CSCI 544 - HW3 - USC ID: 5978435849

Dependencies: Python version: 3.11.0 Conda version: 23.1.0 pytorch version: 1.13.1

References:

1. https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html (https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html)
2. https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist (https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist)
3. https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html (https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)
4. https://youtu.be/WEV61GmmPrk (https://youtu.be/WEV61GmmPrk)
5. https://youtu.be/0_PgWWmauHk (https://youtu.be/0_PgWWmauHk)

In [1]:
```
import pandas as pd
import numpy as np
import gensim
from gensim.utils import simple_preprocess
from sklearn.linear_model import Perceptron
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
import torch
import torch.nn as nn
from gensim.models.word2vec import Word2Vec
from gensim.models import KeyedVectors
import gensim.downloader as dwnld
```

1. Dataset generation

In [2]:

```
# dataframe = pd.read_table('amazon_reviews_us_Beauty_v1_00.tsv', on_bad_lines='skip');
dataframe = pd.read_table('data.tsv', on_bad_lines='skip');
# print(list(dataframe))
df = dataframe[['star_rating','review_body']]
```

```
C:\Users\Sai Kumar Peddholla\AppData\Local\Temp\ipykernel_9816\99976983.py:2: DtypeWarning: Columns (7)
have mixed types. Specify dtype option on import or set low_memory=False.
  dataframe = pd.read_table('data.tsv', on_bad_lines='skip');
```

In [3]:

```
# Generating random sample data, from each class
class1 = df.loc[df['star_rating'].isin([1,2])]
class2 = df.loc[df['star_rating'].isin([3])]
class3 = df.loc[df['star_rating'].isin([4,5])]

class1 = class1.sample(n=20000)
class2 = class2.sample(n=20000)
class3 = class3.sample(n=20000)

class1['star_rating'] = class1['star_rating'].apply(lambda x: 1)
class2['star_rating'] = class2['star_rating'].apply(lambda x: 2)
class3['star_rating'] = class3['star_rating'].apply(lambda x: 3)

sample_data = pd.concat([class1, class2, class3], axis=0)
print(sample_data)
```

```
         star_rating                                        review_body
243524             1                             DON'T WASTE YOUR MONEY.
598830             1  At first glance this was an awesome product un...
3681814            1  I have used this product for about a year and ...
3419093            1  I received my order today and I am a bit conce...
421958             1  I've only had it since April, and it's already...
...              ...                                                ...
1685159            3  ... so relaxing to take bath with this stuff ....
3818342            3  It is the best size pillow for laying in tub. ...
1892285            3                         Leaves my skin silky smooth.
2174487            3  mainly used in conjunction with light moisturi...
4606435            3  I have been looking for something like this fo...

[60000 rows x 2 columns]
```

2. Word Embedding

a) Loading google word2vec model

In [4]:
```
w2v_google_model = dwnld.load('word2vec-google-news-300')
# Used below while testing to save loading time
# w2v_google_model.save('./w2v_google_model.model')
# w2v_google_model = KeyedVectors.load('w2v_google_model.model')
```

In [5]:
```
# Computing simillarities between words
print("king~queen:",w2v_google_model.similarity("king", "queen"))
print("man~woman:",w2v_google_model.similarity("man", "woman"))
print("excellent~outstanding:",w2v_google_model.similarity("excellent", "outstanding"))
print("sad~unhappy:",w2v_google_model.similarity("sad", "unhappy"))
print("fast~quick:",w2v_google_model.similarity("fast", "quick"))
```

```
king~queen: 0.6510957
man~woman: 0.76640123
excellent~outstanding: 0.5567486
sad~unhappy: 0.41572243
fast~quick: 0.57016057
```

