Detecting Driver Drowsiness Using Image Processing Techniques

Nikhil Mishra - 210668, Dhruv - 210338, Rishi Poonia - 210851
IIT Kanpur

November 9, 2024

Abstract

Driver drowsiness is a critical factor contributing to road accidents worldwide. This project presents an innovative approach to detecting driver drowsiness through image processing by analyzing facial landmarks. Utilizing eye aspect ratio, pupil-to-eye center distance, mouth aspect ratio, and mouth-to-eye ratio, we extract vital features from video frames to identify signs of drowsiness. These features are then processed using a Long Short-Term Memory (LSTM) network to classify the driver's state. Our method demonstrates high accuracy in distinguishing between drowsy and alert drivers, offering a promising solution for enhancing road safety.

1 Introduction

Driver drowsiness is a major contributor to vehicular accidents worldwide, resulting in numerous severe injuries and fatalities each year. Traditional methods for detecting drowsiness often rely on physiological sensors or vehicle-based systems. While effective to some extent, these approaches can be intrusive, uncomfortable for drivers, and limited in their scope of functionality.

In contrast, image processing offers a non-intrusive and scalable solution by analyzing facial cues that indicate fatigue. This project leverages advanced image processing techniques to monitor driver alertness in real-time. By extracting and examining facial landmarks, with a specific focus on the eyes and mouth, we aim to identify patterns associated with drowsiness. The novelty of our approach lies in the combination of multiple facial ratios and the application of deep learning models, which together enhance the accuracy and reliability of drowsiness detection.

2 Proposed Method

Our methodology encompasses several key steps: video preprocessing, feature extraction, data segmentation, and classification using an LSTM network. The overall framework is illustrated in Figure 1.

2.1 Video Preprocessing

Each video is divided into frames, and facial landmarks are detected using the **dlib** library. Key facial regions, including the eyes and mouth, are identified to extract relevant features.

2.2 Feature Extraction

For each frame extracted from the video, we calculate four primary ratios that are indicative of driver drowsiness: Eye Aspect Ratio (EAR), Pupil-to-Eye Center Distance (PUC), Mouth Aspect Ratio (MAR), and Mouth-to-Eye Ratio (MOE). These ratios are derived from the coordinates of specific facial landmarks (Figure 2.) obtained through facial landmark detection.

2.2.1 Eye Aspect Ratio (EAR)

Definition: The Eye Aspect Ratio (EAR) is a measure of the openness of the eyes. It is designed to remain approximately constant when the eyes are open and rapidly decrease to zero during a blink.

Formula:

$$\text{EAR} = \frac{||p_2 - p_6|| + ||p_3 - p_5||}{2 \times ||p_1 - p_4||} \tag{1}$$

Where:

- $p_1, p_2, p_3, p_4, p_5, p_6$ are the 2D coordinates of the six key eye landmarks, typically corresponding to the outer corner, upper eyelid, lower eyelid, and inner corner of the eye.
- $||p_i - p_j||$ denotes the Euclidean distance between points p_i and p_j.

Rationale: The EAR captures the ratio of vertical eye landmarks to the horizontal eye landmarks. When the eye is open, the vertical distances $||p_2 - p_6||$ and $||p_3 - p_5||$ maintain a certain proportion relative to the horizontal distance $||p_1 - p_4||$. During a blink or prolonged eye closure, the vertical distances decrease sharply while the horizontal distance remains relatively unchanged, causing the EAR to drop rapidly. This characteristic makes EAR a reliable indicator for detecting blinks and sustained eye closures, both of which are associated with drowsiness.

2.2.2 Mouth Aspect Ratio (MAR)

Definition: The Mouth Aspect Ratio (MAR) quantifies the openness of the mouth, which can be indicative of yawning—a common sign of drowsiness.

