Applied Mathematics for Deep Learning I - Term Project

Group Members: Andre Graham, Adnan Sheik, Majed Qarmout, Yonotan Girma, Younes Bensassi-nour

1. The Business Problem: Understanding Customer Reviews

Understanding customer sentiment is essential for businesses to effectively evaluate the public perception surrounding their products. Though this can be easily gauged on a platform such as Amazon that includes the user's rating out of five in the customer's review, platforms such as YouTube and Twitter provide more of a challenge since there is no inclusion of a rating. For companies to gain a comprehensive understanding of public perception surrounding their products, it is vital that they analyze customer sentiment across all platforms. Our model aims to determine the sentiment of unstructured text, facilitating companies to better understand public perception of their products. This will allow them to tailor their marketing strategies, improve their current and future product offerings, and enhance customer satisfaction.

2. About the Data

The dataset used to train our model is the "Amazon Product Review" dataset from Kaggle. It has 10 features: Id, ProductId, UserId, ProfileName, HelpfulnessNumerator, HelpfulnessDenominator, Score, Time, Summary, and Text. For the purposes of our model, the two key variables are Score, which will be used to create the sentiment labels, and Text, which will be used as the input for our model. This is because, for the use case of our model, the other features will not be available as inputs. The Text column contains the text left in the review. The data was gathered through web scraping; however, the exact web scraping method is unknown, and therefore there may be a bias toward more popular products.

3. Our Approach

Our team's approach is to fine-tune an existing language model with product reviews. The goal is to leverage an existing sentiment analysis model that already contains a deep understanding of natural language. We chose to use the RoBERTa model (robustly optimized BERT pretraining approach), which we accessed through the Hugging Face Transformers library. The issue with using a pretrained model such as RoBERTa is that it is trained on a variety of data, many of which are structured significantly differently from a product review on social media. We account for this by fine-tuning the model using the Amazon product reviews data, with the goal of increasing the model's accuracy when evaluating product reviews. We start by fine-tuning the "distilroberta-base" model by freezing all its layers and adding a feedforward network to the end of the model. Throughout this process, we fine-tune the hyperparameters, in total creating 12 potential models. We then compare this to fine-tuning the last 3 encoding layers of the "distilroberta-base" model, which has 6 encoding layers. Next, we pick the model with the best accuracy and create an inference pipeline. Finally, we list the limitations of our model and have reflections for each team member.

4. Show and plot dataset and apply data analysis on dataset (visualizations complete, data analysis not really)

One of the most important aspects of the dataset to understand is the distribution of the score. The score is the rating out of 5 that the user gave the product and will be used to create the labels to train our model. Figure 1 shows the distribution of the scores.

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Loading the dataset
df = pd.read_csv('Data/Reviews.csv')

# Setting plot size
plt.figure(figsize=(12, 6))

# Plotting the histogram
plt.hist(df['Score'], bins=5, edgecolor='black')

# Set up the labels and axis
plt.title('Figure 1: Score Distribution')
plt.xlabel('Score')
plt.ylabel('Frequency')
plt.xticks(range(1, 6))

# Show the plot
plt.show()
```

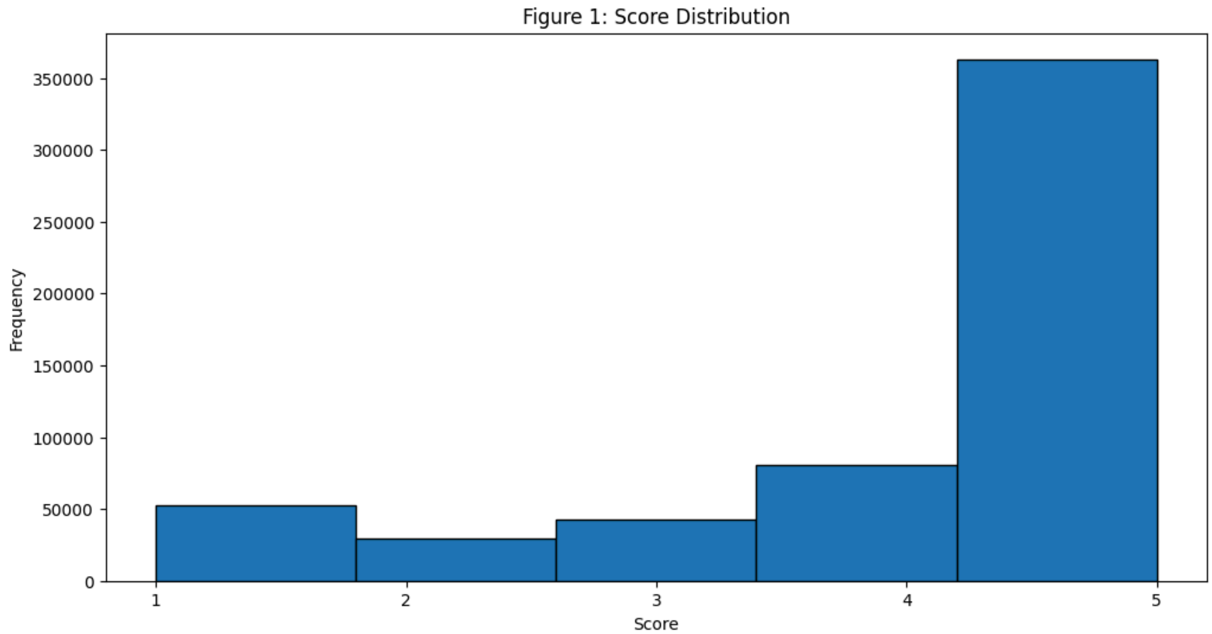


Figure 1 shows that the distribution of the scores is not uniform, which will have to be taken into account during preprocessing. This means that when we are creating our labels, we must either remove samples from categories with higher frequencies or add samples to the categories with low frequencies. Given we have limited computational resources, the more reasonable approach is to reduce the size of the dataset rather than increase it.

Next, we visualize a word cloud of the text in the reviews. This will give us a general understanding of the text content of the reviews, as well as reveal any problematic patterns found in the text. This can be seen in figure 2.

In [3]:
```
from wordcloud import WordCloud

# Word cloud for the text of the user reviews
text = ' '.join(df['Text'].astype(str))
wordcloud = WordCloud(width=800, height=400, background_color='white').gener


# Set the figure size for better readability
plt.figure(figsize=(12, 6))

# Plot the wordcloud
plt.imshow(wordcloud, interpolation='bilinear')

# Set up axis and labels
plt.axis('off')
plt.title('Figure 2: Word Cloud of Reviews', fontsize = 18)

# Show the plot
plt.show()
```

Figure 2: Word Cloud of Reviews

Figure 2 immediatly reveals a problem with the text in our dataset, which is the existence of tags. This is the reason why "br" is written twice in such large letters. It also makes sense that words such as love, favourite, and treat are all included in this word cloud given the skewed distribution of the ratings. These two issues will be addressed in our data preprocessing.

Finally, Figure 3 shows the distribution of the lengths of the reviews in the dataset.

In [4]:
```python
# Calculate the number of words in each review
df['ReviewLength'] = df['Text'].apply(lambda x: len(x.split()))

# Set the figure size
plt.figure(figsize=(12, 6))

# Plot the histogram
plt.hist(df['ReviewLength'], bins=50, color='skyblue', edgecolor='black')

# Add titles and labels
plt.title('Figure 3: Distribution of Review Lengths', fontsize=18)
plt.xlabel('Number of Words', fontsize=14)
plt.ylabel('Frequency', fontsize=14)

# Add grid lines for easier interpretation
plt.grid(axis='y', alpha=0.75)

# Show the plot
plt.show()
```

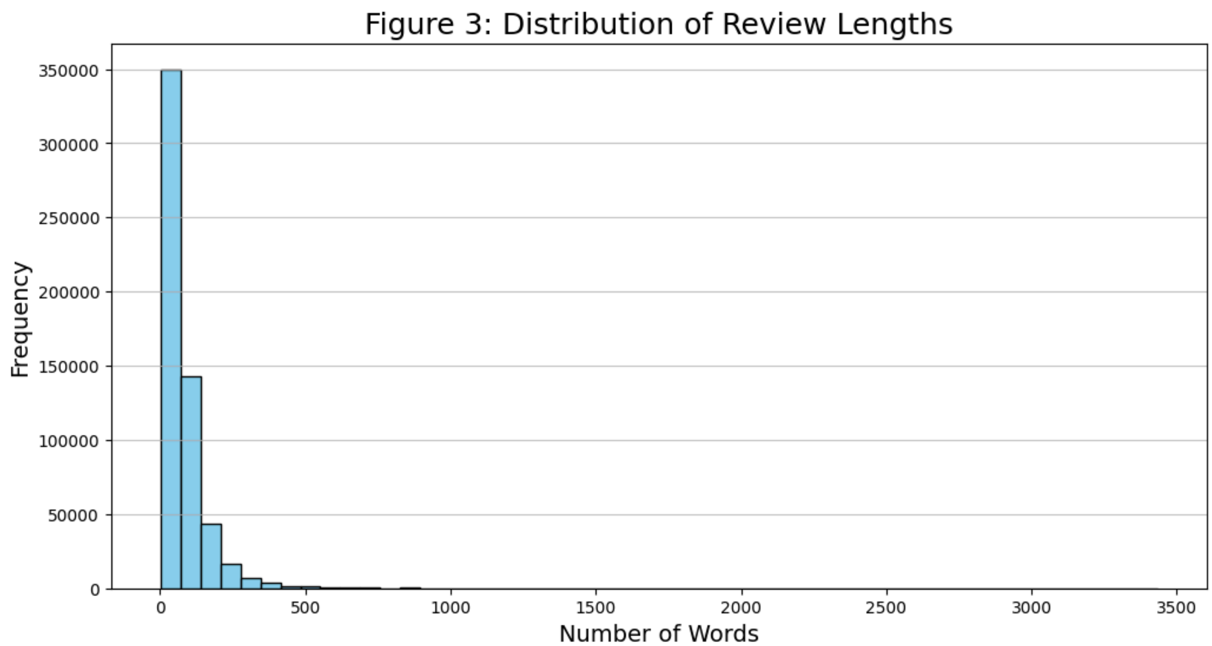


Figure 3 shows that that the distribution of the lengths of reviews is heavily right skewed. Since the goal of the model is to gain insight into the sentiment of customers posting on social media posts which tend to be quite short (think YouTube comments or Tweets), it is reasonable to conclude that the lengths of the text is representitive of the content that the model will be used on.

Though there are additional features included in the reviews dataset, the goal of this model is to categorize unstructured text from platforms such as Twitter and YouTube that do not contain information such as the helpfulness reviews or the product ID. Therefore, for the purposes of our model, we will only be using the text content of the reviews as the input for fine-tuning our model and the score for creating the labels.

5. Our Pipeline

Provide a **detailed overview** of the pipeline for the project. This pipeline should include:

- **Data preprocessing steps**: How are you cleaning and preparing the data?
- **Modeling process**: How are you selecting and training the model?
- **Post-processing**: What steps will you take after obtaining model predictions?

5.1 Data Preprocessing

The first step in the data pre-processing is to remove all the features that we will not be using for fine-tuning our model.

In [5]:
```python
# Make a backup of the original dataframe
df_backup = df.copy()

# List of columns to keep
columns_to_keep = ['Text', 'Score']

# Drop all other columns
df_selected = df.drop(columns=[col for col in df.columns if col not in colum

# Display the first few rows to verify
print(df_selected.head())
```

```
   Score                                               Text
0      5  I have bought several of the Vitality canned d...
1      1  Product arrived labeled as Jumbo Salted Peanut...
2      4  This is a confection that has been around a fe...
3      2  If you are looking for the secret ingredient i...
4      5  Great taffy at a great price.  There was a wid...
```

Next, we want our model to predict whether a review is negative, neutral, or positive. To do this, we make a score of 1 or 2 represent a review with negative sentiment, a score of 3 represent a review with neutral sentiment, and a review of 4 or 5 to represent a review with positive sentiment.

In [6]:
```python
import seaborn as sns

# Function to map 'Score' to 'label' and create a bar chart of label distrib
def plot_sentiment_distribution(df, score_column='Score', label_column='labe
    """
    Maps the score to sentiment labels, creates the 'label' column, and plot

    Parameters:
    df (pd.DataFrame): The input dataset as a pandas DataFrame.
    score_column (str): The name of the column containing the score data.
    label_column (str): The name of the column to store the sentiment label

    Returns:
    None: Displays a bar chart showing the distribution of sentiment labels.
    """
    # Set the background of the plot to a white grid
    sns.set(style="whitegrid")

    # Define the mapping from Score to label
    score_to_label = {
        1: 0,  # Negative sentiment
        2: 0,  # Negative sentiment
        3: 1,  # Neutral sentiment
        4: 2,  # Positive sentiment
        5: 2   # Positive sentiment
    }

    # Map 'Score' to 'label' column
    df[label_column] = df[score_column].map(score_to_label)
```

```
    # Count the occurrences of each label
    label_counts = df[label_column].value_counts().sort_index()

    # Define label names for better readability
    label_names = ['Negative', 'Neutral', 'Positive']

    # Create the bar chart
    plt.figure(figsize=(8, 6))
    bars = plt.bar(label_names, label_counts, color=['red', 'gray', 'green']

    # Add titles and labels
    plt.title(title, fontsize=16)
    plt.xlabel('Sentiment', fontsize=14)
    plt.ylabel('Number of Reviews', fontsize=14)

    # Add count labels on top of each bar
    for bar in bars:
        height = bar.get_height()
        plt.text(bar.get_x() + bar.get_width()/2, height + 5,
                 f'{int(height)}', ha='center', va='bottom', fontsize=12)

    # Show the plot
    plt.tight_layout()
    plt.show()

# Example usage:
# Assuming you have a DataFrame named df_selected with a 'Score' column
plot_sentiment_distribution(df_selected, score_column='Score', label_column=
```

Figure 4: Distribution of Sentiment Labels

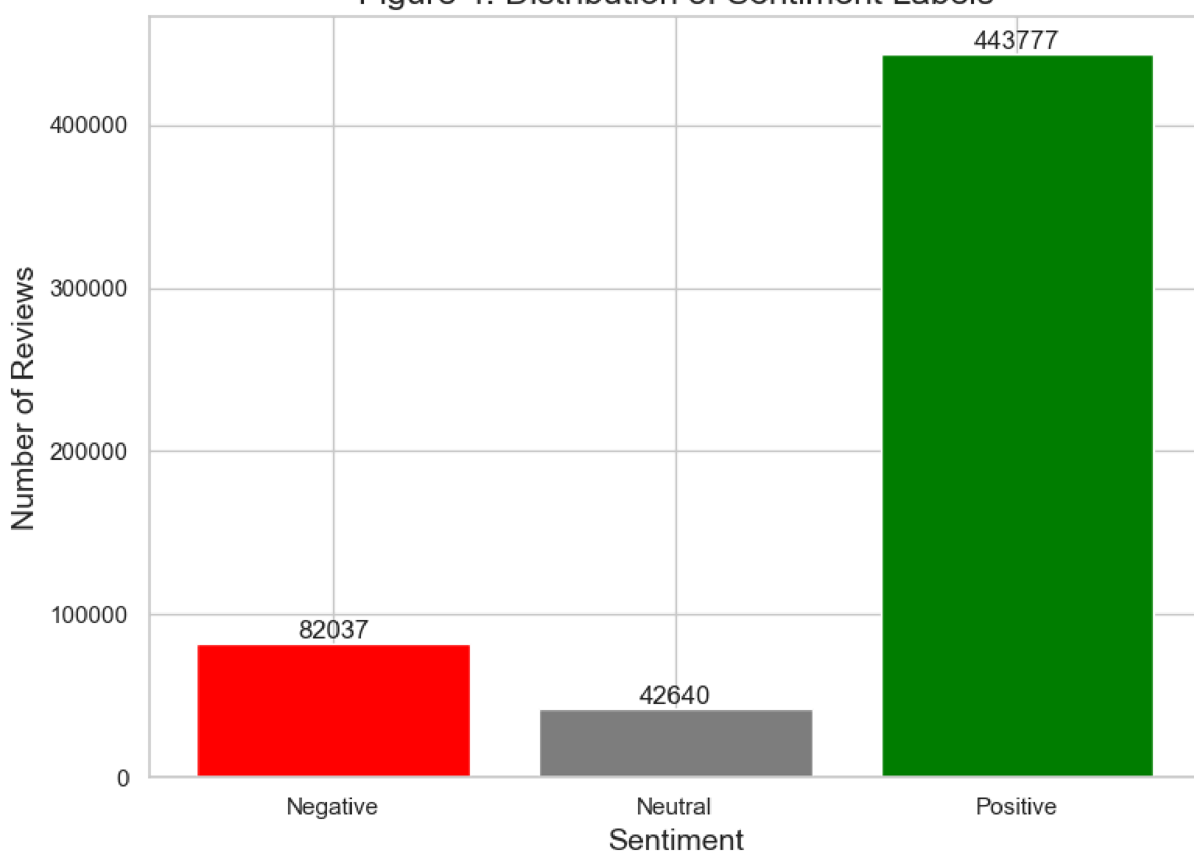
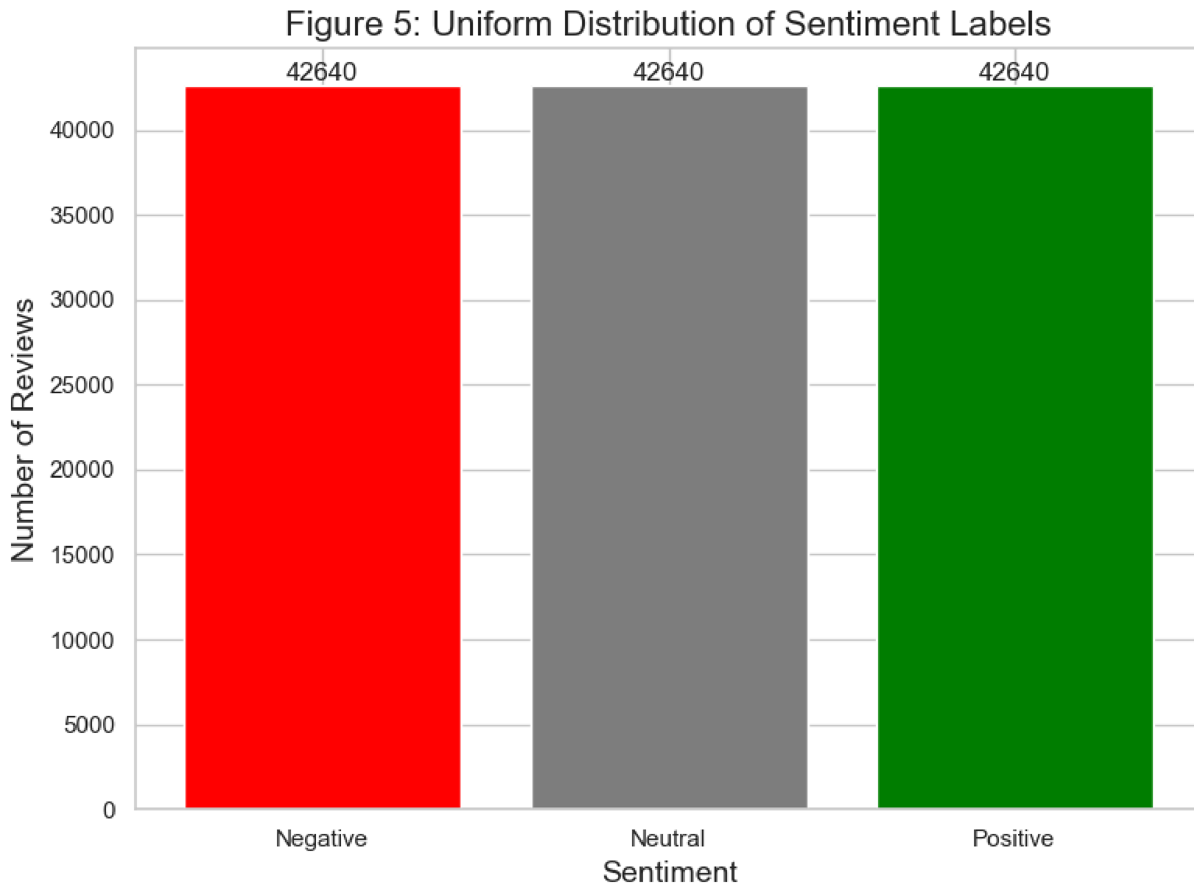


Figure 4 shows the distribution of each label. This plot illustrates the already known issue that we do not have a uniform distribution of the different review sentiments. We fix this by randomly selecting 42,640 samples from the "Positive" sentiment class and the "Negative" sentiment class and dropping the other rows. This fixes two problems for us. The first is the already mentioned non-uniform distribution of the sentiments. If we show our model signficantly more reviews with positive sentiments as opposed to negative or neutral sentiments, the final model will perform poorly when classifying new reviews with neutral or negative sentiment. The second issue it resolves is it reduces the size of our very large dataset. Due to computational limitations, it would be infeasible to use all the data in the original dataset.

In [7]:

```
# Define the target number of samples per sentiment
target_count = 42640

# Set a random seed for reproducibility
random_state = 42

# Sample equal number of reviews for each sentiment
df_balanced = df_selected.groupby('label').apply(
    lambda x: x.sample(n=target_count, random_state=random_state)
).reset_index(drop=True)

# Shuffle the balanced DataFrame
df_balanced = df_balanced.sample(frac=1, random_state=random_state).reset_ir

# Plot the Sentiment-Distribution
plot_sentiment_distribution(df_balanced, score_column='Score', label_column=
```

```
/var/folders/yl/842rvw2j7mq5t40qz5hg61fh0000gn/T/ipykernel_56877/3387499048.
py:8: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping co
lumns. This behavior is deprecated, and in a future version of pandas the gr
ouping columns will be excluded from the operation. Either pass `include_gro
ups=False` to exclude the groupings or explicitly select the grouping column
s after groupby to silence this warning.
  df_balanced = df_selected.groupby('label').apply(
```



Now that the labels quantities in the dataset have a uniform distribution, the next step is to fix the issue of unwanted characters being included in the text.

In [8]:
```python
import re
# Define a function to clean text
def clean_text(text):
    # Remove HTML tags
    text = re.sub(r'<.*?>', '', text)
    # Remove leading and trailing whitespace
    text = text.strip()
    # Replace multiple spaces with a single space
    text = re.sub(r'\s+', ' ', text)
    return text

# Apply the cleaning function to the 'Text' column
df_balanced['Text'] = df_balanced['Text'].apply(clean_text)

# Create a word cloud to confirm adjustments of text worked correctly
text = ' '.join(df_balanced['Text'].astype(str))
wordcloud = WordCloud(width=800, height=400, background_color='white').gener

# Set the figure size for better readability
plt.figure(figsize=(12, 6))

# Plot the wordcloud
plt.imshow(wordcloud, interpolation='bilinear')
```

```python
# Set up axis and labels
plt.axis('off')
plt.title('Figure 6: Word Cloud of Cleaned Reviews', fontsize = 18)

# Show the plot
plt.show()
```

Figue 6 confirms that the desired changes to the dataset have been made. We now define the tokenizer that we will use for our model.

In [9]:
```python
# Setting up the model and tokenizer
from transformers import TFRobertaForSequenceClassification, RobertaTokenize

# Load the pre-trained tokenizer and sentiment fine-tuned model
tokenizer = RobertaTokenizer.from_pretrained('distilroberta-base')
model = TFRobertaForSequenceClassification.from_pretrained('distilroberta-ba
```

```
/opt/anaconda3/envs/venv_py311/lib/python3.11/site-packages/transformers/tok
enization_utils_base.py:1601: FutureWarning: `clean_up_tokenization_spaces`
was not set. It will be set to `True` by default. This behavior will be depr
acted in transformers v4.45, and will be then set to `False` by default. For
more details check this issue: https://github.com/huggingface/transformers/i
ssues/31884
  warnings.warn(
All model checkpoint layers were used when initializing TFRobertaForSequence
Classification.

All the layers of TFRobertaForSequenceClassification were initialized from t
he model checkpoint at cardiffnlp/twitter-roberta-base-sentiment.
If your task is similar to the task the model of the checkpoint was trained
on, you can already use TFRobertaForSequenceClassification for predictions w
ithout further training.
```

The RoBERTa model that we will fine-tune has a maximum input length of 512 tokens. In this next step, we drop any rows that have more than 512 tokens to account for the fact that the RoBERTa model that we are using takes a maximum of 512 tokens. The results of this can be seen in Figure 6.

In [10]:
```
# Function to tokenize text without truncation (to check token length)
def count_tokens(text):
    tokens = tokenizer.encode(text, truncation=False)
    return len(tokens)

# Apply the token counting to the dataset
df_balanced['num_tokens'] = df_balanced['Text'].apply(count_tokens)

# Filter out rows where the token count exceeds max_length
max_length = 512
df_balanced_filtered = df_balanced[df_balanced['num_tokens'] <= max_length].

plot_sentiment_distribution(df_balanced_filtered, score_column='Score', labe
```

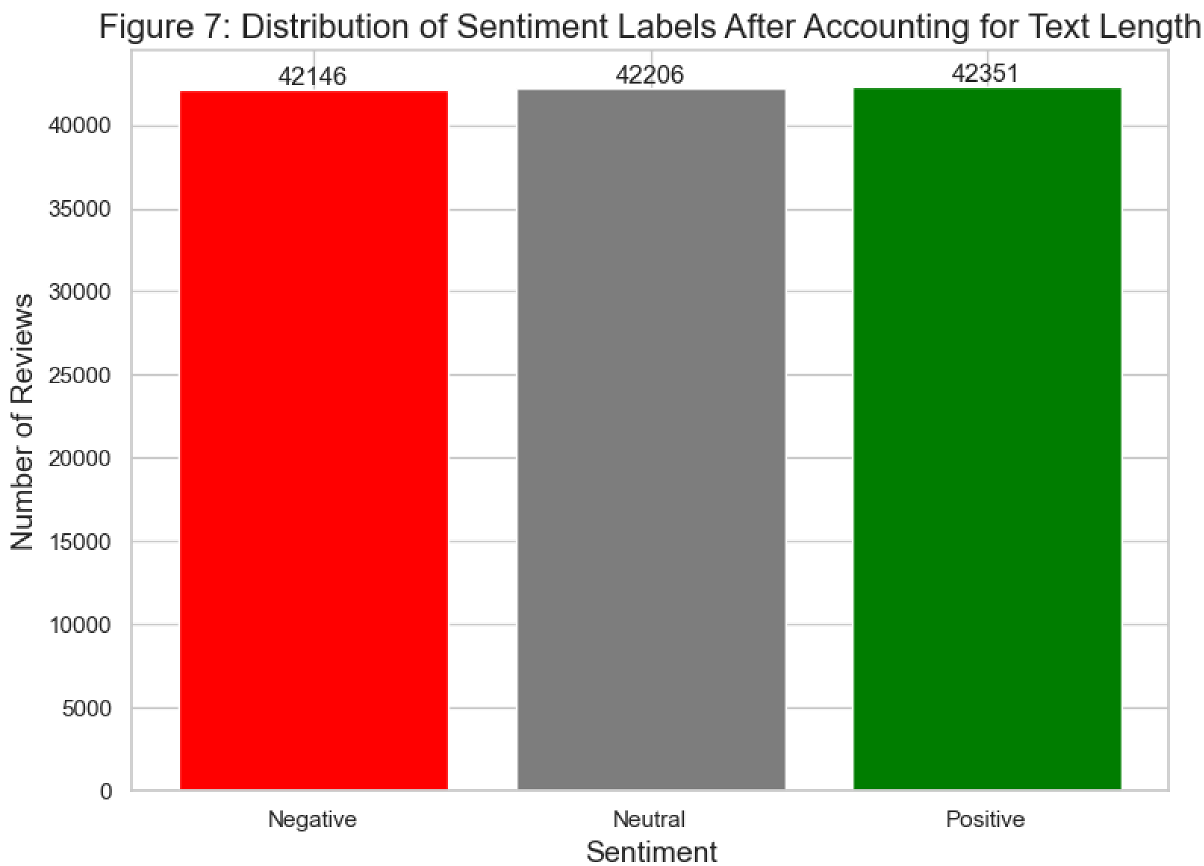
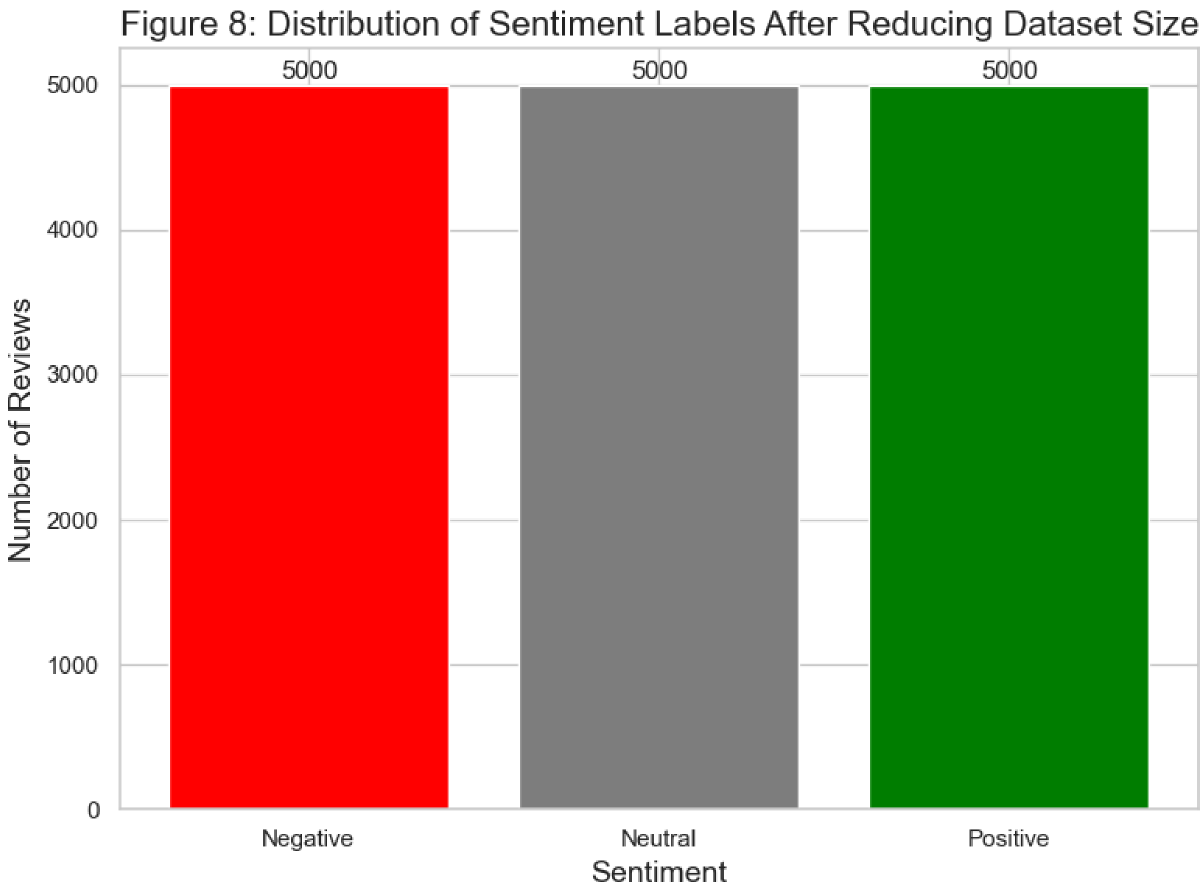


Figure 7 shows that this process had no negative effects on the distribution of sentiments. For the purpose's of training efficiency, we will work with a significantly smaller subset of the data so we can effecitvely experiemtn with the model. To do this, we will reduce each label to have 5000 rows. Once we have a model that starts seeing good performance, we will scale up the data to try to achieve maximum accuracy.

In [45]:
```
# Back up the full sized cleaned dataset
full_cleaned_df = df_balanced_filtered.copy()
```

