

# 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()
```

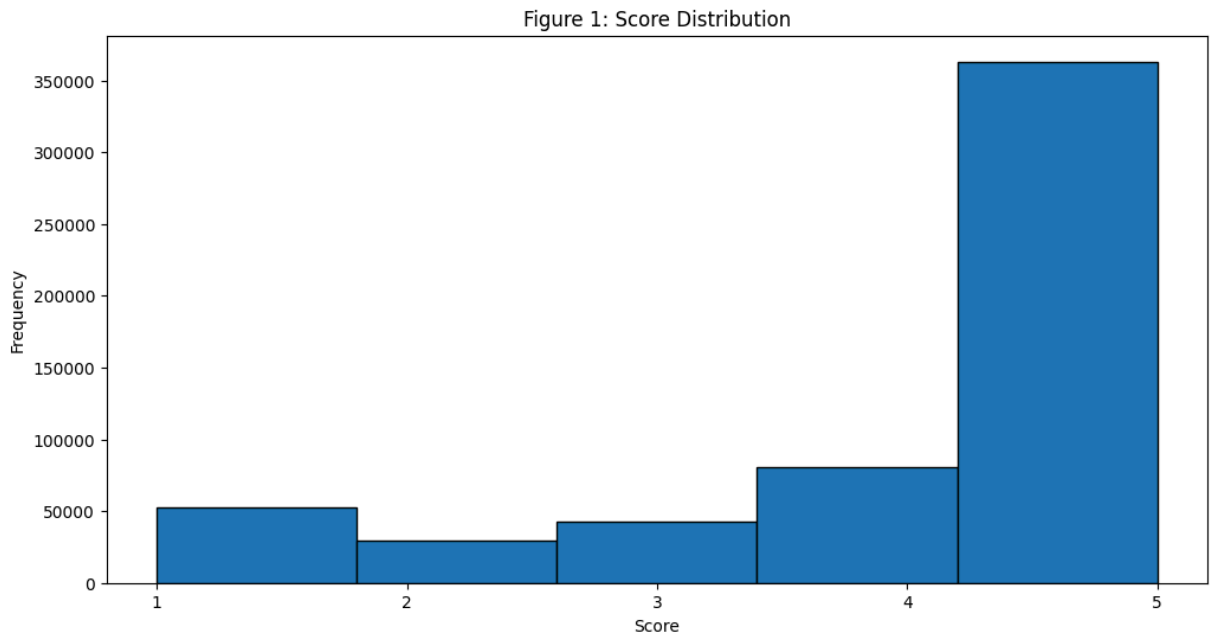


Figure 1 shows that the distribution of the scores is not uniform, which will have to be taken into account during preprocessing. This means that when we are creating our labels, we must either remove samples from categories with higher frequencies or add samples to the categories with low frequencies. 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').generate(text)

# 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()
```

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

```
In [4]: # 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()
```

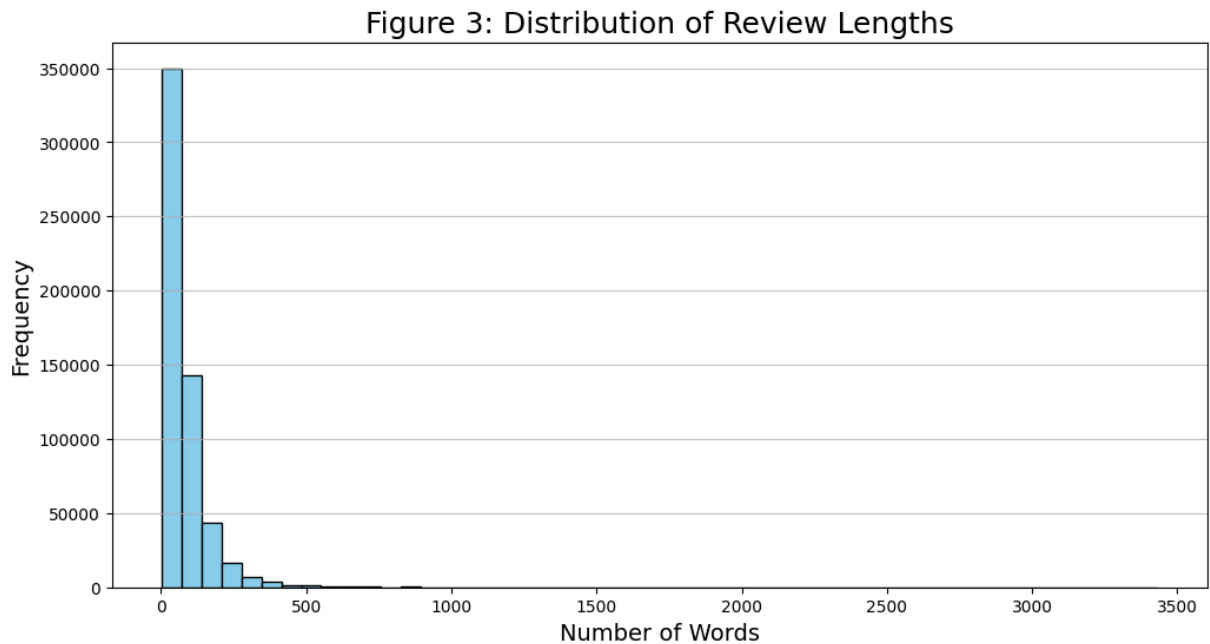


Figure 3 shows that the distribution of the lengths of reviews is heavily right skewed. Since the goal of the model is to gain insight into the sentiment of customers posting on social media posts which tend to be quite short (think YouTube comments or Tweets), it is reasonable to conclude that the lengths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

# 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]: import seaborn as sns

# Function to map 'Score' to 'label' and create a bar chart of label distribution
def plot_sentiment_distribution(df, score_column='Score', label_column='label'):
    """
    Maps the score to sentiment labels, creates the 'label' column, and plots the distribution.

    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=

```

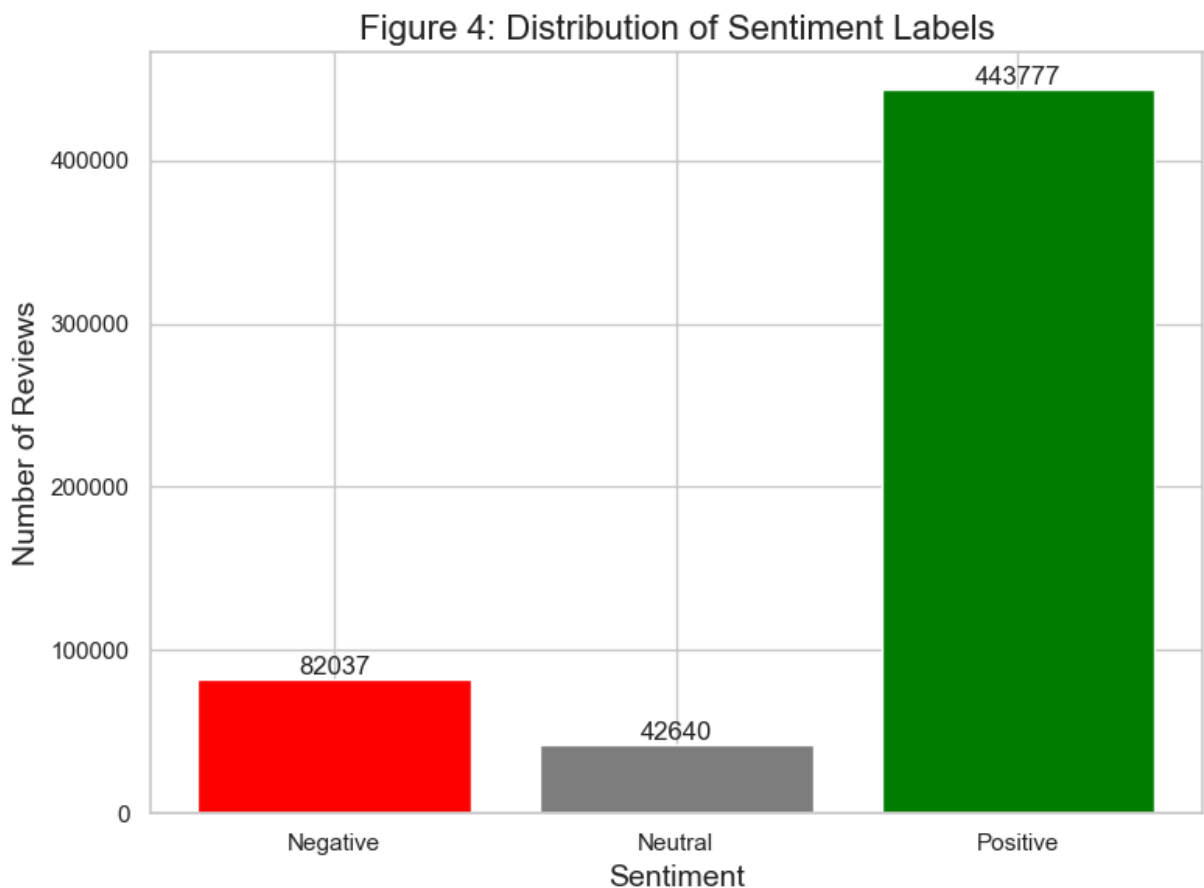


Figure 4 shows the distribution of each label. This plot illustrates the already known issue that we do not have a uniform distribution of the different review sentiments. We fix this by randomly selecting 42,640 samples from the "Positive" sentiment class and the "Negative" sentiment class and dropping 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 significantly 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)

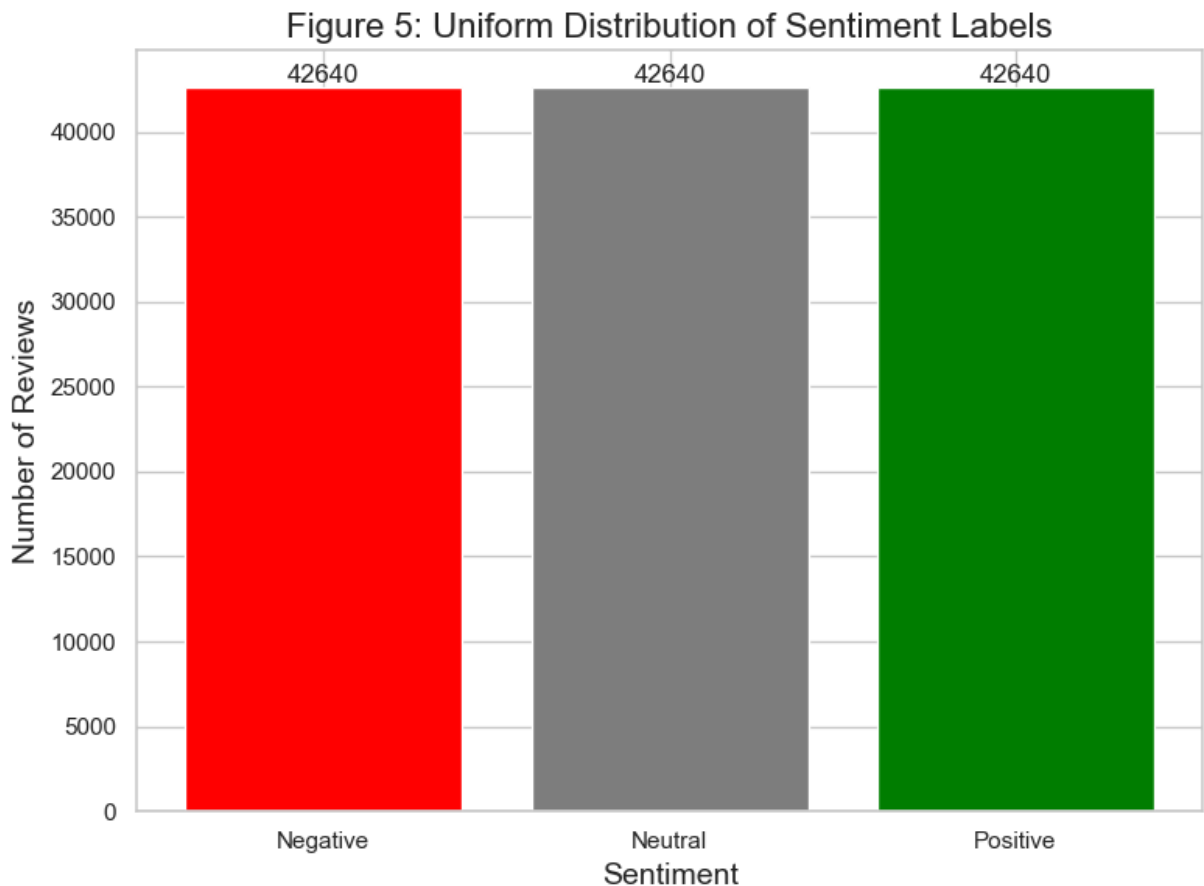
# Shuffle the balanced DataFrame
df_balanced = df_balanced.sample(frac=1, random_state=random_state).reset_index(drop=True)

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

/var/folders/yl/842rvw2j7mq5t40qz5hg61fh0000gn/T/ipykernel\_56877/3387499048.py:8: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns. This behavior is deprecated, and in a future version of pandas the grouping columns will be excluded from the operation. Either pass `include\_groups=False` to exclude the groupings or explicitly select the grouping columns 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]: 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').generate(text)

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

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



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', label
```

Figure 7: Distribution of Sentiment Labels After Accounting for Text Length

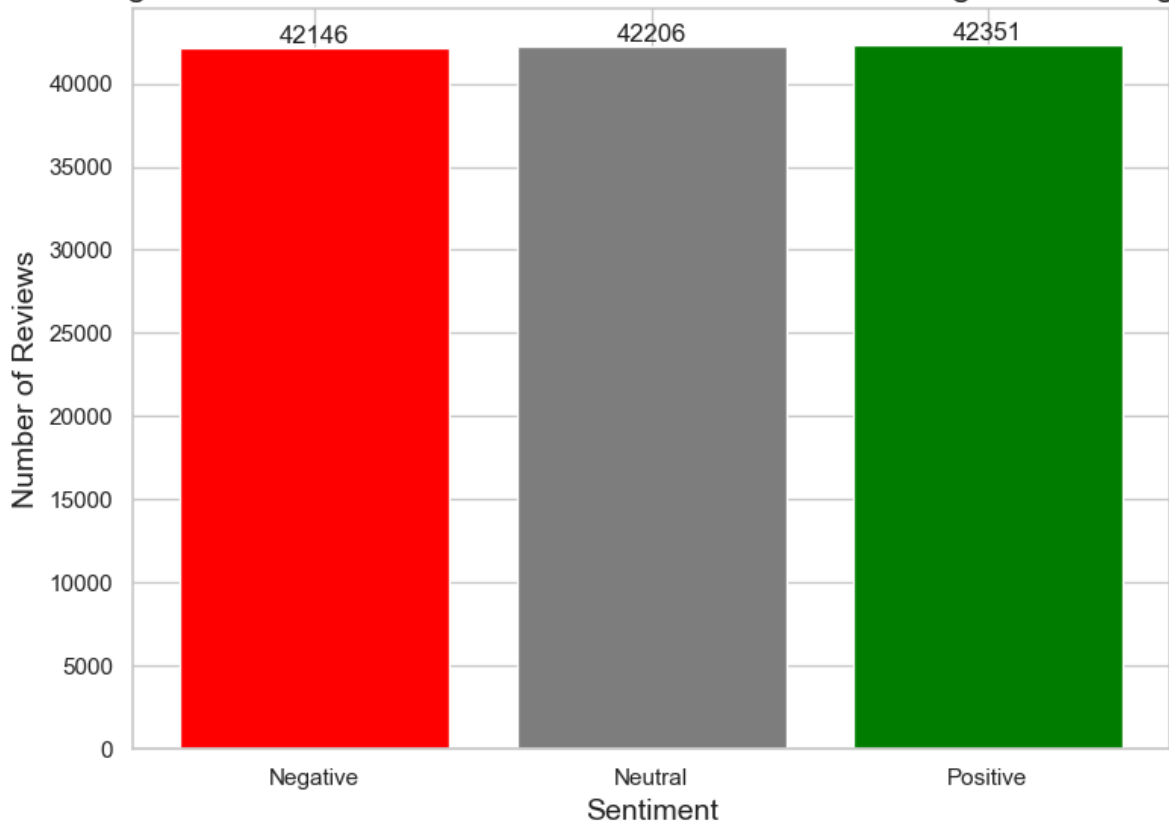


Figure 7 shows that this process had no negative effects on the distribution of sentiments. For the purpose's of training efficiency, we will work with a significantly smaller subset of the data so we can effectively experiemtn with the model. To do this, we will reduce each label to have 5000 rows.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

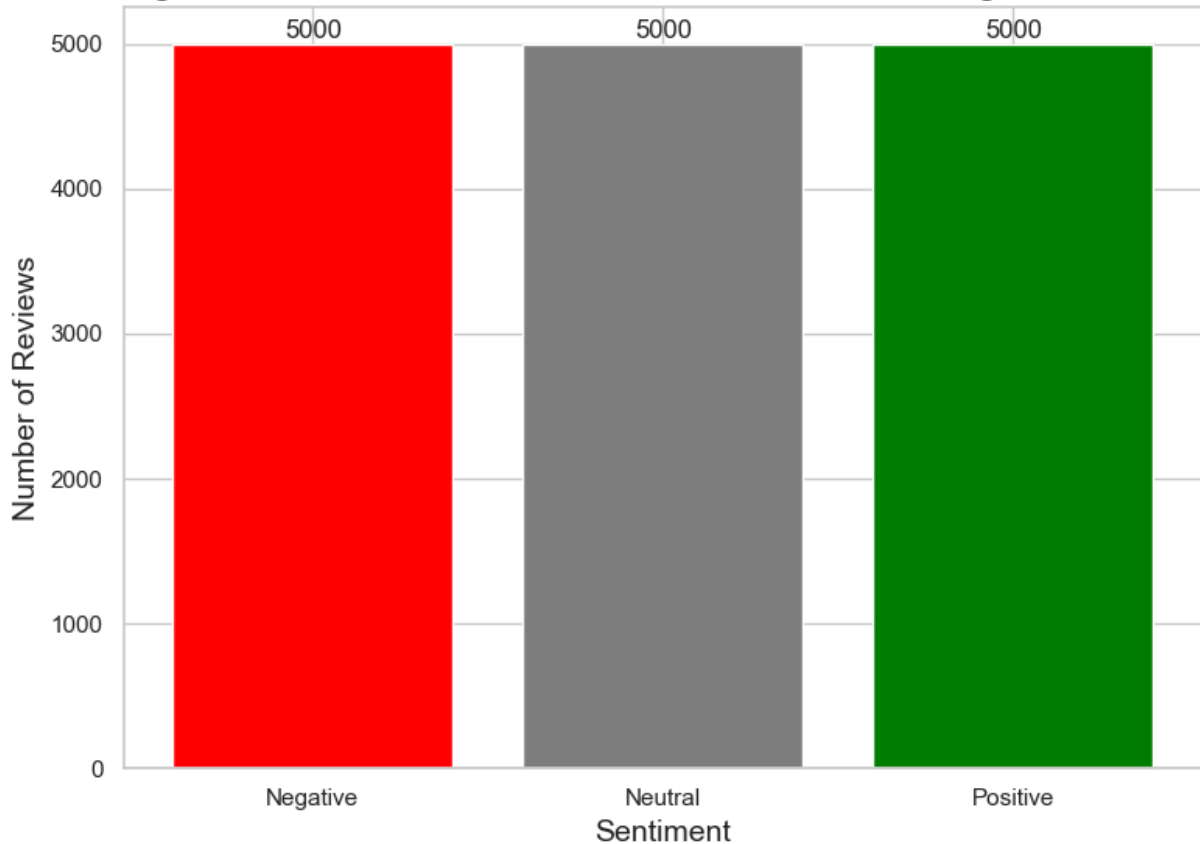
```
In [45]: # Back up the full sized cleaned dataset
full_cleaned_df = df_balanced_filtered.copy()
```

```

experimenting_df = df_balanced_filtered.groupby('label').sample(n=5000, random_state=42)
plot_sentiment_distribution(experimenting_df, score_column='Score', label_column='label')

