hw2

September 29, 2022

0.1 Imports and Function Declarations

[1]:
```
!pip install nltk -q
!pip install gensim -q
!pip install textacy -q
!pip install contractions -q
!pip install -U symspellpy -q
!pip install torch -q
```

[2]:
```
import warnings
warnings.filterwarnings('ignore')

import pandas as pd
import numpy as np
from bs4 import BeautifulSoup

import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

nltk.download('stopwords', quiet=True)
nltk.download('punkt', quiet=True)

from textacy.preprocessing import remove, normalize, replace

import contractions

import gensim.downloader as api
from gensim.models import Word2Vec

import pkg_resources
from symspellpy import SymSpell, Verbosity

from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
```

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
from torch.utils.data.sampler import SubsetRandomSampler
import torch.nn.functional as F
```

0.1.1 Definition of Global variables

[3]:
```
# GLOBALS

F_PATH = 'amazon_reviews_us_Jewelry_v1_00.tsv'

STAR_H = 'star_rating'
REVIEW_H = 'review_body'

COLS=[STAR_H, REVIEW_H]

VAL_STARS = {1, 2, 3, 4, 5}

WV = api.load('word2vec-google-news-300')

SPELLER = SymSpell(max_dictionary_edit_distance=2, prefix_length=7)
dictionary_path = pkg_resources.resource_filename("symspellpy",␣
↪"frequency_dictionary_en_82_765.txt")
bigram_path = pkg_resources.resource_filename("symspellpy",␣
↪"frequency_bigramdictionary_en_243_342.txt")

SPELLER.load_dictionary(dictionary_path, term_index=0, count_index=1)
SPELLER.load_bigram_dictionary(bigram_path, term_index=0, count_index=2)
```

[3]: True

0.1.2 Dataset generation and pre-processing functions

[4]:
```
def read_data(f_path=F_PATH):
    df = pd.read_csv(f_path, sep='\t', usecols=COLS, low_memory=False)
    df.dropna(inplace=True)
    return df.astype({STAR_H:'int32', REVIEW_H: pd.StringDtype()})

def get_sample(df, s_size=20000):
    grouped = df.groupby(STAR_H)
    rat_dfs = [grouped.get_group(rating).sample(n=s_size) for rating in␣
↪VAL_STARS]
    return pd.concat(rat_dfs)

def gen_clean(text):
```

```
    """
    gen text cleanup
    incl removal: extended ws, html tags, urls
    """
    text = BeautifulSoup(text, "html.parser").text #rm html tags
    text = replace.urls(text, '')
    text = contractions.fix(text)
    text = remove.punctuation(text)
    text = replace.numbers(text, '')
    text = normalize.whitespace(text).lower()
    text = replace.emojis(text, '')
    toks = rm_stops(text)

    return toks

def rm_stops(text):
    """
    remove stop words from text
    """
    stops = set(stopwords.words("english"))
    sans_stops = [tok for tok in word_tokenize(text) if tok not in stops]
    return sans_stops
```

0.1.3 Format Word2Vec vectors for networks functions

[5]:
```
def concat_ten(list_of_vecs) :
    # returns concatenation of first ten vectors in a list

    l = len(list_of_vecs)
    if l>=2:
        m = min(10, l)
        c = np.concatenate(list_of_vecs[:m])
    else:
        c = np.array([])
        if l==1: c = np.concatenate([c, list_of_vecs[0]])

    while(l<10):
        l+=1
        c = np.concatenate([c, np.zeros(300,)])

    assert c.shape==(3000,)
    return c

def get_vecs(tok_list):
    # fed a list of tokens and attempts to retreive word2vec vectors for each
 ↪word
```

