1. Opening page

Electrical Engineering

Project Name: Real-Time License Plate Recognition System Using FPGA

Project Chapter Document Abstract

Student Name: Dar Eshel Epstein

Supervisor's Name: Binyamin Abramov

Initiator/Clinical Staff: Binyamin Abramov and Dar Eshel Epstein

2. Table of Contents

3. Abstract

This project focuses on the creation of a real-time license plate recognition system that employs Field-Programmable Gate Arrays (FPGAs) for enhanced hardware acceleration. The system is designed to provide accurate and efficient license plate detection and processing in real-time, suitable for automotive and surveillance applications.

Emphasizing hardware optimization and software integration, this project introduces a hybrid approach to real-time license plate recognition, integrating the computational prowess of FPGAs with advanced software methodologies.

4. Introduction

License plate recognition (LPR) technology is crucial for applications such as traffic management, law enforcement, parking control, and toll collection. However, conventional LPR systems, typically software-based and running on general-purpose computers, often face challenges such as slow processing speeds and high operational costs. To address these issues, this project proposes a novel real-time LPR system that utilizes field-programmable gate arrays (FPGAs), renowned for their parallel processing capabilities and high-speed image processing.

The development process begins with software preprocessing to ensure optimal image quality for recognition. Subsequently, the hardware capabilities of FPGAs are harnessed for efficient and accurate character recognition during the OCR (Optical Character Recognition) phase. This system represents a cohesive integration of software and hardware, meticulously tailored to meet the high-performance demands of LPR applications. The paper will detail the design, implementation, and experimental results, with a focus on the system's optimization and its potential to significantly enhance intelligent transportation systems.

5.1. Goals Objectives and Measures

5.1.1. Goals

- The main goal of this project is to design and implement a system that can process and recognize license plate numbers from a camera in real time.
- The system should be able to capture and analyze images of license plates from different angles, distances, lighting conditions and backgrounds.
- The system should be able to extract the license plate, segment the characters and identify the number using optical character recognition (OCR) algorithms in real time.
- The system should be able to display the recognized license plate number on a screen or store it in a database for further use.

5.1.2. Objectives

- Review the existing literature and methods for license plate recognition.
- Select the appropriate hardware platform, camera and FPGA board for the system.
- Design and implement the preprocessing for the OCR algorithm that will run in real-time.
- Design and implement the OCR algorithms in real-time using hardware description languages (HDL).
- Test and evaluate the system performance and accuracy using various license plate images and videos.
- Compare and analyze the advantages and disadvantages of the hardware-based system versus the software-based system.
- Document and present the project results and findings.

5.1.3. Measures

In the formulation of performance targets for the real-time license plate recognition system, reference was made to industry-standard guidelines, as outlined in [1]. These guidelines offer a benchmark for the desired operational parameters of LPR systems, ensuring that the project's goals are both challenging and achievable. Table 1 presents the key performance metrics and their targets, as adapted from the recommendations provided in [1]:

Performance Metric	**Unit**	**Target**
Processing Speed	ms	100
Recognition Accuracy	%	At least 85
Camera distant from plat	Meters	3-6
Angle if the camera	degrees	Up to 30 from plate
Illumination for LPR	Lux	Minimum 50 (Dimly lit room)

Table 1

Environmental Adaptation Disclaimer:

While this license plate recognition system is designed with robustness in mind, it is important to note that extreme environmental conditions may impact its performance. The deployment of cameras and preprocessing algorithms has been optimized to enhance system reliability under a variety of conditions. However, absolute performance under all possible environmental scenarios cannot be guaranteed (heavy rain, sandstorm, heavy fog and similar extreme environmental scenarios).

5.2. Alternative's solution

In addition to the proposed real-time license plate recognition system, several alternative solutions were considered:

- **NVIDIA's AI Models [2]**:
  Utilizing NVIDIA's cutting-edge AI models provides a robust framework for LPR systems. NVIDIA's technology is known for its high processing power and efficiency, which can significantly enhance image recognition tasks. However, they require specific NVIDIA hardware, which may limit their applicability in certain environments.
- **Platerecognizer.com Technology [3]**:
  Platerecognizer.com provides cloud-based solutions and on-premises software for license plate recognition. Although their on-premises solution is designed to support processing in near real-time, it requires a robust computing environment with substantial processing power to manage the workload efficiently and minimize latency [4].
- **OpenCV and Python [5]**:
  Combining OpenCV with Python is a popular choice for image processing and computer vision projects. This approach allows for the development of custom LPR algorithms that can be tailored to specific requirements. Like Platerecognizer.com's technology, it requires a powerful computer for real-time processing.
- **Embedded Platforms:**
  For a more customized solution, embedded platforms can be employed. These systems often utilize OpenCV and Python for image processing tasks, similar to the methods shown in [4]. Embedded platforms provide the flexibility to design a system that closely aligns with the unique needs of the project.

Each alternative solution has its own set of advantages and limitations, which are detailed in Table 2. The selection of the most appropriate solution will depend on the specific requirements and constraints of the project.

Solution	Pro	Cons
NVIDIA's AI Models [2]	High performance Real-time processing	Requires NVIDIA hardware
Platerecognizer.com Technology [3]	High accuracy Near Real-time processing	Requires streaming of video to a high-performance computing unit for near real-time processing [4].
Embedded Platforms	Low power and portable	Can be challenging to implement complex algorithms
OpenCV and Python [5]	Open-source Flexible	Requires a powerful computer for real-time processing

Table 2

5.3. Engineering challenges

Developing a real-time license plate recognition system involves several engineering challenges that must be carefully navigated to ensure the system's effectiveness and reliability:

- **Hardware Compatibility:**
  A primary challenge is ensuring compatibility between the selected FPGA board and camera. The system must handle the computational demands of real-time processing without bottlenecks, necessitating the selection of hardware that can communicate effectively and process data at the required speeds.

- **Algorithm Optimization:**
  Optimizing algorithms for real-time license plate recognition requires accuracy and efficiency on the FPGA, as well as consideration of the embedded CPU's constraints used for preprocessing. The embedded CPU typically has limited processing power, demanding the development of highly efficient preprocessing routines that minimize computational load while preparing image data for the FPGA.

- **System Integration:**
  Another challenge is integrating the hardware and software components into a cohesive system. The camera, FPGA board, and any additional sensors or modules must work in unison to ensure reliable performance, requiring meticulous planning and testing for seamless system integration.

- **Environmental Conditions:**
  The LPR system must be robust enough to operate reliably under various environmental conditions, including lighting changes, weather conditions, and the dynamics of vehicle speeds and angles. Adapting to these conditions is critical for maintaining system performance.

5. Literature review

6.1. Literature review of solution for LPR systems

License Plate Recognition (LPR) systems are pivotal in intelligent transport systems, providing a means for automated vehicle identification. Recent advancements have focused on enhancing recognition accuracy and adapting to diverse environmental conditions. This literature review examines five significant contributions to the field.

6.1.1. Encoder–Decoder Frameworks for LPR [6]

The Article introduced the EDF-LPR, an encoder–decoder framework that directly detects and recognizes license plate characters without considering the format of the license plate. This approach addresses the challenge of detecting plates with different formats and scales, achieving a detection rate of 99.51% and a recognition rate of 95.3% at about 40 frames per second Test on NVIDIA GeForce GTX 1080 GPU.

6.1.2. Hybrid Edge–Cloud Computing Systems [7]

The Article proposed a light vehicle LPR system that leverages hybrid edge–cloud computing. This system aims to reduce latency and energy consumption by using channel pruning to reconstruct the backbone layer and deploying network models on edge gateways. The result is an impressive accuracy rate of 97% with a total number of parameters of only 0.606 MB.

6.1.3. Layout-independent ALPR Systems [8]

The Article proposed developing an efficient and layout-independent ALPR system based on the YOLO detector. Their system unifies license plate detection and layout classification, achieving an end-to-end recognition rate of 96.9% across eight public datasets. This system outperformed previous works and commercial systems, demonstrating real-time performance even with multiple vehicles in the scene.

6.1.4. Multinational LPR [9]

The Article tackled the challenge of multinational LPR with a system that uses generalized character sequence detection. Their approach, based on YOLO networks, includes steps for LP detection, unified character recognition, and multinational LP layout detection, showcasing its effectiveness across datasets from various countries.

6.1.5. Recognition in Fog-Haze Environments [10]

The Article addressed the difficulty of recognizing license plates in fog-haze environments with their LPRFH method10. Utilizing a dark channel prior algorithm and a convolution-enhanced super-resolution convolutional neural network, their method can accurately recognize license plates even in severe fog-haze conditions.

6.2. Conclusion and Comparison

In conclusion, the comparative analysis presented in the table underscores the diverse methodologies employed to enhance LPR systems. From encoder–decoder frameworks and hybrid computing to layout-independent recognition and adaptability in challenging environmental conditions, each approach contributes uniquely to the field. In table 3. We highlight the strengths and limitations of these systems in relation to an FPGA-based solution, providing a clear perspective on potential areas for further development. As LPR technology continues to evolve, we can anticipate ongoing improvements in accuracy and efficiency, expanding its applicability in real-world scenarios.

