# COMPUTER VISION: OBJECT TRACKING USING SELF-BALANCING TREE FOR OPTIMIZED DATA PROCESSING

**A Thesis By**

**CHRISTOPHER CHAN VI**
**ORCID iD: 0000-0003-3228-3212**

**California State University, Fullerton**
**Fall, 2021**

---

**In partial fulfillment of the degree:**
Master of Science, Computer Science

**Department:**
Department of Computer Science

**Committee:**
Shawn X. Wang, Department of Computer Science, Chair
Christopher Ryu, Department of Computer Science
Anand Panangadan, Department of Computer Science

**DOI:**
10.5281/zenodo.5765023

**Keywords:**
computer vision, machine learning, artificial intelligence, robotics, AVL self-balancing binary search tree, object tracking detection

**Abstract:**
Tracking Multiple Objects and Prioritizing them can be an expensive operation. In a real-world scenario where, for example, many cars are being tracked coming through a highway by a certain characteristic such as car color this leads to one needing to process them efficiently, especially when the cars are quickly coming in and out of the screen. This research aims to see if there is an optimal way to process and sort many tracked objects efficiently; thus, the purpose of this Thesis Research Project is to answer the central question of: “whether the AVL self-balancing Binary Search Tree can be used for optimization of data processing from a Computer Vision Robotics camera source?”

The Tree data structure is used in many areas of Computer Science and Software Engineering; there are different variations, each with different characteristics for various applications. This research uses the AVL (Adelson-Velsky and Landis) self-balancing binary search tree to see differences in Asymptotic Runtime for insert and traversal operations. A number of specialized hardwares and softwares will be used to conduct this research with rigorous test benchmarking to get the most accurate results. The results of this Thesis Research Project shows the AVL tree being stable with the complexity running time maintaining O (logn) for the tested operations with varying batch data sizes making it feasible for post-processing use in Computer Vision and other industry applications.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

## ACKNOWLEDGMENTS

I would like to thank California State University Fullerton for giving me the opportunity to study at the university and pursue research.

# CHAPTER 1

## INTRODUCTION

Computer Vision with Robotics is a more recent field that has stemmed from the field of Artificial Intelligence in Computer Science. Advancements have been made to fields related to Artificial Intelligence such as Machine Learning and Computer Vision. Computer Vision has made more recent advancements due to the rise of its use in real-world applications such as the auto industry and health/medical sector.

### The History of Computer Vision

Computer Vision has made progress since the 1950s and 1960s; the 1950s is composed of Frank Rosenblatt's Perception which marked the early idea of Artificial Intelligence and has stepping stones to what we have today for Deep Learning and Neural Networks. The 1960s marked the beginning in academia where Artificial Intelligence became a field of study leading to the quest for research into solving the problem of human vision; a major landmark was reached in 1963 when transforming two-dimensional images into three-dimensional forms by computers was successful, this marked an era where computers can digital process and store images.

Leading up from the 1970s to the 2000s Smart cameras of those made from Xerox paved the way for layered image processing; the 1970s brought about optical character recognition (OCR) technology with its precedence & relevance still present today in current technology. Coming up to the 1980s scientists have come up with algorithms for machines that can precisely identify shapes, curves, and corners; convolution layers in neural networks at the same time were being used to recognize patterns from a network called The Neocognitron. In the 1990s the Internet and the dot-com boom started to make its way to households, the 2000s made a milestone for facial recognition and object detection through the Viola-Jones object detection framework. This has led up to 2005 where Japan unveiled the first mobile phone with working facial recognition features; in 2010 AI's object recognition error rate surpasses those of human's, 2012 marked Google's X Lab's neural network which looks for cat videos when YouTube is accessed.

The year 2014 made Tesla headlines for their autopilot feature in its Model S electric car. In 2015 Google incepted FaceNet for its facial recognition project and in 2016 the Vision Processing Unit (VPU) microprocessors are created for Machine Vision. Leading up all the way to the present Machine Learning frameworks such as scikit-learn, PyTorch, and TensorFlow have made significant progress to allow computers to model Object Detection from a computer camera source. Robotics have also made advancements and with many of them equipped with cameras it's a landmark time for Robotics with Machine Learning training them to move with a camera source to guide their movement and detect objects.

## The History of Robotics

The history of Robotics can be dated back to the early 1900s where the idea of automation stemmed from Henry Ford's Auto Assembly Line in 1913 and paves the idea of robotics and automation innovation for the future. In 1932 the first toy robot was released in Japan where it is heighted at 15cm with the ability to walk. In 1950 The Turing Test was conducted to test whether Artificial Intelligence was good enough to make computers talk where the voice from that of a human and a computer is indistinguishable.

The year 1954 brought more computers and machinery's into factories and as the development of robots advances the first robot was put into the General Motors assembly line in 1961. 1966 bring of a mobile robot named Shakey from The Artificial Intelligence Center at Stanford; Shakey can perceive its environment and is controlled by a computer. In 1979 Stanford introduces The Stanford Cart which stems from research of Shakey and is equipped with a full television camera for vision and is able to navigate a path to chairs, self-program itself, and learn which was considered a milestone in Artificial Intelligence.

The year 2000 brough Honda out to unveil its latest and long-awaited robot from decades' worth of research, its name is ASIMO and it stands at 4 foot 3 inches and has computer vision that can detect objects and can navigate autonomously; this brough the realization of the dreams of a humanoid machine as more progress is researched from it for the future. Coming up to the present

moment Robotic companies have come about and have released Robotic kits for learning, research, and side projects; companies & foundations such as Nvidia and Raspberry Pi have developed portable artificial intelligence/machine learning computers that can be integrated with a number of robotic kits out there.

## Research Objectives

The aim of this research is to see if there is an optimal way to process and sort many tracked objects efficiently using an AVL Self-Balancing Binary Search Tree with data from a Computer Vision source. The key objective of this Thesis Research Project is to benchmark the AVL Tree for the insertion and traversal operations runtime which will help determine the feasibility of using a BST data structure to process data from a Computer Vision source; the means of getting there will involve test data of varying sizes. The central question to answer is: “whether the AVL self-balancing Binary Search Tree can be used for optimization of data processing from a Computer Vision Robotics camera source?"

The rest of the paper is organized as follows. Section II presents the related works of existing Computer Vision Object Tracking performance improvements, AVL Binary Search Tree improvements, and GPU/CUDA/Jetson Nano improvements. Section III describes Computer Vision Object Tracking Detection. Section IV lays out the background information on the AVL Binary Search Tree. Section V describes the background information on the GPU/CUDA/Jetson Nano. Section VI provides the detail of the design concept and principle of the system. Section VII describes the Experimental Settings and Implementation. Section VIII lays out the Experimental Results and Analysis. Lastly, Section IX goes into the Conclusions & Future Work.

# CHAPTER 2

# RELATED WORKS

## Computer Vision Object Tracking Detection

There are a number of performance related work to Computer Vision, each with different ways of improving performance in their respective area.

### Research 1

Qiu et al. [1] in their research titled “Two motion models for improving video object tracking performance” proposed two motion models to improve video object tracking performance for Visual object tracking (VOT) in Computer Vision. The first model tracks the random walk motion (RW) and the second model is a data-adaptive vector auto-regressive (VAR) model that tracks the regular motion patterns to retrieve information from both models prior (predicts from past trajectories) [1].

In their VOT experiment 1 which corresponds to the Random walk state transition model, given the help of the Random Walk (RW) model the Quantitative tracking performance testing was conducted resulting in performance improvement; the precision of the three tracking (NCC, NSAMF, CFNet) algorithm increased overall along with the AUC score leading to an overall positive note for the RW model helping improve the tracking performance [1].

Subsequently in their VOT experiment 2 which refers to the VAR state transition model, given the help of the Vector Auto-Regressive (VAR) model the Quantitative tracking performance testing was conducted resulting in performance improvement; the precision of the three tracking (NCC, NSAMF, CFNet) algorithm increased overall along with the AUC score leading to an overall positive note for the VAR model helping improve the tracking performance [1].

In their final VOT experiment 3 which is a fusion of the two models there is a 3-sigma range which determines if the RW model is used or the VAR model. Performance wise, the fusion performance averages out better overall for both AOR and CLE, also proving that both prior models have a chance of improving the original performance (built-in) [1].

***Conclusion***

In conclusion, they concluded that the information from both models prior results to performance enhancements of all three trackers. They note that the RW model works for videos that are sufficiently annotated and the VAR model geared towards applications without references. Experiment wise the results indicate better yielding of tracking results for refined models [1].

***Challenges***

A potential challenge they mention is “general visual object tracking is a challenging task due to occlusion, varying appearance, changing scales, rapid motions, and other factors.” suggesting that this is an important application area in Computer Vision that can be improved [1].

**Research 2**

In another performance related research conducted by Karasulu et al. [2] they investigated a method known as background subtraction (BS) for a performance optimization in relation to moving object detection and tracking in videos as these are important initial steps for higher-level video analysis applications in Computer Vision [2].

The BS method is used as a simple method in video sequences to detect moving objects, the idea is to take a fixed background in front of the observed moving objects this results in the moving object being detected by the frame differencing method. The simulated annealing (SA) method modified for the background subtraction method (SA-BS) is proposed to determine the foreground-background segmentation optimal threshold for object detection by learning the background model. From their testing of the BS method and the SA-BS method they found the SA-BS more preferable than the regular BS method [2].

***Conclusion***

In conclusion, they found SA-BS method more preferable than the regular BS method through their performance testing, results, and comparisons. The SA is a solved optimization problem of the BS stages; the stages are composed of “optimal threshold determination” and “determination of

learning rate of frame differencing". This results in their approach (SA-BS) being more robust and accurate for object detection & tracking than the regular BS method [2].

### *Challenges*

A potential challenge they concluded is further improvement (s) of their method, this would involve figuring out additional new parameters for optimization of related methods [2].

