Imports

```
In [1]: import pandas as pd
        import numpy as np
        import nltk
        dl = nltk.downloader.Downloader('https://www.nltk.org/nltk_data/')
        dl.download('wordnet')
        import re
        from bs4 import BeautifulSoup

        from sklearn.model_selection import train_test_split
        from numpy.linalg import norm

        from gensim import utils
        import gensim.downloader
```

```
[nltk_data] Downloading package wordnet to /home/naseela/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

1. Dataset Generation

Since we are just using the **review_body** and **star_rating** for this assignment, I am reading just those columns. I have also renamed the classes as 0,1,2 for simplification in the Pytorch models

```
In [2]: data = pd.read_table('data.tsv', on_bad_lines = 'skip',verbose = Fals
        e,usecols=['review_body','star_rating'])
```

```
/home/naseela/anaconda3/lib/python3.9/site-packages/IPython/core/inte
ractiveshell.py:3444: DtypeWarning: Columns (7) have mixed types.Spec
ify dtype option on import or set low_memory=False.
  exec(code_obj, self.user_global_ns, self.user_ns)
```

```
In [3]: new_data = data[['review_body','star_rating']]
```

```
In [4]: new_data = new_data.dropna()
```

In [5]:
```
# Changing classes to the required 3 classes
classes = []
star_ratings = new_data['star_rating'].to_list()

for rating in star_ratings:

  if int(rating) == 1 or int(rating) == 2:
    classes.append(0)
  elif int(rating) == 3:
    classes.append(1)
  else:
    classes.append(2)

new_data['class'] = classes
```

In [6]:
```
# Sampling from the classes
random_sample_class1 = new_data[new_data['class']==0].sample(n=20000,
replace=False)
random_sample_class2 = new_data[new_data['class']==1].sample(n=20000,
replace=False)
random_sample_class3 = new_data[new_data['class']==2].sample(n=20000,
replace=False)

class_samples = [random_sample_class1,random_sample_class2,random_sam
ple_class3]

df_class = pd.concat(class_samples)
```

As mentioned in the assignment, the train-test split is 80/20. I am following the same

In [7]:
```
# Train test split

X = df_class['review_body']

y = df_class['class']

X_train,X_test,y_train,y_test = train_test_split(X,y,test_size = 0.2)
```

2. Word Embedding (25 points)

(a) (5 points)

Load the pretrained “word2vec-google-news-300” Word2Vec model and learn how to extract word embeddings for your dataset. Try to check semantic similarities of the generated vectors using three examples of your own, e.g., King − M an + Woman = Queen or excellent ∼ outstanding.

For this part, I have created a function, that extracts the embeddings if the word is in the vocab, otherwise it returns a default vector.

In [8]:
```
# function to get the vector for a word
def get_vector(model,word):
    default = [0.0] * model.vector_size
    try:
        return model[word]
    except KeyError:
        return default
```

In [9]:
```
w2v_g300 = gensim.downloader.load('word2vec-google-news-300')
```

Example 1: amazing = awesome

In [10]:
```
vec_amazing, vec_awesome= w2v_g300['amazing'],w2v_g300['awesome']


# Reference = https://www.geeksforgeeks.org/how-to-calculate-cosine-s
imilarity-in-python/
print('Cosine similarity between "amazing" and "awesome" = ',np.dot(v
ec_amazing,vec_awesome)/(norm(vec_amazing)*norm(vec_awesome)))
```

```
Cosine similarity between "amazing" and "awesome" =  0.8282866
```

Example 2: very angry = upset

In [11]:
```
vec_very, vec_angry, vec_furious = w2v_g300['very'],w2v_g300['angry'
],w2v_g300['upset']

lhs = (vec_very + vec_angry)/2

# Reference = https://www.geeksforgeeks.org/how-to-calculate-cosine-s
imilarity-in-python/
print('Cosine similarity between "very angry" and "upset" = ',np.dot(
lhs,vec_furious)/(norm(lhs)*norm(vec_furious)))
```

```
Cosine similarity between "very angry" and "upset" =  0.5078693
```

Example 3: pretty = ugly

In [12]:
```
vec_pretty, vec_ugly = w2v_g300['pretty'],w2v_g300['ugly']


lhs = vec_pretty

# Reference = https://www.geeksforgeeks.org/how-to-calculate-cosine-s
imilarity-in-python/
print('Cosine similarity between "pretty" and "ugly" = ',np.dot(lhs,v
ec_ugly)/(norm(lhs)*norm(vec_ugly)))
```