```
experimenting_df = df_balanced_filtered.groupby('label').sample(n=5000, ranc

plot_sentiment_distribution(experimenting_df, score_column='Score', label_cc
```



Next, we tokenize the text, split the data into training and testing sets, and convert the resulting data into Tensorflow datasets.

In [ ]:
```
# Load the dataset
df = pd.read_csv('Data/Reviews.csv')

# Map 'Score' to sentiment labels
def map_score_to_label(score):
    if score in [1, 2]:
        return 0  # Negative
    elif score == 3:
        return 1  # Neutral
    else:  # 4 or 5
        return 2  # Positive

df['label'] = df['Score'].apply(map_score_to_label)
```

In [ ]:
```
# Balance the dataset by sampling 5000 reviews per class
df_balanced = df.groupby('label').apply(lambda x: x.sample(n=5000, random_st

# Clean the text
def clean_text(text):
```

```
    # Remove HTML tags
    text = re.sub(r'<.*?>', '', text)
    # Remove non-ASCII characters
    text = text.encode('ascii', 'ignore').decode('utf-8')
    # Remove URLs
    text = re.sub(r'http\S+', '', text)
    # Remove special characters and numbers
    text = re.sub(r'[^A-Za-z\s]', '', text)
    # Convert to lowercase
    text = text.lower()
    # Remove extra spaces
    text = re.sub(r'\s+', ' ', text).strip()
    return text

df_balanced['cleaned_text'] = df_balanced['Text'].apply(clean_text)
```

In [ ]:

```
# Load the tokenizer
tokenizer = RobertaTokenizer.from_pretrained('distilroberta-base')

# Tokenize the text
def tokenize_function(texts):
    return tokenizer(
        texts.tolist(),
        padding='max_length',
        truncation=True,
        max_length=128,
        return_tensors='tf'
    )

tokenized_texts = tokenize_function(df_balanced['cleaned_text'])

# Convert tensors to numpy arrays
input_ids = tokenized_texts['input_ids'].numpy()
attention_mask = tokenized_texts['attention_mask'].numpy()
labels = df_balanced['label'].values

# Clear unused variables to save memory
del tokenized_texts, df_balanced
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings t
ab (https://huggingface.co/settings/tokens), set it as secret in your Google
Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
public models or datasets.
  warnings.warn(
tokenizer_config.json:   0%|          | 0.00/25.0 [00:00<?, ?B/s]
vocab.json:   0%|          | 0.00/899k [00:00<?, ?B/s]
merges.txt:   0%|          | 0.00/456k [00:00<?, ?B/s]
tokenizer.json:   0%|          | 0.00/1.36M [00:00<?, ?B/s]
config.json:   0%|          | 0.00/480 [00:00<?, ?B/s]
```

```
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_bas
e.py:1601: FutureWarning: `clean_up_tokenization_spaces` was not set. It wil
l be set to `True` by default. This behavior will be depracted in transforme
rs v4.45, and will be then set to `False` by default. For more details check
this issue: https://github.com/huggingface/transformers/issues/31884
  warnings.warn(
```

In [ ]:

```
# Split into train, validation, and test sets
train_input_ids, temp_input_ids, train_labels, temp_labels, train_attention_
    input_ids,
    labels,
    attention_mask,
    test_size=0.3,  # 70% train, 30% temp
    random_state=42
)

# Further split temp into validation and test sets
val_input_ids, test_input_ids, val_labels, test_labels, val_attention_mask,
    temp_input_ids,
    temp_labels,
    temp_attention_mask,
    test_size=0.5,  # 15% validation, 15% test
    random_state=42
)

# Clear unused variables to save memory
del input_ids, attention_mask, labels, temp_input_ids, temp_labels, temp_att

# Function to create TensorFlow datasets
def create_tf_dataset(input_ids, attention_mask, labels, batch_size):
    dataset = tf.data.Dataset.from_tensor_slices((
        {'input_ids': input_ids, 'attention_mask': attention_mask},
        labels
    ))
    dataset = dataset.batch(batch_size)
    return dataset
```

In [ ]:

```
import tensorflow as tf

# Build the model function
def build_model(base_model, activation_function='relu', dropout_rate=0.2, us
    # Define inputs
    input_ids = tf.keras.Input(shape=(128,), dtype=tf.int32, name='input_ids
    attention_mask = tf.keras.Input(shape=(128,), dtype=tf.int32, name='atte

    # Get outputs from the base model
    outputs = base_model(input_ids, attention_mask=attention_mask)
    cls_token = outputs.last_hidden_state[:, 0, :]

    if additional_layers:
      # Add new encoding layers
      x = tf.keras.layers.Dense(512)(cls_token)
      if use_batchnorm:
          x = tf.keras.layers.BatchNormalization()(x)
      x = tf.keras.layers.Activation(activation_function)(x)
```

```python
        x = tf.keras.layers.Dropout(dropout_rate)(x)

        x = tf.keras.layers.Dense(256)(x)
        if use_batchnorm:
            x = tf.keras.layers.BatchNormalization()(x)
        x = tf.keras.layers.Activation(activation_function)(x)
        x = tf.keras.layers.Dropout(dropout_rate)(x)

        x = tf.keras.layers.Dense(128)(x)
        if use_batchnorm:
            x = tf.keras.layers.BatchNormalization()(x)
        x = tf.keras.layers.Activation(activation_function)(x)
        x = tf.keras.layers.Dropout(dropout_rate)(x)

        x = tf.keras.layers.Dense(64)(x)
        if use_batchnorm:
            x = tf.keras.layers.BatchNormalization()(x)
        x = tf.keras.layers.Activation(activation_function)(x)
        x = tf.keras.layers.Dropout(dropout_rate)(x)

        output = tf.keras.layers.Dense(3, activation='softmax')(x)
    else:
        output = tf.keras.layers.Dense(3, activation='softmax')(cls_token)

    # Build the model
    model = tf.keras.Model(inputs=[input_ids, attention_mask], outputs=outpu

    return model
```

In [ ]:

```python
# Function to get optimizer with specified learning rate
def get_optimizer(name, learning_rate):
    if name == 'adam':
        return tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif name == 'adamw':
        try:
            # For TensorFlow 2.11 and above
            return tf.keras.optimizers.experimental.AdamW(learning_rate=lear
        except AttributeError:
            # For earlier versions, use the experimental namespace
            return tf.keras.optimizers.Adam(learning_rate=learning_rate)  #
    else:
        raise ValueError(f'Unsupported optimizer: {name}')
```

In [ ]:

```python
import random
from sklearn.metrics import confusion_matrix, classification_report

# Define hyperparameter options
optimizers = ['adam', 'adamw']
learning_rates = [1e-5, 3e-5]
batch_sizes = [16, 32]
dropout_rates = [0.1, 0.2]
activation_functions = ['relu', 'tanh']
use_batchnorm_options = [True]

# Limit the number of configurations to manage computational resources
```

```python
max_configs = 6 # Adjust this number based on your resources
configs = []

for _ in range(max_configs):
    config = {
        'optimizer': random.choice(optimizers),
        'learning_rate': random.choice(learning_rates),
        'batch_size': random.choice(batch_sizes),
        'dropout_rate': random.choice(dropout_rates),
        'activation_function': random.choice(activation_functions),
        'use_batchnorm': random.choice(use_batchnorm_options)
    }
    configs.append(config)

# Load the base RoBERTa model and freeze layers
base_model = TFRobertaModel.from_pretrained('distilroberta-base')
for layer in base_model.layers:
    layer.trainable = False

histories = []
results = []
```

```
Some weights of the PyTorch model were not used when initializing the TF 2.0
model TFRobertaModel: ['lm_head.bias', 'lm_head.dense.weight', 'lm_head.laye
r_norm.bias', 'lm_head.layer_norm.weight', 'lm_head.dense.bias']
- This IS expected if you are initializing TFRobertaModel from a PyTorch mod
el trained on another task or with another architecture (e.g. initializing a
TFBertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing TFRobertaModel from a PyTorch
model that you expect to be exactly identical (e.g. initializing a TFBertFor
SequenceClassification model from a BertForSequenceClassification model).
All the weights of TFRobertaModel were initialized from the PyTorch model.
If your task is similar to the task the model of the checkpoint was trained
on, you can already use TFRobertaModel for predictions without further train
ing.
```

In [ ]:

```python
def run_model(num_epochs, patience, configurations, base_model, additional_l

  # Initialize variables to keep track of the best model
  best_accuracy = 0
  best_model = None
  best_config = None
  best_y_pred = None

  # Loop over configurations
  for i, config in enumerate(configurations):
      print(f"Training configuration {i+1}/{len(configurations)}: {config}")

      # Build the model
      model = build_model(
          base_model,
          activation_function=config['activation_function'],
          dropout_rate=config['dropout_rate'],
          use_batchnorm=config['use_batchnorm'],
          additional_layers= additional_layers
      )
```

```
    # Get optimizer with specified learning rate
    optimizer = get_optimizer(config['optimizer'], learning_rate=config['l

    # Compile the model
    model.compile(
        optimizer=optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    # Create datasets with specified batch size
    batch_size = config['batch_size']
    train_dataset = create_tf_dataset(train_input_ids, train_attention_mas
    val_dataset = create_tf_dataset(val_input_ids, val_attention_mask, val
    test_dataset = create_tf_dataset(test_input_ids, test_attention_mask,

    # Set up callbacks
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',

    # Fit the model using the validation set
    history = model.fit(
        train_dataset,
        epochs=num_epochs,
        validation_data=val_dataset,
        callbacks=[early_stopping],
        verbose=1
    )

    # Record the history and config
    histories.append((history, config))

    # Evaluate the final model on the test set
    loss, accuracy = model.evaluate(test_dataset)
    print(f"Test accuracy: {accuracy}")

    # Store results
    results.append({
        'config': config,
        'accuracy': accuracy,
        'loss': loss,
    })

    # Generate predictions for the test set
    y_pred_probs = model.predict(test_dataset)
    y_pred = np.argmax(y_pred_probs, axis=1)

    # Flatten test labels
    y_true = test_labels

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)

    # Classification report
    report = classification_report(y_true, y_pred, target_names=['Negative
    print(f"Classification Report for Configuration {i+1}:\n{report}")
```

```
    # Plot confusion matrix
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Negat
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.title(f'Confusion Matrix for Configuration {i+1}')
    plt.show()

    # Store results
    results.append({
        'config': config,
        'accuracy': accuracy,
        'loss': loss,
        'confusion_matrix': cm,
        'classification_report': report
    })

    # If current model is better, save it
    if accuracy > best_accuracy:
        print("New best model found!")
        # Clear previous best model from memory if exists
        if best_model is not None:
            tf.keras.backend.clear_session()
            del best_model

        best_accuracy = accuracy
        best_model = model
        best_config = config
        best_y_pred = y_pred
        best_y_true = y_true
        # Also store confusion matrix and classification report
        best_cm = confusion_matrix(y_true, y_pred)
        best_report = classification_report(y_true, y_pred, target_names=[
    else:
        # Clear current model from memory
        tf.keras.backend.clear_session()
        del model
return best_accuracy, best_model, best_config, best_y_pred, best_y_true, b
```

6. Training the Model

Below is the code that shows the training process for the first 6 models. After each model is trained, the confusion matrix is printed for the model, along with the combination of hyperparameters used to create it. The first set of 6 models was trained with batch normalization, the second set of 6 models was trained without batch normalization, and the final model was made by freezing only the first 3 encoding layers and unfreezing the 3 closest to the output layer.

6.1 Batch-Normalized Models

6.1.1 Batch-Normalized Models: Training

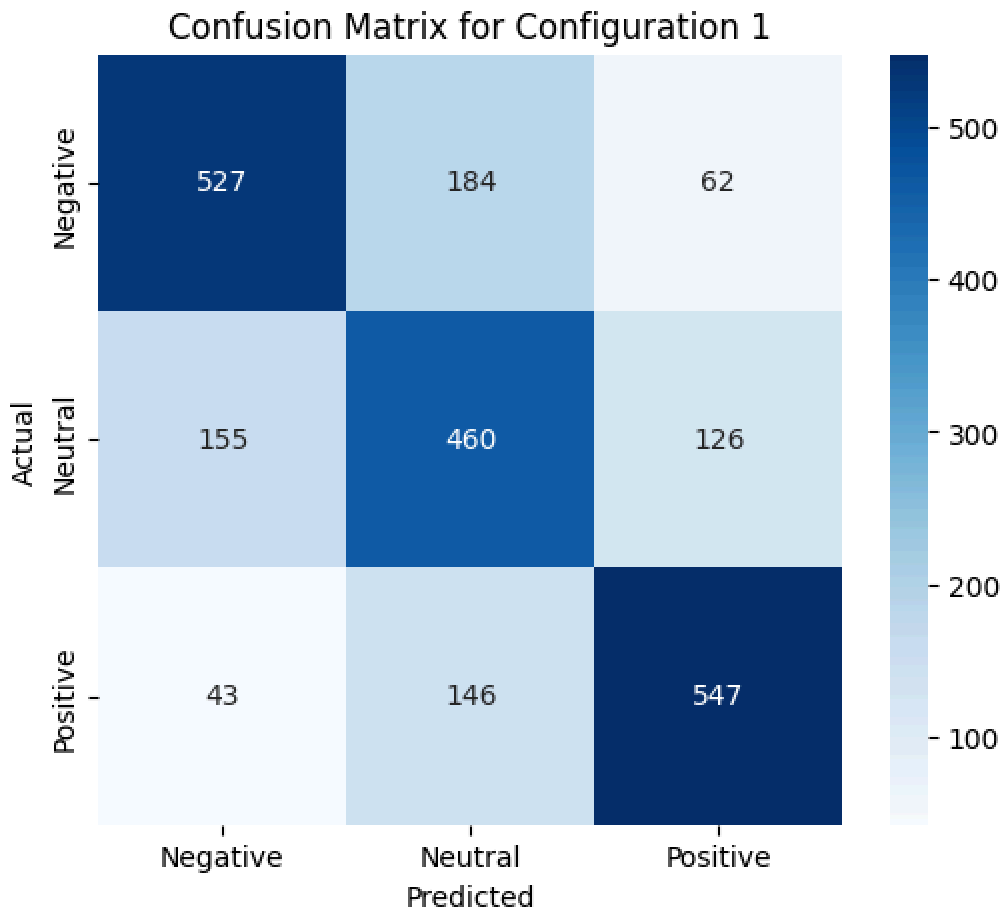
In [ ]:
```
# Run the model
best_accuracy, best_model, best_config, best_y_pred, best_y_true, best_cm, b
```

```
Training configuration 1/6: {'optimizer': 'adam', 'learning_rate': 1e-05, 'b
atch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'relu', 'use_bat
chnorm': True}
Epoch 1/80
657/657 [==============================] - 81s 100ms/step - loss: 1.3414 - a
ccuracy: 0.3602 - val_loss: 1.0844 - val_accuracy: 0.3964
Epoch 2/80
657/657 [==============================] - 63s 96ms/step - loss: 1.2345 - ac
curacy: 0.3923 - val_loss: 1.0120 - val_accuracy: 0.4631
Epoch 3/80
657/657 [==============================] - 63s 96ms/step - loss: 1.1664 - ac
curacy: 0.4189 - val_loss: 0.9703 - val_accuracy: 0.5213
Epoch 4/80
657/657 [==============================] - 64s 97ms/step - loss: 1.1236 - ac
curacy: 0.4438 - val_loss: 0.9312 - val_accuracy: 0.5707
Epoch 5/80
657/657 [==============================] - 66s 101ms/step - loss: 1.0794 - a
ccuracy: 0.4653 - val_loss: 0.8963 - val_accuracy: 0.6018
Epoch 6/80
657/657 [==============================] - 64s 97ms/step - loss: 1.0318 - ac
curacy: 0.4993 - val_loss: 0.8755 - val_accuracy: 0.6129
Epoch 7/80
657/657 [==============================] - 63s 97ms/step - loss: 1.0052 - ac
curacy: 0.5130 - val_loss: 0.8504 - val_accuracy: 0.6271
Epoch 8/80
657/657 [==============================] - 64s 97ms/step - loss: 0.9994 - ac
curacy: 0.5175 - val_loss: 0.8340 - val_accuracy: 0.6338
Epoch 9/80
657/657 [==============================] - 64s 97ms/step - loss: 0.9760 - ac
curacy: 0.5286 - val_loss: 0.8118 - val_accuracy: 0.6471
Epoch 10/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9558 - ac
curacy: 0.5373 - val_loss: 0.8015 - val_accuracy: 0.6507
Epoch 11/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9501 - ac
curacy: 0.5476 - val_loss: 0.7951 - val_accuracy: 0.6502
Epoch 12/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9418 - ac
curacy: 0.5554 - val_loss: 0.7807 - val_accuracy: 0.6551
Epoch 13/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9296 - ac
curacy: 0.5579 - val_loss: 0.7744 - val_accuracy: 0.6609
Epoch 14/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9202 - ac
curacy: 0.5650 - val_loss: 0.7699 - val_accuracy: 0.6640
Epoch 15/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9199 - ac
curacy: 0.5670 - val_loss: 0.7594 - val_accuracy: 0.6658
Epoch 16/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9125 - ac
curacy: 0.5749 - val_loss: 0.7550 - val_accuracy: 0.6640
Epoch 17/80
657/657 [==============================] - 63s 97ms/step - loss: 0.9056 - ac
curacy: 0.5763 - val_loss: 0.7472 - val_accuracy: 0.6720
Epoch 18/80
657/657 [==============================] - 63s 97ms/step - loss: 0.8999 - ac
```

```
curacy: 0.5757 - val_loss: 0.7447 - val_accuracy: 0.6662
Epoch 19/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8932 - ac
curacy: 0.5884 - val_loss: 0.7397 - val_accuracy: 0.6676
Epoch 20/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8941 - ac
curacy: 0.5834 - val_loss: 0.7340 - val_accuracy: 0.6707
Epoch 21/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8808 - ac
curacy: 0.5893 - val_loss: 0.7347 - val_accuracy: 0.6711
Epoch 22/80
657/657 [==============================] - 63s 95ms/step - loss: 0.8758 - ac
curacy: 0.5928 - val_loss: 0.7313 - val_accuracy: 0.6716
Epoch 23/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8801 - ac
curacy: 0.5956 - val_loss: 0.7275 - val_accuracy: 0.6711
Epoch 24/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8775 - ac
curacy: 0.5969 - val_loss: 0.7269 - val_accuracy: 0.6800
Epoch 25/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8747 - ac
curacy: 0.5946 - val_loss: 0.7220 - val_accuracy: 0.6778
Epoch 26/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8730 - ac
curacy: 0.5935 - val_loss: 0.7213 - val_accuracy: 0.6751
Epoch 27/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8589 - ac
curacy: 0.5981 - val_loss: 0.7192 - val_accuracy: 0.6751
Epoch 28/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8608 - ac
curacy: 0.6070 - val_loss: 0.7145 - val_accuracy: 0.6813
Epoch 29/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8500 - ac
curacy: 0.6087 - val_loss: 0.7148 - val_accuracy: 0.6809
Epoch 30/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8467 - ac
curacy: 0.6201 - val_loss: 0.7134 - val_accuracy: 0.6778
Epoch 31/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8475 - ac
curacy: 0.6180 - val_loss: 0.7050 - val_accuracy: 0.6827
Epoch 32/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8518 - ac
curacy: 0.6127 - val_loss: 0.7091 - val_accuracy: 0.6796
Epoch 33/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8484 - ac
curacy: 0.6180 - val_loss: 0.7080 - val_accuracy: 0.6804
Epoch 34/80
657/657 [==============================] - 63s 97ms/step - loss: 0.8473 - ac
curacy: 0.6109 - val_loss: 0.7084 - val_accuracy: 0.6800
Epoch 35/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8444 - ac
curacy: 0.6147 - val_loss: 0.7062 - val_accuracy: 0.6813
141/141 [==============================] - 10s 71ms/step - loss: 0.7266 - ac
curacy: 0.6818
Test accuracy: 0.6817777752876282
141/141 [==============================] - 12s 71ms/step
```

Classification Report for Configuration 1:

	precision	recall	f1-score	support
Negative	0.73	0.68	0.70	773
Neutral	0.58	0.62	0.60	741
Positive	0.74	0.74	0.74	736
accuracy			0.68	2250
macro avg	0.68	0.68	0.68	2250
weighted avg	0.68	0.68	0.68	2250

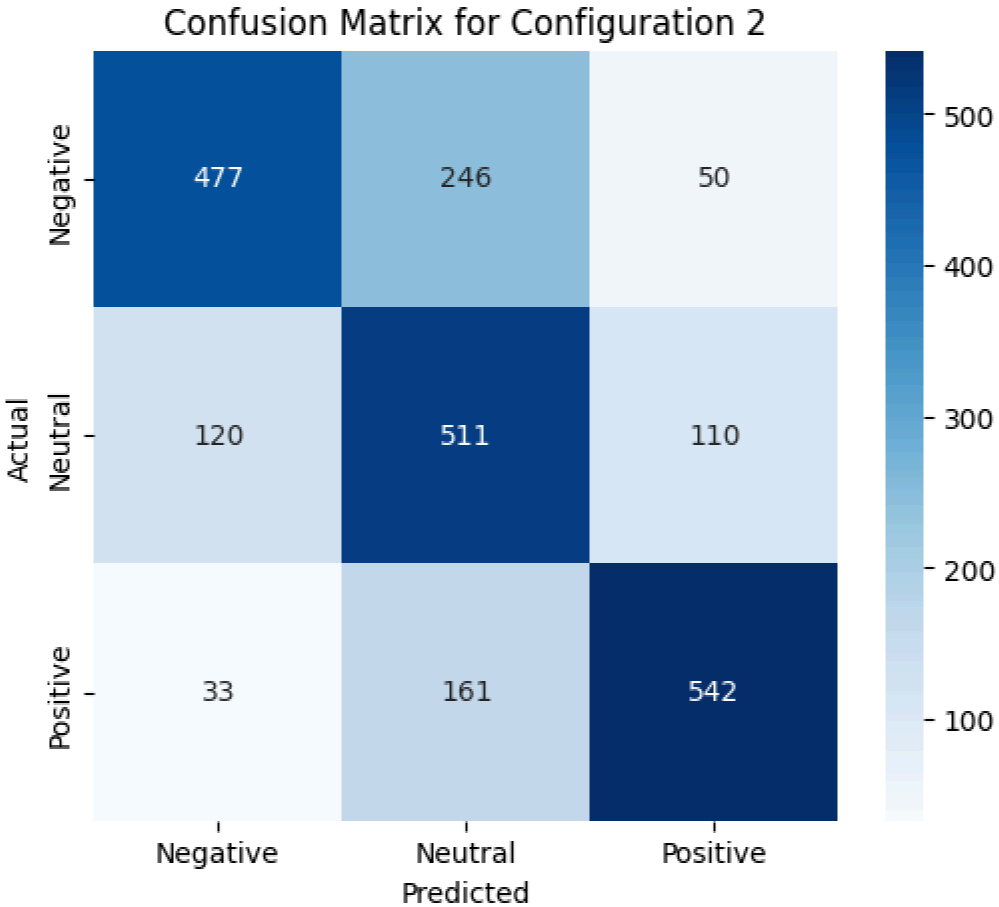


```
New best model found!
Training configuration 2/6: {'optimizer': 'adam', 'learning_rate': 1e-05, 'b
atch_size': 16, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_bat
chnorm': True}
Epoch 1/80
657/657 [==============================] - 75s 100ms/step - loss: 1.3088 - a
ccuracy: 0.3475 - val_loss: 1.0889 - val_accuracy: 0.3671
Epoch 2/80
657/657 [==============================] - 64s 97ms/step - loss: 1.2587 - ac
curacy: 0.3684 - val_loss: 1.0649 - val_accuracy: 0.4169
Epoch 3/80
657/657 [==============================] - 64s 97ms/step - loss: 1.2111 - ac
curacy: 0.3794 - val_loss: 1.0413 - val_accuracy: 0.4684
Epoch 4/80
657/657 [==============================] - 63s 96ms/step - loss: 1.1946 - ac
curacy: 0.3920 - val_loss: 1.0249 - val_accuracy: 0.4893
Epoch 5/80
657/657 [==============================] - 64s 97ms/step - loss: 1.1591 - ac
curacy: 0.4123 - val_loss: 1.0072 - val_accuracy: 0.5036
Epoch 6/80
657/657 [==============================] - 63s 96ms/step - loss: 1.1359 - ac
curacy: 0.4254 - val_loss: 0.9859 - val_accuracy: 0.5200
Epoch 7/80
657/657 [==============================] - 63s 96ms/step - loss: 1.0998 - ac
curacy: 0.4516 - val_loss: 0.9698 - val_accuracy: 0.5329
Epoch 8/80
657/657 [==============================] - 63s 96ms/step - loss: 1.0843 - ac
curacy: 0.4690 - val_loss: 0.9453 - val_accuracy: 0.5551
Epoch 9/80
657/657 [==============================] - 63s 96ms/step - loss: 1.0724 - ac
curacy: 0.4772 - val_loss: 0.9302 - val_accuracy: 0.5582
Epoch 10/80
657/657 [==============================] - 64s 97ms/step - loss: 1.0589 - ac
curacy: 0.4832 - val_loss: 0.9072 - val_accuracy: 0.5764
Epoch 11/80
657/657 [==============================] - 64s 97ms/step - loss: 1.0348 - ac
curacy: 0.4989 - val_loss: 0.8889 - val_accuracy: 0.5880
Epoch 12/80
657/657 [==============================] - 64s 97ms/step - loss: 1.0201 - ac
curacy: 0.5124 - val_loss: 0.8775 - val_accuracy: 0.5951
Epoch 13/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9953 - ac
curacy: 0.5213 - val_loss: 0.8561 - val_accuracy: 0.6076
Epoch 14/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9990 - ac
curacy: 0.5190 - val_loss: 0.8422 - val_accuracy: 0.6164
Epoch 15/80
657/657 [==============================] - 64s 97ms/step - loss: 0.9825 - ac
curacy: 0.5309 - val_loss: 0.8300 - val_accuracy: 0.6276
Epoch 16/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9740 - ac
curacy: 0.5357 - val_loss: 0.8180 - val_accuracy: 0.6373
Epoch 17/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9613 - ac
curacy: 0.5447 - val_loss: 0.8124 - val_accuracy: 0.6356
Epoch 18/80
```

