hw3

March 1, 2023

[ ]:
```
# ! pip install contractions
# ! pip install scikit-learn
# ! pip install pandas
# ! pip install numpy
# ! pip install nltk
# ! pip install gensim
# ! pip3 install torch torchvision torchaudio --extra-index-url https://
↪download.pytorch.org/whl/cu116
# # Dataset: https://s3.amazonaws.com/amazon-reviews-pds/tsv/
# # amazon_reviews_us_Beauty_v1_00.tsv.gz
```

[ ]:
```
import pandas as pd
import numpy as np
import re
from gensim.models import Word2Vec
import gensim.downloader as api
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

import nltk
nltk.download('punkt')
nltk.download('wordnet')
from nltk.corpus import wordnet
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
c:\Users\karav\Desktop\Applied NLP\HW3\venv\lib\site-packages\tqdm\auto.py:22:
TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\karav\AppData\Roaming\nltk_data…
```

```
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\karav\AppData\Roaming\nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
```

```
[ ]: # data = pd.read_csv('amazon_reviews_us_Beauty_v1_00.tsv', sep="\t",␣
     ↪encoding='utf-8', on_bad_lines='skip')
```

```
[ ]: # data['star_rating'] = data['star_rating'].astype('int',errors='ignore')

     # df = data[['review_body', 'star_rating']]
     # df = df.dropna(thresh=2)
```

Form three classes of 20000 reviews randomly

```
[ ]: # class_a = df.loc[df['star_rating'].isin([1,2])].sample(n=20000,␣
     ↪random_state=4)
     # class_b = df.loc[df['star_rating'].isin([3])].sample(n=20000,random_state=7)
     # class_c = df.loc[df['star_rating'].isin([4,5])].sample(n=20000,␣
     ↪random_state=21)

     # df_sampled = pd.concat([class_a, class_b, class_c])
     # df_sampled['star_rating'].value_counts()
```

```
[ ]: # df_sampled['star_rating'] = df_sampled['star_rating'].replace([1,2],0)
     # df_sampled['star_rating'] = df_sampled['star_rating'].replace([3],1)
     # df_sampled['star_rating'] = df_sampled['star_rating'].replace([4,5],2)

     # df_sampled.to_csv("data.tsv", sep='\t')
```

So far, I have read the dataset from the tsv file. I converted the data type of star_rating field to int in order to enforce uniformity as there were some float and date values in them. After dropping null values, sampled 20000 reviews for each class label and merged them into a single dataframe and then stored it for easier read access.

0.1 Task 2

```
[ ]: w2v = api.load('word2vec-google-news-300')
```

```
[ ]: # Functions to clean dataset by removing html, urls, punctuation and multiple␣
     ↪spaces

     def remove_HTML(s):
         return re.sub(r'<.*?>',' ',s)

     def remove_URL(s):
         return re.sub(r'https?:\/\/.*\/\w*',' ',s)
```

```
def remove_nonalphabets(s):
    return re.sub(r'[^a-zA-Z]',' ',s)

def remove_multispace(s):
    return re.sub(r'\s+',' ',s)
```

[ ]:
```
data = pd.read_csv('data.tsv', sep='\t', usecols=['review_body',␣
↪'star_rating'], dtype={'review_body' : str, 'star_rating': int})

data['review_body'] = data['review_body'].apply(lambda x:remove_HTML(x))
data['review_body'] = data['review_body'].apply(lambda x:remove_URL(x))
data['review_body'] = data['review_body'].apply(lambda x:remove_nonalphabets(x))
data['review_body'] = data['review_body'].apply(lambda x:remove_multispace(x))
tfidfdata = data.copy(deep=True)
data['review_body'] = data['review_body'].apply(lambda x: x.split())
tfidfdata['review_body'] = tfidfdata['review_body'].apply(lambda x: x.lower().
↪split())
tfidfdata['review_body'] = tfidfdata['review_body'].apply(lambda x: ' '.join(x))

w2vdata = list(data['review_body'])
```

[ ]:
```
model = Word2Vec(w2vdata, min_count=9, vector_size=300, window=13, workers=6)
```

0.1.1 2(a)

[ ]:
```
#Testing examples on Google's word2vec

test_vec = w2v['China']-w2v['India']+w2v['Indian']
print(w2v.most_similar(positive=[test_vec],topn=3))
print(w2v.similarity('excellent','outstanding'))
print(w2v.similarity('excellent','poor'))
print(w2v.similarity('beautiful','horrible'))
print(w2v.doesnt_match(['excellent','outstanding','poor']))
print(w2v.doesnt_match(['excellent','bad','poor']))
```

```
[('Chinese', 0.8240145444869995), ('China', 0.6561532616615295), ('Indian',
0.643246054649353)]
0.55674857
0.37769592
0.38830176
poor
excellent
```

0.1.2 2(b)

[ ]:
```python
#Testing examples on trained word2vec

test_vec = model.wv['China']-model.wv['India']+model.wv['Indian']
print(model.wv.most_similar(positive=[test_vec],topn=3))
print(model.wv.similarity('excellent','outstanding'))
print(model.wv.similarity('excellent','poor'))
print(model.wv.similarity('beautiful','horrible'))
print(model.wv.doesnt_match(['excellent','outstanding','poor']))
print(model.wv.doesnt_match(['excellent','bad','poor']))
```

```
[('China', 0.9744254350662231), ('china', 0.7366096377372742), ('USA',
0.7354326248168945)]
0.73620814
0.5710197
0.40271857
poor
bad
```

[ ]:
```python
del model
```

The vectors generated by our dataset do not seem to capture the relationships between words accurately. Although it seems to place words that are relevant to our context in proximity in vector space, the relationships are not encoded properly. It is not capable of performing the vector algebra or picking odd one out as well as pretrained word2vec.