```

Figure 8: Distribution of Sentiment Labels After Reducing Dataset Size



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_state=42))

# 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 tab (<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_base.py:1601: FutureWarning: `clean_up_tokenization_spaces` was not set. It will be set to `True` by default. This behavior will be deprecated in transformers 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_mask,
        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_attention_mask

# 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, use_batchnorm=True):
    # 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='attention_mask')

    # 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)
```

```

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=output)

return model

```

```

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=learning_rate)
        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 [ ]: 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

```

```

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.layer\_norm.bias', 'lm\_head.layer\_norm.weight', 'lm\_head.dense.bias']

- This IS expected if you are initializing TFRobertaModel from a PyTorch model 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 TFBertForSequenceClassification 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 training.

```

In [ ]: 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, 'batch\_size': 16, 'dropout\_rate': 0.1, 'activation\_function': 'relu', 'use\_batchnorm': True}

Epoch 1/80  
657/657 [=====] - 81s 100ms/step - loss: 1.3414 - accuracy: 0.3602 - val\_loss: 1.0844 - val\_accuracy: 0.3964

Epoch 2/80  
657/657 [=====] - 63s 96ms/step - loss: 1.2345 - accuracy: 0.3923 - val\_loss: 1.0120 - val\_accuracy: 0.4631

Epoch 3/80  
657/657 [=====] - 63s 96ms/step - loss: 1.1664 - accuracy: 0.4189 - val\_loss: 0.9703 - val\_accuracy: 0.5213

Epoch 4/80  
657/657 [=====] - 64s 97ms/step - loss: 1.1236 - accuracy: 0.4438 - val\_loss: 0.9312 - val\_accuracy: 0.5707

Epoch 5/80  
657/657 [=====] - 66s 101ms/step - loss: 1.0794 - accuracy: 0.4653 - val\_loss: 0.8963 - val\_accuracy: 0.6018

Epoch 6/80  
657/657 [=====] - 64s 97ms/step - loss: 1.0318 - accuracy: 0.4993 - val\_loss: 0.8755 - val\_accuracy: 0.6129

Epoch 7/80  
657/657 [=====] - 63s 97ms/step - loss: 1.0052 - accuracy: 0.5130 - val\_loss: 0.8504 - val\_accuracy: 0.6271

Epoch 8/80  
657/657 [=====] - 64s 97ms/step - loss: 0.9994 - accuracy: 0.5175 - val\_loss: 0.8340 - val\_accuracy: 0.6338

Epoch 9/80  
657/657 [=====] - 64s 97ms/step - loss: 0.9760 - accuracy: 0.5286 - val\_loss: 0.8118 - val\_accuracy: 0.6471

Epoch 10/80  
657/657 [=====] - 63s 96ms/step - loss: 0.9558 - accuracy: 0.5373 - val\_loss: 0.8015 - val\_accuracy: 0.6507

Epoch 11/80  
657/657 [=====] - 63s 96ms/step - loss: 0.9501 - accuracy: 0.5476 - val\_loss: 0.7951 - val\_accuracy: 0.6502

Epoch 12/80  
657/657 [=====] - 63s 96ms/step - loss: 0.9418 - accuracy: 0.5554 - val\_loss: 0.7807 - val\_accuracy: 0.6551

Epoch 13/80  
657/657 [=====] - 63s 96ms/step - loss: 0.9296 - accuracy: 0.5579 - val\_loss: 0.7744 - val\_accuracy: 0.6609

Epoch 14/80  
657/657 [=====] - 63s 96ms/step - loss: 0.9202 - accuracy: 0.5650 - val\_loss: 0.7699 - val\_accuracy: 0.6640

Epoch 15/80  
657/657 [=====] - 63s 96ms/step - loss: 0.9199 - accuracy: 0.5670 - val\_loss: 0.7594 - val\_accuracy: 0.6658

Epoch 16/80  
657/657 [=====] - 63s 96ms/step - loss: 0.9125 - accuracy: 0.5749 - val\_loss: 0.7550 - val\_accuracy: 0.6640

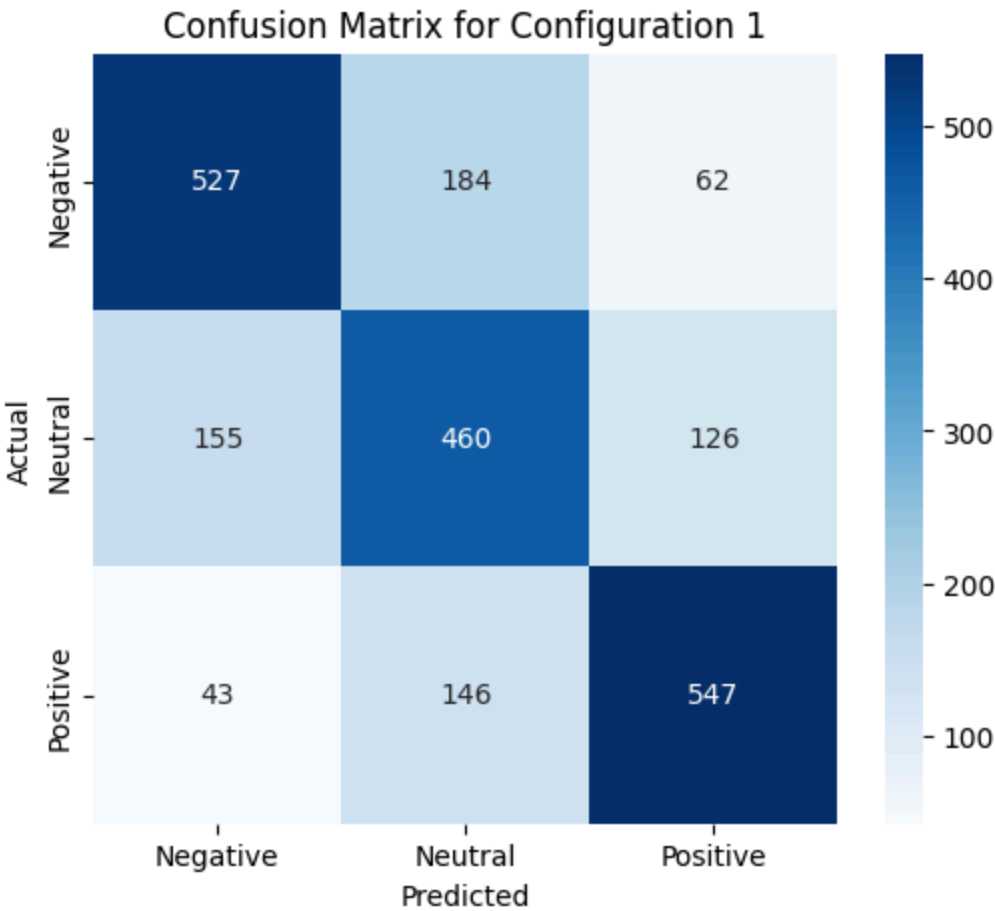
Epoch 17/80  
657/657 [=====] - 63s 97ms/step - loss: 0.9056 - accuracy: 0.5763 - val\_loss: 0.7472 - val\_accuracy: 0.6720

Epoch 18/80  
657/657 [=====] - 63s 97ms/step - loss: 0.8999 - accuracy: 0.5763 - val\_loss: 0.7472 - val\_accuracy: 0.6720

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, 'batch\_size': 16, 'dropout\_rate': 0.2, 'activation\_function': 'relu', 'use\_batchnorm': True}

Epoch 1/80

657/657 [=====] - 75s 100ms/step - loss: 1.3088 - accuracy: 0.3475 - val\_loss: 1.0889 - val\_accuracy: 0.3671

Epoch 2/80

657/657 [=====] - 64s 97ms/step - loss: 1.2587 - accuracy: 0.3684 - val\_loss: 1.0649 - val\_accuracy: 0.4169

Epoch 3/80

657/657 [=====] - 64s 97ms/step - loss: 1.2111 - accuracy: 0.3794 - val\_loss: 1.0413 - val\_accuracy: 0.4684

Epoch 4/80

657/657 [=====] - 63s 96ms/step - loss: 1.1946 - accuracy: 0.3920 - val\_loss: 1.0249 - val\_accuracy: 0.4893

Epoch 5/80

657/657 [=====] - 64s 97ms/step - loss: 1.1591 - accuracy: 0.4123 - val\_loss: 1.0072 - val\_accuracy: 0.5036

Epoch 6/80

657/657 [=====] - 63s 96ms/step - loss: 1.1359 - accuracy: 0.4254 - val\_loss: 0.9859 - val\_accuracy: 0.5200

Epoch 7/80

657/657 [=====] - 63s 96ms/step - loss: 1.0998 - accuracy: 0.4516 - val\_loss: 0.9698 - val\_accuracy: 0.5329

Epoch 8/80

657/657 [=====] - 63s 96ms/step - loss: 1.0843 - accuracy: 0.4690 - val\_loss: 0.9453 - val\_accuracy: 0.5551

Epoch 9/80

657/657 [=====] - 63s 96ms/step - loss: 1.0724 - accuracy: 0.4772 - val\_loss: 0.9302 - val\_accuracy: 0.5582

Epoch 10/80

657/657 [=====] - 64s 97ms/step - loss: 1.0589 - accuracy: 0.4832 - val\_loss: 0.9072 - val\_accuracy: 0.5764

Epoch 11/80

657/657 [=====] - 64s 97ms/step - loss: 1.0348 - accuracy: 0.4989 - val\_loss: 0.8889 - val\_accuracy: 0.5880

Epoch 12/80

657/657 [=====] - 64s 97ms/step - loss: 1.0201 - accuracy: 0.5124 - val\_loss: 0.8775 - val\_accuracy: 0.5951

Epoch 13/80

657/657 [=====] - 63s 96ms/step - loss: 0.9953 - accuracy: 0.5213 - val\_loss: 0.8561 - val\_accuracy: 0.6076

Epoch 14/80

657/657 [=====] - 63s 96ms/step - loss: 0.9990 - accuracy: 0.5190 - val\_loss: 0.8422 - val\_accuracy: 0.6164

Epoch 15/80

657/657 [=====] - 64s 97ms/step - loss: 0.9825 - accuracy: 0.5309 - val\_loss: 0.8300 - val\_accuracy: 0.6276

Epoch 16/80

657/657 [=====] - 63s 96ms/step - loss: 0.9740 - accuracy: 0.5357 - val\_loss: 0.8180 - val\_accuracy: 0.6373

Epoch 17/80

657/657 [=====] - 63s 96ms/step - loss: 0.9613 - accuracy: 0.5447 - val\_loss: 0.8124 - val\_accuracy: 0.6356

Epoch 18/80

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



Epoch 37/80  
657/657 [=====] - 63s 96ms/step - loss: 0.8825 - accuracy: 0.5938 - val\_loss: 0.7283 - val\_accuracy: 0.6787  
Epoch 38/80  
657/657 [=====] - 64s 97ms/step - loss: 0.8624 - accuracy: 0.5997 - val\_loss: 0.7217 - val\_accuracy: 0.6840  
Epoch 39/80  
657/657 [=====] - 63s 97ms/step - loss: 0.8624 - accuracy: 0.6029 - val\_loss: 0.7148 - val\_accuracy: 0.6853  
Epoch 40/80  
657/657 [=====] - 63s 96ms/step - loss: 0.8697 - accuracy: 0.6002 - val\_loss: 0.7165 - val\_accuracy: 0.6836  
Epoch 41/80  
657/657 [=====] - 63s 97ms/step - loss: 0.8584 - accuracy: 0.6061 - val\_loss: 0.7144 - val\_accuracy: 0.6840  
Epoch 42/80  
657/657 [=====] - 63s 97ms/step - loss: 0.8528 - accuracy: 0.6068 - val\_loss: 0.7106 - val\_accuracy: 0.6884  
Epoch 43/80  
657/657 [=====] - 63s 96ms/step - loss: 0.8592 - accuracy: 0.6056 - val\_loss: 0.7118 - val\_accuracy: 0.6853  
Epoch 44/80  
657/657 [=====] - 64s 97ms/step - loss: 0.8487 - accuracy: 0.6084 - val\_loss: 0.7070 - val\_accuracy: 0.6844  
Epoch 45/80  
657/657 [=====] - 63s 96ms/step - loss: 0.8555 - accuracy: 0.6113 - val\_loss: 0.7095 - val\_accuracy: 0.6849  
Epoch 46/80  
657/657 [=====] - 64s 97ms/step - loss: 0.8581 - accuracy: 0.6120 - val\_loss: 0.7033 - val\_accuracy: 0.6871  
Epoch 47/80  
657/657 [=====] - 63s 96ms/step - loss: 0.8490 - accuracy: 0.6094 - val\_loss: 0.7079 - val\_accuracy: 0.6831  
Epoch 48/80  
657/657 [=====] - 63s 96ms/step - loss: 0.8473 - accuracy: 0.6132 - val\_loss: 0.7084 - val\_accuracy: 0.6871  
Epoch 49/80  
657/657 [=====] - 63s 96ms/step - loss: 0.8464 - accuracy: 0.6117 - val\_loss: 0.7050 - val\_accuracy: 0.6822  
Epoch 50/80  
657/657 [=====] - 63s 97ms/step - loss: 0.8429 - accuracy: 0.6107 - val\_loss: 0.7050 - val\_accuracy: 0.6818  
141/141 [=====] - 10s 74ms/step - loss: 0.7324 - accuracy: 0.6800

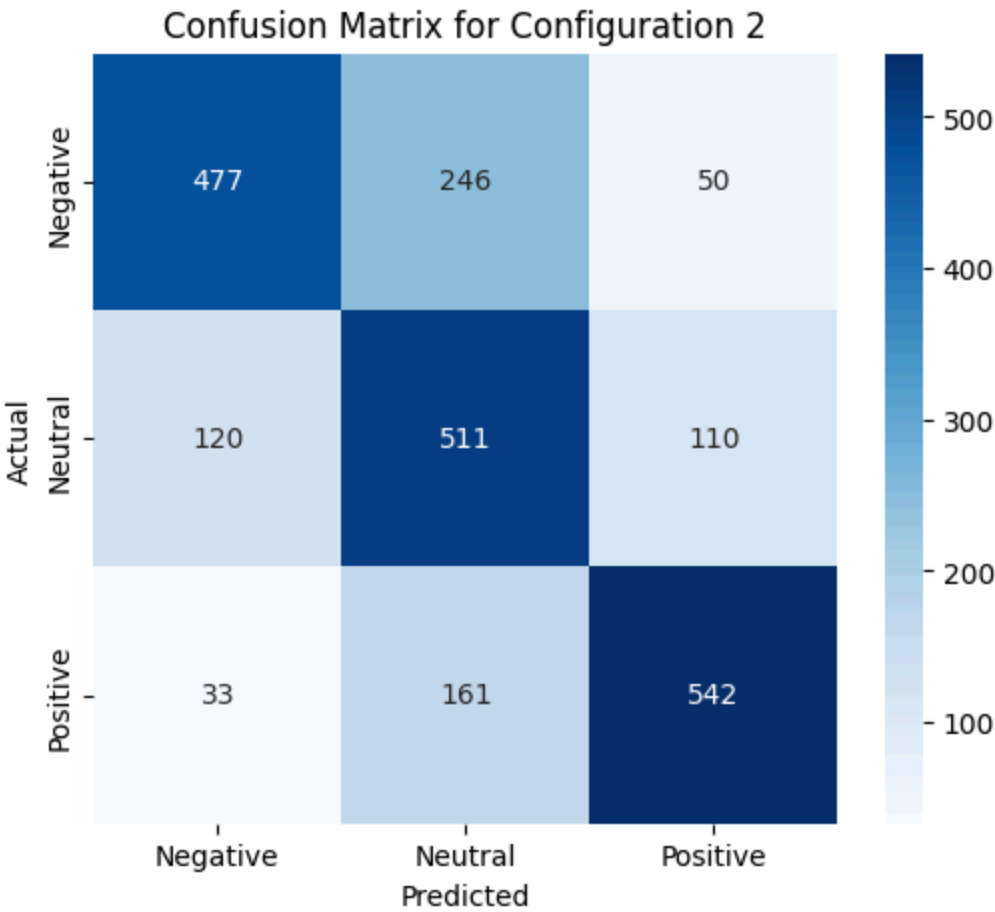
Test accuracy: 0.6800000071525574

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\_batchnorm': True}

Epoch 1/80  
329/329 [=====] - 71s 186ms/step - loss: 1.1646 - accuracy: 0.4133 - val\_loss: 0.9168 - val\_accuracy: 0.5569

Epoch 2/80  
329/329 [=====] - 59s 181ms/step - loss: 1.0241 - accuracy: 0.4986 - val\_loss: 0.8256 - val\_accuracy: 0.6178

Epoch 3/80  
329/329 [=====] - 59s 179ms/step - loss: 0.9853 - accuracy: 0.5235 - val\_loss: 0.7950 - val\_accuracy: 0.6342

Epoch 4/80  
329/329 [=====] - 58s 177ms/step - loss: 0.9407 - accuracy: 0.5493 - val\_loss: 0.7703 - val\_accuracy: 0.6564

Epoch 5/80  
329/329 [=====] - 58s 177ms/step - loss: 0.9257 - accuracy: 0.5671 - val\_loss: 0.7540 - val\_accuracy: 0.6618

Epoch 6/80  
329/329 [=====] - 58s 176ms/step - loss: 0.9061 - accuracy: 0.5725 - val\_loss: 0.7654 - val\_accuracy: 0.6609

Epoch 7/80  
329/329 [=====] - 59s 180ms/step - loss: 0.8978 - accuracy: 0.5814 - val\_loss: 0.7447 - val\_accuracy: 0.6640

Epoch 8/80  
329/329 [=====] - 60s 182ms/step - loss: 0.8962 - accuracy: 0.5843 - val\_loss: 0.7394 - val\_accuracy: 0.6693

Epoch 9/80  
329/329 [=====] - 59s 178ms/step - loss: 0.8791 - accuracy: 0.5898 - val\_loss: 0.7354 - val\_accuracy: 0.6760

Epoch 10/80  
329/329 [=====] - 58s 177ms/step - loss: 0.8792 - accuracy: 0.5903 - val\_loss: 0.7484 - val\_accuracy: 0.6684

Epoch 11/80  
329/329 [=====] - 58s 176ms/step - loss: 0.8602 - accuracy: 0.6021 - val\_loss: 0.7249 - val\_accuracy: 0.6818

Epoch 12/80  
329/329 [=====] - 59s 178ms/step - loss: 0.8667 - accuracy: 0.6015 - val\_loss: 0.7213 - val\_accuracy: 0.6813

Epoch 13/80  
329/329 [=====] - 58s 177ms/step - loss: 0.8571 - accuracy: 0.6025 - val\_loss: 0.7157 - val\_accuracy: 0.6867

Epoch 14/80  
329/329 [=====] - 58s 176ms/step - loss: 0.8540 - accuracy: 0.6099 - val\_loss: 0.7184 - val\_accuracy: 0.6791

Epoch 15/80  
329/329 [=====] - 60s 181ms/step - loss: 0.8424 - accuracy: 0.6165 - val\_loss: 0.7170 - val\_accuracy: 0.6836

Epoch 16/80  
329/329 [=====] - 58s 176ms/step - loss: 0.8435 - accuracy: 0.6099 - val\_loss: 0.7229 - val\_accuracy: 0.6796

Epoch 17/80  
329/329 [=====] - 58s 176ms/step - loss: 0.8460 - accuracy: 0.6117 - val\_loss: 0.7123 - val\_accuracy: 0.6844

Epoch 18/80  
329/329 [=====] - 58s 178ms/step - loss: 0.8430 - a

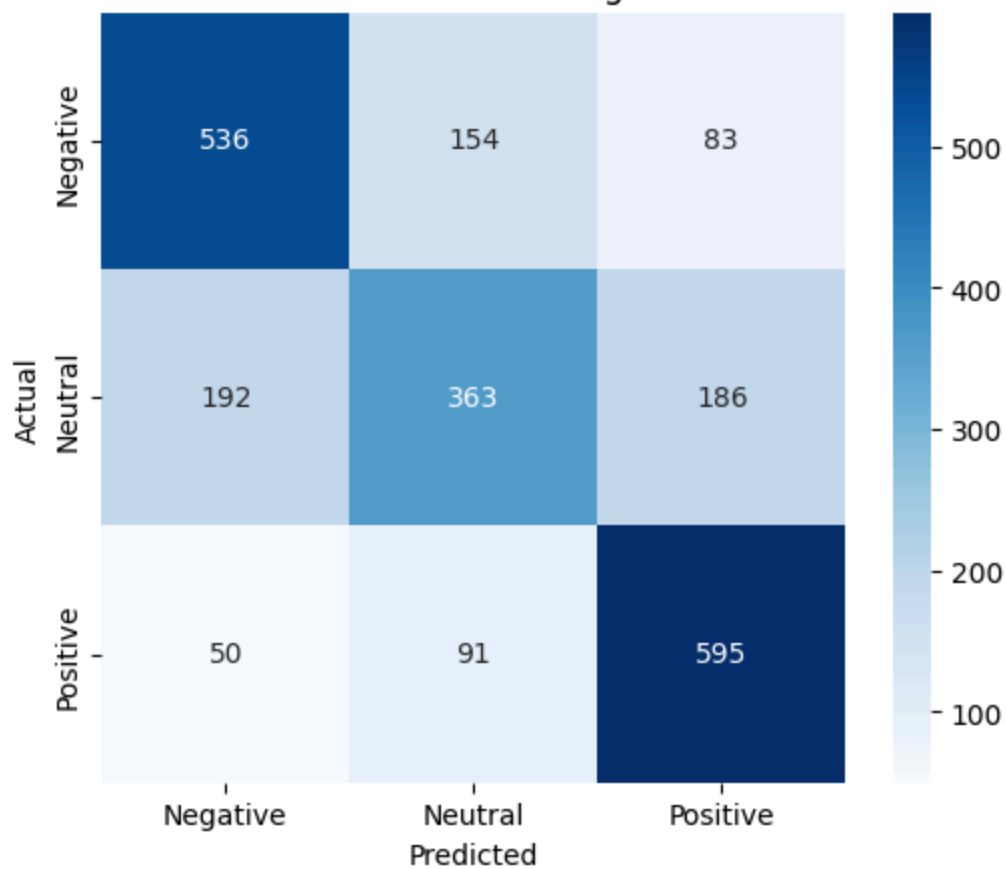
```