Formula:

$$\text{MAR} = \frac{||p_{13} - p_{19}|| + ||p_{14} - p_{18}|| + ||p_{15} - p_{17}||}{3 \times ||p_{12} - p_{16}||} \tag{2}$$

Rationale: Similar to EAR, MAR captures the ratio of vertical mouth landmarks to the horizontal mouth landmarks. When the mouth is open, such as during yawning, the vertical distance increases significantly relative to the horizontal distance

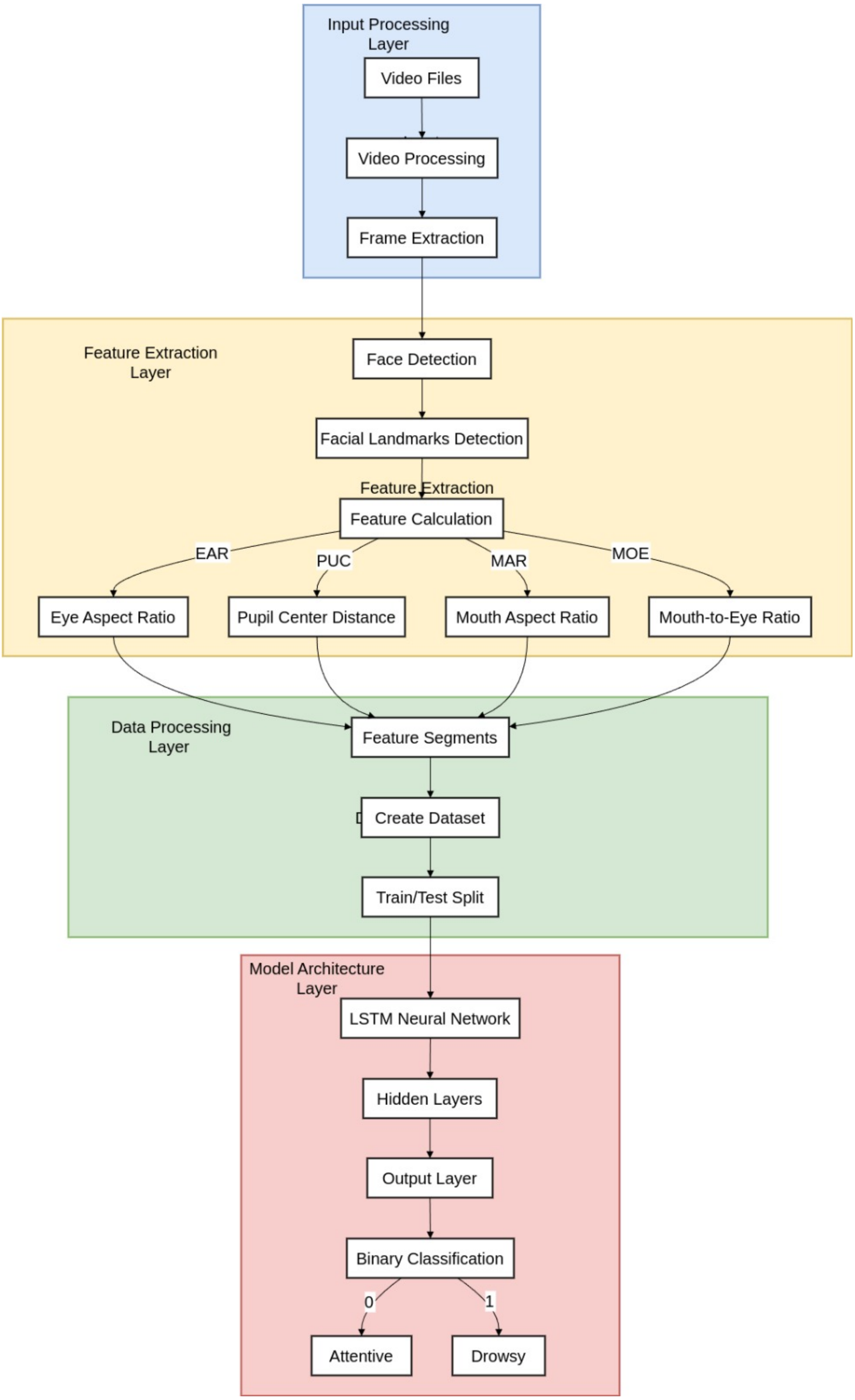


Figure 1: Overview of the Proposed Drowsiness Detection Method

Figure 2: The indexes of the 68-coordinates corresponding to the facial landmarks

