A. Problem Description

The problem that I solved in my project was creating Forsyth-Edwards notation from digital images of 2D chess boards. Forsyth-Edwards notation is a way of quickly documenting a chess position within a game such that the position can be easily recreated, and the game can be resumed. This notation has been helpful to chess players since its adoption and is the standard way of notating any game. While the notation is not particularly difficult to create, an automated system capable of automatically documenting any position would be very helpful when trying to document an entire game as is generally the practice within most tournaments. Also, having a line of text from which a game could be recreated takes up less storage space than a digital image of the board. Therefore, I took it upon myself to solve this problem using neural network algorithms and machine learning methods.

B. Dataset

The dataset I used in my project came from a database hub called Kaggle. Kaggle is an online website containing many different datasets where machine learning engineers can practice their skills and occasionally compete for prizes in contests. The dataset that I found contained 100,000 unique digital 2D images of chess boards that had been randomly created by the poster of the dataset. Each chess board had up to 15 different pieces that were randomly located throughout the board. Different pieces were given different likelihoods of being selected to appear on the boards. Because of the random generation, each board has a unique notation label. Due to hardware restrictions, I was forced to scale down the dataset to about 1500 images.

C. Neural Network Models

I tried two main neural network types in my project: multi-layer perceptron and radial basis functions. Multi-layer perceptron was tried first and I many different models with a wide variety of layer sizes and number of hidden neurons were tried. The activation function was tangential sigmoid for the hidden layers and softmax for the output layer. Networks with one hidden layer were tried initially but I soon realized I would need two layers to accomplish the job. However, even with two layers with 60 hidden neurons each, the maximum accuracy achieved was around 45% classification accuracy for the board squares. For the two-layer perceptron, 5, 10, 15, 20, 25, 35, 45 and 60 hidden neurons per layer were tried for learning rates ranging from 0.1 – 0.3 in increments of 0.05. I discovered that the performance from this type of network was just too low, so I decided to try a different kind of network.

Next, I tried radial basis function, or RBF, neural networks. I used the matlab toolbox command that would incrementally create a network until the mean squared error of the network was below a certain range. I found this type of network worked much better although I did notice that the number of neurons in each network I tried was much larger than the multi-layer perceptron networks. I used gaussian RBF's and varied the spread and mse targets to find what hyperparameters worked the best. I noted that a spread of 0.001 worked the best in terms of convergence speed and with an mse target of 0.015, achieved 85% square classification. This is much improved performance over the MLP networks but still wasn't good enough since even 1 misclassified square would result in a mislabeled board. However, I decided to keep the RBF network and instead augment my postprocessing

methods. This let me achieve better results that I will discuss in future sections. My final model was a RBF neural network with a spread parameter of 0.001 and an mse target of 0.15.

D. Pre-Processing method

Since my dataset contained all unique labels, effectively meaning each board belonged to its own class, a clever way of pre-processing the data was essential to my success. Basically, I preprocessed the data such that the neural network only had to classify one square at a time. This reduced my number of classes from 1500 (the number of unique labels in my dataset) to 13 (6 different pieces for each color plus a blank square). First, I needed to convert the RGB photos into a gray scale image since the neural network can only handle 2-dimensional data. Afterwards, I systematically divided each 400x400 pixel board into 64 50x50 pixel squares and then converted the 50x50 pixel square into a 1x2500 column vector observation. Thus, for each board in the dataset, I generated 64 observations with 2500 features each. I quickly figured out that this was too large of a dataset for my computer to handle so I opted for a dimension reduction technique to make the number of predictors more manageable. I decided to use principle component analysis (PCA) and converted the 2500 predictors into 15 principle components that still explained over 90% of the variation in the response variable. I verified this using the matlab pca tool. Now, I had a dataset with 64 observations per board and only 15 variables per observation. Next, I needed to convert each FEN label of the board to 64 responses corresponding to each square on the board. I accomplished this by creating a mapping of the characters in the notation to a set of numbers (0 for a blank square, 1 for 'K' meaning King, etc.). Using my knowledge of the notation scheme, I was able to create an 8x8 matrix in matlab containing numbers that corresponded to correct classification of each square on the board. Following the same orientation that I used to subdivide the board, I created one-hot, 13 column vectors denoting the appropriate classification of the square for each square. I then added these response vectors to the dataset. Originally, this was the end of the post-processing but after being dissatisfied with the performance of my model, I decided to augment my pre-processing procedure by balancing the dataset. I accomplished this by counting the number of responses corresponding to each class in the training dataset. I then determined the class with the lowest number of responses and under-sampled the other classes to create a balanced dataset. This seems to have fixed the biasing issues I had encountered earlier and led to the model giving better performance.