In [6]:
```
# Computing simillarities between words
print("king->queen::man->",w2v_google_model.most_similar(positive=['king','queen'],negative=['man']))
print("excellent->outstanding::sad->",w2v_google_model.most_similar(positive=['excellent','outstanding'],r
```

```
king->queen::man-> [('queens', 0.595018744468689), ('monarch', 0.5815044641494751), ('kings', 0.56129932
40356445), ('royal', 0.5204525589942932), ('princess', 0.5191516876220703), ('princes', 0.50863921642303
47), ('NYC_anglophiles_aflutter', 0.5057314038276672), ('Queen_Consort', 0.49256712198257446), ('Queen',
0.4822567403316498), ('royals', 0.4781742990016937)]
excellent->outstanding::sad-> [('oustanding', 0.6630989909172058), ('exceptional', 0.6203196048736572),
('superb', 0.5520898103713989), ('exemplary', 0.5066716074943542), ('terrific', 0.5014314651489258), ('E
xcellent', 0.4898416996002197), ('impeccable', 0.47491276264190674), ('superior', 0.4728458523750305),
('superlative', 0.47127777338027954), ('Outstanding', 0.46214401721954346)]
```

b) Word2Vec model using amazon dataset

In [7]:

```
# Define the parameters for Word2Vec model
vector_size = 300 # dimensionality of the word vectors
window_size = 13 # size of the context window
min_count = 9 # minimum frequency of a word to be included in the vocabulary


# Convert the reviews column to a list of sentences.lowercased the reviews and removed all non-alphanumeri
reviews = sample_data['review_body'].str.lower().str.replace('[^\w\s]','').str.split().tolist()

# The below code filters out any invalid values (e.g. floats) from each sub-list using a list comprehensio
filtered_reviews = []
for review in reviews:
    if isinstance(review, list):
        cur_list = []
        for word in review:
            if isinstance(word, str):
                cur_list.append(word)
        filtered_reviews.append(cur_list)
    else:
        filtered_reviews.append([])

reviews = filtered_reviews
# Train the Word2Vec model
model = Word2Vec(reviews, vector_size=vector_size, window=window_size, min_count=min_count)
```

```
C:\Users\Sai Kumar Peddholla\AppData\Local\Temp\ipykernel_9816\839809871.py:8: FutureWarning: The defaul
t value of regex will change from True to False in a future version.
  reviews = sample_data['review_body'].str.lower().str.replace('[^\w\s]','').str.split().tolist()
```

In [8]:

```
# Computing simillarities between words
print("king~queen:",model.wv.similarity("king", "queen"))
print("man~woman:",model.wv.similarity("man", "woman"))
print("excellent~outstanding:",model.wv.similarity("excellent", "outstanding"))
print("sad~unhappy:",model.wv.similarity("sad", "unhappy"))
print("fast~quick:",model.wv.similarity("fast", "quick"))
```

```
king~queen: 0.5034378
man~woman: 0.7376918
excellent~outstanding: 0.7757071
sad~unhappy: 0.55369514
fast~quick: 0.7835034
```

In [9]:

```
# Finding most suitable words based on analogy. Since the dataset is different simmilar words would
# be different and also their probabilities
print("king->queen::man->",model.wv.most_similar(positive=['king','queen'],negative=['man']))
print("excellent->outstanding::sad->",model.wv.most_similar(positive=['excellent','outstanding'],negative=
```

```
king->queen::man-> [('francisco', 0.7195603251457214), ('san', 0.6869110465049744), ('buckthorn', 0.6863
482594490051), ('arbonne', 0.6852318644523621), ('palmitate', 0.6786541938781738), ('origin', 0.67532503
60488892), ('avalon', 0.674170196056366), ('resurfacing', 0.6731386184692383), ('botanical', 0.673132061
958313), ('cell', 0.6716758012771606)]
excellent->outstanding::sad-> [('superb', 0.7255911231040955), ('inferior', 0.605215311050415), ('basi
c', 0.6045570969581604), ('exceptional', 0.6037482023239136), ('adequate', 0.588214099407196), ('interes
ting', 0.5643987655639648), ('providing', 0.5642382502555847), ('equal', 0.5430349111557007), ('include
s', 0.5429861545562744), ('superior', 0.524884045124054)]
```

Q. What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better?

Ans. From the comparison of the vectors generated by the custom model (amazon dataset) and the google model, it can be concluded that the google Word2Vec model appears to encode semantic similarities between words better.

