Design & Development of an Image Processing System Implementing Image Classification, Object Detection, and Image Segmentation in an Unmanned Ground Vehicle for Search and Rescue Applications in Forested Areas

A Project
Presented to the Faculty
of Oregon Institute of Technology
in Partial Fulfillment for Requirements of the Degree
Master of Science
in
Engineering
—
Embedded Systems Engineering & Interconnected Devices

by
Ian Draney
Spring 2020

This is dedicated to my father, Vance Draney, who taught me the values of good work ethic and attitude.

ACKNOWLEDGMENTS

This thesis would not have been possible without all the people I have worked with throughout my life. These relationships developed the path I've traveled thus far. For better or for worse, you have my sincere gratitude.

I would like to thank my advisor, Professor Allan Douglas. My enrollment in his courses, starting from my undergraduate work at Oregon Tech, changed the trajectory of my academic career. His teaching style allowed me to explore electrical engineering beyond that of mere coursework completion to obtain a degree. He inspired me to dive deeper into the subject matter and to explore creative solutions to engineering problems. After completing my undergraduate studies, my pursuit toward a master's degree would have never started without his guidance.

I would like to thank Dr. Mateo Aboy. His contributions to Oregon Institute of Technology are immeasurable. The Portland-Metro (i.e., Wilsonville) campus would not have become the essential satellite campus it is today without him. In my opinion, the Portland-Metro campus is Oregon Tech's *main* campus. Although he currently resides thousands of miles from the the state of Oregon, he continues to develop outstanding academic programs for this institute.

As the final, and the most important acknowledgment of appreciation, I would like to thank my family. All of you have supported my return to academia, even as the envisioned outcome changed throughout the journey. For your endless support, I am extremely grateful.

Ian Draney

June 16, 2021

Abstract

This thesis presents the design and development of an image processing system with implementations of image classification, object detection, and image segmentation in an unmanned ground vehicle for search and rescue applications in forested areas. The problem of the design, development, and implementation of the system is defined. Search and rescue operation significance in the United States provides motivation for the engineering of this system to increase rescue efficiencies. Background and literature review of systems incorporating technological elements required by the unmanned ground vehicle are reviewed. The architecture and specifications of the system are outlined, followed by its design and development, and an assessment of the system's performance results. Finally, this thesis draws conclusions on the system's contributions and future work required to increase the operational efficacy and efficiency of the autonomous search and rescue unmanned ground vehicle.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

2D	Two Dimensional
3D	Three Dimensional
AGV	Autonomous Ground Vehicle
API	Application Programming Interface
bps	Bits per Second
CNN	Convolutional Neural Network
CPU	Central Processing Unit
COCO	Common Objects in Context
CUDA	Compute Unified Device Architecture
cuDNN	CUDA Deep Neural Network library
CV	Computer Vision
DL	Deep Learning
DNN	Deep Neural Network
DSP	Digital Signal Processing
DSSD	Deconvolutional Single Shot Detector
FCN	Fully Convolutional Network
fps	Frames per Second
Gbps	Gigabits per Second
GPS	Global Positioning System
GPU	Graphics Processing Unit
GS	Ground Station
IP	Intellectual Property
IP (2)	Internet Protocol
IMU	Inertial Measurement Unit
L4T	Linux for Tegra
LED	Light Emitting Diode
LIDAR	Light Detection and Ranging
mAP	Mean Average Precision

ML	Machine Learning
NC-AGV	Network-Centric Autonomous Ground Vehicle
NEMA	National Electrical Manufacturers Association
OS	Operating System
PASCAL	Pattern Analysis, Statistical Modeling, and Computational Learning
PCIe	Peripheral Component Interconnect Express
PETG	Polyethylene Terephthalate Glycol
PID	Proportional Integral Derivative
R-FCN	Region-based Fully Convolutional Networks
RADAR	Radio Detection and Ranging
RGB	Red Green Blue
ROI	Region of Interest
ROS	Robot Operating System
RP-SMA	Reverse Polarity SubMiniature Version A
RTP	Real-time Transport Protocol
SAR	Search and Rescue
SAR-UGV	Search and Rescue Autonomous Ground Vehicle
SATA	Serial Advanced Technology Attachment
SLAM	Simultaneous Location and Mapping
SoS	System of Systems
SSD	Single Shot MultiBox Detector
SSD (2)	Solid-State Drive
TGV	Teleoperated Ground Vehicle
SVM	Support Vector Machine
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
V4L2	Video for Linux
VOC	Visual Object Classes
YOLO	You Only Look Once

CHAPTER 1 Introduction

1.1 Introduction & Background

For years, UGV technology has been developed for military, scientific, and civilian applications. With an increasing level of autonomy, UGVs and autonomous vehicles, in general, have been able to better perform tasks that are either too tedious or dangerous for humans to do themselves. While standards for robot and unmanned vehicle autonomy vary, the Committee on Army Unmanned Ground Vehicle Technology has categorized UGVs by four distinct levels of autonomy, ranging from TGVs in which all operation is controlled remotely by a human, to NC-AGVs that have levels of autonomy sufficient to operate as independent nodes in a network-centric warfare model [1]. While the network-centric warfare model does not particularly apply to more peaceful endeavors, independent node operations do relate directly to full vehicle autonomy.

A fully autonomous UGV, or a fleet of fully autonomous UGVs, may also be capable of assisting humans by performing tasks in SAR applications. Many SAR missions take place in forested areas with unmapped trails. By winning the 2005 DARPA Grand Challenge, Montemerlo et al. showed that the use of multiple mapping modules could build a 2D environment based on LIDAR, camera, and RADAR systems [2]. More recent improvements in SLAM algorithms show that mapping modules can be created cheaply using a single monocular camera and a low-cost IMU [3]. Such maps may be actively created to show traversed regions during an SAR operation employing autonomous UGVs.

In order to properly map its mission, the UGV must be able to visually sense a travelable area. This is performed with the assistance of LIDAR or RADAR systems. The primary sensor required for this task, however, is a monocular camera. Adhikari et al. demonstrated that camera input into a system using a DNN and dynamic programming can accurately detect and follow trails in highly unstructured natural environments [4]. It is a goal of this project to use similar methods to develop an object detection system for targets such as lost persons with the use of a DL or a more generalized ML system.

1.2 Problem Definition

The problem that is solved by this project is the design, development, and implementation of an image processing (i.e., computer vision, machine vision, CV, or simply *vision*) system for trail navigation and target object detection for autonomous UGVs capable of adhering to trails with obstacle avoidance and identifying lost persons in a forest environment. The image processing system is an *embedded system* within a fully autonomous UGV that is an SoS. The UGV incorporates additional embedded systems outside of the primary focus of this thesis. These embedded systems include:

- Vehicle mobility (speed, steering, and brake control)
- Short-range wireless communications
- Audio input & DSP with voice recognition
- Long-distance radio communications
- LIDAR
- Power distribution
- Mechanical specification & modification
- Other mission-critical sensors (IMU, accelerometers, temperature, humidity, etc.)

1.3 Significance

Between 2003 and 2006, there were 12,337 SAR operations in United States national parks. These SAR operations ended with 522 fatalities, 4,860 park visitors who were ill or injured, and 2,855 saves. The total cost of these SAR operations was over $16.5 million, with personnel costs accounting for $8.1 million, and vessel costs accounting for $6.9 million [5]. These statistics beg that both rescue results and financial costs of SAR operations can be greatly improved upon by deployment fully autonomous UGVs.

The expectations of this project are that it contributes to the increase of efficiencies of both rescue rates and financial costs in SAR operations. Specifically regarding the image processing system of the project, its motivation is to design, develop, and implement a single vision system/subsystem capable of trail recognition & navigation and target object detection. When applied to SAR operations, the image processing system is capable of navigating trails and identifying rescue targets without human assistance. In other words, these systems enable fully autonomous SAR-UGV operation.

1.4 R&D Objectives and Thesis Contributions

The objectives of this project are to design, develop, and implement an image processing system for SAR-UGV applications in forested areas. With context to the overall objective of the SAR-UGV, as stated in the preliminary requirements by the project advisor, Professor Allan Douglas, the SAR-UGV consists of the following software framework and embedded subsystem components:

- *Software framework.* System software is implemented using ROS. This reduces the workload required for software integration, provides standards for multi-computer communication, and allows students to work independently until the subsystem integration phase. Each subsystem has a computer (e.g., Raspberry Pi or Nvidia TX2 GPU), for

command, control, data collection, and communication. Each computer implements ROS nodes by using a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a *topic*. Nodes can also provide or use a *service*. This framework follows the classical producer-consumer software model.

- *Robot mobility.* The SAR-UGV has variable speed, steering, and braking controls. Speed control is implemented with a PID control loop and uses quadrature encoders for drive axle speed measurement. The ROS node must subscribe to topics (i.e., accepts commands) published by the wireless control or autonomous control subsystems. This subsystem is also responsible for powering down motors in the event of an emergency (e.g., vehicle roll, communications failure, etc.).

- *Wireless control system.* The SAR-UGV is capable of wireless control via Bluetooth or other short-range wireless standards. This facility is required for local robot positioning, correcting vehicle course in the event of errors in the autonomous control subsystem, and for testing the complete system.

- *Video processing & autonomous control systems.* The SAR-UGV navigates the forested terrain (e.g., hiking trails) without human interaction. It is capable of independent navigation to determine the optimal course on the trail and perform obstacle avoidance. The ROS node controlling this subsystem uses data provided by the GPS receiver, accelerometer, LIDAR, and live camera video feeds. The SAR-UGV is also capable of returning to the GS upon command. This is achieved by collecting GPS waypoints during the mission.

- *Voice recognition system.* The SAR-UGV stops periodically, activates an audible rescue beacon, and "listens" for a human call for help. Listening for a call for help is performed using on-board microphones, digital signal processing (i.e., filtering), audio recording, and a trained neural network. Each method used for signal detection generates a confidence factor based on received audio signals. In the event that the confidence factor exceeds a specified threshold, the SAR-UGV transmits the recorded audio file

and GPS location to the GS, and awaits further instructions.

- *Radio communications system.* The SAR-UGV requires an on-board radio for communications with the GS. System status and audio data is transmitted periodically when received on other nodes on the ROS network. Remote commands received from the GS are published so that other nodes may respond appropriately. If a radio with sufficient range and bandwidth can be identified, video will be transmitted to the GS. A compatible radio system is also required at the GS. Together, the two radios provide standard Ethernet communications between the SAR-UGV and the GS.

- *LIDAR system.* The SAR-UGV uses an onboard LIDAR system to map the trail and to assist with obstacle avoidance. Map data is provided to the autonomous control subsystem and periodically transmitted to the GS for vehicle position monitoring.

- *Sensor array & peripheral control system.* The SAR-UGV requires an array of sensors and peripherals for the mission. The required sensors are a GPS receiver, an accelerometer for vehicle roll and inclination measurement, a temperature & humidity sensor, and individual moisture sensors for each of the subsystems for detection of intrusion by rain. Sensor data is published on the ROS network for all other nodes. The required actuators include the on-board LED lighting and rescue beacon.

- *Power system.* The SAR-UGV requires battery power and power distribution/protection circuitry for each subsystem. The system monitors supply voltage, current, and temperature. It also estimates remaining battery capacity. Power system status is periodically reported for transmission over the radio to the GS using ROS.

- *Mechanical system.* The SAR-UGV requires mechanical modification to mount subsystems, mount motors & batteries, and to mechanize the braking system.

- *Ground Station (GS) system.* The GS system consists of a laptop running ROS nodes for data logging and communication to the SAR-UGV via Ethernet and the radio system. It displays GPS waypoints and the LIDAR map during the mission. The

GS is also capable of providing Unix Terminal commands for manual control of the SAR-UGV.

1.5 Thesis Outline

This thesis is structured in six chapters:

- Chapter 1 introduces and defines the problem of design, development, and implementation of the SAR-UGV. The significance of the problem is described. Research & development objectives and overall contributions of the SAR-UGV project are listed. Lastly, the structure of the thesis itself is outlined.

- Chapter 2 provides background information for image processing systems and autonomous control systems implemented for UAVs and UGVs. United States patents relevant to UGVs are discussed, in which some inventions are intended specifically for SAR operations. This chapter also contains a literature review that highlights hardware and software configurations of fully autonomous UGVs, SAR operations, and SAR-UGVs.

- Chapter 3 describes the system architecture and design specifications of the SAR-UGV image processing system. System level design of the SAR-UGV is presented in block diagrams. ROS network testing is introduced with the *turtlesim* tutorial.

- Chapter 4 details the design and development of the SAR-UGV image processing system in both hardware and software. Implementation of the system hardware and software is discussed. A bill of materials required for the project is presented.

- Chapter 5 reviews simulation and experimental results of the image processing system specific to image classification, object detection, and image segmentation. Software performance characterization results are also provided.

- Chapter 6 summarizes this thesis by emphasizing its methodological, technical, and scientific contributions. It also presents future work for the image processing system of the SAR-UGV.

CHAPTER 2 Background & Literature Review

2.1 Related Work

The image processing system of the SAR-UGV has multiple problems to solve during SAR-UGV missions: 1) identify trails that it is capable of traversing, 2) identify objects that could be classified as obstacles, and 3) identify the mission target. Objects encountered on the trails may be common, such as hikers, flora, or fauna. Other existing objects may be uncommon, such as industrial refuse or machinery, given the environment. The SAR-UGV autonomous control system must be capable of responding to the input provided by the image processing system in order to perform navigational tasks such as maintaining itself on the hiking trail, avoiding objects identified as obstacles, and returning to the GS upon mission completion.

2.1.1 Image Processing Systems

Before the SAR-UGV can perform autonomous control functions, it must first understand its environment. The image processing system is necessary for the robot to obtain a visualization of the world in which to navigate. Image processing is inherently the analysis of a single or multiple still images. The SAR-UGV is required to perform image analysis in real-time. This may be executed as three separate tasks: image classification, object detection, and image segmentation. These tasks must be performed successfully in order for the SAR-UGV to properly identify the trail as a path on which it is capable of traveling, detect objects on the trail that it must avoid, and find mission targets.

Image Classification

During operation, the SAR-UGV will constantly be acquiring image data from the image processing system. This image data needs to be classified to ensure that the robot is in the proper locale. In *Image Processing and GIS for Remote Sensing*, Liu et al. provide the following definition for image classification:

> Image classification belongs to a very active field in computing research, *pattern recognition*. Image pixels can either be classified by their multi-variable statistical properties, such as the case of multi-spectral classification, or by segmentation based on both statistics and spatial relationships with neighbouring pixels [6].

The authors argue that image classification can be separated into multiple different approaches:

- Supervised classification
- Unsupervised classification
- Hybrid classification
- Single pass classification
- Iterative classification
- Image scanning classification
- Feature space partition

With these approaches, the authors continue that supervised classification often implements the single pass classification method, while unsupervised classification (i.e., "clustering") implements the iterative classification. Since the SAR-UGV captures a large amount of image data with a great deal of detail, feature space partitioning needs to be implemented to accommodate the required high level of classifier sophistication. Processing images containing multiple simultaneous features, or higher level data dimensionality and quantization,

requires a significant amount of computer memory resources or dynamic resource management. A graphical representation of the relationship between a feature space point in a multi-spectral image (e.g., RGB color space) is displayed below in Figure 2.1.

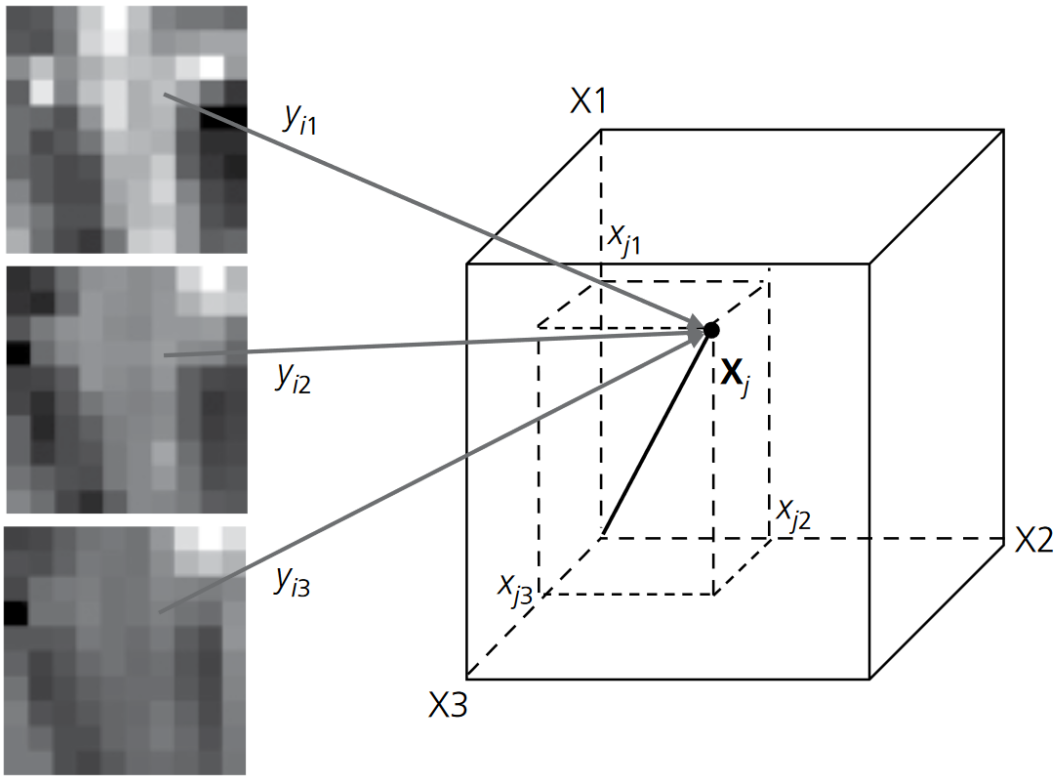


Figure 2.1: A 3D illustration of the relationship between a feature space point X_j and a multi-spectral image pixel $Y_i = (y_{i1}, y_{i3}, y_{i3})$ [6].

In their work, "Image Classification using Random Forests and Ferns", Bosch et al. create a three-step recipe for object detection: 1) shape and appearance representations that support spatial pyramid matching over an ROI, 2) automatic selection of ROIs in image data training, and 3) the use of random forests and random ferns as a multi-way classifier [7]. The terms "random forests" and "random ferns" are not to be confused with the intended environment application for the SAR-UGV of forested areas. These terms, as used by Bosch et al., refer to two different kinds of image classifiers, and are unrelated to any particular environment or its natural flora.

The definition of "random forests" used by Bosch et al. can be cited back to the work of Breiman, where he defines the term as "a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest [8]." Each decision tree acts as an *if/else* statement, where the forest is a combination of decision trees. Hence, in the works of both Bosch et al. and Breiman, the forests are randomized for image classification. By contrast, where trees are hierarchical, Bosch et al. state that "ferns" are "non-hierarchical structures where each consists of a set of binary tests [7]."

The results of Bosch et al. in [7] show that using random forests and ferns reduced the computational expense of image training and testing compared to that of a multi-way SVM. They also state that implementation of ROI detection and sliding windows created improvements, and that generating extra data during training increased image classification performance.

The cost function formula used by Bosch et al. is shown in 2.1, where a known ROI of an image, r_i, of image i, and a subset s of other images j have corresponding object instances in the training image set for the same class. Corresponding ROIs, r_j, of images j were then determined, where $D(r_i)$ and $D(r_j)$ are the ROI descriptors for r_i and r_j, respectively.

$$\mathcal{L}_i = \max_{\{r_j\}} \sum_{j=1}^{s} K(D(r_i), D(r_j)) \tag{2.1}$$

Figure 2.2 shows the automatic ROI detection results from Bosch et al. The images used were from the Caltech-256 object category dataset. The subset $s = 3$ was used for the image categories of cactus, bathtub, watermelon, camel, and windmill within the Caltech-256 dataset.

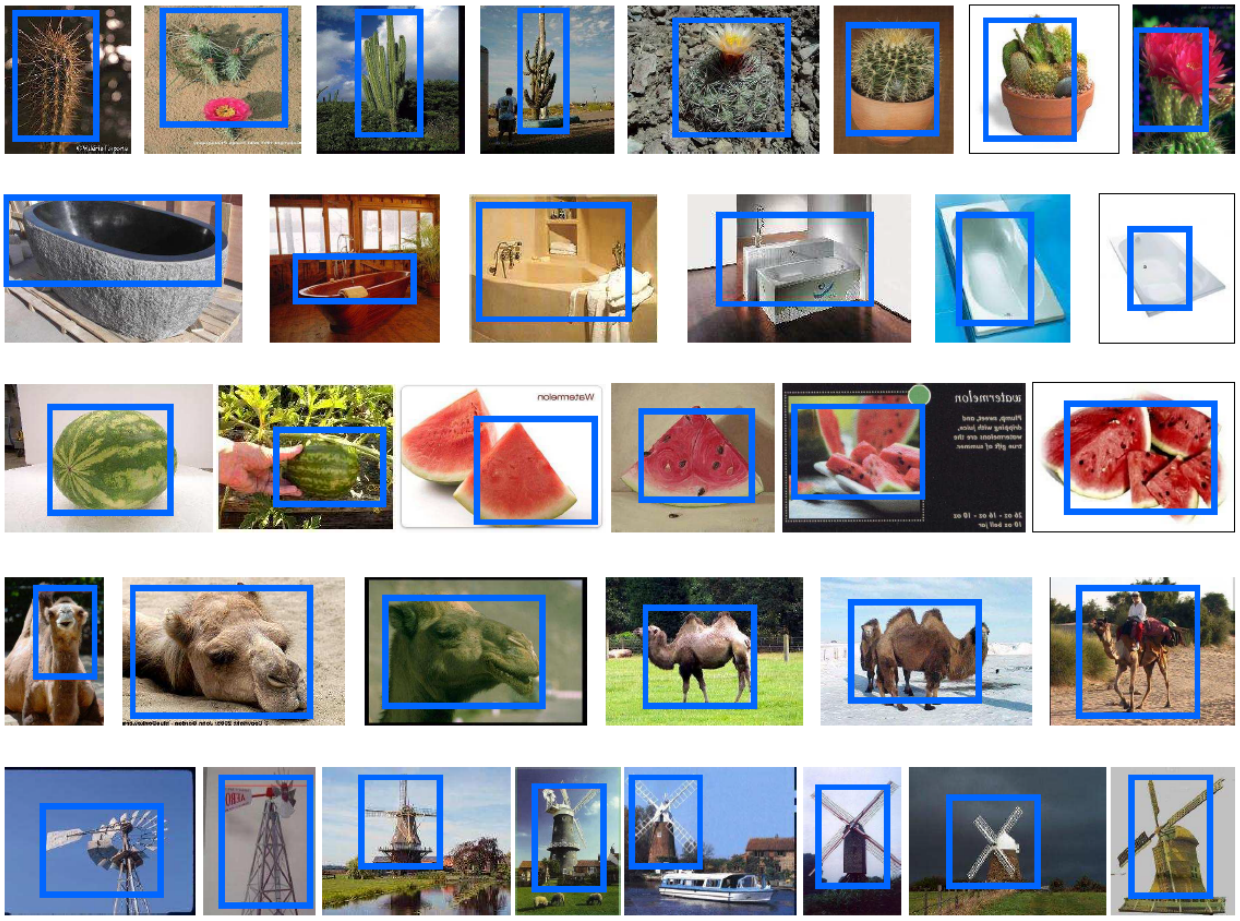


Figure 2.2: Automatic ROI detection on a subset of images within the Caltech-256 dataset by Bosch et al. [7].

Object Detection

As the SAR-UGV is automated, it needs to detect objects in real-time during mission operation. YOLO was the first image object detector to solve the real-time object detection problem with error rates that were competitive with state-of-the-art object detection models at the time. In "You Only Look Once: Unified, Real-Time Object Detection," Redmon et al. pose object detection as a regression problem, whereas traditional models detected objects within an image with repurposed classifiers [9]. In their model, separate components of object detection are unified into a single neural network, in which the neural network uses features within the entire image for prediction of where an object is identified. A bounding box is drawn surrounding the identified object. YOLO is also capable of predicting all

bounding boxes across all segmentation classes simultaneously for an image. The YOLO design "enables end-to-end training and real-time speeds while maintaining high average precision [9]." In other words, YOLO was faster than other models it was compared to, and resulted in error rates similar to the other slower, non-real-time models.

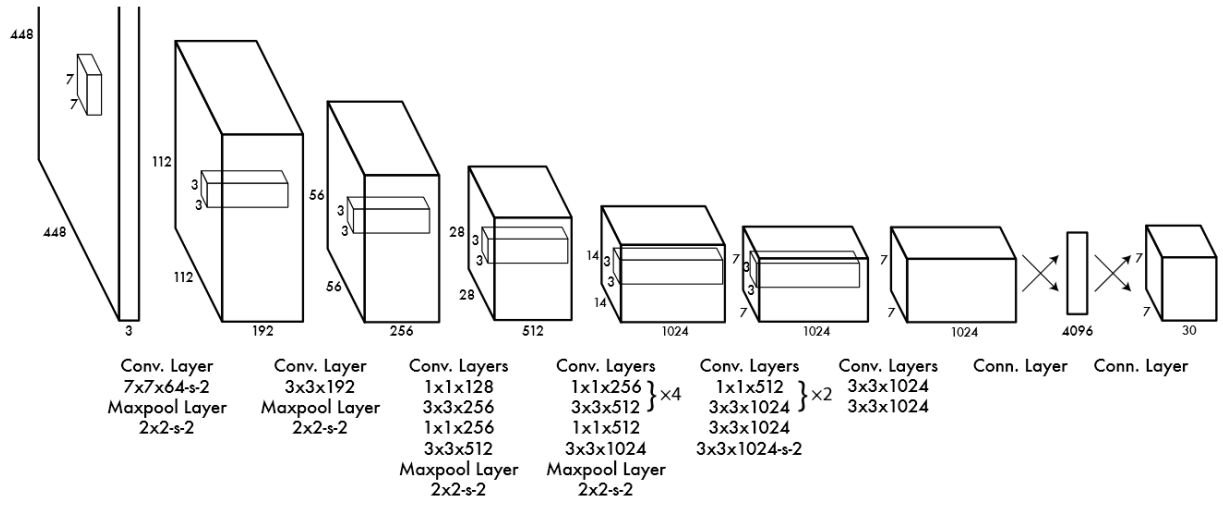


Figure 2.3: YOLO system architecture [9].

YOLO's system architecture, visualized above in Figure 2.3, features a neural network containing 24 convolutional layers followed by two fully connected layers. Alternating 1×1 convolutional layers reduce the feature space from preceding layers. The convolutional layers were pre-trained on the ImageNet classification task at half resolution (224×224 pixels). Being that object detection typically requires fine-grained visual information, the input resolution of the network is increased from 224×224 pixels to 448×448 pixels.

Figure 2.4 shows a graphical representation of the YOLO system model. The image is divided into an $S \times S$ grid. For each cell within the grid, B bounding boxes are predicted with confidence intervals, as well as C class probabilities from a class probability map. Object detection predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor. The final output displays colored bounding boxes around each uniquely detected and identified object.

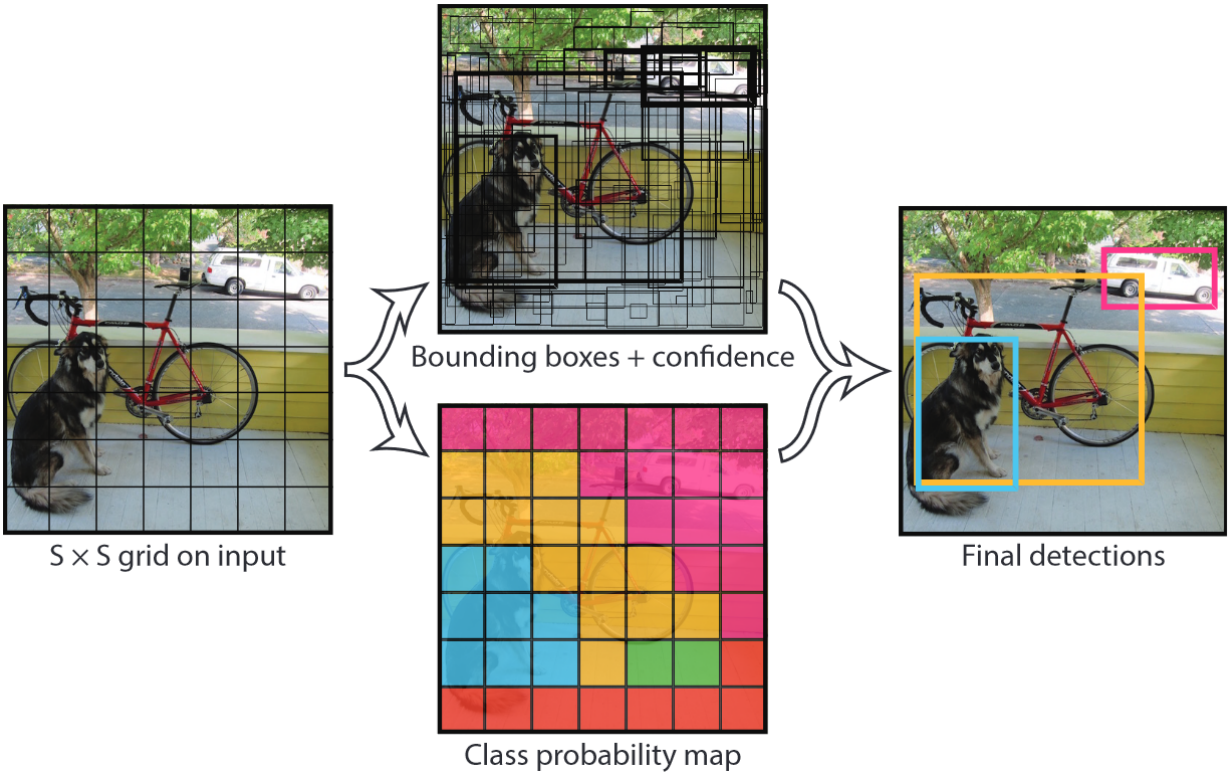


Figure 2.4: The YOLO system model as a regression problem [9].

YOLO was tested against other object detection models on November 6th, 2015 using the PASCAL VOC 2012 dataset [10]. In this test, YOLO was the only real-time object detection model. Hence, all other models had an advantage in regard to processing time. In this test, the Fast R-CNN model was combined with YOLO (Fast R-CNN + YOLO), which achieved the fourth-highest placement in mAP. Table 2.1 shows the object detection models tested for the PASCAL VOC 2012 dataset. The model performances as shown in the table are ranked by mAP. Other per-class average precision performances are also shown, in which the top performing object detection model for each individual object class is shown in bold text. Rows for both the YOLO and the R-CNN + YOLO models are highlighted in the table.

Since the its breakthrough in real-time object detection, YOLO has been updated multiple times. In their 2016 follow-up work, "YOLO9000: Better, Faster, Stronger," Redmon et al. simultaneously introduced YOLOv2 and YOLO9000. The authors claim that YOLOv2 is "state-of-the-art and faster than other detection systems across a variety of detection

Table 2.1: PASCAL VISUAL OBJECT CLASSES 2012 LEADERBOARD [9]

VOC 2012 Test	mAP	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv
MR_CNN_MORE_DATA	**73.9**	**85.5**	**82.9**	**76.6**	**57.8**	**62.7**	**79.4**	77.2	86.6	**55.0**	**79.1**	**62.2**	87.0	**83.4**	**84.7**	78.9	45.3	73.4	65.8	80.3	**74.0**
HyperNet_VGG	71.4	84.2	78.5	73.6	55.6	53.7	78.7	**79.8**	87.7	49.6	74.9	52.1	86.0	81.7	83.3	**81.8**	**48.6**	**73.5**	59.4	79.9	65.7
HyperNet_SP	71.3	84.1	78.3	73.3	55.5	53.6	78.6	79.6	87.5	49.5	74.9	52.1	85.6	81.6	83.2	81.6	48.4	73.2	59.3	79.7	65.6
Fast R-CNN + YOLO	70.7	83.4	78.5	73.5	55.8	43.4	79.1	73.1	**89.4**	49.4	75.5	57.0	**87.5**	80.9	81.0	74.7	41.8	71.5	68.5	**82.1**	67.2
MR_CNN_S_CNN	70.7	85.0	79.6	71.5	55.3	57.7	76.0	73.9	84.6	50.5	74.3	61.7	85.5	79.9	81.7	76.4	41.0	69.0	61.2	77.7	72.1
Faster R-CNN	70.4	84.9	79.8	74.3	53.9	49.8	77.5	75.9	88.5	45.6	77.1	55.3	86.9	81.7	80.9	79.6	40.1	72.6	60.9	81.2	61.5
DEEP_ENS_COCO	70.1	84.0	79.4	71.6	51.9	51.1	74.1	72.1	88.6	48.3	73.4	57.8	86.1	80.0	80.7	70.4	46.6	69.6	**68.8**	75.9	71.4
NoC	68.8	82.8	79.0	71.6	52.3	53.7	74.1	69.0	84.9	46.9	74.3	53.1	85.0	81.3	79.5	72.2	38.9	72.4	59.5	76.7	68.1
Fast R-CNN	68.4	82.3	78.4	70.8	52.3	38.7	77.8	71.6	89.3	44.2	73.0	55.0	87.5	80.5	80.8	72.0	35.1	68.3	65.7	80.4	64.2
UMICH_FGS_STRUCT	66.4	82.9	76.1	64.1	44.6	49.4	70.3	71.2	84.6	42.7	68.6	55.8	82.7	77.1	79.9	68.7	41.4	69.0	60.0	72.0	66.2
NUS_NIN_C2000	63.8	80.2	73.8	61.9	43.7	43.0	70.3	67.6	80.7	41.9	69.7	51.7	78.2	75.2	76.9	65.1	38.6	68.3	58.0	68.7	63.3
BabyLearning	63.2	78.0	74.2	61.3	45.7	42.7	68.2	66.8	80.2	40.6	70.0	49.8	79.0	74.5	77.9	64.0	35.3	67.9	55.7	68.7	62.6
NUS_NIN	62.4	77.9	73.1	62.6	39.5	43.3	69.1	66.4	78.9	39.1	68.1	50.0	77.2	71.3	76.1	64.7	38.4	66.9	56.2	66.9	62.7
R-CNN VGG BB	62.4	79.6	72.7	61.9	41.2	41.9	65.9	66.4	84.6	38.5	67.2	46.7	82.0	74.8	76.0	65.2	35.6	65.4	54.2	67.4	60.3
R-CNN VGG	59.2	76.8	70.9	56.6	37.5	36.9	62.9	63.6	81.1	35.7	64.3	43.9	80.4	71.6	74.0	60.0	30.8	63.4	52.0	63.5	58.7
YOLO	57.9	77.0	67.2	57.7	38.3	22.7	68.3	55.9	81.4	36.2	60.8	48.5	77.2	72.3	71.3	63.5	28.9	52.2	54.8	73.9	50.8
Feature Edit	56.3	74.6	69.1	54.4	39.1	33.1	65.2	62.7	69.7	30.8	56.0	44.6	70.0	64.4	71.1	60.2	33.3	61.3	46.4	61.7	57.8
R-CNN BB	53.3	71.8	65.8	52.0	34.1	32.6	59.6	60.0	69.8	27.6	52.0	41.7	69.6	61.3	68.3	57.8	29.6	57.8	40.9	59.3	54.1
SDS	50.7	69.7	58.4	48.5	28.3	28.8	61.3	57.5	70.8	24.1	50.7	35.9	64.9	59.1	65.8	57.1	26.0	58.8	38.6	58.9	50.7
R-CNN	49.6	68.1	63.8	46.1	29.4	27.9	56.6	57.0	65.9	26.5	48.7	39.5	66.2	57.3	65.4	53.2	26.2	54.5	38.1	50.6	51.6

datasets. Furthermore, it can be run at a variety of image sizes to provide a smooth tradeoff between speed and accuracy." YOLO9000 is described by Redmon et al. as "a real-time framework for detection (of) more than 9000 object categories by jointly optimizing detection and classification [11]." Aside from the massively enhanced number of classification categories of YOLO9000, improvements of YOLOv2 enabled it to surpass the mAP accuracy of other object detection models in real-time with 288×288 pixel images from the PASCAL VOC 2007 dataset, even at increasing frame rates exceeding 90 fps. A graph of mAP versus fps performance for YOLOv2 and other non-real-time object detection models is shown below in Figure 2.5.

The latest YOLO iteration, YOLOv3, was compared not only against non-real-time object detection models, but other real-time models, as well. In "YOLOv3: An Incremental Improvement," Redmon et al. demonstrate that YOLOv3 outperforms other real-time object detection models such as SSD, R-FCN, DSSD, and RetinaNet on the COCO dataset [12] in both inference time and mAP [13]. For example "YOLOv3-320" is an application of the

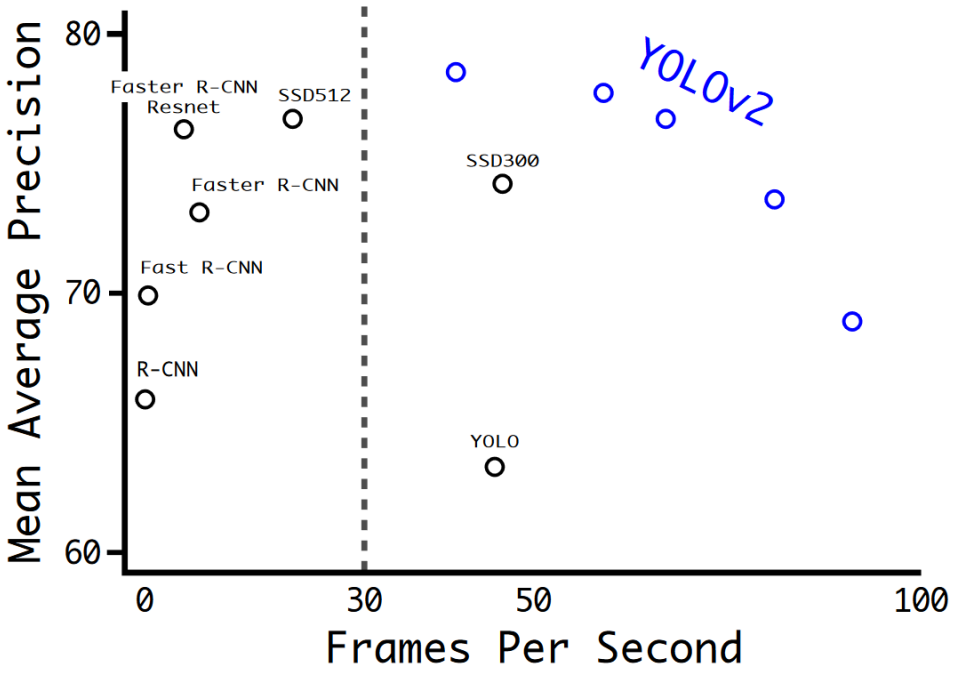


Figure 2.5: mAP performance versus increasing frame rates. YOLOv2 achieves higher mAP percentages than other non-real-time image object detection models while operating at higher frame rates [11].

YOLOv3 model with 320×320 pixel resolution images. COCO mAP-50 (percentage) versus inference time comparing the performance of YOLOv3 with other real-time image object detection models for various image sizes is shown in Figure 2.6.

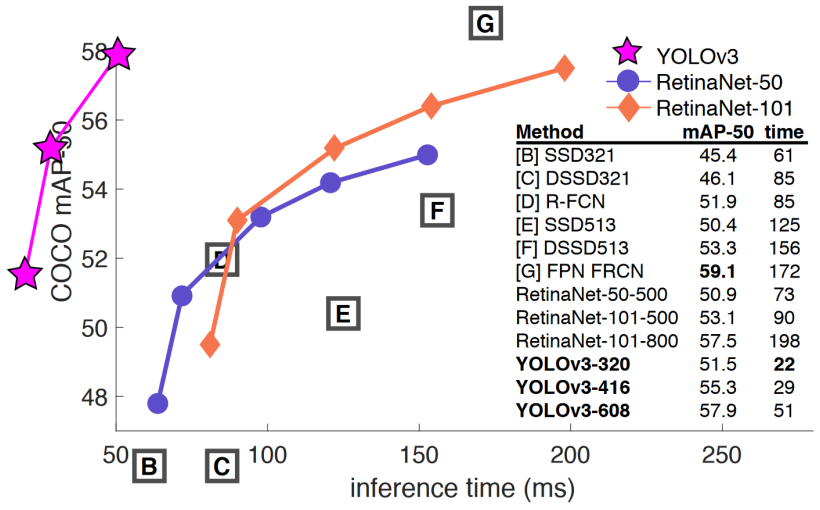


Figure 2.6: COCO mAP-50 performance versus inference time (ms) comparing real-time image object detection models including YOLOv3 [13].

Output of an image that has been processed by YOLOv3 is shown below in Figure 2.7. It can be seen that this object detection model is capable of distinguishing multiple objects from multiple different image classes simultaneously by the bounding box and correct label surrounding each detected object. This kind of object detection model will be extremely useful for an automated robot trained on objects common in a trail environment.

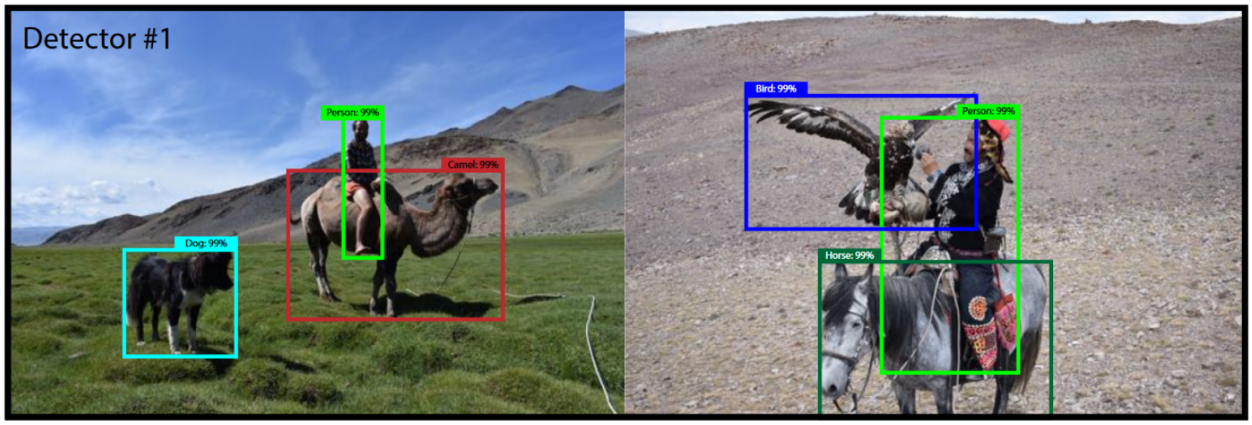


Figure 2.7: Image output after YOLOv3 processing. The processed images overlay a bounding box and label for identified objects [13].

Image Segmentation

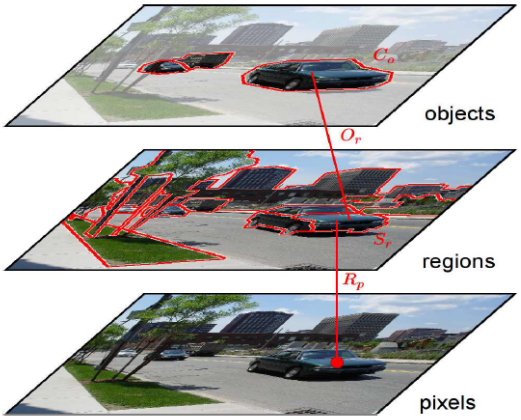
Image segmentation may be executed using sliding windows and multi-class image segmentation. This was the foundation for the work of Gould et al. in "Region-based Segmentation and Object Detection," where the authors present an integrated region-based hierarchical model for joint object detection and image segmentation [14]. This model uses a technique similar to that of multi-class image segmentation, as every pixel of the image is uniquely explained. The model also uses shape and appearance features over candidate object locations with precise boundaries, as is often implemented for object detection. This combined model applied over regions and objects in an image "allows context to be encoded through direct semantic relationships (e.g., 'car' is usually found on 'road') [14]."

The model of Gould et al. combines scene structure and semantics in a coherent energy function. Image scenes are decomposed into a number of semantically consistent regions. Each pixel has a local appearance feature vector, and each region has an appearance vector. This summarizes the region's appearance as a whole with a semantic class, S_r and object-correspondence variable, O_r. An associated class label, C_o, such as "car" or "pedestrian," is then applied to each object. Terms for modeling image horizon location, region label preferences, region boundary quality, object labels, and contextual relationships between objects and regions are included in the function. The combined energy function including these terms to capture image scene structure and semantics is shown in 2.2.

$$E = \psi^{hz}(v^{hz}) + \sum_{r} \psi_{r}^{reg}(S_r, v^{hz}) + \sum_{r,s} \psi_{rs}^{bdry} + \sum_{o} \psi_{o}^{obj}(C_o, v^{hz}) + \sum_{o,r} \psi_{or}^{ctxt}(C_o, S_r) \quad (2.2)$$

In 2.2 above, the energy function (E) is the sum of multiple individual terms made up of their own functions. The horizon term (ψ^{hz}) is implemented as a log-gaussian formula with parameters learned from training images to capture the image scene's deduced horizon location. The region term (ψ^{reg}) obtains the preference for assigning semantic labels to a region in the image. The boundary term (ψ^{bdry}) merges coherent pixels into a single region by penalizing two adjacent regions when they are of similar appearance (i.e., without boundary contrast). The object term (ψ^{obj}) provides scores the probability of an object label to be assigned to a group of image regions. The context term (ψ^{ctxt}) relates objects to the local background with contextual information for object detection improvement.

From the work of Gould et al., Figure 2.8 contains an series of photographs with overlaid illustrations of pixel, region, and object entities in a segmented image. The scene inference algorithm which computes the energy function in 2.2 is also displayed.



Procedure **SceneInference**
Generate over-segmentation dictionary Ω
Initialize R_p using any of the over-segmentations
Repeat until convergence
Phase 1:
Propose a pixel move $\{R_p : p \in \omega\} \leftarrow r$
Update region and boundary features
Run inference over regions $\mathbf{S}$ and v^{hz}
Phase 2:
Propose a pixel $\{R_p\} \leftarrow r$ or region move $\{O_r\} \leftarrow o$
Update region, boundary and object features
Run inference over regions and objects $(\mathbf{S}, \mathbf{C})$ and v^{hz}
Compute total energy E
If $(E < E^{\text{min}})$ then
Accept move and set $E^{\text{min}} = E$
Else reject move

Figure 2.8: Pixel, region, and object entities in a segmented image (left). Image inference algorithm from Gould et al. (right) [14].

The model created by Gould et al. was trained on a street scene dataset containing 3547 images of urban environments. Afterward, they concluded that when a boundary penalty term was too strong, all regions were merged together. Oppositely, when the boundary penalty was too weak, the image was oversegmented. They suggest that a more sophisticated inference procedure could be implemented in future work, but that their model "has the potential for providing holistic unified understanding of an entire scene." Further image segmentation results of Gould et al. using the street scene dataset is displayed in Figure 2.9.

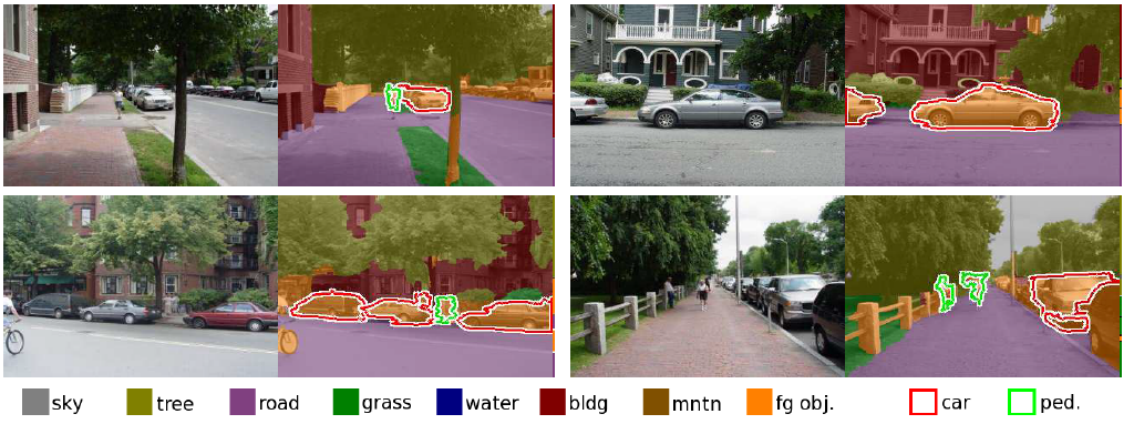


Figure 2.9: Image segmentation qualitative results from Gould et al. Examples show the original image (left) and color overlay by semantic class and detected objects (right) [14].

Trail Detection

An image processing system using methods similar to that of the work in "Accurate Natural Trail Detection Using a Combination of a Deep Neural Network and Dynamic Programming" by Adhikari et al. solves the problem of identifying the trail line needed for motion planning. Its patch-based DNN was trained with supervised data to classify fixed-size image patches into "trail" and "non-trail" categories, and reshaped to a fully convolutional architecture to produce trail segmentation map for arbitrary-sized input images [4]. The neural network architecture could conceivably be extended to recognizing other objects during SAR missions in addition to the trail line.

Similar work to identify trails was performed by Giusti et al. In their paper, "A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots," they describe their research methods that also employed a DNN to learn to perceive a trail from a single monocular camera [15]. Additionally, Guisti and his team were also able to create a simple autonomous control system to maintain the UAV position over the trail identified by the computer vision system. While the vehicle used in their research was a low-flying UAV, the difference as to whether the vehicle was low-flying or ground-based is irrelevant, as the trail perception and navigation tasks are the equally applicable to the SAR-UGV.

2.1.2 Navigation & Mapping

In order to operate autonomously and maintain an "awareness" of its position, the SAR-UGV must include a system capable of navigation and mapping, particularly in an outdoor forested environment. A popular implementation of such a of system that has been proven successful is SLAM. In "Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms," Durrant-Whyte and Bailey pose both the SLAM problem and solution:

> The Simultaneous Localisation and Mapping (SLAM) problem asks if it is possible for a mobile robot to be placed at an unknown location in an unknown

> environment and for the robot to incrementally build a consistent map of this environment while simultaneously determining its location within this map. A solution to the SLAM problem has been seen as a 'holy grail' for the mobile robotics community as it would provide the means to make a robot truly autonomous ... At a theoretical and conceptual level, SLAM can now be considered a solved problem. However, substantial issues remain in practically realizing more general SLAM solutions and notably in building and using perceptually rich maps as part of a SLAM algorithm. [16].

Engel et al. present a direct (feature-less) monocular SLAM algorithm that runs on a real-time CPU in their work "LSD-SLAM: Large-Scale Direct Monocular SLAM." The novelty of their method aligns two (camera) keyframes that explicitly incorporate and detect scale-drift, and takes a probabilistic approach to incorporate noise on the system's estimated depth map into its tracking. The algorithm shown in their article consists of three major components: 1) a *tracking* component that continuously tracks new camera images, 2) a *depth map estimation* component that uses tracked frames that either refine or replace the current keyframe, and 3) a *map optimization* component that incorporates keyframes into a global map.

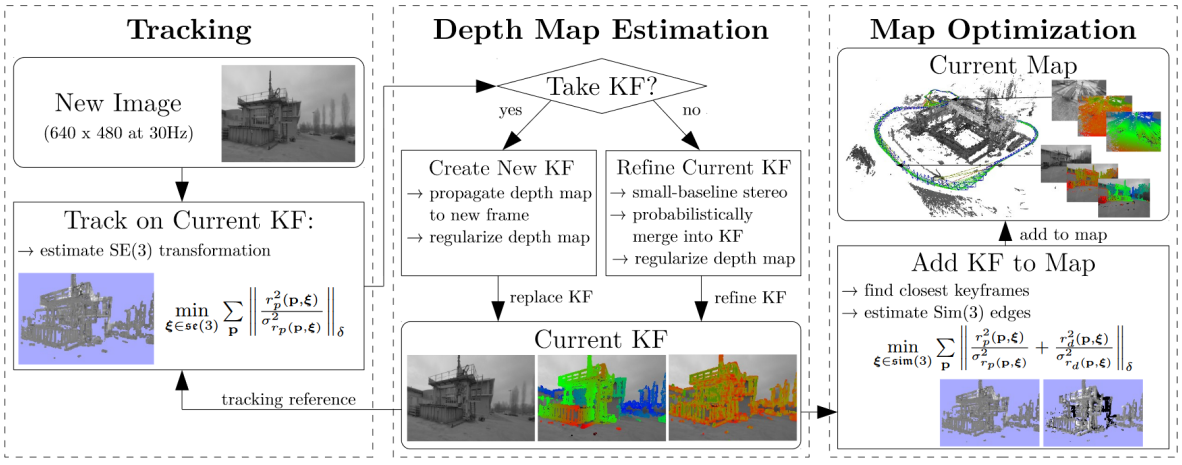


Figure 2.10: Overview of the Large-Scale Direct Monocoluar SLAM algorithm (LSD-SLAM) [17].

The authors state that the LSD-SLAM three-component algorithm is capable of switching seamlessly between differently scaled environments, including both indoor and large-scale outdoor environments. Finally, Engel et al. conclude that it is capable of tracking and mapping trajectories over 500 meters in length [17]. A graphical representation of the LSD-SLAM algorithm is displayed in Figure 2.10.

2.1.3 Autonomous Control Systems

Unlike the UAV used in the work of Guisti et al., the autonomous control system of the UGV must adapt to terrain variability, as the SAR-UGV is required to traverse a multitude of trail conditions such as dry dirt, sand, mud, gravel, and paved surfaces. In their work entitled "Overcoming the Loss of Performance in Unmanned Ground Vehicles Due to the Terrain Variability," Prado et al. specifically address the need to tune the control system of a UGV based on identified features and nature of the landscape. The terrain surface detection system in their paper contains raw data preprocessing, feature extraction, and classification. Based on image feature classification, the system includes motion strategy models, assessment metrics, and controller parameters with multiple approach algorithms including Monte Carlo seed generation, expectation-maximization, and Gaussian mean shifting [18]. In their work, the authors state that the "self-tuning methodology relies on the purpose of obtaining dynamically the best set of gains, $\hat{\mathbf{K}}$, in a bounded range where the stability of the closed-loop system is not affected." The expectation-maximization and Gaussian mean shifting probabilities return the estimation of $\hat{\mathbf{K}}$ with its corresponding covariance matrix, $\mathbf{\Sigma}$. The formula used for calculating the optimum gain set, $\hat{\mathbf{K}}$, which considers the dimension of the Monte Carlo seeds, n and α, as a latent membership variable associated with each probability distribution, $p\left(\mathbf{K}^{(i)}|\alpha, \mathbf{\Sigma}, \mu\right)$, is shown below in 2.3. This formula also considers the normal distribution, $N\left(\mu, \mathbf{\Sigma}\right)$, which is known as a result of the Monte Carlo algorithm presented in their paper prior to the calculation of $\hat{\mathbf{K}}$.

$$\hat{\mathbf{K}} = \underset{\mathbf{K}\in\mathbb{R}^n}{argmax}\left\{ log \prod_{i=1}^{\varepsilon} p\left(\mathbf{K}^{(i)}|\alpha, \mathbf{\Sigma}, \mu\right)\right\} \tag{2.3}$$

In what may be the most closely related thesis for both the image processing system, as well as the SAR-UGV project overall, "Monocular Vision-Based Obstacle Detection for Unmanned Systems" by Carlos Wang discusses computer vision theory topics such as feature extraction, tracking, mapping, and segmentation. With a display of project design, development, and implementation, the thesis also includes the application of the vision system to an autonomous vehicle assembled by a team at the University of Waterloo. Aspects of the systems comprising the autonomous vehicle include its mechanical design, electrical design (sensors & instrumentation, power distribution, and hardware), software design (software architecture, low-level communications, and control systems), LIDAR path planning, and vision processing [19]. Photographs of the autonomous vehicle in Wang's master's thesis are shown below in Figure 2.11.

Figure 2.11: Photographs of earlier stage production (left) and completed version (right) of the autonomous vehicle platform in Carlos Wang's master's thesis at the University of Waterloo [19].

2.2 Patent Landscape

Many US patents exist related to unmanned vehicles in general. Granted US patents encompassing both image processing and control systems for an unmanned ground vehicle, however, do not, as they combine multiple technologies and methods to a specific platform. In US Patent 7499776, "Systems and Methods for Control of an Unmanned Ground Vehicle," a very close combination of technologies are mentioned to be applied to a UGV [20]. The embodiments of this patent use the term "terrain feature" frequently in reference to hazards and obstacles to be avoided, but do not describe the type of terrain in regard to the kind of ground surface (e.g., dirt, sand, pavement, etc.) traveled by the UGV. A flowchart depicting a method for tracking such terrain features claimed in US Patent 7499776 is displayed in Figure 2.12. The patent's embodiments also use the terms "stereo vision" and "infrared vision," in addition to the description of other sensors as input devices for navigation and mapping, but nowhere mention the identification of a traversable area such as a trail or street. The patent's claims are even more general, and do not discuss either of the vision technologies contained in its embodiments. Instead, the claims are generalized to the control of the vehicle's acceleration, steering, and braking via input devices and actuators.

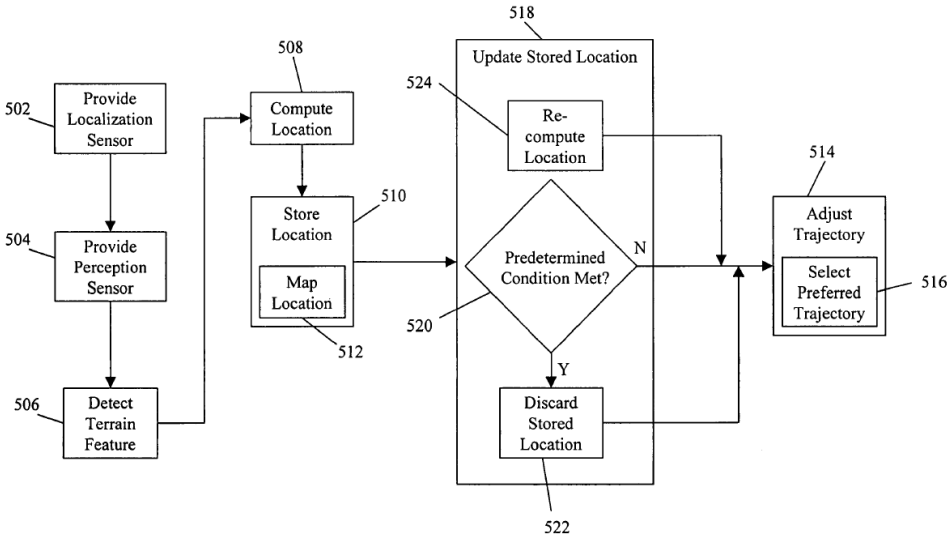


Figure 2.12: Flowchart depicting a method for tracking a terrain feature in US Patent 7499776 [20].

