Tiny ML in Edge Devices

Project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology

In

Computer Science and Engineering

School of Engineering and Sciences

Submitted by

Sayan Biswas(AP2111001129)

Abhishek Arya(AP21110011141)

Under the Guidance of

Dr. Tamoghna Ojha

SRM University–AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh – 522 240

Nov, 2023

Certificate

Date: 1-Dec-23

This is to certify that the work present in this Project entitled "**Tiny ML in Edge Devices**" has been carried out by **Sayan Biswas and Abhishek Arya** under my supervision. The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology in **School of Engineering and Sciences**.

Supervisor

[illegible]

(Signature)

Dr. Tamoghna Ojha

Assistant Professor

Department of CSE

SRM University AP

Acknowledgement

We thank the people who were a part of this project in numerous ways, people who gave their unending support right from the stage the project idea was conceived. The four things that go on to make a successful endeavour are dedication, hard work, patience, and correct guidance. We would like to thank our mentor for all the help he has rendered to ensure the successful completion of the project and gave his suggestions for developing our project in a better way. He was very much kind enough to give us an idea and guide us throughout our project work. Last but not the least I would like to thank all our friends for their support, and all others who have contributed to the completion of this project directly or indirectly.

Table of Contents

Abstract

Tiny Machine Learning (TinyML) is changing how we live our daily lives, popping up in things like smartwatches, sensors, and gadgets at home. This paper gets into the nitty-gritty of making a smart system that can recognize objects using TensorFlow on an Arduino—basically, a cool example of how TinyML is making small devices smarter.

With TinyML, these little devices can run smart programs without always needing to be connected to the internet. We show how you can make an Arduino spot objects in real-time, like for turning lights on at home or keeping an eye out for security.

Making this happen involves training and setting up a smart program with TensorFlow and Ultralytics YOLO, then making sure it works on an Arduino. We also tackle challenges like dealing with the small memory on the device, making everything work even with limited resources. In a typical setting, the system that recognizes vehicle registration plates can detect them; but, when the weather becomes cloudy or wet, the system's ability to do so is compromised [4]. This project is like a sneak peek into how TinyML can make small gadgets do smart things right where you are, without always needing to ask the big, far-away computer for help.

Statement of Contributions

We, Sayan Biswas and Abhishek Arya, hereby declare that the following is an accurate statement of our contributions to the research study:

Sayan Biswas assumed responsibility for the conceptualization of ideas, conducted the data collection and subsequent analysis, prepared figures and tables, created the ML model, and performed error analysis.

Abhishek Arya undertook the critical tasks of conceptualization, conducting an extensive literature review, engaged in descriptor calculations, and contributed significantly to error analysis.

Abbreviations

YOLO	You Only Look Once
TF	Tensor Flow
ML	Machine Learning
Pd	Pandas
Yolo -v8	Yolo- version8
OS	Operating System

List of Figures

List of Algorithms

1. Introduction

This report explores the exciting world of Tiny Machine Learning (TinyML) and how it can be used to detect license plate in cars. TinyML is operated on low-power microcontrollers boards with extensive hardware extraction [7]. With the advancement in technology there is also an increase a criminal activities and people day by day are bypassing the cyber laws. This project is a simple implementation of object detection with many possibilities yet untouched. TinyML is a cool technology that lets small devices like Rasberry pi do smart things on their own, without needing big computers far away. This opens up lots of possibilities, like keeping an eye on the target vehicle or helping with identifying fake plates.

We focused on using Ultralytics, to make a smart assistant called Yolo-v8. This assistant can zoom out number plates from the crowds. Instead of using a camera, we teach it using a set of pictures and videos we already have, making sure the computer knows where the no. plates are in each picture [1]. This shows how TinyML is versatile and can handle different kinds of data situations.

The report guides you through the steps, starting with getting the dataset ready, teaching the model, and then putting it onto a smart edge board. The goal is to show how this tech can be useful for spotting vehicle number plates, with potential uses in tracking and assisting with any govt. uses. As we go through the steps, we highlight why it's cool to have machines making smart decisions right on the device, without always needing help from big computers somewhere else.