Solution	Pros	Cons	Comparison with our proposed system
Encoder–Decoder Frameworks for LPR [6]	High accuracy and fps. Good for various plate formats.	May require substantial computational resources.	The proposed system could incorporate similar techniques on a hardware level of the FPGA for potentially faster processing.
Hybrid Edge–Cloud Computing Systems [7]	Low network size, good accuracy. Less cloud dependency.	Limited by edge device capabilities.	The proposed system offers on-device processing without cloud reliance, potentially more efficient.
Layout-independent ALPR Systems [8]	High accuracy, robust to conditions. Good fps on GPU.	Requires a high-end GPU for best performance.	The proposed system can provide real-time processing with potentially lower hardware requirements.
Multinational LPR [9]	Handles various LP layouts. Quick processing per image.	Required powerful GPU and 24GB RAM to run in Realtime	The proposed system aim to leverage hardware acceleration for enhanced real-time processing efficiency.
Recognition in Fog-Haze Environments [10]	Effective in poor visibility. Utilizes advanced image processing.	Specific to fog-haze conditions.	The proposed system could incorporate similar techniques for robustness in various environments.

Table 3

6.3. Literature Review of Solutions for Object Detection

Efficient object detection is crucial for real-time License Plate Recognition (LPR) systems, especially when deployed on embedded devices with limited computational resources like the dual-core ARM Cortex-A9 MPCore used in our project. This section reviews three prominent object detection models—**MobileNet-SSD [17]**, **YOLOv11n [14]**, and **NanoDet-m [15]**—and examines their suitability based on expected performance on our target hardware.

6.3.1. MobileNet-SSD [17]

MobileNet-SSD combines the lightweight **MobileNet** architecture with the **Single Shot MultiBox Detector (SSD)** framework. MobileNet utilizes depthwise separable convolutions to reduce model size and computation, making it suitable for mobile and embedded applications. Integrated with SSD, it enables object detection in a single pass, achieving real-time performance.

Performance and Application: MobileNet-SSD achieves a balance between speed and accuracy, with a mean Average Precision (mAP) of approximately 68% on datasets like PASCAL VOC. It operates at real-time speeds of about **14 ms per inference** on modern ARM processors like the Pixel [18].

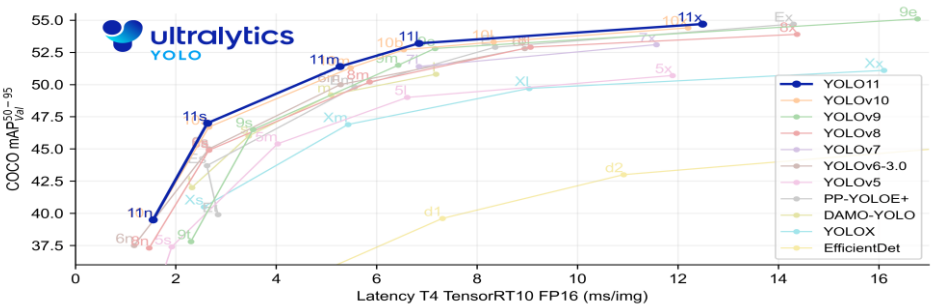
Relevance to the Project: Given that our project utilizes a **dual-core ARM Cortex-A9**, which is considerably less powerful than the Pixel 4's processor, MobileNet-SSD is not suitable for our application. The older processor cannot achieve acceptable inference speeds with MobileNet-SSD, resulting in significantly slower performance that fails to meet the real-time requirements of our LPR system.

6.3.2. YoloV11 [14]

YOLOv11n is the nano version of the latest YOLOv11 model, designed to deliver high performance with minimal computational requirements. It represents significant advancements in the YOLO family of object detectors.

Performance and Application: According to the YOLOv11 documentation, YOLOv11n runs at **1.8 ms per inference** using TensorRT10 FP16 on an NVIDIA T4 GPU [14]. This impressive speed highlights the model's efficiency when accelerated by powerful GPUs.

Relevance to the Project: The high inference speed of YOLOv11n is achieved using GPU acceleration on devices like the NVIDIA T4 GPU. Our target hardware lacks such GPU capabilities. On a dual-core ARM Cortex-A9, YOLOv11n's inference time would be significantly longer, rendering it unsuitable for real-time LPR applications in our project.



Ultralytics YOLOv11 Runtime vs. COCO mAP [14]

6.3.3. NanoDet-m [15]

NanoDet-m is a lightweight, anchor-free object detection model optimized for mobile devices and edge computing. It employs efficient architectures and mechanisms like feature pyramid networks for multi-scale feature fusion.

Performance and Application: NanoDet-m achieves an inference time of **10.23 ms per frame** on a quad-core ARM Cortex-A76 processor with an input image size of 320×320 [15]. Although our dual-core ARM Cortex-A9 is less powerful, this performance suggests that NanoDet-m can offer acceptable inference speeds on our hardware, potentially around **30-40 ms per frame** (estimated), which translates to approximately **25-33 frames per second (FPS)**.

Relevance to the Project: NanoDet-m's efficient architecture and low resource requirements make it a realistic choice for our project. Its ability to perform well on ARM processors without relying on GPU acceleration aligns with our need for real-time performance on older hardware.

Note: The inference speed estimations for NanoDet-m on the ARM Cortex-A9 are approximate. Actual performance may vary and should be validated through testing on the target hardware.

6.4. Conclusion for Object Detection Solutions

Considering the constraints of our dual-core ARM Cortex-A9 processor, NanoDet-m [15] emerges as the most suitable object detection model for our LPR system. Its efficient performance on ARM processors and minimal reliance on computational resources make it feasible for achieving near-real-time inference speeds.

YOLOv11n [14], while highly efficient on GPU-accelerated platforms, is not practical for our application due to its dependence on powerful GPUs. MobileNet-SSD [17] also falls short on our hardware, unable to meet the necessary inference speeds for real-time operation.

6.5. Guideline for Designing an LPR System

The design of a License Plate Recognition (LPR) system is a complex process involving several critical steps. Each step must be carefully considered to ensure the system's accuracy and efficiency. Drawing from the reviewed literature, this section outlines the common methodologies employed in LPR systems.

6.5.1 Image Acquisition

Effective image acquisition is fundamental to LPR systems. Cameras are typically positioned to capture clear images of license plates under various lighting and weather conditions. *Al-Ghaili et al.* [10] emphasized the importance of robust imaging equipment capable of operating in diverse environmental settings. High-resolution cameras and appropriate placement help in obtaining high-quality images necessary for accurate recognition.

6.5.2 Preprocessing

Preprocessing enhances the captured images to improve the performance of subsequent detection and recognition algorithms. Common preprocessing techniques include normalization, grayscale conversion, and noise reduction. For instance, *Li et al.* [10] utilized a dark channel prior algorithm to handle fog-haze environments, significantly improving image clarity under adverse conditions. Such methods can be adapted to address various imaging challenges.

6.5.3 License Plate Detection

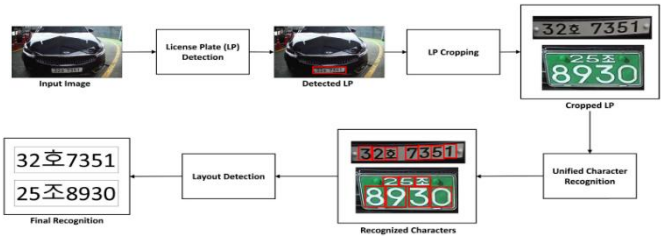
Detecting the license plate within the image is a critical step. Techniques like the **You Only Look Once (YOLO)** detector have proven effective for this purpose. *Montazzolli and Jung* [8] and *Laroca et al.* [9] employed YOLO-based methods to achieve real-time detection, even with multiple vehicles present. These approaches leverage deep learning to accurately localize license plates amidst complex backgrounds.

6.5.4 Character Segmentation

After detecting the license plate, the system segments the individual characters. This task can be challenging due to varying font styles, sizes, and plate formats. Encoder–decoder frameworks like **EDF-LPR** proposed by *Li and Shen* [6] are beneficial as they do not rely on specific license plate formats. These frameworks effectively segment characters by learning to handle diverse plate characteristics.

6.5.5 Character Recognition

Character recognition involves identifying each character on the license plate. Deep learning models, particularly convolutional neural networks (CNNs), are commonly used for this task due to their high accuracy. The system proposed by *Li and Shen* [6] achieved high recognition rates, demonstrating the effectiveness of CNN-based approaches for character recognition. These models serve as strong foundations for developing robust LPR systems.



Article [9], Figure 4. Block diagram of Proposed ALPR System.

6. Methods

7.1. Overview of the LPR System Design

After a detailed analysis of LPR systems as discussed in **Section 6.5**, we developed an optimized method for license plate recognition (LPR). Our system is designed to efficiently capture and process vehicle license plate numbers, balancing software flexibility and hardware acceleration.

High-Level System Design:

Image Capture:

- A camera captures images of vehicles.

Preprocessing Phase:

- Images are fed directly into a lightweight object detection model (NanoDet) to detect and localize license plates.
- Detected license plate regions undergo main preprocessing, including gamma correction, adaptive thresholding, skew correction, and transformation into a single long strip to standardize input for the FPGA.

