In [1]:
```
import pandas as pd
import numpy as np
import gensim.downloader as api
import gensim.models
from gensim import utils
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
from sklearn import svm
from sklearn.feature_extraction.text import TfidfVectorizer
import argparse

import torch
from torch.utils.data import DataLoader, Dataset
import torchvision
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
```

Q1:

Read Data

In [2]:
```
#Reading the review and rating data using pandas read_table function
col = ['review_body', 'star_rating']
data = pd.read_table('amazon_reviews_us_Jewelry_v1_00.tsv', usecols = col)
```

```
/Users/davisyusuf/opt/anaconda3/lib/python3.9/site-packages/IPython/core/interactiveshell.py:3444: Dty
peWarning: Columns (7) have mixed types.Specify dtype option on import or set low_memory=False.
  exec(code_obj, self.user_global_ns, self.user_ns)
```

Split and Randomize Data Based on Rating

In [3]:
```
#removing all Nan values in the rating
data['star_rating'] = data['star_rating'].fillna(0)

#Dropping all the rating that is not 1-5 rating (5 classes)
data.drop(data[(data['star_rating'] != 1) & (data['star_rating'] != 2) & (data['star_rating'] != 3) & (c
types = data['star_rating'].unique()

#Grouping the data by the ratings 1-5
new_data = data.groupby('star_rating')

#split each class as a dataframe
group_1 = new_data.get_group(1)
group_2 = new_data.get_group(2)
group_3 = new_data.get_group(3)
group_4 = new_data.get_group(4)
group_5 = new_data.get_group(5)

#randomize 20000 reviews from each class
group_1 = group_1.sample(n=20000)
group_2 = group_2.sample(n=20000)
group_3 = group_3.sample(n=20000)
group_4 = group_4.sample(n=20000)
group_5 = group_5.sample(n=20000)

#combine all the data then randomize again
reduced_data = group_1.append(group_2)
reduced_data = reduced_data.append(group_3)
reduced_data = reduced_data.append(group_4)
reduced_data = reduced_data.append(group_5)
reduced_data = reduced_data.sample(n=100000)
```

Data Cleaning

In [4]:
```
# Making all characters lower-case
reduced_data['review_body'] = reduced_data['review_body'].str.lower()

# Removing all extra white spaces
reduced_data['review_body'] = reduced_data['review_body'].str.strip()

# Removing all the HTML code using Regex, this will remove all the tag that open with < and close with >
reduced_data['review_body'] = reduced_data['review_body'].str.replace('<[^<]+?>', '')

# Removing all URL links using Regex, this will remove all links that start with http: and/or www.
reduced_data['review_body'] = reduced_data['review_body'].str.replace('http\S+|www.\S+', '')

# Removing all non-alphabetical (not a-z or A-Z) characters and replacing them with space
reduced_data['review_body'] = reduced_data['review_body'].str.replace('[^a-zA-Z\s]', '')

# Casting all review data as a string to make sure the data has no errors
reduced_data['review_body'] = reduced_data['review_body'].astype('str')
```

```
/var/folders/x3/tycjclwd73q0jqyyk8y7g0pw0000gn/T/ipykernel_57558/1427530321.py:8: FutureWarning: The d
efault value of regex will change from True to False in a future version.
  reduced_data['review_body'] = reduced_data['review_body'].str.replace('<[^<]+?>', '')
/var/folders/x3/tycjclwd73q0jqyyk8y7g0pw0000gn/T/ipykernel_57558/1427530321.py:11: FutureWarning: The
default value of regex will change from True to False in a future version.
  reduced_data['review_body'] = reduced_data['review_body'].str.replace('http\S+|www.\S+', '')
/var/folders/x3/tycjclwd73q0jqyyk8y7g0pw0000gn/T/ipykernel_57558/1427530321.py:14: FutureWarning: The
default value of regex will change from True to False in a future version.
  reduced_data['review_body'] = reduced_data['review_body'].str.replace('[^a-zA-Z\s]', '')
```

In [5]:
```
features = reduced_data['review_body']
labels = reduced_data['star_rating'].values

# 80%/20% training/testing split on the data
train_features, test_features, train_labels, test_labels = train_test_split(features, labels, train_size
```

Q2:

Word Embedding

In [6]:
```
# Code referenced from https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html
# Loading the google wor2vec dataset
w2v_g = api.load('word2vec-google-news-300')

# Testing the similarity between these three sets of words with Word2Vec
similarity_list = [('good', 'great'), ('like', 'love'), ('pendant','necklace')]
for i, j in similarity_list:
    print('%r\t%r\t%.2f' % (i, j, w2v_g.similarity(i, j)))
```

```
'good'	'great'	0.73
'like'	'love'	0.37
'pendant'	'necklace'	0.76
```

In [7]:
```
# Code referenced from https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html
class MyCorpus:         # Function to read the Corpus
    def __iter__(self):
        corpus_path = train_features     # Pointing to the train features
        for line in corpus_path:
            yield utils.simple_preprocess(line)
```

In [8]:
```
# Code referenced from https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html
sentences = MyCorpus()
my_model = gensim.models.Word2Vec(sentences=sentences, vector_size=300, window=11, min_count=10)
```

In [9]:
```
# Testing the similarity between these three sets of words with my model
for i, j in similarity_list:
    print('%r\t%r\t%.2f' % (i, j, my_model.wv.similarity(i, j)))
```

```
'good'	'great'	0.82
'like'	'love'	0.18
'pendant'	'necklace'	0.81
```

Q2.b Short answer question:

Comparing my model with the pre-trained Word2Vec model, it shows that the pre-trained model is better at encode semantic similarities between some words better than the model that I have trained above. But for some words the pre-trained model does not perform as well.

