Environmental Sound Classification

Name	ID
Ranime Ahmed Elsayed Shehata	21010531
Habiba Tarek Ramadan Ali	21010445
Karene Antoine Nassif Guirguis	21010969

Problem Statement:

Environmental sounds are critical cues for scene understanding in real-world applications like surveillance, autonomous navigation, and assistive technology. In this assignment, you will implement two deep learning approaches for classifying environmental audio recordings using the UrbanSound8K dataset. The first model will be based on Recurrent Neural Networks (RNNs), and the second will be a Transformer-based model, which you will implement from scratch (i.e., no use of nn.Transformer or Hugging Face models).

This lab aims to classify sounds from the UrbanSound8K dataset using two deep learning models:

- **Bi-directional LSTM (BiLSTM)**
- **Transformer (implemented from scratch)**

Step 1: Dataset and Preprocessing:

- **Dataset:** UrbanSound8K
- **Samples:** 8732 labeled audio clips (≤4 seconds)
- **Classes:** 10 sound categories (e.g., dog_bark, siren, jackhammer, etc.)
- **Splits:**
    - Folds 1–6: Training
    - Folds 7–8: Validation
    - Folds 9–10: Testing

Preprocessing steps:

- Loaded audio using `librosa`
- Extracted and visualized waveforms and mel spectrograms
- Played one sample per class for auditory inspection

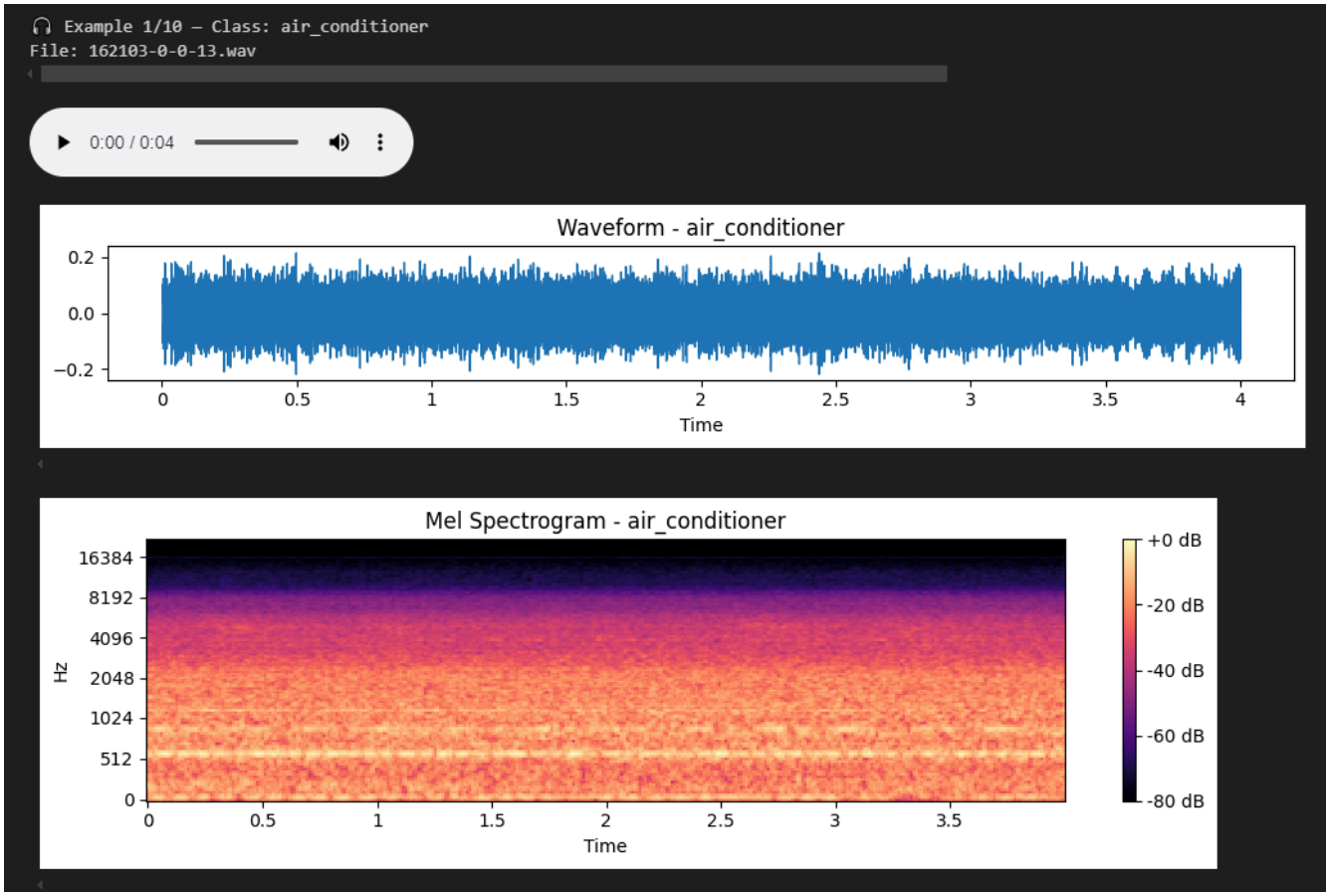
```python
import librosa
import librosa.display
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import os
import IPython.display as ipd

metadata = pd.read_csv('./UrbanSound8K/metadata/UrbanSound8K.csv')

def load_audio(file_path):
    signal, sr = librosa.load(file_path, sr=None)
    return signal, sr

def plot_waveform(signal, sr):
    plt.figure(figsize=(10, 3))
    librosa.display.waveshow(signal, sr=sr)
    plt.title("Waveform")
    plt.show()

def plot_melspectrogram(signal, sr):
    mel_spec = librosa.feature.melspectrogram(y=signal, sr=sr)
    log_mel_spec = librosa.power_to_db(mel_spec, ref=np.max)
    plt.figure(figsize=(10, 3))
    librosa.display.specshow(log_mel_spec, sr=sr, x_axis='time', y_axis='mel')
    plt.title("Mel-Spectrogram")
    plt.colorbar()
    plt.show()


