9-1-2018

Application of Spectral Solution and Neural Network Techniques in Plasma Modeling for Electric Propulsion

Joseph R. Whitman

APPLICATION OF SPECTRAL SOLUTION AND NEURAL NETWORK TECHNIQUES IN PLASMA MODELING FOR ELECTRIC PROPULSION

THESIS

Joseph Whitman, Civilian

AFIT-ENY-MS-18-S-076

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENY-MS-18-S-076

APPLICATION OF SPECTRAL SOLUTION AND NEURAL NETWORK TECHNIQUES IN PLASMA MODELING FOR ELECTRIC PROPULSION

THESIS

Presented to the Faculty

Department of Aeronautics & Astronautics

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science

Joseph Whitman, BS

Civilian

July 2018

AFIT-ENY-MS-18-S-076

APPLICATION OF SPECTRAL SOLUTION AND NEURAL NETWORK TECHNIQUES IN PLASMA MODELING FOR ELECTRIC PROPULSION

THESIS

Joseph Whitman, BS
Civilian

Committee Membership:

Dr. Carl R. Hartsfield, Ph.D.
Chair

Dr. Justin W. Koo, Ph.D.
Member

Dr. Tabitha Dodson, Ph.D.
Member

Abstract

A solver for Poisson's equation was developed using the Radix-2 FFT method first invented by Carl Friedrich Gauss. Its performance was characterized using simulated data and identical boundary conditions to those found in a Hall Effect Thruster. The characterization showed errors below machine-zero with noise-free data, and above 20% noise-to-signal strength, the error increased linearly with the noise. This solver can be implemented into AFRL's plasma simulator, the Thermophysics Universal Research Framework (TURF) and used to quickly and accurately compute the electric field based on charge distributions. The validity of a machine learning approach and data-based complex system modeling approach was demonstrated. To this end, several multilayer perceptrons were created and validated against AFRL-provided Hall Thruster test data, with two networks showing mean error below 1% and standard deviations below 10%. These results, while not ready for implementation as a replacement for lookup tables, strongly suggest paths for future work and the development of networks that would be acceptable in such a role, saving both RAM space and time in plasma simulations.

Acknowledgements

I would not be here except for the help and support from so many people, and though it would be impossible to thank everyone who deserves it, there are some who deserve special acknowledgement and gratitude. I would first like to thank my advisor, Dr. Carl Hartsfield, my research sponsor, Dr. Justin Koo, and Dr. Tabitha Dodson, who together make up my thesis committee. Dr. Koo's staff at AFRL, in particular Mr. David Bilyeu, have provided assistance in understanding an extremely complex code base. I would like to extend special gratitude to Mr. Jacob Henderson, Matthew Jahns, Darryl Masson, and Levi Linville for teaching me so much computer science over time, to say nothing of staying up long hours to explain particularly arcane rules of C++ to me. I would not have been capable of attempting this without your help. To all those friends who put up with lack of free time while I pursued this master's degree, thank you. I owe particular thanks to Ms. Emily Murch, Mr. Jared Loving, and Mr. Nick Savino, who listened to my frustrations patiently time and time (and time, and time, and time...) again, and who kept me on task even when I began doubting myself. This would have been so much harder alone. To my mother and father, who raised me to value intellectual curiosity and to be persistent in my goals above all else, and to Mr. Richard Gargas, who would never have let me provide less than my best. Finally, and especially, to Arika. You've been a constant on the most important parts of my journey, and I cannot imagine more without you.

Joseph Whitman

Table of Contents

Page

Abstract iv

Acknowledgements v

List of Figures viii

List of Tables x

1. Introduction 1

    1.1 Background 1
    1.2 Motivation 2
    1.3 Scope and Objectives 3

2. Theory and Background 5

    2.1 Plasma Physics 5
        2.1.1 Plasma Basics 5
        2.1.2 Anomalous Electron Transport 10
        2.1.3 Cross-Field Mobility 12
        2.1.4 Ion-Acoustic Waves 13
        2.1.5 Electron-Ion Instability 14
        2.1.6 Beam-Cyclotron Waves 15
    2.2 Computational Theory and Mathematics 16
        2.2.1 Spectral Solvers 16
        2.2.2 Time-Memory Tradeoff 17
        2.2.3 Neural Networks 19

3. Experiment 29

    3.1 Plasma simulation 29
        3.1.1 Intended Simulation 29
        3.1.2 Problems Encountered 31
        3.1.3 Solver Characterizatiton 33
    3.2 Neural Network Model 34
        3.2.1 Network Toplogy 37
        3.2.2 IVB Mapping and Training Approach 37
        3.2.3 Code Validation 38

4. Results and Analysis 39

    4.1 Solver Characterization 39
        4.1.1 Performance with Increasingly Complex Waves and Transformation Lengths 39

Page

4.1.2 Noise Performance . . . 40
4.2 ANN Results . . . 42
4.2.1 Varying Neuron Numbers . . . 42
4.2.2 Data Quantity Effects . . . 53
4.2.3 Noise Effects . . . 56

5. Conclusion . . . 61

Appendix A. . . . 64

A.1 Solver Code . . . 64
A.1.1 ValidationFFT Header . . . 64
A.1.2 ValidationFFT Code . . . 65
A.2 Neural Network Code . . . 82
A.2.1 FCNetwork.py . . . 82
A.2.2 Master_Approximator.py . . . 87
Bibliography . . . 102

List of Figures

Figure Page

1. Electron-Ion stability based on λ [3, p. 178]. The first chart shows a stable dispersion relationship with all real roots; the second shows critical stability, with one real root at $\lambda = z$; the final plot indicates an unstable dispersion relationship with two complex conjugate roots. 16

2. Illustration of a Hall Thruster with the computational domain shown in pink as shown in Tran's thesis work [2, p. 16] 30

3. Error (Blue) and Machine-Zero (Red) vs. Number of Component Waves and Transformation Length 40

4. Introduction of any noise worsens performance above the noise floor of this solver, but the performance appears relatively constant with additional noise 41

5. Close-up of the noisy data errors. Note that the error level increases approximately linearly with the noise 41

6. 20-Neuron Results 44

7. 40-Neuron Results 45

8. 80-Neuron Results 45

9. 160-Neuron Results 46

10. 320-Neuron Results 46

11. 640-Neuron Results 47

12. 1280-Neuron Results 47

13. 20 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current 48

14. 40 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current 49

15. 80 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current 50

Figure Page

16. 160 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current 51

17. 320 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current 51

18. 640 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current 52

19. 1280 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current 52

20. Discharge Current vs. Discharge Voltage by Electromagnet Current 53

21. Effects of increasing training data quantity on total cost of the network 54

22. Effects of increasing training data quantity on miss cost of the network 55

23. Effects of increasing noise on total cost of a network trained on 240 datapoints. 57

24. Effects of increasing noise on miss cost of a network trained on 240 datapoints. 57

25. Effects of increasing noise on total cost of a network trained on 1200 datapoints 58

26. Effects of increasing noise on miss cost of a network trained on 1200 datapoints 58

27. Effects of increasing noise on total cost of a network trained on 2400 datapoints 59

28. Effects of increasing noise on miss cost of a network trained on 2400 datapoints 59

List of Tables

Table Page

1. Experimental ANN Statistics 43

APPLICATION OF SPECTRAL SOLUTION AND NEURAL NETWORK TECHNIQUES IN PLASMA MODELING FOR ELECTRIC PROPULSION

1. Introduction

1.1 Background

Although used primarily by the USSR during the Cold War, Hall Effect Thrusters (HETs) have seen greatly increased use since the fall of the Berlin Wall. Like their cousin, the gridded ion thruster (also called an ion drive), they provide low thrust over extremely long timeframes while using much less propellant mass than a chemical rocket would require for a comparable delta-V. Both systems function by ionizing a propellant gas such as Xenon, then accelerating the resulting ions through a powerful electrical field. The Hall thruster makes use of the eponymous Hall current and resulting electron bombardment as an effective ionization mechanism and to magnify the accelerating potential gradient, unlike the gridded ion thruster with physically separate discharge chamber and acceleration zones. The relative lack of moving parts and chemically benign propellants present few opportunities for thruster failure, and the quasineutral physics underlying HETs permits a higher practical thrust-to-power-required ratio and thrust density than the ion drive; when coupled with the advantages both systems share, HETs become a very attractive solution for any space mission that will demand small, repeated thrust events over a very long time [1, p. 325]. With improved technology, longer-lifetime spacecraft and longer-duration missions can be designed, saving the Air Force and U.S. taxpayer money.

1.2 Motivation

Hall thrusters are extremely relevant to the Air Force space mission. Satellites that are part of a constellation may need to shift position, while geosynchronous spacecraft require small station-keeping thrusts – both ideal tasks for the extremely reliable Hall thruster. Improved understanding of the physics underlying HETs will lead to improved thruster designs with higher efficiencies and longer lifetimes. Propellant is a spacecraft lifetime constraint, and expended propellant cannot (generally) be replenished on-orbit. Spacecraft are extremely expensive and meant for one-time use, so anything that extends the spacecraft lifetime has the potential to substantially reduce procurement costs by permitting a longer replacement interval. Improving the thruster performance may also reduce the power or mass required for use, in both cases freeing a key resource for other aspects of the spacecraft design.

The long lifetime characteristic of HETs makes them extremely expensive to test to failure for several reasons. HETs do not function outside of hard vacuum, therefore testing them requires an expensive vacuum chamber, maintaining the vacuum, and test monitoring at all times to prevent any minor issues from becoming a potentially data-invalidating or even catastrophic error. The thruster propellant must be replenished either through costly recovery equipment or through the ongoing cost of buying new propellant and exhausting the old. All of these conditions must be maintained for months or years at a time. The Air Force Institute of Technology (AFIT) has a 600-Watt thruster that would require an estimated $1M per year to test to failure in labor expenses alone. Simulating the thruster's behavior is a solution to these issues, but it is much more complex than even a more standard computational fluid dynamics (CFD) problem, since the plasma is governed by Maxwell's Laws in addition to the Navier-Stokes equations. Implementing the Navier-Stokes equations is further complicated by electric thrusters using rarified plasmas that must be treated

as individual particles rather than a continuum fluid.

While the advance of massively parallel computing has reduced the time required to execute a simulation, limitations on supercomputer node RAM and graphics card (GPU) RAM remain, making the use of lookup tables highly problematic. A five-dimensional input table as has been discussed within AFRL for recalling cross-field electron mobility using 64-bit floating point numbers with ten points along each dimension requires only 800 KB of RAM. Ten datapoints is too coarse for use in a system with such highly nonlinear dynamics and extreme sensitivity to input conditions unless only a tiny domain in each axis is desired. Systems in which tiny changes to input could result in entirely different end states, such as a plasma simulator, are usually described as chaotic. Increasing to 100 datapoints in each dimension, on the other hand, requires 80 GB of space, which is too much to be loaded into most server nodes' RAM and all but the most powerful of workstations. The lookup table may still be too coarse for effective use, and increasing the number of datapoints per dimension will only continue to increase the size, requiring advanced memory pagination techniques, causing a computational bottleneck by exchanging data between RAM and hard disk frequently. Using an artificial neural network provides the ability to approximate such functions to an arbitrarily high degree of accuracy, and it does so without necessarily incurring the memory penalty of a lookup table or the computational cost of exchanging data between disk and RAM.

1.3 Scope and Objectives

This work initially sought both to further the plasma simulator developed at the Air Force Research Lab (AFRL) called the Thermophysics Universal Research Framework (TURF) by adding a Fourier spectral solver for Poisson's equation and to demonstrate the applicability of artificial neural networks to plasma physics problems

as a replacement for large-scale lookup tables. The spectral solver would operate in the azimuthal direction (taking advantage of the periodic nature of a HET acceleration channel) to examine cross-field electron mobility in the radial-azimuthal plane following the work of LaFleur, Baalrud, and Chabert and duplicating the work performed by Tran in his thesis [2,7]. The results of the simulation would be compared with the results Tran achieved using a non-spectral solver. Unfortunately, persistent problems interfacing with TURF required scaling back this goal to developing a solver algorithm and quantifying its performance, leaving integration to future work. The second goal required the development of an artificial neural network and demonstrating its use by training it using data from a HET test that the AFRL conducted. No attempt was made to develop an optimal network as the goal was simply to demonstrate the concept. This effort was entirely successful.

2. Theory and Background

2.1 Plasma Physics

2.1.1 Plasma Basics.

2.1.1.1 Quasineutrality and Debye shielding.

A plasma is a collection of charged and neutral particles with sufficiently high energy that can cause atoms to ionize. Both ions and electrons behave as gases coupled to the local electric and magnetic fields, resulting in behavior that is vastly more complex than that of a simple gas. A plasma is defined as quasineutral when it has approximately as many electrons as ions. The quasineutrality condition is met for any "significantly large" volume of the plasma has no net charge, where "significantly large" means any distance greater than the Debye length, to be discussed below [3, p. 11]. Internal fields occur within the plasma once the quasineutrality condition is violated, and quasineutrality does not forbid local electromagnetic fields within length scales smaller than the Debye length.

The Debye length serves as an important scaling distance in plasma physics as it limits how deeply into a plasma any electrostatic field will penetrate– whether the source is internal to the bulk plasma or external to it. Without loss of generality, slightly displacing an ion within a quasineutral plasma will cause nearby electrons to move towards the new ion position, and nearby ions to move away slightly. This reaction will cause a local electron cloud surrounding the displaced ion, meaning that another particle or observer far away from the displaced ion will see no net change in potential from the displacement [4] [3, p. 8]. The potential of the electron cloud is then said to shield the potential from the test particle at a rate proportional to the exponential of reciprocal Debye length. The Debye length calculations are shown below, with Equation 1 used to calculate the Debye length of an individual species

within the plasma and Equation 2 used for the overall Debye length. Equation 3 shows mathematically the impact of Debye shielding on the potential at some distance r from a potential source. Within these equations, σ refers to an individual species within the plasma, T to the temperature in Kelvin, n to the number density of the species ($\#/m^3$), q to charge in Coulombs, ϵ_0 to free space permittivity, and κ to the Boltzmann Constant.

$$\lambda_\sigma = \left(\frac{\epsilon_0 \kappa T_\sigma}{n_{\sigma,0} q_\sigma^2} \right)^{1/2} \tag{1}$$

$$\lambda_\sigma = \left(\sum_\sigma \frac{1}{\lambda_\sigma^2} \right)^{-1/2} \tag{2}$$

$$\Phi = \Phi_0 \exp \frac{-r}{\lambda_D} \tag{3}$$

2.1.1.2 Poisson's Equation.

One of the most important equations in plasma physics is Poisson's equation as applied to electrostatics. This second-order partial differential equation (shown in Equation 4 below) directly relates the distribution of free charges, to the overall electric potential.

$$\nabla^2 \varphi = -\frac{\rho}{\epsilon} \tag{4}$$

In 4 the electric potential is represented by φ, the charge density is represented by ρ, and ϵ refers to the permittivity of the medium, in this case free space. The most straightforward and obvious approach to solving this equation numerically in 1D requires a tridiagonal solution, the mechanics of which include a matrix inversion. Matrix inversions are very computationally expensive, so other approaches may be

used if the boundary conditions permit. In this work, the boundary conditions are periodic in both the first and second derivative, making a solution relying on Fourier transforms and wave properties (referred to as a spectral solver) a very appealing approach. As will be discussed later, this work focused on a radial-azimuthal simulation and used the spectral solver in the azimuthal direction. In a HET, the acceleration channel is cylindrical, so periodic boundary conditions stem from 0 and 2πr being the same physical location.

2.1.1.3 Maxwellian Distribution.

As large collections of particles, plasmas are thermodynamically best described by some distribution function over their velocities. The Maxwell distribution is most commonly used and will be presented here, though it is not necessarily a valid assumption for plasmas with very few collisions, such as those not in thermal equilibrium [1]. Mathematically, a Maxwellian distribution in three dimensions is described in Equation 5 below [3, p. 51].

$$f_\sigma\left(\vec{x},\vec{v},t\right) = n_\sigma\left(\frac{m_\sigma}{2\pi\kappa T_\sigma}\right)^{3/2}\exp\left[-m_\sigma\frac{\left(\vec{v}-\vec{v_0}\right)^2}{2\kappa T_\sigma}\right] \tag{5}$$

In this distribution, v refers to the particle velocity while $\vec{v_0}$ refers the the electron drift relative to the lab reference frame.

2.1.1.4 Vlasov Equation and EM Acceleration.

The Vlasov equation (Equation 6) governs the rate of change of particles within an infinitesimal plasma element and provides a very powerful method for examining the change of plasma parameters [5]. The right side of the Vlasov equation accounts for the change in the distribution of particles due to collisions. Over timescales shorter than that required for a collision process to take place, the right hand side can be

treated as 0, giving a simpler version of the Vlasov equation.

$$\frac{\partial f_\sigma}{\partial t} + \vec{v} \cdot \nabla f_\sigma + \frac{\vec{F}}{m} \cdot \frac{\partial f_\sigma}{\partial \vec{v}} = \left(\frac{\partial f_\sigma}{\partial t}\right)_{collision} \tag{6}$$

The acceleration term in the Vlasov equation can be further expanded into the Lorentz equation (7), in which $\vec{E}$ and $\vec{B}$ indicate the electric and magnetic fields acting on a given particle. For the purposes of this work, the distribution function will be assumed Maxwellian in all cases as is convention.

$$\vec{F} = q\left(\vec{E} + \vec{v_{drift}} \times \vec{B}\right) \tag{7}$$

The $\vec{v} \times \vec{B}$ component of the Lorentz equation causes particles to move in a helical pattern around the magnetic field lines if a magnetic field is present. The frequency, radius, and period are referred to as the gyrofrequency, gyroradius (or Larmor radius, depending on the author), and gyroperiod, respectively. Another type of motion that occurs with both an applied electric and magnetic field is called the $\vec{E} \times \vec{B}$ drift, causing particles to move in a direction perpendicular to both fields [5, p. 8]. In a Hall thruster, this corresponds to an azimuthal drift described by Equation 8.

$$\vec{v}_{drift} = \frac{\vec{E} \times \vec{B}}{B^2} \tag{8}$$

Equation 7's azimuthal drift velocity, when crossed with the radial magnetic field, leads to a net axial electron acceleration, which affects the thrust force both directly as shown in 9 and indirectly via the electric potential as discussed in 2.1.1.2 [1, p.333]. These mechanisms drive the interest in electron behavior and lead to this work focusing on the electron motion.

$$T = \int \left(\vec{J_H} \times \vec{B}\right) dA \approx I_H B \tag{9}$$

In 9, T stands for the Hall thruster force, J_H for the Hall current density vector, I_H and the area for the cross-sectional area of the Hall current density.

2.1.1.5 Plasma Waves and Damping.

Perturbations in a plasma can propagate in the form of waves, as might be expected from a system of many coupled particles. One of the most basic characteristics of the plasma is the electron plasma frequency, shown in Equation 10. This frequency defines the characteristic timescale with which the electrons in the plasma react to perturbations. Though many forms of wave exist, those of interest to this work are primarily the electron plasma oscillations known as Langmuir waves or as electrostatic waves, which propagate only if the electrons have a distribution of velocities (unlike an idealized free-electron laser, for example) [5, p .10]. These waves have a dispersion relationship as shown in equations 10-13.

$$\omega_p = \left(\frac{4\pi n e^2}{m_e}\right) \tag{10}$$

$$\omega = \omega_r + i\omega_i \tag{11}$$

$$\omega_r = \sqrt{\omega_p^2 + \frac{3\kappa T k^2}{m}} \tag{12}$$

$$\omega_i = -\sqrt{\frac{\pi}{8}}\frac{\omega_p}{(k\lambda_{D,e})^3}\exp\left[-\frac{1}{2}(k\lambda_{D,e})^{-2} - \frac{3}{2}\right] \tag{13}$$

The imaginary term in Equation 13 is only valid when $\omega_i \ll \omega_r$ and when $\lambda_{D,e} = \left(\frac{kT_e}{4\pi n e^2}\right)^{1/2}$ [5, p.12]. Its presence is known as Landau, or collisionless, damping. Under some circumstances, Langmuir waves can become a form of electrostatic instability, or an instability associated with charges grouping together and separating in space. As will be discussed in 2.1.2 and 2.1.3, these instabilities are the phenomenon of interest

in the plasma simulations of this work.

2.1.1.6 Hall Thrusters.

-Hall thrusters rely on a typically annular thrust channel with an axial electric field and radial magnetic field to function. They are classified as electrostatic rather than electromagnetic thrusters because the ion's mass, on order of a million times greater for Xenon, causes a cyclotron radius much larger than the length of the thrust chamber and therefore are not affected significantly by the magnetic field's presence. The magnetic field is responsible for trapping the electrons and the generation of the Hall current from which the thruster gains its name. This azimuthal rotation of electrons along magnetic field lines causes most electrons to spend substantial time near the thruster exit plane in a region of very strong magnetic fields, as Tran mentions in his 2017 work [2]. Electrons may become freed from the magnetic field trap through several methods, including collisions with other particles or the channel walls (in a classical model), or plasma instabilities, and begin to drift towards the anode, referred to as cross-field drift as will be discussed in 2.1.3 [6]. Instabilities in plasmas have been an active area of research since the 1960s, though work initially focused on nuclear fusion and the fully magnetized and higher density plasmas more typical of fusion reactors.