In particular, the similarity scores for the word pairs "king~~queen" and "man~~woman" are higher in the google model, indicating a better encoding of the semantic relationship between these pairs. Additionally, the similarity score for "excellent~~outstanding" in the custom model is higher than the similarity score for "excellent~~outstanding" in the google model, further suggesting that the custom model better captures semantic similarities between words particular to its domain.

Overall, the pre-trained Word2Vec model seems to have learned more general and robust semantic relationships between words, while the custom model may be limited by the specific dataset it was trained on.

3. Simple models

Splitting Dataset into Train and test. [This split is consistent for training all the models]

In [10]:
```
# changes classes from 1->0; 2->1; 3->2 to make it cinsistent in svm, percpetron and FNNs(since target val
y = sample_data['star_rating'].map({1: 0, 2: 1, 3: 2})
X = reviews
X_train_main, X_test_main, y_train_main, y_test_main = train_test_split(X, y, test_size=0.2)
```

In [11]:
```
# Computing average of word2vec features of all words for a review
def get_word2vec(review):
    words = review
    vectors = []
    for word in words:
        if word in w2v_google_model:
            vectors.append(w2v_google_model[word])
    if len(vectors) > 0:
        return np.mean(vectors, axis=0)
    else:
        return np.zeros(300)

X_train = np.array([get_word2vec(review) for review in X_train_main])
X_test = np.array([get_word2vec(review) for review in X_test_main])
y_train = y_train_main
y_test = y_test_main
```

Training Perceptron and SVM using word2vec features

In [12]:
```
# Perceptron model
perceptron = Perceptron()
perceptron.fit(X_train, y_train)
y_pred = perceptron.predict(X_test)
word2vec_perceptron_acc = accuracy_score(y_test, y_pred)

# SVM model
svm = LinearSVC()
svm.fit(X_train, y_train)
y_pred = svm.predict(X_test)
word2vec_svm_acc = accuracy_score(y_test, y_pred)
```

In [13]:
```
print("Word2Vec perceptron accuracy: ",word2vec_perceptron_acc)
print("Word2Vec svm accuracy: ",word2vec_svm_acc)
```

```
Word2Vec perceptron accuracy:  0.4176666666666667
Word2Vec svm accuracy:  0.6515833333333333
```

Training Perceptron and SVM using Tf-Idf features

In [14]:
```
# Initialize a TfidfVectorizer with desired parameters
vectorizer = TfidfVectorizer(ngram_range=(1,2))

X_temp = sample_data['review_body'].fillna('')
# Fit and transform the reviews using the vectorizer
X_tfidf = vectorizer.fit_transform(X_temp)

X_train_tfidf, X_test_tfidf, y_train_tfidf, y_test_tfidf = train_test_split(X_tfidf, y, test_size=0.2)

# Perceptron model
perceptron = Perceptron()
perceptron.fit(X_train_tfidf, y_train_tfidf)
y_pred = perceptron.predict(X_test_tfidf)
tfidf_perceptron_acc = accuracy_score(y_test_tfidf, y_pred)

# SVM model
svm = LinearSVC()
svm.fit(X_train_tfidf, y_train_tfidf)
y_pred = svm.predict(X_test_tfidf)
tfidf_svm_acc = accuracy_score(y_test_tfidf, y_pred)
```

In [15]:
```
print("Tf-Idf perceptron accuracy: ",tfidf_perceptron_acc)
print("Tf-Idf svm accuracy: ",tfidf_svm_acc)
```

```
Tf-Idf perceptron accuracy:  0.6948333333333333
Tf-Idf svm accuracy:  0.72525
```

Q. What do you conclude from comparing performances for the models trained using the two different feature types?

Ans. Based on the accuracy scores, it can be concluded that the models trained using the Tf-Idf feature type outperformed the models trained using the Word2Vec feature type.

