Heat Flux Model Validation Utilizing Convolutional Neural Networks and Sub-surface Thermocouples for NSTX-U

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Thomas Patrick Looby

December 2018

Abstract

A proof of concept convolutional neural network (CNN) has been developed to assist in operating tokamaks outside of existing empirical scalings for the heat flux width, λ_q [lambda-q]. NSTX-U has designed new plasma facing components (PFCs) to withstand increased halo current forces as well as elevated heat fluxes driven by increased poloidal field and neutral beam power compared to NSTX. Larger graphite tiles are castellated to 2.5 cm [centimenter] x 2.5 cm [centimeter] to reduce bending stresses. Maintaining PFCs below engineering limits will be an important consideration for operation of NSTX-U. Sub-surface thermocouples will be utilized to demonstrate validation of the heat load model, using the castellated designs to quantify the shot-integrated energy deposited in the NSTX-U divertor. A Convolutional Neural Network (CNN) has been trained using ANSYS simulations of PFC response to a variety of time-varying heat flux profiles. The CNN accepts time evolving thermocouple data and various 0-D engineering parameters and outputs heat flux model parameters, such as the poloidal field scaling of the heat flux width, λ_q [lambda-q]. The CNN enables high accuracy validation of the heat flux model despite a limited number of simulated NSTX-U shots, noise, and systematic errors in the thermocouple data. This application of machine learning to nuclear fusion diagnostics provides an alternative method to traditional analytical solution inversion, and may be ported over to other diagnostics in the future.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background Information

As researchers attempt to demonstrate net thermal power production from nuclear fusion, a myriad of engineering problems emerge. One such problem is the development of plasma facing components (PFC) that can withstand the incredible heat loads that exist inside a fusion reactor, and the simultaneous development of monitoring systems for these PFCs. The design, validation, and monitoring of PFCs will be necessary for successful operation of a fusion reactor within its engineering limits. One particular magnetic confinement design, the tokamak, has been adopted by researchers worldwide. Many tokamaks have been constructed across the globe, as researchers attempt to harness the energy from nuclear fusion reactions. The largest of these projects is ITER, an international scientific collaboration to construct the largest tokamak on earth. In a demonstration (DEMO) fusion reactor that will be capable of producing consumable power, the heat loads may reach 50 $\mathrm{MW\,m^{-2}}$, and teams of scientist and engineers are actively working to develop the technologies necessary to ensure reliable operation of PFCs [4, 5].

The National Spherical Tokamak eXperiment (NSTX), is a nuclear fusion experiment that has been operating since the late 1990s at low aspect ratio, which is characteristic of spherical tokamaks. The NSTX Upgrade (NSTX-U) Project has developed new heat flux requirements for PFCs, which are necessary to accomodate a narrower scrape off layer, increased heating power, larger halo currents, and increased pulse duration. A plasma facing tile redesign took place, resulting in a castellated graphite concept that greatly minimizes thermal stress, while managing electromagnetically induced stresses. A critical component to the operation of NSTX-U at maximum performance will be maintaining these graphite tiles within their allowable engineering tolerances. As a means of quantifying the evolution of these Plasma Facing Components (PFCs) relative to their respective engineering limits, an investigation into PFC monitoring systems was necessary. An explicit goal from the NSTX-U PFC Working Group Memo 014 (Goal 3 from Milestone R18-1/3) was to demonstrate a pathway for heat flux model validation utilizing sub-surface temperature measurements [6].

The extreme environment inside a tokamak severely constrains the diagnostic options. To measure the heat flux to PFCs in NSTX-U, thermocouples will be embedded below the tile surface in the the divertor. Due to the fact that the new castellated design arrests transverse thermal diffusion across multiple castellations, it is now possible to approximate the shot-integrated deposited energy within each castellation via a thermocouple. That being said, there is limited temperature information available. During the shot, electromagnetic noise will likely result in erroneous thermocouple signals due to induced voltages, so it is necessary to omit thermocouple data recorded while nearby electromagnets are energized. Additionally, within each castellation three dimensional thermal diffusion will result in a spatially integrated energy deposition. Therefore, post-shot thermocouple measurements in which temperatures are representative of the castellation's entire spatial and temporal domain can be used to validate (or invalidate) plasma heat flux models and associated engineering. As the diagnostic operational domain is further constrained by physical processes, heat flux model validation begins to take the shape of an engineering problem with well defined boundaries.

1.2 Project Objectives

In response to the aforementioned NSTX-U milestone [6], an investigation into methods for heat flux validation commenced. T. Looby was tasked with evaluating if sub-surface thermocouples could be utilized to derive heat flux scaling parameters with a limited number of plasma discharges [7]. Looby was also tasked with engineering the systems necessary to provide a proof of concept demonstration of deriving heat flux scaling parameters from subsurface thermocouple data. It is this proof of concept that is the subject of the presented masters thesis. Eventually, this system may be integrated into operational systems at NSTX-U, in a pre-shot, inter-shot, or post-shot, capacity.

As a means of simplifying the aforementioned task, three project objectives were identified to serve as waypoints and guide the investigation. The objectives are as follows:

1. *Simulate the response of NSTX-U graphite PFCs to spatially and time varying heat fluxes.*

2. *Demonstrate how unknown heat flux model parameters can be derived with various sampling mechanisms within a given parameter space.*

3. *Demonstrate objective 2, but now add demonstrated uncertainties to measurement and model support parameters.*

These objectives provided modularity, which enabled the larger heat flux validation objective to be broken down into smaller isolated tasks. This systems engineering approach also enables any of the individual technologies developed to be utilized independently of the greater system. Together the modules serve to answer the primary research question: can subsurface thermocouple measurements alone provide sufficient information to validate the assumed heat flux profile incident upon NSTX-U divertor tiles?

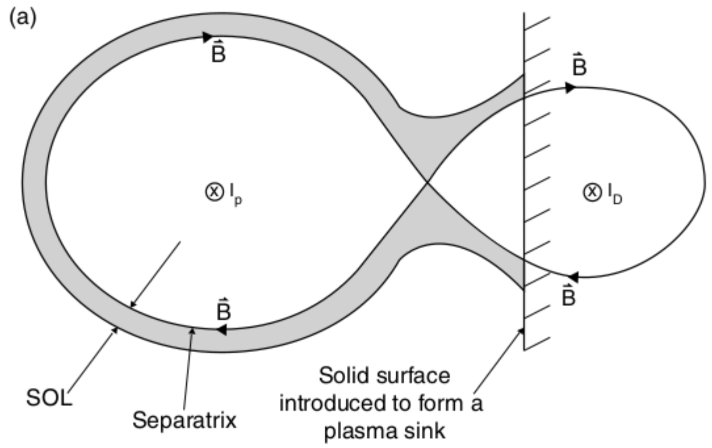
Chapter 2

Background Physics and Engineering

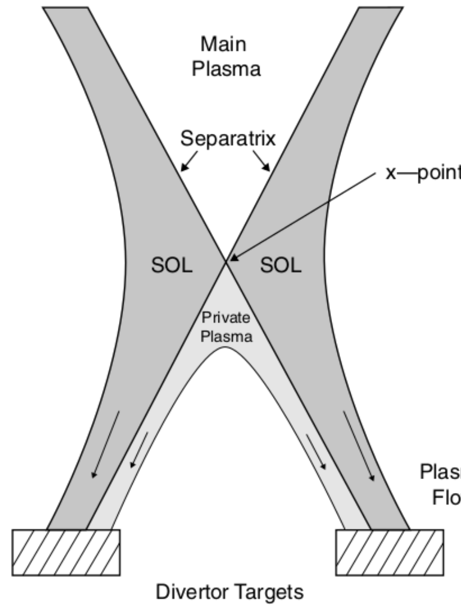
2.1 Eich Heat Flux Model

Many techniques have been investigated as a means of enhancing energy confinement in magnetically confined plasmas. Most modern magnetic confinement concepts, such as tokamaks and stellerators, attempt to leverage some of the inherent magenetohydrodynamic (MHD) phenonomena to increase net energy gain. High confinement H-mode has been demonstrated in many fusion experiments of the past several decades [8]. H-mode plasmas exhibit increased energy confinement times, increased particle confinement, and reduced turbulence, when compared to other confinement regimes (such as low-confinement mode) [9]. The world record for fusion energy production was produced with a Deuterium-Tritium (DT) fueled tokamak operating in H-mode at the Joint European Torus (JET) in the UK [10]. This record shot produced 16.1 MW from 24 MW of input power, corresponding to a net fusion energy gain, Q, of 0.67. Because of the successes observed when operating tokamaks in H-mode, many future machines plan to employ H-mode as the operational standard.

A characteristic of these H-mode plasmas is the development of an edge transport barrier, which appears in the form of a large pressure gradient just inside the last closed flux surface (LCFS) [11]. While this transport barrier is excellent for enhancing fusion energy gain, a byproduct is an occasional release of energy across the gradient, termed an Edge Localized Mode (ELM). In large fusion devices these ELMs can contain sufficient energy to damage the machine if not properly managed. Outside of the LCFS is the scrape off layer (SOL),



(a) Plasma current, I_p, induces magentic field while external divertor coil current, I_D controls diversion shape.



(b) Toroidal cross section of divertor region

Figure 2.1: Toroidal Cross Sections from [1]

a thin layer of plasma where magnetic field lines are not closed, but rather intersect the wall (see figure 2.1). In most modern tokamaks, these open field lines (ideally) terminate at a surface engineered to tolerate high heat loads, in a region called the divertor, which serves as a sort of exhaust system for the entire tokamak. This surface is often referred to as a tile, and is typically carbon or tungsten. The SOL - LCFS boundary is called the seperatrix. The separatrix extends around the entire plasma, and intersects the divertor surface at a location called the strike point. The region inboard of the strike point is the private plasma region, and the region outboard of the strike point is the common flux region. Because the divertor tiles must tolerate large heat loads, it is of critical importance that the nature of these heat loads be understood during the tile design. Additionally, knowledge of where the device is operating relative to the limitations of the tile is necessary to ensure safe engineering operation. The heat flux applied to the tile constrains the allowable domain of tile operation.

Heat flux entering the SOL has a characteristic decay length, λ_q, termed the power decay length or SOL heat flux width. This power decay length is crucial in understanding the peak heat load that will be applied to the tile, which impacts the operational constraints of the tokamak. A multi-machine λ_q parametric scaling was experimentally determined in 2011 using data from JET and the Axially Symmetric Divertor Experiment (ASDEX) [12]. At the divertor entrance, λ_q exhibits an exponential decay in the radial direction (away from LCFS), which can be derived via application of (1-dimensional) Fick's law to determine particle cross field diffusion [1]. This decay can be described by the equation,

$$q(\bar{s}) = q_0 \text{exp}\left(\frac{-\bar{s}}{\lambda_q f_x}\right) \tag{2.1}$$

where q_0 is the heat flux crossing the separatrix, $\bar{s}$ is the radial distance from the confined plasma ($\bar{s} = s - s_0$), and f_x is the magnetic flux expansion (from the seperatrix to a point 5 mm beyond it in the radial direction). As heat travels from the divertor entrance to the tile, thermal diffusion occurs, which results in heat flux leakage into the private flux region. The final heat flux equation is derived by the convolution of the exponential power decay with a thermal diffusion gaussian,

$$q(\bar{s}) = \frac{q_0}{2} \text{exp}\left[\left(\frac{S}{2\lambda_q f_x}\right)^2 - \frac{\bar{s}}{\lambda_q f_x}\right] \text{erfc}\left(\frac{S}{2\lambda_q f_x} - \frac{\bar{s}}{S}\right) + q_{BG} \tag{2.2}$$

where S represents the width of the gaussian and q_{BG} represents the background heat flux [12, 13]. Examples of fitting this heat flux profile to experimental data from various machines are provided in figure 2.2 from from [2]. For the remainder of this work, equation 2.2 will be referred to as the Eich heat flux profile.

Employing multivariate empirical regression, a parametric scaling for λ_q was derived by utilizing equation 2.2 and data from six devices,

$$\lambda_q[mm] = C_0 B_T^{C_B} q_{cyl}^{C_q} P_{SOL}^{C_p} R_{geo}^{C_R} \tag{2.3}$$

where B_T is the toroidal field measured in T, q_{cyl} is the cylindrical safety factor, P_{SOL} is the power crossing into the SOL measured in MW, and R_{geo} is the tokamak major radius measured in m [12]. This formula indicates that λ_q is governed by five scaling parameters, C_0, C_B, C_q, C_p, and C_R. Objective 2 of this thesis is to resolve these scaling parameters via subsurface thermocouples. In other words, can subsurface thermocouple measurements alone

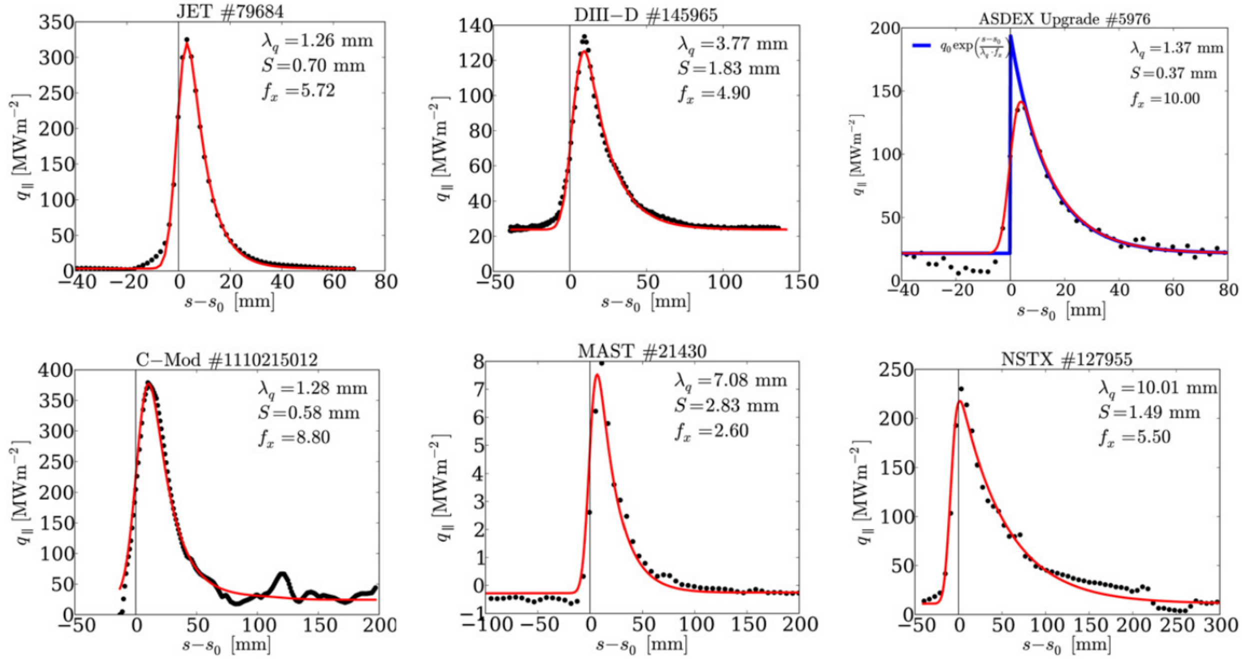


Figure 2.2: Example Eich heat flux profiles fit to experimental data [2].

provide sufficient information to determine these scaling parameters, thereby resolving λ_q and the corresponding Eich heat flux profile? A discussion of previous methods for validation of equation 2.3 is provided in chapter 4. In this case, machine learning was used to determine these scaling parameters. More specifically, a convolutional neural network was constructed that can determine the scaling parameters after receiving subsurface thermocouple data and various machine parameters as inputs.

2.2 Neural Networks

Machine Learning and Artificial Intelligence are gaining widespread popularity across a myriad of industries outside of computer science. These algorithms, when paired with modern microprocessing power, have yielded exceptional results with regard to pattern recognition, object detection, and regression. Medical imaging systems, drug discovery, autonomous vehicles, financial market forecasting, search results, and text prediction are but a few of the thousands of examples of engineers using these systems to solve real world pattern recognition problems [14, 15, 16]. Additionally, machine learning algorithms are aptly suited to handle massive amounts of data that would seem daunting to a human.

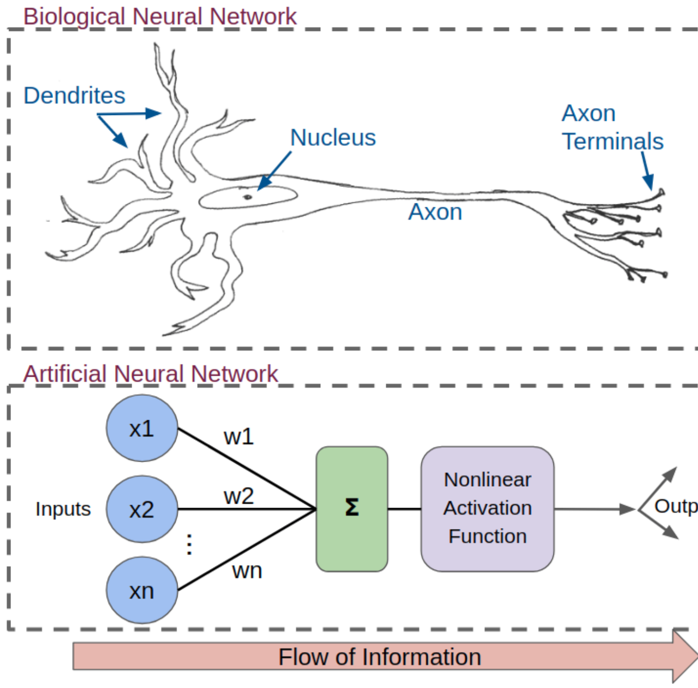


Figure 2.3: Comparison of biological and artificial neural networks

Combining pattern recognition capabilities with data mining algorithms has enabled these machines to surpass human capabilities in many domains where patterns must be derived from large datasets.

In some machine learning algorithms, engineers hand pick specific characteristics that they intend the network to learn, and set about explicitly coding these characteristics into a machine learning algorithm. When learning simple systems this method may suffice, but with more complicated systems the engineer can overlook the intricate details that are necessary to characterize the system. To circumvent human bias, many modern machine learning techniques employ *representation learning*, in which the engineer designs the algorithm architecture, but does not explicity specify what the algorithm should learn. The engineer then feeds large datasets to the algorithm, gives the algorithm an objective (such as error minimization), and the algorithm adjusts internal parameters to find the optimum configuration to maximize the objective. Because the engineer supervises the data input to the algorithm, and defines the ultimate objective, this class of algorithm is considered to be *supervised learning*.

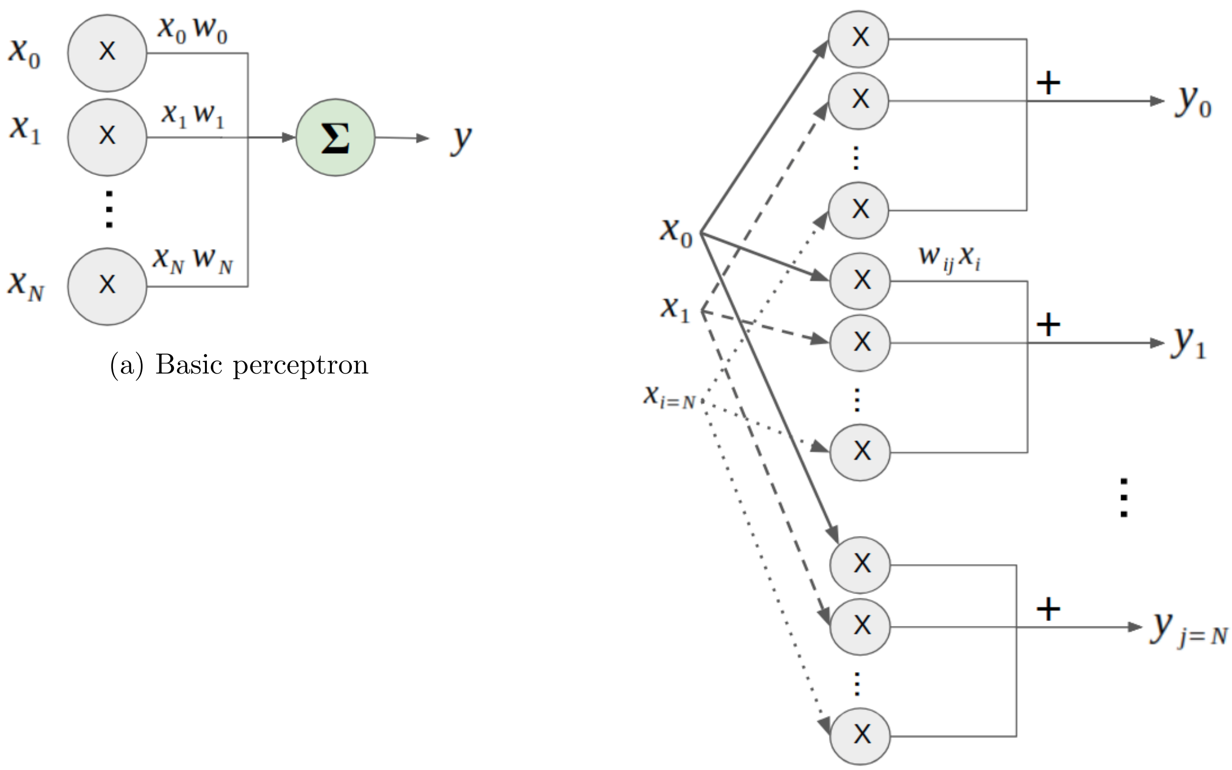
In artificial neural networks (ANN), sometimes also called *multilayer perceptrons*, the process is analogous to the biological counterpart. Figure 2.3 provides an illustration of a biological and an artificial neural network. In both cases, information flows into the inputs, then through a summation node and a nonlinear modulator, and finally out to the next neuron [17]. While the basic building block for a biological neural network is the neuron, the basic building block of an artificial neural network is the *perceptron* [3]. Figure 2.4 illustrates a simple perceptron. Inputs, x_i, arrive from the previous perceptron or from the environment, and are multiplied by a weight that corresponds to each input, w_i. The output from this perceptron, y, is the linear combination of these N inputs multiplied by their corresponding weights where $x_i, w_i \in \mathbb{R}$,

$$y = \sum_{i=1}^{N} w_i x_i + w_0, \quad \text{where } w_0 \text{ is a bias term [3].} \tag{2.4}$$

It may be apparent that if there is only a single input, then the perceptron produces the equation for a line. Ergo, the most basic perceptrons can be utilized for linear regression by iteratively feeding inputs to the perceptron, adjusting the weights, and maximizing some objective function of y. When the number of inputs is greater than one, then the perceptron defines a hyperplane, and can achieve multivariate linear regression [3]. The corresponding multivariate matrix representation is

$$y = \mathbf{w}^T \mathbf{x}, \quad \text{where } \mathbf{x} = [1, x_1, ..., x_N]^T \text{ and } \mathbf{w} = [w_0, w_1, ..., w_N]^T. \tag{2.5}$$

If multiple outputs must be calculated from the same set of inputs, then a parallel architecture is necessary, as shown in figure 2.4. This parallelization enables multiple perceptrons to be connected in a network called an Artificial Neural Network (ANN), where the number of output variables corresponds to the number of parallel perceptrons in the network. While these parallel systems are versatile, they are still only capable of learning the weights associated with linear systems. To learn a non-linear system it is necessary to introduce a non-linearity to the network. This non-linearity is called an *activation function* and comes in many forms, the most historical being the sigmoid function. After the summation of weighted inputs is computed, this value is passed through the activation



(a) Basic perceptron

(b) Perceptrons with multiple outputs

Figure 2.4: Perceptron architectures inspired by [3]

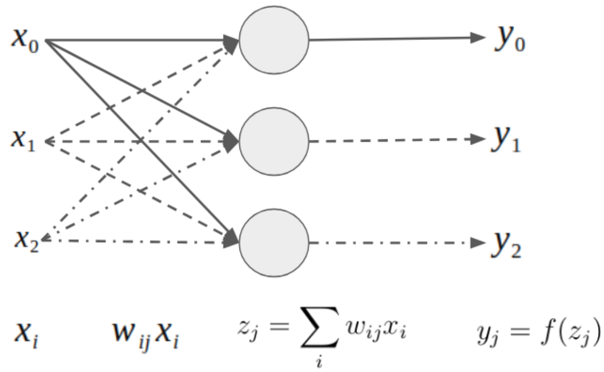


Figure 2.5: Neural network with forward propagation mathematics

function to generate the final output. This process can be described by the following equations, which map inputs x_i to outputs y_j, through the activation function, $f(x)$.

$$\mathbf{z} = \mathbf{w}^T\mathbf{x} \tag{2.6}$$

$$z_j = \sum_i w_{ij}x_i \tag{2.7}$$

$$y_j = f(z_j) \tag{2.8}$$

If the sigmoid activation function is used, then the output becomes,

$$y_j(x_i) = f(z_j) = \frac{1}{1 + exp[-z_j]} = \frac{1}{1 + exp[-\sum_i w_{ij}x_i]} \tag{2.9}$$

While the sigmoid activation function has historical significance, most modern neural networks employ half wave rectification as the activation function of choice. Perceptrons that employ half wave rectifier activation functions are called *Rectified Linear Units* (ReLU). Outputs to these perceptrons are given by the equation,

$$y_j = f(z_j) = \begin{cases} 0 & \text{if } z_j \leq 0 \\ z_j & \text{if } z_j > 0 \end{cases}$$

Figure 2.5 illustrates the forward propogation (from input to output) of a three perceptron parallel network with these mathematical formulations overlayed.

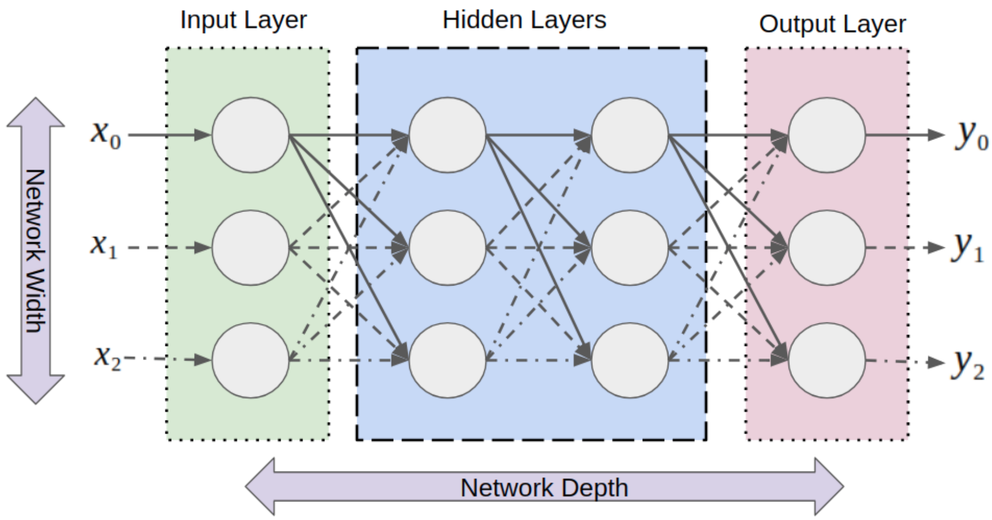


Figure 2.6: Deep neural network with two hidden layers

An engineer must determine the number of parallel perceptrons, or the *width*, of the network in addition to number of serial perceptrons, or *layers*, that constitute the network *depth*. Many of the recent successes utilizing neural networks can be attributed to wide (many parallel perceptrons) and deep (many serial perceptron) algorithms. When there are multiple layers between the input and output neurons, the middle layers are said to be *hidden* and the system is considered a deep neural network capable of *Deep Learning*. Figure 2.6 illustrates a deep neural network that can learn multivariable functions, although it is not very deep. In modern deep neural networks, there can be millions of weights in the weighting matrix.

Once a neural network has been constructed, it must be trained. The training process consists of adjusting the weights such that the neural network represents the system that it is intended to model or learn. This is achieved by first defining an error function, E, which can be described by the simple equation,

$$E_j = (y_j(x) - y'_j(x))^2 \tag{2.10}$$

where $y(x)$ represents the value(s) that the neural net generated, and $y'(x)$ represents the target value that the neural net should have generated. It is also possible to use any other error metric, such as Mahalanobis distance or absolute difference. Once the error has been calculated, the weights must be adjusted to minimize the error. For each weight in the weight matrix, a gradient vector is computed, which indicates the change in error that would occur if that weight was changed by a small amount. This gradient vector represents changes in error per changes in each respective weight and can mathematically be represented by cascade partial derivatives given in equation 2.11.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} \tag{2.11}$$

After the input has been fed into the network and an output y_j has been generated, the error is calculated. Starting at the output layer, a partial derivative of the error (E) is taken with respect to the output (y). Next, a partial derivative of the output (y) is taken with respect to the output of the activation function, (z). Finally the partial derivative of the activation function output is taken with respect to the weights (w) in the final layer. Via the chain rule, the derivative of error with respect to the weight in this layer can be calculated by the cascade (multiplicative) combination of these results (again, equation 2.11). This process is continued at each of the previous layers until the gradient vectors are *backpropagated* through the entire network. Figure 2.7 illustrates backpropagation, where superscripts correspond to layers of the network.

Once the gradient vectors have been backpropagated through the network, the weights are updated via a *learning rate*, $\eta > 0$. The resulting modification that must be applied to each weight is calculated to be

$$\Delta w_{ij} = -\eta\frac{\partial E}{\partial w_{ij}} \tag{2.12}$$

and the updated value for w_{ij} is calculated to be

$$w_{ij}^{[k]} = w_{ij}^{[k-1]} + \Delta w_{ij} \tag{2.13}$$

where the superscript indicates the (forward then back propagation) iteration number. This entire process is often called *gradient descent*, because it is synonymous to descending a topological surface (the function to be modeled) as the network seeks a minimum (minimum error).

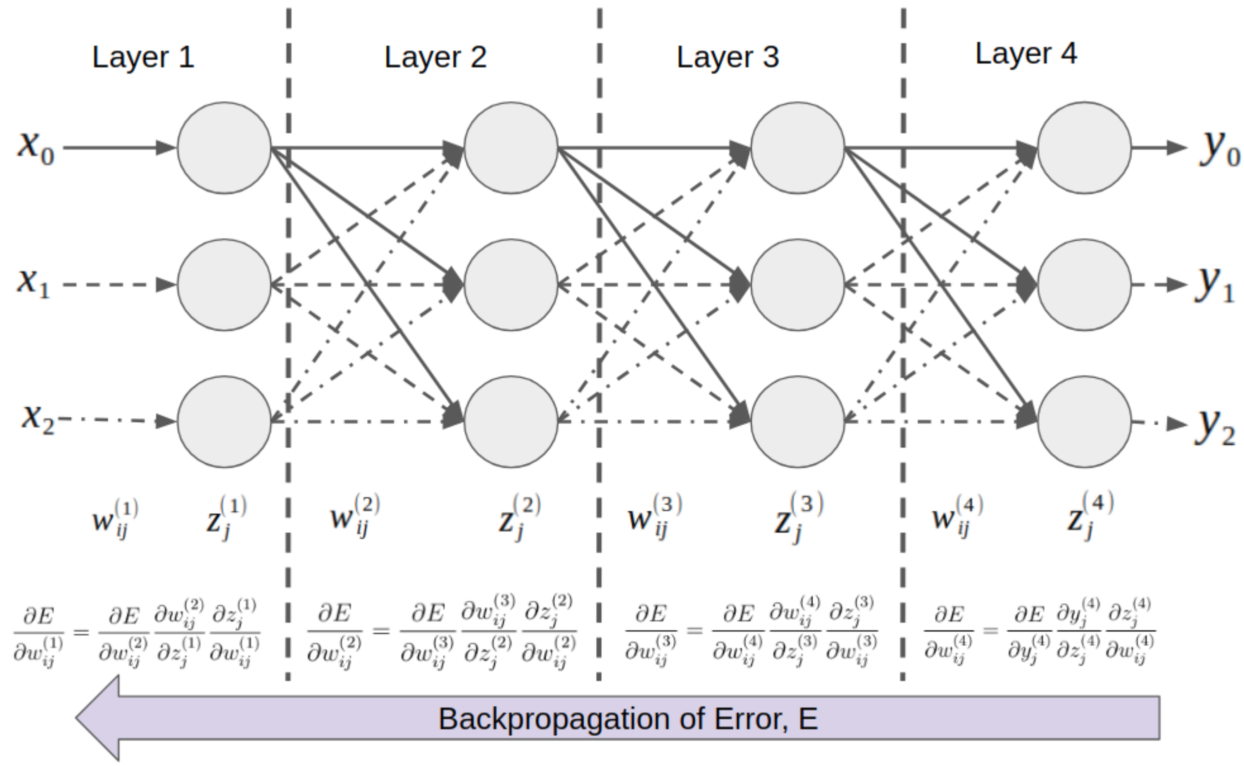


Figure 2.7: Neural network with backpropagation mathematics

2.3 Convolutional Neural Networks

While a straighforward neural network is excellent at multivariable non-linear regression, engineers and computer scientists have implemented a myriad of variations on this basic deep neural network architecture. One particular class of neural network that is of particular interest to this investigation is the convolutional neural network (CNN). Convolutional neural networks are excellent at pattern recognition and object detection. They are utilized in many image recognition tasks such as facial recognition of pictures or computer vision systems for self driving cars. The power of these networks lies in their ability to translate abstract features of the image in question into concrete vectors than can be manipulated by the neural net. In this manner, CNNs are feature extractors, or pattern recognition tools. After feature extraction is completed by the CNN, the resulting vector is used as the input to a deep neural network, which can then extract the nonlinear model and formulate predictions related to the content of the input data.

The CNN is a proven tool for extracting features from multidimensional arrays [18]. To describe the architecture of a CNN, and to clarify why this architecture may be advantageous for analyzing thermocouple data on NSTX-U, an example is helpful. Consider an image which consists of pixels, organized in 2D (x,y). Each pixel has three channels: red, green, blue, which each have an 8 bit value (0-256). The CNN is tasked with identifying specific human faces in each image. In other words, the images are inputs to the CNN, and the desired output is the name of the person who the CNN has identified in the image. A typical CNN consist of several layers that each perform a specific function. First, the image is fed to a convolution layer. In the convolution layer, parallel perceptrons are each responsible for identifying specific motifs within the image that characterize human faces. For humans, the abstraction of facial pattern recognition is so far removed from conscious thought that it would be extremely difficult to code this abstraction. Indeed, the human pattern recognition algorithm is contained in the interconnection of billions of neurons in the human brain [17]. Instead, the motifs are learned by the CNN with no explicit guidance from the engineer, except for the objective function, such as error minimization.

During the neural network training process, various motifs are explored by the neural network in the form of weights and backpropagation weight adjustments within the convolution layer perceptrons [18]. These motifs are called *features*, and the perceptrons that learn these features combine with perceptrons in the next layer to form *feature maps*. In each feedforward pass (input image is fed into the network), the convolution layer searches the input image for any of the features it has stored in its feature maps. It performs this scan with a digital signal processing (DSP) discrete cross correlation (technically not convolution unless signal/kernel is symmetric) which is defined by,

$$(f \star g)[n] = \sum_{-\infty}^{\infty} f^*[m]g[m+n] \tag{2.14}$$

where f^* is defined as the complex conjugate, $f[]$ and $g[]$ are the signals being cross correlated, and m, n represent discrete timesteps. Cross correlation is a technique utilized to determine the similarity between two signals, and in CNNs it is utilized to determine if the input image contains the feature associated with that feature map.

Once the convolution layer has cross-correlated the input image, or determined how similar the input image is to the feature it has learned to identify, the results are fed to the next layer in the CNN, a *pooling* layer. The pooling layer consolidates, or pools, local features in order to eliminate redundant positive feature identifications. Slight rotations in the image, locating a feature at different (x,y) locations within the image, or systematic noise, can manifest into multiple positive feature detections for the same feature. To reduce these redundancies the pooling layer scans the convolution layer output and combines local positive identifications into a single identification [18]. Usually, multiple convolution + pooling layer combinations are placed in cascade, as a means of extracting every abstract feature. The output from the final pooling layer is the input to a *fully connected layer* (FC), which is another name for a typical layer in the basic deep neural network previously discussed. If the layer is considered fully connected, then it is implied that every perceptron is connected to every perceptron in the layers in front and behind it (see figure 2.6). Figure 2.8 illustrates a CNN used for pattern recognition in an image: two convolution layers are fed into a four layer deep neural network. It should be noted that in this figure (2.8) and in the remainder of this work, layers of perceptrons are indicated by long rectangular boxes rather than individual perceptrons (as in figure 2.6). This additional level of abstraction makes it more conventient to illustrate complicated neural network schemes, but the underlying mechanism is the perceptron.

In order for a neural network to be successful in generating predictions, it must train. For training, the engineer must compile a dataset consisting of examples that the neural net should recognize. Each entry in the dataset must be labeled with the target prediction that the neural network should learn. In other words, each data entry should consist of the data to process as well as the correct output value (or class). Continuing with the facial recognition example from above, the engineer would compile images that are each labeled with the appropriate name(s) of the person(s) in the image. In supervised learning, preparing the dataset is indeed the most complicated challenge. The input data should not be biased, it should sample the entire domain of the variables to be predicted, and it should be labeled correctly. Once the data has been prepared, it is randomly divided into three sets: the training set, the test set (also sometimes called validation set), and the publication set [3].