A granted patent that includes an unmanned vehicle for SAR applications is US Patent 9915945, "Lost Person Rescue Drone." The patented systems and methods center around a UAV that is capable of locating a lost individual, record GPS location data, record audio messages from the lost individual, and send the data to emergency response authorities [21]. The vehicle used in this patent is airborne, while the SAR-UGV operates on the ground. However, the patented system contains similar subsystems and performs the same tasks as the SAR-UGV proposed in this thesis such as GPS navigation, audio recording, location of lost individuals, and data transmission back to emergency response authorities (i.e., GS system).

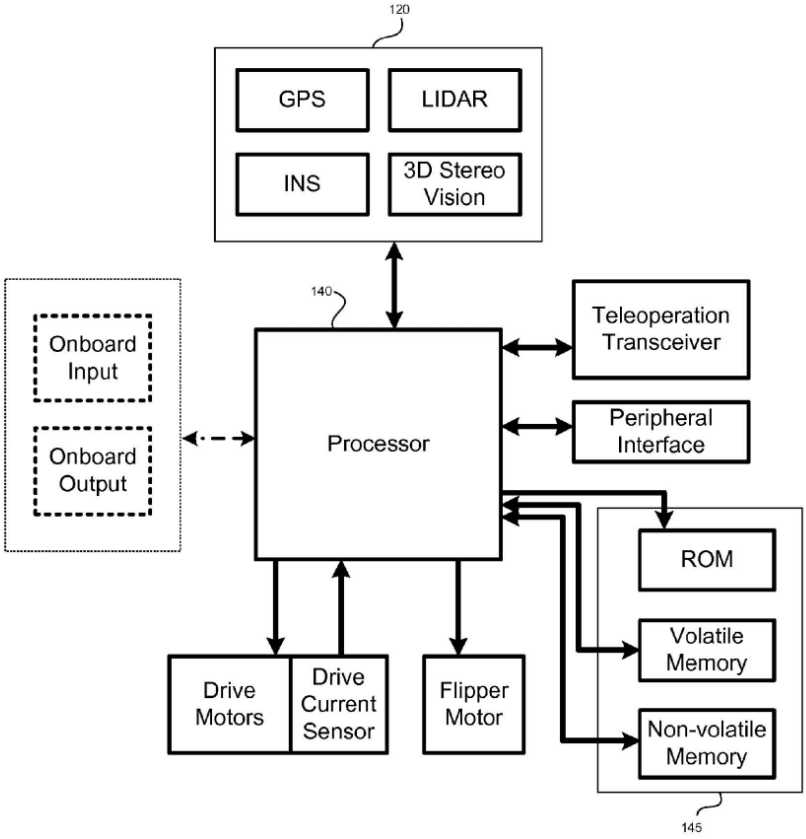


Figure 2.13: Block diagram depicting example hardware organization in US Patent 7539557 [22].

The most relevant patent to the suite of embedded systems in the SAR-UGV is US Patent 7539557 "Autonomous Mobile Robot," invented by Brian Yamauchi, and assigned to iRobot Corporation. The invention claims "a mobile robot for autonomously perform-

ing reconnaissance [22]." The intended military reconnaissance application of this invention makes it somewhat similar to that of the SAR-UGV in that both autonomously search for targets. While the nature of the given target in military applications could be widely varied, the specific target of the SAR-UGV is a lost person. The terrain application of the invention differs from the SAR-UGV, as the invention is primarily intended for urban environments, while the SAR-UGV is designed and developed for non-urban trails and forested terrain. The invention integrates a drive system, processor, memory, sensors (e.g., RADAR, LIDAR, etc.), GPS, and cameras; all of which are also required by the SAR-UGV specifications. The vision system described in the patent summary is a binocular 3D stereo vision system. The patent document also makes mention of video cameras and video feeds, but does not specifically refer to stereo cameras as the video source. This vision system is mentioned frequently throughout the patent document abstract and summary, signifying its importance. However, the vision system or any of its constituent parts for the Autonomous Mobile Robot do not appear whatsoever in the patent claims. The vision system of the SAR-UGV is capable of capturing both video and still images using multiple monocular cameras, where the cameras are oriented to obtain a 360° view around the robot. The vision system data of the SAR-UGV is intended to be integrated with data from other sensors and GPS for SLAM.

The Autonomous Mobile Robot invention in US Patent 7539557 also utilizes SLAM, and describes methodologies that may also be applied to the SAR-UGV. In addition to the Monte Carlo methodology presented in the patent document summary, the invention claims "a localization routine configured to update the occupancy grid map using a scaled vector field histogram based on input from the range sensor and the position reckoner integrated using a Kalman filter with a motion model corresponding to the mobile robot [22]." Kalman filter integration for sensor and vision data could improve SAR-UGV movement and positioning accuracy required by its autonomous control system. While the invention in US patent 7539557 differs in its intended military and environmental applications, its integrates multiple technologies that are also required by the proposed SAR-UGV.

2.3 Literature Review

A major breakthrough in fully autonomous UGV technology was the winner of the 2005 DARPA Grand Challenge, *Stanley*. Stanley was the robot designed and developed by the Stanford AI Lab at Stanford University. In the paper "Winning the DARPA Grand Challenge with an AI Robot," Montemerlo et al. discuss the embedded systems of the vehicle, including dynamic state estimation (e.g., location, speed, etc.), laser (LIDAR) mapping, vision (camera) mapping, and path planning [2]. In scale, Stanley differed greatly from the SAR-UGV, as it was designed using a full-size Volkswagen Touareg R5 automobile. However, many of the subsystems for vision, location, and navigation that were featured by Stanley are also required by the SAR-UGV. Figure 2.14 below displays photograph of the award-winning UGV (left) and a flowchart of the software supporting the hardware subsystems (right).

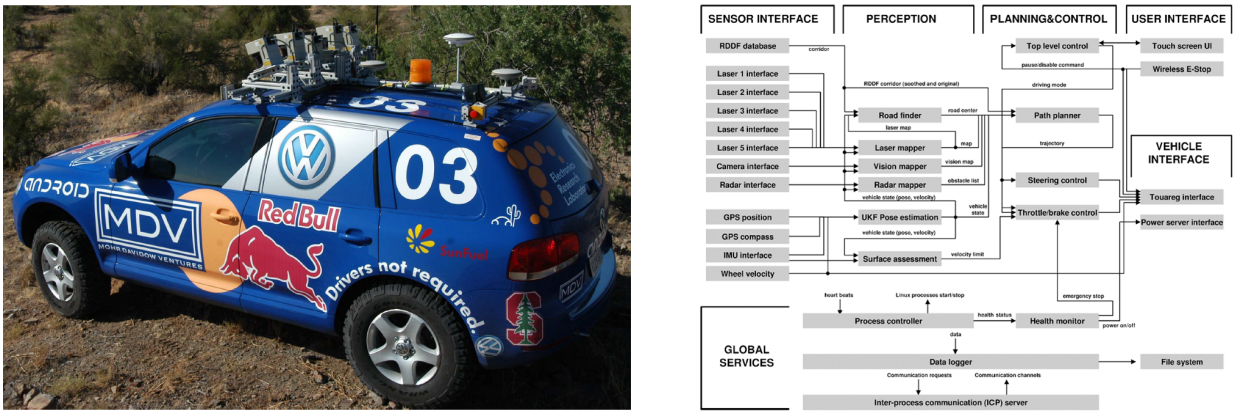


Figure 2.14: Photograph of Stanley, the 2005 DARPA Grand Challenge winner (left). Flowchart of Stanley's software, divided into six functional groups: sensor interface, perception, control, vehicle interface, user interface, and global services (right) [2].

Being that the SAR-UGV is designed to be fully autonomous, it must be equipped with software that allows it to make decisions for successful SAR missions. The data provided to the SAR-UGV prior to mission launch and the data collected by the vehicle's sensors during deployment are both important to the effectiveness of the SAR operation. This data must be analyzed by the SAR-UGV during the mission without human interaction and control.

Therefore, the SAR-UGV must be equipped with sophisticated decision making software for optimal results. In the work by Wysonkiński et al., "Decision Support Software for Search & Rescue Operations," the authors state that "planning the operation requires a specific analysis of the known facts, circumstances, weather conditions and topographical features of the considered area. Additionally, the understanding of the missing person motivation and behaviour plays a crucial role in effective SAR operation [23]." Their research includes lost person behavioral modeling with geographical context that calculates probabilities of how a lost person behaves in an unknown terrain. The models include data categories such as personality profiles (e.g., hiker, hunter, mental retardation, etc.) in combination with topographical features in the area of interest such as horizontal distance, vertical distance, and terrain accessibility (e.g., trails, water features, densely forested areas, etc.). The combined data produce probabilities for *find locations* that can be visualized in a heat map. When compared against random autonomous trail navigation within a general area for an SAR mission, utilizing software to generate probability data built from behavioral and topographical models, and combining it with live analysis performed by the image processing and autonomous control systems to actively search for targets where they are most likely to exist, conceivably increases the possibility of the SAR-UGV's mission success. Figure 2.15 presents a visualization of behavioral model probability with geographical context from Wysonkiński et al. Higher probabilities are presented as a more distinct red color in a heat map overlay on the geographic map. Wysonkiński et al., incorporate the acronym *IPP*, meaning "initial planning point." The IPP is the point in space where the lost person was last seen or their last known location, and appears at the center of each diagram in Figure 2.15. Table 2.2 displays find location type probabilities for two example target lost person profile categories.

Table 2.2: FIND LOCATION EXTRACTION PROBABILITY FROM LOST PERSON PROFILES [23]

Category	**Structure**	**Road**	**Linear**	**Drainage**	**Water**	**Brush**	**Scrub**	**Woods**	**Field**	**Rock**
Hiker	13%	13%	25%	12%	8%	2%	3%	7%	14%	4%
Mental Retardation	24%	12%	2%	5%	7%	7%	2%	31%	7%	2%

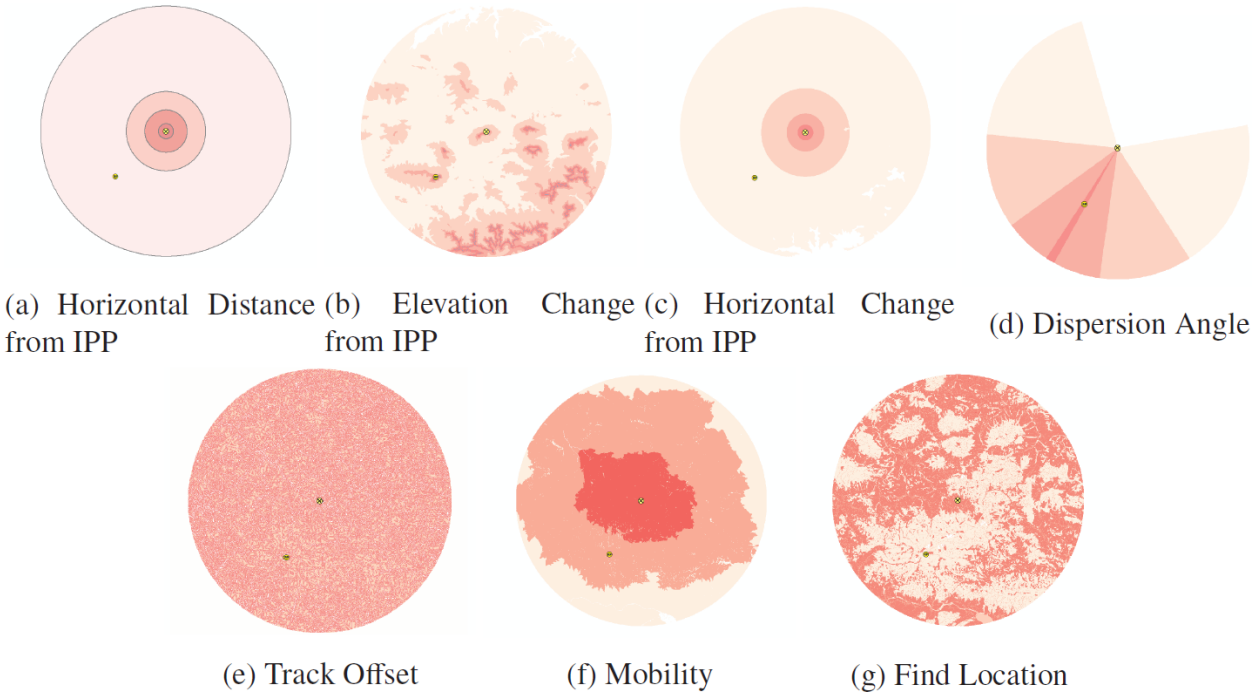


Figure 2.15: Visualization of behavioral models with geographical context to achieve find location probability in SAR missions [23].

In "Sensors and Tracking Methods Used in Wireless Sensor Network Based Unmanned Search and Rescue System - A Review," Thavasi et al., examine a list of sensors that may be applied to both aerial and ground-based robots in SAR operations. The sensors contained in their review include binary sensors (e.g., infrared & doppler shift), vibration sensors, radio sensors, range finding sensors (e.g., radio, ultrasound, and lasers), vision sensors (i.e., various types of cameras), heat sensors, thermopile sensors, sound sensors (i.e., microphones), as well as human gas emission sensors (e.g., CO_2 & SpO_2). These are compared in terms of what each sensor is capable of detecting, sensor size, cost, human distinction capability, and overall strengths and weaknesses. The authors conclude "that wireless sensor network and agent concepts can be effectively integrated to develop a highly autonomous robot for a(n) urban search and rescue application with a reduced communication cost [24]." This conclusion is especially promising being that the SAR-UGV project integrates many of the reviewed sensors in multiple wireless networks (e.g., ROS nodes) to communicate between its embedded systems, control the SAR-UGV, and alert upon detection of the presence of

the target lost person. Inclusion of additional sensors mentioned in this review not initially planned for the SAR-UGV would also create a more robust dataset to be utilized in future missions. Table 2.3 displays a summary of sensors reviewed in the work of Thavasi et al, which includes features such as each sensor's ability to distinguish human versus non-human targets, as well as overall strengths and weaknesses.

Table 2.3: SUMMARY OF HUMAN DETECTING SENSORS FOR SAR-UGV [24]

Sensor Type	**Technology**	**Detectable Feature**	**External Size**	**Cost**	**Human / Non-Human Distinction**	**Strengths**	**Weaknesses**
Linear Camera	CCD / CMOS, EM 0.4 µm - 1.1 µm	Vision	Very Low	Very Low	Very Low	Price	Low Resolution
USB Camera	CCD / CMOS, EM 0.4 µm - 1.1 µm	Vision	Low	Low	High	Cost / Performance	Resolution
Stereo Vision (Camera)	CCD / CMOS, EM 0.4 µm - 1.1 µm	Vision / Distance	High	High	High	Vision & Distance Information	Computationally Expensive
Infrared Camera	CCD / CMOS, EM 7 µm - 14 µm	Heat	High	Very High	Very High	Human Distinction	Price
Pyroelectric	Crystalline Sensor, EM 7 µm - 14 µm	Body Heat	Very Low	Very Low	High	Price, Human Distinction	Only Motion Detection
Thermopile	Thermocouple, EM 5.5 µm - 13 µm, −25 °C - 100 °C	Heat	Very Low	Very Low	High	Price	Only Average Temperature
Microphone	Membrane, SW 100 Hz - 16 kHz	Sound	Very Low	Very Low	High	Price	Noise Sensitive

Table 2.3: SUMMARY OF HUMAN DETECTING SENSORS FOR SAR-UGV [24] (CONTINUED)

Sensor Type	**Technology**	**Detectable Feature**	**External Size**	**Cost**	**Human / Non-Human Distinction**	**Strengths**	**Weaknesses**
Laser Range Finder (LIDAR)	Time of Flight / Triangul., EM 620 µm - 820 µm	Distance	High	Very High	Very Low	Precision of Measure	Price
Ultrasonic Sensor	Membrane, SW 130 kHz - 290 kHz	Distance	Very Low	Very Low	Very Low	Price	Echo Sensitivity
RADAR	Time of Flight, EM 5 GHz - 25 GHz	Distance	Low	Very High	Very Low	Precision with Bug Range	Price
CO_2 Sensor	Electro-chemical	Gas	High	High	High	Human Distinction	Too Directional
SpO_2 Sensor	Light Absorption (650 nm, 805 nm)	Blood Oxygen, Pulse Rate	Very Low	N/A	Very High	Human Distinction	Not Available for Robotics

SAR operations often occur under hazardous or otherwise unsafe conditions, which may drastically affect mission success rates. In their paper "Use of Unmanned Vehicles in Search and Rescue Operations in Forest Fires: Advantages and Limitations Observed in a Field Trial," Karma et al. present a comprehensive report of the use of both UAVs and UGVs for SAR operations in forest fires. They point out that payloads carried by unmanned vehicles can provide valuable data from their sensors. Payloads may also include medical kits that can be dispatched to victims who are trapped in dangerous areas, while at the same time mitigating risk of SAR personnel safety. However, unmanned vehicles can also be as vulnerable as their human counterparts to extremely high temperatures above 400° C [25]. This research is particularly relevant, as forested locations, which can be susceptible to wildfires, are the primary intended operation environment for the SAR-UGV.

CHAPTER 3 System Architecture & Specifications

3.1 Design Specifications

In the first chapter of this thesis, Section 1.4, "R&D Objectives and Thesis Contributions," the design specifications of each system making up the SAR-UGV SoS were itemized. Taking this itemized list as its guide, the image processing system of the SAR-UGV is designed to meet the following specifications:

- Navigate hiking trails without human control/interaction
- Determine optimal course on a trail
- Perform obstacle avoidance
- Utilize data from other SAR-UGV subsystems and sensors via ROS nodes and topics:
  - GPS receiver
  - Accelerometer
  - LIDAR
  - Live camera video image feeds
- Collect GPS waypoints during mission
- Return to the GS upon command using aforementioned GPS waypoints

3.2 System Level Design

All SAR-UGV systems that communicate data via ROS, as well as the power distribution system that powers these systems, are displayed below in Figure 3.1. The GS appears at the top of the block diagram, which communicates with both wireless control and radio communications systems. The wireless connection between the GS and each of these systems is indicated by dashed lines. Power provided by the power distribution system is indicated by dotted lines. Communication between systems via ROS is denoted by solid lines. The autonomous control and image processing (i.e., video processing) systems are highlighted as bold blocks on the right side of the diagram. An additional bold block is included for the camera array that interfaces directly with the image processing system.

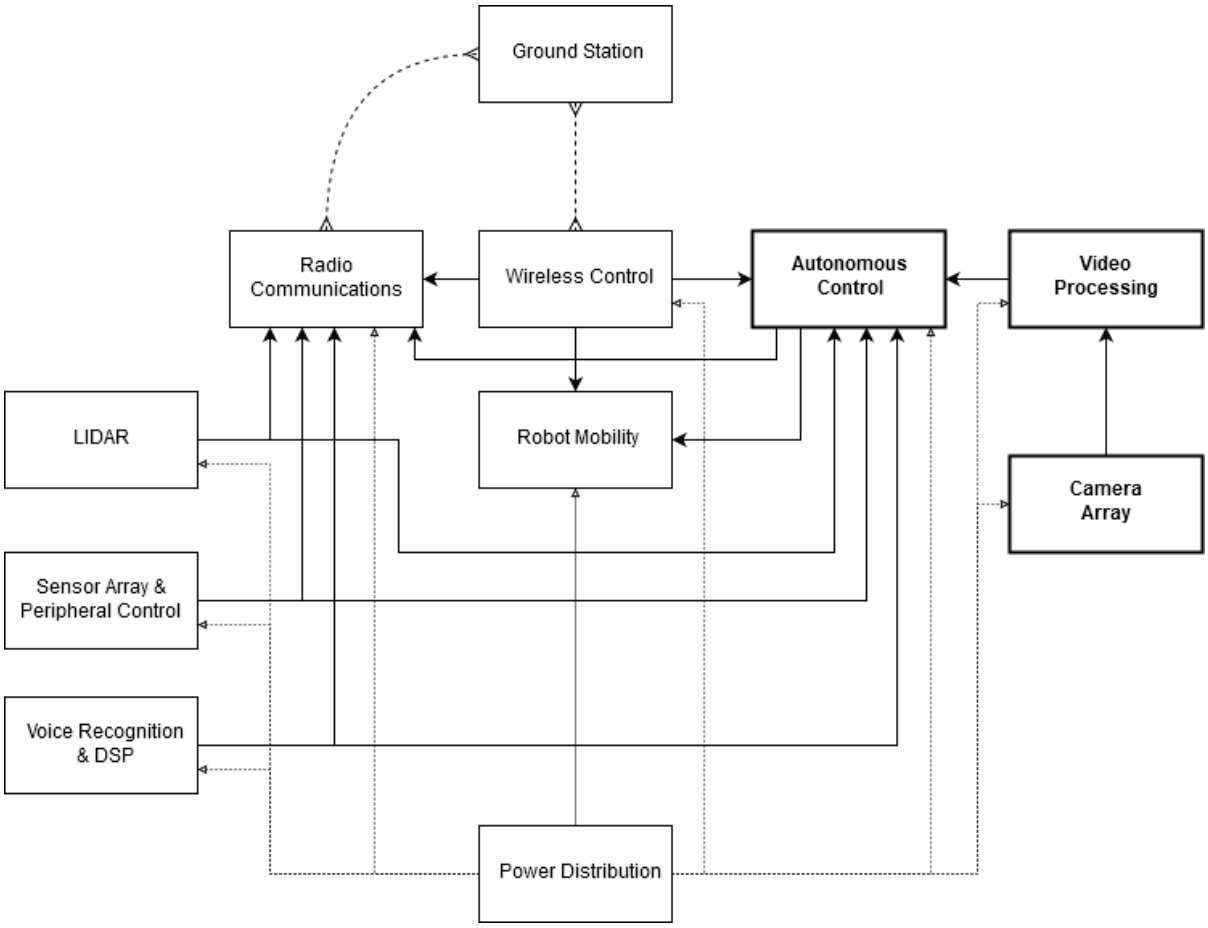


Figure 3.1: SAR-UGV systems block diagram based on initial specifications.

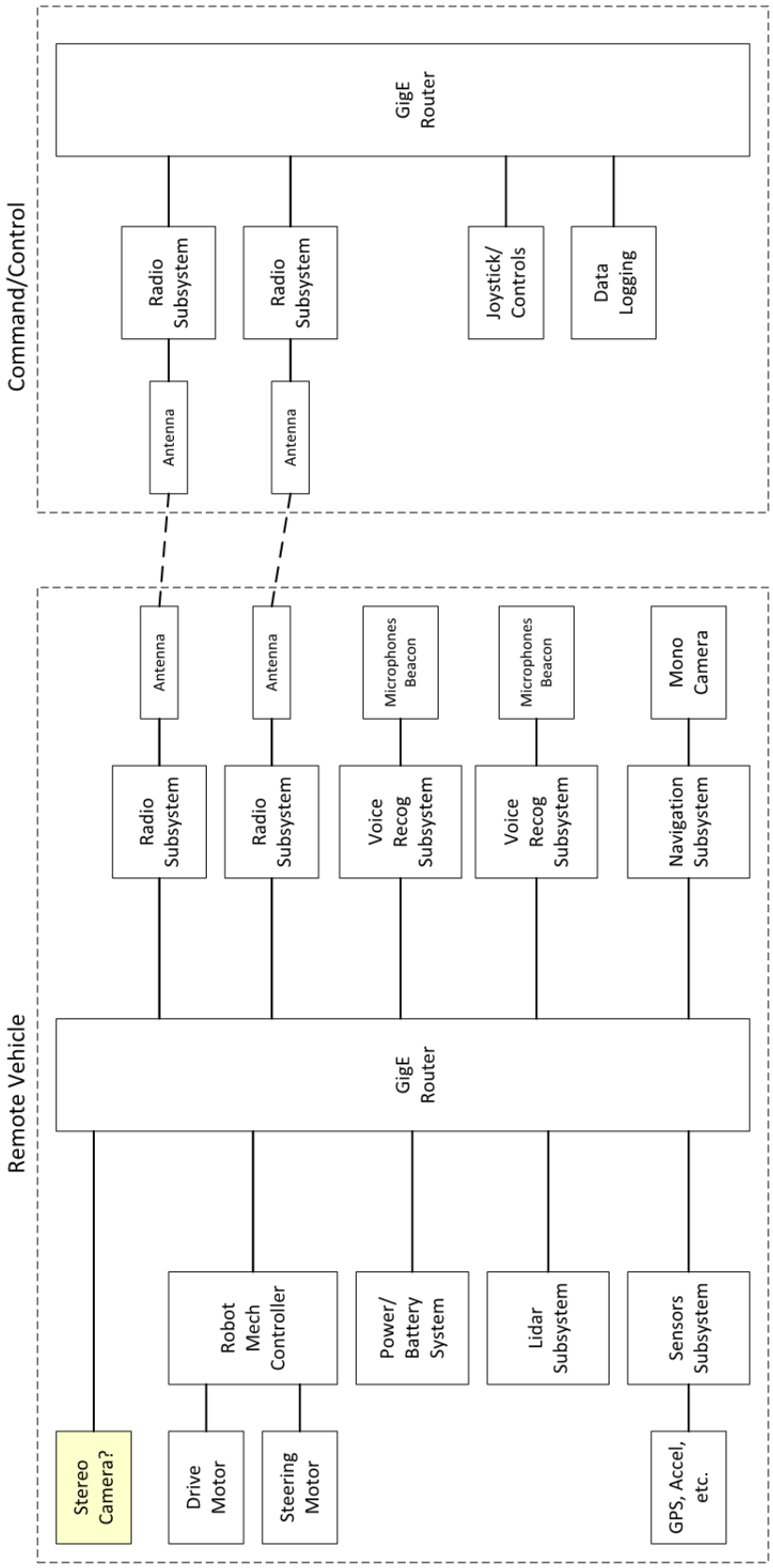


Figure 3.2: SAR-UGV systems block diagram as initially drafted by Professor Allan Douglas.

The team of MSE graduate students involved in the SAR-UGV project met to brainstorm the ROS network. ROS nodes were introduced based on each students' individual project needs. This included many ROS nodes that were not originally envisioned for the SAR-UGV network design shown previously in Figure 3.1. A photograph of the initial design of the ROS network to meet the messaging needs of all student subsystems on the SAR-UGV is displayed below in Figure 3.3.

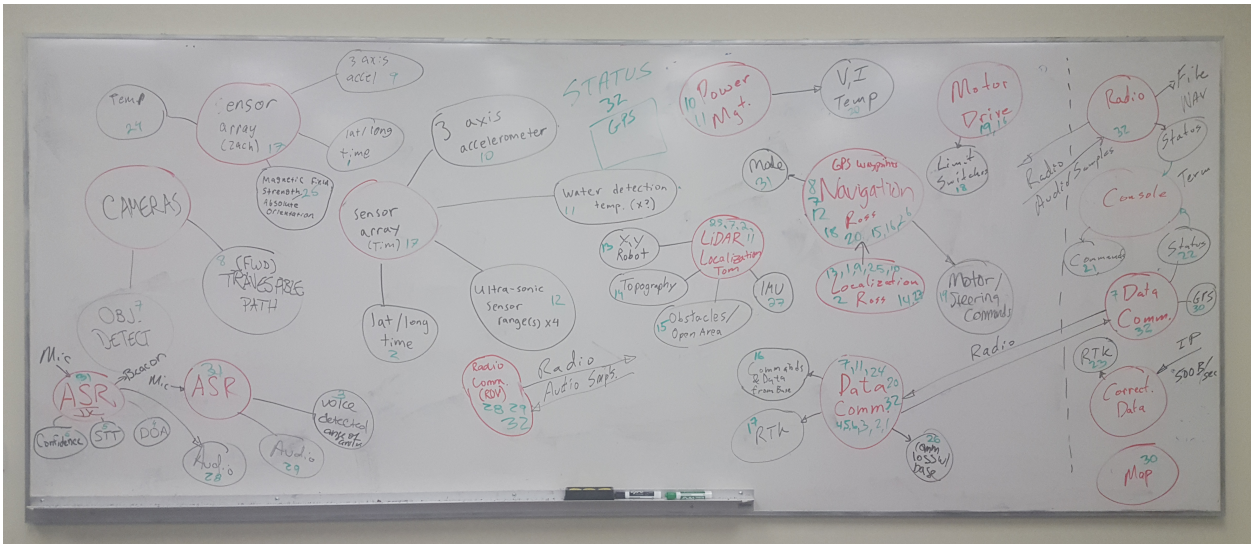


Figure 3.3: Whiteboard result for the first group draft of the SAR-UGV ROS network. Photograph credit: Professor Allan Douglas.

The drafted ROS network after the first brainstorming meeting to address communication between subsystems included additional nodes and sensors that were not initially specified by the project. Sensors added to the SAR-UGV as part of other students individual projects include:

- Temperature sensors
- Water detection sensors
- 3-axis accelerometers
- Magnetic field sensors

- Geolocation sensors
- Ultrasonic range finders
- IMUs
- Limit switches
- Voltage sensors
- Current sensors

Professor Allan Douglass was heavily involved with the initial brainstorming discussions. During this phase of the project, many students were unsure as to what their system contribution would be for the SAR-UGV. He was able to inspire and motivate students by examining and explaining the needs of the robot in SAR missions in an effort to help students start their projects. Furthermore, none of the students involved in the SAR-UGV design project had previous experience working with ROS. Professor Douglas' guidance was absolutely essential in orienting the MSE team with the ROS software, and pointing students to the resources available to begin ROS development to facilitate communications between multiple systems.

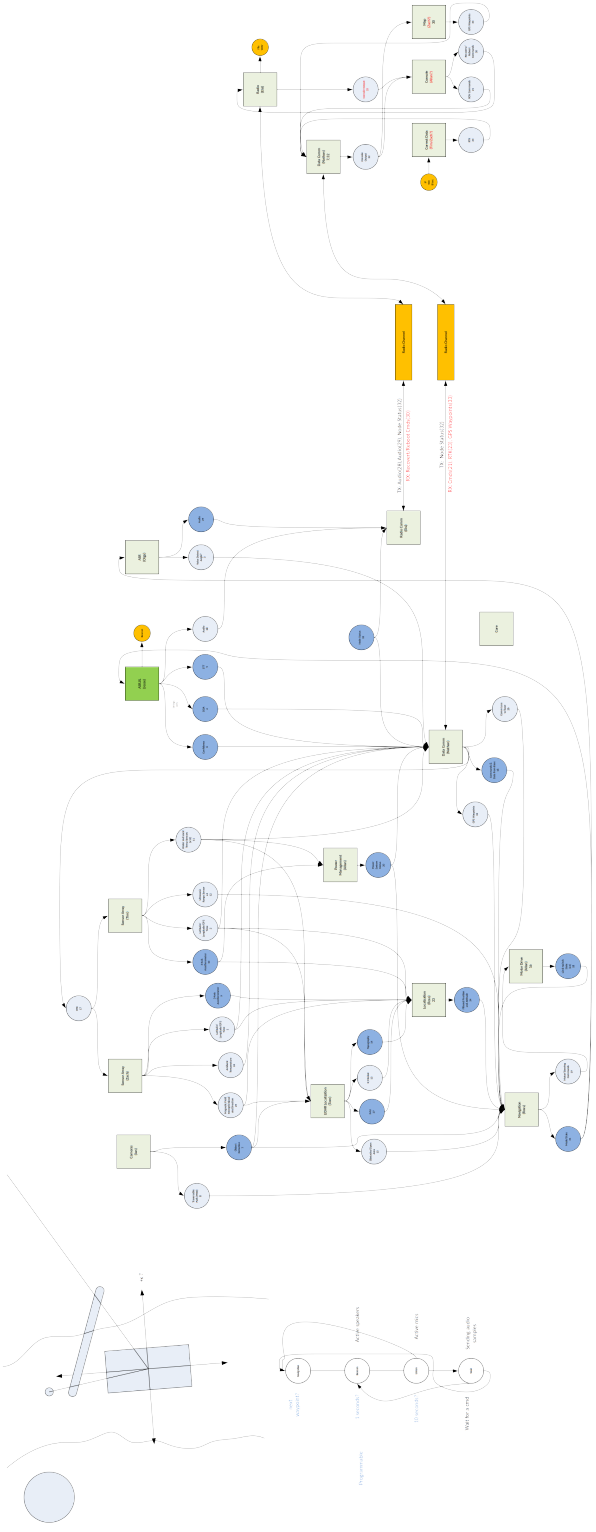


Figure 3.4: SAR-UGV systems block diagram final draft by Professor Allan Douglas.

3.3 ROS Network Testing

Prior to building a ROS network, it was decided that all machines on the network would be capable running ROS Melodic Morena (i.e., ROS Melodic) distribution release [26]. ROS Melodic is compatible with Ubuntu 17.10 (Artful Ardvark), Ubuntu 18.04 (Bionic Beaver), and Debian 9 (Stretch) [26]. If the case occurred that a specific machine or subsystem was not capable of running ROS Melodic, its ROS node(s) [27] may need to be uniquely configured to communicate with the ROS Master [28], as communication between a ROS node and the ROS Master running on different ROS distributions could require additional setup and troubleshooting.

The ROS website states that the purpose of the ROS Master machine is to provide "naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other, they communicate with each other peer-to-peer [28]." By contrast, the ROS website defines a ROS node as "a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes [27]."

Once multiple Ubuntu 18.04 machines had ROS Melodic installed, and were connected on the same local area network at the Oregon Tech campus, the ROS *turtlesim* tutorial [29] was followed. With one machine on the network acting as the ROS Master, and all others running ROS nodes, it was demonstrated that the machines could communicate with each other by remotely operating turtles on multiple ROS nodes. Figure 3.5 displays a screenshot of an Ubuntu 18.04 computer running the turtlesim ROS Master. The figure shows the operation of the ROS Master (`$ roscore`) and other operations in multiple terminal windows, such

as running a turtlesim node (`$ rosrun turtlesim turtlesim_node`) and remotely operating a turtle on the node (`$ rosrun turtlesim turtlesim_teleop_key`). A TurtleSim window (capitalization included for specificity) was also displayed by running the ROS node, which showed the traveled path of the turtle that was controlled over the network. Additionally, Figure 3.5 displays an `rqt_graph`, brought up by running the command `$ rqt_graph` in a Terminal, which shows a network "map," including the publisher, subscriber, and topic relationships of the ROS nodes on the network.

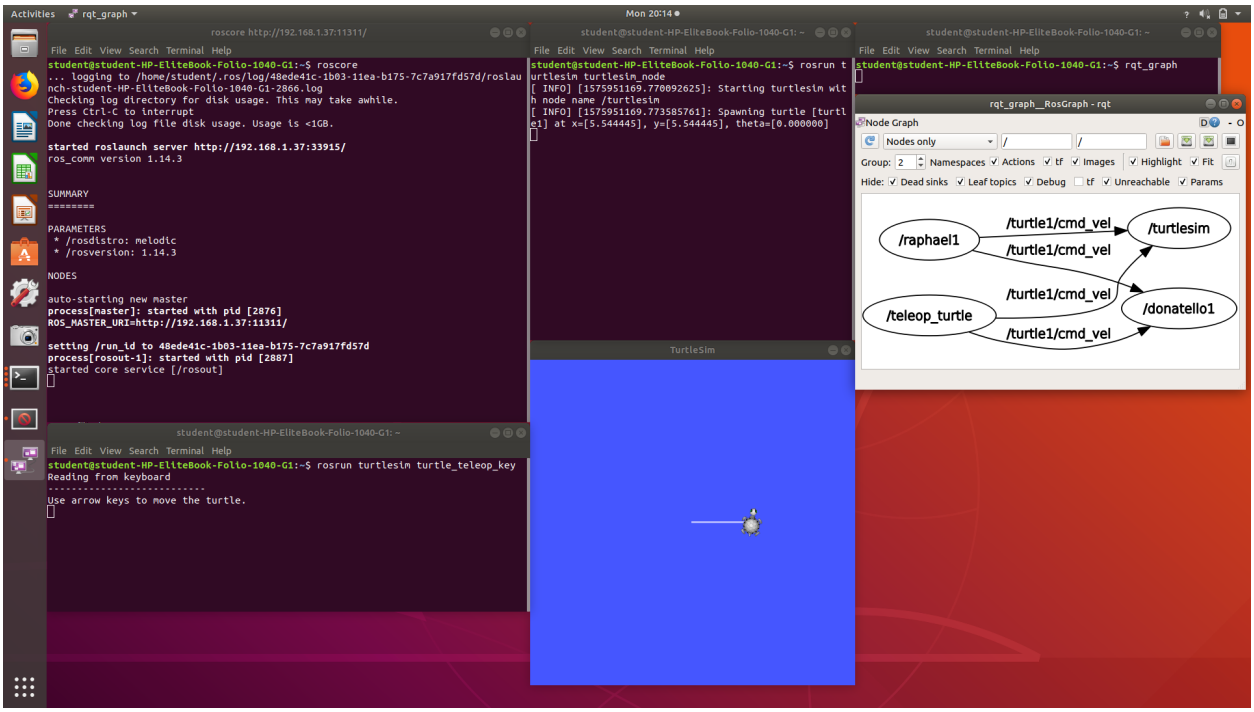


Figure 3.5: Screenshot of a computer running the turtlesim demo as the ROS Master. Photograph credit: Zachary Hofmann.

A computer acting as a ROS node machine on the network (i.e., not the ROS Master) is not limited to running only one single ROS node. Multiple ROS nodes may be simultaneously activated by a single machine. However, it is required that each ROS node running on a network has a unique name by adding the `__name` parameter. The term `teleop_key` specifically refers to the ROS node that publishes keystroke data as a ROS topic, while the `turtlesim` node acts as the subscriber of the topic published by the `teleop_key`

node. This is demonstrated in Figure 3.6, where the turtlesim tutorial was run on a single Ubuntu computer with two distinct `turtlesim` nodes and two distinct `teleop_key` nodes, each with unique names. This resulted in two separate TurtleSim windows that displayed the paths of turtle navigation. The relationship between the `teleop_key` publisher and `turtlesim` subscriber nodes, and `cmd_vel` topic that is shared between them, can be seen by the `rqt_graph` window within the previous screenshot image in Figure 3.5, where each active ROS node on the network exhibits a unique name.

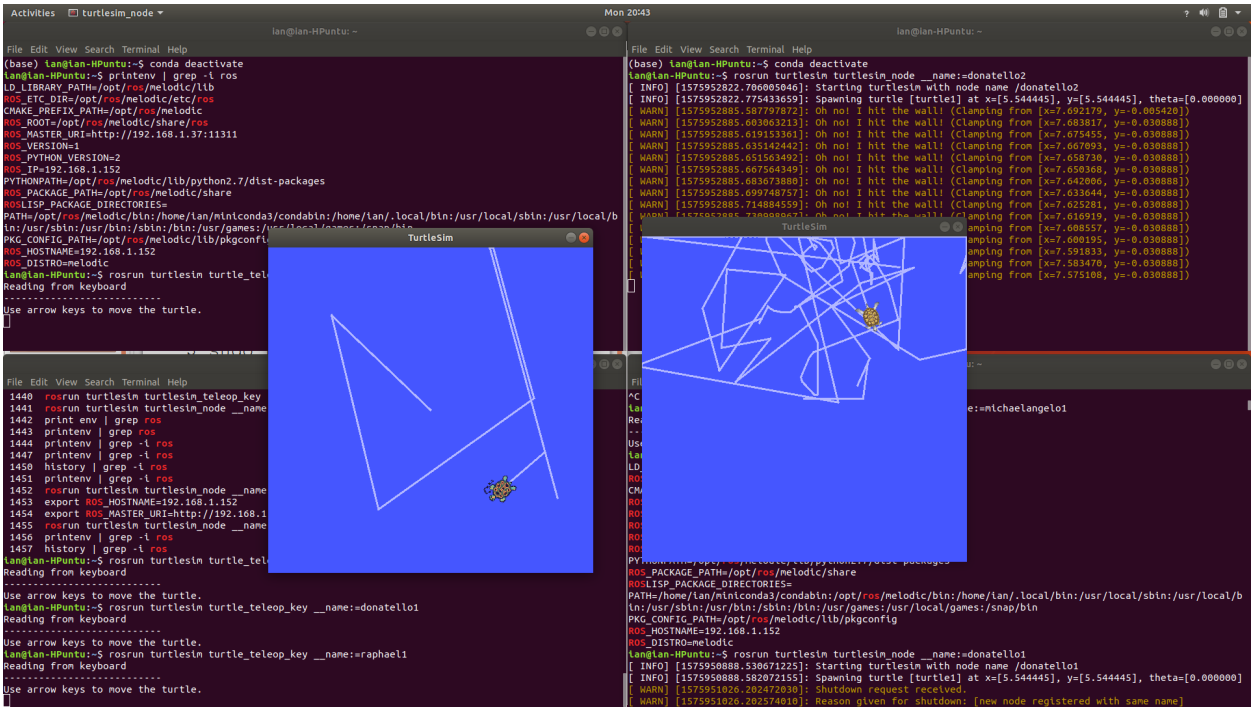


Figure 3.6: Screenshot of a computer running the turtlesim demo with multiple ROS nodes.

CHAPTER 4 Design & Development

The image processing system of the SAR-UGV required a 360° view around the robot. It needed to perform semantic segmentation to distinguish traversable trails versus non-traversable areas. An additional aspect of image processing was object identification, in which entities such as humans, animals, trees, and other things commonplace in an outdoor trail setting could be recognized. Overall scene understanding via image classification was also to be performed. Data resulting from image classification, object identification, and image segmentation were used to provide data to the control subsystem for obstacle avoidance and successful navigation of the vehicle via the SAR-UGV ROS network.

4.1 Design

4.1.1 Hardware

The first design step in this project was to review and acquire hardware to fulfill the requirements of the image processing system. These requirements included a computing platform device capable of collecting image data from connected cameras and an operating system capable of running ROS. It was also required to contain sufficient storage space to collect image and video data.

The NVIDIA Jetson TX2 Developer Kit was chosen as the backbone of the image processing system. Processing power and connectivity featured by the Jetson TX2 [30] fulfilled the specifications of the image processing system, and, hence were the reasons for selecting

this particular computing platform. A list of features for the NVIDIA Jetson TX2 Developer Kit are as follows:

- 256-core NVIDIA Pascal™ GPU
- Dual-Core NVIDIA Denver 1.5 64-Bit CPU and Quad-Core ARM® Cortex®-A57 MPCore processor
- 1 x1 + 1 x4 PCIe Gen2 connector
- (Up to) 6 CSI cameras–12 lanes MIPI CSI-2 (CSI cameras could have been selected as an alternate camera connectivity method.)
- 500 MP/sec video encode
- 1000 MP/sec video decode
- HDMI 2.0 Display
- 10/100/1000 BASE-T Ethernet and WLAN network connectivity
- USB 3.0 connector

Additional hardware on the Jetson TX2 Developer Kit not listed above that was also utilized in this project were its SATA connector and RP-SMA connectors. The SATA connector made it possible to expand the storage disk capacity beyond its onboard 30 GB eMMC drive. The RP-SMA connectors allowed connection of antennas that could fit in an enclosure to maintain wireless connectivity–both WiFi and BlueTooth–to the image processing system from a remote computer.

The Jetson TX2 Developer Kit runs the Ubuntu 18.04 LTS operating system as part of its software release, JetPack 4.3.1. This operating system is capable of running ROS Melodic, which was agreed upon as the ROS version to be used by the MSE graduate project team for all system components that comprise the SAR-UGV. Some other system components of the

SAR-UGV, such as the LIDAR and autonomous control systems, also ran on the Jetson TX2 platform with the same version of JetPack. An image of the NVIDIA Jetson TX2 Developer Kit is shown in Figure 4.1.

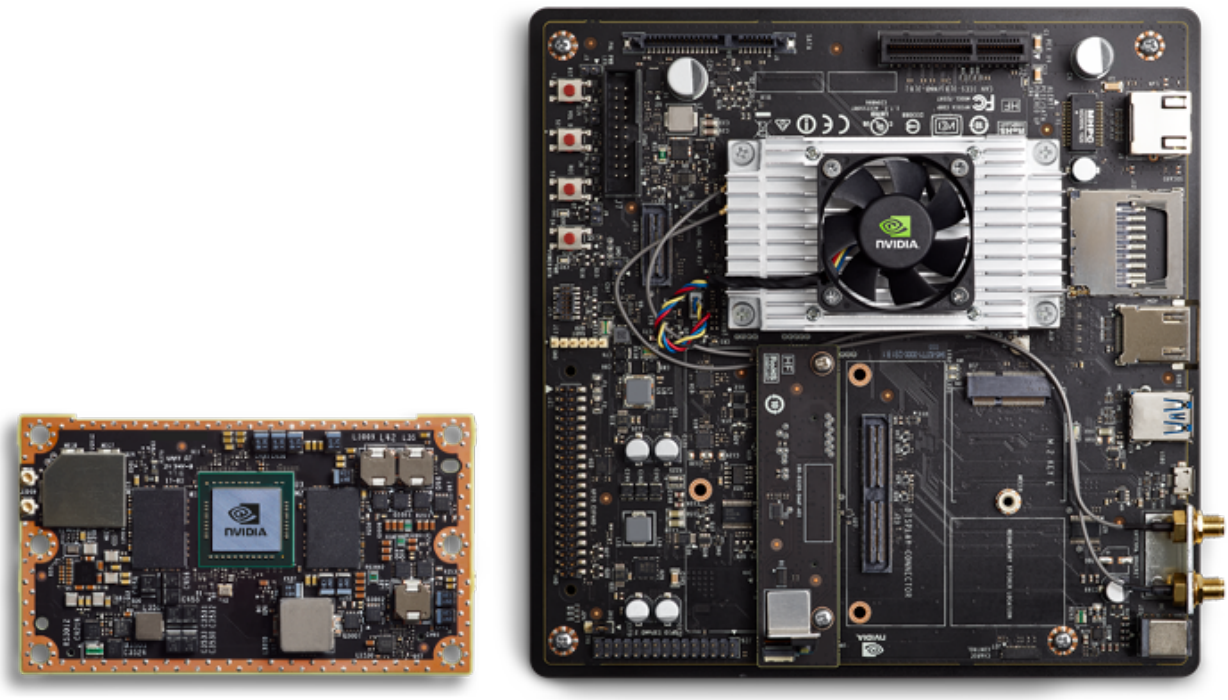


Figure 4.1: The Jetson TX2 module (left). The Jetson TX2 Developer Kit (right). Image credit: https://elinux.org [31].

Four Logitech C930e webcams were chosen as the cameras to provide image and video input. Features of this particular camera [32] that led to it being chosen for the image processing system were:

- HD 1080p video quality at 30 frames-per-second
- 90° field of view
- H.264 UVC 1.5 with Scalable Video Coding

High resolution video and images are ideal for video processing and analysis. Orienting the four cameras as a front-facing camera, rear-facing camera, left-facing camera, and right-facing camera allowed for a total 360°field of view on the SAR-UGV. Utilizing the option

of H.264 encoding that would be performed on the camera hardware itself would result in less bandwidth required to send video feed data to the target hardware, thus preserving processing resources on the Jetson TX2. An image of the Logitech C930e webcam is shown in Figure 4.2.

Figure 4.2: The Logitech C930e webcam. Image credit: Logitech.com [33].

To connect all four Logitech C930e cameras to the Jetson TX2 Development Kit, the HighPoint RocketU 1344A PCI-Express 3.0 [34] was selected. Each of the four ports on this PCIe card features dedicated 10 Gbps of bandwidth. However, the product datasheet shows a transfer rate of 20 Gbps [35]. Bandwidth required by full 1080p HD video, or 1920×1080 pixel resolution, at 30 fps as specified by the Logitech C930e webcam, without video compression, is 1.49 Gbps. Using the lower of the two transfer rates provided by HighPoint (10 Gbps), uncompressed video streaming at 1080p 30 fps still offers approximately an additional 8.5 Gbps of unused capacity on each USB port. In case video transmission at a higher resolution or faster frame rate was to be required on the UGV, an ample amount of bandwidth overhead would still be available for each video stream.

The calculated bit rate does not account for H.264 compression that could optionally be applied on the camera. Applying video compression would reduce the amount of bandwidth required for transmission, the amount of which depending on the type of compression profile activated by the camera. Figure 4.3 displays an image of the HighPoint RocketU 1344A. The formula used for uncompressed video bandwidth calculation is shown in 4.1.

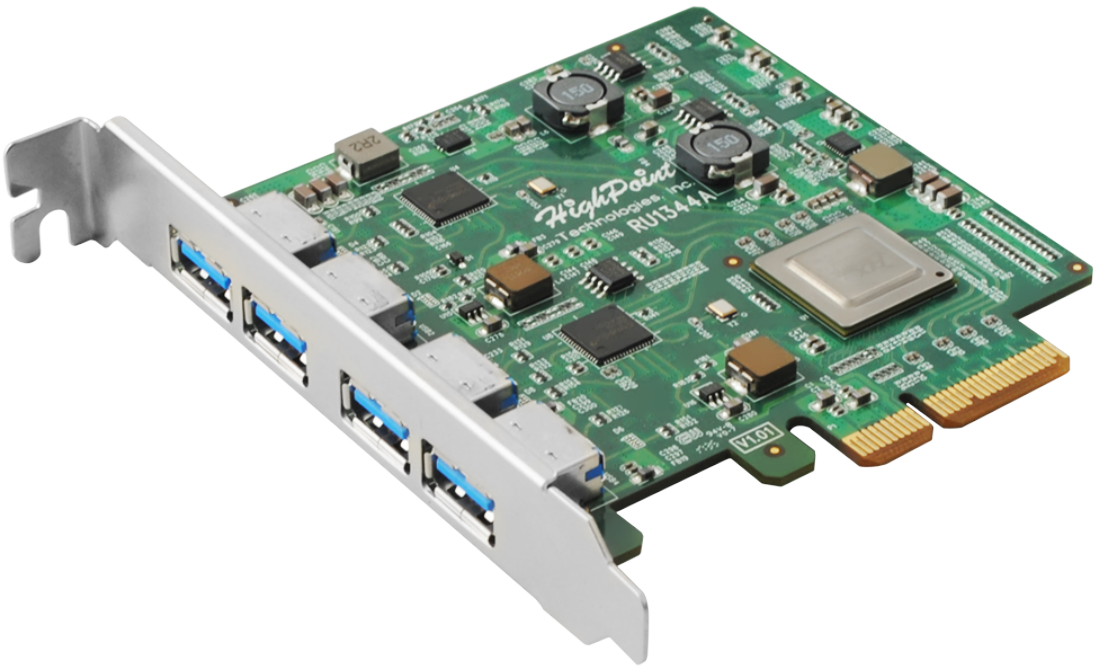


Figure 4.3: The HighPoint RocketU 1344A PCI-Express 3.0 card. Image credit: Highpoint-tech.com [34].

$$
\begin{aligned}
\textit{Uncompressed Bit Rate} &= (\textit{Pixel Resolution})(\textit{Color Depth})(\textit{Frame Rate}) \\
&= \left(\frac{(1920 \times 1080)\,\text{pixels}}{\text{frame}}\right)\left(\frac{8\,\text{bits}}{\text{pixel} \times \text{channel}}\ 3\,\text{channels}\right)\left(\frac{30\,\text{frames}}{\text{second}}\right) \\
&= \frac{1\,492\,992\,000\,\text{bits}}{\text{second}} \\
&\approx 1.49\,\text{Gbps}
\end{aligned}
\tag{4.1}
$$

The Samsung 860 EVO 500 GB 2.5 Inch SATA III Internal SSD was selected to store images and video. This device was required because the relatively small amount of storage device offered by the Jetson TX2's 32 GB eMMC drive. While the stock eMMC drive would be sufficient for storing the operating system and other required software files, additional storage space would be necessary to contain the image and video files that would be amassed during data acquisition. In addition to the simplicity of connecting this SSD to the SATA port on the Jetson TX2 Developer Kit, the Samsung 860 EVO 500 GB also offers 550 MB/s sequential read and 520 MB/s sequential write speeds [36]. Figure 4.4 shows the front and back side views of the an image of the Samsung 860 EVO 500 GB 2.5 Inch SATA III Internal SSD.

Figure 4.4: The Samsung 860 EVO 500 GB 2.5 Inch SATA III Internal SSD. Image credit: Samsung.com [37].

To connect the Samsung 860 EVO 500 GB 2.5 Inch SATA III Internal SSD to the Jetson TX2 Developer kit, a male-to-female SATA cable was required. The "SMAKN(TM) 22-pin (7+15) Sata Male to Female Data and Power Combo Extension Cable - Slimline Sata Extension Cable M/f - 20inch (50cm)" met this requirement. No datasheet could be found for this product. However, its features were listed at Amazon.com as follows [38]:

- 22-Pin SATA Data and Power Combo Extension Cable.

- Hassle-free, Great for backplane adapter. Great for a SATA Internal to External extension Usage.
- Cable Length: 20inch (50cm)
- Connector Details: 7+15 22-pin SATA male, 7+15 22-pin SATA female.

These features met the compatability of the SATA ports on both the Samsung 860 EVO 500 GB 2.5 Inch SATA III Internal SSD to the Jetson TX2 Developer kit. The cable length was sufficient for the two devices to be housed in the same enclosure. A graphic of the of the male-to-female SATA cable is shown in Figure 4.5.

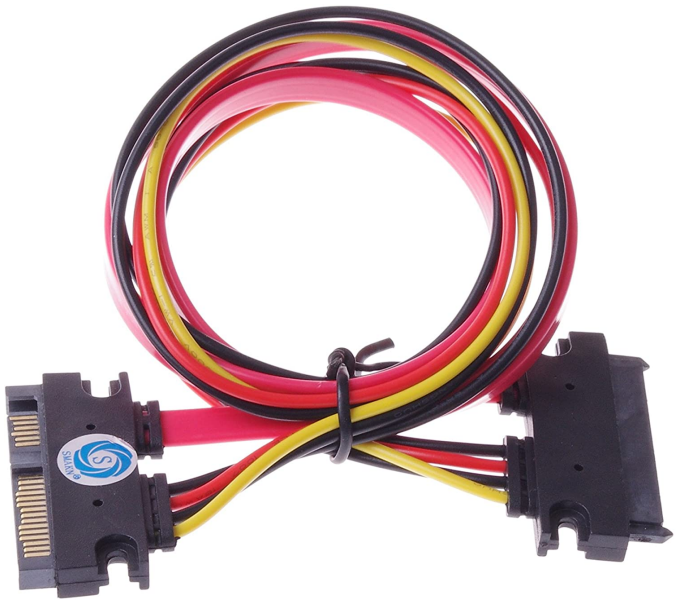


Figure 4.5: The SATA cable used to connect the Jetson TX2 Developer Kit to the Samsun 860 EVO 500 GB SSD. Image credit: Amazon.com [38].

The antennas packaged with the Jetson TX2 Developer Kit are constructed of rigid plastic. Being that the system hardware needed to be placed in an enclosure prior to mounting it on the SAR-UGV, additional modifications needed to be made to maintain a remote wireless connection to the image processing system while making the most of the limited space within the enclosure. The flexible "Bingfu WiFi Antenna Extension Cable (2-Pack) RP-SMA Male to RP-SMA Female Bulkhead Mount RG316 Ca(b)le 30cm 12 inch for WiFi Router Security

IP Camera Wireless Mini PCI Express PCIE Network Card Adapter" made it possible to mount the existing antennas within the enclosure. Like the SATA cable, no datasheet could be found for the RP-SMA antenna extension cables, but the product page on Amazon.com offered the following specifications [39]:

- Compatible with: WiFi USB adapter wireless network router hotspot signal booster repeater range extender mini PCIe card IP camera.
- Connector: reverse polarity SMA female 50 Ohm connector.
- Connector: reverse polarity SMA male 50 Ohm connector.
- Cable: RG316 50 Ohm coaxial cable.
- Length: 1 foot / 12 inch / 30cm.
- Impedance: 50 Ohm.
- Package list: 2 x cable.

Figure 4.6 displays an image of the RP-SMA antenna extension cables used to connect to the Jetson TX2 Developer Kit.

Figure 4.6: RP-SMA antenna extension cables. Image credit: Amazon.com [39].

In order to mitigate high temperatures within the enclosure and prevent electronic equipment from failing, fans needed to be installed. It was decided to install both inlet and outlet fans to maintain cool air passage through the enclosure. At the air inlet location, the "Noctua NF-A4x20 5V PWM" was installed, while at the air outlet location, the "MakerFocus PI-FAN DC Brushless Fan" was installed. Both fans were small enough to mount inside the enclosure box with minimal modifications beyond drilling holes for fasteners and air flow . Full specifications for each fan can be found at their respective Amazon.com pages [40] [41]. Figure 4.7 shows images of both fans.

Figure 4.7: The Noctua NF-A4x20 5V PWM fan (left). The MakerFocus PI-FAN DC Brushless Fan (right). Image credits: Amazon.com [40], Amazon.com [41].

The enclosure selected to contain all of the hardware components for the image processing system was the Altelix NP141105GP NEMA Enclosure. According to the product datasheet, it is "ideal for protecting equipment from harsh environments and tampering [42]." It's 14" × 11" × 5" exterior dimensions and 12" × 8" × 4" interior dimensions met the specifications for mounting to the SAR-UGV in both indoor and outdoor environments, and contained enough inner volume for housing all of the necessary hardware components. The NP141105's aluminum plate also allowed for secure installation of the electronic equipment. Figure 4.8 shows an image of the enclosure.

Figure 4.8: Altelix NP141105GP NEMA Enclosure. Image credits: Altelix.com [43].

With the majority of the electronic components being assembled within the Altelix enclosure, the cameras had to be mounted on the SAR-UGV in such a manner that allowed a 360° view around the vehicle. During the early stages of the the SAR-UGV chassis development, it had been discussed that some sensors and peripherals could be mounted on top of the SAR-UGV using a central mast. For the image processing system, this had the advantage of placing all four cameras close enough together to achieve a combined horizontal 360° field of view. The central mast system, however, also carried disadvantages, largely due to the height that the cameras would need to be mounted above the SAR-UGV's center of mass. This undesired height would have likely resulted in smaller obstacles being undetected by an object detection or image segmentation model when located close to the wheels of the vehicle, as these objects would be located below the cameras' vertical fields of view. An image of one of the initial 3D model drafts for the camera housing appears in Figure 4.9.