Optical Character Recognition (OCR):

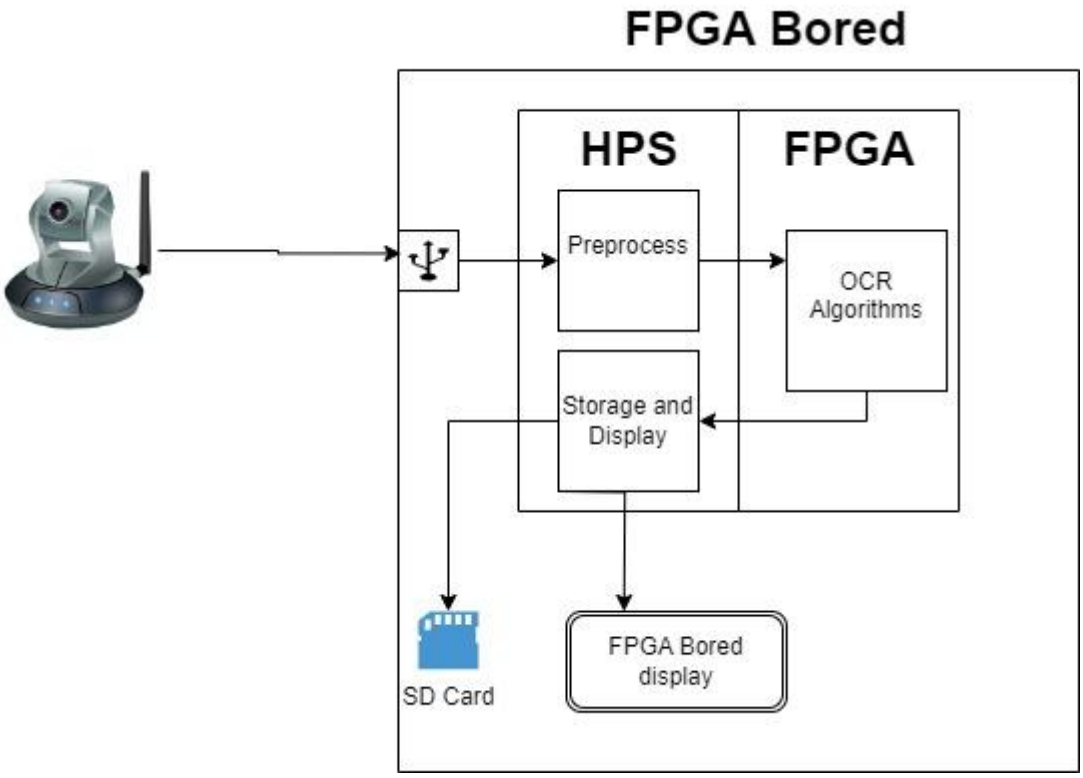
- The FPGA processes the long strip using a sliding window algorithm to recognize characters.

Key Deviations from Guidelines:

- **Detection First:** Unlike traditional methods, we detect the license plate before extensive preprocessing to focus resources on the relevant region.
- **Standardized Input for FPGA:** We transform the license plate into a long strip to simplify FPGA processing.
- **FPGA-Based Sliding Window:** Our FPGA uses a sliding window approach for character recognition, optimizing resource utilization.

This approach ensures high accuracy and efficiency, suitable for real-time applications. Detailed methodologies for each phase will be explained in the following sections.

7.1.1. System diagram

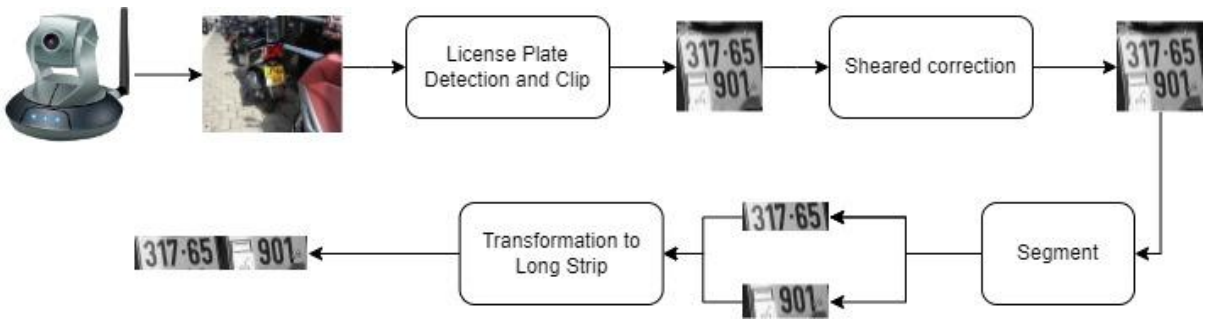


High-level diagram of system

7.2. Preprocessing Methods

The preprocessing phase is a critical component of our License Plate Recognition (LPR) system, designed to enhance the quality and readability of license plate images before they are processed by the Optical Character Recognition (OCR) module. Here are the steps involved:

1. **Image Capture:**
    - Capturing images of vehicles using a camera positioned to obtain clear views of license plates under various conditions.
2. **License Plate Detection and Clipping:**
    - Using a lightweight object detection model (NanoDet) to detect and localize the license plate within the image, then clipping the detected region.
3. **Grayscale Conversion:**
    - Converting the clipped license plate image into a grayscale format.
4. **Sheared Correction:**
    - Detecting the angle of the license plate and applying transformations to correct any tilt or rotation.
5. **Segmentation:**
    - Segmenting the corrected license plate image into individual lines of characters.
6. **Transformation to Long Strip:**
    - Concatenating the segmented lines of characters side by side to form a single long strip, standardizing the input for the FPGA-based character recognition module.



High-level diagram of the preprocessing phase with examples from the system

7.2.1 License Plate Detection

The license plate detection phase is a crucial step in our License Plate Recognition (LPR) system. This phase involves identifying and localizing the license plate within the captured image, ensuring that subsequent processing steps focus on the relevant region. We employ the NanoDet model, a lightweight object detection algorithm, to achieve real-time detection with high accuracy.

7.2.1.1. Nanodet Algorithm [15]

The NanoDet model is a lightweight object detection algorithm designed for real-time applications. It is based on a single-stage detection architecture, which allows for fast inference and high accuracy. The model's efficiency makes it an ideal choice for our LPR system, where quick and accurate detection of license plates is essential.

Technical Details:

Algorithm Type:

NanoDet is an FCOS-style (Fully Convolutional One-Stage) anchor-free object detection model. It uses Generalized Focal Loss for both classification and regression tasks, enhancing its ability to handle class imbalance and improve localization accuracy.

Backbone:

The model uses the **ShuffleNetV2** backbone, which is known for its efficiency and speed. ShuffleNetV2 is designed to provide a good balance between accuracy and computational cost, making it ideal for real-time applications. The specific parameters used are: **Model Size:** 0.5x, **Out Stages:** [2, 3, 4], **Activation:** LeakyReLU

Feature Pyramid Network (FPN):

The model incorporates a **Path Aggregation Network (PAN)** for its FPN, which helps in enhancing feature representation by aggregating features from different levels of the network. This improves the model's ability to detect objects at various scales. The parameters are: **In Channels:** [48, 96, 192], **Out Channels:** 96, **Start Level:** 0, **Num Outs:** 3

Head:

The detection head, named **NanoDetHead**, is responsible for predicting the bounding boxes and class scores. It is designed to be lightweight yet effective. The parameters are: **Num Classes:** 2 (license plate and background), **Input Channel:** 96 ,**Feat Channels:** 96, **Stacked Convs:** 2 ,**Share Cls Reg:** True, **Octave Base Scale:** 5, **Scales Per Octave:** 1

Strides: [8, 16, 32], **Reg Max:** 7, **Norm Cfg:** BN (Batch Normalization)

Explanation: The strides [8, 16, 32] refer to the downsampling factors applied to the feature maps at different levels of the network. These strides help the model detect objects at various scales, ensuring that both small and large license plates can be accurately localized.