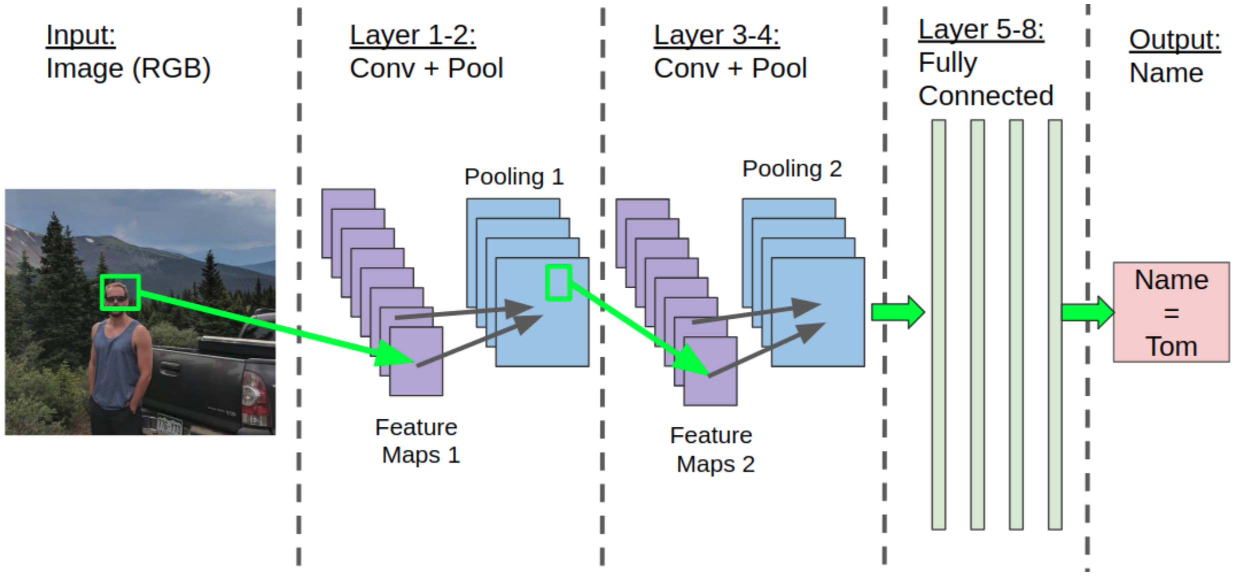


Figure 2.8: CNN with 2 conv + pool layers and 4 FC layers. 8 feature maps per conv layer and pooling layers downsampled by 2

The publication set is reserved for use after all training is complete, and will serve as a final metric of the neural network.

Once the data has been assembled, labeled, and broken into sets, the training process commences. The engineer feeds a dataset entry into the neural net, and the input is forward propagated to the output resulting in a prediction. The prediction is compared against the label (or target), and an error metric such as equation 2.10 is utilized to quantify the error. This error is then backpropagated from the output back to the input, in the form of cascade partial derivatives representing the error with respect to each inter-perceptron weight, as described in equation 2.11 and figure 2.7. The weights are then updated by multiplying this partial derivative with a learning rate, and adding it to the current weight, as described in equations 2.12 and 2.13. The process of forward propagation, backpropagation, then updating the weights is called an *iteration*.

Here, an analogy can be beneficial to intuitively describe the function of the learning rate. If the training (gradient descent) process is perceived as analogous to descending a mountain on foot, then the learning rate represents the size of steps taken as the mountaineer descends the mountain. Smaller learning rates enable greater resolution of the topological surface, at the cost of longer descent time. With very small steps the mountaineer can descend

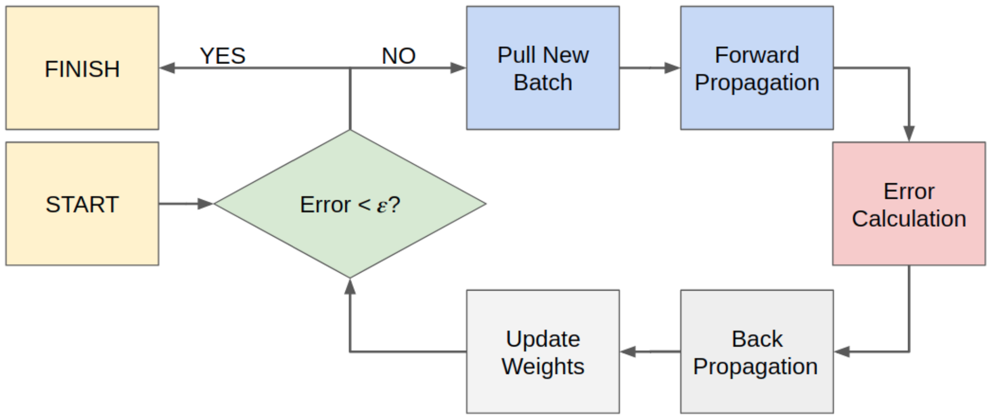


Figure 2.9: Flow chart of a simplified neural network training algorithm

into smaller saddles, squeeze into narrow chimney features, and follow the contours of the mountain more closely. Large learning rates may enable a speedy descent, but may not be capable of descending deep enough into the minimum that corresponds to maximizing the objective function. The mountaineer cannot squeeze into narrow chimney features with large strides. If the only way down into the valley below is through a narrow chimney feature, then the mountaineer will become stuck. However, if there are no local constrictions, the mountaineer taking large steps may descend to the valley below with great alacrity. The engineer must choose a learning rate that learns the system quickly, but does not get stuck in local minimums. Once this learning rate is multiplied by the negative partial derivative with respect to each weight, backpropagation for that dataset entry is complete. This process is repeated for every entry in the training dataset. When all entries in the training dataset have been forward propagated and backpropagated through the network, an *epoch* has completed. Epochs of training are completed iteratively until the inter-perceptron weights have been adjusted such that an accuracy threshold, ε, is reached.

It is also possible to input dataset entries to the network in *batches*. Instead of feeding dataset entries to the network one at a time, it may be desireable to feed multiple dataset entries into the network simultaneously. The *batch size* defines the number of dataset entries input into the network per iteration. For each dataset entry in the batch, the neural network forward propagates then backpropagates, resulting in a gradient vector for each weight, but does not update the weights via the learning rate yet. Rather, for each weight the neural network averages the gradient vectors for that weight, and then uses the average to compute the update weight. Using batches enables the neural network to learn multiple dataset entries in a single iteration, but obviously comes at the cost of longer CPU time. The engineer must select a batch size that reduces noise via the averaging mechanism, but still completes epochs in a reasonable time. The training process is summarized in a flow chart in figure 2.9.

After a certain number of epochs, it is common for the engineer to take a measurement of the neural network's accuracy. To do this, the validation set is used. A single dataset entry from the validation set is fed into the neural network, and the error is calculated. The utilization of the validation set for accuracy checking is termed cross validation [3]. Using the information provided by this measurement, the engineer can determine the performance of the network. Printing the current network performance enables the engineer to monitor the network as it is trained, and this information may also be utilized for statistics or plotting the neural network's accuracy as a function of epoch number.

The engineer defines the accuracy threshold required for any neural network based upon the context of the system the neural network is designed to model. For most regression problems, accuracy is defined as an allowable tolerance window above or below the value to be predicted. After the system has been trained to sufficient accuracy on the training and validation datasets, the model can be saved and converted to a system that does not train, but only makes predictions. The engineer then tests the neural network on the publication dataset, to determine how well the system performs on data that it has never seen. If the system performs well on the publication set, then it can be concluded that the training dataset sufficiently represented the entire model domain and the neural network has learned the correct model. If the system performs poorly on the publication dataset, then the engineer must either increase the training time, or increase the database size. Once a neural network performs well on the publication dataset, it may be applied in a production or an industrial environment.

Chapter 3

Simulating NSTX-U Graphite Tile Response to Heat Flux

The first project objective (chapter 1) is stated as: *Simulate the response of NSTX-U graphite PFCs to spatially and time varying heat fluxes.* Before a method for heat flux model validation could be developed, material-dependent limitations needed to be determined and verified. As a means of determining engineering limits beyond which tile damage may occur, simulations with machine-scale heat fluxes were completed. This engineering analysis of the thermal response of graphite tiles to applied heat fluxes manifested into the allowable operational domain of a heat flux model validation system. NSTX-U has also performed physical testing of these materials at electron beam facilities as a means of experimentally validating the material limitations [19]. Although these physical testing results are outside the scope of this thesis, they generally confirm the findings here. After basic material limitations were established, a system for generating temporally and spatially varying heat flux profiles was created using the Eich heat flux model [11]. These heat flux profiles became the inputs to an ANSYS multiphysics simulation which produced temperature and stress results. These results could then be passed to a system that would extract necessary parameters for heat flux model validation, objective 3, discussed in chapter 4.

3.1 Hardware

The NSTX-U tile redesign employs a castellated structure to reduce internal thermal stresses. Additionally, the castellated design arrests thermal diffusion across large distances within the tile. An image of the InBoard Horizontal Divertor (IBHD) tile is given in figures 3.1, 3.2. The castellations are approximately 3 cm by 3 cm, and each 'cube' is separated by a 1 mm gap. The entire tile spans approximately 15 cm (approximately 7°) in the toroidal direction and 15 cm in the radial direction. It should be mentioned that the tile revision utilized for this investigation is from January 2018, and does not reflect the current engineering revision for the IBDH. That being said, the January 2018 revision is sufficient for the purposes of subsurface temperature simulation. The final NSTX-U tile design can be found here: https://sites.google.com/pppl.gov/20180926pfcs-pempfdr/home.

While there are a myriad of high performance graphites available on the market today, the tile simulations performed for this study utilize Sigrafine R6510 graphite (SGL6510), which will likely be the final selection for the NSTX-U tiles [20]. Sigrafine R6510 is an isostatically pressed fine grain graphite, with grain sizes of approximately 10 µm in diameter. SGL6510 features an ultimate compresive strength (UCS) of approximately 130 MPa, an ultimate flexural strength (UFS) of approximately 60 MPa, and a thermal conductivity of 105 $\mathrm{W\,m^{-1}\,K^{-1}}$ at 20°C.

Thermocouples are inserted into 1 mm diameter holes bored into the bottom of the tiles, which extend to a distance of approximately 1 cm from the tile surface. When installed in this manner, the thermocouples are insulated from the main scrape off layer (SOL) plasma, yet close enough to the tile surface to have a large (ΔT) response peak to changes in heat flux to the tile surface. The thermocouples used for this study are manufactured by Omega Engineering (S/N CASS-116G-12-NHX), and consist of a type K (nickel-chromium) junction insulated by Magnesium Oxide and protected by a metal sheath [21]. These thermocouples are capable of withstanding temperatures up to 1370°C and have a voltage range of approximately 50 mV across the allowable temperature domain.

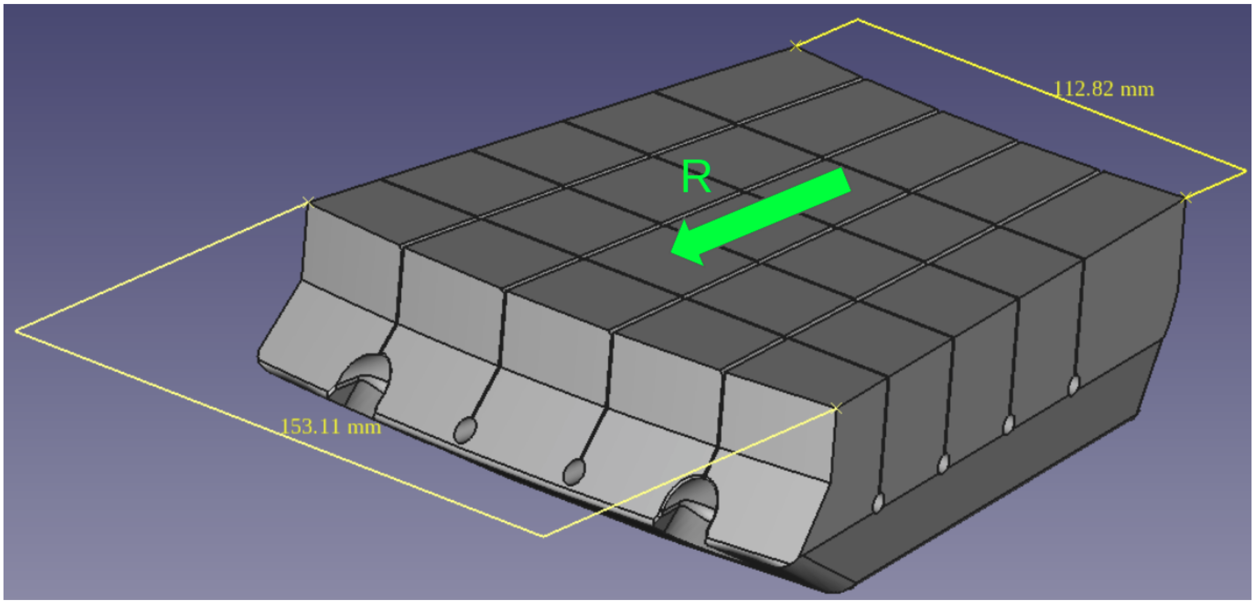


Figure 3.1: Castellated Inboard Horizontal Divertor Tile w/ dimensions: January 2018 version

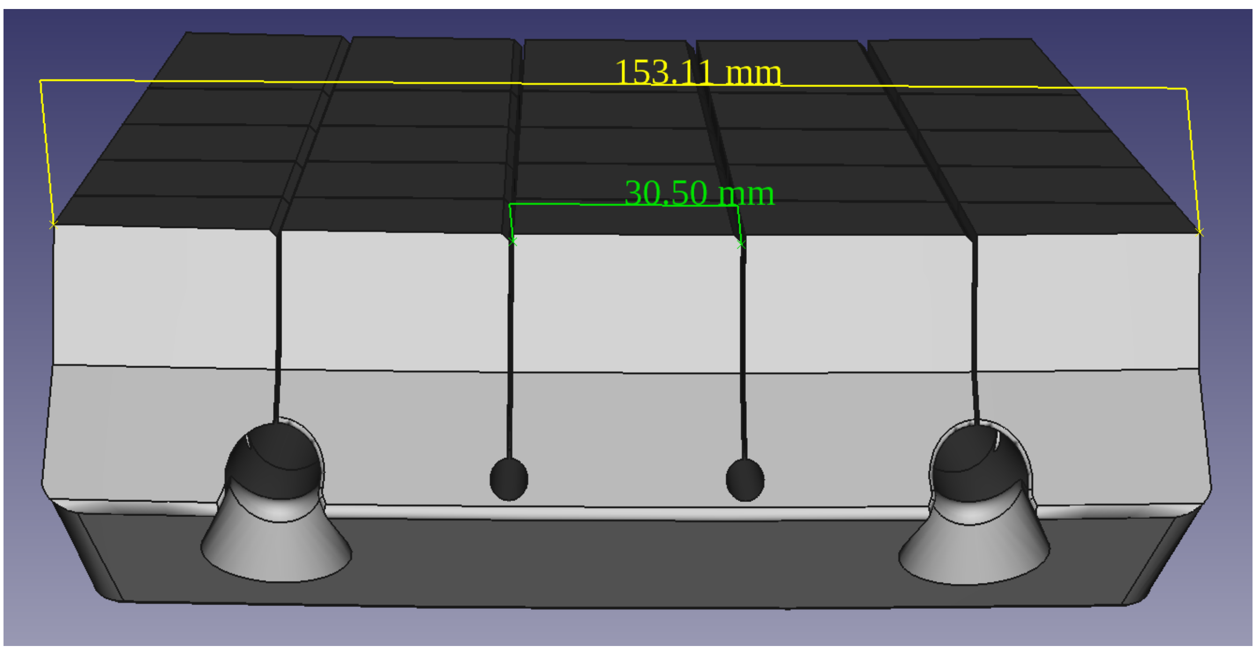


Figure 3.2: Castellated Inboard Horizontal Divertor Tile w/ dimensions: January 2018 version

3.2 Tile Assumptions

Because this proof of concept is largely simulation based, it was decided that simplification of the model would be beneficial to increase computation speed. Starting with a reduced model enables rapid prototyping with regards to software and solver architecture, makes the simulation possible on a workstation computer instead of a cluster of CPUs, and avoids the high performance computing licensure needed for some commercial multiphysics packages. The original CAD model (figures 3.1, 3.2), while not extremely large, was simplified with the help of a few reasonable geometric assumptions. A future goal of this project will be to apply the system to the original complete model.

The following assumptions were incorporated into the CAD model:

- Fish-scaling and chamfering of tile surfaces can be neglected
- Toroidal symmetry exists for the Scrape Off Layer heat flux profile (toroidal dimension can be ignored and a single cross section can be studied)
- The heat flux applied to the divertor target can be adequately described by a single coordinate, r.

These assumptions resulted in a CAD model that was roughly 20% of the original model size, and defined by a single coordinate. Figure 3.3 illustrates the original CAD model with the 'cut planes' utilized to simplify the model and then shows the reduced model after these assumptions were incorporated into the CAD model. Additionally, some simulations were performed on single castellations.

3.3 Constant Heat Flux Results

A quintessential tool for the analysis of thermal loading and mechanical stresses is a multiphysics suite that includes a finite element solver. For this research, ANSYS multiphysics suite (version 19) was the solver of choice, as it is already being utilized by

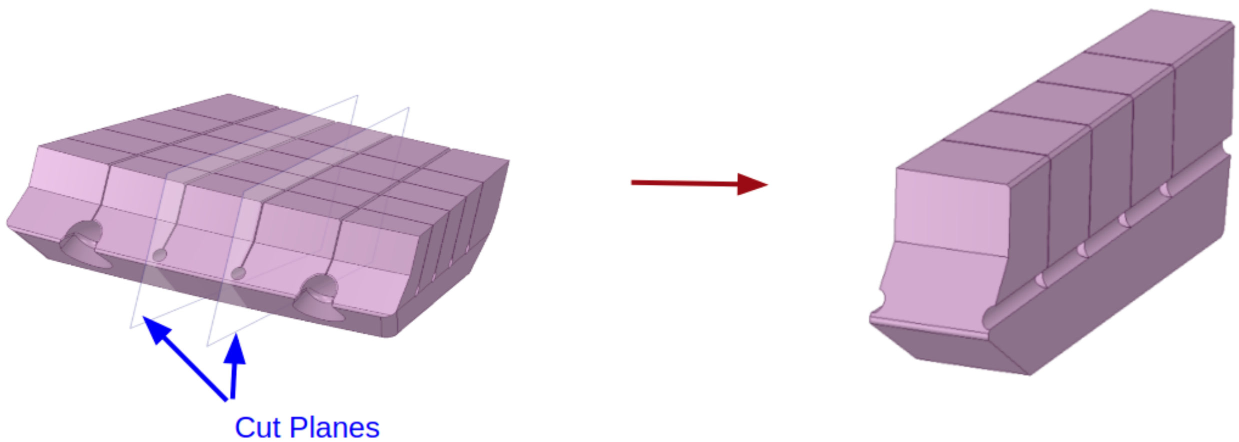
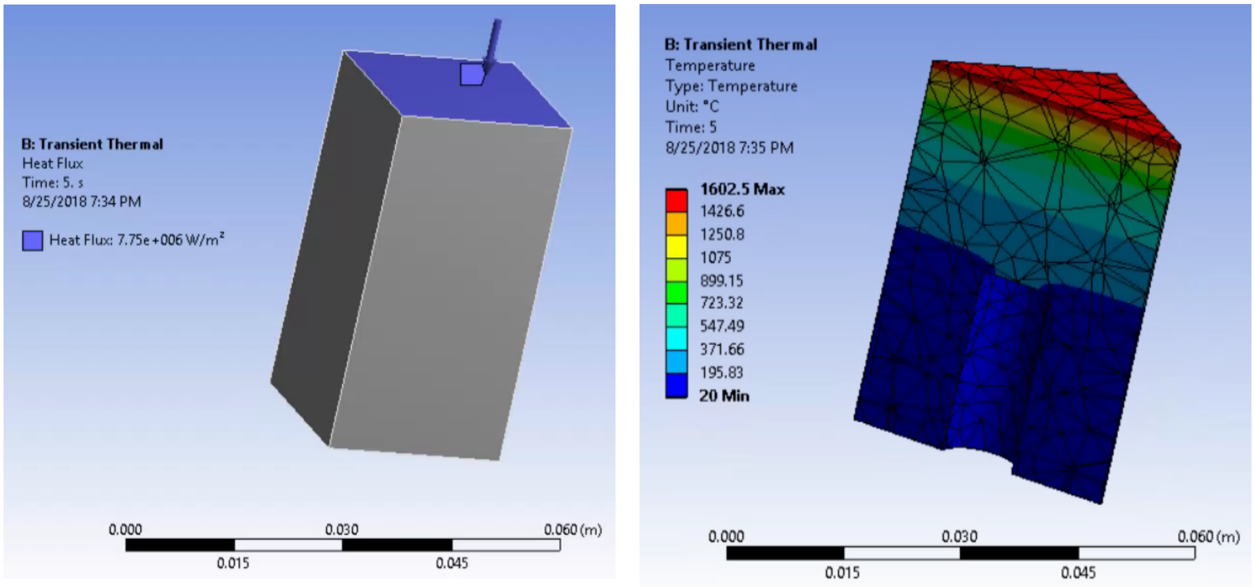


Figure 3.3: Reduction of full CAD model to reduced model

the NSTX-U mechanical engineering team. ANSYS enables users to perform a wide range of simulations ranging in topics from fluid dynamics, to electrostatics, to beam deflection, to heat loading (among many other simulations). Not only does ANSYS offer an intuitive user interface, but also enables low level programming via a parametric design language (APDL), or via a python application-program interface. CAD STL file importation is relatively simple in ANSYS (ANSYS natively reads STL), and the aforementioned reduced model was imported along with the SGL6510 graphite datasheet.

One intentional benefit of the castellated tile design is that the sublimation temperature will be reached before internal thermal stresses rise to levels that may cause fracture or tile destruction. In this manner these tiles are temperature limited to fail via sublimation rather than the more catastrophic mechanical failure. This has been experimentally demonstrated qualitatively by applying electron beam heat fluxes to the tiles [19]. To confirm this result, and to provide a baseline test for subsequent ANSYS simulations, the maximum allowable heat flux that would avoid sublimation was calculated via finite element simulation.

The temperature limit in the NSTX-U PFC requirements is stated to be 1600°C. In this simulation, a constant heat flux was applied to a single castellation for a period of five seconds (also specified by the PFC requirements) and the rise in surface temperature was simulated. For simplicity, a constant temperature boundary condition was placed on the bottom face (lower boundary) of the castellation, and the heat flux was applied orthogonally to the castellation surface. Material properties were obtained directly from the NSTX-U



(a) Applied heat flux

(b) Temperature results

Figure 3.4: Example Heat Flux Applied to Single Castellation with Results

engineers. As can be observed in figure 3.4, the 1600°C sublimation temperature was reached when approximately 7.75 MW m^{-2} was applied across the castellation surface (radiation neglected). This initial simulation confirmed previous NSTX-U engineering results, and also provided a valuable baseline for subsequent simulations. In figure 3.4 the thermocouple aperture is approximately one inch from the tile surface (aperture size exaggerated for image). Because there is little change in temperature at this distance from the heat flux application surface, it is obvious that the thermocouple hole must be less than one inch from the tile surface.

The results from this simulation confirmed that for meaningful temperature changes to be recorded by the thermocouple, the 12 mm configuration is sufficient, and will yield maximum temperatures between 350°C and 600°C, which are within the allowable range for the Omega thermocouples. The simulation also confirmed that 7.75 MW m^{-2} applied uniformly to the upper tile surface will force the upper tile to reach its sublimation temperature of 1600°C. Figure 3.5 shows the temperature distribution of a cross section of the reduced model, after five seconds of uniformly applied 7.75 MW m^{-2} heat flux. The dark regions of finer mesh tetrahedra that extend up from the bottom of the tile are the thermocouple apertures. Again, a 22°C boundary condition was applied at the lower tile surface.

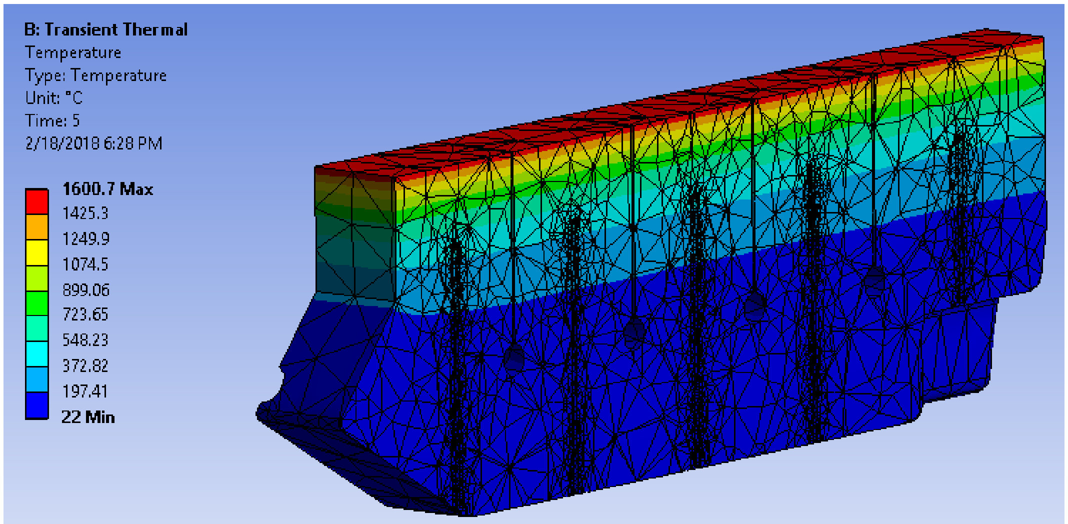


Figure 3.5: Temperature Cross Section for Uniformly Applied 7.75 $\mathrm{MW\,m^{-2}}$ Heat Flux

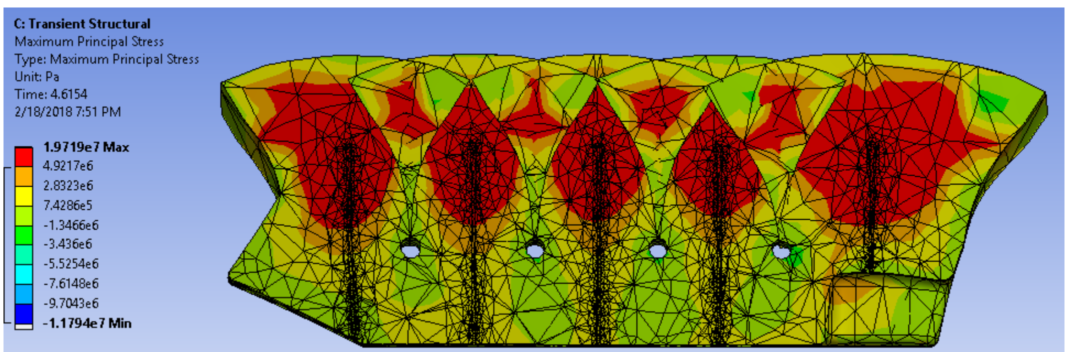


Figure 3.6: Maximum principal stress cross section for uniformly applied 7.75 $\mathrm{MW\,m^{-2}}$ heat flux (tile surface reaches 1600°C). Note: deformation exaggerated.

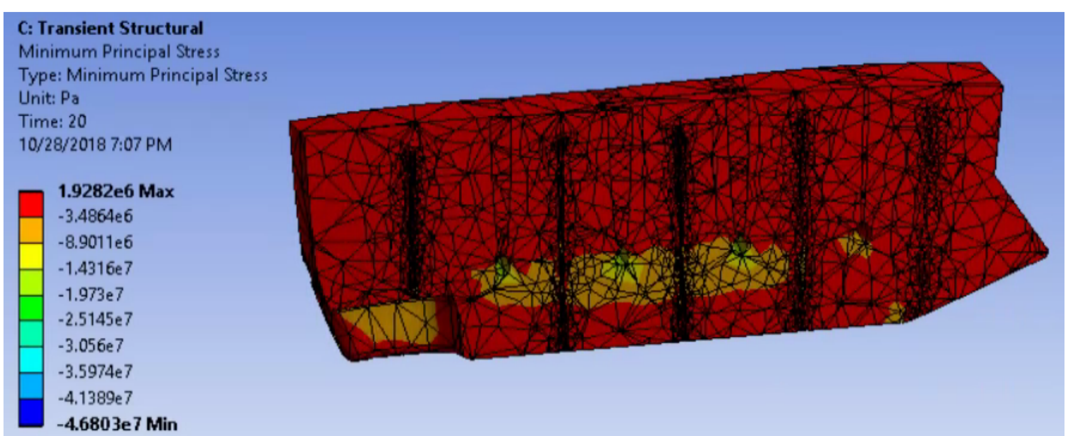


Figure 3.7: Minimum principal stress cross section for uniformly applied 7.75 $\mathrm{MW\,m^{-2}}$ heat flux (tile surface reaches 1600°C). Note: deformation exaggerated.

As was previously mentioned, the advanced graphites used for NSTX-U divertor tiles have an ultimate compressive strength of 130 MPa and an ultimate flexural strength of 60 MPa. Using a factor of safety of 2 yields an allowable compressive stress limit of 65 MPa and an allowable flexural stress limit of 30 MPa. To ensure that the tiles are indeed temperature limited (as compared to stress limited), the aforementioned transient thermal analysis was linked to a static structural analysis, and von Mises stresses were calculated for the case when 1600°C surface temperatures are encountered (7.75 $\mathrm{MW\,m^{-2}}$ uniform heat flux). This analysis indicates that the maximum flexural stress for this case is approximately 19.7 MPa, which is well within the allowable flexural stress limit for the SGL6510 graphite. Figures 3.6 and 3.7 illustrate the stress concentration that occurs around the thermocouple apertures as the castellations thermally expand. The resulting thermal stress is significant but not sufficient to cause mechanical failure. Because this condition occurs simultaneously to the 1600°C tile surface temperature, it can be concluded that this tile is indeed temperature limited.

A similar analysis was performed for the compressive stress, and it was determined to remain well below the 65 MPa limit. The maximum compressive stress, which was evaluated to be approximately 46 MPa, occurs at the lower tile boundary (lower surface). The 22°C heat sink on the lower tile surface is an idealized conception, and stress concentrations of this kind will not occur in NSTX-U. That being said, there may be some stress concentrations associated with the tile mounting system, but that analysis is outside of the scope of this work. Regardless of whether or not the 46 MPa compressive stress is real or not, it is well within the allowable stress limit of Sigrafine graphite, so it can be considered benign. It was necessary, however, to ensure that these idealized boundary conditions did not adversely effect the temperature results that would be eventually used for subsequent project objectives.

To quantify the effects of these idealized boundary conditions with regard to temperature, two simulations were compared. A five second, 8 $\mathrm{MW\,m^{-2}}$ heat flux was applied uniformly to the tile surface in both simulations In one simulation, a 22°C heat sink was applied at the tile boundary (lower surface). In the second simulation, a perfect insulator was placed at the boundary. Figure 3.8 provides a time evolution temperature comparison of both of these cases, for four different thermocouple elevations (0.25 inch, 0.5 inch, 0.75 inch, and 1 inch from the tile surface). At each elevation the perfect insulator and perfect heat sink provide nearly identical temperature profiles until approximately 15s, at which point the profiles diverge slightly. Because the true boundary conditions for NSTX-U will not be a perfect heat sink nor a perfect insulator, but somewhere in between, it can be concluded that the divergence associated with a finite thermal resistance is bounded by these cases, and therefore can be deemed negligible for any thermocouple elevation greater than 1 inch in this study. For the real NSTX-U thermocouples the process of selecting the minimum acceptable ΔT is determined by the resolution of the analog to digital converter (and any associated signal amplification / semiconductors) to which the thermocouple is connected.

Figure 3.8 also provides an illustration of the time delay associated with thermal diffusion in the tiles. A typical NSTX-U shot will consist of a five second plasma 'burn' (or discharge) after which the plasma will be terminated. During this five seconds, a heat flux will be applied directly to the divertor tiles. While there is no longer an applied heat flux after the burn, residual heat within the tile castellations will diffuse as a means of reaching equithermal conditions throughout the tiles. This delayed thermal diffusion can cause secondary thermal stresses within the tiles. The surface temperature increases to a maximum during the shot, and then decays exponentially after the shot has ended. After the plasma heat loading terminates, a thermal 'wave' flows through the tile, carrying information regarding the heat flux directly above it, and reaches the measurement point in time proportional to the distance it must cross. After this thermal wave has passed, transverse (or lateral) thermal diffusion also takes place, and subsequent temperature measurements will yield a spatially and temporally integrated temperature, as the castellation attempts to maximize entropy.

In addition to determining the effect of idealized boundary conditions, a mesh convergence study was undertaken to verify that the results contained herein were derived with a finite element mesh of sufficient resolution. While it is usually advisable to utilize the highest resolution possible when performing finite element simulations, high resolution meshes come at the cost of increased microcontroller cycles. It is therefore desireable to utilize the minimum resolution mesh that provides accurate results to increase simulation speed. To determine the appropriate mesh resolution, A heat flux was applied to the tile and

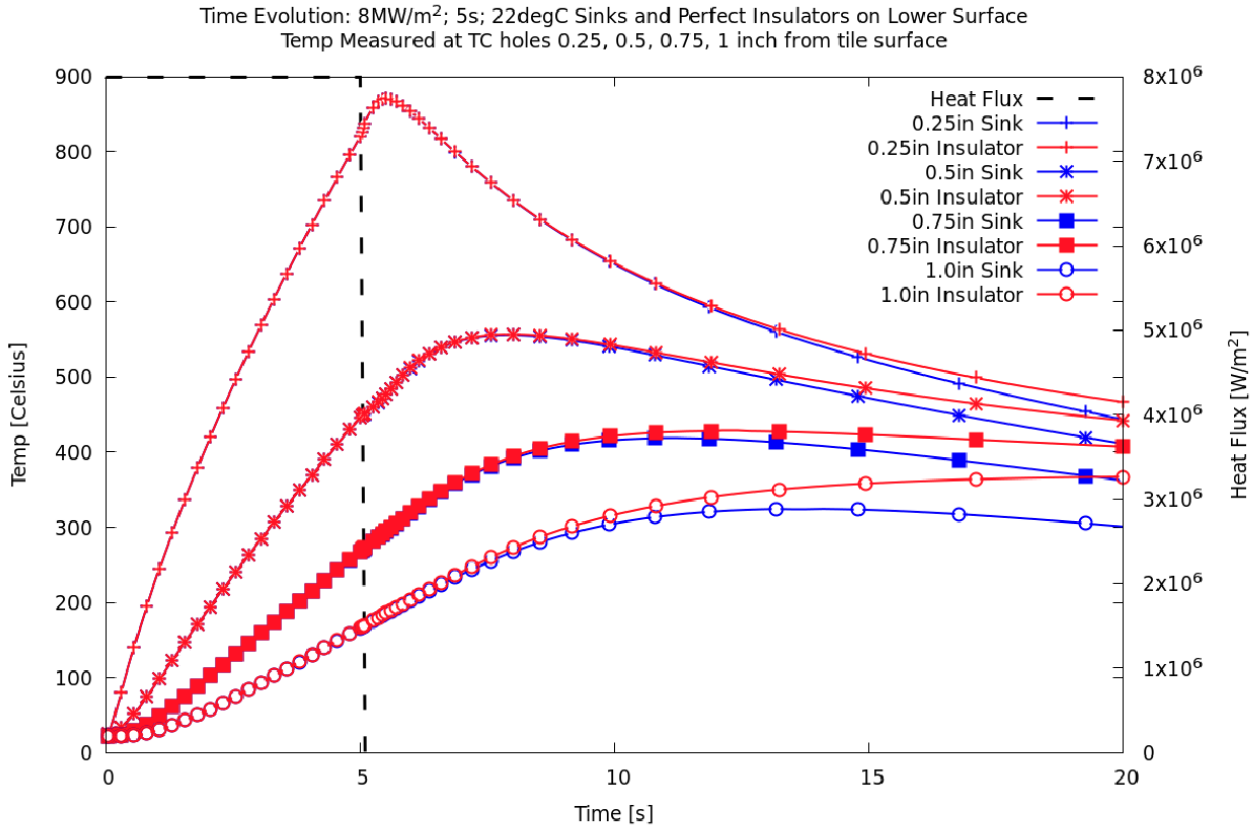


Figure 3.8: Comparison of opposite idealized thermal resistances at lower tile surface for varying thermocouple elevation.

the temperature profile was obtained as previously discussed. This process was repeated seven times, each with increasing mesh resolution. After all seven temperature profiles were generated, they were compared to determine if decreasing mesh resolution results in decreasing accuracy. Figure 3.9 provides the temperature profile obtained from the highest resolution mesh, and figure 3.10 provides the residual obtained from subtracting a lower resolution output from the highest resolution output. As can be observed, there is minimal difference between the high resolution and the lesser resolution with regards to the finite element solver results, and the residual never exceeds 5°C. There is, however, a significant difference in computing time between the two resolutions. The high resolution mesh required 372 seconds compared to 35 seconds required by the lower resolution mesh. Given the similar accuracy between the different mesh resolutions, it was therefore determined that the lower resolution mesh would provide sufficient accuracy while reducing the simulation time by a factor of nearly 10. Given the fact that many thousands of simulations would need to be

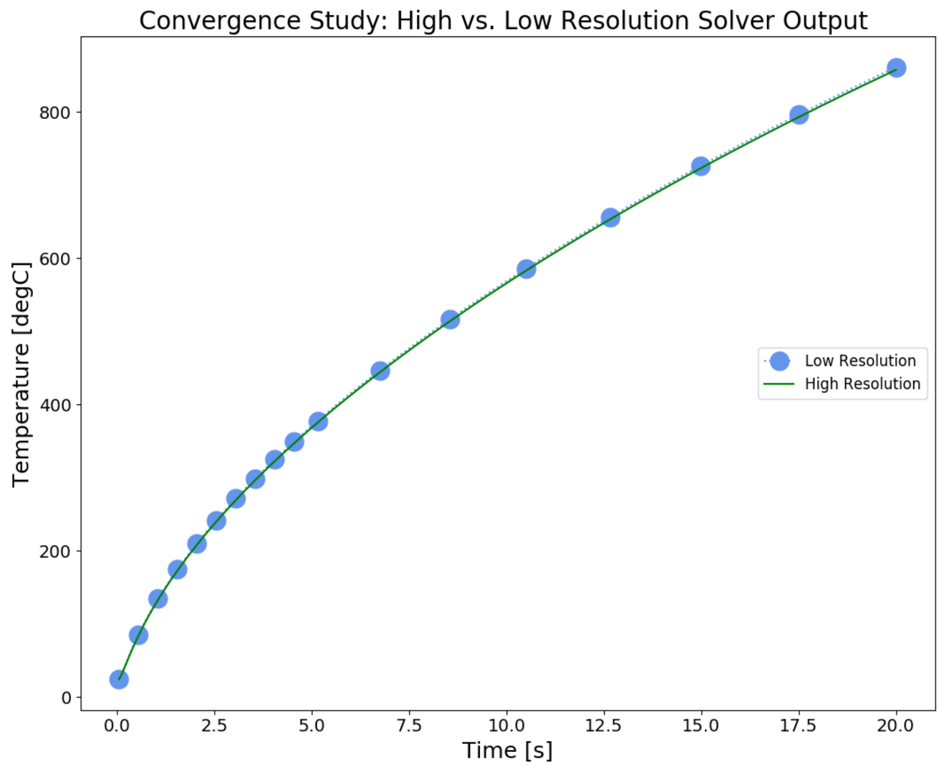


Figure 3.9: High resolution vs. low resolution FEM output

run eventually, the final mesh selection was ANSYS adaptive meshing with a resolution size of two.