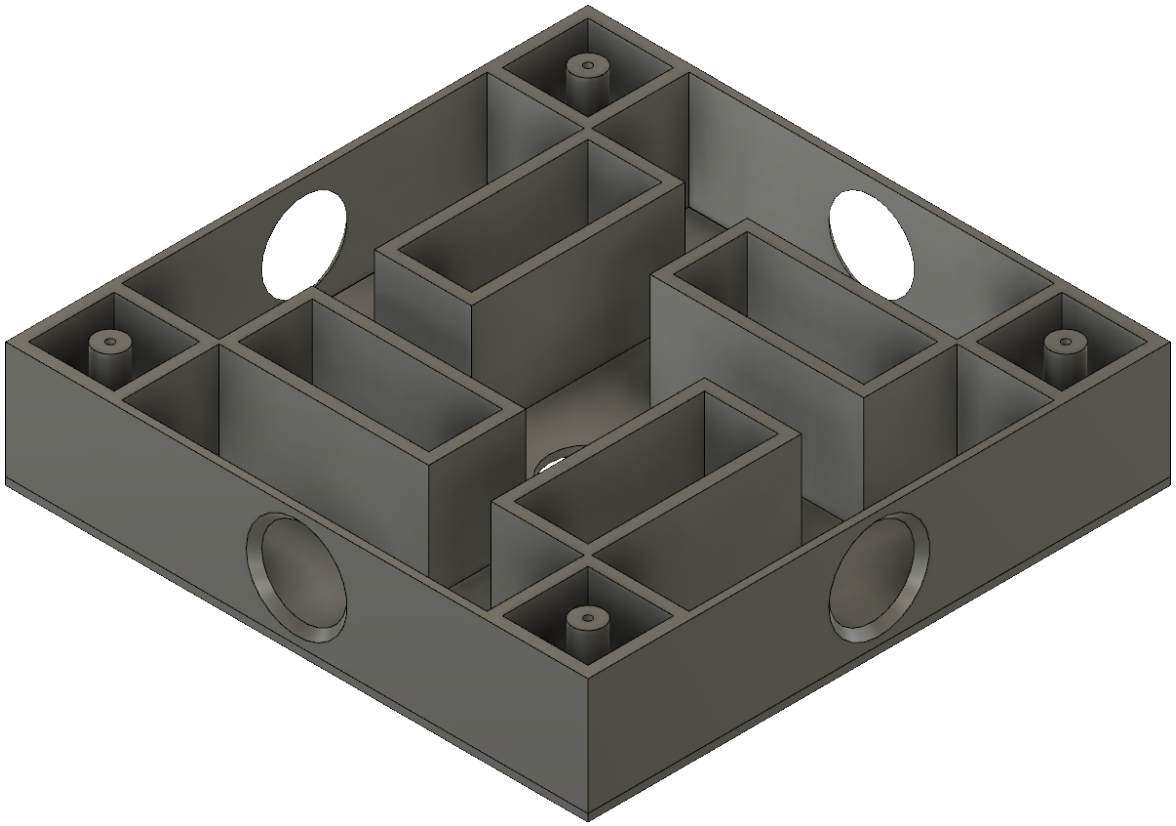


Figure 4.9: 3D model of the central mast-mounted camera mount. This version of the camera mount did not appear on the finalized SAR-UGV chassis.

During later development stages of the SAR-UGV chassis, it became clear that the central mast mounting system was not practical. In addition to the disadvantage of its inability to detect small objects that could move below each camera's vertical field of view, the image processing system would also potentially encounter visual occlusion of target objects due to other mounted subsystem hardware required on the SAR-UGV that could exist within any of the cameras' fields of view. To maintain a clear line of sight from each of the cameras that would not be obstructed by the robot or any of its components, the decision was made to mount each of the four cameras in its own housing near the exterior of the SAR-UGV chassis. This made it possible to mount each camera lower on the vehicle, which would help in the identification of smaller objects when located close to the SAR-UGV. Analysis of images from the edge-mounted cameras also made it easier to distinguishing when an

object was close enough to be in contact with the vehicle, in which the image processing system could aid in the prevention of an unwanted collision. While the updated location of the cameras offered considerable advantages, its most significant trade-off was losing the full horizontal 360° field of view. Hence, small blind spots resulted due to the increased distance between the front, left, back, and right-facing cameras. Figure 4.10 shows images of the final camera housing 3D model design, in which four separate camera housings were mounted to the SAR-UGV to accommodate all of the cameras.

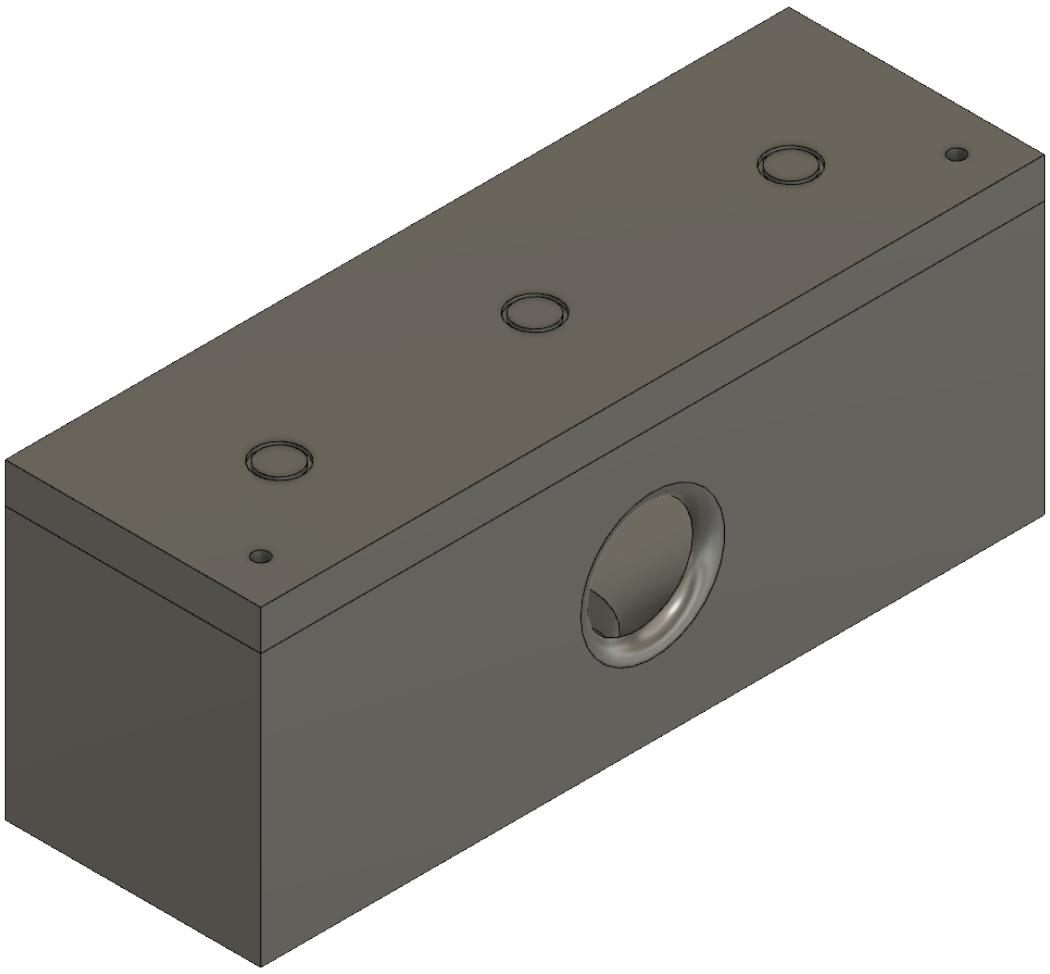


Figure 4.10: 3D model of the final camera mount. This version of the camera mount contained only a single camera, in which one camera was placed on each of the front-, back-, left-, and right-facing sides of the SAR-UGV exterior.

4.1.2 Software

The Jetson TX2 was flashed with the NVIDIA JetPack SDK 4.3.1, which was the most updated version of JetPack at the time the project was started. According to NVIDIA, "JetPack includes OS images, libraries and APIs, developer tools, samples, and documentation [44]." It includes L4T 32.3.1, which is the software distribution package that includes the "board support package for Jetson. It includes Linux Kernel 4.9, bootloader, NVIDIA drivers, flashing utilities, sample filesystem based on Ubuntu 18.04, and more for the Jetson platform [45]." JetPack also includes additional software for support and development of TensorRT, cuDNN, CUDA, multimedia APIs, computer vision, and other developer tools [44].

With the JetPack software installed, and the OS configured, ROS Melodic was the next high-priority software to be installed on the Jetson TX2. As previously discussed in Section 3.3, ROS Melodic was required by all members of the MSE Embedded team, as this version of ROS was intended to enable communication between all subsystems on the SAR-UGV via ROS nodes. As the image processing system evolved, ROS nodes were developed to perform supplementary tasks.

Additional software packages were utilized during the process of design, development, and implementation of the image processing system. However, none of these were as integral as JetPack and ROS. Aside from the software that was developed as part of the image processing system, additional software that was used during this process will be mentioned throughout this thesis as they become relevant, but their features will not be discussed in great detail. The remainder of this section explains the strategy to design and develop the software for the image processing system.

The initial software development design strategy for the image processing system was to write a program in C++ that captured video from all four Logitech C930e cameras simultaneously. Being that it was the first development effort for this project, the new application

was called `cameratest`. The `cameratest` program used the Boost library for C++ to acquire the system time on the Jetson TX2 and output it to each captured video frame. The OpenCV library performed the tasks of capturing, resizing, and tiling all four video frames in a single output image. Additionally, OpenCV performed the functions to allow layering text on each of the tiled frames containing information regarding the specific camera source (e.g., 'FRONT,' 'BACK,' 'LEFT,' and 'RIGHT,'), as well as Jetson TX2 system time text that was acquired by the Boost library. Listing A.1 contains the `cameratest.cpp` source code. Listing A.2 contains source code for the corresponding `CMakeLists.txt` file used to build the `cameratest` application with `cmake`. Figure 4.11 displays sample output from the `cameratest` video capture system.

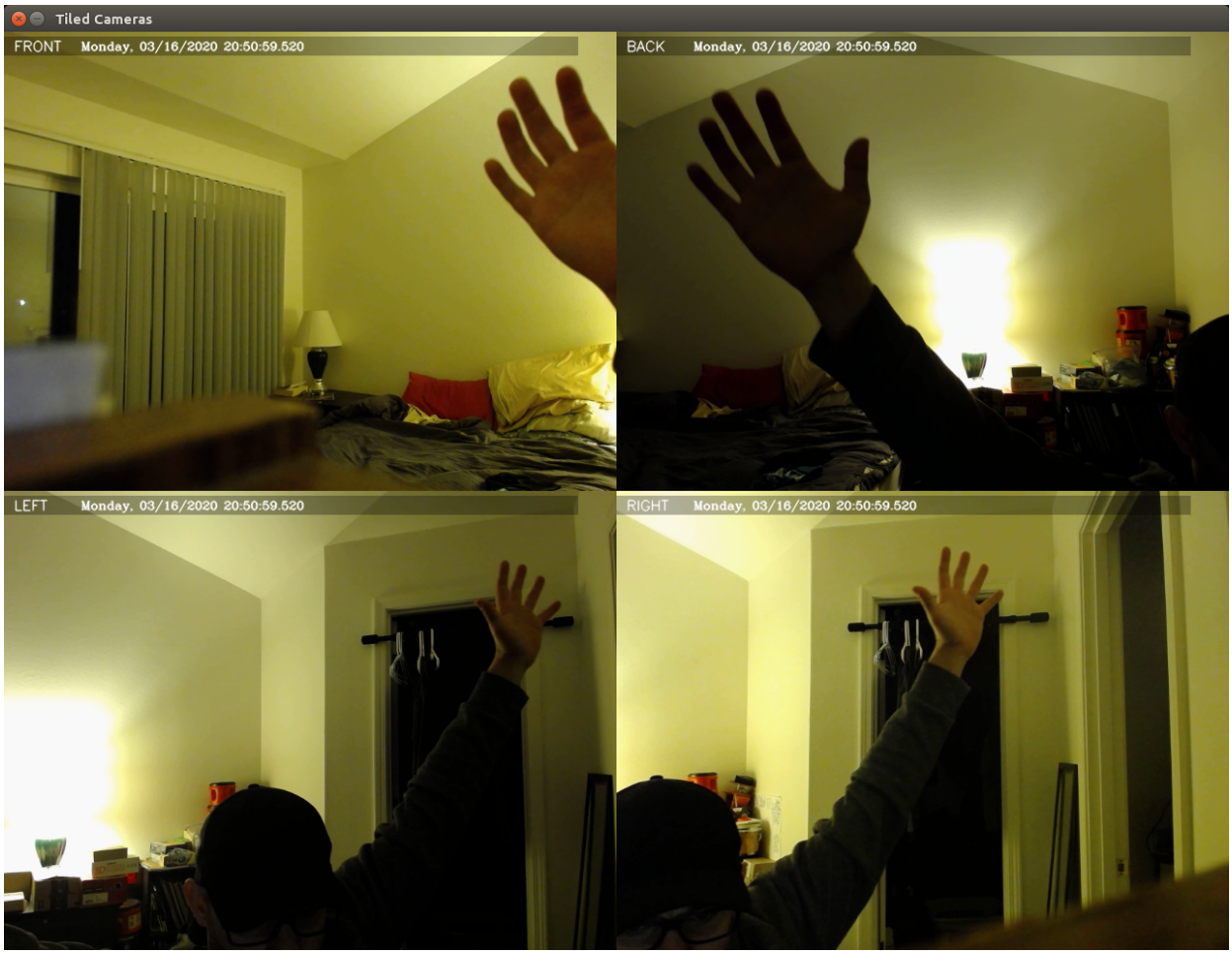


Figure 4.11: Output of the `cameratest` application featured tiled output of simultaneous camera frame captures, as well as an overlaid timestamp on each frame.

Expanding upon the `cameratest` application, the source code was enhanced to write captured video frames to `.avi` video files. This modified application was named `recordtest`. As a result of running the `recordtest` application, four distinct video files could be written simultaneously, one file for each of the four running cameras designed for the SAR-UGV. The `recordtest` application did not include tiled output or timestamps written to each frame like its predecessor. Instead, The Boost library was implemented for the purpose of writing the Jetson TX2 system time as part of each file name that was produced as a result of running the `recordtest` application. The tasks of capturing each video frame, outputting to the screen, and saving the video file were performed by the OpenCV library. A screenshot taken while running `recordtest` can be seen in Figure 4.12. Source code for `recordtest.cpp` appears in Listing A.3. The `CMakeLists.txt` file required to compile the `recordtest` application appears in Listing A.4.

Figure 4.12: Desktop screenshot while running the `recordtest` application on the Jetson TX2. Output from each camera is displayed in a distinct window. While running the `recordtest` application, all video frames captured were saved to a video file on disk containing camera source, time, and date information.

Further exploration continued for effective image processing software systems using the Jetson platform that incorporated image classification, object detection, image segmentation, and ROS integration. Through this exploration, the `cameratest` and `recordtest` systems were abandoned after a GitHub repository, principally authored by NVIDIA developer, Dustin Franklin, was discovered. This particular GitHub repository, called *jetson-inference* [46], was found to contain fully functional and configurable programs built in C++ and Python for image classification, object segmentation, and image segmentation. Furthermore, these programs had all been extended to work with ROS. Source code for these programs was also available, making it possible to modify and recompile them for specific use cases. All software design and development for the SAR-UGV image processing system would henceforth commence through the software architecture found at jetson-inference. A screenshot of the main page for the jetson-inference GitHub repository is shown in Figure 4.13.

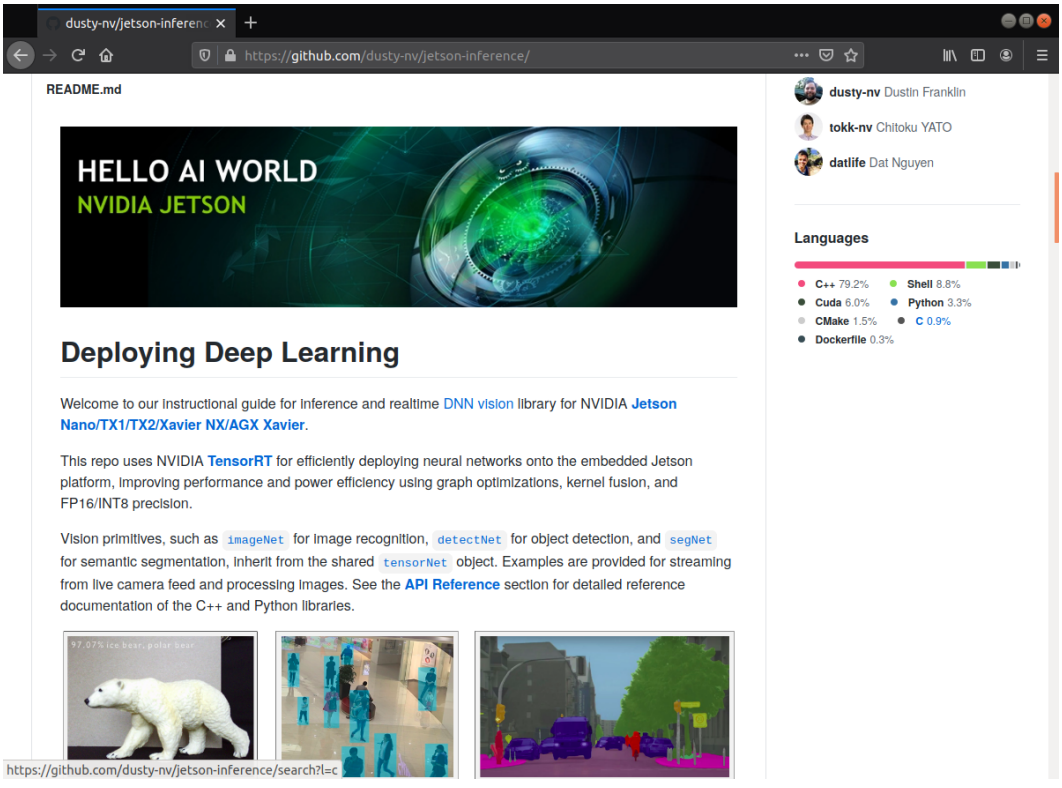


Figure 4.13: The jetson-inference main page on GitHub [46].

4.2 Implementation

4.2.1 Hardware

Hardware implementation of the image processing system manifested itself by assembling the parts described in subsection 4.1.1. Figure 4.14 shows the complete electronics assembly for the image processing system with the enclosure door open. At the time the photograph was taken in Figure 4.14, the image processing system was actively capturing and recording video, as indicated by the illuminated white LEDs on each of the Logitech C930e cameras.

Figure 4.14: The completed hardware assembly for the image processing system.

The enclosure was mounted on the SAR-UGV using a velcro strap. Velcro straps were ideal for the rack system on the vehicle, as they allowed for quick and secure attachment and removal of subsystem enclosures. This feature affirmed its benefit to students' projects every time troubleshooting was required on any subsystem, or when a subsystem needed to be taken home for further development. In addition to his many contributions to the MSE Embedded team, the velcro straps were provided by Nathan Eggleston. Figure 4.15 shows multiple project subsystems mounted on the SAR-UGV. The enclosures used for each students' project were the same Altelix brand. In Figure 4.15, the image processing system enclosure, while partially obscured by other hardware, can be identified by its green color.

Figure 4.15: The SAR-UGV field tested with multiple subsystems mounted in Altelix enclosures.

The cameras for the image processing system were mounted on the rack system that was constructed for the vehicle. The compact housing of the camera mounts shown previously in Figure 4.10 allowed the possibility for the cameras to be mounted nearly anywhere on the robot's rack system rails. This proved to be essential when other subsystems were mounted

on the SAR-UGV, causing available space to become limited for mounting new hardware. Being that the custom 3D-printed camera mounts were made from PETG material, they were easily modified by drilling holes and installing with T-nuts that attached to the rails on the SAR-UGV. Figure 4.16 shows a close-up photograph of the left-facing camera mounted on the SAR-UGV rack system.

Figure 4.16: The left-facing camera of the image processing system mounted on the SAR-UGV.

4.2.2 Software

While the acquisition and assembly of hardware, including the custom manufacture thereof, was a calculated process to ensure functional equipment for the SAR-UGV subsystem, the software involved was required to accomplish multiple tasks. Therefore, this area of design, development, and testing required the greatest time investment, despite some of the software being predesigned and readily available on the Internet. Software development for the image processing system was performed with the motivation to accomplish the following tasks: image capture, image classification, object detection, and image segmentation.

Image Capture

Images were captured using multiple different methods throughout the course of the project. Aside from cases of real-time image capture, obtaining image data was always performed with the intent of outputting video files as opposed to still image files. However, as the project evolved, as did the method of video recording.

Recording video to disk was previously accomplished using the `recordtest` application introduced in Section 4.1.2. It was found that this application was susceptible to crashing when recording at 1080p resolution. This crashing susceptibility increased when multiple cameras were activated simultaneously. To make matters worse, recording at 1080p resulted in a maximum capture frame rate of five frames per second from each camera, a mere 16.67% of the intended 30 fps. The slow frame rate was due to limitations in the v4l2 driver when capturing video from the USB-connected cameras. Figure 4.17 shows Terminal output of running the `recordtest` application at 1080p that resulted in failure.

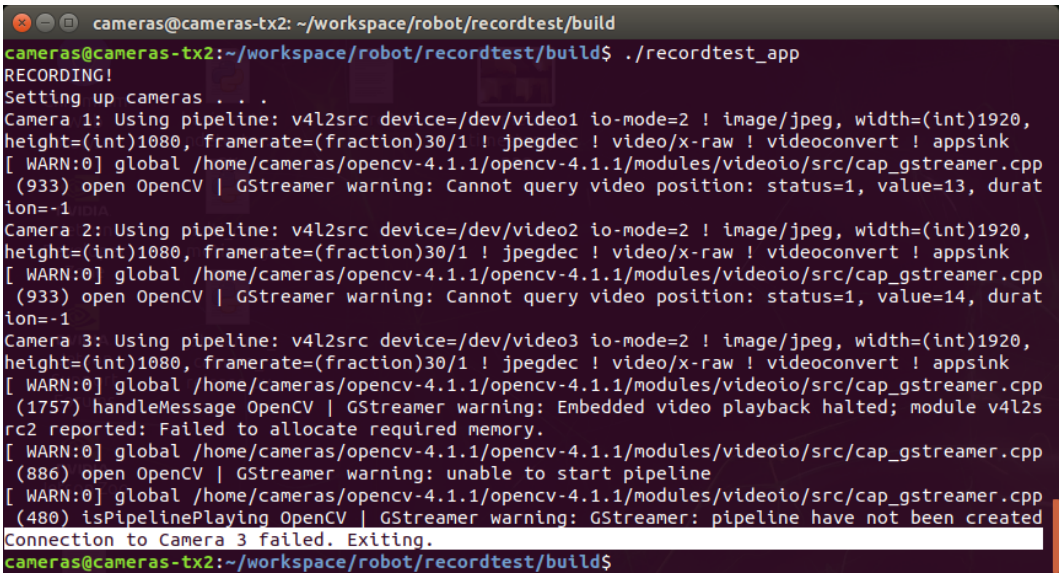


Figure 4.17: Terminal output of an attempt to run the `recordtest` application at 1080p. Memory allocation errors resulted in the application crashing.

The recordtest application was found to run without crashing when only a single camera was activated, albeit with a less than desirable frame rate of 5 fps. This was accomplished by setting up each camera to be activated through a macro defined by the #define preprocessor directive. Listing 4.1 displays the section of source code in recordtest.cpp where only the first camera connected to the image processing system is activated during application run time.

```
#define USE_CAMERA_01   1
// #define USE_CAMERA_02   1
// #define USE_CAMERA_03   1
// #define USE_CAMERA_04   1
```

Listing 4.1: Snippet from recordtest.cpp using #define preprocessor directives to activate a single camera.

The recordtest.cpp source code was saved and compiled multiple times, each time by defining a different single camera to be activated through its corresponding #define preprocessor directive, with the lines referencing the remainder cameras commented out. This produced multiple application files, each of which controlled a different camera, that needed to operate concurrently. A shell script, startcameras.sh was then developed to launch all four applications simultaneously. Listing 4.2 shows the startcameras.sh source code.

```
#!/bin/sh

echo "Running Cameras -- Run stopcameras.sh to stop"
./recordtest_app_1080p_camera01 &
./recordtest_app_1080p_camera02 &
./recordtest_app_1080p_camera03 &
./recordtest_app_1080p_camera04 &
exit 0
```

Listing 4.2: The bash script startcameras.sh used to launch multiple applications in order to run four cameras simultaneously at 1080p.

Running startcameras.sh with all cameras capturing at 1080p, 30 fps consistently resulted in one or more individual camera programs crashing. This meant that while some cameras would stop operating, others would continue to capture and save video data to file. Figure 4.18 displays Terminal output after running the shell script, which attempted to ac-

tivate all four cameras simultaneously. In the figure, it can be seen in the highlighted text that the applications for cameras 2 and 3 failed, while the applications for cameras 1 and 4 continued to run successfully. This was an improvement over the initial `recordtest` application that resulted in total failure at 1080p, but was not suitable to meet the requirements of the image processing system for the SAR-UGV.

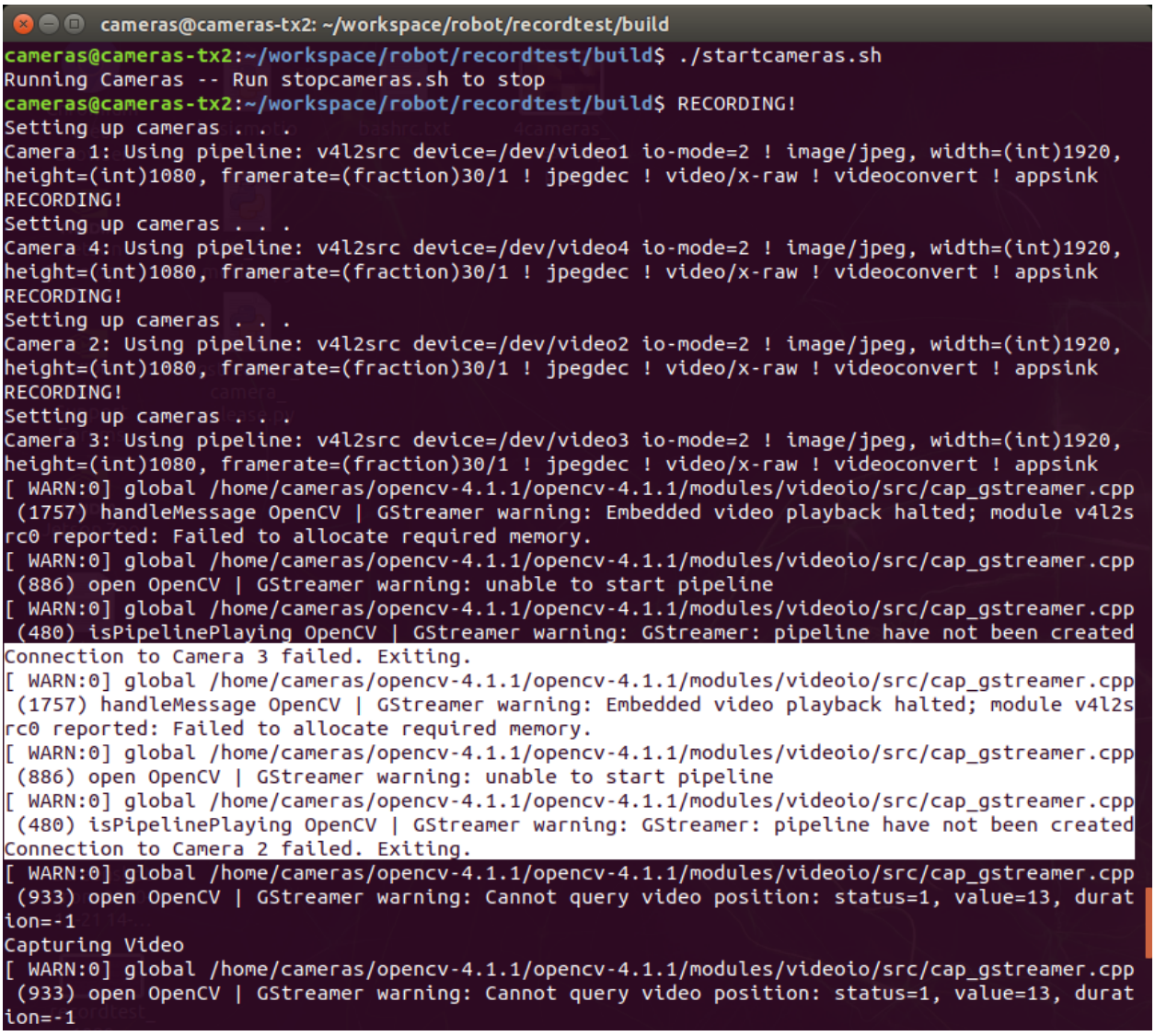


Figure 4.18: Terminal output while running `startcameras.sh`. Two of the four 1080p camera applications crashed, while the other two applications continued to operate.

The `startcameras.sh` script exited after launching each instance of the `recordtest` applications. A `while(1)` loop used in the `recordtest.cpp` source code resulted in the

applications running in perpetuity, pending that the application did not crash. In order to stop the applications, a new script, `stopcameras.sh`, was created. The script used a wildcard (*) to kill all application processes containing the name "recordtest_app." This was effective in stopping all of the applications in the event that any individual application crashed, froze, or continued expected operation. Source code for the `stopcameras.sh` script can be seen in Listing 4.3.

```
#!/bin/sh

echo "Stopping Cameras"
killall -9 recordtest_app*
exit 0
```

Listing 4.3: The bash script `stopcameras.sh` stopped all of the simultaneously running camera applications regardless of program crashing or other unexpected results during run time.

In an effort to eliminate application crashes and improve capture frame rate, the source code of `recordtest.cpp` was modified to capture video at 720p, or 1280×720 pixel resolution. Specifically, modifications were performed by adjusting the values of the `width` and `height` variables within the `main()` function of `recordtest.cpp`. The source code after these modifications is shown in Listing 4.4.

```
// Camera width & height
int width =  1280;
int height = 720;
```

Listing 4.4: Code snippet of `recordtest.cpp` after making the changes to the `width` and `height` variables to capture in 720p.

Just as when running the `recordtest` application previously at 1080p, program crashes were also frequent while capturing at 720p, with one or more cameras failing during application run time. Recompling the `recordtest.cpp` source code after utilizing the `#define` preprocessor directives to produce multiple application files for individual camera capture, and running the `startcameras.sh` script to call multiple applications simultaneously that each operated a single camera appeared to alleviate this problem. Application crashing and camera capture failure became nearly nonexistent in the laboratory environment. Listing

4.5 shows the source code of the bash script `startcameras.sh` that launched individual applications to capture 720p video from each camera.

```
#!/bin/sh

echo "Running Cameras -- Run stopcameras.sh to stop"
./recordtest_app_720p_camera01 &
./recordtest_app_720p_camera02 &
./recordtest_app_720p_camera03 &
./recordtest_app_720p_camera04 &
exit 0
```

Listing 4.5: The bash script `startcameras.sh` used to launch multiple applications in order to run four cameras simultaneously at 720p.

During initial field testing with `startcameras.sh` set to run the 720p `recordtest` applications, unexpected program crashing became as frequent as it had been observed previously when recording at 1080p. The frequent application crashes while capturing at 720p in the field were found to be due to environmental changes. In the laboratory, temperature and humidity were well controlled. In the field, however, placing the Jetson TX2 and other electronics in the Altelix enclosure was required in order to mount on the SAR-UGV. Temperature inside the enclosure increased drastically while the cameras attempted to capture video, resulting in application failure. This led to the need to drill holes in the enclosure and install the fans shown in Figure 4.7 for temperature control by means of increased airflow.

Despite the occurrence of application crashes, the maximum frame rate while capturing at 720p was observed to be lower than the intended 30 fps. Like the limitation of 1080p at 5 fps, examination of the v4l2 driver revealed that capturing 720p uncompressed video from the USB camera was limited to a maximum frame rate 10 fps. Listing 4.6 shows abbreviated output of the `v4l2-ctl` command used in the Terminal to obtain possible frame rate capture capability of the Logitech C930e cameras in uncompressed YUYV pixel format.

```
cameras@cameras-tx2:~/$ v4l2-ctl --list-formats-ext -d /dev/video1
ioctl: VIDIOC\_ENUM\_FMT
Index       : 0
```

```
Type        : Video Capture
Pixel Format: 'YUYV'
Name        : YUYV 4:2:2
...
    Size: Discrete 640x360
        Interval: Discrete 0.033s (30.000 fps)
        Interval: Discrete 0.042s (24.000 fps)
        Interval: Discrete 0.050s (20.000 fps)
        Interval: Discrete 0.067s (15.000 fps)
        Interval: Discrete 0.100s (10.000 fps)
        Interval: Discrete 0.133s (7.500 fps)
        Interval: Discrete 0.200s (5.000 fps)
...
    Size: Discrete 1280x720
        Interval: Discrete 0.100s (10.000 fps)
        Interval: Discrete 0.133s (7.500 fps)
        Interval: Discrete 0.200s (5.000 fps)
...
    Size: Discrete 1920x1080
        Interval: Discrete 0.200s (5.000 fps)
...
```

Listing 4.6: Partial Terminal output of the `v4l2-ctl` command, showing uncompressed video capture frame rate capabilities of the connected camera at 1080p, 720p, and 360p.

In addition to frame rate capabilities for 1080p and 720p video, Listing 4.6 above also lists all capable uncompressed video capture frame rate intervals for 360p, or 640×360 pixel resolution. While the v4l2 driver was also found to be capable of capturing 30 fps for larger resolutions, additional testing was performed at 360p. This selection had largely to do with the additional memory overhead required when incorporating the neural network models discussed later in this chapter.

Capturing video at 360p required additional manipulation to the `recordtest.cpp` source code. Specifically, the `width` and `height` parameters were changed to match the intended resolution. These changes are shown in Listing 4.7.

```
// Camera width & height
int width =  640;
int height = 360;
```

Listing 4.7: Code snippet of `recordtest.cpp` after making the changes to the `width` and `height` variables to capture in 360p.

Separate applications, each activating an individual camera at 360p, were created to further improve performance. The applications were launched simultaneously by use of `startcameras.sh` after editing its source to activate the new applications designed to capture and save video at 360p. Listing 4.8 displays the source code of the bash script.

```
#!/bin/sh

echo "Running Cameras -- Run stopcameras.sh to stop"
./recordtest_app_360p_camera01 &
./recordtest_app_360p_camera02 &
./recordtest_app_360p_camera03 &
./recordtest_app_360p_camera04 &
exit 0
```

Listing 4.8: The bash script `startcameras.sh` used to launch multiple applications to run four cameras simultaneously at 360p.

After the development of the `cameratest` and `recordtest` software applications, investigation of deep learning for image classification, object detection, and image segmentation was conducted. During the exploration of the deep learning capabilities of the jetson-inference software, it was discovered that it contained a standalone application for image capture, `video_viewer`. Along with its image capture capability, it was also found that the jetson-inference software featured ROS integration. This forged a new path in the implementation of the image processing system for the SAR-UGV.

The `video_viewer` application could be executed through the use of a ROS launch file with a command in a Terminal window. This involved the `roslaunch` command that specifically called the ROS launch file, `video_viewer.ros1.launch`. Using the `roslaunch` command also allowed for additional parameters specifying the image source and output, as allowed by the arguments contained in the `video_viewer.ros1.launch` file. Listing 4.9 shows an example `roslaunch` command that executed the `video_viewer` application, where the image source was specified as the first Logitech C930e camera, and the output was specified as the HDMI display monitor connected to the Jetson TX2.

```
cameras@cameras-tx2:~/$ roslaunch ros_deep_learning          \
    video_viewer.ros1.launch                                 \
    input:=v4l2:///dev/video1 output:=display://0
```

Listing 4.9: The Terminal command used to launch the `video_viewer.ros1.launch` ROS launch file.

After running the command in Listing 4.9, the first Logitech C930e camera connected to the Jetson TX2 was activated, and the live video capture was output on the monitor in a new window. A screenshot from the Jetson TX2 after running this command appears in Figure 4.19.

Figure 4.19: Screenshot while running the `video_viewer.ros1.launch` ROS launch file on the Jetson TX2.

Video and image files could also be saved using `video_viewer`. Instead of specifying the display monitor as output in the `roslaunch` command, the output parameter could be changed to specify the location and type of file to be saved. Listing 4.10 contains an exam-

ple `roslaunch` command that executes `video_viewer` and outputs captured images to a `.avi` video file. The name of the file produced by the command contains a time stamp from when the video capture was conducted, as well as the camera source. While the process of saving image and video files was conducted during the implementation and experimental phases of the project, it is not a major topic of this thesis, and is not discussed in detail throughout this document.

```
cameras@cameras-tx2:~/$ roslaunch ros_deep_learning             \
    video_viewer.ros1.launch                                    \
    input:=v4l2:///dev/video1                                   \
    output:=file://$(date +%Y%m%d-%H-%M-%S)_camera1.avi
```

Listing 4.10: The Terminal command used to launch the `video_viewer.ros1.launch` ROS launch file to save a video file of captured image data.

It could be seen in Figure 4.19 that without specified input for image capture width and height in Listing 4.9, the default image capture and output was performed at 720p resolution. Due to the v4l2 driver required to operate the USB-connected cameras, this limited the captures to occur at a maximum frame rate of 10 fps, as shown previously by the `v4l2-ctl` command in Listing 4.6. In order to perform live capture or save video files at a higher frame rate, the capture resolution needed to be scaled down to 360p. This was performed by including the `input_width` and `input_height` in the `roslaunch` command to run `video_viewer`. The augmented command to capture video at 360p and 30 fps is contained in Listing 4.11.

```
cameras@cameras-tx2:~/$ roslaunch ros_deep_learning          \
    video_viewer.ros1.launch                                 \
    input_width:=640 input_height:=360                       \
    input:=v4l2:///dev/video1 output:=display://0
```

Listing 4.11: The Terminal command used to launch the `video_viewer.ros1.launch` ROS launch file at 360p resolution.

The decreased resolution and higher frame rate as a result of the `roslaunch` command used in Listing 4.11 could be seen immediately after the command was executed. Figure 4.20 displays a screenshot from the Jetson TX2 after running this particular instance of

video_viewer in the laboratory environment. In the figure, it can be seen that the title bar of the window displaying the live video capture reported the image resolution and frame rate of 640 × 360 pixels and 30.0 frames per second, respectively.

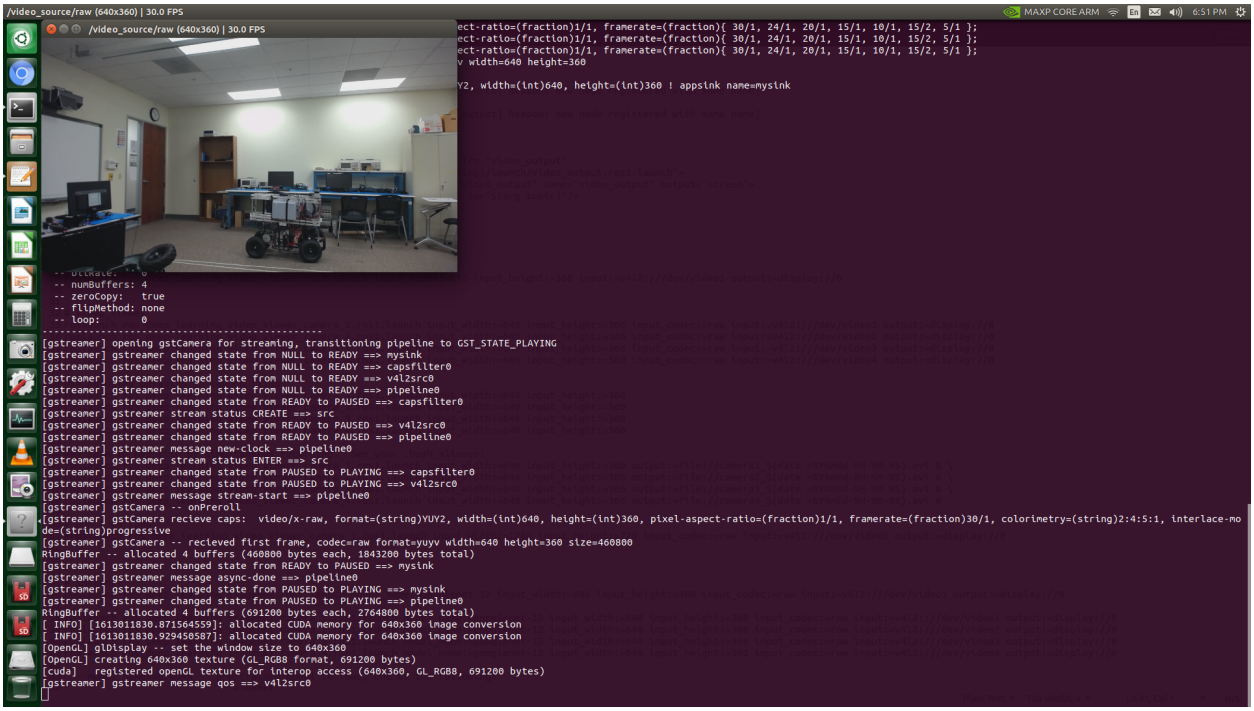


Figure 4.20: Screenshot while running the video_viewer.ros1.launch ROS launch file at 360p on the Jetson TX2.

Running the video_viewer application using the roslaunch command produced new ROS nodes on the network. These nodes could be seen on the ROS Master computer while video_viewer was actively running on the Jetson TX2 machine. Figure 4.21 displays an rqt_graph window after running the $ rqt_graph command from a separate Terminal window on the ROS Master. In the figure, it can be seen that the video_source node published the video_source/raw topic. This topic was then subscribed to by the video_output node. The video_output node then published the rosout topic, which was subscribed to by the rosout node. All data was then published via the rosout_agg ROS topic. It can also be seen that the video_source node also published directly to the rosout topic. While unrelated specifically to image data, the statistics ROS topic was

found to be published to the `rqt_gui_py_node_7035` node, which was also published to the `rosout` topic while the `rqt_graph` interface window was active.

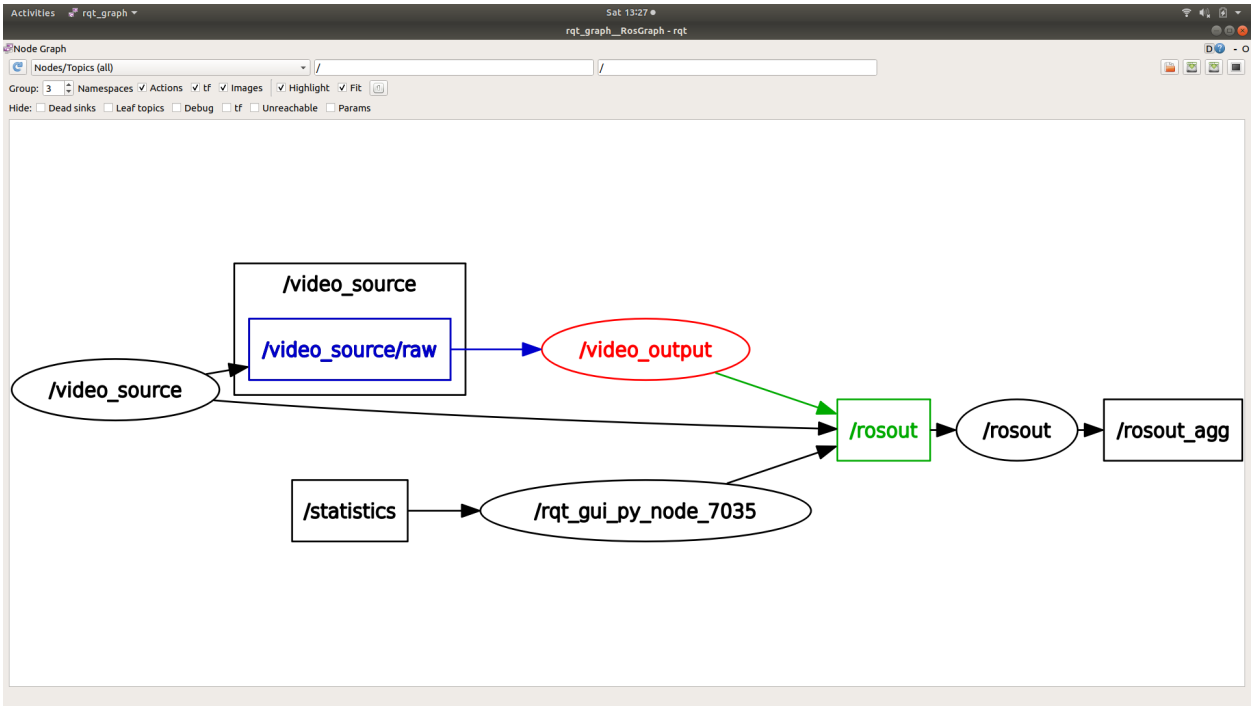


Figure 4.21: ROS `rqt_graph` on the ROS Master computer while running `video_viewer` via the ROS launch file on the Jetson TX2.

In addition to displaying active ROS nodes and topics on the network with `rqt_graph`, these could also be seen directly in the Terminal. Running this command on the ROS Master machine displayed all of the active ROS nodes while `video_viewer` was running. The output of the `$ rosnode list` command appears in Listing 4.12, the output of which includes the same ROS nodes related to image capture and output that were previously introduced: `video_source`, `video_output`, and `rosout`. It is also worth noting that `video_viewer` did not appear in either Figure 4.21 or Listing 4.12. This is due to `video_viewer` not being a ROS node. Instead, the `video_viewer` application, through the use of the `video_viewer.ros1.launch` ROS launch file, only serves as a facilitator for the `video_source` and `video_output` ROS nodes that are created on the network when `video_viewer` is executed.

```
ROSMaster@ROSMaster-Computer:~$ rosnode list
/rosout
/video_output
/video_source
```

Listing 4.12: Active nodes on the ROS network shown after running the command `$ rosnode list` on the ROS Master computer after the `video_viewer` ROS launch file had been executed on the Jetson TX2.

The active ROS topics on the network after running `video_viewer` could be seen in the Terminal by running the command `$ rostopic list`. These included the following topics: `rosout`, `rosout_agg`, `statistics`, and `video_source/raw`. The results of running the command to view all ROS topics active on the network in the Terminal while `video_viewer` was running from a single camera are shown in Listing 4.13.

```
ROSMaster@ROSMaster-Computer:~$ rostopic list
/rosout
/rosout_agg
/statistics
/video_source/raw
```

Listing 4.13: Active topics on the ROS network were revealed by running `$ rostopic list` on the ROS Master machine after the `video_viewer` ROS launch file had been executed on the Jetson TX2.

Data output of each ROS topic could be viewed in the Terminal by using the command `$rostopic echo [ROS-topic-name]`, where `[ROS-topic-name]` could be substituted for any active topic on the network. Running this command revealed data that was output by each individual topic. In the case of the `rosout` topic, data parameters were output for `header`, `seq`, `stamp`, `secs`, `nsecs`, `frame_id`, `level`, `name`, `msg`, `function`, `line`, and `topics`. While details about all of these parameters are not specifically discussed in this thesis, they can be analyzed further to understand each of the working pieces of the `ros_deep_learning` software package from jetson-inference. The results of running the command `$ rostopic echo rosout` appears in Listing 4.14.

```
ROSMaster@ROSMaster-Computer:~$ rostopic echo rosout
header:
seq: 1
stamp:
secs: 1618076149
nsecs: 572984004
frame_id: ''
level: 2
name: "/video_source"
msg: "allocated CUDA memory for 640x360 image conversion"
file: "/home/cameras/catkin_ws/src/ros_deep_learning/src/
   image_converter.cpp"
function: "Resize"
line: 210
topics: [/rosout, /video_source/raw]
---
header:
seq: 2
stamp:
secs: 1618076149
nsecs: 645857263
frame_id: ''
level: 2
name: "/video_output"
msg: "allocated CUDA memory for 640x360 image conversion"
file: "/home/cameras/catkin_ws/src/ros_deep_learning/src/
   image_converter.cpp"
function: "Resize"
line: 210
topics: [/rosout]
---
```

Listing 4.14: Running `$ rostopic echo rosout` on the ROS Master reported the active nodes on the ROS network after launching the `video_viewer` ROS launch file on the Jetson TX2.

The only ROS topic that contained actual captured image data after launching the `video_viewer` ROS launch file was `video_source/raw`. This ROS topic contained RGB pixel image data, which could be used to reconstruct images by any machine on the ROS network that chooses to subscribe to the topic. Aside from the `rosout` topic, `video_viewer/raw` was the only ROS topic that provided any output in the Terminal while `video_viewer` was actively running. As RGB pixel data quickly filled the screen after running the command to view the `video_source/raw` ROS topic on the ROS Mas-

ter computer while `video_viewer` was capturing live video on the Jetson TX2, it would not be reasonable to exhibit all of the data in a single listing, even for a single frame of video captured at 360p, as was displayed previously for the `rosout` topic in Listing 4.14. Partial Terminal output of the `video_source/raw` ROS topic after running the Terminal command `$ rostopic echo video_source/raw` on the ROS Master while `video_viewer` was actively running on the Jetson TX2 machine appears in Figure 4.22.

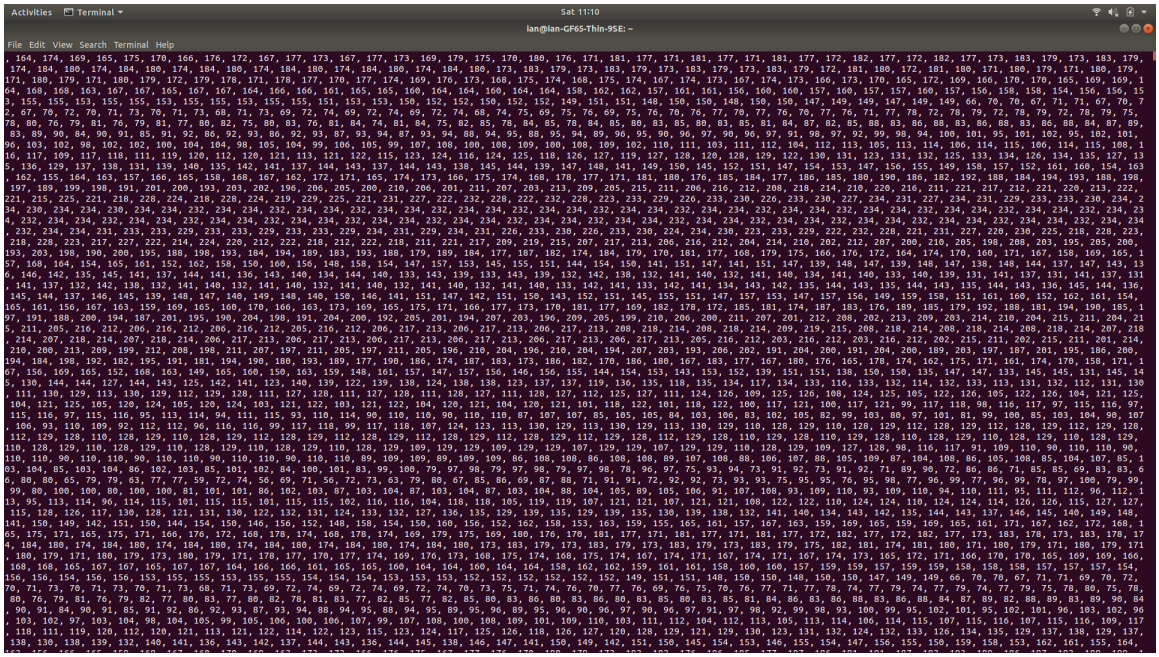


Figure 4.22: Screenshot of the Terminal window while running the command `$ rostopic echo video_source/raw` on the ROS Master computer after activating the `video_viewer` ROS launch file on the Jetson TX2.

The `video_viewer` application execution via the `video_viewer.ros1.launch` ROS launch file was tested successfully with each individual Logitech C930e camera that was connected to the Jetson TX2. However, when attempting to activate multiple cameras simultaneously, failure occurred. For example, when `video_viewer` was capturing from the first Logitech C930e camera connected to the Jetson TX2, the attempt to activate the second Logitech camera camera using a different Terminal window would deactivate the connection to the first camera. Hence, by using the `video_viewer.ros1.launch` file, it was only possible to use one single camera at a time.

Analysis of the XML code contained in `video_viewer.ros1.launch` revealed references to two additional ROS launch files that were called for the capture source and output: `video_source.ros1.launch` and `video_output.ros1.launch`. The source code in `video_viewer.ros1.launch` also contained an additional parameter, or argument name (i.e., `arg name`), for the `video_source/raw` ROS topic to be published. The values for file references and topic names were changed to specify new uniquely titled file names that needed to be created. Additionally, new uniquely named ROS topics had to be initialized, one which referenced each of the camera four cameras intended to be used simultaneously on the SAR-UGV. The newly created and modified source code files are listed as follows:

- `video_viewer_camera_1.ros1.launch`
- `video_viewer_camera_2.ros1.launch`
- `video_viewer_camera_3.ros1.launch`
- `video_viewer_camera_4.ros1.launch`

Within each of these new ROS launch files, the respective new ROS topics contained within them are listed as follows:

- `video_source_camera_1/raw`
- `video_source_camera_2/raw`
- `video_source_camera_3/raw`
- `video_source_camera_4/raw`

Listing A.5, Listing A.6, Listing A.7, and Listing A.8, show the complete XML source code of the newly created and modified ROS launch files that originated from the file included in the jetson-inference `ros_deep_learning` package, `video_viewer.ros1.launch`.

In addition to the creation and modification of ROS launch files for `video_viewer`, similar steps to create new files and modifications specific to the camera source and associated ROS topic name were required based on the XML source code contained in the `video_source.ros1.launch` and `video_output.ros1.launch` ROS launch files. The newly created files required to enable simultaneous capture from four cameras are listed as follows:

- `video_source_camera_1.ros1.launch`
- `video_source_camera_2.ros1.launch`
- `video_source_camera_3.ros1.launch`
- `video_source_camera_4.ros1.launch`
- `video_output_camera_1.ros1.launch`
- `video_output_camera_2.ros1.launch`
- `video_output_camera_3.ros1.launch`
- `video_output_camera_4.ros1.launch`

Within each of these new ROS launch files, an associated new ROS topic for each camera needed to be created. In the `video_source` launch files, specifically, the modification of the ROS node name created for each capture source was required. The new ROS nodes for each of the cameras contained in their respective `video_source` ROS launch files are listed as follows:

- `video_source_camera_1`
- `video_source_camera_2`
- `video_source_camera_3`
- `video_source_camera_4`

Source code for the modified `video_source` ROS launch files appear in Listing A.9, Listing A.10, Listing A.11, and Listing A.12.

Matching associated `video_source` nodes needed to be created within each of their associated `video_output` ROS launch files. The new ROS nodes for each of the cameras contained in their respective `video_output` ROS launch files are listed as follows:

- `video_output_camera_1`
- `video_output_camera_2`
- `video_output_camera_3`
- `video_output_camera_4`

Additionally, within the individual `video_output` ROS launch files for each camera source, the `video_output/image_in` image data was remapped to the associated ROS topic so that it could be output over the ROS network. This required additional source code edits so that image data from each individual camera could be conveyed across the network. The new ROS topic mapping associated with each camera within the `video_output` ROS launch files are as follows:

- `video_output_camera_1/image_in`
- `video_output_camera_2/image_in`
- `video_output_camera_3/image_in`
- `video_output_camera_4/image_in`

Listing A.13, Listing A.14, Listing A.15, and Listing A.16 contain the complete source code within the modified `video_output` ROS launch files.

The creation and modification of ROS launch files for `video_viewer`, `video_source`, and `video_output` made it possible to activate each of the four Logitech C930e cameras

connected to the Jetson TX2 and run them simultaneously. However, this required each camera to be activated individually. In order for the four cameras to be activated at the same time to perform simultaneous video capture, a new command alias was created within the `.bash_aliases` file of the Jetson TX2. All four cameras could then be activated simultaneously by simply entering the command alias, named `rosplay` in a single Terminal window. The complete `rosplay` command alias text is shown in Listing 4.15. Figure 4.23 displays the result of running `rosplay` on the Jetson TX2.

```
alias rosplay='roslaunch ros_deep_learning                          \
    video_viewer_camera_1.ros1.launch                               \
    input_width:=640 input_height:=360 input_codec:=raw             \
    input:=v4l2:///dev/video1 output:=display://0 &                 \
                                                                    \
    roslaunch ros_deep_learning                                     \
    video_viewer_camera_2.ros1.launch                               \
    input_width:=640 input_height:=360 input_codec:=raw             \
    input:=v4l2:///dev/video2 output:=display://0 &                 \
                                                                    \
    roslaunch ros_deep_learning                                     \
    video_viewer_camera_3.ros1.launch                               \
    input_width:=640 input_height:=360 input_codec:=raw             \
    input:=v4l2:///dev/video3 output:=display://0 &                 \
                                                                    \
    roslaunch ros_deep_learning                                     \
    video_viewer_camera_4.ros1.launch                               \
    input_width:=640 input_height:=360 input_codec:=raw             \
    input:=v4l2:///dev/video4 output:=display://0 &'
```

Listing 4.15: The `rosplay` Terminal command alias added to the `.bash_aliases` file to capture images from four cameras simultaneously.

The `rqt_graph` on the ROS Master provided a new graphical display of the ROS nodes and topics active on the network after running the `rosplay` command alias. It was observed that a new `video_source` node was created for each of the cameras that were actively capturing video. The associated `video_source` ROS topics for each camera were published, and consequently subscribed to by their respective `video_output` nodes. All of the ROS topics associated with each of the camera captures were published to the single `rosout` node via the `rosout` topic. The `rqt_graph` produced on the ROS Master machine after running `rosplay` on the Jetson TX2 is displayed in Figure 4.24.

