CSCI-544 HOMEWORK 3

Name: Vorapoom Thirapatarapong

In [1]:

```
# packages
import pandas as pd
import numpy as np
import time
import copy
import nltk
nltk.download('wordnet')
nltk.download('omw-1.4')
nltk.download('averaged_perceptron_tagger')
import re
from bs4 import BeautifulSoup
import contractions
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
import gensim.downloader as api
import gensim.models
from gensim.test.utils import datapath
from gensim import utils
from numpy import dot
from numpy.linalg import norm
import torch
import torch.utils.data as data_utils
import torch.nn as nn
import torch.nn.functional as F
```

```
/opt/anaconda3/lib/python3.8/site-packages/scipy/__init__.py:138: UserWarning:
A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy (de
tected version 1.24.1)
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion} is re
quired for this version of "
[nltk_data] Downloading package wordnet to /Users/boom/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /Users/boom/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /Users/boom/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
```

In [2]:

```
# parameters
r_state = 555
sample_size = 20000
test_ratio = 0.2
torch.manual_seed(r_state)
```

Out[2]:

```
<torch._C.Generator at 0x7f820f478cf0>
```

1. Dataset Generation

Load data + Initial cleaning

In [3]:
```
# read dataset
df = pd.read_csv('data.tsv', sep='\t', on_bad_lines='skip')
```

```
<ipython-input-3-f74a6e2e7113>:2: DtypeWarning: Columns (7) have mixed types.
Specify dtype option on import or set low_memory=False.
  df = pd.read_csv('data.tsv', sep='\t', on_bad_lines='skip')
```

In [4]:
```
# clean invalid rows & format data types
df = df.loc[:, ['review_body', 'star_rating']]
df['star_rating'] = pd.to_numeric(df['star_rating'], errors='coerce')
df = df[~df['star_rating'].isna()]
df = df[~df['review_body'].isna()]
df['star_rating'] = df['star_rating'].astype(int)
```

In [5]:
```
# columns selection
df = df.loc[:, ['review_body', 'star_rating']]
```

In [6]:
```
# drop duplicates
df = df.drop_duplicates()
```

In [7]:
```
# group ratings to 3 classes
mapping = {1: 0, 2: 0, 3: 1, 4: 2, 5: 2}
df = df.replace({'star_rating': mapping})
```

In [8]:
```
# save a copy of the ,000 sample reviews for training custom Word2Vec
list_w2v = [df[df.star_rating == 0].sample(n=20000, random_state=r_state),
            df[df.star_rating == 1].sample(n=20000, random_state=r_state),
            df[df.star_rating == 2].sample(n=20000, random_state=r_state)]
df_w2v = pd.concat(list_w2v)['review_body']
```

In [9]:
```
# sample a balanced dataset of 60k reviews
list_sample = [df[df.star_rating == 0].sample(n=sample_size, random_state=r_s
            df[df.star_rating == 1].sample(n=sample_size, random_state=r_stat
            df[df.star_rating == 2].sample(n=sample_size, random_state=r_stat
df_sample = pd.concat(list_sample)
```

In [10]:
```
# columns renaming
df_sample.columns = ['review', 'stars']
```

In [11]:
```
# verify number of samples
print(df_sample.groupby('stars').count())
```

```
       review
stars
0       20000
1       20000
2       20000
```

Text cleaning

In [12]:
```
def remove_urls(text):
    return re.sub(r'''(?i)\b((?:https?://|www\d{0,3}[.]|[a-z0-9.\-]+[.][a-z]{

def perform_contractions(text):
    return ' '.join([contractions.fix(word) for word in text.split()])

def remove_non_alpha_chars(text):
    return re.sub(r'[^a-zA-Z0-9\s]', ' ', text)

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

In [13]:
```
df_clean = df_sample.copy()

# lowercase
df_clean['review'] = df_clean['review'].apply(str.lower)

# remove HTML
df_clean['review'] = df_clean['review'].str.replace(r'<[^<>]*>', '', regex=Tr

# remove URLs
df_clean['review'] = df_clean['review'].apply(remove_urls)

# contractions
df_clean['review'] = df_clean['review'].apply(perform_contractions)

# non-alphabet chars
df_clean['review'] = df_clean['review'].apply(remove_non_alpha_chars)

# extra spaces
df_clean['review'] = df_clean['review'].apply(remove_extra_spaces)
```

Text Preprocessing

In [14]:
```
def remove_stop_words(text):
    return ' '.join([word for word in text.split() if word not in stop_words]

def perform_lemmatization(lemmatizer, text):
    lemmatized_list = []
    for word, pos_tag in nltk.pos_tag(text.split()):
        if pos_tag.startswith('V'):
            lemmatized_list.append(lemmatizer.lemmatize(word, 'v'))
        elif pos_tag.startswith('J'):
            lemmatized_list.append(lemmatizer.lemmatize(word, 'a'))
        else:
            lemmatized_list.append(lemmatizer.lemmatize(word))
    return ' '.join(lemmatized_list)
```

In [15]:
```
# remove stop words
stop_words = set(stopwords.words('english'))

df_preproc = df_clean.copy()
df_preproc['review'] = df_preproc['review'].apply(remove_stop_words)

# lemmatization
lemmatizer = WordNetLemmatizer()
```

```
df_lemma = df_preproc.copy()
df_lemma['review'] = df_lemma['review'].apply(lambda x: perform_lemmatization
```

Split Training / Testing

In [16]:

```
# define X and y
df_X = df_lemma['review'].reset_index(drop=True)
df_y = df_lemma['stars'].reset_index(drop=True)

# split training/testing = 80/20
X_train, X_test, y_train, y_test = train_test_split(df_X, df_y, test_size=tes
```

TF-IDF features extraction

In [17]:

```
vectorizer = TfidfVectorizer(max_features=10000)

df_X_tfidf_pre = pd.concat([X_train, X_test])
tfidf = vectorizer.fit_transform(df_X_tfidf_pre)
df_X_tfidf = pd.DataFrame(tfidf.toarray(), columns=vectorizer.get_feature_nam