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



Training configuration 4/6: {'optimizer': 'adam', 'learning\_rate': 3e-05, 'batch\_size': 32, 'dropout\_rate': 0.1, 'activation\_function': 'relu', 'use\_batchnorm': True}

Epoch 1/80  
329/329 [=====] - 69s 186ms/step - loss: 1.2325 - accuracy: 0.3845 - val\_loss: 1.0759 - val\_accuracy: 0.4009

Epoch 2/80  
329/329 [=====] - 58s 176ms/step - loss: 1.0995 - accuracy: 0.4637 - val\_loss: 0.9677 - val\_accuracy: 0.5240

Epoch 3/80  
329/329 [=====] - 58s 176ms/step - loss: 1.0272 - accuracy: 0.5068 - val\_loss: 0.9125 - val\_accuracy: 0.5609

Epoch 4/80  
329/329 [=====] - 58s 177ms/step - loss: 0.9839 - accuracy: 0.5324 - val\_loss: 0.8582 - val\_accuracy: 0.5884

Epoch 5/80  
329/329 [=====] - 59s 180ms/step - loss: 0.9542 - accuracy: 0.5526 - val\_loss: 0.8174 - val\_accuracy: 0.6218

Epoch 6/80  
329/329 [=====] - 59s 178ms/step - loss: 0.9307 - accuracy: 0.5630 - val\_loss: 0.7895 - val\_accuracy: 0.6364

Epoch 7/80  
329/329 [=====] - 58s 176ms/step - loss: 0.9017 - accuracy: 0.5827 - val\_loss: 0.7576 - val\_accuracy: 0.6511

Epoch 8/80  
329/329 [=====] - 59s 178ms/step - loss: 0.8905 - accuracy: 0.5927 - val\_loss: 0.7424 - val\_accuracy: 0.6622

Epoch 9/80  
329/329 [=====] - 58s 177ms/step - loss: 0.8758 - accuracy: 0.5964 - val\_loss: 0.7455 - val\_accuracy: 0.6609

Epoch 10/80  
329/329 [=====] - 58s 176ms/step - loss: 0.8596 - accuracy: 0.6075 - val\_loss: 0.7302 - val\_accuracy: 0.6676

Epoch 11/80  
329/329 [=====] - 59s 178ms/step - loss: 0.8585 - accuracy: 0.6048 - val\_loss: 0.7128 - val\_accuracy: 0.6782

Epoch 12/80  
329/329 [=====] - 60s 181ms/step - loss: 0.8542 - accuracy: 0.6094 - val\_loss: 0.7047 - val\_accuracy: 0.6889

Epoch 13/80  
329/329 [=====] - 59s 181ms/step - loss: 0.8429 - accuracy: 0.6165 - val\_loss: 0.7019 - val\_accuracy: 0.6893

Epoch 14/80  
329/329 [=====] - 58s 176ms/step - loss: 0.8335 - accuracy: 0.6172 - val\_loss: 0.6947 - val\_accuracy: 0.6831

Epoch 15/80  
329/329 [=====] - 58s 176ms/step - loss: 0.8286 - accuracy: 0.6240 - val\_loss: 0.6930 - val\_accuracy: 0.6831

Epoch 16/80  
329/329 [=====] - 58s 176ms/step - loss: 0.8269 - accuracy: 0.6218 - val\_loss: 0.6910 - val\_accuracy: 0.6884

Epoch 17/80  
329/329 [=====] - 58s 177ms/step - loss: 0.8261 - accuracy: 0.6211 - val\_loss: 0.6891 - val\_accuracy: 0.6804

Epoch 18/80  
329/329 [=====] - 59s 178ms/step - loss: 0.8168 - accuracy: 0.6211 - val\_loss: 0.6891 - val\_accuracy: 0.6804

```

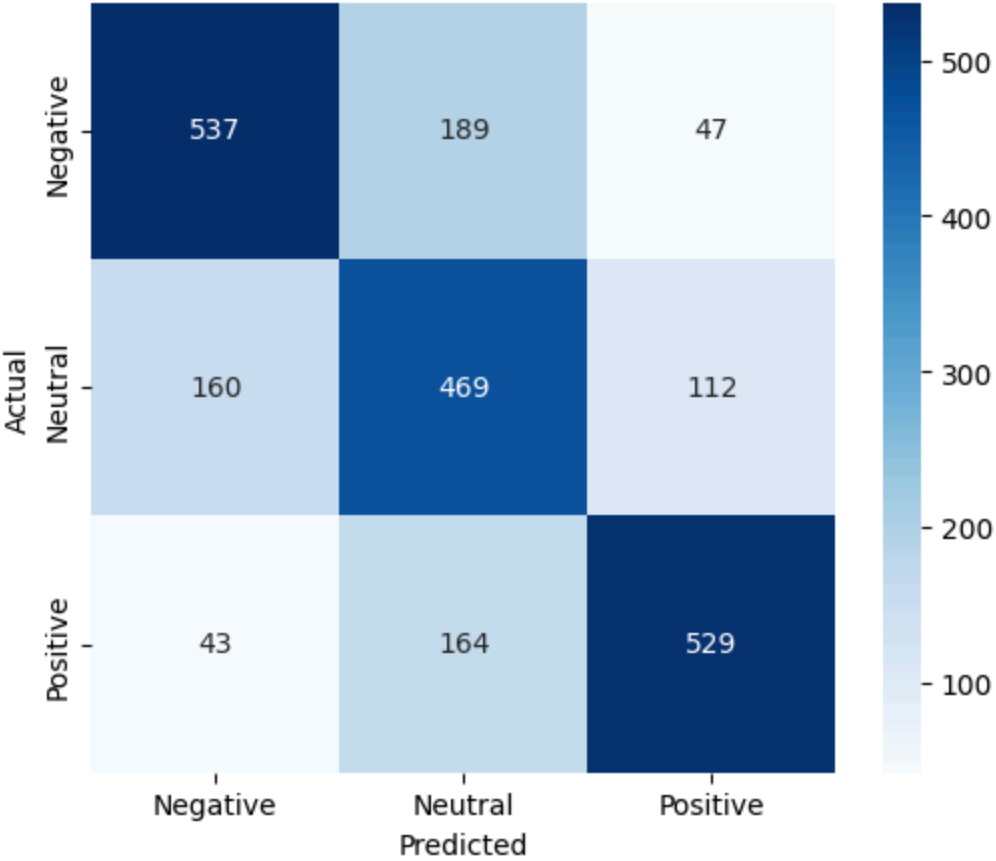
ccuracy: 0.6271 - val_loss: 0.6865 - val_accuracy: 0.6849
Epoch 19/80
329/329 [=====] - 58s 177ms/step - loss: 0.8082 - a
ccuracy: 0.6328 - val_loss: 0.6838 - val_accuracy: 0.6924
Epoch 20/80
329/329 [=====] - 58s 176ms/step - loss: 0.8109 - a
ccuracy: 0.6311 - val_loss: 0.6822 - val_accuracy: 0.6893
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





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\_batchnorm': True}

Epoch 1/80

657/657 [=====] - 74s 100ms/step - loss: 1.1607 - accuracy: 0.4116 - val\_loss: 0.8530 - val\_accuracy: 0.6124

Epoch 2/80

657/657 [=====] - 64s 98ms/step - loss: 1.0218 - accuracy: 0.5026 - val\_loss: 0.7762 - val\_accuracy: 0.6493

Epoch 3/80

657/657 [=====] - 63s 96ms/step - loss: 0.9790 - accuracy: 0.5250 - val\_loss: 0.7458 - val\_accuracy: 0.6569

Epoch 4/80

657/657 [=====] - 63s 96ms/step - loss: 0.9364 - accuracy: 0.5524 - val\_loss: 0.7274 - val\_accuracy: 0.6693

Epoch 5/80

657/657 [=====] - 63s 97ms/step - loss: 0.9222 - accuracy: 0.5597 - val\_loss: 0.7226 - val\_accuracy: 0.6769

Epoch 6/80

657/657 [=====] - 64s 97ms/step - loss: 0.9090 - accuracy: 0.5733 - val\_loss: 0.7170 - val\_accuracy: 0.6733

Epoch 7/80

657/657 [=====] - 65s 98ms/step - loss: 0.8985 - accuracy: 0.5810 - val\_loss: 0.7155 - val\_accuracy: 0.6716

Epoch 8/80

657/657 [=====] - 64s 98ms/step - loss: 0.8905 - accuracy: 0.5907 - val\_loss: 0.7105 - val\_accuracy: 0.6778

Epoch 9/80

657/657 [=====] - 65s 98ms/step - loss: 0.8869 - accuracy: 0.5890 - val\_loss: 0.7080 - val\_accuracy: 0.6849

Epoch 10/80

657/657 [=====] - 64s 98ms/step - loss: 0.8777 - accuracy: 0.5950 - val\_loss: 0.6973 - val\_accuracy: 0.6898

Epoch 11/80

657/657 [=====] - 64s 98ms/step - loss: 0.8786 - accuracy: 0.5947 - val\_loss: 0.6973 - val\_accuracy: 0.6947

Epoch 12/80

657/657 [=====] - 65s 98ms/step - loss: 0.8649 - accuracy: 0.6015 - val\_loss: 0.6896 - val\_accuracy: 0.6902

Epoch 13/80

657/657 [=====] - 64s 97ms/step - loss: 0.8633 - accuracy: 0.6028 - val\_loss: 0.6916 - val\_accuracy: 0.6907

Epoch 14/80

657/657 [=====] - 64s 98ms/step - loss: 0.8517 - accuracy: 0.6082 - val\_loss: 0.6890 - val\_accuracy: 0.6920

Epoch 15/80

657/657 [=====] - 64s 97ms/step - loss: 0.8550 - accuracy: 0.6141 - val\_loss: 0.6916 - val\_accuracy: 0.6902

Epoch 16/80

657/657 [=====] - 63s 96ms/step - loss: 0.8495 - accuracy: 0.6125 - val\_loss: 0.6894 - val\_accuracy: 0.6902

Epoch 17/80

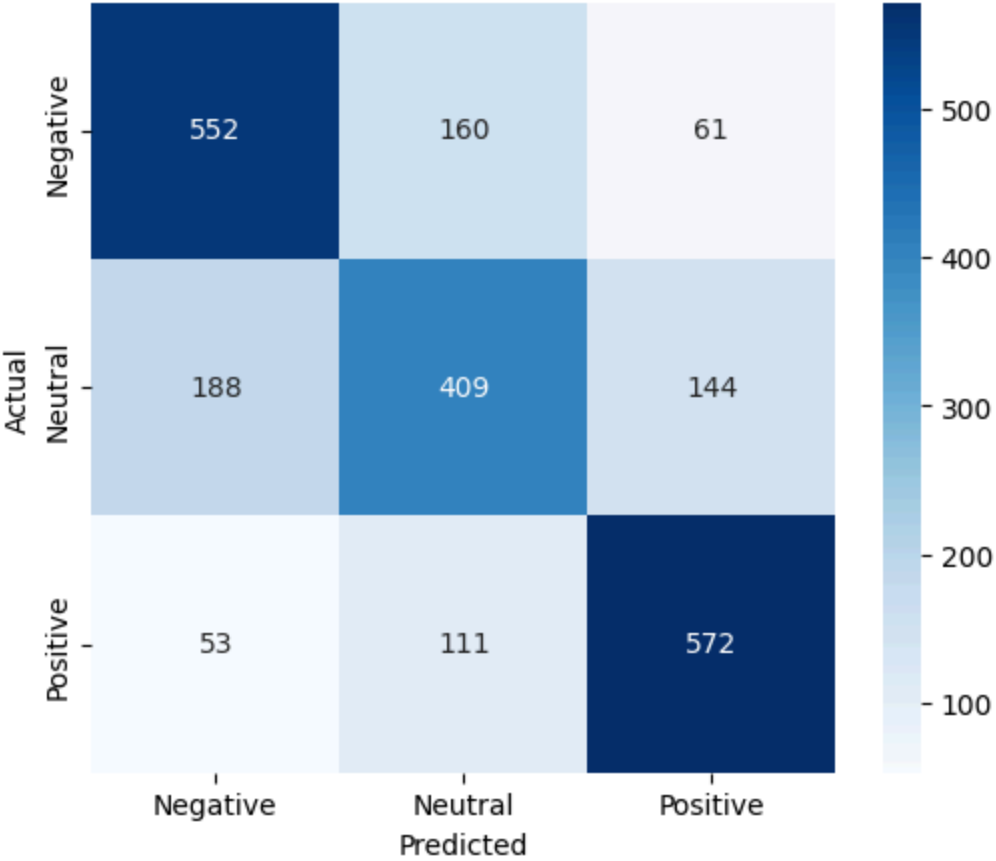
657/657 [=====] - 64s 97ms/step - loss: 0.8397 - accuracy: 0.6161 - val\_loss: 0.6891 - val\_accuracy: 0.6911

Epoch 18/80

657/657 [=====] - 65s 98ms/step - loss: 0.8504 - accuracy: 0.6118 - val\_loss: 0.6872 - val\_accuracy: 0.6916  
Epoch 19/80  
657/657 [=====] - 65s 98ms/step - loss: 0.8328 - accuracy: 0.6201 - val\_loss: 0.6838 - val\_accuracy: 0.6938  
Epoch 20/80  
657/657 [=====] - 64s 98ms/step - loss: 0.8340 - accuracy: 0.6230 - val\_loss: 0.6844 - val\_accuracy: 0.6978  
Epoch 21/80  
657/657 [=====] - 64s 98ms/step - loss: 0.8379 - accuracy: 0.6188 - val\_loss: 0.6932 - val\_accuracy: 0.6898  
Epoch 22/80  
657/657 [=====] - 65s 98ms/step - loss: 0.8309 - accuracy: 0.6245 - val\_loss: 0.6777 - val\_accuracy: 0.6982  
Epoch 23/80  
657/657 [=====] - 64s 98ms/step - loss: 0.8247 - accuracy: 0.6294 - val\_loss: 0.6829 - val\_accuracy: 0.6947  
Epoch 24/80  
657/657 [=====] - 64s 98ms/step - loss: 0.8332 - accuracy: 0.6194 - val\_loss: 0.6795 - val\_accuracy: 0.6973  
Epoch 25/80  
657/657 [=====] - 63s 96ms/step - loss: 0.8281 - accuracy: 0.6230 - val\_loss: 0.6801 - val\_accuracy: 0.6978  
Epoch 26/80  
657/657 [=====] - 64s 97ms/step - loss: 0.8155 - accuracy: 0.6337 - val\_loss: 0.6783 - val\_accuracy: 0.6996  
141/141 [=====] - 10s 74ms/step - loss: 0.7241 - accuracy: 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



Training configuration 6/6: {'optimizer': 'adam', 'learning\_rate': 3e-05, 'batch\_size': 16, 'dropout\_rate': 0.2, 'activation\_function': 'relu', 'use\_batchnorm': True}

Epoch 1/80  
657/657 [=====] - 74s 100ms/step - loss: 1.3482 - accuracy: 0.3550 - val\_loss: 1.0332 - val\_accuracy: 0.4782

Epoch 2/80  
657/657 [=====] - 64s 98ms/step - loss: 1.1878 - accuracy: 0.4230 - val\_loss: 0.9304 - val\_accuracy: 0.5756

Epoch 3/80  
657/657 [=====] - 64s 98ms/step - loss: 1.1114 - accuracy: 0.4662 - val\_loss: 0.8691 - val\_accuracy: 0.6076

Epoch 4/80  
657/657 [=====] - 63s 97ms/step - loss: 1.0518 - accuracy: 0.5013 - val\_loss: 0.8196 - val\_accuracy: 0.6378

Epoch 5/80  
657/657 [=====] - 64s 97ms/step - loss: 1.0183 - accuracy: 0.5194 - val\_loss: 0.7807 - val\_accuracy: 0.6600

Epoch 6/80  
657/657 [=====] - 64s 97ms/step - loss: 0.9804 - accuracy: 0.5428 - val\_loss: 0.7592 - val\_accuracy: 0.6702

Epoch 7/80  
657/657 [=====] - 63s 97ms/step - loss: 0.9561 - accuracy: 0.5544 - val\_loss: 0.7450 - val\_accuracy: 0.6698

Epoch 8/80  
657/657 [=====] - 65s 98ms/step - loss: 0.9548 - accuracy: 0.5575 - val\_loss: 0.7329 - val\_accuracy: 0.6653

Epoch 9/80  
657/657 [=====] - 64s 98ms/step - loss: 0.9325 - accuracy: 0.5627 - val\_loss: 0.7243 - val\_accuracy: 0.6773

Epoch 10/80  
657/657 [=====] - 63s 97ms/step - loss: 0.9246 - accuracy: 0.5732 - val\_loss: 0.7209 - val\_accuracy: 0.6764

Epoch 11/80  
657/657 [=====] - 64s 97ms/step - loss: 0.9088 - accuracy: 0.5789 - val\_loss: 0.7185 - val\_accuracy: 0.6773

Epoch 12/80  
657/657 [=====] - 64s 98ms/step - loss: 0.9068 - accuracy: 0.5835 - val\_loss: 0.7123 - val\_accuracy: 0.6764

Epoch 13/80  
657/657 [=====] - 63s 96ms/step - loss: 0.9009 - accuracy: 0.5838 - val\_loss: 0.7122 - val\_accuracy: 0.6796

Epoch 14/80  
657/657 [=====] - 64s 97ms/step - loss: 0.8809 - accuracy: 0.5937 - val\_loss: 0.7048 - val\_accuracy: 0.6831

Epoch 15/80  
657/657 [=====] - 64s 97ms/step - loss: 0.8779 - accuracy: 0.5977 - val\_loss: 0.7074 - val\_accuracy: 0.6867

Epoch 16/80  
657/657 [=====] - 64s 97ms/step - loss: 0.8689 - accuracy: 0.6047 - val\_loss: 0.6997 - val\_accuracy: 0.6867

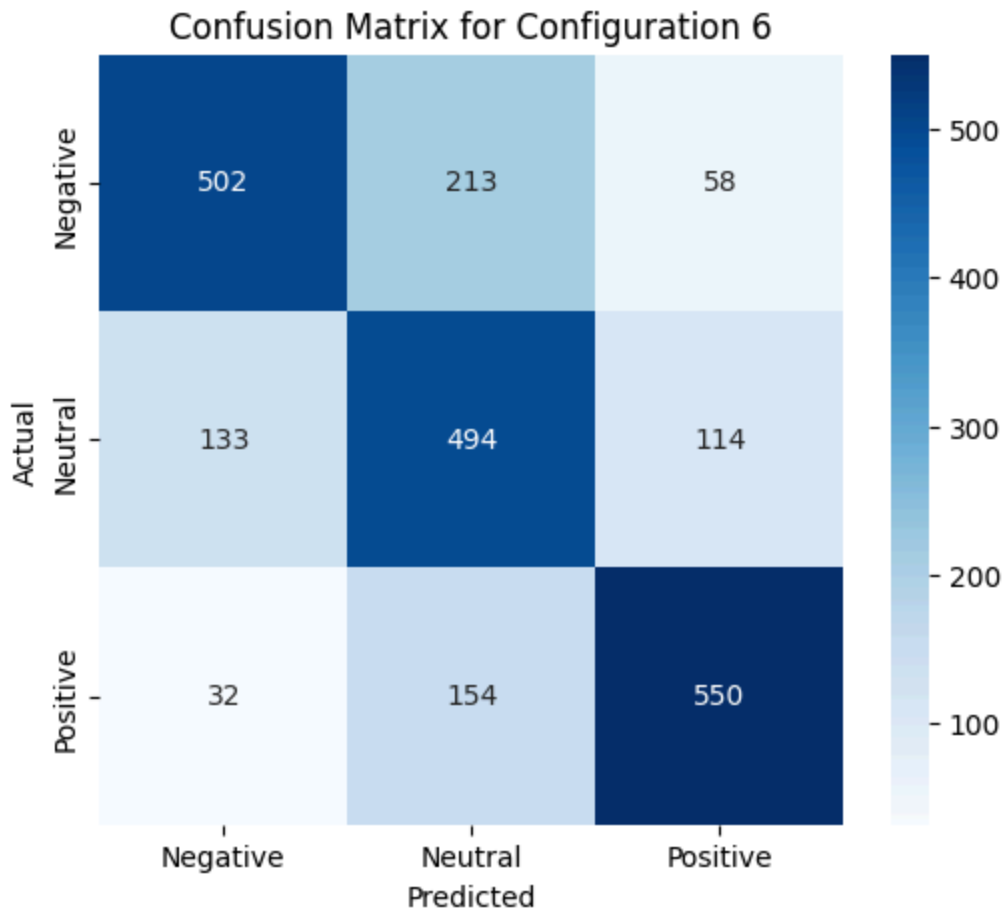
Epoch 17/80  
657/657 [=====] - 63s 97ms/step - loss: 0.8709 - accuracy: 0.6018 - val\_loss: 0.6946 - val\_accuracy: 0.6871

Epoch 18/80  
657/657 [=====] - 64s 98ms/step - loss: 0.8561 - accuracy: 0.6018 - val\_loss: 0.6946 - val\_accuracy: 0.6871

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 - accuracy: 0.6467 - val\_loss: 0.6720 - val\_accuracy: 0.7044  
Epoch 38/80  
657/657 [=====] - 64s 97ms/step - loss: 0.8008 - accuracy: 0.6415 - val\_loss: 0.6745 - val\_accuracy: 0.7031  
Epoch 39/80  
657/657 [=====] - 64s 98ms/step - loss: 0.7926 - accuracy: 0.6440 - val\_loss: 0.6706 - val\_accuracy: 0.7013  
141/141 [=====] - 10s 73ms/step - loss: 0.7023 - accuracy: 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_ids]

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 negatives
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[predicted_label]))
    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[predicted_label]))

# 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, 'batch\_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 mug

True Label: Neutral

Predicted Label: Positive

Text: i love gloria jeans cinnamon nut strudel coffee and decided to try this the flavoring is a little overpowering 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 cliff kid z bars i decided to try these based on all of the great reviews we bought the strawberry and no one in the house likes them my kids are picky so i always try the treats just to see if they are even worth trying i eat healthy and dont mind most healthy snacks but even i did not care for these i am going to try another flavor but the strawberry just didnt cut 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 make funnel cakes i had been using a kitchen funnel which made a big mess no matter 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 some 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 highly recommend this product

True Label: Positive

Predicted Label: Neutral

Text: i got a larger one than my puppy probably required and she loves dragging it around playing tug of war and just chewing on it unfortunately it does shed i dont know if there are rope toys out there that are any sturdier but 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 future version.
      warnings.warn("`seed_generator` is deprecated and will be removed in a future version.", UserWarning)
Best model saved to 'best_model.keras'.
```