In summary, the aforementioned simulations provided valuable insight into the characteristics of the NSTX-U graphite plasma facing components. With the assistance of a capable multiphysics solver, ANSYS, the domain of operation for the NSTX-U tokamak was determined. The maximum allowable heat flux that may be applied uniformly to any tile surface for a duration of 5 seconds is 7.75 $\mathrm{MW\,m^{-2}}$. This level of heat flux yields a 1600°C temperature on the tile surface, the defined maximum temperature. At this heat flux, the maximum compressive and tensile (flexural) stresses were calculated, and it was determined that neither stress exceeded its allowable stress (factor of safety of 2), signifying that the tiles are temperature limited rather than stress limited at maximum heat flux. Additionally, an investigation into the ideal elevation for thermocouple placement was performed. At 0.25 inches from the tile surface, the thermocouples yield the most information, yet may reach the temperature limit for the sheathing of the omega thermocouples. At 1.0 inches from the tile surface, the thermocouples are far less sensitive to the temperature wave propagating through

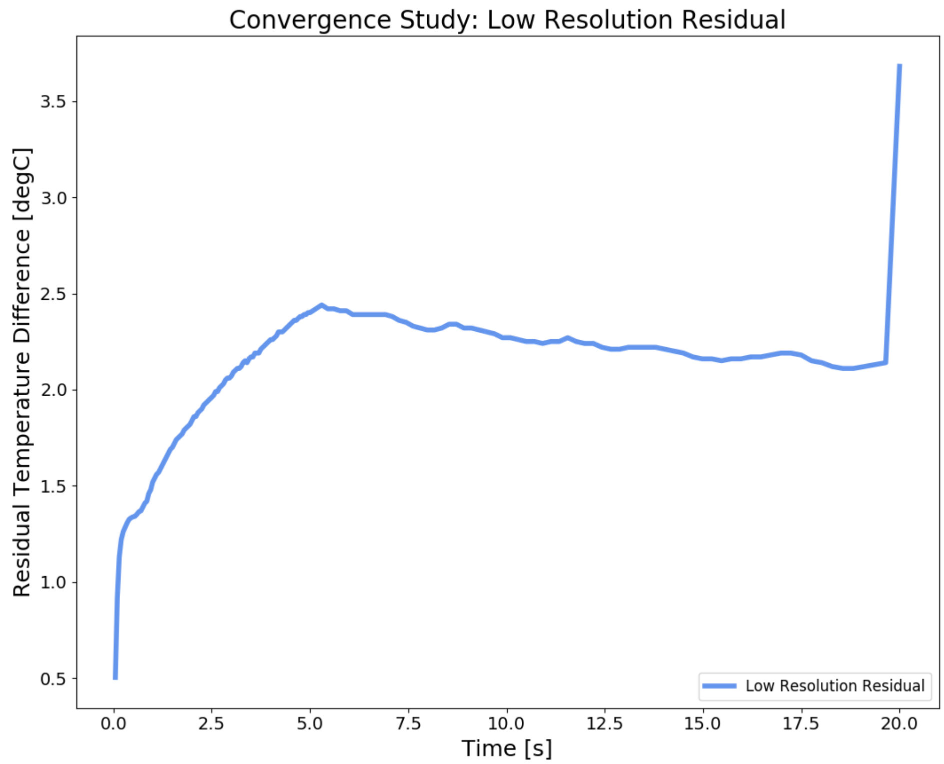


Figure 3.10: Low resolution mesh residual when compared to high resolution mesh.

the tile. Therefore , the ideal elevation for the thermocouples is between 0.25 inches and 1.0 inches from the tile surface, where they balance a large SNR with moderate temperatures.

Having established 1) the heat flux operational limit to maintain temperatures below the sublimation limit and 2) the ideal elevation of thermocouples to balance signal strength and temperature, the next step for completion of objective 1 required that a method for exposing the tiles to spatially and time varying heat fluxes (as opposed to constant heat fluxes) be developed. More specifically, a realistic heat flux profile needed to be applied to the tile surface, rather than a flat 7.75 $\mathrm{MW\,m^{-2}}$ profile. In chapter 2, the Eich heat flux profile was introduced. The Eich heat flux profile provides a realistic inter-ELM profile for tokamaks operating in H-mode, therefore provided useful guidance for this investigation.

3.4 Simplified Eich Model

Chapter 2 provided an introduction to the Eich heat flux model. The Eich model provides a realistic heat flux profile for Scrape off layer (SOL) plasma as it strikes the divertor target.

As a means of satisfying objective 1, it was necessary to implement a realistic heat flux as a means of obtaining simulated time and spatially varying thermocouple data that is similar to the data that would be obtained from a real discharge in NSTX-U. To complete the first objective, it was necessary to:

- Generate realistic heat flux data
- Feed this data to ANSYS as a means of generating realistic thermocouple profiles
- Compile this thermocouple data into a dataset of simulated NSTX-U shots

It was determined that for the proof of concept demonstration, a simplified Eich heat flux profile would suffice, so long as the proof of concept system could be scaled to the original Eich heat flux after the proof of concept was completed. After multivariate regression the original Eich heat flux model can be represented by the empirical formula described in chapter 2. This formula indicates that λ_q is governed by five scaling parameters, C_0, C_B, C_q, C_p, and C_R. For this proof of concept, the original profile was replaced with a simpler profile consisting of only four scaling parameters defining the characteristic heat flux length:

$$\lambda_q[mm] = C_2 P_{heat}^{C_3} B_p^{C_4} \tag{3.1}$$

where B_p [T] is the poloidal field, $C_1; C_2; C_3; C_4$ are scaling parameters that will be called the *Eich parameters*, and P_{heat} [MW] is the integrated power striking the divertor surface, and can be represented by the integral,

$$P_{heat} = \int_{R_0 - x_p}^{R_0 + x_c} q(R) 2\pi R \mathrm{d}R \tag{3.2}$$

where R [m] is the radial coordinate, R_0 [m] is the strike point location, x_p [m] represents the length of flux decay in the private flux region, x_c [m] represents the length of flux decay in the common flux region, and $q(R)$ [MW m^{-2}] represents the new simplified heat flux profile. It is important to recognize that the Eich parameters are not dependent upon machine operation, but rather are physics constructs. An illustration of the original heat flux profile is provided in figure 3.11, and the simplified heat flux profile is provided in figure 3.12.

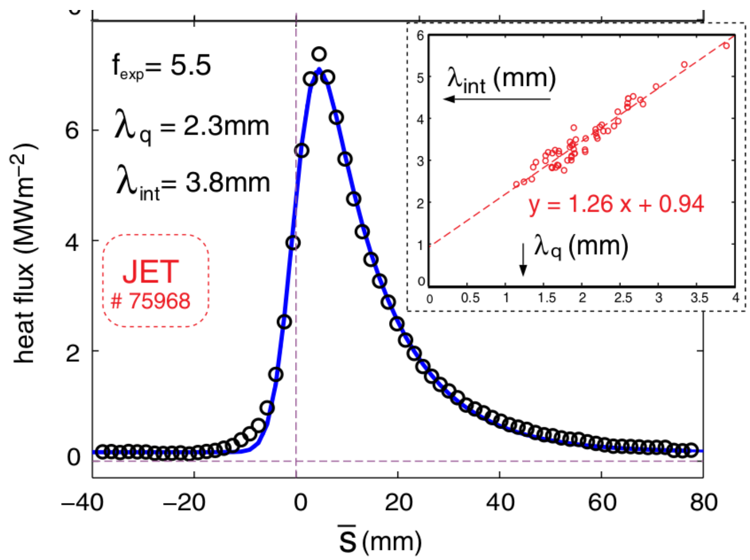


Figure 3.11: Original Eich heat flux profile from [2]

The original heat flux profile provided by Eich is derived by the convolution of the exponential power decay with a thermal diffusion gaussian, as described in chapter 2. This convolution was unnecessary to demonstrate the proof of concept, so the triangular flux shown in figure 3.12 was selected as a surrogate. Clearly, this triangular flux is a piecewise function of the single independent variable, R, consisting of two linear functions,

$$q(R) = \begin{cases} 0 & R \leq \alpha \text{ or } R \geq \beta \\ \frac{q}{x_p}(R - \alpha) & \alpha < R < R_0 \\ q + \frac{q}{x_c}(R_0 - R) & R_0 < R < \beta \end{cases}$$

where α [m] represents the edge of the private flux region $(R_0 - x_p)$, β [m] represents the edge of the common flux region $(R_0 + \beta)$, and q represents the peak heat flux. In order to generate heat fluxes with this simplified model, it would therefore be necessary to solve for the value of q. Solving the integral given in equation 3.2 and performing some mild algebra yielded an explicit result for q, the peak heat flux, in terms of P_{heat}, x_p, x_c, R_0,

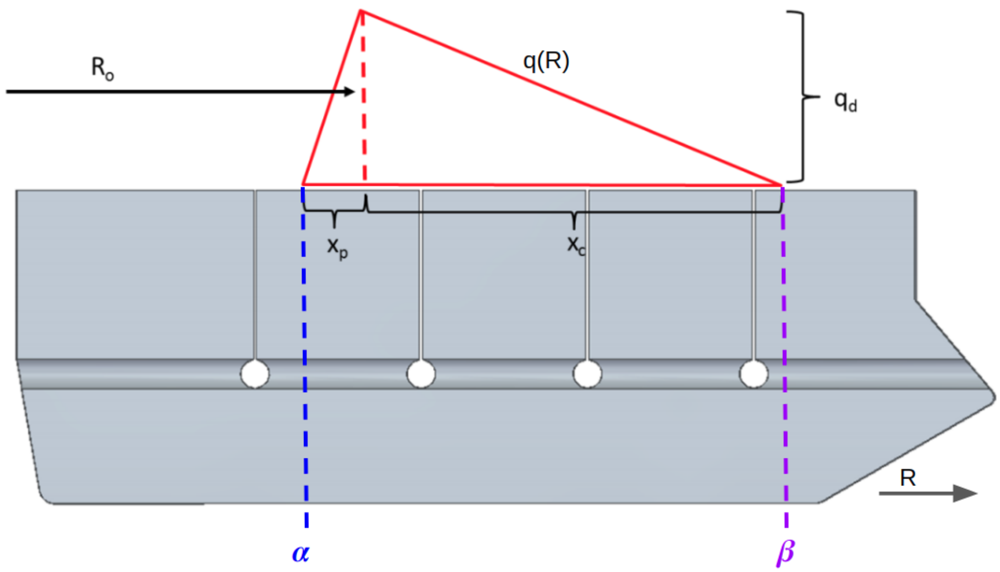


Figure 3.12: Simplified Eich heat flux profile

$$P_{heat} = \int_{R_0-x_p}^{R_0+x_c} q(R) 2\pi R \mathrm{d}R \tag{3.3}$$

$$\Rightarrow \frac{P_{heat}}{2\pi} = \int_0^{\alpha} (0) R\mathrm{d}R + \int_{\beta}^{\infty} (0) R\mathrm{d}R + \int_{\alpha}^{R_0} \frac{q}{x_p}(R-\alpha) R\mathrm{d}R + \int_{R_0}^{\beta} q + \frac{q}{x_c}(R_0 - R) R\mathrm{d}R \tag{3.4}$$

Solving for q yields:

$$q = \frac{P_{heat}}{2\pi}\left[\frac{1}{3x_p}(R_0^3-\alpha^3) - \frac{\alpha}{2x_c}(R_0^2-\alpha^2) + \frac{1}{2}(\beta^2-R_0^2) + \frac{R_0}{2x_c}(\beta^2-R_0^2) - \frac{1}{3x_c}(\beta^3-R_0^3)\right]^{-1} \tag{3.5}$$

In addition to the equations for λ_q, P_{heat}, and $q(R)$, the simplified model consists of a set of relations (given in PFC MEM0 015 [7]) between x_p, x_c, f_x C_1, λ_q, and a private flux width parameter S (synonymous to the Gaussian width S from the original model) which are defined as follows:

$$S = C_1\lambda_q \tag{3.6}$$

$$x_p = Sf_x \tag{3.7}$$

$$x_c = \lambda_q f_x \tag{3.8}$$

$$C_1, C_2, C_3, C_4 \quad \text{Eich Parameters} \tag{3.9}$$

$$B_p, P_{SOL}, f_x, \text{etc.} \quad \text{Machine Specifications} \tag{3.10}$$

Lastly, it is necessary to constrain the domain of these variables to realistic values that are achievable in NSTX-U. In the context of the simulation objectives, this prevents any parameters outside of the operational domain of the NSTX-U tokamak from entering into the input. The values provided in PFC MEMO 015 for NSTX-U machine parameters are given to be:

$$0.2 < B_p < 0.6 \tag{3.11}$$

$$0.5 < P_{heat} < 4.9 \tag{3.12}$$

$$4 < f_x < 30 \tag{3.13}$$

$$46.0 < Ro < 57.5 \tag{3.14}$$

$$1 < t[sec] < 5 \tag{3.15}$$

$$\tag{3.16}$$

where t corresponds to the discharge duration, or shot length. The constraints imposed upon the Eich parameters are similarly defined in the PFC MEMO 015:

$$0.1 < C_1 < 0.3 \tag{3.17}$$

$$1.0 < C_2 < 2.5 \tag{3.18}$$

$$-0.1 < C_3 < 0.25 \tag{3.19}$$

$$-1.4 < C_4 < -0.5 \tag{3.20}$$

Now that the simplified model has been defined and the domain for each variable has been constrained, it is possible to create a simulated heat flux generator that will produce simplified Eich heat fluxes. These fluxes will become the inputs to ANSYS as a means of simulating NSTX-U discharges with regard to the inboard horizontal divertor tiles.

3.5 Monte Carlo Heat Flux Generator

All code for this project is located in the Appendix. A Github Repo with code for this project can be found here: https://github.com/plasmapotential/NSTX_heatflux.

Project objective one requires quantifying the response of NSTX-U PFCs to realistic heat fluxes that will eventually be observed in the machine. The simplified Eich model, now explicitly formulated, provides the mathematics required to generate such a heat flux. Under normal operation, NSTX-U's machine specifications, B_p, P_{heat}, f_x, R_0, and t, will be predefined. Associated with these machine specifications are a specific set (or sets) of Eich parameters, C_1, C_2, C_3, C_4, although these theoretical values are not controlled by an operator nor associated with any hardware. Together, the machine specs and Eich parameters characterize the heat flux profile in the SOL. The Eich parameters must be derived experimentally and correlated to the machine specs to predict the heat flux profile. In other words, for a given set of machine specs there are multiple possible heat flux profiles, each corresponding to a specific set of (unknown, but fixed) Eich parameters. Project objective two requires a subsurface thermocouple measurement system to be capable of deriving the Eich parameters with a limited number of shots, which would enable heat flux model validation. Project objective 3 requires the system to be robust against noise and uncertainties associated with the diagnostic system. In addition to constructing a heat flux generator that would satisfy objective one, it was desirable to construct a modular heat flux generator that could serve all three objectives simultaneously.

In order to validate the heat flux model, it would be necessary to generate a large number of heat fluxes for testing and system validation. This large dataset would need to sample the entire domain of the machine specs and Eich parameters, in order to simulate the entire operational domain of NSTX-U, and scales with the number of variables to be predicted.

Section ?? provides the precise domain for each variable. To sample through the entire operational domain without bias, and to generate large datasets quickly and efficiently, a Monte Carlo flux generator implementation was determined to be the superior selection for a flux generator. Monte Carlo analysis exists at the nexus of statistics and numerical methods, and provides a method for generating randomness in numerical code. In this case, thousands of heat flux profiles that randomly sample the entire operational domain of NSTX-U were generated. A FORTRAN Monte Carlo flux generator was developed for creating heat flux profiles that could easily be imported to ANSYS and applied to the surface of the divertor tiles.

The heat flux generator utilizes a linear congruential random number generator to pull a standard deviate (random number) for each variable that must be predefined in the NSTX-U model. The Heat Flux Generator code is provided in the appendix, and is available via a github repo. The FORTRAN script uses the CPU clock as a seed for the FORTRAN library subroutine `RANDOM_SEED` which generates pseudo-random numbers with a period of $2^{1024}-1$. The bounds of each variable defined in section ?? were placed in an array, and a separate standard deviate was pulled for each variable. This random number was then mapped to a uniform probability density function (PDF) between the bounds defined in the bounds array. This process can be represented mathematically.

Given a non-normalized probability density function, $\widetilde{\pi}(x)$, lower domain bound α, and upper domain bound β, a normalized probabilty density function can be derived via the formula,

$$\pi(x) = \frac{\widetilde{\pi}}{\int_{\alpha}^{\beta} \widetilde{\pi}(x')\mathrm{d}x'}. \tag{3.21}$$

This PDF is then transformed into a cumulative distribution function (CDF), $\Pi(x)$, which represents the probability that a random variable, X, will take a value less than or equal to x. This can be achieved via the equation,

$$\Pi(x) = \int_{\alpha}^{x} \pi(x)\mathrm{d}x. \tag{3.22}$$

The FORTRAN pseudo-random number generator generates standard deviates, ζ, between 0 and 1. In order to map the standard deviates to the given PDF, the standard deviate is set equal to the CDF. The integral is solved, and then the random variable, x, is solved for in terms of ζ:

$$\zeta = \Pi(x) = \int_{\alpha}^{x} \pi(x)\mathrm{d}x. \tag{3.23}$$

$$x = \Pi^{-1}(\zeta) \tag{3.24}$$

In the case of the NSTX-U heat flux generator the desired probability density function is uniform for all variables, so this process is identical for all variables. The flux generator will sample through all of the machine specs and Eich parameters evenly, and generate heat flux profiles that represent the entire operational domain of NSTX-U. As an example, a standard deviate, ζ, will be mapped to a variable, x, within the domain: $a < x < b$. The solution to this mapping will serve as a map for all Eich parameters and machine specs, and only the bounds will need to be replaced for each respective variable. For the uniform PDF case, the PDF is given by,

$$\widetilde{\pi}(x) = 1 \tag{3.25}$$

Normalize to the allowable domain for the variable:

$$\pi(x) = \frac{\widetilde{\pi}}{\int_{\alpha}^{\beta} \widetilde{\pi}(x')\mathrm{d}x'} = \frac{1}{b-a} \tag{3.26}$$

Convert to CDF and set equal to standard deviate:

$$\Pi(x) = \int_{\alpha}^{x} \pi(x)\mathrm{d}x = \int_{a}^{x} \frac{1}{b-a}\mathrm{d}x = \frac{x-a}{b-a} = \zeta \tag{3.27}$$

Solve for $x(\zeta)$:

$$x(\zeta) = \zeta(b-a) + \alpha \tag{3.28}$$

Equation 3.28 enables a machine spec or Eich parameter to be generated from a standard deviate between 0 and 1. The flux generator operates by pulling standard deviates (between 0 and 1) for each machine spec and Eich parameter and then mapping the standard deviates to random variable (between bounds). After all random variables have been generated, the system solves for S, x_p, x_c and λ_q using the relationships defined in equations 3.6 - 3.10, using

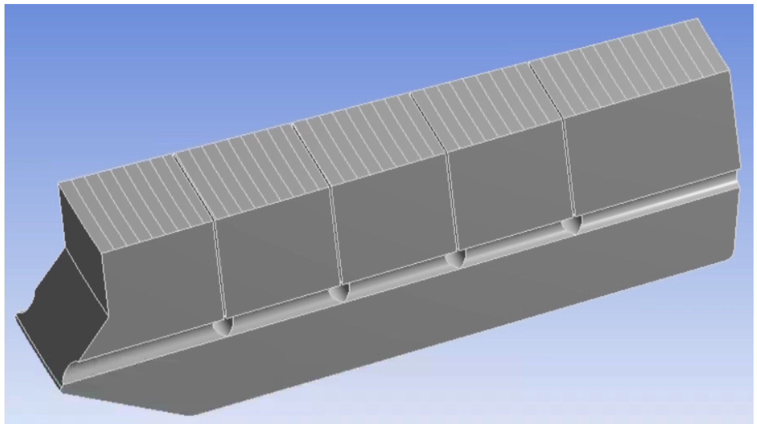


Figure 3.13: Tile surface discretized into 50 slices

the Eich parameters. The newly discovered S, x_p, x_c and λ_q are then utilized to derive q, in equation 3.5. It should be mentioned that the Eich parameters are utilized to generate the original heat flux profile, are hidden from the NSTX-U ANSYS simulations, and then are used to verify the Eich heat flux validation system (objective 2).

Discretization of the spatial and temporal heat flux profile on tile surface is necessary. When the script is launched, the user is queried for the number of discrete radial slices, dx, that the tile surface should be discretized into, as well as the number of time steps the shot should be discretized into, dt (not the same as shot length, Δt). These user inputs define the temporal and spatial resolution of every flux generated, and can be tuned on the fly for specific simulation resolution needs. An example NSTX-U graphite tile slice that has been discretized into 50 slices is given in figure 3.13. In addition to the resolution, the user may specify the number of fluxes to be generated. For the duration of this work, fluxes have been generated in batches of 10k, which takes roughly 30 seconds on a four core i7 microcontroller.

Once a batch of discretized heat flux profiles has been generated by the FORTRAN script, the batch is transferred to the ANSYS machine for importation and simulation of an NSTX-U shot. The output of the heat flux generator is formatted as a CSV file, consisting of a two dimensional array, where one dimension represents time and the other represents the radial direction across a divertor tile surface. To illustrate the heat flux generator output, figure 3.14 provides a plot of the first time step of ten randomly chosen flux profiles in which the machine specs have been fixed to a constant value. Each flux profile corresponds to

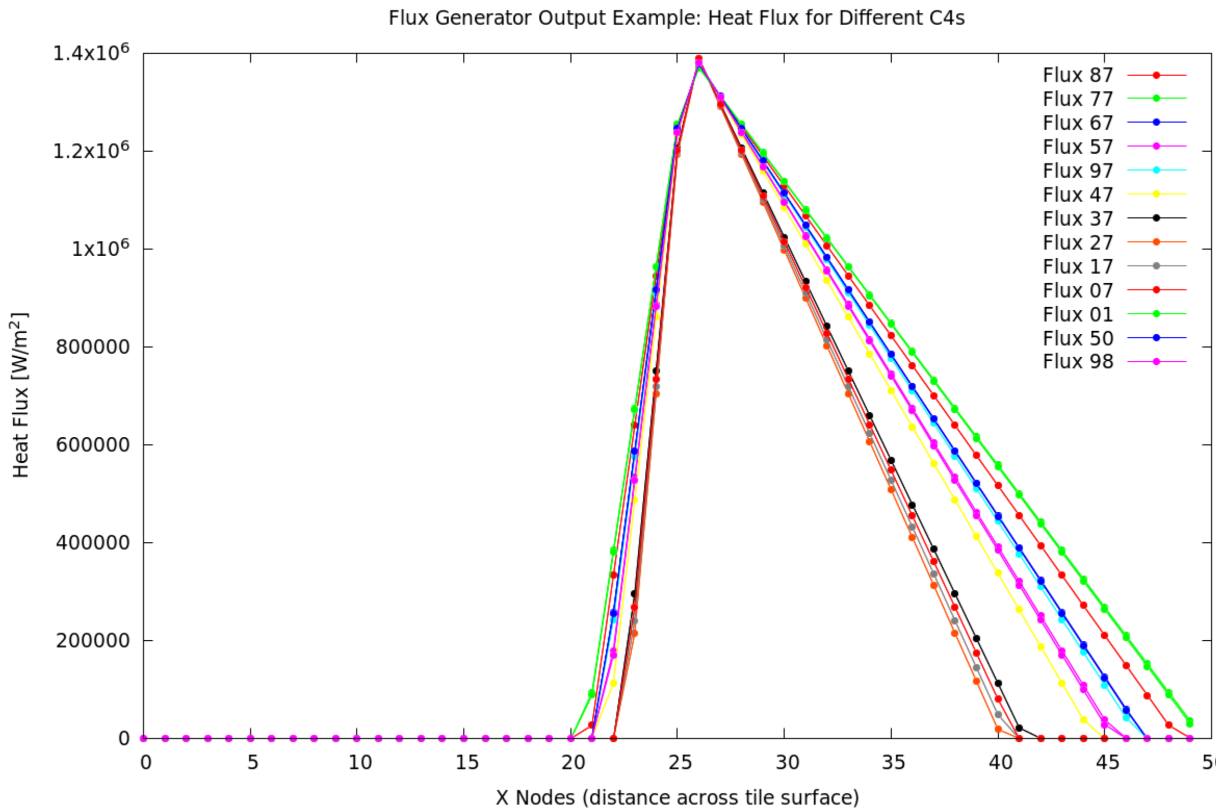


Figure 3.14: Example heat fluxes varying C_4 only. Fixed strike point.

variations in one Eich parameter, C_4, while the strike point, machine specs, and C_1 - C_3 were held constant. The lower axis is the radial direction corresponding to the tile length. For comparison, figure 3.15 provides a plot of the final time step of ten different flux profiles, in which the Eich parameters and machine specs (including strike point) were varied via the Monte Carlo random sampling process. As can be observed, a wide range of heat fluxes is possible within the allowable operational domain, and some heat fluxes even fall off the surface of the tile. Figure 3.16 provides a histogram of all four Eich parameters, and indicates the Monte Carlo method was successful at pulling variables from a uniform distribution.

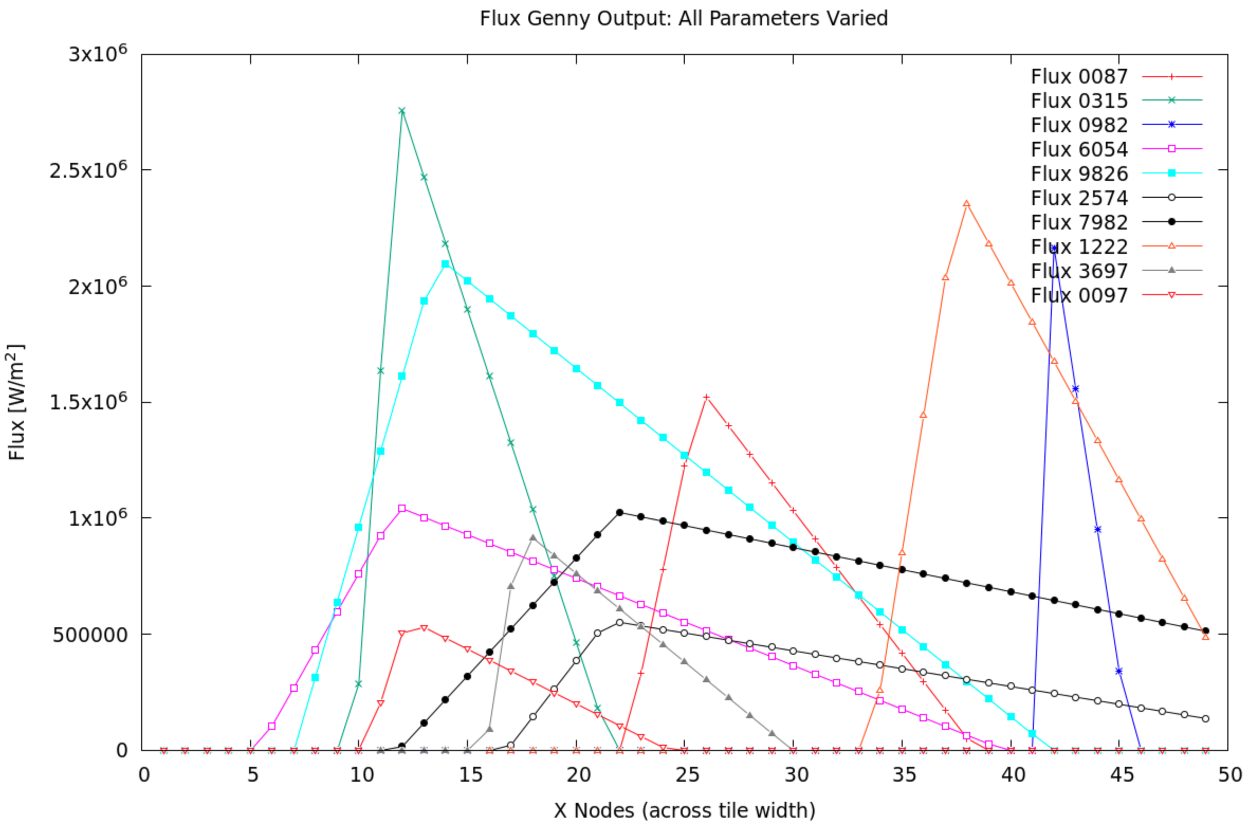


Figure 3.15: Example heat fluxes varying Eich parameters and machine specs

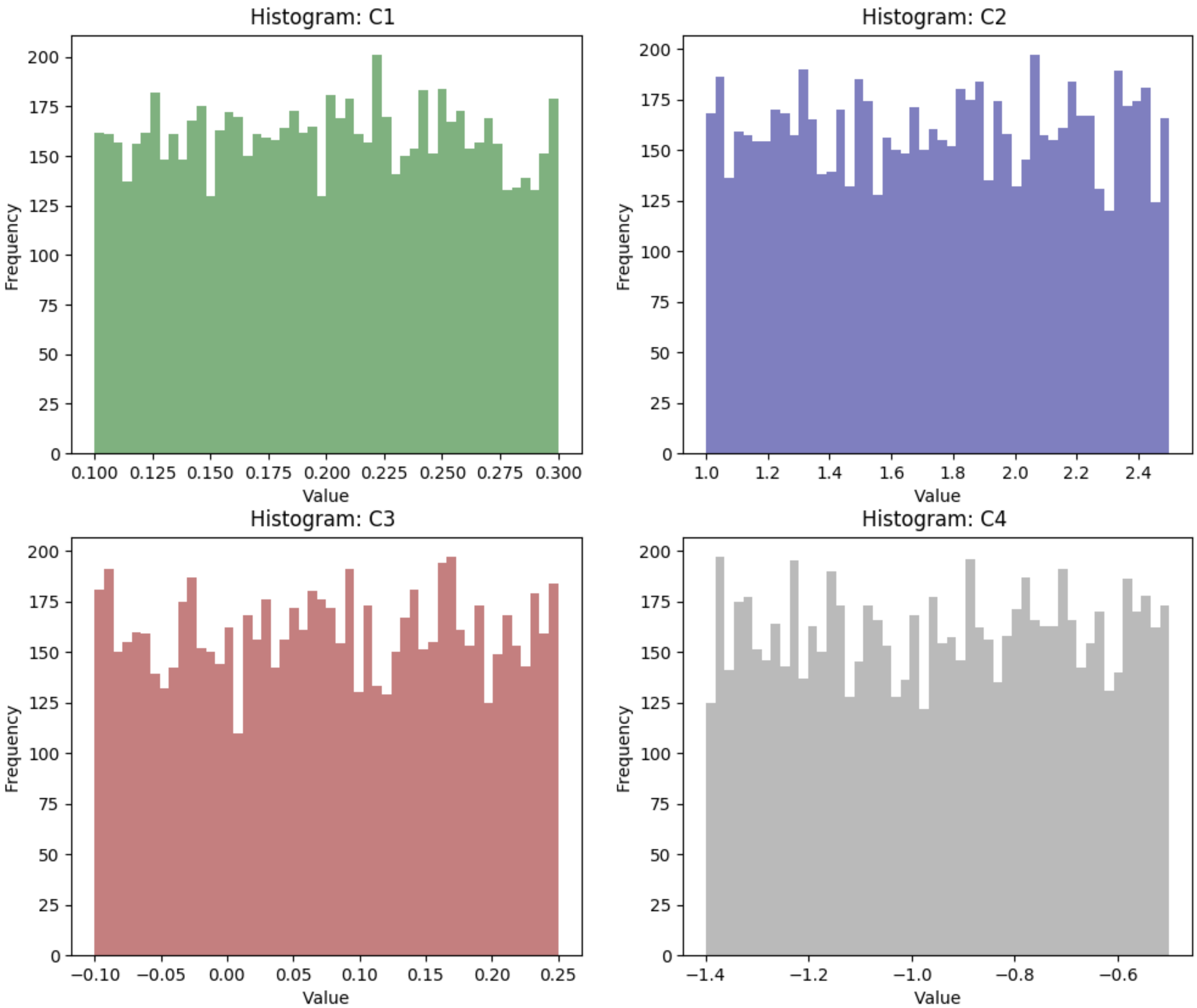


Figure 3.16: Histograms for each Eich Parameter Generated

3.6 ANSYS ACT Solver

All code for this project is located in the Appendix. A Github Repo with code for this project can be found here.

After heat fluxes had been generated with the Monte Carlo heat flux generator, they were ready for importation into ANSYS as a means of simulating NSTX-U shots. By discretizing the divertor tile and generating a similarly discretized heat flux profile, it was possible to import the heat flux directly in the form of a two dimensional array corresponding to the radial direction vs time. The spatial and temporal resolution of the heat flux profile can be as coarse or fine as necessary, depending upon the application requirements. For all of the simulations contained herein, the tile surface and the shot duration were both discretized into 50 steps. In ANSYS, the simplified inboard horizontal divertor tile geometry CAD model was partitioned into 50 radial slices (see figure 3.13) and a 22°C heat sink was placed on the lower tile surface. A thermal probe was placed 0.25 inches from the tile surface in each castellation, at the terminus of the thermocouple apertures. These probes would record the time varying temperature as the heat flux was applied, and for 15-20 seconds afterwards.

One method for applying heat fluxes in ANSYS is manual importation of a time varying heat flux vector to a single surface. This method is sufficient for simple analyses, but in this case each tile consisted of 50 surfaces, each with a separate heat flux vector. Furthermore, it was apparent that several thousand simulations would be required to validate the model. For these reasons, the basic ANSYS multiphysics suite was deemed impractical for the task of generating the NSTX-U simulation dataset. ANSYS allows users to implement external features using either the ANSYS Parametric Design Language (APDL) or via ACT extensions. ACT extensions were chosen because they enable control of data flow through an ANSYS program, and can be coded to augment existing ANSYS functions (or to develop new ones). An ACT extension was selected to circumvent the aforementioned heat flux importation limitations.

ANSYS ACT can be employed to implement functions and graphical user interface (GUI) toolbar buttons in any of the ANSYS programs available from the ANSYS Workbench. ACT extensions consist of (at a minimum) two files. The first file, an eXtensible Markup Language

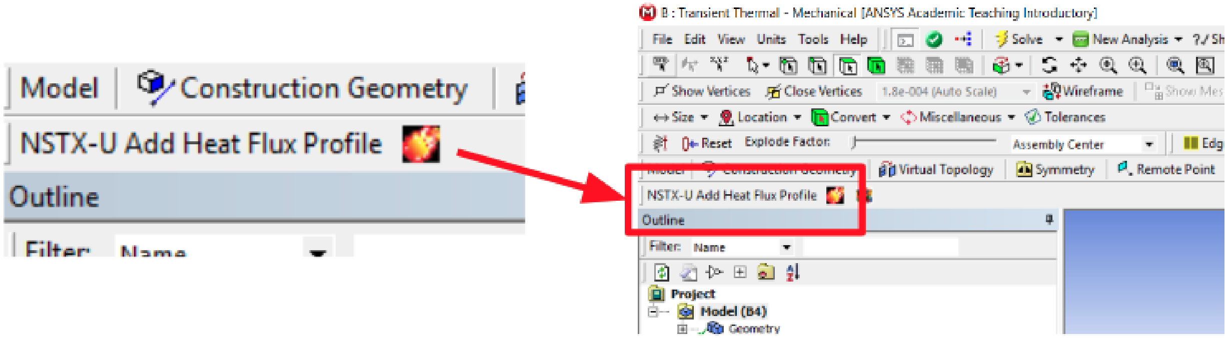


Figure 3.17: ANSYS ACT Flux Importation GUI Button

(XML) file, provides GUI specifications, such as button size and shape. Additionally, the XML file defines the actions to be taken when a GUI button is clicked. In most cases, the action requires an external script; the XML file defines the script location. ANSYS ships from the factory with an IronPython library built into the application, which enables users to call python scripts from the XML file without installing python binaries on the local machine. The second script required for an ACT extension is the python source code that performs the desired function.