```
Cosine similarity between "pretty" and "ugly" =  0.2727458
```

(b) (20 points)

Train a Word2Vec model using your own dataset

In [13]:
```
reviews = df_class.review_body.apply(lambda x: [w for w in x.split
()])
```

In [14]:
```
model = gensim.models.Word2Vec(sentences = reviews, vector_size = 300
, window = 13,min_count=9)
```

In [15]:
```
vec_amazing, vec_awesome= model.wv['amazing'],model.wv['awesome']

lhs = vec_amazing
rhs =  vec_awesome

# Reference = https://www.geeksforgeeks.org/how-to-calculate-cosine-s
imilarity-in-python/
print('Cosine similarity between "amazing" and "awesome" = ',np.dot(l
hs,rhs)/(norm(lhs)*norm(rhs)))
```

```
Cosine similarity between "amazing" and "awesome" =  0.81626076
```

In [16]:
```
vec_very, vec_angry, vec_furious = model.wv['very'],model.wv['angry'
], model.wv['upset']

lhs = (vec_very + vec_angry)/2

# Reference = https://www.geeksforgeeks.org/how-to-calculate-cosine-s
imilarity-in-python/
print('Cosine similarity between "very angry" and "upset" = ',np.dot(
lhs,vec_furious)/(norm(lhs)*norm(vec_furious)))
```

```
Cosine similarity between "very angry" and "upset" =  0.016726496
```

In [17]:
```
vec_pretty, vec_ugly = model.wv['pretty'],model.wv['ugly']

lhs = vec_pretty

# Reference = https://www.geeksforgeeks.org/how-to-calculate-cosine-s
imilarity-in-python/
print('Cosine similarity between "pretty" and "ugly" = ',np.dot(lhs,v
ec_ugly)/(norm(lhs)*norm(vec_ugly)))
```

```
Cosine similarity between "pretty" and "ugly" =  0.3070316
```

```
 (a). 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?
```

From the results above, I noticed that the results of the model trained on my dataset performs worse than the Google-300 pre-trained model. For calculating the semantic similarities, I have used the cosine distance between the vectors. The vector of one sentence is taken using the mean of all the word2vec embeddings.

In conclusion, the pretrained model performs better than the model I have trained. This is expected as the pretraiend model has been trained on a larger corpus. The Google-300 pretrained model encodes the semantic similarities between words better than our model.

3. Simple models (20 points)

In [18]:
```
def get_features(df):
    features = []
    reviews = df.apply(lambda x: [w for w in x.split()])
    for review in reviews:
        rev_vec = []
        for w in review:

            if w  in w2v_g300:
                rev_vec.append(w2v_g300[w])
        rev_vec = np.array(rev_vec)
        features.append(np.mean(rev_vec, axis =0))

#     features = np.array(features)
    return features
X_train_w2v, X_test_w2v = get_features(X_train),get_features(X_test)
```

```
/home/naseela/anaconda3/lib/python3.9/site-packages/numpy/core/fromnu
meric.py:3419: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/home/naseela/anaconda3/lib/python3.9/site-packages/numpy/core/_metho
ds.py:188: RuntimeWarning: invalid value encountered in double_scalar
s
  ret = ret.dtype.type(ret / rcount)
```

Perceptron

In [19]:
```
# train_features = pd.DataFrame(X_train_w2v)

train_features = pd.DataFrame(list(map(np.ravel, X_train_w2v)))
train_features = train_features.fillna(0)
test_features = pd.DataFrame(list(map(np.ravel, X_test_w2v)))
test_features = test_features.fillna(0)



from sklearn.linear_model import Perceptron

perceptron_classifier = Perceptron(warm_start=True)

perceptron_classifier.fit(train_features,y_train)
```

Out[19]:
```
▾          Perceptron
Perceptron(warm_start=True)
```

In [20]:
```
preds = perceptron_classifier.predict(test_features)

from sklearn.metrics import accuracy_score

print("Accuracy of perceptron is: ",accuracy_score(y_test, preds))
```

```
Accuracy of perceptron is:  0.5688333333333333
```

Accuracy value of perceptron using TF-IDF for feature extraction[HW-1] : 0.59 - 0.63

SVM

In [21]:
```
from sklearn.svm import LinearSVC
svc = LinearSVC()
svc.fit(train_features,y_train)
```

Out[21]:
```
▾ LinearSVC
LinearSVC()
```

In [22]:
```
preds = svc.predict(test_features)

