In [30]:
```
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet')
import re
import pickle
import gensim.downloader as api
import gensim
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
import contractions
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score
torch.manual_seed(25)
```

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

Out[30]:
```
<torch._C.Generator at 0x1f4a32bd850>
```

Read Data

Reading data from data.tsv file in the current diectory, using separator as \t and skipping bad data lines

In [2]:
```
raw_dataset = pd.read_csv(
     './data.tsv',
     sep='\t',
     on_bad_lines='skip'
)
```

```
C:\Users\Abhinav Jindal\AppData\Local\Temp\ipykernel_4396\3209690956.py:1: DtypeWarning: Columns (7) have mixed type
s. Specify dtype option on import or set low_memory=False.
  raw_dataset = pd.read_csv(
```

Keep reviews and star rating

only keeping the "review body" and "star rating" columns in the read dataset

In [3]:
```
filtered_dataset = raw_dataset[['review_body', 'star_rating']]
filtered_dataset['review_body'] = filtered_dataset['review_body'].astype('str', errors='ignore')
filtered_dataset['star_rating'] = filtered_dataset['star_rating'].astype('int64', errors='ignore')
```

```
C:\Users\Abhinav Jindal\AppData\Local\Temp\ipykernel_4396\3482623344.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returnin
g-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versu
s-a-copy)
  filtered_dataset['review_body'] = filtered_dataset['review_body'].astype('str', errors='ignore')
C:\Users\Abhinav Jindal\AppData\Local\Temp\ipykernel_4396\3482623344.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returnin
g-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versu
s-a-copy)
  filtered_dataset['star_rating'] = filtered_dataset['star_rating'].astype('int64', errors='ignore')
```

We form three classes and select 20000 reviews randomly from each class.

We create 3 classes similar to HW1 where ratings <= 2 and given class 1, greater than equal to 4 are given class3 and rest are given class 2. We then sample 20000 random samples from each class to create our dataset.

In [4]:
```python
def class_assign(value):
    try:
        value = float(value)
        if value <= 2:
            return 1
        elif value >= 4:
            return 3
        else:
            return 2
    except Exception as e:
        return 4

filtered_dataset['class'] = filtered_dataset['star_rating'].map(class_assign)
df1 = filtered_dataset.loc[filtered_dataset['class'] == 1].sample(20000, random_state=30)
df2 = filtered_dataset.loc[filtered_dataset['class'] == 2].sample(20000, random_state=30)
df3 = filtered_dataset.loc[filtered_dataset['class'] == 3].sample(20000, random_state=30)
dataset = pd.concat([df1, df2, df3])
dataset = dataset.sample(frac=1).reset_index(drop=True)
```

```
C:\Users\Abhinav Jindal\AppData\Local\Temp\ipykernel_4396\808810635.py:13: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returnin
g-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versu
s-a-copy)
  filtered_dataset['class'] = filtered_dataset['star_rating'].map(class_assign)
```

Used pickle for writing and reading some objects to reduce some computational load while testing and finetuning parameters.

In [5]:
```python
def write_pickle(obj, filename):
    with open(filename, 'wb') as f:
        pickle.dump(obj, f)
        f.close()

def load_pickle(filename):
    with open(filename, 'rb') as f:
        obj = pickle.load(f)
        f.close()
        return obj
```

write and read the dataset from pickle for faster computations

In [8]:
```python
write_pickle(dataset, './dataset.pkl')
```

In [31]:
```python
dataset = load_pickle('./dataset.pkl')
dataset
```

Out[31]:

	review_body	star_rating	class
0	great product	5.0	3
1	My wife received this hair dryer as a gift a f...	2	1
2	How do you compare the hundreds of skin care p...	4	3
3	Every well groomed guy needs one of these! My ...	5	3
4	Boy is this stuff hot!	5	3
...	...	...	...
59995	I love this nail polish! All you do is put on ...	5	3
59996	I am allergic to everything; however, I can us...	5	3
59997	So far I haven't seen anything about this eye ...	3	2
59998	Not worth the money for 100% plastic. No linin...	1	1
59999	great product will order more in the future	5	3

60000 rows × 3 columns

In [32]:
```
def clean_review_body(s):
    s = s.lower()
    # remove html tags
    s = re.sub(r'<[^<]+?>', ' ', s)
    # remove urls
    s = re.sub(r'http\S+', ' ', s)
    s = re.sub(r'www\S+', ' ', s)
    # remove extra spaces
    s = re.sub(r"\s+", ' ', s)
    # fix contractions
    s = contractions.fix(s)
    # remove non word characters
    s = re.sub(r"[^a-z\s]", ' ', s)
    return s

dataset['review_body'] = dataset['review_body'].map(clean_review_body)
```

Task 2: Word Embeddings

Part (a): word2vec-google-news-300

Loaded the google news 300 word2vec model and created 2 helper functions, one to get the vec for a word and handle exceptions if any and, other to get similarity score between 2 vectors.

In [34]:
```
try:
    # reading the model from pickle if it's already ran once and saved using pickle
    google_model = load_pickle('./google_model.pkl')
except Exception as e:
    # loading the google news word2vec model  and saving it using pickle if not already present to improve computatior
    google_model = api.load('word2vec-google-news-300')
    write_pickle(google_model, './google_model.pkl')

def get_word_to_vec(word, model):
    try:
        vec = model[word]
    except KeyError:
        print("The word '{}' does not appear in this model".format(word))
        vec = None
    return vec

def get_similarity(vec1, vec2):
    return np.dot(vec1, vec2)/(np.linalg.norm(vec1)* np.linalg.norm(vec2))
```

In [29]:
```
king_vec = get_word_to_vec('king', google_model)
man_vec = get_word_to_vec('man', google_model)
woman_vec = get_word_to_vec('woman', google_model)
queen_vec = get_word_to_vec('queen', google_model)
approx_queen_vec = king_vec - man_vec + woman_vec
print("Google word2vec similarity scores")
print ("Similarity between queen and (king - man + woman):", get_similarity(queen_vec, approx_queen_vec))
print ("Similarity between jacket and coat:", google_model.similarity('jacket', 'coat'))
print ("Similarity between king and france:", google_model.similarity('king', 'france'))
```

```
Google word2vec similarity scores
Similarity between queen and (king - man + woman): 0.73005176
Similarity between jacket and coat: 0.6492949
Similarity between king and france: 0.16112953
```

Part (b): Train word2vec using own dataset

training custom word2vec model using our dataset with window size 13 and vector size 300.