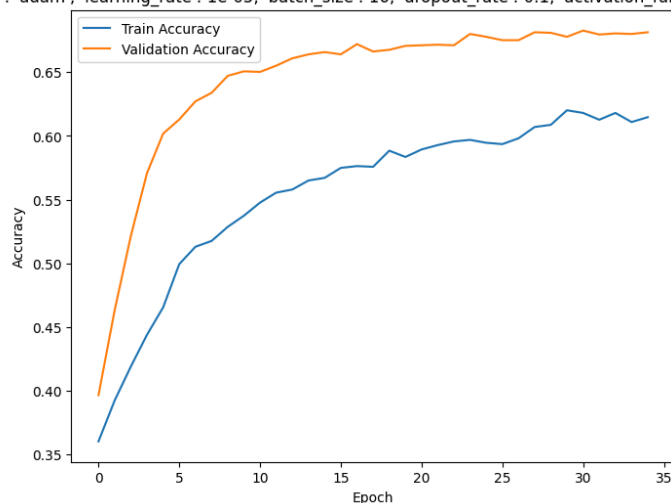
### 6.1.3 Batch-Normalized Models: Visualizations

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).

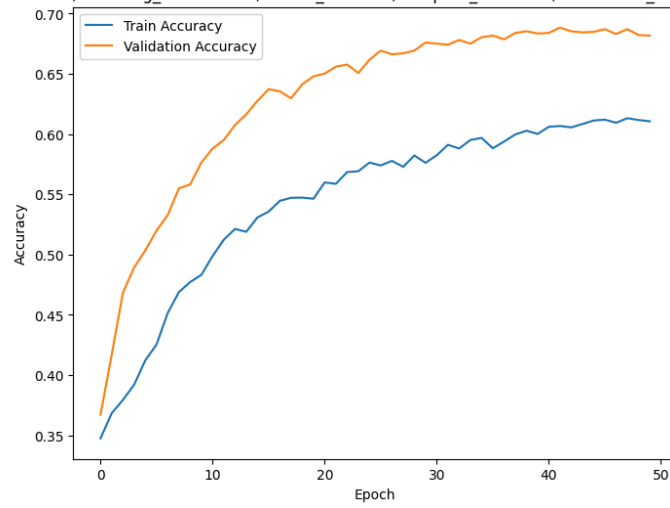
```
In [ ]: # Visualize the results
```

```
# 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()
```

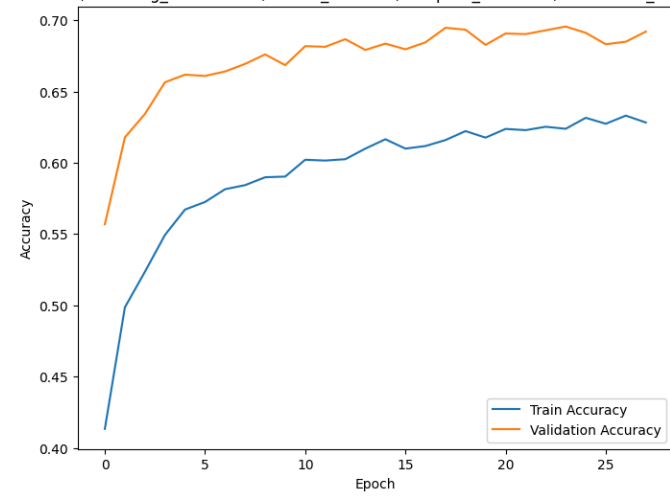
Configuration 1: {'optimizer': 'adam', 'learning\_rate': 1e-05, 'batch\_size': 16, 'dropout\_rate': 0.1, 'activation\_function': 'relu', 'use\_batchnorm': True}



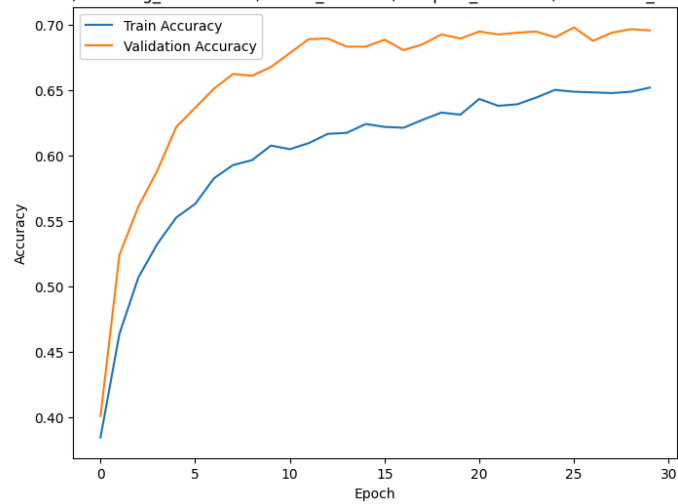
Configuration 2: {'optimizer': 'adam', 'learning\_rate': 1e-05, 'batch\_size': 16, 'dropout\_rate': 0.2, 'activation\_function': 'relu', 'use\_batchnorm': True}



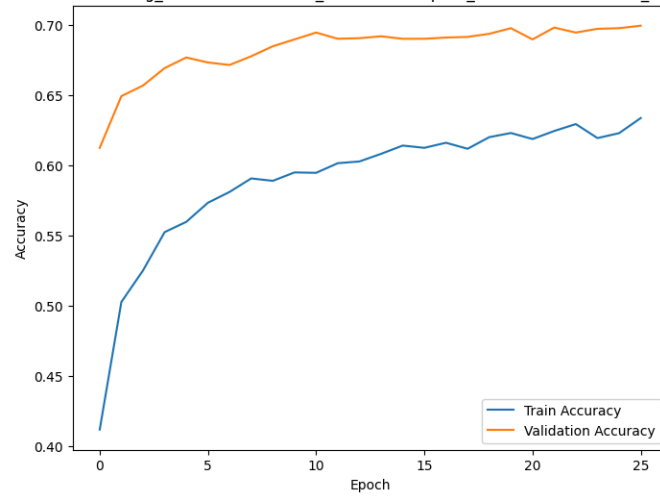
Configuration 3: {'optimizer': 'adamw', 'learning\_rate': 1e-05, 'batch\_size': 32, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': True}



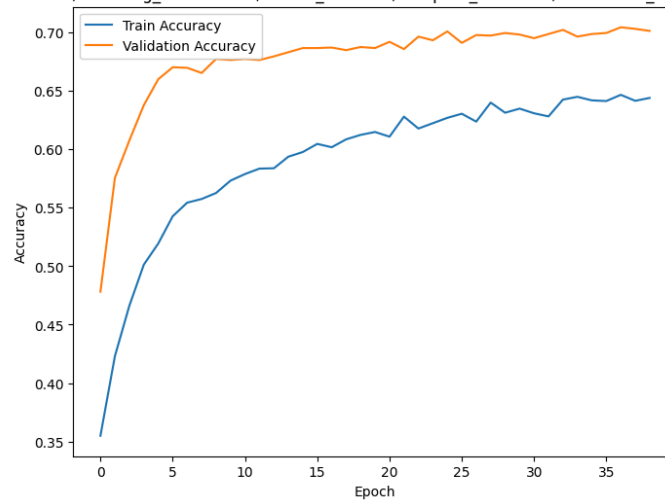
Configuration 4: {'optimizer': 'adam', 'learning\_rate': 3e-05, 'batch\_size': 32, 'dropout\_rate': 0.1, 'activation\_function': 'relu', 'use\_batchnorm': True}



Configuration 5: {'optimizer': 'adamw', 'learning\_rate': 1e-05, 'batch\_size': 16, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': True}

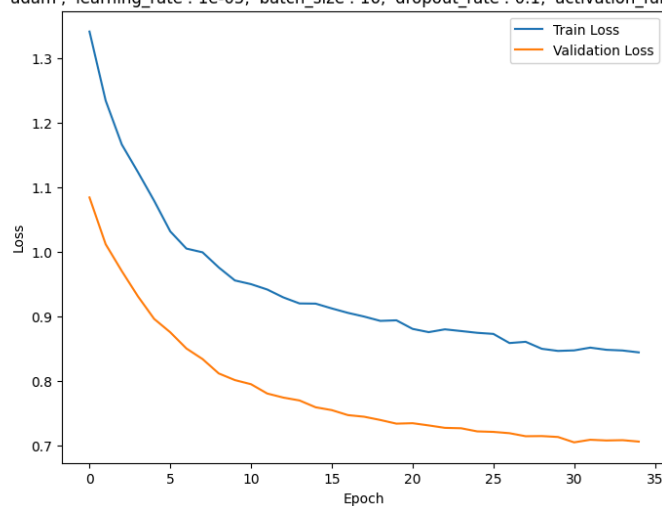


Configuration 6: {'optimizer': 'adam', 'learning\_rate': 3e-05, 'batch\_size': 16, 'dropout\_rate': 0.2, 'activation\_function': 'relu', 'use\_batchnorm': True}