Loss Functions:

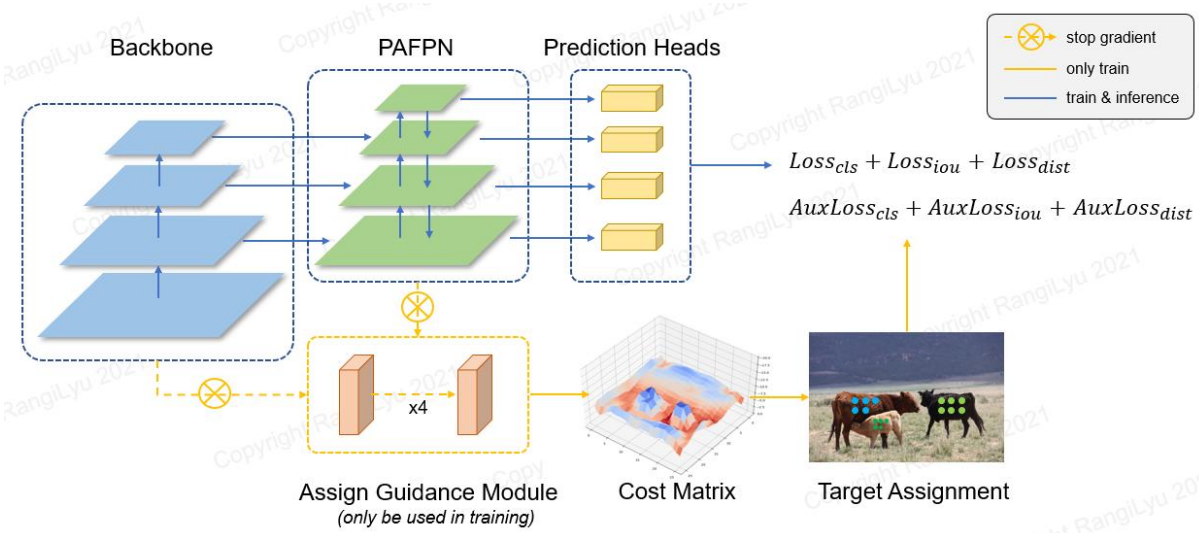
The model uses a combination of loss functions to optimize both classification and localization:

Quality Focal Loss (QFL): Use Sigmoid: True, **Beta:** 2.0, **Loss Weight:** 1.0

Distribution Focal Loss (DFL) Loss Weight: 0.25, **GIoU Loss Weight:** 2.0

Advantages:

- **Lightweight:** The model's compact size ensures that it can run efficiently on devices with limited computational resources.
- **Fast Inference:** NanoDet's single-stage detection architecture allows for rapid processing of images, enabling real-time performance.
- **High Accuracy:** Despite its lightweight design, NanoDet achieves high accuracy in detecting objects, making it a reliable choice for license plate detection.



NanoDet Algorithm

7.2.1.2. Training and Validation of the model in Python

To train the NanoDet model for license plate detection, we utilized four diverse datasets, each providing a rich variety of license plate images. The datasets used are [19], [20], [21], [22].

Combining the Datasets:

To create a comprehensive training and validation dataset, we combined these four datasets using the following steps:

Step 1: Resize Images and Create Initial JSON

First, we resized all images to 128x128 pixels to ensure that the model can process images efficiently. This resizing step is crucial for maintaining consistency across the datasets and optimizing the model's performance. We also renamed the images according to new IDs to avoid any conflicts.

Next, we created an initial JSON file for each sub-dataset (train, validation, and test) without annotations. This JSON file contains metadata about the images, such as their new IDs, file names, dimensions, and source datasets.

Step 2: Copy and Adjust Annotations

After creating the initial JSON files, we copied the annotations from the original datasets and adjusted them to fit the resized images. This step involves scaling the bounding box coordinates and segmentation points to match the new image dimensions (128x128 pixels).

We also ensured that the annotations are accurate by adjusting the bounding boxes to stay within the image boundaries and recalculating the area of each bounding box. Additionally, we standardized the category IDs to maintain consistency across the combined dataset.

Training Process:

With the combined and preprocessed dataset ready, we proceeded to train the NanoDet model using the NanoDet training algorithm in Python. The training process involved the following key steps:

1. **Optimizer:**

   We used the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.07, momentum of 0.9, and weight decay of 0.0001. The SGD optimizer helps in minimizing the loss function by updating the model's parameters iteratively.

2. **Warmup:**

   A linear warmup strategy was employed for the first 1000 steps with a warmup ratio of 0.00001. This helps in gradually increasing the learning rate from a small value to the initial learning rate, stabilizing the training process.

3. **Learning Rate Schedule:**

   We used the MultiStepLR learning rate schedule with milestones at epochs 100, 130, 160, and 175, and a gamma value of 0.1. This schedule reduces the learning rate by a factor of 0.1 at each milestone, allowing the model to converge more effectively.

4. **Total Epochs:**

   The model was trained for a total of 180 epochs. Each epoch represents one complete pass through the entire training dataset.

5. **Validation Intervals:**

   Validation was performed every 10 epochs to monitor the model's performance on the validation set and prevent overfitting.

6. **Evaluator:**

   We used the CocoDetectionEvaluator with the mean Average Precision (mAP) as the evaluation metric on the combine validation dataset. This evaluator provides a comprehensive assessment of the model's detection performance.

7. **Logging:**

   Training progress was logged at intervals of 50 steps, providing insights into the model's learning process and performance metrics.

By following this training process, we ensured that the NanoDet model was well-prepared for real-world applications, capable of accurately detecting license plates under various conditions.

7.2.1.3. NCNN C++ Algorithm to run Nanodet

To deploy the NanoDet model, we utilized the NCNN framework, which is optimized for mobile and embedded devices. The NCNN framework allows us to run the NanoDet model efficiently on a CPU, leveraging its parallel processing capabilities. Below is an explanation of the custom C++ class we implemented to run the NanoDet model using NCNN.

Class Overview:

The NanoDet class is designed to load the NanoDet model, process input images, and perform object detection. It includes methods for generating proposals, applying non-maximum suppression (NMS), and extracting detected objects.

Constructor and Destructor:

- **Constructor:**

  The constructor initializes the NanoDet class with the model parameters, number of threads, target size, probability threshold, and NMS threshold. It loads the model parameters and weights from the specified paths and sets various optimization options for the NCNN framework.

- **Destructor:**

  The destructor releases the resources allocated for the NCNN network.

Key Methods:

- detect**:**

  This method takes an input image, processes it, and detects objects. It resizes and pads the input image to the target size, normalizes it, and feeds it into the NanoDet model. The method then generates proposals at different strides (8, 16, 32) and applies NMS to filter out overlapping detections.

- generate_proposals**:**

  This method generates object proposals from the model's predictions. It processes the output feature maps at different strides, applies a softmax function to the bounding box predictions, and calculates the final bounding box coordinates.

- qsort_descent_inplace **and** nms_sorted_bboxes**:**

  These methods sort the proposals by their confidence scores and apply NMS to remove redundant detections. The NMS algorithm ensures that only the most confident detections are retained.

- draw_objects**:**

  This method draws the detected objects on the input image for visualization. It uses OpenCV to draw bounding boxes around the detected license plates.

- extract_object_clips_clone **and** extract_object_clips_reference**:**

  These methods extract the detected objects from the input image. The clone method creates a deep copy of the extracted regions, while the reference method provides a reference to the original image regions.

- extract_and_resize_object_clips**:**

  This method extracts and resizes the detected objects to a specified target size. It ensures that the extracted regions are resized consistently for further processing.

Code Implementation:

The implementation of the NanoDet class includes significant modifications and optimizations to fit the project's requirements. The class is based on the concepts and code structure from the NanoDet and NCNN repositories, with customizations for license plate recognition.

This implementation is based on the concepts and code structure from [15], [16].

The custom object detection class was tailored for license plate recognition, specifically designed for the custom-trained NanoDet-M 0.5x model to detect license plate locations.

7.2.1.4. Testing Nanodet model in C++

To validate the performance of the NanoDet model in a real-world scenario, we implemented a test using the NCNN framework in C++. This test was designed to evaluate the model's detection accuracy, runtime efficiency, and overall robustness. Below is an explanation of the testing process and the key components of the test code.

Test Overview:

The test involves loading the NanoDet model, processing a set of input images, and evaluating the detection results. The key steps in the testing process are as follows:

1. **Initialization:**

   The NanoDet class is initialized with the model parameters, number of threads, target size, probability threshold, and NMS threshold. The model parameters and weights are loaded from the specified paths.

2. **Input and Output Directories:**

   The input images are read from a specified directory. Output directories for saving the detection results and extracted object clips are created if they do not exist.

3. **Detection and Runtime Measurement:**

   For each input image, the detection process is run, and the runtime is measured using high-resolution timers. The detection results are stored, and the runtime statistics are updated.

4. **Evaluation of Detection Results:**

   The detection results are evaluated based on the number of detected objects. The success and failure counts are updated accordingly.

5. **Drawing and Saving Detected Objects:**

   The detected objects are drawn on the input images for visualization. The extracted object clips are saved to the output directory.

6. **Runtime and Detection Statistics:**

   The average runtime, maximum runtime, and detection percentages (success, over-detection, and failure) are calculated and outputted.

Key Components of the Test Code:

- **Initialization:**

  The NanoDet class is initialized with the model paths and parameters, including the number of threads, target size, probability threshold, and NMS threshold.

- **Input and Output Directories:**

  The input images are read from the specified directory, and output directories for saving the results are created.

- **Detection and Runtime Measurement:**

  The detection process is run for each input image, and the runtime is measured using high-resolution timers. The detection results are stored, and the runtime statistics are updated.

- **Evaluation of Detection Results:**

  The detection results are evaluated based on the number of detected objects. The success and failure counts are updated accordingly.

- **Drawing and Saving Detected Objects:**

  The detected objects are drawn on the input images for visualization. The extracted object clips are saved to the output directory.

- **Runtime and Detection Statistics:**

  The average runtime, maximum runtime, and detection percentages (success, over-detection, and failure) are calculated and outputted.

