Python Version 3.8

NLP HW 3

Importing all the necessary libraries

```
import pandas as pd
import numpy as np
import nltk
import re
# ! pip install bs4
from bs4 import BeautifulSoup
from nltk.tokenize import word_tokenize
import warnings
warnings.filterwarnings('ignore')
from sklearn.linear_model import Perceptron
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score, f1_score, recall_score, precision_score
from sklearn.svm import SVC
from sklearn.svm import LinearSVC as SVC
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

from numpy import argmax
from copy import deepcopy
from numpy import vstack

# ! pip install gensim
import gensim
import gensim.downloader as api

import gensim.models
from gensim import utils
```

```
Requirement already satisfied: bs4 in c:\users\palak\anaconda3\lib\site-packages (0.0.1)
Requirement already satisfied: beautifulsoup4 in c:\users\palak\anaconda3\lib\site-packages (from bs4) (4
Requirement already satisfied: soupsieve>1.2 in c:\users\palak\anaconda3\lib\site-packages (from beautifu
Requirement already satisfied: gensim in c:\users\palak\anaconda3\lib\site-packages (4.1.2)
Requirement already satisfied: smart-open>=1.8.1 in c:\users\palak\anaconda3\lib\site-packages (from gens
Requirement already satisfied: numpy>=1.17.0 in c:\users\palak\anaconda3\lib\site-packages (from gensim)
Requirement already satisfied: scipy>=0.18.1 in c:\users\palak\anaconda3\lib\site-packages (from gensim)
```

Task 1

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.

Loading the dataset in the cell below, by first reading the tsv file and then removing any row of data having inconsistent value, I also drop all the columns of the dataset except the review_body and the star_rating, since these are the only coulmns of use to us.

```
# copied_path = 'amazon_reviews_us_Beauty_v1_00.tsv.gz'
copied_path = 'data.tsv'
data = pd.read_table(copied_path, on_bad_lines='skip')
```

Converting the dataset classifications of 1,2, to 1; 3 to 2; and 4 and 5 to 3

```
data['review_body']=data['review_body'].apply(str)
dff = data.loc[:, ['star_rating','review_body']]
df1=dff.loc[data['star_rating'].isin(['1','2'])]
df1['star_rating']=1
df2=dff.loc[data['star_rating'] == '3']
df2['star_rating']=2
df3=dff.loc[data['star_rating'].isin(['4','5'])]
df3['star_rating']=3
df1=df1.sample(n=20000)
df2=df2.sample(n=20000)
df3=df3.sample(n=20000)
dfr=pd.concat([df1,df2,df3])
```

```
dfr.head(10)
```

	star_rating	review_body
1786360	1	The vial arrived empty
286464	1	I hate giving bad reviews. However, this scal...
994118	1	This foundation looked absolutely beautiful wh...
306992	1	The one I received was not the one in the phot...
280190	1	Didn't get the dvd
1533353	1	I was so excited about this bag. It arrived q...
1138979	1	Glue was already dry!
1558908	1	GREAT! My old brush attachment was getting pre...
159993	1	wasn't the same
1996361	1	I'm disappointed in Sonicare. The wand units ...

Preprocessing review body

```
from bs4 import BeautifulSoup
dfr['review_body'] = dfr['review_body'].str.lower()
dfr['review_body']=dfr['review_body'].apply(str)
 # strip html with BeautifulSoup
dfr['review_body'] = [BeautifulSoup(text).get_text() for text in dfr['review_body'] ]
# remove non alphabetic. keep spaces
dfr['review_body'] = dfr['review_body'].str.replace('[^a-zA-Z ]', '')
# strip leading and trailing spaces. strip extra white spaces
dfr['review_body'] = dfr['review_body'].str.strip()
```

TASK 2

(a) Loading the word2vec-google-news-300 pretrained model

```
import gensim.downloader as api
wordtovec= api.load('word2vec-google-news-300')
```

```
# wordtovec.save('NLP_HW3/wordtovec.kv')
```

```
# from gensim.models import KeyedVectors
# wordtovec = KeyedVectors.load('NLP_HW3/wordtovec.kv')
```

