Importing all the necessary libraries

In [1]:

```
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet',quiet=True)
nltk.download('averaged_perceptron_tagger')
import re
import copy
from nltk.corpus import stopwords
import contractions
from nltk import TweetTokenizer
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, precision_score
import gensim
import contractions
import torch
import torchvision
import warnings
warnings.filterwarnings('ignore')
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\patel\AppData\Roaming\nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
```

Q - 1. Dataset Generation

Generating the same 60k data used in HW1 using the original dataset

In [4]:

```
# data.tsv -> Entire Amazon Review Dataset. Should be in the current directory while runi
df = pd.read_table("data.tsv",on_bad_lines="skip")
```

In [5]:

```
dataF = df[["product_title","review_body","star_rating"]]
dataF.dropna(inplace=True)
dataF["star_rating"] = dataF["star_rating"].apply(lambda x:int(x))

dataF["rating"] = dataF["star_rating"]
dataF["rating"].mask(dataF["rating"]<=2,1,inplace=True)
dataF["rating"].mask(dataF["rating"]==3,2,inplace=True)
dataF["rating"].mask(dataF["rating"]>=4,3,inplace=True)

df_1 = dataF[dataF["rating"]==1]
df_2 = dataF[dataF["rating"]==2]
df_3 = dataF[dataF["rating"]==3]
class_1 = df_1.sample(n=20000, random_state=0)
class_2 = df_2.sample(n=20000, random_state=0)
class_3 = df_3.sample(n=20000, random_state=0)
```

In [6]:

```
# Saving the subset of 60k data for future use
final_df = pd.concat([class_1,class_2,class_3],ignore_index=True)
final_df.to_csv("final_data.csv",index=False)
```

Generating Validation dataset for Model evaluation (Validation set does not contain any data from training and testing)

In [7]:

```
df_1_val = dataF.drop(index=list(class_1.index))[dataF["rating"]==1]
class_1_val = df_1_val.sample(n=4000, random_state=0)
df_2_val = dataF.drop(index=list(class_2.index))[dataF["rating"]==2]
class_2_val = df_2_val.sample(n=4000, random_state=0)
df_3_val = dataF.drop(index=list(class_3.index))[dataF["rating"]==3]
class_3_val = df_3_val.sample(n=4000, random_state=0)
final_df_val = pd.concat([class_1_val,class_2_val,class_3_val],ignore_index=True)
```

In [8]:

```
final_df_val
```

Out[8]:

	product_title	review_body	star_rating	rating
0	Bare Escentuals Sugared Strawberry 100% Natura...	This smelled awful. When I brought it into a S...	1	1
1	Dabur Vatika Sweet Almond Shampoo 400mL	Made my hair fall out	1	1
2	Philips Norelco Rq12+ Replacement Head For Ser...	RQ12/52 head is much better than this thing	1	1
3	Philips Norelco Electric Razor Shaver, Model 6...	I find it interesting that all the reviews up ...	2	1
4	Bain-De-Terre Recovery Complex Anti-Frizz Seru...	Not the product I remembered. Sad but true. Re...	1	1
...	...	...	...	...
11995	China Glaze Nail Lacquer, Escaping Reality, 0....	I'm a huge CG fan and when I saw this color on...	4	3
11996	Jolie Mineral Sheer Tint SPF 20 Tinted Moistur...	This is one of the best products to use as a t...	5	3
11997	Flat Top Kabuki Foundation Brush By Keshima - ...	Best Kabuki brush I have ever owned! Easily wo...	5	3
11998	Home Health Hyaluronic Acid Cream	a good cream.	5	3
11999	Cirepil Pre-Depilatory Oil	Best line for waxing!	5	3

12000 rows × 4 columns

In [9]:

```
# Reading the csv file created in the above section
data = pd.read_csv("final_data.csv")
data
```

Out[9]:

	product_title	review_body	star_rating	rating
0	Sally Hansen Lavender Spa Wax Remover Kit For ...	For the love of all that is holy, DON'T buy th...	1	1
1	BDS - Lovely Fresh Big Bow Headband Makeup Fac...	it looks cute and soft but way too tight which...	2	1
2	Professional Studio Quality 12 Piece Natural C...	They don't pick up powders very well especiall...	2	1
3	Dolce & Gabbana Light Blue 3.4 Oz For Women, B...	Is not original!	1	1
4	Conair Curl Innovation Jumbo Hot Rollers with ...	Did not like these at all. They became so hot ...	1	1
...	...	...	...	...
59995	3 Piece, Genuine Czech, Etched, Crystal Glass ...	Will never use a cardboard or metal nail file ...	5	3
59996	Cosrx Bha Blackhead Power Liquid	This is hands down the best BHA product I've e...	5	3
59997	Coppertone ultraGUARD Sunscreen	I love this sunscreen because of the smell, re...	5	3
59998	Palmers Cocoa Butter With E & Alpha Beta Smoot...	I used this during warmer parts of the year an...	5	3
59999	Groom Mate Platinum XL Nose & Ear Hair Trimmer...	I bought this as a gift for my husband when he...	5	3

60000 rows × 4 columns

Using the same steps for pre-processing used in HW1

In [10]:

```
# Function for Contraction which will use fix function of contraction module to perform
def expandContractions(text):
    exp_words = []
    for w in text.split():
        exp_words.append(contractions.fix(w))
    return ' '.join(exp_words)

# Using re to get rid of html tags and hyperlinks
def preprocessText(x):
    x = re.sub(r'<.*>','',x)
    x = re.sub(r'http[s]?://\S+', '', x)
    return x

# Implementing tokenization of review data using TweetTokenizer which is very similar to
# results here.
tokenizer = TweetTokenizer(strip_handles=True, reduce_len=True)
def tokenize(x):
    return tokenizer.tokenize(x)

# Function that used WordNetLemmatizer() from nltk to lemmatize data.
lemmatizer = WordNetLemmatizer()
def lem(y):
    for i in range(len(y)):
        y[i] = lemmatizer.lemmatize(y[i])
    return y

data["preprocessed_reviews"] = data["review_body"].apply(lambda x:x.lower())
data["preprocessed_reviews"] = data["preprocessed_reviews"].apply(expandContractions)
data["preprocessed_reviews"] = data["preprocessed_reviews"].apply(preprocessText)
data["preprocessed_reviews"] = data["preprocessed_reviews"].apply(tokenize)
data["preprocessed_reviews"] = data["preprocessed_reviews"].apply(lem)
```

In [11]:

```
data
```

Out[11]:

	product_title	review_body	star_rating	rating	preprocessed_reviews
0	Sally Hansen Lavender Spa Wax Remover Kit For ...	For the love of all that is holy, DON'T buy th...	1	1	[for, the, love, of, all, that, is, holy, ,, d...
1	BDS - Lovely Fresh Big Bow Headband Makeup Fac...	it looks cute and soft but way too tight which...	2	1	[it, look, cute, and, soft, but, way, too, tig...
2	Professional Studio Quality 12 Piece Natural C...	They don't pick up powders very well especiall...	2	1	[they, do, not, pick, up, powder, very, well, ...
3	Dolce & Gabbana Light Blue 3.4 Oz For Women, B...	Is not original!	1	1	[is, not, original, !]
4	Conair Curl Innovation Jumbo Hot Rollers with ...	Did not like these at all. They became so hot ...	1	1	[did, not, like, these, at, all, ., they, beca...
...	...	...	...	...	...
59995	3 Piece, Genuine Czech, Etched, Crystal Glass ...	Will never use a cardboard or metal nail file ...	5	3	[will, never, use, a, cardboard, or, metal, na...
59996	Cosrx Bha Blackhead Power Liquid	This is hands down the best BHA product I've e...	5	3	[this, is, hand, down, the, best, bha, product...
59997	Coppertone ultraGUARD Sunscreen	I love this sunscreen because of the smell, re...	5	3	[i, love, this, sunscreen, because, of, the, s...
59998	Palmers Cocoa Butter With E & Alpha Beta Smoot...	I used this during warmer parts of the year an...	5	3	[i, used, this, during, warmer, part, of, the,...
59999	Groom Mate Platinum XL Nose & Ear Hair Trimmer...	I bought this as a gift for my husband when he...	5	3	[i, bought, this, a, a, gift, for, my, husband...

