April 2018

Non-Invasive Neural Controller

Patrick Polley
Worcester Polytechnic Institute

Walter Gage Gallati
Worcester Polytechnic Institute

NON-INVASIVE NEURAL CONTROLLER

A Major Qualifying Project

Submitted to the Faculty of

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

in

Robotics Engineering

By

Walter Gallati

Patrick Polley

Advisor: Marko Popovic

Co-Advisor: Joseph Beck

Table of Contents

Table of Figures

Abstract

With the prevalence of microcontrollers in modern society, the field of prosthetics has advanced rapidly towards neuro-electric control systems. Utilizing technologies such as electroencephalography (EEG), patients can exert a more holistic, natural control of artificial appendages. However, most of the work in the field of EEG control has been related to direct neural network processing- taking input data from an EEG cap, processing it through a deep neural net, and directly correlating that to a desired output for a limb.

This project seeks to evaluate alternative means of controlling a prosthetic (in this case, a hand) using EEG control. The project consists of four methods; an unsure-feedback neural network, a neural network which lets the user know where it assumes the user wants to go, if unsure; a neutrally-iterated tree, which stores a preset list of locations that the user moves between based on how intently they focus on a task; a continuously-trained neural network, which tries to assume the user's hand position and trains relative to that; and a direct neural network, as described above. The selected methods will be tested on a group of seven individuals, comparing the results of each to determine training efficiency, accuracy, and response time relative to each other on a universal platform.

Acknowledgements

The researchers would like to thank Dr. Marco Popovic, Dr. Joseph Beck, and Popovic labs for their support and guidance in this project. Printing assistance for the hardware was provided by Matthew Bowers and the WPI Rapid Prototyping Lab; their knowledge of and assistance with 3D printing was greatly appreciated and essential to our project. The researchers would also like to thank Evan King for his editorial, technical, and motivational assistance throughout this project. Finally, the researchers would like to thank Worcester Polytechnic Institute, without which we would not be here.

Introduction

Electroencephalography (EEG) is defined as the measurement of electrical activity in different parts of the brain (Oxford University). It has applications in a variety of fields, such as neuroscience, computer interfacing, and medicine. While EEG is a relatively new technology, it has grown rapidly because of a wide array of possible applications, particularly prosthetic limb design.

EEG can be used to offer a wide range of control to amputees with intuitive, nonphysical commands. A 2016 study published in Nature demonstrated the effectiveness of this type of prosthetic command. Using 62 EEG sensors (referred to as "nodes"), the researchers obtained between 85 and 93.1 percent accuracy of a robotic hand by developing an artificial intelligence to associate signals from the EEG sensors with particular hand motions. (Meng, et al., 2016).

Meng et al accomplished this by developing a neural network intended to recognize patterns found in the EEG inputs. Using the past signal patterns it had identified, the network was able to accurately predict the most likely desired hand position by comparing unseen input with old input.

Neural networks for prosthetic control have been attempted by many organizations. A 2015 Imperial College of London study obtained 80% accuracy on a prosthetic hand using only 31 EEG nodes distributed evenly across the skull (Walker, 2015).

While an effective design and implementation of a network is important, acquiring accurate sensor data is a prerequisite. To produce the most relevant sensor data, nodes are placed proximal to the most project-relevant areas of the brain. This practice is known as localization, and demonstrably increases accuracy, especially in near-subcortical processes occurring closer to the center of the brain (Song, et al., 2015). Data filtration is also an effective means of increasing accuracy, as it refines the data to only include relevant inputs.

While neural networks show promise, they do not have an effective active control scheme; users will find that the system behaves erratically without their explicit instruction to do so. Another method of control involves direct intervention from the user. EEG nodes are capable of reliably measuring the level of focus a user has. By instructing the user to associate levels of focus with limb positions, the user can control the limb by focusing with a specific intensity. Prior research at WPI's Popovic Labs has achieved an average accuracy of 80.25% across three distinct hand positions using only one EEG sensor (Saint-Elme, et al., 2017).

Despite the variance in the approaches to prosthetic control, there has not been much effort made to categorically compare them in a standardized testing format. As the hardware, software, testing environments, and goals differ greatly between studies, quantifiable comparison of these methods are difficult to substantiate. This project fills that void by evaluating different control schemes in a standardized, unitary environment.

Project Goals

The initial goal of this project was to evaluate and compare three control schemes ("Control Schema Goals", below) across four criteria (impulse registry, accuracy, time delay, and stability) on seven distinct hand positions. The project later added a fourth control scheme, the Continuous Neural Network, and a fifth evaluation criterion, accuracy deviation. Qualitative data on the schema was to be obtained by a three-question survey. The original test plan called for comprehensive testing of all control schemes on 30 subjects. Given the significant logistical challenges associated with conducting wide-scale human testing, however, tests were limited to seven per schema, and each subject tested on a single control scheme.

Evaluation Criteria

The evaluation criteria are the metrics which determine the effectiveness of a given schema.

- Registered Impulse: How often the hand moved to any position, even if incorrect.
- Accuracy: A percentage representing what portion of the attempts resulted in a correct hand motion.
- Time Delay: How long the hand took to move to position.
- Stability: How long the hand can remain in position.
- Accuracy Deviation: A derivative metric produced from the accuracy that measures the standard deviation of the accuracy at a given position.

Control Schema

The four control schema developed are evaluated based on the above criteria. They are all implemented separately, and implement a variety of different approaches.

- Neural Network: A program that, using a neural network, receives information from EEG nodes localized to the motor cortex and attempts to classify input as one of the available hand positions based on the input.
- Neural Tree: An adjustable program that allows users to define hand positions they want to move to based on the level of focus measured. Users can also set the program to move to different hand positions depending on the position they are starting from (akin to a tree data format)
- Unsure Network: A variant of the Neural Network, but alerts the user if the network's certainty in its prediction is below a certain threshold. If so, the user must confirm if the position is accurate before proceeding.
- Continuous Neural Network: The Continuous Neural Network is another derivative of the Neural Network. Unlike the Neural Network, however, the Continuous Neural Network captures not only positional data, but data points while the user is moving their hand.

Hand Positions

The hand position goals were the hand positions to be implemented to measure the above evaluation criteria. Hand positions are all measured moving from Open Palm position; Open Palm is measured moving from Hook Grip position. The seven positions are:

- Hook Grip
- Open Palm
- Peace Sign
- Pinch Grip
- Thumbs Up
- Finger Guns
- Active Index Grip

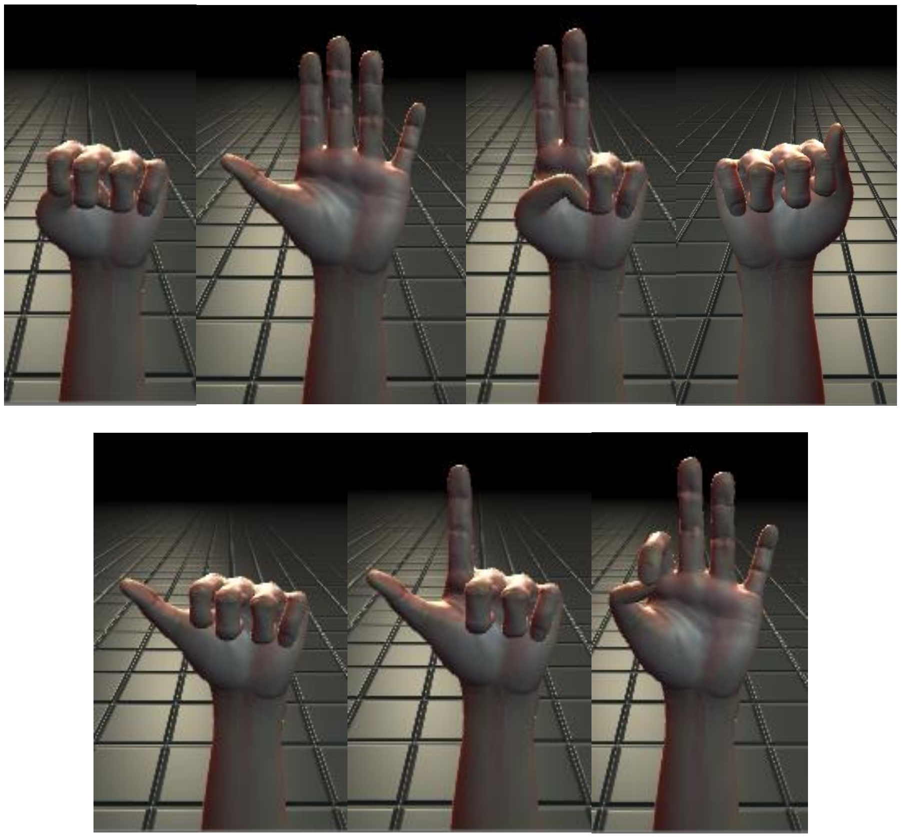


Figure 1: The seven hand positions: Hook Grip, Open Palm, Peace Sign, Pinch Grip, Thumbs Up, Finger Guns, and Index Grip

Survey Questions

Three survey questions were developed to gain a qualitative understanding of user comfort and preference regarding the control schema used.

- On a scale of 1-5, how natural did the arm feel?
- On a scale of 1-5, how much active thought was needed to control the arm?
- How long did you feel you took to learn how to control the arm?

Methodology

Hardware

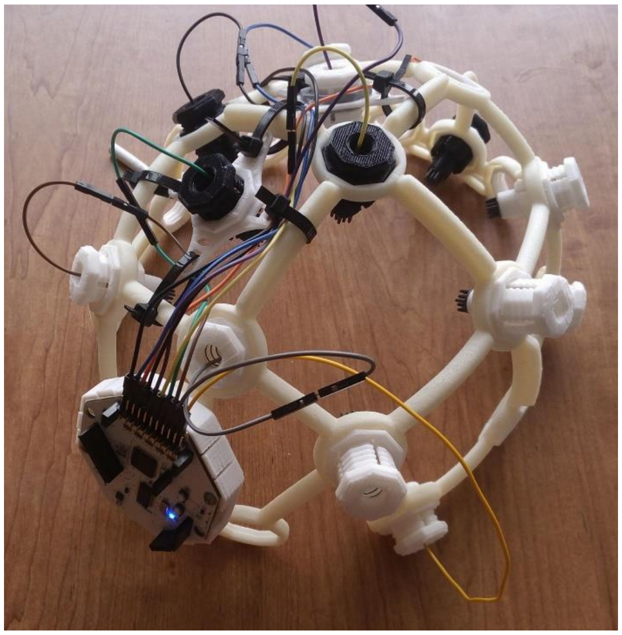


Figure 2: Completed Testing Headset

Headset

To properly conduct test comparisons of the various control schemes, the test equipment had to be standardized to the best ability possible. A rigid headset was chosen to allow for a consistent, firm node placement on the head. All test schemes would use the same helmet, board, and electrodes, though specific electrode count and location would vary based on the needs of the control schema tested.

The headset used was an OpenBCI Ultracortex MkIII Nova. Files for 3D printing the headset are provided free of charge by OpenBCI and are made to interface with OpenBCI products. It provides a rigid frame with modular positions for electrodes following the 10-10 and 10-20 electrode format (see Figure 5: Electrode locations in the 10-10 system. Measured nodes are in red while ground nodes are in yellow. Original image by Brylie Oxley under Creative Commons license). The modularity allows the electrode positions to be localized for their respective control schemes, while the rigid frame ensures proportional spacing for each subject regardless of head size. The frame of the headset was printed by the WPI Rapid Prototyping Lab with peripherals printed by Popovic Labs.

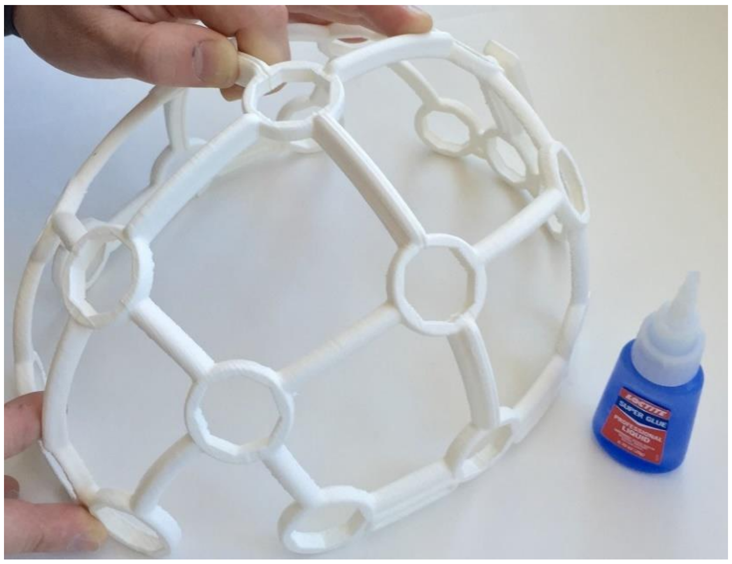


Figure 3: The OpenBCI Mk III Nova - http://docs.openbci.com/Headware/03-Ultracortex-Mark-III-Nova

Electrodes

The nodes used to obtain EEG signals in this project are TDE-210 EEG electrodes from Florida Research Instruments. They are dry electrodes that can operate through 5mm of flattened hair. Dry electrodes were chosen over wet electrodes as the motor cortex is located high up on the head and would be difficult to test without subjects shaving much of their hair. Electrode positions for the Neural Network, Unsure Network, and Continuous Neural Network were determined based on proximity to the motor cortex, specifically the location of motor function in the hand. For the Neural Tree control schema, which relies on focus, an electrode position is also placed at the prefrontal cortex (position FpZ) and is only connected for said schema. Ground nodes were placed in positions A1 (left earlobe) and P8 (proximal to the temporal lobe) to provide input voltage context to the microcontroller. These locations were chosen due to their distance from project-relevant neural processes. See Figure 5: Electrode locations in the 10-10 system. Measured nodes are in red while ground nodes are in yellow. Original image by Brylie Oxley under Creative Commons license for specific node locations.

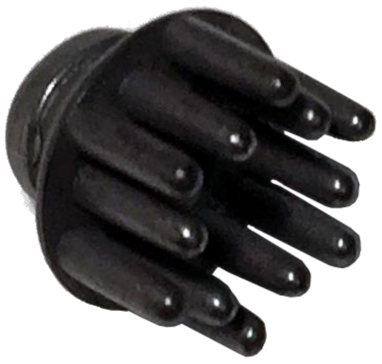


Figure 4: TDE-210 Dry EEG Electrode - https://fri-fl-shop.com/product/new-longer-5mm-spike-disposable-reusable-dry-eeg-electrode-tde-210/

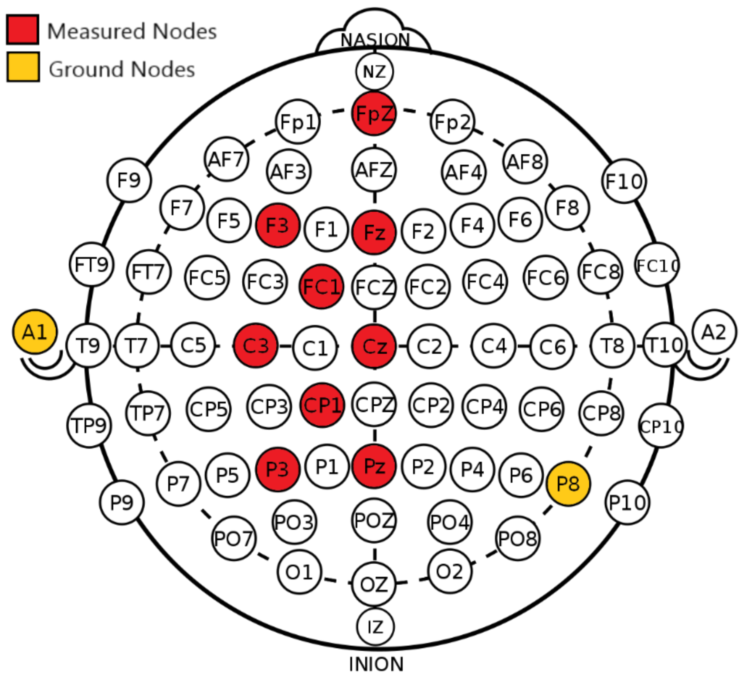


Figure 5: Electrode locations in the 10-10 system. Measured nodes are in red while ground nodes are in yellow. Original image by Brylie Oxley under Creative Commons license

Microcontroller

The microcontroller used was an OpenBCI Cyton. It is a 32-bit microcontroller that supports up to eight measured electrodes with two to three grounding electrodes. The Cyton board was selected due to its nodal measurement capabilities, predetermined interoperability with rigid headsets, and extensive support and documentation. The device collects sampling data at a rate of 250 Hz and transmits it wirelessly to a USB dongle. The sample rate for this project was increased to 500 Hz to facilitate filtration of high-frequency nodal signals.