Finding the cosine similarity between 2 words on the pretrained model

In the cell below we check semantic similarities of the pretrained model on example:

1) man ~ boy

2) man ~ women

3) white ~ color

```
wordtovec.most_similar('bad')
```

```
[('good', 0.7190051078796387),
 ('terrible', 0.6828611493110657),
 ('horrible', 0.6702598333358765),
 ('Bad', 0.669891893863678),
 ('lousy', 0.6647640466690063),
 ('crummy', 0.567781925201416),
 ('horrid', 0.5651682615280151),
 ('awful', 0.5527253150939941),
 ('dreadful', 0.5526429414749146),
 ('horrendous', 0.5445998311042786)]
```

```
wordtovec.similarity("man", "boy")
```

```
0.682487
```

```
wordtovec.similarity("man", "women")
```

```
0.28830528
```

```
wordtovec.similarity("white", "color")

     0.51210797
```

```
print("For Pre-trained Model: Father - Boy + Mother = {}, with a cosine similarity of {}".format(wordtovec.most

     For Pre-trained Model: Father - Boy + Mother = husband, with a cosine similarity of 0.7670101523399353
```

(b) Generating our own enerate Word2Vec features for the dataset

```
from gensim.models import Word2Vec

# Load and tokenize data

tokenized_data = [row.split() for row in dfr.review_body]

# Train Word2Vec model

word_vec_model = Word2Vec(tokenized_data, #Word list
                          vector_size=300, #Embedding size
                          window=13, #Maximum Distance between current and predicted word
                          min_count=9, #Ignore all words with total frequency lower than this
                          workers=4  #Number of CPU Cores
                         )

# Save model
# word_vec_model.save("NLP_HW3/word_vec_model.kv")
```

```
from gensim.models import KeyedVectors
word_vec_model = KeyedVectors.load('NLP_HW3/word_vec_model.kv')
```

```
word_vec_model.wv.most_similar('bad')

     [('strong', 0.5687657594680786),
      ('good', 0.5618583559989929),
      ('terrible', 0.5501324534416199),
      ('horrible', 0.5137025713920593),
      ('weird', 0.45193347334861755),
      ('overwhelming', 0.45024770498275757),
      ('nasty', 0.44143712520599365),
      ('fake', 0.43191030621528625),
      ('awful', 0.4203457534313202),
      ('unpleasant', 0.41711878776550293)]
```

Finding cosine similairity between word on our word2vec model

In the cell below we check semantic similarities of the pretrained model on example:

1) man ~ boy

2) man ~ women

3) white ~ color

```
word_vec_model.wv.similarity("man", "boy")
```

```
     0.25545132
```

```
word_vec_model.wv.similarity("man", "women")
```

```
     0.49298048
```

```
word_vec_model.wv.similarity("white", "color")
```

```
     0.5235777
```

```
print("For Self-trained Model: Father - Boy + Mother = {}, with a cosine similarity of {}".format(word_vec_mode
```

```
     For Self-trained Model: Father - Boy + Mother = husband, with a cosine similarity of 0.6589816212654114
```

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

The pretrained model give the similarity for man and boy as 0.682487 and my model gives it as 0.25545132. The pretrained model give the similarity for man and woman as 0.28830528 and my model gives it as 0.49298048. The pretrained model give the similarity for white and color as 0.51210797 and my model gives it as 0.5235777.

It states that the pretrain model is more accurate than the model trained by me as the similaity scores in higher for the similar words in pretrain model and lower for dissimilar words than my trained model

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

The word2vec-google-news-300 Word2Vec model encode the semantic similarities between words better.