E. Post-Processing

The post-processing of the neural network output can be considered the opposite of the pre-processing. I started by rounding the output of the neural network to a nice vector containing only 0's and 1's (I used 0.5 as my rounding threshold). I then checked each output vector for confusion: I made sure that only 1 column had a 1 and the rest were zeros. If the neural network had an output were multiple pieces were selected as the correct response (a vector with more than one column containing and one), I simply assigned that response to be a blank square. My reasoning for this is that I cannot simply discard a confused response because the FEN requires each square to be accounted for. Therefore, I figured that a blank square is a neutral response between assigning the square to a random piece of one color. In some ways, this skews the response of my network and could possibly cause its perceived performance to be better than reality however I could not conceive of a different method of solving this issue. In any case, the processed responses would then be used to create an 8x8 'board' matrix of what the neural network 'thinks' the board looks like (much like the grid of

numbers in pre-processing). This matrix numbers corresponding to the piece (or lack thereof) that the network thinks is there. Creating an inverse mapping to the one mentioned in pre-processing, I was able to take this matrix of numbers and recreate the predicted Forsyth-Edwards notation for that board.

F. Testing, Training and Validation

To train the MLP networks, I simply used the train function contained in the matlab toolbox. This lets me specify the observations and responses I want to use to train, validate and test the model. The function automatically splits the data that you pass it into training, validation and testing sets by randomly sampling indices until the three sets reach a 70%, 15% and 15% split respectively. Therefore, the training function trains and validates the model iteratively until the mse target is reached and then gives the final metric based on the test set. Since I wanted to use overall predictive accuracy as a metric however, I validated the trained model on the entire training post-training to determine what measure of square classification accuracy I can expect.

For the RBF's, I used the builtin matlab toolbox to create the network by iteratively adding to it until a certain mse over the training set was achieved. In this case, I did not use a validation set and simply trained the model to the desired accuracy and then tested the model on the training set again to receive my desired performance metrics and ensure it delivers the classification accuracy that I needed. The actual testing of this model occurred when I passed it the entire dataset of images and had it create the predictive labels.

G. Performance and performance metrics

Because of the number of different response classes that I had, I found it difficult to use many of the classification metrics we discussed in class (Cohen's Kappa which is only useful for two classes for instance). Considering the weight of misclassification is the same for all classes (any misclassified squares will result in wrong label), I decided to use pure classification accuracy as my performance metric. For both the square classification and correct label prediction, I kept a tally of how many neural network outputs matched the desired response. I then divided the correct responses by the total number of observations to get the overall classification or prediction accuracy. For classification of squares, an accuracy of 92% was achieved. For labels, a prediction accuracy of only 15% was achieved. This was because misclassification of even one square on a board results in the label of that board being incorrectly predicted.

H. Network Architecture and Learning Algorithm

The neural network architecture that I chose to use as my final model is a radial basis function neural network. Radial basis function neural nets operate somewhat differently than multi-layer perceptrons. A RBF neuron accepts inputs as the distance between the input vector and the neurons current weights. These distances are then sent to the radial basis function of the neuron and the output is calculated. The radial basis networks in matlab toolbox have a transfer function such that their maximum output is 1 when the distance between their weights and the input function is zero. The output of the radial basis neuron becomes the input of the linear output neuron. The spread parameter determines the minimum distance the input needs to be away from the weights for the

input to the neuron to become the spread value. This helps keep consistency in the network response and supposedly helps with the network adapting to rapidly fluctuating functions.