2.1.2 Anomalous Electron Transport.

Hall thruster experiments and testing have demonstrated a higher than expected concentration of electrons near the anode. The cathode's position outside the acceleration channel requires that some electrons must travel across the magnetic field (in the axial direction), and as such this motion will be referred to as cross-field motion and anomalous diffusion interchangeably in this work. Classical plasma diffusion the-

ory considers electron transport to be driven by collisions with other species in the plasma, and ignores electron-wall interactions and electron-instability interactions, and scales with the inverse square of the magnetic field strength [3, p. 64]. Many observed systems show diffusion instead proportional to just the inverse magnetic field, known as Bohm diffusion [4]. According to Cunningham, Bohm diffusion is usually attributed to instabilities in a plasma, but the instability responsible for it in HETs remains unknown. Cunningham identifies an azimuthal spoke instability, where there is a region of increased electron density moving in the azimuthal domain as a possible contributing factor, and his work observed a strong correlation between these azimuthal spokes and an increased plasma potential. His work has interesting implications if combined with LaFleur, Baalrud, and Chabert's simulations. These simulations showed that with plasma densities similar to those found near the maximum magnetic field locations in HETs (usually the exit plane) and pressures on order of 1 mTorr, a quasi-steady fluctuation in both electric field and electron number density would form over the course of 1-2 μs. The electron density fluctuated between 20% and 30% with electric field fluctuation amplitudes "larger than the applied 'axial' electric field itself" [7]. At higher pressures, however, their simulations agreed almost perfectly with the classical mobility theory.

Tran's work duplicated the anomalous behavior shown LaFleur, Baalrud, and Chabert's simulations, and identified beam-cyclotron instabilities present under normal HET operating conditions, and that this mode transitions to an ion-acoustic wave that saturates over approximately 2μs. Although he found that the beam-cyclotron instability was insufficient to describe the full anomalous transport, the transition to the ion-acoustic wave was found to be extremely important to correctly model the cross-field electron mobility and suggested that the ionzation fluctuations caused by Landau damping may generate the azimuthal spoke modes [2].

2.1.3 Cross-Field Mobility.

This cross-field electron motion has been an active object of study for more than 50 years but remains incompletely understood yet vital to the design of Hall Thrusters, contributing between 20 and 30 percent to the discharge current [2,8,9]. The cross-field mobility, defined as the constant of proportionality between the axial electron velocity and the axial electric field length, is classically given as shown in 14, as shown in [6]. In this equation, v_m stands for the momentum-transfer collision frequency for electrons, and ω_{ce} stands for the electron cyclotron frequency.

$$\mu_{ez} \equiv \frac{u_{ez}}{E_z} = \frac{e}{m_e \nu_m} * \frac{1}{1 + \frac{w_{ce}^2}{\nu_m^2}} \tag{14}$$

Kwon, Tran, and Koo all agree that the anomalous electron motion cannot be fully explained by this classical model, although Koo's work does identify Bohm diffusion as a major factor [2,8,9]. Lafleur, Baalrud, and Chabert agree and further identify azimuthal instabilities in the electron current as a likely source for the anomalous transport above and beyond what the classical model predicts [7]. Their simulations reproduced the observed anomalously high electron mobility via instability-driven transport alone, as walls and emission were not modeled in the simulation. They propose an effective cross-field mobility model to account for the instability-driven transport, as shown in Equation 15

$$\mu_{eff} = \frac{\frac{|q|}{m\nu_m}}{1 + \frac{\omega_{ce}^2}{\nu_m^2}} \left[1 - \frac{\omega_{ce}}{\nu_m} \frac{< n_e E_y >}{n_e E_z} \right] \tag{15}$$

In this equation n_e represents the electron number density and the angle brackets indicate a time-averaged value. Several instabilities of interest exist, some of which have been topics of research since the early 1970s. These instabilities will be discussed in the following subsections. It is worth noting that much of the early research in

plasmas stemmed from fusion research and therefore focused on plasmas that fully magnetized the ions. This stands in distinct contrast to the plasmas involved in HETs, which by design do not magnetize the ions as discussed above. The difference in behavior can be substantial as mentioned in Tran's work, so research into fusion plasma instabilities cannot necessarily be applied to the plasmas common in HETs [2]. Only instabilities applicable HETs will be discussed here. Full derivations of wave frequencies and growth rates will not be presented in this work for brevity.

2.1.4 Ion-Acoustic Waves.

An ion-acoustic wave can occur in plasmas when conditions obey Equation 16 below. In this case, ω and k refer to the frequency and wavenumber of the wave in question. The ion acoustic velocity is then defined as shown in Equation 17, and the dispersion relationship as Equation 18 [3, p. 152].

$$\sqrt{\kappa T_{i0}/m_i} \ll \omega/k \ll \sqrt{\kappa T_{e0}/m_e} \tag{16}$$

$$c_s^2 = \omega_{pi}^2 \lambda_{D,e}^2 = \kappa T_e/m_i \tag{17}$$

$$\omega^2 = \frac{k^2 c_s^2}{1 + k^2 \lambda_D^2} \tag{18}$$

Notably, these waves can only exist when the electrons are much hotter than the ions, as is the case in a HET. Thermodynamically, the regime described by Equation 16 causes the electrons to behave isothermally while the ions behave adiabatically. When this wave exists and there is no electron drift relative to the ions, the wave becomes weakly Landau damped. If the electron drift velocity relative to the ions is nonzero and exceeds the phase velocity, the instability may grow at a rate according

to Tran and presented in Equation 19.

$$\gamma = \left(\frac{\pi m_e}{8 m_i}\right)^{1/2} \frac{|k|\, c_s}{\left(1 + k^2 \lambda^2\right)^2} \tag{19}$$

Importantly, this wave propagates through the ions in the form of ion density perturbations and can thereby impact the electron behavior through their shared influence on the electric field.

2.1.5 Electron-Ion Instability.

Consider a plasma where electrons stream past ions with a much lower velocity, this relative velocity being v_0, as is the case in a HET [3]. The dispersion relationship of such a scenario, shown as Equation 20 can be simplified by collecting terms as follows: let $z = \omega/\omega_{p,e}$, $\epsilon = m_e/m_i$, and $\lambda = \vec{k} \cdot v_{e,0}/\omega_{p,e}$. The resulting equation is presented below as Equation 21.

$$1 - \frac{\omega_{p,i}^2}{\omega^2} - \frac{\omega_{p,e}^2}{\omega - \vec{k} \cdot \vec{v}_{e,0}} = 0 \tag{20}$$

$$1 = \frac{\epsilon}{z^2} + \frac{1}{(z - \lambda)^2} \tag{21}$$

Equation 21 clearly diverges at z=0 and z=λ, and has a minimum between these two values. For sufficiently large values of λ, this minimum is less than one, resulting in four roots of the dispersion equation; as λ falls, however, these roots converge and eventually vanish once λ falls below a critical value. Once this occurs, there will be two complex conjugate roots, one of which will cause an instability since it has a positive sign. The critical value for instability onset occurs when the condition expressed in Equation 22 [3, p. 179]. These situations are illustrated in Figure 1.

$$\lambda = \left(1 + \epsilon^{1/3}\right)^{3/2} \Rightarrow \vec{k} \cdot \vec{v_0} = \omega_{p,e} \left[1 + \left(\frac{m_e}{m_i}\right)^{1/3}\right]^{3/2} \tag{22}$$

The maximum growth rate of this instability occurs when $\vec{k} \cdot \vec{v_0} \simeq \omega_{p,e}$, and has a value $\omega_{i,max} \simeq \frac{\sqrt{3}}{2} \left(\frac{m_e}{2m_i}\right)^{1/3} \omega_{p,e}$ [3, p. 179].

2.1.6 Beam-Cyclotron Waves.

Lampe, et al. presented both the theory and simulations of the beam-cyclotron theory in 1971 [10]. The instability requires that ions drift with some speed relative to electrons (denoted v_d in the following equations)and across a magnetic field and that the electrons be magnetized but the ions are not. They found that the presence of an external magnetic field discretized the ion-acoustic into separate bands centered on the cyclotron harmonic bands (Equations 23-24), with bandwidths shown in Equations 25 and 26.

$$\omega_k = n\Omega_e \tag{23}$$

$$k = \frac{n\Omega_e}{\left[v_d - \frac{c_s}{\left(1+k^2\lambda_D^2\right)^{1/2}}\right]} \tag{24}$$

$$\delta\omega \simeq 2\gamma \tag{25}$$

$$\delta k \simeq 2\delta\omega / v_d \tag{26}$$

In these equations, Ω_e represents the electron gyroperiod as it orbits a magnetic field line, c_s the ion acoustic speed, λ_D refers to the Debye length covered in 2.1.1. The discretization breaks into a continuous spectrum once the ratio of electron-ion

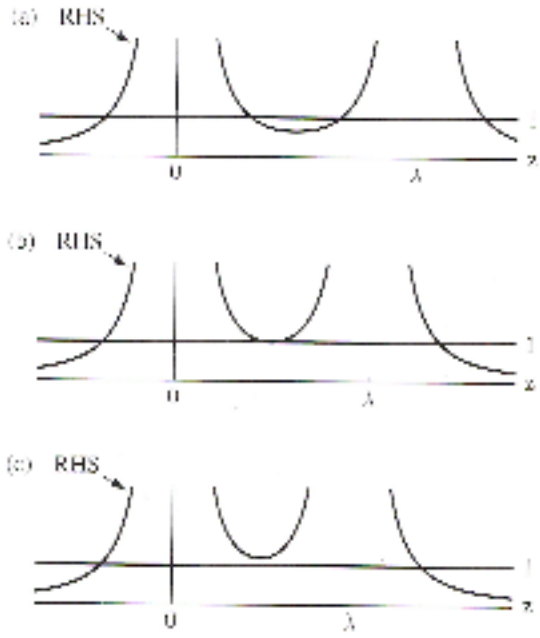

Figure 1. Electron-Ion stability based on λ [3, p. 178]. The first chart shows a stable dispersion relationship with all real roots; the second shows critical stability, with one real root at $\lambda = z$; the final plot indicates an unstable dispersion relationship with two complex conjugate roots.

drift velocity to electron thermal speed grows too high, as can be assumed in a Hall thruster, and the instability degenerates to that discussed in section 2.1.4.

2.2 Computational Theory and Mathematics

2.2.1 Spectral Solvers.

Spectral methods for solving differential equations rely on transforming the input function from one domain, most typically time or space, into a frequency domain with mutually orthogonal basis functions. In doing so, they rely on the core tenet of Fourier theory: that any arbitrary function can be represented as the sum of a (possibly infinite) number of orthogonal basis functions such as sines and cosines. When the boundary conditions are homogeneous or periodic, sines and cosines are indeed appropriate basis functions , though Pozarkidis states that Chebyshev polyno-

mials are more accurate and appropriate for nonperiodic cases [11, p. 511]. A spectral solution provides very high accuracy for smooth functions even with relatively few input points, and may be faster or the only practical method of solving a differential equation with available computing resources [12, ch. 16][11, p. 512].

Koo, Bilyeu, and Martin have demonstrated the use of a pseudospectral 2-D azimuthal-axial model (spectral methods applied in the azimuthal and nonspectral methods in the axial) of a Hall thruster in 2015 intending to investigate azimuthally-driven axial motion [8]. That work largely inspired this one owing to the promise it showed even though the simulation was incomplete as of the time of publication. In particular, spectral methods are ideal for investigating azimuthal motion of any charged species in a HET, since a HET channel is periodic and therefore species number densities must be as well. Azimuthal transport of charged particles, as in Cunningham's work, will cause azimuthal instabilities in the electric field unless the particle distribution is homogeneous, making a spectral approach highly appropriate for updating the electric and magnetic fields as well. This work sought to use a spectral approach to solve Poisson's equation (Equation 4) for electrostatics in the azimuthal direction based on the distribution of charges at a given instant in time [4].

2.2.2 Time-Memory Tradeoff.

When solving any computational problem, the programmer generally faces a choice between pre-computing the results and looking them up when needed, or computing them as needed but not storing the results. The former solution requires the computation of N elements in the data set and only requires the computation once, but it requires a potentially large ongoing memory commitment to store the computed data. Conversely, computing the data only as needed may require less computational effort if the total number of result references is smaller than N, and requires no storage

space whatever. Though such a solution may seem more appealing, computationally expensive problems may require too much time for such an approach to be practical. The chaotic nature of fluids problems that arises from the nonlinear output sensitivity to input conditions makes precomputations very difficult for most parts of the fluid dynamics. Engineers and scientists routinely rely on them for fluid properties by using lookup tables. The U.S. Standard Atmosphere (1976) remains a vital tool for aerospace engineers, and tables of water properties (especially steam) have been so commonly used that references to them in popular culture can be found by at least the 1950s. Tables work for these properties because their behavior is non-chaotic unlike, for example, turbulent flow problems. More complex relationships, or ones that require high fidelity, would be impractical even with today's memory standards.

Consider an arbitrary nonlinear function with five input variables– a simple lookup table using a 64-bit floating point representation that has ten points in each of the five input dimensions would require $10^5 * 64 = 6.4$ megabits (800 kilobytes) of disk space, which is well within modern technology's capabilities but has very few data points per dimension. If the output variable changes only very slowly with the change in an input variable or the domain is very small, the lookup table may be acceptable, but neither is generally true. Using a finer mesh over the domain may make the table more acceptable, but only at the price of a much greater memory cost– using 100 datapoints along each of the input variables would require $100^5 * 64 = 640$ gigabits, or 80 gigabytes of storage space. This is a ludicrous amount of RAM, though still within the realm of possibility for some very large desktop workstations. If the machine lacks sufficient RAM to store the entire lookup table, the program would need to load it from the hard drives at an enormous speed penalty due to the overhead and the specialized hardware underlying the RAM connection to the motherboard. Further increasing the mesh density over the domain would continue to increase the requirements, and this

method all requires either an effective method of interpolation or never attempting to use any value not specified by the table. The interpolation of nonlinear systems at an acceptable accuracy level may not be possible.

2.2.3 Neural Networks.

Neural networks represent an alternative to lookup tables, and represent a sort of middle ground between the two aforementioned options. If the lookup table represents all results of a function over a given domain, the neural network approximates the function itself. As will be discussed, even a very simple network can represent a continuous function to an arbitrarily high degree of accuracy, averting the interpolation restriction imposed on lookup tables [13,14]. A network consists of a number of neurons, organized into a number of layers, as will be discussed in more detail in the following section. Each neuron, broadly speaking, requires two values associated with it (a weight and bias). This fact makes computing the break-even point in memory requirements trivial; while not guaranteed, neural networks have demonstrated remarkable approximation capabilities even with very simple networks.

Using a network imposes some computational cost at run-time since the user must execute the network's inherent math, although this may be less computationally intensive than executing many other 'exact' functions, such as computing the exponential of 1E-15 (in C++ where such behavior is implementation dependent). The greater strength of neural networks stems from removing the requirement that the programmer or user knows the exact functional relationship between the input and output variables. The programmer can instead know or estimate that some relationship between the variables exists based on experimental data, and use the network to determine a useful approximation of that relationship so long as the network is "trained" properly. Training, in this context, refers to the process of adjusting the

neuron parameters until reaching a valid approximation. The number of layers, number of neurons in each layer, and the type of neurons are not adjusted during training and are heavily dependent on the problem the network solves and the input data form.

The universal approximation theorem (UAT), developed by George Cybenko in 1989, states that any continuous, bounded function can be approximated to arbitrarily high accuracy by a feed-forward network (meaning one with no loops in the network itself, so that data flows strictly from input to output) using sigmoidal activation functions such as the hyperbolic tangent [13]. Kurt Hornik extended this in 1991, and proved that the UAT applies to any feedforward network regardless of activation function, making radial basis functions (RBFs) a plausible choice to attempt the approximation required for this work [14]. This incredibly powerful theorem does come with a few implicit caveats, most notably that it makes no statement about the hidden layer size required for such an approximation nor the time needed to train such a network. In effect, while it is possible to build a multilayer perceptron (MLP) with one hidden layer to approximate any function, doing so may be completely impractical. Importantly, the UAT only applies on a bounded function, meaning in effect that a network trained on the domain from A to B can achieve arbitrary accuracy over that range, but may be uselessly inaccurate beyond it. The limitation to continuous functions did not affect this work.

2.2.3.1 Structure of a Neural Network.

Inspired by biological neurons and some of the simpler information-processing networks in the brains of living things, artificial neural networks (ANNs or neural networks hereafter) have found great use across science and engineering fields as an effective mechanism for solving very complex problems, including many that would

be difficult or entirely impractical via more conventional programming methods [15, p. 11]. Broadly speaking, ANNs consist of an input layer, a number of hidden layers, and an output layer. Though strict definitions do not exist, common parlance in the field refers to a network with a single hidden layer as "shallow" and one with several or many hidden layers as "deep". Each connection from one neuron to another has a weight, and may or may not have a bias; each neuron sums the weighted inputs, adds the bias, and executes an activation function of some kind. The number of neurons in each hidden layer, the number of hidden layers, and the connections between layers together make up the topology of the network. Many such topologies exist, and a good survey may be found on the website of the Asimov Institute, including a wide variety of network types along with descriptions of what and how they function [16]. The same networks are covered with slightly differently focused explanations on the "Towards Data Science" website [17]. The details of a problem solved with an ANN fixes the number of input and output neurons; each input variable requires its own neuron, as does each output value. Regression and function-modelling problems such as those covered in this thesis require a single output neuron, while a classification problem may require dozens or hundreds depending on the number of separate classes the network must distinguish. In their simplest form, classification problems can be considered just a regression problem with multiple possible outputs with independent output variables, one per option, although this is not true of all network topologies and is suggested merely as a useful way to conceptualize the difference for an unfamiliar reader.

Only a few of the topologies illustrated in either of the aforementioned surveys received significant consideration for the purposes of this work. The networks entitled "(Deep) Feed Forward", more commonly known as a dense network or MLP, presents the perhaps the most broadly used network topology for approximation tasks such as

those involved in this research [15]. Despite the name, most graphics will show only a single hidden layer since every neuron in layer A connects to every neuron in layer B, so the rules of matrix multiplication and addition permit representing an arbitrary number of hidden layers as effectively a single operation. The radial basis function (RBF hereafter) is structurally identical but specifies the use of a radial basis function as its activation function, the implications of which will be discussed later. Convolutional neural networks, though popular as of 2018, deliberately attempt to model the behavior of the human eye, so while excellent for picking features from images, audio data, and other similar dataforms, were rejected both for the topology's optimization for other tasks and for being needlessly complex approaches to the problem at hand [16,17]. The various memory-related networks shown in those surveys (RNN, LSTM, and GRU) were rejected on the grounds that they are designed to work on time-series data, e.g. predicting text and voice patterns, rather than working on the non time-dependent data examined in this work.

The conditions of the Universal Approximation Theorem (UAT), discussed in the previous section, caused rejection of the remaining types of neural networks shown in the surveys, leading to only the MLP and RBF networks considered. The MLP required substantially less effort to implement using the TensorFlow library and according to the UAT could prove effective, so the MLP was implemented first with the intention to attempt an RBF implementation should the MLP approach have failed.

2.2.3.2 Activation Functions.

Each neuron may have an activation function governing transforming the sum of weighted inputs and bias into a more useful output. Despite Cybenko's initial work, Hornik proved that the activation function is irrelevant to the UAT's applicability [13, 14]. Sigmoidal activation functions typically include the hyperbolic tangent function

because it shares the general shape as the logistic function, often referred to as "the" sigmoid function and defined as $f(x) = \frac{1}{1+\exp(-x)}$. Importantly, the logistic function has a range between zero and one, while the hyperbolic tangent function has a range between negative and positive one. Walia and Karpathy both discuss the use of the rectified linear unit (ReLU) and the leaky ReLU, which are linear for all values zero or greater, but have zero or very small slopes for negative inputs, respectively [18,19]. Samarasinghe also discusses Gaussian and Gaussian complement functions, though neither was considered for use in this work since the other functions mentioned have a well-documented history of success in neural networks used as approximators [15]. Activation functions have substantial impact on both the network's behavior and its Walia and Karpathy both discuss the benefits and drawbacks to the logistic, hyperbolic tangent, and ReLU functions at some length, and their arguments are summarized as follows:

Logistic functions are the prototypical activation function for use with the UAT, but suffer from saturation problems that can lead to an approximately-zero gradient both at large or small activation values, making training the network very difficult since the error signal may approach zero even if the error is very large [cite]. This problem is known as the vanishing gradient problem. Logistic functions also have an output range centered on one-half, leading to strange behavior during the training process since the weight gradient must not change sign. The hyperbolic tangent function mitigates the latter problem since its output range centers on zero, but can still saturate; as Karpathy remarks, the hyperbolic tangent function is always preferred to a logistic function because it otherwise behaves as a scaled logistic function, retaining the advantages but with one fewer disadvantage. ReLU functions are computationally inexpensive and cannot saturate, making them much faster to train and easier to work with, but it is possible for them to become unreachable nodes on the network during

training. Karpathy states that an erroneous learning rate may cause up to 40% of the network to "die" in this manner, a hefty memory penalty [19]. Walia points out that the limitation of a linear function for positive input means it should only see use in the hidden layers, never in the output. The leaky ReLU's small negative slope avoids the issue of "dead" neurons, though still should not see use in output layers. Both sources recommend using the ReLU and switching to a leaky ReLU if "dead" neurons become problematic, though acknowledge that a hyperbolic tangent function is expected to work albeit with worse performance than a ReLU would give [18,19].