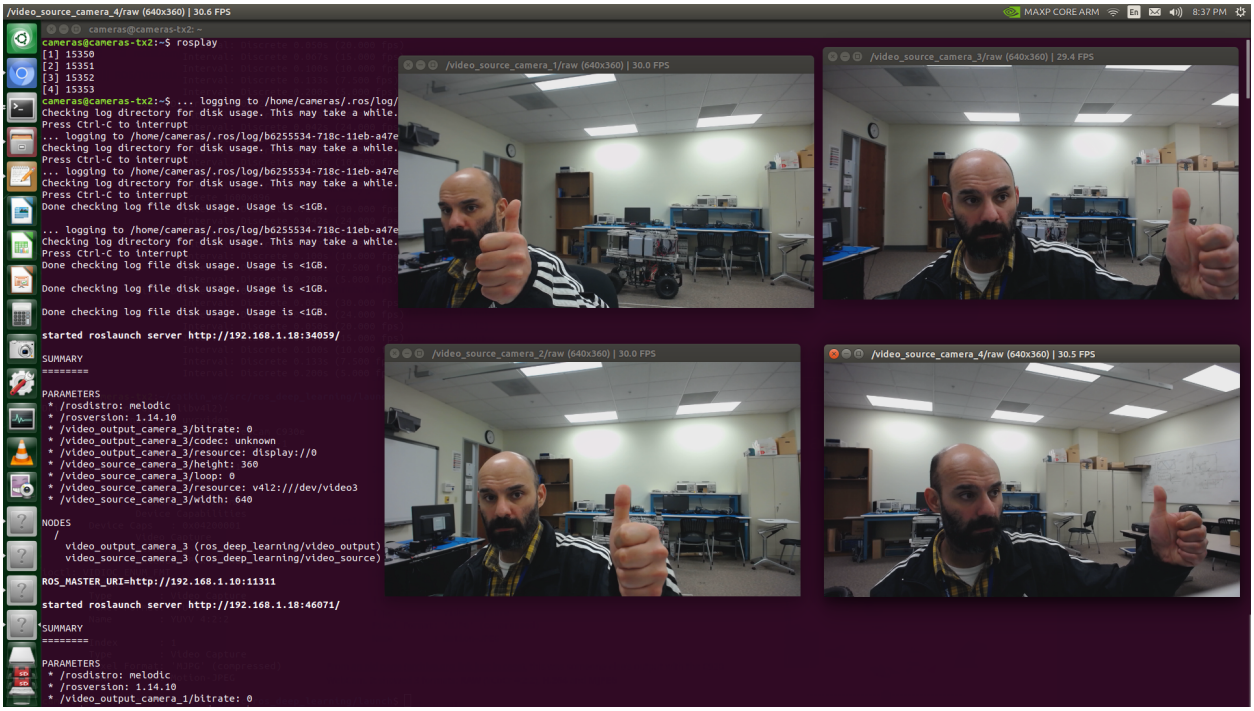


Figure 4.23: Screenshot while running the `rosplay` command alias on the Jetson TX2.

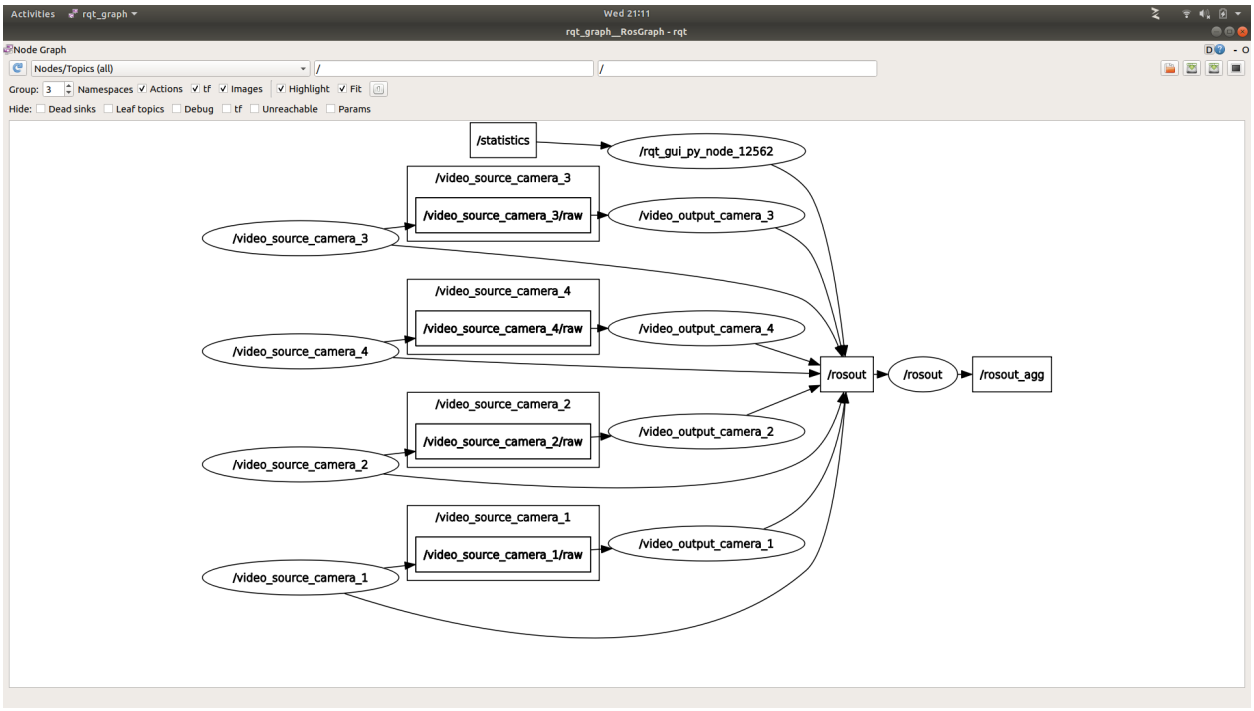


Figure 4.24: ROS `rqt_graph` from the ROS Master computer while running the `rosplay` command alias on the Jetson TX2.

The newly created ROS nodes displayed in the `rqt_graph` were also viewed using the command `$ rosnode list` in the Terminal of the ROS Master computer. The full output of this command after running the `rosplay` command alias on the Jetson TX2 appears in Listing 4.16.

```
ROSMaster@ROSMaster-Computer:~$ rosnode list
/rosout
/video_output_camera_1
/video_output_camera_2
/video_output_camera_3
/video_output_camera_4
/video_source_camera_1
/video_source_camera_2
/video_source_camera_3
/video_source_camera_4
```

Listing 4.16: Active nodes on the ROS network shown after running the command `$ rosnode list` on the ROS Master after the `rosplay` command alias had been executed on the Jetson TX2.

ROS topics on the network after running `rosplay` on the Jetson TX2 were displayed in the Terminal of the ROS Master machine by running the command `$ rostopic list`. These included the newly created `video_source` ROS topics associated with each camera, which contained RGB image data from each individual video capture. The results of running the command to view all ROS topics active on the network from the Terminal on the ROS Master after running the `rosplay` command alias on the Jetson TX2 is contained in Listing 4.17.

```
ROSMaster@ROSMaster-Computer:~$ rostopic list
/rosout
/rosout_agg
/statistics
/video_source_camera_1/raw
/video_source_camera_2/raw
/video_source_camera_3/raw
/video_source_camera_4/raw
```

Listing 4.17: Active topics on the ROS network were revealed by running `$ rostopic list` on the ROS Master machine after the `rosplay` command alias had been executed on the Jetson TX2.

It was found that running the `rosplay` command alias on the Jetson TX2 machine resulted in video capture that would run indefinitely, or until the machine was shut down. In order to terminate the simultaneous capture from the four cameras, a new command alias, `roskill` was created that could be run in a separate Terminal window on the Jetson TX2. The `roskill` command alias ended all of the system processes associated with the ROS nodes created by `rosplay` by running a series of `rosnode kill` commands. For each `rosnode kill` command used in the `roskill` command alias, a specific ROS node associated with video capture, `video_source` or `video_output` was specified. The full text content of the `roskill` command alias is displayed in Listing 4.18.

```
alias roskill='rosnode kill /video_source_camera_1  &    \
               rosnode kill /video_source_camera_2  &    \
               rosnode kill /video_source_camera_3  &    \
               rosnode kill /video_source_camera_4  &    \
               rosnode kill /video_output_camera_1  &    \
               rosnode kill /video_output_camera_2  &    \
               rosnode kill /video_output_camera_3  &    \
               rosnode kill /video_output_camera_4'
```

Listing 4.18: The `roskill` Terminal command alias used to kill ROS nodes created by running the `rosplay` command alias.

Capturing images by using the `video_viewer` application via the ROS launch files opened up new possibilities. Exploiting the capabilities of this software to extend camera capture from multiple simultaneous sources, and the ability to use ROS commands to terminate the processes associated with each individual capture, were shown to be vital in the operability of the image processing system for the SAR-UGV. This latest method of obtaining image data not only produced output that could be displayed on a screen or saved to file through the use of parameters in the command line, but also made it possible to output raw data via ROS topic publishing. The following sections in this chapter further explore the features within the jetson-inference `ros_deep_learning` software for image classification, object detection, and image segmentation.

Image Classification

In addition to the image capture capabilities of the `ros_deep_learning` software package from jetson-inference, it's most compelling features were explored not because of image and video capture, but for its more advanced capabilities of image classification, object detection, and image segmentation. The first of these deep learning capabilities that was explored was image classification with `imagenet`.

Much like `video_viewer`, it was possible to run the `imagenet` application through the use of a `roslaunch` command. This command was initially executed in a Terminal window on the Jetson TX2 machine with minimal parameters. Listing 4.19 shows the full command input to launch `imagenet` as part of the `ros_deep_learning` package, where only the input and output devices of the first camera and HDMI monitor connected to the Jetson TX2, respectively, were specified.

```
cameras@cameras-tx2:~/$ roslaunch ros_deep_learning         \
    imagenetnet.ros1.launch                                 \
    input:=v4l2:///dev/video1 output:=display://0
```

Listing 4.19: The Terminal command used to launch the `imagenet` ROS node with a single camera.

As a result of running the command shown in Listing 4.19 within the Terminal of the Jetson TX2, a new window appeared containing live video capture from the first Logitech C930e camera. Like the default capture window produced by `video_viewer`, the window's title bar reported data on the captured image resolution of 1280×720 pixels and the current frame rate of capture. Due to limitations of the v4l2 driver, the 720p video capture was again limited to a maximum of 10 fps, in which actual live video capture was often found to be lower than the specified driver maximum limitation.

New data points were also featured in the live video capture that resulted from running `imagenet` that were not reported by `video_viewer` alone. These new data points

were the image classification and confidence interval of the classification as reported by the `imagenet` application after the captured frame had been processed. From the screenshot image in Figure 4.25, it can be seen that `imagenet` classified the captured video frame as a "forklift" with a 29.25% confidence interval.

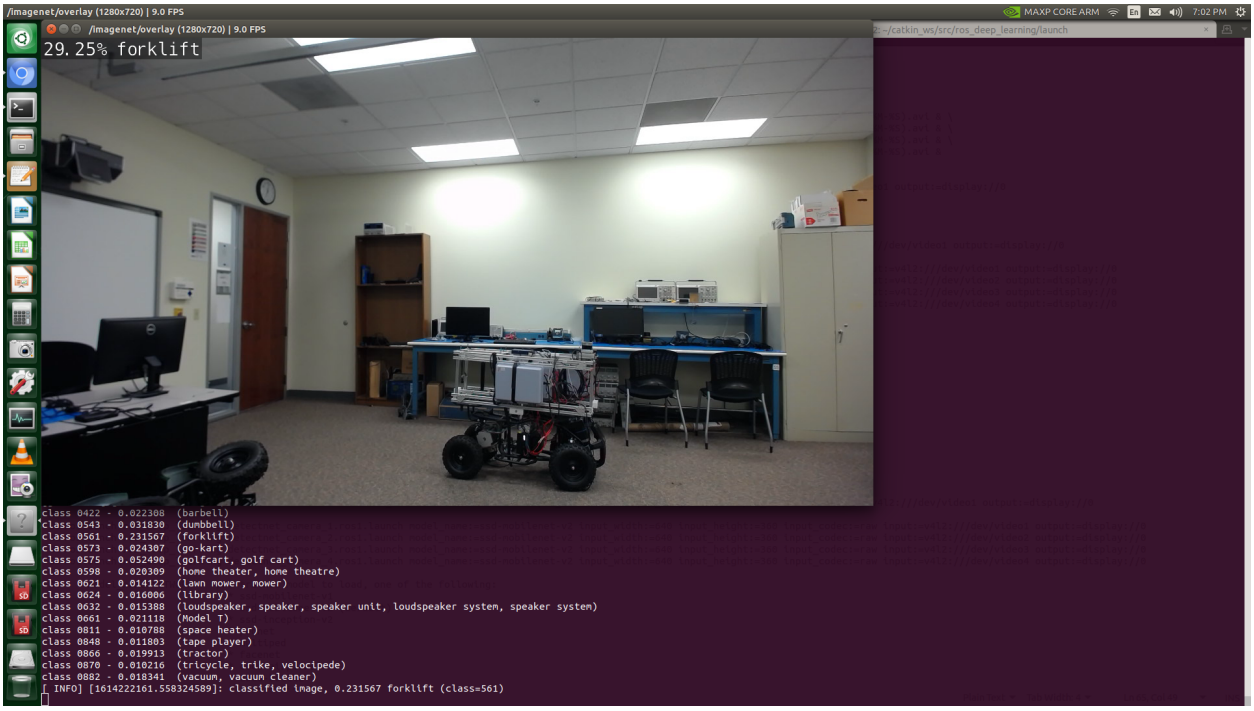


Figure 4.25: Screenshot while running the `imagenet` ROS launch file on the Jetson TX2. Only input and output devices were included as additional command parameters.

The `imagenet` application utilizes an deep neural network model to perform image classification. By accessing the `imagenet` application help, it was found that `GoogleNet` was the model selected by default when the model is unspecified using the `network` parameter, as was the case from the program execution shown previously in Listing 4.19. Listing 4.20 displays partial output of the `imagenet` help, where the `network` parameter could be utilized to specify the image classification model to be used during application run time.

```
cameras@cameras-tx2:~$ imagenet --help
usage: imagenet [--help] [--network=NETWORK] ...
input_URI [output_URI]

```

```
Classify a video/image stream using an image recognition DNN.
See below for additional arguments that may not be shown above.

positional arguments:
    input_URI       resource URI of input stream  (see videoSource
   below)
    output_URI      resource URI of output stream (see videoOutput
   below)

imageNet arguments:
  --network=NETWORK    pre-trained model to load, one of the following:
                           * alexnet
                           * googlenet (default)
                           * googlenet-12
                           * resnet-18
                           * resnet-50
                           * resnet-101
                           * resnet-152
                           * vgg-16
                           * vgg-19
                           * inception-v4
  --model=MODEL        path to custom model to load (caffemodel, uff,
   or onnx)
  --prototxt=PROTOTXT  path to custom prototxt to load (for .caffemodel
    only)
  --labels=LABELS      path to text file containing the labels for each
    class
  --input-blob=INPUT   name of the input layer (default is 'data')
  --output-blob=OUTPUT name of the output layer (default is 'prob')
  --batch-size=BATCH   maximum batch size (default is 1)
  --profile            enable layer profiling in TensorRT
...
```

Listing 4.20: Accessing the `imagenet` help in the Terminal revealed the pre-trained models available after downloading the software from jetson-inference.

From Listing 4.20 above, it can be seen that the following pre-trained deep neural network image classification models were readily available from jetson-inference:

- `AlexNet`
- `GoogleNet`
- `GoogleNet-12`
- `ResNet-18`

- ResNet-50
- ResNet-101
- ResNet-152
- VGG-16
- VGG-19
- Inception-v4

In order to use imagenet as part of the ros_deep_learning package with all four cameras simultaneously, the ROS launch file contained with the jetson-inference software, imagenet.ros1.launch needed to be copied and modified. The modifications required were references to the video_source and video_output ROS launch files associated with each camera on the Jetson TX2 (e.g., video_source_camera_1.ros1.launch and video_output_camera_1.ros1.launch), as well as new imagenet nodes specific to each camera (e.g., imagenet_camera_1). The imagenet ROS node associated with each camera required further remapping from the specific camera's ROS topic imagenet/image_in to its respective video_source/raw ROS topic. Finally, the imagenet/overlay topic for each camera needed to be given a unique value for the associated camera (e.g., imagenet_camera_1/overlay) when providing output to its corresponding video_output ROS launch file. The newly created ROS launch files to allow the use of imagenet from four distinct cameras simultaneously are as follows:

- imagenet_camera_1.ros1.launch
- imagenet_camera_2.ros1.launch
- imagenet_camera_3.ros1.launch
- imagenet_camera_4.ros1.launch

Source code for the modified `imagenet` ROS launch files that allowed simultaneous image classification from multiple cameras appears in Listing A.17, Listing A.18, Listing A.19, and Listing A.20.

While modifying the ROS launch files required to enable `imagenet` with four cameras simultaneously, it was found that the `network` parameter was not contained in the `imagenet` ROS launch files as described in the `imagenet` help text. Where the `imagenet` help specified the `network` parameter for selection of a different image classification model, the `imagenet.ros1.launch` file contained a `model_name` input parameter. Through experimentation, it was found that `imagenet` program execution via the ROS launch files in the command line accepted both `network` and `model_name` parameters as equivalent input. Hence, `imagenet` produced the same results when using the two parameters interchangeably.

In order to run `imagenet` simutaneously from four cameras, a new command alias was created on the Jetson TX2 machine. This new command alias, `rosimagenet`, ran `imagenet` on each of the four connected Logitech C930e cameras through a series of `roslaunch` commands. Each of the `roslaunch` commands within the command alias specified `imagenet` to be run with the same resolution. The `roslaunch` commands each specified the same image classification model, `ResNet-18`, rather than accepting the default `GoogleNet` model. The full text contents of the `rosimagenet` command alias is displayed in Listing 4.21.

```
alias rosimagenet='roslaunch ros_deep_learning                    \
    imagenet_camera_1.ros1.launch model_name:=resnet-18           \
    input_width:=160 input_height:=120 input_codec:=raw           \
    input:=v4l2:///dev/video1 output:=display://0 &               \
                                                                  \
    roslaunch ros_deep_learning                                   \
    imagenet_camera_2.ros1.launch model_name:=resnet-18           \
    input_width:=160 input_height:=120 input_codec:=raw           \
    input:=v4l2:///dev/video2 output:=display://0 &               \
```

```
10                                                                  \
11     roslaunch ros_deep_learning                                  \
12     imagenet_camera_3.ros1.launch model_name:=resnet-18          \
13     input_width:=160 input_height:=120 input_codec:=raw          \
14     input:=v4l2:///dev/video3 output:=display://0 &              \
15                                                                  \
16     roslaunch ros_deep_learning                                  \
17     imagenet_camera_4.ros1.launch model_name:=resnet-18          \
18     input_width:=160 input_height:=120 input_codec:=raw          \
19     input:=v4l2:///dev/video4 output:=display://0 &'
```

Listing 4.21: The `rosimagenet` Terminal command alias added to the `.bash_aliases` file on the Jetson TX2 to perform image classification simultaneously from four cameras.

Running the `rosimagenet` command alias in the Terminal produced four new windows on the monitor, one for each of the connected Logitech cameras. It is important to disucuss the 160×120 resolution used in Listing 4.21. This is the smallest capture size allowable by the Logitech C930e cameras, and it was the only resolution that was capable of running the `rosimagenet` command alias without completely using all of the RAM memory on the Jetson TX2. Running the Ubuntu System Monitor program while also running `rosimagenet` with all four camera captures output to the monitor resulted in complete operating system freeze. However, a screenshot was obtained while running `rosimagenet` with the live video windows output on the screen at 160×120 resolution prior to memory resources being 100% utilized. This screenshot is displayed in Figure 4.26, where a "Low Memory Warning" notification window can be seen obscuring the third and fourth camera output windows.

While the four instances of `imagenet` were active via the `rosimagenet` command alias, their associated ROS nodes and topics could be viewed by using the `$ rosnode list` command, which displayed all active ROS nodes on the network. Listing 4.22 shows the output from running this command on the ROS Master machine. In addition to the `video_source` and `video_output` nodes that were included in the previous section, the `imagenet` ROS nodes for each of the four active Logitech C930e cameras were displayed as part of the command output.

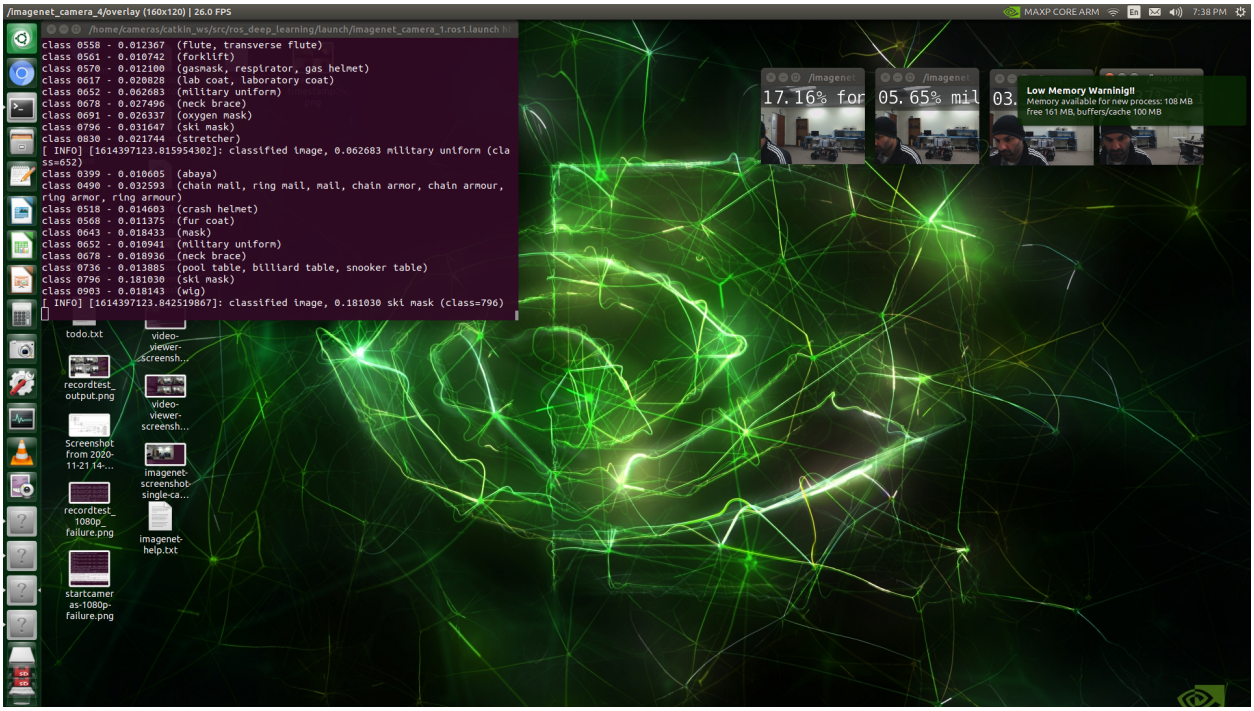


Figure 4.26: Screenshot while running the `rosimagenet` command alias on the Jetson TX2.

```
ROSMaster@ROSMaster-Computer:~$ rosnode list
/cameras_talker
/imagenet_camera_1
/imagenet_camera_2
/imagenet_camera_3
/imagenet_camera_4
/rosout
/video_output_camera_1
/video_output_camera_2
/video_output_camera_3
/video_output_camera_4
/video_source_camera_1
/video_source_camera_2
/video_source_camera_3
/video_source_camera_4
```

Listing 4.22: Active nodes on the ROS network were revealed by running `$ rosnode list` on the ROS Master computer after the `rosimagenet` command alias had been executed on the Jetson TX2.

The `imagenet` ROS node reported three unique topics: `classification`, `overlay`, and `vision_info`. The active topics on the ROS network were reported after running the

`$ rostopic list` command in the Terminal on the ROS Master machine. The output of this command showed that each of the four cameras had its own set of `classification`, `overlay`, and `vision_info` topics. Listing 4.23 displays the full output of running the `$ rostopic list` command on the ROS Master computer while `imagenet` was active for all four cameras.

```
ROSMaster@ROSMaster-Computer:~$ rostopic list
/cameras_chatter
/imagenet_camera_1/classification
/imagenet_camera_1/overlay
/imagenet_camera_1/vision_info
/imagenet_camera_2/classification
/imagenet_camera_2/overlay
/imagenet_camera_2/vision_info
/imagenet_camera_3/classification
/imagenet_camera_3/overlay
/imagenet_camera_3/vision_info
/imagenet_camera_4/classification
/imagenet_camera_4/overlay
/imagenet_camera_4/vision_info
/rosout
/rosout_agg
/video_source_camera_1/raw
/video_source_camera_2/raw
/video_source_camera_3/raw
/video_source_camera_4/raw
```

Listing 4.23: Active topics on the ROS network were revealed by running `$ rostopic list` on the ROS Master computer after the `rosimagenet` command alias had been executed on the Jetson TX2.

The `classification` topic included results of the image classification, including class identification number and confidence probability. In Listing 4.24, the results of running the command `$ rostopic echo /imagenet_camera_1/classification` from the ROS Master computer are displayed. In these particular results, `id: 561` was reported, corresponding to "forklift," by which the list of names for each classification identification number was found in the file `ilsvrc12_synset_words.txt` within the jetson-inference software files. A `score` of 0.7705078125 was also reported by the ROS topic, which meant a confidence probability of approximately 77.05%.

```
ROSMaster@ROSMaster-Computer:~$ rostopic echo /imagenet_camera_1/
   classification
header:
seq: 3409
stamp:
secs: 1614398264
nsecs: 774649322
frame_id: ''
results:
-
id: 561
score: 0.7705078125
source_img:
header:
seq: 0
stamp:
secs: 0
nsecs:          0
frame_id: ''
height: 0
width: 0
encoding: ''
is_bigendian: 0
step: 0
data: []
---
...
```

Listing 4.24: Running `$ rostopic echo /imagenet_camera_1/classification` on the ROS Master computer reported the classification identification number and confidence score from the capture on the first camera connected to the Jetson TX2.

Input image raw data overlaid with classification results were included in the `overlay` topic. Figure 4.27 shows Terminal output on the ROS Master machine after running the command `$ rostopic echo /imagenet_camera_1/overlay`. Its output included image capture data such as `height`, `width`, `encoding`, `is_bigendian` (i.e., endianness), and `data` (i.e., raw pixel color data).

The `vision_info` topic contained metadata related to the image classification that was being performed whe running `imagenet` on the Jetson TX2 machine. Command output of `$ rostopic echo /imagenet_camera_1/vision_info` is displayed in Listing 4.25, which shows the list of output parameters relevant to the classification of the captured

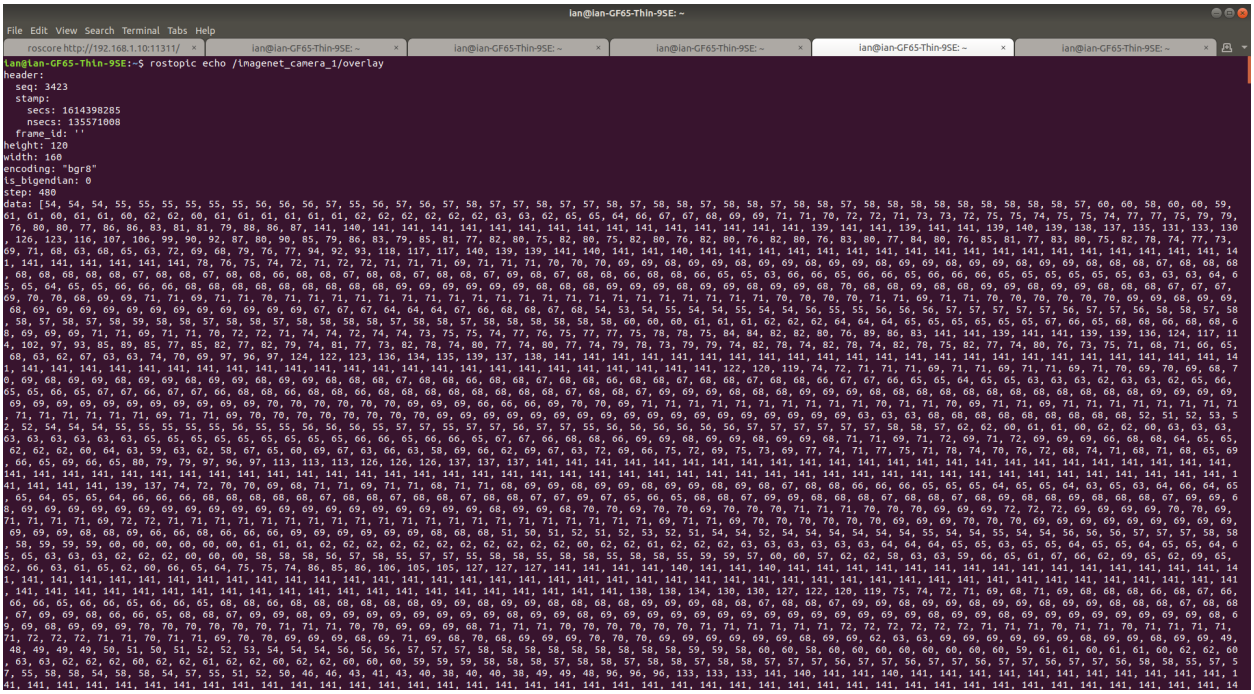


Figure 4.27: Output of the `$ rostopic echo /imagenet_camera_1/overlay` command on the ROS Master computer while running the `rosimagenet` command alias on the Jetson TX2.

image. The `method` parameter reported the file location of the `ResNet-18` model being utilized by `imagenet`, while the `database_location` parameter output the file location of the database that was created to store image classification results.

```
ROSMaster@ROSMaster-Computer:~$ rostopic echo /imagenet_camera_1/
   vision_info
header:
seq: 0
stamp:
secs: 0
nsecs:          0
frame_id: ''
method: "/usr/local/bin/networks/ResNet-18/ResNet-18.caffemodel"
database_location: "/imagenet_camera_1/class_labels_8241445496304598679
   "
database_version: 0
---
```

Listing 4.25: Running `$ rostopic echo /imagenet_camera_1/vision_info` on the ROS Master computer reported the image classification model used and the location of the output classification labels database.

The rqt_graph on the ROS Master provided a new graphical display of all of the ROS nodes and topics incorporated after running the rosimagenet command alias. Each of the camera captures started with the video_source node, which published its associated video_source/raw topic. This topic was subscribed to by its respective imagenet node, that in turn published its classification, overlay, and vision_info topics. By default, only one of these imagenet-related ROS topics, overlay, was subscribed to by the video_output node. Finally, data from each respective video_output node was sent to the rosout node via the rosout topic. A screenshot of the rqt_graph running on the ROS Master computer while imagenet was running for all four cameras via the rosimagenet command alias is shown in Figure 4.28.

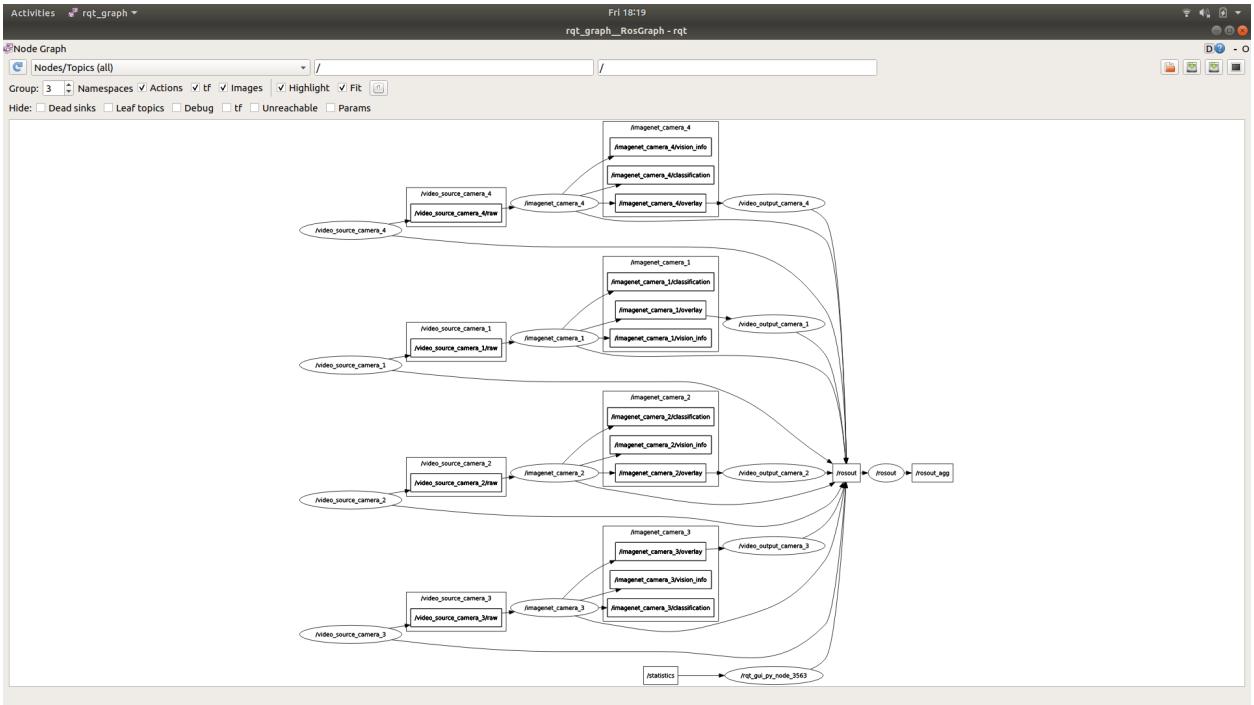


Figure 4.28: ROS rqt_graph from the ROS Master computer while running the command alias rosimagenet on the Jetson TX2.

The roskill command alias used in Listing 4.18 became insufficient, as it did not terminate the imagenet ROS nodes. Therefore, the command alias was enhanced to include the imagenet nodes running for each of the four cameras: imagenet_camera_1,

imagenet_camera_2, imagenet_camera_3, and imagenet_camera_4. This enhancement made it possible to completely start and stop image classification at will. The updated roskill command appears in Listing 4.26.

```
alias roskill='rosnode kill /video_source_camera_1  &    \
               rosnode kill /video_source_camera_2  &    \
               rosnode kill /video_source_camera_3  &    \
               rosnode kill /video_source_camera_4  &    \
               rosnode kill /video_output_camera_1  &    \
               rosnode kill /video_output_camera_2  &    \
               rosnode kill /video_output_camera_3  &    \
               rosnode kill /video_output_camera_4  &    \
               rosnode kill /imagenet_camera_1      &    \
               rosnode kill /imagenet_camera_2      &    \
               rosnode kill /imagenet_camera_3      &    \
               rosnode kill /imagenet_camera_4'
```

Listing 4.26: The roskill Terminal command alias after augmentation to kill imagenet nodes.

The addition of the imagenet node allowed for the capability of classifying images captured by the camera. This could be utilized to identify images that the SAR-UGV would encounter in the field. Image classification, however, may only classify an image as a single entity for general scene understanding. The ability to classify multiple objects in a scene is made possible by object detection.

Object Detection

The problem of detecting objects within a single image, without identifying the entire image itself as a single classification, had already been addressed by jetson-inference. Like imagenet, object detection could also be executed within the ros_deep_learning package. In its operation, detectnet takes input via image file, video file, or live video capture. It then applies a pre-trained neural network model to the image, which then identifies one or more objects within the image frame. The detectnet application uses the same basic architecture as its image classification counterpart when it is launched as part of the ros_deep_learning package, where video input and output are handled by the

video_source and video_output ROS nodes, respectively, but with the detectnet node in between. Allowable output from the detectnet ROS node may be in the form of image files, video files, RTP stream, or live output to a video monitor.

Initial testing with detectnet was performed with its input parameters set as close to default as possible. Only two parameters were specified in order to run detectnet successfully with live video for both capture and output. The specified parameters were the input and output devices being the first Logitech C930e camera and the video monitor connected to the Jetson TX2, respectively. The command used in the Terminal used to launch detectnet with these settings are shown in Listing 4.27.

```
cameras@cameras-tx2:~/$ roslaunch ros_deep_learning          \
    detectnet.ros1.launch                                    \
    input:=v4l2:///dev/video1 output:=display://0
```

Listing 4.27: The Terminal command used to launch the detectnet ROS node.

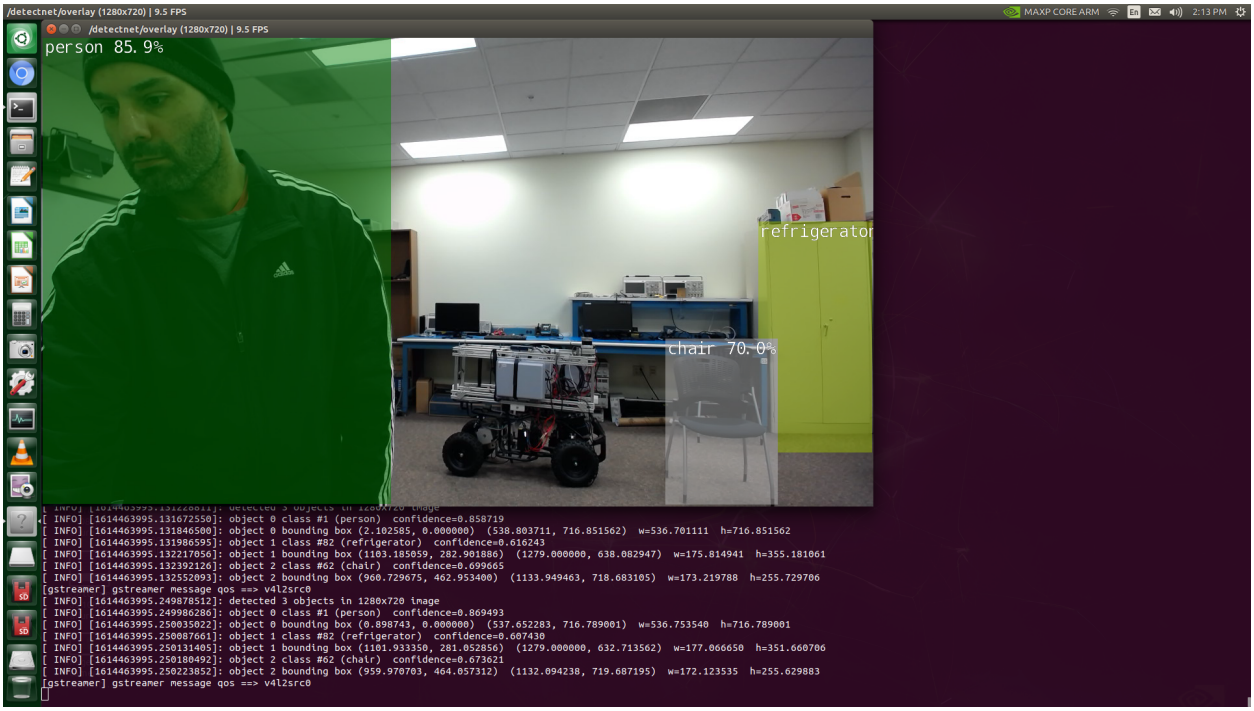


Figure 4.29: Screenshot while running the detectnet ROS launch file on the Jetson TX2. The only parameters specified were input and output devices.

Since the only included parameters for running `detectnet` were the camera and video monitor, the program captured and output the image by default at 720p. Also included in its default set of parameters was the selected neural network model, by which `detectnet` used the `SSD-Mobilenet-v2` model. Figure 4.29 displays a screenshot from the Jetson TX2 while running `detectnet` with the parameters from Listing 4.27. In the image, `detectnet` reported detection of three separate objects in the image:

- `Person` with a confidence interval of 85.9 percent
- `Refrigerator` with a confidence interval of 61.6 percent
- `Chair` with a confidence interval of 70.0%

While `detectnet` matched two of the three objects in the image to ground truth, it must be clarified that the misidentified object, identified as a refrigerator, in Figure 4.29 is not a refrigerator, but a storage cabinet. Also in Figure 4.29, additional output including detected objects and confidence values were reported at the bottom of the Terminal window. The confidence intervals of the latest detection results do not match the picture, as the Terminal and image output did not report synchronously when running `detectnet`.

Like `imagenet` with image classification models, `detectnet` is capable of running inference on other object detection models. Other pre-trained models were included with the jetson-inference software from GitHub. The `detectnet` application help was used to find the complete list of included models. Listing 4.28 displays the output of the `detectnet` program help, which lists the included models for object detection. The `detectnet` help showed the `network` parameter as the argument to be used when selecting the object detection model during program execution.

```
cameras@cameras-tx2:~$ detectnet --help
usage: detectnet [--help] [--network=NETWORK] [--threshold=THRESHOLD]
    ...
input_URI [output_URI]

Locate objects in a video/image stream using an object detection DNN.
See below for additional arguments that may not be shown above.

positional arguments:
    input_URI       resource URI of input stream  (see videoSource
   below)
    output_URI      resource URI of output stream (see videoOutput
   below)

detectNet arguments:
  --network=NETWORK     pre-trained model to load, one of the following
   :
                            * ssd-mobilenet-v1
                            * ssd-mobilenet-v2 (default)
                            * ssd-inception-v2
                            * pednet
                            * multiped
                            * facenet
                            * coco-airplane
                            * coco-bottle
                            * coco-chair
                            * coco-dog
  --model=MODEL         path to custom model to load (caffemodel, uff,
   or onnx)
  --prototxt=PROTOTXT   path to custom prototxt to load (for .
   caffemodel only)
  --labels=LABELS       path to text file containing the labels for
   each class
  --input-blob=INPUT    name of the input layer (default is 'data')
  --output-cvg=COVERAGE name of the coverge output layer (default is '
   coverage')
  --output-bbox=BOXES   name of the bounding output layer (default is '
   bboxes')
  --mean-pixel=PIXEL    mean pixel value to subtract from input (
   default is 0.0)
  --batch-size=BATCH    maximum batch size (default is 1)
  --threshold=THRESHOLD minimum threshold for detection (default is
   0.5)
  --alpha=ALPHA         overlay alpha blending value, range 0-255 (
   default: 120)
  --overlay=OVERLAY     detection overlay flags (e.g. --overlay=box,
   labels,conf)
                        valid combinations are:  'box', 'labels', 'conf
   ', 'none'
```

```
  --profile             enable layer profiling in TensorRT
...
```

Listing 4.28: Partial output of the `detectnet` application help.

From Listing 4.28 above, it can be seen that the following models were available from jetson-inference:

- `SSD-Mobilenet-v1`
- `SSD-Mobilenet-v2`
- `SSD-Inception-v2`
- `DetectNet-COCO-Dog`
- `DetectNet-COCO-Bottle`
- `DetectNet-COCO-Chair`
- `DetectNet-COCO-Airplane`
- `ped-100`
- `multiped-500`
- `facenet-120`

`Detectnet`, like `imagenet`, allowed for the `network` and `model_name` parameters to be used interchangeably. Using the `network` parameter, `detectnet` functioned as expected. During testing, however, changing the `network` parameter name in the command line to `model_name` often produced varying and unexpected results that led to program failure. During these failures, the specified camera activated, but did not produce a captured image on the screen. Terminal output on the Jetson TX2 while running `detectnet` with the `model_name` parameter also indicated that no image was being processed for object detection. It was found that in the instances of using the `model_name` parameter, the `video_source` node would fail. Figure 4.30 shows two instances of running `rqt_graph`

on the ROS Master computer. On the top of Figure 4.30, `detectnet` had been run with the `network` parameter, which provided expected results. On the bottom, `detectnet` had been run with the `model_name` parameter, which showed failure at the `video_source` node. Due to the dead ROS node, its topic, `video_source/raw` was no longer being subscribed to by the `detectnet` node. In addition, two of the ROS topics produced by `detectnet` were no longer existent on the network. The one ROS topic still existent on the network was being published by the `video_output` node, but was no longer actually being published by `detectnet`.

The command used in the Terminal that produced the expected results for running `detectnet` is shown in Listing 4.29. In the listing, it can be seen that the `network` parameter was used, which was required for consistent program output and linking of ROS nodes. This command also specified the `SSD-Inception-v2` model for object detection, specifically, not the default `SSD-Mobilenet-v2` model.

```
cameras@cameras-tx2:~/$ roslaunch ros_deep_learning          \
    detectnet.ros1.launch                                    \
    network:=ssd-inception-v2                                \
    input:=v4l2:///dev/video1 output:=display://0
```

Listing 4.29: The Terminal command used to launch the `detectnet` ROS node.

Listing 4.30 displays the command used in the Terminal that often produced unexpected `detectnet` results by using the `model_name` parameter. As in Listing 4.29, this command also utilized the `SSD-Inception-v2` model for object detection. The use of any one particular model used did not appear to have an affect on program success or failure. Hence, it was decided to use the `network` parameter in future runs of `detectnet` acquire the most consistent successful results possible.

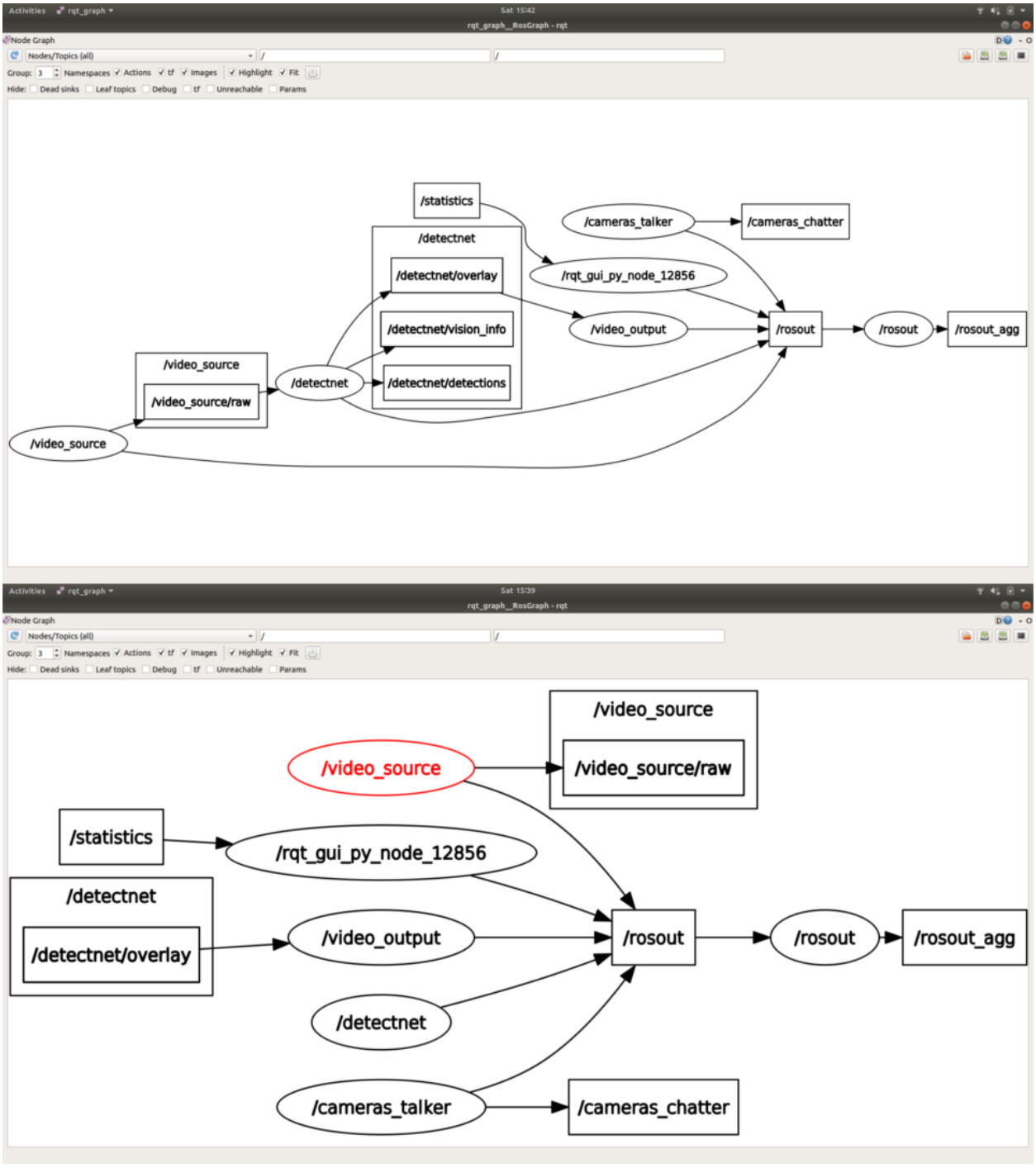


Figure 4.30: Expected results of running `detectnet` with the `network` parameter in the `rqt_graph` (top). Failure of the `video_source` node when running `detectnet` with the `model_name` parameter in the `rqt_graph` (bottom).

```
cameras@cameras-tx2:~/$ roslaunch ros_deep_learning          \
    detectnet.ros1.launch                                    \
    model_name:=ssd-inception-v2                             \
    input:=v4l2:///dev/video1 output:=display://0
```

Listing 4.30: The Terminal command used to launch the `detectnet` ROS node.

In order to run `detectnet` with all four cameras simultaneously on the Jetson TX2, the ROS launch file `detectnet.ros1.launch` needed to be modified. In each file, the capture source of each corresponding camera from which to capture from was specified. Additionally, each `detectnet` launch file was linked to its corresponding `video_source` and `video_output` nodes by referencing their associated launch files. Most other parameters within the launch files were left at their default values, as these parameters could be changed at run time, such as when, for example, a different object detection model needed to be tested. Listing A.21, Listing A.22, Listing A.23, and Listing A.24, contain the source code for all of the launch files that were created and modified to run object detection models with `detectnet`. The list of modified `detectnet` ROS launch files to allow for simultaneous capture and object detection from multiple camera sources are as follows:

- `detectnet_camera_1.ros1.launch`
- `detectnet_camera_2.ros1.launch`
- `detectnet_camera_3.ros1.launch`
- `detectnet_camera_4.ros1.launch`

A command alias was created to launch `detectnet` with all four cameras simultaneously. This was performed similarly to the `rosplay` and `rosimagenet` command aliases discussed previously. The new command alias, `rosdetectnet`, consisted of four separate `roslaunch` commands. Each `roslaunch` command called an individual ROS launch file that corresponded to one of the four cameras. The `SSD-Inception-v2` model was selected to perform object detection on all four of the image captures using the `network` parameter. The contents of the `rosdetectnet` command alias displayed in Listing 4.31.

```
alias rosdetectnet='roslaunch ros_deep_learning                    \
    detectnet_camera_1.ros1.launch                                 \
    network:=ssd-inception-v2                                      \
    input_width:=320 input_height:=180 input_codec:=raw            \
    input:=v4l2:///dev/video1 output:=display://0 &                \
                                                                   \
    roslaunch ros_deep_learning                                    \
    detectnet_camera_2.ros1.launch                                 \
    network:=ssd-inception-v2                                      \
    input_width:=320 input_height:=180 input_codec:=raw            \
    input:=v4l2:///dev/video2 output:=display://0 &                \
                                                                   \
    roslaunch ros_deep_learning                                    \
    detectnet_camera_3.ros1.launch                                 \
    network:=ssd-inception-v2                                      \
    input_width:=320 input_height:=180 input_codec:=raw            \
    input:=v4l2:///dev/video3 output:=display://0 &                \
                                                                   \
    roslaunch ros_deep_learning                                    \
    detectnet_camera_4.ros1.launch                                 \
    network:=ssd-inception-v2                                      \
    input_width:=320 input_height:=180 input_codec:=raw            \
    input:=v4l2:///dev/video4 output:=display://0 &'
```

Listing 4.31: The `rosdetectnet` Terminal command alias added to the `.bash_aliases` file to perform image object detection from four cameras simultaneously at at 320×180 resolution.

Running the `rosdetectnet` command alias in the Terminal produced four new live video windows on the monitor, one for each of the connected Logitech cameras. The 320×180 pixel resolution specified by the `input_width` and `input_height` parameters in Listing 4.31 came as a result of testing the processing and memory capabilities of the Jetson TX2 while running multiple instances of `detectnet`. This resolution was the largest capture size capable of running the `rosdetectnet` command alias without completely exhausting all of the available RAM memory on the Jetson TX2. Running the Ubuntu System Monitor program while executing `rosdetectnet` resulted in live video display of all four camera captures on the monitor. However, the operating system was nearly unusable. While the operating system did not freeze completely, and the instances of `detectnet` continued to perform object detection, preparation of the image shown in Figure 4.31 required about 15 minutes of time to successfully obtain a screenshot while `detectnet` was active after the `rosdetectnet` command alias was executed. While not particularly costly for the CPU,

the Ubuntu System monitor reported 86.7% memory utilization while running `detectnet` from all four cameras simultaneously at 320×180 resolution by use of the `rosdetectnet` command alias.

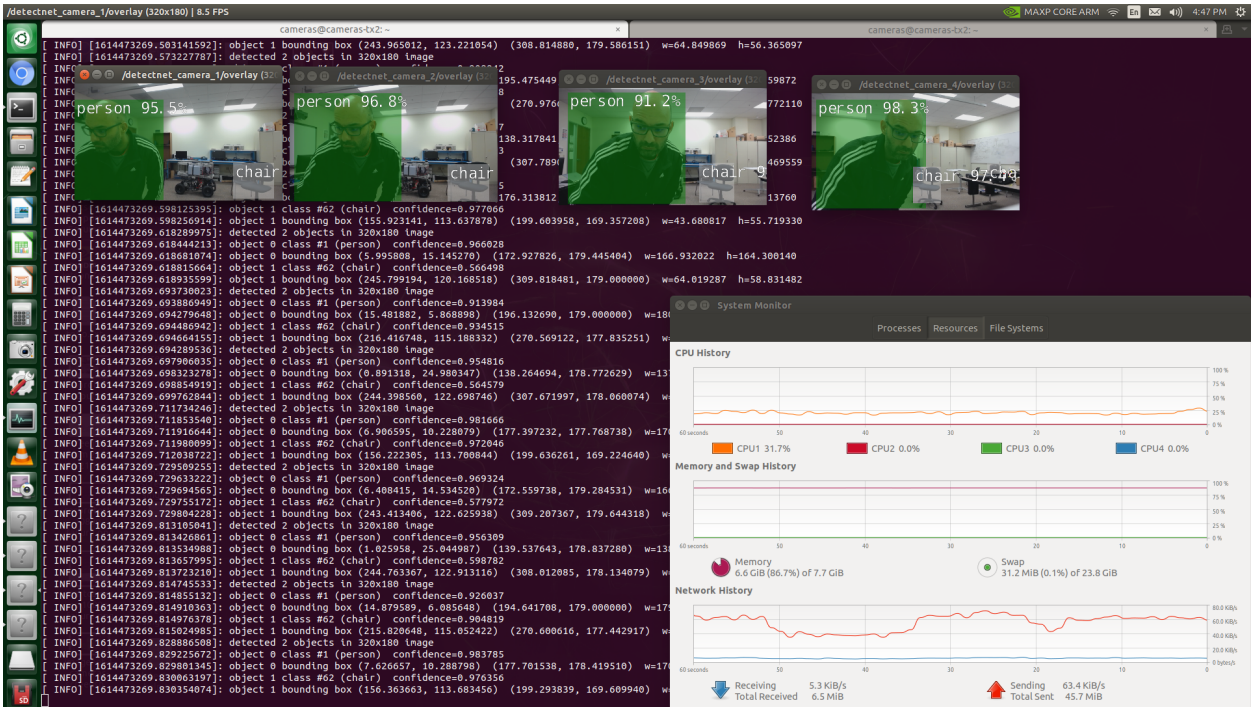


Figure 4.31: Screenshot while running the `rosdetectnet` command alias on the Jetson TX2.

The active ROS nodes and topics associated with `detectnet` could be viewed after running the `rosdetectnet` command alias. Listing 4.32 shows the output from running the command `$ rosnode list` on the ROS Master, which output the active ROS nodes on the network. In addition to the `video_source` and `video_output` nodes, the `detectnet` nodes for each of the four active cameras were reported.

```
ROSMaster@ROSMaster-Computer:~$ rosnode list
/cameras_talker
/detectnet_camera_1
/detectnet_camera_2
/detectnet_camera_3
/detectnet_camera_4
/rosout
/video_output_camera_1
```

```
/video_output_camera_2
/video_output_camera_3
/video_output_camera_4
/video_source_camera_1
/video_source_camera_2
/video_source_camera_3
/video_source_camera_4
```

Listing 4.32: Active nodes on the ROS network shown after running the command `$ rosnode list` on the ROS Master computer after the `rosdetectnet` command alias had been executed on the Jetson TX2.

Three ROS topics for each camera were reported as a result of running the `rosdetectnet` command alias: `detections`, `overlay`, and `vision_info`. Active topics on the ROS network were reported by running the `$ rostopic list` command in the Terminal on the ROS Master machine. The output of this command showed that each of the four cameras had its own set of `detections`, `overlay`, and `vision_info` topics. Listing 4.33 displays the full output of running the `rostopic list` command on the ROS Master computer while `detectnet` was active for all four cameras.

```
ROSMaster@ROSMaster-Computer:~$ rostopic list
/cameras_chatter
/detectnet_camera_1/detections
/detectnet_camera_1/overlay
/detectnet_camera_1/vision_info
/detectnet_camera_2/detections
/detectnet_camera_2/overlay
/detectnet_camera_2/vision_info
/detectnet_camera_3/detections
/detectnet_camera_3/overlay
/detectnet_camera_3/vision_info
/detectnet_camera_4/detections
/detectnet_camera_4/overlay
/detectnet_camera_4/vision_info
/rosout
/rosout_agg
/video_source_camera_1/raw
/video_source_camera_2/raw
/video_source_camera_3/raw
/video_source_camera_4/raw
```

Listing 4.33: Active topics on the ROS network were revealed by running `$ rostopic list` on the ROS Master computer after the `rosdetectnet` command alias had been executed on the Jetson TX2.

The `detections` ROS topic reported object detection results. Among a multitude of other image metadata, output reported by `detections` included class identification number, confidence interval, and detected object position. Listing 4.34, displays the results of running the command `$ rostopic echo /detectnet_camera_1/classification` on the ROS Master. The classification identification results in Listing 4.34 reported `id: 62`, which corresponded with the classification name "chair," as this text was found in line 63 of the file `ssd_coco_labels.txt`, included with jetson-inference, thus aligning the class identification number and name. The `score` of the detection, or confidence interval, was reported at `0.787604629993`, or approximately 78.76%. Detected object position center `x` and `y` coordinates were reported, as well as the `size_x` and `size_y` output parameters providing data necessary to draw a box encompassing the detected object in the output image.

```
ROSMaster@ROSMaster-Computer:~$ rostopic echo /detectnet_camera_1/
   detections
header:
seq: 10617
stamp:
secs: 1614473840
nsecs: 919145530
frame_id: ''
detections:
-
header:
seq: 0
stamp:
secs: 0
nsecs:         0
frame_id: ''
results:
-
id: 62
score: 0.787604629993
pose:
pose:
position:
x: 0.0
y: 0.0
z: 0.0
orientation:
```

```
x: 0.0
y: 0.0
z: 0.0
w: 0.0
covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
   0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
   0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
bbox:
center:
x: 260.913726807
y: 148.725189209
theta: 0.0
size_x: 38.2052764893
size_y: 59.0124740601
source_img:
header:
seq: 0
stamp:
secs: 0
nsecs:         0
frame_id: ''
height: 0
width: 0
encoding: ''
is_bigendian: 0
step: 0
data: []
---
...
```

Listing 4.34: Running `$ rostopic echo /detectnet_camera_1/detections` on the ROS Master computer reported the classification identification number and confidence score from the capture on the first camera connected to the Jetson TX2.

The `overlay` ROS topic reported by `detectnet` contained input image raw data overlaid with object detection results. As it would be unreasonable to show lengthy output text from this topic within this document, Figure 4.32 shows Terminal output on the ROS Master after running the command `$ rostopic echo /detectnet_camera_1/overlay`. Its output included data on the captured image such as `height`, `width`, `encoding`, `step`, `is_bigendian` (i.e., endianness), and `data` (i.e., raw pixel color data).