```
    # if a word is not found, attempts to correct misspelled words before␣
↪attempting
    # word2vec vector retreival again

    w2v = []
    skipped = 0
    new_toks = None

    for w in tok_list:
        try:
            w2v.append(WV[w])
        except KeyError:
            skipped = 1
            break

    if skipped:
        w2v = []
        new_toks = spell_check(" ".join(tok_list))
        skipped = 0
        for w in new_toks:
            if w not in set(stopwords.words("english")):
                try: w2v.append(WV[w])
                except KeyError: continue

    return w2v, new_toks

def format_vecs(df, only_20=False):
    # adds either a column for the first 20 vectors from a review
    # or two columns for the average vector and concatenated first ten
    # returns df with above changes

    avg_vecs = []
    first_ten = []
    first_20 = []

    for _, row in df.iterrows():
        w2v, new_toks = get_vecs(row['cl_toks'])

        if new_toks is not None:
            row['cl_toks'] = new_toks

        if only_20: first_20.append(get_twenty(w2v))

        else:
            w2v_arr = np.array(w2v) if w2v else np.zeros((1,300))

            avg_vecs.append(np.mean(w2v_arr, axis=0))
```

```python
            first_ten.append(concat_ten(w2v))

    if only_20:

        df['first_20'] = first_20
        return df

    df['avg_vecs'] = avg_vecs
    df['first_ten_vecs'] = first_ten

    return df

def get_twenty(list_of_vecs):
    # returns a np.array of the first 20 vectors in a review
    # padded if needed

    m = min(20, len(list_of_vecs))

    if m>=2:
        c = np.array(list_of_vecs[:m])
    else:
        if m==1:
            c = np.array(list_of_vecs)
        else:
            m+=1
            c = np.zeros((1,300))

    while(m<20):
        m+=1
        c = np.append(c, np.zeros((1,300)), axis=0)

    assert c.shape==(20, 300)
    return c

def spell_check(text):
    # attempts to spell check

    sugs = SPELLER.lookup_compound(text, max_edit_distance=2)
    term = [sug.term for sug in sugs]
    n_str = "  ".join(term).split()
    return [t for t in n_str if t in SPELLER.words.keys()]
```

0.1.4 PyTorch Classes for Dataset generation and Networks

- I referred to the kaggle tutorial in the assignment spec for the implementation of the `FNNDataset` and `FNN` classes (https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook)

- For the RNN implementation I followed the PyTorch tutorial mentioned in the spec (https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)
- For GRU, I implemented code from the following article: https://www.educba.com/pytorch-gru/?source=leftnav

[6]:
```
# PyTorch Classes

class FNNDataset(Dataset):

    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

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

    def __getitem__(self, index):

        review = torch.from_numpy(self.data[index])
        label = self.labels[index]-1

        return review.to(torch.float32), label

class FNN(nn.Module):
    def __init__(self, dims=300):
        super(FNN, self).__init__()

        hidden_1 = 50
        hidden_2 = 10

        self.fc1 = nn.Linear(dims, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 5)

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

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
```

```
        self.softmax = nn.LogSoftmax()

    def forward(self, samp, hidden):
        combined = torch.cat((samp, hidden))

        hidden = self.i2h(combined)

        output = self.i2o(combined)

        output = self.softmax(output)

        return output, hidden

    def init_hidden(self):
        return torch.zeros(self.hidden_size,)

class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GRU, self).__init__()

        self.hidden_size = hidden_size

        self.gru = nn.GRU(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()

    def forward(self, samp, hidden):
        output, hidden = self.gru(samp, hidden)
        output = self.fc(self.relu(output[:,-1]))
        return output, hidden

    def init_hidden(self):
        return torch.zeros(1, self.hidden_size)
```

0.1.5 PyTorch Training Functions

- I referred to the kaggle tutorial referenced in the assignment spec for the implementation of the training function definitions below. (https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook)

[7]:
```
def run_fnn(train_loader, valid_loader, model_out, dims=300, n_epochs=50):
    valid_loss_min = np.Inf

