A MULTIMODAL DRIVER-SAFETY SYSTEM TO SUPPORT TEEN DRIVERS USING COMPUTER VISION AND SENSOR FUSION

Max Liu[1], Andrew Park[2]

[1]William P Clements High School, 4200 Elkins Rd, Sugar Land, TX 77479
[2]University of California, Irvine, Irvine, CA 92697

ABSTRACT

Teen drivers experience disproportionately high crash rates, largely due to inexperience and inconsistent attention to basic traffic rules. To address this issue, we developed a multimodal driver-safety coaching system deployed on a low-cost Raspberry Pi platform. The system uses three trained YOLO-based computer vision models to detect traffic lights, illuminated bulbs within traffic lights, and traffic signs. An Optical Character Recognition (OCR) module extracts numerical speed limit values, which are combined with GPS data to identify speeding behavior, while a trained audio classification model and IMU data are used to determine whether turn signals are activated during turning maneuvers. Post-detection processing techniques are applied to smooth noisy detections over time and trigger prioritized voice alerts via text-to-speech (TTS). Key challenges include achieving sufficient model accuracy under diverse environmental conditions, maintaining real-time performance on resource-constrained hardware, and coordinating multiple hardware devices through sensor fusion. Experimental results demonstrate both the strengths and limitations of the system, guiding future improvements. Overall, the proposed system illustrates a practical, low-cost approach to helping novice drivers develop safer driving behaviors in real-world settings.

KEYWORDS

Computer vision, Multimodal sensor fusion, Driver behavior monitoring

1. INTRODUCTION

Learning to drive is one of the riskiest transitions in a teenager's life. Around the world, road traffic crashes kill an estimated 1.19 million people around the world each year and leave between 20 and 50 million people with non-fatal injuries [1]. In the United States, teen drivers ages 16–19 have a fatal crash rate nearly three to four times higher than that of older drivers when adjusted for miles driven [2]. Recent national data further indicate that this problem is worsening: a large-scale study by Bumper reports that road fatalities involving teen drivers have increased by approximately 25% over the past decade [3].

For novice drivers, the problem is not just a lack of skill, but also inexperience with real-world road conditions, difficulty managing distractions, and limited awareness of how quickly situations can change at an intersection or on a highway. Prior work in smartphone sensing and telematics has shown that sensors, such as GPS and IMU, can be used to detect harsh braking, speeding, and other risky behaviors, offering a way to monitor and coach drivers [16]. However, many existing systems either focus on post-incident analysis or are limited to simple metrics like speed and acceleration without understanding the visual context on the road. Additionally, many teen drivers drive older or

David C. Wyld et al. (Eds): ACSTY, MLSC, SVC, AIBD, ITCSS, ADCOM, NATP, SOFE – 2026
pp. 133-151, 2026. CS & IT - CSCP 2026 DOI: 10.5121/csit.2026.160411

cheaper vehicles that lack the advanced safety systems found in newer cars, further contributing to this issue.

This gap matters because the consequences of unsafe driving accumulate over time. A teenager who repeatedly engages in unsafe driving is not just at risk in the moment; these behaviors can become ingrained and increase the likelihood of serious crashes over years of driving. The long-term impact extends to families, schools, and communities, making it critical to develop tools that actively support safe driving habits from the very beginning of a driver's experience [17].

Methodology A: Nine et al. demonstrate that traffic-light detection using deep-learning models, such as MobileNetSSD, is feasible on Raspberry Pi hardware. The system processes forward-facing camera video to classify traffic-light states in real time. However, it does not include traffic-sign detection, sensor fusion, or any driver alerting or feedback mechanisms.

Methodology B: Isa et al. present a real-time traffic-sign recognition system implemented on a Raspberry Pi using a TensorFlow-based model. The system achieves high accuracy under controlled conditions but is limited to visual traffic-sign detection only. It does not detect traffic lights or integrate additional sensor data, such as GPS, IMU, or audio inputs.

Methodology C: Setiawan, Y. et al. present a YOLO-based traffic-sign detection system that processes live camera input to notify drivers in real time. While the system achieves reliable performance, similar to Methodology B, it is limited to traffic-sign detection and lacks multimodal sensor fusion. In addition, it delivers feedback through visual, vibration, and RF signals rather than eye-free voice alerts that better support safer driving.

To address this gap, we developed a driving safety system that leverages computer vision and multimodal sensor fusion [5] and operates on low-cost, portable hardware, such as Raspberry Pi, to provide real-time voice feedback to help novice drivers stay safe on the road.

The system collects data from a front-facing camera, GPS module, IMU module, and microphone to monitor the driving environment in real time. A camera mounted above the dashboard captures a live video stream, which is analyzed by computer vision models to detect traffic lights and traffic signs, a GPS keeps track of driving speed to ensure the vehicle stays within the limit, and an IMU monitors rotational movements. In parallel, a microphone listens for turn-signal clicking patterns, which are detected by an audio classification model to detect whether the car's blinker was on during turns. These signals are fused by the system to identify potentially dangerous driving situations. When a traffic light changes color, a relevant traffic sign appears, the vehicle goes over the speed limit, or the vehicle turns without signaling, the system immediately provides a voice alert. These alerts consist of short, actionable spoken messages, such as "Red light detected. Please slow down." or "You are above the detected speed limit." Such real-time reminders help drivers avoid common and preventable mistakes. Compared to passive dashcams and post-incident review tools, this approach is proactive and instructional, explicitly recognizing traffic controls in the roadway scene and alerting drivers in real time. It provides a practical, low-cost way for student drivers to develop safer driving habits and build confidence behind the wheel.

In Experiment A, we tested a key blind spot: how reliably the system detects and interprets traffic signs in real driving footage, since incorrect readings could produce unsafe feedback. We recorded road video clips under daytime, sunset, and nighttime lighting conditions, manually labeled each clip with ground-truth traffic-sign classes and speed limit values, and then ran the full pipeline, trained a YOLO-based traffic-sign recognition model followed by speed-limit OCR on frames sampled at fixed intervals. The most significant finding was that end-to-end performance was strongest under daylight conditions and dropped noticeably at night, largely because OCR accuracy degraded as a result of headlight glare, reflective sign materials, motion blur, and lower sign visibility at distance.

In Experiment B, we evaluated the turn-signal audio detector's reliability across different vehicles and noise environments. Audio recordings were collected for each vehicle under silence, music, conversation, and combined music + conversation environments. Model predictions ("blinker on" or "blinker off") were aligned with labeled ground-truth timelines, and F1-scores were computed for each noise condition. Performance was highest in quiet cabins and degraded as background noise increased, driven by masking effects and variations in turn-signal click timbre across vehicle models.

2. CHALLENGES

In order to build the system, a few challenges have been identified as follows.