```
657/657 [==============================] - 63s 96ms/step - loss: 0.9604 - ac
curacy: 0.5471 - val_loss: 0.8090 - val_accuracy: 0.6298
Epoch 19/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9607 - ac
curacy: 0.5473 - val_loss: 0.7986 - val_accuracy: 0.6413
Epoch 20/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9526 - ac
curacy: 0.5464 - val_loss: 0.7801 - val_accuracy: 0.6480
Epoch 21/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9429 - ac
curacy: 0.5599 - val_loss: 0.7774 - val_accuracy: 0.6502
Epoch 22/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9338 - ac
curacy: 0.5588 - val_loss: 0.7700 - val_accuracy: 0.6560
Epoch 23/80
657/657 [==============================] - 64s 97ms/step - loss: 0.9265 - ac
curacy: 0.5685 - val_loss: 0.7672 - val_accuracy: 0.6578
Epoch 24/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9181 - ac
curacy: 0.5692 - val_loss: 0.7654 - val_accuracy: 0.6507
Epoch 25/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9129 - ac
curacy: 0.5764 - val_loss: 0.7574 - val_accuracy: 0.6618
Epoch 26/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9180 - ac
curacy: 0.5740 - val_loss: 0.7513 - val_accuracy: 0.6693
Epoch 27/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9142 - ac
curacy: 0.5778 - val_loss: 0.7499 - val_accuracy: 0.6662
Epoch 28/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9062 - ac
curacy: 0.5729 - val_loss: 0.7480 - val_accuracy: 0.6671
Epoch 29/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9037 - ac
curacy: 0.5823 - val_loss: 0.7429 - val_accuracy: 0.6693
Epoch 30/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9043 - ac
curacy: 0.5762 - val_loss: 0.7327 - val_accuracy: 0.6760
Epoch 31/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9029 - ac
curacy: 0.5825 - val_loss: 0.7330 - val_accuracy: 0.6751
Epoch 32/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8911 - ac
curacy: 0.5911 - val_loss: 0.7333 - val_accuracy: 0.6742
Epoch 33/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8874 - ac
curacy: 0.5881 - val_loss: 0.7310 - val_accuracy: 0.6782
Epoch 34/80
657/657 [==============================] - 63s 97ms/step - loss: 0.8786 - ac
curacy: 0.5951 - val_loss: 0.7255 - val_accuracy: 0.6751
Epoch 35/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8831 - ac
curacy: 0.5969 - val_loss: 0.7253 - val_accuracy: 0.6804
Epoch 36/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8802 - ac
curacy: 0.5883 - val_loss: 0.7252 - val_accuracy: 0.6818
```

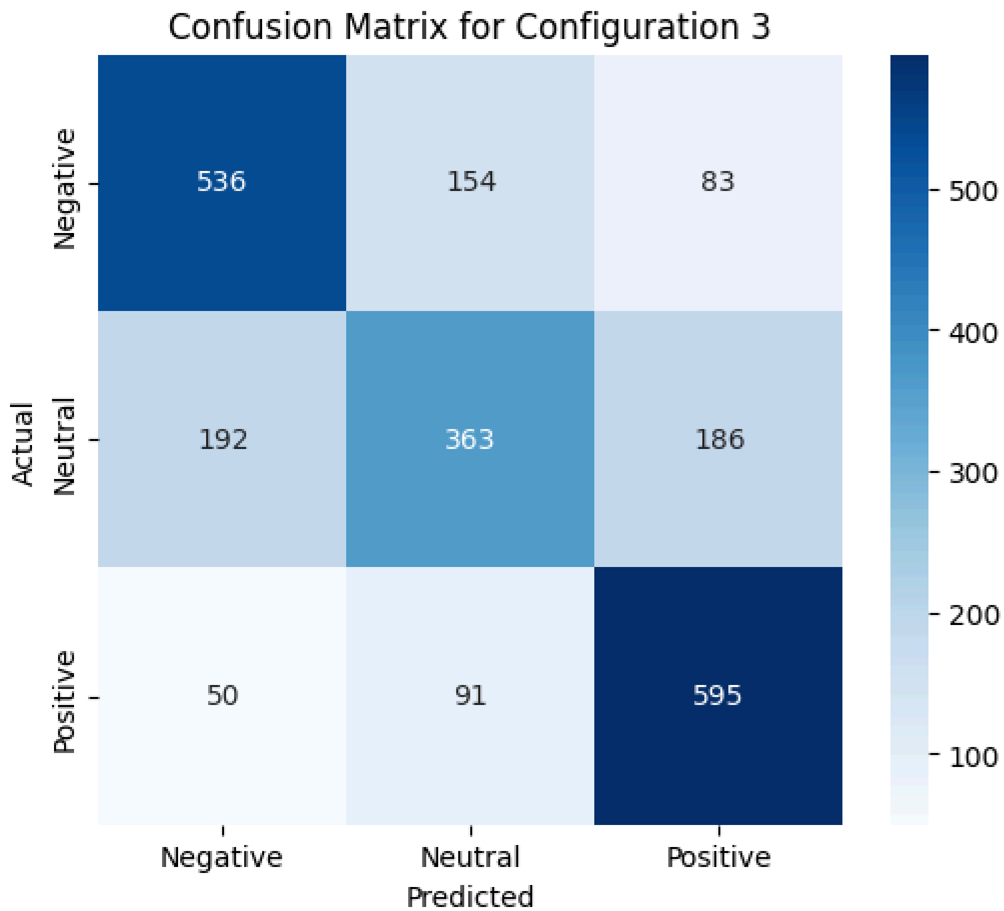
```
Epoch 37/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8825 - ac
curacy: 0.5938 - val_loss: 0.7283 - val_accuracy: 0.6787
Epoch 38/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8624 - ac
curacy: 0.5997 - val_loss: 0.7217 - val_accuracy: 0.6840
Epoch 39/80
657/657 [==============================] - 63s 97ms/step - loss: 0.8624 - ac
curacy: 0.6029 - val_loss: 0.7148 - val_accuracy: 0.6853
Epoch 40/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8697 - ac
curacy: 0.6002 - val_loss: 0.7165 - val_accuracy: 0.6836
Epoch 41/80
657/657 [==============================] - 63s 97ms/step - loss: 0.8584 - ac
curacy: 0.6061 - val_loss: 0.7144 - val_accuracy: 0.6840
Epoch 42/80
657/657 [==============================] - 63s 97ms/step - loss: 0.8528 - ac
curacy: 0.6068 - val_loss: 0.7106 - val_accuracy: 0.6884
Epoch 43/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8592 - ac
curacy: 0.6056 - val_loss: 0.7118 - val_accuracy: 0.6853
Epoch 44/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8487 - ac
curacy: 0.6084 - val_loss: 0.7070 - val_accuracy: 0.6844
Epoch 45/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8555 - ac
curacy: 0.6113 - val_loss: 0.7095 - val_accuracy: 0.6849
Epoch 46/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8581 - ac
curacy: 0.6120 - val_loss: 0.7033 - val_accuracy: 0.6871
Epoch 47/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8490 - ac
curacy: 0.6094 - val_loss: 0.7079 - val_accuracy: 0.6831
Epoch 48/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8473 - ac
curacy: 0.6132 - val_loss: 0.7084 - val_accuracy: 0.6871
Epoch 49/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8464 - ac
curacy: 0.6117 - val_loss: 0.7050 - val_accuracy: 0.6822
Epoch 50/80
657/657 [==============================] - 63s 97ms/step - loss: 0.8429 - ac
curacy: 0.6107 - val_loss: 0.7050 - val_accuracy: 0.6818
141/141 [==============================] - 10s 74ms/step - loss: 0.7324 - ac
curacy: 0.6800
Test accuracy: 0.6800000071525574
141/141 [==============================] - 12s 72ms/step
Classification Report for Configuration 2:
              precision    recall  f1-score   support

    Negative       0.76      0.62      0.68       773
     Neutral       0.56      0.69      0.62       741
    Positive       0.77      0.74      0.75       736

    accuracy                           0.68      2250
   macro avg       0.70      0.68      0.68      2250
```

```
weighted avg       0.70      0.68      0.68      2250
```



```
Training configuration 3/6: {'optimizer': 'adamw', 'learning_rate': 1e-05,
'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_b
atchnorm': True}
Epoch 1/80
329/329 [==============================] - 71s 186ms/step - loss: 1.1646 - a
ccuracy: 0.4133 - val_loss: 0.9168 - val_accuracy: 0.5569
Epoch 2/80
329/329 [==============================] - 59s 181ms/step - loss: 1.0241 - a
ccuracy: 0.4986 - val_loss: 0.8256 - val_accuracy: 0.6178
Epoch 3/80
329/329 [==============================] - 59s 179ms/step - loss: 0.9853 - a
ccuracy: 0.5235 - val_loss: 0.7950 - val_accuracy: 0.6342
Epoch 4/80
329/329 [==============================] - 58s 177ms/step - loss: 0.9407 - a
ccuracy: 0.5493 - val_loss: 0.7703 - val_accuracy: 0.6564
Epoch 5/80
329/329 [==============================] - 58s 177ms/step - loss: 0.9257 - a
ccuracy: 0.5671 - val_loss: 0.7540 - val_accuracy: 0.6618
Epoch 6/80
329/329 [==============================] - 58s 176ms/step - loss: 0.9061 - a
ccuracy: 0.5725 - val_loss: 0.7654 - val_accuracy: 0.6609
Epoch 7/80
329/329 [==============================] - 59s 180ms/step - loss: 0.8978 - a
ccuracy: 0.5814 - val_loss: 0.7447 - val_accuracy: 0.6640
Epoch 8/80
329/329 [==============================] - 60s 182ms/step - loss: 0.8962 - a
ccuracy: 0.5843 - val_loss: 0.7394 - val_accuracy: 0.6693
Epoch 9/80
329/329 [==============================] - 59s 178ms/step - loss: 0.8791 - a
ccuracy: 0.5898 - val_loss: 0.7354 - val_accuracy: 0.6760
Epoch 10/80
329/329 [==============================] - 58s 177ms/step - loss: 0.8792 - a
ccuracy: 0.5903 - val_loss: 0.7484 - val_accuracy: 0.6684
Epoch 11/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8602 - a
ccuracy: 0.6021 - val_loss: 0.7249 - val_accuracy: 0.6818
Epoch 12/80
329/329 [==============================] - 59s 178ms/step - loss: 0.8667 - a
ccuracy: 0.6015 - val_loss: 0.7213 - val_accuracy: 0.6813
Epoch 13/80
329/329 [==============================] - 58s 177ms/step - loss: 0.8571 - a
ccuracy: 0.6025 - val_loss: 0.7157 - val_accuracy: 0.6867
Epoch 14/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8540 - a
ccuracy: 0.6099 - val_loss: 0.7184 - val_accuracy: 0.6791
Epoch 15/80
329/329 [==============================] - 60s 181ms/step - loss: 0.8424 - a
ccuracy: 0.6165 - val_loss: 0.7170 - val_accuracy: 0.6836
Epoch 16/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8435 - a
ccuracy: 0.6099 - val_loss: 0.7229 - val_accuracy: 0.6796
Epoch 17/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8460 - a
ccuracy: 0.6117 - val_loss: 0.7123 - val_accuracy: 0.6844
Epoch 18/80
329/329 [==============================] - 58s 178ms/step - loss: 0.8430 - a
```

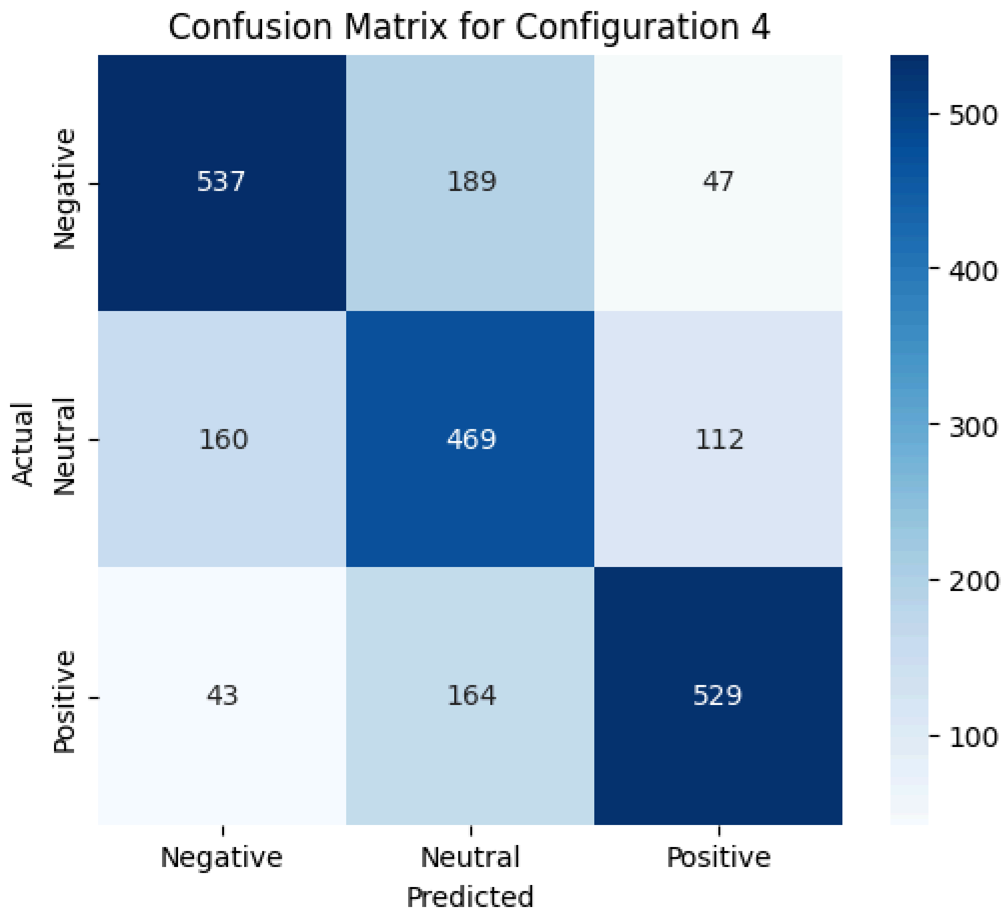
```
ccuracy: 0.6159 - val_loss: 0.7031 - val_accuracy: 0.6947
Epoch 19/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8379 - a
ccuracy: 0.6222 - val_loss: 0.7014 - val_accuracy: 0.6933
Epoch 20/80
329/329 [==============================] - 58s 177ms/step - loss: 0.8321 - a
ccuracy: 0.6176 - val_loss: 0.7160 - val_accuracy: 0.6827
Epoch 21/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8240 - a
ccuracy: 0.6237 - val_loss: 0.7063 - val_accuracy: 0.6907
Epoch 22/80
329/329 [==============================] - 58s 177ms/step - loss: 0.8282 - a
ccuracy: 0.6229 - val_loss: 0.6997 - val_accuracy: 0.6902
Epoch 23/80
329/329 [==============================] - 58s 177ms/step - loss: 0.8264 - a
ccuracy: 0.6252 - val_loss: 0.6994 - val_accuracy: 0.6929
Epoch 24/80
329/329 [==============================] - 58s 177ms/step - loss: 0.8312 - a
ccuracy: 0.6238 - val_loss: 0.6977 - val_accuracy: 0.6956
Epoch 25/80
329/329 [==============================] - 59s 180ms/step - loss: 0.8231 - a
ccuracy: 0.6315 - val_loss: 0.7000 - val_accuracy: 0.6911
Epoch 26/80
329/329 [==============================] - 58s 177ms/step - loss: 0.8238 - a
ccuracy: 0.6273 - val_loss: 0.7112 - val_accuracy: 0.6831
Epoch 27/80
329/329 [==============================] - 59s 180ms/step - loss: 0.8189 - a
ccuracy: 0.6330 - val_loss: 0.7077 - val_accuracy: 0.6849
Epoch 28/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8166 - a
ccuracy: 0.6282 - val_loss: 0.6983 - val_accuracy: 0.6920
71/71 [==============================] - 10s 135ms/step - loss: 0.7444 - acc
uracy: 0.6640
Test accuracy: 0.6639999747276306
71/71 [==============================] - 12s 131ms/step
Classification Report for Configuration 3:
              precision    recall  f1-score   support

    Negative       0.69      0.69      0.69       773
     Neutral       0.60      0.49      0.54       741
    Positive       0.69      0.81      0.74       736

    accuracy                           0.66      2250
   macro avg       0.66      0.66      0.66      2250
weighted avg       0.66      0.66      0.66      2250
```



Confusion Matrix for Configuration 3
Negative
Neutral
Positive
Actual
Predicted
536
154
83
192
363
186
50
91
595
500
400
300
200
100

```
Training configuration 4/6: {'optimizer': 'adam', 'learning_rate': 3e-05, 'b
atch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'relu', 'use_bat
chnorm': True}
Epoch 1/80
329/329 [==============================] - 69s 186ms/step - loss: 1.2325 - a
ccuracy: 0.3845 - val_loss: 1.0759 - val_accuracy: 0.4009
Epoch 2/80
329/329 [==============================] - 58s 176ms/step - loss: 1.0995 - a
ccuracy: 0.4637 - val_loss: 0.9677 - val_accuracy: 0.5240
Epoch 3/80
329/329 [==============================] - 58s 176ms/step - loss: 1.0272 - a
ccuracy: 0.5068 - val_loss: 0.9125 - val_accuracy: 0.5609
Epoch 4/80
329/329 [==============================] - 58s 177ms/step - loss: 0.9839 - a
ccuracy: 0.5324 - val_loss: 0.8582 - val_accuracy: 0.5884
Epoch 5/80
329/329 [==============================] - 59s 180ms/step - loss: 0.9542 - a
ccuracy: 0.5526 - val_loss: 0.8174 - val_accuracy: 0.6218
Epoch 6/80
329/329 [==============================] - 59s 178ms/step - loss: 0.9307 - a
ccuracy: 0.5630 - val_loss: 0.7895 - val_accuracy: 0.6364
Epoch 7/80
329/329 [==============================] - 58s 176ms/step - loss: 0.9017 - a
ccuracy: 0.5827 - val_loss: 0.7576 - val_accuracy: 0.6511
Epoch 8/80
329/329 [==============================] - 59s 178ms/step - loss: 0.8905 - a
ccuracy: 0.5927 - val_loss: 0.7424 - val_accuracy: 0.6622
Epoch 9/80
329/329 [==============================] - 58s 177ms/step - loss: 0.8758 - a
ccuracy: 0.5964 - val_loss: 0.7455 - val_accuracy: 0.6609
Epoch 10/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8596 - a
ccuracy: 0.6075 - val_loss: 0.7302 - val_accuracy: 0.6676
Epoch 11/80
329/329 [==============================] - 59s 178ms/step - loss: 0.8585 - a
ccuracy: 0.6048 - val_loss: 0.7128 - val_accuracy: 0.6782
Epoch 12/80
329/329 [==============================] - 60s 181ms/step - loss: 0.8542 - a
ccuracy: 0.6094 - val_loss: 0.7047 - val_accuracy: 0.6889
Epoch 13/80
329/329 [==============================] - 59s 181ms/step - loss: 0.8429 - a
ccuracy: 0.6165 - val_loss: 0.7019 - val_accuracy: 0.6893
Epoch 14/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8335 - a
ccuracy: 0.6172 - val_loss: 0.6947 - val_accuracy: 0.6831
Epoch 15/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8286 - a
ccuracy: 0.6240 - val_loss: 0.6930 - val_accuracy: 0.6831
Epoch 16/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8269 - a
ccuracy: 0.6218 - val_loss: 0.6910 - val_accuracy: 0.6884
Epoch 17/80
329/329 [==============================] - 58s 177ms/step - loss: 0.8261 - a
ccuracy: 0.6211 - val_loss: 0.6891 - val_accuracy: 0.6804
Epoch 18/80
329/329 [==============================] - 59s 178ms/step - loss: 0.8168 - a
```

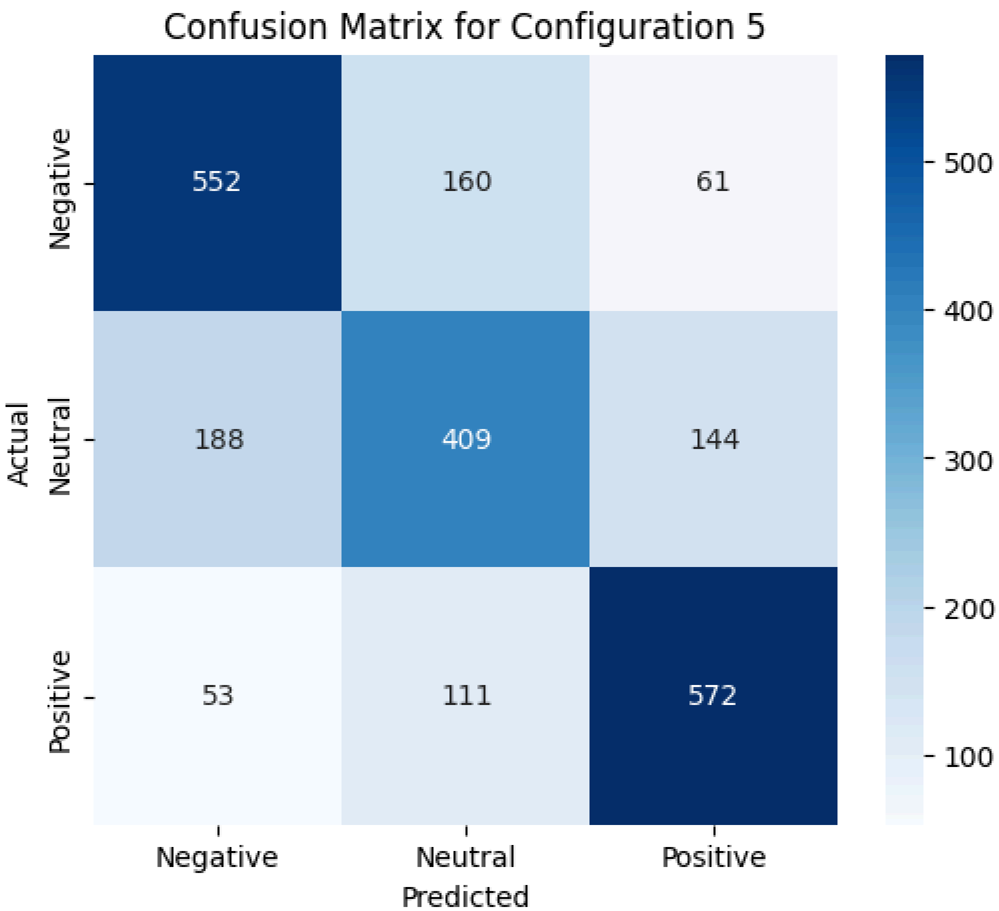
```
ccuracy: 0.6271 - val_loss: 0.6865 - val_accuracy: 0.6849
Epoch 19/80
329/329 [==============================] - 58s 177ms/step - loss: 0.8082 - a
ccuracy: 0.6328 - val_loss: 0.6838 - val_accuracy: 0.6924
Epoch 20/80
329/329 [==============================] - 58s 176ms/step - loss: 0.8109 - a
ccuracy: 0.6311 - val_loss: 0.6822 - val_accuracy: 0.6893
Epoch 21/80
329/329 [==============================] - 58s 176ms/step - loss: 0.7962 - a
ccuracy: 0.6431 - val_loss: 0.6836 - val_accuracy: 0.6947
Epoch 22/80
329/329 [==============================] - 59s 180ms/step - loss: 0.8017 - a
ccuracy: 0.6379 - val_loss: 0.6822 - val_accuracy: 0.6924
Epoch 23/80
329/329 [==============================] - 60s 181ms/step - loss: 0.7976 - a
ccuracy: 0.6390 - val_loss: 0.6798 - val_accuracy: 0.6938
Epoch 24/80
329/329 [==============================] - 58s 176ms/step - loss: 0.7858 - a
ccuracy: 0.6442 - val_loss: 0.6751 - val_accuracy: 0.6947
Epoch 25/80
329/329 [==============================] - 58s 176ms/step - loss: 0.7825 - a
ccuracy: 0.6501 - val_loss: 0.6792 - val_accuracy: 0.6902
Epoch 26/80
329/329 [==============================] - 58s 177ms/step - loss: 0.7823 - a
ccuracy: 0.6487 - val_loss: 0.6706 - val_accuracy: 0.6978
Epoch 27/80
329/329 [==============================] - 59s 180ms/step - loss: 0.7838 - a
ccuracy: 0.6482 - val_loss: 0.6847 - val_accuracy: 0.6876
Epoch 28/80
329/329 [==============================] - 59s 180ms/step - loss: 0.7813 - a
ccuracy: 0.6476 - val_loss: 0.6747 - val_accuracy: 0.6938
Epoch 29/80
329/329 [==============================] - 58s 175ms/step - loss: 0.7813 - a
ccuracy: 0.6487 - val_loss: 0.6785 - val_accuracy: 0.6964
Epoch 30/80
329/329 [==============================] - 58s 176ms/step - loss: 0.7787 - a
ccuracy: 0.6518 - val_loss: 0.6717 - val_accuracy: 0.6956
71/71 [==============================] - 10s 135ms/step - loss: 0.7081 - acc
uracy: 0.6822
Test accuracy: 0.6822222471237183
71/71 [==============================] - 24s 131ms/step
Classification Report for Configuration 4:
              precision    recall  f1-score   support

    Negative       0.73      0.69      0.71       773
     Neutral       0.57      0.63      0.60       741
    Positive       0.77      0.72      0.74       736

    accuracy                           0.68      2250
   macro avg       0.69      0.68      0.68      2250
weighted avg       0.69      0.68      0.68      2250
```



Confusion Matrix for Configuration 4
Actual
Negative
Neutral
Positive
537
189
47
160
469
112
43
164
529
Negative
Neutral
Positive
Predicted
500
400
300
200
100

```
New best model found!
Training configuration 5/6: {'optimizer': 'adamw', 'learning_rate': 1e-05,
'batch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_b
atchnorm': True}
Epoch 1/80
657/657 [==============================] - 74s 100ms/step - loss: 1.1607 - a
ccuracy: 0.4116 - val_loss: 0.8530 - val_accuracy: 0.6124
Epoch 2/80
657/657 [==============================] - 64s 98ms/step - loss: 1.0218 - ac
curacy: 0.5026 - val_loss: 0.7762 - val_accuracy: 0.6493
Epoch 3/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9790 - ac
curacy: 0.5250 - val_loss: 0.7458 - val_accuracy: 0.6569
Epoch 4/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9364 - ac
curacy: 0.5524 - val_loss: 0.7274 - val_accuracy: 0.6693
Epoch 5/80
657/657 [==============================] - 63s 97ms/step - loss: 0.9222 - ac
curacy: 0.5597 - val_loss: 0.7226 - val_accuracy: 0.6769
Epoch 6/80
657/657 [==============================] - 64s 97ms/step - loss: 0.9090 - ac
curacy: 0.5733 - val_loss: 0.7170 - val_accuracy: 0.6733
Epoch 7/80
657/657 [==============================] - 65s 98ms/step - loss: 0.8985 - ac
curacy: 0.5810 - val_loss: 0.7155 - val_accuracy: 0.6716
Epoch 8/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8905 - ac
curacy: 0.5907 - val_loss: 0.7105 - val_accuracy: 0.6778
Epoch 9/80
657/657 [==============================] - 65s 98ms/step - loss: 0.8869 - ac
curacy: 0.5890 - val_loss: 0.7080 - val_accuracy: 0.6849
Epoch 10/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8777 - ac
curacy: 0.5950 - val_loss: 0.6973 - val_accuracy: 0.6898
Epoch 11/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8786 - ac
curacy: 0.5947 - val_loss: 0.6973 - val_accuracy: 0.6947
Epoch 12/80
657/657 [==============================] - 65s 98ms/step - loss: 0.8649 - ac
curacy: 0.6015 - val_loss: 0.6896 - val_accuracy: 0.6902
Epoch 13/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8633 - ac
curacy: 0.6028 - val_loss: 0.6916 - val_accuracy: 0.6907
Epoch 14/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8517 - ac
curacy: 0.6082 - val_loss: 0.6890 - val_accuracy: 0.6920
Epoch 15/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8550 - ac
curacy: 0.6141 - val_loss: 0.6916 - val_accuracy: 0.6902
Epoch 16/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8495 - ac
curacy: 0.6125 - val_loss: 0.6894 - val_accuracy: 0.6902
Epoch 17/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8397 - ac
curacy: 0.6161 - val_loss: 0.6891 - val_accuracy: 0.6911
Epoch 18/80
```

