4-24-2019

In Situ Over-the-Air Software Update Verification

Alex Michael Dillon

Arvind Carlton Draffen

In Situ Over-the-Air Software Update Verification

by

Alex Michael Dillon
Arvind Carlton Draffen

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

4/24/2019

Honors Capstone Director: Dr. Rhonda Gaede, Associate Professor, Electrical & Computer Engineering Department

[illegible]	5/1/19
Student	Date
[illegible]	5/1/19
Student	Date
Rhonda Kay Gaede	5/1/19
Director	Date
[illegible]	5/1/19
Department Chair	Date
[illegible]	5/6/19
Honors College Dean	Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

Alex Dillon

Student Name (printed)

[illegible]

Student Signature

5/1/19

Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

Arvind Draffen

Student Name (printed)

[illegible]

Student Signature

5/1/19

Date

In Situ Over-the-Air Software Update Verification

by

Alex Michael Dillon
Arvind Carlton Draffen

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

4/24/2019

Honors Capstone Director: Dr. Rhonda Gaede, Associate Professor, Electrical & Computer Engineering Department

Student	Date
Student	Date
Director	Date
Department Chair	Date
Honors College Dean	Date

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

____

Student Name (printed)

____

Student Signature

____

Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

____

Student Name (printed)

____

Student Signature

____

Date

Table of Contents

Abstract

Very few engineering disciplines have evolved as rapidly as software engineering. One of the largest interest areas for modern software systems is improving the testability and reliability of complex computer systems. Until now, exhaustive and large-scale unit testing has been an industry standard methodology for verification of software functionality. However, the nearly non-deterministic nature of machine learning and artificial intelligence algorithms presents an intractable testability problem. When presented with new input data, the behavior of some algorithms can only be determined experimentally. In safety-critical systems, such as self-driving cars, the inability to effectively test machine learning algorithms presents a substantial problem. One potential solution to this challenge is the use of In Situ Over-the-Air Software Update Verification. In critical systems, by using virtualization techniques and a "difference engine", new software can be tested side-by-side with existing software, allowing for mass-scale verification of unknown algorithms. By doing so wirelessly, in-place, end users would notice no differences during operation, yet software engineers would be able to collect extensive amounts of real-world test data. This data empowers engineers to experiment with both iterative improvements to algorithms as well as innovative complete overhauls and empirically verify correctness.

Project Goals

Development of automated and autonomous systems is at the forefront of both industry and research, with a variety of products having made their way into the business and consumer markets and many more being created and tested for viability in attempts to penetrate these markets. Such products range from the speech processing found in smart home hubs to the lane departure warning systems available in many modern vehicles. Such systems require large amounts of training data for appropriate responses to various stimuli and, as a result, produce voluminous data for developers to analyze. Not only are these systems complex, their malfunction can have serious and potentially fatal consequences, thus, their proper function is of paramount importance. As many technologies in this field are in early stages of existence, there are many improvements to be made in their development. New updates to existing software are being produced at prodigious rates in attempts to provide the most extensive set of features and robust products. The complexity of these systems exposes the limitations of longstanding testing techniques and some newly-developed systems can only be tested through analysis of empirical evidence.

The primary goal of this project is to provide a solution to this testing problem by producing a comprehensive analysis of the outputs from a set of neural networks where differences in outputs can be noted and classified based on their severity. This solution, the Difference Engine, is based upon analyzing differences in two neural networks for image detection that run in conjunction with the Robot Operating System (ROS), an open-source middleware which can be used in development of semi-autonomous and autonomous vehicles. The output of the Difference Engine should be reported in a format that allows for further analysis of the noted differences. Although the Difference Engine as initially developed integrates with ROS, extensibility was a key factor in the design process as there is not a standard output format shared between neural networks.

Development of the Difference Engine

Topology and Setup

Before development could commence on the Difference Engine, a test environment needed to be established with a pair of neural networks operating on real-world sensor data and producing realistic outputs. This test environment is shown in Figure 1. A choice was made to have neural networks detect objects from a camera feed on ROS-networked Raspberry Pis. One Raspberry Pi serves as the "master" unit and has the direct camera connection. The master Pi is also responsible for running the Difference Engine software and being the ultimate consumer of the object detection data. In order to perform the neural network processing, two additional Raspberry Pis were utilized, "sub1" and "sub2", respectively. Due to the computational complexity of running a neural network, each Raspberry Pi was only capable of running a single network. In this topology, the sub Raspberry Pi units act as ROS Subscriber Nodes and consume raw camera data being published from the master. Independently, each Pi runs the raw frame data through neural network processing and produces a result set detailing the objects detected within the image. Finally, each Pi also operates as a ROS Publisher Node, publishing the processed data back into the ROS Network. From there, it is available to the Difference Engine, along with any other potential nodes in the system that seek to consume the results.

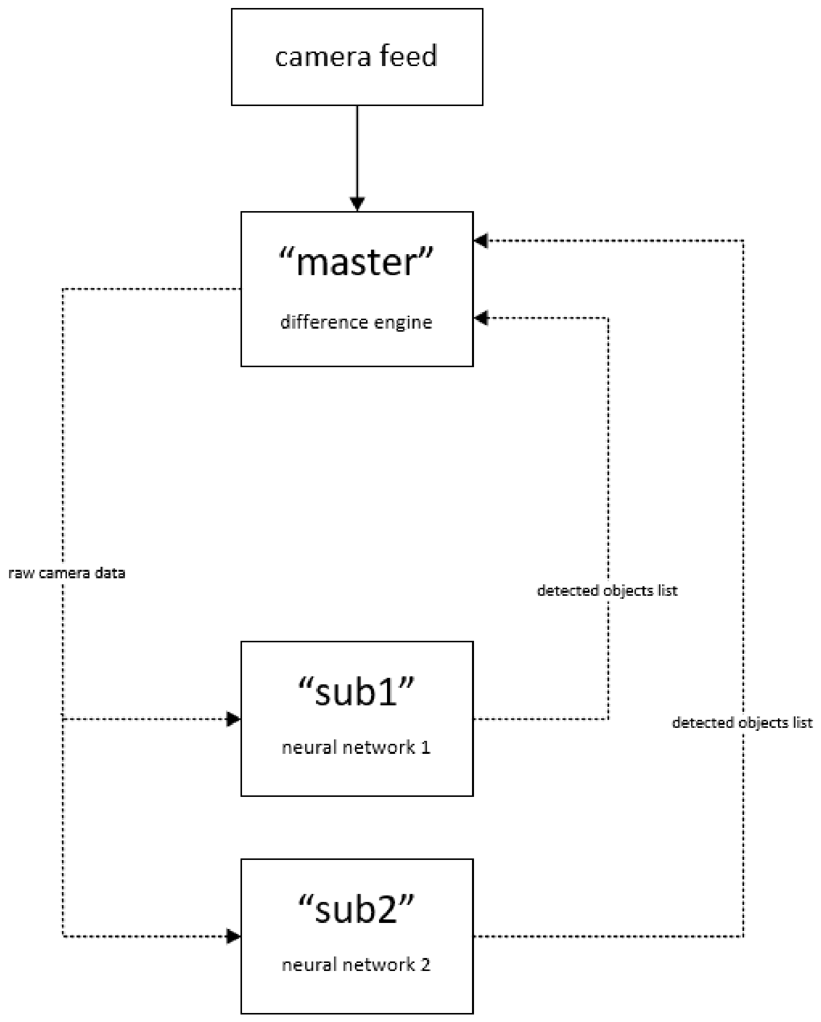
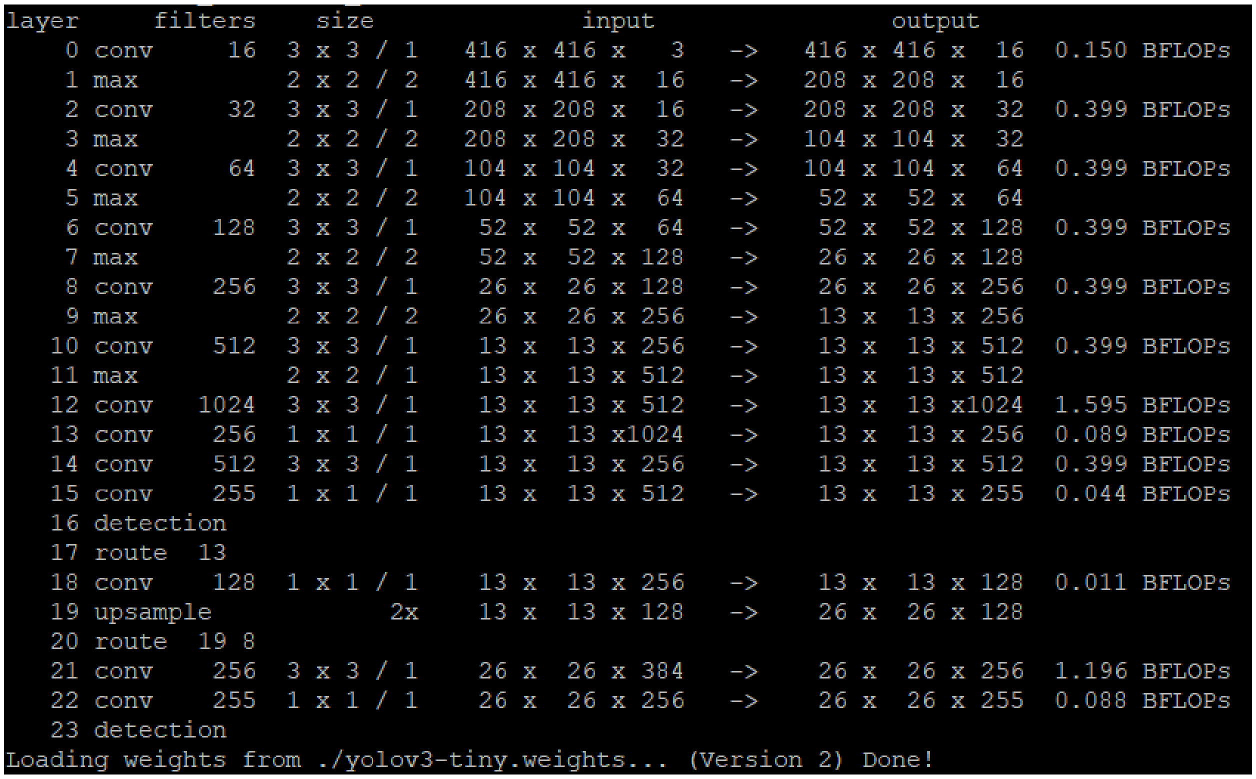