60000 rows × 5 columns

Applying same set of preprocessing to validation data

In [12]:

```
final_df_val["preprocessed_reviews"] = final_df_val["review_body"].apply(lambda x:x.lowe
final_df_val["preprocessed_reviews"] = final_df_val["preprocessed_reviews"].apply(expand
final_df_val["preprocessed_reviews"] = final_df_val["preprocessed_reviews"].apply(prepro
final_df_val["preprocessed_reviews"] = final_df_val["preprocessed_reviews"].apply(tokeni
final_df_val["preprocessed_reviews"] = final_df_val["preprocessed_reviews"].apply(lem)
```

In [ ]:

```
```

In [ ]:

```
```

Q - 2. Word Embedding

a) Loading the pretrained “word2vec-google-news-300” Word2Vec model

In [13]:

```
# Reference - 1
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

In [14]:

```
result1 = wv.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result1)
```

```
[('queen', 0.7118193507194519)]
```

In [15]:

```
wv.most_similar(positive=['hair', 'moisturizer'], negative=['skin'], topn=1)
```

Out[15]:

```
[('shampoo', 0.6024006009101868)]
```

In [16]:

```
wv.most_similar(positive=['scalp', 'shampoo'], negative=['conditioner'], topn=1)
```

Out[16]:

```
[('hair', 0.4457809627056122)]
```

In [ ]:

```
```

b) Training word2vec model on training data

In [17]:

```
model = gensim.models.Word2Vec(sentences=data["preprocessed_reviews"],min_count=9,vector
```

In [18]:

```
model.wv.most_similar(positive=['hair', 'moisturizer'], negative=['skin'], topn=1)
```

Out[18]:

```
[('conditioner', 0.7012938261032104)]
```

In [19]:

```
model.wv.most_similar(positive=['scalp', 'shampoo'], negative=['conditioner'], topn=1)
```

Out[19]:

```
[('dandruff', 0.618808388710022)]
```

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

Ans. The output generated by myself and pretrained model are quite similar for the above 2 examples. As it can be seen, "shampoo" and "conditioner" quite similar for hair. Also "hair" and "dandruff" both are associated with the scalp.

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

Ans. Both the models are working well on the above examples but in general pretrained word2vec model will be better because it is trained on more data. If we try to use the example of king, man and woman to get answer as queen, the model trained on the particular data gives "Word not found error". So Google-News-300 word2vec is better as it will handle new words in Test Data.

In [ ]:

```

```

In [ ]:

```

```

Train Data Embeddings

In [20]:

```
# Transforming the tokenized reviews of training data into its vector using word2vec-goo
remove_list = []
final_embeddings = []
mean_embeddings = []
for i in range(len(data)):
    word_embeddings = []

    for word in data["preprocessed_reviews"][i]:
        try:
            word_embeddings.append(wv[word])
        except KeyError:
            word_embeddings.append(np.zeros(300))
    final_embeddings.append(word_embeddings)
    mean_embeddings.append(np.mean(word_embeddings,axis=0))

# Pretrained_Embeddings_Reviews -> Contains vectors for all the words in the review
data["pretrained_embeddings_reviews"] = final_embeddings

# mean_embeddings -> Contains the mean of all the words for a review
data["mean_embeddings"]  = mean_embeddings
```

Validation Data Embeddings

In [21]:

```
# Transforming the tokenized reviews of validation data into its vector using word2vec-g
remove_list_val = []
final_embeddings_val = []
mean_embeddings_val = []

for i in range(len(final_df_val)):
    word_embeddings_val = []

    for word in final_df_val["preprocessed_reviews"][i]:
        try:
            word_embeddings_val.append(wv[word])
        except KeyError:
            word_embeddings_val.append(np.zeros(300))

    final_embeddings_val.append(word_embeddings_val)
    mean_embeddings_val.append(np.mean(word_embeddings_val,axis=0))

# Pretrained_Embeddings_Reviews -> Contains vectors for all the words in the review
final_df_val["pretrained_embeddings_reviews"] = final_embeddings_val

# mean_embeddings -> Contains the mean of all the words for a review
final_df_val["mean_embeddings"]  = mean_embeddings_val
```

In [22]:

```
data["mean_embeddings"]
```

Out[22]:

```
0        [0.027972835964626738, 0.03650338914659288, 0....
1        [-0.0003587676257621951, 0.03917959259777534, ...
2        [0.05609130859375, 0.03489990234375, 0.0506408...
3        [0.03960418701171875, -0.04815673828125, 0.130...
4        [0.048212076822916665, 0.030977147420247396, 0...
                              ...
59995    [0.04057728160511364, 0.044389204545454544, -0...
59996    [0.011457722981770833, 0.03865999221801758, 0....
59997    [0.014290771484375, 0.0109033203125, 0.0166210...
59998    [0.017403602600097656, 0.04505568742752075, 0....
59999    [0.057936295219089676, 0.014142741327700407, 0...
Name: mean_embeddings, Length: 60000, dtype: object
```

In [23]:

```
final_df_val["pretrained_embeddings_reviews"]
```

Out[23]:

```
0        [[0.109375, 0.140625, -0.03173828, 0.16601562,...
1        [[-0.055908203, 0.11767578, 0.2109375, 0.00836...
2        [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
3        [[-0.22558594, -0.01953125, 0.09082031, 0.2373...
4        [[0.08496094, -0.095214844, 0.119140625, 0.111...
                              ...
11995    [[-0.22558594, -0.01953125, 0.09082031, 0.2373...
11996    [[0.109375, 0.140625, -0.03173828, 0.16601562,...
11997    [[-0.12695312, 0.021972656, 0.28710938, 0.1533...
11998    [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
11999    [[-0.12695312, 0.021972656, 0.28710938, 0.1533...
Name: pretrained_embeddings_reviews, Length: 12000, dtype: object
```

In [24]:

```
# Deleting if there is any null values in the reviews of train data after text-preproces
i=0
delete = []
for rv in data['preprocessed_reviews']:
    if(len(rv) == 0):
        delete.append(i)
    i+=1
```

In [25]:

```
# Deleting if there is any null values in the reviews of validation data after text-prep
i=0
delete_val = []
for rv in final_df_val['preprocessed_reviews']:
    if(len(rv) == 0):
        delete_val.append(i)
    i+=1
```

In [26]:

```
print("Number of Null Values in Train Data : ", len(delete))
print("Number of Null values in validation data : ",len(delete_val))
```