```

Metadata:

	slice_file_name	fsID	start	end	salience	fold	classID	class
0	100032-3-0-0.wav	100032	0.000000	0.317551	1	5	3	dog_bark
1	100263-2-0-117.wav	100263	58.500000	62.500000	1	5	2	children_playing
2	100263-2-0-121.wav	100263	60.500000	64.500000	1	5	2	children_playing
3	100263-2-0-126.wav	100263	63.000000	67.000000	1	5	2	children_playing
4	100263-2-0-137.wav	100263	68.500000	72.500000	1	5	2	children_playing
...	...	...	...	...	...	...	...	...
8727	99812-1-2-0.wav	99812	159.522205	163.522205	2	7	1	car_horn
8728	99812-1-3-0.wav	99812	181.142431	183.284976	2	7	1	car_horn
8729	99812-1-4-0.wav	99812	242.691902	246.197885	2	7	1	car_horn
8730	99812-1-5-0.wav	99812	253.209850	255.741948	2	7	1	car_horn
8731	99812-1-6-0.wav	99812	332.289233	334.821332	2	7	1	car_horn

8732 rows × 8 columns

```python
# Visualize and listen to one random sample from each class

# Number of unique classes
num_classes = metadata['class'].nunique()

# Sample one file from each class
samples = metadata.groupby('class').apply(lambda x: x.sample(1)).reset_index(drop=True)

# Base path to the audio folder
AUDIO_BASE_PATH = './UrbanSound8K/audio'

# Loop over each sample and visualize
for i, row in samples.iterrows():
    file_name = row['slice_file_name']
    fold = row['fold']
    class_label = row['class']

    file_path = os.path.join(AUDIO_BASE_PATH, f"fold{fold}", file_name)
    print(f"\n🎧 Example {i+1}/{num_classes} — Class: {class_label}")
    print(f"File: {file_name}")

    # Load audio
    y, sr = librosa.load(file_path, sr=None)

    # Play audio
    display(ipd.Audio(y, rate=sr))

    # Plot waveform
    plt.figure(figsize=(10, 2))
    librosa.display.waveshow(y, sr=sr)
    plt.title(f"Waveform - {class_label}")
    plt.tight_layout()
    plt.show()

    # Plot Mel spectrogram
    S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128)
    S_DB = librosa.power_to_db(S, ref=np.max)

    plt.figure(figsize=(10, 3))
    librosa.display.specshow(S_DB, sr=sr, x_axis='time', y_axis='mel')
    plt.colorbar(format='%+2.0f dB')
    plt.title(f"Mel Spectrogram - {class_label}")
    plt.tight_layout()
    plt.show()

```



Step 2: Generating Feature Space:

For each audio file, the following features were extracted:

- **MFCCs (13 coefficients)**
- **Delta and Delta-Delta of MFCCs**
- **Spectral features:** Centroid, Bandwidth, Flatness, Roll-off
- **Zero-Crossing Rate**
- **Root Mean Square Energy (RMS)**

Total feature vector per clip: **45 dimensions**

```python
def extract_features(signal, sr):
    features = []

    # MFCCs
    mfcc = librosa.feature.mfcc(y=signal, sr=sr, n_mfcc=13)
    mfcc_mean = np.mean(mfcc, axis=1)
    features.extend(mfcc_mean)

    # Delta + Delta-Delta
    delta = librosa.feature.delta(mfcc, width=5)
    delta2 = librosa.feature.delta(mfcc, order=2 , width=5)
    features.extend(np.mean(delta, axis=1))
    features.extend(np.mean(delta2, axis=1))

    # Spectral features
    spec_centroid = librosa.feature.spectral_centroid(y=signal, sr=sr)
    spec_bw = librosa.feature.spectral_bandwidth(y=signal, sr=sr)
    spec_flat = librosa.feature.spectral_flatness(y=signal)
    spec_roll = librosa.feature.spectral_rolloff(y=signal, sr=sr)

    features.extend([
        np.mean(spec_centroid),
        np.mean(spec_bw),
        np.mean(spec_flat),
        np.mean(spec_roll)
    ])

    # Zero-crossing rate
    zcr = librosa.feature.zero_crossing_rate(signal)
    features.append(np.mean(zcr))

    # RMS Energy
    rms = librosa.feature.rms(y=signal)
    features.append(np.mean(rms))

    return np.array(features)

def prepare_dataset(folds):
    X, y = [], []
    for _, row in metadata.iterrows():
        if row['fold'] in folds:
            file_path = f"./UrbanSound8K/audio/fold{row['fold']}/{row['slice_file_name']}"
            signal, sr = librosa.load(file_path, sr=None)
            label = row['classID']
            features = extract_features(signal , sr)
            X.append(features)
            y.append(label)
    return np.array(X), np.array(y)

X_train, y_train = prepare_dataset([1,2,3,4,5,6])
X_val, y_val = prepare_dataset([7,8])
X_test, y_test = prepare_dataset([9,10])

```

Step 3: Building the Model:

➔ First: Bidirectional LSTM Model

Architecture:

- Input: (45, 1)
- Bidirectional **LSTM** with various hidden sizes and layers meaning the model learns patterns in both forward and backward time directions.
- **hidden_size:** Number of features in the LSTM hidden state per direction.
- Total output size from LSTM: **`hidden_size * 2`**
- **Fully Connected** After LSTM, it maps the concatenated forward and backward outputs to the final 10 class logits.