2.2.3.3 Preprocessing.

Another neural network limitation stems from the need to ensure input data have similar ranges. As described in Samarasinghe, having wildly disparate variable ranges can lead to masking problems in that variables with larger magnitudes may receive much stronger weights and smaller variables may get ignored– even if the smaller-magnitude variable matters more [15, p. 253]. Samarasinghe illustrates a number of mechanisms to normalize the input data, as do the course materials from Stanford University's CS231N class dating from Spring 2017 and the Stonybrook University course materials for CSE634, "Data Mining", dating from Spring 2018 [19,20]. All three sources identify normalization (called max-min normalization in the Stonybrook materials) as a mechanism for changing the range of input variables between 0 and 1 or -1 and 1, trivially depending on the implementation. Another mechanism, referred to as whitening-normalization (Samarasinghe, pg 254) or principal component analysis and whitening (Karpathy and Johnson), uses the variances and covariances to scale each dimension by that dimension's standard deviation and using new variables from the covariance matrix. These new variables "correspond to a set of new rescaled variables that have unit variance and are independent of one another" [15, p. 254],

making this approach very appropriate for data with many input variables of unknown impact on the output (particularly if some variables may be irrelevant) and interrelationships unknown a priori. Karpathy also suggests subtracting the mean of each variable, effectively centering the data about the origin, an appropriate approach if no scaling is necessary or if the variables should not necessarily have approximately equal impact on the output. Finally, Samarasinghe discusses standardizing each input variable by subtracting the mean and dividing by the standard deviation of that variable across the entire dataset; using this mechanism also requires applying it to output data rather than only the input [15, p. 253].

2.2.3.4 Initialization.

Training the network requires weights and biases in order to calculate the first-iteration results, and so they must be initialized to some values first. As Karpathy and Johnson discuss, setting the intial weights to zero can cause a symmetrical error signal, resulting in all neurons always outputting the same values or can cause zero error signal due to neuron saturation [19]. The identical error signal to all neurons would be an identical problem regardless of the initial value set, so long as all values are equal. The simplest method to avoid this uses random numbers to initialize the weights; these numbers are typically small, again to avoid causing a near-zero error signal. Karpathy and Hao also discusses using random numbers and dividing by the square root of the number of neurons to reduce the overall variance of neuron outputs [19,21]. Hao also discusses a method for optimally assigning weights for large networks using many layers called Xavier initialization, the proof of which will not be repeated here. Xavier initialization attempts to optimally set each weight, but this work did not intend to optimize the network design and because only a single layer was necessary or investigated here owing to the UAT, the method did not receive

consideration in this work. Unlike the weights, setting the initial bias values to zero poses no issue for the network, since non-identical weights break the initial symmetry [19].

2.2.3.5 Optimizers.

Choosing the correct optimization algorithm (referred to as an optimizer hereafter) can make the difference between training a network in a reasonable timeframe and permanent oscillation between poor options. As mathematical optimization has been actively studied since at least the time of Newton (i.e.: Newton's Method), too many optimization algorithms exist to reasonably discuss in this work, so only those most relevant to the task at hand will be covered. The venerable gradient descent algorithm (also known as steepest descent) is widely known to any calculus student, and uses the gradient at each training step to compute the best error surface direction to move in via updating the network. This solution suffers from using a fixed step size; setting this size too small may result in an impractically long training time, while setting the step size too large will cause oscillatory behavior as the network cannot reach its best solution [12]. Google developed several optimizers for inclusion in their TensorFlow machine learning package, including the ADADELTA optimizer and its related ADAGRAD, ADAGRAD Dual Averaging (AdagradDA), and Proximal ADAGRAD optimizers. Zeiler's paper describes the ADADELTA algorithm as a per-dimension optimizer able to adapt the learning rate dynamically over time and "has minimal computational overhead beyond vanilla stochastic gradient descent." The algorithm's robustness to input parameters makes it very appealing, as does the low computational expense [22]. Duchi, Hazan, and Singer developed the ADAGRAD algorithm to emphasize the importance of "infrequently-ocurring features" versus common features in the input data [23]. This emphasis makes it an appro-

priate choice for convolutional networks solving image processing or computer vision problems, but does not obviously make it a good choice for developing an approximation network. The AdagradDA algorithm is similar, but intended for use "when there is a need for large sparsity in the trained model" [24]. Hinton, Srivastava, and Swersky from the university of Toronto discuss the RMSProp optimizer in their series of lectures, though it is better summarized by Kingma and Ba in their paper introducing the Adam optimizer as effective in online settings [25,26]. Kingma and Ba's Adam optimizer combines the advantages of AdaGrad with and RMSProp in an optimizer that works even in noisy or sparse gradients as may occur with noisy input data or with a small input data set, along with a certain amount of reducing user-error; since the hyperparameters "have intuitive interpretations and typically require little tuning". This fact substantially reduces the opportunity for overenthusiastic graduate students to cost themselves lots of time and migraines obsessing over the perfect hyperparameter settings, making the Adam optimizer an appealing choice.

2.2.3.6 Potential Issues for an ANN.

One potential issue that can arise when using a neural network stems from the network's ability to model the data given rather than the underlying relationship. When this happens the network has essentially just memorized the training data but cannot generalize from the examples to arbitrary inputs. Ultimately, overfitting stems from the network having too many hidden neurons relative to the training data, and Samarsinghe and Karpathy both highlight several methods to reduce overfitting while still preserving the network's ability to model the relationship without bias [15,19]. Based on this description, an obvious solution is to ensure a sufficiently large training data set relative to the number of hidden neurons; unfortunately this is not always possible in real applications. A somewhat naive solution would be attempting to

optimize networks exhaustively until finding the ideal number of neurons, but doing so necessitates an enormous time and computational resource commitment and is a generally impractical solution, especially with larger or deeper networks [15, p. 197].

Randomly deactivating some of the neurons during training in a process known as dropout has proven an effective method to prevent overfitting as discovered by Srivastava et al [27]. When using dropout, the probability of retaining a neuron as active during training is a hyperparameter, and all neurons are retained for testing. Dropout seems to have superseded the optimal brain damage, optimal cell damage, and optimal brain surgeon techniques for pruning a network as Samarasinghe discusses. Both Samarasinghe and Karpathy recommend regularizing the network, or penalizing larger weights by including the weights in the loss function calculation. Karpathy goes into more detail, discussing both L1 and L2 forms of regularization, and notes that when unconcerned with feature selection L2 norms will likely give superior performance to the L1 while using the L1 norm tends to generate neurons that tend to become resistant to noisy input vectors. Mathematically, the L2 norm is described by $\frac{1}{2}\lambda \sum_{i=1}^{n} w_i^2$, while the L1 norm is the simpler $\lambda \sum_{i=1}^{n} |w_i|$

3. Experiment

3.1 Plasma simulation

3.1.1 Intended Simulation.

AFRL's TURF simulation code is still in development, and this work originally sought to further that development by adding a pseudospectral solver for Poisson's equation using the model demonstrated by Koo, Bilyeu, and Martin in 2015 [8]. Time constraints ultimately made completing this task impossible, but a brief description of the intended simulation will be provided here. As the goal was to develop an additional solver for TURF, the simulation described in Tran's thesis would have been duplicated [2]. This simulation is ultimately a highly simplified version of that described in Lafleur, Baalrud, and Chabert's second 2016 paper, and would have used the radial-azimuthal plane to examine the plasma behavior. The coordinate correlation is illustrated well in figure 2 below.

In all HETs, the magnetic field is applied radially (in the computational $\hat{x}$ direction), while the electrostatic accelerating field is applied axially (computational $\hat{z}$ direction). The obvious result of this coordinate approach is that the $\vec{E} \times \vec{B}$ drift takes place in the azimuthal direction (computationally $\hat{y}$) [2]. The fluctuations in electric field would generate an azimuthal electric field component from solving Poisson's equation, and this azimuthal component, could result in spoke-mode behavior. A spectral solver was considered an ideal approach to handling the azimuthal Poisson solution since both the electric field and its first derivative must obviously be azimuthally periodic. The hope was that the spectral solver would operate faster than a direct integration method without sacrificing accuracy. While such a solver algorithm was developed, the problems described in 3.1.2 prevented its implementation in the TURF framework.

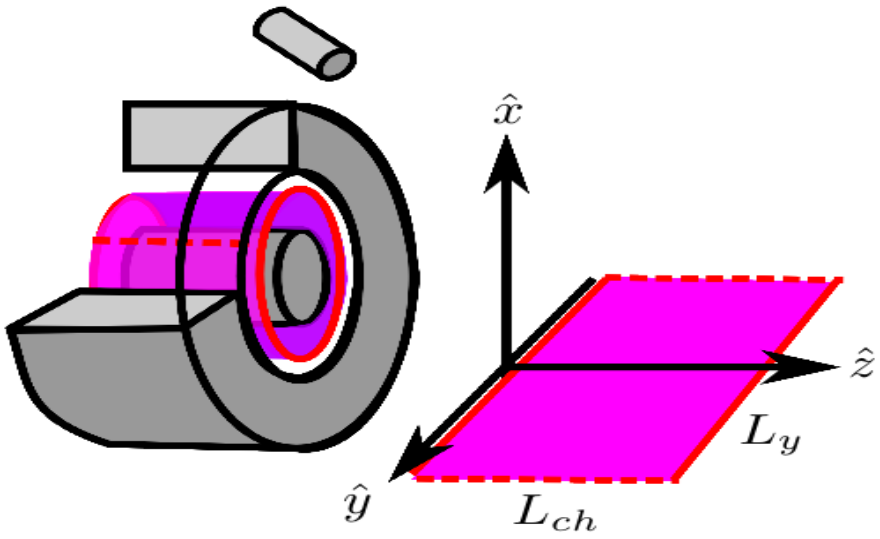

Figure 2. Illustration of a Hall Thruster with the computational domain shown in pink as shown in Tran's thesis work [2, p. 16]

Fourier transforms are a common mathematical tool in engineering and scientific fields, so rather than writing the necessary functions from scratch and risk missing problematic edge cases or poor performance this work sought to implement a preexisting and preferably already optimized library. The "Fastest Fourier Transform in the West" (FFTW) library met these requirements. It was developed by Matteo Frigo and Steven G. Johnson at MIT and initially released in 1997 with the latest update released on Oct 29, 2017 [28]. The library is so well-optimized (claiming N*logN operational time for arbitrarily-sized transformations) that even MathWorks licenses it for use in their MATLAB programming language, renowned for high computational speed even with very large problems [29]. The FFTW library's availability in both C and FORTRAN meant it would not bottleneck the simulator from a programming-language-speed perspective, since C is typically marginally faster than C++ and FORTRAN is famously even faster for purely mathematical (i.e. non-logical) operations. While no theoretical guarantee exists that the simulation software would not

bottleneck on the solver, attempting to further optimize these functions would have been vastly beyond the scope of this work and more appropriately part of a computer science thesis than an one on aerospace engineering.

3.1.2 Problems Encountered.

Unfortunately, the FFTW library implementation in TURF proved extremely problematic and eventually revealed compiler-dependent segmentation fault behavior preventing its further use. Validation code using FFTW compiled correctly when using G++ (the GNU C++ Compiler) and the compiler switches described in the FFTW documentation, but failed when using NVCC, the NVidia CUDA Compiler. Compilation with NVCC would permit running the simulator on a compatible graphics processing unit (GPU, or colloquially a graphics card), enormously accelerating the simulation process by running computations in parallel on hardware optimized for floating-point optimizations rather than the logical-decision-optimized CPU. At the time of this writing significant parts of the TURF code base lack much of the code necessary to take full advantage of a GPU, in particular the code required to move data to and from local GPU memory rather than the computer's RAM or even the hard drive. AFRL personnel have confirmed an intent to add these capabilities at a future date. NVidia GPUs require using the NVCC, so while algorithmic code restricted to running on the CPU may be acceptable today since the rest of TURF cannot take advantage of the GPU, code that does not compile correctly using NVCC creates much larger problems for maintenance coders and future releases. The segmentation fault occurred when using the FFTW functions to perform an inverse FFT, and may stem from NVCC's inconsistent behavior with importing and using C-language libraries.

Once the problem was traced to a conflict between the compiler and library, FFTW was abandoned since NVCC is NVidia-proprietary so the source code is unavailable

for modification and modifying the library would probably not have fixed the problem and risked compromising performance. NVidia's CUDA extension to C++ natively includes a version of the FFTW library as cuFFTW, but using it requires using other CUDA-specific functionality and appears to require execution on the GPU [30]. If the functionality requires GPU execution, the clock-time penalty incurred from transferring the data between RAM and local GPU memory could bottleneck the entire program; transferring the particle data but only using the GPU for the few operations involved in solving Poisson's equation wastes the advantages provided by implementing the parallel behavior. These issues and time concerns caused cuFFTW's rejection for use in this work, though future releases of TURF should consider its use.

Searching for other libraries that provide C++ Fast Fourier Transform functions led to a website with examples of these functions provided in a variety of languages including C++ [31]. The code was validated with appropriate modifications to reflect different input and output data types and the validated code was implemented in the TURF spectral solver. It uses the well-known radix-2 Cooley-Tukey algorithm first published in 1965 (though originally invented by Gauss in 1805) [32][12, p. 128]. The specific variant of the algorithm implemented in this code was chosen for its simplicity (lending itself well to understanding and debugging), and because it runs in N*logN steps rather than the N^2 required by a naive Discrete Fourier Transform operation [11,12]. On attempting to implement this in the TURF library, undocumented problems occurred when interfacing with TURF's data structure responsible for storing the various fields. These problems could not be resolved, so the solver was fully characterized up to 1024 bins using data structurally similar to what would have been used if integration had been successful.

3.1.3 Solver Characterizatiton.

Despite the problems interfacing with TURF, the solver code was fully developed and tested. As such, its performance was validated via several mechanisms. Firstly, both a FFT and discrete fourier transform (DFT) were calculated using a user-specified arbitrary power-of-two transformation length. A single value in the input vector was set to a nonzero value, and the resulting transformed vectors were compared. Secondly, the program takes a separate user-specified arbitrary power-of-two transform length to generate a waveform with a user-specified number of component cosine waves with random amplitudes and frequencies. The method of manufactured solutions was used in that the waveform's second derivative is calculated and integrated twice using the spectral solver, then transformed back into the original domain and the results compared with the exact output computed from the original waveform. The frequencies used are constrained to integer values because of the periodic boundary conditions, and further limited to values below the Nyquist limit.

If the individual second-derivative waveforms in the above test are considered the modes of the electron distribution function about a Hall thruster, then the random-waveform test checks the intended use of the solver using simulated data. To fully characterize the solver's performance, the test was repeated for each power-of-two transformation length between four and 1024, inclusively, and each transformation length tested using up to 64 individual waveforms. No case tested more waveforms than the transformation length for obvious reasons. The solver-calculated solutions to Poisson's equation were then compared to the known solution, and the results presented in Chapter 4.

3.2 Neural Network Model

Initially this work sought to approximate the cross-field electron mobility based on several parameters as determined by Dr. Koo and the AFRL team, but time constraints related to building a functional spectral solver in SMURF prevented generating the appropriate data. AFRL provided experimental data from testing a Hall-Effect Thruster (included as Appendix A) to demonstrate neural network approximators' general applicability to HET-related problems and to performance predictions in particular. Using such a model requires sufficient training data, which may be thruster-design specific depending on the data features in question. As discussed in the time-memory tradeoff and neural network subsections of Chapter 2, a properly trained neural network will not suffer from the problems inherent in lookup table interpolation, and required working data storage becomes one of storing only the network rather than the entire performance envelope. Additionally, using a neural network may reduce the overall number of tests required to effectively describe the envelope. This possibility is dependent on the level of accuracy and discretization required, but a lookup table with the desired discretization may be significantly larger than the amount of training data required for a neural network to achieve sufficient accuracy over the same domain. The difficulty, cost, and time required for extensive HET testing makes using such a model an appealing prospect from engineering, cost, and computational perspectives.

The neural network model used in this work was created by leveraging the Python-language implementation of the TensorFlow library developed by Google, and the code (and documentation for it) is presented in A. TensorFlow's production by Google and high popularity correspond to good documentation and community support while its broad industrial acceptance and provision of a low-level API make it appealing from a code maintenance perspective. These reasons together caused its acceptance

for this work [33]. In hindsight, another library known as Keras that provides a higher-level interface (though still requiring a separate backend such as TensorFlow) would work at least as well and may be both more intuitive and easier to work with. Were this work repeated today, Keras would see use, and it should be considered in future work. Google has released TensorFlow for several languages, but warns that others lag Python in the development process and explicitly warns that the documentation may not be accurate for other languages [33,34]. Non-Python versions of TensorFlow were rejected because the network developed herein was never intended for operational deployment within the SMURF package, making the higher speed of other languages unnecessary. Additionally, the time constraints and issues caused by insufficient documentation while implementing the spectral solver presented substantial concern. Importantly, the goal of this work was not to optimize the network but to demonstrate ANN applicability, meaning there almost certainly exists smaller or more efficient MLPs capable of comparable accuracy to the network developed herein.

The network code used in this work consists of two parts: a network class (FCNetwork) and a script including the training loop and output. The FCNetwork class creates an MLP object given the number of input arguments and a list each of the hidden layer sizes and the activation functions. Error-checking means that adding new or custom activation functions will require modifying the FCNetwork code. Metaprogramming and advanced Python functionality could circumvent that limitation, but would not have added anything for the purposes of this work and would have made the code more difficult to understand for those unfamiliar with the mechanisms in question. Relevant columns (in this case hardcoded as the columns corresponding to electromagnet current, discharge voltage, and discharge current) are scaled and retained in the data structure passed to the script, while irrelevant columns are discarded to avoid wasting working memory. The provided data consists solely of text,

and requires far less space than a typical machine has RAM; larger amounts of input data, especially images or audio, would require using a data queue to avoid causing a segmentation fault by attempting to load too much data into working memory simultaneously, but a queued input pipeline was unnecessary here and so avoided. The input data were scaled using the normalization method discussed in chapter 2, reducing input variables to a range between 0 and 1. Doing so means attempting to give the network a value greater than the maximum presented in the training data could cause saturation and inaccuracy, as suggested by the UAT's restriction to bounded functions. As Chapter 2 covered, exceeding the bounds of the network invalidates the UAT's guarantee of arbitrary accuracy. Knowing that the input variables are fully independent a priori makes the whitening and primary component analysis approach unnecessary, and knowing that the inputs were not sampled statistically but resulted from a methodical sweep through part of the thruster's performance envelope makes standardizing by the mean and standard deviation an equally poor choice. These factors are not generally true, so the choice of data normalization depends heavily on the expected input data the network will use both for training and operationally. Modifying the provided code for use on other data will require careful consideration of preprocessing techniques.

The script file includes the weight and bias initializations along with the optimizer chosen. As noted in Karpathy's course notes from Stanford, the intuitive approach of all-zero (or any other uniform value) initial weights is highly problematic since there's no asymmetry and so all neurons will receive the same feedback signal [19]. For simplicity of implementation and understanding, the weights and biases were initialized to small random values, though as covered in chapter 2 other methods reliably result in faster network training. Optimizers were also covered in chapter 2, emphasizing that the optimizer algorithm has great impact on the required training

time. The ADADELTA optimizer was chosen both to avoid problems stemming from a continuous learning rate and for its demonstrated high performance.

3.2.1 Network Toplogy.

Based on the Universal Approximation Theorem discussed in Chapter 2 and the relative performance of MLP and RBF networks on potentially irregular data, an MLP was chosen to model the Hall thruster behavior. The UAT guarantees that this network can approximate the surface shown in Figure 20 to an arbitrary accuracy within the domain trained using only a single layer. The data's analog nature guarantees that it must be continuous. Initially, an attempt was made to use the hyperbolic tangent function, but scaling the data such that the maximum value of each input was treated as 1 caused the vanishing gradient problem and poor results that converged unacceptably slowly. This problem was handled by switching to using a leaky ReLU as the activation function. Lacking any good method of determining the necessary number of neurons a priori, networks were tested with 20, 40, 80, 160, 320, 640, and 1280 neurons in experimental mode.

3.2.2 IVB Mapping and Training Approach.

The data AFRL provided included human-supplied measurements of electromagnet voltage, the magnetic field strength, and the discharge current. The neural network used the former two as inputs, and penalized differences from the known discharge current. Each of the three were scaled by the maximum value present in the input data, such that the data presented to the network only saw values between 0 and 1, as mentioned above. The total cost function at any step included the sum of the L2 norm of this inaccuracy and one ten-thousandth the L2 norm of the weights involved in the network. Including the weights in the cost function pushed any weights

that were not strengthened during any given training step towards zero in an attempt to combat overfitting. During each training step, a fairly typical 80%/20% training and validation split was used, though the user can easily modify this in the master approximator function. Training took place over 5000 epochs for both code validation and the experimental data.