0.2 Task 3

[ ]:
```
Vectorizer = TfidfVectorizer()
tfidf = Vectorizer.fit_transform(tfidfdata['review_body'])

TF_train, TF_test, tf_train, tf_test  = train_test_split(tfidf,
                            tfidfdata['star_rating'],
                            stratify=tfidfdata['star_rating'],
                            test_size=0.2, random_state=1)

p = Perceptron(random_state=7)
p.fit(TF_train, tf_train)

print("--------------------TF-IDF Features--------------------")
print('Perceptron')
print(classification_report(tf_test, p.predict(TF_test)))

print('-------------------------------------------------------')
print('SVM')
s = LinearSVC(random_state=7, tol= 1e-5)
s.fit(TF_train, tf_train)
print(classification_report(tf_test, s.predict(TF_test)))
```

```
--------------------TF-IDF Features--------------------
Perceptron
              precision    recall  f1-score   support

           0       0.62      0.69      0.65      4000
           1       0.56      0.51      0.53      4000
           2       0.71      0.70      0.70      4000

    accuracy                           0.63     12000
   macro avg       0.63      0.63      0.63     12000
weighted avg       0.63      0.63      0.63     12000

-------------------------------------------------------
SVM
              precision    recall  f1-score   support

           0       0.71      0.71      0.71      4000
           1       0.61      0.59      0.60      4000
           2       0.76      0.79      0.77      4000

    accuracy                           0.70     12000
   macro avg       0.69      0.70      0.69     12000
weighted avg       0.69      0.70      0.69     12000
```

[ ]:
```
del tfidfdata, Vectorizer, tfidf, TF_train, tf_train, TF_test, tf_test
```

[ ]:
```
def transform_w2v(data):
    w2vfeats = []
    for sentence in data:
        word_vecs = [w2v[word] for word in sentence if word in w2v]
        if len(word_vecs):
            sent_vec = np.mean(word_vecs, axis=0)
        else:
            sent_vec = np.zeros(300)
        w2vfeats.append(sent_vec)
    return np.array(w2vfeats)

w2vfeats_np = transform_w2v(w2vdata)
w2vlabels_np = np.array(list(data['star_rating'].astype('int')))
```

I have averaged all the word vectors for each review. For reviews that have no words in the w2v vocabulary, the feature vector is a list of zeros.

[ ]:
```
X_train, X_test, y_train, y_test  = train_test_split(w2vfeats_np,
                            w2vlabels_np,
                            stratify=w2vlabels_np,
                            test_size=0.2, random_state=1)
```

[ ]:
```
print("--------------------Word2Vec Features--------------------")
print('Perceptron')
p = Perceptron(random_state=7)
p.fit(X_train, y_train)
print(classification_report(y_test, p.predict(X_test)))

print("---------------------------------------------------------")
print('SVM')
s = LinearSVC(random_state=7, tol= 1e-5)
s.fit(X_train, y_train)
print(classification_report(y_test, s.predict(X_test)))



del p,s
```

```
--------------------Word2Vec Features--------------------
Perceptron
              precision    recall  f1-score   support

           0       0.79      0.19      0.30      4000
           1       0.39      0.93      0.55      4000
           2       0.87      0.33      0.48      4000

    accuracy                           0.48     12000
   macro avg       0.68      0.48      0.44     12000
weighted avg       0.68      0.48      0.44     12000
```

```
------------------------------------------------------------
SVM
              precision    recall  f1-score   support

           0       0.65      0.68      0.67      4000
           1       0.59      0.55      0.57      4000
           2       0.71      0.72      0.72      4000

    accuracy                           0.65     12000
   macro avg       0.65      0.65      0.65     12000
weighted avg       0.65      0.65      0.65     12000
```

TF-IDF features -

Perceptron : 63% accuracy | SVM : 70% accuracy

Word2Vec features -

Perceptron : 48% accuracy | SVM - 65% accuracy

The models were able to classify better based on TF-IDF features than the word2vec features. This could be due to few reasons - TF-IDF is a statistical method that is intended to improve metrics such as precision and recall. Word2Vec is aimed at capturing semantic relationships. Google's pretrained word2vec was trained on a wide context that may not have seen all these emotions to weigh them well in vector space.

[ ]:
```python
# if torch.cuda.is_available():
#     device = torch.device("cuda")
#     print(device)
# else:
device = torch.device("cpu")
```

0.3 Task 4

[ ]:
```python
class TrainDataset(Dataset):
    def __init__(self, reviews, labels, transform=None):
        self.reviews = torch.from_numpy(reviews).float()
        self.labels = torch.from_numpy(labels).long()
        self.transform = transform

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

    def __getitem__(self, idx):
        review = self.reviews[idx]
        label = self.labels[idx]
        if self.transform:
            review = self.transform(review)
```

```
        return review, label
```

[ ]:
```
params = {
    'batch_size': 64,
    'shuffle': True,
    'num_workers': 0
}

train_data = TrainDataset(X_train, y_train)
train_loader = DataLoader(train_data, **params)

valid_data = TrainDataset(X_test, y_test)
valid_loader = DataLoader(valid_data, **params)
```

Created a custom Dataset class inheriting from torch's Dataset and used a dataloader for batching

0.3.1 4(a)

[ ]:
```
class MLP1(nn.Module):
    def __init__(self):
        super(MLP1, self).__init__()
        self.fc1 = nn.Linear(300, 100)
        self.fc2 = nn.Linear(100, 10)
        self.fc3 = nn.Linear(10, 3)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = F.log_softmax(self.fc3(x), dim=1)
        # x = self.fc3(x)
        return x

mlp1 = MLP1()
mlp1 = mlp1.to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(mlp1.parameters(), lr=0.01)
```