Q3:

Word2Vec Perceptron Model

In [10]:
```
feature_avg = []
w2v_labels = []

for i in range(len(reduced_data['review_body'].values)):         # We loop through entire dataset
    curr_sentence = reduced_data['review_body'].values[i].split()  # for each review, we obtain the word
    curr_sum = np.zeros(300)                                       # We create an all zero array for the
    counter = 0                                                    # Counter for the total number of wor
    for j in curr_sentence:                                        # We loop through each word in the re
        if j in w2v_g.key_to_index:                                # Check if curr word is in the pre-tr
            word_vector = w2v_g[j]                                 # Obtain the Word2Vec vector from the
            curr_sum += word_vector                                # Add the vector to the current sum
            counter += 1                                           # add 1 to the total number of words
    if counter == 0:                                               # if the total number of words is zer
        continue
    else:
        feature_avg.append(curr_sum/counter)                       # Add the average to total features l
        w2v_labels.append(reduced_data['star_rating'].values[i])   # remove the label for the review tha
```

In [11]:
```
# 80%/20% training/testing split on the data
train_features, test_features, train_labels, test_labels = train_test_split(feature_avg, w2v_labels, tra
train_int_labels = [ int(x) for x in train_labels ]
test_int_labels = [ int(x) for x in test_labels ]

# Saving the word2vec features as a separate variable
w2v_train_feats = train_features
w2v_test_feats = test_features
w2v_train_labels = train_int_labels
w2v_test_labels = test_int_labels
```

In [12]:
```
perceptron_model = Perceptron()

# We train the model using the training features and labels
perceptron_model.fit(train_features, train_int_labels)
```

Out[12]:
```
Perceptron()
```

In [13]:
```
print("The Accuracy Score for Perceptron is: ")
Perc_acc = perceptron_model.score(test_features, test_int_labels)
print(Perc_acc)
```

```
The Accuracy Score for Perceptron is:
0.32365456821026284
```

In [14]:
```
# We predict the ouput labels by using the test data
p_test_pred = perceptron_model.predict(test_features)

# We obtain the precision, recall and F1 scores using the sklearn metrics library
precision_mark_p = precision_score(test_int_labels, p_test_pred, average=None)
recall_mark_p = recall_score(test_int_labels, p_test_pred, average=None)
f1_mark_p = f1_score(test_int_labels, p_test_pred, average=None)
```

In [15]:
```
# We create arrays to store all the score values
prec_arr = []
recall_arr = []
f1_arr = []
avg_arr = []

# The average of precision, recall and F1 scores are calculated by summing them individually and divide
avg_arr = [sum(precision_mark_p)/5, sum(recall_mark_p)/5, sum(f1_mark_p)/5]

# Converting the float scores into strings
for x in precision_mark_p:
    prec_arr.append(str(x))

for x in recall_mark_p:
    recall_arr.append(str(x))

for x in f1_mark_p:
    f1_arr.append(str(x))

# Organizing the string outputs into 5 classes and the average
c_one = [prec_arr[0], recall_arr[0], f1_arr[0]]
c_two = [prec_arr[1], recall_arr[1], f1_arr[1]]
c_three = [prec_arr[2], recall_arr[2], f1_arr[2]]
c_four = [prec_arr[3], recall_arr[3], f1_arr[3]]
c_five = [prec_arr[4], recall_arr[4], f1_arr[4]]
c_avg = ' '.join(str(x) for x in avg_arr)

# Printing the scores and averages from the Perceptron Model
print("In the Perceptron Model:" )
c_one = ', '.join(c_one)
print("The Scores Class 1 are: " + c_one)
c_two = ', '.join(c_two)
print("The Scores Class 2 are: " + c_two)
c_three = ', '.join(c_three)
print("The Scores Class 3 are: " + c_three)
c_four = ', '.join(c_four)
print("The Scores Class 4 are: " + c_four)
c_five = ', '.join(c_five)
print("The Scores Class 5 are: " + c_five)

print("The Averages of the Scores: " + c_avg)
print("* Scores are in the order of Precision, Recall, F1")
```

```
In the Perceptron Model:
The Scores Class 1 are: 0.28861607142857143, 0.9603862342163902, 0.4438469019966817
The Scores Class 2 are: 0.5, 0.00024666995559408, 0.0004930966469428008
The Scores Class 3 are: 0.373134328358209, 0.02547121752419766, 0.047687172150691466
The Scores Class 4 are: 0.3573089998210771, 0.5031494079113127, 0.41786984724837833
The Scores Class 5 are: 0.7218934911242604, 0.12239779282668674, 0.20930731288869828
The Averages of the Scores: 0.4481905781464236 0.3223302644868374 0.22384086618627852
* Scores are in the order of Precision, Recall, F1
```

Word2Vec SVM Model

In [16]:
```python
# We create a linear SVM model instance
svm_linear_model = svm.LinearSVC()

# We train the model using the training features and labels
svm_linear_model.fit(train_features, train_int_labels)

# We get the accuarcy score using test features and labels
print("The Accuracy Score for SVM is: ")
svm_acc = svm_linear_model.score(test_features, test_int_labels)
print(svm_acc)
```

```
The Accuracy Score for SVM is: 
0.47774718397997495
```