2.1. Balancing Accuracy and Alert Reliability

The major challenge is ensuring that the perception models are accurate enough to assist the driver rather than cause distraction. If the system often fails to detect traffic signals, then it wouldn't be effective; conversely, frequent and possibly inaccurate alerts can become annoying and potentially unsafe. To improve the accuracy of traffic-light and traffic-sign detections, we could begin by selecting a baseline model designed for detecting small, dynamic objects. The training dataset could then combine locally recorded driving footage with curated public datasets to incorporate regional traffic control variations and capture a wider range of environmental conditions. Model accuracy could be further enhanced through structured evaluation on diverse test sets with data augmentation, careful tuning of detection thresholds, and the incorporation of temporal logic that requires consistent detections across multiple consecutive frames before confirming a detection. For turn-signal detection, audio classification could be cross-checked with IMU based turning estimates so that prompts are issued only when multiple signals agree, reducing false positives.

2.2. Real-Time Performance on Limited Hardware

Another challenge is meeting real-time performance constraints on limited hardware. Running deep learning inference on every video frame while simultaneously monitoring multiple hardware can overload a Raspberry Pi, causing latency that reduces the effectiveness of real-time feedback. To mitigate this, the system can separate expensive inference workloads from the user interface by using worker threads or asynchronous processing, ensuring the user interface remains responsive. Video processing can be rate-limited by sampling frames at a fixed interval, while audio can be analyzed in short windows to preserve sensitivity to turn-signal clicks. Models can also be optimized through resolution adjustments and confidence-threshold tuning to balance accuracy with throughput.

2.3. Sensor Fusion

The system integrates multiple hardware components: the camera streams live video, the microphone captures audio signatures, the GPS and IMU modules provide continuous vehicle-motion data, and the speaker outputs voice alerts. Consequently, another challenge is efficiently handling the data from these components promptly and simultaneously. To address this challenge, dedicated controllers and managers could be implemented for each component to ensure they function reliably and cohesively within the overall system.

3. SOLUTION

The system is developed in Python and consists of three main components: (1) the AI-model component, (2) the hardware integration component, and (3) the real-time processing component.

We chose the YOLOv8 framework to train our traffic-light and traffic-sign detection models. YOLOv8 [11] was selected because it is a single-stage detector that achieves high accuracy in

detecting small, dynamic objects while operating at real-time speeds, which is critical for driving safety systems. Given the inherent complexity of traffic lights, each containing multiple bulbs, with each bulb potentially showing one of three colors (red, green, or yellow) and indicating one of three directions (straight, left-turn, or right-turn), we adopted a two-stage approach [6] to improve detection accuracy. The first model (best_light.pt) was trained to detect complete traffic lights, while the second model (best_bulb.pt) was trained to identify individual illuminated bulbs from cropped images of detected traffic lights, determining their color and direction. The third model (best_sign.pt) was trained to detect traffic signs. Additionally, we trained an audio classification CNN (turn_signal_cnn.pt) to detect the presence of repetitive turn signal clicks [7].
Upon system initialization, the main program launches two worker threads to manage device-specific tasks concurrently. One thread continuously monitors the microphone and applies the turn_signal_cnn.pt model to detect turn-signal sounds, reporting real-time results to the main program. The second thread continuously reads alert messages from the message queue and uses the Piper TTS engine to deliver natural-sounding voice alerts to the driver.

Meanwhile, the main processing thread continuously captures live videos from the camera but analyzes only every N-th frame to maintain computational efficiency. Traffic lights are detected using the best_light.pt and best_bulb.pt models, while traffic signs are detected using the best_sign.pt model. When a speed-limit sign is detected, the EasyOCR model is applied to extract the numerical speed limit value [8]. In parallel, a GPS module monitors the vehicle's current speed [9], and an IMU monitors turning behavior [10].

Inputs from all perception and sensor modules are fused within a decision layer to generate alerts. These alerts are pushed into the message queue, which is consumed by the second thread to enforce ordered, non-overlapping playback. Alerts are issued under the following conditions:

(1) a turning maneuver is detected without the detection of corresponding turn-signal audio;
(2) the vehicle's speed exceeds an extracted speed limit value; or
(3) a traffic-light color change or a "stop", "stopAhead", "yield", or "yieldAhead" sign is detected and confirmed through consistent detections across multiple consecutive frames.

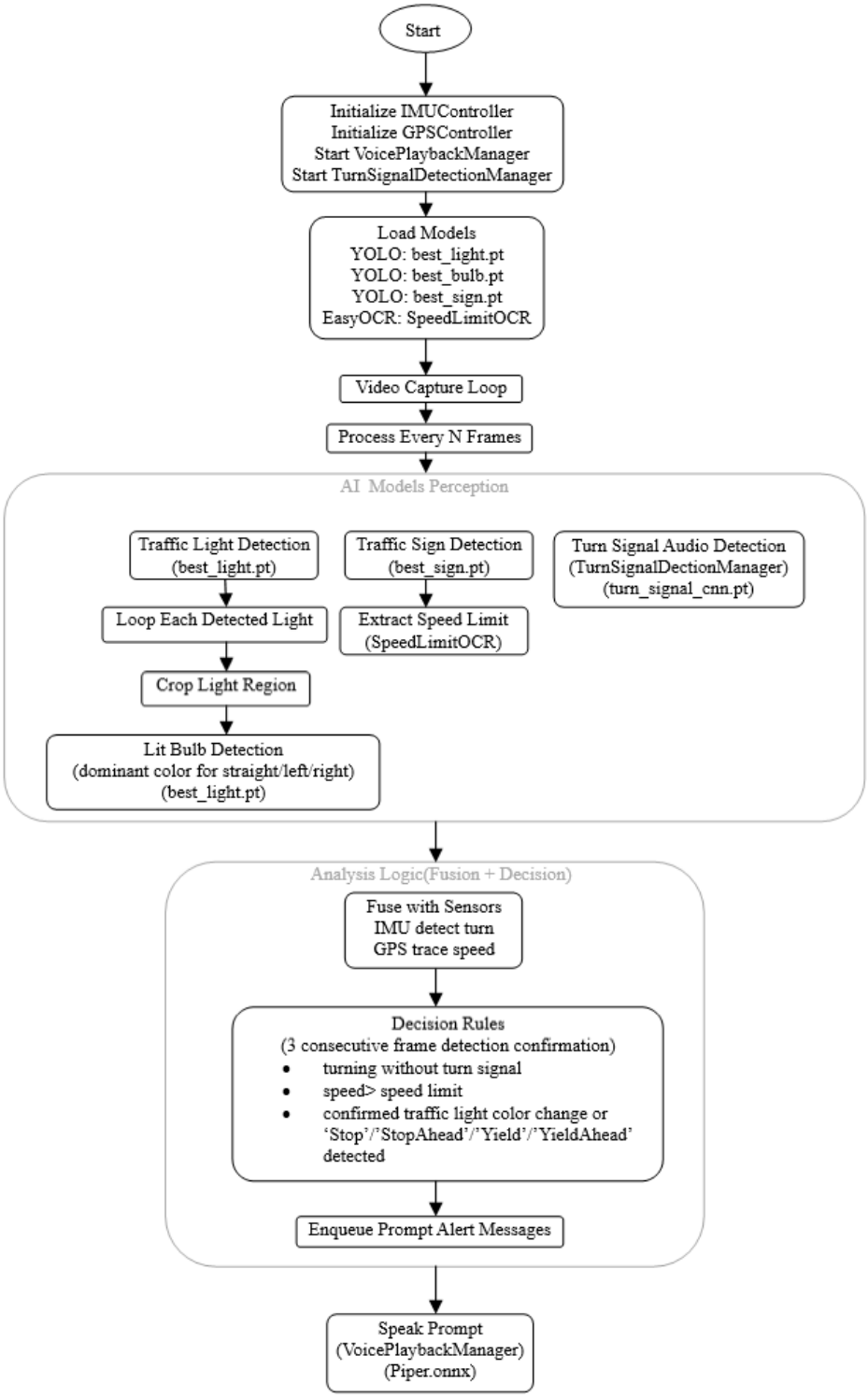


Figure 1. System Diagram

The first component of the system is AI models for detecting traffic lights, traffic signs, and turning signals. We adopted the pretrained YOLOv8 framework to train computer vision models for traffic-light and traffic-sign detection, and we developed an audio classification CNN using Mel-spectrogram features to detect the presence of a turn-signal sound.