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

```python
    for epoch in range(n_epochs):
        train_loss = 0.0
        valid_loss = 0.0

        model.train()
        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()*data.size(0)

        model.eval()
        for data, target in valid_loader:
            output = model(data)
            loss = criterion(output, target)
            valid_loss += loss.item()*data.size(0)

        train_loss = train_loss/len(train_loader.dataset)
        valid_loss = valid_loss/len(valid_loader.dataset)

        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch+1,
            train_loss,
            valid_loss
            ))

        if valid_loss <= valid_loss_min:
            torch.save(model.state_dict(), model_out)
            valid_loss_min = valid_loss
    return model

def run_rnn(train_loader, valid_loader, model_out, batch_size=1000, n_epochs=4, n_hidden=20, lr=0.005):

    valid_loss_min = np.Inf

    rnn = RNN(300, n_hidden, 5)
    optimizer = torch.optim.SGD(rnn.parameters(), lr=lr, weight_decay=1e-5)
    criterion = nn.NLLLoss()

    for epoch in range(n_epochs):
        train_loss = 0.0
        valid_loss = 0.0
```

```
        rnn.train()
        for data, targets in train_loader:
            for i in range(data.size()[0]):

                review_tensor = data[i]
                rating = targets[i]
                hidden = rnn.init_hidden()
                optimizer.zero_grad()

                for j in range(review_tensor.size()[0]):
                    output, hidden = rnn(review_tensor[j], hidden)

                loss = criterion(output, rating)

                loss.backward()
                optimizer.step()

                train_loss += loss.item()*data.size(0)

        rnn.eval()
        for data, target in valid_loader:
            for i in range(data.size()[0]):

                review_tensor = data[i]
                rating = targets[i]
                hidden = rnn.init_hidden()

                for j in range(review_tensor.size()[0]):
                    output, hidden = rnn(review_tensor[j], hidden)

                loss = criterion(output, rating)

                valid_loss += loss.item()*data.size(0)

        train_loss = train_loss/len(train_loader.dataset)
        valid_loss = valid_loss/len(valid_loader.dataset)

        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.
 ↪format(
            epoch+1,
            train_loss,
            valid_loss
            ))

        if valid_loss <= valid_loss_min:
            torch.save(rnn.state_dict(), model_out)
            valid_loss_min = valid_loss
```

```python
    return rnn

def run_gru(train_loader, valid_loader, model_out, batch_size=1000, n_epochs=4,␣
 ↪n_hidden=20, lr=0.005):

    valid_loss_min = np.Inf

    gru = GRU(300, n_hidden, 5)
    optimizer = torch.optim.SGD(gru.parameters(), lr=lr, weight_decay=1e-5)
    criterion = nn.NLLLoss()

    for epoch in range(n_epochs):
        train_loss = 0.0
        valid_loss = 0.0

        gru.train()
        for data, targets in train_loader:
            for i in range(data.size()[0]):

                review_tensor = data[i]
                rating = targets[i]
                hidden = gru.init_hidden()
                optimizer.zero_grad()

                output, hidden = gru(review_tensor, hidden)

                loss = criterion(output, rating)

                loss.backward()
                optimizer.step()

                train_loss += loss.item()*data.size(0)

        gru.eval()
        for data, target in valid_loader:
            for i in range(data.size()[0]):

                review_tensor = data[i]
                rating = targets[i]
                hidden = gru.init_hidden()

                output, hidden = gru(review_tensor, hidden)

                loss = criterion(output, rating)

                valid_loss += loss.item()*data.size(0)
```

```
        train_loss = train_loss/len(train_loader.dataset)
        valid_loss = valid_loss/len(valid_loader.dataset)

        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.
↪format(
            epoch+1,
            train_loss,
            valid_loss
            ))

        if valid_loss <= valid_loss_min:
            torch.save(gru.state_dict(), model_out)
            valid_loss_min = valid_loss

    return gru
```

0.1.6 Dataset loaders and Accuracy functions

- I referred to the kaggle tutorial provided in the assignment spec for the dataset loader function (https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook)