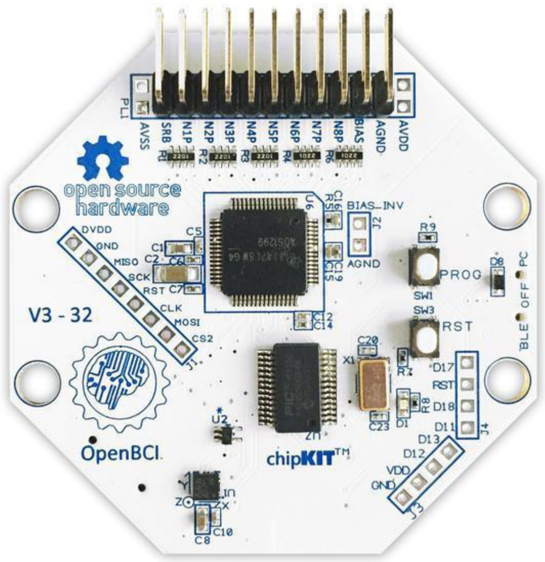


Figure 6: OpenBCI Cyton Biosensing Board - https://shop.openbci.com/products/cyton-biosensing-board-8-channel?variant=38958638542

Software

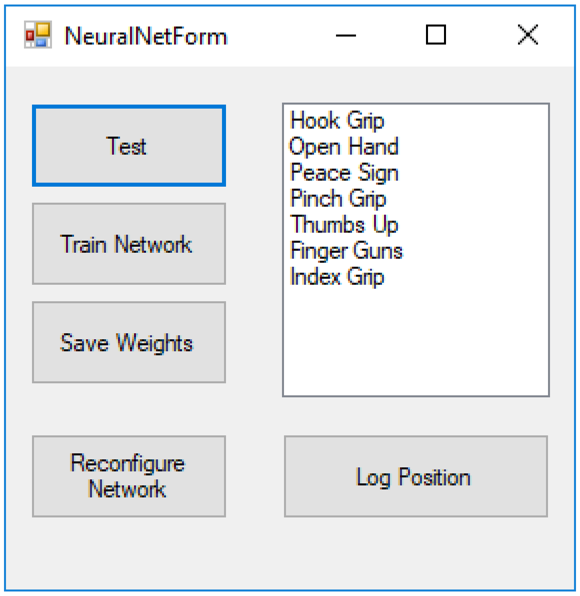
Data Processing

Data processing plays an important role in the uniform processing of EEG information. The OpenBCI Cyton sends data over a serial connection as a 33-byte packet consisting of a header, index number, set of eight three-byte node readings (one for each node), accelerometer data, and a footer. This information must be caught, processed, and stripped to obtain usable data.

Upon opening, the program waits for a header, then begins to log serial data. If it gets something other than a header, it waits until a header is given to it. Once a header is received, the program collects the next 32 bytes of input data. The first 24 bytes (the node readings) are collected and split into eight sets of three bytes; the remaining data in the packet is discarded as the accelerometer data is not relevant to the project. As the node information is three bytes signed, it does not innately convert to the Int32 (four bytes) or Int16 (two bytes) data format. To compensate for this, the program uses a function published by OpenBCI to convert the input to Int32.

Filtration can also be used to increase accuracy by removing less relevant nodal input. Filtering is done after node information is converted to Int32 and is performed to isolate relevant nodal signals. Motor activity is found to be most significant at 76-100 Hz (Miller, et al., 2010), while active thought and problem solving is best captured at 13-30 Hz (West Pomeranian University of Technology, 2014). By only measuring amplitudes in these frequency bands, data can be tailored to the specific control schema. Modifying code publicly provided by GitHub user nekrodezynfekator (nekrodezynfekator), the program runs two band-pass filters on the data: one reads amplitudes only in the 15-50 Hz range to isolate conscious thought, while the other reads amplitudes in the 75-100 Hz range to isolate motor activity. The filtered data from both are appended into the same array and stored in the SerialReader class, where it can be accessed by control schema programs.

Neural Network



Hook Grip
Open Hand
Peace Sign
Pinch Grip
Thumbs Up
Finger Guns
Index Grip

Figure 7: The Neural Network UI

The Neural Network control scheme utilizes a neural network and eight localized EEG nodes to predict the position a user wishes to move their hand to. As the schema is measuring motor activity, the program retrieves node data filtered to the 75-100 Hz range from the SerialReader class. The neural network used in the Neural Network control scheme is from the Accord module for C#. The code used for the Neural Network can be found in Appendix B.

Neural Net structure

The structure of the Neural Network consists of a fixed input and output size (eight, the filtered node input, and seven, the number of hand positions involved in this test), along with an indiscriminate number of hidden nodes. By default, the program has three hidden layers consisting of 100 nodes each. Preliminary testing versions only had 10 nodes per layer; that number was increased after early testing results suggested the network was not adequately sized.

The Neural Network program contains a function to perform K-Fold Cross Validation, which can reshape the network to best suit the training inputs and outputs required by the user. To reshape the network, the user clicks "Reconfigure Network". This procedure runs K-Fold Cross Validation using the data collected in training, splitting it into 80% for training and 20% for testing. Once complete, it replaces the old neural network with a new one shaped optimally based on the training data.

Training the Neural Network

The Neural Network needs to be trained on a user-specific data set before it returns nonrandom hand positions. To train the Neural Network, the user selects a hand position in the list menu and clicks "Log Position". When "Log Position" is clicked, the user moves their hand to the selected position from Open Hand (unless they are moving to Open Hand, in which case they move from Hook Grip). The Neural Net records 100 samples from the eight localized EEG nodes over 0.5 seconds and adds them to a list of arrays, with the position being moved to stored in a separate list under the same indexes. Once sufficient data has been collected, the user can train the Neural Net by clicking the "Train Network" button. When this button is clicked, the Neural Network runs for 1000 training epochs, using the Accord ResiliantBackPropagation class. The default learn rate is 0.1. Both the number of training epochs and the learn rate are evaluated in K-Fold Cross Validation and are changed based on the validation outcome.

Saving the Neural Network

All the data in the Neural Network (the neural network, K Fold values, and collected input/output data) can be saved and loaded. To save data, the user clicks "Save Data". All information is saved to and read from in the folder C:/BCIDataDirectory. Information is automatically loaded when the program starts, and initializes with aforementioned default values if no files are available.

Using the Neural Network

To predict a hand position with the Neural Network, the user clicks the "Test" button in the upper left-hand corner, then moves their hand from Open Hand to the desired position. The headset collects filtered data from the localized EEG nodes, and inputs it into the neural network. The network outputs the probabilities the user intended to move to each location, and selects the one with the highest probability. This is then output to the shared file system.

Neural Tree

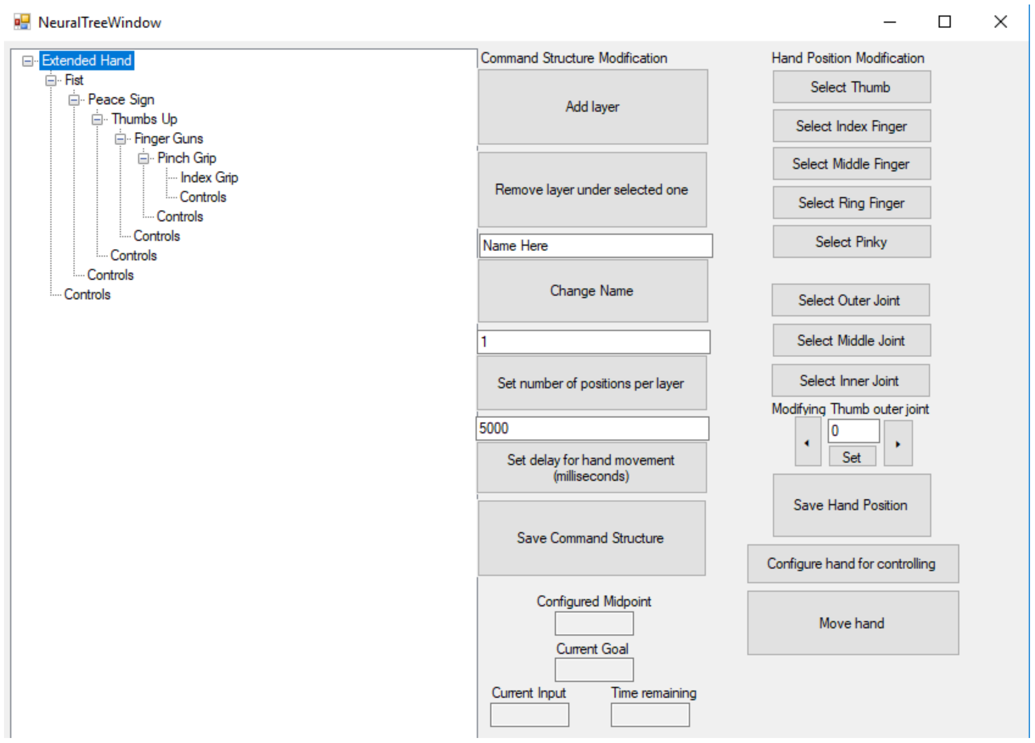


Figure 8: The Neural Tree UI

The Neural Tree control schema is the outlier of the four tested, deriving its operation through focus intensity in the prefrontal cortex rather than motor neuron firing in the motor cortex. As such, it is the only schema that reads from the 15-50 Hz band. The code used for the Neural Tree can be found in Appendix B.

Tree Configuration

The Neural Tree schema uses the amplitude of filtered nodal input relative to a maximum and minimum input to select a certain hand position in a list of hand positions. To obtain a maximum and minimum for each user, the user selects “Configure hand for controlling” in the lower right side of the window. This creates a popup that tells the user to let their mind wander; when the user clicks “Okay”, the Neural Tree program records node activity for a period of five seconds, averages it, and stores it as the minimum focus value. It then prompts the user with a popup to focus intently on something. When the user clicks “Okay”, the program records node activity for five seconds, averages it, and stores it as the maximum focus value. If the maximum value is lower than the minimum value, it starts the process over again by prompting the user to let their mind wander. If not, it informs the user that it has been properly calibrated, and displays the focus values in the Current Max and Current Min boxes on the bottom of the window.

Using the Neural Tree

Hand positions in the Neural Tree are displayed on the lefthand side of the window in a dropdown menu format. Each layer has a set number of positions it can move to, with a default of 4 positions. The number of positions can be changed by entering the desired number in the text box above “Set number of positions per layer” and clicking the button. Layers can be added under a hand position by selecting the hand position and clicking the “Add layer” button. In the same way, a layer can be removed by selecting the parent position and clicking the “Remove layer under selected one” button. To select a hand position in a layer, the user clicks the “Move hand” button. Once clicked, the Neural Tree program reads the filtered node data for the desired delay time (200 milliseconds by default) and averages it. The delay time can be changed by entering the desired time in milliseconds in the text box above “Set delay for hand movement” and clicking that button. The average amount of focus is displayed in the Current Goal box to facilitate biofeedback in training. Once the delay time has elapsed, the final average is converted to a percentage of the user’s maximum focus; this percentage is then used relative to the number of positions per layer to determine the desired hand position. On the default 4 positions, any percentage range 0-25% returns the first hand position, 25-50% returns the second hand position, and so on. If a hand position has a layer of other positions underneath it, it informs the user and runs the process again; if the same hand position is selected (closest to max focus), the hand stays in the position that it is and the processing ends.

Saving the Neural Tree

Positions entered in the Neural Tree, along with number of positions per layer and iteration time, can be saved by clicking the "Save Command Structure" button. Hand positions can be edited by selecting the position, phalange, and joint to edit, then entering the joint angle in the text box above the "Set" button and pressing "Enter". Hand positions can be saved by clicking the "Save Hand Position" button; if not pressed, selecting another hand position will undo all unsaved work.

Unsure Network

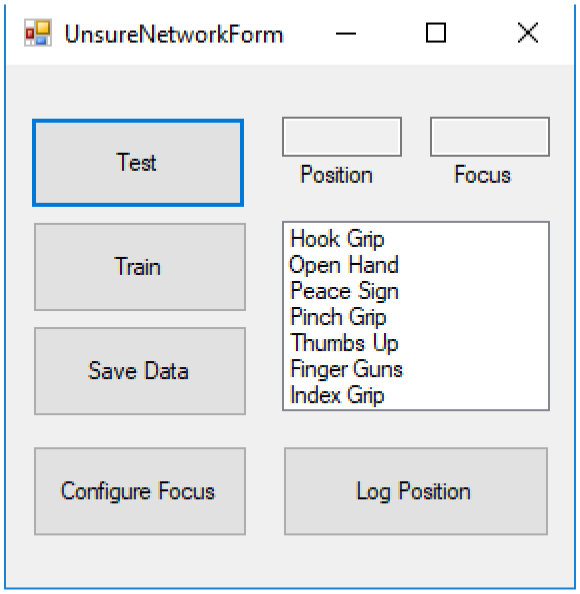


Figure 9: The Unsure Network

The Unsure Network is a combination of the Neural Network and the Neural Tree. Identical to the Neural Network schema in inputs, outputs, structure, training, and stored data, the program operates akin to the Neural Network until it is unsure if it predicted the right hand position. At that point, it uses the focus controls used in the Neural Tree to allow the user to confirm or deny the hand position in question. The program uses seven nodes operating in the 75-100 Hz band and a singular "focus node" operating in the 15-50 Hz band.

Unsure Network Configuration

Configuration of the Unsure Network is similar to that of the Neural Tree. Using the focus node, the program collects a maximum and minimum focus value for each user by having the user click the "Configure Focus" button. This creates a popup that tells the user to let their mind wander. When the user clicks "Okay", the Unsure Network program records the focus node activity for a period of five seconds, averages it, and stores the average as the minimum focus value. Once stored, the program creates a popup telling the user to focus intently on something. Upon clicking "Okay", the program records node activity, averages it, and stores the average as the maximum focus value. If the maximum value is lower than the minimum value, it starts the process over again by prompting the user to let their mind wander. If not, it informs the user that it has been properly calibrated.

Unsure Network Operation

The Unsure Network is operated similarly to the Neural Network. To predict a hand position, the user clicks "Test", then moves their hand from the Open Hand position to the desired position. The headset collects filtered data from the EEG nodes, and inputs it into the network. The network outputs the probabilities the user intended to move to. If the highest value is below a certain confidence threshold (in this case, 95% confidence), the program prints the position it assumes to be moved to in the "Position" text box. The user has 5 seconds to position their focus level above or below the threshold: the current focus level, and whether or not that will confirm or deny the position, is displayed in the "Focus" text box. Once a position is confirmed, the program moves to that position. The program continues until a position is confirmed or all possible positions are denied; if this happens, the hand does not move.

Continuous Neural Network

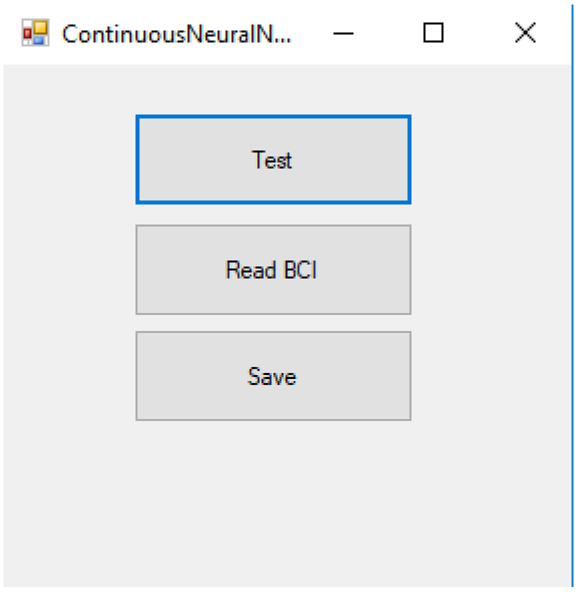


Figure 10: The Continuous Neural Network UI

The Continuous Neural Network is identical to the Neural Network schema in inputs, outputs, structure, use, and stored data. The only difference between the two is the training process.