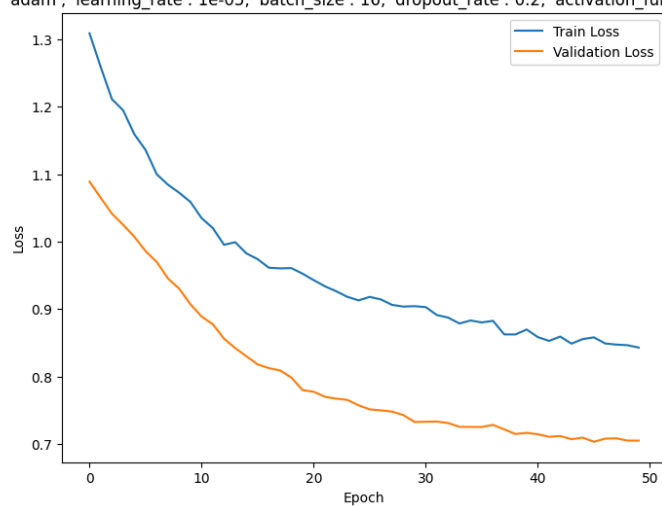


```
In [ ]: # Plot training and validation loss for each configuration
for i, (history, config) in enumerate(histories):
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title(f"Configuration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

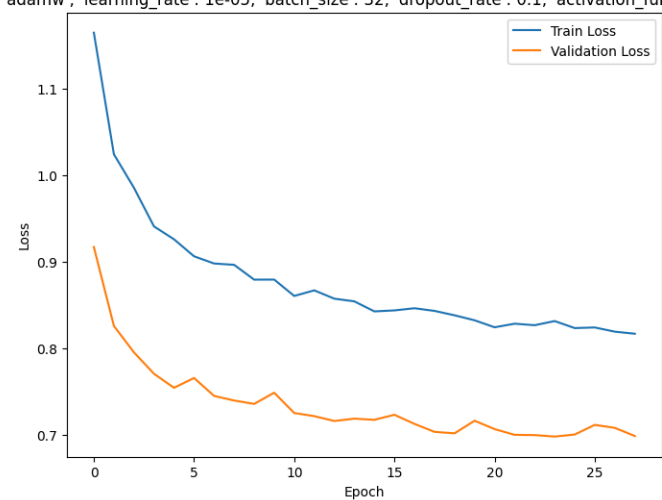
Configuration 1: {'optimizer': 'adam', 'learning\_rate': 1e-05, 'batch\_size': 16, 'dropout\_rate': 0.1, 'activation\_function': 'relu', 'use\_batchnorm': True}



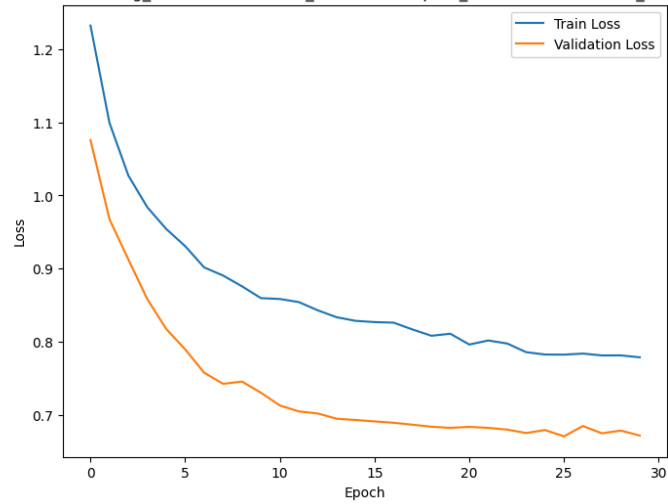
Configuration 2: {'optimizer': 'adam', 'learning\_rate': 1e-05, 'batch\_size': 16, 'dropout\_rate': 0.2, 'activation\_function': 'relu', 'use\_batchnorm': True}



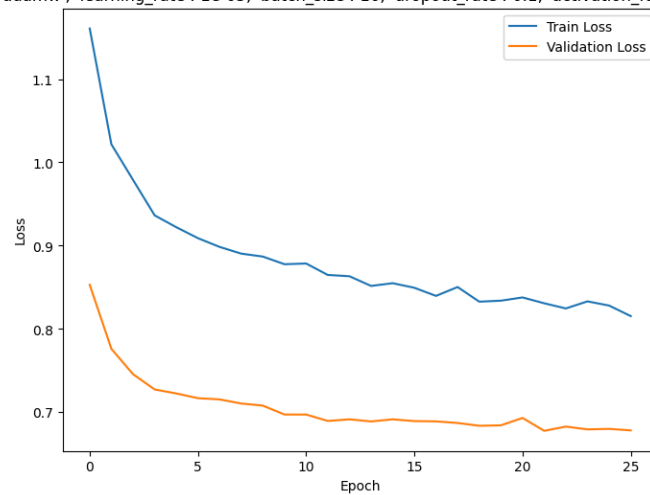
Configuration 3: {'optimizer': 'adamw', 'learning\_rate': 1e-05, 'batch\_size': 32, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': True}



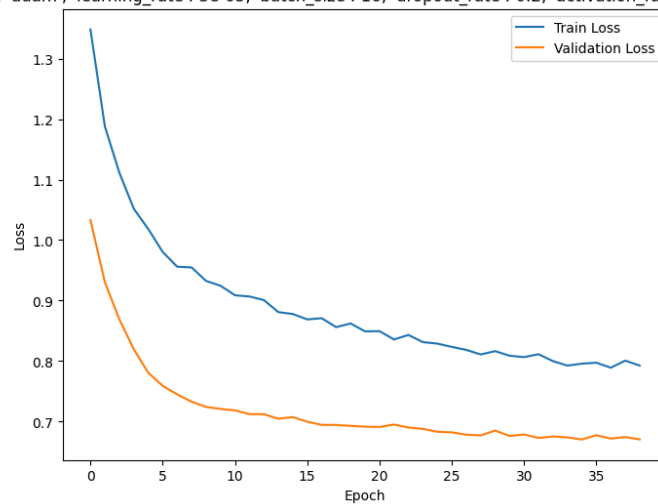
Configuration 4: {'optimizer': 'adam', 'learning\_rate': 3e-05, 'batch\_size': 32, 'dropout\_rate': 0.1, 'activation\_function': 'relu', 'use\_batchnorm': True}



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=learning_rate)
        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 [ ]: # 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.layer\_norm.bias', 'lm\_head.layer\_norm.weight', 'lm\_head.dense.bias']

- This IS expected if you are initializing TFRobertaModel from a PyTorch model 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 TFBertForSequenceClassification 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 training.

```

In [ ]: 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', 'Positive'])
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=['Negative', 'Positive'], yticklabels=['Actual', 'Predicted'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
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=['Negative', 'Positive'])
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, best_cm, best_report

```

### 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\_batchnorm': False}

Epoch 1/80  
657/657 [=====] - 73s 101ms/step - loss: 1.0993 - accuracy: 0.3907 - val\_loss: 0.9956 - val\_accuracy: 0.5182

Epoch 2/80  
657/657 [=====] - 65s 99ms/step - loss: 0.9773 - accuracy: 0.5142 - val\_loss: 0.8534 - val\_accuracy: 0.6067

Epoch 3/80  
657/657 [=====] - 64s 98ms/step - loss: 0.8854 - accuracy: 0.5789 - val\_loss: 0.7991 - val\_accuracy: 0.6120

Epoch 4/80  
657/657 [=====] - 64s 98ms/step - loss: 0.8366 - accuracy: 0.6171 - val\_loss: 0.7604 - val\_accuracy: 0.6422

Epoch 5/80  
657/657 [=====] - 65s 98ms/step - loss: 0.8206 - accuracy: 0.6250 - val\_loss: 0.7507 - val\_accuracy: 0.6476

Epoch 6/80  
657/657 [=====] - 64s 98ms/step - loss: 0.8045 - accuracy: 0.6375 - val\_loss: 0.7394 - val\_accuracy: 0.6542

Epoch 7/80  
657/657 [=====] - 65s 98ms/step - loss: 0.7951 - accuracy: 0.6441 - val\_loss: 0.7198 - val\_accuracy: 0.6671

Epoch 8/80  
657/657 [=====] - 64s 97ms/step - loss: 0.7969 - accuracy: 0.6417 - val\_loss: 0.7224 - val\_accuracy: 0.6698

Epoch 9/80  
657/657 [=====] - 65s 98ms/step - loss: 0.7824 - accuracy: 0.6493 - val\_loss: 0.7193 - val\_accuracy: 0.6662

Epoch 10/80  
657/657 [=====] - 65s 98ms/step - loss: 0.7811 - accuracy: 0.6508 - val\_loss: 0.7222 - val\_accuracy: 0.6680

Epoch 11/80  
657/657 [=====] - 65s 98ms/step - loss: 0.7776 - accuracy: 0.6549 - val\_loss: 0.7002 - val\_accuracy: 0.6853

Epoch 12/80  
657/657 [=====] - 65s 99ms/step - loss: 0.7664 - accuracy: 0.6570 - val\_loss: 0.7019 - val\_accuracy: 0.6849

Epoch 13/80  
657/657 [=====] - 65s 98ms/step - loss: 0.7627 - accuracy: 0.6632 - val\_loss: 0.7052 - val\_accuracy: 0.6773

Epoch 14/80  
657/657 [=====] - 65s 98ms/step - loss: 0.7631 - accuracy: 0.6637 - val\_loss: 0.6993 - val\_accuracy: 0.6853

Epoch 15/80  
657/657 [=====] - 65s 99ms/step - loss: 0.7620 - accuracy: 0.6630 - val\_loss: 0.6938 - val\_accuracy: 0.6867

Epoch 16/80  
657/657 [=====] - 66s 100ms/step - loss: 0.7544 - accuracy: 0.6654 - val\_loss: 0.6934 - val\_accuracy: 0.6844

Epoch 17/80  
657/657 [=====] - 65s 98ms/step - loss: 0.7601 - accuracy: 0.6652 - val\_loss: 0.6895 - val\_accuracy: 0.6933

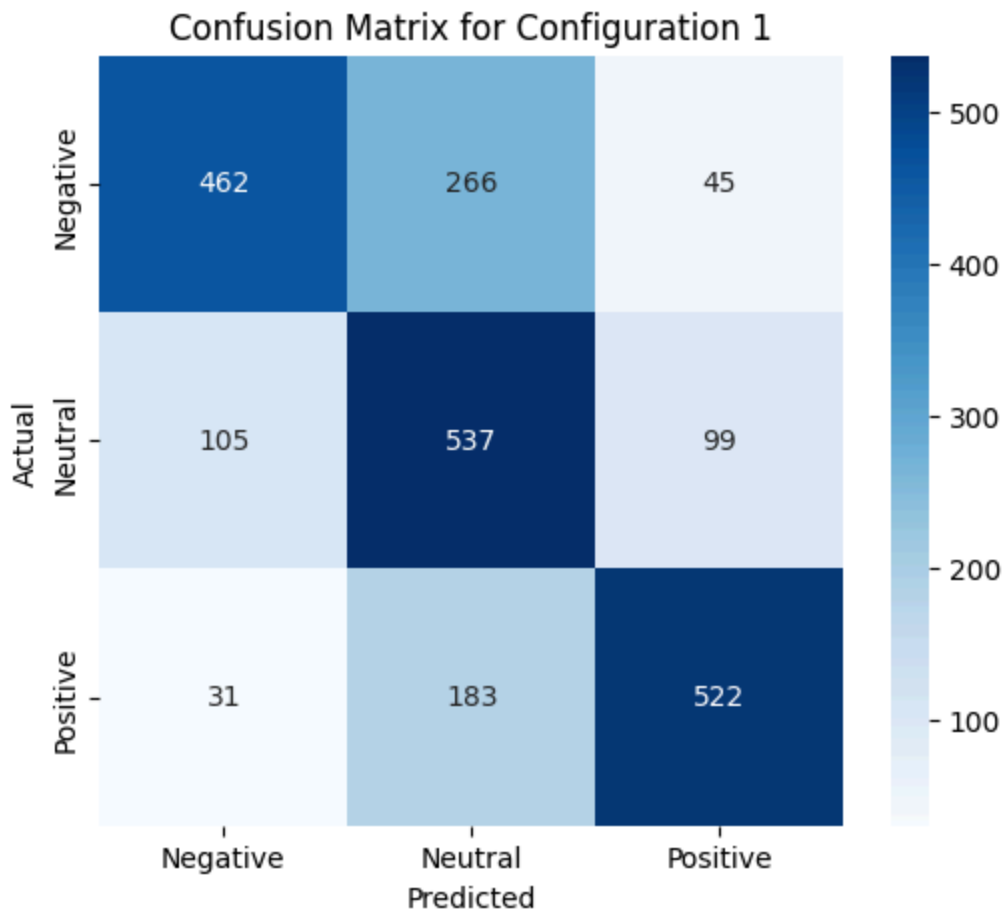
Epoch 18/80  
657/657 [=====] - 64s 98ms/step - loss: 0.7556 - accuracy: 0.6654 - val\_loss: 0.6895 - val\_accuracy: 0.6933

```

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\_batchnorm': False}

Epoch 1/80

329/329 [=====] - 72s 194ms/step - loss: 1.1209 - accuracy: 0.3676 - val\_loss: 1.0282 - val\_accuracy: 0.5738

Epoch 2/80

329/329 [=====] - 60s 184ms/step - loss: 1.0380 - accuracy: 0.4625 - val\_loss: 0.9283 - val\_accuracy: 0.5907

Epoch 3/80

329/329 [=====] - 61s 186ms/step - loss: 0.9422 - accuracy: 0.5429 - val\_loss: 0.8324 - val\_accuracy: 0.6298

Epoch 4/80

329/329 [=====] - 60s 182ms/step - loss: 0.8771 - accuracy: 0.5938 - val\_loss: 0.7770 - val\_accuracy: 0.6453

Epoch 5/80

329/329 [=====] - 61s 184ms/step - loss: 0.8437 - accuracy: 0.6037 - val\_loss: 0.7508 - val\_accuracy: 0.6631

Epoch 6/80

329/329 [=====] - 60s 183ms/step - loss: 0.8198 - accuracy: 0.6279 - val\_loss: 0.7322 - val\_accuracy: 0.6764

Epoch 7/80

329/329 [=====] - 61s 186ms/step - loss: 0.8069 - accuracy: 0.6335 - val\_loss: 0.7208 - val\_accuracy: 0.6796

Epoch 8/80

329/329 [=====] - 61s 185ms/step - loss: 0.8039 - accuracy: 0.6338 - val\_loss: 0.7129 - val\_accuracy: 0.6813

Epoch 9/80

329/329 [=====] - 60s 183ms/step - loss: 0.7923 - accuracy: 0.6405 - val\_loss: 0.7175 - val\_accuracy: 0.6778

Epoch 10/80

329/329 [=====] - 60s 183ms/step - loss: 0.7834 - accuracy: 0.6409 - val\_loss: 0.7073 - val\_accuracy: 0.6884

Epoch 11/80

329/329 [=====] - 62s 187ms/step - loss: 0.7881 - accuracy: 0.6470 - val\_loss: 0.7088 - val\_accuracy: 0.6849

Epoch 12/80

329/329 [=====] - 60s 183ms/step - loss: 0.7698 - accuracy: 0.6571 - val\_loss: 0.7033 - val\_accuracy: 0.6880

Epoch 13/80

329/329 [=====] - 62s 187ms/step - loss: 0.7778 - accuracy: 0.6505 - val\_loss: 0.7005 - val\_accuracy: 0.6911

Epoch 14/80

329/329 [=====] - 60s 183ms/step - loss: 0.7690 - accuracy: 0.6579 - val\_loss: 0.6930 - val\_accuracy: 0.6951

Epoch 15/80

329/329 [=====] - 61s 184ms/step - loss: 0.7641 - accuracy: 0.6619 - val\_loss: 0.6933 - val\_accuracy: 0.6933

Epoch 16/80

329/329 [=====] - 61s 185ms/step - loss: 0.7692 - accuracy: 0.6568 - val\_loss: 0.6938 - val\_accuracy: 0.6933

Epoch 17/80

329/329 [=====] - 61s 185ms/step - loss: 0.7666 - accuracy: 0.6602 - val\_loss: 0.6976 - val\_accuracy: 0.6933

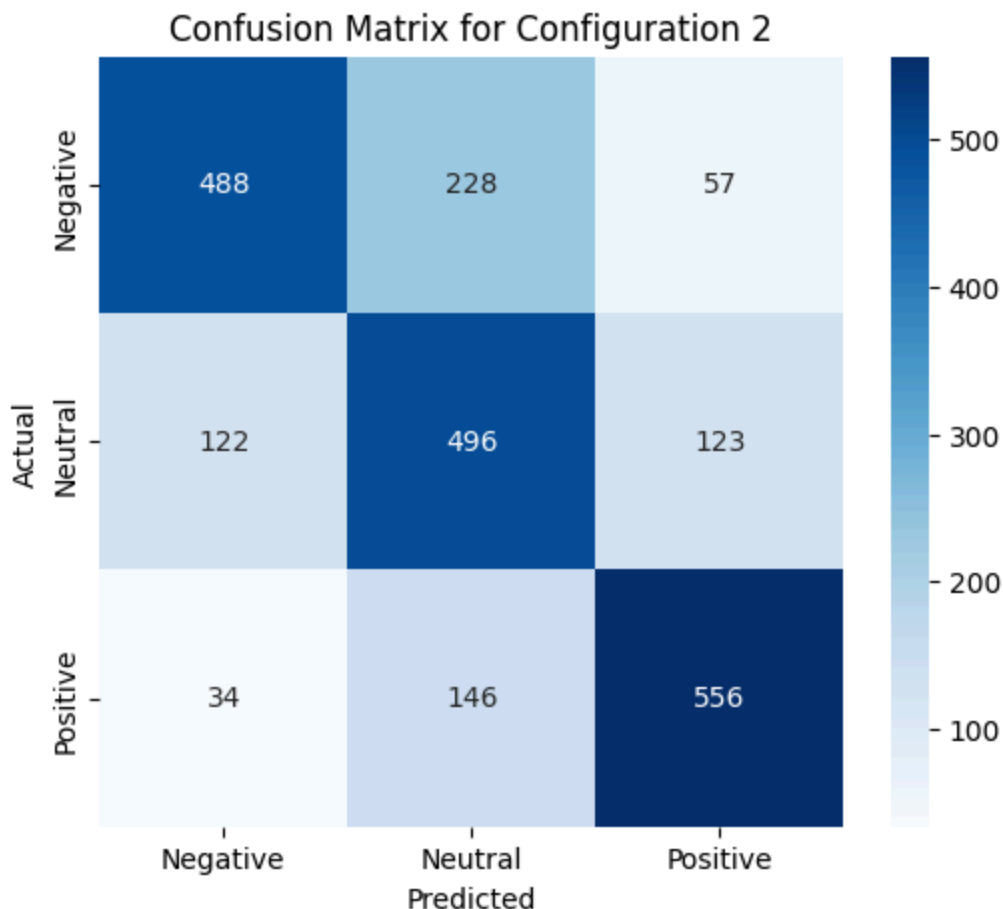
Epoch 18/80

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

Epoch 37/80  
 329/329 [=====] - 60s 183ms/step - loss: 0.7415 - accuracy: 0.6709 - val\_loss: 0.6751 - val\_accuracy: 0.7013  
 Epoch 38/80  
 329/329 [=====] - 61s 185ms/step - loss: 0.7388 - accuracy: 0.6723 - val\_loss: 0.6742 - val\_accuracy: 0.6982  
 Epoch 39/80  
 329/329 [=====] - 60s 182ms/step - loss: 0.7421 - accuracy: 0.6699 - val\_loss: 0.6779 - val\_accuracy: 0.6960  
 Epoch 40/80  
 329/329 [=====] - 60s 183ms/step - loss: 0.7393 - accuracy: 0.6774 - val\_loss: 0.6749 - val\_accuracy: 0.7013  
 71/71 [=====] - 10s 138ms/step - loss: 0.7090 - accuracy: 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\_batchnorm': False}

Epoch 1/80

329/329 [=====] - 73s 193ms/step - loss: 1.1361 - accuracy: 0.3505 - val\_loss: 1.0738 - val\_accuracy: 0.4000

Epoch 2/80

329/329 [=====] - 62s 187ms/step - loss: 1.1099 - accuracy: 0.3741 - val\_loss: 1.0474 - val\_accuracy: 0.5320

Epoch 3/80

329/329 [=====] - 60s 183ms/step - loss: 1.0849 - accuracy: 0.4051 - val\_loss: 1.0149 - val\_accuracy: 0.5960

Epoch 4/80

329/329 [=====] - 60s 183ms/step - loss: 1.0496 - accuracy: 0.4467 - val\_loss: 0.9792 - val\_accuracy: 0.6044