[8]:
```
def get_loaders(features, categories, batch_size=100, valid_size=0.2):

    X_train, X_test, train_labels, test_labels = train_test_split(features.
↪tolist(), categories.tolist(), test_size=0.2, random_state=42)

    train_data = FNNDataset(X_train, train_labels)
    test_data = FNNDataset(X_test, test_labels)

    num_workers = 0

    num_train = len(train_data)
    indices = list(range(num_train))
    np.random.shuffle(indices)
    split = int(np.floor(valid_size * num_train))
    train_idx, valid_idx = indices[split:], indices[:split]

    train_sampler = SubsetRandomSampler(train_idx)
    valid_sampler = SubsetRandomSampler(valid_idx)

    train_loader = DataLoader(train_data, batch_size=batch_size,␣
↪sampler=train_sampler, num_workers=num_workers)
    valid_loader = DataLoader(train_data, batch_size=batch_size,␣
↪sampler=valid_sampler, num_workers=num_workers)
```

```
    test_loader = DataLoader(test_data, batch_size=batch_size,␣
↪num_workers=num_workers)

    return train_loader, valid_loader, test_loader

def get_fnn_acc(model, test_data):
    corr = 0
    total = 0
    with torch.no_grad():
        for data, targets in test_loader:
            outs = model(data)
            _, preds= torch.max(outs, 1)
            corr += len([targ for targ, pred in zip(targets, preds) if␣
↪targ==pred])
            total += len(targets)

    return corr / total

def get_rnn_acc(rnn, test_data):
    corr = 0
    total = 0
    with torch.no_grad():
        for data, targets in test_loader:
            for i in range(data.size()[0]):
                review_tensor = data[i]
                rating = targets[i]
                hidden = rnn.init_hidden()

                for j in range(review_tensor.size()[0]):
                    output, hidden = rnn(review_tensor[j], hidden)

                _, pred= torch.max(output, 0)
                corr += 1 if rating.int()==pred.int() else 0
                total += 1

    return corr / total

def get_gru_acc(gru, test_data):
    corr = 0
    total = 0
    with torch.no_grad():
        for data, targets in test_loader:
            for i in range(data.size()[0]):
                review_tensor = data[i]
                rating = targets[i]
                hidden = gru.init_hidden()
```

```
                output, hidden = gru(review_tensor, hidden)

                _, pred= torch.max(output, 0)
                corr += 1 if rating.int()==pred.int() else 0
                total += 1

    return corr / total
```

0.2 Question 1: Dataset Generation

[9]:
```
df = read_data()
sampled = get_sample(df)
sampled['cl_toks'] = sampled[REVIEW_H].apply(gen_clean)
sampled.drop(columns=[REVIEW_H], inplace=True)

# sampled.to_pickle('sample_dfs/samp_raw.pkl')
# sampled = pd.read_pickle('sample_dfs/samp_raw.pkl')

## OTHER PICKLES:
# - 'sample_dfs/samp_toks.pkl': just clean tokens and star rating
# - 'sample_dfs/samp_avg_cats.pkl': avg vectors, concatenated first 10 vecs
# - 'sample_dfs/samp_20.pkl': first 20 vectors of each review
```

0.3 Question 2: Word Embedding

0.3.1 Part A:

[10]:
```
wv_bracelet = WV.most_similar(negative=["wrist"], positive=['bracelet',␣
    ↪'neck'], topn=1)
wv_girl = WV.most_similar( positive=['girl', 'age'], topn=1)
wv_family = WV.most_similar(negative=['child'], positive=['family'], topn=1)

print(f"Bracelet - Wrist + Neck = {wv_bracelet}")
print(f"Girl + age = {wv_girl}")
print(f"Family - Child = {wv_family}")
```

```
Bracelet - Wrist + Neck = [('necklace', 0.5466936826705933)]
Girl + age = [('boy', 0.7243723273277283)]
Family - Child = [('friends', 0.3765709400177002)]
```

0.3.2 Part B

Q: What do you conclude from comparing vectors generated by yourself and the pretrained model?