## AVL Self-Balancing Binary Search Tree

AVL Self-Balancing Binary Search Trees have been around since 1962 and have seen a number of proposed improvements to them.

### Research 1

"An Evaluation of Self-adjusting Binary Search Tree Techniques" conducted by Bell et al. [3] aims to look into the techniques and performance of a number of Binary Search Trees taking into account tree sizes, levels of key-access-frequency skewness and ratios of insertions and deletions to searches [3].

Their findings, given the initial procedures which consists of 4095 keys and 100,000 operations, is the AVL tree has the lowest operation compute cost for operations insertion, deletion, and searching given that the search-key distribution is uniform; it should be noted as well that the low maintenance cost attributes to good CPU time. Given a highly-skewed key access they found the AVL tree is just about 25 percent faster compared to the best self-adjusting Binary Search Tree technique (top-down splaying) [3].

### *Conclusion*

In conclusion, they found that the AVL tree in most cases outperforms all the self-adjusting trees; they also note though that the random binary search tree in many cases has better performance than the self-adjusting trees CPU time wise with highly dynamic environments being a strong point of self-adjusting trees. In all, with the skew factor close to 1 the AVL tree's search performance in most cases is better than that of any of the self-adjusting Binary Search Trees and the top-down splaying being the best in most situations of the self-adjusting techniques observed [3].

***Challenges***

A challenge they encountered is Zipf's law where it does not allow them to generate a number of distributions with various degrees of skewness. Zipf's law is a common probability distribution that is used for skewed distributions [3].

**Research 2**

Another AVL Binary Search Tree research conducted by Tripathi [4] looks into using Virtual Nodes to balance the AVL Tree. The idea is using a Virtual Node as a hypothetical node which is inserted into the BST using an inorder traversal (left, root, right); this results in a BST tree until the Virtual Node gets deleted which then makes it an AVL tree [4].

In more detail the "Virtual node is the hypothetical node whose value is greater than the values of nodes in left subtree but less than the values of nodes in right subtree. If we get only three nodes into the traversal, there is no need to insert a virtual node. The mid node will be the root and the predecessor of the mid will be the left child and successor of the mid will be the right child [4]."

For example, when two new values need to be inserted where the values are greater than the previous three nodes then two new virtual nodes will be inserted below the root as subtrees where then the two new values will be inserted under the right virtual node, if the two new values were less than the previous three nodes then the two values are inserted under the left virtual node. After deletion of the virtual nodes we now get an AVL tree [4].

***Conclusion***

From the research, he concluded studies on the Virtual Node where complexity is the same taking into account rotations with the implementation being easier; the balancing of the BST in this research is for both insertion and deletion operations. The result being it becomes easy to balance a Binary Search Tree [4].

***Challenges***

A potential challenge could be more memory consumption up front for the Virtual Node (s) as the BST expands by magnitudes [4].

**Research 3**

A research conducted by Zhang et al. [5] looks into the “Concurrent Manipulation of Expanded AVL Trees” the idea behind this research is being able to perform full concurrent processes/operations for searching, insertion, and deletion on a tree. They want to further raise new questions, ideas, and insights from this research in the domain of concurrent search structure manipulation [5].

The data to be processed on the proposed EAVL tree are stored in a single shared global memory where one of the insertion, searching, and deletion operations can be performed per process which is asynchronous and independent. Simulation results for the performance of the system is conducted by considering the degree of concurrency and the degree of the degradation of the tree structure [5].

***Conclusion***

From the simulation they found the system to benefit from a high degree of concurrency with the tradeoff of marginal degradation of the tree structure [5].

***Challenges***

A challenge they mentioned is **“**When processes are allowed to manipulate the tree concurrently, they may introduce temporary redundancy and/or unbalance into the tree simultaneously [5].”

## GPU/CUDA/Jetson Nano

GPU Processing technology has advanced a number of ways since it’s inception outside of Graphics processing to leverage itself to compute for other applications using GPU cores. For this Thesis research project the Nvidia CUDA (Compute Unified Device Architecture), a parallel compute platform and API, is used to process Computer Vision tasks.

**Research 1**

A research from Cervera [6] looks into the CUDA module integrated with the OpenCV framework to see the benefits of their combined use specifically to encourage it’s use in the area of

Robotics as Computer Vision is associated with Image Processing and utilizes parallel computation which modern GPUs are capable of [6].

A number of benchmarks were ran the most notable one being the ORB Feature Extraction which is an algorithm used for feature extraction and takes into account performance and computational costs; it is benchmarked under OpenCV. They tested the ORB Feature Extraction algorithm using CPU and GPU CUDA source versions with a Desktop, Laptop, and Embedded PC (Nvidia Jetson Nano) as test machines [6].

The results being the GPU processing resulting in a lower computation time overall. Another benchmark conducted is the computation of the edge detection algorithm which is a less demanding operation with the results being the opposite of the previous benchmark, in the sense that the CPU Processing is faster than the GPU [6].

***Conclusion***

Overall, he concluded that CUDA for OpenCV is good for Robotics projects using GPU acceleration through CUDA. With most notable being that simple processes/tasks are better suited for CPU computation whereas more complex processes are better suited for the GPU. Example of complex processes can include image operations, parallelization use, and Deep Learning. Thus, he encourages Robotics researchers to migrate over to GPU computation [6].

***Challenges***

A potential challenge noticed in the paper is the mentioning of the data being loaded into the CPU memory first before being transferred to the GPU memory which results in extra processing time/overhead [6].

**Research 2**

A research conducted by Wang et al. [7] looks into OpenCL GPGPU Co-Processing on Mobile SoCs for Computer Vision accelerators. With a focus on Mobile devices they implemented a Computer Vision algorithm for the removal of image inpainting-based objects. They researched into OpenCL which is an open compute platform and formulated optimization strategies primarily to

determine compute capabilities on Mobile SoC devices, specifically in this case the Snapdragon mobile SoC [7].

They applied an exemplar-based inpainting algorithm for object removal for a test platform that is Snapdragon based and uses OpenCL. They implemented a findBestPatch Kernel function which is an OpenCL based implementation. Through Search Space optimization, Memory optimization and Patch Size optimization they found that increasing the patch size while reducing the search area leads to reduced processing time [7].

***Conclusion***

They concluded, given the OpenCL-based function implementations, the Mobile SoC with CPU-GPU heterogeneous implementation version reduces the processing time compared to the single-threaded CPU version. With the proposed optimizations implemented the processing time can further be reduced. In conclusion, given the experimental/test results from the OpenCL based implementations Mobile SoCs on real-world mobile devices can run more complex Computer Vision applications [7].

***Challenges***

A challenge presented in the paper states [7] “The algorithm sometimes can lead to false matching, in which the reported “best” patch with a high correlation score may have very distinctive textural or color information compared to the object patch”. Another possible challenge given their test datasets along with the specifications of the images could suggest that higher resolution images could pose a computational power challenge, but given the fast advancements of Mobile SoCs it’s possible to keep up with the advancements of the HD resolution standards as well [7].

***Overall***

Existing researches on Computer Vision Object Tracking Detection looks to improve the performance using methods such as Simulated Annealing and Background Subtraction to determine the foreground-background segmentation optimal threshold for object detection by learning the

background model to further optimize the performance. Evaluation reports were done on the AVL tree which supports the use of it given the benefits such as computational cost.

Also, a number of improvements to the AVL BST are proposed, usage and performance-wise, which for example, demonstrates that it can run concurrently with a minimal tradeoff in Tree Structure. The benefits of GPU/CUDA/Jetson Nano are documented [6], [8]-[11] as well which proves GPU processing is beneficial for Computer Vision tasks as GPUs have many cores that are used to enable high degree of parallel computation [8], [10], [12] on large blocks of data.

# CHAPTER 3

## COMPUTER VISION OBJECT TRACKING DETECTION BACKGROUND

Early Development of Computer Vision Object Tracking Detection can be traced back to about the year 2012 with Computer Vision, in general, earlier where AlexNet won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Real world application uses can include Robotics, Surveillance Cameras, Car Auto-Pilot, Character/Optical Recognition, Medical Imaging, etc.

Examples of Robotics companies can include Nvidia, Yahboom, and DJI. Nvidia has Robots such the Waveshare Jetbot & Seeed Jetbot, Yahboom has Robots such as the Jetbot & 4WD Smart Robot, and DJI has Robots such as the RoboMaster EP Core & RoboMaster TT all of which are good for learning and projects use. Car companies such as Tesla utilizes Computer Vision for their Auto-Pilot feature.

Computer Vision tasks includes Object Detection, Object Tracking, Image classification, and Content-based image retrieval. Object Detection refers to the classification of images and highlights the object in the video or picture, Object Tracking refers to following the object once it is detected, Image Classification observes the image and classifies it, and Content-based Image Retrieval refers to using Computer Vision to selecting, finding, and searching images from a datastore based on the objects in the image rather than using the metadata associated with the images. Computer Vision utilizes Detection Algorithms/Methods for Object Detection & Tracking.

### Detection Algorithms

Detection Algorithms in popular use includes YOLO (You Only Look Once), SSD (Single Shot MultiBox Detector), and R-CNN (Region Based Convolutional Neural Networks). Detection Algorithms is what the camera hardware uses to process images and apply the bounding box (s) to before displaying it to the user. Looking at the YOLO algorithm first there are a number of characteristics which makes it a popular choice for use [13].

## YOLO

The YOLO (You Only Look Once) algorithm, as shown in Figure 1, is regression based and uses a single neural network for predicting class probabilities and bounding boxes from a whole image in one algorithm run; it is currently considered one of the most precise and accurate and is based on a modified architecture called Darknet. It is based on a Convolution Neural Network architecture that spans a number of layers with layers from DarkNet-19 and extra layers for the detection of natural objects utilizing the MS COCO dataset [13].