Task 3

Training simple models using Google pre-trained Word2Vec features

```
from gensim.models import KeyedVectors
wordtovec = KeyedVectors.load('NLP_HW3/wordtovec.kv')
```

```
tokenized_data = [row.split() for row in dfr.review_body]
```

Creating scentence vectors using each dataset wwords vector given by the Google pre-trained Word2Vec features

```
#Convert Review to a Word List

w2vvectors=[]                            #List to hold all word vectors in a lists of lists format
for i in range(dfr.shape[0]):
    rowvec=[]                             #List to hold all word vectors in each row
    for word in tokenized_data[i]:
        if(word in wordtovec):
            rowvec.append(wordtovec.get_vector(word))
        else:
            rowvec.append(np.zeros(300, dtype=float))     #if word is not in the google word2vec model
    w2vvectors.append(rowvec)
```

```
avgw2v=[]                             #list to hold the final vectors for each row
for row in w2vvectors:
    if(len(row)>0):
        vecavg=sum(row)/len(row)       # to calculate the average of all the vectors in each scentence
    else:
        vecavg=[0]*300
    avgw2v.append(vecavg)
```

```
len(avgw2v[0])
```

```
     300
```

Creating the test train split to train the perceptron and SVM model

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(avgw2v, dfr['star_rating'], test_size = 0.2, random_state =
y_train = y_train.astype('int')
y_test = y_test.astype('int')
```

Perceptron(word2vec)

```
from sklearn.linear_model import Perceptron
from sklearn.metrics import classification_report
p = Perceptron(n_jobs = -1, max_iter = 10000, random_state = 42)
p.fit(X_train, y_train)
print(classification_report(p.predict(X_test), y_test))
```

```
                   precision    recall  f1-score   support

                1       0.74      0.54      0.63      5365
                2       0.63      0.45      0.53      5584
                3       0.24      0.90      0.37      1051

         accuracy                           0.53     12000
        macro avg       0.54      0.63      0.51     12000
     weighted avg       0.64      0.53      0.56     12000
```

SVM(word2vec)

```
from sklearn.svm import LinearSVC
from sklearn.metrics import classification_report
# LinearSVC(max_iter=1000, random_state=42)
lsvc = LinearSVC(max_iter=1000, random_state=42)
lsvc.fit(X_train, y_train)
print(classification_report(lsvc.predict(X_test), y_test))
```

```
                   precision    recall  f1-score   support

                1       0.70      0.65      0.67      4266
                2       0.57      0.60      0.59      3828
                3       0.72      0.74      0.73      3906

         accuracy                           0.67     12000
        macro avg       0.67      0.67      0.66     12000
     weighted avg       0.67      0.67      0.67     12000
```

TF-IDF

```
from sklearn.feature_extraction.text import TfidfVectorizer
v = TfidfVectorizer()
x = v.fit_transform(dfr['review_body'])
```

```
X_train, X_test, y_train, y_test = train_test_split(x, dfr['star_rating'], test_size = 0.2, random_state = 42)
y_train = y_train.astype('int')
y_test = y_test.astype('int')
```

Perceptron(TF-IDF)

```
p = Perceptron(n_jobs = -1, max_iter = 10000, random_state = 42)
p.fit(X_train, y_train)
print(classification_report(p.predict(X_test), y_test))
```

```
              precision    recall  f1-score   support

           1       0.63      0.65      0.64      3837
           2       0.51      0.55      0.53      3695
           3       0.77      0.69      0.73      4468

    accuracy                           0.64     12000
   macro avg       0.64      0.63      0.63     12000
weighted avg       0.64      0.64      0.64     12000
```

SVM(TF-IDF)

```
from sklearn.svm import LinearSVC
# LinearSVC(max_iter=1000, random_state=42)
lsvc = LinearSVC(max_iter=1000, random_state=42)
lsvc.fit(X_train, y_train)
print(classification_report(lsvc.predict(X_test), y_test))
```

```
              precision    recall  f1-score   support

           1       0.72      0.70      0.71      4080
           2       0.59      0.62      0.61      3820
           3       0.80      0.78      0.79      4100

    accuracy                           0.70     12000
   macro avg       0.70      0.70      0.70     12000
weighted avg       0.71      0.70      0.70     12000
```

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

Tf-idf vectorization outstands the performance of Word2Vec in Simple models like Perceptron and SVM. As accuracy for Perceptron is 0.64 and 0.53 for tf-idf and word2vec respectively and accuracy for SVM is 0.70 and 0.67 for tf-idf and word2vec respectively.

TASK 4

4a

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(avgw2v, dfr['star_rating'], test_size = 0.2, random_state =
y_train = y_train.astype('int')
y_test = y_test.astype('int')
```

