In [144…

```
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet')
```

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

Out[144…

```
True
```

In [145…

```
# Might need this
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/jadriantan/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[145…

```
True
```

In [146…

```
! pip3 install bs4 # in case you don't have it installed
```

```
Requirement already satisfied: bs4 in /Users/jadriantan/miniconda3/lib/python3.11/site-packages (0.0.1)
Requirement already satisfied: beautifulsoup4 in /Users/jadriantan/miniconda3/lib/python3.11/site-packages (from bs4) (4.12.2)
Requirement already satisfied: soupsieve>1.2 in /Users/jadriantan/miniconda3/lib/python3.11/site-packages (from beautifulsoup4->bs4) (2.5)
```

Data Generation (from HW1)

In [147…

```
df = pd.read_csv("data.csv", sep=',', on_bad_lines='skip')
```

```
/var/folders/z4/sk_p32bx74j8lpnjdx_bmjz80000gn/T/ipykernel_43668/2324558469.py:1: DtypeWarning: Columns (1,7,17,18,19,20,21) have mixed types. Specify dty
pe option on import or set low_memory=False.
  df = pd.read_csv("data.csv", sep=',', on_bad_lines='skip')
```

In [148…

```
# Extract Reviews and Ratings fields
df = df.loc[:, ['review_body', 'star_rating']]

print(df.head())
```

```
                                         review_body star_rating
0                                     Great product.           5
1  What's to say about this commodity item except...           5
2    Haven't used yet, but I am sure I will like it.           5
3  Although this was labeled as &#34;new&#34; the...           1
4                    Gorgeous colors and easy to use           4
```

In [149…

```
# Converting into binary classification problem
df['label'] = df['star_rating'].apply(lambda x: 1 if x in [1,2,3] else 2)
df['review'] = df['review_body']

# Selecting 50,000 random reviews from each rating class
# Randomizing to avoid biases
df_class_1 = df[df['label'] == 1].sample(n=50000, random_state=55)
df_class_2 = df[df['label'] == 2].sample(n=50000, random_state=55)

# Creating a new df concatenating both classes
balanced_df = pd.concat([df_class_1, df_class_2])

print(balanced_df.head(10))
```

```
                                              review_body star_rating  label  \
98510   These do not work for refill ink it states one...           1      1   
233824  Once the ink cartriges were installed, things ...           3      1   
877033  Nothing special to say here - they are postcar...           3      1   
929541               It is a machine for counting bills.?           1      1   
201881  I guess you get what you pay for.  The magenta...           2      1   
456205               Not the best quality. Pale in color.           3      1   
645041               They do what they're supposed to do.           3      1   
862255               Very light weight, barely hold paper           1      1   
633612                          Only for 5 sheets or less           1      1   
736944  Not happy with these. I am not able to print w...           1      1   

                                                   review  
98510   These do not work for refill ink it states one...  
233824  Once the ink cartriges were installed, things ...  
877033  Nothing special to say here - they are postcar...  
929541               It is a machine for counting bills.?  
201881  I guess you get what you pay for.  The magenta...  
456205               Not the best quality. Pale in color.  
645041               They do what they're supposed to do.  
862255               Very light weight, barely hold paper  
633612                          Only for 5 sheets or less  
736944  Not happy with these. I am not able to print w...  
```

Preprocessing (from HW1)

In [150…

```
average_len_before = balanced_df['review'].str.len().mean()

# 1)converting all reviews into lowercase.
# Ensures consistency: "Hello" and "hello" are now the same.
balanced_df['review'] = balanced_df['review'].str.lower()

# 2)removing the HTML and URLs from the reviews
# HTML/URLs don't provide valuable information for sentiment analysis, so we remove them.
balanced_df['review'] = balanced_df['review'].str.replace(r'<.*?>', '', regex=True)
balanced_df['review'] = balanced_df['review'].str.replace(r'http\S+', '', regex=True)

# 5)performing contractions on the reviews
    # Need to process this before before removing non-alphanum chars and extra spaces
# This task provides uniformity and simplifies tokenization
def contractions_helper(ss):
```

```
    # To avoid attributionError
    if type(ss) != str:
        return
    contractions_dict = {
        "won't": "will not",
        "ain't": "am not",
        "aren't": "are not",
        "can't": "cannot",
        "hasn't": "has not",
        "coudn't": "could not",
        "they're": "they are",
        "you're": "you are",
        "we'll": "we will",
        "it's": "it is",
        "i'll": "i will",
        "he's": "he is",
        "she's": "she is"
    }
    # loop through the string and replace all contractions
    for cont, exp in contractions_dict.items():
            if cont in ss:
                 ss = ss.replace(cont, exp)
    return ss

balanced_df['review'] = balanced_df['review'].apply(contractions_helper)

# 3)removing non-alphabetical characters
# Removing non-alphanum characters since they could be noise in sentiment analysis
balanced_df['review'] = balanced_df['review'].str.replace(r'[^a-zA-Z\s]', '', regex=True)

# 4)removing extra spaces
balanced_df['review'] = balanced_df['review'].str.replace(r'\s+', ' ', regex=True)


average_len_after = balanced_df['review'].str.len().mean()

# average length decreased after cleaning due to the removal of unwanted characters, spaces, and expansion of contractions
print(f'Average length before data cleaning:{average_len_before:.4f}, Average length after data cleaning:{average_len_after:.4f}')
```

```
Average length before data cleaning:189.4582, Average length after data cleaning:179.7509
```

Feature Extraction

In [151…

```
# imports
from gensim.models import Word2Vec
import gensim.downloader as api
```

In [152…

```
# Filtering out rows where 'review' is not a string
balanced_df = balanced_df[balanced_df['review'].apply(lambda x: isinstance(x, str))]

# Word2Vec model trained with amazon reviews
sentences = balanced_df['review'].str.split().tolist() # get review sentences
# train word2vec model
my_model = Word2Vec(sentences, vector_size=300, window=13, min_count=9, workers=4)
my_model.train(sentences, total_examples=len(sentences), epochs=10)

# Pre-trained word2vec model
wv_model = api.load('word2vec-google-news-300')
```

2. Word Embedding: part (a) and (b) combined

In [153…

```
print("Example 1")
# Pretrained: check for semantic similarities
print("Word2Vec Model: Similarity for words 'excellent' and 'outstanding':", wv_model.similarity('excellent', 'outstanding'))

# My model: check semantic similarities
if 'excellent' in my_model.wv and 'outstanding' in my_model.wv:
    print("My Model: Similarity between 'excellent' and 'outstanding':", my_model.wv.similarity('excellent', 'outstanding'))

print("Example 2")
# Pretrained: check for analogy: King - Man + Woman
result = wv_model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print("Word2Vec Model: King - Man + Woman = ", result[0][0])

# My model: check analogy: King - Man + Woman
if all(word in my_model.wv for word in ['woman', 'king', 'man']):
    result = my_model.wv.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
    print("My Model: King - Man + Woman = ", result[0][0])

print("Example 3 ('delicious' and 'tasty' not in my model)")
# Pretrained model:
print("Word2Vec Model: Similarity for words 'delicious' and 'tasty':", wv_model.similarity('delicious', 'tasty'))

# My model:
if 'delicious' in my_model.wv.key_to_index and 'tasty' in my_model.wv.key_to_index:
    print("My Model: Similarity between 'delicious' and 'tasty':", my_model.wv.similarity('delicious', 'tasty'))
```

```
Example 1
Word2Vec Model: Similarity for words 'excellent' and 'outstanding': 0.55674857
My Model: Similarity between 'excellent' and 'outstanding': 0.6556493
Example 2
Word2Vec Model: King - Man + Woman =  queen
My Model: King - Man + Woman =  nurse
Example 3 ('delicious' and 'tasty' not in my model)
Word2Vec Model: Similarity for words 'delicious' and 'tasty': 0.873039
```

Simple Models: Perceptron and SVM

In [154…

```
# imports
import numpy as np
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
from sklearn.metrics import f1_score, precision_score, recall_score

# TFIDF
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
```

TFIDF Features

In [155…

```
from sklearn.model_selection import train_test_split
```

```
# TFIDF Features
tfidf_vector = TfidfVectorizer(max_features=5000)
tfidf_features = tfidf_vector.fit_transform(balanced_df['review'])

# Splitting data into train and test sets
x_train_tfidf, x_test_tfidf, y_train_tfidf, y_test_tfidf = train_test_split(
    tfidf_features,
    balanced_df['label'],
    test_size= 0.2,
    random_state=55
)
```

Word2Vec Features

In [156…

```
# Calculating the average Word2Vec for each review
def average_word2vec(review, model, dimension):
    avg_w2v = np.zeros((dimension,))
    num_words = 0
    for word in review:
        if word in model:
            avg_w2v += model[word]
            num_words += 1
    if num_words > 0:
        avg_w2v /= num_words
    return avg_w2v

# Splitting reviews into sentences for Word2Vec
sentences = balanced_df['review'].str.split().tolist()

# Convert reviews into feature vectors using average Word2Vec helper function
word2vec_features = np.array([average_word2vec(review, wv_model, 300) for review in sentences])

# Splitting into train and test sets
x_train_word2vec, x_test_word2vec, y_train_word2vec, y_test_word2vec = train_test_split(
    word2vec_features,
    balanced_df['label'],
    test_size= 0.2,
    random_state=55
)
```

Perceptron Model w/ TFIDF features and word2vec features

In [157…

```
# Perceptron on tfidf features
tfidf_perceptron = Perceptron(max_iter=1000, random_state=55)

# Fit training set into Perceptron Model
tfidf_perceptron.fit(x_train_tfidf, y_train_tfidf)

# Make predictions w/ testing set
tfidf_prediction = tfidf_perceptron.predict(x_test_tfidf)

# Report Precision, Recall, and f1-score
tfidf_test_precision = precision_score(y_test_tfidf, tfidf_prediction, average='binary')
tfidf_test_recall = recall_score(y_test_tfidf, tfidf_prediction, average='binary')
tfidf_test_f1 = f1_score(y_test_tfidf, tfidf_prediction, average='binary')

# Print
print(f"TF-IDF Perceptron Model~ Precision:{tfidf_test_precision:.4f}, Recall:{tfidf_test_recall:.4f}, F1-Score:{tfidf_test_f1:.4f}")


# Perceptron on word2vec features
word2vec_perceptron = Perceptron(max_iter=1000, random_state=55)

# Fit training set into Perceptron Model
word2vec_perceptron.fit(x_train_word2vec, y_train_word2vec)

# Make predictions w/ testing set
word2vec_prediction = word2vec_perceptron.predict(x_test_word2vec)

# Report Precision, Recall, and f1-score
word2vec_test_precision = precision_score(y_test_word2vec, word2vec_prediction, average='binary')
word2vec_test_recall = recall_score(y_test_word2vec, word2vec_prediction, average='binary')
word2vec_test_f1 = f1_score(y_test_word2vec, word2vec_prediction, average='binary')

# Print
print(f"Word2Vec Perceptron Model~ Precision:{word2vec_test_precision:.4f}, Recall:{word2vec_test_recall:.4f}, F1-Score:{word2vec_test_f1:.4f}")
```

```
TF-IDF Perceptron Model~ Precision:0.7969, Recall:0.8345, F1-Score:0.8153
Word2Vec Perceptron Model~ Precision:0.8374, Recall:0.7370, F1-Score:0.7840
```

SVM Model w/ TFIDF features and word2vec features

In [158…

```
# SVM on tfidf features
tfidf_svm = LinearSVC(dual=False, max_iter=1000, random_state=55)

# Fit training set into SVM classifier
tfidf_svm.fit(x_train_tfidf, y_train_tfidf)

# Predict on testing set
tfidf_svm_prediction = tfidf_svm.predict(x_test_tfidf)

# Report Precision, Recall, and f1-score
tfidf_svm_precision = precision_score(y_test_tfidf, tfidf_svm_prediction, average='binary')
tfidf_svm_recall = recall_score(y_test_tfidf, tfidf_svm_prediction, average='binary')
tfidf_svm_f1 = f1_score(y_test_tfidf, tfidf_svm_prediction, average='binary')

# Print
print(f"TF-IDF SVM Model~ Precision:{tfidf_svm_precision:.4f} Recall:{tfidf_svm_recall:.4f} F1-Score:{tfidf_svm_f1:.4f}")

# SVM on word2vec features
word2vec_svm = LinearSVC(dual=False, max_iter=1000, random_state=55)

# Fit training set into SVM classifier
word2vec_svm.fit(x_train_word2vec, y_train_word2vec)

# Predict on testing set
word2vec_svm_prediction = word2vec_svm.predict(x_test_word2vec)

# Report Precision, Recall, and f1-score
word2vec_svm_precision = precision_score(y_test_word2vec, word2vec_svm_prediction, average='binary')
word2vec_svm_recall = recall_score(y_test_word2vec, word2vec_svm_prediction, average='binary')
word2vec_svm_f1 = f1_score(y_test_word2vec, word2vec_svm_prediction, average='binary')

# Print
print(f"Word2Vec SVM Model~ Precision:{word2vec_svm_precision:.4f} Recall:{word2vec_svm_recall:.4f} F1-Score:{word2vec_svm_f1:.4f}")
```

```
TF-IDF SVM Model~ Precision:0.8554 Recall:0.8767 F1-Score:0.8659
Word2Vec SVM Model~ Precision:0.7926 Recall:0.8762 F1-Score:0.8323
```

My explanation:

- Deep learning models perform better with embeddings like Word2Vec than with sparse representations like TF-IDF
- Linear models like Perceptron and SVM might sometimes favor the discriminative power of TF-IDF

Feedforward Neural Networks

In [159…

```python
# import
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

from sklearn.metrics import accuracy_score
```

In [160…

```python
# Convert to PyTorch tensors
x_train_tensor = torch.FloatTensor(x_train_word2vec)
y_train_tensor = torch.LongTensor(y_train_word2vec.values)
x_test_tensor = torch.FloatTensor(x_test_word2vec)
y_test_tensor = torch.LongTensor(y_test_word2vec.values)

# Check before moving forward
print("Unique values in y_train_tensor:", torch.unique(y_train_tensor))
print("Unique values in y_test_tensor:", torch.unique(y_test_tensor))

print("Shape of X_train_tensor:", x_train_tensor.shape)
# print("Sample values from X_train_tensor:", X_train_tensor[:5])
print("Shape of X_test_tensor:", x_test_tensor.shape)
# print("Sample values from X_test_tensor:", X_test_tensor[:5])

# Re-map the values from [1, 2] to [0, 1] for proper training
y_train_tensor -= 1
y_test_tensor -= 1

print("Updated unique values in y_train_tensor:", torch.unique(y_train_tensor))
print("Updated unique values in y_test_tensor:", torch.unique(y_test_tensor))
```

```
Unique values in y_train_tensor: tensor([1, 2])
Unique values in y_test_tensor: tensor([1, 2])
Shape of X_train_tensor: torch.Size([79988, 300])
Shape of X_test_tensor: torch.Size([19998, 300])
Updated unique values in y_train_tensor: tensor([0, 1])
Updated unique values in y_test_tensor: tensor([0, 1])
```

In [161…

```python
# Create train loader with batch size 64
train_data = TensorDataset(x_train_tensor, y_train_tensor)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)

class MLP(nn.Module):
    def __init__(self, input_size, hidden1_size, hidden2_size, output_size):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_size, hidden1_size),
            nn.ReLU(),
            nn.Linear(hidden1_size, hidden2_size),
            nn.ReLU(),
            nn.Linear(hidden2_size, output_size),
        )

    def forward(self, x):
        return self.layers(x)

# Hyperparameters: two hidden layers, each with 50 and 5 nodes, respectively
input_size = 300
hidden1_size = 50
hidden2_size = 5
output_size = 2

# Initialize model, loss, and optimizer
model = MLP(input_size, hidden1_size, hidden2_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training loop
epochs = 10
for epoch in range(epochs):
    for _, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}")

# Testing
with torch.no_grad():
    test_outputs = model(x_test_tensor)
    _, predicted = test_outputs.max(1)
    accuracy = (predicted == y_test_tensor).sum().item() / len(y_test_word2vec)
    print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Epoch 1/10, Loss: 0.3715682029724121
Epoch 2/10, Loss: 0.32229456305503845
Epoch 3/10, Loss: 0.7002264857292175
Epoch 4/10, Loss: 0.38312360644340515
Epoch 5/10, Loss: 0.36629727482795715
Epoch 6/10, Loss: 0.35043641924858093
Epoch 7/10, Loss: 0.27408459782600403
Epoch 8/10, Loss: 0.24510405957698822
Epoch 9/10, Loss: 0.38804447650909424
Epoch 10/10, Loss: 0.23674869537353516
Test Accuracy: 84.89%
```

Concatenate the first 10 Word2Vec vectors for each review as the input feature

In [162…

```
def concat_first_10_word2vec(review, model, dimension):
    feature_vec = []
    for i in range(10):
        if i < len(review) and review[i] in model:
            feature_vec.extend(model[review[i]])
        else:
            feature_vec.extend(np.zeros((dimension,)))
    return feature_vec

sentences = balanced_df['review'].str.split().tolist()

# Convert reviews into feature vectors using concatenated Word2Vec
word2vec_10words_features = np.array([concat_first_10_word2vec(review, wv_model, 300) for review in sentences])
```

In [163…

```
# Split into train and test sets using word2vec_10words_features
x_train_10words, x_test_10words, y_train_10words, y_test_10words = train_test_split(
    word2vec_10words_features,
    balanced_df['label'],
    test_size= 0.2,
    random_state=55
)

# Convert to PyTorch tensors
x_train_tensor = torch.FloatTensor(x_train_10words)
y_train_tensor = torch.LongTensor(y_train_10words.values)
x_test_tensor = torch.FloatTensor(x_test_10words)
y_test_tensor = torch.LongTensor(y_test_10words.values)

# re-map the values from [1, 2] to [0, 1] for proper training
y_train_tensor -= 1
y_test_tensor -= 1
```

In [164…

```
# Create train loader with batch size 64
train_data = TensorDataset(x_train_tensor, y_train_tensor)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)

# Hyperparameters: two hidden layers, each with 50 and 5 nodes, respectively
class MLP(nn.Module):
    def __init__(self, input_size, hidden1_size, hidden2_size, output_size):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_size, hidden1_size),
            nn.ReLU(),
            nn.Linear(hidden1_size, hidden2_size),
            nn.ReLU(),
            nn.Linear(hidden2_size, output_size),
        )

    def forward(self, x):
        return self.layers(x)

# Hyperparameters: two hidden layers, each with 50 and 5 nodes, respectively
input_size = 3000
hidden1_size = 50
hidden2_size = 5
output_size = 2

# Initialize model, loss, and optimizer
model = MLP(input_size, hidden1_size, hidden2_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training loop
epochs = 10
for epoch in range(epochs):
    for _, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}")

# Testing
with torch.no_grad():
    test_outputs = model(x_test_tensor)
    _, predicted = test_outputs.max(1)
    accuracy = (predicted == y_test_tensor).sum().item() / len(y_test_10words)
    print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Epoch 1/10, Loss: 0.4302016794681549
Epoch 2/10, Loss: 0.3950132727622986
Epoch 3/10, Loss: 0.3557981252670288
Epoch 4/10, Loss: 0.44132062792778015
Epoch 5/10, Loss: 0.2974357604980469
Epoch 6/10, Loss: 0.32928821444511414
Epoch 7/10, Loss: 0.16482013463974
Epoch 8/10, Loss: 0.2622988820075989
Epoch 9/10, Loss: 0.14231359958648682
Epoch 10/10, Loss: 0.17618674039840698
Test Accuracy: 79.17%
```

My explanation:

- The test accuracy drops after concatenating the first 10 words of each review.
- This is because when you concatenate the first 10 Word2Vec vectors, you might be losing some information that is crucial for the classification task
- Word2Vec vectors capture semantic meaning. By taking only the first 10 words, the model might be missing out on important context that comes from the rest of the review

Recurrent Neural Network

In [165…

```
# imports
import torch.nn.functional as F
```

A little bit of preprocessing

In [166…

```
# NOTE: using x_train_word2vec, x_test_word2vec, y_train_word2vec, y_test_word2vec
```

```
# Maintain the sequence of vectors since RNN processes inputs one step at a time and maintains a hidden state across those steps
def get_word2vec_sequence(review, model, dimension):
    w2v_sequence = []
    for word in review:
        if word in model:
            w2v_sequence.append(model[word])
        else:
            w2v_sequence.append(np.zeros((dimension,)))  # Using a zero vector for unknown words
    return w2v_sequence

# Convert reviews into sequences of Word2Vec vectors
sentences = balanced_df['review'].str.split().tolist()
word2vec_sequences = [get_word2vec_sequence(review, wv_model, 300) for review in sentences]

# To feed your data into our RNN, limit the maximum review length to 10
# by truncating longer reviews and padding shorter reviews with a null value (0)
def pad_or_truncate_sequence(sequence, max_length):
    if len(sequence) > max_length:
        sequence = sequence[:max_length]
    else:
        while len(sequence) < max_length:
            sequence.append(np.zeros((300,)))  # Padding with zero vectors
    return sequence

padded_word2vec_sequences = np.array([pad_or_truncate_sequence(seq, 10) for seq in word2vec_sequences])

# Split into train and test sets using padded_word2vec_sequences
x_train_word2vec, x_test_word2vec, y_train_word2vec, y_test_word2vec = train_test_split(
    padded_word2vec_sequences,
    balanced_df['label'],
    test_size= 0.2,
    random_state=55
)
```

In [167…

```
# check before moving forward
print(x_train_word2vec.shape, x_test_word2vec.shape)
```

```
(79988, 10, 300) (19998, 10, 300)
```

In [168…

```
# Hyperparameters: one hidden layer with 10
input_size = 300
hidden_size = 10
output_size = 2
```

Simple RNN

In [169…

```
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleRNN, self).__init__()

        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), hidden_size).to(x.device)  # Initialize hidden state
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])  # Only take the last time step's output for classification
        return out

model = SimpleRNN(input_size, hidden_size, output_size)
```

In [170…

```
# Convert to PyTorch tensors
x_train_rnn_tensor = torch.FloatTensor(x_train_word2vec)
x_test_rnn_tensor = torch.FloatTensor(x_test_word2vec)
y_train_rnn_tensor = torch.LongTensor(y_train_word2vec.values)
y_test_rnn_tensor = torch.LongTensor(y_test_word2vec.values)

# print("Unique values in y_train_tensor:", torch.unique(y_train_rnn_tensor))
# print("Unique values in y_test_tensor:", torch.unique(y_test_rnn_tensor))

# re-map the values from [1, 2] to [0, 1] for proper training
y_train_rnn_tensor -= 1
y_test_rnn_tensor -= 1

# print("Updated unique values in y_train_tensor:", torch.unique(y_train_rnn_tensor))
# print("Updated unique values in y_test_tensor:", torch.unique(y_test_rnn_tensor))
```

In [171…

```
# Create train loader with batch size 64
train_data_rnn = TensorDataset(x_train_rnn_tensor, y_train_rnn_tensor)
train_loader_rnn = DataLoader(train_data_rnn, batch_size=64, shuffle=True)

# Initialize loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

epochs = 10
for epoch in range(epochs):
    for _, (data, target) in enumerate(train_loader_rnn):
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}")

with torch.no_grad():
    test_outputs = model(x_test_rnn_tensor)
    _, predicted = test_outputs.max(1)
    accuracy = (predicted == y_test_rnn_tensor).sum().item() / len(y_test_word2vec)
    print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Epoch 1/10, Loss: 0.505071222782135
Epoch 2/10, Loss: 0.4083254635334015
Epoch 3/10, Loss: 0.5449137091636658
Epoch 4/10, Loss: 0.38038599491119385
Epoch 5/10, Loss: 0.4555507004261017
Epoch 6/10, Loss: 0.3713168799877167
Epoch 7/10, Loss: 0.43283843994140625
Epoch 8/10, Loss: 0.4200085401535034
Epoch 9/10, Loss: 0.5686778426170349
Epoch 10/10, Loss: 0.4306006133556366
Test Accuracy: 78.41%
```

GRU

In [172…

```
class SimpleGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleGRU, self).__init__()

        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), hidden_size).to(x.device)
        out, _ = self.gru(x, h0)
        out = self.fc(out[:, -1, :])  # NOTE: Only taking the last time step's output for classification
        return out

# Initialize model, loss, and optimizer
model = SimpleGRU(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training
epochs = 10
for epoch in range(epochs):
    for _, (data, target) in enumerate(train_loader_rnn):
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}")

# Testing
with torch.no_grad():
    test_outputs = model(x_test_rnn_tensor)
    _, predicted = test_outputs.max(1)
    accuracy = (predicted == y_test_rnn_tensor).sum().item() / len(y_test_word2vec)
    print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Epoch 1/10, Loss: 0.28653085231781006
Epoch 2/10, Loss: 0.33367300033569336
Epoch 3/10, Loss: 0.2835990786552429
Epoch 4/10, Loss: 0.4465479254722595
Epoch 5/10, Loss: 0.19093047082424164
Epoch 6/10, Loss: 0.2864411175251007
Epoch 7/10, Loss: 0.28328919410705566
Epoch 8/10, Loss: 0.37917467951774597
Epoch 9/10, Loss: 0.2826038599014282
Epoch 10/10, Loss: 0.3535066246986389
Test Accuracy: 83.47%
```

LSTM

In [173…

```
class SimpleLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleLSTM, self).__init__()

        # Using LSTM instead of GRU
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # Initialize both hidden state and cell state for LSTM
        h0 = torch.zeros(1, x.size(0), hidden_size).to(x.device)
        c0 = torch.zeros(1, x.size(0), hidden_size).to(x.device)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])  # Only take the last time step's output for classification
        return out

# Model, Criterion, Optimizer Initialization
model = SimpleLSTM(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training
epochs = 10
for epoch in range(epochs):
    for _, (data, target) in enumerate(train_loader_rnn):
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}")

# Testing
with torch.no_grad():
    test_outputs = model(x_test_rnn_tensor)
    _, predicted = test_outputs.max(1)
    accuracy = (predicted == y_test_rnn_tensor).sum().item() / len(y_test_word2vec)
    print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Epoch 1/10, Loss: 0.3438650965690613
Epoch 2/10, Loss: 0.29859182238578796
Epoch 3/10, Loss: 0.23388096690177917
Epoch 4/10, Loss: 0.46848025918006897
Epoch 5/10, Loss: 0.29186028242111206
Epoch 6/10, Loss: 0.30844613909721375
Epoch 7/10, Loss: 0.30707797408103943
Epoch 8/10, Loss: 0.2040465623140335
Epoch 9/10, Loss: 0.33895012736320496
Epoch 10/10, Loss: 0.3702055513858795
Test Accuracy: 83.80%
```

My explanation:

- Both GRU and LSTM have significantly higher accuracies compared to the Simple RNN.
- This underscores the effectiveness of gating mechanisms in capturing long-term dependencies and mitigating issues like vanishing and exploding gradients that simple RNNs suffer from.