[ ]:
```
max_epochs = 30
valid_loss_min = np.Inf

for epoch in range(max_epochs):
    train_loss = 0
    valid_loss = 0
    mlp1.train()
```

```
    train_correct=0
    for batch, labels in train_loader:
        batch, labels = batch.to(device), labels.to(device)
        optimizer.zero_grad()
        output = mlp1(batch)
        loss = loss_fn(output, labels)
        predicted = torch.argmax(output, dim=1)
        loss.backward()
        optimizer.step()
        train_loss+= loss.item()
        train_correct+= (predicted==labels).sum().item()

    mlp1.eval()
    valid_correct=0
    with torch.no_grad():
        for batch, labels in valid_loader:
            batch, labels = batch.to(device), labels.to(device)
            output = mlp1(batch)
            loss = loss_fn(output, labels)
            predicted = torch.argmax(output, dim=1)
            valid_loss+= loss.item()
            valid_correct+=(predicted==labels).sum().item()

    train_loss = train_loss/len(train_loader)
    valid_loss = valid_loss/len(valid_loader)
    train_acc = 100 * train_correct / len(train_data)
    valid_acc = 100 * valid_correct / len(valid_data)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}␣
 ↪\tTraining Accuracy: {:.6f} \t Validation Accuracy: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss,
        train_acc,
        valid_acc
        ))

    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
 ↪'.format(
        valid_loss_min,
        valid_loss))
        torch.save(mlp1.state_dict(), 'mlp1.pt')
        valid_loss_min = valid_loss

del mlp1
```

```
Epoch: 1        Training Loss: 0.882419         Validation Loss: 0.805861
Training Accuracy: 58.970833     Validation Accuracy: 64.666667
Validation loss decreased (inf --> 0.805861).  Saving model …
Epoch: 2        Training Loss: 0.838510         Validation Loss: 0.840016
Training Accuracy: 62.397917     Validation Accuracy: 62.116667
Epoch: 3        Training Loss: 0.823994         Validation Loss: 0.791876
Training Accuracy: 62.995833     Validation Accuracy: 64.216667
Validation loss decreased (0.805861 --> 0.791876).  Saving model …
Epoch: 4        Training Loss: 0.820845         Validation Loss: 0.786919
Training Accuracy: 63.077083     Validation Accuracy: 64.991667
Validation loss decreased (0.791876 --> 0.786919).  Saving model …
Epoch: 5        Training Loss: 0.811417         Validation Loss: 0.802333
Training Accuracy: 63.672917     Validation Accuracy: 65.258333
Epoch: 6        Training Loss: 0.803838         Validation Loss: 0.788787
Training Accuracy: 64.104167     Validation Accuracy: 64.850000
Epoch: 7        Training Loss: 0.800308         Validation Loss: 0.787960
Training Accuracy: 64.539583     Validation Accuracy: 63.950000
Epoch: 8        Training Loss: 0.799988         Validation Loss: 0.773323
Training Accuracy: 64.083333     Validation Accuracy: 65.500000
Validation loss decreased (0.786919 --> 0.773323).  Saving model …
Epoch: 9        Training Loss: 0.795391         Validation Loss: 0.783199
Training Accuracy: 64.558333     Validation Accuracy: 65.475000
Epoch: 10       Training Loss: 0.793381         Validation Loss: 0.767247
Training Accuracy: 64.854167     Validation Accuracy: 65.966667
Validation loss decreased (0.773323 --> 0.767247).  Saving model …
Epoch: 11       Training Loss: 0.792889         Validation Loss: 0.769620
Training Accuracy: 64.683333     Validation Accuracy: 65.750000
Epoch: 12       Training Loss: 0.789712         Validation Loss: 0.780824
Training Accuracy: 64.731250     Validation Accuracy: 64.975000
Epoch: 13       Training Loss: 0.788710         Validation Loss: 0.782966
Training Accuracy: 64.991667     Validation Accuracy: 64.800000
Epoch: 14       Training Loss: 0.787117         Validation Loss: 0.771606
Training Accuracy: 64.793750     Validation Accuracy: 65.833333
Epoch: 15       Training Loss: 0.787902         Validation Loss: 0.774148
Training Accuracy: 65.154167     Validation Accuracy: 65.391667
Epoch: 16       Training Loss: 0.781290         Validation Loss: 0.775664
Training Accuracy: 65.266667     Validation Accuracy: 65.608333
Epoch: 17       Training Loss: 0.779198         Validation Loss: 0.769034
Training Accuracy: 65.466667     Validation Accuracy: 66.341667
Epoch: 18       Training Loss: 0.780767         Validation Loss: 0.762907
Training Accuracy: 65.554167     Validation Accuracy: 66.291667
Validation loss decreased (0.767247 --> 0.762907).  Saving model …
Epoch: 19       Training Loss: 0.780964         Validation Loss: 0.763635
Training Accuracy: 65.177083     Validation Accuracy: 65.883333
Epoch: 20       Training Loss: 0.780560         Validation Loss: 0.796080
Training Accuracy: 65.337500     Validation Accuracy: 63.791667
Epoch: 21       Training Loss: 0.776748         Validation Loss: 0.764099
Training Accuracy: 65.718750     Validation Accuracy: 66.158333
```

```
Epoch: 22       Training Loss: 0.780137         Validation Loss: 0.767298
Training Accuracy: 65.433333     Validation Accuracy: 66.300000
Epoch: 23       Training Loss: 0.771557         Validation Loss: 0.765212
Training Accuracy: 65.685417     Validation Accuracy: 65.925000
Epoch: 24       Training Loss: 0.774043         Validation Loss: 0.765254
Training Accuracy: 65.566667     Validation Accuracy: 66.341667
Epoch: 25       Training Loss: 0.772692         Validation Loss: 0.799089
Training Accuracy: 65.539583     Validation Accuracy: 65.008333
Epoch: 26       Training Loss: 0.773041         Validation Loss: 0.766370
Training Accuracy: 65.804167     Validation Accuracy: 66.200000
Epoch: 27       Training Loss: 0.770929         Validation Loss: 0.765811
Training Accuracy: 65.929167     Validation Accuracy: 66.216667
Epoch: 28       Training Loss: 0.773404         Validation Loss: 0.773933
Training Accuracy: 65.641667     Validation Accuracy: 65.400000
Epoch: 29       Training Loss: 0.773131         Validation Loss: 0.784065
Training Accuracy: 65.754167     Validation Accuracy: 65.333333
Epoch: 30       Training Loss: 0.773960         Validation Loss: 0.760797
Training Accuracy: 65.477083     Validation Accuracy: 66.108333
Validation loss decreased (0.762907 --> 0.760797).  Saving model …
```

[ ]:
```
#model.load_state_dict(torch.load('mlp1.pt')) to load best valid_loss model
```

0.3.2 4(b)

[ ]:
```
def transform_w2v_10words(data):
    w2vfeats = []
    for sentence in data:
        word_vecs = [w2v[word] for word in sentence if word in w2v]
        sent_vec = np.concatenate([word_vecs[i] if i < len(word_vecs) else np.
 ↪zeros(300) for i in range(10)], axis=0)
        w2vfeats.append(sent_vec)
    return np.array(w2vfeats)

w2vfeats_np = transform_w2v_10words(w2vdata)
```

Concatenated only first 10 words to generate a vector for reviews.