Continuous Neural Net Training

The Continuous Neural Net is trained based on its assumptions. Once training is initiated by the user, the schema moves to the Hook Grip position. The user then moves to that position as the program reads and stores the user's node activity created by movement. It then repeats this for the remaining six hand positions. Once all of these positions are logged, the network enters the last logged input into the neural network and outputs the highest probability position. The user then moves to that position as the program records the node activity. This node data is stored and used to briefly train the neural network using the Accord ResilientBackPropagation class. The program then takes the last logged input and repeats the process. This is intended to provide a sort of biofeedback to the user, allowing them to alter their behaviors in a way conducive to more accurate training. Prior to preliminary testing, the program started with a random position; after poor performance in preliminary tests, it was modified to sample all positions first so it would not leave out un-sampled positions.

Unity Output

To mimic a prosthetic limb, a program was created in Unity game engine. The program was chosen to provide visual feedback for control in a more stochastic environment. Actual prosthetics were not used as mechanical failure could muddle results. Control schemes communicated with Unity through the use of the UnityCommunicationHub class. The class communicates with Unity through the transferInfo.txt file in the C:/BCIDataDirectory folder; when the class wants to send data, it edits the transferInfo.txt file with the joint positions it wants to send, deletes an existing WFATurn.mutex file if it exists, and creates a new file titled UnityTurn.mutex. When Unity discovers that the UnityTurn.mutex file exists, it opens transferInfo.txt, sets the joints in the model accordingly, then writes its current position to transferInfo.txt, deletes UnityTurn.mutex, and creates a new WFATurn.mutex file. The UnityCommunicationHub class reads this, stores the current position, and waits until it needs to send information again. Given that desired joint positions are saved prior to processing in the UnityCommunicationHub, the program can be easily modified to output to any file format required for a prosthetic.

Testing Procedure

Test Preparation

Testing was performed on a group of 8 test subjects. Prior to arrival at testing, each subject was assigned a control scheme by random dice roll. Preliminary tests were assigned by rolling a six-sided die and dividing by 2, rounded up (implementation of Unsure Neural Network was incomplete at time of testing). Final tests were assigned by rolling an eight-sided die and dividing by two, rounded up. All wires connecting electrodes to the Cyton board were unplugged, and all electrodes were unscrewed to the maximum allowable sizing configuration for the headset.

Upon arrival, subjects were briefed on their rights as a participant of the study and were allowed to answer any questions regarding the study or their role. Once all questions were answered satisfactorily, the test subjects were seated, and the headset was placed on their head according to the positioning displayed in Figure 5: Electrode locations in the 10-10 system. Measured nodes are in red while ground nodes are in yellow. Original image by Brylie Oxley under Creative Commons license Electrodes in positions P8 and F7 were screwed in first, and continued to be until seated firmly against the test subject's skull. Electrodes in position P7 and F8 were then screwed in until seated similarly firm. The test subject was then asked if they were uncomfortable with the tightness; if so, the electrodes were adjusted until the subject felt comfortable. With the headset now locked into place on the head, the ear clip electrode was then placed in position A1. From there, the Cyton board was turned on and the OpenBCI USB dongle inserted into the testing computer. To ensure node connectivity, OpenBCIHub.exe and OpenBCI_GUI.exe, programs both provided by OpenBCI, were run. From there, the setup procedure differed depending on control schema. For the Neural Tree, the electrode in position FpZ was connected to input 2 on the Cyton board and slowly screwed in until readings were displayed in the OpenBCI GUI. For all other control schema, the electrode in position F3 was connected to input 1 on the Cyton board and slowly screwed in until it displayed readings in the OpenBCI GUI. This process was repeated for the nodes Fz, C3, FC1, Cz, CP1, P3, and Pz in order. Once all nodes read properly and consistently in the OpenBCI GUI, testing on the individual schema began.

Test Procedure

All test procedure began with opening and running the Unity program, followed by opening the program to the designated control schema. Once the respective window was opened, the test subject was briefed in its use as outlined in the Methodology section, as well as briefed on test procedure. The subject was then allowed to ask questions regarding operation of the schema. Once all questions were satisfactorily answered, testing began.

Neural Net

The Neural Net testing procedure began with training the data set. The test subject was instructed to raise their hand in a manner they felt comfortable holding for an extended period of time (often resulting in the subject resting their elbow on the table). Maintaining the hand in position reduced the number of confounding variables in the study. Once their hand was raised, the test subject was instructed to move to Open Palm position. From there, they were instructed to go to Hook Grip when they hear a click and remain there. The test administrator then audibly clicked the "Log Position" button, and the test subject moved to position. After one second, the test administrator gave the command "Release" to let the subject know to return to the Open Palm position. This was repeated four more times. Once the fifth Hook Grip was logged, the test subject remained in Hook Grip and the process was repeated for Open Palm. This process was repeated for all hand positions, starting from Open Palm and transitioning to the logged position, such that each hand position had been logged five times. Once complete, the process started over again with Hook Grip, and continued until all positions had been logged 10 times. Once this was complete, the "Train Network" button was pressed to train the network with the stored data. The time taken from the beginning of data logging to the end of network training was recorded. Following completion of network training, the "Save Network" button was pressed to ensure a backup of the neural network and the training data. The test subject was then instructed to go to the Open Palm position. Similar to the training of the Neural Network, the subject was instructed to go to Hook Grip when they heard a click and remain there until told to release their grip. The test administrator then audibly clicked "Test", recorded the position the hand moved to given the position the test subject attempted, then gave the test subject the command to release their grip. Similar to training, the process was repeated four more times, then performed five times for every remaining hand position. Once all positions had been attempted five times, the test subject started over again at Hook Grip, and attempted each position another five times. The attempted position and the position moved to for each was recorded, and the test concluded.

Neural Tree

The Neural Tree testing procedure began with the test subject configuring the program with their minimum and maximum focus values. Once complete, they were allowed to practice up to three hours or until they felt comfortable enough in their skills to operate the Neural Tree. The time between the completion of configuration and the beginning of the testing period is recorded. In this time they were allowed to change the order of hand positions in the tree, the number of positions per layer, the location of layers, and the time delay for aggregating and averaging input data. They were also allowed to reconfigure the program, but the time would not restart should this be the case. Once ready, the subject was instructed to move to Hook Grip. The subject would then click "Control hand", with the final resting position of the hand being recorded. This was continued for another four times, then performed five times for every remaining hand position. Once all positions had been attempted five times, the test subject started over again at Hook Grip, and attempted each position another five times. The attempted position and the position moved to for each was recorded along with the final time delay used by the test subject. Once all data was collected, the test was concluded.

For the second iteration, the tree control schema was modified into something of a list of positions with binary inputs. Instead of being able to access any number of positions from some position, each position was linked to another in a list. During the operation of the hand, the program would iterate through every hand position and stop on some position when the user focused. This allowed the user to remain in low focus until the program highlighted the position they wanted to move to.

Unsure Network

The Unsure Network schema began with training the neural network. This process was performed identical to the Neural Network schema, where each position was logged 10 times by the test subject and used to train the neural network. Following that, the "Configure Focus" button was pressed by the test subject, and maximum and minimum focus values set. Once complete, testing was performed the same way as the Neural Network, with each position being attempted by the test subject five times with the results recorded, followed by each position being attempted and recorded another five times for a total of 10 attempts per position.

Continuous Neural Network

Similar to the Neural Network schema, the Continuous Neural Network began with the test subject instructed to raise their hand in a manner they felt comfortable and to move to the Open Palm position. Once situated, the test subject was then told to click the "Train" button when ready. When clicked, the hand moves to Hook Grip, which the user follows. The program then moved the Unity model hand to Open Palm position, then to Peace Sign, and continued through the seven hand positions until all had been trained once. Every time the hand moved, the test subject moved their hand to match position. Once all positions had been iterated through, the Continuous Neural Network moved the model hand to the position it believes the hand to currently be at. The user then moved to that position, or, if the hand location was the same as the user currently was, remains in the same position. The Continuous Neural Network then processed the information and moved the model hand to the position it believes the user to be in based on the most recent data. The process continued for 20 iterations. Once completed, testing proceeded identically to the Neural Network scheme; from Open Palm, the test subject was asked to move to Hook Grip when they heard a click and remain in that position until told to release their hand. The test administrator then audibly clicked "Test", recorded the position the hand moved to given the position attempted, then gave the test subject the command to release their grip. This process was performed five times for each hand position, then performed another five times for each hand position for a total of 10 attempts for each hand position. Attempted positions and the actual position moved to by the control schema were recorded, and the test concluded.

For preliminary testing, the initial seven hand positions were not included, and the program initiated with a single random hand position. From there, it would go to the position it believed to be accurate. This lead to certain positions not getting trained, resulting in poor accuracy.

Post Test Procedure

At the conclusion of testing, the test subjects were thanked for their participation and asked to fill out a survey consisting of the following:

- On a scale of 1-5, how natural did the arm feel?
- On a scale of 1-5, how much active thought was needed to control the arm?
- How long did you feel you took to learn how to control the arm?

Survey results were recorded with the user data. Survey results were aggregated and the average values used to evaluate user perception of each scheme.

Different survey questions were asked in preliminary testing to gather general quantitative data about the control schemes for future improvement. The questions asked were:

- How well they thought they did
- How comfortable they were while training/testing
- Ways they feel the scheme or testing could be improved
- Things they think they did well relative to others
- Things they think they did poorly relative to others

Preliminary survey responses can be found in Appendix A, Preliminary Testing Data.

Test Results and Data Analysis

Preliminary Testing

Scheme	Average Accuracy	Average Standard Deviation
Neural Tree	26.67%	34.75%
Neural Network	10.95%	16.30%
Continuous Neural Net	15.71%	31.36%

Table 1: Scheme Accuracy and Standard Deviation in Preliminary Testing

Based on the data collected (see Appendix A, Preliminary Testing Data), the Neural Tree was the most accurate of the control schemes, with an average accuracy for any given hand position of 26.67% and a peak accuracy of 100% with certain hand positions for certain subjects. The Neural Net and the Continuous Neural Net fared worse, with 10.95% and 15.71% accuracy respectively. Given the accuracy of randomly assigning hand positions is 14.29%, the Neural Net performed worse than random selection, and the Continuous Neural Net fared only marginally better. Given the similar values, the discrepancy appeared to be an issue with the neural network's shape and learning characteristics.

To determine the source of error in the Neural Network, a test environment was used to train independently on saved input and output data from user tests. The test environment used was provided by GitHub user Ares513 using the Python Keras module (King). Evaluating across multiple network depths and breadths, using different loss functions, the library was able to achieve between 18 and 36% accuracy depending on the dataset.

As mentioned above, the Neural Tree had the highest accuracy, with a peak of 100% in the hand positions assigned to maximum and minimum values. Mid-tree positions, however, were not as easily reachable, obtaining between 0% and 40% accuracy. As the focus control allowed for such high accuracy at distinctive points, changing the tree to operate using binary commands could allow for it to perform with similar high accuracies but across all points.

Despite the inaccuracies, the Neural Network scheme has a significantly lower standard deviation than the Neural Tree scheme, indicating a more even distribution of positional accuracy, even if lower. In this regard, improvements to the accuracy of the neural network may lead the program to exceed the Neural Tree in usability. The Continuous Neural Network, however, possesses a standard deviation near that of the Neural Tree, indicating discrepancies in differentiating between types of nodes. This was likely due to node training, as certain positions during training were never reached, resulting in the network excessively focusing on certain positions and completely ignoring others.

Survey Results

Analysis of survey responses return a common theme of disappointment. All but two test subjects reported feeling that they performed poorly - however, none suggested that the training time was inadequate or that it should be extended. The Neural Tree users found that they were easily able to move to two or three positions (most often Open Palm and Hook Grip), while neural network-based users did not feel particularly effective moving to any specific position. More accurate hardware would likely return the best net benefit based off of survey responses, as it would provide for better data for schema processing without any change in training strategy or time.

Final Results

Scheme	Avg. Accuracy	Avg. Standard Deviation	Avg. Impulses per Position	Avg. Stability per Position	Time Delay
Neural Network	23.57%	23.62%	6.71	2.79	0.5s
Neural Tree	78.57%	18.64%	2.43	6	20s
Unsure Network	61.43%	21.21%	5.28	3.79	20s
Continuous Neural Network	14.29%	37.80%	0	10	0.5s

Table 2: Final Calculations

Based on the data collected (see Appendix A, Final Testing Data), the Neural Tree was the most accurate of the four control schemes tested. Evaluations were broken down as follows:

- Accuracy: Percentage of times scheme moved hand to position attempted by test subject.
- Standard Deviation: Standard deviation between accuracy values for each position attempted by each test subject.
- Impulses per Position: Number of times hand position changes while attempting same position.
- Stability per Position: Longest sequence of unchanged positions (accurate or not) while attempting same position.
- Time Delay: Calculated program time to move to position from test administrator command to move.

Given the above data, the accuracy appears independent of the impulses and stability per position. The Neural Tree has the second smallest average impulses per position and second largest average stability per position, yet the Continuous Neural Network has the smallest impulses per position and the largest average stability per position and performs no better than a randomly assigned hand position. The Neural Tree, however, has an average accuracy of 78.57%, with some hand positions achieving 100% accuracy.

The Neural Network performed better than it did in preliminary testing, more than doubling its accuracy from 10.95% to 23.57%. Reducing the network input to only process 75-100 Hz, along with reducing the training epochs to 300 from 1000, removed noise and mitigated the threat of overfitting the neural network. Standard deviation increased in the process, however, implying that a ground between the two will return more consistent results at better accuracies.

The Neural Tree drastically increased in functionality, almost tripling in accuracy from 26.67% to 78.57%. This can be attributed to reducing the tree to a binary, which allowed for more accurate control by the user. The average standard deviation was also reduced from 34.75% to 23.62%, likely a result of all nodes being assigned easily-achievable focus ranges as opposed to varying intensities on the earlier tree using a wider brachiating factor.

The Unsure Network, while not evaluated in preliminary testing, performed remarkably well given the previous performance of its parent, the Neural Network. The high accuracy can be attributed to a conscious "fact-checking" by the test subject, where errors can be caught and dealt with before the hand can move. Though slower than the Neural Network, the program should be able to exceed the speed of the Neural Tree as it can "preselect" a position instead of having to wait its way through a tree. The speed of the Unsure Network, as such, relies on the accuracy of its neural network program.

The Continuous Neural Network was the outlier in the final testing, having decreased rather than increased in accuracy (14.29%, compared to 15.71% in preliminary testing). The changes made after preliminary testing, to reduce the training rate and ensure the program iterates through every hand position during training, did not have the intended effect, instead overtraining the network to the point where it only returns a singular value.

Survey Results

Scheme	Naturality of Control	Thought Required
Neural Network	2.75	2.5
Neural Tree	4	5
Unsure Network	4	2.5
Continuous Neural Network	1	2.5

Table 3: Survey Results

Results from the survey indicated a distinct difference in how natural each control scheme feels to use, but less of a difference in the amount of thought required. The Neural Tree and Unsure Network, the two control schemes with the greatest accuracy, were both considered the most natural to control, with the Unsure Network requiring half as much thought (qualitatively) as the Neural Tree. The Neural Network and the Continuous Neural Network both had as much thought required, but the Neural Network was reported as feeling more natural to use. The naturality of control appears to scale relative to how well test subjects did, while the thought required appears to be based distinctly on operation (while the network-based programs require the user to move their limbs to control the prosthetic, the Neural Tree requires the user to focus). This implies that the perceived level of thought required to control the prosthetic is based solely upon the mechanics of how input is obtained, while control of the prosthetic is deemed more natural the more accurate the control scheme is. Potentially, "natural" control of prosthetics can be unrelated to actual motor function, as long as the scheme operates successfully.