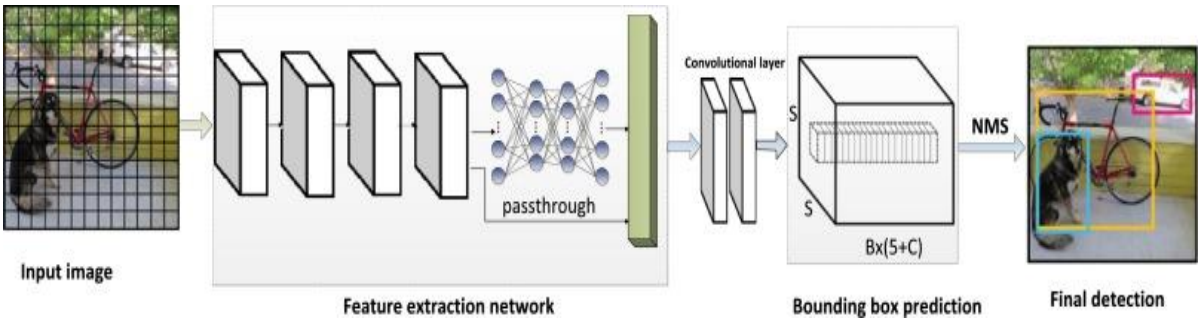


*Figure 1.* YOLO Network showing inputted image being processed for detection.

The algorithm is considered precise and speedy with a note that it struggles with small object detection and small pixels in pictures. YOLOv3 is considered one of the best modifications to YOLO that makes it even more precise by using multiple layers; it is composed of the latest darknet features such as the 53 layers and uses one of the most reliable dataset, ImageNet, to train on. With these changes the architecture is considered one of the top in terms of response time with accuracy taken into consideration. It also improves the processing of small pixels in whole image (s) which was considered a need for improvement in prior versions; it comes at the cost of some processing time but is still considered high performing overall [13].

For feature extraction & analysis YOLOv3 uses a logical regression where it computes an objective score which leads to the framing of the bounding box on the object. The bounding box is shown in Figure 2 with the final equation depicted in Equation 1; for finding the best bounding boxes the grouping of K-Means is used with DarkNet-53 being used to extract the feature [13].

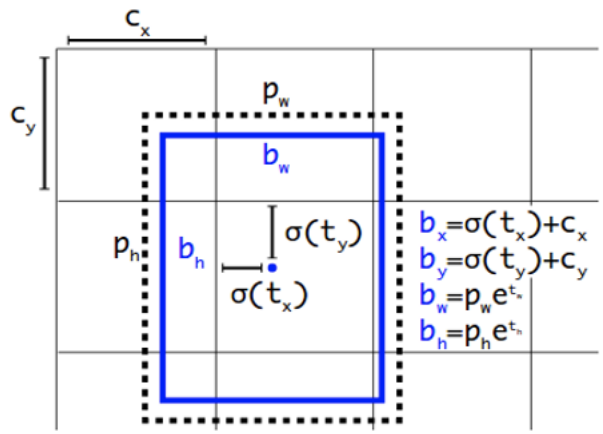


*Figure 2.* Predict the width and height of the box as offsets from cluster centroids and predict center coordinates using a sigmoid function. See Equation 1 for the final equation.

$$\sigma\,(x) = 1/(1 + e^{-x}) \tag{1}$$

Overall, YOLOv3 is hailed by many as a milestone update which brought improvements in terms of preciseness and utilizes the DarkNet-53 architecture which brings a number of improvements as well. Subsequent version looks to bring improvements as well such as in YOLOv4 and YOLOv5. Performance benchmarks for YOLO are depicted in Figure 3 for reference [13].

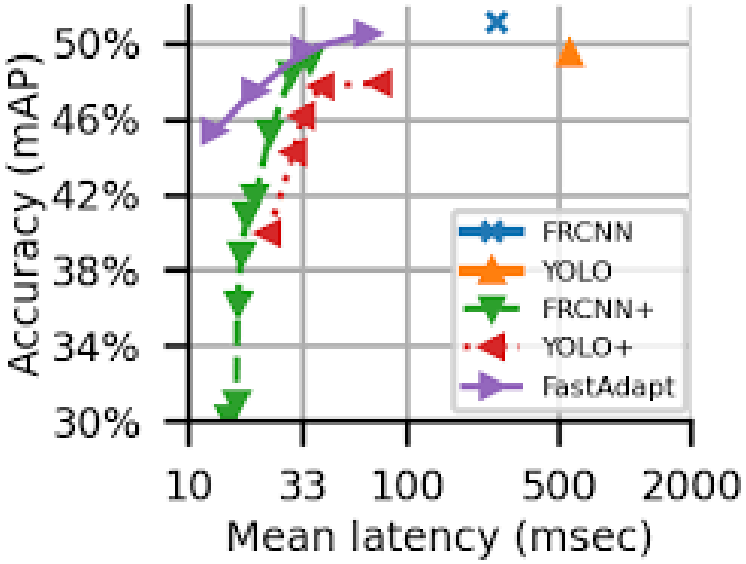


*Figure 3.* Performance benchmark for a number of object detection algorithms

**SSD**

The SSD (Single Shot MultiBox Detector) algorithm, as shown in Figure 4, is based on a single deep neural network with the MultiBox referring to the technique for bounding box regression. Compared to other object detection algorithms SSD operations run at a relatively higher speed. Prior to the development of SSD there have been attempts to come up with faster Object Detectors by inspecting the detection pipeline and modifying every stage of it; much of the attempts while yielded faster results resulted in a tradeoff of accuracy in detection thus, researchers concluded to come up with a new model in SSD rather than modifying any existing model [13].

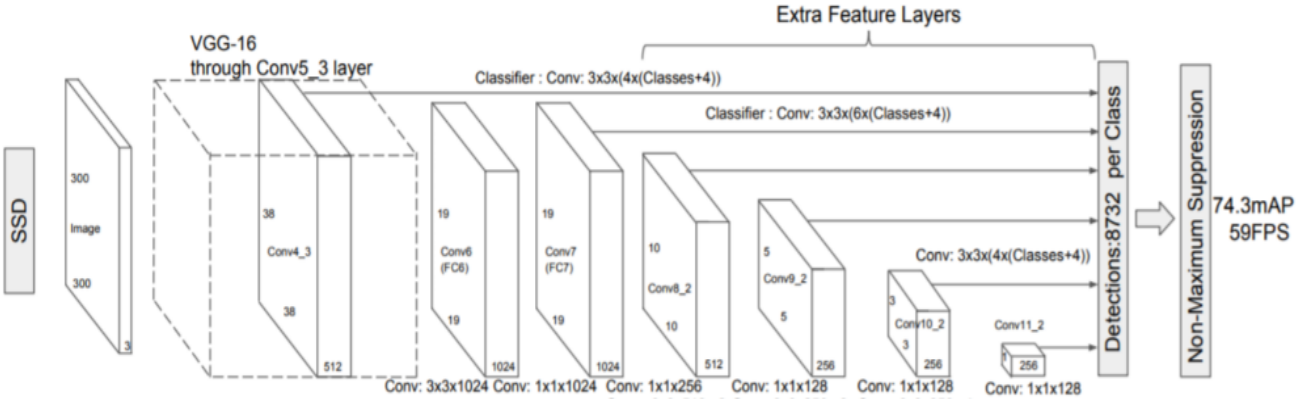


*Figure 4.* The SSD Network depicting the VGG-16 Convolutional layer (s) and Extra Feature Layers

The SSD architecture relies on the creation of bounding boxes and the extraction of feature maps which is also called default bounding boxes. SSD is based on the back-propagation algorithm with the calculated loss value included, this enables SSD to learn the best optimal filter structures to identify the object features accurately. The foundation of SSD relies on a feed-forward complex network that aggregates a collection of standard-size bounding boxes and for each object occurrence in the boxes a score is generated & assigned. An architecture called the VGG-16 network is used for high quality image classification [13].

The use of VGG-16 by SSD for extraction of feature maps yields good performance for the classification of high quality images; the VGG-16 network is depicted in Figures 5. Convolutional filters are used to produce predictions which is a fixed number. In can be inferred that for multiple

feature maps in the network every single feature map cell is associated with a corresponding default bounding box [13].

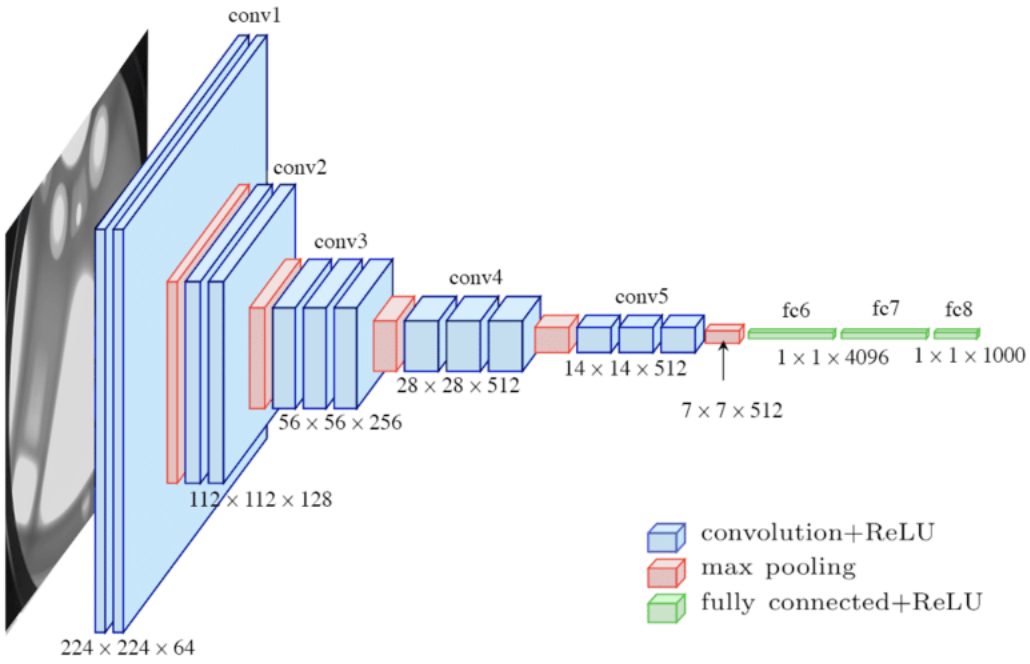


*Figure 5.* The VGG-16 Convolutional network

Created by Szegedy et al. MultiBox is a bounding box regression technique which has a loss function that is composed of two important components, the Confidence Loss and the Location Loss. The Confidence Loss refers to the confidence of the network in computing the bounding box; to compute this loss categorical cross-entropy is utilized. Location Loss refers to the comparison between the ground truth and the network's predicted bounding boxes [13].