```
657/657 [==============================] - 65s 98ms/step - loss: 0.8504 - ac
curacy: 0.6118 - val_loss: 0.6872 - val_accuracy: 0.6916
Epoch 19/80
657/657 [==============================] - 65s 98ms/step - loss: 0.8328 - ac
curacy: 0.6201 - val_loss: 0.6838 - val_accuracy: 0.6938
Epoch 20/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8340 - ac
curacy: 0.6230 - val_loss: 0.6844 - val_accuracy: 0.6978
Epoch 21/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8379 - ac
curacy: 0.6188 - val_loss: 0.6932 - val_accuracy: 0.6898
Epoch 22/80
657/657 [==============================] - 65s 98ms/step - loss: 0.8309 - ac
curacy: 0.6245 - val_loss: 0.6777 - val_accuracy: 0.6982
Epoch 23/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8247 - ac
curacy: 0.6294 - val_loss: 0.6829 - val_accuracy: 0.6947
Epoch 24/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8332 - ac
curacy: 0.6194 - val_loss: 0.6795 - val_accuracy: 0.6973
Epoch 25/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8281 - ac
curacy: 0.6230 - val_loss: 0.6801 - val_accuracy: 0.6978
Epoch 26/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8155 - ac
curacy: 0.6337 - val_loss: 0.6783 - val_accuracy: 0.6996
141/141 [==============================] - 10s 74ms/step - loss: 0.7241 - ac
curacy: 0.6813
Test accuracy: 0.6813333630561829
141/141 [==============================] - 12s 71ms/step
Classification Report for Configuration 5:
              precision    recall  f1-score   support

    Negative       0.70      0.71      0.70       773
     Neutral       0.60      0.55      0.58       741
    Positive       0.74      0.78      0.76       736

    accuracy                           0.68      2250
   macro avg       0.68      0.68      0.68      2250
weighted avg       0.68      0.68      0.68      2250
```

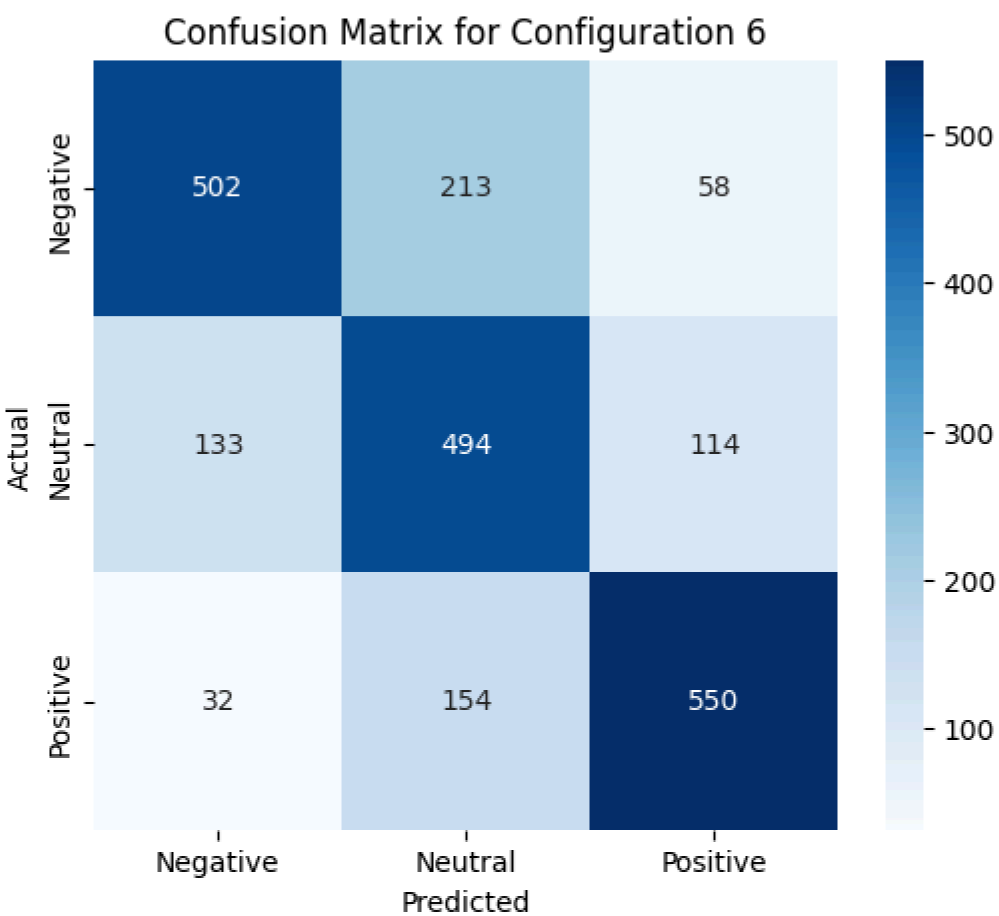


Confusion Matrix for Configuration 5
Negative
Neutral
Positive
Actual
Predicted
552
160
61
188
409
144
53
111
572
100
200
300
400
500

```
Training configuration 6/6: {'optimizer': 'adam', 'learning_rate': 3e-05, 'b
atch_size': 16, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_bat
chnorm': True}
Epoch 1/80
657/657 [==============================] - 74s 100ms/step - loss: 1.3482 - a
ccuracy: 0.3550 - val_loss: 1.0332 - val_accuracy: 0.4782
Epoch 2/80
657/657 [==============================] - 64s 98ms/step - loss: 1.1878 - ac
curacy: 0.4230 - val_loss: 0.9304 - val_accuracy: 0.5756
Epoch 3/80
657/657 [==============================] - 64s 98ms/step - loss: 1.1114 - ac
curacy: 0.4662 - val_loss: 0.8691 - val_accuracy: 0.6076
Epoch 4/80
657/657 [==============================] - 63s 97ms/step - loss: 1.0518 - ac
curacy: 0.5013 - val_loss: 0.8196 - val_accuracy: 0.6378
Epoch 5/80
657/657 [==============================] - 64s 97ms/step - loss: 1.0183 - ac
curacy: 0.5194 - val_loss: 0.7807 - val_accuracy: 0.6600
Epoch 6/80
657/657 [==============================] - 64s 97ms/step - loss: 0.9804 - ac
curacy: 0.5428 - val_loss: 0.7592 - val_accuracy: 0.6702
Epoch 7/80
657/657 [==============================] - 63s 97ms/step - loss: 0.9561 - ac
curacy: 0.5544 - val_loss: 0.7450 - val_accuracy: 0.6698
Epoch 8/80
657/657 [==============================] - 65s 98ms/step - loss: 0.9548 - ac
curacy: 0.5575 - val_loss: 0.7329 - val_accuracy: 0.6653
Epoch 9/80
657/657 [==============================] - 64s 98ms/step - loss: 0.9325 - ac
curacy: 0.5627 - val_loss: 0.7243 - val_accuracy: 0.6773
Epoch 10/80
657/657 [==============================] - 63s 97ms/step - loss: 0.9246 - ac
curacy: 0.5732 - val_loss: 0.7209 - val_accuracy: 0.6764
Epoch 11/80
657/657 [==============================] - 64s 97ms/step - loss: 0.9088 - ac
curacy: 0.5789 - val_loss: 0.7185 - val_accuracy: 0.6773
Epoch 12/80
657/657 [==============================] - 64s 98ms/step - loss: 0.9068 - ac
curacy: 0.5835 - val_loss: 0.7123 - val_accuracy: 0.6764
Epoch 13/80
657/657 [==============================] - 63s 96ms/step - loss: 0.9009 - ac
curacy: 0.5838 - val_loss: 0.7122 - val_accuracy: 0.6796
Epoch 14/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8809 - ac
curacy: 0.5937 - val_loss: 0.7048 - val_accuracy: 0.6831
Epoch 15/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8779 - ac
curacy: 0.5977 - val_loss: 0.7074 - val_accuracy: 0.6867
Epoch 16/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8689 - ac
curacy: 0.6047 - val_loss: 0.6997 - val_accuracy: 0.6867
Epoch 17/80
657/657 [==============================] - 63s 97ms/step - loss: 0.8709 - ac
curacy: 0.6018 - val_loss: 0.6946 - val_accuracy: 0.6871
Epoch 18/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8561 - ac
```

```
curacy: 0.6086 - val_loss: 0.6944 - val_accuracy: 0.6849
Epoch 19/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8622 - ac
curacy: 0.6124 - val_loss: 0.6930 - val_accuracy: 0.6876
Epoch 20/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8491 - ac
curacy: 0.6149 - val_loss: 0.6916 - val_accuracy: 0.6867
Epoch 21/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8495 - ac
curacy: 0.6109 - val_loss: 0.6912 - val_accuracy: 0.6920
Epoch 22/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8357 - ac
curacy: 0.6280 - val_loss: 0.6951 - val_accuracy: 0.6858
Epoch 23/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8432 - ac
curacy: 0.6178 - val_loss: 0.6902 - val_accuracy: 0.6964
Epoch 24/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8314 - ac
curacy: 0.6224 - val_loss: 0.6880 - val_accuracy: 0.6933
Epoch 25/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8289 - ac
curacy: 0.6270 - val_loss: 0.6832 - val_accuracy: 0.7009
Epoch 26/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8235 - ac
curacy: 0.6305 - val_loss: 0.6822 - val_accuracy: 0.6911
Epoch 27/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8185 - ac
curacy: 0.6237 - val_loss: 0.6784 - val_accuracy: 0.6978
Epoch 28/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8110 - ac
curacy: 0.6400 - val_loss: 0.6772 - val_accuracy: 0.6973
Epoch 29/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8164 - ac
curacy: 0.6313 - val_loss: 0.6852 - val_accuracy: 0.6996
Epoch 30/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8089 - ac
curacy: 0.6349 - val_loss: 0.6762 - val_accuracy: 0.6982
Epoch 31/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8066 - ac
curacy: 0.6309 - val_loss: 0.6787 - val_accuracy: 0.6951
Epoch 32/80
657/657 [==============================] - 63s 97ms/step - loss: 0.8114 - ac
curacy: 0.6282 - val_loss: 0.6730 - val_accuracy: 0.6987
Epoch 33/80
657/657 [==============================] - 63s 96ms/step - loss: 0.8000 - ac
curacy: 0.6426 - val_loss: 0.6755 - val_accuracy: 0.7022
Epoch 34/80
657/657 [==============================] - 63s 97ms/step - loss: 0.7926 - ac
curacy: 0.6450 - val_loss: 0.6740 - val_accuracy: 0.6964
Epoch 35/80
657/657 [==============================] - 64s 97ms/step - loss: 0.7958 - ac
curacy: 0.6419 - val_loss: 0.6704 - val_accuracy: 0.6987
Epoch 36/80
657/657 [==============================] - 63s 96ms/step - loss: 0.7974 - ac
curacy: 0.6413 - val_loss: 0.6775 - val_accuracy: 0.6996
Epoch 37/80
```

```
657/657 [==============================] - 64s 97ms/step - loss: 0.7891 - ac
curacy: 0.6467 - val_loss: 0.6720 - val_accuracy: 0.7044
Epoch 38/80
657/657 [==============================] - 64s 97ms/step - loss: 0.8008 - ac
curacy: 0.6415 - val_loss: 0.6745 - val_accuracy: 0.7031
Epoch 39/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7926 - ac
curacy: 0.6440 - val_loss: 0.6706 - val_accuracy: 0.7013
141/141 [==============================] - 10s 73ms/step - loss: 0.7023 - ac
curacy: 0.6871
Test accuracy: 0.6871111392974854
141/141 [==============================] - 12s 72ms/step
Classification Report for Configuration 6:
              precision    recall  f1-score   support

    Negative       0.75      0.65      0.70       773
     Neutral       0.57      0.67      0.62       741
    Positive       0.76      0.75      0.75       736

    accuracy                           0.69      2250
   macro avg       0.70      0.69      0.69      2250
weighted avg       0.70      0.69      0.69      2250
```




```
New best model found!
```

6.1.2 Batch-Normalized Models: Examples of Mis-classified Points

In [ ]:

```
# After the training loop
print(f"\nBest Model Configuration: {best_config}")
print(f"Best Model Test Accuracy: {best_accuracy}")

# Use the best model's predictions
y_pred = best_y_pred
y_true = best_y_true

# Identify misclassified examples
misclassified_indices = np.where(y_pred != y_true)[0]

# Decode test texts
def decode_texts(input_ids):
    return [tokenizer.decode(ids, skip_special_tokens=True) for ids in input

test_texts = decode_texts(test_input_ids)

# Extract false positives and false negatives
false_positives = []
false_negatives = []

# Mapping of label indices to label names
label_map = {0: 'Negative', 1: 'Neutral', 2: 'Positive'}

# Loop through misclassified examples to separate false positives and false
for idx in misclassified_indices:
    true_label = y_true[idx]
    predicted_label = y_pred[idx]
    text = test_texts[idx]

    if predicted_label == 2 and true_label != 2:
        # Model predicted Positive, but true label is Negative or Neutral
        false_positives.append((text, label_map[true_label], label_map[predi
    elif predicted_label != 2 and true_label == 2:
        # Model predicted Negative or Neutral, but true label is Positive
        false_negatives.append((text, label_map[true_label], label_map[predi

# Display a few examples of false positives
print("\nExamples of False Positives:")
for i in range(min(3, len(false_positives))):
    text, true_label, predicted_label = false_positives[i]
    print(f"\nText: {text}")
    print(f"True Label: {true_label}")
    print(f"Predicted Label: {predicted_label}")

# Display a few examples of false negatives
print("\nExamples of False Negatives:")
for i in range(min(3, len(false_negatives))):
    text, true_label, predicted_label = false_negatives[i]
    print(f"\nText: {text}")
    print(f"True Label: {true_label}")
    print(f"Predicted Label: {predicted_label}")
```

```
Best Model Configuration: {'optimizer': 'adam', 'learning_rate': 3e-05, 'bat
ch_size': 16, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_batch
norm': True}
Best Model Test Accuracy: 0.6871111392974854

Examples of False Positives:

Text: this is quite good ive taken to starting my mornings with a nice hot m
ug
True Label: Neutral
Predicted Label: Positive

Text: i love gloria jeans cinnamon nut strudel coffee and decided to try thi
s the flavoring is a little over powering for me if youve tried this flavor
before and like it you wont be disappointed with it in the kcups packaging
True Label: Neutral
Predicted Label: Positive

Text: we love clif kid z bars i decided to try these based on all of the gre
at reviews we bought the strawberry and no one in the house likes them my ki
ds are picky so i always try the treats just to see if they are even worth t
rying i eat healthy and dont mind most healthy snacks but even i did not car
e for these i am going to try another flavor but the strawberry just didnt c
ut it here
True Label: Negative
Predicted Label: Positive

Examples of False Negatives:

Text: i really only needed the funnelpitcher but it was easy and neat to mak
e funnel cakes i had been using a kitchen funnel which made a big mess no ma
tter how careful i was i didnt use the ring because i made the funnel cakes
in a small deep fryer the mix was fine but i prefer my own recipe if i could
have just bought the funnelpitcher that would have been fine and saved me so
me money but im not disappointed
True Label: Positive
Predicted Label: Neutral

Text: i was very impressed with the flavor growing up on mrs butterworths i
developed a taste for commercial syrup and had a hard time adjusting to the
real thing but when i opened up the jug of coombs it was delish i would high
ly recommend this product
True Label: Positive
Predicted Label: Neutral

Text: i got a larger one than my puppy probably required and she loves dragg
ing it around playing tug of war and just chewing on it unfortunately it doe
s shed i dont know if there are rope toys out there that are any sturdier bu
t this one does leave little threads behind otherwise great toy
True Label: Positive
Predicted Label: Neutral
```

In [ ]:

```
# Save the best model to a .keras file
best_model.save('best_model.keras')
print("\nBest model saved to 'best_model.keras'.")
```

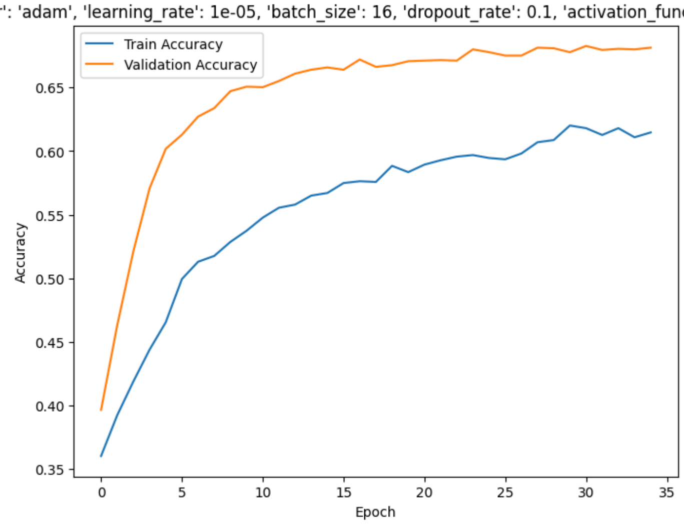
```
/usr/local/lib/python3.10/dist-packages/transformers/generation/tf_utils.py:
465: UserWarning: `seed_generator` is deprecated and will be removed in a fu
ture version.
  warnings.warn("`seed_generator` is deprecated and will be removed in a fut
ure version.", UserWarning)
Best model saved to 'best_model.keras'.
```

6.1.3 Batch-Normalized Models: Visualizations

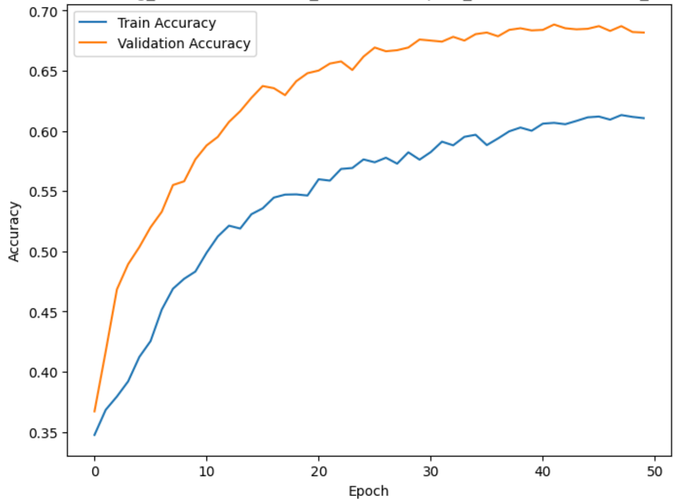
We will show the process of training and plotting the training (not exactly sure what she means by this since the next section is to plot the validation set and analyze the behaviour of training and validation, but we'll figure it out).

In [ ]:
```python
# Visualize the results

# Plot training and validation accuracy for each configuration
for i, (history, config) in enumerate(histories):
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title(f"Configuration {i+1}: {config}")
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
```

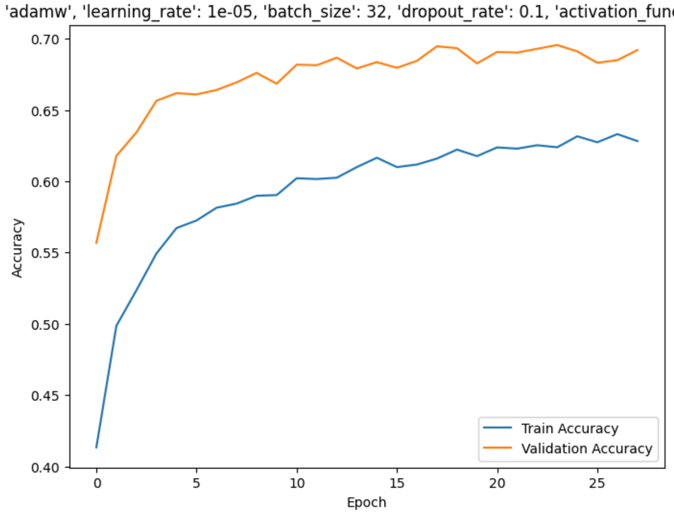
Configuration 1: {'optimizer': 'adam', 'learning_rate': 1e-05, 'batch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'relu', 'use_batchnorm': True}



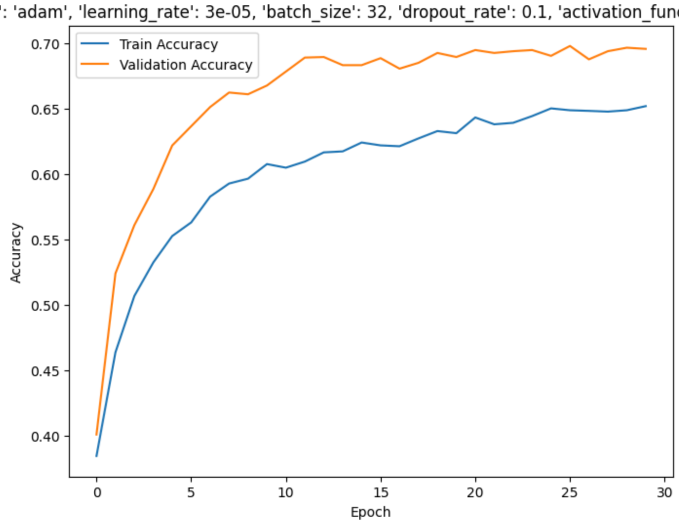
Configuration 2: {'optimizer': 'adam', 'learning_rate': 1e-05, 'batch_size': 16, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_batchnorm': True}



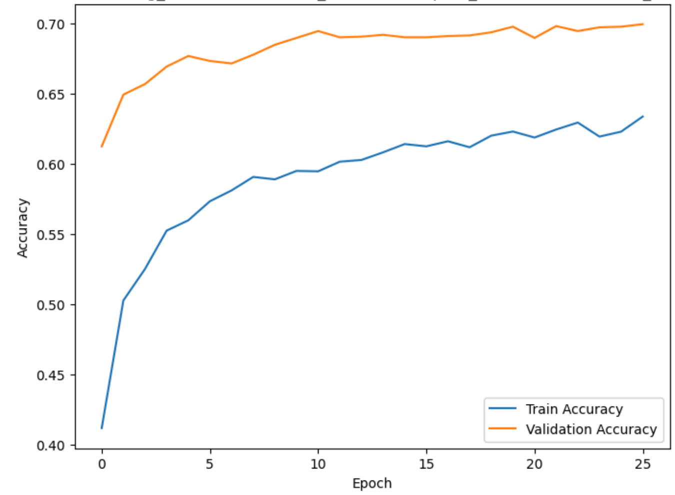
Configuration 3: {'optimizer': 'adamw', 'learning_rate': 1e-05, 'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': True}



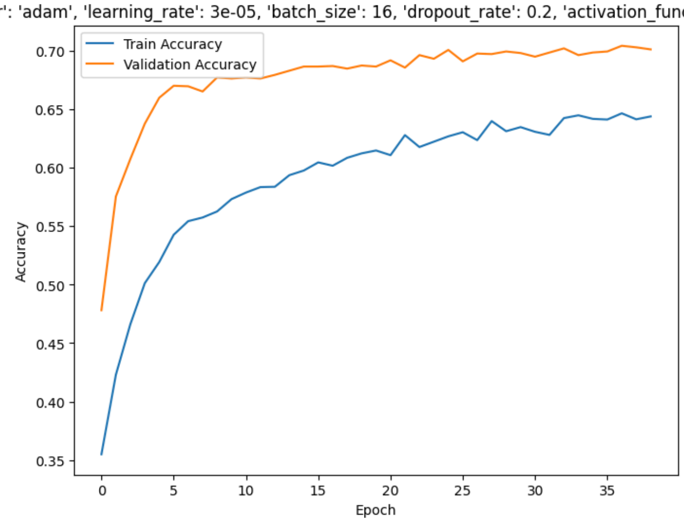
Configuration 4: {'optimizer': 'adam', 'learning_rate': 3e-05, 'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'relu', 'use_batchnorm': True}



Configuration 5: {'optimizer': 'adamw', 'learning_rate': 1e-05, 'batch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': True}

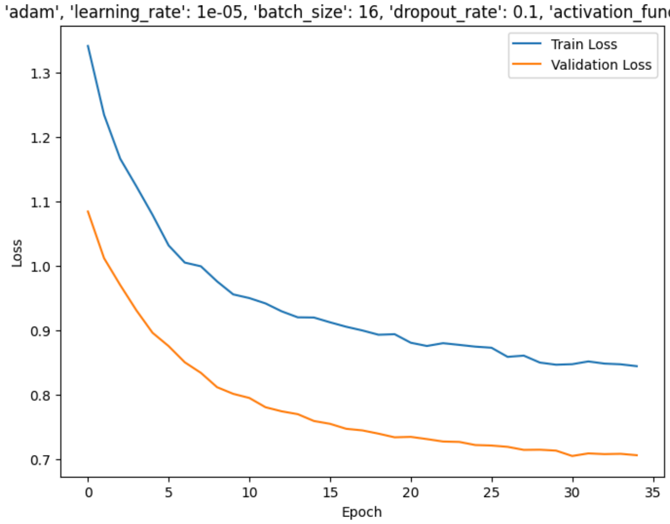


Configuration 6: {'optimizer': 'adam', 'learning_rate': 3e-05, 'batch_size': 16, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_batchnorm': True}

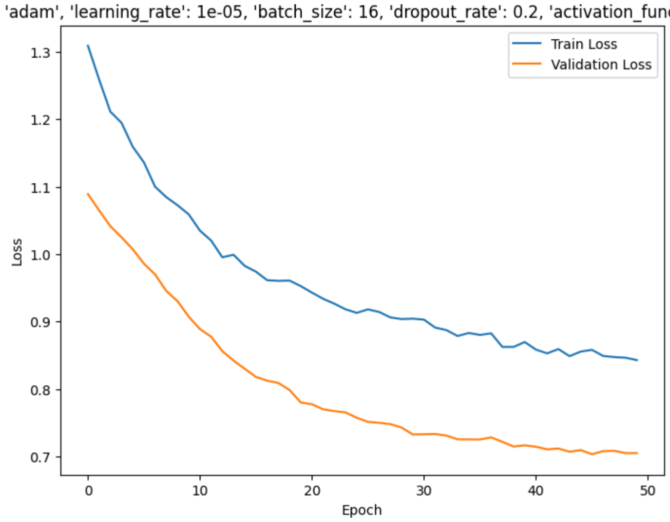


In [ ]:
```
# Plot training and validation loss for each configuration
for i, (history, config) in enumerate(histories):
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title(f"Configuration {i+1}: {config}")
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

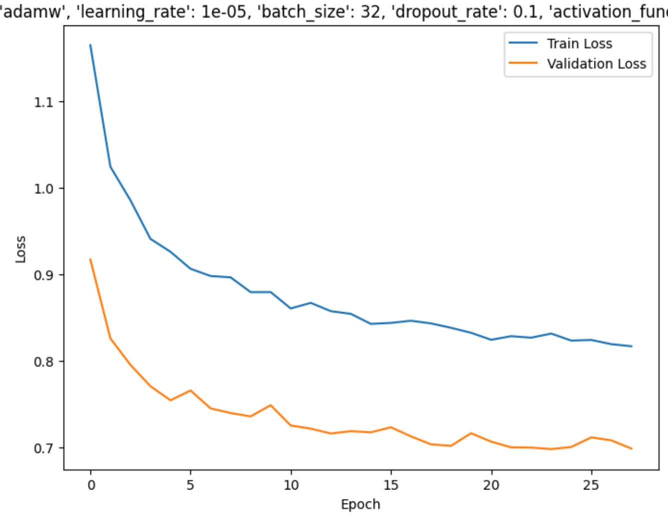
Configuration 1: {'optimizer': 'adam', 'learning_rate': 1e-05, 'batch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'relu', 'use_batchnorm': True}



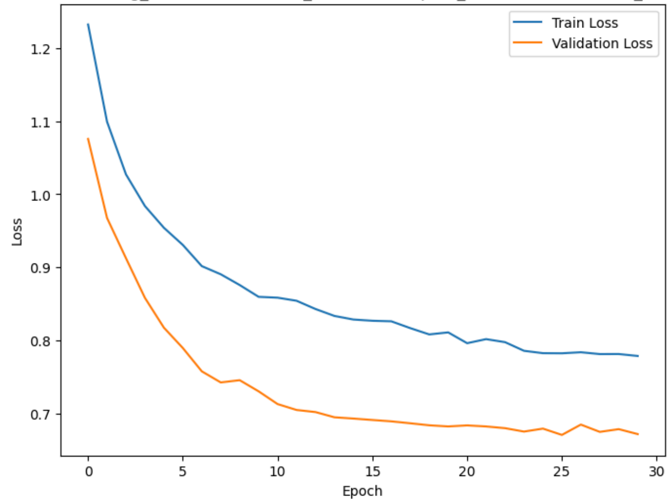
Configuration 2: {'optimizer': 'adam', 'learning_rate': 1e-05, 'batch_size': 16, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_batchnorm': True}