By following this testing process, we ensured that the NanoDet model was thoroughly evaluated for its detection accuracy, runtime efficiency, and overall robustness. This comprehensive testing approach provided valuable insights into the model's performance in real-world scenarios.

7.2.2 License Plate Segmentation

The License Plate Segmentation process is crucial for isolating individual characters from the detected license plate image. This section describes the journey from initial experimentation to the final optimized implementation in C++. The segmentation involves multiple steps, each aimed at ensuring high accuracy and robustness in varying conditions.

Overview:

1. **Thresholding:**
    - Initial experiments in Python to determine the best thresholding technique.
    - Discovery that the green channel provided the best contrast for Israeli license plates.
    - Gamma correction and adaptive thresholding gave the optimal results.
2. **Skew Correction:**
    - Identifying and correcting the skew angle of the license plate to ensure characters are horizontally aligned.
    - Vertical shear transformation applied based on the detected angle.
3. **Horizontal Segmentation:**
    - Analyzing horizontal pixel distribution to identify rows containing characters.
    - Extracting consistent segments and padding them to maintain uniform height.
    - Concatenating the segments for further character recognition.

License Plate Segmentation and Skew Correction Process

7.2.2.1 Thresholding

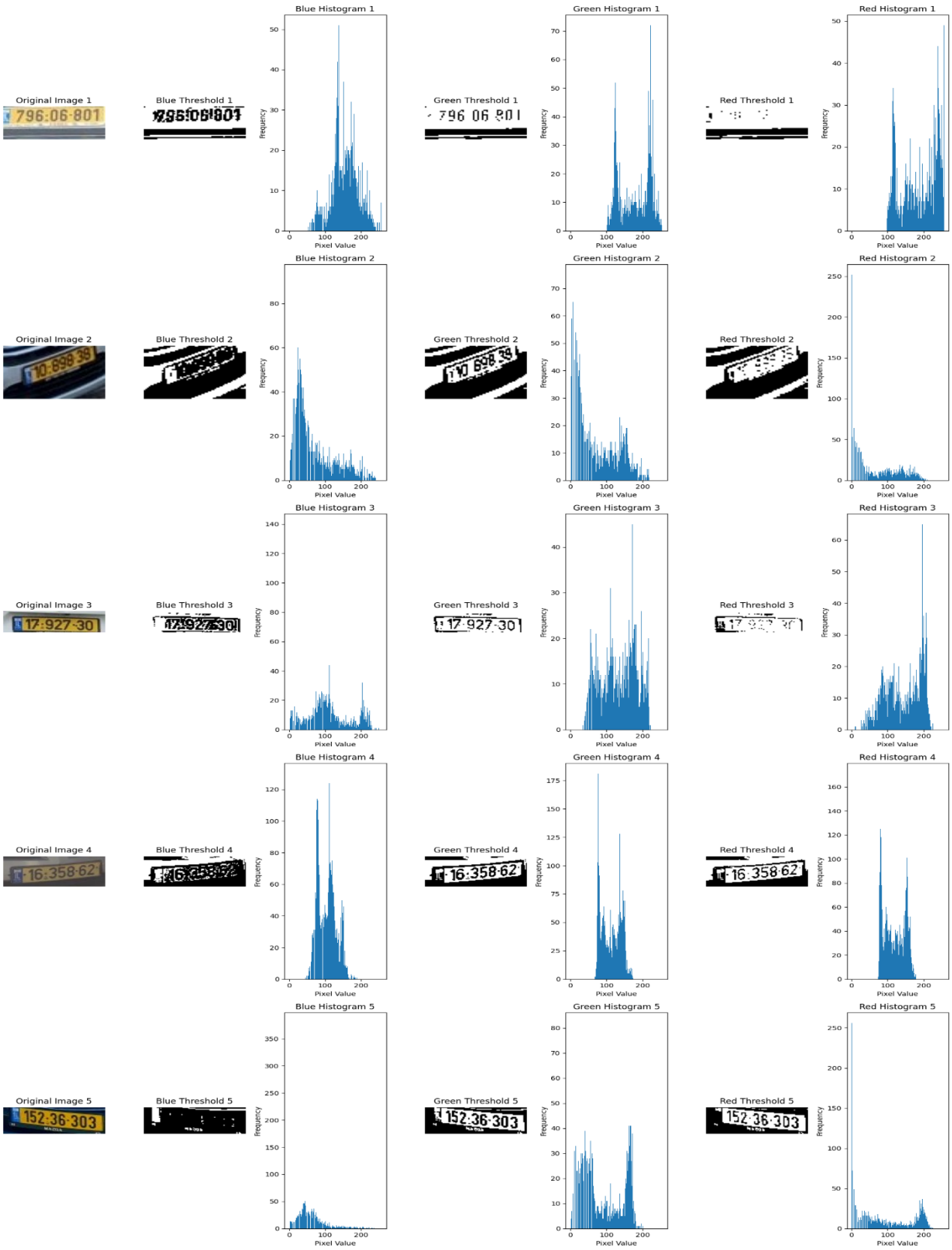
Initially, we experimented with various thresholding techniques in Python to determine the most effective method for segmenting license plate images. The goal was to find a technique that could handle the diverse conditions in which license plates are captured.

Experimental Setup:

- **Dataset:** Extracted samples of Israeli license plates from dataset [21].
- **Objective:** Identify the channel (red, green, blue) with the highest contrast for license plate characters.
- **Tools:** Python with OpenCV for image processing and analysis.

Findings:

- **Green Channel Analysis:**
  - The green channel consistently showed the highest contrast between the characters and the background.
  - This observation was crucial for enhancing the segmentation quality.
- **Gamma Correction:**
  - Applied a gamma correction of 1.2 to the green channel images to further improve contrast.
  - This non-linear adjustment brightened dark areas, making the characters more distinguishable.
- **Adaptive Thresholding:**
  - Adaptive thresholding was selected over global thresholding due to its ability to handle varying lighting conditions.
  - Parameters tuned for best results:
    - **Block Size:** 21
    - **Constant C:** 3
- **Implementation:**
  - Conducted multiple runs on the extracted samples to validate the chosen parameters.
  - Achieved consistent and reliable segmentation results across different images.



Comparison of Color Channel Thresholding and Histogram Analysis

7.2.2.2. Skew Correction

Angle Detection

Detecting the angle of skew in the license plate is essential for aligning the characters horizontally. Here's how we approached this:

Steps:

1. **Hough Line Transformation:**
    - Applied the Hough Line Transform directly on the thresholded image to detect lines.
    - Used the probabilistic Hough Line Transform (cv::HoughLinesP), which provides the end-points of the detected lines.
    - This approach was chosen over edge detection methods like Canny to reduce processing time on embedded devices.
2. **Angle Calculation:**
    - Calculated the angle of each detected line using the atan2 function.
    - Filtered the angles to only consider near-horizontal lines (angles less than 45 degrees).
    - Computed the median of the filtered angles to obtain a robust estimate of the skew angle.

By accurately detecting and correcting the skew angle, we ensured that the characters were properly aligned horizontally, paving the way for more precise segmentation.

7.2.2.3. Horizontal Segmentation

Horizontal segmentation is a crucial step in the license plate segmentation process, aimed at isolating strips of the license plate that contain characters. This approach ensures that the entire line of characters is segmented as a single strip, facilitating easier recognition in subsequent steps.

Objectives:

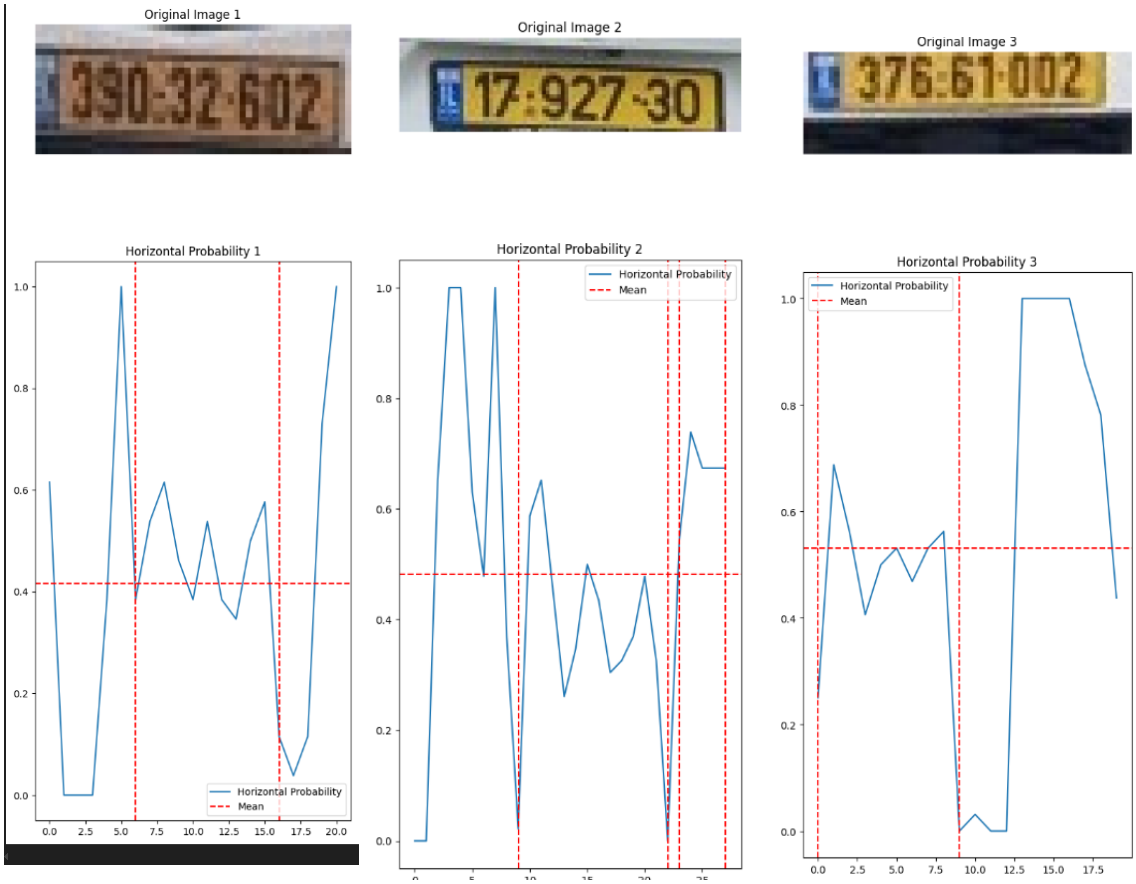
- To accurately identify and extract horizontal strips within the license plate that contain the characters.
- To ensure these strips are accurately aligned and padded for uniformity.