2. Methodology

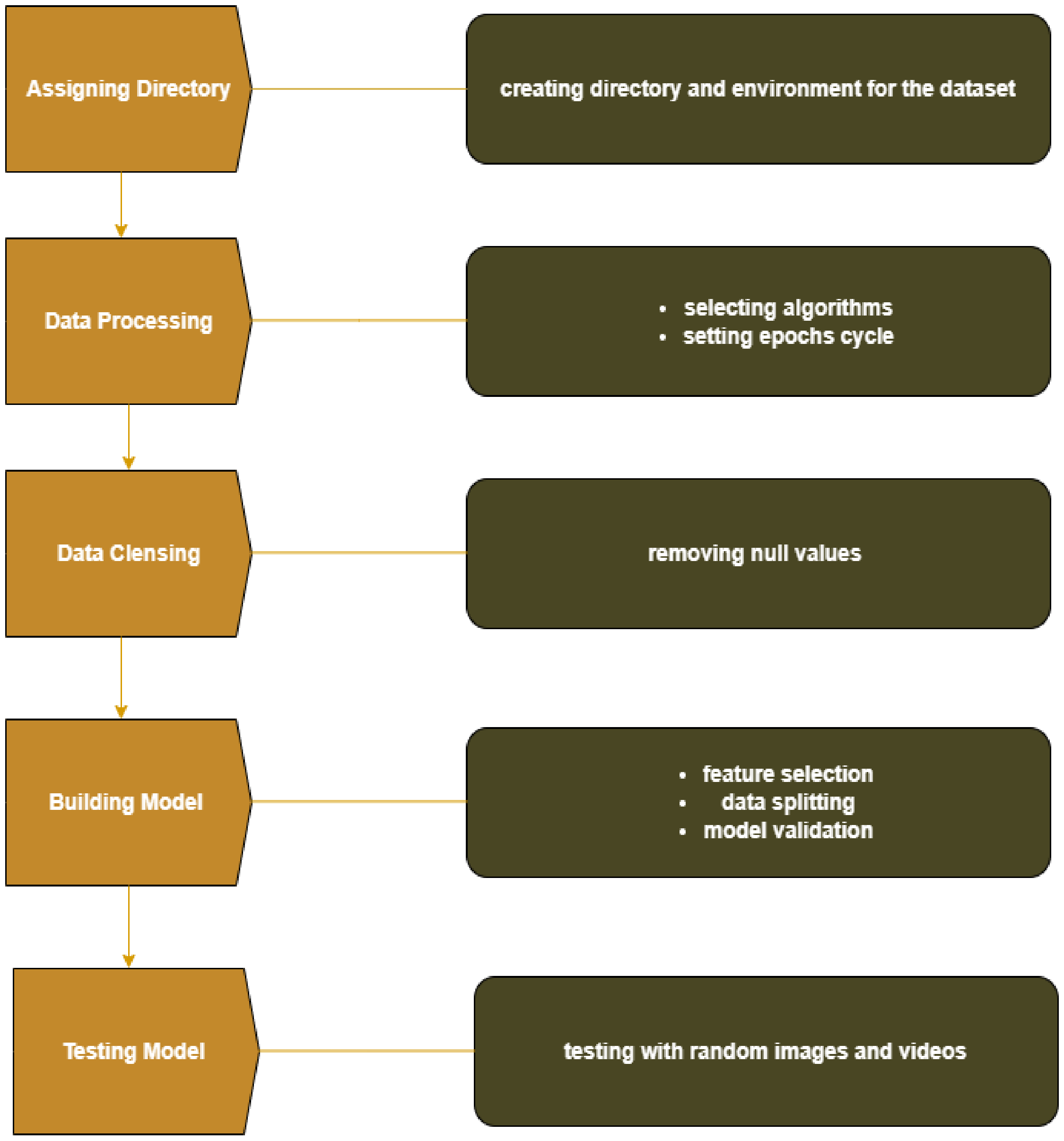


Figure 1: Workflow of the study

Assigning directory:

- Organize the dataset in (YOLOv8) format. Divide the dataset into testing, training and validation sets, specifying the paths
- **(test: ../test/images**

- **train: License-Plate-Detector-2/train/images**
- **val: License-Plate-Detector-2/valid/images**).
- Set up the Python environment with necessary dependencies, including TensorFlow and, Ultralytics. Install required packages and libraries.

Data Processing:

- Utilize the **object_detector.DataLoader.from_yolo_v8** method to load the training and validation datasets.
- YOLO is popular for its real-time object detection techniques, as it processes the entire image in a single pass-through neural network, making it efficient for applications such as video analysis and real-time object tracking.
- **Data Loading:** The training dataset, which consists of labelled images with annotations indicating the presence and locations of objects, is loaded into the training pipeline. Many free and public datasets are relevant to TinyML use cases [8].
- **Forward Pass:** Each image in the dataset is passed through the neural network system. The model processes the input image and generates predictions for the location and class of objects within the image.
- **Loss Computation:** The predictions made by the model are compared to the ground truth labels, and a loss function is computed. The loss represents the difference between the predicted values and the actual values.
- **Backward Pass (Backpropagation):** The magnitude of loss with respect to the model conditions are calculated. This involves iterating the error backward through the network. These gradients indicate how much each parameter should be adjusted to reduce the error.
- **Parameter Update (Optimization):** The model's parameters (weights and biases) are updated using optimization algorithms like SGD or one of its variants. The goal is to minimize the loss, thereby improving the model's ability to make accurate predictions.
- **Repeat:** Steps 2-5 are repeated for each batch of data in the dataset until the entire dataset has been processed. This completes one epoch.
- **Validation:** After one epoch of training, the model's performance may be evaluated on a separate validation dataset to assess its generalization to unseen data and avoid overfitting.

Data Cleansing:

- Begin by identifying which columns or features in the dataset contain null values. This can be done using descriptive statistics or visualization tools to highlight missing values.
- Evaluate the nature and extent of missing data in each column. Understanding the pattern of missing values can help determine the appropriate strategy for handling them.
- One common and straightforward approach is to remove entire rows that contain null values. This is suitable when the missing values are sparse and do not represent a significant portion of the dataset.
- After handling missing values, it's essential to validate that the dataset is now free of null values. Verify the completeness of the dataset and ensure that the chosen strategy aligns with the goals of the analysis or modelling.

Building Model:

- Collect and define a dataset for training the object detection model. Walk through the images to mark the bounding boxes around the objects of interest.
- YOLO typically requires a configuration file that specifies the model architecture, training parameters, and other settings. This file is essential for customizing the model based on your specific requirements.
- Train the YOLO model on defined dataset. This will help in optimizing the model's parameters to reduce the difference between its predictions and the ground truth annotations.