A: The google model seems to have a higher degree of accuracy when identifying encoded similarity. This behavior is expected since the google model was trained on data with greater variance. That is, the google model was provided more context on similarities between words and had the ability to tune the vectors to a higher degree of accuracy.

Q: Which of the Word2Vec models seems to encode semantic similarities between words better?

A: The imported google model seems to encode semantic similarities better.

[11]:
```
model = Word2Vec(sentences=sampled['cl_toks'], vector_size=300, window=11,␣
↪min_count=10, workers=4)
```

[12]:
```
m_bracelet = model.wv.most_similar(negative=["wrist"], positive=['bracelet',␣
↪'neck'], topn=1)
m_girl = model.wv.most_similar(positive=['girl', 'age'], topn=1)
m_family = model.wv.most_similar(negative=['child'], positive=['family'],␣
↪topn=1)

print(f"Bracelet - Wrist + Neck = {m_bracelet}")
print(f"Girl + Age = {m_girl}")
print(f"Family - Child = {m_family}")
```

```
Bracelet - Wrist + Neck = [('necklace', 0.7398298382759094)]
Girl + Age = [('teen', 0.9063260555267334)]
Family - Child = [('february', 0.6471871137619019)]
```

0.4 Question 3: Simple Models

Note: For full comparison between the features' performance, I opted for the same hyperparameters as the first assignment TF-IDF Accuracies from CA #1: **Perceptron:** 0.39 **SVM:** 0.50

Q: What do you conclude from comparing performances for the models trained using the two different feature types (TF-IDF and your trained Word2Vec features)?

A: The TF-IDF had higher accuracy ratings for both simple models. Intuitively, this makes sense because of TF-IDF's ability to represent individual term importance. This is likely advantageous when distinguishing the difference between generally positive and generally negative reviews. For example, generally positive reviews will include terms with positive connotations like 'good', 'nice', 'beatutiful', etc. While negative reviews will include terms that have negative connotations like 'bad', 'broken', 'poor', etc.

The version of Word2Vec utilized in the models below could only account for the average of vectors per review. Thus, diluting sematic simlarities between reviews and making the classification more difficult.

[13]:
```
s = format_vecs(sampled)

X_train, X_test, train_labels, test_labels = train_test_split(s.avg_vecs.
↪tolist(), s[STAR_H].tolist(), test_size=0.2, random_state=42)
```

[14]:
```
p = Perceptron(random_state=42, class_weight='balanced', max_iter=20,␣
↪n_iter_no_change=3)
p.fit(X_train, train_labels)
p_pred = p.predict(X_test)
```

```
print(accuracy_score(test_labels, p_pred))
```

```
0.3567
```

[15]:
```
svm = LinearSVC(penalty='l1', dual=False, random_state=42, max_iter=300)
svm.fit(X_train, train_labels)
s_pred = svm.predict(X_test)

print(accuracy_score(test_labels, s_pred))
```

```
0.46695
```