A flux importation ACT extension was developed, which enabled the CSV output from the flux generator to be imported into ANSYS. In ANSYS Mechanical, a GUI button appears on the ANSYS toolbar ribbon in the form of a fireball, and is labeled *NSTX-U Add Heat Flux Profile* see figure 3.17. When clicked, this fireball icon initiates a python script. This python script directly accesses the ANSYS application programming interface (API) to interact with ANSYS Mechanical. First, the program scans the surface of the divertor tile and autonomously applies the time varying heat flux vectors to the appropriate surface sections. The script then runs the simulation (the simulated NSTX-U shot) and records time evolving temperature data at each of the radially located thermocouple locations. After the shot completes, all of the time evolving thermocouple vectors are concatenated into a single 2D array, with five column vectors that correspond to the five radial thermocouple probes. Each row corresponds to a new time step, whose resolution is configurable. The data across a single row is therefore the radial thermocouple profile generated at a specific timestep. The ACT extension loops through the entire batch of heat fluxes generated by the Monte Carlo heat flux generator, and creates a time varying temperature profile for each shot. This process is moderately time consuming, although not personnel intensive. A 16 logical core workstation will take approximately 72 hours to complete 1000 simulations. Figure 3.17 is a screenshot of the ANSYS Mechanical toolbar, with exploded view of the NSTX-U heat flux button. Figures 3.18 and 3.19 provide an example of the Monte Carlo flux generator output, and the corresponding thermal solution generated by the ACT extension.

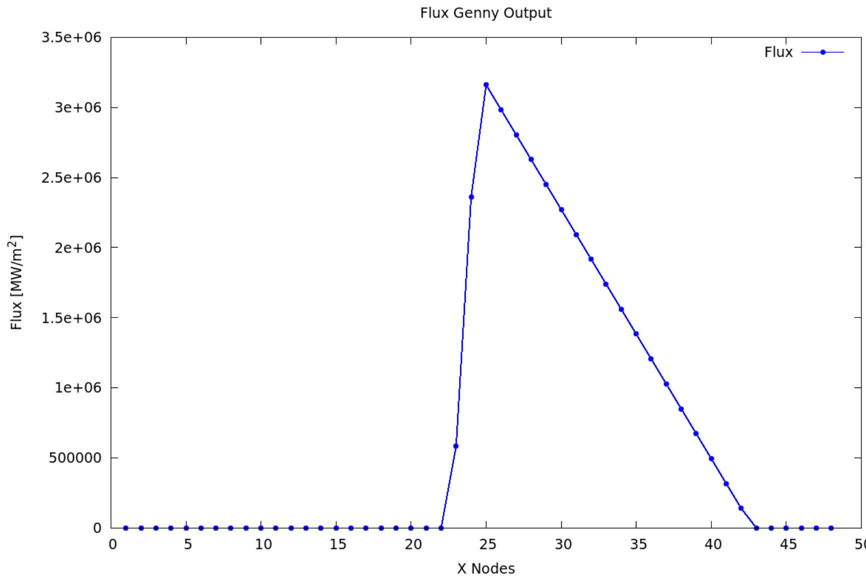


Figure 3.18: Example flux generator output

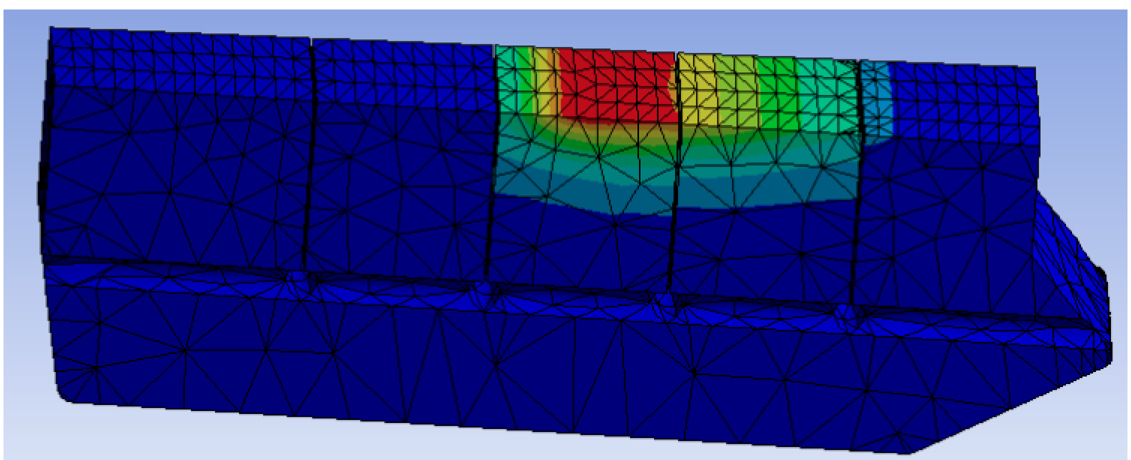


Figure 3.19: Example thermal solution to flux applied by ACT flux importation algorithm

Chapter 4

Deriving Eich Scaling Parameters

The second project objective is stated as: *Demonstrate how unknown heat flux model parameters can be derived with various sampling mechanisms within a given parameter space.* The primary topic of this chapter is the development of a method for reconstructing the Eich heat flux profile directly from subsurface thermocouple data, which would satisfy this objective.

4.1 Creating a Closed Loop System

As was previously discussed, a Monte Carlo heat flux generator pulled random values within the allowable domains of the Eich parameters and the machine specs, and constructed a heat flux profile based upon the simplified Eich model. The output from each instance of this heat flux generator was a temporally and spatially varying heat flux profile, as well as the values of the randomly chosen aforementioned variables, all in the form of a CSV file. The ANSYS ACT extension looped through the entire batch of heat flux profiles, and output the simulated temperature profiles that correspond to each heat flux profile. Once the simulated time evolving thermocouple dataset had been generated, it was necessary to develop a method to extract, or to reconstruct, the Eich parameters, C_1, C_2, C_3, C_4.

In the most simplistic form, this task amounts to providing subsurface thermocouple data to some transfer function, whose output is the simplified Eich model parameters. If the system operates correctly, then the transfer function output will be identical to the Eich

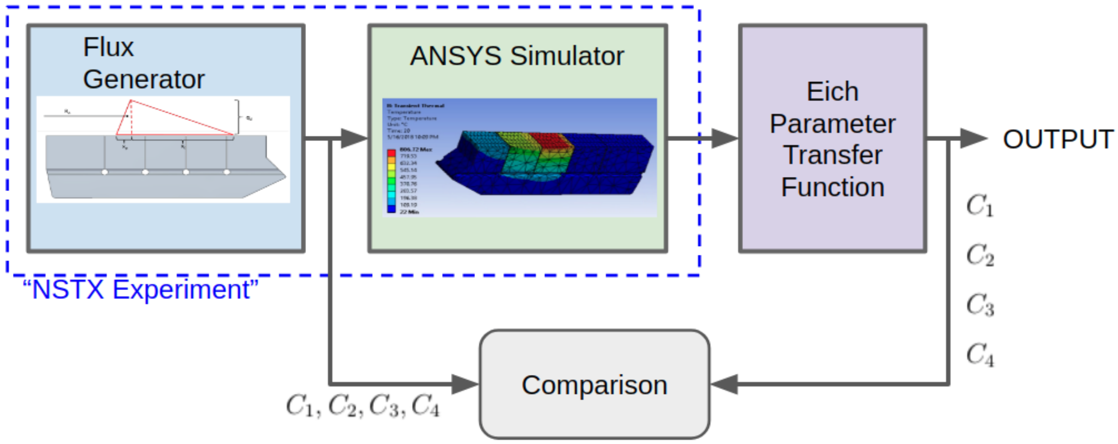


Figure 4.1: Closed Loop Process Flow Chart

parameters utilized by the flux generator to create the original heat flux profile, thereby satisfying the second project objective. This system could then be applied to a real NSTX-U shot where the heat flux to the divertor is defined by real physical processes rather than the Monte Carlo flux generator. In a real system, the Eich parameters will be undefined until the transfer function predicts them via the analysis of real subsurface thermocouple measurements. By creating a closed loop simulation system where the Eich parameters are predefined and the transfer function predictions are compared to these predefined parameters, it is possible to validate the accuracy of the transfer function before deployment.

This closed loop simulation method can be described by figure 4.1. In this figure, the flux generator output serves as input to the ANSYS simulation, and also serves as input to a comparison node. After the ANSYS simulation has been completed, the thermocouple data output serves as input to the Eich parameter transfer function. The output from the Eich parameter transfer function serves as the second input to the comparison node, where the output from the heat flux generator is compared to the Eich transfer function output and verified for accuracy. This closed loop method enables the Eich parameter transfer function to use training or regression to become more accurate, via the comparison node. Once the transfer function has reached optimum training, the NSTX Experiment box (flux generator + ANSYS Simulator) can be replaced with the real NSTX-U machine. Figure 4.2 displays the open loop flow chart after the NSTX Experiment box has been replaced by a real NSTX shot.

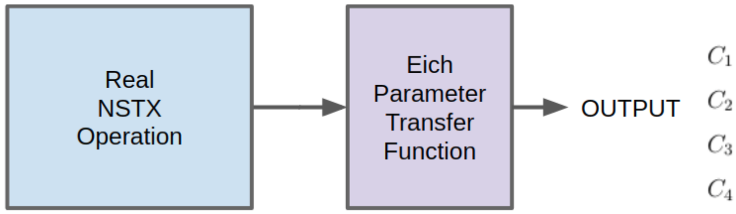


Figure 4.2: Open Loop Process Flow Chart

4.2 Trade Study

Extracting Eich parameters from subsurface thermocouple data can be accomplished with a myriad of different methods, each with its own respective advantages and disadvantages. As a means of selecting an Eich parameter transfer function, a trade study was completed in which various methods were investigated and compared. Generally, the methods fall into two categories. The first category involves employing an analytical solution to the heat diffusion equation, which eventually yields the heat flux profile. The engineer would then proceed to solve for the values of the Eich parameters from the profile derived analytically utilizing the simplified Eich model. The second category employs the power of statistics and machine learning, and involves utilizing a regression scheme to derive an effective function relating the thermocouple data to the Eich parameters. Obviously, these categories are not all encompassing nor are they mutually exclusive, but utilizing this discriminant can be beneficial for comparison.

Because the heat flux to the divertor is not a constant heat flux, the mathematical model that represents heat flow must be dependent upon time and space. The well known heat equation describes the time evolution of temperature within solids, and has been around for over a century. The heat diffusion equation can be derived via basic thermodynamics, and takes the form of a second order parabolic partial differential equation (PDE) of the form,

$$\frac{1}{\alpha}\nabla^2 T = \frac{\partial T}{\partial t} \tag{4.1}$$

where α is a constant corresponding to the thermal diffusivity of the material and ∇ represents the Laplace operator [22].

The analytical methods employed for solving this equation depend upon a series of assumptions, oftentimes beginning with the reduction to heat flow propagating in a single dimension, x, and that this PDE is separable. The elementary solution to the 1D heat diffusion equation is simply a Fourier series, whose Fourier coefficients and characteristic frequency must be derived. If the problem is more complicated, then oftentimes the semi-infinite solid model is used. The semi-infinite solid model assumes that the solid is infinite, yet has a single solid surface through which heat flow is possible in a single dimension. Additionally, the model requires that thermal conductivity, specific heat, and density, be homogeneous and isotropic throughout the medium [22, 23]. Given these assumptions, with a series of substitution and manipulations one arrives at the analytical solution equations,

$$\frac{T(x,t) - T_\infty}{T_{surface} - T_\infty} = \frac{2}{\sqrt{\pi}} \int_0^\eta \exp(-u^2)\mathrm{du} = \mathrm{erf}(\eta) \tag{4.2}$$

$$\eta = \frac{x}{\sqrt{4\alpha t}}, \tag{4.3}$$

where $T(x,t)$ is the temporally and spatially varying temperature throughout the medium, T_∞, is the temperature at an infinite distance from the surface, $T_{surface}$ is the surface temperature, and u is a dummy integration variable. As a means of deriving the heat flux at the divertor tile surface for NSTX-U, application of Fourier's law yields the final result,

$$q_{surface} = -k\nabla T(0,t) = \frac{k(T_{surface} - T_\infty)}{\sqrt{\pi\alpha t}} \tag{4.4}$$

where $q_{surface}$ is the surface heat flux, k is the thermal conductivity of the medium, and α is the thermal diffusivity. Once the heat flux above each thermocouple has been determined, a radial profile of time varying heat fluxes can be constructed by combining them. This profile can then be combined with the previously defined Eich model to generate equations for C_1, C_2, C_3, C_4. This method allows an explicit formulation of the Eich heat flux profile as a function of temperature inputs. There are many other analytical methods to solve the heat diffusion equation based upon various assumptions and boundary conditions, but a complete overview of all of these methods is outside the scope of this work. An industry standard text for heat conduction is Carslaw and Jaeger, *Heat Conduction in Solids*, which provides many of these additional solutions [23].

Most modern tokamaks utilize a combination of thermocouples, langmuir probes, IR thermography, and inverse solutions to the heat diffusion equation to calculate heat fluxes incident upon divertor tiles. Ideally, these diagnostic systems complement each other and provide a composite model of the heat flux reaching the divertor tile. Subsurface thermocouples have been used in several machines for offline reconstruction of the heat flux profile, including DIII-D and JET [19, 24, 25, 26, 27, 28, 29]. Early heat flux profile reconstructions were performed at JET by discharging an ELMy H-mode plasma and recording subsurface thermocouple data. The tiles cooled for approximately 1200 seconds between successive shots, and then an inverse solution was used to derive the heat flux [26]. This process was replicated at DIII-D, where a thermocouple was mounted to the divertor materials experimental station (DiMES). A discharge with fixed strike point was performed, and the semi-infinite heat diffusion equation solution was linearized and exploited to generate a surface heat flux from the thermocouple data [27]. Later, JET implemented a method in which the strike point did not need to remain in a constant position for the duration of the shot, but instead was swept at a slow rate (10-15 $\mathrm{mm\,s^{-1}}$) across the tile surface [24]. After the shot had ended, the surface heat fluxes were calculated via the inverse method and a regression algorithm fit the heat flux profile to finite element simulation results. During the DIID-D metal ring campaign a method called *thermocouple ELM heat flux deconvolution* was developed, in which subsurface thermocouples are utilized in conjunction with langmuir probes to resolve the total heat flux, as well as ELM heat flux, to the plasma facing components [29]. This method also employed the heat diffusion equation to resolve the surface heat fluxes.

As is evident from these examples, the utilization of thermocouples for heat flux extraction has been demonstrated and validated on several different machines. Common to all of these implementations is the utilization of the heat diffusion equation to extract the heat flux profile from subsurface temperature measurements. This enables a direct relationship to be formed between the temperature below the surface and the heat flux applied to the tile, yet it inherently enshrines all the assumptions made during the analytical solution process. These assumptions may have negligible impacts upon the final results with regard to the allowable engineering tolerances, yet they are not completely insignificant. No material is truly infinite, so the semi-infinite solution is a fundamentally flawed method to define a finite system. Modern graphite may be composed of nano-grains and strive for isotropic properties, but the thermal conductivity, specific heat, density, and thermal diffusivity, are all spatially variant. While the thermocouples are embedded close the tile surface and may warrant the single dimension assumption reasonable, it is technically incorrect.

The last and perhaps most significant downside to the aforementioned reconstruction methodology is the requirement that the plasma be constrained to a specific configuration and a specific location during the reconstruction discharge. Regardless of whether the fixed strike point or sweeping strike point method is used, both methods demand very articulate control over the plasma. If the plasma breaches this constrained configuration the results may become inapplicable. This method may be suited for general inquiries of the SOL plasma shape, but cannot be utilized as a tool for plasma control across the entire operational domain of the machine with high accuracy. For these reasons, it was decided to investigate an alternative to the heat equation inversion method, in order to complete a thorough trade study. It was discovered that modern machine learning statistical methods provided a secondary option that appeared impervious to the aforementioned assumptions and pitfalls.

Where analytical methods must adhere to the laws of physics to explicitly derive a solution, neural networks construct models adhering to the law of large numbers and the power of statistics. Chapter 2 provided an introduction to deep learning and neural networks. Training on hundreds or thousands of datapoints enables a neural network to learn from observation, in reflection of the abstract method in which the laws of physics were originally derived by the human brain. If ample data is available, machine learning algorithms are indeed capable of outperforming humans in a variety of tasks [14, 15, 16]. Perhaps one of the most promising facets of neural network algorithms is that they can be applied to almost any problem that can be framed with a sufficient dataset. This feature makes neural networks highly adaptive; they can learn *anything* about any type of system given sufficient data. Furthermore, once a single robust neural network architecture has been developed, it can be applied to many classes of diagnostics with little redesign. Deep learning neural networks provide a modular, adaptive, and powerful method for modeling complex, high-dimensional, nonlinear systems for which analytical representation is difficult or impossible.

All that being said, neural networks are ultimately only capable of being applied in methods conceived by humans, and can reflect human bias. If the dataset a neural network is trained on is not a true statistical representation of the entire population of data, the neural network will learn this bias as truth. In the NSTX-U simulator case, it is important to consider the assumptions made by every sub-process in the entire closed loop system, including the heat flux generator and the ANSYS simulator. Additionally, because modern neural networks can consist of millions of interconnections between perceptrons, if the system is biased it is very difficult to troubleshoot, if not impossible. These deep learning algorithms are oftentimes a black box, and while their success is extremely impressive, there is no way to concisely describe what happens inside the box to perfectly tune the system. It is, therefore, imperative that the engineer develop methods for validating the system performance in the entire domain of operation before considering the system ready. Understanding where the system fails and where it thrives is of critical importance.

A review of a few examples of neural networks applied to nuclear fusion diagnostics is useful here. The first successful implementation of neural networks as a means of plasma control occured on the Tokamak COMPASS in the early 1990's [30]. Researchers at COMPASS trained a multilayer perceptron network with simulation data generated by a numerical Grad Shafranov equation solver, and then deployed the neural net as a real time plasma control system actuating plasma shaping coils. Similarly, researchers at the Reversed Field eXperiment (RFX) used a multilayer perceptron network to interpret langmuir probe signals after training on simulated data, and reported a factor of 30 improvement in CPU processing time over traditional fitting methods [31]. More recently, researchers at JET have utilized a modified convolutional neural network architecture to produce a pixel by pixel tomographic reconstruction of bolometer data [32]. In each of these examples, researchers reported significant improvements in computational speed, while achieving high accuracy reconstructions.

Performance Variables	Weight	Heat Diffusion EQ Score	Heat Diffusion EQ S*W	Machine Learning Score	Machine Learning S*W
Requires Minimal Assumptions	15.00%	50.00	7.5	90.00	13.5
Requires Small / Sparse Dataset	10.00%	80.00	8	20.00	2
Provides Intuitive Insight	15.00%	90.00	13.5	40.00	6
Minimal Training	10.00%	70.00	7	40.00	4
Can Be Expanded to More Complex Problems	15.00%	30.00	4.5	90.00	13.5
Can Be Expanded to other Scientific Domains	10.00%	40.00	4	80.00	8
Potential to be Utilized in Real Time Systems (<1ms)	15.00%	60.00	9	95.00	14.25
Novel Approach to Model Reconstruction	10.00%	20.00	2	95.00	9.5
Total			55.5		70.75

Figure 4.3: Trade Study Matrix

In order to compare the heat equation and machine learning reconstruction techniques in a clear and concise manner, a simple trade study matrix was developed and appears in figure 4.3. This matrix scores the heat diffusion equation method and machine learning algorithms with respect to several of the performance variables previously discussed. Each performance variable is weighted based upon its importance to the project objectives, and upon the larger objectives of NSTX-U and the University of Tennessee Nuclear Fusion Team. The performance variables that are higher priority have been given a weight of 15% while the secondary performance variables only carry 10% weight each. In the score columns, both techniques are rated based upon their respective performance. In alignment with what has already been discussed in this section, the heat diffusion equation excels at requiring a small(er) dataset, providing intuitive insight into the nature of the problem, and being implemented efficiently with minimal training. On the other hand, machine learning algorithms require far fewer assumptions about the model, scale easily to new problems or further constraints imposed on the existing model, can be expanded easily to other diagnostics and scientific domains, offer the capability for sub-millisecond real time control system integration (after training, obviously), and enable an investigation into a novel heat flux reconstruction algorithm. Because machine learning scored 70.75 compared to the heat diffusion equation's score of 55.5, machine learning was selected as the final choice for the Eich parameter reconstruction method.

4.3 Tools for CNN Implementation

All the recent hype surrounding deep learning has resulted in a torrent of open source software for machine learning implementation becoming available to the public. There are many packages available including Caffe, Theano, PyTorch, Keras, etc. (it is a long list). The most popular software toolkit for implementing deep learning algorithms is called TensorFlow. TensorFlow is the brain child of the Google Brain team, a research and development team working on artificial intelligence and machine learning systems. Originally, TensorFlow was a proprietary package and only available to Google employees, but in 2015 the software was released as open source under the Apache 2.0 License. Google describes the software as a high performance dataflow application programming interface (API) for numerical computation and machine learning. Users may choose APIs in python or C++, and extensive tutorials have been published on the TensorFlow website. Additionally, the API consists of multiple installation options for running on CPUs, GPUs, or TPUs (Google's proprietary hardware optimized for TensorFlow). TensorFlow was chosen as the deep learning package for NSTX-U because of its simple interface, its performance capabilities, and the massive amount of support and documentation available online.

In order to successfully implement a deep learning algorithm for the NSTX-U Eich reconstruction problem, it was necessary to become familiarized with the process of coding with the TensorFlow package on much simpler problems. The TensorFlow tutorials provided a guide to this learning process, which was admittedly steep. Four of the TensorFlow tutorials were completed. The first tutorial was a basic linear regression scheme utilizing a simple neural network. Next, an iris classification convolutional neural network trained a CNN to recognize four different types of iris flowers in images. The third tutorial completed was the classic Modified National Institute of Standards and Technology (MNIST) handwritten digit classification problem. In this problem, a CNN is trained to recognize handwritten digits from 28X28 pixel images. Finally, a Recurrent Neural Network (RNN), which is a neural network that includes hysteresis via a feed-back loop, was trained to do basic language processing. After these tutorials were completed, a basic understanding of the functionality of TensorFlow had been developed, and it was possible to attempt the more complicated Eich reconstruction problem.

The entirety of the CNN thermocouple CNN was developed and deployed on a `x86_64` Ubuntu Linux 18.04 operating system (OS). This OS ran on a 16 core Intel(R) Xeon(R) E5-2670 Centra CPU running at 2.60GHz. The workstation has 32GB of RAM, and a 500GB Solid State Hard Drive. An advantage to utilizing TensorFlow is that multi-threading is handled by the API so there is no need to code posix threads or anything of that nature. On average, during training the CPU would load all 16 cores to approximately 25% of capacity. It was therefore possible to train three neural networks simultaneously without overloading the micro-controller. That being said, utilizing TensorFlow on a cluster of GPUs, or via a distributed parallel cloud network could decrease training time.

For all deep learning development in this project, Python3 was the language of choice. In addition to the TensorFlow software module, several other open source python modules were utilized. The **os** module was used for interacting with the Linux operating system. Both **matplotlib** and **GNUplot** were utilized to generate figures. The **datetime** and **time** modules were utilized for system clocking and performance metrics. Lastly, the **numpy** and **csv** modules were used for numerical calculation and data handling, respectively. All of the packages employed in the CNN development are open source, and available on most operating systems.

4.4 CNN Architecture

To reconstruct Eich parameters a convolutional neural network architecture was the method of choice, although other deep learning algorithms would likely succeed as well. Chapter 2 provides an introduction to neural networks and convolutional neural networks, and provides several examples of traditional applications for CNNs. Convolutional neural networks are excellent at image processing, and rival human performance in some cases. For the Eich reconstruction problem, the input data does not come in the form of an image, but instead in the form of the following parameters:

- B_p Poloidal Magnetic Field Magnitude;

- P_{heat} Power entering SOL;
- R_0 Strike Point Initial Position;
- f_x Flux Expansion Coefficient;
- TC 2D Array of Thermocople Data.

The biggest development challenge with this deep learning algorithm was determining the best method for arranging / organizing the input data. Ultimately, the decision to utilize a CNN rather than a simple deep neural network was based upon the observation that the thermocouple data, TC, came in the form of a 2-dimensional array, which could be interpreted as a *thermocouple image* to the neural network.

As previously discussed, in the conventional heat diffusion equation inverse method deriving a heat flux from the thermocouple data required finding an analytical solution and extracting the heat flux from that solution. In contrast, applying a CNN to the thermocouple data extracts characteristic features, or motifs, associated with a specific heat flux profile, and forges stronger inter-perceptron connections as a means of iteratively finding a global minimum in the error function. The CNN interprets the 2D array of the thermocouple data in the same way it interprets an array of pixels that compose an image. Where a regular image consists of two spatial dimensions observed at a single instant in time, the thermocouple image consists of one spatial dimension and one temporal dimension. Figure 4.4 illustrates an example of perceiving thermocouple data as an image. In the figure, the x coordinate represents the thermocouple number, increasing in the radial direction. The y coordinate represents temperature. The color gradient represents the flow of time. While 4.4 is a plot with no real relationship to the CNN, it is useful to help perceive the 2D thermocouple array as an image, much like the CNN does. Figure 4.5 compares an example thermocouple array to the array of pixels of a handwritten digit.

Each thermocouple array was therefore treated as an image would be treated in a typical CNN. The array was fed into a series of convolution + pooling layers, in which it was cross correlated with learned temperature motifs, then down sampled. Ultimately, the 2D array became a 1D vector that served as input to a fully connected layer.

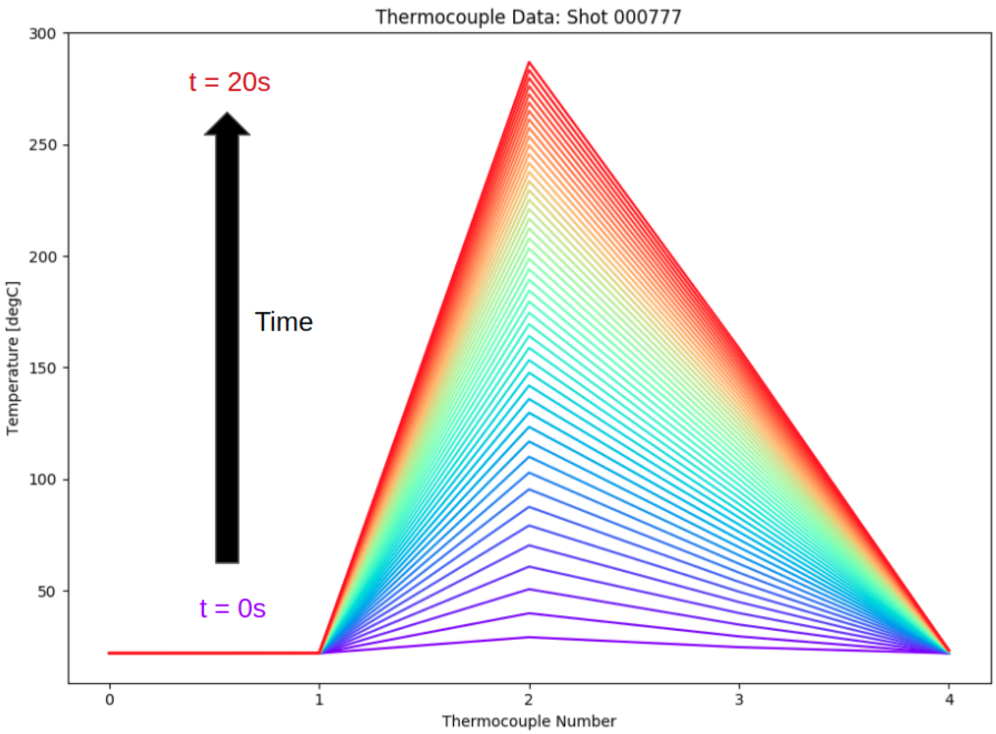
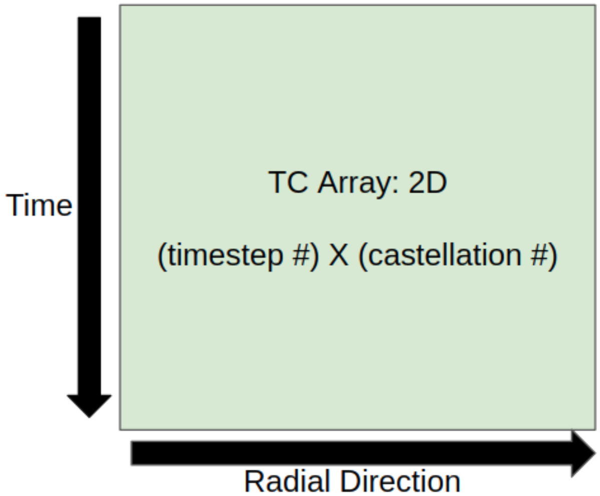


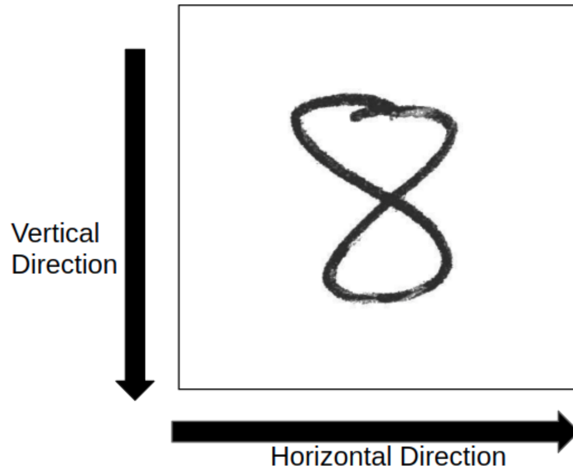
Figure 4.4: Visual Example of Thermocouple Data as an Image

When constructing neural networks, the engineer has the capability of tuning the network by changing specific architecture variables called hyperparameters. The stride size, number of convolution + pooling layers, number of feature maps, and neurons per fully connected layer, are all hyperparameters. Tuning these hyperparameters is an experimental process, much like tuning a PID controller in control system design without state space representations. Usually, for any given hyperparameter, the method was to start at an excessive level and increment or decrement the variable until a significant negative impact was observed with regard to the system performance. Once a negative impact was observed, the hyperparameter was restored to the level just before the degradation in performance occurred. There are many other hyperparameters that were not mentioned here, but the tuning process is similar each of them.

Several revisions were necessary to identify the appropriate method for combining the 2D thermocouple array with the scalar machine spec inputs, $B_p, P_{heat}, f_x, R_0,$. Originally, these scalars were appended to the TC array and fed through the convolutional layers with the thermocouple image. This method never yielded accurate results, primarily because



(a) Example Thermocouple Array



(b) Example Handwritten Digit

Figure 4.5: TC Array vs Handwritten Digit

embedding the machine specs in the thermocouple image resulted in invalid temperature feature identification by the cross correlation process. After some time, it was determined that appending the scalar machine specs to the thermocouple image was unnecessary. Instead, these values were concatenated with the final 1D temperature vector that was produced by the convolution layers, and this appended vector was fed to the fully connected layers. This enabled the 2D thermocouple array to be processed as an image, and the scalar machine specs to be processed correctly as scalar inputs. Figure 4.6 provides a visual representation of this CNN architecture. Two convolution + pooling layers, each with 16 feature maps, were deemed sufficient for recognizing characteristics in the thermocouple images. Four fully connected layers consisting of 32 parallel perceptrons are connected serially and output to a 4 X 1 Eich parameter vector. As the figure indicates, the machine specs and temperature data are concatenated just before the first full width fully connected layer.

The original CNN architecture used the raw temperature data and machine specs as inputs, but this made the neural network slow to train. Because the temperature values sometimes reached several hundred degrees Celsius, while the machine specs were much smaller, the raw temperature data was effectively weighted as more significant than the smaller machine specs. In an effort to equalize all inputs, a normalization process was

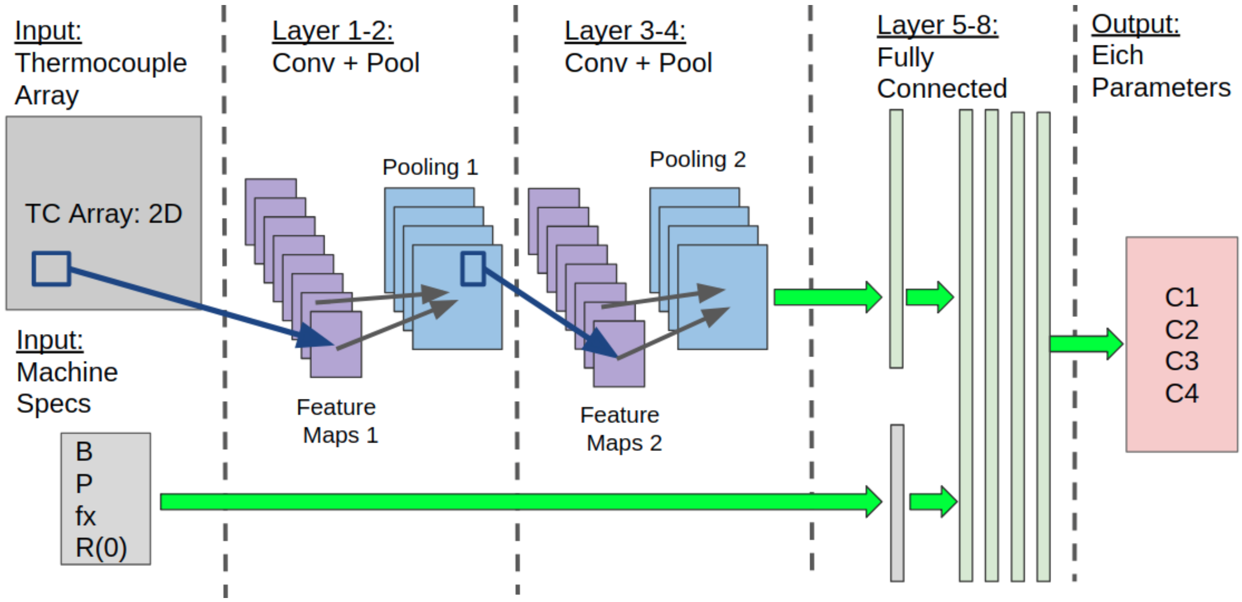


Figure 4.6: CNN Architecture June 2018

necessary. The values for each input variable were linearly mapped between -1 and 1. In other words, the -1 and 1 values correspond to the minimum and maximum datapoints in the dataset, respectively, and the data is normalized between these values. Mathematically, this mapping can be described by the equation,

$$x_{normalized} = \frac{2}{x_{max} - x_{min}}(x - x_{min}) - 1 \tag{4.5}$$

where $x_{normalized}$ is the normalized variable, x_{max} and x_{min} are the maximum and minimum values of that variable in the entire dataset, and x is the un-normalized variable. Scaling the inputs between -1 and 1 instead of between 0 and 1 gives the neural network the ability to optimize itself in two directions. Technically, the neural network should be capable of performing this normalization on its own via the adjustment of weights, but performing the normalization manually decreases the required training time. After this data normalization process was developed, the neural network rate of accuracy increase improved dramatically. It should be mentioned that when a neural network is normalized with respect to the domain of a specific dataset, then trained on that dataset, predictions made on a different dataset should be normalized to the same domain as the training dataset.

It was not uncommon for the training process for this CNN to last 24-36 hours running on the aforementioned workstation. That being said that amount of time is insignificant when one considers the millennia of mathematical formulation needed before analytical solutions to the heat equation were possible. The metric utilized for training was a scaled absolute error. The mean of this absolute error was taken across all Eich parameters, C_1, C_2, C_3, C_4, and the reduced mean was the value utilized in error backpropagation. Mathematically, this error function can be described by the equation,

$$Error = \frac{\alpha}{4} \sum_{i=1}^{4} |C_i - target_i| \tag{4.6}$$

where C_i represents the Eich parameters predicted by the neural network, $target_i$ represents the true Eich parameter, and α represents a scaling hyperparameter that can be tuned for performance. Originally, basic gradient descent was employed for error backpropagation. Using such a basic optimization algorithm proved incredibly slow, and the network became stuck in local minimums. To improve the speed, basic gradient descent was replaced with stochastic gradient descent (SGD). Rather than iteratively backpropagating error from each training example, SGD samples from the training dataset and backpropagates error with a reduced set. This enables the CNN to optimize itself much more quickly, yet in this case the CNN was still getting trapped in local minimum, where it would spend days hovering at a constant accuracy level without improvement. To combat both the slow training time and the local minimum problem, the Adam Optimizer was utilized. Rather than maintaining a fixed learning rate as the neural network descends the gradient, Adam Optimizer dynamically adjusts the learning rate during the descent by computing the first and second moments of the gradient [33]. Integrating the Adam Optimizer in TensorFlow required a single function call, greatly reduced the training time, and enabled the neural network to achieve high accuracy during training.