Testing Model:

- Load the pre-trained or trained YOLOv8 model. This involves specifying the configuration file and the weights file obtained after training.
- Provide input data for testing, which can be images, videos, or a real-time video stream. Ensure that the input data is in a format compatible with the model.
- Run the inference process using the specified input data. The model will process the input and generate predictions, including bounding boxes and class labels for detected objects.
- Visualize the output of the model to assess its performance. This may involve drawing bounding boxes around detected objects, displaying class labels, and providing confidence scores.

- Check the performance of the model by comparing its predictions with ground truth definition if available. Common metrics for object detection include precision, recall, and Mean Average Precision.
- Based on the test results, you may decide to fine-tune the model or adjust parameters to improve its performance.
- If the model performs well during testing, deploy it for real-world applications, such as surveillance, autonomous vehicles, or any scenario requiring object detection.

2.2 Comparing Models

2.2.1 Objective detection Performance Comparison

YOLOv5 and YOLOv8 are both popular object detection models, but they have different strengths and weaknesses.

YOLOv5 is a lightweight model that is easy to use and train. It is a good choice for applications that need to be fast and efficient.

YOLOv8 is a more accurate model than YOLOv5, but it is also more computationally expensive. It is a good choice for applications that need to be very accurate.

Model Size	YOLO-v5	YOLO-v8	Difference
Nano	28	37.3	+33.21%
Small	37.4	44.9	+20.05%
Medium	45.4	50.2	+10.57%
Large	49	52.9	+7.69%
Extra Large	50.7	53.9	+6.31%

Fig.2: Yolo version5 vs version 8

2.3 Comparing different Edge devices

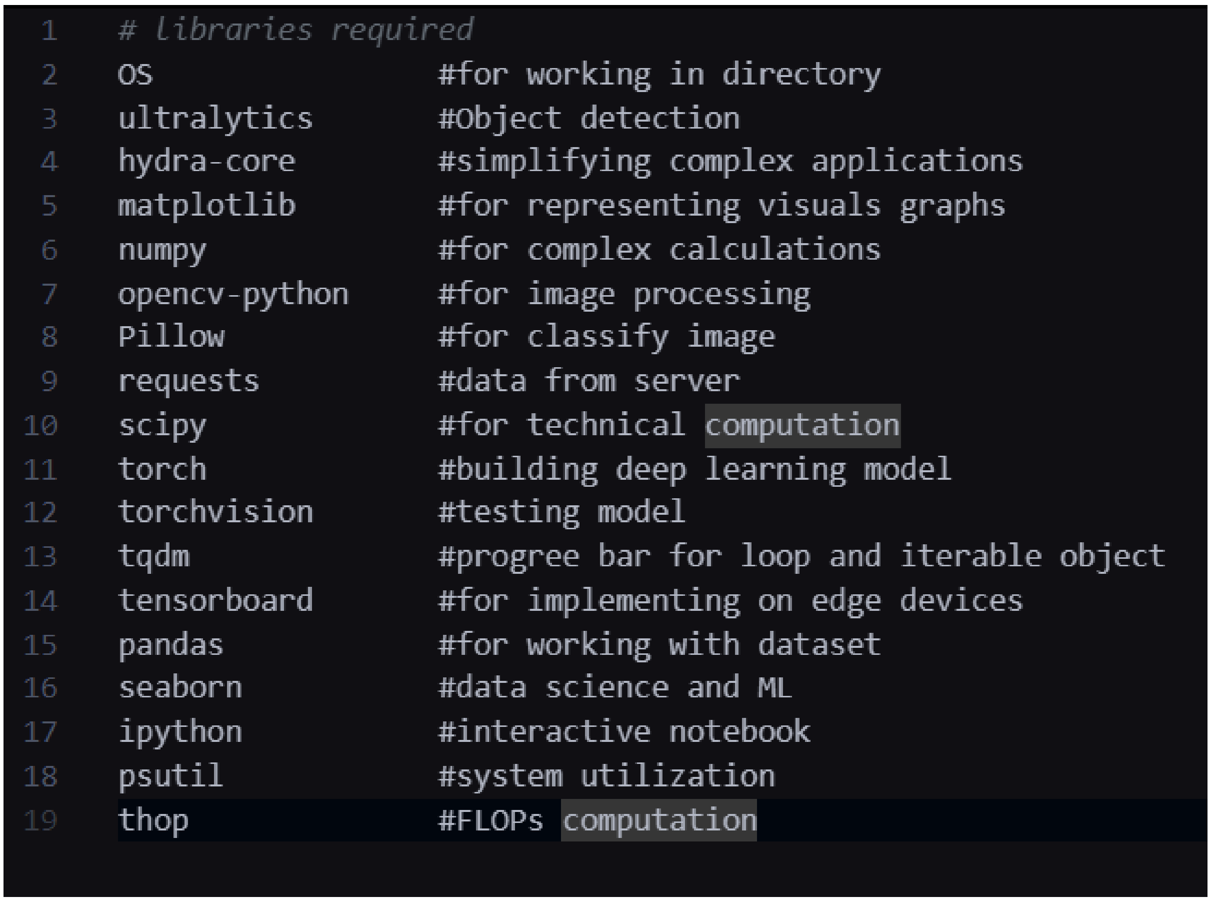
Factors	Raspberry pi 4	Raspberry pi Zero	Arduino giga R1	Arduino Nano
Processor	Broadcom BCM2711 (Quad-core Cortex-A72)	ARM1176JZF-S (1 GHz)	ESP32C3 RISC-V	ATmega328 (16 MHz)
RAM	2GB, 4GB, or 8GB LPDDR4	512 MB	436KB	32 KB (ATmega328)
Storage	MicroSD card	MicroSD card	4MB SPI flash	Flash (usually 32 KB)
Operating System	Raspbian (Linux), various others	Raspbian (Linux)	Depends on application	Depends on application
Cost	Higher	Lower	Medium	Lower