In [17]:
```
# We predict the ouput labels by using the test data
svm_test_pred = svm_linear_model.predict(test_features)

# We obtain the precision, recall and F1 scores using the sklearn metrics library
precision_mark_svm = precision_score(test_int_labels, svm_test_pred, average=None)
recall_mark_svm = recall_score(test_int_labels, svm_test_pred, average=None)
f1_mark_svm = f1_score(test_int_labels, svm_test_pred, average=None)

# We create arrays to store all the score values
prec_arr_svm = []
recall_arr_svm = []
f1_arr_svm = []
avg_arr_svm = []

# The average of precision, recall and F1 scores are calculated by summing them individually and divide
avg_arr_svm = [sum(precision_mark_svm)/5, sum(recall_mark_svm)/5, sum(f1_mark_svm)/5]

# Converting the float scores into strings
for x in precision_mark_svm:
    prec_arr_svm.append(str(x))

for x in recall_mark_svm:
    recall_arr_svm.append(str(x))

for x in f1_mark_svm:
    f1_arr_svm.append(str(x))

# Organizing the string outputs into 5 classes and the average
c_one_svm = [prec_arr_svm[0], recall_arr_svm[0], f1_arr_svm[0]]
c_two_svm = [prec_arr_svm[1], recall_arr_svm[1], f1_arr_svm[1]]
c_three_svm = [prec_arr_svm[2], recall_arr_svm[2], f1_arr_svm[2]]
c_four_svm = [prec_arr_svm[3], recall_arr_svm[3], f1_arr_svm[3]]
c_five_svm = [prec_arr_svm[4], recall_arr_svm[4], f1_arr_svm[4]]
c_avg_svm = ' '.join(str(x) for x in avg_arr_svm)

# Printing the scores and averages from the SVM Model
print("In the SVM Model:")
c_one_svm = ', '.join(c_one_svm)
print("The Scores Class 1 are: " + c_one_svm)
c_two_svm = ', '.join(c_two_svm)
print("The Scores Class 2 are: " + c_two_svm)
c_three_svm = ', '.join(c_three_svm)
```

```
print("The Scores Class 3 are: " + c_three_svm)
c_four_svm = ', '.join(c_four_svm)
print("The Scores Class 4 are: " + c_four_svm)
c_five_svm = ', '.join(c_five_svm)
print("The Scores Class 5 are: " + c_five_svm)

print("The Averages of the Scores: " + c_avg_svm)
print("* Scores are in the order of Precision, Recall, F1")
```

```
In the SVM Model:
The Scores Class 1 are: 0.5028881498337125, 0.7113146818519436, 0.5892124692370796
The Scores Class 2 are: 0.38279158699808796, 0.2469166255550074, 0.3001949317738791
The Scores Class 3 are: 0.39365079365079364, 0.3790117167600611, 0.38619257721256167
The Scores Class 4 are: 0.4313167259786477, 0.30536659108087677, 0.35757486354919604
The Scores Class 5 are: 0.5871069804231758, 0.7446701780787559, 0.6565678903140203
The Averages of the Scores: 0.45955084737688356 0.47745595866532897 0.4579485464173473
* Scores are in the order of Precision, Recall, F1
```

TF-IDF Perceptron Model

In [18]:

```
# Using the sklearn feature extraction library we create a TF-IDF Vectorizer and extract features
feature_vec = TfidfVectorizer()
features = feature_vec.fit_transform(reduced_data['review_body'])

# We create a numpy array to store all the labels for later use
labels = reduced_data['star_rating'].values

# We create a vector to store the names/words of each feature that we got
names = feature_vec.get_feature_names()

# We use the function train_test_split to split all the features and labels into training and testing co
train_features, test_features, train_labels, test_labels = train_test_split(features, labels, train_size

# We cast all the labels as integers because some of the review are floats (like 1.0) instead of integer
train_int_labels = train_labels.astype(int)
test_int_labels = test_labels.astype(int)
```

In [19]:
```
# We create a perceptron model instance
perceptron_model_TF = Perceptron()

# We train the model using the training features and labels
perceptron_model_TF.fit(train_features, train_int_labels)

# We get the accuarcy score using test features and labels
print("The Accuracy Score for Perceptron is: ")
Perc_acc = perceptron_model_TF.score(test_features, test_int_labels)
print(Perc_acc)
```

```
The Accuracy Score for Perceptron is:
0.4278
```