In [18]:
```
## applying preprocessing provided by gensim library
review_text = dataset['review_body'].apply(gensim.utils.simple_preprocess)
model = gensim.models.Word2Vec(
    window=13,
    vector_size=300,
    min_count=9,
    workers=4,
)
model.build_vocab(review_text)
model.train(review_text, total_examples=model.corpus_count, epochs=model.epochs)

model.save("./word2vec-custom.model")
```

In [28]:
```python
model = gensim.models.Word2Vec.load("./word2vec-custom.model")
king_vec = get_word_to_vec('king', model.wv)
man_vec = get_word_to_vec('man', model.wv)
woman_vec = get_word_to_vec('woman', model.wv)
queen_vec = get_word_to_vec('queen', model.wv)
approx_queen_vec = king_vec - man_vec + woman_vec
print("Our custom word embedding model similarity scores")
print ("Similarity between queen and (king - man + woman):", get_similarity(queen_vec, approx_queen_vec))
print ("Similarity between jacket and coat:", model.wv.similarity('jacket', 'coat'))
print ("Similarity between king and france:", model.wv.similarity('king', 'france'))
```

```
Our custom word embedding model similarity scores
Similarity between queen and (king - man + woman): 0.32972342
Similarity between jacket and coat: 0.041718163
Similarity between king and france: 0.57025325
```

The embeddings generated by the pretrained google word2vec seems to be much better compared to the model trained on our own dataset. The pretrained google model encodes semantic similarities between words better. This is probably because of the difference in the amount of data used to train these models. Google model is trained on a large dataset while our model just uses 60k reviews.

From our examples we see, it assigns higher score of 0.73 to similarity between woman and (king-man+woman) compared to 0.33 in our custom dataset. Also, jacket and coat are given a similarity score of 0.65 in google model whereas our model gives it a small score of 0.04 even though jacket and coat are highly similar and are often used in the same context of clothes. Similarly for very dissimilar words like king and france, google model gives a lower score of 0.16 and our model gives 0.57 which is not appropriate as these 2 words are not that highly related and are rarely used in the same context. Hence, google pretrained model returns much better encoding compared to our model. This could be primarily because google model is trained on a very large dataset compared to our model which is just trained on 60k reviews which is not a lot of data for a good word2vec model.

Task 3

we use gensims simple preprocess to do further minor preprocessing already provided by gensim to improve our predictions

In [35]:
```python
dataset['review_preprocessed_tokens'] = dataset['review_body'].apply(gensim.utils.simple_preprocess)
```

we calculate the mean vectors from the embeddings by google model, if none of the words are present in the google model we return a vector of zeros of size 300

we then split the dataset and train models on our training dataset and test on testing dataset

In [51]:
```python
def mean_vector(review_preprocessed_tokens):
    vectors = [google_model[word] for word in review_preprocessed_tokens if word in google_model]
    if len(vectors) > 0:
        feature_vec = np.mean(vectors, axis=0)
    else:
        feature_vec = np.zeros(300)
    return feature_vec

dataset['reviews_vector'] = dataset['review_preprocessed_tokens'].map(mean_vector)
finished_dataset = dataset[['reviews_vector', 'class']]

training_data, testing_data = train_test_split(finished_dataset, test_size=0.2, random_state=25)

train_X = np.stack(training_data['reviews_vector'])
train_Y = np.array(training_data['class'])
test_X = np.stack(testing_data['reviews_vector'])
test_Y =np.array(testing_data['class'])
```

In [38]:
```python
clf = Perceptron(penalty='elasticnet', alpha=0.00001, tol=1e-7, random_state=10)
clf.fit(train_X, train_Y)
print ("Accuracy for word2vec perceptron:", accuracy_score(test_Y, clf.predict(test_X)))
```

```
Accuracy for word2vec perceptron: 0.59025
```

Accuracy using tf-idf for perceptron (calculated from HW1): 0.6474166666666666

In [39]:
```python
clf = LinearSVC(penalty='l2', loss='squared_hinge', tol=1e-7, dual=True, random_state=25)
clf.fit(train_X, train_Y)
print ("Accuracy for word2vec SVM:", accuracy_score(test_Y, clf.predict(test_X)))
```

```
Accuracy for word2vec SVM: 0.6615833333333333
```

Accuracy using tf-idf for SVM (calculated from HW1): 0.7070833333333333

We see that tf-idf performs better for these 2 models, perceptron and SVM when compared with word2vec. This could be because word2vec vectors have complicated relations which are not easily captured by these simple models. Also, taking mean of the word2vec vectors might lead to loss of

information which could be another cause of lesser accuracy

Task 4

In [40]:
```
## change to cuda if want to use GPU
device = "cpu"
```

In [41]:
```
## move data to tensors and appropriate device and also change label to torch conventions
def format_data_for_model(train_X, train_Y, test_X, test_Y, device):
    X_train = torch.tensor(train_X).to(torch.float)
    X_test = torch.tensor(test_X).to(torch.float)
    Y_train = torch.tensor(train_Y).long()
    Y_test = torch.tensor(test_Y).long()
    X_train = X_train.to(device)
    Y_train = Y_train.to(device)
    X_test = X_test.to(device)
    Y_test = Y_test.to(device)
    # move labels from 1,2,3 to 0,1,2 to fit to pytorch conventions
    Y_train = Y_train - 1
    Y_test = Y_test - 1
    return X_train, X_test, Y_train, Y_test
```

In [42]:
```
## trains model using parameters passed and prints train and test loss and accuracy after each epoch
def train_model(model, optimizer, loss_fn, X_train, Y_train, X_test, Y_test, num_epochs, batch_size):
    for epoch in range(num_epochs):
        model.train()

        # Shuffle the training data
        indices = torch.randperm(X_train.shape[0])
        x_train = X_train[indices]
        y_train = Y_train[indices]

        # batch for training data
        for i in range(0, x_train.shape[0], batch_size):
            X_batch = x_train[i:i+batch_size]
            y_batch = y_train[i:i+batch_size]

            # Compute the forward pass through the network
            y_pred = model(X_batch)

            ## zero out the gradient, calcualate loss and back propagate loss
            optimizer.zero_grad()
            loss = loss_fn(y_pred, y_batch)
            loss.backward()

            # Update the model parameters
            optimizer.step()

        # Evaluate performance after each epoch
        with torch.no_grad():
            model.eval()

            # train loss and accuracy
            y_pred = model(X_train)
            train_loss = loss_fn(y_pred, Y_train)
            y_pred = torch.argmax(y_pred, dim=1)
            train_accuracy = (y_pred == Y_train).float().mean()

            # test loss and accuracy
            y_pred = model(X_test)
            test_loss = loss_fn(y_pred, Y_test)
            y_pred = torch.argmax(y_pred, dim=1)
            test_accuracy = (y_pred == Y_test).float().mean()

        # Print the epoch, loss, and accuracy
        print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.4f}")
        print(f"Epoch {epoch+1}/{num_epochs}, Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}")
    return model
```

Task 4(a) MLP using average Word2Vec vectors

We want to use the same mean vectors data we used for simple models, hence no separate preprocessing

In [43]:
```
X_train, X_test, Y_train, Y_test = format_data_for_model(train_X, train_Y, test_X, test_Y, device)
```

We create a MLP model class with 2 hidden layers (100 and 10 nodes each) and use Relu as the activation function. We also use SGD optimizer and CrossEntropyLoss as this was giving the best accuracy.