```
Number of Null Values in Train Data :  1
Number of Null values in validation data :  0
```

In [27]:

```
data.drop(index=delete,axis=0,inplace=True)
```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

3. Simple models

In [28]:

```
# Preparing data for applying TF-IDF vectorizer
data["tfidf_review"] = data["review_body"]
data["tfidf_review"] = data["tfidf_review"].apply(lambda x:x.lower())
data["tfidf_review"] = data["tfidf_review"].apply(expandContractions)
data["tfidf_review"] = data["tfidf_review"].apply(preprocessText)
data["tfidf_review"] = data["tfidf_review"].apply(tokenize)
data["tfidf_review"] = data["tfidf_review"].apply(lem)
```

In [29]:

```
def joinList(t):
    return ' '.join(t)
data["tfidf_review"] = data["tfidf_review"].apply(joinList)
```

In [30]:

```
data["tfidf_review"]
```

Out[30]:

```
0        for the love of all that is holy , do not buy ...
1        it look cute and soft but way too tight which ...
2        they do not pick up powder very well especiall...
3                                        is not original !
4        did not like these at all . they became so hot...
                              ...
59995    will never use a cardboard or metal nail file ...
59996    this is hand down the best bha product i have ...
59997    i love this sunscreen because of the smell , r...
59998    i used this during warmer part of the year and...
59999    i bought this a a gift for my husband when he ...
Name: tfidf_review, Length: 59999, dtype: object
```

In [31]:

```
X_train_tfidf, X_test_tfidf, y_train_tfidf, y_test_tfidf = train_test_split(data["tfidf_
```

In [32]:

```
# Applying Tf-Idf vectorizer to the data
from sklearn.feature_extraction.text import TfidfVectorizer

vec = TfidfVectorizer()
text = list(X_train_tfidf.values)+list(X_test_tfidf.values)
vec.fit(text)
```

Out[32]:

```
TfidfVectorizer()
```

In [33]:

```
X_TRAIN_TFIDF = vec.transform(X_train_tfidf)
X_TEST_TFIDF = vec.transform(X_test_tfidf)
```

Perceptron Model with TF-IDF as embeddings

In [47]:

```
# Perceptron Model implemented using scikit-learn
prc_tfidf = Perceptron(random_state=5)
prc_tfidf.fit(X_TRAIN_TFIDF,y_train_tfidf)
```

Out[47]:

```
Perceptron(random_state=5)
```

In [48]:

```
y_pred_tfidf = prc_tfidf.predict(X_TEST_TFIDF)
print("Accuracy score for Percetron with TF-IDF as embeddings : ",accuracy_score(y_test_
```

```
Accuracy score for Percetron with TF-IDF as embeddings :  62.4
```

SVM Model with TF-IDF as embeddings

In [36]:

```
# SVM Model implemented using scikit-learn
svm_tfidf = LinearSVC(C=0.1)
svm_tfidf.fit(X_TRAIN_TFIDF,y_train_tfidf)
svm_pred_tfidf = svm_tfidf.predict(X_TEST_TFIDF)
print("Accuracy score for SVC with TF-IDF Embeddings : ",accuracy_score(y_test_tfidf,svm
```

```
Accuracy score for SVC with TF-IDF Embeddings :  71.26666666666667
```

In [ ]:

```

```

In [ ]:

```

```

Converting mean embeddings of reviews arrays to train Perceptron and SVM

In [37]:

```
vectors = [[] for i in range(300)]
for vec in data['mean_embeddings']:
    for i in range(300):
        vectors[i].append(vec[i])
vectors = np.array(vectors)
```

In [38]:

```
X = pd.DataFrame()
for i in range(300):
    X['vec'+str(i)] = vectors[i, :]
```

In [39]:

```
X
```

Out[39]:

	vec0	vec1	vec2	vec3	vec4	vec5	vec6	vec7	
0	0.027973	0.036503	0.031750	0.069017	-0.070432	0.001462	0.036540	-0.101999	0
1	-0.000359	0.039180	-0.000855	0.095080	-0.076810	0.001209	0.037842	-0.063374	0
2	0.056091	0.034900	0.050641	0.124902	-0.080347	0.012112	0.053739	-0.049774	0
3	0.039604	-0.048157	0.130859	0.023956	-0.070984	0.068390	0.100708	-0.015015	0
4	0.048212	0.030977	0.036350	0.083562	-0.033732	-0.022835	0.037558	-0.085736	0
...	...	...	...	...	...	...	...	...	
59994	0.040577	0.044389	-0.006126	0.055520	-0.042503	0.019842	0.050526	-0.023837	0
59995	0.011458	0.038660	0.028087	0.057663	-0.065738	0.006164	0.029206	-0.069214	0
59996	0.014291	0.010903	0.016621	0.043959	-0.059263	0.021224	0.048660	-0.075557	0
59997	0.017404	0.045056	0.018658	0.084900	-0.081627	0.025398	0.028895	-0.069254	0
59998	0.057936	0.014143	0.027200	0.070340	-0.027605	-0.012484	0.024951	-0.078717	0

59999 rows × 300 columns

In [ ]:

```

```

Splitting data into X_train, X_test, y_train and y_test using train_test_split method

In [40]:

```
X_train, X_test, y_train, y_test = train_test_split(X, data['rating'], stratify=data['ra
```

In [ ]:

```

```

Perceptron model with pretrained Word2Vec

In [41]:

```
prc = Perceptron(random_state=5)
prc.fit(X_train,y_train)
```

Out[41]:

```
Perceptron(random_state=5)
```

In [42]:

```
prc_pred = prc.predict(X_test)
```

In [43]:

```
print("Accuracy score for Perceptron model with Pretrained Word Embeddings : ",accuracy_
```

```
Accuracy score for Perceptron model with Pretrained Word Embeddings :  63.
4
```

In [44]:

```
svc = LinearSVC(C=0.1)
svc.fit(X_train,y_train)
```

Out[44]:

```
LinearSVC(C=0.1)
```

In [45]:

```
svc_pred = svc.predict(X_test)
```

In [46]:

```
svc_precision = precision_score(y_test,svc_pred,average=None)
print("Accuracy score for SVC model with Pretrained Word Embeddings : ",accuracy_score(y
```

```
Accuracy score for SVC model with Pretrained Word Embeddings :  65.9
```

In [ ]:

```

```

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

Ans. On the basis of the accuracy score, it can be concluded that models with TF-IDF embeddings are equivalent or better than those with pretrained Word2Vec. As it can be seen that accuracy for Perceptron with TFIDF is 62.4% almost comparable to 63.4% with Pretrained Embeddings. While accuracy for SVC with TFIDF is 71.2 compared to 65.9% with Pretrained Embeddings where the difference is considerable.

In [ ]:

```

```

In [ ]:

```

```

Q - 4. FeedForward Neural Networks

In [89]:

```
# I have trained all the models on GPU available in my PC.
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

In [90]:

```
device
```

Out[90]:

```
device(type='cuda', index=0)
```

4-a

In [95]:

```
# prepareData function will preprocess the data to convert it to torch loader
def prepareData(x,y, batch_size):
    data_list = []
    label_list = np.array(y)
    for i in range(len(x)):
        data_list.append((x.iloc[i,:].values, label_list[i]-1))


    loader = torch.utils.data.DataLoader(data_list, batch_size=batch_size, shuffle=True)
    return loader

train_loader = prepareData(X_train, y_train,32)
test_loader = prepareData(X_test, y_test,1)
```