Epoch 5/80

329/329 [=====] - 60s 184ms/step - loss: 1.0173 - accuracy: 0.4775 - val\_loss: 0.9390 - val\_accuracy: 0.6227

Epoch 6/80

329/329 [=====] - 60s 184ms/step - loss: 0.9832 - accuracy: 0.5094 - val\_loss: 0.9004 - val\_accuracy: 0.6302

Epoch 7/80

329/329 [=====] - 61s 185ms/step - loss: 0.9536 - accuracy: 0.5308 - val\_loss: 0.8660 - val\_accuracy: 0.6396

Epoch 8/80

329/329 [=====] - 61s 186ms/step - loss: 0.9247 - accuracy: 0.5549 - val\_loss: 0.8363 - val\_accuracy: 0.6422

Epoch 9/80

329/329 [=====] - 61s 186ms/step - loss: 0.9010 - accuracy: 0.5650 - val\_loss: 0.8120 - val\_accuracy: 0.6462

Epoch 10/80

329/329 [=====] - 61s 186ms/step - loss: 0.8818 - accuracy: 0.5771 - val\_loss: 0.7985 - val\_accuracy: 0.6462

Epoch 11/80

329/329 [=====] - 61s 186ms/step - loss: 0.8631 - accuracy: 0.5956 - val\_loss: 0.7805 - val\_accuracy: 0.6533

Epoch 12/80

329/329 [=====] - 60s 183ms/step - loss: 0.8477 - accuracy: 0.6029 - val\_loss: 0.7685 - val\_accuracy: 0.6560

Epoch 13/80

329/329 [=====] - 61s 186ms/step - loss: 0.8468 - accuracy: 0.6103 - val\_loss: 0.7598 - val\_accuracy: 0.6622

Epoch 14/80

329/329 [=====] - 61s 186ms/step - loss: 0.8340 - accuracy: 0.6140 - val\_loss: 0.7503 - val\_accuracy: 0.6649

Epoch 15/80

329/329 [=====] - 60s 183ms/step - loss: 0.8351 - accuracy: 0.6130 - val\_loss: 0.7435 - val\_accuracy: 0.6689

Epoch 16/80

329/329 [=====] - 60s 183ms/step - loss: 0.8267 - accuracy: 0.6209 - val\_loss: 0.7404 - val\_accuracy: 0.6684

Epoch 17/80

329/329 [=====] - 60s 183ms/step - loss: 0.8197 - accuracy: 0.6273 - val\_loss: 0.7397 - val\_accuracy: 0.6707

Epoch 18/80



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

Epoch 37/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7680 - accuracy: 0.6582 - val\_loss: 0.6961 - val\_accuracy: 0.6907  
Epoch 38/80  
329/329 [=====] - 60s 184ms/step - loss: 0.7687 - accuracy: 0.6596 - val\_loss: 0.6949 - val\_accuracy: 0.6920  
Epoch 39/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7742 - accuracy: 0.6533 - val\_loss: 0.6936 - val\_accuracy: 0.6933  
Epoch 40/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7707 - accuracy: 0.6583 - val\_loss: 0.6950 - val\_accuracy: 0.6951  
Epoch 41/80  
329/329 [=====] - 60s 184ms/step - loss: 0.7708 - accuracy: 0.6521 - val\_loss: 0.6932 - val\_accuracy: 0.6920  
Epoch 42/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7722 - accuracy: 0.6561 - val\_loss: 0.6937 - val\_accuracy: 0.6938  
Epoch 43/80  
329/329 [=====] - 61s 186ms/step - loss: 0.7690 - accuracy: 0.6592 - val\_loss: 0.6924 - val\_accuracy: 0.6964  
Epoch 44/80  
329/329 [=====] - 61s 185ms/step - loss: 0.7671 - accuracy: 0.6608 - val\_loss: 0.6942 - val\_accuracy: 0.6942  
Epoch 45/80  
329/329 [=====] - 61s 184ms/step - loss: 0.7627 - accuracy: 0.6572 - val\_loss: 0.6927 - val\_accuracy: 0.6933  
Epoch 46/80  
329/329 [=====] - 61s 186ms/step - loss: 0.7606 - accuracy: 0.6599 - val\_loss: 0.6896 - val\_accuracy: 0.6960  
Epoch 47/80  
329/329 [=====] - 60s 182ms/step - loss: 0.7632 - accuracy: 0.6594 - val\_loss: 0.6922 - val\_accuracy: 0.6938  
Epoch 48/80  
329/329 [=====] - 61s 187ms/step - loss: 0.7617 - accuracy: 0.6627 - val\_loss: 0.6901 - val\_accuracy: 0.6960  
Epoch 49/80  
329/329 [=====] - 61s 185ms/step - loss: 0.7638 - accuracy: 0.6605 - val\_loss: 0.6877 - val\_accuracy: 0.6960  
Epoch 50/80  
329/329 [=====] - 61s 186ms/step - loss: 0.7611 - accuracy: 0.6611 - val\_loss: 0.6879 - val\_accuracy: 0.6973  
Epoch 51/80  
329/329 [=====] - 61s 186ms/step - loss: 0.7566 - accuracy: 0.6668 - val\_loss: 0.6865 - val\_accuracy: 0.6938  
Epoch 52/80  
329/329 [=====] - 61s 186ms/step - loss: 0.7543 - accuracy: 0.6674 - val\_loss: 0.6868 - val\_accuracy: 0.6951  
Epoch 53/80  
329/329 [=====] - 61s 185ms/step - loss: 0.7556 - accuracy: 0.6629 - val\_loss: 0.6886 - val\_accuracy: 0.6982  
Epoch 54/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7520 - accuracy: 0.6677 - val\_loss: 0.6861 - val\_accuracy: 0.6969  
Epoch 55/80  
329/329 [=====] - 61s 186ms/step - loss: 0.7600 - accuracy: 0.6600 - val\_loss: 0.6861 - val\_accuracy: 0.6969

```

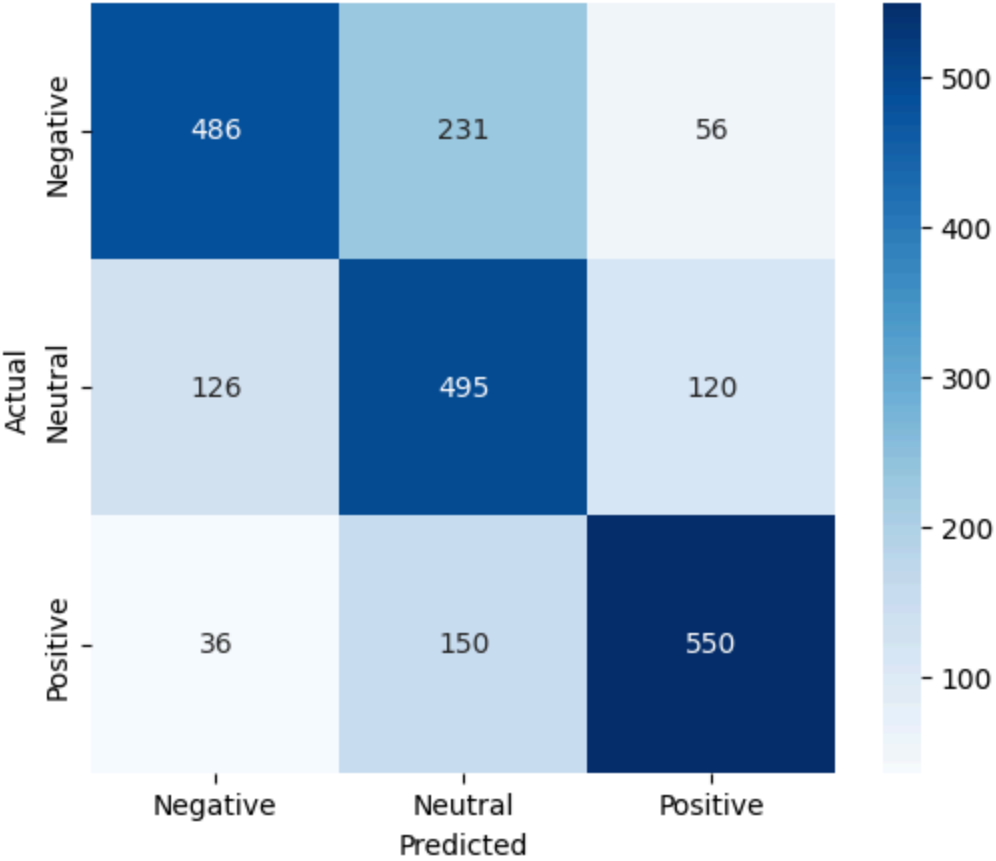
ccuracy: 0.6619 - val_loss: 0.6853 - val_accuracy: 0.6964
Epoch 56/80
329/329 [=====] - 61s 185ms/step - loss: 0.7530 - a
ccuracy: 0.6676 - val_loss: 0.6831 - val_accuracy: 0.6947
Epoch 57/80
329/329 [=====] - 61s 187ms/step - loss: 0.7577 - a
ccuracy: 0.6670 - val_loss: 0.6821 - val_accuracy: 0.6969
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



Training configuration 4/6: {'optimizer': 'adam', 'learning\_rate': 3e-05, 'batch\_size': 32, 'dropout\_rate': 0.2, 'activation\_function': 'relu', 'use\_batchnorm': False}

Epoch 1/80  
329/329 [=====] - 69s 191ms/step - loss: 1.1155 - accuracy: 0.3318 - val\_loss: 1.0964 - val\_accuracy: 0.3342

Epoch 2/80  
329/329 [=====] - 60s 183ms/step - loss: 1.1025 - accuracy: 0.3490 - val\_loss: 1.0937 - val\_accuracy: 0.3324

Epoch 3/80  
329/329 [=====] - 62s 187ms/step - loss: 1.0987 - accuracy: 0.3498 - val\_loss: 1.0883 - val\_accuracy: 0.3840

Epoch 4/80  
329/329 [=====] - 60s 183ms/step - loss: 1.0907 - accuracy: 0.3711 - val\_loss: 1.0680 - val\_accuracy: 0.4951

Epoch 5/80  
329/329 [=====] - 61s 186ms/step - loss: 1.0743 - accuracy: 0.4113 - val\_loss: 1.0329 - val\_accuracy: 0.5276

Epoch 6/80  
329/329 [=====] - 60s 183ms/step - loss: 1.0440 - accuracy: 0.4538 - val\_loss: 0.9725 - val\_accuracy: 0.5800

Epoch 7/80  
329/329 [=====] - 61s 184ms/step - loss: 1.0057 - accuracy: 0.4925 - val\_loss: 0.9113 - val\_accuracy: 0.5809

Epoch 8/80  
329/329 [=====] - 62s 187ms/step - loss: 0.9552 - accuracy: 0.5353 - val\_loss: 0.8570 - val\_accuracy: 0.6058

Epoch 9/80  
329/329 [=====] - 61s 185ms/step - loss: 0.9235 - accuracy: 0.5483 - val\_loss: 0.8266 - val\_accuracy: 0.6249

Epoch 10/80  
329/329 [=====] - 61s 187ms/step - loss: 0.9071 - accuracy: 0.5683 - val\_loss: 0.8044 - val\_accuracy: 0.6311

Epoch 11/80  
329/329 [=====] - 60s 183ms/step - loss: 0.8834 - accuracy: 0.5806 - val\_loss: 0.7896 - val\_accuracy: 0.6382

Epoch 12/80  
329/329 [=====] - 61s 184ms/step - loss: 0.8752 - accuracy: 0.5900 - val\_loss: 0.7806 - val\_accuracy: 0.6436

Epoch 13/80  
329/329 [=====] - 61s 184ms/step - loss: 0.8619 - accuracy: 0.5988 - val\_loss: 0.7704 - val\_accuracy: 0.6480

Epoch 14/80  
329/329 [=====] - 60s 181ms/step - loss: 0.8536 - accuracy: 0.6059 - val\_loss: 0.7618 - val\_accuracy: 0.6569

Epoch 15/80  
329/329 [=====] - 60s 183ms/step - loss: 0.8500 - accuracy: 0.6100 - val\_loss: 0.7476 - val\_accuracy: 0.6622

Epoch 16/80  
329/329 [=====] - 59s 180ms/step - loss: 0.8396 - accuracy: 0.6161 - val\_loss: 0.7480 - val\_accuracy: 0.6676

Epoch 17/80  
329/329 [=====] - 60s 184ms/step - loss: 0.8270 - accuracy: 0.6250 - val\_loss: 0.7364 - val\_accuracy: 0.6733

Epoch 18/80  
329/329 [=====] - 61s 185ms/step - loss: 0.8216 - accuracy: 0.6311 - val\_loss: 0.7250 - val\_accuracy: 0.6780

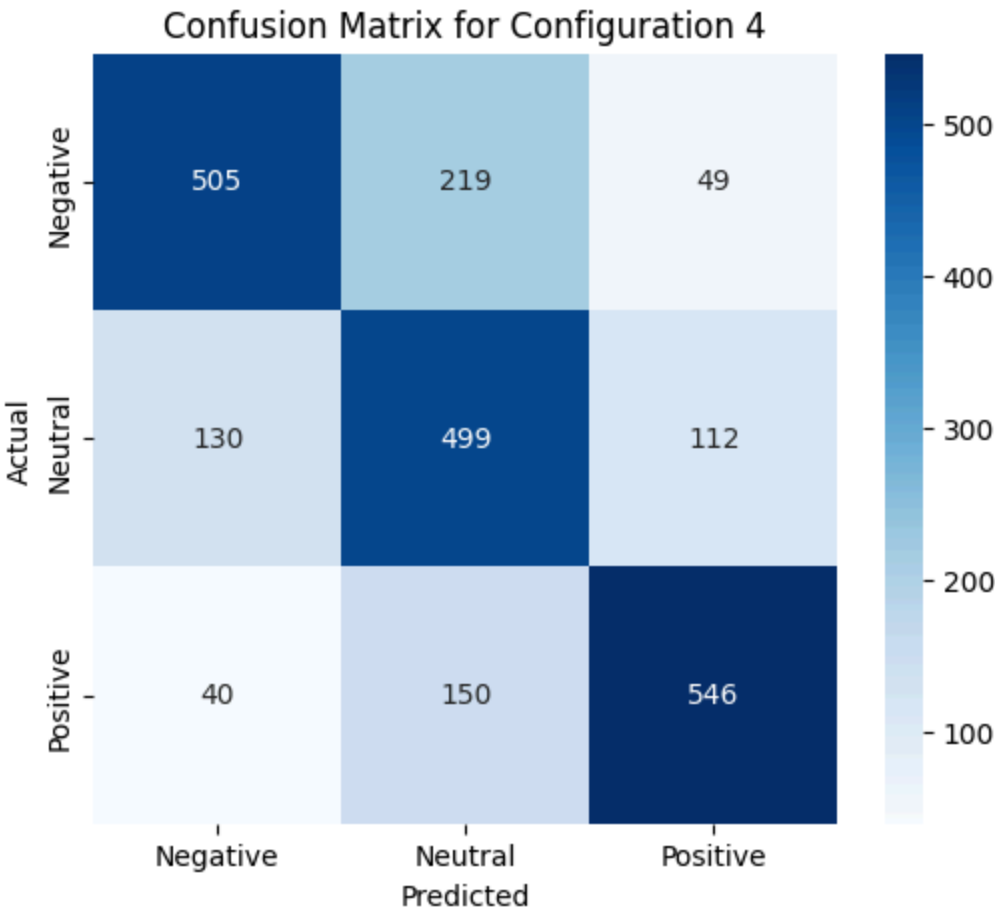
ccuracy: 0.6270 - val\_loss: 0.7375 - val\_accuracy: 0.6742  
Epoch 19/80  
329/329 [=====] - 60s 183ms/step - loss: 0.8101 - a  
ccuracy: 0.6372 - val\_loss: 0.7276 - val\_accuracy: 0.6733  
Epoch 20/80  
329/329 [=====] - 60s 183ms/step - loss: 0.8089 - a  
ccuracy: 0.6321 - val\_loss: 0.7247 - val\_accuracy: 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 - accuracy: 0.6647 - val\_loss: 0.6881 - val\_accuracy: 0.6920  
Epoch 38/80  
329/329 [=====] - 60s 182ms/step - loss: 0.7560 - accuracy: 0.6691 - val\_loss: 0.6842 - val\_accuracy: 0.6938  
Epoch 39/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7549 - accuracy: 0.6688 - val\_loss: 0.6824 - val\_accuracy: 0.6920  
Epoch 40/80  
329/329 [=====] - 60s 182ms/step - loss: 0.7461 - accuracy: 0.6697 - val\_loss: 0.6817 - val\_accuracy: 0.6942  
Epoch 41/80  
329/329 [=====] - 60s 182ms/step - loss: 0.7571 - accuracy: 0.6699 - val\_loss: 0.6803 - val\_accuracy: 0.6929  
Epoch 42/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7515 - accuracy: 0.6698 - val\_loss: 0.6808 - val\_accuracy: 0.6942  
Epoch 43/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7481 - accuracy: 0.6696 - val\_loss: 0.6800 - val\_accuracy: 0.6911  
Epoch 44/80  
329/329 [=====] - 61s 185ms/step - loss: 0.7524 - accuracy: 0.6659 - val\_loss: 0.6784 - val\_accuracy: 0.6942  
Epoch 45/80  
329/329 [=====] - 59s 179ms/step - loss: 0.7455 - accuracy: 0.6678 - val\_loss: 0.6802 - val\_accuracy: 0.6916  
Epoch 46/80  
329/329 [=====] - 59s 180ms/step - loss: 0.7462 - accuracy: 0.6706 - val\_loss: 0.6801 - val\_accuracy: 0.6924  
Epoch 47/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7423 - accuracy: 0.6724 - val\_loss: 0.6778 - val\_accuracy: 0.6933  
Epoch 48/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7449 - accuracy: 0.6716 - val\_loss: 0.6793 - val\_accuracy: 0.6933  
Epoch 49/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7469 - accuracy: 0.6712 - val\_loss: 0.6767 - val\_accuracy: 0.6951  
Epoch 50/80  
329/329 [=====] - 60s 184ms/step - loss: 0.7450 - accuracy: 0.6673 - val\_loss: 0.6740 - val\_accuracy: 0.6991  
Epoch 51/80  
329/329 [=====] - 61s 185ms/step - loss: 0.7447 - accuracy: 0.6663 - val\_loss: 0.6751 - val\_accuracy: 0.6942  
Epoch 52/80  
329/329 [=====] - 60s 184ms/step - loss: 0.7434 - accuracy: 0.6723 - val\_loss: 0.6757 - val\_accuracy: 0.6956  
Epoch 53/80  
329/329 [=====] - 61s 184ms/step - loss: 0.7439 - accuracy: 0.6753 - val\_loss: 0.6743 - val\_accuracy: 0.6956  
Epoch 54/80  
329/329 [=====] - 60s 183ms/step - loss: 0.7377 - accuracy: 0.6696 - val\_loss: 0.6755 - val\_accuracy: 0.6951  
71/71 [=====] - 10s 141ms/step - loss: 0.7105 - accuracy: 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, 'batch\_size': 16, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': False}

Epoch 1/80

657/657 [=====] - 74s 102ms/step - loss: 1.1318 - accuracy: 0.3467 - val\_loss: 1.0639 - val\_accuracy: 0.4324

Epoch 2/80

657/657 [=====] - 65s 99ms/step - loss: 1.0935 - accuracy: 0.3970 - val\_loss: 1.0220 - val\_accuracy: 0.5209

Epoch 3/80

657/657 [=====] - 64s 98ms/step - loss: 1.0479 - accuracy: 0.4506 - val\_loss: 0.9673 - val\_accuracy: 0.6080

Epoch 4/80

657/657 [=====] - 65s 98ms/step - loss: 1.0032 - accuracy: 0.4947 - val\_loss: 0.9150 - val\_accuracy: 0.6133

Epoch 5/80

657/657 [=====] - 65s 99ms/step - loss: 0.9579 - accuracy: 0.5386 - val\_loss: 0.8674 - val\_accuracy: 0.6316

Epoch 6/80

657/657 [=====] - 65s 98ms/step - loss: 0.9182 - accuracy: 0.5549 - val\_loss: 0.8318 - val\_accuracy: 0.6320

Epoch 7/80

657/657 [=====] - 65s 99ms/step - loss: 0.8878 - accuracy: 0.5826 - val\_loss: 0.8031 - val\_accuracy: 0.6404

Epoch 8/80

657/657 [=====] - 65s 99ms/step - loss: 0.8721 - accuracy: 0.5877 - val\_loss: 0.7815 - val\_accuracy: 0.6516

Epoch 9/80

657/657 [=====] - 64s 98ms/step - loss: 0.8611 - accuracy: 0.5963 - val\_loss: 0.7684 - val\_accuracy: 0.6520

Epoch 10/80

657/657 [=====] - 65s 98ms/step - loss: 0.8403 - accuracy: 0.6081 - val\_loss: 0.7611 - val\_accuracy: 0.6551

Epoch 11/80

657/657 [=====] - 65s 99ms/step - loss: 0.8322 - accuracy: 0.6164 - val\_loss: 0.7481 - val\_accuracy: 0.6622

Epoch 12/80

657/657 [=====] - 65s 99ms/step - loss: 0.8231 - accuracy: 0.6254 - val\_loss: 0.7434 - val\_accuracy: 0.6698

Epoch 13/80

657/657 [=====] - 64s 98ms/step - loss: 0.8150 - accuracy: 0.6304 - val\_loss: 0.7380 - val\_accuracy: 0.6676

Epoch 14/80

657/657 [=====] - 65s 98ms/step - loss: 0.8141 - accuracy: 0.6256 - val\_loss: 0.7280 - val\_accuracy: 0.6702

Epoch 15/80

657/657 [=====] - 65s 99ms/step - loss: 0.8037 - accuracy: 0.6350 - val\_loss: 0.7249 - val\_accuracy: 0.6733

Epoch 16/80

657/657 [=====] - 65s 99ms/step - loss: 0.8022 - accuracy: 0.6378 - val\_loss: 0.7227 - val\_accuracy: 0.6742

Epoch 17/80

657/657 [=====] - 65s 99ms/step - loss: 0.7997 - accuracy: 0.6364 - val\_loss: 0.7159 - val\_accuracy: 0.6711

Epoch 18/80

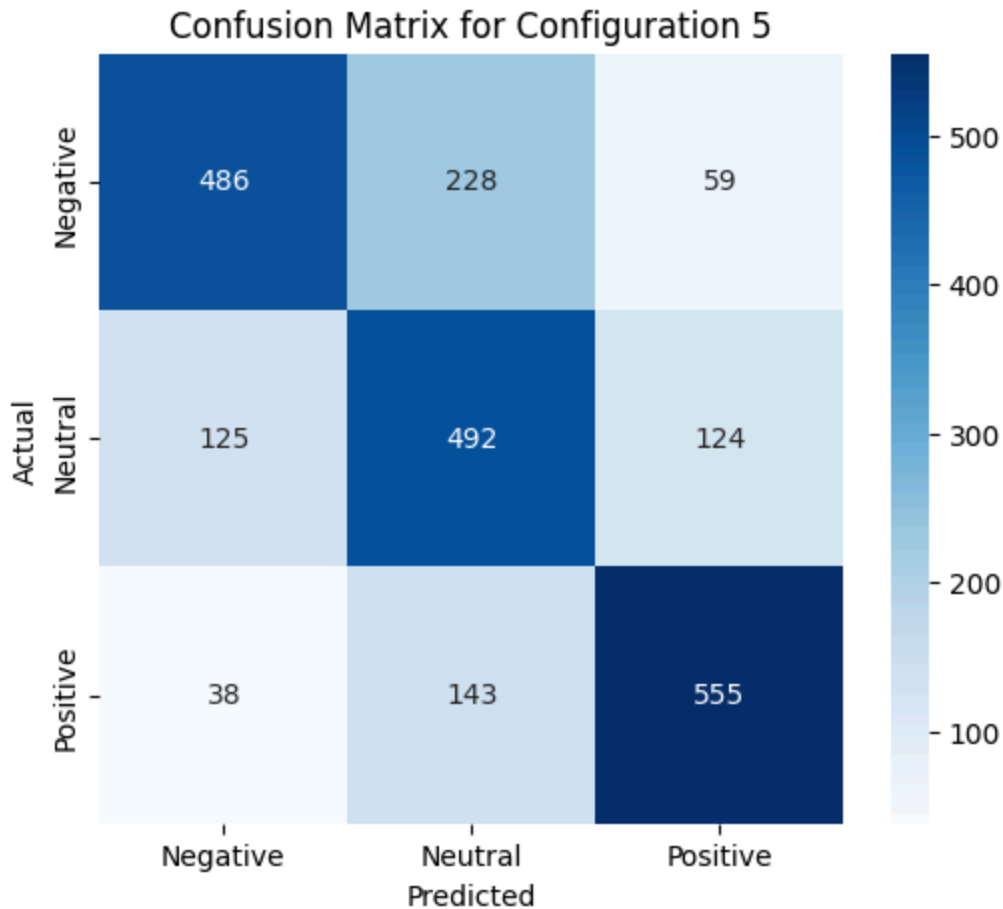
657/657 [=====] - 64s 98ms/step - loss: 0.7959 - accuracy: 0.6381 - val\_loss: 0.7161 - val\_accuracy: 0.6782  
Epoch 19/80  
657/657 [=====] - 65s 99ms/step - loss: 0.7888 - accuracy: 0.6425 - val\_loss: 0.7151 - val\_accuracy: 0.6791  
Epoch 20/80  
657/657 [=====] - 65s 98ms/step - loss: 0.7906 -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

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

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, 'batch\_size': 16, 'dropout\_rate': 0.2, 'activation\_function': 'relu', 'use\_batchnorm': False}

Epoch 1/80  
657/657 [=====] - 74s 102ms/step - loss: 1.1243 - accuracy: 0.3430 - val\_loss: 1.0926 - val\_accuracy: 0.3338

Epoch 2/80  
657/657 [=====] - 65s 98ms/step - loss: 1.0952 - accuracy: 0.3697 - val\_loss: 1.0708 - val\_accuracy: 0.4542

Epoch 3/80  
657/657 [=====] - 65s 99ms/step - loss: 1.0765 - accuracy: 0.4084 - val\_loss: 1.0314 - val\_accuracy: 0.5991

Epoch 4/80  
657/657 [=====] - 65s 99ms/step - loss: 1.0347 - accuracy: 0.4702 - val\_loss: 0.9296 - val\_accuracy: 0.6027

Epoch 5/80  
657/657 [=====] - 65s 99ms/step - loss: 0.9706 - accuracy: 0.5165 - val\_loss: 0.8441 - val\_accuracy: 0.6142

Epoch 6/80  
657/657 [=====] - 65s 99ms/step - loss: 0.9237 - accuracy: 0.5471 - val\_loss: 0.8108 - val\_accuracy: 0.6231

Epoch 7/80  
657/657 [=====] - 65s 99ms/step - loss: 0.8954 - accuracy: 0.5716 - val\_loss: 0.7948 - val\_accuracy: 0.6347

Epoch 8/80  
657/657 [=====] - 65s 99ms/step - loss: 0.8795 - accuracy: 0.5766 - val\_loss: 0.7734 - val\_accuracy: 0.6462

Epoch 9/80  
657/657 [=====] - 65s 99ms/step - loss: 0.8602 - accuracy: 0.5939 - val\_loss: 0.7602 - val\_accuracy: 0.6591

Epoch 10/80  
657/657 [=====] - 65s 99ms/step - loss: 0.8543 - accuracy: 0.5951 - val\_loss: 0.7535 - val\_accuracy: 0.6627

Epoch 11/80  
657/657 [=====] - 65s 99ms/step - loss: 0.8321 - accuracy: 0.6167 - val\_loss: 0.7391 - val\_accuracy: 0.6631

Epoch 12/80  
657/657 [=====] - 65s 99ms/step - loss: 0.8308 - accuracy: 0.6216 - val\_loss: 0.7370 - val\_accuracy: 0.6667

Epoch 13/80  
657/657 [=====] - 65s 99ms/step - loss: 0.8199 - accuracy: 0.6295 - val\_loss: 0.7322 - val\_accuracy: 0.6707

Epoch 14/80  
657/657 [=====] - 65s 99ms/step - loss: 0.8203 - accuracy: 0.6317 - val\_loss: 0.7274 - val\_accuracy: 0.6756

Epoch 15/80  
657/657 [=====] - 65s 98ms/step - loss: 0.8101 - accuracy: 0.6375 - val\_loss: 0.7192 - val\_accuracy: 0.6849

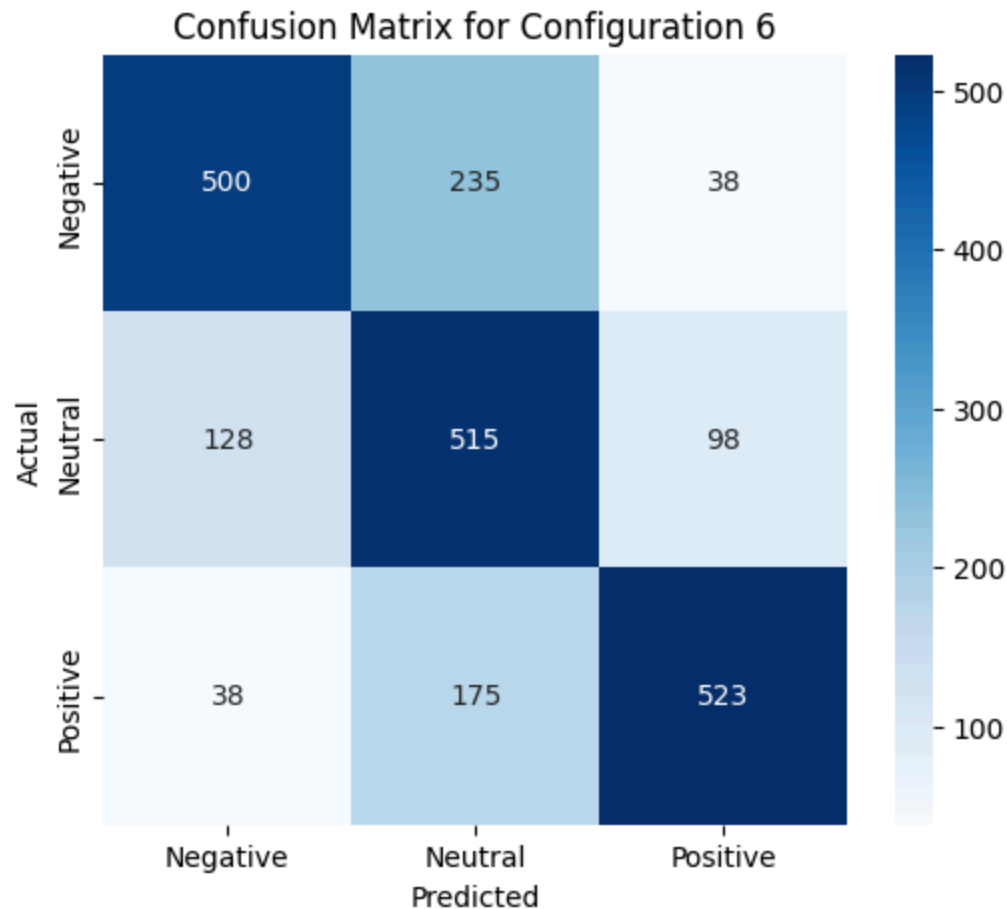
Epoch 16/80  
657/657 [=====] - 65s 99ms/step - loss: 0.8003 - accuracy: 0.6398 - val\_loss: 0.7176 - val\_accuracy: 0.6840

Epoch 17/80  
657/657 [=====] - 65s 99ms/step - loss: 0.8015 - accuracy: 0.6411 - val\_loss: 0.7122 - val\_accuracy: 0.6844

Epoch 18/80  
657/657 [=====] - 65s 99ms/step - loss: 0.7923 - accuracy: 0.6411 - val\_loss: 0.7122 - val\_accuracy: 0.6844

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_ids]

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, 'batch_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 mug

True Label: Neutral

Predicted Label: Positive

Text: i love gloria jeans cinnamon nut strudel coffee and decided to try this the flavoring is a little overpowering 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 cliff kid z bars i decided to try these based on all of the great reviews we bought the strawberry and no one in the house likes them my kids are picky so i always try the treats just to see if they are even worth trying i eat healthy and dont mind most healthy snacks but even i did not care for these i am going to try another flavor but the strawberry just didnt cut 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 make funnel cakes i had been using a kitchen funnel which made a big mess no matter 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 some 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 highly recommend this product

True Label: Positive

Predicted Label: Neutral

Text: i got a larger one than my puppy probably required and she loves dragging it around playing tug of war and just chewing on it unfortunately it does shed i dont know if there are rope toys out there that are any sturdier but this one does leave little threads behind otherwise great toy

True Label: Positive

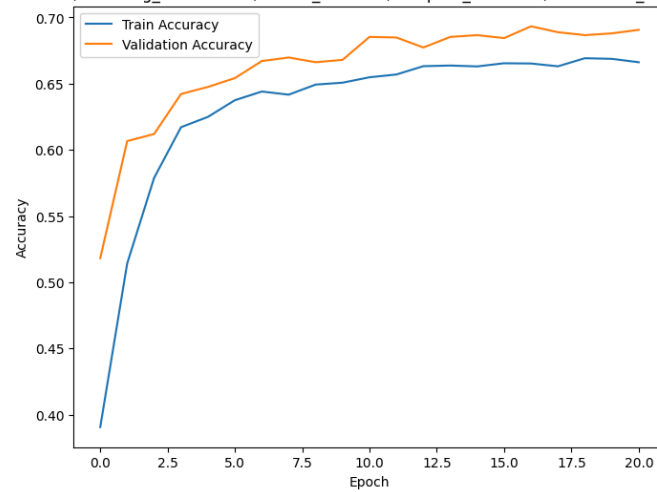
Predicted Label: Neutral

### 6.2.3 Non-Batch-Normalized Models: Visualizations

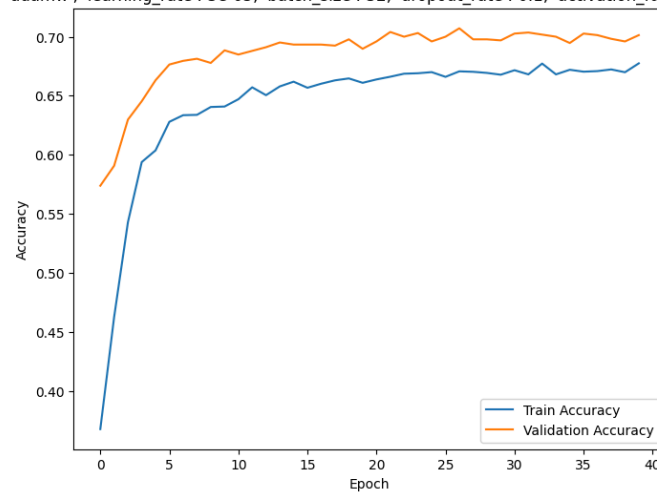
```
In [ ]: # Visualize the results
```