3.2.3 Code Validation.

When in code-validation mode, the approximator generates a nonlinear 2D polynomial function with randomly-chosen, arbitrary coefficients between 0 and 50. This range was chosen only to demonstrate the effect of possibly-large coefficients that better reflect real, general data than the 0-1 coefficients that the NumPy random number generator function would otherwise return. All data validation occurred using 80 neurons, the center value of the initially-intended neuron list. Since noise levels on the AFRL-provided data are unknown, they were assumed Gaussian, and as such the code was validated against a function with zero noise and with SNRs of 1, 2, 3, 4, and 5. The AFRL-provided dataset includes fewer than 300 datapoints, so the network was validated with 240 datapoints, 1200 datapoints, and 2400 datapoints in an effort to characterize the network's ability to handle noise relative to the amount of data provided.

4. Results and Analysis

4.1 Solver Characterization

4.1.1 Performance with Increasingly Complex Waves and Transformation Lengths.

The solver's performance was demonstrated using Poisson's equation on synthetic data similar to what it would have been solving with TURF. The waveforms can be considered without loss of generality to be the electron density about a Hall Thruster's azimuthal axis, in which case the solver generates the electrical potential. The algorithm can only accept transformation lengths (number of samples or measurements) that are powers of two, so the powers of two between 4 and 1024 inclusively were tested. No technical reason prohibits longer transformations, rather 1024 was used as the upper limit since many libraries including FFTW default to such a length. The waveforms used were made up of the lower of 64 modes or the transformation length in order to keep the problem tractable. The results are shown in blue in Figure 3, below, while the machine-zero level is shown in red.

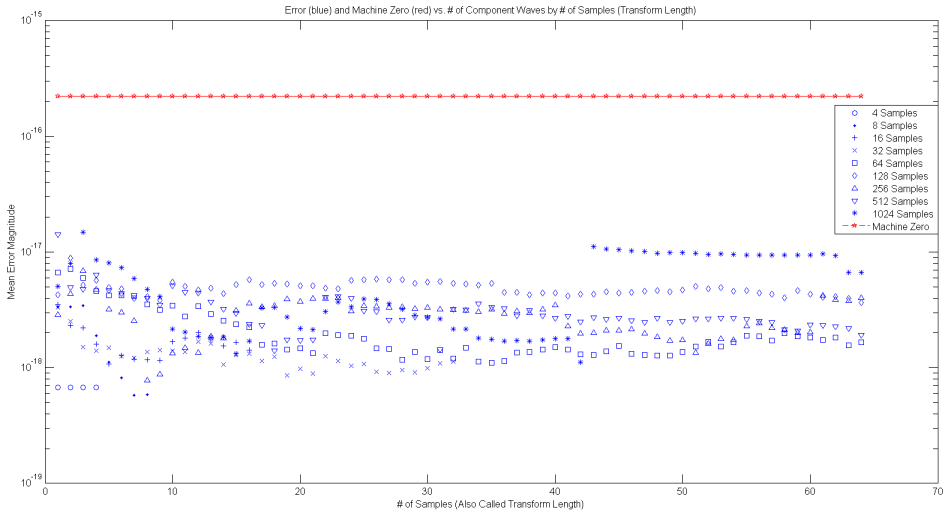

Figure 3. Error (Blue) and Machine-Zero (Red) vs. Number of Component Waves and Transformation Length

These results clearly demonstrate that with zero noise, as would be expected from a perfect simulator, even very short transformation lengths have mean errors more than an order of magnitude lower than the machine-zero level. In practical terms, these results indicate that there will be no compounding error from using this solver as the computer will not propagate errors smaller than the machine-zero level. This fact is hardware dependent, so the test should be run again before implementing this solver on any new hardware.

4.1.2 Noise Performance.

Any real simulator will have some level of noise present in the signal, and therefore the solver's performance was characterized with composite signals comprised of a waveform normalized to an amplitude of one and noise levels between zero and one in increments of 0.2. The waveform was itself made up of eight base waveforms as discussed in 3.1.3. The results of this test are presented in Figures 4 and 5 below.

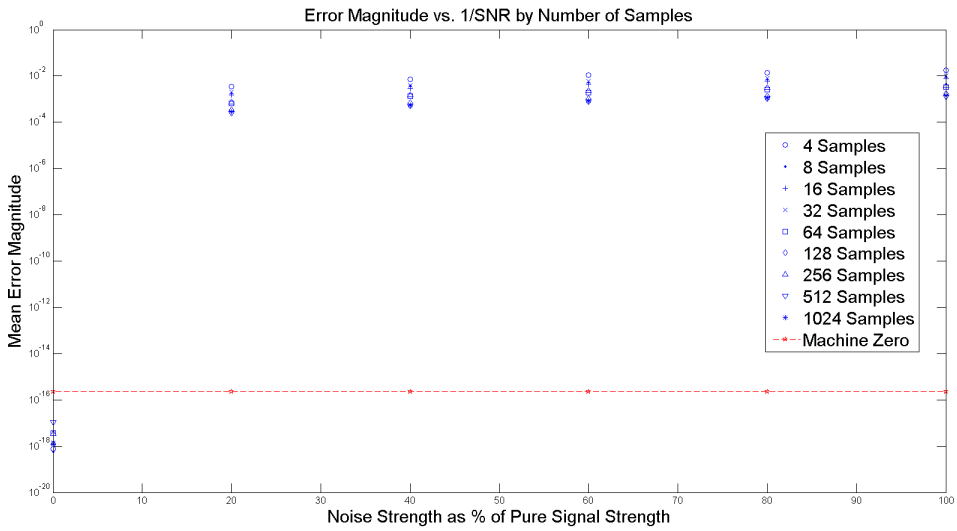

Figure 4. Introduction of any noise worsens performance above the noise floor of this solver, but the performance appears relatively constant with additional noise

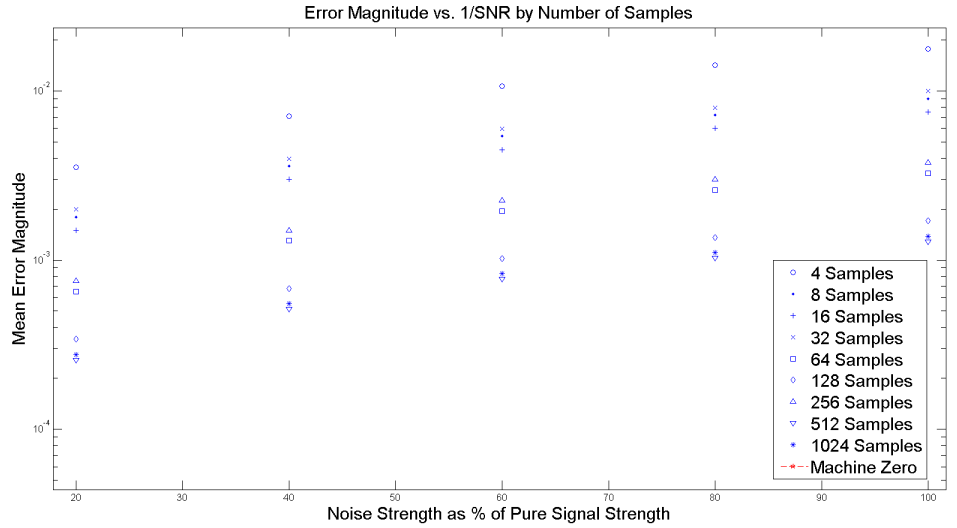

Figure 5. Close-up of the noisy data errors. Note that the error level increases approximately linearly with the noise

Figure 4 shows that in noiseless data, there is no practical error in the solver output. Errors below the machine-zero level will not be carried forward as floating point error by definition of the machine-zero level. Figure 5 demonstrates that as

the noise level increases, the error increases linearly above 20% noise strength. Based on the available data, there must be a nonlinear 'knee' in the error level somewhere between the 0 and 20% relative noise, but this cannot be determined with certainty from the available data. Interestingly, these results also show that increasing the number of samples does not necessarily lead to lower error levels, as shown by the 32-sample error being higher than the eight-sample or 16-sample error for all noise levels tested. This behavior was consistent across multiple test runs, eliminating the random function as the source of the behavior. It is possible that the behavior is machine-dependent, and further tests could confirm this. This solver's completion makes it possible for future work to integrate the algorithm into TURF directly. Once this has been completed, it can be used to further investigate the azimuthal electron waves and instabilities and in so doing provide some insight into the anomalous electron mobility.

4.2 ANN Results

4.2.1 Varying Neuron Numbers.

Table 1 below shows the mean error and standard deviation of the results for each network width, as computed by the final-epoch network. The 20 and 40-neuron networks performed very poorly, with nearly identical error percentages. Interestingly, the 40-neuron network has a much higher standard deviation than the 20-neuron network. The 80-neuron network had the lowest mean error (at only 0.04%) than any other network width, and had the second-lowest standard deviation of any network, and only the 1280-neuron network had lower standard deviation.

These results clearly illustrate that while the UAT guarantees that arbitrary accuracy is possible with sufficient neurons, simply adding neurons to a given network and training it does not guarantee that the new network will have greater accuracy

# of Neurons	Mean Percent Error Magnitude	Std Dev
20	5.30%	21.89%
40	5.31%	28.79%
80	0.04%	9.11%
160	0.93%	11.13%
320	0.64%	15.76%
640	0.99%	19.43%
1280	0.22%	6.62%

Table 1. Experimental ANN Statistics

or precision if trained identically to the original network. The error and standard deviation present here show that even 1280 neurons is not 'sufficient' for perfect accuracy, although either the 80 or 1280 neuron network could be useful for some industrial applications. They strongly suggest that the network can be used to learn the functions describing complex behavior, such as the cross-field electron mobility, based on experimental data. In other words, these neural networks make it possible to model behaviors that are observed but not yet sufficiently understood to describe mathematically

The following figures show the improvement in both total cost (including scaled weight cost) and squared miss cost (defined as $\sum (expected - computed)^2$) as a function of the epoch during network training for each network. The 20-neuron model shows minimal improvement, likely indicating the network was insufficiently complex to reasonably capture the function. The 40-neuron model shows much greater improvement, but the total and miss cost functions have nearly identical shapes, illustrating that the miss cost dominates the total cost, and again that the network probably does not have sufficient complexity to capture the overall function. The 80-neuron network, however, shows almost an order of magnitude improvement in the squared miss cost and two orders of magnitude in the total cost, both of which are substantially better than the 160, 320, and 640-neuron models by the end of training. The 1280-neuron model had a higher total cost than the 80-neuron model, but with

approximately half the miss cost. Moreover, while the miss cost continues to fall after the 1200th epoch, the total cost (the cost that the network trained on) has converged at that point.