Fig.3: Edge devices comparison

3. Discussion

3.1 Setting up directory and Environment.

In setting up the environment and directory for this project, these libraries in Python played a crucial role. By importing the 'OS' module, I was able to configure the working directory on my local disk. The '**os.mkdir()**' function helps with the creation of file to the designated project path. Additionally, a brief check using '**os.getcwd()**' provided confirmation of the current working directory.

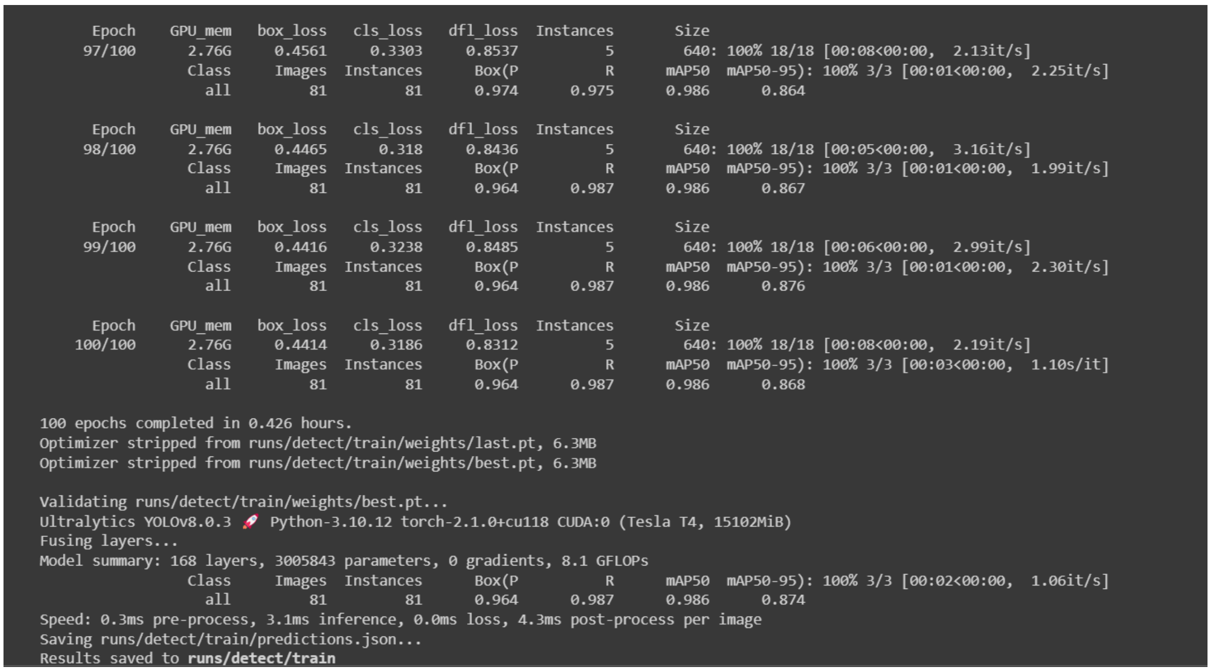


```
# libraries required
OS                  #for working in directory
ultralytics         #Object detection
hydra-core          #simplifying complex applications
matplotlib          #for representing visuals graphs
numpy               #for complex calculations
opencv-python       #for image processing
Pillow              #for classify image
requests            #data from server
scipy               #for technical computation
torch               #building deep learning model
torchvision         #testing model
tqdm                #progree bar for loop and iterable object
tensorboard         #for implementing on edge devices
pandas              #for working with dataset
seaborn             #data science and ML
ipython             #interactive notebook
psutil              #system utilization
thop                #FLOPs computation
```

Fig.4: Libraries Required

3.2 Loading &Training the model.

In the snippet provided below is the implementation of loading, training of the model. Breakdown is explained further.



```
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
     97/100      2.76G     0.4561     0.3303     0.8537          5        640: 100% 18/18 [00:08<00:00,  2.13it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 3/3 [00:01<00:00,  2.25it/s]
                   all         81         81      0.974      0.975      0.986      0.864

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
     98/100      2.76G     0.4465      0.318     0.8436          5        640: 100% 18/18 [00:05<00:00,  3.16it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 3/3 [00:01<00:00,  1.99it/s]
                   all         81         81      0.964      0.987      0.986      0.867

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
     99/100      2.76G     0.4416     0.3238     0.8485          5        640: 100% 18/18 [00:06<00:00,  2.99it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 3/3 [00:01<00:00,  2.30it/s]
                   all         81         81      0.964      0.987      0.986      0.876

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
    100/100      2.76G     0.4414     0.3186     0.8312          5        640: 100% 18/18 [00:08<00:00,  2.19it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 3/3 [00:03<00:00,  1.10s/it]
                   all         81         81      0.964      0.987      0.986      0.868

100 epochs completed in 0.426 hours.
Optimizer stripped from runs/detect/train/weights/last.pt, 6.3MB
Optimizer stripped from runs/detect/train/weights/best.pt, 6.3MB

Validating runs/detect/train/weights/best.pt...
Ultralytics YOLOv8.0.3 🚀 Python-3.10.12 torch-2.1.0+cu118 CUDA:0 (Tesla T4, 15102MiB)
Fusing layers...
Model summary: 168 layers, 3005843 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 3/3 [00:02<00:00,  1.06it/s]
                   all         81         81      0.964      0.987      0.986      0.874
Speed: 0.3ms pre-process, 3.1ms inference, 0.0ms loss, 4.3ms post-process per image
Saving runs/detect/train/predictions.json...
Results saved to runs/detect/train
```