To prepare datasets for traffic-light and traffic-sign detection, we manually recorded dashcam videos of real-world driving scenes. Because traffic lights are temporal objects whose appearances persist across consecutive frames, we split entire videos [12], rather than individual frames, into training, validation, and test sets to prevent data leakage and reduce overfitting.

```python
# ==============================
# Image Model
# ==============================

# Train a YOLOv8 model with given dataset and parameters.
def train_yolo_model(best_model_save_to_path,
                     yaml_path, project,
                     epochs=100, batch=32, imgsz=640,
                     save_period=10, patience=20, close_mosaic=10,
                     cos_lr=True, device=0, workers=2, cache=True, seed=0,
                     hsv_h=0.015, hsv_s=0.7, hsv_v=0.4,
                     degrees=0.0, translate=0.1, scale=0.5, shear=0.0, perspective=0.0,
                     flipud=0.0, fliplr=0.5,
                     bgr=0.0, mosaic=1.0, mixup=0.0, cutmix=0.0):

    # Load a pretrained YOLOv8 model
    model = YOLO("yolov8n.pt")

    # Train the model
    model.train(data=yaml_path, project=project, ···

    # Copy best.pt to specified path
    best_model_path = os.path.join(os.path.join(project, os.listdir(project)[-1]), "weights", "best.pt")
    os.makedirs(os.path.dirname(best_model_save_to_path), exist_ok=True)
    shutil.copy(best_model_path, best_model_save_to_path)


def main():
    # Train traffic light model
    l_yaml = "dataset/traffic_light/data.yaml"
    l_results = "results/traffic_light"
    l_best_model_save_to_path = "models/image/best_light.pt"
    train_yolo_model(best_model_save_to_path=l_best_model_save_to_path,
                     data_yaml=l_yaml, project=l_results,
                     hsv_h=0.0, hsv_s=0.4,
                     degrees=5, scale=0.3,
                     fliplr=0.2,
                     mosaic=0.5)

    # Train traffic bulb model
    b_yaml = "dataset/traffic_bulbs/data.yaml"
    b_results = "results/traffic_bulbs"
    b_best_model_save_to_path = "models/image/best_bulb.pt"
    train_yolo_model(best_model_save_to_path=b_best_model_save_to_path,
                     data_yaml=b_yaml, project=b_results,
                     hsv_h=0.0, hsv_s=0.4,
                     scale=0.15,
                     fliplr=0.0,
                     mosaic=0.2)

    # Train traffic sign model
    s_yaml = "dataset/traffic_signs/data.yaml"
    s_results = "results/traffic_signs"
    s_best_model_save_to_path = "models/image/best_sign.pt"
    train_yolo_model(best_model_save_to_path=s_best_model_save_to_path,
                     data_yaml=s_yaml, project=s_results,
                     hsv_h=0.0, hsv_s=0.2, hsv_v=0.2,
                     degrees=5, scale=0.3,
                     fliplr=0.0,
                     mosaic=0.2)


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

Figure 2. Python code for training the traffic-light and traffic-sign detection models (best_light.pt, best_bulb.pt and best_sign.pt)

```python
# ==============================
# Audio Model
# ==============================

# Create CNN architecture
class SmallAudioCNN(torch.nn.Module):
    def __init__(self, n_mels: int, n_classes: int):···

    def forward(self, x):···


# Create Dataset Class
class TurnSignalSoundDataset(Dataset):
    def __init__(self, root_dir, cfg, labels):···

    def __len__(self):···

    def __getitem__(self, idx):···


def main():
    # Define Configuration
    cfg = {···

    labels = ["noise", "turn"]

    # Create datasets and loaders
    train_dataset = TurnSignalSoundDataset("data/train", cfg, labels)
    val_dataset = TurnSignalSoundDataset("data/val", cfg, labels)

    train_loader = DataLoader(train_dataset, batch_size=cfg["batch_size"], shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=cfg["batch_size"])

    # Initialize model
    n_classes = len(labels)
    model = SmallAudioCNN(cfg["n_mels"], n_classes).to(cfg["device"])

    # Loss and optimizer
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=cfg["learning_rate"])

    # Train and validate
    for epoch in range(cfg["epochs"]):···

    # Save model to specified path
    os.makedirs("models/sound", exist_ok=True)
    torch.save({
        "model": model.state_dict(),
        "labels": labels,
        "cfg": cfg
    }, os.path.join("models", "sound", "turn_signal_cnn.pt"))

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

Figure 3. Python code for training the turn signal audio classification model (turn_signal_cnn.pt)

In the Figure 2 code snippet, we defined a wrapper function called train_yolo_model() to encapsulate the YOLOv8 training API. This function initializes a pretrained YOLOv8 Nano model (yolov8n.pt), trains it on a specified dataset with configurable hyperparameters, data augmentation strategies, and automatically saves the best-performing model checkpoint to a designated location. Using this function, we trained three image recognition models: best_light.pt, best_bulb.pt and best_sign.pt. The best_light.pt model, which detects traffic lights, contains a single class (“trafficLight”). The best_bulb.pt model, which identifies individual illuminated bulbs, includes nine classes (“green”, “greenLeft”, “greenRight”, “red”, “redLeft”, “redRight”, “yellow”, “yellowLeft”, “yellowRight”). The best_sign.pt model, which detects traffic signs, includes five classes (“stop”, “stopAhead”, “yield”, “yieldAhead”, “speedLimit”).