print("Accuracy of SVM is:",accuracy_score(y_test, preds))
```

```
Accuracy of SVM is: 0.63475
Accuracy value of SVM using TF-IDF for feature extraction[HW-1]: 0.67
```

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

Looking at the accuracy of the values on Perceptron as well as SVM, we can clearly see that using Word2Vec features perform at par with than how they performed using the TF-IDF features.

During experimentation, sometimes Word2Vec performs a little better than TF-IDF and vice versa. The reason for this can be many, the data subset both the features get, pre-processing etc

However, SVM performs better than Perceptron using TF-IDF as well as Word2Vec features

4. Feedforward Neural Networks (25 points)

(a) (10 points) To generate the input features, use the average Word2Vec vectors similar to the "Simple models" section and train the neural network. Report accuracy values on the testing split for your MLP.

Imports

In [23]:
```
import torch
from torch.utils.data import DataLoader,Dataset
import torchvision
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
```

Create DataLoaders

Both the train dataloader as well as the test dataloader return the tensors of the features and labels

In [24]:
```
class TrainDataset(Dataset):
    def __init__(self,train_features,labels):

        self.train_features = train_features


        self.labels = labels
    def __len__(self):
        return len(self.train_features)
    def __getitem__(self,index):

#         return torch.tensor(self.train_features[index]), torch.nn.f
unctional.one_hot(torch.tensor(self.labels[index]),num_classes = 3)
        return torch.tensor(self.train_features[index]), torch.tensor
(self.labels[index])

class TestDataset(Dataset):
    def __init__(self,test_features,labels):

        self.test_features = test_features
        self.labels = labels

    def __len__(self):
        return len(self.test_features)
    def __getitem__(self,index):

        return torch.tensor(self.test_features[index]), torch.tensor(
self.labels[index])
```

For creating train and test data we are using the features that we used for Simple models.

In [25]:
```
train_data = TrainDataset(train_features.values, y_train.values)
test_data = TestDataset(test_features.values,y_test.values)
```

After experimenting,the batch_size of 16 gives the best performance. So while creating the dataloaders, I used a batch_size of 16

In [26]:
```
num_workers = 0
# how many samples per batch to load
batch_size = 16



# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=bat
ch_size,num_workers=num_workers)

test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch
_size,
    num_workers=num_workers)
```

Architecture

I have used the activation as relu. I have also used dropout to prevent overfitting of the data

In [27]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the NN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        hidden_1 = 100
        hidden_2 = 10

        self.fc1 = nn.Linear(300, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 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 = self.fc3(x)

        return x

model = Net()
print(model)
```

```
Net(
  (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)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

CrossEntropyLoss performs well on multiclass classificaion. I have used the optimizer to be Adam with a learning rate of 0.01. I tried SGD optimizer as well but this performs the best.

After experimenting, I used the number of epochs to be 20

In [28]:
```
criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [29]:
```python
n_epochs = 20



for epoch in range(n_epochs):

    train_loss = 0.0


    model.train()
    for data, target in train_loader:
        optimizer.zero_grad()
        output = model(data.float())

        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)


    train_loss = train_loss/len(train_loader.dataset)


    print('Epoch: {} \tTraining Loss: {:.6f} '.format(
        epoch+1,
        train_loss
        ))
```

```
Epoch: 1 	Training Loss: 0.909088
Epoch: 2 	Training Loss: 0.856726
Epoch: 3 	Training Loss: 0.839670
Epoch: 4 	Training Loss: 0.829170
Epoch: 5 	Training Loss: 0.818103
Epoch: 6 	Training Loss: 0.811943
Epoch: 7 	Training Loss: 0.804650
Epoch: 8 	Training Loss: 0.798262
Epoch: 9 	Training Loss: 0.791184
Epoch: 10 	Training Loss: 0.787318
Epoch: 11 	Training Loss: 0.783119
Epoch: 12 	Training Loss: 0.777745
Epoch: 13 	Training Loss: 0.771641
Epoch: 14 	Training Loss: 0.768381
Epoch: 15 	Training Loss: 0.764289
Epoch: 16 	Training Loss: 0.762391
Epoch: 17 	Training Loss: 0.757201
Epoch: 18 	Training Loss: 0.754969
Epoch: 19 	Training Loss: 0.751023
Epoch: 20 	Training Loss: 0.746655
```

In [30]:
```
with torch.no_grad():
    correct = 0
    total = 0
    for x,y in test_loader:
        outputs = model(x.float())
        _, predicted = torch.max(outputs.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()


    print('Test Accuracy of the FNN on the average word2vec: {} %'.fo
rmat(100 * correct / total))
```

```
Test Accuracy of the FNN on the average word2vec: 63.24166666666667 %
```

(b) (15 points) To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature (x = [WT 1 , ..., WT 10]) and train the neural network. Report the accuracy value on the testing split for your MLP model. What do you conclude by comparing accuracy values you obtain with those obtained in the “’Simple Models” section.

