Epoch: 22/50 --- Loss: 0.6334
Epoch: 23/50 --- Loss: 0.6329
Epoch: 24/50 --- Loss: 0.6767
Epoch: 25/50 --- Loss: 0.6488
Epoch: 26/50 --- Loss: 0.6440
Epoch: 27/50 --- Loss: 0.5709
Epoch: 28/50 --- Loss: 0.6340
Epoch: 29/50 --- Loss: 0.6269
Epoch: 30/50 --- Loss: 0.5909
Epoch: 31/50 --- Loss: 0.6136
Epoch: 32/50 --- Loss: 0.5522
Epoch: 33/50 --- Loss: 0.6351
Epoch: 34/50 --- Loss: 0.5892
Epoch: 35/50 --- Loss: 0.6278
Epoch: 36/50 --- Loss: 0.6496
Epoch: 37/50 --- Loss: 0.7316
Epoch: 38/50 --- Loss: 0.6359
Epoch: 39/50 --- Loss: 0.7289
Epoch: 40/50 --- Loss: 0.5386
Epoch: 41/50 --- Loss: 0.6284
Epoch: 42/50 --- Loss: 0.5025
Epoch: 43/50 --- Loss: 0.6688
```

```
Epoch: 44/50 --- Loss: 0.6125
Epoch: 45/50 --- Loss: 0.5285
Epoch: 46/50 --- Loss: 0.5508
Epoch: 47/50 --- Loss: 0.5792
Epoch: 48/50 --- Loss: 0.5902
Epoch: 49/50 --- Loss: 0.5889
Epoch: 50/50 --- Loss: 0.6043
```

In [63]:
```
# Get predictions.

prediction_list = []
for i, batch in enumerate(test_loader_fnn):
    batch[0] = batch[0].to(torch.float32)
    output = model_4a(batch[0])
    _, predicted = torch.max(output.data, 1)
    prediction_list = prediction_list + list(predicted.numpy())
```

In [64]:
```
print("Accuracy for FNN using average Word2Vec vectors:", accuracy(prediction_list, y_test_fnn.values))
```

```
Accuracy for FNN using average Word2Vec vectors: 0.6564166666666666
```

(b) To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature (x = [WT 1 , ...,WT 10]) and train the neural network. Report the accuracy value on the testing split for your MLP model. What do you conclude by comparing accuracy values you obtain with those obtained in the “’Simple Models” section.

In [65]:
```
Append first 10 Word2Vec vectors.
Same procedure as question 3, however, we are not going to divide by the total number of words.

v_features_4b = []
r review in reviews:
  count = 0
  words = review.split(" ")
  for word in words:
      if count == 0:
          try:
              array = [wv[word]]
              count = count + 1
          except:
              continue
      else:
          try:
              array = np.concatenate((array, [wv[word]]), axis = 0)
              count = count + 1
              if count == 10:
                  w2v_features_4b.append(array.flatten("F"))
                  break
          except:
              continue

  # If some words in the review are not in the Word2Vec corpus, or if a review length is shorter than 10, we p

  if count == 0:
      array = [np.zeros(300)]
      for i in range(count, 9):
          array = np.concatenate((array, [np.zeros(300)]), axis = 0)
      w2v_features_4b.append(array.flatten("F"))
  else:
      if count < 10:
          for i in range(count, 10):
              array = np.concatenate((array, [np.zeros(300)]), axis = 0)
          w2v_features_4b.append(array.flatten("F"))
```

In [66]:
```
# Prepare datasets.

X_fnn = w2v_features_4b
y = balanced_dataset["class"] - 1
X_train_fnn, X_test_fnn, y_train_fnn, y_test_fnn = train_test_split(X_fnn, y, test_size = 0.2)
train_data_fnn = pd.DataFrame(data = {"feature": X_train_fnn, "label": y_train_fnn}).reset_index(drop = True)
test_data_fnn = pd.DataFrame(data = {"feature": X_test_fnn, "label": y_test_fnn}).reset_index(drop = True)
train_data_fnn = ReviewDataset(train_data_fnn)
test_data_fnn = ReviewDataset(test_data_fnn)
```

In [67]:
```
# Create data loaders.