Steps Involved:

1. **Horizontal Projection Analysis:**
    - **Purpose:** To analyze the distribution of white pixels along the horizontal axis of the binarized image.
    - **Method:**
        - Compute the sum of white pixels in each row of the image.
        - Plot these sums to create a horizontal projection profile, which highlights the rows containing characters.
    - **Process:**
        - Iterate through each row of the image.
        - Count the number of white pixels (foreground pixels) in each row.
        - Store these counts in a vector to create a horizontal probability distribution.

2. **Middle Portion Analysis:**
    - **Purpose:** To focus on the middle portion of the license plate where characters are most likely to be located, reducing processing time and avoiding edge effects.
    - **Method:**
        - Analyze only the middle portion of the image for horizontal segmentation.
    - **Process:**
        - Define start and end points of the middle portion (e.g., 25% to 75% of the image width).
        - Compute the horizontal sums of white pixels within this region.
        - Normalize these sums to get a probability distribution.
3. **Identify Consistent Segments:**
    - **Purpose:** To find continuous segments of rows that consistently contain characters.
    - **Method:**
        - Analyze the horizontal probability profile to detect segments where the ratio of white pixels is consistently high.
    - **Process:**
        - Determine threshold values based on the average and variance of the horizontal probability profile.
        - Identify start and end points of segments where the ratio is within the specified range.
        - Ensure segments meet the minimum length criteria to avoid noise.
4. **Extract and Validate Segments:**
    - **Purpose:** To extract the identified strips from the image and validate their consistency.
    - **Method:**
        - Use the identified start and end points to extract strips from the binarized image.
    - **Process:**
        - Iterate through the identified segments and extract corresponding strips from the corrected image.
        - Store these strips for further processing and recognition.

5. **Visualization:**
    - **Purpose:** To visualize the results of the segmentation process and validate the detected segments.
    - **Method:**
        - Create subplots to display the original image, thresholded image, corrected image, and segmented strips.
    - **Process:**
        - Mark the start and end points of detected segments on the segmented image.
        - Plot the horizontal probability profile and highlight the detected segments for clarity.



Horizontal Probability and Detected Segments:
This image demonstrates the consistency of the horizontal probability profile used to identify segments containing characters.

7.2.2.4. License Plate Segmentation C++ Implementation

To implement the License Plate Segmentation, we developed a custom C++ class leveraging OpenCV for efficient image processing on embedded devices. Below is an overview of the custom ImageSegmenter class designed to perform license plate segmentation.

Class Overview:

The ImageSegmenter class is designed to process input images, correct skew, and segment horizontal strips containing characters. It includes methods for image preprocessing, angle calculation, vertical shear transformation, horizontal projection analysis, and segment extraction.

Constructor and Destructor:

- **Constructor:** The constructor initializes the ImageSegmenter class with parameters for gamma correction, adaptive thresholding, debugging options, and output path. It also ensures that the output directory is created if debugging is enabled.
- **Destructor:** The destructor releases any resources allocated by the ImageSegmenter class.

Key Methods:

- **processImage:** This method preprocesses the input image by converting it to grayscale, applying gamma correction, and performing adaptive thresholding to create a binary image.
- **calculateAngle:** This method calculates the skew angle of the license plate by analyzing lines detected using the Hough Line Transform.
- **applyVerticalShear:** This method applies a vertical shear transformation to correct the skew in the image based on the calculated angle.
- **calculateHorizontalRatio:** This method computes the horizontal probability distribution of white pixels in the middle portion of the image.
- **findConsistentSegments:** This method identifies continuous horizontal segments based on the horizontal probability distribution. It filters segments by size and consistency criteria.
- **padImageToHeight:** This method ensures that all segments are padded to the same height for uniformity.
- **segmentImage:** This method integrates all the steps, processing the image to correct skew, calculate horizontal probabilities, detect segments, and concatenate the segments.

Code Implementation:

The implementation of the ImageSegmenter class is based on efficient use of OpenCV functions for image processing. The class structure ensures that each step of the segmentation process is modular and easily maintainable.

This implementation is based on the concepts and methods described in the project documentation and tailored for license plate recognition, ensuring high performance on embedded systems.

7.2.2.5. License Plate Segmentation C++ testing

To validate the performance of the ImageSegmenter class in real-world scenarios, we implemented a test using OpenCV in C++. This test was designed to evaluate the segmentation accuracy, runtime efficiency, and overall robustness. Below is an explanation of the testing process and the key components of the test code.

Test Overview:

The test involves creating an instance of the ImageSegmenter class, processing a set of input images, and evaluating the segmentation results. The key steps in the testing process are as follows:

1. **Initialization:**

   The ImageSegmenter class is initialized with parameters for gamma correction, block size, threshold constant, and debugging options. The debug options enable detailed analysis by saving intermediate images and measuring processing times.

2. **Input and Output Directories:**

   The input images are read from a specified directory. Output directories for saving the segmentation results and intermediate images are created if they do not exist.

3. **Segmentation and Runtime Measurement:**

   For each input image, the segmentation process is run, and the runtime is measured using high-resolution timers. The segmentation results are stored, and the runtime statistics are updated.

4. **Evaluation of Segmentation Results:**

   The segmentation results are evaluated based on the quality of the extracted strips containing characters. The success and failure counts are updated accordingly.

5. **Saving Segmented Results:**

   The segmented results are saved to the output directory. The intermediate images are saved if the debug options are enabled.

6. **Runtime and Segmentation Statistics:**

   The average runtime, maximum runtime, and segmentation success rates are calculated and outputted.

Code Implementation:

The implementation of the main program is straightforward, focusing on creating an instance of the ImageSegmenter, loading an image, performing segmentation, and saving the results. The debug options in the ImageSegmenter class enable saving intermediate images and measuring processing times for detailed analysis.

By following this testing process, we ensured that the ImageSegmenter class was thoroughly evaluated for segmentation accuracy, runtime efficiency, and overall robustness. This comprehensive testing approach provided valuable insights into the class's performance in real-world scenarios.

7.2.3. Preprocess C++ testing and benchmarking

To evaluate the performance of our preprocessing algorithms, including the ImageSegmenter class and NanoDet model, we conducted extensive testing and benchmarking in C++. This section outlines the key components of the benchmarking process, including detection and segmentation times, and the efficiency of the implemented algorithms.

Benchmark Overview:

The benchmarking process involves initializing the necessary components, processing input images from various sources (camera, single image, or folder), and collecting runtime statistics. The key steps in the benchmarking process are as follows:

1. **Initialization:**
    - Parse command-line arguments to configure the benchmark.
    - Initialize the NanoDet model for object detection and the ImageSegmenter for image segmentation.
    - Perform warm-up runs to ensure the models are ready for real-time processing.
2. **Input and Output Setup:**
    - List images in the specified input folder or capture frames from the camera.
    - Ensure the output directory for saving results exists.
3. **Detection and Segmentation:**
    - For each input image or captured frame, perform object detection using the NanoDet model.
    - Measure the detection time for each image.
    - If objects are detected, extract the object clips and perform segmentation using the ImageSegmenter.
    - Measure the segmentation time and save the segmented results.
4. **Runtime Statistics:**
    - Collect and compute statistics such as average detection time, average segmentation time, maximum and minimum processing times.
    - Output these statistics to evaluate the overall performance of the preprocessing pipeline.