```
# 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()
```

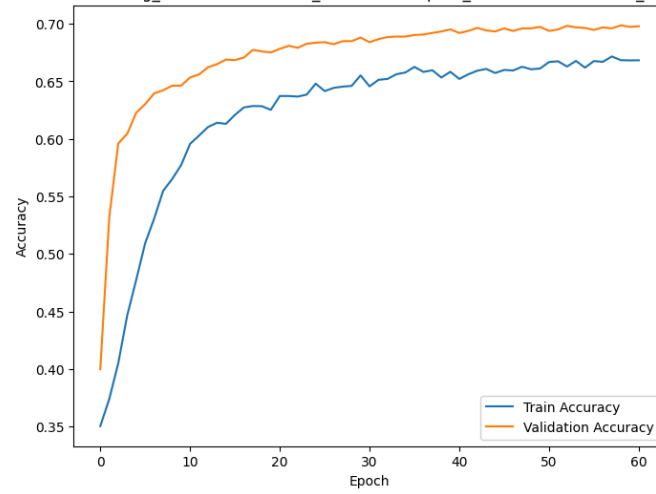
Configuration 1: {'optimizer': 'adamw', 'learning\_rate': 3e-05, 'batch\_size': 16, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': False}



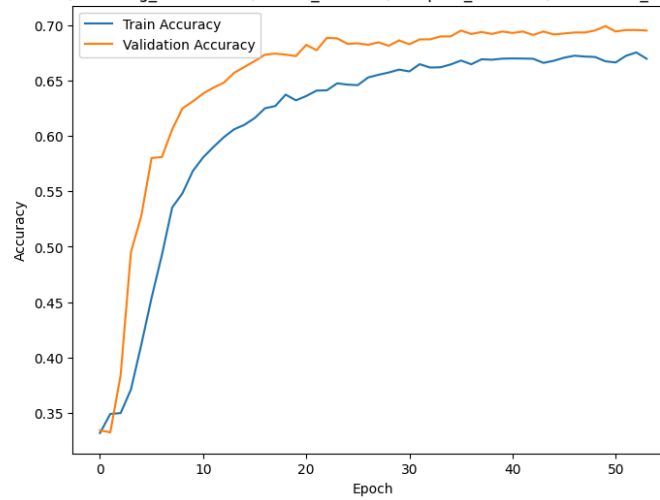
Configuration 2: {'optimizer': 'adamw', 'learning\_rate': 3e-05, 'batch\_size': 32, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': False}



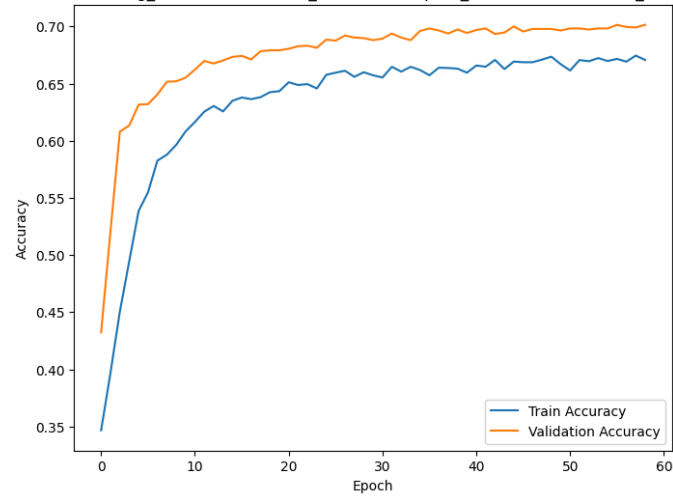
Configuration 3: {'optimizer': 'adamw', 'learning\_rate': 1e-05, 'batch\_size': 32, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': False}



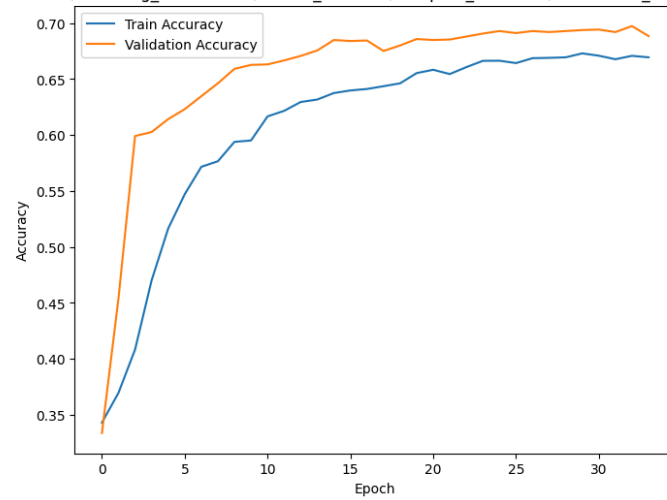
Configuration 4: {'optimizer': 'adam', 'learning\_rate': 3e-05, 'batch\_size': 32, 'dropout\_rate': 0.2, 'activation\_function': 'relu', 'use\_batchnorm': False}



Configuration 5: {'optimizer': 'adam', 'learning\_rate': 1e-05, 'batch\_size': 16, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': False}

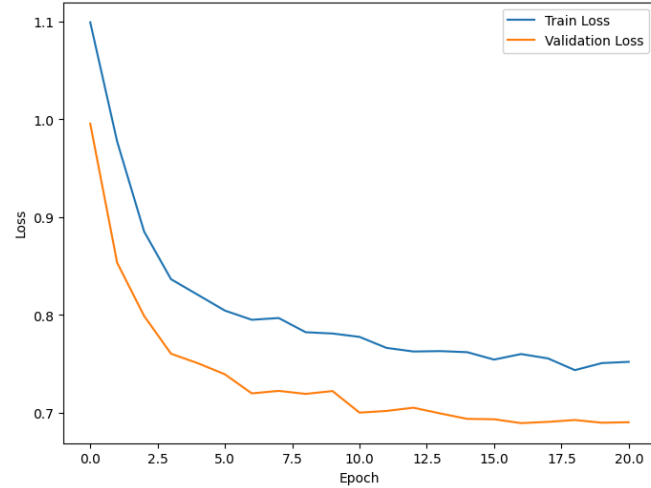


Configuration 6: {'optimizer': 'adam', 'learning\_rate': 3e-05, 'batch\_size': 16, 'dropout\_rate': 0.2, 'activation\_function': 'relu', 'use\_batchnorm': False}

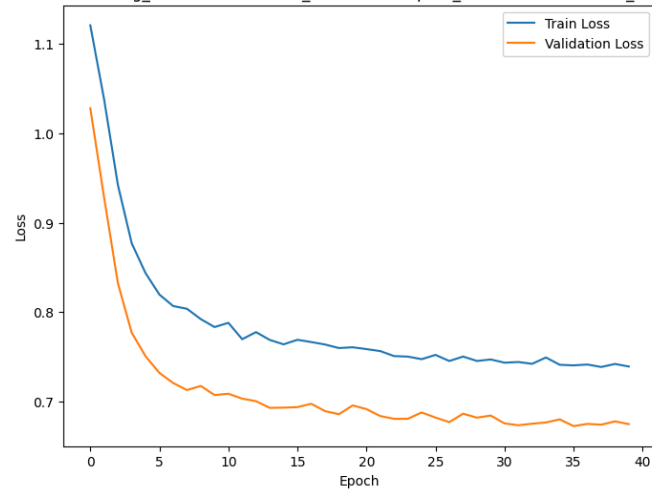