batch_size = 20

train_loader_fnn = torch.utils.data.DataLoader(train_data_fnn, batch_size = batch_size)
test_loader_fnn = torch.utils.data.DataLoader(test_data_fnn, batch_size = batch_size)
```

In [68]:
```
# Create FNN model for 4b.

model_4b = FNN(3000,100,10)
print(model_4b)
```

```
FNN(
  (fc1): Linear(in_features=3000, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [69]:
```
# Initialize loss function and optimizer.

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_4b.parameters(), lr=0.01)
```

In [70]:
```
# Model training.

n_epochs = 50
for epoch in range(n_epochs):
    for data, target in train_loader_fnn:
        optimizer.zero_grad()
        data = data.to(torch.float32)
        output = model_4b(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    print('Epoch: {}/{} --- Loss: {:.4f}'.format(epoch+1, n_epochs, loss.item()))
```

```
Epoch: 1/50 --- Loss: 1.0317
Epoch: 2/50 --- Loss: 1.0071
Epoch: 3/50 --- Loss: 0.9900
Epoch: 4/50 --- Loss: 0.7967
Epoch: 5/50 --- Loss: 0.8089
Epoch: 6/50 --- Loss: 0.8250
Epoch: 7/50 --- Loss: 0.6987
Epoch: 8/50 --- Loss: 0.7625
Epoch: 9/50 --- Loss: 0.6885
Epoch: 10/50 --- Loss: 0.6945
Epoch: 11/50 --- Loss: 0.6529
Epoch: 12/50 --- Loss: 0.6522
Epoch: 13/50 --- Loss: 0.6158
Epoch: 14/50 --- Loss: 0.4772
Epoch: 15/50 --- Loss: 0.5560
Epoch: 16/50 --- Loss: 0.5966
Epoch: 17/50 --- Loss: 0.4544
Epoch: 18/50 --- Loss: 0.4222
Epoch: 19/50 --- Loss: 0.4072
```

In [71]:
```
# Get predictions.

prediction_list = []
for i, batch in enumerate(test_loader_fnn):
    batch[0] = batch[0].to(torch.float32)
    output = model_4b(batch[0])
    _, predicted = torch.max(output.data, 1)
    prediction_list = prediction_list + list(predicted.numpy())
```

In [72]:
```
print("Accuracy for FNN using 10 first Word2Vec vectors:", accuracy(prediction_list, y_test_fnn.values))
```

```
Accuracy for FNN using 10 first Word2Vec vectors: 0.5546666666666666
```

Conclusion

MLP using the average Word2Vec vectors performed quite similarly to the simple models in part 3 (~ 0.65 accuracy). However, MLP using the concateneated Word2Vec vectors underperformed in this situation (~ 0.55 accuracy).

5. Recurrent Neural Networks

Using the Word2Vec features, train a recurrent neural network (RNN) for classification. You can refer to the following tutorial to familiarize yourself: [https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html (https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)](https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

(a) Train a simple RNN for sentiment analysis. You can consider an RNN cell with the hidden state size of 20. To feed your data into our RNN, limit the maximum review length to 20 by truncating longer reviews and padding shorter reviews with a null value (0). Report accuracy values on the testing split for your RNN model. What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models.

In [73]:
```
# Create Word2Vec features with lenght 20 reviews.
# Samme procedure as question 4, except instead of length 10 we are considering length 20.

w2v_features_5 = []
for review in reviews:
    count = 0
    words = review.split(" ")
    for word in words:
        if count == 0:
            try:
                array = [wv[word]]
                count = count + 1
            except:
                continue
        else:
            try:
                array = np.concatenate((array, [wv[word]]), axis = 0)
                count = count + 1
                if count == 20:
                    w2v_features_5.append(array)
                    break
            except:
                continue

    # If some words in the review are not in the Word2Vec corpus, or if a review length is shorter than 20, w

    if count == 0:
        array = [np.zeros(300)]
        for i in range(1, 20):
            array = np.concatenate((array, [np.zeros(300)]), axis = 0)
        w2v_features_5.append(array)
    else:
        if count < 20:
            for i in range(count, 20):
                array = np.concatenate((array, [np.zeros(300)]), axis = 0)
            w2v_features_5.append(array)
```