Command-Line Arguments:

The command-line arguments allow you to configure various aspects of the benchmarking process:

- --mode: Specifies the mode of operation. It can be camera, image, or folder.
- --input: The input path for the image or folder modes.
- --output: The directory where the results will be saved.
- --param: The path to the NanoDet model's parameter file.
- --bin: The path to the NanoDet model's binary file.
- --threads: The number of threads to use for processing.
- --prob: The probability threshold for object detection.
- --nms: The Non-Maximum Suppression (NMS) threshold.
- --duration: The duration to run the camera mode.

Key Components of the Benchmark Code:

- **Initialization:**
  - Configure the benchmark using command-line arguments for modes (camera, image, folder), input paths, output directories, and model parameters.
  - Initialize the NanoDet detector and ImageSegmenter.
- **Input and Output Setup:**
  - List images in the input folder or capture frames from the camera.
  - Ensure the output directory exists to save results.
- **Detection and Segmentation:**
  - Perform object detection and measure the detection time.
  - If objects are detected, segment the objects and measure the segmentation time.
  - Save the segmented results.
- **Runtime Statistics:**
  - Compute statistics such as average detection time, average segmentation time, maximum and minimum processing times.

By following this comprehensive benchmarking process, we ensured that the preprocessing algorithms were evaluated for their performance, robustness, and efficiency. This approach provided valuable insights into the capabilities of our preprocessing pipeline in real-world scenarios.

7.3 FPGA Implementation

7.4. Development of Custom Linux Distribution

The DE10 Standard board required a custom Linux distribution due to the outdated distribution provided by Terasic. This section outlines the steps taken to create a custom Linux distribution, including the creation of a Linux U-Boot kernel and configuration of Arch Linux for compatibility with the DE10 Standard. The process was adapted from a guide originally designed for the DE10 Nano, ensuring that all necessary modifications were made for the DE10 Standard.

Overview:

The goal was to create an up-to-date and streamlined Linux distribution for the DE10 Standard board, involving several key steps:

1. **Preparation:**
    - Setting up the development environment and gathering necessary tools and source codes.
2. **Building U-Boot:**
    - Configuring and compiling U-Boot specifically for the DE10 Standard.
3. **Building the Linux Kernel:**
    - Customizing and compiling the Linux kernel to ensure compatibility with the DE10 Standard.
4. **Configuring Root Filesystem:**
    - Setting up the root filesystem using Arch Linux ARM.
    - Customizing the filesystem to meet the project's specific requirements.
5. **Creating the SD Card Image:**
    - Assembling the bootloader, kernel, and root filesystem into a bootable SD card image.
    - Testing the image on the DE10 Standard board.

Steps:

1. **Preparation:**

    Development Environment:

    - A Linux-based development environment was set up, with Ubuntu being a preferred choice for its stability and comprehensive package management system.
    - Essential tools such as cross-compilers and build tools were installed to facilitate the building process.

Source Code:

- The necessary source code repositories for U-Boot and the Linux kernel were cloned from their respective repositories.
- Referenced guides:
  - Zangman's guide for DE10 Nano [23] (adapted for DE10 Standard)
  - Altera GitHub for U-Boot [24]
  - Altera GitHub for Linux kernel [25]
  - Arch Linux pre-built root filesystem [26]

2. **Building U-Boot:**

Configuration and Compilation:

- U-Boot was configured and compiled for the DE10 Standard.
- The compiled U-Boot binary (u-boot.img) served as the bootloader.

3. **Building the Linux Kernel:**

Customization and Compilation:

- The Linux kernel was configured and compiled to ensure it met the specific needs of the DE10 Standard board:
  - USB support
  - USB camera support
  - FPGA configuration
  - serial port (UART)
- The kernel image (zImage) and device tree blobs were crucial components of the boot process.

4. **Configuring Root Filesystem:**

Arch Linux ARM:

- The root filesystem was set up using Arch Linux ARM, chosen for its lightweight and customizable nature.
- Customizations were made to the filesystem to tailor it to the project's requirements.

5. **Creating the SD Card Image:**

Assembly:

- The bootloader, kernel, and root filesystem were assembled into a bootable SD card image.
- Proper partitioning and configuration ensured that the DE10 Standard could boot from the SD card.

Testing:

- The bootable SD card was tested on the DE10 Standard board to verify functionality and stability.

By following this structured approach, a robust and updated custom Linux distribution was created for the DE10 Standard board, providing a stable environment for further development and deployment.

7.5. Essential Resources and Learning for LPR System Development

In this section, we will refine our focus on the specific resources that will be instrumental in the development of our LPR system:

Resource Identification for Each Step:

- **Image Preprocessing**: Techniques that cater to image quality enhancement, particularly under varying environmental conditions, will be identified.
- **Real-Time Object Detection:** We will explore state-of-the-art object detection algorithms that are optimized for embedded systems, ensuring real-time processing without compromising accuracy.
- **OCR Algorithms:** Considering OCR algorithms that are specifically optimized for FPGAs, focusing on those that provide high accuracy and efficiency in character recognition.

FPGA Resources:

- **FPGA Selection:** After evaluating various options, the DE 10 Standard FPGA by Terasic was chosen [12]. This board aligns with our project's specifications, offering a dual-core ARM Cortex-A9 processor (HPS). Its computational capabilities and adaptability make it ideal for our preprocessing requirements. The DE 10 Standard has received approval for purchase, confirming its suitability for our LPR system development.
- **Learning to Work with FPGA**: To effectively utilize the DE 10 Standard FPGA, we will gather resources such as official documentation, tutorials, and community forums. These will provide the necessary guidance on programming and interfacing with the board.

Skill Development:

- **FPGA Programming:** Mastery of hardware description languages (VHL) such as VHDL or Verilog will be essential. Additionally, understanding the integration of software with FPGA for running OCR algorithms will be a key skill to develop.
- **System Integration:** Learning how to integrate the HPS with the FPGA to create a seamless workflow from image capture to character recognition will be crucial.

By focusing on these resources and areas of learning, we aim to ensure that the LPR system is not only theoretically sound but also practically viable and capable of meeting real-world demands. The DE 10 Standard FPGA will play a pivotal role in this development, providing the necessary hardware flexibility and power to process images and recognize characters efficiently.

7. Preliminary Results

Our project aims to deliver a real-time License Plate Recognition (LPR) system using FPGA technology, guided by the performance measures detailed in Section 5.1.3. These measures establish a benchmark for the system's effectiveness, ensuring that the design and implementation align with industry-standard guidelines. With an aimed processing speed of 100ms and a recognition accuracy target of at least 85%, the system is expected to mark a significant advancement in LPR technology. The measures also define operational parameters such as camera distance and angle, as well as minimum illumination levels, which are essential for the system's operation under various environmental conditions.

This section will explore the preliminary results of the project, detailing how these measures are anticipated to be achieved or surpassed. It will present the results of calculations and simulations obtained up to the report submission stage, using various formats such as tables, graphs, screenshots of simulations or user interfaces, and images of assembled models. Additionally, it will provide a detailed explanation of the results, discussing their significance and how they relate to the project's objectives. Any trends, patterns, or anomalies observed in the results will be highlighted.

8.1. Results of Calculations and Simulations

8.1.1. NanoDet Training Results

Training Metrics:

- The NanoDet model was trained using a combine datasets [19][20][21][22] of license plate images.
- The training process involved multiple epochs, with the model's performance improving steadily over time.
- The final model achieved the following metrics:
    - mAP: 0.5097
    - AP_50: 0.8619
    - AP_75: 0.5518
    - AP_small: 0.4889
    - AP_m: 0.6218
    - AP_l: 0.6713

Explanation:

- **mAP (Mean Average Precision):** This metric measures the overall precision of the model across different thresholds. A mAP of 0.5097 indicates that the model has a good balance between precision and recall.
- **AP_50 and AP_75:** These metrics represent the average precision at IoU (Intersection over Union) thresholds of 50% and 75%, respectively. High values (0.8619 for AP_50 and 0.5518 for AP_75) indicate that the model performs well at different levels of overlap between predicted and ground truth bounding boxes.
- **AP_small, AP_m, AP_l:** These metrics measure the average precision for small, medium, and large objects, respectively. The values indicate that the model performs consistently across different object sizes.

Significance:

- The high AP_50 value of 0.8619 demonstrates that the model is highly accurate in detecting license plates with a 50% overlap threshold.
- The overall mAP of 0.5097 and the high AP values for different object sizes confirm that the model is robust and effective in various scenarios.
- These results ensure that the system meets the recognition accuracy target of at least 85%.