Time-based survey results seem to have similar opinions on time spend relative to their schemes, but not between them. Neural Network test subjects did not believe that they spent much time learning their control schemes, while Continuous Neural Network subjects believed they spent more time than they did (one test subject claimed that they spent 20 minutes reviewing, when in reality they only spent three). The Neural Tree user, meanwhile, felt the learning process only took five minutes while it instead took upwards of 15. Unsure Network users understood that they spent time, but believed that they recognized how to use the scheme earlier than expected. The amount of time each believed they spent on the network appears to factor into perceived naturality of control.

Conclusion and Future Work

In conclusion, the Neural Tree was the most successful from an accuracy and consistency standpoint, with over 75% average accuracy across all seven hand positions. The Continuous Neural Net was a success in the evaluations of impulses registered and stability, but was not successful in terms of accuracy, with a success rate equal to that of a random distribution. The Unsure Network achieved similar success to the Neural Tree, while the Neural Network, though underperforming, made drastic improvements between iterations. As such, the researchers may conclude that the Neural Tree, Unsure Network, and Neural Network control schemes can be effective controllers for prosthetic appendages, while the Continuous Neural Network control scheme is unlikely to be successful. Furthermore, perceptions of neural network ease-of-use appear to be affected by success rate, while the amount of perceived conscious effort relies on the methods involved.

For future work, testing on superior hardware would provide great benefit. Improvements observed in the Neural Network showed that isolating frequencies to remove noise greatly benefits the success rate of the schema. More inputs with increased accuracy in measurement could drastically improve performance. Furthermore, research into more accurate filtering may improve results by isolating important signals even more. Increasing the number of nodes for the Neural Tree averaging may also be of benefit, as it may detect more relevant firing of electrodes. If this option is pursued, the implementation of a neural network in the Neural Tree inputs may prove beneficial for additional data processing. Finally, the improvement of the underlying neural network program would prove beneficial to research, as improvements in that field can better detect subtle patterns in EEG inputs.

Bibliography

King, E. (n.d.). *NonInvasive Neural Controller.* Retrieved from GitHub: https://github.com/Ares513/NonInvasiveNeuralController

Meng, J., Zhang, S., Bekyo, A., Olsoe, J., Baxter, B., & He, B. (2016, December 14). Noninvasive Electroencephalogram Based Control of a Robotic Arm for Reach and Grasp Tasks. *Nature*. Retrieved from https://www.nature.com/articles/srep38565

Miller, K. J., Schalk, G., Fetz, E. E., den Nijs, M., Ojemann, J. G., & Rao, R. P. (2010, March 2). Cortical activity during motor execution, motor imagery, and imagery-based online feedback. *Proceedings fo the National Acadamy of Sciences of the United States of America*, 4430-4435.

nekrodezynfekator. (n.d.). *OpenBCI_GUI.* Retrieved from GitHub: https://github.com/nekrodezynfekator/OpenBCI_GUI

Oxford University. (n.d.). Electroencephalography. *Oxford English Dictionary*. Oxford University Press.

Saint-Elme, E., Larrier, Jr., M. A., Kracinovich, C., Renshaw, D., Troy, K., & Popovic, M. (2017). Design of a Biologically Accurate Prosthetic Hand. *International Symposium on Wearable Robotics.* Houston, Texas (USA).

Song, J., Davey, C., Poulsen, C., Luu, P., Turovets, S., Anderson, E., . . . Tucker, D. (2015, Decembeer 30). EEG Source Localization: Sensor Density and Head Surface Coverage. *Journal of Neuroscience Methods*, 9-21. Retrieved from https://www.sciencedirect.com/science/article/pii/S0165027015003064

Walker, I. (2015). *Deep Convolutional Neural Networks for Brain Computer Interface using Motor Imagery.* London: Imperial College of London. Retrieved from http://www.doc.ic.ac.uk/~mpd37/theses/DeepEEG_IanWalker2015.pdf

Wentrup, M. G., Gramann, K., Wascher, E., & Buss, M. (2005). EEG Source Localization for Brain-Computer Interfaces. *2- International IEEE EMBS Conference on Neural Engineering* (pp. v-viii). Arlington, VA: IEEE. Retrieved from http://www.kyb.tuebingen.mpg.de/fileadmin/user_upload/files/publications/GrosseWentrup2005_NeuralEngConf_[0].pdf

West Pomeranian University of Technology. (2014, February). Processing and spectral analysis of the raw EEG signal from the MindWave. *Przeglad Elektrotechniczny*, 169-173.

Appendix A: Final Testing Data

	1	2	3	4	5	6	7
Subject 1	2	2	1	5	7	2	2
Neural Network	1	2	2	5	2	2	1
Gender: M	2	7	2	2	1	2	6
Age: 22	2	2	4	2	2	2	2
Training Time: 15:32	6	2	2	6	5	2	5
Survey Results:	3	2	3	2	2	2	7
Naturality: 2.5	2	2	2	2	1	2	7
Thought Required: 3	1	2	2	1	1	2	7
It didn't take too long to learn, arm movements simple	7	2	3	2	5	3	6
	2	2	6	7	1	2	2
Impulses	8	2	7	6	8	2	7
Stability	2	7	2	2	2	8	3
Accuracy	20.00%	90.00%	20.00%	0.00%	20.00%	0.00%	30.00%
Subject 2	1	1	1	1	1	1	1
Continuous Neural Net	1	1	1	1	1	1	1
Gender: M	1	1	1	1	1	1	1
Age: 22	1	1	1	1	1	1	1
Training Time: 3:27	1	1	1	1	1	1	1
Survey Results:	1	1	1	1	1	1	1
Naturality: 1	1	1	1	1	1	1	1
Thought Required: 4	1	1	1	1	1	1	1
About 20 minutes	1	1	1	1	1	1	1
	1	1	1	1	1	1	1
Impulses	0	0	0	0	0	0	0
Stability	10	10	10	10	10	10	10
Accuracy	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Subject 3	1	7	3	1	3	6	7
Unsure Neural Net	1	2	3	4	5	3	7
Gender: M	6	4	3	4	1	7	3
Age: 21	1	6	3	3	3	7	7
Training Time: 22:21	1	2	3	4	3	1	3
Survey Results:	3	3	3	5	3	6	5
Naturality: 3	4	2	3	4	2	5	7
Thought Required: 2	1	2	3	7	7	6	7
[illegible] picked up on tendencies but took too long testing them	1	3	3	4	5	6	7
	1	3	3	3	5	1	7
Impulses	5	7	0	8	6	7	5
Stability	3	2	10	2	2	2	4
Accuracy	70.00%	40.00%	100.00%	50.00%	30.00%	40.00%	70.00%
Subject 4	2	2	3	2	5	6	7
Neural Tree	2	2	3	1	5	6	2
Gender: F	1	2	3	3	5	6	2
Age: 21	2	2	1	4	5	6	2
Training Time: 17:30	2	2	3	4	5	6	7
Survey Results:	1	2	3	4	5	6	7
Naturality: 4	1	2	3	4	1	6	7
Thought Required: 5	1	2	2	1	5	6	7
5 minutes	1	2	3	4	5	6	7
	1	2	3	4	5	6	5
Impulses	3	0	4	5	2	0	3
Stability	5	10	3	3	6	10	5
Accuracy	60.00%	100.00%	80.00%	60.00%	90.00%	100.00%	60.00%

Subject 5	1	2	3	1	5	6	6
Unsure Neural Net	2	2	3	4	4	6	7
Gender: F	1	2	3	4	5	6	3
Age: 22	1	6	3	6	6	6	6
Training Time: 42:00	4	2	3	4	5	6	5
Survey Results:	1	3	3	2	5	6	7
Naturality: 5	1	5	6	6	7	6	7
Thought Required: 3	1	7	3	6	2	6	3
See below*	2	2	2	4	5	6	7
	6	2	6	4	5	6	7
Impulses	6	6	4	6	7	0	7
Stability	3	3	6	2	2	10	2
Accuracy	60.00%	60.00%	80.00%	50.00%	60.00%	100.00%	50.00%
Subject 6	6	7	7	4	1	6	4
Neural Net	7	6	1	6	2	1	1
Gender: M	6	3	6	1	6	5	7
Age: 22	2	6	6	6	1	6	7
Training Time: 9:19	6	1	7	1	5	1	1
Survey Results:	6	2	1	7	1	1	6
Naturality: 3	1	2	7	1	2	6	1
Thought Required: 2	4	6	6	1	7	6	6
Not long at all. It w as easy	1	1	4	6	1	1	6
to pick up once explained.	1	6	4	7	7	6	7
Impulses	7	8	7	9	9	7	7
Stability	2	2	2	2	1	2	2
Accuracy	30.00%	20.00%	0.00%	10.00%	10.00%	50.00%	30.00%
Subject 7	1	1	1	1	1	1	1
Continuous Neural Net	1	1	1	1	1	1	1
Gender: M	1	1	1	1	1	1	1
Age: 22	1	1	1	1	1	1	1
Training Time: 12:13	1	1	1	1	1	1	1
Survey Results:	1	1	1	1	1	1	1
Naturality: 1	1	1	1	1	1	1	1
Thought Required: 1	1	1	1	1	1	1	1
I w as not able to control this	1	1	1	1	1	1	1
arm.	1	1	1	1	1	1	1
Impulses	0	0	0	0	0	0	0
Stability	10	10	10	10	10	10	10
Accuracy	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

*Test subject 6: Not that long in comparison to w hat I originally thought. It did require intensive focus, but I think it took approximately about 45-60 minutes to feel comfortable

Appendix B: Preliminary Testing Data

Raw Data

	1	2	3	4	5	6	7
1/Neural Net	4	3	5	5	5	5	5
	5	4	3	4	5	5	5
	3	5	5	5	5	5	5
	4	5	4	5	4	3	5
	4	5	5	3	5	5	3
	5	3	3	5	4	5	4
	3	5	4	3	4	3	5
	5	5	3	3	3	3	5
	5	5	5	5	3	5	3
	4	3	4	5	5	3	5
2/Neural Tree	1	2	1	1	4	2	2
	2	1	1	1	1	2	2
	2	1	3	1	3	2	2
	1	1	3	4	1	2	3
	1	1	1	3	1	5	1
	1	1	1	3	5	2	2
	1	2	1	3	3	1	1
	1	1	2	1	1	1	2
	1	1	1	1	1	1	1
	1	1	1	2	3	1	1
3/Neural Tree	1	2	2	3	7	3	2
	1	2	1	3	3	3	1
	1	2	1	3	3	7	3
	1	2	1	7	2	1	1
	1	2	3	7	1	2	2
	1	2	1	1	2	1	1
	1	2	3	3	6	2	1
	1	2	3	2	7	2	3
	1	2	3	3	2	2	1
	1	2	2	3	1	1	3
4/Neural Net	1	1	1	1	6	6	1
	1	1	5	1	6	1	1
	6	6	6	1	6	1	6
	6	6	1	6	3	1	3
	6	1	1	1	6	5	6
	5	5	5	5	5	5	3
	5	5	3	3	3	5	3
	5	5	5	3	3	5	5
	3	5	5	3	3	5	3
	5	5	3	5	5	5	5

5/Neural Net	2	2	2	2	2	2	7
	2	2	7	2	2	7	2
	7	7	2	2	2	7	2
	2	7	2	7	2	2	2
	2	7	2	7	2	2	2
	2	2	2	2	7	7	2
	2	7	7	7	7	2	2
	2	7	7	7	2	2	7
	2	2	2	2	2	7	2
	2	2	2	2	2	2	2
6/Cont. Neural Net	1	1	1	1	1	7	1
	1	7	7	1	1	1	1
	1	7	1	7	1	7	1
	1	1	1	7	7	7	1
	1	1	1	1	1	1	7
	7	1	1	1	1	1	7
	1	1	1	1	1	1	7
	1	7	7	7	7	1	1
	1	7	1	7	2	7	1
	1	7	7	1	1	1	1
7/Neural Tree	1	6	3	1	2	3	1
	1	2	2	3	1	1	2
	1	2	3	1	1	1	1
	3	3	1	3	1	6	1
	3	3	3	4	1	1	1
	1	3	1	1	3	1	2
	1	1	7	1	3	4	1
	1	3	1	1	2	1	1
	1	2	1	1	2	4	1
	1	2	3	1	2	1	1
8/Cont. Neural Net	1	7	7	7	7	1	7
	1	1	7	7	1	1	7
	7	7	1	7	7	7	7
	1	7	1	1	1	7	7
	7	7	7	7	1	1	1
	1	7	7	7	7	7	1
	7	7	7	7	7	7	7
	1	7	1	1	7	7	7
	7	7	7	7	7	7	7
	7	7	7	7	1	1	7

Processed Data

ID NUMBER	1	2	3	4	5	6	7	ACCURACY
1/Neural Net	0.00%	0.00%	20.00%	40.00%	40.00%	0.00%	0.00%	0.00%
	0.00%	0.00%	30.00%	10.00%	60.00%	0.00%	0.00%	0.00%
	0.00%	0.00%	30.00%	30.00%	40.00%	0.00%	0.00%	30.00%
	0.00%	0.00%	30.00%	10.00%	60.00%	0.00%	0.00%	10.00%
	0.00%	0.00%	20.00%	30.00%	50.00%	0.00%	0.00%	50.00%
	0.00%	0.00%	40.00%	0.00%	60.00%	0.00%	0.00%	0.00%
	0.00%	0.00%	20.00%	10.00%	70.00%	0.00%	0.00%	0.00%
Mean	0.00%	0.00%	27.14%	18.57%	54.29%	0.00%	0.00%	12.86%
St. Dev	0.00%	0.00%	7.56%	14.64%	11.34%	0.00%	0.00%	19.76%
2/Neural Tree	80.00%	20.00%	0.00%	0.00%	0.00%	0.00%	0.00%	80.00%
	80.00%	20.00%	0.00%	0.00%	0.00%	0.00%	0.00%	20.00%
	70.00%	10.00%	20.00%	0.00%	0.00%	0.00%	0.00%	20.00%
	50.00%	10.00%	30.00%	10.00%	0.00%	0.00%	0.00%	10.00%
	50.00%	0.00%	30.00%	10.00%	10.00%	0.00%	0.00%	10.00%
	40.00%	50.00%	0.00%	0.00%	10.00%	0.00%	0.00%	0.00%
	40.00%	50.00%	10.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Mean	58.57%	22.86%	12.86%	2.86%	2.86%	0.00%	0.00%	20.00%
St. Dev	17.73%	19.76%	13.80%	4.88%	4.88%	0.00%	0.00%	27.69%
3/Neural Tree	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
	40.00%	29.00%	40.00%	0.00%	0.00%	0.00%	0.00%	40.00%
	10.00%	10.00%	60.00%	0.00%	0.00%	0.00%	20.00%	0.00%
	20.00%	30.00%	20.00%	0.00%	0.00%	10.00%	20.00%	0.00%
	30.00%	40.00%	20.00%	0.00%	0.00%	0.00%	10.00%	0.00%
	50.00%	20.00%	30.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Mean	35.71%	32.71%	24.29%	0.00%	0.00%	1.43%	7.14%	34.29%
St. Dev	33.09%	32.53%	21.49%	0.00%	0.00%	3.78%	9.51%	47.21%
4/Neural Net	20.00%	0.00%	10.00%	0.00%	40.00%	20.00%	0.00%	20.00%
	30.00%	0.00%	0.00%	0.00%	50.00%	20.00%	0.00%	0.00%
	30.00%	0.00%	20.00%	0.00%	40.00%	10.00%	0.00%	20.00%
	40.00%	0.00%	30.00%	0.00%	20.00%	10.00%	0.00%	0.00%
	0.00%	0.00%	40.00%	0.00%	20.00%	40.00%	0.00%	20.00%
	40.00%	0.00%	10.00%	0.00%	50.00%	10.00%	0.00%	10.00%
	20.00%	0.00%	40.00%	0.00%	20.00%	20.00%	0.00%	0.00%
Mean	25.71%	0.00%	21.43%	0.00%	34.29%	18.57%	0.00%	10.00%
St. Dev	13.97%	0.00%	15.74%	0.00%	13.97%	10.69%	0.00%	10.00%
5/Neural Net	0.00%	90.00%	0.00%	0.00%	0.00%	0.00%	10.00%	0.00%
	0.00%	50.00%	0.00%	0.00%	0.00%	0.00%	50.00%	50.00%
	0.00%	70.00%	0.00%	0.00%	0.00%	0.00%	30.00%	0.00%
	0.00%	60.00%	0.00%	0.00%	0.00%	0.00%	40.00%	0.00%
	0.00%	80.00%	0.00%	0.00%	0.00%	0.00%	20.00%	0.00%
	0.00%	60.00%	0.00%	0.00%	0.00%	0.00%	40.00%	0.00%
	0.00%	80.00%	0.00%	0.00%	0.00%	0.00%	20.00%	20.00%
Mean	0.00%	70.00%	0.00%	0.00%	0.00%	0.00%	30.00%	10.00%
St. Dev	0.00%	14.14%	0.00%	0.00%	0.00%	0.00%	14.14%	19.15%
6/Cont. Neural Net	10.00%	0.00%	0.00%	0.00%	0.00%	0.00%	90.00%	10.00%
	20.00%	0.00%	0.00%	0.00%	0.00%	0.00%	80.00%	0.00%
	2.00%	0.00%	0.00%	0.00%	0.00%	0.00%	80.00%	0.00%
	10.00%	0.00%	0.00%	0.00%	0.00%	0.00%	80.00%	0.00%
	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%
	10.00%	0.00%	0.00%	0.00%	0.00%	0.00%	90.00%	0.00%
	20.00%	0.00%	0.00%	0.00%	0.00%	0.00%	80.00%	80.00%
Mean	10.29%	0.00%	0.00%	0.00%	0.00%	0.00%	85.71%	12.86%
St. Dev	7.78%	0.00%	0.00%	0.00%	0.00%	0.00%	7.87%	29.84%
7/Neural Tree	80.00%	0.00%	20.00%	0.00%	0.00%	0.00%	0.00%	80.00%
	10.00%	40.00%	40.00%	0.00%	0.00%	10.00%	0.00%	40.00%
	40.00%	10.00%	40.00%	0.00%	0.00%	0.00%	10.00%	40.00%
	70.00%	0.00%	20.00%	10.00%	0.00%	0.00%	0.00%	10.00%
	40.00%	40.00%	20.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	60.00%	0.00%	10.00%	20.00%	0.00%	10.00%	0.00%	10.00%
	80.00%	20.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Mean	54.29%	15.71%	21.43%	4.29%	0.00%	2.86%	1.43%	25.71%
St. Dev	25.73%	18.13%	14.64%	7.87%	0.00%	4.88%	3.78%	29.36%
8/Cont. Neural Net	50.00%	0.00%	0.00%	0.00%	0.00%	0.00%	50.00%	50.00%
	10.00%	0.00%	0.00%	0.00%	0.00%	0.00%	90.00%	0.00%
	30.00%	0.00%	0.00%	0.00%	0.00%	0.00%	70.00%	0.00%
	20.00%	0.00%	0.00%	0.00%	0.00%	0.00%	80.00%	0.00%
	40.00%	0.00%	0.00%	0.00%	0.00%	0.00%	60.00%	0.00%
	40.00%	0.00%	0.00%	0.00%	0.00%	0.00%	60.00%	0.00%
	20.00%	0.00%	0.00%	0.00%	0.00%	0.00%	80.00%	80.00%
Mean	30.00%	0.00%	0.00%	0.00%	0.00%	0.00%	70.00%	18.57%
St. Dev	14.14%	0.00%	0.00%	0.00%	0.00%	0.00%	14.14%	32.88%