[ ]:
```
X_train, X_test, y_train, y_test  = train_test_split(w2vfeats_np,
                            w2vlabels_np,
                            stratify=w2vlabels_np,
                            test_size=0.2, random_state=1)
```

[ ]:
```
params = {
    'batch_size': 64,
    'shuffle': True,
    'num_workers': 0
}
```

```
train_data = TrainDataset(X_train, y_train)
train_loader = DataLoader(train_data, **params)

valid_data = TrainDataset(X_test, y_test)
valid_loader = DataLoader(valid_data, **params)
```

[ ]:
```
class MLP2(nn.Module):
    def __init__(self):
        super(MLP2, self).__init__()
        self.fc1 = nn.Linear(3000, 100)
        self.fc2 = nn.Linear(100, 10)
        self.fc3 = nn.Linear(10, 3)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = F.log_softmax(self.fc3(x), dim=1)
        return x

mlp2 = MLP2()
mlp2 = mlp2.to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(mlp2.parameters(), lr=0.005)
```

[ ]:
```
max_epochs = 30
valid_loss_min = np.Inf

for epoch in range(max_epochs):
    train_loss = 0
    valid_loss = 0
    mlp2.train()
    train_correct=0
    for batch, labels in train_loader:
        batch, labels = batch.to(device), labels.to(device)
        optimizer.zero_grad()
        output = mlp2(batch)
        loss = loss_fn(output, labels)
        predicted = torch.argmax(output, dim=1)
        loss.backward()
        optimizer.step()
        train_loss+= loss.item()
        train_correct+= (predicted==labels).sum().item()

    mlp2.eval()
```

```
    valid_correct=0
    with torch.no_grad():
        for batch, labels in valid_loader:
            batch, labels = batch.to(device), labels.to(device)
            output = mlp2(batch)
            loss = loss_fn(output, labels)
            predicted = torch.argmax(output, dim=1)
            valid_loss+= loss.item()
            valid_correct+=(predicted==labels).sum().item()

    train_loss = train_loss/len(train_loader)
    valid_loss = valid_loss/len(valid_loader)
    train_acc = 100 * train_correct / len(train_data)
    valid_acc = 100 * valid_correct / len(valid_data)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}␣
 ↪\tTraining Accuracy: {:.6f} \t Validation Accuracy: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss,
        train_acc,
        valid_acc
        ))

    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
 ↪'.format(
        valid_loss_min,
        valid_loss))
        torch.save(mlp2.state_dict(), 'mlp2.pt')
        valid_loss_min = valid_loss

del mlp2
```

```
Epoch: 1        Training Loss: 0.976678         Validation Loss: 0.924451
Training Accuracy: 50.704167     Validation Accuracy: 54.841667
Validation loss decreased (inf --> 0.924451).  Saving model …
Epoch: 2        Training Loss: 0.917402         Validation Loss: 0.902421
Training Accuracy: 56.052083     Validation Accuracy: 56.175000
Validation loss decreased (0.924451 --> 0.902421).  Saving model …
Epoch: 3        Training Loss: 0.884060         Validation Loss: 0.900157
Training Accuracy: 57.893750     Validation Accuracy: 56.175000
Validation loss decreased (0.902421 --> 0.900157).  Saving model …
Epoch: 4        Training Loss: 0.850590         Validation Loss: 0.909461
Training Accuracy: 59.993750     Validation Accuracy: 56.275000
Epoch: 5        Training Loss: 0.820458         Validation Loss: 0.920390
Training Accuracy: 61.864583     Validation Accuracy: 56.175000
```

```
Epoch: 6        Training Loss: 0.790396         Validation Loss: 0.934739
Training Accuracy: 63.789583     Validation Accuracy: 54.891667
Epoch: 7        Training Loss: 0.762101         Validation Loss: 0.934167
Training Accuracy: 65.118750     Validation Accuracy: 55.950000
Epoch: 8        Training Loss: 0.743555         Validation Loss: 0.948798
Training Accuracy: 66.289583     Validation Accuracy: 56.308333
Epoch: 9        Training Loss: 0.715384         Validation Loss: 0.999014
Training Accuracy: 67.735417     Validation Accuracy: 55.400000
Epoch: 10       Training Loss: 0.692863         Validation Loss: 0.995963
Training Accuracy: 69.102083     Validation Accuracy: 54.825000
Epoch: 11       Training Loss: 0.673906         Validation Loss: 0.999969
Training Accuracy: 69.691667     Validation Accuracy: 55.408333
Epoch: 12       Training Loss: 0.662720         Validation Loss: 1.002574
Training Accuracy: 70.443750     Validation Accuracy: 55.166667
Epoch: 13       Training Loss: 0.643923         Validation Loss: 1.020750
Training Accuracy: 71.410417     Validation Accuracy: 54.875000
Epoch: 14       Training Loss: 0.632360         Validation Loss: 1.061070
Training Accuracy: 72.102083     Validation Accuracy: 55.141667
Epoch: 15       Training Loss: 0.622716         Validation Loss: 1.056160
Training Accuracy: 72.402083     Validation Accuracy: 55.425000
Epoch: 16       Training Loss: 0.607511         Validation Loss: 1.085207
Training Accuracy: 73.291667     Validation Accuracy: 55.341667
Epoch: 17       Training Loss: 0.598487         Validation Loss: 1.114869
Training Accuracy: 73.583333     Validation Accuracy: 55.158333
Epoch: 18       Training Loss: 0.591536         Validation Loss: 1.111149
Training Accuracy: 73.964583     Validation Accuracy: 54.691667
Epoch: 19       Training Loss: 0.581423         Validation Loss: 1.144386
Training Accuracy: 74.447917     Validation Accuracy: 54.616667
Epoch: 20       Training Loss: 0.579383         Validation Loss: 1.146685
Training Accuracy: 74.779167     Validation Accuracy: 54.716667
Epoch: 21       Training Loss: 0.569262         Validation Loss: 1.178676
Training Accuracy: 75.316667     Validation Accuracy: 55.358333
Epoch: 22       Training Loss: 0.561153         Validation Loss: 1.165771
Training Accuracy: 75.602083     Validation Accuracy: 55.191667
Epoch: 23       Training Loss: 0.553000         Validation Loss: 1.181841
Training Accuracy: 75.793750     Validation Accuracy: 53.525000
Epoch: 24       Training Loss: 0.550625         Validation Loss: 1.216485
Training Accuracy: 76.018750     Validation Accuracy: 54.425000
Epoch: 25       Training Loss: 0.541611         Validation Loss: 1.220801
Training Accuracy: 76.566667     Validation Accuracy: 54.675000
Epoch: 26       Training Loss: 0.539132         Validation Loss: 1.136490
Training Accuracy: 76.789583     Validation Accuracy: 54.591667
Epoch: 27       Training Loss: 0.530489         Validation Loss: 1.258120
Training Accuracy: 77.212500     Validation Accuracy: 54.616667
Epoch: 28       Training Loss: 0.533744         Validation Loss: 1.198413
Training Accuracy: 76.943750     Validation Accuracy: 54.591667
Epoch: 29       Training Loss: 0.526562         Validation Loss: 1.253449
Training Accuracy: 77.431250     Validation Accuracy: 54.316667
```

```
Epoch: 30       Training Loss: 0.524036         Validation Loss: 1.316890
Training Accuracy: 77.437500     Validation Accuracy: 55.033333
```

