A HEALTH-AWARE REPLANNING FRAMEWORK FOR UNMANNED AERIAL VEHICLES IN STOCHASTIC ENVIRONMENTS

By

Timothy Darrah

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

December 16, 2023

Nashville, Tennessee

Approved

Gautam Biswas, Ph.D.

Jeremy Frank, Ph.D.

Akos Ledeczi, Ph.D.

Gabor Karsai, Ph.D.

Xenofon Koutsoukos, PhD.

Copyright © 2023 Timothy S. Darrah

All Rights Reserved

Acknowledgements

I would like to thank Dr. Gautam Biswas, my thesis advisor, for his immense support and guidance over the last six years. His mentorship has been invaluable, and without it I would not be where I am today. I would like to thank Dr. Jeremy Frank, my NASA advisor, for his support, insight, and flexibility throughout my NASA fellowship. Collaborating with him has been a reward and a life changing experience, and I am forever grateful to him for this opportunity. I would like to thank Dr. Marcos Quiñones Gruiero for always being available when I needed someone to talk to and providing valuable direction and feedback where I needed it. I would like to thank my thesis committee for holding the line and keeping me accountable to delivering high quality research. Finally, I would like to thank my wife and daughter for their grace and patience, dealing with my long nights, absent days, and all the time I spent away. This is for them.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . . **viii**

LIST OF FIGURES . . . . **x**

1 Introduction **1**

1.1 Motivation 3

1.2 Overview of Approach 4

1.3 Challenges & Contributions 5

1.3.1 Challenges 5

1.3.2 Contributions 7

1.4 Organization of Thesis 8

2 Literature Review **9**

2.1 System-Level Prognostics for Cyber-Physical Systems 9

2.1.1 Data-Driven Prognostics with Deep Learning 12

2.1.1.1 Deep Learning Approaches 13

2.2 Model Compression 16

2.2.1 Pruning Recurrent Networks 18

2.2.2 Explainable-Informed Pruning 21

2.2.3 Quantization 23

2.3 Replanning 25

2.3.1 Fault-Aware Replanning 27

3 Overview of Approach **30**

3.1 Simulation & Data Generation 30

3.2 System-Level Prognostics 31

3.3 Model Compression 31

3.4 Health-Aware Replanning 32

4 Simulation Testbed and Data Generation . . . 33

4.1 System Description . . . 36

4.1.1 Airframe Dynamics . . . 38

4.1.2 Battery Dynamics . . . 39

4.1.3 Motor Dynamics . . . 41

4.1.4 System Performance Parameters . . . 43

4.1.5 Trajectory Generation . . . 44

4.2 Data Management Framework . . . 46

4.2.1 Requirements . . . 46

4.2.2 Assets . . . 48

4.2.3 Processes . . . 51

4.2.4 Data . . . 53

4.3 System-Level Prognostics Dataset . . . 54

4.3.1 Flight Data . . . 57

5 Data-Driven System-Level Prognostics . . . 61

5.1 Deep Learning Model Development . . . 62

5.1.1 Feature Selection & Engineering for Prognostics . . . 63

5.1.2 Hyperparameter Optimization . . . 66

5.1.3 Training Procedure . . . 70

5.2 Training Results . . . 74

5.3 Auxiliary Neural Networks . . . 75

5.4 Summary . . . 77

6 Model Reduction for Long-Short Term Memory Networks . . . 78

6.1 Understanding the Behavior of an LSTM-based Prognostics Model . . . 79

6.1.1 Improved Saliency Maps . . . 80

6.1.2 Neural Activation Patterns . . . 82

6.2 Compression for LSTM-based Prognostics Models . . . 83

6.2.1 Experiment Overview . . . 85

6.3 Results . . . 87

6.3.1 Results on Desktop Computer . . . 89

6.3.2 Results on the Embedded Hardware 90
6.3.3 Summary 92

7 Health-Aware Replanning 93

7.1 Health Awareness 93
7.1.1 Replanning Framework 94
7.2 Replanning as a Modified Orienteering Problem 96
7.2.1 Solutions to The Orienteering Problem 97
7.2.2 Genetic Algorithms 99
7.3 Fast-Start Adaptive Genetic Algorithm for Constrained Routing with Rewards 100
7.3.1 Population Initialization 101
7.3.2 Calculate Target Fitness 102
7.3.3 Calculate Candidate Fitness 103
7.3.4 Crossover & Mutation 103
7.3.5 Population Diversity 105
7.3.6 Adaptive Rates 106
7.3.7 The Genetic Algorithm 107
7.4 Experiments 109
7.4.1 Demonstration of Health Awareness with Uncompressed Model & A* 110
7.4.2 Demonstration of Compressed Models with Fast-Start GA 111
7.4.3 GA Performance Analysis 112
7.5 Results & Discussion 113
7.5.1 GA Performance on Embedded Hardware 113
7.5.2 Replanning Scenarios with Compressed RUL Model 116
7.5.3 Solution Variability & Quality 118
7.5.4 Summary 119

8 Conclusions & Future Work 121

8.1 Limitations & Future Work 123

9 Appendix 125

9.1 UAV Telemetry Data 125
9.2 Additional Replanning Scenarios 127

9.2.1 Scenario A. 127

9.2.2 Scenario B. 128

9.2.3 Scenario C. 129

9.2.4 Scenario D. 130

9.2.5 Scenario E. 130

9.2.6 Scenario F. 131

9.2.7 Scenario G. 132

9.2.8 Scenario H. 134

9.3 Experimental Demonstration of GA Performance 135

References 137

LIST OF TABLES

Table Page

4.1 Data sources for stochastic simulation . . . 34

4.2 Dynamical variables of the system . . . 36

4.3 Airframe Parameters . . . 39

4.4 Battery Parameters . . . 40

4.5 Motor Parameters . . . 41

4.6 System performance parameters . . . 43

4.7 asset-types . . . 49

4.8 Asset table for an octorotor UAV system. . . . 50

4.9 Process Type Table . . . 52

4.10 Summary statistics for the system-level prognostics dataset. . . . 55

4.11 UAV Flight Data . . . 56

5.1 Model Search Parameters . . . 69

5.2 Bayesian Optimization Parameters . . . 69

5.3 RUL model parameters found through Bayesian optimization. . . . 70

7.1 GA search parameters. . . . 108

7.2 System degradation parameters . . . 110

7.3 Top GA performance results on desktop computer when controlling for time and reward such that running time is less than 5 seconds and reward is greater than or equal to 41. . . . 115

7.4 Top GA performance results on embedded when controlling for reward only, such that the reward is greater than or equal to 41. The running time on the embedded hardware is prohibitive for online use. . . . 115

7.5 Replanning results. [1]Represents the average of 8 motors. [2]Times column represents (a) the flight time required to finish the original trajectory after the fault occured; (b) the actual flight time of the UAV after the fault until failure; and (c) the flight time of the new trajectory that results in a safe landing at base. [3] Plan did not result in a safe return to base. [4] Same scenario but with the original uncompressed RUL estimation model resulting in a safe return to base. . . . 117

9.1 Telemetry data sampled at 1Hz. . . . 125

9.2 Top fastest results for solutions with 37 or greater reward for scenario A. 128
9.3 Top fastest results for solutions with 24 or greater reward for scenario B. 129
9.4 Top fastest results for solutions with 42 or greater reward for scenario C. 130
9.5 Top fastest results for solutions with 42 or greater reward for scenario E. 131
9.6 Top fastest results for solutions with 20 or greater reward for scenario F. 132
9.7 Top fastest results for solutions with 55 or greater reward for scenario G. 133
9.8 Top fastest results for solutions for scenario H. The actual maximum reward attainable while remaining within resource constraints is 71, however, all but one solution exceeds this amount and violates resource constraints. 135

LIST OF FIGURES

Figure Page

1.1 Nominal flight with no fault (L); failed flight with a battery cell loss (R). Bottom plots show voltage and *State of Charge* (SOC) of the battery. 2

2.1 EOL distributions for the powertrain system of a UAV and individual components. (Darrah et al. (2020)) 10

2.2 General Prognostics Framework (Khorasgani et al. (2016)) 11

2.3 Long-Short Term Memory (LSTM) Cell 15

2.4 Bi-Directional Long-Short Term Memory Network Architecture. 16

2.5 Unstructured (left) and Structured (right) pruning (Chen et al. (2021)) 18

2.6 H-LSTM Cell 19

2.7 (L) Input to Hidden State; (R) Hidden to Hidden State 20

2.8 Replanning Approach using Off-board and On-board Processing (Quiñones-Grueiro et al. (2021)) 26

2.9 Integration of operational planning with active diagnosis (Kuhn (2011)) 28

3.1 Overall approach of the thesis. 30

4.1 Simulation Input-Output Diagram. 35

4.2 Single flight simulation Process 35

4.3 Simulation Model 37

4.4 Nominal voltage distribution. 38

4.5 Degradation profiles for battery internal resistance (R_0) (L) and charge capacity (Q) (R) . 40

4.6 Motor Degradation (R_m) 42

4.7 Available Trajectories 45

4.8 Trajectory generation. (L) too few nodes have been supplied and the waypoint at (160, 390) cannot be reached. (M) the connection distance is increased but now it is clear that the flight path is not optimal. (R) enough nodes have been selected and the connection distance has been reduced, providing the optimal trajectory. 46

4.9 Asset fields and relationships for an octorotor UAV system 50

4.10 Relation Diagram (Assets) for an octorotor UAV system 51

4.11 (a) entity relationship among processes and assets; (b) entity relationship with data fields and types. 52
4.12 Data-Driven Model Development Lifecycle 53
4.13 RUL mean, standard deviation, and distribution for 75 UAVs. 55
4.14 Example flight profile. Trajectory and waypoints (L), velocity, acceleration, and jerk (R). 57
4.15 Example flight profile showing elevation, takeoff, and landing. 58
4.16 Flight data from a nominal flight showing position, velocity, acceleration, orientation, and angular velocity. This data comprises the system state, a key input for data-driven prognostics. 59
4.17 Flight data from a nominal flight showing power statics from the battery and motors. This data is the second key set of features used for data-driven prognostics. 59
4.18 Flight data from a nominal flight showing position and control errors. The top row is the real-valued error in the X and Y axis. The middle row is the euclidean position error and the standard deviation in the euclidean position error. The bottom row is the control error in the X and Y axis. 60
4.19 Flight data from a nominal flight showing wind (constant + gust), which is an important input as external disturbances affect system operation. 60

5.1 Bi-Directional LSTM Architecture for RUL Estimation. 63
5.2 Feature importances of the data evaluated for suitability to be used as input to the RUL model. 65
5.3 Autoencoder architecture. 66
5.4 Original and reconstructed features of the RUL model input data. 66
5.5 Comparing the effect of regularization on the loss function. 67
5.6 Dataset split based on random UAV assignment. 71
5.7 Learning rate decay function. The function is evaluated after each training unit, where 56 training units comprise of a single epoch. 73
5.8 Training loss function. 73
5.9 RUL estimation results from the test and validation datasets, showing the mean predicted value (blue), the actual value (green), and the 95% confidence interval (shaded grey). 74
5.10 RUL estimation plots for the UAVs in the test and validation datasets. 74

5.11 RUL estimation error comparison for 3 Bi-LSTM Variants: The encoder-BiLSTM architecture presented in Section 4.1 (blue); and the same architecture with two different input sequence lengths and without the encoder (orange and green). The encoder-BiLSTM has both a lower RMSE and a smaller variance. 75
5.12 Input data plots for the flight time and power estimation model, sorted by value to show range of inputs. 75
5.13 Scatter plot of 30 random data samples of flight time estimations (top) and power consumption estimations (bottom). 76
5.14 Estimation error distribution in units of seconds (top), comparing estimations with and without system-level RUL information; Estimation error distribution in units of amp-hours for power consumption (bottom, same comparison). The model without RUL information has both a higher error and a larger variance. 76
5.15 Remaining flight time estimation with RUL information. 77

6.1 Saliency maps for RUL Estimation model at different stages of life. The y-axis represents the encoded features. The x-axis represents timesteps, where each step represents 7.5 minutes, for a total of 2.5 hours of operation per input sequence. Colors towards white represent little influence. Colors towards red or black represent a large influence, either negative or positive. 81
6.2 Highlighted differences among three stages of life of the mean values of the encoded features from Figure 6.1. The top portion shows the key differences highlighted. The bottom portion shows the original mean values. The key takeaway is that the same feature impacts the RUL estimation differently at different stages of life, where of most importance, is near End of Life (EOL). 82
6.3 Normalized neural activation absolute values of the Bi-LSTM hidden layers. White $\rightarrow$ 0, and dark red $\rightarrow$ 1. Colors towards white show neurons with little activation, where colors towards dark red show maximal activation. 83
6.4 Graphical depiction of the different pruning approaches. Blue is preserve, orange means open to pruning, and red means pruned. (A) is the baseline method, magnitude-based weight pruning. (B) is the activation-based method described above. (C) is the activation-based method with preservation of the cell state. 10% represents pruning at 10% sparsity. 20% represents pruning at 20% sparsity. 30% represents pruning at 30% sparsity. 40%+ represents pruning at sparsity levels of 40% or greater, where neural activations are used to prune the bottom 30% neurons and weight magnitudes are used to prune the bottom 10%+ weights of the remaining weights. 86
6.5 RUL estimation for 18 units from the test and validation set. 87

6.6 RUL estimation for 18 units from the test and validation set. 88
6.7 Comparison of different model reduction approaches on the desktop computer across varying sparsity levels from 0% (original model) to 90%. 89
6.8 4D plot showing inference speed (Z-axis), model size (Y-axis), sparsity level (X-axis), and absolute error (color). 90
6.9 Comparison of different model reduction approaches on the Unibap Q7 across varying sparsity levels from 0% (original model) to 90%. 91
6.10 4D plot showing inference speed (Z-axis), model size (Y-axis), sparsity level (X-axis), and absolute error (color) on the Unibap Q7. 91
7.1 Online Architecture for Health-Awareness. 94
7.2 Reward comparison between the desktop computer (orange) and embedded hardware (blue) shown for up to 30 seconds of run time using the data from Example Scenario D (see Figure 7.4). Each reward point represents an additional waypoint added to the new trajectory, which represents an increase in utilization while still returning to base safely. 95
7.3 Replanning framework. The GCS uploads the trajectory and performance thresholds to the UAV prior to departure. Online, the UAV continually monitors for faults and estimates RUL. When it determines it cannot finish the original trajectory it requests a new plan from the ground station. 96
7.4 Replanning scenario D. Left: The UAV does not enter replanning mode and fails shortly before reaching base. Middle: The replanner is activated and selects a new trajectory using waypoints near the existing flight path. Right: The UAV executes the new trajectory and returns to base safely. The replanner discared one waypoint from the original trajectory, and added two additional waypoints. . 112
7.5 Left: Naive replanner. Right: Health-Aware replanner. Red circles show different waypoints selected. The UAV travels clockwise in this particular trajectory. 113
7.6 GA performance plots on desktop computer. 114
7.7 GA performance plots on the Unibap Qseven embedded hardware. 115
7.8 Similar solutions to the same replanning scenario with the same reward value and path distances. Waypoint differences are highlighted with orange circles. 118

7.9 Left: Health-aware replanning solution when controlling for execution time (< 5 seconds → population size = 500, generations = 100) results in success. Middle: Health-*unaware* replanning solution results in failure. Right: Health-aware replanning solution when running time is not a factor (population size = 2000, generations = 200), also resulting in success but with 2 additional alternate waypoints. 118
7.10 Left: Using the compressed model resulted in an overestimation of remaining flight time and a failure prior to returning to base. Right: The same scenario but with the original uncompressed model. The remaining flight time was not overestimated, and the UAV safely landed. The discrepant waypoint is highlighted in red. 119

9.1 Flight Data Feature Ranges. 126
9.2 Replanning scenario A. 127
9.3 GA results for scenario A. 127
9.4 Replanning scenario B. 128
9.5 GA results for scenario B. 128
9.6 Replanning scenario C. 129
9.7 GA results for scenario C. 129
9.8 Replanning scenario E. 130
9.9 GA results for scenario E. 131
9.10 Replanning scenario F. 131
9.11 GA results for scenario F. 132
9.12 Replanning scenario G. 132
9.13 GA results for scenario G. 133
9.14 Replanning scenario H. 134
9.15 GA results for scenario H. 134
9.16 GA Performance. 136

CHAPTER 1

Introduction

The use of *Unmanned Aerial Vehicles* (UAVs) has rapidly grown over the last several years across a wide variety of applications that include aerial photography, surveillance, package delivery, cartography, agriculture, military missions, and more. In other words, the use of UAVs provides several benefits to society and their use and implementation will only continue to increase. They have the potential to revolutionize the delivery industry by providing faster and more cost-effective delivery solutions, especially for lightweight packages in urban areas. They can assist law enforcement agencies in surveillance operations, traffic monitoring, and crowd management during events. They can be employed to inspect critical infrastructure like bridges, power lines, and pipelines, and reach places too difficult for human inspectors.

These are just some of the many benefits UAV operations can bring to society, but these benefits can quickly be offset without adequate safety and reliability measures in place that specifically account for the health of the UAV. The field of safety and reliability is quite large and encompasses risk assessment using failure mode and effects analysis, software verification, reachability studies, reliability modeling, fault detection and isolation, fault adaptive control, and prognostics, to name a few areas.

As the adoption and use of these vehicles increase, the potential for flight failures becomes a significant concern. Such failures can lead to financial loss, decreased productivity, loss of equipment, and most importantly, endangering human lives. Maintaining safe and reliable operations is of utmost importance to minimize the risk of loss or injury from unforeseen events, such as the occurrence of faults and system degradation that leads to loss of performance and the risk of collisions and crashes during flight.

Previous research in this field does not account for *wear-and-tear* degradation that happens as a UAV flies multiple missions under a variety of operational and environmental conditions. The occurrence of abrupt faults during flight can significantly increase flight risks and necessitate the need for replanning (i.e., modifying a set of waypoints that the UAV can visit in its current mission before it has to return and land safely in its home base. The plans generated must not violate safety and performance constraints, and need accurate *State of Health* (SOH) information to ensure this.

Prognostics, a core foundation of this work, is the science of predicting failure and deriving SOH information. A motivating example is depicted in Figure 1.1, where a UAV operates in an urban environment with some approximate level of degradation (normal *wear and tear*) in its 8 motors and 6-cell battery. The UAV must reach a predetermined number of waypoints, stop at each for a time period, and return back to the starting location safely. On the left, no fault occurs and the vehicle is able to complete its flight. On the right, a fault has occurred early in the flight, and later the UAV fails to complete its mission. We revisit this scenario in Section 7.4 and show that replanning with SOH information is the only way to generate safe plans under these conditions.

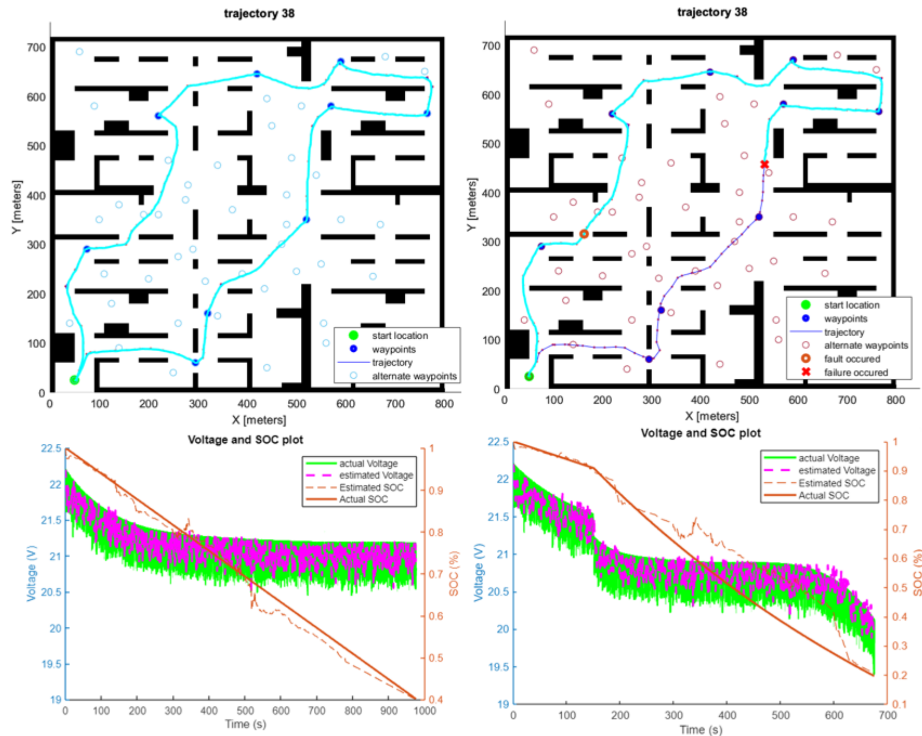

Figure 1.1: Nominal flight with no fault (L); failed flight with a battery cell loss (R). Bottom plots show voltage and *State of Charge* (SOC) of the battery.

This thesis research focuses on the problem of using *Neural Networks* (NN) for prognostics of complex systems, and combining this with replanning algorithms that can operate online on resource-limited hardware. Case studies are conducted on a simulation platform for UAVs. Since component or system degradation data is not readily available for lightweight, octocopter UAVs we construct a simulation testbed to generate degradation data over multiple UAV flight runs under varying operational and environmental conditions.

In subsequent chapters of the thesis, we discuss the motivation for using NN architectures for system-level *Remaining Useful Life* (RUL) prediction, and the challenges in generating sufficient data for training and testing these NNs are also discussed. The challenges of running NNs online for inference and prediction on

resource-limited hardware in inference and methods for compressing NNs are also discussed.

Our experiments demonstrate that compressing the neural network structures reduces resource usage (runtime and memory), and we study the effects of different model reduction methods on solution quality. Finally, we use our resource-constrained onboard RUL predictors to replan the UAV trajectories after faults and degradation (i.e., select the largest subset of waypoints that allow for safe flight) to ensure that the overall mission is safe (i.e., it can complete its mission without crashing).

1.1 Motivation

Health management schemes have typically focused on individual component degradation. Most current research does not focus on the interactions between different degrading components that make up the system, as well as the impact of the cascading effect of component degradation on overall system performance. Recent work on *System-Level Prognostics* has begun to address this problem. Components in a CPS typically degrade at different rates, and, therefore, require continual monitoring to avoid unexpected failures. Moreover, the effects of multiple degrading components on system performance are highly non-linear, and developing and maintaining accurate physics-based system models cannot be guaranteed for all systems. Typically, it is infeasible to run a true system to failure, so researchers and practitioners have resorted to using simulation-based techniques to better evaluate the effect of degrading components on overall system performance.

The increasing complexity of CPS has led us to use machine learning to develop prognostics models, which require high-quality input data to produce reliable results. The output of these models enable *prognostics-informed decision making*, such as *just-in-time* maintenance, or in the case of this work, *health-aware* re-planning. However, often times these models have drawbacks when they must be deployed to hardware with limited resources or computing power and must be compressed. The literature is sparse with methods for compressing *Long-Short Term Memory* (LSTM) models, which are specifically designed for time-series regression tasks. Compressing these models for safety-critical applications requires astute attention to the temporal dependencies within the model and the temporal characteristics of the input data. Currently, the literature lacks such focus for these types of networks, and instead is abundant with methods for other architectures and for classification tasks, primarily in computer vision. Developing such an approach is a key motivating factor of this research.

The deployment of these models is just as important as their development - ultimately these models will be used in the real world, where in the domain of CPS and complex systems, resource constraints must be considered. When deploying system-level prognostics models for online use, the tradeoffs between model size, inference speed, and solution quality must be well understood by system designers. The hardware used for model development in the lab is vastly different from the hardware used by the systems these models are developed for, which are typically less powerful and operate under stringent resource constraints. A model that has perfect accuracy is useless if it can't fit into the memory of the onboard embedded computing platform. A model that has undergone a transformation to reduce its footprint likely will not perform to the level of accuracy as the original model, and these differences need to be quantified.

1.2 Overview of Approach

PHM technologies have vastly improved over the last 20 years, although system complexity has continued to increase at a greater pace. In conjunction, the number of data points available for monitoring systems has also significantly increased, and adequately capturing the relationships among these features is a challenge that we continue to face. There are many ways to accomplish these tasks; however, with complex systems, researchers have resorted to using deep learning architectures because they have proven to be accurate nonlinear function estimators. In the case of prognostics, which is a time-series regression problem, *Recurrent neural Networks* (RNN) or their variants are favored. LSTM networks are a type of RNN network that can capture temporal dependencies in the sensor data, allowing the UAV to learn and predict its health state accurately.

Prognostics models are used to enable better decision-making for condition-based maintenance and operational scheduling, and the derived state of health information can be used for more immediate online actions such as replanning under faulty conditions, as in the case of this work. However, it is not feasible to run real systems to failure, and therefore this work starts with a simulation environment. A systematic approach to data generation, curation, and storage that supports studies in fault management and system-level prognostics for real-world and simulated operations is first developed. A data-driven simulation-based approach coupled with a data management methodology that enforces constraints on data types and interfaces, and decouples various parts of the simulation to enable proper linkage of related metadata allows for reliable and reproducible studies in system-level prognostics. This simulation framework facilitates data analysis and the development of data-driven models for prognostics using machine learning methods as well as incorporating these models back into the simulation to study their performance. Using this simulation testbed, the following

core tasks describe the approach:

1. Generate a run-to-failure dataset consisting of multiple UAVs operating under stochastic environmental, operational, and degrading conditions.
2. Develop a time series Bi-LSTM model with optimally tuned parameters to estimate RUL based on telemetry data and environmental conditions.
3. Utilize a model-explainable approach that accounts for the temporal characteristics of LSTM models performing time-series regression tasks to perform model compression.
4. Compare the size, speed, and accuracy of this approach to baseline methods that use magnitude based pruning.
5. Implement an anytime algorithm (such as a genetic algorithm) as the basis for a replanning agent that accounts for system-level RUL estimates and real-time health and operating conditions.
6. Demonstrate the performance of the compressed model when used with the replanning agent and compare against the original model and a naive battery discharge model in the target application domain.

1.3 Challenges & Contributions

Three primary challenges addressed in this work relate to developing nonlinear estimators for system RUL, model compression, and online deployment of the compressed RUL models for replanning after faults occur that cause significant degradation in the system. We elaborate on the challenges below.

1.3.1 Challenges

1. There are mixed results in the literature regarding the performance and tradeoffs of different model compression methods and the applications in which they are used. Moreover, there are no other studies of model compression methods applied to RUL prediction using deep learning architectures. Part of this lies in the difference between developing and evaluating a prognostics model on a workstation as opposed to deploying and evaluating that model on application-grade embedded hardware. The former is what most academic researchers do, while the latter is typically done in industry, where techniques and methods are kept proprietary. In addition to analyzing different model compression methods for

PHM applications, the process of deploying and evaluating these models is not well-illuminated in the academic/open-source PHM community.

2. Complex system behavior is both hybrid (discrete and continuous modes) and non-linear; the result of combinations and feedback loops of individual component operation and degradation. Developing data-driven prognostics models for CPS requires high-fidelity data that adequately captures the dynamics of the components, the environment, component degradation, and the effects of degradation on overall system performance. Very little real operational data is available for systems that are run to failure due to several reasons, primarily being that of safety and cost. On the one hand, simulation models are not an exact replica of the real system or how the system may operate in different environments. On the other hand, mathematical representations of system performance degradation are not easily derived, even when component degradation and system dynamics models are known. A number of run-to-failure experiments have been conducted for single components in isolation (Celaya et al. (2012); Heng et al. (2009); Kulkarni et al. (2010)), but these experiments do not capture the non-linear feedback loops among multiple components degrading simultaneously within a system, and the effects of their degradation on system performance.

3. Developing a simulation-based data repository for system-level prognostics of complex systems cannot be done without a *data-centric* approach to data management. Traditional methods for managing data from experiments employ predefined naming conventions and file hierarchies, with logs written during runtime as part of code execution. However, (Almas et al. (2016)) explain that there is a critical need to describe and document more details of the properties of the component and system behavior data, as well as the relationships between the components, that cannot be done by writing files to disk. These details are captured in *metadata*, which needs to be made an explicit and integral part of documentation to ensure that the data descriptions are accurate, and validation exercises can be performed. Data generated from complex, multi-component simulations must ensure that robust data storing conventions are followed, which becomes a daunting task without a well-defined interface and data schema available. This is known as *data provenance*, and guarantees that the stored data conforms to predefined standards.

These nuances in developing data-driven models for system-level RUL estimation have to do with the fact that they are used in safety-critical environments to make decisions that affect the safety of life or property. A wrong answer isn't just a model performance statistic; real world consequences follow. It is shown that using the system-level RUL estimation and real-time health state and operating conditions as additional inputs to a

replanning agent results in safe plans that do not violate safety or performance constraints. Such is not the case when the replanner lacks this information. Therefore, the final task (item 6 from above) is a tradeoff study of model performance in the target application domain. The obvious tradeoff to consider is solution quality and model size, however the method in which the model is reduced has been shown to affect the solution quality, even among models that are compressed to the same size. The literature contains conflicting results on this and there is some evidence that the results are application specific, which has strong implications. This is supported by findings in Chapter 6. Efforts to tailor pruning techniques to accommodate sequential data dependencies are crucial to maximize the benefits of model size reduction without compromising prediction accuracy in time series regression scenarios.

1.3.2 Contributions

This thesis centers around the development of an end-to-end framework for *health-aware* replanning of complex systems undergoing usage-based degradation and operating in unknown environments applied to *Unmanned Aerial Vehicles*. The first contribution to the field is a systematic study of model reduction methods specifically tailored to LSTM networks regression tasks, the type of task suited for prognostics. Approaches have been presented in the literature for LSTM networks, but they do not account for the temporal aspects of the underlying process and how model behavior changes and the process changes over time. A novel model compression approach not previously explored in the literature is presented, known as *neural activity gate-based pruning with cell-state preservation*, **which is specifically tailored to LSTM-based time series regression networks using neural activation patterns**. An experimental analysis is conducted with the reduced models on flight-certified hardware at NASA Ames Research Center, to understand how these models perform on application-specific hardware. A comparison between the original model, baseline model reduction method found in the literature, and the developed approach is carried out.

The second contribution to the field is the framework for *health-aware* replanning, whereas again, such an approach that incorporates system-level RUL estimations in conjunction with faulty conditions is absent in the literature. This work builds on top of research over the last decade in areas of prognostics, deep learning model development, and replanning, and brings the fields together for a replanning methodology that accounts for system-level state of health information. The need for accurate system-level RUL estimations is demonstrated with comparative examples of replanning that result in failed plans without this information.

These contributions are facilitated by the development of a stochastic UAV simulation testbed that allows for the seamless composition of high-fidelity component models, degradation models, environmental models, operating profiles, and an underlying database. Such an end-to-end simulation framework supports the data generation process, and am open-sourced complex simulation such as this is not available to the research community at large. This simulation includes (1) wind effects; (2) physics-based models of the airframe, batteries, and motors; (3) data-driven models of component faults and degradation for the battery, *Electronic Speed Controllers* (ESCs), and motors; (4) safety monitors that track a set of predefined performance parameters; and (5) data feeds to and from the simulation that allow for the seamless integration into a database management and model version tracking system. Finally, this simulation framework is used to generate the first system-level prognostics dataset with **continuous-time** remaining useful life, as opposed to discrete cycles where intra-cycle operational data can be lost.

The deep learning methodology is presented within the context of system-level RUL estimation models. Namely, feature selection & engineering, hyperparameter tuning, and training loss function. These topics are discussed through the lense of system-level prognostics. Feature selection methods specific to prognostics applications and a modified version of the standard hyperparameter optimization procedure is implemented. Finally, when we conceptually think about RUL for CPS, we want to know when it is no longer safe to operate *before* failure. The training function should therefore be designed such that overestimates are penalized more than underestimates. Such a function was originally presented by NASA researchers to score performance *after* training, and a modification to that function is given which allows it to be used *during* training.

1.4 Organization of Thesis

The rest of this thesis is organized as follows. Chapter 2 presents a literature review of system level prognostics, model compression, and replanning. Chapter 3 provides a concise overview of the thesis approach. Chapter 4 presents the simulation testbed and data generation framework, as well as the dataset used for model development. Chapter 5 introduces the system-level prognostics framework and deep learning model development approach. Chapter 6 discusses the model reduction methodology for time-series LSTM networks. Chapter 7 presents the health-aware replanning framework and demonstrates the efficacy of the compressed RUL model for health-awareness as well as a uniquely modified GA for replanning. Concluding remarks and a discussion of future works is given in Chapter 8.

CHAPTER 2

Literature Review

The topics covered in this work center around system-level prognostics, model reduction, and replanning. Of concern for CPS is the system's performance over time relative to some set of safety or performance metrics, and how to use this information for online or offline decision making. The increasing complexity in CPS over the last 20 years has brought the field from simple equation based approaches to the use of intricate deep neural networks. In recent years, using a mix of tools and custom software written in various languages, researchers around the world have developed prognostics applications with great results. However, a gap still exists between a holistic framework for system-level prognostics and then utilizing these approaches on embedded hardware. As it specifically to health-aware replanning, key points of discussion relate to the evolution of prognostics for CPS, prognostics datasets, simulations, deep learning, model reduction, and replanning.

2.1 System-Level Prognostics for Cyber-Physical Systems

Prognostics is the science of predicting how a physical asset's health and performance changes with use over time, as well as when a future event, typically an *End Of Life* (EOL) state, defined by a safety or performance threshold violation (Goebel et al. (2017)), will occur. Prognostics uses estimation and prediction methods to improve system reliability and maintainability. In the estimation step, the physical asset's SOH is derived, which monotonically decreases with use over time. In the prediction step, future usage is included to derive *Remaining Useful Life* (RUL), which is a measure of how long the system can remain in operation while meeting all of its safety and performance goals. The safety and performance goals are typically expressed as constraints on one or more monitored parameter that map to the system's SOH, whereby exceeding a threshold transitions the system to it's EOL state (Goebel et al. (2017)). *Prognostics and Health Management* (PHM) brings together all of the above concepts with life-cycle management to ensure safe and proper operations of the system (Peng et al. (2010)). Due to all of the sources of aelotoric, episodic, and temporal uncertainty, as well system behavior into the future, the predictions are often expressed as probability distributions with confidence bounds (Sankararaman (2015)).

Prognostics contributes to the overall safety of operations of complex *Cyber Physical Systems* (CPS), such as automobiles, aircraft, power plants, and manufacturing processes. These systems integrate physical processes and networking with embedded computing platforms, and operate in physical environments which may include some form of human interactions. All physical systems degrade with usage over time, and the added uncertainties in their operating conditions (e.g., changing weather conditions, abrupt faults, etc) make it imperative to track degradation in system components and overall system performance as the system operates. The inability to proactively manage system health can result in failures that lead to unnecessary downtimes, loss of assets, or even loss of human life.

Typically, prognostics is used at the component level to predict when an individual component, such as a battery or motor, will fail. This failure is defined based on the physical limits of the component, and not necessarily related to performance within the context of its operating environment. However, it is the rate at which the performance of the overall system degrades that is of greater interest to the operators and maintenance personnel. In system-level prognostics, we are interested in determining when the performance of a system will fall below pre-defined levels of acceptable performance. Accurate and reliable predictions make it possible to plan the future operations of the system, optimize maintenance scheduling activities, and maximize system life.

In previous work, it was demonstrated that system performance degrades much faster than individual components due to non-linear system behavior and the joint interactions between the components and their degradation effects. The failure PDFs are depicted in Figure 2.1, highlighting the key point of system-level prognostics: **systems will violate a safety or performance constraint before the individual components fail.**

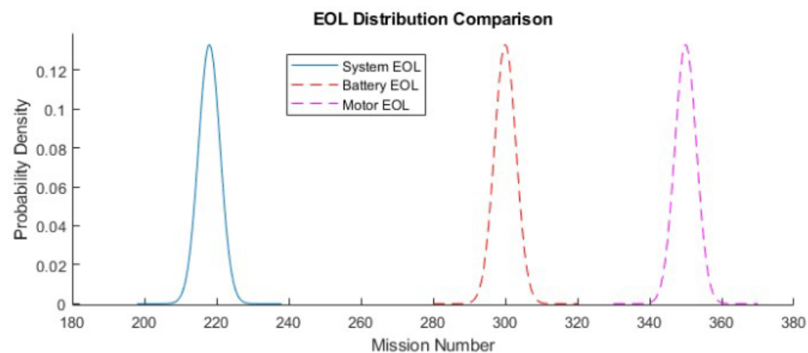

Figure 2.1: EOL distributions for the powertrain system of a UAV and individual components. (Darrah et al. (2020))

A lot of work has been done on RUL estimation for individual components, and the literature that studies

multiple component degradation and the affects on overall system performance is growing but still sparse. The next generation of space and aerospace operations need to incorporate system-level prognostics methods to ensure effective and safe operations. The general system-level prognostics problem is a two-step process where the joint state and degradation parameters are estimated first, and then the model is used in a prediction step which is simulated until EOL is reached. This process is depicted in Figure 2.2:

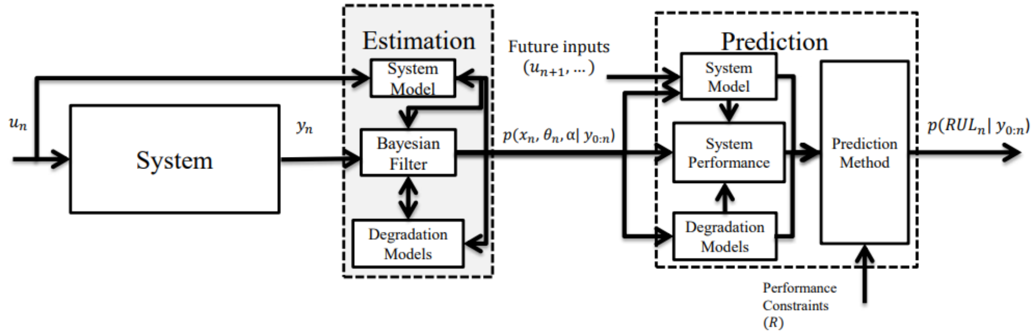

Figure 2.2: General Prognostics Framework (Khorasgani et al. (2016))

By looking at system performance, we inherently aggregate all forms and sources of degradation within that system, and therefore come up with different results for system-level RUL as opposed to component-level RUL, which leads us to a more specific definition than previously specified.

Definition 1 (System-Level Prognostics:) *System-Level Prognostics is a methodology for estimating the future performance of a system comprised of multiple interacting components that degrade simultaneously, with respect to a set of predefined safety and performance constraints, whereby an event such as End Of Life (EOL) occurs when one or more of these constraints is violated, and the time of that event less the current operational running time of the system is known as the Remaining Useful Life (RUL) (Darrah et al. (2022a)).*

This must account for uncertainty, external influences, component damage accumulation, the interactions among these components, and the effects of multiple component degradation on system performance. Components interact with one another, so individual component degradation is no longer an isolated function of inputs and environment, but includes the complexities of feedback loops and interactions with other components. Degradation models that predict performance at the system level are an aggregation of the individual component degradation models, and because the interactions can be complex and nonlinear, they are not solvable analytically. In addition, variability in the environment parameters and uncertainty in the aggregated degradation functions necessitates the use of stochastic simulation methods to derive system performance over multiple missions.

Using a proprietary simulation that hasn't been released to the public, Arias Chao et al. (2022) presented an approach that combines the strengths of physics-based and data-driven methods by using physics models to estimate unobservable parameters related to the system's health and component degradation. These parameters are then fused with sensor readings as input to a deep neural network, which represents a data-driven prognostics model with augmented physics-based features. The framework is evaluated using real flight data from a fleet of turbofan engines with promising results. Moreover, this research study has showcased the hybrid framework's capacity to deliver exceptional prognostics performance for units that operated under significantly different conditions than those used during the algorithm training process. However, even this work is limited by the discrete nature of the RUL function - discrete RUL functions are not capable of providing information *during operation*, only between cycles. **Currently, there are no system-level prognostics datasets with continuous RUL values**.

2.1.1 Data-Driven Prognostics with Deep Learning

Due to the complexities involved with prognostics at the system level, data-driven approaches must be utilized. These approaches use component measurements to learn a mapping from sensor data to damage accumulation or RUL, and require run-to-failure training data (An et al. (2015)). Data driven approaches are further divided into statistical methods (Si et al. (2011), Tsui et al. (2015)) or neural networks approaches (Zhang et al. (2019), Rezaeianjouybari and Shang (2020)). Approaches such as Gaussian Processes (Liu et al. (2013)), Support Vector Machines (SVM) (Chen et al. (2013)), or Bayesian techniques (Youn and Wang (2012)) fall under statistical methods. Recurrent Neural Networks (RNN) (Wu et al. (2020), Dong et al. (2017)) or Convolutional Neural Network (CNN) (Li et al. (2018)) are the most common deep learning architectures used.

Gaussian Processes have the advantage of not needing a model of the system, but assume all random variables are Gaussian distributions, and also tend to have large confidence intervals for long time horizons. SVMs are used for regression with prognostics applications, which seeks to find the hyperplane with the maximum number of points that is also as far separate from any other hyperplane. SVMs are limited, however, in that they offer point estimates only, and do not provide any information on the posterior probability $p(Y=y|X=x)$ necessary for uncertainty quantification. Bayesian techniques usually modify an existing technique to incorporate probabilistic information into the solution so that distribution values are returned and not point estimates. The Relavence Vector Machine (RVM) is the best example of this (Tipping (2001)), which adds a

prior distribution to the model parameters, and each observation results in a posterior distribution from which the output can be derived. The literature is rich with these approaches and countless variations of them, but over the last five to ten years deep learning approaches have grown more popular.

2.1.1.1 Deep Learning Approaches

Deep learning approaches typically fall under RNNs or CNNs. Autoencoder networks (AE) have been used as well, but the mechanism for tracking SOH is fundamentally different than the former network architectures. In AEs, the network is trained on healthy input data with the objective to minimize the reconstruction error. As the asset degrades over time, the reconstruction error grows, and it is the reconstruction error that in turn is used as the monitored variable. The second way AEs are used is as a feature extractor, whereby the decoder portion of a trained AE is discarded, and the latent layer is connected to the input layer of the model.

Convolutional Neural Networks (CNNs) are a class of deep learning models designed for image and visual data analysis. They are inspired by the visual processing in the human brain and are particularly effective in tasks like image classification, object detection, and image segmentation. The fundamental building blocks of CNNs are convolutional layers, which apply filters or kernels to the input image, enabling the network to detect various features and patterns at different spatial scales. CNNs work by convolving the input with one or more kernels to extract local features and then these features are further selected based on their significance during pooling. When CNN layers are stacked, the network is capable of learning hierarchical representations, whereby subsequent layers represent higher level patterns in the data. The pooling layers, such as max pooling, help downsample the spatial dimensions, reducing computational complexity while retaining important information. The use of shared weights and pooling allows CNNs to capture translation-invariant features, making them suitable for handling images of different sizes and orientations. With time series data such as telemetry data from a CPS used for prognostics, 1D or 2D CNN networks are used, where 1D networks consist of multiple timesteps of a single feature, 2D networks consist of multiple timesteps of multiple features. 2D networks rely on large matrix multiplications, while 1D networks utilize simpler array operations, and are typically faster. Lövberg (2021) used a stacked CNN model with dilated convolutions to perform RUL estimation of jet aircraft engines using the N-CMAPSS dataset (Chao et al. (2021)) and had the best results of the 2021 PHM data challenge competition.

RNNs are capable of accounting for temporal dependencies by connecting the output of layer n to layer $n-1$.

However, they suffer from the famous *vanishing gradient* problem, which is where the gradient updates tend towards zero the further back in time they are propagated. This is caused by the fact that the derivative of the sigmoid function is always under .25. LSTM networks are typically used in favor of the original formulation of the RNN. LSTMs introduce the concept of *gates* and a *memory cell*. The gates control the flow of information into and out of the cell, as well as determines how much *memory* should be forgotten. Bi-Directional LSTMs are a further enhancement to these types of networks that allow for future information to pass backwards in time and affect the gradient during the update step of training. This was the network used in (Darrah et al. (2022b)), which showed that even a very simple network can attain results on par with networks with far more advanced architectures (such as those that use skip layers, parallel channels, etc).

LSTMs are a type of recurrent Neural Network (RNN) which is a class of networks specifically designed to handle sequential data and time-series tasks. Unlike traditional feedforward neural networks, RNNs possess recurrent connections that allow them to maintain internal states or memory, enabling them to process sequences of varying lengths and capture temporal dependencies. The basic architecture of an RNN consists of a hidden state that evolves over time and serves as the memory of the network. At each time step, the RNN takes an input and updates its hidden state using the input and the previous hidden state. This sequential processing allows RNNs to incorporate information from past inputs to influence future predictions. However, standard RNNs suffer from the vanishing and exploding gradient problems, limiting their ability to capture long-term dependencies effectively. To address these issues, various advanced RNN architectures have been proposed, such as Long Short-Term Memory (LSTM) networks introduced by Hochreiter and Schmidhuber (1997) and Gated Recurrent Units (GRUs) proposed by Cho et al. (2014). These architectures utilize specialized gating mechanisms that control the flow of information, facilitating the preservation of important information over longer sequences. RNNs have demonstrated significant success in a wide range of sequential data tasks, including natural language processing, speech recognition, time-series prediction, and more. They are particularly well-suited for tasks where the order and context of data are critical, enabling them to model dependencies across time steps. LSTM networks and variants have been used widely in prognostics with excellent results (Wu et al. (2017); Huang et al. (2019); Darrah et al. (2022b)). The LSTM cell has 8 weight matrices and 4 bias, where the four W_x matrices have a size of $[out_{dim}\ x\ in_{dim}]$, the four W_h matrices have a size of $[out_{dim}\ x\ out_{dim}]$, and the four bias have a size of $[out_{dim}\ x\ 1]$. The LSTM cell is the basic structure of LSTM networks, and is depicted below in Figure 2.3.

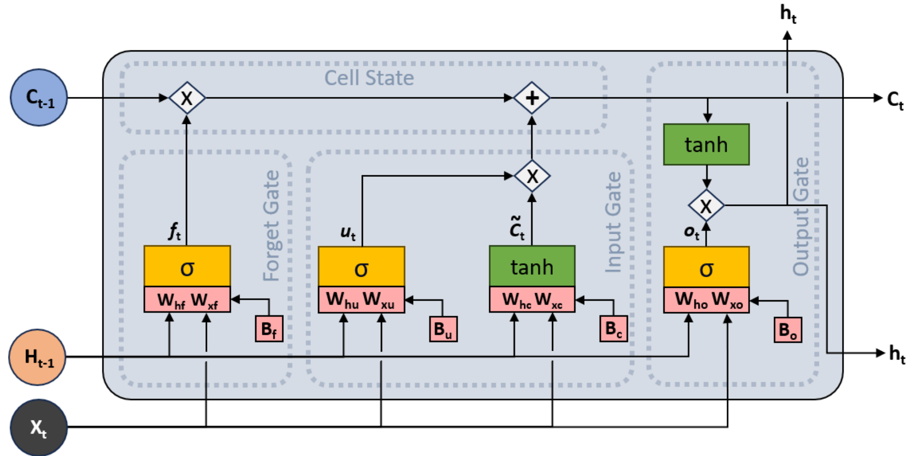

Figure 2.3: Long-Short Term Memory (LSTM) Cell

A significant advancement for RNNs was the development of the *bi-directional* RNN, first proposed by Schuster and Paliwal (1997). The authors propose a novel architecture that extends traditional recurrent neural networks (RNNs) by incorporating bidirectional information flow. In conventional RNNs, information flows in one direction, from past to future elements in the sequence. In contrast, Bi-directional RNNs process the input sequence in two passes: one in the forward direction (past to future) and another in the backward direction (future to past). By combining both passes, the network is able to capture dependencies from both past and future elements, enabling them to model long-range dependencies more effectively. The authors evaluate the performance of the bi-directional architecture on a speech recognition task where experimental results demonstrate that bi-directional RNNs outperform unidirectional RNNs and other architectures. By incorporating information from both past and future elements, bi-directional RNNs exhibit superior ability to model temporal dependencies, leading to improved accuracy in sequence modeling tasks. This approach was extended to LSTM networks by Graves and Schmidhuber (2005) for the same task, with results better than previously attained. The Bi-Directional LSTM architecture is shown below in Figure 2.4, originally created by Cui et al. (2018).

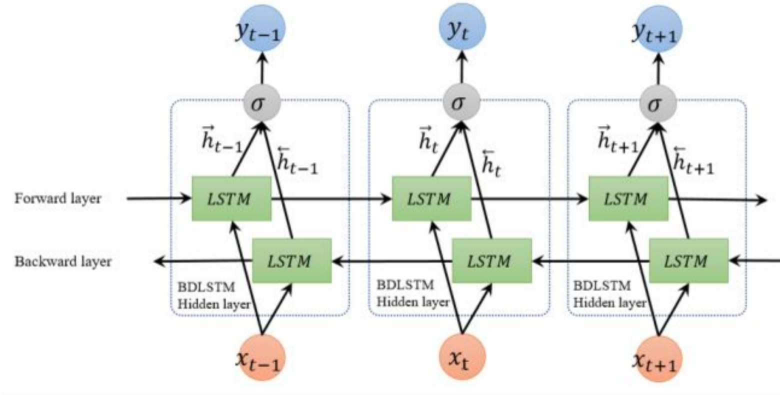

Figure 2.4: Bi-Directional Long-Short Term Memory Network Architecture.

The Bi-LSTM cell consists of two LSTM cells: the forward LSTM layer, which generates the output sequence h-right in the conventional manner, and the backward LSTM layer, which calculates the output sequence h-left using the reversed inputs from time t-1 to t-n. Both output sequences are then combined through a sigma function to generate the final output (Li et al. (2020b)). This unique approach allows the model to leverage bidirectional processing while producing an output that is equivalent to that of a unidirectional LSTM. By considering information from both past and future elements in the input sequence, the Bi-LSTM enhances the network's ability to capture intricate patterns and long-range dependencies, contributing to more accurate predictions or estimations in various applications, such as time series forecasting, natural language processing, and speech recognition.

2.2 Model Compression

Model compression techniques have gained popularity over the last few years given the rise of on-edge computing, mobile computing, and the Internet-of-Things. There are over 200 pre-trained tensorflow-lite models available from the official repository[1], models that have already undergone reduction, and the repository continues to grow. The goal is to deploy models to resource-constrained embedded platforms where model size, inference speed, power consumption, and thermal load are the primary considerations. Large neural networks use a considerable amount of memory bandwidth and processor time for matrix operations, requiring lots of energy. A clear summary of this point is given by Han et al. (2016):

> *"Energy consumption is dominated by memory access. Under 45nm CMOS technology, a 32 bit floating point add consumes 0:9pJ, a 32bit SRAM cache access takes 5pJ, while a 32bit DRAM memory access*

[1] https://tfhub.dev/s?deployment-format=lite

takes 640pJ, which is 3 orders of magnitude of an add operation. Large networks do not fit in on-chip storage and hence require the more costly DRAM accesses. Running a 1 billion connection neural network, for example, at 20fps would require (20Hz)(1G)(640pJ) = 12:8W just for DRAM access - well beyond the power envelope of a typical mobile device."

In addition to energy consumption, the other primary reason for model compression is to reduce the amount of physical resources needed to both store (memory) and use (compute) the neural network model. *Pruning* and *quantization* are two primary methods used to compress a neural network model that have been extensively explored in the literature. These techniques have different effects on the resultant model, and subsequently the model's performance when compared to the original model. These trade-offs must be carefully understood and taken into account for PHM applications, where safety and reliability are of utmost importance. In all cases it is desired to retain the highest amount of accuracy, but specifically to PHM, it is desired to ensure that overpredictions are minimized.

Pruning was first described in a groundbreaking paper titled *"Optimal Brain Damage"* (LeCun et al. (1989)) where better generalization and improved inference speed were found when unimportant weights from a network were removed. The idea is to find parameters with minimal salience, that is whereby the removal of said parameter will have a minimal effect on the validation error. A common pruning approach is to first train a network until it converges, then score each parameter or parameter group based on the amount of influence they have on network activations (Blalock et al. (2020)), or the magnitude of the weight. Afterwards, low scoring elements are removed either all at once or according to a scheduling procedure, and then the network is retrained for a final fine-tuning step. This is necessary to recover as much accuracy loss as possible during pruning. The basic pruning algorithm from which all other variations derive from is given in Algorithm 1.

Algorithm 1: Basic Pruning Process

Input: M, a pre-trained model, X, the training dataset, γ, desired sparsity level, n, number of fine-tune epochs

Output: M'

1 $S_M \leftarrow score(M)$
2 $M' \leftarrow prune(M, S_M, \gamma)$
3 $M' \leftarrow finetune(M', X, n)$

There are two types of pruning, *structured* and *unstructured* (Chen et al. (2021)), as depicted in Figure 2.5. In unstructured pruning, individual parameters are pruned without any respect to their relationship to other parameters or the overall network architecture. In structured pruning, larger network elements such as layers,

channels, filters, or other aggregate network structures are pruned.

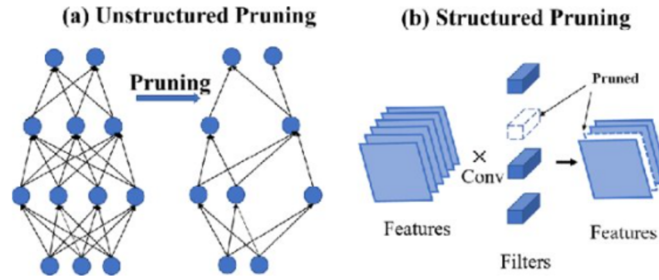

Figure 2.5: Unstructured (left) and Structured (right) pruning (Chen et al. (2021))

Furthermore, pruning can occur either *locally*, such as scoring and pruning on a layer-by-layer, or it can be done *globally*, in which the parameters from all layers are considered jointly (Liu et al. (2018)). In (Frankle and Carbin (2018)), the authors found that global pruning resulted in smaller networks and hypothesized it had to do with the amount of information in the layers. In that work, the authors were using Resnet-18 and VGG-19, wich are networks designed for image analysis tasks. In (Mishra and Mohna (2021)), the authors found the exact opposite result using a BERT-based language model, and local pruning resulted in a network that was half the size as one pruned globally before significant performance drop. The only major difference between their work was the application. This is an area of research that needs explored further in other domains to draw more conclusive results. There are no examples of a comparison between these methods for prognostics in the literature, one of the contributions of this thesis.

2.2.1 Pruning Recurrent Networks

Most approaches to pruning of neural networks have primarily focused on CNNs and their application to image classification tasks. In contrast, pruning techniques for Long Short-Term Memory (LSTM) networks, which are widely used for time series regression tasks, have received comparatively less attention. LSTM networks are designed to capture temporal dependencies and process sequential data, making them well-suited for regression tasks and the processing of time series data. However, their recurrent nature and intricate memory cells pose unique challenges for pruning. Most approaches in the literature for LSTM networks utilize variations of existing methods without actually accounting for the temporal nature of the model. One example of this is the Random Connectivity LSTM (Hua et al. (2019)), which is an approach to pruning whereby the connection between nodes is made randomly based on a pre-assigned probability distribution of their connection. The authors are able to achieve a high degree of sparcity, however, their model performs

significantly worse than an un-pruned LSTM, with an RMSE 36% higher than the baseline model.

The paper titled "Grow and Prune Compact, Fast, and Accurate LSTMs" by Dai et al. (2018) is often cited in the model compression literature for LSTM networks. The paper proposes a novel approach called hidden-layer LSTM (H-LSTM) to address issues related to LSTM's depth when used for sequential data modeling. Instead of stacking LSTM cells to increase depth, H-LSTM replaces the LSTM's control gates with a Dense layer. This is shown in Figure 2.6.

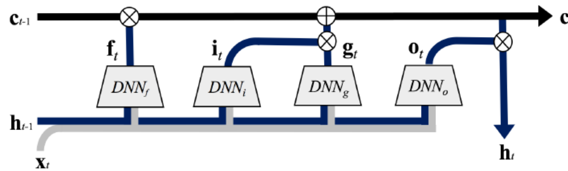

Figure 2.6: H-LSTM Cell

During training, the network starts with a compact architecture with fewer hidden layers or connections. As the model is trained on the data, the gradients of the loss function with respect to the model parameters are computed during the backpropagation step. In the grow phase, the network dynamically adds new hidden layers or connections based on the gradients of the loss function. If certain connections or layers have large gradients, indicating that they are crucial for improving the model's performance, the growth algorithm adds these connections or layers to the network. By doing so, the model expands its architecture during training to better accommodate complex patterns and dependencies in the data. The growth process is guided by the principle that the network should adapt its architecture to capture important information and optimize its performance effectively. The gradient-based growth helps the model to learn which parts of the network are most beneficial for the task at hand, allowing it to dynamically adjust its architecture to suit the data and avoid unnecessary redundancy. After the growth phase, the pruning step aims to simplify the network architecture by removing less important connections. Connections with small magnitudes, which means their weights are close to zero, are considered less essential for the model's performance. In magnitude-based pruning, these less influential connections are pruned from the network, reducing the model's complexity and memory footprint.

Experiments were conducted on image captioning and speech recognition tasks using Microsoft's Common Object in Context (MSCOCO) and AN4 speech recognition dataset by Carnegie Mellon University. The paper details mixed results that in one experiment show that H-LSTM models trained with GP reduce the number of parameters and run-time latency significantly while achieving higher accuracy compared to traditional

stacked LSTM models. While in another experiment, their model did have higher accuracy than traditional stacked LSTM models, however, when a single-stacked H-LSTM model was compared to a single-stacked LSTM model, the H-LSTM model had nearly 50% more parameters, and an inference speed that 12% slower. In nearly all of their experiments the H-LSTM model outperformed all other models in terms of accuracy, and in several cases their model had fewer parameters, however, the authors did not provide any clarity on the mixed results.

Another approach to pruning for LSTM networks was proposed by Wen et al. (2020), where the authors developed a method for pruning individual neurons through neuron selection mechanisms, represented by sets of binary "gate" or "switch" variables, to control the presence of specific input and hidden neurons in the LSTM architecture. In this approach, two sets of binary variables are introduced: one set controls the presence of input neurons (z), and the other set controls the presence of hidden neurons (s). Each binary variable takes values of 0 or 1, indicating whether a neuron is included (1) or excluded (0) in the network. When a binary gate variable is set to 0, it effectively switches off all hidden neurons connected to the corresponding input neuron (resulting in a column of zeros in the weight matrix). Similarly, when a hidden neuron's binary gate variable is 0, it switches off that entire row in the weight matrix. This is depicted below in Figure 2.7.

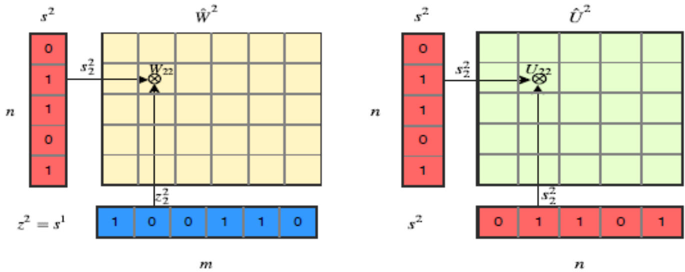

Figure 2.7: (L) Input to Hidden State; (R) Hidden to Hidden State

The uncertainty of each gate variable is modeled using Bernoulli distributions, $z_i \sim \mathscr{B}(\pi_{z_i})$ and $s_j \sim \mathscr{B}(\pi_{s_j})$, where π_{z_i} and π_{s_j} are the distribution parameters for each element in z and s, respectively. The objective is to optimize the network by minimizing the L_0 norm of the weight matrix, which explicitly penalizes non-zero parameters of the model. The distribution parameters are found through a continuous relaxation method first proposed by Maddison et al. (2016). The key idea behind the technique is to transform each random variable into a differentiable function that depends on its parameters and a fixed-distribution random variable. By making this transformation, the gradients of the loss propagated through the graph become low variance unbiased estimators of the gradients of the expected loss. Their approach is evaluated on the word-level Penn Treebank dataset, which contains over 1M words comprised of a vocabulary of 10k words. The resulting

model was 10 times smaller than a vanilla LSTM model, nearly 20 times faster, and only suffered a drop in accuracy of 1.1%.

The problems under study in these works were various language modeling tasks, which while the underlying problem has sequential characteristics, they inherently differ from time series regression tasks. Language modeling tasks focus on generating meaningful text by modeling contextual dependencies in a sequence of discrete symbols, while time series regression tasks involve predicting numerical values in a continuous sequence, capturing temporal patterns and dependencies in the data. Despite both involving sequential data, their objectives, data representations, and evaluation criteria differ due to the specific nature of the problems they address. In time series regression tasks, where accurate predictions depend on capturing temporal patterns, pruning must be carefully applied to avoid significant performance degradation. Effective pruning methods for LSTM networks and time series regression tasks have not been specifically proposed in the literature, and such techniques require the help of model explainability methods to inform proper pruning.

2.2.2 Explainable-Informed Pruning

Explainability has becoming an increasingly important topic as the use of machine learning and deep learning continues to grow in our society. Explainability methods aim to enhance the transparency and interpretability of AI models, providing understandable justifications for their decisions. These methods encompass various approaches, such as feature importance analysis, which quantifies the relevance of input features in model predictions (Lundberg and Lee (2017)). Local interpretability methods, exemplified by techniques like Local Interpretable Model Explainations (LIME), focus on providing explanations for specific instances rather than the entire model, enabling users to understand the model's behavior around particular data points (Ribeiro et al. (2016)). Saliency methods, like Grad-CAM, visualize the regions in images that influenced the model's prediction, offering insights into the model's focus (Rs et al. (2017)).

Deep Learning Important Features, or DeepLIFT, by Shrikumar et al. (2017), is an explainability technique used to attribute the contributions of individual input features to the output of a deep neural network. It starts by defining a reference baseline, representing the default input. In the case of the MNIST dataset, for example, the reference baseline is an input of 0's. The authors explain that the choice of a reference input relies heavily on the specific problem and domain knowledge. Then, it computes the differences between each input feature's value and the reference baseline. These differences are propagated through each layer

and the propagated differences are rescaled to ensure that the contributions of all features sum to the difference between the model's output for the actual input and the reference baseline. The rescaled differences serve as relevance scores, indicating the importance of each feature in the model's prediction. Positive relevance scores indicate features that contribute positively to the prediction, while negative relevance scores suggest features that have a negative impact. Mathematically, this is expressed as follows: let t be a target output neuron and let $x_1, x_2, ..., x_n$ be a set of neurons in an intermediate layer who's output is used to compute t. Let t^0 represent the reference activation of t. Then, Δt is the *difference from reference*, and $C_{\Delta x_i \Delta t}$ is the contribution score assigned to x_i that is the amount of contribution that neuron x_i has on t, such that the following equation holds:

$$\sum_{i=1}^{n} C_{\Delta x_i \Delta t} = \Delta t$$

Sabih et al. (2020) discusses the utilization of explainable AI methods, particularly the DeepLIFT method, to improve deep neural network compression techniques such as pruning. They employ a layer-by-layer sensitivity analysis to find an adequate scaling factor to derive the global importance of each neuron in each layer by distorting each layer using random pruning. In the context of classification, the reasoning behind this is that they seek to find how sensitive each layer is to distortion by looking at the pairwise distance between classes in a classification task. Each layer also has a *load* value, which is simply the number of parameters in the layer or the number of multiply-accumulate (MAC) operations, this allows flexibility in determining if reducing the number of operations or reducing the number of parameters should be prioritized. Their algorithm utilizes Global Pruning, and given below.

Algorithm 2: DeepLIFT Based Global Pruning by Sabih et al. (2020)

Input: M, a pre-trained model, E, the maximum allowable error, N,maximum number of iterations
Output: M', a pruned model

```
1  while error < E and n < N do
2      S ← layer sensitivities
3      L ← layer load (MACs) or # params
4      P ← S x L
5      foreach layer do
6          l1 ← l^1 norm of DeepLIFT importances
7          l1 ← sort(l1)
8          M' ← prune to P
9      end
10 end
```

Layerwise relevance propogation is another technique from explainable AI proposed by Binder et al. (2016). Layer-wise Relevance Propagation (LRP) is an explainability technique used to attribute the contributions of individual neurons or features in a deep neural network (DNN) to the model's output. LRP starts by assigning relevance to the model's output, typically setting it to 1 for the predicted class and 0 for all other classes. Then, it proceeds with a backward propagation process, starting from the output layer and moving towards the input layer. During this process, the relevance assigned to the output is propagated backward through the network's layers. As the relevance is propagated backward, it is redistributed among the neurons or features in each layer based on their contributions to the layer's output. The final relevance score for each neuron or feature is obtained by summing the redistributed relevance across all paths that pass through that neuron or feature. Yeom et al. (2021) utilizes this method to inform the pruning process of a CNN for a classification task. The relevance score calculation is as follows:

$$R_i^l = \sum_j \frac{a_i^l w_{ij}}{\sum_{0..j} a_i^l w_{ij}} R_j^{l+1},$$

which is calculated in a similar fashion as backpropogation, starting out the output, where the relevance score for the target class is 1, and all other output nodes are 0. Then, using a modified version of Algorithm 1 (Algorithm 2 is also a modified version), relavence scores for each neuron are used to inform which nodes to prune. They compare this method to weight based and gradient based pruning, and show it outperforms both methods on AlexNet, VGG-16, and ResNet. However, this approach, like other explainable-based approaches to pruning in the literature, focus on classification tasks with CNNs. The literature is lacking with contributions that focus on LSTM networks for regression tasks, and that is an area this thesis focuses on.

2.2.3 Quantization

Quantization is used to map a continuous valued parameter to a discrete valued parameter to minimize the number of bits required for storing parameters and matrix operations. Converting a network of 32-bit floating-point parameters to 8-bit integer significantly reduces the amount of memory required to hold the network in memory as well as well as increases inference speed. The quantization function, $Q(\rho)$, takes floating point values and maps them to an integer range, typically $INT8$, as follows in Equation 2.1 (Jacob et al. (2017)):

$$Q(\rho) = \lfloor S\rho \rceil - Z \tag{2.1}$$

where ρ is a network parameter value, S is a scaling factor which determines the quantized step size, $\lfloor \rceil$ is an integer rounding function, and Z is the zero point of the integer range, which ensures quantization errors are not introduced with common operations such as zero padding or ReLU activation. The scaling factor, S, is defined by the range of input values as well as the bit width (i.e. 8 for $INT8$ quantization). S is defined in Equation 2.2 (Gholami et al. (2021)):

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{2.2}$$

where $[\beta, \alpha]$ is the bounding range that ρ is clipped to, and b is the bit width. Choices for α & β vary but one of the most common approaches uses the min and max parameter values from a small *calibration* dataset. This is known as *asymmetric quantization*, and zero in the input space does not map to zero in the quantized space. Another common method to determine the bounding range is to take twice the maximum of the absolute values of ρ in the calibration centered on zero, which is known as *symmetric quantization*. Asymmetric quantization typically results in better performing networks, but symmetric quantization typically results in faster networks (since there is no mapping from 0 to Z).

In either case, these quantization techniques is known as *uniform quantization*, in which the quantized bins are equally spaced apart. *Non-uniform quantization* is exactly the opposite, where a given strategy can be used to customize the discretization space and better preserve the underlying information (Liu et al. (2021)). However, these schemes pose hardware related challenges and most approaches in the literature use the uniform method. There are also two different processes for quantization, *post-training quantization* (PTQ) and *quantization-aware training* (QAT) (Nagel et al. (2021)). QAT requires network alterations before training, and learns quantized valued parameters through training, as opposed to a transformation later (Equation 2.2). QAT can achieve 4- and 2-bit precision levels while retaining acceptable accuracy. PTQ does not require re-training and in general is more than adequate to achieve $INT8$ quantization. PTO does not require network modifications either, so and allows for any pre-trained model to be quantized. Quantization can also be done to the model parameters only, or the output of the activation functions can be quantized as well.

Quantization can be further categorized by whether only the weights are quantized, or if the weights and activation functions are quantized (hybrid). In weights quantization, only the model's weights are converted to 16-bit floating-point or 8-bit integer representations, halving or quartering the model size, respectively.

When using 16-bit floating-point quantized weights, they are converted back to 32-bit floating-point during CPU execution due to CPU limitations. However, 16-bit floating-point can be beneficial for GPU execution. On the other hand, 8-bit integer quantized weights use hybrid operators, enabling dynamic quantization of activation values to 8-bit integer and computation with 8-bit weights and activations (Jacob et al. (2017)). This approach provides latencies close to fully fixed-point inference while maintaining float precision for outputs, leading to speedup over pure floating-point computation, although the memory footprint remains unchanged. Hybrid operators are utilized for compute-intensive tasks in a network, such as fully-connected and Conv2D operations.

2.3 Replanning

Planning is a crucial aspect of autonomous cyber physical systems that allow them to operate and perform their function with minimal human interaction. *Replanning* involves generating new plans in response to changes in the environment, changes in the system, or changes in the system's goals, and is necessary when operating in dynamic and stochastic environments. Replanning methods aim to ensure the system is adaptable, robust, and efficient by continuously updating plans or trajectories to achieve desired objectives. Virtually all planning algorithms can be modified to be a replanning algorithm, but this poses additional constraints and considerations that will be discussed. In the field of robotics (mobile robots, robotic arms, and multi-robot systems), planning studies have explored various techniques, including potential fields and model-predictive control (Li et al. (2021)), and hierarchical task planning (Ryu (2020)) to enable robots to plan their actions. In the domain of UAVs, planning and is vital for tasks such as surveillance, reconnaissance, and package delivery in dynamic environments. Research works focus on developing real-time planning algorithms that consider factors like obstacle avoidance (Yang et al. (2022)) or energy efficiency (Ahmed et al. (2016)).

Online replanning assumes that an agent is initially following a predefined plan and due to new information must change the plan (Bonet and Geffner (2011)). With an updated knowledge base, a new plan can be computed that would allow it to achieve a subset of its objectives (Komarnitsky and Shani (2016)). In the domain of CPS, an unviolable objective will always be to maintain safe operations and minimize risk of failure. Risk is therefore a probability function of the system's SOH and performance constraints. A vast majority of the algorithms to solve these types of problems are well known, but their implementation in these systems is not a trivial task. In a static environment, everything can be computed offline without worry of processor limitations or computational complexity. In a dynamic environment, a complex system such as a

UAV must repeatedly make these calculations and trigger a replanning mode if it detects a failure along its current trajectory. In (Quiñones-Grueiro et al. (2021)), a multi-objective cost function was implemented to assess risk, and a path search algorithm is utilized offline to generate a new trajectory and then communicated to the UAV. The replanning approach is depicted in Figure 7.3.

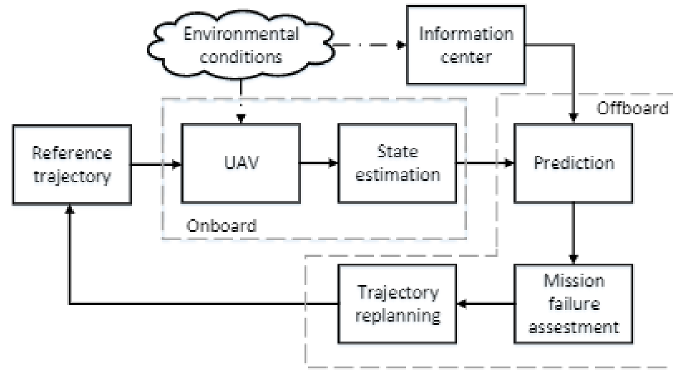

Figure 2.8: Replanning Approach using Off-board and On-board Processing (Quiñones-Grueiro et al. (2021))

In (Krishnan and Manimala (2020)), the authors consider both the minimal path length and the minimal risk of collision for designing a path-planning algorithm suitable for real-time applications. They, however, do not consider the state of health of the UAV in the optimization problem. Real-time path planning for UAVs in the context of obstacles as dynamic traffic and geofences is explored in (Chatterjee and Reza (2019)). They use RRTs and build on top of the DAIDALUS software by NASA to implement maneuver computation for collision avoidance. To save computation time, the authors propose heuristics for early termination of tree generation for RRTs. In (Zammit and Van Kampen (2020)), the authors compare an A* graph search algorithm with limited look-ahead and intermediate goals, against RRTs for path planning. In their scenario of traversing through openings in walls laid out in a single principal direction, A outperforms the more random exploration of RRT. This may be because the heuristics for intermediate goal points and the cost to goal are monotonic with respect to the optimal trajectory and its true cost. Performance may change if the environment required moving in directions at obtuse angles to the position vector to the goal.

Benders et al. (2020) propose an adaptive path planning strategy that takes into account performance uncertainties associated with the parameters of the kinematic model of the UAV. Convex optimization strategies have also been proposed for real-time trajectory planning (Aggarwal and Kumar (2020)). Chakrabarty et al. (2019) propose a recursive tree based path planning strategy for UAVs and precompute branches of the tree to reduce computation while operating online. The root of the tree is initialized with the position of the UAV and the time when started. A graph is incrementally constructed where the vertices represent the inertial

positions of the UAV and edges consist of collision-free paths between them. From the start position, the tree is expanded via pre-computed primitives based on the UAVs kinematics. In their previous work (Chakrabarty and Langelaan (2013)), they describe the motion primatives as a tuple, $(T_i,\ v_j\ \Delta\Psi_k\ \phi_l)$ where T is thrust, v is velocity, $\Delta\Psi$ is the change in heading, and ϕ is the bank angle. Each of these parameters are assigned values in a discrete value set, effectively precomputing $|T|\ x\ |v|\ x\ |\Delta\Psi|\ x\ |\phi|$ states. In the recursive tree based approach in 2020, they define the cost function in terms of minimum jerk, instead of in their 2013 work, where they used minimum energy. The Tree is grown in a manner similar to A*, which utilizes the cost function and a heuristic function to explore the graph. The recursive version of this re-initializes the tree at a rate of 1 Hz, using the UAVs current position as the root of the tree, once again.

These approaches consider the kinematic constraints associated with the UAV in the path planning method. However, they do not consider the dynamic constraints associated with, for example, faults. Replanning informed by fault management systems is discussed next.

2.3.1 Fault-Aware Replanning

Replanning under faulty conditions is a fundamental aspect of safe and reliable UAV operations and comes into play when unforeseen disturbances or faults occur during a flight mission. The primary aim of fault-aware replanning is to amend the current flight plan in order to ensure that the system is still able to operate and achieve some of its goals while finishing safely. This involves real-time adjustments to the flight path and waypoints, influenced by the dynamic feedback of the UAV's state and the environment. A necessary precursor is some form of fault detection and isolation that can identify when a fault occurs in real time as well as the fault magnitude. The literature is rich with techniques for fault detection and diagnosis methods, which can be classified as data driven (hypothesis testing, machine learning), model based (filtering, residuals), or hybrid approaches (filtering + hypothesis testing). The reader is directed to (Hu et al. (2020); Venkatasubramanian et al. (2003); Badihi et al. (2022)) for a comprehensive review.

Fault management incorporates these techniques into the operation of the underlying process and in addition to detection and isolation, includes *recovery*. The objective of fault recovery is to restore normal operation as quickly a possible, prevent the fault from escalating, and other actions that might take the system offline such as shutdown and repair. Levinson et al. (2018) presented a comprehensive framework for fault management in autonomous spacecraft habitat operations. In safety critical applications, it is absolutely necessary to

incorporate fault recovery techniques as the loss of life risk can be extremely high. The fault management system is comprised of state estimation, fault detection, and a fault impacts reasoner, all which are modules within the Core Flight System, the operating system used in space applications. The state estimation and fault detection modules work together to identify off nominal conditions, and the reasoner performs diagnosis and identification of the fault magnitude. This information is then fed to a higher level control which performs recovery, where the actions are altering the operating parameters of the affected system. These systems are therefore *fault-tolerant*, i.e., they can continue to operate in the face of faults.

Most fault tolerant systems employ some form of *Fault-Adaptive Control* or *Fault-Tolerant Control* (FTC), which are control mechanisms that allow the system to operate within acceptable levels of performance or safety while under faulty conditions. Multiple FTC methods for UAV systems are presented in (Zhang et al. (2012); Ahmed et al. (2023)). These approaches focus on low-level control signals applied to the motors and are used to maintain a safe trajectory. While these strategies allow the system to continue to operate under faulty conditions, they are not the same as *fault-aware replanning*. The primary difference is that FTC is concerned with faults that affect *how* the system operates and interacts with the environment at the control-loop level, without any reference to remaining operating time. Fault-aware replanning is concerned with faults that affect how long the system can continue to operate before catastrophic failure is reached.

Kuhn (2011) presented a framework for integrating continuous diagnosis with replanning for information gain to ensure continuous operation, depicted in Figure 2.9. The central concept is that the diagnosis engine gains information by the outcomes of executing a certain plan to achieve a given objective, where the objective can be achieved in multiple ways. If there is a fault in the system then the outcome will be undesirable, and therefore the diagnoser tells the replanner to achieve the objective using a different plan. This results in further information gain for the diagnoser, which in the case of continued undesired outcomes, gets more and more specific with how the replanner should alter the operating parameters to achieve the desired outcome.

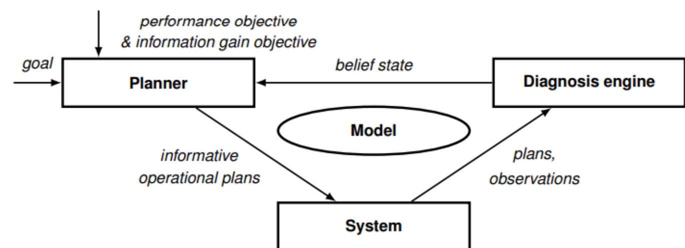

Figure 2.9: Integration of operational planning with active diagnosis (Kuhn (2011))

While on the surface this looks exactly like what we seek to achieve, the devil is in the details. Such an approach is not feasible for safety critical systems, where the undesirable outcome is usually the total loss of equipment or the loss of life. This approach requires that the end state is reached, and that system is performing a non-critical repetitive tasks.

Xu et al. (2022) introduced a novel approach called "Isolate and Repair Plan Failures" (IRPF), particularly tailored for spacecraft systems which consist of multiple subsystems including attitude control, communication, and data management. In case of a malfunction in one subsystem, the IRPF method isolates defective actions unsuitable for the current operational environment and then repairs a segregated set of actions while the remaining executable parts of the plan continue. These repairs can later be integrated at a suitable time to formulate a new complete plan, thereby aiding in achieving the mission objectives more efficiently than waiting for ground control to develop a new plan. The IRPF method essentially categorizes a spacecraft plan into three segments: completed, executable, and defective sections. It focuses on accurately identifying and isolating unfeasible actions and handling potential conflicts between repair processes and ongoing operations, which are often resource-intensive and might clash with the resources required for repair, causing disruption and complexity in determining the optimal start point for the repair initiative. This approach most closely aligns with what we refer to as fault-aware replanning. It explicitly involves the identification and isolation of "defective" or "invalid" actions - indicative of faults in the initial plan - and incorporates mechanisms to repair these within the broader operational framework. It's "aware" in the sense that it not only identifies faults but actively seeks to recover from them in real time.

However, this approach does not account for the health of the individual components and the subsequent health of the overall system after the fault occurs. This is what is meant by *Health-Aware*, where component and system *State of Health* (SOH) information is used to generate safer and more reliable plans when faulty conditions occur during a flight. The literature is absent of replanning examples that incorporate system SOH and component degradation information in conjunction with faulty operating conditions. Such an approach was first presented in our previous work (Darrah et al. (2023)), but it was a very limited study that only included a single example to demonstrate the approach and did not fully capitalize on the SOH information available. Moreover, the approach implemented an A* algorithm, which can be intractable for large search spaces, and, the ideal approach would utilize an anytime algorithm instead.

CHAPTER 3

Overview of Approach

The overall approach to deriving a health-aware framework for replanning is comprised of four key steps and is depicted in Figure 3.1. The first step involves simulation and data generation; the second step involves building a deep learning model for RUL estimation; model reduction takes place in the third step; and replanning in the fourth. The details of the approach are described in the subsequent subsections.

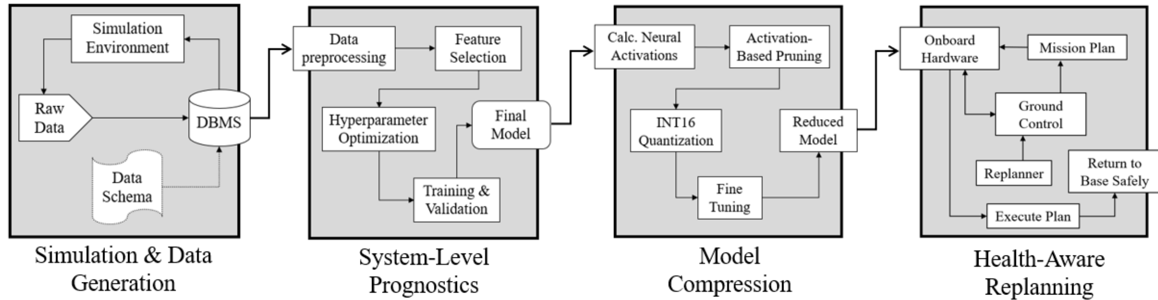

Figure 3.1: Overall approach of the thesis.

3.1 Simulation & Data Generation

Developing the simulation environment and data management framework is a necessary precursor to all other tasks in the thesis. Simulations are needed for a wide variety of reasons including faster development time and data generation, as in the case of this work. We don't purposely run systems to failure for data collection, and run-to-failure data is needed to generate RUL models. Therefore, simulations are necessary. The simulation environment is comprised of a system model of the Tarot T-18 UAV, degradation models, the environment (wind and flight area of operation in an urban setting), and various trajectories. These are discussed in Chapter 4 (introduction) and Section 4.1.

The raw data generated from these simulations must be organized to be effectively used for prognostics applications, and, therefore, a data schema and underlying database management system (DBMS) must be developed. This is discussed in Section 4.2. The stochastic simulation of multiple UAVs flying multiple

missions and different trajectories, under varying operating conditions until failure is reached, results in the system-level prognostics dataset. This is discussed in Section 4.3, with some additional figures and descriptions in the Appendix.

3.2 System-Level Prognostics

Chapter 5 discusses data-driven system-level prognostics. The core of the work is shown in block 2 of Figure 3.1, system-level prognostics. Prognostics theory (not depicted in the figure) is presented in the chapter introduction. The key steps in developing the data-driven RUL model include data preprocessing, feature selection & engineering, hyperparameter optimization, and training & validation. These steps are presented in Section 5.1. As the data is already organized and structured, standard data preprocessing steps, such as time alignment or handling missing values can be ignored. Feature selection methods specific to prognostics and regression tasks are discussed, as well as an Autoencoder for feature extraction/dimensionality reduction.

The two-step hyperparameter optimization process follows, whereby the network parameters are optimized first, followed by the regularizing parameters. Then the training procedure is detailed. The training results are presented in Section 5.2. Section 5.3 shows how this information enhances other estimation tasks such as segment-based flight time and power consumption estimates, and remaining flight time. The replanning agent uses this information (discussed in Chapter 7), and an overview of Chapter 7 is provided below in Section 3.4.

3.3 Model Compression

The third key step, shown as the 3rd block in Figure 3.1, Model Compression, is one of the primary contributions of the thesis. This is where the temporal neural activation patterns are calculated (see Section 6.1) based on telemetry from the late stages of life of the UAVs in the training set. This is then used to inform the hybrid structured-unstructured pruning process called *activation-based pruning* (see Section 6.2), which is the developed approach tailored towards time-series regression problems.

Experiments are conducted that show in addition to activation patterns, preservation of the cell-state weight matrices further improves the model accuracy. This is a modification is specific for LSTM networks. Different

pruning methods are compared, along with the effects of quantization, and the results are shown in Figure 6.3.

3.4 Health-Aware Replanning

The thesis work culminates in Chapter 7, where the framework for health-aware planning is presented. This framework incorporates a reduced system-level prognostics model that runs continuously onboard, and a replanning agent on the ground control station that is activated if a fault occurs during flight. The bridge between replanning and system-level prognostics is discussed in Section 7.1, where *health awareness* is formally defined. The replanning framework is presented in Section 7.1.1, and is formulated as a modified *Orienteering Problem*, discussed in Section 7.2.

An adaptive fast-start genetic algorithm is developed and presented in Section 7.3. The algorithm is *adaptive* by means of altering the crossover, mutation, and elitism rate based on performance over each iteration. In addition, a restart feature allows it to reinitialize the population if it is not improving and the target fitness has not been reached. A stopping feature halts if there is no improvement and the target fitness has been reached. The algorithm has a *fast-start* modification by incorporating domain knowledge into the population initialization function such as the trajectory, waypoints, and visiting order. The experiments and results in Sections 7.4 and 7.5.

CHAPTER 4

Simulation Testbed and Data Generation

A well-organized approach to planning and reporting simulation studies involves formally defining the purpose of the study, enumerating the input-output model, defining the parameters that are being estimated, the methods used, and the performance measures (Morris et al. (2019)). *This chapter discusses the groundwork in simulation and data generation necessary to develop system-level prognostics technologies.* Typically, the literature puts a lot of emphasis on the purpose of the study and the methods used, with little attention given to properly documenting the component modeling, simulation environment, and the data these components generate. Simulation-based and data-driven environments have grown to a level of complexity where managing the generated data becomes a burdensome task fraught with errors. Therefore, it is imperative for system health management platforms to adopt a robust and well-defined framework that focuses on the data management aspects as much as the simulation model itself.

Modeling and simulation tools such as Simulink, LabVIEW or Modelica represent industry standards for a number of engineering applications. They also facilitate the generation and use of data. However, these tools do not enforce any policies or rules on how the generated data is managed. These are *open-ended* tools, meaning there is no framework to specify and implement such policies or interfaces, it is all left to the user. Given the skill-set disconnect between engineers or research scientists, the users of these tools, and the database engineers or the enterprise software developers who typically implement data schemas, the data management practices and implementations needed to facilitate experimentation often fall through the cracks. Therefore, for the purposes of this research, a simulation environment that includes an underlying data management framework is developed to support system-level prognostics research of UAVs.

These simulations are needed because we don't have real-world run-to-failure data, and high-fidelity simulations in conjunction with a well-orchestrated data management scheme can provide this. This framework provides the context for developing data-driven machine learning models that support the development of *health-aware* technologies, which includes methods for computing RUL, and making safe decisions, which includes AI-based methods such as replanning. Monte-Carlo stochastic simulations are conducted using this environment to generate operational data for UAV systems with multiple degrading components flying a

number of trajectories that are subjected to varying wind conditions. This simulation environment and data management framework facilitate the generation, curation, usage, and storage of simulation data that encompasses multiple operating and environmental conditions to enable studies with multiple machine learning algorithms for the development of health management technologies. This simulation environment has been open-sourced and released to the public under the GNU General Public License.

The testbed is designed to support stochastic simulations given the system model along with a set of component and degradation models that make up the system; an environmental model (i.e. wind); a series of inputs (i.e. mission profile); and performance threshold functions. The simulation environment implements a set of MATLAB® scripts that run in parallel to take advantage of multi-core CPU architectures without requiring any specialized parallelization computing libraries. The stochastic simulation of the UAV system and environment uses input/model data from several different sources, shown below in Table 4.1. The torque-load relationship model was derived via polynomial fitting of test data obtained from a publicly available dataset. The aerodynamics, DC motor, and continuous battery models were adapted from previous publications (Osmić et al. (2016), Plett (2015), Valavanis and Vachtsevanos (2015), Darrah et al. (2020), Quiñones-Grueiro et al. (2021)). The battery degradation model came from test data obtained from NASA's data repository. Motor degradation models were developed by experimentation using reasonable assumptions on how long the motors are expected to last based on manufacturers' datasheets. Wind gust models for typical urban environments were generated similarly (Patrikar et al. (2020)), with force vectors applied directly to the airframe.

data model	**source**	**reference**
torque/load dynamics	propeller dataset	Advanced Precision Composites (2019)
aerodynamics	publication	Osmić et al. (2016)
DC motor	publication	Schacht-Rodriguez et al. (2018)
continuous battery	publication	Plett (2015)
battery degradation	NASA data repository	Saha and Goebel (2007)
motor degradation	experimentation	Gorospe et al. (2017)
wind gusts	experimentation	Patrikar et al. (2020)

Table 4.1: Data sources for stochastic simulation

An input-output diagram of the simulation is shown in Figure 4.1. A single execution of the simulation, i.e., a *single flight process*, is shown in Figure 4.2. First, the workspace is initialized by connecting to the database and loading information such as available trajectories, system performance thresholds, and initial conditions (1). Other simulation parameters, such as the controller gains for the PID controllers are also specified. This data is stored in the base workspace of the calling process, where parallel processes have

their own information and do not share data. Next, an existing UAV model is loaded from the database or a new UAV model is created (2). If multiple UAVs are part of the system-level prognostics study, one can be selected by serial number or ID. Then, a trajectory is randomly assigned to the UAV from a set of predefined trajectories (3).

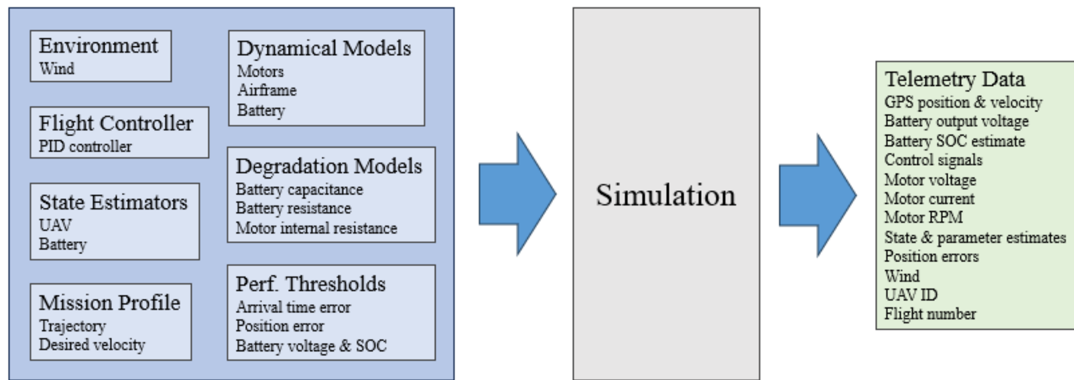

Figure 4.1: Simulation Input-Output Diagram.

Once steps 1-3 is complete, all of the necessary inputs have now been acquired and the flight can now be simulated (4). During simulation, a total of 83 features are collected at a rate of 1Hz. After flight termination, the degradation parameters for the UAV model are updated (5) and the system is evaluated to determine if it is safe to continue flying missions or if maintenance needs to be performed (6). The flight can either end successfully, or end with a *stop code* (i.e., a performance parameter violation), and this information is logged. Stop codes are tied to components and inform which maintenance action needs to take place, if any. At the end of every simulation run (i.e., a mission), the battery is recharged (7). Charging cycles also add to battery degradation just like the battery discharging process, but the rates of degradation for these two processes are typically not the same. This is captured by a parameter whose value ranges between 1 (same effect as discharging) and 0 (no effect). In this work, it is set to 0.75. The telemetry data, degradation data, and component usage data are stored in the database (8).

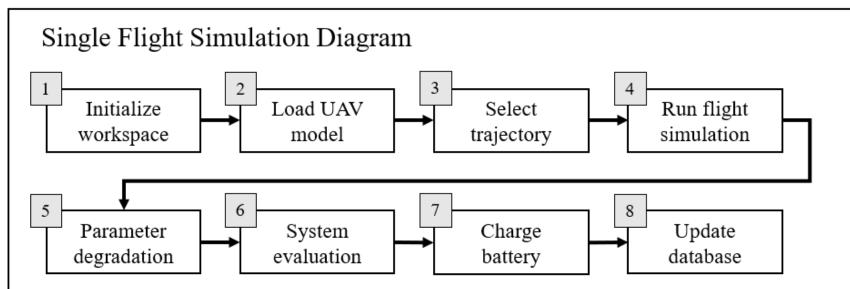

Figure 4.2: Single flight simulation Process

4.1 System Description

The system used in our experiments is an octocopter UAV modeled using parameter values provided in (Osmić et al., 2016). Table 4.2 lists the variables of the system and their descriptions. The power-train system is derived from continuous processes that are interspersed with discrete mode changes due to software-enabled control. This results in complex, nonlinear operational behaviors, which also produce complex, nonlinear component degradation profiles. The octocopter is formed by different components, which simultaneously have an effect on overall system performance. For example, the degradation of a motor affects the resulting generated forces and moments (F_M and M_M) over time, which affects the ability to follow a reference trajectory. Trajectory tracking is also affected by sensor noise from the GPS, which is modeled after (Abeywickrama et al. (2018)), and consumes an average of 690mA with an average power consumption rate of 8.26Wh. A degraded battery affects the downstream electronics and can result in unstable sensor readings, reduced thrust, or complete failures. These points and a complete system description are discusse below.

Variable	Description
F_b	force applied to body frame
F_M	force generated by the motors
F_D	drag force
ω_i	angular velocity of the i_{th} motor
$\dot{\omega}_i$	angular acceleration of i_{th} motor
i_{mi}	current demand from i_{th} motor
v_m	input voltage to the motors
i_T	total current demand from entire system
i_d	internal diffusion current
v_h	hysteresis voltage
v_out	output voltage
v_ocv	open circuit voltage
$s(i_T)$	sign of current
z	battery state of charge

Table 4.2: Dynamical variables of the system

The five major blocks of the system-environment simulation are navigation, control, powertrain, dynamics, and environment, as shown in Figure 4.3. Initially, values for position and velocity are provided in the UAV's start state in a continuous space 3-D world. A reference trajectory is derived offline before the flight and supplies the **navigation block** with coordinates to fly to. The GPS model adds position and velocity noise (with a bias of 0 as we are not modeling sensor faults) and supplies the **control block** with the reference position, true position, true velocity, GPS position, and GPS velocity. The control block calculates the position errors

and uses PD controllers for orientation and angular velocity. It outputs an N-dimensional array of voltage reference signals; in the case of an octorotor, N = 8.

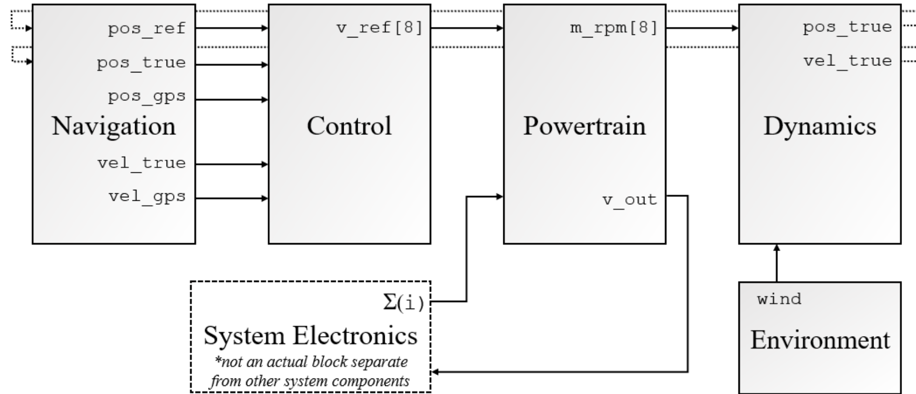

Figure 4.3: Simulation Model

The voltage reference signals are fed to the **powertrain** block, which contains the motors and battery models. The second input to the powertrain block is the current demand from all other components in the system (such as the GPS). The output of the powertrain block is a vector of motor RPMs that is sent to the **dynamics** block, which implements the aerodynamic equations of the UAV. The **environment** block is also an input to the dynamics block, where external factors such as wind gusts are translated to forces that are applied to the airframe. The output of the airframe block is updated position and velocity in the inertial axis, which is sent to the navigation block, and the flight control process is repeated.

It is important to also discuss the sources of uncertainty and noise, as well as the random variables in the simulation. Noise is attributed to processes and measurements within the system, system components, and environment. Measurement noise for a given parameter is modeled as an additive value. Uncertainty is modeled similarly and is captured in the confidence intervals for a given random variable, such as for the component degradation models. For each degradation parameter, Q, R_0, and R_m, we use a standard deviation of 2% of the mean value. Noise associated with the position and velocity measurements are modeled as $N(0,\sigma)$ distributions with σ values of .2 meters and .02 meters per second, respectively. Noise and uncertainty are also captured in the battery state estimator mentioned above, and for a more in-depth review we direct the reader to (Merwe et al. (2004), Darrah et al. (2021)).

Another source of uncertainty is the variation in the fully charged battery voltage for each charging episode. This is modeled as a skewed distribution in Figure 4.4 with approximately 90% of the values falling between 21.2 and the nominal voltage, 22.2, and 10% of the values fall between 22.2 and 22.6. This effectively models

what is experienced in the real world for charging batteries: sometimes they overcharge or undercharge, and usually, the final charge value is within some range of the rated output voltage value.

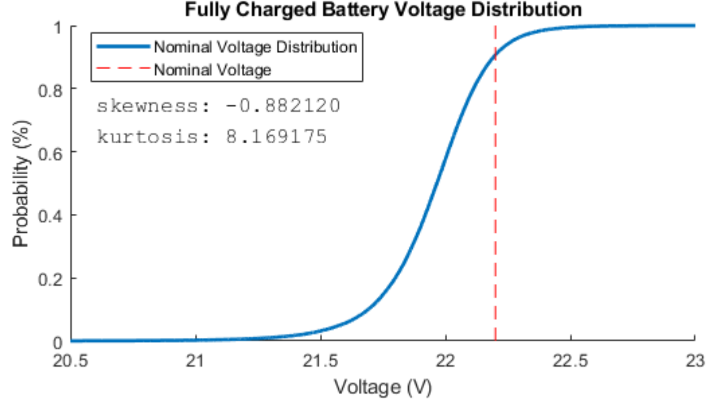

Figure 4.4: Nominal voltage distribution.

The effects of the simultaneous degradation of eight motors and the battery pack are also modeled. Specifically, stochastic degradation profiles are generated for three parameters: the equivalent electric resistance of the motor coils (R_m), the battery charge capacity (Q), and the battery internal resistance (R_0). Altogether, the degradation profiles describe the degradation parameters. We empirically derived degradation profiles for three parameters considered through run-to-failure experiments as described in (Saha and Goebel, 2007; Jackey et al., 2009) (battery), and (Xuan et al., 2017) (motor). Degradation is discussed in more detail in the following sections.

4.1.1 Airframe Dynamics

The Newton-Euler equations of motion for a rigid body dictate the dynamic behavior of the system as follows(Valavanis and Vachtsevanos, 2015):

$$\dot{\mathbf{v}}_B = m^{-1}(F_M + F_D) + gR_{IB}\mathbf{e}_z - \mathbf{w}_B \times \mathbf{v}_B, \tag{4.1}$$

$$\dot{\mathbf{w}}_B = {J_B}^{-1}((M_M) - \mathbf{w}_B \times J_B\mathbf{w}_B), \tag{4.2}$$

where $\mathbf{v}_B$ and $\mathbf{w}_B$ are linear and angular acceleration in the body frame, respectively; $F_M \in \Re^3$ is the resultant force generated by the motors, $F_D \in \Re^3$ is the drag force resulting due to the movement of a UAV through the air, $M_M \in \Re^3$ is the moment generated by the motors, m is the mass of the UAV, J_m is the inertia matrix of

Parameter	Desc	Value
mass	total mass	10.66kg
Jb	inertia matrix	$\begin{bmatrix} .2506 & & \\ & .2506 & \\ & & .4538 \end{bmatrix} kg-m^2$
A	cross sectional area	$\begin{bmatrix} 1.6129 & & \\ & & .508 \\ & .508 & \end{bmatrix} m^2$
l	arm length	0.635m

Table 4.3: Airframe Parameters

vehicle, g is the acceleration due to gravity, $R_{IB} \in \Re^{3\times 3}$ is the rotation matrix from the inertial frame to the body frame, and $\mathbf{e}_z = [0\ 0\ 1]^T$. The rotation matrix is calculated based on the Euler angles $[\phi, \theta, \psi]$ (Mahony et al., 2012).

4.1.2 Battery Dynamics

The battery is modeled using an equivalent circuit representation similar to ((Plett, 2015), Darrah et al. (2020)), with dynamic equations that characterize the battery behavior given by:

$$H = e^{-|u\eta\gamma\frac{dt}{3600}Q|} \tag{4.3}$$

$$\dot{h} = Hh + (H-1) * sign(u) \tag{4.4}$$

$$RC = e^{\frac{-dt}{|RC|}}{}^T \tag{4.5}$$

$$\dot{i_d} = RCi_d + (1-RC)u \tag{4.6}$$

$$\dot{z} = z - \frac{\eta Qu}{3600} \tag{4.7}$$

$$V_{out} = V_{ocv} + Mh + M_0 * sign(u) - Ri_d - R_0u; \tag{4.8}$$

where H and $\dot{h}$ represent hysteresis, which is the lagging effect where the charging and discharging behavior of the battery exhibits different voltage levels for the same state of charge. Q represents the total charge capacitance, u represents the input current, RC represents the diffusion resistance and capacitance, i_d represents the current going through the diffusion resistance, z is the state of charge, R and R_0 are internal resistances, and M and M_0 are polarizing constants. The parameters for the battery modelled are given below in Table 4.4.

Parameter	Description	Value
$Q*$	total charge capacity	22000
η	coulombic efficiency	.9929
γ	voltage decay constant	163.441
M_0	polarization constant	.0019
M	polarization constant	.0092
RC	Warburg impedance	14.25
R_0*	internal resistance	.0011
R	diffusion resistance	$2.83e^{-4}$
V_0	initial voltage	$22.2v$

Table 4.4: Battery Parameters
** degradation parameter*

The battery charge capacity is the amount of charge available in a fully charged battery. Battery charge capacity degrades as the battery undergoes multiple charge-discharge cycles over time. This can be attributed to the internal chemical processes within the battery as well as environmental factors, such as temperature. The degradation rate for Q is also a function of power delivered to the loads. Therefore, fast discharge of the battery ages the battery faster than a slow discharge. The charge cycle also causes aging effects and degradation, however, the degradation rate is lower during charging than it is during discharging. The battery internal resistance (R_0) also increases over time caused by Lithium corrosion, plating, and electrolyte layer formation (Daigle and Kulkarni, 2013). Consequently, this increases the current drawn from the battery, which, in turn, affects Q and causes the voltage delivered by the battery to drop. Figure 4.5 depicts several degradation profiles for internal resistance (R_0) and charge capacitance (Q). When a new battery is created, it is randomly assigned one of each profile.

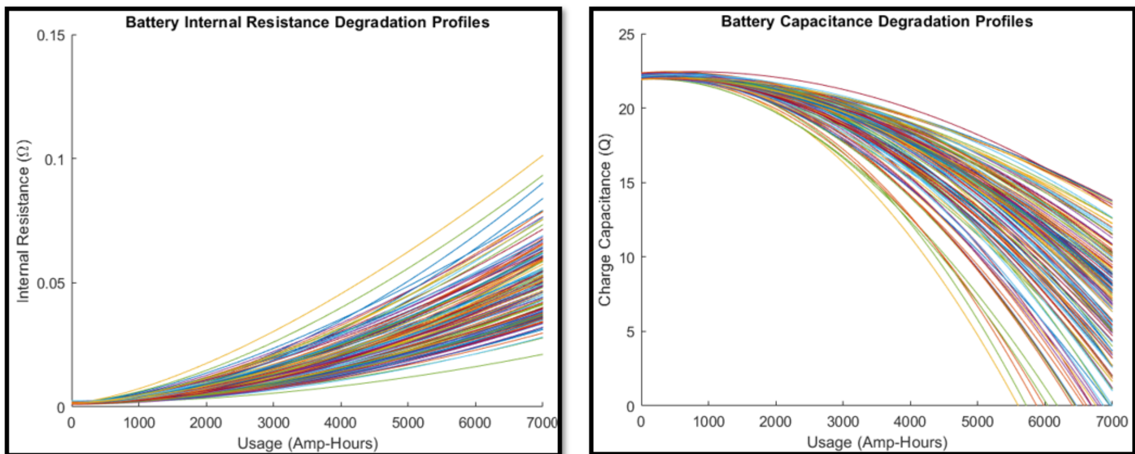

Figure 4.5: Degradation profiles for battery internal resistance (R_0) (L) and charge capacity (Q) (R)

$$Q = -c_1 x^{c_2} + c_3 x + c_4, \textit{ where} \tag{4.9}$$

$$c_1 \sim \mathcal{N}(1300, 50)$$
$$c_2 \sim \mathcal{N}(1.8, .075)$$
$$c_3 \sim \mathcal{N}(1300, 50)^{-1}$$
$$c_4 \sim \mathcal{N}(21.95, .25).$$

$$R_0 = c_1 x^{c_2} + c_3, \text{ where} \tag{4.10}$$
$$c_1 \sim \mathcal{N}(33500, 1000)$$
$$c_2 \sim \mathcal{N}(1.5, .1)$$
$$c_3 \sim \mathcal{N}(.0014, .0005).$$

4.1.3 Motor Dynamics

Brushless DC motors commonly used in UAVs can be modeled as generic DC-motor models, as opposed to three-phase AC models. We adopt this approach and Equations 4.11 and 4.12 describe the dynamic behavior of each motor i:

$$\dot{\omega}_i = \frac{1}{J_m}(K_e i_{mi} - T_p - D_f \omega - T_f), \tag{4.11}$$
$$i_{mi} = \frac{1}{R_m}(v_m - K_e \omega_i), \tag{4.12}$$

where $R_m = \frac{2}{3}\sum_{j=1}^{3} R_j$ is the equivalent electric resistance of the motor coils j of a three-phase motor, K_e is the back electromotive force constant, ω_i is the angular velocity of the ith motor, T_f is the static friction torque, D_f is the viscous damping coefficient that allows to estimate the dynamic friction torque ($D_f \omega$), J_m is the inertia of the motor, v_m is the input voltage control signal, i_c is the current demanded from the battery pack, and T_p represents the torque load generated by the propellers.

Parameter	Description	Value
R_m*	equivalent motor resistance	.27
Ke	back EMF	.0265
T_f	friction torque	$1e^{-8}$
D_f	viscous dampening	$1e^{-8}$
j_m	inertia	$5.0e^{-5}$
c_t	rotor thrust coef.	$2.2144x10^{-4}$
c_q	rotor drag coef.	$1.601x10^{-8}$
c_d	translational drag coef.	$1.8503x10^{-6}$

Table 4.5: Motor Parameters
** degradation parameter*

Motor faults can be classified as originating internally or externally falling into the categories (Bazurto et al. (2016)): Inherent weakness of material, design and manufacturing; misuse; gradual deterioration as a result of wear and tear and corrosion. Faults relating to bearings, a mass unbalance, and a low insulation resistance can result in motor loss in effectiveness, which may translate into reduced controllability or stability.

Motor coil resistance is used as a proxy for modeling the loss of performance of the motors, which may be attributed to factors such as continued use over time and exposure to adverse weather conditions. As a result, a throttle increase is required to compensate for these effects, thus increasing the battery's rate of drainage. In previous experiments Darrah et al. (2020, 2021); Quiñones-Grueiro et al. (2021), we only considered the degradation of one motor, which resulted in a consistent position error deviation with respect to the direction of flight and the direction of the wind. In real life, all of the motors can degrade simultaneously, and this would result in unpredictable trajectory deviations. In previous experiments (unpublished), the UAV underwent three battery changes before the effects of motor degradation was apparent. However, it is known for motors to last for several thousand hours of operation, and therefore the degradation profiles were updated to reflect real-world expectations. Figure 4.6 depicts 50 different randomly generated motor degradation profiles, one of which is clearly an outlier, yet no less valid than any other.

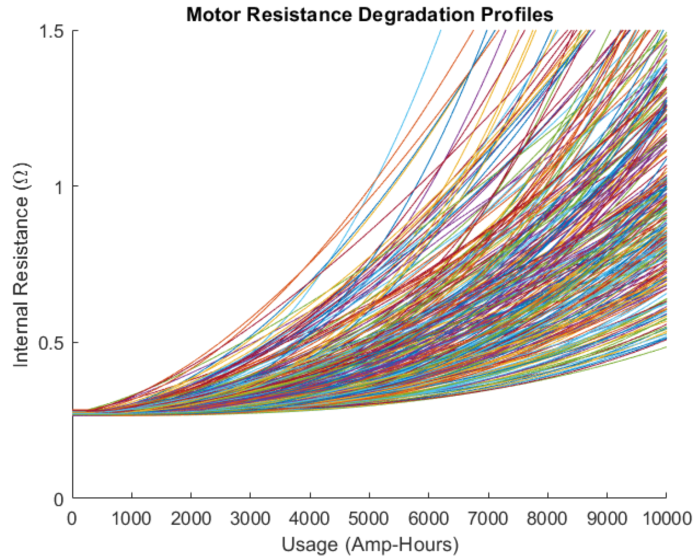

Figure 4.6: Motor Degradation (R_m)

$$R_m = c_1 x^{c_2} + c_3, \ where \tag{4.13}$$
$$c_1 \sim \mathcal{N}(13000, 1200)$$
$$c_2 \sim \mathcal{N}(1.7, .3)$$
$$c_3 \sim \mathcal{N}(.27, .005).$$

4.1.4 System Performance Parameters

System-level performance parameters are *random variables* computed from system variables, and they are a *health indicators* for system SOH. For the UAV, overall system performance is a function of the SOH of its components, namely the battery, ESCs, and motors. A Sigma Point Kalman Filter Merwe et al. (2004) is used to perform battery (state and degradation) parameters estimation, and resistance measurements with additive noise are used to assess the motor (degradation) parameter. There are no direct measurements or models of ESC failure that can be estimated, so this will be an abrupt fault that will have an effect on controllability. The system performance parameters are shown below in Table 4.6.

Parameter	**Name**	**Description**	**Value**
z	battery state of charge	impacts flight time and load capacity	30%
v_{out}	battery output voltage	impacts flight time and load capacity	20.9v
$\varepsilon_{\mu pos}$	average position error	impacts risk of collision	2.0m
ε_t	arrival time error	impacts risk of not finishing flight	8s

Table 4.6: System performance parameters

The first two parameters are the battery *State Of Charge* (SOC) (z) and the battery output voltage (v_{out}), the third is the average position error ($\varepsilon_{\mu pos}$), and the fourth is the arrival time error (ε_t) (see Table 4.6). At first glance, one might think the two battery-specific parameters are component-level parameters, as they specifically relate to the battery. This is misleading, however, due to the feedback loops among degrading components and the effects these components have on battery performance outside of the battery's operating characteristics and degradation in isolation. Therefore, these battery indicators are directly tracked, as they inherently provide information related to overall system performance. The performance parameters for the battery are computed from its degradation parameters, the internal resistance, R_0, and overall charge capacity, Q, which are indicators for battery SOH. Battery SOH in turn, contributes to overall system health. A similar comparison is made for the position error parameters; the combined interactions among multiple components degrading effect the UAV's ability to stay on its intended course, not just motor degradation (R_m). Together, these four parameters provide the information needed to perform system-level prognostics.

The set of system performance parameter constraint functions ($\wp_i(y_n))$) are implemented as a set of linear temporal logic (LTL) equations (Eqs. 4.14 - 4.17) that specify certain states the system should never reach during flight, and if so, appropriate action should be taken. It is prudent to set the threshold values for these parameters *below* that of actual or unrecoverable failure. In order to tie back into the higher-level safety

goals our system should achieve, one must leave some buffer space to take action (such as immediate landing procedures, return to base, etc), if necessary, prior to actual failure. These functions together comprise $(T(\wp_i(y_n)))$ in Equation 5.1, the system performance threshold equation, defined in section 3.

Definition 2 *$\wp_z$, System Performance Parameter Constraint Expressions*

$$\Box\,(z > \ 30\%) \tag{4.14}$$

The battery state of charge shall always remain above 30%.

$$\Box\,(v_{out} > \ 20.9v) \tag{4.15}$$

The battery output voltage shall always be greater than 20.9 volts.

$$\Box\,(\varepsilon_{pos_\mu} < \ 2.0m) \tag{4.16}$$

The average position error shall always remain below 2.0 meters.

$$\Box\,(\varepsilon_t < \ 8s) \tag{4.17}$$

The arrival time error shall always be less than 8 seconds.

4.1.5 Trajectory Generation

The flight trajectory of the UAV (see Figure 4.7) plays a role in its overall safety and risk of failure during flight. Trajectories with sharp movements can also impose heavier impulse loads on components, which as discussed above, impact degradation rates. To study these effects in depth, the trajectories must be tracked and linked to other data objects, and incorporated into the database management system. In general, a trajectory is represented as a multidimensional array capturing a time series, where each axis ([0,1,2]) is the equivalent *x,y,z* axis in space. A *triplet* is a single slice across the matrix that represents a specific point in space where the UAV is located at a specific time. Trajectories can be grouped into sets based on distance, estimated flight time, risk, or reward (although risk and reward are not considered in this work). Six trajectories are shown in Figure 4.7, there are over 50 trajectories available. Trajectories are formed in two steps using probabilistic roadmaps (PRM), followed by B-spline smoothing described below.

A *Probabilistic Roadmap* (PRM) is a graph structure where the nodes represent *collision-free* way-points and the edges represent a realizable path between these way-points (Kavraki et al. (1996)). Two parameters, *number of nodes* and *maximum connection distance*, must be supplied by the user. First, the graph is created by generating random nodes in the search space, and then checking if they are valid nodes. Then, each node is connected to every other node within the *maximum connection distance*, so long as that connection is also valid using the *k-nearest neighbors* method. Generating a graph is the first phase of the algorithm, which is executed off-line based on the navigation map. In the query phase, the start location, waypoints, and end location are connected to their closest nodes of the graph, and the path is obtained by using Dijkstra's shortest path algorithm. If there is no valid path (i.e., collision-free) from a supplied location to an existing node on the graph, then that location is deemed unreachable. Figure 4.8 depicts three scenarios of the trajectory generation process that show the outcome of different parameter selections.

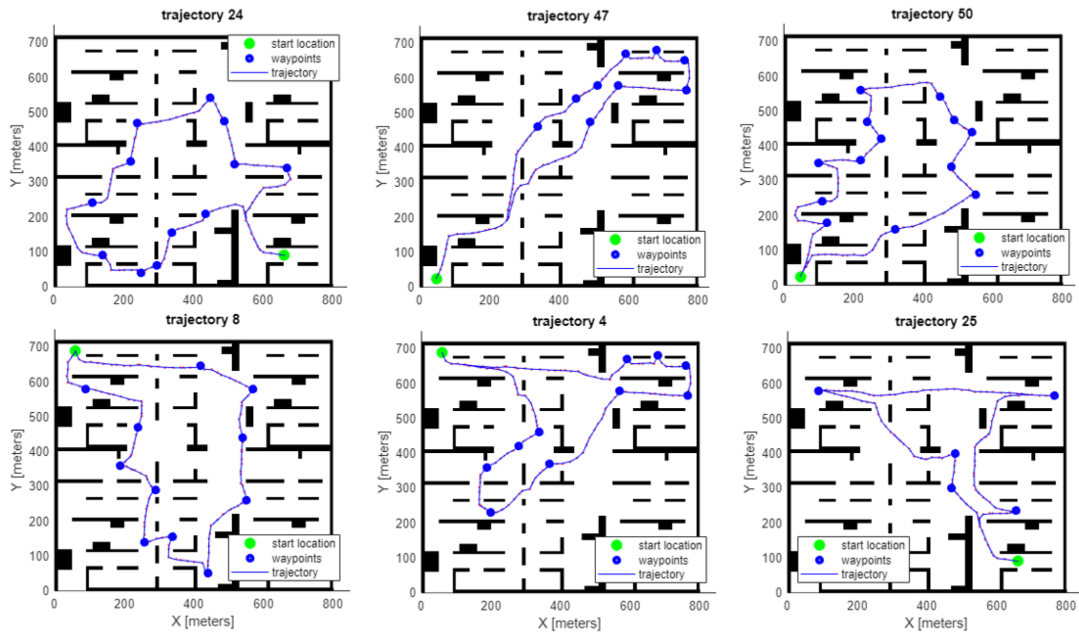

Figure 4.7: Available Trajectories

Piecewise B-spline curves of degree k (Magid et al. (2006)) are used for generating a smooth trajectory based on the way-points derived by the PRM method for a desired cruise speed. This method has been widely used in robotic applications because of its desirable properties of convex hull and maintaining continuity up to the $k+1$ derivative for a curve of degree k. Specifically, we considered cubic B-splines (Farin (2014)) with a desired cruise speed of 1.0 m/s for the UAV flights. This ensures that any hard pivot points in the trajectory are curved for optimal performance with respect to energy consumption, flight time, and control.

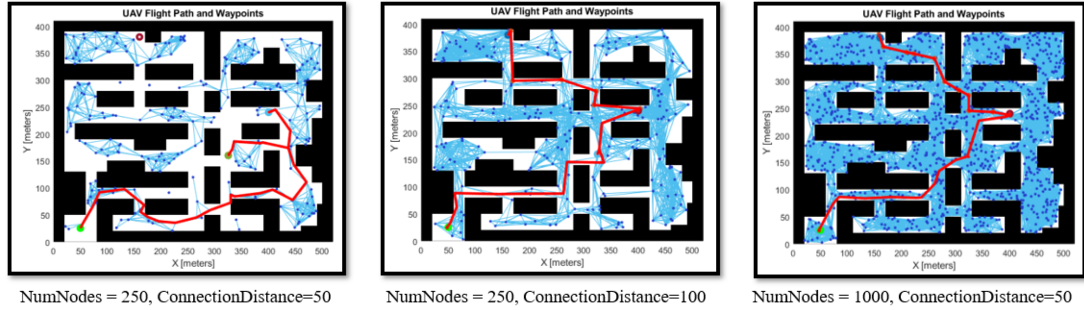

Figure 4.8: Trajectory generation. (L) too few nodes have been supplied and the waypoint at (160, 390) cannot be reached. (M) the connection distance is increased but now it is clear that the flight path is not optimal. (R) enough nodes have been selected and the connection distance has been reduced, providing the optimal trajectory.

4.2 Data Management Framework

Data sets that lack an organizational structure do not provide effective support for health management applications. Useful data sets that support PHM analyses must conform to explicit rules for data creation, storage, retrieval, and update. They also need to include *Persistent and Unique IDentifiers* (PIDs/UIDs) for data assets. Data curation can become a tedious and error-prone task in the absence of a properly designed *Extract, Transform, Load* (ETL) pipeline. To address these concerns, we adopt an *object oriented design* methodology and bring together intersecting concepts from enterprise software development, data management, and engineering experimental design to establish our data management framework. We begin by outlining the key requirements and specifications of the framework, followed by an in-depth discussion of the key attributes and features.

4.2.1 Requirements

Design decisions are driven by requirements, which are themselves driven by higher-level goals. In this case, the goal is to simplify and facilitate system health management analyses, and in particular, SLP research. We have developed four general and five specific requirements to meet this goal, as outlined below.

R1 - Reproducibility. This represents the ability to reproduce experiments, and, therefore, supports a better peer-review process while establishing connections with other experiments. This is accomplished by explicitly specifying data, metadata, and asset organization. This is also an important requirement for validation tasks.

R2 - Explainability. This is directed towards making the specific methodologies employed for tracking

all of the influences explicit and transparent, from modeling approaches to algorithms used on the simulation and data generation process. Explainability is required to achieve proper *data provenance*.

R3 - Extensibility. The code base needs to be developed using a modular, decoupled approach behind well-defined interfaces that facilitate superficial changes and the running of new experiments without having to make substantial alterations to the underlying codebase or having to rewrite significant amounts of code. This makes it easier for engineers not familiar with data management practices to incorporate the framework into their experimental approaches. Overall, it supports wider adoption.

R4 - Maintainability. This is focused on ensuring that the codebase and data storage is consistent across multiple experiments. Maintainable systems also make it easier to track bugs and fix errors. Maintaining a consistent and modular code base makes it easier to run machine learning experiments that generate consistent and reproducible results. We do not deal with the software aspects of PHM applications (e.g., see the GSAP framework**?**), but these concepts apply to data management, and this requires the specification of a well-defined data schema.

R5 - System Composition. The framework should enable seamless composition of component and degradation models to construct the desired system models for simulation. This enables running a variety of prognostic experiments by pulling component and degradation models from a system library, instead of building models from scratch or trying to embed them in the executable code for every experiment.

R6 - Operational Variability. The framework should support multiple operational *cycles*, where a cycle is a single run of the system. Operational cycles are defined by assigning values to a set of parameters, such as the load setting for the charge-discharge cycle for a battery, and setting the waypoints to define an aircraft flight trajectory, or a sequence of deliveries for a UAV.

R8 - Data Validation. Automatic validation of the data is a key requirement that ensures the data conforms to explicit constraints and can be guaranteed to be correct, without overburdening the user. This allows researchers to focus more on their specific application, and less on the nuances of ensuring the data is correct.

R9 - Data Integrity and Availability. The framework should also guarantee accessibility, discoverability, and persistence of data. This should be accomplished with an application programming in-

terface (API) that exposes various methods for storing, retrieving, and updating records (among other functions).

A system designed to meet these requirements will allow for automated or semi-automated workflows that are *self-documenting* in terms of the data, metadata, and data provenance. Such a design pattern for data management within the context of PHM will improve the development pace and collaboration among researchers. Our data management framework adopts an *asset, process, and data* methodology to address these requirements. This framework captures and organizes high-fidelity simulation data to support the development, testing, and evaluation of health management applications. Processes are linked to assets, and both processes and assets are linked to generated data. This framework is designed such that it can be used for both simulations and real-world applications, and with other applications beyond UAVs. Any number and type of physical systems can be modeled as an asset, which is affected by internal and external processes, all of which generate data.

4.2.2 Assets

Assets are the tangible components that make up a system, in our example the system is a UAV, and the assets are the UAV itself and its components, i.e., the battery, motors, airframe, and a GPS sensor. Each asset acts as a *first class object*, meaning it is the archetype model that all components of the UAV inherit from. This includes many other types of components not listed above that may make up the UAV. All physical components are assets, including the UAV. This is perhaps one of the most fundamental concepts of the framework and central to interoperability among components, systems, environmental effects, and degradation models. By approaching our simulations and other experiments with a "data first" view, these principles unfold naturally, and many of the issues facing researchers today simply disappear.

All assets have an associated *asset-type* (with predefined asset-types given in Table 4.7), which holds metadata about the asset. The table name is `asset_type_tb`, and it contains the fields `id`, `type`, `subtype`, and `description`. The `type` property is the high-level component type, such as *airframe* or *battery*. The `subtype` property is used for further specification, such as whether or not the battery is a *discrete_eqc* (discrete equivalent circuit) or a *continuous_ec* (continuous electro-chemistry). There is a one-to-one mapping between an asset type and a data table for specific component information of that type. For example, asset-type with id 1 (table 4.7), has an associated table called `airframe_octorotor_tb`

that holds the model parameters for the airframe dynamics of an octorotor UAV. When a user defines a new asset type, an associated table is created automatically with the necessary fields required to work with the framework. The user can then specify additional fields relevant to their application. The `description` property contains information regarding the source of the *model* that is used by the *asset*, or any other contextual information about the model the user wishes to provide. For example, DC motor dynamics are well understood and simple models like the one used here can be considered *generic*. But it may be more appropriate to cite an author and year for more complex models, such as `plett_2015` for a specific battery model. Further asset-types might include *electronics* or *equipment*, among many other possibilities.

id	**type**	**subtype**	**description**
1	airframe	octorotor	osmic_2016
2	battery	continuous_eqc	plett_2015
3	motor	dc	generic
4	sensor	gps	generic
5	uav	-	-

Table 4.7: asset-types

An asset type must exist before an asset of that type can be created. This is stored in a table called `asset_tb`. Table 4.8 depicts the assets and necessary information for the UAV system used in this work. Required fields necessary to properly interface with the rest of the framework are `id`, `owner`, `type_id`, `process_id`, and `serial_number`. The `id` field is automatically generated and cannot be altered; `owner` and `serial_number` fields will automatically generate values if none are supplied (the current user and a random 8-digit hexadecimal value); the `process_id` is not required. However, if one is supplied it will be validated against an existing process ID; and finally, the `type_id` is required and the API automatically assigns the correct value for the given type. The `type_id` field of the `asset_tb` table is a *foreign key reference* to the `id` field of the `asset_type_tb` table. The relationship between the asset table and asset-type table is depicted at the bottom of Figure 4.10.

Four other tables are depicted in Figure 4.10 that store the model parameters for the asset types listed in Table 4.7. These are parameters of the component models and are specifically related to that component model type, a.k.a., *asset-_type*. There is a one-to-one relationship between a given model class and an entry in the asset-type table, however, there can be any number of model instances of the same model class. Each table (with the exception of `uav_tb`) presented in Figure 4.10 comes with default parameters that are provided from the information in the `description` field of the `asset_type_tb`. For the battery, we use the

model given by (Plett (2015)), and thus the default parameters for this model come from that source. Each table is linked to the asset table via foreign key relationships on the `id` fields, whereby the asset table entry must exist before the derived component model entry is created.

id	v_id	owner	type_id	process_id	serial_number	common_name
13	1	tim darrah	1	4	7428BD	tarot airframe
15	1	tim darrah	3	120,170	36E2ED	tarot motor
16	1	tim darrah	3	127,173	D1F51A	tarot motor
17	1	tim darrah	3	132,181	19BD01	tarot motor
18	1	tim darrah	3	136,188	14D674	tarot motor
19	1	tim darrah	3	143,195	E0E385	tarot motor
20	1	tim darrah	3	145,202	630386	tarot motor
21	1	tim darrah	3	158,206	0D44A2	tarot motor
22	1	tim darrah	3	164,212	52763C	tarot motor
23	1	tim darrah	4	NULL	C45FE4	tarot gps
24	1	tim darrah	5	NULL	ABC4A6	tarot t18 uav
14	1	tim darrah	2	7,9	224DA5	tarot battery

Table 4.8: Asset table for an octorotor UAV system.

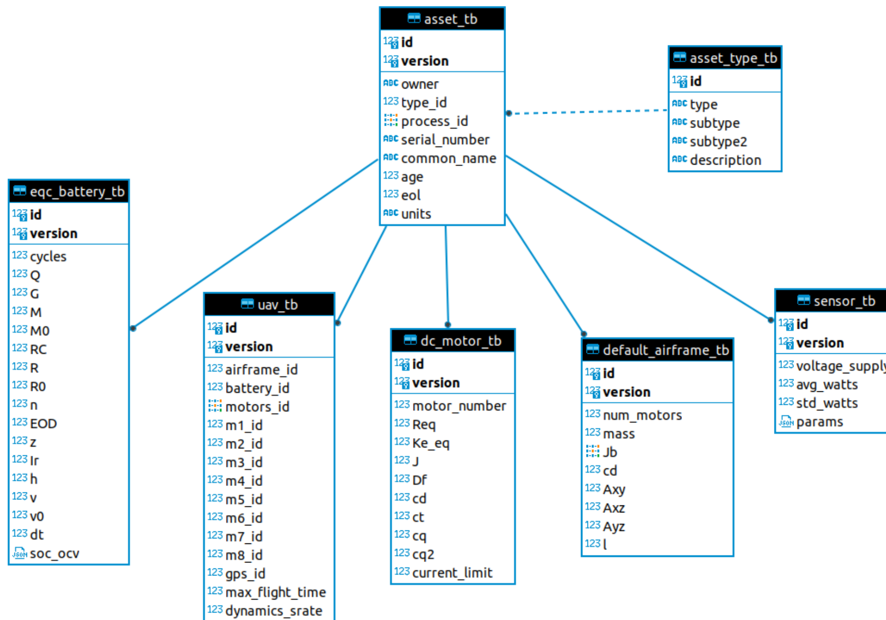

Figure 4.9: Asset fields and relationships for an octorotor UAV system

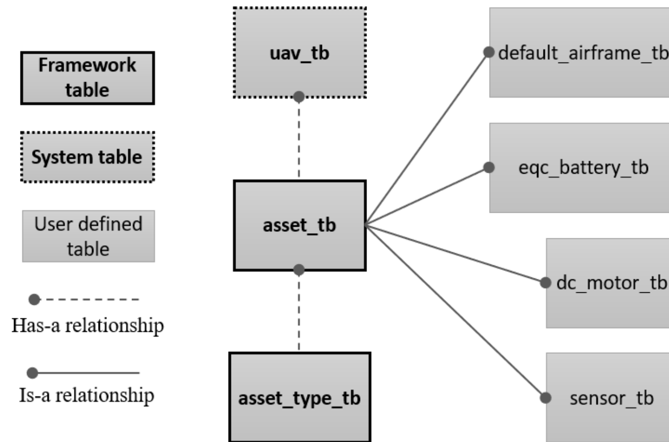

Figure 4.10: Relation Diagram (Assets) for an octorotor UAV system

This enforces a required constraint that is necessary to properly track a component that generates data to the model parameters are for that model, and it decouples the model definition from the source code. This decoupling between model parameters and the object representation they hold when used in simulation is another aspect that supports a robust data generation process and improves traceability. Finally, there is the `uav` type (see Table 4.7), a special type of asset that serves as a *container* to store asset IDs of the installed components and links to the process and environmental models. This is one of many concepts applied that provide modularity and data organization and serve as the common interface among all components. Correctly storing and extracting data generated from different sources and experiments can be very tedious and prone to errors. The metadata provided by the asset makes it easy to track the components with all other factors and sources of data within the simulation. This is a subtle, but very critical piece of the framework that is repeated with not just assets like motors and batteries, but with *processes*, like degradation or wind.

4.2.3 Processes

Processes capture dynamic interactions between the system and the environment. Many different types of processes can be modeled. In this work, we model internal component degradations and external wind gusts. Each component has its own set of degradation profiles and failure modes that are separate from, but affect, the overall operation and health of the system. Therefore, the framework must have the ability to track processes just like tracking components. In a similar manner to modeling components as *assets* having *asset types*, the *process* archetype also has *process types* (Table 4.9). There are five different process types shown, but there are nearly 300 variations of these process models stored in the database. Processes and

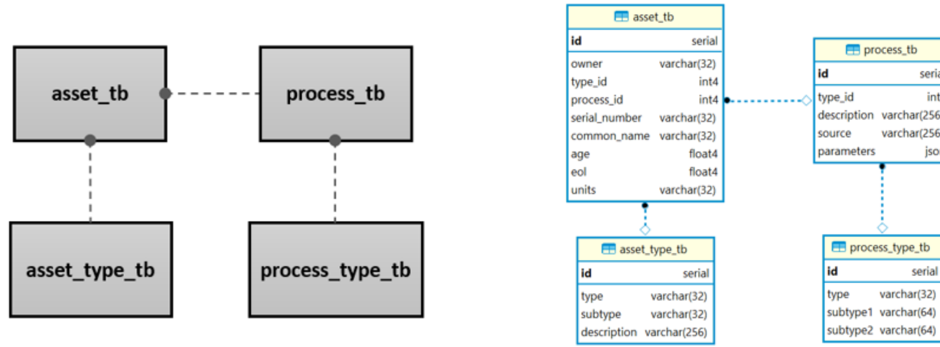

Figure 4.11: (a) entity relationship among processes and assets; (b) entity relationship with data fields and types.

assets are closely related as depicted in Figure 4.11. It is required that every process modeled be mapped to an asset, so as to track the relations between processes and components. This is something that is easy to lose sight of, and something as simple as a small parameter value change can have large differences in the resultant data. Therefore, processes and assets are tracked together. Each process has a foreign key relation `type_id` on the `process_type_tb` `id` field, which has a foreign key relation `asset_type_id` on the `asset_type_tb` `id` field. With these relationships and constraints established, the joint interactions between various processes and whole system performance is captured in the database *by design*, and requires no special attention by the researcher as long as the framework is implemented correctly.

id	type	subtype1	subtype2	asset_type_id
1	degradation	battery	capacitance	3
2	degradation	battery	internal resistance	3
3	degradation	motor	internal resistance	2
4	environment	wind	gust	1
5	environment	wind	constant	1

Table 4.9: Process Type Table

Having a well-defined foundation for managing assets also makes parallelization and performing Monte Carlo simulations easier. Multiple assets can be created to run the same code in separate instances, separate machines, or even in the cloud. Data driven experiments, this one included, are typically based on stochastic simulations and this flexibility simplifies planning, optimizing, and executing them. *"The devil is in the details"*, and correctly storing and extracting data generated from these experiments can be very tedious and prone to errors. This framework addresses these challenges simply through its inherent organization and use of an underlying database management system.

4.2.4 Data

Data is influenced by a multitude of factors and generated from a wide variety of sources within the simulation environment. We are especially concerned with components, degradation models, environmental models, and other internal and external events that generate information useful for prognostics applications. *Metadata* is just as important to capture as the raw data as well. Ensuring these complex interactions are captured and all relevant metadata and data from components, processes, the environment, and all other sources is the cornerstone of any data-driven CPS process, especially that of system-level prognostics. All data that is generated is inherently organized correctly when this framework is used in conjunction with a simulation environment. In this manner *the entire process of data generation, storage, retrieval, and analysis among flights with different components can be executed with the same code.* It is left to the individual practitioner to implement the dynamical models of their system; this framework handles everything else.

The data-driven model development lifecycle is depicted in Figure 4.12. A data schema provides standards, explicit rules, and organizational structure necessary for data management of PHM applications that support system-level prognostics. Data sets that lack organizational structure do not provide effective support for health management applications. Useful data sets that support PHM must conform to explicit rules for data creation, storage, retrieval, and update. They also need to include *Persistent and Unique IDentifiers* (PIDs/UIDs) for data assets. This is rarely ever done with file-based methods (*.csv*, *.txt*, etc.).

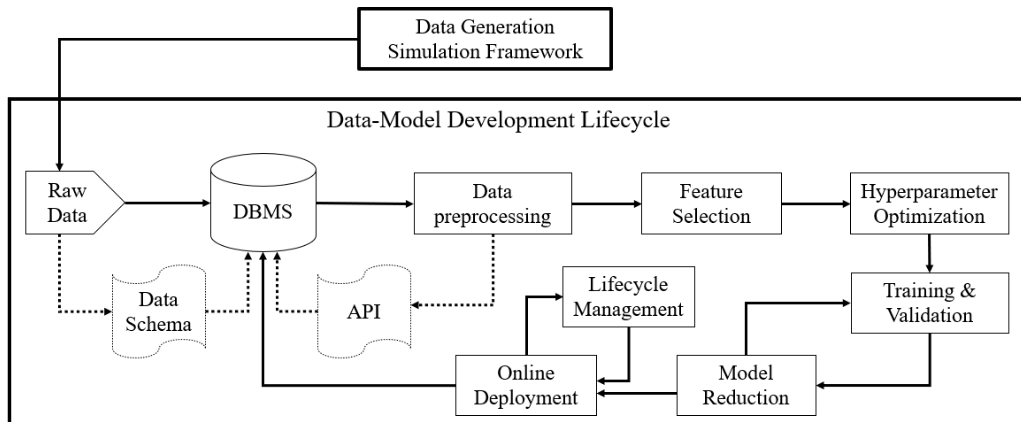

Figure 4.12: Data-Driven Model Development Lifecycle

Data curation can become a tedious and error prone task in the absence of a properly designed *Extract, Transform, Load* (ETL) pipeline. This data management framework adopts an *asset, process, and data* - centric methodology to capture and organize high fidelity simulation data to support the development, testing, and evaluation of PHM algorithms. The foundation lies in the data generation simulation testbed depicted at

the top and discussed in the previous section. The dataset that is generated with the simulation testbed and captured and organized with the data management framework, is discussed next.

4.3 System-Level Prognostics Dataset

The system-level prognostics dataset generation process is given in Algorithm 3. It takes as input **N** UAVs, where each UAV is comprised of a battery, **b**, and m motors, **M**, where in this case m=8 for an octorotor. Furthermore, a set of trajectories, **T**, an environmental model, **E**, a set of degradation models, **D**, and a set of performance thresholds, **P**, given in Definition 2 are the remaining inputs to the algorithm. At the beginning of each flight, a trajectory is randomly selected (line 3), and the wind conditions are sampled (line 4). The flight is then simulated (line 5), where the telemetry data (discussed below) is sampled at 1Hz. After the flight, the battery is charged (line 5), which is an important detail because battery degredation occurs during both charge and discharge cycles, and hysteresis is more accurately modeled this way. After this, the components degradation are updated based on their usage (line 7), and finally the dataset is updated with the new set of flight data (line 8).

Algorithm 3: System-Level Prognostics Dataset Generation

Input: **N** UAVs each consisting of a battery **b** and motors **M**; **T** trajectories; an environment model **E**; **D** degradation models; **P** performance thresholds

Output: ***D***, a dataset consisting of flight data ***d*** from each flight of **N** UAVs flown until failure

```
1  foreach UAV do
2  |  while eval(P) → ⊥ do
3  |  |  t ∼ T
4  |  |  e ∼ E
5  |  |  d ← simulate(UAV, e, t, P)
6  |  |  charge(B)
7  |  |  b, M ← update.degradation(b, M, D, d)
8  |  |  D ← D ∪ d
9  |  end
10 end
```

The composition of the UAV is fully described in Section 4.1. Since the battery and motors are the only components where degradation is modeled, they are the only components included in the algorithm for brevity. The environmental model comprises of stochastic constant wind with additive gusts, modeled as additive force on the airframe of the UAV, with an absolute range of $[0, 8]$ Newtons, given by the following equation

$$F_w \sim U_{const}(0,6) + \mathscr{N}_{gust}(.5,.\bar{1}), \tag{4.18}$$

where F_w is a 3x1 vector of the force of components on the x, y, and z planes; *const* represents the constant wind that remains the same throughout the duration of a given flight; and *gust* represents wind gusts in the three planes. There are 28 trajectories available to choose from, with a sample shown in Figure 4.7. The degradation models for the battery and motor are given in Equations 4.9, 4.10, and 4.13, and there are over 200 different degradation profiles generated from these models. Each component is randomly assigned a specific profile at the time of creation.

When a performance parameter threshold is violated, the UAV must undergo some type of maintenance action to keep flying. The first time this experiment was conducted, the effects of motor degradation was not noticeable. over 95% of failures were attributed to low SOC. Therefore, the battery was replaced, meaning new batteries were created and assigned to each UAV. In addition, the motors were artifically aged an amount equal to approximately 3 times their age after the first experiment. For example, the average motor age after the first experiment was between 140 and 180 amp-hours, and the age of each motor was subsequently sampled from a uniform distribution between 500 and 600 amp-hours. With a new battery and aged motors, a second experiment was conducted, the one in which the dataset collected is used in this work, summarized in Table 4.10. Table 4.11 shows the UAVs, number of flights flown, the failure code, and the performance threshold that was violated.

Number of Records	Number of Flights	Number of UAVs	Hours of Operation
7,816,430	5,989	75	2,171

Table 4.10: Summary statistics for the system-level prognostics dataset.

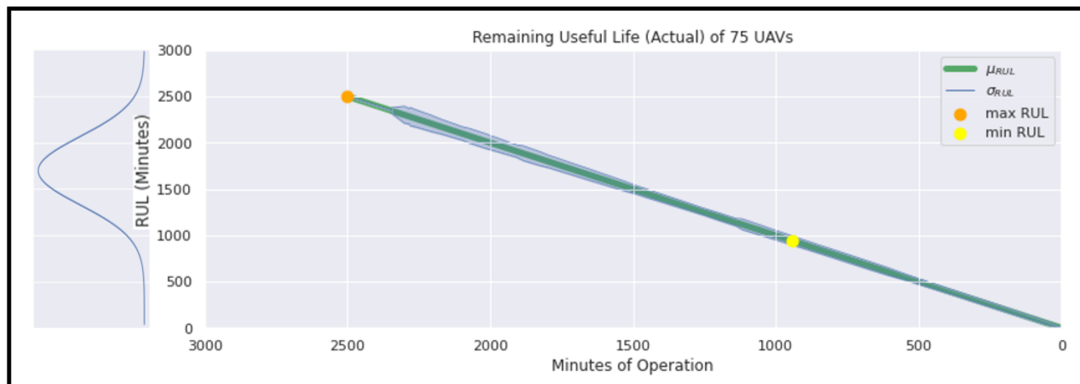

Figure 4.13: RUL mean, standard deviation, and distribution for 75 UAVs.

UAV ID	Flight Count	Unique Trajectories	Failure Code	Reason
712	77	27	2	low voltage
752	112	28	1	low SOC
772	82	27	1	low SOC
852	96	26	1	low SOC
892	106	28	1	low SOC
1032	107	27	1	low SOC
1052	123	28	3	position error
1072	58	26	1	low SOC
1092	77	28	1	low SOC
1112	65	24	2	low voltage
1132	104	28	1	low SOC
1152	92	27	1	low SOC
1192	94	28	2	low voltage
1212	72	25	2	low voltage
1232	73	27	1	low SOC
1252	68	26	1	low SOC
1272	136	28	1	low SOC
1372	117	28	3	position error
1412	78	28	1	low SOC
1432	92	28	1	low SOC
1452	91	25	1	low SOC
1472	126	28	2	low voltage
1492	100	27	5	position error variance
1532	112	27	1	low SOC
⋮	⋮	⋮	⋮	⋮
2092	96	27	1	low SOC
2112	121	28	1	low SOC
2172	113	28	3	position error
2232	67	26	3	position error
2772	87	27	1	low SOC
2792	125	28	3	position error
2812	102	28	3	position error
2832	61	25	3	position error
2852	95	28	1	low SOC
2872	91	27	5	position error variance
2892	103	27	1	low SOC
2952	97	27	1	low SOC
2972	98	27	1	low SOC
2992	96	27	1	low SOC
3012	52	26	3	position error
3032	101	28	1	low SOC
3052	123	28	1	low SOC
3092	107	27	2	low voltage
3112	79	25	1	low SOC
3152	105	27	1	low SOC
3172	94	28	1	low SOC
3192	113	28	1	low SOC
3232	65	25	3	position error
3292	84	27	2	low voltage
3352	74	25	1	low SOC
3392	118	28	1	low SOC
3512	78	28	1	low SOC
3572	76	23	5	position error variance
3652	105	27	1	low SOC
3672	68	25	1	low SOC
3692	60	25	2	low voltage

Table 4.11: UAV Flight Data

4.3.1 Flight Data

Each flight generates 84 data features sampled at 1Hz, listed in Table 9.1 (see Appendix), with 5 additional data features that are unique to each flight (first row). The average flight time was 18 minutes and 48 seconds, with a minimum flight time of 4 seconds (a failed flight), and a maximum flight time of 24 minutes and 45 seconds (a combination of a low setspeed and long trajectory). The value ranges for these features are shown in Figure 9.1 (see Appendix), having been sorted from minimum value to maximum value for each feature. An example flight profile is shown in Figure 4.14, which depicts the trajectory and waypoints, along with the velocity, acceleration, and jerk profiles. The UAV is required to stop for approximately 10 seconds at each waypoint, which is shown in the velocity profile. Figure 4.15 shows the GPS position and true position in a 3-dimensional perspective, in which the takeoff, landing, and sensor noise along with the effects of wind gusts can be seen in GPS signal (yellow). The reference altitude is reached after approximately 15 meters.

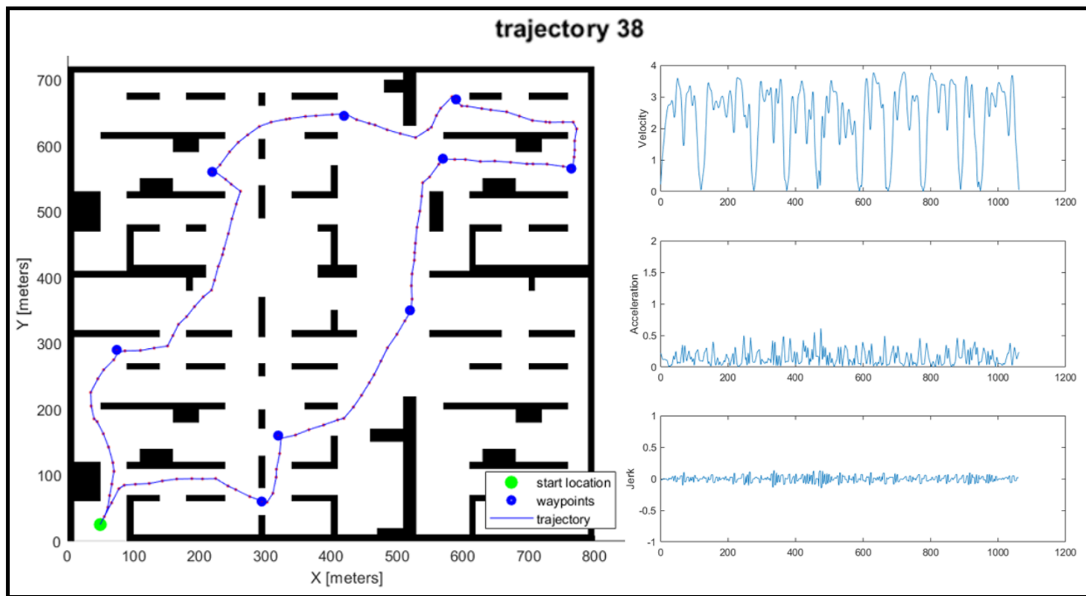

Figure 4.14: Example flight profile. Trajectory and waypoints (L), velocity, acceleration, and jerk (R).

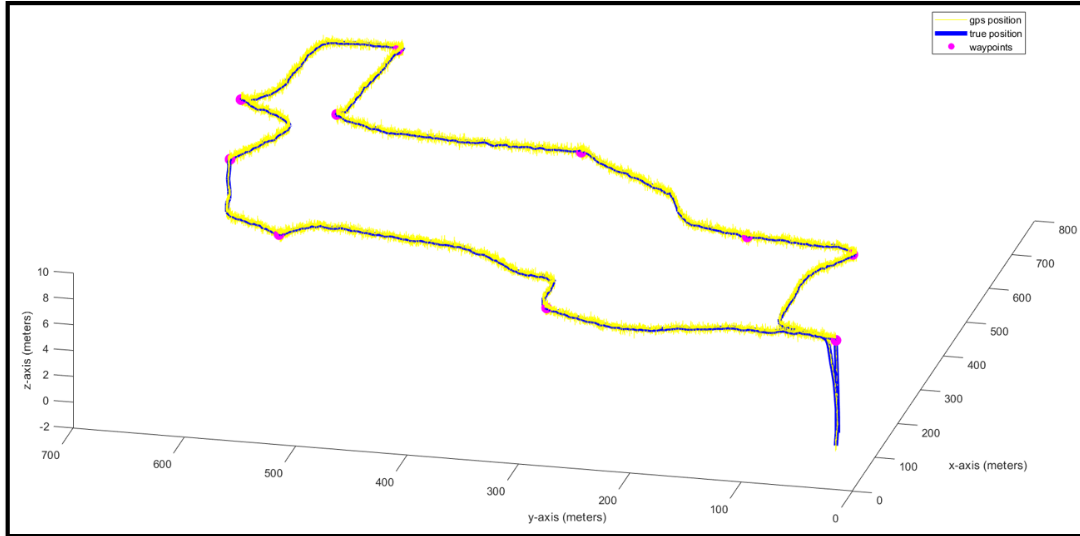

Figure 4.15: Example flight profile showing elevation, takeoff, and landing.

The primary data sources relevant to health management and prognostics used in model development (see Chapter 5) are depicted in Figures 4.16 - 4.19 for a nominal flight. Figure 4.16 depicts position (x, y, z), velocity $(\dot{x}, \dot{y}, \dot{z})$, acceleration $(\ddot{x}, \ddot{y}, \ddot{z})$, orientation, and angular velocity. Figure 4.17 shows power-related data from the battery and motors - voltage (v), SOC (z), internal resistance (r), and total current draw (i) from the battery, and current consumption (i), rpm (rpm), torque τ, and reference voltage (v_{ref}) for the motors. Note, the values in the motor plots are averaged over the 8 motors. Figure 4.18 shows position and control errors, and Figure 4.19 depicts the x, y, and z wind components, comprised of constant wind with stochastic gusts.

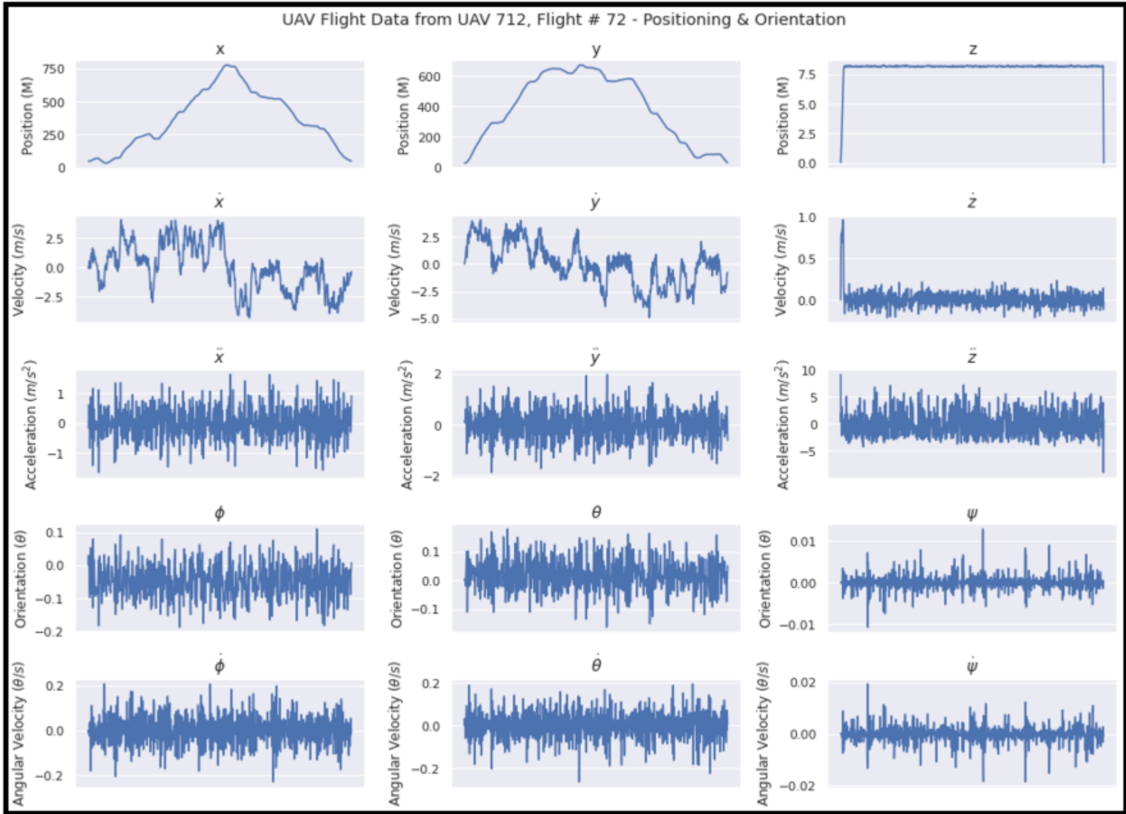

Figure 4.16: Flight data from a nominal flight showing position, velocity, acceleration, orientation, and angular velocity. This data comprises the system state, a key input for data-driven prognostics.

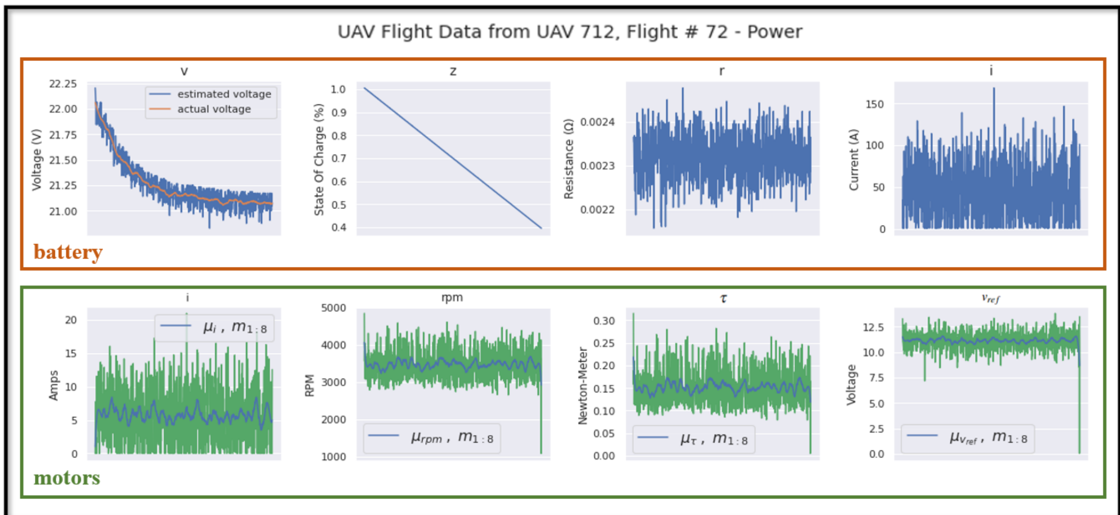

Figure 4.17: Flight data from a nominal flight showing power statics from the battery and motors. This data is the second key set of features used for data-driven prognostics.

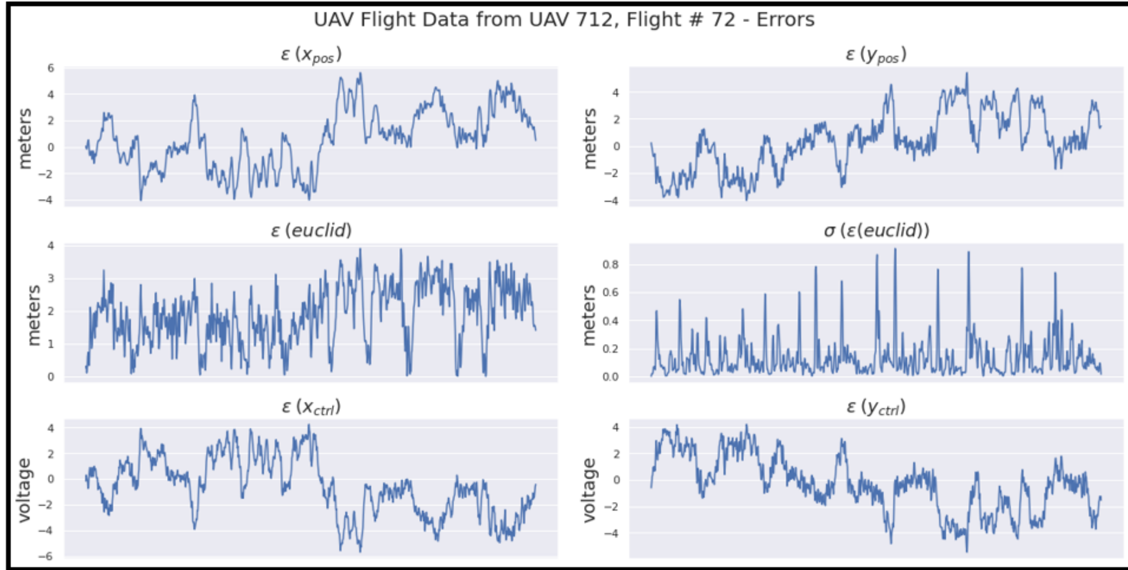

Figure 4.18: Flight data from a nominal flight showing position and control errors. The top row is the real-valued error in the X and Y axis. The middle row is the euclidean position error and the standard deviation in the euclidean position error. The bottom row is the control error in the X and Y axis.

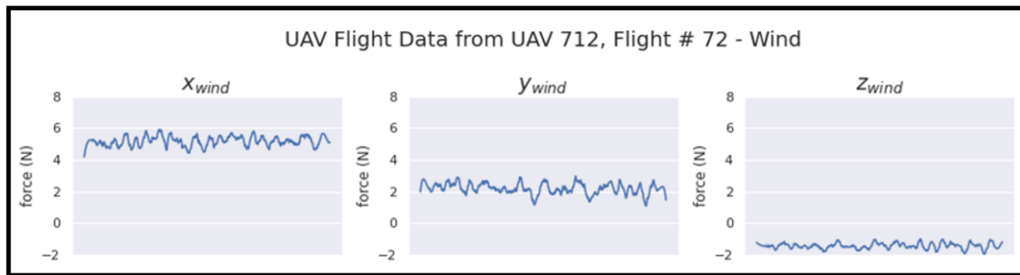

Figure 4.19: Flight data from a nominal flight showing wind (constant + gust), which is an important input as external disturbances affect system operation.

This simulation framework effectively generates, organizes, and stores high-fidelity flight data. The system, components, operating conditions, and flight profiles are all appropriately linked such that telemetry data can be traced back to the exact system configuration, inputs, and external disturbances that it was generated from. This is *data provenance*, and with it, we now have the necessary confidence to use the data for learning system-level prognostics models, discussed next.

CHAPTER 5

Data-Driven System-Level Prognostics

The goal of system-level prognostics (definition 1) is to predict the RUL of a system considering that multiple components in the system may be degrading simultaneously, and the interactions among these components affect overall system performance in complex ways as system use and time progress. *This chapter builds on previous work described in Chapter 3 to develop a data-driven system-level RUL model for prognostics.* Moreover, this chapter presents one of the first data-driven system-level RUL approaches, where the RUL is derived as a*continuous-time* function, as opposed to a function of discrete cycles in previous approaches. System degradation is attributed to a combination of individual degrading components in the system. Therefore, system EOL is a result of *physical failures* attributed to the effects of multiple component degradation, and the *operational failures* that result from the loss of system performance with respect to a set of predefined performance constraints (e.g., maintain system trajectory deviations to $< 3m$). We formally define the system-level RUL problem in equations 5.1 - 5.7 below, and then specify how the required variable values generated from a simulation model using the equations are organized into a database with a specific tabular structure to impose logging discipline.

The system RUL_n at time instant $n \in N$ is computed as a distribution that takes into account the different sources of uncertainty associated with the prediction process. System RUL takes as input the (hidden) state x_n, measurements y_n, and degrading parameters ϑ_n. More formally, following Khorasgani et al. (2016), system-level RUL_n is defined as:

$$RUL_n = (EOL_n - n)\Delta t, \tag{5.1}$$

where n represents the time step, Δt represents the sampling time, and EOL_n is the end of life (EOL) prediction of the system at time step n. System EOL is then defined as:

$$EOL_n = inf\{z \geq n \,|\, T(\wp(y_z, \vartheta_z)) = \top\}, \tag{5.2}$$

which is the earliest point in time when a system performance parameter threshold is violated. The system performance threshold function $T(\wp(y_z, \vartheta_z))$ is defined by:

$$T(\wp(y_z, \vartheta_z)) = \neg(\bigwedge_i \wp_i(y_z, \vartheta_z)), \tag{5.3}$$

where $\wp_i : y_z, \vartheta_z \rightarrow \{\top, \perp\}$ maps a specific performance parameter or a component parameter of the system to a Boolean domain $\{\top, \perp\}$ and denotes the set of performance constraint functions. Each constraint function $\wp_i(y_n, \vartheta_z)$ is described using a linear temporal logic (LTL) formula. Equation 5.3 returns true if at least one of the performance constraint functions returns false at time z.

The set of constraints is defined to account for both component and system-level performance metrics, and they are computed based on states x_n obtained from the system's degradation model. A system degradation model describes the behavior of a system as it degrades over time. Formally, the degradation model is defined as follows:

Definition 3 (System degradation model) *The system degradation model is defined as*

$$\mathbf{x}_{n+1} = f(\mathbf{x}_n, \mathbf{u}_n, \vartheta_n, \mathbf{w}_x), \tag{5.4}$$

$$\vartheta_{n+1} = g(\vartheta_n, \alpha_n, \mathbf{x}_n, \mathbf{w}_d), \tag{5.5}$$

$$\mathbf{y}_{n+1} = h(\mathbf{x}_n, \mathbf{u}_n, \vartheta_n, \mathbf{w}_y), \tag{5.6}$$

$$\alpha_{n+1} = g'(\mathbf{x}_n, \mathbf{w}_a); \tag{5.7}$$

where $\mathbf{x}_n \in \Re^m$ denotes the state vector describing the dynamics of the system; $\mathbf{u}_n \in \Re^o$ denotes the input; $\mathbf{y}_n \in \Re^p$ represents the measured variables in the system at time step n; and $\vartheta_n \in \Re^q$ is the set of degrading parameters. $\alpha_n \in \Re^q$ define the degradation rates for the set of component degradation parameters in the system. Since the degradation of components may be nonlinear, equation 5.7 (g') captures the dynamics of how the component degradation rates themselves may vary over time. Equations f, g, and h are the state update, parameter update, and system output functions, respectively. Together, they characterize the system dynamics. $\mathbf{w}_x$, $\mathbf{w}_d$, $\mathbf{w}_y$, and $\mathbf{w}_a$ capture the uncertainties and disturbances associated with the syfstem model, the system degradation model, the measurement noise, and component degradation models, respectively.

The noise and disturbance parameters, $\mathbf{w}_x$, $\mathbf{w}_d$, $\mathbf{w}_y$, and $\mathbf{w}_a$ are modeled as $N(0,1)$ normal distributions, and therefore, all of the system variables $\mathbf{x}_n$ and $\mathbf{y}_n$ and parameters ϑ_n are stochastic variables, and equations 5.4 - 5.7 represent a stochastic simulation model of the degrading system.

5.1 Deep Learning Model Development

We adopt the Bi-LSTM architecture, shown in Figure 5.1, as the basis for the RUL estimation model. The primary steps in a model development pipeline are (1) preprocessing and exploratory analysis, (2) feature selection and engineering, (3) hyperparameter optimization, and (4) training and validation. Preprocessing typically consists of "data cleaning", i.e., ensuring proper time alignment, checking for missing values and

choosing how to fill or remove them, and fixing data types and rounding issues. This is followed by feature selection and engineering. Following that, hyperparameter optimization is discussed, followed by training and evaluation.

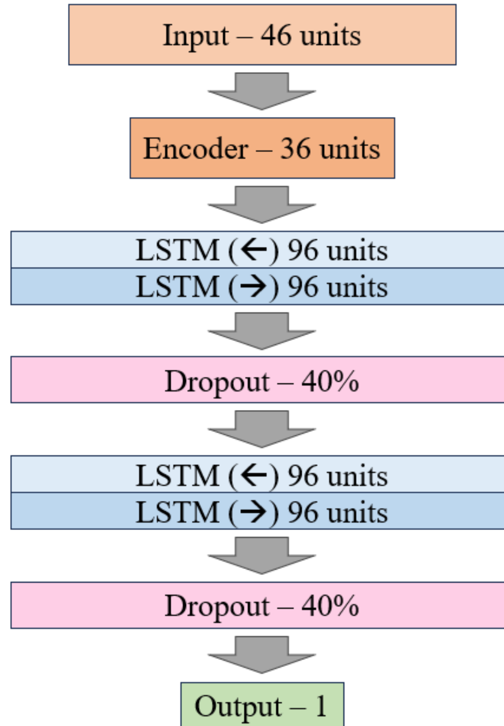

Figure 5.1: Bi-Directional LSTM Architecture for RUL Estimation.

5.1.1 Feature Selection & Engineering for Prognostics

Feature selection and engineering are two steps in the model development pipeline that are used to either reduce the size of the input space by creating more aggregated new features. The goal is to build simpler models that are easier to understand, improve visualization, reduce storage and processing requirements, and minimize prediction errors (Guyon and Elisseeff (2003)). Feature selection relies on the use of similarity-based or information-based methods (Li et al. (2016a)) to assign importance scores to each of the features, whereby a subset of those features is selected based on a predefined cutoff. Feature engineering, also sometimes known as feature extraction, is the process of creating entirely new features using the raw data and various transformations (Zheng and Casari (2018)). We discuss the desired characteristics of features that are amenable for prognostics applications. Two metrics are combined to perform feature selection: (1) *prognosability* and (2) *coefficient of determination*. Prognosability characterizes the variance of a signal at failure time, and a wide spread in critical failure values can make it difficult to accurately extrapolate a prognostic parameter to failure (Coble (2010)). This metric is calculated as the variance of the final values near and at failure time of

the population of units divided by the average range of values, as given in Equation 5.8:

$$P_x = exp\left(-\frac{\sigma x_{if}}{\mu(|x_{if} - x_{is}|)}\right), i = 1 : p \tag{5.8}$$

where x_i is the xth feature of the ith unit in a population p, f represents the final values, and s represents the starting values. In the original work and subsequent papers, the amount of values for f and s is not made clear, therefore, the first and last 10% of values were chosen. The second metric chosen is known as the coefficient of determination, better known as R^2, which has shown to be an excellent predictor of a feature's importance on a given target variable (Chicco et al. (2021)). R^2 is a statistical measure used in regression analysis to provide a concise indication of how well the independent variables in a regression model explain the variability in the dependent variable. In other words, it quantifies the proportion of the variance in the dependent variable that is predictable from the independent variables. This calculation is performed over the entire population p of units, and, therefore, the average R^2 values of each feature over p are taken. The formula is given as follows:

$$R_j^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2},$$

where y_i is the actual target value for the i-th sample, $\hat{y}_i$ is the predicted value, and $\bar{y}$ is the mean of the actual target values. The final importance of each feature is taken as $.5 * P_j + .5 * R_j^2$, with the results shown in Figure 5.2. These methods for feature selection were chosen as the first method is specific to prognostics applications, and the second method is specific for regression tasks. Both of these are characteristics of the underlying problem of RUL estimation.

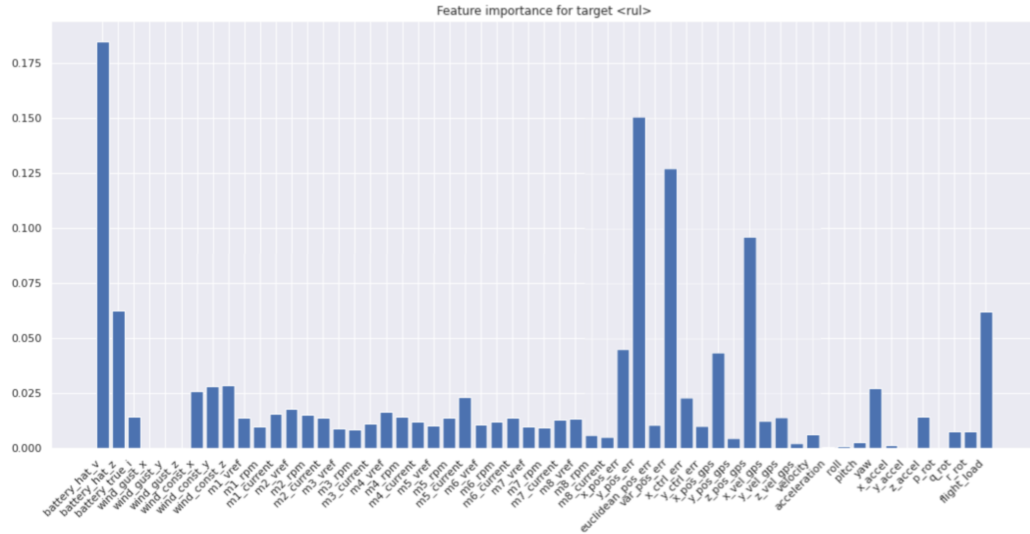

Figure 5.2: Feature importances of the data evaluated for suitability to be used as input to the RUL model.

Dimensionality reduction is a common form of feature engineering when developing deep learning models that project the input space onto fewer dimensions. Methods such as Principle Component Analysis (PCA), Singular Value Decomposition (SVD), or Distributed Stochastic Neighbor Embedding (t-SNE) are discussed in (Verdonck et al. (2021), Zheng and Casari (2018)). See (Zebari et al. (2020)) for a complete review of dimensionality reduction methods.Wang et al. (2014) proposed a dimensionality reduction technique based on Autoencoders, which is the method we employ in this work. We can see from the above figures that some features, such as wind gusts, do not have any bearing on the RUL of the system. The wind gust is an interesting input feature that results in near-0 feature importance scores when the target variable is RUL, however, conceptually we know that environmental conditions affect UAV operation directly or indirectly (e.g., wind gusts can cause the length of a flight trajectory to increase). Wind gusts are stochastic and frequently change during a given flight randomly. Constant wind, on the other hand, does affect RUL, because it typically remains the same magnitude for the entire duration of a given flight. Stronger constant winds result in increased power demand, which increases battery aging and wear-and-tear on the motors. We will see in Section 5.3 that wind is used as input for the purposes of calculating flight time and power consumption. To reduce dimensionality while retaining as much information as possible an Autoencoder is employed. The autoencoder architecture is shown below in Figure 5.3. It was trained with the Adam optimizer and a learning rate of .0025. The number of layers, units, and learning rate were tuned as described below, however, no regularizing parameters were required. The resultant reconstruction map is shown in Figure 5.4, where each feature and it's reconstructed value is depicted. The decoder portion of the autoencoder is discarded, and only the encoder is used in the final RUL model (see Figure 5.1 above).

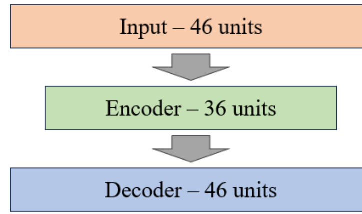

Figure 5.3: Autoencoder architecture.

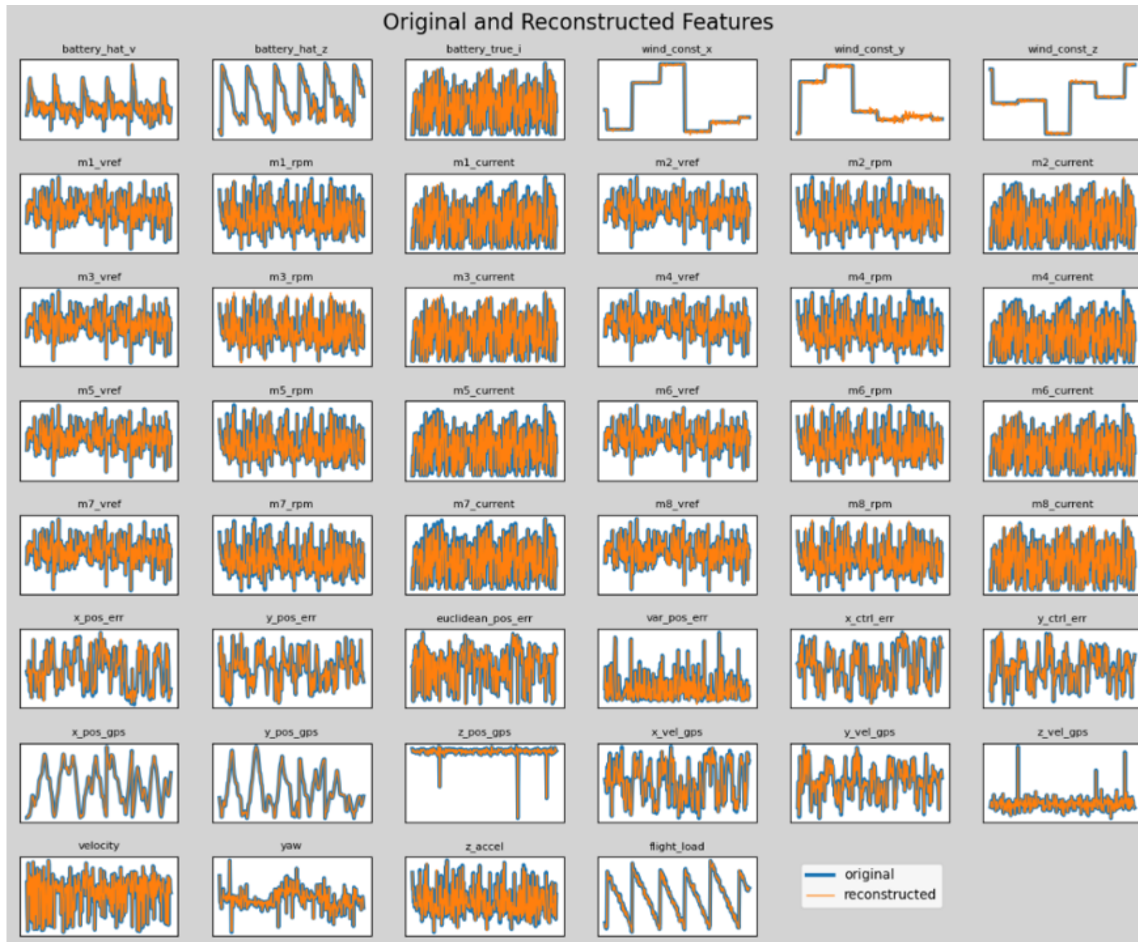

Figure 5.4: Original and reconstructed features of the RUL model input data.

5.1.2 Hyperparameter Optimization

Proper hyperparameter configuration is essential to developing high performing models. We do not want to create big models when smaller models would suffice, or manually select hyperparameter values that lead to sub-optimal configurations. Models tuned with hyperparameter optimization methods have shown to be superior than guess-and-check methods (Bergstra et al. (2011); Feurer and Hutter (2019)). In our initial

experiments, we found that configurations with non-zero regularizing parameters did not even make the list of top 5 best configurations. Figure 5.5 illustrates why this is the case. Regularization is a method (or set of methods) to improve a model's ability to generalize (Kukačka et al. (2017)) and imparts a time-quality tradeoff during training (Goodfellow et al. (2016)). This means that the resultant model will have a smaller loss than its non-regularized counterpart, but will take longer to achieve a lower loss.

In Figure 5.5, four loss functions are depicted along with the search horizon ($\hbar$) for the optimization procedure. It is clear that at this stage in training, the network with non-zero regularization parameters performs the worst. Informed searches, such as Bayesian Optimization Snoek et al. (2012a) or Hyperband (Li et al. (2016b)) either reduce the probability of improvement or expected improvement around this candidate solution or discard the configuration all together.

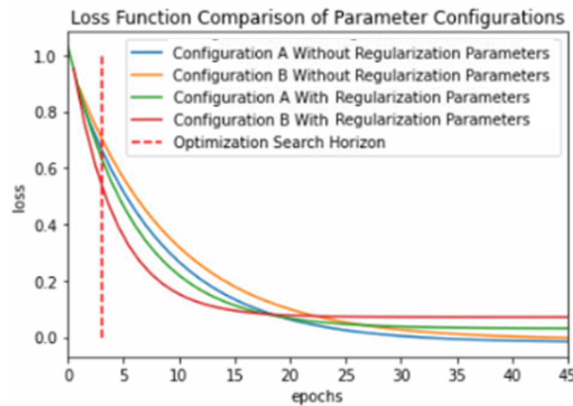

Figure 5.5: Comparing the effect of regularization on the loss function.

However, this turns out to be the best solution, but only after the 25th epoch has been employed for training. Therefore, a two-step hyperparameter search is implemented whereby the layers, units, input sequence length, batch size, and learning rate are searched first (Darrah et al. (2022b)). Initially, we chose the top-scoring configurations but later realized the results proved to be misleading. Instead of concluding after the search process is complete (by means of evaluating N_t number of trials) that the top-scoring networks are the best, the distribution of scores for each configuration should be calculated, and the mean of the distribution should be used to rank configurations. In this manner, the top N_c candidates are selected from this step, and the search is then repeated with dropout, l1, and l2 regularization. A Bayesian approach to hyperparameter optimization is utilized.

Bayesian optimization operates by creating a posterior distribution of functions using a Gaussian process that best represents the target function being optimized (Snoek et al. (2012b)), in the case of regression

tasks, this is typically mean squared error. As more data points are collected, the posterior distribution improves, enhancing the algorithm's certainty about which regions in the parameter space are valuable to explore and which are not. Through iterative steps, the algorithm strikes a balance between exploration and exploitation, leveraging its knowledge of the target function, in the case of regression tasks, this is typically the mean squared error. At each iteration, a Gaussian Process is fitted to the known samples from previous explorations and the posterior distribution, and then an acquisition function such as upper confidence bound (UCB) or expected improvement (EI) guides the selection of the next point to explore. The core idea behind Bayesian optimization can be mathematically represented by the general optimization problem, and Bayes formula (Wu et al. (2019)), as follows:

$$h^* = arg\max_{h \in H} f(x,h), \tag{5.9}$$

$$P(f|f(x)) \propto P(f(x)|f)P(f), \tag{5.10}$$

where H denotes the search space of h, f is an unknown function, and $f(x)$ is the functions output. In the case of machine learning, f is not a known closed-form function and is thus computed as a *black-box*. In this case, Bayesian optimization maintains a probabilistic model of f,

$$P(f|f(x)) = GP(f;\mu_f|f(x), \sigma_f|f(x)), \tag{5.11}$$

and employs an acquisition function to determine the next points to evaluate. For a complete treatment of acquisition functions the reader is directed to (Wu et al. (2019)). In this work, UCB is used. UCB balances exploration and exploitation effectively by considering both the mean and variance of the surrogate model. It encourages exploration in regions of high uncertainty, which makes it well-suited for situations with limited data or noisy objective functions. The UCB function is defined as

$$A(h, f, \beta) = \mu_f(h) + \beta\sigma_f(h), \tag{5.12}$$

where A is the acquisition function, $\mu_f(h)$ and $\sigma_f(h)$ are the mean and standard deviation of f given the set of hyperparameters h, respectively, and β is the explore-vs-exploit parameter. Algorithm 4 is the general Bayesian optimization process for hyperparameter tuning. In this algorithm, H represents the hyperparameter search space, M is the machine learning model, f is the objective function, and T is the maximum number of iterations. The algorithm starts by initializing an empty dataset $\mathscr{D}$ and a surrogate model. It then iterates

over a fixed number of iterations T where at each iteration, the surrogate model is updated with the current dataset $\mathscr{D}$, and the best hyperparameters $\mathbf{h}_t$ are found by maximizing the acquisition function. The objective function is evaluated with the chosen hyperparameters, and the dataset is updated with the new observation. After all iterations, the algorithm returns the best hyperparameters found in the dataset $\mathbf{h}_{\text{best}}$.

Table 5.1 lists the search space for the Bi-directional LSTM remaining useful life estimation model. The top 5 candidates from stage 1 were selected out of 304 trials, whereby a total number of 116 different configurations were tested. These candidate configurations then become another hyperparameter for the search in stage 2, where dropout, l1, and l2 regularization are searched. The search parameters are given in Table 5.2. Alpha (α) governs the expected noise in the observed performance of the underlying surrogate model and beta (β) governs how far to draw samples from the current best solution. The search horizon ($\hbar$) is the number of epochs to evaluate each configuration and N_c is the number of top-scoring configurations to keep for step 2, which is searching over the regularization parameters. The final configuration of the RUL model is shown in Table 5.3.

Algorithm 4: Bayesian Optimization for Hyperparameter Tuning

Require: Hyperparameter search space H, machine learning model M, objective function f (performance metric), maximum number of iterations T.

1: Initialize dataset $\mathscr{D} \leftarrow \{\}$ and surrogate model.
2: **for** $t = 1$ **to** T **do**
3: Fit the surrogate model to the dataset $\mathscr{D}$.
4: Find hyperparameters $\mathbf{h}_t$ by optimizing the acquisition function: $\mathbf{h}_t = \text{argmax}_{\mathbf{h} \in \mathscr{H}} A(\mathbf{h}; \text{surrogate model})$.
5: Evaluate the objective function: $y_t = f(\mathbf{h}_t)$.
6: Update the dataset: $\mathscr{D} \leftarrow \mathscr{D} \cup \{(\mathbf{h}_t, y_t)\}$.
7: **end for**
8: **return** Best hyperparameters found: $\mathbf{h}_{\text{best}} = \text{argmax}_{(\mathbf{h},y) \in \mathscr{D}} y$.

Stage	Parameter	Symbol	Range	Step
1	number of layers	l	1 - 4	1
	units per layer	u	8 - 96	8
	learning rate	lr	$1e^{-4}$ - $5e^{-3}$	$5e^{-4}$
	batch size	b	32 - 256	32
	sequence length	s	60 - 600	60
2	dropout rate	d	.1 - .5	.1
	l_1 regularization	l_1	$1e^{-7}$ - $1e^{-4}$	x10
	l_2 regularization	l_2	$1e^{-7}$ - $1e^{-4}$	x10

Table 5.1: Model Search Parameters

Parameter	Symbol	Value
number of trials	N_t	512
alpha	α	.0001
beta	β	4.2
search horizon	$\hbar$	3
number of candidates	N_c	5

Table 5.2: Bayesian Optimization Parameters

Parameter	Symbol	value
number of layers	l	2
units per layer	u	96, 96
learning rate	lr	$5e^{-4}$
batch size	b	192
sequence length	s	400
dropout rate	d	.4
l_1 regularization	l_1	$1e^{-5}$
l_2 regularization	l_2	0

Table 5.3: RUL model parameters found through Bayesian optimization.

5.1.3 Training Procedure

The dataset was split into a *training* set, *validation* set, and *testing* set by randomly assigning each UAV to a set. The training set consists of 56 UAVs and a total of 5,938,740 records; the validation set consists of 10 UAVs and 1,020,855 records; and the testing set consists of 9 UAVs and 856,835 records. This process is depicted in Figure 5.6. Because system degradation occurs over large timeframes, the training data was decimated by a factor of 20, whereby every 20th sample was taken, which is 3 samples per minute. The average flight lasted a little over 21 minutes, and so each flight resulted in an average of 63 samples per flight. In this manner, each flight is actually comprised of 20 unique sets of samples, whereby the decimation originally starts with the first and subsequently every 20th thereafter; then the second reading and every 20th thereafter; ending at the 19th reading, and every 20th thereafter. This logic came from previous experiments with the N-CMAPSS dataset (Darrah et al. (2022b)) in which downsampling the data had the effect of both reducing noise and extending the amount of time in history the model is able to look for a given sequence length. For example, without downsampling the 1Hz telemetry data, a sequence length of 400 would provide only 8 minutes of historical data, and only cover data from a singe flight. When downsampling by a factor of 20, the same sequence length now captures 133 minutes of historical data, and captures data from 6 to 8 flights. The training process is presented below in Algorithm 5. Since there are 56 units in the training set, each epoch comprises of iterating over each unit once. Since the training set is decimated by a factor of 20, the model is trained for 20 epochs. In this manner, the model is trained with a sufficient amount of data such that it never trains on the same data twice.

Line 1 of the algorithm is the outermost loop, iterating over the training set by decimation. The training and validation arrays are initialized in lines 2-5. Lines 6-11 iterate over the training set to reshape the data in

the format expected by the LSTM network, with a shape of (num-samples, num-timesteps, num-features), in a function called *temporalize*. Lines 13-19 do the same for the validation set. Line 20 is the fit procedure, which takes in the training set, validation set, batch size, and learning rate decay function. The learning rate decay function is triggered after each UAV in the training set completes a single iteration. For a training set with 56 UAVs, this means the function is called 56 times in a single training epoch. Learning rate decay is discussed below. After the complete training process is complete (lines 1 - 22), the trained model is evaluated with a separate evaluation set. The data is reshaped in the same manner as before and the model is evaluated using the root mean squared error (RMSE) function. Finally, the algorithm returns the trained model, **M'**, and its evaluation score, **e**.

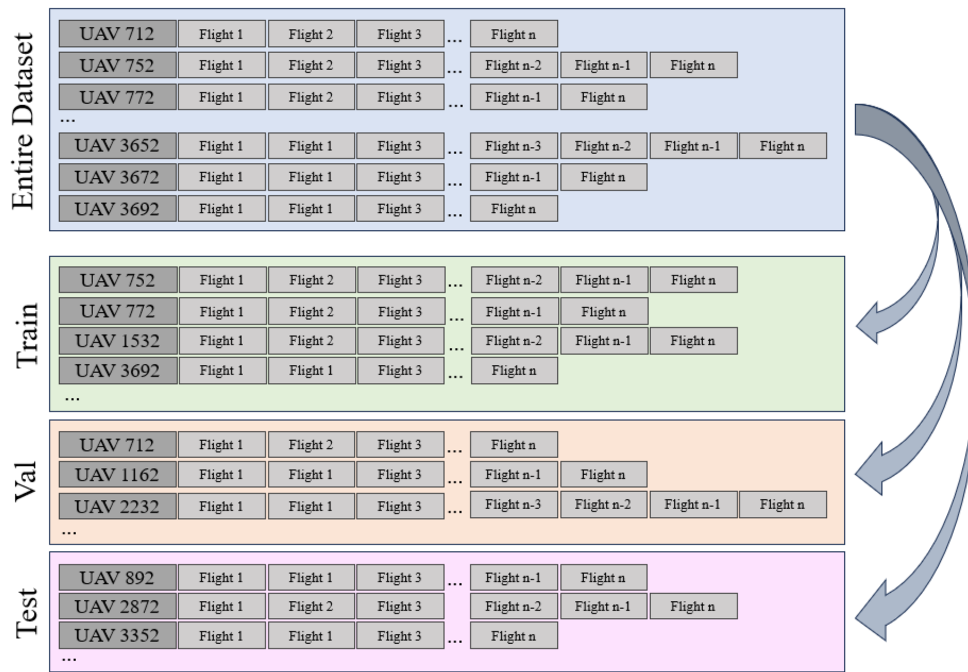

Figure 5.6: Dataset split based on random UAV assignment.

Algorithm 5: Training Process for System-Level RUL Estimation Model

Input: **M**, a compiled Encoder-Bi-LSTM model; a training dataset, **T**; a validation dataset, **V**; an evaluation dataset, **E**; a list of input features, **F**; an target output feature, *rul*; a sequence length, **l**; a batch size, **bs**; a learning rate decay function, **R**, and a decimation factor, **d**

Output: A trained model, **M'** and its evaluation score, **e**

```
1  for i = 0 ... d do
2  |   X ← ∅
3  |   Y ← ∅
4  |   X_val ← ∅
5  |   Y_val ← ∅
6  |   foreach UAV in T do
7  |   |   x ← T[UAV, F, i :: d]
8  |   |   y ← T[UAV, rul, i :: d]
9  |   |   x, y ← temporalize(x, y, l)
10 |   |   X ← X ∪ x
11 |   |   Y ← Y ∪ y
12 |   end
13 |   foreach UAV in V do
14 |   |   x_val ← V[UAV, F, i :: d]
15 |   |   y_val ← V[UAV, rul, i :: d]
16 |   |   x_val, y_val ← temporalize(x_val, y_val, l)
17 |   |   X_val ← X_val ∪ x_val
18 |   |   Y_val ← Y_val ∪ y_val
19 |   end
20 |   M.fit(X, Y, bs, X_val, Y_val, R)
21 end
22 M' ← M
23 X_test ← ∅
24 Y_test ← ∅
25 foreach UAV in E do
26 |   x_test ← E[UAV, F, i :: d]
27 |   y_test ← E[UAV, rul, i :: d]
28 |   x_test, y_test ← temporalize(x_test, y_test, l)
29 |   X_test ← X_test ∪ x_test
30 |   Y_test ← Y_test ∪ y_test
31 end
32 e ← M'.evaluate(X_test, Y_test)
33 return M', e
```

Learning rate decay is implemented during the training process. By gradually reducing the learning rate over time, the model is able to take larger steps initially in the parameter space, allowing it to progress quickly towards an optimal solution. As training proceeds, the learning rate is decreased, enabling the model to fine-tune its parameters with smaller, more precise steps. This approach not only improves convergence speed but also enhances the model's generalization to unseen data by preventing overshooting and oscillations. Moreover, learning rate decay contributes to training stability, preventing abrupt changes in the model's parameters during optimization (You et al. (2019); Ding (2021)). Exponential decay is the method implemented here,

which exponentially reduces the learning rate either by multiplying it by a decay factor at a given interval, in this case, after each iteration of a training unit. The learning rate decay function is shown in Figure 5.7. The learning rate stays at its initial value for an entire training epoch, after which, it is exponentially decayed.

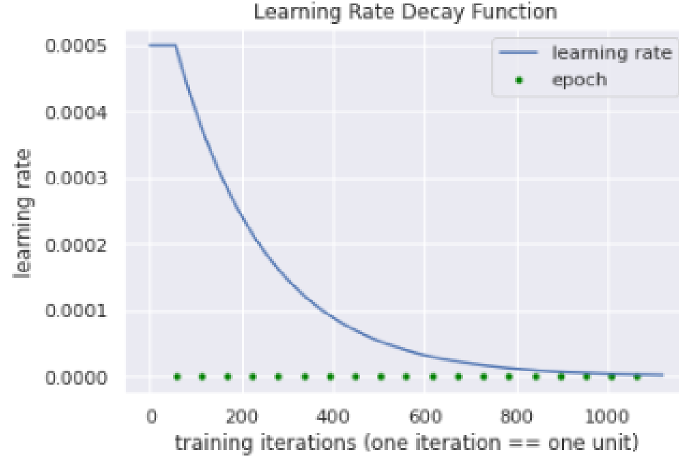

Figure 5.7: Learning rate decay function. The function is evaluated after each training unit, where 56 training units comprise of a single epoch.

NASA's scoring function is used (Saxena et al. (2008)) as the loss function during, shown in Equation 5.13. This function penalizes over estimations due to the fact that when used in real life, over estimations can lead to unsafe operating conditions resulting in a loss of property or life and must be avoided at all costs. Under estimations will reduce operation and incur maintenance costs, but the impacts of these errors are much less than that of over predictions. It is better to be conservative with RUL estimates for this reason.

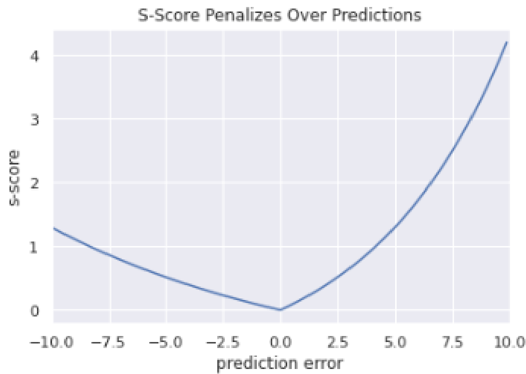

Figure 5.8: Training loss function.

$$
\begin{aligned}
\Delta_j &= \hat{y}_j - y_j, \\
\alpha &= \begin{cases} \gamma_1 - 1 & \text{if } \Delta_j \leq 0,\ \gamma_1 = \frac{1}{12} \\ \gamma_2 - 1 & \text{if } \Delta_j > 0,\ \gamma_2 = \frac{1}{6} \end{cases} \\
s &= \sum_{j=1}^{m_*} e^{\alpha}, \\
MSE &= \frac{1}{m_*} \sum_{j=1}^{m_*} \Delta_j^2 \\
loss &= (s + MSE)/2.0
\end{aligned}
\tag{5.13}
$$

Training loss function for prognostics.

5.2 Training Results

The results of the trained model are shown in Figure 5.9, which aggregates the RUL predictions of 18 UAVs from the test and validation sets. Individual prediction plots of each of these UAVs are shown in Figure 5.10 using the encoder-Bi-LSTM network with an input sequence length of 400 timesteps.

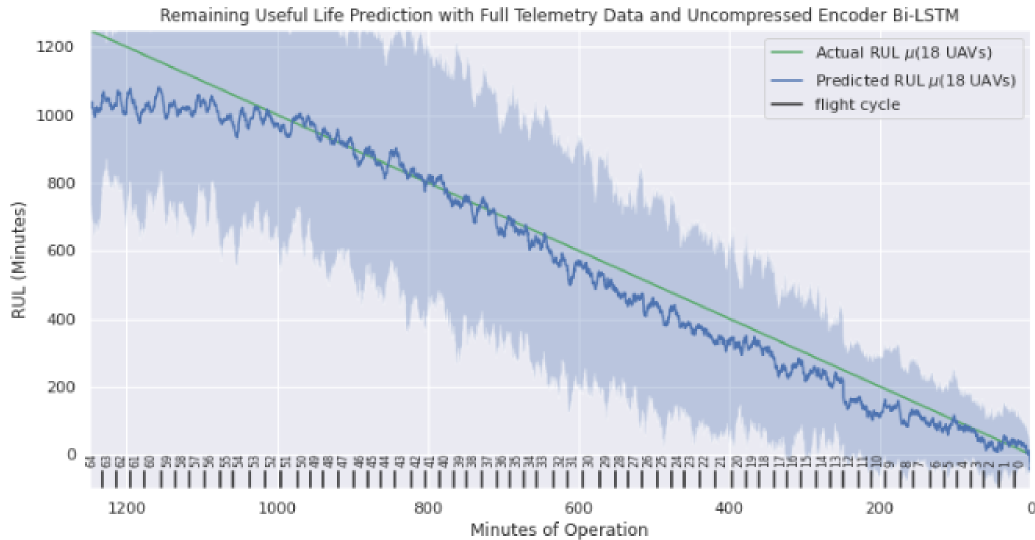

Figure 5.9: RUL estimation results from the test and validation datasets, showing the mean predicted value (blue), the actual value (green), and the 95% confidence interval (shaded grey).

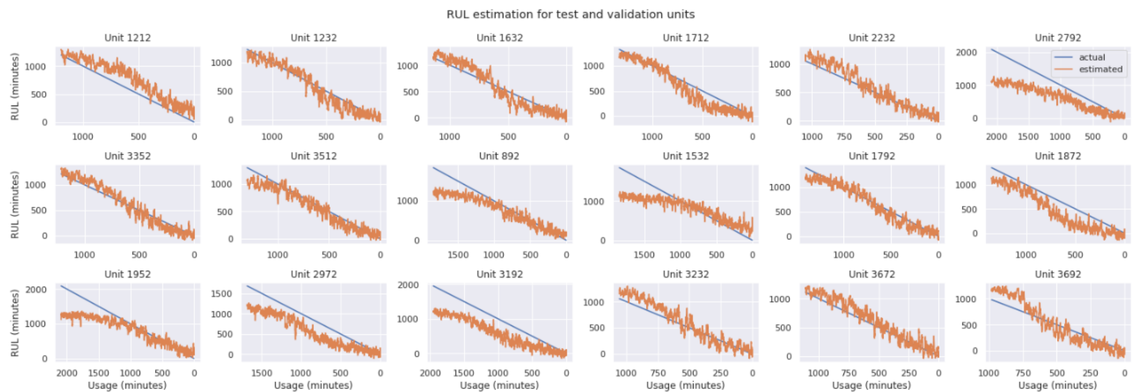

Figure 5.10: RUL estimation plots for the UAVs in the test and validation datasets.

To demonstrate that encoder-LSTM architectures are superior,, the encoder-Bi-LSTM is compared against two Bi-LSTM networks, all of the same architecture, the difference among the three being (1) the encoder in the first network, and (2) the timesteps used for training and prediction, shown in Figure 5.11. We can see from these results that the encoder-based Bi-LSTM model with an input sequence length of 400 is superior to a regular Bi-LSTM for input sequences of 400 and 600. With an accurate RUL estimation model, further tasks can be conducted which rely on RUL information, discussed in the following section.

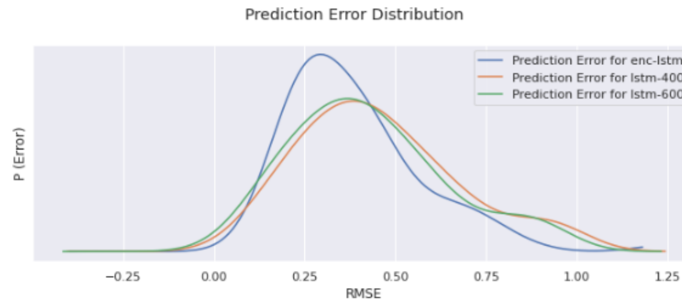

Figure 5.11: RUL estimation error comparison for 3 Bi-LSTM Variants: The encoder-BiLSTM architecture presented in Section 4.1 (blue); and the same architecture with two different input sequence lengths and without the encoder (orange and green). The encoder-BiLSTM has both a lower RMSE and a smaller variance.

5.3 Auxiliary Neural Networks

With the trained RUL model in place, it's output can be used to augment the input feature space of other models to enhance their performance. Specifically, this is for flight time and power consumption estimation of an arbitrary flight segment. This information is relevant for the replanning agent, which uses these estimates during the replanning process to derive the most feasible plan without overshooting how much remaining time is available. These two estimates come from a simple 1-layer DNN with an input feature space shown in Figure 5.12. The input features consist of the segment distance, the current velocity, the RUL estimate, current wind conditions, battery internal resistance, and motor current consumption. The outputs are segment flight time and segment power consumption. 30 samples were randomly selected for visualization in Figure 5.13.

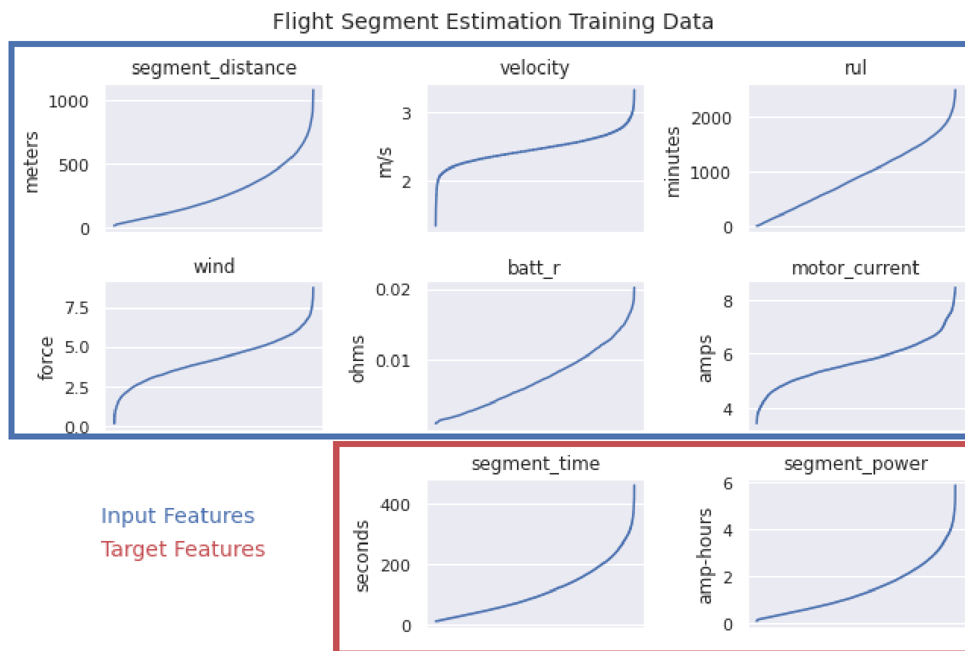

Figure 5.12: Input data plots for the flight time and power estimation model, sorted by value to show range of inputs.

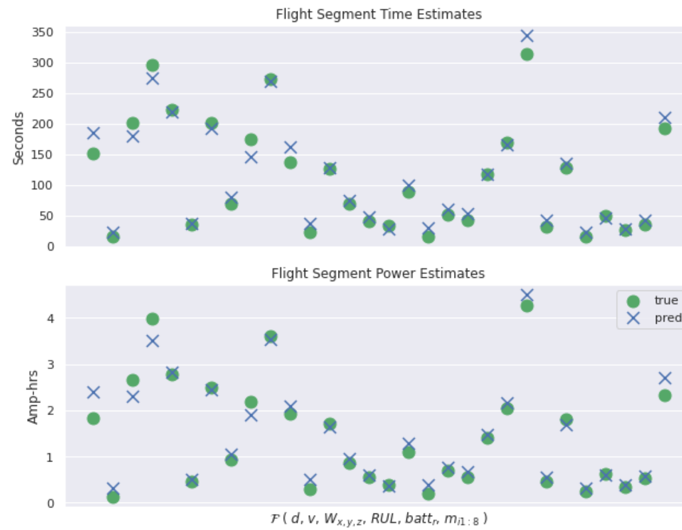

Figure 5.13: Scatter plot of 30 random data samples of flight time estimations (top) and power consumption estimations (bottom).

To demonstrate the importance of having the RUL estimation as an input to the auxillary model, a second experiment was conducted to train the model without the RUL estimates. The results show that the model trained without RUL information has a higher mean error rate and a larger standard deviation. This is shown in Figure 5.14. Finally, the remaining flight time plot for the 18 units in the test and validation set is shown in Figure 5.15.

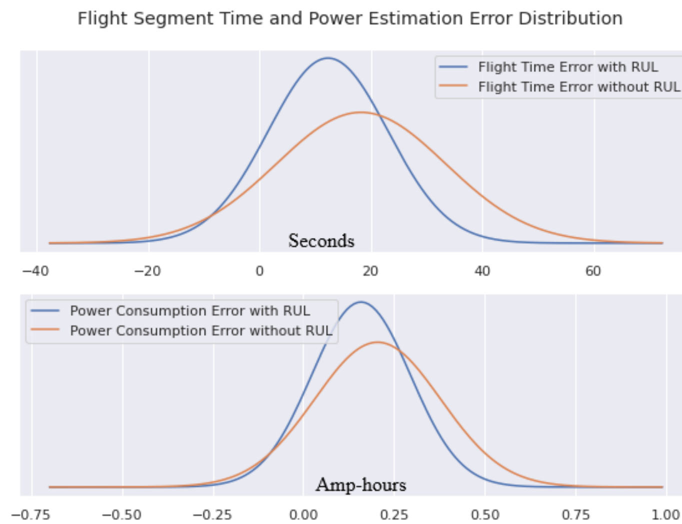

Figure 5.14: Estimation error distribution in units of seconds (top), comparing estimations with and without system-level RUL information; Estimation error distribution in units of amp-hours for power consumption (bottom, same comparison). The model without RUL information has both a higher error and a larger variance.

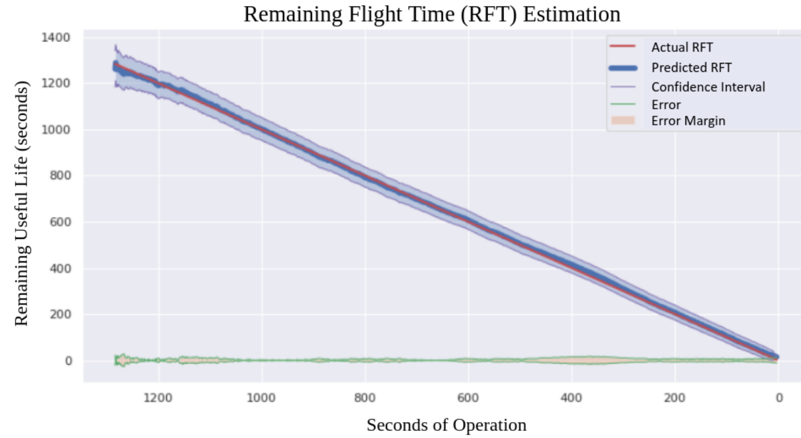

Figure 5.15: Remaining flight time estimation with RUL information.

5.4 Summary

A deep learning methodology for system-level prognostics of an octorotor UAV is developed in this chapter. This is demonstrated with a continuous-valued RUL target variable, which has not been done before. All previous system-level RUL studies found in the literature use discrete-valued RUL in units of cycles - not actual operating time. An encoder-BiLSTM network is used with hyperparameters tuned in a two-step optimization procedure that searched over regularizing parameters separate from network parameters. This approach was shown to be more effective than searching over all parameters at once, however more studies should be conducted to prove this in the general case. The network was trained with the NASA scoring function that has been modified to be used as the loss function during training, which increases the penalty for over-predictions. This results in a network specifically trained to avoid over-predictions, and the RUL for only 2 out of 18 units in the test and validation sets was over-predicted. Additionally, auxillary models were developed to perform resource consumption estimates on a shorter timeframe than the RUL model. These models were trained with and without the RUL estimations as input and it was shown that they have a lower error rate and variance when this information is available. This hypothesis is further supported in Chapter 7, where this information is used in the replanning process.

CHAPTER 6

Model Reduction for Long-Short Term Memory Networks

In recent years, model compression has gained significant attention as a necessary technique to alleviate the resource and deployment challenges posed by the deployment of deep neural networks. *The purpose of this chapter is to present an application-specific method for model reduction that is tailored for LSTM-based time series regression models. Our application is RUL estimation in the context of system-level prognostics.* The reduced models will then be tested on both a desktop computer and embedded hardware (See Sections 7.4 - 7.5). We hypothesize that there may be differences in execution time and solution quality due to CPU limitations such as clock speed, instruction set, or supported mathematical optimizations. Therefore, it will be necessary to evaluate the reduced model on the target hardware for the target application.

Most model reduction methods discussed in the literature have been developed for CNN compression applied to image classification tasks. However, the compression of LSTM networks used in time series regression presents unique challenges due to the intricate temporal dependencies and long-term context preservation requirements. In contrast to time series regression, LSTM models for language tasks, are more adaptable to varying input lengths and contextual representations, allowing them to capture linguistic nuances and syntactic structures with sparse numeric precision (Chen et al. (2016)). Furthermore, language tasks often focus on discrete categorical outputs, reducing the need for continuous value preservation. In time series regression, careful preservation during compression is required to ensure that temporal relations are not lost, and accurate predictions in time can be made. Additionally, the continuous nature of numeric values necessitates precision preservation to capture subtle variations in the data, especially as the data changes over time, as in the case of prognostics tasks, such as RUL estimation. As a result, effective model compression for time series regression must address these challenges by adapting existing methods to safeguard temporal patterns, long-range dependencies, and fine-grained numeric information.

6.1 Understanding the Behavior of an LSTM-based Prognostics Model

Predicting RUL of complex autonomous systems is a critical task that has far-reaching implications for maintenance and operational planning or replanning, as in the case of this work. Many systems found in space and aerospace applications are equipped with small, resource-constrained embedded computing platforms due to size, weight, and power consumption considerations. The desire to run deep neural networks on these systems is the key motivation behind model compression. In this context, the behavior of LSTM models in the domain of prognostics becomes a prime area of exploration that has received little attention in the literature. Prognostics deals with predicting the time to failure, and this prediction is inherently tied to the temporal aspect of the data, making LSTM networks a natural choice due to their ability to capture sequences and long-range dependencies. The essence of this problem lies in understanding how LSTMs behave when tasked with predicting the RUL as the underlying system transitions from a state of health to that of impending failure. All CPS components undergo gradual degradation in performance over time, and this can be observed as a monotonic decrease in the target performance variables until they reach a critical point known as "failure." This puts a specific focus on the accuracy of the predictions as the UAV approaches failure, during the latter stages of life.

The focus on accuracy near the end of life is driven by the need to avoid catastrophic failures during operation, and if possible perform replanning actions that allow for the completion of as much of the original mission without sacrificing system safety. To gain insights into LSTM behavior, it is necessary to turn to methods that have been employed in explainable AI (Adadi and Berrada (2018); Sundararajan et al. (2017)). By examining the inner workings of LSTM networks, we can identify the weights and neurons that play pivotal roles in accurate predictions, particularly near the failure point. By assessing the importance of weights and neurons as the system's health deteriorates, we can identify those that contribute most to accurate RUL predictions close to the failure point. This insight aids in identifying critical failure-related patterns that guide predictions. Moreover, distinguishing between significant and insignificant weights and neurons is crucial. During the early stages of UAV operation, the network might emphasize features that correspond to normal functioning. As the system approaches failure, these early-stage features become less relevant, and the network prioritizes different patterns related to degradation. By determining the features, weights, and neurons that contribute minimally to RUL predictions near the failure point, we can more optimally guide the model reduction process to minimize accuracy loss more effectively.

To determine important features critical to accurate predictions near the failure point, gradient-based attribution methods can be quite informative (Ancona et al. (2019)). Gradient-based methods are a class of techniques employed in machine learning and neural networks to extract insights and understanding from complex models. Gradients are mathematical values that tell us how much the output of the model changes when we make a small change in the input features. If the gradient value is large for a feature, it means that changing that feature a little bit will have a big impact on the model's prediction. These methods operate by leveraging the gradients of model outputs with respect to input features, thereby quantifying the sensitivity of predictions to changes in those features. By analyzing these gradients, gradient-based methods aim to uncover the factors that drive model decisions and highlight the most influential components of the input data. These techniques offer a unique window into the inner workings of complex models, shedding light on the relationships between input features and their impact on predictions. Gradient-based methods are versatile tools that provide interpretable explanations for both image and sequential data, enabling users to comprehend how specific features contribute to the model's outcomes. Overall, these methods play a pivotal role in bridging the gap between model complexity and human understanding, empowering practitioners to trust, validate, and improve the reliability of intricate machine learning models.

6.1.1 Improved Saliency Maps

Saliency maps are a gradient-based method used in explainable AI to quantify the contribution of individual features to the final prediction, thereby highlighting features that most influence RUL estimation. Typically, saliency maps are used with CNNs for imaging tasks, where the map is a 2D overlay of the original image (Simonyan et al. (2014)). In the context of LSTMs with time series data, one dimension comprises the features, and the other dimension comprises time steps that help capture the temporal signature of the RUL. Typically, saliency maps are created by calculating the gradients and multiplying the gradient value by the input feature value. This multiplication tells us how much that specific feature contributed to the overall prediction. However, a more informative approach is the use of *integrated gradients*, a method first proposed by Sundararajan et al. (2017). Integrated gradients are an interpretability method used to attribute a model's prediction to its input features by integrating the gradients of the model's output with respect to the difference of the input and a *baseline*. The baseline could be a black image for image networks, a zero embedding vector for language models, or a matrix of random numbers for regression models as in this work. This method provides a way to measure the importance of each feature in influencing the model's prediction. The method was originally formulated for images with a black image as the baseline but here, we can set the

baseline to be a tensor of random values. Integrated gradients are calculated as follows:

$$\mathrm{IG}_i(x) = (x_i - x_i') \times \int_{\alpha=0}^{1} \frac{\partial F(x' + \alpha \times (x - x'))}{\partial x_i'} \, d\alpha, \tag{6.1}$$

where x_i is the value of feature i in the actual input x; x_i' is the value of feature i in the baseline input x'; F is the model's prediction output; α is the integration parameter from 0 to 1, representing the point along the integration path. The right-hand side of the integral tells us how much the model's prediction changes as we vary the feature while moving from the starting point (x', the baseline value) to the actual point (x). This shows the sensitivity of the prediction to changes in that feature. These gradients are integrated, and then multiplied by the difference between the baseline and the actual value and then used to form a saliency map for the RUL estimation model at different stages of life for the UAV.

Figure 6.1 visualizes the effect of the encoded features on RUL estimation at different stages of life and when viewed through saliency maps, it becomes apparent how this effect changes with time. In Figure 6.2, the mean encoded values are presented next to each other, to distinguish the contributions that change the most. These figures were generated from a total of 3,000 data samples randomly taken from the entire dataset with 75 different UAVs, and each data sample represents 2.5 hours of operation.

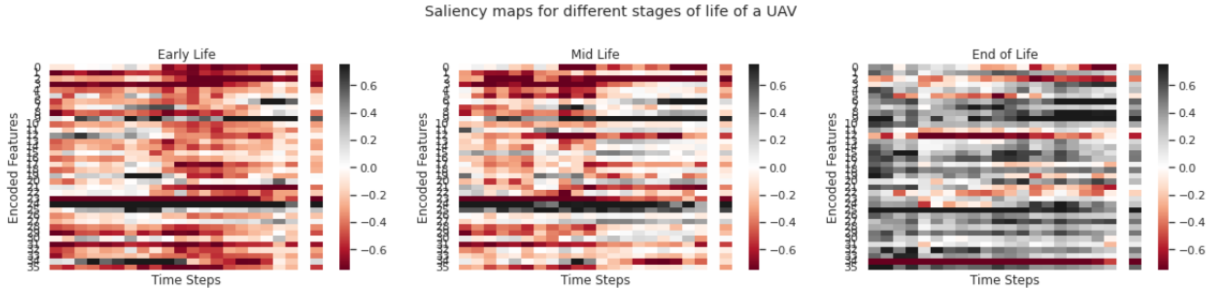

Figure 6.1: Saliency maps for RUL Estimation model at different stages of life. The y-axis represents the encoded features. The x-axis represents timesteps, where each step represents 7.5 minutes, for a total of 2.5 hours of operation per input sequence. Colors towards white represent little influence. Colors towards red or black represent a large influence, either negative or positive.

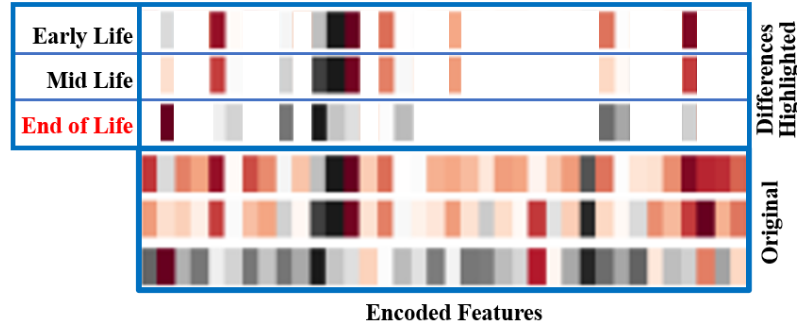

Figure 6.2: Highlighted differences among three stages of life of the mean values of the encoded features from Figure 6.1. The top portion shows the key differences highlighted. The bottom portion shows the original mean values. The key takeaway is that the same feature impacts the RUL estimation differently at different stages of life, where of most importance, is near End of Life (EOL).

The purpose of saliency maps in this study is to demonstrate that the model behaves differently at different stages of life, which has not been proven before in the literature. While informative for understanding the behavior of the network with different input sequences and identifying important/non-important weights at the input layer, saliency maps alone are not enough to explain the inner workings of a neural network (Adebayo et al. (2018)), especially when applied to LSTM networks or with time-series data. Moreover, they are primarily useful when evaluating individual features, as opposed to individual neurons. Since we do not want to remove any of the encoded features, we must look deeper into the model for neuron activations.

6.1.2 Neural Activation Patterns

Neural activity patterns play a crucial role in unraveling the intricate dynamics of neural networks, and like many explainable AI methods were initially developed for CNNs and image classification tasks (Bäuerle et al. (2022)). However, understanding neural activity is just as important for time series prediction tasks. This is especially true for data-driven prognostics, where understanding the temporal dependencies and patterns encoded within neural activation structures is very important to dissect and understand dependencies between the different variables in the system. By analyzing how individual neurons behave and contribute over time, we can decipher how the network reacts to changing input patterns. One of the distinctive strengths of LSTM networks is their capacity to capture long-range time-dependent contextual information. Neural activity patterns offer a glimpse into how neurons communicate across distinct time steps to form a coherent understanding of the data.

Each neuron's participation in the sequence reflects its contribution to the model's final prediction. This understanding enables us to perform precision pruning, which allows us to remove insignificant weights and

neurons in a network more effectively. Neural activity patterns for input sequences near EOL for each layer of the Bi-LSTM are shown below in Figure 6.3. These patterns are calculated from random input sequences sampled from a subset of the training set, where the subset consists of the final 25% of flights before EOL for each UAV in the training set. The resultant color map is on a [0, 1] scale where colors toward white mean the given neuron has little contribution to the output activation relative to the input. Colors towards red have large contributions to the output activation relative to the input.

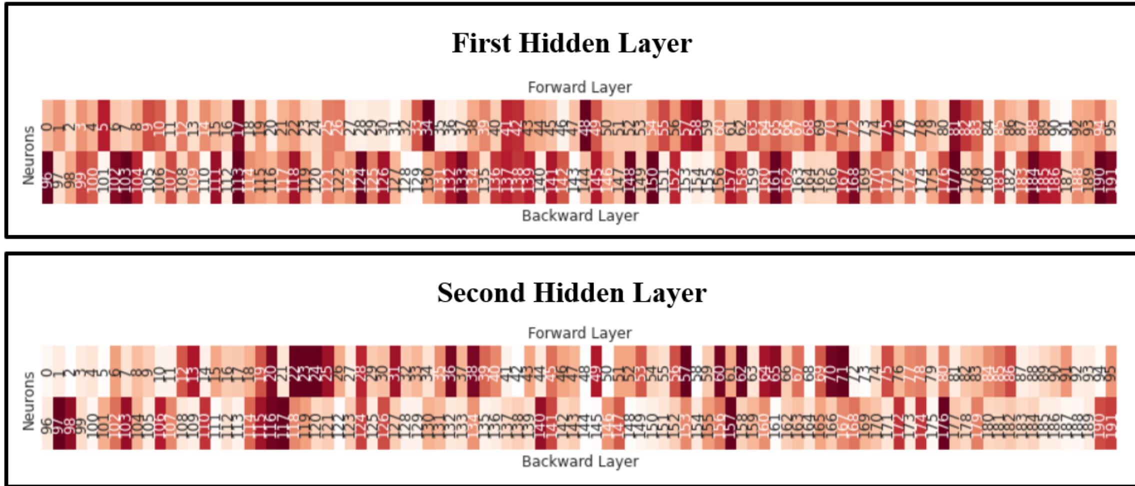

Figure 6.3: Normalized neural activation absolute values of the Bi-LSTM hidden layers. White $\rightarrow$ 0, and dark red $\rightarrow$ 1. Colors towards white show neurons with little activation, where colors towards dark red show maximal activation.

6.2 Compression for LSTM-based Prognostics Models

Information from the neural activation patterns guide the pruning process by grouping the weights and neurons into three distinct sets: (1) weights and neurons that are very important for accurate RUL estimation near EOL; (2) weights and neurons that are not; and (3) weights and neurons whose contribution is somewhere in between. The important set is automatically marked as non-prunable parameters, the unimportant set is automatically pruned, and the middle set goes through an iterative pruning process to find the optimal amount to prune based on the layer-wise distribution of weights until the desired sparsity is reached or when the error grows beyond a desired error level. This is an application-dependent parameter, however, in our exploratory analysis, the error value is disabled to show pruning effects based on sparsity values up to 90%.

Algorithm 6 depicts the activity-based pruning process. The *prune_by_activity* function prunes the forward and backward layers of each Bi-LSTM layer in the network. It takes as input the model, **M**; the neural activations, **A**; the activity threshold to preserve important neurons and mark them as un-prunable, α (set to

.1); two sparsity thresholds, γ_A and γ_W ($\gamma_A \leq \gamma_W$), where γ_A is used to mark the least important neurons to prune (set to .3), and γ_W is used to mark the least important weights to prune (if $\gamma_A \geq \gamma_W$ then $\gamma_A = \gamma_W$); and a boolean value p, to determine whether to preserve the cell state or not. The cell state is what encodes the temporal dependencies in the model, and is like the A-matrix in state space dynamical models. Each threshold is given as a percentage, where α is the percentage of neurons to keep; γ_A is the percentage of neurons to prune; and γ_W is the total desired sparsity level. The algorithm iterates over layers in Line 2, and line 3 checks whether or not the cell state is preserved. If it is, line 4 retrieves the forward and backward weight matrices without the cell state weights, or the cell state weights are included in line 7 otherwise. W_f and W_b contain the matrices for the forget, input, cell state, and output gates, for a total of 8 matrices that undergo the pruning process. The matrices are pruned in lines 9 and 10 by calling the *prune_matrix*, described next, and then the weights are reassigned back to the model in line 11. The model is returned in Line 12.

The *Prune Matrix* function takes as input a list of matrices, **W**; and then the same parameters as described above (A, α, γ_A, γ_W). Lines 15 and 16 capture the indices of the neural activations below and above the respective threshold parameters. Line 15 reads: "the neurons to prune, Z, is the index of all neural activations, i_a, where the absolute activation is less than the value of the threshold percentile, given by the function $f(A, \gamma_A)$". Line 16 is the same, except it finds the neurons to preserve, based on values greater than the alpha percentile, $f(A, \alpha)$. Line 17 iterates through each weight matrix in W, and line 18 defines a temporary matrix that holds the weights of the un-prunable neurons. Line 19 sets the weights of the neurons that are pruned to 0, and line 20 prunes the remainder of the weight matrix via unstructured magnitude-based weight pruning. Line 21 reassigns back the weights for the un-prunable neurons, as in some cases weights of un-prunable neurons could have been pruned in line 20. The matrices are returned in line 22.

Algorithm 6: Activation-Based Pruning Algorithm

1 **Function** `prune_by_activity`($\boldsymbol{M}$, $\boldsymbol{A}$, α, γ_A, γ_M, p)**:**
2   **for** l *in* $\mathbf{M}_{layers}^{Bi-LSTM}$ **do**
3   **if** $p \equiv True$ **then**
4        $W_f, W_b \leftarrow \mathbf{M}(l).get_weights(cell_state = False)$
5   **end**
6   **else**
7        $W_f, W_b \leftarrow \mathbf{M}(l).get_weights()$
8   **end**
9      $w_f \leftarrow$ prune_matrix$(W_f, \mathbf{A_f}, \alpha, \gamma_A, \gamma_M)$
10      $w_b \leftarrow$ prune_matrix$(W_b, \mathbf{A_b}, \alpha, \gamma_A, \gamma_M)$
11      $\mathbf{M}_{W_f, W_b} \leftarrow W_f, W_b$
12   **return** M
13   **end**
14 **Function** `prune_matrix`($\boldsymbol{W}$, $\boldsymbol{A}$, α, γ_A, γ_M)**:**
15    $Z \leftarrow i_a, \forall |a| < f(\mathbf{A}, \gamma_A) \in \mathbf{A}$
16    $K \leftarrow i_a, \forall |a| > f(\mathbf{A}, \alpha) \in \mathbf{A}$
17   **for** m *in* $\mathbf{W}$ **do**
18      $temp \leftarrow m_K$
19      $m_Z \leftarrow 0$
20      $m_{|m| < \gamma_W(m)} \leftarrow 0$
21      $m_K \leftarrow$ temp
22   **return** W
23   **end**

The above algorithm for *neural activity gate-based pruning with cell-state preservation*, **which is specifically tailored to LSTM-based time series regression networks using neural activation patterns**, is a novel approach not previously explored in the literature. It is compared to the baseline method, magnitude based weight pruning (Hassibi et al. (1993) - Algorithm 1).

6.2.1 Experiment Overview

The approach to compress and evaluate the model is now described. Both pruning and quantization are implemented, and the original model is pruned at 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90%. The prune and fine-tune method is implemented, where the original model is initially pruned to 20%, and then a subset of the training set taken from UAVs during the 2nd half of their life is randomly selected for fine-tuning. The models are quantized after pruning, prior to fine-tuning. A small experiment was conducted to find the optimal number of batches and learning rate which resulted in 3 epochs, where each epoch contained approximately 500 training samples and a learning rate of $1e^{-4}$. It is typical to use a smaller learning rate for fine-tuning than when training from scratch (Li et al. (2020a)), as this helps to effectively preserve the

knowledge already encoded in the weights, as opposed to overwriting them. For the same reason, typically, only a few epochs are required for learning, and in this case, overfitting was observed after the 3rd epoch. At each sparsity level, a pruned and quantized model was generated and saved, and the process repeated up to the next sparsity level.

We evaluated a total of four pruning approaches: (1) the baseline method, (2) the activity-based method, (3) the baseline method with cell state preservation, and (4) the activity method with cell state preservation. This is depicted in Figure 6.4 (with the exception of baseline with cell preservation, which is excluded to simplify the figure). Each column represents a neuron, and each cell represents a weight. The depiction shows an ordering of neurons by activation for each method, from least important (left), to most important (right). Each 2-row pair represents the weight matrix for a specific gate in an LSTM network, where the forget gate is the bottom two, the input gates are the two above, and the cell state is represented by the two blue rows (signifying they are not pruned), and the output gate is the top two rows. In the two activation-based methods (B and C), if the sparsity level is below 30%, then the bottom least relevant neurons are pruned by activity, according to the desired sparsity. If the desired sparsity is above 30%, then again the bottom 30% are pruned by activity, and then magnitude-based pruning is used to prune the network up to the desired level. The same is true when the cell state is preserved (C), except that the weights associated with the cell state weight matrices are not pruned. The 30% value is a parameter that can be adjusted, however, 30% strikes a good balance between removing the most irrelevant neurons while also allowing irrelevant weights in neurons with higher activation scores to be removed as well.

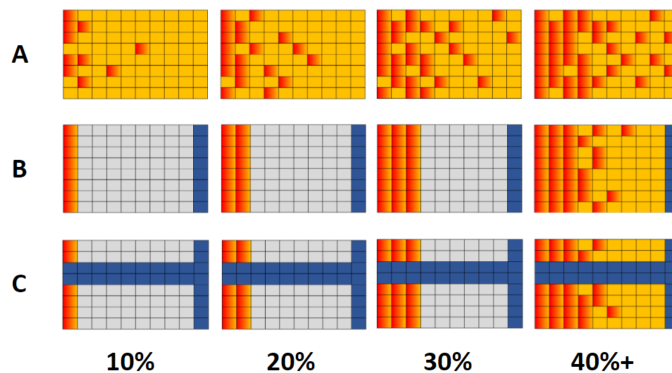

Figure 6.4: Graphical depiction of the different pruning approaches. Blue is preserve, orange means open to pruning, and red means pruned. (A) is the baseline method, magnitude-based weight pruning. (B) is the activation-based method described above. (C) is the activation-based method with preservation of the cell state. 10% represents pruning at 10% sparsity. 20% represents pruning at 20% sparsity. 30% represents pruning at 30% sparsity. 40%+ represents pruning at sparsity levels of 40% or greater, where neural activations are used to prune the bottom 30% neurons and weight magnitudes are used to prune the bottom 10%+ weights of the remaining weights.

This experiment is conducted on both a desktop computer and embedded hardware for reasons discussed on Page 77 that include needing to understanding the difference in execution time and inference speed on different computers. The desktop computer is equipped with A 24 core Ryzen Threadripper 3960X CPU at 3.8 GHz and an Nvidia RTX 3060 GPU with 3584 CUDA cores, a clock speed of 1.78GHz, and 12GB of GDDR6 memory. The model is able to utilize the GPU for computation and take advantage of optimizations suitable for sparse matrix multiplication. The embedded hardware used for the experiments is a Unibab Spacecloud Q7[1] with a dual-core AMD G-series SOC CPU operating at 1.8GHz. These compute modules are radiation tolerant and used for space applications. This module is not equipped with a GPU.

6.3 Results

The estimation results using the 50% compressed RUL model (activation-based with cell-state preservation) are shown below in Figure 6.5. The model has a higher error than the original uncompressed model (see Figure 6.10), and over estimates more as well. Individual prediction plots are shown below in Figure 6.6. In Section 7.5, results are presented using this model in the replanning process where the effect of the overestimation in the target task is minimal.

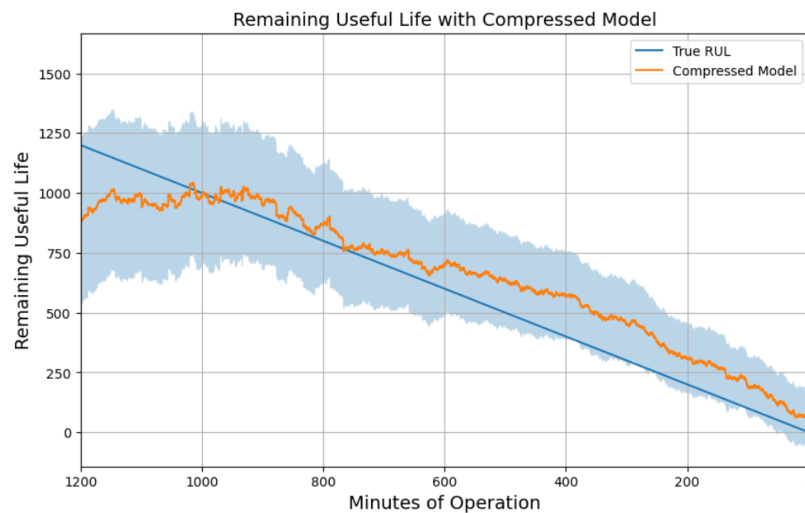

Figure 6.5: RUL estimation for 18 units from the test and validation set.

[1] https://unibap.com/wp-content/uploads/2020/03/1004001-unibap-information-sheet-on-unibap-e2000e2100-modules.pdf

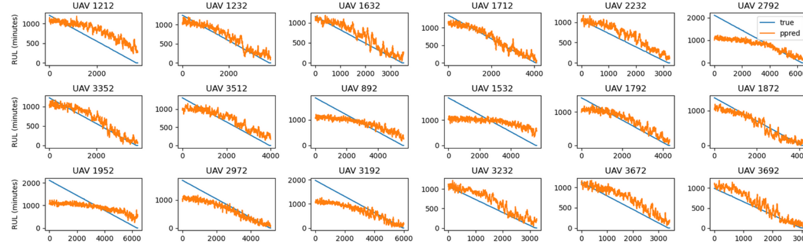

Figure 6.6: RUL estimation for 18 units from the test and validation set.

The methods that preserve the cell state are not capable of achieving sparsity greater than 70% (67.5% sparsity in this method is equivalent to 90% sparsity in the other methods). However, the other two methods are pruned to 80% and 90%, resulting in a total of 32 pruned models. Each of these models is quantized, resulting in an additional 32 models, for a total of 64 models. Each model was evaluated on a desktop computer with a GPU (detailed below in Section 6.3.1 and an embedded computing platform (Section 6.3.2).

There are several key takeaways from the results that are true for both the desktop computer experiment and extend to the hardware experiment that we discuss later. The first is that the quantized models had a higher error rate than the pruned models. This is expected as information is lost during the quantization process. The second result is that INT8 quantization was not achievable. Through multiple different trials, the network diverged after approximately 100 training samples, even when using lower learning rates. We hypothesized that this was due to the fact that the target task was a regression problem, and the target variable (RUL) was continuous. This is different from other sequence tasks, such as language problems where the underlying domain is in fact discrete - a word is a word, and words are distinctly different from one another. CNNs for classification tasks are able to achieve this level quantization and in some cases INT4/INT2 quantization because the underlying task and the network modifications go hand in hand. Quantization has the effect of 'bucketing' results, which is exactly what we desire to do with classification. However with time series regression problems where the target variable is in the continuous domain, the same is not true. This is discussed further in Chapter 8.

The third takeaway is that the activation-based method which preserved the cell state had the lowest error among each of the approaches. This result is an important contribution of the thesis research and opens the door to further studies for compressing LSTM networks for regression tasks. These results support the hypothesis that the cell state weights have a higher influence on the output of the LSTM than the other weight matrices, due to the fact that the cell state is what retains information, and gives the LSTM network its ability to capture long-range dependencies.

The fourth takeaway is more of an explanation of why the error in the quantized networks shoots up initially, and then decreases, level off, and then increases again. This effect was noticed in both the desktop and hardware experiments, across multiple runs. The reasoning for this is in the iterative fine tuning process. At each sparsity level the model is fine tuned, so a quantized model at 50% sparsity, for example, has undergone four iterations of fine tuning, where the model at 20% sparsity, has only undergone one. Then eventually the error rate goes back up because the sparsity level dominates the performance more than fine tuning.

6.3.1 Results on Desktop Computer

The desktop results are shown below in two separate figures. In Figure 6.10, the absolute error (Y axis) is plotted against the model sparsity (X axis) for the pruned models (top), and quantized models (bottom). We can see that the proposed method which preserves the cell state and compresses via neural activations, outperforms the other approaches in both the pruned and quantized models.

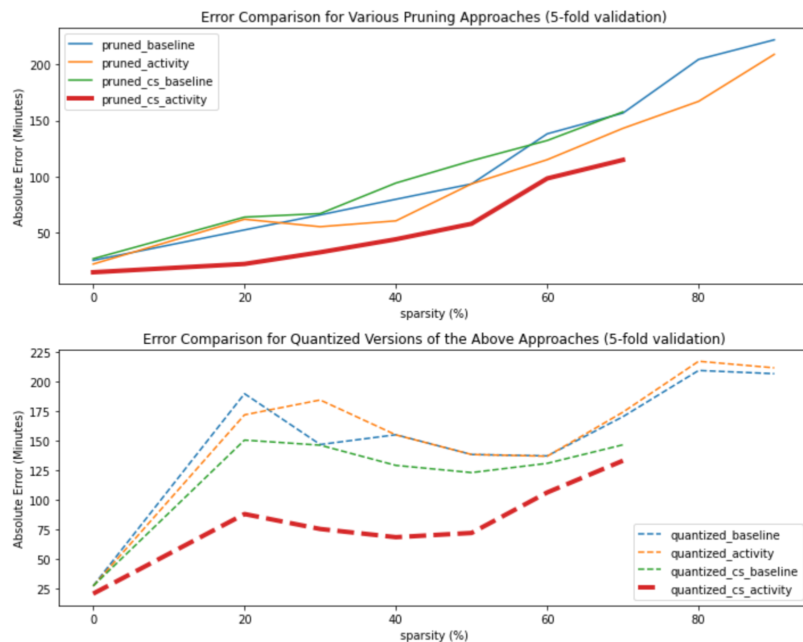

Figure 6.7: Comparison of different model reduction approaches on the desktop computer across varying sparsity levels from 0% (original model) to 90%.

Figure 6.8 is a four-dimensional plot that depicts sparsity level (%), model size (MB), speed (seconds), and error (minutes) for the pruned models (left) and quantized models (right). There is a linear correlation between error, model size, and sparsity, as would be expected. In the same manner, there is a linear correlation between speed, model size, and sparsity, as also would be expected. The max error observed in the pruned models was just over 200 minutes, and the max speedup, achieved at 90% sparsity, was approximately 1.5x.

For the quantized models, the max error observed was nearly 225 minutes, and the max speedup was 2.75x. In general, the quantized models had much higher error overall, but near maximum sparsity around 80% and 90%, the error for the pruned models and quantized models was nearly the same.

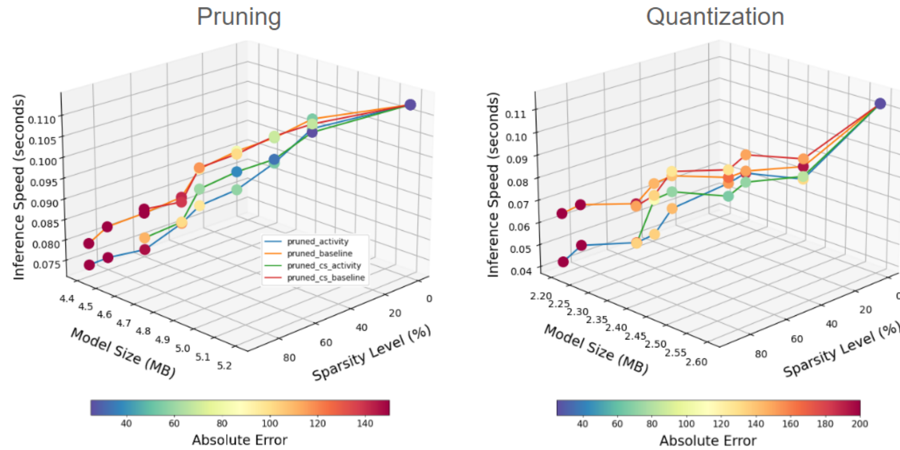

Figure 6.8: 4D plot showing inference speed (Z-axis), model size (Y-axis), sparsity level (X-axis), and absolute error (color).

6.3.2 Results on the Embedded Hardware

The results on the hardware are similar to that of the results on the desktop computer, with a few key distinctions. First, the same amount of speedup (1.5x and 2.75x) was not observed on the hardware testbed. This is due to the fact that CPUs do not efficiently handle sparse matrix multiplication, whereas GPUs are customized for such multiplications.

The hardware testbed achieved a mere 1.1x speedup, and therefore, the accuracy loss far outweighs the inference speedup. Secondly, the accuracy loss was slightly higher on the hardware than on the desktop computer. This is due to the way math operations are executed on the CPU versus the GPU, and various math optimizations that are enabled on the GPU, but not on the embedded CPU (e.g., rounding). Further experiments with GPU-enabled embedded hardware will shed more light on these results.

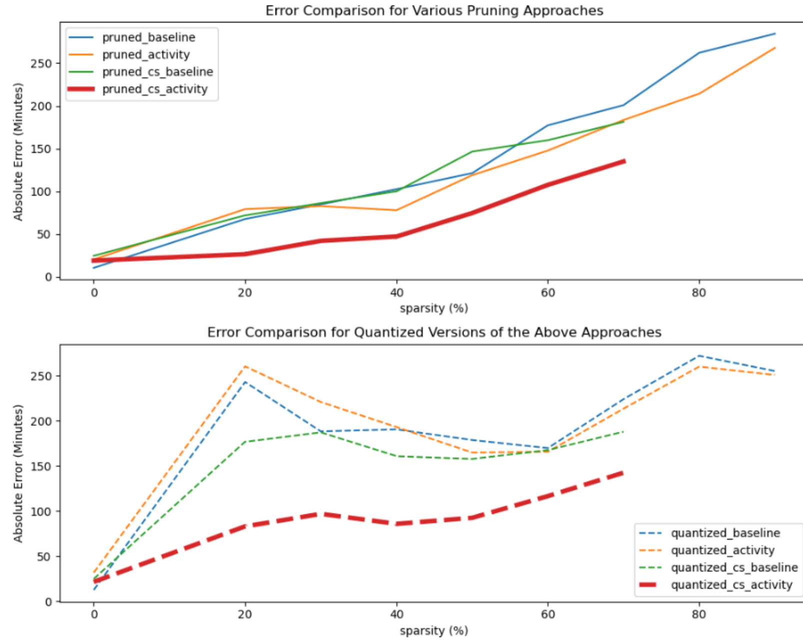

Figure 6.9: Comparison of different model reduction approaches on the Unibap Q7 across varying sparsity levels from 0% (original model) to 90%.

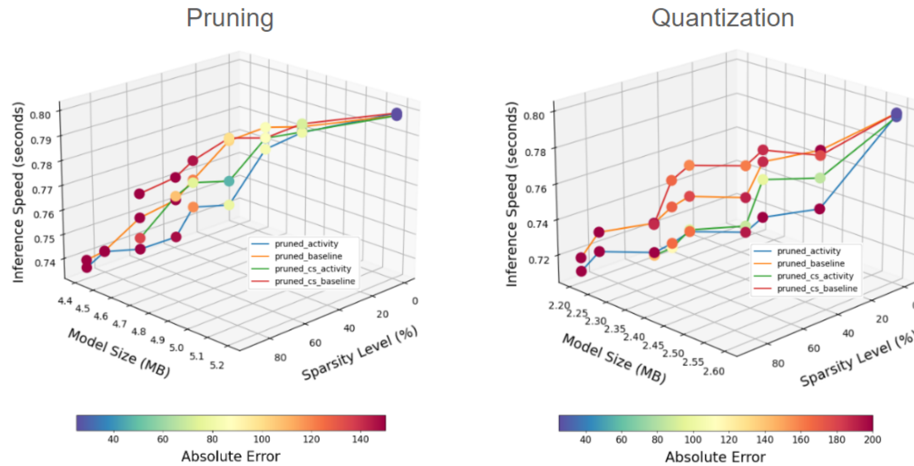

Figure 6.10: 4D plot showing inference speed (Z-axis), model size (Y-axis), sparsity level (X-axis), and absolute error (color) on the Unibap Q7.

For mission designers looking to assess solution quality, model size, and inference speed, this is the tradeoff study that needs to be replicated for their specific task. Deciding which combination of factors fits within the parameters of the target application can only be done by subject matter experts. In this work, UAVs are the target system, and performing RUL estimation is the target application. For this, we choose to use the 50% compressed model under the newly proposed approach and compare the results of replanning using RUL values estimated from these two models in the following chapter.

6.3.3 Summary

This chapter explores a novel approach to model compression for LSTM networks and time-series regression tasks. RUL estimation is inherently a time-series task that exhibits non-stationarity, and it was hypothesised that the model behaves differently at different stages of life. This was proved with a saliency analysis using integrated gradients, and such a proof has not been conducted before. With respect to RUL estimation, accuracy is of most concern towards EOL, and the saliency analysis supported the hypothesis that input samples in the latter stages of life should be used when incorporating explainable AI into the model compression process. This was accomplished with neural activation patterns, which provides insight as to which neurons are important, and which are not. In addition, a novel technique that accounts for the effects of process changes over time on an LSTM network was developed that preserves the cell-state weight matrices during pruning. This resulted in a neural activation, gate-based approach to pruning for LSTM networks, something that has not been done before. Additionally, the compressed model was evaluated on resource-constrained embedded hardware and shown to outperform other compression methods. The tradeoff of this approach is that it cannot be pruned beyond 67.5%, something that mission designers will have to consider for their target application.

CHAPTER 7

Health-Aware Replanning

Unmanned Aerial Vehicles (UAVs) are rapidly becoming an indispensable part of a variety of sectors from commercial to military operations. Ensuring their robust and reliable performance while operating with a multitude of uncertain factors is of critical importance. These factors include environmental conditions such as wind, internal conditions such as degradation, and unanticipated factors such as abrupt faults. *The purpose of this chapter is to present the health-aware framework for replanning, that incorporates system-level health information derived from the prognostics model into the replanning process.* Leveraging the deep learning RUL and auxillary models detailed in Chapter 5, a replanning agent is able to generate safer plans than with a naive battery discharge model while still abiding by safety and performance constraints. This holds even when utilized a reduced model, which is more amenable to deployment on a resource-constrained processor.

The uniqueness of this approach lies in the integration of predictive health monitoring and intelligent replanning. Current fault-based replanning approaches do not incorporate system-level state of health information into the replanning logic, and as a result do not generate plans that maximize remaining operation. On the other hand, the replanning agent may think that more operating time is available than there actually is, and generate a plan that leads to failure. This is initially demonstrated with a comparison of replanning with and without system health information using a modified A* algorithm (Section 7.4). The replanning is formulated as a modified *Orienteering Problem*, which is a type of constraint satisfaction problem that seeks to find a cycle in on the input graph that maximize reward while not violating any constraints. The modification is that we are not finding a cycle, since replanning is triggered from an arbitrary point that is different than the goal location.

7.1 Health Awareness

Health awareness is having system-level SOH knowledge and the functionality to use that knowledge when making decisions or building plans. The online Health-Aware architecture is depicted in Figure 7.1, consisting of an estimation scheme to estimate the system state, $\mathbf{x}$, and degradation parameters, γ; a prediction

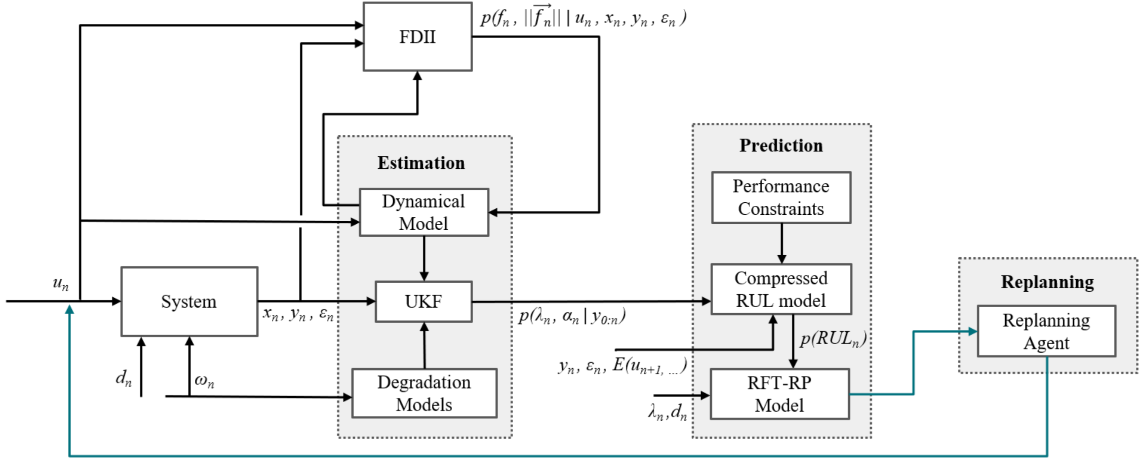

Figure 7.1: Online Architecture for Health-Awareness.

scheme to predict the remaining flight time, t_r, (RUL), and a fault detection, identification, and isolation (FDII) scheme to detect faults and their magnitude, f and $||\vec{\mathbf{f_n}}||$, which are used to recompute state and degradation parameters of the system. The inputs, $\mathbf{u}$, consist of the velocity profile, wind conditions, and motor control signals, and are fed to the system, the dynamical model of the system, the degradation models of the components within the system, and the FDII monitor. The uncertainties and noise, ω, are also input to the system and degradation models as Gaussian distributions. The FDII block also receives the system parameters, ϑ, and outputs fault information back to the dynamical system model to update the parameters of the system if a fault is detected. There is a two-way link between the dynamical system model and the UKF, as the UKF estimates the system parameters and updates them as necessary as well. This is a central part of the Health-Aware architecture. As these estimates are continuously calculated online, the system state and degradation parameters are passed to the RUL model in the prediction step. The RUL model also receives a vector of errors, ε, which are comprised of position errors, arrival time errors, and control errors. In addition, the RUL model takes the future expected load, $E(u_{n+1}...)$, and the performance constraints, which is a set of Boolean statements on both system-level and component-level parameters, such as cumulative position error (system-level), or battery state of charge (component-level). Together, the composition of these pieces form the online architecture for Health-Awareness.

7.1.1 Replanning Framework

The high level goal of replanning is to generate new mission parameters that allow the UAV to operate while still returning to base safely. Often the case with complex systems in safety critical environments, time is of the essence, and mission designers need to know how the algorithms behave on the target embedded hard-

ware. Figure 7.2 shows the running time comparison for one replanning scenario when ran on the embedded hardware compared to the desktop computer. This must be shown first to justify the architecture of the different software components, which is composed of both on-board and off-board components. A more detailed discussion is given in Section 7.5.

The reward function of the replanner is such that original waypoints have a reward value of 5 units, and alternate waypoints have a reward value of 1 unit. Individual solutions are integer based, depicted as dotted lines in the figure. In this example, the theoretical maximum reward possible is 43 units, or 8 waypoints from the original trajectory (out of 9), and 3 additional waypoints. The UAV stops at each waypoint for approximately 10 seconds, and in this case the UAV returns to base safely with less than 8 seconds before failure is reached. Any mission designer will say this is to close for comfort. A solution of 41 reward units, equivalent to 8 out of the original 9 waypoints and one additional waypoint, offers a buffer of approximately 28 seconds of flight time. In Figure 7.2 we can see that the embedded hardware took 27.5 seconds to reach a solution of this quality, while the desktop computer took only 2.5 seconds. Now consider that we are able to communicate with the UAV and the round-trip communication time is 2 seconds. This means that we can move the replanner to the ground control station (GCS) and generate and execute a new plan in under 5 seconds.

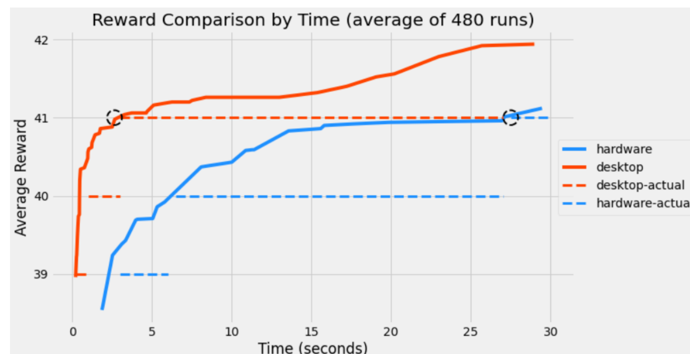

Figure 7.2: Reward comparison between the desktop computer (orange) and embedded hardware (blue) shown for up to 30 seconds of run time using the data from Example Scenario D (see Figure 7.4). Each reward point represents an additional waypoint added to the new trajectory, which represents an increase in utilization while still returning to base safely.

The replanning framework is presented in Figure 7.3, consisting of the off-board replanning agent on the GCS, a pre-departure component (L), and an online component (R). During pre-departure the mission plan is generated on the GCS and is loaded onto the embedded hardware with the performance thresholds. The RUL and auxiliary model that estimates flight time and power consumption are also loaded onto the embedded hardware. Since we are moving the replanning agent to the GCS one my ask why not also move the estimation

models as well. The reasoning is that transmitting data consumes power and if we are constantly transmitting input data for the models then the battery will drain faster. Moreover, the RUL and auxiliary model execute in under a second - and moving these models to the GCS would increase the time to action by 200%. For these reasons, the models remain onboard, and the replanner is moved to the GCS.

During the online phase, the Health-Awareness architecture performs monitoring and estimation tasks. The distance, flight time, a power consumption estimates for each trajectory are known ahead of time. Every 10 seconds the onboard models estimate the RUL, remaining flight time, and remaining power. If at anytime either of these values are less than the original trajectory estimates or if the FDII module detects a fault, the UAV enters a hover mode and replanning agent on the GCS is activated. The GCS receives the UAVs current position and remaining distance, flight time, and power consumption estimates. It performs replanning and then sends the new trajectory back to the UAV, which then resumes flight.

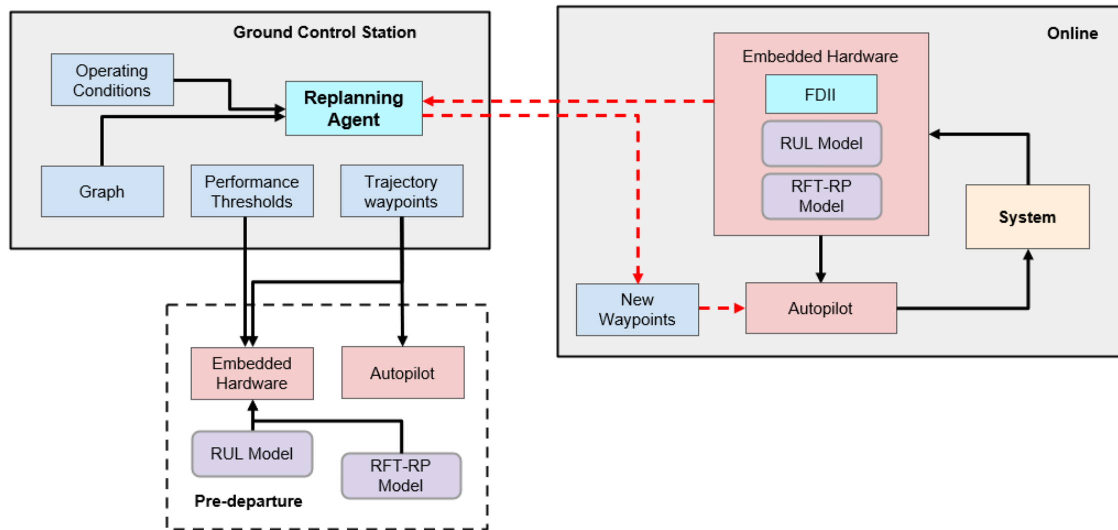

Figure 7.3: Replanning framework. The GCS uploads the trajectory and performance thresholds to the UAV prior to departure. Online, the UAV continually monitors for faults and estimates RUL. When it determines it cannot finish the original trajectory it requests a new plan from the ground station.

7.2 Replanning as a Modified Orienteering Problem

The Orienteering Problem (OP) is a well-known optimization problem in operations research and computer science. It is a combinatorial optimization problem that models the challenge of finding the most profitable route through a graph while visiting a subset of nodes without violating some budgetary constraint (Gunawan et al. (2016)). The formulation of this problem is closely related to the Traveling Salesman Problem (TSP) and

the Knapsack Problem (KP). The TSP is a well-known NP-hard problem that involves finding the shortest possible route visiting all nodes in a graph exactly once and returns to the starting node (Laporte (1992)). Both the OP and the TSP seek to find optimal routes through a graph that visits a set of specified nodes while minimizing cost. However, in the TSP, the primary constraint is that all nodes must be visited exactly once, whereas in the OP the primary constraint is the travel cost.

Theoretically, replanning is no easier than planning (Nebel and Koehler (1993)), and in fact replanning itself *is* a planning problem. As this is the case, the modified OP (a "planning" problem) is a suitable representation to frame the replanning problem of this thesis. We leave the kinematics to a separate solver and focus on replanning the waypoints in a given trajectory. The basic OP can be formulated as 4-tuple of elements, (G, r, v_0, t) where $G = (\mathbf{W}, \mathbf{R})$ is an undirected graph with waypoints, $\mathbf{W}$, representing locations, $\mathbf{W} = [w_1, w_2, ...w_n]$ and edges represented by resources, $\mathbf{R}$, are a tuple of distance, time, and power consumption between each node $R = [e_{ij} \; \forall \; (w_i, w_j) \in \mathbf{W}]$. Each waypoint has an associated reward value, denoted by $v(w)$, $\forall \; w \; \in \mathbf{W}$. The start and goal location are the same for the original plan, $w_s \; = w_g$, but are different for the case of replanning. There is a total resource constraint that must not be violated, r_{max}, which is a tuple of max distance, max flight time, and max power consumption. This defines a graph-based optimization problem, and there are well known solutions for it.

7.2.1 Solutions to The Orienteering Problem

The initial approach taken and described in Section 7.4 uses an A* variant, where the waypoints on the original trajectory are prioritized with higher rewards than alternate waypoints. The problem with A* is that it can be computationally intractable for a large search space, and exhaustively iterates over all possible solutions. There are variations to A* such as anytime A* (Likhachev et al. (2005)), however, in the worst case A* and its variants have time and space complexity of $O(b^d)$, where b is the branching factor and d is the maximum depth.

The *Branch-and-Cut* algorithm is an exact optimization method that combines techniques from the branch-and-bound algorithm and cutting-plane methods to systematically explore the search space and prune branches that cannot lead to the optimal solution. Fischetti et al. (1998) propose an adaptation of the original branch-and-cut algorithm, developed for the travelling salesman problem (Padberg and Rinaldi (1991)), and apply it to the OP as described below. First, they formulate the OP as an *Integer Linear Program* (ILP) using two

decision variables, x_e (for edges) and y_v (for nodes), as follows:

$$x_e = \begin{cases} 1 & \text{if } e \text{ } is \text{ } used, \\ 0 & otherwise; \end{cases}$$

$$y_v = \begin{cases} 1 & \text{if } v \text{ } is \text{ } used, \\ 0 & otherwise. \end{cases}$$

Then, this can be formulated via

$$OP = max \sum_{v \in V} r_v y_v,$$

subject to

$$\sum_{e \in E} e x_e \leq t.$$

The algorithm starts by creating an initial node and performs branching, which creates two child nodes corresponding to two possible values of that variable (e.g., including or excluding a node). These child nodes represent different subproblems. In the bounding phase, the algorithm proceeds to bound the objective function for each subproblem. It uses lower bounds to determine whether each child node has the potential to lead to an optimal solution or if it can be pruned. If a given vertex, v, can lead to an optimal solution, then y_v is set to 1, and $x_{e_{v_{ij}}}$ is also set to 1. If the bounding phase does not yield an optimal solution, cutting planes are added to further tighten the lower bounds and eliminate suboptimal solutions. These cutting planes are constraints that can improve the lower bounds by excluding certain feasible solutions. During the search process, the algorithm prunes branches that are determined to be suboptimal based on the bounds and constraints. It focuses on exploring only those branches that have the potential to lead to the optimal solution, effectively reducing the search space. They implement this approach on several problems and show that upper and lower bounds (computed from the cutting phase) are discovered quite early in the search, improving overall search time.

A *Dynamic Programming* (DP) approach to the sister problem of the OP, the TSP, was first introduced by Bellman (1962). In this formulation, starting at a given city, c_0, proceeding to city c_i, and having to visit k remaining cities $c_1, c_2, ..., c_k$ before returning to c_0, and d_{ij} is the distance from city i to city j, then the minimal path length is defined by

$$f(i; j_1, j_2, ..., j_k) = \min_{1 \leq m \leq k} \{d_{ij_m} + f(i; j_1, j_2, ..., j_{m-1}, j_{m+1}, ..., j_k)\}.$$

Bellman shows that for the case of 21 cities to visit, the distances of 200,000 partial paths would have to be stored, which at the time of this publication, was not possible.

Righini et al. (2006) build off this approach and develop a DP algorithm for the OP with *time windows*, OPTW, an extension to the original problem that specifies time windows for a given node to be visited in. They implement a bi-directional dynamic programming strategy with decremental state space relaxation. In this formulation, multiple visits to a given node are allowed, but the set of nodes that are allowed more than one visit are iteratively reduced. This is a compromise between exact solutions, which can be compute intensive, and fully relaxes state space methods, which do not guarantee the optimal solution will be found, on that a satisfactory one will be. In dynamic programming for path planning, a state for node i is the path starting from the starting node, 0, passing through one or more intermediary nodes, j, and ending at node i, provided that edges along the way exist $(e_{0,j}, e_{j,j+1}, ..., e_{j+n,i} \in E)$. In their work, the authors extend the definition of a state to be a tuple, (S, τ, P, i), where S represents the subset of visited nodes, τ is the overall time elapsed, P is the total reward so far, and i is the current node. Due to each node having a window of time where a valid visit can occur, the authors formulate a forward and backward search strategy, bounded $N/2$, where N is the total number of nodes. When adding a state to the plan, a *dominance* test is administered to ensure only one successor state is recorded.

7.2.2 Genetic Algorithms

GAs are search-based algorithms inspired by the principles of natural evolution and genetics, which include selection, crossover, and mutation (Goldberg (1989)). There are many variations to GAs applied to a wide variety of problems, and for an indepth review the reader is directed to the works by Chahar et al. (2021). The GA process begins with a randomly initialized population, where each individual represents a potential solution to the given problem, in this case, it is a search optimization problem. In each generation, individuals are evaluated based on a fitness function, which quantifies the quality of their solutions with respect to some value or reward in the problem domain. Next, selection operates to choose parents based on fitness, and then the chosen individuals produce offspring through crossover and mutation operations. Crossover combines the parts of the parent solutions into a new solution, and mutation introduces small random changes into the offspring of the parents afterwards. Elitism is used to retain high-performing solutions from generation to generation.

The initial population is a crucial element, as it aids in reducing the time required for the GA to produce an optimal solution (Khaji and Mohammadi (2014)). In addition, initialization can significantly impact the quality of the final solution. Da Silva Arantes et al. (2015) implemented a greedy heuristic in the population

initialization function for a fixed-wing UAV tasked with finding an optimal landing spot and found that it safely landed the UAV 97% of the time, compared to 88% of the time without the greedy heuristic. Gyenes, Zoltán and Bölöni, Ladislau and Szádeczky-Kardoss, Emese Gincsainé (2023) presented a GA for online obstacle avoidance with sub-second execution times. However, the size of the search space was under 100 square meters, which renders the approach not applicable on larger scales. de Moura Souza and Toledo (2020) presented a hybrid GA for path planning of a UAV combined with ray casting for obstacle avoidance. Their approach was implemented on a Raspberry Pi with a maximum timeout of 180 seconds. Unfortunately, 180 seconds of runtime is not feasible for online use. It is important to utilize domain knowledge in this regard when desiging an algorithm to be used online. Generating the initial population in accordance with the goals and constraints of the problem as the first step to developing a GA that rapidly converges on an adequate solution in time-critical applications. This is especially true for replanning. The fast-start, adaptive genetic algorithm for constrained routing with rewards is discussed next.

7.3 Fast-Start Adaptive Genetic Algorithm for Constrained Routing with Rewards

We first begin by providing an overview of the algorithm. The GA algorithm is comprised of several helper functions that initialize the population, calculate target fitness, calculate candidate fitness, perform the crossover and mutation operations, calculate population diversity, and calculate the adaptive rates for crossover, mutation, and elitism. These functions are given in Algorithms 7 - 13, followed by the GA itself in Algorithm 14.

The replanning problem is represented by a graph, $\mathbf{G} = (\mathbf{W}, \mathbf{R})$, where the vertices, $\mathbf{W}$, are all of the possible waypoints, and the edges, $\mathbf{R}$, are the resources it takes to travel between any two waypoints. Each edge in $\mathbf{R}$ is a tuple, $r = (d, t, p)$, which represents the segment distance, flight time, and power consumption. Each waypoint in $\mathbf{W}$ has an associated value, denoted by w^V. In addition to the graph, there is the starting waypoint, w_s, which is the current location of the UAV at the time of replanning (the UAV enters a hover mode to replan), and the goal waypoint, w_g, which is the goal waypoint from the original trajectory, the same as the start waypoint from the original trajectory. Next are the resource threshold values for distance, time, and power, represented by a tuple $r_{max} = (d_{max}, t_{max}, p_{max})$. Finally we have the original trajectory, $\mathbf{T}$, which is represented by a series of intermediate path points from waypoint to waypoint, beginning with the starting waypoint and ending at the goal waypoint. The algorithm is now detailed in the following subsections.

7.3.1 Population Initialization

The population initialization function, given in Algorithm 7, is where the *fast-start* property of the algorithm is contained. The algorithm is unique in that it prioritizes high-reward, close to the original path waypoints via the value function, and that each candidate solution is initialized by sampling from the remaining waypoints in the trajectory. To define what we mean by "close to the original path" we need to introduce a cutoff value, c, for which we use to consider waypoints to begin with. This allows us to restrict the search space and speed up the execution time of the algorithm. Formally, this is

$$W' = w \in \mathbf{W} : \exists D(w,p) \leq c, \forall p \in P, \forall w \in \mathbf{W}$$

,

where W' is the reduced set of waypoints, w are individual waypoints in the set of all waypoints $\mathbf{W}$, p are intermediate points along the path P, $D(w,p)$ is the distance between a waypoint w and an intermediate path point p, and c is the distance cutoff with which to ignore or consider waypoints. Moving forward, $\mathbf{W}$ will denote the reduced set of waypoints used by the replanning agent.

The algorithm takes as input the population size, the graph, $\mathbf{G}$, the starting waypoint, w_s, the goal waypoint, w_g, the original trajectory, $\mathbf{T}$, and the resource threshold values, r_{max}. The *values*, line 2, is the value function used to guide the initial population generation process which is defined by the reward-per-waypoint to distance-to-trajectory ratio. In line 3 the population is initialized to an empty set and in line 4 the population outerloop begins. A candidate, C, is initialized in line 5 by selecting a random number of waypoints from the remaining waypoints in the original trajectory, while maintaining the initial ordering. The accumulated resources are initialized in line 6.

The inner loop starts on line 7, which builds an individual candidate solution. Lines 8-10 select a waypoint via weighted sampling from the *values* (the value function, line 2). Line 11 assigns the current resource utilization to r, which on the first iteration will be 0. Lines 12-17 accumulate these values based on if its a new candidate, line 13, or line 14 if not. Then in line 18, the resource thresholds, r_{max}, are checked, and the current waypoint is added to the candidate if the check is satisfied in line 19. The total resource utilization is updated in line 20. In line 23 the candidate is added to the population.

Algorithm 7: Generate initial population for GA

1 **Function** `generate_initial_population`(*population-size,* $\mathbf{G}(\mathbf{W},\mathbf{R}),\mathbf{T},w_s,w_g,r_{max}$)) **:**
2 $values \leftarrow \frac{w^V_{1:|\mathbf{W}|}}{dist(\mathbf{T}_{1:|\mathbf{W}|})}, \forall\, w \,\in\, \mathbf{W}$
3 population $\leftarrow \emptyset$
4 **for** *i = 1 ... population-size* **do**
5 $C \sim \mathscr{U}(\mathbf{W},[0,|\mathbf{W}|-2])$
6 $r \leftarrow (0,0,0)$
7 **while** $r \,<\, r_{max}$ **do**
8 $E(w) \leftarrow \frac{values(w)}{\sum values}$
9 $wp \sim (waypoints, E(w))$
10 $waypoints \leftarrow waypoints - wp$
11 $r` \leftarrow r$
12 **if** $|C| \equiv 0$ **then**
13 $r`+= \mathbf{R}(w_s, wp)$
14 **end**
15 **else**
16 $r`+= \mathbf{R}(C_{-1}, wp)$
17 **end**
18 **if** $r`$ + $\mathbf{R}(wp,w_g) \leq r_{max}$ **then**
19 $C \leftarrow C \cup wp$
20 $r \leftarrow r`$
21 **end**
22 **end**
23 population $\leftarrow$ population $\cup\, C$
24 **end**
25 **return** population

7.3.2 Calculate Target Fitness

The next function, shown in Algorithm 8, calculates the target fitness that the average fitness of the population should exceed. This value is to allow the parameters of the algorithm to dynamically change in real time based on the fitness of the overall population. One approach would be to calculate the reward per unit distance based on the total reward available in the search space and the max distance. However, this assumes an even distribution of rewards and travel segment distances, and there are many cases where this approach will yield a poor estimate. A better estimation of the target fitness would account for the fact that some waypoints might provide more reward per unit distance than others. An improved approach involves sorting the waypoints based on their reward-to-distance ratio, using the distances from the starting waypoint and the goal waypoint. This is the approach given in Algorithm 8. The waypoints are sorted according to their *values* in line 2. In line 3 fitness, f, is initialized, and in line 4 the resources, r, are initialized. The main loop begins on line 5, which iterates through the sorted list of waypoints and gets the distance, time, and power for the

selected waypoint in line 6. Line 7 checks the thresholds if the thresholds are not violated then the fitness and and threshold values are accumulated in lines 8 and 9. The algorithm terminates when a resource threshold exceed the max allowable value, and the target fitness is returned in line 15.

Algorithm 8: Calculate target fitness

```
1  Function calculate_target_fitness(G(W,R), w_s, w_g, r_max, values):
2      waypoints ← sort(W, values(W))
3      f ← 0
4      r ← (0,0,0)
5      for each wp ∈ waypoints do
6          r‘ ← R(w_s, wp) + R(wp, w_g)
7          if r + r‘ ≤ r_max then
8              f ← f + w^V
9              r ← r + r‘
10         end
11         else
12             break
13         end
14     end
15     return f
```

7.3.3 Calculate Candidate Fitness

The fitness of each candidate solution is calculated in the next function, given in Algorithm 9. Line 2 calculates the total resource consumption of the candidate. The fitness is the total reward of the candidate if the resource threshold, r_{max}, is not exceeded, otherwise it is zero (line 3).

Algorithm 9: Calculate fitness

1 **Function** `calculate_fitness`$(C, \mathbf{G}(\mathbf{W},\mathbf{R}), w_s, w_g, r_{max})$ **:**

2 $\quad r \leftarrow \sum_{w_{i:n} \in C} \mathbf{R}_{w_i,w_j} + R_{w_s,w_0} + R_{w_n,w_g}$

3 $$f = \begin{cases} \sum_{w_{i:n} \in C} w^V & r \leq r_{max} \\ 0 & r > r_{max} \end{cases}$$

4 $\quad$ **return** f

7.3.4 Crossover & Mutation

The crossover function (Algorithm 10) takes as input two candidates, C_1 and C_2, (i.e. *parents*), the crossover rate, x_{rate}, the resource thresholds, r_{max}, and the resource matrix, $\mathbf{R}$, to perform the *crossover* operation. If a

random number is less than the crossover rate (line 2), then the first half of C_1 is combined with the second half of C_2 in line 3. Then, if the total resources consumed by the new candidate does not exceed the resource thresholds (line 4), then the new candidate is returned in line 5. If the crossover operation does not happen, then a parent is selected at random and returned in line 8.

Algorithm 10: Crossover function

```
1 Function crossover(C_1, C_2, x_rate, r_max, R):
2     if random() ≤ x_rate then
3         C_new ← C_1^{1:|C_1|/2} + C_2^{|C_1|/2:n}
4         if Σ_{w_i,w_j ∈ C_new} R(w_i, w_j) ≤ r_max then
5             return C_new
6         end
7     end
8     return random(C_1, C_2)
```

Algorithm 11 performs the *mutation* operation, and takes as input a candidate, C, the list of waypoints (excluding the start and goal waypoints, $\mathbf{W}$, the mutation rate, m_{rate}, the resource thresholds, r_{max}, and the resource matrix, $\mathbf{R}$. Mutation operations operate on individual waypoints, or *genes*, within the candidate solution, using waypoints from the candidate, $w \in C$ or waypoints in the set of waypoints, $w \in \mathbf{W}$. If a random number is less than the mutation rate then a mutation operation is randomly selected (lines 2 and 3). To ensure maximal entropy in the resultant population, multiple mutation operations are implemented. The *addition* operation adds a waypoint to the candidate, C_{new} (line 5); the *subtraction* operation removes a waypoint from the candidate, C_{new} (line 7); and the *replace* operation replaces a waypoint in C_{new} (lines 9-11). If the total resource consumption is less than the resource thresholds then the new candidate is returned (lines 12-13). If the mutation operation doesn't take place or if the resource thresholds are exceeded then the original candidate is returned (line 16).

Algorithm 11: Mutation function

```
1  Function mutate(C, W, m_rate, r_max, R):
2      if random() ≤ m_rate then
3          op ← random(add, subtract, replace)
4          if op = add then
5              C_new ← C + random(w, w ∈ W, w ∉ C)
6          else if op = subtract then
7              C_new ← C − random(w, w ∈ C)
8          else if op = replace then
9              idx ← random(|C|)
10             C_new ← C
11             C_new^idx ← random(w, w ∈ W, w ∉ C)
12         if Σ_{w_i,w_j ∈ C_new} R(w_i, w_j) ≤ r_max then
13             return C_new
14         end
15     end
16     return C
```

7.3.5 Population Diversity

Algorithm 12 calculates the *population diversity*, which is one of the metrics used to guide the adaptive crossover and mutation processes, as well as regenerate the population if necessary. The diversity of a population is a measure of how different the individuals in the population are from each other. There are many ways to measure this diversity, one common method is the pairwise Hamming distance, which compares the similarity of two strings based on positional encodings. However, calculating the Hamming distance between each pair of candidates in the population is computationally costly, as it has a time complexity of $O(n^2m)$, where n is the population size and m is the length of the candidates. Such calculation is not feasible for online use. Alternatively, a frequency based approach is implemented, whereby the occurrences of genes (waypoints) and length of candidate solutions are tabulated. If every gene is equally likely, the population has high diversity. If some genes are much more common than others, the population has low diversity. This approach has a time complexity of $O(n*m)$, where n is the population size and m is the average size of the candidate, and is significantly faster than the Hamming distance method. In lines 2-4, the frequency set (F), accumulated length (L), and diversity (δ) are initialized. F is a dictionary-like structure that maintains an occurrence count per waypoint. L is a running sum of the candidate solution lengths for each candidate, C, in the population, $\mathbf{P}$. The diversity, δ, is on a 0 to 1 range where values less than .5 are diverse, and values greater than .5 are similar. The outer loop begins on line 5, iterating through each candidate in the population and accumulating the length, L, on line 6. The inner loop begins on line 7, iterating through each waypoint in the candidate, and updating the frequency set in line 8. Line 11 calculates the average candidate length, and

line 12 normalizes the frequency values by the size of the population. Line 13 computes the raw diversity as the sum of the squared normalized frequency values, and in line 14 the diversity is normalized by the average length of the candidates in the population, making the metric compatible with different populations throughout the execution of the main algorithm.

Algorithm 12: Calculate population diversity

```
1  Function calculate_diversity(P):
2      F ← ∅
3      L ← 0
4      δ ← 0
5      for C ∈ P do
6          L ← L + |C|
7          for w ∈ C do
8              F_w = { 1        w ∉ F
                     { F_w + 1  otherwise
9          end
10     end
11     L ← L / |P|
12     F_w ← F_w / |P|, ∀_w ∈ F
13     δ ← Σ_{w∈F} F_w^2
14     return δ / L
```

7.3.6 Adaptive Rates

Adaptive rates are calculated for crossover, mutation, and elitism based on the algorithms performance. The algorithms performance is characterized by the average population fitness, the population diversity, and the number of iterations since last improvement. This function is given in Algorithm 13. It takes as input the average population fitness, $\bar{f}$; the target fitness, f^*; the population diversity, δ; the GA parameters, θ (x_{rate} is the crossover rate, m_{rate} is the mutation rate, e_{rate} is the elitism rate, α is the crossover and mutation step size, and β is the elitism step size); and the number of iterations since last improvement, i. The first if block, lines 2-5, executes when the algorithm hasn't reached or exceeded the target fitness, has low diversity, or isn't improving. The crossover and mutation rates increase while the elitism rate decreases. The second if block, lines 6-9, execute when the target fitness has been reached and the algorithm is improving, and decreases the crossover and mutation rates, while increasing the elitism rate. The else block, lines 10-13, execute when a "good" solution has been found and the population is diverse, but the solution isn't improving. This effectively resets the crossover and elitism rates while increasing the mutation rate. This is because both crossover and elitism utilize information within the population, but if the population is diverse but not producing better

solutions, then we want to impart new candidates into the population using mutation instead. The parameters θ_α and θ_β are set at .05 and .02 (found through experimentation). This means that on a range of .1 to 1, the crossover and mutation rates take 18 generations to traverse the entire range. The elitism rate is on a range of .1 to .5 and takes 20 generations to traverse the entire range.

Algorithm 13: Calculate adaptive rates for crossover, mutation, and elitism

```
1  Function calculate_dynamic_rates(f̄, f*, δ, θ, i):
2      if f̄ < f* or δ > .5 or i ≥ θ_I then
3          θ_xrate ← min(1, θ_xrate + θ_α)
4          θ_mrate ← min(1, θ_mrate + θ_α)
5          θ_erate ← max(.1, θ_erate − θ_β)
6      else if f̄ ≥ f* and i < θ_I then
7          θ_xrate ← max(.1, θ_xrate − θ_α)
8          θ_mrate ← max(.1, θ_mrate − θ_α)
9          θ_erate ← min(.5, θ_erate − θ_β)
10     else
11         θ_xrate ← .5
12         θ_mrate ← min(1, θ_mrate + θ_α)
13         θ_erate ← .2
14     return θ_xrate, θ_mrate, θ_erate
```

7.3.7 The Genetic Algorithm

The GA, given below in Algorithm 14, uses the above functions and implements the *Fast-Start Adaptive Genetic Algorithm for Constrained Routing with Rewards*. It is adaptive in that the crossover, mutation, and elitism rates change with the behavior of the population. Moreover, a population restart functionality is implemented, whereby if the algorithm hasn't reach a satisfactory level of fitness after a given amount of iterations for improvement, a new population is generated. The algorithm is given below. Lines 1-7 initialize the best candidate, C^*, the fitness values, F, the iterations for improvement and the improvement counter, I & i, the *reset* counter, the target fitness values, f^*, and the population, $\mathbf{P}$. The main loop begins on line 7 at iterates for θ_N generations. The fitness for each candidate in the population is calculated in line 8, and the average fitness, $\bar{f}$, is calculated in line 9. The population diversity, δ, is calculated in line 10, and line 11 contains the adaptive check. If the number of iterations for improvement have passed without improvement, or if the population diversity is poor (values > .5 are poor) and the average fitness is less than the target fitness, then a new population is initialized in line 12, and the reset counter is incremented in line 13. The only stopping condition outside of loop termination is in line 14, where the algorithm terminates if the population has been reinitialized 3 times. Otherwise, in line 16 the crossover, mutation, and elite rates, $\theta_{x_{rate}}$, $\theta_{m_{rate}}$, $\theta_{e_{rate}}$, respectively, are calculated.

Elitism is performed in line 18 by sorting the population based on fitness, and selecting the top N candidates, where $N = \theta_{e_{rate}}\ x\ |\mathbf{P}|$, the elitism rate multiplied by the length of the population. Next, in lines 19-23 the crossover operation is performed. First, a weighted sampling operation selects two random candidates, C_a & C_b, weighted by the population fitness. Then, the two candidates are crossed over and added to the offspring set in line 22. Line 24 performs mutation, mutating all candidates in the offspring set, whereby some candidates are returned unaltered based on the mutation rate, $\theta_{m_{rate}}$. The existing population is updated in line 25 with the elites and offspring, and in 26 the best potential candidate, $\hat{C}$ is taken. Line 27 performs two checks to determine the best candidate, C^*. If the fitness of the potential candidate is equal to the fitness of best candidate, C^*, and the resource consumption of the potential candidate is less than the resource consumption of the best candidate, then the best candidate is replaced. Second, if the fitness of the potential candidate is greater than the fitness of the best candidate and the resource consumption of the potential candidate is less than the resource thresholds, then the best candidate is replaced. If either check is satisfied, then the best candidate is updated in line 28, and the improvement and reset counters are reset in lines 29 and 30. Otherwise, the improvement counter increments in line 32. The algorithm returns the best candidate in line 35.

The GA parameters are given below in Table 7.1.

Parameter	**Symbol**	**value**
number of generations	θ_N	200
population size	θ_{size}	500
improvement iterations	θ_I	20
resets to halt	θ_{reset}	3
crossover rate	$\theta_{x_{rate}}$	1.0
mutation rate	$\theta_{m_{rate}}$	.25
elitism rate	$\theta_{e_{rate}}$	.2
rate increment 1	θ_α	.05
rate increment 2	θ_β	.02

Table 7.1: GA search parameters.

Algorithm 14: Adaptive Genetic Algorithm for Constrained Path Planning with Rewards

Input: The graph, $\mathbf{G}$, and GA parameters, θ
Output: C^*, the best candidate

1. $C^* \leftarrow \emptyset$
2. $F \leftarrow \emptyset$
3. $I, i \leftarrow \theta_I, 1$
4. $reset \leftarrow 0$
5. $f^* \leftarrow calculate_target_fitness(\mathbf{G})$
6. $\mathbf{P} \leftarrow generate_initial_population(\theta_{size}, \mathbf{G})$
7. **for** $j = 1 : \theta_N$ **do**
8. $F \leftarrow calculate_fitness(C, \mathbf{G}),\ \forall_C \in \mathbf{P}$
9. $\bar{f} \leftarrow \frac{\sum F}{|\mathbf{P}|}$
10. $\delta \leftarrow calculate_diversity(\mathbf{P})$
11.   **if** $i \mod I \equiv 0$ or $(\delta > .75$ and $\bar{f} < f^*)$ **then**
12. $\mathbf{P} \leftarrow generate_initial_population(\theta_{size}, \mathbf{G})$
13. $reset \leftarrow reset + 1$
14.   **if** $reset \equiv \theta_{reset}$ **then** break
15.   **else**
16. $\theta_{x_{rate}}, \theta_{m_{rate}}, \theta_{e_{rate}} \leftarrow calculate_dynamic_rates(\bar{f}, f^*, \delta, \theta, i)$
17.   **end**
18. $elites \leftarrow sort(\mathbf{P}, F)_{1:\theta_{e_{rate}}\, x\, |\mathbf{P}|}$
19. $offspring \leftarrow \emptyset$
20.   **for** $n = 1 : |\mathbf{P}| - |elites|$ **do**
21. $C_a, C_b \sim (\mathbf{P}, F)$
22. $offspring \leftarrow offspring \cup crossover(C_a, C_b, \theta_{x_{rate}}, \mathbf{G})$
23.   **end**
24. $offspring \leftarrow mutate(C, \mathbf{G}, \theta_{m_{rate}})\ \forall_C \in offspring$
25. $\mathbf{P} \leftarrow elites \cup offspring$
26. $\hat{C} \leftarrow max(\mathbf{P}, F)$
27.   **if** $(F(\hat{C}) \equiv F(C^*)$ and $\mathbf{R}(\hat{C}) < \mathbf{R}(C^*))$ or $((F(\hat{C}) > F(C^*)$ and $\mathbf{R}(\hat{C}) \leq r_{max})$ **then**
28. $C^* \leftarrow \hat{C}$
29. $i \leftarrow 1$
30. $reset \leftarrow 0$
31.   **else**
32. $i \leftarrow i + 1$
33.   **end**
34. **end**
35. **return** C^*

An experimental demonstration of the GA performance for general replanning case can be found in the Appendix (Section 9.3).

7.4 Experiments

A series of experiments are conducted to demonstrate the health-aware replanning framework under different scenarios that include different trajectories, different UAVs with different component degradation levels,

Parameter	description	Initial	Actual
Q	Battery Capacitance	22.0 aH	20.95 aH
R_0	Battery resistance	.0011 Ω	.0043 Ω
R_{m1}	Motor 1 resistance	.2371 Ω	.2674 Ω
R_{m2}	Motor 2 resistance	.2370 Ω	.2709 Ω
R_{m3}	Motor 3 resistance	.2371 Ω	.2799 Ω
R_{m4}	Motor 4 resistance	.2372 Ω	.2703 Ω
R_{m5}	Motor 5 resistance	.2369 Ω	.2727 Ω
R_{m6}	Motor 6 resistance	.2371 Ω	.2671 Ω
R_{m7}	Motor 7 resistance	.2369 Ω	.2806 Ω
R_{m8}	Motor 8 resistance	.2374 Ω	.2684 Ω

Table 7.2: System degradation parameters

different wind conditions, different fault onset times, and different faults. This approach is designed to work with faults that affect the overall operating time of the UAV, which include the loss of a battery cell, a parasitic load, and inability to maintain the set-point speed. The first experiment demonstrates the health aware framework in a comparative study of replanning with a simple battery discharge model and with the RUL estimation model using an A* algorithm. The remainder of the experiments demonstrate the performance of the compressed RUL model and the GA algorithm for replanning.

7.4.1 Demonstration of Health Awareness with Uncompressed Model & A*

The Health-Aware replanning framework is demonstrated by showing the effects of replanning for a UAV while undergoing usage-based degradation and an abrupt fault. A UAV is selected that has completed between 40 and 60 flights, having flown at least 100km, with the relevant degradation parameters shown in Table 7.2.

The abrupt fault implement is the loss of a battery cell, and for a LiPo 6S battery this results in the total charge capacitance being reduced by $\frac{1}{6}$. An exemplary trajectory is shown in Figure 1.1, with the AO, trajectory, and waypoints depicted for a successful flight in the top left. Alternate waypoints are depicted that could be considered during replanning. The bottom left depicts the battery state of charge and output voltage for a nominal flight. The right half of the figure shows what happens when a fault occurs with no replanning at all, where the UAV ultimately fails mid-flight, and the effects of the abrupt loss of a battery cell are evident.

The remaining useful life calculation is the primary input to the replanner. Current industry standards use the battery discharge rate to determine max flight time, and by this method the remaining useful life can be calculated by

$$t_r = t_m - t_m \cdot \frac{(\Sigma i_c)}{Q},$$

where t_r is the time remaining, t_m is the max flight time (typically provided by the manufacturer), Σi_c is the cumulative current consumed, and Q is the total charge capacitance. Under ideal conditions ignoring degradation, this is a perfectly acceptable method of calculating remaining flight time. However, degradation is a natural phenomenon that occurs in all complex systems and therefore must be accounted for. A functional mapping between system operation and system state of health must be incorporated into a replanning agent to provide more accurate flight time estimates. Being that the battery discharge model does offer this type of information, it results in an overestimation of remaining flight time, and the UAV fails. This is discussed in more detail in Section 7.5.

7.4.2 Demonstration of Compressed Models with Fast-Start GA

The remainder of the experiments use the compressed model and the GA presented above in Section 7.3. The GA was tested on the desktop computer as well as the embedded hardware. The experiments are described using one example case to frame the approach, and further example cases can be found in the appendix. *In these experiments, we are evaluating the efficacy of the compressed model to assess the impact of information loss due to compression when used in the target application.* The scenario for the experiments will now be described.

The trajectory, UAV, fault time, fault type, and fault magnitude are determined randomly. There are two modes of testing: Mode 1 executes the flight without replanning to show how many original waypoints the UAV can hit and where UAV failure occurs; Mode 2 executes the flight and triggers the replanning agent after the fault has occured. Figure 7.4 depicts a typical replanning scenario, which will now be described moving from left to right. The left plot shows the UAV fly the original trajectory without replanning. The UAV starts in the upper left corner (green dot), and moves counter clockwise. There are a total of 12 waypoints (blue dots), and the fault (battery cell loss) occurs shortly after reaching the 3rd waypoint (black X), 207 seconds into the flight. Without replanning, the UAV is able to reach all 12 waypoints but fails shortly before returning to base (red X). The middle plot of Figure 7.4 depicts waypoints that are ignored by the replanner (red circles), waypoints that are considered by the replanner (green circles), and the new trajectory (purple line). The *distance-to-path* is a parameter of the replanner which is used to delineate waypoints that are

considered and waypoints that are ignored. In this case, that value was set at 60 meters. The right plot shows that UAV flew the new trajectory and reached base safely (green triangle). We can also see that an original waypoint was removed (red circle), and two alternate waypoints were added (green circles). The rest of these experiments can be found in the appendix.

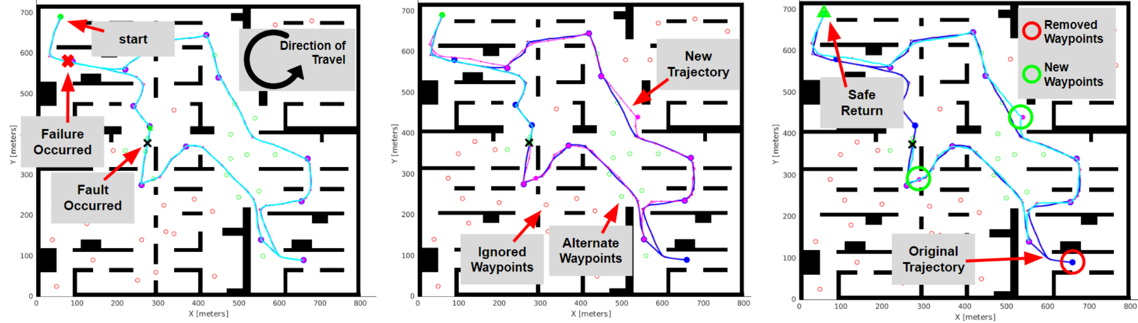

Figure 7.4: Replanning scenario D. Left: The UAV does not enter replanning mode and fails shortly before reaching base. Middle: The replanner is activated and selects a new trajectory using waypoints near the existing flight path. Right: The UAV executes the new trajectory and returns to base safely. The replanner discared one waypoint from the original trajectory, and added two additional waypoints.

7.4.3 GA Performance Analysis

A performance test of the GA was conducted to determine the relationship between the population size, number of generations, solution quality, and run time. This was initially discussed above with a comparison plot between the hardware and desktop computer in Figure 7.2. The test will now be described, with the above example scenario as context. In the above scenario, the original waypoint had 12 waypoints total, and 9 remaining waypoints after the failure occured. Of the remaining original waypoints, the replanning agent is allowed to select at maximum 8, or 1 less than the remaining total. The replanning agent was executed a total of 10 times for each population size and number of generation combinations taken from the following:

$$population_size \leftarrow [100, 250, 500, 1000, 1500, 2000]$$

$$num_generations \leftarrow [25, 50, 75, 100, 125, 150, 175, 200]$$

This results in a total of 480 runs of the GA, executed for the above scenario on both the embedded hardware and desktop computer. The purpose of this was to assess the performance of the algorithm on different computing platforms for an arbitrary case. Next, on the desktop only, the same performance test was performed for each other scenario with a goal of finding the optimal population size and number of generations across the general case. Are these values the same across different problems when controlling for execution time?

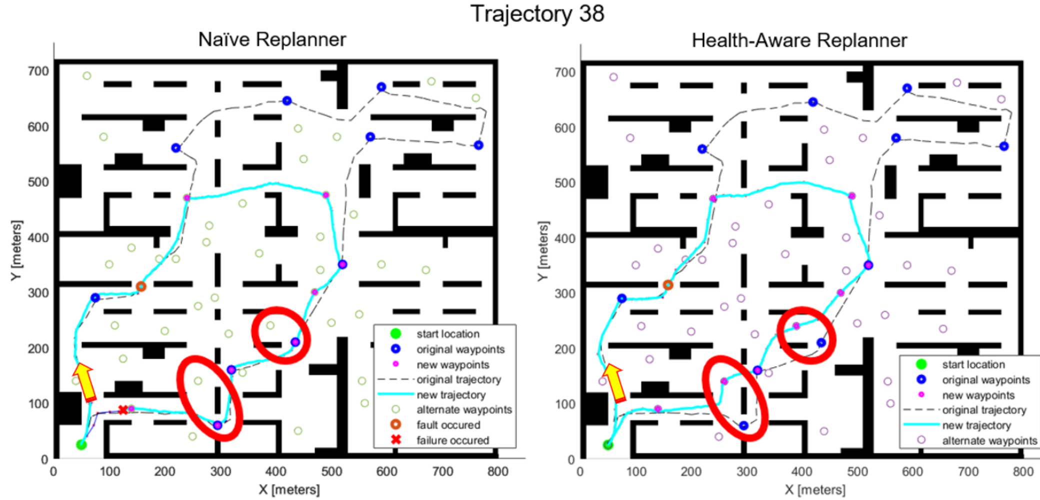

Figure 7.5: Left: Naive replanner. Right: Health-Aware replanner. Red circles show different waypoints selected. The UAV travels clockwise in this particular trajectory.

7.5 Results & Discussion

The results of the first experiment to demonstrate health awareness is presented first, followed by the GA performance analysis, and then the results of the replanning experiments. Figure 7.5 shows the comparison between health-aware replanning and replanning with a naive battery discharge model. The planner on the left receives a poor quality estimate from the naive remaining flight time calculation and comes up with a plan that is too optimistic and overestimates the RUL. Due to this, it attempts to reach two waypoints from the original trajectory, but ultimately fails before finishing the flight. It would have received a total reward of 24 points, as opposed to 16 points received by the Health-Aware replanner. However, the Health-Aware replanner had a more accurate RUL estimate that accounted for system degradation and estimated a shorter remaining flight time. It opted to not attempt to reach the the last waypoints (circled in red) and chose alternate waypoints instead due to the fact reaching the goal is an inviolable constraint.

7.5.1 GA Performance on Embedded Hardware

Figures 9.16 and 7.7 each show three plots for the GA algorithm using the data from Scenario D. Figure 9.16 is for the desktop test, and Figure 7.7 is the hardware test. The top left plot of each figure is 3-D and depicts running time (Z-axis), population size (Y-axis), and number of generations (X-axis), with the color bar tied to running time. The top right plot of each figure is also 3-D has the same X and Y axes, however the Z-axis and color bar represent the average reward. The Bottom middle plot of each figure is a 4-D plot, with the same X and Y axes as the other plots, but where the Z-axis represents running time, and the colorbar represents

reward. This plot allows us to see all four variables at once, and find the optimal manifold for which to choose our parameters from. The biggest different between the desktop and embedded hardware is in the running time, which is expected. Another thing we see is that increasing the population size is the dominating factor in the running time of the algorithm, which would also be expected. Answering the question above, *in all but one case the population size and number of generations where in the same range when controlling for execution time,* with an average population size of 500 and number of generations between 125 and 175.

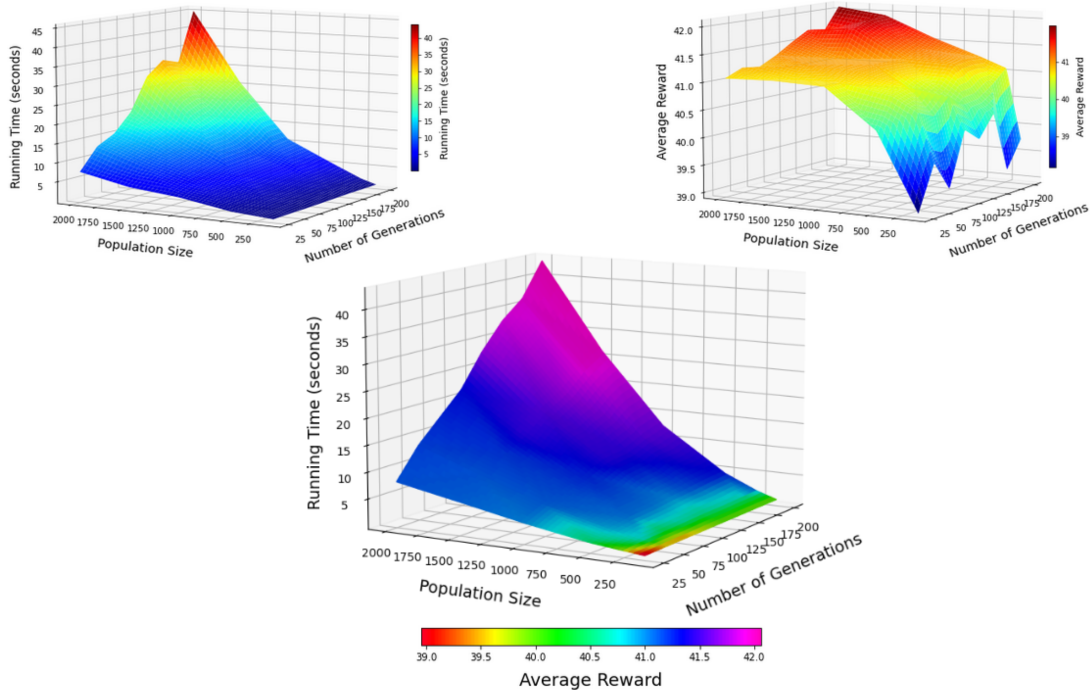

Figure 7.6: GA performance plots on desktop computer.

What we also see is that there is a wide range of population and generations that generate a reward value of 41 or greater (keeping in mind, the reward value is specific to the problem). When we restrict the running time to 5 seconds and a minimum reward value of 41, we can see that a population size of 500 is consistent across the different solutions. Other scenarios have similar results with consistent population sizes among the top solutions. An additional factor that affects the performance of the algorithm is the distance threshold with which to either consider or discard alternate waypoints. More waypoints are considered for larger distances, and the resulting population contains more solutions with trajectories that may be sub-optimal, which affects convergence time.

num_generations	population_size	avg_time	avg_reward
200	250	1.75	41.06
75	500	2.62	41.01
100	500	2.89	41.01
150	500	3.71	41.10
175	500	4.14	41.22
200	500	4.46	41.29

Table 7.3: Top GA performance results on desktop computer when controlling for time and reward such that running time is less than 5 seconds and reward is greater than or equal to 41.

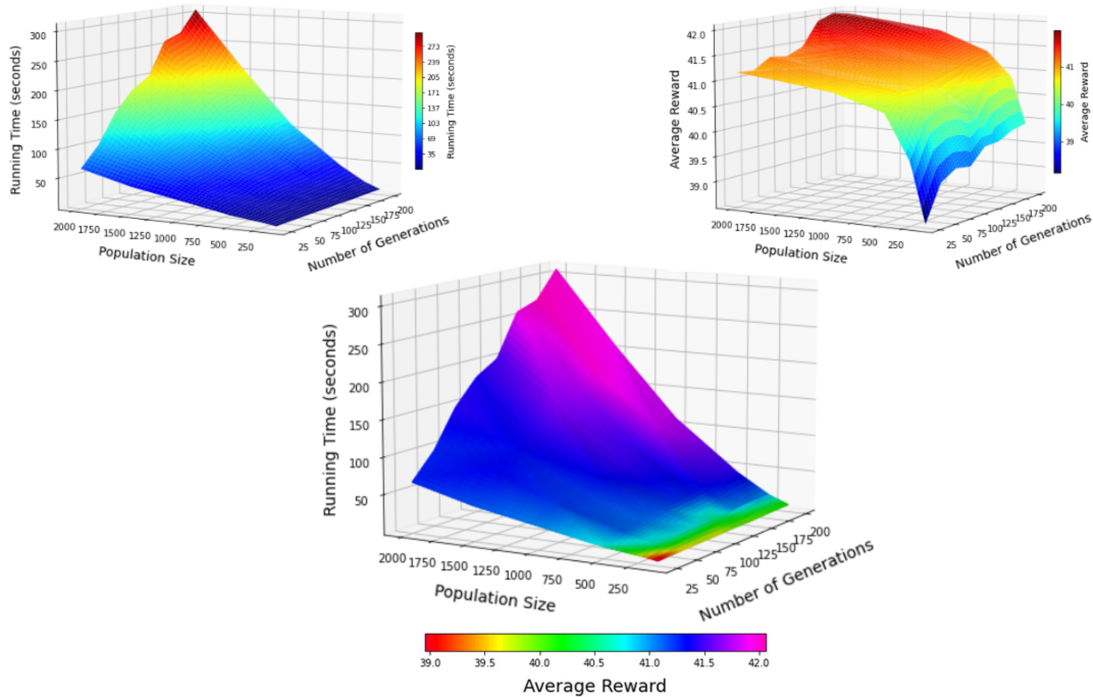

Figure 7.7: GA performance plots on the Unibap Qseven embedded hardware.

num_generations	population_size	avg_time	avg_reward
100	500	27.11	41.01
125	500	29.46	41.12
150	500	32.45	41.20
175	500	37.58	41.30
200	500	42.17	41.36

Table 7.4: Top GA performance results on embedded when controlling for reward only, such that the reward is greater than or equal to 41. The running time on the embedded hardware is prohibitive for online use.

7.5.2 Replanning Scenarios with Compressed RUL Model

Eight different trajectories were selected that are representative of the various types of trajectories in the database and fit certain criteria such as distance between 2300m and 3000m, not crossing paths, and landing with a remaining SOC of between 20% and 35%. The results of the health-aware framework for replanning using the 50% compressed model are shown in Table 7.5. A safe solution was generated for all but one scenario, where the UAV failed before landing with the new trajectory. When the original uncompressed model was used for this scenario, the UAV did not fail. This comparison is shown in Figure 7.9.

Scenario	UAV ID	Velocity	Q	R_b	[1] R_m	Wind	Orig. Wpts	Fault Time	Fault Type	Fault Mag.	Final Wpts	Max Dist.	[2] Times (a|b|c)	Ref
A	152	2.44m/s	20.11	.0091	.2799	2.64	12	127s	cell loss	$\frac{4}{6}Q$	9	75m	911s 711s 669s	Sec. 9.2.1
B	412	2.31m/s	18.76	.0118	.3104	1.96	11	392s	reduced speed	1.65m/s	7	100m	732s 544s 498s	Sec. 9.2.2
C	52	2.79m/s	18.06	.014	.3043	2.80	11	86s	cell loss	$\frac{5}{6}Q$	9	50m	825s 709s 699s	Sec. 9.2.3
D	372	2.84m/s	20.62	.0137	.2901	2.42	12	204s	cell loss	$\frac{5}{6}Q$	11	60m	744s 689s 653s	Sec. 7.4
E	132	2.22m/s	21.04	.0105	.3186	3.14	11	155s	reduced speed	1.51m/s	9	50m	1184s 802s 788s	Sec. 9.2.5
F	352	2.61m/s	19.91	.0093	.2857	1.77	6	249s	parasitic load	+.628Ah	4	80m	918s 722s 705s	Sec. 9.2.6
G	272	2.34m/s	19.12	.0038	.2982	1.89	15	62s	parasitic load	+.592Ah	12	100m	1106s 908s 837s	Sec 9.2.7
H[3]	452	2.39	21.16	.0082	.2752	3.27	18	193s	cell loss	$\frac{5}{6}Q$	17	60m	1018s 842s 842s	Sec. 9.2.8
H[4]	452	2.39	21.16	.0082	.2752	3.27	18	193s	cell loss	$\frac{5}{6}Q$	16	60m	1018s 842s 819s	Sec. 9.2.8

Table 7.5: Replanning results.
[1] Represents the average of 8 motors.
[2] Times column represents (a) the flight time required to finish the original trajectory after the fault occured; (b) the actual flight time of the UAV after the fault until failure; and (c) the flight time of the new trajectory that results in a safe landing at base.
[3] Plan did not result in a safe return to base.
[4] Same scenario but with the original uncompressed RUL estimation model resulting in a safe return to base.

7.5.3 Solution Variability & Quality

The GA is capable of finding multiple different solutions to the same scenario that all have the same reward value and comparable travel distances when controlling for population size and number of generations. This is shown below in Figure 7.8 for replanning scenario C (Section 9.2.3). We can see that the first trajectory differs from the second and third by only one waypoint, and the second and third differ by two waypoints, yet the trajectories are equivalent in terms of reward and distance.

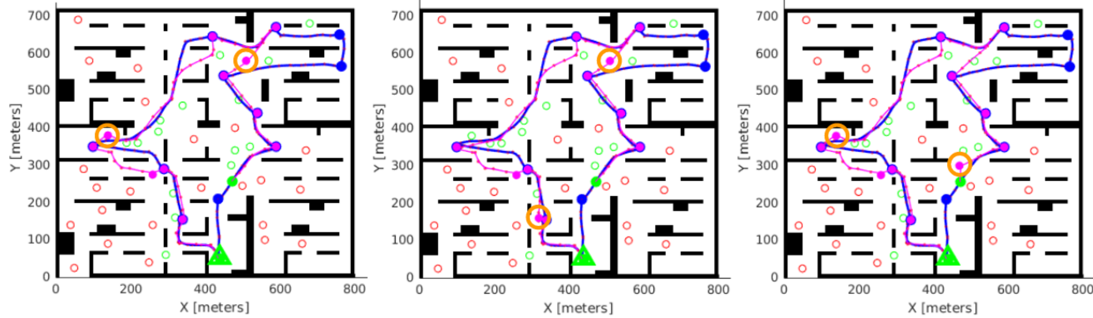

Figure 7.8: Similar solutions to the same replanning scenario with the same reward value and path distances. Waypoint differences are highlighted with orange circles.

The next comparison in Figure 7.9 (see Section 9.2.7) depicts the health-aware replanning solution on the left with a population size of 500 and 100 generations; the health-*unaware* replanning solution (i.e. without system-level SOH); and the health-aware replanning solution when running time is ignored on the right. As previous results demonstrated (Darrah et al. (2023), without system-level SOH information the replanning agent does not have accurate resource constraints (distance, time, power consumption) and as a result does not generate safe solutions. When comparing the left and right plots, we can see that both solutions discarded the same 3 waypoints from the original trajectory. However, when running time is ignored the population size is set to 2000 with 200 generations, and 2 additional waypoints are added to the trajectory. This results in a reward of 38 vs 36 for this particular example.

Figure 7.9: Left: Health-aware replanning solution when controlling for execution time (< 5 seconds → population size = 500, generations = 100) results in success. Middle: Health-*unaware* replanning solution results in failure. Right: Health-aware replanning solution when running time is not a factor (population size = 2000, generations = 200), also resulting in success but with 2 additional alternate waypoints.

All but one scenario resulted in a safe plan that allowed the UAV to return to base without failure. In this case (Scenario 9.2.8), the estimation model overestimated the remaining flight time and as a result the replanning agent returned a solution that required more flight time than available. This is shown below in Figure 7.10.

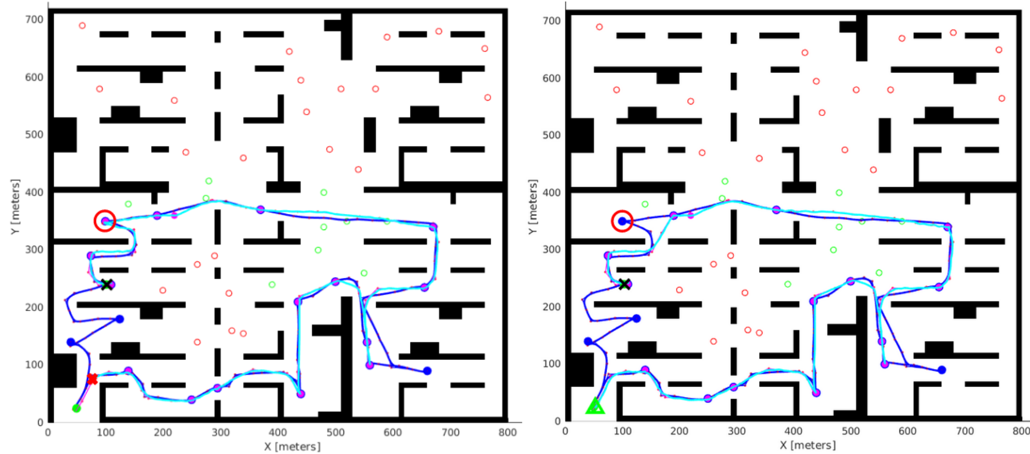

Figure 7.10: Left: Using the compressed model resulted in an overestimation of remaining flight time and a failure prior to returning to base. Right: The same scenario but with the original uncompressed model. The remaining flight time was not overestimated, and the UAV safely landed. The discrepant waypoint is highlighted in red.

The different scenarios presented highlight the efficacy of the health-aware replanning framework, which is comprised of system-level RUL estimation and a replanning agent. The results demonstrate that replanning with system-level RUL estimations result in solutions that do not violate hard constraints, which is not always the case when replanning without. Furthermore, these results demonstrate that a compressed model can deliver adequate estimates to the replanning agent most of the time, but not always. This should be further explored. While promising, safety critical applications require a much higher level of confidence than 7 out of 8 trials.

7.5.4 Summary

In this chapter, a novel health-aware framework for replanning UAV operations has been presented. Such an approach to replanning that accounts for usage-based degradation in conjunction with abrupt faults has not been presented in the literature before. Health-awareness was demonstrated in a comparative analysis between replanning tasks with and without system-level RUL estimates. Additionally, a uniquely modified GA specific to replanning trajectories is detailed. The GA is tested on embedded hardware and a hybrid on-board off-board methodology is shown to be more time efficient than on-board only. Multiple replanning scenarios are discussed that used the compressed RUL model on-board with the GA off-board with seven out

of 8 scenarios resulting in successful plans. The one unsuccessful scenario was repeated with the original uncompressed model and the replanning agent generated a safe and successful plan. In conclusion, the health-aware replanning framework for UAV operations is a clear improvement over existing replanning approaches. Using a compressed model for RUL estimation in conjunction with a task-specific application such as replanning shows great promise, but further research is warranted.

CHAPTER 8

Conclusions & Future Work

The integration of system-level prognostics with mission replanning is crucial for the resilience and adaptability of UAV missions. UAVs face significant challenges when tasked with dynamic replanning in the face of abrupt faults, especially when they also have to cope with nominal wear-and-tear degradation, which may be further compounded as a result of the fault. Central to addressing this challenge is the accurate estimation of system-level RUL online, after the occurrence of the fault, which enables the generation of reliable remaining flight time estimates, a critical input for the replanning process. The deep learning framework for system-level prognostics is a contribution summarized in Section 5.4. Model-based approaches, while robust for specific component-level predictions, fall short when tasked with capturing these multifaceted interactions and the resultant systemic degradation patterns. This complexity necessitates the use of deep learning methods which are adept at modeling high-dimensional and non-linear data. In essence, by adopting a holistic view of system health, it becomes feasible to anticipate and counteract issues that might not be immediately evident when analyzing components in isolation. System-level prognostics represents a paradigm shift from traditional component-level prognostics, in which components degrades at different rates, and their combined interactions produce emergent behaviors that are inherently non-linear and complex. In addition, the interactions that occur among the various components as well as the cascading effects of faults in the system must be combined with system degradation to estimate RUL, i.e., in this case, the remaining safe flight time.

Building on this comprehensive prognostics framework, model reduction becomes essential to make deep learning models operationally viable for resource constrained applications. The developed RUL estimation model is compressed using a unique technique informed by neural activation patterns and cell state preservation, which is summarized in Section 6.3.3. Model reduction is vital for deploying models to resource constrained computing platforms like those found on UAVs. An innovative avenue explored in this research is the incorporation of techniques from the realm of explainable AI into model reduction for LSTMs. One foundational idea from XAI is that by understanding how a neural network generates output – through mechanisms such as examining neuron activations in hidden layers – we can gain insights into which aspects of the model are most critical to its performance. In the context of model reduction, such insights are invaluable. If specific neurons consistently show low activations across various input samples or don't contribute

significantly to the model's final output during the late stages of life of the UAV, they may be deemed less crucial. A key point here is evaluating these activations with input data *during the late stages of life*, as this is application specific to RUL estimation. Guided by these observations, the developed pruning process is capable of determining which neurons and weights can be safely removed without significantly compromising the model's predictive accuracy. This method ensures that the pruning decisions are informed, systematic, and less likely to inadvertently impair the model's core functionality. In addition, LSTM networks are different from other architectures in that they have a built-in memory mechanism known as the *cell state*. The cell state has separate weight matrices from the rest of the weights in the cell, and by preserving these weights during compression, the model is able to retain more information and have lower error rates compared to other approaches. The tradeoff is that the maximum sparsity the model can attain is 67.5%.

Building on the compressed prognostics model, the next critical phase in enhancing UAV resilience lies in effective replanning to complete as much of the original system goals as possible in a safe manner. The replanning process is further complicated by the need for time-constrained, i.e., near real time, on-the-fly computations. The goal during replanning is twofold: to maximize rewards by flying through attainable waypoints, with a preference for waypoints on the initial trajectory, and to ensure the UAV does not exceed its remaining flight time. However, inaccuracies in flight time estimates can lead the UAV to mistakenly believe it possesses more flight time, culminating in mission failure before returning to base. This research introduces a novel genetic algorithm tailored to the online constrained path planning with rewards problem. Key innovations include a "fast start" population initialization technique, where candidate solutions are seeded with parts of the original trajectory, and waypoints beyond a threshold distance from the original path are not considered. The algorithm also exhibits adaptive behaviors in its crossover, mutation, and elitism functions, adapting responsively based on algorithmic performance metrics such as population diversity, population fitness, and improvement iterations. A population restart mechanism is also incorporated to rejuvenate the search when required. Such an algorithm was shown to generate optimal solutions in under 5 seconds for multiple different scenarios. While the GA algorithm is uniquely modified, the key contribution is in the demonstration of *health-awareness*, summarized in Section 7.5.4. It is shown that replanning without system-level RUL estimates results in plans that violate resource constraints, something that cannot happen for complex systems operating in critical environments. With this information, the replanning agent is capable of generating safe solutions. Furthermore, this was accomplished with the compressed RUL model in 7 out of 8 trials.

8.1 Limitations & Future Work

In synthesizing these innovations, this research presents a comprehensive health-aware framework designed for the online replanning of UAV missions. This integrated approach accounts for the complexities of UAV health monitoring, accurate flight time estimation, and the nuanced challenges of real-time path replanning. The proposed methodologies not only contribute to the immediate context of UAV operations but also hold promise for broader applications in dynamic systems requiring responsive and accurate decision-making under constraints. However, there is room for further research.

Simulated data. As will all research studies, there are limitations with this work that should be highlighted to help guide future research activities in this area. The first limitation is with the data used to train the prognostics model - it was all simulated. While useful to develop the methodology, the approach needs to be validated with real-world data, which is currently the biggest challenge that led us to use simulation data to begin with. Data is currently being collected on octorotor UAVs by collaborative researchers at MIT and NASA Langley, and as more and more of this data is collected this approach should incorporate real data into the datasets.

Model compression. Another limitation is that the model compression method was developed for LSTM networks performing time-series regression tasks was tested with only one dataset. While it was intentional to develop an application-specific approach for RUL estimation, there are other time-series regression datasets that could be used to assess how well the approach generalizes to other tasks. Furthermore, advanced optimization techniques were not employed such as conversion to flat buffer or ONNX format - formats specifically designed to take full advantage of compressed models. Such conversion requires a deeper expertise into use and manipulation of binary data, which is outside the scope of this thesis. While the current study employed compression techniques informed by neural activations, our attempts to leverage quantization, specifically converting models to INT8, yielded suboptimal results. One compelling hypothesis warranting further exploration is the inherent suitability of INT8 quantization for classification tasks, which operate in a discrete space, as opposed to regression tasks that demand precision in a continuous space. Could it be that quantization methods like INT8 are inherently ill-suited for the nuanced demands of regression? In addition, knowledge distillation techniques could be explored through the lense of LSTM networks and model compression, which were not studied in this thesis.

Hardware-In-The-Loop testing. A third limitation is that performing hardware-in-the-loop testing was not possible due to a licensing mismatch, NASA IT policies, and time. The simulation was developed with certain packages in MATLAB that could not be accessed with the installation of MATLAB on the NASA computer that was available, and NASA IT policy prohibits personal computers from connecting to their network, where the specialized hardware could only be accessed. There was only a limited amount of time to finish these experiments, which prevented us from addressing this issue. Future work could address this by including hardware in the loop testing, and eventually, testing on a real UAV.

On the front of replanning, there lies potential in the notion of proactive replanning. Instead of reactive adjustments post-failure, could we devise a tandem agent, constantly keeping a roster of viable alternate plans? This agent, equipped with contingency blueprints for varying levels of potential failure severities, would usher in a paradigm of anticipatory path optimization. Such a dual-agent system not only augments robustness but also imbues UAV missions with a resilience that stems from foresight. As we delve deeper into these research domains, our primary mission is clear: to construct intelligent systems that not only adapt to challenges but anticipate them, and operate as efficiently as possible, continuing to push the frontier of our technological capabilities and enhance the operation, reliability, and safety, of the complex systems we so heavily rely on in our everyday lives.

CHAPTER 9

Appendix

9.1 UAV Telemetry Data

This sections lists all of the features of the telemetry data, which is generated at 1Hz for all UAVs and flights. This is shown below in Table 9.1. Additionally, each feature is plotted from a nominal flight in Figure 9.1, which shows the range of of values these features take for a typical flight.

Features				
flight_id	flight_num	uav_id	trajectory_id	stop_code
acceleration	amp_hours	battery_hat_r	battery_hat_r_var	battery_hat_v
battery_hat_z	battery_hat_z_var	battery_true_i	battery_true_r	battery_true_v
battery_true_z	euclidean_pos_err	flight_load	m1_current	m1_rpm
m1_torque	m1_vref	m2_current	m2_rpm	m2_torque
m2_vref	m3_current	m3_rpm	m3_torque	m3_vref
m4_current	m4_rpm	m4_torque	m4_vref	m5_current
m5_rpm	m5_torque	m5_vref	m6_current	m6_rpm
m6_torque	m6_vref	m7_current	m7_rpm	m7_torque
m7_vref	m8_current	m8_rpm	m8_torque	m8_vref
p_rot	pitch	q_rot	r_rot	roll
var_pos_err	velocity	wind_const_x	wind_const_y	wind_const_z
wind_gust_x	wind_gust_y	wind_gust_z	x_accel	x_ctrl_err
x_pos_err	x_pos_gps	x_pos_true	x_vel_gps	x_vel_true
y_accel	y_ctrl_err	y_pos_err	y_pos_gps	y_pos_true
y_vel_gps	y_vel_true	yaw	z_accel	z_pos_gps
z_pos_true	z_vel_gps	z_vel_true		

Table 9.1: Telemetry data sampled at 1Hz.

Figure 9.1: Flight Data Feature Ranges.

9.2 Additional Replanning Scenarios

This section contains the description, plots, and data table for each of the scenarios listed in Table 7.5.

9.2.1 Scenario A.

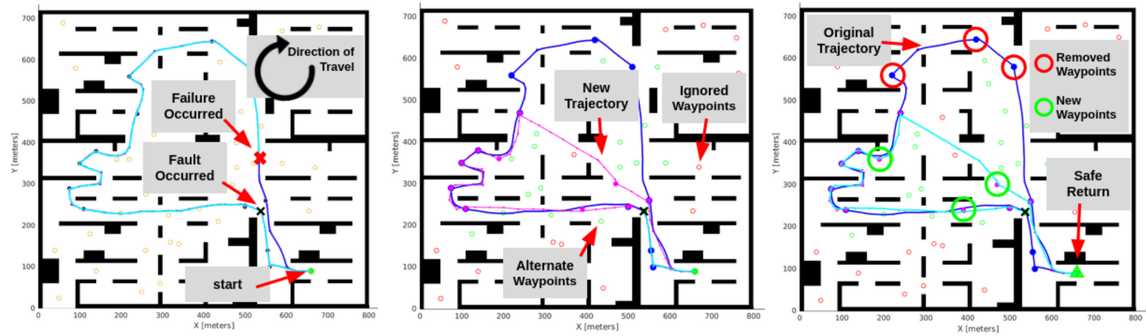

Figure 9.2: Replanning scenario A.

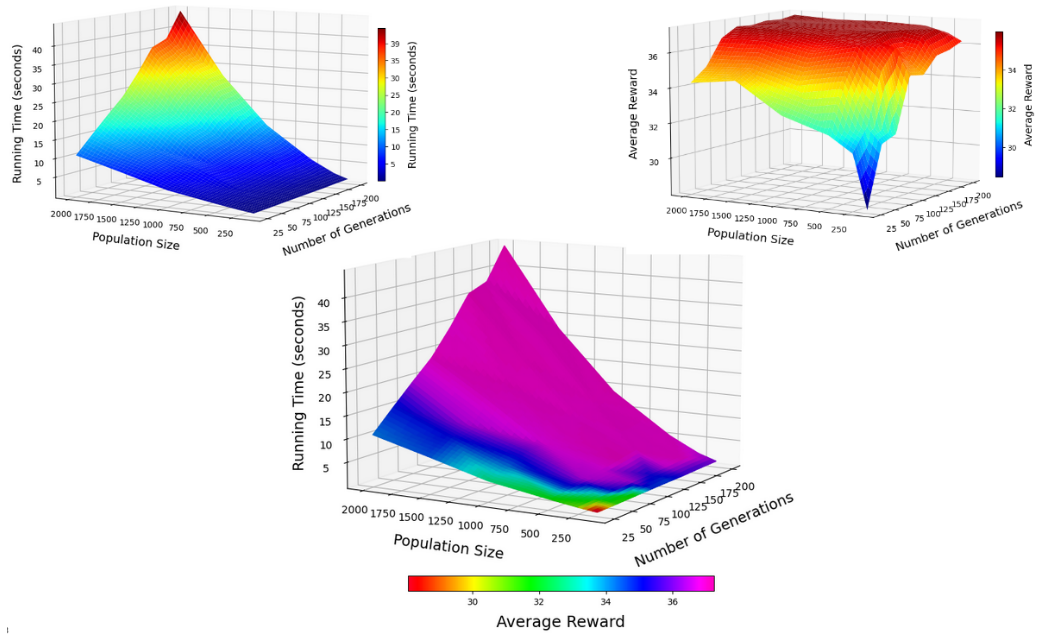

Figure 9.3: GA results for scenario A.

num_generations	population_size	avg_time	avg_reward
175	250	1.79	37.00
125	500	3.49	37.08
150	500	4.23	37.00
175	500	4.40	37.00
200	500	5.20	37.00

Table 9.2: Top fastest results for solutions with 37 or greater reward for scenario A.

9.2.2 Scenario B.

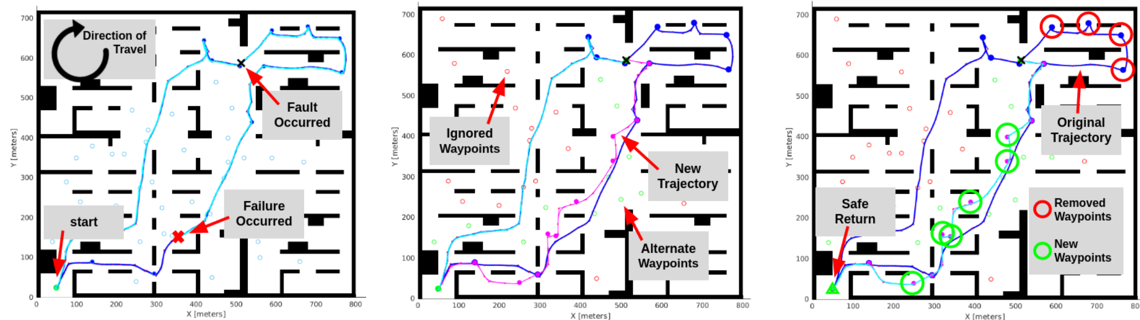

Figure 9.4: Replanning scenario B.

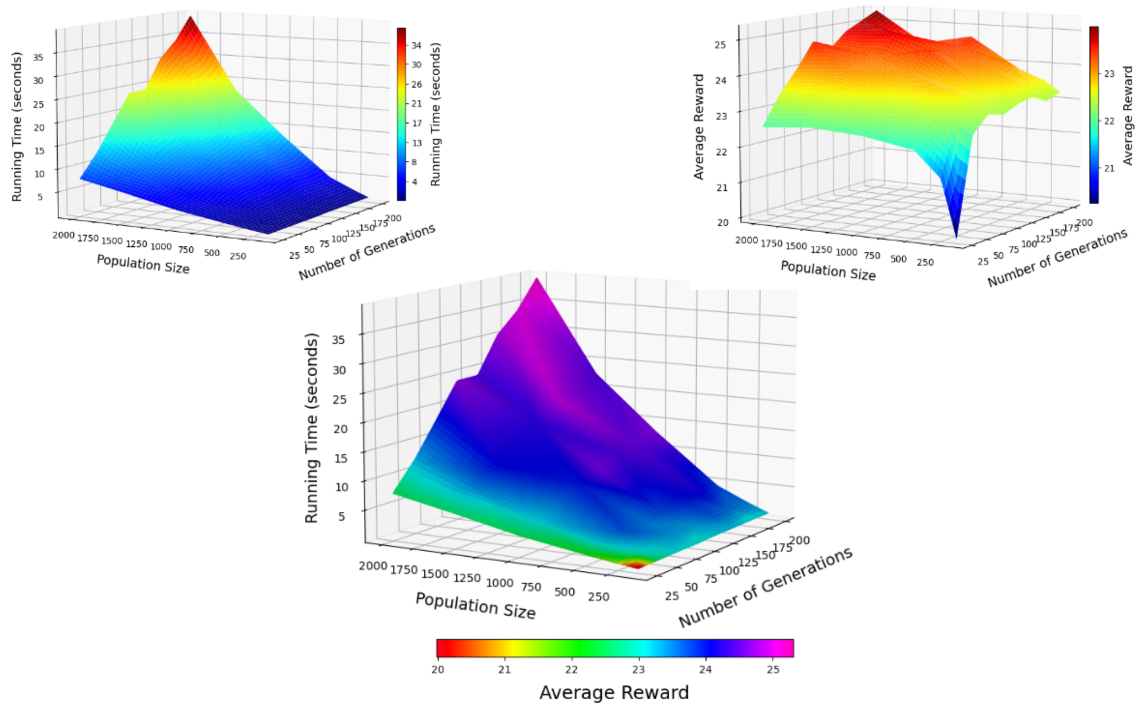

Figure 9.5: GA results for scenario B.

num_generations	population_size	avg_time	avg_reward
175	500	3.90	24.16
150	500	4.04	24.02
125	1000	8.78	24.58
150	1000	10.55	24.40
175	1000	10.80	24.50

Table 9.3: Top fastest results for solutions with 24 or greater reward for scenario B.

9.2.3 Scenario C.

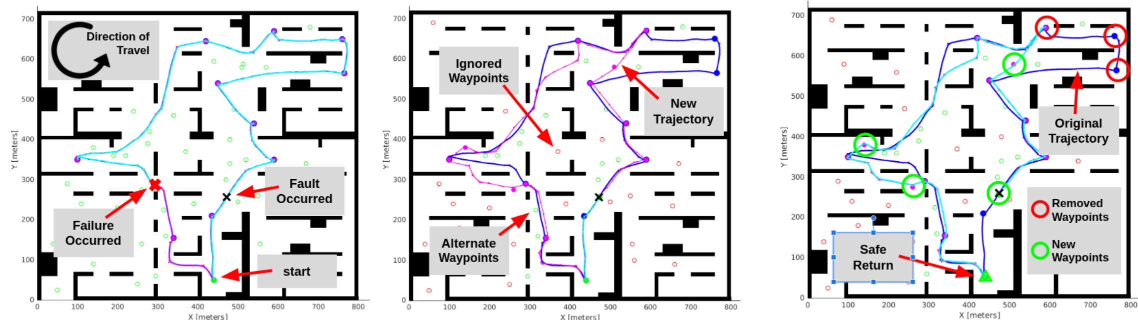

Figure 9.6: Replanning scenario C.

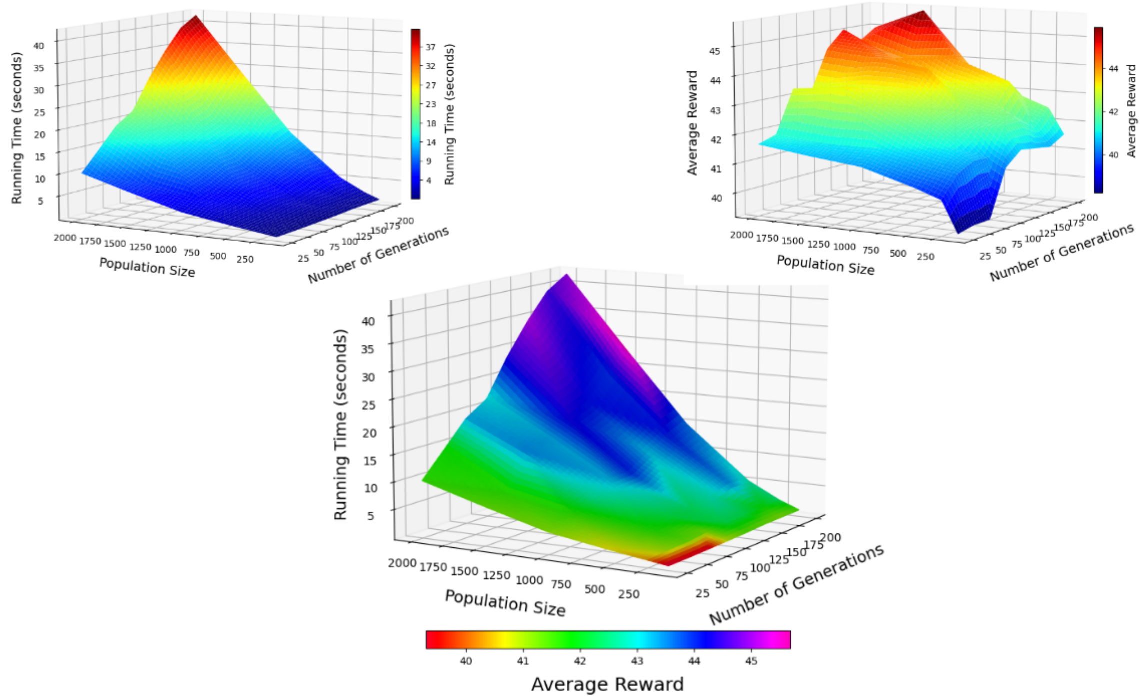

Figure 9.7: GA results for scenario C.

num_generations	population_size	avg_time	avg_reward
125	250	1.64	42.26
200	250	2.07	42.46
100	500	3.05	42.16
125	500	3.12	42.32
175	500	5.30	43.48
100	1000	8.65	43.30

Table 9.4: Top fastest results for solutions with 42 or greater reward for scenario C.

9.2.4 Scenario D.

See Section 7.4

9.2.5 Scenario E.

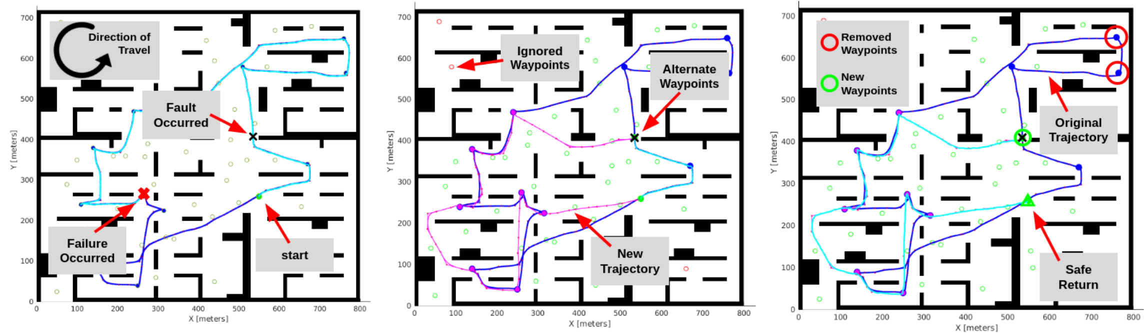

Figure 9.8: Replanning scenario E.

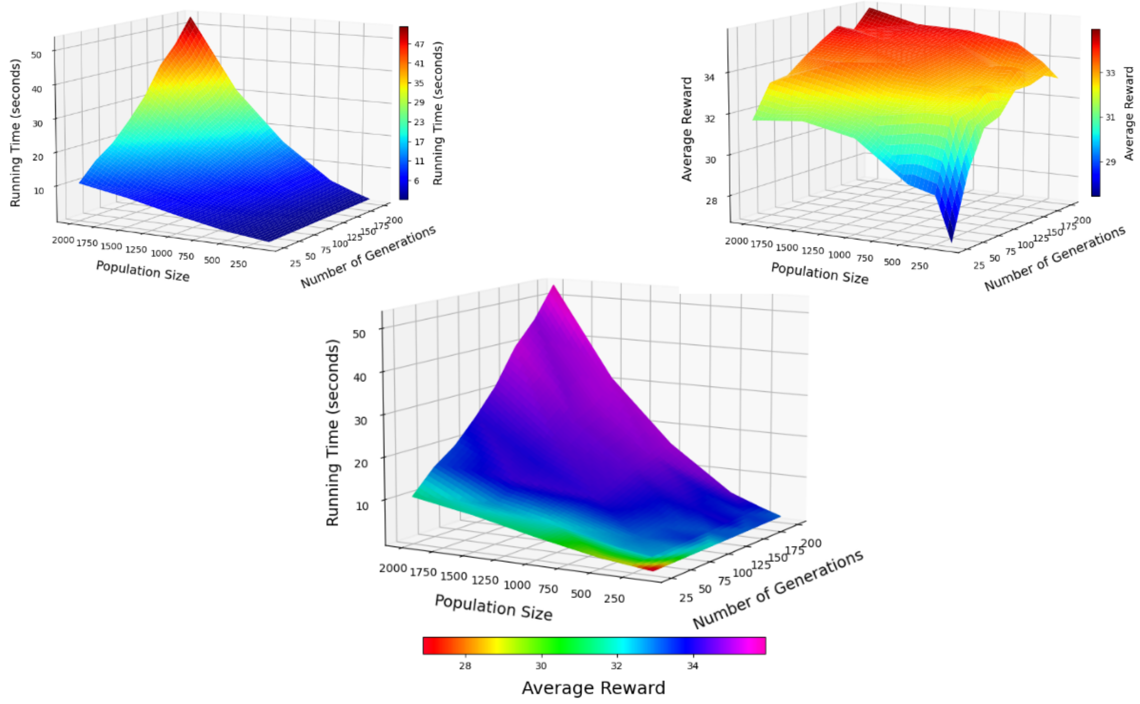

Figure 9.9: GA results for scenario E.

num_generations	population_size	avg_time	avg_reward
125	100	0.51	33.10
150	100	0.58	33.12
175	100	0.64	33.40
200	500	5.04	34.64
100	1000	8.10	34.22
200	1000	15.65	35.10

Table 9.5: Top fastest results for solutions with 42 or greater reward for scenario E.

9.2.6 Scenario F.

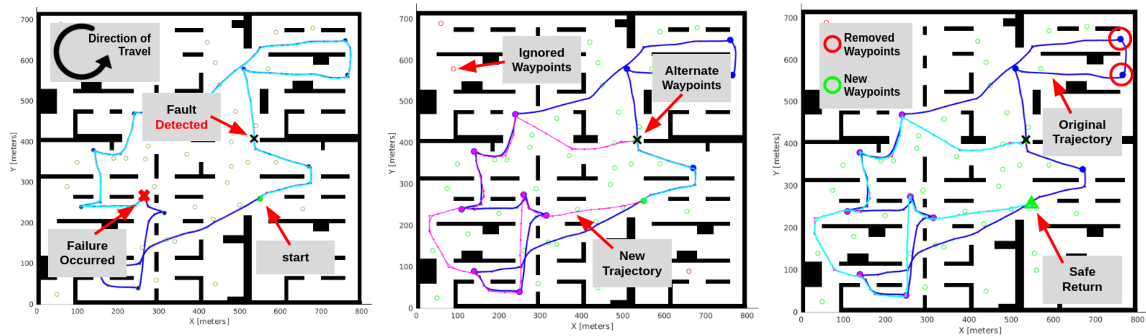

Figure 9.10: Replanning scenario F.

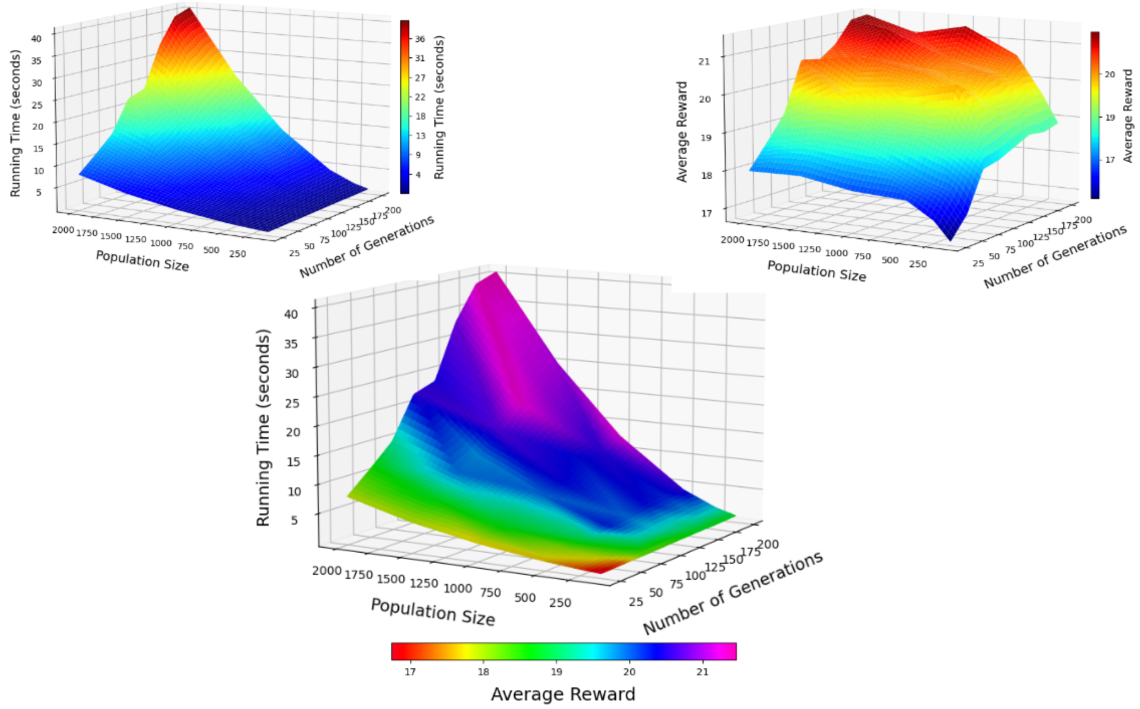

Figure 9.11: GA results for scenario F.

num_generations	population_size	avg_time	avg_reward
100	500	2.50	20.28
150	500	3.38	20.34
175	500	3.67	20.24
200	500	4.10	20.62
200	1000	12.61	21.32

Table 9.6: Top fastest results for solutions with 20 or greater reward for scenario F.

9.2.7 Scenario G.

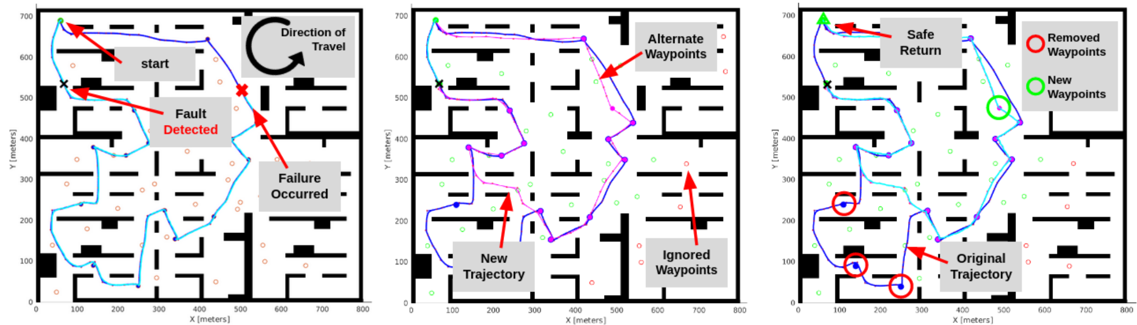

Figure 9.12: Replanning scenario G.

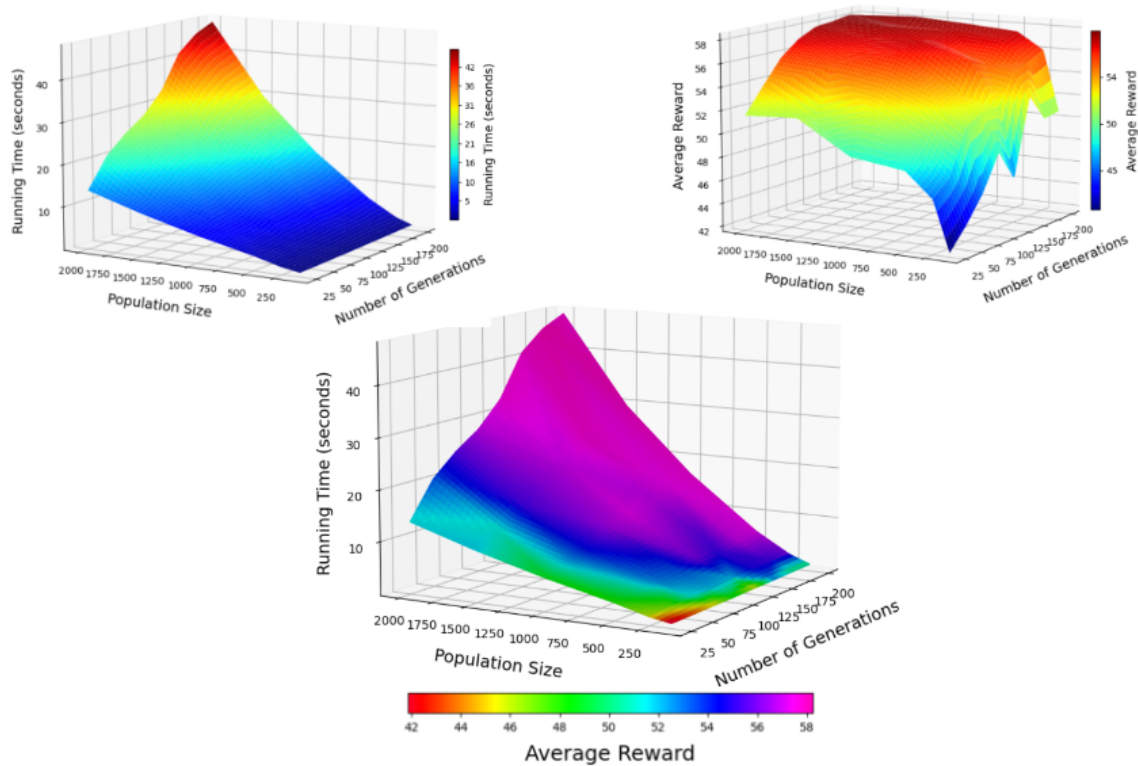

Figure 9.13: GA results for scenario G.

num_generations	population_size	avg_time	avg_reward
175	250	2.23	55.24
200	250	2.39	55.96
125	500	4.54	55.94
150	500	4.56	55.20
175	500	5.37	56.98
200	500	6.34	57.44

Table 9.7: Top fastest results for solutions with 55 or greater reward for scenario G.

9.2.8 Scenario H.

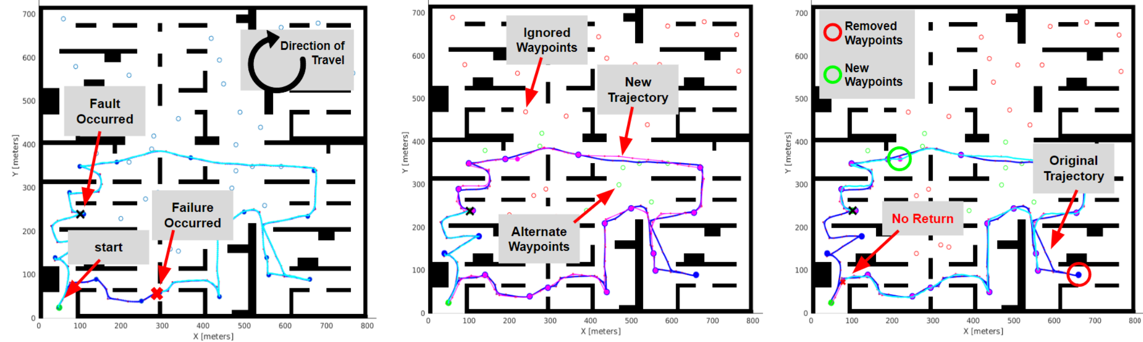

Figure 9.14: Replanning scenario H.

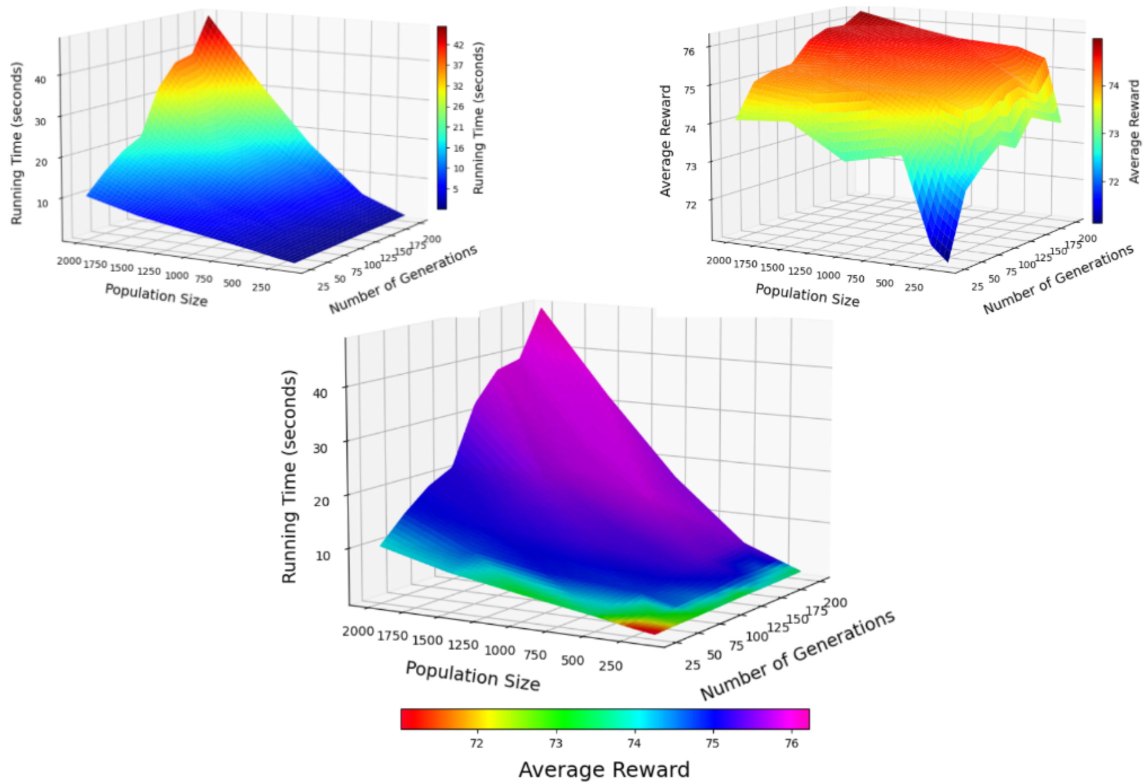

Figure 9.15: GA results for scenario H.

num_generations	population_size	avg_time	avg_reward
25	100	0.34	71.02
50	100	0.46	72.68
75	100	0.49	73.16
100	100	0.59	73.56
125	100	0.71	73.32

Table 9.8: Top fastest results for solutions for scenario H. The actual maximum reward attainable while remaining within resource constraints is 71, however, all but one solution exceeds this amount and violates resource constraints.

9.3 Experimental Demonstration of GA Performance

To demonstrate the performance of the algorithm, a test case was implemented using a graph with the following parameters: 30 waypoints with distance values between 5 and 15 units; reward values of either 0, 1, or 5 units; a max distance threshold of 70 units; and randomly selected start and goal waypoints. To approximate the distribution of solutions that meet the maximum distance constraint, a population of 1M candidates was generated using Algorithm 7. The top half of Figure 9.16 shows the actual distribution in blue, the normalized distribution in orange, and the maximum reward value out of the 1M candidates in red, which is 50 units. Then, the GA algorithm (Algorithm 14) was executed with the same graph a total of 50 times, forming the reward distribution shown in green in the top plot of Figure 9.16. The average reward found was 47 units, and the highest reward found was 51 units, higher than that found in a sample space of 1M candidates. The performance distributions are shown in the second row, depicting the distribution of generations until the algorithm halts (bottom left), the average execution time in the bottom middle (in this experiment, the CPU is an AMD Ryzen Threadripper 3960X 24-Core Processor 4.8GHz), and the number of generations before a top 2% solution is found (bottom right).

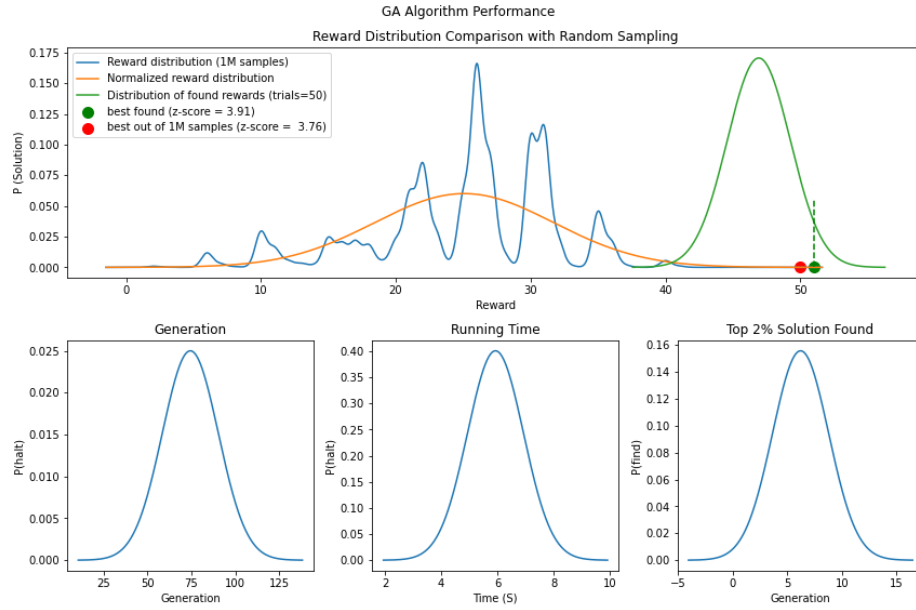

Figure 9.16: GA Performance.

References

Abeywickrama, H. V., Jayawickrama, B. A., He, Y., and Dutkiewicz, E. (2018). Comprehensive energy consumption model for unmanned aerial vehicles, based on empirical studies of battery performance. *IEEE Access*, 6:58383–58394.

Adadi, A. and Berrada, M. (2018). Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*.

Adebayo, J., Gilmer, J., Muelly, M., Goodfellow, I., Hardt, M., and Kim, B. (2018). Sanity checks for saliency maps. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31.

Advanced Precision Composites (Accessed: 2019). 8x4 propeller performance dataset. *UIUC Propeller Dataset*.

Aggarwal, S. and Kumar, N. (2020). Path planning techniques for unmanned aerial vehicles: A review, solutions, and challenges. *Computer Communications*, 149(July 2019):270–299.

Ahmed, I., Quinones-Grueiro, M., and Biswas, G. (2023). Adaptive fault-tolerant control of octo-rotor uav under motor faults in adverse wind conditions.

Ahmed, S., Mohamed, A., Harras, K., Kholief, M., and Mesbah, S. (2016). Energy efficient path planning techniques for uav-based systems with space discretization. In *2016 IEEE Wireless Communications and Networking Conference*.

Almas, B., Bicarregui, J., Blatecky, A., Hill, S., Lannom, L., Pennington, R., Stotzka, R., Treloar, A., Wilkinson, R., Wittenburg, P., and Yunqiang, Z. (2016). Data management trends, principles and components what needs to be done next? *Research Data Alliance*.

An, D., Kim, N. H., and Choi, J.-H. (2015). Practical options for selecting data-driven or physics-based prognostics algorithms with reviews. *Reliability Engineering & System Safety*, 133:223–236.

Ancona, M., Ceolini, E., Öztireli, C., and Gross, M. (2019). *Gradient-Based Attribution Methods*, pages 169–191.

Arias Chao, M., Kulkarni, C., Goebel, K., and Fink, O. (2022). Fusing physics-based and deep learning models for prognostics. *Reliability Engineering & System Safety*, 217.

Badihi, H., Zhang, Y., Jiang, B., Pillay, P., and Rakheja, S. (2022). A comprehensive review on signal-based and model-based condition monitoring of wind turbines: Fault diagnosis and lifetime prognosis. *Proceedings of the IEEE*, 110.

Bäuerle, A., Jönsson, D., and Ropinski, T. (2022). Neural activation patterns (naps): Visual explainability of learned concepts. *CoRR*, abs/2206.10611.

Bazurto, A. J., Quispe, E. C., and Mendoza, R. C. (2016). Causes and failures classification of industrial electric motor. In *2016 IEEE ANDESCON*, pages 1–4.

Bellman, R. (1962). Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9.

Benders, S., Schopferer, S., and Nawrath, A. (2020). In-flight Kinematic Model Parameter Estimation and Adaptive Path Planning for Unmanned Aircraft. In *AIAA Scitech 2020 Forum*, page 0.135.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In Shawe-Taylor, J., Zemel, R., Bartlett, P., Pereira, F., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc.

Binder, A., Montavon, G., Bach, S., Müller, K., and Samek, W. (2016). Layer-wise relevance propagation for neural networks with local renormalization layers. *CoRR*, abs/1604.00825.

Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., and Guttag, J. (2020). What is the state of neural network pruning? In *Proceedings of Machine Learning and Systems*.

Bonet, B. and Geffner, H. (2011). Planning under partial observability by classical replanning: Theory and experiments. In *IJCAI*.

Celaya, J. R., Kulkarni, C., Saha, S., Biswas, G., and Goebel, K. (2012). Accelerated aging in electrolytic capacitors for prognostics. In *2012 Proceedings Annual Reliability and Maintainability Symposium*, pages 1–6. IEEE.

Chahar, V., Katoch, S., and Chauhan, S. (2021). A review on genetic algorithm: Past, present, and future. *Multimedia Tools and Applications*, 80.

Chakrabarty, A. and Langelaan, J. (2013). Uav flight path planning in time varying complex wind-fields. In *American Control Conference*.

Chakrabarty, A., Stepanyan, V., Krishnakumar, K. S., and Ippolito, C. A. (2019). *Real-Time Path Planning for Multi-copters flying in UTM -TCL4*.

Chao, M. A., Kulkarni, C., Goebel, K., and Fink, O. (2021). Aircraft engine run-to-failure dataset under real flight conditions for prognostics and diagnostics. *Data*.

Chatterjee, A. and Reza, H. (2019). Path planning algorithm to enable low altitude delivery drones at the city scale. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 750–753. IEEE.

Chen, L., Chen, Y., Xi, J., and Le, X. (2021). Knowledge from the original network: restore a better pruned network with knowledge distillation. *Complex & Intelligent Systems*, 8.

Chen, X., Shen, Z., He, Z., Sun, C., and Liu, Z. (2013). Remaining life prognostics of rolling bearing based on relative features and multivariable support vector machine. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 227.

Chen, Y., Mou, L., Xu, Y., Li, G., and Jin, Z. (2016). Compressing neural language models by sparse word representations. *CoRR*.

Chicco, D., Warrens, M. J., and Jurman, G. (2021). The coefficient of determination r-squared is more informative than smape, mae, mape, MSE and RMSE in regression analysis evaluation. *PeerJ Comput. Sci.*, 7.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078.

Coble, J. B. (2010). Merging data sources to predict remaining useful life – an automated method to identify prognostic parameters. In *PhD Thesis, University of Tennessee*.

Cui, Z., Ke, R., and Wang, Y. (2018). Deep bidirectional and unidirectional LSTM recurrent neural network for network-wide traffic speed prediction. *CoRR*.

Da Silva Arantes, J., Da Silva Arantes, M., Toledo, C. F. M., and Williams, B. C. (2015). A multi-population genetic algorithm for uav path re-planning under critical situation. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*.

Dai, X., Yin, H., and Jha, N. K. (2018). Grow and prune compact, fast, and accurate lstms. *IEEE Transactions on Computers*, 69.

Daigle, M. and Kulkarni, C. (2013). Electrochemistry-based battery modeling for prognostics. In *Annual Conference of the Prognostics and Health Management Society 2013*, pages 249–261.

Darrah, T., Biswas, G., Frank, J., Quinones-Grueiro, M., and Teubert, C. (2022a). A data-centric approach to the study of system-level prognostics for cyber physical systems: application to safe uav operations. *Journal of Surveillance, Security and Safety*, 3(2).

Darrah, T., Kulkarni, C. S., and Biswas, G. (2020). *The Effects of Component Degradation on System-Level Prognostics for the Electric Powertrain System of UAVs*.

Darrah, T., Lovberg, A., Frank, J., Biswas, G., and Quinones-Gruiero, M. (2022b). Developing deep learning models for system remaining useful life predictions: Application to aircraft engines. In *Annual Conference of the PHM Society*.

Darrah, T., Quiñones-Grueiro, M., Biswas, G., and Kulkarni, C. (2021). Prognostics based decision making for safe and optimal uav operations. In *AIAA Scitech 2021 Forum*.

Darrah, T., Quiñones-Grueiro, M., Frank, J., and Biswas, G. (2023). A health-aware framework for online replanning of unmanned aerial vehicle missions under faulty conditions. In *International Conference on Automated Planning and Scheduling, InTEX Workshop*.

de Moura Souza, G. and Toledo, C. F. M. (2020). Genetic algorithm applied in uav's path planning. In *2020 IEEE Congress on Evolutionary Computation (CEC)*.

Ding, Y. (2021). The impact of learning rate decay and periodical learning rate restart on artificial neural network. In *Proceedings of the 2021 2nd International Conference on Artificial Intelligence in Electronics Engineering*.

Dong, D., Li, X.-Y., and Sun, F.-Q. (2017). Life prediction of jet engines based on lstm-recurrent neural networks. In *2017 Prognostics and System Health Management Conference (PHM-Harbin)*, pages 1–6.

Farin, G. (2014). *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Elsevier.

Feurer, M. and Hutter, F. (2019). *Automated Machine Learning, Chapter 1*. Springer.

Fischetti, M., Salazar González, J. J., and Toth, P. (1998). Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing*, 10:133–148.

Frankle, J. and Carbin, M. (2018). The lottery ticket hypothesis: Training pruned neural networks. *CoRR*.

Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. (2021). A survey of quantization methods for efficient neural network inference. *CoRR*.

Goebel, K., Celaya, J., Sankararaman, S., Roychoudhury, I., Daigle, M., and Saxena, A. (2017). *Prognostics: The Science of Making Predictions*.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Goodfellow, I. J., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA, USA. http://www.deeplearningbook.org.

Gorospe, G. E., Kulkarni, C. S., Hogge, E. F., Hsu, A., and Ownby, N. B. (2017). A study of the degradation of electronic speed controllers for brushless dc motors.

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18.

Gunawan, A., Lau, H. C., and Vansteenwegen, P. (2016). Orienteering problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, 255(2):315–332.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182.

Gyenes, Zoltán and Bölöni, Ladislau and Szádeczky-Kardoss, Emese Gincsainé (2023). Can genetic algorithms be used for real-time obstacle avoidance for lidar-equipped mobile robots? *Sensors*.

Han, S., Mao, H., and Dally, W. J. (2016). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding.

Hassibi, B., Stork, D., and Wolff, G. (1993). Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*.

Heng, A., Zhang, S., Tan, A. C., and Mathew, J. (2009). Rotating machinery prognostics: State of the art, challenges and opportunities. *Mechanical systems and signal processing*, 23(3):724–739.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hu, X., Zhang, K., Liu, K., Lin, X., Dey, S., and Onori, S. (2020). Advanced fault diagnosis for lithium-ion battery systems: A review of fault mechanisms, fault features, and diagnosis procedures. *IEEE Industrial Electronics Magazine*, 14.

Hua, Y., Zhao, Z., Li, R., Chen, X., Liu, Z., and Zhang, H. (2019). Deep learning with long short-term memory for time series prediction. *IEEE Communications Magazine*, 57.

Huang, C.-G., Huang, H.-Z., and Li, Y.-F. (2019). A bidirectional lstm prognostics method under multiple operational conditions. *IEEE Transactions on Industrial Electronics*, 66(11).

Jackey, R., Plett, G., and Klein, M. (2009). Parameterization of a battery simulation model using numerical optimization methods. *SAE International*.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. G., Adam, H., and Kalenichenko, D. (2017). Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877.

Kavraki, L. E., Svestka, P., Latombe, J. ., and Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12.

Khaji, E. and Mohammadi, A. S. (2014). A heuristic method to generate better initial population for evolutionary methods. *CoRR*.

Khorasgani, H., Biswas, G., and Sankararaman, S. (2016). Methodologies for system-level remaining useful life prediction. *Reliability Engineering & System Safety*, 154:8–18.

Komarnitsky, R. and Shani, G. (2016). Computing contingent plans using online replanning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*.

Krishnan, P. S. and Manimala, K. (2020). Implementation of optimized dynamic trajectory modification algorithm to avoid obstacles for secure navigation of UAV. *Applied Soft Computing*, 90:106168.

Kuhn, L. D. (2011). Self-diagnosing agent: Tight integration of operational planning and active diagnosis.

Kukačka, J., Golkov, V., and Cremers, D. (2017). Regularization for deep learning: A taxonomy.

Kulkarni, C., Biswas, G., Koutsoukos, X., Celaya, J., and Goebel, K. (2010). Integrated diagnostic/prognostic experimental setup for capacitor degradation and health monitoring. In *2010 IEEE AUTOTESTCON*, pages 1–7. IEEE.

Laporte, G. (1992). The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2).

LeCun, Y., Denker, J., and Solla, S. (1989). Optimal brain damage. In *Advances in Neural Information Processing Systems*.

Levinson, R., Frank, J. D., Iatauro, M., Knight, C. D., Sweet, A., Aaseng, G. B., Scott, M., Ossenfort, J., Soeder, J., Ngo, T., Greenwood, Z., Csank, J., Carrejo, D., and Loveless, A. T. (2018). *Development and Testing of a Vehicle Management System for Autonomous Spacecraft Habitat Operations*.

Li, H., Chaudhari, P., Yang, H., Lam, M., Ravichandran, A., Bhotika, R., and Soatto, S. (2020a). Rethinking the hyperparameters for fine-tuning. *CoRR*.

Li, J., Cheng, K., Wang, S., Morstatter, F., Trevino, R. P., Tang, J., and Liu, H. (2016a). Feature selection: A data perspective. *CoRR*.

Li, J., Sun, J., Liu, L., and Xu, J. (2021). Model predictive control for the tracking of autonomous mobile robot combined with a local path planning. *Measurement and Control*, 54(9-10):1319–1325.

Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2016b). Hyperband: A novel bandit-based approach to hyperparameter optimization. *CoRR*, abs/1603.06560.

Li, X., Ding, Q., and Sun, J.-Q. (2018). Remaining useful life estimation in prognostics using deep convolution neural networks. *Reliability Engineering & System Safety*, 172.

Li, Y.-H., Harfiya, L. N., Purwandari, K., and Lin, Y.-D. (2020b). Real-time cuffless continuous blood pressure estimation using deep learning model. *Sensors*, 20.

Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., and Thrun, S. (2005). Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling*.

Liu, D., Pang, J., Zhou, J., Peng, Y., and Pecht, M. (2013). Prognostics for state of health estimation of lithium-ion batteries based on combination gaussian process functional regression. *Microelectronics Reliability*, 53(6):832–839.

Liu, Z., Cheng, K., Huang, D., Xing, E. P., and Shen, Z. (2021). Nonuniform-to-uniform quantization: Towards accurate quantization via generalized straight-through estimation. *CoRR*, abs/2111.14826.

Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. (2018). Rethinking the value of network pruning.

Lövberg, A. (2021). Remaining useful life prediction of aircraft engines with variable length input sequences. In *Annual Conference of the Prognostics and Health Management Society*.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Maddison, C. J., Mnih, A., and Teh, Y. W. (2016). The concrete distribution: A continuous relaxation of discrete random variables. *CoRR*, abs/1611.00712.

Magid, E., Keren, D., Rivlin, E., and Yavneh, I. (2006). Spline-based robot navigation.

Mahony, R., Kumar, V., and Corke, P. (2012). Modeling, Estimation, and Control of Quadrotor. (August).

Merwe, R. V. D., Wan, E. A., and Julier, S. I. (2004). Sigma-point Kalman filters for nonlinear estimation and sensor-fusion: Applications to integrated navigation. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, page 5120, Washington, D.C.

Mishra, A. and Mohna, C. (2021). Does local pruning offer task-specific models to learn effectively?

Morris, T. P., White, I. R., and Crowther, M. J. (2019). Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, 38(11):2074–2102.

Nagel, M., Fournarakis, M., Amjad, R. A., Bondarenko, Y., van Baalen, M., and Blankevoort, T. (2021). A white paper on neural network quantization. *CoRR*.

Nebel, B. and Koehler, J. (1993). Plan modification versus plan generation: A complexity-theoretic perspective. *13th International Joint Conference on Artifical Intelligence*.

Osmić, N., Kurić, M., and Petrović, I. (2016). Detailed octorotor modeling and pd control. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*.

Padberg, M. and Rinaldi, G. (1991). A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100.

Patrikar, J., Moon, B. G., and Scherer, S. (2020). Wind and the city: Utilizing uav-based in-situ measurements for estimating urban wind fields. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1254–1260. IEEE.

Peng, Y., Dong, M., and Zuo, M. J. (2010). Current status of machine prognostics in condition-based maintenance: a review. *The International Journal of Advanced Manufacturing Technology*, 50(1-4):297–313.

Plett, G. L. (2015). *Battery Management Systems Volume 2: Equivalent-Circuit Methods*. Artech House.

Quiñones-Grueiro, M., Biswas, G., Ahmed, I., Darrah, T., and Kulkarni, C. (2021). Online decision making and path planning framework for safe operation of unmanned aerial vehicles in urban scenarios. *Aerospace Journal (to appear)*.

Rezaeianjouybari, B. and Shang, Y. (2020). Deep learning for prognostics and health management: State of the art, challenges, and opportunities. *Measurement*, 163.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "why should I trust you?": Explaining the predictions of any classifier. *CoRR*.

Righini, G., Salani, M., et al. (2006). Dynamic programming for the orienteering problem with time windows.

Rs, R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., and Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. pages 618–626.

Ryu, H. (2020). Hierarchical path-planning for mobile robots using a skeletonization-informed rapidly exploring random tree*. *Applied Sciences*, 10.

Sabih, M., Hannig, F., and Teich, J. (2020). Utilizing explainable AI for quantization and pruning of deep neural networks. *CoRR*, abs/2008.09072.

Saha, B. and Goebel, K. (2007). Battery data set. *NASA Ames Prognostics Data Repository*.

Sankararaman, S. (2015). Significance, interpretation, and quantification of uncertainty in prognostics and remaining useful life prediction. *Mechanical Systems and Signal Processing*, 52:228–247.

Saxena, A., Goebel, K., Simon, D., and Eklund, N. (2008). Damage propagation modeling for aircraft engine run-to-failure simulation. In *2008 International Conference on Prognostics and Health Management*.

Schacht-Rodriguez, R., Ponsart, J., Garcia-Beltran, C., Astorga-Zaragoza, C., Theilliol, D., and Zhang, Y. (2018). Path planning generation algorithm for a class of uav multirotor based on state of health of lithium polymer battery. *Journal of Intelligent and Robotic Systems, 91:115-131*.

Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.*, 45:2673–2681.

Shrikumar, A., Greenside, P., and Kundaje, A. (2017). Learning important features through propagating activation differences. *CoRR*, abs/1704.02685.

Si, X.-S., Wang, W., Hu, C.-H., and Zhou, D.-H. (2011). Remaining useful life estimation–a review on the statistical data driven approaches. *European Journal of Operational Research, 213(1), (2011): 1-14*.

Simonyan, K., Vedaldi, A., and Zisserman, A. (2014). Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Workshop at International Conference on Learning Representations*.

Snoek, J., Larochelle, H., and Adams, R. P. (2012a). Practical bayesian optimization of machine learning algorithms.

Snoek, J., Larochelle, H., and Adams, R. P. (2012b). Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, volume 25.

Sundararajan, M., Taly, A., and Yan, Q. (2017). Axiomatic attribution for deep networks. *CoRR*, abs/1703.01365.

Tipping, M. E. (2001). Sparse bayesian learning and the relevance vector machine. *Journal of Machine Learning Research*.

Tsui, K. L., Chen, N., Zhou, Q., Hai, Y., and Wang, W. (2015). Prognostics and health management: A review on data driven approaches. *Mathematical Problems in Engineering*.

Valavanis, K. P. and Vachtsevanos, G. J. (2015). chapter Quadcopter Kinematics and Dynamics. Springer.

Venkatasubramanian, V., Rengaswamy, R., Yin, K., and Kavuri, S. N. (2003). A review of process fault detection and diagnosis: Part i: Quantitative model-based methods. *Computers & Chemical Engineering*, 27(3):293–311.

Verdonck, T., Baesens, B., Óskarsdóttir, M., and vanden Broucke, S. (2021). Special issue on feature engineering editorial. *Machine Learning*.

Wang, W., Huang, Y., Wang, Y., and Wang, L. (2014). Generalized autoencoder: A neural network framework for dimensionality reduction. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 496–503.

Wen, L., Zhang, X., Bai, H., and Xu, Z. (2020). Structured pruning of recurrent neural networks through neuron selection. *Neural Networks*.

Wu, J., Chen, X.-Y., Zhang, H., Xiong, L.-D., Lei, H., and Deng, S.-H. (2019). Hyperparameter optimization for machine learning models based on bayesian optimizationb. *Journal of Electronic Science and Technology*, 17:26–40.

Wu, Q., Ding, K., and Huang, B. (2020). Approach for fault prognosis using recurrent neural network. *Journal of Intelligent Manufacturing*, 31.

Wu, Y., Yuan, M., Dong, S., Lin, L., and Liu, Y. (2017). Remaining useful life estimation of engineered systems using vanilla lstm neural networks. *Neurocomputing*, 275.

Xu, R., Chen, C., Lu, S., and Li, Z. (2022). Autonomous recovery from spacecraft plan failures by regulatory repair while retaining operability. *Aerospace*, 9.

Xuan, J., Wang, X., Lu, D., and Wang, L. (2017). Research on the safety assessment of the brushless dc motor based on the gray model. *Advances in Mechanical Engineering*.

Yang, F., Fang, X., Gao, F., Zhou, X., Li, H., Jin, H., and Song, Y. (2022). Obstacle avoidance path planning for uav based on improved rrt algorithm. *Discrete Dynamics in Nature and Society*, 2022.

Yeom, S.-K., Seegerer, P., Lapuschkin, S., Binder, A., Wiedemann, S., Müller, K.-R., and Samek, W. (2021). Pruning by explaining: A novel criterion for deep neural network pruning. *Pattern Recognition*.

You, K., Long, M., Wang, J., and Jordan, M. I. (2019). How does learning rate decay help modern neural networks. *arXiv: Learning*.

Youn, B. and Wang, P. (2012). *A Generic Bayesian Framework for Real-Time Prognostics and Health Management (PHM)*.

Zammit, C. and Van Kampen, E.-J. (2020). Comparison of a* and rrt in real–time 3d path planning of uavs. In *AIAA Scitech 2020 Forum*, page 0861.

Zebari, R., Abdulazeez, A., Zeebaree, D., Zebari, D., and Saeed, J. (2020). A comprehensive review of dimensionality reduction techniques for feature selection and feature extraction. *Journal of Applied Science and Technology Trends*, 1.

Zhang, L., Lin, J., Liu, B., Zhang, Z., Yan, X., and Wei, M. (2019). A review on deep learning applications in prognostics and health management. *IEEE Access*, 7.

Zhang, Y., Chamseddine, A., Rabbath, C. A., Gordon, B. W., Su, C.-Y., Fulford, C., Apkarian, J., and Gosselin, P. (2012). Fault diagnosis, fault-tolerant and cooperative control for unmanned systems. *8th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes*.

Zheng, A. and Casari, A. (2018). *Feature engineering for machine learning: principles and techniques for data scientists*. " O'Reilly Media, Inc.".