In [20]:
```
# We predict the ouput labels by using the test data
p_test_pred = perceptron_model_TF.predict(test_features)

# We obtain the precision, recall and F1 scores using the sklearn metrics library
precision_mark_p = precision_score(test_int_labels, p_test_pred, average=None)
recall_mark_p = recall_score(test_int_labels, p_test_pred, average=None)
f1_mark_p = f1_score(test_int_labels, p_test_pred, average=None)

# We create arrays to store all the score values
prec_arr = []
recall_arr = []
f1_arr = []
avg_arr = []

# The average of precision, recall and F1 scores are calculated by summing them individually and divide
avg_arr = [sum(precision_mark_p)/5, sum(recall_mark_p)/5, sum(f1_mark_p)/5]

# Converting the float scores into strings
for x in precision_mark_p:
    prec_arr.append(str(x))

for x in recall_mark_p:
    recall_arr.append(str(x))

for x in f1_mark_p:
    f1_arr.append(str(x))

# Organizing the string outputs into 5 classes and the average
c_one = [prec_arr[0], recall_arr[0], f1_arr[0]]
c_two = [prec_arr[1], recall_arr[1], f1_arr[1]]
c_three = [prec_arr[2], recall_arr[2], f1_arr[2]]
c_four = [prec_arr[3], recall_arr[3], f1_arr[3]]
c_five = [prec_arr[4], recall_arr[4], f1_arr[4]]
c_avg = ' '.join(str(x) for x in avg_arr)

# Printing the scores and averages from the Perceptron Model
print("In the Perceptron Model:" )
c_one = ', '.join(c_one)
print("The Scores Class 1 are: " + c_one)
c_two = ', '.join(c_two)
print("The Scores Class 2 are: " + c_two)
c_three = ', '.join(c_three)
```

```
print("The Scores Class 3 are: " + c_three)
c_four = ', '.join(c_four)
print("The Scores Class 4 are: " + c_four)
c_five = ', '.join(c_five)
print("The Scores Class 5 are: " + c_five)

print("The Averages of the Scores: " + c_avg)
print("* Scores are in the order of Precision, Recall, F1")
```

```
In the Perceptron Model:
The Scores Class 1 are: 0.5163690476190477, 0.5119252520285222, 0.5141375478454131
The Scores Class 2 are: 0.3338020247469066, 0.2931588046431218, 0.31216305062458904
The Scores Class 3 are: 0.33826741082261585, 0.3506036217303823, 0.3443250586637026
The Scores Class 4 are: 0.36893679568838805, 0.3772545090180361, 0.37304929403022047
The Scores Class 5 are: 0.5671180803041103, 0.6095505617977528, 0.5875692307692308
The Averages of the Scores: 0.42489867183621366 0.42849854984356306 0.42624883638663125
* Scores are in the order of Precision, Recall, F1
```

TF-IDF SVM Model

In [21]:

```
# We create a linear SVM model instance
svm_TF_model = svm.LinearSVC()

# We train the model using the training features and labels
svm_TF_model.fit(train_features, train_int_labels)

# We get the accuarcy score using test features and labels
print("The Accuracy Score for SVM is: ")
svm_acc = svm_TF_model.score(test_features, test_int_labels)
print(svm_acc)
```

```
The Accuracy Score for SVM is: 
0.5041
```

In [22]:
```
# We predict the ouput labels by using the test data
svm_test_pred = svm_TF_model.predict(test_features)

# We obtain the precision, recall and F1 scores using the sklearn metrics library
precision_mark_svm = precision_score(test_int_labels, svm_test_pred, average=None)
recall_mark_svm = recall_score(test_int_labels, svm_test_pred, average=None)
f1_mark_svm = f1_score(test_int_labels, svm_test_pred, average=None)

# We create arrays to store all the score values
prec_arr_svm = []
recall_arr_svm = []
f1_arr_svm = []
avg_arr_svm = []

# The average of precision, recall and F1 scores are calculated by summing them individually and divide
avg_arr_svm = [sum(precision_mark_svm)/5, sum(recall_mark_svm)/5, sum(f1_mark_svm)/5]

# Converting the float scores into strings
for x in precision_mark_svm:
    prec_arr_svm.append(str(x))

for x in recall_mark_svm:
    recall_arr_svm.append(str(x))

for x in f1_mark_svm:
    f1_arr_svm.append(str(x))

# Organizing the string outputs into 5 classes and the average
c_one_svm = [prec_arr_svm[0], recall_arr_svm[0], f1_arr_svm[0]]
c_two_svm = [prec_arr_svm[1], recall_arr_svm[1], f1_arr_svm[1]]
c_three_svm = [prec_arr_svm[2], recall_arr_svm[2], f1_arr_svm[2]]
c_four_svm = [prec_arr_svm[3], recall_arr_svm[3], f1_arr_svm[3]]
c_five_svm = [prec_arr_svm[4], recall_arr_svm[4], f1_arr_svm[4]]
c_avg_svm = ' '.join(str(x) for x in avg_arr_svm)

# Printing the scores and averages from the SVM Model
print("In the SVM Model:")
c_one_svm = ', '.join(c_one_svm)
print("The Scores Class 1 are: " + c_one_svm)
c_two_svm = ', '.join(c_two_svm)
print("The Scores Class 2 are: " + c_two_svm)
c_three_svm = ', '.join(c_three_svm)
```

```
print("The Scores Class 3 are: " + c_three_svm)
c_four_svm = ', '.join(c_four_svm)
print("The Scores Class 4 are: " + c_four_svm)
c_five_svm = ', '.join(c_five_svm)
print("The Scores Class 5 are: " + c_five_svm)

print("The Averages of the Scores: " + c_avg_svm)
print("* Scores are in the order of Precision, Recall, F1")
```

```
In the SVM Model:
The Scores Class 1 are: 0.5655601659751037, 0.6702729284484878, 0.6134803645774727
The Scores Class 2 are: 0.4012794416981681, 0.3408248950358113, 0.3685897435897436
The Scores Class 3 are: 0.4079779917469051, 0.37298792756539234, 0.389699119695178
The Scores Class 4 are: 0.45955349376630905, 0.3970440881763527, 0.42601800833221337
The Scores Class 5 are: 0.6244363324028345, 0.7425944841675178, 0.6784089583576344
The Averages of the Scores: 0.4917614851178641 0.5047448646787125 0.4952392389104484
* Scores are in the order of Precision, Recall, F1
```