0.5 Question 4: Feedforward Neural Networks

0.5.1 Part A:

[16]:
```
train_loader, valid_loader, test_loader = get_loaders(s.avg_vecs, s[STAR_H])
run_fnn(train_loader, valid_loader, model_out='models/fnn1.pt')

# load best model saved from training
fnn1 = FNN(dims=300)
fnn1.load_state_dict(torch.load('models/fnn1.pt'))
acc = get_fnn_acc(fnn1, test_loader)

print(f"\n\nAccuracy: {acc}")
```

```
Epoch: 1        Training Loss: 1.289831         Validation Loss: 0.321946
Epoch: 2        Training Loss: 1.287078         Validation Loss: 0.321687
Epoch: 3        Training Loss: 1.286229         Validation Loss: 0.321438
Epoch: 4        Training Loss: 1.284878         Validation Loss: 0.321009
Epoch: 5        Training Loss: 1.282790         Validation Loss: 0.320302
Epoch: 6        Training Loss: 1.279050         Validation Loss: 0.319021
Epoch: 7        Training Loss: 1.271714         Validation Loss: 0.316366
Epoch: 8        Training Loss: 1.256041         Validation Loss: 0.310587
Epoch: 9        Training Loss: 1.222833         Validation Loss: 0.299115
Epoch: 10       Training Loss: 1.169470         Validation Loss: 0.284829
Epoch: 11       Training Loss: 1.120206         Validation Loss: 0.274533
Epoch: 12       Training Loss: 1.088898         Validation Loss: 0.268469
Epoch: 13       Training Loss: 1.069969         Validation Loss: 0.265134
Epoch: 14       Training Loss: 1.057697         Validation Loss: 0.262311
Epoch: 15       Training Loss: 1.049132         Validation Loss: 0.260301
Epoch: 16       Training Loss: 1.042693         Validation Loss: 0.258892
Epoch: 17       Training Loss: 1.037508         Validation Loss: 0.257791
Epoch: 18       Training Loss: 1.033101         Validation Loss: 0.256611
Epoch: 19       Training Loss: 1.028670         Validation Loss: 0.255626
Epoch: 20       Training Loss: 1.024466         Validation Loss: 0.254581
Epoch: 21       Training Loss: 1.020242         Validation Loss: 0.253377
```

```
Epoch: 22	Training Loss: 1.015621		Validation Loss: 0.252358
Epoch: 23	Training Loss: 1.010983		Validation Loss: 0.251327
Epoch: 24	Training Loss: 1.006361		Validation Loss: 0.249915
Epoch: 25	Training Loss: 1.001914		Validation Loss: 0.248782
Epoch: 26	Training Loss: 0.997809		Validation Loss: 0.247905
Epoch: 27	Training Loss: 0.994027		Validation Loss: 0.247101
Epoch: 28	Training Loss: 0.990567		Validation Loss: 0.246089
Epoch: 29	Training Loss: 0.987479		Validation Loss: 0.245600
Epoch: 30	Training Loss: 0.984749		Validation Loss: 0.244827
Epoch: 31	Training Loss: 0.982233		Validation Loss: 0.244364
Epoch: 32	Training Loss: 0.979973		Validation Loss: 0.243902
Epoch: 33	Training Loss: 0.977932		Validation Loss: 0.243460
Epoch: 34	Training Loss: 0.975936		Validation Loss: 0.243046
Epoch: 35	Training Loss: 0.974161		Validation Loss: 0.242657
Epoch: 36	Training Loss: 0.972507		Validation Loss: 0.242480
Epoch: 37	Training Loss: 0.970823		Validation Loss: 0.242117
Epoch: 38	Training Loss: 0.969304		Validation Loss: 0.241784
Epoch: 39	Training Loss: 0.967818		Validation Loss: 0.241441
Epoch: 40	Training Loss: 0.966544		Validation Loss: 0.241260
Epoch: 41	Training Loss: 0.965096		Validation Loss: 0.240967
Epoch: 42	Training Loss: 0.963770		Validation Loss: 0.240654
Epoch: 43	Training Loss: 0.962558		Validation Loss: 0.241007
Epoch: 44	Training Loss: 0.961434		Validation Loss: 0.240327
Epoch: 45	Training Loss: 0.960330		Validation Loss: 0.240094
Epoch: 46	Training Loss: 0.959091		Validation Loss: 0.240323
Epoch: 47	Training Loss: 0.957974		Validation Loss: 0.240632
Epoch: 48	Training Loss: 0.956971		Validation Loss: 0.239614
Epoch: 49	Training Loss: 0.955921		Validation Loss: 0.239613
Epoch: 50	Training Loss: 0.955168		Validation Loss: 0.239321
```

```
Accuracy: 0.47265
```

0.5.2 Part B:

Q: What do you conclude by comparing accuracy values you obtain with those obtained in the "'Simple Models" section?