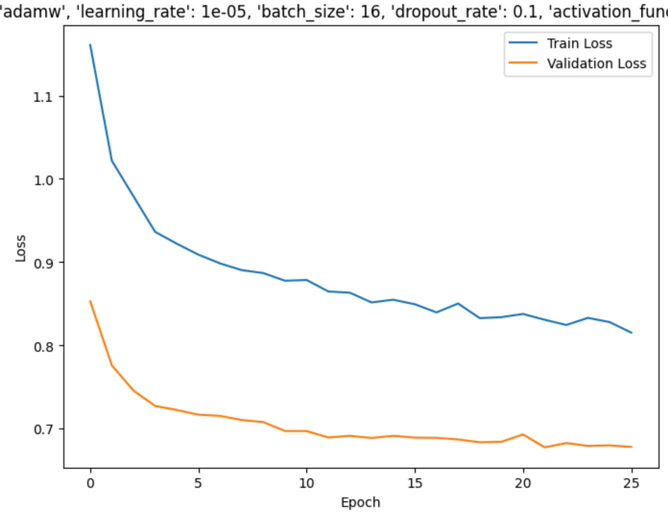
Configuration 3: {'optimizer': 'adamw', 'learning_rate': 1e-05, 'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': True}



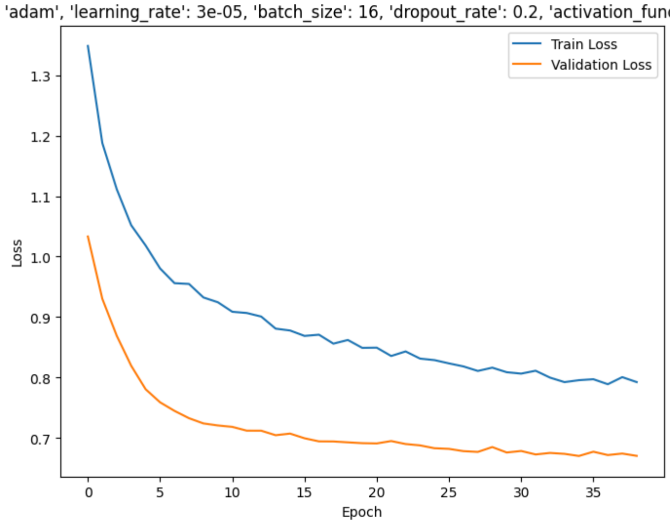
Configuration 4: {'optimizer': 'adam', 'learning_rate': 3e-05, 'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'relu', 'use_batchnorm': True}



Configuration 5: {'optimizer': 'adamw', 'learning_rate': 1e-05, 'batch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': True}



Configuration 6: {'optimizer': 'adam', 'learning_rate': 3e-05, 'batch_size': 16, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_batchnorm': True}



6.1.4 Batch-Normalized Models: Table of Models

In [ ]:
```
# Create a DataFrame of the results
results_df = pd.DataFrame([{
    'optimizer': res['config']['optimizer'],
    'learning_rate': res['config']['learning_rate'],
    'batch_size': res['config']['batch_size'],
```

```
    'dropout_rate': res['config']['dropout_rate'],
    'activation_function': res['config']['activation_function'],
    'use_batchnorm': res['config']['use_batchnorm'],
    'accuracy': res['accuracy'],
    'loss': res['loss']
} for res in results])

print(results_df)
```

```
   optimizer  learning_rate  batch_size  dropout_rate activation_function  \
0       adam        0.00001          16           0.1                relu
1       adam        0.00001          16           0.1                relu
2       adam        0.00001          16           0.2                relu
3       adam        0.00001          16           0.2                relu
4      adamw        0.00001          32           0.1                tanh
5      adamw        0.00001          32           0.1                tanh
6       adam        0.00003          32           0.1                relu
7       adam        0.00003          32           0.1                relu
8      adamw        0.00001          16           0.1                tanh
9      adamw        0.00001          16           0.1                tanh
10      adam        0.00003          16           0.2                relu
11      adam        0.00003          16           0.2                relu

    use_batchnorm  accuracy      loss
0            True  0.681778  0.726639
1            True  0.681778  0.726639
2            True  0.680000  0.732409
3            True  0.680000  0.732409
4            True  0.664000  0.744429
5            True  0.664000  0.744429
6            True  0.682222  0.708122
7            True  0.682222  0.708122
8            True  0.681333  0.724130
9            True  0.681333  0.724130
10           True  0.687111  0.702287
11           True  0.687111  0.702287
```

6.2 Non-Batch-Normalized Models

6.2.1 Non-Batch-Normalized Models: Training

In [ ]:
```
# Function to get optimizer with specified learning rate
def get_optimizer(name, learning_rate):
    if name == 'adam':
        return tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif name == 'adamw':
        try:
            # For TensorFlow 2.11 and above
            return tf.keras.optimizers.experimental.AdamW(learning_rate=lear
        except AttributeError:
            # For earlier versions, use the experimental namespace
            return tf.keras.optimizers.Adam(learning_rate=learning_rate)  #
    else:
        raise ValueError(f'Unsupported optimizer: {name}')
```

In [ ]:
```python
# Define hyperparameter options
optimizers = ['adam', 'adamw']
learning_rates = [1e-5, 3e-5]
batch_sizes = [16, 32]
dropout_rates = [0.1, 0.2]
activation_functions = ['relu', 'tanh']
use_batchnorm_options = [False] # prevent underfitting

# Limit the number of configurations to manage computational resources
max_configs = 6 # Adjust this number based on your resources
configs = []

for _ in range(max_configs):
    config = {
        'optimizer': random.choice(optimizers),
        'learning_rate': random.choice(learning_rates),
        'batch_size': random.choice(batch_sizes),
        'dropout_rate': random.choice(dropout_rates),
        'activation_function': random.choice(activation_functions),
        'use_batchnorm': random.choice(use_batchnorm_options)
    }
    configs.append(config)

# Load the base RoBERTa model and freeze layers
base_model = TFRobertaModel.from_pretrained('distilroberta-base')
for layer in base_model.layers:
    layer.trainable = False

histories = []
results = []
```

```
Some weights of the PyTorch model were not used when initializing the TF 2.0
model TFRobertaModel: ['lm_head.bias', 'lm_head.dense.weight', 'lm_head.laye
r_norm.bias', 'lm_head.layer_norm.weight', 'lm_head.dense.bias']
- This IS expected if you are initializing TFRobertaModel from a PyTorch mod
el trained on another task or with another architecture (e.g. initializing a
TFBertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing TFRobertaModel from a PyTorch
model that you expect to be exactly identical (e.g. initializing a TFBertFor
SequenceClassification model from a BertForSequenceClassification model).
All the weights of TFRobertaModel were initialized from the PyTorch model.
If your task is similar to the task the model of the checkpoint was trained
on, you can already use TFRobertaModel for predictions without further train
ing.
```

In [ ]:
```python
def run_model(num_epochs, patience, configurations, base_model, additional_l

  # Initialize variables to keep track of the best model
  best_accuracy = 0
  best_model = None
  best_config = None
  best_y_pred = None

  # Loop over configurations
  for i, config in enumerate(configurations):
```

```
    print(f"Training configuration {i+1}/{len(configurations)}: {config}")

    # Build the model
    model = build_model(
        base_model,
        activation_function=config['activation_function'],
        dropout_rate=config['dropout_rate'],
        use_batchnorm=config['use_batchnorm'],
        additional_layers= additional_layers
    )

    # Get optimizer with specified learning rate
    optimizer = get_optimizer(config['optimizer'], learning_rate=config['l

    # Compile the model
    model.compile(
        optimizer=optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    # Create datasets with specified batch size
    batch_size = config['batch_size']
    train_dataset = create_tf_dataset(train_input_ids, train_attention_mas
    val_dataset = create_tf_dataset(val_input_ids, val_attention_mask, val
    test_dataset = create_tf_dataset(test_input_ids, test_attention_mask,

    # Set up callbacks
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',

    # Fit the model using the validation set
    history = model.fit(
        train_dataset,
        epochs=num_epochs,
        validation_data=val_dataset,
        callbacks=[early_stopping],
        verbose=1
    )

    # Record the history and config
    histories.append((history, config))

    # Evaluate the final model on the test set
    loss, accuracy = model.evaluate(test_dataset)
    print(f"Test accuracy: {accuracy}")

    # Store results
    results.append({
        'config': config,
        'accuracy': accuracy,
        'loss': loss,
    })

    # Generate predictions for the test set
    y_pred_probs = model.predict(test_dataset)
    y_pred = np.argmax(y_pred_probs, axis=1)
```

```
    # Flatten test labels
    y_true = test_labels

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)

    # Classification report
    report = classification_report(y_true, y_pred, target_names=['Negative
    print(f"Classification Report for Configuration {i+1}:\n{report}")

    # Plot confusion matrix
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Negat
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.title(f'Confusion Matrix for Configuration {i+1}')
    plt.show()

    # Store results
    results.append({
        'config': config,
        'accuracy': accuracy,
        'loss': loss,
        'confusion_matrix': cm,
        'classification_report': report
    })

    # If current model is better, save it
    if accuracy > best_accuracy:
        print("New best model found!")
        # Clear previous best model from memory if exists
        if best_model is not None:
            tf.keras.backend.clear_session()
            del best_model

        best_accuracy = accuracy
        best_model = model
        best_config = config
        best_y_pred = y_pred
        best_y_true = y_true
        # Also store confusion matrix and classification report
        best_cm = confusion_matrix(y_true, y_pred)
        best_report = classification_report(y_true, y_pred, target_names=[
    else:
        # Clear current model from memory
        tf.keras.backend.clear_session()
        del model
return best_accuracy, best_model, best_config, best_y_pred, best_y_true, b
```

6.2.3 Non-Batch-Normalized Models: Visualizations

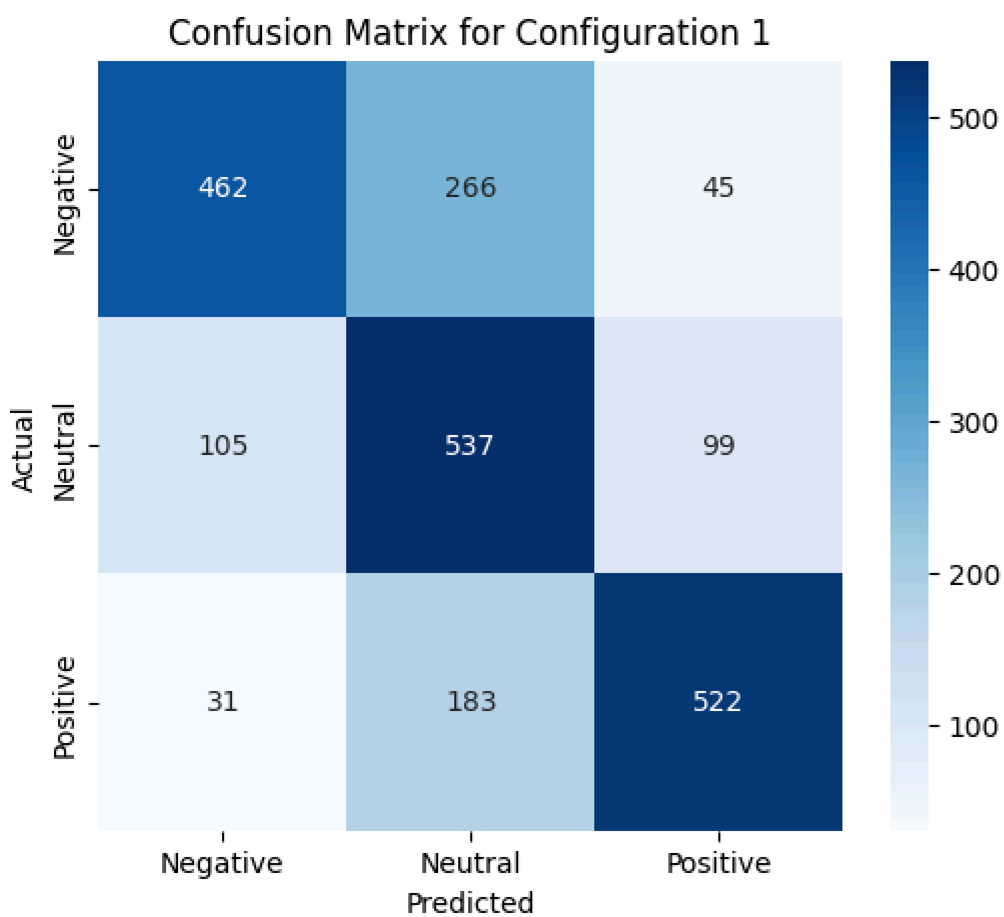
6.2.4 Non-Batch-Normalized Models: Table of Models

In [ ]:

```
# Run the model
best_accuracy, best_model, best_config, best_y_pred, best_y_true, best_cm, b
```

```
Training configuration 1/6: {'optimizer': 'adamw', 'learning_rate': 3e-05,
'batch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_b
atchnorm': False}
Epoch 1/80
657/657 [==============================] - 73s 101ms/step - loss: 1.0993 - a
ccuracy: 0.3907 - val_loss: 0.9956 - val_accuracy: 0.5182
Epoch 2/80
657/657 [==============================] - 65s 99ms/step - loss: 0.9773 - ac
curacy: 0.5142 - val_loss: 0.8534 - val_accuracy: 0.6067
Epoch 3/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8854 - ac
curacy: 0.5789 - val_loss: 0.7991 - val_accuracy: 0.6120
Epoch 4/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8366 - ac
curacy: 0.6171 - val_loss: 0.7604 - val_accuracy: 0.6422
Epoch 5/80
657/657 [==============================] - 65s 98ms/step - loss: 0.8206 - ac
curacy: 0.6250 - val_loss: 0.7507 - val_accuracy: 0.6476
Epoch 6/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8045 - ac
curacy: 0.6375 - val_loss: 0.7394 - val_accuracy: 0.6542
Epoch 7/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7951 - ac
curacy: 0.6441 - val_loss: 0.7198 - val_accuracy: 0.6671
Epoch 8/80
657/657 [==============================] - 64s 97ms/step - loss: 0.7969 - ac
curacy: 0.6417 - val_loss: 0.7224 - val_accuracy: 0.6698
Epoch 9/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7824 - ac
curacy: 0.6493 - val_loss: 0.7193 - val_accuracy: 0.6662
Epoch 10/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7811 - ac
curacy: 0.6508 - val_loss: 0.7222 - val_accuracy: 0.6680
Epoch 11/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7776 - ac
curacy: 0.6549 - val_loss: 0.7002 - val_accuracy: 0.6853
Epoch 12/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7664 - ac
curacy: 0.6570 - val_loss: 0.7019 - val_accuracy: 0.6849
Epoch 13/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7627 - ac
curacy: 0.6632 - val_loss: 0.7052 - val_accuracy: 0.6773
Epoch 14/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7631 - ac
curacy: 0.6637 - val_loss: 0.6993 - val_accuracy: 0.6853
Epoch 15/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7620 - ac
curacy: 0.6630 - val_loss: 0.6938 - val_accuracy: 0.6867
Epoch 16/80
657/657 [==============================] - 66s 100ms/step - loss: 0.7544 - a
ccuracy: 0.6654 - val_loss: 0.6934 - val_accuracy: 0.6844
Epoch 17/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7601 - ac
curacy: 0.6652 - val_loss: 0.6895 - val_accuracy: 0.6933
Epoch 18/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7556 - ac
```

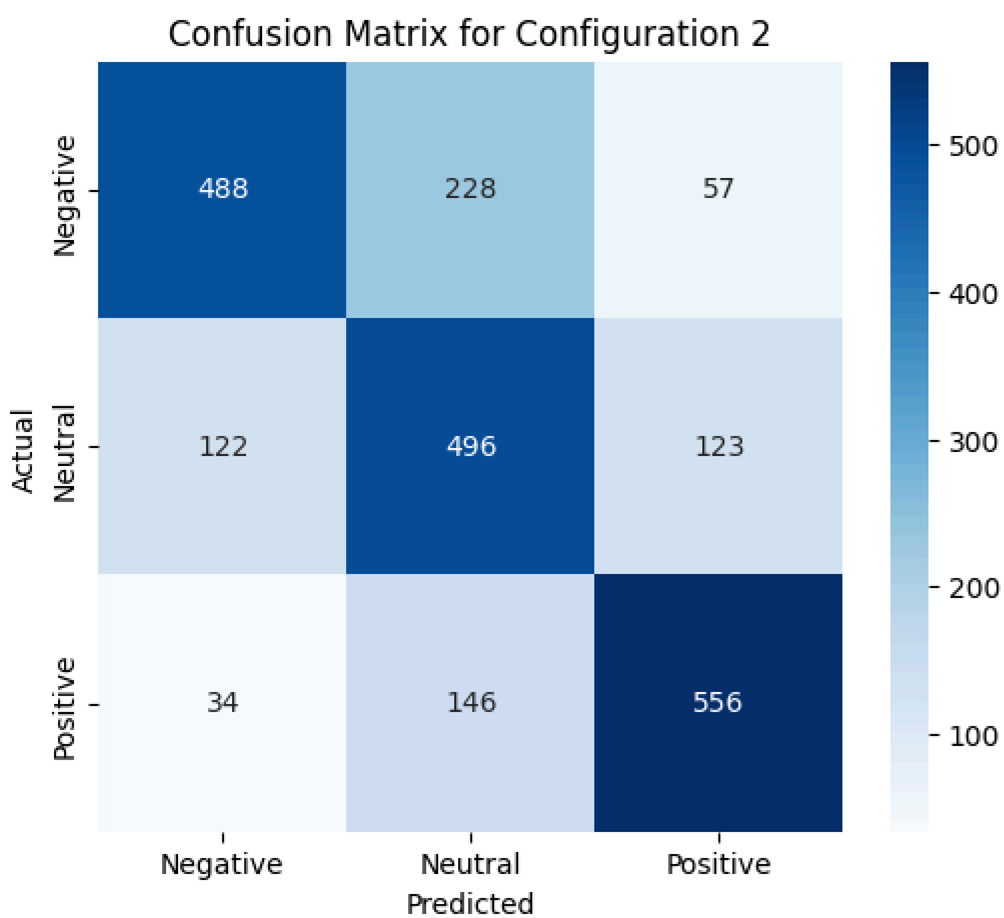
```
curacy: 0.6631 - val_loss: 0.6907 - val_accuracy: 0.6889
Epoch 19/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7436 - ac
curacy: 0.6692 - val_loss: 0.6926 - val_accuracy: 0.6867
Epoch 20/80
657/657 [==============================] - 64s 97ms/step - loss: 0.7508 - ac
curacy: 0.6688 - val_loss: 0.6899 - val_accuracy: 0.6880
Epoch 21/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7521 - ac
curacy: 0.6662 - val_loss: 0.6903 - val_accuracy: 0.6907
141/141 [==============================] - 10s 74ms/step - loss: 0.7254 - ac
curacy: 0.6760
Test accuracy: 0.6759999990463257
141/141 [==============================] - 12s 73ms/step
Classification Report for Configuration 1:
              precision    recall  f1-score   support

    Negative       0.77      0.60      0.67       773
     Neutral       0.54      0.72      0.62       741
    Positive       0.78      0.71      0.74       736

    accuracy                           0.68      2250
   macro avg       0.70      0.68      0.68      2250
weighted avg       0.70      0.68      0.68      2250
```



```
New best model found!
Training configuration 2/6: {'optimizer': 'adamw', 'learning_rate': 3e-05,
'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_b
atchnorm': False}
Epoch 1/80
329/329 [==============================] - 72s 194ms/step - loss: 1.1209 - a
ccuracy: 0.3676 - val_loss: 1.0282 - val_accuracy: 0.5738
Epoch 2/80
329/329 [==============================] - 60s 184ms/step - loss: 1.0380 - a
ccuracy: 0.4625 - val_loss: 0.9283 - val_accuracy: 0.5907
Epoch 3/80
329/329 [==============================] - 61s 186ms/step - loss: 0.9422 - a
ccuracy: 0.5429 - val_loss: 0.8324 - val_accuracy: 0.6298
Epoch 4/80
329/329 [==============================] - 60s 182ms/step - loss: 0.8771 - a
ccuracy: 0.5938 - val_loss: 0.7770 - val_accuracy: 0.6453
Epoch 5/80
329/329 [==============================] - 61s 184ms/step - loss: 0.8437 - a
ccuracy: 0.6037 - val_loss: 0.7508 - val_accuracy: 0.6631
Epoch 6/80
329/329 [==============================] - 60s 183ms/step - loss: 0.8198 - a
ccuracy: 0.6279 - val_loss: 0.7322 - val_accuracy: 0.6764
Epoch 7/80
329/329 [==============================] - 61s 186ms/step - loss: 0.8069 - a
ccuracy: 0.6335 - val_loss: 0.7208 - val_accuracy: 0.6796
Epoch 8/80
329/329 [==============================] - 61s 185ms/step - loss: 0.8039 - a
ccuracy: 0.6338 - val_loss: 0.7129 - val_accuracy: 0.6813
Epoch 9/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7923 - a
ccuracy: 0.6405 - val_loss: 0.7175 - val_accuracy: 0.6778
Epoch 10/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7834 - a
ccuracy: 0.6409 - val_loss: 0.7073 - val_accuracy: 0.6884
Epoch 11/80
329/329 [==============================] - 62s 187ms/step - loss: 0.7881 - a
ccuracy: 0.6470 - val_loss: 0.7088 - val_accuracy: 0.6849
Epoch 12/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7698 - a
ccuracy: 0.6571 - val_loss: 0.7033 - val_accuracy: 0.6880
Epoch 13/80
329/329 [==============================] - 62s 187ms/step - loss: 0.7778 - a
ccuracy: 0.6505 - val_loss: 0.7005 - val_accuracy: 0.6911
Epoch 14/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7690 - a
ccuracy: 0.6579 - val_loss: 0.6930 - val_accuracy: 0.6951
Epoch 15/80
329/329 [==============================] - 61s 184ms/step - loss: 0.7641 - a
ccuracy: 0.6619 - val_loss: 0.6933 - val_accuracy: 0.6933
Epoch 16/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7692 - a
ccuracy: 0.6568 - val_loss: 0.6938 - val_accuracy: 0.6933
Epoch 17/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7666 - a
ccuracy: 0.6602 - val_loss: 0.6976 - val_accuracy: 0.6933
Epoch 18/80
```

```
329/329 [==============================] - 60s 183ms/step - loss: 0.7639 - a
ccuracy: 0.6630 - val_loss: 0.6894 - val_accuracy: 0.6924
Epoch 19/80
329/329 [==============================] - 60s 184ms/step - loss: 0.7599 - a
ccuracy: 0.6647 - val_loss: 0.6859 - val_accuracy: 0.6978
Epoch 20/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7608 - a
ccuracy: 0.6610 - val_loss: 0.6957 - val_accuracy: 0.6898
Epoch 21/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7588 - a
ccuracy: 0.6638 - val_loss: 0.6916 - val_accuracy: 0.6960
Epoch 22/80
329/329 [==============================] - 61s 184ms/step - loss: 0.7565 - a
ccuracy: 0.6661 - val_loss: 0.6838 - val_accuracy: 0.7040
Epoch 23/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7509 - a
ccuracy: 0.6687 - val_loss: 0.6808 - val_accuracy: 0.7000
Epoch 24/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7503 - a
ccuracy: 0.6690 - val_loss: 0.6809 - val_accuracy: 0.7031
Epoch 25/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7475 - a
ccuracy: 0.6700 - val_loss: 0.6879 - val_accuracy: 0.6960
Epoch 26/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7523 - a
ccuracy: 0.6660 - val_loss: 0.6822 - val_accuracy: 0.7000
Epoch 27/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7454 - a
ccuracy: 0.6707 - val_loss: 0.6770 - val_accuracy: 0.7071
Epoch 28/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7504 - a
ccuracy: 0.6703 - val_loss: 0.6865 - val_accuracy: 0.6978
Epoch 29/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7455 - a
ccuracy: 0.6693 - val_loss: 0.6821 - val_accuracy: 0.6978
Epoch 30/80
329/329 [==============================] - 61s 184ms/step - loss: 0.7472 - a
ccuracy: 0.6678 - val_loss: 0.6844 - val_accuracy: 0.6969
Epoch 31/80
329/329 [==============================] - 62s 188ms/step - loss: 0.7436 - a
ccuracy: 0.6716 - val_loss: 0.6756 - val_accuracy: 0.7027
Epoch 32/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7443 - a
ccuracy: 0.6681 - val_loss: 0.6736 - val_accuracy: 0.7036
Epoch 33/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7423 - a
ccuracy: 0.6772 - val_loss: 0.6753 - val_accuracy: 0.7018
Epoch 34/80
329/329 [==============================] - 61s 187ms/step - loss: 0.7494 - a
ccuracy: 0.6681 - val_loss: 0.6767 - val_accuracy: 0.7000
Epoch 35/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7412 - a
ccuracy: 0.6720 - val_loss: 0.6802 - val_accuracy: 0.6947
Epoch 36/80
329/329 [==============================] - 60s 184ms/step - loss: 0.7406 - a
ccuracy: 0.6704 - val_loss: 0.6726 - val_accuracy: 0.7027
```

```
Epoch 37/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7415 - a
ccuracy: 0.6709 - val_loss: 0.6751 - val_accuracy: 0.7013
Epoch 38/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7388 - a
ccuracy: 0.6723 - val_loss: 0.6742 - val_accuracy: 0.6982
Epoch 39/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7421 - a
ccuracy: 0.6699 - val_loss: 0.6779 - val_accuracy: 0.6960
Epoch 40/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7393 - a
ccuracy: 0.6774 - val_loss: 0.6749 - val_accuracy: 0.7013
71/71 [==============================] - 10s 138ms/step - loss: 0.7090 - acc
uracy: 0.6844
Test accuracy: 0.6844444274902344
71/71 [==============================] - 12s 135ms/step
Classification Report for Configuration 2:
              precision    recall  f1-score   support

    Negative       0.76      0.63      0.69       773
     Neutral       0.57      0.67      0.62       741
    Positive       0.76      0.76      0.76       736

    accuracy                           0.68      2250
   macro avg       0.69      0.69      0.69      2250
weighted avg       0.70      0.68      0.69      2250
```