Survey Responses

Test Subject 1 (Neural Net):

1. I don't think I did very well, as the training didn't seem to get the hand to do what it was supposed to during testing. Oh, and I am not the best at doing the right gesture at the right time.
2. Emotionally/Mentally - fine. The cap was a bit uncomfy but tolerable-ish. When I needed a break I got one, the testers were very accomodating.
3. I think it'd be good if maybe there was a "cheat sheet" just showing each gesture + the name of the gesture. Other than that, I thought it was cool!
4. I think I sat there and tried? Not really thinking I'm better than the other participants. It's not really that kind of test.
5. I am bad at going to the right position sometimes, and messed up some. I figure everyone probably messes up sometimes too! I had fun!

Test Subject 2 (Neural Tree):

1. I thought I put in a fair effort, however performance was mediocre.
2. I was slightly uncomfortable from the brainwave probes of the apparatus digging into my skull.
3. Use a more ergonomic design for the cap; include gel cushioning.
4. I think I was pretty good at generating a fist hand position. However, getting the different levels of concentration (rather than just 0/1 input) was difficult.
5. Perhaps other participants were able to generate higher or more distinguished levels of concentration than I was able to.

Test Subject 3 (Neural Tree):

1. I do not think I did well. I had difficulty moving the hand to positions that required intermediate levels of concentration.
2. I was comfortable.
3. I would have liked a clearer indication of which positions required more/less concentration. I tried to move the hand by watching the 'current goal' value and concentrating more or less to shift the value. Because that was the way I tried to move the hand, I would have liked an indicator of how much time I had left and more frequent updates of the current goal.
4. I could easily move the hand to the extended hand and fist positions.
5. I had difficulty moving the hand to positions that required intermediate levels of concentration.

Test Subject 4 (Neural Net):

1. Hard to tell; was not clear that it was a classification algorithm until the end
2. Pretty uncomfortable, but c'est la vie
3. If the inputs are dynamic but the outputs are classification, invest time into making more distinct actions
4. Kept hand in same position to reduce confounding variables
5. "Open hand" had fingers spread, which I think later confused the model

Test Subject 5 (Neural Net):

1. Not well. At least it was working
2. Physically: Head itched on top. Otherwise, just fine
3. Method seemed fine, better than what I would do.
4. Not thinking during the calibration
5. Others probably had more/different movements intentionally.

Test Subject 6 (Continuous Neural Net)

1. I believe I performed terribly. The hand seemed to favor only index grip, and any deviation was more random than derived. A random number generator could move that hand better than I did
2. Aside from the headset, the constant hand movement became uncomfortable the longer the program went on
3. It would be nice if it actually learned where my hand is moving. Whether that is training it more or less, I don't know.
4. I moved to index grip better than most, but not by choice.
5. I'm pretty sure other people could control the limb.

Test Subject 7 (Neural Tree)

1. I thought I did alright, considering that controlling how much you concentrate is difficult
2. I was very comfortable doing the experiment
3. The experiment could be improved by displaying the ranges for each type of configuration, instead of just giving percentages
4. I heard some people only got one or two hand positions, but I was able to get 3 on a regular basis
5. Near the end of the experiment, the consistency and accuracy of results was reduced, and I rarely got the hand configuration I wanted

Test Subject 8 (Continuous Neural Net) neglected to give a response

Appendix C: Code
All code available at https://github.com/pjpolley/Gallati_Polley_MQP_Code

CertaintyPrompt.cs
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Sender
{
    public partial class CertaintyPrompt : Form
    {
        public bool Continue;
        public CertaintyPrompt()
        {
            InitializeComponent();
        }

        private void ContinueButton_Click(object sender, EventArgs e)
        {
            Continue = true;
            this.DialogResult = DialogResult.OK;
            this.Close();
        }

        private void CancelButton_Click(object sender, EventArgs e)
        {
            Continue = false;
            this.DialogResult = DialogResult.OK;
            this.Close();
        }
    }
}
```

ContinuousNeuralNetForm.cs
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Sender
{
    public partial class ContinuousNeuralNetForm : Form
```

```
{

    private NeuralNet net;
    private SerialReader serial;
    private int currentHandPosition;
    private List<double[]> inputTrainingData;
    private List<double[]> outputTrainingData;
    private object dataLock = new object();

    private Dictionary<int, SetPoint> setPointList;
    private SetPoint lastSetPoint;
    private int expirationTimer = 10;

    private List<int> positionsToVisit = new List<int>();
    private string ANNfilename = Globals.NeuralNetSaveLocation;
    private string KFoldFilename = Globals.KFoldDataSaveLocation;


    public ContinuousNeuralNetForm()
    {
        InitializeComponent();

        net = new NeuralNet(8, 7, ANNfilename, KFoldFilename);
        inputTrainingData = new List<double[]>();
        outputTrainingData = new List<double[]>();

        if (inputTrainingData.Count > expirationTimer)
        {
            inputTrainingData.RemoveAt(0);
            outputTrainingData.RemoveAt(0);
        }

        serial = new SerialReader();
        serial.Read();

        UnityCommunicationHub.InitializeUnityCommunication();
        UnityCommunicationHub.TwoWayTransmission();

        Random rand = new Random(23);

        setPointList = Globals.GetBasicPositions();

        var firstPoint = setPointList[3];
        Globals.A1DesiredPosition = firstPoint.A1Position;
        Globals.A2DesiredPosition = firstPoint.A2Position;
        Globals.A3DesiredPosition = firstPoint.A3Position;
        Globals.B1DesiredPosition = firstPoint.B1Position;
        Globals.B2DesiredPosition = firstPoint.B2Position;
        Globals.B3DesiredPosition = firstPoint.B3Position;
        Globals.C1DesiredPosition = firstPoint.C1Position;
        Globals.C2DesiredPosition = firstPoint.C2Position;
        Globals.C3DesiredPosition = firstPoint.C3Position;
        Globals.D1DesiredPosition = firstPoint.D1Position;
        Globals.D2DesiredPosition = firstPoint.D2Position;
        Globals.D3DesiredPosition = firstPoint.D3Position;
        Globals.T1DesiredPosition = firstPoint.T1Position;
        Globals.T2DesiredPosition = firstPoint.T2Position;
```

```
            lastSetPoint = firstPoint;
            UnityCommunicationHub.WriteData(true);
        }

        private void Test()
        {
            lock (dataLock)
            {

                var percievedPosition = new 
SetPoint(0,0,0,0,0,0,0,0,0,0,0,0,0,0);
                var rand = new Random();
                //UnityCommunicationHub.WriteData(true);

                var percievedPositionArray = new double[7];

                if (inputTrainingData.Count != 0)
                {
                    percievedPositionArray = 
net.Think(inputTrainingData[inputTrainingData.Count - 1]);

                    double bestVal = 0;
                    SetPoint bestSetPoint = new SetPoint();


                    for (int i = 0; i < percievedPositionArray.Length; i++)
                    {
                        if (percievedPositionArray[i] > bestVal)
                        {
                            bestVal = percievedPositionArray[i];
                            bestSetPoint = setPointList[i];
                        }
                    }

                    if (lastSetPoint.Equals(bestSetPoint))
                    {
                        percievedPosition = setPointList[rand.Next(0, 7)];
                    }
                    else
                    {
                        percievedPosition = bestSetPoint;
                    }
                }
                else
                {

                    percievedPosition = setPointList[rand.Next(0, 7)];
                }

                if (positionsToVisit.Count > 0)
                {
                    percievedPosition = 
setPointList[positionsToVisit.First()];
```

```
                    for (int i = 0; i < 7; i++)
                    {
                        if ((7-i) == positionsToVisit.Count)
percievedPositionArray[i] = 1;
                    }
                    positionsToVisit.RemoveAt(0);
                }

                Globals.T1DesiredPosition = percievedPosition.T1Position;
                Globals.T2DesiredPosition = percievedPosition.T2Position;
                Globals.A1DesiredPosition = percievedPosition.A1Position;
                Globals.A2DesiredPosition = percievedPosition.A2Position;
                Globals.A3DesiredPosition = percievedPosition.A3Position;
                Globals.B1DesiredPosition = percievedPosition.B1Position;
                Globals.B2DesiredPosition = percievedPosition.B2Position;
                Globals.B3DesiredPosition = percievedPosition.B3Position;
                Globals.C1DesiredPosition = percievedPosition.C1Position;
                Globals.C2DesiredPosition = percievedPosition.C2Position;
                Globals.C3DesiredPosition = percievedPosition.C3Position;
                Globals.D1DesiredPosition = percievedPosition.D1Position;
                Globals.D2DesiredPosition = percievedPosition.D2Position;
                Globals.D3DesiredPosition = percievedPosition.D3Position;

                UnityCommunicationHub.WriteData(true);
                lastSetPoint = percievedPosition;
                Thread.Sleep(200);
                for (int i = 0; i < 250; i++)
                {
                    var input = serial.GetData();
                    double[] inputData = new double[8];
                    for (int j = 0; j < 8; j++)
                    {
                        inputData[j] = input[j];
                    }
                    UnityCommunicationHub.ReadData();


                    inputTrainingData.Add(inputData);
                    outputTrainingData.Add(percievedPositionArray);
                    Thread.Sleep(1);
                }
            }
        }

        private void Train()
        {
            lock (dataLock)
            {
                var networkTrainingInput = new
double[inputTrainingData.Count][];
                var networkTrainingOutput = new
double[outputTrainingData.Count][];
                for (int i = 0; i < inputTrainingData.Count; i++)
                {
                    networkTrainingInput[i] = inputTrainingData[i];
                    networkTrainingOutput[i] = outputTrainingData[i];
                }
                net.Train(networkTrainingInput, networkTrainingOutput, 10,
.1f);
            }
        }

        private void SaveButton_Click(object sender, EventArgs e)
        {
            net.Save();
        }

        private void TestButton_Click(object sender, EventArgs e)
        {
            UnityCommunicationHub.WriteData(true);
            for (int i = 0; i < 7; i++)
            {
                positionsToVisit.Add(i);
            }
            for (int i = 0; i < 22; i++)
            {
                Thread testThread = new Thread(Test);
                testThread.Start();
                Thread trainingThread = new Thread(Train);
                trainingThread.Start();
                Thread.Sleep(1000);
            }
        }

        private void ReconfigureButton_Click(object sender, EventArgs e)
        {
            var inDataArray = new double[inputTrainingData.Count][];
            var outDataArray = new double[outputTrainingData.Count][];

            for (int i = 0; i < inputTrainingData.Count; i++)
            {
                inDataArray[i] = inputTrainingData[i];
                outDataArray[i] = (outputTrainingData[i]);
            }

            net.dataset_in = inDataArray;
            net.dataset_out = outDataArray;

            CertaintyPrompt prompt = new CertaintyPrompt();

            if (prompt.ShowDialog() == DialogResult.OK && prompt.Continue)
            {
                net.Validate(8, 6);
            }
        }

        private double[] ScaleOutputStorageData(double[] inputData)
        {
            var returnData = new double[inputData.Length];
            for (int i = 0; i < inputData.Length; i++)
            {
                returnData[i] = inputData[i] / 90;
```

```
    }

    return returnData;
}

private float[] ScaleOutputData(double[] inputData)
{
    var returnData = new float[inputData.Length];
    for (int i = 0; i < inputData.Length; i++)
    {
        returnData[i] = (float)inputData[i] * (90);
    }

    return returnData;
}

private void Run()
{
    lock (dataLock)
    {


        var input = serial.GetData();
        double[] inputData = new double[8];
        for (int j = 0; j < 8; j++)
        {
            inputData[j] = input[j];
        }
        var percievedPositionArray = net.Think(inputData);

        double bestVal = 0;
        SetPoint bestSetPoint = new SetPoint();


        for (int i = 0; i < percievedPositionArray.Length; i++)
        {
            if (percievedPositionArray[i] > bestVal)
            {
                bestVal = percievedPositionArray[i];
                bestSetPoint = setPointList[i];
            }
        }


        var percievedPosition = bestSetPoint;

        Globals.T1DesiredPosition = percievedPosition.T1Position;
        Globals.T2DesiredPosition = percievedPosition.T2Position;
        Globals.A1DesiredPosition = percievedPosition.A1Position;
        Globals.A2DesiredPosition = percievedPosition.A2Position;
        Globals.A3DesiredPosition = percievedPosition.A3Position;
        Globals.B1DesiredPosition = percievedPosition.B1Position;
        Globals.B2DesiredPosition = percievedPosition.B2Position;
        Globals.B3DesiredPosition = percievedPosition.B3Position;
        Globals.C1DesiredPosition = percievedPosition.C1Position;
        Globals.C2DesiredPosition = percievedPosition.C2Position;
        Globals.C3DesiredPosition = percievedPosition.C3Position;
```

```
                Globals.D1DesiredPosition = percievedPosition.D1Position;
                Globals.D2DesiredPosition = percievedPosition.D2Position;
                Globals.D3DesiredPosition = percievedPosition.D3Position;

                UnityCommunicationHub.WriteData(true);
            }
        }

        private void Reader_Click(object sender, EventArgs e)
        {
            Thread.Sleep(200);
            Thread testThread = new Thread(Run);
            testThread.Start();
        }
    }
}
```