In [50]:
```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(300, 100)
        self.fc2 = nn.Linear(100, 10)
        self.fc3 = nn.Linear(10, 3)
        self.relu = nn.ReLU()

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

# Create the MLP model and optimizer
model = MLP()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# As the loss is CrossEntropy we don't need to add softmax separately
loss_fn = nn.CrossEntropyLoss()
model.to(device)
```

Out[50]:
```
MLP(
  (fc1): Linear(in_features=300, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (relu): ReLU()
)
```

I have used the following hyperparameters:
learning rate = 0.01
loss function = CrossEntropyLoss
optimizer = SGD
momentum = 0.9
non-linear activation = Relu
epochs = 50
batch size = 32

In [45]:
```
## trains the model and prints accuracy after each epoch
model = train_model(model, optimizer, loss_fn, X_train, Y_train, X_test, Y_test, num_epochs=50, batch_size=32)
```

```
Epoch 1/50, Train Loss: 0.8410, Train Accuracy: 0.6064
Epoch 1/50, Test Loss: 0.8459, Test Accuracy: 0.6040
Epoch 2/50, Train Loss: 0.7828, Train Accuracy: 0.6531
Epoch 2/50, Test Loss: 0.7882, Test Accuracy: 0.6518
Epoch 3/50, Train Loss: 0.7750, Train Accuracy: 0.6447
Epoch 3/50, Test Loss: 0.7869, Test Accuracy: 0.6363
Epoch 4/50, Train Loss: 0.7711, Train Accuracy: 0.6522
Epoch 4/50, Test Loss: 0.7830, Test Accuracy: 0.6495
Epoch 5/50, Train Loss: 0.7504, Train Accuracy: 0.6685
Epoch 5/50, Test Loss: 0.7641, Test Accuracy: 0.6602
Epoch 6/50, Train Loss: 0.7992, Train Accuracy: 0.6382
Epoch 6/50, Test Loss: 0.8207, Test Accuracy: 0.6277
Epoch 7/50, Train Loss: 0.7520, Train Accuracy: 0.6643
Epoch 7/50, Test Loss: 0.7749, Test Accuracy: 0.6511
Epoch 8/50, Train Loss: 0.7308, Train Accuracy: 0.6803
Epoch 8/50, Test Loss: 0.7498, Test Accuracy: 0.6712
Epoch 9/50, Train Loss: 0.7390, Train Accuracy: 0.6708
Epoch 9/50, Test Loss: 0.7643, Test Accuracy: 0.6584
Epoch 10/50, Train Loss: 0.7477, Train Accuracy: 0.6601
Epoch 10/50, Test Loss: 0.7736, Test Accuracy: 0.6478
Epoch 11/50, Train Loss: 0.7281, Train Accuracy: 0.6793
Epoch 11/50, Test Loss: 0.7516, Test Accuracy: 0.6707
Epoch 12/50, Train Loss: 0.7332, Train Accuracy: 0.6776
Epoch 12/50, Test Loss: 0.7617, Test Accuracy: 0.6631
Epoch 13/50, Train Loss: 0.7237, Train Accuracy: 0.6810
Epoch 13/50, Test Loss: 0.7488, Test Accuracy: 0.6700
Epoch 14/50, Train Loss: 0.7064, Train Accuracy: 0.6919
Epoch 14/50, Test Loss: 0.7378, Test Accuracy: 0.6760
Epoch 15/50, Train Loss: 0.7147, Train Accuracy: 0.6859
Epoch 15/50, Test Loss: 0.7464, Test Accuracy: 0.6704
Epoch 16/50, Train Loss: 0.7069, Train Accuracy: 0.6835
Epoch 16/50, Test Loss: 0.7428, Test Accuracy: 0.6654
Epoch 17/50, Train Loss: 0.7042, Train Accuracy: 0.6893
Epoch 17/50, Test Loss: 0.7384, Test Accuracy: 0.6745
Epoch 18/50, Train Loss: 0.7013, Train Accuracy: 0.6924
Epoch 18/50, Test Loss: 0.7379, Test Accuracy: 0.6733
Epoch 19/50, Train Loss: 0.6911, Train Accuracy: 0.6968
Epoch 19/50, Test Loss: 0.7306, Test Accuracy: 0.6766
Epoch 20/50, Train Loss: 0.6982, Train Accuracy: 0.6921
Epoch 20/50, Test Loss: 0.7394, Test Accuracy: 0.6740
Epoch 21/50, Train Loss: 0.7192, Train Accuracy: 0.6808
Epoch 21/50, Test Loss: 0.7638, Test Accuracy: 0.6562
Epoch 22/50, Train Loss: 0.6926, Train Accuracy: 0.6948
Epoch 22/50, Test Loss: 0.7457, Test Accuracy: 0.6712
Epoch 23/50, Train Loss: 0.7041, Train Accuracy: 0.6872
Epoch 23/50, Test Loss: 0.7520, Test Accuracy: 0.6676
Epoch 24/50, Train Loss: 0.7447, Train Accuracy: 0.6683
Epoch 24/50, Test Loss: 0.7964, Test Accuracy: 0.6407
Epoch 25/50, Train Loss: 0.6775, Train Accuracy: 0.7017
Epoch 25/50, Test Loss: 0.7340, Test Accuracy: 0.6743
Epoch 26/50, Train Loss: 0.6737, Train Accuracy: 0.7056
Epoch 26/50, Test Loss: 0.7292, Test Accuracy: 0.6795
Epoch 27/50, Train Loss: 0.6877, Train Accuracy: 0.6955
Epoch 27/50, Test Loss: 0.7388, Test Accuracy: 0.6689
Epoch 28/50, Train Loss: 0.6718, Train Accuracy: 0.7045
Epoch 28/50, Test Loss: 0.7421, Test Accuracy: 0.6776
Epoch 29/50, Train Loss: 0.6977, Train Accuracy: 0.6846
Epoch 29/50, Test Loss: 0.7685, Test Accuracy: 0.6573
Epoch 30/50, Train Loss: 0.6638, Train Accuracy: 0.7073
Epoch 30/50, Test Loss: 0.7310, Test Accuracy: 0.6769
Epoch 31/50, Train Loss: 0.6658, Train Accuracy: 0.7095
Epoch 31/50, Test Loss: 0.7342, Test Accuracy: 0.6798
Epoch 32/50, Train Loss: 0.6752, Train Accuracy: 0.7005
Epoch 32/50, Test Loss: 0.7458, Test Accuracy: 0.6689
Epoch 33/50, Train Loss: 0.6651, Train Accuracy: 0.7060
Epoch 33/50, Test Loss: 0.7458, Test Accuracy: 0.6731
Epoch 34/50, Train Loss: 0.6537, Train Accuracy: 0.7098
Epoch 34/50, Test Loss: 0.7402, Test Accuracy: 0.6736
Epoch 35/50, Train Loss: 0.6460, Train Accuracy: 0.7179
Epoch 35/50, Test Loss: 0.7341, Test Accuracy: 0.6840
Epoch 36/50, Train Loss: 0.6586, Train Accuracy: 0.7109
Epoch 36/50, Test Loss: 0.7426, Test Accuracy: 0.6777
Epoch 37/50, Train Loss: 0.6675, Train Accuracy: 0.7039
Epoch 37/50, Test Loss: 0.7528, Test Accuracy: 0.6672
Epoch 38/50, Train Loss: 0.6626, Train Accuracy: 0.7060
Epoch 38/50, Test Loss: 0.7577, Test Accuracy: 0.6618
Epoch 39/50, Train Loss: 0.6457, Train Accuracy: 0.7171
Epoch 39/50, Test Loss: 0.7381, Test Accuracy: 0.6722
Epoch 40/50, Train Loss: 0.6516, Train Accuracy: 0.7149
Epoch 40/50, Test Loss: 0.7460, Test Accuracy: 0.6733
Epoch 41/50, Train Loss: 0.6393, Train Accuracy: 0.7209
Epoch 41/50, Test Loss: 0.7388, Test Accuracy: 0.6781
Epoch 42/50, Train Loss: 0.6405, Train Accuracy: 0.7161
Epoch 42/50, Test Loss: 0.7459, Test Accuracy: 0.6715
Epoch 43/50, Train Loss: 0.6360, Train Accuracy: 0.7222
Epoch 43/50, Test Loss: 0.7388, Test Accuracy: 0.6740
Epoch 44/50, Train Loss: 0.6328, Train Accuracy: 0.7230
Epoch 44/50, Test Loss: 0.7416, Test Accuracy: 0.6756
```

```
Epoch 45/50, Train Loss: 0.6431, Train Accuracy: 0.7183
Epoch 45/50, Test Loss: 0.7555, Test Accuracy: 0.6722
Epoch 46/50, Train Loss: 0.6260, Train Accuracy: 0.7267
Epoch 46/50, Test Loss: 0.7488, Test Accuracy: 0.6752
Epoch 47/50, Train Loss: 0.6395, Train Accuracy: 0.7208
Epoch 47/50, Test Loss: 0.7556, Test Accuracy: 0.6716
Epoch 48/50, Train Loss: 0.6186, Train Accuracy: 0.7310
Epoch 48/50, Test Loss: 0.7424, Test Accuracy: 0.6786
Epoch 49/50, Train Loss: 0.6353, Train Accuracy: 0.7216
Epoch 49/50, Test Loss: 0.7617, Test Accuracy: 0.6672
Epoch 50/50, Train Loss: 0.6133, Train Accuracy: 0.7333
Epoch 50/50, Test Loss: 0.7495, Test Accuracy: 0.6718
```