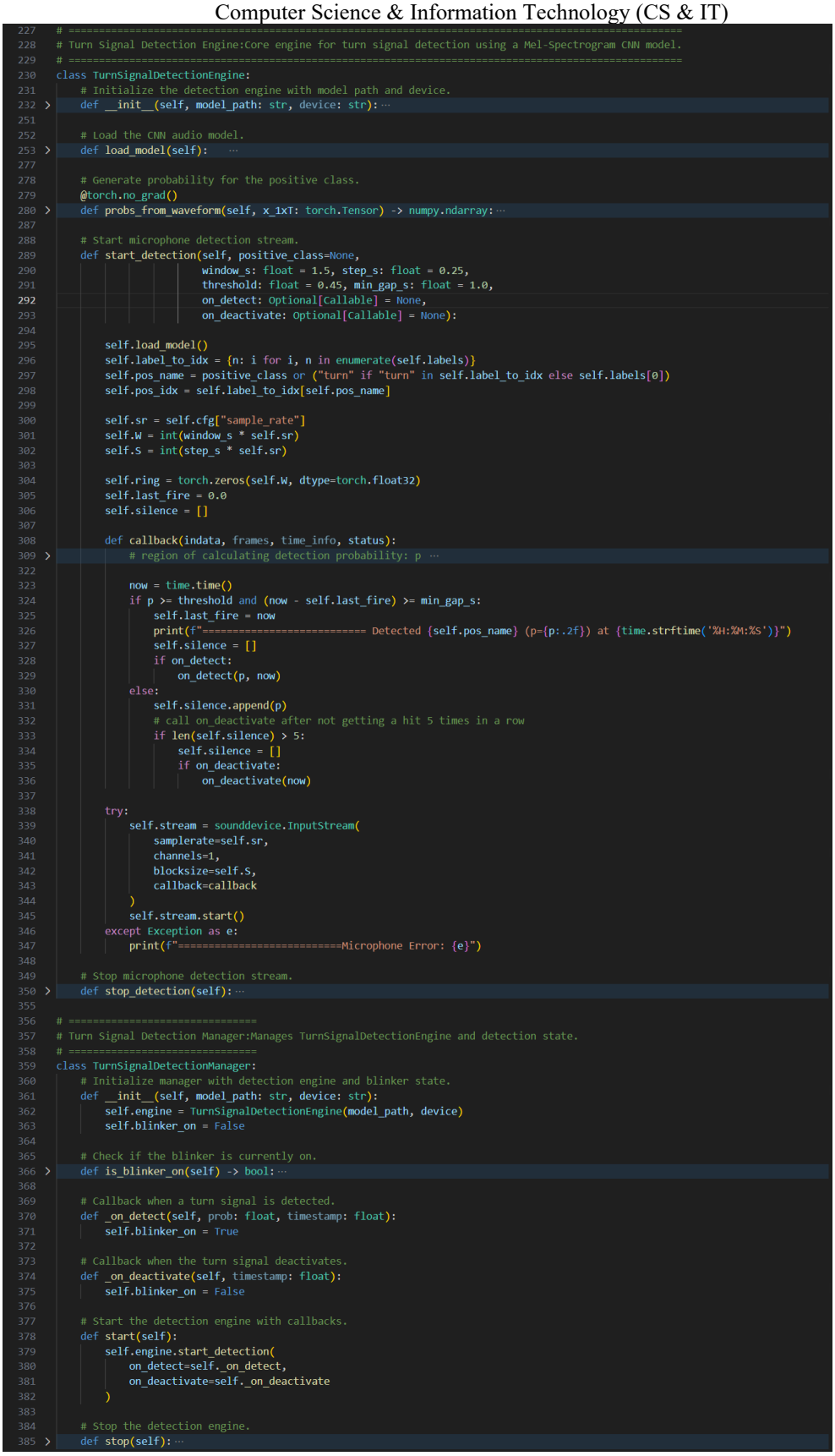
In the Figure 3 code snippet, to train an audio classification model for detecting turn signal sounds, we first developed the SmallAudioCNN class to define the model architecture, which operates on Mel-spectrogram representations extracted from audio. We then created the TurnSignalSoundDataset class to load .wav files, convert them into Mel spectrograms, and prepare them for training. Finally, we

trained the model using predefined hyperparameters, and the best-performing checkpoint was saved as turn_signal_cnn.pt at the specified location.

The second component is the hardware management component, responsible for coordinating the multiple devices required for smooth operation. Controllers were developed for the GPS and IMU modules to continuously track driving speed and vehicle turns. A multithreading mechanism is used, allowing all components, including the microphone, camera, and speaker, to run in parallel. For voice alerts, the Piper TTS engine converts text-based commands into natural-sounding speech, providing real-time guidance to assist the driver.

```python
# ============================================================
# TTS Voice Engine: Core text-to-speech engine using PiperVoice
# ============================================================
class TTSVoiceEngine:
    # Initialize TTS engine with model path.
    def __init__(self, model_path: str):···

    # Convert text to speech, save as WAV, and play it.
    def speak_to_file(self, text: str, filename: str = "misc/temp.wav"):···

    # Convert text to speech and play directly from memory.
    def speak_raw(self, text: str):
        buf = io.BytesIO()
        with wave.open(buf, "wb") as wav_buf:
            self.voice.synthesize(text, wav_buf)
        buf.seek(0)
        player = "afplay" if sys.platform == "darwin" else "aplay"
        subprocess.run([player, "-"], input=buf.read(), check=True)

# ================================================================================================
# Voice Playback Manager: Manages TTSVoiceEngine and queued playback in a background thread.
# ================================================================================================
class VoicePlaybackManager:
    # Initialize voice playback manager with a TTS engine and worker thread.
    def __init__(self, model_path: str):
        self.voice_queue = queue.Queue()
        self.voice_engine = TTSVoiceEngine(model_path)
        self.worker_thread = threading.Thread(target=self._voice_worker, daemon=True)

    # Add text to the voice queue for playback.
    def speak_queued(self, text: str):
        self.voice_queue.put(text)

    # Continuously process queued text and play speech.
    def _voice_worker(self):
        while True:
            text = self.voice_queue.get()
            if text is None:
                break
            self.voice_engine.speak_raw(text)
            self.voice_queue.task_done()

    # Start the voice worker thread.
    def start(self):
        self.worker_thread.start()

    # Stop the voice worker thread and clean up resources.
    def stop(self):···
```

Figure 4. Python code for TTSVoiceEngine and VoicePlaybackManager



```python
# ============================================================================================================
# Turn Signal Detection Engine:Core engine for turn signal detection using a Mel-Spectrogram CNN model.
# ============================================================================================================
class TurnSignalDetectionEngine:
    # Initialize the detection engine with model path and device.
    def __init__(self, model_path: str, device: str):···

    # Load the CNN audio model.
    def load_model(self):    ···

    # Generate probability for the positive class.
    @torch.no_grad()
    def probs_from_waveform(self, x_1xT: torch.Tensor) -> numpy.ndarray:···

    # Start microphone detection stream.
    def start_detection(self, positive_class=None,
                        window_s: float = 1.5, step_s: float = 0.25,
                        threshold: float = 0.45, min_gap_s: float = 1.0,
                        on_detect: Optional[Callable] = None,
                        on_deactivate: Optional[Callable] = None):

        self.load_model()
        self.label_to_idx = {n: i for i, n in enumerate(self.labels)}
        self.pos_name = positive_class or ("turn" if "turn" in self.label_to_idx else self.labels[0])
        self.pos_idx = self.label_to_idx[self.pos_name]

        self.sr = self.cfg["sample_rate"]
        self.W = int(window_s * self.sr)
        self.S = int(step_s * self.sr)

        self.ring = torch.zeros(self.W, dtype=torch.float32)
        self.last_fire = 0.0
        self.silence = []

        def callback(indata, frames, time_info, status):
            # region of calculating detection probability: p ···

            now = time.time()
            if p >= threshold and (now - self.last_fire) >= min_gap_s:
                self.last_fire = now
                print(f"========================== Detected {self.pos_name} (p={p:.2f}) at {time.strftime('%H:%M:%S')}")
                self.silence = []
                if on_detect:
                    on_detect(p, now)
            else:
                self.silence.append(p)
                # call on_deactivate after not getting a hit 5 times in a row
                if len(self.silence) > 5:
                    self.silence = []
                    if on_deactivate:
                        on_deactivate(now)

        try:
            self.stream = sounddevice.InputStream(
                samplerate=self.sr,
                channels=1,
                blocksize=self.S,
                callback=callback
            )
            self.stream.start()
        except Exception as e:
            print(f"==========================Microphone Error: {e}")

    # Stop microphone detection stream.
    def stop_detection(self):···

# ==============================
# Turn Signal Detection Manager:Manages TurnSignalDetectionEngine and detection state.
# ==============================
class TurnSignalDetectionManager:
    # Initialize manager with detection engine and blinker state.
    def __init__(self, model_path: str, device: str):
        self.engine = TurnSignalDetectionEngine(model_path, device)
        self.blinker_on = False

    # Check if the blinker is currently on.
    def is_blinker_on(self) -> bool:···

    # Callback when a turn signal is detected.
    def _on_detect(self, prob: float, timestamp: float):
        self.blinker_on = True

    # Callback when the turn signal deactivates.
    def _on_deactivate(self, timestamp: float):
        self.blinker_on = False

    # Start the detection engine with callbacks.
    def start(self):
        self.engine.start_detection(
            on_detect=self._on_detect,
            on_deactivate=self._on_deactivate
        )

    # Stop the detection engine.
    def stop(self):···
```