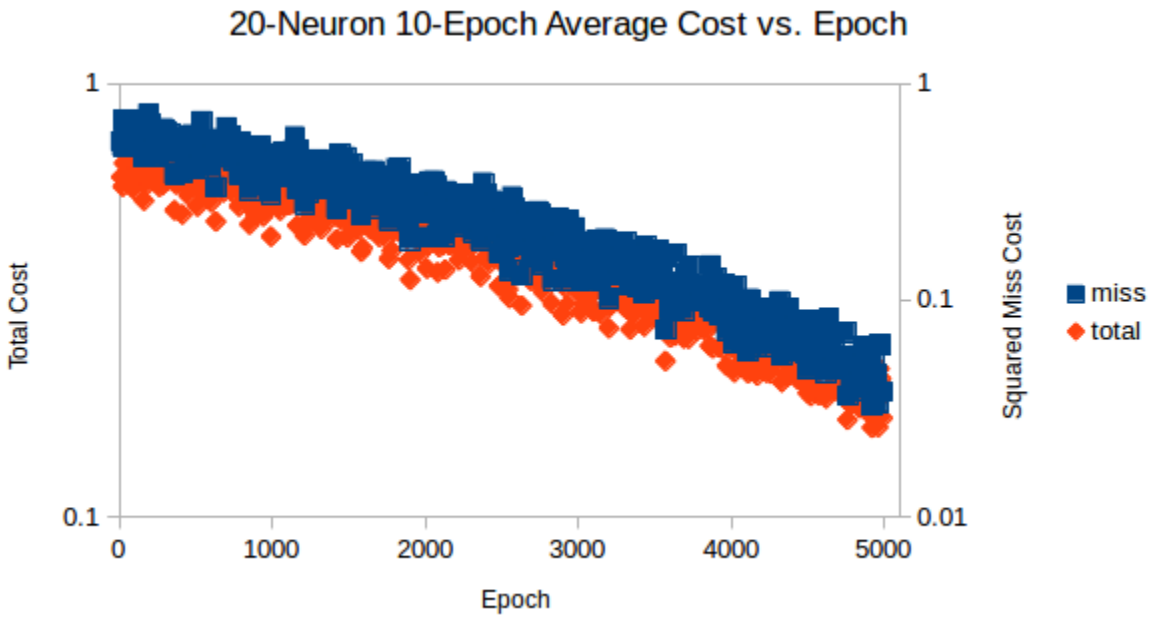

Figure 6. 20-Neuron Results

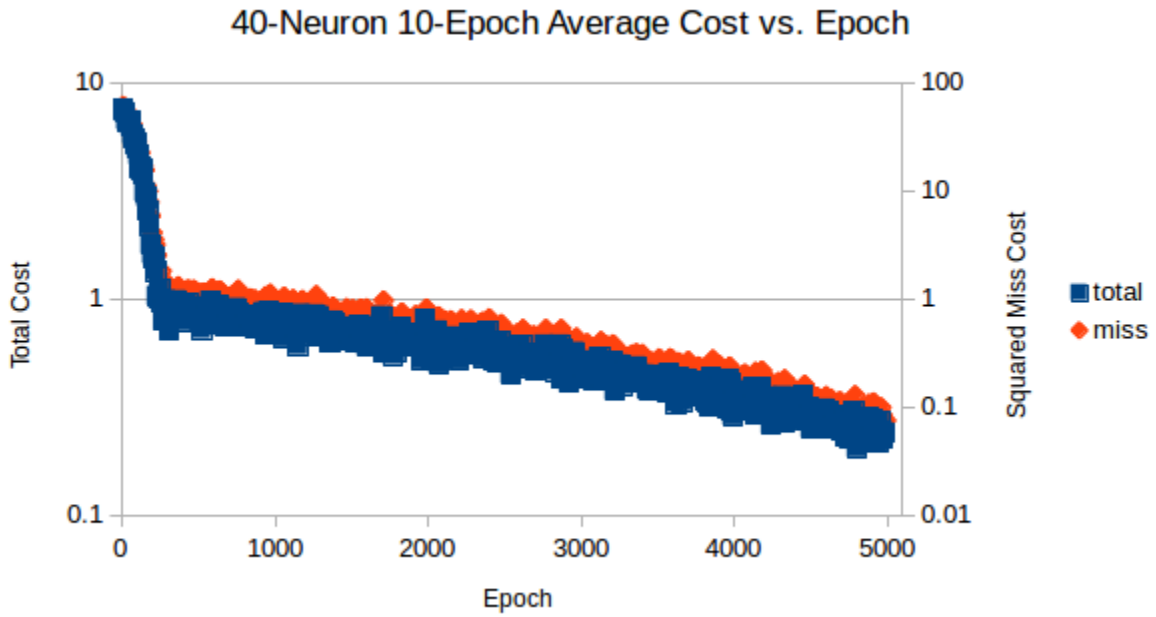

Figure 7. 40-Neuron Results

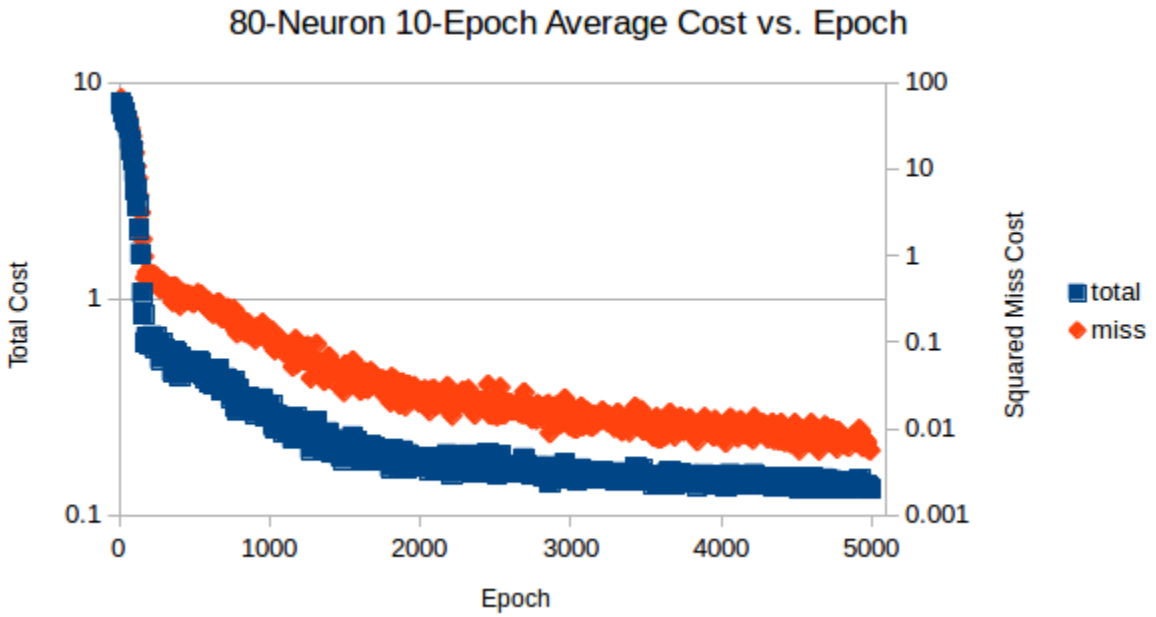

Figure 8. 80-Neuron Results

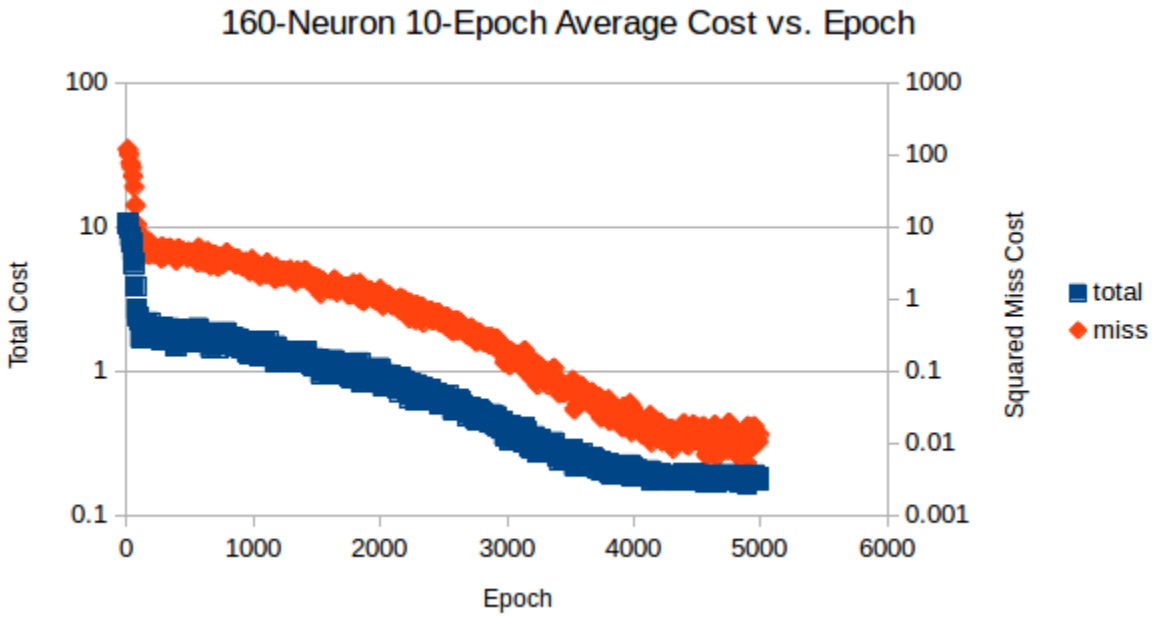

Figure 9. 160-Neuron Results

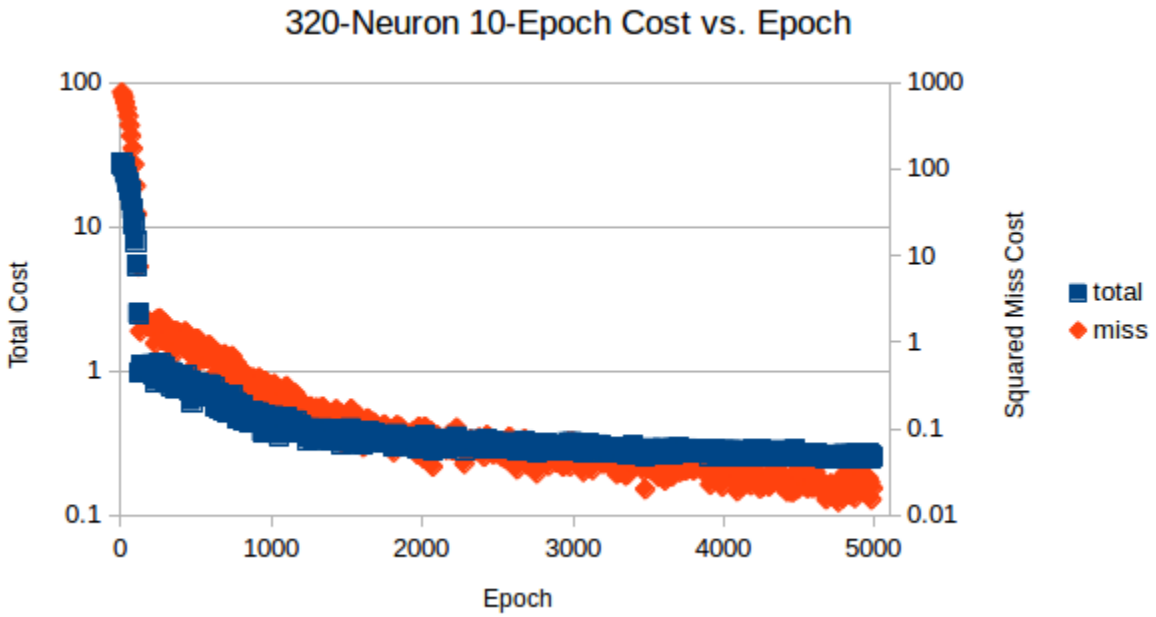

Figure 10. 320-Neuron Results

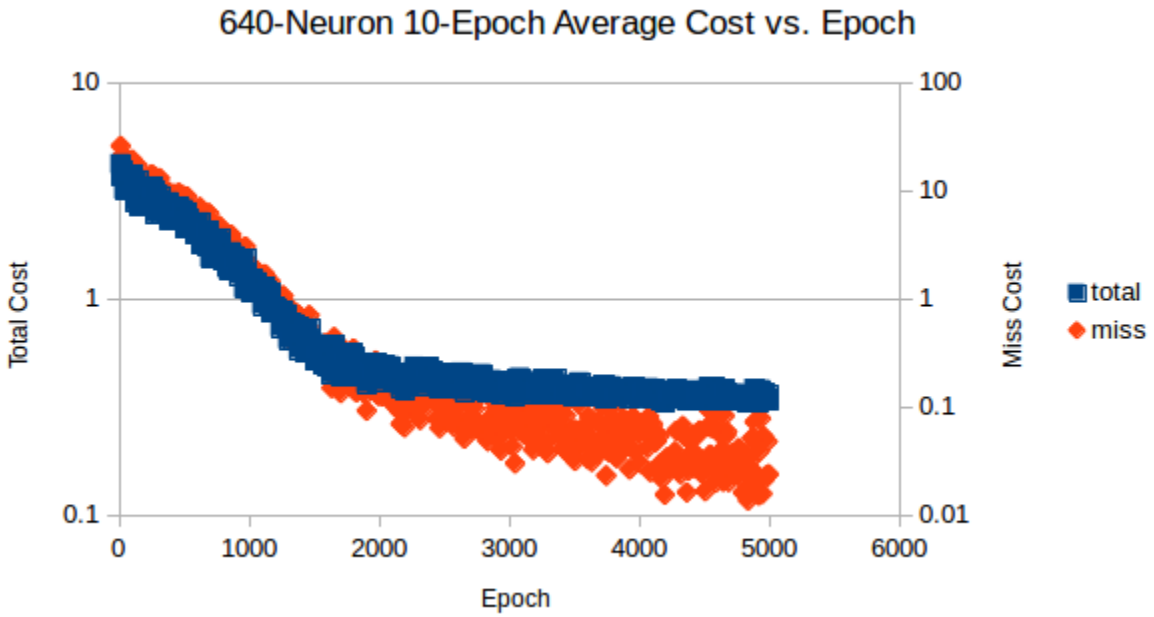

Figure 11. 640-Neuron Results

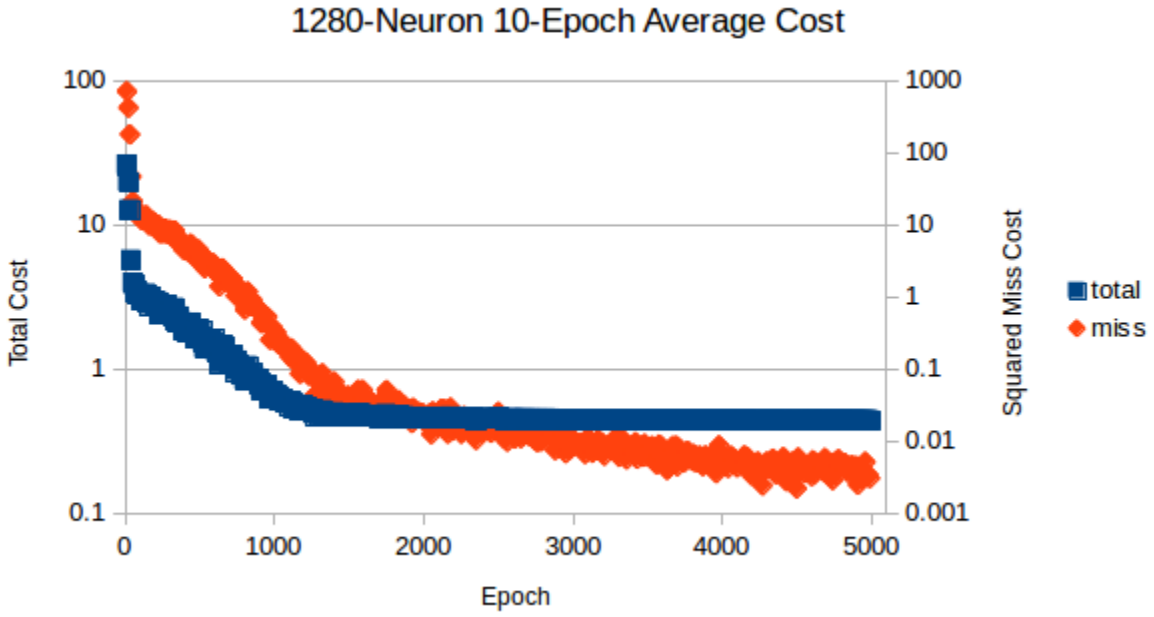

Figure 12. 1280-Neuron Results

AFRL provided 280 datapoints, meaning that the 320, 640, and 1280-neuron net-

works all have more neurons than input data, putting them at significant risk for eventual overfitting. It is also possible that having too many neurons may result in conflicting behavior, reducing the accuracy, particularly since the network was trained on the total cost rather than the miss cost. Using additional layers may mitigate this behavior somewhat, but more investigation would be required to confirm this theory. The AFRL-provided data is highly quantized along the electromagnet current dimension, a poor situation for training a neural network. The relative errors ([calculated - expected]/expected) are dependent on the network topology, but both the 20 and 40-neuron networks show nearly linear behavior, again indicative of network topology insufficient to capture the data's underlying structure, as shown in Figures 13 and 14.

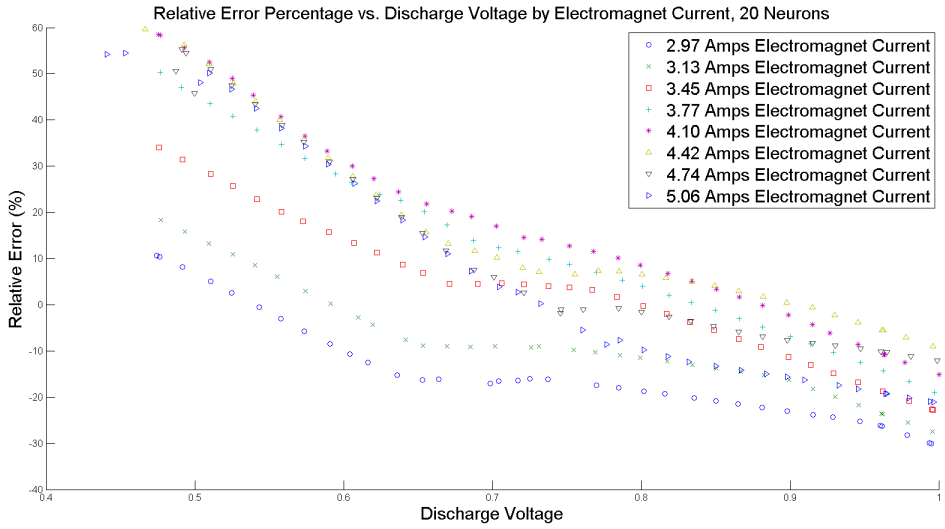

Figure 13. 20 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current

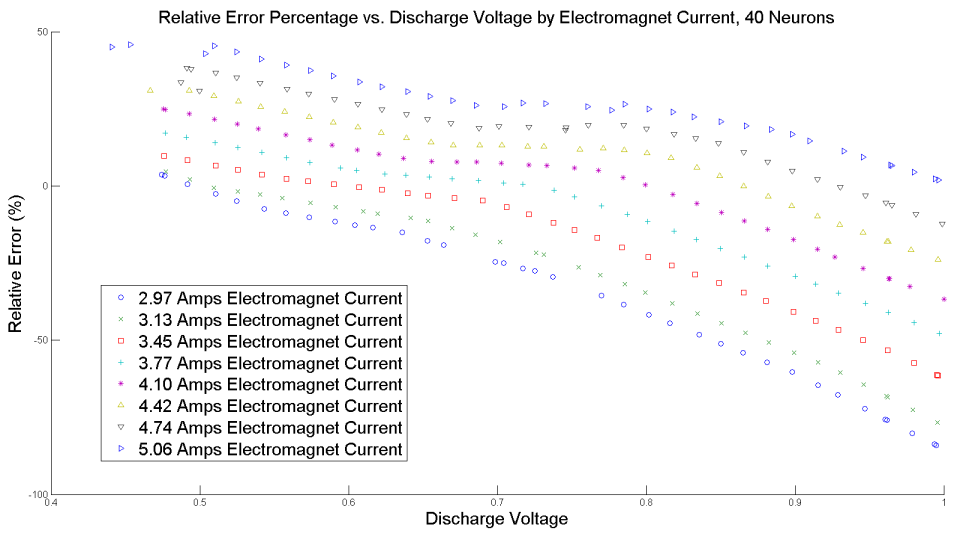

Figure 14. 40 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current

Other networks tested show variation based on the discharge current, suggesting that sufficient neurons exist to capture at least some of the underlying complexity in the data. The 80-neuron results seem to show less structure in the error measurement than the other networks. Its maximum error spread (maximum error-minimum error percentage) is approximately 40%, and error is notably higher towards the lower discharge voltages. Lower voltages also show larger errors with greater electromagnet currents, a structure that is shared with the 160-neuron network but vanishes with the larger networks. The 320-neuron network in particular has a very large error magnitude for the smallest electromagnet currents, with error shrinking with increasing electromagnet current. The 640-neuron network in particular shows a spectacularly large error spread (approximately 180%) in the lower voltages tested. Finally, the 1280 neuron network shows behavior consistent with polynomials across each electromagnet current, with an error spread never greater than 25%. Notably, the structures vary across electromagnet currents, suggesting a good model– or perhaps an exces-

sive number of neurons to model – the underlying structure. Plots of the relative errors are shown below. None of the models show structure in the error similar to the structure present in the raw data, shown in Figure 20, suggesting that the error is not driven by the value of discharge current in the AFRL-provided data. Instead, the errors stem from errors in the neural network's data modeling.

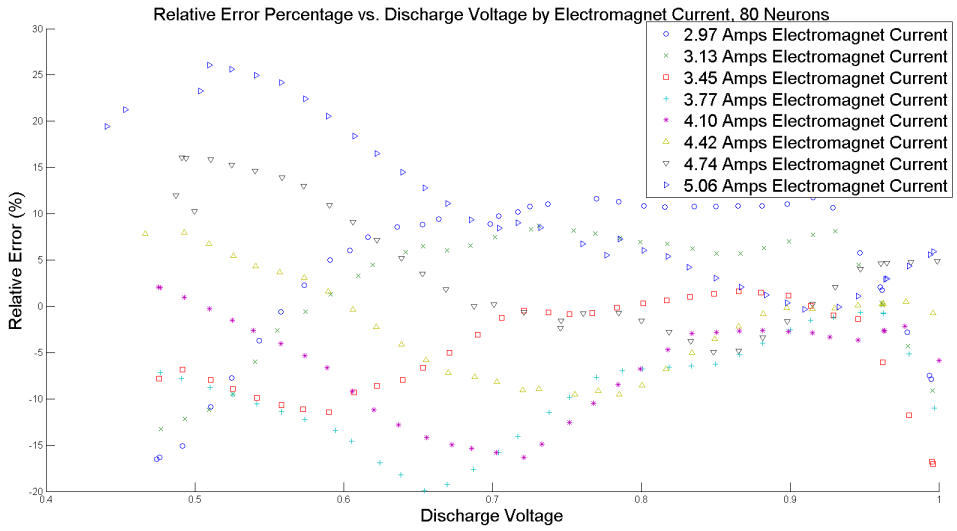

Figure 15. 80 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current

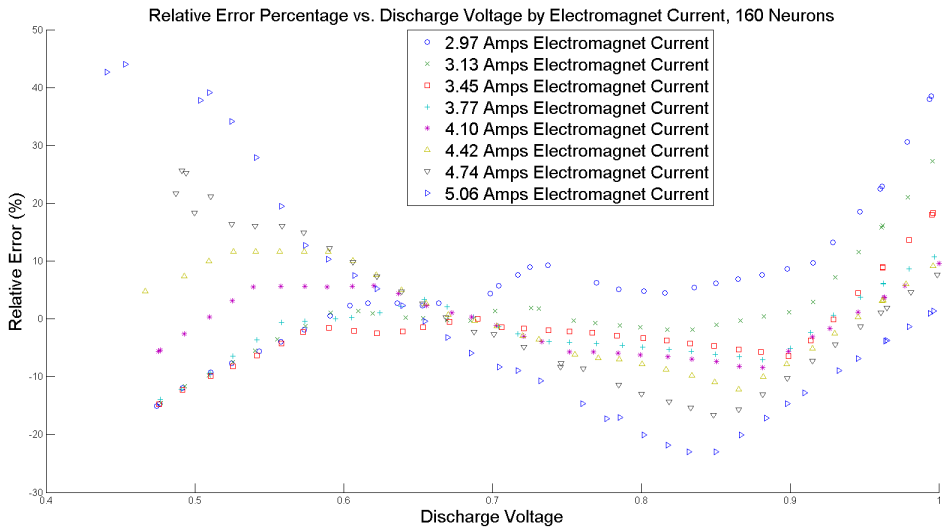

Figure 16. 160 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current

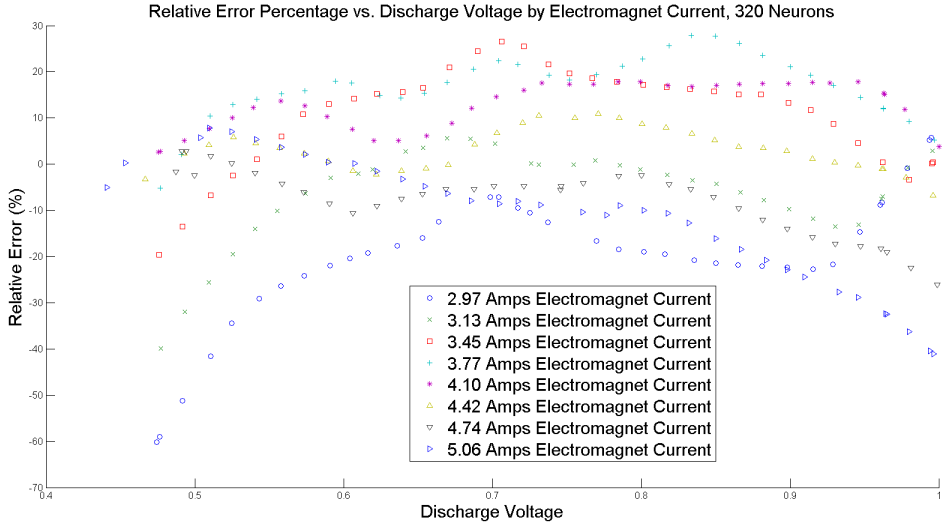

Figure 17. 320 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current

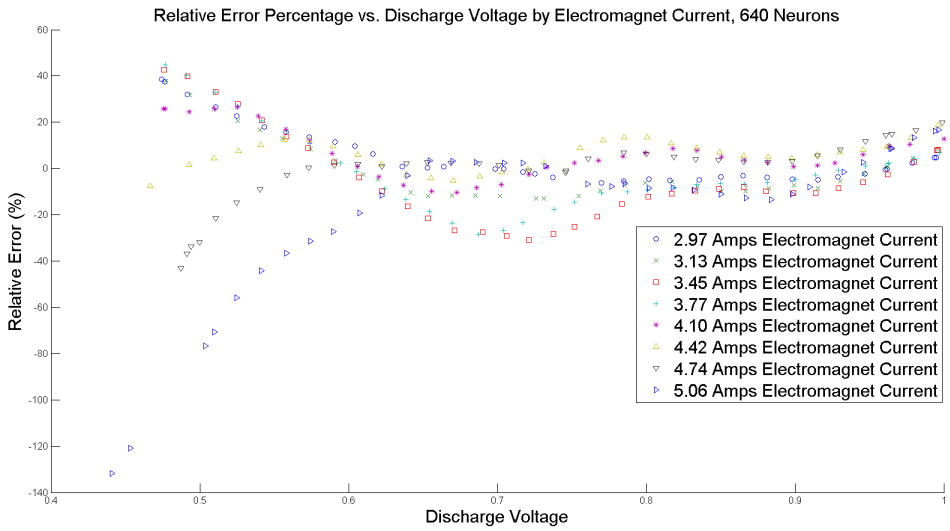

Figure 18. 640 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current

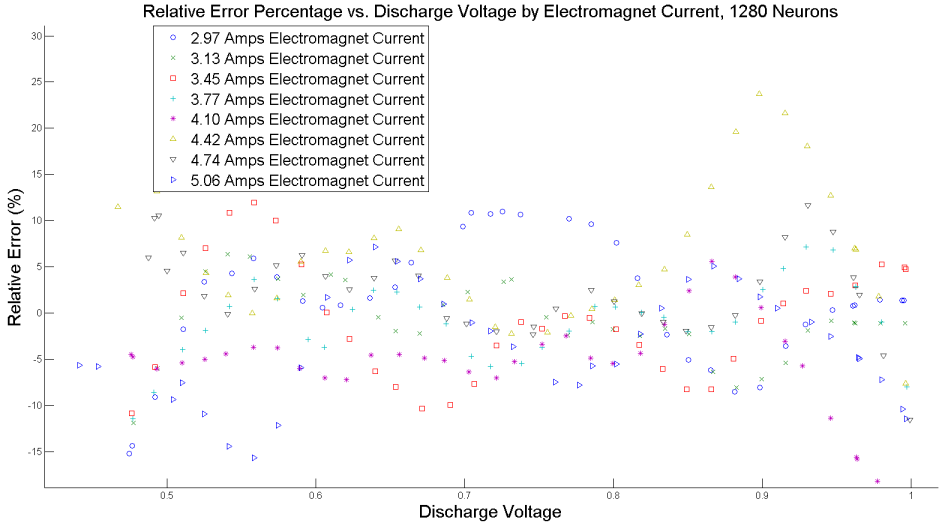

Figure 19. 1280 Neuron Error % vs. Normalized Discharge Voltage and Electromagnet Current

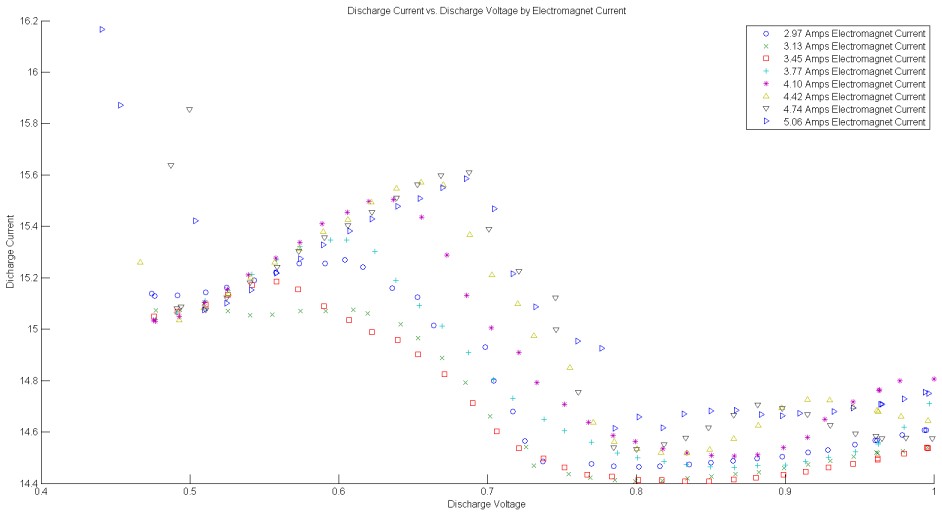

Figure 20. Discharge Current vs. Discharge Voltage by Electromagnet Current

4.2.2 Data Quantity Effects.

The validation data was used to examine the impact the quantity of training data had on the network's performance. As stated above, all networks acting on validation data used 80 neurons, removing network topology as a variable. Even with completely noise-free training data, Figures 21 and 22 clearly demonstrate that increasing the amount of data can have a highly nonlinear impact on the quality of the final network. The network trained on 240 data points had a final squared miss cost approximately three orders of magnitude greater than that of a network trained using 1200 datapoints. Further increasing to 2400 datapoints, however, made a difference of only about half an order of magnitude, suggesting that the nonlinear impact of adding data eventually transitions to merely linear improvements. This falloff could eventually become asymptotic, although such behavior is not demonstrated by the data present in this work. Importantly, the squared miss cost was lowest throughout the entire training procedure, including during the very first training step, showing that even with more datapoints and therefore chances to miss with a marginally-

trained network, increasing the number of datapoints improved network accuracy. Finally, both the 1200 and 2400-point cases were both improving faster than the 240-point case at the end of training. From this it seems logical to conclude that increasing data quantities cause improved network performance entirely independently of the time used to train the network; that is, two topologically-identical networks trained for the same number of epochs but with different quantities of data will always see the network trained on more data have more accurate results. This conclusion should be tested before acceptance as fact, preferably by testing with more than just three different levels of datapoints.