Converting our train and test splits to np array and then tensors

```
X_train = np.array(X_train)
X_test = np.array(X_test)
y_train = np.array(y_train)
y_test = np.array(y_test)
```

```
import torch
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.int64)
y_test = torch.tensor(y_test, dtype=torch.int64)
```

Defining a class for Multi-Layer Perceptron

```
import torch.nn as nn

class MLP(nn.Module):

    # Initialize the MLP's layers and activation functions
    def __init__(self, input_size, hidden_size1, hidden_size2, num_classes):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)    # First fully connected layer with input_size input f
        self.relu1 = nn.ReLU()                            # ReLU activation function applied to the output of f
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)   # Second fully connected layer with hidden_size1 inp
        self.relu2 = nn.ReLU()                            # ReLU activation function applied to the output of f
        self.fc3 = nn.Linear(hidden_size2, num_classes)    # Third fully connected layer with hidden_size2 inpu

    # Define the forward pass through the MLP's layers
    def forward(self, x):
        out = self.fc1(x)           # Pass the input x through the first fully connected layer
        out = self.relu1(out)       # Apply ReLU activation function to the output of fc1
        out = self.fc2(out)         # Pass the output of ReLU through the second fully connected layer
        out = self.relu2(out)       # Apply ReLU activation function to the output of fc2
        out = self.fc3(out)         # Pass the output of ReLU through the third fully connected layer
        return out                  # Return the final output of the MLP
```

```
import torch.optim as optim

# Define hyperparameters
```

```
input_size = 300            #As this the size our vector
hidden_size1 = 100
hidden_size2 = 10
output_size = 4            #As we want to classify it into 3 classes; 4 because we cant to classify it into 1,2
learning_rate = 0.001
num_epochs = 10
batch_size = 32

# Initialize the model, criterion and optimizer
model = MLP(input_size, hidden_size1, hidden_size2, output_size)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()


learning_rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Training the model

```
num_epochs = 10

for epoch in range(num_epochs):
    # Loop over batches of training data
    for i in range(0, len(X_train), batch_size):
        # Zero out the gradients
        optimizer.zero_grad()

        # Compute the forward pass
        outputs = model(X_train[i:i+batch_size])

        # Compute the loss
        loss = criterion(outputs, y_train[i:i+batch_size])

        # Compute the gradients
        loss.backward()

        # Update the parameters
        optimizer.step()
```

Testing the model and finiding out the accuracy

```
with torch.no_grad():
    outputs = model(X_test)
    _, predicted = torch.max(outputs.data, 1)
    accuracy = (predicted == y_test).sum().item() / y_test.size(0)
    print(f"Test Accuracy: {accuracy:.4f}")

     Test Accuracy: 0.6044
```

4b

To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature

```
first10w2v=[]                         #list for final vectors for each scentence having length 3000 after c
for i in range(dfr.shape[0]):
    rowvec=[]                         #list to iterate over the words in each row
    for j in range(10):
        if(j<len(tokenized_data[i])):     #if the scentence is greater than 10
            if(tokenized_data[i][j] in wordtovec):
                rowvec.extend(wordtovec.get_vector(tokenized_data[i][j]))
            else:
                rowvec.extend(np.zeros(300, dtype=float))
        else:                         # else padding is done to make it size of 10
            rowvec.extend(np.zeros(300, dtype=float))
    first10w2v.append(rowvec)
```

Spliting into train and test and then converting them into tensors

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(first10w2v, dfr['star_rating'], test_size = 0.2, random_sta
y_train = y_train.astype('int')
y_test = y_test.astype('int')
```

```
X_train = np.array(X_train)
X_test = np.array(X_test)
y_train = np.array(y_train)
y_test = np.array(y_test)
```

```
import torch
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.int64)
y_test = torch.tensor(y_test, dtype=torch.int64)
```

Define hyperparameters

```
input_size = 3000           #As the concatinated vector is of size 3000 i.e. 300*10 words
hidden_size1 = 100
hidden_size2 = 10
output_size = 4
learning_rate = 0.001
num_epochs = 10
batch_size = 32
```

```
model = MLP(input_size, hidden_size1, hidden_size2, output_size)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()
```

