A NEURAL-NETWORK-BASED CONTROLLER FOR MISSED-THRUST INTERPLANETARY TRAJECTORY DESIGN

by

Paul Witsberger

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Aeronautics and Astronautics

West Lafayette, Indiana

May 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. James M. Longuski, Chair

School of Aeronautics and Astronautics

Dr. William A. Crossley

School of Aeronautics and Astronautics

Dr. Kathleen C. Howell

School of Aeronautics and Astronautics

Dr. Yung C. Shin

School of Mechanical Engineering

Approved by:

Dr. Gregory A. Blaisdell

Dedicated to my wife, who is the best spouse,
to my parents, who are the best parents,
and to my dog, who is the best dog.

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. James Longuski, for his support and guidance throughout my graduate education. I also want to thank the other members of my advisory committee, Prof. Bill Crossley, Prof. Kathleen Howell, and Prof. Yung Shin, for their insightful questions and guidance as well as for teaching my favorite classes.

I want to acknowledge the contributions of my fellow research group members, past and current: Dr. Peter Edelman, Jim Moore, Rob Potter, Alec Mudek, Jeffrey Pekosh, Dr. Athul Girija, Rachana Agrawal, Jay Millane, Archit Arora, and Weston Buchanan. They suffered through many hours of research meetings where I attempted to explain machine learning again and again. Also, I thank my colleagues outside of my research group who for some reason decided to help me out when I asked: Bobby Rolley, Andrew Williams, Stephen Fulton, Eiji Shibata, Nick LaFarge, RJ Power, Dr. Robert Pritchett, Andrew Berger, and Dr. Charles Meade. Charles, as requested, gets 1/10,000th of this degree.

I would like to acknowledge the support of the faculty and staff in the Purdue University School of Aeronautics and Astronautics for providing me with support for my 5 years of doctoral studies, plus 1 year of master's studies, plus 4 years of undergraduate studies...it's been a long ride.

I also want to thank my many mentors over the years for their expert guidance: Dr. Geoffrey Landis, Dr. Al Hepp, Dr. Oleg Sindiy, Dr. Anastassios Petropoulos, Dr. Jon Sims, Dr. Lee Coduti, and Mr. Kevin Albarado.

Finally, I absolutely would not have made it this far without the unwavering support of my family and my wife. My parents, Todd and Kathleen, have supported me every step of the way and believed in me even when I did not believe in myself. My sisters, Sarah and Emily, accepted me even though I was the black sheep of the family. My parents-in-law, Dave and Marcia, and my sister-in-law, Casey, have accepted me into their pack and taught me more quotes than I thought any one person could know. My wife, Chelsey, is the coolest person ever and I'm lucky that we're able to help each other reach our optimal levels of success.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Δv	delta v (change in velocity)
m	mass
v	velocity
μ	standard gravitational parameter
a	semi-major axis
e	eccentricity
ω	argument of periapsis
f	true anomaly
$\oplus$	Earth planetary symbol
$\male$	Mars planetary symbol
J	performance index
m	mass
c	exhaust velocity
I_{sp}	specific impulse (impulse per unit mass)
T_{max}	maximum available thrust
δ_m	throttle (fraction of max thrust used)
ϵ	smoothing parameter

ABBREVIATIONS

2BP	two-body problem
CR3BP	circular restricted three-body problem
EP	electric propulsion
FCN	fully-cascaded network
FFNN	feed-forward neural network
GA	genetic algorithm (note: *not* gravity assist)
IMLEO	initial mass in low-Earth orbit
LEO	low-Earth orbit
LVLH	local-vertical, local-horizontal
MEE	Modifed Equinoctial Elements
MLP	Multi-Layer Perceptron
MTE	missed-thrust event
NEAT	NeuroEvolution of Augmenting Topologies
NN	neural network
RBDO	Reliability-Based Design Optimization
ReLU	rectified linear unit
RNN	recurrent neural network
RL	reinforcement learning
SOI	sphere of influence
tanh	hyperbolic tangent
TOF	time-of-flight
TPBVP	two-point boundary value problem

ABSTRACT

The missed-thrust problem is a modern challenge in the field of mission design. While some methods exist to quantify its effects, there still exists room for improvement for algorithms which can fully anticipate and plan for a realistic set of missed-thrust events. The present work investigates the use of machine learning techniques to provide a robust controller for a low-thrust spacecraft. The spacecraft's thrust vector is provided by a neural network controller which guides the spacecraft to the target along a trajectory that is robust to missed thrust, and the controller does not need to re-optimize any trajectories if it veers off its nominal course. The algorithms used to train the controller to account for missed thrust are supervised learning and neuroevolution. Supervised learning entails showing a neural network many examples of what inputs and outputs should look like, with the network learning over time to duplicate the patterns it has seen. Neuroevolution involves testing many neural networks on a problem, and using the principles of biological evolution and survival of the fittest to produce increasingly competitive networks. Preliminary results show that a controller designed with these methods provides mixed results, but performance can be greatly boosted if the controller's output is used as an initial guess for an optimizer. With an optimizer, the success rate ranges from around 60% to 96% depending on the problem.

Additionally, this work conducts an analysis of a novel hyperbolic rendezvous strategy which was originally conceived by Dr. Buzz Aldrin. Instead of rendezvousing on the outbound leg of a hyperbolic orbit (traveling away from Earth), the spacecraft performs a rendezvous while on the inbound leg (traveling towards Earth). This allows for a relatively low Δv abort option for the spacecraft to return to Earth if a problem arose during rendezvous. Previous work that studied hyperbolic rendezvous has always assumed rendezvous on the outbound leg because the total Δv required (total propellant required) for the insertion alone is minimal with this strategy. However, I show that when an abort maneuver is taken into consideration, inserting on the inbound leg is both lower Δv overall, and also provides an abort window which is up to a full day longer.

1. INTRODUCTION

1.1 Motivation

Electric propulsion is a very efficient form of propulsion compared to traditional chemical propulsion, but the high efficiency comes at the cost of low levels of thrust. To compensate for the low amount of thrust provided, spacecraft that use electric propulsion must thrust for long periods of time. In cislunar space, this could mean days to months of thrusting; for trajectories to Mars or beyond, this could mean months to years of thrusting. For such long time spans, system reliability becomes a significant concern as the probability of safe-mode events occurring on the spacecraft due to failures of components increases. During a safe-mode event, the spacecraft shuts down most major functions, including propulsion [1]. Thus, if a safe-mode event occurs when the spacecraft should be thrusting, the spacecraft would "miss" its thrust, and the spacecraft would now be off its nominal trajectory. This possibility of missed thrust, and the attempts to mitigate its risks, is called the missed-thrust problem. A representative diagram of a missed-thrust event is shown in Figure 1.1.

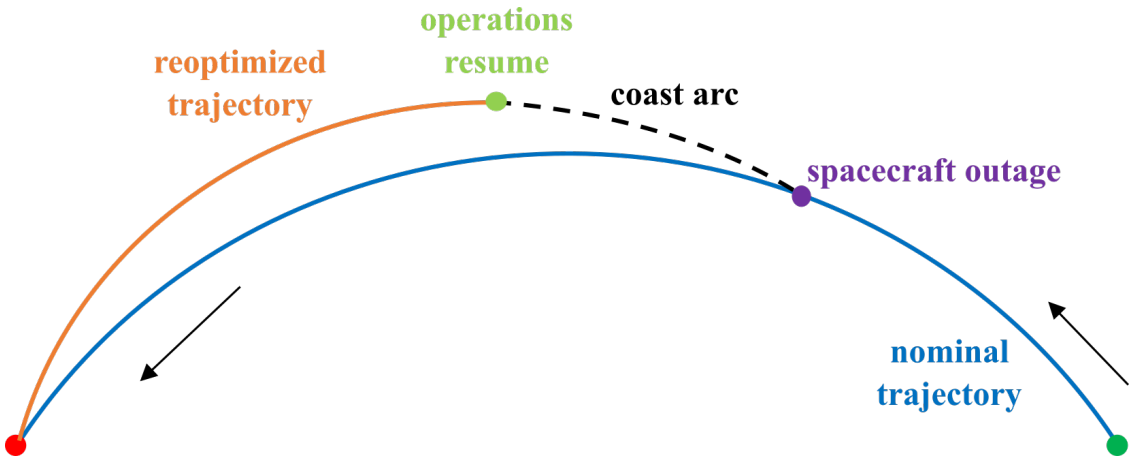

Figure 1.1. Representative diagram of a missed-thrust event. When a spacecraft experiences an outage when it should be thrusting, it can drift away from the nominal trajectory. Once its operations resume, the spacecraft's trajectory must be re-optimized to reach the target.

Looking at data regarding safe mode events from historical missions over the past 40 years, we see that safe mode events are actually more likely to occur during a mission than

not [1]. Thus, there is great motivation to be able to effectively quantify the effects of missed-thrust events as well as develop robust methods to prepare for, mitigate, and recover from such events.

Previous attempts to address the missed-thrust problem involve using traditional trajectory design techniques to create a nominal trajectory, discretizing this trajectory into several subsections, and then observing the effects when the spacecraft is forced to coast for each subsection [2]–[4]. All of these forced-coast arcs result in a new trajectory, each of which then needs to be re-optimized to reach the target. However, each new trajectory can also experience future missed-thrust events, and the process needs to be repeated for every one [5]. Performing such an analysis yields approximate margins on propellant mass and time-of-flight, but it does not directly mitigate the risk, and it is very time-consuming both computationally and in terms of human effort. If there are any sections of the trajectory that are discovered to be particularly sensitive to a missed-thrust event, mission designers must manually adjust what the thrust history of the spacecraft should be to reduce risk [6].

Recently, new types of analyses using machine learning have been suggested. Ozaki et al. used stochastic differential dynamic programming to perform low-thrust trajectory optimization [7]. The addition of stochastic disturbances during training enabled their method to perform favorably compared to a model that did not have such disturbances in training. While promising, the stochastic disturbances that they applied are smaller in magnitude than typical missed-thrust events, so future work could involve testing their controller on a more representative test problem.

Independently, papers by Izzo, Sprague, and Tailor, and Rubinsztejn, Sood, and Laipert looked into using supervised learning to train a neural network to control a spacecraft [8], [9]. Izzo et al. looked at the problem from a trajectory design perspective, and did not explicitly consider missed thrust. They looked at a 3D, elliptical transfer from Earth to Mars with promising results. Rubinsztejn et al. used the algorithm as a way to address the missed-thrust problem. They found that training a neural network on optimized trajectories within a small range of boundary conditions was successfully able to guide the spacecraft to its target, and that including missed-thrust events in the training data did not increase the neural network's success rate. However, their problem of interest was a 2D, circular,

co-planar, Mars-to-Earth transfer with a fixed time of flight. Future work in this area should include proving that such an algorithm could be applied to more complex problems.

From the work that has been performed to date, we can see that there are a few margin-based options to quantify the effects of missed-thrust events on a trajectory, and there are a few machine-learning-based approaches to creating a robust controller that can account for missed thrust. With this information and based on prior personal experience, I believe that a different type of machine learning with strengths of Ozaki et al., Izzo et al., and Rubinsztejn et al. can serve as a robust controller to a broad range of complex missed-thrust trajectories.

Ozaki et al. include stochastic disturbances in their training, and these disturbances allow them in theory to move towards a more globally-optimal solution. This ability to step out of a local optimum is desirable. However, I want to account for more than just small disturbances. Rubinsztejn et al. use supervised learning to train a neural network to account for missed thrust on a simple problem. I use a neural network as a controller to account for missed thrust as they do, but I would like to explore options aside from supervised learning since supervised learning is not able to provide solutions of a different form from what is in the training data set.

As I tried to frame my research, my initial goal became to use a genetic algorithm to train a neural network which acts as a controller that is robust to missed-thrust events. Properly training such a neural network would mean that, in the event of a missed-thrust event, the spacecraft would be able to guide itself to the target from its new state without needing to optimize any new trajectories. The genetic algorithm enables searching for a globally-optimal solution, and it can be applied to arbitrarily complex problems since there are no constraints on continuity or derivatives. Unlike with supervised learning, a training data set does not need to be generated. In fact, using an evolutionary method allows the spacecraft to learn behaviors that may not be present in any training data that would be used—that is, a robust trajectory may include features that are not present in a typical fuel-optimal trajectory. Also, since reinforcement learning is not being used, I do not need to worry about formulating an intermediate cost function and back-propagating the weight updates. Instead, I just need the boundary conditions for the trajectory problem I am trying to solve, set up an appropriate terminal cost function, and let the algorithm run. The resulting trajectories

may not be fuel-optimal when missed thrust is not considered, but they are likely to use less propellant or experience fewer delays when missed-thrust events are allowed to occur.

The algorithm is general enough that it can be applied to the two-, three-, or n-body problem, it can include missed-thrust events or not, and the cost function can be a complex weighted function of many factors and constraints. As I show, there is a special type of genetic algorithm that is formulated to evolve neural networks known as Neuro-Evolution of Augmenting Topologies (NEAT). Using NEAT, I demonstrate that a neural network controller can successfully guide a spacecraft to its target in the presence of missed-thrust events and can even handle variable boundary conditions.

While NEAT has its advantages, the fact that it does not use derivatives is a double-edged sword. There are numerous very capable calculus-based trajectory optimization and control algorithms available to optimize a single trajectory at a time. We can utilize these existing techniques to create large sets of optimal trajectories, which a neural network can then learn from using supervised learning. By training a network with supervised learning in addition to NEAT, we can understand more fully the pros and cons of each strategy. While different missions have different objectives, in this dissertation I will primarily look at fuel-optimal trajectories; that is, trajectories which use as little propellant as possible to complete a desired transfer while satisfying all of the problem constraints.

A neuroevolution-based algorithm that is similar to what I use was formerly demonstrated by Dachwald and Seboldt, who used it to design solar-sailcraft, dlow-thrust, and low-thrust gravity-assist trajectories [10]–[13]. In their work, they successfully demonstrate that a genetic algorithm can be used to find the weights of a neural network that provides a near-optimal steering law for various complex mission scenarios. However, they did not look at the missed-thrust problem, and they used a plain genetic algorithm instead of NEAT.

A benefit of using a neural-network-based controller is that once the network is trained, no more trajectory optimization or training is required. The neural network can be used during trajectory propagation, which is a much more computationally light operation than optimization. This feature could allow neural networks to be deployed on a flight computer, where computational resources are very limited, and prevents mission designers on the ground from needing to quickly find a new optimal trajectory. The controller could act as an initial

recovery method if a safe-mode event occurred while mission designers on Earth determine what the full recovery plan should be.

1.2 Overview

Chapter 2 provides a background of the existing algorithms and underlying principles used in this dissertation. Generally speaking, this includes neural networks, genetic algorithms, the missed-thrust problem, the forces that I include which act on the spacecraft, and indirect optimal control.

Chapter 3 uses neuroevolution to train a neural network that acts as a controller for a low-thrust spacecraft. First, a standard implementation of neuroevolution is used to compare the performance of a recurrent neural network compared to a feed-forward neural network. After this, a more complex version of neuroevolution called NEAT is used to look at the missed-thrust problem.

Chapter 4 discusses trajectory design using supervised learning. This includes describing the algorithm used to generate a large training data set, strategies to use intermediate versions of the neural network to boost the overall training time, and presents preliminary results using this method. The chapter then goes a step further to measure its performance on the missed-thrust problem.

Chapter 5 compares supervised learning and NEAT and discusses the pros and cons of each method. Each algorithm is applied to a more challenging trajectory design problems, both with and without missed thrust. Additionally, a brute force method is used to benchmark the performance of the neural networks against traditional trajectory design techniques.

Chapter 6 analyzes a novel hyperbolic insertion paradigm for human missions to destinations beyond cislunar space that was originally conceived by Dr. Buzz Aldrin. Dr. Aldrin reached out to Professor Longuski's research group requesting an analysis on his idea, and in this chapter I discuss the results.

Chapter 7 provides a brief discussion regarding conclusions from the work, as well as options for future research.

1.3 Dissertation Contributions

In this dissertation I investigate the performance of neural-network-based controllers in the context of interplanetary trajectory design. The major contributions of this work include the following points.

1. A new method for trajectory design using NeuroEvolution of Augmenting Topologies is presented. This method is very computationally expensive compared to traditional methods when considering a single TPBVP, but it is more competitive when taking into account the more complex missed-thrust problem. Additionally, NEAT enables searching for a globally optimal solution that local optimizers may struggle to find. However, NEAT does not guarantee a locally optimal solution. NEAT struggles when moving from 2D to 3D, and may perform better if given access to additional computing resources.

2. The effectiveness of a controller based on a neural network trained with supervised learning is investigated in the context of the missed-thrust problem. A neural network's performance is shown to improved by increasing its size and the amount of training data, although this comes at the cost of greatly increased computational resources. Using the output of a neural network trained with supervised learning directly has mixed results, but passing the outputs to an optimizer shows greatly improved performance.

3. The two training methods, NEAT and supervised learning, are compared to each other as well as a non-neural-network-based trajectory design strategy to show their relative performance. NEAT was not able to successfully train a neural network to complete this problem. Supervised learning showed a clear increase in effectiveness compared to the brute force method.

4. A novel method for an abort maneuver during hyperbolic rendezvous is presented. This method is demonstrated to have a lower propellant cost than other methods in literature when an abort maneuver is taken into consideration.

2. BACKGROUND

This chapter is meant to provide a brief description of the existing algorithms and concepts that are foundational to the research performed later in this dissertation. I explain the theory and computation of neural networks, genetic algorithms, the missed-thrust problem, and the indirect optimal control problem.

2.1 Neural Networks

In this section, I discuss the motivation behind neural networks as well as some types of topologies and training methods.

2.1.1 Overview

Artificial neural networks, typically referred to as neural networks, are inspired by the parallel information processing scheme of living creatures' brains and provide a mathematical framework for representing nonlinear functions or relationships between sets of inputs and outputs [14]. A neural network is composed of processing nodes called neurons which take an incoming signal or set of signals, apply a nonlinear function, and generate an output. This process is an abstraction of an organic neuron receiving inputs from other neurons, and then firing once a threshold has been reached. By chaining many neurons together in series and in parallel, a neural network can represent very complex functions. In fact, it has been shown that a single-layer neural network can approximate any bounded, continuous mapping between two Euclidean spaces with arbitrary accuracy given sufficient hidden nodes through the universal approximation theorem [15].

Each connection between individual neurons, sometimes referred to as perceptrons or nodes, is assigned a weight, and the output of the earlier neuron is multiplied by the weight of the connection. Each neuron computes the sum of its incoming signals, adds a scalar bias, applies an activation function to the sum, and outputs a single scalar value. This output value is then passed to neurons downstream in the network. A diagram of the computation across one neuron is shown in Figure 2.1. The equation to compute the output of a neuron is

shown in Equation (2.1), where x_i are the inputs, w_i are the weights connecting each input to the neuron, b is the bias, Σ represents the summation function, σ represents an activation function, and y is the output.

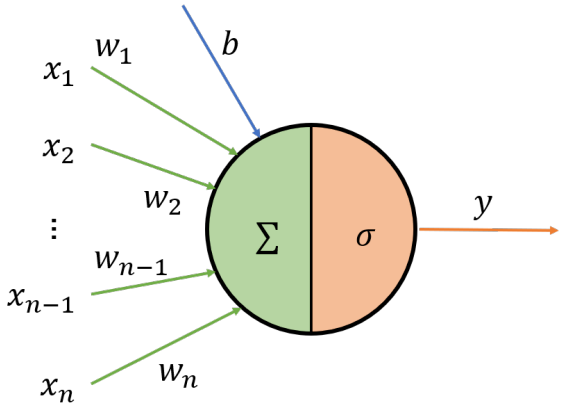

Figure 2.1. A single neuron receives weighted inputs (green arrows) and a bias (blue arrow), sums them together, and applies an activation function to get its output (orange arrow).

$$y = \sigma\left(\sum_{j=0}^{n} w_j x_j + b\right) \tag{2.1}$$

Many neurons can be combined together both in serial or in parallel. A group of neurons placed in parallel connections with each other is commonly referred to as a layer, and layers can be serially connected to one another. Internal layers of a neural network (i.e. layers other than the inputs or outputs) are called hidden layers. A diagram of a simple neural network with one hidden layer of three neurons connecting two inputs and two outputs is shown in Figure 2.2. Here the dimensions of the input space and output space are each two, but they can each be any arbitrary positive integer.

To compute the output of a network, the computations must flow from the input layer through the hidden layer(s) to the output layer. The computation can be simplified by writing Equation (2.1) in vector notation as shown in Equation (2.2).

$$y = \sigma(\mathbf{w}^T\mathbf{x} + b) \tag{2.2}$$

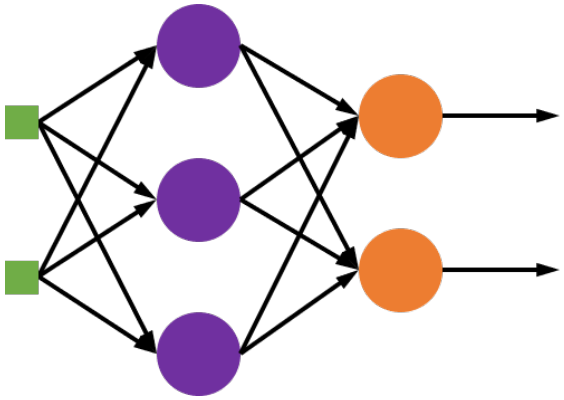

Figure 2.2. A neural network with two inputs (green squares), three hidden nodes (purple circles), and two output nodes (orange circles).

Now, computing the weighted sum of the inputs to each neuron in a layer can be performed simultaneously using matrix operations as shown in Equation (2.3) [16]. Here, $\mathbf{y}$ and $\mathbf{b}$ (bold face) are vectors and W (capital letter) is a matrix. Additionally, $\sigma(\cdot)$ operates element-wise on the vector output of the term $W\mathbf{x} + b$. Finally, Equation (2.4) shows how the output of the network is obtained by recursively repeating Equation (2.3) for each layer, where $\mathbf{x}_1$ is the input layer and, assuming the computation is repeated $n-1$ times, x_n is the output layer. Each layer will have its own set of weights, biases, and activation functions.

$$\mathbf{y} = \sigma(W\mathbf{x} + \mathbf{b}) \tag{2.3}$$

$$\mathbf{x}_{i+1} = \sigma_i(W_i\mathbf{x}_i + \mathbf{b}_i), i \in \{1, ..., n-1\} \tag{2.4}$$

Activation functions are an abstraction of the concept of a real neuron "firing" when a threshold has been met. Mathematically this can be represented by the Heaviside step function, where zero represents being inactive and one represents being active. However, the Heaviside function is discontinuous at zero and its derivative is zero everywhere, which is problematic when training a neural network. To overcome this, researchers started using the standard logistic function, which is a type of sigmoid function. In the machine learning community the standard logistic function is commonly referred to simply as sigmoid. Sigmoid is a monotonically increasing function with a range of $(0, 1)$ on the domain $(-\infty, +\infty)$. Over

time, hyperbolic tangent (tanh) began replacing sigmoid as an activation function due to its stronger gradient near zero and its lack of bias in the gradient [17]. Hyperbolic tangent is defined on the range $(-1, 1)$ with domain $(-\infty, +\infty)$ and its derivative is in the range $(0, 1]$, compared to sigmoid's derivative's range of $(0, 0.5]$. Figure 2.3 shows a comparison of the Heaviside, sigmoid, and hyperbolic tangent functions, and Equation (2.5)-(2.7) provide the definitions of each function, respectively.

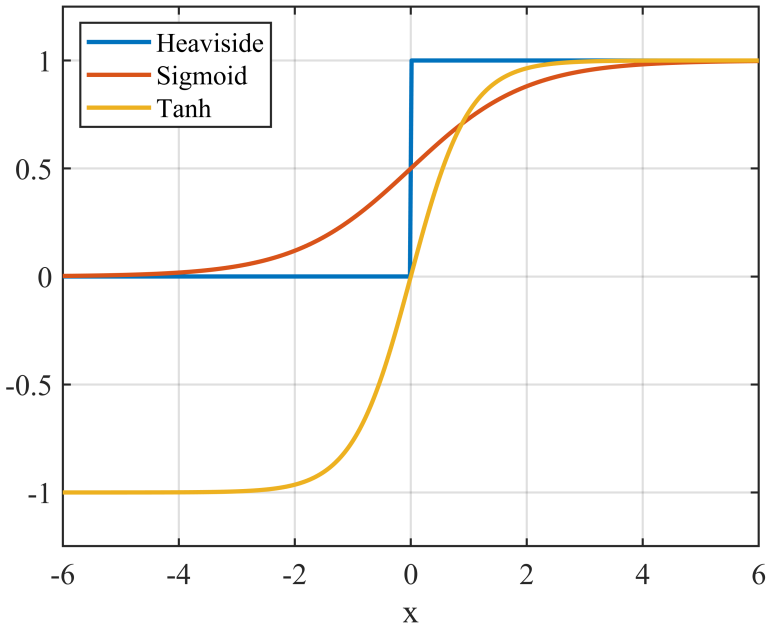

Figure 2.3. The Heaviside function represents a binary switch from "off" (zero) to "on" (one). The sigmoid function is monotonically increasing with a positive derivative defined at all points. Hyperbolic tangent is centered around zero with a larger gradient at zero than sigmoid.

$$H(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leqslant 0 \end{cases} \tag{2.5}$$

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.7}$$

Both sigmoid and hyperbolic tangent, and squashing functions in general, suffer from the "vanishing gradient" problem. This term refers to the reduction in the gradient of the functions as the input moves farther from zero. The result of this problem is that networks can have a hard time learning when the inputs are large because there is little information provided by the gradients. To get around this, researchers began moving to the rectified linear unit (ReLU). "Rectified" means that negative values are set to zero, so ReLU means that the function is zero when x is negative, and the function is x when x is positive. Figure 2.4 shows ReLU around zero and Equation (2.8) provides its mathematical definition. Also, due to the simple nature of the function, ReLU's derivative is much faster to calculate than sigmoid and tanh which both require computing the relatively expensive exponential function.

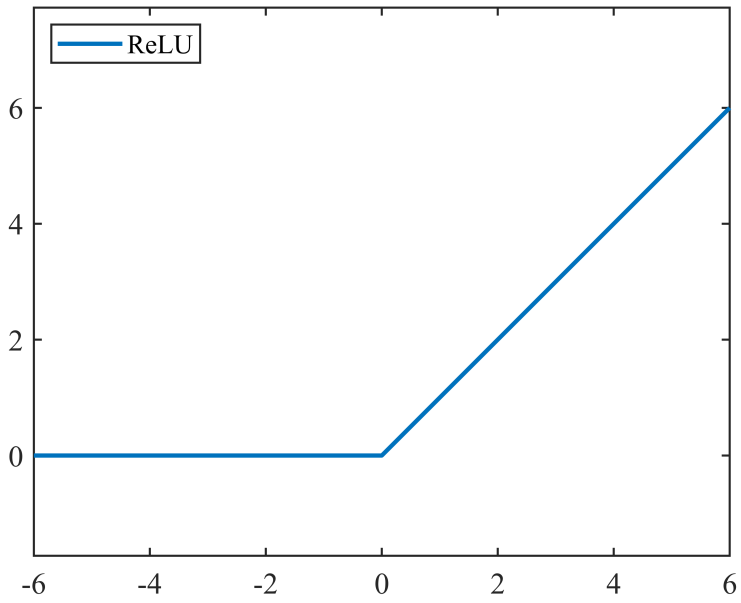

Figure 2.4. The ReLU function returns zero when the input is negative, and returns the input when it is positive.

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leqslant 0 \end{cases} \tag{2.8}$$

The activation functions discussed above prove sufficient for this work, although it is worth noting that there has been much research into the effectiveness of modifications or alternatives to these functions, and other applications may benefit from other types of activation functions.

2.1.2 Types of Neural Networks Topologies

A network which does not contain any feedback loops between nodes is called a feed-forward neural network (FFNN). A FFNN that is organized into discrete layers is called a multi-layer perceptron (MLP). Figure 2.2 shows an example of an MLP with one hidden layer where every node connects to nodes farther down the network. MLPs are commonly used for function regression or classification tasks.

Recurrent neural networks, or RNNs, are similar to FFNNs, except they have an additional feedback loop in the network. At each sequence step, hidden nodes receive as input their output from the previous sequence step. An RNN is shown in Figure 2.5 with the feedback connections shown in light blue. RNNs are used for sequence modeling, such as time series forecasting or language processing.

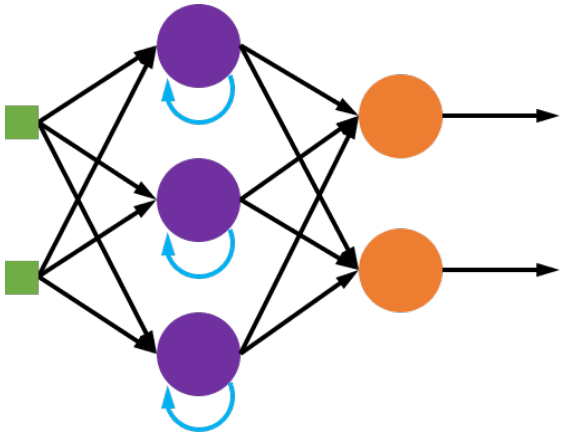

Figure 2.5. A recurrent neural network has feedback connections between some or all of its neurons.

Neural networks do not require explicit layers, but rather can take very complicated topologies. Fully-cascaded networks (FCN), sometimes called cascade-forward networks, are a class of FFNNs in which every neuron has a connection to every subsequent neuron as

shown in Figure 2.6. The top network in the figure shows the same FFNN as in Figure 2.2 but rearranged to help show similarities with the bottom network. The new connections are shown in light blue. FCNs are generally used for the same types of problems as MLPs, and FCNs typically need far fewer neurons than an MLP due to the increased number of connections within the network.

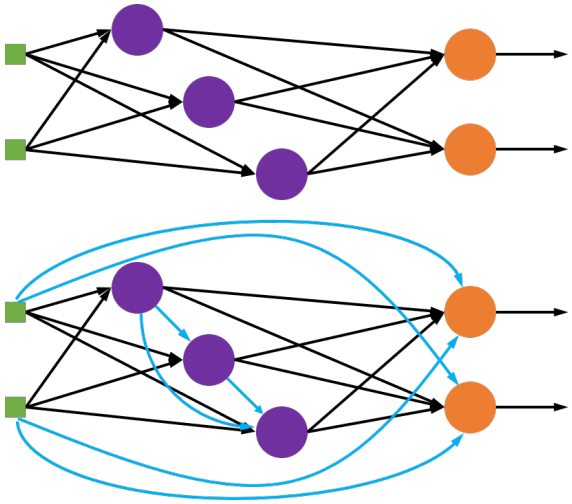

Figure 2.6. The top network is the same as the network in Figure 2.2, but with the hidden neurons rearranged. The bottom network shows a cascade-forward network, where each neuron connects to each subsequent neuron.

Neural networks can have a mix of connections amongst inputs, outputs, and hidden nodes without following a strict structure. An example of a neural network without any particular layered structure is shown in Figure 2.7. This network has a connection from an input directly to an output node, connections to a hidden node from both another hidden node and an input, and has hidden nodes that pass their output to only some of the subsequent hidden nodes or output nodes.

There are many other network topologies that are outside the scope of this work. Some such types include: 2D and 3D convolutional neural networks, which are used for image and video processing; Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRU), which are used for sequence modeling and natural language processing (NLP); and Generative Adversarial Networks (GAN), which are commonly used for image generation.

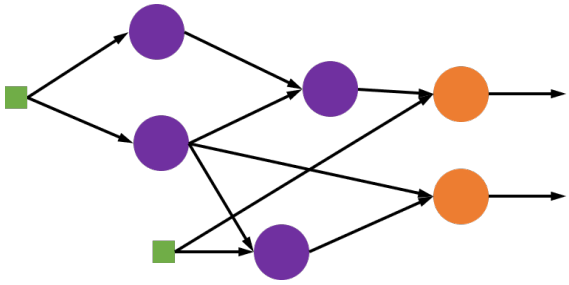

Figure 2.7. A network can have complicated connections between inputs, outputs, and hidden nodes.

2.1.3 Training Methods

The process of finding appropriate weights and biases so that a neural network produces the desired function or relationship is called training. When a neural network is created, its weights and biases are generated with random values. This is similar to an infant's brain at birth. It has an immense capacity to learn, but it does not yet possess any "knowledge" (beyond instinctive behavior). When a network is initialized, the relationship between the inputs and outputs is nothing meaningful, and we need some way to update the weights such that the network represents our desired function. I discuss two methods of training. Supervised learning is where the network learns from patterns that it is shown. This method is analogous to an infant learning to say words when its parents say words to it, or to the expression "monkey see, monkey do". The other method is called neuroevolution, where many networks with different weights are compared to each other, and the higher performers move to subsequent iterations. This method is analagous to survival of the fittest, where stronger animals will have more offspring in future generations. It also has similarities to reinforcement learning, such as learning that touching a hot stove is bad, eating candy provides instant gratification, and studying and researching for a PhD provides (very) delayed gratification.

Supervised Learning

Supervised learning requires a training data set which is composed of pairs of inputs and the corresponding desired outputs. During training, the network is given the inputs and its

output is computed. The error between the actual and desired outputs is computed, and this error can be back-propagated through the network so that the network outputs the desired value when given that set of inputs. By repeating this process for a sufficiently large training set, a neural network can "learn" to produce very complex, nonlinear relationships. The process of computing the error and updating the weights is called back-propagation[18].