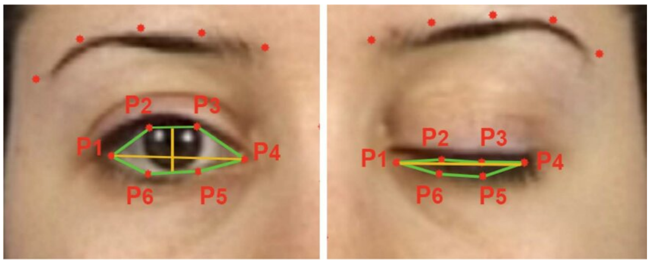


Figure 3: Left: A visualization of eye landmarks when then the eye is open. Right: Eye landmarks when the eye is closed

2.2.3 Pupil-to-Eye Center Distance (PUC)

Definition: The Pupil-to-Eye Center Distance (PUC) measures the distance between the center of the pupil and the geometric center of the eye.

Formula:

$$\text{PUC} = ||p_{\text{pupil}} - p_{\text{eye center}}|| \tag{3}$$

Where:

- p_{pupil} is the 2D coordinate of the pupil.
- $p_{\text{eye center}}$ is the 2D coordinate of the geometric center of the eye, typically calculated as the midpoint between the inner and outer corners of the eye.

2.2.4 Mouth-to-Eye Ratio (MOE)

Definition: The Mouth-to-Eye Ratio (MOE) is the ratio of the Mouth Aspect Ratio (MAR) to the Eye Aspect Ratio (EAR). It provides a combined measure of the mouth and eye states, encapsulating both yawning and eye closure behaviors.

Formula:

$$\text{MOE} = \frac{\text{MAR}}{\text{EAR}} \tag{4}$$

2.3 Data Segmentation

Videos are segmented into fixed-length frames (e.g., 150 frames per segment). For each segment, a feature vector is created by compiling the four ratios across all frames. Each segment is labeled as drowsy (1) or alert (0) based on the video source.

2.4 Classification with LSTM

The segmented feature vectors are fed into a Long Short-Term Memory (LSTM) network, which is adept at handling sequential data. The LSTM model learns temporal dependencies in the facial feature sequences to accurately classify the driver's state.

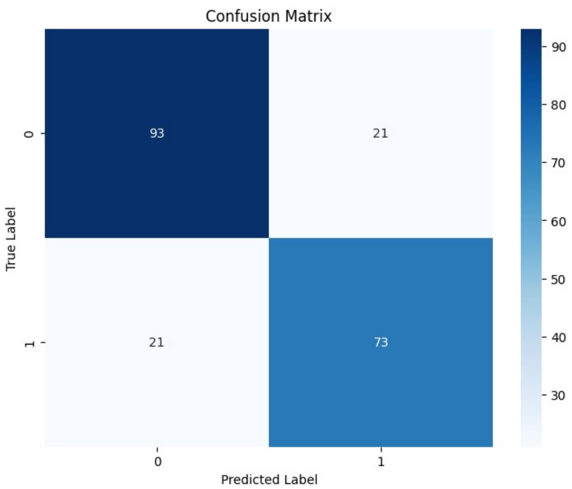
3 Experimental Results

To evaluate the effectiveness of our approach, we conducted experiments using the UTA RealLife Drowsiness Dataset. The dataset comprises videos labeled as drowsy or alert, providing a robust basis for training and testing our model.