Because the Eich reconstruction problem is not a classification problem, it was necessary to create a definition for accuracy. In a classification problem, the accuracy of the network can be calculated by directly counting the number of correct predictions. In this case, the predictions are continuous, and it is unlikely that the neural network will predict the correct Eich parameter to 32 bits of precision. It was therefore necessary to define the accuracy with regards to an allowable tolerance in the prediction error. A tolerance of 5% was originally chosen as the threshold for a correct prediction; if the neural network predicted value was within 5% of the true value, it was considered correct. This tolerance was another hyperparameter that could be tuned to optimize performance, and it was regularly updated (between 0.5% to 5.0%) during the CNN development process.

In total, approximately 8500 NSTX-U discharges were generated by the Monte Carlo Flux Generator and by the ANSYS simulator. As is common practice in Machine Learning, the dataset was divided into three parts: the training set, the test set, and the publication set. The training set is utilized for backpropagation and optimizing the CNN, the test set is used for validation as the network is training (but is not backpropagated), and the publication set is reserved for generating final results. For the final CNN revisions, the data was partitioned as follows:

- Training Set: 7200;
- Test Set: 800;
- Publication Set: 500.

Traditionally, the test set consists of approximately 20% of the entire dataset, but because this dataset is large, 800 discharges sufficed. While training, it was convenient to print statistics to the screen at regular intervals as a means of following the progress of the network. The accuracy with respect to epoch was a useful metric to follow. A boxcar convolution running average filter was employed to print accuracy results. The length of the boxcar window was another hyperparameter, but was generally set to average the accuracy over the past 100 epochs. This running average was typically printed to the screen every 1000-5000 epochs, depending on the training session. The accuracy value for the Eich parameters was also recorded in an array every few thousand epochs for plotting. Figure 4.7 is an example of such a plot, and provides the accuracy as a function of training epoch for an early training session. In addition to the training data accuracy, the test data accuracy is plotted in this figure. As can be observed, the CNN performs slightly better on the training data than

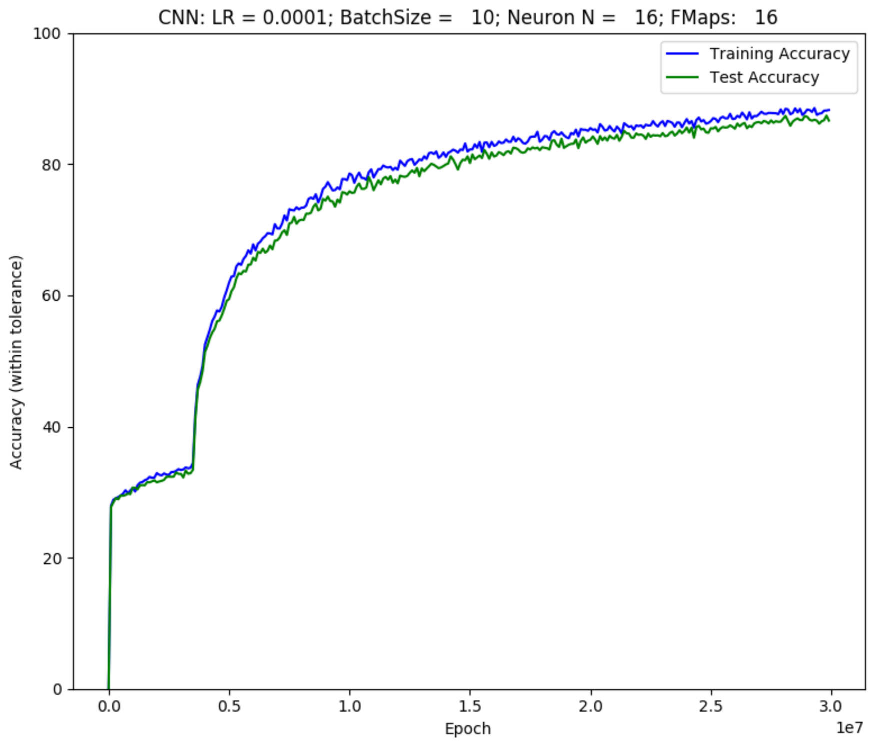


Figure 4.7: Accuracy as a Function of Epoch for an Early Revision CNN

it performs on the test data. This minor divergence between the two datasets is because the CNN is superior at predicting the results to data that it has learned (backpropagated). Furthermore, the fact that these two datasets produce very similar curves (a few % difference) indicate that the test dataset is adequately represented by the training dataset.

TensorFlow allows users to save and load the entire model, including the weighting matrix, so that the CNN does not need to be trained for each use. After the CNN was trained to sufficient accuracy, the entire model was saved into a directory for future use. When it was time to run the CNN on the publication dataset, the model would be loaded from the directory into a predictor CNN. The predictor CNN is a forward propagation only neural network; it does not backpropagate error and train. The engineer serves the predictor CNN a single randomly chosen NSTX-U discharge from the publication dataset, and the CNN outputs a prediction for the Eich parameters, C_1, C_2, C_3, C_4. After the CNN has been trained this process is extremely fast, with forward propagation times of approximately a few milliseconds.

One valuable lesson gained from the early CNN architectures was the importance of comprehending the mathematical degeneracies, boundaries, and limitations of the model the CNN is designed to approximate. Early CNN revisions would train to very high (93%) accuracy on the training data, but then fail intermittently on the publication data. This failure was due to an oversight with regards to degeneracies associated with the simplified Eich model. Only after understanding the degeneracy between Eich parameters in the Eich model was it possible to construct a final CNN architecture that performed with high accuracy on the publication dataset. Section 4.5 highlights these degeneracies, how they were discovered, and how they were resolved.

4.5 Degeneracies

CNN revisions that reflected the architecture discussed in the previous section (figure 4.6) often trained to high accuracy (93%) and then would intermittently perform poorly on the publication dataset. After extensive hyperparameter adjustment, increasing the training time to 20 million epochs, implementing various input data normalization algorithms, and a series of Mayan sacrifices, no improvement was observed. As a means of obtaining a better grasp on the problem, the predictor CNN was run iteratively, and statistics were generated to observe the nature of the prediction distributions. Figure 4.8 provides an example of one such distribution. Ten NSTX-U discharges were fed to the predictor CNN, which predicted ten sets of Eich parameters, C_1, C_2, C_3, C_4. In the figure, this distribution (or spread) is plotted for each Eich parameter. Additionally the standard deviation for each Eich parameter (shaded grey and gold regions), the mean (location where grey and gold meet), and the true Eich parameters (red dotted line) are overlaid. As the figure indicates, the predictor achieved remarkably good predictions for C_1 and C_3, and the standard deviations for these parameters are small. For C_2 and C_4 however, the prediction distribution is spread fairly evenly across the entire domain of these variables ($1.0 < C_2 < 2.5$; $-1.4 < C_4 < -0.5$), indicating that no valid transfer function has been learned by the CNN. The reason for the poor prediction performance becomes clear if one considers the fundamental equations that

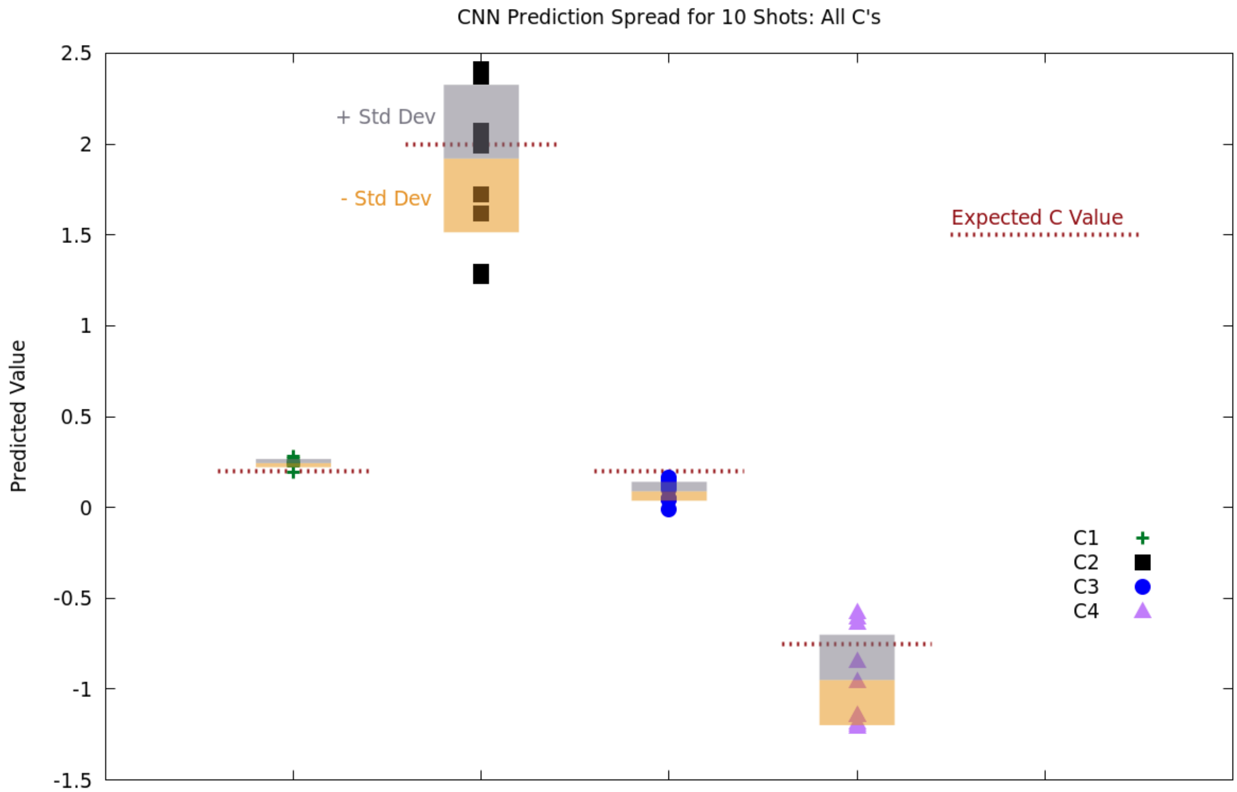


Figure 4.8: Predictor CNN Degeneracies

define the simplified Eich model, as defined in chapter 3,

$$\lambda_q = C_2 P_{heat}^{C_3} B_p^{C_4}$$

$$S = C_1 \lambda_q \ .$$

$$x_p = S f_x$$

$$x_c = \lambda_q S$$

Clearly, there are multiple degeneracies between C_2, C_3, C_4, because a single value of λ_q can be constructed with a variety of combinations of C_2, C_3, C_4. In other words, the same characteristic decay length, λ_q, can be generated by multiple combinations of Eich parameters. There is no degeneracy associated with B_p or P_{heat} because those machine specs are inputs to the neural network, whereas the Eich parameters are not. For a given NSTX-U shot, the CNN is provided a fixed value of B_p and P_{heat} and must reconstruct C_1, C_2, C_3, C_4. From this observation, it can be concluded that it is impossible for the neural network to reconstruct all four Eich parameters from a single shot. Furthermore, it was observed that the neural network can resolve S and λ_q from a single shot, as indicated by figure 4.9.

Because with a single shot there is an obvious degeneracy between two variables, C_2 and C_4, increasing the number of shots was investigated as a means of properly constraining this system. It was determined that by serving three NSTX-U discharges to the CNN, each with the same Eich parameters but varying machine specs, it would be possible to resolve C_1, C_2, C_3, C_4, simultaneously. Consider a neural network that has been trained to predict S and λ_q only, rather than C_1, C_2, C_3, C_4. Three NSTX-U discharges are created by the flux generator that share common Eich parameters but allow the machine specifications to be chosen randomly from a flat distribution. These three shots are then input sequentially to the CNN, and the CNN makes three corresponding predictions for S and λ_q. From these three sets of S and λ_q it is possible to predict all three Eich parameters. This can be described mathematically by the following system of equations,

$$ {}^{A}\lambda_q = C_2({}^{A}P_{heat}^{C_3})({}^{A}B_p^{C_4}) \tag{4.7} $$

$$ {}^{B}\lambda_q = C_2({}^{B}P_{heat}^{C_3})({}^{B}B_p^{C_4}) \tag{4.8} $$

$$ {}^{C}\lambda_q = C_2({}^{C}P_{heat}^{C_3})({}^{C}B_p^{C_4}) \tag{4.9} $$

where A, B, C correspond to different NSTX-U shots, and the left superscripts indicate the machine spec for that respective shot. This is a nonlinear (linear in log-space) system of equations. If the CNN correctly predicts ${}^{A}\lambda_q, {}^{B}\lambda_q$, and ${}^{C}\lambda_q$, and all values of B_p, and P_{heat} are known inputs, then this system of three equations consists of three unknowns C_2, C_3, C_4, and is fully constrained.

Lastly, observe that for any single shot C_1 can be predicted via the relationship between S, and λ_q,

$$ C_1 = \frac{S}{\lambda_q} \quad . \tag{4.10} $$

Given these four equations and four unknowns, a CNN scheme in which three shots with common Eich parameters are inputs would be fully constrained and a reconstruction of

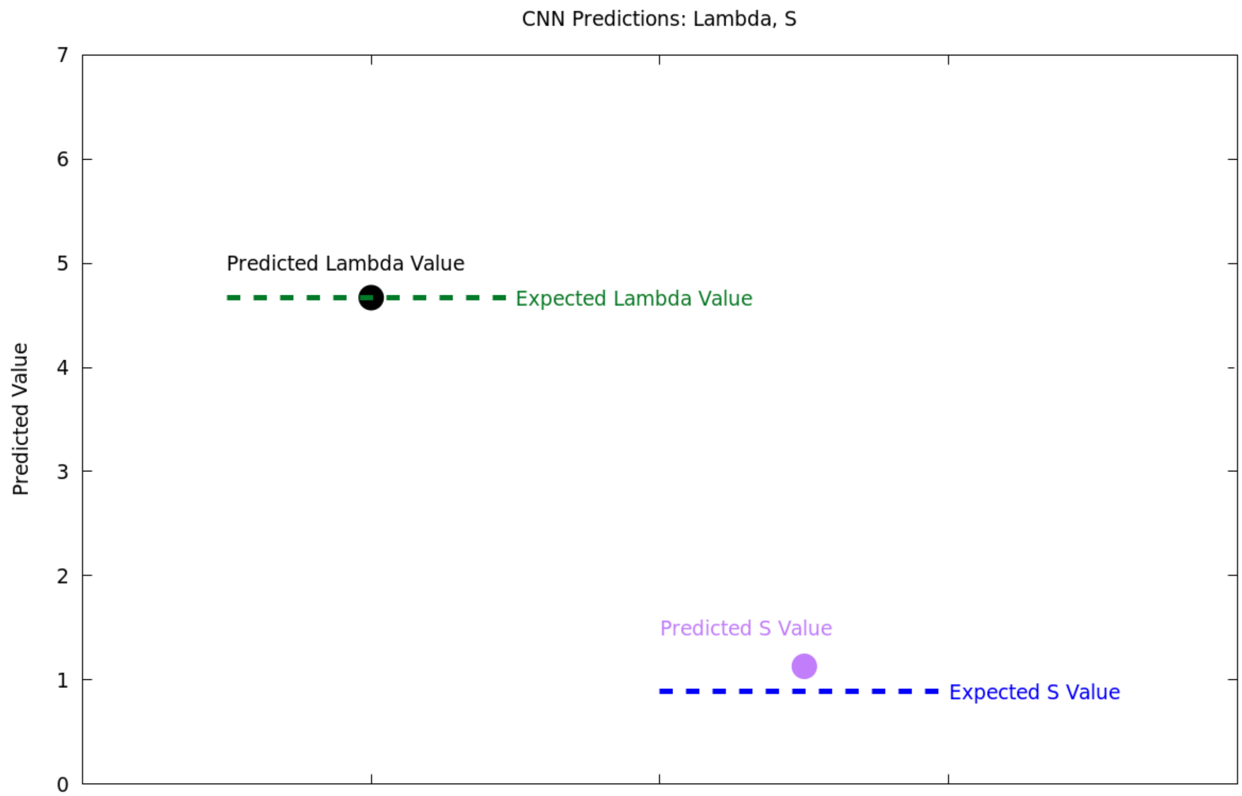


Figure 4.9: Predictions for S and λ_q from a single shot are possible

C_1, C_2, C_3, C_4, is possible. Section 4.6 discusses the CNN architecture utilized to generate high accuracy predictions for the Eich parameters.

4.6 The Final NSTX-U Thermocouple CNN

The final NSTX-U Thermocouple CNN was developed in response to the degeneracy issue discussed in section 4.5. This final revision accepts three sequential NSTX-U shots (and accompanying machine specs) and outputs correct values for λ_q and S. In order to extract the values of C_2, C_3, C_4, from the system of nonlinear equations derived in 4.5, it was necessary to solve the system after all λ_q and S were predicted by the neural network. One method for solving systems of nonlinear equations that has stood the test of time is Newton's method. Newton's method, a root finding algorithm, iteratively approximates the zeroes of a function, until it converges to within a specified tolerance. For a single variable, Newton's method calculates the next approximation of a root of the function with the following formulation,

$$x_{n+1} = x_n + \frac{f(x_n)}{f'(x_n)} \tag{4.11}$$

where x_{n+1} represents the next approximation and x_n represents the previous approximation to a root of the function, $f(x)$, and $f'(x)$ represents the derivative of $f(x)$ with respect to x. Initially, a value for x_n is guessed, and then subsequent approximations, x_{n+1}, are developed iteratively until the next approximations converge to within an allowable tolerance, ϵ, as described by the following algorithm:

Algorithm 1: Newton's Method

```
x_n = initial guess
ε = some error threshold
error = some value larger than ε
while error > ε do
    x_{n+1} = x_n + f(x_n)/f'(x_n)
    error = |x_{n+1} − x_n|
```

This algorithm can be extended to multivariable systems by writing the algorithm in matrix form. Given three shots with the same Eich parameters that generate the following system of equations,

$$\begin{aligned}
{}^{A}\lambda_q &= C_2({}^{A}P_{heat}^{C_3})({}^{A}B_p^{C_4}) \\
{}^{B}\lambda_q &= C_2({}^{B}P_{heat}^{C_3})({}^{B}B_p^{C_4}) \\
{}^{C}\lambda_q &= C_2({}^{C}P_{heat}^{C_3})({}^{C}B_p^{C_4}) \ ,
\end{aligned}$$

it is possible to arrange these equations into a matrix, $\mathbf{F}$, such that:

$$\mathbf{F}(\mathbf{C}) = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} C_2({}^{A}P_{heat}^{C_3})({}^{A}B_p^{C_4}) \\ C_2({}^{B}P_{heat}^{C_3})({}^{B}B_p^{C_4}) \\ C_2({}^{C}P_{heat}^{C_3})({}^{C}B_p^{C_4}) \end{bmatrix} \qquad \text{where } \mathbf{C} = \begin{bmatrix} C_2 \\ C_3 \\ C_4 \end{bmatrix}$$

Constructing the the Jacobian, $\mathbf{J}$ of this matrix can be achieved by taking the partial derivatives of $\mathbf{F}$, with respect to each Eich parameter,

$$\mathbf{J}_{ij} = \frac{\partial f_i}{\partial C_j} = \begin{bmatrix} \frac{\partial f_1}{\partial C_2} & \frac{\partial f_1}{\partial C_3} & \frac{\partial f_1}{\partial C_4} \\ \frac{\partial f_2}{\partial C_2} & \frac{\partial f_2}{\partial C_3} & \frac{\partial f_2}{\partial C_4} \\ \frac{\partial f_3}{\partial C_2} & \frac{\partial f_3}{\partial C_3} & \frac{\partial f_3}{\partial C_4} \end{bmatrix}$$

The matrix corollary to equation 4.11 then becomes,

$$\mathbf{C}_{n+1} = \mathbf{C}_n - \mathbf{J}_n^{-1}\mathbf{F}_n \tag{4.12}$$

where the subscript $n+1$ represents the next approximation and the subscript n represents the previous approximation. Successive approximations of the correct Eich Parameters, C_2, C_3, C_4, can then be generated by the algorithm:

Algorithm 2: Newton's Method: Multivariate

$\mathbf{C}_n =$ initial guess
$\epsilon =$ some error threshold
$error =$ some value larger than ϵ
while $(error > \epsilon)$ **do**
  $\mathbf{C}_{n+1} = \mathbf{C}_n - \mathbf{J}_n^{-1}\mathbf{F}_n$
  $error = \mathrm{sum}(\mathbf{C}_{n+1} - \mathbf{C}_n)$

Figure 4.10 provides an illustration of the final NSTX-U CNN architecture. The 2D thermocouple data serves as the input to a series of two convolution + pooling layers, after which a 1D vector emerges and is concatenated with the machine specs. This concatenated vector is fed to four fully connected layers, whose output is λ_q and S. This process completes three times for three different NSTX-U shots that all share common Eich parameters but have different machine specs. After the three NSTX shots have been forward propagated through the network, they are fed to a Newton's Method algorithm, which solves for the values of C_1, C_2, C_3, C_4.

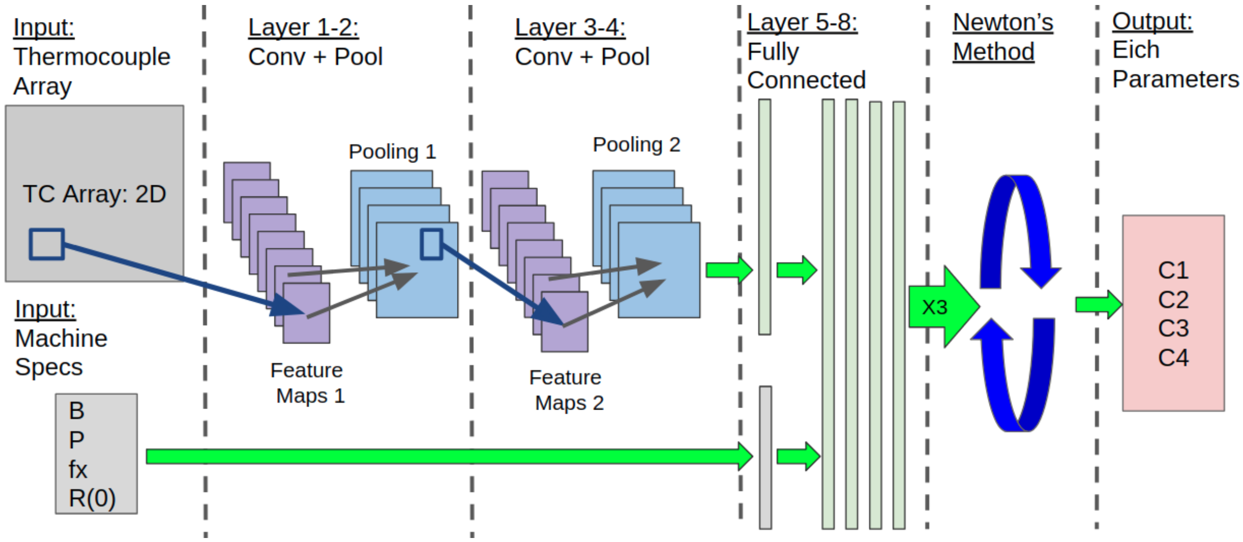
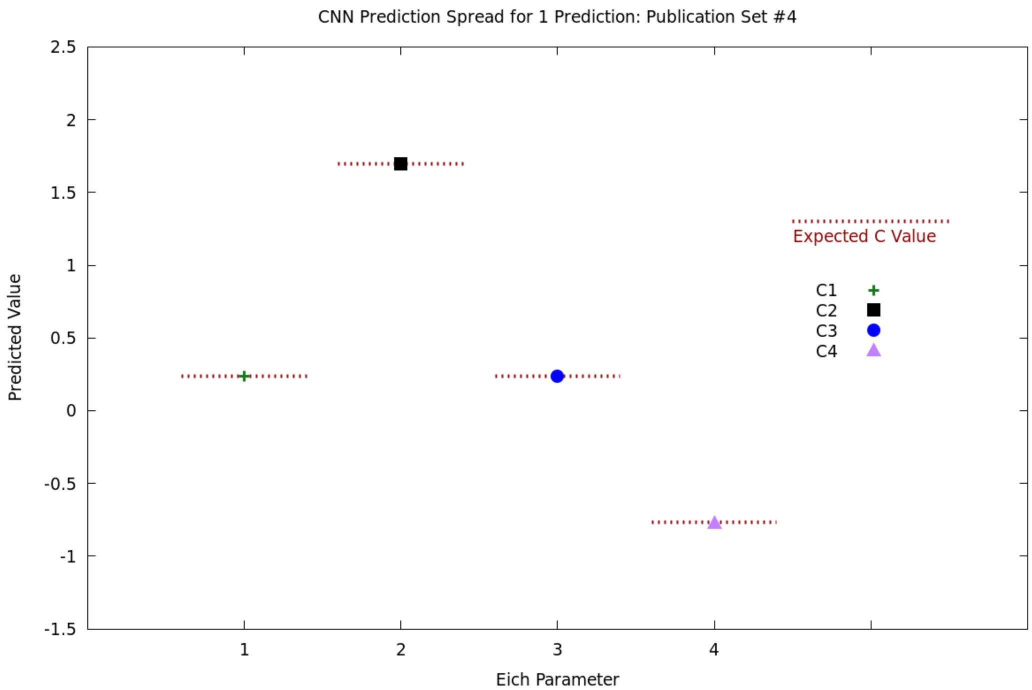


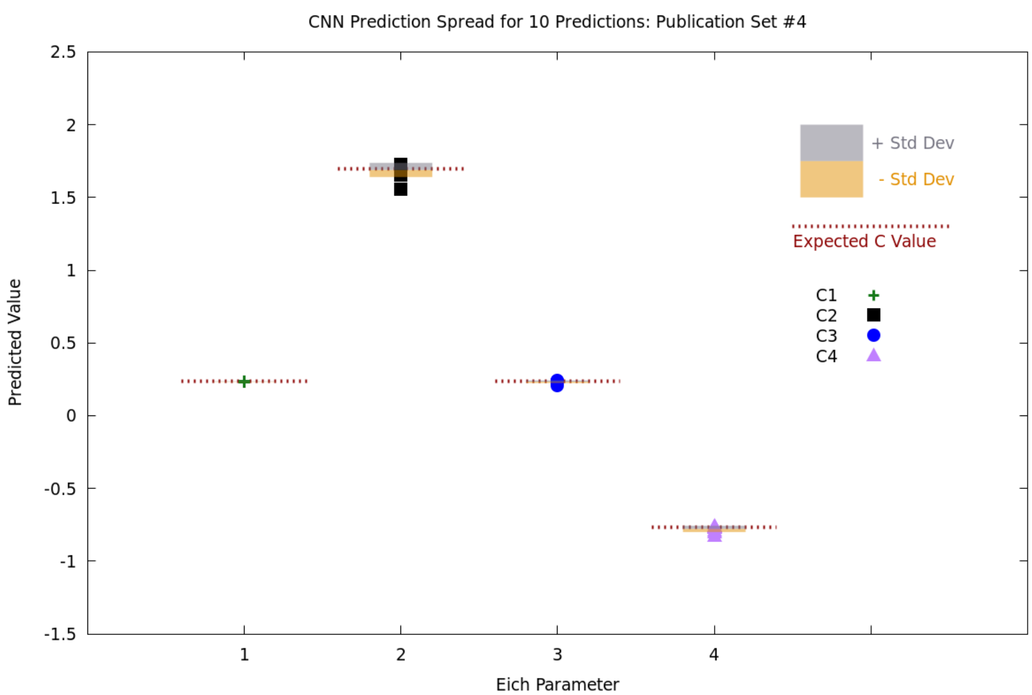
Figure 4.10: Final CNN Architecture

This new architecture increased the prediction time significantly when compared to the time the CNN takes to make a single prediction. If this system were to be applied to a real time system, replacing Newton's method with a direct matrix decomposition in log space could eliminate the need for recursive root finding, and solve the system of equations in a single pass, thereby decreasing the CPU time significantly. That being said, the current prediction time is usually less than 1-2 seconds, so Newton's method more than suffices for this generic application. An implementation that uses direct matrix inversion to derive the Eich parameters was also tested and it was verified that the results are identical to Newton's method, with sub-second prediction speeds.

To test this new architecture, five new publication heat fluxes were created by the Monte Carlo heat flux generator, each consisting of 100 shots of common Eich parameters but varying machine specifications. These heat fluxes were pushed through the ANSYS simulation, resulting in new publication thermocouple datasets, which became inputs to the CNN. The Eich parameters plotted in figure 4.11a were derived using the new architecture from three NSTX simulated shots with constant Eich parameters. The figure demonstrates the accuracy that can be obtained from this CNN reconstruction system. The CNN utilized to perform these predictions was trained to 95% accuracy where an accurate prediction is defined as within 0.5% of range of predicted variable (S, λ_q). Each Eich parameter, C_1, C_2, C_3, C_4, is plotted alongside the C value that was expected for that respective



(a)



(b)

Figure 4.11: Final CNN prediction results for (a) 1 set of 3 shots and (b) 10 sets of 3 shots. CNN trained to 95% accuracy where an accurate prediction is defined as within 0.5% of range of predicted variable (S, λ_q)

parameter. In order to generate a distribution of predictions that may be compared against 4.8, 10 sets of simulated NSTX discharges were served to the CNN, and the mean and standard deviation were calculated. Figure 4.11b illustrates the prediction distribution for the new CNN architecture. The mean of the distribution for each respective Eich parameter is extremely close to the expected value, and the standard deviation (grey and gold bars) are very tightly clustered about the mean. When compared to 4.8, figure 4.11b indicates that the degeneracies have been greatly mitigated.

Table 4.1 provides the statistical results from all five publication datasets. Each publication dataset was generated independently, and consists of 100 simulated NSTX shots. The Eich parameters are held constant for all 100 shots in each respective publication dataset, but the Eich parameters differ between publication datasets. As can be observed in the table, the CNN is accurate in predicting all four Eich parameters, regardless of the dataset. It can therefore be concluded that given three NSTX-U discharges in which the Eich parameters (C_1, C_2, C_3, C_4) are fixed but the machine specifications $(B_p, P_{heat}, f_x,$, etc.) are varying, it is possible to reconstruct the Simplified Eich heat flux profile.

To further increase the accuracy and reduce the standard deviation of the predictions, several measures could be taken with regards to the training. The obvious method to increase the accuracy of the CNN is to allow it to train longer on the training data. This would result in a higher accuracy level overall, but comes at the cost of training time. On this CNN, approximately 1-5 million epochs of training would most likely increase the accuracy to approximately 98% (from 95% where error threshold is 0.5% of variable range). Further training beyond 98% - 99% is not recommended, as the CNN can overtrain and any further increases on training data accuracy will result in decreases in publication dataset accuracy (the CNN learns the test dataset, not the generic model), but there is significant room to improve before this overtraining regime is reached. Lastly, experimentation with tuning hyperparameters may yield gains in performance. More specifically, increasing the width and depth of the CNN may increase accuracy at the expense of increased training time.

It is evident that there are many options for optimizing this CNN and constructing an extremely accurate system, but for the purposes of this project, the aforementioned results suffice. Having generated a reconstruction method that is capable of deriving unknown Eich

Table 4.1: *Statistical results from five publication datasets. Each publication dataset consist of 100 NSTX-U shots with common Eich Parameters, but the Eich parameters differ between different publication datasets. CNN trained to 95% accuracy where an accurate prediction is defined as within 0.5% of predicted variable (S, λ_q) with respect to variable domain. μ = Mean; σ = Standard Deviation*

			Number of Predictions (3 shots each)			
			1 Prediction		10 Predictions	
Publication Dataset	Eich Parameter	Expected Value	μ	σ	μ	σ
1	C_1	0.159	0.153	0	0.157	± 0.0015
	C_2	1.223	1.263	0	1.241	± 0.0515
	C_3	0.053	0.054	0	0.053	± 0.0185
	C_4	-0.796	-0.760	0	-0.782	± 0.0304
2	C_1	0.277	0.273	0	0.276	± 0.0026
	C_2	1.348	1.410	0	1.318	± 0.0660
	C_3	0.149	0.174	0	0.161	± 0.0166
	C_4	-0.974	-0.901	0	-0.986	± 0.0454
3	C_1	0.224	0.216	0	0.218	± 0.0080
	C_2	1.224	1.183	0	1.254	± 0.0639
	C_3	0.233	0.240	0	0.202	± 0.0601
	C_4	-0.548	-0.576	0	-0.550	± 0.0192
4	C_1	0.235	0.236	0	0.234	± 0.0016
	C_2	1.698	1.697	0	1.689	± 0.0486
	C_3	0.238	0.236	0	0.234	± 0.0089
	C_4	-0.769	-0.770	0	-0.778	± 0.0216
5	C_1	0.198	0.195	0	0.192	± 0.0105
	C_2	1.417	1.396	0	1.452	± 0.0913
	C_3	0.221	0.226	0	0.185	± 0.0824
	C_4	-0.581	-0.592	0	-0.583	± 0.0360

scaling parameters from subsurface thermocouple measurements in the divertor of an NSTX-U tile, project objective two was satisfied. Project objective three required a demonstration of project objective two, but with noise and systematic error added to the thermocouple data.

4.7 Simulating Systematic Error

On NSTX-U, voltage signals from subsurface thermocouples may experience unexpected impedance or induced voltages that result in scaling or biases superimposed upon the original signal. This systematic error may sometimes be calibrated out of the signal, or filtered by digital signal processing (DSP), but in some cases it is unavoidable. To ascertain the competency of this CNN reconstruction system with respect to noise and systematic error, a series of tests were conducted that artificially injected systematic error into the thermocouple data. These tests served to validate the prototype CNN's performance with regard to project objective three: *Demonstrate project objective two, but now add demonstrated uncertainties to measurement and model support parameters.*

To generate artificial noise, the original thermocouple data that was output from the ANSYS simulator was scaled and biased. Due to the fact that periodic time varying alterations to the thermocouple signal can be digitally filtered by frequency component, the applied error for these tests was DC, or time invariant. The scaling and bias was applied to a single thermocouple (second in radial direction) for the duration of the data acquisition period. The general mathematical relationship between the true signal, TC_{true}, and the signal with an applied error, TC_{err}, can be described as follows,

$$TC_{err} = (TC_{true})m + b \tag{4.13}$$

where m is a scaling factor and b [°C] is an added bias term. In order to determine the CNN performance as a function of the scaling and bias term, a single set of three simulated NSTX-U discharges were reproduced 100 times, while varying m and b. The ranges for m

and b are given by:

$$0.5 < m < 1.5 \tag{4.14}$$

$$-10 < b < 10 \tag{4.15}$$

The 100 reproductions were evenly spaced through this two dimensional (m x b) space, and the error for each Eich parameter was calculated at regular intervals. Figures 4.12-4.15 illustrate the contours of CNN prediction error as a function of m and b for each Eich parameter. The center of each contour plot represents the ideal case, when no systematic error has been injected. As one moves away from the ideal case, the error increases. Interestingly, bands of similar error elevation with negative slope ($-b/m$) indicate that if m is increased while decreasing b, then the CNN prediction error will not increase. However, if m and b are both increased (or decreased) simultaneously, then the CNN prediction error will increase indicating inferior performance. Large areas of minimal error in the center of each contour plot illustrate the inherent adaptive nature of the CNN with regards to error. Particularly in C_2 (plot 4.13), injecting error into the thermocouple data has relatively little effect on the CNN prediction capability. This large plateau of minimal C_2 error falls off sharply, however, near the edges of the plot, and the maximum relative error occurs when $m = 1.5$ and $b = 10.0$, which corresponds to a prediction error of approximately 48% relative to the domain of C_2. As can be observed in figure 4.12, C_1 is the most sensitive variable to noise. In all Eich parameters, however, the CNN never exceeds an error of 50%, despite an error in the thermocouple data that exceeds 50%!

It should be emphasized that in figures 4.12-4.15 only a single thermocouple (second thermocouple in radial direction) had error injected into its thermocouple values. If this system were to be incorporated into experimental data acquisition systems on the real NSTX-U, a more thorough investigation would be necessary to quantify the effect of simultaneous noise and error signals on multiple thermocouples. However, the included demonstration suffices to validate that the CNN is robust against error / noise injected onto a single thermocouple signal, which validates the third project objective.