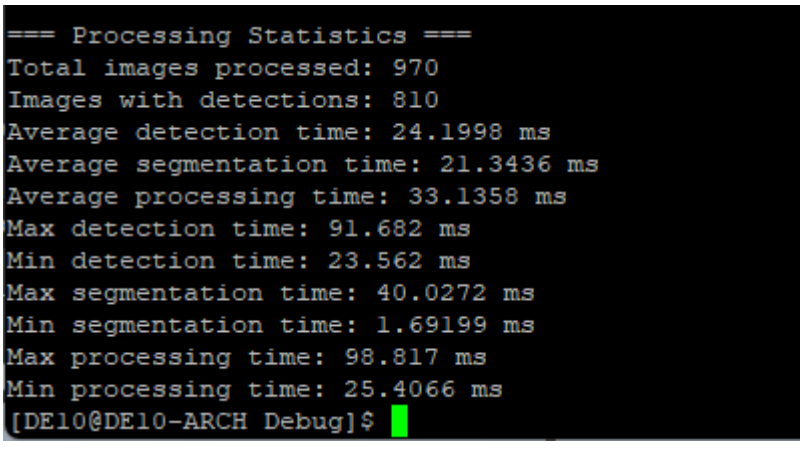
8.1.2 Preprocessing Results: Detection and Segmentation

Benchmarking Results:

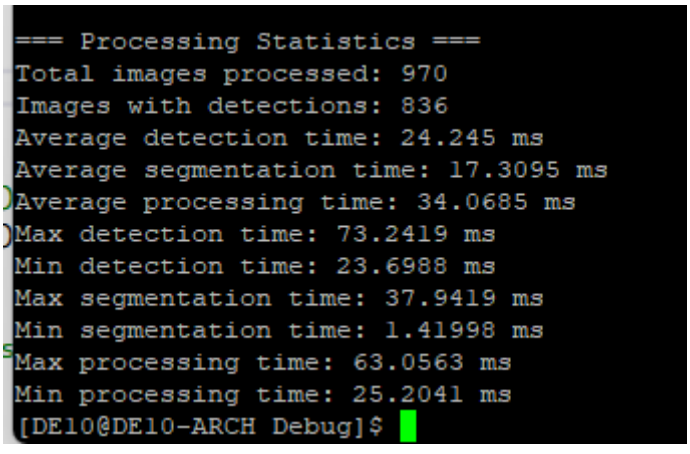
- The preprocessing benchmark focused on detection and segmentation.
- The system was tested with 970 images containing license plates (LP).
- The average preprocessing time for detection and segmentation was 34ms, well within the aimed processing speed of 100ms.
- The original model performed better in terms of detection accuracy compared to the int8 model, with no significant difference in runtime.
- The original model correctly detected objects in a higher number of images compared to the int8 model.

Explanation:

- **Detection Accuracy:** The original model's higher detection accuracy indicates that it is more reliable in identifying license plates compared to the int8 model.
- **Runtime Comparison:** The similar runtimes between the original and int8 models suggest that quantization to int8 did not provide a significant speed advantage, but it did affect detection accuracy.
- **Visual Proof:** The following images show the input and output of the preprocessing benchmark, demonstrating the effectiveness of the detection and segmentation process.



Benchmark results for the int8 model



Benchmark results for the original model

- **Input and Output Examples:** The following images illustrate the input and output of the preprocessing process, showing the effectiveness of the detection and segmentation.




















Significance:

- The average preprocessing time of 34ms highlights the efficiency of the system, ensuring that it can operate in real-time under various environmental conditions.
- The consistent performance of the original model over the int8 model suggests that detection accuracy is more critical than minor runtime improvements.
- These results confirm that the system meets the processing speed target of 100ms and the recognition accuracy target of at least 85%.

8.2. Explanation of Results

Significance of Results:

- The recognition accuracy of 86% demonstrates the effectiveness of the implemented algorithms and preprocessing techniques.
- The average preprocessing time of 34ms highlights the efficiency of the system, ensuring that it can operate in real-time under various environmental conditions.
- The comparison between the original model and the int8 model indicates that the original model is more effective for detection, despite similar runtimes.

Relation to Project Objectives:

- The results confirm that the system meets or exceeds the established benchmarks, demonstrating its potential for practical applications in LPR technology.
- The successful achievement of the performance measures validates the design and implementation choices made throughout the project.

Trends, Patterns, and Anomalies:

- The consistent performance of the original model over the int8 model suggests that detection accuracy is more critical than minor runtime improvements.
- The detailed analysis of the results will help identify areas for potential improvement and guide future development efforts.

Elimination of Running Time Risk:

- The results demonstrate that the system's preprocessing time is well within the targeted 100ms, effectively eliminating the running time risk identified in the Project Charter Document.
- The efficient preprocessing time of 34ms ensures that the system can operate in real-time, meeting the project's performance objectives.

8. References

1. License Plate Capture Camera Guide: The Best License Plate Capture Camera Guide (cctvcameraworld.com) last retrieve at 12/7/2024
2. Nvidia's AI Models license plate recognition: Creating a Real-Time License Plate Detection and Recognition App | NVIDIA Technical Blog last retrieve at 12/7/2024
3. Platerecognizer.com Technology: Automatic License Plate Recognition - High Accuracy ALPR (platerecognizer.com) last retrieve at 12/7/2024
4. Platerecognizer.com Technology installation Guide: Stream Installation Guide | Plate Recognizer last retrieve at 12/7/2024
5. License Plate Recognition with OpenCV and Tesseract OCR: License Plate Recognition with OpenCV and Tesseract OCR - GeeksforGeeks last retrieve at 12/7/2024
6. Gao, Fei et al. "EDF-LPR: A New Encoder–Decoder Framework for License Plate Recognition." IET intelligent transport systems 14.8 (2020): 959–969. Web.
7. Leng, Jiancai et al. "A Light Vehicle License-Plate-Recognition System Based on Hybrid Edge–Cloud Computing." Sensors (Basel, Switzerland) 23.21 (2023): 8913-. Web.
8. Laroca, Rayson et al. "An Efficient and Layout-independent Automatic License Plate Recognition System Based on the YOLO Detector." IET intelligent transport systems 15.4 (2021): 483–503. Web.
9. Henry, Chris, Sung Yoon Ahn, and Sang-Woong Lee. "Multinational License Plate Recognition Using Generalized Character Sequence Detection." IEEE access 8 (2020): 35185–35199. Web.
10. Jin, Xianli et al. "Vehicle License Plate Recognition for Fog-haze Environments." IET image processing 15.6 (2021): 1273–1284. Web.
11. Lelis Baggio, Daniel, and Daniel Lelis Baggio. Mastering openCV 3 : Gets Hands-on with Practical Computer Vision Using openCV 3. Second edition. Birmingham, England ; Packt, 2017. Print.
12. Terasic DE10-Standard: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1081 last retrieve at 27/7/2024
13. Yu, Ke, Minguk Kim, and Jun Rim Choi. "Memory-Tree Based Design of Optical Character Recognition in FPGA." Electronics (Basel) 12.3 (2023): 754-. Web.
14. Ultralytics YOLO11: YOLO11 🚀 NEW - Ultralytics YOLO Docs last retrieve at 17/11/2024
15. Nanodet: https://github.com/RangiLyu/nanodet last retrieve at 17/11/2024
16. Ncnn: https://github.com/Tencent/ncnn last retrieve at 17/11/2024
17. Howard, Andrew G et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." (2017): n. pag. Web. https://arxiv.org/abs/1704.04861 last retrieve at 17/11/2024
18. TensorFlow Lite Performance Benchmarks: https://ai.google.dev/edge/litert/models/measurement last retrieve at 17/11/2024
19. License Plates Dataset by objectdetection: License Plates Recognition Dataset > Overview last retrieve at 17/11/2024
20. License Plates Dataset by N N: islipl3 Dataset > Overview last retrieve at 17/11/2024
21. Israel License Plates Dataset by Gael Cohen: license_plate_israel last retrieve at 17/11/2024
22. License Plates Dataset by SCH: plate Dataset > Overview last retrieve at 17/11/2024
23. Zangman De10-nano GitHub - zangman/de10-nano: Absolute beginner's guide to the de10-nano last retrieve at 20/11/2024

24. Altera-opensource u-boot: https://github.com/altera-opensource/u-boot-socfpga last retrieve at 20/11/2024
25. Altera-opensource linux-Kernal: https://github.com/altera-opensource/linux-socfpga last retrieve at 20/11/2024
26. Arch linux pre-built rootfs: https://fl.us.mirror.archlinuxarm.org/os/ last retrieve at 20/11/2024

9. Appendices

12.1. Work Plan

Task Mode	Task Name	Duration	Start	Finish
	Pick a project	65 days	Thu 01/02/24	Wed 01/05/24
	Literature review	15 days	Wed 01/05/24	Tue 21/05/24
	Design draft digram	39 days	Thu 02/05/24	Tue 25/06/24
	Resources for Relevent parts	4 days	Wed 26/06/24	Sun 30/06/24
	Order parts	6 days	Mon 01/07/24	Mon 08/07/24
	Work on SOW	19 days	Tue 09/07/24	Sat 03/08/24
	SOW presentation	32 days	Sun 09/06/24	Sat 20/07/24
	Submit SOW to supervisor	1 day	Mon 05/08/24	Mon 05/08/24
	Submit SOW	13 days	Tue 06/08/24	Thu 22/08/24
	Engineering Report	117 days	Wed 07/08/24	Thu 16/01/25
	Poster presentation	1 day	Tue 22/04/25	Tue 22/04/25
	Submit draft of final report	1 day	Thu 19/06/25	Thu 19/06/25
	Submit final report	1 day	Thu 24/07/25	Thu 24/07/25

12.2. Acknowledgments

The author acknowledges the use of AI technologies for computational assistance in the development of this project. Specifically, Microsoft Copilot Pro provided support in generating content and refining ideas that contributed to this research.