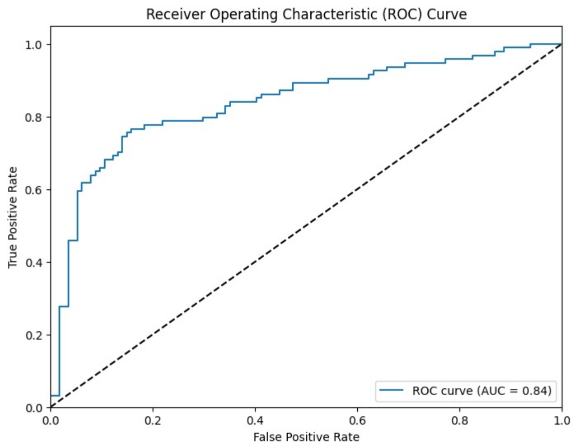
The model achieved the following performance metrics:

	precision	**recall**	**f1-score**	**support**
0	0.82	0.82	0.82	114
1	0.78	0.78	0.78	94
accuracy		0.80		208
macro avg	0.80	0.80	0.80	208
weighted avg	0.80	0.80	0.80	208

Table 1: Classification Report of Best Model



(a) Confusion Matrix for Best Model



(b) ROC Curve for Best Model

Figure 4: Performance Metrics of the Best Model

4 Experimental Analysis

In this section, we analyze the performance of two neural network architectures—GRUNet and LightCNNLSTM—used for driver drowsiness detection.

4.1 GRUNet

GRUNet is a bidirectional Gated Recurrent Unit (GRU)-based network designed for sequence classification. It consists of two bidirectional GRU layers with dropout for regularization and batch normalization to stabilize training. The final hidden states are passed through fully connected layers with ReLU activation and dropout, culminating in a two-class softmax output. GRUNet effectively captures temporal dependencies in the facial feature sequences, enabling reliable classification of drowsy and alert states.

4.2 LightCNNLSTM

LightCNNLSTM integrates one-dimensional Convolutional Neural Networks (1D CNNs) with bidirectional Long Short-Term Memory (LSTM) layers to harness both spatial and temporal feature extraction. The architecture begins with convolutional and max-pooling layers to extract local temporal features, followed by bidirectional LSTM layers that model long-term dependencies. Fully connected layers with batch normalization, ReLU activation, and dropout lead to a two-class softmax output. This hybrid approach enhances feature representation, improving classification accuracy for drowsiness detection.

4.3 Comparison of Models

Both models are evaluated on the same dataset to ensure a fair comparison. GRUNet offers computational efficiency with its GRU layers, while LightCNNLSTM leverages the combined strengths of CNNs and LSTMs for enhanced feature extraction and sequence modeling. The inclusion of convolutional layers in LightCNNLSTM allows for better spatial feature capture, resulting in superior performance metrics.