I have modified the get_features function previously to limit the words to 10 words

In [31]:
```
def get_features_FNN(df):

    features = []
    reviews = df.apply(lambda x: [w for w in x.split()])

    for review in reviews:
        rev_vec = []
        for w in review:

            if w  in w2v_g300:
                rev_vec.append(w2v_g300[w])
            if len(rev_vec) == 10:
              break

        if len(rev_vec) < 10:
            for i in range(10-len(rev_vec)):
                rev_vec.append([0]*300)

        features.append(np.concatenate(rev_vec))

    return features
X_train_w2v, X_test_w2v = get_features_FNN(X_train),get_features_FNN(
X_test)
```

In [32]:
```
train_data = TrainDataset(X_train_w2v, y_train.values)
test_data = TestDataset(X_test_w2v,y_test.values)
```

In [33]:
```
num_workers = 0
batch_size = 16


train_loader = torch.utils.data.DataLoader(train_data, batch_size=bat
ch_size,num_workers=num_workers)

test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch
_size,
    num_workers=num_workers)
```

The model architecture is similar, just the input length has changed

In [34]:
```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        hidden_1 = 100
        hidden_2 = 10

        self.fc1 = nn.Linear(3000, hidden_1)

        self.fc2 = nn.Linear(hidden_1, hidden_2)

        self.fc3 = nn.Linear(hidden_2, 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 = self.fc3(x)

        return x

model = Net()
print(model)
```

```
Net(
  (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)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [35]:
```
criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [36]:
```
n_epochs = 20


for epoch in range(n_epochs):

    train_loss = 0.0



    model.train()
    for data, target in train_loader:
        optimizer.zero_grad()
        output = model(data.float())


        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)




    train_loss = train_loss/len(train_loader.dataset)

    print('Epoch: {} \tTraining Loss: {:.6f}'.format(
        epoch+1,
        train_loss
        ))
```

```
Epoch: 1        Training Loss: 0.964631
Epoch: 2        Training Loss: 0.890520
Epoch: 3        Training Loss: 0.834062
Epoch: 4        Training Loss: 0.768162
Epoch: 5        Training Loss: 0.705709
Epoch: 6        Training Loss: 0.643937
Epoch: 7        Training Loss: 0.596140
Epoch: 8        Training Loss: 0.555489
Epoch: 9        Training Loss: 0.514732
Epoch: 10       Training Loss: 0.480082
Epoch: 11       Training Loss: 0.457447
Epoch: 12       Training Loss: 0.429844
Epoch: 13       Training Loss: 0.412119
Epoch: 14       Training Loss: 0.386939
Epoch: 15       Training Loss: 0.370552
Epoch: 16       Training Loss: 0.354751
Epoch: 17       Training Loss: 0.344845
Epoch: 18       Training Loss: 0.331737
Epoch: 19       Training Loss: 0.320827
Epoch: 20       Training Loss: 0.305609
```

In [37]:
```
with torch.no_grad():
    correct = 0
    total = 0
    for x,y in test_loader:
        outputs = model(x.float())
        _, predicted = torch.max(outputs.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()


    print('Test Accuracy of the FNN model on the first 10 word2vec:
{} %'.format(100 * correct / total))
```

```
Test Accuracy of the FNN model on the first 10 word2vec: 52.341666666
66667 %
```

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

Looking at the accuracy of Forward Neural Network with the simple models, I can conclude the following from my results:

1. On the average Word2Vec features, the Feed Forward Neural Network gives a performance boost,
2. Considering, just the first 10 words, the FNN model does not perform good. This is in accordance with the fact that we consider just the first few words. For different subsets of the data - the accuracy ranges in 50-60%