Train the model on train split

```
for epoch in range(num_epochs):
    # Loop over batches of training data
    for i in range(0, len(X_train), batch_size):
        # Zero out the gradients
        optimizer.zero_grad()

        # Compute the forward pass
        outputs = model(X_train[i:i+batch_size])

        # Compute the loss
        loss = criterion(outputs, y_train[i:i+batch_size])

        # Compute the gradients
        loss.backward()

        # Update the parameters
        optimizer.step()
```

Test the model and find out the accuracy

```
with torch.no_grad():
    outputs = model(X_test)
    _, predicted = torch.max(outputs.data, 1)
    accuracy = (predicted == y_test).sum().item() / y_test.size(0)
    print(f"Test Accuracy: {accuracy:.4f}")
```

```
     Test Accuracy: 0.5769
```

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

The accuracy of Simple Models are 0.53 and 0.67 for perceptron and svm respectively. Here, the MLP gives accuracy 0.6044 and 0.5769 with simple model(SVM) performing better than FNN.

Task 5

Generating word2vec vectors with the maximum limit of 20 words and padding the shorter reviews

```
tokenized_data = [row.split() for row in dfr.review_body]
```

```
first20w2v=[]
for i in range(dfr.shape[0]):
    rowvec=[]
    for j in range(20):
        if(j<len(tokenized_data[i])):
            if(tokenized_data[i][j] in wordtovec):
                rowvec.append(wordtovec.get_vector(tokenized_data[i][j]))
            else:
                rowvec.append(np.zeros(300, dtype=float))
        else:
            rowvec.append(np.zeros(300, dtype=float))
    first20w2v.append(rowvec)
```

```
avgw2v=[]
for row in first20w2v:
    if(len(row)>0):
        vecavg=sum(row)/len(row)
    else:
        vecavg=[0]*300
    avgw2v.append(vecavg)
```

spliting into train and test and converting them into tensors

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(avgw2v, dfr['star_rating'], test_size = 0.2, random_state =
y_train = y_train.astype('int')
y_test = y_test.astype('int')
```

```
X_train = np.array(X_train)
X_test = np.array(X_test)
y_train = np.array(y_train)
y_test = np.array(y_test)
```

```
import torch
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.int64)
y_test = torch.tensor(y_test, dtype=torch.int64)
```

RNN

RNN Model with hidden state size of 20

```
from torch.utils.data import Dataset,DataLoader
import torch
class RNN_Data(Dataset):
    # Constructor method, initializes X_data and Y_data
    def __init__(self, X_data, Y_data):
        self.X_data = X_data
        self.Y_data = Y_data

    # Length method, returns the length of the dataset
    def __len__(self):
        return len(self.X_data)

    # Getitem method, returns a single item from the dataset at the given index
    def __getitem__(self, index):
        # Create a zero-filled numpy array of size (20, 300) as padding
        pad = np.zeros((20, 300), dtype=float)

        # Fill the last rows of the padding array with the current X_data element
        pad[-len(self.X_data[index]):] = np.array(self.X_data[index])

        # Convert the padded array to a PyTorch FloatTensor object
        X = torch.FloatTensor(pad)

        # Convert the current Y_data element to a PyTorch tensor object
        Y = torch.tensor(self.Y_data[index])

        # Return the X and Y tensors as a tuple
        return X, Y


def my_collate(batch):
    data = [item[0] for item in batch]
    target = [item[1] for item in batch]
    return data, target
```

Training the RNN Model

```
import torch.nn as nn
import torch

# Define training parameters
input_size = 300
output_size = 4
hidden_dim = 20
n_layers = 1

#Define a RNN layer with input_size, hidden_dim, n_layers and batch_first=True
layer = nn.RNN(input_size, hidden_dim, n_layers, batch_first=True)
# Define a fully connected linear layer with input size of 400 and output size of output_size
fc = nn.Linear(400, output_size)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


# Define a function to initialize the hidden state of the RNN layer with zeros
def init_hidden(batch_size):
    hidden = torch.zeros(n_layers, batch_size, hidden_dim)
    return hidden