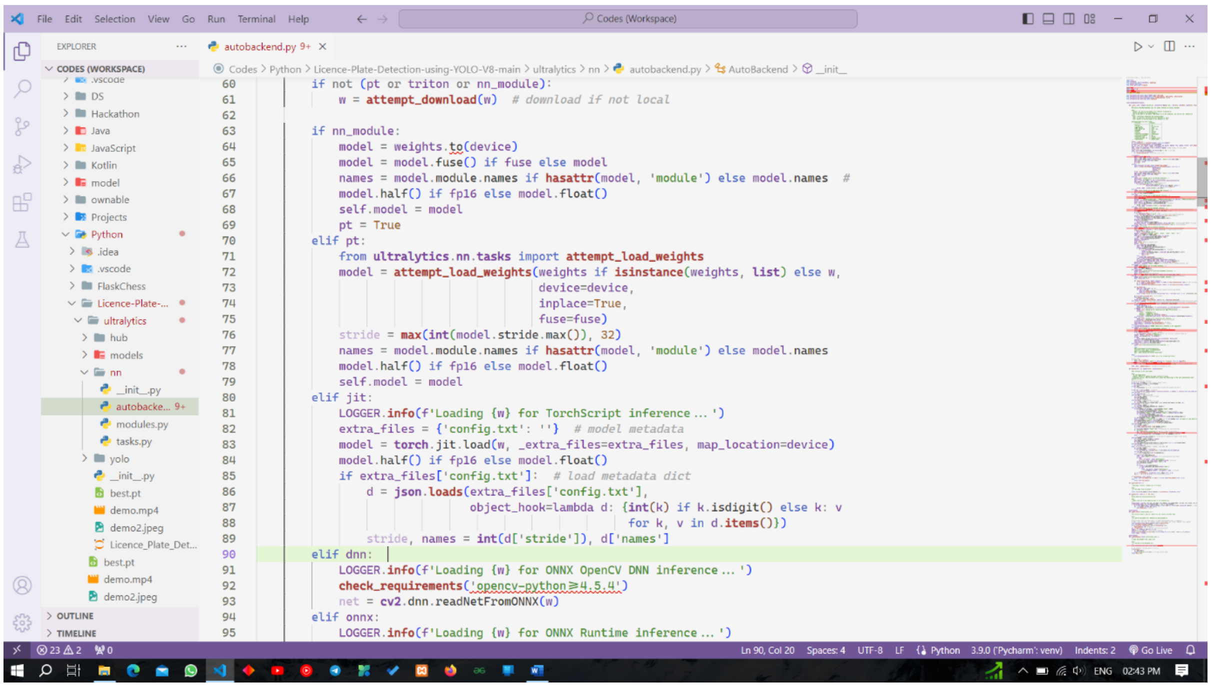
Fig. 5: Model Training

Here, training and validation datasets are loaded using the **from_pascal_voc** method from the **object_detector.DataLoader** class. This method is suitable for datasets in Pascal VOC format. **TRAIN_DATASET_PATH** and **VALID_DATASET_PATH** are the paths to the training and validation datasets, and **CLASSES** is a list of object types present in the datasets.

The code tells about the model architecture using **model_spec.get(MODEL)**, where **MODEL** is a variable representing the chosen model architecture. The **object_detector.create** function then constructs the object detection model using the specified architecture (**model_spec**), training data (**train_data**), and other parameters such as batch size, training duration (**epochs**), and validation data (**val_data**).

3.3 Building Model

Precision settings are adjusted based on our device, and the model is configured with specific backend, device, and precision options. Relevant attributes such as stride, pt, Jit, and engine are extracted from the model. The image size is checked and set accordingly. Batch size and device settings are handled based on the type of model engine. ANPR consists of 4 phases: - Preprocessing, number plate extraction, character segmentation, character recognition [5]