The perceptron and SVM models trained on Tf-Idf features achieved higher accuracy scores compared to those trained on Word2Vec features. The perceptron model achieved an accuracy of 0.69 with Tf-Idf features, while it achieved only 0.41 with Word2Vec features. Similarly, the SVM model achieved an accuracy of 0.72 with Tf-Idf features, while it achieved only 0.65 with Word2Vec features.

This suggests that the Tf-Idf feature type is better suited for sentiment analysis tasks than the Word2Vec feature type. This may be because the Tf-Idf representation captures the frequency of words in each document, which can be indicative of sentiment, while the Word2Vec representation may not capture all aspects (since we are computing average some significant features might be diluted) of sentiment.

However, the computation time for training Tf-Idf models is way more than the computation time for word2vec model training.

4. Feedforward Neural Networks

a) Simple FNN unit cell using average word2vec

In [16]:

```
# Defining FNN architecture
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_size2, output_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu1(out)
        out = self.fc2(out)
        out = self.relu2(out)
        out = self.fc3(out)
        return out

input_size = 300 # size of Word2Vec vectors
hidden_size1 = 100 # first layer
hidden_size2 = 10 # second layer
output_size = 3 # number of rating categories

# Defining hyperparamters
learning_rate = 0.001
num_epochs = 50
batch_size = 128
```

In [17]:

```
# Creating input vectors for test and train
X_train = np.array([get_word2vec(review) for review in X_train_main])
X_test = np.array([get_word2vec(review) for review in X_test_main])

# Initialize model
model = MLP(input_size, hidden_size1, hidden_size2, output_size)

# Defining loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Training network
for epoch in range(num_epochs):
    for i in range(0, len(X_train), batch_size):
        # Get batch
        batch_X = torch.FloatTensor(X_train[i:i+batch_size])
        batch_y = torch.LongTensor(y_train[i:i+batch_size].values)

        # Forward pass
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)

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

    # Print loss after every 10th epoch
    if (epoch+1)%10==0:
        print('Epoch [{}/{}], Loss: {}'.format(epoch+1, num_epochs, loss.item()))
```

```
Epoch [10/50], Loss: 0.6190446615219116
Epoch [20/50], Loss: 0.5855628848075867
Epoch [30/50], Loss: 0.5434885621070862
Epoch [40/50], Loss: 0.5073877573013306
Epoch [50/50], Loss: 0.4767744839191437
```

In [18]:

```
# Testing the model
with torch.no_grad():
    # Convert testing set to tensors
    test_X = torch.FloatTensor(X_test)
    test_y = torch.LongTensor(y_test.to_numpy())
    # Predict classes using testing set
    outputs = model(test_X)
    _, predicted = torch.max(outputs.data, 1)
    # Calculate accuracy
    total = test_y.size(0)
    correct = (predicted == test_y).sum().item()
    accuracy = correct / total
    # Print accuracy
    print('Accuracy on FNN (Average word2vec): {}'.format(accuracy))
```

```
Accuracy on FNN (Average word2vec): 0.6546666666666666
```

b) Simple FNN unit cell using first 10 words of a review

In [19]:

```
# generating word2vec for first 10 words by concatenation, setting default to 0's
def get_word2vec_consider_first_10_words(reviews):
    features = np.zeros((len(reviews), 3000))
    for i, review in enumerate(reviews):
        words = review[:10]
        padded_vectors = np.zeros((10, 300))
        vectors = []
        for word in words:
            if word in w2v_google_model:
                vectors.append(w2v_google_model[word])
        if len(vectors) > 0:
            padded_vectors[:len(vectors), :] = vectors
            features[i] = padded_vectors.flatten()
    return features

X_train = get_word2vec_consider_first_10_words(X_train_main)
X_test = get_word2vec_consider_first_10_words(X_test_main)
```