Observed that in the case of 4(b), the training loss decreases while validation loss increases. Experimented with dropout and lower learning rates but this phenomenon seems to occur regardless. The model in 4(a) performs significantly better than a single perceptron. Adding complexities to the model with non-linearity and layers has improved the performance from 48% to 66.1%. The disparity between 4(a) and 4(b) is possibly because 4(a) tries to capture the entirety of a review while still maintaining lower dimensions whereas 4(b) concatenates the first 10 words, which may not contain all necessary information to classify, and also increases the dimensions of features.

0.4 Task 5

[ ]:
```
def transform_rnn(data):
    w2vfeats = []
    for sentence in data:
        word_vecs = [w2v[word] for word in sentence if word in w2v]
        sent_vec = [word_vecs[i] if i < len(word_vecs) else np.zeros(300) for i␣
 ↪in range(20)]
        w2vfeats.append(sent_vec)
    return np.array(w2vfeats)

w2vfeats_np = transform_rnn(w2vdata)
```

Created word2vec features for review by appending 20 word vectors individually without performing any mathematical operations. If there are less than 20 words, the feature is padded with zeros; If there are more than 20, review is truncated at 20 words. This is the input required for RNNs.

[ ]:
```
X_train, X_test, y_train, y_test  = train_test_split(w2vfeats_np,
                            w2vlabels_np,
                            stratify=w2vlabels_np,
                            test_size=0.2, random_state=1)
```

[ ]:
```
params = {
    'batch_size': 64,
    'shuffle': True,
    'num_workers': 0
}

train_data = TrainDataset(X_train, y_train)
train_loader = DataLoader(train_data, **params)

valid_data = TrainDataset(X_test, y_test)
valid_loader = DataLoader(valid_data, **params)
```

0.4.1 5(a)

[ ]:
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, 1, batch_first=True,␣
↪nonlinearity='relu')
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size = x.size(0)
        hidden = torch.zeros(1, batch_size, self.hidden_size).to(device)
        out, hidden = self.rnn(x, hidden)
        out = F.log_softmax(self.fc(out[:,-1,:]), dim=1) #Last output alone to␣
↪calculate loss
        return out

rnn = RNN(300, 20, 3)
rnn = rnn.to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(rnn.parameters(), lr=0.005)
```

[ ]:
```
max_epochs = 30
valid_loss_min = np.Inf

for epoch in range(max_epochs):
    train_loss = 0
    valid_loss = 0
    rnn.train()

    train_correct=0
    for batch, labels in train_loader:
        batch, labels = batch.to(device), labels.to(device)
        optimizer.zero_grad()
        output = rnn(batch)
        # print(output.shape, '|', labels.shape)
        loss = loss_fn(output, labels)
        loss.backward()
        optimizer.step()
        train_loss+= loss.item()
        train_correct+= (torch.argmax(output, dim=1)==labels).sum().item()

    rnn.eval()
    valid_correct=0
    with torch.no_grad():
        for batch, labels in valid_loader:
```

```
            batch, labels = batch.to(device), labels.to(device)
            output = rnn(batch)
            loss=loss_fn(output, labels)
            valid_loss+= loss.item()
            valid_correct+= (torch.argmax(output, dim=1)==labels).sum().item()

    train_loss = train_loss/len(train_loader)
    valid_loss = valid_loss/len(valid_loader)
    train_acc = 100 * train_correct / len(train_data)
    valid_acc = 100 * valid_correct / len(valid_data)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}␣
↪\tTraining Accuracy: {:.6f} \t Validation Accuracy: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss,
        train_acc,
        valid_acc
        ))

    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
↪'.format(
        valid_loss_min,
        valid_loss))
        torch.save(rnn.state_dict(), 'rnn.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.973483         Validation Loss: 0.924852
Training Accuracy: 50.408333     Validation Accuracy: 55.008333
Validation loss decreased (inf --> 0.924852).  Saving model …
Epoch: 2        Training Loss: 0.884785         Validation Loss: 0.866593
Training Accuracy: 57.091667     Validation Accuracy: 57.658333
Validation loss decreased (0.924852 --> 0.866593).  Saving model …
Epoch: 3        Training Loss: 0.859548         Validation Loss: 0.863384
Training Accuracy: 59.220833     Validation Accuracy: 58.991667
Validation loss decreased (0.866593 --> 0.863384).  Saving model …
Epoch: 4        Training Loss: 0.841908         Validation Loss: 0.880996
Training Accuracy: 60.531250     Validation Accuracy: 58.741667
Epoch: 5        Training Loss: 0.828599         Validation Loss: 0.850196
Training Accuracy: 61.337500     Validation Accuracy: 60.008333
Validation loss decreased (0.863384 --> 0.850196).  Saving model …
Epoch: 6        Training Loss: 0.822405         Validation Loss: 0.841068
Training Accuracy: 61.489583     Validation Accuracy: 60.483333
Validation loss decreased (0.850196 --> 0.841068).  Saving model …
Epoch: 7        Training Loss: 0.811936         Validation Loss: 0.874489
Training Accuracy: 62.456250     Validation Accuracy: 57.950000
```