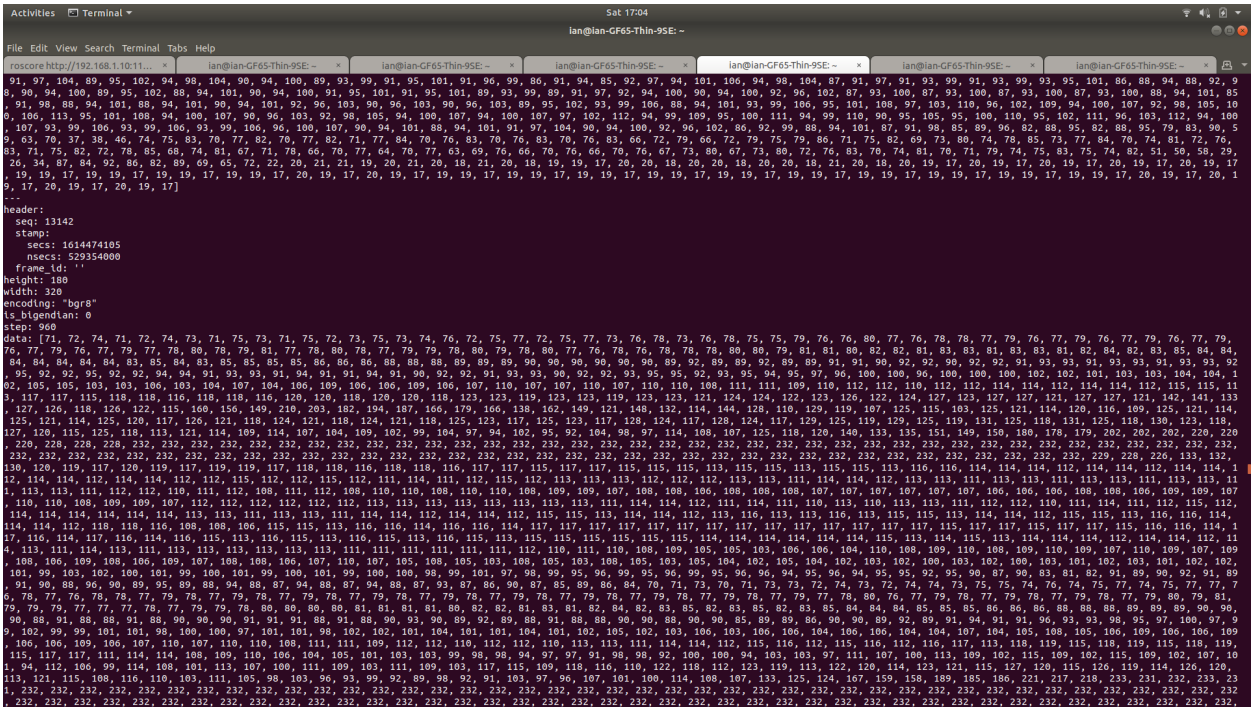


Figure 4.32: Running `$ rostopic echo /detectnet_camera_1/overlay` from the ROS Master computer while running the `rosdetectnet` command alias on the Jetson TX2.

Like `imagenet`, the `detectnet` ROS topic featured a node named `vision_info`. The `vision_info` topic similarly contained vision metadata, such as class labels and parameter name list. From the ROS Master computer, output from using the Terminal command `$ rostopic echo /detectnet_camera_1/vision_info` returned information on the `SSD-Mobilenet-v2` model being used as its `method`. The location of the database created for the reported class labels specific to the `detectnet_camera_1` ROS node was contained in the `database_location` parameter. Listing 4.35 displays the output after executing this command, which displayed additional output parameters, such as `header`, `seq` (i.e., sequence), `stamp`, and `frame_id`.

```
1 ROSMaster@ROSMaster-Computer:~$ rostopic echo /detectnet_camera_1/
     vision_info
2 header:
3 seq: 0
4 stamp:
5 secs: 0
```

```
nsecs:         0
frame_id: ''
method: "/usr/local/bin/networks/SSD-Mobilenet-v2/ssd_mobilenet_v2_coco
   .uff"
database_location: "/detectnet_camera_1/
   class_labels_3185375291238328062"
database_version: 0
---

e_version: 0
---
```

Listing 4.35: Running `$ rostopic echo /detectnet_camera_1/vision_info` on the ROS Master computer reported the object detection model used and the location of the output detection labels database.

The `rqt_graph` on the ROS Master provided a new graphical display of all of the ROS nodes and topics incorporated after running the `rosdetectnet` command alias. Each of the camera captures started with the `video_source` node, which published its `video_source/raw` topic. This topic was subscribed to by its corresponding `detectnet` node, which each then published its own `detect`, `overlay`, and `vision_info` topics after the captured image data had been processed. Of the topics published by the `detectnet` node, the `video_output` node only subscribes to the corresponding `overlay` topic. Data from each respective `video_output` node was sent to the `rosout` node via the `rosout` topic. A screenshot of the `rqt_graph` running on the ROS Master computer, which provided a complete graphical representation of the active ROS nodes and topics on the network after the `rosdetectnet` command alias was executed on the Jetson TX2, is shown in Figure 4.33.

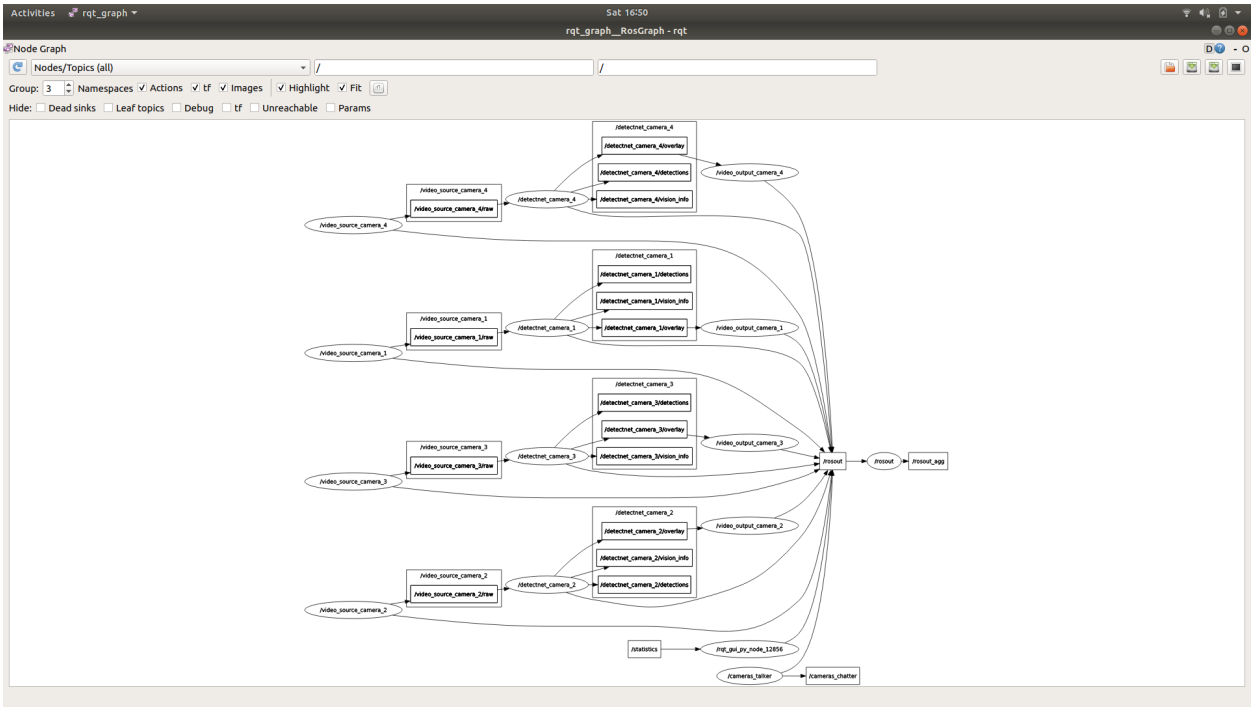


Figure 4.33: ROS `rqt_graph` on the ROS Master computer while running the command alias `rosdetectnet` on the Jetson TX2.

The `roskill` command alias used in Listings 4.18 and 4.36 was once again augmented to handle the termination of the `detectnet` ROS nodes that were created as a result of running the `rosdetectnet` command alias. Lines of text were added to the `.bash_aliases` file to include `detectnet_camera_1`, `detectnet_camera_2`, `detectnet_camera_3`, and `detectnet_camera_4` as part of the `roskill` command alias. This addition of `rosdetectnet` and enhancement of the `roskill` command alias made it possible to control, and specifically stop, object detection jobs on the Jetson TX2. The updated `roskill` command containing the `detectnet` ROS nodes appears in Listing 4.36.

```
alias roskill='rosnode kill /video_source_camera_1  &     \
               rosnode kill /video_source_camera_2  &     \
               rosnode kill /video_source_camera_3  &     \
               rosnode kill /video_source_camera_4  &     \
               rosnode kill /video_output_camera_1  &     \
               rosnode kill /video_output_camera_2  &     \
               rosnode kill /video_output_camera_3  &     \
               rosnode kill /video_output_camera_4  &     \
               rosnode kill /imagenet_camera_1      &     \
```

```
10                  rosnode kill /imagenet_camera_2      &    \
11                  rosnode kill /imagenet_camera_3      &    \
12                  rosnode kill /imagenet_camera_4      &    \
13                  rosnode kill /detectnet_camera_1     &    \
14                  rosnode kill /detectnet_camera_2     &    \
15                  rosnode kill /detectnet_camera_3     &    \
16                  rosnode kill /detectnet_camera_4'
```

Listing 4.36: The `roskill` Terminal command alias after augmentation to kill `detectnet` ROS nodes.

Inclusion of the `detectnet` application allowed the capability to detect objects within images captured by the camera. This could be utilized to identify persons or things that the SAR-UGV would encounter in real-time during mission deployment. Object detection plays a critical role for obstacle avoidance when the model has been trained to identify specific obstacles that could be encountered during mission deployments.

Image Segmentation

Software for performing semantic segmentation on image data was also included at the jetson-inference GitHub repository. Image segmentation could be performed through the `ros_deep_learning` package, just as it had been done previously with `imagenet` and `detectnet`. In the case of semantec segmentation, the ROS node, `segnet`, acquires image or video input via the raw image topic of the `video_source` node. `Segnet` then applies a pre-trained fully convolutional network (FCN) model to the image. The FCN performs much like image classification as seen with `imagenet`, but classification is performed for every pixel rather than for the entire image as a whole. After image segmentation has been completed, the `overlay` topic data is then subscribed to by the `video_output` node, which then can be sent to image or video file output, or viewed via video monitor or live stream.

Input and output parameters for `segnet` were set as close to possible at default values during initial testing. The desired input was a single Logitech camera with an output of live

processed video to the monitor connected to the Jetson TX2. The command used in the Terminal to launch `segnet` with only input and output parameters specified during initial testing is shown in Listing 4.37.

```
cameras@cameras-tx2:~/$ roslaunch                                    \
    ros_deep_learning                                                \
    segnet.ros1.launch                                               \
    input:=v4l2:///dev/video1 output:=display://0
```

Listing 4.37: The Terminal command used to launch the `segnet` ROS node.

As image size was not part of the custom parameter set used in Listing 4.37, the `segnet` program captured and output the image by default at 720p. The model selected by default when `segnet` was launched was `FCN-ResNet18-MHP-512x320`, which is based on the Multi-Human Parsing dataset [47]. Figure 4.34 displays a screenshot from the Jetson TX2 while running `detectnet` with the parameters from Listing 4.37.

Figure 4.34: Screenshot on the Jetson TX2 while running the `segnet` ROS launch file. The only parameters specified were input and output devices.

From Figure 4.34, vivid colors can be seen, indicating that `detectnet` reported classification of known objects in the image at the pixel level. These objects included *background*, *face*, *hand*, *shirt*, *jacket/coat*, *pants*, and *shoe/boot*. The complete list of classes and colors for the `FCN-ResNet18-MHP-512x320` model was found to be contained in the `classes.txt` and `colors.txt` files, respectively, located within the jetson-inference software files, specifically in its `data/networks/FCN-ResNet18-MHP-512x320` directory on the Jetson TX2 machine. Table 4.1 shows the complete list of classes and their associated colors in the `FCN-ResNet18-MHP-512x320` model. The color legend of Table 4.1 is also displayed in Table B.6. All other image segmentation model color legends are contained in Appendix B.

The `segnet` application help was accessed to obtain information regarding other available parameters that could be included when the program is launched, and specifically to learn what other image segmentation models were readily available. Listing 4.38 displays partial output of the `segnet` help output from the command line on the Jetson TX2, where it could be seen that eleven different semantic segmentation models were available from the software obtained from the jetson-inference GitHub repository. All of the models included, as stated in the model names, are based off of the `FCN-ResNet18` network. The `FCN-ResNet18` is a transformed, or fully convolutional version of the `Resnet-18` image classification model available for `imagenet`.

Table 4.1: FCN-RESNET18-MHP-512X320 MODEL CLASSES

Class ID	Class Name	R	G	B	Color
0	background	0	0	0	
1	hat/helmet/hair	139	69	19	
2	face	222	184	135	
3	hair	210	105	30	
4	arm	255	255	0	
5	hand	255	165	0	
6	shirt	0	255	0	
7	jacket/coat	60	179	113	
8	dress/robe	107	142	35	
9	bikini/bra	255	0	0	
10	torso_skin	245	222	179	
11	pants	0	0	255	
12	shorts	0	255	255	
13	socks/stockings	238	130	238	
14	shoe/boot	128	0	128	
15	leg	255	0	0	
16	foot	255	0	255	
17	backpack/purse/bag	128	128	128	
18	sunglasses/eyewear	192	192	192	
19	other_accessory	128	128	128	
20	other_item	128	128	128	

```
cameras@cameras-tx2:~$ segnet --help
usage: segnet [--help] [--network NETWORK] ...
input_URI [output_URI]

Segment and classify a video/image stream using a semantic segmentation
    DNN.
See below for additional arguments that may not be shown above.

positional arguments:
    input_URI       resource URI of input stream  (see videoSource
   below)
```

```
    output_URI       resource URI of output stream (see videoOutput
   below)

segNet arguments:
  --network=NETWORK    pre-trained model to load, one of the following:
                           * fcn-resnet18-cityscapes-512x256
                           * fcn-resnet18-cityscapes-1024x512
                           * fcn-resnet18-cityscapes-2048x1024
                           * fcn-resnet18-deepscene-576x320
                           * fcn-resnet18-deepscene-864x480
                           * fcn-resnet18-mhp-512x320
                           * fcn-resnet18-mhp-640x360
                           * fcn-resnet18-voc-320x320 (default)
                           * fcn-resnet18-voc-512x320
                           * fcn-resnet18-sun-512x400
                           * fcn-resnet18-sun-640x512
  --model=MODEL        path to custom model to load (caffemodel, uff,
   or onnx)
  --prototxt=PROTOTXT  path to custom prototxt to load (for .caffemodel
    only)
  --labels=LABELS      path to text file containing the labels for each
    class
  --colors=COLORS      path to text file containing the colors for each
    class
  --input-blob=INPUT   name of the input layer (default: 'data')
  --output-blob=OUTPUT name of the output layer (default: '
   score_fr_21classes')
  --batch-size=BATCH   maximum batch size (default is 1)
  --alpha=ALPHA        overlay alpha blending value, range 0-255 (
   default: 120)
  --visualize=VISUAL   visualization flags (e.g. --visualize=overlay,
   mask)
                       valid combinations are:  'overlay', 'mask'
  --profile            enable layer profiling in TensorRT

...
```

Listing 4.38: The `segnet` application help in the Terminal revealed the pre-trained semantic segmentations models available after downloading the software from jetson-inference.

The semantic segmentation models included with jetson-inference were trained from a variety of different datasets at different image resolutions. These datasets differ in the content of the images they contain, ranging from outdoor forest settings to indoor home and office settings. The five datasets that served as the foundations for the jetson-inference image segmentation models were: Cityscapes [48], DeepScene [49], Multi-Human Parsing [50], PASCAL VOC [10], and SUN RGB-D [51]. Table 4.2 contains a list of the semantic segmen-

tation models packaged with the jetson-inference software and the image datasets they were built from.

Table 4.2: SEMANTIC SEGMENTATION MODELS AND SOURCE DATASETS

Model	Dataset
FCN-ResNet18-Cityscapes-512x256	Cityscapes
FCN-ResNet18-Cityscapes-1024x512	Cityscapes
FCN-ResNet18-Cityscapes-2048x1024	Cityscapes
FCN-ResNet18-DeepScene-576x320	DeepScene
FCN-ResNet18-DeepScene-864x480	DeepScene
FCN-ResNet18-MHP-512x320	Multi-Human Parsing
FCN-ResNet18-MHP-640x360	Multi-Human Parsing
FCN-ResNet18-VOC-320x320	PASCAL VOC
FCN-ResNet18-VOC-512x320	PASCAL VOC
FCN-ResNet18-SUN-512x400	SUN RGB-D
FCN-ResNet18-SUN-640x512	SUN RGB-D

The use of the `network` parameter when attempting to run `segnet` with a different semantic segmentation model always loaded the default `FCN-ResNet18-VOC-320x320` model. This exhibited different behavior than `imagenet` and `detectnet`, which both allowed for the `network` and `model_name` parameters to be used interchangeably, despite some varying results with `detectnet`. As a result, using `segnet` required the specific use of the `model_name` parameter to successfully load a model rather than the `network` parameter specifically listed in the program help text. Listing 4.39 shows the command used in the Terminal to launch `segnet` with the `FCN-ResNet18-SUN-640x512` model.

```
1 cameras@cameras-tx2:~/$ roslaunch ros_deep_learning          \
2     segnet.ros1.launch                                       \
3     model_name:=fcn-resnet18-sun-640x512                     \
4     input:=v4l2:///dev/video1 output:=display://0
```

Listing 4.39: The Terminal command used to launch the `segnet` ROS node with the `FCN-ResNet18-SUN-640x512` model.

When testing in the laboratory, using the `FCN-ResNet18-SUN-640x512` model produced results that successfully segmented pixels in the captured video and labeled them to match ground truth. The SUN RGB-D dataset contains image data from home and office environments. Since the `FCN-ResNet18-SUN-640x512` model from jetson-inference was trained on this dataset, it was able to identify objects in the lab such as the chair and desk, highlighted in purple and white, respectively, in the forefront of Figure 4.35. The `FCN-ResNet18-SUN-640x512` model also identified the storage cabinet, floor, wall, and ceiling in the lab after specifying it when running `segnet`.

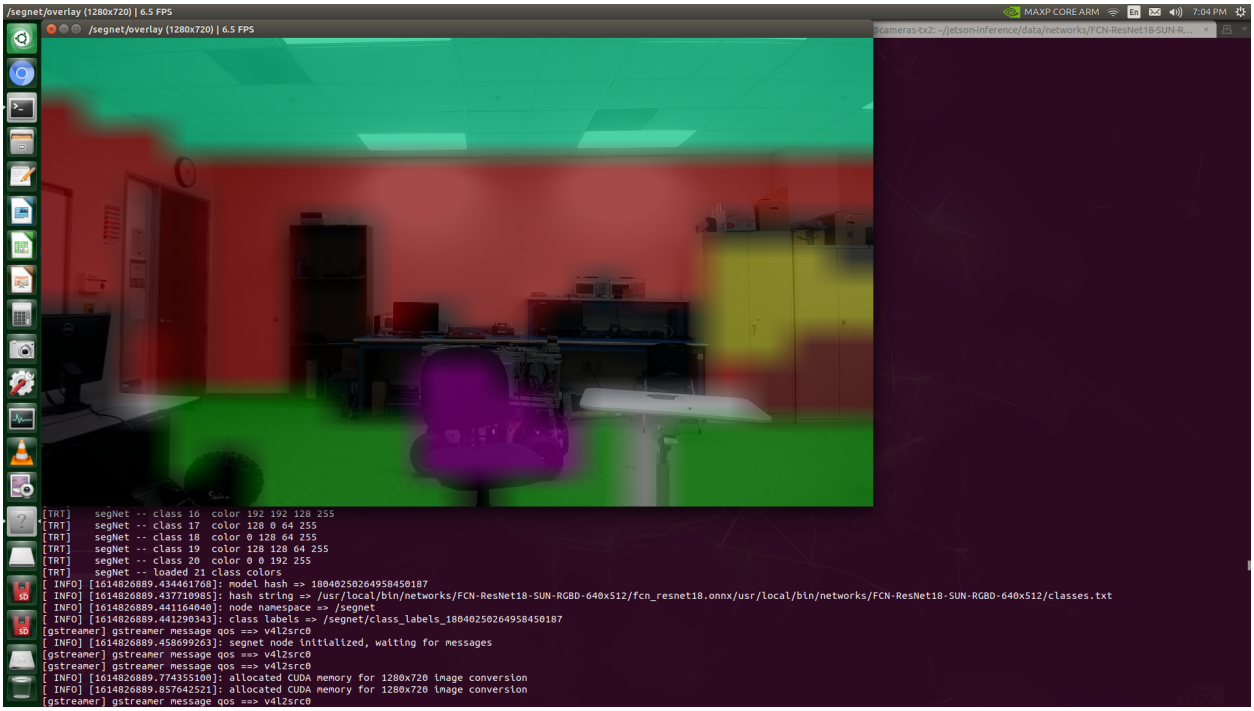


Figure 4.35: Running `segnet` with the `FCN-ResNet18-SUN-640x512` model in the laboratory successfully segmented pixels in the image and labeled them properly.

The `segnet.ros1.launch` ROS launch file was copied and modified in order to create unique `segnet` ROS nodes that would enable simultaneous video capture from the four Logitech C930e cameras connected to the Jetson TX2. Like `imagenet` and `detectnet`, each individual `segnet` node subscribed to its corresponding `video_source` ROS node for data input. The data published by the particular `segnet` node was then subscribed to

by its respective `video_output` node. The source code of all of the new `segnet` ROS launch files appear in Listings A.25, A.26, A.27, and A.28. The complete list of the new `segnet` ROS launch files created to allow simultaneous capture from multiple cameras and simultaneous output to different displays, streams or files are as follows:

- `segnet_camera_1.ros1.launch`
- `segnet_camera_2.ros1.launch`
- `segnet_camera_3.ros1.launch`
- `segnet_camera_4.ros1.launch`

To quickly launch `segnet` from all four cameras simultaneously, a command alias was created in the `.bash_aliases` file of the Jetson TX2. This command alias, `rossegnet`, contained four separate `roslaunch` commands, in which each of the `roslaunch` commands called the particular ROS launch file corresponding to one of the four cameras. The `FCN-ResNet18-Cityscapes-512x256` model was selected to perform semantic segmentation on all four of the image captures using the `network` parameter. The contents of the `rosdetectnet` command alias that was saved to the `.bash_aliases` file on the Jetson TX2 is displayed in Listing 4.40.

```
alias rossegnet='roslaunch ros_deep_learning                    \
    segnet_camera_1.ros1.launch                                 \
    model_name:=fcn-resnet18-cityscapes-512x256                 \
    input_width:=640 input_height:=360 input_codec:=raw         \
    input:=v4l2:///dev/video1 output:=display://0 &             \
                                                                \
    roslaunch ros_deep_learning                                 \
    segnet_camera_2.ros1.launch                                 \
    model_name:=fcn-resnet18-cityscapes-512x256                 \
    input_width:=640 input_height:=360 input_codec:=raw         \
    input:=v4l2:///dev/video2 output:=display://0 &             \
                                                                \
    roslaunch ros_deep_learning                                 \
    segnet_camera_3.ros1.launch                                 \
    model_name:=fcn-resnet18-cityscapes-512x256                 \
    input_width:=640 input_height:=360 input_codec:=raw         \
```

```
    input:=v4l2:///dev/video3 output:=display://0 &             \
                                                               \
    roslaunch ros_deep_learning                                \
    segnet_camera_4.ros1.launch                                \
    model_name:=fcn-resnet18-cityscapes-512x256                \
    input_width:=640 input_height:=360 input_codec:=raw        \
    input:=v4l2:///dev/video4 output:=display://0 &'
```

Listing 4.40: The `rossegnet` Terminal command alias added to the `.bash_aliases` file to perform image segmentation from four cameras simultaneously at at 360p resolution.

After executing the `rossegnet` command alias in the Terminal of the Jetson TX2, four new video capture windows were output on the monitor. Each of the captures coincided with one of the connected Logitech cameras. The 640×360 pixel resolution used in Listing 4.40 came as a result of testing the processing and memory capabilities of the Jetson TX2 while running multiple instances of `segnet`. During initial testing of `segnet` using the default 720p resolution, subsequent runs of the program would fail to reinitialize the camera. Being aware of this failure, smaller capture resolutions were tested when attempting to obtain multiple simultaneous captures. While running at 360p via the `rossegnet` command alias, the Jetson TX2 System Monitor showed 86.5% memory usage, and the cameras were only capable of capturing live video at approximately seven frames per second. This reduction of performance, both for the capture rate and overall system performance, led to the decision not to further increase the capture resolution during simultaneous camera capture. Simultaneous capture from all four cameras with live video output to the Jetson TX2 monitor from the execution of the `rossegnet` command alias is shown in Figure 4.36.

After running the `rossegnet` command alias on the Jetson TX2, the active nodes associated with `segnet` could be viewed seen from a Terminal window on the ROS Master machine. Listing 4.41 shows the output from running the command `$ rosnode list` on the ROS Master. The new `segnet` nodes were reported alongside their associated `video_source` and `video_output` ROS nodes.

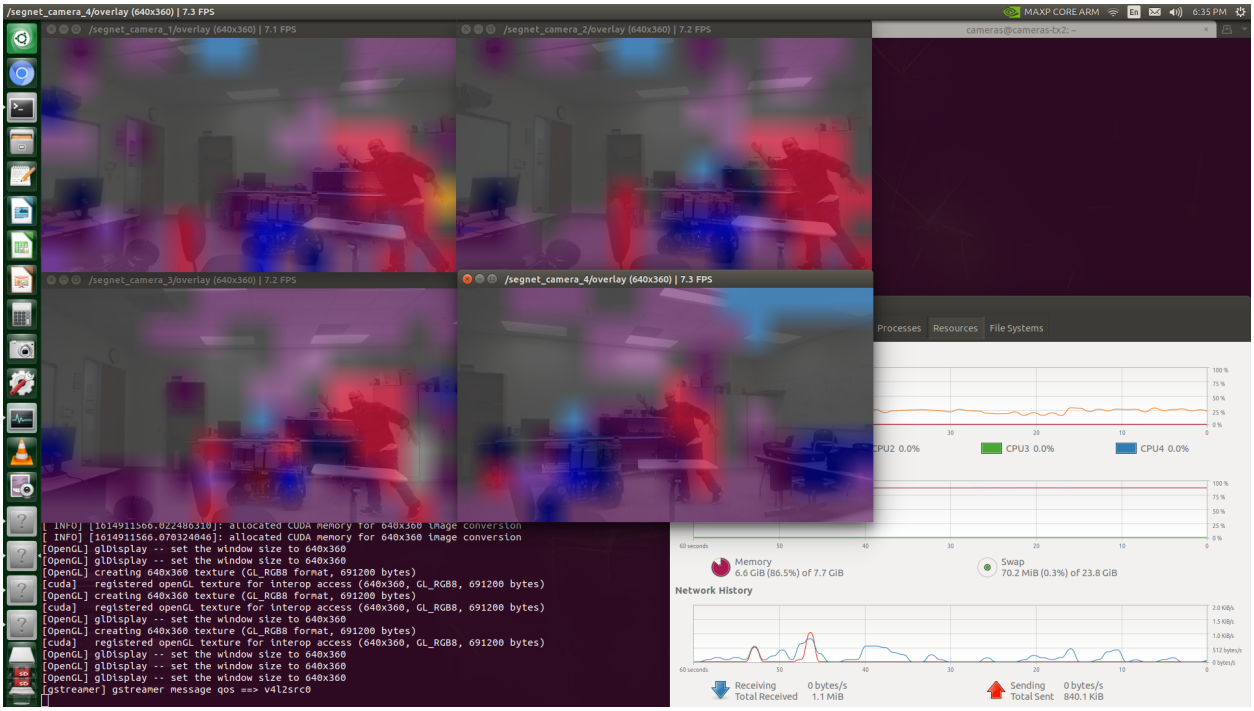


Figure 4.36: Screenshot while running the `rossegnet` command alias on the Jetson TX2.

```
ROSMaster@ROSMaster-Computer:~$ rosnode list
/segnet_camera_1
/segnet_camera_2
/segnet_camera_3
/segnet_camera_4
/rosout
/video_output_camera_1
/video_output_camera_2
/video_output_camera_3
/video_output_camera_4
/video_source_camera_1
/video_source_camera_2
/video_source_camera_3
/video_source_camera_4
```

Listing 4.41: Active nodes on the ROS network shown after running the command `$ rosnode list` on the ROS Master computer after the `rossegnet` command alias had been executed on the Jetson TX2.

Executing the `rossegnet` command alias resulted in four new ROS topics for each `segnet` node: `class_mask`, `color_mask`, `overlay`, and `vision_info`. Active topics on the ROS network were reported by running the `$ rostopic list` command in the Ter-

minal on the ROS Master machine. The output of this command showed that each of the four cameras had its own set of `class_mask`, `color_mask`, `overlay`, and `vision_info` ROS topics. Listing 4.42 displays the full output of running the `$ rostopic list` command on the ROS Master machine while `segnet` was active for all four cameras via the `rossegnet` command alias.

```
ROSMaster@ROSMaster-Computer:~$ rostopic list
/rosout
/rosout_agg
/segnet_camera_1/class_mask
/segnet_camera_1/color_mask
/segnet_camera_1/overlay
/segnet_camera_1/vision_info
/segnet_camera_2/class_mask
/segnet_camera_2/color_mask
/segnet_camera_2/overlay
/segnet_camera_2/vision_info
/segnet_camera_3/class_mask
/segnet_camera_3/color_mask
/segnet_camera_3/overlay
/segnet_camera_3/vision_info
/segnet_camera_4/class_mask
/segnet_camera_4/color_mask
/segnet_camera_4/overlay
/segnet_camera_4/vision_info
/video_source_camera_1/raw
/video_source_camera_2/raw
/video_source_camera_3/raw
/video_source_camera_4/raw
```

Listing 4.42: Active topics on the ROS network were revealed by running `$ rostopic list` on the ROS Master computer after the `rossegnet` command alias had been executed on the Jetson TX2.

The `class_mask` ROS topic from the `segnet` node contained data on the 8-bit mono single-channel image, where each pixel was represented by the classification class ID in its matrix `data` parameter. Also included was header data, such as frame sequence (`seq`) and time stamp (`secs` and `nsecs`). The `height` and `width` data points of each `class_mask` output were also included for each frame. Listing 4.43 shows partial output of running the command `$ rostopic echo /segnet_camera_1/class_mask` on the ROS Master computer.

```
ROSMaster@ROSMaster-Computer:~$ rostopic echo /segnet_camera_1/
   class_mask
header:
seq: 0
stamp:
secs: 1615058207
nsecs: 220260586
frame_id: ''
height: 8
width: 16
encoding: "mono8"
is_bigendian: 0
step: 16
data: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
---
header:
seq: 1
stamp:
secs: 1615058207
nsecs: 362008016
frame_id: ''
height: 8
width: 16
encoding: "mono8"
is_bigendian: 0
step: 16
data: [3, 3, 12, 4, 2, 4, 4, 4, 4, 2, 4, 4, 4, 4, 4, 4, 4, 4, 3, 4,
   4, 4, 4, 4, 2, 10, 8, 4, 1, 3, 4, 6, 3, 3, 14, 2, 7, 9, 2, 2, 2, 9,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
---
...
```

Listing 4.43: Running `$ rostopic echo /segnet_camera_1/class_mask` on the ROS Master machine reported the classification identification output from the capture on the first camera connected to the Jetson TX2.

Semantic segmentation using `segnet` produced a color for each classified pixel on the original image capture. The transparency of which could be adjusted when the program was executed by changing its `overlay_alpha` input parameter. Valid values for this parame-

ter range from 0 (fully transparent) to 255 (fully visible). The RGB color data was found to be contained in the `color_mask` ROS topic. It's output could be seen by running the command `$ rostopic echo /segnet_camera_1/color_mask` on the ROS Master. A screenshot of this command's output in the Terminal is displayed in Figure 4.37.

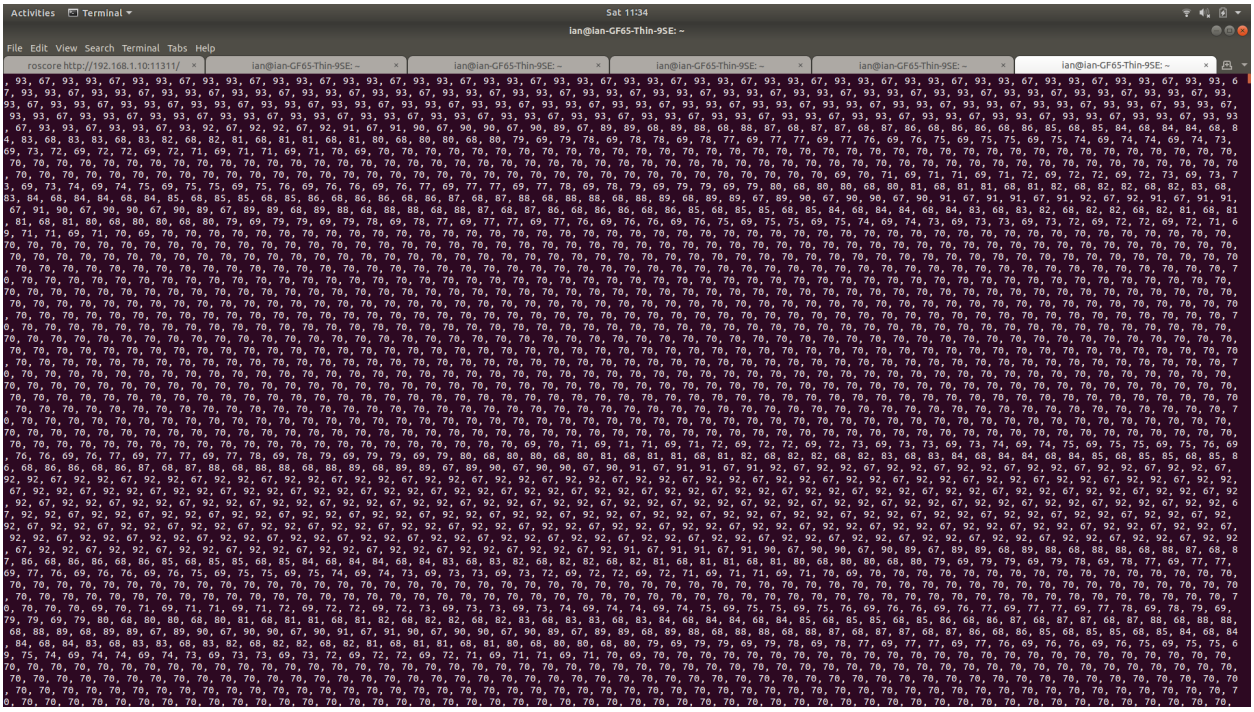


Figure 4.37: Running `$ rostopic echo /segnet_camera_1/color_mask` from the ROS Master computer while the `rossegnet` command alias was active on the Jetson TX2.

The `segnet` ROS node output a topic named `overlay`. This topic contained similar data as the `overlay` topic from the `imagenet` and `detectnet` nodes. The `segnet` node's `overlay` ROS topic featured input image raw data overlaid with per-pixel classification results. Terminal output from the ROS Master computer after executing the command `$ rostopic echo /segnet_camera_1/overlay` is displayed in Figure 4.38.

Just as the `imagenet` and `detectnet` nodes both featured `vision_info` topics, the `segnet` node also published a `vision_info` ROS topic. This particular instance of `vision_info` ROS topic contained vision metadata, such as class labels and parameter

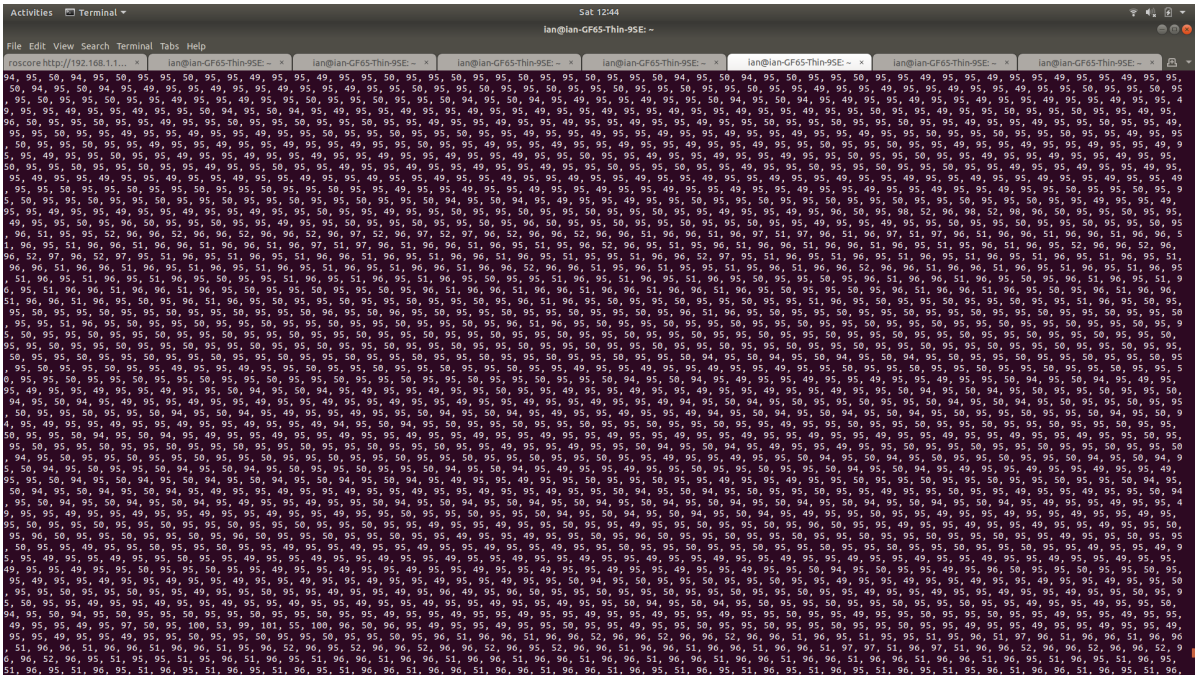


Figure 4.38: Running `$ rostopic echo /segnet_camera_1/overlay` from Terminal of the ROS Master computer while the `rossegnet` command alias was active on the Jetson TX2.

name list. Output of the `$ rostopic echo /segnet_camera_1/vision_info` command on the ROS Master reported the `FCN-ResNet18-Cityscapes-512x256` model being used as its `method` by its file location, as well as the the database file location created for the specific `segnet_camera_1` node's reported object class labels. The text in Listing 4.44 contains the Terminal output after using the ROS Master computer to execute the command `$ rostopic echo /segnet_camera_1/vision_info`, and displays additional output metadata related to the captured frame such as `header`, `seq` (i.e., sequence), `stamp`, and `frame_id`.

```
ROSMaster@ROSMaster-Computer:~$ rostopic echo /segnet_camera_1/
   vision_info
header:
seq: 0
stamp:
secs: 0
nsecs:          0
frame_id: ''
method: "/usr/local/bin/networks/FCN-ResNet18-Cityscapes-512x256/
   fcn_resnet18.onnx"
```

```
database_location: "/segnet_camera_1/class_labels_47739257020531159O4"
database_version: 0
---
```

Listing 4.44: Running `rostopic echo /segnet_camera_1/vision_info` on the ROS Master reported the semantic segmentation model used and the location of the output classification labels database.

Running the `rqt_graph` command from the Terminal of the ROS Master provided a new graphical display of all of the ROS nodes and topics that were active after running the `rossegnet` command alias on the Jetson TX2. The camera captures each started with its respective `video_source` node, which published its `video_source/raw` topic. This topic was subscribed to by the corresponding `segnet` node, which each subsequently published its own `color_mask`, `class_mask`, `overlay`, and `vision_info` topics after the captured image data had been processed. The associated `video_output` ROS node was the only subscriber to any of the topics published by `segnet`. Of the four topics that `segnet` published, the `overlay` topic was the only one that associated `video_output` ROS nodesubscribed to. Data from each respective `video_output` node was sent to the `rosout` node via the `rosout` topic. Figure 4.39 displays screenshot of the `rqt_graph` running on the ROS Master computer while the `segnet` ROS nodes were active on the network.

Just as in previous iterations of adding support for other programs in the `roskill` command alias, it was once again modified to include the `segnet` ROS nodes when all related processes needed to be closed. After the addition of the `segnet` nodes to the `roskill` command alias, it was capable of terminating the `segnet` ROS nodes that were created as a result of running the `rossegnet` command alias. The `.bash_aliases` file on the Jetson TX2 then included the ROS nodes `segnet_camera_1`, `segnet_camera_2`, `segnet_camera_3`, and `segnet_camera_4` as part of the `roskill` command alias. The addition of the `rossegnet` command alias and the augmentation to the `roskill` command alias made it possible to start and stop simultaneous semantic segmentation processes on images captured from the four connected Logitech C930e cameras on the Jetson TX2. The final iteration of the `roskill` command alias is contained in Listing 4.45.

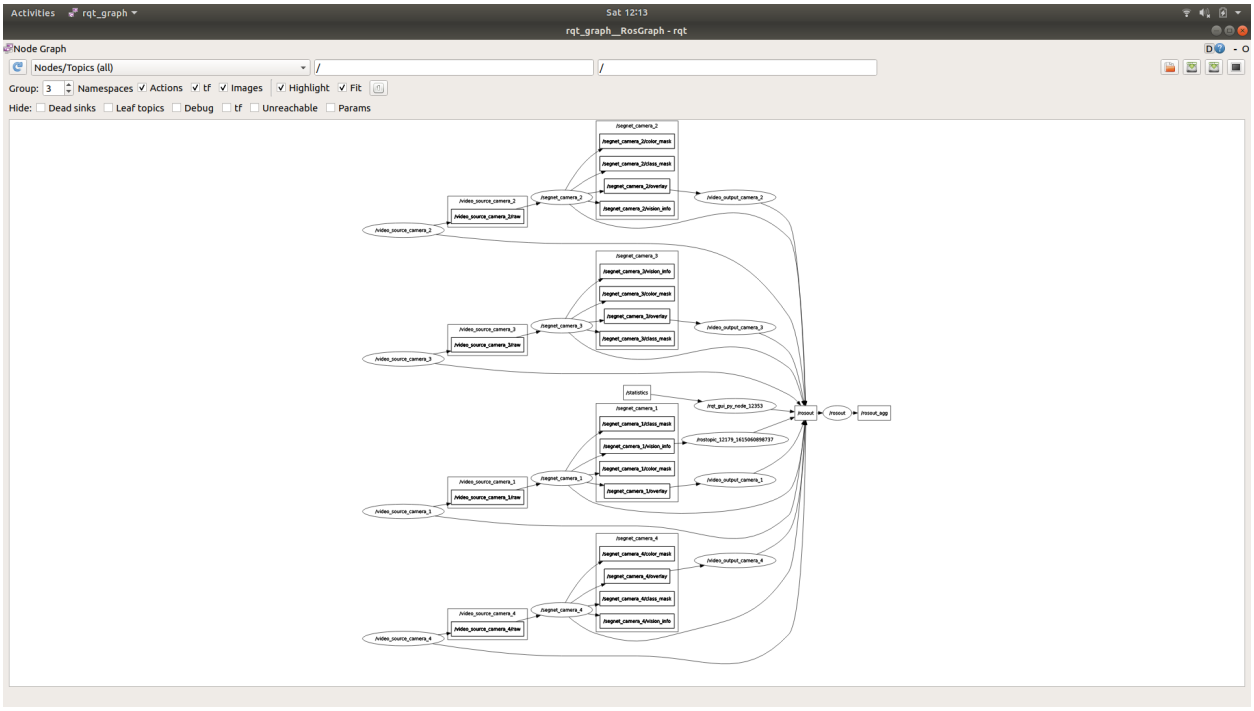


Figure 4.39: ROS `rqt_graph` on the ROS Master computer while running the `rossegnet` command alias on the Jetson TX2.

```
alias roskill='rosnode kill /video_source_camera_1   &     \
               rosnode kill /video_source_camera_2   &     \
               rosnode kill /video_source_camera_3   &     \
               rosnode kill /video_source_camera_4   &     \
               rosnode kill /video_output_camera_1   &     \
               rosnode kill /video_output_camera_2   &     \
               rosnode kill /video_output_camera_3   &     \
               rosnode kill /video_output_camera_4   &     \
               rosnode kill /imagenet_camera_1       &     \
               rosnode kill /imagenet_camera_2       &     \
               rosnode kill /imagenet_camera_3       &     \
               rosnode kill /imagenet_camera_4       &     \
               rosnode kill /detectnet_camera_1      &     \
               rosnode kill /detectnet_camera_2      &     \
               rosnode kill /detectnet_camera_3      &     \
               rosnode kill /detectnet_camera_4      &     \
               rosnode kill /segnet_camera_1         &     \
               rosnode kill /segnet_camera_2         &     \
               rosnode kill /segnet_camera_3         &     \
               rosnode kill /segnet_camera_4'
```

Listing 4.45: The `roskill` Terminal command alias after augmentation to kill `imagenet` nodes.

Implementation of the `segnet` node demonstrated the capability for a fully convolutional deep neural network model to perform analysis on images and video captured live from multiple cameras. Semantic segmentation could be applied to image data capture from the SAR-UGV in an outdoor forested environment to distinguish navigable trails from obstacles and otherwise untraversable terrain. Per-pixel image classification could potentially be utilized for multiple simultaneous mission tasks, such as identifying persons while at the same time maintaining course on a trail.

4.2.3 Bill of Materials

This section pertains to the hardware that was purchased in order to implement the image processing system for the SAR-UGV. The software implementation that was used in this project consisted entirely of open source code or other free software resources, programs, etc. Table 4.3 shows the bill of materials for the image processing system.

Table 4.3: IMAGE PROCESSING SYSTEM BILL OF MATERIALS

Item	**Price**	**Quantity**	**Total**
NVIDIA Jetson TX2 Developer Kit	$399.00	1	$399.00
Logitech C930e Camera	$75.00	4	$300.00
HighPoint RocketU 1344A PCI-Express 3.0	$159.00	1	$159.00
Samsung 860 EVO 500 GB 2.5 Inch SATA III Internal SSD	$53.99	1	$53.99
SATA Cable, Male-to-Female	$5.69	1	$5.69
Antenna Cables	$6.99	1	$6.99
Noctua NF-A4x20 5V PWM fan	$14.95	1	$14.95
MakerFocus PI-FAN DC Brushless Fan	$10.99	1	$10.99
Altelix Green NEMA Enclosure	$59.99	1	$59.99
			$1010.60

CHAPTER 5 Results & Performance Assessment

This chapter discusses the image processing system in experiment following its design, development, and implementation. First, simulations will be reviewed. Experimental results of the system are then discussed from field evaluations. Finally, this chapter provides a performance characterization of the image processing system.

5.1 Simulations

Two simulation methods were prepared for the image processing system prior to performing field experiments. The first simulation method used images from the Freiburg Forest data set [49]. The second simulation method used live video recordings shot in the walking trails at Memorial Park in Wilsonville, OR, located near the Oregon Institute of Technology Portland-Metro campus (i.e., Oregon Tech). Both data sets featured images of landscapes that the SAR-UGV could possibly encounter, in which it would have to perform image classification, object detection, and image segmentation.

5.1.1 Image Classification

Still images from the Freiburg Forest dataset were processed by `imagenet` for image classification. The Freiburg Forest dataset "contains over 15,000 images of unstructured forest environments [52]," making it ideal for the SAR-UGV. Ten different models available from jetson-inference were used with `imagenet` to simulate the image classification results. Figure 5.1 shows one representative image from the Freiburg Forest dataset.

Figure 5.1: Sample image from the Freiburg Forest dataset.

Performing image classification by executing the `imagenet` program with the inputs of the image in Figure 5.1 and ten different neural network models produced ten output images. These images were combined into a montage for display purposes in this document so that the output results may be compared. This montage is displayed in Figure 5.2, where the unique classifications from each model can be seen in the top-left of each image cell.

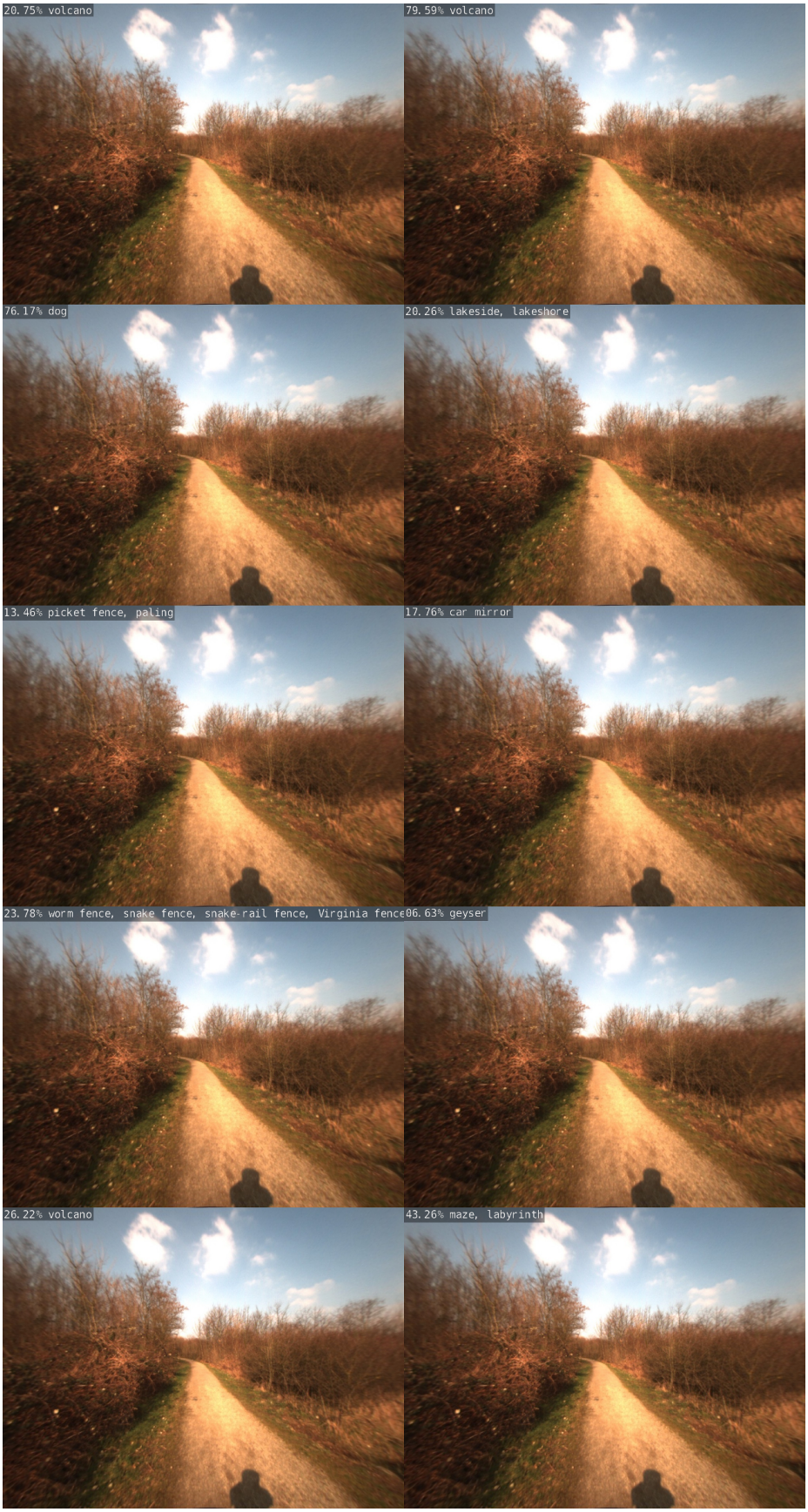


Figure 5.2: Image output after running `imagenet` with ten different image classification models on the image obtained from the Freiburg Forest dataset in Figure 5.1.

Table 5.1 contains the image classification results after using the `imagenet` program with all ten models. In addition to the primary classification label output from each model, `imagenet` reported the confidence interval associated with each individual image classification. The confidence interval data reported from each model is also contained in Table 5.1.

Table 5.1: IMAGE CLASSIFICATION MODEL RESULTS FROM FREIBURG FOREST DATASET

Model	**Primary Classification**	**Confidence (%)**
`AlexNet`	volcano	20.75
`GoogleNet`	volcano	79.59
`GoogleNet-12`	dog	76.17
`ResNet-18`	lakeside	20.26
`ResNet-50`	picket fence	13.46
`ResNet-101`	car mirror	17.76
`ResNet-152`	worm fence	23.78
`VGG-16`	geyser	6.63
`VGG-19`	volcano	26.22
`Inception-v4`	maze	43.26

A representative frame captured from a video that was shot in the trails of Memorial Park is displayed in Figure 5.3. It can be seen that this image contains features of a forested landscape, such as an unpaved dirt trail, trees, leaves, and other foliage.

Figure 5.3: Still image taken from a video clip of a forested trail scene without obstructions at Memorial Park in Wilsonville, OR.

The `imagenet` program was run using the image in Figure 5.3. Like the simulations performed on the Freibug Forest dataset, the same ten image classification models that had been applied previously as input were again used when running `imagenet`. The output images that resulted after image classification processing are shown in the montage in Figure 5.4. As a standard feature of the `imagenet` program, the image classification output appears at the top-left of each captured frame cell. Text of the `imagenet` classifications and corresponding confidence intervals from each neural network model applied to Figure 5.3 are contained in Table 5.2.

Figure 5.4: Results of running `imagenet` with ten different image classification models on the captured image in Figure 5.3.

Table 5.2: IMAGE CLASSIFICATION MODEL RESULTS FROM WILSONVILLE SIMULATION

Model	Primary Classification	Confidence (%)
`AlexNet`	stone wall	25.93
`GoogleNet`	stone wall	33.30
`GoogleNet-12`	bottle	19.43
`ResNet-18`	stone wall	26.51
`ResNet-50`	Chesapeake Bay retriever	16.06
`ResNet-101`	head cabbage	45.12
`ResNet-152`	Rhodesian ridgeback	27.20
`VGG-16`	stone wall	31.42
`VGG-19`	stone wall	18.80
`Inception-v4`	lakeside	32.64

5.1.2 Object Detection

Object detection simulations were performed on the Freiburg Forest dataset and the video that was captured from Memorial Park. In this section, the same source images that were featured in the Section 5.1.1, Figure 5.1 and Figure 5.3, were again applied for the simulation demonstration for object detection.

The `detectnet` program was used to detect objects from images in the Freiburg Forest dataset. The simulations were performed using ten different object detection models. The output from each of these program runs are shown in the montage of images displayed in Figure 5.5. In the figure, it can be seen that no objects were detected from the specific image, as no colored boxes or text callouts appear to indicate an identified object.

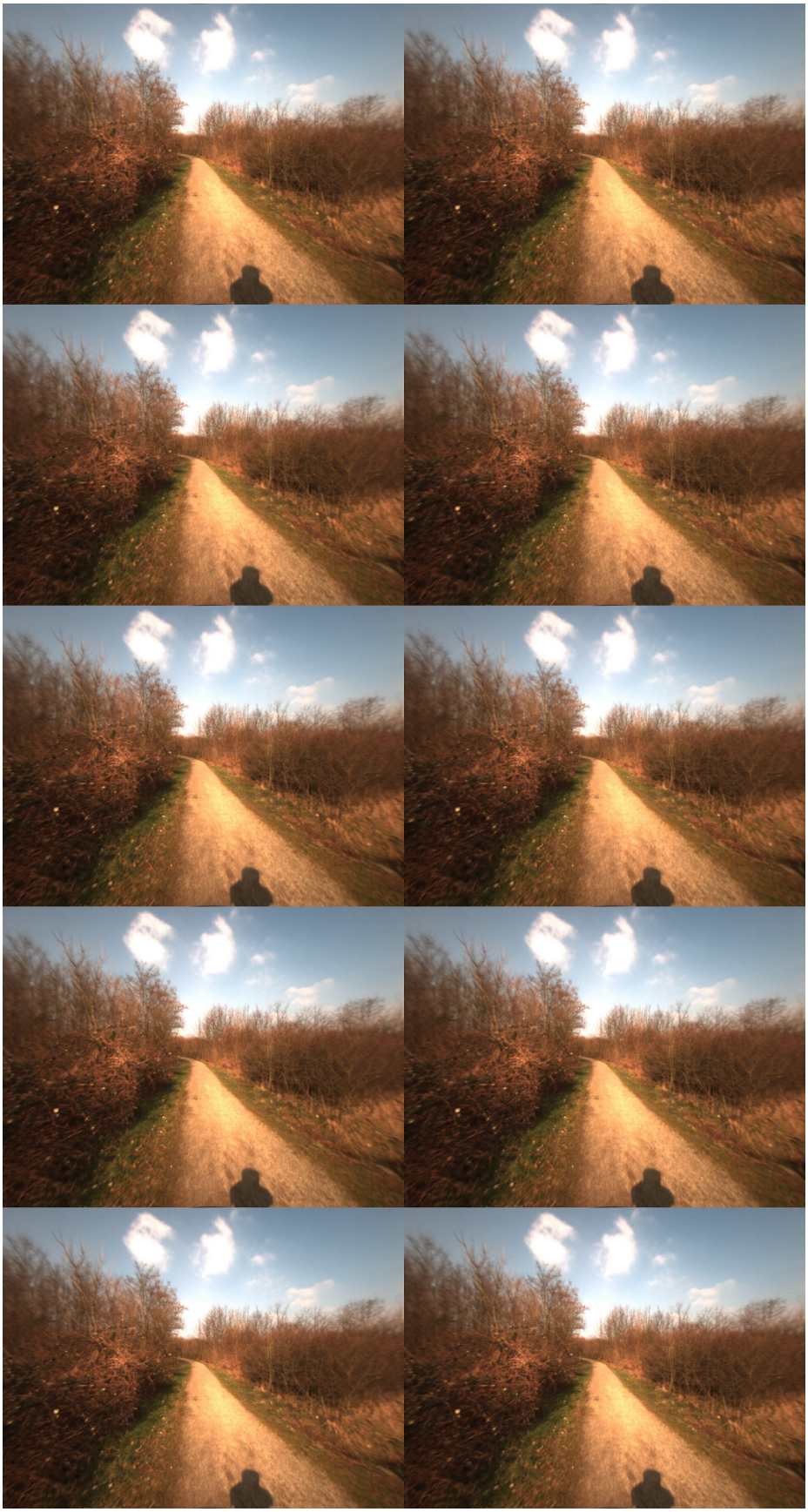


Figure 5.5: Results of running `detectnet` with ten different object detection models on the captured image from the Freiburg Forest dataset in Figure 5.1.