In [47]:
```
print (f"Accuracy for MLP for mean word2vec: {accuracy_score(Y_test, torch.argmax(model(X_test), dim=1)):.4f}")
```

```
Accuracy for MLP for mean word2vec: 0.6717
```

In [52]:
```
## deleting these variables to improve memory load
del model, train_X, train_Y, test_X, test_Y, X_train, X_test, Y_train, Y_test
```

Task 4(b) concatenate the first 10 Word2Vec vectors for each review as the input feature

For concatenation we use ignore the words that are not present in google model and if the number of valid words are less than 10, we pad the rest of the vector with zeros

We create a MLP model class with 2 hidden layers (100 and 10 nodes each) and use Relu as the activation function. We also use SGD optimizer and CrossEntropyLoss as this was giving the best accuracy.

In [54]:
```
def concatenated_vector(words):
    num_vectors = 0
    i = 0
    vector = np.empty((10*300))
    while num_vectors < 10:
        if i < len(words):
            current_vector = google_model[words[i]] if words[i] in google_model else None
            i += 1
            if current_vector is None:
                continue
        else:
            current_vector = np.zeros((300))
        vector[num_vectors*300: (num_vectors+1)*300] = current_vector
        num_vectors += 1
    return vector

dataset['reviews_vector'] = dataset['review_preprocessed_tokens'].map(concatenated_vector)
finished_dataset = dataset[['reviews_vector', 'class']]

training_data, testing_data = train_test_split(finished_dataset, test_size=0.2, random_state=25)

train_X = np.stack(training_data['reviews_vector'])
train_Y = np.array(training_data['class'])
test_X = np.stack(testing_data['reviews_vector'])
test_Y = np.array(testing_data['class'])

## have data as tensors
X_train, X_test, Y_train, Y_test = format_data_for_model(train_X, train_Y, test_X, test_Y, device)
```

In [57]:
```
# Define the multilayer perceptron network
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(300*10, 100)
        self.fc2 = nn.Linear(100, 10)
        self.fc3 = nn.Linear(10, 3)
        self.relu = nn.ReLU()

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

# Create the MLP model, optimizer and loss function
model = MLP()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()
model.to(device)
```

Out[57]:
```
MLP(
  (fc1): Linear(in_features=3000, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (relu): ReLU()
)
```

I have used the following hyperparameters:
learning rate = 0.001
loss function = CrossEntropyLoss
optimizer = SGD
momentum = 0.9
non-linear activation = Relu
epochs = 20
batch size = 32