In [74]:
```
# RNN architecture.

class RNN(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers):
        super(RNN, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.rnn = nn.RNN(input_size, hidden_dim, n_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, x):
        batch_size = x.size(0)
        hidden = self.init_hidden(batch_size)
        out, hidden = self.rnn(x, hidden)
        out = self.fc(out[:,-1,:])

        return out, hidden

    def init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim)
        return hidden
```

In [75]:
```
# Prepare datasets.

X_5 = w2v_features_5
y = balanced_dataset["class"] - 1
X_train_5, X_test_5, y_train_5, y_test_5 = train_test_split(X_5, y, test_size = 0.2)
train_data_5_df = pd.DataFrame(data = {"feature": X_train_5, "label": y_train_5}).reset_index(drop = True)
test_data_5_df = pd.DataFrame(data = {"feature": X_test_5, "label": y_test_5}).reset_index(drop = True)
train_data_5 = ReviewDataset(train_data_5_df)
test_data_5 = ReviewDataset(test_data_5_df)
```

In [76]:
```
# Create data loaders.

batch_size = 24

train_loader_5 = torch.utils.data.DataLoader(train_data_5, batch_size = batch_size)
test_loader_5 = torch.utils.data.DataLoader(test_data_5, batch_size = batch_size)
```

In [77]:
```
# Create RNN model for 5a.

model_5a = RNN(300, 3, 20, 1)
```

In [78]:
```
# Initialize loss function and optimizer.

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_5a.parameters(), lr = 0.001)
```

In [79]:
```
# Model training.

n_epochs = 50
for epoch in range(n_epochs):
    for data, target in train_loader_5:
        optimizer.zero_grad()
        data = data.to(torch.float32)
        output, hidden = model_5a(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    print('Epoch: {}/{} --- Loss: {:.4f}'.format(epoch+1, n_epochs, loss.item()))
```

```
Epoch: 31/50 --- Loss: 0.9872
Epoch: 32/50 --- Loss: 0.6990
Epoch: 33/50 --- Loss: 0.7559
Epoch: 34/50 --- Loss: 0.5985
Epoch: 35/50 --- Loss: 0.6191
Epoch: 36/50 --- Loss: 0.5630
Epoch: 37/50 --- Loss: 0.6748
Epoch: 38/50 --- Loss: 0.6070
Epoch: 39/50 --- Loss: 0.7756
Epoch: 40/50 --- Loss: 0.7846
Epoch: 41/50 --- Loss: 0.6263
Epoch: 42/50 --- Loss: 0.9077
Epoch: 43/50 --- Loss: 0.7504
Epoch: 44/50 --- Loss: 0.6752
Epoch: 45/50 --- Loss: 0.9100
Epoch: 46/50 --- Loss: 0.8026
Epoch: 47/50 --- Loss: 0.6536
Epoch: 48/50 --- Loss: 0.7944
Epoch: 49/50 --- Loss: 0.8209
Epoch: 50/50 --- Loss: 0.7573
```

In [80]:
```
# Get predictions.

prediction_list = []
for i, batch in enumerate(test_loader_5):
    batch[0] = batch[0].to(torch.float32)
    output, hidden = model_5a(batch[0])
    _, predicted = torch.max(output.data, 1)
    prediction_list = prediction_list + list(predicted.numpy())
```

In [81]:
```
print("Accuracy for RNN using 20 first Word2Vec vectors:", accuracy(prediction_list, y_test_5.values))
```

```
Accuracy for RNN using 20 first Word2Vec vectors: 0.61175
```

(b) Repeat part (a) by considering a gated recurrent unit cell.

In [82]:
```
# GRU architecture.

class GRU(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers):
        super(GRU, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.gru = nn.GRU(input_size, hidden_dim, n_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, x):
        batch_size = x.size(0)
        hidden = self.init_hidden(batch_size)
        out, hidden = self.gru(x, hidden)
        out = self.fc(out[:,-1,:])

        return out, hidden

    def init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim)
        return hidden
```

In [83]:
```
# Create GRU model for 5b.
model_5b = GRU(300, 3, 20, 1)
```

In [84]:
```
# Initialize loss function and optimizer.

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_5b.parameters(), lr = 0.001)
```