X_train_tfidf = df_X_tfidf[:int(3*sample_size*(1-test_ratio))]
X_test_tfidf = df_X_tfidf[int(3*sample_size*(1-test_ratio)):]
```

2. Word Embedding

2(a). Pre-trained Word2Vec

In [18]:

```
wv = api.load('word2vec-google-news-300')
```

In [19]:

```
# Semantics similarity example #1

a = wv['boat'] - wv['water'] + wv['air']
b = wv['plane']

cos_sim = dot(a, b)/(norm(a)*norm(b))
print(cos_sim)
```

```
0.5111032
```

In [20]:

```
# Semantics similarity example #2

a = wv['sea']
b = wv['ocean']

cos_sim = dot(a, b)/(norm(a)*norm(b))
print(cos_sim)
```

```
0.76435417
```

In [21]:

```
# Semantics similarity example #3

a = wv['vegetable']
b = wv['star']
```

```
cos_sim = dot(a, b)/(norm(a)*norm(b))
print(cos_sim)
```

```
0.023547666
```

2(b). Custom Word2Vec

In [22]:
```
# create a corpus using 500,000 reviews
class MyCorpus:
    """An iterator that yields sentences (lists of str)."""

    def __iter__(self):
        for index, row in df_w2v.iteritems():
#             print(row)
            yield utils.simple_preprocess(row)
```

In [23]:
```
sentences = MyCorpus()
model = gensim.models.Word2Vec(sentences, min_count=9, vector_size=300, windo
```

```
<ipython-input-22-2c0d12710906>:6: FutureWarning: iteritems is deprecated and
will be removed in a future version. Use .items instead.
  for index, row in df_w2v.iteritems():
```

In [24]:
```
# Semantics similarity example #1

a = model.wv['boat'] - model.wv['water'] + model.wv['air']
b = model.wv['plane']

cos_sim = dot(a, b)/(norm(a)*norm(b))
print(cos_sim)
```

```
-0.046374116
```

In [25]:
```
# Semantics similarity example #2

a = model.wv['sea']
b = model.wv['ocean']

cos_sim = dot(a, b)/(norm(a)*norm(b))
print(cos_sim)
```

```
0.59367234
```

In [26]:
```
# Semantics similarity example #3

a = model.wv['vegetable']
b = model.wv['star']

cos_sim = dot(a, b)/(norm(a)*norm(b))
print(cos_sim)
```

```
0.09536163
```

> What do you conclude from comparing vectors generated by yourself and the pretrained model?
> The dimensions of the vectors from both pretrained(word2vec-google-news-300) and custom Word2Vec models are the same which is 300. However, the pretrained model is able to encode the semantics similarity much better -

> when looking at two similar words, the pretrained model is able to give higher similarity score & when looking at 2 words with no relationship at all, the pretrained model is able to give a slightly lower similarity score. This might be because that the traning samples of the pretrained model is much larger which makes it generalizes better.
>
> Which of the Word2Vec models seems to encode semantic similarities between words better?
> From the 3 examples chosen, it seems like the pretrained Word2Vec model is able to encode the semantic similarities better. For the first 2 examples, the two objects are highly related which should give higher similarity value, and the pretrained model is giving higher similarities. For the third example which is the two objects that are not related, both models seem to satisfy the condition where they give relatively low similarity scores between the two objects. The reason for better semantic similarity encoding for the pretrained model might be a much larger size of training samples.

3. Simple Models: Single perceptron & SVM

In [27]:

```
# Functions for W2V Feature extraction
def extract_features_w2v(data, max_vec_concat, max_vec_rnn):
    data = list(data)
    list_output_avg = list()
    list_output_concat = list()
    df_output_concat = pd.DataFrame(columns=['c'+str(i) for i in range(1, 300
    np_output_3d = np.zeros((len(data), max_vec_rnn, 300), dtype=np.float32)
    x = 0
    time_bf, time_start = time.time(), time.time()

    for i, instance in enumerate(data):

        cnt = 0
        list_temp = []
        list_temp2 = []
        for word in instance.split():
            try:
                vec = wv[word]
                list_temp2.append(list(vec))
                list_temp = list_temp + list(vec)
                if cnt < 20:
                    np_output_3d[i, cnt, :] = vec
                    cnt += 1
            except:
                pass

        temp_avg = np.mean(list_temp2, axis=0)
        list_output_avg.append(temp_avg)
        list_temp = (list_temp + max_vec_concat*300*[0.0])[:max_vec_concat*30
        list_output_concat.append(list_temp)

        x += 1
        if x % 2000 == 0:
            print(x, time.time() - time_bf)
            time_bf = time.time()

    list_output_avg = [x if isinstance(x, np.ndarray) else np.zeros(300) for 
```

```
    list_output_concat = [x if len(x) > 0 else [0.0]*3000 for x in list_outpu

    df_output_avg = pd.DataFrame(list_output_avg, columns=['c'+str(i) for i i
    df_output_concat = pd.DataFrame(list_output_concat, columns=['c'+str(i) f

    print(time.time() - time_start)

    return df_output_avg, df_output_concat, np_output_3d
```

3(a). Single perceptron

In [28]:

```
# extract Word2Vec features
X_train_w2v, X_train_w2v_concat, X_train_3d = extract_features_w2v(X_train, 1
X_test_w2v, X_test_w2v_concat, X_test_3d = extract_features_w2v(X_test, 10, 2
```

```
2000 5.053712844848633
/opt/anaconda3/lib/python3.8/site-packages/numpy/core/fromnumeric.py:3464: Run
timeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/opt/anaconda3/lib/python3.8/site-packages/numpy/core/_methods.py:192: Runtime
Warning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
4000 5.233086109161377
6000 5.562352895736694
8000 6.524694919586182
10000 6.205254793167114
12000 5.549623250961304
14000 6.020545959472656
16000 5.951589822769165
18000 4.73419976234436
20000 5.9541168212890625
22000 4.642481088638306
24000 5.937429666519165
26000 5.508419036865234
28000 6.115911245346069
30000 5.167878150939941
32000 5.232169151306152
34000 5.269054889678955
36000 5.526584148406982
38000 6.3384950160980225
40000 5.78859806060791
42000 6.483400821685791
44000 4.395051002502441
46000 5.530379056930542
48000 5.462535858154297
192.25431990623474
/opt/anaconda3/lib/python3.8/site-packages/numpy/core/fromnumeric.py:3464: Run
timeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/opt/anaconda3/lib/python3.8/site-packages/numpy/core/_methods.py:192: Runtime
Warning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
2000 5.040863037109375
4000 4.650351285934448
6000 6.271064043045044
8000 6.033645153045654
10000 5.058319807052612
12000 4.746193885803223
43.062626123428345
```

In [29]:

```
X_train_w2v.fillna(0, inplace=True)
X_train_w2v_concat.fillna(0, inplace=True)
X_test_w2v.fillna(0, inplace=True)
X_test_w2v_concat.fillna(0, inplace=True)
```

In [30]:
```
# TFIDF features
clf_perceptron = Perceptron(random_state=r_state, penalty='elasticnet')
clf_perceptron.fit(X_train_tfidf, y_train)
acc_tfidf_perceptron = clf_perceptron.score(X_test_tfidf, y_test)
print('Accuracy (TFIDF, Perceptron) =', acc_tfidf_perceptron)
```

```
Accuracy (TFIDF, Perceptron) = 0.6088333333333333
```

In [31]:
```
# Word2Vec features
clf_perceptron = Perceptron(random_state=r_state, penalty='elasticnet')
clf_perceptron.fit(X_train_w2v, y_train)
acc_w2v_perceptron = clf_perceptron.score(X_test_w2v, y_test)
print('Accuracy (W2V, Perceptron) =', acc_w2v_perceptron)
```

```
Accuracy (W2V, Perceptron) = 0.5390833333333334
```

3(b). SVM

In [32]:
```
# TFIDF features
clf_SVC = LinearSVC(random_state=r_state, multi_class='ovr', dual=True, max_i
clf_SVC.fit(X_train_tfidf, y_train)
acc_tfidf_svm = clf_SVC.score(X_test_tfidf, y_test)
print('Accuracy (TFIDF, SVM) =', acc_tfidf_svm)
```

```
Accuracy (TFIDF, SVM) = 0.65425
```

In [33]:
```
# Word2Vec features
clf_SVC = LinearSVC(random_state=r_state, multi_class='ovr', dual=True, max_i
clf_SVC.fit(X_train_w2v, y_train)
acc_w2v_svm = clf_SVC.score(X_test_w2v, y_test)
print('Accuracy (W2V, SVM) =', acc_w2v_svm)
```

```
Accuracy (W2V, SVM) = 0.62775
```

> What do you conclude from comparing performances for the models trained using the two different feature types (TF-IDF and your trained Word2Vec features)?
> The accuracy values from the models that use Word2Vec as input features are much lower using perceptron and slightly lower using SVM, meaning that word-level feature extraction in this case is less meaningful for model training than using a document-level features. My assumption on this behavior is that using average Word2Vec as features are not telling anything about the sentence as a whole - the effect of the keyword that might tell the sentiment got diluted from averaging all the words / it does not consider the sequence of the words or the long-term dependencies / etc.

4. Feedforward Neural Networks

In [34]:
```
# Converting data to tensors
X_train_tensor = torch.from_numpy(X_train_w2v.to_numpy().astype(np.float32))
X_test_tensor = torch.from_numpy(X_test_w2v.to_numpy().astype(np.float32))

y_train_tensor = torch.tensor(y_train.values)
y_test_tensor = torch.tensor(y_test.values)
```

```
# Passing to DataLoader
train_tensor = data_utils.TensorDataset(X_train_tensor, y_train_tensor)
train_loader = data_utils.DataLoader(train_tensor, batch_size=10, shuffle=Tru

test_tensor = data_utils.TensorDataset(X_test_tensor, y_test_tensor)
test_loader = data_utils.DataLoader(test_tensor, batch_size=10, shuffle=True)
```

In [35]:

```
class FNN(nn.Module):
    def __init__(self, n_features):
        super(FNN, self).__init__()
        # number of hidden nodes in each layer
        hidden_1 = 100
        hidden_2 = 10

        self.fc1 = nn.Linear(n_features, 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):
        # add hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add output layer
        x = self.fc3(x)
        return x
```

In [36]:

```
def get_acc(y_pred, y_true):
    max_scores, max_idx_class = y_pred.max(dim=1)
    num_match = (y_true == max_idx_class).sum().item()
    acc = num_match / y_true.size(0)
    return num_match, acc
```

In [37]:

```
def train(n_epochs, model, train_loader, valid_loader, optimizer, criterion, 
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf # set initial "min" to infinity
    len_train = len(train_loader.dataset)
    len_valid = len(valid_loader.dataset)
    best_acc = -np.Inf
    best_sd = None

    for epoch in range(n_epochs):
        # monitor training loss
        train_loss = 0.0
        train_acc = 0.0
        valid_loss = 0.0
        valid_acc = 0.0

        ###################
        # train the model #
        ###################
        model.train() # prep model for training

        for data, target in train_loader:
            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to th
            output = model(data)
```

```
            # calculate the loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to mod
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update running training loss
            train_loss += loss.item()    # *data.size(0)
            # update training accuracy
            num_match, _ = get_acc(output, target)
            train_acc += num_match

        ######################
        # validate the model #
        ######################
        model.eval() # prep model for evaluation
        with torch.no_grad():
            for data, target in valid_loader:
                # forward pass: compute predicted outputs by passing inputs t
                output = model(data)
                # calculate the loss
                loss = criterion(output, target)
                # update running validation loss
                valid_loss += loss.item()
                # update training accuracy
                num_match, _ = get_acc(output, target)
                valid_acc += num_match

        train_acc = train_acc / len_train
        valid_acc = valid_acc / len_valid

        train_loss = train_loss / len_train * 10
        valid_loss = valid_loss / len_valid * 10

        # save params of the best model
        if valid_acc > best_acc:
            best_acc = valid_acc
            best_sd = copy.deepcopy(model.state_dict())

        if epoch % print_every == 0:
            print('Epoch: {} \tTrain Loss: {:.4f}\tValid Loss: {:.4f}\tTrain A
                epoch+1, train_loss, valid_loss, train_acc, valid_acc))

    return best_sd
```

4(a). Average Word2Vec vectors

In [38]:

```
# initialize the NN
model = FNN(300)
print(model)

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.05
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
```

```
FNN(
  (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)
)
```

In [39]:

```python
best_sd = train(30, model, train_loader, test_loader, optimizer, criterion, p
```

```
Epoch: 1        Train Loss: 0.9719      Valid Loss: 0.8644      Train Acc: 0.4
985     Valid Acc: 0.5916
Epoch: 2        Train Loss: 0.8584      Valid Loss: 0.8527      Train Acc: 0.6
022     Valid Acc: 0.6048
Epoch: 3        Train Loss: 0.8387      Valid Loss: 0.8933      Train Acc: 0.6
128     Valid Acc: 0.5831
Epoch: 4        Train Loss: 0.8272      Valid Loss: 0.8182      Train Acc: 0.6
199     Valid Acc: 0.6283
Epoch: 5        Train Loss: 0.8194      Valid Loss: 0.8260      Train Acc: 0.6
244     Valid Acc: 0.6246
Epoch: 6        Train Loss: 0.8133      Valid Loss: 0.8310      Train Acc: 0.6
272     Valid Acc: 0.6168
Epoch: 7        Train Loss: 0.8065      Valid Loss: 0.8122      Train Acc: 0.6
315     Valid Acc: 0.6279
Epoch: 8        Train Loss: 0.8002      Valid Loss: 0.8073      Train Acc: 0.6
338     Valid Acc: 0.6306
Epoch: 9        Train Loss: 0.7949      Valid Loss: 0.8278      Train Acc: 0.6
380     Valid Acc: 0.6217
Epoch: 10       Train Loss: 0.7901      Valid Loss: 0.8030      Train Acc: 0.6
395     Valid Acc: 0.6325
Epoch: 11       Train Loss: 0.7849      Valid Loss: 0.8123      Train Acc: 0.6
435     Valid Acc: 0.6250
Epoch: 12       Train Loss: 0.7806      Valid Loss: 0.8053      Train Acc: 0.6
441     Valid Acc: 0.6327
Epoch: 13       Train Loss: 0.7759      Valid Loss: 0.8014      Train Acc: 0.6
488     Valid Acc: 0.6347
Epoch: 14       Train Loss: 0.7711      Valid Loss: 0.8129      Train Acc: 0.6
501     Valid Acc: 0.6293
Epoch: 15       Train Loss: 0.7659      Valid Loss: 0.8011      Train Acc: 0.6
509     Valid Acc: 0.6406
Epoch: 16       Train Loss: 0.7606      Valid Loss: 0.8076      Train Acc: 0.6
534     Valid Acc: 0.6359
Epoch: 17       Train Loss: 0.7572      Valid Loss: 0.8144      Train Acc: 0.6
596     Valid Acc: 0.6327
Epoch: 18       Train Loss: 0.7530      Valid Loss: 0.8047      Train Acc: 0.6
619     Valid Acc: 0.6387
Epoch: 19       Train Loss: 0.7479      Valid Loss: 0.8076      Train Acc: 0.6
616     Valid Acc: 0.6357
Epoch: 20       Train Loss: 0.7450      Valid Loss: 0.8027      Train Acc: 0.6
633     Valid Acc: 0.6388
Epoch: 21       Train Loss: 0.7381      Valid Loss: 0.8125      Train Acc: 0.6
713     Valid Acc: 0.6273
Epoch: 22       Train Loss: 0.7350      Valid Loss: 0.8501      Train Acc: 0.6
700     Valid Acc: 0.6061
Epoch: 23       Train Loss: 0.7309      Valid Loss: 0.8092      Train Acc: 0.6
730     Valid Acc: 0.6324
Epoch: 24       Train Loss: 0.7250      Valid Loss: 0.8080      Train Acc: 0.6
766     Valid Acc: 0.6336
Epoch: 25       Train Loss: 0.7213      Valid Loss: 0.8086      Train Acc: 0.6
770     Valid Acc: 0.6361
Epoch: 26       Train Loss: 0.7146      Valid Loss: 0.8282      Train Acc: 0.6
834     Valid Acc: 0.6270
Epoch: 27       Train Loss: 0.7116      Valid Loss: 0.8224      Train Acc: 0.6
798     Valid Acc: 0.6286
Epoch: 28       Train Loss: 0.7079      Valid Loss: 0.8150      Train Acc: 0.6
843     Valid Acc: 0.6324
Epoch: 29       Train Loss: 0.7034      Valid Loss: 0.8204      Train Acc: 0.6
885     Valid Acc: 0.6360
Epoch: 30       Train Loss: 0.6973      Valid Loss: 0.9531      Train Acc: 0.6
915     Valid Acc: 0.5795
```

In [40]:

```python
model_best_fnn_avg = FNN(300)
model_best_fnn_avg.load_state_dict(best_sd)
```

```
_, train_acc = get_acc(model_best_fnn_avg(X_train_tensor), y_train_tensor)
_, test_acc = get_acc(model_best_fnn_avg(X_test_tensor), y_test_tensor)

print('FNN using average Word2Vec vectors:')
print('Training Accuracy =', train_acc)
print('Testing Accuracy =', test_acc)
```

```
FNN using average Word2Vec vectors:
Training Accuracy = 0.6687916666666667
Testing Accuracy = 0.6405833333333333
```

4(b). Concatenated Word2Vec vectors

In [41]:
```
# Converting data to tensors
X_train_tensor_concat = torch.from_numpy(X_train_w2v_concat.to_numpy().astype
X_test_tensor_concat = torch.from_numpy(X_test_w2v_concat.to_numpy().astype(n

# Passing to DataLoader
train_tensor = data_utils.TensorDataset(X_train_tensor_concat, y_train_tensor
train_loader = data_utils.DataLoader(train_tensor, batch_size=10, shuffle=Tru

test_tensor = data_utils.TensorDataset(X_test_tensor_concat, y_test_tensor)
test_loader = data_utils.DataLoader(test_tensor, batch_size=10, shuffle=True)
```

In [42]:
```
# initialize the NN
model = FNN(3000)
print(model)

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.005)
```

```
FNN(
  (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 [43]:
```
best_sd = train(20, model, train_loader, test_loader, optimizer, criterion, p
```

```
Epoch: 1        Train Loss: 1.0627      Valid Loss: 0.9910      Train Acc: 0.4
349     Valid Acc: 0.4943
Epoch: 2        Train Loss: 0.9529      Valid Loss: 0.9380      Train Acc: 0.5
219     Valid Acc: 0.5373
Epoch: 3        Train Loss: 0.9135      Valid Loss: 0.9217      Train Acc: 0.5
595     Valid Acc: 0.5558
Epoch: 4        Train Loss: 0.8848      Valid Loss: 0.9114      Train Acc: 0.5
817     Valid Acc: 0.5657
Epoch: 5        Train Loss: 0.8646      Valid Loss: 0.9025      Train Acc: 0.5
958     Valid Acc: 0.5694
Epoch: 6        Train Loss: 0.8459      Valid Loss: 0.9021      Train Acc: 0.6
082     Valid Acc: 0.5660
Epoch: 7        Train Loss: 0.8275      Valid Loss: 0.9084      Train Acc: 0.6
183     Valid Acc: 0.5675
Epoch: 8        Train Loss: 0.8050      Valid Loss: 0.9111      Train Acc: 0.6
318     Valid Acc: 0.5700
Epoch: 9        Train Loss: 0.7787      Valid Loss: 0.9102      Train Acc: 0.6
486     Valid Acc: 0.5653
Epoch: 10       Train Loss: 0.7435      Valid Loss: 0.9413      Train Acc: 0.6
```

```
705      Valid Acc: 0.5614
Epoch: 11        Train Loss: 0.7002      Valid Loss: 0.9490      Train Acc: 0.6
959      Valid Acc: 0.5589
Epoch: 12        Train Loss: 0.6453      Valid Loss: 0.9825      Train Acc: 0.7
289      Valid Acc: 0.5584
Epoch: 13        Train Loss: 0.5747      Valid Loss: 1.0422      Train Acc: 0.7
694      Valid Acc: 0.5488
Epoch: 14        Train Loss: 0.4997      Valid Loss: 1.1095      Train Acc: 0.8
083      Valid Acc: 0.5472
Epoch: 15        Train Loss: 0.4153      Valid Loss: 1.2141      Train Acc: 0.8
465      Valid Acc: 0.5347
Epoch: 16        Train Loss: 0.3360      Valid Loss: 1.3394      Train Acc: 0.8
821      Valid Acc: 0.5331
Epoch: 17        Train Loss: 0.2590      Valid Loss: 1.4817      Train Acc: 0.9
157      Valid Acc: 0.5302
Epoch: 18        Train Loss: 0.1949      Valid Loss: 1.6066      Train Acc: 0.9
412      Valid Acc: 0.5248
Epoch: 19        Train Loss: 0.1425      Valid Loss: 1.7546      Train Acc: 0.9
603      Valid Acc: 0.5288
Epoch: 20        Train Loss: 0.1011      Valid Loss: 1.8605      Train Acc: 0.9
734      Valid Acc: 0.5292
```

In [44]:

```
model_best_fnn_concat = FNN(3000)
model_best_fnn_concat.load_state_dict(best_sd)

_, train_acc = get_acc(model_best_fnn_concat(X_train_tensor_concat), y_train_
_, test_acc = get_acc(model_best_fnn_concat(X_test_tensor_concat), y_test_ten

print('FNN using concat Word2Vec vectors:')
print('Training Accuracy =', train_acc)
print('Testing Accuracy =', test_acc)
```

```
FNN using concat Word2Vec vectors:
Training Accuracy = 0.664125
Testing Accuracy = 0.57
```

> What do you conclude by comparing accuracy values you obtain with those obtained in the “Simple Models” section?
>
> - Considering only models using Word2Vec vectors as input features, FNN (with average W2V) gives higher accuracy which is as expected since the model is more complex than using just a single perceptron.
> - When considering different types of input W2V vectors for FNN, model using average W2V is better than using concatenated W2V

5. Recurrent Neural Networks

5(a). Simple RNN

In [45]:
```
# Converting data to tensors
X_train_tensor_3d = torch.from_numpy(X_train_3d)
X_test_tensor_3d = torch.from_numpy(X_test_3d)

# Passing to DataLoader
train_tensor = data_utils.TensorDataset(X_train_tensor_3d, y_train_tensor)
train_loader = data_utils.DataLoader(train_tensor, batch_size=10, shuffle=Fal

test_tensor = data_utils.TensorDataset(X_test_tensor_3d, y_test_tensor)
test_loader = data_utils.DataLoader(test_tensor, batch_size=10, shuffle=False
```

In [46]:
```
class model_RNN(nn.Module):
    def __init__(self, hidden_dim, n_layers):
        super(model_RNN, self).__init__()

        # Defining some parameters
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        #Defining the layers
        # RNN Layer
        self.rnn = nn.RNN(input_size=300, hidden_size=hidden_dim, num_layers=
        # Fully connected layer
        self.fc = nn.Linear(hidden_dim, 3)

    def forward(self, x):

        batch_size = x.size(0)

        # Initializing hidden state for first input using method defined belo
        hidden = self.init_hidden(batch_size)

        # Passing in the input and hidden state into the model and obtaining 
        out, hidden = self.rnn(x, hidden)

        # Reshaping the outputs such that it can be fit into the fully connec
        out = out[:, -1, :]
        out = self.fc(hidden[-1])

        return out

    def init_hidden(self, batch_size):
        # This method generates the first hidden state of zeros which we'll u
        # We'll send the tensor holding the hidden state to the device we spe
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim)
        return hidden
```

In [59]:
```
# initialize the NN
model = model_RNN(20, 2)
print(model)

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.005)
```

```
model_RNN(
  (rnn): RNN(300, 20, num_layers=2, batch_first=True)
```

```
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

In [60]:
```
best_sd = train(30, model, train_loader, test_loader, optimizer, criterion, p
```

```
Epoch: 1        Train Loss: 1.1003      Valid Loss: 1.0990      Train Acc: 0.3
327     Valid Acc: 0.3372
Epoch: 2        Train Loss: 1.0980      Valid Loss: 1.0975      Train Acc: 0.3
436     Valid Acc: 0.3417
Epoch: 3        Train Loss: 1.0948      Valid Loss: 1.0812      Train Acc: 0.3
575     Valid Acc: 0.4044
Epoch: 4        Train Loss: 1.0049      Valid Loss: 0.9439      Train Acc: 0.4
827     Valid Acc: 0.5282
Epoch: 5        Train Loss: 0.9472      Valid Loss: 0.9196      Train Acc: 0.5
256     Valid Acc: 0.5492
Epoch: 6        Train Loss: 0.9339      Valid Loss: 0.9110      Train Acc: 0.5
381     Valid Acc: 0.5558
Epoch: 7        Train Loss: 0.9225      Valid Loss: 0.9046      Train Acc: 0.5
465     Valid Acc: 0.5603
Epoch: 8        Train Loss: 0.9138      Valid Loss: 0.9013      Train Acc: 0.5
513     Valid Acc: 0.5674
Epoch: 9        Train Loss: 0.9075      Valid Loss: 0.8965      Train Acc: 0.5
560     Valid Acc: 0.5664
Epoch: 10       Train Loss: 0.9028      Valid Loss: 0.8919      Train Acc: 0.5
593     Valid Acc: 0.5713
Epoch: 11       Train Loss: 0.8989      Valid Loss: 0.8927      Train Acc: 0.5
614     Valid Acc: 0.5683
Epoch: 12       Train Loss: 0.8970      Valid Loss: 0.8954      Train Acc: 0.5
649     Valid Acc: 0.5718
Epoch: 13       Train Loss: 0.8934      Valid Loss: 0.8920      Train Acc: 0.5
679     Valid Acc: 0.5713
Epoch: 14       Train Loss: 0.8925      Valid Loss: 0.8890      Train Acc: 0.5
678     Valid Acc: 0.5716
Epoch: 15       Train Loss: 0.8894      Valid Loss: 0.8864      Train Acc: 0.5
699     Valid Acc: 0.5722
Epoch: 16       Train Loss: 0.8850      Valid Loss: 0.8828      Train Acc: 0.5
737     Valid Acc: 0.5756
Epoch: 17       Train Loss: 0.8838      Valid Loss: 0.8895      Train Acc: 0.5
749     Valid Acc: 0.5721
Epoch: 18       Train Loss: 0.8814      Valid Loss: 0.8822      Train Acc: 0.5
767     Valid Acc: 0.5833
Epoch: 19       Train Loss: 0.8794      Valid Loss: 0.8821      Train Acc: 0.5
785     Valid Acc: 0.5797
Epoch: 20       Train Loss: 0.8783      Valid Loss: 0.8774      Train Acc: 0.5
799     Valid Acc: 0.5777
Epoch: 21       Train Loss: 0.8764      Valid Loss: 0.8862      Train Acc: 0.5
820     Valid Acc: 0.5832
Epoch: 22       Train Loss: 0.8741      Valid Loss: 0.8802      Train Acc: 0.5
834     Valid Acc: 0.5829
Epoch: 23       Train Loss: 0.8734      Valid Loss: 0.8797      Train Acc: 0.5
859     Valid Acc: 0.5746
Epoch: 24       Train Loss: 0.8728      Valid Loss: 0.8825      Train Acc: 0.5
861     Valid Acc: 0.5778
Epoch: 25       Train Loss: 0.9082      Valid Loss: 0.9653      Train Acc: 0.5
593     Valid Acc: 0.5389
Epoch: 26       Train Loss: 0.9248      Valid Loss: 0.9105      Train Acc: 0.5
509     Valid Acc: 0.5647
Epoch: 27       Train Loss: 0.9318      Valid Loss: 0.9223      Train Acc: 0.5
460     Valid Acc: 0.5498
Epoch: 28       Train Loss: 0.9029      Valid Loss: 0.9365      Train Acc: 0.5
729     Valid Acc: 0.5465
Epoch: 29       Train Loss: 0.9028      Valid Loss: 0.8848      Train Acc: 0.5
737     Valid Acc: 0.5885
Epoch: 30       Train Loss: 0.8874      Valid Loss: 0.8896      Train Acc: 0.5
884     Valid Acc: 0.5883
```

In [61]:
```
model_best_rnn = model_RNN(20, 2)
model_best_rnn.load_state_dict(best_sd)
```

```
_, train_acc = get_acc(model_best_rnn(X_train_tensor_3d), y_train_tensor)
_, test_acc = get_acc(model_best_rnn(X_test_tensor_3d), y_test_tensor)

print('RNN:')
print('Training Accuracy =', train_acc)
print('Testing Accuracy =', test_acc)
```

```
RNN:
Training Accuracy = 0.5914583333333333
Testing Accuracy = 0.5885
```

5(b). GRU: Gated Recurrent Unit

In [50]:

```
class model_GRU(nn.Module):
    def __init__(self, hidden_dim, n_layers):
        super(model_GRU, self).__init__()

        # Defining some parameters
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        #Defining the layers
        # RNN Layer
        self.rnn = nn.GRU(input_size=300, hidden_size=hidden_dim, num_layers=
        # Fully connected layer
        self.fc = nn.Linear(hidden_dim, 3)

    def forward(self, x):

        batch_size = x.size(0)

        # Initializing hidden state for first input using method defined belo
        hidden = self.init_hidden(batch_size)

        # Passing in the input and hidden state into the model and obtaining 
        out, _ = self.rnn(x, hidden)

        # Reshaping the outputs such that it can be fit into the fully connec
        out = out[:, -1, :]
        out = self.fc(out)

        return out

    def init_hidden(self, batch_size):
        # This method generates the first hidden state of zeros which we'll u
        # We'll send the tensor holding the hidden state to the device we spe
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim)
        return hidden
```

In [51]:

```
# initialize the NN
model = model_GRU(20, 2)
print(model)

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
```

```
model_GRU(
  (rnn): GRU(300, 20, num_layers=2, batch_first=True)
```

```
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

In [52]:

```
best_sd = train(30, model, train_loader, test_loader, optimizer, criterion, p
```

```
Epoch: 1         Train Loss: 1.0956      Valid Loss: 1.0796      Train Acc: 0.3
530      Valid Acc: 0.3924
Epoch: 2         Train Loss: 0.9740      Valid Loss: 0.8967      Train Acc: 0.5
028      Valid Acc: 0.5582
Epoch: 3         Train Loss: 0.9000      Valid Loss: 0.8669      Train Acc: 0.5
615      Valid Acc: 0.5842
Epoch: 4         Train Loss: 0.8690      Valid Loss: 0.8438      Train Acc: 0.5
881      Valid Acc: 0.5987
Epoch: 5         Train Loss: 0.8463      Valid Loss: 0.8308      Train Acc: 0.6
048      Valid Acc: 0.6096
Epoch: 6         Train Loss: 0.8312      Valid Loss: 0.8206      Train Acc: 0.6
156      Valid Acc: 0.6178
Epoch: 7         Train Loss: 0.8196      Valid Loss: 0.8127      Train Acc: 0.6
219      Valid Acc: 0.6248
Epoch: 8         Train Loss: 0.8102      Valid Loss: 0.8071      Train Acc: 0.6
273      Valid Acc: 0.6273
Epoch: 9         Train Loss: 0.8025      Valid Loss: 0.8028      Train Acc: 0.6
317      Valid Acc: 0.6278
Epoch: 10        Train Loss: 0.7959      Valid Loss: 0.7995      Train Acc: 0.6
341      Valid Acc: 0.6298
Epoch: 11        Train Loss: 0.7902      Valid Loss: 0.7967      Train Acc: 0.6
374      Valid Acc: 0.6312
Epoch: 12        Train Loss: 0.7849      Valid Loss: 0.7944      Train Acc: 0.6
405      Valid Acc: 0.6330
Epoch: 13        Train Loss: 0.7801      Valid Loss: 0.7924      Train Acc: 0.6
433      Valid Acc: 0.6346
Epoch: 14        Train Loss: 0.7756      Valid Loss: 0.7907      Train Acc: 0.6
467      Valid Acc: 0.6358
Epoch: 15        Train Loss: 0.7713      Valid Loss: 0.7892      Train Acc: 0.6
489      Valid Acc: 0.6368
Epoch: 16        Train Loss: 0.7672      Valid Loss: 0.7880      Train Acc: 0.6
519      Valid Acc: 0.6372
Epoch: 17        Train Loss: 0.7633      Valid Loss: 0.7869      Train Acc: 0.6
542      Valid Acc: 0.6377
Epoch: 18        Train Loss: 0.7595      Valid Loss: 0.7861      Train Acc: 0.6
562      Valid Acc: 0.6379
Epoch: 19        Train Loss: 0.7558      Valid Loss: 0.7855      Train Acc: 0.6
581      Valid Acc: 0.6385
Epoch: 20        Train Loss: 0.7522      Valid Loss: 0.7850      Train Acc: 0.6
599      Valid Acc: 0.6395
Epoch: 21        Train Loss: 0.7487      Valid Loss: 0.7848      Train Acc: 0.6
622      Valid Acc: 0.6395
Epoch: 22        Train Loss: 0.7452      Valid Loss: 0.7847      Train Acc: 0.6
641      Valid Acc: 0.6396
Epoch: 23        Train Loss: 0.7418      Valid Loss: 0.7848      Train Acc: 0.6
662      Valid Acc: 0.6394
Epoch: 24        Train Loss: 0.7385      Valid Loss: 0.7850      Train Acc: 0.6
676      Valid Acc: 0.6401
Epoch: 25        Train Loss: 0.7352      Valid Loss: 0.7853      Train Acc: 0.6
696      Valid Acc: 0.6397
Epoch: 26        Train Loss: 0.7319      Valid Loss: 0.7858      Train Acc: 0.6
703      Valid Acc: 0.6395
Epoch: 27        Train Loss: 0.7287      Valid Loss: 0.7863      Train Acc: 0.6
720      Valid Acc: 0.6387
Epoch: 28        Train Loss: 0.7255      Valid Loss: 0.7869      Train Acc: 0.6
741      Valid Acc: 0.6390
Epoch: 29        Train Loss: 0.7223      Valid Loss: 0.7876      Train Acc: 0.6
755      Valid Acc: 0.6372
Epoch: 30        Train Loss: 0.7192      Valid Loss: 0.7884      Train Acc: 0.6
780      Valid Acc: 0.6362
```

In [53]:

```
model_best_gru = model_GRU(20, 2)
model_best_gru.load_state_dict(best_sd)
```

```
_, train_acc = get_acc(model_best_gru(X_train_tensor_3d), y_train_tensor)
_, test_acc = get_acc(model_best_gru(X_test_tensor_3d), y_test_tensor)

print('GRU:')
print('Training Accuracy =', train_acc)
print('Testing Accuracy =', test_acc)
```

```
GRU:
Training Accuracy = 0.6696041666666667
Testing Accuracy = 0.6400833333333333
```

5(c). LSTM

In [54]:

```
class model_LSTM(nn.Module):
    def __init__(self, hidden_dim, n_layers):
        super(model_LSTM, self).__init__()

        # Defining some parameters
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        #Defining the layers
        # RNN Layer
        self.rnn = nn.LSTM(input_size=300, hidden_size=hidden_dim, num_layers
        # Fully connected layer
        self.fc = nn.Linear(hidden_dim, 3)

    def forward(self, x):

        batch_size = x.size(0)

        # Initializing hidden state for first input using method defined belo
        h0, c0 = self.init_hidden(batch_size)

        # Passing in the input and hidden state into the model and obtaining
        out, _ = self.rnn(x, (h0, c0))

        # Reshaping the outputs such that it can be fit into the fully connec
        out = out[:, -1, :]
        out = self.fc(out)

        return out

    def init_hidden(self, batch_size):
        # This method generates the first hidden state of zeros which we'll u
        # We'll send the tensor holding the hidden state to the device we spe
        h0 = torch.randn(self.n_layers, batch_size, self.hidden_dim)
        c0 = torch.randn(self.n_layers, batch_size, self.hidden_dim)
        return h0, c0
```

In [55]:

```
# initialize the NN
model = model_LSTM(20, 2)
print(model)

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.03)
```

```
model_LSTM(
  (rnn): LSTM(300, 20, num_layers=2, batch_first=True)
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

In [56]:

```
best_sd = train(30, model, train_loader, test_loader, optimizer, criterion, p
```

```
Epoch: 1         Train Loss: 1.0992      Valid Loss: 1.0988      Train Acc: 0.3
311      Valid Acc: 0.3333
Epoch: 2         Train Loss: 1.0981      Valid Loss: 1.0959      Train Acc: 0.3
412      Valid Acc: 0.3375
Epoch: 3         Train Loss: 1.0347      Valid Loss: 0.9716      Train Acc: 0.4
532      Valid Acc: 0.5158
Epoch: 4         Train Loss: 0.9684      Valid Loss: 0.9413      Train Acc: 0.5
137      Valid Acc: 0.5275
Epoch: 5         Train Loss: 0.9424      Valid Loss: 0.9254      Train Acc: 0.5
326      Valid Acc: 0.5370
Epoch: 6         Train Loss: 0.9230      Valid Loss: 0.8986      Train Acc: 0.5
487      Valid Acc: 0.5666
Epoch: 7         Train Loss: 0.9079      Valid Loss: 0.8851      Train Acc: 0.5
671      Valid Acc: 0.5785
Epoch: 8         Train Loss: 0.8925      Valid Loss: 0.8725      Train Acc: 0.5
758      Valid Acc: 0.5877
Epoch: 9         Train Loss: 0.8816      Valid Loss: 0.8616      Train Acc: 0.5
830      Valid Acc: 0.5968
Epoch: 10        Train Loss: 0.8729      Valid Loss: 0.8581      Train Acc: 0.5
909      Valid Acc: 0.5996
Epoch: 11        Train Loss: 0.8665      Valid Loss: 0.8529      Train Acc: 0.5
940      Valid Acc: 0.6028
Epoch: 12        Train Loss: 0.8591      Valid Loss: 0.8476      Train Acc: 0.5
978      Valid Acc: 0.6023
Epoch: 13        Train Loss: 0.8524      Valid Loss: 0.8433      Train Acc: 0.6
020      Valid Acc: 0.6062
Epoch: 14        Train Loss: 0.8480      Valid Loss: 0.8415      Train Acc: 0.6
031      Valid Acc: 0.6112
Epoch: 15        Train Loss: 0.8424      Valid Loss: 0.8362      Train Acc: 0.6
072      Valid Acc: 0.6129
Epoch: 16        Train Loss: 0.8376      Valid Loss: 0.8377      Train Acc: 0.6
108      Valid Acc: 0.6113
Epoch: 17        Train Loss: 0.8332      Valid Loss: 0.8318      Train Acc: 0.6
146      Valid Acc: 0.6125
Epoch: 18        Train Loss: 0.8307      Valid Loss: 0.8308      Train Acc: 0.6
138      Valid Acc: 0.6151
Epoch: 19        Train Loss: 0.8261      Valid Loss: 0.8311      Train Acc: 0.6
173      Valid Acc: 0.6107
Epoch: 20        Train Loss: 0.8232      Valid Loss: 0.8274      Train Acc: 0.6
183      Valid Acc: 0.6150
Epoch: 21        Train Loss: 0.8191      Valid Loss: 0.8293      Train Acc: 0.6
205      Valid Acc: 0.6119
Epoch: 22        Train Loss: 0.8165      Valid Loss: 0.8233      Train Acc: 0.6
230      Valid Acc: 0.6143
Epoch: 23        Train Loss: 0.8115      Valid Loss: 0.8193      Train Acc: 0.6
262      Valid Acc: 0.6166
Epoch: 24        Train Loss: 0.8096      Valid Loss: 0.8200      Train Acc: 0.6
284      Valid Acc: 0.6213
Epoch: 25        Train Loss: 0.8069      Valid Loss: 0.8190      Train Acc: 0.6
291      Valid Acc: 0.6191
Epoch: 26        Train Loss: 0.8051      Valid Loss: 0.8164      Train Acc: 0.6
305      Valid Acc: 0.6252
Epoch: 27        Train Loss: 0.7997      Valid Loss: 0.8219      Train Acc: 0.6
317      Valid Acc: 0.6158
Epoch: 28        Train Loss: 0.7972      Valid Loss: 0.8184      Train Acc: 0.6
322      Valid Acc: 0.6203
Epoch: 29        Train Loss: 0.7956      Valid Loss: 0.8148      Train Acc: 0.6
350      Valid Acc: 0.6215
Epoch: 30        Train Loss: 0.7926      Valid Loss: 0.8166      Train Acc: 0.6
354      Valid Acc: 0.6200
```

In [58]:
```
model_best_lstm = model_LSTM(20, 2)
model_best_lstm.load_state_dict(best_sd)

_, train_acc = get_acc(model_best_lstm(X_train_tensor_3d), y_train_tensor)
_, test_acc = get_acc(model_best_lstm(X_test_tensor_3d), y_test_tensor)

print('LSTM:')
print('Training Accuracy =', train_acc)
print('Testing Accuracy =', test_acc)
```

```
LSTM:
Training Accuracy = 0.6322708333333333
Testing Accuracy = 0.6245
```

> What do you conclude by comparing accuracy values you obtain by GRU, LSTM, and simple RNN?
>
> - GRU gives the best test accuracy at 64% while LSTM's accuracy is slightly lower at 62.5%. For simple RNN, the best test accuracy achieved is at 58.9%. This can be inferred that this sentiment prediction task also relies on the long-term dependency of the input reviews, and LSTM and GRU are doing better task in storing and retrieving long-term dependency than simple RNN

References

- https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html
- https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook
- https://stackoverflow.com/questions/3438756/some-built-in-to-pad-a-list-in-python
- https://blog.floydhub.com/a-beginners-guide-on-recurrent-neural-networks-with-pytorch/
- https://androidkt.com/copy-pytorch-model-using-deepcopy-and-state_dict/