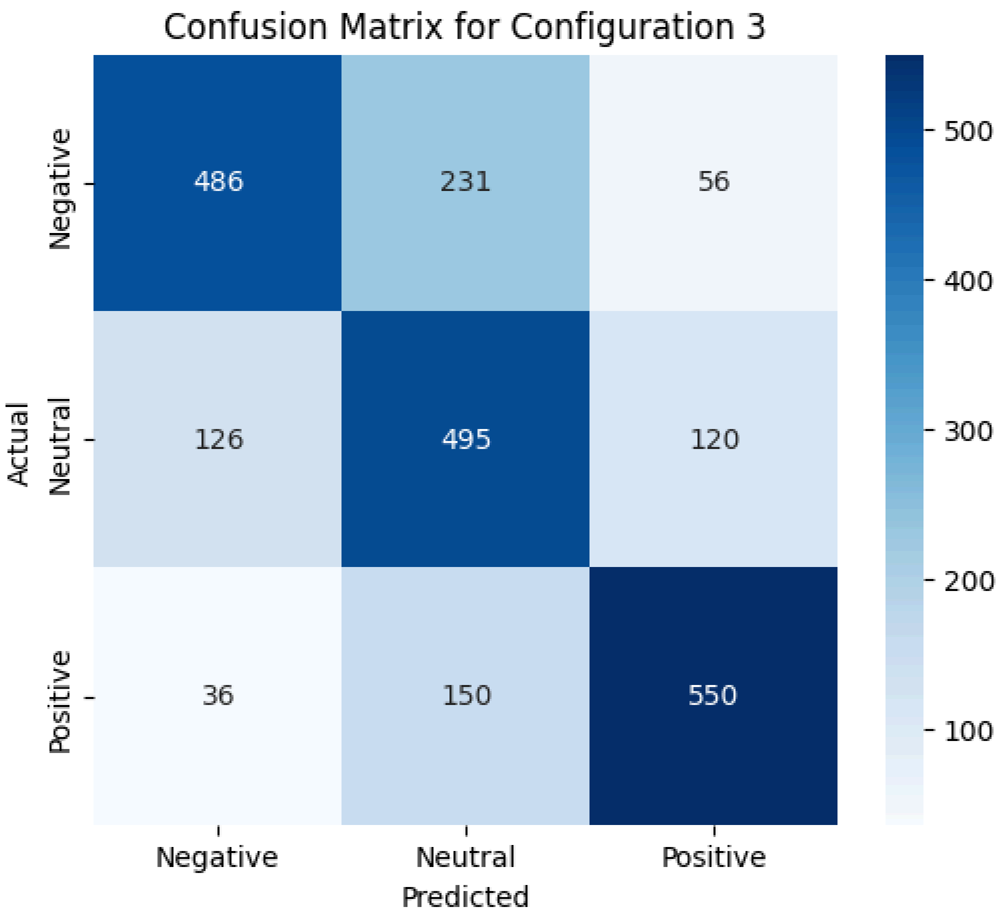


```
New best model found!
Training configuration 3/6: {'optimizer': 'adamw', 'learning_rate': 1e-05,
'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_b
atchnorm': False}
Epoch 1/80
329/329 [==============================] - 73s 193ms/step - loss: 1.1361 - a
ccuracy: 0.3505 - val_loss: 1.0738 - val_accuracy: 0.4000
Epoch 2/80
329/329 [==============================] - 62s 187ms/step - loss: 1.1099 - a
ccuracy: 0.3741 - val_loss: 1.0474 - val_accuracy: 0.5320
Epoch 3/80
329/329 [==============================] - 60s 183ms/step - loss: 1.0849 - a
ccuracy: 0.4051 - val_loss: 1.0149 - val_accuracy: 0.5960
Epoch 4/80
329/329 [==============================] - 60s 183ms/step - loss: 1.0496 - a
ccuracy: 0.4467 - val_loss: 0.9792 - val_accuracy: 0.6044
Epoch 5/80
329/329 [==============================] - 60s 184ms/step - loss: 1.0173 - a
ccuracy: 0.4775 - val_loss: 0.9390 - val_accuracy: 0.6227
Epoch 6/80
329/329 [==============================] - 60s 184ms/step - loss: 0.9832 - a
ccuracy: 0.5094 - val_loss: 0.9004 - val_accuracy: 0.6302
Epoch 7/80
329/329 [==============================] - 61s 185ms/step - loss: 0.9536 - a
ccuracy: 0.5308 - val_loss: 0.8660 - val_accuracy: 0.6396
Epoch 8/80
329/329 [==============================] - 61s 186ms/step - loss: 0.9247 - a
ccuracy: 0.5549 - val_loss: 0.8363 - val_accuracy: 0.6422
Epoch 9/80
329/329 [==============================] - 61s 186ms/step - loss: 0.9010 - a
ccuracy: 0.5650 - val_loss: 0.8120 - val_accuracy: 0.6462
Epoch 10/80
329/329 [==============================] - 61s 186ms/step - loss: 0.8818 - a
ccuracy: 0.5771 - val_loss: 0.7985 - val_accuracy: 0.6462
Epoch 11/80
329/329 [==============================] - 61s 186ms/step - loss: 0.8631 - a
ccuracy: 0.5956 - val_loss: 0.7805 - val_accuracy: 0.6533
Epoch 12/80
329/329 [==============================] - 60s 183ms/step - loss: 0.8477 - a
ccuracy: 0.6029 - val_loss: 0.7685 - val_accuracy: 0.6560
Epoch 13/80
329/329 [==============================] - 61s 186ms/step - loss: 0.8468 - a
ccuracy: 0.6103 - val_loss: 0.7598 - val_accuracy: 0.6622
Epoch 14/80
329/329 [==============================] - 61s 186ms/step - loss: 0.8340 - a
ccuracy: 0.6140 - val_loss: 0.7503 - val_accuracy: 0.6649
Epoch 15/80
329/329 [==============================] - 60s 183ms/step - loss: 0.8351 - a
ccuracy: 0.6130 - val_loss: 0.7435 - val_accuracy: 0.6689
Epoch 16/80
329/329 [==============================] - 60s 183ms/step - loss: 0.8267 - a
ccuracy: 0.6209 - val_loss: 0.7404 - val_accuracy: 0.6684
Epoch 17/80
329/329 [==============================] - 60s 183ms/step - loss: 0.8197 - a
ccuracy: 0.6273 - val_loss: 0.7397 - val_accuracy: 0.6707
Epoch 18/80
```

```
329/329 [==============================] - 61s 186ms/step - loss: 0.8123 - a
ccuracy: 0.6286 - val_loss: 0.7303 - val_accuracy: 0.6773
Epoch 19/80
329/329 [==============================] - 60s 183ms/step - loss: 0.8092 - a
ccuracy: 0.6284 - val_loss: 0.7276 - val_accuracy: 0.6760
Epoch 20/80
329/329 [==============================] - 61s 186ms/step - loss: 0.8087 - a
ccuracy: 0.6252 - val_loss: 0.7249 - val_accuracy: 0.6751
Epoch 21/80
329/329 [==============================] - 61s 185ms/step - loss: 0.8006 - a
ccuracy: 0.6372 - val_loss: 0.7235 - val_accuracy: 0.6782
Epoch 22/80
329/329 [==============================] - 61s 186ms/step - loss: 0.7969 - a
ccuracy: 0.6372 - val_loss: 0.7184 - val_accuracy: 0.6809
Epoch 23/80
329/329 [==============================] - 62s 187ms/step - loss: 0.7977 - a
ccuracy: 0.6368 - val_loss: 0.7168 - val_accuracy: 0.6791
Epoch 24/80
329/329 [==============================] - 61s 186ms/step - loss: 0.7940 - a
ccuracy: 0.6385 - val_loss: 0.7120 - val_accuracy: 0.6827
Epoch 25/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7844 - a
ccuracy: 0.6480 - val_loss: 0.7113 - val_accuracy: 0.6836
Epoch 26/80
329/329 [==============================] - 60s 184ms/step - loss: 0.7907 - a
ccuracy: 0.6415 - val_loss: 0.7086 - val_accuracy: 0.6840
Epoch 27/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7879 - a
ccuracy: 0.6443 - val_loss: 0.7094 - val_accuracy: 0.6822
Epoch 28/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7863 - a
ccuracy: 0.6454 - val_loss: 0.7066 - val_accuracy: 0.6849
Epoch 29/80
329/329 [==============================] - 61s 186ms/step - loss: 0.7842 - a
ccuracy: 0.6460 - val_loss: 0.7091 - val_accuracy: 0.6849
Epoch 30/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7775 - a
ccuracy: 0.6551 - val_loss: 0.7024 - val_accuracy: 0.6880
Epoch 31/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7888 - a
ccuracy: 0.6456 - val_loss: 0.7029 - val_accuracy: 0.6840
Epoch 32/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7811 - a
ccuracy: 0.6513 - val_loss: 0.7036 - val_accuracy: 0.6867
Epoch 33/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7752 - a
ccuracy: 0.6521 - val_loss: 0.6987 - val_accuracy: 0.6884
Epoch 34/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7759 - a
ccuracy: 0.6562 - val_loss: 0.7007 - val_accuracy: 0.6889
Epoch 35/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7756 - a
ccuracy: 0.6577 - val_loss: 0.7008 - val_accuracy: 0.6889
Epoch 36/80
329/329 [==============================] - 61s 186ms/step - loss: 0.7736 - a
ccuracy: 0.6626 - val_loss: 0.6966 - val_accuracy: 0.6902
```

```
Epoch 37/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7680 - a
ccuracy: 0.6582 - val_loss: 0.6961 - val_accuracy: 0.6907
Epoch 38/80
329/329 [==============================] - 60s 184ms/step - loss: 0.7687 - a
ccuracy: 0.6596 - val_loss: 0.6949 - val_accuracy: 0.6920
Epoch 39/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7742 - a
ccuracy: 0.6533 - val_loss: 0.6936 - val_accuracy: 0.6933
Epoch 40/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7707 - a
ccuracy: 0.6583 - val_loss: 0.6950 - val_accuracy: 0.6951
Epoch 41/80
329/329 [==============================] - 60s 184ms/step - loss: 0.7708 - a
ccuracy: 0.6521 - val_loss: 0.6932 - val_accuracy: 0.6920
Epoch 42/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7722 - a
ccuracy: 0.6561 - val_loss: 0.6937 - val_accuracy: 0.6938
Epoch 43/80
329/329 [==============================] - 61s 186ms/step - loss: 0.7690 - a
ccuracy: 0.6592 - val_loss: 0.6924 - val_accuracy: 0.6964
Epoch 44/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7671 - a
ccuracy: 0.6608 - val_loss: 0.6942 - val_accuracy: 0.6942
Epoch 45/80
329/329 [==============================] - 61s 184ms/step - loss: 0.7627 - a
ccuracy: 0.6572 - val_loss: 0.6927 - val_accuracy: 0.6933
Epoch 46/80
329/329 [==============================] - 61s 186ms/step - loss: 0.7606 - a
ccuracy: 0.6599 - val_loss: 0.6896 - val_accuracy: 0.6960
Epoch 47/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7632 - a
ccuracy: 0.6594 - val_loss: 0.6922 - val_accuracy: 0.6938
Epoch 48/80
329/329 [==============================] - 61s 187ms/step - loss: 0.7617 - a
ccuracy: 0.6627 - val_loss: 0.6901 - val_accuracy: 0.6960
Epoch 49/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7638 - a
ccuracy: 0.6605 - val_loss: 0.6877 - val_accuracy: 0.6960
Epoch 50/80
329/329 [==============================] - 61s 186ms/step - loss: 0.7611 - a
ccuracy: 0.6611 - val_loss: 0.6879 - val_accuracy: 0.6973
Epoch 51/80
329/329 [==============================] - 61s 186ms/step - loss: 0.7566 - a
ccuracy: 0.6668 - val_loss: 0.6865 - val_accuracy: 0.6938
Epoch 52/80
329/329 [==============================] - 61s 186ms/step - loss: 0.7543 - a
ccuracy: 0.6674 - val_loss: 0.6868 - val_accuracy: 0.6951
Epoch 53/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7556 - a
ccuracy: 0.6629 - val_loss: 0.6886 - val_accuracy: 0.6982
Epoch 54/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7520 - a
ccuracy: 0.6677 - val_loss: 0.6861 - val_accuracy: 0.6969
Epoch 55/80
329/329 [==============================] - 61s 186ms/step - loss: 0.7600 - a
```

```
ccuracy: 0.6619 - val_loss: 0.6853 - val_accuracy: 0.6964
Epoch 56/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7530 - a
ccuracy: 0.6676 - val_loss: 0.6831 - val_accuracy: 0.6947
Epoch 57/80
329/329 [==============================] - 61s 187ms/step - loss: 0.7577 - a
ccuracy: 0.6670 - val_loss: 0.6821 - val_accuracy: 0.6969
Epoch 58/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7543 - a
ccuracy: 0.6716 - val_loss: 0.6836 - val_accuracy: 0.6960
Epoch 59/80
329/329 [==============================] - 61s 184ms/step - loss: 0.7498 - a
ccuracy: 0.6684 - val_loss: 0.6843 - val_accuracy: 0.6987
Epoch 60/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7582 - a
ccuracy: 0.6682 - val_loss: 0.6834 - val_accuracy: 0.6973
Epoch 61/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7504 - a
ccuracy: 0.6683 - val_loss: 0.6828 - val_accuracy: 0.6978
71/71 [==============================] - 10s 140ms/step - loss: 0.7195 - acc
uracy: 0.6804
Test accuracy: 0.6804444193840027
71/71 [==============================] - 11s 136ms/step
Classification Report for Configuration 3:
              precision    recall  f1-score   support

    Negative       0.75      0.63      0.68       773
     Neutral       0.57      0.67      0.61       741
    Positive       0.76      0.75      0.75       736

    accuracy                           0.68      2250
   macro avg       0.69      0.68      0.68      2250
weighted avg       0.69      0.68      0.68      2250
```

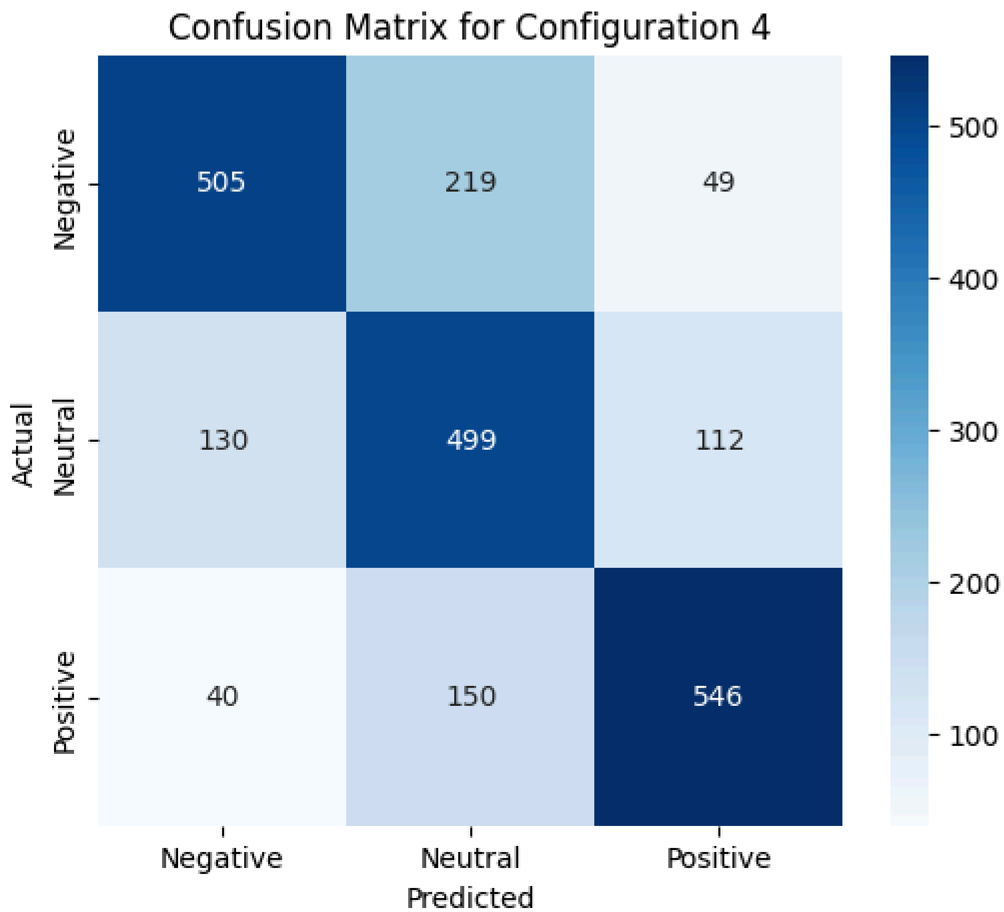

Confusion Matrix for Configuration 3
Negative
Neutral
Positive
Actual
486
231
56
126
495
120
36
150
550
Negative
Neutral
Positive
Predicted
500
400
300
200
100

```
Training configuration 4/6: {'optimizer': 'adam', 'learning_rate': 3e-05, 'b
atch_size': 32, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_bat
chnorm': False}
Epoch 1/80
329/329 [==============================] - 69s 191ms/step - loss: 1.1155 - a
ccuracy: 0.3318 - val_loss: 1.0964 - val_accuracy: 0.3342
Epoch 2/80
329/329 [==============================] - 60s 183ms/step - loss: 1.1025 - a
ccuracy: 0.3490 - val_loss: 1.0937 - val_accuracy: 0.3324
Epoch 3/80
329/329 [==============================] - 62s 187ms/step - loss: 1.0987 - a
ccuracy: 0.3498 - val_loss: 1.0883 - val_accuracy: 0.3840
Epoch 4/80
329/329 [==============================] - 60s 183ms/step - loss: 1.0907 - a
ccuracy: 0.3711 - val_loss: 1.0680 - val_accuracy: 0.4951
Epoch 5/80
329/329 [==============================] - 61s 186ms/step - loss: 1.0743 - a
ccuracy: 0.4113 - val_loss: 1.0329 - val_accuracy: 0.5276
Epoch 6/80
329/329 [==============================] - 60s 183ms/step - loss: 1.0440 - a
ccuracy: 0.4538 - val_loss: 0.9725 - val_accuracy: 0.5800
Epoch 7/80
329/329 [==============================] - 61s 184ms/step - loss: 1.0057 - a
ccuracy: 0.4925 - val_loss: 0.9113 - val_accuracy: 0.5809
Epoch 8/80
329/329 [==============================] - 62s 187ms/step - loss: 0.9552 - a
ccuracy: 0.5353 - val_loss: 0.8570 - val_accuracy: 0.6058
Epoch 9/80
329/329 [==============================] - 61s 185ms/step - loss: 0.9235 - a
ccuracy: 0.5483 - val_loss: 0.8266 - val_accuracy: 0.6249
Epoch 10/80
329/329 [==============================] - 61s 187ms/step - loss: 0.9071 - a
ccuracy: 0.5683 - val_loss: 0.8044 - val_accuracy: 0.6311
Epoch 11/80
329/329 [==============================] - 60s 183ms/step - loss: 0.8834 - a
ccuracy: 0.5806 - val_loss: 0.7896 - val_accuracy: 0.6382
Epoch 12/80
329/329 [==============================] - 61s 184ms/step - loss: 0.8752 - a
ccuracy: 0.5900 - val_loss: 0.7806 - val_accuracy: 0.6436
Epoch 13/80
329/329 [==============================] - 61s 184ms/step - loss: 0.8619 - a
ccuracy: 0.5988 - val_loss: 0.7704 - val_accuracy: 0.6480
Epoch 14/80
329/329 [==============================] - 60s 181ms/step - loss: 0.8536 - a
ccuracy: 0.6059 - val_loss: 0.7618 - val_accuracy: 0.6569
Epoch 15/80
329/329 [==============================] - 60s 183ms/step - loss: 0.8500 - a
ccuracy: 0.6100 - val_loss: 0.7476 - val_accuracy: 0.6622
Epoch 16/80
329/329 [==============================] - 59s 180ms/step - loss: 0.8396 - a
ccuracy: 0.6161 - val_loss: 0.7480 - val_accuracy: 0.6676
Epoch 17/80
329/329 [==============================] - 60s 184ms/step - loss: 0.8270 - a
ccuracy: 0.6250 - val_loss: 0.7364 - val_accuracy: 0.6733
Epoch 18/80
329/329 [==============================] - 61s 185ms/step - loss: 0.8216 - a
```

```
ccuracy: 0.6270 - val_loss: 0.7375 - val_accuracy: 0.6742
Epoch 19/80
329/329 [==============================] - 60s 183ms/step - loss: 0.8101 - a
ccuracy: 0.6372 - val_loss: 0.7276 - val_accuracy: 0.6733
Epoch 20/80
329/329 [==============================] - 60s 183ms/step - loss: 0.8089 - a
ccuracy: 0.6321 - val_loss: 0.7247 - val_accuracy: 0.6720
Epoch 21/80
329/329 [==============================] - 61s 186ms/step - loss: 0.8082 - a
ccuracy: 0.6360 - val_loss: 0.7208 - val_accuracy: 0.6822
Epoch 22/80
329/329 [==============================] - 60s 181ms/step - loss: 0.8000 - a
ccuracy: 0.6409 - val_loss: 0.7230 - val_accuracy: 0.6773
Epoch 23/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7949 - a
ccuracy: 0.6411 - val_loss: 0.7104 - val_accuracy: 0.6884
Epoch 24/80
329/329 [==============================] - 60s 182ms/step - loss: 0.8006 - a
ccuracy: 0.6474 - val_loss: 0.7097 - val_accuracy: 0.6880
Epoch 25/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7916 - a
ccuracy: 0.6463 - val_loss: 0.7085 - val_accuracy: 0.6831
Epoch 26/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7864 - a
ccuracy: 0.6458 - val_loss: 0.7031 - val_accuracy: 0.6836
Epoch 27/80
329/329 [==============================] - 59s 181ms/step - loss: 0.7835 - a
ccuracy: 0.6528 - val_loss: 0.7034 - val_accuracy: 0.6822
Epoch 28/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7830 - a
ccuracy: 0.6551 - val_loss: 0.7039 - val_accuracy: 0.6844
Epoch 29/80
329/329 [==============================] - 59s 180ms/step - loss: 0.7778 - a
ccuracy: 0.6572 - val_loss: 0.6976 - val_accuracy: 0.6813
Epoch 30/80
329/329 [==============================] - 59s 180ms/step - loss: 0.7718 - a
ccuracy: 0.6598 - val_loss: 0.6959 - val_accuracy: 0.6862
Epoch 31/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7684 - a
ccuracy: 0.6582 - val_loss: 0.6988 - val_accuracy: 0.6827
Epoch 32/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7642 - a
ccuracy: 0.6648 - val_loss: 0.6940 - val_accuracy: 0.6871
Epoch 33/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7668 - a
ccuracy: 0.6617 - val_loss: 0.6914 - val_accuracy: 0.6871
Epoch 34/80
329/329 [==============================] - 61s 184ms/step - loss: 0.7603 - a
ccuracy: 0.6619 - val_loss: 0.6883 - val_accuracy: 0.6898
Epoch 35/80
329/329 [==============================] - 60s 184ms/step - loss: 0.7617 - a
ccuracy: 0.6645 - val_loss: 0.6872 - val_accuracy: 0.6898
Epoch 36/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7564 - a
ccuracy: 0.6680 - val_loss: 0.6864 - val_accuracy: 0.6951
Epoch 37/80
```

```
329/329 [==============================] - 61s 184ms/step - loss: 0.7605 - a
ccuracy: 0.6647 - val_loss: 0.6881 - val_accuracy: 0.6920
Epoch 38/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7560 - a
ccuracy: 0.6691 - val_loss: 0.6842 - val_accuracy: 0.6938
Epoch 39/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7549 - a
ccuracy: 0.6688 - val_loss: 0.6824 - val_accuracy: 0.6920
Epoch 40/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7461 - a
ccuracy: 0.6697 - val_loss: 0.6817 - val_accuracy: 0.6942
Epoch 41/80
329/329 [==============================] - 60s 182ms/step - loss: 0.7571 - a
ccuracy: 0.6699 - val_loss: 0.6803 - val_accuracy: 0.6929
Epoch 42/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7515 - a
ccuracy: 0.6698 - val_loss: 0.6808 - val_accuracy: 0.6942
Epoch 43/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7481 - a
ccuracy: 0.6696 - val_loss: 0.6800 - val_accuracy: 0.6911
Epoch 44/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7524 - a
ccuracy: 0.6659 - val_loss: 0.6784 - val_accuracy: 0.6942
Epoch 45/80
329/329 [==============================] - 59s 179ms/step - loss: 0.7455 - a
ccuracy: 0.6678 - val_loss: 0.6802 - val_accuracy: 0.6916
Epoch 46/80
329/329 [==============================] - 59s 180ms/step - loss: 0.7462 - a
ccuracy: 0.6706 - val_loss: 0.6801 - val_accuracy: 0.6924
Epoch 47/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7423 - a
ccuracy: 0.6724 - val_loss: 0.6778 - val_accuracy: 0.6933
Epoch 48/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7449 - a
ccuracy: 0.6716 - val_loss: 0.6793 - val_accuracy: 0.6933
Epoch 49/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7469 - a
ccuracy: 0.6712 - val_loss: 0.6767 - val_accuracy: 0.6951
Epoch 50/80
329/329 [==============================] - 60s 184ms/step - loss: 0.7450 - a
ccuracy: 0.6673 - val_loss: 0.6740 - val_accuracy: 0.6991
Epoch 51/80
329/329 [==============================] - 61s 185ms/step - loss: 0.7447 - a
ccuracy: 0.6663 - val_loss: 0.6751 - val_accuracy: 0.6942
Epoch 52/80
329/329 [==============================] - 60s 184ms/step - loss: 0.7434 - a
ccuracy: 0.6723 - val_loss: 0.6757 - val_accuracy: 0.6956
Epoch 53/80
329/329 [==============================] - 61s 184ms/step - loss: 0.7439 - a
ccuracy: 0.6753 - val_loss: 0.6743 - val_accuracy: 0.6956
Epoch 54/80
329/329 [==============================] - 60s 183ms/step - loss: 0.7377 - a
ccuracy: 0.6696 - val_loss: 0.6755 - val_accuracy: 0.6951
71/71 [==============================] - 10s 141ms/step - loss: 0.7105 - acc
uracy: 0.6889
Test accuracy: 0.6888889074325562
```

```
71/71 [==============================] - 11s 136ms/step
Classification Report for Configuration 4:
              precision    recall  f1-score   support

    Negative       0.75      0.65      0.70       773
     Neutral       0.57      0.67      0.62       741
    Positive       0.77      0.74      0.76       736

    accuracy                           0.69      2250
   macro avg       0.70      0.69      0.69      2250
weighted avg       0.70      0.69      0.69      2250
```