In [20]:
```
input_size = 3000 # size of Word2Vec vectors
hidden_size1 = 100 # first layer
hidden_size2 = 10 # second layer
output_size = 3 # number of rating categories

learning_rate = 0.001
num_epochs = 50
batch_size = 128
```

In [21]:

```
# Initialize model, using same architecture as defined earlier
model = MLP(input_size, hidden_size1, hidden_size2, output_size)

# Defining loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Training model
for epoch in range(num_epochs):
    for i in range(0, len(X_train), batch_size):
        # Get batch
        batch_X = torch.FloatTensor(X_train[i:i+batch_size])
        batch_y = torch.LongTensor(y_train[i:i+batch_size].values)

        # Forward pass
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)

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

    # Print loss after every 10th epoch
    if (epoch+1)%10==0:
        print('Epoch [{}/{}], Loss: {}'.format(epoch+1, num_epochs, loss.item()))
```

```
Epoch [10/50], Loss: 0.3133710026741028
Epoch [20/50], Loss: 0.09786497801542282
Epoch [30/50], Loss: 0.07481760531663895
Epoch [40/50], Loss: 0.05096546560525894
Epoch [50/50], Loss: 0.02985224686563015
```

In [22]:

```
# Evaluating model on testing set
with torch.no_grad():
    # Convert testing set to tensors
    test_X = torch.FloatTensor(X_test)
    test_y = torch.LongTensor(y_test.to_numpy())
    # Predict classes using testing set
    outputs = model(test_X)
    _, predicted = torch.max(outputs.data, 1)

    # Calculate accuracy
    total = test_y.size(0)
    correct = (predicted == test_y).sum().item()
    accuracy = correct / total

    # Print accuracy
    print('Accuracy on FNN (considering first 10 words): {}'.format(accuracy))
```

```
Accuracy on FNN (considering first 10 words): 0.5158333333333334
```

Q. What do you conclude by comparing accuracy values you obtain with those obtained in the “Simple Models” section.

Ans. Comparing the accuracies obtained for the two cases of modeling a simple FNN using average word2vec features and using only the first 10 words of a review, it can be concluded that the FNN model using average word2vec features performed better than the one using only the first 10 words of a review. The accuracy of 0.65 for the FNN model with average word2vec features is higher than the accuracy of 0.515 for the FNN model with only the first 10 words of a review.

When compared to the accuracies obtained for the SVM and perceptron models, it can be seen that the FNN models with average word2vec features and the SVM models with Tf-Idf features have similar accuracies, FNN is slightly better because a multi layer perceptron can learn better than a single perceptron (comples function can be learned by MLP). However, the FNN models with only the first 10 words of a review have lower accuracies. This suggests that using only the first few words of a review may not provide enough information for the models to accurately predict the sentiment of the review.

Overall, it can be concluded that using more comprehensive feature representations, such as average word2vec or Tf-Idf features, can lead to better model performance.

5. Recurrent Neural Networks

a) Considering Simple RNN unit cell

In [23]:

```
# Define fixed review length
max_review_length = 20

# truncate reviews to length 20 if more, otherwise padding with '<PAD>'
def truncate_reviews(reviews):
    truncate_reviews = []
    for review in reviews:
        words = []
        # Truncate or pad review to fixed length
        if len(review) > max_review_length:
            words = review[:max_review_length]
        else:
            words = review + ['<PAD>'] * (max_review_length - len(review))
        truncate_reviews.append(words)
    return truncate_reviews

# Get word to vec features of reviews. Concatenating all word2vec features of 20 words simillar to 4b
def get_word2vec_consider_20_words(reviews):
    features = np.zeros((len(reviews), 6000))
    for i, review in enumerate(reviews):
        words = review
        padded_vectors = np.zeros((20, 300))
        vectors = []
        for word in words:
            if word in w2v_google_model:
                vectors.append(w2v_google_model[word])
        if len(vectors) > 0:
            padded_vectors[:len(vectors), :] = vectors
            features[i] = padded_vectors.flatten()
    return features

processed_reviews_train = truncate_reviews(X_train_main)
processed_reviews_test = truncate_reviews(X_test_main)
X_train = get_word2vec_consider_20_words(processed_reviews_train)
X_test = get_word2vec_consider_20_words(processed_reviews_test)
```