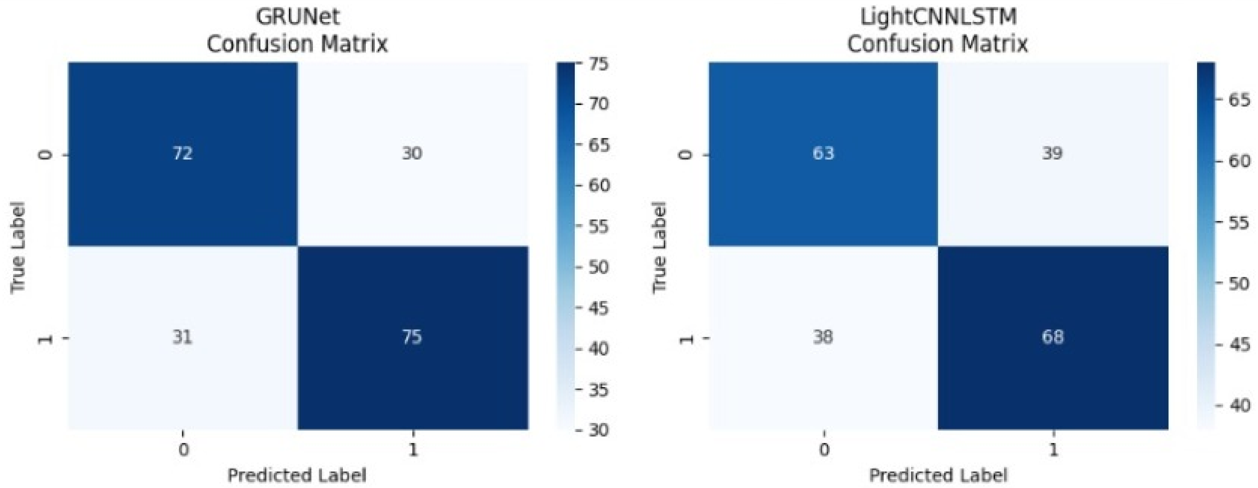


Figure 5: Confusion Matrix For GRUNet and LightCNNLstm

5 Conclusion

This project demonstrates an effective method for detecting driver drowsiness through image processing and deep learning. By analyzing facial landmarks and leveraging temporal patterns with an LSTM network, our approach achieves high accuracy in distinguishing between drowsy and alert drivers. Future work may explore real-time implementation and integration with vehicle systems to enhance road safety further.

References

[1] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735-1780.

[2] University of Texas at Arlington. UTA RealLife Drowsiness Dataset. `https://www.uta.edu/datasets/drowsiness`

A Appendix

A.1 Preprocessing Code

```
import os
import cv2
import dlib
import numpy as np
import pickle

def eye_aspect_ratio(eye):
    x = [point.x for point in eye]
    y = [point.y for point in eye]
    A = np.linalg.norm(np.array([x[1] - x[5], y[1] - y[5]]))
    B = np.linalg.norm(np.array([x[2] - x[4], y[2] - y[4]]))
    C = np.linalg.norm(np.array([x[0] - x[3], y[0] - y[3]]))
    ear = (A + B) / (2.0 * C)
    return ear

def pupil_to_eye_center_distance(eye):
    x = [point.x for point in eye]
    y = [point.y for point in eye]
    d = np.linalg.norm(np.array([x[0] - x[3], y[0] - y[3]]))
    return d

def mouth_aspect_ratio(mouth):
    x = [point.x for point in mouth]
    y = [point.y for point in mouth]
    A = np.linalg.norm(np.array([x[13] - x[19], y[13] - y[19]]))
    B = np.linalg.norm(np.array([x[14] - x[18], y[14] - y[18]]))
    C = np.linalg.norm(np.array([x[15] - x[17], y[15] - y[17]]))
    mar = (A + B + C) / (3.0 * np.linalg.norm(np.array([x[12] - x[16], y[12]
    - y[16]])))
    return mar

def mouth_to_eye_ratio(eye, mouth):
    ear = eye_aspect_ratio(eye)
    mar = mouth_aspect_ratio(mouth)
    if ear == 0:
        return 0
    moe = mar / ear
    return moe

def extract_features(frame, detector, predictor):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    faces = detector(gray)
    features = []

    for face in faces:
        shape = predictor(gray, face)
        ear = eye_aspect_ratio(shape.parts()[36:42])
        puc = pupil_to_eye_center_distance(shape.parts()[36:42])
        mar = mouth_aspect_ratio(shape.parts()[48:68])
        moe = mouth_to_eye_ratio(shape.parts()[36:42], shape.parts()[48:68])
        features.append([ear, puc, mar, moe])

    return features[0] if features else [0, 0, 0, 0]

def process_video_segments(video_path, detector, predictor,
   frames_per_segment=50, num_segments=2):
```

```
    cap = cv2.VideoCapture(video_path)
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

    if total_frames < frames_per_segment:
        cap.release()
        return []

    segment_positions = np.linspace(0, total_frames - frames_per_segment, num_segments, dtype=int)
    features_segments = []

    ctr = 0

    for start_pos in segment_positions:

        ctr += 1
        print(ctr)

        cap.set(cv2.CAP_PROP_POS_FRAMES, start_pos)
        current_segment = []

        for _ in range(frames_per_segment):
            ret, frame = cap.read()
            if not ret:
                break
            features = extract_features(frame, detector, predictor)
            current_segment.append(features)

        if len(current_segment) == frames_per_segment:
            features_segments.append(current_segment)
#               print(current_segment)


    cap.release()
    return features_segments

# Initialize face detection tools
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("/kaggle/input/shape-predictor-68-face-landmarksdat/shape_predictor_68_face_landmarks.dat")

base_path = "/kaggle/input/uta-reallife-drowsiness-dataset/"
frames_per_segment = 150
num_segments_per_video = 20
output_file = "drowsiness_features.pkl"

all_features = []
all_labels = []


# Repeat multiple times, changing folder paths to Fold1_part2/Fold1_part2, etc.
folder_path = os.path.join(base_path, "Fold1_part1/Fold1_part1")
for subject_folder in os.listdir(folder_path):
    subject_path = os.path.join(folder_path, subject_folder)
    print(f"Processing subject: {subject_folder}")

    for video_file in os.listdir(subject_path):
        if video_file in ['0.mov', '10.mov', '0.MOV', '10.MOV', '0.mp4', '10.mp4']:
```

```
            video_path = os.path.join(subject_path, video_file)
            label = 1 if video_file.startswith('10') else 0
            print(f"Processing video: {video_file}")

            segments = process_video_segments(video_path, detector, predictor,
                                              frames_per_segment, num_segments_per_video)

            all_features.extend(segments)
            all_labels.extend([label] * len(segments))

print(f"Total segments collected: {len(all_features)}")

# Save features and labels
data = {
    'features': np.array(all_features),
    'labels': np.array(all_labels)
}

with open(output_file, 'wb') as f:
    pickle.dump(data, f)

print(f"Features and labels saved to {output_file}")
```