Overall, in comparison to other Object Detection algorithms SSD's performance can be faster or better, but it's not to say it comes without room for improvement. For example, SSD requires a large amount of data for the purposes of training; this can be an expensive and time consuming operation. On another note it can be better at detecting smaller objects than other detection algorithms [13]. For a general overview of the Object Detection algorithms see Table 1.

**R-CNN**

The Faster R-CNN (Region Based Convolutional Neural Networks) algorithm, as shown in Figure 6, has names tied to it's predecessors Fast R-CNN and R-CNN but differs in it's technique

where it doesn't use the Selective Search Algorithm for finding the region proposals due to it's slow and time-consuming process which affects the network performance. This is in contrast to R-CNN as shown in Figure 7, for example, where it uses the slower Selective Search Algorithm to retrieve a number of candidate region proposals which are extracted from the input image where then the extracted features are sent to an SVM (Support Vector Machine) for object recognition based on the proposal [13].

Table 1. Object Detection Algorithms Summary

| Object Detection Algorithm | Type | Complexity | Description |
|---|---|---|---|
| YOLO | CNN | **K-Means:** O (n^kd)<br>**k:** number of images<br>**d:** image (s) dimensions | Is based on Darknet and is regression based |
| SSD | Single DNN | Depends on time to train deep-learning models and running inference time | Single Shot Detector and uses VGG-16 network |
| R-CNN | CNN | **RPN:** O (N2/2)<br>**ROI:** O (1) | Composed of Regions of Interest (RoI) and Region Proposal Network (RPN) |

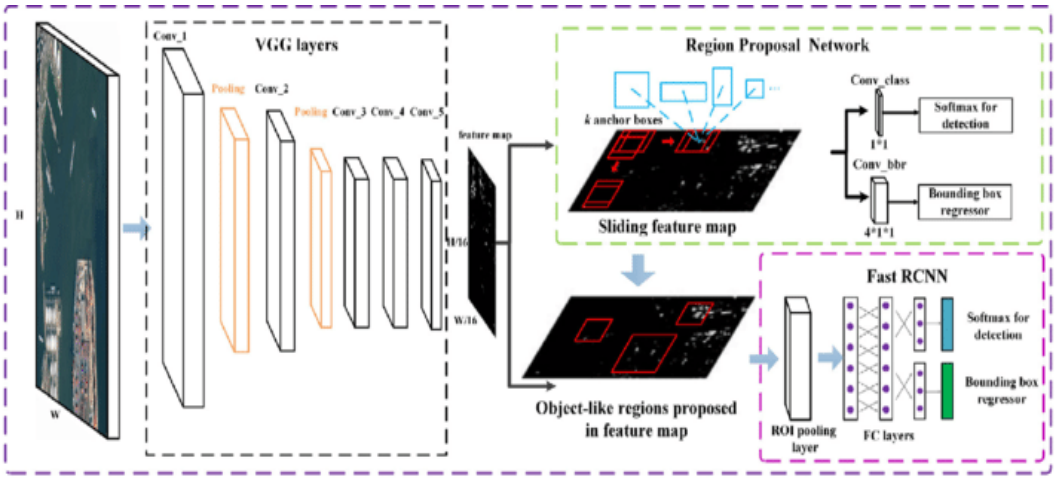


*Figure 6.* The Faster R-CNN Network depicting the VGG layers, Region Proposal Network, etc.

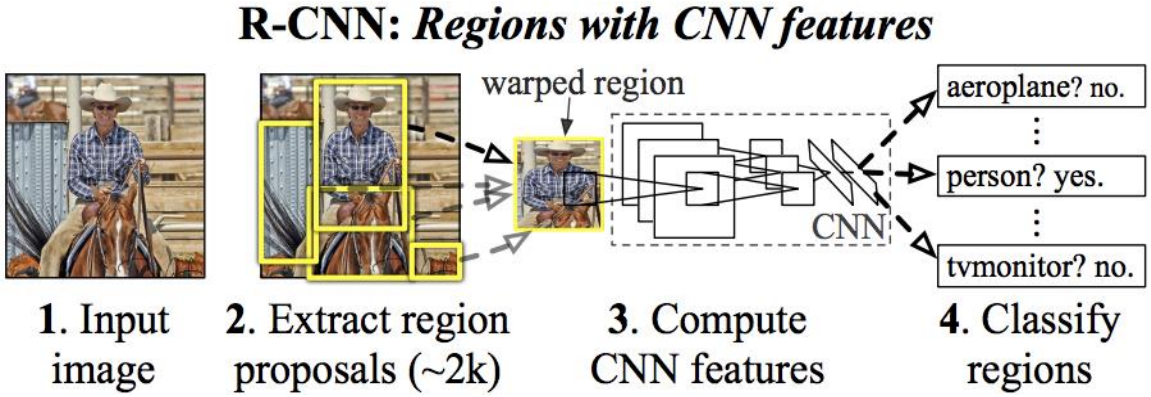


*Figure 7.* The R-CNN process for classifying regions with CNN features

The Selective Search Algorithm has a number of drawbacks, the most notable being the requirement to classify about 2000 region proposals which makes the R-CNN slower and time-consuming. The Fast R-CNN differs in that it doesn't require feeding around 2000 proposals to the Convolutional Neural Network as input but is still plagued by the slow and time-consuming Selective Search algorithm which is considered exhaustive [13].

The Fast R-CNN algorithm makes improvements on R-CNN in the sense that it uses the image itself for generating the convolutional feature map. A feature map is generated only once per image as a convolution operation. In comparison to R-CNN this improves the time for both training and testing as Fast R-CNN doesn't require feeding around 2000 proposals to the network [13].

The Faster R-CNN differs from both R-CNN and Fast R-CNN in that it doesn't use the Selective Search algorithm for determining the region proposals, rather it uses the network to learn the proposals. Faster R-CNN experimented with two convolutional networks architectures in The ZF (Zeiler and Fergus) and VGG-16. The changes to Faster R-CNN is considered significant enough that is now possible for use in real-time object detection [13].

Overall, in comparison to other detection algorithms Faster R-CNN makes improvement over R-CNN and Fast R-CNN though it still slower time complexity wise compared to YOLO where it requires only a single pass unlike multiple passes over a single image for that of Faster R-CNN [13].

# CHAPTER 4

## AVL BINARY SEARCH TREE BACKGROUND

The AVL (Adelson-Velsky and Landis) Self-balancing Binary Search Tree was incepted in 1962 where the difference in height of a node's leftmost and rightmost sub-trees is at most one. Whenever the height exceeds one a rebalance operation is performed on the tree using the balance factor as well to make sure the height remains one; this is to ensure that the running time for all basic operations remains O (logn).

The Node class contains properties of its value, left & right properties for references to their respective nodes, and a property for its height. The Tree class contains operations such as insertion, search, delete, and traversal. For the purposes of this Thesis research project the Insertion and Traversal are primarily used. Other supplementing operations includes the left and right rotation operations, height retrieval, and balance factor retrieval.

The insertion operation, as depicted in Figure 8, is a recursive implementation which inserts the key depending on if the key is greater than or less than the current node, the parent node's height is updated, the balance factor is then computed, and then checks if the height is greater than one or less than zero; if this is the case then it checks four cases, as shown in Figure 9, which are: LeftLeft, RightRight, LeftRight, & RightLeft and performs the left or right rotations as necessary; the root is finally returned.

The traversal operation is done as a preorder traversal (see Figure 10) and the rotations are done using the four cases mentioned previously. In addition to performing the rotation operations, the nodes' height is updated and the new root is returned. The balance factor, as depicted in Figure 11, is calculated by getting the height of the left sub node and the height of the right sub node and subtracting them to get the computed Balance Factor value.