Back-propagation works by computing the gradient of the error with respect to each element of the network via the chain rule. Let us look at an example with a simple MLP with an input layer, two hidden layers, and an output layer. Forward propagation through the network be computed by Equation (2.9). Here $\mathbf{x}$ is the input, $\mathbf{z}^{(j)}$ is the vector of weighted sums of inputs and biases for the j^{th} layer, $\mathbf{a}^{(j)}$ is the vector of activations (outputs) for the j^{th} layer, $W^{(j)}$ is the weight matrix connecting the j^{th} layer to the $(j+1)^{th}$ layer, b is the vector of biases for the j^{th} layer, and $f^{(j)}$ is the activation function for the j^{th} layer, and $\mathbf{p}$ is the linear activation of the output layer.

$$\begin{aligned} \mathbf{x} &= \mathbf{a}^{(1)} \\ \mathbf{z}^{(2)} &= W^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\ \mathbf{a}^{(2)} &= f(\mathbf{z}^{(2)}) \\ \mathbf{z}^{(3)} &= W^{(2)}\mathbf{a}^{(2)} + \mathbf{b}^{(2)} \\ \mathbf{a}^{(3)} &= f(\mathbf{z}^{(3)}) \\ \mathbf{p} &= W^{(3)}\mathbf{a}^{(3)} + \mathbf{b}^{(3)} \end{aligned} \tag{2.9}$$

Now we can compute the error between the neural network's output and the target value using a cost function. Common cost functions are mean squared error for regression and cross-entropy for classification. Regression is, generally speaking, the task of finding the best curve fit for data (such as fitting a linear relationship to scattered data). Classification is the task of picking one choice from many options (e.g. an image contains either a cat or a dog). For mean squared error, the cost C can be computed by Equation (2.10).

$$C(\mathbf{p}, \mathbf{y}) = \frac{1}{2} \sum_{k=0}^{n} \left\| \mathbf{y}_k^2 - \mathbf{p}_k^2 \right\| \tag{2.10}$$

With the cost computed, we can begin back-propagating to update the weights. Equation (2.11) shows the computation of the gradient of the cost C with respect to the weight w_{ik}^j, where i is input index for the current layer, j is the layer index, and k is the output index for the current layer.

$$\begin{aligned} \frac{\partial C}{\partial w_{ik}^j} &= \frac{\partial C}{\partial z_i^j} \frac{\partial z_i^j}{\partial w_{ik}^j} \\ z_i^j &= \sum_{k=1}^{m} w_{ik}^j a_k^{j-1} + b_i^j \\ \frac{\partial z_i^j}{\partial w_{ik}^j} &= a_k^{j-1} \\ \frac{\partial C}{\partial w_{ik}^j} &= \frac{\partial C}{\partial z_i^j} a_k^{j-1} \end{aligned} \tag{2.11}$$

The biases b_i^j can be computed in a similar manner, shown in Equation (2.12).

$$\begin{aligned} \frac{\partial C}{\partial b_i^j} &= \frac{\partial C}{\partial z_i^j} \frac{\partial z_i^j}{\partial b_i^j} \\ \frac{\partial z_i^j}{\partial b_i^j} &= 1 \\ \frac{\partial C}{\partial b_i^j} &= \frac{\partial C}{\partial z_i^j} \end{aligned} \tag{2.12}$$

The common term in both Equation (2.11) and (2.12) is called the local gradient.

$$\delta_i^j = \frac{\partial C}{\partial z_i^j} \tag{2.13}$$

The local gradient can be computed by finding the partial derivative of the activation.

$$\delta_i^j = \frac{\partial C}{\partial z_i^j} = \frac{\partial C}{\partial a_i^j} \frac{\partial a_i^j}{\partial z_i^j} \tag{2.14}$$

The equation for the partial derivative of the activation with respect to the sum depends on the activation function itself. The derivatives of sigmoid, hyperbolic tangent, and ReLU are shown in Equation (2.15).

$$\begin{aligned} \frac{\partial}{\partial z}\sigma(z) &= \sigma(z)(1-\sigma(z)) \\ \frac{\partial}{\partial z}\tanh(z) &= 1-\tanh^2(z) \\ \frac{\partial}{\partial z}\mathrm{ReLU}(z) &= \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leqslant 0 \end{cases} \end{aligned} \tag{2.15}$$

Starting from the output, every partial derivative in the network can be computed. Since the derivatives are computed from the output "backwards" towards the inputs, the process is called back-propagation. Once all of the derivatives have been computed, we can update weights and biases by Equation (2.16).

$$\begin{aligned} W &= W - \epsilon\frac{\partial C}{\partial W} \\ \mathbf{b} &= \mathbf{b} - \epsilon\frac{\partial C}{\partial \mathbf{b}} \end{aligned} \tag{2.16}$$

The term ϵ is called the learning rate, and it affects the amount by which the current iteration's error should affect the weight and bias updates. The process of computing the gradients and updating the weights is repeated until terminal conditions are met such as the cost function reaching an optimum or an iteration limit being reached.

Neuroevolution

There are several other categories of training aside from supervised learning including neuroevolution, unsupervised learning, reinforcement learning, generative adversarial networks, and more. Neuroevolution is the process of training a neural network using an evolutionary algorithm. With neuroevolution, a network's weights are set as the design variables of a genetic algorithm (or another evolutionary algorithm) and the network's performance

on a desired task is measured to determine its fitness (for more information on genetic algorithms, see section 2.2). In this way, the network is trained to produce a desired function, but no training data is required a priori. Neuroevolution can be used in a similar manner to reinforcement learning, in which an agent takes actions to interact with its environment with the goal to maximize cumulative reward.

A popular method of neuroevolution is called NeuroEvolution of Augmenting Topologies (NEAT) [19]. As the name suggests, NEAT adjusts not only the weights of a neural network but also the topology. The topology of a network includes the number of nodes, connections between nodes, and activation functions of nodes. During training, NEAT can add or remove nodes and connections, allowing it to find an appropriate amount of complexity to solve the problem. Other forms of neuroevolution typically begin with a fixed structure or topology - that is, the number of nodes and the connections between them are fixed. By enabling the ability to have variable topology, NEAT optimizes and "complexifies" solutions simultaneously.

Each genome is composed of a set of connection genes and node genes. Node genes are assigned an integer value to represent each of the input, hidden, and output nodes that can be connected, whereas connection genes specify the in-node, out-node, weight, whether or not the gene is expressed, and innovation number. The innovation number is simply an index or ID number that uniquely identifies every connection in the network. During crossover, only connections with the same innovation number can be combined. Genes that are not included in both parents are called disjoint or excess, depending on whether the innovation number is less than the maximum innovation number of the other parent. This scheme of crossover enables any two networks in the population to be crossed over with each other, even if they have different sizes. Disjoint and excess genes are inherited from the more fit parent.

During the mutation phase, connections can be added or removed from existing nodes, or new nodes can be placed along a connection, splitting it into two. Since a change in topology can have a significant impact on the fitness of the network, NEAT utilizes the concept of speciation to protect new members for a short time to allow them to be optimized somewhat before being compared to the broader population. Speciation divides the total population

based on the similarity of networks to one another, and only individuals within each species will be compared to each other. It is common for networks generated by NEAT to have asymmetric, non-layered structures such as the network shown in Figure 2.7 due to the random nature of how the structure is generated.

NEAT biases the search toward minimal-dimensional spaces by starting with the minimum-size network - direct connections from the inputs to the outputs with no hidden nodes. The algorithm introduces new structure incrementally as mutations occur, and only the mutations that provide a benefit survive. Thus, by finding a minimal-dimension (or near-minimal) solution, NEAT offers significant computational advantages compared to other training approaches that begin with often much larger spaces. Here, minimum dimension refers to the dimension of the training space, or the number of trainable parameters. These include the weights and biases of the network, which are dependent on the number of nodes and connections within the network.

2.2 Genetic Algorithms

Genetic algorithms are global search algorithms based on natural selection and biological evolution [20], [21]. First, a set of designs is created where each design is randomly generated such that its design variables fall within acceptable bounds. Each design is evaluated using a cost function that determines the design's "fitness". The fitness represents how well an individual achieves a specified task. Next, a subset of the designs is randomly selected, with higher-fitness designs having a higher probability of being selected. From this subset of "parent" designs, new "children" designs are generated with combinations of the parents as well as random mutations. The children designs are then evaluated by the cost function to obtain their fitnesses. In principle, subsequent iterations or generations have a higher probability of containing high-fitness designs since more fit individuals from each generation are used to create the next. This process of reproduction (selecting the parents), crossover (choosing traits from each parent), and mutation (randomly altering some traits in the children) is repeated until some stopping criteria is met - typically after some number of

generations, a number of generations with no improvement (stagnation), or a target fitness value has been reached.

The design variables or traits can be represented by binary encoding, real-value encoding, and integer encoding. Binary and integer encoding allow for discrete optimization such as launch vehicle choice (SLS vs. Falcon Heavy vs. Atlas V) or material choice (steel or aluminum) where there cannot be an intermediate value between two options. Real-value encoding is useful for continuous design variables such as time-of-flight or V_∞ magnitude, although binary and integer encoding can also handle continuous variables by discretizing them into many choices within acceptable bounds.

Reproduction or selection is the process of choosing individuals to pass to the next generation. In general, we want the most fit individuals to continue. However, to enable exploration of the design space (as opposed to exploitation only) we can make selection probabilistic in nature. A common method is called roulette wheel selection or fitness proportionate selection. With roulette wheel selection, an individual's probability of being passed to the next generation is its fraction of the population's total fitness. For a population of size N, the probability p_i of an individual with fitness f_i being selected is given by Equation (2.17). This process can also be visualized in Figure 2.8.

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{2.17}$$

Equation (2.17) is useful when the fitness values are positive and the objective is to maximize the fitness. Another approach to the optimization problem is to have negative fitness values which represent a cost or error that should be driven to zero. In this case we need to modify the method by which the probabilities are selected. Two options include taking the inverse of f_i, shown by Equation (2.18), and taking the exponential, shown in Equation (2.19).

$$p_i = \frac{\frac{1}{f_i}}{\sum_{j=1}^{N} f_j} \tag{2.18}$$

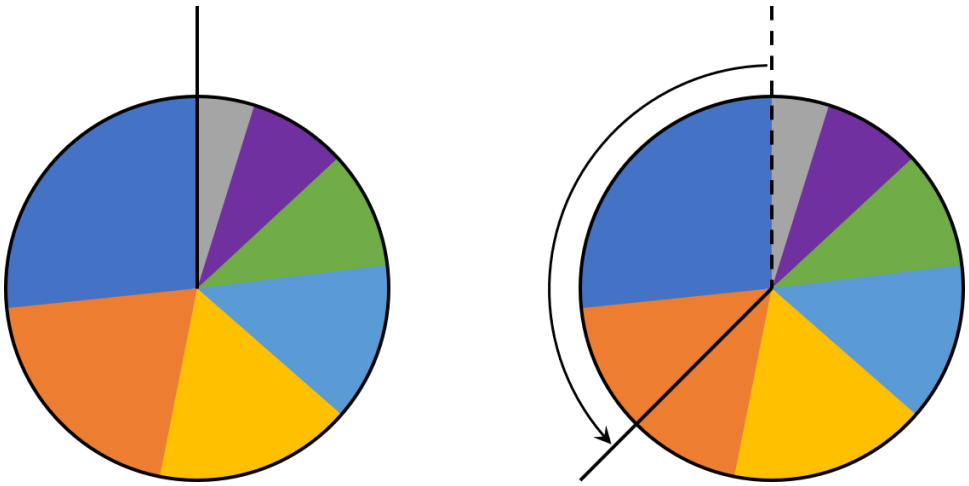

Figure 2.8. A visual representation of roulette wheel selection, where the area of each section represents its probability of being selected, and is proportionate to its fitness.

$$p_i = \frac{e^{f_i}}{\sum_{j=1}^{N} f_j} \tag{2.19}$$

A comparison of these functions is shown in Figure 2.9. Here we see that the inverse greatly favors the high performers, with a very small chance of selecting any individuals towards the low end. The exponential provides a more balanced distribution that still favors the high performers but allows for a non-negligible chance of having lower fitness individuals being chosen. It can be advantageous to enable lower fitness individuals to occasionally pass through because this allows for more exploration of the design space. When only the best individuals are selected, the algorithm will often quickly settle into a local minimum and struggle to get out.

Crossover is the process by which new "children" individuals are created from the "parent" individuals selected during the previous step. In general, crossover aims to mix the features of two individuals. For binary encoding, this can be achieved by selecting a point along the bit string to "cut", and then taking the first part of the bit string from the first parent, and the latter part of the bit string from the second parent. This procedure is shown in Figure 2.10. This method can be extended by choosing multiple points to crossover, se-

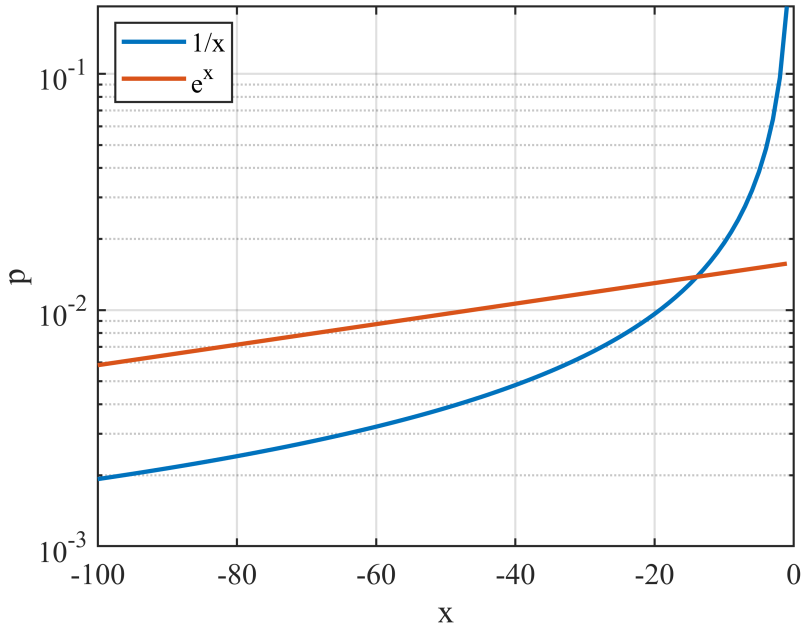

Figure 2.9. Two options for probability distributions during selection are compared. The inverse strongly prefers higher performance individuals, while the exponential has a more balanced profile.

lecting bits randomly from parents, or performing binary operations (AND, OR, ...) to get new individuals.

Figure 2.10. Crossover takes part of each parent's bit string to make new individuals.

For real encoding, either the values that compose each individual can be pulled directly from the parents in a similar manner to binary encoding, or a combination of the values can be used. For example, Equation (2.20) can be used to compute a value of the offspring, where ϵ is a uniform random number in the interval $[0, 1]$.

$$x_3 = \epsilon x_1 + (1 - \epsilon)x_2 \tag{2.20}$$

Once the children have been created, the mutation operator is applied to create small random perturbations in the population. During mutation, a small number of individuals from the population is selected and parts of them are perturbed. For binary encoding, a random part of the bit string is inverted (0 to 1 or 1 to 0). For real encoding, the values can receive a small perturbation drawn from a normal distribution or can multiplied by a random number close to 1.

2.2.1 Application to Trajectory Design and Optimization

Genetic algorithms have been studied previously with applications in trajectory design. Many of the studies include a hybrid optimization approach, where a genetic algorithm changes variables such as the launch and arrival dates, V_∞, launch vehicle choice, and other boundary conditions [22]–[26]. Sometimes other global search algorithms are used in place of a genetic algorithm, although the principle is the same [27]–[30]. In other problem formulations, the genetic algorithm directly optimizes the thrust vectors along the trajectory [31]–[33]. In this dissertation I use a genetic algorithm to train a neural network which acts as a controller for a spacecraft. This has been shown to work previously for low-thrust spacecraft and solar-sail-craft [12], [13], [34]–[36].

2.3 The Missed-Thrust Problem

In the event of an unforeseen problem with a component or subsystem, spacecraft enter a minimally-functional operational mode whose purpose is to prevent further cascading failures, reestablish communication with Earth, and enter a steady-state, power-positive, and thermally stable configuration until the issues have been resolved. This operational mode is often called a “safe mode”, and its occurrence is termed a “safe mode event” or “safing event” [1].

When a spacecraft enters a safe mode, its electric propulsion (EP) engines are typically considered non-essential, and thus no orbit-transfer thrust maneuvers can be performed. Because of this, safe mode events can cause the spacecraft to coast during planned thrust arcs - a missed-thrust event (MTE). Other factors can cause a spacecraft to miss thrust, such

as an improperly aligned thrust vector or having a partial thrust (as opposed to no thrust). However, there are not accessible data to determine the rate at which such other types of missed-thrust events occur, so for the purpose of this dissertation, I assume that the only source of missed thrust comes from safe mode events. This means that, in the context of this work, modeling safe mode events is the same as modeling missed-thrust events.

2.3.1 Modeling Safe Mode Events

Imken et al. published a study of safe mode events in space missions over the past 30 years [1]. The missions included satellites in Low-Earth Orbit (LEO), cis-lunar space, and interplanetary space with missions ranging from a few months to many years. In order to normalize the information, they used two metrics to model the data: time between safe mode events, and recovery duration. Both metrics can be fit using Weibull distributions, a type of distribution commonly used in failure analysis. The parameters they selected are shown in Table 2.1 and the curves are shown in Figures 2.11 and 2.12. Equation (2.21) gives the Weibull probability density function, where k is the shape and λ is the scale.

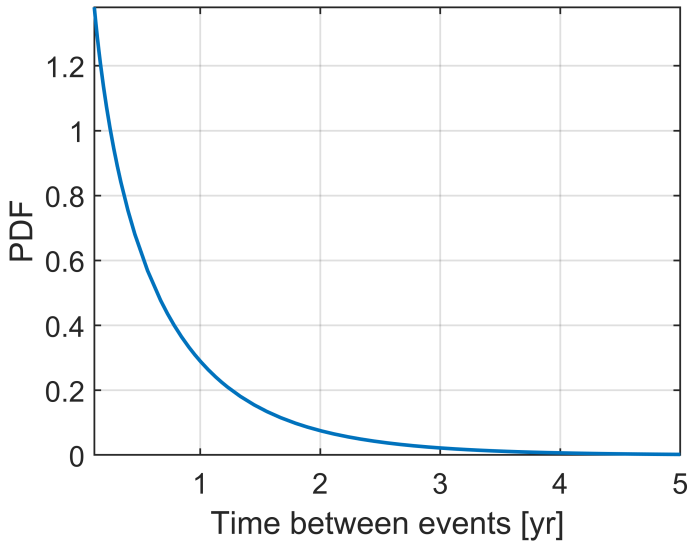

Figure 2.11. Weibull fit for time between the start of missed-thrust events.

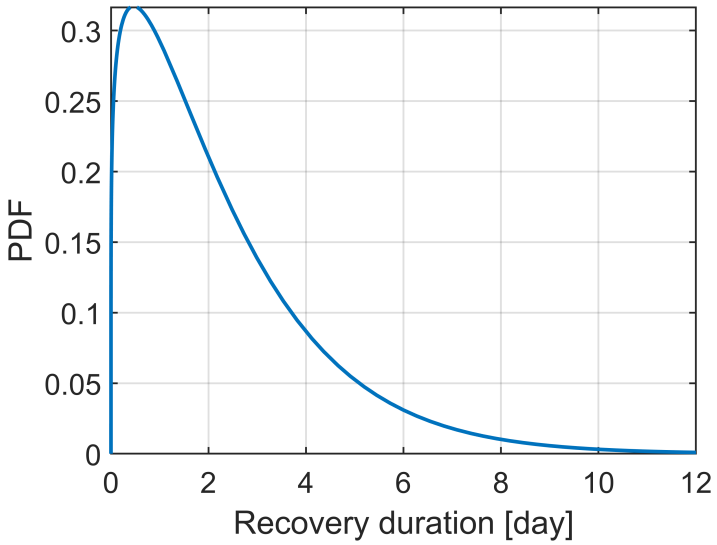

Figure 2.12. Weibull fit for recovery duration, or the time required for the spacecraft to resume operations.

Table 2.1. Weibull parameters for time-between-events and recovery duration.

Metric Type	Scale	Shape
Time-between-events (years)	0.62	0.87
Recovery duration (days)	2.41	1.17

$$f\left(x;k,\lambda\right)=\begin{cases}\frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1}e^{-\left(\frac{x}{\lambda}\right)^{k}} & \text{if } x\geqslant 0\\ 0 & \text{if } x<0\end{cases} \tag{2.21}$$

Imken et al. modeled safe mode events by breaking missions into a few discrete phases. First, there is a discovery delay between when the safe mode occurs and when the ground team actually learns that the event has occurred. The length of this delay is dependent upon the frequency of the communications passes. Next, the recovery duration is composed of the time to investigate the issue, find a solution, and upload the command to exit safe mode. This duration is one of the parameters being modeled. Finally, there is a delay while the spacecraft incrementally resumes functionality before it is fully operational. Multiple safe mode events can occur during a mission, so the elapsed time between events is modeled as well. Imken et al. suggest using a random uniform distribution for the discovery delay

(usually between 1 to 3 days), and using a fixed 12-hr duration for time the spacecraft needs to resume full operational capability.

2.3.2 Quantifying the Effects of Missed Thrust

Early efforts to account for missed thrust simply involved applying a duty cycle, especially before newer techniques emerged. More recently, efforts to quantify the effects of missed thrust involve system margins, namely the propellant margin and time-of-flight margin (or lateness). These metrics are useful when comparing a trajectory with missed thrust to a nominal trajectory. However, since they are each a scalar value, they do not provide the full story. Other strategies to get a more complete picture are to compute the Δv required to recover from a fixed-length outage at each point along a trajectory, and the maximum outage that can be sustained at each point [2], [3], [6], [37], [38]. These analyses provide a better look at the varying sensitivity of sections of trajectory to missed-thrust events. Performing these types of analysis can be computationally expensive since tens to hundreds of intermediate trajectories need to be optimized to create a plot for one base trajectory.

Duty Cycle

Applying a duty cycle can be considered adding margin on the performance of the electric propulsion system. This is computed by Equation (2.22).

$$\text{Duty cycle} = \left(1 - \frac{\text{Useful operating time}}{\text{Thrust arc duration}}\right) * 100\% \tag{2.22}$$

During preliminary mission design, a duty cycle can be used to limit the amount of thrust applied along the trajectory. Applying a duty cycle will prevent the trajectory from becoming fully thrust-saturated, meaning that there is some guarantee of having extra time to recover from a missed-thrust event. However, there is no guarantee that the coast arcs will provide sufficient time or that the spacecraft will have enough extra propellant to successfully recover from a missed-thrust event. As a "quick and dirty" preliminary method, however, this appears to provide a very computationally cheap way to insert some amount

of robustness into the trajectory. Another practical use for duty cycle is to account for the fact that spacecraft periodically need to reorient to contact Earth or adjust their pointing for navigation, science, or other requirements/constraints [39]. A value of 90% is common, although lower values can be used for more conservative estimates [4], [27], [39]–[42].

Propellant and Time-of-Flight Margins

Propellant margin M is the fraction of extra propellant required to recover from missed-thrust events along a trajectory with respect to the nominal amount of propellant needed. This quantity is computed in Equation (2.23), where m_f^* is the final mass of the nominal trajectory (no missed thrust), $\tilde{m}_f$ is the final mass of a trajectory that experiences missed thrust, and m_{prop} is the propellant mass.

$$M = \frac{m_f^* - \tilde{m}_f}{m_{prop}} \tag{2.23}$$

Lateness L is the amount of extra time required to arrive at the destination after a missed-thrust event compared to the nominal time-of-flight. It is computed in Equation (2.24), where $\tilde{T}_f$ is the time-of-flight of the missed-thrust case, and T_f is the time-of-flight of the nominal case.

$$L = \tilde{T}_f - T_f^* \tag{2.24}$$

These two metrics are closely related, because when one margin is decreased, the other typically increases. Laipert and Longuski created a Pareto front using the cost function shown in Equation (2.25), where η is a constant between 0 and 1 [4].

$$\min\ J = \eta M + (1 - \eta)L \tag{2.25}$$

When η is close to 0, L will be prioritized, meaning that the spacecraft will use more propellant to try to arrive on-time. Conversely, when η is close to 1, M will be prioritized, meaning that the spacecraft will conserve extra propellant but will take longer to arrive. The value of η can be tuned by the mission designer to satisfy mission requirements.

When performing a missed-thrust analysis many trajectories will generated, and each trajectory will have a corresponding M and L. Looking into the statistics of these values can provide insight, such as the max, min, average, and the extremal percentiles like 5% and 95%. With these values, mission designers can provide confidence intervals, such as there being a 99% chance that the spacecraft will arrive within an acceptable time with an acceptable amount of propellant.

Computing Margins Along a Trajectory

Some authors have used the following strategies to perform missed-thrust analysis on an existing nominal trajectory to determine propellant and time-of-flight margins along the entire trajectory [6], [37]. First, a forced-coast arc of several days is implemented, and an optimizer then computes the cost to recover from such an outage. The cost could either be extra propellant required to reach the target within an acceptable time, or extra mission duration required to recover given the amount of available propellant. Repeating this analysis along each point of the trajectory reveals which sections are most sensitive to a missed-thrust event. Once sensitive regions are identified, heuristic methods can be applied such as forcing a coast-arc immediately before a gravity assist or rendezvous [43], [44]. Another similar method involves computing the maximum duration outage that can be sustained at each point along the trajectory while still meeting terminal constraints. Both strategies provide a more comprehensive look at the effect of missed thrust on a trajectory, but are relatively computationally intensive because many trajectory optimizations must be performed. Also, this method assumes that only a single outage occurs.

2.4 Force Models

The analysis performed in this dissertation assumes that the spacecraft is acted upon by two forces: gravity and thrust. Unless otherwise noted, the problems use the two-body problem (2BP) model for gravity, which assumes that the gravity affecting a small body (such as a spacecraft) comes from only one source (such as a planet or the sun), and that the mass of the small body is negligible compared to the gravitating mass. Additionally,

the spacecraft is assumed to be a point-mass, and aside from the thrust there are no other forces or moments from gravitational torques, oblate (non-spherical) gravity models, solar radiation pressure, n-body gravity, or any other sources.

2.4.1 Gravity

For many interplanetary trajectories, especially during the preliminary trajectory design phase, a two-body gravity model is an acceptable assumption. The two-body problem (2BP) assumes that one of the two bodies is so much greater than the other that the mass of the second body can be ignored (such as the sun and a spacecraft). The sun and planets can be approximated as spheres, and the gravitational field (above the surface) of a sphere and a point mass are equivalent. Small bodies such as rockets and artificial satellites are negligibly small in terms of size and mass, so they can be approximated as a point mass as well. Using these assumptions of point masses, the inertial frame can then be shifted and fixed at the center of the massive body, providing the following relative vector equation of motion for the motion of the second body around the first [45]:

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^2}\hat{\mathbf{r}} \tag{2.26}$$

Equation (2.26) shows that the gravitational force at a point in space is inversely proportional to the square of the radial distance from that point to the center of the attracting body, and acts in the direction of the attracting body. Such a system provides solutions known as conic sections, and the resulting orbits take the form of circles, ellipses, parabolas, or hyperbolas. For all of the work described in this paper, a heliocentric two-body environment will be used.

2.4.2 Electric Propulsion

Electric propulsion (EP) is a very efficient form of propulsion which in general uses electric and magnetic forces to accelerate propellant to very high exhaust velocities [46]. From Newton's third law, we know that every action has an equal and opposite reaction [47].

Thus, if we want to make a rocket engine as efficient as possible, we want to impart as much kinetic energy as possible to the propellant, which in effect means we want to expel the propellant as fast as possible. EP engines are very good at accelerating small amounts of propellant, which lowers the total amount of propellant required. Since only a small amount of propellant is expelled per unit time, the overall propulsive force is quite low compared to traditional chemical propulsion. A major drawback to EP with respect to chemical propulsion is that EP needs an external source of power. Conversely, chemical propulsion uses the chemical potential energy contained within the propellant, and no external source (aside from something to ignite it initially) is required to maintain a propulsive force. Two general type of EP are solar electric propulsion (SEP), which converts energy from light emitted by the sun into electric energy, and nuclear electric propulsion (NEP), which converts thermal energy from a nuclear power source into electric energy. SEP engines are less complex, do not require radioactive material, and have been used in space before, whereas NEP engines are still being researched and developed. However, NEP engines offer the potential for much greater power generation than SEP which translates to both higher efficiency and higher thrust. Additionally, their power generation capacity is not dependent on distance from the sun, which falls off approximately according to the inverse square law for SEP.

To model the thrust from a EP engine, we add the acceleration (or control) vector $\bar{\mathbf{u}}$ to the right-hand-side of Equation (2.26). We must also account for the change in mass of the spacecraft as it expends propellant. Equation (2.27) shows the augmented acceleration EOM, and Equation (2.28) shows the rate of change of mass. T is the thrust magnitude, and depends on the available power P_a, and time t. P_a is a function of heliocentric distance r (for SEP only) and t. Mass flow rate $\dot{m}$ depends on the thrust magnitude, Earth-sea-level acceleration g_0, and specific impulse I_{sp}.

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^2}\hat{\mathbf{r}} + \frac{T\left(P_a\left(r,t\right),t\right)}{m}\bar{\mathbf{u}} \tag{2.27}$$

$$\dot{m} = -\frac{T\left(P_a\left(r,t\right),t\right)}{g_0 I_{sp}\left(P_a\left(r,t\right)\right)} \tag{2.28}$$

In preliminary trajectory design, P_a can be assumed to be constant when the range of r does not greatly vary over the trajectory, and engine performance degradation can be ignored when missions are not more than a few years. If P_a is constant and engine degradation is neglected, then T and I_{sp} are constant.

2.5 Indirect Optimal Control

The optimal control problem is defined by minimizing the performance index given by the Bolza form of the cost functional in Equation (2.29).

$$J = \phi[\mathbf{x}(t_f), t_f] + \int_0^{t_f} L[\mathbf{x}(t), \mathbf{u}(t), t]dt \tag{2.29}$$

Here, the scalar function ϕ represents a terminal cost and the function inside the integral represents the cost accumulated across the entire trajectory. For a trajectory optimization problem, Equation (2.29) is subject to the system dynamics which can represented as differential constraints in Equation (2.30), as well as any additional terminal constraints as in Equation (2.31).

$$\dot{\mathbf{x}}(t) = f[\mathbf{x}(t), \mathbf{u}(t), t] \tag{2.30}$$

$$\boldsymbol{\Psi}(t_f, \mathbf{x}_f) = \mathbf{0} \tag{2.31}$$

With the system dynamics specified, the problem becomes a matter of finding the optimal control $\mathbf{u}(t)$ along the trajectory for $t \in [0, t_f]$. To do this we use calculus of variations and the Euler-Lagrange theorem, which states that if the control $\mathbf{u}$ is optimal, then there exist a time-varying vector $\boldsymbol{\lambda}$ and a constant vector $\boldsymbol{\nu}$ of Lagrange multipliers which define a set of necessary conditions as well as the transversality condition [48]. The Lagrange multipliers in $\boldsymbol{\lambda}$ are called costates because each value corresponds to one of the state variables. The Hamiltonian H is defined according to Equation (2.32), and the augmented terminal cost function Φ is defined in Equation (2.33).

$$H(t, \mathbf{x}, \mathbf{u}, \boldsymbol{\lambda}) \equiv L(t, \mathbf{x}, \mathbf{u}) + \boldsymbol{\lambda}^T \mathbf{f}(t, \mathbf{x}, \mathbf{u}) \tag{2.32}$$

$$\Phi(t_f, \mathbf{x}_f) \equiv \phi(t_f, \mathbf{x}_f) + \boldsymbol{\nu}^T \boldsymbol{\Psi}(t_f, \mathbf{x}_f) \tag{2.33}$$

The necessary conditions that we gain from the Euler-Lagrange theorem are defined in Equation (2.34).

$$\begin{aligned} \dot{\boldsymbol{\lambda}}^T &= -\frac{\partial H}{\partial \mathbf{x}} = -H_{\mathbf{x}} \\ \boldsymbol{\lambda}^T(t_f) &= \frac{\partial \Phi}{\partial \mathbf{x}_f} \\ H_{\mathbf{u}} &= \mathbf{0}^T \end{aligned} \tag{2.34}$$

However, the third equation in Equation (2.34) only holds if the control is unbounded. If the control is bounded, then we must apply Pontryagin's Minimum Principle as shown in Equation (2.35). Generally speaking, the Minimum Principle states that the optimal control $\mathbf{u}^*(t)$ is that which minimizes the value of the Hamiltonian while also reaching the optimal state $\mathbf{x}^*(t)$.

$$H\left[t, \mathbf{x}^*(t), \mathbf{u}^*(t), \boldsymbol{\lambda}(t)\right] \leqslant H\left[t, \mathbf{x}^*(t), \mathbf{u}(t), \boldsymbol{\lambda}(t)\right] \tag{2.35}$$

The transversality condition is defined in Equation (2.36) in both its algebraic and differential forms. The algebraic form leads to the "adjoined" method, and the differential form leads to the "un-adjoined" method.

$$\begin{aligned} \Omega(t_f, \mathbf{x}_f, \mathbf{u}_f) &\equiv L_f + \frac{d\Phi}{dt_f} = 0 \\ H_f dt_f - \boldsymbol{\lambda}_f^T d\mathbf{x}_f + d\phi &= 0 \end{aligned} \tag{2.36}$$

At this point we can simplify the results of applying the Euler-Lagrange theorem by stating that the following conditions in Equation (2.37) must be true, following the un-

adjoined methodology. Additionally, the problem is subject to the differential form of the transversality condition in Equation (2.36), to the Minimum Principle in Equation (2.35), and to the boundary conditions in Equation (2.38).

$$\begin{aligned} \dot{\mathbf{x}} &= H_{\boldsymbol{\lambda}}^T \\ \dot{\boldsymbol{\lambda}} &= -H_{\mathbf{x}}^T \end{aligned} \tag{2.37}$$

$$\begin{aligned} \mathbf{x}(t_0) &= \mathbf{x}_0 \\ \boldsymbol{\Psi}(\mathbf{x}_f, t_f) &= \mathbf{0} \end{aligned} \tag{2.38}$$

Finally, to solve the optimal control problem, we must find the set of Lagrange multipliers at the initial time $\boldsymbol{\lambda}_0 = \boldsymbol{\lambda}(t_0)$ that satisfy the above conditions. This process can be performed by a variety of numerical solvers such as MATLAB's `bvp4c` or `fsolve`, or Python's `scipy.integrate.solve_bvp` [49]–[51]. I use `fsolve` for any indirect optimization work in this dissertation. Once the initial costates are found (or the costates at any point in time), the entire problem is solved.

2.6 Coordinate Systems

One of the biggest drawbacks to indirect optimization is the sensitivity to the initial guess of costates. If the guessed values of the costates are too far away from their optimal values, then the solver will fail to converge. The distance away from the optimal values that the initial guesses can be and still converge is called the radius of convergence. In general, a large radius of convergence is desirable because a low accuracy guess is sufficient for the problem to converge. Thus, it is beneficial to modify the problem in any reasonable way we can to increase the radius of convergence.

One such way is to change the coordinate system in which we represent the state variables. The intuitive first choice of coordinates for most people would be the Cartesian coordinate system, in which there are 3 orthogonal position coordinates x, y, z and 3 corresponding

velocity components v_x, v_y, v_z. The equations of motion of a spacecraft with mass m subject to the gravitational force of one body with gravitational parameter μ, as well as thrust force from propulsion $\mathbf{T}$ are shown in Equation (2.39).

$$\begin{aligned}
\dot{x} &= v_x \\
\dot{y} &= v_y \\
\dot{z} &= v_z \\
\dot{v}_x &= -\frac{\mu x}{r^2} + \frac{T_x}{m} \\
\dot{v}_y &= -\frac{\mu y}{r^2} + \frac{T_y}{m} \\
\dot{v}_z &= -\frac{\mu z}{r^2} + \frac{T_z}{m}
\end{aligned} \tag{2.39}$$

While this coordinate system is easy to visualize, all six of its state variables are considered "fast". One way to describe this would be look at the change in values when a spacecraft is one side of the planet versus the other while in a circular orbit. On one side, the position variables might all be positive with a magnitude of the spacecraft's radial distance, and the velocity variables might all be negative with a magnitude of the spacecraft's tangential velocity. However, on the other side of the orbit, the signs of these values have all flipped. This means across the orbit each of the six values has on average changed by roughly twice their current value (from -100 to +100 is a change of +200). Having six fast state variables has been shown to lead to fairly small radii of convergence.

An alternative choice of coordinate system is the classical orbital elements (COE). This coordinate system is composed of the semi-major axis a, the orbital eccentricity e, the orbital inclination i, the longitude of the ascending node Ω, the argument of periapsis ω, and the true anomaly f.

Finally, an even better choice of coordinate system that has five slow variables and one fast variable is Modified Equinoctial Elements (MEE), sometimes called Modified Equinoctial Orbital Elements. This coordinate system exhibits well-behaved numerical properties

including a large radius of convergence on many problems [52]–[54]. The MEE state variables are listed in Equation (2.40) in terms of COE.

$$\begin{aligned} p &= a(1-e^2) \\ f &= e\cos(\omega+\Omega) \\ g &= e\sin(\omega+\Omega) \\ h &= \tan\left(\frac{i}{2}\right)\cos\Omega \\ k &= \tan\left(\frac{i}{2}\right)\sin\Omega \\ L &= \Omega+\omega+f \end{aligned} \tag{2.40}$$

Using the MEE coordinate system, we can rewrite the EOMs from Equation (2.39) as Equation (2.41). The thrust vector $\mathbf{T}$ is now represented in the local-vertical, local-horizontal (LVLH) frame which has components in the radial (r), tangential (θ), and normal (n) directions. When there is no thrust present, the only nonzero derivative is the true longitude L, which is very efficient to numerically integrate.

$$\begin{aligned} \dot{p} &= \frac{2p}{w}\sqrt{\frac{p}{\mu}}T_\theta \\ \dot{f} &= \sqrt{\frac{p}{\mu}}\left\{T_r\sin L + \left[(w+1)\cos L + f\right]\frac{T_\theta}{w} - (h\sin L - k\cos L)\frac{gT_n}{w}\right\} \\ \dot{g} &= \sqrt{\frac{p}{\mu}}\left\{-T_r\cos L + \left[(w+1)\sin L + g\right]\frac{T_\theta}{w} + (h\sin L - k\cos L)\frac{gT_n}{w}\right\} \\ \dot{h} &= \sqrt{\frac{p}{\mu}}\frac{s^2T_n}{2w}\cos L \\ \dot{k} &= \sqrt{\frac{p}{\mu}}\frac{s^2T_n}{2w}\sin L \\ \dot{L} &= \sqrt{\mu p}\left(\frac{w}{p}\right)^2 + \frac{1}{w}\sqrt{\frac{p}{\mu}}(h\sin L - k\cos L)T_n \end{aligned} \tag{2.41}$$

where

$$
\begin{aligned}
\alpha^2 &= h^2 - k^2 \\
s^2 &= 1 + h^2 + k^2 \\
r &= p/w \\
w &= 1 + f\cos L + g\sin L
\end{aligned}
\tag{2.42}
$$

2.7 Equations of Motion

The system state $\mathbf{x}_{MEE}$ and its derivatives $\dot{\mathbf{x}}_{MEE}$ are in the form given in Equation (2.43), with the state being represented in the MEE coordinate system. To distinguish these six variables describing the position and velocity of the spacecraft from the full 7D state that includes mass m, I use the subscript MEE for these variables in particular.

$$
\begin{aligned}
\mathbf{x}_{MEE} &= \begin{bmatrix} p & f & g & h & k & L \end{bmatrix}^T \\
\dot{\mathbf{x}}_{MEE} &= \begin{bmatrix} \dot{p} & \dot{f} & \dot{g} & \dot{h} & \dot{k} & \dot{L} \end{bmatrix}^T
\end{aligned}
\tag{2.43}
$$

The EOMs for the system can be written as shown Equation (2.44), where $\mathbf{u}$ is the control vector represented in the LVLH frame. The EOM for the change in mass is given by Equation (2.45). In these equations, μ is the gravitational parameter of the central body, T_{max} is the magnitude of the maximum thrust force that can be applied, c is the exhaust velocity of the engine, and δ is the throttle.

$$
\dot{\mathbf{x}}_{MEE} = A\mathbf{x}_{MEE} + B\mathbf{u} \tag{2.44}
$$

where

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sqrt{\mu p}\left(\frac{w}{p}\right)^2 \end{bmatrix}$$

$$B = \sqrt{\frac{p}{\mu}} \begin{bmatrix} 0 & \frac{2p}{w} & 0 \\ \sin(L) & \frac{1}{w}\left[(w+1)\cos(L)+f\right] & -\frac{g}{w}\left[h\sin(L)-k\cos(L)\right] \\ -\cos(L) & \frac{1}{w}\left[(w+1)\sin(L)+g\right] & \frac{f}{w}\left[h\sin(L)-k\cos(L)\right] \\ 0 & 0 & \frac{s^2\cos(L)}{2w} \\ 0 & 0 & \frac{s^2\sin(L)}{2w} \\ 0 & 0 & \frac{1}{w}\left[h\sin(L)-k\cos(L)\right] \end{bmatrix}$$

$$\mathbf{u} = T_{max}\delta\hat{\mathbf{u}}$$

$$\dot{m} = -\frac{T_{max}}{c}\delta \tag{2.45}$$

3. TRAJECTORY OPTIMIZATION USING NEUROEVOLUTION

In the previous chapter, I explored the ability of a neural network trained using supervised learning to control a spacecraft. In this chapter, I investigate a different training method called neuroevolution. First, I explore the efficacy of neuroevolution on a FFNN and a RNN to determine if providing a feedback loop improves performance. Next I use a method called NeuroEvolution of Augmenting Topologies (NEAT) to see if I can get further performance enhancements.

3.1 Evolutionary Neurocontrol

3.1.1 Introduction

Some previous work has used neuroevolution to train a controller for the spacecraft, and aptly called the work an evolutionary neurocontroller. In this work, a neural network's weights are set using a genetic algorithm with the objective function to provide an optimal spacecraft steering law [10], [11], [55]. This technique of evolutionary neurocontrol stems from previous work in genetic reinforcement learning, which is the use of genetic algorithms to train neural networks for the solution of reinforcement learning problems[56]. The current work aims to explore and expand upon previous work by investigating the advantages and disadvantages of using an RNN as the controller instead of a FFNN.

Recurrent neural networks are networks that include a feedback connection such that at least one hidden layer receives its previous state as an input to itself. This feedback connection allows the RNN to have a "memory" of past events which in turn allows for time-dependent relations to be represented. Additionally, most efforts in neuroevolution use a genetic algorithm as the optimizer, but other global search algorithms may show improved convergence. For this reason, particle swarm is also tested as a method for training the networks [57]–[59].

3.1.2 Training

While there are many architectures of RNNs such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), we felt that for the purpose of time-series prediction of a dynamical system a single layer network with one time delay would be sufficient [60], [61]. Since an RNN depends on its internal state at the previous time step, the backpropagation algorithm must account for all previous time steps when calculating the weight updates. To account for this fact, the RNN can be "unraveled" back through time and trained like an a multi-layer FNN. While there are many algorithms available to perform backprogation, the default method used by MATLAB's `train` function is the Levenberg-Marquardt algorithm [62].

In the framework of neuroevolution, the process of supervised learning is no longer required. Instead, a genetic algorithm (or other evolutionary algorithm) takes the weights and biases of the neural network as its optimization variables. This type of problem is similar to reinforcement learning, in which an actor takes an action that moves it from its current state to a future state, and this future state is given a reward based on the new state's perceived "fitness". However, a spacecraft trajectory is often evaluated by its final values, such as time of flight or final mass delivered, and these properties cannot be reliably estimated in the middle of a trajectory. Since these values of interest are not known until the end of the trajectory, the entire trajectory must be integrated to determine the fitness of a given neural network. Thus, for a given set of initial conditions, a neural network uniquely specifies a trajectory.

Let us look at how the computation of an RNN differs from that of an MLP. If we combine Equation (2.3) and Equation (2.6) we get Equation (3.1), the activation from a sigmoid node in an MLP.

$$\sigma(\mathbf{x}) = \frac{1}{1 + e^{-(W\mathbf{x}+\mathbf{b})}} \tag{3.1}$$

$$\sigma_t(\mathbf{x}) = \frac{1}{1 + e^{-(W\mathbf{x}_t+U\mathbf{x}_{t-1}+\mathbf{b})}} \tag{3.2}$$

When feedback connections are introduced, the output of at time t is of the form of Equation (3.2), where U is the matrix of weights connecting the current layer back to itself, and the subscripts represent the time step.

$$n_{var,FNN} = (n_{in})(n_{hid}) + (n_{hid})(n_{out}) + n_{hid} + n_{out} \tag{3.3}$$

$$n_{var,RNN} = (n_{in})(n_{hid}) + (n_{hid})(n_{hid}) + (n_{hid})(n_{out}) + n_{hid} + n_{out} \tag{3.4}$$

As shown in Equation (3.1) and Equation (3.2), the only difference between the calculation of a node in an FNN and RNN is the inclusion of the matrix U which specifies the weights from the hidden nodes back to themselves. The number of variables for which the optimizer must solve for in an FNN and an RNN are shown in Equation (3.3) and Equation (3.4), respectively. Here, W_1 is the matrix of weights from the input layer to the hidden layer, which has dimension $n_{in} \times n_{hid}$; U is the matrix of weights from the hidden layer back to the hidden layer, which has dimension $n_{hid} \times n_{hid}$; W_2 is the matrix of weights from the hidden layer to the output layer, which has dimension $n_{hid} \times n_{out}$; $\mathbf{b}_h$ is the vector of biases added to the hidden layer, which has dimension n_{hid}; and $\mathbf{b}_o$ is the vector of biases added to the output layer, which has dimension n_{out}.

3.1.3 Method of Solution

The trajectory optimizer consists of the two major parts: the numerical integrator, and the optimizer. A high-level flow chart of the algorithm logic is shown in Figure 3.1. The integrator computes the position and velocity of the spacecraft subject to gravity from any number of gravitating bodies as well as the thrust. Analytical ephemerides for the planets are used to increase run time [63]. The thrust vector (direction and magnitude) at each time step is calculated by querying the neural network with the spacecraft's current state vector, any gravity-assist body state vectors, the target body state vector, and the current spacecraft mass as inputs. A coplanar model is used, so each state vector has four total values from the position and velocity vectors, bringing the total number of inputs to thirteen

(assuming one gravity-assist body). When using an FNN, these inputs are independent of time (explicitly), so a variable step-size integrator is acceptable. However, when using an RNN, its inputs also include its previous internal state. Therefore, either a fixed step-size integrator must be used, or a variable step-size integrator must be used in a way such that the RNN can be used at fixed times throughout the integration.

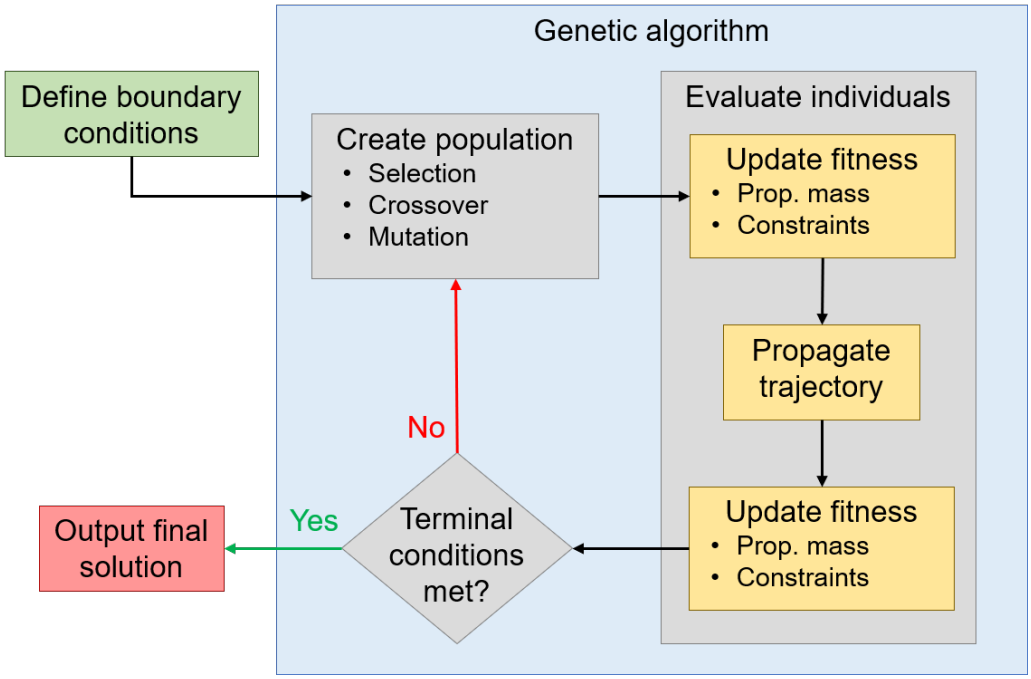

Figure 3.1. The algorithm receives boundary conditions from the user, specifiying the target body, bounds on the time of flight and launch date, and any gravity-assist bodies to target. These boundary conditions are passed to the optimizer, which evaluates individuals based on the final objective function.

Time was initially used as an input to the network, but when training the neural network, the weights were usually set such that values other than time were "zeroed out" and the network simply became a function of time. While this may be useful in specific circumstances, the network does not actually learn anything about the relation between the dynamics of the solar system and the optimal thrust vector. We believe that while using time may be useful if the problem is posed correctly, it was unnecessary for the present problem.

The optimizer takes the weights and biases of the neural network as its optimization variables such that the design vector or an "individual" uniquely specifies a neural network.

In turn, since the weights and biases of a network are constant for an entire trajectory, the objective function for the resulting trajectory can then be used to assign a fitness to the neural network. The optimizer therefore works to minimize the objective function (final mass delivered or time of flight) by modifying the neural network and calculating the resulting trajectory. The evolutionary search algorithms used are a genetic algorithm (GA) whose operators are tournament selection and uniform crossover with empirically-derived meta-parameters, and the MATLAB implementation of particle swarm [64]. Since these algorithms can take any number of inputs and since a trajectory is specified by its neural network, additional parameters relevant to the trajectory optimization problem such as launch date and time of flight can also be added to an individual.

The only inputs required by the mission designer are the target body and any gravity-assist bodies (all trajectories are assumed to start at Earth with the sun as the central body), and the range of launch dates and times of flight to consider. After this the optimizer creates an initial population of random individuals, and proceeds until termination criteria are met. Termination criteria include bit string affinity (how closely the individuals in a population match each other) and a maximum number of generations.

Since a GA only sees the objective function and does not inherently handle constraints aside from bounds on the design variables, constraint violations must be appended to the objective function. Constraints imposed upon the trajectory include terminal constraints for position and velocity vectors, closest approach to the sun (so the spacecraft does not get too hot), and any desired features such as proximity to a desired gravity assist body. Permitting proximity to a specified planetary body as a constraint allows the mission designer to specify bodies the spacecraft should fly by. While only the final position and velocity vectors are required, additional or dynamic constraints can be added to help "lead" the GA to find solutions more easily. For this reason, we changed the terminal constraints to include angular momentum magnitude, radius, speed, eccentricity, and true anomaly.

The present analysis is structured with the assumption that the spacecraft can thrust with some percentage of a fixed value along the entire trajectory, regardless of distance from the sun. One such propulsion method that adheres to this assumption and that also offers high performance is nuclear electric propulsion (NEP). The propulsion system used

here has a characteristic acceleration a_0 of $0.11\,mm/s^2$ and a specific impulse I_{sp} of $6000\,s$. These values are within the ranges used by Yam investigating similar problems [65]. The maximum allowable thrust is calculated from the characteristic acceleration and the initial mass (specified by the mission designer) by Equation (3.5). The mass flow rate is then determined using the relation in Equation (3.6), where the magnitude of $\mathbf{T}$ is determined by the neural network.

$$a_0 = \frac{|\mathbf{T}_{max}|}{m_0} \tag{3.5}$$

$$\dot{m} = -\frac{|\mathbf{T}|}{g_0 I_{sp}} \tag{3.6}$$

3.1.4 Results

Earth-to-Mars Transfer

To start, the optimizer was created using the traditional approach of neuroevolution, meaning that an FNN and GA were used. Using the spacecraft position, velocity, and mass as inputs with only four hidden nodes, the simple planar circular-to-circular orbit transfer with a fixed time of flight of 400 days shown in Figure 3.2 was found. Here the spacecraft had an initial mass of 10,000 kg, and could vary its launch date with respect to Mars by ± 2 years (since they are circular and coplanar there are no additional factors concerning eccentricity of the orbits). The arrows in the figure represent the direction and magnitude (relative to the other arrows) of thrust vector at each point. The thrust vector is held constant and is numerically integrated between points along the trajectory. Similar problems that include one or two revolutions also are not challenging for this implementation, and the solution typically takes on the order of 15-30 minutes to find depending on complexity.

Next, the FNN was replaced with an RNN and the same problem was attempted using the same GA structure. For the same inputs and number of hidden nodes, the RNN was able converge upon a similar solution, although the exact spacing of the thrust vectors was slightly different since a fixed time step integrator was used instead of a variable time step

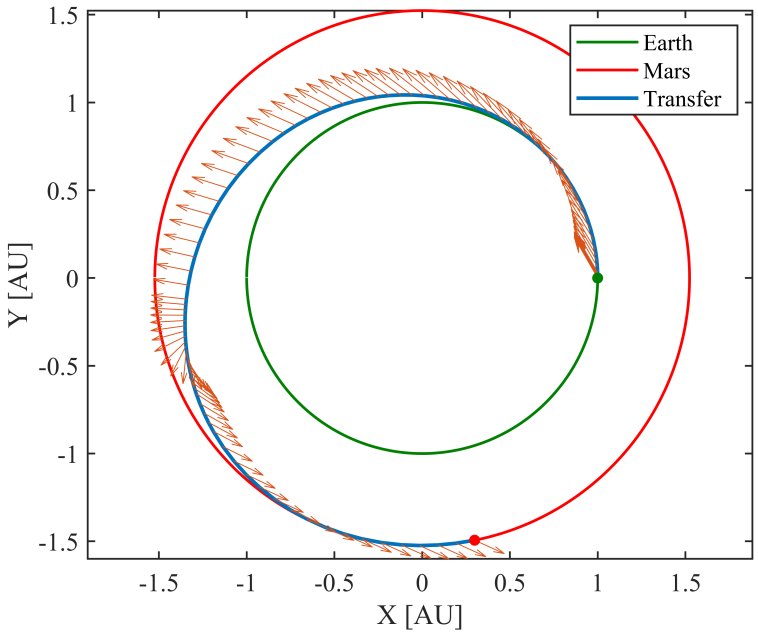

Figure 3.2. Example trajectory depicting a planar orbital tranfer from an Earth-radius to a Mars-radius circular orbit with a 400 day time of flight.

integrator. As per Equation (3.3) and (3.4), changing from the FNN to RNN increases the number of weights and biases from 30 to 46. This increase in the number of optimization variables caused an increase in the average run time at around 30-45 minutes. Thus, for around a 50 percent increase in number of optimization variables, the runtime increased by a similar proportion.

Gravity-Assist Low-Thrust Trajectories

Since the concept was successfully demonstrated on relatively simple cases, the optimizer was tested on more difficult trajectories that would require or be improved by a gravity assist. The first test case considered was an Earth-Jupiter-Uranus trajectory using the spacecraft described previously with an initial mass of 20,000 kg. The launch date was bounded between August 01, 2018 $\pm$6 years, and the time of flight was bounded between 9 years and 18 years. The terminal constraints included zero hyperbolic excess velocity at launch from Earth and

arrival at Uranus. Proximity to Jupiter was added as an additional parameter to minimize, thus forcing the optimizer to perform the gravity assist.

The increased complexity of the problem made it much harder for the optimizer to find solutions that result in trajectories that are close to feasible, meaning much more time was spent finding a neural network that could satisfy all the constraints as opposed to optimizing the trajectory. In fact, no neural network produced a trajectory that satisfied every constraint. The constraints that were not satisfied and the types of trajectories found were not necessarily similar between runs, either. Sometimes the solution would successfully perform a close flyby of Jupiter but would not make it to Uranus or would overshoot Uranus. Other times it would get into a near-circular orbit in a different phase near Uranus' orbit, and sometimes the solution would not be close at all.

Run time became an issue, with a single run of the optimizer taking anywhere from 5 to 15 hours before terminating, with the termination due to the maximum number of generations reached (as opposed to convergence on an optimal solution). The GA was switched out for a particle swarm algorithm using MATLAB's builtin `particleswarm` function, with a similar lack of convergence.

Despite not finding any valid solutions, we believe that this method *could* find solutions, and that we were simply unable to in this study. With further tuning of the model, we would expect to find a solution.

3.2 NeuroEvolution of Augmenting Topologies

One of the problems with the neuroevolution strategy used above is that usually either the network is so small that it is difficult to learn complex relationships, or it is so large that the number of optimization variables is too high and the algorithm cannot effectively learn in a reasonable amount of time. To address, another method has been developed called NeuroEvolution of Augmenting Topologies (NEAT). NEAT encodes the structure of a neural network in such a way that the genetic operators crossover and reproduction can be applied even when two networks have different structures. By enabling these genetic operators, NEAT can change the structure of the network over the course of the optimization. The

variable structure can be exploited by starting with a small neural network that has relatively few parameters, and then gradually adding additional nodes or connections as they help the network. Similarly, nodes or connections that do not help can be removed.

3.2.1 Problem Description

Boundary Conditions

To test this algorithm, I start with a transfer from Earth to Mars. To simplify the problem, I assume that their orbits are coplanar so that the problem is two-dimensional. There is no fundamental difference between a 2D and a 3D formulation of this problem with regard to the neural network design aside from a smaller input and output dimensionality. A 2D problem converges faster due to its smaller dimensionality, and is thus a more convenient test problem to start with.

Each trajectory has a fixed time-of-flight of 1000 days (2.74 yr). The spacecraft will have an initial mass of $13,000$ kg, a dry mass of $10,000$ kg, maximum thrust of 1.2 N, and an I_{sp} of 2780 sec. The engine parameters correspond to two HERMeS EP engines, where the available power is assumed to be sufficient to provide the maximum thrust along the entire trajectory ($>$ 25 kW) [66]. The available propellant can provide up to 7.15 km/s of Δv, compared to the minimum-Δv solution from a planar elliptical-to-elliptical Hohmann-like transfer of 5.58 km/s.

Objective Function

Several considerations must be taken into account to assign a fitness value to a neural network controller. Since the trajectory integration is forward-propagation only, the terminal boundary condition will not be met exactly. To incentivize the network to end the integration as close to the target as possible, the difference between the target final state and the actual final state is calculated. Then we multiply by a weighting term, take the square, and then sum each of the state errors to obtain a scalar measure of error. To improve the convergence of the algorithm, we calculate the error in terms of classical orbital elements instead of Cartesian elements. Also, to be able to handle cases with circular orbits (where eccentricity

is zero), we use apoapsis radius r_a and periapsis radius r_p instead of semi-major axis a and eccentricity e. For a 2D problem, we also have argument of periapsis ω and true anomaly f, so that the state vector is $\bar{x} = [a, e, \omega, f]$. The state error is computed by Equation (3.7).

$$\delta\bar{x} = \frac{|\bar{x}_f^* - \bar{x}_f|}{|\bar{x}_f^*|} \tag{3.7}$$

Next, we multiply each component of the state errors by a weighting term, and choose the maximum of the quadratic or linear term as shown in Equation (3.8). The quadratic term dominates when the spacecraft is far away and is useful to drive all terms towards zero initially. The linear term helps to continue to funnel the error to zero as the spacecraft gets closer to the target when the quadratic term becomes very small.

$$J_{\bar{x}} = \sum_{i=1}^{4} \max\left(\alpha_{x,i}^2 \delta x_i^2, \alpha_{x,i} |\delta x_i|\right) \tag{3.8}$$

In some cases, we can choose to add two impulsive maneuvers at the end of the integration to target the terminal boundary condition exactly. To penalize the network from relying on this final correction maneuver, Equation (3.9) computes the extra TOF as a fraction of the original TOF, and then multiplies by a weighting term.

$$J_{TOF} = \alpha_{TOF}\left(\frac{t_f^*}{t_f} - 1\right) \tag{3.9}$$

The objective function that we want to minimize is the propellant mass used. Equation (3.10) computes the fraction of propellant used to propellant available and multiplies by a weighting term.

$$J_m = \alpha_m \left|\frac{m_0 - m_f}{m_0 - m_{dry}}\right| \tag{3.10}$$

Finally, Equation (3.11) adds each of the intermediate costs together to get the total fitness.

$$J = J_{\bar{x}} + J_{TOF} + J_m \tag{3.11}$$

Since the algorithm works by maximizing fitness, but TOF, consumed propellant mass, and state error are all positive quantities which should be minimized, I make the final fitness negative. Thus, the algorithm will drive the fitness as positive as possible, which corresponds to making each term as close to zero as possible.

3.2.2 Algorithm Details

The location and duration of missed-thrust events that occur on a mission are not known beforehand. Therefore, we want to design a controller that can recover from any arbitrary sequences of missed-thrust events that can realistically be expected to occur. By creating a series of test problems which have random missed-thrust events, and by choosing a neural network that performs well on many of these randomized tests, we can find a network that produces trajectories which are more resilient to missed-thrust events.

There are a few ways to approach the design of a neural network for this problem. If the desired initial and final states are known and fixed, then these can be held constant during training. If these boundary states are not known or do not need to be fixed but can be bounded, then during training we can sample randomly from within the bounds and test the network on these varying cases. Furthermore, we can choose to include or omit missed-thrust events during the trajectory propagation, which can cause the network to learn different behaviors. For the case of fixed boundary conditions without missed-thrust events, we only need to propagate a single trajectory, because the result will be same every time. However, if the boundary conditions can change between runs or if missed-thrust events are allowed to occur, then several trajectories should be propagated to get a statistically representative evaluation of the network. The four types of cases that can be tested are:

1. Fixed boundary conditions without missed-thrust events
2. Fixed boundary conditions with missed-thrust events
3. Variable boundary conditions without missed-thrust events
4. Variable boundary conditions with missed-thrust events

Additionally, each type of case can have three progressively more complex tasks:

a. Circular to circular fixed-time orbital transfer (requires a, e)

b. Elliptical to elliptical fixed-time orbital transfer (requires a, e, ω)

c. Elliptical to elliptical fixed-time rendezvous (requires a, e, ω, f)

When the boundary conditions are fixed, the network on needs the current state, the time ratio, and the mass ratio as inputs (6 total). It does not need the target state, because proper training would enable the network to implicitly learn the target state. When the boundary conditions are variable, the target state also needs to be an input to the network (10 total).

Thus, there are 12 cases that can be evaluated to determine the ability of this technique to address the missed-thrust problem. Cases 1a-c provide tests to ensure that the algorithm can successfully optimize a trajectory, while the remaining cases provide measures for how capable the neural network is. Cases 2a-c train a neural network to transfer between fixed points in the presence of missed-thrust events, providing both a potential on-board controller as well as a relatively low sensitivity nominal trajectory. Cases 3 and 4 are similar to 1 and 2, respectively, except that during training the boundary conditions can shift. These pose more difficult problems for the neural network to solve, but can lead to a more robust controller (since it will be exposed to a wider range of states during training) as well more flexibility in the presence of a wide launch and/or arrival window. A diagram of the algorithm logic is shown in Figure 3.3.

3.2.3 Implementation Details

I am using Python 3.6 to implement this algorithm, with the library neat-python as a complete implementation of NEAT, and Boost.Python for a compiled Runge-Kutta-Fehlberg 7-8 integrator written in C++ that provides significant speed improvements compared to the standard Python integrator, scipy.odeint. An input file contains the boundary conditions and defines the problem to be solved. This file is parsed, and converts the problem into one that NEAT can readily solve. The task of evaluating the fitness of every individual in a population in a genetic algorithm is considered "embarrassingly parallel" - that is, each member can be evaluated in parallel with no dependence on the other individuals. Thus, significant run

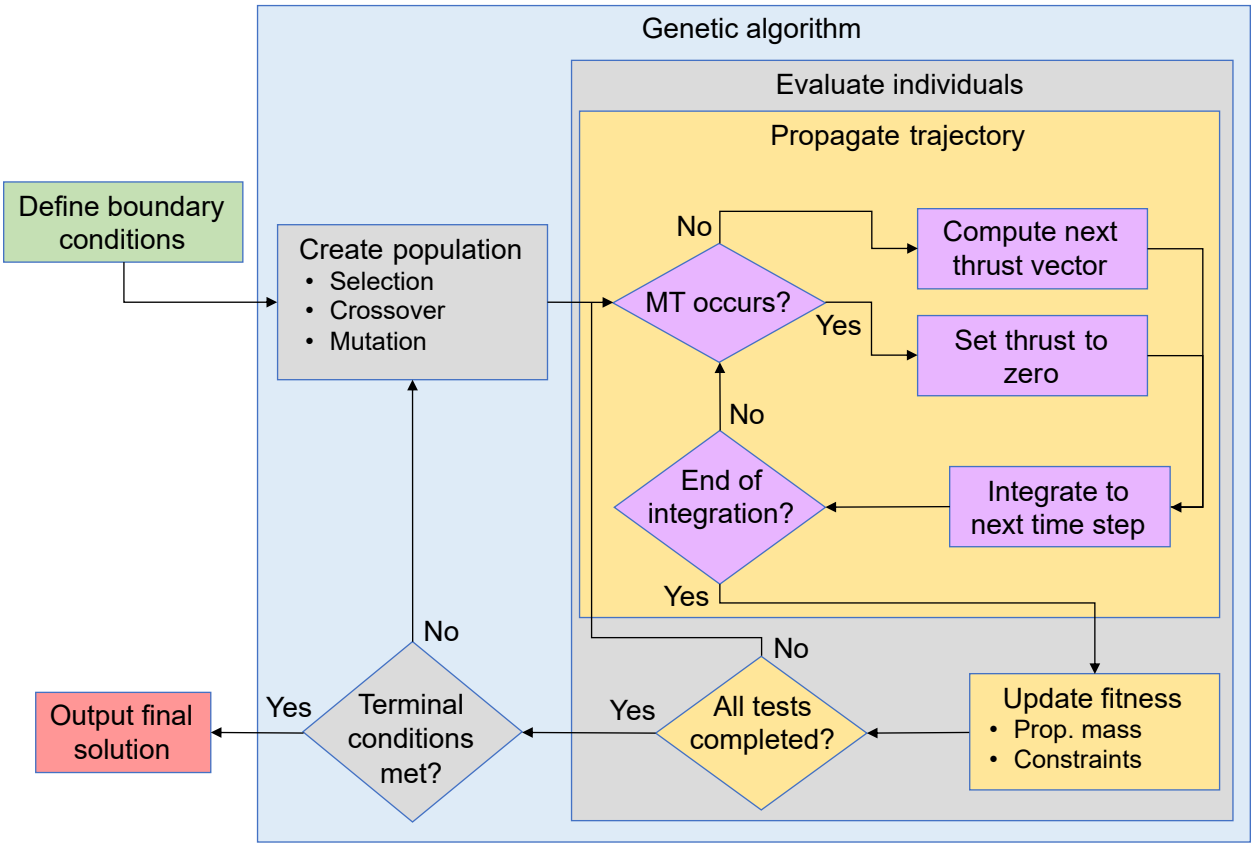

Figure 3.3. Algorithm logical flow-chart. Boundary conditions are given to the GA, which evaluates each network by testing it on several trajectories that each have a series of random missed-thrust events. The GA repeats until stopping conditions are met.

time improvements are gained when many compute threads are available. Purdue's ECN offers several compute servers which are available to graduate students, so I typically send jobs to run on 55 threads (out of 56 on the machine) operating at 2.8 GHz. Using this setup, I can evaluate around 700 trajectories/second with 150 nodes/trajectory or around 150 trajectories/second with 1000 nodes/trajectory.

While it can vary depending on the complexity of the problem, I set NEAT to have a population size of 400 individuals, either 6 inputs (4 for current state, 1 for mass ratio, 1 for time ratio) or 10 inputs (plus 4 for target state), 2 outputs (thrust magnitude and thrust angle), 5 initial hidden nodes, initial connections between nodes with an occurrence rate of 0.8, node and connection addition and deletion mutation rates of 0.01 and 0.005,

weight and bias initial mean and standard deviation of 0 and 0.05, weight and bias mutation rates of 0.05, max allowable stagnation period of 50 generations, an elitism of 2, among many other small hyperparameters. I selected these values after experimenting on several test problems and observing the effects of changing each. The integrator has relative and absolute tolerances of $1e$-9 with an adaptive step size. Trajectories have 200 points at which the control is updated. These points are equally spaced in time, giving 199 possible thrust- or coast-arcs. For a 1000 day TOF, each leg is approximately 5 days.

3.2.4 Missed-Thrust Trajectory Design

The algorithm described by Figure 3.3 was successfully implemented and tested on a progressively more difficult test cases. Figure 3.4 shows the nominal case for an Earth-Mars transfer assuming the boundary conditions described in section 3.2.1, where the launch date is set to September 27, 2019 and the arrival date is June 23, 2022. During training, each neural network was tested on 6 random test cases and its fitness was the average of the cases. In the plot, green indicates thrust, black is coasting, orange denotes direction and magnitude of thrust, magenta is the target state, and blue is the actual final state. Table 3.1 shows the final mass and state error when the trained neural network was tested on 100 random test cases.

Table 3.1. Performance statistics on 100 test cases after training.

	Mean	**St. Dev.**
Final Mass (kg)	10,486	17.23
Position Error (km)	1.480e7	8.742e6
Velocity Error (km/s)	1.809	1.164

The trajectory shown in Figure 3.4 is the trajectory created by the neural network when no missed-thrust events are present. This trajectory can be thought of as the baseline or reference trajectory. When considering which trajectory to use for a mission, this baseline missed-thrust trajectory should be considered instead of the fuel-optimal trajectory. While the fuel-optimal trajectory has the lowest propellant mass when missed thrust is not taken

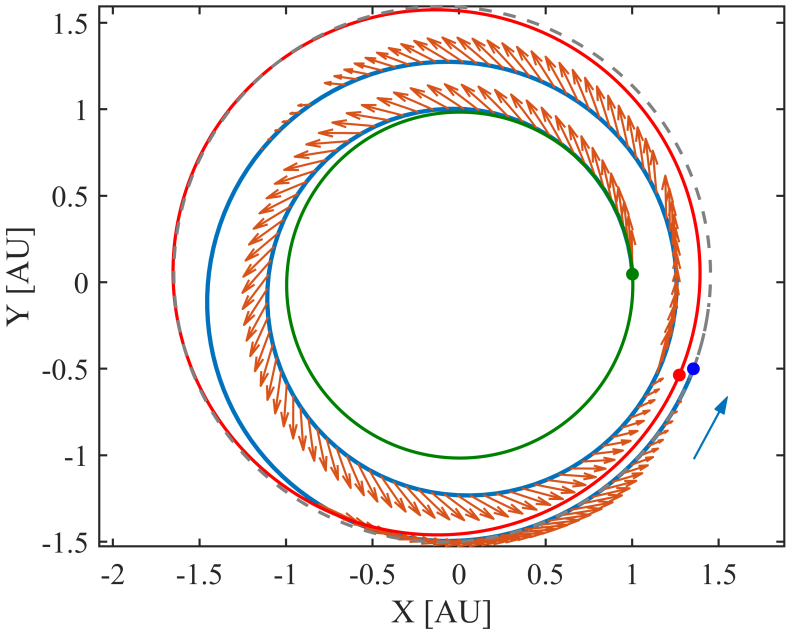

Figure 3.4. Baseline trajectory when trained with missed-thrust events.

into account, the missed-thrust baseline trajectory has the lowest expected value of propellant mass when missed thrust is considered.

4. TRAJECTORY OPTIMIZATION USING SUPERVISED LEARNING

In this chapter I investigate using supervised learning to train a neural network to be able to generate optimal spacecraft trajectories. I use two strategies: predicting the optimal costates at a particular state, and predicting the optimal thrust vector at a particular state. When the network predicts the optimal costates at a given point along the trajectory, these costates specify the entire trajectory from the current state to the final state as discussed in section 2.5. However, integrating a trajectory with costates requires complex derivatives to be analytically calculated ahead of time, and these analytical derivatives can place limits on the types of constraints which can be incorporated into the problem. Also, the radius of convergence of the costates places a soft limit on the complexity of problems that can be effectively solved with this method, because too complex of problems will have an impractically small radius of convergence. To get around these shortcomings associated with costates, the second strategy shown in this chapter is to predict the optimal thrust vector at a given point on the trajectory. Because the output is simply a force vector, it can be easily added to rest of the forces acting on the spacecraft to compute the net resultant force. While I only examine thrust and gravity, other forces could include solar radiation pressure, n-body gravity, non-spherical gravity perturbations, rigid body dynamics, or any other high fidelity force models. A drawback to predicting the thrust vector at a point along the trajectory is that the thrust vector is only useful at that exact point. The thrust vector can be used to integrate the spacecraft state forward for a short period, but then the neural network must predict a new thrust vector. This process must be repeated many times from the beginning to the end of the trajectory, and with each neural network prediction comes the chance for error to be introduced. Therefore we need to be mindful about the possibility of error stacking up over time, forcing the neural network away from the optimal trajectory and into a region for which the neural network was not trained.

4.1 Problem Definition

With supervised learning, we provide a neural network with examples of the desired relationship to be learned, and then try to minimize error between the network's outputs and the desired outputs for a set of inputs. Before jumping in to training a neural network, we must first define the problem to be solved. Since the goal is ultimately to investigate the missed-thrust problem, I use a trajectory defined by Laipert and Longuski on which the authors perform missed-thrust analysis [4]. By using the same problem, I can compare the final results to help determine the effectiveness of a neural-network-based controller.

The trajectory is defined by a spacecraft traveling from Earth to Mars in the heliocentric two-body problem (2BP). The spacecraft has a launch date from Earth of 24 Aug 2024, a 30-day checkout period ending 23 Sep 2024 during which no thrusting can occur, and an arrival date at Mars of 07 Nov 2025. The states for Earth and Mars are determined using NASA Jet Propulsion Laboratory's SPICE ephemeris system at the corresponding dates which provides 3D, elliptical orbits for both bodies [67], [68]. In the paper, the spacecraft has a final mass of 2643 kg, a propellant mass of 655 kg, a launch C_3 of 3.12 km^2/s^2 and a total Δv of 3.97 km/s. The spacecraft is assumed to have two XR-5 Hall thrusters which each have a maximum thrust of 281 mN and I_{sp} of 1850 s at max power (20 kW). Laipert and Longuski model thrust and I_{sp} as functions of available power, which itself is function of distance from the sun. In the current work I assume a constant available maximum thrust (i.e. the spacecraft has sufficient solar panels to produce sufficient power for the thrusters at all points along the trajectory). Figure 4.1 shows a north-pole plot of the nominal case for this trajectory. The arrows represent the thrust vectors along the trajectory. Launch from Earth occurs at (1), the checkout period ends at (2), and arrival at Mars occurs at (3). Figure 4.2 shows the corresponding thrust profile along the trajectory, including the throttle (fraction of maximum thrust used) and the three components of the thrust vector expressed in the local-vertical, local-horizontal (LVLH) frame. This trajectory is composed of two segments with maximum thrust and one segment of zero thrust, with the thrust direction primarily in the tangential direction but with varying levels in the radial and normal directions.

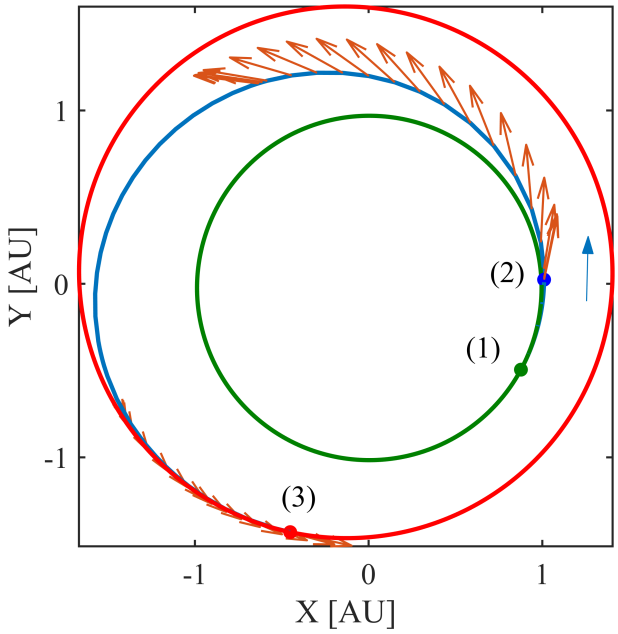

Figure 4.1. The nominal fuel-optimal trajectory with 20kW from Laipert and Longuski [4]. Green is Earth's orbit, blue is the transfer orbit, and red is Mars's orbit. The orange arrows represent the direction and magnitude of the thrust at each location along the transfer.

To allow the neural network to provide reliable predictions to recover from missed-thrust events, the training data must include examples in a range around the nominal trajectory - otherwise, if the spacecraft got off-course, then the network would see inputs that it has never encountered before, and would not be able to provide a useful output. Starting from the nominal trajectory described above, I include a launch window and arrival window of $\pm$ 20 days. Due to the nature of the problem, the launch date can only move forward up to around 7 to 10 days and depends on the arrival date. If it is pushed out farther than that then the problem becomes infeasible and it is not possible for the spacecraft to reach the destination within the given constraints. To simplify the problem, I use a maximum forward launch date offset of 7 days. Thus the actual date range being considered is an arrival window of $\pm$20 days and a launch window bounded between $-$20 days and +7 days.

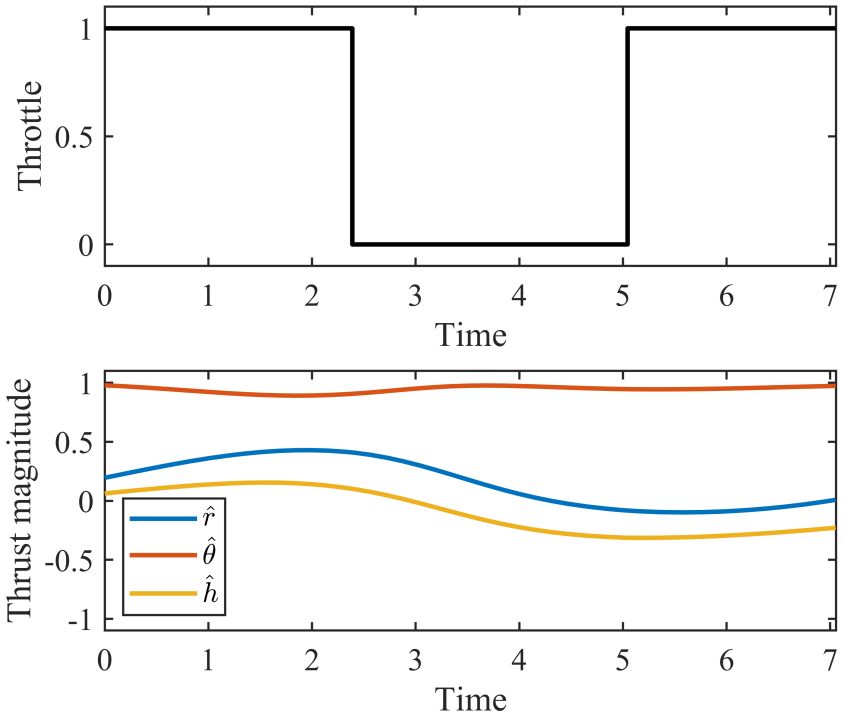

Figure 4.2. The corresponding thrust profile for the nominal trajectory in 4.1.

4.2 Training Data Generation

Before we can train a neural network to guide a spacecraft, we must consider what the purpose of the network should be. There are some elements of the problem that must be determined early on. What does the network need as inputs? What should the outputs be and how should they be interpreted? For this problem, I want the neural network to provide a control law for the spacecraft during trajectory integration. That is, given some formulation of the spacecraft's current state, target state, available propellant, time-to-go, time before the next thrust update, and available thrust or power, the network should provide either the optimal thrust vector or the optimal costates at the current state. Since solving for the optimal control law of a spacecraft is performed by solving a two-point boundary value problem (TPBVP) and this is the step which is desired to be replaced by a neural network, the neural network needs as inputs all of the components of the TPBVP to perform the task well.

In this work I represent the spacecraft state using a 3-dimensional representation in the MEE coordinate system, plus the mass of the spacecraft. Although mass can be removed from the state in certain problem formulations, it is an input to the neural network and so it is needed when propagating the state [69]. Thus, it is included to avoid unnecessarily complicating the math.

A large dataset of optimal input-output pairs is needed to train a neural network to predict either optimal costates or optimal thrust vectors. Since the optimal thrust vector can be obtained from the costates, we only need to worry about collecting costate information as we are building the dataset. To generate a sufficiently large number of pairs in a reasonable amount of time, ironically we need a tool to do what we are trying to replace - a fast and robust method for indirect optimization that can converge on a feasible solution without a particularly accurate initial guess. To do this, I use the algorithm presented in Taheri et al. that uses homotopy (sometimes called continuation) to step from an easier problem with a large radius of convergence to the more sensitive minimum-fuel problem with the familiar "bang-bang" control scheme [53]. In their work, they augment the performance index for a minimum-fuel trajectory given by Equation (4.1) with a "logarithmic barrier" as developed in Bertrand and Epenoy to form the perturbed performance index in Equation (4.2) [70].

$$J = -m(t_f) = \frac{T_{max}}{c} \int_{t_i}^{t_f} \delta_m dt \tag{4.1}$$

$$J = \frac{T_{max}}{c} \int_{t_i}^{t_f} \left\{ \delta_m - \epsilon \left[-\delta_m \log\left(\delta_m\right) - \left(1 - \delta_m\right) \log\left(1 - \delta_m\right) \right] \right\} dt \tag{4.2}$$

When $\epsilon = 1$, the resulting thrust profiles are generally smooth without sharp discontinuities, and have a comparatively large radius of convergence. However, as $\epsilon \to 0$ the problem reduces to the minimum-fuel problem. Once we solve for $\epsilon = 1$, this solution can be used as an initial guess for a problem with a reduced value of ϵ. This process can be repeated until ϵ is sufficiently small. Around 10^{-4} is generally acceptable, although the value is somewhat problem dependent. Near regions where the form of the solution bifurcates, such as going from two thrust arcs to three thrust arcs, a smaller value of ϵ may be desired to maintain accuracy. Decreasing ϵ by a factor of 10 between iterations is close enough to converge from

a previous solution, so going from $\epsilon = 1$ to $\epsilon = 10^{-4}$ requires 5 total optimizations. The motivation for using the homotopy approach to find the solution is that with the augmented logarithmic barrier performance index, a nonlinear solver such as MATLAB's `fsolve` can successfully converge with a less accurate initial guess. A random uniform guess in the range $[0, 1]$ for each costate and $\epsilon = 1$ succeeds around $60 - 65\%$ of the time in my experience. Figure 4.3 shows the throttle profile of the fuel-optimal trajectory for the problem described in Section 4.1 with values of ϵ between 1 and 10^{-4}.

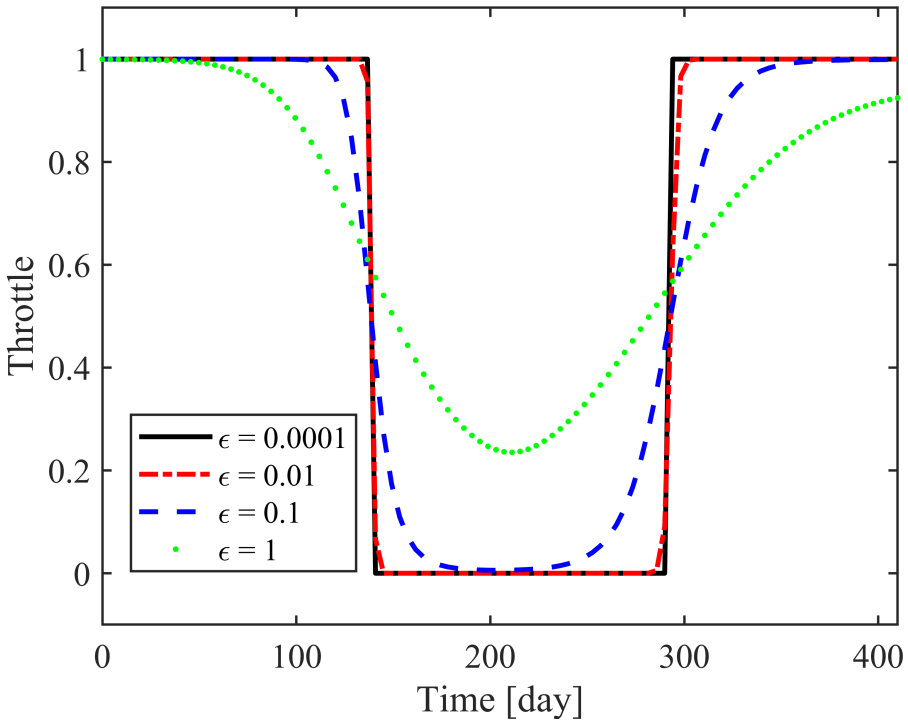

Figure 4.3. The throttle profile is relatively smooth for $\epsilon = 1$, and approaches the "bang-bang" control scheme as $\epsilon \to 0$.

4.3 Training Setup

With the problem defined and the training data generation method identified, we can now proceed to create the training data set. To start, I set up a network with 14 inputs and 7 outputs. The inputs are the current state in MEE frame, current mass, target state in MEE frame, and time-to-go. The outputs are the 7 costates at the current time. To help

the training process, I normalize all the inputs and targets aside from time-to-go to have a mean of zero and standard deviation of one [71]. I do not normalize time-to-go in this manner because its values need to be greater than or equal to zero, and since the input time is divided by the time unit its values are typically less than 5-10 already.

I use two different training methods, depending on size of the network. For relatively small networks (less than around 40 hidden nodes total), training was performed using MATLAB's `trainlm` function, which performs Levenberg-Marquardt back-propagation [51]. This algorithm computes the Jacobian and an approximate Hessian which enables the algorithm to approach second-order training speed. Due to the required Jacobian calculation, training is limited to a CPU, but empirical testing has shown that this method outperforms GPU-based back-propagation algorithms such as Adam (in Python's Tensorflow package) and stochastic conjugate gradient via MATLAB's `trainscg`. However, the training time scales with the square of the number of trainable parameters as well as size of training data, so Levenberg-Marquardt back-propagation is only practical for fairly small networks (up to around a few hundred trainable parameters). A network with 14 inputs, 7 outputs, and 40 hidden nodes has $(14 + 7) \times 40 + (40 + 7) = 887$ trainable parameters. CPU training was performed on an AMD Ryzen Threadripper 3970X with 32 cores at 4.0 GHz with 64 GB of memory. Since Levenberg-Marquardt back-propagation is very computationally intensive it is recommended not to use more workers in parallel than the number of physical cores. That means that although the processor can support 64 threads via hyper-threading, it is actually faster to just use 32 threads (and therefore 32 workers in MATLAB).

For networks that are larger than those described above, I use Tensorflow's Adam optimizer. Tensorflow is a Python library, although the underlying code is compiled C++. Tensorflow enables training on GPUs, which facilitate much larger computations. CPUs typically have fairly few cores (4-32), but each core can operate independently. GPUs have many cores (1000+), but each core performs the same operation at each time step on a different chunk of data. GPUs were originally developed to work with images and videos, where the same operation needs to performed on many pixels at the same time. This computational paradigm also lends itself nicely to large-scale matrix operations such as those

required in machine learning. GPU training was performed on an Nvidia GeForce RTX 2080 Ti, which has 4352 CUDA cores operating at 1350 MHz with 11 GB of memory.

When using MATLAB for training, I use `elliotsig` as the activation functions of the hidden nodes. `elliotsig` is a symmetric sigmoid function with a range of $[-1, 1]$ on the domain $[-\infty, \infty]$, and can be used as a replacement for hyperbolic tangent. `elliotsig` does not require exponential or trigonometric functions and is therefore faster to compute than hyperbolic tangent or sigmoid. The two activation functions are compared in Figure 4.4.

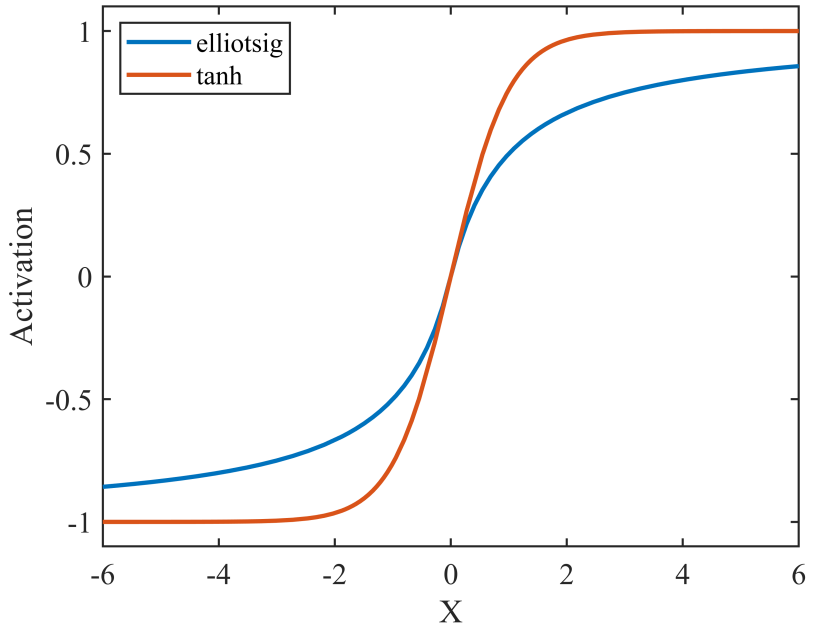

Figure 4.4. `elliotsig` is a faster alternative to hyperbolic tangent with similar qualitative behavior.

4.4 Evaluating a Trajectory

When generating the training data set, there will be some cases in which the optimizer will think that it has not converged due to numerical difficulties resulting from the behavior of many factors including relative changes in the cost function between iterations or the approximated Jacobian. However, some of these cases might actually have a low terminal error in state and could serve as useful training data. This scenario is particularly true

for more sensitive problems, which occurs as ϵ decreases in value. Thus, instead of simply listening to the exit flag returned by the optimizer, I determine whether the optimizer has "converged" based on the final value of the cost function. For the TPBVP, the cost function is defined by the terminal state error as well as one terminal costate value. I have found that, generally speaking, if the final error in radial distance is low then the solution can be considered more reliable. However, since every time the optimizer runs it will inherently have some amount of error. The error may be very small, even approaching machine precision, but it is still there. Therefore, I have to choose a cutoff value on the position error below which I will consider the trajectory a success, and above which I will consider the trajectory a failure. While this cutoff value can be picked somewhat arbitrarily, I set the limit to be the sphere-of-influence (SOI) of Mars. In other words, if the trajectory ends within Mars' SOI, then it is a success. While this limit allows the velocity error to be uncapped, I will show later that it remains reasonably low in all cases that the position error is low.

In addition to evaluating cases returned from an optimizer, I also need to evaluate trajectories that result from directly integrating the costates returned by a neural network. The same metric as above—the terminal position error—can be used for the integrator as well, since the optimizer's success criterion is set post-optimization. The same cutoff value is used in this situation as well.

Later in this dissertation I use the phrase "optimizer success" to mean success as defined by the terminal position error after receiving the initial guess from the neural network, sending the initial guess through the optimizer, and then integrating the optimized values of the costates. Similarly, I use the phrase "integrator success" to mean success as defined by the terminal position error after receiving the initial guess from the neural network and then integrating the trajectory using these values directly.

4.5 Predicting Optimal Costates at the Initial State

I started by training a neural network to predict the optimal costates only at the beginning of the trajectory as opposed to during any time along the trajectory. This network served both a test case and also as a utility to help speed up the training data genera-

tion process. For the network architecture, I used an MLP with 1 hidden layer of 40 nodes, `elliotsig` as the activation function of the hidden nodes, and a linear activation for the output nodes. The costates' values are unbounded, so a bounded activation function like tanh or sigmoid would not be acceptable. A diagram of the network, generated with MATLAB's Deep Learning Toolbox, is shown in Figure 4.5.

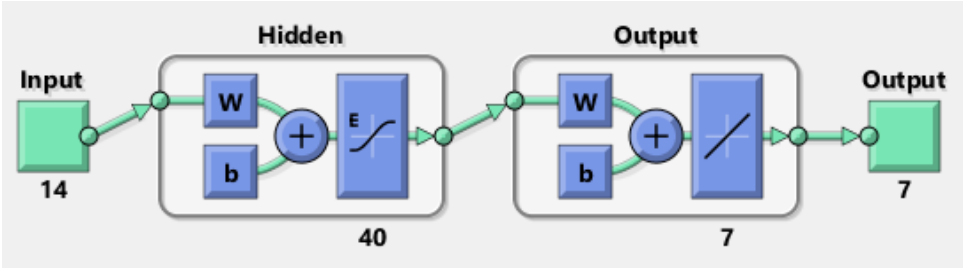

Figure 4.5. A schematic of the neural network architecture used for predicting the optimal costates produced by MATLAB's built-in neural network viewing tool.

First, I tried 1000 random sets of boundary conditions with random guesses for the costates and $\epsilon = 1$ to get a baseline for the performance of the data generation algorithm. Of the 1000 cases, 603 converged in 32.4 seconds, giving an optimizer convergence success rate of around 60%. In this test case, I stepped ϵ down from 1 to 10^{-4} by an order of magnitude at a time, giving the following intermediate values: $[1, 0.1, 10^{-2}, 10^{-3}, 10^{-4}]$. Next, I attempted the same problem but with ϵ starting at 10^{-1} instead of 1. No cases converged. Repeating the test starting at each of the lower values of ϵ all failed in every case. The fact that no cases were successful unless $\epsilon = 1$ proves that, for the vast majority of cases, if the neural network's output does converge when starting at a smaller value of ϵ, then the convergence was due to the neural network and not random chance.

To bump the success rate up from 60%, I decided to retry failed cases with a new guess of random costates up to a maximum of 5 times. With this new method, I was able to get 981 successes out of 1000 cases in 54.5 sec. There were no cases which converged at a higher value of ϵ and failed at a lower value of ϵ, and no cases that converged at lower values if they failed at higher values. Thus, we can attribute the success rate to the largest of the ϵ values used—in this case, $\epsilon = 1$. I then trained a network to these data with the knowledge that the

network would not be perfect, but could possibly give a better estimate than random with the limited information it had. The network trained for 610 iterations in 1:33 (1 minute, 33 seconds). Even with such a small initial training set, using the neural network's output as the initial guess for the solver worked on 916 out of 1000 test cases. When testing a neural network, I use its output as the initial guess for the optimizer with ϵ set the to smallest value from training unless specified otherwise. In this case, that means the network had a 91.6% success rate when $\epsilon = 10^{-4}$. I also tested the network when $\epsilon = 10^{-3}$, with a success rate of 99.4%.

Next, I used this network to generate initial conditions for 100,000 new trajectories with $\epsilon = [10^{-3}, 10^{-4}]$. Of these, 97,085 converged in 30:48. I then trained a new neural network on this new data set, which lasted for 1000 iterations in 36:31 and had a final MSE of 6.80×10^{-6}. The new neural network was able to provide initial guesses which converged in 998 out of 1000 test cases for $\epsilon = 10^{-4}$. The results of the data generation up to this point are summarized in Table 4.1. From the first to the second iteration, there was a large jump in the relative number of successes at the cost of a slight reduction in overall efficiency of number of successes generated per second. Both of these iterations relied on random initial guesses for the costates. The third iteration is when the neural network is used to provide initial guess for the costates. Here the efficiency of number of successes generated per second increases nearly by a factor of 3, which in turn reduces overall runtime by a factor of 3.

Table 4.1. Results of different data generation strategies. The success rate can be increased by reattempting failed cases multiple times, but this comes at a slight cost in efficiency. Using a trained neural network to provide initial guesses for the costates yields both a high success rate and a high efficiency.

Total Cases	Successes	Attempts	Initial Guess	Total Time [sec]	Successes/sec
1000	603	1	random	32.4	18.6
1000	981	5	random	54.5	18.0
100,000	97,085	1	NN	1848	52.5

The convergence status of the neural network as a function of launch and arrival date offset when its output is used as an initial guess for the optimizer is shown in Figure 4.6 and has a success rate of 99.8%. The only failures that occur are near the boundary of the

launch and arrival windows when both moved forward by around 20 days. The fact that the errors only occur in this corner suggests that providing more training data in this region could improve accuracy in this region. Similarly, Figure 4.7 shows the convergence status when the neural network's output is used directly as the initial conditions for the integrator. This method gave a success rate of 77.5%, with the failures forming a distinct pattern in the region where the arrival date is pushed back.

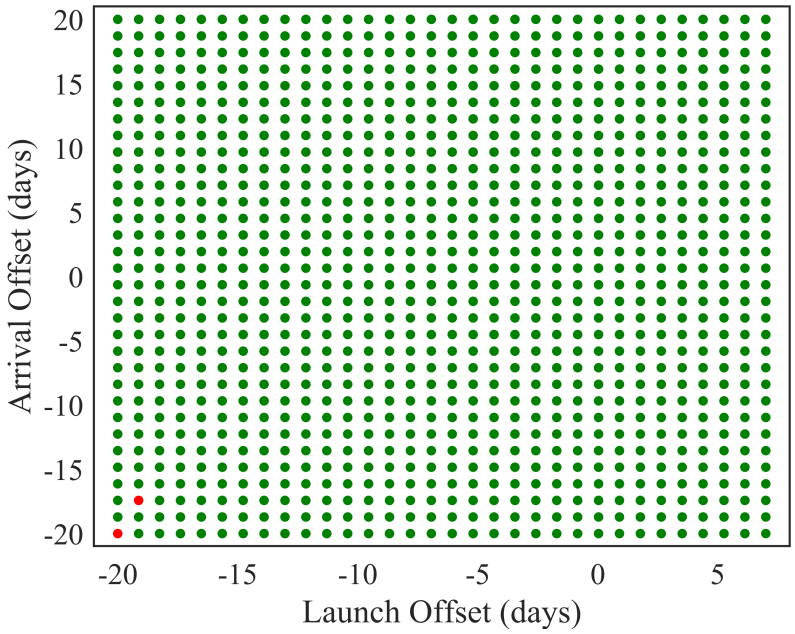

Figure 4.6. Convergence status within the launch and arrival windows for $\epsilon = 10^{-4}$ when the solver uses the neural network's guess as initial conditions. Green signifies a success, and red means the case failed.

To gain some additional insight into the problem, we can directly investigate the outputs of the network to determine if there are any patterns we can detect. Since it looks like there is some structure to the failed cases in Figure 4.7, we should see if there are any similar trends either within the values of the costates or within the network's error in predicting the costates. Figures 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, and 4.14 show contour plots of the outputs of the neural network corresponding to λ_p, λ_f, λ_g, λ_h, λ_k, λ_L, and λ_m, respectively, on the left, and the error of the network output compared to the output from the optimizer on the right. In the error plots, the size of the marker at each point is proportional to the magnitude

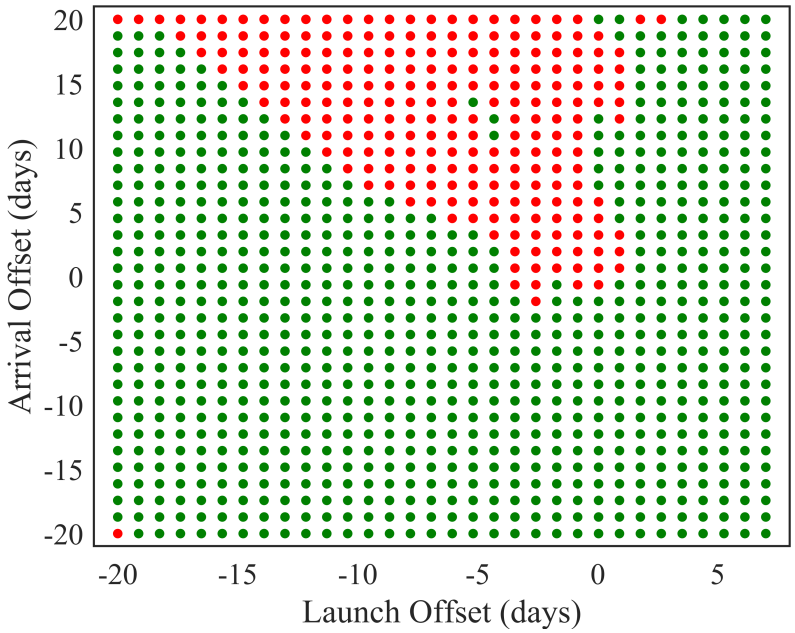

Figure 4.7. Convergence status within the launch and arrival windows for $\epsilon = 10^{-4}$ when minimum miss distance is used as the success criterion. Green cases succeeded and red cases failed.

of the error at that point, and the color signifies the sign of the error. In the value plots, white represents higher (more positive) values, while black represents lower values. We can see that, in general, the magnitudes of the errors are much lower than the magnitudes of the actual values.

After looking at these figures, we can see that two trends where the neural network struggles the most. First, there is a region occurring around a launch offset of -2 days and all arrival offsets (vertical line) that has high relative error in most of the costates. Second, there is a region that appears as a diagonal line on the error plots that moves back along the launch date and forward along the arrival date (diagonal up and to the left). By looking at trajectories around these points, we can see that around these regions the qualitative behavior of the thrust history bifurcates as the number of thrust arcs required changes from (moving forward across the launch offsets) three arcs to two arcs and back to three arcs. The region between these two bifurcations corresponds to where two thrust arcs are required, and is also approximately the area where the network fails without the optimizer. This suggests

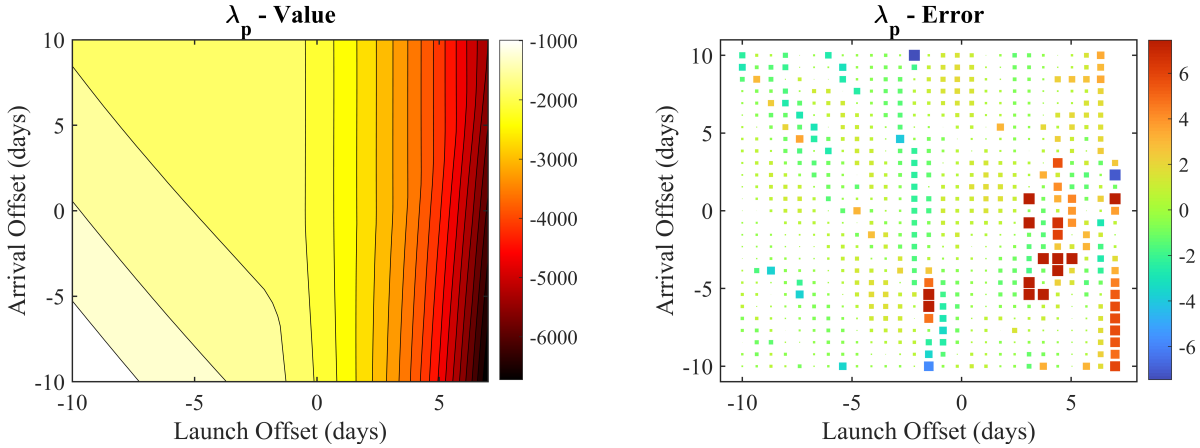

Figure 4.8. Value of and error in the prediction for λ_p.

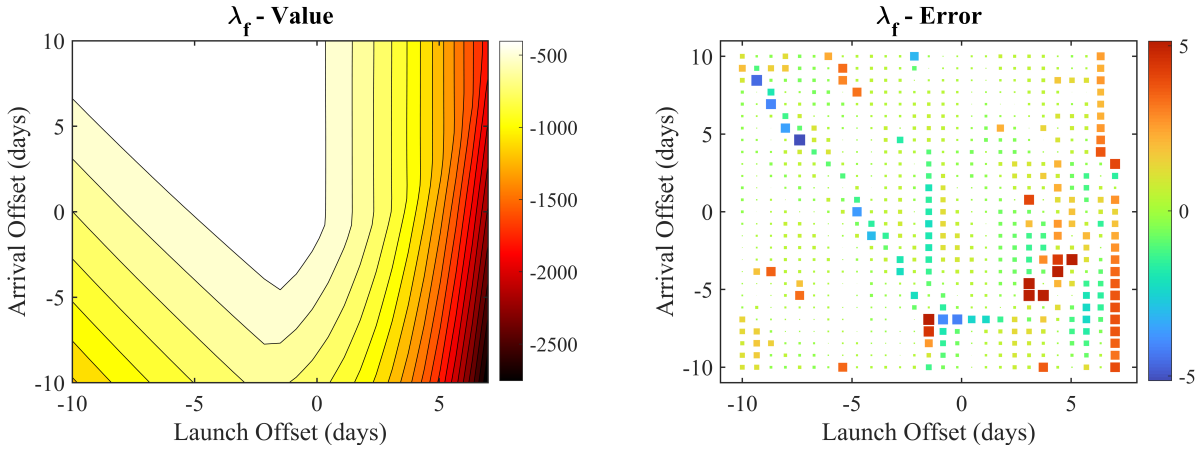

Figure 4.9. Value of and error in the prediction for λ_f.

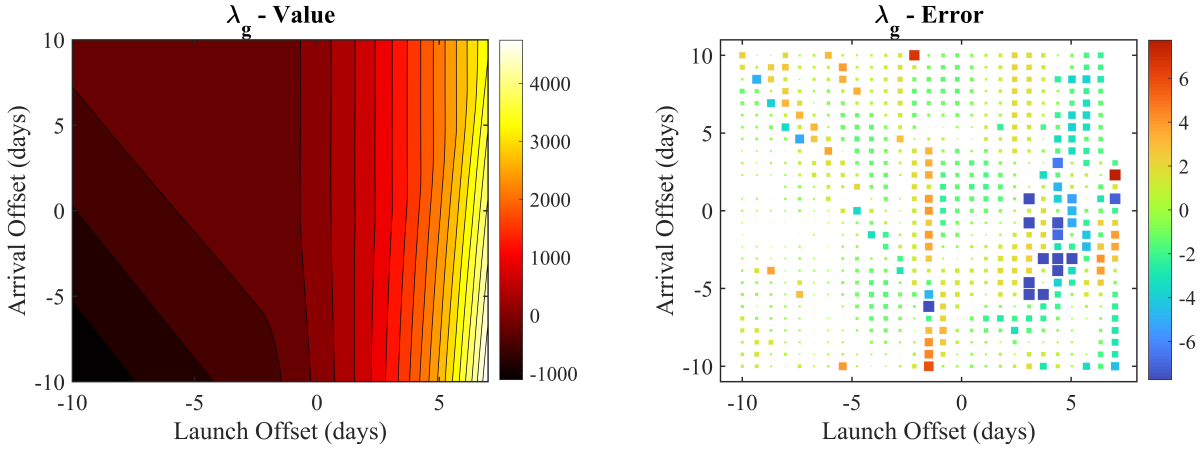

Figure 4.10. Value of and error in the prediction for λ_g.

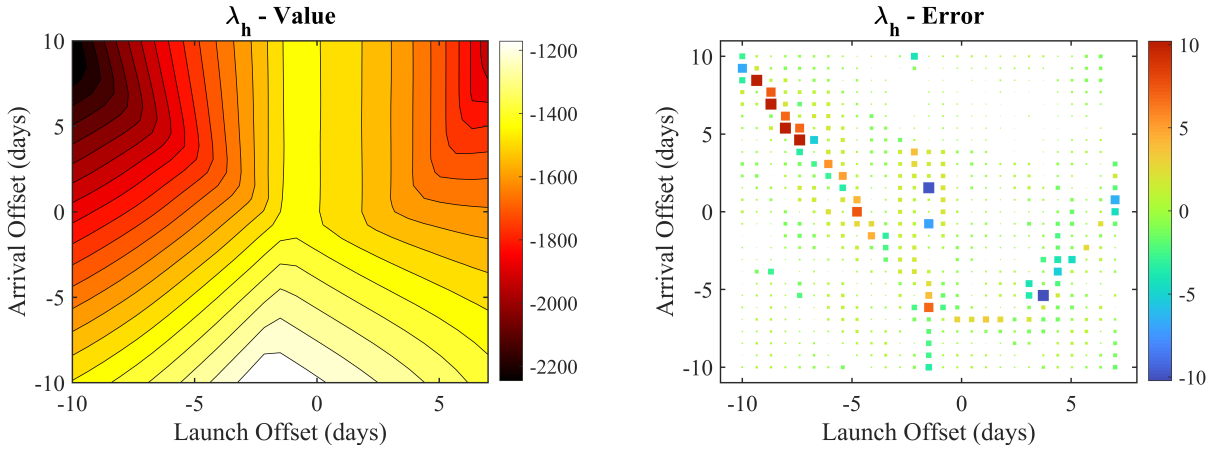

Figure 4.11. Value of and error in the prediction for λ_h.

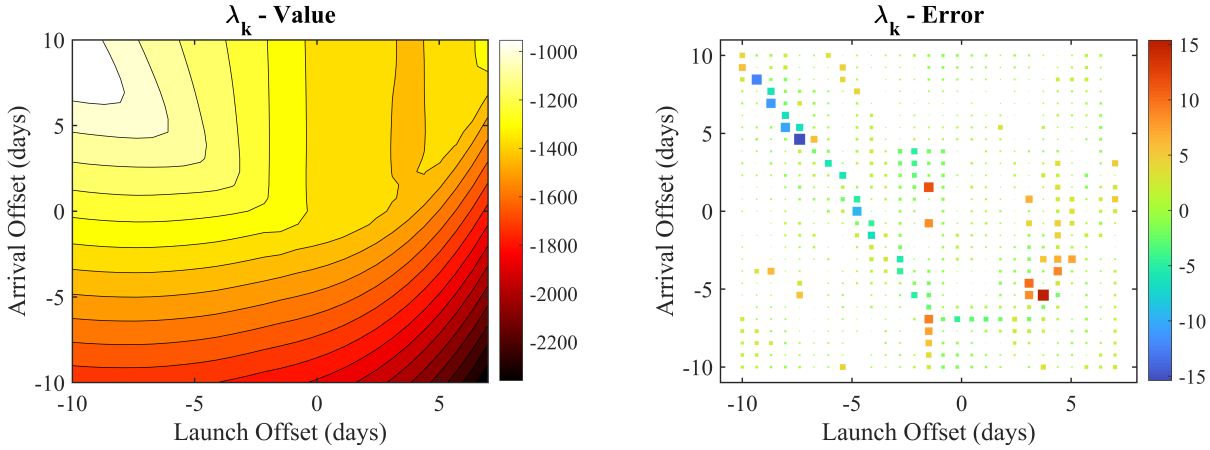

Figure 4.12. Value of and error in the prediction for λ_k.

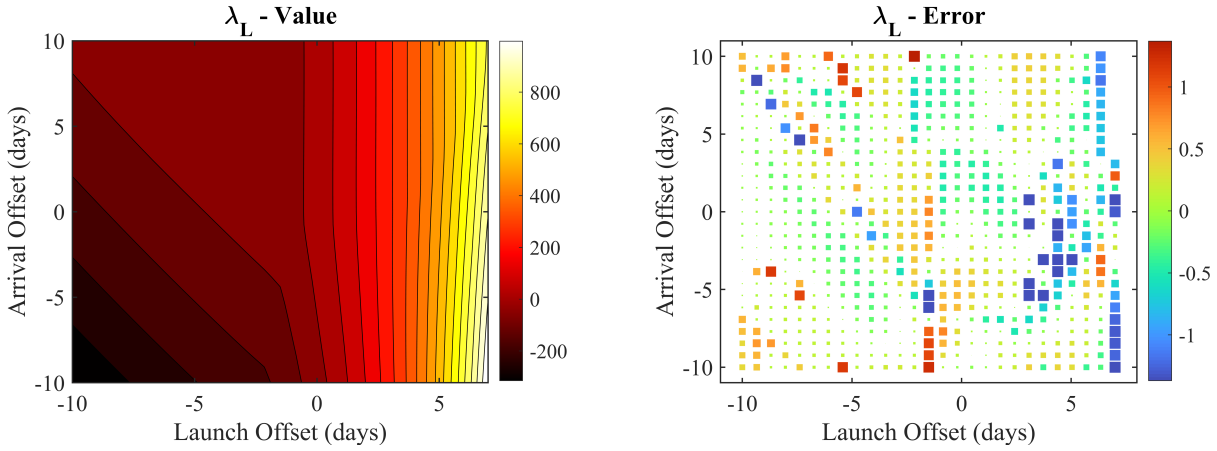

Figure 4.13. Value of and error in the prediction for λ_L.

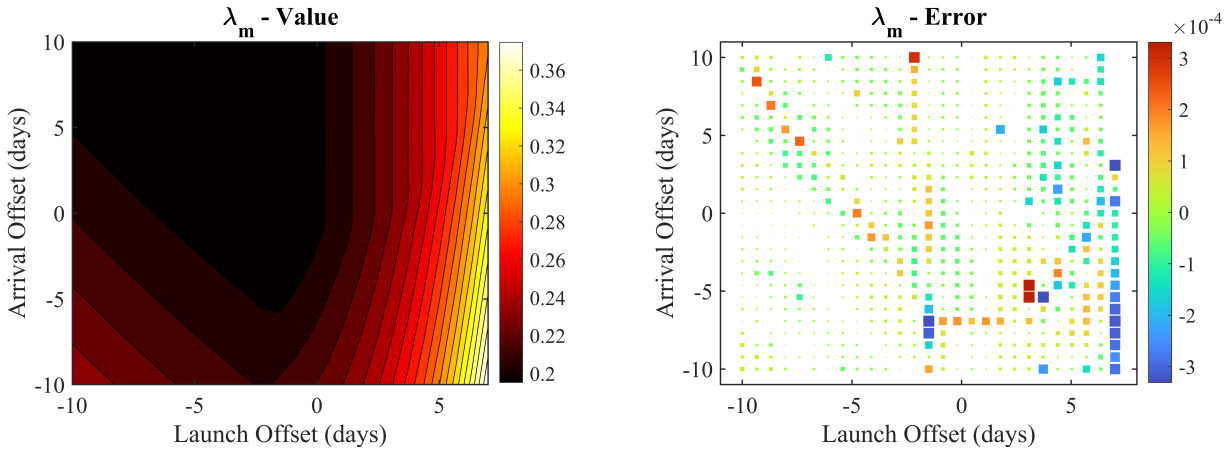

Figure 4.14. Value of and error in the prediction for λ_m.

that the network is not as effective on this problem when two thrust arcs are required. One possible explanation is that this region is simply more sensitive than the surrounding region, so initial guesses would need to have more precision to be successful. Another possibility could be that more of the training data includes three thrust arcs, so the network is more likely to produce solutions which include three thrust arcs.

The costate value and error figures all correspond to the case when $\epsilon = 10^{-4}$. However, we can test how well these costate predictions work for other values of ϵ. Table 4.2 shows the percent of successful cases for both optimizing, and directly integrating. It should be noted that in all of these cases the same neural network was used, and that the neural network was trained to predict costates for the $\epsilon = 10^{-4}$ problem. We see that for higher values of ϵ, the optimizer is able to successfully utilize the large radius of convergence of the problem despite the error becoming quite large. As ϵ decreases, the optimizer loses a few cases but still performs reasonably well. When using the neural network output as initial conditions for the integrator, accuracy with respect to the correct values is important. This means that when using higher values of ϵ the integrator does not perform well because the difference in expected states and costates grows too large. As ϵ decreases, integrator performance improves up to a maximum of around 77.7%, with the problematic region shown in Figure 4.7.

Table 4.2. Percent of successful test cases for different values of ϵ when the neural network output is used as an initial guess for the optimizer versus when they are used as initial conditions for an integrator.

ϵ	Optimizer Success Rate [%]	Integrator Success Rate [%]
1	100	0
0.1	100	6.3
10^{-2}	100	70.8
10^{-3}	99.8	77.5
10^{-4}	82.2	39.1
10^{-5}	81.6	77.6

Depending on the application of this neural network, these results can be quite promising. If this network is used to find the optimal costates for new problems with the intent to use an optimizer, then we can start by using the network with $\epsilon = 10^{-2}$ or 10^{-3} to take advantage of the near-100% convergence in this area and quickly step down to the target $\epsilon = 10^{-4}$. If this network is used for initial conditions for an integrator, we could set bounds on its applicable region according to Figure 4.7 and use it anywhere outside the problematic region.

4.6 Predicting Optimal Costates at Arbitrary Times Along the Trajectory

With the neural network to predict the optimal costates at the initial time working, I broadened the scope of the problem and looked to train a new neural network to attempt to predict the optimal costates at any time along any trajectory within the same region of launch and arrival dates as before. If the neural network can successfully complete this objective, then it should be able to work as a controller for a spacecraft subject to missed thrust since it would be able to find the optimal path to the target from any point that the spacecraft would reasonably reach.

As before, the training data set was generated with assistance from the initial-costate neural network. Of the 100,000 trajectories, 99,926 converged in 31:06 using $\epsilon = [10^{-3}, 10^{-4}]$. Each successful case was then integrated to the final time, and 20 random samples from each trajectory was collected. This provides just under 2 million training data pairs. Next, the new neural network was trained over 1000 iterations in 14:41:40, with a final MSE of 7.01×10^{-4}.

To test the network, I generate sample trajectories and randomly select states along each one to serve as initial conditions for the neural network. The same success criterion defined in section 4.4 is used for this problem. The results of testing with the optimizer are shown in Figure 4.15, and with the integrator in Figure 4.16.

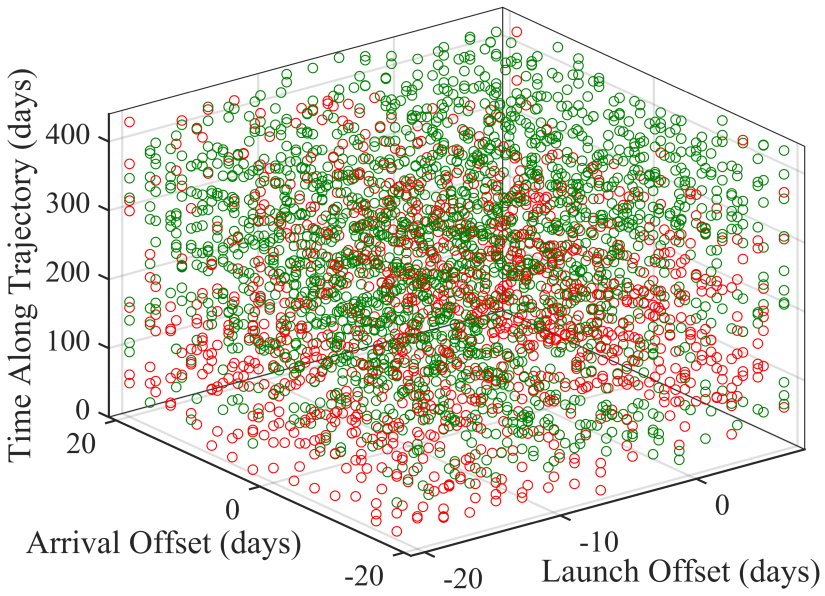

Figure 4.15. Successes and failures when using the neural network output as an initial guess for the optimizer with $\epsilon = 10^{-4}$.

These results are clearly not as strong overall when compared to the results from section 4.5, although there are still some strong points. When the output is used as initial conditions for the integrator, it only succeeds 43.9% of the time. When it is used as an initial guess for the optimizer, the success rate bumps up 70.0%. If we first set $\epsilon = 10^{-2}$ and then step down to $\epsilon = 10^{-4}$, however, the success rate with the optimizer jumps up to 99.1%. Table 4.3 shows the success rates for the optimizer and integrator for each value of ϵ. At this stage, it is not very reliable to use the output of the neural network as the initial costates directly considering they reach the destination less than half the time. However, the output is still usable if it is first passes through an optimizer. Performance would likely be boosted if a larger neural network was used, the training data set was larger, and/or the neural network was trained for longer.

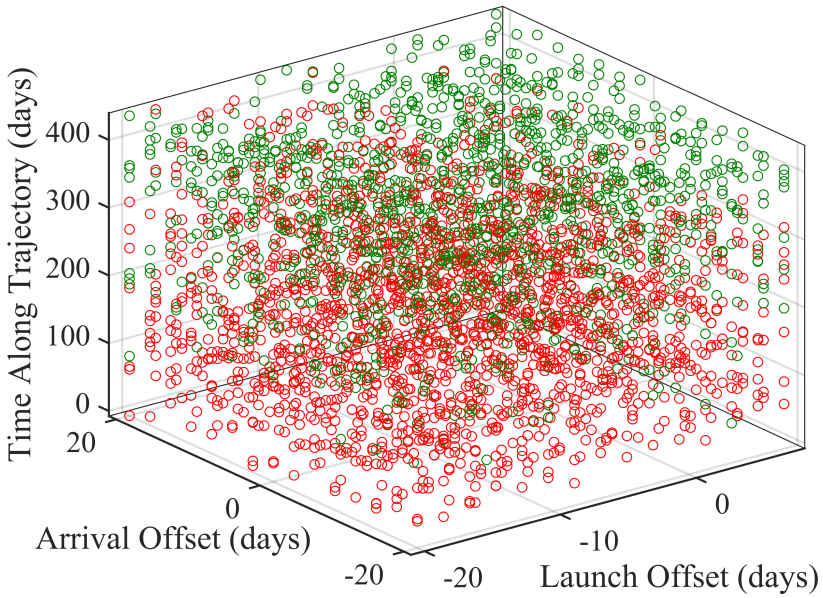

Figure 4.16. Successes and failures when using the neural network output as initial conditions for the integrator with $\epsilon = 10^{-4}$.

Table 4.3. Success rates for the optimizer and integrator when trained with 2 million data pairs and a network with 40 hidden nodes. Using an optimizer with higher values of ϵ yields a high success rate, but without an optimizer the neural network struggles.

ϵ	Optimizer Success Rate [%]	Integrator Success Rate [%]
10^{-2}	99.1	43.0
10^{-3}	90.6	43.4
10^{-4}	70.0	43.9

To test the theory that the network and training data set are too small, I augmented the training data set to include 1 million random pairs of launch and arrival times, with 25 random samples along each trajectory. In total, the training data set now includes 25 million input-output pairs, up from 2 million before. Additionally, I increased the neural network size from one hidden layer of 40 neurons to 3 hidden layers of 200 neurons each. A schematic of this network is shown in Figure 4.17. Considering that a network with 887 trainable parameters and a data set with 2 million input-output pairs took over 14

hours, using Levenberg-Marquardt back-propagation on a network with 84,807 trainable parameters and 25 million input-output pairs would lead to an impractically long training time. To accommodate the increase in the problem size, I switched to Tensorflow's Adam optimizer to train using a GPU. I also changed the hidden layer activations to ReLU since Tensorflow does not support elliotsig and the number of back-propagation computations is scaled up tremendously. Also, as the number of layers increase, ReLU's advantages become clearer. Table 4.4 shows the results of this training.

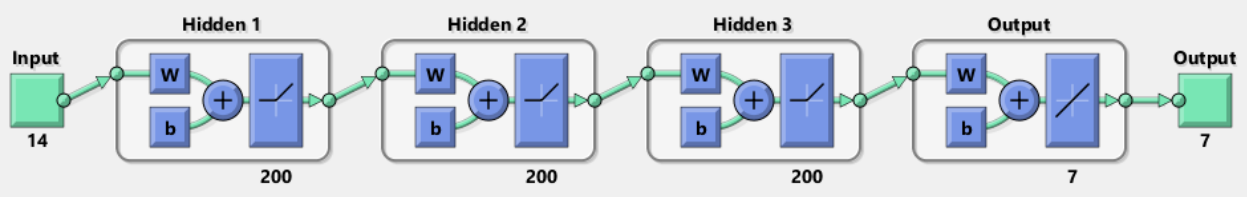

Figure 4.17. A neural network with three hidden layers. Each hidden layer has 200 neurons and uses ReLU as the activation function. The output layer has a linear activation.

Table 4.4. Success rates for the optimizer and integrator when trained with 25 million data pairs and a network with 3 hidden layers of 200 neurons. All of the metrics have been boosted, although an optimizer is still needed for reliable results.

ϵ	Optimizer Success Rate [%]	Integrator Success Rate [%]
10^{-2}	100	56.9
10^{-3}	99.3	58.7
10^{-4}	95.1	58.9

We see that increasing the network size and amount of training data does improve performance, especially when used as an initial guess for the optimizer. The optimizer success rate at $\epsilon = 10^{-4}$ jumps 25.1% from 70.0% to 95.1% and the integrator success rate jumps 15.0% from 43.9% to 58.9%. While the integrator success rate is still not great, the optimizer success rate is now quite good, especially when using a higher value of ϵ first. At $\epsilon = 10^{-3}$, the optimizer success rate is over 99%, and at $\epsilon = 10^{-2}$, every test case converged. Moving forward, we should keep in mind that small networks probably won't have sufficient capacity

to fully model the design space, and that moderately sized networks with at least a few hundred hidden nodes are necessary.

To see if there can be any more improvement in performance from a further increase in network size, I trained a network with 4 hidden layers of 500 hidden neurons each. This increase in size changes the number of trainable parameters (weights and biases) from just under 85,000 to just over 750,000. A schematic of this network is shown in Figure 4.18, and the results of training are shown in Table 4.5.

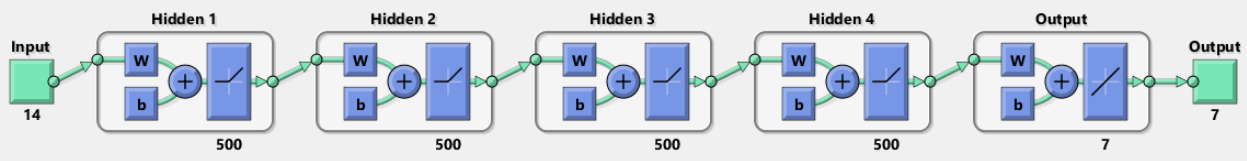

Figure 4.18. A neural network with four hidden layers. Each hidden layer has 500 neurons and uses ReLU as the activation function. The output layer has a linear activation.

Table 4.5. Success rates for the optimizer and integrator when trained with 25 million data pairs and a network with 4 hidden layers of 500 neurons. The optimizer success rate continues to approach 100%, and while the integrator success rate is higher compared to the smaller networks, it is still rather unreliable.

ϵ	Optimizer Success Rate [%]	Integrator Success Rate [%]
10^{-2}	100	62.4
10^{-3}	99.8	68.3
10^{-4}	98.8	68.3

While there was an additional increase in performance, the network size was increased by nearly a factor of 10. The optimizer success rate increased from 95.1% to 98.8%, and the integrator success rate increased from 58.9% to 68.3%. These modest increases in performance came at the cost of much longer training. Increasing the network size any further will likely lead to marginal increasing returns in performance, although it is possible that with enough training data and with a large enough network that the integrator success rate could approach 100% as well.

Up to this point, I have only considered variable boundary conditions - not missed thrust. Now I introduce missed thrust into the problem to evaluate the robustness of the controller. Using the parameters described in section 2.3.1, I can accurately represent the frequency and duration of missed-thrust events that would be expected in real missions. To simulate a single trajectory, I perform the following steps. First, I generate the boundary conditions and compute the time of flight. Once the time of flight is known, a sequence of missed-thrust events that lies within the allowable time frame is generated. After I have the sequence of missed-thrust events, I alternate the type of integration according to whether thrust is allowed or not. If the spacecraft is operating normally on a particular leg of a trajectory, then I use the neural network to compute the costates at the initial state of the leg and use these costates to integrate until the end of the leg. If the spacecraft is experiencing a missed-thrust event, then a simple 2BP propagation is used to compute the spacecraft state at the end of the leg.

I simulated 10,000 trajectories subject to random missed-thrust events using the nominal boundary conditions. The results of this analysis are shown in Figure 4.19 on a log-log plot with terminal velocity error on the x-axis and terminal position error on the y-axis. The horizontal line in the figure corresponds to the radial distance of Mars' SOI. Trajectories whose final positions are within this threshold are considered successful, and trajectories outside this are unsuccessful. Of the 10,000 trajectories simulated, 7189 converged.

Next, I simulated 10,000 more trajectories with random boundary conditions within the acceptable launch and arrival windows that are subject to random sequences of missed-thrust events. The results of this analysis are shown in Figure 4.20. 5839 of the 10,000 cases were successful, a decrease of 13.5% from the nominal boundary condition analysis.

4.7 Predicting Optimal Thrust Vectors with Missed-Thrust Events

With the technique of predicting optimal costates along the trajectory and simulating missed thrust now demonstrated, we can move to predicting the optimal thrust vectors. The methodology for predicting optimal thrust vectors is essentially the same as that to predict

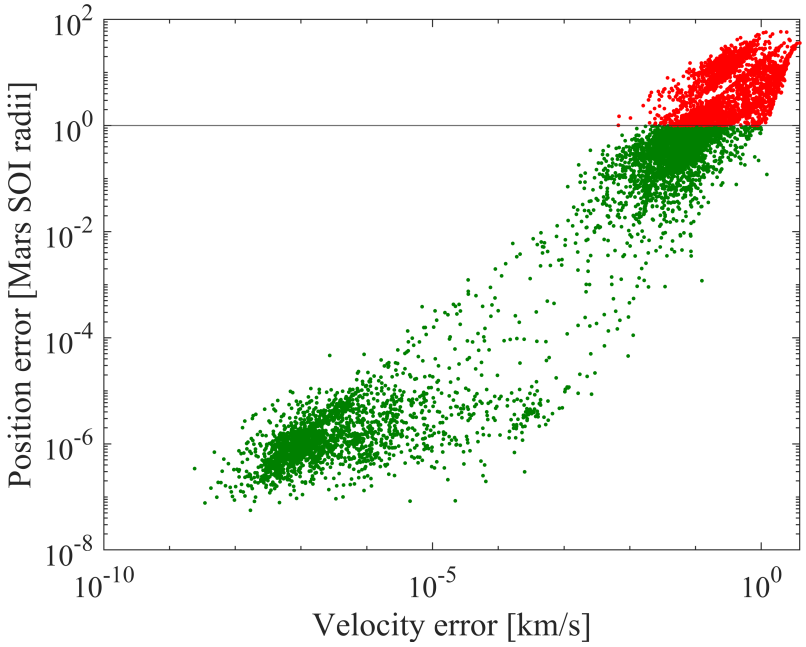

Figure 4.19. Terminal position and velocity error dispersion for 10,000 random sequences of missed-thrust events for the nominal boundary conditions when using an optimizer on the neural network output. Cases that have a final position error within Mars' sphere-of-influence are considered successful. The success rate is 71.9%.

the optimal costates, except the costates in the prior training data set are converted to thrust vectors.

In section 4.6, we saw that a network with four layers of 500 hidden nodes performed decently, but that further increases in size were likely going to provide marginal benefit. I use the same size neural network to predict the thrust vectors, except the output layer has four nodes with tanh activations. The four nodes correspond to throttle (fraction of maximum thrust used) and the three components of the thrust direction represented in the LVLH frame. Hyperbolic tangent is used for the activations since all four outputs are bounded. The throttle can be rescaled from $[-1, 1]$ to $[0, 1]$, and the direction components all have a magnitude in the range $[-1, 1]$ already. A schematic of this network is shown in Figure 4.21. I trained this network on the 25 million pair data set for 2500 iterations in a time of 7:54:17, with a final MSE of 0.0018 on the test set.

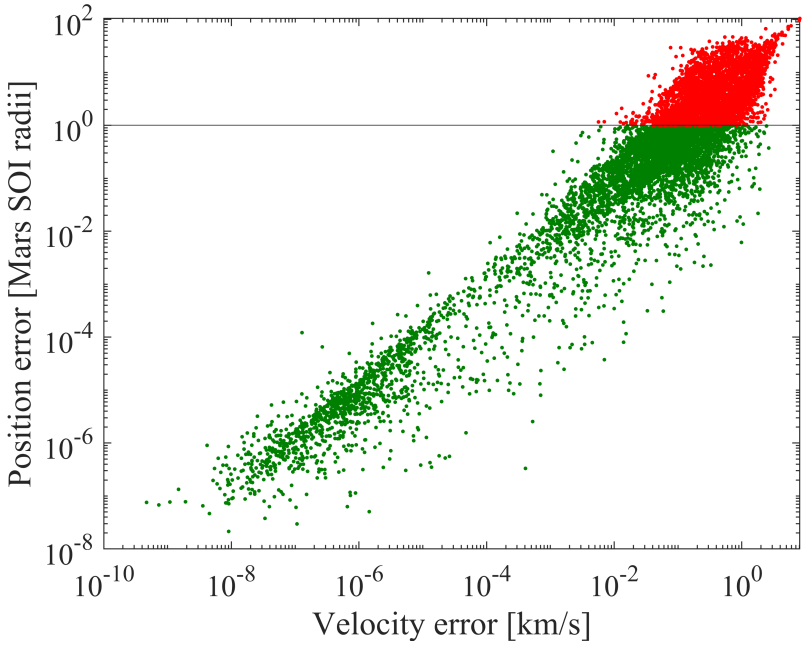

Figure 4.20. Terminal position and velocity error dispersion for 10,000 random sequences of missed-thrust events with random boundary conditions when using an optimizer on the neural network output. Cases that have a final position error within Mars' sphere-of-influence are considered successful. The success rate is 58.4%.

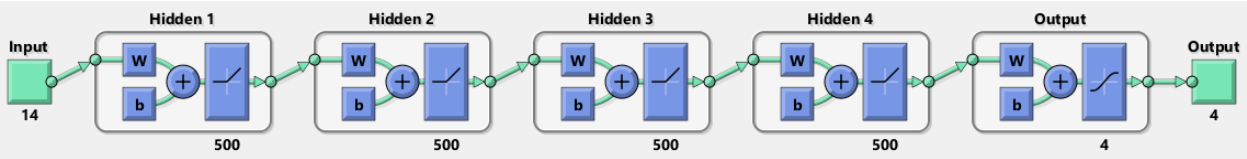

Figure 4.21. A neural network with four hidden layers. Each hidden layer has 500 neurons and uses ReLU as the activation function. The output layer has a linear activation.

Testing this neural network involves integrating the trajectory in multiple legs in the same way as in Chapter 3. The thrust vector that the neural network outputs is only valid at the state at which the network predicts the values. In this problem the direction components do not change rapidly, so each thrust vector prediction can be used for a few days along the trajectory. After a short period, the neural network predicts a new thrust vector based on its new state. The thrust vector is held constant in the LVLH frame for the duration of each leg. Figures 4.22 and 4.23 show the trajectory and thrust history, respectively, of a test case with

the nominal boundary conditions and 50 intermediate updates. With a total time-of-flight of just over 400 days, each leg is set to 8 days long. Note that the thrust history is not smooth; rather, it is composed of short segments of fixed value. The final position error is 1.64×10^6 km, or 2.84 Mars SOI radii. Based on the threshold of being within Mars' SOI, this test case is considered a failure.

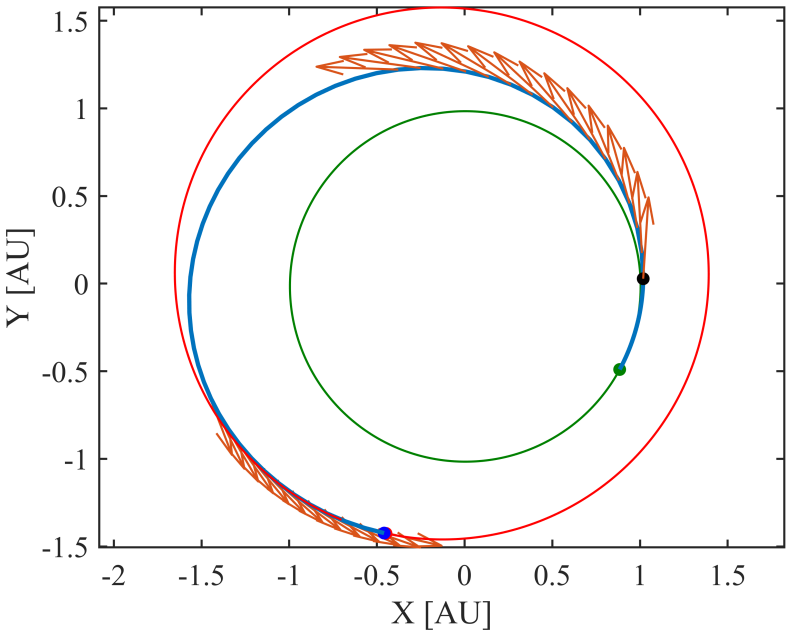

Figure 4.22. North pole plot of a trajectory from Earth to Mars, using a neural network which provides a prediction for the optimal thrust vector every 8 days.

To further test the network, I simulated 10,000 missed-thrust trajectories with variable boundary conditions. We see in Figure 4.24 that most cases failed with only 154 successes out of 10,000 cases. These results are striking, although not as surprising as it may seem at first glance with respect to the costate results. Looking at the integrator success rates in Table 4.5, we see that the neural network is successful around two-thirds of the time. Individual predictions of the thrust vector may have similar accuracy, with some predictions being close to optimal, and others being inaccurate. Since integrating to the end of the trajectory requires many predictions, the error from each prediction can compound. Especially as the network moves closer to the edge of the states seen in the training data set, the error will

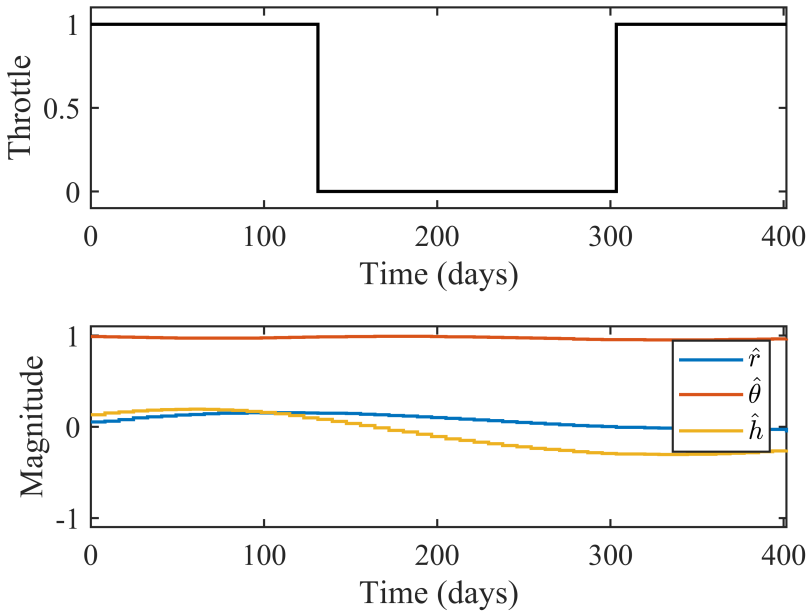

Figure 4.23. The thrust history corresponding to the trajectory shown in Figure 4.22.

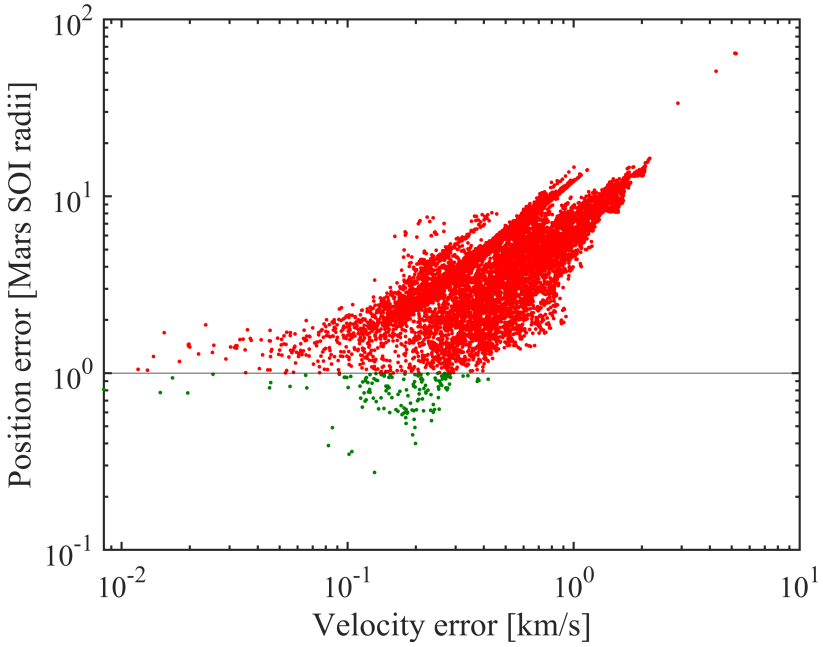

Figure 4.24. Terminal position and velocity error dispersion for 10,000 random sequences of missed-thrust events with the neural network output used as the thrust vector. The success rate was 1.54%.

continue to increase. Once the network has moved out of this region, it will almost certainly diverge from the optimal solution because it is encountering previously unseen inputs.

One way to test this theory is to compute the full state history of an optimal trajectory, and then pass a random sample of states to the neural network. We can then compare its output to the optimal solution. Figure 4.25 shows the trajectory with nominal boundary conditions, with the neural network's predictions overlaid along the trajectory. The optimal solution is shown as a solid line, and the neural network's predictions are dots. In this figure we see that, given the optimal state, the neural network gives reasonably close outputs along the entirety of the trajectory. The mean square error of these predictions with respect to the optimal solution is 1.76×10^{-3}.

One factor to keep in mind is that the input space has a dimension of 14. If any one of these inputs is significantly different from the the neural network has seen in training, that will cause error in the output. Variations that could induce these errors include for example if the spacecraft has more or less propellant at a certain position compared to the

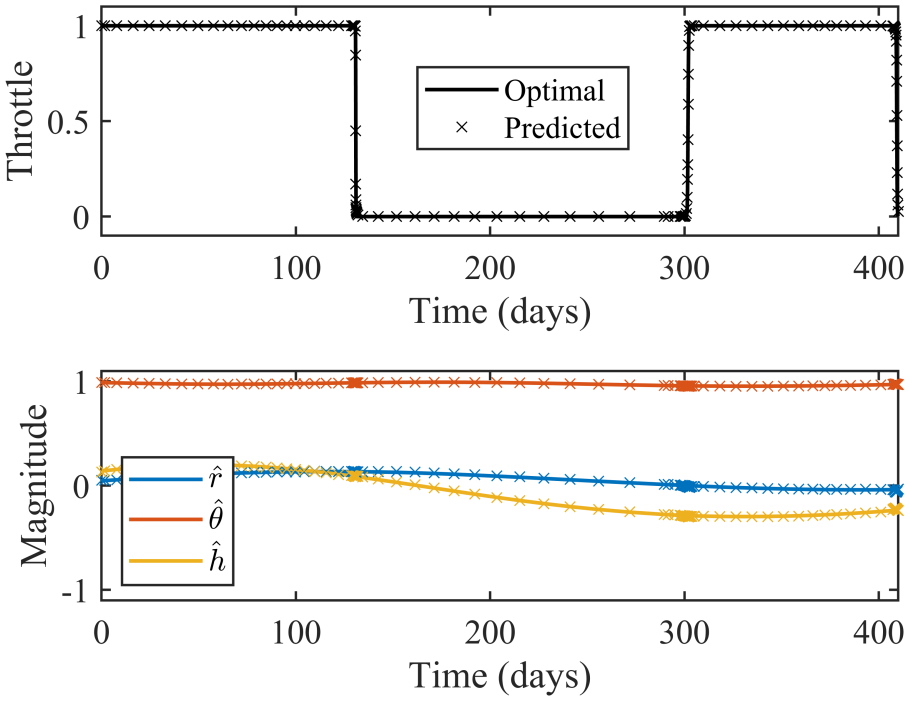

Figure 4.25. The optimal solution with the neural network's predictions overlaid. The optimal solution is shown by the solid lines, and the predictions are shown as the dots. There is good agreement along the trajectory, although small errors are still present.

optimal solutions, or if there is more or less time-to-go when the spacecraft is at a particular state. Additionally, being located a far distance away or large velocity away from the typical optimal solutions will cause errors in the thrust vector predictions.

5. COMPARISON OF RESULTS

This chapter compares the NEAT and supervised learning training methods on a new and more complex problem than the problems previously investigated. The goal of this comparison is to highlight the utility of each training method with respect to each other, as well as their general shortcomings.

Additionally, a new method is introduced which is a derivative of the supervised learning method. I refer to this method as "brute force", and it entails repeatedly attempting to solve the problem without any prior knowledge. This method does not require machine learning, and is meant to represent a comparison to more traditional optimization techniques.

5.1 Additional Method - Brute Force Optimization

When generating the data set for supervised learning, I used a homotopy-based indirect optimization method to solve the TPBVP. A notable feature of this algorithm is that it is able to converge even with a relatively poor initial guess. Considering that one of the potential purposes of using a neural network is to provide a prediction for the costates at a given point, it would be useful to compare its performance to a method that does not use a neural network. We saw in Table 4.1 that while a neural network was able to boost the speed at which optimal solutions could be generated, using multiple attempts at a high value of ϵ was still able to provide a strong convergence rate. We could try this method either with a completely random guess, or we could use a small amount of intuition for the initial guess and use the optimal costates corresponding to the nominal initial conditions with a small perturbation added.

5.2 Earth-to-Venus Transfer

For a direct comparison of the methods, I use an Earth to Venus transfer using a low-thrust spacecraft that requires multiple thrust arcs and revolutions to reach the destination. This problem is 3D and considers only solar gravity. The nominal trajectory has a launch date of 05 May 2025 and an arrival date of 02 September 2027 for a total TOF of 850 days.

The spacecraft has a dry mass of 2150 kg, a propellant mass of 850 kg, a maximum thrust of 0.281 N (corresponding to one XR-5 Hall thruster at max power), and an I_{sp} of 1850 s. The transfer is shown in the inertial heliocentric frame in Figure 5.1 and 5.2, and the corresponding thrust profile is shown in 5.3. Figure 5.1 shows the trajectory, looking down the inertial "north" vector. There are nearly three full revolutions, with thrust occurring on several arcs. Figure 5.2 exaggerates the Z components of the trajectory to highlight the plane change aspect of the problem, and how the direction of thrust is dependent on location along the trajectory. In Figure 5.3 we see there are five distinct full-thrust arcs with four null-thrust arcs.

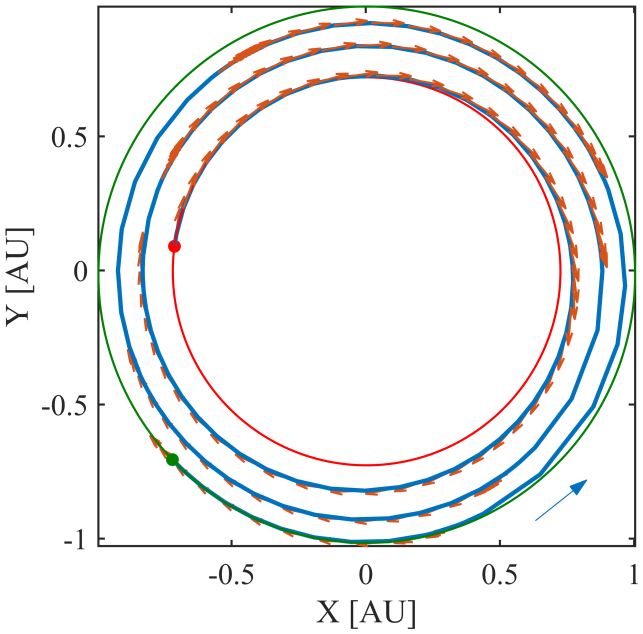

Figure 5.1. A fuel-optimal low-thrust orbit transfer from Earth to Venus with a time-of-flight of 850 days and multiple alternating thrust and coast arcs.

This problem offers a challenge due to the multiple spirals and multiple thrust arcs that are required. An early outage affects every thrust arc later along the trajectory, which, especially early on, can be difficult for a neural network to compensate for when the time horizon is still very long.

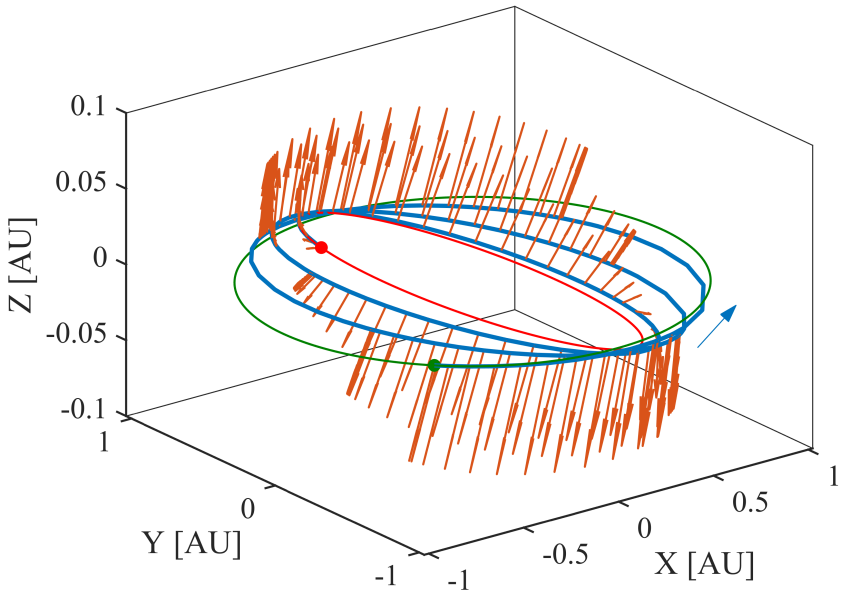

Figure 5.2. This figure exaggerates the Z-component of the transfer to highlight the out-of-plane component of the thrust vectors.

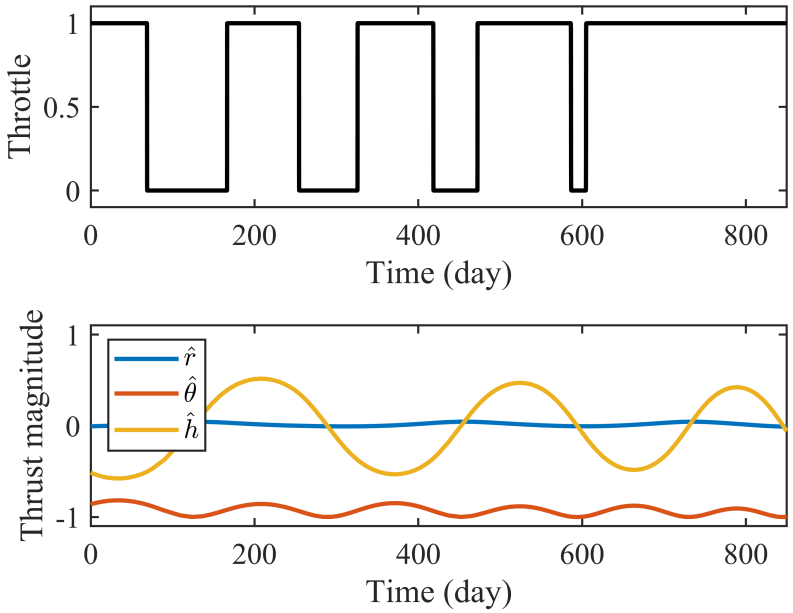

Figure 5.3. The optimal Earth to Venus transfer has five thrust arcs and four coast arcs, and a clear oscillation in the required out-of-plane thrust.

5.2.1 NEAT

In this section I investigate the performance of a neural network trained with NEAT on the Earth to Venus transfer problem defined above with variable launch and arrival dates. I attempted several configurations of hyperparameters to reliably converge, but in all attempts the network struggled to reach a final position error within Venus' SOI. The position errors after each attempt were typically around 5-10 million km, or around 8-16 Venus SOI radii. The network configurations included various combinations of the parameters shown in Table 5.1. Because training with NEAT was not able to succeed with variable boundary conditions without missed thrust, I did not consider the more challenging problem of variable boundary conditions with missed thrust.

Table 5.1. Hyperparameter values used during testing for the Earth-to-Venus transfer with variable boundary conditions. The underlined values correspond to the case shown below in Figure 5.4.

Parameter	**Values**
initial hidden neurons	0, 2, 4, 6, 8, 10, 15, 20, 30
initial connection rate	0.5, 0.6, 0.7, 0.8, 0.9, 1
allow direct connections	yes, no
weight mutation rate	0.1, 0.3, 0.5, 0.6, 0.7, 0.8
weight mutation power	0.1, 0.3, 0.5, 0.7, 1
weight replacement rate	0.01, 0.05, 0.1
bias mutation rate	0.1, 0.3, 0.5, 0.6, 0.7, 0.8
bias mutation power	0.1, 0.3, 0.5, 0.7, 1
bias replacement rate	0.01, 0.05, 0.1
activation function replace rate	0.01, 0.025, 0.05, 0.1
hidden activations	tanh, sigmoid, relu, sin, gauss, step
connection add rate	0.01, 0.05, 0.1
connection delete rate	0.01, 0.05, 0.1
node add rate	0.01, 0.05, 0.1
node delete rate	0.01, 0.05, 0.1
population size	500, 1000, 1500, 2000
number of cases per iteration	5, 10, 15, 20, 25, 50, 100, 150
iterations	100, 150, 250, 500
stagnation period	50, 100, 150
terminal Lambert arc	yes, no
input frame	Cartesian, COE, MEE
output frame	VNC, LVLH

Other factors not listed in Table 5.1 include elitism, survival threshold, compatibility threshold, weight and disjoint compatibility coefficient, response mutation parameters, aggregation parameters, maximum and minimum weight and bias limits, and initial mean and standard deviation for weights and biases. In all of the cases tested, the inputs to the network were current state, target state, time to go, and spacecraft mass (all scaled to approximately order 1), and the outputs were throttle and the three components of the thrust direction vector. The terminal state error was computed in the MEE frame, scaled by a weighting factor, and added to the total propellant mass consumption to determine each neural network's fitness function. If the terminal state is within Venus' SOI, then the state error would not be included in the fitness function—only the total propellant mass consumption would be considered. The results of a "typical" run are shown in below in Figure 5.4 using the underlined values in Table 5.1. This particular trajectory has the nominal boundary conditions, with a terminal position error of 6.54×10^6 km or around 10.6 Venus SOI radii.

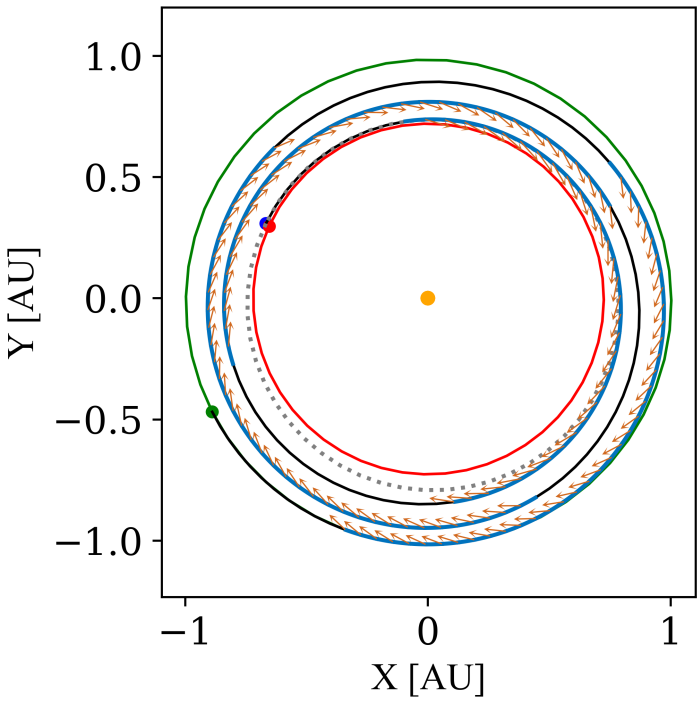

Figure 5.4. The resulting trajectory when using a neural network trained with NEAT with variable boundary conditions.

In addition to trying different hyperparameters on separate runs, I also attempted multi-phase training that starts with high variability parameters (e.g. weight mutate rate = 0.8, weight mutate power = 0.5) and then changes to smaller variability parameters (e.g. weight mutate rate = 0.3, weight mutate power = 0.1) later in training. The intent behind this strategy was to begin with a more global search to investigate a large area of the solution space, and then end with a more local search once a promising region has been identified to more closely approach a local minimum. In practice, however, the multi-phase training did not provide significant differences compared to single-stage training.

An interesting finding from working with NEAT was the variability in qualitative behavior of the output. Sometimes the neural network would output a constant (body-fixed) thrust vector that performs well on a few cases and poorly on others. Other times the output would have a clear sinusoidal behavior. In some cases the output has a more complex behavior, but the contributing functions can be fairly clearly made out within the response. Figures 5.5, 5.6, and 5.7 show control histories with nearly constant, sinusoidal, and more complex values, respectively. The clear dependence of the output on a single function is likely the result of having fairly few (0 to 10) hidden neurons, and could be alleviated by either including more initial neurons or by increasing the probability of neurons and connections being added. However, as more neurons and connections are introduced into networks, the time to evaluate and reproduce between generations increases. Also, it is best practice to scale the number of individuals of the population with the number of trainable parameters, which leads a further increase on run time [64].

Another empirical finding came from adjusting the number of cases to evaluate per iteration. When the number of cases per iteration is relatively low (i.e. less than 10), it is common at some point during the training for a network that performs well in one particular region to get many samples in that region on one random run, which is then set as the best network. Because this network "got lucky" once, it is difficult for other networks to make improvements against it. Alternatively, when the number of cases per iteration is relatively high (i.e. greater than 50), the solutions often tend to find a solution that ends in an "average" state; these solutions perform well in the middle of the launch and arrival windows, but poorly on the ends. In theory, it would be preferable to evaluate a network

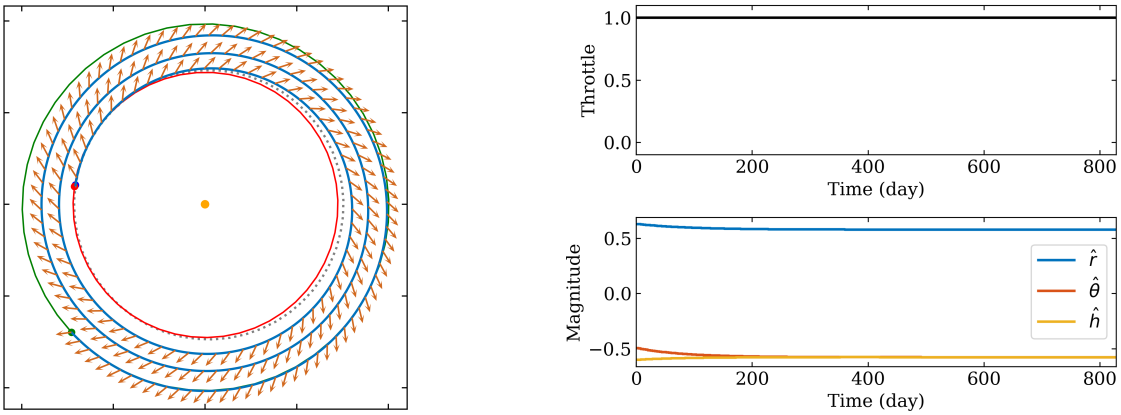

Figure 5.5. A control history from a neural network trained with NEAT that results in constant values.

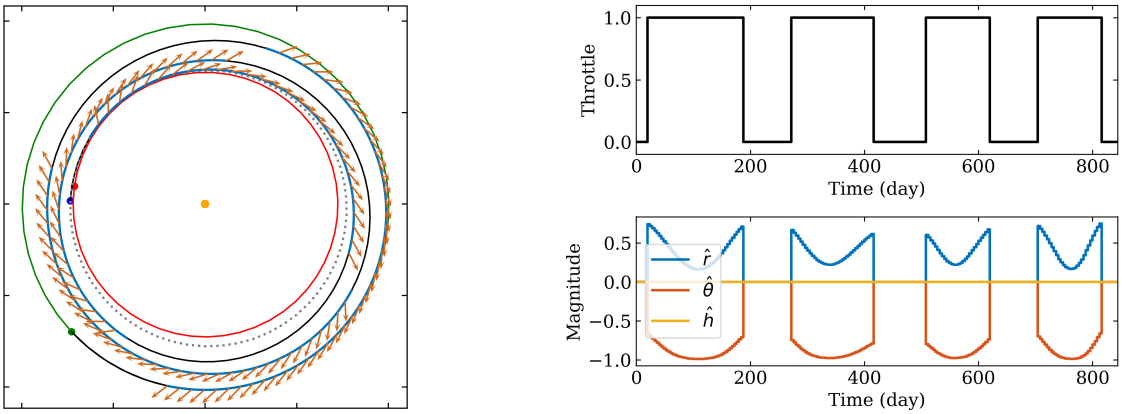

Figure 5.6. A control history from a neural network trained with NEAT that results in sinusoidal values.

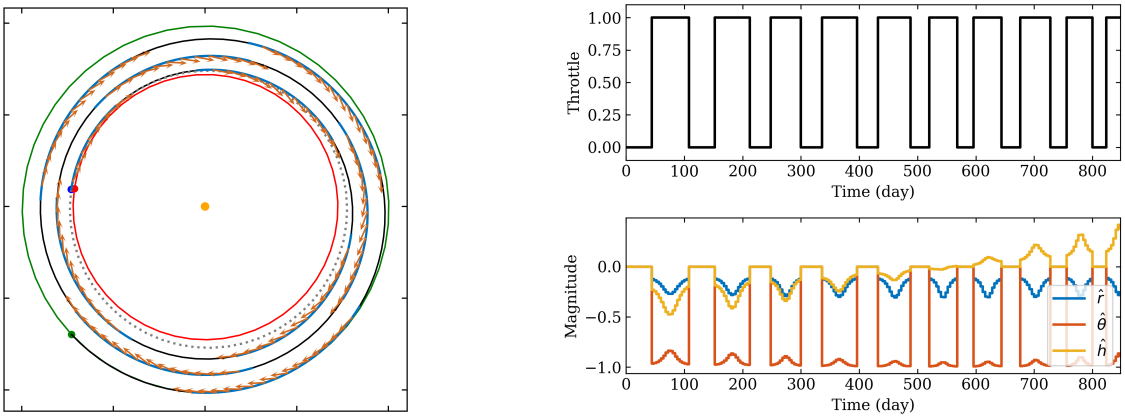

Figure 5.7. A control history from a neural network trained with NEAT that results in more complex values than the other two examples.

on as many cases per iteration as possible to get a statistically meaningful measure of the network's performance. However, the run time directly scales with the number of cases per iteration. Training with 50 cases per iteration will take 10 times longer than training with 5 cases per iteration.

To put the run time issue into perspective, consider that a population of 1500 individuals with 150 sample cases per generation requires integrating 225,000 trajectories per generation. To let the algorithm run for 500 iterations would require integrating over 112 million trajectories. If we can integrate at a rate of 1000 trajectories per second, it would take over 31 hours to complete 500 iterations. Two other factors that impact run time are hyperparameter selection and performing validation runs. Hyperparameter selection and tuning requires multiple runs to find the best parameters for the problem. While this process does not necessarily need to be performed when using a very high number of individuals, cases per generation, and total generations, it still requires many runs overall. Due to the stochastic nature of evolutionary algorithms, it is considered best practice to run the algorithm several times to verify that the same or similar final solutions are found each time. With high-performance computing (HPC) resources, this would be a more tractable problem; however, on a single CPU computer, the run time can be excessively long (with consumer-grade CPUs that exist in 2022).

5.2.2 Supervised Learning

First, I investigate the performance of a neural network trained with supervised learning when the launch and arrival dates are variable. Next, I take it a step further and look at the performance when missed-thrust events are included in the trajectory.

As in Chapter 4, I must start by creating a training data set of optimal trajectories from which the network can learn. A method to help speed up the process is to train a network to predict the optimal costates at the beginning of a trajectory, given information about its current (initial) state and target state. Such a network can then be used to provide initial guesses for other trajectories which can then converge faster than when a random guess for the costates is provided. The initial-guess network was originally trained

on 100 trajectories with a 10-day launch and arrival window. This network was then used to generate 1000 trajectories in a 20-day launch and arrival window. The intermediate initial-guess network was used to generate 100,000 trajectories with a 40-day launch and arrival window. Each variant of the initial-guess network had one hidden layer of 40 neurons and `elliotsig` activation, with a linear output activation. Using these intermediate networks to help generate additional trajectories was not necessary, but resulted in decreasing the total time required to create the training data set.

The fully-trained version of the initial-guess network was used to help generate an additional 100,000 trajectories which were each then integrated to their final time, with 20 random samples along each trajectory being saved. The original 100,000 trajectories from the initial-guess network were integrated and collected into the data set. In total, this provided 200,000 trajectories $\times$ 20 samples per trajectory = 4 million training data pairs. A neural network with 4 hidden layers of 500 neurons with ReLU activation and a linear output activation (same as shown in Figure 4.18) was trained on this data set.

To first test the network's performance, I simulated 500 random trajectories with 20 points sampled randomly along each for a total of 10,000 test cases. Integrating the neural network's costate predictions directly succeeded in 8643 cases, and using the optimizer with $\epsilon = 10^{-4}$ before integrating succeeded in every case. Next, I simulated 10,000 random missed-thrust sequences with variable boundary conditions. The results of this analysis are shown in Figure 5.8. 9600 cases reached Venus' SOI successfully when using an optimizer, and cases reached Venus' SOI when directly integrating the costates. An interesting point to note is the decrease in success rate between non-missed-thrust and missed-thrust analyses. Even though the network is given and is predicting the same information for both scenarios, the network performs worse when the states at which it is predicting result from missed-thrust events instead of being on an already optimal trajectory.

5.2.3 Brute Force

Now I demonstrate the performance of using a brute force method to directly solve the Earth-to-Venus TPBVP for both a standard trajectory optimization problem and with

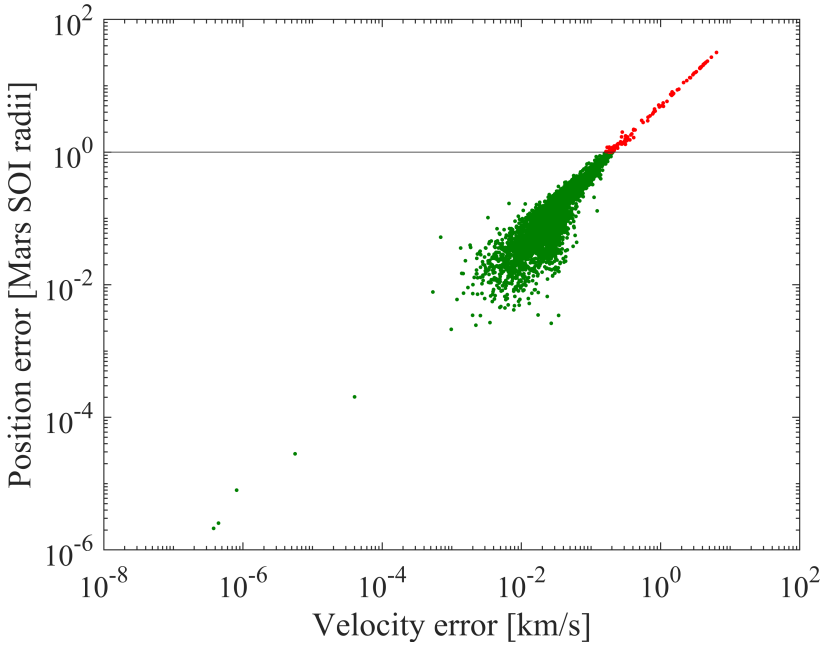

Figure 5.8. Terminal position and velocity error dispersion for 10,000 random sequences of missed-thrust events with random boundary conditions when using an optimizer on the neural network's prediction for optimal costates. Cases that have a final position error within Venus' sphere-of-influence are considered successful. The success rate is 96.0%.

missed thrust included. Using the method described in Section 5.1, I first investigated computing the optimal costates along many trajectories without considering missed thrust. To do this, I generate a batch of random boundary conditions within the allowable launch and arrival windows, and for each pair I attempt to find the optimal costates at the initial time. Once the initial costates are found, I integrate each trajectory to the final time and take samples at random times along the trajectory. Once I have a data set of random states, I attempt to find the corresponding optimal costates at each state.

I performed a test with a launch and arrival window of 40 days, 1000 random boundary conditions, and 10 random samples along each trajectory. Of the 1000 sets of boundary conditions, I was able to successfully compute the initial costates for each case. Of the 10,000 total cases, I successfully compute the costates in 9984 cases. Next, I tested the method by introducing missed-thrust events. Of the 10,000 random cases, 6850 converged. The results are shown in Figure 5.9.

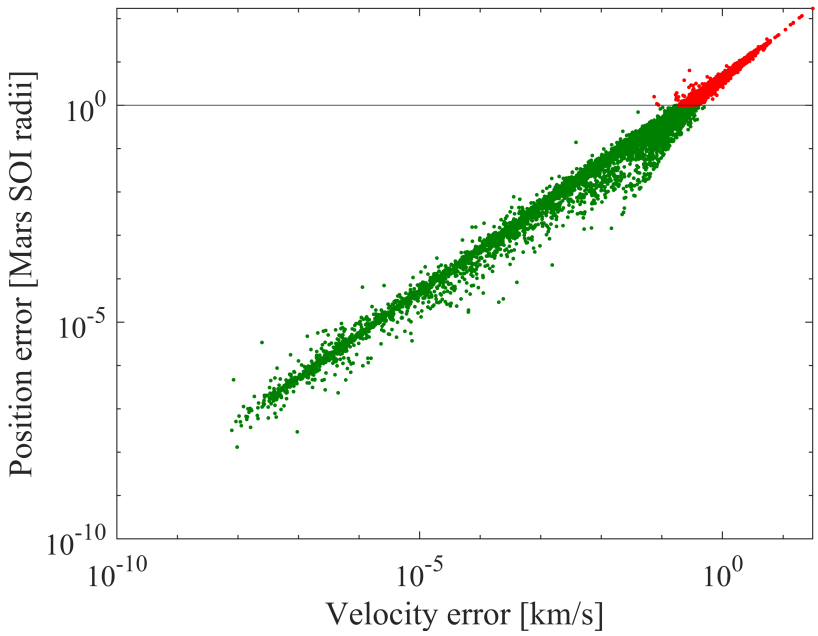

Figure 5.9. Terminal position and velocity error dispersion for 10,000 random sequences of missed-thrust events with random boundary conditions when using an optimizer with random initial guesses for the costates. Cases that have a final position error within Venus' sphere-of-influence are considered successful. The success rate is 68.5%.

5.3 Discussion

After comparing NEAT and supervised learning as training methods on the same problem, we can see some clear pros and cons. NEAT was not able to train a neural network to successfully guide a spacecraft to the target given a range of launch and arrival dates without considering missed thrust. Despite trying many different configurations of hyperparameters, I was not able to train a consistently reliable network. I believe the sizes of the neural networks that I investigated were too small to be able to guide the spacecraft with the required level of accuracy. With additional computational resources that enable training larger networks, this problem may become more tractable. Conversely, a neural network trained with supervised learning was able to guide a spacecraft to the target in the presence of missed thrust 96% of the time, compared to the baseline performance of the brute force method which has a success rate of 68.5%.

The fact that both supervised learning and brute force had a lower success rate when missed thrust events were included suggests that the sensitivity of the costates after missed thrust events is higher compared to costates along an optimized trajectory. To account for this increase in sensitivity, future work could examine including states resulting from missed-thrust events into the training data set.

6. HYPERBOLIC ABORT OPTIONS FOR HUMAN MISSIONS TO MARS

6.1 Introduction

Missions to Mars will require hyperbolic rendezvous of one form or another, since every interplanetary mission to Mars has in a sense performed a rendezvous with Mars after departing Earth with a hyperbolic velocity. However, the more standard use of the phrase "hyperbolic rendezvous" has one vehicle rendezvous with another in a span of days, not months. A typical scenario has a cycler vehicle in a hyperbolic trajectory with a taxi vehicle launching from Earth that must dock to the cycler.

The comparison between launching to Mars or to a cycler as a form of hyperbolic rendezvous is to highlight a feeling of unease that only appears whenever the idea of hyperbolic rendezvous is discussed in relation to a cycler. Research has shown that simple guidance algorithms are capable of performing rendezvous with a cycler [72]. Now, I discuss the remaining fear associated with hyperbolic rendezvous to show that several abort strategies exist with compelling capabilities that can reduce risk.

Traditionally, a cycler vehicle is a massive spacecraft similar in scope to the International Space Station that travels between Earth and Mars without stopping. Many such cycler trajectories have been studied including the original Aldrin cycler [73]–[79].

The taxi vehicle is typically much smaller and is designed to carry crew and supplies from the surface of Earth or Mars to the cycler. Two examples of this scenario have even appeared recently in popular culture in the movie *The Martian*. The first is when supplies are launched from Earth to rendezvous with the crew ship, *The Hermes*, as it flies by Earth before heading back to Mars. The second example is Mark Watney launching from the surface of Mars to rendezvous with the crew vehicle. Both times the cycler vehicle is traveling at hyperbolic velocities past the planets.

6.2 Overview of Previous Studies Regarding Hyperbolic Rendezvous

Previous work has investigated the relationship between the taxi and the cycler vehicle when attempting to minimize the initial mass in low Earth orbit (IMLEO) [80], [81], and propellant required for hyperbolic rendezvous in general [76], [82]–[91]. Minimizing IMLEO is a standard practice because sending mass to orbit is extremely expensive—on the order of $10,000 per kg. Studies that have looked into decreasing the risks of hyperbolic rendezvous found a 10 percent increase in IMLEO when various safety measures and redundant systems were employed [72], [92]. Figure 6.1 shows the general strategy of these studies.

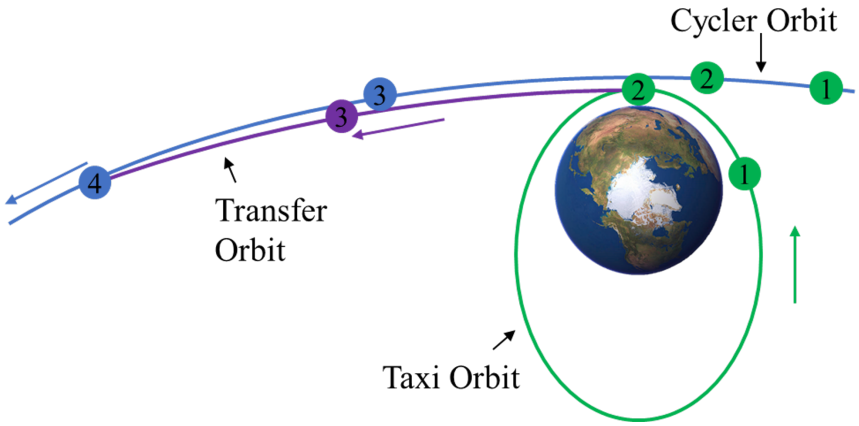

Figure 6.1. Previously studied hyperbolic insertion and rendezvous strategy. The spacecraft starts from an elliptical orbit whose perigee is aligned with that of the target orbit, performs a maneuver at periapsis to launch it onto a hyperbolic transfer orbit, and then performs a small correction maneuver far from Earth to enter onto the target orbit.

Therefore, when investigating abort strategies, we must be aware of any additional increase in the Δv requirements. The Δv can be thought of as a mass ratio between the required spacecraft final mass after the propellant has been spent, and the initial mass when the required propellant is still inside the spacecraft. Most hyperbolic rendezvous strategies use an outbound rendezvous where the spacecraft departs from a low parking orbit or periapsis around the Earth or Mars before a small burn is used to complete rendezvous with the cycler around 24 hours later [72].

6.3 Proposed Method

While the rendezvous (and more generally, orbital insertion) strategy discussed above may be optimal to simply enter into the hyperbolic orbit, it does not consider the Δv that would be required if the spacecraft has to divert back to Earth in the event of an emergency or other anomaly. With humans on board, there is no tolerance for error. Therefore, a new hyperbolic insertion approach is presented with this concern in mind. Instead of performing the orbital insertion on the outbound leg, the spacecraft enters a highly elliptical orbit and then performs a maneuver along the inbound leg of the target hyperbolic orbit. Doing so allows for an early abort option with a low additional cost as well as a second abort option several hours later when the spacecraft reaches periapsis. Figure 6.2 shows an overview of this orbital insertion strategy. Starting from LEO, the spacecraft performs a maneuver that puts it into the blue taxi orbit. After the spacecraft passes apoapsis and approaches the target orbit, it performs a small maneuver to put it on the red transfer orbit which crosses paths with the target orbit. Finally, the spacecraft performs a maneuver at the specified location to enter into the yellow target orbit.

Using this technique results in a lower IMLEO than attempting to abort on the outbound arc when the taxi is already outside of most of Earth's gravity well, which can require several km/s of Δv to recapture into an elliptical orbit. Therefore, when abort strategies are included in the design of a taxi vehicle, an inbound hyperbolic orbit injection appears to be a safer and cheaper solution than an outbound orbit injection.

6.3.1 Orbital Insertion

The following assumptions are made for this analysis. First, the analysis only investigates insertion into the hyperbolic orbit corresponding to the flyby of a cycler in the S1L1 orbit in 2035 [78]. The S1L1 cycler orbit is one of the leading candidates for cycler trajectories, and its approach in 2035, as well as the preceding and following approaches, are characterized in Table 6.1. Also, since the point of the analysis is to create a safe method for orbital insertion, all transfers are elliptic and the spacecraft only has $C_3 > 0$ after the final maneuver. In other words, the spacecraft is guaranteed to stay in cis-lunar space (in the short term) up until the

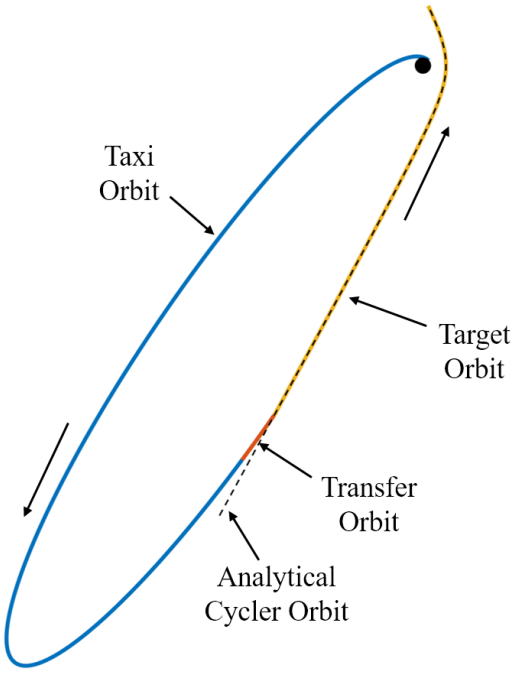

Figure 6.2. Representative orbital insertion sequence. The spacecraft starts in a circular low-Earth orbit, performs a maneuver to enter the taxi orbit (blue), performs a small maneuver to enter the transfer orbit (red), and then performs a final maneuver to enter the target orbit (yellow).

final maneuver is complete. The spacecraft starts from a 200 km altitude circular orbit and is co-planar with the target orbit. The period of the taxi orbit is limited to 10 days, which corresponds to a maximum eccentricity of approximately 0.966 and, with the 200 km altitude periapsis, to an apoapsis of roughly 385,000 km. This apoapsis radius is close to the Moon's semi-major axis of 384,400 km. Furthermore, all maneuvers are impulsive, and low-thrust is not considered. Finally, orbital insertion is only considered for distances closer than the radius of the Moon's orbit. The 10-day limit on the taxi orbit period was selected based on the Orion capsule's 21-day undocked lifetime requirement [93]. A taxi orbit period of half of the acceptable operational lifetime of a representative taxi vehicle allows for extra time if the vehicle must abort at any point leading up to or during the rendezvous. If necessary, the taxi vehicle could safely stay in its orbit for a second revolution before needing to return to Earth. Having a relatively long 10-day abort period also reduces the Δv requirements when compared to having a shorter window.

Table 6.1. Parameters defining the 2033, 2035, and 2037 approaches of the S1L1 cycler trajectory by Earth.

Year	r_p **[km]**	V_∞ **[km/s]**
2033	33100	4.41
2035	16100	3.75
2037	15400	4.25

Two methods are pursued to find the minimum Δv to insert into the target hyperbolic orbit at an arbitrary true anomaly. First, assuming a constant periapsis altitude, there exists a single taxi orbit for each true anomaly along the target orbit such that the taxi orbit is tangent at that point. However, the required transfers are parabolic or hyperbolic past a certain true anomaly, so this transition point acts as an upper limit. Since this analysis aims to insert on the inbound leg, we are naturally limited to only elliptical cases. The results of this analysis are shown in Figure 6.3. This figure shows the orbital insertion Δv as a function of the true anomaly of insertion along the target orbit. The blue line represents the Δv to transfer from the parking orbit to the taxi orbit, the red line represents the Δv to transfer from the taxi orbit to the target orbit, and the yellow line represents the total of these two maneuvers. Inserting close to periapsis on the target orbit requires less Δv to transfer from the parking orbit to the taxi orbit, but requires more Δv to transfer to the target orbit since the difference in energy between these orbits is greater. As the orbital insertion moves farther from the target orbit periapsis, these trends flip since more energy must be put into the taxi orbit in the first maneuver. The Oberth effect states that a maneuver that occurs at higher speeds results in a greater change in total mechanical energy compared to the same maneuver occurring at a lower speed. Thus, performing more of the total required Δv in the parking orbit will be more efficient from an orbit insertion perspective. The total Δv required is highest at the target orbit periapsis, and decreases slightly at farther distances. The minimum total Δv for this method occurs at the farthest distance for which the taxi orbit is still elliptical, which in this analysis occurs at true anomalies of around $\pm 105^\circ$. Due to the symmetry of conic orbits and the vis-viva equation, the Δv magnitudes are symmetric about periapsis. Examples of these trajectories are shown in Figure 6.4.

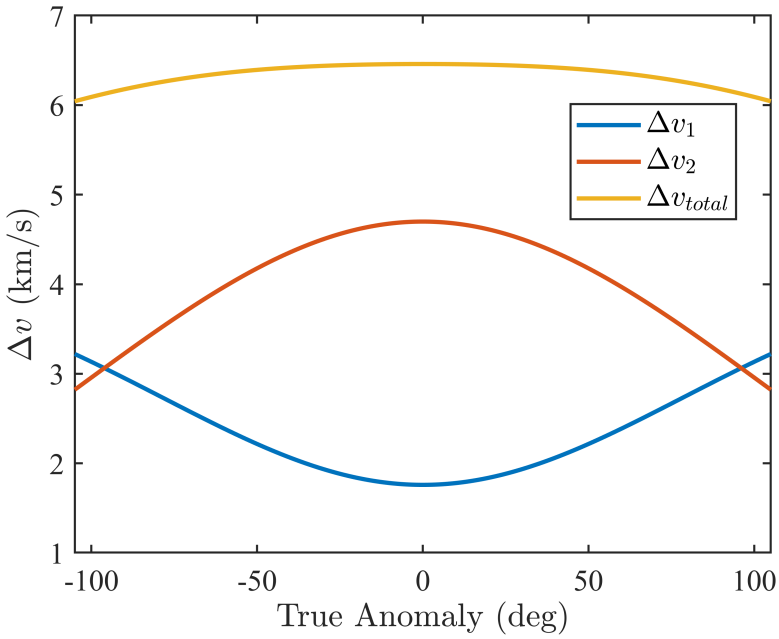

Figure 6.3. Δv to perform orbital insertion on a tangent taxi orbit. As more Δv is performed closer to Earth, the total Δv decreases.

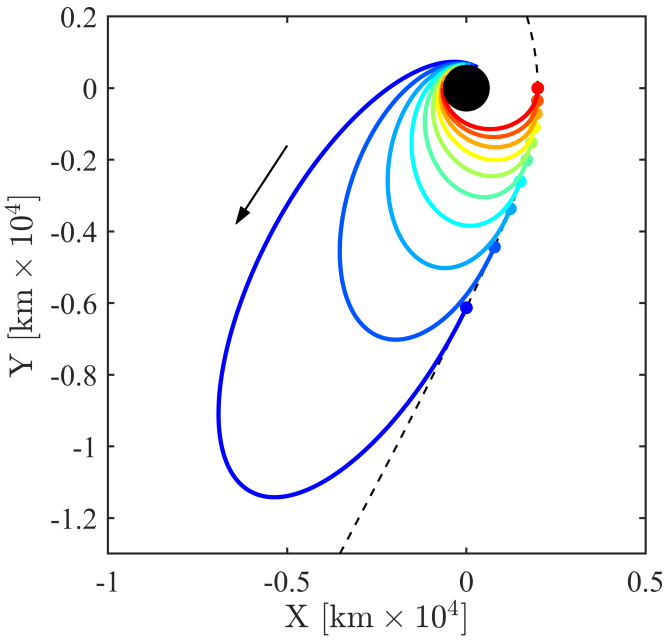

Figure 6.4. Orbital insertion sequence for various true anomalies of arrival using the tangent constraints.

The second method of finding the minimum Δv to perform orbital insertion involves direct optimization. This method implements a local optimizer whose optimization variables include the argument of perigee of the taxi orbit, the eccentricity of the taxi orbit, the true anomaly of departure from the taxi orbit, and the time of flight of the transfer orbit. Since the periapsis altitude of the taxi orbit is fixed to 200 km (coming from the 200 km circular parking orbit), the eccentricity also specifies the semi-major axis. The argument of perigee is measured with respect to that of the target S1L1 orbit. A Lambert arc can be found using the radius of the taxi orbit at the departure location, the radius of the target orbit at the arrival location, the transfer angle between the two radial vectors, and the time of flight optimization variable—this is the transfer orbit. A locally optimal total Δv can be found at a specified arrival true anomaly along the hyperbolic orbit, and sweeping along the entire hyperbolic trajectory provides the lowest Δv required to perform orbital insertion at any point along the trajectory. The results of this analysis are shown in Figure 6.5. Examples of these trajectories are shown in Figure 6.6.

6.3.2 Abort Options

The need to abort from a hyperbolic insertion may arise for many reasons. To cover as many scenarios as possible, I assume that the spacecraft needs to abort after already being placed on the target orbit to find a worst-case estimate of the Δv. I assume that the burns are impulsive and can happen instantaneously after entering the target orbit, and that the spacecraft has not lost any control or functionality of its engines. I define an acceptable abort orbit in this analysis to mean any non-hyperbolic trajectory.

Two methods to abort from a hyperbolic orbit are considered. The first method uses a direct optimizer to find the minimum Δv Lambert arc that has a periapsis altitude of 100 km while staying within the 10-day time-of-flight constraint. The other method performs a deflection maneuver such that the periapsis of the new orbit has a periapsis altitude of 100 km, the flight path angle is negative (the spacecraft is descending), and the spacecraft maintains the same orbital energy. Therefore, the deflection maneuver changes the flight path angle, but doesn't affect the velocity magnitude. Assuming the spacecraft has a heat

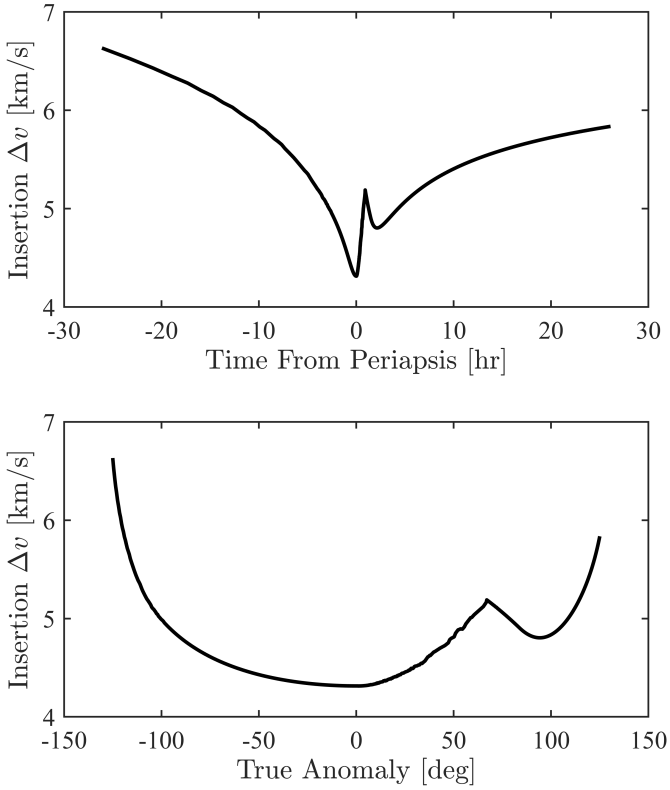

Figure 6.5. Minimum Δv to perform orbital insertion using direct optimization to compute the optimal taxi orbit size and orientation.

shield rated for Earth re-entry (e.g. Orion), the low pass at 100 km through the atmosphere allows it to aerobrake and eventually re-enter. This abort method is very advantageous when the spacecraft is far away from Earth on the inbound leg since the change in flight path angle is very small. However, it becomes impractical as the spacecraft moves closer to Earth, and even more so on the outbound leg as the change in flight path angle approaches 180°.

Although aerobrake trajectories were not specifically targeted, the difference between a 100 km altitude and 0 km altitude periapsis or between a 100 km and 200 km altitude periapsis is within ±75 m/s at all points along the trajectory and approaches 0 m/s moving away from periapsis. Since this study investigates trends as opposed to specific, high-fidelity computations, these differences are not included and a targeting a 100 km altitude periapsis in all cases is considered sufficient.

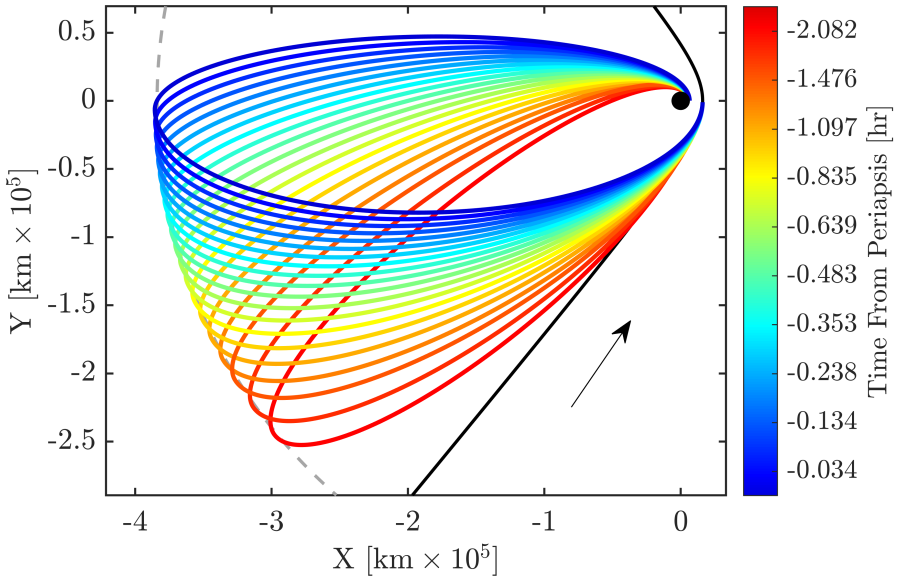

Figure 6.6. Orbital insertion sequence for various true anomalies of arrival. These trajectories were found using the direct optimization method and include a small transfer orbit. The gray dashed line represents the Moon's orbital radius, and the black line represents the target S1L1 orbit.

Figure 6.7 shows the magnitudes of the two types of aborts maneuvers, with the Δv is plotted as a function of both true anomaly and time past periapsis. While the plot with true anomaly may seem like the low-Δv benefits occur only when far away, on the plot with time we see that the spacecraft is only within a true anomaly of $\pm 100^\circ$ for a relatively short time, and that most of the time is spent towards the extremes. Thus, in addition to the lower Δv requirements, the inbound leg strategy also allows for a longer time to abort. If a spacecraft had a redundant stage that could provide 2 km/s, that would enable the spacecraft to abort at any location before a true anomaly of 105° according to the scenario shown in Figure 6.7. If the spacecraft performed orbital insertion on the inbound leg, it would have until 2.6 hours before periapsis to perform an abort maneuver for under 1 km/s. Depending on where the rendezvous maneuver is planned to occur, the crew could have around a full day of time to perform checkout maneuvers while keeping the option to perform a low-cost abort. When

performing orbital insertion on the outbound leg, the cost to abort continues to increase as soon as the spacecraft enters the hyperbolic trajectory.

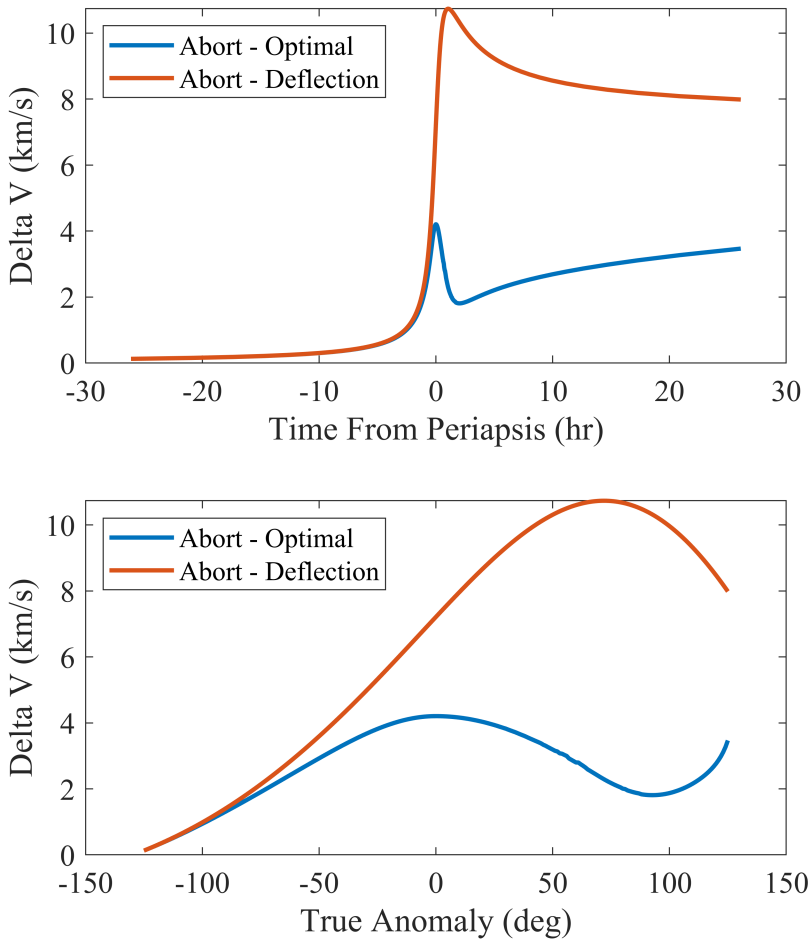

Figure 6.7. Δv required to perform an abort maneuver as a function of the true anomaly at which the maneuver occurs along the hyperbolic orbit. The deflection abort maneuver is very effective far away from Earth on the inbound leg, but becomes impractically large as it approaches and passes Earth. The parabolic abort maneuver is lowest at and symmetric about perigee.

In Figure 6.7, we see that the cost to abort is much lower on the inbound leg compared to the outbound leg. We also see that on the inbound leg, the optimal Δv is very close to the Δv to perform the deflection maneuver described above. The similarity between these two on the inbound leg demonstrates the geometric advantage between the two sides of the

trajectory. Additionally, the fact that the optimal abort is lower than deflection maneuver on the outbound leg is due to the 10-day abort window. This abort window is also why the Δv drops after periapsis until around 90° true anomaly. If the 10-day period was shortened for mission requirements or for a time-dependent emergency such as a life-threatening injury or loss of life support, then the optimal abort magnitude would increase and approach that of the deflection maneuver.

6.4 Results

Figure 6.8 compares the maneuver magnitudes for orbital insertion, abort, and the sum of the two. From this figure we can see that the total Δv including the orbital insertion maneuver and the abort maneuver is lower when orbital insertion is performed on the inbound leg of the hyperbolic orbit. This figure also shows the minimum available abort magnitude if the spacecraft can wait until a future point—essentially the minimum abort magnitude between the current and all future locations. The total Δv on the inbound leg stays below 6.75 km/s until a true anomaly of around 101° or 2.91 hr after periapsis, with the minimum of 5.88 km/s occuring at 3.38 hr before periapsis or at a true anomaly of $-107.5°$.

Future work could investigate the relative Δv required for different S1L1 approaches. Only the orbit for the 2035 is examined, so the effects of varying V_∞ and r_p on the abort maneuver cost has not been determined. Additionally, the orbital insertion maneuvers in this analysis were subject to constraints on TOF and orbital energy. Relaxing these constraints would lower the magnitude of the orbital insertion maneuver. Further analysis could investigate the cost and risk trade-off between keeping the TOF and/or orbital energy constraints or not.

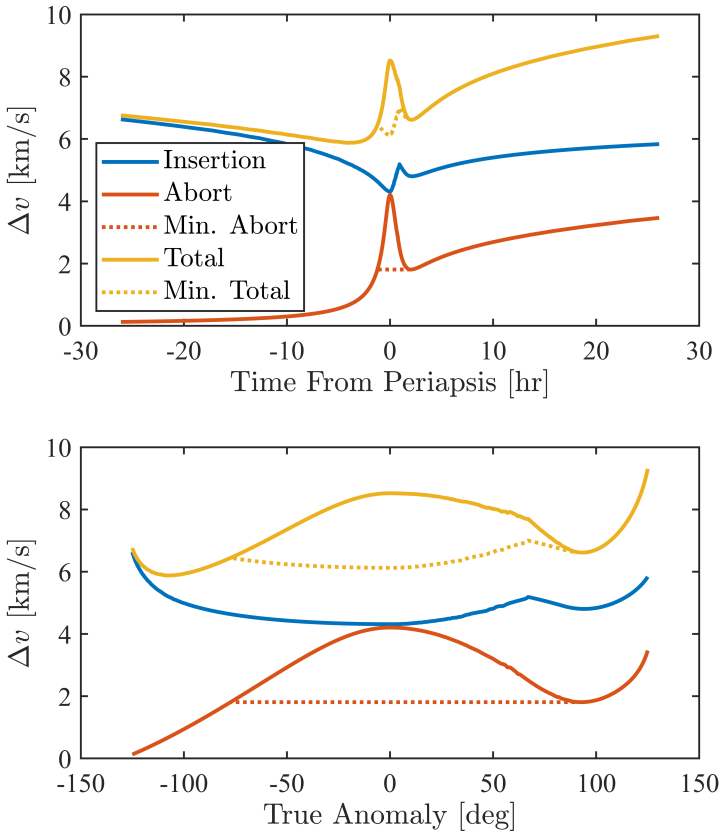

Figure 6.8. Total Δv required when considering both the orbital insertion and abort maneuvers.

7. CONCLUSIONS AND FUTURE WORK

This dissertation has touched on multiple different topics. Here I weigh the conclusions of each chapter and discuss opportunities for future research.

7.1 Neuroevolution

7.1.1 Evolutionary Neurocontrol with an RNN

The current investigation has shown that the proposed method of using an RNN as a controller was not able to provide meaningful advantages over using an FFNN. The genetic algorithm and particle swarm algorithm both have long run times to find a single trajectory on the order of minutes to hours. The RNN did not demonstrate any significant additional generalization capability or improved convergence compared to an FFNN used in the same capacity for the cases studied. While the trajectory considered to test the RNN is likely too simple to showcase its full effectiveness, I believe it is unlikely for an RNN to be able to effectively control a spacecraft in a manner that an FFNN cannot and decided to forgo further study.

7.1.2 NEAT

Using NEAT to train a neural network yielded mixed results when performing missed-thrust trajectory design. Because it does not require prior knowledge of the problem via a training data set, it can be applied to problems which would otherwise be very difficult or even intractable for other types of solvers. However, because it is powered by a global search algorithm as opposed to a local optimizer, the final solution is not guaranteed to be locally optimal. Similarly, due to the stochastic nature of genetic algorithms, running the algorithm on the same problem multiple times may yield different results. There are many avenues for improvement of the current work and exploration into new applications. With an improved speed of the implementation and additional computational resources, this algorithm has the potential to provide solutions to problems where other methods may fail. One drawback to NEAT is the importance of hyperparameter tuning (e.g. mutation rate, population size,

initial weight values, etc.), since these values have a strong effect on convergence and quality of the final solutions.

Another drawback to NEAT is that the solution is not guaranteed to be locally optimal. One avenue for future work could be to look into using a gradient-based optimizer on the final nominal trajectory to "clean up" the final result. The optimizer could either minimize propellant usage, or it could minimize a measure of sensitivity along the trajectory. Such a measure could involve minimizing the amount that the Δv will change for a change in the state $\bar{x}$.

Genetic algorithms are powerful global search algorithms, but especially for high-dimension problems we see that the run time can become a limiting factor. With randomly initialized weights and biases, a neural network "knows" nothing about the problem and most of the computational effort is spent training the network to get in the neighborhood of the final solution, as opposed to finding the final solution itself. Thus, it would be beneficial if the network were already trained such that it could output a reasonable initial guess based on known trajectories. With a more reasonable starting point, the algorithm could then take less time to satisfy boundary conditions and convergence could be reached more rapidly. One such way to do this would be to train a network using supervised learning, and then pass that solution to a genetic algorithm.

When training a neural network on a stochastic problem such as the missed-thrust problem, there is no guarantee that it will produce suitable results in every case if it is tested only a few times. To get a more statistically representative metric, we can use Reliability-Based Design Optimization (RBDO) [20]. Using an RBDO formulation, a reliability constraint is introduced so that the probability of violating the original constraints is less than a specified value. For the current problems, such a formulation could be given by Equation (7.1), where $\mathrm{E}\left(m_{\mathrm{prop}}\right)$ is the expected value of the propellant mass required, $\delta\bar{x}$ are the constraint violations, $\delta\bar{x}_{\max}$ are the maximum allowable constraint violations, and ϵ is a value to measure the reliability (smaller is more reliable).

$$\begin{aligned} \min\ & J = \mathrm{E}\left(m_{\mathrm{prop}}\right) \\ \text{s.t. } & \mathrm{P}(\delta\bar{x} > \delta\bar{x}_{\max}) < \epsilon \end{aligned} \tag{7.1}$$

7.2 Supervised Learning

My work with supervised learning showed generally promising results. My work in Chapter 5 demonstrated that a neural-network-based controller is able to guide a spacecraft to the target destination in the presence of missed thrust with a 96% success rate when used in conjunction with a local optimizer. However, we see this method's problem dependency with Chapter 4's success rate of 58%. This problem dependency is more a reflection of the varying difficulty and sensitivities of different trajectory design problems, but it noted that these variations are shown in the success rate.

Previous work by other authors have demonstrated success rates approaching 100%, so a 58% success rate at first glance may seem concerning [94]–[97]. However, these previous studies have all used a 2D environment, and in some of the studies the disturbances were fairly small compared to what may be expected from the model described in section 2.3.1. Furthermore, in LaFarge et al., the authors had access to HPC resources that, on the computer used for the simulations in this dissertation, would have taken over two years to match. Given the available computing resources and the increase of problem complexity, the drop in success rate makes sense while still providing evidence that a neural-network-based controller can guide a spacecraft in the presence of missed thrust.

There are many opportunities for future work in this area. In this dissertation I considered inputs in the inertial and MEE frames, and I did not do a direct comparison between the two. It would be useful to perform an experiment comparing a network's performance when given inputs in inertial, polar, MEE, COE, and possibly other coordinates to see if the choice of coordinate system affects a neural network's ability to learn. Similarly, the output vector was only interpreted in the LVLH coordinate system. Future study could examine VNC, inertial, or perifocal coordinate systems. There could also be a direct comparison between 2D vs. 3D as well as elliptical vs. circular orbits to see the effect of adding each additional variable into consideration. A comparison between different types of inputs, such as taking the difference between target and current state instead of each individually, or one or the other, could be useful. Furthermore, the network could predict other values such as switching function instead of or in addition to thrust and/or costates. Additionally, duplicating this

work with additional computing resources to generate larger data sets and/or train for longer could provide stronger results. Finally, a deeper analysis into where the missed-thrust events occur along the trajectories for the failed cases could provide further insight into the problem.

7.3 Comparison

After using NEAT and supervised learning to train neural networks for the same problem, we can see that supervised learning was much more successful. Supervised learning in cases without missed thrust and utilizing an optimizer was successful in 100% of cases whereas missed thrust with the optimizer was 96%. NEAT did not reach the target in any cases even without taking into account missed thrust. Utilizing the brute force method, a method not dependent on neural networks, yielded a success rate of 68.5%. From these results we can easily see that training a neural network with supervised learning is a more useful method than training with NEAT for this problem.

NEAT was unable to successfully train a neural network to guide a spacecraft to the target even with a large variety of hyperparameter combinations. This could be due to a limitation with small neural networks and may prove to have more utility with capabilities that allow for training larger networks.

A drawback to both of the training methods with respect to the brute force method is that neural networks will only give a valid output when the inputs are in the range of what has been seen during training. If an input is given that is outside of the training data input space, the network will most likely give an erroneous or nonsensical answer. This could mean, for example, if the launch or arrival dates are pushed outside of the windows seen in training, the mass of the spacecraft is higher or lower than it normally would be at a certain position compared to in training, or if the spacecraft drifts too far away from the nominal trajectory. The brute force method does not depend on any previous training, so changing the problem parameters will not affect the method's effectiveness (aside from changes in problem sensitivity). While a neural network trained on a launch window of $\pm$20 days could handle a tightened launch window of $\pm$10 days, it would not be expected to work well on an expanded launch window of $\pm$40 days for the dates between $\pm$20-40 days.

7.4 Hyperbolic Abort

As astronauts inevitably travel away from Earth on their way to Mars and beyond, they must be placed into a hyperbolic orbit. In the scenario where the astronauts must rendezvous with another spacecraft on a hyperbolic orbit, there is an inherent risk to the crew and to the mission. It is therefore desirable to drive the likelihood of this risk down as much as possible. Landau and Longuski investigated the risks of hyperbolic rendezvous along the outbound leg and found that, while most errors had viable recovery strategies, a failure during the maneuver to transfer to the hyperbolic orbit did not have a successful recovery strategy [72]. The analysis presented in this paper provides a new strategy for hyperbolic orbital insertion that seeks to minimize the risk while accounting for the Δv of an abort maneuver. It is shown that performing orbital insertion along the inbound leg of the target hyperbolic orbit offers much cheaper abort options when compared to orbital insertion along the outbound leg. This insertion provides the spacecraft and crew with a much longer opportunity to abort, allowing them to work through issues as they arise. Furthermore, using the tangent taxi orbit case is simpler than the 3-burn strategy along either the inbound or the outbound leg since the tangent taxi orbit eliminates the transfer orbit altogether. This strategy reduces the number of maneuvers simplifying the process from an operations perspective and decreasing the overall risk involved.

REFERENCES

[1] T. Imken, T. M. Randolph, M. DInicola, and A. K. Nicholas, "Modeling Spacecraft Safe Mode Events," in *IEEE Aerospace Conference Proceedings*, Big Sky, MT, 2018, pp. 1–13, ISBN: 9781538620144. DOI: 10.1109/AERO.2018.8396383.

[2] J. T. Olympio, "Designing Robust Low-Thrust Interplanetary Trajectories Subject to One Temporary Engine Failure," in *Advances in the Astronautical Sciences*, 2010, pp. 1071–1089, ISBN: 9780877035602.

[3] J. T. Olympio and C. H. Yam, "Deterministic Method for Space Trajectory Design with Mission Margin Constraints," *61st International Astronautical Congress 2010, IAC 2010*, 2010.

[4] F. E. Laipert and J. M. Longuski, "Automated Missed-Thrust Propellant Margin Analysis for Low-Thrust Trajectories," *Journal of Spacecraft and Rockets*, vol. 52, no. 4, pp. 1135–1143, 2015, ISSN: 15336794. DOI: 10.2514/1.A33264.

[5] F. E. Laipert and T. Imken, "A Monte Carlo Approach to Measuring Trajectory Performance Subject to Missed Thrust," in *Advances in the Astronautical Sciences*, Kissimmee, FL, 2018, pp. 1–11. DOI: 10.2514/6.2018-0966.

[6] P. Muñoz, "Missed-Thrust Analysis of BepiColombo's Interplanetary Transfer to Mercury Orbit," in *Advances in the Astronautical Sciences*, Portland, ME, 2019, pp. 1–14.

[7] N. Ozaki, S. Campagnola, R. Funase, and C. H. Yam, "Stochastic Differential Dynamic Programming With Unscented Transform for Low-Thrust Trajectory Design," *Journal of Guidance, Control, and Dynamics*, vol. 41, no. 2, pp. 377–387, 2018. DOI: 10.2514/1.G002367.

[8] D. Izzo, C. Sprague, and D. Tailor, "Machine Learning and Evolutionary Techniques in Interplanetary Trajectory Design," in *Modeling and Optimization in Space Engineering: State of the Art and New Challenges*, Springer International Publishing, Mar. 2019, pp. 191–210, ISBN: 978-3-030-10501-3. DOI: 10.1007/978-3-030-10501-3{_}8. [Online]. Available: http://arxiv.org/abs/1802.00180.

[9] A. Rubinsztejn, R. Sood, and F. E. Laipert, "Neural Network Based Optimal Control : Resilience To Missed Thrust Events," in *Advances in the Astronautical Sciences*, Maui, HI, 2019, pp. 1–14.

[10] B. Dachwald and W. Seboldt, "Optimization of Interplanetary Rendezvous Trajectories for Solar Sailcraft Using a Neurocontroller," in *Advances in the Astronautical Sciences*, Monterey, CA, 2002, pp. 1–8, ISBN: 9781624101243. DOI: 10.2514/6.2002-4989.

[11] B. Dachwald, "Evolutionary Neurocontrol: A Smart Method for Global Optimization of Low-Thrust Trajetories," in *Advances in the Astronautical Sciences*, 2004, ISBN: 1563477149. DOI: doi:10.2514/6.2004-5405.

[12] B. Dachwald, "Optimization of very-low-thrust trajectories using evolutionary neurocontrol," *Acta Astronautica*, vol. 57, no. 2, pp. 175–185, 2005. DOI: 10.1016/j.actaastro.2005.03.004.

[13] I. Carnelli, B. Dachwald, and M. Vasile, "Evolutionary Neurocontrol: A Novel Method for Low-Thrust Gravity-Assist Trajectory Optimization," *Journal of Guidance, Control, and Dynamics*, vol. 32, no. 2, pp. 615–624, 2009, ISSN: 07315090. DOI: 10.2514/1.32633.

[14] Y. C. Shin and C. Xu, *Intelligent Systems: Modeling, Optimization, and Control*. CRC Press, 2017, pp. 1–433, ISBN: 9781420051773. DOI: 10.1201/9781420051773.

[15] K. Hornik, M. Stinchcombe, and H. White, "Multilayer Feedforward Networks are Universal Approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989, ISSN: 08936080. DOI: 10.1016/0893-6080(89)90020-8.

[16] A. Browne, L. Niklasson, M. Bodén, *et al.*, *Neural Network Perspectives on Cognition and Adaptive Robotics*. Institute of Physics Publishing, 1997, p. 270.

[17] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient BackProp," in *Neural Networks: Tricks of the Trade*, 2nd ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48, ISBN: 978-3-642-35289-8.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. DOI: 10.1038/323533a0. [Online]. Available: https://doi.org/10.1038/323533a0.

[19] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks Through Augmenting Topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. DOI: 10.1162/106365602320169811.

[20] J. S. Arora, *Introduction to Design Optimization*, 4th. Academic Press, 2017. DOI: 10.1016/b978-0-12-800806-5.00001-9.

[21] G. N. Vanderplaats, *Multidiscipline Design Optimization*, 1st. Monterey, CA: Vanderplaats Research and Development, Inc., 2007, ISBN: 0-944956-04-1.

[22] M. A. Vavrina, "A Hybrid Genetic Algorithm Approach To Global Low-Thrust Trajectory Optimization," Ph.D. dissertation, Purdue University, 2008.

[23] M. A. Vavrina and K. C. Howell, "Global Low-Thrust Trajectory Optimization Through Hybridization of a Genetic Algorithm and a Direct Method," *Advances in the Astronautical Sciences*, pp. 1–27, 2008. DOI: 10.2514/6.2008-6614.

[24] C. M. Chilan and B. A. Conway, "Automated Design of Multiphase Space Missions Using Hybrid Optimal Control," *Journal of Guidance, Control, and Dynamics*, vol. 36, no. 5, pp. 1410–1424, 2013, ISSN: 15333884. DOI: 10.2514/1.58766.

[25] J. A. Englander, B. A. Conway, and T. Williams, "Automated Mission Planning via Evolutionary Algorithms," *Journal of Guidance, Control, and Dynamics*, vol. 35, no. 6, pp. 1878–1887, 2012, ISSN: 15333884. DOI: 10.2514/1.54101.

[26] J. A. Englander, B. A. Conway, and T. Williams, "Automated Interplanetary Trajectory Planning," in *Advances in the Astronautical Sciences*, 2012, ISBN: 9781624101823. DOI: 10.2514/6.2012-4517.

[27] J. A. Englander, M. A. Vavrina, and D. W. Hinckley, "Global Optimization of Low-Thrust Interplanetary Trajectories Subject to Operational Constraints," *Advances in the Astronautical Sciences*, vol. 158, pp. 179–198, 2016, ISSN: 00653438.

[28] M. A. Vavrina, J. A. Englander, and A. R. Ghosh, "Coupled Low-Thrust Trajectory and Systems Optimization Via Multi-Objective Hybrid Optimal Control," in *Advances in the Astronautical Sciences*, 2015.

[29] J. A. Englander, M. A. Vavrina, and A. R. Ghosh, "Multi-Objective Hybrid Optimal Control for Multiple-Flyby Low-Thrust Mission Design," in *Advances in the Astronautical Sciences*, 2015.

[30] C. H. Yam, D. Izzo, and D. D. Lorenzo, "Low-thrust trajectory design as a constrained global optimization problem," *Journal of Aerospace Engineering*, vol. 225, no. 11, pp. 1243–1251, 2011. DOI: 10.1177/0954410011401686.

[31] G. A. Rauwolf and V. L. Coverstone-Carroll, "Near-Optimal Low-Thrust Orbit Transfers Generated by a Genetic Algorithm," *Journal of Spacecraft and Rockets*, vol. 33, no. 6, pp. 859–862, 1996, ISSN: 15336794. DOI: 10.2514/3.26850.

[32] A. Gad and O. Abdelkhalik, "Hidden Genes Genetic Algorithm for Multi-Gravity-Assist Trajectories Optimization," *Journal of Spacecraft and Rockets*, vol. 48, no. 4, pp. 629–641, 2011, ISSN: 15336794. DOI: 10.2514/1.52642.

[33] C. H. Yam, F. Biscani, and D. Izzo, "Global Optimization of Low-Thrust Trajectories via Impulsive Delta-V Transcription," in *27th International Symposium on Space Technology and Science*, 2009. [Online]. Available: http://www.esa.int/gsp/ACT/doc/MAD/pub/ACT-RPR-MAD-2009-GlobalOptLowThrust.pdf.

[34] B. Dachwald and W. Seboldt, "Multiple Near-Earth Asteroid Rendezvous and Sample Return Using First Generation Solar Sailcraft," *Acta Astronautica*, vol. 57, no. 11, pp. 864–875, 2005. DOI: 10.1016/j.actaastro.2005.04.012.

[35] B. Dachwald, "Optimization of Interplanetary Solar Sailcraft Trajectories Using Evolutionary Neurocontrol," *Journal of Guidance, Control, and Dynamics*, vol. 27, no. 1, pp. 66–72, 2004.

[36] A. Ohndorf, B. Dachwald, and B. Gill, "Optimization of Low-Thrust Earth-Moon Transfers Using Evolutionary Neurocontrol," in *2009 IEEE Congress on Evolutionary Computation, CEC 2009*, 2009, pp. 358–364, ISBN: 9781424429592. DOI: 10.1109/CEC.2009.4982969.

[37] D. Y. Oh, D. F. Landau, T. M. Randolph, *et al.*, "Analysis of System Margins on Deep Space Missions Using Solar Electric Propulsion," in *44th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, Hartford, CT, 2008, pp. 1–29. DOI: 10.2514/6.2008-5286.

[38] M. D. Rayman, T. C. Fraschetti, C. A. Raymond, and C. T. Russell, "Coupling of system resource margins through the use of electric propulsion: Implications in preparing for the Dawn mission to Ceres and Vesta," *Acta Astronautica*, vol. 60, no. 10-11, pp. 930–938, May 2007, ISSN: 00945765. DOI: 10.1016/j.actaastro.2006.11.012.

[39] A. Madni, W. Hart, T. Imken, D. Y. Oh, and J. S. Snyder, "Missed Thrust Requirements for Psyche Mission," *AIAA Propulsion and Energy 2020 Forum*, no. 3120, pp. 1–14, 2020. DOI: 10.2514/6.2020-3608.

[40] S. L. Mccarty and D. J. Grebow, "Missed Thrust Analysis and Design for Low Thrust Cislunar Transfers," *AAS/AIAA Astrodynamics Specialist Conference*, pp. 1–16, 2020.

[41] V. Arya, E. Taheri, and J. L. Junkins, "Low-Thrust Gravity-Assist Trajectory Design Using Optimal Multimode Propulsion Models," *Journal of Guidance, Control, and Dynamics*, vol. 44, no. 7, pp. 1280–1294, 2021, ISSN: 15333884. DOI: 10.2514/1.G005750.

[42] O. Çelik, D. A. Dei Tos, T. Yamamoto, N. Ozaki, Y. Kawakatsu, and C. H. Yam, "Multiple-Target Low-Thrust Interplanetary Trajectory of DESTINY+," *Journal of Spacecraft and Rockets*, vol. 58, no. 3, pp. 830–847, 2021, ISSN: 0022-4650. DOI: 10.2514/1.a34804.

[43] R. Jehn, D. Garcia Yarnoz, J. Schoenmaekers, and V. Companys, "Trajectory Design for BepiColombo Based on Navigation Requirements," *Journal of Aerospace Engineering, Sciences and Applications*, vol. 4, no. 1, pp. 1–9, 2012, ISSN: 2236577X. DOI: 10.7446/jaesa.0401.01.

[44] D. Garcia Yarnoz, R. Jehn, and M. Croon, "Interplanetary navigation along the low-thrust trajectory of BepiColombo," *Acta Astronautica*, vol. 59, no. 1-5, pp. 284–293, 2006, ISSN: 00945765. DOI: 10.1016/j.actaastro.2006.02.028.

[45] D. A. Vallado, *Fundamentals of Astrodynamics and Applications*, 4th ed. Microcosm Press, 2013, p. 1108.

[46] D. M. Goebel and I. Katz, *Fundamentals of Electric Propulsion: Ion and Hall Thrusters*. 2008, ISBN: 9780470429273. DOI: 10.1002/9780470436448.

[47] I. Newton, *Philosophiae Naturalis Principia Mathematica*. London, 1687.

[48] J. M. Longuski, J. J. Guzman, and J. E. Prussing, *Optimal Control with Aerospace Applications*. El Segundo, CA: Microcosm Press and Springer, 2014, p. 273, ISBN: 978-1-4614-8944-3. DOI: 10.1007/978-1-4614-8945-0.

[49] T. F. Coleman and Y. Li, "An interior trust region approach for nonlinear minimization subject to bounds," *SIAM Journal on Optimization*, vol. 6, no. 2, pp. 418–445, 1996, ISSN: 10526234. DOI: 10.1137/0806023.

[50] L. F. Shampine and J. Kierzenka, "A BVP Solver based on residual control and the MATLAB PSE," *ACM Transactions on Mathematical Software*, vol. 27, no. 3, pp. 299–316, 2001.

[51] J. J. Moré, "The Levenberg-Marquardt algorithm: Implementation and theory," in *Numerical Analysis*, G. A. Watson, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 105–116, ISBN: 978-3-540-35972-2. DOI: 10.1007/BFb0067700.

[52] M. J. H. Walker, B. Ireland, and J. Owens, "A Set of Modified Equinoctial Orbit Elements," *Celestial Mechanics*, vol. 36, no. 4, pp. 409–419, 1985. DOI: 10.1007/BF01227493.

[53] E. Taheri, I. Kolmanovsky, and E. Atkins, "Enhanced Smoothing Technique for Indirect Optimization of Minimum-Fuel Low-Thrust Trajectories," *Journal of Guidance, Control, and Dynamics*, vol. 39, no. 11, pp. 2500–2511, 2016, ISSN: 15333884. DOI: 10.2514/1.G000379.

[54] J. T. Betts, "Survey of Numerical Methods for Trajectory Optimization," *Journal of Guidance, Control, and Dynamics*, vol. 21, no. 2, pp. 193–207, 1998, ISSN: 15333884. DOI: 10.2514/2.4231.

[55] I. Carnelli, B. Dachwald, and M. Vasile, "Evolutionary Neurocontrol: A Novel Method for Low-Thrust Gravity-Assist Trajectory Optimization," *Journal of Guidance, Control, and Dynamics*, vol. 32, no. 2, pp. 616–625, 2009, ISSN: 0731-5090. DOI: 10.2514/1.32633. [Online]. Available: http://arc.aiaa.org/doi/10.2514/1.32633.

[56] D. Whitley, S. Dominic, R. Das, and C. Anderson, "Genetic Reinforcement Learning for Neurocontrol Problems," *Machine Learning*, vol. 13, pp. 259–284, 1993.

[57] J. Kennedy and R. Eberhard, "Particle Swarm Optimization," in *Proceedings of the IEEE International Conference on Neural Networks*, Perth, Australia, 1995, pp. 1942–1945.

[58] E. Mezura-Montes and C. A. Coello Coello, "Constraint-Handling in Nature-Inspired Numerical Optimization: Past, Present, and Future.," *Swarm and Evolutionary Computation*, pp. 173–194, 2011.

[59] M. E. Pederson, "Good Parameters for Particle Swarm Optimization," in *Hvass Laboratories*, Luxembourg, 2010.

[60] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[61] J. L. Elman, "Finding Structure in Time," *Cognitive Science*, vol. 14, pp. 179–211, 1990.

[62] M. T. Hagan and M. Menhaj, "Training Feedforward Networks with the Marquardt Algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994.

[63] F. M. Sturms Jr., "Polynomial Expressions for Planetary Equators and Orbit Elements With Respect to the Mean 1950.0 Coordinate System," Jet Propulsion Laboratory, Pasadena, California, Tech. Rep., 1971.

[64] E. A. Williams and W. A. Crossley, "Empirically-Derived Population Size and Mutation Rate Guidelines for a Genetic Algorithm with Uniform Crossover," *Soft Computing in Engineering Design and Manufacturing*, 1998.

[65] C. H. Yam, "Design of Missions To the Outer Planets and Optimization of Low-Thrust, Gravity-Assist Trajectories Via Reduced Parameterization," Ph.D. dissertation, Purdue University, 2008, p. 176, ISBN: 0115629203.

[66] R. W. Conversano, R. R. Hofer, M. J. Sekerak, H. Kamhawi, and P. Y. Peterson, "Performance Comparison of the 12.5 kW HERMeS Hall Thruster Technology Demonstration Units," in *52nd AIAA/SAE/ASEE Joint Propulsion Conference*, Salt Lake City, UT: American Institute of Aeronautics and Astronautics Inc, AIAA, 2016, ISBN: 9781624104060. DOI: 10.2514/6.2016-4827.

[67] C. H. Acton, "Ancillary data services of NASA's Navigation and Ancillary Information Facility," *Planetary and Space Science*, vol. 44, no. 1 SPEC. ISS. Pp. 65–70, 1996, ISSN: 00320633. DOI: 10.1016/0032-0633(95)00107-7.

[68] C. H. Acton, N. Bachman, B. Semenov, and E. Wright, "A look towards the future in the handling of space science mission geometry," *Planetary and Space Science*, vol. 150, no. February 2017, pp. 9–12, 2018, ISSN: 00320633. DOI: 10.1016/j.pss.2017.02.013. [Online]. Available: https://doi.org/10.1016/j.pss.2017.02.013.

[69] J. E. Prussing, *Optimal Spacecraft Trajectories*, 1st ed. Oxford, UK: Oxford University Press, 2018, ISBN: 978-0-19-881111-4.

[70] R. Bertrand and R. Epenoy, "New smoothing techniques for solving bang-bang optimal control problems—numerical results and statistical interpretation," *Optimal Control Applications and Methods*, vol. 23, no. 4, pp. 171–197, 2002, ISSN: 01432087. DOI: 10.1002/oca.709.

[71] C. M. Bishop, *Neural Networks for Pattern Recognition*, 1st ed. Oxford University Press, 1996, ISBN: 978-0198538646.

[72] D. F. Landau and J. M. Longuski, "Guidance Strategy for Hyperbolic Rendezvous," *Journal of Guidance, Control, and Dynamics*, vol. 30, no. 4, pp. 1209–1213, 2007, ISSN: 15333884. DOI: 10.2514/1.30071.

[73] T. T. McConaghy, R. P. Russell, and J. M. Longuski, "Toward a Standard Nomenclature for Earth-Mars Cycler Trajectories," *Journal of Spacecraft and Rockets*, vol. 42, pp. 694–698, 2005.

[74] D. F. Landau and J. M. Longuski, "Mars Exploration Via Earth-Mars Semi-Cyclers," in *Advances in the Astronautical Sciences*, AAS, 2005.

[75] R. P. Russell and C. A. Ocampo, "Optimization of a Broad Class of Ephemeris Model Earth-Mars Cyclers," *Journal of Guidance, Control, and Dynamics*, vol. 29, pp. 354–367, 2006.

[76] D. V. Byrnes, J. M. Longuski, and B. Aldrin, "Cycler Orbit Between Earth and Mars," *Journal of Spacecraft and Rockets*, vol. 30, no. 3, pp. 334–336, 1993, ISSN: 00224650. DOI: 10.2514/3.25519.

[77] D. V. Byrnes, "Analysis of Various Two Synodic Period Earth-Mars Cycler Trajectories," in *Advances in the Astronautical Sciences*, Monterey, CA, 2002. DOI: 10.2514/6.2002-4423.

[78] T. T. McConaghy, D. F. Landau, C. H. Yam, and J. M. Longuski, "Notable two-synodic-period earth-mars cycler," *Journal of Spacecraft and Rockets*, vol. 43, no. 2, pp. 456–465, 2006, ISSN: 15336794. DOI: 10.2514/1.15215.

[79] R. P. Russell and C. A. Ocampo, "Systematic Method for Constructing Earth-Mars Cyclers Using Free-Return Trajectories," *Journal of Guidance, Control, and Dynamics*, vol. 27, pp. 321–335, 2004.

[80] K. J. Chen, B. A. Rogers, M. Okutsu, D. F. Landau, and J. M. Longuski, "Low-Thrust Aldrin Cycler with Reduced Encounter Velocities," *Journal of Spacecraft and Rockets*, vol. 49, no. 5, pp. 955–961, 2012, ISSN: 15336794. DOI: 10.2514/1.A32109.

[81] D. F. Landau and J. M. Longuski, "Comparative Assessment of Human-Mars-Mission Technologies and Architectures," *Acta Astronautica*, vol. 65, no. 7, pp. 893–911, 2009. DOI: https://doi.org/10.1016/j.actaastro.2009.02.008.

[82] S. Ross, "A Systematic Approach to the Study of Non-Stop Interplanetary Round Trips," in *Advances in the Astronautical Sciences*, American Astronautical Society, 1963, p. 6.

[83] W. M. Hollister, "Castles in Space," *Astronautica Acta*, vol. 14, no. 2, pp. 311–315, 1969.

[84] W. M. Hollister, "Periodic Orbits for Interplanetary Flight," *Journal of Spacecraft*, vol. 6, pp. 366–369, 1969. DOI: 10.2514/3.29664.

[85] W. M. Hollister and M. D. Menning, "Interplanetary Orbits for Multiple Swingby Missions," in *Advances in the Astronautical Sciences*, American Institute of Aeronautics and Astronautics, Princeton, NJ, 1969. DOI: 10.2514/6.1969-931.

[86] C. S. Rall and W. M. Hollister, "Free-Fall Periodic Orbits Connecting Earth and Mars," *Journal of Spacecraft and Rockets*, vol. 8, no. 10, pp. 1017–1020, 1971. DOI: 10.2514/3.59763.

[87] A. L. Friedlander, J. C. Niehoff, D. V. Byrnes, and J. M. Longuski, "Circulating Transportation Orbits Between Earth and Mars," in *Advances in the Astronautical Sciences*, American Institute of Aeronautics and Astronautics, 1986.

[88] J. C. Niehoff, "Pathways to Mars: New Trajectory Opportunities," in *Advances in the Astronautical Sciences*, American Astronautical Society, 1986.

[89] B. Aldrin, "Cyclic Trajectories Concepts," Science Applications International Corp., Aerospace Systems Group, Hermosa Beach, CA, Tech. Rep., 1985.

[90] K. T. Nock and A. L. Friedlander, "Elements of a Mars Transportation System," *Acta Astronautica*, vol. 15, no. 6, pp. 505–522, 1987. DOI: 10.1016/0094-5765(87)90189-5.

[91] P. A. Penzo and K. T. Nock, "Hyperbolic Rendezvous for Earth-Mars Cycler Missions," *Advances in the Astronautical Sciences*, vol. 112, no. AAS 02-162, pp. 763–772, 2002.

[92] R. Jedrey, D. F. Landau, and R. Whitley, "Hyperbolic Rendezvous At Mars: Risk Assessments and Mitigation Strategies," in *Advances in the Astronautical Sciences*, AAS, 2015, pp. 4325–4346.

[93] J. O. Burns, D. A. Kring, J. B. Hopkins, S. Norris, T. J. W. Lazio, and J. Kasper, "A lunar L2-Farside exploration and science mission concept with the Orion Multi-Purpose Crew Vehicle and a teleoperated lander/rover," *Advances in Space Research*, vol. 52, no. 2, pp. 306–320, 2013, ISSN: 02731177. DOI: 10.1016/j.asr.2012.11.016. [Online]. Available: http://dx.doi.org/10.1016/j.asr.2012.11.016.

[94] N. B. LaFarge, D. Miller, K. C. Howell, and R. Linares, "Autonomous closed-loop guidance using reinforcement learning in a low-thrust, multi-body dynamical environment," *Acta Astronautica*, vol. 186, no. May, pp. 1–23, 2021, ISSN: 00945765. DOI: 10.1016/j.actaastro.2021.05.014. [Online]. Available: https://doi.org/10.1016/j.actaastro.2021.05.014.

[95] A. Rubinsztejn, R. Sood, and F. E. Laipert, "Neural Network optimal control in astrodynamics: Application to the missed thrust problem," *Acta Astronautica*, vol. 176, pp. 192–203, 2020. DOI: https://doi.org/10.1016/j.actaastro.2020.05.027.

[96] A. Rubinsztejn, R. Sood, and F. E. Laipert, "Neural Network Based Optimal Control: Resilience To Missed Thrust Events For Long Duration Transfers," in *Advances in the Astronautical Sciences*, vol. 171, 2020, pp. 3585–3598, ISBN: 9780877036654.

[97] A. Zavoli and L. Federici, "Reinforcement learning for robust trajectory design of interplanetary missions," *Journal of Guidance, Control, and Dynamics*, vol. 44, no. 8, pp. 1440–1453, 2021, ISSN: 15333884. DOI: 10.2514/1.G005794.

VITA

Paul received his BS in Aeronautical and Astronautical Engineering from Purdue University in 2016, after which he joined Prof. Jim Longuski's Advanced Astrodynamics Concepts research group. During his undergraduate studies, Paul helped restart the Purdue Lunabotics club and helped lead the team to a 2nd place finish in NASA's 2016 Robotic Mining Competition. Paul also helped start and was the project manager for the Purdue Hyperloop Design Team, which ended up being 1 of only 7 teams to pass all qualification tests out of over 1000 initial applicants for SpaceX's 1st Hyperloop Pod Competition.

In 2014, Paul was a member of the NASA Glenn Research Center's Space Academy, where he worked with Dr. Geoffrey Landis and Dr. Al Hepp on a concept study for a CubeSat testbed for photovoltaic cells, as well as a concept study for a self-drilling "mole" that could dig beneath Mars's surface. In 2015, Paul interned at NASA Jet Propulsion Laboratory with Dr. Oleg Sindiy and Dr. Charles Budney working on Model-Based Systems Engineering (MBSE) for the Asteroid Redirect Robotic Mission. In 2017, Paul returned to JPL, this time working with Dr. Anastassios Petropoulos on an astrodynamics tool called Bender 2 which is used for rapid multi-gravity-assist analysis. In 2018, Paul interned at Dynetics under the guidance of Dr. Lee Coduti and Kevin Albarado working on a machine-learning-based agent in simulations to control swarms of UAVs to cooperatively overcome dynamically changing targets. Paul returned to Dynetics in 2019 to help develop an in-house machine learning tool that could be used by other engineers without in-depth machine learning expertise.

Following graduation, Paul is moving to Ann Arbor, MI where his wife is starting as a medical resident in otolaryngology.