Figure 1: System Topology

After an extensive period of research, the Darknet + NNPACK Neural Network, operating with the YOLO Object Detection Model was selected as the primary neural network for this project [1]. Consideration was given to TensorFlow and Keras. TensorFlow is a popular neural network framework which performs numerical computation using data flow graphs. It is supported by Google, open source, and is a highly popular neural network [2]. The first choice was using a neural network based on either Keras or TensorFlow, however neither of these neural networks support Python 2.7 which is used by ROS [2]. Ultimately, the Darkness + NNPACK neural network provided the necessary software support for ROS and hardware support via optimizations for the limited computational ability of the Raspberry Pi, as will be discussed.

Darknet is an open source neural network framework developed in C, which is designed to be fast, and also supports the fast You Only Look Once (YOLO) detection algorithm [3, 4]. Since the Raspberry Pi is a low-power embedded system and has far fewer system resources than desktop computers and lacks the capability to perform Graphic Processing Unit (GPU) acceleration, efficiency and optimization is key. To further improve performance, a variant of Darknet that employs NNPACK optimizations was selected [5]. NNPACK is an open source acceleration package that allows neural networks to more efficiently use multi-core processors, like the Quad Core ARM Cortex A53 CPU in the Raspberry Pi, in lieu of a GPU [5]. The amalgamation of these components results in a neural network implementation that can perform object detection in real-time, at approximately 1 frame per second [1]. Darknet was provided with a Python wrapper for the C code, which was expanded to form the basis for the test application running on each sub Pi [1]. Modifications to the Python code were required in order to subscribe to camera data over ROS, as well as to publish results back into the ROS system instead of displaying them locally. Additionally, slight modifications to both the Python and Darknet C code were required to extract vector data corresponding to detections instead of just an annotated image. Furthermore, several bug fixes were implemented in both components to improve reliability and increase performance. Once completed, the Darknet + NNPACK with YOLO neural network successfully was tested on each subscriber Pi, and was verified to be correctly receiving image data, processing and detecting object, and republishing the results back to ROS.

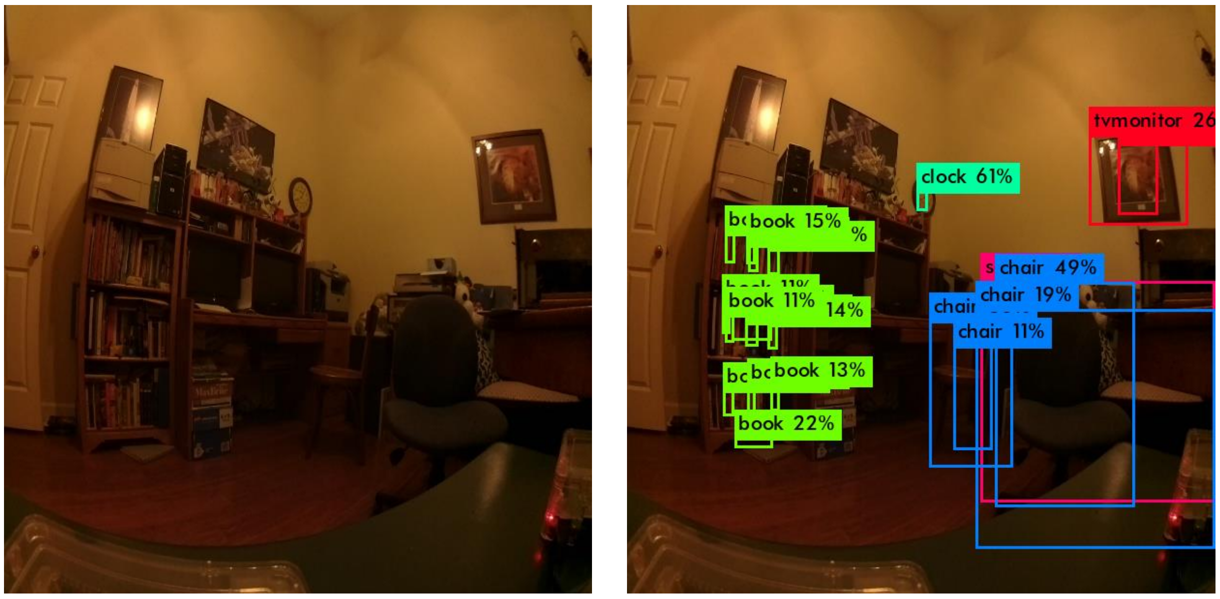
While exact specification for the Neural Network were not a primary focus of the Difference Engine project since it is designed to be versatile and extensible for any source of data, careful selection of the network was still a significant portion of the project. Not only did the Neural Network need to perform in real-time on the provided Raspberry Pi hardware, it needed to provide detailed information to facilitate comparison operations. As previously mentioned, the YOLO v3 Tiny model was selected to facilitate high-speed, real-time operation. YOLO is a Convolutional Neural Network, which, in the Tiny version, uses a total of 25 layers [4]. In Figure 2, the weight configuration employed in the Difference Engine project is detailed. The 13 Convolutional Layers (varying between 3x3 and 1x1 sized), 6 Maximum Pooling Layers, and 5 additional assorted layers that form the complete network for the object detection process can be observed in figure as well.



```
layer     filters    size              input                output
    0 conv     16  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16  0.150 BFLOPs
    1 max          2 x 2 / 2   416 x 416 x  16   ->   208 x 208 x  16
    2 conv     32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32  0.399 BFLOPs
    3 max          2 x 2 / 2   208 x 208 x  32   ->   104 x 104 x  32
    4 conv     64  3 x 3 / 1   104 x 104 x  32   ->   104 x 104 x  64  0.399 BFLOPs
    5 max          2 x 2 / 2   104 x 104 x  64   ->    52 x  52 x  64
    6 conv    128  3 x 3 / 1    52 x  52 x  64   ->    52 x  52 x 128  0.399 BFLOPs
    7 max          2 x 2 / 2    52 x  52 x 128   ->    26 x  26 x 128
    8 conv    256  3 x 3 / 1    26 x  26 x 128   ->    26 x  26 x 256  0.399 BFLOPs
    9 max          2 x 2 / 2    26 x  26 x 256   ->    13 x  13 x 256
   10 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFLOPs
   11 max          2 x 2 / 1    13 x  13 x 512   ->    13 x  13 x 512
   12 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024  1.595 BFLOPs
   13 conv    256  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 256  0.089 BFLOPs
   14 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFLOPs
   15 conv    255  1 x 1 / 1    13 x  13 x 512   ->    13 x  13 x 255  0.044 BFLOPs
   16 detection
   17 route  13
   18 conv    128  1 x 1 / 1    13 x  13 x 256   ->    13 x  13 x 128  0.011 BFLOPs
   19 upsample            2x    13 x  13 x 128   ->    26 x  26 x 128
   20 route  19 8
   21 conv    256  3 x 3 / 1    26 x  26 x 384   ->    26 x  26 x 256  1.196 BFLOPs
   22 conv    255  1 x 1 / 1    26 x  26 x 256   ->    26 x  26 x 255  0.088 BFLOPs
   23 detection
Loading weights from ./yolov3-tiny.weights... (Version 2) Done!
```

Figure 2: YOLOv3 Tiny Convolutional Layer Configuration and Weights

As discussed previously, the neural network is responsible for receiving camera data and performing object detection on the images. During initial development, a visual diagnostics mode was implemented in the Python wrapper to support visually viewing both the raw source frame data and an annotated image denoting the detected objects. While the annotated image is useless for automated analysis and comparison, it can be very advantageous to visualize the process during development and validate the functionality of the neural network. Figure 3 depicts a sample camera frame and demonstrates the operation of the neural network by comparing it to the annotated image produced from the neural network's output.



Raw Frame Data *Corresponding Annotated Image*

Figure 3: Raw Camera Data, and the Corresponding Annotated Image Depicting Detected Objects

With the test foundation in-place, development on the core Difference Engine could commence.

Core Difference Engine Top Level Architecture

When developing the top-level architecture for the Difference Engine, one of the primary goals was modularity and versatility. The Difference Engine should serve as a high-level, modular framework for analyzing the outputs of neural networks (among other sensors), regardless of the type of data being produced. To achieve this goal, a very modular and flexible class hierarchy was developed. In addition to allowing for extensibility, this design pattern promotes the development of highly cohesive classes with a low degree of coupling. This design pattern increases unit-testability, reduces the risk of side-effects, and helps to reduce development time. A class diagram depicting the top-level architecture of the Difference Engine is shown in Figure 4.

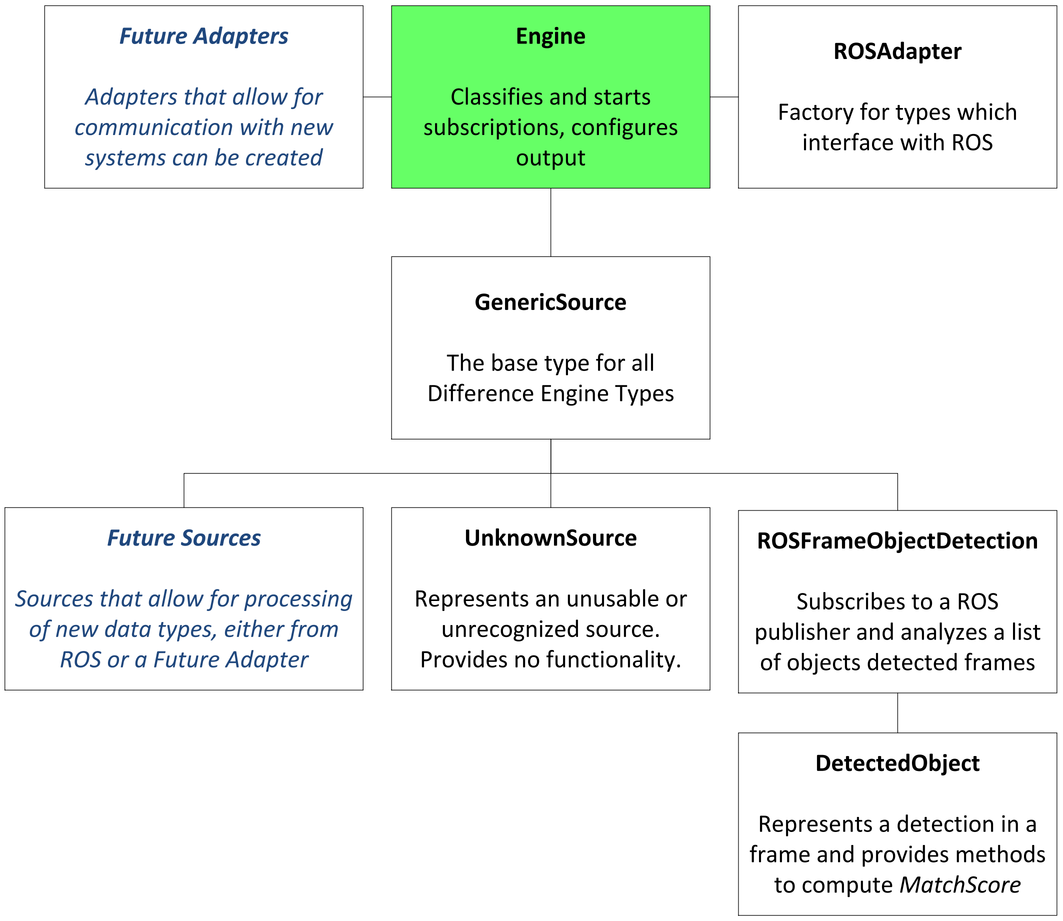
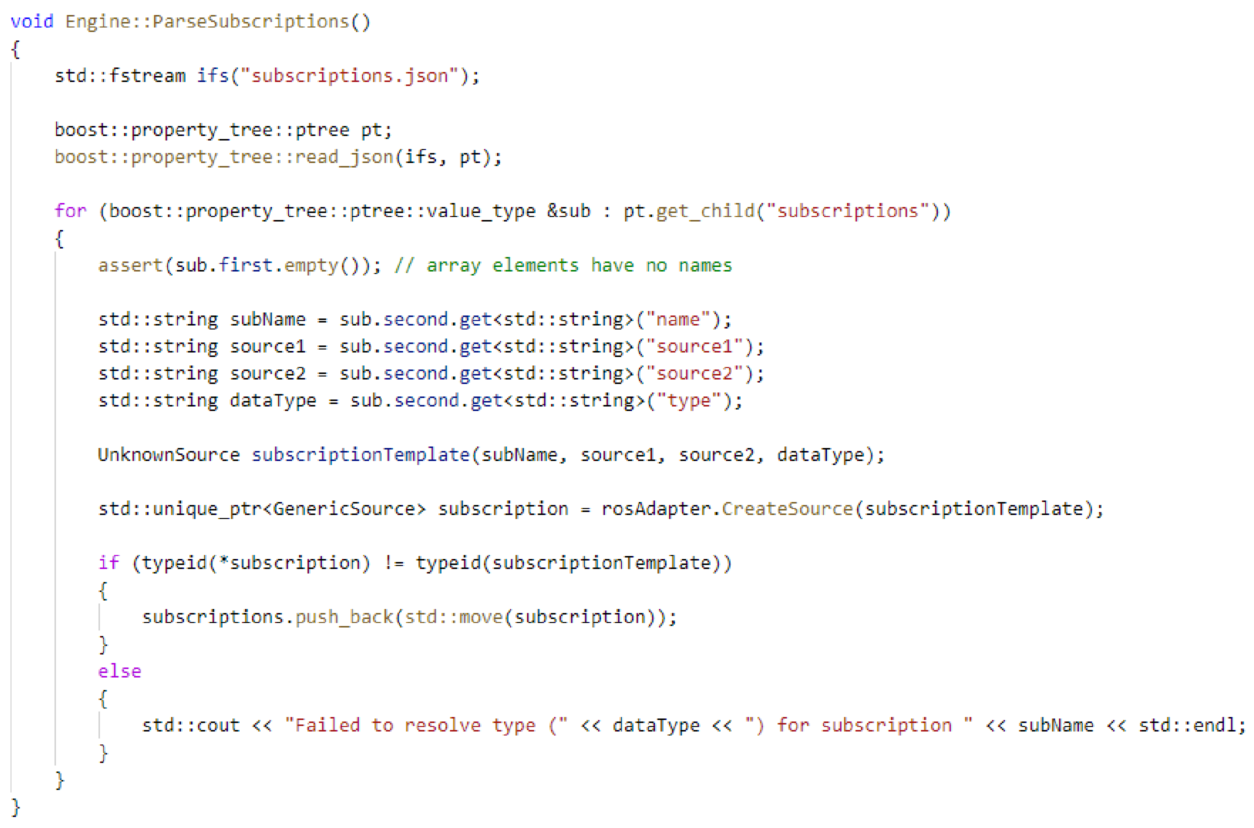

Future Adapters
Adapters that allow for communication with new systems can be created
Engine
Classifies and starts subscriptions, configures output
ROSAdapter
Factory for types which interface with ROS
GenericSource
The base type for all Difference Engine Types
Future Sources
Sources that allow for processing of new data types, either from ROS or a Future Adapter
UnknownSource
Represents an unusable or unrecognized source. Provides no functionality.
ROSFrameObjectDetection
Subscribes to a ROS publisher and analyzes a list of objects detected frames
DetectedObject
Represents a detection in a frame and provides methods to compute MatchScore

Figure 4: Class Hierarchy of the Difference Engine

Notably shown in the class diagram in blue italicized text are two spots for future expansion to support both different Publisher-Subscriber providers and different data types. The Future Adapters section represents the ability to create new communication adapters to interface with systems other than ROS. Likewise, the Future Sources section represents the ability to extend the Difference Engine to support new data types, either from an ROS source, or from a new source added as a “Future Adapter”. This extensibility is a core result of the Dynamic Modular Type System the Difference Engine is based around.

Dynamic Modular Type System

For the test application, the primary output from the neural network were vector objects corresponding to detected objects in raw camera data, however, in automotive applications, numerous other types of data can be produced through machine learning algorithms. Possible control signals range from steering vectors to brake application levels. As a result, a Dynamic, Modular Type System was developed for the Difference Engine to allow for extensibility for future applications. The fundamental input unit to the Engine is a set of subscriptions. Each subscription is a named object that specifies the source of the raw output data that is to be compared, and a comparison operator that should be applied.



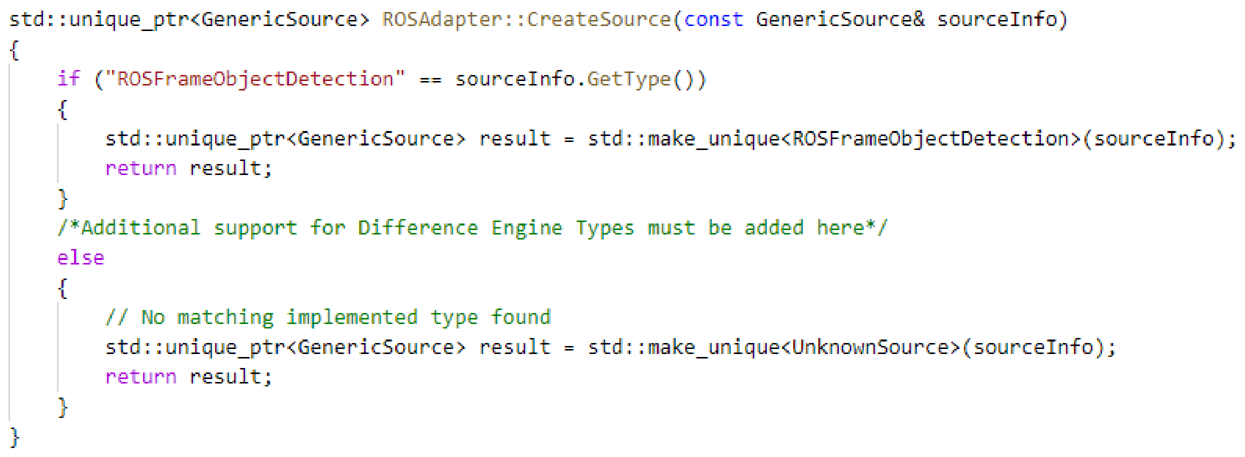
```cpp
void Engine::ParseSubscriptions()
{
    std::fstream ifs("subscriptions.json");

    boost::property_tree::ptree pt;
    boost::property_tree::read_json(ifs, pt);

    for (boost::property_tree::ptree::value_type &sub : pt.get_child("subscriptions"))
    {
        assert(sub.first.empty()); // array elements have no names

        std::string subName = sub.second.get<std::string>("name");
        std::string source1 = sub.second.get<std::string>("source1");
        std::string source2 = sub.second.get<std::string>("source2");
        std::string dataType = sub.second.get<std::string>("type");

        UnknownSource subscriptionTemplate(subName, source1, source2, dataType);

        std::unique_ptr<GenericSource> subscription = rosAdapter.CreateSource(subscriptionTemplate);

        if (typeid(*subscription) != typeid(subscriptionTemplate))
        {
            subscriptions.push_back(std::move(subscription));
        }
        else
        {
            std::cout << "Failed to resolve type (" << dataType << ") for subscription " << subName << std::endl;
        }
    }
}
```

Figure 5: Adding Subscription to Difference Engine (Engine.cpp)

Different comparison operators can be easily added into the Difference Engine source code to accommodate comparisons for new data structures. Figure 5 depicts a snippet of code from Engine.cpp responsible for adding the two Raspberry Pis into the Difference Engine for comparison. As can be seen in Figure 5, the information is provided in JSON format containing the name, two sources, and type of the subscription. The type of subscription used in this project is of ROSFrameObjectDetection, therefore if the type of the subscription matches it will be successfully added as a valid subscription. The information on the Difference Engine types not only allow for variable data structures, but they also allow for different sources to obtain data from. Potential data sources could include neural networks involved in object detection or other applications as long as they produce a parsable output such as JSON or XML. To adapt to evolving networking and interconnection technologies, custom Difference Engine Types can be implemented to support retrieving data from these unique interfaces and communication channels. Today, while Robot Operating System (ROS) is the most popular middleware for robotics applications, other publisher-subscriber platforms also exist. Even if a system primarily uses ROS, designers may choose to provide some sensor data over alternative, non-ROS interfaces. This could be used, for example, to retrieve sensor data over a custom interface like Bluetooth, or over a vendor-proprietary HTTP interface that is incompatible with ROS. In a system where these alternative interfaces are used, the Difference Engine could easily be modified to receive custom data from these custom interfaces and perform difference analysis on it. A new Difference Engine Type could be easily created and added into the source code to encapsulate the complexity of retrieving data from this custom interface. Custom comparison operators can also be defined to match the exact format of the data being retrieved. Corresponding with the Class Hierarchy of the Difference Engine shown in Figure 4, any additional logic needed to communicate with this custom interface could be plugged in as a class in the "Future Adapters" section. Then, any additional types or comparison operations that need to be defined in order to properly analyze the data could be implemented as classes in the "Future Sources" section, while still being able to derive most base functionality from the provided GenericSource implementation. This ability for modular expansion greatly bolsters the appeal of the Difference Engine.

Robot Operating System Adapter

The primary Difference Engine types that we have created and fully implemented are the ROSAdapter and ROSFrameObjectDetection types. ROSAdapter is a generic wrapper for ROS middleware functions that allows the Engine to operate as a ROS Node and participate in Publisher-Subscriber operations. ROSAdapter serves as a factory for all Difference Engine types that consume data through ROS. A factory in object-oriented programming is a function or object which returns objects of a varying prototype or class from some function call [6]. The use of factories rather than constructors or prototypes allows for the use of polymorphism for object creation and provides encapsulation, releasing the code from ties to specific classes or objects [6]. The specific type being used in the demonstration is the ROSFrameObjectDetection type, which handles subscribing to, parsing, and performing difference operations on lists of objects detected within a single image frame. The code for resolving subscriptions into specific Difference Engine Types that can be executed and processed in the Engine is shown in Figure 6.



```cpp
std::unique_ptr<GenericSource> ROSAdapter::CreateSource(const GenericSource& sourceInfo)
{
    if ("ROSFrameObjectDetection" == sourceInfo.GetType())
    {
        std::unique_ptr<GenericSource> result = std::make_unique<ROSFrameObjectDetection>(sourceInfo);
        return result;
    }
    /*Additional support for Difference Engine Types must be added here*/
    else
    {
        // No matching implemented type found
        std::unique_ptr<GenericSource> result = std::make_unique<UnknownSource>(sourceInfo);
        return result;
    }
}
```

Figure 6: Creating ROSFrameObjectDetection type from Subscription (ROSAdapter.cpp)

In order to process incoming data, two unique steps are required. First, frame data must be correlated between the two neural networks. While seemingly obvious, it is critical to ensure that detected object data being analyzed resulted from the same frame. Since both neural networks are processing input data in parallel, output data should also be processed in parallel, as matched pairs of data between the networks. Since the Difference

Engine is designed to handle multiple concurrent steams of data and most underlying hardware platforms contain multi-core processors, the analysis process has been designed to take advantage of threading and parallel processing techniques. While multi-threading has the potential to greatly increase system performance and capability, it adds substantial complexity to the correlation process. Two independent queues of incoming data must be maintained, each one mapped to the output from a neural network. Mutexes must be used to protect access to the queue data structures, both when adding new incoming frame data, and when removing data to perform analysis.

Since the Difference Engine establishes independent Publisher-Subscriber relationships with each of the two neural network (publishers), and each neural network is running on a different physical device, transmission latency causes equal frame data to not always be delivered concurrently. For example, assume the system is powered on at time moment *t = 0* and neural network results are published immediately from both subscribers. Due to the overhead of the ROS system, along with physical network latency, the Difference Engine receives results from the first neural network at *t = 4*, while results from the second network do not arrive until *t = 7*. Only after this point can analysis begin on the data. A timing diagram of this possible scenario is provided in Figure 7.

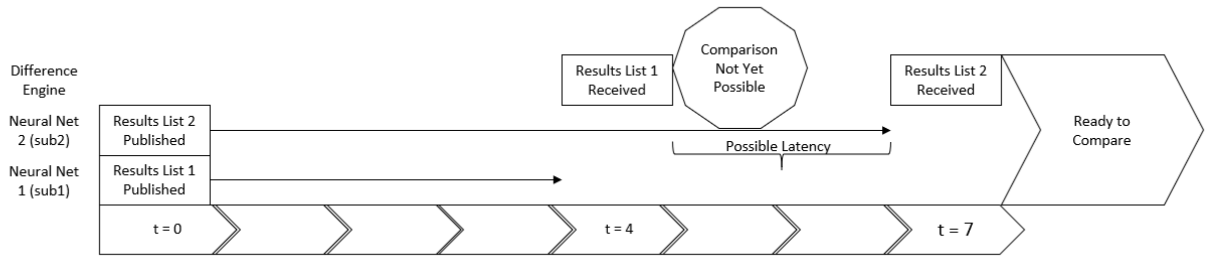


Figure 7: Synchronization Issues Caused by Latency or Out-Of-Sync Neural Networks

When the first result message is received, the communication thread is required to obtain access to the first queue (using the mutex), and then enqueue the results. At this point, the analysis thread will begin execution. However, the thread will quickly stall since a matched pair of results is not yet available. After the second set of results arrives, exclusive access to the second queue is obtained on the communication thread, and the message is stored. The analysis thread is then signaled to resume. Walking through the queue, the thread searches for messages with corresponding origination timestamps. In experimental testing, we found that network congestion can substantially delay the delivery of results from either network, and that in some extreme cases, new incoming result data differed by several seconds between the first and second neural networks. To accommodate for this, the analysis thread has additional control logic to help re-synchronize the data streams to allow for proper analysis. After all correlation and synchronization actions are performed, a pair of results from each neural network, matched in time, will be produced.

Difference Criteria

After a matched pair of data is available in the Engine, the actual comparison operation between the results can begin. For variable length lists of arbitrary data, equality comparisons become a challenging problem. Many neural networks, or their corresponding software implementations, lack output stability, meaning that the ordering of output results is not guaranteed to remain the same, even if the results do. As a result, the ordering of individual objects delivered from the neural network cannot be considered in equality operations. The problem is further complicated when the consideration that output values are expected to differ between different neural network implementations is added to the algorithm. Any number of changes to a neural network (new training data, different number of layers, etc.) can result in different output. While the output from the network under test is expected to be similar to the output from the reference network, results are expected to differ slightly. It also must be considered that a serious defect in either neural network could cause an object to be completely not detected, or an extraneous object to be incorrectly detected. An explanation of the algorithm we have developed to perform these comparison operations is detailed next.

A "detected object" is defined as an n-tuple composed of (x_position, y_position, object_width, object_height, object_guessed_name, probability). Each neural network produces an arbitrary-length list of detected objects, referred to here as *results1* and *results2*, respectively. To determine the probability that two detected objects are the same, a MatchScore function has been established to compare them. The MatchScore function produces a probability ranging from 0 to 1 that two detected objects are identical. In brief, the *MatchScore* value is an averaged composite score of the percentage difference between each component of the n-tuple for each detected object. More formally expressed for two detected objects, a and b,

$$MatchScore(a,b) = \frac{x_{score} + y_{score} + width_{score} + height_{score} + name_{score} + prob_{score}}{6}$$

$$where$$

$$x_{score} = \left(1 - \frac{MAX\left(\left|a.x_{position} - b.x_{position}\right|, T\right)}{T}\right)$$

$$y_{score} = \left(1 - \frac{MAX\left(\left|a.y_{position} - b.y_{position}\right|, T\right)}{T}\right)$$

$$width_{score} = \left(1 - \frac{MAX(|a.objectwidth - b.objectwidth|, T)}{T}\right)$$

$$height_{score} = \left(1 - \frac{MAX(|a.objectheight - b.objectheight|, T)}{T}\right)$$

$$name_{score} = \begin{cases} 1, if\ a.object\_guessed\_name\ = b.object\_guessed\_name \\ 0, otherwise \end{cases}$$

$$prob_{score} = \begin{cases} 1 - \frac{|a.probability - b.probability|}{100} \\ 0, otherwise \end{cases}$$

where MAX is a standard function to select the largest of two values and

T is a threshold value that represents the largest allowable difference in that parameter for it to be weighted in the computation of MatchScore

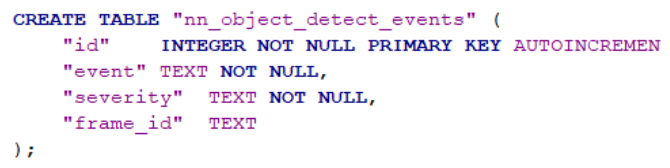
With the comparison metric established, we exhaustively compare the *results1* and *results2* datasets to establish match relationship pairs between every object. Since every object from one set is being compared to every object in the other set, this comparison has a time complexity of $O(n^2)$. This forms a set of results, *r*, that consists of pairs of detected objects, the detected object they are compared against, and the resulting *MatchScore*. From this, every element in *result1* can be looked up in *r*, and the pair with the highest valued *MatchScore* selected. When this is performed for the entire set of *r*, the match list, *m,* is created.

Before the match list can be effectively utilized, a matching threshold must first be established. Let *MATCH_THRESH* represent the minimum *MatchScore* that is required to assert that two detected objects are correlated.

- From *m*, any entries with a *MatchScore* higher than *MATCH_THRESH,* but below 1.0 can be reported as a minor difference between the outputs.
- From *m*, any entries with a *MatchScore* of 1.0 can be reported as being identical between the results sets
- From *m, any entries with a MatchScore below MATCH_THRESH* can be reported as being unique and unmatched.
- Any detected object that appears in *results2*, but not in *m* can be reported as being unique and unmatched.

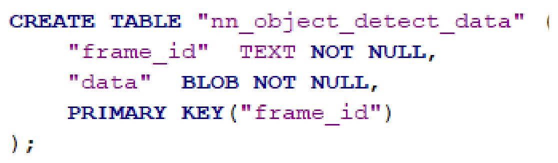
Output Format

With the algorithm for determining differences in the dataset established, and a functional implementation of it coded, the search for the best format for the Difference Engine to use to report findings commenced. Ultimately, an SQLite database was selected as the most versatile output container. SQLite is a lightweight, portable, relational database system that can be used to store structured data in a format that can easily be queried and manipulated [7]. In addition to reporting a listing of discrepancies found between the data streams, storing the raw frame data from the neural network that was responsible for causing the discrepancy is highly valuable to engineers who might attempt to troubleshoot the problem later. Since raw image data is expensive in terms of storage, avoiding duplication of data is a critical design parameter. To achieve this, the database consists of two unique tables for each subscription, a detected events table and a raw frame data table.



```
CREATE TABLE "nn_object_detect_events" (
    "id"    INTEGER NOT NULL PRIMARY KEY AUTOINCREMEN
    "event" TEXT NOT NULL,
    "severity"  TEXT NOT NULL,
    "frame_id"  TEXT
);
```

Detected Events Table Schema

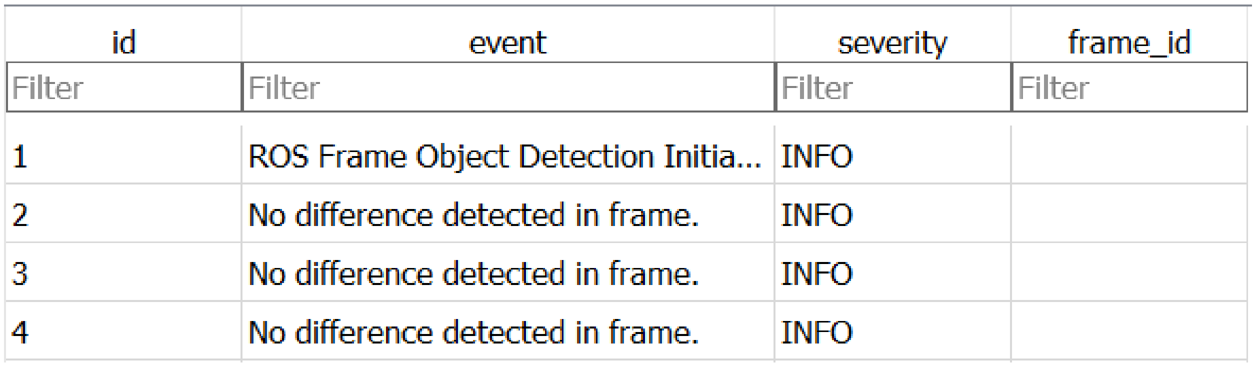


```
CREATE TABLE "nn_object_detect_data" (
    "frame_id"  TEXT NOT NULL,
    "data"  BLOB NOT NULL,
    PRIMARY KEY("frame_id")
);
```

Raw Frame Data Table Schema

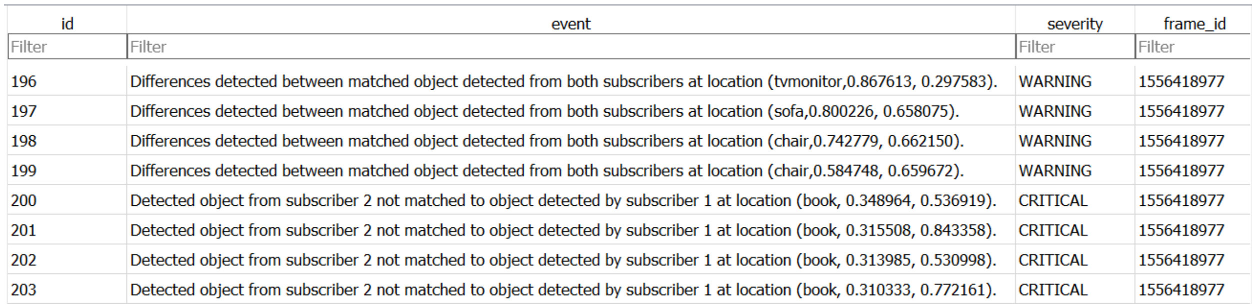
Figure 8: SQL Table Schemas

The detected events table is responsible for logging each detected difference, which consists of a message detailing the discrepancy, a severity level, and the frame number that caused the difference. Each event is also assigned a unique, monotonically increasing identifier. The first time an event is raised from an image frame, an entry is stored in the raw frame data table containing the frame number and the raw frame data. This allows for cross-table lookups to find the corresponding frame data for any difference reported in the detected events table, without inducing any duplication of frame data when multiple differences arise from a single frame. Sample output produced during testing demonstrating capabilities of the database format can be seen in the following figures.



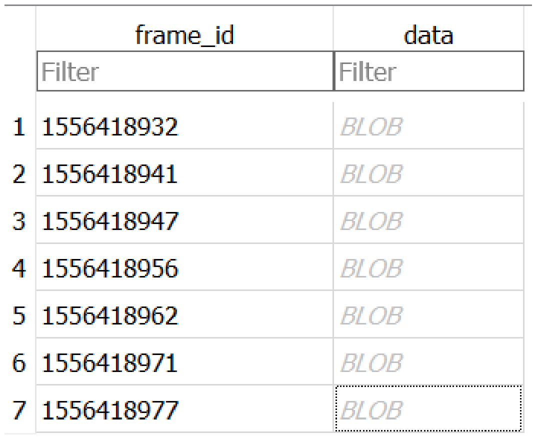
id	event	severity	frame_id
Filter	Filter	Filter	Filter
1	ROS Frame Object Detection Initia...	INFO	
2	No difference detected in frame.	INFO	
3	No difference detected in frame.	INFO	
4	No difference detected in frame.	INFO	

Figure 9: Information Messages in the Events Table indicating that no differences were detected. Since this is an information message, no frame data was stored, leading to frame_id being blank



id	event	severity	frame_id
Filter	Filter	Filter	Filter
196	Differences detected between matched object detected from both subscribers at location (tvmonitor,0.867613, 0.297583).	WARNING	1556418977
197	Differences detected between matched object detected from both subscribers at location (sofa,0.800226, 0.658075).	WARNING	1556418977
198	Differences detected between matched object detected from both subscribers at location (chair,0.742779, 0.662150).	WARNING	1556418977
199	Differences detected between matched object detected from both subscribers at location (chair,0.584748, 0.659672).	WARNING	1556418977
200	Detected object from subscriber 2 not matched to object detected by subscriber 1 at location (book, 0.348964, 0.536919).	CRITICAL	1556418977
201	Detected object from subscriber 2 not matched to object detected by subscriber 1 at location (book, 0.315508, 0.843358).	CRITICAL	1556418977
202	Detected object from subscriber 2 not matched to object detected by subscriber 1 at location (book, 0.313985, 0.530998).	CRITICAL	1556418977
203	Detected object from subscriber 2 not matched to object detected by subscriber 1 at location (book, 0.310333, 0.772161).	CRITICAL	1556418977

Figure 10: Event entries showing both minor differences ('WARNING', top), and total detection failure ('CRITICAL', bottom) occurring within a single frame. Since a deviation occurred, frame data was also stored in the database, and can be indexed by the frame_id value.



	frame_id	data
	Filter	Filter
1	1556418932	BLOB
2	1556418941	BLOB
3	1556418947	BLOB
4	1556418956	BLOB
5	1556418962	BLOB
6	1556418971	BLOB
7	1556418977	BLOB

Figure 11: Entries in the Data Table. 'BLOB' indicates raw binary frame data is stored.