5. Recurrent Neural Networks (30 points)

In [38]:
```
class TrainDataset(Dataset):
    def __init__(self,train_features,labels):

        self.train_features = train_features



        self.labels = labels
    def __len__(self):
        return len(self.train_features)
    def __getitem__(self,index):

        return torch.tensor(self.train_features[index]), torch.tensor
(self.labels[index])

class TestDataset(Dataset):
    def __init__(self,test_features,labels):

        self.test_features = test_features
        self.labels = labels

    def __len__(self):
        return len(self.test_features)
    def __getitem__(self,index):

        return torch.tensor(self.test_features[index]), torch.tensor(
self.labels[index])
```

In [39]:
```
from gensim import utils
def get_features_RNN(df):

    features = []
    reviews = df.apply(lambda x: [w for w in x.split()])

    for review in reviews:
        rev_vec = []
        for w in review:

            if w  in w2v_g300:
                rev_vec.append(w2v_g300[w])
            if len(rev_vec) == 20:
              break

        if len(rev_vec) < 20:
            for i in range(20-len(rev_vec)):
                rev_vec.append([0]*300)

        features.append(rev_vec)

    return features
X_train_w2v, X_test_w2v = get_features_RNN(X_train),get_features_RNN(
X_test)
```

In [40]:
```
train_data = TrainDataset(X_train_w2v, y_train.values)
test_data = TestDataset(X_test_w2v,y_test.values)
```

In [41]:
```
num_workers = 0
batch_size = 32

train_loader = torch.utils.data.DataLoader(train_data, batch_size=bat
ch_size,num_workers=num_workers)

test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch
_size,
    num_workers=num_workers)
```

In [42]:
```
class MyRNNModule(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, n_classes
):
        super(MyRNNModule, self).__init__()
        self.rnn = torch.nn.RNN(input_size, hidden_size, num_layers,
batch_first=True)
        self.fc = torch.nn.Linear(hidden_size, n_classes)

    def forward(self, x):
        _, h = self.rnn(x)
        h  = h.squeeze(0)
        out = self.fc(h)
        return out
```

In [43]:
```
model = MyRNNModule(input_size=300, hidden_size=20, num_layers=1, n_c
lasses=3)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=32,
shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=32, s
huffle=True)
```

In [44]:
```
for epoch in range(20):

    for x, y in train_loader:
        # Forward pass
        outputs = model(x)
        loss = criterion(outputs, y)

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

    print ('Epoch {}, Loss: {:.4f}'
                     .format(epoch+1, loss.item()))
```

```
/tmp/ipykernel_46810/3896146456.py:12: UserWarning: Creating a tensor
from a list of numpy.ndarrays is extremely slow. Please consider conv
erting the list to a single numpy.ndarray with numpy.array() before c
onverting to a tensor. (Triggered internally at  ../torch/csrc/utils/
tensor_new.cpp:201.)
  return torch.tensor(self.train_features[index]), torch.tensor(self.
labels[index])
```

```
Epoch 1, Loss: 0.9017
Epoch 2, Loss: 0.8870
Epoch 3, Loss: 0.9140
Epoch 4, Loss: 0.7937
Epoch 5, Loss: 0.7175
Epoch 6, Loss: 1.0559
Epoch 7, Loss: 0.9040
Epoch 8, Loss: 0.8322
Epoch 9, Loss: 1.1945
Epoch 10, Loss: 0.9748
Epoch 11, Loss: 0.7747
Epoch 12, Loss: 0.9769
Epoch 13, Loss: 0.8258
Epoch 14, Loss: 0.7168
Epoch 15, Loss: 1.0199
Epoch 16, Loss: 0.8571
Epoch 17, Loss: 0.6332
Epoch 18, Loss: 0.8812
Epoch 19, Loss: 0.7856
Epoch 20, Loss: 0.9537
```

In [45]:
```
# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for x,y in test_loader:
        outputs = model(x)
        _, predicted = torch.max(outputs.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()


    print('Test Accuracy of the RNN model: {} %'.format(100 * correct
/ total))
```

```
Test Accuracy of the RNN model: 59.725 %
```

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

By comapring the accuracy values of RNN with FNN, there are two points to be noted:

1. The performance of RNN as comapared to FNN with the average Word2Vec is less since the accuarcy value of FNN trained on average Word2Vec is more than RNN.
2. The performance of RNN when compared to FNN with the first 10 Word2Vec concatenated is bettey since the accuarcy value of FNN trained on first 10 concatenated Word2Vec is less than RNN trained on first 20 concatenated Word2Vec.

(b) (10 points) Repeat part (a) by considering a gated recurrent unit cell.