```
In [ ]: # Plot training and validation loss for each configuration
for i, (history, config) in enumerate(histories):
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title(f"Configuration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

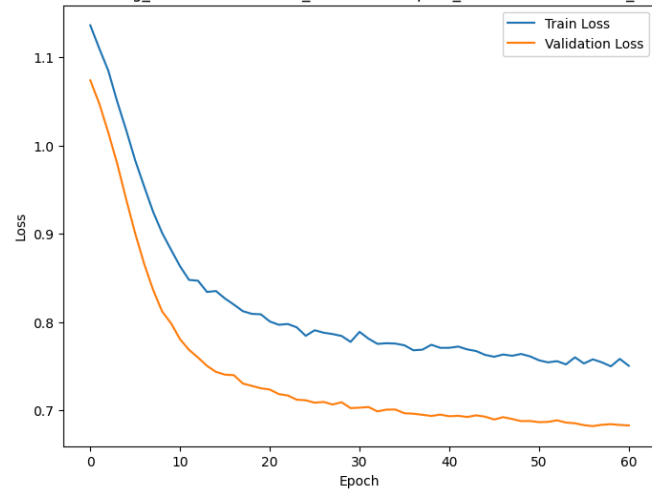
Configuration 1: {'optimizer': 'adamw', 'learning\_rate': 3e-05, 'batch\_size': 16, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': False}



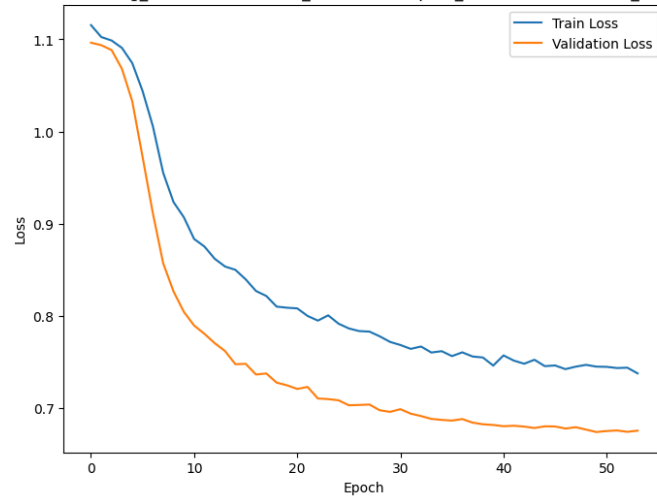
Configuration 2: {'optimizer': 'adamw', 'learning\_rate': 3e-05, 'batch\_size': 32, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': False}



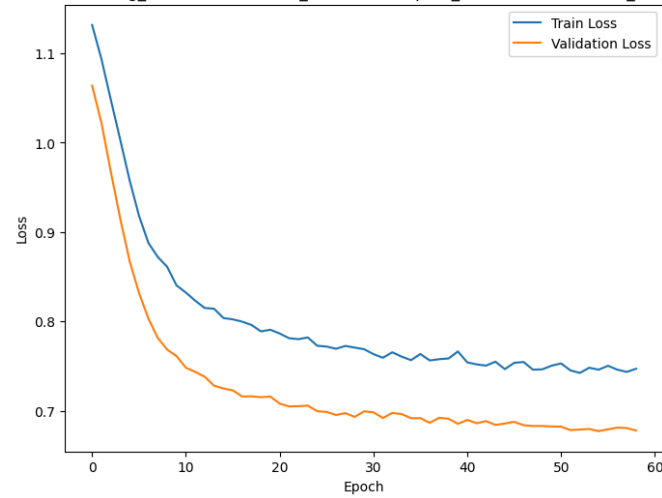
Configuration 3: {'optimizer': 'adamw', 'learning\_rate': 1e-05, 'batch\_size': 32, 'dropout\_rate': 0.1, 'activation\_function': 'tanh', 'use\_batchnorm': False}



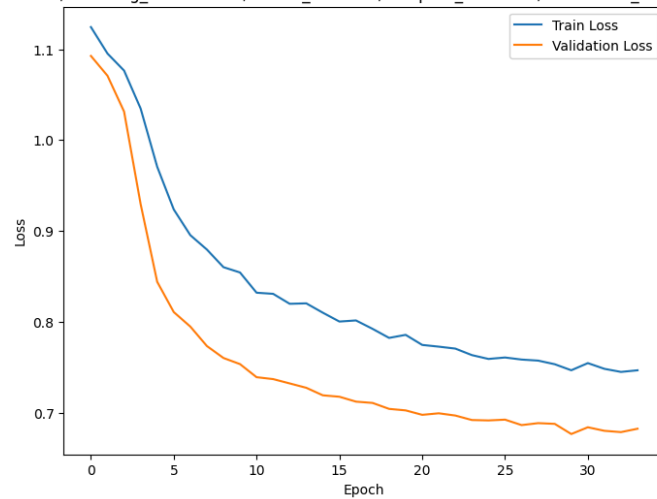
Configuration 4: {'optimizer': 'adam', 'learning\_rate': 3e-05, 'batch\_size': 32, 'dropout\_rate': 0.2, 'activation\_function': 'relu', 'use\_batchnorm': False}



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-base')

# 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', patier

# 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({

```



```

    '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

```

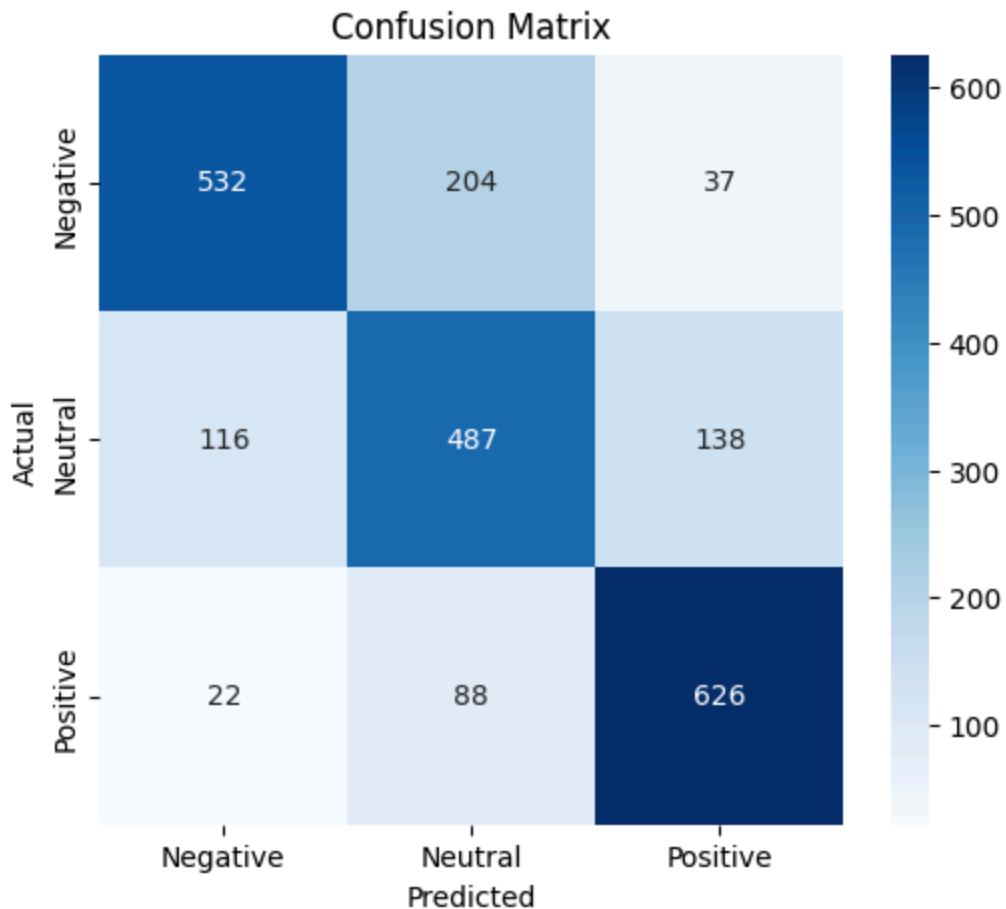
All PyTorch model weights were used when initializing TFRobertaForSequenceClassification.

Some weights or buffers of the TF 2.0 model TFRobertaForSequenceClassification were not initialized from the PyTorch model and are newly initialized: ['classifier.dense.weight', 'classifier.dense.bias', 'classifier.out\_proj.weight', '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 - accuracy: 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_ids]

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[predicted_label]))
    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[predicted_label]))

# 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 mug

True Label: Neutral

Predicted Label: Positive

Text: i love gloria jeans cinnamon nut strudel coffee and decided to try this the flavoring is a little overpowering 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 one of these is a jura s avantgarde and after trying dozens and dozens of various beans to find the perfect mate intelligentsia black cat classic prevailed as the very clear winner there may be better beans out there but not for this machine ive served thousands of shots of this espresso out of the jura and everyone is impressed by it actually not just impressed blown away its not 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 with 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 make funnel cakes i had been using a kitchen funnel which made a big mess no matter 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 some 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 dragging it around playing tug of war and just chewing on it unfortunately it does shed i dont know if there are rope toys out there that are any sturdier but this one does leave little threads behind otherwise great toy

True Label: Positive

Predicted Label: Neutral

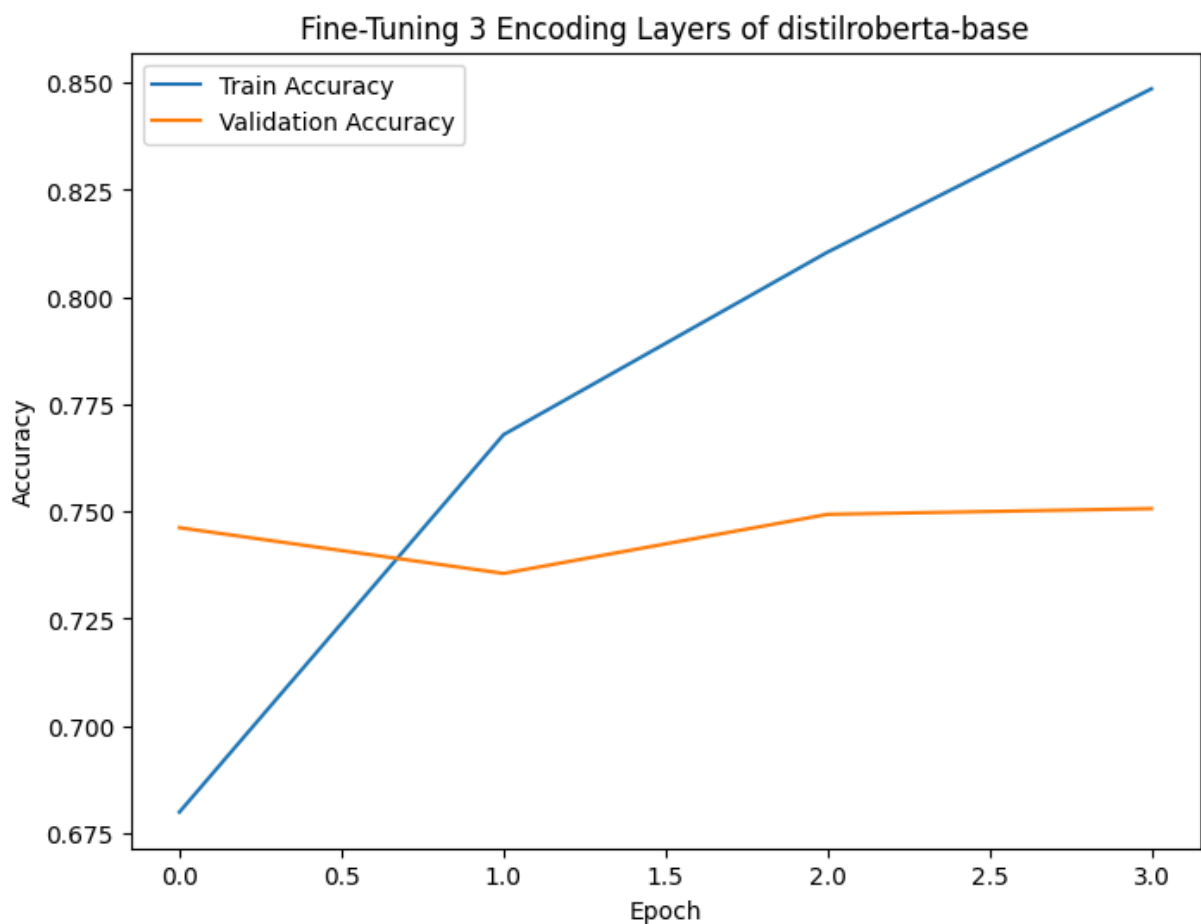
Text: our dogs love dingo bones but these peanut butter flavor bones are an obsession of our smallest rat terrier hes only pounds so bones tend to last several sittings for him give him one of these and hes occupied for as long as it takes to finish hell still play with other bones but he loses his mind 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, TFRobertaForSequenceClassification

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

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

```
In [ ]: 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 [ ]: 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 the 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 without 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: Previously, I implemented a Bidirectional GRU model on 100,000 samples, evenly split between 50,000 positive and 50,000 negative reviews. I used 100-dimensional GloVe for embedding and set the sample size to 716 characters, as 95% of the samples had a length of 716 or below. The model structure consisted of a Bidirectional GRU layer with 64 units and L2 regularization ( $1e-5$ ), followed by a dropout layer of 0.4, a dense layer with 32 units, another dropout layer of 0.5, a dense layer with 16 units, a dropout layer of 0.3, and finally, a dense layer with 1 unit. The learning rate was 0.005, the optimizer was Adam, and the model was trained for 10 epochs with a batch size of 128, using early stopping with patience set to 5. I did not generate a confusion matrix for this model.

Subsequently, I reduced the number of dropout layers to two, with lower dropout values of 0.3 and 0.2. I also changed the learning rate to  $5e-4$ , increased the batch size to 256, and set early stopping with a patience of 3. After generating a confusion matrix, I obtained the following results for the first model (using Adam): Accuracy: 80.83%, Precision: 76.62%, Recall: 88.94%, F1 Score: 82.32%.

For the second model, the structure and hyperparameters remained the same, but I switched to the Nadam optimizer. This led to slightly improved results: Accuracy: 81.14%, Precision: 78.26%, Recall: 86.43%, F1 Score: 82.14%. In my opinion, the Nadam optimizer performed better than Adam.

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.