Table 5.3 contains the results for each of the object detection models used as input for the `detectnet` program. The table contains columned categories for the model used, the detected object(s) found, and the confidence interval of each detection. The contents of the table reflect the actual object identification results shown in Figure 5.5, where no objects were detected.

Table 5.3: OBJECT DETECTION MODEL RESULTS FROM FREIBURG FOREST DATASET

Model	**Detections**	**Confidence (%)**
`SSD-Mobilenet-v1`	*N/A*	–
`SSD-Mobilenet-v2`	*N/A*	–
`SSD-Inception-v2`	*N/A*	–
`pednet`	*N/A*	–
`multiped`	*N/A*	–
`facenet`	*N/A*	–
`DetectNet-COCO-Airplane`	*N/A*	–
`DetectNet-COCO-Bottle`	*N/A*	–
`DetectNet-COCO-Chair`	*N/A*	–
`DetectNet-COCO-Dog`	*N/A*	–

Figure 5.3 was used as the input to simulate `detectnet` results using ten different object detection models in a forest scene from Memorial Park. As this required the `detectnet` program to be executed ten distinct times, each execution using a different model as input, ten new images were produced containing object detection results. The output was then combined into a single montage of images, which is displayed in Figure 5.6.

Figure 5.6: Results of running `detectnet` with ten different object detection models on the captured image in Figure 5.3.

Results after running `detectnet` using the ten object detection neural network models on Figure 5.3 are shown in Table 5.4. Of the ten models, only one model detected objects in the image. The model that produced object detections was `multiped`, which reported the detection of two "person" objects. However, there were no humans in the captured image, making these object detections false-positives. These false-positive object detections can also be seen in the third row of the first column image cell in Figure 5.6, where the two "person" detections are outlined by small cyan-colored boxes.

Table 5.4: OBJECT DETECTION MODEL RESULTS FROM WILSONVILLE SIMULATION

Model	**Detections**	**Confidence (%)**
`SSD-Mobilenet-v1`	*N/A*	–
`SSD-Mobilenet-v2`	*N/A*	–
`SSD-Inception-v2`	*N/A*	–
`pednet`	*N/A*	–
`multiped`	person (x2)	62.3, 63.8
`facenet`	*N/A*	–
`DetectNet-COCO-Airplane`	*N/A*	–
`DetectNet-COCO-Bottle`	*N/A*	–
`DetectNet-COCO-Chair`	*N/A*	–
`DetectNet-COCO-Dog`	*N/A*	–

Due to the lack of positive object detection results from the plain forest scenes in Figure 5.1 and Figure 5.3, another image capture from the video that was taken at Memorial Park was used as input for the `detectnet` program for simulation. This image featured a scene containing objects that human intelligence would easily identify in a forest park setting, and is displayed in Figure 5.7.

Figure 5.7: Still image taken from a video capture of a park bench scene at Memorial Park in Wilsonville, OR.

Figure 5.8 contains a montage of the `detectnet` program results from all ten object detection models applied to 5.7. It can be seen from the figure that some models that were used as program input detected one or more objects in the image, while other models did not detect any objects at all.

Figure 5.8: Results of running `detectnet` with ten different object detection models on the captured image in Figure 5.7.

Object detection text output from `detectnet` after being run with the inputs of the image in Figure 5.3 and the ten object detection models are shown in Table 5.5. Four of the ten models failed to detect any objects in the image whatsoever. The `SSD-Mobilenet-v1`, `SSD-Mobilenet-v2`, and `SSD-Mobilenet-v3` networks each identified a single "bench" object. A "person" object was detected by the `multiped` network. The last two models that resulted in detections, whose names perhaps may contain the most direct indicators as to the variety of objects that they were trained to detect, `DetectNet-COCO-Airplane` and `DetectNet-COCO-Chair`, detected an "airplane" object and two "chair" objects, respectively.

Table 5.5: OBJECT DETECTION MODEL RESULTS FROM WILSONVILLE SIMULATION, PARK BENCH SCENE

Model	Detections	Confidence (%)
`SSD-Mobilenet-v1`	bench	75.2
`SSD-Mobilenet-v2`	bench	95.2
`SSD-Inception-v2`	bench	97.3
`pednet`	*N/A*	–
`multiped`	person	63.8
`facenet`	*N/A*	–
`DetectNet-COCO-Airplane`	airplane	50.4
`DetectNet-COCO-Bottle`	*N/A*	–
`DetectNet-COCO-Chair`	chair (x2)	72.0, 52.8
`DetectNet-COCO-Dog`	*N/A*	–

5.1.3 Image Segmentation

Image segmentation simulations were performed on the Freiburg Forest image dataset and Memorial Park video. The same source images that were used in Sections 5.1.1 and 5.1.2, Figure 5.1 and Figure 5.3, were again used for image segmentation.

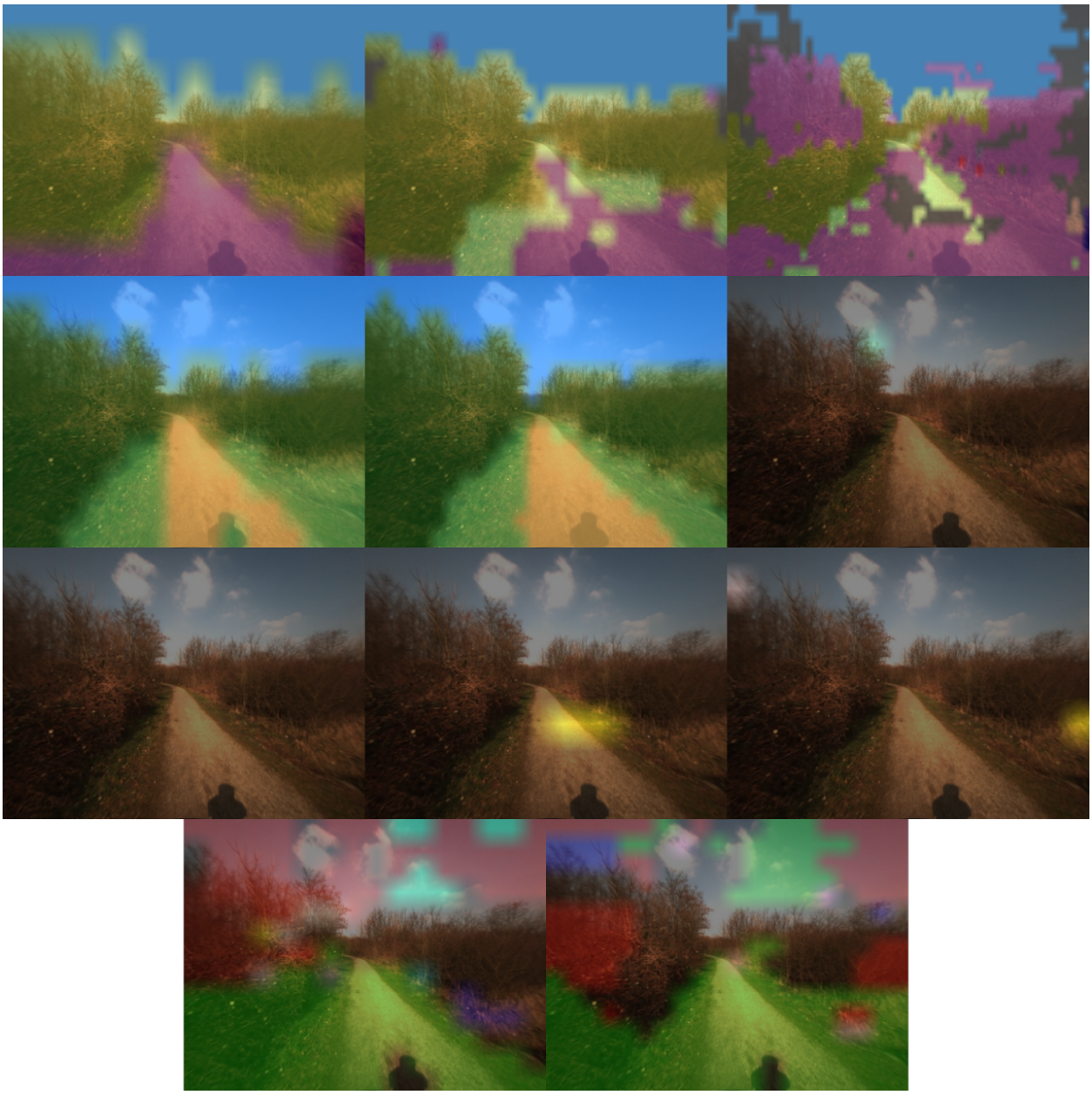


Figure 5.9: Results of running `segnet` with eleven different image segmentation models on the captured image from the Freiburg Forest dataset in Figure 5.1.

The `segnet` program was used to detect objects from images in the Freiburg Forest dataset. The simulations were performed using eleven different image segmentation models. The output from each of these program runs are shown in the montage of images displayed in Figure 5.9. The figure features pixels that have been classified with a colorized overlay. In other words, the pixels that have been shaded a different color than the original image in Figure 5.1 have been identified as being something understood by the deep neural network model via semantic segmentation, which then produced a color mask to represent each classified pixel. Segmented color masks for each individual image cell in Figure 5.9 are displayed in Figures C.1 through C.11 within Appendix C.

Table 5.6 contains image segmentation results for all of the models that were used as input when running `segnet` with the image in Figure 5.1. While `imagenet` and `detectnet` both output confidence intervals when reporting their respective image classification and object detection reporting, `segnet` did not. In terms of image segmentation results, ignoring other output parameters such as color overlay and color mask, the `segnet` program output simply reported the classification of each pixel in the processed image. Color legends for all of the image segmentation models used with the `segnet` program appear in Appendix B.

Table 5.6: IMAGE SEGMENTATION MODEL RESULTS FROM THE FREIBURG FOREST DATASET

Model	Classifications
FCN-ResNet18-Cityscapes-512x256	ground, road, vegetation, sky
FCN-ResNet18-Cityscapes-1024x512	ground, building, road, vegetation, terrain, sky
FCN-ResNet18-Cityscapes-2048x1024	ground, road, building, fence, vegetation, terrain, sky, person, car
FCN-ResNet18-DeepScene-576x320	trail, grass, vegetation, sky
FCN-ResNet18-DeepScene-864x480	trail, grass, vegetation, sky
FCN-ResNet18-MHP-512x320	background, jacket/coat
FCN-ResNet18-MHP-640x360	background
FCN-ResNet18-VOC-320x320	background, horse
FCN-ResNet18-VOC-512x320	background, horse, person
FCN-ResNet18-SUN-512x400	other, wall, floor, cabinet/shelves/bookshelf/dresser, table, bed/pillow, sofa, window, ceiling, fridge
FCN-ResNet18-SUN-640x512	other, floor, bed/pillow, window, blinds/curtain, fridge

The video captured at Memorial Park in Wilsonville, OR was used as `segnet` input for image segmentation simulation. These simulations were performed using the same eleven image segmentation models that were used previously on the Freiburg Forest dataset. The output from each of these program runs are shown in the montage of images displayed in Figure 5.10. The figure features pixels that have been classified with a colorized overlay, and can be compared with the original image in Figure 5.3. Segmented color masks for each individual image in Figure 5.10 are displayed in Appendix C, Figures C.12 - C.22.

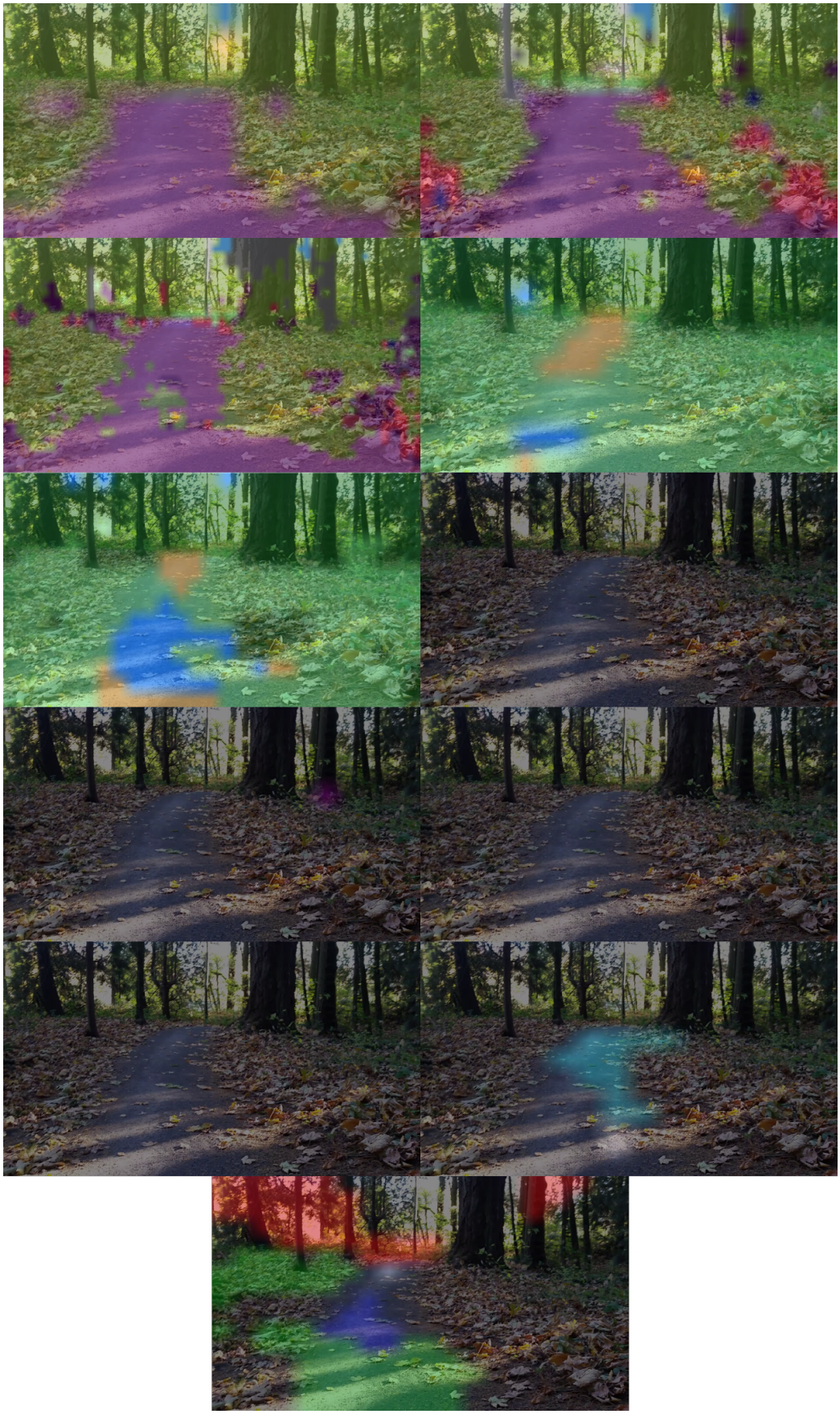


Figure 5.10: Results of running `segnet` with eleven different image segmentation models on the captured image in Figure 5.3.

Table 5.7 contains image segmentation results for the eleven models that were used as input when running `segnet` with the image in Figure 5.3. Color legends for all of the image segmentation models used with the `segnet` program appear in Appendix B.

Table 5.7: IMAGE SEGMENTATION MODEL RESULTS FROM WILSONVILLE SIMULATION

Model	Classifications
`FCN-ResNet18-Cityscapes-512x256`	ground, road, fence, traffic_light, vegetation, sky, person
`FCN-ResNet18-Cityscapes-1024x512`	ground, road, sidewalk, building, fence, pole, traffic_light, vegetation, terrain, sky, person
`FCN-ResNet18-Cityscapes-2048x1024`	ground, road, sidewalk, building, sidewalk, pole, traffic_light, terrain, vegetation, sky, person, car
`FCN-ResNet18-DeepScene-576x320`	trail, grass, vegetation, sky
`FCN-ResNet18-DeepScene-864x480`	trail, grass, vegetation, sky
`FCN-ResNet18-MHP-512x320`	background
`FCN-ResNet18-MHP-640x360`	background, foot
`FCN-ResNet18-VOC-320x320`	background
`FCN-ResNet18-VOC-512x320`	background
`FCN-ResNet18-SUN-512x400`	other, sofa, table
`FCN-ResNet18-SUN-640x512`	other, floor, bed/pillow, window, table, blinds/curtain

The image classification, object detection, and image segmentation simulations performed on the Freiburg Forest dataset and the Memorial Park video provided output which demonstrated that `imagenet`, `detectnet`, and `segnet` could be used in field experiments with a live video capture image processing system mounted on the SAR-UGV.

5.2 Experimental Results

The image processing system hardware was mounted on the SAR-UGV. Field experiments occurred in the parking lot of the Oregon Tech Portland-Metro campus, as resources were unavailable to deploy the SAR-UGV in a forested environment. Multiple subsystems on the vehicle were tested simultaneously by other students involved with the SAR-UGV project. During this time, the image processing system experiments of image classification, object detection, and image segmentation were also conducted. Figure 5.11 and Figure 5.12 show two example frame captures of live video that was shot from the SAR-UGV image processing system during experimentation.

Figure 5.11: Frame taken from a video clip recorded by the image processing system on the SAR-UGV during field trials in the Oregon Tech parking lot.

Figure 5.12: Another frame taken from a video clip recorded by the image processing system on the SAR-UGV during field trials in the Oregon Tech parking lot.

5.2.1 Image Classification

Image classification was performed by launching the `imagenet` program with the inputs of the image in Figure 5.11 and ten different image classification models produced ten output images. As with previous demonstrations resulting in output containing multiple images, these images were combined into a montage. This montage is displayed in Figure 5.13. The reported classifications from each model are contained in the top-left of each image cell in Figure 5.13.

Figure 5.13: Results of running `imagenet` with ten different image classification models on the captured image in Figure 5.11.

Table 5.8 contains the classification results after using the `imagenet` program with all ten image classification models. The confidence interval that was reported alongside each primary classification is also contained in Table 5.8.

Table 5.8: IMAGE CLASSIFICATION MODEL RESULTS FROM OREGON TECH PARKING LOT

Model	Classifications	Confidence (%)
`AlexNet`	unicycle	9.44
`GoogleNet`	garbage truck	12.79
`GoogleNet-12`	dog	93.90
`ResNet-18`	fur coat	9.14
`ResNet-50`	unicycle	62.50
`ResNet-101`	ambulance	22.38
`ResNet-152`	unicycle	19.71
`VGG-16`	crutch	24.96
`VGG-19`	triumphal arch	8.59
`Inception-v4`	minibu	11.87

The captured video frame from the SAR-UGV in Figure 5.12 was also processed for image classification. As with all previous image classification simulations, and the previous field experiment, `imagenet` was run with the video frame image and the ten image classification network models. The ten images that were output by these executions of the `imagenet` program were combined into a single montage image. This image is displayed in Figure 5.14. The montage contains the primary classification data on the top-left of each image cell as an overlay.

Figure 5.14: Results of running `imagenet` with ten different image classification models on the captured image in Figure 5.12.

Image classification results after running `imagenet` on Figure 5.12 with all ten image classification models are contained in Table 5.9. The table reports the image classification model used, the primary classification of the image, and the confidence interval.

Table 5.9: IMAGE CLASSIFICATION MODEL RESULTS FROM OREGON TECH PARKING LOT, PART TWO

Model	Classifications	Confidence (%)
`AlexNet`	submarine	32.50
`GoogleNet`	traffic light	84.67
`GoogleNet-12`	vehicle	67.24
`ResNet-18`	traffic light	28.66
`ResNet-50`	obelisk	24.47
`ResNet-101`	traffic light	55.08
`ResNet-152`	traffic light	21.30
`VGG-16`	traffic light	23.46
`VGG-19`	traffic light	35.82
`Inception-v4`	maze	69.97

5.2.2 Object Detection

The `detectnet` program was run on the captured video frame from the SAR-UGV in Figure 5.11 for object detection. Each object detection model was used as additional input when launching `detectnet` in separate application runs. The ten images that were output as a result of running `detectnet` with all ten models on the captured video frame appear in the montage in Figure 5.15. The objects detected by each model, for the models that were able to successfully detect one or more objects, appear within each of their respective image cells as an overlay. The confidence interval output for objects that were detected are included with each overlay in Figure 5.15.

Figure 5.15: Results of running `detectnet` with ten different object detection models on the captured image in Figure 5.11.

Table 5.10 contains object detection results for each of the models that were used as input for the `detectnet` program with Figure 5.11. In the table, columned categories for the model used, the detected object(s) found, and the confidence interval of each detection are included.

Table 5.10: OBJECT DETECTION MODEL RESULTS FROM OREGON TECH PARKING LOT

Model	Detections	Confidence (%)
`SSD-Mobilenet-v1`	person (x2)	86.4, 59.3
`SSD-Mobilenet-v2`	person (x2)	81.4, 65.6
`SSD-Inception-v2`	person (x3); skateboard	88.2, 72.9, 55.7; 58.4
`pednet`	person (x4)	79.0, 263.5, 51.8, 113.97
`multiped`	person (x5)	70.5, 84.0, 54.1, 93.7, 82.2
`facenet`	face (x2)	60.7, 90.9
`DetectNet-COCO-Airplane`	*N/A*	–
`DetectNet-COCO-Bottle`	*N/A*	–
`DetectNet-COCO-Chair`	*N/A*	–
`DetectNet-COCO-Dog`	*N/A*	–

The `detectnet` program was once again executed using the same ten object detection models on Figure 5.12. This execution of `detectnet` on the captured video frame produced ten different images, each containing distinct overlay data where objects had been detected. The output images displaying overlays of detected objects and their respective confidence intervals are shown in the montage, Figure 5.16.

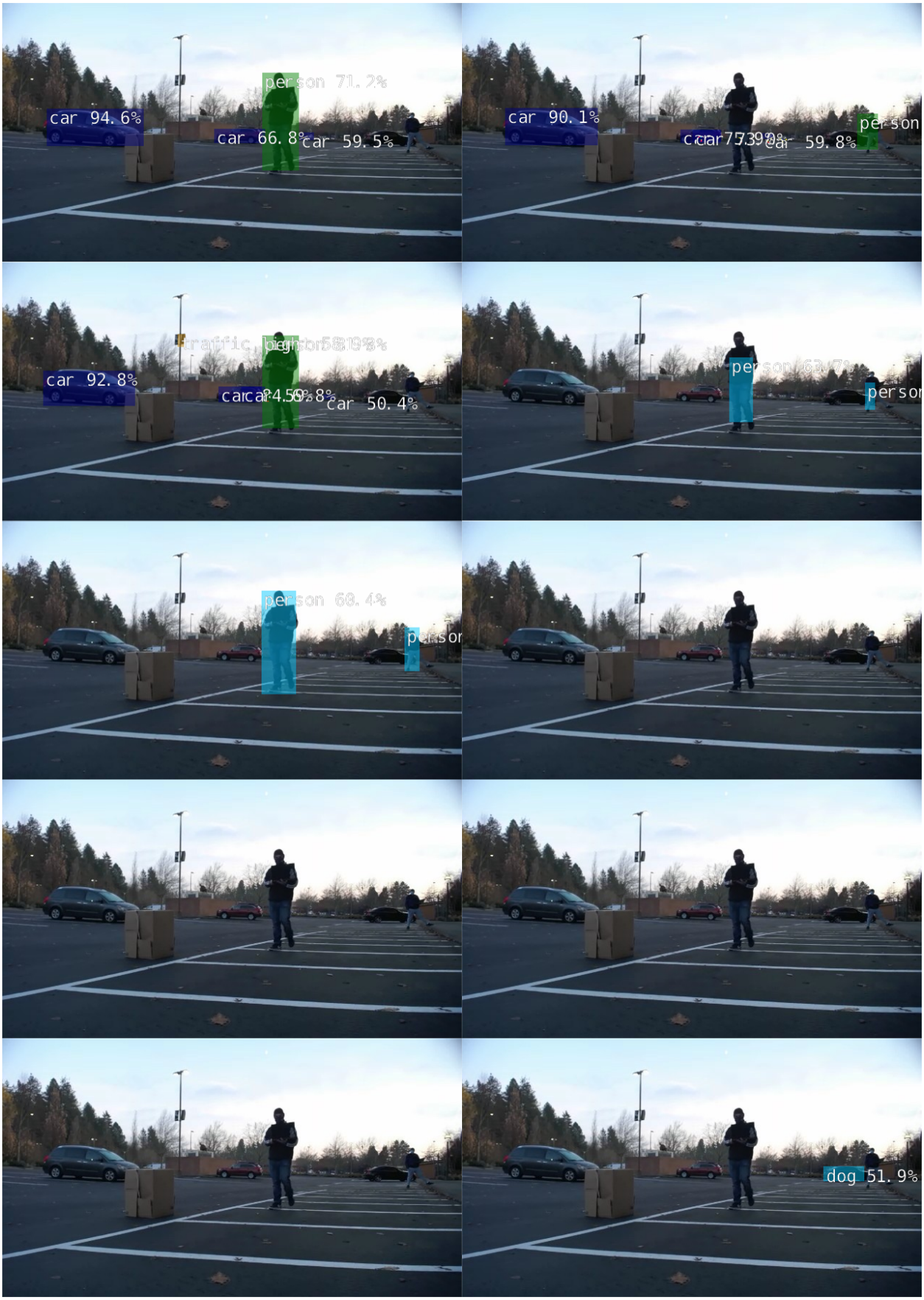


Figure 5.16: Results of running `detectnet` with ten different object detection models on the captured image in Figure 5.12.

The full listing of object detection results is displayed in Figure 5.16 includes confidence intervals that were reported in the `detectnet` Terminal output, but not visible in the image, is shown in Table 5.11. Columns for reported program output include the model used, the detected object(s) found, and the respective confidence interval of each detection.

Table 5.11: OBJECT DETECTION MODEL RESULTS FROM OREGON TECH PARKING LOT, PART TWO

Model	Object(s) Detected	Confidence (%)
`SSD-Mobilenet-v1`	car (x3); person	94.6, 66.8, 59.5; 71.2
`SSD-Mobilenet-v2`	car (x4); person	90.1, 75.9, 73.0, 59.8; 67.2
`SSD-Inception-v2`	car (x4); person; traffic light	92.8, 84.6, 59.8, 50.4; 81.3; 58.9
`pednet`	person (x2)	63.7, 117.5
`multiped`	person (x2)	60.4, 60.5
`facenet`	*N/A*	–
`DetectNet-COCO-Airplane`	*N/A*	–
`DetectNet-COCO-Bottle`	*N/A*	–
`DetectNet-COCO-Chair`	*N/A*	–
`DetectNet-COCO-Dog`	dog	51.9

5.2.3 Image Segmentation

The `segnet` program was executed on the captured video frame in Figure 5.11. The eleven image segmentation models that were included with the jetson-inference software were used as additional input. As running `segnet` with each model produced a separate image, the output from all of the program runs were combined into a single montage, which is displayed in Figure 5.17. The figure shows classified pixels as colorized overlay. This can be compared with the original image in Figure 5.11. Appendix C, Figures C.23 through C.33, contains images of the segmented color masks for all of the individual images in Figure 5.17.

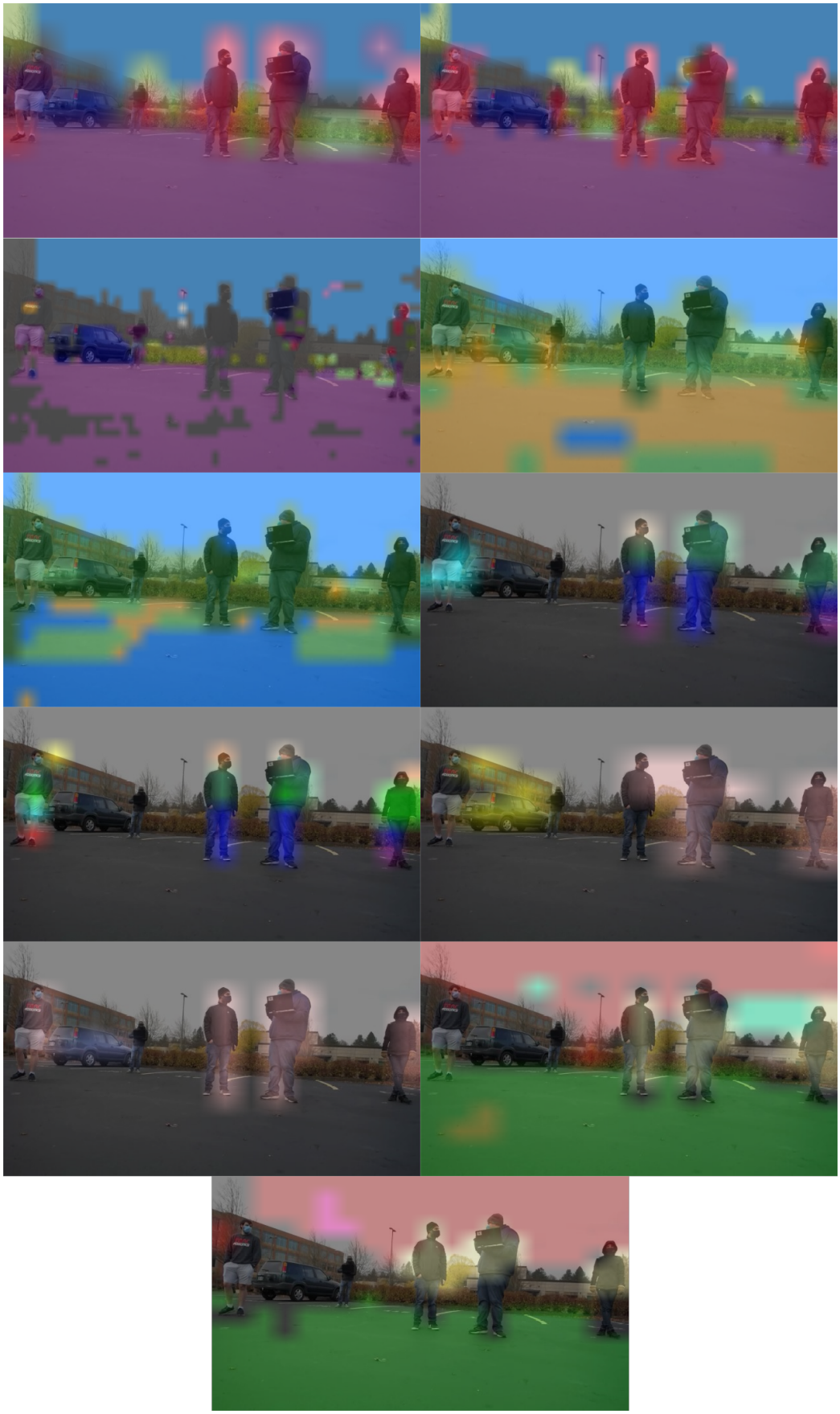


Figure 5.17: Results of running `segnet` with eleven different Image segmentation models on the captured image in Figure 5.11.

Pixel classification results after running `segnet` with the eleven image segmentation models on Figure 5.11 are contained in Table 5.12. Appendix B contains the color legends for all of the image segmentation models utilized as `segnet` input.

Table 5.12: IMAGE SEGMENTATION MODEL RESULTS FROM OREGON TECH PARKING LOT

Model	Classifications
`FCN-ResNet18-Cityscapes-512x256`	ground, road, building, vegetation, terrain, sky, person, car
`FCN-ResNet18-Cityscapes-1024x512`	ground, road, building, sidewalk, fence, traffic_light, vegetation, terrain, sky, person, car
`FCN-ResNet18-Cityscapes-2048x1024`	ground, road, sidewalk, building, wall, pole, fence, traffic_light, vegetation, terrain, sky, person, car, truck, train
`FCN-ResNet18-DeepScene-576x320`	trail, grass, vegetation, obstacle, sky
`FCN-ResNet18-DeepScene-864x480`	trail, grass, vegetation, obstacle, sky
`FCN-ResNet18-MHP-512x320`	background, face, jacket/coat, pants, shorts, shoe/boot
`FCN-ResNet18-MHP-640x360`	background, face, hair, arm, shirt, jacket/coat, pants, shorts, shoe/boot
`FCN-ResNet18-VOC-320x320`	background, horse, person
`FCN-ResNet18-VOC-512x320`	background, car, chair, person
`FCN-ResNet18-SUN-512x400`	other, wall, floor, table, window, picture/tv/mirror, ceiling, person
`FCN-ResNet18-SUN-640x512`	other, wall, floor, blinds/curtain, person

As a final display of image segmentation results during field experiment, `segnet` was run on the captured video frame in Figure 5.12. All eleven image segmentation models that have been discussed throughout this document were added as input during execution of the `segnet` program. The eleven images that were produced as output from the `segnet` program executions were combined into the image montage in Figure 5.18. The `segnet` default colorized image segmentation overlay is displayed within each image cell in the figure. Color mask images for each individual image segmentation model's output are contained in Appendix C, Figures C.34 - C.44.

Table 5.13 contains pixel classification results of running `segnet` with the eleven image segmentation models on the latter of the two example images from the Oregon Tech parking lot, Figure 5.12. Columns in the table include the image segmentation model used and the classifications that were output. Appendix B contains the color legends for all of the image segmentation models that were used as `segnet` input.

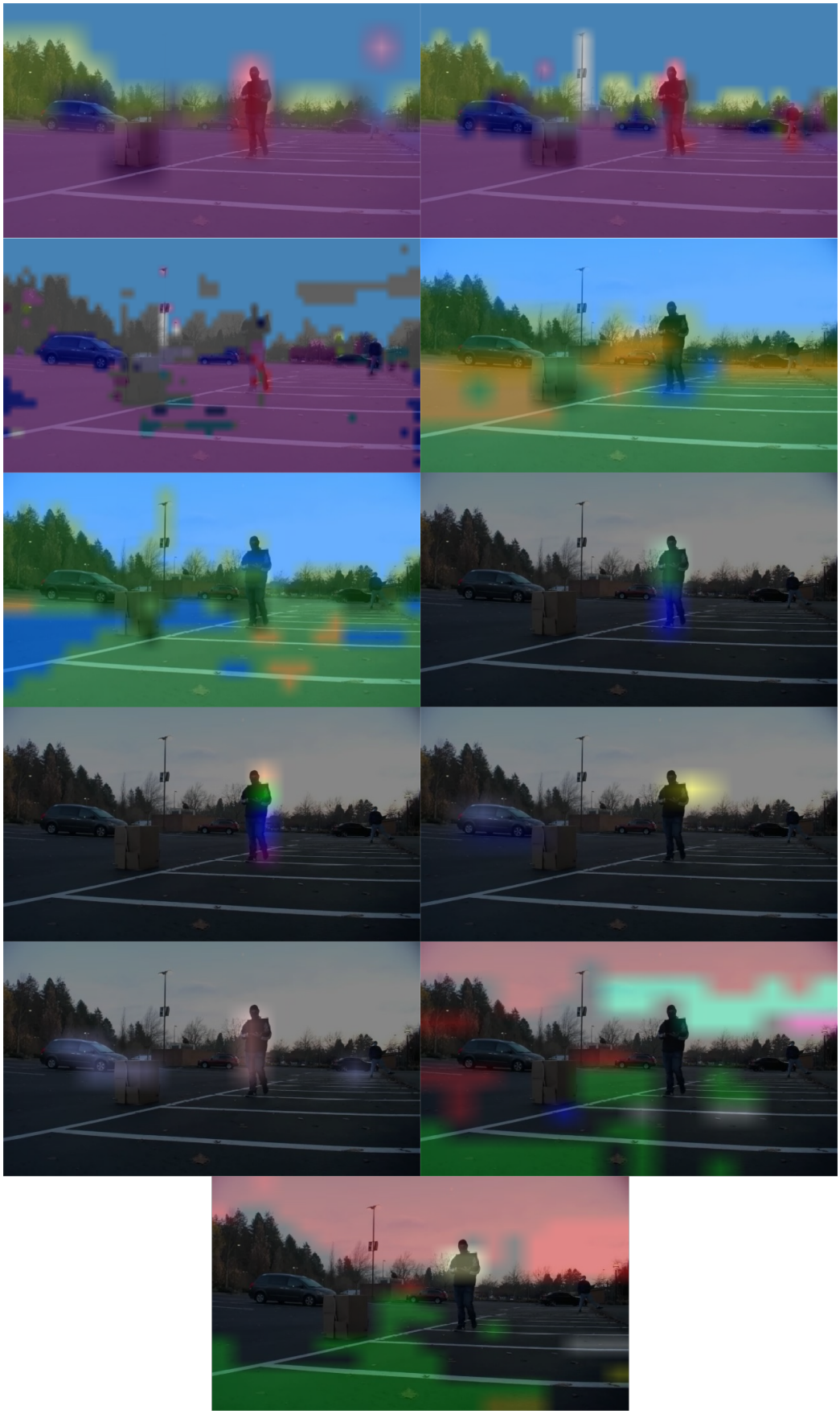


Figure 5.18: Results of running `segnet` with eleven different image segmentation models on the captured image in Figure 5.12.

Table 5.13: IMAGE SEGMENTATION MODEL RESULTS FROM OREGON TECH PARKING LOT, PART TWO

Model	Classifications
FCN-ResNet18-Cityscapes-512x256	ground, road, sidewalk, building, vegetation, sky, person, car
FCN-ResNet18-Cityscapes-1024x512	ground, road, building, sidewalk, wall, pole, vegetation, sky, person, car
FCN-ResNet18-Cityscapes-2048x1024	ground, road, sidewalk, building, wall, pole, fence, vegetation, sky, person, car, truck, train
FCN-ResNet18-DeepScene-576x320	trail, grass, vegetation, obstacle, sky
FCN-ResNet18-DeepScene-864x480	trail, grass, vegetation, sky
FCN-ResNet18-MHP-512x320	background, jacket/coat, pants
FCN-ResNet18-MHP-640x360	background, hair, shirt, jacket/coat, pants, shoe/boot
FCN-ResNet18-VOC-320x320	background, car, sofa, horse
FCN-ResNet18-VOC-512x320	car, person
FCN-ResNet18-SUN-512x400	other, wall, floor, sofa, table, blinds/curtain, ceiling, bathtub
FCN-ResNet18-SUN-640x512	other, wall, floor, cabinet/shelves/bookshelf/dresser, table, window, person, lamp

5.3 Performance Characterization

The ROS launch files for the `imagenet`, `detectnet`, and `segnet` programs were all profiled using `Callgrind`. This particular profiling tool "records call history among functions in a program's run as a call-graph. By default, the collected data consists of the number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the numbers of such calls [53]." The data collected from `Callgrind` was then imported into `Kcachegrind` for visual analysis. `KCachegrind` is described as

"a profile data visualization tool, used to determine the most time consuming parts in the execution of a program [54]."

Collecting program profiling data required additional preparation of the ROS launch files before the programs could be profiled. Specifically, the `node` tags within the `imagenet`, `detectnet`, and `segnet` ROS launch files were all modified so that they would output `Callgrind` profile data when the programs were launched using the `roslaunch` command. Listing 5.1, Listing 5.2, and Listing 5.3 each show the specific line of code that was modified in each of the three files. The modifications made for program profiling that are displayed in Listing 5.1, Listing 5.2, and Listing 5.3 do not appear in their respective full source code listings shown in Listing A.17, Listing A.21, and Listing A.25 within Appendix A.

```
1  <node pkg="ros_deep_learning" type="imagenet"
2      name="imagenet_camera_1" output="screen"
3      launch-prefix="valgrind --tool=callgrind
4      --callgrind-out-file='callgrind.imagenet_camera_1.%p'" >
```

Listing 5.1: Snippet of the ROS launch file in A.17 after modification for profiling with `Callgrind` and `Kcachegrind`.

```
1    <node pkg="ros_deep_learning" type="detectnet"
2        name="detectnet_camera_1" output="screen"
3        launch-prefix="valgrind --tool=callgrind
4        --callgrind-out-file='callgrind.detectnet_camera_1.%p'" >
```

Listing 5.2: Snippet of the ROS launch file in A.21 after modification for profiling with `Callgrind` and `Kcachegrind`.

```
1    <node pkg="ros_deep_learning" type="segnet"
2        name="segnet_camera_1" output="screen"
3        launch-prefix="valgrind --tool=callgrind
4        --callgrind-out-file='callgrind.segnet_camera_1.%p'" >
```

Listing 5.3: Snippet of the ROS launch file in A.25 after modification for profiling with `Callgrind` and `Kcachegrind`.

Once the ROS launch files had been run using the `roslaunch` command, the `Callgrind` data could be opened with `Kcachegrind` using the following command structure:

```
$ kcachegrind  /.ros/callgrind.<program-name>.<process-id>
```

For example, if the imagenet ROS launch file associated with the first camera connected to the Jetson TX2 was run for profiling, it could be given a process identification number (PID) of 12345 by the operating system, and Callgrind would produce a file called callgrind.imagenet_camera_1.12345. The profile data could then be visualized using the command:

```
$ kcachegrind  /.ros/callgrind.imagenet_camera_1.12345
```

It can also be seen from the above command that Kcachegrind retrieved the Callgrind profile data file from the /.ros/ directory. Hence, this is the location where all Callgrind data was saved by default.

5.3.1 Image Classification

Kcachegrind analysis of imagenet initially focused on the _dl_sysdep_start function. This particular function was shown only to be called once during program execution, and reported a cost estimate of 428 cycles. This starting function contributed very little to the overall cost estimate of imagenet, which produced a total cost estimate of 3,803,545,239 cycles. The most expensive function in imagenet, according to the data viewed in Kcachegrind, was a function called <cycle 17>, which reported a cost estimate of 2,290,839,862 processing cycles while being called 18 times. Hence, 60.2% of the total imagenet processing cost was caused by this looping part of the program. A screenshot of Kcachegrind while observing the Callgrind data exported after running imagenet can be seen in Figure 5.19, where the *Callee Map* and *Call Graph* visualizations of the _dl_sysdep_start function are visible in the interface.

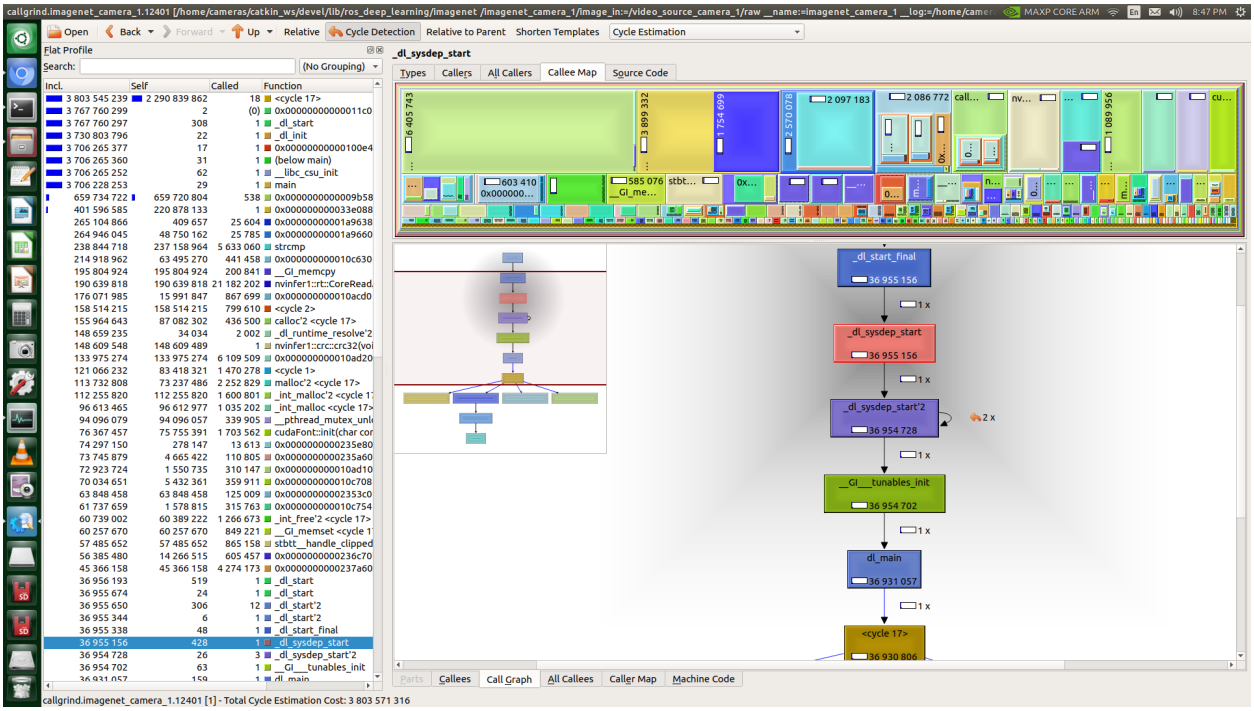


Figure 5.19: Screenshot while running `Kcachegrind` on profile data from `imagenet` on the Jetson TX2.

5.3.2 Object Detection

Output of `Callgrind` profiling data while running the modified `detectnet` ROS launch file was imported into `Kcachegrind` for performance analysis. Again, by default, `Kcachegrind` initially focused on the `_dl_sysdep_start` function, which made it apparent that this was the primary starting function of the deep learning software programs found at jetson-inference. Like the `Kcachegrind` profile analysis of `imagenet`, the `_dl_sysdep_start` function was only called one time, and contributed a cost estimate of 428 processing cycles while running `detectnet`. The total cost estimate of running `detectnet` was 6,025,123,961 processing cycles, where the most expensive function was a function called `<cycle 20>`, which reported a cost estimate of 4,321,831,358 cycles while being called 18 times, resulting in 71.7% of the total program processing cost. Figure 5.20 displays a screenshot of the `Kcachegrind` interface while observing the *All Callers* and *All Callees* tabs for the `_dl_sysdep_start` function of `detectnet`.

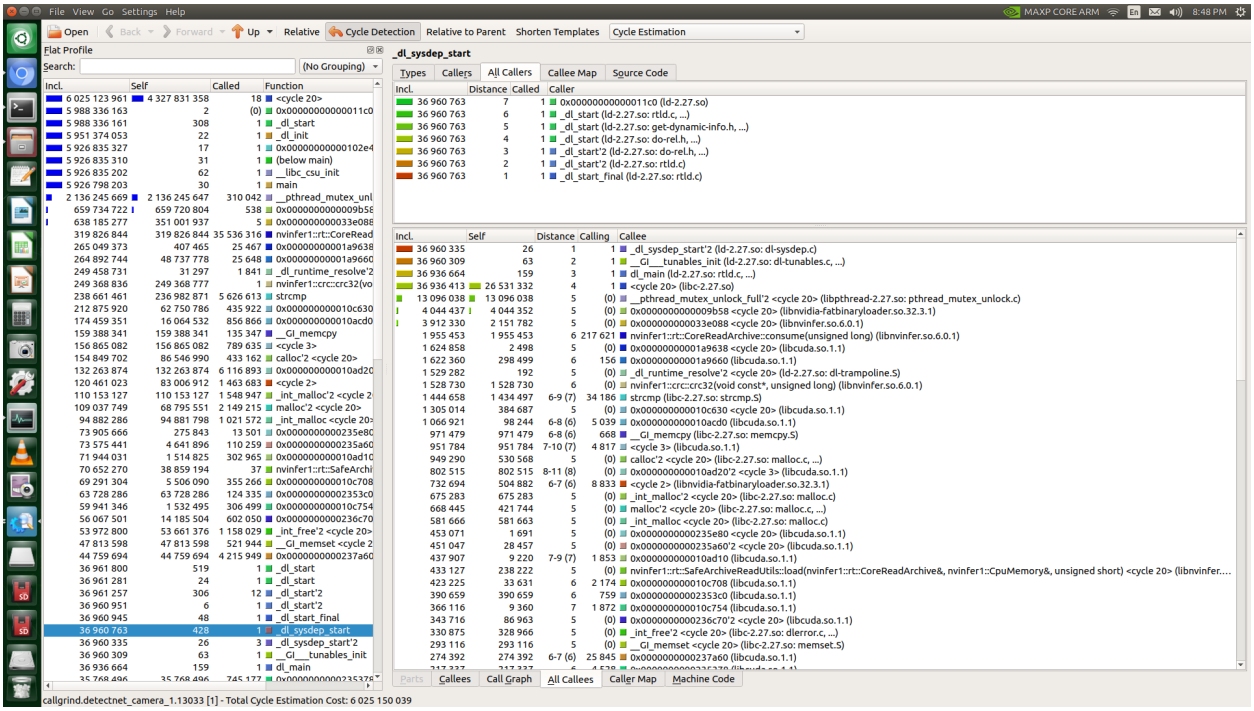


Figure 5.20: Screenshot while running `Kcachegrind` on profile data from `detectnet` on the Jetson TX2.

5.3.3 *Image Segmentation*

Profiling analysis using `Callgrind` and `Kcachegrind` was lastly performed on `segnet`. Like both `imagenet` and `detectnet`, `Kcachegrind` focused on the `segnet` program's function `_dl_sysdep_start` initially, where the particular function was called only one time in total, and contributed a cost estimate of 428 processing cycles. Cross comparison revealed that the source of this function was the same for `imagenet`, `detectnet`, and `segnet`. All three of these programs were found to use the same `_dl_sysdep_start` function, which was sourced by the same shared object library, `ld-2-27.so`. Running `segnet` resulted in a total processing cost estimate of 4,537,566,288 cycles. The most expensive function was `<cycle 20>`, which reported a cost estimate of 2,466,768,089 cycles while being called 18 times during program execution. This looping function accounted for 54.4% of the total program processing cost of `segnet`. A screenshot of `Kcachegrind` profile analysis of `segnet` on the Jetson TX2, in which the Callee Map and Call Graph tabs for the `_dl_sysdep_start` function were observed, appears in Figure 5.21.

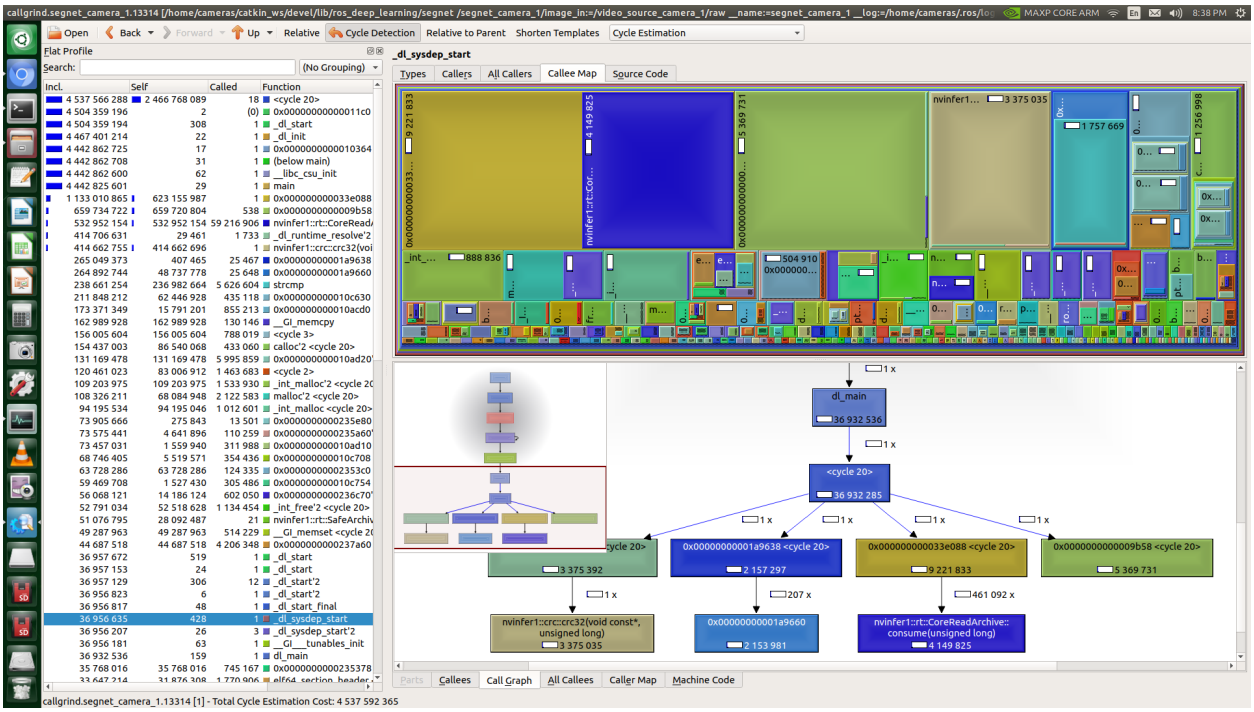


Figure 5.21: Screenshot while running `Kcachegrind` on profile data from `segnet` on the Jetson TX2.

System profiling using the `Callgrind` and `Kcachegrind` tools was shown to be effective in characterizing the performance of the `imagenet`, `detectnet`, and `segnet` applications when used via the `roslaunch` command. These tools could be used to further debug the programs or any of the libraries involved with the software found at jetson-inference, particularly in the instances that operations involving large image resolution resulted in high system memory consumption, and often program failure. If the scope of this project had involved modification of C++ or Python source code in order to develop improvements to the deep learning software found at jetson-inference, `Callgrind` and `Kcachegrind` would have been used more extensively for system performance analysis during the development process.

CHAPTER 6 Conclusions

The final chapter in this thesis provides a summary of conclusions based on the results that were achieved during the process of engineering the image processing system that is the primary focus of this document. The design, development, and implementation of the image processing system for the search and rescue unmanned ground vehicle are reviewed in terms of its contributions and future work.

6.1 Contributions

With the integration of the image processing system, the SAR-UGV has been provided with the added ability to visually sense the world which surrounds it. This was performed by image capture through the use of multiple cameras. The utilization of camera hardware and accompanying software programs accomplished the tasks of acquiring both live video and still images. By image capture alone, sufficient data was provided for manual analysis regarding the environment surrounding the SAR-UGV.

The captured image data, however, was shown to be processed through advanced computer vision software systems with deep neural network model architectures to accomplish specific automated tasks. These included solutions to perform the highly sophisticated problems of image classification, object detection, and image segmentation. While each of the solutions discussed throughout this thesis carries both strengths and weaknesses, their use as part of the image processing system each provide a particular capability that the SAR-UGV

would otherwise not be able to perform without the collection and processing of image data from the perspective of the vehicle.

The additional post-capture processed data produced by the `imagenet`, `detectnet`, and `segnet` software programs was integrated into the SAR-UGV ROS network, which made image classification, object detection, and image segmentation data readily available to other subsystems. By subscribing to the ROS topics published by the image processing system, other subsystems could achieve a greater understanding of their environment. In particular, the SAR-UGV autonomous control system has become furnished with additional data that could be used to make decisions regarding the state of the vehicle based on its surroundings, from navigating a trail to the detecting a lost person.

In sum, the contributions of the image processing system enable the SAR-UGV to better perform the mission critical functions that were part of the vehicle's original concept.

6.2 Future Work

Like any technology, the image processing system for the SAR-UGV can be further improved and optimized to better perform its intended functions. This section discusses future work to the image processing system that could be explored by engineers to further improve upon such a system.

In regard to image and video capture, the most fundamental of tasks for the image processing system, data was shown to be saved to an image file, video file, or output directly to a monitor connected to the system. This could be further extended to live video streaming. Such an improvement could implemented to send live video data directly to the Ground Station via the to the SAR-UGV radio communications system. This would require sufficient bandwidth for the radio communications system in order for the Ground Station to receive

live image data. The bandwidth requirement would be largely dependent on the quality of the video stream (i.e., image resolution) and frame rate in addition to data being transmitted over the radio communications system by other SAR-UGV subsystems.

The image classification, object detection, and image segmentation by deep neural networks showed to be effective tools for an autonomous vehicle. However, this thesis merely demonstrated how the image processing system could perform these tasks with a wide variety of pre-trained models. To properly understand its surroundings and better perform SAR-UGV missions, additional neural network training is absolutely mandatory. Image classification model training would benefit from overall scene understanding to ensure that it operates in a forested environment. Object detection model training could be performed on pictures of a particular lost person, or mission target, so that he or she could be distinguished from other persons that the SAR-UGV may encounter during operation. Furthermore, other objects of interest could be used in object detection model training, such as landmarks in a particular location to aid in location tracking. Lastly, image segmentation, the best candidate for navigation and maintaining the SAR-UGV on the intended trail, would benefit most from additional neural network model training. During simulations and field experiments, it was shown that the `segnet` program could be operated with input of the image segmentation models build from the DeepScene dataset. While this dataset was based entirely from images in outdoor and forested environments, it was observed that the results were far from perfect, and that the results varied between the DeepScene dataset-based models pre-trained at different image resolutions. Additional neural network training based on the DeepScene dataset, including images specific to mission locations and varying weather conditions, could prove beneficial to effective image segmentation in forested environments.

The ROS topics published by the image processing system could undergo further enhancement. For image classification and object detection data produced by `imagenet` and `detectnet`, respectively, the ROS topics are rather straightforward. These topics can sim-

ply be refined to the overall understanding of a single scene, or to detect specific objects understood by the particular neural network being model used. Again, image segmentation tasks performed by the image processing system could be vastly improved by ROS software optimization. Strictly regarding navigation, the pixel-level classification output over the ROS network could be improved by including data specific only to a region of interest, which would aid in determining whether or not the area of SAR-UGV intended direction of movement is actually traversable. While the ROI could be determined directly by the `segnet` source code prior to the integration of ROS output, ROI determination could be just as effectively be pre-defined in the ROS output itself. The ROI determination could be applied to its `color_mask` topic to make the distinction of a traversable area ahead based on a specific region of pixels, and to output an additional ROS topic that publishes data simply refined to messages of "traversable" or "non-traversable" to the autonomous control system. Implementation of such a feature could be performed through matrix operations, and while the majority of the source code of the image processing system was created with C++, additional Python mathematics libraries such as NumPy or SciPy could be introduced for their rapid deployment capabilities on top of the existing software architecture. Navigability reporting itself could be further enhanced to provide directional data, where the image processing system's `segnet` ROS output provides additional navigation suggestions based on matrix operation results such as "turn left," "turn right," "move ahead," or "stop," depending on the optimal route computed by the system.

Additional camera technologies could be implemented for the hardware design of the image processing system. A major limitation was observed during image capture from the image resolution and frame rate of Logitech C930e USB cameras, especially when multiple cameras were simultaneously active. This limitation could be overcome by using MIPI CSI-2 cameras (i.e., Bayer sensors) connected to the Jetson TX2, which would greatly improve overall system performance by saving CPU overhead [55]. The Logitech USB cameras used in this project worked during initial experiments with image capture and processing, but

as the system became more sophisticated, the need for computational cost savings became quickly recognized. While the financial cost of the USB cameras was much less prohibitive than that of a system consisting of multiple MIPI CSI-2 cameras, the USB cameras are much less suitable for the field deployment, where the SAR-UGV needs to perform computations and make autonomous decisions in real time, those of which cannot be easily controlled when outside of the laboratory environment.