In [58]:
```
model = train_model(model, optimizer, loss_fn, X_train, Y_train, X_test, Y_test, num_epochs=20, batch_size=32)
```

```
Epoch 1/20, Train Loss: 1.0758, Train Accuracy: 0.4897
Epoch 1/20, Test Loss: 1.0760, Test Accuracy: 0.4869
Epoch 2/20, Train Loss: 0.9562, Train Accuracy: 0.5220
Epoch 2/20, Test Loss: 0.9591, Test Accuracy: 0.5193
Epoch 3/20, Train Loss: 0.9066, Train Accuracy: 0.5654
Epoch 3/20, Test Loss: 0.9209, Test Accuracy: 0.5518
Epoch 4/20, Train Loss: 0.8707, Train Accuracy: 0.5950
Epoch 4/20, Test Loss: 0.8978, Test Accuracy: 0.5739
Epoch 5/20, Train Loss: 0.8460, Train Accuracy: 0.6092
Epoch 5/20, Test Loss: 0.8853, Test Accuracy: 0.5823
Epoch 6/20, Train Loss: 0.8235, Train Accuracy: 0.6217
Epoch 6/20, Test Loss: 0.8747, Test Accuracy: 0.5918
Epoch 7/20, Train Loss: 0.8080, Train Accuracy: 0.6319
Epoch 7/20, Test Loss: 0.8720, Test Accuracy: 0.5928
Epoch 8/20, Train Loss: 0.7924, Train Accuracy: 0.6418
Epoch 8/20, Test Loss: 0.8683, Test Accuracy: 0.5901
Epoch 9/20, Train Loss: 0.7752, Train Accuracy: 0.6497
Epoch 9/20, Test Loss: 0.8674, Test Accuracy: 0.5932
Epoch 10/20, Train Loss: 0.7629, Train Accuracy: 0.6561
Epoch 10/20, Test Loss: 0.8707, Test Accuracy: 0.5908
Epoch 11/20, Train Loss: 0.7419, Train Accuracy: 0.6706
Epoch 11/20, Test Loss: 0.8692, Test Accuracy: 0.5888
Epoch 12/20, Train Loss: 0.7181, Train Accuracy: 0.6873
Epoch 12/20, Test Loss: 0.8706, Test Accuracy: 0.5902
Epoch 13/20, Train Loss: 0.6974, Train Accuracy: 0.6973
Epoch 13/20, Test Loss: 0.8746, Test Accuracy: 0.5900
Epoch 14/20, Train Loss: 0.6665, Train Accuracy: 0.7203
Epoch 14/20, Test Loss: 0.8763, Test Accuracy: 0.5907
Epoch 15/20, Train Loss: 0.6334, Train Accuracy: 0.7403
Epoch 15/20, Test Loss: 0.8833, Test Accuracy: 0.5901
Epoch 16/20, Train Loss: 0.6049, Train Accuracy: 0.7559
Epoch 16/20, Test Loss: 0.9073, Test Accuracy: 0.5838
Epoch 17/20, Train Loss: 0.5582, Train Accuracy: 0.7849
Epoch 17/20, Test Loss: 0.9148, Test Accuracy: 0.5862
Epoch 18/20, Train Loss: 0.5090, Train Accuracy: 0.8131
Epoch 18/20, Test Loss: 0.9319, Test Accuracy: 0.5861
Epoch 19/20, Train Loss: 0.4641, Train Accuracy: 0.8350
Epoch 19/20, Test Loss: 0.9738, Test Accuracy: 0.5779
Epoch 20/20, Train Loss: 0.4054, Train Accuracy: 0.8694
Epoch 20/20, Test Loss: 0.9961, Test Accuracy: 0.5791
```

In [59]:
```
print (f"Accuracy for MLP for concatenated word2vec: {accuracy_score(Y_test, torch.argmax(model(X_test), dim=1)):.4f}'
```

```
Accuracy for MLP for concatenated word2vec: 0.5791
```

In [60]:
```
del model, X_train, X_test, Y_train, Y_test
```

The accuracy using mean word2vec is greater than concatenated word2vec on testing set. This could be because we are just considering the first 10 words in a review which might not contain a lot of useful information many times.

When compared with simple models with word2vec embeddings (Task 3), we see that MLP with mean vectors performs better than Perceptron and SVM. This is probably because it is able to encode complex information in the vectors more effectively when comapred to the simple models. MLP with concatenated vectors still performs worse. This is probably because of the same reason as mentioned above that taking first 10 words might be leading to loss of information present in the rest of the review.

Task 5

For serial vectors, we consider the first 20 valid words present in google model in the review and pad the remaining vectors (if 20 valid vectors are not found) with zeros. Thus we get a 20*300 input vector for each instance.

In [61]:
```
def series_vector(words):
    num_vectors = 0
    i = 0
    vector = np.empty((20,300))
    while num_vectors < 20:
        if i < len(words):
            current_vector = google_model[words[i]] if words[i] in google_model else None
            i += 1
            if current_vector is None:
                continue
        else:
            current_vector = np.zeros((300))
        vector[num_vectors] = current_vector
        num_vectors += 1
    return vector

dataset['reviews_vector'] = dataset['review_preprocessed_tokens'].map(series_vector)
finished_dataset = dataset[['reviews_vector', 'class']]

training_data, testing_data = train_test_split(finished_dataset, test_size=0.2, random_state=25)

train_X = np.stack(training_data['reviews_vector'])
train_Y = np.array(training_data['class'])
test_X = np.stack(testing_data['reviews_vector'])
test_Y =np.array(testing_data['class'])

X_train, X_test, Y_train, Y_test = format_data_for_model(train_X, train_Y, test_X, test_Y, device)
```

Task 5 (a) RNN

We create a RNN model class with a hidden layer of size 20. The output of the hidden layer (20 outputs from each series) is averaged before passing to the FC later during forward pass. We also use SGD optimizer and CrossEntropyLoss as this was giving the best accuracy.

In [62]:
```
class RNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.rnn = nn.RNN(input_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        output, hidden = self.rnn(text)
        output = output.mean(dim=1)
        fc_output = self.fc(output)
        return fc_output

model = RNN(300, 20, 3)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()
model.to(device)
```

Out[62]:
```
RNN(
  (rnn): RNN(300, 20, batch_first=True)
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

I have used the following hyperparameters:
learning rate = 0.01
loss function = CrossEntropyLoss
optimizer = SGD
momentum = 0.9
epochs = 40
batch size = 32

In [63]:
```
model = train_model(model, optimizer, loss_fn, X_train, Y_train, X_test, Y_test, num_epochs=40, batch_size=32)
```