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

class BiRNNModel(nn.Module):
    def __init__(self, sequence_length=45, input_size=1, hidden_size=64, num_layers=1, num_classes=10):
        super(BiRNNModel, self).__init__()
        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            bidirectional=True
        )
        # hidden_size is doubled because of bidirectional
        self.fc = nn.Linear(hidden_size * 2, num_classes)

    def forward(self, x):
        # x: [batch_size, seq_len, input_size] = [B, 45, 1]
        _, (hn, _) = self.lstm(x)
        forward = hn[-2, :, :]  # [B, hidden_size]
        backward = hn[-1, :, :]  # [B, hidden_size]
        combined = torch.cat((forward, backward), dim=1)  # [B, 2*hidden_size]
        return self.fc(combined)

def train_model_rnn(model, train_loader, val_loader, epochs=50, lr=0.001):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    for epoch in range(epochs):
        model.train()
        for xb, yb in train_loader:
            xb, yb = xb.to(device), yb.to(device)
            out = model(xb.float())
            loss = criterion(out, yb)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        print(f"Epoch {epoch+1}/{epochs} - Loss: {loss.item():.4f}")

    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for xb, yb in val_loader:
            xb, yb = xb.to(device), yb.to(device)
            outputs = model(xb.float())
            _, predicted = torch.max(outputs.data, 1)
            total += yb.size(0)
            correct += (predicted == yb).sum().item()