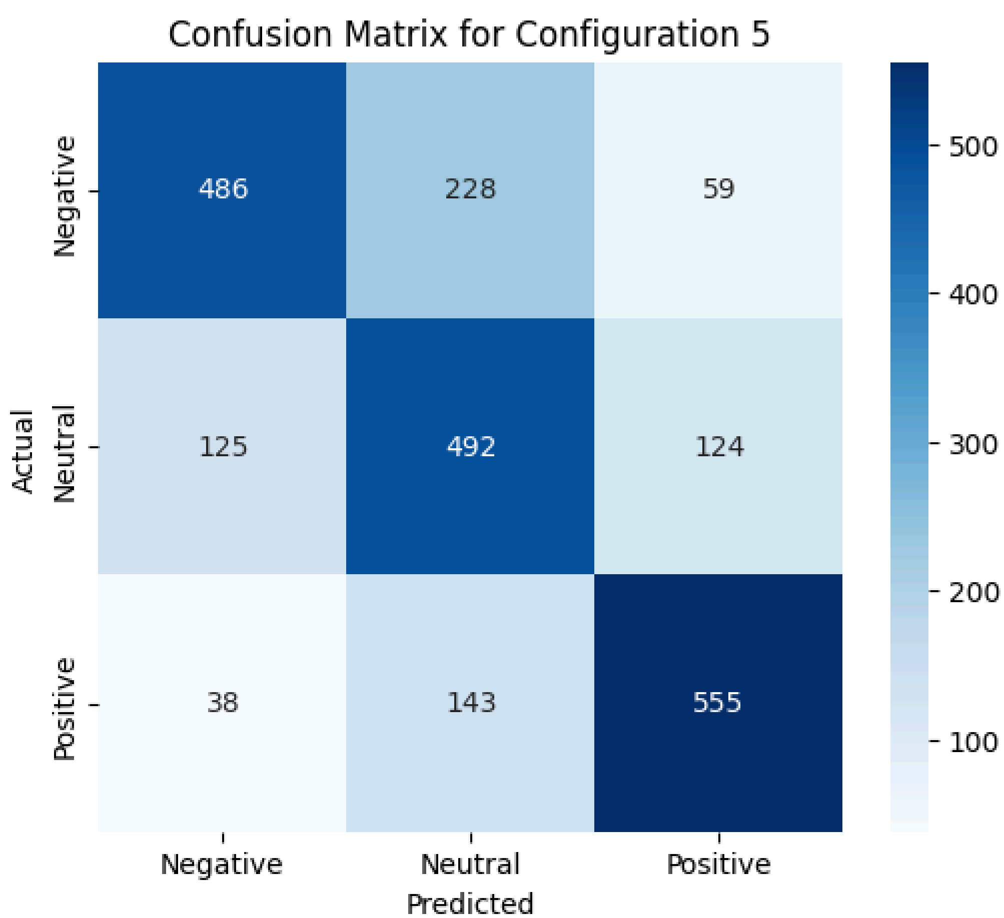


```
New best model found!
Training configuration 5/6: {'optimizer': 'adam', 'learning_rate': 1e-05, 'b
atch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_bat
chnorm': False}
Epoch 1/80
657/657 [==============================] - 74s 102ms/step - loss: 1.1318 - a
ccuracy: 0.3467 - val_loss: 1.0639 - val_accuracy: 0.4324
Epoch 2/80
657/657 [==============================] - 65s 99ms/step - loss: 1.0935 - ac
curacy: 0.3970 - val_loss: 1.0220 - val_accuracy: 0.5209
Epoch 3/80
657/657 [==============================] - 64s 98ms/step - loss: 1.0479 - ac
curacy: 0.4506 - val_loss: 0.9673 - val_accuracy: 0.6080
Epoch 4/80
657/657 [==============================] - 65s 98ms/step - loss: 1.0032 - ac
curacy: 0.4947 - val_loss: 0.9150 - val_accuracy: 0.6133
Epoch 5/80
657/657 [==============================] - 65s 99ms/step - loss: 0.9579 - ac
curacy: 0.5386 - val_loss: 0.8674 - val_accuracy: 0.6316
Epoch 6/80
657/657 [==============================] - 65s 98ms/step - loss: 0.9182 - ac
curacy: 0.5549 - val_loss: 0.8318 - val_accuracy: 0.6320
Epoch 7/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8878 - ac
curacy: 0.5826 - val_loss: 0.8031 - val_accuracy: 0.6404
Epoch 8/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8721 - ac
curacy: 0.5877 - val_loss: 0.7815 - val_accuracy: 0.6516
Epoch 9/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8611 - ac
curacy: 0.5963 - val_loss: 0.7684 - val_accuracy: 0.6520
Epoch 10/80
657/657 [==============================] - 65s 98ms/step - loss: 0.8403 - ac
curacy: 0.6081 - val_loss: 0.7611 - val_accuracy: 0.6551
Epoch 11/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8322 - ac
curacy: 0.6164 - val_loss: 0.7481 - val_accuracy: 0.6622
Epoch 12/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8231 - ac
curacy: 0.6254 - val_loss: 0.7434 - val_accuracy: 0.6698
Epoch 13/80
657/657 [==============================] - 64s 98ms/step - loss: 0.8150 - ac
curacy: 0.6304 - val_loss: 0.7380 - val_accuracy: 0.6676
Epoch 14/80
657/657 [==============================] - 65s 98ms/step - loss: 0.8141 - ac
curacy: 0.6256 - val_loss: 0.7280 - val_accuracy: 0.6702
Epoch 15/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8037 - ac
curacy: 0.6350 - val_loss: 0.7249 - val_accuracy: 0.6733
Epoch 16/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8022 - ac
curacy: 0.6378 - val_loss: 0.7227 - val_accuracy: 0.6742
Epoch 17/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7997 - ac
curacy: 0.6364 - val_loss: 0.7159 - val_accuracy: 0.6711
Epoch 18/80
```

```
657/657 [==============================] - 64s 98ms/step - loss: 0.7959 - ac
curacy: 0.6381 - val_loss: 0.7161 - val_accuracy: 0.6782
Epoch 19/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7888 - ac
curacy: 0.6425 - val_loss: 0.7151 - val_accuracy: 0.6791
Epoch 20/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7906 - ac
curacy: 0.6433 - val_loss: 0.7159 - val_accuracy: 0.6791
Epoch 21/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7864 - ac
curacy: 0.6511 - val_loss: 0.7079 - val_accuracy: 0.6804
Epoch 22/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7811 - ac
curacy: 0.6488 - val_loss: 0.7048 - val_accuracy: 0.6827
Epoch 23/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7801 - ac
curacy: 0.6495 - val_loss: 0.7050 - val_accuracy: 0.6831
Epoch 24/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7820 - ac
curacy: 0.6457 - val_loss: 0.7057 - val_accuracy: 0.6813
Epoch 25/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7727 - ac
curacy: 0.6576 - val_loss: 0.6994 - val_accuracy: 0.6884
Epoch 26/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7718 - ac
curacy: 0.6595 - val_loss: 0.6986 - val_accuracy: 0.6876
Epoch 27/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7693 - ac
curacy: 0.6612 - val_loss: 0.6952 - val_accuracy: 0.6920
Epoch 28/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7725 - ac
curacy: 0.6559 - val_loss: 0.6972 - val_accuracy: 0.6902
Epoch 29/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7706 - ac
curacy: 0.6599 - val_loss: 0.6931 - val_accuracy: 0.6898
Epoch 30/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7688 - ac
curacy: 0.6572 - val_loss: 0.6993 - val_accuracy: 0.6880
Epoch 31/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7633 - ac
curacy: 0.6553 - val_loss: 0.6981 - val_accuracy: 0.6893
Epoch 32/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7592 - ac
curacy: 0.6647 - val_loss: 0.6918 - val_accuracy: 0.6938
Epoch 33/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7654 - ac
curacy: 0.6605 - val_loss: 0.6975 - val_accuracy: 0.6902
Epoch 34/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7605 - ac
curacy: 0.6647 - val_loss: 0.6962 - val_accuracy: 0.6880
Epoch 35/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7566 - ac
curacy: 0.6619 - val_loss: 0.6916 - val_accuracy: 0.6960
Epoch 36/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7635 - ac
curacy: 0.6572 - val_loss: 0.6916 - val_accuracy: 0.6982
```

```
Epoch 37/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7562 - ac
curacy: 0.6639 - val_loss: 0.6864 - val_accuracy: 0.6964
Epoch 38/80
657/657 [==============================] - 64s 97ms/step - loss: 0.7576 - ac
curacy: 0.6636 - val_loss: 0.6921 - val_accuracy: 0.6938
Epoch 39/80
657/657 [==============================] - 64s 97ms/step - loss: 0.7584 - ac
curacy: 0.6630 - val_loss: 0.6910 - val_accuracy: 0.6973
Epoch 40/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7663 - ac
curacy: 0.6595 - val_loss: 0.6853 - val_accuracy: 0.6942
Epoch 41/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7540 - ac
curacy: 0.6658 - val_loss: 0.6896 - val_accuracy: 0.6969
Epoch 42/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7518 - ac
curacy: 0.6648 - val_loss: 0.6859 - val_accuracy: 0.6982
Epoch 43/80
657/657 [==============================] - 74s 113ms/step - loss: 0.7504 - a
ccuracy: 0.6707 - val_loss: 0.6883 - val_accuracy: 0.6933
Epoch 44/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7548 - ac
curacy: 0.6627 - val_loss: 0.6840 - val_accuracy: 0.6947
Epoch 45/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7465 - ac
curacy: 0.6692 - val_loss: 0.6856 - val_accuracy: 0.7000
Epoch 46/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7535 - ac
curacy: 0.6687 - val_loss: 0.6875 - val_accuracy: 0.6956
Epoch 47/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7545 - ac
curacy: 0.6687 - val_loss: 0.6837 - val_accuracy: 0.6978
Epoch 48/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7459 - ac
curacy: 0.6710 - val_loss: 0.6828 - val_accuracy: 0.6978
Epoch 49/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7463 - ac
curacy: 0.6735 - val_loss: 0.6828 - val_accuracy: 0.6978
Epoch 50/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7504 - ac
curacy: 0.6670 - val_loss: 0.6823 - val_accuracy: 0.6964
Epoch 51/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7529 - ac
curacy: 0.6614 - val_loss: 0.6821 - val_accuracy: 0.6982
Epoch 52/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7451 - ac
curacy: 0.6706 - val_loss: 0.6783 - val_accuracy: 0.6982
Epoch 53/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7423 - ac
curacy: 0.6695 - val_loss: 0.6789 - val_accuracy: 0.6973
Epoch 54/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7481 - ac
curacy: 0.6723 - val_loss: 0.6795 - val_accuracy: 0.6982
Epoch 55/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7459 - ac
```

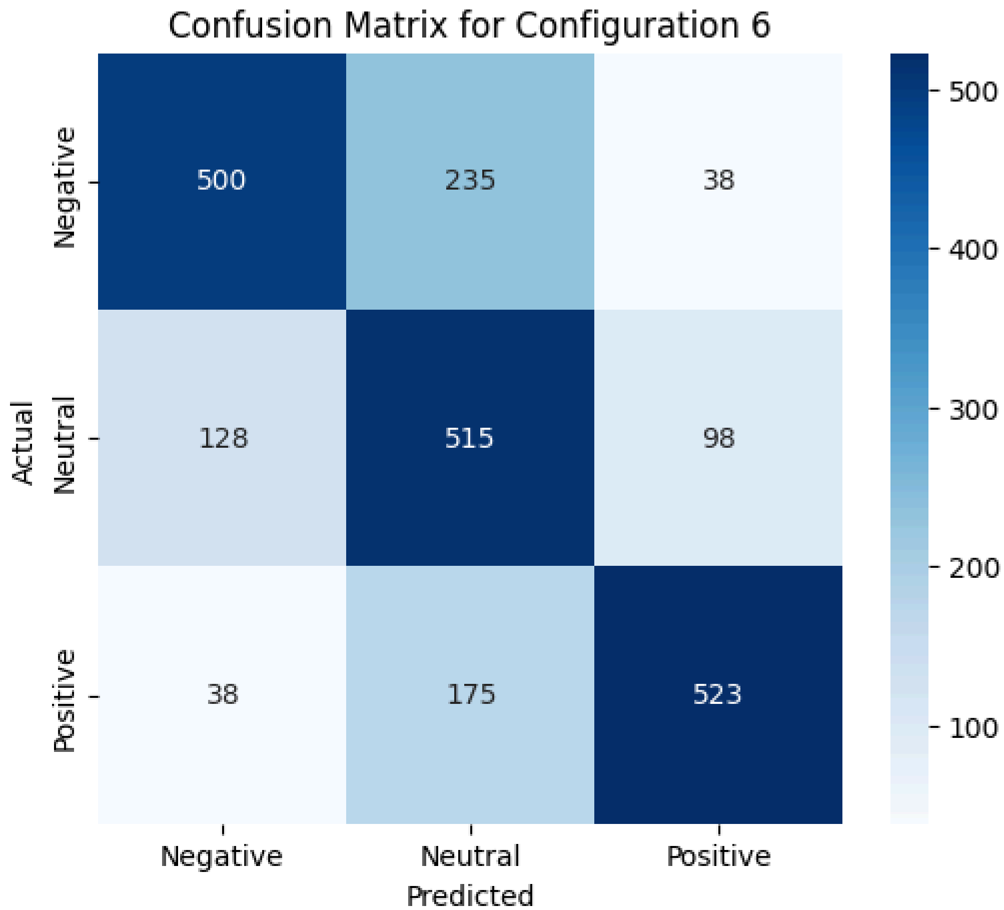
```
curacy: 0.6697 - val_loss: 0.6771 - val_accuracy: 0.6982
Epoch 56/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7503 - ac
curacy: 0.6716 - val_loss: 0.6792 - val_accuracy: 0.7013
Epoch 57/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7459 - ac
curacy: 0.6691 - val_loss: 0.6810 - val_accuracy: 0.6996
Epoch 58/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7434 - ac
curacy: 0.6745 - val_loss: 0.6806 - val_accuracy: 0.6991
Epoch 59/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7470 - ac
curacy: 0.6707 - val_loss: 0.6778 - val_accuracy: 0.7013
141/141 [==============================] - 11s 76ms/step - loss: 0.7135 - ac
curacy: 0.6813
Test accuracy: 0.6813333630561829
141/141 [==============================] - 12s 74ms/step
Classification Report for Configuration 5:
              precision    recall  f1-score   support

    Negative       0.75      0.63      0.68       773
     Neutral       0.57      0.66      0.61       741
    Positive       0.75      0.75      0.75       736

    accuracy                           0.68      2250
   macro avg       0.69      0.68      0.68      2250
weighted avg       0.69      0.68      0.68      2250
```



```
Training configuration 6/6: {'optimizer': 'adam', 'learning_rate': 3e-05, 'b
atch_size': 16, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_bat
chnorm': False}
Epoch 1/80
657/657 [==============================] - 74s 102ms/step - loss: 1.1243 - a
ccuracy: 0.3430 - val_loss: 1.0926 - val_accuracy: 0.3338
Epoch 2/80
657/657 [==============================] - 65s 98ms/step - loss: 1.0952 - ac
curacy: 0.3697 - val_loss: 1.0708 - val_accuracy: 0.4542
Epoch 3/80
657/657 [==============================] - 65s 99ms/step - loss: 1.0765 - ac
curacy: 0.4084 - val_loss: 1.0314 - val_accuracy: 0.5991
Epoch 4/80
657/657 [==============================] - 65s 99ms/step - loss: 1.0347 - ac
curacy: 0.4702 - val_loss: 0.9296 - val_accuracy: 0.6027
Epoch 5/80
657/657 [==============================] - 65s 99ms/step - loss: 0.9706 - ac
curacy: 0.5165 - val_loss: 0.8441 - val_accuracy: 0.6142
Epoch 6/80
657/657 [==============================] - 65s 99ms/step - loss: 0.9237 - ac
curacy: 0.5471 - val_loss: 0.8108 - val_accuracy: 0.6231
Epoch 7/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8954 - ac
curacy: 0.5716 - val_loss: 0.7948 - val_accuracy: 0.6347
Epoch 8/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8795 - ac
curacy: 0.5766 - val_loss: 0.7734 - val_accuracy: 0.6462
Epoch 9/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8602 - ac
curacy: 0.5939 - val_loss: 0.7602 - val_accuracy: 0.6591
Epoch 10/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8543 - ac
curacy: 0.5951 - val_loss: 0.7535 - val_accuracy: 0.6627
Epoch 11/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8321 - ac
curacy: 0.6167 - val_loss: 0.7391 - val_accuracy: 0.6631
Epoch 12/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8308 - ac
curacy: 0.6216 - val_loss: 0.7370 - val_accuracy: 0.6667
Epoch 13/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8199 - ac
curacy: 0.6295 - val_loss: 0.7322 - val_accuracy: 0.6707
Epoch 14/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8203 - ac
curacy: 0.6317 - val_loss: 0.7274 - val_accuracy: 0.6756
Epoch 15/80
657/657 [==============================] - 65s 98ms/step - loss: 0.8101 - ac
curacy: 0.6375 - val_loss: 0.7192 - val_accuracy: 0.6849
Epoch 16/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8003 - ac
curacy: 0.6398 - val_loss: 0.7176 - val_accuracy: 0.6840
Epoch 17/80
657/657 [==============================] - 65s 99ms/step - loss: 0.8015 - ac
curacy: 0.6411 - val_loss: 0.7122 - val_accuracy: 0.6844
Epoch 18/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7923 - ac
```

```
curacy: 0.6436 - val_loss: 0.7108 - val_accuracy: 0.6751
Epoch 19/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7823 - ac
curacy: 0.6462 - val_loss: 0.7042 - val_accuracy: 0.6800
Epoch 20/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7857 - ac
curacy: 0.6553 - val_loss: 0.7026 - val_accuracy: 0.6858
Epoch 21/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7746 - ac
curacy: 0.6583 - val_loss: 0.6977 - val_accuracy: 0.6849
Epoch 22/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7727 - ac
curacy: 0.6545 - val_loss: 0.6994 - val_accuracy: 0.6853
Epoch 23/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7706 - ac
curacy: 0.6607 - val_loss: 0.6969 - val_accuracy: 0.6880
Epoch 24/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7634 - ac
curacy: 0.6663 - val_loss: 0.6920 - val_accuracy: 0.6907
Epoch 25/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7592 - ac
curacy: 0.6664 - val_loss: 0.6915 - val_accuracy: 0.6929
Epoch 26/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7608 - ac
curacy: 0.6643 - val_loss: 0.6924 - val_accuracy: 0.6911
Epoch 27/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7585 - ac
curacy: 0.6687 - val_loss: 0.6864 - val_accuracy: 0.6929
Epoch 28/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7573 - ac
curacy: 0.6690 - val_loss: 0.6886 - val_accuracy: 0.6920
Epoch 29/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7534 - ac
curacy: 0.6694 - val_loss: 0.6878 - val_accuracy: 0.6929
Epoch 30/80
657/657 [==============================] - 65s 99ms/step - loss: 0.7467 - ac
curacy: 0.6730 - val_loss: 0.6766 - val_accuracy: 0.6938
Epoch 31/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7546 - ac
curacy: 0.6709 - val_loss: 0.6840 - val_accuracy: 0.6942
Epoch 32/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7483 - ac
curacy: 0.6677 - val_loss: 0.6801 - val_accuracy: 0.6920
Epoch 33/80
657/657 [==============================] - 65s 98ms/step - loss: 0.7449 - ac
curacy: 0.6708 - val_loss: 0.6787 - val_accuracy: 0.6973
Epoch 34/80
657/657 [==============================] - 64s 98ms/step - loss: 0.7467 - ac
curacy: 0.6694 - val_loss: 0.6824 - val_accuracy: 0.6884
141/141 [==============================] - 11s 76ms/step - loss: 0.7176 - ac
curacy: 0.6836
Test accuracy: 0.683555543422699
141/141 [==============================] - 69s 74ms/step
Classification Report for Configuration 6:
              precision    recall  f1-score   support
```

```
    Negative       0.75      0.65      0.69       773
     Neutral       0.56      0.70      0.62       741
    Positive       0.79      0.71      0.75       736

    accuracy                           0.68      2250
   macro avg       0.70      0.68      0.69      2250
weighted avg       0.70      0.68      0.69      2250
```



6.2.2 Non-Batch-Normalized Models: Examples of Mis-classified Points

In [ ]:
```
# After the training loop
print(f"\nBest Model Configuration: {best_config}")
print(f"Best Model Test Accuracy: {best_accuracy}")

# Use the best model's predictions
y_pred = best_y_pred
y_true = best_y_true

# Identify misclassified examples
misclassified_indices = np.where(y_pred != y_true)[0]

# Decode test texts
def decode_texts(input_ids):
    return [tokenizer.decode(ids, skip_special_tokens=True) for ids in input

test_texts = decode_texts(test_input_ids)
```

```
# Extract false positives and false negatives
false_positives = []
false_negatives = []

# Mapping of label indices to label names
label_map = {0: 'Negative', 1: 'Neutral', 2: 'Positive'}

# Loop through misclassified examples to separate false positives and false
for idx in misclassified_indices:
    true_label = y_true[idx]
    predicted_label = y_pred[idx]
    text = test_texts[idx]

    if predicted_label == 2 and true_label != 2:
        # Model predicted Positive, but true label is Negative or Neutral
        false_positives.append((text, label_map[true_label], label_map[predi
    elif predicted_label != 2 and true_label == 2:
        # Model predicted Negative or Neutral, but true label is Positive
        false_negatives.append((text, label_map[true_label], label_map[predi

# Display a few examples of false positives
print("\nExamples of False Positives:")
for i in range(min(3, len(false_positives))):
    text, true_label, predicted_label = false_positives[i]
    print(f"\nText: {text}")
    print(f"True Label: {true_label}")
    print(f"Predicted Label: {predicted_label}")

# Display a few examples of false negatives
print("\nExamples of False Negatives:")
for i in range(min(3, len(false_negatives))):
    text, true_label, predicted_label = false_negatives[i]
    print(f"\nText: {text}")
    print(f"True Label: {true_label}")
    print(f"Predicted Label: {predicted_label}")
```

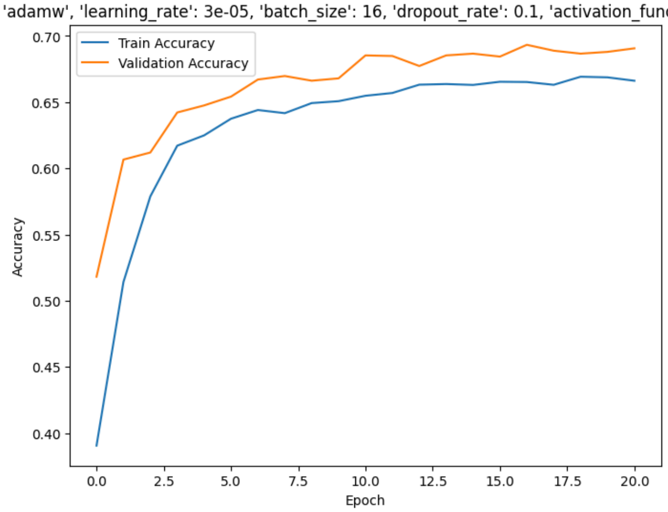
```
Best Model Configuration: {'optimizer': 'adam', 'learning_rate': 3e-05, 'bat
ch_size': 32, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_batch
norm': False}
Best Model Test Accuracy: 0.6888889074325562

Examples of False Positives:

Text: this is quite good ive taken to starting my mornings with a nice hot m
ug
True Label: Neutral
Predicted Label: Positive

Text: i love gloria jeans cinnamon nut strudel coffee and decided to try thi
s the flavoring is a little over powering for me if youve tried this flavor
before and like it you wont be disappointed with it in the kcups packaging
True Label: Neutral
Predicted Label: Positive

Text: we love clif kid z bars i decided to try these based on all of the gre
at reviews we bought the strawberry and no one in the house likes them my ki
ds are picky so i always try the treats just to see if they are even worth t
rying i eat healthy and dont mind most healthy snacks but even i did not car
e for these i am going to try another flavor but the strawberry just didnt c
ut it here
True Label: Negative
Predicted Label: Positive

Examples of False Negatives:

Text: i really only needed the funnelpitcher but it was easy and neat to mak
e funnel cakes i had been using a kitchen funnel which made a big mess no ma
tter how careful i was i didnt use the ring because i made the funnel cakes
in a small deep fryer the mix was fine but i prefer my own recipe if i could
have just bought the funnelpitcher that would have been fine and saved me so
me money but im not disappointed
True Label: Positive
Predicted Label: Neutral

Text: i was very impressed with the flavor growing up on mrs butterworths i
developed a taste for commercial syrup and had a hard time adjusting to the
real thing but when i opened up the jug of coombs it was delish i would high
ly recommend this product
True Label: Positive
Predicted Label: Neutral

Text: i got a larger one than my puppy probably required and she loves dragg
ing it around playing tug of war and just chewing on it unfortunately it doe
s shed i dont know if there are rope toys out there that are any sturdier bu
t this one does leave little threads behind otherwise great toy
True Label: Positive
Predicted Label: Neutral
```

6.2.3 Non-Batch-Normalized Models: Visualizations

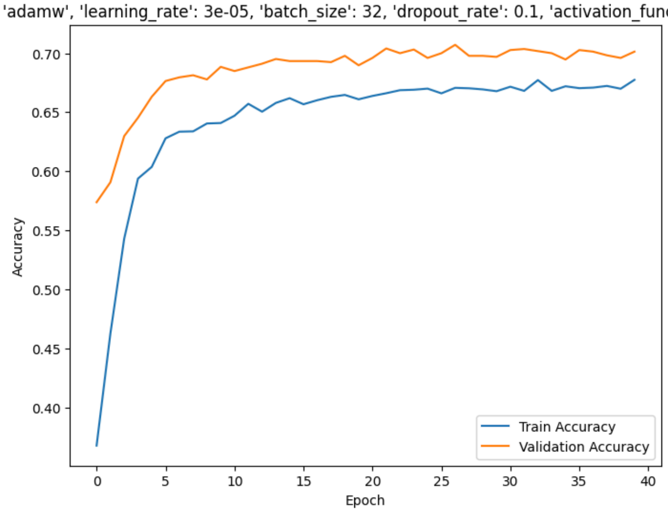
In [ ]:
```python
# Visualize the results
```

```python
# Plot training and validation accuracy for each configuration
for i, (history, config) in enumerate(histories):
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title(f"Configuration {i+1}: {config}")
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
```

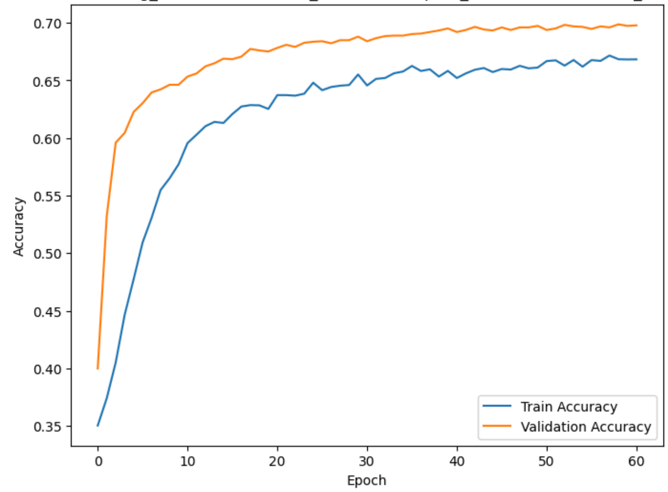
Configuration 1: {'optimizer': 'adamw', 'learning_rate': 3e-05, 'batch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': False}



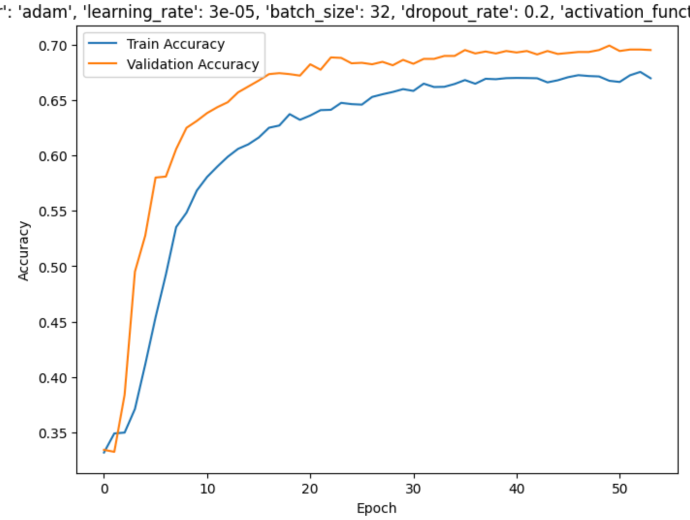
Configuration 2: {'optimizer': 'adamw', 'learning_rate': 3e-05, 'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': False}



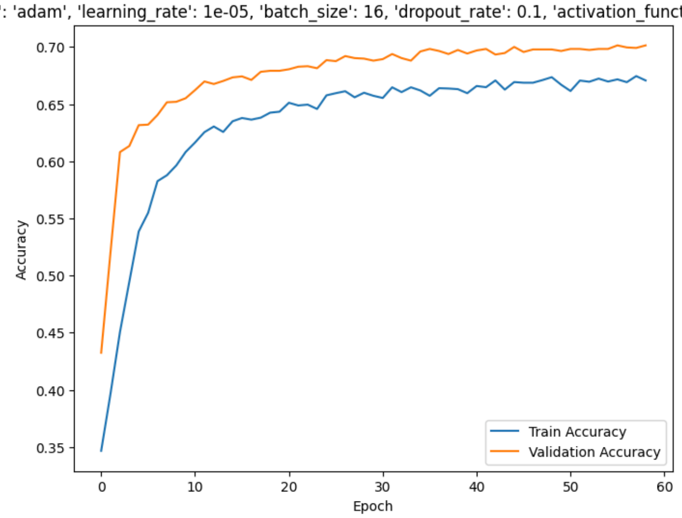
Configuration 3: {'optimizer': 'adamw', 'learning_rate': 1e-05, 'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': False}



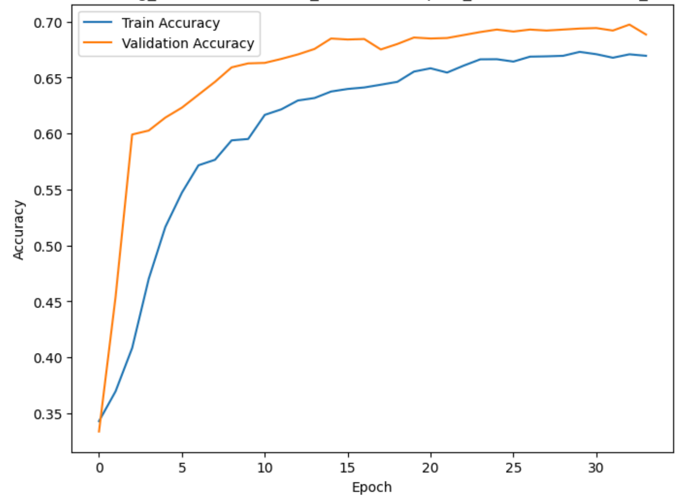
Configuration 4: {'optimizer': 'adam', 'learning_rate': 3e-05, 'batch_size': 32, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_batchnorm': False}



Configuration 5: {'optimizer': 'adam', 'learning_rate': 1e-05, 'batch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': False}

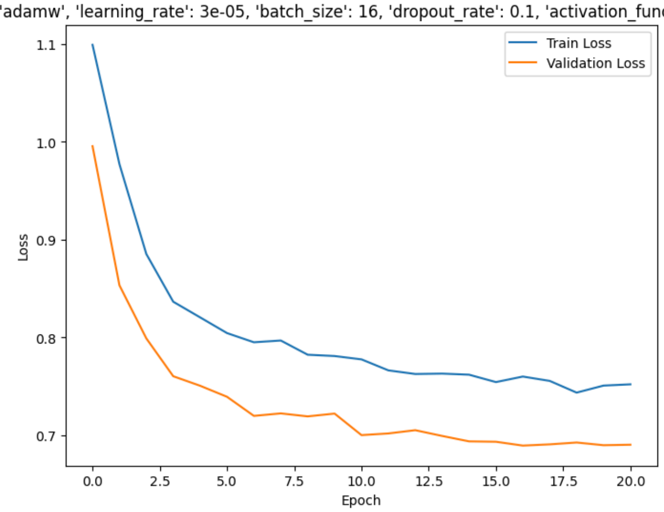


Configuration 6: {'optimizer': 'adam', 'learning_rate': 3e-05, 'batch_size': 16, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_batchnorm': False}