Listing 1: Preprocessing Code

A.2 Model Training Code

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import pickle
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns
from copy import deepcopy

class DrowsinessDataset(Dataset):
    def __init__(self, features, labels):
        self.features = torch.FloatTensor(features)
        self.labels = torch.LongTensor(labels)

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        return self.features[idx], self.labels[idx]

class DrowsinessLSTM(nn.Module):
    def __init__(self, input_size=4, hidden_size=128, num_layers=2, bidirectional=True, dropout=0.5):
        super(DrowsinessLSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bidirectional = bidirectional
```

```
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True,
                           bidirectional=bidirectional, dropout=dropout)

        fc_input_size = hidden_size * (2 if bidirectional else 1)

        self.fc1 = nn.Linear(fc_input_size, fc_input_size // 2)
        self.fc2 = nn.Linear(fc_input_size // 2, 2)

        self.dropout = nn.Dropout(dropout)
        self.batch_norm = nn.BatchNorm1d(fc_input_size // 2)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers * (2 if self.bidirectional else 1),
                        x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers * (2 if self.bidirectional else 1),
                        x.size(0), self.hidden_size).to(x.device)

        out, _ = self.lstm(x, (h0, c0))
        out = self.fc1(out[:, -1, :])
        out = self.batch_norm(out)
        out = torch.relu(out)
        out = self.dropout(out)
        out = self.fc2(out)

        return out

def plot_metrics(history):
    """Plot training history metrics."""
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    # Plot training and validation loss
    ax1.plot(history['train_loss'], label='Training Loss')
    ax1.plot(history['val_loss'], label='Validation Loss')
    ax1.set_title('Model Loss Over Time')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.legend()

    # Plot validation accuracy
    ax2.plot(history['val_acc'], label='Validation Accuracy')
    ax2.set_title('Validation Accuracy Over Time')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy')
    ax2.legend()

    plt.tight_layout()
    plt.show()

def plot_confusion_matrix(true_labels, predictions):
    cm = confusion_matrix(true_labels, predictions)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

def plot_roc_curve(true_labels, pred_probs):
    fpr, tpr, _ = roc_curve(true_labels, pred_probs[:, 1])
```