In [178]:

```
# Reference - 2
# Class Net -> inherits Module class of nn and implements multilayer perceptron model wi
class Net(torch.nn.Module):
    def __init__(self,input_size,hidden_1,hidden_2):
        super(Net, self).__init__()
        self.layers = torch.nn.Sequential(
            torch.nn.Linear(input_size, hidden_1).to(device),
            # I have tried different activation functions like ReLu and tanh but LeakyRe
            torch.nn.LeakyReLU(),
            torch.nn.Linear(hidden_1, hidden_2).to(device),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(hidden_2, 3).to(device),
            # I have used dropout layer to make the model generalize well.
            torch.nn.Dropout(0.35)
        )


    def forward(self, x):
        return self.layers(x)
```

In [98]:

```
model = Net(300,100,10)
# Loading model on GPU
model = model.to(device)
```

In [99]:

```
# I have used CrossEntropy as loss function and Adam optimizer with learning rate 0.001
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [100]:

```
model
```

Out[100]:

```
Net(
  (layers): Sequential(
    (0): Linear(in_features=300, out_features=100, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Linear(in_features=100, out_features=10, bias=True)
    (3): LeakyReLU(negative_slope=0.01)
    (4): Linear(in_features=10, out_features=3, bias=True)
    (5): Dropout(p=0.35, inplace=False)
  )
)
```

In [101]:

```
# I have trained MLP for 30 epochs and at each epoch I have printed the respective loss
valid_loss_min = np.Inf
n_epochs = 30
for epoch in range(n_epochs):
    model.train()
    train_loss = 0.0

    for i, (train_data, target) in enumerate(train_loader):
        train_data = train_data.to(device)
        target = target.to(device)
        optimizer.zero_grad()
        output = model(train_data.float())
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*train_data.size(0)

    model.eval()
    train_loss = train_loss/len(train_loader.dataset)
    print("Epoch --> "+str(epoch+1)+" :", train_loss)
```

```
Epoch --> 1 : 0.917518109658745
Epoch --> 2 : 0.8568418967619288
Epoch --> 3 : 0.8420849370358872
Epoch --> 4 : 0.8321219237884553
Epoch --> 5 : 0.8258142589593908
Epoch --> 6 : 0.8245183939513953
Epoch --> 7 : 0.8155691596666409
Epoch --> 8 : 0.8090611771843419
Epoch --> 9 : 0.8011510197342767
Epoch --> 10 : 0.7961904746673637
Epoch --> 11 : 0.7954849854501904
Epoch --> 12 : 0.7881521744880481
Epoch --> 13 : 0.7880292705611469
Epoch --> 14 : 0.7800116352050085
Epoch --> 15 : 0.7784965042298122
Epoch --> 16 : 0.7753646418627045
Epoch --> 17 : 0.7703143528989038
Epoch --> 18 : 0.7676170001107158
Epoch --> 19 : 0.7621021360384305
Epoch --> 20 : 0.7581244316366121
Epoch --> 21 : 0.7588211240556136
Epoch --> 22 : 0.7521176020782037
Epoch --> 23 : 0.7471179353681444
Epoch --> 24 : 0.7442011698258092
Epoch --> 25 : 0.7404425133491332
Epoch --> 26 : 0.7418690804228261
Epoch --> 27 : 0.7356238405034776
Epoch --> 28 : 0.7313561851378418
Epoch --> 29 : 0.7265083836628479
Epoch --> 30 : 0.7276578983946833
```

In [102]:

```
# Function to measure model's accuracy on test loader or any loader.
def evalAccuracy(testModel, loader):
    testModel.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = testModel(inputs.float())
            loss = loss_fn(outputs, labels)
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
#     print("Accuracy : ", correct/total)
    return correct/total
```

In [172]:

```
print("Test Accuracy for Multilayer Perceptron for Task-A is : ",evalAccuracy(model,test
```

```
Test Accuracy for Multilayer Perceptron for Task-A is :  67.11666666666667
```

In [ ]:

In [ ]:

In [ ]:

4-b

In [105]:

```
# Extracting first 10 embedding vectors for each review if length is less than 10 words
ten_embeddings = []
for embd in data["pretrained_embeddings_reviews"]:
    temp_embd=[]
    for i in range(min(len(embd),10)):
        temp_embd.extend(embd[i])
    if len(temp_embd)<3000:
        for i in range(3000-len(temp_embd)):
            temp_embd.extend([0.0])
    ten_embeddings.append(temp_embd)
data["top_ten_vectors"] = ten_embeddings
```

In [106]:

```
data["top_ten_vectors"]
```

Out[106]:

```
0        [-0.011779785, -0.04736328, 0.044677734, 0.063...
1        [0.084472656, -0.0003528595, 0.053222656, 0.09...
2        [0.064453125, 0.036132812, 0.03857422, 0.09472...
3        [0.0070495605, -0.07324219, 0.171875, 0.022583...
4        [0.20019531, 0.15429688, 0.103027344, 0.008666...
                              ...
59995    [0.048828125, 0.16699219, 0.16894531, 0.087402...
59996    [0.109375, 0.140625, -0.03173828, 0.16601562, ...
59997    [-0.22558594, -0.01953125, 0.09082031, 0.23730...
59998    [-0.22558594, -0.01953125, 0.09082031, 0.23730...
59999    [-0.22558594, -0.01953125, 0.09082031, 0.23730...
Name: top_ten_vectors, Length: 59999, dtype: object
```

In [107]:

```
# Dividing data into training and testing data using train_test_split method
X_train_b, X_test_b, y_train_b, y_test_b = train_test_split(data["top_ten_vectors"],data
```

In [108]:

```
df_embd_ratings_train = pd.DataFrame(data={"embeddings_train":X_train_b,"Ratings_train":
df_embd_ratings_test = pd.DataFrame(data={"embeddings_test":X_test_b,"Ratings_test":y_te
```

In [109]:

```
df_embd_ratings_train.reset_index(inplace=True,drop=True)
df_embd_ratings_test.reset_index(drop=True,inplace=True)
```

In [110]:

```
df_embd_ratings_train
```

Out[110]:

	embeddings_train	Ratings_train
0	[-0.115234375, -0.15527344, 0.20019531, 0.2968...	2
1	[-0.22558594, -0.01953125, 0.09082031, 0.23730...	3
2	[0.084472656, -0.0003528595, 0.053222656, 0.09...	1
3	[-0.13964844, -0.03466797, -0.053710938, 0.179...	2
4	[-0.22558594, -0.01953125, 0.09082031, 0.23730...	1
...	...	...
47994	[-0.016235352, 0.09423828, 0.091796875, 0.1196...	3
47995	[0.08496094, -0.095214844, 0.119140625, 0.1118...	2
47996	[0.109375, 0.140625, -0.03173828, 0.16601562, ...	2
47997	[0.16894531, 0.063964844, -0.084472656, 0.1738...	2
47998	[0.123046875, 0.012817383, 0.01940918, 0.23046...	1

47999 rows × 2 columns

In [112]:

```
trainX_b = []
labelX_b = np.array(df_embd_ratings_train.iloc[:,1])
for i in range(len(X_train_b)):
    trainX_b.append((np.array(df_embd_ratings_train.iloc[i,0]),labelX_b[i]-1))

train_loader_b = torch.utils.data.DataLoader(trainX_b,batch_size=64,shuffle=True)
```

In [113]:

```
testX_b = []
label_X_test_b = np.array(df_embd_ratings_test.iloc[:,1])
for i in range(len(X_test_b)):
    testX_b.append((np.array(df_embd_ratings_test.iloc[i,0]),label_X_test_b[i]-1))

test_loader_b = torch.utils.data.DataLoader(testX_b,batch_size=1,shuffle=True)
```

In [ ]:

```

```

In [114]:

```
# Class NetB -> inherits Module class from nn and implements MLP for task 4-B
class NetB(torch.nn.Module):
    def __init__(self,input_size,hidden_1,hidden_2):
        super(NetB, self).__init__()
        self.layers = torch.nn.Sequential(
            torch.nn.Linear(input_size, hidden_1).to(device),
            torch.nn.LeakyReLU(),
            torch.nn.Dropout(0.25),
            # Here I have used BatchNorm1d which basically takes 1d input and normalizes
            torch.nn.BatchNorm1d(hidden_1),
            torch.nn.Linear(hidden_1, hidden_2).to(device),
            torch.nn.LeakyReLU(),
            torch.nn.Dropout(0.25),
            torch.nn.BatchNorm1d(hidden_2),
            torch.nn.Linear(hidden_2, 3).to(device),
            torch.nn.Dropout(0.4)
        )


    def forward(self, x):
        return self.layers(x)
```

In [115]:

```
# Creating instance of the NetB
mlp_model_b = NetB(3000,100,10)
mlp_model_b = mlp_model_b.to(device)
```

In [116]:

```
mlp_model_b
```

Out[116]:

```
NetB(
  (layers): Sequential(
    (0): Linear(in_features=3000, out_features=100, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Dropout(p=0.25, inplace=False)
    (3): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
    (4): Linear(in_features=100, out_features=10, bias=True)
    (5): LeakyReLU(negative_slope=0.01)
    (6): Dropout(p=0.25, inplace=False)
    (7): BatchNorm1d(10, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (8): Linear(in_features=10, out_features=3, bias=True)
    (9): Dropout(p=0.4, inplace=False)
  )
)
```

In [117]:

```
# I have used CrossEntropy as loss function and Adam optimizer with learning rate 0.001
loss_fn_b = torch.nn.CrossEntropyLoss()
optimizer_b = torch.optim.Adam(mlp_model_b.parameters(), lr=0.001)
```

In [118]:

```
# I have trained for longer period and found 36 to be a good value for epochs
valid_loss_min = np.Inf
n_epochs = 36
for epoch in range(n_epochs):
    mlp_model_b.train()
    train_loss_b = 0.0

    for i, (train_data, target) in enumerate(train_loader_b):
        train_data = train_data.to(device)
        target = target.to(device)
        optimizer_b.zero_grad()
        output = mlp_model_b(train_data.float())
        loss = loss_fn_b(output, target)
        loss.backward()
        optimizer_b.step()
        train_loss_b += loss.item()*train_data.size(0)

    mlp_model_b.eval()
    train_loss_b = train_loss_b/len(train_loader_b.dataset)
    print("Epoch --> "+str(epoch+1)+" :", train_loss_b)
```

```
Epoch --> 1 : 1.0375636212316492
Epoch --> 2 : 0.9675931537962722
Epoch --> 3 : 0.9442634624703113
Epoch --> 4 : 0.9301355587320752
Epoch --> 5 : 0.9162100125691323
Epoch --> 6 : 0.9010592327122589
Epoch --> 7 : 0.8850530789589847
Epoch --> 8 : 0.8775866482416246
Epoch --> 9 : 0.8579570799584801
Epoch --> 10 : 0.8478754551667129
Epoch --> 11 : 0.8295534680817495
Epoch --> 12 : 0.8226434250976069
Epoch --> 13 : 0.8085739037966181
Epoch --> 14 : 0.7995906678678423
Epoch --> 15 : 0.786379399931742
Epoch --> 16 : 0.7707403294878549
Epoch --> 17 : 0.7687466458821387
Epoch --> 18 : 0.7593901240616288
Epoch --> 19 : 0.745616045920123
Epoch --> 20 : 0.7377580535936118
Epoch --> 21 : 0.7274228677036548
Epoch --> 22 : 0.7134033683406882
Epoch --> 23 : 0.7133043493543353
Epoch --> 24 : 0.7025346198579481
Epoch --> 25 : 0.6963275897755976
Epoch --> 26 : 0.684203249346095
Epoch --> 27 : 0.6835889105794429
Epoch --> 28 : 0.6734783782676155
Epoch --> 29 : 0.6601606641350399
Epoch --> 30 : 0.6571511364871778
Epoch --> 31 : 0.6549592974205862
Epoch --> 32 : 0.6480018911804963
Epoch --> 33 : 0.6376820499716844
Epoch --> 34 : 0.6297598756209779
Epoch --> 35 : 0.6272112084812451
Epoch --> 36 : 0.6233612171619961
```

In [173]:

```
print("Accuracy for MLP for task 4-b : ",evalAccuracy(mlp_model_b,test_loader_b)*100)
```

```
Accuracy for MLP for task 4-b :  56.43333333333334
```

In [ ]:

```

```

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

Ans. The accuracies of Multilayer Perceptron for part a and b are 67.11% and 56.43% respectively. The accuracy for MLP of part-a is more than accuracy of Perceptron and SVC models with pretrained word embeddings. On the other hand, model trained as of part-b is not doing better comapred to Simple Models. The statistics lead us to model of part 4-a as best so far.

In [ ]:

```

```

Q - 5. Recurrent Neural Networks

Preparing training data and validation data for RNN, GRU and LSTM

Training Data

In [127]:

```
# Extracting embeddings for 20 words in review. If length is less than 20 than padding w
embedding_20 = []

for embd in data["pretrained_embeddings_reviews"]:
    temp_embd=[]
    for i in range(min(len(embd),20)):
        temp_embd.append(embd[i])
    if len(temp_embd)<20:
        for j in range(20-len(temp_embd)):
            temp_embd.append(np.zeros(300))
    embedding_20.append(temp_embd)
data["top_20_embeddings"] = embedding_20
```

Validation Data

In [128]:

```
# Extracting embeddings for 20 words in review. If length is less than 20 than padding w
embedding_20_val = []

for embd in final_df_val["pretrained_embeddings_reviews"]:
    temp_embd=[]
    for i in range(min(len(embd),20)):
        temp_embd.append(embd[i])
    if len(temp_embd)<20:
        for j in range(20-len(temp_embd)):
            temp_embd.append(np.zeros(300))
    embedding_20_val.append(temp_embd)
final_df_val["top_20_embeddings"] = embedding_20_val
```

In [129]:

```
final_df_val["top_20_embeddings"]
```

Out[129]:

```
0        [[0.109375, 0.140625, -0.03173828, 0.16601562,...
1        [[-0.055908203, 0.11767578, 0.2109375, 0.00836...
2        [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
3        [[-0.22558594, -0.01953125, 0.09082031, 0.2373...
4        [[0.08496094, -0.095214844, 0.119140625, 0.111...
                              ...
11995    [[-0.22558594, -0.01953125, 0.09082031, 0.2373...
11996    [[0.109375, 0.140625, -0.03173828, 0.16601562,...
11997    [[-0.12695312, 0.021972656, 0.28710938, 0.1533...
11998    [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
11999    [[-0.12695312, 0.021972656, 0.28710938, 0.1533...
Name: top_20_embeddings, Length: 12000, dtype: object
```

In [130]:

```
# splitting embeddings of first 20 words into train and test
X_train_seq, X_test_seq, y_train_seq, y_test_seq = train_test_split(data["top_20_embeddi
```

In [131]:

```
# Converted them to dataframe for faster computation when loading in torch loader
df_seq_embd_ratings_train = pd.DataFrame(data={"embeddings_train":X_train_seq,"Ratings_t
df_seq_embd_ratings_test = pd.DataFrame(data={"embeddings_test":X_test_seq,"Ratings_test
df_seq_embd_ratings_val = pd.DataFrame(data={"embeddings_test":final_df_val["top_20_embe
```

In [133]:

```
df_seq_embd_ratings_train.reset_index(inplace=True,drop=True)
df_seq_embd_ratings_test.reset_index(drop=True,inplace=True)
df_seq_embd_ratings_val.reset_index(drop=True, inplace=True)
```

In [ ]:

```
```

In [134]:

```
# Preparing Training Loader
trainX_seq = []
labelX_seq = np.array(df_seq_embd_ratings_train.iloc[:,1])
for i in range(len(X_train_seq)):
    trainX_seq.append((np.array(df_seq_embd_ratings_train.iloc[i,0]),labelX_seq[i]-1))

train_loader_seq = torch.utils.data.DataLoader(trainX_seq,batch_size=64,shuffle=True)
```

In [135]:

```
# Preparing Validation Loader
trainX_seq_val = []
labelX_seq_val = np.array(df_seq_embd_ratings_val.iloc[:,1])
for i in range(len(final_df_val)):
    trainX_seq_val.append((np.array(df_seq_embd_ratings_val.iloc[i,0]),labelX_seq_val[i]

val_loader_seq = torch.utils.data.DataLoader(trainX_seq_val,batch_size=1,shuffle=True)
```

In [136]:

```
# Preaparing Testing Loader
testX_seq = []
label_X_test_seq = np.array(df_seq_embd_ratings_test.iloc[:,1])
for i in range(len(X_test_seq)):
    testX_seq.append((np.array(df_seq_embd_ratings_test.iloc[i,0]),label_X_test_seq[i]-1

test_loader_seq = torch.utils.data.DataLoader(testX_seq,batch_size=1,shuffle=True)
```

In [152]:

```
# evalAccuracySeq() -> returns loss and accuracy of a give model on given torch loader
def evalAccuracySeq(testModel, loader):
    testModel.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = testModel(inputs.float())
            loss = loss_fn_b(outputs, labels)
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return running_loss/len(loader.dataset), correct/total
```

A) RNN

In [137]:

```
# Reference -3
# class RNN -> inherits Module class and implements RNN model with hidden state size 20
class RNN(torch.nn.Module):
    def __init__(self, input_size=300, hidden_size=20,num_layers = 1,num_classes=3):
        super(RNN, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        # Default nonlinearity is tanh but I have kept relu as it is giving better accur
        # I have also added dropout=0.5 which seems give better performance
        self.rnn = torch.nn.RNN(input_size,hidden_size,num_layers,batch_first=True,dropo
        self.fc = torch.nn.Linear(hidden_size,num_classes)

    def forward(self, x):
        # h0->hidden layer
        h0 = torch.zeros(self.num_layers,x.size(0), self.hidden_size).to(device)

        out,_ = self.rnn(x,h0)
        out = out[:,-1,:]
        out = self.fc(out)
        return out
```

In [139]:

```
rnn_model = RNN()
```

In [140]:

```
# I have used CrossEntropy as loss function and Adam optimizer with learning rate 0.001
rnn_model = rnn_model.to(device)
loss_fn_seq = torch.nn.CrossEntropyLoss()
optimizer_seq = torch.optim.Adam(rnn_model.parameters(), lr=0.001)
```

In [141]:

```
# I have trained RNN model for 50 epochs. At each epoch I am testing my model on unseen
# with lowest loss on validation data.
valid_loss_min = np.Inf
n_epochs = 50
best_rnn_model = None
best_model_epoch = None
for epoch in range(n_epochs):
    rnn_model.train()
    train_loss_seq = 0.0

    for i, (train_data, target) in enumerate(train_loader_seq):
        train_data = train_data.to(device)
        target = target.to(device)
        optimizer_seq.zero_grad()
        output = rnn_model(train_data.float())
        loss = loss_fn_seq(output, target)
        loss.backward()
        optimizer_seq.step()
        train_loss_seq += loss.item()*train_data.size(0)

    rnn_model.eval()
    val_loss, val_acc = evalAccuracySeq(rnn_model,val_loader_seq)
    print("Validation Loss : ", val_loss)
    if valid_loss_min>val_loss:
        valid_loss_min = val_loss
        best_rnn_model = copy.deepcopy(rnn_model)
        best_model_epoch = epoch+1

    train_loss_seq = train_loss_seq/len(train_loader_seq.dataset)
    print("Epoch --> "+str(epoch+1)+" :", train_loss_seq)
```

```
Validation Loss :  0.9121199746123336
Epoch --> 1 : 1.024930572573842
Validation Loss :  0.8840110482301874
Epoch --> 2 : 0.8829115798998436
Validation Loss :  0.8583522320714546
Epoch --> 3 : 0.8534464557152639
Validation Loss :  0.8514065974607365
Epoch --> 4 : 0.8348199229130743
Validation Loss :  0.8327555922352476
Epoch --> 5 : 0.8209328012515208
Validation Loss :  0.8670534083273136
Epoch --> 6 : 0.80963838869846
Validation Loss :  0.821337249206553
Epoch --> 7 : 0.801229065233336
Validation Loss :  0.8288414204498598
Epoch --> 8 : 0.7920327537588637
Validation Loss :  0.8086143044211155
Epoch --> 9 : 0.7873898357517404
Validation Loss :  0.8140081022060476
Epoch --> 10 : 0.7811669717828652
Validation Loss :  0.8011406514071908
Epoch --> 11 : 0.775950805583684
Validation Loss :  0.7997821332118425
Epoch --> 12 : 0.769126855565095
Validation Loss :  0.8073431345584169
Epoch --> 13 : 0.7669704111618967
Validation Loss :  0.8082664725364507
Epoch --> 14 : 0.7618929618259139
Validation Loss :  0.7944464061671728
Epoch --> 15 : 0.7610737944511591
Validation Loss :  0.8316101614856787
Epoch --> 16 : 0.7539335992682037
Validation Loss :  0.7949570479562014
Epoch --> 17 : 0.7527433570758678
Validation Loss :  0.8068976753890902
Epoch --> 18 : 0.7489280111715345
Validation Loss :  0.7965284801718423
Epoch --> 19 : 0.7480764484519266
Validation Loss :  0.8003228601131859
Epoch --> 20 : 0.7448852832184382
Validation Loss :  0.8059805760409703
Epoch --> 21 : 0.7419721984600519
Validation Loss :  0.7989793733621486
Epoch --> 22 : 0.7379297347473014
Validation Loss :  0.7920900129346701
Epoch --> 23 : 0.7402250002470088
Validation Loss :  0.793125555472199
Epoch --> 24 : 0.7362106114221728
Validation Loss :  0.7988260890480745
Epoch --> 25 : 0.7341260145899132
Validation Loss :  0.8022769339020266
Epoch --> 26 : 0.7317475160225404
Validation Loss :  0.8101300812755492
Epoch --> 27 : 0.7291878377296753
Validation Loss :  0.7944867914390342
Epoch --> 28 : 0.7274726466595222
Validation Loss :  0.8113429220474855
Epoch --> 29 : 0.7272760035080245
Validation Loss :  0.8184510934572393
Epoch --> 30 : 0.7234726738263156
Validation Loss :  0.7954166719597197
```