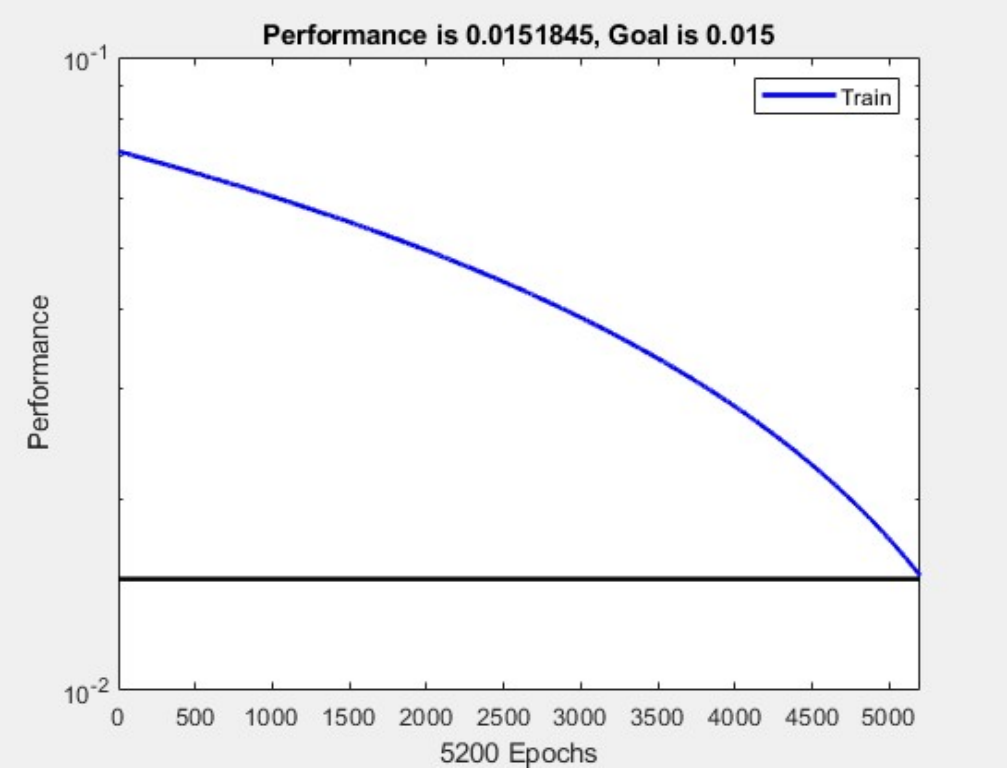
The RBF adds neurons over time to increase the accuracy of the network. The more neurons there are, the higher the likelihood that an input vector is close to the neuron's weights, causing it to fire. The more neurons that fire due to an input, the more accurate the networks response to that input will be. In this way, neurons are increased until there are enough neurons to accurately describe all the vector inputs in the training set to a desired measure of accuracy. Note, however, that this typically involves creating networks that are much larger than traditional multi-layer perceptron networks.

I. Hyperparameters used and why

The matlab toolbox function that I used to solve my problem was newrb. This function has 6 arguments and 4 hyperparameters. The hyperparameter defaults are: mse goal: 0.0, spread: 1.0, maximum number of neurons: the default value is hardware and application specific, and number of neurons to add between displays: 25. The two hyperparameters that I changed for my model were mse goal and spread. The spread value that I used was 0.001 and I determined this experimentally by varying the value and noting the performance change. Of the values that I tried, 0.001 seemed to cause the best performance. I changed the mse goal because I have limited resources in time and in hardware. I determined what the minimum goal that I required to get good classification results on the squares was experimentally. The value that I finally chose was 0.015.

J. MSE Plots (incomplete)

This is the MSE plot of the final model during testing. The MSE plots of the other models were included in the project updates.



K. Validation Method

For my project, I did not use a special validation method to select a model such as k-fold cross-validation or leave-one-out. When comparing models, I simply varied the hyperparameter values and stored the performance metrics obtained from the matlab toolbox functions and later compared to find the best combination. To test the models, I had them predict labels over the entire dataset of images and noted its performance.

L. References

The information and knowledge required to complete this project were provided to me by Dr. Cihan Dagli through his instruction and visual aids.

Haykin, S. "Neural Networks and Learning Machines". 3rd edition. Pearson: Upper Saddle River, New Jersey. 2009.

https://www.mathworks.com/help/deeplearning/ug/radial-basis-neural-networks.html