In [24]:
```
# Defining RNN architecture
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, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out

input_size = 6000 # size of word2vec features
hidden_size = 20
output_size = 3 # number of ratings

# Defining hyperparameters
batch_size = 128
learning_rate = 0.001
epochs = 50
```

In [25]:

```
# Initialize the model, loss function, and optimizer
model = RNN(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Training network
for epoch in range(epochs):
    for i in range(0, len(X_train), batch_size):
        # Get batch
        batch_X = torch.FloatTensor(X_train[i:i+batch_size])
        batch_y = torch.LongTensor(y_train[i:i+batch_size].values)

        batch_X = batch_X.reshape(batch_size, -1, input_size)
        # Forward pass
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)

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

    # Print the loss after every 10th epoch
    if (epoch+1)%10==0:
        print('Epoch [{}/{}], Loss: {}'.format(epoch+1, epochs, loss.item()))
```

```
Epoch [10/50], Loss: 0.4484310746192932
Epoch [20/50], Loss: 0.16086478531360626
Epoch [30/50], Loss: 0.18855616450309753
Epoch [40/50], Loss: 0.06513053923845291
Epoch [50/50], Loss: 0.04815263673663139
```

In [26]:
```
# Evaluate the model on the test set
with torch.no_grad():
    h = torch.zeros(1, len(X_test), hidden_size)
    # Convert testing set to tensors
    inputs = torch.tensor(X_test, dtype=torch.float32)
    inputs = inputs.reshape(12000, -1, input_size)
    labels = torch.tensor(y_test.values, dtype=torch.long)
    # Predict classes using testing set
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    _, predicted = torch.max(outputs.data, 1)
    # Calculate accuracy
    accuracy = (predicted == labels).sum().item() / len(labels)
    print('Accuracy on Simple RNN: {}'.format(accuracy))
```

```
Accuracy on Simple RNN: 0.5170833333333333
```

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

Ans. Comparing the accuracies obtained for the FNN models and the Simple RNN model, it can be concluded that the FNN model using average word2vec features performed better than the Simple RNN model considering the first 20 words of a review.

It is worth noting that the Simple RNN model may have been affected by the vanishing gradient problem, which can make it difficult for the model to learn long-term dependencies in the sequence data. In contrast, FNN models are more suitable for shallow learning tasks, where the sequence data may not contain long-term dependencies. Therefore, it is not surprising to see that the FNN model with average word2vec outperforms the Simple RNN model in this case.

Comparing with 3b, since RNN is considering 20 words while FNN is considering only 10 words. RNN has slightly better accuracy than FNN.

Overall, it can be concluded that the FNN model with average word2vec features is a better choice for sentiment analysis tasks than the Simple RNN model considering the first 20 words of a review.

b) Considering Gated recurrent Unit cell

In [27]:

```
# Defining the GRU architecture
class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GRU, self).__init__()
        self.hidden_size = hidden_size
        self.gru = nn.GRU(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        out, _ = self.gru(x, h0)
        out = self.fc(out[:, -1, :])
        return out

input_size = 6000 # size of word2vec features
hidden_size = 20
output_size = 3 # number of ratings

# Defining hyperparameters
batch_size = 128
learning_rate = 0.001
epochs = 50
```