```
Epoch: 8        Training Loss: 0.805039         Validation Loss: 0.833688
Training Accuracy: 63.006250     Validation Accuracy: 60.850000
Validation loss decreased (0.841068 --> 0.833688).  Saving model …
Epoch: 9        Training Loss: 0.802850         Validation Loss: 0.843161
Training Accuracy: 62.750000     Validation Accuracy: 60.425000
Epoch: 10       Training Loss: 0.797369         Validation Loss: 0.834855
Training Accuracy: 63.375000     Validation Accuracy: 60.158333
Epoch: 11       Training Loss: 0.795914         Validation Loss: 0.861628
Training Accuracy: 63.385417     Validation Accuracy: 60.258333
Epoch: 12       Training Loss: 0.799824         Validation Loss: 0.824695
Training Accuracy: 63.541667     Validation Accuracy: 61.400000
Validation loss decreased (0.833688 --> 0.824695).  Saving model …
Epoch: 13       Training Loss: 0.792791         Validation Loss: 0.839745
Training Accuracy: 63.675000     Validation Accuracy: 60.766667
Epoch: 14       Training Loss: 0.787382         Validation Loss: 0.826666
Training Accuracy: 64.014583     Validation Accuracy: 61.275000
Epoch: 15       Training Loss: 0.786393         Validation Loss: 0.870654
Training Accuracy: 64.162500     Validation Accuracy: 58.650000
Epoch: 16       Training Loss: 0.798419         Validation Loss: 0.863961
Training Accuracy: 63.493750     Validation Accuracy: 60.275000
Epoch: 17       Training Loss: 0.783020         Validation Loss: 0.878881
Training Accuracy: 64.277083     Validation Accuracy: 58.891667
Epoch: 18       Training Loss: 0.781670         Validation Loss: 0.831605
Training Accuracy: 64.418750     Validation Accuracy: 61.716667
Epoch: 19       Training Loss: 0.780028         Validation Loss: 0.915492
Training Accuracy: 64.408333     Validation Accuracy: 55.991667
Epoch: 20       Training Loss: 0.776635         Validation Loss: 0.826320
Training Accuracy: 64.731250     Validation Accuracy: 61.233333
Epoch: 21       Training Loss: 0.796501         Validation Loss: 0.842651
Training Accuracy: 63.597917     Validation Accuracy: 61.108333
Epoch: 22       Training Loss: 0.780859         Validation Loss: 0.835370
Training Accuracy: 64.397917     Validation Accuracy: 61.625000
Epoch: 23       Training Loss: 0.778767         Validation Loss: 0.820345
Training Accuracy: 64.720833     Validation Accuracy: 62.083333
Validation loss decreased (0.824695 --> 0.820345).  Saving model …
Epoch: 24       Training Loss: 0.779166         Validation Loss: 1.127440
Training Accuracy: 64.633333     Validation Accuracy: 36.166667
Epoch: 25       Training Loss: 0.835730         Validation Loss: 0.844096
Training Accuracy: 60.952083     Validation Accuracy: 60.116667
Epoch: 26       Training Loss: 0.786725         Validation Loss: 0.831346
Training Accuracy: 64.439583     Validation Accuracy: 61.166667
Epoch: 27       Training Loss: 0.777375         Validation Loss: 0.825478
Training Accuracy: 64.979167     Validation Accuracy: 61.791667
Epoch: 28       Training Loss: 0.840067         Validation Loss: 0.850965
Training Accuracy: 61.150000     Validation Accuracy: 60.283333
Epoch: 29       Training Loss: 0.806351         Validation Loss: 0.876543
Training Accuracy: 63.060417     Validation Accuracy: 59.841667
Epoch: 30       Training Loss: 0.787632         Validation Loss: 0.832393
```

```
Training Accuracy: 64.231250     Validation Accuracy: 61.600000
```

FFNN - 66.1% | RNN - 61.4% (Accuracy of models with best validation loss)

RNN accuracy is slightly lower as compared to FFNN. The training loss is decreasing while the validation loss is increasing. Experimented by adding dropout and reducing learning rate but RNN still seems to try to overfit on the data. The discrepancy in accuracy values could be possibly due to RNN only considering 20 words whereas the FFNN averages the entire review.

0.4.2 5(b)

[ ]:
```
class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(GRU, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers,␣
 ↪batch_first=True, dropout=0.2)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size = x.size(0)
        hidden = torch.zeros(self.num_layers, batch_size, self.hidden_size).
 ↪to(device)
        out, hidden = self.gru(x, hidden)
        out = F.log_softmax(self.fc(out[:,-1,:]), dim=1) #Taking the last␣
 ↪output alone to calculate loss
        return out

gru = GRU(300, 20, 2, 3)
gru = gru.to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(gru.parameters(), lr=0.005)
```

[ ]:
```
max_epochs = 30
valid_loss_min = np.Inf

for epoch in range(max_epochs):
    train_loss = 0
    valid_loss = 0
    gru.train()

    train_correct=0
    for batch, labels in train_loader:
        batch, labels = batch.to(device), labels.to(device)
        optimizer.zero_grad()
        output = gru(batch)
        # print(output.shape, '|', labels.shape)
```

```
        loss = loss_fn(output, labels)
        loss.backward()
        optimizer.step()
        train_loss+= loss.item()
        train_correct+= (torch.argmax(output, dim=1)==labels).sum().item()

    gru.eval()
    valid_correct=0
    with torch.no_grad():
        for batch, labels in valid_loader:
            batch, labels = batch.to(device), labels.to(device)
            output = gru(batch)
            loss=loss_fn(output, labels)
            valid_loss+= loss.item()
            valid_correct+= (torch.argmax(output, dim=1)==labels).sum().item()

    train_loss = train_loss/len(train_loader)
    valid_loss = valid_loss/len(valid_loader)
    train_acc = 100 * train_correct / len(train_data)
    valid_acc = 100 * valid_correct / len(valid_data)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} 
 ↪\tTraining Accuracy: {:.6f} \t Validation Accuracy: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss,
        train_acc,
        valid_acc
        ))

    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
 ↪'.format(
        valid_loss_min,
        valid_loss))
        torch.save(gru.state_dict(), 'gru.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.852240         Validation Loss: 0.782885
Training Accuracy: 59.668750     Validation Accuracy: 63.900000
Validation loss decreased (inf --> 0.782885).  Saving model …
Epoch: 2        Training Loss: 0.758170         Validation Loss: 0.759034
Training Accuracy: 65.779167     Validation Accuracy: 65.200000
Validation loss decreased (0.782885 --> 0.759034).  Saving model …
Epoch: 3        Training Loss: 0.726154         Validation Loss: 0.749555
Training Accuracy: 67.587500     Validation Accuracy: 66.350000
Validation loss decreased (0.759034 --> 0.749555).  Saving model …
```