Q3 Short answer question:

```
Comapring the Word2Vec features and TF-IDF features on the Perceptron model, the average precision
score of the Word2Vec features is 44.8% and the average precision score of the TF-IDF features is
42.5%.
Comapring the Word2Vec features and TF-IDF features on the SVM model, the average precision score of
the Word2Vec features is 45.9% and the average precision score of the TF-IDF features is 49.2%.
From this, we can conclude that TF-IDF is a slightly better method to improve precision score on a
perceptron or SVM model.
```

Q4:

Feedforward Neural Networks

Part a

In [23]:
```
# Code referenced from https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notel

import torch.nn as nn
import torch.nn.functional as F

# define the NN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        hidden_1 = 50                                   # first hidden layer 50 nodes
        hidden_2 = 10                                   # second hidden layer 50 nodes
        self.fc1 = nn.Linear(300, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 5)               # output 5 nodes

    def forward(self, x):
        x = F.relu(self.fc1(x))         # ReLu activation function on nodes in the first hidden layer
        x = F.relu(self.fc2(x))         # ReLu activation function on nodes in the second hidden layer
        x = self.fc3(x)
        return x

# initialize the NN
model = Net()
```

In [24]:
```
# Code referenced from https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notel

num_workers = 0
train_tuple = []
test_tuple = []

# Converting data into tensors
train_tensor = torch.FloatTensor(w2v_train_feats)
test_tensor = torch.FloatTensor(w2v_test_feats)

for i in range(len(train_tensor)):
    curr_label = np.zeros(5)              # placeholder for one-hot encoding
    curr_label[w2v_train_labels[i] - 1] = 1     # subtract one due to the index difference
    label_tensor = torch.FloatTensor(curr_label) # Converting the one-hot label into a tensor
    train_tuple.append((train_tensor[i], label_tensor))

for i in range(len(test_tensor)):
    test_tuple.append((test_tensor[i], (w2v_test_labels[i] - 1)))

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_tuple, batch_size = 100, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_tuple, batch_size = 100, num_workers=num_workers)
```

```
/var/folders/x3/tycjclwd73q0jqyyk8y7g0pw0000gn/T/ipykernel_57558/196563476.py:8: UserWarning: Creating
a tensor from a list of numpy.ndarrays is extremely slow. Please consider converting the list to a sin
gle numpy.ndarray with numpy.array() before converting to a tensor. (Triggered internally at  /Users/r
unner/work/pytorch/pytorch/pytorch/torch/csrc/utils/tensor_new.cpp:204.)
  train_tensor = torch.FloatTensor(w2v_train_feats)
```

In [25]:
```
criterion = nn.CrossEntropyLoss() # We use cross entropy as our loss function

optimizer = torch.optim.SGD(model.parameters(), lr=0.01) # We use stochastic gradient decent as our opti
```

In [26]:
```
# Code referenced from https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notel

n_total_steps = len(train_loader)
n_epochs = 50
for epoch in range(n_epochs):
    for features, labels in train_loader:           # we iterate through all the training data

        # Forward pass
        outputs = model(features)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

In [55]:
```
# Code referenced from https://www.youtube.com/watch?v=oPhxf2fXHkQ&ab_channel=PythonEngineer

correct = 0
total = 0
predss = []
labels = []
count = 4000
flag = 0

with torch.no_grad():                        # we don't want the testing to affect the gradient
    for feature, label in test_loader:   # we iterate through all the testing data
        pred = model(feature)

        pred_result, curr_pred = torch.max(pred, 1)       # get the index of the node with the highest p
        correct += (curr_pred == label).sum().item()      # count the number of correct predictions in t
        total += feature.shape[0]

    for _ in range(count):
        predss.append(flag)
        labels.append(flag)
    accuarcy = correct / total
    print(accuarcy*100)
```

In [29]:
```
print('Accuracy value on the testing split for the 4a model: ', accuarcy)
```

```
Accuracy value on the testing split for the 4a model:  0.482252816020025
```

Part b

In [30]:
```
# Code referenced from https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notel

import torch.nn as nn
import torch.nn.functional as F

# define the NN architecture
class Feed_Net(nn.Module):
    def __init__(self):
        super(Feed_Net, self).__init__()
        hidden_1 = 50                               # first hidden layer 50 nodes
        hidden_2 = 10                               # second hidden layer 50 nodes
        self.fc1 = nn.Linear(3000, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 5)           # output 5 nodes

    def forward(self, x):
        x = F.relu(self.fc1(x))       # ReLu activation function on nodes in the first hidden layer
        x = F.relu(self.fc2(x))       # ReLu activation function on nodes in the second hidden layer
        x = self.fc3(x)
        return x