Figure 5. Python code for TurnSignalDetectionEngine and TurnSignalDetectionManager

In this component, we developed six classes: IMUController, GPSController, TTSVoiceEngine, VoicePlaybackManager, TurnSignalDetectionEngine, and TurnSignalDetectionManager to control and manage the IMU module, GPS module, speaker, and microphone.

Among them, IMUController reads gyroscope and accelerometer data from the MPU6050 module to detect rotational acceleration, while GPSController retrieves the vehicle's speed from a GPS module.

TTSVoiceEngine is a core text-to-speech engine built on the Piper voice model, which is specified via the model path parameter in the __init__() method. It uses the subprocess.run() command to call the system audio player and play the synthesized speech through the speaker. VoicePlaybackManager manages TTSVoiceEngine and queued voice playback in a background thread. The speak_queued() method adds text to the queue. The _voice_worker() method processes queued text and calls the speak() method of the class's TTSVoiceEngine instance to convert text messages into speech to playback. The start() method launches a daemon thread to run _voice_worker asynchronously in the background.

The TurnSignalDetectionEngine class serves as the core engine for real-time turn signal audio detection. Its start_detection() method first calls the load_model() method to load the trained turn_signal_cnn.pt model, which is passed via the model_path parameter in the __init__() method, and then defines a real-time audio callback() method that is automatically invoked whenever a new audio block is captured from the microphone. This callback converts the audio into a Mel spectrogram and triggers the caller's on_detect() callback when the confidence of the turn-signal model exceeds a specified threshold, or the caller's on_deactivate() callback after several consecutive silent intervals. Finally, it calls the sound device.InputStream() method to start the microphone stream in a separate thread using the following class.

TheTurnSignalDetectionManager class manages the detection engine and its state by encapsulating an instance of TurnSignalDetectionEngine. The is_blinker_on() method returns the current turn signal state. The start() method initializes the detection process by invoking the start_detection() method of the class's TurnSignalDetectionEngine instance, passing two callback functions: _on_detect, which sets the blinker_on flag to "True" upon detection of a turn signal sound, and _on_deactivate, which resets the flag to "False" when the model no longer detects the sound.

How these classes are invoked in the main program to operate various hardware (IMU, GPS, speaker, microphone, and camera) will be explained in the section about the real-time processing component.

The third component is the real-time processing component, which provides a comprehensive implementation of a driver-assistance system. It captures live video, detects traffic lights and traffic signs using trained AI models, tracks which traffic controls are currently active, and fuses visual inputs with sensor inputs (GPS speed, IMU turning, turn-signal audio) to determine when voice alerts should be enqueuer. In addition, the system applies multi-frame confirmation to smooth noisy detections and retains previously confirmed detection states to prevent redundant or overlapping alerts.