In [46]:
```
class MyGRUModule(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, n_classes
):
        super(MyGRUModule, self).__init__()
        self.gru = torch.nn.GRU(input_size, hidden_size, num_layers,
batch_first=True)
        self.fc = torch.nn.Linear(hidden_size, n_classes)

    def forward(self, x):
        _, h = self.gru(x)
        h  = h.squeeze(0)
        out = self.fc(h)
        return out
```

In [47]:
```
model = MyGRUModule(input_size=300, hidden_size=20, num_layers=1, n_c
lasses=3)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=32,
shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=32, s
huffle=True)
```

In [48]:
```
for epoch in range(20):

    for x, y in train_loader:
        # Forward pass
        outputs = model(x)
        loss = criterion(outputs, y)

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

    print ('Epoch {}Loss: {:.4f}'
                     .format(epoch+1, loss.item()))
```

```
Epoch 1Loss: 0.8753
Epoch 2Loss: 0.8942
Epoch 3Loss: 0.8456
Epoch 4Loss: 0.5885
Epoch 5Loss: 0.6874
Epoch 6Loss: 0.7712
Epoch 7Loss: 0.5330
Epoch 8Loss: 0.6769
Epoch 9Loss: 0.7032
Epoch 10Loss: 0.5011
Epoch 11Loss: 0.8105
Epoch 12Loss: 0.6409
Epoch 13Loss: 0.5740
Epoch 14Loss: 0.7851
Epoch 15Loss: 0.6700
Epoch 16Loss: 0.8148
Epoch 17Loss: 0.5101
Epoch 18Loss: 1.0453
Epoch 19Loss: 0.6948
Epoch 20Loss: 0.6510
```

In [49]:
```
# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for x,y in test_loader:
        outputs = model(x)
        _, predicted = torch.max(outputs.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()
#          correct += (list(int(p) for p in predicted) == y).sum().ite
m()


    print('Test Accuracy of the GRU model: {} %'.format(100 * correct
/ total))
```

```
Test Accuracy of the GRU model: 64.7 %
```

Repeat part (a) by considering an LSTM unit cell.

In [50]:
```
class MyLSTMModule(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, n_classes
):
        super(MyLSTMModule, self).__init__()
        self.lstm = torch.nn.LSTM(input_size, hidden_size, num_layers
, batch_first=True)
        self.fc = torch.nn.Linear(hidden_size, n_classes)

    def forward(self, x):
        _, h = self.lstm(x)
        h  = h[1].squeeze(0)
        out = self.fc(h)
        return out
```

In [51]:
```
model = MyLSTMModule(input_size=300, hidden_size=20, num_layers=1, n_
classes=3)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=32,
shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=32, s
huffle=True)
```

In [52]:
```
for epoch in range(20):

    for x, y in train_loader:
        # Forward pass
        outputs = model(x)
        loss = criterion(outputs, y)

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

    print ('Epoch {}, Loss: {:.4f}'
                   .format(epoch+1, loss.item()))
```

```
Epoch 1, Loss: 0.8822
Epoch 2, Loss: 1.0440
Epoch 3, Loss: 0.6420
Epoch 4, Loss: 0.8885
Epoch 5, Loss: 0.6598
Epoch 6, Loss: 0.8232
Epoch 7, Loss: 0.8359
Epoch 8, Loss: 0.7274
Epoch 9, Loss: 0.7394
Epoch 10, Loss: 0.5944
Epoch 11, Loss: 0.7048
Epoch 12, Loss: 0.7006
Epoch 13, Loss: 0.7930
Epoch 14, Loss: 0.5196
Epoch 15, Loss: 0.6895
Epoch 16, Loss: 0.4376
Epoch 17, Loss: 0.8017
Epoch 18, Loss: 0.7053
Epoch 19, Loss: 0.4792
Epoch 20, Loss: 0.6742
```

In [53]:
```
# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for x,y in test_loader:
        outputs = model(x)
        _, predicted = torch.max(outputs.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()
#          correct += (list(int(p) for p in predicted) == y).sum().ite
m()


    print('Test Accuracy of the LSTM model: {} %'.format(100 * correc
t / total))
```

```
Test Accuracy of the LSTM model: 64.89166666666667 %
```

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

The following conclusions can be made by comparing the accuarcy values obtained by GRU, LSTM and simple RNN:

- GRU and LSTM perform far better than RNN
- The performance of GRU and LSTM is comaparable although LSTM performs a bit better than GRU