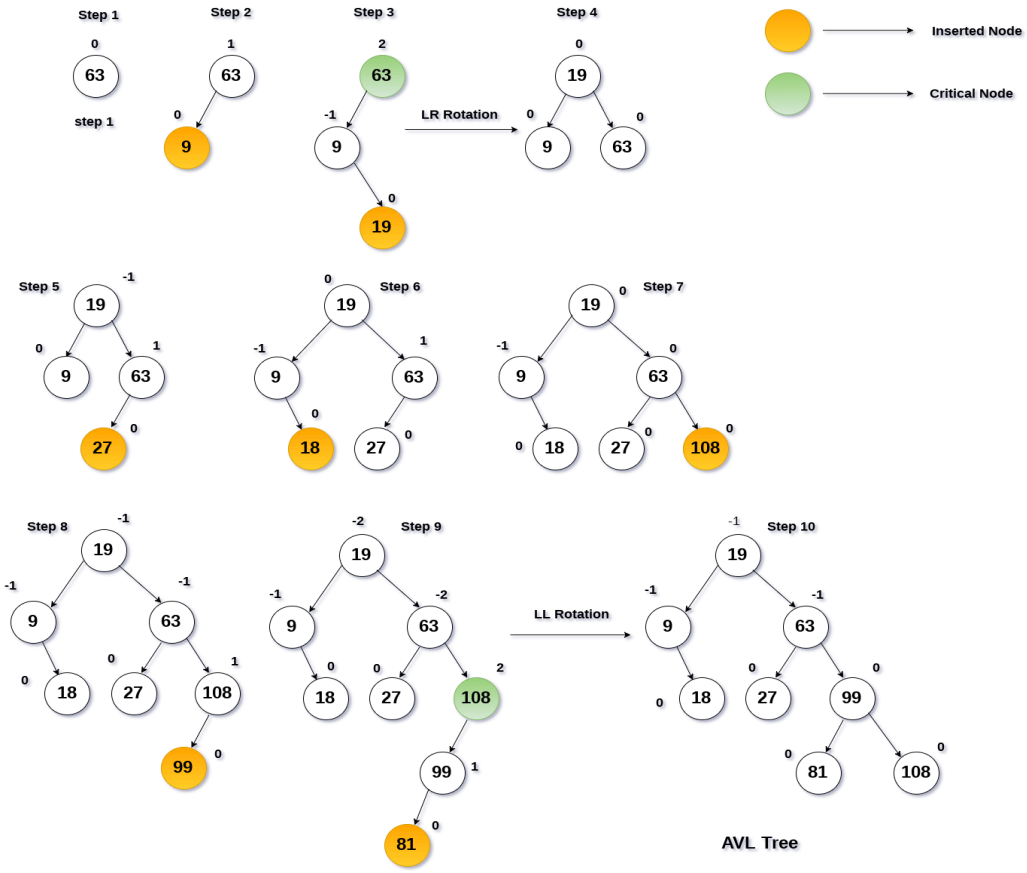


*Figure 8.* The insertion operation being shown on an example AVL tree step-by-step

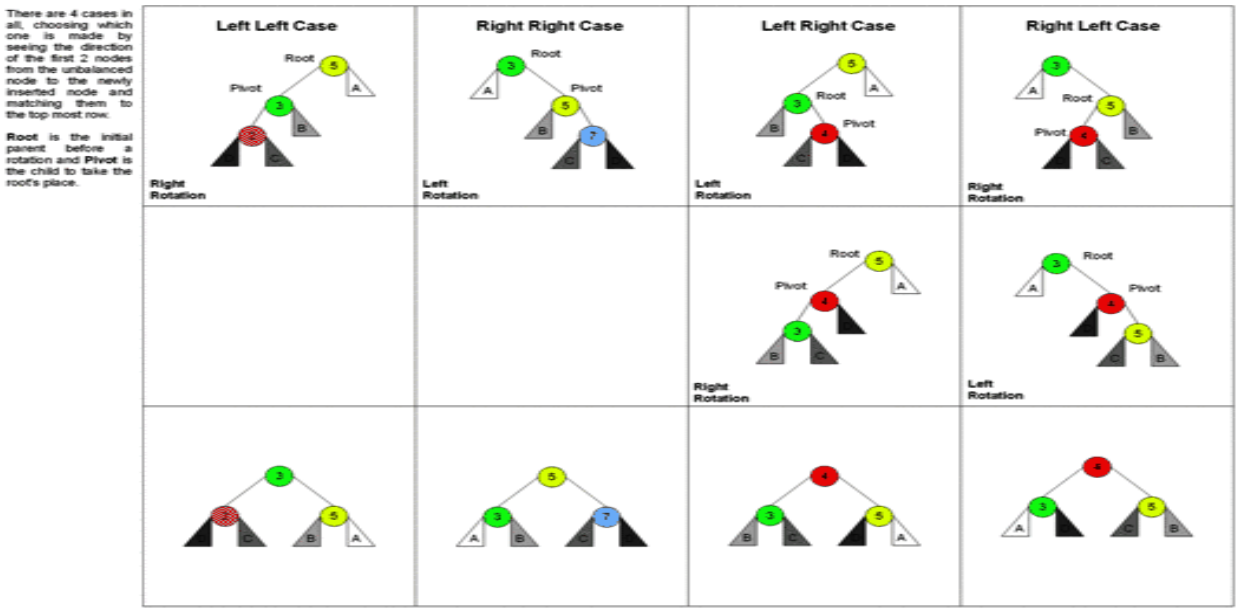


*Figure 9.* The Four Rotation cases as shown above on an example AVL tree

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

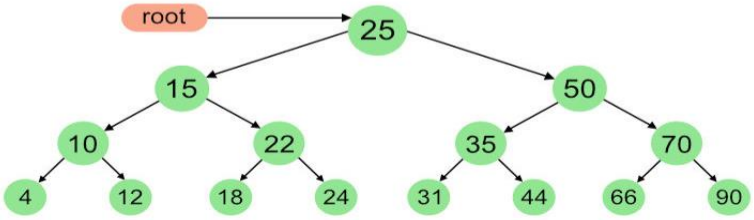


*Figure 10.* The traversal of an AVL tree using multiple traversal methods

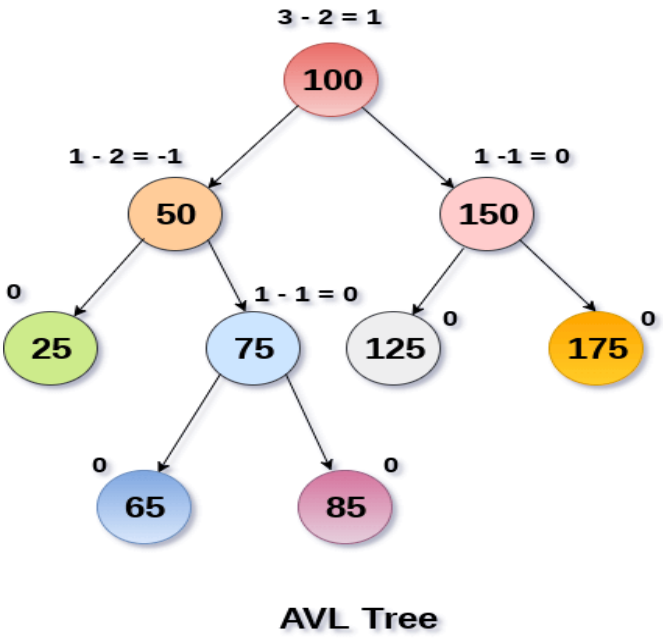


*Figure 11.* The calculation of the Balance Factors on an example AVL tree

Real world application of the AVL Self-balancing Binary Search Tree can be found in Databases where transactions are conducted in searching/retrieval. Similar tree structures include the Red-Black tree which is just as efficient and the Splay Tree which has a simpler implementation than both. The AVL tree is considered stricter when it comes to balancing and rotations; it is fast searching.

# CHAPTER 5

# GPU/CUDA/JETSON NANO BACKGROUND

## GPU/CUDA Background

The GPU has come ways when it comes to computational power [6], [14]-[15]. Companies such as Nvidia and AMD have developed their GPUs to a point where they have supplemented their hardware with computational frameworks to support GPU general purpose computing/processing [7]-[8]; this is also known as GPGPU. Nvidia has developed CUDA (Compute Unified Device Architecture) which enables GPGPU on their proprietary GPU hardware that supports it.

AMD adopted an open platform version of GPU computing called OpenCL. It was originally incepted by Apple in which they proposed to the Khronos Group in 2008 to form the OpenCL 1.0 spec. AMD first supported it in their Radeon HD 4000 graphics card in 2008 which supports OpenCL version 1.0 and 1.1, but not without using their own “Close to Metal (CTM)” implementation first prior to transitioning.

Parallel computation has become possible for mainstream consumers as the results of the high GPU core count and frameworks available by their respective companies. Parallel computation is the processing of data concurrently through the GPU cores to expedite the processing/computation of the data; this is also known as high performance computation.

Parallel processing began earlier through multi-core CPUs but the advancements of GPUs have enabled the GPU to be more mainstream in parallel processing. Early success of parallel processing can be traced back to the massively parallel processors (MPPs) consisting of 64 Intel 8086 & 8087 processors for a scientific computing project at Caltech in the mid 1980s. Prior to multi-core CPUs and the MPPs earlier traces of parallel processing can be further dated back to Supercomputers in the 60s & 70s with shared data being manipulated through a shared main memory source in which multiple processors can access, this is known as shared memory multiprocessors.

## Jetson Nano Background

The Nvidia Jetson Nano is an embedded computer specifically designed for Artificial Intelligence, Machine Learning, and Computer Vision processing as a separate onboard computer for Robotics kits that supports the Jetson Nano. The Nvidia platform provides Jetson Nano a suit of AI softwares & products through their JetPack SDK.

More details of the JetPack SDK as per Nvidia, “JetPack SDK includes the Jetson Linux Driver Package (L4T) with Linux operating system and CUDA-X accelerated libraries and APIs for Deep Learning, Computer Vision, Accelerated Computing and Multimedia. It also includes samples, documentation, and developer tools for both host computer and developer kit, and supports higher level SDKs such as DeepStream for streaming video analytics and Isaac for robotics.”

The Nvidia Jetson Nano supports the TensorRT SDK, as shown in Figure 12, through the onboard Nvidia Maxwell GPU which tremendously speeds up applications that support it by up to 40X compared to CPU-only applications for inference. As described by Nvidia, “TensorRT is built on CUDA®, NVIDIA’s parallel programming model, and enables you to optimize inference leveraging libraries, development tools, and technologies in CUDA-X™ for artificial intelligence, autonomous machines, high-performance computing, and graphics. With new NVIDIA Ampere Architecture GPUs, TensorRT also leverages sparse tensor cores providing an additional performance boost.”