Figure 6. Real-time visual feedback during operation

```python
# ====================================
# Main function
# ====================================
def main():

    imu_controller = IMUController()
    gps_controller = GPSController()
    turn_signal_detection_manager = TurnSignalDetectionManager("models/sound/turn_signal_cnn.pt", "cpu")
    voice_playback_manager = VoicePlaybackManager("models/voice/piper.onnx")

    print("==========Starting turn signal detection engine...")
    turn_signal_detection_manager.start()

    print("==========Starting voice playback engine...")
    voice_playback_manager.start()

    # Initialize traffic lights and signs to false
    traffic_control_flag = {
        "RedTrafficLight": False,
        "RedLeftTurnTrafficLight": False,
        "RedRightTurnTrafficLight": False,
        "StopTrafficSign": False,
        "StopAheadTrafficSign": False,
        "YieldTrafficSign": False,
        "YieldAheadTrafficSign": False,
        "SpeedLimitTrafficSign": False}

    traffic_control_flag_history = deque(maxlen=3)
    for _ in range(3):
        traffic_control_flag_history.append(traffic_control_flag.copy())

    last_confirmed_traffic_control_flag = traffic_control_flag.copy()

    # Load YOLO models
    l_model = YOLO('models/image/best_light.pt')
    b_model = YOLO('models/image/best_bulb.pt')
    s_model = YOLO('models/image/best_sign.pt')

    # OCR engine for speed-limit signs
    speed_limit_ocr = SpeedLimitOCR(['en'])

    l_conf_threshold=0.5
    b_conf_threshold=0.5
    s_conf_threshold=0.4
    speed_limit_ocr_conf_threshold = 80
    diff_conf = 0.1

    # Start video stream
    print("==========Starting video capture...")
    cap = cv2.VideoCapture(0)
    count = 0
    PROCESS_EVERY_N = 10

    while cap.isOpened():
        # Read video frame
        ret, frame = cap.read()
        if not ret:
            break

        count += 1
        if (count % PROCESS_EVERY_N== 0):
            # Each iteration, reset current_traffic_control_flag
            current_traffic_control_flag = traffic_control_flag.copy()

            # ==================================
            # 1: Traffic-light Detection via YOLO
            # ==================================
            # Detect traffic lights
            l_results = l_model(frame, verbose=False)

            # For each traffic light
            for l_box in l_results[0].boxes:
                if l_box.conf[0] > l_conf_threshold:

                    # Crop around the detected traffic light
                    l_x, l_y, l_w, l_h = map(int, l_box.xywh[0])
                    cropped_traffic_light_frame = crop_image(frame, (l_x - l_w/2, l_y - l_h/2, l_w, l_h))

                    # Detect bulbs inside the light
                    b_results = b_model(cropped_traffic_light_frame, verbose=False)
                    b_class_names = b_model.names

                    # Determine the dominant color for straight, turn-left, and turn-right directions
                    dominant_color, dominant_color_box_list = analyze_light_bulbs(
                        b_results, b_class_names, b_conf_threshold, diff_conf
                    )

                    if dominant_color.get("forward") == "red":
                        current_traffic_control_flag["RedTrafficLight"] = True
                    if dominant_color.get("left") == "red":
                        current_traffic_control_flag["RedLeftTurnTrafficLight"] = True
                    if dominant_color.get("right") == "red":
                        current_traffic_control_flag["RedRightTurnTrafficLight"] = True
```

```
        # ======================================
        # 2: Traffic-sign Detection via YOLO + OCR
        # ======================================
        # Detect traffic signs.
        s_results = s_model(frame, verbose=False)
        s_class_names = s_model.names

        # Iterate through all detected traffic signs.
        best_speed_limit_conf = 0
        best_speed_limit_value = None
        for s_box in s_results[0].boxes:
            if s_box.conf[0] > s_conf_threshold:
                s_box_name = s_class_names[int(s_box.cls[0].item())]
                match s_box_name:
                    case "stop":
                        current_traffic_control_flag["StopTrafficSign"]=True
                    case "stopAhead":
                        current_traffic_control_flag["StopAheadTrafficSign"]=True
                    case "yield":
                        current_traffic_control_flag["YieldTrafficSign"]=True
                    case "yieldAhead":
                        current_traffic_control_flag["YieldAheadTrafficSign"]=True
                    case _:
                        # Treat as potential speed-limit sign
                        s_x1, s_y1, s_x2, s_y2 = map(int, s_box.xyxy[0])
                        cropped = frame[s_y1:s_y2, s_x1:s_x2]
                        speed_limit_value = speed_limit_ocr.extract_speed_limit(cropped, speed_limit_ocr_conf_threshold)
                        if speed_limit_value is not None:
                            current_traffic_control_flag["SpeedLimitTrafficSign"]=True
                            if (s_box_conf := s_box.conf[0].item()) >= best_speed_limit_conf:
                                best_speed_limit_conf, best_speed_limit_value = s_box_conf, speed_limit_value

        # ==========================================================
        # 3: Add the detection status of current frame to history queue
        # ==========================================================
        traffic_control_flag_history.append(current_traffic_control_flag)

        # ================
        # 4: Voice Playback
        # ================
        # Voice Alert for turn-signal coaching
        if not turn_signal_detection_manager.is_blinker_on() and imu_controller.detect_turn() != "Straight":
            voice_playback_manager.speak_queued("Do not forget to use your turn signal when turning! ")

        # Voice Alert for speeding coaching
        if (current_traffic_control_flag["SpeedLimitTrafficSign"] and
            not last_confirmed_traffic_control_flag["SpeedLimitTrafficSign"]):
            gps_controller.update()
            driving_mph = gps_controller.get_speed_mph()
            if best_speed_limit_value is not None and driving_mph > best_speed_limit_value:
                voice_playback_manager.speak_queued("You are above the detected speed limit. ")
            last_confirmed_traffic_control_flag["SpeedLimitTrafficSign"] = current_traffic_control_flag["SpeedLimitTrafficSign"]

        # Voice Alert for traffic-control coaching with 3-frame confirmation
        for key in current_traffic_control_flag:
            # If the traffic-control is detected in three consecutive frames and was not previously confirmed as detected
            if all(traffic_control_flag_history[i][key] for i in range(3)) and not last_confirmed_traffic_control_flag[key]:
                if key in ("RedTrafficLight", "RedLeftTurnTrafficLight", "RedRightTurnTrafficLight",
                           "StopTrafficSign", "StopAheadTrafficSign", "YieldTrafficSign", "YieldAheadTrafficSign"):
                    voice_playback_manager.speak_queued(f"{key} detected. ")

                last_confirmed_traffic_control_flag[key] = current_traffic_control_flag[key]

            # If the traffic-control is not detected in three consecutive frames and was previously confirmed as detected
            if all(not traffic_control_flag_history[i][key] for i in range(3)) and last_confirmed_traffic_control_flag[key]:
                if key in ("RedTrafficLight", "RedLeftTurnTrafficLight", "RedRightTurnTrafficLight"):
                    voice_playback_manager.speak_queued(f"{key} no longer detected.")

                last_confirmed_traffic_control_flag[key] = current_traffic_control_flag[key]
```

Figure 7. Python code for the real-time system processing pipeline

The main program begins by creating instances of IMUController, GPSController, VoicePlaybackManager, TurnSignalDetectionManager, and EasyOCR. It then calls turn_signal_detection_manager.start() method and voice_playback_manager.start() method to launch separate background threads for turn signal audio detection and voice playback, respectively.

Next, three key data structures are defined. The traffic_control_flag data structure stores the detection status for each traffic-control type, with keys such as “RedTrafficLight”, “RedLeftTurnTrafficLight”, “RedRightTurnTrafficLight”, “StopTrafficSign”, and others. The traffic_control_flag_history maintains the detection status from the most recent three processed frames to avoid single-frame glitches. A detection is considered confirmed only when all three entries in traffic_control_flag_history report the same value. The last_confirmed_traffic_control_flag stores the most recently confirmed detection states to prevent overlapping alerts.

Before entering the video processing loop, the required models for traffic-light and road-sign detection are loaded and cv2.VideoCapture() is called to capture live video input. Within the loop, every N-th frame is processed to balance detection accuracy and real-time performance, and for each processed frame, the system creates a fresh current_traffic_control_flag, initialized as a copy of traffic_control_flag, to store the detection status for that frame. Traffic-light detection is performed first using best_light.pt. Each detected traffic-light bounding box is then cropped and passed to best_bulb.pt to detect individual illuminated bulbs. The analyze_light_bulbs() function is subsequently invoked to determine the dominant bulb color for each direction (straight, left-turn, and right-turn). For example, if the dominant color of the left-turn signal is red, current_traffic_control_flag["RedLeftTurnTrafficLight"] is set to True. Next, the system uses best_sign.pt to detect traffic signs (“stop”, “stopAhead”, “yield”, “yieldAhead”, “speedLimit”). When a “speedLimit” sign is detected, the corresponding image region is cropped and passed to the EasyOCR engine to validate the detection and extract the numerical speed limit value. The current_traffic_control_flag is then updated accordingly.

After all detections are completed for the frame, the traffic_control_flag_history.append() method is called to addcurrent_traffic_control_flag to the history queue.

The system then issues alerts and updates last_confirmed_traffic_control_flag under the following conditions:

(1) The vehicle is detected to be turning while no active turn signal audio is present, as determined by calling turn_signal_detection_manager.is_blinker_on() and imu_manager.detect_turn();
(2) The current vehicle speed, obtained by calling gps_controller.update() method and gps_controller.get_speed_mph() method, exceeds the speed limit detected in the frame, while the corresponding entry in last_confirmed_traffic_control_flag is “False”; or
(3) For a red light or traffic signs (“stop”, “stopAhead”, “yield” and “yieldAhead”), a voice alert is triggered when the object has been detected in three consecutive frames while the corresponding entry in last_confirmed_traffic_control_flag is “False”. For a red light specifically, a voice alert is also triggered when the object has been absent in three consecutive frames while the corresponding entry in last_confirmed_traffic_control_flag is “True”.