```
Epoch 1/40, Train Loss: 0.9357, Train Accuracy: 0.5101
Epoch 1/40, Test Loss: 0.9421, Test Accuracy: 0.5083
Epoch 2/40, Train Loss: 0.8597, Train Accuracy: 0.6046
Epoch 2/40, Test Loss: 0.8688, Test Accuracy: 0.6024
Epoch 3/40, Train Loss: 0.8344, Train Accuracy: 0.6231
Epoch 3/40, Test Loss: 0.8428, Test Accuracy: 0.6167
Epoch 4/40, Train Loss: 0.8351, Train Accuracy: 0.6244
Epoch 4/40, Test Loss: 0.8460, Test Accuracy: 0.6168
Epoch 5/40, Train Loss: 0.8035, Train Accuracy: 0.6359
Epoch 5/40, Test Loss: 0.8181, Test Accuracy: 0.6252
Epoch 6/40, Train Loss: 0.7936, Train Accuracy: 0.6407
Epoch 6/40, Test Loss: 0.8079, Test Accuracy: 0.6327
Epoch 7/40, Train Loss: 0.8058, Train Accuracy: 0.6293
Epoch 7/40, Test Loss: 0.8182, Test Accuracy: 0.6198
Epoch 8/40, Train Loss: 0.8001, Train Accuracy: 0.6373
Epoch 8/40, Test Loss: 0.8203, Test Accuracy: 0.6289
Epoch 9/40, Train Loss: 0.7878, Train Accuracy: 0.6453
Epoch 9/40, Test Loss: 0.8127, Test Accuracy: 0.6347
Epoch 10/40, Train Loss: 0.7734, Train Accuracy: 0.6490
Epoch 10/40, Test Loss: 0.7962, Test Accuracy: 0.6344
Epoch 11/40, Train Loss: 0.7729, Train Accuracy: 0.6487
Epoch 11/40, Test Loss: 0.8017, Test Accuracy: 0.6332
Epoch 12/40, Train Loss: 0.7897, Train Accuracy: 0.6391
Epoch 12/40, Test Loss: 0.8124, Test Accuracy: 0.6258
Epoch 13/40, Train Loss: 0.7773, Train Accuracy: 0.6530
Epoch 13/40, Test Loss: 0.8089, Test Accuracy: 0.6396
Epoch 14/40, Train Loss: 0.7583, Train Accuracy: 0.6582
Epoch 14/40, Test Loss: 0.7925, Test Accuracy: 0.6426
Epoch 15/40, Train Loss: 0.7506, Train Accuracy: 0.6612
Epoch 15/40, Test Loss: 0.7879, Test Accuracy: 0.6446
Epoch 16/40, Train Loss: 0.7706, Train Accuracy: 0.6547
Epoch 16/40, Test Loss: 0.7915, Test Accuracy: 0.6400
Epoch 17/40, Train Loss: 0.7929, Train Accuracy: 0.6436
Epoch 17/40, Test Loss: 0.8260, Test Accuracy: 0.6310
Epoch 18/40, Train Loss: 0.7553, Train Accuracy: 0.6660
Epoch 18/40, Test Loss: 0.7836, Test Accuracy: 0.6476
Epoch 19/40, Train Loss: 0.7594, Train Accuracy: 0.6554
Epoch 19/40, Test Loss: 0.7981, Test Accuracy: 0.6384
Epoch 20/40, Train Loss: 0.7437, Train Accuracy: 0.6653
Epoch 20/40, Test Loss: 0.7863, Test Accuracy: 0.6459
Epoch 21/40, Train Loss: 0.7369, Train Accuracy: 0.6694
Epoch 21/40, Test Loss: 0.7802, Test Accuracy: 0.6467
Epoch 22/40, Train Loss: 0.7325, Train Accuracy: 0.6729
Epoch 22/40, Test Loss: 0.7749, Test Accuracy: 0.6461
Epoch 23/40, Train Loss: 0.7745, Train Accuracy: 0.6490
Epoch 23/40, Test Loss: 0.8022, Test Accuracy: 0.6312
Epoch 24/40, Train Loss: 0.7304, Train Accuracy: 0.6752
Epoch 24/40, Test Loss: 0.7747, Test Accuracy: 0.6500
Epoch 25/40, Train Loss: 0.7242, Train Accuracy: 0.6789
Epoch 25/40, Test Loss: 0.7753, Test Accuracy: 0.6493
Epoch 26/40, Train Loss: 0.7237, Train Accuracy: 0.6782
Epoch 26/40, Test Loss: 0.7739, Test Accuracy: 0.6509
Epoch 27/40, Train Loss: 0.7262, Train Accuracy: 0.6787
Epoch 27/40, Test Loss: 0.7793, Test Accuracy: 0.6506
Epoch 28/40, Train Loss: 0.7358, Train Accuracy: 0.6700
Epoch 28/40, Test Loss: 0.7921, Test Accuracy: 0.6423
Epoch 29/40, Train Loss: 0.7184, Train Accuracy: 0.6827
Epoch 29/40, Test Loss: 0.7715, Test Accuracy: 0.6521
Epoch 30/40, Train Loss: 0.7167, Train Accuracy: 0.6825
Epoch 30/40, Test Loss: 0.7734, Test Accuracy: 0.6515
Epoch 31/40, Train Loss: 0.7193, Train Accuracy: 0.6822
Epoch 31/40, Test Loss: 0.7716, Test Accuracy: 0.6505
Epoch 32/40, Train Loss: 0.7189, Train Accuracy: 0.6774
Epoch 32/40, Test Loss: 0.7826, Test Accuracy: 0.6437
Epoch 33/40, Train Loss: 0.7264, Train Accuracy: 0.6719
Epoch 33/40, Test Loss: 0.7954, Test Accuracy: 0.6405
Epoch 34/40, Train Loss: 0.7225, Train Accuracy: 0.6791
Epoch 34/40, Test Loss: 0.7816, Test Accuracy: 0.6440
Epoch 35/40, Train Loss: 0.7367, Train Accuracy: 0.6646
Epoch 35/40, Test Loss: 0.7969, Test Accuracy: 0.6371
Epoch 36/40, Train Loss: 0.7136, Train Accuracy: 0.6839
Epoch 36/40, Test Loss: 0.7835, Test Accuracy: 0.6495
Epoch 37/40, Train Loss: 0.7177, Train Accuracy: 0.6798
Epoch 37/40, Test Loss: 0.7987, Test Accuracy: 0.6437
Epoch 38/40, Train Loss: 0.7157, Train Accuracy: 0.6812
Epoch 38/40, Test Loss: 0.7851, Test Accuracy: 0.6472
Epoch 39/40, Train Loss: 0.7066, Train Accuracy: 0.6872
Epoch 39/40, Test Loss: 0.7767, Test Accuracy: 0.6519
Epoch 40/40, Train Loss: 0.7033, Train Accuracy: 0.6894
Epoch 40/40, Test Loss: 0.7757, Test Accuracy: 0.6478
```

In [65]:
```
print (f"Accuracy for RNN: {accuracy_score(Y_test, torch.argmax(model(X_test), dim=1)):.4f}")
```

```
Accuracy for RNN: 0.6478
```

The accuracy is slightly less than MLP with mean word2vec vectors but it is higher than the MLP with concatenated word2vec vectors. This is because in RNN we are considering the first 20 words which have more information than 10 words in concatenated word2vec. Also, MLP with mean vectors might be performing better because of the vanishing gradient problem in RNN and that still some information might be missing for reviews greater than length 20.

Task 5 (b) GRU

same as RNN model but GRU layer instead of RNN

In [66]:
```
class GRU(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.rnn = nn.GRU(input_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        output, hidden = self.rnn(text)
        output = output.mean(dim=1)
        fc_output = self.fc(output)
        return fc_output

model = GRU(300, 20, 3)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()
model.to(device)
```

Out[66]:
```
GRU(
  (rnn): GRU(300, 20, batch_first=True)
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