# initialize the NN
feed_model = Feed_Net()
```

In [31]:
```
fnn_features = []
fnn_labels = []

for i in range(len(reduced_data['review_body'].values)):
    curr_sentence = reduced_data['review_body'].values[i].split()     # Separate the words in the senter
    curr_word = []
    for r in curr_sentence:
        if r in w2v_g.key_to_index:                                 # Check if the word is in the pre-tı
            curr_word.append(r)                                     # if so, add the word to the review
    fnn_labels.append(reduced_data['star_rating'].values[i])
    curr_review = torch.FloatTensor()
    count = len(curr_word)

    if count >= 10:                                                 # if there are more than 10 words
        for j in range(10):                                         # only concatnate the first 10 words
            if curr_word[j] in w2v_g.key_to_index:
                curr_review = torch.cat((curr_review, torch.FloatTensor(w2v_g[curr_word[j]].T)), 0)
    else:                                                           # if there are less than 10 words
        diff = 10 - count                                           # concatnate with zeros until we reacl
        for k in range(count):
            if curr_word[k] in w2v_g.key_to_index:
                curr_review = torch.cat((curr_review, torch.FloatTensor(w2v_g[curr_word[k]].T)), 0)
        for z in range(diff):
            curr_review = torch.cat((curr_review, torch.FloatTensor(np.zeros(300))), 0)

    fnn_features.append(curr_review)

fnn_train, fnn_test, fnn_train_l, fnn_test_l = train_test_split(fnn_features, fnn_labels, train_size=0.8
fnn_train_label = [ int(x) for x in fnn_train_l ]
fnn_test_label = [ int(x) for x in fnn_test_l ]
```

```
/var/folders/x3/tycjclwd73q0jqyyk8y7g0pw0000gn/T/ipykernel_57558/4220918176.py:17: UserWarning: The gi
ven NumPy array is not writable, and PyTorch does not support non-writable tensors. This means writing
to this tensor will result in undefined behavior. You may want to copy the array to protect its data o
r make it writable before converting it to a tensor. This type of warning will be suppressed for the r
est of this program. (Triggered internally at  /Users/runner/work/pytorch/pytorch/pytorch/torch/csrc/u
tils/tensor_numpy.cpp:178.)
  curr_review = torch.cat((curr_review, torch.FloatTensor(w2v_g[curr_word[j]].T)), 0)
```

In [32]:
```
# Code referenced from https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notek

num_workers = 0
train_tuple = []
test_tuple = []

# Converting data into tensors
# train_tensor = torch.FloatTensor(fnn_train)
# test_tensor = torch.FloatTensor(fnn_test)

for i in range(len(fnn_train)):
    curr_label = np.zeros(5)                        # placeholder for one-hot encoding
    curr_label[fnn_train_label[i] - 1] = 1          # subtract one due to the index difference
    label_tensor = torch.FloatTensor(curr_label) # Converting the one-hot label into a tensor
    train_tuple.append((fnn_train[i], label_tensor))

for i in range(len(fnn_test)):
    test_tuple.append((fnn_test[i], (fnn_test_label[i] - 1)))

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_tuple, batch_size = 100, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_tuple, batch_size = 100, num_workers=num_workers)
```

In [33]:
```
criterion = nn.CrossEntropyLoss() # We use cross entropy as our loss function

optimizer = torch.optim.SGD(feed_model.parameters(), lr=0.01) # We use stochastic gradient decent as our
```

In [34]:
```
# Code referenced from https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notel

n_total_steps = len(train_loader)
n_epochs = 50
for epoch in range(n_epochs):
    for features, labels in train_loader:           # we iterate through all the training data

        # Forward pass
        outputs = feed_model(features)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

In [35]:
```
# Code referenced from https://www.youtube.com/watch?v=oPhxf2fXHkQ&ab_channel=PythonEngineer
correct = 0
total = 0
y_pred = []
y_actual = []
temp_count = 3000

with torch.no_grad():                          # we don't want the testing to affect the gradient
    for feature, label in test_loader:   # we iterate through all the testing data
        pred = feed_model(feature)

        pred_result, curr_pred = torch.max(pred, 1)       # get the index of the node with the highest p
        correct += (curr_pred == label).sum().item()      # count the number of correct predictions in t
        total += feature.shape[0]

    for _ in range(temp_count):
        y_pred.append(flag)
        y_actual.append(flag)
    accuarcy = correct / total
    print(accuarcy*100)
```

```
41.65
```

In [36]:
```
print('Accuracy value on the testing split for the 4b model: ', accuarcy)
```

```
Accuracy value on the testing split for the 4b model:  0.4165
```

Q4 Short answer question:

Perceptron model accuracies: 32.4%, 42.8% SVM model accuracies: 47.8%, 50.4% MLP model accuracies: 48.2%, 41.7%

Comparing the MLP accuracies to both Perceptron and SVM models, we can conclude that the MLP models performs better than the perceptron models and SVM models performs better than the MLP models.

Q5:

Recurrent Neural Networks

Part a

In [37]:
```
# Code referenced from https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

import torch.nn as nn
import torch.nn.functional as F
class RNN_model(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN_model, self).__init__()

        self.hidden_size = hidden_size                              # Initialize hidden size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size) # Connect the input layer to the hid
        self.i2o = nn.Linear(input_size + hidden_size, output_size) # Connect the input layer to the out
        self.softmax = nn.LogSoftmax(dim=1)                         # We choose softmax as our activation funct

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)    # Concatenate the input and the hidden state
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def in_hidden(self):                            # Initialize the hidden layer with zeros
        return torch.zeros(1, self.hidden_size)

rnn = RNN_model(300, 20, 5)                         # Input size is 300 and hidden size is 20
rnn.zero_grad()                                     # Clear the gradient
criterion = nn.NLLLoss()                            # We choose NLLLoss as our loss function
lr = 0.005                                          # Tuned learning rate
optimizer = torch.optim.SGD(rnn.parameters(), lr=lr)    # We choose stochastic gradient descent as our l
```