The issued alerts are enqueuer via voice_playback_manager.speak_queued() method, and the voice-playback thread retrieves the queued messages and uses the TTS engine to alert the driver.

4. EXPERIMENT

4.1. Experiment 1

We needed to test how accurately the system detects traffic signs in real driving footage, especially under different lighting and distances, because incorrect traffic-sign readings could lead to unsafe feedback.

To evaluate traffic-sign detection performance, we will construct a labeled dataset of road video clips recorded on the Raspberry Pi under daytime, sunset, and nighttime conditions. For each clip, ground-truth annotations will be manually created for the presence and class of traffic signs (“stop”, “stopAhead”, “yield”, “yieldAhead”, and “speedLimit”), as well as the corresponding speed limit values. The full traffic-sign processing pipeline, consisting of the trained YOLO-based traffic-sign recognition model followed by OCR-based speed limit value extraction, will then be applied to frames sampled at fixed intervals, such as every 10th frame. For each annotated traffic sign, we will measure

whether the system correctly detects it and, for each speed-limit sign, whether the extracted numeric value matches the ground truth.

Clip	Lighting	End-to-end Accuracy (%)
D1	Daytime	92
D2	Daytime	90
D3	Daytime	93
D4	Daytime	88
S1	Sunset	86
S2	Sunset	84
S3	Sunset	82
S4	Sunset	85
N1	Nighttime	74
N2	Nighttime	78
N3	Nighttime	71
N4	Nighttime	76

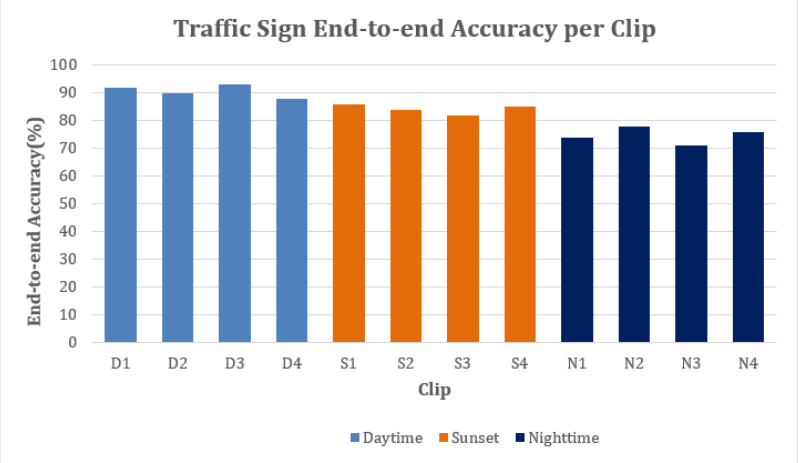


Figure 8. Quantitative comparison of traffic-sign detection across lighting conditions

Across 12 real-world driving clips, the end-to-end traffic-sign pipeline achieved a mean accuracy of 83.25%, with a median of 84.5%. The lowest performance was observed in a nighttime clip (N3) at 71%, while the highest score was 93% in a daytime clip (D3). Performance was strongest under daylight conditions (mean 90.75%) and declined at sunset (mean 84.25%) and at night (mean 74.75%). A notable drop in performance occurred at night relative to sunset. While the traffic-sign recognition model often continued to detect signs correctly, the OCR stage failed more frequently under nighttime conditions due to headlight glare, reflective sign materials, motion blur, and reduced effective resolution when signs were farther away. Because the pipeline requires both correct traffic-sign recognition and correct OCR for a prediction to be counted as correct, OCR failures disproportionately reduce the final score. In practice, lighting and viewing distance had the largest effect on performance, especially for speed-limit signs, where a single misread digit makes the output incorrect.

4.2. Experiment 2

We also wanted to test how reliably the turn signal audio model detects real blinker clicks in different vehicles and noise environments, since missed or false detections could lead to incorrect coaching about turn-signal use.

To evaluate the turn-signal detector, we will record a dataset of driving audio segments using the same microphone setup deployed on the Raspberry Pi. Each segment will be labeled with ground-truth turn-signal events ("signal on" and "signal off"), along with background conditions including silence, conversation, music, and music combined with conversation. The trained turn signal audio model will run continuously on these recordings to generate a binary prediction ("blinker on" or "blinker off") over time. Model predictions will be aligned with the ground-truth timeline, and performance will be evaluated using metrics, such as true positive rate, false positive rate, false negative rate, and F1-score, reported both overall and separately for each noise condition.

Vehicle	Silence	Music	Conversation	Music + Conversation
Vehicle A	0.95	0.90	0.88	0.82
Vehicle B	0.93	0.87	0.85	0.80
Vehicle C	0.91	0.86	0.84	0.78

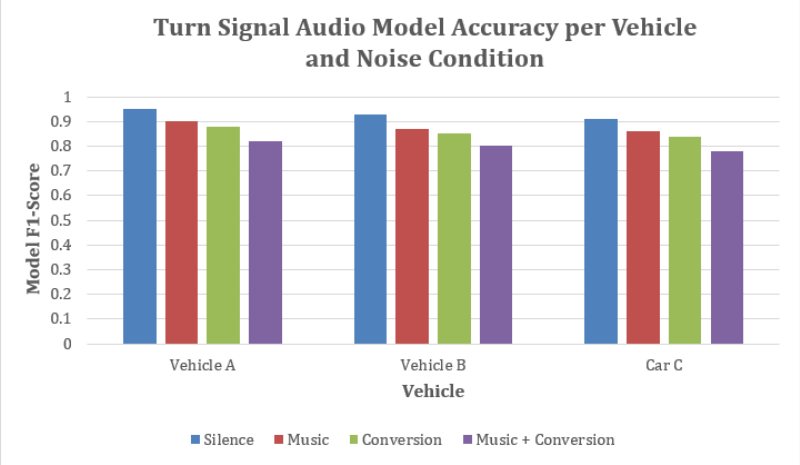


Figure 9. Quantitative comparison of turn signal audio detection across multiple vehicles

Across 12 recordings (3 vehicles × 4 conditions), the turn-signal detector achieved a mean F1-score of 0.866, with a median of 0.865. The lowest result performance 0.78 (Vehicle C with music + conversation), while the highest score was 0.95 (Vehicle A in silence). As expected, performance was best in quiet cabin environments (mean 0.93) and degraded progressively as background noise increased (music mean 0.877, conversation mean 0.857, music+conversation mean 0.80). Error analysis indicates that false positives increased when rhythmic music overlaps the blinker's click cadence, while false negatives became more common when clicks were partially masked by speech or when the microphone gain was reduced to avoid clipping.

Performance differences across vehicles were also observed, reflecting variation in turn-signal click timbre between vehicle models. Consequently, a model trained on one set of clicks may generalize imperfectly. Overall, the ambient noise level was the primary driver of F1-score variation, followed by vehicle-to-vehicle sound variation, suggesting that training on a more diverse set of vehicles and more noisy audio conditions would yield the greatest performance improvements.