KFoldData.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Sender
{
    public class KFoldData
    {
        public int Breadth { get; }
        public int Depth { get; }
        public double TrainingSpeed { get;  }
        public int Iterations { get; }
        public double K { get; }



        public KFoldData(int breadth, int depth, double trainingSpeed, int 
iterations, double k)
        {
            this.Breadth = breadth;
            this.Depth = depth;
            this.TrainingSpeed = trainingSpeed;
            this.Iterations = iterations;
            this.K = k;
        }
    }
}
```

NeuralNet.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Media.Imaging;
using Accord.Math;
using Accord.Neuro;
using Accord.Neuro.Learning;


namespace Sender
{
    class NeuralNet
    {
        public double[][] dataset_in;
        public double[][] dataset_out;


        private ActivationNetwork network;
        private KFoldData topResults;

        private string ANNfilename;
        private string KFoldFilename;


        //private CrossValidation<ActivationNetwork, double, double>
validator;

        public NeuralNet(int inputs, int outputs, string ANNfilename, string
KFoldFilename)
        {
            NodeSavingReading reader = new NodeSavingReading();
            this.ANNfilename = ANNfilename;
            this.KFoldFilename = KFoldFilename;

            try
            {
                network = (ActivationNetwork)Network.Load(ANNfilename);
                topResults = reader.GetKFoldDataFromFile(KFoldFilename);
            }
            catch (FileNotFoundException e)
            {
                Console.WriteLine("Could not find file, generating new one");
                network = new ActivationNetwork(new SigmoidFunction(),
inputs, new int[4] {10, 10, 10,  outputs});
            }



        }

        public double Train(double[][] input, double[][] outputs)
        {
            var teacher = new ResilientBackpropagationLearning(network);
            teacher.LearningRate = topResults.TrainingSpeed;

            double error = 0;
            for (int iteration = 0; iteration < topResults.Iterations;
iteration++)
            {
                error = teacher.RunEpoch(input, outputs);
            }
            return error;
        }

        public double Train(double[][] input, double[][] outputs, int 
iterations, float rate)
        {
            var teacher = new ResilientBackpropagationLearning(network);
            teacher.LearningRate = rate;
            var inVal = input;
            var outVal = outputs;

            double error = 0;
            for (int iteration = 0; iteration < iterations; iteration++)
            {
                error = teacher.RunEpoch(inVal, outVal);
            }

            return error;
        }

        public double[] Think(double[] input)
        {
            return (network.Compute(input));
        }

        public void Save()
        {
            NodeSavingReading reader = new NodeSavingReading();
            network.Save(ANNfilename);
            reader.pushDataToFile(KFoldFilename,topResults);
        }



        public async void Validate(int inputSize, int outputSize)
        {
            List<KFoldData> inputsList = new List<KFoldData>();
            for (double trainingweights = 0.01; trainingweights <= 1.6; 
trainingweights += 0.1)
            {
                for (int breadth = 10; breadth <= 1000; breadth += 50)
                {
                    for (int depth = 1; depth < 5; depth++)
                    {
                        inputsList.Add(new KFoldData(breadth, depth, 
trainingweights, 0, 0));
                    }
                }
            }

            var kFoldList = await Task.WhenAll(inputsList.Select(i => 
kfold(inputSize, outputSize, i.Breadth, i.Depth, i.TrainingSpeed)));

            KFoldData returnedKFoldData = new KFoldData(0, 0, 0, 0, 
double.MaxValue);

            KFoldData[] KFoldArray = kFoldList;
            foreach (var tempKFoldData in KFoldArray)
            {
                if (tempKFoldData.K < returnedKFoldData.K)
                {
                    returnedKFoldData = tempKFoldData;
                }
            }

            Console.WriteLine("Best value is:");
            Console.WriteLine("Depth: " + returnedKFoldData.Depth);
            Console.WriteLine("Breadth: " + returnedKFoldData.Breadth);
            Console.WriteLine("Training Speed: " +
returnedKFoldData.TrainingSpeed);
            Console.WriteLine("Avg K Value: " + returnedKFoldData.K);

            int[] returnArray = new int[returnedKFoldData.Depth];
            for (int fillVal = 0; fillVal < returnedKFoldData.Depth;
fillVal++)
            {
                if (fillVal == returnedKFoldData.Depth - 1)
                {
                    returnArray[fillVal] = outputSize;
                }
                else
                {
                    returnArray[fillVal] = returnedKFoldData.Breadth;
                }
            }

            topResults = returnedKFoldData;
            network = new ActivationNetwork(new SigmoidFunction(), inputSize,
returnArray);
            network.Randomize();
            Save();
            Console.WriteLine("Done!");
        }


        async Task<KFoldData> kfold(int inputSize, int outputSize, int
breadth, int depth, double trainingweights)
        {
            await Task.Delay(1).ConfigureAwait(false);
            double bestKVal = double.MaxValue;
            KFoldData bestVal = new KFoldData(0, 0, 0, 0, 0);
            for (int iterations = 10; iterations < 10000; iterations =
iterations * 10)
            {

                int[] nodeArray = new int[depth + 1];
                for (int fillVal = 0; fillVal < depth; fillVal++)
                {
                    if (fillVal == 0) // depth - 1)
                    {
                        nodeArray[0] = outputSize;
                    }
                    else
                    {
                        nodeArray[fillVal] = breadth;
                    }
                }



                double kSumAvg = 0;
                for (int i = 0; i < 5; i++)
                {
                    var testNet = new ActivationNetwork(new
SigmoidFunction(), inputSize, nodeArray);
                    var testLearner = new
ResilientBackpropagationLearning(testNet);
                    testLearner.LearningRate = trainingweights;

                    int length = dataset_in.GetLength(0) / 5;

                    var trainingArrayIn = new double[dataset_in.GetLength(0)
* 4 / 5][];
                    var trainingArrayOut = new
double[dataset_out.GetLength(0) * 4 / 5][];
                    var testingArrayIn = new double[dataset_in.GetLength(0) /
5][];
                    var testingArrayOut = new double[dataset_out.GetLength(0)
/ 5][];

                    dataset_in.Take(i *
length).ToArray().CopyTo(trainingArrayIn, 0);
                    dataset_in.Skip((i * length) + length).Take((length * 5)
- (i * length + length)).ToArray().CopyTo(trainingArrayIn, i * length);

                    testingArrayIn = dataset_in.Skip(i *
length).Take(length).ToArray();

                    dataset_out.Take(i *
length).ToArray().CopyTo(trainingArrayOut, 0);
                    dataset_out.Skip((i * length) + length).Take((length * 5)
- (i * length + length)).ToArray().CopyTo(trainingArrayOut, i * length);

                    testingArrayOut = dataset_out.Skip(i *
length).Take(length).ToArray();


                    for (int iteration = 0; iteration < iterations;
iteration++)
                    {
                        testLearner.RunEpoch(trainingArrayIn,
trainingArrayOut);
                    }



                    double kSum = 0;
                    for (int k = 0; k < testingArrayIn.GetLength(0); k++)
                    {
                        var testResults = testNet.Compute(testingArrayIn[k]);
                        for (int j = 0; j < testResults.Length; j++)
                        {
                            kSum += Math.Abs(testResults[j] - 
testingArrayOut[k][j]);
                        }

                        kSumAvg += kSum;
                    }
                }

                kSumAvg = kSumAvg / dataset_in.GetLength(0);
                if (kSumAvg == 0)
                {
                    bestKVal = kSumAvg;
                    bestVal = new KFoldData(breadth, depth, trainingweights, 
iterations, bestKVal);
                    Console.WriteLine("Thread Complete " + breadth+ " " + 
depth + " " + trainingweights + " " + iterations + " " + bestKVal);
                    return bestVal;


                }

                if (kSumAvg < bestKVal)
                {
                    bestKVal = kSumAvg;
                    bestVal = new KFoldData(breadth, depth, trainingweights, 
iterations, bestKVal);

                }
            }
            Console.WriteLine("Thread Complete " + breadth + " " + depth + " 
" + trainingweights + " " + bestVal.Iterations + " " + bestKVal);

            return bestVal;
            //return new Task<KFoldData>(() => helperFunction(inputSize, 
outputSize, breadth,depth,trainingweights));

        }

        //KFoldData helperFunction(int inputSize, int outputSize, int 
breadth, int depth, double trainingweights)
        //{

        //}
    }
}
```

NeuralNetForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Controls;
using System.Windows.Forms;
using System.Windows.Media.Converters;
using Accord.Neuro.Learning;

namespace Sender
{
    public partial class NeuralNetForm : Form
    {
        private NeuralNet net;
        private SerialReader serial;
        private int currentHandPosition;
        private List<double[]> inputTrainingData;
        private List<double[]> outputTrainingData;
        private object dataLock = new object();
        private Dictionary<string, int> indexList;
        private Dictionary<int, SetPoint> setPointList;

        private string ANNfilename = Globals.NeuralNetSaveLocation;
        private string KFoldFilename = Globals.KFoldDataSaveLocation;

        public NeuralNetForm()
        {
            InitializeComponent();

            NodeSavingReading reader = new NodeSavingReading();

            net = new NeuralNet(8, 7, ANNfilename, KFoldFilename);
            inputTrainingData =
reader.GetStoredDataFromFile(Globals.inputDataStorage);
            outputTrainingData =
reader.GetStoredDataFromFile(Globals.outputDataStorage);

            //inputTrainingData = new List<double[]>();
            //outputTrainingData = new List<double[]>();



            serial = new SerialReader();
            serial.Read();

            UnityCommunicationHub.InitializeUnityCommunication();
            UnityCommunicationHub.TwoWayTransmission();


            indexList = Globals.GetBasicValues();
            setPointList = Globals.GetBasicPositions();

            foreach (KeyValuePair<string, int> position in indexList)
            {
                DefaultPositionsBox.Items.Add(position.Key);
            }



        }
```

```
private void NeuralNetForm_Load(object sender, EventArgs e)
{

}

private void Run()
{
    //while (true)
    {
        lock (dataLock)
        {

            var input = serial.GetData();
            double[] inputData = new double[8];
            for (int j = 0; j < 8; j++)
            {
                inputData[j] = input[j];
            }
            var percievedPositionArray = net.Think(inputData);

            double bestVal = 0;
            SetPoint bestSetPoint = new SetPoint();


            for (int i = 0; i < percievedPositionArray.Length; i++)
            {
                if (percievedPositionArray[i] > bestVal)
                {
                    bestVal = percievedPositionArray[i];
                    bestSetPoint = setPointList[i];
                }
            }


            var percievedPosition = bestSetPoint;

            Globals.T1DesiredPosition = percievedPosition.T1Position;
            Globals.T2DesiredPosition = percievedPosition.T2Position;
            Globals.A1DesiredPosition = percievedPosition.A1Position;
            Globals.A2DesiredPosition = percievedPosition.A2Position;
            Globals.A3DesiredPosition = percievedPosition.A3Position;
            Globals.B1DesiredPosition = percievedPosition.B1Position;
            Globals.B2DesiredPosition = percievedPosition.B2Position;
            Globals.B3DesiredPosition = percievedPosition.B3Position;
            Globals.C1DesiredPosition = percievedPosition.C1Position;
            Globals.C2DesiredPosition = percievedPosition.C2Position;
            Globals.C3DesiredPosition = percievedPosition.C3Position;
            Globals.D1DesiredPosition = percievedPosition.D1Position;
            Globals.D2DesiredPosition = percievedPosition.D2Position;
            Globals.D3DesiredPosition = percievedPosition.D3Position;

            UnityCommunicationHub.WriteData(true);
        }
    }
}
```

```
        private void Train()
        {
            lock (dataLock)
            {
                var networkTrainingInput = new
double[inputTrainingData.Count][];
                var networkTrainingOutput = new
double[outputTrainingData.Count][];
                for (int i = 0; i < inputTrainingData.Count; i++)
                {
                    networkTrainingInput[i] = inputTrainingData[i];
                    networkTrainingOutput[i] = outputTrainingData[i];
                }


                net.Train(networkTrainingInput, networkTrainingOutput, 100,
0.1f);
            }
        }

        private void DefaultPositionsBox_SelectedIndexChanged(object sender,
EventArgs e)
        {

            var inputItemName = (System.Windows.Forms.ListBox) sender;
            currentHandPosition =
indexList[(string)inputItemName.SelectedItem];
        }



        private void SaveButton_Click(object sender, EventArgs e)
        {
            NodeSavingReading reader = new NodeSavingReading();
            net.Save();
            reader.pushDataToFile(Globals.inputDataStorage,
inputTrainingData);
            reader.pushDataToFile(Globals.outputDataStorage,
outputTrainingData);
        }

        private void logButton_Click(object sender, EventArgs e)
        {
            Thread.Sleep(200);
            for (int i = 0; i < 50; i++)
            {
                double[] inData = serial.GetData();
                double[] inputData = new double[8];
                for (int j = 0; j < 8; j++)
                {
                    inputData[j] = inData[j];
                }
                inputTrainingData.Add(inputData);
                double[] outputData = new double[7];
                outputData[currentHandPosition] = 1;
                outputTrainingData.Add(outputData);
                Thread.Sleep(1);
```

```
    }
}

private void TrainButton_Click(object sender, EventArgs e)
{
    Thread trainingThread = new Thread(Train);
    trainingThread.Start();
}

private void TestButton_Click(object sender, EventArgs e)
{
    Thread.Sleep(200);
    Thread testThread = new Thread(Run);
    testThread.Start();
}

private void ReconfigureButton_Click(object sender, EventArgs e)
{
    var inDataArray = new double[inputTrainingData.Count][];
    var outDataArray = new double[outputTrainingData.Count][];

    for (int i = 0; i < inputTrainingData.Count; i++)
    {
        inDataArray[i] = inputTrainingData[i];
        outDataArray[i] = (outputTrainingData[i]);
    }

    net.dataset_in = inDataArray;
    net.dataset_out = outDataArray;

    CertaintyPrompt prompt = new CertaintyPrompt();

    if (prompt.ShowDialog() == DialogResult.OK && prompt.Continue)
    {
        net.Validate(8, 7);
    }
}

private double[] ScaleOutputStorageData(double[] inputData)
{
    var returnData = new double[inputData.Length];
    for (int i = 0; i < inputData.Length; i++)
    {
        returnData[i] = inputData[i] /90;
    }

    return returnData;
}

private float[] ScaleOutputData(double[] inputData)
{
    var returnData = new float[inputData.Length];
    for (int i = 0; i < inputData.Length; i++)
    {
        returnData[i] = (float)inputData[i] * (90);
    }
```

```
            return returnData;
        }
    }
}
```

NeuralTreeWindow.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;

namespace Sender
{
    public partial class NeuralTreeWindow : Form
    {
        TreeNode activeNode = null;

        PatsControlScheme controls;

        private int fingerSelected = Globals.THUMB;
        private int jointSelected = Globals.OUTERJOINT;

        volatile bool continueControlling = true;
        long numInputs = 0;
        TreeNode lastNode = null;
        Stopwatch timer = new Stopwatch();
        Node currentNode;
        SerialReader reader = new SerialReader();
        int rate;

        public NeuralTreeWindow()
        {
            this.Size = new System.Drawing.Size(1710, 1301);
            this.FormClosing += Globals.CloseAllForms;
            InitializeComponent();

            updateFingerDisplay();

            UnityCommunicationHub.InitializeUnityCommunication();

            //initialize tree structure
            controls = new PatsControlScheme();
            controls.Initialize();

            loadPositions(controls.root);

            List<Node> newList = controls.allNodes.Values.ToList<Node>();
            //order the list from lowest to highest to ensure all nodes are
populated in order
            newList.OrderBy(o => o.id);

            //make sure the list of nodes is clear
            NeuronTreeView.Nodes.Clear();
            //sequentially add all the nodes to the tree
            foreach (Node n in newList)
            {
                if (n.id != Globals.CONTROLNODE)
                {
                    //figure out what the parent id is
                    int parentID = n.parent;
                    //if it's the root node, just add it to the tree
                    if (parentID == Globals.NULLPARENT)
                    {
                        TreeNode newNode = new TreeNode(n.name);
                        newNode.Tag = n.id;
                        NeuronTreeView.Nodes.Add(newNode);
                        activeNode = newNode;
                    }
                    //in the case that it's a child node, figure out what
display node to add it to
                    else
                    {
                        //iterate through until you find the parent
                        TreeNode parentNode = findNodeInTree(parentID);
                        //once we find the parent, add it to the parent's
children
                        TreeNode newNode = new TreeNode(n.name);
                        newNode.Tag = n.id;
                        parentNode.Nodes.Add(newNode);
                    }
                }
            }

            foreach (Node n in newList)
            {
                if (n.id != Globals.CONTROLNODE &&
n.children.Contains(Globals.CONTROLNODE))
                {
                    TreeNode tn = findNodeInTree(n.id);
                    bool done = false;
                    foreach (TreeNode tnn in tn.Nodes)
                    {
                        if (!done && (int)tnn.Tag == Globals.CONTROLNODE)
                        {
                            tnn.Remove();
                            done = true;
                        }
                    }
                    TreeNode newControlNode = new TreeNode("Controls");
                    newControlNode.Tag = Globals.CONTROLNODE;
                    tn.Nodes.Add(newControlNode);
                }
            }
        }

        private void NeuronTreeView_NodeClicked(object sender,
TreeNodeMouseClickEventArgs e)
        {
            Console.WriteLine(e.Node.Text + " Clicked");
            Console.WriteLine("Tag is: " + e.Node.Tag);
            if ((int)e.Node.Tag != Globals.CONTROLNODE)
            {
                activeNode = e.Node;
                Node thisNode = controls.allNodes[(int)e.Node.Tag];
                loadPositions(thisNode);
                Console.WriteLine("Not a control node");
            }
            else
            {
                Console.WriteLine("A control node");
            }
        }

        private void loadPositions(Node thisNode)
        {
            Globals.A1DesiredPosition = thisNode.A1Position;
            Globals.A2DesiredPosition = thisNode.A2Position;
            Globals.A3DesiredPosition = thisNode.A3Position;
            Globals.B1DesiredPosition = thisNode.B1Position;
            Globals.B2DesiredPosition = thisNode.B2Position;
            Globals.B3DesiredPosition = thisNode.B3Position;
            Globals.C1DesiredPosition = thisNode.C1Position;
            Globals.C2DesiredPosition = thisNode.C2Position;
            Globals.C3DesiredPosition = thisNode.C3Position;
            Globals.D1DesiredPosition = thisNode.D1Position;
            Globals.D2DesiredPosition = thisNode.D2Position;
            Globals.D3DesiredPosition = thisNode.D3Position;
            Globals.T1DesiredPosition = thisNode.T1Position;
            Globals.T2DesiredPosition = thisNode.T2Position;
            UnityCommunicationHub.TwoWayTransmission();
        }

        private void setHandPositionButton_Click(object sender, EventArgs e)
        {
            UnityCommunicationHub.ReadData(true);
            SetPoint newPoint = new SetPoint
            {
                A1Position = Globals.A1DesiredPosition,
                A2Position = Globals.A2DesiredPosition,
                A3Position = Globals.A3DesiredPosition,
                B1Position = Globals.B1DesiredPosition,
                B2Position = Globals.B2DesiredPosition,
                B3Position = Globals.B3DesiredPosition,
                C1Position = Globals.C1DesiredPosition,
                C2Position = Globals.C2DesiredPosition,
                C3Position = Globals.C3DesiredPosition,
                D1Position = Globals.D1DesiredPosition,
                D2Position = Globals.D2DesiredPosition,
                D3Position = Globals.D3DesiredPosition,
                T1Position = Globals.T1DesiredPosition,
                T2Position = Globals.T2DesiredPosition
            };
            controls.allNodes[(int)activeNode.Tag].setHandPosition(newPoint);
}

        private void AddAnotherLayerButton_Click(object sender, EventArgs e)
        {
            Node thisNode = controls.allNodes[(int)activeNode.Tag];
            if (thisNode.children.Count != 0)
            {
                return;//node already has children. do nothing
            }
            for (int i = 0; i < controls.childrenPerNode; i++)
            {
                //calculate the id for the new node
                int newID = (thisNode.id * 10) + i + 1;
                //create a new node in the backend
                controls.createNewNode(newID, controls.childrenPerNode,
thisNode.id);
                //and then add it to the frontend
                TreeNode newDisplayNode = new
TreeNode(controls.allNodes[newID].name);
                newDisplayNode.Tag = controls.allNodes[newID].id;
                activeNode.Nodes.Add(newDisplayNode);
            }
            controls.allNodes[thisNode.id].children.Add(Globals.CONTROLNODE);
            TreeNode newControlNode = new TreeNode("Controls");
            newControlNode.Tag = Globals.CONTROLNODE;
            activeNode.Nodes.Add(newControlNode);
        }

        private void removeLayerButton_Click(object sender, EventArgs e)
        {
            Node thisNode = controls.allNodes[(int)activeNode.Tag];
            List<int> children = thisNode.children;
            foreach (int i in children)
            {
                if (i != Globals.CONTROLNODE)
                {
                    controls.allNodes.Remove(i);
                }
            }
            activeNode.Nodes.Clear();
            thisNode.children.Clear();
        }

        private void changeNameButton_Click(object sender, EventArgs e)
        {
            activeNode.Text = desiredNameBox.Text;
            controls.allNodes[(int)activeNode.Tag].name =
desiredNameBox.Text;
        }

        private void purgeChildren(TreeNode node)
        {
            if (node != null && (int)node.Tag != Globals.CONTROLNODE)
            {
                foreach (TreeNode n in node.Nodes)
                {
                    purgeChildren(n);
                }
                node.Remove();
                controls.allNodes.Remove((int)node.Tag);
            }
```

```
    else
    {
        //the controls option
        if (node != null)
        {
            node.Remove();
        }
    }
}

private TreeNode findNodeInTree(int id)
{
    if (id == Globals.CONTROLNODE)
    {
        return null;
    }
    TreeNode root = NeuronTreeView.Nodes[0];
    if ((int)root.Tag == id)
    {
        return root;
    }
    else
    {
        foreach (TreeNode n in root.Nodes)
        {
            TreeNode checkedNode = recursiveFindNode(n, id);
            if (checkedNode != null)
            {
                return checkedNode;
            }
        }
        return null;
    }
}

private TreeNode recursiveFindNode(TreeNode n, int id)
{
    if (id == Globals.CONTROLNODE)
    {
        return null;
    }
    if ((int)n.Tag == id)
    {
        return n;
    }
    else
    {
        TreeNode foundNode;
        foreach (TreeNode child in n.Nodes)
        {
            foundNode = recursiveFindNode(child, id);
            if (foundNode != null)
            {
                return foundNode;
            }
        }
        return null;
```

```
            }
        }

        private void handDelayButton_Click(object sender, EventArgs e)
        {
            controls.timeNeededForChange =
System.Convert.ToInt32(handDelayBox.Text);
        }

        private void saveCommandStructure_Click(object sender, EventArgs e)
        {
            controls.pushDataToFile();
        }

        private void hardResetButton_Click(object sender, EventArgs e)
        {
            //purge all nodes
            NeuronTreeView.Nodes.Clear();

            //and reset the tree
            controls.instantiateNewTree(1,
System.Convert.ToInt32(handDelayBox.Text));
            TreeNode newNode = new TreeNode(controls.root.name);
            newNode.Tag = controls.root.id;
            NeuronTreeView.Nodes.Add(newNode);
            NeuronTreeView.ExpandAll();
        }

        private void ThumbSelectButton_Click(object sender, EventArgs e)
        {
            fingerSelected = Globals.THUMB;
            InnerJointButton.Hide();
            if (jointSelected == Globals.INNERJOINT)
            {
                jointSelected = Globals.MIDDLEJOINT;
            }
            updateFingerDisplay();
        }

        private void IndexSelectButton_Click(object sender, EventArgs e)
        {
            fingerSelected = Globals.POINTER;
            updateFingerDisplay();
            InnerJointButton.Show();
        }

        private void MiddleSelectButton_Click(object sender, EventArgs e)
        {
            fingerSelected = Globals.MIDDLE;
            updateFingerDisplay();
            InnerJointButton.Show();
        }

        private void RingSelectButton_Click(object sender, EventArgs e)
        {
            fingerSelected = Globals.RING;
            updateFingerDisplay();
            InnerJointButton.Show();
        }

        private void PinkySelectButton_Click(object sender, EventArgs e)
        {
            fingerSelected = Globals.PINKY;
            InnerJointButton.Hide();
            if (jointSelected == Globals.INNERJOINT)
            {
                jointSelected = Globals.MIDDLEJOINT;
            }
            updateFingerDisplay();
        }

        private void OuterJointButton_Click(object sender, EventArgs e)
        {
            jointSelected = Globals.OUTERJOINT;
            updateFingerDisplay();
        }

        private void MiddleJointButton_Click(object sender, EventArgs e)
        {
            jointSelected = Globals.MIDDLEJOINT;
            updateFingerDisplay();
        }

        private void InnerJointButton_Click(object sender, EventArgs e)
        {
            jointSelected = Globals.INNERJOINT;
            updateFingerDisplay();
        }

        private void updateFingerDisplay()
        {
            currentlyModifyingBox.Text = "Modifying " +
Globals.valuesToStrings[fingerSelected] + " " +
Globals.valuesToStrings[jointSelected];

            if (fingerSelected == Globals.THUMB)
            {
                if (jointSelected == Globals.OUTERJOINT)
                {
                    DesiredAngleInput.Text =
Globals.T2DesiredPosition.ToString();
                }
                else if (jointSelected == Globals.MIDDLEJOINT)
                {
                    DesiredAngleInput.Text =
Globals.T1DesiredPosition.ToString();
                }
            }
            else if (fingerSelected == Globals.POINTER)
            {
                if (jointSelected == Globals.OUTERJOINT)
                {
                    DesiredAngleInput.Text =
Globals.A3DesiredPosition.ToString();
```

```
                }
                else if (jointSelected == Globals.MIDDLEJOINT)
                {
                    DesiredAngleInput.Text =
Globals.A2DesiredPosition.ToString();
                }
                else if (jointSelected == Globals.INNERJOINT)
                {
                    DesiredAngleInput.Text =
Globals.A1DesiredPosition.ToString();
                }
            }
            else if (fingerSelected == Globals.MIDDLE)
            {
                if (jointSelected == Globals.OUTERJOINT)
                {
                    DesiredAngleInput.Text =
Globals.B3DesiredPosition.ToString();
                }
                else if (jointSelected == Globals.MIDDLEJOINT)
                {
                    DesiredAngleInput.Text =
Globals.B2DesiredPosition.ToString();
                }
                else if (jointSelected == Globals.INNERJOINT)
                {
                    DesiredAngleInput.Text =
Globals.B1DesiredPosition.ToString();
                }
            }
            else if (fingerSelected == Globals.RING)
            {
                if (jointSelected == Globals.OUTERJOINT)
                {
                    DesiredAngleInput.Text =
Globals.C3DesiredPosition.ToString();
                }
                else if (jointSelected == Globals.MIDDLEJOINT)
                {
                    DesiredAngleInput.Text =
Globals.C2DesiredPosition.ToString();
                }
                else if (jointSelected == Globals.INNERJOINT)
                {
                    DesiredAngleInput.Text =
Globals.C1DesiredPosition.ToString();
                }
            }
            else if (fingerSelected == Globals.PINKY)
            {
                if (jointSelected == Globals.OUTERJOINT)
                {
                    DesiredAngleInput.Text =
Globals.D2DesiredPosition.ToString();
                }
                else if (jointSelected == Globals.MIDDLEJOINT)
                {
                    DesiredAngleInput.Text =
Globals.D1DesiredPosition.ToString();
                }
            }
        }

        private void setDesiredAngle()
        {
            float desiredAngle =
(float)System.Convert.ToDouble(DesiredAngleInput.Text);
            if (fingerSelected == Globals.THUMB)
            {
                if (jointSelected == Globals.OUTERJOINT)
                {
                    Globals.T2DesiredPosition = desiredAngle;
                }
                else if (jointSelected == Globals.MIDDLEJOINT)
                {
                    Globals.T1DesiredPosition = desiredAngle;
                }
            }
            else if (fingerSelected == Globals.POINTER)
            {
                if (jointSelected == Globals.OUTERJOINT)
                {
                    Globals.A3DesiredPosition = desiredAngle;
                }
                else if (jointSelected == Globals.MIDDLEJOINT)
                {
                    Globals.A2DesiredPosition = desiredAngle;
                }
                else if (jointSelected == Globals.INNERJOINT)
                {
                    Globals.A1DesiredPosition = desiredAngle;
                }
            }
            else if (fingerSelected == Globals.MIDDLE)
            {
                if (jointSelected == Globals.OUTERJOINT)
                {
                    Globals.B3DesiredPosition = desiredAngle;
                }
                else if (jointSelected == Globals.MIDDLEJOINT)
                {
                    Globals.B2DesiredPosition = desiredAngle;
                }
                else if (jointSelected == Globals.INNERJOINT)
                {
                    Globals.B1DesiredPosition = desiredAngle;
                }
            }
            else if (fingerSelected == Globals.RING)
            {
                if (jointSelected == Globals.OUTERJOINT)
                {
                    Globals.C3DesiredPosition = desiredAngle;
                }
                else if (jointSelected == Globals.MIDDLEJOINT)
                {
                    Globals.C2DesiredPosition = desiredAngle;
                }
                else if (jointSelected == Globals.INNERJOINT)
                {
                    Globals.C1DesiredPosition = desiredAngle;
                }
            }
            else if (fingerSelected == Globals.PINKY)
            {
                if (jointSelected == Globals.OUTERJOINT)
                {
                    Globals.D2DesiredPosition = desiredAngle;
                }
                else if (jointSelected == Globals.MIDDLEJOINT)
                {
                    Globals.D1DesiredPosition = desiredAngle;
                }
            }
        }

        private double configuredMidPoint;

        private void IncreaseAngleButton_Click(object sender, EventArgs e)
        {
            float desiredAngle =
(float)System.Convert.ToDouble(DesiredAngleInput.Text) + 1.0f;
            DesiredAngleInput.Text = desiredAngle.ToString();
            setDesiredAngle();
            UnityCommunicationHub.TwoWayTransmission();
        }

        private void DecreaseAngleButton_Click(object sender, EventArgs e)
        {
            float desiredAngle =
(float)System.Convert.ToDouble(DesiredAngleInput.Text) - 1.0f;
            DesiredAngleInput.Text = desiredAngle.ToString();
            setDesiredAngle();
            UnityCommunicationHub.TwoWayTransmission();
        }

        private void DesiredAngleInput_KeyDown(object sender, KeyEventArgs e)
        {
            if (e.KeyCode == Keys.Enter)
            {
                setDesiredAngle();
                UnityCommunicationHub.TwoWayTransmission();
                e.Handled = e.SuppressKeyPress = true;
            }
        }

        private void beginControllingHandButton_Click(object sender,
EventArgs e)
        {
            int desiredMillisecondDelay = controls.timeNeededForChange;
            int arraySize = (desiredMillisecondDelay / 1000) * rate;
            reader.Read();
            rate = reader.getRate();

            double lowConcentration = 0;
            double highConcentration = 0;

            //first get threshholds
            bool done = false;
            while (!done)
            {

                int reads = 0;
                decimal allReads = 0;
                MessageBox.Show("First try to let your mind wander until the
next popup appears. Hit OK when ready.", string.Empty, MessageBoxButtons.OK);
                timer.Start();
                while (timer.ElapsedMilliseconds <
Globals.threshholdAquisitionTime)
                {
                    decimal currentIn =
(decimal)Math.Abs(reader.GetData()[Globals.inputNode]);
                    allReads += currentIn;
                    reads++;
                    currentAverageBox.Text = (allReads / reads).ToString();
                    currentInputBox.Text = currentIn.ToString();
                }

                timer.Reset();
                lowConcentration = (double)(allReads / reads);
                midpointTextBox.Text = lowConcentration.ToString();
                Console.WriteLine("Low concentration was: " +
lowConcentration);

                reads = 0;
                allReads = 0;

                MessageBox.Show("Next try to focus as hard as possible
something. Hit OK when ready.", string.Empty, MessageBoxButtons.OK);

                timer.Start();
                while (timer.ElapsedMilliseconds <
Globals.threshholdAquisitionTime)
                {
                    decimal currentIn =
(decimal)Math.Abs(reader.GetData()[Globals.inputNode]);
                    allReads += currentIn;
                    reads++;
                    currentAverageBox.Text = (allReads / reads).ToString();
                    currentInputBox.Text = currentIn.ToString();
                }
                timer.Reset();

                highConcentration = (double)(allReads / reads);
                configuredMidPoint = ((highConcentration + lowConcentration)
/ 2);
                midpointTextBox.Text = configuredMidPoint.ToString();
                Console.WriteLine("High concentration was: " +
highConcentration);

                if (highConcentration > lowConcentration)
                {
                    done = true;
                }
            }

            MessageBox.Show("Ready to control hand. Press OK when ready.",
string.Empty, MessageBoxButtons.OK);
        }

        private void stopButton_Click(object sender, EventArgs e)
        {
            continueControlling = false;
        }

        private void iterateButton_Click(object sender, EventArgs e)
        {
            runTree();
        }

        private void runTree()
        {
            continueControlling = true;
            currentNode = controls.root;
            int desiredMillisecondDelay = controls.timeNeededForChange;

            TreeNode n = findNodeInTree(currentNode.id);
            n.BackColor = Color.Yellow;
            lastNode = n;

            timer = new Stopwatch();
            while (continueControlling)
            {
                //get the inputs and average them for the desired output
                double averageInput = 0;

                if (currentNode.children.Count < 1)
                {
                    continueControlling = false;
                    loadPositions(currentNode);
                    break;
                }
                Random rand = new Random();
                numInputs = 0;
                decimal accruedValues = 0;
                timer.Reset();
                timer.Start();
                decimal currentIn;
                while (timer.ElapsedMilliseconds < desiredMillisecondDelay)
                {
                    currentIn =
(decimal)Math.Abs(reader.GetData()[Globals.inputNode]);
                    accruedValues += currentIn;
                    numInputs++;
                    currentAverageBox.Text = (accruedValues / 
numInputs).ToString();
                    timeLeftBox.Text = (desiredMillisecondDelay - 
timer.ElapsedMilliseconds).ToString();
                    currentInputBox.Text = currentIn.ToString();
                }

                averageInput = (double)(accruedValues / numInputs);
                currentAverageBox.Text = averageInput.ToString();

                if (averageInput >= configuredMidPoint)
                {
                    //set current hand position as the one to move to
                    continueControlling = false;
                    timeLeftBox.Text = "Done";
                }
                else
                {
                    continueControlling = true;
                    currentNode = 
controls.allNodes[currentNode.children.First()];
                }

                n = findNodeInTree(currentNode.id);
                n.BackColor = Color.Yellow;

                lastNode.BackColor = Color.White;
                lastNode = n;

                Console.WriteLine(averageInput);

                numInputs = 0;
                accruedValues = 0;

                Console.WriteLine("Desired was " + currentNode.name);

                if (currentNode.id == Globals.CONTROLNODE)
                {
                    continueControlling = false;
                    loadPositions(currentNode);
                }
                else
                {
                    if (!continueControlling)
                    {
                        loadPositions(currentNode);
                    }
                }
            }
        }

        private void handDelayBox_KeyDown(object sender, KeyEventArgs e)
        {
            if (e.KeyCode == Keys.Enter)
            {
                controls.timeNeededForChange = 
System.Convert.ToInt32(handDelayBox);
```

```
                e.Handled = e.SuppressKeyPress = true;
            }
        }

        private void desiredNameBox_KeyDown(object sender, KeyEventArgs e)
        {
            if (e.KeyCode == Keys.Enter)
            {
                activeNode.Text = desiredNameBox.Text;
                controls.allNodes[(int)activeNode.Tag].name =
desiredNameBox.Text;
                e.Handled = e.SuppressKeyPress = true;
            }
        }
    }
}
```