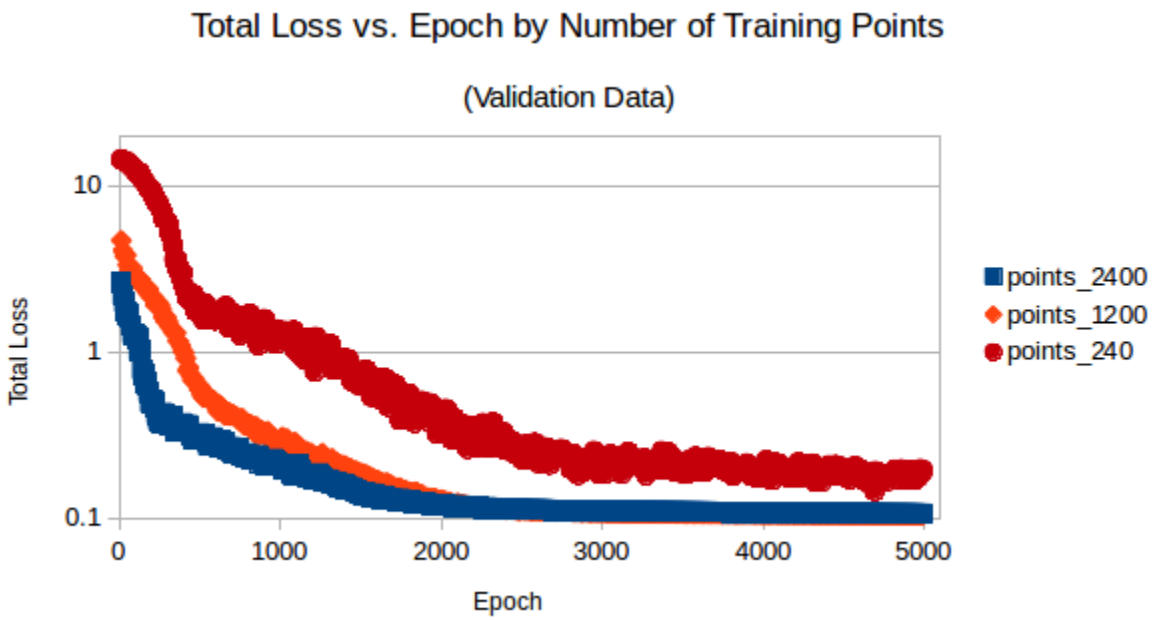

Figure 21. Effects of increasing training data quantity on total cost of the network

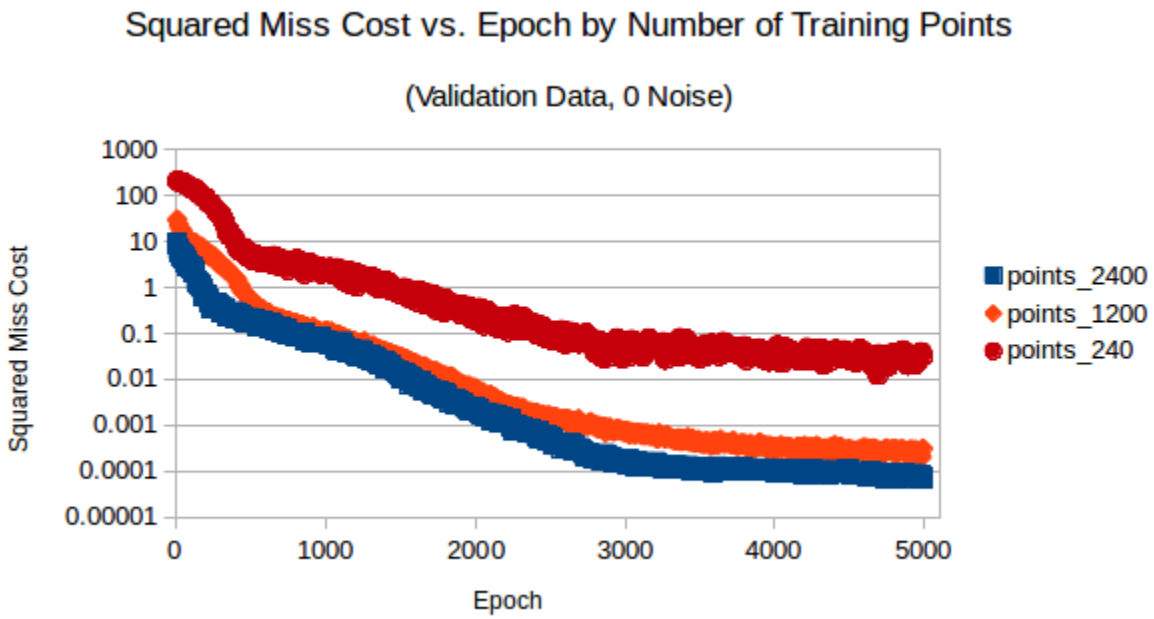

Figure 22. Effects of increasing training data quantity on miss cost of the network

The improvements in total network cost, though less dramatic than the squared miss cost, show similar results: using either 1200 or 2400 datapoints improved the total cost by approximately a factor of two, though the 240 point case was still improving when the training ended. Both the 1200 and 2400 point cases reached an asymptote of approximately 0.1 for the total loss just after the 2000th epoch, while the 240-point had yet to arrive at its asymptote. The asymptotic behavior for total cost is expected since there are a nonzero number of neurons, each of which has some weight that is included in the total cost. As expected given the regularization used, the continued improvement in miss cost coupled with the asymptotic total cost means that the weights must be the driving factor in the total cost in the 1200 and 2400-point cases. Continuing to increase network size may increase the total cost of the network due to the greater number of neurons used and the penalty included in the total cost (that is very useful for training purposes and regularizing), so selection of the best network should be based on the miss cost as it more directly reflects the

network's performance.

4.2.3 Noise Effects.

Noisy validation data was generated to examine the impact of noise in the training data on eventual network performance. The zero-noise dataset was used as a baseline for comparison, and noise levels up to 100% that of the generated polynomial data (i.e. a signal-to-noise ratio of 1) were examined in increments of 20%, and networks were trained using each of the three data quantities (240, 1200, and 2400 datapoints). Lower noise levels did not necessarily improve accuracy as shown by Figure 26, which shows that the network modeled data at 40% and 60% relative noise levels better than 20%. This difference is not reflected in plots of total cost, where lower noise invariably resulted in lower network cost. As such it is reasonable to conclude that the apparently-strange lower miss cost behavior is an artifact of training the network on the total cost for regularization rather than anomalous data or network behavior.

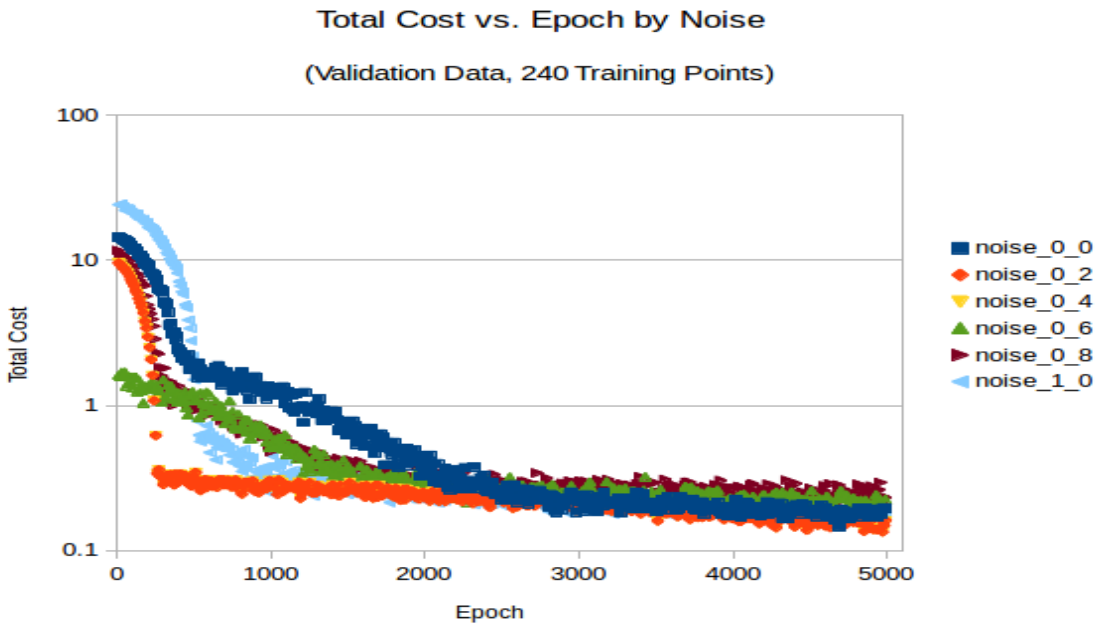

Figure 23. Effects of increasing noise on total cost of a network trained on 240 datapoints.

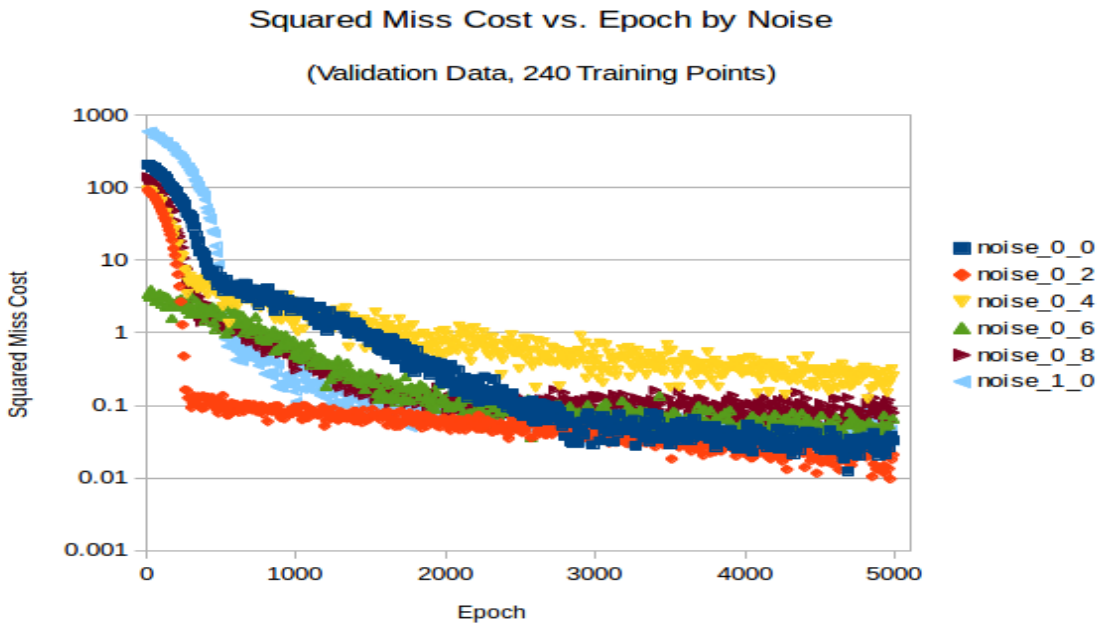

Figure 24. Effects of increasing noise on miss cost of a network trained on 240 datapoints.

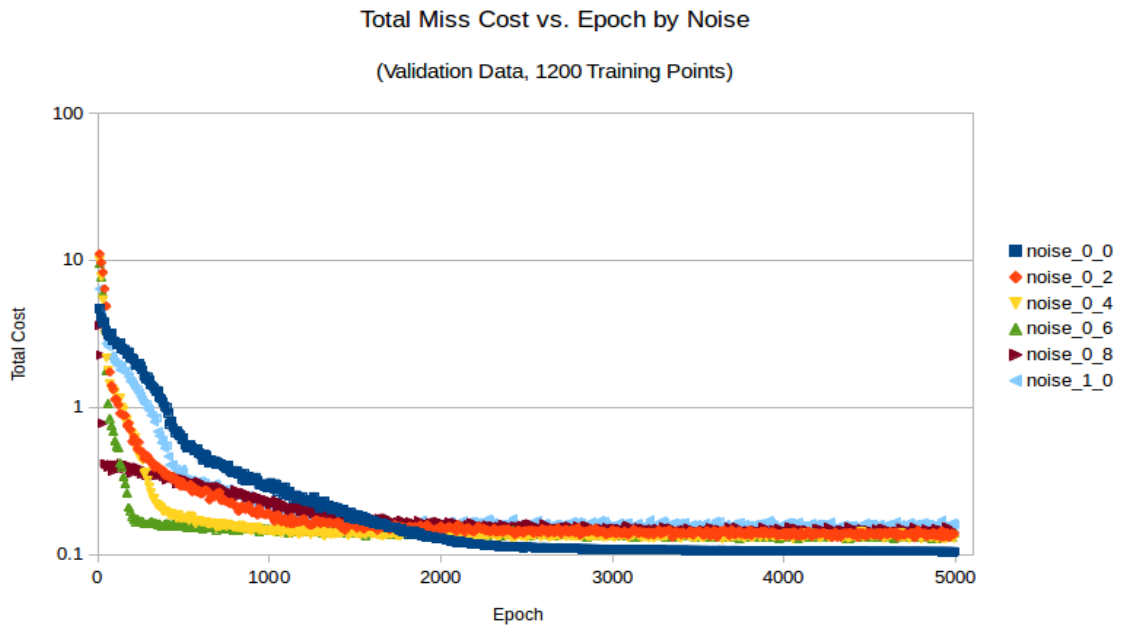

Figure 25. Effects of increasing noise on total cost of a network trained on 1200 datapoints

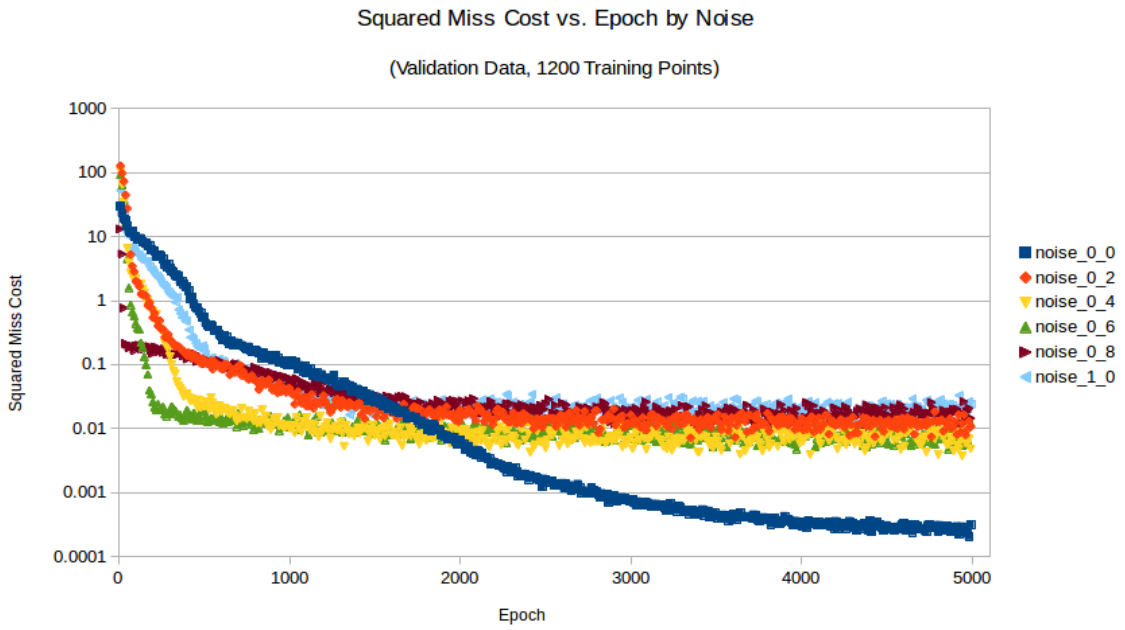

Figure 26. Effects of increasing noise on miss cost of a network trained on 1200 datapoints

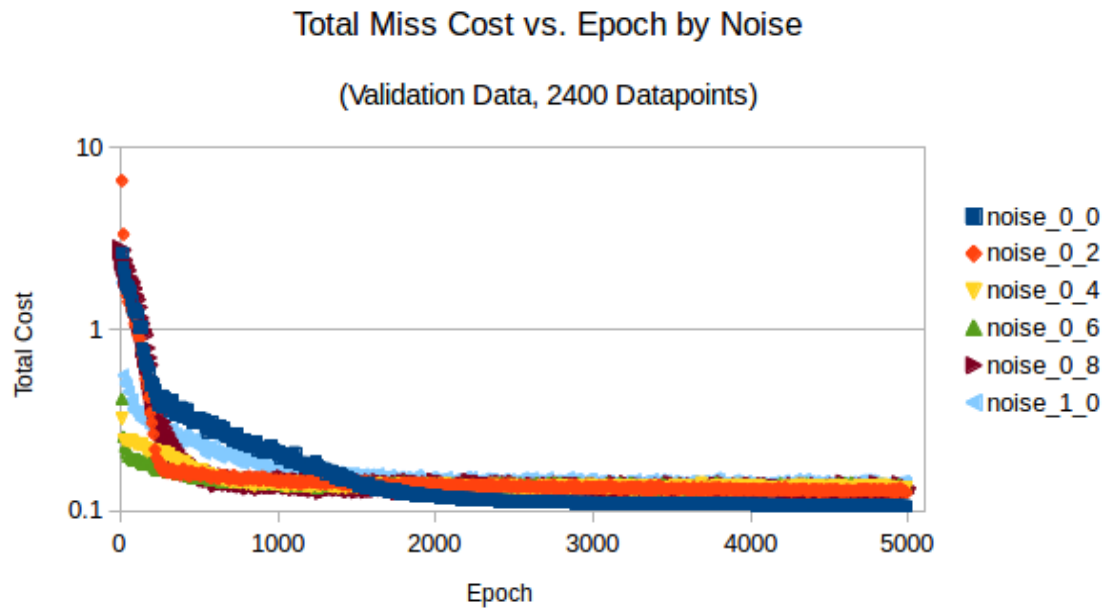

Figure 27. Effects of increasing noise on total cost of a network trained on 2400 datapoints

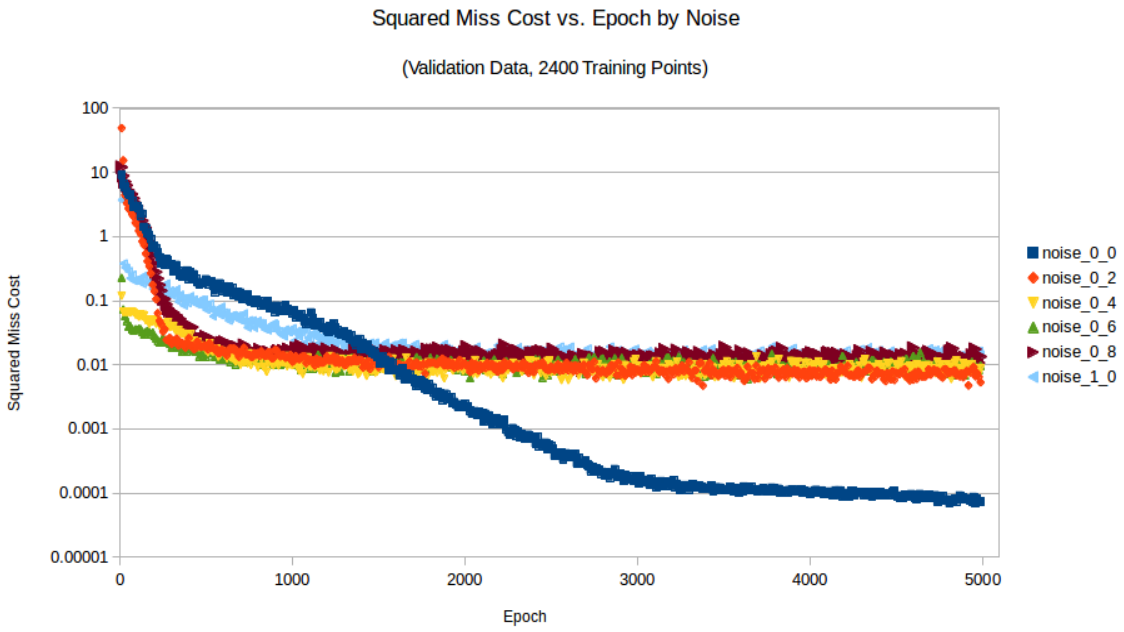

Figure 28. Effects of increasing noise on miss cost of a network trained on 2400 datapoints

Nonzero noise resulted in asymptotic or apparently-asymptotic accuracy regardless of the number of datapoints provided to the network. The 240-datapoint network was still slowly falling when training ended, but its fall rate was clearly slowing; given more training epochs, it would likely become fully asymptotic. Both the 1200 and 2400 datapoint noisy trials became asymptotic at a squared miss cost of 0.01, while the noisy trials with 240 datapoints were between approximately 0.05 and 0.5 squared miss cost when training ceased. The noiseless data was falling most rapidly in the 240-datapoint case and had not become asymptotic (although likely would eventually) in either the 1200 or the 2400-datapoint case, and in the latter two had several orders of magnitude better accuracy than even the least noisy data. This behavior is echoed in the plots of total cost vs. epoch, which show all noisy data becoming asymptotic before the 2000-epoch mark, while the noiseless trial was still rapidly improving. As expected, increasing the noise levels slowed the convergence of the total cost, reflecting the greater difficulty in modeling the underlying behavior. The number of provided datapoints showed a correlation with the number of epochs required to reach asymptotic error, again as expected.

5. Conclusion

This research developed and demonstrated the effectiveness of a spectral solver that can be integrated into TURF in a future work to solve Poisson's equation. Its performance was characterized against Poisson's equation and the results showed that for a wide range of transformation lengths and waveform component modes there was no error above machine-zero. At and above 20% relative noise, the solver also showed linearly increasing error with noise showing that in this domain noise is the sole source of error. Future work should examine the region between noiseless and 20% relative noise to determine the shape of the nonlinearity that lies there. These results are very encouraging, and suggest that the solver is ready for integration into TURF.

Additional avenues of investigation involving the spectral solver would include integrating the solver into TURF and duplicating Tran's simulation using the new solver, then comparing the results. Using TURF and the spectral solver to simulate Cunningham's experiment and comparing with his observed results the spoke-mode he mentions may give further insight into the cross-field mobility by better modeling the azimuthal electron waves and instabilities. Replicating Lafleur, et al.s simulations might have similar impact. From the mathematical and programming side, the spectral solver could be expanded to operate on more than a single dimension at once, although as a linear operator AFRL may find that unnecessary. The algorithm invoked is the Radix-2 FFT originally conceived by Gauss and could be improved with a more modern algorithm. Other future work could involve characterizing the solver's performance against still larger transformation lengths and number of component waveform modes, although increasing the transformation length is expected to improve performance insofar as it can be. The hardware-dependent nature of the above results suggest that the test should be repeated prior to implementing this algorithm on a new computing system or even following significant hardware changes

in a previously-tested system.

Additionally, a neural network was developed and demonstrated the effectiveness of even exceptionally crude networks in performing function regressions. The network created was a single-layer feedforward network using a leaky ReLU activation function in the hidden layer and trained on data provided by AFRL. The network preprocessing took advantage of significant a priori knowledge about the relationship between electromagnet voltage, magnetic field strength, and beam current, particularly the knowledge that the function is continuous and that the two input parameters (voltage and magnetic field strength) are entirely independent of each other. A more general approach cannot take advantage of such knowledge and should use more robust preprocessing methods. Despite this, even an 80-neuron network was trained with mean errors of significantly less than 1% and an error standard deviation below 10%.

While the standard deviation found in the neural network results is higher than may be desirable, it is reasonable for many industrial processes, particularly ones that are still undergoing improvement (e.g. a new production line coming online). The networks developed were exceptionally simple, and more complex networks would likely show better performance. Future work could investigate the use of additional hidden layers, different layer widths, and other activations to examine the effects on network performance. AFRL provided 280 datapoints, a very small quantity for machine-learning tasks. Additional data would serve to improve modeling the underlying phenomenology and combat overfitting, and with sufficient data regularization may not be required so the network could be trained on the miss cost directly. Performance would almost certainly improve if more training data were provided (as shown in 4.2.2), and future work could take more data from experiments or even simulations to use in training a network. While the results from this experiment showed

that a neural network can be used to model the output of a HET, it is important to remember that an operational networks output can only be treated as valid within the bounds of the training data.

The network presented here demonstrated its ability to model Hall thruster output data, suggesting that it will be useful for more interesting applications than those tested in this work. The limitations of the UAT imply that a neural network will probably not be able to effectively model the behavior of a Hall thruster outside the tested envelope, but the UAT simultaneously means that a network can be generated to approximate the tested values– and, perhaps more importantly, interpolate between the tested values– to an arbitrary accuracy, saving both computational time and potentially a great deal of time in the testing process for any given Hall thruster. This is especially true if the network is used to model underlying and poorly understood physics rather than merely engineering parameters. The cross-field electron mobility in particular may be a prime test case for this, particularly since there remain great questions about the underlying dynamics. A neural network model provides the advantage that the programmers need not know the underlying dynamics a priori, only provide the system with sufficient data to model them.

Appendix A.

A.1 Solver Code

A.1.1 ValidationFFT Header.

```
/*
 * ValidationFFT.h
 *
 *  Created on: Jun 19, 2018
 *      Author: Joseph Whitman
 */

#ifndef VALIDATIONFFT_H_
#define VALIDATIONFFT_H_

#include <stdio.h>
#include <cstdio>
#include <stdlib.h>
#include <complex>
#include <cfloat>
#include <iostream>
#include <fstream> //because iostream wouldn't play nice with
    ofstreams.
#include <valarray>
#include <vector>
#include <time.h>
#include <math.h>
```

```
#include <string>

typedef std::complex<double> Complex;
typedef std::valarray<Complex> CArray;

const double PI = 3.14159265358979323846;
const Complex IMAG = std::complex<double>(0.0, 1.0);

class Validation_FFT {
public:
        Validation_FFT();
        virtual ~Validation_FFT();
        void fft(CArray&);
        void ifft(CArray&);
        double computeMean(double*, double*, int);
        double computeStdev(double*, double, int);
};

#endif /* VALIDATIONFFT_H_ */
```

A.1.2 ValidationFFT Code.

```
/*
 * ValidationFFT.cpp
 *
 *  Created on: Jun 19, 2018
 *      Author: Joseph Whitman
```

```
 */

#include "ValidationFFT.h"

Validation_FFT::Validation_FFT() {
        // TODO Auto-generated constructor stub

}

Validation_FFT::~Validation_FFT() {
        // TODO Auto-generated destructor stub
}

double realStdev(CArray& experimental, double mean, int
    length){
        //this function computes the standard deviation
        double stdev =0;
        double junk=0;
        for(int ct =0; ct<length; ct++){
                junk = (experimental[ct].real()*experimental[
                    ct].real());
                stdev += (junk-mean)*(junk-mean)/length;
        }
        /*for(int ct = 0; ct< length; ct++){

        }*/
```

```
        stdev = sqrt(stdev);
        return stdev;
}

CArray getComplexMean(CArray& experimental, CArray&
    theoretical, int length){
        //this function computes the mean error
        CArray Cmean(1);
        for(int ct =0; ct < length; ct++){
                Cmean[0].real() = (experimental[ct].real()-
                    theoretical[ct].real());
                Cmean[1].imag() = (experimental[ct].imag()-
                    theoretical[ct].imag());
        }
        //mean /= length;
        return Cmean;
}

double computeMean(CArray& experimental, CArray& theoretical,
    int length){
        //this function computes the mean error
        double mean = 0;
        double junk1 = 0;
        double junk2 = 0;
        for(int ct =0; ct<length; ct++){
                //compute real and imaginary, then take the
```

```
            magnitude of the error
            junk1 = experimental[ct].real()-theoretical[
                ct].real();
            junk2 = experimental[ct].imag()-theoretical[
                ct].imag();
            mean += sqrt(junk1*junk1 + junk2*junk2);
        }
        mean /= length;
        return mean;
}

void manualCalc(CArray& x_n, CArray& X, int N){
        // Hold this so we only calculate it once rather than
            each loop
        // Hmmm...CPP threw a fit when I didn't cast N and
            wouldn't build.  Odd.
        std::complex<double> hold = -IMAG*2.0*PI/ (std::
            complex<double> (N));
        //Run the loop to manually calculate the values of
            the DFT
        for (int k=0; k<N; k++){
            for (int n = 0; n<N; n++){
                X[k] += x_n[n]*std::exp(hold*(Complex
                    (n))*(Complex (k)));
            }
        }
```

```
        /* Debug lines
        for(int ct=0; ct<N; ct++){
                printf("ct:%.12d\t%12.12f\t%12.12f\n",ct, X[
                    ct].real(), X[ct].imag());
        }*/


}

// fft in-place radix-2 transform
void fft(CArray& x)
{
    const size_t N = x.size();
    if (N <= 1) return;

    // divide
    CArray even = x[std::slice(0, N/2, 2)];
    CArray  odd = x[std::slice(1, N/2, 2)];

    // conquer
    fft(even);
    fft(odd);

    // combine
    for (size_t k = 0; k < N/2; ++k)
    {
```

```
        Complex t = std::polar(1.0, -2 * PI * k / N) * odd[k
            ];
        x[k     ] = even[k] + t;
        x[k+N/2] = even[k] - t;
    }
}

void ifft(CArray& x)
{
    // conjugate the complex numbers
    x = x.apply(std::conj);

    // forward fft
    fft( x );

    // conjugate the complex numbers again
    x = x.apply(std::conj);

    // scale the numbers
    x /= x.size();
}

// Test case— delta function
void TestDelta(int length){
        /*
         * TestDelta: This function runs a unit test to make
```

```
    sure that the
 * FFT returns the correct results if given a delta
    function-- a single
 * nonzero value in the data structure.  It does so
    by comparing the
 * results of the FFT function elsewhere in this file
     with the results
 * of a DFT computed using the classic method.
 *
 */
// Declare the three data structures and their length
CArray data(length), manual(length), manual_out(
   length);

srand (time(NULL));                //initialize the
   random seed

/* Don't bother setting any values to more than 1
 * since FFT is a linear operation, you can always
    normalize down.
 * That said, initialize the delta value to a nonzero
    .
 */

int ind = rand()%length;
data[ind] = 1;              // this structure goes
```

```
        through the FFT
    manual[ind] = 1;          // this structure goes
        through the DFT

    printf("Single-cell test: loading cell: %d\n", ind);
    //Run the manual computation
    manualCalc(manual, manual_out, length);

    //Run the FFT computation— IFFT is just an
        invocation of the FFT function here.
    fft(data);

    //Print the comparisons here.
    for(int ct=0; ct<length; ct++){
            printf("ct:%.12d\tFFT: %12.12f+%12.12fi\t\
                tDFT: %12.12f+%12.12fi\n",
                        ct, data[ct], manual_out[ct])
                            ;
    }
}


void TestRandFunc(int n_samples, int n_waves, FILE* error_out
    , double nz=0.0){
    /*
     * This function builds a random function of sines
```

```
    and cosines
 * and attempts to integrate it twice using the FFT
    above, then
 * compares it against the known integration.
 *
 * Note that the frequencies are limited to integers
    here due to
 * the assumption of a periodic domain, though that
    need not be
 * generally true.
 *
 * Note that the Nyquist limit means that the maximum
     frequency
 * allowable is half the number of samples.
 *
 * This function relies on a number of for loops, and
     they could
 * almost certainly be condensed to increase
    efficiency.  They
 * were left this way for clarity and ease of reading
    .
 */

//set up problem bounds and derived quantities
double xmin = 0.0;
double xmax = 1.0;
```

```
double L = xmax-xmin;
double dx = L/n_samples;
double dk = 2*PI/L;

//set the maximum frequency allowed
int max_freq = std::floor(n_samples/2.0);

//set up the output file
/*char buffer[4];
std::sprintf(buffer, "%4.3f", nz);
std::string fname = "random_test_function_"+std::
    string(buffer)+".txt";
//std::ofstream out_file("random_test_function.txt");
std::ofstream out_file(fname.c_str());*/

//set up the data structures-- use CArrays to avoid
    type-operator issues.
CArray x(n_samples);              // this just stores
    the x-coordinate of each point
CArray kvec(n_samples);           // divisor for each
    frequency
CArray data(n_samples);           // this will be the
    structure to integrate
CArray known(n_samples);          // this will be the
    structure with known output
CArray freqs(n_waves);            // store the
```

```
    frequencies here
CArray amps(n_waves);              // store the
    amplitudes here

/* initialize the random seed— not necessary when
 * TestDelta runs beforehand, but for portability
 */
srand(time(NULL));

/* Get the frequency and amplitude values
 * Use all cosines because that way I'm guaranteed a
    normalized 1 value at
 * the first point, and because sines are just a
    phase shift.
 */
for (int w = 0; w < n_waves; w++){
        //frequency between 0 and max_freq— add 1 to
            make max_freq possible
        freqs[w] = rand() % max_freq+1;
        //amplitude by dividing the result by
            whatever the machine allows as rand_max
        amps[w] = static_cast <float> (rand()) /
            static_cast <float> (RAND_MAX);
        //printf("ampl: %12.12f\tfreq: %12.12f\n",
            amps[w].real(), freqs[w].real());
}
```

```
//normalize the amplitudes so they sum to one
amps /= amps.sum();

// Set up the values for the test and known-answer
   array
for (int p = 0; p < n_samples; p++){
        x[p] = std::complex<double>(dx*p, 0.0); //set
            up X
        //set up the k-vector
        if(p <= n_samples/2){ double hold = dk*p;
                kvec[p] = std::complex<double>(dk*p,
                    0.0); }
        else{ kvec[p] = -kvec[n_samples-p]; }
        /*
        printf("dk_list\tct: %d\tdx: %15.15f\tx:
            %15.15f\tdk: %15.15f\tkvec: %15.15f\n",
                        p, dx, x[p].real(), dk,kvec[p
                            ].real());
        */
        for (int w = 0; w < n_waves; w++){
                //test function fi.real()rst
                data[p] += amps[w]*cos(2*PI/L*freqs[w
                    ].real()*x[p].real())
                                +nz*(-1+
                                static_cast <float> (
```

```
                                        rand()) /(
                                        static_cast <float
                                        > (RAND_MAX/(2))))
                                        ;
                                //known results: second integral of
                                    data
                                //not a typo-- squaring is slower
                                    than double-multiply.
                                known[p] += amps[w].real()*(1./(2*PI/
                                    L*freqs[w].real())*
                                                1./(2*PI/L*freqs[w].
                                                    real()))*
                                                cos(2*PI/L*freqs[w].
                                                    real()*x[p].real()
                                                    );
                        }
                }

/*      for (int p = 0; p < n_samples; p++){
                printf("Before FFT: ct: %d\t%15.15f+%15.15fi\
                    n", p, data[p]);
        }
*/

        //execute forward FFT
        fft(data);
```

```
/*
        for (int p = 0; p < n_samples; p++){
                printf("After FFT: ct: %d\t%15.15f+%15.15fi\n
                    ", p, data[p]);
        }
*/

        //divide by k-vector; integration constant (DC) known
            zero in this case
        for (int p = 0; p < n_samples; p++){
                if(p==0) data[p] = std::complex<double>(0,0);
                else{ data[p] /= (kvec[p]*kvec[p]); }
        }

/*
        for (int p = 0; p < n_samples; p++){
                printf("After Division: ct: %d\t%15.15f
                    +%15.15fi\n", p, data[p]);
        }
*/

        //execute reverse FFT
        ifft(data);

        for (int p = 0; p < n_samples; p++){
```

```
                printf("ct: %d\tCalculated: %15.15f+%15.15fi\
                    t\tKnown: %15.15f+%15.15fi\n",
                                p, data[p], known[p]);
        }

        //compute error mean
        double mean = computeMean(data, known, n_samples);

        //compute standard deviation
        //CArray Cmean(1);
        //Cmean = getComplexMean(data, known, n_samples);
        double stdev = realStdev(data, mean, n_samples);

        std::fprintf(error_out, "%d\t%d\t%30.30f\t%30.30f\t
            %30.30f\n",
                        n_samples, n_waves, mean, stdev,
                            DBL_EPSILON);


}

std::vector<double> crossprod(std::vector<double> A, std::
    vector<double> B){
        /*
         * This helper function takes two 3-vectors and
            returns their cross product.
```

```
        */
        std::vector<double> retvec (3);
        retvec[0] = A[1]*B[2]-B[1]*A[2];
        retvec[1] = A[2]*B[0]-B[2]*A[0];
        retvec[2] = A[0]*B[1]-B[0]*A[1];

        return retvec;
}

int main()
{
        //Set up arguments for each test
        int Delta_test_length = 16;             // must be a
           power of two
        int num_rand_func_bins = 1024;    // must be a power
           of two
        int num_waves = 8;
        int max_characterization_bins = 1024;
        int max_characterization_waves = 64;
        double RandTestNoise = 1.0;
        double NzStep = 0.2;


        //TestDelta
        TestDelta(Delta_test_length);
```

```
//Test with the random waveform
FILE* char_err = fopen("noisy_characterization_error.
   txt", "w");
for(int nbins=4;nbins<=num_rand_func_bins;nbins*=2){
        for(int NzMult=0; NzMult*NzStep <=
           RandTestNoise; NzMult++){
                TestRandFunc(nbins, num_waves,
                   char_err, NzMult*NzStep);
        }
}
fclose(char_err);

/*
 * Characterize the solver by running it against up
    to 1024 bins
 * (powers of 2 only) and up to 64 waveforms
 *
 * This is the characterization test
 */
FILE* char_err2 = fopen("
   poisson_characterization_error.txt", "w");
for(int ct=4; ct <= max_characterization_bins; ct*=2)
   {
        for(int waves=1; waves<
           max_characterization_waves; waves++){
                if (waves > ct) { break; }
```

```
                        TestRandFunc(ct, waves, char_err2,
                            0.0);
                }
            }
            fclose(char_err2);

            printf("\nDone!\n");
        return 0;
    }
```

A.2 Neural Network Code

A.2.1 FCNetwork.py.

```
1  # -*- coding: utf-8 -*-
2  """
3  Author: Joseph Whitman
4  Last edited: 08 July 2018
5  File: FCNetwork.py
6
7  This file allows the creation of a fully-connected network object
       , and
8  keeps all this code out of the already-long approximator code.
9  """
10
11 import tensorflow as tf
12
13
14 class FCNetwork:
```

```
15      """This class creates a fully-connected network
16
17      This layer should ease the use of multilayer perceptrons in
            other coding.
18      It needs each of the following inputs: a tuple of layer
            widths, a tuple of
19      layer activation functions, and a tuple giving the shapes of
            the input and
20      output tensors."""
21
22      wb_list = None # Defined here to make it an attribute
23
24      def addLayer(self, layer, activ):
25          if activ == 'relu':
26              return tf.nn.relu(layer)
27          elif activ == 'leaky_relu':
28              return tf.nn.leaky_relu(layer)
29          elif activ == 'relu6':
30              return tf.nn.relu6(layer)
31          elif activ =='elu':
32              return tf.nn.elu(layer)
33          elif activ == 'selu':
34              return tf.nn.selu(layer)
35          elif activ == 'softplus':
36              return tf.nn.softplus(layer)
37          elif activ == 'softsign':
38              return tf.nn.softsign(layer)
39          elif activ == 'sigmoid':
```

```
40              return tf.nn.sigmoid(layer)
41          elif activ == 'tanh':
42              return tf.nn.tanh(layer)
43          else:
44              raise ValueError('How are you even seeing this error?
                    '+\
45              'The program should have already thrown an exception
                   on whatever'+\
46              'you even did!  Beep boop does not compute.')
47
48      def __init__(self, layer_widths, layer_activs, inlen, X, ):
49          """Establishes the fully connected network.
50
51          Inputs:
52          layer_widths: a tuple containing the layer widths
53          layer_activs: a tuple of the layer activation functions,
                per layer
54          X: a placeholder for the input tensor
55          inlen: the size of the input vector; needs to be a vector
                or problems will arise
56
57          This function assumes that X is a vector and that it's
                looking for a
58          single output; giving it anything else, in either regard,
                will cause
59          some significant problems.
60          """
61          # regression, so only one output.
```

```
62          outlen = 1
63
64          # basic error checking— throw an exception if the user
                tries to give a
65          # different number of layer widths and activation
                functions
66          if len(layer_widths) != len(layer_activs):
67              raise ValueError("Error: layer_widths and
                    layer_activs must be the same length")
68          # throw a different error if the user tries to give an
                invalid activation fcn
69          ok_activs = ['relu', 'leaky_relu', 'relu6', 'elu', 'selu'
                , 'softplus',
70                       'softsign', 'sigmoid', 'tanh']
71          if (set(layer_activs)-set(ok_activs)) != set():
72              raise ValueError("Error!  Unacceptable activation
                    function given. "+\
73                               "You gave: " + str(layer_activs) + "
                                    \n"+\
74                               "Acceptable values are: "+str(
                                    ok_activs))
75
76          self.layer_widths = layer_widths
77          self.layer_activs = layer_activs
78
79          # create an empty list
80          layer_widths.insert(0, inlen)
81          layer_widths.append(outlen)
```

```
82          self.wb_list = [[None]*2]*(len(layer_widths)-1)
83
84          # set weights and biases
85          for ct, val in enumerate(layer_widths[:-1]):
86              self.wb_list[ct] = [tf.Variable(tf.random_normal([val
                    , layer_widths[ct+1]])),
87                      tf.Variable(tf.random_normal([layer_widths[ct
                        +1]]))]
88
89          # print(str(wb_list))
90          # make the layers
91          layers = []
92          # make the first one
93          layers.append(tf.add(tf.matmul(X, self.wb_list[0][0]),
                self.wb_list[0][1]))
94          layers[0] = self.addLayer(layers[0], layer_activs[0])
95          # print('layer 1: ' +str(layers[0]))
96          # make the next n-1 layers. Enumerate will protect
                against having no
97          # layers here like someone will eventually try to do.
98          # note the use of negative 2 rather than negative one--
                ending at -1
99          # causes the sequence to end at the last element, using
                -2 ends at the
100         # penultimate element.
101         for ct, val in enumerate(layer_widths[1:-2]):
102             ct += 1 # because enumerate forces to start at 0
103             layers.append(tf.add(tf.matmul(layers[ct-1], self.
```

```
                wb_list[ct][0]), self.wb_list[ct][1]))
104             layers[ct] = self.addLayer(layers[ct], layer_activs[
                    ct])
105         # okay, now we make the last element
106         layers.append(tf.matmul(layers[-1], self.wb_list[-1][0])
                + self.wb_list[-1][-1])
107         #layers[-1] = self.addLayer(layers[-1], 'tanh')
108         self.Model = layers[-1]
109
110
111     def GetModel(self):
112         return self.Model
113
114     def GetWeights(self):
115         """This function is a getter for the weights.
116         In retrospect, I really should have made those a numpy
                array
117         instead of a basic list.
118
119         Returns a numpy array of the weights
120         """
121         weights = []
122         for x in range(0, len(self.wb_list)):
123             weights.append( self.wb_list[x][0] )
124         return weights # tf.convert_to_tensor(weights)
```

A.2.2 Master_Approximator.py.

```
1 # -*- coding: utf-8 -*-
```

```
2  """
3  Author: Joseph Whitman
4  Last edited: 08 July 2018
5  File: Master_Approximator.py
6
7  This file runs a neural network either in validation ('val') or
       experimental
8  ('exp') mode; if run in validation mode, it will generate a
       nonlinear 2D
9  polynomial function with randomly-chosen, arbitrary coefficients
       between 0
10 and 50. Larger coefficients are possible but won't make any
       difference, and
11 this range was chosen only to demonstrate the effect of possibly-
       large
12 coefficients that better reflect real, general data than the 0-1
       coefficients
13 that the numpy.random.rand() function would otherwise return. If
       run in
14 experimental mode, the program will load the AFRL-provided data
       file and use
15 that as the inputs and target parameters. All values will be
       scaled to the
16 range 0-1 in order to prevent any one feature of the data from
       artificially
17 dominating the others when training the network.
18
19 The network itself is a network object in FCNetwork.py. Much of
```

```
    the network
20 setup in this file could be offloaded to other files, but was
    kept here for
21 clarity and to let this more easily demonstrate how to set up and
     train a
22 network.
23
24 """
25 import tensorflow as tf
26 import numpy as np
27 import FCNetwork
28 from copy import copy
29 from numpy import genfromtxt, zeros
30 from numpy.random import randn
31
32
33 def getData(mode, numel=None, nz=0.0):
34     """
35     This function returns a tensor of the inputs (X) and outputs
         (Y) that the
36     network is using.  If set to validation mode, it will
         generate the data,
37     otherwise it will extract the data from the AFRL-provided
         file.
38
39     Inputs:
40     -mode ----> 'val' for validation mode, 'exp' to run on AFRL
         data
```

```
41      -numel ---> number of elements to generate for validation
            mode.
42                  Note: this is ignored in 'exp' mode!
43      """
44      if mode == 'val':
45          if numel is None:
46              raise ValueError("Error!  Number of elements must be
                    specified" +
47                               " in the selected mode!")
48          # Generate the data according to how many elements are
                desired
49          # only generating 2-d data because AFRL data is 2d
50          print("Generating synthetic data!")
51          invals = np.array([np.linspace(0, 1, numel),
52                             np.linspace(0, 1, numel)]).transpose
                                 ()
53          # print(invals)
54          # The multipliers on this function were chosen
                arbitrarily
55          coeffs = 50*np.random.rand(4,)
56          # coeffs = [2, 3, 1, 4] # debug line
57          outvals = coeffs[0]*invals[:, 0]**2+coeffs[1]*invals[:,
                0]*invals[:, 1] +\
58                    coeffs[2]*invals[:, 1] + coeffs[3]*invals[:, 1]
                          + nz*np.sum(coeffs)*randn(numel)**2
59          # Create a file containing the synthetic data for
                comparison later
60          with open(mode + "_" + str(numel) + "_" + str(nz) + ".txt
```

```
            ", 'w') as f:
61              for x in range(invals.shape[0]):
62                  f.write(
63                      str(invals[x]).strip('[]').replace(' ', ',')
                            + ',' +
64                          str(outvals[x]) + '\n'
65                  )
66
67      elif mode == 'exp':
68          # Read in the data file as provided by AFRL
69          # In this case, I have only a single, fairly small file
                of input data, so
70          # I can get away without enqueueing the data.  Doing so
                is clunky and
71          # unnecessary for this, so I'm not going to do so here.
72          path = 'small_Processed_IVB_data.csv'
73          inputs = genfromtxt(path, delimiter=',')
74          print("Loading from file!")
75          outvals = inputs[:, 2]
76          invals = inputs[:, 0:2]
77          # print(invals)     # Debug line
78          # print(outvals)    # Debug line
79      else:
80          raise ValueError("Error!  Mode provided was " + str(mode)
                + ", but " +
81                           "only 'val' and 'exp' are valid modes,
                                and must be strings!")
82
```

```
83      return invals, outvals
84
85
86  def scaleValues(data):
87      """
88      This function scales each column of the data by its maximum
            value.  It's
89      only intended for use on columns of positive values, so be
            careful.
90
91      param:data: an array-like structure of numbers with at least
            one col
92      return: scaled data and the max/min in each dimension
93      """
94
95      # check for errors
96      if len(data.shape) > 2:
97          raise ValueError("Error: this function can only operate
                on " +
98                           "tensors of rank 2 at a maximum to avoid
                                confusion, etc.")
99      # set up the data we'll need
100     try:
101         ncols = data.shape[1]
102     except IndexError:
103         # This can occur if there is only one column.
104         ncols = 1
105
```

```
106     bounds = zeros((2, ncols))
107     # execute the loop
108     for x in range(0, ncols):
109         if ncols == 1:
110             # this is somewhat kludgy but will work and is clear
111             bounds[0, x] = min(data[:])
112             bounds[1, x] = max(data[:])
113             if bounds[0][x] < 0:
114                 raise ValueError("Warning!  Input data has
                        negative values! " +
115                                  "This function can only handle
                                     positive-valued input.")
116             data[:] /= bounds[1, x]
117             # This avoids a potential shape problem later by
                    forcing
118             # one column in the numpy array's metadata despite it
                     being
119             # de facto true anyway.
120             data = data.reshape((data.shape[0], 1))
121         else:
122             # Set the bounds— these will get returned
123             bounds[0, x] = min(data[:, x])
124             bounds[1, x] = max(data[:, x])
125             if bounds[0, x] < 0:
126                 raise ValueError("Warning!  Input data has
                        negative values! "+
127                                  "This function can only handle
                                     positive-valued input.")
```

```
128                 data[:, x] /= bounds[1, x]
129         return data, bounds
130 
131 
132 # Create the network
133 
134 if __name__ == "__main__":
135         # decide on inputs
136         layer_widths = [80]
137         layer_activs = ['leaky_relu']
138         num_synth_pts = 1200
139         synth_func_nz = 0.0
140         mode = 'val'
141 
142         fname = mode+"_"+str(layer_activs[0])+"_"+str(layer_widths)
143         if mode == 'val':
144             fname += "_"+str(num_synth_pts) + "_" + str(synth_func_nz
                    )
145         fname += '.txt'
146 
147         f2name = mode + str(layer_widths)+"_net-eval.txt"
148         if mode == 'val':
149             f2name = 'nz'+str(synth_func_nz) + "_"+str(num_synth_pts)
                    + f2name
150 
151         f2 = open(f2name, 'w')
152 
153         # Get the data
```

```
154     iv, ov = getData(mode, num_synth_pts, synth_func_nz)
155     # print("inputs: ", iv)       #debug line
156     # print("Outputs: ", ov)      #debug line
157
158     # Scale the data
159     siv, input_bounds = scaleValues(iv)
160     sov, output_bounds = scaleValues(ov)
161     # print("inputs: ", siv)      #debug line
162     # print("Outputs: ", sov)     #debug line
163     nfeatures = siv.shape[1]      # Use this a bunch, just make it
            a variable
164
165     # Open the output file in write mode
166     # If this fails, I don't want the rest of this running
167     f = open(fname, 'w')
168
169     # Create variables for the NN
170     X = tf.placeholder(tf.float32, [None, nfeatures])
171     Y = tf.placeholder(tf.float32)
172
173     # Create the neural network
174     net = FCNetwork.FCNetwork(layer_widths, layer_activs,
            nfeatures, X )
175     logits = net.GetModel()
176
177     # generate the loss function description
178     Beta = .0001 # This is a multiplier for the weight costs
179     miss_cost = tf.losses.mean_squared_error(Y, logits)
```

```
180     netweights = net.GetWeights()
181     weightcost = 0
182     for x in netweights:
183         weightcost += Beta*tf.nn.l2_loss(
184             tf.convert_to_tensor(x)
185         )
186     total_cost = tf.sqrt(weightcost+miss_cost)
187
188     # Choose the training operator
189     train_op = tf.train.AdadeltaOptimizer(0.001, rho=0.95).
            minimize(total_cost)
190
191     # initialize the vars
192     init = tf.global_variables_initializer()
193
194     # Determine how many samples in training vs. test
195     num_training_points = int(.8*len(siv))
196     num_test_points = len(siv)-num_training_points
197
198     # Use the context manager to run each session, it'll make
            life easier
199     with tf.Session() as sess:
200         tf.global_variables_initializer().run()  # initializes
                all vars
201
202         best_miss_cost = 0       # store the lowest /miss/ (not
                total) cost seen
203         lowest_miss_net = 0      # store the network with the
```

```
        above miss cost
204
205         # Execute the train-test loop as many times as is
                necessary.  While
206         # the approach shown here is one valid method— just run
                for a very
207         # large number of iterations— it is also naive in that
                it will
208         # continue training the network whether or not it has a
                good solution.
209         # Networks optimized for a production environment should
                iterate
210         # until some good solution is reached, with that good
                solution being
211         # something to discuss with the customer.
212         for loop in range(1, 5000):
213
214             # Shuffle the input variables
215             # Technically this is not strictly necessary: if I'd
                    recombined
216             # siv and sov above, I could just shuffle the
                    resulting array.
217             # I've done this for clarity's sake.
218             concat_net_ins_outs = np.concatenate((siv, sov), axis
                    =1)
219             np.random.shuffle(concat_net_ins_outs)
220
221             # execute training
```

```
222             # The reshape commands in this section are because of
                    a subtlety within numpy— I was getting shapes of
                    (2,)
223             # indicating a vector— but attempting to transpose
                   it did me no good.  It's something to keep in mind
                    and
224             # to fix in future versions of this code.
225             for ct in range(0, num_training_points):
226                 sess.run(train_op, feed_dict={X:
                        concat_net_ins_outs[ct, 0:siv.shape[1]].
                        reshape((1, siv.shape[1])),
227                                               Y:
                                                   concat_net_ins_outs
                                                   [ct, siv.shape
                                                   [1]::].reshape
                                                   ((1, 1))})
228
229             # execute test
230             ave_test_cost = 0.
231             ave_miss_cost = 0.
232             for ct2 in range(num_training_points, len(siv)):
233                 ave_test_cost += sess.run(total_cost,
234                                           feed_dict={X:
                                              concat_net_ins_outs[ct2,
                                               0:siv.shape[1]].reshape
                                              ((1, siv.shape[1])),
235                                                      Y:
                                                          concat_net_ins_outs
```

```
                                                    [ct2, siv.
                                                    shape[1]::].
                                                    reshape((1,
                                                    1))})\
236                                                /
                                                    num_test_points

237                 ave_miss_cost += sess.run(miss_cost,
238                                           feed_dict={X:
                                                  concat_net_ins_outs[ct2,
                                                   0:siv.shape[1]].reshape
                                                  ((1, siv.shape[1])),
239                                                       Y:
                                                    concat_net_ins_outs
                                                    [ct2, siv.
                                                    shape[1]::].
                                                    reshape((1,
                                                    1))}) \
240                                                /
                                                    num_test_points

241
242             # Rather than choose an arbitrary large value to
                    initialize the
243             # best_miss_cost to, do it this way.  It avoids the
                    issue of a
244             # colossally bad network never having a worse miss
                    cost than
```

```
245             # whatever value it was initialized to.
246             if loop == 1:
247                 best_miss_cost = ave_miss_cost
248                 # Create a full copy rather than just pointing to
                        the existing
249                 # network with a pointer.
250                 lowest_miss_net = copy(logits)
251             elif ave_miss_cost < best_miss_cost:
252                 best_miss_cost = ave_miss_cost
253                 lowest_miss_net = copy(logits)
254
255             if loop % 10 == 0:
256                 print("Average total cost for epoch " + str(loop)
                        + " :\t" + str(ave_test_cost) \
257                           + "\t Average miss cost:\t " + str((
                              ave_miss_cost)))
258                 f.write("Average total cost for epoch " + str(
                        loop) + " :\t" + str(ave_test_cost) \
259                           + "\t Average miss cost:\t " + str(
                              ave_miss_cost) + "\n")
260
261         # if mode == 'exp':
262         for inp in range(0, len(siv)):
263             # Compute the network estimate at each point
264             o = sess.run(logits, feed_dict={X:siv[inp, 0:siv.
                    shape[1]].reshape((1, siv.shape[1]))})
265             junkstring = " ".join(str(siv[inp]).strip('[] ').
                    split())
```

```
266             outstring = junkstring + " " + str(o*output_bounds[1,
                    :]).strip('[] ') + \
267                         " " + str(o).strip('[] ')+" "+str(
                                output_bounds[1, :]).strip('[] ') + \
268                         " " + str(ov[inp]) + "\n"
269             f2.write(outstring)
```

Bibliography

1. D. M. Goebel and I. Katz, *Fundamentals of electric propulsion: ion and Hall thrusters*. John Wiley & Sons, 2008, vol. 1.

2. J. Tran, "Numerical Study of Current Driven Instabilities and Anomalous Electron Transport in Hall-effect Thrusters," Master's thesis, University of California, Los Angeles, 2017.

3. P. M. Bellan, *Fundamentals of plasma physics*. Cambridge University Press, 2008.

4. D. A. Cunningham, "Localized Plasma Measurement During Instability Modes In a Hall Thruster," *Master's thesis, Air Force Institute of Technology*, 2016.

5. N. A. Krall and A. W. Trivelpiece, *Principles of plasma physics*. McGraw-Hill, Inc, 1973.

6. E. Fossum and L. King, "Design and Construction of an Electron Trap for Studying Cross-Field Mobility in Hall Thrusters," in *43rd AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit*, 2007, p. 5207.

7. T. Lafleur, S. Baalrud, and P. Chabert, "Theory for the anomalous electron transport in Hall effect thrusters. I. Insights from particle-in-cell simulations," *Physics of Plasmas*, vol. 23, no. 5, p. 053502, 2016.

8. J. Koo, D. Bilyeu, and R. Martin, "Pseudospectral Model for Hybrid PIC Hall-effect Thruster Simulation," AIR FORCE RESEARCH LAB EDWARDS AFB CA ROCKET PROPULSION DIV, Tech. Rep., 2015.

9. K. Kwon, M. L. Walker, and D. N. Mavris, "Study on Anomalous Electron Diffusion in the Hall Effect Thruster," *International Journal of Aeronautical and Space Sciences*, vol. 15, no. 3, pp. 320–334, 2014.

10. M. Lampe, W. Manheimer, J. McBride, J. Orens, K. Papadopoulos, R. Shanny, and R. Sudan, "Theory and simulation of the beam cyclotron instability," *The Physics of Fluids*, vol. 15, no. 4, pp. 662–675, 1972.

11. C. Pozrikidis, *Numerical computation in science and engineering*. Oxford University Press, 1998.

12. K. Novak, *NUMERICAL METHODS FOR SCIENTIFIC COMPUTING*. LULU COM, 2017.

13. G. Cybenko, "Approximation by superposition of sigmoidal functions," *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, 1989.

14. K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, p. 251257, 1991.

15. S. Samarasinghe, *Neural networks for applied sciences and engineering: from fundamentals to complex pattern recognition*. Auerbach, 2007.

16. F. v. Veen, "The Neural Network Zoo," Nov 2017. [Online]. Available: http://www.asimovinstitute.org/neural-network-zoo/

17. A. Tchircoff, "The mostly complete chart of Neural Networks, explained," Aug 2017. [Online]. Available: https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

18. A. S. Walia, "Activation functions and it's types-Which is better?" May 2017. [Online]. Available: https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f

19. A. Karpathy and J. Johnson, "CS231n Convolutional Neural Networks for Visual Recognition Course Materials," 2017.

20. A. Wasilewska, "CSE634 Lecture Notes, Chapter 6," Jan 2018.

21. Z. Hao, "Weight Initialization Methods in Neural Networks," May 2017. [Online]. Available: https://isaacchanghau.github.io/post/weight_initialization/

22. M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

23. J. Duchi and Y. Singer, "Efficient online and batch learning using forward backward splitting," *Journal of Machine Learning Research*, vol. 10, no. Dec, pp. 2899–2934, 2009.

24. J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

25. G. Hinton, N. Srivastava, and K. Swersky, "Neural Networks for Machine Learning-Lecture 6a-Overview of mini-batch gradient descent," 2012.

26. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

27. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

28. M. Frigo and S. G. Johnson, "FFTW Home Page." [Online]. Available: http://www.fftw.org/

29. "Does the FFT function in MATLAB 6.5 (R13) use the FFTW library routines?" Jun 2009. [Online]. Available: https://www.mathworks.com/matlabcentral/answers/100596-does-the-fft-function-in-matlab-6-5-r13-use-the-fftw-library-routines

30. "cuFFT Documentation," Mar 2018. [Online]. Available: https://docs.nvidia.com/cuda/cufft/index.html

31. "Fast Fourier transform." [Online]. Available: https://rosettacode.org/wiki/Fast_Fourier_transform

32. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

33. "Installing TensorFlow — TensorFlow." [Online]. Available: https://www.tensorflow.org/install/

34. "TensorFlow Version Compatibility — TensorFlow." [Online]. Available: https://www.tensorflow.org/programmers_guide/version_compat

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
12–09–2018	Masters Thesis	Jan 2017 - Aug 2018

4. TITLE AND SUBTITLE

Application of Spectral Solution and Neural Network Techniques in Plasma Modeling for Electric Propulsion

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Whitman, Joseph R

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENY-MS-18-S-076

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Resarch Laboratory (AFMC)
Dr. Justin Koo
AFRL/RQR
5 Pollux Drive
Edwards AFB, CA 93524-7048

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution Statement A. Approved for public release; Distribution Unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

A solver for Poisson's equation was developed using the Radix-2 FFT method first invented by Carl Friedrich Gauss. Its performance was characterized using simulated data and identical boundary conditions to those found in a Hall Effect Thruster. The characterization showed errors below machine-zero with noise-free data, and above 20% noise-to-signal strength, the error increased linearly with the noise. This solver can be implemented into AFRL's plasma simulator, the Thermophysics Universal Research Framework (TURF) and used to quickly and accurately compute the electric field based on charge distributions. The validity of a machine learning approach and data-based complex system modeling approach was demonstrated. To this end, several multilayer perceptrons were created and validated against AFRL-provided Hall Thruster test data, with two networks showing mean error below 1% and standard deviations below 10%. These results, while not ready for implementation as a replacement for lookup tables, strongly suggest paths for future work and the development of networks that would be acceptable in such a role, saving both RAM space and time in plasma simulations.

15. SUBJECT TERMS

Hall Thruster, Electric Propulsion, Plasma Physics, Spectral Solver, Neural Networks

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Hartsfield, Carl R., AFIT/ENY
U	U	U	UU	116	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4667; carl.hartsfield@afit.edu