In [28]:

```
# Initializing the model, loss function, and optimizer
model = GRU(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Training network
for epoch in range(epochs):
    for i in range(0, len(X_train), batch_size):
        # Get batch
        batch_X = torch.FloatTensor(X_train[i:i+batch_size])
        batch_y = torch.LongTensor(y_train[i:i+batch_size].values)

        batch_X = batch_X.reshape(batch_size, -1, input_size)
        # Forward pass
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)

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

    # Print the loss after every 10th epoch
    if (epoch+1)%10==0:
        print('Epoch [{}/{}], Loss: {}'.format(epoch+1, epochs, loss.item()))
```

```
Epoch [10/50], Loss: 0.1830853968858719
Epoch [20/50], Loss: 0.04943244531750679
Epoch [30/50], Loss: 0.03577134758234024
Epoch [40/50], Loss: 0.031040778383612633
Epoch [50/50], Loss: 0.025461552664637566
```

In [29]:
```
# Evaluating the model on the test set
with torch.no_grad():
    h = torch.zeros(1, len(X_test), hidden_size)
    # Convert testing set to tensors
    inputs = torch.tensor(X_test, dtype=torch.float32)
    inputs = inputs.reshape(12000, -1, input_size)
    labels = torch.tensor(y_test.values, dtype=torch.long)
    # Predict classes using testing set
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    _, predicted = torch.max(outputs.data, 1)
    # Calculate accuracy
    accuracy = (predicted == labels).sum().item() / len(labels)
    print('Accuracy on GRU: {}'.format(accuracy))
```

```
Accuracy on GRU: 0.5400833333333334
```

c) Considering an LSTM unit cell

In [30]:

```
# Defining the LSTM architecture
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        c0 = torch.zeros(1, x.size(0), self.hidden_size)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out

input_size = 6000 # size of word2vec features
hidden_size = 20
output_size = 3 # number of ratings

# Defining hyperparameters
batch_size = 128
learning_rate = 0.001
epochs = 50
```

In [31]:

```
# Initializing the model, loss function, and optimizer
model = LSTM(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Training loop
for epoch in range(epochs):
    for i in range(0, len(X_train), batch_size):
        # Get batch
        batch_X = torch.FloatTensor(X_train[i:i+batch_size])
        batch_y = torch.LongTensor(y_train[i:i+batch_size].values)

        batch_X = batch_X.reshape(batch_size, -1, input_size)
        # Forward pass
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)

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

    # Print the loss after every 10th epoch
    if (epoch+1)%10==0:
        print('Epoch [{}/{}], Loss: {}'.format(epoch+1, epochs, loss.item()))
```

```
Epoch [10/50], Loss: 0.1725163757801056
Epoch [20/50], Loss: 0.04603668302297592
Epoch [30/50], Loss: 0.030680838972330093
Epoch [40/50], Loss: 0.030271146446466446
Epoch [50/50], Loss: 0.028017841279506683
```

In [32]:
```
# Evaluating the model on the test set
with torch.no_grad():
    h = torch.zeros(1, len(X_test), hidden_size)
    c = torch.zeros(1, len(X_test), hidden_size)
    # Convert testing set to tensors
    inputs = torch.tensor(X_test, dtype=torch.float32)
    inputs = inputs.reshape(len(X_test), -1, input_size)
    labels = torch.tensor(y_test.values, dtype=torch.long)
    # Predict classes using testing set
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    _, predicted = torch.max(outputs.data, 1)
    # Calculate accuracy
    accuracy = (predicted == labels).sum().item() / len(labels)
    print('Accuracy on LSTM: {}'.format(accuracy))
```

```
Accuracy on LSTM: 0.5415833333333333
```

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

Ans. Based on the accuracy values, it appears that the LSTM model performs slightly better than the GRU and Simple RNN models in terms of accuracy. The LSTM model achieved an accuracy of 0.541, while the GRU model achieved an accuracy of 0.540 and the Simple RNN model achieved an accuracy of 0.517.

It is possible that the LSTM's ability to remember longer-term dependencies within the sequence of words in each review may be helping it to better capture the sentiment expressed in each review. In this particular case, the LSTM model appears to be better suited for the sentiment analysis task, and this may be due to its ability to capture longer-term dependencies and selectively remember relevant information.