```
Epoch --> 31 : 0.7229187821107084
Validation Loss :  0.8005852148699423
Epoch --> 32 : 0.7217583358803153
Validation Loss :  0.8079685193101405
Epoch --> 33 : 0.7261075617974047
Validation Loss :  0.7961922488490404
Epoch --> 34 : 0.7246152985634666
Validation Loss :  0.8095404806322488
Epoch --> 35 : 0.7202977265672671
Validation Loss :  0.7992606390170295
Epoch --> 36 : 0.7161319970726959
Validation Loss :  0.8016431342152461
Epoch --> 37 : 0.7150224668785577
Validation Loss :  0.8023346285665465
Epoch --> 38 : 0.7162019091250115
Validation Loss :  0.8137389407481552
Epoch --> 39 : 0.714105490524398
Validation Loss :  0.8079768286606436
Epoch --> 40 : 0.7145796009157024
Validation Loss :  0.8077537073146448
Epoch --> 41 : 0.7103274406789529
Validation Loss :  0.8073979101674779
Epoch --> 42 : 0.7106081614005257
Validation Loss :  0.8074600832321147
Epoch --> 43 : 0.7102850249673991
Validation Loss :  0.8134710043006911
Epoch --> 44 : 0.7062750402973424
Validation Loss :  0.8162958212723961
Epoch --> 45 : 0.7055845024344867
Validation Loss :  0.8127965447189346
Epoch --> 46 : 0.704162996019775
Validation Loss :  0.8228490963920007
Epoch --> 47 : 0.705629743805314
Validation Loss :  0.8016590155557282
Epoch --> 48 : 0.7024262284904215
Validation Loss :  0.8154712539865537
Epoch --> 49 : 0.7013454380568178
Validation Loss :  0.8313960676351299
Epoch --> 50 : 0.70213546960234
```

In [145]:

```
# I am comparing the 2 models. One with lowest validation loss and one after training fo
rnn_accuracy = max(evalAccuracy(rnn_model,test_loader_seq),evalAccuracy(best_rnn_model,t
print("Accuracy for RNN Model is : ", rnn_accuracy*100)
```

```
Accuracy for RNN Model is :  64.075
```

In [ ]:

```

```

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

The accuracy for RNN model is 64.075% which is better than feedforward network of part 4-b but not that good compared to part 4-a. So compared to simple RNN, simple MLP is better.

In [ ]:

```
```

B) GRU

In [153]:

```
# Reference - 4
# class GRU -> inherits Module class and implements GRU model using torch.nn.GRU
class GRU(torch.nn.Module):
    def __init__(self, input_size=300, hidden_size=20,num_layers = 1,num_classes=3):
        super(GRU, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.gru = torch.nn.GRU(input_size,hidden_size,num_layers,batch_first=True)
        self.fc = torch.nn.Linear(hidden_size,num_classes,bias=False)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers,x.size(0), self.hidden_size).to(device)

        out,_ = self.gru(x,h0)
        out = out[:,-1,:]
        out = self.fc(out)
        return out
```

In [154]:

```
gru_model = GRU()
```

In [155]:

```
# I have used CrossEntropy as loss function and Adam optimizer with learning rate 0.001
gru_model = gru_model.to(device)
loss_fn_gru = torch.nn.CrossEntropyLoss()
optimizer_gru = torch.optim.Adam(gru_model.parameters(), lr=0.001)
```

In [156]:

```
# I have trained RNN model for 30 epochs. At each epoch I am testing my model on unseen
# with lowest loss on validation data.
valid_loss_min_gru = np.Inf
best_gru_model = None
n_epochs = 30
best_model_epoch_gru = None
for epoch in range(n_epochs):
    gru_model.train()
    train_loss_gru = 0.0

    for i, (train_data, target) in enumerate(train_loader_seq):
        train_data = train_data.to(device)
        target = target.to(device)
        optimizer_gru.zero_grad()
        output = gru_model(train_data.float())
        loss = loss_fn_gru(output, target)
        loss.backward()
        optimizer_gru.step()
        train_loss_gru += loss.item()*train_data.size(0)

    gru_model.eval()
    val_loss, val_acc = evalAccuracySeq(gru_model,val_loader_seq)
    print("Validation Loss : ", val_loss)
    if valid_loss_min_gru>val_loss:
        valid_loss_min_gru = val_loss
        best_gru_model = copy.deepcopy(gru_model)
        best_model_epoch_gru = epoch+1
    train_loss_gru = train_loss_gru/len(train_loader_seq.dataset)
    print("Epoch --> "+str(epoch+1)+" :", train_loss_gru)
```

```
Validation Loss :  0.8588361076399063
Epoch --> 1 : 0.9426594571202082
Validation Loss :  0.7983265329046796
Epoch --> 2 : 0.8084148433959053
Validation Loss :  0.7811873122374527
Epoch --> 3 : 0.7799193326782521
Validation Loss :  0.7651650989886063
Epoch --> 4 : 0.7641641930238975
Validation Loss :  0.7601570966724152
Epoch --> 5 : 0.7507465100230772
Validation Loss :  0.7517233541087093
Epoch --> 6 : 0.738889479853714
Validation Loss :  0.7521697708365973
Epoch --> 7 : 0.7291795091941066
Validation Loss :  0.750002233688836
Epoch --> 8 : 0.7198788620707427
Validation Loss :  0.7469934388620313
Epoch --> 9 : 0.712457402212828
Validation Loss :  0.7444149389219238
Epoch --> 10 : 0.7046318964033406
Validation Loss :  0.7473530151669013
Epoch --> 11 : 0.6976235789364459
Validation Loss :  0.743473089588418
Epoch --> 12 : 0.6908087807547666
Validation Loss :  0.7503929966408759
Epoch --> 13 : 0.6858488955026657
Validation Loss :  0.7424962635354216
Epoch --> 14 : 0.6799942090098104
Validation Loss :  0.7461857929259617
Epoch --> 15 : 0.674474870717268
Validation Loss :  0.7477122474936962
Epoch --> 16 : 0.6691091107938182
Validation Loss :  0.7472434088547598
Epoch --> 17 : 0.6656105481422668
Validation Loss :  0.7675320944672761
Epoch --> 18 : 0.6591718043011202
Validation Loss :  0.7542550562940887
Epoch --> 19 : 0.6552648637619571
Validation Loss :  0.7519973795972958
Epoch --> 20 : 0.6512970717445672
Validation Loss :  0.7594286853461333
Epoch --> 21 : 0.6463186627375702
Validation Loss :  0.7668595398638669
Epoch --> 22 : 0.6414436349998119
Validation Loss :  0.765846122892622
Epoch --> 23 : 0.6372566962294778
Validation Loss :  0.7682680008135601
Epoch --> 24 : 0.6337197695293239
Validation Loss :  0.7582267376540113
Epoch --> 25 : 0.62999451500557
Validation Loss :  0.7806177387650532
Epoch --> 26 : 0.6251856710017255
Validation Loss :  0.7720253044535057
Epoch --> 27 : 0.6223553858880423
Validation Loss :  0.7923862466261198
Epoch --> 28 : 0.6166545756496651
Validation Loss :  0.795018259244515
Epoch --> 29 : 0.6137450187165885
Validation Loss :  0.7759987655244962
Epoch --> 30 : 0.6110216700907048
```

In [170]:

```
# I am comparing the 2 models. One with lowest validation loss and one after training fo
gru_acc = max(evalAccuracy(gru_model,test_loader_seq),evalAccuracy(best_gru_model,test_l
print("Accuracy for GRU model is : ", gru_acc*100)
```

```
Accuracy for GRU model is :  66.48333333333333
```

C) LSTM

In [162]:

```
# Reference - 5
# class LSTM -> inherits Module class and implements LSTM model

class LSTM(torch.nn.Module):
    def __init__(self, input_size=300, hidden_size=20,num_layers = 1,num_classes=3):
        super(LSTM, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.lstm = torch.nn.LSTM(input_size,hidden_size,num_layers,batch_first=True)
        self.fc = torch.nn.Linear(hidden_size,num_classes)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers,x.size(0), self.hidden_size).to(device)
        c0 = torch.zeros(self.num_layers,x.size(0), self.hidden_size).to(device)

        out,_ = self.lstm(x,(h0,c0))
        out = out[:,-1,:]
        out = self.fc(out)
        return out
```

In [179]:

```
lstm_model = LSTM()
```

In [180]:

```
# I have used CrossEntropy as loss function and Adam optimizer with learning rate 0.001
lstm_model = lstm_model.to(device)
loss_fn_lstm = torch.nn.CrossEntropyLoss()
optimizer_lstm = torch.optim.Adam(lstm_model.parameters(), lr=0.001)
```

In [181]:

```
# I have trained LSTM model for 35 epochs. At each epoch I am testing my model on unseen
# with lowest loss on validation data.
valid_loss_min_lstm = np.Inf
n_epochs = 35
best_lstm_model = None
best_model_epoch_lstm = None
for epoch in range(n_epochs):
    lstm_model.train()
    train_loss_lstm = 0.0

    for i, (train_data, target) in enumerate(train_loader_seq):
        train_data = train_data.to(device)
        target = target.to(device)
        optimizer_lstm.zero_grad()
        output = lstm_model(train_data.float())
        loss = loss_fn_lstm(output, target)
        loss.backward()
        optimizer_lstm.step()
        train_loss_lstm += loss.item()*train_data.size(0)

    lstm_model.eval()
    val_loss, val_acc = evalAccuracySeq(lstm_model,val_loader_seq)
    print("Validation Loss : ", val_loss)
    if valid_loss_min_lstm>val_loss:
        valid_loss_min_lstm = val_loss
        best_lstm_model = copy.deepcopy(lstm_model)
        best_model_epoch_lstm = epoch+1

    train_loss_lstm = train_loss_lstm/len(train_loader_seq.dataset)
    print("Epoch --> "+str(epoch+1)+" :", train_loss_lstm)
```

```
Validation Loss :  0.8598460981920362
Epoch --> 1 : 0.9449802254802965
Validation Loss :  0.8134103210962688
Epoch --> 2 : 0.8296320334583842
Validation Loss :  0.8099127588672563
Epoch --> 3 : 0.7973930859119187
Validation Loss :  0.7981440318631939
Epoch --> 4 : 0.7756513978802201
Validation Loss :  0.7752794862599112
Epoch --> 5 : 0.7617303925371783
Validation Loss :  0.7785191499522577
Epoch --> 6 : 0.7474318179464029
Validation Loss :  0.7660064006700802
Epoch --> 7 : 0.7369827116642667
Validation Loss :  0.760266359740713
Epoch --> 8 : 0.7276806716121816
Validation Loss :  0.7586528738953638
Epoch --> 9 : 0.7188625254169891
Validation Loss :  0.7564961961104224
Epoch --> 10 : 0.7110380084430822
Validation Loss :  0.7495433731268083
Epoch --> 11 : 0.70138484895243
Validation Loss :  0.7568781503534798
Epoch --> 12 : 0.6943743809535778
Validation Loss :  0.7612862069418188
Epoch --> 13 : 0.6854527098762137
Validation Loss :  0.7610923639490114
Epoch --> 14 : 0.6802471411081897
Validation Loss :  0.7505471139163322
Epoch --> 15 : 0.6752571038433496
Validation Loss :  0.7613220828777024
Epoch --> 16 : 0.6684342876392423
Validation Loss :  0.7713389622014754
Epoch --> 17 : 0.6614185166474682
Validation Loss :  0.764179432997092
Epoch --> 18 : 0.6562074472979755
Validation Loss :  0.7581971031600357
Epoch --> 19 : 0.6501246542368718
Validation Loss :  0.7818551755864755
Epoch --> 20 : 0.6458683998543172
Validation Loss :  0.755870524059787
Epoch --> 21 : 0.64117485430343
Validation Loss :  0.7673226903422522
Epoch --> 22 : 0.6357150963749567
Validation Loss :  0.7745751184346057
Epoch --> 23 : 0.6298766217556304
Validation Loss :  0.7670661081820241
Epoch --> 24 : 0.6251389492007354
Validation Loss :  0.7685798803286646
Epoch --> 25 : 0.6196899007252324
Validation Loss :  0.7817130763338452
Epoch --> 26 : 0.6166199971858197
Validation Loss :  0.7814831843579789
Epoch --> 27 : 0.6103027417522219
Validation Loss :  0.7901971648456917
Epoch --> 28 : 0.6064158004644888
Validation Loss :  0.7917905605277192
Epoch --> 29 : 0.6022392654086642
Validation Loss :  0.7848745081653712
Epoch --> 30 : 0.6001430251662484
Validation Loss :  0.8072425712012992
```

```
Epoch --> 31 : 0.5918001058222158
Validation Loss :  0.8029920337052512
Epoch --> 32 : 0.5899289524449157
Validation Loss :  0.7959735731140632
Epoch --> 33 : 0.5848147302792096
Validation Loss :  0.8303118767112634
Epoch --> 34 : 0.5816757199567999
Validation Loss :  0.8014717965544357
Epoch --> 35 : 0.5767542025718037
```

In [182]:

```
# I am comparing the 2 models. One with lowest validation loss and one after training fo
lstm_acc = max(evalAccuracy(lstm_model, test_loader_seq),evalAccuracy(best_lstm_model, t
print("Accuracy for LSTM model is : ",lstm_acc*100)
```

```
Accuracy for LSTM model is :  66.5
```

In [ ]:

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

Ans. The accuracy for simple RNN, GRU and LSTM are 64.075%, 66.483% and 66.5% respectively. On the basis of the accuracy, it can be said that LSTM is quite better than RNN and almost equivalent to GRU.

In [ ]:

In [ ]:

References

[1] https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html (https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html)
[2] https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist (https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist)
[3] https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html (https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)
[4] https://pytorch.org/docs/stable/generated/torch.nn.GRU.html (https://pytorch.org/docs/stable/generated/torch.nn.GRU.html)
[5] https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html (https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html)

In [ ]: