Machine Learning Approaches for Integration of Sensors on Legacy Aircraft

by Zachary Dennis

B.S. Electrical Engineering, May 2013, Oklahoma Christian University
M.S. Engineering Management, May 2015, The University of Texas at Austin

A Dissertation submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

August 31, 2023

Dissertation directed by

Thomas Mazzuchi
Professor of Engineering Management and Systems Engineering & of Decision Sciences

Shahram Sarkani
Professor of Engineering Management and Systems Engineering

The School of Engineering and Applied Science of The George Washington University certifies that Zachary Scott Dennis has passed the Final Examination for the degree of Doctor of Philosophy as of June 27, 2023. This is the final and approved form of the dissertation.

Machine Learning Approaches for Integration of Sensors on Legacy Aircraft

Zachary Dennis

Dissertation Research Committee:

Shahram Sarkani, Professor of Engineering Management and Systems Engineering, Dissertation Co-Director

Thomas Mazzuchi, Professor of Engineering Management and Systems Engineering & Decision Sciences, Dissertation Co-Director

Amir Etemadi, Associate Professor of Engineering and Applied Science, Committee Member

Joseph P. Blackford, Professorial Lecturer of Engineering Management and Systems Engineering, Committee Member

Hadi Mohammadi, Professorial Lecturer of Engineering Management and Systems Engineering, Committee Member

© Copyright 2023 by Zachary Dennis
All Rights Reserved

Dedication

This Dissertation is dedicated to my wife Lavon who is ceaseless in her loyalty and service to God, our family, and our friends. May the world gain an ounce of her compassion and ethics.

Acknowledgements

I would first like to thank my wife Lavon for supporting me by pushing me to spend the requisite amount of time to pursue a Ph.D. degree, through regular late nights on coursework and research. I could not have completed this program without her constant commitment.

Thanks to my parents and in-laws for encouraging me and helping take care of my children as I prepared the research for this dissertation.

Thanks to my three boys Lawson (4), Graham (3), and Ian (1) for always being willing to tell me about their days, show me their art, and roughhouse in my needed stress breaks.

Thank you to “Nana” Nell Dennis who valued higher education and inspired me to pursue my Ph.D. Although she passed during my program, I know she was proud. She is loved and missed.

Dr. Thomas Holzer was instrumental in helping me scope this dissertation. His frank feedback, advice, and guidance during challenges helped me to grow as an engineer and researcher.

Abstract of Dissertation

Machine Learning Approaches for Integration of Sensors on Legacy Aircraft

Rapid integration of advanced sensors onto legacy military aircraft is critical for maintaining technological advantage in warfighting domains. Integration of these sensors is accomplished through upgrade programs that often fail during integration due to defect discovery and interoperability issues. Existing Department of Defense initiatives related to open architectures have improved sensor integration but have not eliminated the need for custom interface software to account for behavioral disparities across different sensors. The subject research proposes that reinforcement machine learning algorithms can be applied to aircraft sensor interfaces during integration and verifies effectiveness by training and testing Greedy, Q-Learning, Deep Q-Learning, Double Deep Q-Learning, and Instance-Based Learning algorithms against modeled GPS, Optical, LIDAR, and Infrared sensor functions, both as deterministic and stochastic processes. The results are useful to open architecture standards management groups, sensor vendors, and systems and software engineers who are developing strategies and designs to accelerate subsystem integration timelines by reducing failures discovered during integration.

Table of Contents

Dedication ..... **iv**

Acknowledgements ..... **v**

Abstract of Dissertation ..... **vi**

Table of Contents ..... **vii**

List of Figures ..... **ix**

List of Tables ..... **xi**

Chapter 1 – Introduction 1

1.1 Overview 1
1.2 Background 5
1.3 Problem Statement 13
1.4 Research Objectives 15
1.5 Research Questions and Hypotheses 16
1.6 Thesis Statement 18
1.7 Organization of Dissertation 18

Chapter 2 – Literature Review 20

2.1 Overview 20
2.2 Integration and Interfaces 20
2.3 Prior Studies on Integration Growth in Acquisition Programs 22
2.4 Architectures, Standards, and Specifications 23
2.5 Data Interoperability 24
2.6 Machine Learning 27
2.7 Application Programming Interface (API) Sequence Learning Techniques 28

Chapter 3 – Research Methodology 29

3.1 Overview 29
3.2 Top Factors that Impact Sensor Interoperability 32
3.3 Reinforcement Learning 37
3.3.1 Overview 37
3.3.2 Agents 49
3.3.3 Environments 55

3.3.4 Stochastic Variations 62
3.4 Reinforcement Learning Methodology Considerations 67
3.4.1 Agent Training 68
3.4.2 Agent Testing 69
3.5 Software Development Tools and Methods 70
3.5.1 Cloud Execution 70
3.5.2 Artifacts and Version Control 70
3.5.3 Software Packages 72
Chapter 4 – Research Results 73
4.1 Top Factors Driving Integration 73
4.1.1 Integration Factors 73
4.1.2 Z-Score per Defect Cause Code 75
4.2 Deterministic Environments 76
4.2.1 Training Rewards 76
4.2.2 Test Accuracy 80
4.2.3 Discussion 82
4.3 Stochastic Environments 87
4.3.1 Training Rewards 87
4.3.2 Test Accuracy 91
4.3.3 Discussion 93
4.4 Research Hypotheses Results 95
4.5 Research Questions Results 97
Chapter 5 – Conclusions and Recommendations 99
5.1 Conclusion 99
5.2 Summary of Research Objectives 101
5.3 Research Limitations and Future Work 103
References 109

List of Figures

Figure 1: Monolithic Architecture Example 6

Figure 2: Modular Architecture Example 7

Figure 3: Example Data Transformations 8

Figure 4: Range of Design Information 10

Figure 5: Standards Across Processing Stack 11

Figure 6: Data Point Reduction 35

Figure 7: Integration Defect Cause Codes 36

Figure 8: Reinforcement Learning Interactions 39

Figure 9: ESM Markov Decision Process Example 40

Figure 10: Mapping Protocols and Procedures to the OSI/RM 42

Figure 11: Psuedo-code for Training and Test Execution 44

Figure 12: Execution Engine Training Process 45

Figure 13: Logical Architecture 46

Figure 14: Subsystems and Interactions 47

Figure 15: Initialization Use Case 48

Figure 16: Training Use Case 49

Figure 17: Neural Network Structure 52

Figure 18: Input State Representation 54

Figure 19: Sensor Model Complexity 56

Figure 20: Markov Decision Process for Dungeon Environment 57

Figure 21: Integrated GPS Environment 59

Figure 22: Stochastic Dungeon Markov Decision Process 63

Figure 23: Stochastic GPS Markov Decision Process 64
Figure 24: Stochastic Infrared Markov Decision Process..... 65
Figure 25: Stochastic LIDAR Markov Decision Process 66
Figure 26: Stochastic Optical Markov Decision Process..... 67
Figure 27: Training in Dungeon Environment..... 77
Figure 28: Training in GPS Environment..... 78
Figure 29: 32,768-Step Training in Optical Environment..... 78
Figure 30: 262,144-Step Training in Optical Environment..... 79
Figure 31: Training in Infrared Environment 79
Figure 32: Training in LIDAR Environment..... 80
Figure 33: Training in Stochastic Dungeon Environment..... 87
Figure 34: Training in Stochastic GPS Environment 88
Figure 35: 32,768-Step Training in Stochastic Optical Environment..... 88
Figure 36: 262,144-Step Training in Stochastic Optical Environment..... 89
Figure 37: Training in Stochastic Infrared Environment..... 89
Figure 38: Training in Stochastic LIDAR Environment..... 90

List of Tables

Table 1: Key DoDAF Models for System Interoperability [39] 27

Table 2: Variable Breakout by Hypothesis 30

Table 3: Tests of Hypotheses 31

Table 4: Hypothesis Testability 32

Table 5: Z-Scores per Defect Cause Code 76

Table 6: Dungeon Environment Accuracy Results 81

Table 7: GPS Environment Accuracy Results 81

Table 8: Optical Environment Accuracy Results 81

Table 9: Infrared Environment Accuracy Results 82

Table 10: LIDAR Environment Accuracy Results 82

Table 11: Stochastic Dungeon Environment Accuracy Results 91

Table 12: Stochastic GPS Environment Accuracy Results 92

Table 13: Stochastic Optical Environment Accuracy Results 92

Table 14: Stochastic Infrared Environment Accuracy Results 92

Table 15: Stochastic LIDAR Environment Accuracy Results 93

Table 16: Research Hypotheses Results 95

Table 17: Research Questions Results 97

Chapter 1 – Introduction

1.1 Overview

Integration of advanced sensors onto legacy military aircraft is critical to maintaining an effective air force. Sensors give aircrews awareness of information that is detectable across the electromagnetic spectrum. Aircraft sensors can tell aircrews where they are located, where enemies are located, when they have been detected by enemy radar, and when missiles are tracking them. Having advanced sensors that are effective against modern threats helps keep aircrew safe, increases the types of missions that aircraft can execute, and improves mission outcomes [1].

Unfortunately, keeping aircraft sensor capabilities updated is a challenge. Aircraft needed for missions are often very old. The most widely used Air Force bomber, the B-52, became operational over sixty-five years ago [2]. Because legacy aircraft can be outdated with long service lives, they deal with obsolescence and effectiveness challenges as adversary capabilities improve. Sensors need to be updated regularly to keep the legacy aircraft operationally relevant. The 2018 U.S. National Defense Strategy recognized that platform electronics and software need to be "designed for routine replacement instead of static configurations that last more than a decade [3]." The 2022 U.S. National Defense Strategy included sensors as an enabler for detection of nuclear attacks, the sharing of sensor data as critical for regional defense, and acquisition programs for sensors as the priority over Command and Control and interceptor programs [4]. One of the Air Force's Seven Operational Imperatives is "Achieving Moving Target Engagement at Scale in a Challenging Operational Environment," and Air Force Secretary Frank Kendall stated, "for the scenarios of interest it all starts with these

sensors [5].” Unfortunately, aerospace acquisition programs can be slow and expensive, often due to components not meeting higher level system functions. For example, the James Webb Telescope, a new space telescope with state-of-the art sensors and deployment systems, experienced substantial problems integrating telescope components, and the program was delayed by seven years with 95% cost growth to $9.7 billion [6]. Dealing with integration risks and interoperability is key for keeping aerospace systems relevant and affordable. Recent research using data from the B-52 bomber, A-10 fighter, F-117 stealth fighter, and OH-58B helicopter determined that interoperability is the second highest factor on real system age and determining weapon system viability [2]. Even though slow traditional programs could mitigate integration risk with large budgets and plenty of schedule margin, future sensor integration must become faster and more affordable to meet operational needs.

Despite rapid sensor integration being a priority for the aerospace industry, current Department of Defense open standards do not enable modern sensor subsystem interoperability with legacy subsystems, restricting mission capabilities. Better Buying Power 3.0 directed use of Modular Open Systems Approaches (MOSA) for integration interfaces [7]. MOSA in Department of Defense programs further directed the use of widely supported and consensus-based standards for major system interfaces [8]. Examples of these standards are Future Airborne Capability Environment (FACE) and Open Missions Systems (OMS) [9, 10]. These architectures enable isolated replacement of components, enable portability of components across different types of computing environments, and have non-proprietary interfaces that are used directly by components communicating within the architectures [11]. Integration of sensors into these

architectures is easier than in other aircraft architectures because the data structures that make up the interfaces are defined by open standards. Integrators can connect new sensors into avionics systems and have greater confidence that the data syntax will align, which can also be described as “plug-and-talk.”

OMS’s goal is to “develop industry consensus for a non-proprietary mission system architectural standard that enables affordable technical refresh and insertion, simplified mission systems integration, service reuse and interoperability, and competition across the life-cycle [10].” OMS defines how the requirements for mission package to be considered OMS-compliant. These requirements include which operating systems are allowed, which services and subsystems are needed for different tiers of compliance, and what messages are required for the different types of services and subsystems. OMS also requires services and subsystems to use a Critical Abstraction Layer, which has a defined interface used for sending and receiving messages over a logical bus.

FACE defines an approach that “allows software-based ‘capabilities’ to be developed as software components that are exposed to other software components through defined interfaces. It also provides for the reuse of software across different hardware configurations. Ultimately, the FACE key objectives are to reduce development costs, integration costs, and time-to-field for avionics capabilities [9].” To accomplish this, FACE defines a reference architecture with five segments including the Operating System Segment (OSS), Input/Output Services Segment (IOSS), Platform-Specific Services Segment (PSSS), Transport Services Segment (TSS), and Portable Components Segment (PCS). FACE describes how software should be broken up into segments and

how software in those segments communicate with each to achieve portability and reuse [9].

While newer MOSA-aligned standards like FACE and OMS make integration easier, sensors from different vendors can behave differently and still be compliant to those standards. OMS specifies that functional behavior is defined by vendor-specific Service Contracts. The Sensor Open Sensor Architecture (SOSA) Consortium is working to address behavior standardization by defining conceptual, logical, and physical data models of data needs of sensors and their relationships, as well a set of functions that are needed for sensors [12]. However, when replacing a sensor, even following modern MOSA-aligned standards, legacy sensor controller software that implements the interfaces to the sensors must often be reengineered to accommodate changes in sensor behavior. Differences in behavior and data semantics drive legacy sensor controller software changes that prolong sensor integration and take away from a "plug-and-play" paradigm.

To address the lack of interoperability of legacy sensors on aircraft, the subject research proposes that reinforcement learning can be applied to learn interfaces, reducing integration errors and presents a set of reinforcement learning algorithms with their effectiveness against modeled aircraft sensor interfaces. The contributions of the research include the concept of applying reinforcement learning to sensor integration on aircraft, results to show feasibility of the approach for both deterministic and stochastic sensor behaviors, and a recommendation that reinforcement learning be more broadly applied for systems integration.

1.2 Background

Open architectures are designed to improve subsystem integration, and while many new platforms since 2017 are designed for compliance, there remains substantial complexity when these open architectures are retrofitted onto legacy platforms.

Open architecture approaches use nonproprietary hardware and software interfaces with modular components, where elements operate independently and are loosely coupled with other elements in the system. Environment specifications standardize use of Application Programming Interfaces (APIs), operating systems, and their configurations. Open architecture approaches also take advantage of adaptive acquisition strategies where cost/schedule are fixed, and scope is allowed to be refined to meet user priorities.

To meet these open architecture elements, standards such as FACE and OMS have been implemented in new systems and are evolving and maturing in these systems today. New standards such as SOSA are continuing to improve how to specify open architectures. These standards are letting small businesses and U.S. Government organic employees compete for upgrades and sustainment where traditional acquisition would have utilized the prime contractors. They are allowing supplemental and upgraded capabilities to be inserted into aircraft baselines with reduced impacts on other system components. Standardized environment specification enables reuse of services and subsystems across aircraft that would traditionally have different environments and not be compatible. Modularity is also enabling upgrades to be incrementally delivered into aircraft baselines, to fit funding limitations and deploy capabilities earlier. However,

industry still has large acquisition programs fail in integration even though they were supposed to be compliant with at least one open architecture standard.

There are several reasons why integration of sensors is challenging on legacy aircraft. First, legacy architectures can be complicated to touch due to their monolithic nature.

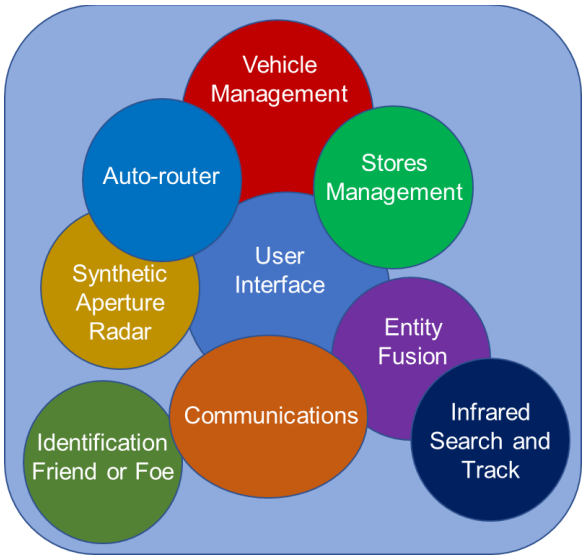

Figure 1: Monolithic Architecture Example

Legacy aircraft frequently do not have separation of functionality so many different subsystems can overlap. Figure 1 shows an example of a monolithic architecture where subsystems such as Communications overlap with Entity Fusion, User Interface, and Identification of Friend or Foe. These overlapping subsystems can share code, data, and access to resources, and each may depend on functions executing in another subsystem for correction operation. Legacy aircraft with these monolithic architectures have some advantages when it comes to computing resource utilization. Infrastructure that supports a subsystem can be shared with other subsystems and the memory and processor utilization can be optimized. This is ideal for old, low-capacity computing systems. However, a downside of monolithic architectures is that making changes to one

subsystem likely impacts other subsystems because they are tightly coupled. Subsystems can even become safety or flight critical only because they are coupled with other subsystems that are responsible for safety or flight critical functions, which drives additional testing. Monolithic architectures can also have proprietary, vendor-specific interfaces between subsystems, which results in locking customers into original equipment manufacturers with reduced competition and higher costs. With tightly coupled, monolithic architectures, upgrades and replacements can become large, expensive programs with extensive regression testing requirements.

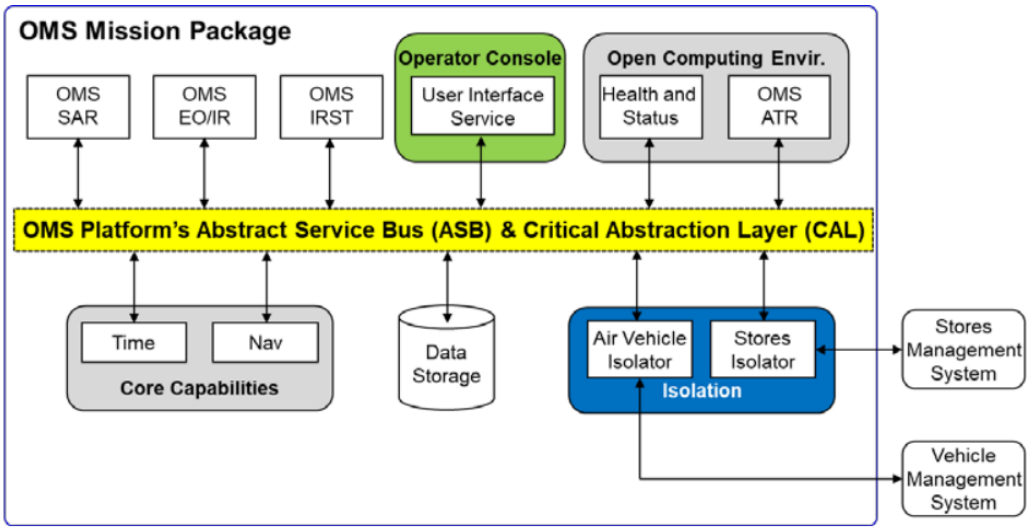

Figure 2: Modular Architecture Example *[13]*

Modular architectures address many of the problems inherent in monolithic architectures. Figure 2 shows an example of an OMS Mission Package, where each of the services are cleanly separated and communicate through the Critical Abstraction Layer, which has a non-proprietary interface. This mitigates vendor lock, and upgrades and replacements become more isolated. For example, it becomes easier to upgrade a Synthetic Aperture Radar (SAR) by replacing a SAR module, and there may not be a need for a costly regression test of the vehicle management requirements.

There is also a significant amount of complexity associated with adding advanced sensors on legacy aircraft that are not designed for interfacing with new architectures. Figure 3 shows potential disconnects between legacy system expected interfaces on the right and advanced sensor interfaces on the left.

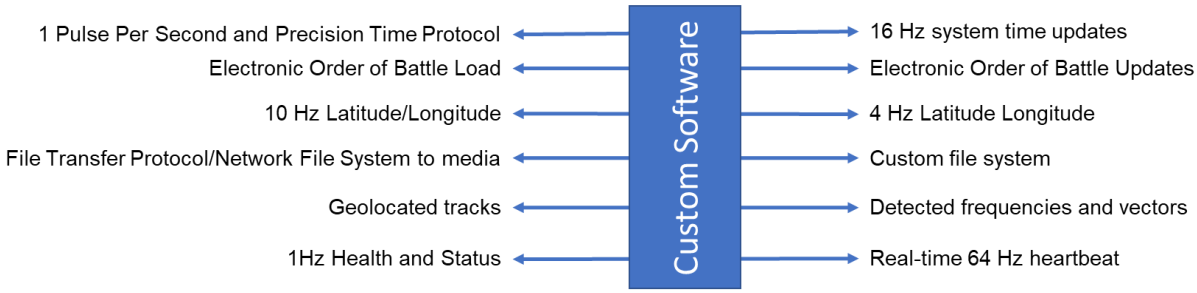

Figure 3: Example Data Transformations

Time could be specified differently, with advanced sensors expecting Precision Time Protocol (PTP) and one pulse per second (PPS) where legacy aircraft may only be able to provide a 16 Hertz (Hz) time update. Custom software may have to be written to manage time to satisfy advanced sensors, and complexity would increase if operators can manually set aircraft system time. Electronic Orders of Battle (EOBs) are how known radio-frequency emitters are communicated from intel groups and mission planners to onboard aircraft sensors. Old aircraft with more limited bandwidth may expect to send updates to EOBs, where advanced sensors may expect to receive full new EOB files from the aircraft. To integrate with advanced sensors, old aircraft may now need to keep track of and provide a full EOB to the sensor when needed. Different rates of aircraft position updates between what legacy aircraft are tracking and what advanced sensors are expecting may result in adding custom software with a flight model that interpolates position to provide to the sensor.

Legacy aircraft and advanced sensors may have different mechanisms for file transfer services. Legacy aircraft are likely to have custom or outdated file systems, where advanced sensors may expect to utilize network file transfer protocols. Legacy aircraft may expect to get raw frequencies and vectors to radio frequency emitters where advanced sensors may expect to send geolocated and identified threats. Advanced sensors may expect to respond to 1 Hz heartbeat messages for health and status when a legacy aircraft may expect a more rigid 64 Hz health update. Each of these differences drive custom software to provide the intermediate data transformations and systems required to keep both the legacy aircraft and the advanced sensor operational.

If these differences are not expected or known during project planning, there may be significant scope growth. Sensor behavior is not standardized, even for open architecture sensors, and integration programs can overlook and oversimply these interfaces because they are integrating open architecture sensors. Acquisition programs that fail to sufficiently specify these interfaces experience unplanned cost and schedule growth.

The third reason why integration of sensors onto legacy aircraft is challenging is that design and interface documentation for legacy aircraft systems can be insufficient. Figure 4 shows the range of design information that may be available on aircraft and their subsystems, depending on the best practices and processes in place at the time of their creation.

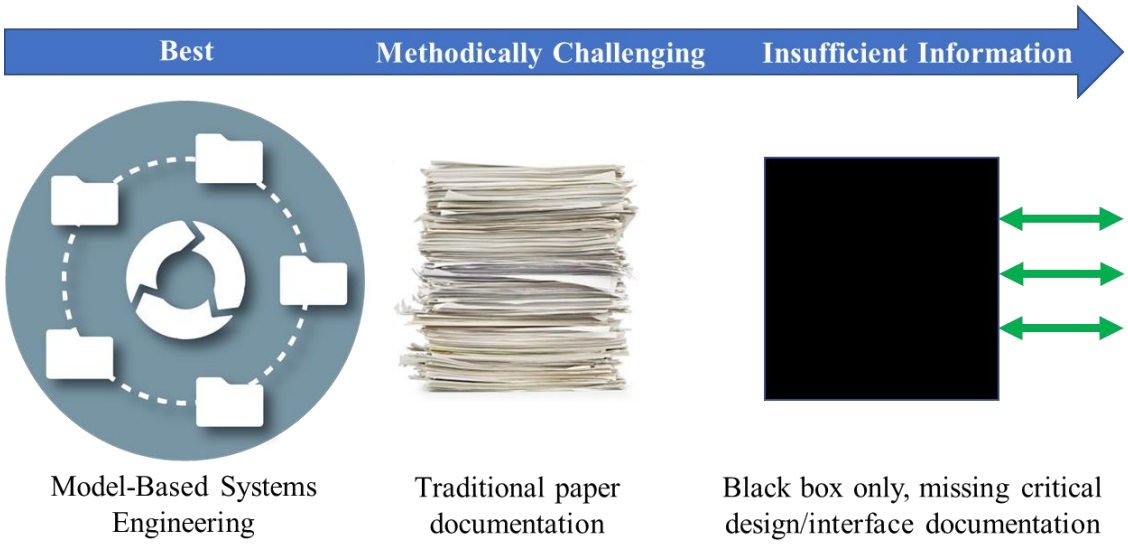

Figure 4: Range of Design Information

New systems are being developed that have full Department of Defense Architecture Framework (DoDAF)-based digital twins that can be tested end-to-end before parts are fabricated, but older systems can be very different. Some aircraft only have paper design documentation that is kept in safes due to classification. With traditional paper documentation, often not even digitized, it can be challenging to locate appropriate interface and design information required for adding new sensors onto a legacy platform. Documentation on systems and components may not be available at all due to proprietary restrictions or being lost over time. When sensors are integrated with systems where the existing design is often outdated or even a black box, figuring out how to correlate legacy functions with new sensors can be challenging. An optimistic outcome in these cases could be that there is reliable interface control documentation that can be replicated in a digital model.

To address the three issues of monolithic architectures, data transformation complexity, and interface data availability, a substantial number of standards and

specifications are in place across multiple levels of a processing stack. Figure 5 shows architectural levels from Application Behavior down to Hardware Interfaces. These levels are inspired by the Open Systems Interconnection networking reference model, but they were modified to show how standards build on each other and address specific elements of computing environments.

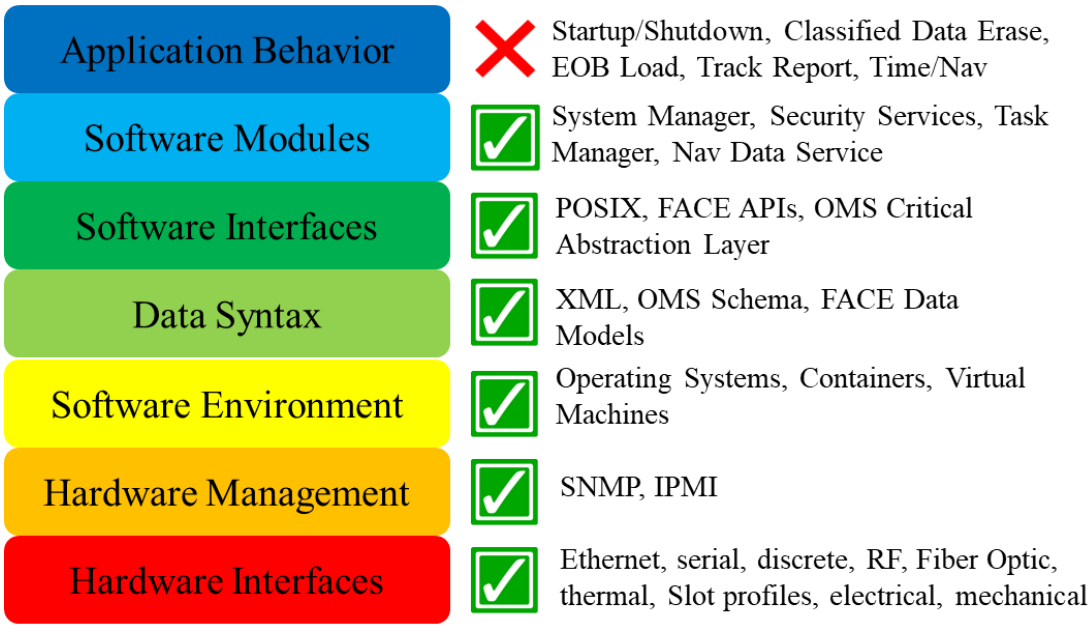

Figure 5: Standards Across Processing Stack

SOSA specifies many elements of hardware architecture, including a range from thermal interfaces to chassis slot profiles, along with connector shapes and sizes for sensor hardware. Management of network hardware is specified with protocols such as Simple Network Management Protocol (SNMP) and Intelligent Platform Management Interface (IPMI). These protocols are used for network devices as well as server hardware. Open architecture standards specify what software environments should contain, including specific operating systems, programming languages, virtual machines, and software containers. Another layer is data syntax, which defines how data is

organized. Data syntax defines the structure of the data, including bytes and fields within messages. For example, a navigation message could have latitude fields and longitude fields. This data syntax is often described using XML and these are documented as part of OMS schemas and FACE data models. Software interfaces are another important level for open architectures. Software interfaces include APIs that your software applications use to access underlying messages and computer resources. Portable Operating System Interface (POSIX) is a standard that makes application software less dependent on operating systems, FACE provides APIs for sending data from one architecture segment to another, and OMS provides APIs for sending and receiving data coming across on the Critical Abstraction Layer.

SOSA goes a layer higher and specifies the types of software modules that need to be provided by an application. For example, a SOSA compliant sensor would need a system manager, security services, task manager, and a nav data service.

Although some MOSA standards like SOSA have specified what the sensor functions should be, no MOSA standard has specified how sensor functions should be performed. For example, there are messages for startup and shutdown, which are used in a function that should be handled by a system manager, but there has only been limited specification on the sensor behavior. Two sensor vendors can be compliant to MOSA standards, have the same messages and use the same functions, but not be interoperable due to the sequencing of the messaging and the way each vendor interprets the fields in the messages, also known as data semantics. There are standard sets of functions that sensors need to perform, and there are messages that are required to execute those

functions, but until the way the sensors should behave is specified, sensors will continue to be plug and talk and will not achieve plug and play.

1.3 Problem Statement

Originally, the research was intended to be focused more broadly than integration of advanced sensors onto legacy aircraft. Refining the problem statement was one of the most time-consuming aspects of the dissertation research. Originally, the problem was scoped to include the challenges of integration of legacy systems into Systems of Systems (SoS) while dealing with a misalignment of data formats and interoperability with emerging systems. This would have been timely and meaningful as it directly relates to the Secretary of the Air Force's operational imperative regarding the B-21 Family of Systems. However, modeling for this problem would have been complex and likely to require access to more classified data than was feasible for the duration of the research. Potential applications to SoS are covered in conclusion.

Another iteration of the problem included general integration of subsystems into systems. While integration of subsystems into systems is an important aspect of the Systems Engineering process, scoping the research to subsystem integration was too broad to be testable. Even integration of sensors into systems was too broad of a topic due to the diverse sets of systems that sensors are integrated into. The research was scoped to advanced sensor integration on legacy military aircraft, but there are applications in problem domains beyond sensor integration on aircraft, which are covered in the conclusion section.

Integration of subsystems onto legacy aircraft specifically was also too broad of a topic as there is a large range of subsystems that could be integrated onto aircraft

including communications, electronic warfare, weapons, and many others. Having test cases that test a wide enough range of these subsystems was not feasible for the research timeline. Testing with other types of aircraft subsystems and even testing for other types of systems such as ground vehicles is potential for future research.

Integration of advanced sensors on legacy military aircraft was the problem area chosen because the problem is specific, important, and timely. The U.S. Department of Defense (DoD) and Defense Industrial Base are developing Joint All-Domain Command and Control (JADC2), which will connect systems across DoD portfolio [14]. Integrating advanced sensors onto legacy aircraft is important for JADC2 because legacy aircraft are fielded today with operational sensors and could provide important battlefield information that feeds into future common operating pictures. If aircraft can integrate sensors and share sensor data, then they can work together to accomplish missions, which is important for future collaborative operations.

Given the importance of sensor integration and the limitations of open architectures that are becoming prevalent in the DOD, the research is intended to explore the top factors impacting the lack of sensor interoperability, as well as ways to improve sensor integration and interoperability.

Once advanced sensor integration on legacy aircraft was selected as the problem area, clearly defining the problem was a focus. The problem was not that integration is complex or that large acquisition programs fail. The problem was not that integration is expensive or that complex custom software is required for data transformations. The final problem statement was scoped to:

Problem Statement: Existing Department of Defense open standards do not enable modern sensor subsystem interoperability among legacy military aircraft with legacy subsystems, which limits sensor tech refresh and upgrades, restricting mission capabilities.

The problem statement is clear and concise without jargon or acronyms. It is specific in that all terms have standard definitions. It is a single issue focused on addressing sensor interoperability through finding and addressing gaps in standardization, and it does not suggest a solution, either by changing standards or finding other technologies

1.4 Research Objectives

There are several research objectives that flow down from the overall problem statement. The first objective is:

O1: Decrease sensor integration timelines on legacy aircraft.

If the research can be used to reduce cost and schedule and field capabilities faster, then Air Force pilots will be better equipped and more likely to survive future combat missions.

The second research objective that flows down from the problem statement is:

O2: Remove the interoperability limitation of proprietary behavioral specifications.

Even open architecture-compliant advanced sensors have proprietary behavior, which is a detriment to supportability and interoperability. If this limitation is removed, then vendor lock will be limited, opening the door for competition for sensor replacements.

The final research objective is:

O3: Provide algorithm recommendations for fault-tolerant, robust sensor controllers.

Sensor controllers on aircraft need to be able to handle a range of sensor models and types, and they need to work in degraded, wartime conditions. If sensors became interoperable on legacy aircraft but could not withstand dynamic environments, then they would not be effective in the field.

These three research objectives break down the problem statement into key outcomes, and each would provide a meaningful benefit to the military aircraft and sensor industry if accomplished.

1.5 Research Questions and Hypotheses

These three research objectives flow down into three key research questions. The first research question is:

Q1: What are the top factors that impact sensor interoperability?

This research question supports O1 because sensor interoperability would enable faster integration timelines on aircraft.

The second research question is:

Q2: What are the best algorithms to apply to machine learning of sensor behavior?

This research question supports both O1 and O2 because if behavior can be learned by machine learning algorithms, then sensor integration timelines will decrease. If vendor specific behavioral interfaces could be learned by algorithms, then vendor lock would also be mitigated.

The final research question is:

Q3: What are the top methods for accounting for dynamic aircraft and sensor environments for machine learning-based sensor controllers?

This research question supports O1 and O2 because if behavior can be learned and algorithm implementations are robust to dynamic environments, then controllers would support interoperability as well as fault tolerance of sensor controllers.

Each of these research questions has a corresponding hypothesis. The first hypothesis corresponds to Q1 and is as follows:

H1: Faulty requirements, faulty programming logic, and interface issues are the top factors that impact sensor integration.

The second hypothesis corresponds to Q2 and is:

H2: Rewarding correct subsystem interfaces through reinforcement learning algorithms results in the correct subsystem behavior on 100% of the test cases.

Sensors are used in safety and flight critical functions on aircraft and performance less than 100% could result in loss of life.

The final hypothesis corresponds to Q3 and is as follows:

H3. Training sensor controllers using reinforcement learning algorithms in stochastic environments results in correct controller actions to dynamic environmental conditions on 100% of the test cases.

The last hypothesis contrasts with H2 and focuses on stochastic environments, where actions may result in a range of outcomes depending on the environmental conditions and probability distributions.

1.6 Thesis Statement

The dissertation will show that existing integration strategies and open architectures are insufficient for sensor interoperability on legacy aircraft. It will show that learning behavior is a missing element of sensor integration and that machine learning algorithms can be applied to solve the issue in ways that achieve sensor operations with 100% accuracy and handle dynamic environments.

Thesis Statement: Reinforcement learning algorithms for sensor behavior are required for sensor interoperability on legacy aircraft.

The dissertation research will result in reinforcement learning algorithm implementations that can be used to learn advanced sensor behavior. These algorithm implementations can be used by companies to accelerate technology integration.

Research Deliverable: Reinforcement learning algorithms that augment open standards to enable sensor interoperability.

1.7 Organization of Dissertation

Chapter 1 – Introduction lays the foundation for issues surrounding sensor integration on legacy aircraft and outlines the problem statement, research objectives, research questions, hypothesis, and thesis statement. Chapter 2 – Literature Review will document key elements of the literature review that informed the research. Chapter 3 – Research Methodology will describe the methods used for evaluating the hypotheses as well as discuss challenges and considerations regarding the experimentation. Chapter 4 – Research Results will describe the test results from the experiments and provide an analysis of the findings. Chapter 5 – Conclusions and Recommendations will summarize

the dissertation and completion of the objectives, document the limitations, and detail the future work important to advancing machine learning-based subsystem integration.

Chapter 2 – Literature Review

2.1 Overview

To provide a baseline for the dissertation research, a literature review was performed across a variety of topics including subsystem integration, interoperability, MOSA, and machine learning applied to interface specification. Several researchers have addressed integration interface issues, and others have explored reinforcement learning algorithms more generally than interfaces to improve the algorithm performance against complex problem sets. However, no research was found using machine learning algorithms directly to integrate subsystems.

2.2 Integration and Interfaces

Howard Eisner made two recommendations related to integration in *Thinking: A Guide to Systems Engineering Problem Solving*. First was that design is required to facilitate integration. He described integration as a key challenge in systems engineering that is addressable by starting with clean sheet design with the goal of facilitating future integration and upgrades. His second applicable recommendation was to only integrate "stovepipes" when it is cost effective to do so. He criticized government guidelines to "integrate all stovepipes" and discussed both the good intentions and dangers of the direction. He attributed the cause of the government program integration failures to the systems not being originally designed for integration, driving cost and schedule when people try to integrate these legacy systems together [15]. Government guidelines on integrating systems following the JADC2 initiatives meet the criteria for Eisner's concerns. Eisner's concerns on legacy systems without clean sheet designs, not designed for integration and upgrades were supportive of the dissertation's problem definition.

Several researchers have focused on simplifying integration with recommendations regarding interfaces. Sauser et. al defined an integration readiness level (IRL) with maturation criteria to help analyze system interoperability within defense acquisition programs, citing technology maturity as the principal driver to program performance risks. These levels are similar to technology readiness levels (TRLs) in that they move from one to nine, from types of semantic interoperability, through syntactic interoperability, and finally to pragmatic interoperability [16]. The behavioral integration gap discussed previously is best described by IRLs four and five, where data exchanges are already assumed to be in place and where one system can control another. This dissertation research tests integration between sensors and sensor controllers at IRL five.

Goldschmid and Corns built on the IRL research and grouped interfaces into clusters or communities using genetic algorithms. The communities would then receive assessments of their technical maturity before integration phases start. This approach is meant to reduce integration risks by making sure interfaces are complete prior to integration [17].

Morgan, Holzer, and Eveleigh addressed rapid component replacement by requiring interface definition as part of a container modeling method (CMM) [18]. Designing system elements as containers with all their dependencies would be considered a design-for-integration approach, where all elements including interfaces are known when it is time to replace a system due to obsolescence. The paper described a strong way to document configuration items for clean sheet designs.

Grumbach and Thomas defined the Principle of Reuse and Evolvability that allowed for existing components to adapt to new applications with a small increase in

system complexity and a large decrease in uncertainty [19]. This is accomplished due to the complexity of integrating a component being realized during the initial integration and subsequent integrations of the reused component having the benefit of higher maturity. Grumbach and Thomas further validated their results and found that success in component reuse is affected by adherence to hierarchic verification approaches and treating subsystem integration as part of full system integration [20].

2.3 Prior Studies on Integration Growth in Acquisition Programs

Several sources were evaluated to determine key drivers for integration. The first was a Weapon System Integration Cost Analysis from Defense Systems Information Analysis Center [21]. The summary from the report is that detailed weapon and platform integration costs are difficult to quantify for several reasons. First is that programs are at different phases which may leverage work from other programs or previous phases, and the second is that programs do not uniformly report data. One insight of the report was that relative complexity to other programs in a cost model can be used to estimate integration cost.

A second source was a UK report "Weapons Integration Cost Modelling Presentation to SCAF [22]." The goal of this study was to identify and quantify non-recurring weapon integration costs and timeline drivers. The authors used expert judgement to assess program complexity and allocate relative percentages to integration cost drivers. The model effectively predicted weapon integration costs, but it was specific to weapon elements that are not present in sensor integration (such as weapon separation, weapon carriage trials, weapon system evaluation).

The last main source for research on integration drivers was the Government Accountability Office (GAO). A report released in 2015 identified that 62% of ACAT II and III have GAO-identified data reliability issues and that 13 out of 15 programs reviewed exceeded costs or schedule targets. GAO cited drivers of scope growth from key programs, with the top being requirements changing during the program [23].

Each of these integration studies highlighted the lack of data needed to identify top drivers for integration cost and schedule.

2.4 Architectures, Standards, and Specifications

There are many papers and reports in the field of open architecture systems and specifications that informed the dissertation. Norton provided an overview on how interoperability is enabled by open systems architectures (OSA) and what design principles are applicable for open systems [24]. Shepherd gave an overview and comparison of OSAs including Future Airborne Capability Environment (FACE), Hardware Open Systems Technologies (HOST), Open Radio Architecture, Sensor Open Systems Architecture (SOSA), Open Mission Systems (OMS), and Joint Common Architecture (JCA) [25]. One contribution was a summary of how each of the standards cover certain architectural layers. This was used as a starting reference for the mapping of specifications to architecture elements in Figure 5: Standards Across Processing Stack. Zimmerman et al. also provided a strong MOSA overview and recommended sufficient standardization at interfaces as part of the MOSA system design. The paper said that standards implemented at interfaces can contain options that when selected lead to interoperability problems [26].

Research into SOSA was also conducted because of its focus on sensors, and there was a presentation by Sargent that was useful in understanding SOSA modules and what levels are being specified. According to Sargent, SOSA's objectives were to standardize key interfaces only and specify what boxes do but not how they do it [27]. Dubois also tackled limitations of open architectures using FACE and JCA as examples and recommended thorough semantic data modeling to support interoperability as well as creation of mediation services to facilitate adaptation of legacy systems to open architectures [28].

The Universal Plug and Play (UPnP) specification is a commercial specification with similar goals as MOSA standards. UPnP is used to control many devices, including Internet of Things (IoT) sensors. UPnP allows devices to advertise their sensor capabilities which can then be used by applications. This standard enables behavioral interoperability. However, UPnP has substantial vulnerabilities that are addressed by supplementary protocols such as Secure UPnP (SUPnP) [29]. No occurrences were found where UPnP protocols were used on military aircraft, but the protocol itself suggests mechanisms such as dynamic advertisement, discovery, and access of services for creating sensor interoperability, along with the risks of implementing the strategy.

2.5 Data Interoperability

One of the mechanisms for achieving interoperability is with data modeling and building interoperability into the design of a system. This requires use of domain ontologies and semantic modeling on top of traditional interfaces. Today, these data models are specific to systems and are not standardized in MOSA architectures. Dubois

recognized a lack of expertise on data modeling and data science that may contribute to the scarcity of these tools in systems development settings [28].

Domain ontologies are important for relating data from different entities for interoperability. Domain ontologies are ways that data scientists can relate concepts within a particular subject area. Defining how data elements relate back to their foundational ontologies can help generate data transformations needed for enabling interoperability [30, 31]. Yang, Cormican, and Yu modeled a domain ontology using unstructured text documents and natural language processing, providing a strategy for constructing a domain ontology using machine learning techniques. They used linguistic and statistical approaches to detect key terms and created semantic maps to conceptualize the Systems Engineering Body of Knowledge [32]. Similarly, Maree and Belkhatir provided a framework for ontology mapping, using machine learning techniques to analyze ontologies. They used multiple knowledge bases to correct inconsistencies between ontologies and were able to merge ontologies as well as fill in background knowledge [33]. Asfand-E-Yar and Ali also addressed resolving semantic, syntactic, and structural heterogeneity using a methodology for mapping and integrating data that deals with semantic differences. Their paper dealt with the problem of resolving data heterogeneity using structured techniques, experimented with Semantic Web ontology, and provided results [34]. Semantic Web is a meaningful representation of information exchanged on the web in machine-readable form [35].

Using these methods for building ontologies could be useful for designing systems for interoperability by tracing data elements back to higher levels of meaning for what the data elements represent. The methods above included mapping between

heterogeneous ontologies, or ontologies that do not have consistent information with each other. Legacy aircraft data elements that do not align with advanced sensor data elements could be related and transformed using mappings to common ontologies.

The use of data models with ontologies is already being used for IoT applications. Lian et al. showed how semantic technologies addressed heterogeneity between IoT devices by providing machine-readable semantic descriptions and building a semantic model of IoT [36]. Lu et al. proposed a scalable method for IoT semantic interoperability using a dataspaces model that uses distributed semantic information and machine learning algorithm to consolidate different IoT entities [37]. Issues with the number of IoT devices, the variations on their data, and the need for integrating all of them into IoT monitoring and control systems drive the need for data modeling and ontologies that enable interoperability.

Systems Engineering research also recognizes the need for data modeling as part of the systems development process. Li et al. recommended adding Model-Based Systems Engineering (MBSE) and data analysis steps in the V model as a triple V [38]. The triple V model proposed considered data analysis throughout the lifecycle, but the addition of data analysis for each element on a double V was not specific enough. It did not include a focus on integration or taking existing systems into account.

Bellack, Dillon, and Hart met the design-for-interoperability need in a paper published by the Defense Standardization Program Journal. They detailed how system interoperability should be enabled in design with data standardization using DoDAF [39]. They identified key DoDAF models and data integration points that result in standardized data and allow for integration of processes and data exchanges.

Model	Name
DIV-1	Conceptual Data Model
DIV-2	Logical Data Model
DIV-3	Physical Data Model
OV-2	Operational Resource Flow Description
OV-3	Operational Resource Flow Matrix
OV-5b	Operational Activity Model
OV-6c	Business Process Flow
SV-1	Systems Interface Description
SV-4	Systems Functionality Description
SV-6	Systems Resource Flow Matrix
SV-10	Systems Event-Trace Description

Table 1: Key DoDAF Models for System Interoperability [39]

Table 1 lists the key DoDAF models they recommended during design phases to enable interoperability.

In summary, interoperability of sensors on legacy aircraft is dependent on interoperability of data. This can be achieved through static design and specification with domain ontology mapping or DoDAF modeling. Interoperability could also be achieved during system operation. Dynamic UPnP and IoT device integration solutions are potential technologies that are leverageable for sensor integration on aircraft if they can be modified to meet the constraints associated with airborne military platforms, such as cybersecurity vulnerability mitigation.

2.6 Machine Learning

Prior research in machine learning was also explored to provide a foundation in the machine learning techniques used to evaluate Hypothesis 2 and Hypothesis 3. Several researchers have explored reinforcement learning algorithms generally to improve their performance against complex problem sets, although not tied specifically to learning interfaces. Abdolshah et al. proposed model-based reinforcement learning, that could be used to plug and play object-oriented software objects within a software application [40].

The methodology in the paper allowed objects to be inserted into different variations of an environment without needing to retrain the objects. This paper provided a good description of a multi-agent reinforcement learning scenario and could be useful in scaling the reinforcement learning approaches outlined in this dissertation.

Gupta, Roy, and Dutt compared Instance-Based Learning (IBL) and Q-Learning algorithms in a dynamic test environment and found that IBL is more effective for certain dynamic environments [41]. Nguyen and Gonzales similarly tested IBL and Q-Learning algorithms in a maze environment and also found that IBL is more accurate in complex environments [42]. The results from these papers were why IBL was selected as one of the algorithms in the dissertation research. While previous research has addressed integration issues and machine learning separately, research has not been performed using machine learning to integrate subsystems.

2.7 Application Programming Interface (API) Sequence Learning Techniques

One application of machine learning loosely related to sensor interfaces is training software to use APIs. Two sources provided potential paths to accomplish pieces of this. Chen et al. proposed using machine learning to process developed software to recommend sequences of APIs as software development aids [43]. A beam search algorithm was used to generate message sequences from a set of different encoder and decoder models. Also, Gu et al. proposed using a recurrent neural network encoder-decoder to generate API usage sequences based on a natural language query. It also referenced other approaches that treat queries and APIs as bags-of-words for more traditional natural language processing techniques [44].

Chapter 3 – Research Methodology

3.1 Overview

To evaluate the three hypotheses, a set of dependent and independent variables were identified for data collection. These variables were documented in the following table and used to guide experiment design and implementation. The table was also updated to reflect and communicate changes to the experimentation design throughout the development of the dissertation research. The table shows which variables are independent and which are dependent. Hypothesis 1 uses data from a database for a large acquisition program that details the tasks involved with integration of sensors. Hypotheses 2 and 3 use data from evaluating machine learning algorithm implementations against modeled sensor functions.

	Source	Defect Cause Codes	Number of Repair Hours	Number of Steps to Train	Model Accuracy	Cumulative Reward
H1	Classified Program Database	Independent	Dependent			
H2	Greedy Algorithm			Independent	Dependent	Dependent
H2	Q-Learning Model			Independent	Dependent	Dependent
H2	Deep Q-Network			Independent	Dependent	Dependent
H2	Double Deep Q-Network			Independent	Dependent	Dependent
H2	Instance Based Learning Model			Independent	Dependent	Dependent
H3	Greedy Algorithm			Independent	Dependent	Dependent
H3	Q-Learning Model			Independent	Dependent	Dependent
H3	Deep Q-Network			Independent	Dependent	Dependent
H3	Double Deep Q-Network			Independent	Dependent	Dependent
H3	Instance Based Learning Model			Independent	Dependent	Dependent

Table 2: Variable Breakout by Hypothesis

Table 3 shows the analysis techniques applied to the data collected for the different hypotheses. Univariate analysis was used for each of them. The statistical approach applied to H1 will be detailed in Section 3.2: Top Factors that Impact Sensor Interoperability.

Hypothesis	Data Needed	Data Source	Analysis Technique
H1	Repair hours categorized by scope growth cause during integration	Sensor integration classified project database	Univariate Analysis
H2	Model accuracy, cumulative reward by algorithm type	Simulations of sensor functions with a sensor controller implementing Greedy, Q-Learning, Deep Q-Network, Double Deep-Q-Network, and Instance Based Learning algorithms	Univariate Analysis
H3	Model accuracy, cumulative reward by algorithm type	Environments and algorithms from H2 updated for dynamic environments	Univariate Analysis

Table 3: Tests of Hypotheses

The testability of each of the hypotheses was also an important factor for hypothesis creation. Table 4 details how each hypothesis was written for testability. Details on the methodologies are included in their respective sections in Chapter 3 – Research Methodology. The word count (WC) was also considered for each hypothesis and testability description for conciseness and explicability.

(A) Deliverable	(B) Format	C) WC
Hypothesis 1	Faulty requirements, faulty programming logic, and interface issues are the top factors that impact sensor integration.	*16*
Independent Variables	Defect cause code	*NA*
Dependent Variable	Number of repair hours	NA
Testable	Identify number of repair hours found for each defect cause code for integration growth. Quantify whether each is a significant top driver. Repeat evaluation for each factor.	*27*
Hypothesis 2	Rewarding correct subsystem interfaces through reinforcement learning algorithms results in the correct subsystem behavior on 100% of the test cases.	*20*
Independent Variables	Number of steps to train	*NA*
Dependent Variable	Model accuracy, Cumulative Reward	NA
Testable	Provide controller and sensor simulation, model as a Markov decision process, develop Greedy, Q-Learning, Deep Q-Network, Double Deep Q-Network, Instance-based learning algorithms that reward correct actions by controller.	*28*
Hypothesis 3	Training sensor controllers using reinforcement learning algorithms in stochastic environments results in correct controller actions to dynamic environmental conditions on 100% of the test cases.	*25*
Independent Variables	Number of steps to train	*NA*
Dependent Variable	Model accuracy, Cumulative Reward	NA
Testable	Provide controller and sensor simulation, model as a Markov decision process to include probability of communication failures, sensor faults, non-deterministic response times, and reward correct actions by controller.	*28*

Table 4: Hypothesis Testability

3.2 Top Factors that Impact Sensor Interoperability

Hypothesis 1 was “Faulty requirements, faulty programming logic, and interface issues are the top factors that impact sensor integration.” To evaluate this hypothesis, data was collected and analyzed from a large-scale sensor integration program. The details on the company and the specific tasks involved have been redacted due to the data being related to software from operational military aircraft. The source data was company-owned and approved for public release.

The company developed sensor controllers and a legacy aircraft isolator as part of an aircraft upgrade program. The sensor controllers and sensors themselves were intended to be plug-and-play, leveraged from another program with a high TRL.

However, the company integrating these sensors experienced substantial scope growth over time, resulting in untenable cost and schedule increases and the cancellation of the program.

During integration and test phases of the program, software, systems, and test engineers on the program documented defects found, identified root causes of the errors, assigned error codes to them, and recorded the repair hours associated with the defects. Analyzing why the additional work was required based on integration findings helped identify the top factors that drove the lack of sensor interoperability and associated integration growth.

The process of getting access to the data was prolonged and complex. First, there had to be executive approval of the strategy to remove the data. Then there was a legal review of the request, followed by clearance from human resources. After the data extraction was completed, there was a formal public release approval process for the data involving a string of reviewers and approvers including company executives. The release process for the data took three months to complete.

The processing for the data was especially challenging as the data originally contained data that was proprietary and classified. The original data set included 5,575 tasks with 245,000 data points, but it increased to 7,066 tasks with 291,000 data points based on additional data needs identified during analysis. The data used for the analysis included more than data about integration defects. It included information on planned

work tasks, referred to as stories based on agile processes. Story points were a measure of complexity. The data included descriptive data on each of the tasks as well as what project features, capabilities, and requirements that the tasks were supporting in addition to many other details on the tasks.

The original concept for the first research question was to review what planned tasks took the most amount of time for integrating sensors, but there was insufficient information in the database on cost and time. Planned task complexity as an independent variable versus actual task complexity in story points by integrated feature as a dependent variable were also data points of interest, but measures of complexity in story points were not uniformly applied to the tasks and could not be used for meaningful evaluation. Regardless, focusing on what types of defects were found during sensor integration and how much they impacted integration was more applicable to Hypothesis 1.

The total data set of 291,000 data points was too large to feasibly be redacted and released, so data reduction was performed on a classified network. Figure 6 shows that 185,829 data points were removed based on applicability to sensor integration. 26,984 data points were sanitized to remove company proprietary data, and 77,728 data points were unchanged. This data reduction to 104,712 total data points simplified the data extraction process and subsequent analysis.

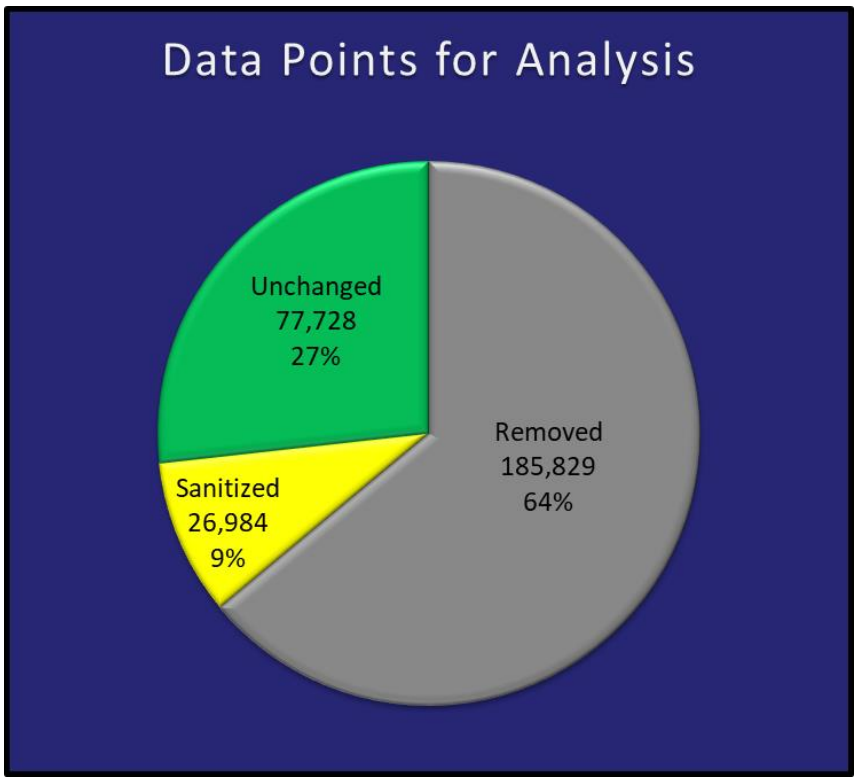

Figure 6: Data Point Reduction

Because the data extraction was so complex, it was important to get enough data out initially so that another extraction of data would not be necessary. Digital extraction of the data required several steps that were not anticipated. Sanitization of business information and system information was not expected but was required. Training in Information System Security was required, as well as training, paperwork, and authorization for data transfer authority. It was also not expected to need a disk drive installed on the PC, getting read/write permissions for the disk drive, getting access to the extraction software, or meeting additional information security requirements. The security requirements took about one month of the three-month data extraction timeline.

After the data extraction was completed, the data analysis could take place. The possible integration error codes are included in the following list and fall into categories

of administrative, engineering error, product definition, program/customer directed, and test.

Administrative - A1: Unrevised SRS updates
Administrative - A2: Unrevised Design Documents
Administrative - A3: Tools
Administrative - A4: Unrevised ICD Updates
Administrative - A5: Not a Defect
Administrative - A6: Duplicate Defect
Administrative - A7: Legacy Defect
Engineering Error - E1: Code does not meet specifications
Engineering Error - E1A: Faulty systems requirements
Engineering Error - E1B: Faulty software requirements
Engineering Error - E1C: Faulty software detailed design
Engineering Error - E2: Bad program logic introduces error
Engineering Error - E2A: Faulty timing
Engineering Error - E2B: Faulty state management
Engineering Error - E2C: Faulty data
Engineering Error - E2D: Faulty error handling
Engineering Error - E2E: Faulty logic or sequence
Engineering Error - E2F: General coding
Engineering Error - E3: Merge / integration introduces error
Engineering Error - E4: Interface issues
Engineering Error - E5: Open Defect / Bug Dependency
Product Definition - D1: Code improvement
Product Definition - D2: Test workaround added
Product Definition - D3: Premature code inclusion
Program / Customer Directed - P1: Prog/Sched directed change
Program / Customer Directed - P2: Non-SRS requirements change
Test - T1: Test Procedure Error
Test - T2: Lab Hardware Failure
Test - T4: Lab Set Up/Installation Error
Test - T5: Cannot Duplicate

Figure 7: Integration Defect Cause Codes

The hypothesis assumed that faulty requirements, faulty programming logic, and interface issues were the top factors that impact sensor integration. These items corresponded to the defect cause codes as follows:

- Faulty requirements:
    - A1: Unrevised software requirements specification (SRS) updates
    - E1A: Faulty systems requirements
    - E1B: Faulty software requirements
- Faulty programming logic:
    - E2: Bad program logic introduces error
    - E2A: Faulty timing
    - E2B: Faulty state management
    - E2C: Faulty data
    - E2D: Faulty error handling
    - E2E: Faulty logic or sequence
- Interface issues:
    - A4: Unrevised Interface Control Document (ICD) updates
    - E4: Interface issues

To evaluate whether these are significant drivers for integration, population proportion was used with the null hypothesis being that are no significant drivers of integration. The Null Hypotheses were that faulty requirements, faulty programming logic, and interfaces issues were each < 10%, which was chosen as a significance threshold.

3.3 Reinforcement Learning

3.3.1 Overview

The second hypothesis was “Rewarding correct subsystem interfaces through reinforcement learning algorithms results in the correct subsystem behavior on 100% of

the test cases," and the third hypothesis was "Training sensor controllers using reinforcement learning algorithms in stochastic environments results in correct controller actions to dynamic environmental conditions on 100% of the test cases." To evaluate these hypotheses of using reinforcement learning to learn sensor interfaces, sensor controller algorithms were implemented to learn the correct behaviors of modeled sensors. The approach was developed to bridge the gap between standards specification and sensor-specific behavior. The sensor controllers were not developed to decode sensor-specific messages or sample and interpret sensor signals, as those types of information exchanges are often standardized. Rather, the sensor controllers were designed to work at the application level to decide the correct actions for operating the sensors, transitioning them from one state to another. Also, these sensor controllers were not implementing autonomous system behaviors that execute without pilots in the loop, as in the case of Boeing 737-MAX's faulty Maneuvering Characteristics Autonomous System (MCAS) [45]. This approach teaches sensor controllers the behavior of sensors and builds policies that execute functions when required by other parts of the system.

Reinforcement learning was the machine learning algorithm category selected for learning sensor behavior. Reinforcement learning is one of the three main approaches to machine learning, in addition to supervised and unsupervised learning. While supervised and unsupervised learning use data to train models that are used to make predictions, reinforcement learning uses information that comes from trial and error. When engineers are integrating new sensors onto aircraft, they do not have a pre-existing data set that algorithms can use. However, since reinforcement learning algorithms are not dependent on a data set, they are able to learn from the new sensor or model itself.

In reinforcement learning, an agent performs an action against an environment to transition the environment from one state to another, and the agent receives a reward based on the chosen action. This reward is used by the agent to update its decision policy. The agent can then use its updated decision policy to recommend the next action to take based on the new state of the environment. The interactions between the agent and environment are shown in Figure 8.

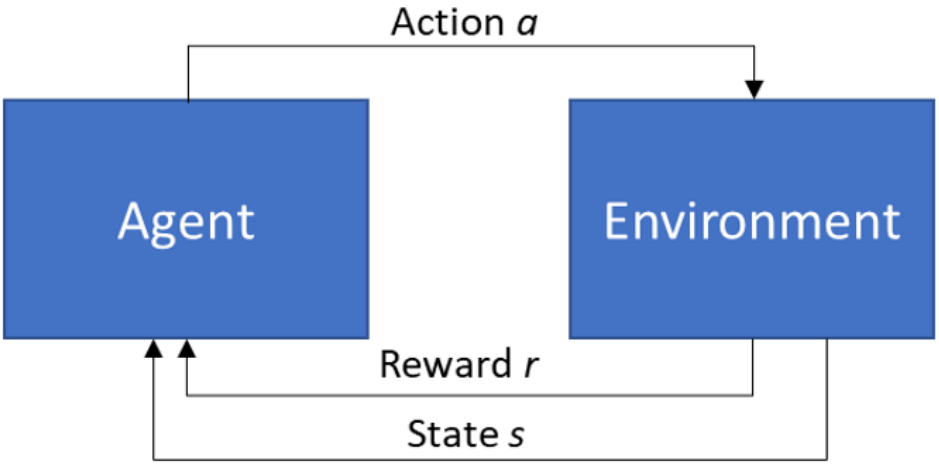

Figure 8: Reinforcement Learning Interactions

Reinforcement learning environments are modeled as Markov Decision Processes, which have a set of states, a set of actions, the probabilities that actions taken in certain states will transition to other certain states, and the rewards associated with those state transitions. Aircraft sensors can be modeled as Markov Decision Processes. Figure 9 shows a simplified startup model for a hypothetical Electronic Support Measures (ESM) subsystem. An ESM controller has four states including Off, Loading Operational Flight Program (OFP), Initialized, and Active. The four actions that can be taken by the agent (e.g., a controller) on the ESM are Power On, Wait for Initialization, Mode Change to Active, or Power Off. 100% of the time when Power On is applied when the ESM is in

the Off state, the ESM transitions to Loading OFP. Markov Decision Processes can also have stochastic behavior, where an action could transition from a state to a number of other states, based on a set of probabilities. In this example, the ESM model is a reinforcement learning environment and the controller executing the actions is the reinforcement learning agent.

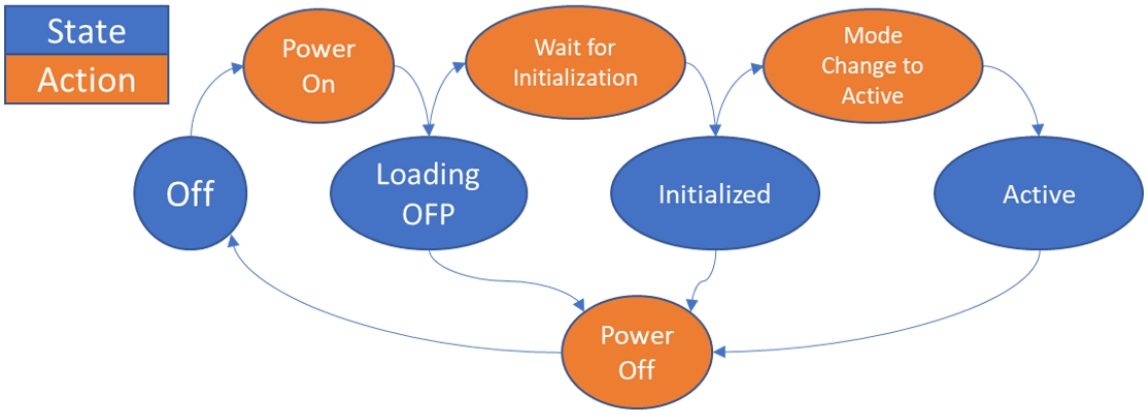

Figure 9: ESM Markov Decision Process Example

Another reason reinforcement learning is a good fit for sensor learning is because aircraft sensors with their states and actions that cause state transitions have clear equivalents to reinforcement learning elements. A set of experiments was designed to test how well sensor controller agents can train policies that implement messages and commands that execute sensor functions as sensor vendors intend without them being manually specified and coded.

The sensors in the experiments were modeled as Markov Decision Processes based on the formal framework outlined by Otterlo and Wiering [46]. The modeled sensors were implemented as reinforcement learning environments, and the sensor controllers were implemented as reinforcement learning agents. The sensor controller agents took actions against the sensor environments. These actions represented

commands via software and hardware interfaces to sensors. The actions caused the sensors to transition from one state to another according to a Markov Decision Process within the sensor model. The sensors then provided rewards and state updates to the sensor controllers that modified their decision policies for future action selection.

For the purposes of the subject research experiments, the probabilities associated with Markov Decision Processes were set to 100%. In these experiments, when an environment experienced an action in a specific state, it deterministically transitioned to another specific state. The only exception to this was in an Infrared sensor environment, where the sensor tried to detect an object that had a five percent probability of being present.

The Markov Decision Processes represented behavior logic that exists at Level 7 of the Open Systems Interconnection Reference Model (OSM/RM) as seen in Figure 10. The environments did not implement network interfaces or send bytes over data links. The rationale behind this is that advanced sensors today use open architectures like SOSA and the interfaces at the lower levels are already specified. Sensor controllers need to learn the proprietary behavior logic at the application layer.

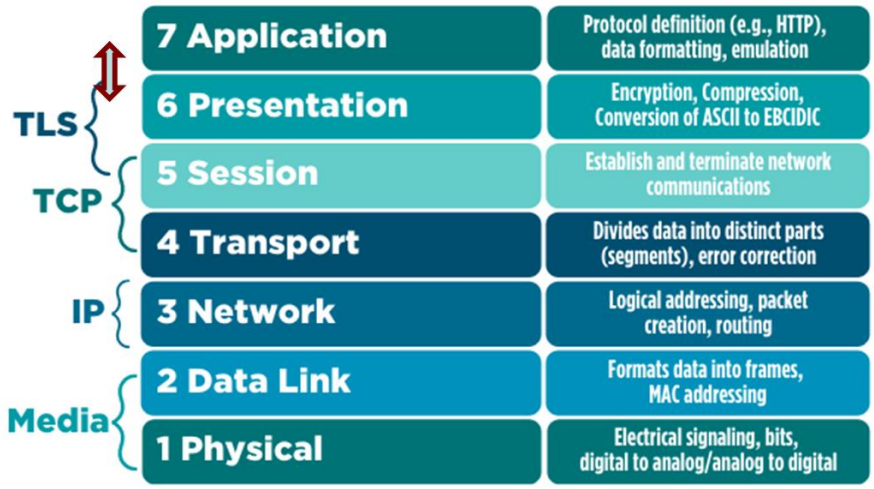

Figure 10: Mapping Protocols and Procedures to the OSI/RM [47]

The objective in the experiments was to see if rewarding correct subsystem interfaces through reinforcement learning algorithms would result in the correct subsystem behavior 100% of the time. Because aircraft sensors can be flight critical, safety critical or mission critical, less than 100% correct behavior could result in software defects that cause mission failure or loss of life.

Five different algorithms were used to learn sensor interfaces. First, the Greedy algorithm was implemented to seek high short-term rewards, and a Q-Learning algorithm was developed to seek high long-term rewards. A Deep Q-Learning algorithm was implemented that used a neural network to predict optimal actions following the same base algorithm as Q-Learning. A Double Deep Q-Learning algorithm was implemented that used a second neural network for preventing the algorithm from overestimating the utility of non-optimal actions. Lastly, an Instance-Based Learning Algorithm was implemented that could choose among actions based on its memory of previously learned experiences.

These five algorithms were implemented into agents and executed against modeled sensor functions for Global Positioning System (GPS), Optical, Infrared, and Light Detection and Ranging (LIDAR) sensors. The algorithms were also executed against a simple environment that modeled an agent moving through a maze, which was useful to determine if the algorithms were behaving as expected. Each combination of agents and environments was evaluated with a different number of total training steps, either 128, 512, 2,048, 8,192, or 32,768. Changing training steps as an independent variable was useful for determining how quickly algorithms reached optimal policies in training. The methodology for combining agents and environments for performance evaluation was similar to past studies using grid-based goal seeking environments [41] [42] rather than modeled sensor environments.

As part of this experiment, a generic reinforcement learning structure was developed that enabled execution of each of the agents against each of the environments without having to change either for different combinations. This mitigated potential variations in environment and agent implementations that could have affected testing and prevented comparisons across different agent and environment combinations.

Figure 11 shows the activities that occurred to train agents and test them against an environment. Each agent trained for a fixed number of training steps. A single training step involved three stages shown in Figure 11, line numbers 5-7. First, the agents recommended an action to take based on the current state of the environment. Next, the action was taken against the environment, with a corresponding change in environment state and reward. Finally, the agent updated its internal policy for use the next time it was

asked to recommend an action. These three stages were repeated over the course of all the training steps.

After the agents trained for the specified number of steps, the testing occurred. The trained agents executed against the modeled sensor environments as shown in Figure 11, line numbers 10-12. The agents recommended an exploitative action and took the action against the environment, changing the state. During testing, policies were not updated based on rewards received from the environment. For deterministic environments, testing occurred for 1000 steps and the results were recorded. For stochastic environments, testing occurred for 10,000 steps before the results were recorded.

Line No.	Activity
1	Execution:
2	Initialize Environment
3	Initialize Agent
4	For each training step:
5	Get an action from the agent based on the current state and exploration rate.
6	Take the action on the environment and receive reward
7	Update agent policy with the reward
8	Update the current state and the exploration rate.
9	For each test step:
10	Get an exploitative action from the agent based on the current state
11	Take the action on the environment
12	Update the current state
13	Record results

Figure 11: Psuedo-code for Training and Test Execution

The final execution engine had a logic flow as follows to execute the training:

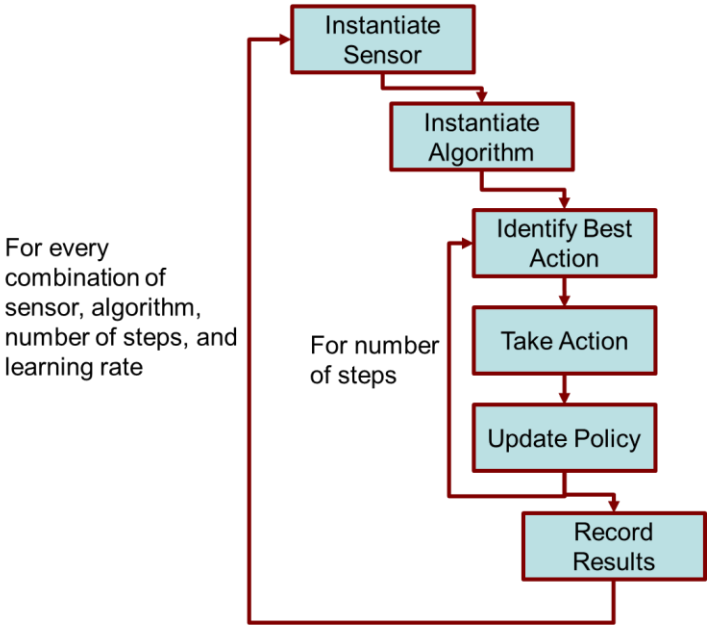

Figure 12: Execution Engine Training Process

According to Otterlo and Wiering [46], there is an exploration-exploitation problem where agents need to balance deciding whether to learn more about the environment or maximize reward gain based on what it has already learned. When agents explore, they take random actions to learn more about how their actions affect the environment. When agents exploit, they will choose the action that they expect will provide them with the maximum reward. Exploitation is a greedy decision whereas exploration is a random decision. The Q-Learning-based experiments followed the Ɛ-greedy method as described by Otterlo and Wiering [46], and the probability of taking an exploration action versus an exploitative action linearly decreased from occurring 100% of the time to 0% across the number of training steps taken. At the beginning of training, agents explored most of the time and toward the end of the training, agents exploited most of the time. During testing, exploitive actions were always selected.

Figure 13 through Figure 16 show high level SysML structure of the system under test. Figure 13 shows how the system is composed of an environment, an agent, and an

execution engine. The environment is a generalization of five child environments and the agent is a generalization of five child agents. The block diagram shows a sixth agent (Random) that was implemented but not used in any formal testing.

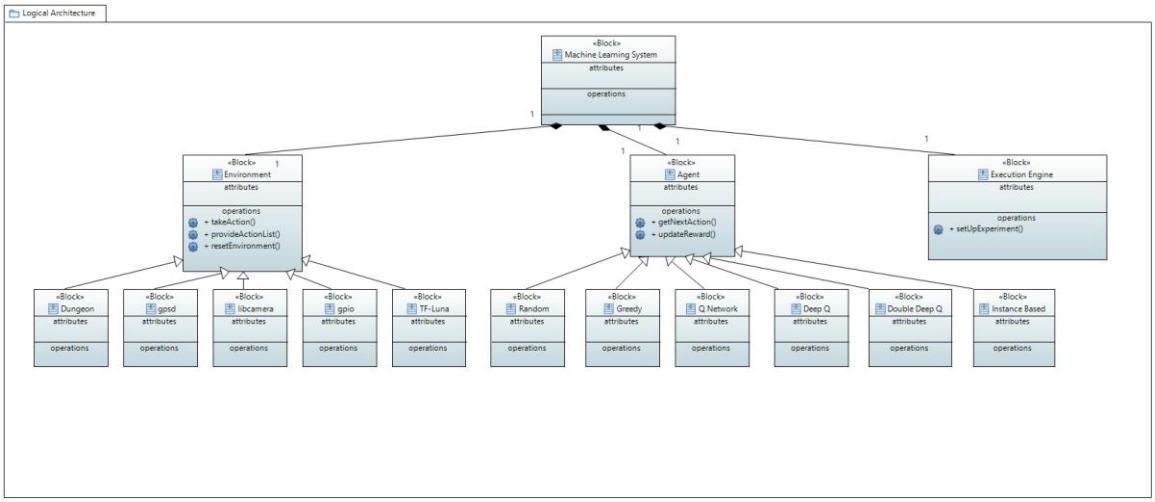

Figure 13: Logical Architecture

Figure 14 shows the three subsystems and their interfaces to each other. The execution engine gets the set of actions from the environment, gets the recommended action from the agent, takes the action on the environment, and updates the agent with the results.

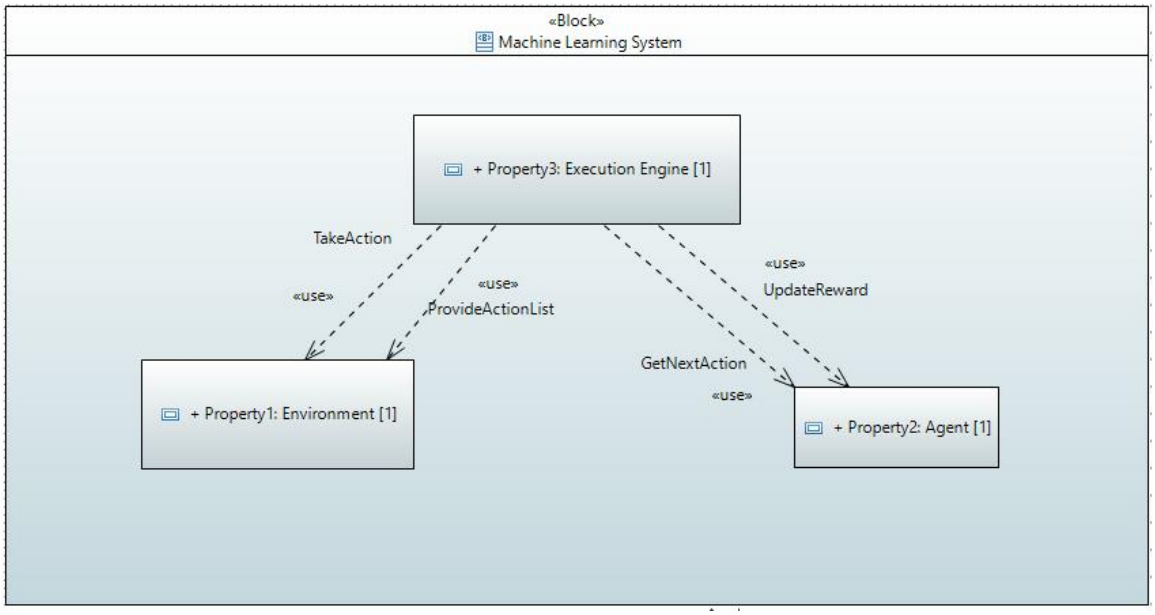

Figure 14: Subsystems and Interactions

Figure 15 is a use case diagram that represents the initialization of the execution. Learning rate, discount rate, and number of training iterations are initialized and then the environment is created. The set of actions from the environment are passed to the agent on initialization.

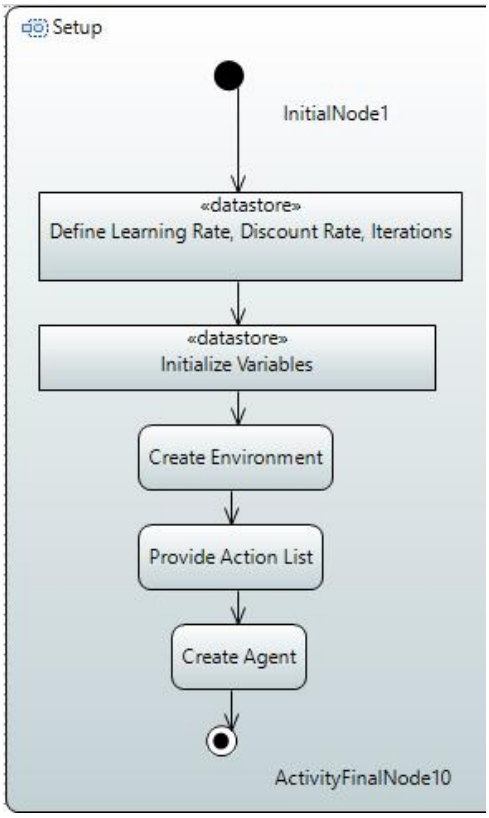

Figure 15: Initialization Use Case

Figure 16 shows another use case for training the reinforcement learning agents. The execution engine gets an action recommendation from the agent, takes the action on the environment, and then updates the agent with the reward. The test use case is identical to the training use case except that the test use case does not update the agents' decision policies with new reward values.

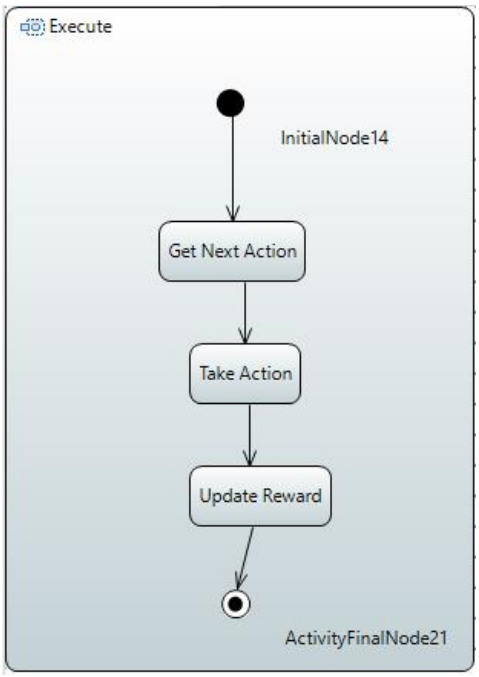

Figure 16: Training Use Case

3.3.2 Agents

Sensor controller agents implemented different algorithms under test, and these agents were all able to initialize, recommend actions, and update their internal policies. Five reinforcement learning algorithms were selected for the agents to show a variety of algorithm characteristics and build on existing research [41, 42]. Greedy and Q-learning algorithms were chosen to differentiate exploitation from exploration. Deep Q-Learning was chosen because it has higher expressive power than Q-learning. For example, it can handle continuous state spaces and high numbers of discrete states. Double Deep Q-Learning was chosen to solve a particular weakness of Deep Q-Learning related to how it estimates reward. Lastly, Instance-Based Learning was chosen due to its ability to handle dynamic environments.

The first and simplest agent was the Greedy agent, which was adapted from the Valohai Q-Learning GitHub repository [48]. The Greedy agent was a completely

exploitative algorithm. The Greedy agent maintained a table of reward values for each state and action combination. The initial values in the table were empty and were populated as the Greedy agent took actions and received rewards. The Greedy agent always recommended the action with the highest reward value given the state that the environment was in and was never explorative. If multiple actions had the same reward value, then it randomly chose between them. After the action was taken, the reward associated with the action in a particular state was added to the reward value table. Greedy agents can get stuck with locally optimal policies and take actions based on small rewards, rather than exploring to find actions that would lead to larger rewards.

The next agent was a Q-Learning agent and was also adapted from Valohai repository [48]. The Q-Learning algorithm has several main elements. Q-values represent the utility of taking an action in a certain state. Q-values are stored in a Q-table, which has a Q-value for each action-state combination. When being exploitative, a Q-learning agent selects the action with the highest Q-value, and when the agent is being explorative, it chooses a random action. Q-values are updated based on the Bellman equation [49].

$$Q^{new}(s_t,a_t) = Q(s_t,a_t) + \alpha * (r_t + \gamma * \max_a Q(s_{t+1}, a) - Q(s_t,a_t))$$

$Q^{new}(s_t,a_t)$ is the new Q-value that is stored in the Q-table. $Q(s_t,a_t)$ is the current Q-value. r_t is the reward that comes from the environment based on the action taken (a_t). $\max_a Q(s_{t+1}, a)$ is the best predicted value in the next state, and it is scaled by γ, which is the discount rate. The discount rate changes how important future values are versus current reward values. In the experiments, the discount rate was 0.95, which gave the algorithm a strong consideration for rewards in future states. α is the learning rate and adjusts how fast Q-values are updated by new rewards. When α is near zero, new data has

little effect on the Q-values, and when α is one, the old Q-values are completely overridden by the new Q-values. On the test cases that used Q-learning algorithms, 0.1, 0.3, 0.5, 0.7, and 0.9 were the learning rates $\alpha \in (0,1)$, based on Bertsekas and Tsitsikils's definition of well-behaved iterative stochastic algorithms. Q-learning algorithms with these learning rates are guaranteed to eventually converge to optimal solutions [50].

The third agent tested had a Deep Q-Learning algorithm. The implementation of the Deep Q-Learning algorithm was adapted from the Balsys's GitHub repository [51]. Deep Q-Learning is similar to Q-Learning except instead of a Q-table, there is a neural network that maps the states to the actions. Neural networks are made up of layers. The input layer is the set of variables coming into the neural network, and the output layer is the set of outputs coming from the neural network. Between them are one or more hidden layers made up of nodes that each have an associated weight. The weight in each node is multiplied by each input and added together. Then the values are used in the nodes of the next layer with their own weights. The values move through each of the layers and then calculate values for output nodes. The maximum value in the output nodes is the predicted output. This process is known as forward propagation. The predicted output is compared to the target output and a process called backpropagation occurs to adjust the weights of the nodes to achieve more accurate predictions.

For Deep Q-Learning in particular, the input nodes are the possible states from the environment, and the output nodes are the actions that the agent can take. These experiments used three hidden nodes as shown in the following figure, with 512, 256, and 64 nodes in each layer. The number of input states varied based on the environment, and

the number of output states varied based on the number of possible actions that the agent could take. The values in the output nodes represented the Q-values.

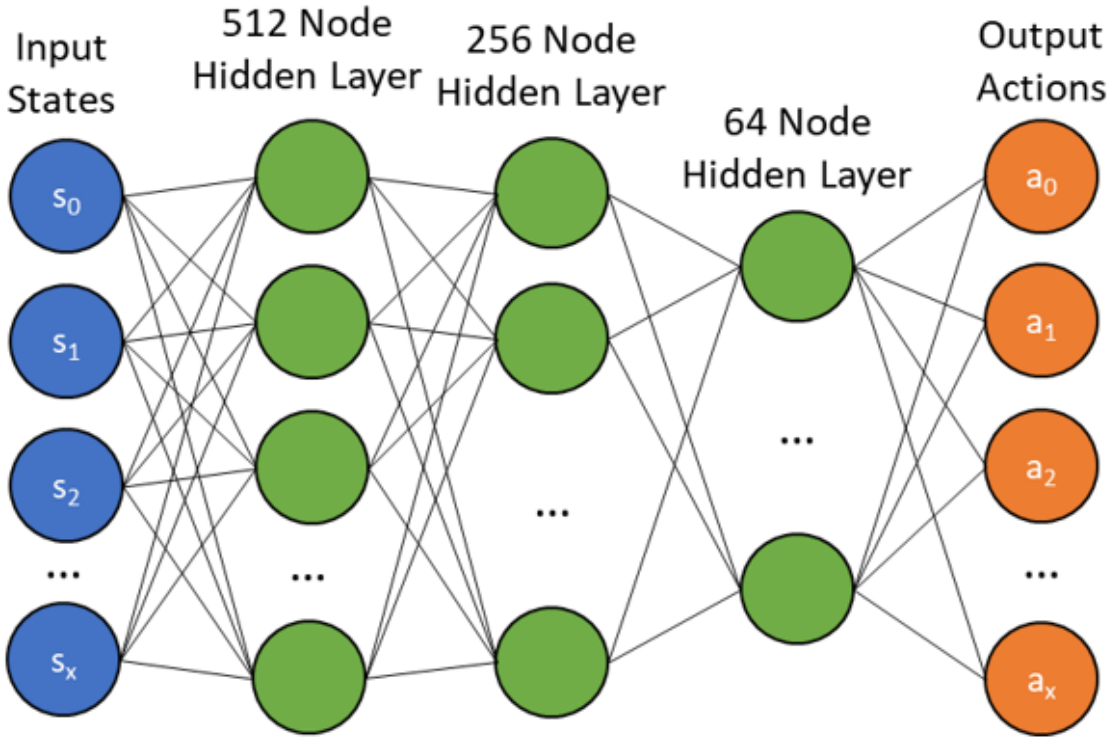

Figure 17: Neural Network Structure

In training of the Deep Q-Learning agent, the set of past actions, states, and rewards was sampled and used to retrain the model and set the weights for the layers. The Bellman equation was used like in Q-Learning to set the target Q-values.

The next agent was a Double Deep Q-Learning agent. In normal Deep Q-Learning, a single neural network is used to make predictions of Q-values and select which action to take. Q-Learning and Deep Q-Learning have the same potential weakness of overestimating Q-values if a non-optimal action is selected and continually given a reward. Double Deep Q-Learning addresses this by having two neural networks. The first is used to predict and select which action to take, and the second is used to calculate the

Q-values for the next state. In Deep Q-Learning, the predicted Q-values and actions are used to train a single model, but in Double Deep Q-Learning, the predicted Q-values and recommended actions come from different models. The predictive model is trained with the Q-values from the target model, and the actions are selected from the predictive model. The Double Deep Q-Learning implementation was adapted from Balsys's repository [51].

There were several issues with the initial implementation of the deep neural network. The first deep neural network model would fail on initialization when there were greater than 64 valid actions from the environment. The first model had 4 layers total, with two hidden layers and 8 neurons in each. To address this, a third hidden layer was added, and the number of neurons were increased to 512, 256, and 64 neurons per layer. This is a shallow neural network, and future research will explore neural network design, including the number of layers and neurons in each layer. After the additional third layer and neurons, there were no further identified action limitations.

Another question regarding neural network design was how to provide inputs into the model. There were three ways identified to pass data in, as Figure 18 shows. First, there could be binary inputs. The number of binary inputs equals the number of possible states in an environment. If there were four possible states, then there would be four possible inputs: [0,0,0,1], [0,0,1,0], [0,1,0,0], or [1,0,0,0]. In a continuous state vector input, there would be one input into the model. Using the previous example, [1], [2], [3], or [4] would be the possible inputs. An advantage of this approach is that any real number could be valid as an input. The third approach was the blended approach, where there could be both continuous and discrete binary inputs into the model. If there were

three relevant states in the model, such as positions in degrees for X and Y and an on/off of an aperture, then the inputs could be represented by [0 to 180, 0 to 180, 0 or 1]. The neural networks implemented for the experiments all used binary inputs as in the first example. This created a large number of inputs into the model for the environments with large numbers of states. There are potential efficiencies that could be realized by trying other input state representations which will be evaluated by future research.

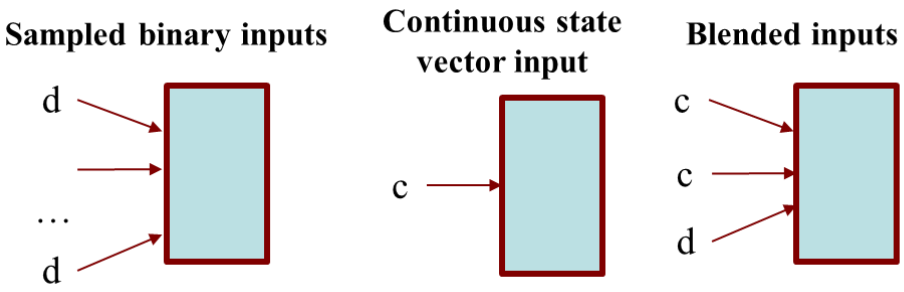

Figure 18: Input State Representation

The final agent was Instance-Based Learning (IBL). IBL is a cognitive algorithm that makes decisions following the model of human decision-making known as Instance-Based Learning Theory [52]. States, actions, and rewards are represented as instances that accumulate in memory as they are experienced. When an agent is asked to recommend an action, it will evaluate past instances for similarity to recommend what action would be most likely to produce the highest reward. Prior research by others indicated that IBL may work faster and be more adaptable to dynamic environments than Q-Learning [41] [42]. IBL has also been used to implement collaborative multi-agent systems [53]. However, IBL does not scale very well due to memory and processing requirements that increase as the number of observations increase. The IBL algorithm used for the study was provided by the open source SpeedyIBL project [54].

IBL was chosen as an algorithm because of the prior research comparing IBL with Q-Learning [41, 42], but it was not clear how to use IBL, an unsupervised clustering algorithm, in a reinforcement learning environment. SpeedyIBL and PyIBL were both considered as the IBL API, and both had the ability for agents to pick from a list of options and then respond with the utility of the choice. The IBL implementations stored actions, states, and their utilities in memory and then would classify new instances based on similarity to previous instances. In this sense, the implemented IBL algorithm was both an unsupervised clustering algorithm and a reinforcement learning algorithm.

3.3.3 Environments

Reinforcement Learning environments represented sensors and were responsible for executing state transitions, handling actions that were taken, and providing a reward based on which actions were taken. Environments were implementations of Markov Decision Processes, and they modeled sensor behavior. Five environments represented a broad set of functions from different aircraft sensors and had varying numbers of possible states and actions to gauge algorithm performance at different levels of complexity. The range of actions and states across the different sensor models are shown in Figure 19.

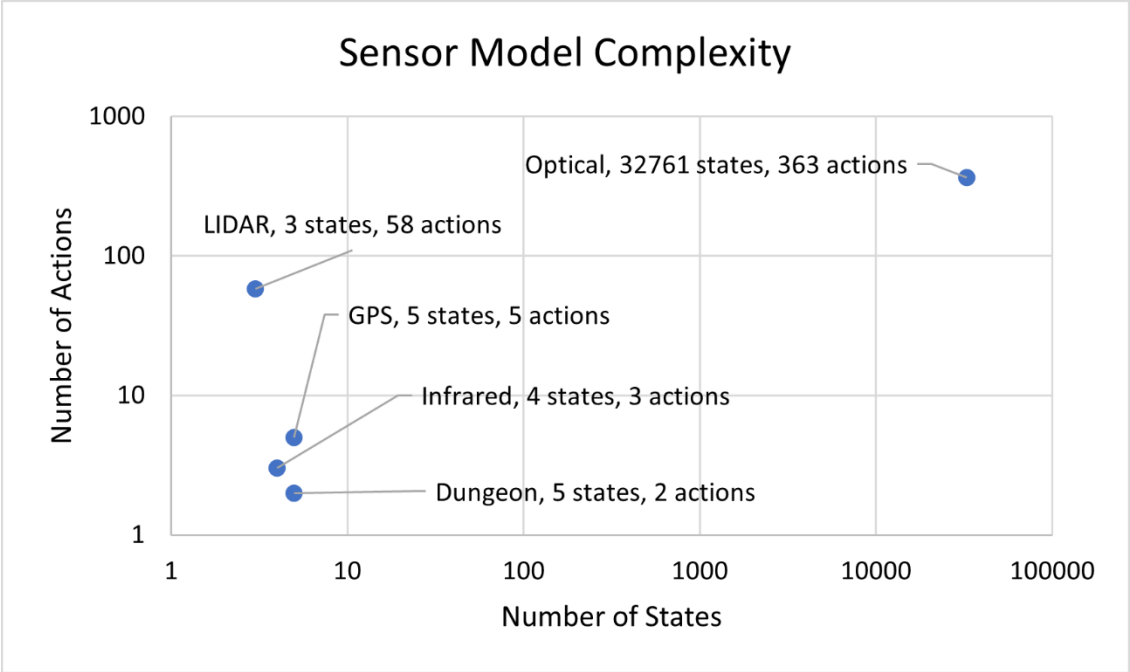

Figure 19: Sensor Model Complexity

The reward strategy was typically based on small rewards being provided for correct actions chosen leading to a big reward being provided for the final action in a sequence being chosen. If an action was chosen that would result in an error, then it would receive a negative reward. No reward would be provided otherwise. This strategy incentivized short term correct choices as well as long term exploration for the optimal sequences.

The first and simplest environment was the Dungeon environment, based on Valohai's implementation [48]. Dungeon was a test environment that illustrated the effects of Deep Q and Q-Learning algorithms compared to random and exploitative algorithms and was designed to promote exploration over exploitation. In Dungeon, there were five squares leading from the entrance to the exit. When an agent entered the Dungeon, it received two points for going back out the front and ten points for going all the way through. Move Forward was an action and Move Backward was another action,

which went all the way back to the front. Exploitative algorithms could quickly find that two points were associated with exiting the front door while exploratory algorithms could find that the better reward was at the end of the Dungeon, causing agents to take short term losses for longer term reward gains. The Dungeon environment was refactored as part of the subject research to handle more complex state representations used in algorithm implementations. The Dungeon environment was also refactored to align with the environment framework from the sensor models. Dungeon was modified so that actions and resulting states were deterministic. While not a sensor model, the Dungeon environment was useful for developing and testing algorithm implementations due to its small number of states and actions and clear global optimal solution. The Markov Decision Process for Dungeon is shown in Figure 20. This Markov Decision Process is simpler to depict than the Markov Decision Processes for the actual sensor models, which have more states and actions with more complicated transitions.

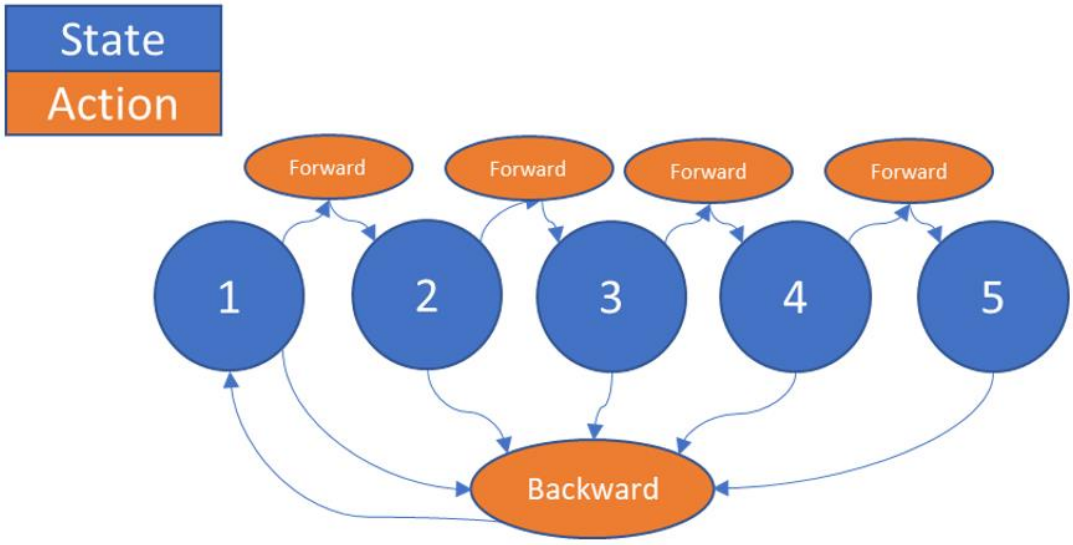

Figure 20: Markov Decision Process for Dungeon Environment

The second environment tested was a modeled Global Positioning System (GPS) sensor. GPS is a navigation system that uses satellites to provide location and time to a receiver on an aircraft [55]. GPS Daemon (GPSD) is a software package that collects data from GPS receivers and then distributes it over aircraft networks [56]. GPSD is used to manage a range of navigation and precision time devices, and it is widespread in embedded systems. A prevalent standard for GPS communications is National Marine Electronics Association (NMEA) 0183 [57]. GPSD can communicate with GPS receivers using NMEA 0183 over serial, USB, Bluetooth, Transmission Control Protocol and User Datagram Protocol. GPSD is simple because the daemon and monitor applications can be launched with minimal configuration [56]. In the experiments, a sensor controller was responsible for sending a software command to launch the daemon that connected to the receiver and then a command to launch a monitor that listened for data to come from the daemon. The desired GPS function modeled for the experiment was the initial Start Daemon command followed by the Start Monitor command. The states included Daemon On and Daemon Off along with Monitor On and Monitor Off. The Error state was also entered if the commands were executed in a way that would cause the application to fail. The possible actions were Start Daemon, Stop Daemon, Start Monitor, Stop Monitor, and Reset Environment. With five states, and five actions, the GPS environment was slightly more complex than the Dungeon environment.

Prior to the use of the Markov models for training, GPS was integrated with a controller using a physical GPS sensor. A GPS controller was hosted on a Linux virtual machine, and it could start the daemon and the monitor also running in the virtual machine. GPSD connected with the GPS sensor on an Apple iPhone, and the monitor was

able to receive location information. The machine learning algorithms were implemented in Google Colab's cloud environment, but the GPS environment that communicated with the physical sensor was not able to be ported to cloud environment. Google Colab was not able to provide parallel processes to accommodate separate sensor, listener, and monitor processes while also enabling the appropriate ports for NMEA. Figure 21 shows the components included in the Linux VM environment. This environment was discarded and replaced with the Google Colab-based MDP environment for the formal testing.

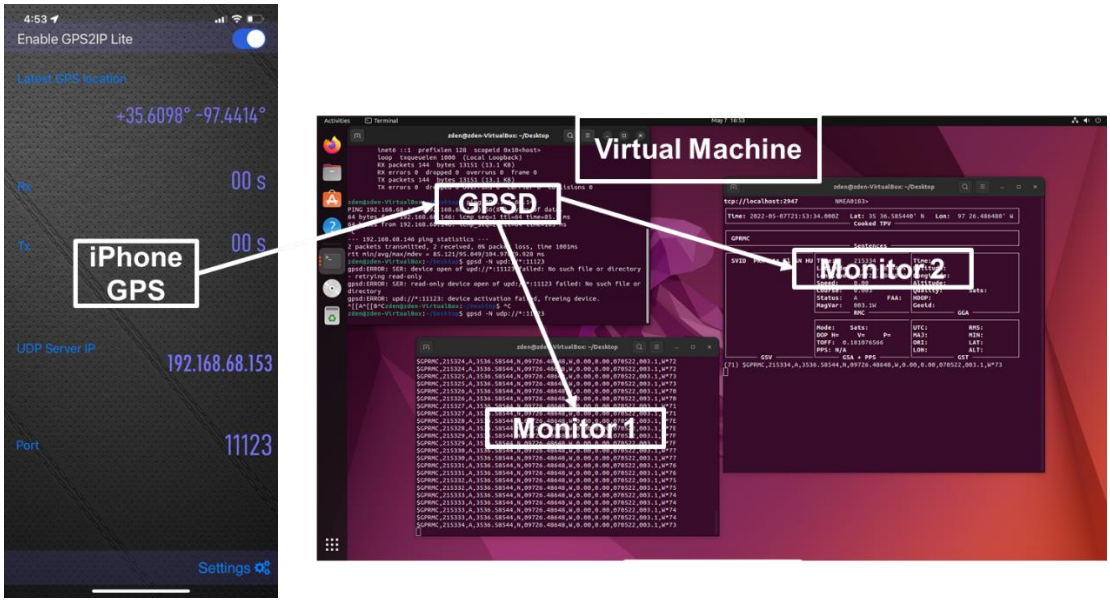

Figure 21: Integrated GPS Environment

An optical sensor model was the next environment included in the experiments. Optical sensors can be used for taking surveillance images on aircraft. These sensors can be mounted on a pan and tilt turret. The interfaces modeled for this experiment were the interfaces for the Pimoroni Pan and Tilt Hat [58] and the Raspberry Pi Official Camera Module V2 [59]. The interface to the Pimoroni Pan and Tilt Hat has separate Pan and Tilt commands that each require inputs from -90 degrees to 90 degrees. The interface to the camera module takes a single capture command. The desired functionality in the

experiments was to move to a target pan angle, a target tilt angle, and collect an image. The number of possible actions that could be taken by the agent was the sum of the number of Pan, Tilt, and Collect Image actions, which was 363. The Pan and Tilt actions were granular to each degree between -90 and 90 degrees. The number of states was the number of Pan actions multiplied by the number of Tilt actions, which was 32,761. The state space was composed of a range of discrete binary inputs, one per possible state. The Optical sensor interface was the most complex of the tested sensor model environments due to the large number of possible states and actions. Because the optical model was the most complex model, the algorithms were trained with 262,144 steps in addition to the other training durations.

Prior to the MDP-based virtual environment, a virtual optical environment was constructed. However, the challenges in the virtual optical sensor environment resulted in the pivot to the MDP model and Google Colab approach. First, the libcamera and libcamera-apps libraries and software interfaces were computing resource-intensive, driving the need to increase virtual machine size to accommodate them. After building them and testing them, it was challenging to fake an optical sensor interface using I2C over a virtual serial channel. The desired test scenario also used a pan and tilt system that would be too slow for training models if it were physically implemented. Image processing to validate successful collection would also have been processor intensive. The challenges encountered in the implementation of optical were at lower layers than OSI Layer 7, which was where the research was intending to evaluate effectiveness. The solution was to combine the sensor and environment processing and emulate it at the libcamera and servo interface. The libcamera and pan tilt hat functions were replicated

into actions that could be executed by agents. The research did not need to determine if physical pictures could be collected but it did need to evaluate if behavior could be learned. In practical applications of this methodology, the same paradigm could be followed, where controllers are trained with virtual models rather than physical or virtual implementations.

The next sensor model was an Infrared Search and Track (IRST) model. These sensors are used to detect objects that have infrared signatures. Infrared can be used to passively detect objects in battlespaces like missiles or enemy aircraft. The function modeled was the scanning for objects and notification of whether or not an object is present. The interface was modeled after a Raspberry Pi 3 connected to an infrared sensor module as configured by Raj [60]. A correctly trained agent scanned for objects and, if an object was present, turned on an LED. If the object was not present, the LED was off. Five percent of the time, the environment reported that an object was present, and the remaining 95% of the time, the environment reported that an object was not present. This introduced a stochastic process into the infrared environment.

The final environment tested was a modeled Light Detection and Ranging (LIDAR) sensor. LIDAR sensors are used to determine distance to objects. There are many applications of LIDAR sensors on an aircraft. LIDAR can be used for high resolution 3D mapping, underwater mine detection, and airborne threat detection and identification. The functions modeled were the initialization, distance detection, and shutdown for the TF-Luna LIDAR module [61]. The environment had three states. The first was when the serial connection to the module was closed, and the second was when the serial connection was open. The third was when the serial connection was open with

data being read from the serial connection. This environment had a simple state space, but the action space was more complicated because the agent had to learn the appropriate combinations of baud rates and serial ports to use to connect to the modeled sensor. The agent had 14 possible baud rates and four possible serial devices to choose from.

3.3.4 Stochastic Variations

Separate tests were run to evaluate different research hypotheses. Hypothesis 2 was, “Rewarding correct subsystem interfaces through reinforcement learning algorithms results in the correct subsystem behavior 100% of the time.” This hypothesis was tested using primarily deterministic sensor environments where Markov Decision Processes transitioned to set states with 100% probability. Hypothesis 3 was, “Training sensor controllers using deep learning algorithms results in correct controller actions to dynamic environmental conditions 100% of the time.” This hypothesis was tested using stochastic sensor environments where Markov Decision Processes modeled moving from one state to potentially a set of states based on an action and a probability distribution.

The original deterministic dungeon environment provided a reward of 10 points for reaching the end of the dungeon and two points for going back out the front. In the stochastic environment variation, 10% of the time, the agent “slipped” and moved in the opposite direction of what the agent intended to do. To continue to incentivize the long-term reward, the reward had to be increased to 15 vs 10 because the probability of the “slips” made it more advantageous to leave the dungeon and get the short term two point reward vs explore the dungeon for the long term reward. Figure 22 shows the Markov Decision Process for the dungeon environment with 10% probability of slipping occurring.

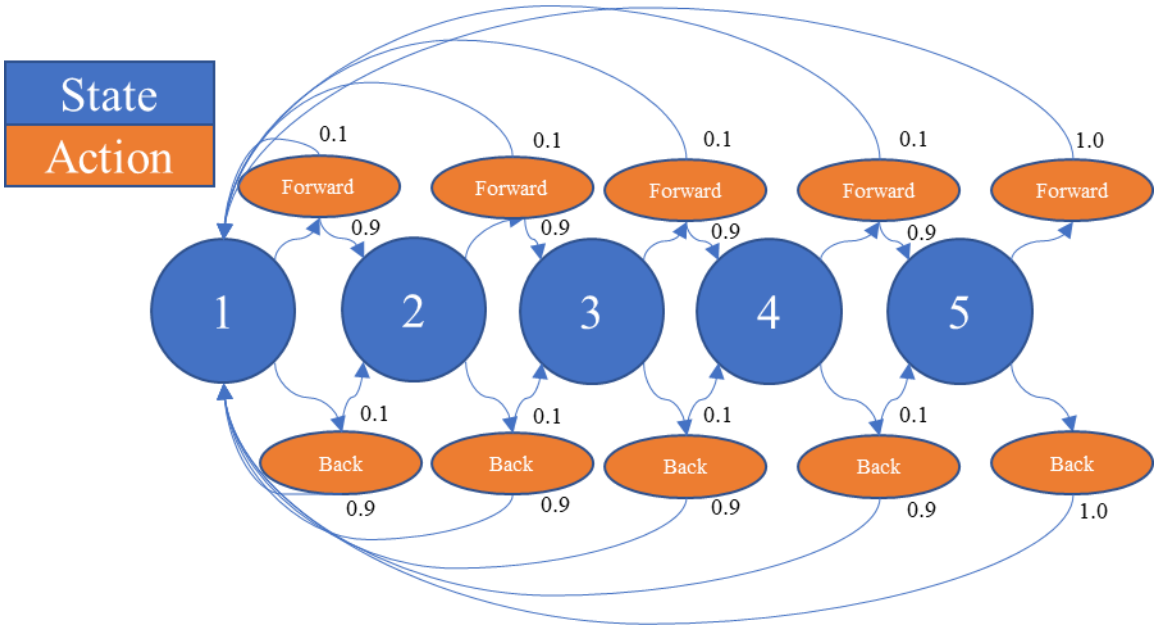

Figure 22: Stochastic Dungeon Markov Decision Process

The original GPS environment allowed a controller agent to turn on and off monitor processes and GPS daemons. The stochastic variation simulated an occasional software failure when a controller process would attempt to start the daemon or the monitor. In a real-world implementation, this type of failure could occur due to a hung Linux process or a packet loss for a command over a network interface. The stochastic GPS environment introduced a 5% probability of failure after a command to turn on the monitor or the daemon as shown in Figure 23.

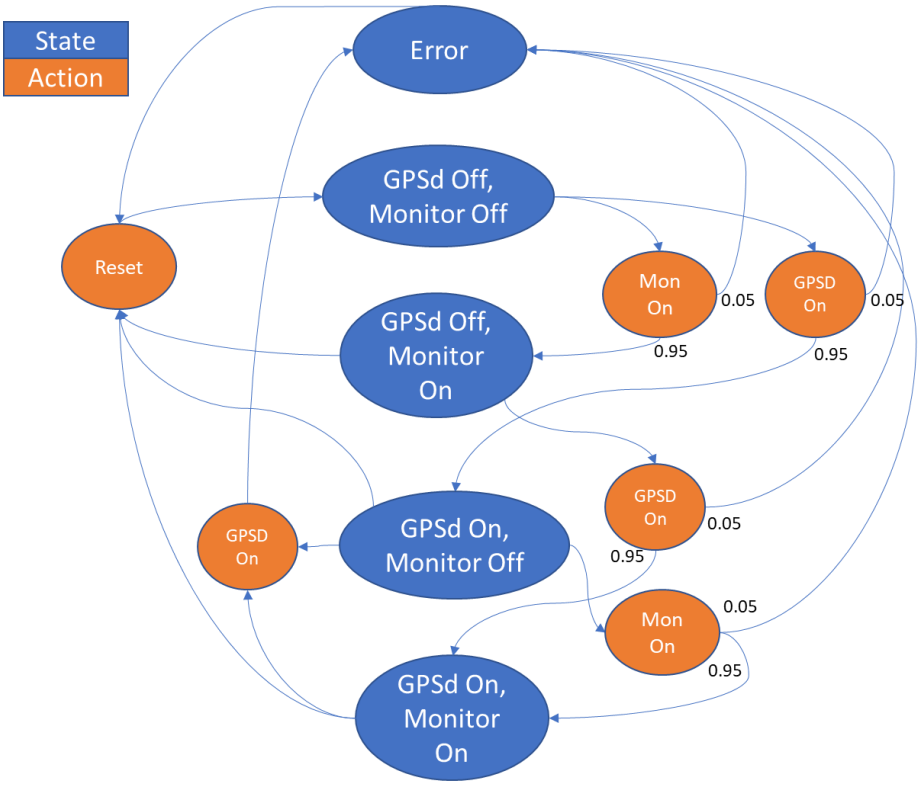

Figure 23: Stochastic GPS Markov Decision Process

Infrared had no changes between the original implementation and the stochastic implementation because there was a stochastic element originally included in the test for the GPS environment. The Infrared environment's goal was to detect whether an object was present in the environment, and there was a 5% probability that the object was present, as seen in Figure 24.

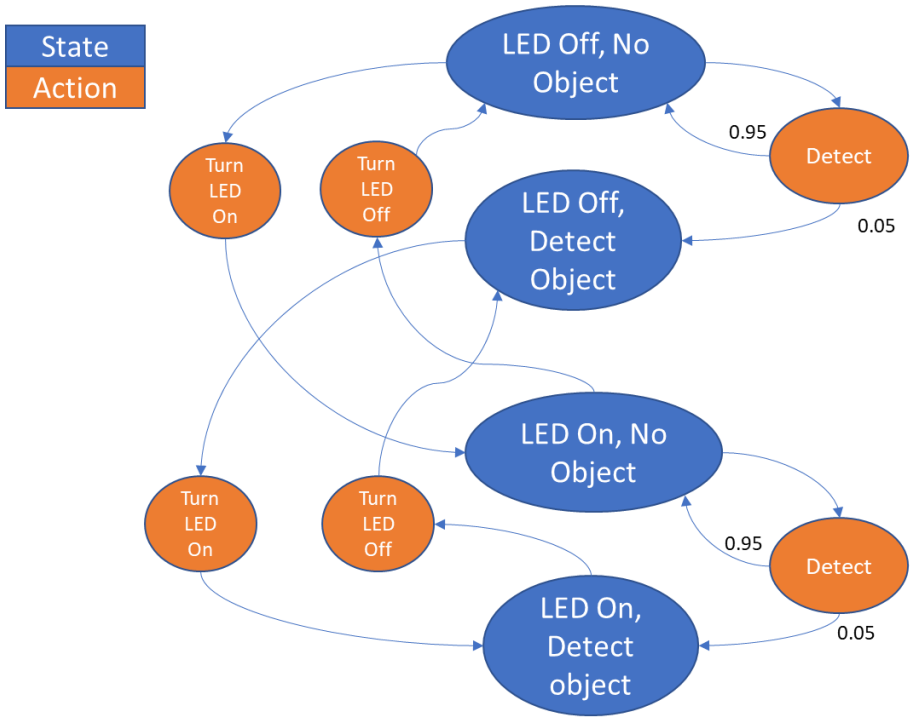

Figure 24: Stochastic Infrared Markov Decision Process

The LIDAR environment transitioned a controller through a series of serial commands to include opening a serial port with the correct baud rate and port number and reading the serial data. The stochastic variant introduced a possible hardware failure with 20% probability that a hardware failure would cause a data read issue. This type of failure can happen if a serial connection is faulty or intermittent. The appropriate controller behavior in these instances was to raise an error and then close the port to try again. The associated MDP is shown in Figure 25.

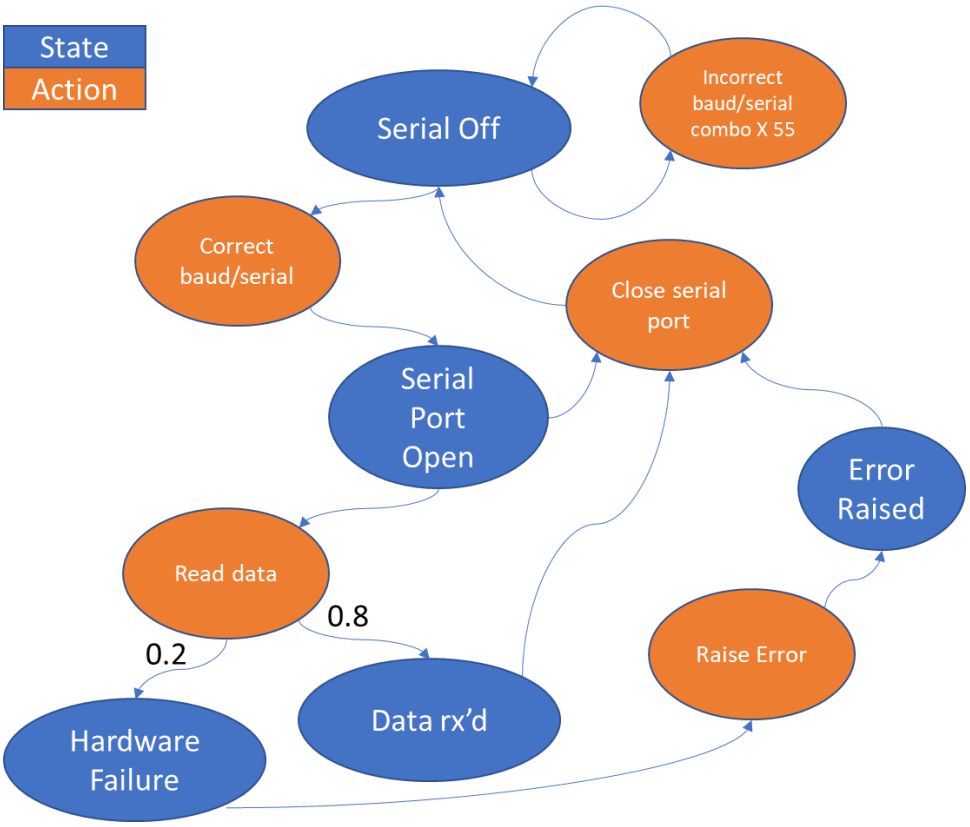

Figure 25: Stochastic LIDAR Markov Decision Process

The original optical environment had pan and tilt actions as well as an action to collect an image. The variant of the optical environment had simulated timing, where the modeled turret had a certain amount of time to pan or tilt to a location or else would not achieve the desired position. The addition of timing considerations increased the complexity of an already complex model.

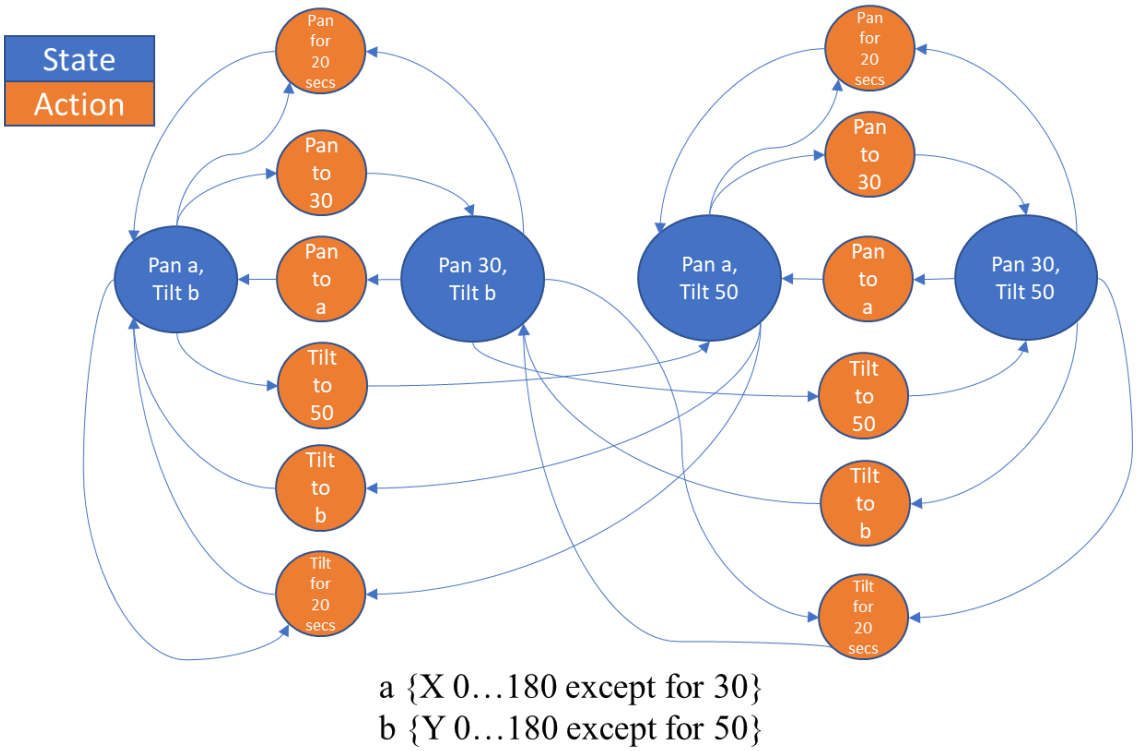

Figure 26: Stochastic Optical Markov Decision Process

3.4 Reinforcement Learning Methodology Considerations

Execution was one of the three subsystems for the reinforcement learning experiments, and it evolved to become more generic to broadly execute different types of agents and environments. It was important for there to be a single execution engine to reduce variability in how agents were trained and tested. The final execution software was identical regardless of which agent was combined with which environment.

The agent and environment subsystems were also executed as base classes that child classes inherited from, allowing them to have common methods and variables. In the beginning of development, agents would be implemented differently from each other. They had different ways of executing algorithms and different ways of representing states and actions. For example, neural network-based agents used vector inputs for states rather than single variable inputs. Each time a new agent or environment was added, the base

classes had to be refactored so that they were more generic and able to accommodate the range of use cases. This lengthened the development process, but as a result, the software implementations became highly adaptable to combine any agent with any environment.

Another software mechanism that enabled versatility was that actions were implemented as functions within the constructor of each environment class and pointers to the functions were added to a list of actions. After the execution engine used an agent to select an action, it would execute the action through an environment object, which would execute the function that corresponded to the action. This mechanism was highly adaptable to different types of scenarios. Actions could correspond to an API with a set of inputs or to a function that sends a message in a specific way. Actions that the algorithms selected could be anything that a software function could implement or point to, expanding the set of potential applications.

3.4.1 Agent Training

There are some additional methodological considerations related to agent training. Reward was recorded for each training step, but the cumulative reward values are not as useful as rate of cumulative reward increase. Faster reward gain indicates that learning is taking place. Even though cumulative reward may be high, it does not indicate the model will be effective for the test.

Training and test resources were challenging to maintain. Agent training was accomplished in Google Colab, but the free version was not sufficient for high RAM requirements and long runtimes required for testing. Training would frequently fail or stop. Colab Pro was required for priority on CPUs and GPUs, additional RAM, and longer run times. Hypothesis 2 was evaluated serially but to save time, Hypothesis 3 was

evaluated using three parallel sessions. One of the resource issues was that the neural network-based models kept growing during training, up to 81 GB of RAM that was able to be allocated by Colab. Tensorflow was the API used for the neural network model creation, and future research needs to consider if the model growth is expected and scalable or if an alternate neural network implementation is required.

3.4.2 Agent Testing

Training and testing were originally consolidated into a single step. Accuracy was originally tracked during training for exploitative actions. Successful training resulted in 100% of the exploitative actions being chosen. Accuracy at the end of the training would indicate how well the algorithm performed. To visualize this, accuracy was calculated based on a moving average, which was processor-intensive and did not effectively represent the accuracy at completion. To improve this, the stage for training and testing was split up into a distinct training stage and testing stage. Rather than get a moving average for accuracy over the course of training, there would be a single accuracy value associated with a trained model executing against a test set. The difference between the training and test stage was whether or not the agent was allowed to update its policies based on the rewards from the environment. Where before accuracy improvements were tracked during testing, for the final configuration, training occurred first and then executed the trained models against the sensor simulations, quantitatively recording the accuracy in a single number.

3.5 Software Development Tools and Methods

3.5.1 Cloud Execution

The agents, environments, and the execution engine were implemented in Python and run in Google's Colab Pro environment. Other configurations attempted were local execution on a desktop PC as well as a Colab environment using the desktop PC's resources. The Google Colab Pro cloud environment was much faster than local PC execution. The ability to set up packages for the Colab session and then be able to restart fresh versions of them was useful for reproducibility. No residual data elements that may have caused errors in testing could be carried over from one session to another since the sessions were new instances of an environment.

There were several downsides of Google Colab environment including accessing system resources like network interfaces, supporting desktops, supporting graphic user output, and supporting multi-threading. The language was also natively limited to Python. For the purposes of rapid execution of single-threaded Python scripts, Google Colab was effective, but there is a need for a production machine learning environment for future work.

3.5.2 Artifacts and Version Control

Source code was written in a Jupyter notebook and saved as a single .ipynb file. All source code written was contained in the file to promote portability and a single file that could be saved along with the test artifacts collected. Storage for all files was accomplished using a Google Drive that was mounted by the Python code on startup, rather than stored locally on the Colab cloud instances. This provided the ability to have a

version history for files and a way to get test artifacts during the test runs to ensure that execution was working as expected.

Due to the extended duration of tests, it was also important to be able to save and load trained models to and from the cloud if a session stopped for any reason. Saving and loading models was also important as it provided the ability to repeat the experiments and reproduce the test results. Saving and loading models was provided through NumPy save and load methods, treating trained models as binary files. Since the Google Cloud drive was mounted to the server, NumPy could save and load models as if they were being saved and loaded locally.

Early versions of test environment also included results that were printed out to the console as the tests were executed. To reduce post-processing requirements, the code was updated to track and save results to Google Drive rather than print them on the screen. This was also accomplished using NumPy's savetxt method, similar to the model save approach. The loadtxt method was also used to access the results prior to generating data analytics and graphics.

During implementation of the algorithms and environments, debugging code issues was very challenging. Training the models was a very slow process, and understanding why a model was recommending a certain action was often unclear. To address this, the training and test processes were instrumented, to include what actions were taken, what the start and end environment states were, and why the specific action was taken (i.e., exploratory vs exploitative). Post-processing the data would show where unexpected state transitions were happening or where algorithms were recommending something other than what they should have been learning. Understanding neural network

decision rationale is challenging but the lack of runtime debugging capabilities in Google Colab slowed down development. A production machine learning training and test environment should provide runtime debugging.

3.5.3 Software Packages

The software environment contained several software packages that were used to accomplish certain specialty functions. NumPy was used to save and load text-based results and binary trained models from Google Cloud. Results were loaded, consolidated, and summarized, and matplotlib provided visualization of the results. Tensorflow provided the neural network model framework. The IBL algorithm implementation was provided by speedyIBL. PyIBL was also used prior to speedyIBL but was not chosen as the final IBL implementation due to it not being clear how environment states were handled by the library.

Chapter 4 – Research Results

4.1 Top Factors Driving Integration

4.1.1 Integration Factors

Hypothesis 1 was "Faulty requirements, faulty programming logic, and interface issues are the top factors that impact sensor integration." To evaluate whether these factors are significant, each of these factors were considered separately against a 10% repair hours threshold. The null hypothesis H_0 assumed there were no primary drivers for integration, which would mean that none of the factors had a less than 10% probability of occurrence p. The significance level was $\alpha < 0.05$ per Fisher [62]. The alternative hypothesis was that each of the three factors in Hypothesis 1 were greater than 10%.

Significance level = 0.05

H_0: $p_0 = 10\%$

H_a: $p > 10\%$

The variable p was the proportion of repairs hours associated with the integration driver to the total number of repair hours. n was the number of repair hour samples for each factor. For the normal model to be used to find the P-values, the following conditions had to be true: $np_0 \geq 10$ and $n(1-p_0) \geq 10$.

n = 21689 samples

$p_0 = 0.1$

$n(p_0) = 2.169*10^3$

$n(p_0) \geq 10$ = true

$n(1-p_0) = 1.952*10^4$

$n(1-p_0) \geq 10$ = true

Because both conditions are true, the normal model can be used to find the P-values.

The z-score of a sample proportion is:

$$Z = \frac{\hat{p} - p}{\sqrt{\frac{p(1-p)}{n}}}$$

Factor 1: Faulty requirements:

Faulty requirements made up 20.46% of the sampled repair hours.

$$\hat{p}_1 := 20.46\%$$

$$Z_1 := \frac{\hat{p}_1 - p_0}{\sqrt{\frac{p_0 \cdot (1-p_0)}{n}}} = 51.349$$

$$1 - \text{cnorm}(Z_1) = 0$$

Since 0 < α, the null hypothesis is rejected in favor of the hypothesis that faulty requirements are a significant driver for integration.

Factor 2: Faulty programming logic:

Faulty programming logic made up 23.21% of the sampled repair hours.

$$\hat{p}_2 := 23.21\%$$

$$Z_2 := \frac{\hat{p}_2 - p_0}{\sqrt{\frac{p_0 \cdot (1-p_0)}{n}}} = 64.849$$

$$1 - \text{cnorm}(Z_2) = 0$$

Since 0 < α, the null hypothesis is rejected in favor of the hypothesis that faulty programming logic is a significant driver for integration.

Factor 3: Interface issues:

Interface issues made up 13.27% of the sampled repair hours.

$$\hat{p}_3 := 13.27\%$$

$$Z_3 := \frac{\hat{p}_3 - p_0}{\sqrt{\frac{p_0 \cdot (1 - p_0)}{n}}} = 16.053$$

$$1 - \text{cnorm}(Z_3) = 0$$

Since 0 < α, the null hypothesis was rejected in favor of the hypothesis that interface issues were a significant driver for integration.

Each of the categories in Hypothesis 1 were shown to be statistically significant drivers for integration. The very low 1 – cnorm(Z_X) values indicated that a very high number of samples were collected and the difference between the null hypothesis and the actual proportions for the defects was large and highly significant.

4.1.2 Z-Score per Defect Cause Code

The following table shows the total repair hours for the defect cause codes and their associated Z-Scores. The breakout by cause code shows Z-Scores greater than zero for the top three cause codes and less than zero for the remainder of the cause codes. A1 was contained in the faulty requirements category. E2 was contained in the faulty programming logic category, and E4 was contained in the interface issues category.

Defect Cause Code	Z-Score
Administrative - A1: Unrevised SRS updates	22.62152077
Engineering Error - E2: Bad program logic introduces error	20.49165125
Engineering Error - E4: Interface issues	0.863353137
Engineering Error - E2F: General coding	-15.2679754
Test - T5: Cannot Duplicate	-17.6895985
Engineering Error - E2E: Faulty logic or sequence	-18.8351107

Administrative - A5: Not a Defect	-21.6720698
Engineering Error - E1: Code does not meet specifications	-24.9101052
Administrative - A2: Unrevised Design Documents	-30.5980564
Engineering Error - E1B: Faulty software requirements	-31.8429323
Administrative - A4: Unrevised ICD Updates	-33.880002
Engineering Error - E1A: Faulty systems requirements	-37.6033127
Test - T1: Test Procedure Error	-37.8022665
Product Definition - D1: Code improvement	-39.3212415
Engineering Error - E2B: Faulty state management	-39.6290654
Product Definition - D3: Premature code inclusion	-40.6143281
Engineering Error - E3: Merge / integration introduces error	-42.6371383
Program / Customer Directed - P1: Prog/Sched directed change	-42.718621
Administrative - A6: Duplicate Defect	-42.9170995
Test - T4: Lab Set Up/Installation Error	-43.0524741
Test - T2: Lab Hardware Failure	-44.3140993
Engineering Error - E1C: Faulty software detailed design	-44.5180326
Administrative - A7: Legacy Defect	-44.6991055
Program / Customer Directed - P2: Non-SRS requirements change	-45.7855426
Engineering Error - E2D: Faulty error handling	-46.1929566
Engineering Error - E5: Open Defect / Bug Dependency	-46.2810032
Engineering Error - E2C: Faulty data	-48.0715875
Product Definition - D2: Test workaround added	-48.2051287
Engineering Error - E2A: Faulty timing	-48.3545138
Administrative - A3: Tools	-48.3656046

Table 5: Z-Scores per Defect Cause Code

4.2 Deterministic Environments

4.2.1 Training Rewards

The following graphs show results of training the agents in each of the different environments. The X-axes show the training steps, or iterations, and the Y-axes represent the cumulative reward received by the agent over the course of the training steps. A straight slope over training reflects that the agents were not getting any better at making decisions in the environment. A convex slope reflects that learning was taking place and the agent was getting better at receiving rewards. Only results from the 32,768-step

trainings are shown in Figure 27, Figure 28, Figure 29, Figure 31 and Figure 32 for comparing algorithm performance in each environment. Figure 30 shows the 262,144-step trainings for the Optical Environment. The tests with the highest number of training steps converged to the most optimal solutions for 22 out of 25 of the test configurations as shown in Table 6 through Table 10. One exception was for IBL in the Dungeon environment as shown in Table 6, where the algorithm recommended leaving the Dungeon versus exploring the Dungeon more frequently as the training steps increased. The other two exceptions are for the Deep Q-Learning and Double Deep Q-Learning environments for the 262,144-step training on the Optical environment, where the RAM requirements exceeded the capacity of the cloud server, and the tests were unable to complete. The Q-learning agent's learning rate $\alpha = 0.9$ resulted in the highest average accuracies across all the test cases with the different learning rates and is the Q-learning case shown in the figures.

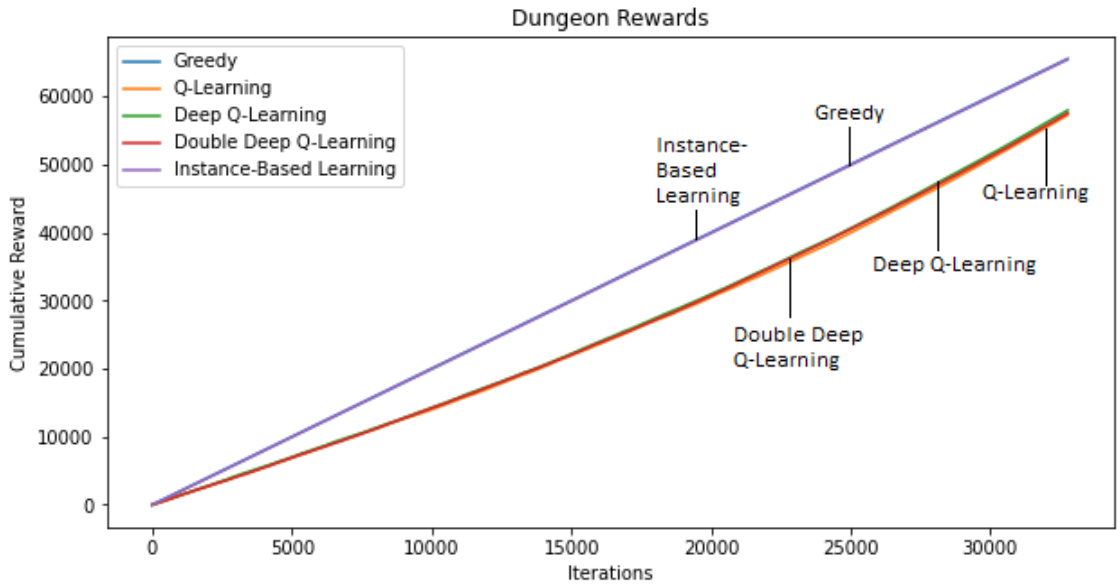

Figure 27: Training in Dungeon Environment

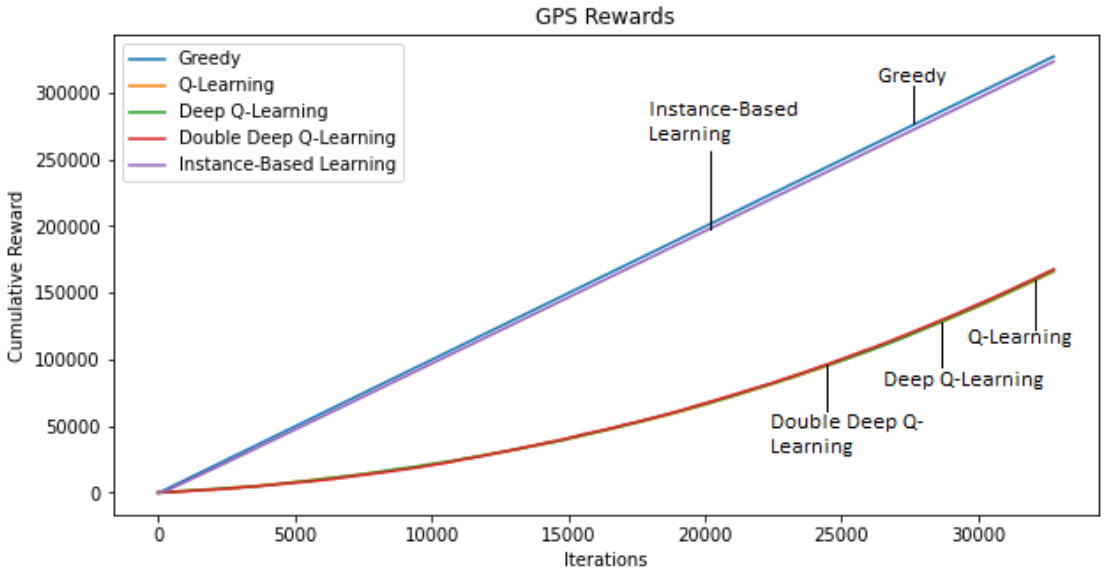

Figure 28: Training in GPS Environment

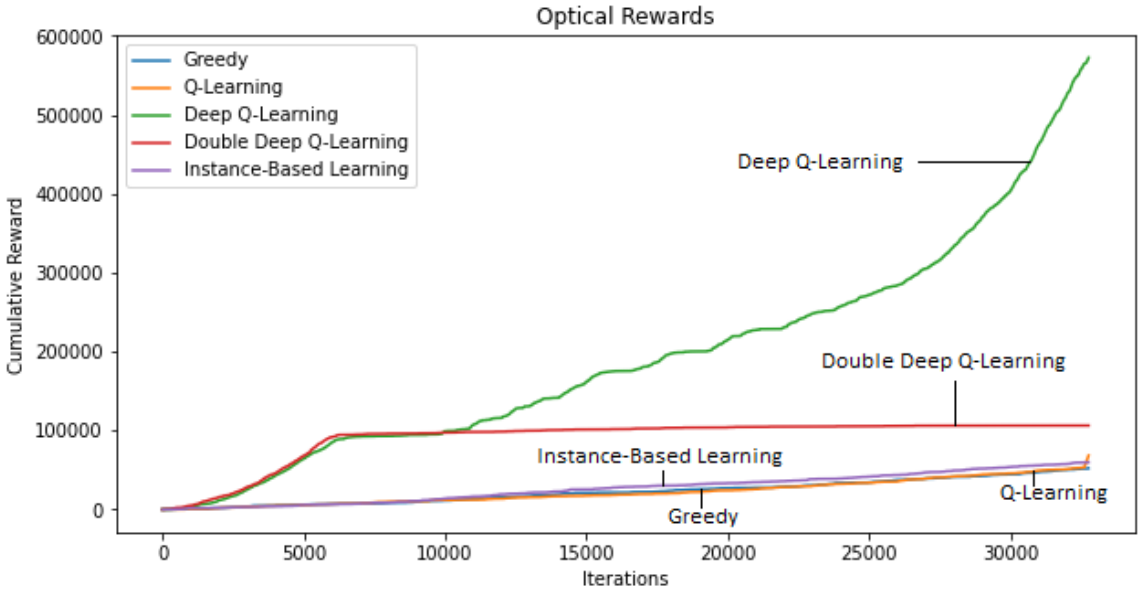

Figure 29: 32,768-Step Training in Optical Environment

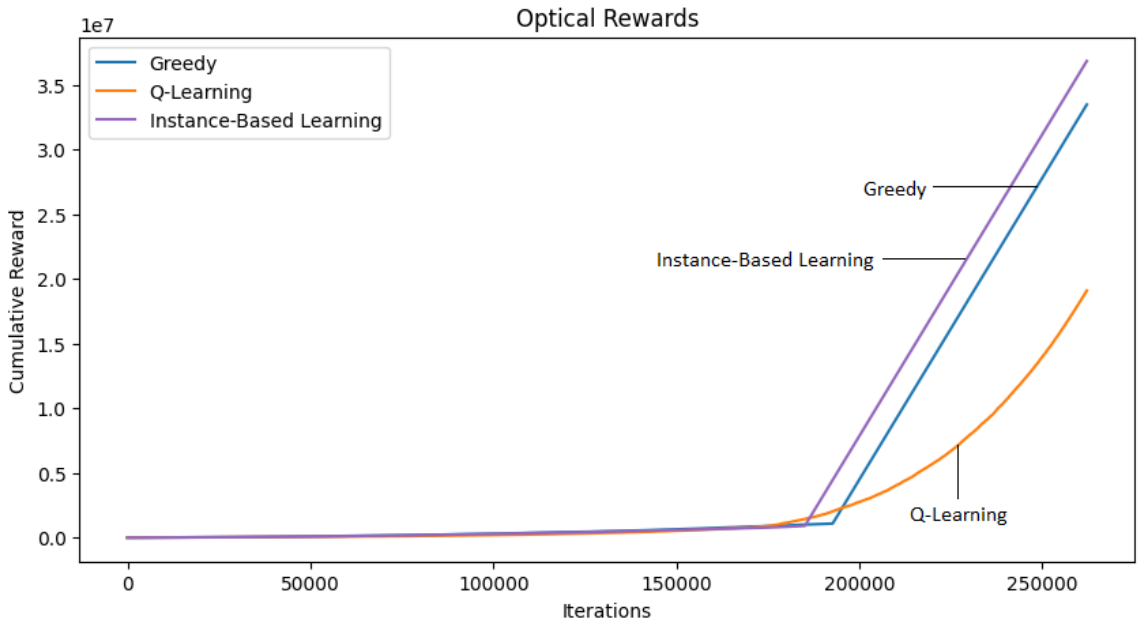

Figure 30: 262,144-Step Training in Optical Environment

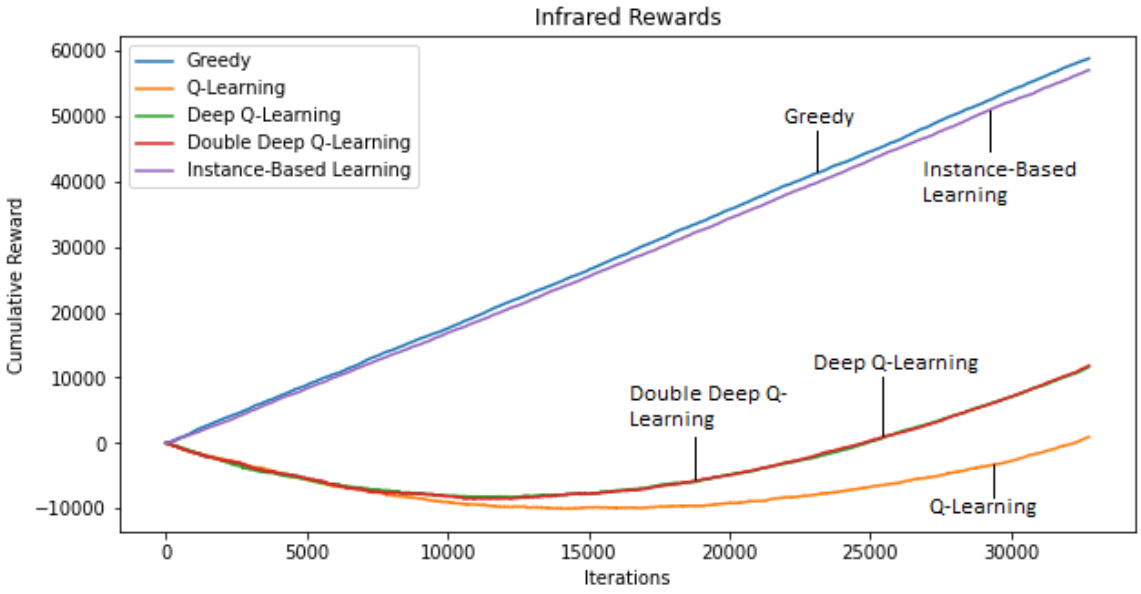

Figure 31: Training in Infrared Environment

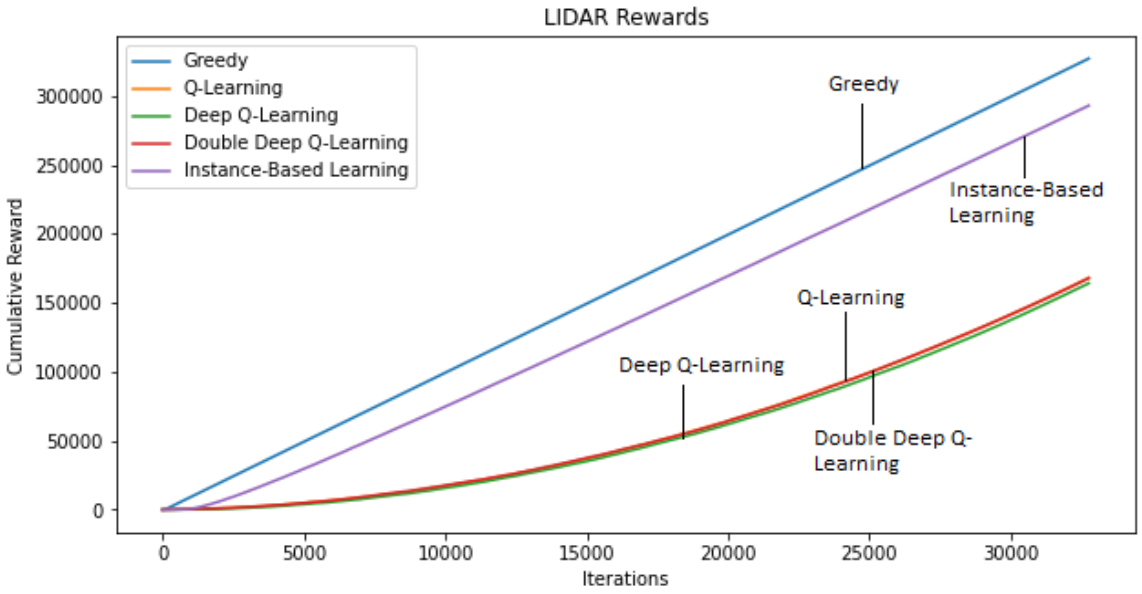

Figure 32: Training in LIDAR Environment

Figure 27, Figure 28, Figure 31, and Figure 32 all show the same training trends with Greedy and IBL algorithms in the upper straight-line cluster, and Q-Learning, Deep Q-Learning, and Double Deep Q-Learning in the lower convex curve cluster. Figure 29 shows slow learning rates for Greedy, Q-Learning, and IBL and rapid learning rates for Deep Q-Learning and Double Deep Q-Learning. Double Deep Q-Learning stopped learning between 5,000 and 6,000 iterations. Q-Learning had a learning rate spike towards the end of the training sequence. Figure 30 shows slow learning and then rapid learning for the IBL and Greedy algorithms, while Q-Learning gradually finds more optimal solutions as it moves through its training.

4.2.2 Test Accuracy

After the agent trainings completed, each of the trained agents were run against each of the sensor model environments for 1,000 steps and their performance was recorded in Table 6 through Table 10. Each table shows the accuracy of the agents tested against a specific sensor model environment. Five different versions of each algorithm

were trained, each with the number of training steps listed in the table, and their accuracy during testing is included in the tables.

	Accuracy % at 128 steps	Accuracy % at 512 steps	Accuracy % at 2,048 steps	Accuracy % at 8,192 steps	Accuracy % at 32,768 steps
Greedy	0	0	0	0	0
Q-Learning @ 0.9 Learning Rate	0	100	100	100	100
Deep Q-Learning	0	100	100	100	100
Double Deep Q-Learning	0	100	100	100	100
Speedy IBL	2.8	1.5	0.9	0.1	0

Table 6: Dungeon Environment Accuracy Results

	Accuracy % at 128 steps	Accuracy % at 512 steps	Accuracy % at 2,048 steps	Accuracy % at 8,192 steps	Accuracy % at 32,768 steps
Greedy	100	100	100	100	100
Q-Learning @ 0.9 Learning Rate	50	100	100	100	100
Deep Q-Learning	100	100	100	100	100
Double Deep Q-Learning	50	100	100	100	100
Speedy IBL	74.2	88.8	98.5	99.3	99.9

Table 7: GPS Environment Accuracy Results

	Accuracy % at 128 steps	Accuracy % at 512 steps	Accuracy % at 2,048 steps	Accuracy % at 8,192 steps	Accuracy % at 32,768 steps	Accuracy % at 262,144 steps
Greedy	0.5	0.4	0.5	0.9	1.5	100
Q-Learning @ 0.9 Learning Rate	0.7	0.3	0.6	0.8	31.7	100
Deep Q-Learning	0.1	0.1	0.1	0	50	N/A
Double Deep Q-Learning	0	0.1	0.1	0	0	N/A
Speedy IBL	0.9	0.5	0.8	0.7	1	100

Table 8: Optical Environment Accuracy Results

	Accuracy % at 128 steps	Accuracy % at 512 steps	Accuracy % at 2,048 steps	Accuracy % at 8,192 steps	Accuracy % at 32,768 steps
Greedy	100	100	99.9	100	100
Q-Learning @ 0.9 Learning Rate	100	95	100	100	100
Deep Q-Learning	99.9	100	100	100	100
Double Deep Q-Learning	100	100	99.9	100	100
Speedy IBL	87.3	95.8	98.6	99.7	100

Table 9: Infrared Environment Accuracy Results

	Accuracy % at 128 steps	Accuracy % at 512 steps	Accuracy % at 2,048 steps	Accuracy % at 8,192 steps	Accuracy % at 32,768 steps
Greedy	100	100	100	100	100
Q-Learning @ 0.9 Learning Rate	50	50	100	100	100
Deep Q-Learning	0.1	0.1	100	100	100
Double Deep Q-Learning	0.1	50	100	100	100
Speedy IBL	0.1	1.5	68.3	90.6	97.5

Table 10: LIDAR Environment Accuracy Results

4.2.3 Discussion

In the Dungeon environment, the Greedy and IBL algorithms did not experience any learning as demonstrated by the straight slope in Figure 27. They trained to leave the Dungeon out the front for a small, short-term reward and did not find that it was better to move through the Dungeon to the end. Each of the other algorithms has a curve that shows they were learning and becoming more efficient over time. The Q-learning, Deep Q-Learning, and Double Deep Q-Learning agents that were trained for more than 512 steps were equally successful in learning the correct actions as shown in Table 6.

Figure 28 shows the GPS environment had the same trends in the cumulative reward curves. Greedy and IBL did not have an improvement in the rate of reward receipt, while the other algorithms showed a learning curve. A main difference between the Dungeon and the GPS environment was that with the GPS environment, all the algorithms except for Instance-Based Learning were successful at learning the correct functional behavior as shown in Table 7. The model rewarded the optimal actions with the highest rewards that could be achieved for each training step. There was no need for exploration to find higher rewards. The implication for applications of reinforcement learning to sensor integration on aircraft is that a Greedy algorithm may be good enough to train sensor controller policies if sensor behavior is simple and deterministic. In these cases, during the sensor design phase, vendors can plan rewards that specifically target exploitative algorithms in legacy sensor controller training.

The agents had different performances in the Optical environment. Table 8 shows that none of the algorithms were effectively able to learn the interfaces by 32,768 training steps because each fell short of the 100% objective threshold for accuracy. This was a particularly hard model because the one-degree sampling of the Pan and Tilt actions decreased the probability that the Collect Image discrete action would be randomly selected. However, Q-Learning and Deep Q-Learning partially learned the correct functional behavior with 31.7% accuracy and 50% accuracy respectively. In Figure 29, Q-Learning occurred very slowly until nearly the end of the training when its efficiency increased. The Deep Q-Learning agent was able to continually improve but did not learn the optimal functional behavior. The Double Deep Q-Learning agent was making learning progress for around 5,000 steps. However, in step 5,242, it panned and tilted to

the correct angles and took an image for a very high reward, but then the rate of learning began to decrease. Because Q-Learning is guaranteed to converge to optimum action-values provided that the actions are repeatedly sampled and represented discretely [63], tests were also run with 262,144 training steps, which generated viable agent policies. The cloud environment for the experiment allocated 81 GB of memory to the virtual machine, and the deep learning algorithms each consumed the available memory, causing the tests to fail. Future research should address neural network optimization and scalable deployment methods to address greater degrees of complexity. Regardless, the other algorithms were successful in finding optimal policies with the most complex model, showing that this sensor integration method can scale to higher complexity sensor integrations.

Another implication of the Optical results is that they illustrate how efficient reinforcement learning can be versus a brute force approach or simple automata. There are 11,892,243 possible state-action combinations in the optical model and reinforcement learning algorithms learned the optimal behaviors in 262,144 training steps.

Table 9 shows the results for the Infrared environment, in which all agents reached 100% accuracy, very similar to the results from the GPS environment. Figure 31 shows cumulative rewards going negative because the sensor model environment penalized false indications of detected objects with negative reward values. While the model was being trained and exploring, it was trying state-action combinations that resulted in negative rewards, driving cumulative rewards below zero before recovering as it learned optimal state-action combinations. Despite the Infrared environment being a stochastic environment, all algorithms were able to achieve the 100% threshold.

All of the agents were able to meet the 100% accuracy requirement on the LIDAR environment as shown in Table 10, with the exception of the Instance-Based Learning agent. The LIDAR sensor model had a complex action set, but each agent was able to learn the appropriate data baud rates and serial ports to use to connect to the TF-Luna Module model.

Some of the agents had similar training performance as shown by clusters of lines in the cumulative reward graphs. The IBL agents were clustered with the Greedy agents, which indicated that exploration did not happen in the algorithm implementation as expected. Deep Q-Learning and Double Deep Q-Learning had similar training performance in most environments but diverged in training performance on the Optical environment due to the susceptibility to timing of random exploratory discoveries. Because Deep Q-Learning and Double Deep Q-Learning had such similar results, Q-value overestimation, which Double Deep Q-Learning is designed to address, was not a factor for agent accuracy.

To determine success or failure, accuracy needed to be at 100%. However, high accuracy in machine learning algorithms can indicate overfitting and a lack of generalizability. However, in these scenarios, the training models were identical to the test models and generalizability was not a desired characteristic. In the cases where 100% accuracy was achieved, the sensor controllers learned the optimal policies. If the sensor models that are available during policy development are fully representative of operational sensors, finding the optimal policy and achieving 100% test accuracy would be the desired outcome.

Using reinforcement learning for integration of legacy sensors has a few limitations. First, it requires software simulation models to train controllers. Practically, these would need to be provided by sensor vendors to sensor integrators ahead of or along with the sensors to be integrated. Another limitation is that rewards need to be explicitly provided by the models, rather than being derived by the controller. Controllers assigning rewards to state transitions would increase complexity, lower accuracy, and require manual programming of a priori knowledge of the sensor behavior. A third limitation is that the approach assumes that interface syntax has been standardized and that actions are discrete and finite. Lastly, sensors may support multiple valid paths of actions and state transitions to achieve a desired state, but the reinforcement learning algorithms will learn the optimal sequences and exclude the sub-optimal sequences, which may still be valid use cases.

There are a couple practical issues that have to be overcome when implementing this approach on real systems. Reinforcement learning algorithms, and deep learning algorithms in particular, can be processor intensive. Lengthy training with complex models can take days in desktop or simple cloud computing environments. Implementers will need to plan for scalable production-grade cloud computing infrastructure for training complex models. Secondly, if vendor sensor models do not accurately represent the actual sensor, then the trained sensor controller may have defects. Using reinforcement learning to build sensor controllers does not alleviate the need for verification and validation processes after integration is completed. Finally, sensor controllers have to support multiple threads of behavior, and the research evaluated

developing single policies for single behavioral threads. Practitioners should develop sensor controller software that can train and execute multiple threads.

4.3 Stochastic Environments

4.3.1 Training Rewards

Figure 33 through Figure 38 show the results of training the agents against the stochastic variations of the environments.

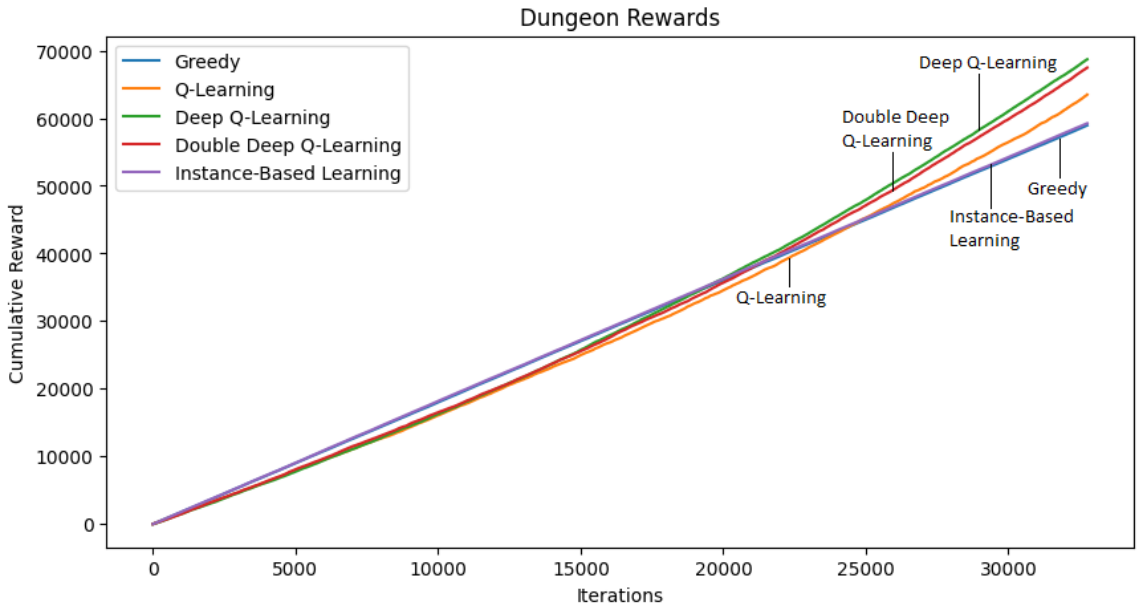

Figure 33: Training in Stochastic Dungeon Environment

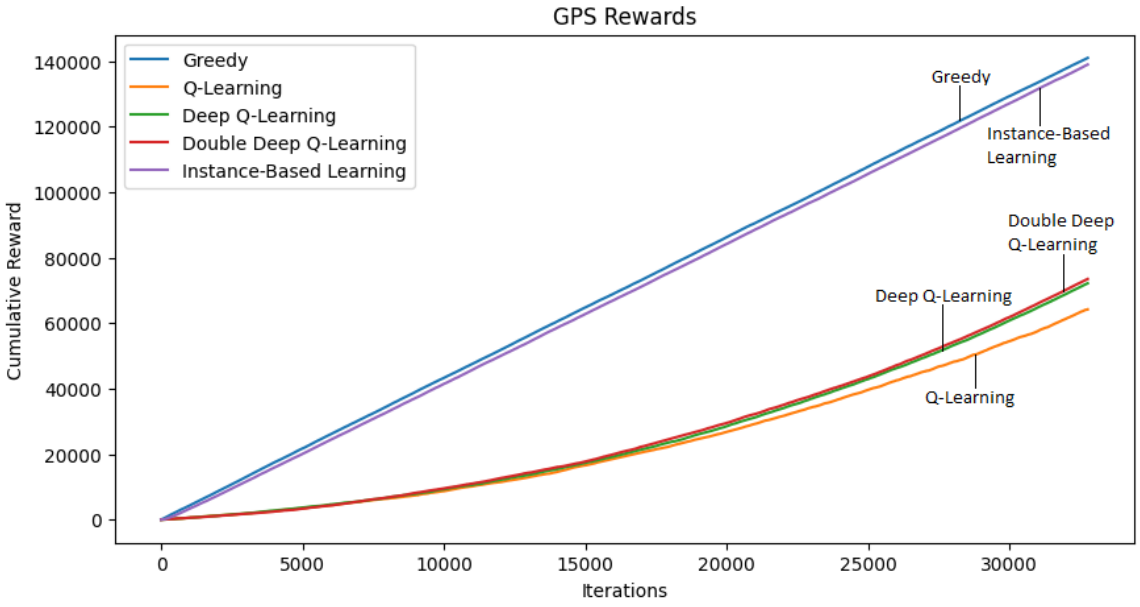

Figure 34: Training in Stochastic GPS Environment

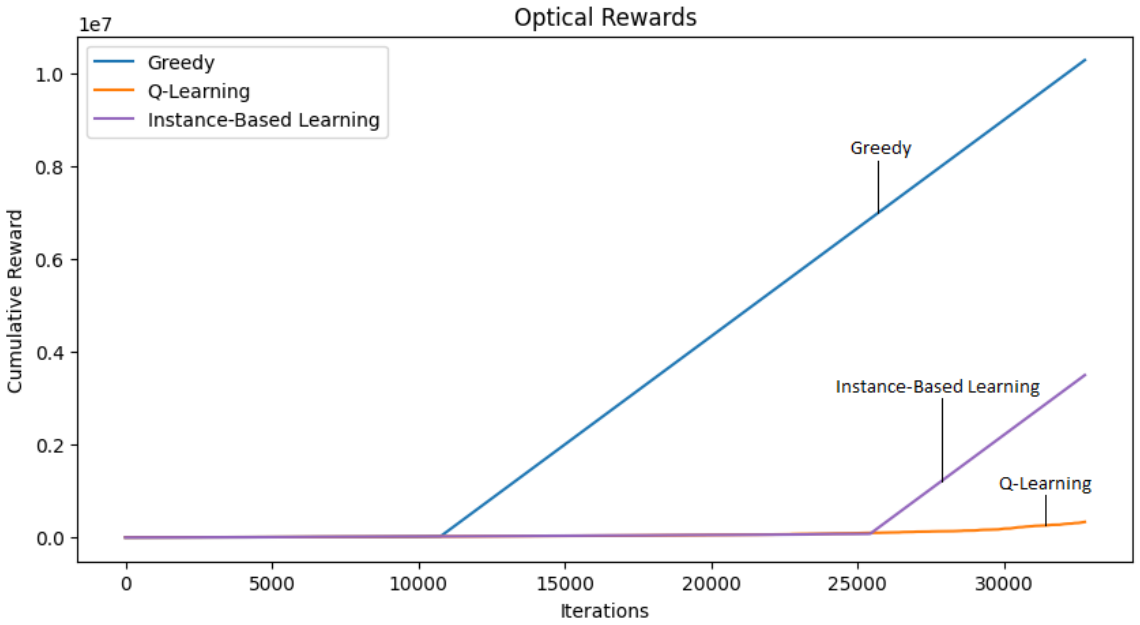

Figure 35: 32,768-Step Training in Stochastic Optical Environment

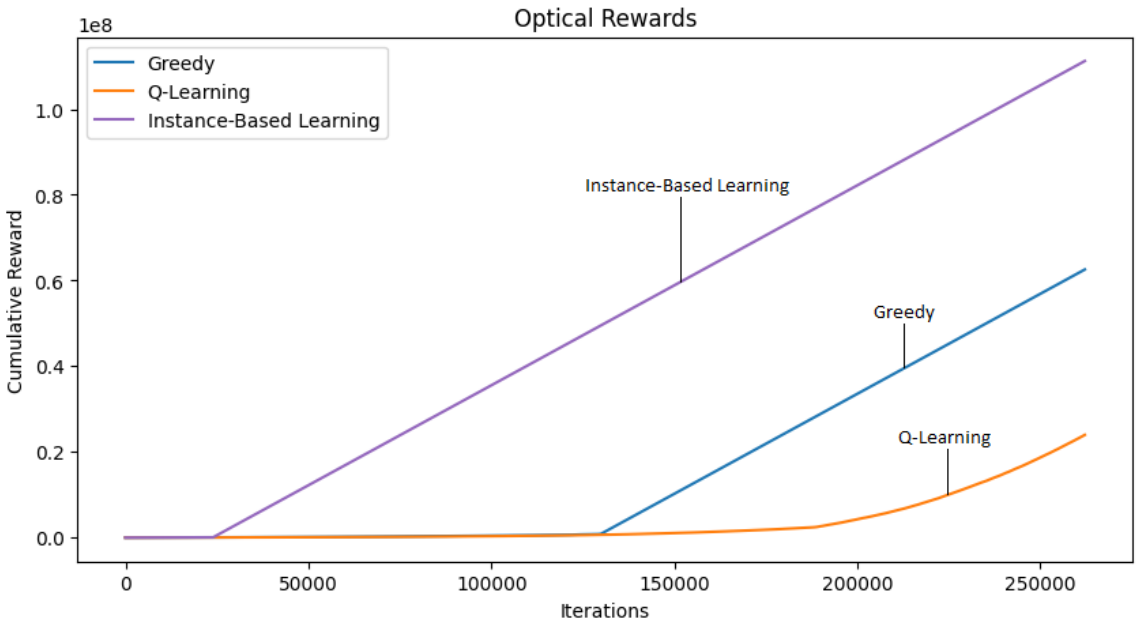

Figure 36: 262,144-Step Training in Stochastic Optical Environment

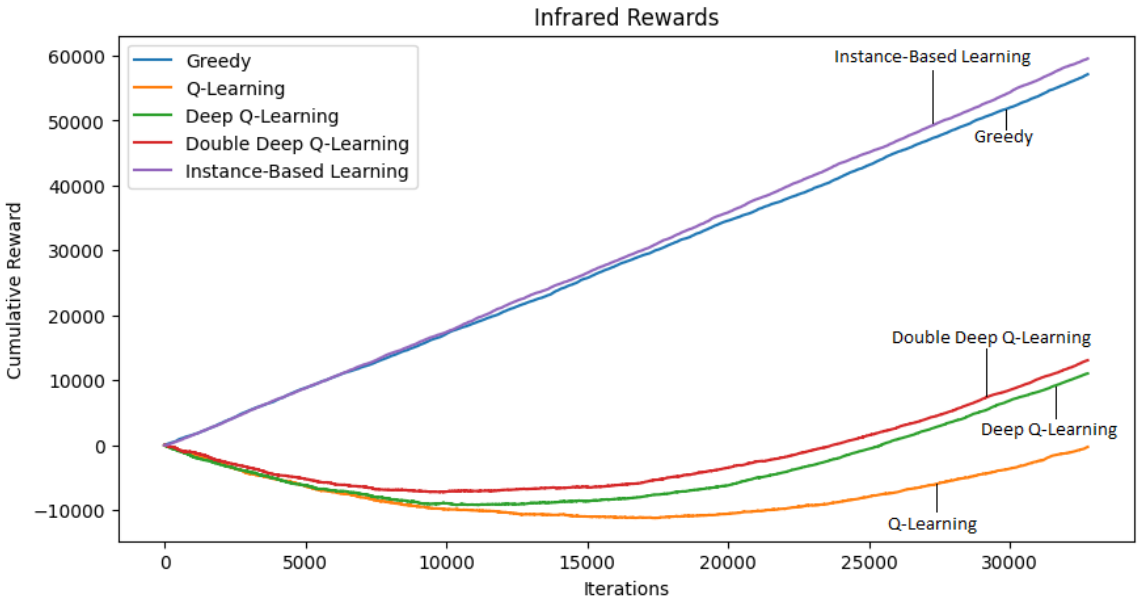

Figure 37: Training in Stochastic Infrared Environment

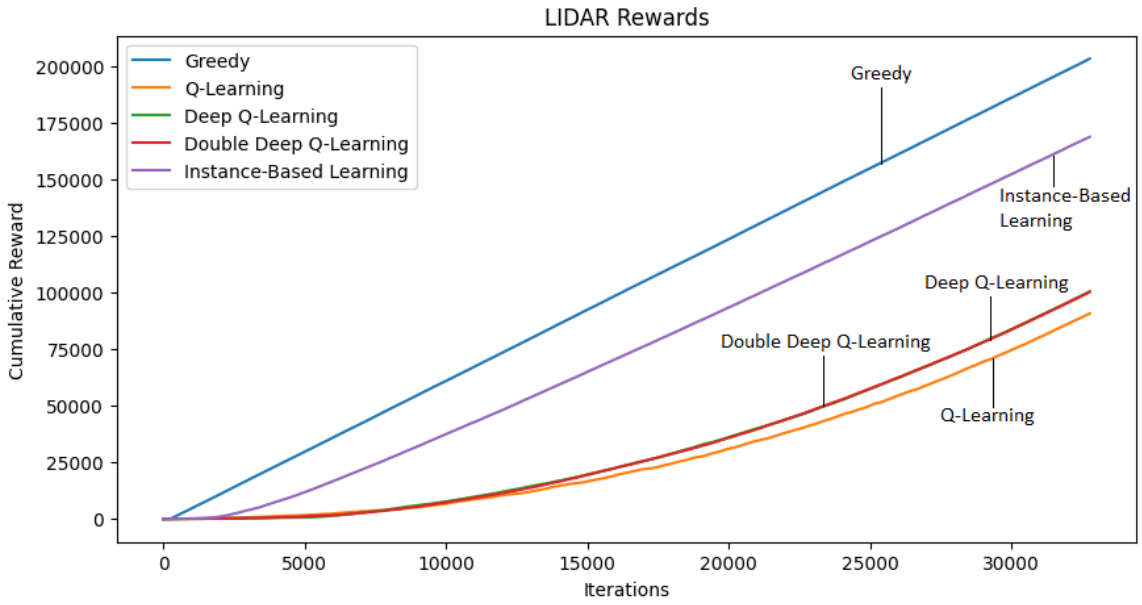

Figure 38: Training in Stochastic LIDAR Environment

Figure 33, Figure 34,

Figure 37, and Figure 38 included all the environments except for Optical, and they all showed similar trends. Deep Q-Learning and Double Deep Q-Learning had similar learning rates and were clustered together for each of the environments. Q-Learning had a consistent increase in learning but lagged the neural network-based algorithms in each case. Greedy and Instance-Based Learning had linear reward gain throughout training with the exception of LIDAR in Figure 38, where Instance-Based Learning had a reward rate increase early in the training, around 2,000 iterations. Q-Learning, Deep Q-Learning, and Double Deep Q-Learning all had higher rewards than Greedy and Instance-Based, except for Dungeon, which was specifically designed to promote exploration over exploitation.

Optical training results shown in

Figure 35 and Figure 36 were similar to each other but had unique trends from the other environments. Both showed no rewards occurring until discrete points when learning would take place. In

Figure 35, the Greedy algorithm started gaining rewards ahead of IBL, which gained rewards ahead of Q-Learning. In Figure 36, IBL gained rewards ahead of Greedy, which gained rewards ahead of Q-Learning. In both Optical training cases shown, deep learning algorithms failed to complete training against the test environment, so cumulative rewards were not recorded.

4.3.2 Test Accuracy

Table 11 through Table 15 show the results of testing the agents against the stochastic variations of the environments. The agents and trials that were run match the first experimentation round with the original environments.

	Accuracy % at 128 steps	Accuracy % at 512 steps	Accuracy % at 2,048 steps	Accuracy % at 8,192 steps	Accuracy % at 32,768 steps
Greedy	0.56	8.4	0.07	0.14	0
Q-Learning @ 0.9 Learning Rate	25.13	2.25	100	100	100
Deep Q-Learning	27.05	100	26.41	100	100
Double Deep Q-Learning	0.02	26.04	100	100	100
Speedy IBL	4.95	2.23	0.7	0.25	0.24

Table 11: Stochastic Dungeon Environment Accuracy Results

	Accuracy % at 128 steps	Accuracy % at 512 steps	Accuracy % at 2,048 steps	Accuracy % at 8,192 steps	Accuracy % at 32,768 steps
Greedy	35.77	100	100	100	100
Q-Learning @ 0.9 Learning Rate	52.46	100	100	100	100
Deep Q-Learning	52.54	100	100	100	100
Double Deep Q-Learning	100	100	100	100	100

Speedy IBL	61.47	86.62	96.93	99.31	99.66

Table 12: Stochastic GPS Environment Accuracy Results

	Accuracy % at 128 steps	Accuracy % at 512 steps	Accuracy % at 2,048 steps	Accuracy % at 8,192 steps	Accuracy % at 32,768 steps	Accuracy % at 262,144 steps
Greedy	0.67	0.72	0.93	1.11	100	100
Q-Learning @ 0.9 Learning Rate	1.02	0.92	50	14.9	50	100
Deep Q-Learning	0.01	0.01	0.01	0.01	N/A	N/A
Double Deep Q-Learning	0.01	0.01	0.01	0	N/A	N/A
Speedy IBL	0.86	0.7	1.58	1.28	100	100

Table 13: Stochastic Optical Environment Accuracy Results

	Accuracy % at 128 steps	Accuracy % at 512 steps	Accuracy % at 2,048 steps	Accuracy % at 8,192 steps	Accuracy % at 32,768 steps
Greedy	100	100	100	100	100
Q-Learning @ 0.9 Learning Rate	99.99	100	100	100	100
Deep Q-Learning	100	100	100	100	100
Double Deep Q-Learning	100	100	100	100	100
Speedy IBL	76.6	97.98	98.87	99.59	99.9

Table 14: Stochastic Infrared Environment Accuracy Results

	Accuracy % at 128 steps	Accuracy % at 512 steps	Accuracy % at 2,048 steps	Accuracy % at 8,192 steps	Accuracy % at 32,768 steps
Greedy	10.31	31.39	100	100	100
Q-Learning @ 0.9 Learning Rate	9.79	50	66.67	93.16	100
Deep Q-Learning	0	0	50	66.67	100
Double Deep Q-Learning	0	0	0.01	66.67	100

Speedy IBL	0	0.51	21.14	83.2	95.58

Table 15: Stochastic LIDAR Environment Accuracy Results

Q-Learning, Deep Q-Learning and IBL algorithms had successful results against the Dungeon, GPS, and Infrared environments with each reaching 100% accuracy by the 32,768-step trial. The Greedy algorithm was also effective against GPS and Infrared environments. IBL was not as effective against GPS and Infrared, but accuracy exceeded 99% for both. Greedy and IBL were both ineffective against the Dungeon environment.

The Optical environment had several notable results. First is that neural network trials above 8,192 were not able to execute. In the first round of experiments, the 32,768-step trial was able to run, but the 262,144-step trial was not able to complete. The second round of experiments for the stochastic environments could not complete the 32,768-step trial or the 262,144-step trial. This was due to a reduction in available RAM. Colab Pro decreased the RAM from 81GB to 51 GB for the second round of tests, despite them being run with a high RAM configuration. The other surprising result was that Greedy and IBL again were able to solve the Optical case, even at the lower 32,768 step trial.

All the algorithms were able to solve LIDAR by 32,768 except for IBL, which had accuracy above 95%. LIDAR had a 20% probability of hardware failure which was a higher probability of stochastic behavior than other environments. An earlier iteration of LIDAR would fail because even if the controller selected the correct action, a hardware defect would result in no rewards being gained for the read action, resulting in unintended actions being chosen after serial open was successful for subsequent steps. A reward was added for correct behavior regardless of failure conditions, and the LIDAR behavior was successfully learned.

4.3.3 Discussion

Greedy was 100% accurate after 32,768 steps on all the environments except for Dungeon, both in deterministic and stochastic configurations. It could often find optimal solutions faster because it was able to get stuck on the next correct step and then randomly guess until the next correct step was identified. This was in contrast to the Bellman-based algorithms that would explore away from the known rewards to explore a potential future reward. Greedy was effective for finding sequentially correct actions because it would get stuck repeating a sequence that provided the highest value action up to a known state and then would try random actions until it got another reward, which would then add the next correct action and state transition to the sequence.

Q-Learning, Deep Q-Learning, and Double Deep Q-Learning were 100% accurate after 32,768 steps for all environments, except for Optical where the neural networks did not have enough RAM to finish executing. IBL was 100% accurate in some of the models. Since it is a classification algorithm, it may need more training steps than the other algorithms before it can recommend controller actions with 100% accuracy. IBL accuracy increased as steps increased with the exception of Dungeon, and it was always 100% effective at the 262,144-step trials.

When moving from the deterministic environments to the stochastic environments, there had to be a change in the rewards. The previous strategy of incentivizing the completion of a behavioral thread with a larger reward could get interrupted if stochastic behavior like a simulated hardware failure occurred. The long-term incentives had to be increased for the algorithms to be sufficiently rewarded for exploration.

For the algorithms to be successful, the reward structure had to be applied to the correct action being applied by the controller, even if the environment produced undesirable stochastic behavior. Initial implementations of the stochastic environments were not able to be learned due to only rewarding the successful environment transitions. Even in failure conditions, agents should be positively rewarded for taking the correct action prior to the failure as well as the subsequent recovery actions.

Even in stochastic environments where actions can result in multiple states, these machine learning algorithm implementations were effective in finding the correct sensor controller behavior.

4.4 Research Hypotheses Results

Hypothesis Number	Hypothesis	Supported
H1	Faulty requirements, faulty programming logic, and interface issues are the top factors that impact sensor integration.	Yes
H2	Rewarding correct subsystem interfaces through reinforcement learning algorithms results in the correct subsystem behavior on 100% of the test cases.	Partial
H3	Training sensor controllers using reinforcement learning algorithms in stochastic environments results in correct controller actions to dynamic environmental conditions on 100% of the test cases.	Partial

Table 16: Research Hypotheses Results

H1: Faulty requirements, faulty programming logic, and interface issues are the top factors that impact sensor integration.

Result: Hypothesis 1 is supported by the results.

Rationale: Results show the defects found during integration for a major acquisition program and their associated repair hours that extended the length of

integration. Faulty requirements, faulty programming logic, and interface issues were each associated with a set of defect cause codes, and each were associated with greater than 10% of the integration repair hours. Results were shown to be statistically significant based on the large amount of total repair hours recorded and the difference between the 10% threshold and the actual value of the repairs hour for each category.

H2: Rewarding correct subsystem interfaces through reinforcement learning algorithms results in the correct subsystem behavior on 100% of the test cases.

Result: Partial

Rationale: Q-Learning, Deep Q-Learning, and Double Deep Q-Learning were able to learn all sensor behavior tested after the largest number of training steps. Greedy was able to learn all sensor behavior except for Dungeon, which was designed to penalize exploitative algorithms. IBL was able to achieve increasing performance for all sensor environments except for Dungeon, but IBL had not yet achieved 100% accuracy with the number of training steps evaluated.

H3: Training sensor controllers using reinforcement learning algorithms in stochastic environments results in correct controller actions to dynamic environmental conditions on 100% of the test cases.

Result: Partial

Rationale: Q-Learning, Deep Q-Learning, and Double Deep Q-Learning were able to learn stochastic sensor behavior tested after the largest number of training steps. Greedy was able to learn all the stochastic sensor behavior except for Dungeon. IBL was able to achieve increasing performance for all sensor environments except for Dungeon, but IBL had not yet achieved 100% accuracy with the number of training steps evaluated.

4.5 Research Questions Results

Research Question Number	Research Question	Answered by Research	Supporting Hypotheses
Q1	What are the top factors that impact sensor interoperability?	Yes	H1
Q2	What are the best algorithms to apply to machine learning of sensor behavior?	Yes	H2
Q3	What are the top methods for accounting for dynamic aircraft and sensor environments for machine learning-based sensor controllers?	Yes	H3

Table 17: Research Questions Results

Q1: What are the top factors that impact sensor interoperability?

Result: Answered by the research

Rationale: The research identified faulty requirements, faulty programming logic, and interface issues as the top factors that impact sensor integration based on a statistically significant sampling of repair hours for integration defects for a major defense acquisition program integrating a set of sensors.

Q2: What are the best algorithms to apply to machine learning of sensor behavior?

Result: Answered by the research

Rationale: Greedy, Q-Learning, Deep Q-Learning, and Double Deep Q-Learning algorithms were shown to be effective at learning sensor behavior due to 100% accurate performance against modeled GPS, LIDAR, Optical, and Infrared sensor functions. IBL was not shown to be a sufficiently effective algorithm implementation.

Q3: What are the top methods for accounting for dynamic aircraft and sensor environments for machine learning-based sensor controllers?

Result: Answered by the research

Rationale: Greedy, Q-Learning, Deep Q-Learning, and Double Deep Q-Learning algorithms were shown to be effective at learning dynamic sensor behavior due to 100% accurate performance against modeled GPS, LIDAR, Optical, and Infrared sensor functions with stochastic elements modeled as Markov Decision Processes. IBL was not shown to be a sufficiently effective algorithm implementation.

Chapter 5 – Conclusions and Recommendations

5.1 Conclusion

An analysis of the integration defect data from the major defense program showed that faulty requirements, faulty programming logic, and interface issues were the top drivers impacting integration of sensors. These issues can be mitigated by the machine learning integration approach evaluated in the dissertation. Rather than specifications for behavioral controller requirements, models can be provided that provide rewards based on the intent of the sensor vendor. Behavior for sensor controllers do not have to be as rigidly specified in requirements when following a machine learning-based integration approach. Faulty programming logic includes behavioral sequences, application-level state management, and application-level fault handling. These are all OSI Layer 7 behaviors that can be learned using a machine learning-based approach. While not all programming logic errors can be eliminated, behavioral logic can be trained by models and remove sources of error related to manual human-in-the-loop software coding. Lastly interface issues can be mitigated using machine learning. Training controllers with models rather than having humans read and translate from paper ICDs to software logic mitigates errors arising from interface issues. The results from Research Question 1 are specifically addressed and mitigated by the results from Research Questions 2 and 3.

Training behavioral logic depends on underlying standards being in place. Open systems architectures like Open Mission Systems or FACE are vital because behavioral training evaluated in the dissertation only provides logic at the application layer of the OSI model. The results from the research fill an important gap for sensor integration where behavioral logic is not specified by open standards but can now be trained by

machine learning, achieving a “plug and play” paradigm, provided that sensor models are developed and delivered for training controllers.

The results from these experiments show that machine learning algorithms can be used for integrating sensors onto aircraft, specifically to learn functional behavior, which is normally defined by vendor-specific interface specifications. Where today, messages and data structures are standardized and functional integration happens manually, this research shows that reinforcement learning could train sensor controller policies to sequence and structure those messages into higher level functional behavior. Instead of human-readable interface documentation for functional behavior, sensor vendors can provide software simulation models of their sensors that provide actions, state information, and rewards for sensor controllers taking correct actions. Aircraft sensor integrators can use these sensor models to train sensor controller policies. Due to large numbers of trial and error, training with software models would be more efficient than with hardware in the loop, and training would be less prone to damaging hardware. The policies that are trained can either be incorporated during the design and development phases or loaded and utilized during operational phases as new sensors become available. This application has benefits across the systems development lifecycle. In design, policies are created by machine learning algorithms. The development phase becomes focused on building sensor controllers that consume trained policies rather than implementing specific functional behaviors. Integration timelines are reduced due to earlier training of the controllers against sensor models, and since human-readable functional descriptions are replaced by machine readable policies, testing will find fewer defects. Verification for leveraged components remains key to integration success based

on Grumbach and Thomas [64]. However, the ability to rapidly replace and upgrade sensors improves system obsolescence and decreases time to field for new sensor capabilities. MOSA-aligned standards management bodies such as OMS, FACE, and SOSA, should incorporate provisions for providing sets of actions, rewards, and state information. Sensor vendors should make sensor models available along with their sensors so that they can be used for sensor controller development and training. When these are in place, legacy aircraft sensor integrators can have the necessary data to build reinforcement learning-based controllers that adapt to sensors with different functional behaviors.

The results from this study contribute to the systems engineering field because it is more broadly applicable than sensors on legacy aircraft. While the specific experiments evaluated performance of reinforcement learning algorithms against sensor models, the approach can be extended to integrate other systems or subsystems modeled by discrete-time finite Markov Decision Processes. The extensibility of the implementation of the action list as function pointers that are selectable by agents further opens the potential for use in other applications. If a system can be modeled with a set of actions and states, providing rewards and state information to a controller, then the same approach can be applied.

5.2 Summary of Research Objectives

Objective Number	Research Objective	Result	Supporting Research Questions
O1	Decrease sensor integration timelines on legacy aircraft.	Met	Q1, Q2, Q3

O2	Remove the interoperability limitation of proprietary behavioral specifications.	Met	Q2, Q3
O3	Provide algorithm recommendations for fault-tolerant, robust sensor controllers.	Met	Q2, Q3

O1: Decrease sensor integration timelines on legacy aircraft.

Result: Met

Rationale: The research has decreased the timeline for sensor integration. Behavior for sensor controllers has been shown to be trainable, removing the need for manual development of controllers implementing sensor functions. The top drivers for integration have been mitigated using machine learning approaches. Future efforts that are scaling and applying the research to physical sensor systems will continue to mature and improve machine learning-based sensor integration, but the research opens new mechanisms for integration.

O2: Remove the interoperability limitation of proprietary behavioral specifications.

Result: Met

Rationale: Vendor-specific sensor behavior has been shown to be learnable, provided that models for the sensors are provided and that underlying layers of the OSI model are standardized.

O3: Provide algorithm recommendations for fault-tolerant, robust sensor controllers.

Result: Met

Rationale: Machine learning algorithms were able to handle stochastic environments that accounted for simulated hardware failures, software failures, and external conditions. Reinforcement learning agents can be effectively trained to handle faults. This was particularly shown in the testing for Hypothesis 3.

5.3 Research Limitations and Future Work

First, future research needs to evaluate the effectiveness of the approach in production environments. While reinforcement learning can be used for more complex environments as shown in the Optical environment, models need to be able to handle even more complex threads with higher numbers of states and actions, including training in production-equivalent environments.

While IBL was not shown to be effective at learning sensor behavior, it is anticipated that additional training steps would have produced sufficiently accurate IBL agents. Future work should evaluate IBL with larger numbers of training steps.

Development also needs to occur to tie multiple sensor functions together in single controllers. Sensor functions were trained and tested individually but production sensor controllers will need to be able to select from a set of policies based on the desired sensor function to be executed. This could be accomplished with a meta-model, treating each trained policy as an action.

Hyperparameter optimization at scale is also going to be an important factor for future work. High-capacity machine learning environments have the processing capability to test with larger sets of hyperparameters such as learning rates, and evaluating and selecting the optimum hyperparameters for the models is important for

performance. Understanding the right shape of the neural networks including number of layers and number of neurons per layer will also be important.

Associated with hyperparameters, learning rate decay is another approach to evaluate with reinforcement learning algorithms. One of the expectations for algorithms that could handle stochastic environments was that their accuracy would improve if they had the ability to "forget" past learned experiences, resulting in more generalizable solutions. In fact, using decay rate as an independent variable was originally the plan for testing Hypothesis 3. It was thought that a separate decay rate that erased a small fraction of Q-values periodically would result in a better solution that would be more accurate in stochastic environments. The following Q-Learning algorithm based on the Bellman equation shows the associated variables [49].

$$Q^{new}(s_t,a_t) = Q(s_t,a_t) + \alpha * (r_t + \gamma * \max_a Q(s_{t+1}, a) - Q(s_t,a_t))$$

As a decay rate was being explored and as evident in examination of the equation above, learning rate α determines how much of the old Q-value information would be overwritten by new Q-value estimates. Because a separate decay rate would provide a redundant effect on the impact of previously learned Q-values, there should not be a separate decay rate from the learning rate. However, further research found that there have been positive results from a learning rate decay rate in neural networks, resulting in improved optimization and generalization [65]. According to You, et al., learning rate decay starts with a large learning rate and decays it multiple times, which is effective because it keeps the neural network from learning noisy data initially with large learning rates but then enables learning complex patterns with the smaller learning rates. It took days of execution in the Google Colab environment to complete the tests that were

performed for the experiments for the dynamic and stochastic environments. Varying learning decay rates for deep learning models would take a substantially longer amount of time and would be a practical constraint without moving to a production AI environment. Future research should evaluate the effectiveness in a production environment of learning rate decay with both deep and traditional reinforcement learning algorithms on modeled sensor functions with stochastic behavior.

Future research should also explore alternatives to the utilized reinforcement learning reward strategies. It may be effective to penalize ineffective actions with a floor-is-lava type approach to encourage faster achievements of correct action-state combinations. It might also be more efficient to alter the epsilon-greedy approach and establish a preference to explore unexplored action-state combinations rather than randomly select from the superset of possible actions.

The dissertation discussed that requirements specification was potentially reduced based on use of machine learning-based sensor controllers. However, requirements verification of machine-learning based sensor controllers has not yet been addressed. Flight critical, safety critical, and cyber hardened systems may have issues with behavior logic being defined by policies rather than explicitly coded, and research needs to address what vulnerabilities may exist in these types of software applications and how they can be mitigated.

One factor that was not considered in the sensor controller agent evaluation was time and computing resources required for training. The number of steps for training was used as the independent variable, but that does not correlate to time for training. The neural network algorithm implementations took substantially more time and computing

resources than each of the other algorithm implementations. This may be due to how the agents were programmed, but use of these algorithms at scale will require additional attention to resource allocation and optimization.

Future research should explore whether Double Deep Q-Learning with soft updates between the target model and the prediction model is effective in preventing over estimation of Q-values. Performance between Double Deep Q-Learning and Deep Q-Learning was consistently similar across all tests performed. The environments chosen may not have been vulnerable to Q-value estimation, so similar performance could be expected.

The value of neural networks over basic reinforcement learning algorithms like Greedy and Q-Learning algorithms was not evident with the environments tested. Future research in more complex production environments may show that neural networks are superior to basic algorithms due to their ability to handle complexity, or the models may be built simply enough that basic algorithms are sufficient at scale.

Future research should evaluate effectiveness of reinforcement learning for non-linear systems, where an action taken in a state changes the expected result or utility from an action taken in a different state. For example, perhaps a sensor mode change executed as part of a thread changes the message sequence required for future mode changes that would return it to that state.

State space representation is another potential research area, partially explored in the Optical environment. Future work should provide both continuous and discrete inputs into the algorithms as well as binary inputs. This may simplify the environment complexity. The same need is true for actions. Agents should be able to select the best

value for an analog action as well as the best value for a discrete action. If actions represented by sampled analog values outnumber actions represented by discrete values, then exploratory algorithms are less likely to select the outnumbered discrete values. This may also be accomplished by breaking a single thread with a single policy up into multiple policies to execute a single behavioral thread.

Expanding the use of machine learning lower than OSI Layer 7 would expand the application set beyond systems where data elements have already been standardized. The current research does not support data transformation, and modeling, including data modeling, may be required to relate inputs to outputs in ways that can be sequenced and executed by machine learning algorithms. Using machine learning approaches in conjunction with data modeling to automatically map ontologies from one domain to the other and generate data transformations would be beneficial to a wider set of sensor integration applications. Large language models may also be able to process ICDs and recommend sequences of messages and data inputs or even generate reinforcement learning models to use for training controllers. Furthermore, models may be derived and reverse-engineered from large sets of existing subsystem test data.

Future work includes testing the integration approach for other types of aircraft subsystems, such as radars, displays, and communications subsystems, including verifying the approach using physical hardware rather than modeled sensor functions. Testing the approach for other types of systems beyond aircraft is also planned, specifically with ground vehicles. The approach for integrating sensors onto aircraft may also be applicable to integrating aircraft behaviors across mission threads in Systems of

Systems contexts. Finding optimal ways to achieve mission targets using potential actions from a set of assets in theater could be addressed using reinforcement learning.

Although there is a large set of potential future work and research, the dissertation provides initial evidence that reinforcement learning can address the top factors that impact sensor interoperability, ultimately enabling faster and more affordable delivery of sensor capabilities to the field, improving survivability of warfighters.

References

[1] M. Pywell, "A question of survival - military aircraft vs the electromagnetic environment," *The Aeronautical Journal,* vol. 108, no. 1087, pp. 453-464, September 2004.

[2] J. Enos, "Apply real system age to DOD Systems," *Systems Engineering,* vol. 25, no. 3, pp. 242-253, 2022.

[3] J. Mattis, "Summary of the 2018 National Defense Strategy of the United States of America," Department of Defense, Washington D.C., 2018.

[4] L. Austin, "2022 National Defense Strategy," Department of Defense , Washington, D.C., 2022.

[5] C. Pope, "Kendall Details 'Seven Operational Imperatives' & How They Forge the Future Force," 3 March 2022. [Online]. Available: https://www.af.mil/News/Article-Display/Article/2953552/kendall-details-seven-operational-imperatives-how-they-forge-the-future-force/. [Accessed 19 April 2023].

[6] U.S. GAO, "James Webb Space Telescope: Project nearing completion, but work to resolve challenges continues. James Webb Space Telescope: Project Nearing Completion, but Work to Resolve Challenges Continues," 13 May 2021. [Online]. Available: https://www.gao.gov/products/gao-21-406. [Accessed 19 April 2023].

[7] F. Kendall, "Better Buying Power 3.0," 9 April 2015. [Online]. Available: https://www.acq.osd.mil/fo/docs/betterBuyingPower3.0(9Apr15).pdf. [Accessed 20 April 2023].

[8] Office of the Under Secretary of Defense, Research and Engineering, "Modular Open Systems Approach (MOSA) Reference Frameworks in Defense Acquisition Programs," Washington, D.C., 2020.

[9] The Open Group, "Future Airborne Capability Environment™ (FACE)," 19 January 2023. [Online]. Available: https://www.opengroup.org/face. [Accessed 19 April 2023].

[10] United States Air Force, "Open Mission Systems (OMS)," 30 June 2022. [Online]. Available: https://www.vdl.afrl.af.mil/programs/oam/oms.php. [Accessed 19 April 2023].

[11] P. Shah, T. Eveleigh and T. Holzer, "Modular Open Systems Approach (MOSA) and its Impact on the US Defense Acquisition Methods and Programs," The George Washington University, Washington, D.C., 2021.

[12] The Open Group, "Technical Standard for SOSA™ Reference Architecture, Edition 1.0," 29 September 2021. [Online]. Available: https://publications.opengroup.org/standards/sosa/c212. [Accessed 19 April 2023].

[13] United States Air Force, "OMS in a Nutshell," 30 June 2022. [Online]. Available: https://www.vdl.afrl.af.mil/programs/oam/OMS_Marketing.pdf. [Accessed 29 June 2023].

[14] J. R. Hoehn, "Joint All-Domain Command and Control (JADC2)," Congressional Research Service , 2022.

[15] H. Eisner, Thinking: A Guide to Systems Engineering Problem-Solving, Boca Raton: CRC Press, 2019.

[16] B. Sauser, M. Long, E. Forbes and S. McGrory, "Defining an Integration Readiness Level for Defense Acquisition," *INCOSE International Symposium,* vol. 19, no. 1, pp. 352-367, 2009.

[17] J. Goldschmid and S. Corns, "A cluster-based framework for Interface Analysis in large-scale Aerospace Systems," *Systems Engineering,* vol. 24, no. 5, pp. 339-351, 2021.

[18] M. Morgan, T. Holzer and T. Eveleigh, "Synergizing model-based systems engineering, modularity, and software container concepts to manage obsolescence," *Systems Engineering,* vol. 24, no. 5, pp. 369-380, 2021.

[19] G. JL and T. LD, "Integration Principles for Complex Systems," *Systems Engineering,* vol. 23, no. 6, pp. 684-706, 2020.

[20] G. JL and T. LD, "Quantitative validation of Complex Systems Integration principles," *Systems Engineering,* vol. 26, no. 1, pp. 32-47, 2022.

[21] S. Armistead, "DSIAC Technical Inquiry (TI) Response Report: Weapon System Integration Cost Analysis," Defense Systems Information Analysis Center, Belcamp, MD, 2016.

[22] Cost Analysis & Assurance Service, Defence Equipment and Support, "Weapons Integration Cost Modelling Presentation to SCAF," UK Ministry of Defence, London, 2013.

[23] M. Sullivan, "GAO-15-188: Better Approach Needed to Account for Number, Cost, and Performance of Non-Major Programs," United States Government Accountability Office, Washington, D.C., 2015.

[24] J. Norton, "Enhancing Interoperability through Open Systems Architecture," *Defense Standardization Program Journal,* Vols. October-December, pp. 11-16, 2015.

[25] K. Shepherd and J. Wills, "Avionics Open Systems Architecture Standardization," in *AHS Inteernation 74th Annual Forum & Technology Display*, Phoenix, AZ, 2018.

[26] P. Zimmerman, M. Ofori, J. Soler, D. Barrett and A. Harriman, "Considerations and examples of a modular open systems approach in defense systems," *Journal of Defense Modeling and Simulation: Applications, Methodology, Technology,* vol. 16, no. 4, pp. 373-388, 2019.

[27] G. C. Sargent, C. Collier and I. Lipkin, "Designing the next generation of sensor systems using the SOSA standard," in *SPIE Defense + Commercial Sensing Conference*, Baltimore, MD, 2019.

[28] T. DuBois, S. Wiggington, M. Orlovsky and W. Kinaham, "Benefits and Limitations of Reliance on an Open Architecture Technical Standard to Meet Expectations of an Open System," in *Proceedings of the 73rd Annual AHS International Forum and Technology Display*, 2017.

[29] G. Kayas, M. Hossain, J. Payton and S. M. R. Islam, "SUPnP: Secure Access and Service Registration for UPnP-Enabled Internet of Things," *IEEE Internet of Things Journal,* vol. 8, no. 14, pp. 11561-11580, 2021.

[30] M. Sellami, P. De Vettor, M. Mrissa, D. Benslimane and B. Defude, "DMaaS: Syntactic, structural and semantic mediation for service composition," *International Journal of Autonomous and Adaptive Communications Systems,* vol. 9, no. 3-4, pp. 310-330, 2016.

[31] S. Bowers, "An Ontology-Driven Framework for Data Transformation in Scientific Workflows," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics),* vol. 2994, pp. 1-16, 2004.

[32] L. Yang, K. Cormican and M. Yu, "Learning systems engineering domain ontologies from text documents," in *2019 International Symposium on Systems Engineering (ISSE),* Edinburgh, UK, 2019.

[33] M. Maree and M. Belkhatir, "A coupled statistical/semantic framework for merging heterogeneous domain-specific ontologies," in *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, Arras, France, 2010.

[34] M. Asfand-E-Yar and R. Ali, "Semantic Integration of Heterogeneous," *IEEE Access*, vol. 8, pp. 77903-77919, 2020.

[35] M. M. Taye, "Understanding Semantic Web and," *Journal of Computing*, vol. 2, no. 6, pp. 182-192, 2010.

[36] M. Lian, D. Yang, Z. Yin, J. Liu, M. Li and A. Chai, "Research and Design of Semantic Model of IoT Entity Based on Ontology," in *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, Chengdu, China, 2020.

[37] J. Lu, L. T. Yang, B. Guo, Q. Li, H. Su, G. Li and J. Tang, "A Sustainable Solution for IoT Semantic Interoperability: Dataspaces Model via Distributed Approaches," *IEEE Internet of Things Journal*, vol. 9, no. 10, pp. 7228-7242, 2022.

[38] Q. Li, H. Wei, C. Yu and S. Wang, "Data and model-based triple V product development framework and methodology," *Enterprise Information Systems*, vol. 16, no. 5, 2022.

[39] D. Bellack, J. Dillon and T. Hart, "Enabling System Interoperability," *Defense Standardization Program Journal*, pp. 17-26, 2013.

[40] M. Abdolshah, H. Le, T. K. George, S. Gupta, S. Rana and S. Venkatesh, "Plug and Play, Model-Based Reinforcement Learning," *arXiv preprint arXiv*, vol. 2108, no. 08960, pp. 1-9, 2021.

[41] G. A, R. PP and D. V, "Evaluation of instance-based learning and Q-learning algorithms in Dynamic Environments," *IEEE Access*, vol. 9, pp. 138775-138790, 2021.

[42] T. N. Nguyen and C. Gonzalez, "Effects of Decision Complexity in Goal seeking Gridworlds: A Comparison of Instance Based Learning and Reinforcement Learning Agents," Defense Technical Information Center, Fort Belvoir, VA, 2020.

[43] C. Chen, P. X. B. Chen, J. Sun, Z. Xing, X. Wang and W. Zhao, "“More Than Deep Learning”: post-processing for API sequence recommendation," *Empirical Software Engineering*, vol. 27, no. 1, 2022.

[44] X. Gu, H. Zhang, D. Zhang and S. Kim, "Deep API Learning," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016.

[45] Boeing, "MCAS. Boeing 737 MAX Updates," [Online]. Available: https://www.boeing.com/737-max-updates/mcas. [Accessed 19 April 2023].

[46] M. Wiering and M. Otterlo, "Reinforcement Learning and Markov Decision Processes," in *Reinforcement Learning State-of-the-Art*, Berlin, Springer, 2012, pp. 3-42.

[47] J. Stanger, "Innovating and Problem Solving with the Open Systems Interconnection Reference Model (OSI/RM)," CompTIA, 6 August 2021. [Online]. Available: https://www.comptia.org/blog/open-systems-interconnection-reference-model. [Accessed 26 May 2023].

[48] J. Kiili, "Valohai/qlearning-simple: For the tutorial blogpost," GitHub, 2 May 2020. [Online]. Available: https://github.com/valohai/qlearning-simple. [Accessed 19 April 2023].

[49] F. Melo, "Convergence of Q-learning: A simple proof," Institute of Systems and Robotics, Lisboa, Portugal, 2001.

[50] B. DP and T. JN, Neuro-Dynamic Programming, Belmont, MA: Athena Scientific, 1996.

[51] R. Balsys, "Reinforcement_learning/02_cartpole-reinforcement-learning_ddqn," GitHub, 7 October 2020. [Online]. Available: https://github.com/pythonlessons/Reinforcement_Learning/tree/master/02_CartPole-reinforcement-learning_DDQN. [Accessed 19 April 2023].

[52] C. Gonzalez, J. Lerch and C. Lebiere, "Instance-based learning in Dynamic Decision making," *Cognitive Science,* vol. 27, no. 4, pp. 591-635, 2003.

[53] T. Nguyen, D. Phan and G. C, "Speedyibl: A comprehensive, precise, and fast implementation of instance-based learning theory," *Behavior Research Methods,* 2022.

[54] Nhatpd, "Speedy implementation of instance-based learning (IBL) agents in python," 6 April 2022. [Online]. Available: https://github.com/nhatpd/speedyibl. [Accessed 19 April 2023].

[55] GPS.gov, "Aviation Applications," 10 February 2022. [Online]. Available: https://www.gps.gov/applications/aviation/. [Accessed 19 April 2023].

[56] "About gpsd," Gpsd — a GPS Service Daemon, 5 April 2023. [Online]. Available: https://gpsd.gitlab.io/gpsd. [Accessed 19 April 2023].

[57] National Marine Electronics Association, "NMEA 0183," 2021. [Online]. Available: https://www.nmea.org/nmea-0183.html. [Accessed 19 April 2023].

[58] Pimoroni, "Python library for the Pimoroni Pantilt Servo and lighting hat," GitHub, 23 September 2022. [Online]. Available: https://github.com/pimoroni/pantilt-hat. [Accessed 19 April 2023].

[59] Raspberry Pi, "Camera.," [Online]. Available: https://www.raspberrypi.com/documentation/accessories/camera.html. [Accessed 19 April 2023].

[60] A. Raj, "Raspberry Pi Infrared (IR) Sensor Interfacing Tutorial," Circuit Digest, 14 September 2017. [Online]. Available: https://circuitdigest.com/microcontroller-projects/raspberry-pi-ir-sensor-tutorial. [Accessed 19 April 2023].

[61] Seeed Studio, "Product Manual of TF-Luna," [Online]. Available: https://files.seeedstudio.com/wiki/Grove-TF_Mini_LiDAR/res/SJ-PM-TF-Luna-A03-Product-Manual.pdf. [Accessed 20 April 2023].

[62] L. Kennedy-Shaffer, "Before $p < 0.05$ to Beyond $p < 0.05$: Using History to Contextualize p-Values and Significance Testing," *The American Statistician* , vol. 73, pp. 82-90, 2019.

[63] C. Watkins and P. Dayan, "Q-learning," *Machine Learning,* vol. 8, no. 1, pp. 279-292, 1992.

[64] J. Grumbach and T. LD, "Systems integration implications of component reuse," *Systems Engineering,* vol. 25, no. 6, pp. 561-573, 2022.

[65] K. You, M. Long, J. Wang and M. I. Jordan, "How Does Learning Rate Decay Help Modern Neural Networks?," *arXiv preprint arXiv,* vol. 1908, no. 01878, pp. 1-15, 2019.