    val_accuracy = correct / total
    return model, val_accuracy

```

Training:

- Optimizer: Adam
- Epochs: 50
- Loss: CrossEntropy
- Batch size: 32

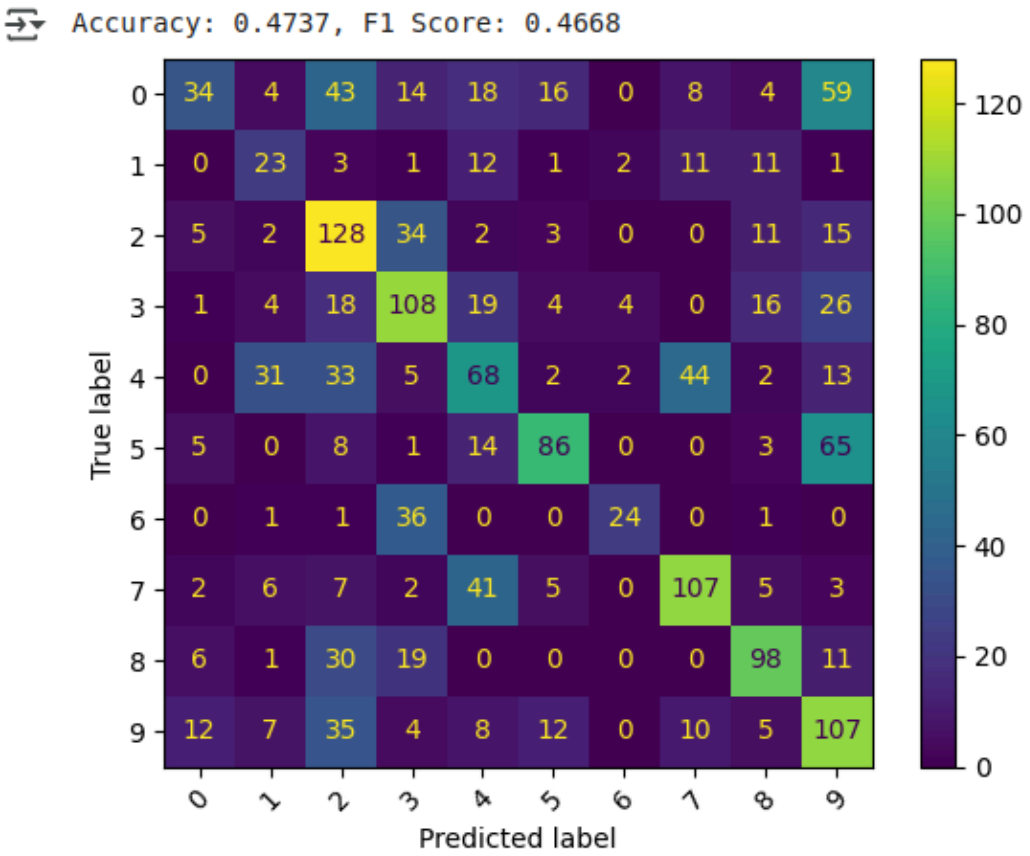
Hyperparameter Tuning

- Trying different combinations of:
  - **`hidden_size`: `[64, 128, 256, 512]`**
  - **`num_layers`: `[1, 2, 4, 6]`**

Best Configuration:

- `hidden_size=128`, `num_layers=1`
- **Validation Accuracy:** 48.24%
- **Test Accuracy:** 47.37%
- **Test F1 Score:** 46.68%

Confusion Matrix:



The most **confused pairs** in the model are:

1. **Air Conditioner ↔ Street Music**
2. **Air Conditioner ↔ Dog Bark**