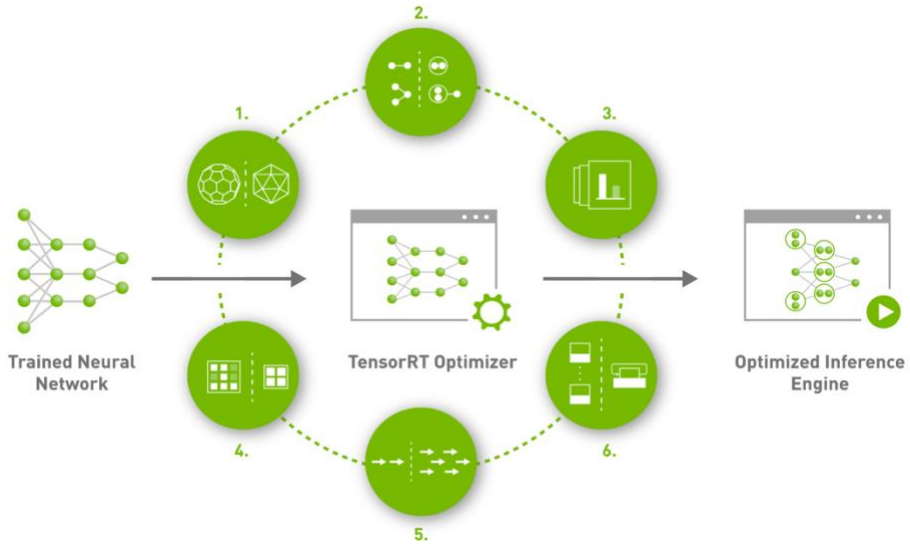


*Figure 12.* Nvidia TensorRT in works for the optimization of the Inference Engine

The Nvidia Jetson Nano also supports workload for IoT computation which collects vast data from a number of sensors and cameras and the vast data needs to be processed for analytical insights. This is supported through Nvidia's DeepStream SDK which as described by Nvidia, "DeepStream enables a broad set of use cases and industries, to unlock the power of NVIDIA GPUs for smart retail and warehouse operations management, parking management, optical inspection and traffic planning and more. The SDK is a core part of Nvidia Metropolis; the application framework for building and deploying intelligent video analytics based applications."

Use cases for the Nvidia Jetson Nano includes computation & parallel processing for Artificial Intelligence tasks & running models for applications such as image classification, object detection, segmentation, and speech processing. It also supports a rich set of GPU-accelerated libraries for enhanced GPU computation. In addition to Robotic kits that support the Jetson Nano there are enclosure cases that can be used to make it a mini portable computer for other AI/ML/CV tasks. Professionals, researchers, and students are encouraged to look into the use of the Nvidia Jetson Nano whether as a project or to learn Artificial Intelligence/Machine Learning/Computer Vision along with Robotics; Nvidia provides an ecosystem to facilitate learning through rich documentation and a helpful community.

# CHAPTER 6

## DESIGN CONCEPT AND PRINCIPLE OF SYSTEM

The components of the system are composed of the Yahboom Jetbot and Nvidia Jetson Nano. The Jetbot is a rover like robot that has a mounted camera on the hardware head that can rotate 180 degrees vertical & horizontally with a vertical lift for the whole camera head up and down. The Jetbot uses ‘caterpillar’ like tracks to move with the ability to move forward, backward, left, right, etc.

The Robot was assembled from the packaging by Yahboom. It contains the metal parts for the body, track motor & tracks, camera unit, and other miscellaneous parts. This required the use of a screwdriver and pliers for securing the screws on the robot. Wired connections are attached to the Jetbot expansion board in which the expansion board is mounted on the Nvidia Jetson Nano embedded computer.

The Nvidia Jetson Nano is a separate onboard/embedded computer used for AI/ML/CV tasks; it uses an open source operating system that can be accessed like any functional computer. It also has input ports such as USB and video output ports such as DisplayPort. As described by Nvidia, “The NVIDIA® Jetson Nano™ Developer Kit delivers the compute performance to run modern AI workloads at unprecedented size, power, and cost.”

The assembled robot, as shown in Figure 13, is a fully functional rover like robot that that is supported with an associated mobile app (Android and IOS) to move it with some basic AI/ML/CV functionalities such as face tracking and following. The robot is connected to a WIFI network along with the mobile device to make use of these functionalities. The Operating System image provided by Yahboom has all the necessary drivers for the basic Robotic functionalities such as movement and camera head rotation.

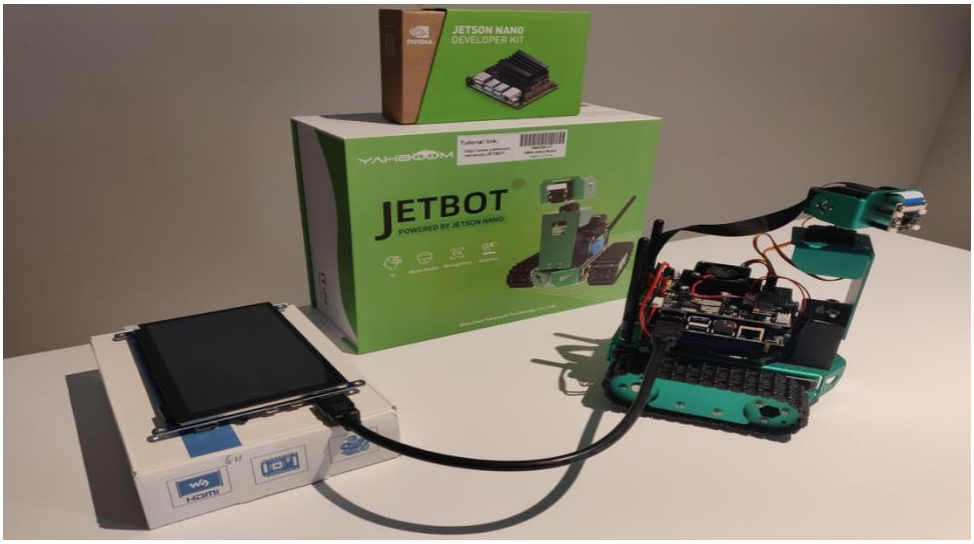


*Figure 13.* The assembled Yahboom Jetbot robot with an attached portable touchscreen

The class files for the Robotic Computer Vision System is composed of the main Computer Vision class, AVL tree class, License Plate class, and Robotics Controls class. All of which makes up the functionalities of the system to detect the car object, assigning the license plate number, and then adding it to the AVL tree. The Robotic Controls class opens a GUI for the movement controls of the robot; the UML class diagram is depicted in Figure 14.

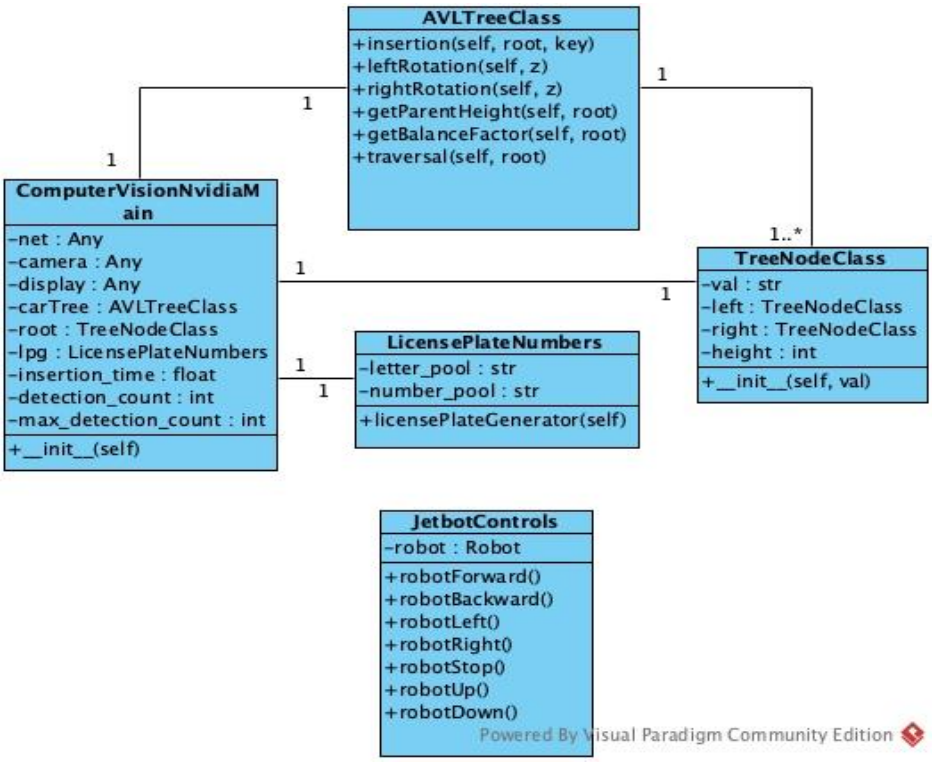


*Figure 14.* The associated Python classes for the Computer Vision Robotics System

# CHAPTER 7

# EXPERIMENTAL SETTINGS AND IMPLEMENTATION

## Hardware and Software

The Thesis research project is composed of a Robotic System which includes the Jetbot Robot and Jetson Nano computer. The Robotic Computer Vision System is composed of a number of softwares on the Jetson Nano itself to run the AI/ML/CV tasks. This includes the Operating System image provided by Yahboom which consists of the Robotic drivers and other miscellaneous softwares. The following softwares were used for the CV tasks:

- Operating System: Ubuntu 18.04 Release
- Object Detection Model: SSD-MobileNetV2
- Languages: Python
- IDE: Visual Studio Code
- Packages: jetson.inference, jetson.utils, jetbot.Robot

The following hardware were used to construct the Robotic Computer Vision system:

- Yahboom Jetbot (see Table 2 for specs)
- Nvidia Jetson Nano (see Table 3 for specs)

Table 2. Yahboom Jetbot specs

| Input | Power Solution | Life Time | Traveling Motor | Motor Drive |
|---|---|---|---|---|
| Custom 800W HD camera | 18650 battery pack (12.6V) | 180 minutes | L-type 370 motor *2 | Single maximum continuous output current 4.0A, Dual maximum continuous output current 8.0A |

Table 3. Nvidia Jetson Nano specs

| GPU | CPU | Operating System | Memory | Camera |
|---|---|---|---|---|
| 128-core NVIDIA Maxwell | Quad-core ARM® A57 | Ubuntu 18.04 LTS (Yahboom image) | 4 GB 64-bit LPDDR4; 25.6 gigabytes/second | MIPI CSI-2 DPHY lanes, 12x (Module) and 1x (Developer Kit) |

The Yahboom Jetbot, as depicted in Figure 15, again is a rover like robot with the associated Jetson Nano computer to power the AI/ML/CV tasks. The Jetson Nano, as depicted in Figure 16, uses a proprietary package/library specifically to work on the Jetson Nano and associated onboard Maxwell GPU; the package/library required a custom installation. The package used is called jetson with the ‘inference’ and ‘utils’ subpackages used for object detection and custom functionalities all of which is based on the Python programming language.