# Define a function to perform forward pass through the RNN and linear layers
def rnn_model(x):
    batch_size = x.size(0)
    hidden = init_hidden(batch_size)
    out, hidden = layer(x, hidden)
    out = out.contiguous().view(-1, out.shape[1] * out.shape[2])
    out = fc(out)
    return out, hidden


# Create a custom dataset object "rnn_train" using the "RNN_Data" class and pass training data
rnn_train = RNN_Data(X_train,y_train)
train_loader_mode = DataLoader(dataset=rnn_train, batch_size=8, shuffle=True, collate_fn=my_collate, drop_last=
criterion = nn.CrossEntropyLoss()
criterion = criterion.to(device)
optimizer = torch.optim.Adam(list(layer.parameters()) + list(fc.parameters()), lr=0.0001)

for ep in range(1, 5):

    # Loop through each batch in the training data
    for input_data, label in train_loader_mode:

        # Zero out the gradients in the optimizer
        optimizer.zero_grad()

        # Stack the input data and labels as PyTorch tensors and move to the device
        input_data = torch.stack(input_data)
        label = torch.stack(label)
        input_data = input_data.to(device)
        label = label.to(device)

        # Perform forward pass through the RNN and linear layers
        output, hidden = rnn_model(input_data)

        # Calculate the loss using the CrossEntropyLoss function and the predicted output and true labels
        loss = criterion(output, label)

        # Perform backpropagation to calculate gradients
        loss.backward()

        # Update the weights in the optimizer
        optimizer.step()

    # Print the epoch number and training loss
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(ep, loss.item()))
```

```
Epoch: 1        Training Loss: 1.214972
Epoch: 2        Training Loss: 0.580988
Epoch: 3        Training Loss: 0.886190
Epoch: 4        Training Loss: 0.895759
```

Testing the RNN Model

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
rnn_test = RNN_Data(X_test, y_test)
test_loader_mode = DataLoader(dataset = rnn_test, batch_size=8, collate_fn=my_collate, drop_last=True)

predictions, actual = list(), list()            #Initialize empty lists to store predicted labels and actual l
for test_data, test_label in test_loader_mode:
    test_data = torch.stack(test_data)
    test_label = torch.stack(test_label)
    pred, hid = rnn_model(test_data.to('cpu'))    #Pass the test data through the RNN model to obtain the predi
    pred = pred.to('cpu')
    pred = pred.detach().numpy()                # Detach the predicted output from the computation graph and c
    pred = argmax(pred, axis= 1)                #Get the index of the maximum value in the predicted output al
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)         #vertically stack the predicted and actual la
acc = accuracy_score(actual, predictions)
print('RNN has an Accuracy: {}'.format(acc))
```

```
RNN has an Accuracy: 0.6374166666666666
```

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

The accuracy for RNN model considering sentence vectors for first 20 words and padding the shorter sentences is 0.6374166666666666. The FNN model considering sentence vectors of whole sentence is 0.6044 and while considering combined vectors of first 10 words is 0.5769. So, the accuracy is better with RNN model than FNN model.

GRU

Training the GRU Model

Trained in a similar way as the RNN

```
import torch.nn as nn
import torch

input_size = 300
output_size = 4
hidden_dim = 20
n_layers = 1

layer = nn.GRU(input_size, hidden_dim, n_layers, batch_first=True)
fc = nn.Linear(400, output_size)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

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

def gru_model(x):
    batch_size = x.size(0)
    hidden = init_hidden(batch_size)
    out, hidden = layer(x, hidden)
    out = out.contiguous().view(-1, out.shape[1] * out.shape[2])
    out = fc(out)
    return out, hidden

rnn_train = RNN_Data(X_train,y_train)
train_loader_mode = DataLoader(dataset = rnn_train,batch_size=8, shuffle = True, collate_fn=my_collate, drop_la
criterion = nn.CrossEntropyLoss()
criterion = criterion.to(device)
optimizer = torch.optim.Adam(list(layer.parameters()) + list(fc.parameters()), lr=0.0001)

for ep in range(1, 5):
    for input_data, label in train_loader_mode:
        optimizer.zero_grad()
        input_data = torch.stack(input_data)
        label = torch.stack(label)
        output, hidden = gru_model(input_data.to(device))
        label = label.to(device)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()

    print('Epoch: {} \tTraining Loss: {:.6f}'.format(ep, loss.item()))
```

```
Epoch: 1        Training Loss: 1.057024
Epoch: 2        Training Loss: 0.512950
Epoch: 3        Training Loss: 0.748324
Epoch: 4        Training Loss: 0.776006
```