The event reporting capabilities of the Difference Engine are demonstrated in Figures 9 and 10. In the detected events table shown in Figure 9, an informational message indicating that the system is running can be seen in the first entry. Additionally, three entries are logged revealing that no differences were detected in frames. Since no discrepancies were reported, the frame data was not stored (in the interest of reducing file size) and the corresponding frame_id field is blank. Sample data from the detected events table is continued in Figure 10, where 6 deviations are reported, 3 WARNING level and 3 CRITICAL level. The mechanism to prevent duplication of raw frame data can be observed in action in Figures 10 and 11. Since events 196-203 in Figure 10 all originated from the same image (frame number 1556418977), only a single copy of the frame is stored, as seen in Entry 7 in the table in Figure 11. Preventing duplication of frame data in the database both reduces the size of the output file and increases system performance.

Using a standardized data format like an SQLite database perfectly allows for follow-on work or tools to be developed for our Difference Engine system. As discussed in the sample usage case of an autonomous automobile, software engineers would likely need to review the finding the Difference Engine produces between a current-generation production neural network, and one they are evaluating prior to deploying. In a safety critical system, it is difficult and potentially dangerous to rely solely on an algorithmic decision of the criticality of a difference. In this case, a human operator would likely need to sift through the reported differences above a certain threshold to determine whether they are actually significant. To achieve this, an intuitive graphical interface could easily be constructed to consume data from this report. Since the format is standardized, no custom parsing would be needed, and the query capabilities of SQL allow for powerful, yet concise, targeted retrieval of data.

Problems Encountered and Lessons Learned

Problems

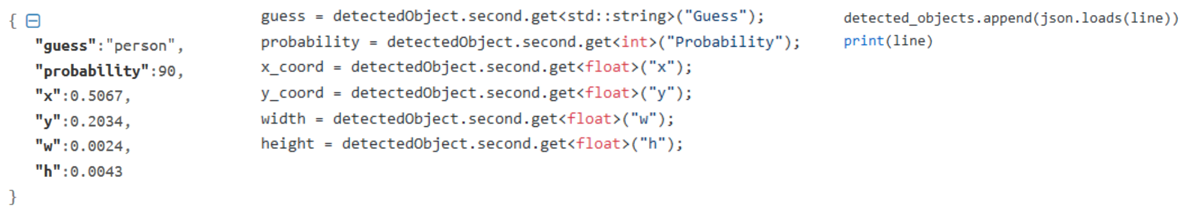
The project overall was a success and provided both a functional product whose basis can be employed for real-world applications and as a learning experience in both system architecture and development. As with any project, there were some impeding factors encountered throughout the process. A chief difficulty encountered was finding image-recognition neural networks that would run on ROS. Neural networks are complex programs that are often run on both CPUs and GPUs to reduce the time cost of their analysis. A neural network which possessed both the desired image-recognition capabilities and ability to run on a relatively low-power Raspberry Pi were few in availability. Additionally, the ROS middleware used in this project runs on Python 2.7 exclusively, with support for Python 3.x planned for a future release. TensorFlow recently made a version of their neural network available for Raspberry Pi's but it was restricted to Python 3.x versions only. A singular neural network which met the demands and constraints for the project was found with Darknet and YOLO. Documentation on the reference implementation of YOLO with Darknet was unfortunately extremely lacking, and the Python wrapper that invoke the C functions for the neural network was very inadequately written. After modifications were made to the wrapper, a functional neural network implementation was achieved. Although this neural network worked as required, the problem of being able to compare outputs from two separate neural networks arose. Ideally, two different sets of training data would be available for the reference network, and the "network under test" to evaluate how the outputs differ. However, only a single pre-built model (YOLOv3Tiny) was available, and raw training data used to compile this model was unavailable. This problem was resolved by adding an artificial test mode that performs a Gaussian blur on the camera feed into one of the subscriber nodes using a function included with OpenCV. Gaussian blurring of image input into the Pi can be enabled using a command line argument. A slight alteration of the image fed into the neural network affected the neural network enough to where its outputs no longer exactly matched those of the other concurrently-running neural network, enabling testing of the detection analysis portion of the Difference Engine.

Lessons Learned

During the development of the Difference Engine, several valuable lessons regarding effective modern software engineering techniques were learned.

Following with the conventions of major ROS components, the custom ROS "catkin make" system was utilized in conjunction with CMake for compiling the codebase. CMake is a modern C++ build system that seeks to provide greater extensibility and portability than can be achieved with a traditional Makefile system. CMake was utilized to coordinate the compilation of the various source files for the Difference Engine, as well as the linking of external components, such as ROS Resources, Boost, and SQLite. While the initial learning curve for CMake was steep in contrast to traditional systems, like GNU Make, the result was worthwhile. At the conclusion of the development, the Difference Engine components can quickly and effectively be compiled due to the CMake build system.

In order to augment the capabilities of the C++ Language and standard library, the Difference Engine utilizes the free and open source Boost Framework [8]. Boost is a set of libraries that vastly extend the capabilities of the standard libraries, especially in the areas of effective memory management, asynchronous operations, operating system abstractions, and other high-level operations. Boost strives to remain highly compatible and interoperable with the C++ Standard Libraries, but nonetheless was an unfamiliar technology. In the Difference Engine, Boost's parsing libraries were used to greatly simplify the process of loading and parsing configuration files, as well as decoding and parsing incoming detected object data that was serialized in the Javascript Object Notation (JSON), a typical representation format for structured data. While the Boost implementation was not as simple to use as the JSON parsing functionality in Python, it was still nearly trivial to use. In Figure 12, comparisons of our C++/Boost and Python implementations of JSON parsing are given, along with the sample input JSON data they operate on.



```
{
   "guess":"person",
   "probability":90,
   "x":0.5067,
   "y":0.2034,
   "w":0.0024,
   "h":0.0043
}
```

```
guess = detectedObject.second.get<std::string>("Guess");
probability = detectedObject.second.get<int>("Probability");
x_coord = detectedObject.second.get<float>("x");
y_coord = detectedObject.second.get<float>("y");
width = detectedObject.second.get<float>("w");
height = detectedObject.second.get<float>("h");
```

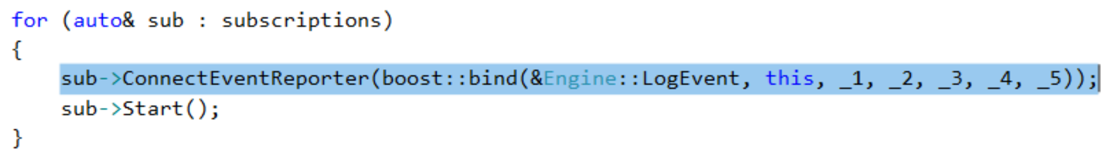
```
detected_objects.append(json.loads(line))
print(line)
```

Input JSON Data *Boost JSON Parsing Implementation from DetectedObject.cpp* *Python JSON Parsing Implementation from rpi_ros_video.py*

Figure 12: Comparison of JSON Parsing Implementations in Boost and Python

Given the nature of the C++ Language (strongly typed, type-safe), Boost made the JSON parsing process as simple as possible, which was a huge productivity advantage.

Boost's Signals2 library exposes a robust signal-slot event system, which was utilized to form a simple, yet powerful interface for Difference Engine Types to report findings back to the Engine, where they can be logged to the results database [9]. This system can be observed in Figure 13, depicting the connection of a ReportEvent signal from a Difference Engine Type with the logging slot available in the core engine.



```
for (auto& sub : subscriptions)
{
    sub->ConnectEventReporter(boost::bind(&Engine::LogEvent, this, _1, _2, _3, _4, _5));
    sub->Start();
}
```

Figure 13: Connection of ReportEvent signal with LogEvent slot via Boost (Engine.cpp)

Without the Boost signal-slot system, a similar system could be configured using lambda functions available in the C++ 11 Standard, but substantially more code would be required, which would be detrimental to both code quality and development time. Overall, the inclusion of Boost served to greatly enhance the Difference Engine project in functionality, ease-of-development, and resulting code quality.