I have used the following hyperparameters:

learning rate = 0.01

loss function = CrossEntropyLoss

optimizer = SGD

momentum = 0.9

epochs = 30

batch size = 32

In [67]:
```
model = train_model(model, optimizer, loss_fn, X_train, Y_train, X_test, Y_test, num_epochs=30, batch_size=32)
```

```
Epoch 1/30, Train Loss: 0.8834, Train Accuracy: 0.5820
Epoch 1/30, Test Loss: 0.8865, Test Accuracy: 0.5762
Epoch 2/30, Train Loss: 0.8301, Train Accuracy: 0.6193
Epoch 2/30, Test Loss: 0.8340, Test Accuracy: 0.6189
Epoch 3/30, Train Loss: 0.8181, Train Accuracy: 0.6231
Epoch 3/30, Test Loss: 0.8221, Test Accuracy: 0.6220
Epoch 4/30, Train Loss: 0.7996, Train Accuracy: 0.6370
Epoch 4/30, Test Loss: 0.8061, Test Accuracy: 0.6348
Epoch 5/30, Train Loss: 0.7844, Train Accuracy: 0.6437
Epoch 5/30, Test Loss: 0.7954, Test Accuracy: 0.6352
Epoch 6/30, Train Loss: 0.7980, Train Accuracy: 0.6357
Epoch 6/30, Test Loss: 0.8120, Test Accuracy: 0.6268
Epoch 7/30, Train Loss: 0.7885, Train Accuracy: 0.6415
Epoch 7/30, Test Loss: 0.8021, Test Accuracy: 0.6342
Epoch 8/30, Train Loss: 0.7570, Train Accuracy: 0.6581
Epoch 8/30, Test Loss: 0.7743, Test Accuracy: 0.6455
Epoch 9/30, Train Loss: 0.7557, Train Accuracy: 0.6581
Epoch 9/30, Test Loss: 0.7759, Test Accuracy: 0.6445
Epoch 10/30, Train Loss: 0.7443, Train Accuracy: 0.6668
Epoch 10/30, Test Loss: 0.7646, Test Accuracy: 0.6575
Epoch 11/30, Train Loss: 0.7597, Train Accuracy: 0.6508
Epoch 11/30, Test Loss: 0.7807, Test Accuracy: 0.6382
Epoch 12/30, Train Loss: 0.7593, Train Accuracy: 0.6556
Epoch 12/30, Test Loss: 0.7854, Test Accuracy: 0.6413
Epoch 13/30, Train Loss: 0.7342, Train Accuracy: 0.6706
Epoch 13/30, Test Loss: 0.7627, Test Accuracy: 0.6534
Epoch 14/30, Train Loss: 0.7232, Train Accuracy: 0.6779
Epoch 14/30, Test Loss: 0.7520, Test Accuracy: 0.6614
Epoch 15/30, Train Loss: 0.7506, Train Accuracy: 0.6617
Epoch 15/30, Test Loss: 0.7813, Test Accuracy: 0.6486
Epoch 16/30, Train Loss: 0.7148, Train Accuracy: 0.6821
Epoch 16/30, Test Loss: 0.7492, Test Accuracy: 0.6654
Epoch 17/30, Train Loss: 0.7237, Train Accuracy: 0.6752
Epoch 17/30, Test Loss: 0.7609, Test Accuracy: 0.6562
Epoch 18/30, Train Loss: 0.7040, Train Accuracy: 0.6869
Epoch 18/30, Test Loss: 0.7439, Test Accuracy: 0.6648
Epoch 19/30, Train Loss: 0.7029, Train Accuracy: 0.6869
Epoch 19/30, Test Loss: 0.7480, Test Accuracy: 0.6628
Epoch 20/30, Train Loss: 0.6982, Train Accuracy: 0.6885
Epoch 20/30, Test Loss: 0.7476, Test Accuracy: 0.6660
Epoch 21/30, Train Loss: 0.6960, Train Accuracy: 0.6900
Epoch 21/30, Test Loss: 0.7480, Test Accuracy: 0.6656
Epoch 22/30, Train Loss: 0.6872, Train Accuracy: 0.6945
Epoch 22/30, Test Loss: 0.7389, Test Accuracy: 0.6657
Epoch 23/30, Train Loss: 0.6862, Train Accuracy: 0.6948
Epoch 23/30, Test Loss: 0.7395, Test Accuracy: 0.6672
Epoch 24/30, Train Loss: 0.6905, Train Accuracy: 0.6928
Epoch 24/30, Test Loss: 0.7510, Test Accuracy: 0.6610
Epoch 25/30, Train Loss: 0.6849, Train Accuracy: 0.6961
Epoch 25/30, Test Loss: 0.7426, Test Accuracy: 0.6626
Epoch 26/30, Train Loss: 0.6843, Train Accuracy: 0.6975
Epoch 26/30, Test Loss: 0.7469, Test Accuracy: 0.6633
Epoch 27/30, Train Loss: 0.6776, Train Accuracy: 0.7001
Epoch 27/30, Test Loss: 0.7415, Test Accuracy: 0.6638
Epoch 28/30, Train Loss: 0.6729, Train Accuracy: 0.7013
Epoch 28/30, Test Loss: 0.7409, Test Accuracy: 0.6666
Epoch 29/30, Train Loss: 0.6623, Train Accuracy: 0.7078
Epoch 29/30, Test Loss: 0.7348, Test Accuracy: 0.6691
Epoch 30/30, Train Loss: 0.6791, Train Accuracy: 0.6980
Epoch 30/30, Test Loss: 0.7501, Test Accuracy: 0.6603
```

In [68]:
```
print (f"Accuracy for GRU: {accuracy_score(Y_test, torch.argmax(model(X_test), dim=1)):.4f}")
```

```
Accuracy for GRU: 0.6603
```

The accuracy is better than simple RNN. It is now comparable to MLP with mean word2vec vectors and much higher than the MLP with concatenated word2vec vectors. Rest of the factors while comparing with MLP remain same as the RNN explanation above, but the vanishing gradient problem is reduced a bit and hence we see some better results than simple RNN.