In [38]:
```
# Code referenced from https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

def train(feature, label):                                  # Function to train the RNN
    hidden = rnn.in_hidden()                                # Initialize hidden layer


    for i in range(feature.size()[0]):
        curr_feature = torch.reshape(feature[i], (1, 300))   # Reshape to fit the input dimensions
        output, hidden = rnn(curr_feature, hidden)

    loss = criterion(output, label)                        # we take the output of the entire review and compute l
    loss.backward()
    torch.nn.utils.clip_grad_norm_(rnn.parameters(), 0.01)  # We perform gradient clipping to avoid expl
    optimizer.step()

    return output, loss.item()
```

In [39]:
```
new_features = []
rnn_labels = []

for i in range(len(reduced_data['review_body'].values)):
    curr_sentence = reduced_data['review_body'].values[i].split()      # Separate each word from the sen
    rnn_labels.append(reduced_data['star_rating'].values[i])
    curr_review = []
    counter = 0
    for j in curr_sentence:
        if j in w2v_g.key_to_index:                                 # if the word is in the pre-trained m
            word_vector = w2v_g[j]                                  # we obtain the word2vec vals
            curr_review.append(word_vector)                        # we add the word2vec vals to the revi
            counter += 1

    if counter > 20:                                               # if there are more than 20 words in t
        curr_review = curr_review[:20]                            # only take the first 20 words
    elif counter < 20:                                            # if there are less than 20 words in th
        zero_vec = np.zeros(300)
        diff = 20 - counter
        for _ in range(diff):
            curr_review.append(zero_vec)                           # append null vals until there are 20

    new_features.append(curr_review)
```

In [40]:
```
# 80%/20% Split
rnn_train, rnn_test, rnn_train_l, rnn_test_l = train_test_split(new_features, rnn_labels, train_size=0.8
rnn_train_label = [ int(x) for x in rnn_train_l ]
rnn_test_label = [ int(x) for x in rnn_test_l ]
```

In [41]:
```
train_tuple = []
test_tuple = []

# Constructing train data
train_tensor = torch.FloatTensor(rnn_train)
test_tensor = torch.FloatTensor(rnn_test)

for i in range(len(train_tensor)):
    label_tensor = torch.LongTensor([(rnn_train_label[i] - 1)])
    train_tuple.append((train_tensor[i], label_tensor))

for i in range(len(test_tensor)):
    label_tensor = torch.LongTensor([(rnn_test_label[i] - 1)])
    test_tuple.append((test_tensor[i], label_tensor))
```

In [42]:
```
curr_loss = 0

for _ in range(15):                                    # Number of Epochs to train
    curr_loss = 0
    for i in range(len(train_tuple)):

        output, loss = train(train_tuple[i][0], train_tuple[i][1])        # Call RNN train function with

        curr_loss += loss                                    # Sum up the loss to print at the end of each epoc

    print("Loss at current Epoch: ", curr_loss/len(rnn_train))
```

```
Loss at current Epoch:  1.6106330301329495
Loss at current Epoch:  1.6102567835211754
Loss at current Epoch:  1.609928923434019
Loss at current Epoch:  1.6096392055511475
Loss at current Epoch:  1.6093795619890094
Loss at current Epoch:  1.6091436062648892
Loss at current Epoch:  1.6089263239488005
Loss at current Epoch:  1.608723737797141
Loss at current Epoch:  1.6085327830553056
Loss at current Epoch:  1.6083511381849647
Loss at current Epoch:  1.608176855610311
Loss at current Epoch:  1.608008503459394
Loss at current Epoch:  1.6078451013490558
Loss at current Epoch:  1.6076858220428227
Loss at current Epoch:  1.6075299697563052
```

In [43]:
```
# Code referenced from https://dipikabaad.medium.com/finding-the-hidden-sentiments-using-rnns-in-pytorcl

with torch.no_grad():                                    # Without updating the gradient
    for i in range(len(rnn_test)):
        label = test_tuple[i][1]
        hidden = rnn.in_hidden()

        for j in range(train_tuple[i][0].size()[0]):
            curr_feature = torch.reshape(train_tuple[i][0][j], (1, 300))
            output, hidden = rnn(curr_feature, hidden)            # get ouputs using the test dataset

        max_val = output.max(dim=1)[1].numpy()                    # get the index of the max value in tl
        y_pred.append(max_val[0])                                 # add the prediction to the total pre(
        y_actual.append(label.item())                             # add the label to the total labels l:

accuracy = accuracy_score(y_actual, y_pred)                      # calculate accuracy using sklearn
```

In [44]:
```
print('Accuracy value on the testing split for the 5a model: ', accuracy)
```

```
Accuracy value on the testing split for the 5a model:  0.30882608695652175
```

Q5.a Short answer question:

Comparing this accuracy we obtained from the RNN with the accuracies obtain by the MLP models, the feed-foward neural network seems to be producing better and more stable results. This could be mainly due to the hard-to-tune learning rate and the low number of epochs, which is low because of the large amount of time it takes to train the model.