Future Work and Extensibility

Extensibility was a core focus of the design. Software engineering is a diverse field with innumerable applications and an even greater variety in which a particular problem can be solved. As such, there is no true universal solution which can be implemented. With this at the forefront, care was taken in constructing both the inputs and output of the Difference Engine so that its function is not restricted to specific formats. The implementation and benefits of Difference Engine types were explored when discussing the dynamic modular type system and those benefits were the chief factor in implementing the Difference Engine in this manner among other candidate implementations.

In addition to supporting variant input types, the Difference Engine in application would constitute the first phase of a testing process. Once the neural networks have been processed by the Difference Engine, the differences detected must be analyzed through some other means to determine their nature and potential impact on the system as a whole in which they may reside. SQLite was chosen as the data format to hold and organize the data produced by the Difference Engine. Employing a standard data format allows for secondary analysis on this data and simple interfacing with the entire dataset through use of existing commands for data query and manipulation. For example, a simple GUI application could be written that selects only differences where the severity level is critical and retrieves the respective data from each subscriber and the frame on which they differed. If the data was stored in a text log file, for example, a nearly trivial data retrieval operation could become significantly more complex. Using a standardized data format like SQLite allows a user to simply retrieve data, and not have to manage parsing a custom storage format first. Another scenario could exist where a neural network may be tested against the existing one in multiple environments. Cross-checking data from not only one test but multiple simultaneous tests could quickly become a daunting task and choosing a less versatile output format could render the Difference Engine useless in larger operations.

References

[1] X Zhai, ‘darknet-nnpack', 2018. [Online]. Available: https://github.com/zxzhaixiang/darknet-nnpack/

[2] “TensorFlow Guide | TensorFlow Core | TensorFlow,” *TensorFlow*. [Online]. Available: https://www.tensorflow.org/guide/.

[3] J Redmon, ‘Darknet: Open Source Neural Networks in C', 2013. [Online]. Available: https://pjreddie.com/darknet/

[4] J Redmon, ‘YOLOv3: An Incremental Improvement', 2018. [Online]. Available: https://pjreddie.com/darknet/yolo/

[5] M Dukhan, ‘NNPACK Acceleration package for neural networks on multi-core CPUs', 2017. [Online]. Available: https://github.com/shizukachan/NNPACK

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston: Addison-Wesley, 1994.

[7] ‘SQLite: Home Page', 2018. [Online]. Available: https://www.sqlite.org/index.html

[8] ‘Boost C++ Libraries', 2018. [Online]. Available: https://www.boost.org/

[9] D. Gregor, ‘Chapter 36. Boost.Signals2', 2007. [Online]. Available: https://www.boost.org/doc/libs/1_66_0/doc/html/signals2.html

Appendix A – Developer Information

Appendix A consists of directions to assist in obtaining, compiling, and running the Difference Engine and supporting test neural networks. Two sections are provided, one detailing the instructions for configuring the Difference Engine Master Node, and one detailing how to configure each Sub Node with an instance of the neural network.

Configuring the Master Node with the Difference Engine Application

Step 0 - Configure the Raspberry Pi and Download Sources

The Master Raspberry Pi should be configured with Ubuntu and ROS Kinetic. The prebuilt images available from Ubiquity Robotics at https://downloads.ubiquityrobotics.com/pi.html can be used to simplify the configuration. Follow their directions to download and install the prebuilt images.

The hostname of the Raspberry Pi should be updated appropriately. The naming convention used is cpe499-rpX, starting at 1 for the Master RPi. As such, in our setup, cpe499-rp1 is used for the master node.

Change to the catkin source code directory in the workspace

```
cd ~/catkin_ws/src
```

Checkout a copy of this repository into that folder.

Step 1 - Build from Source

Ensure pre-requisites are installed. Boost should be included with the ROS distribution. SQLite3 should be manually installed.

```
sudo apt-get install sqlite3
```

Change to the catkin source code directory in the workspace and run the Catkin CMake tool. The Catkin build system will traverse the workspace and automatically build the Difference Engine and all dependencies.

```
cd ~/catkin_ws/src

catkin_make -j1
```

Step 2 - Publish Camera Data to the Sub Raspberry Pis

The master node is responsible for publishing camera data to the neural networks running on the sub Pis. The raspicam_node tool that is included with ROS can be used to publish the camera data. Before running Raspicam_node though, the specific configuration file that specifies the target frame size and rate must be installed. The configuration file is provided in this respository and must be copied into the system path.

```
sudo cp ~/catkin_ws/src/difference_engine/camerav2_nn_config.launch
   /opt/ros/kinetic/share/raspicam_node/launch/
```

Once the configuration file has been installed, the camera publisher can be started. The configuration file only has to be installed one time.

```
roslaunch raspicam_node camerav2_nn_config.launch &
```

Step 3 - Run the Difference Engine

By default, catkin stores the built executable in the ~/catkin/devel folder. If a custom build directory has been specified, this step must be updated accordingly.

Change to the Difference Engine build directory

```
cd ~/catkin_ws/devel/lib/difference_engine
```

If this is the first time running the Difference Engine, a configuration file must be created in the working directory. A sample is provided in this repository. Copy it in and modify it to match the target configuration. Correct paths for the neural networks must be provided in order to obtain data.

```
cp ~/catkin_ws/src/difference_engine/src/subscriptions.json ./

vim subscriptions.json
```

Finally, run the Difference Engine

```
./difference_engine
```

Finding will be reported to the difference_engine.db file in the same directory, and debug information will be printed to the console.

Configuring each Sub Node with the Object Detection Neural Network

Step 0: Configuring the Raspberry Pi

The Raspberry Pi should be configured with Ubuntu and ROS. The prebuilt images available from Ubiquity Robotics at https://downloads.ubiquityrobotics.com/pi.html can be used to simplify the configuration.

The hostname of the Raspberry Pi should be updated appropriately. The naming convention used is cpe499-rpX, starting at 1 for the Master RPi. In our setup, cpe499-rp2 and cpe499-rp3 are used for the two neural network implementations.

Once the hostname has been updated, the sub (slave) Pi needs to be joined in the ROS Network. The following file should be appended to the `~/.bashrc` file.

```
export ROS_MASTER_URI=http://cpe499-rp1.local:11311
```

Clone this repository into `~/catkin_ws/src/`

Step 1: Install NNPACK

Installing Ninja

Install PeachPy and confu

```
sudo pip install --upgrade git+https://github.com/Maratyszcza/PeachPy

sudo pip install --upgrade git+https://github.com/Maratyszcza/confu
```

Install Ninja

```
cd ~/catkin_ws/src/difference_engine_nn/ninja

./configure.py --bootstrap

export NINJA_PATH=$PWD
```

Installing NNPACK

```
cd ~/catkin_ws/src/difference_engine_nn/NNPACK

confu setup

python ./configure.py --backend auto
```

Build with

```
$NINJA_PATH/ninja
```

Do a `ls` and you should be able to find the folders `lib` and `include` if all went well.

Test if NNPACK is working:

```
bin/convolution-inference-smoketest
```

From the Blogspot page referenced in the source repository, users have reported that "In my case, the test actually failed in the first time. But I just ran the test again and all items are passed. So, if your test failed, don't panic, try one more time.". Errors here can likely be ignored, or retested until success occurs.

Copy the libraries and header files to the system environment:

```
sudo cp -a lib/* /usr/lib/

sudo cp include/nnpack.h /usr/include/

sudo cp deps/pthreadpool/include/pthreadpool.h /usr/include/
```

Step 2. Install darknet-nnpack

We can now build our customized version of darknet and the corresponding Python wrapper that integrates ROS functionality

```
cd ~/catkin_ws/src/difference_engine_nn/darknet-nnpack/

make -j2
```

Step 3. Build ROS Dependencies and Custom Messages

```
cd ~/catkin_ws

catkin_make
```

Step 4. Run It

Before running the Neural Network, make sure camera data is being published from the master.

```
cd ~/catkin_ws/src/difference_engine_nn/

./rpi_ros_video.py
```