In [85]:
```
# Model training.

n_epochs = 50
for epoch in range(n_epochs):

    for data, target in train_loader_5:
        optimizer.zero_grad()
        data = data.to(torch.float32)
        output, hidden = model_5b(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    print('Epoch: {}/{} --- Loss: {:.4f}'.format(epoch+1, n_epochs, loss.item()))
```

```
Epoch: 1/50 --- Loss: 0.6357
Epoch: 2/50 --- Loss: 0.5890
Epoch: 3/50 --- Loss: 0.5727
Epoch: 4/50 --- Loss: 0.5622
Epoch: 5/50 --- Loss: 0.5531
Epoch: 6/50 --- Loss: 0.5423
Epoch: 7/50 --- Loss: 0.5303
Epoch: 8/50 --- Loss: 0.5204
Epoch: 9/50 --- Loss: 0.5144
Epoch: 10/50 --- Loss: 0.5118
Epoch: 11/50 --- Loss: 0.5123
Epoch: 12/50 --- Loss: 0.5147
Epoch: 13/50 --- Loss: 0.5187
Epoch: 14/50 --- Loss: 0.5244
Epoch: 15/50 --- Loss: 0.5268
Epoch: 16/50 --- Loss: 0.5341
Epoch: 17/50 --- Loss: 0.5376
Epoch: 18/50 --- Loss: 0.5426
Epoch: 19/50 --- Loss: 0.5202
Epoch: 20/50 --- Loss: 0.5407
Epoch: 21/50 --- Loss: 0.5474
Epoch: 22/50 --- Loss: 0.5436
Epoch: 23/50 --- Loss: 0.5435
Epoch: 24/50 --- Loss: 0.5650
Epoch: 25/50 --- Loss: 0.5520
Epoch: 26/50 --- Loss: 0.6068
Epoch: 27/50 --- Loss: 0.5752
Epoch: 28/50 --- Loss: 0.5879
Epoch: 29/50 --- Loss: 0.5866
Epoch: 30/50 --- Loss: 0.6010
Epoch: 31/50 --- Loss: 0.5879
Epoch: 32/50 --- Loss: 0.6194
Epoch: 33/50 --- Loss: 0.6069
Epoch: 34/50 --- Loss: 0.6163
Epoch: 35/50 --- Loss: 0.6032
Epoch: 36/50 --- Loss: 0.6097
Epoch: 37/50 --- Loss: 0.5870
Epoch: 38/50 --- Loss: 0.6098
Epoch: 39/50 --- Loss: 0.6132
Epoch: 40/50 --- Loss: 0.6496
Epoch: 41/50 --- Loss: 0.6096
Epoch: 42/50 --- Loss: 0.6333
Epoch: 43/50 --- Loss: 0.6438
```

```
Epoch: 44/50 --- Loss: 0.6503
Epoch: 45/50 --- Loss: 0.6100
Epoch: 46/50 --- Loss: 0.6252
Epoch: 47/50 --- Loss: 0.6058
Epoch: 48/50 --- Loss: 0.6506
Epoch: 49/50 --- Loss: 0.6443
Epoch: 50/50 --- Loss: 0.6501
```

In [88]:
```
# Get predictions.

prediction_list = []
for i, batch in enumerate(test_loader_5):
    batch[0] = batch[0].to(torch.float32)
    output, hidden = model_5b(batch[0])
    _, predicted = torch.max(output.data, 1)
    prediction_list = prediction_list + list(predicted.numpy())
```

In [89]:
```
print("Accuracy for GRU using 20 first Word2Vec vectors:", accuracy(prediction_list, y_test_5.values))
```

```
Accuracy for GRU using 20 first Word2Vec vectors: 0.6403333333333333
```

(c) Repeat part (a) by considering an LSTM unit cell. What do you conclude by comparing accuracy values you obtain by GRU, LSTM, and simple RNN.

In [90]:
```
# LSTM architecture.

class LSTM(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers):
        super(LSTM, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.lstm = nn.LSTM(input_size, hidden_dim, n_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, x):
        batch_size = x.size(0)
        hidden = self.init_hidden(batch_size)
        cell = self.init_cell(batch_size)
        out, hidden = self.lstm(x, (hidden, cell))
        out = self.fc(out[:,-1,:])

        return out, hidden

    def init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim)

        return hidden

    def init_cell(self, batch_size):
        cell = torch.zeros(self.n_layers, batch_size, self.hidden_dim)

        return cell
```