```
Epoch: 4        Training Loss: 0.703309         Validation Loss: 0.748379
Training Accuracy: 68.800000     Validation Accuracy: 66.275000
Validation loss decreased (0.749555 --> 0.748379).  Saving model …
Epoch: 5        Training Loss: 0.682284         Validation Loss: 0.742501
Training Accuracy: 69.950000     Validation Accuracy: 66.800000
Validation loss decreased (0.748379 --> 0.742501).  Saving model …
Epoch: 6        Training Loss: 0.665869         Validation Loss: 0.742956
Training Accuracy: 70.700000     Validation Accuracy: 66.650000
Epoch: 7        Training Loss: 0.652505         Validation Loss: 0.772581
Training Accuracy: 71.654167     Validation Accuracy: 65.325000
Epoch: 8        Training Loss: 0.639380         Validation Loss: 0.760187
Training Accuracy: 72.245833     Validation Accuracy: 66.441667
Epoch: 9        Training Loss: 0.625784         Validation Loss: 0.774371
Training Accuracy: 72.816667     Validation Accuracy: 65.750000
Epoch: 10       Training Loss: 0.615435         Validation Loss: 0.779434
Training Accuracy: 73.287500     Validation Accuracy: 66.025000
Epoch: 11       Training Loss: 0.608425         Validation Loss: 0.765682
Training Accuracy: 73.835417     Validation Accuracy: 65.916667
Epoch: 12       Training Loss: 0.597958         Validation Loss: 0.787514
Training Accuracy: 74.368750     Validation Accuracy: 65.916667
Epoch: 13       Training Loss: 0.589329         Validation Loss: 0.792497
Training Accuracy: 74.758333     Validation Accuracy: 65.850000
Epoch: 14       Training Loss: 0.585013         Validation Loss: 0.811597
Training Accuracy: 75.120833     Validation Accuracy: 65.950000
Epoch: 15       Training Loss: 0.576118         Validation Loss: 0.823616
Training Accuracy: 75.637500     Validation Accuracy: 64.666667
Epoch: 16       Training Loss: 0.571046         Validation Loss: 0.810080
Training Accuracy: 75.852083     Validation Accuracy: 65.333333
Epoch: 17       Training Loss: 0.566572         Validation Loss: 0.841204
Training Accuracy: 75.881250     Validation Accuracy: 65.541667
Epoch: 18       Training Loss: 0.558577         Validation Loss: 0.852746
Training Accuracy: 76.381250     Validation Accuracy: 65.258333
Epoch: 19       Training Loss: 0.556633         Validation Loss: 0.848621
Training Accuracy: 76.427083     Validation Accuracy: 65.125000
Epoch: 20       Training Loss: 0.552553         Validation Loss: 0.839151
Training Accuracy: 76.752083     Validation Accuracy: 65.091667
Epoch: 21       Training Loss: 0.544731         Validation Loss: 0.857952
Training Accuracy: 77.156250     Validation Accuracy: 64.750000
Epoch: 22       Training Loss: 0.543537         Validation Loss: 0.849694
Training Accuracy: 77.129167     Validation Accuracy: 63.841667
Epoch: 23       Training Loss: 0.540733         Validation Loss: 0.864469
Training Accuracy: 77.168750     Validation Accuracy: 64.833333
Epoch: 24       Training Loss: 0.534506         Validation Loss: 0.867184
Training Accuracy: 77.456250     Validation Accuracy: 64.716667
Epoch: 25       Training Loss: 0.533053         Validation Loss: 0.864193
Training Accuracy: 77.564583     Validation Accuracy: 64.391667
Epoch: 26       Training Loss: 0.526711         Validation Loss: 0.874959
Training Accuracy: 77.825000     Validation Accuracy: 64.550000
```

```
Epoch: 27       Training Loss: 0.525434         Validation Loss: 0.901615
Training Accuracy: 78.062500     Validation Accuracy: 64.108333
Epoch: 28       Training Loss: 0.523132         Validation Loss: 0.892494
Training Accuracy: 78.016667     Validation Accuracy: 63.983333
Epoch: 29       Training Loss: 0.521680         Validation Loss: 0.907037
Training Accuracy: 78.102083     Validation Accuracy: 63.641667
Epoch: 30       Training Loss: 0.517928         Validation Loss: 0.903834
Training Accuracy: 78.370833     Validation Accuracy: 64.258333
```

GRU - 66.8% (Accuracy after 5 epochs with best validation loss)

Validation loss is increasing with more epochs while training loss is decreasing, showing signs of overfitting. At end of 30 epochs, validation accuracy is 64.25% which is still better than a simple RNN.

0.4.3 5(c)

[ ]:
```
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,␣
 ↪batch_first=True, dropout=0.2)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size = x.size(0)
        hidden = torch.zeros(self.num_layers, batch_size, self.hidden_size).
 ↪to(device)
        cell_state = torch.zeros(self.num_layers, batch_size, self.hidden_size).
 ↪to(device)
        out, (hidden, cell_state) = self.lstm(x, (hidden, cell_state))
        # print(out.shape)
        out = F.log_softmax(self.fc(out[:,-1,:]), dim=1) #Taking the last␣
 ↪output alone to calculate loss
        return out

lstm = LSTM(300, 20, 2, 3)
lstm = lstm.to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(lstm.parameters(), lr=0.001)
```

[ ]:
```
max_epochs = 30
valid_loss_min = np.Inf

for epoch in range(max_epochs):
    train_loss = 0
```

```
    valid_loss = 0
    lstm.train()

    train_correct=0
    for batch, labels in train_loader:
        batch, labels = batch.to(device), labels.to(device)
        optimizer.zero_grad()
        output = lstm(batch)
        # print(output.shape, '|', labels.shape)
        loss = loss_fn(output, labels)
        loss.backward()
        optimizer.step()
        train_loss+= loss.item()
        train_correct+= (torch.argmax(output, dim=1)==labels).sum().item()

    lstm.eval()
    valid_correct=0
    with torch.no_grad():
        for batch, labels in valid_loader:
            batch, labels = batch.to(device), labels.to(device)
            output = lstm(batch)
            loss=loss_fn(output, labels)
            valid_loss+= loss.item()
            valid_correct+= (torch.argmax(output, dim=1)==labels).sum().item()

    train_loss = train_loss/len(train_loader)
    valid_loss = valid_loss/len(valid_loader)
    train_acc = 100 * train_correct / len(train_data)
    valid_acc = 100 * valid_correct / len(valid_data)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}␣
↪\tTraining Accuracy: {:.6f} \t Validation Accuracy: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss,
        train_acc,
        valid_acc
        ))