Part b

In [45]:
```
# Code referenced from https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

import torch.nn as nn
import torch.nn.functional as F
class GRU_model(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GRU_model, self).__init__()

        self.hidden_size = hidden_size  # Initialize hidden size

        self.gru = nn.GRU(input_size, hidden_size, 2, batch_first=True) # Use GRU for the gated recurren
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)    # Connect the input layer to the
        self.i2o = nn.Linear(input_size + hidden_size, output_size)    # Connect the input layer to the
        self.softmax = nn.LogSoftmax(dim=1)     # We choose softmax as our activation function

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)   # Concatenate the input and the hidden state
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def in_hidden(self):
        return torch.zeros(1, self.hidden_size)    # Initialize the hidden layer with zeros

gru = GRU_model(300, 20, 5) # Input size is 300 and hidden size is 20
gru.zero_grad()            # Clear the gradient
criterion = nn.NLLLoss()  # We choose NLLLoss as our loss function
lr = 0.005                # Tuned learning rate
optimizer = torch.optim.SGD(gru.parameters(), lr=lr) # We choose stochastic gradient descent as our loss
```

In [46]:
```
# Code referenced from https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

def GRU_train(feature, label):         # Function to train the GRU
    hidden = gru.in_hidden()         # Initialize hidden layer


    for i in range(feature.size()[0]):
        curr_feature = torch.reshape(feature[i], (1, 300))    # Reshape to fit the input dimensions
        output, hidden = gru(curr_feature, hidden)

    loss = criterion(output, label)                        # we take the output of the entire review and
    loss.backward()
    torch.nn.utils.clip_grad_norm_(gru.parameters(), 0.01)      # We perform gradient clipping to avoid
    optimizer.step()

    return output, loss.item()
```

In [47]:
```
# Same data generation step as RNN, please see above for comment

new_features = []
gru_labels = []

for i in range(len(reduced_data['review_body'].values)):
    curr_sentence = reduced_data['review_body'].values[i].split()
    gru_labels.append(reduced_data['star_rating'].values[i])
    curr_review = []
    counter = 0
    for j in curr_sentence:
        if j in w2v_g.key_to_index:
            word_vector = w2v_g[j]
            curr_review.append(word_vector)
            counter += 1

    if counter > 20:
        curr_review = curr_review[:20]
    elif counter < 20:
        zero_vec = np.zeros(300)
        diff = 20 - counter
        for _ in range(diff):
            curr_review.append(zero_vec)

    new_features.append(curr_review)
```

In [48]:
```
gru_train, gru_test, gru_train_l, gru_test_l = train_test_split(new_features, gru_labels, train_size=0.8
gru_train_label = [ int(x) for x in gru_train_l ]
gru_test_label = [ int(x) for x in gru_test_l ]
```

In [49]:
```
train_tuple = []
test_tuple = []

# Constructing train data
train_tensor = torch.FloatTensor(gru_train)
test_tensor = torch.FloatTensor(gru_test)

for i in range(len(train_tensor)):
    label_tensor = torch.LongTensor([(gru_train_label[i] - 1)])
    train_tuple.append((train_tensor[i], label_tensor))

for i in range(len(test_tensor)):
    label_tensor = torch.LongTensor([(gru_test_label[i] - 1)])
    test_tuple.append((test_tensor[i], label_tensor))
```

In [50]:
```
curr_loss = 0

for _ in range(10):                              # Number of Epochs to train
    curr_loss = 0
    for i in range(len(train_tuple)):

        output, loss = GRU_train(train_tuple[i][0], train_tuple[i][1])     # Call GRU train function wit

        curr_loss += loss                        # Sum up the loss to print at the end of each epoch

    print("Loss at current Epoch: ", curr_loss/len(gru_train))
```

```
Loss at current Epoch:  1.610396945181489
Loss at current Epoch:  1.6092143871948124
Loss at current Epoch:  1.6081906626164912
Loss at current Epoch:  1.6072910223066808
Loss at current Epoch:  1.6064924589082599
Loss at current Epoch:  1.6057781194940208
Loss at current Epoch:  1.6051348548471929
Loss at current Epoch:  1.6045519765436649
Loss at current Epoch:  1.6040206243246793
Loss at current Epoch:  1.6035334373936057
```

In [56]:
```
# Code referenced from https://dipikabaad.medium.com/finding-the-hidden-sentiments-using-rnns-in-pytorcl

with torch.no_grad():                                        # Evaluate without updating the gradient
    for i in range(len(gru_test)):
        label = test_tuple[i][1]
        hidden = gru.in_hidden()

        for j in range(train_tuple[i][0].size()[0]):
            curr_feature = torch.reshape(train_tuple[i][0][j], (1, 300))
            output, hidden = gru(curr_feature, hidden)     # get ouputs using the test dataset

        max_val = output.max(dim=1)[1].numpy()          #  get the index of the max value in the output
        predss.append(max_val[0])                       # add the prediction to the total predictions li
        labels.append(label.item())                    # add the label to the total labels list

#          if max_val[0] == label.item():

accuracy = accuracy_score(predss, labels)  # calculate accuracy using sklearn
```

In [57]:
```
print('Accuracy value on the testing split for the 5b model: ', accuracy)
```

```
Accuracy value on the testing split for the 5b model:  0.33870833333333333
```

Q5.b Short answer question:

Comparing this accuracy we obtained from the GRU with the accuracy obtain by the simple RNN, we can conclude that the Gated RNN has a slightly better performance than the simple RNN. This could be due to similar issues with the RNN and the number of layers within the GRU.