A: The model in part A was able to increase accuracy with an added perceptron layer. This is consistent with my expectations since a model with multiple layers can utilize backpropogation to update weights and biases, thus, enhancing the learning capabilities of the model. Part B, as expected, performed worse than the model trained in part A and SVM. I suspect the lack of accurate performance is due to the difference of features the model was trained on. That is, both SVM and model A were trained on the average of word vectors across each review. These features are likely more representative of each review as a whole in comparison to model B since it was trained using only the first ten terms in each review.

[17]:
```python
train_loader, valid_loader, test_loader = get_loaders(s.first_ten_vecs,␣
    ↪s[STAR_H])
run_fnn(train_loader, valid_loader, model_out='models/fnn2.pt', dims=3000)

# load best model saved from training
fnn2 = FNN(dims=3000)
fnn2.load_state_dict(torch.load('models/fnn2.pt'))
acc = get_fnn_acc(fnn2, test_loader)

print(f"\n\nAccuracy: {acc}")
```

```
Epoch: 1        Training Loss: 1.293552         Validation Loss: 0.321969
Epoch: 2        Training Loss: 1.286143         Validation Loss: 0.321154
Epoch: 3        Training Loss: 1.280997         Validation Loss: 0.319141
Epoch: 4        Training Loss: 1.266904         Validation Loss: 0.313557
Epoch: 5        Training Loss: 1.228594         Validation Loss: 0.299646
Epoch: 6        Training Loss: 1.163305         Validation Loss: 0.283670
Epoch: 7        Training Loss: 1.108798         Validation Loss: 0.273373
Epoch: 8        Training Loss: 1.073895         Validation Loss: 0.267057
Epoch: 9        Training Loss: 1.051187         Validation Loss: 0.263268
Epoch: 10       Training Loss: 1.035969         Validation Loss: 0.260855
Epoch: 11       Training Loss: 1.024991         Validation Loss: 0.259609
Epoch: 12       Training Loss: 1.016362         Validation Loss: 0.258689
Epoch: 13       Training Loss: 1.009476         Validation Loss: 0.258398
Epoch: 14       Training Loss: 1.003632         Validation Loss: 0.257479
Epoch: 15       Training Loss: 0.998481         Validation Loss: 0.256827
Epoch: 16       Training Loss: 0.993861         Validation Loss: 0.256957
Epoch: 17       Training Loss: 0.989612         Validation Loss: 0.256587
Epoch: 18       Training Loss: 0.985948         Validation Loss: 0.256281
Epoch: 19       Training Loss: 0.982135         Validation Loss: 0.256467
Epoch: 20       Training Loss: 0.978908         Validation Loss: 0.256420
Epoch: 21       Training Loss: 0.975518         Validation Loss: 0.256527
Epoch: 22       Training Loss: 0.972725         Validation Loss: 0.256265
Epoch: 23       Training Loss: 0.969448         Validation Loss: 0.256291
Epoch: 24       Training Loss: 0.966579         Validation Loss: 0.256426
Epoch: 25       Training Loss: 0.963458         Validation Loss: 0.256488
Epoch: 26       Training Loss: 0.960554         Validation Loss: 0.256552
Epoch: 27       Training Loss: 0.957317         Validation Loss: 0.258122
Epoch: 28       Training Loss: 0.954529         Validation Loss: 0.256715
Epoch: 29       Training Loss: 0.950997         Validation Loss: 0.256806
Epoch: 30       Training Loss: 0.947747         Validation Loss: 0.257582
Epoch: 31       Training Loss: 0.944337         Validation Loss: 0.257218
Epoch: 32       Training Loss: 0.940869         Validation Loss: 0.257222
Epoch: 33       Training Loss: 0.937251         Validation Loss: 0.257585
Epoch: 34       Training Loss: 0.933257         Validation Loss: 0.257878
Epoch: 35       Training Loss: 0.929474         Validation Loss: 0.258311
Epoch: 36       Training Loss: 0.925617         Validation Loss: 0.258529
```

```
Epoch: 37       Training Loss: 0.921346        Validation Loss: 0.258751
Epoch: 38       Training Loss: 0.917286        Validation Loss: 0.258863
Epoch: 39       Training Loss: 0.912283        Validation Loss: 0.259329
Epoch: 40       Training Loss: 0.907787        Validation Loss: 0.259878
Epoch: 41       Training Loss: 0.902919        Validation Loss: 0.260063
Epoch: 42       Training Loss: 0.897883        Validation Loss: 0.260654
Epoch: 43       Training Loss: 0.892583        Validation Loss: 0.261700
Epoch: 44       Training Loss: 0.886888        Validation Loss: 0.261547
Epoch: 45       Training Loss: 0.881278        Validation Loss: 0.262745
Epoch: 46       Training Loss: 0.875221        Validation Loss: 0.263229
Epoch: 47       Training Loss: 0.868867        Validation Loss: 0.264021
Epoch: 48       Training Loss: 0.862615        Validation Loss: 0.264368
Epoch: 49       Training Loss: 0.856232        Validation Loss: 0.267167
Epoch: 50       Training Loss: 0.849752        Validation Loss: 0.265781


Accuracy: 0.4384
```