To further improve the capabilities of the image processing system, camera technologies beyond those that operate in daylight alone, and those that capture images in only two-dimensions, could be explored. Camera sensors designed for low-light conditions would be an improvement, allowing SAR-UGV mission operations to extend from early in the morning or later into the evening when less sunlight is available. Thermal imaging cameras could take this one step further, in which no light in the visible spectrum would be required, thus enabling the possibility for the SAR-UGV to operate during night-time search and rescue missions. Another beneficial feature of thermal imaging cameras is that they are capable of seeing through fog and sun glare, making them less susceptible to mission disruption in such environments these phenomena occur, and which would cause a system using only traditional digital camera sensors to fail. Stereo cameras (i.e., 3D cameras) could also be implemented to provide spatial data, making the SAR-UGV autonomous control system less reliant on LIDAR and RADAR subsystems. For a best-in-class solution, the SAR-UGV could be equipped with thermal stereo vision cameras, such as those manufactured by FLIR Systems [56]. Such a solution combines the features of the latter two previously mentioned camera technologies for an increased level of image data acquisition unmatched by any other single camera hardware system alone. Hence, utilization of a thermal stereo camera system would allow the SAR-UGV to operate any time of day or night in virtually any environment reasonable for mission operations.

The possibilities are endless for the design and implementation of a better image processing system applied to search and rescue missions. Barring the insurmountable engineering burdens of finances and time, and given the proper specifications of mission critical and supplemental tasks, the features and performance of such an embedded system can be limited only by one's imagination and creativity. Hence, this thesis serves as substance that such an image processing system implementing image classification, object detection, and image segmentation–all of which produce data communicated over a shared network of subsystems–can be integrated into a an unmanned ground vehicle for search and rescue applications in forested areas, and save more lives.

APPENDIX A Source Code

A.1 cameratest.cpp

Description of the cameratest application appears in Section 4.1.2.

```
/*------------------------------------------------
    File:       cameratest.cpp
    Author:     Ian Draney
    Purpose:    Captures live video streams from four
                connected video sources simultaneously.
                OpenCV library tiles captured frames
                from each camera into a single frame.
                Boos library overlays a timestamp
                from system time on each frame.
------------------------------------------------*/

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
#include <string>
#include <sstream>
#include <stdio.h>
#include <vector>
#include <math.h>
#include <opencv2/cudacodec.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/gregorian/gregorian.hpp>

using namespace std;
using namespace cv;

////////////////////////////////////////////////////////////////////////
// String array of days of the week
const static std::string days_of_the_week_name[] = {
    "Sunday",
```

```
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
};



////////////////////////////////////////////////////////////////////////
// Function to get day of the week index from Boost Gregorian Library
int get_day_index(int day, int month, int year) {
    boost::gregorian::date d(year, month, day);
    return d.day_of_week();
}



////////////////////////////////////////////////////////////////////////
// Function to print day, date & time to milliseconds
std::string datetime_str()
{
    // Get current time from the clock (to microseconds if necessary)
    const boost::posix_time::ptime now =
        boost::posix_time::microsec_clock::local_time();

    // Get the time offset in current day
    const boost::posix_time::time_duration td = now.time_of_day();

    const long years        = now.date().year();
    const long months       = now.date().month();
    const long days         = now.date().day();
    const long hours        = td.hours();
    const long minutes      = td.minutes();
    const long seconds      = td.seconds();
    const long milliseconds = td.total_milliseconds() -
                              ((hours * 3600 + minutes * 60 + seconds)
   * 1000);

    // Get the day of the week by name name
    const int index = get_day_index(days, months, years);
    std::string dayname = days_of_the_week_name[index];

    char datetime_buffer[32];
    sprintf(datetime_buffer, "%02ld/%02ld/%04ld %02ld:%02ld:%02ld.%03ld
   ",
        months, days, years, hours, minutes, seconds, milliseconds);

    char daydatetime_buffer[64];
```

```
80      memset(daydatetime_buffer, 0, sizeof(daydatetime_buffer)); // Clear
        buffer
81      strcat(daydatetime_buffer, dayname.c_str());
82      strcat(daydatetime_buffer, ", ");
83      strcat(daydatetime_buffer, datetime_buffer);
84
85      return daydatetime_buffer;
86  }
87
88
89
90  ////////////////////////////////////////////////////////////////////////
91  // Main Function
92  ////////////////////////////////////////////////////////////////////////
93  int main() {
94
95      // OpenCV frame matrices
96      cv::Mat camera01, camera02, camera03, camera04;
97
98      // Logitech C930e Pipelines. 640x480 resolution.
99      string pipeline01 = "v4l2src device=/dev/video1 ! video/x-raw,
        width=(int)640, height=(int)480, format=(string)YUY2, framerate=(
        fraction)30/1 ! videoconvert ! appsink";
100     string pipeline02 = "v4l2src device=/dev/video2 ! video/x-raw,
        width=(int)640, height=(int)480, format=(string)YUY2, framerate=(
        fraction)30/1 ! videoconvert ! appsink";
101     string pipeline03 = "v4l2src device=/dev/video3 ! video/x-raw,
        width=(int)640, height=(int)480, format=(string)YUY2, framerate=(
        fraction)30/1 ! videoconvert ! appsink";
102     string pipeline04 = "v4l2src device=/dev/video4 ! video/x-raw,
        width=(int)640, height=(int)480, format=(string)YUY2, framerate=(
        fraction)30/1 ! videoconvert ! appsink";
103
104     std::cout << "Camera 1: Using pipeline: " << pipeline01 << std::
        endl;
105     std::cout << "Camera 2: Using pipeline: " << pipeline02 << std::
        endl;
106     std::cout << "Camera 3: Using pipeline: " << pipeline03 << std::
        endl;
107     std::cout << "Camera 4: Using pipeline: " << pipeline04 << std::
        endl;
108
109     // Create OpenCV capture objects.
110     cv::VideoCapture capture01(pipeline01, cv::CAP_GSTREAMER);
111     cv::VideoCapture capture02(pipeline02, cv::CAP_GSTREAMER);
112     cv::VideoCapture capture03(pipeline03, cv::CAP_GSTREAMER);
113     cv::VideoCapture capture04(pipeline04, cv::CAP_GSTREAMER);
114
115     if (!capture01.isOpened())
116     {
```

```
117         std::cout << "Connection to Camera 1 failed. Exiting." << std::
    endl;
118         return -1;
119      }
120      if (!capture02.isOpened())
121      {
122         std::cout << "Connection to Camera 2 failed. Exiting." << std::
    endl;
123         return -1;
124      }
125      if (!capture03.isOpened())
126      {
127         std::cout << "Connection to Camera 3 failed. Exiting." << std::
    endl;
128         return -1;
129      }
130      if (!capture04.isOpened())
131      {
132         std::cout << "Connection to Camera 4 failed. Exiting." << std::
    endl;
133         return -1;
134      }
135
136      // Capture cameras to generate tile size. Dynamic based on pipeline
     input.
137      capture01 >> camera01;
138      capture02 >> camera02;
139      capture03 >> camera03;
140      capture04 >> camera04;
141
142      cv::Mat tile(cv::Size(camera01.cols + camera02.cols, camera01.rows
    + camera03.rows), CV_8UC3); // This will break if camera04 is a
    different size.
143
144      cv::Mat dashboard01 = camera01(cv::Rect(0, 5, 600, 20));
145      cv::Mat dashcolor01(dashboard01.size(), CV_8UC3, cv::Scalar(0, 0,
    0));
146      cv::Mat dashboard02 = camera02(cv::Rect(0, 5, 600, 20));
147      cv::Mat dashcolor02(dashboard02.size(), CV_8UC3, cv::Scalar(0, 0,
    0));
148      cv::Mat dashboard03 = camera03(cv::Rect(0, 5, 600, 20));
149      cv::Mat dashcolor03(dashboard03.size(), CV_8UC3, cv::Scalar(0, 0,
    0));
150      cv::Mat dashboard04 = camera04(cv::Rect(0, 5, 600, 20));
151      cv::Mat dashcolor04(dashboard04.size(), CV_8UC3, cv::Scalar(0, 0,
    0));
152      double alpha = 0.3;
153
154      // Capture video in infinite loop.  ESC to exit.
155      while (1) //(frame_count < 1000)
```

```
156     {
157         // Capture camera images
158         capture01 >> camera01;
159         // imshow("Camera 1", camera01);
160 
161         capture02 >> camera02;
162         // imshow("Camera 2", camera02);
163 
164         capture03 >> camera03;
165         // imshow("Camera 3", camera03);
166 
167         capture04 >> camera04;
168         // imshow("Camera 4", camera04);
169 
170         // Display camera titles & datetimes
171         cv::addWeighted(dashcolor01, alpha, dashboard01, 1.0 - alpha ,
    0.0, dashboard01);
172         cv::putText(camera01, "FRONT", Point(10, 20),
    FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 255, 255), 1.0, LINE_AA);
173         cv::putText(camera01, datetime_str(), Point(80, 19),
    FONT_HERSHEY_COMPLEX, 0.4, Scalar(255, 255, 255), 1.0, LINE_AA);
174 
175         cv::addWeighted(dashcolor02, alpha, dashboard02, 1.0 - alpha ,
    0.0, dashboard02);
176         cv::putText(camera02, "BACK", Point(10, 20),
    FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 255, 255), 1.0, LINE_AA);
177         cv::putText(camera02, datetime_str(), Point(80, 19),
    FONT_HERSHEY_COMPLEX, 0.4, Scalar(255, 255, 255), 1.0, LINE_AA);
178 
179         cv::addWeighted(dashcolor03, alpha, dashboard03, 1.0 - alpha ,
    0.0, dashboard03);
180         cv::putText(camera03, "LEFT", Point(10, 20),
    FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 255, 255), 1.0, LINE_AA);
181         cv::putText(camera03, datetime_str(), Point(80, 19),
    FONT_HERSHEY_COMPLEX, 0.4, Scalar(255, 255, 255), 1.0, LINE_AA);
182 
183         cv::addWeighted(dashcolor04, alpha, dashboard04, 1.0 - alpha ,
    0.0, dashboard04);
184         cv::putText(camera04, "RIGHT", Point(10, 20),
    FONT_HERSHEY_SIMPLEX, 0.5, Scalar(255, 255, 255), 1.0, LINE_AA);
185         cv::putText(camera04, datetime_str(), Point(80, 19),
    FONT_HERSHEY_COMPLEX, 0.4, Scalar(255, 255, 255), 1.0, LINE_AA);
186 
187         // Create tiled view
188         camera01.copyTo(tile(cv::Rect(0, 0, camera01.cols, camera01.
    rows)));
189         camera02.copyTo(tile(cv::Rect(camera02.cols, 0, camera02.cols,
    camera02.rows)));
190         camera03.copyTo(tile(cv::Rect(0, camera03.rows, camera03.cols,
    camera03.rows)));
```

```
191         camera04.copyTo(tile(cv::Rect(camera04.cols, camera04.rows,
    camera04.cols, camera04.rows)));
192         imshow("Tiled Cameras", tile);
193 
194         // Press ESC to exit
195         char c = (char)waitKey(1);
196         if( c == 27 )
197             break;
198 
199     }
200 
201     // Cleanup
202     capture01.release();
203     capture02.release();
204     capture03.release();
205     capture04.release();
206     cv::destroyAllWindows();
207 
208     return 0;
209 }
```

Listing A.1: Source code for `cameratest.cpp`.

A.2 `CMakeLists.txt` (`cameratest`)

Description of the `cameratest` application appears in Section 4.1.2.

```
1  cmake_minimum_required (VERSION 3.5)
2 
3  project(cameratest)
4 
5  find_package(OpenCV REQUIRED)
6  find_package(PkgConfig REQUIRED)
7  find_package(CUDA REQUIRED)
8 
9  pkg_check_modules(GST REQUIRED gstreamer-1.0>=1.4
10                                 gstreamer-sdp-1.0>=1.4
11                                 gstreamer-video-1.0>=1.4
12                                 gstreamer-app-1.0>=1.4)
13 
14 include_directories(${OpenCV_INCLUDE_DIRS}, ${GST_INCLUDE_DIRS}, ${
    CUDA_INCLUDE_DIRS})
15 add_definitions(${GSTREAMER_DEFINITIONS})
16 
17 add_executable(cameratest_app cameratest.cpp)
18 
```

```
MESSAGE( STATUS "GST_INCLUDE_DIRS:    " ${GST_INCLUDE_DIRS} )
MESSAGE( STATUS "GST_LIBRARIES:       " ${GST_LIBRARIES} )

target_link_libraries(cameratest_app ${OpenCV_LIBS})
target_link_libraries(cameratest_app ${GST_LIBRARIES})
target_link_libraries(cameratest_app ${CUDA_LIBRARIES})
```

Listing A.2: Source code for the `CMakeLists.txt` file used to build the `cameratest` application.

A.3 `recordtest.cpp`

Description of the `recordtest` application appears in Section 4.1.2.

```
/*-----------------------------------------------
    File:       recordtest.cpp
    Author:     Ian Draney
    Purpose:    Record video from four web cameras
                simultaneously on the Jetson TX2 platform.
                OpenCV library saves each video stream
                to file on disk.
                Boost library used to incorporate time
                stamp in file name.
-----------------------------------------------*/

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
#include <string>
#include <sstream>
#include <stdio.h>
#include <vector>
#include <math.h>
#include <opencv2/cudacodec.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/gregorian/gregorian.hpp>

using namespace std;
using namespace cv;

#define USE_CAMERA_01   1
#define USE_CAMERA_02   1
#define USE_CAMERA_03   1
#define USE_CAMERA_04   1

```

```
33 ////////////////////////////////////////////////////////////////////////
34 // String array of days of the week
35 const static std::string days_of_the_week_name[] = {
36     "Sunday",
37     "Monday",
38     "Tuesday",
39     "Wednesday",
40     "Thursday",
41     "Friday",
42     "Saturday"
43 };
44
45
46
47 ////////////////////////////////////////////////////////////////////////
48 // Function to get day of the week index from Boost Gregorian Library
49 int get_day_index(int day, int month, int year) {
50     boost::gregorian::date d(year, month, day);
51     return d.day_of_week();
52 }
53
54
55
56 ////////////////////////////////////////////////////////////////////////
57 // Function to print day, date & time to milliseconds
58 std::string datetime_str()
59 {
60     // Get current time from the clock (to microseconds if necessary)
61     const boost::posix_time::ptime now =
62         boost::posix_time::microsec_clock::local_time();
63
64     // Get the time offset in current day
65     const boost::posix_time::time_duration td = now.time_of_day();
66
67     const long years        = now.date().year();
68     const long months       = now.date().month();
69     const long days         = now.date().day();
70     const long hours        = td.hours();
71     const long minutes      = td.minutes();
72     const long seconds      = td.seconds();
73     const long milliseconds = td.total_milliseconds() -
74                               ((hours * 3600 + minutes * 60 + seconds)
    * 1000);
75
76     // Get the day of the week by name name
77     const int index = get_day_index(days, months, years);
78     std::string dayname = days_of_the_week_name[index];
79
80     char datetime_buffer[32];
81     sprintf(datetime_buffer, "%02ld/%02ld/%04ld %02ld:%02ld:%02ld.%03ld
```

```
    ",
82          months, days, years, hours, minutes, seconds, milliseconds);
83
84      char daydatetime_buffer[64];
85      memset(daydatetime_buffer, 0, sizeof(daydatetime_buffer)); // Clear
        buffer
86      strcat(daydatetime_buffer, dayname.c_str());
87      strcat(daydatetime_buffer, ", ");
88      strcat(daydatetime_buffer, datetime_buffer);
89
90      return daydatetime_buffer;
91  }
92
93
94
95  ////////////////////////////////////////////////////////////////////////
96  // Main Function
97  ////////////////////////////////////////////////////////////////////////
98  int main() {
99
100     // OpenCV frame matrices
101     cv::Mat camera01, camera02, camera03, camera04;
102
103     // Camera width & height
104     // Currently 360p
105     int width =  640;
106     int height = 360;
107
108     // Pipeline and file path constructors
109     ostringstream ss01, ss02, ss03, ss04;
110     ostringstream vv01, vv02, vv03, vv04;
111
112     // Strings for pipelines and file paths
113     string pipeline01, pipeline02, pipeline03, pipeline04;
114     string path01, path02, path03, path04;
115
116     // Create VideoWriter and codec
117     VideoWriter video01, video02, video03, video04;
118
119     //int codec = VideoWriter::fourcc('M','P','4','V');
120     int codec = VideoWriter::fourcc('M','J','P','G');
121     int fps = 30;
122
123     time_t t = time(0);   // get time now
124     struct tm * now = localtime( & t );
125
126     char timestamp_buffer [80];
127     strftime(timestamp_buffer,80,"%Y%m%d%H%M%S",now);
128
129     std::cout << "RECORDING!" << std::endl;
```

```
130     std::cout << "Setting up cameras . . . " << std::endl;
131 
132 #ifdef USE_CAMERA_01
133     ss01 << "v4l2src device=/dev/video1 io-mode=2 ! image/jpeg, width=(
    int)" << width << ", height=(int)" << height << " ! jpegdec ! video/
    x-raw ! videoconvert ! video/x-raw, format=BGR ! appsink";
134 
135     pipeline01 = ss01.str();
136 
137     std::cout << "Camera 1: Using pipeline: " << pipeline01 << std::
    endl;
138 
139     cv::VideoCapture capture01(pipeline01, cv::CAP_GSTREAMER);
140 
141     if (!capture01.isOpened())
142     {
143         std::cout << "Connection to Camera 1 failed. Exiting." << std::
    endl;
144         return -1;
145     }
146 #else
147     std::cout << "Camera 1 not defined" << std::endl;
148 #endif
149 
150 #ifdef USE_CAMERA_02
151     ss02 << "v4l2src device=/dev/video2 io-mode=2 ! image/jpeg, width=(
    int)" << width << ", height=(int)" << height << " ! jpegdec ! video/
    x-raw ! videoconvert ! video/x-raw, format=BGR ! appsink";
152 
153     pipeline02 = ss02.str();
154 
155     std::cout << "Camera 2: Using pipeline: " << pipeline02 << std::
    endl;
156 
157     cv::VideoCapture capture02(pipeline02, cv::CAP_GSTREAMER);
158 
159     if (!capture02.isOpened())
160     {
161         std::cout << "Connection to Camera 2 failed. Exiting." << std::
    endl;
162         return -1;
163     }
164 #else
165     std::cout << "Camera 2 not defined" << std::endl;
166 #endif
167 
168 #ifdef USE_CAMERA_03
169     ss03 << "v4l2src device=/dev/video3 io-mode=2 ! image/jpeg, width=(
    int)" << width << ", height=(int)" << height << " ! jpegdec ! video/
    x-raw ! videoconvert ! video/x-raw, format=BGR ! appsink";
```

```
170
171     pipeline03 = ss03.str();
172
173     std::cout << "Camera 3: Using pipeline: " << pipeline03 << std::
        endl;
174
175     cv::VideoCapture capture03(pipeline03, cv::CAP_GSTREAMER);
176
177     if (!capture03.isOpened())
178     {
179         std::cout << "Connection to Camera 3 failed. Exiting." << std::
        endl;
180         return -1;
181     }
182 #else
183     std::cout << "Camera 3 not defined" << std::endl;
184 #endif
185
186 #ifdef USE_CAMERA_04
187     ss04 << "v4l2src device=/dev/video4 io-mode=2 ! image/jpeg, width=(
        int)" << width << ", height=(int)" << height << " ! jpegdec ! video/
        x-raw ! videoconvert ! video/x-raw, format=BGR ! appsink";
188
189     pipeline04 = ss04.str();
190
191     std::cout << "Camera 4: Using pipeline: " << pipeline04 << std::
        endl;
192
193     cv::VideoCapture capture04(pipeline04, cv::CAP_GSTREAMER);
194
195     if (!capture04.isOpened())
196     {
197         std::cout << "Connection to Camera 4 failed. Exiting." << std::
        endl;
198         return -1;
199     }
200
201     capture04 >> camera04;
202
203 #else
204     std::cout << "Camera 4 not defined" << std::endl;
205 #endif
206
207 ////////////////////////////////////////////////////
208 ///// END PIPELINE STRINGS
209 ////////////////////////////////////////////////////
210
211     std::cout << "Capturing Video" << std::endl;
212
213 #ifdef USE_CAMERA_01
```

```
214     vv01 << "camera01_" << timestamp_buffer << ".avi";
215     path01 = vv01.str();
216     video01.open(path01, codec, fps, Size(width, height), 1);
217 #endif
218 #ifdef USE_CAMERA_02
219     vv02 << "camera02_" << timestamp_buffer << ".avi";
220     path02 = vv02.str();
221     video02.open(path02, codec, fps, Size(width, height), 1);
222 #endif
223 #ifdef USE_CAMERA_03
224     vv03 << "camera03_" << timestamp_buffer << ".avi";
225     path03 = vv03.str();
226     video03.open(path03, codec, fps, Size(width, height), 1);
227 #endif
228 #ifdef USE_CAMERA_04
229     vv04 << "camera04_" << timestamp_buffer << ".avi";
230     path04 = vv04.str();
231     video04.open(path04, codec, fps, Size(width, height), 1);
232 #endif
233 
234     // Capture video in infinite loop.  ESC to exit.
235     // Optional Frame Count: (e.g., frame_count < 1000)
236     while (1)
237     {
238         // Capture and record
239 #ifdef USE_CAMERA_01
240         capture01 >> camera01;
241         imshow("Camera 1", camera01);
242 
243         // Break on empty frame
244         if (camera01.empty())
245         {
246             std::cerr << "ERROR! blank frame grabbed\n\r";
247             return -1;
248         }
249 
250         // Output to file
251         video01.write(camera01);
252 #endif
253 #ifdef USE_CAMERA_02
254         capture02 >> camera02;
255         imshow("Camera 2", camera02);
256 
257         // Break on empty frame
258         if (camera02.empty())
259         {
260             std::cerr << "ERROR! blank frame grabbed\n\r";
261             return -1;
262         }
263 
```

```
264         // Output to file
265         video02.write(camera02);
266 #endif
267 #ifdef USE_CAMERA_03
268         capture03 >> camera03;
269         imshow("Camera 3", camera03);
270 
271         // Break on empty frame
272         if (camera03.empty())
273         {
274             std::cerr << "ERROR! blank frame grabbed\n\r";
275             return -1;
276         }
277 
278         // Output to file
279         video03.write(camera03);
280 #endif
281 #ifdef USE_CAMERA_04
282         capture04 >> camera04;
283         imshow("Camera 4", camera04);
284 
285         // Break on empty frame
286         if (camera04.empty())
287         {
288             std::cerr << "ERROR! blank frame grabbed\n\r";
289             return -1;
290         }
291 
292         // Output to file
293         video04.write(camera04);
294 #endif
295 
296         // Press ESC to exit -- if imshow() is in use
297         char c = (char)waitKey(1);
298         if( c == 27 )
299             break;
300 
301     }
302 
303     // Cleanup
304 #ifdef USE_CAMERA_01
305     capture01.release();
306     video01.release();
307 #endif
308 #ifdef USE_CAMERA_02
309     capture02.release();
310     video02.release();
311 #endif
312 #ifdef USE_CAMERA_03
313     capture03.release();
```

```
314     video03.release();
315 #endif
316 #ifdef USE_CAMERA_04
317     capture04.release();
318     video04.release();
319 #endif
320 
321     cv::destroyAllWindows();
322 
323     std::cout << "Finished Recording!" << std::endl;
324 
325     return 0;
326 }
```

Listing A.3: Source code for `recordtest.cpp`.

A.4 `CMakeLists.txt` (`recordtest`)

Description of the `recordtest` application appears in Section 4.1.2.

```
1  cmake_minimum_required (VERSION 3.5)
2  
3  project(recordtest)
4  
5  find_package(OpenCV REQUIRED)
6  find_package(PkgConfig REQUIRED)
7  find_package(CUDA REQUIRED)
8  
9  pkg_check_modules(GST REQUIRED gstreamer-1.0>=1.4
10                                gstreamer-sdp-1.0>=1.4
11                                gstreamer-video-1.0>=1.4
12                                gstreamer-app-1.0>=1.4)
13 
14 include_directories(${OpenCV_INCLUDE_DIRS}, ${GST_INCLUDE_DIRS}, ${
      CUDA_INCLUDE_DIRS})
15 add_definitions(${GSTREAMER_DEFINITIONS})
16 
17 add_executable(recordtest_app recordtest.cpp)
18 
19 MESSAGE( STATUS "GST_INCLUDE_DIRS:    " ${GST_INCLUDE_DIRS} )
20 MESSAGE( STATUS "GST_LIBRARIES:       " ${GST_LIBRARIES} )
21 
```

```
target_link_libraries(recordtest_app ${OpenCV_LIBS})
target_link_libraries(recordtest_app ${GST_LIBRARIES})
target_link_libraries(recordtest_app ${CUDA_LIBRARIES})
```

Listing A.4: Source code for the `CMakeLists.txt` file used to build `recordtest`.

A.5 `video_viewer_camera_1.ros1.launch`

Description of the `video_viewer_camera_1.ros1.launch` file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video1"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="raw"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_1
   .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_1
   .ros1.launch">
    <arg name="topic" value="/video_source_camera_1/raw"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.5: Source code for `video_viewer_camera_1.ros1.launch`.

A.6 video_viewer_camera_2.ros1.launch

Description of the video_viewer_camera_2.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video2"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="raw"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_2
    .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_2
    .ros1.launch">
    <arg name="topic" value="/video_source_camera_2/raw"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.6: Source code for video_viewer_camera_2.ros1.launch.

A.7 video_viewer_camera_3.ros1.launch

Description of the video_viewer_camera_3.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video3"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="raw"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_3
    .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_3
    .ros1.launch">
    <arg name="topic" value="/video_source_camera_3/raw"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.7: Source code for video_viewer_camera_3.ros1.launch.

A.8 video_viewer_camera_4.ros1.launch

Description of the video_viewer_camera_4.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video4"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="raw"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_4
    .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_4
    .ros1.launch">
    <arg name="topic" value="/video_source_camera_4/raw"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.8: Source code for video_viewer_camera_4.ros1.launch.

A.9 video_source_camera_1.ros1.launch

Description of the video_source_camera_1.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <arg name="input" default="v4l2:///dev/video1"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <node pkg="ros_deep_learning" type="video_source" name="
   video_source_camera_1" output="screen">
    <param name="resource" value="$(arg input)"/>
    <param name="width" value="$(arg input_width)"/>
    <param name="height" value="$(arg input_height)"/>
    <param name="loop" value="$(arg input_loop)"/>
  </node>

</launch>
```

Listing A.9: Source code for video_source_camera_1.ros1.launch.

A.10 video_source_camera_2.ros1.launch

Description of the video_source_camera_2.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <arg name="input" default="v4l2:///dev/video2"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <node pkg="ros_deep_learning" type="video_source" name="
   video_source_camera_2" output="screen">
    <param name="resource" value="$(arg input)"/>
```

```
    <param name="width" value="$(arg input_width)"/>
    <param name="height" value="$(arg input_height)"/>
    <param name="loop" value="$(arg input_loop)"/>
  </node>

</launch>
```

Listing A.10: Source code for `video_source_camera_2.ros1.launch`.

A.11 `video_source_camera_3.ros1.launch`

Description of the `video_source_camera_3.ros1.launch` file appears in Section 4.2.2.

```
<launch>

  <arg name="input" default="v4l2:///dev/video3"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <node pkg="ros_deep_learning" type="video_source" name="
   video_source_camera_3" output="screen">
    <param name="resource" value="$(arg input)"/>
    <param name="width" value="$(arg input_width)"/>
    <param name="height" value="$(arg input_height)"/>
    <param name="loop" value="$(arg input_loop)"/>
  </node>

</launch>
```

Listing A.11: Source code for `video_source_camera_3.ros1.launch`.

A.12 video_source_camera_4.ros1.launch

Description of the video_source_camera_4.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <arg name="input" default="v4l2:///dev/video4"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <node pkg="ros_deep_learning" type="video_source" name="
   video_source_camera_4" output="screen">
    <param name="resource" value="$(arg input)"/>
    <param name="width" value="$(arg input_width)"/>
    <param name="height" value="$(arg input_height)"/>
    <param name="loop" value="$(arg input_loop)"/>
  </node>

</launch>
```

Listing A.12: Source code for video_source_camera_4.ros1.launch.

A.13 video_output_camera_1.ros1.launch

Description of the video_output_camera_1.ros1.launch file appears in Section 4.2.2.

```
<launch>
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>
  <arg name="topic"/>

  <node pkg="ros_deep_learning" type="video_output" name="
   video_output_camera_1" output="screen">
    <remap from="/video_output_camera_1/image_in" to="$(arg topic)"/>
    <param name="resource" value="$(arg output)"/>
    <param name="codec" value="$(arg output_codec)"/>
```

```
    <param name="bitrate" value="$(arg output_bitrate)"/>
  </node>

</launch>
```

Listing A.13: Source code for `video_output_camera_1.ros1.launch`.

A.14 `video_output_camera_2.ros1.launch`

Description of the `video_output_camera_2.ros1.launch` file appears in Section 4.2.2.

```
<launch>
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>
  <arg name="topic"/>

  <node pkg="ros_deep_learning" type="video_output" name="
   video_output_camera_2" output="screen">
    <remap from="/video_output_camera_2/image_in" to="$(arg topic)"/>
    <param name="resource" value="$(arg output)"/>
    <param name="codec" value="$(arg output_codec)"/>
    <param name="bitrate" value="$(arg output_bitrate)"/>
  </node>

</launch>
```

Listing A.14: Source code for `video_output_camera_2.ros1.launch`.

A.15 `video_output_camera_3.ros1.launch`

Description of the `video_output_camera_3.ros1.launch` file appears in Section 4.2.2.

```
<launch>
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>
  <arg name="topic"/>
```

```
6
7   <node pkg="ros_deep_learning" type="video_output" name="
     video_output_camera_3" output="screen">
8     <remap from="/video_output_camera_3/image_in" to="$(arg topic)"/>
9     <param name="resource" value="$(arg output)"/>
10    <param name="codec" value="$(arg output_codec)"/>
11    <param name="bitrate" value="$(arg output_bitrate)"/>
12  </node>
13
14 </launch>
```

Listing A.15: Source code for `video_output_camera_3.ros1.launch`.

A.16 `video_output_camera_4.ros1.launch`

Description of the `video_output_camera_4.ros1.launch` file appears in Section 4.2.2.

```
1  <launch>
2    <arg name="output" default="display://0"/>
3    <arg name="output_codec" default="unknown"/>
4    <arg name="output_bitrate" default="0"/>
5    <arg name="topic"/>
6
7    <node pkg="ros_deep_learning" type="video_output" name="
      video_output_camera_4" output="screen">
8      <remap from="/video_output_camera_4/image_in" to="$(arg topic)"/>
9      <param name="resource" value="$(arg output)"/>
10     <param name="codec" value="$(arg output_codec)"/>
11     <param name="bitrate" value="$(arg output_bitrate)"/>
12   </node>
13
14 </launch>
```

Listing A.16: Source code for `video_output_camera_4.ros1.launch`.

A.17 imagenet_camera_1.ros1.launch

Description of the imagenet_camera_1.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video1"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_1
   .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- IMAGENET -->
  <arg name="model_name" default="googlenet"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
  <arg name="class_labels_path" default=""/>
  <arg name="input_blob" default=""/>
  <arg name="output_blob" default=""/>

  <node pkg="ros_deep_learning" type="imagenet" name="imagenet_camera_1
   " output="screen">
    <remap from="/imagenet_camera_1/image_in" to="/
   video_source_camera_1/raw"/>
    <param name="model_name" value="$(arg model_name)"/>
    <param name="model_path" value="$(arg model_path)"/>
    <param name="prototxt_path" value="$(arg prototxt_path)"/>
    <param name="class_labels_path" value="$(arg class_labels_path)"/>
    <param name="input_blob" value="$(arg input_blob)"/>
    <param name="output_blob" value="$(arg output_blob)"/>
  </node>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

```

```
  <include file="$(find ros_deep_learning)/launch/video_output_camera_1
   .ros1.launch">
    <arg name="topic" value="/imagenet_camera_1/overlay"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.17: Source code for `imagenet_camera_1.ros1.launch`.

A.18 `imagenet_camera_2.ros1.launch`

Description of the `imagenet_camera_2.ros1.launch` file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video2"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_2
   .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- IMAGENET -->
  <arg name="model_name" default="googlenet"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
  <arg name="class_labels_path" default=""/>
  <arg name="input_blob" default=""/>
  <arg name="output_blob" default=""/>

  <node pkg="ros_deep_learning" type="imagenet" name="imagenet_camera_2
   " output="screen">
    <remap from="/imagenet_camera_2/image_in" to="/
   video_source_camera_2/raw"/>
```

```
    <param name="model_name" value="$(arg model_name)"/>
    <param name="model_path" value="$(arg model_path)"/>
    <param name="prototxt_path" value="$(arg prototxt_path)"/>
    <param name="class_labels_path" value="$(arg class_labels_path)"/>
    <param name="input_blob" value="$(arg input_blob)"/>
    <param name="output_blob" value="$(arg output_blob)"/>
  </node>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_2
   .ros1.launch">
    <arg name="topic" value="/imagenet_camera_2/overlay"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.18: Source code for `imagenet_camera_2.ros1.launch`.

A.19 `imagenet_camera_3.ros1.launch`

Description of the `imagenet_camera_3.ros1.launch` file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video3"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_3
   .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>
```

```
17
18    <!-- IMAGENET -->
19    <arg name="model_name" default="googlenet"/>
20    <arg name="model_path" default=""/>
21    <arg name="prototxt_path" default=""/>
22    <arg name="class_labels_path" default=""/>
23    <arg name="input_blob" default=""/>
24    <arg name="output_blob" default=""/>
25
26    <node pkg="ros_deep_learning" type="imagenet" name="imagenet_camera_3
       " output="screen">
27      <remap from="/imagenet_camera_3/image_in" to="/
      video_source_camera_3/raw"/>
28      <param name="model_name" value="$(arg model_name)"/>
29      <param name="model_path" value="$(arg model_path)"/>
30      <param name="prototxt_path" value="$(arg prototxt_path)"/>
31      <param name="class_labels_path" value="$(arg class_labels_path)"/>
32      <param name="input_blob" value="$(arg input_blob)"/>
33      <param name="output_blob" value="$(arg output_blob)"/>
34    </node>
35
36    <!-- VIDEO OUTPUT -->
37    <arg name="output" default="display://0"/>
38    <arg name="output_codec" default="unknown"/>
39    <arg name="output_bitrate" default="0"/>
40
41    <include file="$(find ros_deep_learning)/launch/video_output_camera_3
       .ros1.launch">
42      <arg name="topic" value="/imagenet_camera_3/overlay"/>
43      <arg name="output" value="$(arg output)"/>
44      <arg name="output_codec" value="$(arg output_codec)"/>
45      <arg name="output_bitrate" value="$(arg output_bitrate)"/>
46    </include>
47
48  </launch>
```

Listing A.19: Source code for `imagenet_camera_3.ros1.launch`.

A.20 `imagenet_camera_4.ros1.launch`

Description of the `imagenet_camera_4.ros1.launch` file appears in Section 4.2.2.

```
1   <launch>
2
3     <!-- VIDEO SOURCE -->
4     <arg name="input" default="v4l2:///dev/video4"/>
```

```
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_4
   .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- IMAGENET -->
  <arg name="model_name" default="googlenet"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
  <arg name="class_labels_path" default=""/>
  <arg name="input_blob" default=""/>
  <arg name="output_blob" default=""/>

  <node pkg="ros_deep_learning" type="imagenet" name="imagenet_camera_4
   " output="screen">
    <remap from="/imagenet_camera_4/image_in" to="/
   video_source_camera_4/raw"/>
    <param name="model_name" value="$(arg model_name)"/>
    <param name="model_path" value="$(arg model_path)"/>
    <param name="prototxt_path" value="$(arg prototxt_path)"/>
    <param name="class_labels_path" value="$(arg class_labels_path)"/>
    <param name="input_blob" value="$(arg input_blob)"/>
    <param name="output_blob" value="$(arg output_blob)"/>
  </node>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_4
   .ros1.launch">
    <arg name="topic" value="/imagenet_camera_4/overlay"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.20: Source code for `imagenet_camera_4.ros1.launch`.

A.21 detectnet_camera_1.ros1.launch

Description of the detectnet_camera_1.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video1"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_1
   .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- DETECTNET -->
  <arg name="model_name" default="ssd-mobilenet-v2"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
  <arg name="class_labels_path" default=""/>
  <arg name="input_blob" default=""/>
  <arg name="output_cvg" default=""/>
  <arg name="output_bbox" default=""/>
  <arg name="overlay_flags" default="box,labels,conf"/>
  <arg name="mean_pixel_value" default="0.0"/>
  <arg name="threshold" default="0.5"/>

  <node pkg="ros_deep_learning" type="detectnet" name="
   detectnet_camera_1" output="screen">
    <remap from="/detectnet_camera_1/image_in" to="/
   video_source_camera_1/raw"/>
    <param name="model_name" value="$(arg model_name)"/>
    <param name="model_path" value="$(arg model_path)"/>
    <param name="prototxt_path" value="$(arg prototxt_path)"/>
    <param name="class_labels_path" value="$(arg class_labels_path)"/>
    <param name="input_blob" value="$(arg input_blob)"/>
    <param name="output_cvg" value="$(arg output_cvg)"/>
    <param name="output_bbox" value="$(arg output_bbox)"/>
    <param name="overlay_flags" value="$(arg overlay_flags)"/>
    <param name="mean_pixel_value" value="$(arg mean_pixel_value)"/>
```

```
    <param name="threshold" value="$(arg threshold)"/>
  </node>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_1
    .ros1.launch">
    <arg name="topic" value="/detectnet_camera_1/overlay"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.21: Source code for `detectnet_camera_1.ros1.launch`.

A.22 `detectnet_camera_2.ros1.launch`

Description of the `detectnet_camera_2.ros1.launch` file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video2"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_2
    .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- DETECTNET -->
  <arg name="model_name" default="ssd-mobilenet-v2"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
```

```
  <arg name="class_labels_path" default=""/>
  <arg name="input_blob" default=""/>
  <arg name="output_cvg" default=""/>
  <arg name="output_bbox" default=""/>
  <arg name="overlay_flags" default="box,labels,conf"/>
  <arg name="mean_pixel_value" default="0.0"/>
  <arg name="threshold" default="0.5"/>

  <node pkg="ros_deep_learning" type="detectnet" name="
   detectnet_camera_2" output="screen">
    <remap from="/detectnet_camera_2/image_in" to="/
   video_source_camera_2/raw"/>
    <param name="model_name" value="$(arg model_name)"/>
    <param name="model_path" value="$(arg model_path)"/>
    <param name="prototxt_path" value="$(arg prototxt_path)"/>
    <param name="class_labels_path" value="$(arg class_labels_path)"/>
    <param name="input_blob" value="$(arg input_blob)"/>
    <param name="output_cvg" value="$(arg output_cvg)"/>
    <param name="output_bbox" value="$(arg output_bbox)"/>
    <param name="overlay_flags" value="$(arg overlay_flags)"/>
    <param name="mean_pixel_value" value="$(arg mean_pixel_value)"/>
    <param name="threshold" value="$(arg threshold)"/>
  </node>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_2
   .ros1.launch">
    <arg name="topic" value="/detectnet_camera_2/overlay"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.22: Source code for `detectnet_camera_2.ros1.launch`.

A.23 detectnet_camera_3.ros1.launch

Description of the detectnet_camera_3.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video3"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_3
   .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- DETECTNET -->
  <arg name="model_name" default="ssd-mobilenet-v2"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
  <arg name="class_labels_path" default=""/>
  <arg name="input_blob" default=""/>
  <arg name="output_cvg" default=""/>
  <arg name="output_bbox" default=""/>
  <arg name="overlay_flags" default="box,labels,conf"/>
  <arg name="mean_pixel_value" default="0.0"/>
  <arg name="threshold" default="0.5"/>

  <node pkg="ros_deep_learning" type="detectnet" name="
   detectnet_camera_3" output="screen">
    <remap from="/detectnet_camera_3/image_in" to="/
   video_source_camera_3/raw"/>
    <param name="model_name" value="$(arg model_name)"/>
    <param name="model_path" value="$(arg model_path)"/>
    <param name="prototxt_path" value="$(arg prototxt_path)"/>
    <param name="class_labels_path" value="$(arg class_labels_path)"/>
    <param name="input_blob" value="$(arg input_blob)"/>
    <param name="output_cvg" value="$(arg output_cvg)"/>
    <param name="output_bbox" value="$(arg output_bbox)"/>
    <param name="overlay_flags" value="$(arg overlay_flags)"/>
    <param name="mean_pixel_value" value="$(arg mean_pixel_value)"/>
```

```
    <param name="threshold" value="$(arg threshold)"/>
  </node>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_3
    .ros1.launch">
    <arg name="topic" value="/detectnet_camera_3/overlay"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.23: Source code for `detectnet_camera_3.ros1.launch`.

A.24 detectnet_camera_4.ros1.launch

Description of the `detectnet_camera_4.ros1.launch` file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video4"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_4
    .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- DETECTNET -->
  <arg name="model_name" default="ssd-mobilenet-v2"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
```

```
22    <arg name="class_labels_path" default=""/>
23    <arg name="input_blob" default=""/>
24    <arg name="output_cvg" default=""/>
25    <arg name="output_bbox" default=""/>
26    <arg name="overlay_flags" default="box,labels,conf"/>
27    <arg name="mean_pixel_value" default="0.0"/>
28    <arg name="threshold" default="0.5"/>
29
30    <node pkg="ros_deep_learning" type="detectnet" name="
       detectnet_camera_4" output="screen">
31      <remap from="/detectnet_camera_4/image_in" to="/
       video_source_camera_4/raw"/>
32      <param name="model_name" value="$(arg model_name)"/>
33      <param name="model_path" value="$(arg model_path)"/>
34      <param name="prototxt_path" value="$(arg prototxt_path)"/>
35      <param name="class_labels_path" value="$(arg class_labels_path)"/>
36      <param name="input_blob" value="$(arg input_blob)"/>
37      <param name="output_cvg" value="$(arg output_cvg)"/>
38      <param name="output_bbox" value="$(arg output_bbox)"/>
39      <param name="overlay_flags" value="$(arg overlay_flags)"/>
40      <param name="mean_pixel_value" value="$(arg mean_pixel_value)"/>
41      <param name="threshold" value="$(arg threshold)"/>
42    </node>
43
44    <!-- VIDEO OUTPUT -->
45    <arg name="output" default="display://0"/>
46    <arg name="output_codec" default="unknown"/>
47    <arg name="output_bitrate" default="0"/>
48
49    <include file="$(find ros_deep_learning)/launch/video_output_camera_4
       .ros1.launch">
50      <arg name="topic" value="/detectnet_camera_4/overlay"/>
51      <arg name="output" value="$(arg output)"/>
52      <arg name="output_codec" value="$(arg output_codec)"/>
53      <arg name="output_bitrate" value="$(arg output_bitrate)"/>
54    </include>
55
56  </launch>
```

Listing A.24: Source code for `detectnet_camera_4.ros1.launch`.

A.25 segnet_camera_1.ros1.launch

Description of the segnet_camera_1.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video1"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_1.ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- SEGNET -->
  <arg name="model_name" default="fcn-resnet18-mhp-512x320"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
  <arg name="class_labels_path" default=""/>
  <arg name="class_colors_path" default=""/>
  <arg name="input_blob" default=""/>
  <arg name="output_blob" default=""/>
  <arg name="mask_filter" default="linear"/>
  <arg name="overlay_filter" default="linear"/>
  <arg name="overlay_alpha" default="180.0"/>

  <node pkg="ros_deep_learning" type="segnet" name="segnet_camera_1" output="screen">
    <remap from="/segnet_camera_1/image_in" to="/video_source_camera_1/raw"/>
    <param name="model_name" value="$(arg model_name)"/>
    <param name="model_path" value="$(arg model_path)"/>
    <param name="prototxt_path" value="$(arg prototxt_path)"/>
    <param name="class_labels_path" value="$(arg class_labels_path)"/>
    <param name="class_colors_path" value="$(arg class_colors_path)"/>
    <param name="input_blob" value="$(arg input_blob)"/>
    <param name="output_blob" value="$(arg output_blob)"/>
    <param name="mask_filter" value="$(arg mask_filter)"/>
    <param name="overlay_filter" value="$(arg overlay_filter)"/>
```

```
    <param name="overlay_alpha" value="$(arg overlay_alpha)"/>
  </node>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_1.ros1.launch">
    <arg name="topic" value="/segnet_camera_1/overlay"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.25: Source code for `segnet_camera_1.ros1.launch`.

A.26 `segnet_camera_2.ros1.launch`

Description of the `segnet_camera_2.ros1.launch` file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video2"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_2.ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- SEGNET -->
  <arg name="model_name" default="fcn-resnet18-mhp-512x320"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
```

```
  <arg name="class_labels_path" default=""/>
  <arg name="class_colors_path" default=""/>
  <arg name="input_blob" default=""/>
  <arg name="output_blob" default=""/>
  <arg name="mask_filter" default="linear"/>
  <arg name="overlay_filter" default="linear"/>
  <arg name="overlay_alpha" default="180.0"/>

  <node pkg="ros_deep_learning" type="segnet" name="segnet_camera_2" output="screen">
    <remap from="/segnet_camera_2/image_in" to="/video_source_camera_2/raw"/>
    <param name="model_name" value="$(arg model_name)"/>
    <param name="model_path" value="$(arg model_path)"/>
    <param name="prototxt_path" value="$(arg prototxt_path)"/>
    <param name="class_labels_path" value="$(arg class_labels_path)"/>
    <param name="class_colors_path" value="$(arg class_colors_path)"/>
    <param name="input_blob" value="$(arg input_blob)"/>
    <param name="output_blob" value="$(arg output_blob)"/>
    <param name="mask_filter" value="$(arg mask_filter)"/>
    <param name="overlay_filter" value="$(arg overlay_filter)"/>
    <param name="overlay_alpha" value="$(arg overlay_alpha)"/>
  </node>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_2.ros1.launch">
    <arg name="topic" value="/segnet_camera_2/overlay"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.26: Source code for `segnet_camera_2.ros1.launch`.

A.27 segnet_camera_3.ros1.launch

Description of the segnet_camera_3.ros1.launch file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video3"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_3.ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- SEGNET -->
  <arg name="model_name" default="fcn-resnet18-mhp-512x320"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
  <arg name="class_labels_path" default=""/>
  <arg name="class_colors_path" default=""/>
  <arg name="input_blob" default=""/>
  <arg name="output_blob" default=""/>
  <arg name="mask_filter" default="linear"/>
  <arg name="overlay_filter" default="linear"/>
  <arg name="overlay_alpha" default="180.0"/>

  <node pkg="ros_deep_learning" type="segnet" name="segnet_camera_3" output="screen">
    <remap from="/segnet_camera_3/image_in" to="/video_source_camera_3/raw"/>
    <param name="model_name" value="$(arg model_name)"/>
    <param name="model_path" value="$(arg model_path)"/>
    <param name="prototxt_path" value="$(arg prototxt_path)"/>
    <param name="class_labels_path" value="$(arg class_labels_path)"/>
    <param name="class_colors_path" value="$(arg class_colors_path)"/>
    <param name="input_blob" value="$(arg input_blob)"/>
    <param name="output_blob" value="$(arg output_blob)"/>
    <param name="mask_filter" value="$(arg mask_filter)"/>
    <param name="overlay_filter" value="$(arg overlay_filter)"/>
```

```
    <param name="overlay_alpha" value="$(arg overlay_alpha)"/>
  </node>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_3
    .ros1.launch">
    <arg name="topic" value="/segnet_camera_3/overlay"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.27: Source code for `segnet_camera_3.ros1.launch`.

A.28 `segnet_camera_4.ros1.launch`

Description of the `segnet_camera_4.ros1.launch` file appears in Section 4.2.2.

```
<launch>

  <!-- VIDEO SOURCE -->
  <arg name="input" default="v4l2:///dev/video4"/>
  <arg name="input_width" default="0"/>
  <arg name="input_height" default="0"/>
  <arg name="input_codec" default="unknown"/>
  <arg name="input_loop" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_source_camera_4
    .ros1.launch">
    <arg name="input" value="$(arg input)"/>
    <arg name="input_width" value="$(arg input_width)"/>
    <arg name="input_height" value="$(arg input_height)"/>
    <arg name="input_codec" value="$(arg input_codec)"/>
    <arg name="input_loop" value="$(arg input_loop)"/>
  </include>

  <!-- SEGNET -->
  <arg name="model_name" default="fcn-resnet18-mhp-512x320"/>
  <arg name="model_path" default=""/>
  <arg name="prototxt_path" default=""/>
```

```
  <arg name="class_labels_path" default=""/>
  <arg name="class_colors_path" default=""/>
  <arg name="input_blob" default=""/>
  <arg name="output_blob" default=""/>
  <arg name="mask_filter" default="linear"/>
  <arg name="overlay_filter" default="linear"/>
  <arg name="overlay_alpha" default="180.0"/>

  <node pkg="ros_deep_learning" type="segnet" name="segnet_camera_4" output="screen">
    <remap from="/segnet_camera_4/image_in" to="/video_source_camera_4/raw"/>
    <param name="model_name" value="$(arg model_name)"/>
    <param name="model_path" value="$(arg model_path)"/>
    <param name="prototxt_path" value="$(arg prototxt_path)"/>
    <param name="class_labels_path" value="$(arg class_labels_path)"/>
    <param name="class_colors_path" value="$(arg class_colors_path)"/>
    <param name="input_blob" value="$(arg input_blob)"/>
    <param name="output_blob" value="$(arg output_blob)"/>
    <param name="mask_filter" value="$(arg mask_filter)"/>
    <param name="overlay_filter" value="$(arg overlay_filter)"/>
    <param name="overlay_alpha" value="$(arg overlay_alpha)"/>
  </node>

  <!-- VIDEO OUTPUT -->
  <arg name="output" default="display://0"/>
  <arg name="output_codec" default="unknown"/>
  <arg name="output_bitrate" default="0"/>

  <include file="$(find ros_deep_learning)/launch/video_output_camera_4.ros1.launch">
    <arg name="topic" value="/segnet_camera_4/overlay"/>
    <arg name="output" value="$(arg output)"/>
    <arg name="output_codec" value="$(arg output_codec)"/>
    <arg name="output_bitrate" value="$(arg output_bitrate)"/>
  </include>

</launch>
```

Listing A.28: Source code for `segnet_camera_4.ros1.launch`.

APPENDIX B Image Segmentation Color Legends

Each image segmentation model used when executing the `segnet` program had its own particular RGB color scheme for the color mask image that was created after pixels had been classified. This color mask was overlaid upon the original image by default. Implementation of image segmentation using `segnet` is first introduced in Section 4.2.2. Image segmentation simulations are discussed in Section 5.1.3. Experimental results when the `segnet` program was executed from the SAR-UGV using the image segmentation models listed in this appendix are contained in Section 5.2.3.

B.1 FCN-ResNet18-Cityscapes-512x256

Table B.1: FCN-RESNET18-CITYSCAPES-512X256 MODEL CLASSES

Class ID	Class Name	R	G	B	Color
0	void	0	0	0	
1	ego_vehicle	128	64	128	
2	ground	81	0	81	
3	road	128	64	128	
4	sidewalk	150	75	200	
5	building	70	70	70	
6	wall	102	102	156	
7	fence	190	153	153	
8	pole	153	153	153	
9	traffic_light	250	170	30	
10	traffic_sign	250	170	30	
11	vegetation	107	142	35	
12	terrain	152	251	152	
13	sky	70	130	180	
14	person	220	20	60	
15	car	0	0	142	
16	truck	0	0	70	
17	bus	0	60	100	
18	train	0	80	100	
19	motorcycle	0	0	230	
20	bicycle	119	11	32	

B.2 FCN-ResNet18-Cityscapes-1024x512

Table B.2: FCN-RESNET18-CITYSCAPES-1024X512 MODEL CLASSES

Class ID	Class Name	R	G	B	Color
0	void	0	0	0	
1	ego_vehicle	128	64	128	
2	ground	81	0	81	
3	road	128	64	128	
4	sidewalk	150	75	200	
5	building	70	70	70	
6	wall	102	102	156	
7	fence	190	153	153	
8	pole	153	153	153	
9	traffic_light	250	170	30	
10	traffic_sign	250	170	30	
11	vegetation	107	142	35	
12	terrain	152	251	152	
13	sky	70	130	180	
14	person	220	20	60	
15	car	0	0	142	
16	truck	0	0	70	
17	bus	0	60	100	
18	train	0	80	100	
19	motorcycle	0	0	230	
20	bicycle	119	11	32	

B.3 FCN-ResNet18-Cityscapes-2048x1024

Table B.3: FCN-RESNET18-CITYSCAPES-2048X1024 MODEL CLASSES

Class ID	Class Name	R	G	B	Color
0	void	0	0	0	
1	ego_vehicle	128	64	128	
2	ground	81	0	81	
3	road	128	64	128	
4	sidewalk	150	75	200	
5	building	70	70	70	
6	wall	102	102	156	
7	fence	190	153	153	
8	pole	153	153	153	
9	traffic_light	250	170	30	
10	traffic_sign	250	170	30	
11	vegetation	107	142	35	
12	terrain	152	251	152	
13	sky	70	130	180	
14	person	220	20	60	
15	car	0	0	142	
16	truck	0	0	70	
17	bus	0	60	100	
18	train	0	80	100	
19	motorcycle	0	0	230	
20	bicycle	119	11	32	

B.4 FCN-ResNet18-DeepScene-576x320

Table B.4: FCN-ResNet18-DeepScene-576x320 Model Classes

Class ID	Class Name	R	G	B	Color
0	trail	200	155	75	
1	grass	85	210	100	
2	vegetation	15	100	20	
3	obstacle	255	185	0	
4	sky	0	120	255	

B.5 FCN-ResNet18-DeepScene-864x480

Table B.5: FCN-ResNet18-DeepScene-864x480 Model Classes

Class ID	Class Name	R	G	B	Color
0	trail	200	155	75	
1	grass	85	210	100	
2	vegetation	15	100	20	
3	obstacle	255	185	0	
4	sky	0	120	255	

B.6 FCN-ResNet18-MHP-512x320

Table B.6: FCN-RESNET18-MHP-512X320 MODEL CLASSES

Class ID	Class Name	R	G	B	Color
0	background	0	0	0	
1	hat/helmet/hair	139	69	19	
2	face	222	184	135	
3	hair	210	105	30	
4	arm	255	255	0	
5	hand	255	165	0	
6	shirt	0	255	0	
7	jacket/coat	60	179	113	
8	dress/robe	107	142	35	
9	bikini/bra	255	0	0	
10	torso_skin	245	222	179	
11	pants	0	0	255	
12	shorts	0	255	255	
13	socks/stockings	238	130	238	
14	shoe/boot	128	0	128	
15	leg	255	0	0	
16	foot	255	0	255	
17	backpack/purse/bag	128	128	128	
18	sunglasses/eyewear	192	192	192	
19	other_accessory	128	128	128	
20	other_item	128	128	128	

B.7 FCN-ResNet18-MHP-640x360

Table B.7: FCN-RESNET18-MHP-640X360 MODEL CLASSES

Class ID	Class Name	R	G	B	Color
0	background	0	0	0	
1	hat/helmet/hair	139	69	19	
2	face	222	184	135	
3	hair	210	105	30	
4	arm	255	255	0	
5	hand	255	165	0	
6	shirt	0	255	0	
7	jacket/coat	60	179	113	
8	dress/robe	107	142	35	
9	bikini/bra	255	0	0	
10	torso_skin	245	222	179	
11	pants	0	0	255	
12	shorts	0	255	255	
13	socks/stockings	238	130	238	
14	shoe/boot	128	0	128	
15	leg	255	0	0	
16	foot	255	0	255	
17	backpack/purse/bag	128	128	128	
18	sunglasses/eyewear	192	192	192	
19	other_accessory	128	128	128	
20	other_item	128	128	128	

B.8 FCN-ResNet18-VOC-320x320

Table B.8: FCN-RESNET18-VOC-320X320 MODEL CLASSES

Class ID	Class Name	R	G	B	Color
0	background	0	0	0	
1	aeroplane	255	0	0	
2	bicycle	0	255	0	
3	bird	0	255	120	
4	boat	0	0	255	
5	bottle	255	0	255	
6	bus	70	70	70	
7	car	102	102	156	
8	cat	190	153	153	
9	chair	180	165	180	
10	cow	150	100	100	
11	diningtable	153	153	153	
12	dog	250	170	30	
13	horse	220	220	0	
14	motorbike	107	142	35	
15	person	192	128	128	
16	pottedplant	70	130	180	
17	sheep	220	20	60	
18	sofa	0	0	142	
19	train	0	0	70	
20	tvmonitor	119	11	32	

B.9 FCN-ResNet18-VOC-512x320

Table B.9: FCN-RESNET18-VOC-512X320 MODEL CLASSES

Class ID	Class Name	R	G	B	Color
0	background	0	0	0	
1	aeroplane	255	0	0	
2	bicycle	0	255	0	
3	bird	0	255	120	
4	boat	0	0	255	
5	bottle	255	0	255	
6	bus	70	70	70	
7	car	102	102	156	
8	cat	190	153	153	
9	chair	180	165	180	
10	cow	150	100	100	
11	diningtable	153	153	153	
12	dog	250	170	30	
13	horse	220	220	0	
14	motorbike	107	142	35	
15	person	192	128	128	
16	pottedplant	70	130	180	
17	sheep	220	20	60	
18	sofa	0	0	142	
19	train	0	0	70	
20	tvmonitor	119	11	32	

B.10 FCN-ResNet18-SUN-512x400

Table B.10: FCN-RESNET18-SUN-512X400 MODEL CLASSES

Class ID	Class Name	R	G	B	Color
0	other	0	0	0	
1	wall	128	0	0	
2	floor	0	128	0	
3	cabinet/shelves/bookshelf/dresser	128	128	0	
4	bed/pillow	0	0	128	
5	chair	128	0	128	
6	sofa	0	128	128	
7	table	128	128	128	
8	door	64	0	0	
9	window	192	0	0	
10	picture/tv/mirror	192	128	0	
11	blinds/curtain	192	0	128	
12	clothes	128	64	128	
13	ceiling	0	192	128	
14	books	128	192	128	
15	fridge	64	64	0	
16	person	192	192	128	
17	toilet	128	0	64	
18	sink	0	128	64	
19	lamp	128	128	64	
20	bathtub	0	0	192	

B.11 FCN-ResNet18-SUN-640x512

Table B.11: FCN-RESNET18-SUN-640X512 MODEL CLASSES

Class ID	Class Name	R	G	B	Color
0	other	0	0	0	
1	wall	128	0	0	
2	floor	0	128	0	
3	cabinet/shelves/bookshelf/dresser	128	128	0	
4	bed/pillow	0	0	128	
5	chair	128	0	128	
6	sofa	0	128	128	
7	table	128	128	128	
8	door	64	0	0	
9	window	192	0	0	
10	picture/tv/mirror	192	128	0	
11	blinds/curtain	192	0	128	
12	clothes	128	64	128	
13	ceiling	0	192	128	
14	books	128	192	128	
15	fridge	64	64	0	
16	person	192	192	128	
17	toilet	128	0	64	
18	sink	0	128	64	
19	lamp	128	128	64	
20	bathtub	0	0	192	

APPENDIX C Segmented Color Mask Images

The image segmentation models used when executing the `segnet` program each produced their own unique color mask output. In Chapter 4 and Chapter 5, image segmentation output images are displayed with this color mask overlaid on the original image as an alpha channel at a preset opacity as the `segnet` program's default output setting. Images containing the default overlay appear in Section 5.1.3 and Section 5.2.3. Images containing only the `segnet` color mask output appear in the pages to follow within this appendix. The model used for each image is indicated by its file name. A color legend of each image segmentation model is contained in Appendix B.