Testing the GRU Model

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
gru_test = RNN_Data(X_test, y_test)
test_loader_mode = DataLoader(dataset = gru_test, batch_size=8, collate_fn=my_collate, drop_last=True)

predictions, actual = list(), list()
for test_data, test_label in test_loader_mode:
    test_data = torch.stack(test_data)
    test_label = torch.stack(test_label)
    pred, hid = gru_model(test_data.to('cpu'))
    pred = pred.to('cpu')
    pred = pred.detach().numpy()
    pred = argmax(pred, axis= 1)
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)
acc = accuracy_score(actual, predictions)
print('GRU has an Accuracy: {}'.format(acc))
```

```
     GRU has an Accuracy: 0.63725
```

LSTM

Training the LSTM Model

Training it in a similar way as the RNN model

```
import torch.nn as nn
import torch

input_size = 300
output_size = 4
hidden_dim = 20
n_layers = 1

layer = nn.LSTM(input_size, hidden_dim, n_layers, batch_first=True)
fc = nn.Linear(400, output_size)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def init_hidden(batch_size):
    # The LSTM model has 2 hidden layers
    hidden = (torch.zeros(n_layers, batch_size, hidden_dim),torch.zeros(n_layers, batch_size, hidden_dim))
    return hidden

def lstm_model(x):
    batch_size = x.size(0)
    hidden = init_hidden(batch_size)
    out, hidden = layer(x, hidden)
    out = out.contiguous().view(-1, out.shape[1] * out.shape[2])
    out = fc(out)
    return out, hidden

rnn_train = RNN_Data(X_train,y_train)
train_loader_mode = DataLoader(dataset = rnn_train,batch_size=8, shuffle = True, collate_fn=my_collate, drop_la
criterion = nn.CrossEntropyLoss()
criterion = criterion.to(device)
optimizer = torch.optim.Adam(list(layer.parameters()) + list(fc.parameters()), lr=0.0001)

for ep in range(1, 6):
    for input_data, label in train_loader_mode:
        optimizer.zero_grad()
        input_data = torch.stack(input_data)
        label = torch.stack(label)
        output, hidden = lstm_model(input_data.to(device))
        label = label.to(device)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()

    print('Epoch: {} \tTraining Loss: {:.6f}'.format(ep, loss.item()))
```

```
Epoch: 1        Training Loss: 0.804450
Epoch: 2        Training Loss: 0.736467
Epoch: 3        Training Loss: 1.105232
Epoch: 4        Training Loss: 0.776341
Epoch: 5        Training Loss: 0.923788
```

Testing the LSTM Model

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
lstm_test = RNN_Data(X_test, y_test)
test_loader_mode = DataLoader(dataset = lstm_test, batch_size=8, collate_fn=my_collate, drop_last=True)

predictions, actual = list(), list()
for test_data, test_label in test_loader_mode:
    test_data = torch.stack(test_data)
    test_label = torch.stack(test_label)
    pred, hid = lstm_model(test_data.to('cpu'))
    pred = pred.to('cpu')
    pred = pred.detach().numpy()
    pred = argmax(pred, axis= 1)
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)
acc = accuracy_score(actual, predictions)
print('LSTM has an Accuracy: {}'.format(acc))
```

```
LSTM has an Accuracy: 0.6286666666666667
```

What do you conclude by comparing accuracy values you obtain by GRU, LSTM, and simple RNN.

The accuracy are 0.6374166666666666, 0.63725, 0.6324166666666666 for simple RNN, GRU and LSTM respectively. Over all they give the similar accuracies. But, the accuracy for RNN is just slightly higher than the other two.