5. RELATED WORK

Nine, J. et al.[13] investigate the feasibility of performing traffic-light detection on a Raspberry Pi 3B+ using deep-learning models, such as MobileNetSSD. A camera captures forward-facing video, and the system processes frames to identify traffic lights and classify their states under varying lighting conditions. The study shows that traffic-light detection is achievable on embedded hardware, with acceptable accuracy and frame rates for real-time operation. However, this work does not include traffic-sign detection, sensor fusion, or any form of driver alerting or decision logic. In comparison, our system builds a full driver-assistance pipeline by combining traffic-light and traffic-sign detection with GPS, IMU, and turn signal audio classification, and by providing real-time voice feedback to help drivers respond to potentially unsafe situations.

Isa, I. S. B. M. et al.[14] present a real-time traffic-sign recognition system implemented on a Raspberry Pi 3 using a camera and a TensorFlow-based machine-learning model. The system captures live video, processes frames to detect and classify a limited set of traffic signs, and outputs the recognized sign to the user. Experimental results show detection accuracy above 90% under controlled conditions, demonstrating that low-cost embedded hardware can support basic deep-learning inference for driving applications. However, the system is limited to traffic-sign detection only and does not recognize traffic lights. It also relies exclusively on visual data, without integrating GPS, IMU, or audio inputs. In contrast, our system extends this approach by detecting both traffic signs and traffic lights, integrating multiple sensor modalities, and issuing voice alerts to assist drivers proactively.

Setiawan, Y. et al.[15] propose a traffic-sign detection system based on the YOLO object-detection framework that provides notifications when traffic signs are identified. The system captures live video from a camera, processes each frame using a trained YOLO model, and detects traffic signs in real time to alert the driver. Their results show that YOLO-based detection can achieve reliable real-time performance, demonstrating the suitability of deep-learning models for traffic-sign recognition. Similar to Methodology B, the system is limited to traffic-sign detection and does not incorporate multimodal sensor fusion. Although it provides notification to drivers compared to Methodology B, the feedback is delivered through an LCD, a vibration motor, and a radio frequency (RF) module; in contrast, our system delivers eye-free, descriptively phrased voice alerts to better support safer driving behavior.

6. CONCLUSIONS

Although this driving safety system demonstrates that low-cost hardware, such as Raspberry Pi, can provide effective real-time coaching for novice drivers, several limitations remain in the current system. First, the datasets used for traffic-light and traffic-sign detections are limited in size and variety, which reduces detection accuracy, particularly under challenging conditions. To improve reliability, post-processing methods are applied, including using OCR to extract numerical speed limit values and applying multi-frame smoothing to reduce noisy detections. However, these techniques increase processing time and introduce additional latency. In addition, the accuracy of the turn-signal audio detection model can be affected by loud music or high cabin noise, which may lead to missed or incorrect detections. With additional development time, future work would expand the datasets to include not only a larger number of images but also data collected under diverse lighting conditions, varying weather conditions, and a wider range of vehicle interiors.

Second, the current voice feedback relies on relatively simple TTS, which may become repetitive and reduce user engagement over time; future versions could incorporate higher-quality neural TTS and more varied phrasing while maintaining concise alerts.

Finally, the current Raspberry Pi prototype remains physically bulky. A more product-ready design would integrate system components into a smaller enclosure and explore more power-efficient hardware or dedicated accelerators to reduce overall size, heat, and power consumption.

The system demonstrates that combining computer vision, audio analysis, and multi-sensor fusion can turn a Raspberry Pi into an effective coaching tool for novice drivers. By providing clear, real-time audio feedback on traffic lights, traffic signs, and turn-signal usage, the system aims to help novice drivers develop safer driving habits from their earliest miles on the road.

REFERENCES

[1] Road safety. (n.d.). www.who.int. https://www.who.int/health-topics/road-safety
[2] CDC. (2024, July 11). Teen Drivers. Teen Drivers. https://www.cdc.gov/teen-drivers
[3] Bumper. (2025, September 29). Teenage Driving Accidents Keep Growing. Here's Where Young Drivers Are Most at Risk. Bumper.com. https://www.bumper.com/analysis/teen-driving-accidents-keep-growing/
[4] NHTSA. (2023). NHTSA. https://www.nhtsa.gov/risky-driving/speeding/
[5] Huang, K., Shi, B., Li, X., Li, X., Huang, S., & Li, Y. (2022). Multi-modal sensor fusion for auto driving perception: A survey. arXiv preprint arXiv:2202.02703.
[6] Lin, H. Y., Lin, S. Y., &Tu, K. C. (2024). Traffic Light Detection and Recognition Using a Two-Stage Framework From Individual Signal Bulb Identification. IEEE Access.
[7] Marchegiani, L., & Newman, P. (2022). Listening for sirens: Locating and classifying acoustic alarms in city scenes. IEEE transactions on intelligent transportation systems, 23(10), 17087-17096
[8] Reddy, P. P., Shruthi, P. S., Himanshu, P., & Singh, T. (2024, June). License plate detection using yolo v8 and performance evaluation of easyocr, paddleocr and tesseract. In 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT) (pp. 1-6). IEEE.
[9] Khan, M. A., & Khan, S. F. (2018, April). IoT based framework for Vehicle Over-speed detection. In 2018 1st International Conference on Computer Applications & Information Security (ICCAIS) (pp. 1-4). IEEE.
[10] Liu, X., Mei, H., Lu, H., Kuang, H., & Ma, X. (2017). A vehicle steering recognition system based on low-cost smartphone sensors. Sensors, 17(3), 633.
[11] Jocher, G. (2020, May 18). YOLO Documentation. Docs.ultralytics.com. https://docs.ultralytics.com/
[12] LISA Traffic Light Dataset. (n.d.). LISA traffic light dataset [Data set]. Kaggle. https://www.kaggle.com/datasets/mbornoe/lisa-traffic-light-dataset
[13] Nine, J., &Mathavan, R. (2021). Traffic Light and Back-light Recognition using Deep Learning and Image Processing with Raspberry Pi. Embedded Selforganising Systems, 8(2), 15-19.
[14] Isa, I. S. B. M., Choy, J. Y., &Shaari, N. L. A. B. M. (2022). Real-time traffic sign detection and recognition using Raspberry Pi. International Journal of Electrical and Computer Engineering, 12(1), 331.
[15] Setiawan, Y., Puji, M. N., &Astuti, W. (2024, November). Traffic Signs Detection System Using YOLO (You Only Look Once) That Provides Notification. In 2024 IEEE International Conference on Smart Mechatronics (ICSMech) (pp. 95-100). IEEE.
[16] MDokuch. (2024, November 4). Mobile Telematics in Driver Education: Implementing Teen Drivers Safety in Driving Schools. Damoov. https://damoov.com/mobile-telematics-in-driver-education-teen-drivers-safety/
[17] Papatheocharous, E., Kaiser, C., Moser, J., & Stocker, A. (2023). Monitoring distracted driving behaviours with smartphones: an extended systematic literature review. Sensors, 23(17), 7505.
[18] Ondruš, Ján, et al. "Smartphone Sensors in Motion: Advancing Traffic Safety with Mobile Technology." Applied Sciences 14.13 (2024): 5404.
[19] Wu, Minglin, Sheng Zhang, and Yuhan Dong. "A novel model-based driving behavior recognition system using motion sensors." Sensors 16.10 (2016): 1746.