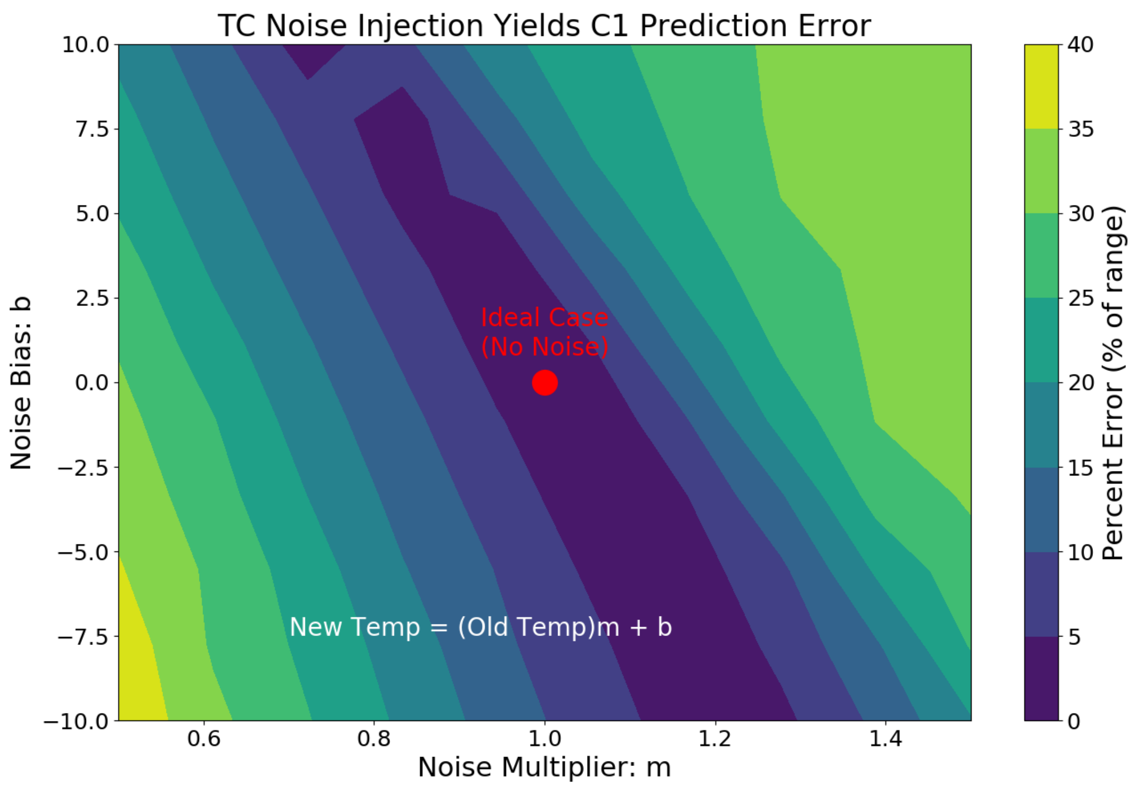


Figure 4.12: C1 error contours: noise injected on thermocouple 2

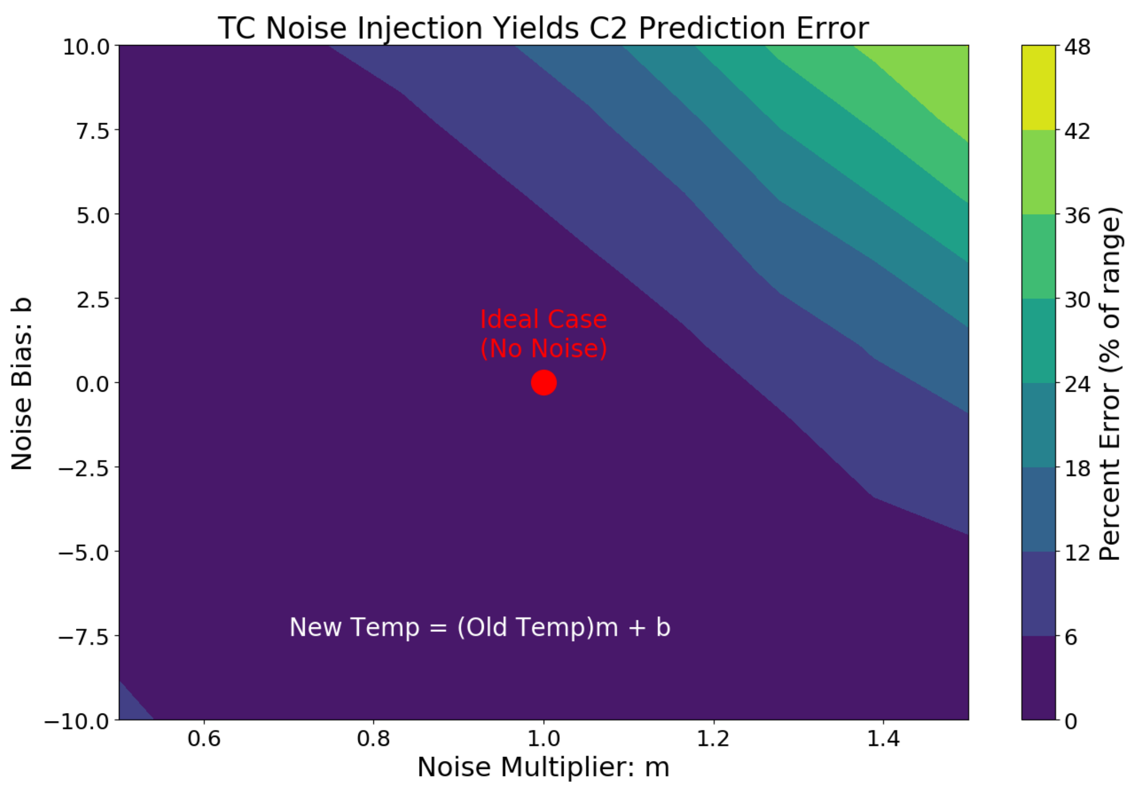


Figure 4.13: C2 error contours: noise injected on thermocouple 2

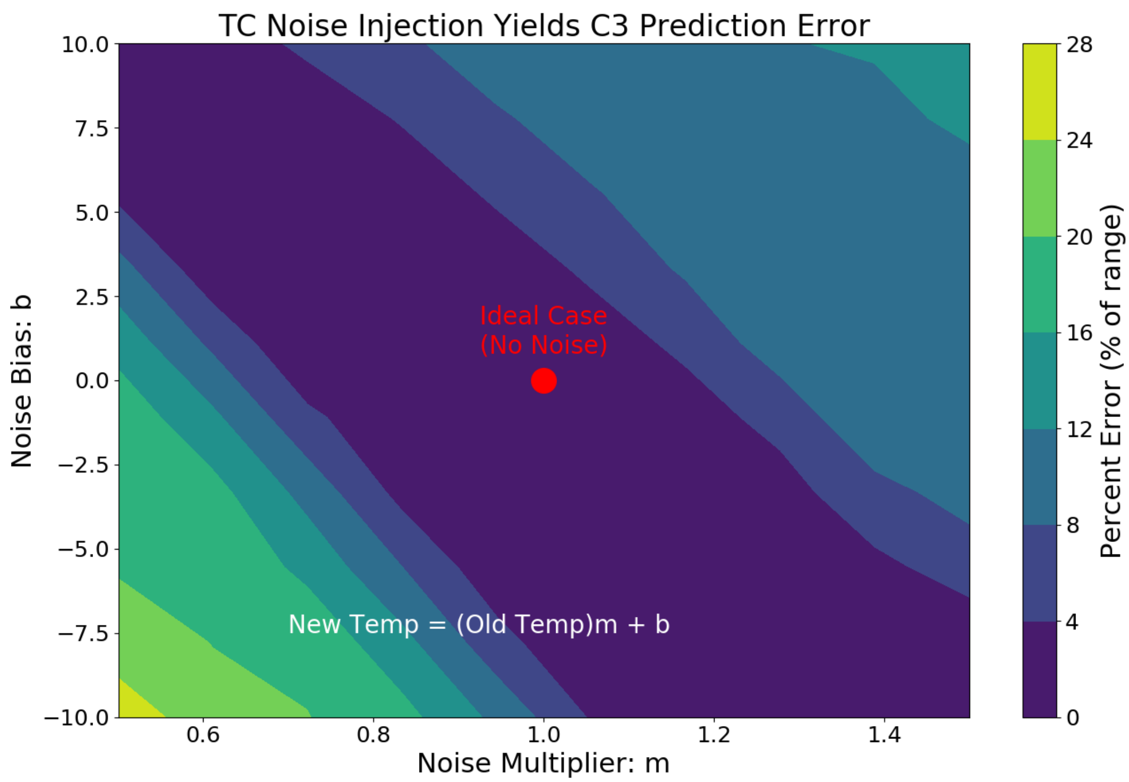


Figure 4.14: C3 error contours: noise injected on thermocouple 2

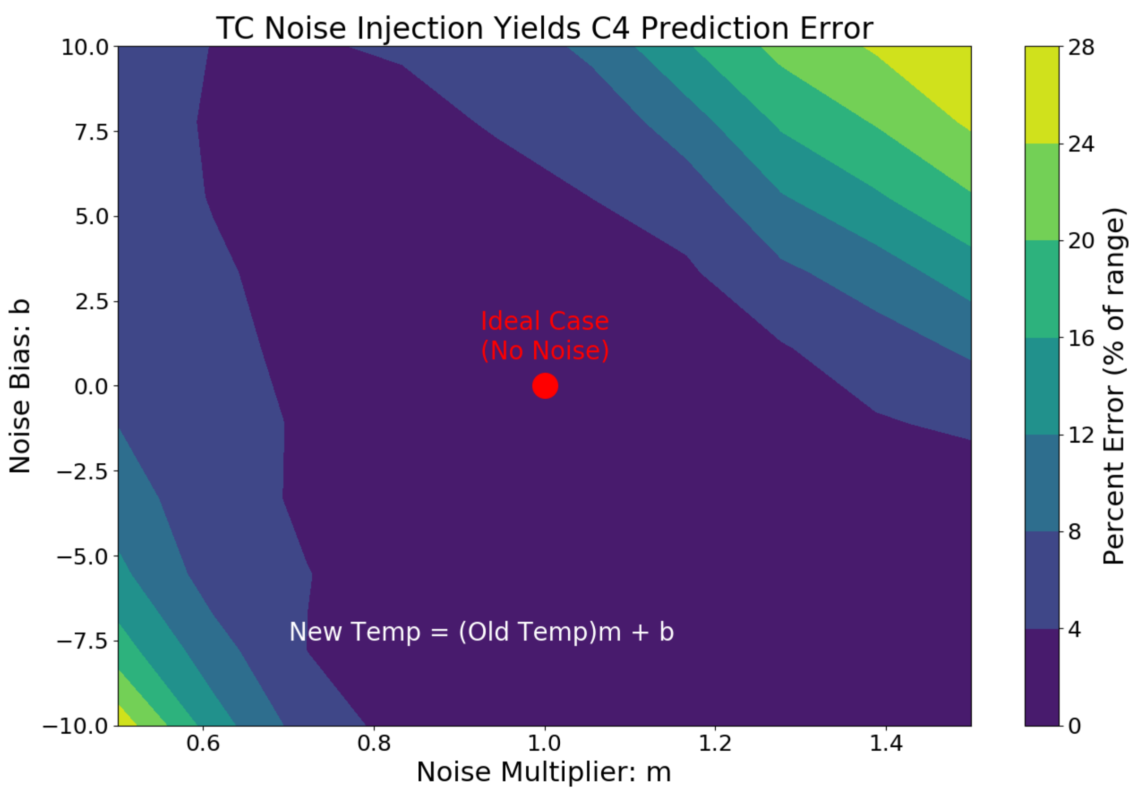


Figure 4.15: C4 error contours: noise injected on thermocouple 2

Chapter 5

Concluding Remarks

Modern tokamak designs require plasma facing components that are capable of withstanding large heat loads in the divertor. The NSTX-U selection of a castellated tile design has greatly minimized the internal stresses that accompany heat loading, but will still require tile monitoring systems to prevent exceeding the engineering limits of the tiles. These monitoring systems must be capable of resolving the heat flux applied to the divertor tile, solely from subsurface thermocouple measurements. Many previous demonstrations of heat flux profile reconstruction from subsurface thermocouples have been performed on tokamaks over the past several decades, in which an inverse solution to the heat diffusion equation was obtained to relate subsurface temperature to surface incident heat flux. While these methods are all successful, they require assumptions associated with deriving analytical solutions to the heat diffusion equation, and also require careful experimental execution with very specific operational windows. In an effort to discover alternative heat flux profile reconstruction methods, modern artificial intelligence and machine learning techniques were explored. More specifically, Convolutional Neural Networks were tested as a mechanism for Eich heat flux profile reconstruction from subsurface thermocouple data.

The Convolutional Neural Network (CNN) employed in this proof of concept can predict Eich scaling parameters with high precision. The CNN accepts machine specifications and thermocouple data as input, and outputs the Eich scaling parameters necessary to resolve the incident heat flux profile. In order to train this CNN, a database of simulated NSTX-U shots was required. To create this database, several software modules were developed, and the ANSYS multiphysics finite element solver was used to generate thermocouple profiles. Roughly 8000 simulated NSTX-U shots were created in this manner, and these shots served to train the CNN to recognize specific thermocouple characteristics from the thermocouple profiles. The final CNN is capable of resolving the incident heat flux profile after only three NSTX-U shots. These three shots must share theoretical Eich scaling parameters, and the machine specifications such as B_P and P_{heat} must change between shots. So long as these restrictions are satisfied, the three shots can sample from any corner of the machine's operational domain without affecting the CNNs ability to make accurate predictions. Furthermore, the CNN is capable of reconstruction despite moderate noise and systematic error. Overall, this CNN proof of concept provides a worthy alternative to the traditional heat diffusion equation inverse method. While the CNN prototype satisfied the project objectives, many improvements and further explorations could be undertaken using this system, which could lead to more advanced reconstruction tasks that are outside of the capabilities of any modern analytical method.

5.0.1 Potential Improvements / Further Applications

The CNN described in this work was trained and tested on NSTX-U shots in which the strike point was not swept, but rather remained in a single position for the duration of the shot. It is entirely possible for the CNN to learn the intricacies associated with any time evolving strike point profile, so long as it is trained with a database that includes time evolving strike point data. The time constraints on this project prevented the development and testing of a strike point sweep database, but if more time were available it would be relatively straightforward to develop a CNN that could perform these predictions. Generating the strike point sweep dataset would take considerable amounts of time, as it would need to be fairly large. The preferred method for generating this dataset would be the utilization of the ANSYS ACT script included in the Appendix on a cluster of CPUs utilizing an ANSYS high performance computing license. The Monte Carlo heat flux generator and CNN described in this work (code in appendix) both already include the option for strike point sweeping, and the sweeping frequency would serve as an additional machine specification that would be fed to the CNN. If the strike point was not swept periodically, but rather swept linearly in a single direction for the entirety of the shot, the strike point velocity could also serve as the input. A more complicated architecture could even include an array of the time evolving strike point position as input.

Another potential improvement to this CNN system would be deriving the relationship between training database size and prediction accuracy. Roughly 8000 NSTX-U simulations were developed to train the CNN, but that number is likely excessive. A rigorous engineering analysis of database size could yield valuable insight, and perhaps result in a reduction to the necessary database size, thereby reducing database production time. More specifically, determining the database resolution needed for each machine specification across its domain may enable a clustering algorithm to greatly reduce the size of the training dataset. Hyperparameter tuning can also affect the prediction accuracy, and deriving the relationship between each hyperparameter and the prediction accuracy could yield optimum configurations for CNN operation. The combination of database analytics and hyperparameter tuning would together enable the CNN to train faster and predict more accurately with higher precision.

The last potential improvement that will be mentioned here is incorporating the full Eich heat flux profile, rather than the simplified Eich model described in chapter 3. The CNN is capable of deriving any multivariate nonlinear model, and full Eich profile model reconstruction would be relatively straightforward. Including the full Eich heat flux model in the Monte Carlo heat flux generator would only require minor adjustments, and the ANSYS simulation and CNN training process would be identical. An additional two shots for each prediction (a total of 5 shots) would then be required to resolve the six full Eich parameters. This would require minor alterations to the CNN architecture, and minor modifications to the system of equation solver.

Because Neural Networks are capable of learning any system for which a database can be generated, the options for applying these techniques are truly endless. Advances in computational power have yielded large databases for every fusion machine, all with data that can be mined and used in machine learning algorithms such as this one. One such example is modeling Edge Localized Modes (ELMs), which are extremely nonlinear phenomenon that escape the prediction capabilities of ideal magnetohydrodynamics (MHD). Because neural networks do not depend on prior analytical derivation, but rather learn through observation, they are the perfect candidate for studying ELMs. Large databases that contain ELM data already exist, and could be mined and used for CNN training.

Another potential neural network application is in real time controls. When utilized with the matrix inversion system equation solver (instead of Newton's method), the CNN in this thesis is capable of generating predictions with approximately 100 ms latency (without any significant effort to increase prediction speed). A plasma control system that controls divertor heat flux profile location and shaping via real time subsurface thermocouple measurements could be developed using a CNN similar to the one described in this work, although it would be limited by the subsurface thermocouple's elevation (proportional to thermal diffusion speed). Not only would the system be capable of real time control, but it could be continually learning as more shots are observed by the CNN. Other diagnostics with faster response times would also be perfect candidates for real time control using artificial intelligence.

Lastly, compiling a database across multiple tokamaks of varying size could be used to train a neural network that can be applied to any tokamak. Including data such as machine major radius, aspect ratio, etc., in the neural network training could result in a system that develops a model that can be scaled to future devices, as well as ported over to unseen machines. One engineering challenge associated with modern tokamak diagnostic systems is that they are each specifically designed for a specific machine. Generating a system that can be modularly applied to any tokamak is possible with machine learning due to its ability to progressively improve the model. And while the discussion provided in this thesis is solely regarding heat flux and thermocouples, these artificial intelligent systems can be applied to any diagnostic.

Regardless of whether or not future improvements are explored, the three project objectives that defined this thesis were all achieved. The NSTX-U thermocouple CNN prototype demonstration was successful, and provided an alternative heat flux reconstruction methodology that leverages the power of modern computer science as a means of modeling a complicated plasma profile in a fusion machine.

Bibliography

[1] P. C. Stangeby, *The plasma boundary of magnetic fusion devices*, ser. Plasma physics series. Institute of Physics Pub. vii, 5, 6

[2] T. Eich, A. Leonard, R. Pitts, W. Fundamenski, R. Goldston, T. Gray, A. Herrmann, A. Kirk, A. Kallenbach, O. Kardaun, A. Kukushkin, B. LaBombard, R. Maingi, M. Makowski, A. Scarabosio, B. Sieglin, J. Terry, A. Thornton, ASDEX Upgrade Team, and JET EFDA Contributors, "Scaling of the tokamak near the scrape-off layer h-mode power width and implications for ITER," vol. 53, no. 9, p. 093031. [Online]. Available: http://stacks.iop.org/0029-5515/53/i=9/a=093031?key=crossref.85c8939d5380098d23c4644a2a101ec9 vii, viii, 6, 7, 33

[3] E. Alpaydin, *Introduction to machine learning*, 3rd ed. MIT Press. vii, 9, 10, 16, 19

[4] D. Stork and S. Zinkle, "Introduction to the special issue on the technical status of materials for a fusion reactor," vol. 57, no. 9, p. 092001. [Online]. Available: http://stacks.iop.org/0029-5515/57/i=9/a=092001?key=crossref.42f13131369b4ae4ec6fa0cd1b5a0036 1

[5] C. Linsmeier, M. Rieth, J. Aktaa, T. Chikada, A. Hoffmann, J. Hoffmann, A. Houben, H. Kurishita, X. Jin, M. Li, A. Litnovsky, S. Matsuo, A. von Mller, V. Nikolic, T. Palacios, R. Pippan, D. Qu, J. Reiser, J. Riesch, T. Shikama, R. Stieglitz, T. Weber, S. Wurster, J.-H. You, and Z. Zhou, "Development of advanced high heat flux and plasma-facing materials," vol. 57, no. 9, p. 092007. [Online]. Available: http://stacks.iop.org/0029-5515/57/i=9/a=092007?key=crossref.4e3ac0cd978f8a92a985f4c55d65c872 1

[6] M. Reinke, "PFCR MEMO 014: GOALS FOR FY18 R18-1 PCRF-WG." [Online]. Available: https://nstx.pppl.gov/DragNDrop/Working_Groups/PFCR/memos/PFCR-MEMO-014-00.pdf 2

[7] ——, "PFCR MEMO 015: INITIAL HEAT FLUX PROFILES AND CONSTRAINTS FOR MODEL VALIDATION SIMULATIONS FOR R18-1/3-G3." [Online]. Available: https://nstx.pppl.gov/DragNDrop/Working_Groups/PFCR/memos/PFCR-MEMO-015-00.pdf 2, 34

[8] F. Wagner, "A quarter-century of h-mode studies," vol. 49, no. 12, pp. B1–B33. [Online]. Available: http://stacks.iop.org/0741-3335/49/i=12B/a=S01?key=crossref.4b1659cd2933e3e40b713717cd852006 4

[9] F. F. Chen, *Introduction to plasma physics and controlled fusion.* Springer Science+Business Media. 4

[10] M. Keilhacker, A. Gibson, C. Gormezano, P. Lomas, P. Thomas, M. Watkins, P. Andrew, B. Balet, D. Borba, C. Challis, I. Coffey, G. Cottrell, H. D. Esch, N. Deliyanakis, A. Fasoli, C. Gowers, H. Guo, G. Huysmans, T. Jones, W. Kerner, R. Knig, M. Loughlin, A. Maas, F. Marcus, M. Nave, F. Rimini, G. Sadler, S. Sharapov, G. Sips, P. Smeulders, F. Sldner, A. Taroni, B. Tubbing, M. v. Hellermann, D. Ward, and J. Team, "High fusion performance from deuterium-tritium plasmas in JET," vol. 39, no. 2, pp. 209–234. [Online]. Available: http://stacks.iop.org/0029-5515/39/i=2/a=306?key=crossref.d40b7269f54a17935a01f96e91953a6b 4

[11] J. P. Freidberg, *Plasma Physics and Fusion Energy.* Cambridge University Press. 4, 20

[12] T. Eich, B. Sieglin, A. Scarabosio, W. Fundamenski, R. J. Goldston, A. Herrmann, and ASDEX Upgrade Team, "Inter-ELM power decay length for JET and ASDEX upgrade: Measurement and comparison with heuristic drift-based model," vol. 107, no. 21. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.107.215001 6

[13] F. Wagner, "A study of the perpendicular particle transport properties in the scrape-off layer of ASDEX," vol. 25, no. 5, pp. 525–536. [Online]. Available: http://stacks.iop.org/0029-5515/25/i=5/a=002?key=crossref.d0e263077ca266b35fe58f00014e1135 6

[14] K. Kamnitsas, D. C. Castro, L. L. Folgoc, I. Walker, R. Tanno, D. Rueckert, B. Glocker, A. Criminisi, and A. Nori, "Semi-supervised learning via compact latent space clustering." [Online]. Available: http://arxiv.org/abs/1806.02679 7, 52

[15] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars." [Online]. Available: http://arxiv.org/abs/1604.07316 7, 52

[16] T. Fischer and C. Krauss, "Deep learning with long short-term memory networks for financial market predictions," vol. 270, no. 2, pp. 654–669. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0377221717310652 7, 52

[17] A. Scott, *Neuroscience: a mathematical primer.* Springer. 9, 15

[18] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," vol. 521, no. 7553, pp. 436–444. [Online]. Available: http://www.nature.com/articles/nature14539 15, 16

[19] T. K. Gray, N. Allen, M. L. Reinke, G. Smalley, D. L. Youchison, R. Ellis, T. Looby, M. Mardenfeld, and D. E. Wolfe, "Integrated plasma facing component calorimetry for measurement of shot integrated deposited energy in NSTX-u," p. 6. 20, 24, 51

[20] *SIGRAFINE TDS-R6510.02 Datasheet*, SGL Carbon SE, 08 2015. [Online]. Available: http://www.sglgroup.com/cms/_common/downloads/products/product-groups/gs/tds/iso/SIGRAFINE_TDS-R6510.02.pdf 21

[21] *SICSS-SHX High Temperature Quick Disconnect Thermocouples Datasheet*, Omega Engineering. [Online]. Available: https://www.omega.com/temperature/pdf/SICSS-SHX.pdf 21

[22] J. R. Cannon, *The one-dimensional heat equation.* Cambridge University Press, 1984, vol. 23. 49, 50

[23] H. Carslaw and J. Jaeger, *Conduction of heat in solids: Oxford Science Publications.* Oxford, England, 1959. 50, 51

[24] V. Riccardo, W. Fundamenski, and G. F. Matthews, "Reconstruction of power deposition profiles using JET MkIIGB thermocouple data for ELMy h-mode plasmas,"

vol. 43, no. 7, pp. 881–906. [Online]. Available: http://stacks.iop.org/0741-3335/43/i=7/a=304?key=crossref.6b22e35d3f8d34c9753c426d4e315548 51

[25] W. Fundamenski, S. Sipil, G. F. Matthews, V. Riccardo, P. Andrew, T. Eich, L. C. Ingesson, T. Kiviniemi, T. Kurki-Suonio, V. Philipps, and c. t. t. E.-J. W. programme, “Interpretation of recent power width measurements in JET MkIIGB ELMy h-modes,” vol. 44, no. 6, pp. 761–793. [Online]. Available: http://stacks.iop.org/0741-3335/44/i=6/a=311?key=crossref.7b79da7ed6c109b8a702b85803b95354 51

[26] G. Matthews, S. Erents, W. Fundamenski, C. Ingesson, R. Monk, and V. Riccardo, “Divertor energy distribution in JET h-modes,” vol. 290-293, pp. 668–672. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0022311500004827 51

[27] J. G. Watkins, C. J. Lasnier, D. G. Whyte, P. C. Stangeby, and M. A. Ulrickson, “Calorimeter probe for the DIII-d divertor,” vol. 74, no. 3, pp. 1574–1577. [Online]. Available: http://aip.scitation.org/doi/10.1063/1.1527241 51

[28] G. Matthews, P. Bunting, S. Devaux, P. Drewelow, C. Guillemaut, D. King, E. Lerche, S. Silburn, G. Szepesi, V. Riccardo, and V. Thompson, “Energy balance in JET,” vol. 12, pp. 227–233. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2352179116301661 51

[29] J. L. Barton, R. E. Nygren, E. A. Unterberg, J. G. Watkins, M. A. Makowski, A. Moser, D. L. Rudakov, and D. Buchenauer, “Comparison of heat flux measurement techniques during the DIII-d metal ring campaign,” vol. T170, p. 014007. [Online]. Available: http://stacks.iop.org/1402-4896/2017/i=T170/a=014007?key=crossref.6ebc06afe3e6f1f08773b39aa349a9d7 51

[30] C. M. Bishop, P. S. Haynes, M. E. U. Smith, T. N. Todd, and D. L. Trotman, “Fast feedback control of a high temperature fusion plasma,” vol. 2, no. 3, pp. 148–159. [Online]. Available: http://link.springer.com/10.1007/BF01415011 53

[31] O. Barana and G. Manduchi, “Application of neural networks for the measurement of electronic temperature in nuclear fusion experiments,” vol. 10, no. 4, pp. 351–356. [Online]. Available: http://link.springer.com/10.1007/s005210200007 53

[32] F. A. Matos, D. R. Ferreira, and P. J. Carvalho, “Deep learning for plasma tomography using the bolometer system at JET,” vol. 114, pp. 18–25. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0920379616306883 53

[33] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization.” [Online]. Available: http://arxiv.org/abs/1412.6980 61

Appendix

A Monte Carlo Flux Generator - Fortran95

```fortran
!flux_genny_rev5.f08

!Title:       Flux Generator (genny)
!Engineer:     Tom Looby
!Date:        03/16/2018
!Description: Makes a set of Fluxes ANSYS / TensorFlow
!Project:        NSTX-U Recovery

!==============================================================================
!             ***MAIN PROGRAM***
!==============================================================================
program flux_genny
use iso_fortran_env
implicit none

integer :: i, j, ntime, nspace, iter, maxiter, failcheck
integer(8) :: t1, t2, rate, cmax

real(real64), ALLOCATABLE :: flux(:,:), r0(:), q(:), alpha(:), beta(:)
real(real64) :: t, dt, PI, dx, r, w, f_strike, spec_arr(9,2)
real(real64) :: c1, c2, c3, c4, xp, xc, P, Bp, fx, lambda, S
real(real64) :: rmin, rmax, tilemin, tilemax, tile_len
real(real64) :: squig1, squig2, squig3, squig4, squig5
real(real64) :: squig6, squig7, squig8, squig9
!real(real64) ::
character(len=200) :: outfile, dir, command, flux_file


!==============================================================================
!              Setup
!==============================================================================
!Initialize Random Number Generator
call init_random_seed()

```

```
!MC iterator
iter = 0

! Directory where results will be saved
dir = '/home/mobile1/school/grad/masters/flux_input/fluxes/fluxes_test/'
command = "mkdir " // trim(dir)
print *, command
failcheck = SYSTEM(trim(command))
if (failcheck == 1) then
   print *, "Failed to create subdirectory"
   CALL EXIT(0)
end if

!==========================================================================
!               Variables, Parameters, Machine Specs
!==========================================================================
PI=4.D0*DATAN(1.D0)
iter = 0

! User defined simulation parameters
print *, "How many spatial slices?"
read (*,*) nspace
print *, "How many time steps?"
read (*,*) ntime
print *, "How many Monte Carlo Runs?"
read (*,*) maxiter

ALLOCATE(flux(ntime, nspace), r0(ntime), q(ntime))
ALLOCATE(alpha(ntime), beta(ntime))

!Tile Specs
tile_len = 0.15431
dx = tile_len/nspace
tilemin = 0.438
tilemax = tilemin + tile_len

```

```
!From Memo, r0 range:
rmin = .46
rmax = .575

! Create Array with min and max values for machine specs
!   for each parameter: [minval, maxval]
!   C1 -> C4 are Eich Model Parameters
!
!rows are machine specs
!Generally, array looks like this:
!
! for array (i,j)
! i                    j1         j2
! Row #    Spec        MinVal     MaxVal
! 1        Bp           0.2        0.6
! 2        P            0.5        4.9
! 3        fx           4.0       30.0
! 4        t            1.0        5.0
! 5        c1           0.1        0.3
! 6        c2           1.0        2.5
! 7        c3          -0.1        0.25
! 8        c4          -1.4       -0.5
! 9        f_strike     0.0       20.0

! Build min/max array
!Bp [T]
spec_arr(1,1) = 0.2
spec_arr(1,2) = 0.6
!P [MW]
spec_arr(2,1) = 0.5
spec_arr(2,2) = 4.9
!fx
spec_arr(3,1) = 4.0
spec_arr(3,2) = 30.0
!t [s]
spec_arr(4,1) = 1.0
```

```fortran
spec_arr(4,2) = 5.0
!c1
spec_arr(5,1) = 0.1
spec_arr(5,2) = 0.3
!c2
spec_arr(6,1) = 1.0
spec_arr(6,2) = 2.5
!c3
spec_arr(7,1) = -0.1
spec_arr(7,2) = 0.25
!c4
spec_arr(8,1) = -1.2
spec_arr(8,2) = -0.5
!f_strike [Hz]
spec_arr(9,1) = 0.0
spec_arr(9,2) = 20.0

!==========================================================================
!              Monte Carlo Machine Parameters
!==========================================================================
! Use a Monte Carlo method to pick random standard deviates from
! uniform distribution between boundaries in spec_arr
! This loop is designed to be run for as long as the user desires
! Alternatively, the user may edit the "iteration checker" to break
! after a specific number of runs.

!This is for CPU clocking.  Grab initial time.
CALL SYSTEM_CLOCK(t1, rate, cmax)

do while (1.NE.0)
   iter = iter + 1

   call random_number(squig1)
   call random_number(squig2)
   call random_number(squig3)
   call random_number(squig4)
```

```
call random_number(squig5)
call random_number(squig6)
call random_number(squig7)
call random_number(squig8)
call random_number(squig9)

Bp = spec_arr(1,1) + squig1*(spec_arr(1,2) - spec_arr(1,1))
P = spec_arr(2,1) + squig2*(spec_arr(2,2) - spec_arr(2,1))
fx = spec_arr(3,1) + squig3*(spec_arr(3,2) - spec_arr(3,1))

! Choose t from dist or keep constant
!t = spec_arr(4,1) + squig4*(spec_arr(4,2) - spec_arr(4,1))
t = 5.0

!Eich Model Parameters (See NSTX PFC Memo)
c1 = spec_arr(5,1) + squig6*(spec_arr(5,2) - spec_arr(5,1))
c2 = spec_arr(6,1) + squig7*(spec_arr(6,2) - spec_arr(6,1))
c3 = spec_arr(7,1) + squig8*(spec_arr(7,2) - spec_arr(7,1))
c4 = spec_arr(8,1) + squig9*(spec_arr(8,2) - spec_arr(8,1))

! Sweep strike point (also have to uncomment R0 stuff below)
f_strike = spec_arr(9,1) + squig5*(spec_arr(9,2) - spec_arr(9,1))

dt = t/real(ntime)
w = 2*PI*f_strike



! r0 can change with time
do i=1,ntime
   ! Function for r0 goes here:
   !time varying sinusoid
!    r0(i) = (tile_len/2.0)*sin(w*(i-1)*dt) + tilemin

   !constant
   r0(i) = tilemin + tile_len/2.0

```

```
    ! Bound min and max r0 per project requirements
!    if (r0(i) > rmax) then
!       r0(i) = rmax
!    elseif (r0(i) < rmin) then
!       r0(i) = rmin
!    end if
end do


!=============================================================================
!                 Solve for S, lambda, and qmax
!=============================================================================
lambda = c2*(P**c3)*(Bp**c4)*10**(-3.0) ! convert to meters
S = lambda*c1
xp = S*fx
xc = lambda*fx

!q can be calculated from integral:
! P = integral(q(r)*2*pi*r*dr), bounded by r0-xp to r0 + xc

do i=1,ntime
   alpha(i) = r0(i) - xp
   beta(i) = r0(i) + xc

   q(i) = (P/(2*PI)) * ( (1/(3*xp)*(r0(i)**3 - alpha(i)**3)) - &
                        (alpha(i)/(2*xc)*(r0(i)**2 - alpha(i)**2)) + &
                        (1/2*(beta(i)**2 - r0(i)**2)) + &
                        (r0(i)/(2*xc)*(beta(i)**2 - r0(i)**2)) - &
                        (1/(3*xc)*(beta(i)**3 - r0(i)**3)) )**(-1.0)
end do

!=============================================================================
!                 Build Flux Profile
!=============================================================================

! Build profile across tile surface
```

```fortran
do i = 1,ntime
   do j = 1, nspace
      r = tilemin + (j-1)*dx
      if (r >= alpha(i) .AND. r < r0(i)) then
         flux(i,j) = q(i)/xp * (r - alpha(i))
      elseif (r >= r0(i) .AND. r < beta(i)) then
         flux(i,j) = q(i) + q(i)/xc * (r0(i) - r)
      else
         flux(i,j) = 0
      end if
   end do
end do

!===============================================================================
!              Write to CSV
!===============================================================================
!Change filename based upon iteration number
write(outfile, "(A13,I0.6)") "flux_profile_", iter
flux_file = trim(dir) // trim(outfile) // ".txt"

OPEN(UNIT=1,FILE=trim(flux_file),FORM="FORMATTED",STATUS="REPLACE",ACTION="
WRITE")
! write basic parameters for this test
WRITE(1,*) '
#==========================================================================='
WRITE(1,*) '# Heat flux generated by fortran program'
WRITE(1,*) '# Parameters Listed Below:'
WRITE(1,*) '# c1 = ', c1
WRITE(1,*) '# c2 = ', c2
WRITE(1,*) '# c3 = ', c3
WRITE(1,*) '# c4 = ', c4
WRITE(1,*) '# Bp = ', Bp
WRITE(1,*) '# P = ', P
WRITE(1,*) '# fx = ', fx
WRITE(1,*) '# t = ', t
WRITE(1,*) '# R0 time varying'
```

```
WRITE(1,*) '#   '
WRITE(1,*) '# This yields the following results...:'
WRITE(1,*) '# Lambda [m]:  ', lambda
WRITE(1,*) '# S:  ', S
WRITE(1,*) '# xp  [m]:  ', xp
WRITE(1,*) '# xc  [m]:  ', xc
WRITE(1,*) '
#===================================================================='


! This uses an inner implied do loop to iterate over array without newline
do i = 1,ntime
   WRITE(UNIT=1, FMT=*) ((i-1)*dt), ",", (flux(i,j)*10**6, ",", j=1,nspace
-1), flux(i,nspace)*10**6
end do
CLOSE(UNIT=1)

!=====================================================================
!             Final Output
!=====================================================================
if (mod(iter,1000) == 0) then
   print *, 'Iteration Number', iter
   CALL SYSTEM_CLOCK(t2, rate, cmax)
   print *, "Elapsed Time [s]: ", real(t2-t1)/real(rate)
   print *, ' '
   !print *, 'Directory Size: ', dir_size
   print *, ' '
   print *, ' '
end if

!=====Iteration checker=====
!Use this if you want to break out after n iterations
!Otherwise leave commented out
if (iter >= maxiter) then
   print *, 'Reached Iteration Maximum'
   exit
```

```fortran
    end if
    !==============================
end do



print *, ' '
print *, ' '
print *, 'Program Exexuted Successfully'
CALL SYSTEM_CLOCK(t2, rate, cmax)
print *, "Total Elapsed Time [s]: ", real(t2-t1)/real(rate)
print *, "Number of flux profiles created: ", iter


CONTAINS
    !This subroutine generates random number seed based upon clock
    SUBROUTINE init_random_seed()
      INTEGER :: i, n, clock
      INTEGER, DIMENSION(:), ALLOCATABLE :: seed
      CALL RANDOM_SEED(size = n)
      ALLOCATE(seed(n))
      CALL SYSTEM_CLOCK(COUNT=clock)
      seed = clock + 37 * (/ (i - 1, i = 1, n) /)
      CALL RANDOM_SEED(PUT = seed)
      DEALLOCATE(seed)
    END SUBROUTINE

end program flux_genny
```