*Figure 15.* The Yahboom Jetbot front-facing

*Figure 16.* The Nvidia Jetson Nano computer used for AI/ML/CV tasks

The inference subpackage provides functionalities for Image Recognition, Object Detection, Segmentation, Pose Estimation, and Monocular Depth as depicted in Table 4; for the purposes of this Thesis research project the Object Detection functionality is used from the inference subpackage. The utils subpackage provides functionalities for the usage of video/camera/GPU hardware as depicted in Table 5.

Table 4. Jetson Inference package functionalities

| Computer Vision Technology/Technique | jetson.inference functions |
|---|---|
| Image Recognition | imageNet (…) |
| Object Detection | detectNet (…) |
| Segmentation | segNet (…) |
| Pose Estimation | poseNet (…) |
| Monocular Depth | depthNet (…) |

Table 5. Jetson Utils package functionalities

| jetson.utils functions | Description |
|---|---|
| videoSource (…) | Interface for cameras, video streams, and images |
| videoOutput (…) | Interface for streaming video and images |

**SSD-MobileNetV2**

The SSD-MobileNetV2 model is used as the object detection model. The SSD (Single Shot MultiBox Detector) algorithm, as referenced in a previous chapter, is based on a single deep neural network with the MultiBox referring to the technique for bounding box regression. The SSD architecture relies on the creation of bounding boxes and the extraction of feature maps which is also called the default bounding boxes.

The MobileNetV2, as shown in Figure 17, was developed by Google and is a mobile architecture which is Neural Network based that uses depthwise separable convolution as efficient building blocks which incepted from MobileNetV1. It improves on the previous V1 version by

differentiating it with new features made to the architecture in adding linear bottlenecks between the layers and shortcutting connections between the bottlenecks.

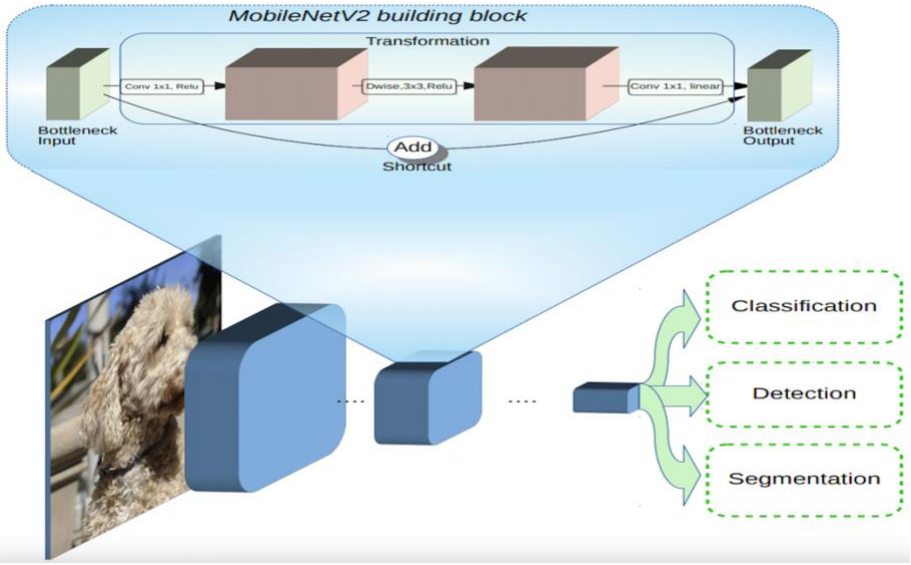


*Figure 17.* The building block of MobileNetV2

The test data collected is based on the Computer Vision System setup with the Jetbot and Jetson Nano. The purpose of this Thesis Research Project is to answer the central question of whether the AVL self-balancing Binary Search Tree can be used for optimization of data processing from a Computer Vision Robotics camera source; this entails a number of systematic steps to collect the data.

The experiment procedures involve multiple Python class files which includes the main Computer Vision class, AVL Tree class, Tree Node class, License Plate Number class, and Robotics Control class. The experiment procedures involve the whole Computer Vision Robotics system running to scan a field of view/video source for car (s) to be detected and tracked in real-time.

The System Pipeline is composed of car (s) being detected from the camera image source where the SSD-MobileNetV2 model completes the detection of car (s) and then returns a list of the detected car (s); a concise snippet of the object detection code is shown in Figure 18. A License Plate Number is generated and assigned to each of the detected car (s) in the list, the Car License Plate Number is then inserted into the AVL tree.

```python
import jetson.inference

net = jetson.inference.detectNet("ssd-mobilenet-v2", threshold=0.3)

while display.IsStreaming() and detection_count < max_detection_count:
    img = camera.Capture()
    detections = net.Detect(img)

    for detection in detections:
        #.. if car detected...
        #.. generate license plate number
        #.. insert into AVL tree

    display.Render(img)
```

*Figure 18.* Concise code snippet of the Object Detection code

The experiment procedures is composed of collecting the detected cars in increment batch sizes of 20K, 50K, 100K, 150K, 200K, and 500k as these are multiple captured cars for all the batch sizes but is interpreted for the purposes of this Thesis Research Project as a simulation to fill the AVL Tree to test its specifications & properties. To start the Computer Vision Robotics system the Computer Vision Main class is executed to bootstrap the system & testing.

The test metrics to be collected are the Pre-Process, Network, Post-Process, Visualize, Total, Car License Plates Insertion time average, and AVL Binary Search Tree traversal time metrics. All of which is utilizing the CUDA platform for the processing of the image inference through the SSD-MobileNetV2 model. The CPU time and CUDA time are taken for the Pre-Process, Network, Post-Process, Visualize, and Total metrics in the millisecond (ms) unit which uses the framework's built-in profiler function. The Car License Plate Insertion time average and AVL Binary Search Tree traversal time metrics are taken in as seconds unit and uses the Python timeit module.

# CHAPTER 8

## EXPERIMENTAL RESULTS AND ANALYSIS

The goal to be achieved from this Thesis research project is to answer the question of whether the AVL Self-Balancing Binary Search Tree can be used for optimization of data processing from a Computer Vision Robotics camera source which involves observing the trends from the project experimental results taking into account the CPU times and CUDA/GPU times for the Computer Vision inference tasks and times for the AVL self-balancing Binary Search Tree operations (insertion and traversal).

From there we can determine if the AVL tree is feasible for optimized data processing. The test metrics to be observed are the Pre-Process, Network, Post-Process, Visualize, Total, Car License Plate Insertion average time, and AVL Binary Search Tree traversal time metrics. The data collected, again, is detected cars in increment batch sizes of 20K, 50K, 100K, 150K, 200K, and 500k where the profile times are collected for the previously mentioned metrics along with the AVL tree operation times. The experimental results are shown for batch sizes 20K (see Tables 6 & 7), 50K (see Tables 8 & 9), 100K (see Tables 10 & 11), 150K (see Tables 12 & 13), 200K (see Tables 14 & 15), and 500k (see Tables 16 & 17).

Table 6. 20k - Nvidia Jetson Nano profile metrics

| | CPU times (ms) | CUDA times (ms) |
|---|---|---|
| Pre-Process | 0.06990 | 1.26344 |
| Network | 57.03245 | 50.63583 |
| Post-Process | 0.05469 | 0.05490 |
| Visualize | 0.33376 | 2.54073 |
| Total | 57.49080 | 54.49490 |

Table 7. 20k - AVL Tree profile metrics

| Car License Plate Insertion time average (sec.) | AVL Binary Search Tree traversal time (sec.) |
|---|---|
| 0.00024025841156597623 | 0.033730052999999316 |

Table 8. 50k - Nvidia Jetson Nano profile metrics

| | CPU times (ms) | CUDA times (ms) |
|---|---|---|
| Pre-Process | 0.22740 | 1.26854 |
| Network | 49.39505 | 43.89255 |
| Post-Process | 0.10974 | 0.10563 |
| Visualize | 1.67525 | 2.55953 |
| Total | 51.40744 | 47.82625 |

Table 9. 50k - AVL Tree profile metrics

| Car License Plate Insertion time average (sec.) | AVL Binary Search Tree traversal time (sec.) |
|---|---|
| 0.000244837573846422 | 0.09324569000000338 |

Table 10. 100k - Nvidia Jetson Nano profile metrics

| | CPU times (ms) | CUDA times (ms) |
|---|---|---|
| Pre-Process | 0.06740 | 1.17786 |
| Network | 59.23953 | 52.70693 |
| Post-Process | 0.07834 | 0.07901 |
| Visualize | 0.41314 | 4.80406 |
| Total | 59.79840 | 58.76786 |

Table 11. 100k - AVL Tree profile metrics

| Car License Plate Insertion time average (sec.) | AVL Binary Search Tree traversal time (sec.) |
|---|---|
| 0.00022577852529993375 | 0.16379203399992548 |