3. **Children Playing ↔ Dog Bark**
4. **Engine Idling ↔ Dog Bark**
5. **Street Music ↔ Dog Bark**

These pairs are what the model finds hard to distinguish, mostly because:

- They have **overlapping frequency patterns**
- They are often recorded in **noisy urban environments**
- MFCCs and spectral features may not capture their fine differences

---

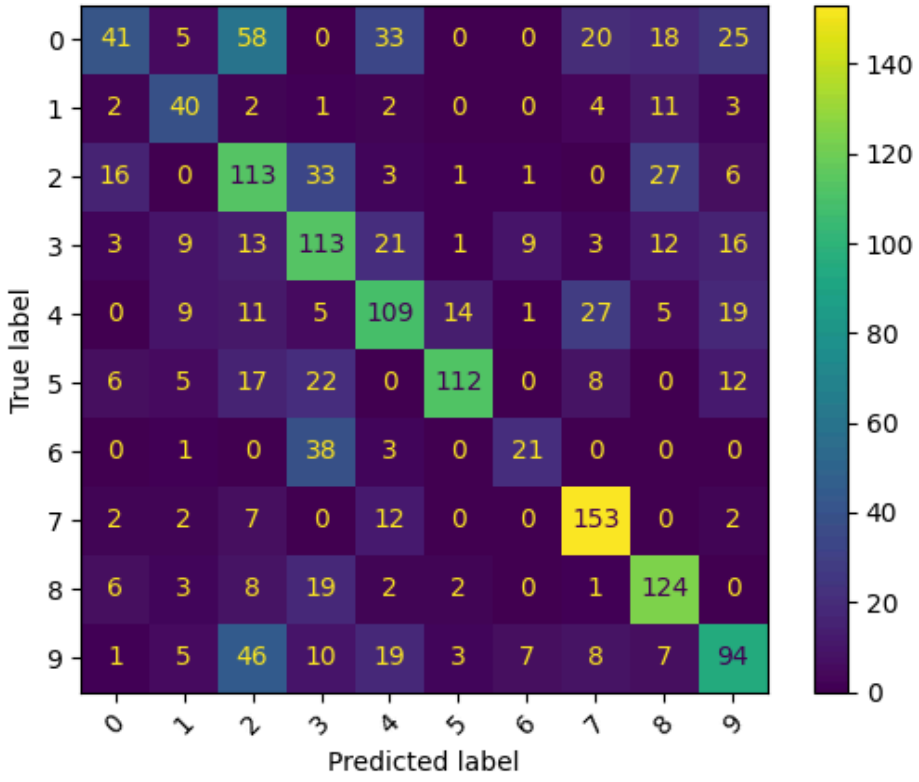
➔ Second: Transformer Model

A. Without Reshaping

- Input: (45, 1)
- **Best config**: `d_model=512, num_heads=4, d_ff=128, num_layers=6`
- **Validation Accuracy:** 58.21%
- **Test Accuracy:** 55.66%
- **Test F1 Score:** 54.59%
- **Confusion Matrix:**