    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
↪'.format(
        valid_loss_min,
        valid_loss))
        torch.save(lstm.state_dict(), 'lstm.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.943884         Validation Loss: 0.885581
Training Accuracy: 52.981250     Validation Accuracy: 57.966667
Validation loss decreased (inf --> 0.885581).  Saving model …
Epoch: 2        Training Loss: 0.834866         Validation Loss: 0.825576
Training Accuracy: 61.293750     Validation Accuracy: 61.766667
Validation loss decreased (0.885581 --> 0.825576).  Saving model …
Epoch: 3        Training Loss: 0.797658         Validation Loss: 0.809182
Training Accuracy: 63.581250     Validation Accuracy: 62.716667
Validation loss decreased (0.825576 --> 0.809182).  Saving model …
Epoch: 4        Training Loss: 0.777975         Validation Loss: 0.789377
Training Accuracy: 64.693750     Validation Accuracy: 63.858333
Validation loss decreased (0.809182 --> 0.789377).  Saving model …
Epoch: 5        Training Loss: 0.760450         Validation Loss: 0.779643
Training Accuracy: 65.758333     Validation Accuracy: 64.375000
Validation loss decreased (0.789377 --> 0.779643).  Saving model …
Epoch: 6        Training Loss: 0.745741         Validation Loss: 0.768820
Training Accuracy: 66.468750     Validation Accuracy: 64.991667
Validation loss decreased (0.779643 --> 0.768820).  Saving model …
Epoch: 7        Training Loss: 0.730878         Validation Loss: 0.771671
Training Accuracy: 67.454167     Validation Accuracy: 64.791667
Epoch: 8        Training Loss: 0.721521         Validation Loss: 0.767454
Training Accuracy: 67.639583     Validation Accuracy: 65.216667
Validation loss decreased (0.768820 --> 0.767454).  Saving model …
Epoch: 9        Training Loss: 0.711574         Validation Loss: 0.776195
Training Accuracy: 68.450000     Validation Accuracy: 64.541667
Epoch: 10       Training Loss: 0.701459         Validation Loss: 0.762840
Training Accuracy: 68.931250     Validation Accuracy: 65.566667
Validation loss decreased (0.767454 --> 0.762840).  Saving model …
Epoch: 11       Training Loss: 0.691796         Validation Loss: 0.765836
Training Accuracy: 69.410417     Validation Accuracy: 65.475000
Epoch: 12       Training Loss: 0.685782         Validation Loss: 0.751288
Training Accuracy: 69.700000     Validation Accuracy: 66.100000
Validation loss decreased (0.762840 --> 0.751288).  Saving model …
Epoch: 13       Training Loss: 0.677693         Validation Loss: 0.761019
Training Accuracy: 70.185417     Validation Accuracy: 65.950000
Epoch: 14       Training Loss: 0.671068         Validation Loss: 0.745408
Training Accuracy: 70.512500     Validation Accuracy: 66.375000
Validation loss decreased (0.751288 --> 0.745408).  Saving model …
Epoch: 15       Training Loss: 0.663661         Validation Loss: 0.759947
Training Accuracy: 70.770833     Validation Accuracy: 65.541667
Epoch: 16       Training Loss: 0.658640         Validation Loss: 0.755860
Training Accuracy: 70.827083     Validation Accuracy: 65.866667
Epoch: 17       Training Loss: 0.650487         Validation Loss: 0.764561
Training Accuracy: 71.500000     Validation Accuracy: 65.891667
Epoch: 18       Training Loss: 0.645009         Validation Loss: 0.761992
Training Accuracy: 71.833333     Validation Accuracy: 65.991667
Epoch: 19       Training Loss: 0.639872         Validation Loss: 0.770475
Training Accuracy: 72.125000     Validation Accuracy: 66.391667
```

```
Epoch: 20       Training Loss: 0.634310         Validation Loss: 0.763986
Training Accuracy: 72.445833     Validation Accuracy: 66.008333
Epoch: 21       Training Loss: 0.627621         Validation Loss: 0.780688
Training Accuracy: 72.775000     Validation Accuracy: 65.991667
Epoch: 22       Training Loss: 0.624477         Validation Loss: 0.776929
Training Accuracy: 72.868750     Validation Accuracy: 66.208333
Epoch: 23       Training Loss: 0.620246         Validation Loss: 0.820825
Training Accuracy: 73.133333     Validation Accuracy: 64.758333
Epoch: 24       Training Loss: 0.617020         Validation Loss: 0.778197
Training Accuracy: 73.147917     Validation Accuracy: 65.983333
Epoch: 25       Training Loss: 0.609854         Validation Loss: 0.776264
Training Accuracy: 73.545833     Validation Accuracy: 65.333333
Epoch: 26       Training Loss: 0.605385         Validation Loss: 0.793560
Training Accuracy: 73.808333     Validation Accuracy: 66.241667
Epoch: 27       Training Loss: 0.600010         Validation Loss: 0.784294
Training Accuracy: 74.104167     Validation Accuracy: 66.375000
Epoch: 28       Training Loss: 0.596917         Validation Loss: 0.782613
Training Accuracy: 74.316667     Validation Accuracy: 66.016667
Epoch: 29       Training Loss: 0.591855         Validation Loss: 0.787732
Training Accuracy: 74.533333     Validation Accuracy: 66.041667
Epoch: 30       Training Loss: 0.589975         Validation Loss: 0.788648
Training Accuracy: 74.785417     Validation Accuracy: 66.116667
```

RNN - Best model : 61.4% accuracy | GRU - Best model : 66.8% accuracy | LSTM - Best model : 66.37% accuracy

GRU offers significant improvement over RNN due to the mechanism of gates that are capable of learning which inputs are important and need to be remembered. This additional gate mechanism solves the vanishing gradient problem of RNN but is causing the GRU to overfit on training data.

LSTM is computationally slightly more expensive than GRU and offers solutions to the problem of overfitting by adding a forget gate. This results in better performance in both training and validation data.

0.5 References

1. https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html
2. https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook
3. https://pytorch.org/docs/stable/generated/torch.nn.RNN.html
4. https://pytorch.org/docs/stable/generated/torch.nn.GRU.html
5. https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html