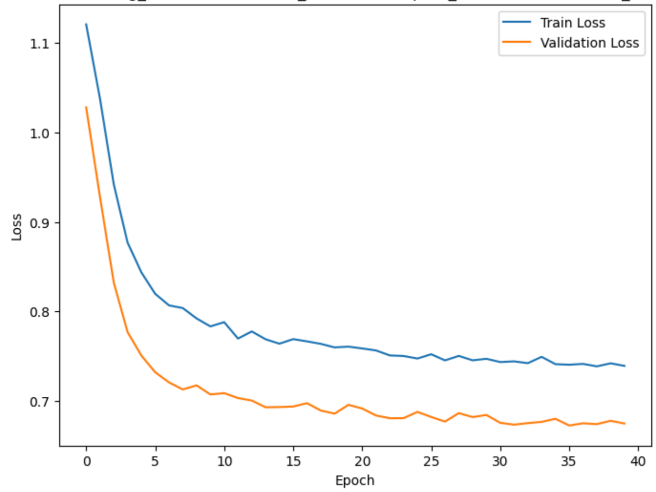



```python
# Plot training and validation loss for each configuration
for i, (history, config) in enumerate(histories):
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title(f"Configuration {i+1}: {config}")
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

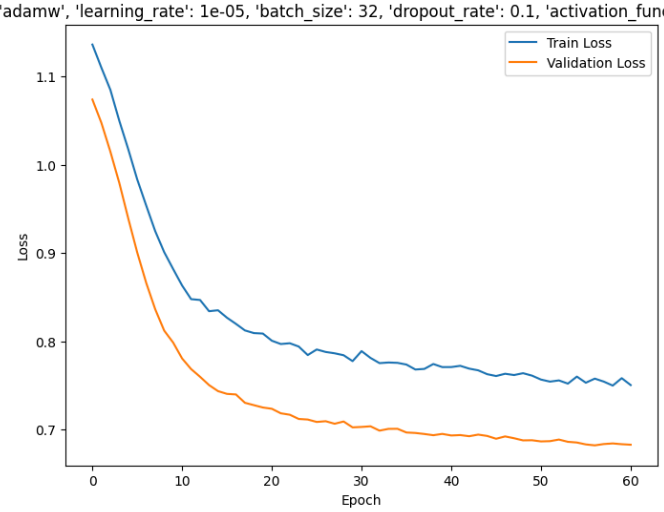
Configuration 1: {'optimizer': 'adamw', 'learning_rate': 3e-05, 'batch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': False}



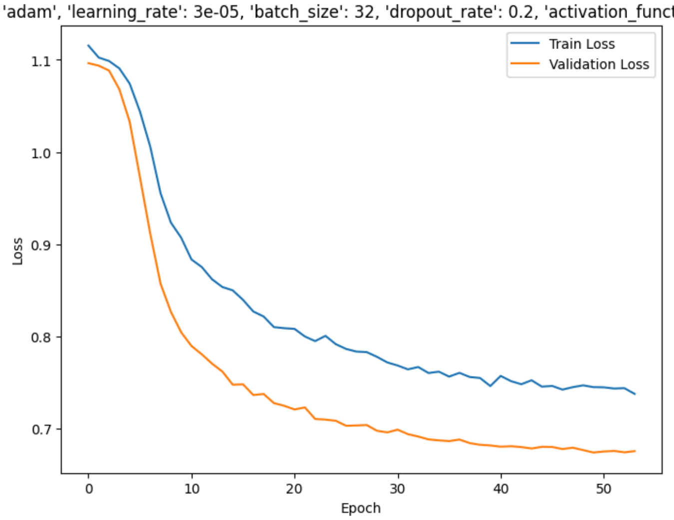
Configuration 2: {'optimizer': 'adamw', 'learning_rate': 3e-05, 'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': False}



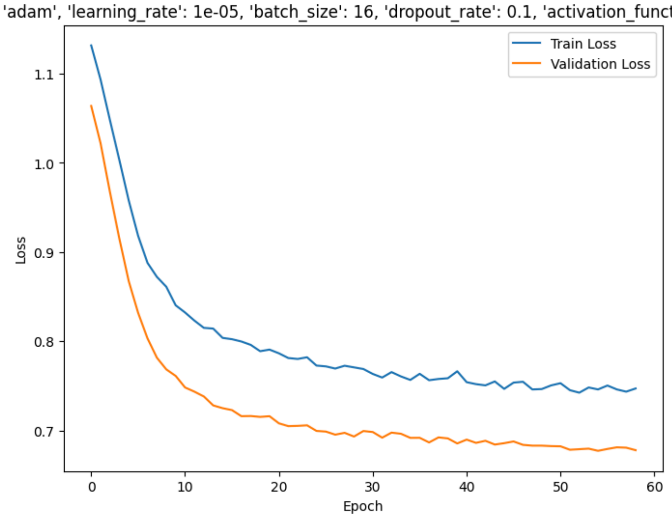
Configuration 3: {'optimizer': 'adamw', 'learning_rate': 1e-05, 'batch_size': 32, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': False}



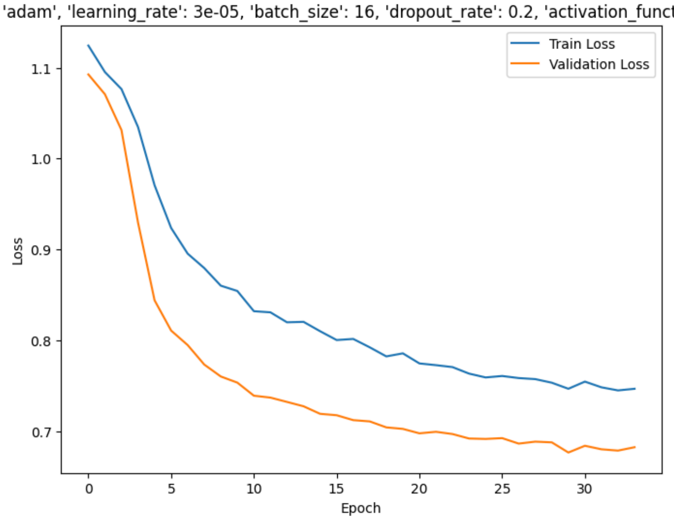
Configuration 4: {'optimizer': 'adam', 'learning_rate': 3e-05, 'batch_size': 32, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_batchnorm': False}



Configuration 5: {'optimizer': 'adam', 'learning_rate': 1e-05, 'batch_size': 16, 'dropout_rate': 0.1, 'activation_function': 'tanh', 'use_batchnorm': False}



Configuration 6: {'optimizer': 'adam', 'learning_rate': 3e-05, 'batch_size': 16, 'dropout_rate': 0.2, 'activation_function': 'relu', 'use_batchnorm': False}



6.2.4 Non-Batch-Normalized Models: Model Results

In [ ]:
```
# Create a DataFrame of the results
results_df = pd.DataFrame([{
    'optimizer': res['config']['optimizer'],
    'learning_rate': res['config']['learning_rate'],
    'batch_size': res['config']['batch_size'],
    'dropout_rate': res['config']['dropout_rate'],
    'activation_function': res['config']['activation_function'],
    'use_batchnorm': res['config']['use_batchnorm'],
    'accuracy': res['accuracy'],
    'loss': res['loss']
} for res in results])

print(results_df)
```

```
   optimizer  learning_rate  batch_size  dropout_rate activation_function  \
0      adamw        0.00003          16           0.1                tanh
1      adamw        0.00003          16           0.1                tanh
2      adamw        0.00003          32           0.1                tanh
3      adamw        0.00003          32           0.1                tanh
4      adamw        0.00001          32           0.1                tanh
5      adamw        0.00001          32           0.1                tanh
6       adam        0.00003          32           0.2                relu
7       adam        0.00003          32           0.2                relu
8       adam        0.00001          16           0.1                tanh
9       adam        0.00001          16           0.1                tanh
10      adam        0.00003          16           0.2                relu
11      adam        0.00003          16           0.2                relu

    use_batchnorm  accuracy      loss
0           False  0.676000  0.725449
1           False  0.676000  0.725449
2           False  0.684444  0.709029
3           False  0.684444  0.709029
4           False  0.680444  0.719499
5           False  0.680444  0.719499
6           False  0.688889  0.710524
7           False  0.688889  0.710524
8           False  0.681333  0.713452
9           False  0.681333  0.713452
10          False  0.683556  0.717630
11          False  0.683556  0.717630
```

6.3 Frozen Encoder Layers

6.3.1 Frozen Encoder Layers: Training the Model

In [ ]:

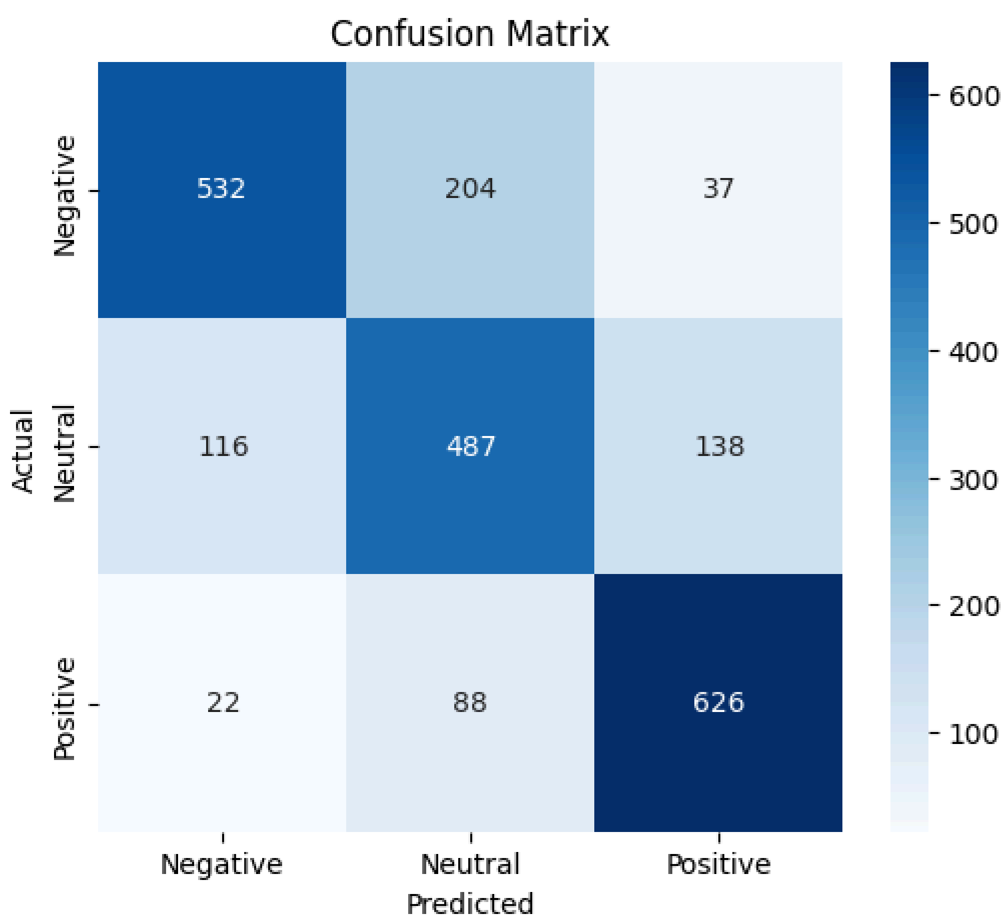
```
from transformers import TFRobertaForSequenceClassification

# Load the pre-trained distilroberta model for sequence classification
model = TFRobertaForSequenceClassification.from_pretrained('distilroberta-ba

# Freeze the first 3 encoder layers
for layer in model.roberta.encoder.layer[:3]:
    layer.trainable = False

# Define the optimizer
optimizer = tf.keras.optimizers.Adam(learning_rate=3e-5)

# Define the loss with from_logits=True
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# Compile the model
model.compile(
    optimizer=optimizer,
    loss=loss,
    metrics=['accuracy']
)

# Set batch size and create datasets
```

```
batch_size = 16
train_dataset = create_tf_dataset(train_input_ids, train_attention_mask, tra
val_dataset = create_tf_dataset(val_input_ids, val_attention_mask, val_label
test_dataset = create_tf_dataset(test_input_ids, test_attention_mask, test_l

# Set up callbacks
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patien

# Fit the model using the validation set
history = model.fit(
    train_dataset,
    epochs=10,
    validation_data=val_dataset,
    callbacks=[early_stopping],
    verbose=1
)

# Record the history and config
histories.append((history, {'learning_rate': 3e-5, 'batch_size': 16}))

# Evaluate the final model on the test set
loss, accuracy = model.evaluate(test_dataset)
print(f"Test accuracy: {accuracy}")

# Store results
results.append({
    'learning_rate': 3e-5,
    'batch_size': 16,
    'accuracy': accuracy,
    'loss': loss,
})

# Generate predictions for the test set
y_pred_probs = model.predict(test_dataset)
y_pred = np.argmax(y_pred_probs.logits, axis=1)

# Flatten test labels
y_true = test_labels

# Compute confusion matrix
cm = confusion_matrix(y_true, y_pred)

# Classification report
report = classification_report(y_true, y_pred, target_names=['Negative', 'Ne
print(f"Classification Report:\n{report}")

# Plot confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Negative',
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title(f'Confusion Matrix')
plt.show()

# Store results
results.append({
```

```python
    'learning_rate': 3e-5,
    'batch_size': 16,
    'accuracy': accuracy,
    'loss': loss,
    'confusion_matrix': cm,
    'classification_report': report
})

# If current model is better, save it

# Clear previous best model from memory if exists

best_accuracy = accuracy
best_model = model
best_config = {'learning_rate': 3e-5, 'batch_size': 16}
best_y_pred = y_pred
best_y_true = y_true
# Also store confusion matrix and classification report
best_cm = cm
best_report = report
```

```
All PyTorch model weights were used when initializing TFRobertaForSequenceCl
assification.

Some weights or buffers of the TF 2.0 model TFRobertaForSequenceClassificati
on were not initialized from the PyTorch model and are newly initialized:
['classifier.dense.weight', 'classifier.dense.bias', 'classifier.out_proj.we
ight', 'classifier.out_proj.bias']
You should probably TRAIN this model on a down-stream task to be able to use
it for predictions and inference.
```

```
Epoch 1/10
657/657 [==============================] - 156s 217ms/step - loss: 0.7083 -
accuracy: 0.6799 - val_loss: 0.5798 - val_accuracy: 0.7462
Epoch 2/10
657/657 [==============================] - 134s 204ms/step - loss: 0.5649 -
accuracy: 0.7679 - val_loss: 0.6187 - val_accuracy: 0.7356
Epoch 3/10
657/657 [==============================] - 134s 204ms/step - loss: 0.4767 -
accuracy: 0.8105 - val_loss: 0.6324 - val_accuracy: 0.7493
Epoch 4/10
657/657 [==============================] - 134s 205ms/step - loss: 0.3889 -
accuracy: 0.8486 - val_loss: 0.6656 - val_accuracy: 0.7507
141/141 [==============================] - 10s 73ms/step - loss: 0.6086 - ac
curacy: 0.7311
Test accuracy: 0.7311111092567444
141/141 [==============================] - 12s 72ms/step
Classification Report:
              precision    recall  f1-score   support

    Negative       0.79      0.69      0.74       773
     Neutral       0.63      0.66      0.64       741
    Positive       0.78      0.85      0.81       736

    accuracy                           0.73      2250
   macro avg       0.73      0.73      0.73      2250
weighted avg       0.73      0.73      0.73      2250
```



In [ ]:
```
# Save the tokenizer (recommended)
tokenizer.save_pretrained('model_directory')

# Save the model
model.save_pretrained('model_directory')
```

6.3.2 Frozen Encoder Layers: Mis-classified Prices

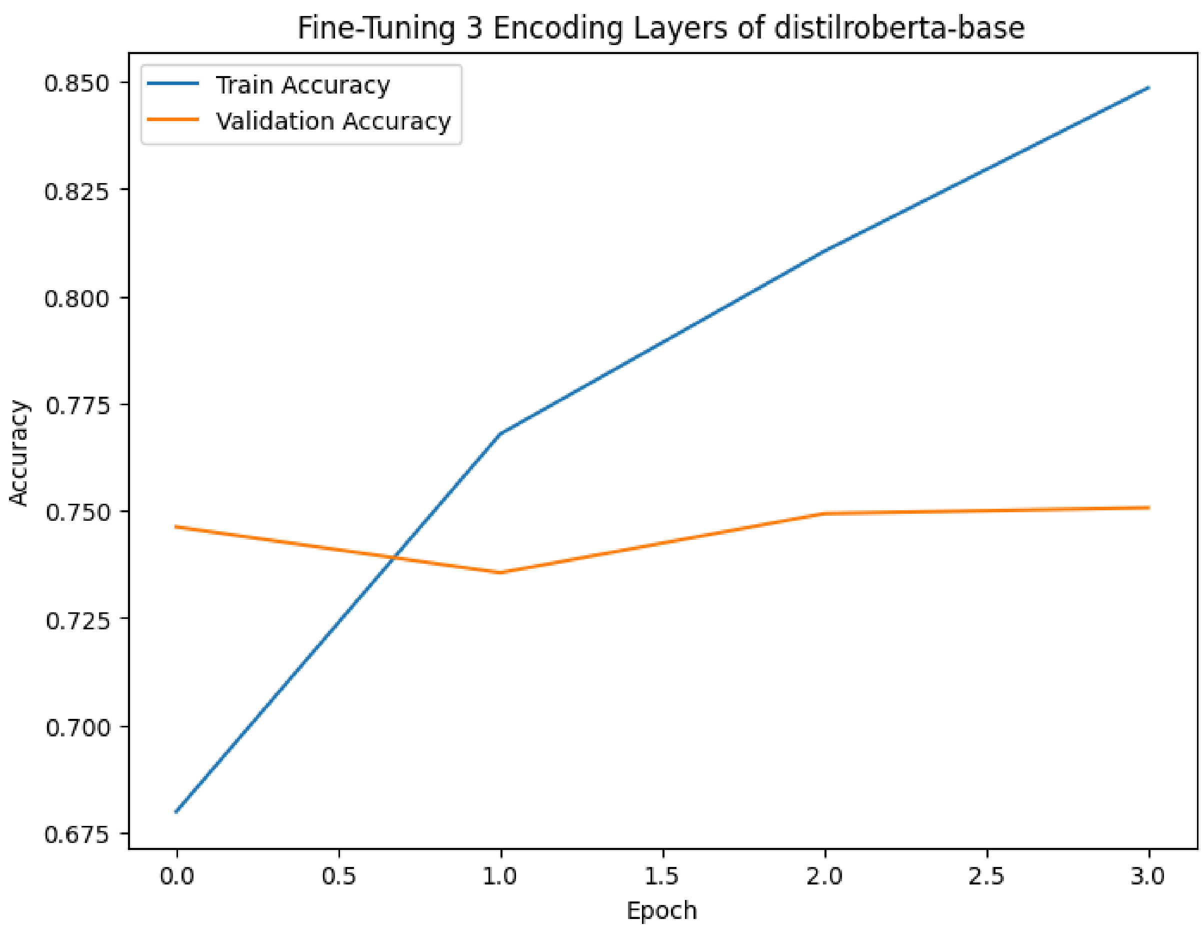
In [ ]:
```
# After the training loop
print(f"\nBest Model Configuration: {best_config}")
print(f"Best Model Test Accuracy: {best_accuracy}")

# Use the best model's predictions
y_pred = best_y_pred
y_true = best_y_true

# Identify misclassified examples
misclassified_indices = np.where(y_pred != y_true)[0]

# Decode test texts
def decode_texts(input_ids):
    return [tokenizer.decode(ids, skip_special_tokens=True) for ids in input

test_texts = decode_texts(test_input_ids)

# Extract false positives and false negatives
false_positives = []
false_negatives = []

# Mapping of label indices to label names
label_map = {0: 'Negative', 1: 'Neutral', 2: 'Positive'}

# Loop through misclassified examples to separate false positives and false
for idx in misclassified_indices:
    true_label = y_true[idx]
    predicted_label = y_pred[idx]
    text = test_texts[idx]

    if predicted_label == 2 and true_label != 2:
        # Model predicted Positive, but true label is Negative or Neutral
        false_positives.append((text, label_map[true_label], label_map[predi
    elif predicted_label != 2 and true_label == 2:
        # Model predicted Negative or Neutral, but true label is Positive
        false_negatives.append((text, label_map[true_label], label_map[predi

# Display a few examples of false positives
print("\nExamples of False Positives:")
for i in range(min(3, len(false_positives))):
    text, true_label, predicted_label = false_positives[i]
    print(f"\nText: {text}")
    print(f"True Label: {true_label}")
    print(f"Predicted Label: {predicted_label}")

# Display a few examples of false negatives
print("\nExamples of False Negatives:")
```

```
for i in range(min(3, len(false_negatives))):
    text, true_label, predicted_label = false_negatives[i]
    print(f"\nText: {text}")
    print(f"True Label: {true_label}")
    print(f"Predicted Label: {predicted_label}")
```

```
Best Model Configuration: {'learning_rate': 3e-05, 'batch_size': 16}
Best Model Test Accuracy: 0.7311111092567444

Examples of False Positives:

Text: this is quite good ive taken to starting my mornings with a nice hot m
ug
True Label: Neutral
Predicted Label: Positive

Text: i love gloria jeans cinnamon nut strudel coffee and decided to try thi
s the flavoring is a little over powering for me if youve tried this flavor
before and like it you wont be disappointed with it in the kcups packaging
True Label: Neutral
Predicted Label: Positive

Text: im an espresso nut and have several machines that i call my friends on
e of these is a jura s avantgarde and after trying dozens and dozens of vari
ous beans to find the perfect mate intelligentsia black cat classic prevaile
d as the very clear winner there may be better beans out there but not for t
his machine ive served thousands of shots of this espresso out of the jura a
nd everyone is impressed by it actually not just impressed blown away its no
t uncommon to have neighbors coming in the back door in the morning to grab
a shot this bean produces a perfect espresso great crema thick legs smooth w
ith no bitterness and a slight nut
True Label: Neutral
Predicted Label: Positive

Examples of False Negatives:

Text: i really only needed the funnelpitcher but it was easy and neat to mak
e funnel cakes i had been using a kitchen funnel which made a big mess no ma
tter how careful i was i didnt use the ring because i made the funnel cakes
in a small deep fryer the mix was fine but i prefer my own recipe if i could
have just bought the funnelpitcher that would have been fine and saved me so
me money but im not disappointed
True Label: Positive
Predicted Label: Neutral

Text: i got a larger one than my puppy probably required and she loves dragg
ing it around playing tug of war and just chewing on it unfortunately it doe
s shed i dont know if there are rope toys out there that are any sturdier bu
t this one does leave little threads behind otherwise great toy
True Label: Positive
Predicted Label: Neutral

Text: our dogs love dingo bones but these peanut butter flavor bones are an
obsession of our smallest rat terrier hes only pounds so bones tend to last
several sittings for him give him one of these and hes occupied for as long
as it takes to finish hell still play with other bones but he loses his mind
when he sees me go for this bag i just wish they were priced as well as the
other dingo products and less difficult to find
True Label: Positive
Predicted Label: Neutral
```

6.3.3 Frozen Encoder Layers: Training vs. Validation

In [ ]:
```
# Plot training and validation accuracy for each configuration
for i, (history, config) in enumerate(histories):
  if i == 6:
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title("Fine-Tuning 3 Encoding Layers of distilroberta-base")
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
  else:
    None
```



7. Inference Pipeline

Below is the inference pipeline for our model. It is a function that takes text and returns the predicted sentiment.

In [ ]:
```
from transformers import RobertaTokenizer, TFRobertaForSequenceClassificatic

# Load the tokenizer
tokenizer = RobertaTokenizer.from_pretrained('model_directory')
```

```python
# Load the model
best_model = TFRobertaForSequenceClassification.from_pretrained('model_direc
```

In [ ]:
```python
import re

def clean_text(text):
    # Remove HTML tags
    text = re.sub(r'<.*?>', '', text)
    # Remove non-ASCII characters
    text = text.encode('ascii', 'ignore').decode('utf-8')
    # Remove URLs
    text = re.sub(r'http\S+', '', text)
    # Remove special characters and numbers
    text = re.sub(r'[^A-Za-z\s]', '', text)
    # Convert to lowercase
    text = text.lower()
    # Remove extra spaces
    text = re.sub(r'\s+', ' ', text).strip()
    return text
```

In [ ]:
```python
import tensorflow as tf

def predict_sentiment(text):
    # Clean the input text
    cleaned_text = clean_text(text)

    # Tokenize the text
    inputs = tokenizer(
        cleaned_text,
        padding='max_length',
        truncation=True,
        max_length=128,
        return_tensors='tf'
    )

    # Get model predictions
    outputs = best_model(inputs)
    logits = outputs.logits

    # Get the predicted class
    predicted_class = tf.argmax(logits, axis=1).numpy()[0]

    # Map the predicted class to sentiment label
    sentiment_labels = {0: 'Negative', 1: 'Neutral', 2: 'Positive'}
    predicted_sentiment = sentiment_labels[predicted_class]

    return predicted_sentiment

# Example usage
sample_text = "This product is fantastic! I absolutely love it."
predicted_sentiment = predict_sentiment(sample_text)
print(f"\nText: {sample_text}")
print(f"Predicted Sentiment: {predicted_sentiment}")
```

```
All model checkpoint layers were used when initializing TFRobertaForSequence
Classification.

All the layers of TFRobertaForSequenceClassification were initialized from t
he model checkpoint at model_directory.
If your task is similar to the task the model of the checkpoint was trained
on, you can already use TFRobertaForSequenceClassification for predictions w
ithout further training.
Text: This product is fantastic! I absolutely love it.
Predicted Sentiment: Positive
```

8. Reflections

Andre: Creating these models was very eye opening for me. In particular, I learned a lot about how long it takes in order to properly train models. Along with this, I learned about the importance of the quantity of training data for fine tuning models. Finally, I learned about the importance of saving models having lost 2 models that took upwards of an hour to train each. Having done this project, I am now far more comfortable with google colab and how to train models using GPUs.

Adnan: After preprocessing the data and ensuring that each class had an equal number of samples, I used the pre-trained model "cardiffnlp/twitter-roberta-base-sentiment-latest" and applied the model's tokenizer for tokenization. I used the full set of 127,920 samples, with 42,640 samples for each class: Negative, Neutral, and Positive. I utilized the Trainer and set the following parameters in the TrainingArguments: output directory as "./results", evaluation strategy as "steps", with evaluations and model saves every 450 steps, a reduced batch size of 16 for both training and evaluation, 4 training epochs, a learning rate of 2e-5, weight decay of 0.01, seed set to 42, logging every 50 steps, and mixed precision enabled (fp16). Additionally, gradient accumulation steps were set to 4, simulating a larger batch size, and tensorboard was used for logging. The results I obtained were: for Class 0 (Negative), Precision: 0.85, Recall: 0.84, F1-Score: 0.85; for Class 1 (Neutral), Precision: 0.78, Recall: 0.80, F1-Score: 0.79; and for Class 2 (Positive), Precision: 0.92, Recall: 0.91, F1-Score: 0.92. The overall Precision for each class was [0.8547, 0.7764, 0.9239], Recall was [0.8409, 0.8036, 0.9063], and F1-Score was [0.8478, 0.7898, 0.9151]. I wanted to experiment with changes in batch size, learning rate, and the number of epochs, as well as trying a manual optimizer with grid search. However, the training took 6 hours and 30 minutes, and I lacked the time and resources to implement these changes.

Majed: Training these models taught me the importance of resource management, early in the model training, I had given access to Younes to train his model and he spent all 100 units I bought that day. Additionally, I learned that brute force is never the answer, while training on the entire dataset might be enticing, we should train on a portion of the dataset until we reach a desirable model with proven potential to perform well using the entire dataset. Lastly, while this work isn't in the final report, it is on the github, I learned how to create a pipeline which can intake raw data, and convert it through multiple steps to a desirable and usable output.

Yonotan: In my recent work, I gained valuable insights into the use of LSTM models for sentiment analysis, particularly in handling sequential data like product reviews. Implementing GloVe embeddings enhanced my understanding of pre-trained word embeddings and their role in improving model performance while reducing training time. I also tackled the challenge of data imbalance, learning the importance of addressing skewed label distributions to avoid biased models. Through model evaluation, I observed the differences in accuracy between validation and testing datasets, emphasizing the need for robust evaluation to ensure generalizability. Moving forward, I aim to focus on tuning hyperparameters such as learning rates, dropout rates, and epochs to further optimize performance. Additionally, a deeper analysis of misclassified instances will provide valuable insights into areas where the model struggles, allowing for targeted improvements.

Younes: We applied the pre-trained model from hugging face roBERTa, version “roberta-base”. We reduced the data as we have limited GPU resources. Description of hyperparameters: Batch size = 16 Max length Tokenization = 128 Learning_rate=1e-6 Optimizer= Adam Loss function = Sparse Categorical Cross-entropy Number of epochs = 10

We performed 3 trainings: -First training with 3000 samples resulted in underfitting, (Test loss: 58.82% - Test Accuracy: 76%) -Second training with 30000 samples resulted in overfitting, (Test loss: 56.45% - Test Accuracy: 76.32%) -Third training with 30000 samples resulted in overfitting too, but a slight improvement (Test loss: 55.26% - Test Accuracy: 77.23%)

To improve, we can tune the hyperparameters by increasing the size of batches or adding more epochs. We can also try transfer learning by freezing the first layers, and see how the model can behave by training the last layers. We can also try other learning rates.