```
#Scaling without reshaping "one step"
evaluate_model(best_config_wor[4], X_test_wor, y_test)
```

```
Accuracy: 0.5566, F1 Score: 0.5459
```



The Transformer model showed confusion between several sound classes. The most commonly confused pairs were:

- **Air Conditioner → Children Playing**
- **Street Music → Children Playing**
- **Gun Shot → Dog Bark**
- **Children Playing → Dog Bark**
- **Air Conditioner → Drilling**

These confusions occur because:

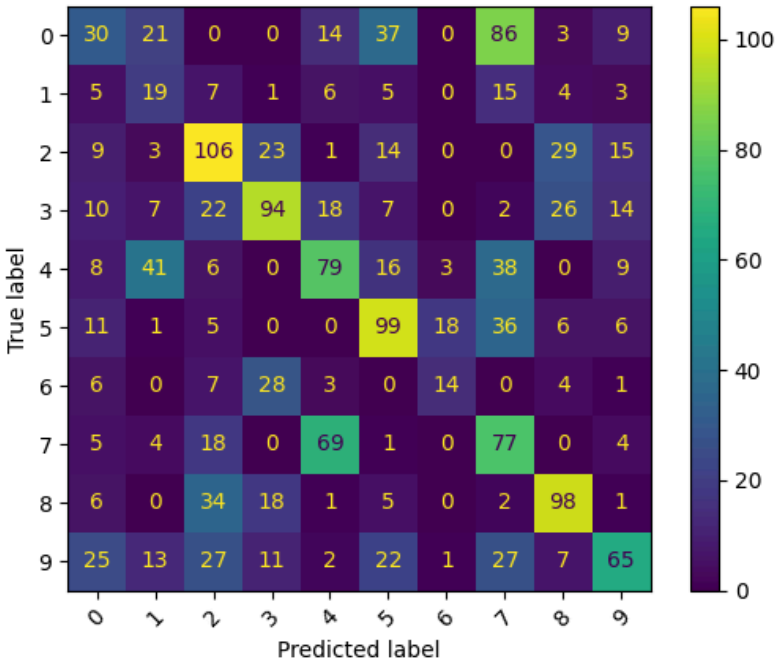
- Many classes **share overlapping acoustic features**, such as similar frequency ranges or sound textures (e.g., ambient noise, mechanical hums, or high-energy bursts).
- **Background noise and mixed recordings** in real-world urban environments lead to ambiguous patterns.
- **Feature extraction (e.g., MFCCs)** may not fully capture differences between continuous or impulsive sounds.
- **Transformer limitations** on short sequences may reduce its ability to distinguish subtle time-based patterns.

B. With Reshaping (9, 5)

- Reshaped features to simulate time series
- **Best config**: `d_model=256`, `num_heads=2`, `d_ff=256`, `num_layers=4`
- **Validation Accuracy:** 33.00%
- **Test Accuracy:** 41.20%
- **Test F1 Score:** 40.86%
- **Confusion Matrix:**

```
#Reshape without scaling
evaluate_model(best_configtf[4], X_testf, y_test)
```