Task 5 (b) LSTM

same as RNN model but LSTM layer instead of RNN

In [69]:
```python
class LSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.rnn = nn.LSTM(input_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        output, hidden = self.rnn(text)
        output = output.mean(dim=1)
        fc_output = self.fc(output)
        return fc_output

model = LSTM(300, 20, 3)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()
model.to(device)
```

Out[69]:
```
LSTM(
  (rnn): LSTM(300, 20, batch_first=True)
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

I have used the following hyperparameters:

learning rate = 0.01

loss function = CrossEntropyLoss

optimizer = SGD

momentum = 0.9

epochs = 30

batch size = 32

In [70]:
```
model = train_model(model, optimizer, loss_fn, X_train, Y_train, X_test, Y_test, num_epochs=30, batch_size=32)
```

```
Epoch 1/30, Train Loss: 0.9286, Train Accuracy: 0.5455
Epoch 1/30, Test Loss: 0.9339, Test Accuracy: 0.5402
Epoch 2/30, Train Loss: 0.8642, Train Accuracy: 0.5926
Epoch 2/30, Test Loss: 0.8701, Test Accuracy: 0.5867
Epoch 3/30, Train Loss: 0.8247, Train Accuracy: 0.6250
Epoch 3/30, Test Loss: 0.8313, Test Accuracy: 0.6184
Epoch 4/30, Train Loss: 0.8333, Train Accuracy: 0.6178
Epoch 4/30, Test Loss: 0.8407, Test Accuracy: 0.6158
Epoch 5/30, Train Loss: 0.7945, Train Accuracy: 0.6382
Epoch 5/30, Test Loss: 0.8065, Test Accuracy: 0.6302
Epoch 6/30, Train Loss: 0.7854, Train Accuracy: 0.6469
Epoch 6/30, Test Loss: 0.7969, Test Accuracy: 0.6381
Epoch 7/30, Train Loss: 0.7695, Train Accuracy: 0.6532
Epoch 7/30, Test Loss: 0.7849, Test Accuracy: 0.6453
Epoch 8/30, Train Loss: 0.7646, Train Accuracy: 0.6533
Epoch 8/30, Test Loss: 0.7821, Test Accuracy: 0.6416
Epoch 9/30, Train Loss: 0.7702, Train Accuracy: 0.6531
Epoch 9/30, Test Loss: 0.7876, Test Accuracy: 0.6408
Epoch 10/30, Train Loss: 0.7542, Train Accuracy: 0.6622
Epoch 10/30, Test Loss: 0.7747, Test Accuracy: 0.6488
Epoch 11/30, Train Loss: 0.7452, Train Accuracy: 0.6633
Epoch 11/30, Test Loss: 0.7694, Test Accuracy: 0.6534
Epoch 12/30, Train Loss: 0.7487, Train Accuracy: 0.6642
Epoch 12/30, Test Loss: 0.7730, Test Accuracy: 0.6488
Epoch 13/30, Train Loss: 0.7343, Train Accuracy: 0.6704
Epoch 13/30, Test Loss: 0.7635, Test Accuracy: 0.6544
Epoch 14/30, Train Loss: 0.7310, Train Accuracy: 0.6719
Epoch 14/30, Test Loss: 0.7606, Test Accuracy: 0.6561
Epoch 15/30, Train Loss: 0.7317, Train Accuracy: 0.6733
Epoch 15/30, Test Loss: 0.7641, Test Accuracy: 0.6548
Epoch 16/30, Train Loss: 0.7196, Train Accuracy: 0.6780
Epoch 16/30, Test Loss: 0.7567, Test Accuracy: 0.6566
Epoch 17/30, Train Loss: 0.7148, Train Accuracy: 0.6806
Epoch 17/30, Test Loss: 0.7516, Test Accuracy: 0.6590
Epoch 18/30, Train Loss: 0.7228, Train Accuracy: 0.6763
Epoch 18/30, Test Loss: 0.7595, Test Accuracy: 0.6543
Epoch 19/30, Train Loss: 0.7245, Train Accuracy: 0.6713
Epoch 19/30, Test Loss: 0.7687, Test Accuracy: 0.6513
Epoch 20/30, Train Loss: 0.7091, Train Accuracy: 0.6814
Epoch 20/30, Test Loss: 0.7529, Test Accuracy: 0.6577
Epoch 21/30, Train Loss: 0.7037, Train Accuracy: 0.6870
Epoch 21/30, Test Loss: 0.7496, Test Accuracy: 0.6584
Epoch 22/30, Train Loss: 0.7020, Train Accuracy: 0.6848
Epoch 22/30, Test Loss: 0.7514, Test Accuracy: 0.6601
Epoch 23/30, Train Loss: 0.6947, Train Accuracy: 0.6909
Epoch 23/30, Test Loss: 0.7510, Test Accuracy: 0.6609
Epoch 24/30, Train Loss: 0.6916, Train Accuracy: 0.6916
Epoch 24/30, Test Loss: 0.7555, Test Accuracy: 0.6614
Epoch 25/30, Train Loss: 0.6917, Train Accuracy: 0.6911
Epoch 25/30, Test Loss: 0.7458, Test Accuracy: 0.6640
Epoch 26/30, Train Loss: 0.7105, Train Accuracy: 0.6799
Epoch 26/30, Test Loss: 0.7728, Test Accuracy: 0.6542
Epoch 27/30, Train Loss: 0.6784, Train Accuracy: 0.6967
Epoch 27/30, Test Loss: 0.7409, Test Accuracy: 0.6645
Epoch 28/30, Train Loss: 0.6809, Train Accuracy: 0.6991
Epoch 28/30, Test Loss: 0.7516, Test Accuracy: 0.6624
Epoch 29/30, Train Loss: 0.6744, Train Accuracy: 0.7000
Epoch 29/30, Test Loss: 0.7472, Test Accuracy: 0.6666
Epoch 30/30, Train Loss: 0.6692, Train Accuracy: 0.7031
Epoch 30/30, Test Loss: 0.7456, Test Accuracy: 0.6638
```

In [71]:
```
print (f"Accuracy for LSTM: {accuracy_score(Y_test, torch.argmax(model(X_test), dim=1)):.4f}")
```

```
Accuracy for LSTM: 0.6638
```

The accuracy is better than simple RNN and slightly better (almost same) than GRU. The comparison with MLP remains the same as RNN as well as the explanations. The vanishing gradient problem is reduced a bit compared to RNN and hence we see some better results than simple RNN. LSTM and GRU give almost similar results as they both handle vanishing gradient in their own way and both seem to performing almost equally in our case. (GRU and LSTM order varies from run to run).

GRU ~= LSTM > Simple RNN.

In this case, we see that LSTM performs the best, than GRU and finally simple RNN. This is because RNN faces the problem of vanisihing gradients which both LSTM and GRU try to handle in their own ways. GRU and LSTM both are equally good and their relative order is usually varying from run to run.

In [ ]:
```

```