Table 12. 150k - Nvidia Jetson Nano profile metrics

| | CPU times (ms) | CUDA times (ms) |
|---|---|---|
| Pre-Process | 0.12927 | 1.16614 |
| Network | 61.68370 | 54.42234 |
| Post-Process | 0.17396 | 0.17240 |
| Visualize | 0.67580 | 6.92302 |
| Total | 62.66273 | 62.68391 |

Table 13. 150k - AVL Tree profile metrics

| Car License Plate Insertion time average (sec.) | AVL Binary Search Tree traversal time (sec.) |
|---|---|
| 0.00023977467304605227 | 0.24423291699986294 |

Table 14. 200k - Nvidia Jetson Nano profile metrics

| | CPU times (ms) | CUDA times (ms) |
|---|---|---|
| Pre-Process | 0.16172 | 1.17432 |
| Network | 61.31961 | 55.52911 |
| Post-Process | 0.05766 | 0.05719 |
| Visualize | 0.38339 | 1.85698 |
| Total | 61.92238 | 58.61760 |

Table 15. 200k - AVL Tree profile metrics

| Car License Plate Insertion time average (sec.) | AVL Binary Search Tree traversal time (sec.) |
|---|---|
| 0.00023508169364162124 | 0.34366351300013775 |

Table 16. 500k - Nvidia Jetson Nano profile metrics

| | CPU times (ms) | CUDA times (ms) |
|---|---|---|
| Pre-Process | 0.07636 | 1.74385 |
| Network | 54.91892 | 47.69875 |
| Post-Process | 0.03271 | 0.03255 |
| Visualize | 0.18120 | 0.31755 |
| Total | 55.20920 | 49.79271 |

Table 17. 500k - AVL Tree profile metrics

| Car License Plate Insertion time average (sec.) | AVL Binary Search Tree traversal time (sec.) |
|---|---|
| 0.00024253449937982157 | 0.7993257340003765 |

Based on the above experimental test results we can observe that the Car License Plate insertion average time into the AVL self-balancing Binary Search Tree remains relatively stable for the most part. The AVL tree traversal time on the other hand shows a relatively slow growth over the varying batch data sizes, but overall remains relatively stable as the time is under a whole unit number for all batch data sizes. Therefore, Insertion and Traversal complexity time is O (logn) where any potential concerns could point to the memory space.

Observing the Jetson Nano object detection inference profile metrics results we can see difference in the CPU and CUDA times for all batch data sizes where the CUDA/GPU processing yields lower times compared to the CPU for the most part. Looking at some of the individual inference tasks we can notice for example the Post-Process times remaining stable across the varying batch data sizes for the CPU and CUDA/GPU; looking at the Network times across the varying batch data sizes we can see that in general the CUDA times are lower than the CPU times. Finally, we can also see that the Visualize times for the CPU is lower across the varying batch data sizes compared to the CUDA times.

# CHAPTER 9

# CONCLUSIONS AND FUTURE WORK

In conclusion, looking at the experiment data results we can conclude that the AVL tree Insertion and Traversal operations remains consistent for the most part with running time of worse-case complexity O (logn) as the Car License Plate insertion time average remains relatively stable for the most part and the AVL tree traversal time on the other hand shows a relatively slow growth over the varying batch data sizes.

This makes the AVL Self-Balancing Binary Search Tree feasible for Computer Vision post processing tasks along with other industry applications where post-processing of data is a business requirement which includes the processing & storage of data in a data structure medium. This helps facilitate the flow of data in an application and helps reduce any potential bottlenecks; thus, increasing overall performance of the application.

This answers the central question of: “whether the AVL self-balancing Binary Search Tree can be used for optimization of data processing from a Computer Vision Robotics camera source?" This is further proves the research conducted by Bell et al. [3] that AVL Trees perform good In general. In Tripathi’s research [4] for the proposed usage of Virtual Nodes in AVL trees it could be worth looking into the parallelization of the Virtual Nodes using GPU cores as GPU processing yields good performance [6], [14]-[15].

Possible future work could include improvements to the Neural Networks of the detection models. Also, the area of Nvidia CUDA parallel processing could be further improved as the Nvidia GPU hardware improves and is not necessarily dependent on each other as they can be improved independently as well; an example of a CUDA improvement is the addition of Nvidia TensorRT which, as mentioned in a previous chapter, involves improving the Computer Vision inference performance through optimizations using an Nvidia supported GPU. Further research could be looking into the parallelization of a height balanced tree on a GPU.

# APPENDIX

## MISCELLANEOUS

### Equations

$$\sigma\,(x) = 1/(1 + e^{-x}) \tag{2}$$

$$\begin{aligned} b_x &= \sigma(t_x) + c_x \\ b_y &= \sigma\big(t_y\big) + c_y \\ b_w &= p_w e^{t_w} \\ b_h &= p_h e^{t_h} \end{aligned} \tag{3}$$

### Tables

Table 18. Python Programming Language description

| | Python |
|---|---|
| Type: | High level, simpler more concise syntax makes it easier to understand |
| Productivity: | Less verbose, fewer lines. More productive than other programming languages |
| Speed: | Python is slower compared to other programming languages; it is an interpreted language |
| Best for: | AI, ML, Data Science |

### Pictures


*Figure 19.* The assembled standalone Yahboom Jetbot robot

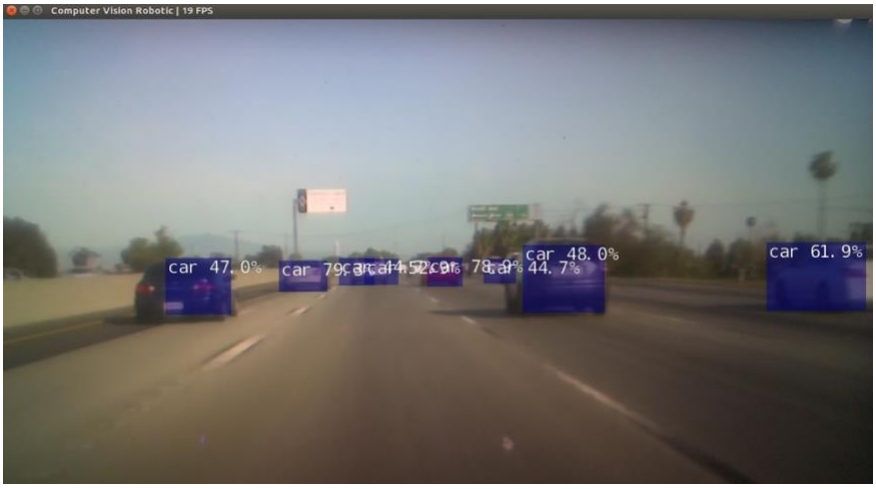


*Figure 20.* The object detection program running on the assembled Yahboom Jetbot robot

## REFERENCES


[1] J. Qiu, L. Wang, Y. H. Hu, and Y. Wang, “Two motion models for improving video object tracking performance,” *Comput. Vis. Image Underst.*, vol. 195, no. 102951, p. 102951, 2020.

[2] B. Karasulu and S. Korukoglu, “Moving object detection and tracking by using annealed background subtraction method in videos: Performance optimization,” *Expert Syst. Appl.*, vol. 39, no. 1, pp. 33–43, 2012.

[3] J. Bell and G. Gupta, “An evaluation of self-adjusting binary search tree techniques,” *Softw. Pract. Exp.*, vol. 23, no. 4, pp. 369–382, 1993.

[4] R. R. K. Tripathi, “Balancing of AVL tree using virtual node,” *Int. J. Comput. Appl.*, vol. 7, no. 14, pp. 12–15, 2010.

[5] Y. Zhang and Z. Xu, “Concurrent manipulation of expanded AVL trees,” *J. Comput. Sci. Technol.*, vol. 13, no. 4, pp. 325–336, 1998.

[6] E. Cervera, “GPU-accelerated vision for robots: Improving system throughput using OpenCV and CUDA,” *IEEE Robot. Autom. Mag.*, vol. 27, no. 2, pp. 151–158, 2020.

[7] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, “Computer vision accelerators for mobile systems based on OpenCL GPGPU co-processing,” *J. Signal Process. Syst.*, vol. 76, no. 3, pp. 283–299, 2014.

[8] P. Czarnul, “Investigation of parallel data processing using hybrid high performance CPU + GPU systems and CUDA streams,” *Comput. Inform.*, vol. 39, no. 3, pp. 510–536, 2020.

[9] Khoirudin and J. Shun-Liang, “GPU Application in Cuda Memory,” *Adv. Comput. Int. J.*, vol. 6, no. 2, pp. 01–10, 2015.

[10] D. Kirk, “NVIDIA cuda software and gpu parallel computing architecture,” in *Proceedings of the 6th international symposium on Memory management - ISMM ’07*, 2007.

[11] R. S. Dehal, C. Munjal, A. A. Ansari, and A. S. Kushwaha, “GPU Computing Revolution: CUDA,” in *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, 2018, pp. 197–201.

[12] L. Haji, R. Zebari, S. R. M. Zeebaree, W. M. Abduallah, H. Shukur, and O. Ahmed, Eds., *GPUs Impact on Parallel Shared Memory Systems Performance*, vol. 24, no. 08. nternational Journal of Psychosocial Rehabilitation, 2020.

[13] S. Srivastava, A. V. Divekar, C. Anilkumar, I. Naik, V. Kulkarni, and V. Pattabiraman, “Comparative analysis of deep learning image detection algorithms,” *J. Big Data*, vol. 8, no. 1, 2021.

[14] K. Y. Erofeev, E. M. Khramchenkov, and E. V. Biryal’tsev, “High-performance processing of covariance matrices using GPU computations,” *Lobachevskii J. Math.*, vol. 40, no. 5, pp. 547–554, 2019.

[15] F. Li, Y. Ye, Z. Tian, and X. Zhang, “CPU versus GPU: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms,” *Neural Comput. Appl.*, vol. 31, no. 8, pp. 4353–4365, 2019.