```
Accuracy: 0.4120, F1 Score: 0.4086
```



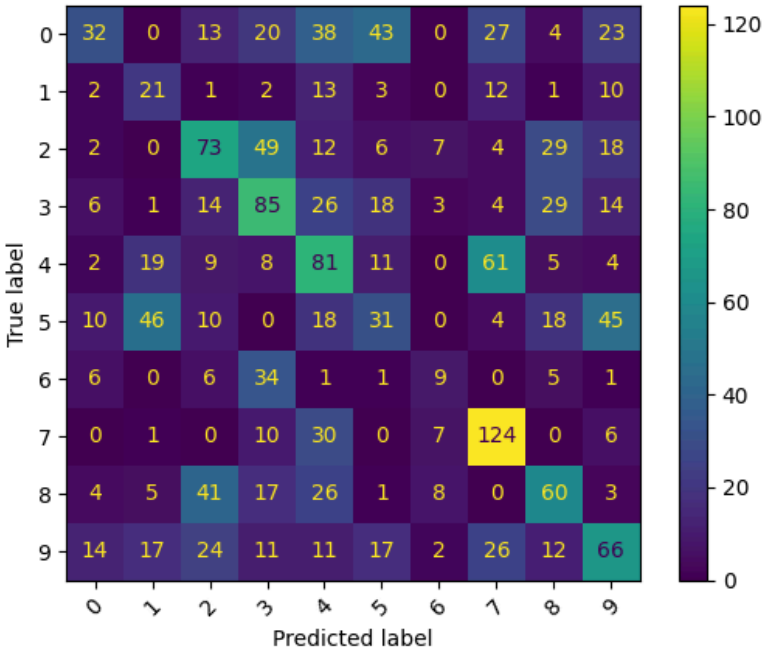
C. With Reshaping + Feature Scaling

- StandardScaler used
- **Best config**: `d_model=256`, `num_heads=4`, `d_ff=256`, `num_layers=4`
- **Validation Accuracy:** 34.25%
- **Test Accuracy:** 35.21%

- **Test F1 Score:** 33.95%
- **Confusion Matrix:**

```
#Scaling with reshaping
evaluate_model(best_config_scaled[4], X_test_scaled, y_test)
```

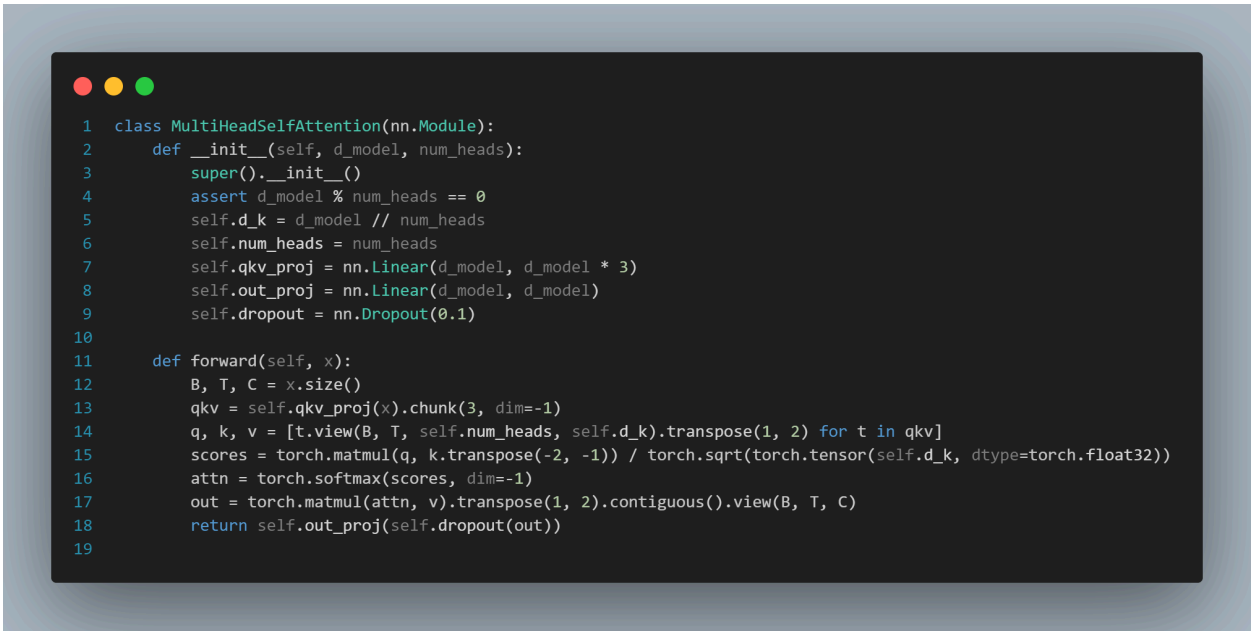
```
Accuracy: 0.3521, F1 Score: 0.3395
```



➔ Positional Encoding:

```python
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=500):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1).float()
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-torch.log(torch.tensor(10000.0)) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        self.pe = pe.unsqueeze(0)

    def forward(self, x):
        return x + self.pe[:, :x.size(1)].to(x.device)

```

➔ Architecture Class:



```python
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_k = d_model // num_heads
        self.num_heads = num_heads
        self.qkv_proj = nn.Linear(d_model, d_model * 3)
        self.out_proj = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(0.1)

    def forward(self, x):
        B, T, C = x.size()
        qkv = self.qkv_proj(x).chunk(3, dim=-1)
        q, k, v = [t.view(B, T, self.num_heads, self.d_k).transpose(1, 2) for t in qkv]
        scores = torch.matmul(q, k.transpose(-2, -1)) / torch.sqrt(torch.tensor(self.d_k, dtype=torch.float32))
        attn = torch.softmax(scores, dim=-1)
        out = torch.matmul(attn, v).transpose(1, 2).contiguous().view(B, T, C)
        return self.out_proj(self.dropout(out))

```

```python
class TransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, d_ff):
        super().__init__()
        self.attn = MultiHeadSelfAttention(d_model, num_heads)
        self.norm1 = nn.LayerNorm(d_model)
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(d_ff, d_model)
        )
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(0.1)

    def forward(self, x):
        x = self.norm1(x + self.attn(x))
        x = self.norm2(x + self.ff(x))
        return x

```

➔ From the notebook:

This class defines a complete Transformer-based model for sequence classification, which consists of:

Input Embedding:

Projects input features from input_dim to d_model using a linear layer.

Positional Encoding:

Adds positional information to the embedded input to retain sequence order.

Stacked Transformer Encoder Blocks:

A configurable number (num_layers) of TransformerBlock layers, each including:

- Multi-head self-attention
- Feed-forward network
- Layer normalization
- Residual connections
- Dropout

Global Sequence Pooling:

Applies AdaptiveAvgPool1d to collapse the sequence dimension and produce a fixed-size feature vector.

Classification Head:

A final linear layer maps the pooled output to num_classes class logits.

```python
class TransformerClassifier(nn.Module):
    def __init__(self, input_dim, d_model, num_heads, d_ff, num_layers, num_classes):
        super().__init__()
        self.embedding = nn.Linear(input_dim, d_model)
        self.pos_encoding = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([TransformerBlock(d_model, num_heads, d_ff) for _ in range(num_layers)])
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.classifier = nn.Linear(d_model, num_classes)

    def forward(self, x):
        x = self.embedding(x)
        x = self.pos_encoding(x)
        for layer in self.layers:
            x = layer(x)
        x = x.transpose(1, 2)
        x = self.pool(x).squeeze(-1)
        return self.classifier(x)

```

```python
def train_model_transformer(model, train_loader, val_loader, epochs=50, lr=1e-4):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.8)
    criterion = nn.CrossEntropyLoss()

    for epoch in range(epochs):
        model.train()
        for xb, yb in train_loader:
            xb, yb = xb.to(device).float(), yb.to(device)
            outputs = model(xb)
            loss = criterion(outputs, yb)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        scheduler.step()
        print(f"Epoch {epoch+1}/{epochs} - Loss: {loss.item():.4f}")

    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for xb, yb in val_loader:
            xb, yb = xb.to(device).float(), yb.to(device)
            preds = model(xb).argmax(dim=1)
            correct += (preds == yb).sum().item()
            total += yb.size(0)
    val_acc = correct / total
    return model, val_acc


```

Comparison between the 2 Models:

Metrics	Bi-LSTM	Transformer “without reshaping”
Accuracy	47.37%	55.66%
F1 Score	46.68%	54.59%