B ANSYS ACT Script - Python

```python
# FluxImport2.py

# Date:          20180315
# Description:   ANSYS ACT Extension for adding heat fluxes from CSVs to Tile Faces
# Engineer:      Tom Looby
# Project:       PPPL NSTX-U Recovery

import os
import datetime
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
from Ansys.UI.Toolkit import *
import units
import graphics
import time



# Start Logging.  Lives in <working_directory>_files\dp0\SYS\MECH
def init(context):
    ExtAPI.Log.WriteMessage("\n\n===== Flux Profile Importer ACT Script Initialized... =====\n\n")

def OnClickB1(analysis):

    #Start a clock for timestamping
    t0 = time.time()

    for file_iter in range(0,99):
        # Print Stuff to Log File
        t1 = time.time()
        ExtAPI.Log.WriteMessage(" ")
```

```
    ExtAPI.Log.WriteMessage("==============================================================")
    ExtAPI.Log.WriteMessage("Iteration Number: {:0>6}".format(file_iter+1))
    ExtAPI.Log.WriteMessage("Time Elapsed at Iteration Start [s]: {:f}".format(t1 - t0))
    ExtAPI.Log.WriteMessage(" ")

    #See if there are any heat fluxes from last time and delete them
    child_count = ExtAPI.DataModel.Project.Model.Analyses[0].Children.Count
    if child_count > 3:
        for del_iter in range(3,child_count):
            ExtAPI.DataModel.Project.Model.Analyses[0].Children[2].Delete()

    #Set up file IO for this iteration through loop
    infile = (r'C:\Users\thoma\Documents\school\grad\masters_thesis\NSTX\flux_import\input_data\tensorflow\flux_profile_{:0>6}.txt'.format(file_iter+1))
    outfile = (r'C:\Users\thoma\Documents\school\grad\masters_thesis\NSTX\flux_import\input_data\tensorflow\TC_profile_{:0>6}.txt'.format(file_iter+1))
    ExtAPI.Log.WriteMessage("Input File:")
    ExtAPI.Log.WriteMessage(infile)
    ExtAPI.Log.WriteMessage("Output File:")
    ExtAPI.Log.WriteMessage(outfile)

    #Basic Initialization Stuff
    model = ExtAPI.DataModel.Project.Model
    part1 = model.Geometry.Children[0]
    body1 = part1.Children[0]
    face_id = []
    face_area = []
    face_ctr = []
    face_data = []
    face_arr = []
    facecounter = 0
#==========================================================================
```

```
#                                  Import Flux Data
#==============================================================================

    # Import Data into data array.  Should be comma delimited.  fh1 opens CSV
    fh1 = open(infile)
    data = []
    for line in fh1:
        li=line.strip()
        if not li.startswith("#"):
            data.append(li.split(', '))
    fh1.close

    #Number of Time Steps
    length = len(data)
    #Number of Spatial Steps + 1
    length2 = len(data[0])
    #Number of spatial steps
    n = length2 - 1
    # Tile Width at narrowest point [m]
    width = 0.02242
    # Length Overall [m]
    loa = 0.15431
    # Area [m^2]
    n_area = width*(loa/n)
    ExtAPI.Log.WriteMessage("Area per Slice: %.7f" % (n_area,))

    # Sometimes weird garbage faces are created when slicing: ignore them.
    # Also, sometimes if a face is too small ANSYS complains about
    # applying a load to it.  This throws a comment into the log file.
    if n_area < 0.000005:
        ExtAPI.Log.WriteMessage("Note!!!  Any slices with area < 5E-6m^2 are ignored!")
        ExtAPI.Log.WriteMessage("ANSYS cannot apply loads to arbitrarily small surfaces.")
        ExtAPI.Log.WriteMessage("Try increasing the number of slices.")
```

```
    # Build Time Array
    time_mag = "[ "
    for i in range(length - 1):
        time_mag += ("Quantity(\"%s [s]\"), " % (data[i][0],))
    time_mag += ("Quantity(\"%s [s]\") ]" % (data[length - 1][0],))

    # Build Flux Magnitude Array
    flux_mag = ["[ " for x in range(length2-1)]
    i=0
    col_sum = [0]*(length2-1)
    for i in range(length-1):
        j=0
        for j in range(length2 - 1):
            flux_mag[j] += ("Quantity(\"%s [W/m^2]\"), " % (data[i][j+1],))
            col_sum[j] = float(data[i][j+1]) + col_sum[j]
            j=j+1
        i=i+1
    j=0

    for j in range(length2-1):
        flux_mag[j] += ("Quantity(\"%s [W/m^2]\") ]" % (data[length-1][j+1],))
        j=j+1

#===============================================================================
#                                Add Heat Flux
#===============================================================================

    # Get data for top surface faces
    for index, face in enumerate(body1.GetGeoBody().Faces, start=0):
        centroid = face.Centroid
        if centroid[1] > 0.053:
            face_data.append([face.Id, face.Area, face.Centroid[0], face.Centroid[1], face.Centroid[2]])

    # Sort the faces in the Z direction (radially)
```

```
    face_data.sort(key=lambda face_arr: face_arr[4])

    #Assign the model we are working on to variable
    model = ExtAPI.DataModel.Project.Model

    i=0
    slice = 0
    j=0
    #Loop through face data and add heat fluxes as required
    for index, face_arr in enumerate(face_data):
        #For troubleshooting...
        #ExtAPI.Log.WriteMessage("INDEX: %i, SLICE: %i" % (index, slice))

        # Sometimes weird garbage faces are created when slicing: ignore them.
        # Also, sometimes if a face is too small ANSYS complains about
        # applying a load to it.
        if face_arr[1] < 0 or face_arr[1] < 0.01*n_area or face_arr[1] < 0.000005:
            continue

        # If no flux is on slice for all time, bail!
        elif col_sum[slice] == 0:
          pass

        # Apply heat flux per conditions below
        else:
            flux = model.Analyses[0].AddHeatFlux()
            # Create Empty Selection
            selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
            #Assign Flux Location by Geometric ID
            selection.Entities = [ExtAPI.DataModel.GeoData.GeoEntityById(face_data[index][0])]
            flux.Location = selection
            #Build Commands from CSV Data
```

```
            time_command = "flux.Magnitude.Inputs[0].DiscreteValues = %s" % (time_mag,)
            mag_command = "flux.Magnitude.Output.DiscreteValues = %s" % (flux_mag[slice],)
            #Execute Commands to Create Fluxes
            exec(time_command)
            exec(mag_command)
            # ExtAPI.Log.WriteMessage("INDEX: %i  ID: %d  ZCoord: %.7f  Area: %.7f" % (index, face_arr[0], face_arr[4], face_arr[1]))

        # Check for multi-castellation face and act accordingly
        if j < 1:
            slice_area = face_arr[1]
        else:
            slice_area = face_arr[1] + slice_area
        if slice_area < n_area*0.9:
            j = j+1
        else:
            slice = slice + 1
            j=0

        if slice > length2 - 1:
            break


    # For reference, this is format of python ACT command to create flux table
    #flux.Magnitude.Inputs[0].DiscreteValues = [ Quantity("0 [s]"), Quantity("5 [s]"), Quantity("6 [s]")]
    #flux.Magnitude.Output.DiscreteValues = [Quantity("10000000 [W/m^2]"), Quantity("10000000 [W/m^2]"), Quantity("0 [W/m^2]")]

    # To print to a file for error checking
    #outfile = open(r'C:\Users\thoma\Desktop\logger.txt','w')
    #outfile.write(command1)
    #outfile.close()
```

```

    #Solve The Model
    ExtAPI.DataModel.Project.Model.Solve(1)

    #===This is for creating an array "tc_arr" with all TC data for all time
    #  This method takes forever, but it works until a better method is found
    T1 = ExtAPI.DataModel.Project.Model.Analyses[0].Solution.Children[1]
    T2 = ExtAPI.DataModel.Project.Model.Analyses[0].Solution.Children[2]
    T3 = ExtAPI.DataModel.Project.Model.Analyses[0].Solution.Children[3]
    T4 = ExtAPI.DataModel.Project.Model.Analyses[0].Solution.Children[4]
    T5 = ExtAPI.DataModel.Project.Model.Analyses[0].Solution.Children[5]
    tc_arr = []
    # Fill tc_arr with time evolving TC data
    for i in range (length-1):
        disp = "%f [s]" % float(data[i][0])
        T1.DisplayTime =Quantity (disp)
        #T1.EvaluateAllResults()

        T2.DisplayTime =Quantity (disp)
        #T2.EvaluateAllResults()

        T3.DisplayTime =Quantity (disp)
        #T3.EvaluateAllResults()

        T4.DisplayTime =Quantity (disp)
        #T4.EvaluateAllResults()

        T5.DisplayTime =Quantity (disp)
        #T5.EvaluateAllResults()

        ExtAPI.DataModel.Project.Model.Analyses[0].Solution.EvaluateAllResults()

```

```
            tc_arr.append([T1.Temperature, T2.Temperature, T3.Temperature, T4.Temperature, T5.Temperature])

        #Write Data to output file for TensorFlow
        fh_out = open(outfile, "w")
        for item in tc_arr:
            fh_out.write("%s\n" % item)
        fh_out.close()

        #Write To Log File
        ExtAPI.Log.WriteMessage("Output Written")
        t2 = time.time()
        ExtAPI.Log.WriteMessage("Iteration Time [s]: {:f}".format(t2 - t1))
        ExtAPI.Log.WriteMessage(" ")
        ExtAPI.Log.WriteMessage("============================================================")
        ExtAPI.Log.WriteMessage(" ")
        #====
    ExtAPI.Log.WriteMessage("All Flux Profiles Crunched...")
    ExtAPI.Log.WriteMessage("Total Time Elapsed [s]: {:f}".format(t2 - t0))
    ExtAPI.Log.WriteMessage("Script Exiting...")
    ExtAPI.Log.WriteMessage(" ")


def LogButtonClicked(toolbarId, buttonId, analysis):
    now = datetime.datetime.now()
    outFile = SetUserOutput("ExtToolbarSample.log", analysis)
    f = open(outFile,'a')
    f.write("*.*.*.*.*.*.*.*\n")
    f.write(str(now)+"\n")
    f.write("Toolbar "+toolbarId.ToString()+" - Button "+buttonId.ToString()+" Clicked. \n")
    f.write("*.*.*.*.*.*.*.*\n")
    f.close()
    MessageBox.Show("Toolbar "+toolbarId.ToString()+" - Button "+buttonId.ToString()+" Clicked.")
```

```python

def SetUserOutput(filename, analysis):
    solverDir = analysis.WorkingDir
    return os.path.join(solverDir,filename)
```

C ANSYS ACT Script - XML

```xml
<extension version="1" minorversion="0" name="FluxImport2">
  <script src="C:\Program Files\ANSYS Student\v182\Addins\ACT\extensions\FluxImport2\FluxImport2.py" />
  <interface context="Mechanical">
    <images>C:\Program Files\ANSYS Student\v182\Addins\ACT\extensions\FluxImport2\images</images>
    <callbacks>
      <oninit>init</oninit>
    </callbacks>
    <toolbar name="Flux Profile" caption="NSTX-U Add Heat Flux Profile">
      <entry name="Add Flux" icon="fire">
        <callbacks>
          <onclick>OnClickB1</onclick>
    </callbacks>
      </entry>
    </toolbar>
  </interface>
</extension>
```

D Data Cleaner Script - Perl

```perl
#!/usr/bin/perl
# Date:          20180418
# Description:  Cleans Data for Tensorflow
# Engineer:     Tom Looby


use strict;
#se warnings;
use Path::Tiny qw(path);

#Set this variable for number of files to be cleaned
my $experN = 99;
my $i;
my $filename;
my $file;
my $data;

for ($i=1; $i<=$experN; $i++){

    $filename = sprintf("/home/workhorse/school/grad/masters/tensorflow/data_test_Cs_const1/TC_profile_%06i.txt", $i);
    $file = path($filename);

   #Read In Data, remove everything for TF, write new data
    $data = $file->slurp_utf8;
    $data =~ s/(\[|\]|C|')//g;
    $file->spew_utf8( $data );
}


print "\nCompleted.  Data Clean.\n\n"
```

E Tensorflow Training Script - Python

```
# cnn_20180819.py

# Date:          20180819
# Description:   Tensorflow CNN Trainer (can import and save model)(S and
    lambda predictions)
# Engineer:      Tom Looby

from __future__ import absolute_import, division, print_function
import os
import os.path
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from numpy import array
from numpy import genfromtxt
from numpy import unravel_index
import functools
import time
import datetime as dt
#import tensorflow.contrib.eager as tfe
#tf.enable_eager_execution()
import csv


start_time = time.time()
print("\nTensorFlow version: {}\n".format(tf.VERSION))

#==========================================================================
#                          Constants, User Inputs
#==========================================================================
# Number of thermocouples
tcN = 5.0
#Number of Machine Specs
machN = 4
```

```
# Number of timesteps
timeN = 48.0
# Number of elements in TC X TIME matrix
N = tcN * timeN
# Number of bins (classes) for answer
outN = 2
# Number of training datasets - goes from (experfirst, experN)
experN = 7200
# Number to begin training on
experfirst = 1
# Number of test datasets - goes from (experN, experN + testN)
testN = 844
# Number of Epochs
num_epochs = 500000
# Batch Size
batch_size = 20
# Learning Rate (1e-3 is good)
lr = 1e-4
# Feature Map Number
fmapN = 16
# Number of Neurons per fully connected layer
neuronN = 32
# How often to record for plotting
samplelen = 5000
# How often to print to screen
printlen = 5000

# Acceptable Error threshold for S [mm]
error_thresh_S = 0.000075/2.0 #  (0.5% of range)
#error_thresh_S = 0.000075 #  (1% of range)
#error_thresh_S = 0.000225 #  (3% of range)
# error_thresh_S = 0.000377 # (5% of range)

# Acceptable Error threshold for Lambda [mm]
error_thresh_lam = 0.000244/2.0 #  (0.5% of range)
#error_thresh_lam = 0.000244 #  (1% of range)
```

```
#error_thresh_lam = 0.000732 #  (3% of range)
# error_thresh_lam = 0.00122 # (5% of range)

# Acceptable Accuracy threshold
acc_thresh = 99.0
# Moving Average Boxcar Window Length
box_window_size = 100.0


#Counter
counter = 0
#Root directory where we are working
#root_dir = '/home/workhorse/school/grad/masters/tensorflow/
    data_20s_nosweep_allrandom/'
root_dir = '/home/workhorse/school/grad/masters/tensorflow/data_nosweep_all/'

#Path for saving plots
figure_dir = '/home/workhorse/school/grad/masters/tensorflow/figures/20
    s_nosweep_allrandom'
#figure_dir = '/home/workhorse/school/grad/masters/tensorflow/figures/20
    s_Cs_const'

#Path for saving weights
weights_dir = '/home/workhorse/school/grad/masters/tensorflow/weights/'


#===Importing Models and Weights===
#Set this flag to 1 for importing model, 0 to start from scratch
import_flag = 1

#Original Test Model
#import_model = '20180625--21_10_24'
import_model = '20180824--01_54_53'
weight_path = '/home/workhorse/school/grad/masters/tensorflow/weights/' +
    import_model + '/weights'

```

```
#Path for Graph Meta Data (only sometimes used here)
meta_path = '/home/workhorse/school/grad/masters/tensorflow/weights/' +
    import_model + '/weights.meta'



truth = []
predicted = []
train_loss_epoch = []
train_loss_results = []
train_error = np.zeros((num_epochs//samplelen, outN))
test_error = np.zeros((num_epochs//samplelen, outN))
train_errorc1 = []
train_errorc2 = []
train_errorc3 = []
train_errorc4 = []
test_loss_results = []
train_acc = []
test_acc = []



Bp = []
P = []
freq = []
fx = []

C_range = np.zeros((4, 4), dtype=np.float32)
C_range[0][0] = 1.0/0.2
C_range[1][1] = 1.0/1.5
C_range[2][2] = 1.0/0.35
C_range[3][3] = 1.0/0.7

#Lambda Range (technically in mm but whatever)
slam_range = np.zeros((2, 2), dtype=np.float32)
slam_range[0][0] = 1.0/7.53954
```

```python
slam_range[1][1] = 1.0/24.4

#err_idx = []
#==============================================================================
#                         Import Dataset
#==============================================================================
def import_data(start_ind, stop_ind):
    """
    This function reads CSV files within the input parameter bounds and
    returns a TF dataset object that includes TC data and machine specs.
    """
    TC_parms = np.zeros((int(timeN), 5))
    TC_data = np.zeros(((stop_ind - start_ind), int(timeN), int(tcN), 1))
    mach_data = np.zeros(((stop_ind - start_ind), machN))
    eich_data = np.zeros(((stop_ind - start_ind), outN))

    #Read data into numpy array
    for i in range(start_ind, stop_ind):
        TCfile = root_dir + 'TC_profile_{:0>6}.txt'.format(i)

        #===Data Testing: Ensure we have no crap data
        #Test Filepath first, if file doesnt exist skip it
        if os.path.isfile(TCfile):
            pass
        else:
            print("Missing TC File: {:6d}...skipping...".format(i) )
            continue

        #Read TC data from ANSYS (has to be cleaned with cleaner.pl)
        #Note: we skip first line because ANSYS makes funky first lines
        TC_parms = (genfromtxt(TCfile, delimiter=',', skip_header=1))

        #===Data Testing: Ensure we have no crap data
        #Check to make sure there is temp data
        if np.sum(TC_parms) == 0.0:
            print("WARNING: TC DATA = 0.0, No. {:0>6}".format(i))
```

```
            continue
        #Check to make sure we cleaned data and dont have any NaNs
        elif np.isnan(np.amax(TC_data)):
            print("WARNING: NAN error: TC DATA, No. {:0>6}".format(i))
            print("Did you clean this data with cleaner.pl...?")
            continue

        # Data in flux array is as follows:
        # [c1, c2, c3, c4]
        #temp1 = [0.0, 0.0, 0.0, 0.0]
        # [S, lambda]
        temp1 = [0.0, 0.0]

        # Data in machine specs array:
        # [Bp, P, freq, fx]
        temp2 = [0.0, 0.0, 0.0, 0.0]

        file2 = root_dir + 'flux_profile_{:0>6}.txt'.format(i)
        fluxfile = open(file2, 'r')
        for line in fluxfile:
            #Skip header line
            if 'Parameters' in line:
                pass
            #Write data into arrays
#             elif 'c1' in line:
#                 temp1[0]= float(line.replace("# c1 = ", " "))
#             elif 'c2' in line:
#                 temp1[1]= float(line.replace("# c2 = ", " "))
#             elif 'c3' in line:
#                 temp1[2]= float(line.replace("# c3 = ", " "))
#             elif 'c4' in line:
#                 temp1[3]= float(line.replace("# c4 = ", " "))
            elif 'S:' in line:
                temp1[0]= float(line.replace("# S:     ", " "))
            elif 'Lambda' in line:
                temp1[1]= float(line.replace("# Lambda [m]:     ", " "))
```

```
            elif 'B' in line:
                temp2[0] = float(line.replace("# Bp =  ", " "))
            elif 'P' in line:
                temp2[1] = float(line.replace("# P =   ", " "))
            elif 'R0' in line:
                #Comment the next line for constant freq
                #temp2[2] = -float(line.replace("# R0 time varying, Freq = ", " "))
                temp2[2] = 0
            elif 'fx' in line:
                temp2[3] = float(line.replace("# fx =    ", " "))

        if temp1[0] == 0.0 or temp1[1] == 0.0: # or temp1[2] == 0.0 or temp1[3] == 0.0:
            print("WARNING: EICH PARAMETER = 0: No. {:0>6}".format(i))

        #Build Eich Data numpy array
        for j in range(outN):
            eich_data[i-start_ind][j] = temp1[j]

        #Build Machine Specs array
        for j in range(machN):
            # for k in range(3):
                # if j == k:
            mach_data[i-start_ind][j] = temp2[j]
        #mach_data = tf.tanh(mach_data)
        for j in range(int(timeN)):
            TC_data[i-start_ind][j][0] = TC_parms[j][0]
            TC_data[i-start_ind][j][1] = TC_parms[j][1]
            TC_data[i-start_ind][j][2] = TC_parms[j][2]
            TC_data[i-start_ind][j][3] = TC_parms[j][3]
            TC_data[i-start_ind][j][4] = TC_parms[j][4]


    #===Normalize TC Data
    maxtemp = np.amax(TC_data)
```

```
    mintemp = np.amin(TC_data)
    print("\nMaximum Temperature in Dataset: {:f}".format(maxtemp))
    print("Minimum Temperature in Dataset: {:f}".format(mintemp))
    for i in range(0, (stop_ind - start_ind)):
        for j in range(int(timeN)):
            for k in range(int(tcN)):
                val = TC_data[i][j][k]
                TC_data[i][j][k]= 2.0/(maxtemp - mintemp) * (val - mintemp) - 1.0

    #===Normalize Machine Spec Data
    for shot in range(0,(stop_ind - start_ind)):
        Bp.append(mach_data[shot][0])
        P.append(mach_data[shot][1])
        freq.append(mach_data[shot][2])
        fx.append(mach_data[shot][3])
    maxBp = np.max(Bp)
    maxP = np.max(P)
    maxfreq = np.max(freq)
    maxfx = np.max(fx)
    minBp = np.min(Bp)
    minP = np.min(P)
    minfreq = np.min(freq)
    minfx = np.min(fx)

    print("Maximum Bp in Dataset: {:f}".format(maxBp))
    print("Minimum Bp in Dataset: {:f}".format(minBp))
    print("Maximum P in Dataset: {:f}".format(maxP))
    print("Minimum P in Dataset: {:f}".format(minP))
    print("Maximum Freq in Dataset: {:f}".format(maxfreq))
    print("Minimum Freq in Dataset: {:f}".format(minfreq))
    print("Maximum fx in Dataset: {:f}".format(maxfx))
    print("Minimum fx in Dataset: {:f}\n".format(minfx))

    #Take Normalized data from range (0,1) to (-1,1)
    for i in range(0, (stop_ind - start_ind)):
        val = mach_data[i][0]
```

```
        mach_data[i][0]= 2.0/(maxBp - minBp) * (val - minBp) - 1.0
        val = mach_data[i][1]
        mach_data[i][1]= 2.0/(maxP - minP) * (val - minP) - 1.0
        val = mach_data[i][2]
        #mach_data[i][2]= 2.0/(maxfreq - minfreq) * (val - minfreq) - 1.0
        mach_data[i][2]= 0.0
        val = mach_data[i][3]
        mach_data[i][3]= 2.0/(maxfx - minfx) * (val - minfx) - 1.0

    print("Read Data From Files...\n\n")

    return TC_data, eich_data, mach_data   # dataset



#==============================================================================
#                    Functions, Classes, Properties
#==============================================================================

def lazy_property(function):
    attribute = '_' + function.__name__

    @property
    @functools.wraps(function)
    def wrapper(self):
        if not hasattr(self, attribute):
            setattr(self, attribute, function(self))
        return getattr(self, attribute)
    return wrapper

class CNNclass:
    def __init__(self, TCdata, machdata, target):
        self.TCdata = TCdata
        self.machdata = machdata
        self.target = target

```

```python
        self.prediction
        self.error
        self.optimize



    @lazy_property
    def prediction(self):

        # Convolution 1 - 2X2 filter => fmapN feature maps
        with tf.name_scope('conv1'):
            W_conv1 = self.weight_variable([5, 5, 1, fmapN])
            b_conv1 = self.bias_variable([fmapN])
            h_conv1 = tf.nn.relu(self.conv2d(self.TCdata, W_conv1) + b_conv1)

        # Pooling Layer 1 - Downsample X2
        #with tf.name_scope('pool1'):
        #    h_pool1 = max_pool_2x2(h_conv1)

        #Convolution 2 - 2X2 filter => 64 feature maps
        with tf.name_scope('conv2'):
            W_conv2 = self.weight_variable([5, 5, fmapN, 2*fmapN])
            b_conv2 = self.bias_variable([2*fmapN])
        #    h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
            h_conv2 = tf.nn.relu(self.conv2d(h_conv1, W_conv2) + b_conv2)

        # Pooling Layer 2 - Downsample X2
        with tf.name_scope('pool2'):
            h_pool2 = self.max_pool_2x2(h_conv2)

        # Fully connected layer 0.5
        with tf.name_scope('fchalf'):
            W_fc1 = self.weight_variable([(int(tcN)+1)//2 * int(timeN)//2 *2*
fmapN, neuronN])
            b_fc1 = self.bias_variable([neuronN])

```

```
        h_pool2_flat = tf.reshape(h_pool2, [-1, (int(tcN)+1)//2 * int(timeN)//2 *2* fmapN])
        #h_pool2_flat = tf.reshape(h_conv2, [-1, 3 * 25 * 64])
        h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)


    # # ALTERNATE 1: Fully connected layer 1: for use with no CNN
    # with tf.name_scope('fc1'):
        # W_fc1 = self.weight_variable([int(N), machN])
        # b_fc1 = self.bias_variable([machN])

        # h_pool2_flat = tf.reshape(self.TCdata, [-1, int(N)])
        # #h_pool2_flat = tf.reshape(h_conv2, [-1, 3 * 25 * 64])
        # h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

    # # OPTIONAL Fully connected layer 2
    # with tf.name_scope('fc2'):
        # W_fc2 = self.weight_variable([neuronN, machN])
        # b_fc2 = self.bias_variable([machN])
        # h_fc2 = tf.nn.relu(tf.matmul(h_fc1, W_fc2) + b_fc2)

    # # ALTERNATE 2:  Multiply machine specs with TC conv data
    # # Fully connected layer 3 (MACHINE SPECS ADDED HERE)
    # with tf.name_scope('fc3'):
        # W_fc3 = self.weight_variable([machN, neuronN])
        # b_fc3 = self.bias_variable([neuronN])
        # # Machine specs go are multiplied with h_fc2
        # mach_h = tf.multiply(h_fc1, self.machdata)
        # #mach_h = tf.add(h_fc2, self.machdata)
        # h_fc3 = tf.nn.relu(tf.matmul(mach_h, W_fc3) + b_fc3)
        # #h_fc3 = tf.nn.relu(tf.matmul(h_fc2, W_fc3) + b_fc3)
        # #y = tf.matmul(mach_h, W_fc3) + b_fc3

    # ALTERNATE 3:  Treat machine specs as separate inputs
    # Fully connected layer 1
    with tf.name_scope('fc4'):
```

```
            mach_h = tf.concat([h_fc1, self.machdata], 1)

            W_fc4 = self.weight_variable([neuronN+machN, neuronN])
            b_fc4 = self.bias_variable([neuronN])
            h_fc4 = tf.nn.relu(tf.matmul(mach_h, W_fc4) + b_fc4)

        # OPTIONAL Fully connected layers 3-8
        with tf.name_scope('fc5'):
            W_fc5 = self.weight_variable([neuronN, neuronN])
            b_fc5 = self.bias_variable([neuronN])
            h_fc5 = tf.nn.relu(tf.matmul(h_fc4, W_fc5) + b_fc5)
        with tf.name_scope('fc6'):
            W_fc6 = self.weight_variable([neuronN, neuronN])
            b_fc6 = self.bias_variable([neuronN])
            h_fc6 = tf.nn.relu(tf.matmul(h_fc5, W_fc6) + b_fc6)
        with tf.name_scope('fc7'):
            W_fc7 = self.weight_variable([neuronN, neuronN])
            b_fc7 = self.bias_variable([neuronN])
            h_fc7 = tf.nn.relu(tf.matmul(h_fc6, W_fc7) + b_fc7)
        with tf.name_scope('fc8'):
            W_fc8 = self.weight_variable([neuronN, neuronN])
            b_fc8 = self.bias_variable([neuronN])
            h_fc8 = tf.nn.relu(tf.matmul(h_fc7, W_fc8) + b_fc8)
        # with tf.name_scope('fc9'):
            # W_fc9 = self.weight_variable([neuronN, neuronN])
            # b_fc9 = self.bias_variable([neuronN])
            # h_fc9 = tf.nn.relu(tf.matmul(h_fc8, W_fc9) + b_fc9)
        # with tf.name_scope('fc10'):
            # W_fc10 = self.weight_variable([neuronN, neuronN])
            # b_fc10 = self.bias_variable([neuronN])
            # h_fc10 = tf.nn.relu(tf.matmul(h_fc9, W_fc10) + b_fc10)
        # with tf.name_scope('fc6'):
            # W_fc11 = self.weight_variable([neuronN, neuronN])
            # b_fc11 = self.bias_variable([neuronN])
            # h_fc11 = tf.nn.relu(tf.matmul(h_fc10, W_fc11) + b_fc11)
        # with tf.name_scope('fc7'):
```

```
      # W_fc12 = self.weight_variable([neuronN, neuronN])
      # b_fc12 = self.bias_variable([neuronN])
      # h_fc12 = tf.nn.relu(tf.matmul(h_fc11, W_fc12) + b_fc12)
    # with tf.name_scope('fc8'):
      # W_fc13 = self.weight_variable([neuronN, neuronN])
      # b_fc13 = self.bias_variable([neuronN])
      # h_fc13 = tf.nn.relu(tf.matmul(h_fc12, W_fc13) + b_fc13)
    # with tf.name_scope('fc9'):
      # W_fc14 = self.weight_variable([neuronN, neuronN])
      # b_fc14 = self.bias_variable([neuronN])
      # h_fc14 = tf.nn.relu(tf.matmul(h_fc13, W_fc14) + b_fc14)
    # with tf.name_scope('fc10'):
      # W_fc15 = self.weight_variable([neuronN, neuronN])
      # b_fc15 = self.bias_variable([neuronN])
      # h_fc15 = tf.nn.relu(tf.matmul(h_fc14, W_fc15) + b_fc15)


    # Fully connected layer 5 - Output Layer
    with tf.name_scope('fc_OUT'):
      W_fc_out = self.weight_variable([neuronN, int(outN)])
      b_fc_out = self.bias_variable([int(outN)])
      y = tf.matmul(h_fc8, W_fc_out) + b_fc_out

    return y



  @lazy_property
  def loss(self):
    #loss = tf.losses.mean_squared_error(self.prediction, self.target)

#     loss = tf.reduce_mean( tf.matmul(tf.abs(self.prediction - self.target), slam_range) )
    loss = 1000*tf.reduce_mean(tf.abs(self.prediction - self.target))

    return loss
```

```
    @lazy_property
    def optimize(self):
        optimizer = tf.train.AdamOptimizer(lr)
        # op1 = optimize.minimize(self.loss1)
        # op2 = optimize.minimize(self.loss2)
        # op3 = optimize.minimize(self.loss3)
        # op4 = optimize.minimize(self.loss4)
        #optimizer = tf.train.GradientDescentOptimizer(lr)
        return optimizer.minimize(self.loss)

    # @lazy_property
    # def optimize(self):
        # #optimizer = tf.train.AdamOptimizer(lr)
        # # op1 = optimize.minimize(self.loss1)
        # optimizer = tf.train.GradientDescentOptimizer(lr)
        # return optimizer.minimize(self.loss)
    # @lazy_property
    # def optimize(self):
        # #optimizer = tf.train.AdamOptimizer(lr)
        # # op2 = optimize.minimize(self.loss2)
        # optimizer = tf.train.GradientDescentOptimizer(lr)
        # return optimizer.minimize(self.loss)
    # @lazy_property
    # def optimize(self):
        # #optimizer = tf.train.AdamOptimizer(lr)
        # # op3 = optimize.minimize(self.loss3)
        # optimizer = tf.train.GradientDescentOptimizer(lr)
        # return optimizer.minimize(self.loss)
    # @lazy_property
    # def optimize(self):
        # #optimizer = tf.train.AdamOptimizer(lr)
        # # op4 = optimize.minimize(self.loss4)
        # optimizer = tf.train.GradientDescentOptimizer(lr)
        # return optimizer.minimize(self.loss)
    # @lazy_property
    # def optimize(self):
```

```
        # optimizer = tf.train.AdamOptimizer(lr)
        # #optimizer = tf.train.GradientDescentOptimizer(lr)
        # return optimizer.minimize(self.loss)

    @lazy_property
    def error(self):
        error = tf.subtract(self.target, self.prediction)
#        error = tf.matmul((self.prediction - self.target), C_range)
        return error

    @staticmethod
    # conv2d returns a 2d convolution layer with full stride
    def conv2d(x, W):
      return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

    @staticmethod
    # max_pool_2x2 downsamples a feature map by 2X
    def max_pool_2x2(x):
      return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

    @staticmethod
    # Generates a weight variable of a given shape
    def weight_variable(shape):
        initial = tf.truncated_normal(shape, stddev=0.01)
        return tf.Variable(initial)
    @staticmethod
    # Generates a bias variable of a given shape
    def bias_variable(shape):
        initial = tf.constant(0.0, shape=shape)
        return tf.Variable(initial)

#===========================================================================
#                       Main Program
#===========================================================================

```

```python

def main():
    train_statcounter = 0.0
    train_hits = 0.0
    train_acc_window = np.zeros((int(box_window_size)))
    test_statcounter = 0.0
    test_hits = 0.0
    test_acc_window = np.zeros((int(box_window_size)))



    #=== First, create training Dataset
    TC_data, eich_data, mach_data = import_data(experfirst, experN + experfirst)
    # Organize Data into dataset object for tensorflow
    dataset = tf.data.Dataset.from_tensor_slices((TC_data, mach_data, eich_data))
# Commented for triplet. All other cases uncomment
    dataset = dataset.shuffle(buffer_size=10000)
    dataset = dataset.batch(batch_size)
    dataset = dataset.repeat(num_epochs)
    # Create iterator for dataset
    iterator = dataset.make_one_shot_iterator()
    next_element = iterator.get_next()

    #=== Create Test Dataset
    TC_data_test, eich_data_test, mach_data_test = import_data(experN + experfirst, experN + experfirst + testN )
    # Organize Data into dataset object for tensorflow
    dataset_test = tf.data.Dataset.from_tensor_slices((TC_data_test, mach_data_test, eich_data_test))
    dataset_test = dataset_test.shuffle(buffer_size=10000)
    dataset_test = dataset_test.batch(1)
    dataset_test = dataset_test.repeat(num_epochs)
    # Create iterator for dataset
    iterator_test = dataset.make_one_shot_iterator()
```

```
    next_element_test = iterator.get_next()

    # Input Data
    with tf.name_scope('TC_input_data'):
        data_TC = tf.placeholder(tf.float32, [None, int(timeN), int(tcN), 1])

    with tf.name_scope('mach_input_data'):
        data_mach = tf.placeholder(tf.float32, [None, machN])

    # Expected Result (y_ is expected)
    with tf.name_scope('Expected_Result'):
        target = tf.placeholder(tf.float32, [None, outN])

    # Build the model
    model = CNNclass(data_TC, data_mach, target)

    sess = tf.InteractiveSession()

    # Add ops to save and restore all the variables.
    saver = tf.train.Saver()

    # If weight importer flag is set then we import weights and dont
    # initialize variables
    if import_flag ==1:
        global weight_path
        saver = tf.train.import_meta_graph(meta_path)
        saver.restore(sess, weight_path)
        print("Read model from file.  Model Restored\n")
    else:
        tf.global_variables_initializer().run()

    # Create Filewriter for Tensorboard Visualization
    # writer = tf.summary.FileWriter("/tmp/tf_test")
    # writer.add_graph(sess.graph)

    #Debugging stuff
```

```python
#var = [v for v in tf.trainable_variables() if v.name == "fc1/Variable_1
:0"][0]

#==========================================================================
#               Training
#==========================================================================
# Train
for epoch in range(num_epochs):
    #Training Data
    x_TC, x_mach, y_eich = sess.run(next_element)
    sess.run(model.optimize, {data_TC: x_TC, data_mach: x_mach, target:
 y_eich})
    error = sess.run(model.error, {data_TC: x_TC, data_mach: x_mach, target:
  y_eich})

    #Test Data
    x_TC_test, x_mach_test, y_eich_test = sess.run(next_element_test)
    test_error = sess.run(model.error, {data_TC: x_TC_test, data_mach:
 x_mach_test, target: y_eich_test})

    #===== Accuracy Windows =====
    # Training Batch Statistics
    for idx in range(len(error)):
        train_statcounter += 0.02
        for err_idx in range(outN):
            # Check if prediction was within allowerable tolerance
            if err_idx == 0:
                if abs(error[idx][err_idx]) < error_thresh_S:
                    train_hits += 1.0
            elif err_idx == 1:
                if abs(error[idx][err_idx]) < error_thresh_lam:
                    train_hits += 1.0
    # Calculate Accuracy
    # Shift boxcar window then write new value in: moving window
    train_acc_window = np.roll(train_acc_window, 1)
    train_acc_window[0] = train_hits/train_statcounter
```

```
        # Reset accuracy counters
        train_hits = 0.0
        train_statcounter = 0.0

        # Test Batch Statistics
        for idx in range(len(test_error)):
            test_statcounter += 0.02
            for err_idx in range(outN):
                # Check if prediction was within allowerable tolerance
#                print("test error========{:f}".format(test_error[0][0]))
                if err_idx == 0:
                    if abs(test_error[idx][err_idx]) < error_thresh_S:
                        test_hits += 1.0
                elif err_idx == 1:
                    if abs(test_error[idx][err_idx]) < error_thresh_lam:
                        test_hits += 1.0

        # Calculate Accuracy
        # Shift boxcar window then write new value in: moving window
        test_acc_window = np.roll(test_acc_window, 1)
        test_acc_window[0] = test_hits/test_statcounter
        # Reset accuracy counters
        test_hits = 0.0
        test_statcounter = 0.0

        if epoch % samplelen == 0:
            loss_value = sess.run(model.loss, {data_TC: x_TC, data_mach: x_mach, target: y_eich} )

            #Append for plot
            train_loss_results.append(loss_value)
            train_loss_epoch.append(epoch)

            error_sum = np.zeros((2))
            # Batch Statistics
            for batch_idx in range(batch_size):
```

```
            for err_idx in range(2):
                # Regular Sum
                error_sum[err_idx] += abs(error[batch_idx][err_idx])

        #Append for plot
        for err_idx in range(2):
            train_error[epoch//samplelen][err_idx] = error_sum[err_idx]/
batch_size

        #Accuracy Stats
        train_accuracy = sum(train_acc_window)/box_window_size
        train_acc.append(train_accuracy)

        test_accuracy = sum(test_acc_window)/box_window_size
        test_acc.append(test_accuracy)


        #=== Test Dataset
        # loss_test = sess.run(model.loss, {data_TC: x_TC_test, data_mach:
x_mach_test, target: y_eich_test})
        # test_loss_results.append(loss_test)
        # for err_idx in range(4):
            # test_error[epoch//samplelen][err_idx] = error_test[0][err_idx]


    if epoch % printlen == 0:
        test_resulty_ = sess.run(model.target, {data_TC: x_TC_test, data_mach
: x_mach_test, target: y_eich_test})
        test_resulty = sess.run(model.prediction, {data_TC: x_TC_test,
data_mach: x_mach_test, target: y_eich_test})
        # resulty_ = sess.run(model.target, {data_TC: x_TC, data_mach: x_mach
, target: y_eich})
        # resulty = sess.run(model.prediction, {data_TC: x_TC, data_mach:
x_mach, target: y_eich})

        # Debugging Stuff
```

```
            #print(tf.trainable_variables())
            #test = sess.run(var)
            #print(test)


            print("========================Epoch {:06d}========================".format(epoch,))
            print("Time Elapsed ~ {:d} seconds\n".format(int(time.time() - start_time),))
            print("Batch Averaged Training Data Error:")
            print(train_error[epoch//samplelen])
            print("Training Data Loss:")
            print(loss_value)
            print("Training Accuracy:")
            print(train_accuracy)
            #print("\n")

            print("\n+++=== Test Data Example Results ===+++")
            print("Expected: ")
            print(test_resulty_[0])
            print("Predicted: ")
            print(test_resulty[0])
            print("\n")
            print("Test Data Accuracy:")
            print(test_accuracy)
            print("\n")


            # print("\n+++=== Test Data Results ===+++")
            # print("Expected: ")
            # print(test_resulty_[0])
            # print("Predicted: ")
            # print(test_resulty[0])
            # print("\n")
        if test_accuracy > acc_thresh:
            break
```

```python

#==============================================================================
#               Save Weights for Predictor
#==============================================================================

#Save Weights
ts = time.time()
st = dt.datetime.fromtimestamp(ts).strftime('%Y%m%d—%H_%M_%S')
os.mkdir(os.path.join(weights_dir, st))
weight_path = os.path.join(weights_dir, st, 'weights' )
print("\n Saving weights to:")
print(weight_path)
save_path = saver.save(sess, weight_path)

#==============================================================================
#               Plots
#==============================================================================

# Plot Loss with Matplotlib
plt.figure(1)

#plt.subplot(211)
plt.title("CNN: LR = {:.4f}; BatchSize = {:4d}; Neuron N = {:4d}; FMaps: {:4d}".format(lr, batch_size, neuronN, fmapN))
#plt.title('Loss vs. Epoch')
axes = plt.gca()
axes.set_ylim([0.0,100.0])
plt.plot(train_loss_epoch, train_acc, 'b', label="Training Accuracy")
plt.plot(train_loss_epoch, test_acc, 'g', label="Test Accuracy")
#plt.plot(train_loss_epoch, train_loss_results, 'b', label="Training Data")
#plt.plot(train_loss_epoch, test_loss_results, 'g', label="Test Data")
#plt.xlabel('Epoch')
plt.ylabel('Accuracy (within tolerance)')
plt.xlabel('Epoch')
plt.legend(loc='upper right')
# ts = time.time()
```

```
    # st = datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d--%H:%M:%S')
    # plotname = "lr{:f}_bs{:d}_neuronN{:d}_fmap{:d}.png".format(lr, batch_size, neuronN, fmapN)
    # plt.savefig(os.path.join(figure_dir, st ,plotname)) # , bbox_inches='tight'
    plt.show()

    #===LOSS PLOT
    # plt.figure(2)
    # #plt.subplot(212)
    # axes = plt.gca()
    # axes.set_ylim([0.0,0.01])
    # #Train Data
    # plt.plot(train_loss_epoch, [row[0] for row in train_error], label="Train S Error")
    # plt.plot(train_loss_epoch, [row[1] for row in train_error], label="Train Lambda Error")
    # #plt.plot(train_loss_epoch, [row[2] for row in train_error], label="Train C3 Error")
    # #plt.plot(train_loss_epoch, [row[3] for row in train_error], label="Train C4 Error")

    # #Test Data
    # # plt.plot(train_loss_epoch, [row[0] for row in test_error], label="Test C1 Error")
    # # plt.plot(train_loss_epoch, [row[1] for row in test_error], label="Test C2 Error")
    # # plt.plot(train_loss_epoch, [row[2] for row in test_error], label="Test C3 Error")
    # # plt.plot(train_loss_epoch, [row[3] for row in test_error], label="Test C4 Error")
    # plt.legend(loc='upper right')
    # plt.xlabel('Epoch')
    # plt.ylabel('Absolute Error')
    # #ts = time.time()
    # #st = datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d--%H:%M:%S')
```

```
    # #plotname = "lr{:f}_bs{:d}_neuronN{:d}_fmap{:d}.png".format(lr,
     batch_size, neuronN, fmapN)
    # #plt.savefig(os.path.join(figure_dir, st ,plotname)) # , bbox_inches='
     tight'
    # plt.show()





if __name__ == '__main__':
    main()
```

F Tensorflow Predictor Script - Python

```
# cnn_20180819.py

# Date:          20180819
# Description:   Tensorflow CNN Trainer (can import and save model)(S and lambda predictions)
# Engineer:      Tom Looby

from __future__ import absolute_import, division, print_function
import os
import os.path
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from numpy import array
from numpy import genfromtxt
from numpy import unravel_index
from numpy.linalg import inv
import functools
import time
import datetime as dt
#import tensorflow.contrib.eager as tfe
#tf.enable_eager_execution()
import csv


start_time = time.time()
print("\nTensorFlow version: {}\n".format(tf.VERSION))

#==============================================================================
#                          Constants, User Inputs
#==============================================================================
# Number of thermocouples
tcN = 5.0
#Number of Machine Specs
```

```
machN = 4
# Number of timesteps
timeN = 48.0
# Number of elements in TC X TIME matrix
N = tcN * timeN
# Number of bins (classes) for answer
outN = 2
# Number of training datasets - goes from (experfirst, experN)
experN = 0
# Number to begin training on
experfirst = 1
# Number of test datasets - goes from (experN, experN + testN)
testN = 50
# Number of Epochs
num_shots = 3
# Number of times we run num_shots for statistics
num_tests = 5
# Batch Size
batch_size = 1
# Learning Rate (1e-3 is good)
lr = 1e-5
# Feature Map Number
fmapN = 16
# Number of Neurons per fully connected layer
neuronN = 32
# How often to record for plotting
samplelen = 1
# How often to print to screen
printlen = 1
# Acceptable Error threshold
error_thresh = 0.05
# Acceptable Accuracy threshold
acc_thresh = 99.0
# Moving Average Boxcar Window Length
box_window_size = 1.0

```

```
#Counter
counter = 0
#Root directory where we are working
#root_dir = '/home/workhorse/school/grad/masters/tensorflow/
    data_20s_Cs_const_nosweep/'
root_dir = '/home/workhorse/school/grad/masters/tensorflow/data_test_Cs_const1
    /'
#Path for saving plots
figure_dir = '/home/workhorse/school/grad/masters/tensorflow/figures/20
    s_nosweep_allrandom'
#figure_dir = '/home/workhorse/school/grad/masters/tensorflow/figures/20
    s_Cs_const'

#Path for saving weights
weights_dir = '/home/workhorse/school/grad/masters/tensorflow/weights/'


#===Importing Models and Weights===
#Set this flag to 1 for importing model, 0 to start from scratch
import_flag = 1
import_model = '20180824--01_54_53'
weight_path = '/home/workhorse/school/grad/masters/tensorflow/weights/' +
    import_model + '/weights'

#Path for Graph Meta Data (only sometimes used here)
meta_path = '/home/workhorse/school/grad/masters/tensorflow/weights/' +
    import_model + '/weights.meta'

#Path for prediction CSV output
csv_path = '/home/workhorse/school/grad/masters/tensorflow/prediction_results/
    csvs/'


truth = []
```

```
predicted = []
train_loss_epoch = []
train_loss_results = []
train_error = np.zeros((num_shots//samplelen, outN))
test_error = np.zeros((num_shots//samplelen, outN))
train_errorc1 = []
train_errorc2 = []
train_errorc3 = []
train_errorc4 = []
test_loss_results = []
train_acc = []
test_acc = []



Bp = []
P = []
freq = []
fx = []

C_range = np.zeros((4, 4), dtype=np.float32)
C_range[0][0] = 1.0/0.2
C_range[1][1] = 1.0/1.5
C_range[2][2] = 1.0/0.35
C_range[3][3] = 1.0/0.7

#err_idx = []
#==============================================================================
#                          Import Dataset
#==============================================================================
def import_data(start_ind, stop_ind):
    """
    This function reads CSV files within the input parameter bounds and
    returns a TF dataset object that includes TC data and machine specs.
    """
    TC_parms = np.zeros((int(timeN), 5))
```

```
    TC_data = np.zeros(((stop_ind - start_ind), int(timeN), int(tcN), 1))
    mach_data = np.zeros(((stop_ind - start_ind), machN))
    mach_data_norm = np.zeros(((stop_ind - start_ind), machN))
    eich_data = np.zeros(((stop_ind - start_ind), outN))

    #Read data into numpy array
    for i in range(start_ind, stop_ind):
        TCfile = root_dir + 'TC_profile_{:0>6}.txt'.format(i)

        #===Data Testing: Ensure we have no crap data
        #Test Filepath first, if file doesnt exist skip it
        if os.path.isfile(TCfile):
            pass
        else:
            print("Missing TC File: {:6d}...skipping...".format(i) )
            continue

        #Read TC data from ANSYS (has to be cleaned with cleaner.pl)
        #Note: we skip first line because ANSYS makes funky first lines
        TC_parms = (genfromtxt(TCfile, delimiter=',', skip_header=1))

        #===Data Testing: Ensure we have no crap data
        #Check to make sure there is temp data
        if np.sum(TC_parms) == 0.0:
            print("WARNING: TC DATA = 0.0, No. {:0>6}".format(i))
            continue
        #Check to make sure we cleaned data and dont have any NaNs
        elif np.isnan(np.amax(TC_data)):
            print("WARNING: NAN error: TC DATA, No. {:0>6}".format(i))
            print("Did you clean this data with cleaner.pl...?")
            continue

        # Data in flux array is as follows:
        # [c1, c2, c3, c4]
        #temp1 = [0.0, 0.0, 0.0, 0.0]
        # [S, lambda]
```

```
    temp1 = [0.0, 0.0]

    # Data in machine specs array:
    # [Bp, P, freq, fx]
    temp2 = [0.0, 0.0, 0.0, 0.0]

    file2 = root_dir + 'flux_profile_{:0>6}.txt'.format(i)
    fluxfile = open(file2, 'r')
    for line in fluxfile:
        #Skip header line
        if 'Parameters' in line:
            pass
        #Write data into arrays
#        elif 'c1' in line:
#            temp1[0]= float(line.replace("# c1 = ", " "))
#        elif 'c2' in line:
#            temp1[1]= float(line.replace("# c2 = ", " "))
#        elif 'c3' in line:
#            temp1[2]= float(line.replace("# c3 = ", " "))
#        elif 'c4' in line:
#            temp1[3]= float(line.replace("# c4 = ", " "))
        elif 'S:' in line:
            temp1[0]= float(line.replace("# S:    ", " "))
        elif 'Lambda' in line:
            temp1[1]= float(line.replace("# Lambda [m]:    ", " "))
        elif 'B' in line:
            temp2[0] = float(line.replace("# Bp =  ", " "))
        elif 'P' in line:
            temp2[1] = float(line.replace("# P =   ", " "))
        elif 'R0' in line:
            #Comment the next line for constant freq
            #temp2[2] = -float(line.replace("# R0 time varying, Freq = ", " ")
)
            temp2[2] = 0
        elif 'fx' in line:
            temp2[3] = float(line.replace("# fx =    ", " "))
```

```python

    if temp1[0] == 0.0 or temp1[1] == 0.0: # or temp1[2] == 0.0 or temp1[3]
 == 0.0:
        print("WARNING: EICH PARAMETER = 0: No. {:0>6}".format(i))

    #Build Eich Data numpy array
    for j in range(outN):
        eich_data[i-start_ind][j] = temp1[j]

    #Build Machine Specs array
    for j in range(machN):
        # for k in range(3):
            # if j == k:
        mach_data[i-start_ind][j] = temp2[j]
    #mach_data = tf.tanh(mach_data)
    for j in range(int(timeN)):
        TC_data[i-start_ind][j][0] = TC_parms[j][0]
        TC_data[i-start_ind][j][1] = TC_parms[j][1]
        TC_data[i-start_ind][j][2] = TC_parms[j][2]
        TC_data[i-start_ind][j][3] = TC_parms[j][3]
        TC_data[i-start_ind][j][4] = TC_parms[j][4]


#===Normalize TC Data
maxtemp = np.amax(TC_data)
mintemp = np.amin(TC_data)
print("\nMaximum Temperature in Dataset: {:f}".format(maxtemp))
print("Minimum Temperature in Dataset: {:f}".format(mintemp))
for i in range(0, (stop_ind - start_ind)):
    for j in range(int(timeN)):
        for k in range(int(tcN)):
            val = TC_data[i][j][k]
            TC_data[i][j][k]= 2.0/(maxtemp - mintemp) * (val - mintemp) - 1.0

#===Normalize Machine Spec Data
for shot in range(0,(stop_ind - start_ind)):
```

```
        Bp.append(mach_data[shot][0])
        P.append(mach_data[shot][1])
        freq.append(mach_data[shot][2])
        fx.append(mach_data[shot][3])
    maxBp = np.max(Bp)
    maxP = np.max(P)
    maxfreq = np.max(freq)
    maxfx = np.max(fx)
    minBp = np.min(Bp)
    minP = np.min(P)
    minfreq = np.min(freq)
    minfx = np.min(fx)

    print("Maximum Bp in Dataset: {:f}".format(maxBp))
    print("Minimum Bp in Dataset: {:f}".format(minBp))
    print("Maximum P in Dataset: {:f}".format(maxP))
    print("Minimum P in Dataset: {:f}".format(minP))
    print("Maximum Freq in Dataset: {:f}".format(maxfreq))
    print("Minimum Freq in Dataset: {:f}".format(minfreq))
    print("Maximum fx in Dataset: {:f}".format(maxfx))
    print("Minimum fx in Dataset: {:f}\n".format(minfx))

    #Take Normalized data from range (0,1) to (-1,1)
    for i in range(0, (stop_ind - start_ind)):
        val = mach_data[i][0]
        mach_data_norm[i][0]= 2.0/(maxBp - minBp) * (val - minBp) - 1.0
        val = mach_data[i][1]
        mach_data_norm[i][1]= 2.0/(maxP - minP) * (val - minP) - 1.0
        val = mach_data[i][2]
        #mach_data[i][2]= 2.0/(maxfreq - minfreq) * (val - minfreq) - 1.0
        mach_data_norm[i][2]= 0.0
        val = mach_data[i][3]
        mach_data_norm[i][3]= 2.0/(maxfx - minfx) * (val - minfx) - 1.0

    print("Read Data From Files...\n\n")

```

```python
    return TC_data, eich_data, mach_data_norm, mach_data    # dataset



#==========================================================================
#                 Functions, Classes, Properties
#==========================================================================

def lazy_property(function):
    attribute = '_' + function.__name__

    @property
    @functools.wraps(function)
    def wrapper(self):
        if not hasattr(self, attribute):
            setattr(self, attribute, function(self))
        return getattr(self, attribute)
    return wrapper

class CNNclass:
    def __init__(self, TCdata, machdata, target):
        self.TCdata = TCdata
        self.machdata = machdata
        self.target = target

        self.prediction
        self.error
        self.optimize



    @lazy_property
    def prediction(self):

        # Convolution 1 - 2X2 filter => fmapN feature maps
        with tf.name_scope('conv1'):
```

```
            W_conv1 = self.weight_variable([5, 5, 1, fmapN])
            b_conv1 = self.bias_variable([fmapN])
            h_conv1 = tf.nn.relu(self.conv2d(self.TCdata, W_conv1) + b_conv1)

        # Pooling Layer 1 - Downsample X2
        #with tf.name_scope('pool1'):
        #    h_pool1 = max_pool_2x2(h_conv1)

        #Convolution 2 - 2X2 filter => 64 feature maps
        with tf.name_scope('conv2'):
            W_conv2 = self.weight_variable([5, 5, fmapN, 2*fmapN])
            b_conv2 = self.bias_variable([2*fmapN])
        #    h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
            h_conv2 = tf.nn.relu(self.conv2d(h_conv1, W_conv2) + b_conv2)

        # Pooling Layer 2 - Downsample X2
        with tf.name_scope('pool2'):
            h_pool2 = self.max_pool_2x2(h_conv2)

        # Fully connected layer 0.5
        with tf.name_scope('fchalf'):
            W_fc1 = self.weight_variable([(int(tcN)+1)//2 * int(timeN)//2 *2* fmapN, neuronN])
            b_fc1 = self.bias_variable([neuronN])

            h_pool2_flat = tf.reshape(h_pool2, [-1, (int(tcN)+1)//2 * int(timeN)//2 *2* fmapN])
            #h_pool2_flat = tf.reshape(h_conv2, [-1, 3 * 25 * 64])
            h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)


        # # ALTERNATE 1: Fully connected layer 1: for use with no CNN
        # with tf.name_scope('fc1'):
            # W_fc1 = self.weight_variable([int(N), machN])
            # b_fc1 = self.bias_variable([machN])

```

```
            # h_pool2_flat = tf.reshape(self.TCdata, [-1, int(N)])
            # #h_pool2_flat = tf.reshape(h_conv2, [-1, 3 * 25 * 64])
            # h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

        # # OPTIONAL Fully connected layer 2
        # with tf.name_scope('fc2'):
            # W_fc2 = self.weight_variable([neuronN, machN])
            # b_fc2 = self.bias_variable([machN])
            # h_fc2 = tf.nn.relu(tf.matmul(h_fc1, W_fc2) + b_fc2)

        # # ALTERNATE 2:  Multiply machine specs with TC conv data
        # # Fully connected layer 3 (MACHINE SPECS ADDED HERE)
        # with tf.name_scope('fc3'):
            # W_fc3 = self.weight_variable([machN, neuronN])
            # b_fc3 = self.bias_variable([neuronN])
            # # Machine specs go are multiplied with h_fc2
            # mach_h = tf.multiply(h_fc1, self.machdata)
            # #mach_h = tf.add(h_fc2, self.machdata)
            # h_fc3 = tf.nn.relu(tf.matmul(mach_h, W_fc3) + b_fc3)
            # #h_fc3 = tf.nn.relu(tf.matmul(h_fc2, W_fc3) + b_fc3)
            # #y = tf.matmul(mach_h, W_fc3) + b_fc3

        # ALTERNATE 3:  Treat machine specs as separate inputs
        # Fully connected layer 1
        with tf.name_scope('fc4'):
            mach_h = tf.concat([h_fc1, self.machdata], 1)

            W_fc4 = self.weight_variable([neuronN+machN, neuronN])
            b_fc4 = self.bias_variable([neuronN])
            h_fc4 = tf.nn.relu(tf.matmul(mach_h, W_fc4) + b_fc4)

        # OPTIONAL Fully connected layers 3-8
        with tf.name_scope('fc5'):
            W_fc5 = self.weight_variable([neuronN, neuronN])
            b_fc5 = self.bias_variable([neuronN])
            h_fc5 = tf.nn.relu(tf.matmul(h_fc4, W_fc5) + b_fc5)
```

```
        with tf.name_scope('fc6'):
            W_fc6 = self.weight_variable([neuronN, neuronN])
            b_fc6 = self.bias_variable([neuronN])
            h_fc6 = tf.nn.relu(tf.matmul(h_fc5, W_fc6) + b_fc6)
        with tf.name_scope('fc7'):
            W_fc7 = self.weight_variable([neuronN, neuronN])
            b_fc7 = self.bias_variable([neuronN])
            h_fc7 = tf.nn.relu(tf.matmul(h_fc6, W_fc7) + b_fc7)
        with tf.name_scope('fc8'):
            W_fc8 = self.weight_variable([neuronN, neuronN])
            b_fc8 = self.bias_variable([neuronN])
            h_fc8 = tf.nn.relu(tf.matmul(h_fc7, W_fc8) + b_fc8)
        # with tf.name_scope('fc9'):
            # W_fc9 = self.weight_variable([neuronN, neuronN])
            # b_fc9 = self.bias_variable([neuronN])
            # h_fc9 = tf.nn.relu(tf.matmul(h_fc8, W_fc9) + b_fc9)
        # with tf.name_scope('fc10'):
            # W_fc10 = self.weight_variable([neuronN, neuronN])
            # b_fc10 = self.bias_variable([neuronN])
            # h_fc10 = tf.nn.relu(tf.matmul(h_fc9, W_fc10) + b_fc10)
        # with tf.name_scope('fc6'):
            # W_fc11 = self.weight_variable([neuronN, neuronN])
            # b_fc11 = self.bias_variable([neuronN])
            # h_fc11 = tf.nn.relu(tf.matmul(h_fc10, W_fc11) + b_fc11)
        # with tf.name_scope('fc7'):
            # W_fc12 = self.weight_variable([neuronN, neuronN])
            # b_fc12 = self.bias_variable([neuronN])
            # h_fc12 = tf.nn.relu(tf.matmul(h_fc11, W_fc12) + b_fc12)
        # with tf.name_scope('fc8'):
            # W_fc13 = self.weight_variable([neuronN, neuronN])
            # b_fc13 = self.bias_variable([neuronN])
            # h_fc13 = tf.nn.relu(tf.matmul(h_fc12, W_fc13) + b_fc13)
        # with tf.name_scope('fc9'):
            # W_fc14 = self.weight_variable([neuronN, neuronN])
            # b_fc14 = self.bias_variable([neuronN])
            # h_fc14 = tf.nn.relu(tf.matmul(h_fc13, W_fc14) + b_fc14)
```

```
        # with tf.name_scope('fc10'):
            # W_fc15 = self.weight_variable([neuronN, neuronN])
            # b_fc15 = self.bias_variable([neuronN])
            # h_fc15 = tf.nn.relu(tf.matmul(h_fc14, W_fc15) + b_fc15)


        # Fully connected layer 5 - Output Layer
        with tf.name_scope('fc_OUT'):
            W_fc_out = self.weight_variable([neuronN, int(outN)])
            b_fc_out = self.bias_variable([int(outN)])
            y = tf.matmul(h_fc8, W_fc_out) + b_fc_out

        return y



    @lazy_property
    def loss(self):
        #loss = tf.losses.mean_squared_error(self.prediction, self.target)

        loss = tf.reduce_mean( tf.abs(self.prediction - self.target) )
        #loss = tf.reduce_sum(tf.losses.absolute_difference(self.prediction, self.target))
        #loss = tf.losses.absolute_difference(self.prediction[0][3], self.target[0][3])
        return loss
    @lazy_property
    def optimize(self):
        optimizer = tf.train.AdamOptimizer(lr)
        # op1 = optimize.minimize(self.loss1)
        # op2 = optimize.minimize(self.loss2)
        # op3 = optimize.minimize(self.loss3)
        # op4 = optimize.minimize(self.loss4)
        #optimizer = tf.train.GradientDescentOptimizer(lr)
        return optimizer.minimize(self.loss)

```

```
    # @lazy_property
    # def optimize(self):
        # #optimizer = tf.train.AdamOptimizer(lr)
        # # op1 = optimize.minimize(self.loss1)
        # optimizer = tf.train.GradientDescentOptimizer(lr)
        # return optimizer.minimize(self.loss)
    # @lazy_property
    # def optimize(self):
        # #optimizer = tf.train.AdamOptimizer(lr)
        # # op2 = optimize.minimize(self.loss2)
        # optimizer = tf.train.GradientDescentOptimizer(lr)
        # return optimizer.minimize(self.loss)
    # @lazy_property
    # def optimize(self):
        # #optimizer = tf.train.AdamOptimizer(lr)
        # # op3 = optimize.minimize(self.loss3)
        # optimizer = tf.train.GradientDescentOptimizer(lr)
        # return optimizer.minimize(self.loss)
    # @lazy_property
    # def optimize(self):
        # #optimizer = tf.train.AdamOptimizer(lr)
        # # op4 = optimize.minimize(self.loss4)
        # optimizer = tf.train.GradientDescentOptimizer(lr)
        # return optimizer.minimize(self.loss)
    # @lazy_property
    # def optimize(self):
        # optimizer = tf.train.AdamOptimizer(lr)
        # #optimizer = tf.train.GradientDescentOptimizer(lr)
        # return optimizer.minimize(self.loss)

    @lazy_property
    def error(self):
        error = tf.subtract(self.target, self.prediction)
#        error = tf.matmul((self.prediction - self.target), C_range)
        return error

```

```
  @staticmethod
  # conv2d returns a 2d convolution layer with full stride
  def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

  @staticmethod
  # max_pool_2x2 downsamples a feature map by 2X
  def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

  @staticmethod
  # Generates a weight variable of a given shape
  def weight_variable(shape):
     initial = tf.truncated_normal(shape, stddev=0.01)
     return tf.Variable(initial)
  @staticmethod
  # Generates a bias variable of a given shape
  def bias_variable(shape):
     initial = tf.constant(0.0, shape=shape)
     return tf.Variable(initial)

#==============================================================================
#                 Main Program
#==============================================================================


def main():
  train_statcounter = 0.0
  train_hits = 0.0
  train_acc_window = np.zeros((int(box_window_size)))
  test_statcounter = 0.0
  test_hits = 0.0
  test_acc_window = np.zeros((int(box_window_size)))
  mach = np.zeros((int(num_shots),int(4)))
  eich = np.zeros((int(num_shots),int(outN)))
```

```
  eich_pred = np.zeros((int(num_shots),int(outN)))

  #=== Create Test Dataset
  TC_data_test, eich_data_test, mach_data_test, mach_data = import_data(
   experN + experfirst, experN + experfirst + testN )
  # Organize Data into dataset object for tensorflow
  dataset_test = tf.data.Dataset.from_tensor_slices((TC_data_test,
   mach_data_test, eich_data_test, mach_data))
  dataset_test = dataset_test.shuffle(buffer_size=10000)
  dataset_test = dataset_test.batch(1)
  dataset_test = dataset_test.repeat(10000)
  # Create iterator for dataset
  iterator_test = dataset_test.make_one_shot_iterator()
  next_element_test = iterator_test.get_next()

  # Input Data
  with tf.name_scope('TC_input_data'):
     data_TC = tf.placeholder(tf.float32, [None, int(timeN), int(tcN), 1])

  with tf.name_scope('mach_input_data'):
     data_mach = tf.placeholder(tf.float32, [None, machN])

  # Expected Result (y_ is expected)
  with tf.name_scope('Expected_Result'):
     target = tf.placeholder(tf.float32, [None, outN])

  # Build the model
  model = CNNclass(data_TC, data_mach, target)

  sess = tf.InteractiveSession()

  # Add ops to save and restore all the variables.
  saver = tf.train.Saver()

  # If weight importer flag is set then we import weights and dont
  # initialize variables
```

```
    if import_flag ==1:
        global weight_path
        saver = tf.train.import_meta_graph(meta_path)
        saver.restore(sess, weight_path)
        print("Read model from file.  Model Restored\n")
    else:
        tf.global_variables_initializer().run()

    # Create Filewriter for Tensorboard Visualization
    # writer = tf.summary.FileWriter("/tmp/tf_test")
    # writer.add_graph(sess.graph)

    #Debugging stuff
    #var = [v for v in tf.trainable_variables() if v.name == "fc1/Variable_1
    :0"][0]


    #==========================================================================
    #               Test the Data with the Model
    #==========================================================================
    # Here, epochs = # of tests we are performing
    #
    rangeflag =  0
    testshot = 0
    predict_mat = np.zeros((num_tests,4))
    err_sq = np.zeros((4))

    while (testshot < num_tests):
        for epoch in range(num_shots):
            #Test Data
            x_TC_test, x_mach_test, y_eich_test, x_mach = sess.run(
    next_element_test)

            # To test input:
#           print("===TEST===")
#           print (x_mach)
```

```
#          print(y_eich_test)

        test_resulty_ = sess.run(model.target, {data_TC: x_TC_test, data_mach: x_mach_test, target: y_eich_test})
        test_resulty = sess.run(model.prediction, {data_TC: x_TC_test, data_mach: x_mach_test, target: y_eich_test})

        #Eich and Machine specs for postprocessing
        for err_idx in range(2):
            eich_pred[epoch][err_idx] = test_resulty[0][err_idx]
            eich[epoch][err_idx] = test_resulty_[0][err_idx]

        for err_idx in range(4):
            mach[epoch][err_idx] = x_mach[0][err_idx]
        # Debugging Stuff
        #print(tf.trainable_variables())
        #test = sess.run(var)
        #print(test)
    #print("+ ===== Prediction Matrix ====== +")
    #print(eich_pred)
    #print("+ ===== Expected Matrix ====== +")
    #print(eich)
    #
    #print("\n+ ===== Machine Specs Matrix ===== +")
    #print(mach)

    #==========================================================================
    #            Newton's Method (solve nonlinear Eich system)
    #==========================================================================
    #print("\nSolving Nonlinear System with Newton's Method...")
    counter = 0
    # Initial Guess for C (c[0] is scaled by 1e-3 for conversion from [m] to [mm])
    c = ([0.00175], [0.075], [-0.85])
    c=np.array(c)
    dF = np.zeros((3,3))
```

```
    F = np.zeros((3,1))

    newt_thresh = 1e-12
    newt_error = 1.0


    while True:
        counter += 1
        for shot in range(3):
            F[shot] = (c[0] \
                        # P^C3
                        * np.power(mach[shot][1], c[1]) \
                        # B^C4
                        * np.power(mach[shot][0], c[2]) \
                        # lambda
                        - eich_pred[shot][1])
            for cval in range(3):
                if cval == 0:
                    dF[shot][cval] = np.power(mach[shot][1], c[1]) * np.power(
mach[shot][0], c[2]) # P^C3 * B^C4

                elif cval == 1:
                    dF[shot][cval] = (c[0] \
                                      * np.power(mach[shot][1], c[1]) \
                                      * np.power(mach[shot][0], c[2]) \
                                     # ln(P)
                                      * np.log(mach[shot][1]))

                elif cval == 2:
                    dF[shot][cval] = (c[0] \
                                      * np.power(mach[shot][1], c[1]) \
                                      * np.power(mach[shot][0], c[2]) \
                                      # ln(B)
                                      * np.log(mach[shot][0]))


```

```
        c_new = c - np.matmul(inv(dF), F)
        newt_error = (np.sum( np.abs(c_new - c), axis=0))
        c = c_new

        # Calculate C1 (not included in Newton's Method)
        n = float(num_shots)
        eich_avg = (eich_pred.sum(axis=0))/n
        c1 = eich_avg[0]/eich_avg[1]

        # If C values add up to over 1000, we are probalby diverging
        if (np.sum(c) > 100 or np.sum(c) < -100):
            rangeflag = 1
            break


        if (newt_error < newt_thresh):
            break

    #===Error checking our results for incorrect solutions
    #===If found, request another <num_shots> shots

    # If we diverge, request a new shot
    if (rangeflag ==1):
        testshot += 0
        #reset the flag
        rangeflag = 0
        print("Newtons Method Overflow Error: Requesting New Shot")
        print(c)

    # If a C value is outside domain, request another shot
    elif (c1 > 0.3 or c1 < 0.1 or
            1000*c[0] > 2.5 or 1000*c[0] < 1.0 or
            c[1] > 0.25 or c[1] < -0.1 or
            c[2] > -0.5 or c[2] < -1.2):
        print("Newtons Method Range Error: Requesting New Shot")
        #For Error Checking / debugging
```

```
            #print(c1)
            #print(float(1000*c[0]))
            #print(float(c[1]))
            #print(float(c[2]))
            testshot += 0
        else:
            predict_mat[testshot][0] = c1
            predict_mat[testshot][1] = c[0]*1000
            predict_mat[testshot][2] = c[1]
            predict_mat[testshot][3] = c[2]

            testshot += 1


    #==============================================================================
    #               Results
    #==============================================================================
    #Calculate mean for each output variable
    n = float(num_tests)
    predict_avg = (predict_mat.sum(axis=0))/n

    #Calculate variance
    for idx in range(num_tests):
        for idx2 in range(4):
            err_sq[idx2] += (predict_mat[idx][idx2] - predict_avg[idx2])**2

    var = err_sq/n


    print("\n\n\n+++ ====================================================================== +++")
    print("                 Prediction Results: ")
    print("+++ ====================================================================== +++\n")
    print("Time Elapsed ~ {:d} seconds".format(int(time.time() - start_time)),)
```

```
    print("\nPredicted Eich Values (mean):")
    print("C1: {:f} +/- {:f}".format(predict_avg[0], np.sqrt(var[0])))
    print("C2: {:f} +/- {:f}".format(predict_avg[1], np.sqrt(var[1])))
    print("C3: {:f} +/- {:f}".format(predict_avg[2], np.sqrt(var[2])))
    print("C4: {:f} +/- {:f}\n".format(predict_avg[3], np.sqrt(var[3])))

#    print("Expected Eich Values:")
#    print("C1: {:f}".format(eich[0][0]))
#    print("C2: {:f}".format(eich[0][1]))
#    print("C3: {:f}".format(eich[0][2]))
#    print("C4: {:f}".format(eich[0][3]))

    print("\n")


    #==========================================================================
    #              Save Results to CSV
    #==========================================================================

    # Save the S / lambda preidictions in a matrix
    if not os.path.exists(csv_path + import_model):
        print("Creating new results directory...")
        os.mkdir(csv_path + import_model)
    prediction_path = csv_path + import_model + '/predict_matrix_{:d}shots'.format(num_shots*num_tests,)
    np.savetxt(prediction_path, predict_mat, delimiter=",")

    #Append means and std devs to summary file
    summary_path = csv_path + import_model + '/mean_stddev.csv'

    header_row = '# of Shots,C1 Mean,C1 StdDev,\
                                C2 Mean,C2 StdDev,\
                                C3 Mean,C3 StdDev,\
                                C4 Mean,C4 StdDev\n'

```

```
    data_row = '{:d},{:f},{:f},{:f},{:f},{:f},{:f},{:f},{:f}\n'.format(
                                            num_shots*num_tests,
                                            predict_avg[0],np.sqrt(var[0]),
                                            predict_avg[1],np.sqrt(var[1]),
                                            predict_avg[2],np.sqrt(var[2]),
                                            predict_avg[3],np.sqrt(var[3])
                                            )

    #Check if file exists for creating a header row
    existsflag = 0
    if not os.path.exists(csv_path + import_model + '/mean_stddev.csv'):
        existsflag = 1

    fh = open(summary_path, 'a')
    if existsflag == 1:
        print("Creating new summary file...")
        fh.write(header_row)
    fh.write(data_row)
    fh.close()

    print("Wrote data to csvs here:\n{:s}\n".format((csv_path + import_model),)
    )


if __name__ == '__main__':
    main()
```

Vita

Tom was raised in a small town, at 9400ft elevation, deep in the Colorado Rockies. He spent a large portion of his childhood outside, exploring the wilderness in his backyard. After high school, he moved to the Pacific Coast, where he began working on Motor-yachts as a deckhand, and was first introduced to engineering in the engine room of a 130 foot motoryacht off the coast of Mexico. After cruising the western coast of North America, he relocated to Boise, where he spent a year working as a locomotive electrician and exploring the northern Rockies. He returned to Colorado and obtained his associates degree from Colorado Mountain College in Glenwood Springs while working as an electrician in Aspen.

Due to his upbringing in the mountains, Tom was inspired to take action to promote environmental conservation. He determined that the best method for reducing oil consumption was to replace fossil fuel with a cheaper, higher energy density alternative, and his quest to contribute to the development of nuclear fusion as an energy source was born. Tom completed his Bachelor's Degree in Electrical Engineering at the University of Denver. Simultaneously, he spent two years working as an electrical engineer for an electric utility, where he designed and tested advanced technologies for intelligent infrastructure systems (smart cities / smart grid) under the guidance of veteran engineer Dan Nordell, PE. Currently, Tom is a Nuclear Engineering graduate student working with Dr. David Donovan at the University of Tennessee - Knoxville. During his first semester in graduate school, Tom connected with Dr. Matt Reinke (Oak Ridge National Lab) and began assisting him with work on the National Spherical Tokamak eXperiment (NSTX), which is the origin of the contents of this thesis. Additionally, Tom spent a summer working at Sandia National Laboratories, under the supervision of Dr. Mark Savage at the Z-Machine. When Tom isn't working, you are likely to find him exploring somewhere in the closest mountain range, with his trusty dog, Tesla.