C.1 freiburg-forest-sim-01-fcn-resnet18-cityscapes-512x256-color-mask.jpg

Figure C.1: Color mask results from the `FCN-ResNet18-Cityscapes-512x256` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.2 freiburg-forest-sim-01-fcn-resnet18-cityscapes-1024x512-color-mask.jpg

Figure C.2: Color mask results from the `FCN-ResNet18-Cityscapes-1024x512` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.3 freiburg-forest-sim-01-fcn-resnet18-cityscapes-2048x1024-color-mask.jpg

Figure C.3: Color mask results from the `FCN-ResNet18-Cityscapes-2048x1024` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.4 freiburg-forest-sim-01-fcn-resnet18-deepscene-576x320-color-mask.jpg

Figure C.4: Color mask results from the `FCN-ResNet18-DeepScene-576x320` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.5 freiburg-forest-sim-01-fcn-resnet18-deepscene-864x480-color-mask.jpg

Figure C.5: Color mask results from the `FCN-ResNet18-DeepScene-864x480` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.6 freiburg-forest-sim-01-fcn-resnet18-mhp-512x320-color-mask.jpg

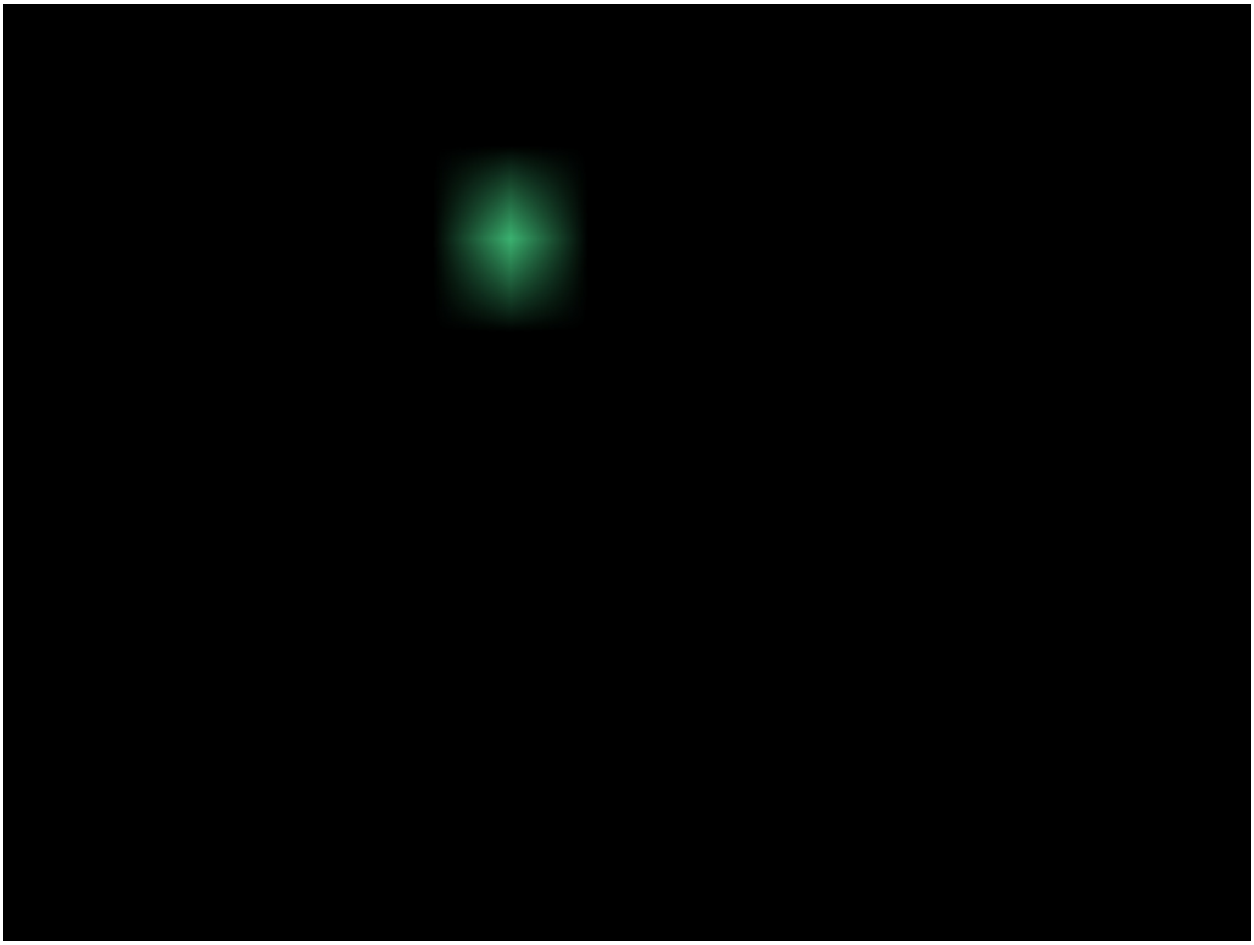


Figure C.6: Color mask results from the `FCN-ResNet18-MHP-512x320` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.7 freiburg-forest-sim-01-fcn-resnet18-mhp-640x360-color-mask.jpg

Figure C.7: Color mask results from the `FCN-ResNet18-MHP-640x360` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.8 freiburg-forest-sim-01-fcn-resnet18-voc-320x320-color-mask.jpg

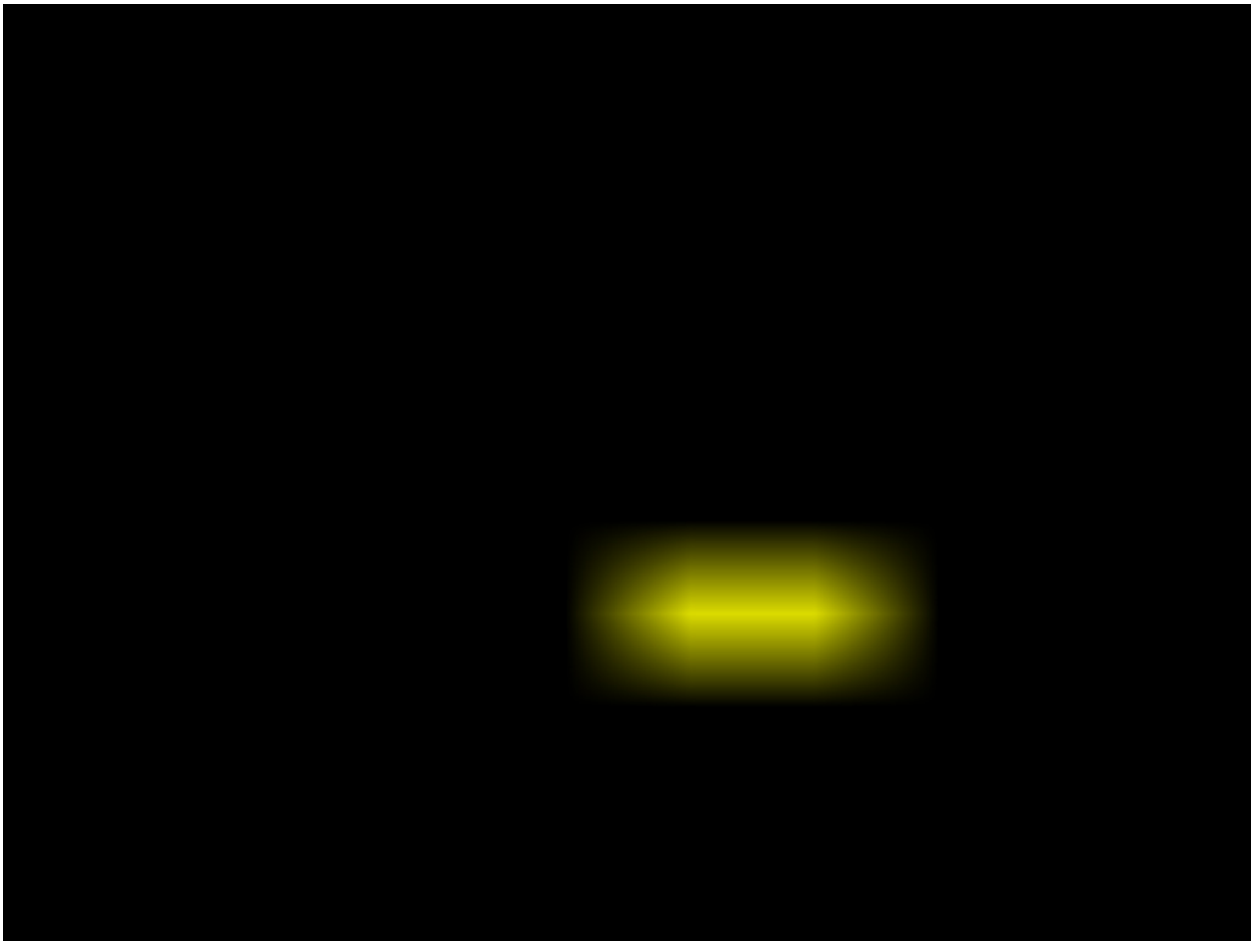


Figure C.8: Color mask results from the `FCN-ResNet18-VOC-320x320` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.9 freiburg-forest-sim-01-fcn-resnet18-voc-512x320-color-mask.jpg

Figure C.9: Color mask results from the `FCN-ResNet18-VOC-512x320` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.10 freiburg-forest-sim-01-fcn-resnet18-sun-512x400-color-mask.jpg

Figure C.10: Color mask results from the `FCN-ResNet18-SUN-512x400` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.11 freiburg-forest-sim-01-fcn-resnet18-sun-640x512-color-mask.jpg

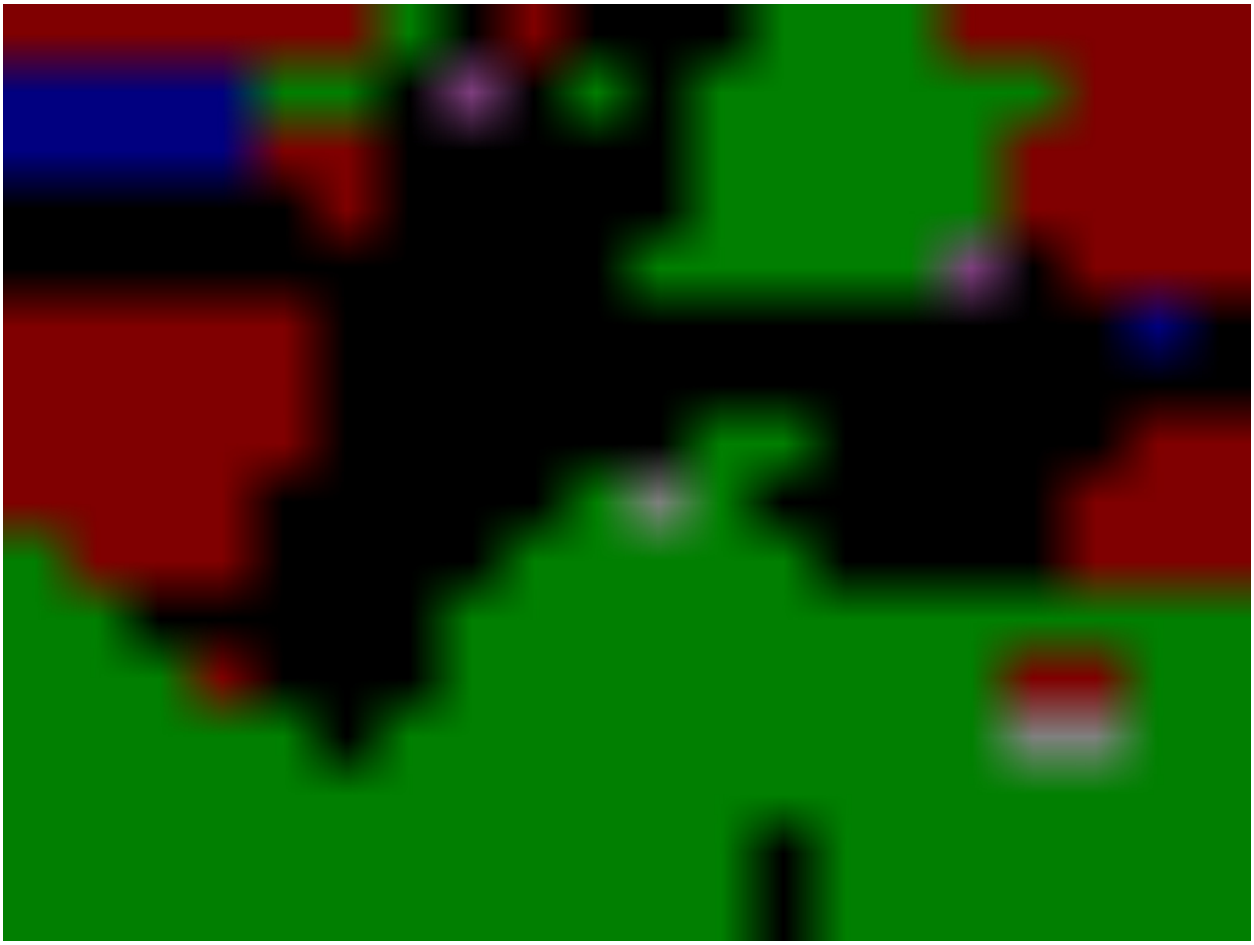


Figure C.11: Color mask results from the `FCN-ResNet18-SUN-640x512` image segmentation model used with the image from the Freiburg Forest dataset in Figure 5.1.

C.12 wilsonville-trail-sim-01-fcn-resnet18-cityscapes-512x256-color-mask.jpg

Figure C.12: Color mask results from the `FCN-ResNet18-Cityscapes-512x256` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.13 wilsonville-trail-sim-01-fcn-resnet18-cityscapes-1024x512-color-mask.jpg

Figure C.13: Color mask results from the `FCN-ResNet18-Cityscapes-1024x512` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.14 wilsonville-trail-sim-01-fcn-resnet18-cityscapes-2048x1024-color-mask.jpg

Figure C.14: Color mask results from the `FCN-ResNet18-Cityscapes-2048x1024` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.15 wilsonville-trail-sim-01-fcn-resnet18-deepscene-576x320-color-mask.jpg

Figure C.15: Color mask results from the `FCN-ResNet18-DeepScene-576x320` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.16 wilsonville-trail-sim-01-fcn-resnet18-deepscene-864x480-color-mask.jpg

Figure C.16: Color mask results from the `FCN-ResNet18-DeepScene-864x480` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.17 wilsonville-trail-sim-01-fcn-resnet18-mhp-512x320-color-mask.jpg

Figure C.17: Color mask results from the `FCN-ResNet18-MHP-512x320` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.18 wilsonville-trail-sim-01-fcn-resnet18-mhp-640x360-color-mask.jpg

Figure C.18: Color mask results from the `FCN-ResNet18-MHP-640x360` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.19 wilsonville-trail-sim-01-fcn-resnet18-voc-320x320-color-mask.jpg

Figure C.19: Color mask results from the `FCN-ResNet18-VOC-320x320` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.20 wilsonville-trail-sim-01-fcn-resnet18-voc-512x320-color-mask.jpg

Figure C.20: Color mask results from the `FCN-ResNet18-VOC-512x320` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.21 wilsonville-trail-sim-01-fcn-resnet18-sun-512x400-color-mask.jpg

Figure C.21: Color mask results from the `FCN-ResNet18-SUN-512x400` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.22 wilsonville-trail-sim-01-fcn-resnet18-sun-640x512-color-mask.jpg

Figure C.22: Color mask results from the `FCN-ResNet18-SUN-640x512` image segmentation model used with the captured video frame from Memorial Park in Figure 5.3.

C.23 oregon-tech-parking-lot-01-fcn-resnet18-cityscapes-512x256-color-mask.jpg

Figure C.23: Color mask results from the `FCN-ResNet18-Cityscapes-512x256` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.24 oregon-tech-parking-lot-01-fcn-resnet18-cityscapes-1024x512-color-mask.jpg

Figure C.24: Color mask results from the `FCN-ResNet18-Cityscapes-1024x512` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.25 oregon-tech-parking-lot-01-fcn-resnet18-cityscapes-2048x1024-color-mask.jpg

Figure C.25: Color mask results from the `FCN-ResNet18-Cityscapes-2048x1024` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.26 oregon-tech-parking-lot-01-fcn-resnet18-deepscene-576x320-color-mask.jpg

Figure C.26: Color mask results from the `FCN-ResNet18-DeepScene-576x320` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.27 oregon-tech-parking-lot-01-fcn-resnet18-deepscene-864x480-color-mask.jpg

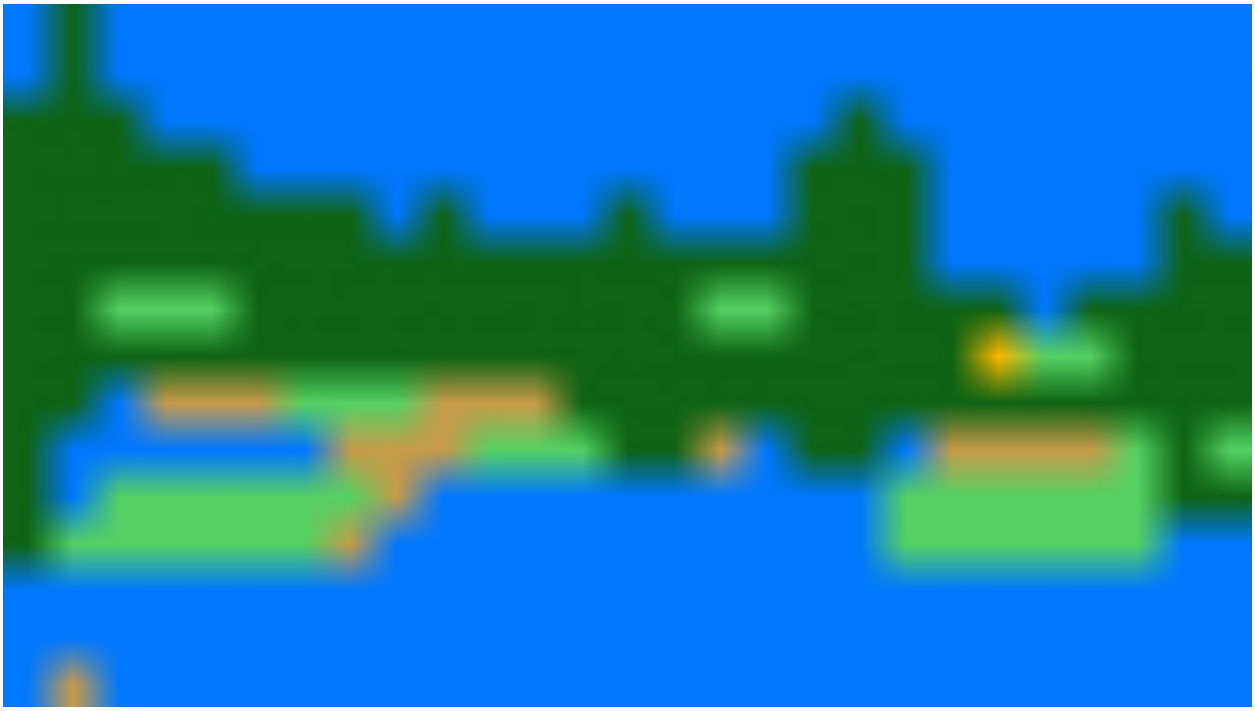


Figure C.27: Color mask results from the `FCN-ResNet18-DeepScene-864x480` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.28 oregon-tech-parking-lot-01-fcn-resnet18-mhp-512x320-color-mask.jpg

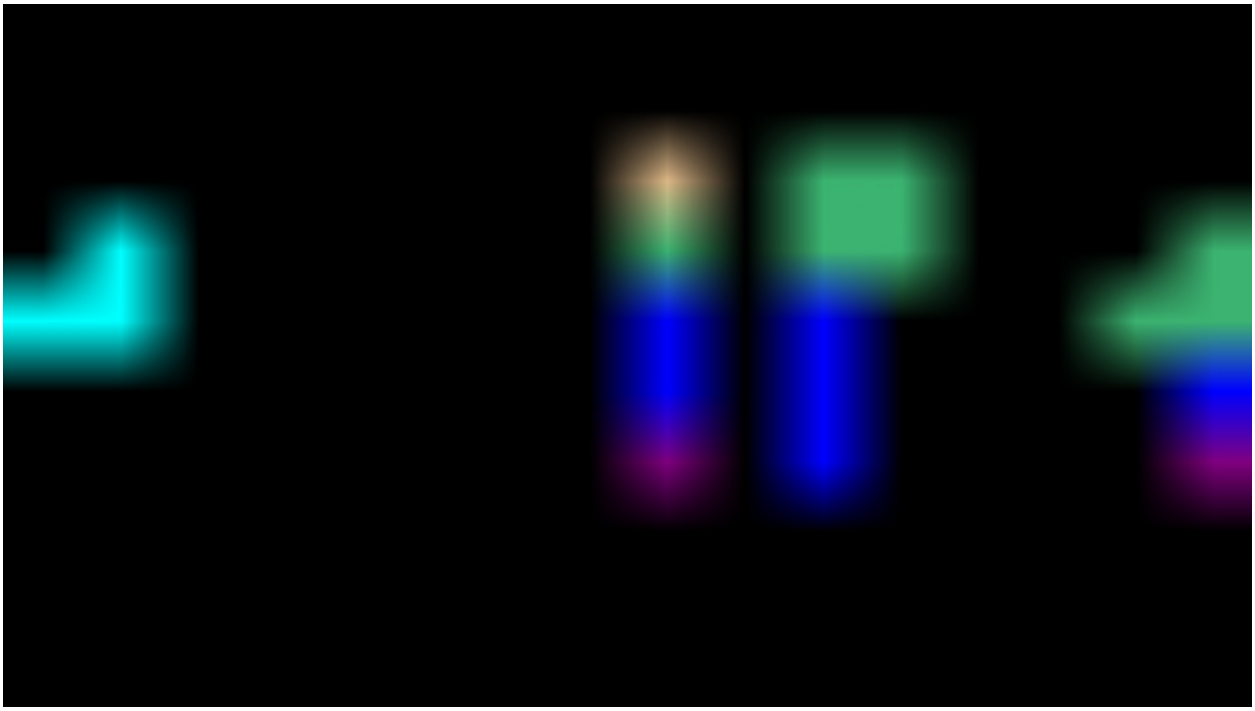


Figure C.28: Color mask results from the `FCN-ResNet18-MHP-512x320` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.29 oregon-tech-parking-lot-01-fcn-resnet18-mhp-640x360-color-mask.jpg

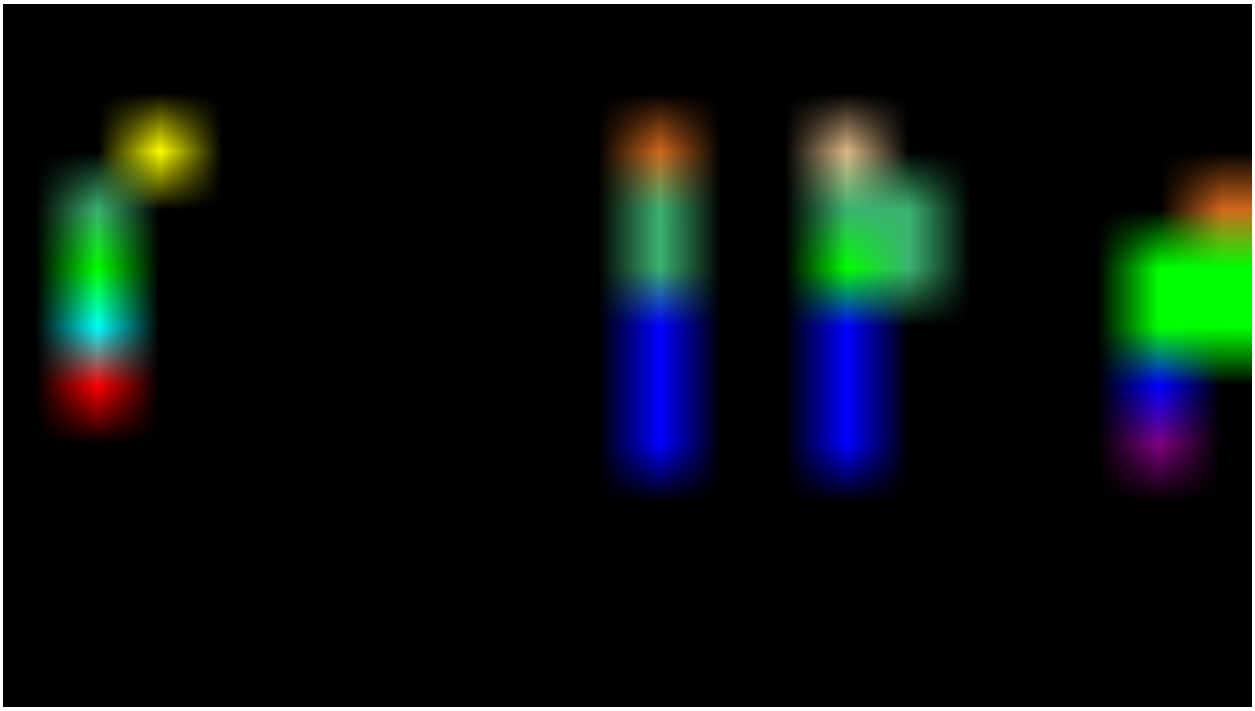


Figure C.29: Color mask results from the `FCN-ResNet18-MHP-640x360` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.30 oregon-tech-parking-lot-01-fcn-resnet18-voc-320x320-color-mask.jpg

Figure C.30: Color mask results from the `FCN-ResNet18-VOC-320x320` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.31 oregon-tech-parking-lot-01-fcn-resnet18-voc-512x320-color-mask.jpg

Figure C.31: Color mask results from the `FCN-ResNet18-VOC-512x320` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.32 oregon-tech-parking-lot-01-fcn-resnet18-sun-512x400-color-mask.jpg

Figure C.32: Color mask results from the `FCN-ResNet18-SUN-512x400` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.33 oregon-tech-parking-lot-01-fcn-resnet18-sun-640x512-color-mask.jpg

Figure C.33: Color mask results from the `FCN-ResNet18-SUN-640x512` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.11.

C.34 oregon-tech-parking-lot-02-fcn-resnet18-cityscapes-512x256-color-mask.jpg

Figure C.34: Color mask results from the `FCN-ResNet18-Cityscapes-512x256` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

C.35 oregon-tech-parking-lot-02-fcn-resnet18-cityscapes-1024x512-color-mask.jpg

Figure C.35: Color mask results from the `FCN-ResNet18-Cityscapes-1024x512` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

C.36 oregon-tech-parking-lot-02-fcn-resnet18-cityscapes-2048x1024-color-mask.jpg

Figure C.36: Color mask results from the `FCN-ResNet18-Cityscapes-2048x1024` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

C.37 oregon-tech-parking-lot-02-fcn-resnet18-deepscene-576x320-color-mask.jpg

Figure C.37: Color mask results from the `FCN-ResNet18-DeepScene-576x320` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

C.38 oregon-tech-parking-lot-02-fcn-resnet18-deepscene-864x480-color-mask.jpg

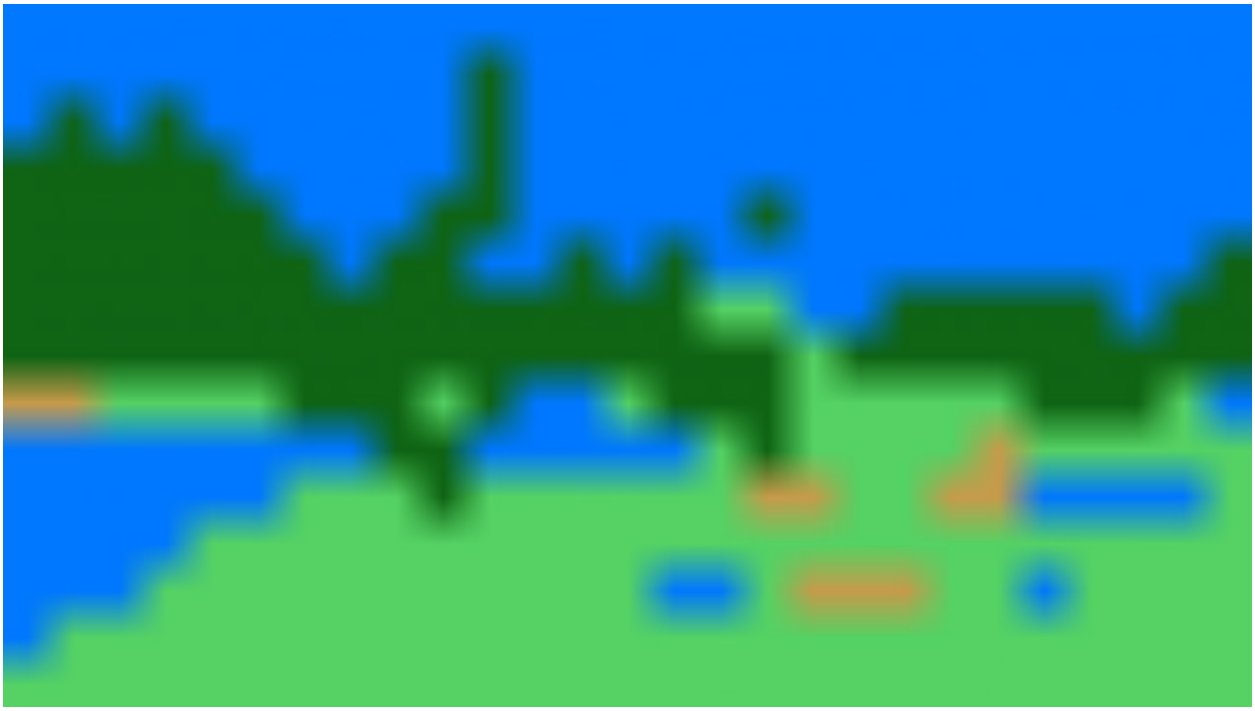


Figure C.38: Color mask results from the `FCN-ResNet18-DeepScene-864x480` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

C.39 oregon-tech-parking-lot-02-fcn-resnet18-mhp-512x320-color-mask.jpg

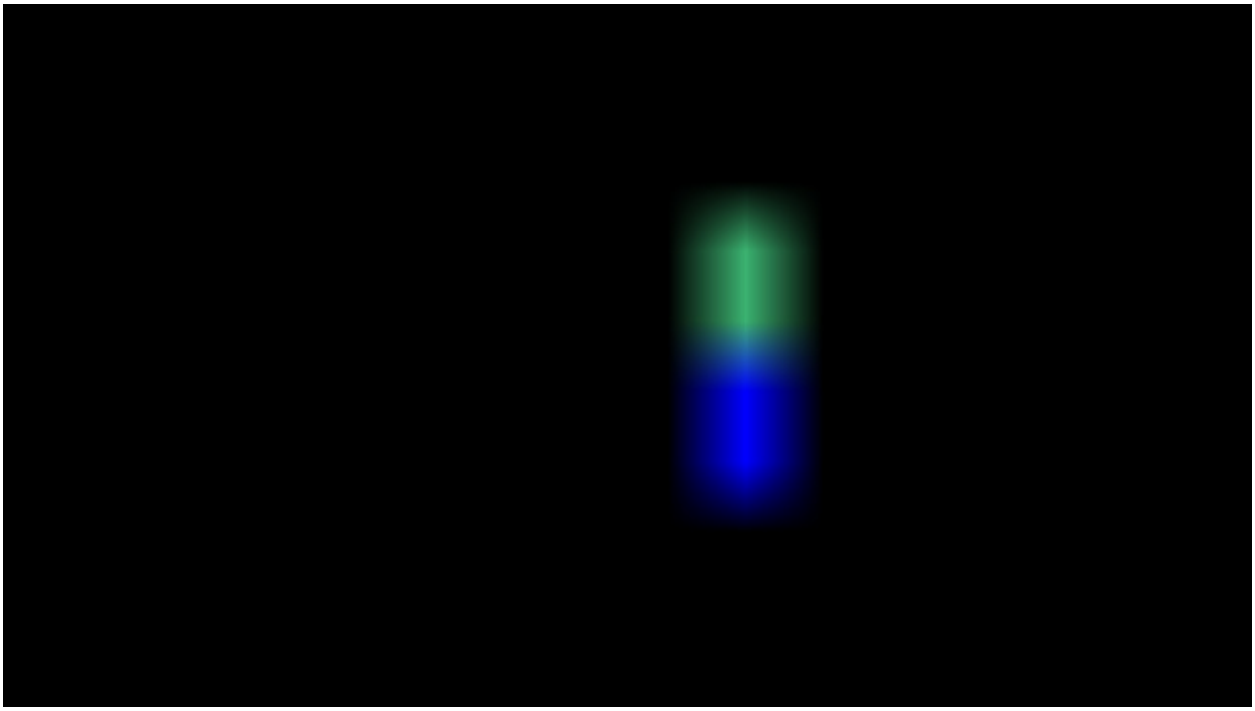


Figure C.39: Color mask results from the `FCN-ResNet18-MHP-512x320` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

C.40 oregon-tech-parking-lot-02-fcn-resnet18-mhp-640x360-color-mask.jpg

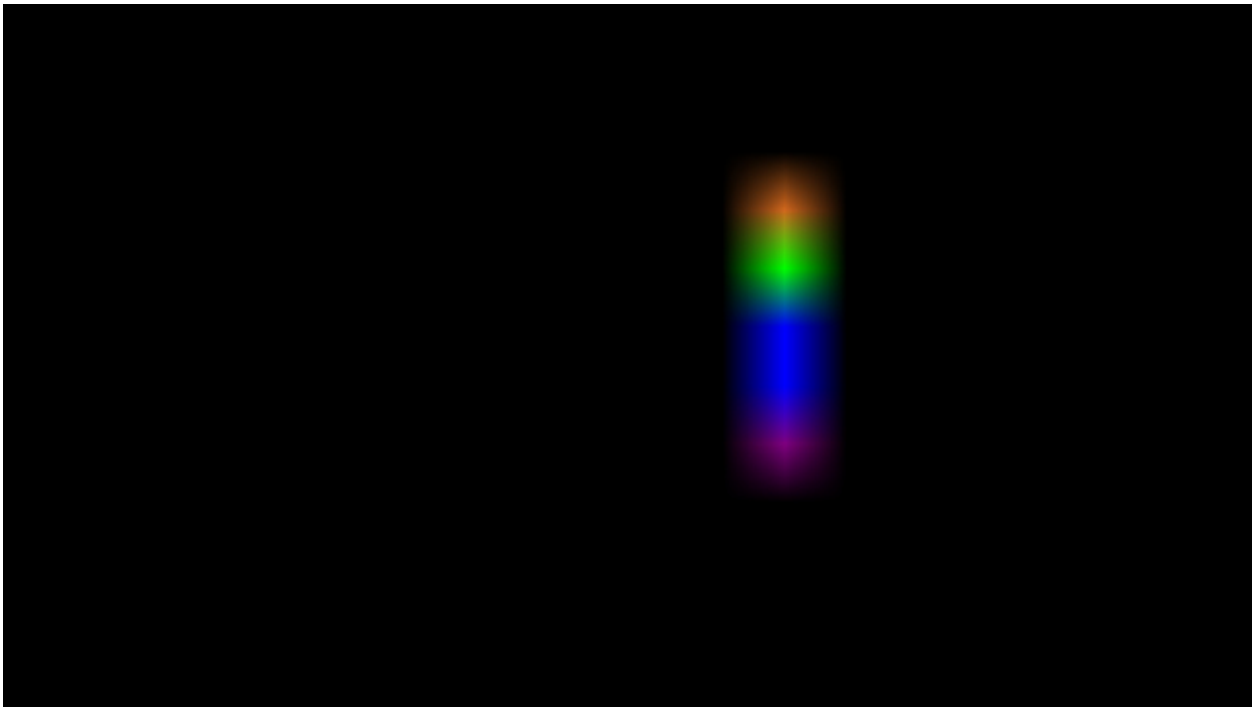


Figure C.40: Color mask results from the `FCN-ResNet18-MHP-640x360` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

C.41 oregon-tech-parking-lot-02-fcn-resnet18-voc-320x320-color-mask.jpg

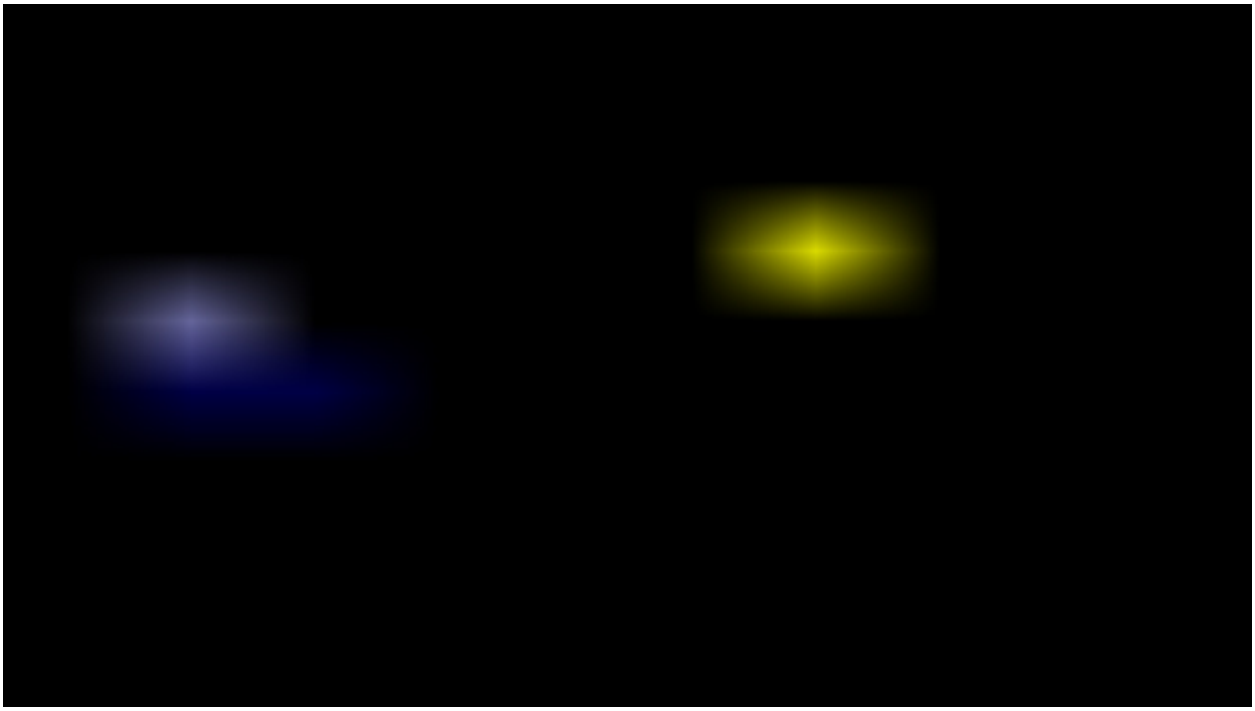


Figure C.41: Color mask results from the `FCN-ResNet18-VOC-320x320` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

C.42 oregon-tech-parking-lot-02-fcn-resnet18-voc-512x320-color-mask.jpg

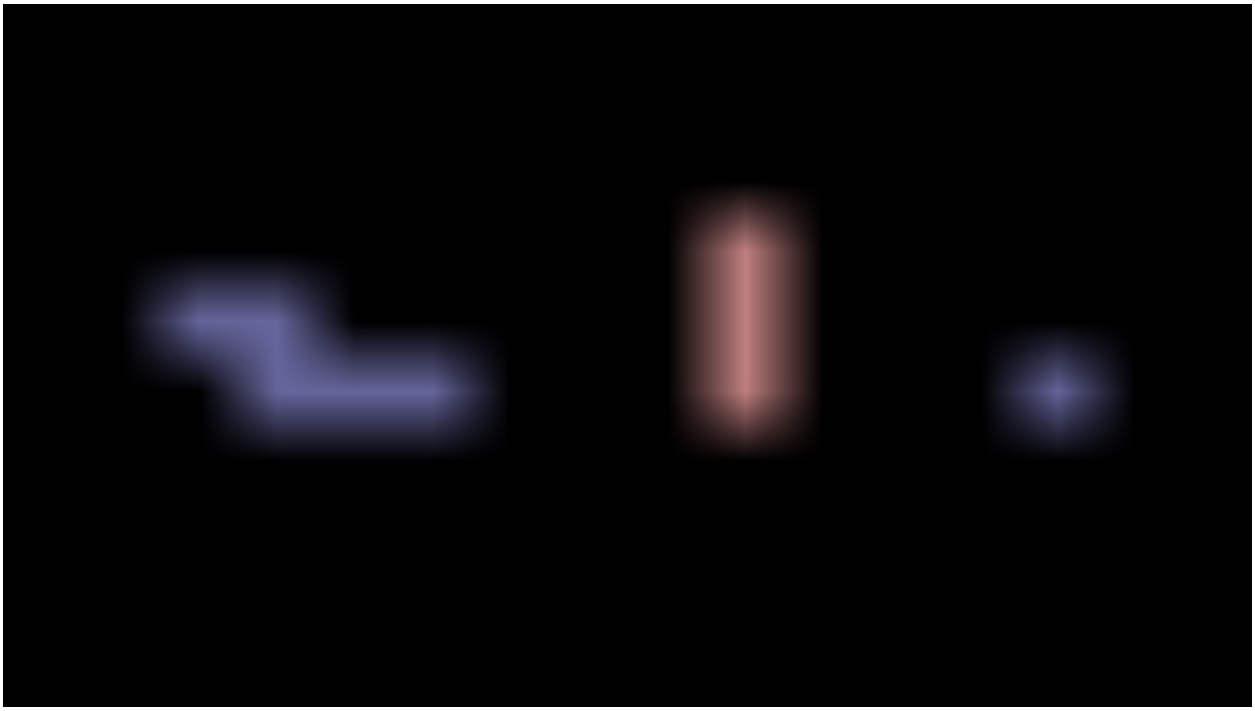


Figure C.42: Color mask results from the `FCN-ResNet18-VOC-512x320` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

C.43 oregon-tech-parking-lot-02-fcn-resnet18-sun-512x400-color-mask.jpg

Figure C.43: Color mask results from the `FCN-ResNet18-SUN-512x400` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

C.44 oregon-tech-parking-lot-02-fcn-resnet18-sun-640x512-color-mask.jpg

Figure C.44: Color mask results from the `FCN-ResNet18-SUN-640x512` image segmentation model used with the captured video frame from the Oregon Tech parking lot in Figure 5.12.

References

[1] Committee on Army Unmanned Ground Vehicle Technology, *Technology Development for Army Unmanned Ground Vehicles*. National Academies Press, 2003, p. 2.

[2] M. Montemerlo, S. Thrun, H. Dahlkamp, D. Stavens, and S. Strohband, "Winning the DARPA Grand Challenge with an AI Robot," in *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, ser. AAAI'06. AAAI Press, 2006, pp. 982–987. [Online]. Available: http://www.aaai.org/Papers/AAAI/2006/AAAI06-154.pdf

[3] Y. Liu, Z. Chen, W. Zheng, H. Wang, and J. Liu, "Monocular Visual-Inertial SLAM: Continuous Preintegration and Reliable Initialization," *Sensors*, vol. 17, no. 11, 2017. [Online]. Available: http://www.mdpi.com/1424-8220/17/11/2613

[4] S. P. Adhikari, C. Yang, K. Slot, and H. Kim, "Accurate Natural Trail Detection Using a Combination of a Deep Neural Network and Dynamic Programming," *Sensors*, vol. 18, no. 1, 2018. [Online]. Available: http://www.mdpi.com/1424-8220/18/1/178

[5] T. M. Heggie and T. W. Heggie, "Search and Rescue Trends Associated With Recreational Travel in US National Parks," *Journal of Travel Medicine*, vol. 16, no. 1, pp. 23–27, January 2009. [Online]. Available: https://dx.doi.org/10.1111/j.1708-8305.2008.00269.x

[6] J. G. Liu and P. Mason, *Image Processing and GIS for Remote Sensing: Techniques and Applications*, 2nd ed. Hoboken, NJ: John Wiley & Sons, Incorporated, 2016, ch. 8, pp. 91–92, Accessed on: May, 21, 2019. [Online]. Available: https://onlinelibrary-wiley-com.libproxy.oit.edu/doi/pdf/10.1002/9781118724194.

[7] A. Bosch, A. Zisserman, and X. Munoz, "Image Classification using Random Forests and Ferns," in *2007 IEEE 11th International Conference on Computer Vision*, Oct 2007, pp. 1–8, Accessed on: May, 21, 2019. [Online]. Available: http://www.cs.huji.ac.il/~daphna/course/CoursePapers/bosch07a.pdf.

[8] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[9] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[10] Y. Aytar, "The PASCAL Visual Object Classes Homepage," *host.robots.ox.ac.uk*, Jan. 2019, [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/. [Accessed: May 30, 2019].

[11] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242

[12] Common Objects in Context Collaborators, "COCO - Common Objects in Context," *cocodataset.org*, Nov. 2018, [Online]. Available: http://cocodataset.org/. [Accessed: May 30, 2019].

[13] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *CoRR*, vol. abs/1804.02767, 2018. [Online]. Available: http://arxiv.org/abs/1804.02767

[14] S. Gould, T. Gao, and D. Koller, "Region-based segmentation and object detection," in *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, Eds. Curran Associates, Inc., 2009, pp. 655–663. [Online]. Available: http://papers.nips.cc/paper/3766-region-based-segmentation-and-object-detection.pdf

[15] A. Giusti, J. Guzzi, D. C. Cireşan, F. He, J. P. Rodríguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. D. Caro, D. Scaramuzza, and L. M. Gambardella, "A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, pp. 661–667, July 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7358076

[16] H. Durrant-Whyte and T. Bailey, "Simultaneous Localization and Mapping: Part I," *IEEE Robotics Automation Magazine*, vol. 13, no. 2, pp. 99–110, June 2006. [Online]. Available: https://people.eecs.berkeley.edu/~pabbeel/cs287-fa09/readings/Durrant-Whyte_Bailey_SLAM-tutorial-I.pdf

[17] J. Engel, T. Schöps, and D. Cremers, "LSD-SLAM: Large-Scale Direct Monocular SLAM," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 834–849. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.646.7193&rep=rep1&type=pdf

[18] J. Prado, F. Yandun, M. T. Torriti, and F. A. Cheein, "Overcoming the Loss of Performance in Unmanned Ground Vehicles Due to the Terrain Variability," *IEEE Access*, vol. 6, pp. 17 391–17 406, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8325418

[19] C. Wang, "Monocular Vision-Based Obstacle Detection for Unmanned Systems," Master's thesis, Mech. Eng., Univ. of Waterloo, Ontario, 2011, accessed on: Jan., 29, 2019. [Online]. Available: http://hdl.handle.net/10012/5890http://hdl.handle.net/10012/5890. [Online]. Available: http://hdl.handle.net/10012/5890

[20] J. Allard, D. S. Barrett, M. Filippov, R. T. Pack, and S. Svendsen, "Systems and Methods for Control of an Unmanned Ground Vehicle," U.S. Patent 7,499,776, March 2009.

[21] J. R. Fox, A. R. Jones, N. T.-H. Liu, and B. Sivasubramanian, "Lost Person Rescue Drone," U.S. Patent 9,915,945, March 2018.

[22] B. Yamauchi, "Autonomous Mobile Robot," U.S. Patent 7,539,557, May 2009.

[23] M. Wysokiński, R. Marcjan, and J. Dajda, "Decision Support Software for Search & Rescue Operations," *Procedia Computer Science*, vol. 35, pp. 776 – 785, 2014, knowledge-Based and Intelligent Information & Engineering Systems 18th Annual Conference, KES-2014 Gdynia, Poland, September 2014 Proceedings. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050914011259

[24] P. T. Thavasi and C. Suriyakala, "Sensors and Tracking Methods Used in Wireless Sensor Network Based Unmanned Search and Rescue System - A Review," *Procedia Engineering*, vol. 38, pp. 1935 – 1945, 2012, INTERNATIONAL CONFERENCE ON MODELLING OPTIMIZATION AND COMPUTING. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877705812021492

[25] S. Karma, E. Zorba, G. Pallis, G. Statheropoulos, I. Balta, K. Mikedi, J. Vamvakari, A. Pappa, M. Chalaris, G. Xanthopoulos, and M. Statheropoulos, "Use of Unmanned Vehicles in Search and Rescue Operations in Forest Fires: Advantages and Limitations Observed in a Field Trial," *International Journal of Disaster Risk Reduction*, vol. 13, pp. 307 – 312, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2212420915300364

[26] ROS.org, "melodic - ROS Wiki," *wiki.ros.org*, Aug. 2018, [Online]. Available: http://wiki.ros.org/turtlesim. [Accessed: Dec. 13, 2019].

[27] ROS.org, "Nodes - ROS Wiki," *wiki.ros.org*, Dec. 2018, [Online]. Available: http://wiki.ros.org/Nodes. [Accessed: Dec. 13, 2019].

[28] ROS.org, "Master - ROS Wiki," *wiki.ros.org*, Jan. 2018, [Online]. Available: http://wiki.ros.org/Master. [Accessed: Dec. 13, 2019].

[29] ROS.org, "turtlesim - ROS Wiki," *wiki.ros.org*, Oct. 2019, [Online]. Available: http://wiki.ros.org/melodic. [Accessed: Dec. 13, 2019].

[30] NVIDIA, "Hardware For Every Situation | NVIDIA Developer," *developer.nvidia.com*, Dec. 2019, [Online]. Available: https://developer.nvidia.com/embedded/develop/hardware. [Accessed: Dec. 13, 2019].

[31] Embedded Linux Wiki, "Jetson TX2 - eLinux.org," *elinux.org*, Dec. 2019, [Online]. Available: https://elinux.org/Jetson_TX2. [Accessed: Dec. 13, 2019].

[32] Logitech, "Logitech Webcam C930e," Logitech Webcam C930e datasheet, July 2017, Accessed: Dec. 13, 2019. [Online]. Available: https://www.logitech.com/assets/64665/c930edatasheet.ENG.pdf.

[33] Logitech, "Logitech C930e 1080p HD Webcam with H.264 Compression & Wide Field of View," *logitech.com*, Dec. 2019, [Online]. Available: https://www.logitech.com/en-us/product/c930e-webcam. [Accessed: Dec. 13, 2019].

[34] HighPoint Technologies, Inc., "RocketU 1344A 4-Port USB 3.1 PCI-Express 3.0 x4 USB Card - HighPoint Global Website," *Highpoint-tech.com*, Jul. 2020, [Online]. Available: https://highpoint-tech.com/USA˙new/series-ru1344a-overview.htm. [Accessed: Oct. 31, 2020].

[35] HighPoint Technologies, Inc., "RocketU 1344A PCI-Express 3.0 Host Interface, 4x USB 3.2 Gen 2 Type-A Ports," *Highpoint-tech.com*, Jun. 2020, [Online]. Available: https://highpoint-tech.com/PDF/RU/RU1344A/RU1344A_Datasheet_20_06_26.pdf. [Accessed: Oct. 31, 2020].

[36] Samsung, "Samsung V-NAND SSD 860 EVO 2018 Data Sheet, Rev. 1.0," *Samsung.com*, Dec. 2018, [Online]. Available: https://www.samsung.com/semiconductor/global.semi.static/Samsung_SSD_860_EVO_Data_Sheet_Rev1.pdf. [Accessed: Nov. 4, 2020].

[37] Samsung, "SSD 860 EVO 2.5" SATA III 500GB Memory & Storage - MZ-76E500B/AM | Samsung US," *Samsung.com*, Nov. 2020, [Online]. Available: https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-860-evo-2-5-sata-iii-500gb-mz-76e500b-am/. [Accessed: Nov. 4, 2020].

[38] Amazon, "Amazon.com: SMAKN(TM) 22-pin (7+15) Sata Male to Female Data and Power Combo Extension Cable - Slimline Sata Extension Cable M/f - 20inch (50cm): Computers & Accessories," *Amazon.com*, Nov. 2020, [Online]. Available: https://www.amazon.com/gp/product/B00L9R3AKA. [Accessed: Nov. 4, 2020].

[39] Amazon, "Amazon.com: Bingfu WiFi Antenna Extension Cable (2-Pack) RP-SMA Male to RP-SMA Female Bulkhead Mount RG316 Cale 30cm 12 inch for WiFi Router Security IP Camera Wireless Mini PCI Express PCIE Network Card Adapter: Home Audio & Theater," *Amazon.com*, Nov. 2020, [Online]. Available: https://www.amazon.com/gp/product/B07MT3VZXZ. [Accessed: Nov. 5, 2020].

[40] Amazon, "Amazon.com: Noctua NF-A4x20 5V PWM, Premium Quiet Fan, 4-Pin, 5V Version (40x20mm, Brown): Computers & Accessories," *Amazon.com*, Nov. 2020, [Online]. Available: https://www.amazon.com/Noctua-NF-A4x20-5V-PWM-Premium-Quality/dp/B071FNHVXN. [Accessed: Nov. 5, 2020].

[41] Amazon, "Amazon.com: MakerFocus 4pcs Raspberry Pi 4B Fan DC Brushless Cooling Fan 3.3V 5V Heatsink Cooler Radiator Connector Separating One-to-Two Interface for Raspberry Pi 4/ Pi 3/3B+ and Pi Zero/Zero W or Other Project: Computers & Accessories," *Amazon.com*, Nov. 2020, [Online]. Available: https://www.amazon.com/gp/product/B07D3TVVC7/. [Accessed: Nov. 5, 2020].

[42] Altelix, "Altelix NP Series Polycarbonate + ABS Weatherproof NEMA Enclosure Datasheeet," *Altelix.com*, Nov. 2020, [Online]. Available: https://docs.altelix.com/datasheets/DS˙NP˙SERIES.pdf. [Accessed: Nov. 5, 2020].

[43] Altelix, "Altelix 14x11x5 Green Polycarbonate + ABS Weatherproof NEMA Enclosure with Aluminum Equipment Mounting Plate - Altelix," *Altelix.com*, Nov. 2020, [Online]. Available: https://altelix.com/altelix-14x11x5-green-polycarbonate-abs-weatherproof-nema-enclosure-with-aluminum-equipment-mounting-plate/. [Accessed: Nov. 5, 2020].

[44] NVIDIA, "Jetpack 4.3 Archive | NVIDIA Developer," *developer.nvidia.com*, Nov. 2020, [Online]. Available: https://developer.nvidia.com/jetpack-43-archive. [Accessed: Nov. 12, 2020].

[45] NVIDIA, "L4T | NVIDIA Developer," *developer.nvidia.com*, Nov. 2020, [Online]. Available: https://developer.nvidia.com/embedded/linux-tegra. [Accessed: Nov. 12, 2020].

[46] D. Franklin, C. Yato, and D. Nguyen, "dusty-nv/jetson-inference: Hello AI World guide to deploying deep-learning inference networks and deep vision primitives with TensorRT and NVIDIA Jetson." *github.com*, Nov. 2020, [Online]. Available: https://github.com/dusty-nv/jetson-inference/. [Accessed: Nov. 17, 2020].

[47] J. Zhao, J. Li, Y. Cheng, L. Zhou, T. Sim, S. Yan, and J. Feng, "Understanding humans in crowded scenes: Deep nested adversarial learning and A new benchmark for multi-human parsing," *CoRR*, vol. abs/1804.03287, 2018. [Online]. Available: http://arxiv.org/abs/1804.03287

[48] Cityscapes Team, "Cityscapes Dataset – Semantic Understanding of Urban Street Scenes," *cityscapes-dataset.com*, Mar. 2021, [Online]. Available: https://www.cityscapes-dataset.com. [Accessed: Mar. 3, 2021].

[49] A. Valada, "DeepScene," *deepscene.cs.uni-freiburg.de*, Mar. 2021, [Online]. Available: http://deepscene.cs.uni-freiburg.de. [Accessed: Mar. 3, 2021].

[50] J. Feng, J. Zhao, J. Li, X. Nie, and D. Hu, "MHP:Multi-Human Parsing," *lv-mhp.github.io*, Nov. 2018, [Online]. Available: https://lv-mhp.github.io. [Accessed: Mar. 3, 2021].

[51] Princeton Vision & Robotics Labs, "SUN RGB-D: A RGB-D Scene Understanding Benchmark Suite," *rgbd.cs.princeton.edu*, Dec. 2020, [Online]. Available: http://rgbd.cs.princeton.edu. [Accessed: Mar. 3, 2021].

[52] A. Valada, G. Oliveira, T. Brox, and W. Burgard, "Deep multispectral semantic scene understanding of forested environments using multimodal fusion," in *International Symposium on Experimental Robotics (ISER)*, 2016.

[53] Valgrind™ Developers, "Valgrind," *valgrind.org*, Mar. 2021, [Online]. Available: https://valgrind.org/docs/manual/cl-manual.html. [Accessed: Mar. 9, 2021].

[54] KDE Webmasters, "KCachegrind," *apps.kde.org*, Mar. 2021, [Online]. Available: https://apps.kde.org/en/kcachegrind. [Accessed: Mar. 9, 2021].

[55] Allied Vision Technologies GmbH, "MIPI CSI-2: The embedded-ready interface - Allied Vision," *alliedvision.com*, Mar. 2021, [Online]. Available: https://www.alliedvision.com/en/products/introduction-to-embedded-vision-cameras/mipi-csi-2-the-embedded-ready-interface.html. [Accessed: Mar. 28, 2021].

[56] FLIR® Systems, Inc., "THERMAL STEREO VISION MAKES SAFER CARS," *flir.com*, Nov. 2020, [Online]. Available: https://www.flir.com/globalassets/discover/oem/oem-cores-thermal-stereo-vision.pdf. [Accessed: Mar. 28, 2021].