0.6 Question 5:

0.6.1 Part A:

Q: What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models?

A: As we have discussed in class, RNNs perform best with sequential data. Thus, the performance of the RNN, in comparison to the FNN models, is as expected. While FNN's directly associate inputs with outputs, RNNs focus on the prediction task of what comes next. This attribute of the RNN is not particularly useful for the classification of reviews, so, intuitively the FNNs should have a higher accuracy in this case.

[18]:
```python
s = format_vecs(sampled, only_20=True)
```

[19]:
```python
train_loader, valid_loader, test_loader = get_loaders(s.first_20, s[STAR_H],␣
 ↪batch_size=1000)
run_rnn(train_loader, valid_loader, 'models/rnn.pt')

# load best model saved from training
rnn = RNN(300, 20, 5)
rnn.load_state_dict(torch.load('models/rnn.pt'))
acc = get_rnn_acc(rnn, test_loader)

print(f"\n\nAccuracy: {acc}")
```

```
Epoch: 1        Training Loss: 1282.675225     Validation Loss: 323.739831
Epoch: 2        Training Loss: 1256.728412     Validation Loss: 327.203029
Epoch: 3        Training Loss: 1253.001726     Validation Loss: 327.260734
Epoch: 4        Training Loss: 1248.526298     Validation Loss: 325.983668
```

```
Accuracy: 0.2129
```

0.6.2 Part B:

Q: What do you conclude by comparing accuracy values you obtain with those obtained using simple RNN? **A:** As mentioned previously, RNNs are not the ideal network for text classification problems. It follows then, that even with a gated unit cell, the accuracies between the two models would be very similar.

[20]:
```python
run_gru(train_loader, valid_loader, 'models/gru.pt')

# load best model saved from training
gru = GRU(300, 20, 5)
gru.load_state_dict(torch.load('models/gru.pt'))
acc = get_gru_acc(gru, test_loader)

print(f"\n\nAccuracy: {acc}")
```

```
Epoch: 1        Training Loss: -505280.279537   Validation Loss: -260304.574089
Epoch: 2        Training Loss: -1576231.698891  Validation Loss: -527594.950278
Epoch: 3        Training Loss: -2644560.905676  Validation Loss: -793889.721201
Epoch: 4        Training Loss: -3705604.375861  Validation Loss: -1058912.408795


Accuracy: 0.2001
```