In [91]:
```
model_5c = LSTM(300, 3, 20, 1)
```

In [92]:
```
# Initialize loss function and optimizer.

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_5c.parameters(), lr = 0.001)
```

In [93]:
```
# Model training.

n_epochs = 50
for epoch in range(n_epochs):
    for data, target in train_loader_5:
        optimizer.zero_grad()
        data = data.to(torch.float32)
        output, hidden = model_5c(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    print('Epoch: {}/{} --- Loss: {:.4f}'.format(epoch+1, n_epochs, loss.item()))
```

```
Epoch: 1/50 --- Loss: 0.7002
Epoch: 2/50 --- Loss: 0.5616
Epoch: 3/50 --- Loss: 0.5159
Epoch: 4/50 --- Loss: 0.4927
Epoch: 5/50 --- Loss: 0.4783
Epoch: 6/50 --- Loss: 0.4713
Epoch: 7/50 --- Loss: 0.4669
Epoch: 8/50 --- Loss: 0.4601
Epoch: 9/50 --- Loss: 0.4634
Epoch: 10/50 --- Loss: 0.4657
Epoch: 11/50 --- Loss: 0.4620
Epoch: 12/50 --- Loss: 0.4631
Epoch: 13/50 --- Loss: 0.4890
Epoch: 14/50 --- Loss: 0.5002
Epoch: 15/50 --- Loss: 0.4838
Epoch: 16/50 --- Loss: 0.5322
Epoch: 17/50 --- Loss: 0.5346
Epoch: 18/50 --- Loss: 0.5774
Epoch: 19/50 --- Loss: 0.5292
Epoch: 20/50 --- Loss: 0.5185
Epoch: 21/50 --- Loss: 0.5793
Epoch: 22/50 --- Loss: 0.5048
Epoch: 23/50 --- Loss: 0.6153
Epoch: 24/50 --- Loss: 0.5379
Epoch: 25/50 --- Loss: 0.5040
Epoch: 26/50 --- Loss: 0.5810
Epoch: 27/50 --- Loss: 0.5286
Epoch: 28/50 --- Loss: 0.5338
Epoch: 29/50 --- Loss: 0.5289
Epoch: 30/50 --- Loss: 0.5964
Epoch: 31/50 --- Loss: 0.4849
Epoch: 32/50 --- Loss: 0.5416
Epoch: 33/50 --- Loss: 0.4779
Epoch: 34/50 --- Loss: 0.4941
Epoch: 35/50 --- Loss: 0.5734
Epoch: 36/50 --- Loss: 0.5993
Epoch: 37/50 --- Loss: 0.5761
Epoch: 38/50 --- Loss: 0.5774
Epoch: 39/50 --- Loss: 0.5553
Epoch: 40/50 --- Loss: 0.5568
Epoch: 41/50 --- Loss: 0.6009
Epoch: 42/50 --- Loss: 0.5959
Epoch: 43/50 --- Loss: 0.6301
```

```
Epoch: 44/50 --- Loss: 0.6177
Epoch: 45/50 --- Loss: 0.7080
Epoch: 46/50 --- Loss: 0.5930
Epoch: 47/50 --- Loss: 0.6443
Epoch: 48/50 --- Loss: 0.5760
Epoch: 49/50 --- Loss: 0.5711
Epoch: 50/50 --- Loss: 0.6213
```

In [94]:
```
# Get predictions.

prediction_list = []
for i, batch in enumerate(test_loader_5):
    batch[0] = batch[0].to(torch.float32)
    output, hidden = model_5c(batch[0])
    _, predicted = torch.max(output.data, 1)
    prediction_list = prediction_list + list(predicted.numpy())
```

In [95]:
```
print("Accuracy for LSTM using 20 first Word2Vec vectors:", accuracy(prediction_list, y_test_5.values))
```

```
Accuracy for LSTM using 20 first Word2Vec vectors: 0.6334166666666666
```

Conclusion

We can see that GRU and LSTM performed better than RNN when it comes to this classification problem. The hidden states and cell states helped improved the accuracy.

Reference

Feedforward Neural Networks: https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist (https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist)

Recurrent Neural Networks:https://blog.floydhub.com/a-beginners-guide-on-recurrent-neural-networks-with-pytorch/ (https://blog.floydhub.com/a-beginners-guide-on-recurrent-neural-networks-with-pytorch/)