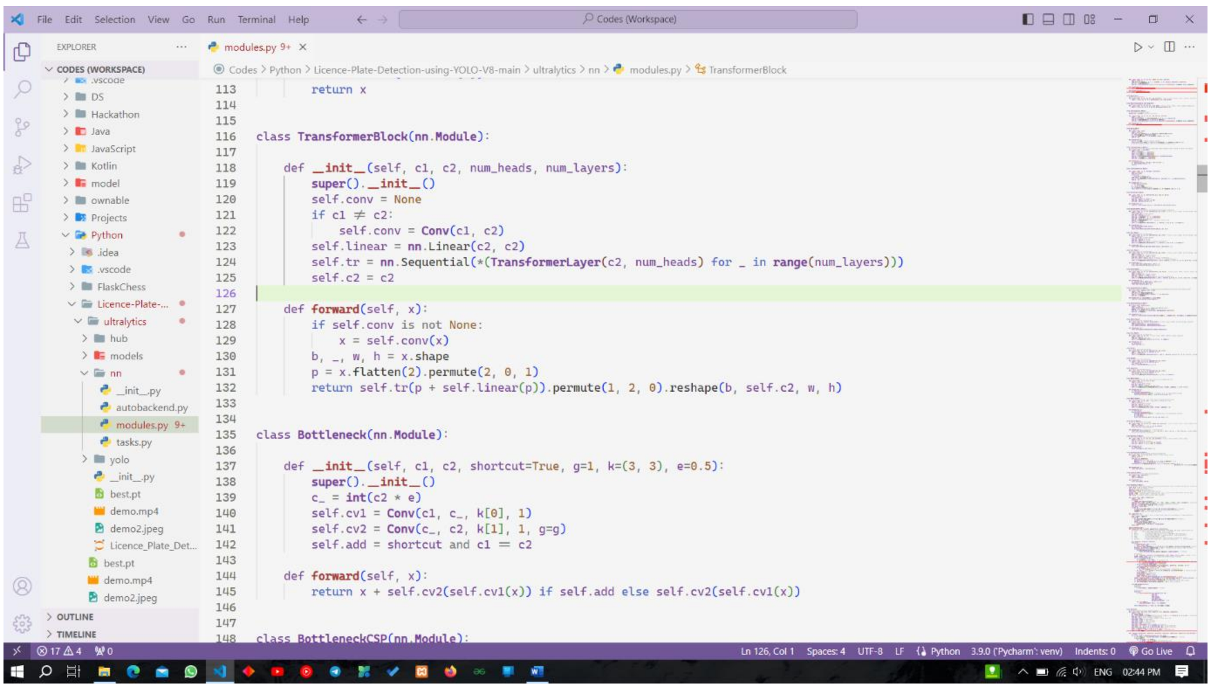
```
        if not (pt or triton or nn_module):
            w = attempt_download(w)  # download if not local

        if nn_module:
            model = weights.to(device)
            model = model.fuse() if fuse else model
            names = model.module.names if hasattr(model, 'module') else model.names  #
            model.half() if fp16 else model.float()
            self.model = model
            pt = True
        elif pt:
            from ultralytics.nn.tasks import attempt_load_weights
            model = attempt_load_weights(weights if isinstance(weights, list) else w,
                                         device=device,
                                         inplace=True,
                                         fuse=fuse)
            stride = max(int(model.stride.max()), 32)
            names = model.module.names if hasattr(model, 'module') else model.names
            model.half() if fp16 else model.float()
            self.model = model
        elif jit:
            LOGGER.info(f'Loading {w} for TorchScript inference...')
            extra_files = {'config.txt': ''}  # model metadata
            model = torch.jit.load(w, _extra_files=extra_files, map_location=device)
            model.half() if fp16 else model.float()
            if extra_files['config.txt']:  # load metadata dict
                d = json.loads(extra_files['config.txt'],
                               object_hook=lambda d: {int(k) if k.isdigit() else k: v
                                                      for k, v in d.items()})
                stride, names = int(d['stride']), d['names']
        elif dnn:
            LOGGER.info(f'Loading {w} for ONNX OpenCV DNN inference...')
            check_requirements('opencv-python>=4.5.4')
            net = cv2.dnn.readNetFromONNX(w)
        elif onnx:
            LOGGER.info(f'Loading {w} for ONNX Runtime inference...')
```

Fig.6: Building Model

3.4 Testing & Result

For training and testing of the data, we performed Machine Learning with the help of Yolo version 8. To train the data we use 100% of the data we recorded/supplied and for testing we used any random image from the data or a real time video/image can also be used.



modules.py 9+
Codes > Python > Licence-Plate-Detection-using-YOLO-V8-main > ultralytics > nn > modules.py > TransformerBlock

```
        return x


class TransformerBlock(nn.Module):

    def __init__(self, c1, c2, num_heads, num_layers):
        super().__init__()
        self.conv = None
        if c1 ≠ c2:
            self.conv = Conv(c1, c2)
        self.linear = nn.Linear(c2, c2)
        self.tr = nn.Sequential(*(TransformerLayer(c2, num_heads) for _ in range(num_layers)))
        self.c2 = c2

    def forward(self, x):
        if self.conv is not None:
            x = self.conv(x)
        b, _, w, h = x.shape
        p = x.flatten(2).permute(2, 0, 1)
        return self.tr(p + self.linear(p)).permute(1, 2, 0).reshape(b, self.c2, w, h)


class Bottleneck(nn.Module):

    def __init__(self, c1, c2, shortcut=True, g=1, k=(3, 3), e=0.5):
        super().__init__()
        c_ = int(c2 * e)
        self.cv1 = Conv(c1, c_, k[0], 1)
        self.cv2 = Conv(c_, c2, k[1], 1, g=g)
        self.add = shortcut and c1 == c2

    def forward(self, x):
        return x + self.cv2(self.cv1(x)) if self.add else self.cv2(self.cv1(x))


class BottleneckCSP(nn.Module):
```

Fig 7: Testing Result

Input Image:

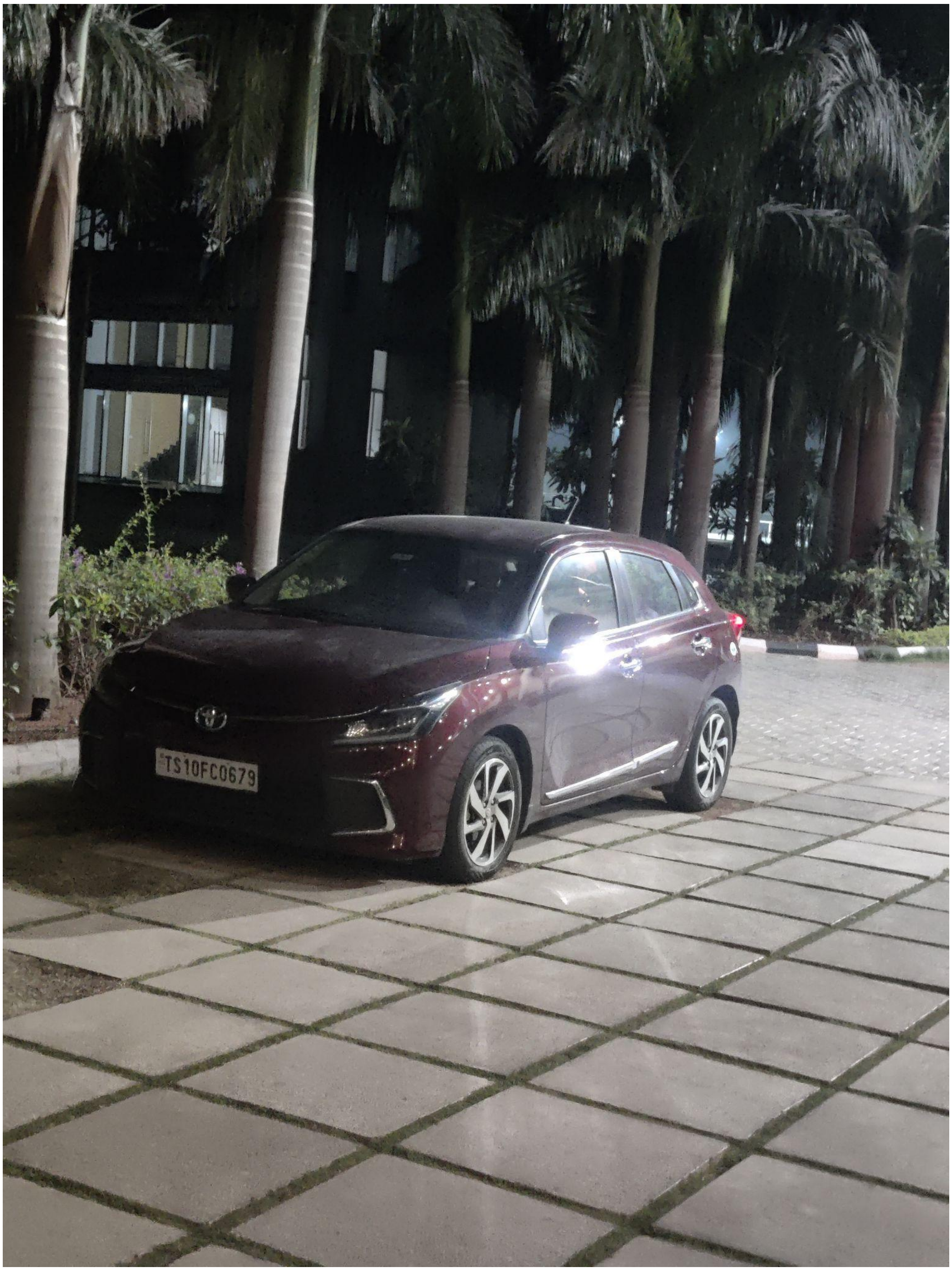


Fig 8: Input Image

Output Image:

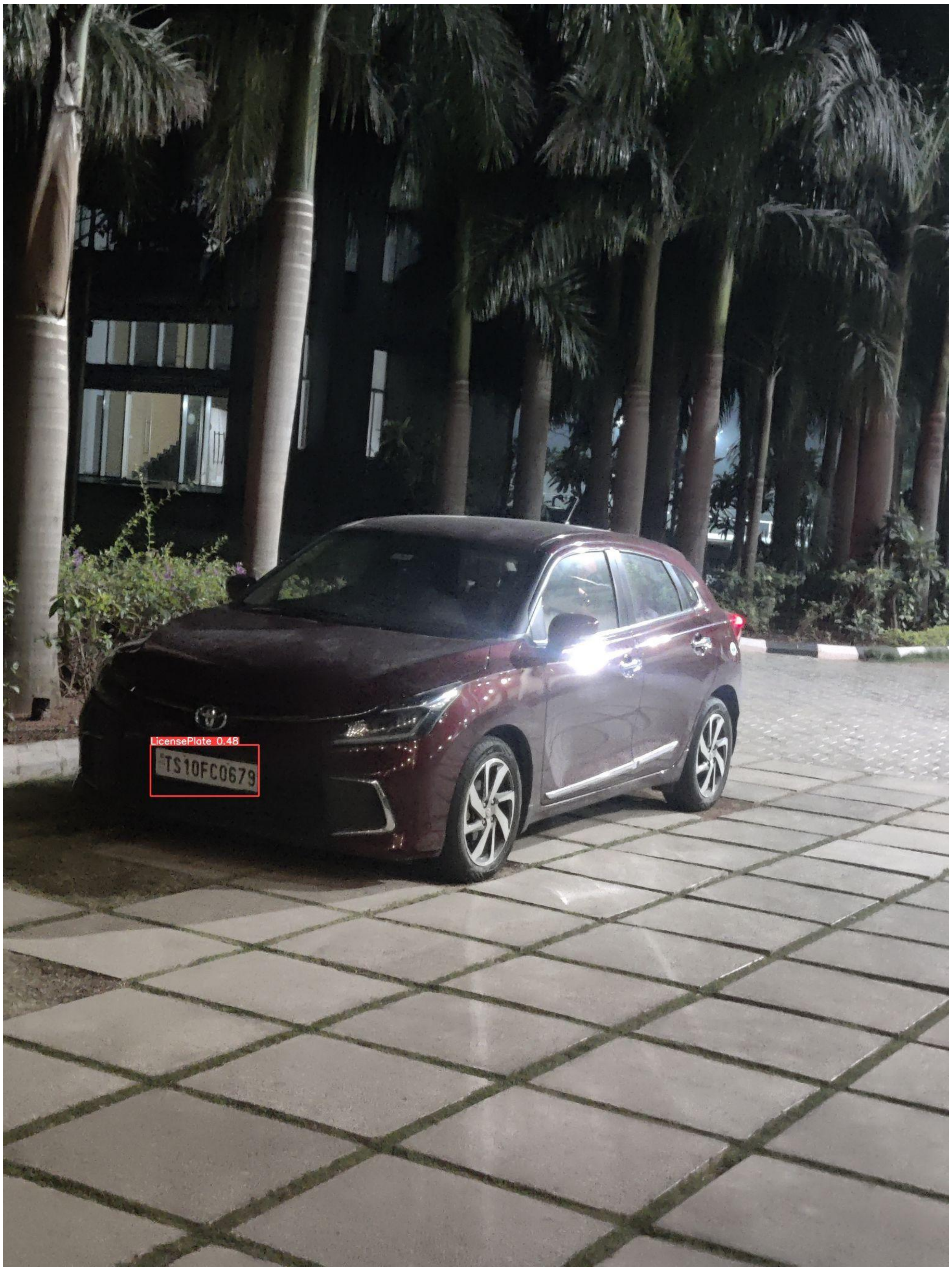


Fig 9: Output Image

4. Concluding Remarks

In summary, this project successfully combined Tiny Machine Learning (TinyML) with the Yolo version 8 algorithm given by Ultralytics for License Plate detection on edge devices. The collaboration between Yolo v8, a lightweight object detection method, and Pytorch, a strong image processing tool, highlighted the potential of using machine learning on small devices. As we explore the world of TinyML and edge computing, this project demonstrates the opportunities and challenges in bringing smart capabilities to compact devices. TinyML is an essential and fast-developing field that requires trade-offs among diverse integral components (hardware, software, machine learning algorithms) The integration of TinyML, Ultralytics, and PyTorch on edge devices shows promise for creating smart, independent systems capable of making quick decisions in real-time. This paves the way for future innovations where machine learning meets edge computing.

5. Future Work

The future of object detection using TinyML (Tiny Machine Learning) holds significant promise and is likely to see several advancements. TinyML refers to the deployment of machine learning models on small, resource-constrained devices, such as microcontrollers and edge devices. Continued efforts will be made to optimize object detection models for deployment on resource-constrained devices. This includes model quantization, pruning, and other techniques to reduce model size and computational requirements while maintaining acceptable accuracy. As TinyML gains traction, education and training programs will become more widespread, fostering greater adoption of TinyML technologies. TinyML has the potential to exploit an entirely new domain of smart applications throughout manufacturing and business and personal life areas.[9] Developers, engineers, and businesses will increasingly recognize the benefits of incorporating TinyML into their applications. the future of object detection using TinyML holds exciting possibilities, driven by advancements in model efficiency, hardware capabilities, and the growing need for edge-based, real-time AI applications.

6. References

[1] Ultralytics YOLOv8 Docs. URL: https://docs.ultralytics.com/

[2] Pytorch Documentation: https://pytorch.org/docs/stable/index.html

[3] Real State of The Art: https://medium.com/mlearning-ai/yolo-v8-the-real-state-of-the-art-eda6c86a1b90

[4] Savneet Kaur, Kamaljit Kaur (2014). An Automatic System for Detecting the Vehicle Registration Plate from Video in Foggy and Rain Environments Using Restoration Technique. International journal of Computer Applications (0975- 8887) Volume 97.

[5] Sarbjit Kaur, S. K. (2017). An Efficient Approach for Automatic Number Plate Recognition Under Image Processing. International Journal of Advanced Research in Computer Science Vol. 5 (3).

[6] B. Sudharsan et al., "TinyML benchmark: executing fully connected neural networks on commodity microcontrollers," in IEEE World Forum on Internet of Things, pp. 19–21

[7] V. J. Reddi et al., "Widening access to applied machine learning with tinyML," 2021.

[8] C. R. Banbury et al., "Benchmarking tinyML systems: challenges and direction," arXiv preprint arXiv:2003.04821, 2020. [Online]. Available: http://arxiv.org/abs/2003.04821.

[9] S. Soro, "TinyML for ubiquitous edge AI," 2021.