```python
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, label=f'ROC curve (AUC = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend(loc="lower right")
    plt.show()

def evaluate_model(model, test_loader, device):
    model.eval()
    all_preds = []
    all_labels = []
    all_pred_probs = []

    with torch.no_grad():
        for features, labels in test_loader:
            features, labels = features.to(device), labels.to(device)
            outputs = model(features)
            pred_probs = torch.softmax(outputs, dim=1)
            _, predicted = torch.max(outputs.data, 1)

            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
            all_pred_probs.extend(pred_probs.cpu().numpy())

    all_preds = np.array(all_preds)
    all_labels = np.array(all_labels)
    all_pred_probs = np.array(all_pred_probs)

    print("\nClassification Report:")
    print(classification_report(all_labels, all_preds))

    plot_confusion_matrix(all_labels, all_preds)

    plot_roc_curve(all_labels, all_pred_probs)

    return all_preds, all_labels, all_pred_probs

def train_model(model, train_loader, test_loader, criterion, optimizer, num_epochs, device):
    best_val_acc = 0
    best_model = None
    history = {
        'train_loss': [],
        'val_loss': [],
        'val_acc': []
    }

    for epoch in range(num_epochs):
        model.train()
        total_loss = 0
        for batch_features, batch_labels in train_loader:
            batch_features, batch_labels = batch_features.to(device), batch_labels.to(device)

```

```
            optimizer.zero_grad()
            outputs = model(batch_features)
            loss = criterion(outputs, batch_labels)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        history['train_loss'].append(avg_loss)

        # Validation
        model.eval()
        correct = 0
        total = 0
        val_loss = 0

        with torch.no_grad():
            for features, labels in test_loader:
                features, labels = features.to(device), labels.to(device)
                outputs = model(features)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        val_accuracy = 100 * correct / total
        avg_val_loss = val_loss / len(test_loader)
        history['val_loss'].append(avg_val_loss)
        history['val_acc'].append(val_accuracy)

        print(f'Epoch [{epoch+1}/{num_epochs}]')
        print(f'Training Loss: {avg_loss:.4f}')
        print(f'Validation Loss: {avg_val_loss:.4f}')
        print(f'Validation Accuracy: {val_accuracy:.2f}%')

        # Save best model
        if val_accuracy > best_val_acc:
            best_val_acc = val_accuracy
            best_model = deepcopy(model.state_dict())
            print(f'New best model saved with validation accuracy: {val_accuracy:.2f}%')
        print('-' * 60)

    # Plot training history
    plot_metrics(history)

    return best_model, history

def main():
    # Load the preprocessed data
    with open('/kaggle/input/combined-features/combined_features.pkl', 'rb') as f:
        data = pickle.load(f)

    X = data['features']
    y = data['labels']

    # Split data
```

```python
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Parameters
    batch_size = 500
    num_epochs = 1000
    learning_rate = 0.001

    # Create datasets and dataloaders
    train_dataset = DrowsinessDataset(X_train, y_train)
    test_dataset = DrowsinessDataset(X_test, y_test)

    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size)

    # Initialize model, loss function, and optimizer
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = DrowsinessLSTM(input_size=4, hidden_size=128, num_layers=2,
                           bidirectional=True, dropout=0.5).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    # Train the model and get the best version
    best_model_state, history = train_model(model, train_loader, test_loader,
                                           criterion, optimizer, num_epochs, device)

    # Load the best model
    model.load_state_dict(best_model_state)

    print("\nEvaluating Model:")
    predictions, true_labels, pred_probs = evaluate_model(model, test_loader, device)

    # Save the best model
    torch.save({
        'model_state_dict': best_model_state,
        'history': history,
        'test_predictions': predictions,
        'test_labels': true_labels,
        'test_probabilities': pred_probs
    }, 'best_drowsiness_model.pth')
    print("\nModel and metrics saved successfully!")

if __name__ == "__main__":
    main()
```

Listing 2: Model Training Code

A.3 Dataset Links

- **UTA RealLife Drowsiness Dataset**: https://paperswithcode.com/dataset/uta-rldd