Node.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Sender
{
    public class Node
    {
        public string name;

        public int id;
        public List<int> children = new List<int>();
        public int parent = Globals.NULLPARENT;//indicates the root node

        public float T1Position = 0.0f;
        public float T2Position = 0.0f;
        public float A1Position = 0.0f;
        public float A2Position = 0.0f;
        public float A3Position = 0.0f;
        public float B1Position = 0.0f;
        public float B2Position = 0.0f;
        public float B3Position = 0.0f;
        public float C1Position = 0.0f;
        public float C2Position = 0.0f;
        public float C3Position = 0.0f;
        public float D1Position = 0.0f;
        public float D2Position = 0.0f;
        public float D3Position = 0.0f;

        public Node(string name, SetPoint handPosition, int id, List<int>
children, int parent)
        {
            this.name = name;
            this.id = id;
            this.children = children;
            this.parent = parent;
            this.setHandPosition(handPosition);
        }

        public void setHandPosition(SetPoint input)
        {
            //needed for error catching
            if (input != null)
            {
                this.T1Position = input.T1Position;
                this.T2Position = input.T2Position;
                this.A1Position = input.A1Position;
                this.A2Position = input.A2Position;
                this.A3Position = input.A3Position;
                this.B1Position = input.B1Position;
                this.B2Position = input.B2Position;
                this.B3Position = input.B3Position;
                this.C1Position = input.C1Position;
                this.C2Position = input.C2Position;
                this.C3Position = input.C3Position;
                this.D1Position = input.D1Position;
                this.D2Position = input.D2Position;
                this.D3Position = input.D3Position;
            }
        }

        public SetPoint getHandPosition()
        {
            return new SetPoint()
            {
                T1Position = this.T1Position,
                T2Position = this.T2Position,
                A1Position = this.A1Position,
                A2Position = this.A2Position,
                A3Position = this.A3Position,
                B1Position = this.B1Position,
                B2Position = this.B2Position,
                B3Position = this.B3Position,
                C1Position = this.C1Position,
                C2Position = this.C2Position,
                C3Position = this.C3Position,
                D1Position = this.D1Position,
                D2Position = this.D2Position,
                D3Position = this.D3Position,
            };
        }
    }
}
```

NodeSavingReading.cs

```
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Sender
{
    public class NodeSavingReading
    {

        public void pushDataToFile(String fileLocation, List<Node>
nodesToSave)
        {
            List<Node> listOfNodes = nodesToSave.ToList<Node>();
            string output = JsonConvert.SerializeObject(listOfNodes);
            using (StreamWriter sw = new
StreamWriter(Globals.TreeSaveLocation))
            {
                sw.WriteLine(output);
            }
        }

        public void pushDataToFile(String fileLocation, List<SetPoint>
pointsToSave)
        {
            List<SetPoint> listofSetPoints = pointsToSave.ToList<SetPoint>();
            string output = JsonConvert.SerializeObject(listofSetPoints);
            using (StreamWriter sw = new StreamWriter(fileLocation))
            {
                sw.WriteLine(output);
            }
        }

        public void pushDataToFile(String fileLocation, List<double[]>
pointsToSave)
        {
            List<double[]> listofSetPoints = pointsToSave.ToList<double[]>();
            string output = JsonConvert.SerializeObject(listofSetPoints);
            using (StreamWriter sw = new StreamWriter(fileLocation))
            {
                sw.WriteLine(output);
            }
        }

        public void pushDataToFile(String fileLocation, KFoldData dataPoint)
        {
            string output = JsonConvert.SerializeObject(dataPoint);
            using (StreamWriter sw = new StreamWriter(fileLocation))
            {
                sw.WriteLine(output);
            }
        }

        public List<Node> GetDataFromFile(String fileLocation)
        {
            if (File.Exists(fileLocation))
            {
                string inputData = File.ReadAllText(fileLocation);
                try
                {
                    return
JsonConvert.DeserializeObject<List<Node>>(inputData);
```

```
                }
                catch (Exception e)
                {
                    return new List<Node>();
                }
            }
            else
            {
                return new List<Node>();
            }
        }

        public List<double[]> GetStoredDataFromFile(String fileLocation)
        {
            if (File.Exists(fileLocation))
            {
                string inputData = File.ReadAllText(fileLocation);
                try
                {
                    return
JsonConvert.DeserializeObject<List<double[]>>(inputData);
                }
                catch (Exception e)
                {
                    return new List<double[]>();
                }
            }
            else
            {
                return new List<double[]>();
            }
        }

        public List<SetPoint> GetSetPointDataFromFile(String fileLocation)
        {
            if (File.Exists(fileLocation))
            {
                string inputData = File.ReadAllText(fileLocation);
                try
                {
                    return
JsonConvert.DeserializeObject<List<SetPoint>>(inputData);
                }
                catch (Exception e)
                {
                    return new List<SetPoint>();
                }
            }
            else
            {
                return new List<SetPoint>();
            }
        }

        public KFoldData GetKFoldDataFromFile(String fileLocation)
        {
            if (File.Exists(fileLocation))
            {
                string inputData = File.ReadAllText(fileLocation);
                try
                {
                    return
JsonConvert.DeserializeObject<KFoldData>(inputData);
                }
                catch (Exception e)
                {
                    return null;
                }
            }
            else
            {
                return null;
            }
        }
    }
}
```

PatsControlScheme.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Newtonsoft.Json;

namespace Sender
{
    class PatsControlScheme : ControlSystemInterface
    {
        public int timeNeededForChange = 10000;//in milliseconds
        public Node root = null;
        public Dictionary<int, Node> allNodes = new Dictionary<int, Node>();
        //max value is 9 due to indexing implementation
        public int childrenPerNode = 1;//default value

        public void DetermineSetpointsFromInputs()
        {

        }

        public void PushInformationToHand()
        {

        }

        public void Initialize()
        {
            if (!GetDataFromFile())
            {
                instantiateNewTree(1, 10000);
                childrenPerNode = 1;
            }
            else
            {
                if (root.children.Count > 0)
                {
                    childrenPerNode = root.children.Count - 1;
                }
                else
                {
                    childrenPerNode = 1;
                }
            }
        }

        public bool GetDataFromFile()
        {
            if (File.Exists(Globals.TreeSaveLocation))
            {
                string inputData =
File.ReadAllText(Globals.TreeSaveLocation);
                List<Node> retrievedNodes = null;
                try
                {
                    retrievedNodes =
JsonConvert.DeserializeObject<List<Node>>(inputData);
                }
                catch (Exception e)
                {
                    return false;
                }

                if (retrievedNodes == null)
                {
                    return false;
                }
                for(int i = 0; i < retrievedNodes.Count; i++)
                {
                    if(retrievedNodes[i].name == null || (
(retrievedNodes[i].id < 0 || retrievedNodes[i].getHandPosition() == null) &&
retrievedNodes[i].id != Globals.CONTROLNODE))
                    {
                        //check each node to make sure they saved correctly
                        return false;
                    }
                }
                if (retrievedNodes != null)
                {
                    bool foundRoot = false;
                    foreach (Node n in retrievedNodes)
                    {
                        allNodes.Add(n.id, n);
                        if (!foundRoot && n.parent == Globals.NULLPARENT)
                        {
                            root = n;
                            foundRoot = true;
                        }
                        if (n.id != Globals.CONTROLNODE &&
!n.children.Contains(Globals.CONTROLNODE) && n.children.Count > 0)
                        {
                            n.children.Add(Globals.CONTROLNODE);
                        }
                    }
                    foreach(Node n in allNodes.Values)
                    {
                        if (n.id != Globals.CONTROLNODE &&
n.children.Contains(Globals.CONTROLNODE))
                        {
                            n.children.Remove(Globals.CONTROLNODE);
                            n.children.Add(Globals.CONTROLNODE);
                        }
                    }
                    return true;
                }
                return false;
            }
            else
            {
                return false;
            }
        }

        public void instantiateNewTree(int positionsPerSplit, int delay)
        {
            this.allNodes.Clear();
            this.root = createNewNode(Globals.ROOTNODE, positionsPerSplit,
Globals.NULLPARENT);
            this.childrenPerNode = positionsPerSplit;

            this.timeNeededForChange = delay;
        }

        public Node createNewNode(int id, int positionsPerSplit, int
parentID)
        {
            Node newNode = new Node("Extended Hand", new SetPoint(), id, new
List<int>(positionsPerSplit), parentID);
            if (id != Globals.CONTROLNODE)
            {
                this.allNodes.Add(newNode.id, newNode);
            }
            if(parentID != Globals.NULLPARENT && id != Globals.CONTROLNODE)
            {
                allNodes[parentID].children.Add(id);
            }
            return newNode;
        }

        public void pushDataToFile()
        {
            List<Node> listOfNodes = allNodes.Values.ToList<Node>();
            string output = JsonConvert.SerializeObject(listOfNodes);
            using (StreamWriter sw = new
StreamWriter(Globals.TreeSaveLocation))
            {
                sw.WriteLine(output);
            }
        }

        public void cleanupReferences()
        {
            List<Node> everyNode = allNodes.Values.ToList().OrderBy(k =>
k.id).ToList();
            foreach (Node n in allNodes.Values)
            {
                if(!everyNode.Exists(node => node.parent == n.parent)){
                    allNodes.Remove(n.id);
                }
            }
        }
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Sender
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Globals.welcomeScreen = new WelcomeScreen();
            Application.Run(Globals.welcomeScreen);
        }
    }
}
```

SerialReader.cs

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Linq;
using System.Runtime.Serialization;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using Accord;
using Newtonsoft.Json.Linq;

namespace Sender
{
    class SerialReader
    {
        private volatile double[] dataOut;
        private volatile double[] betaDataOut;
        private volatile List<double[]> lastDataOut;
        private Mutex bciDataLock;
        private SerialPort serialPort1;
        private int rate;

        //Scale for OpenBCI data to mV (highest setting)
        private static float scale = 0.02235f;

        //Previous data for filter
        static double[,] prev_x_notch = new double[8, 5];
        static double[,] prev_y_notch = new double[8, 5];
        static double[,] prev_x_standard = new double[8, 5];
        static double[,] prev_y_standard = new double[8, 5];

        static double[,] prev_x_notchBeta = new double[8, 5];
        static double[,] prev_y_notchBeta = new double[8, 5];
        static double[,] prev_x_standardBeta = new double[8, 5];
        static double[,] prev_y_standardBeta = new double[8, 5];

        public SerialReader()
        {
            bciDataLock = new Mutex();
            serialPort1 = new SerialPort("COM5", 115200);
            serialPort1.Open();
            serialPort1.Write("s");
            serialPort1.Write("~5");
            dataOut = new double[16];
            lastDataOut = new List<double[]>();

            setRate(250);
        }

        public int getRate()
        {
            return this.rate;
        }

        //Set rate in Hz
        public void setRate(double desiredRate) { rate = (int)(desiredRate *
255 / 250); }

        //Starts board output
        public void Start() { serialPort1.Write("b"); }

        //Stops board output
        public void Stop() { serialPort1.Write("s"); }

        //Reads board output
        public void Read()
        {
            Start();

            Task dataReader = new Task(getData);
            dataReader.Start();
        }
        public double[] GetData()
        {
            bciDataLock.WaitOne();
            double[] returnData = dataOut;
            bciDataLock.ReleaseMutex();
            return returnData;

        }

        private void getData()
        {
            var inData = new Byte[32];
            bool frequencyToggle = true;
                while (true)
                {
                    try
                    {
                        bciDataLock.WaitOne();
                    }
                    catch (AbandonedMutexException e)
                    {
                        bciDataLock.ReleaseMutex();
                    }


                    if (serialPort1.ReadByte() == 0xA0)
                    {
                        serialPort1.Read(inData, 0, 32);
                        if (inData[31] > 0xBF && inData[31] < 0xD0 &&
inData[0] <= rate)
                        {
                            var loggingData = new double[8];
                            for (int i = 0; i < 8; i++)
                            {
                                int outVal = interpret24bitAsInt32(inData[i *
3 + 1], inData[i * 3 + 2],
                                    inData[i * 3 + 3]);
                                dataOut[i] = (double) (outVal * scale);
                                if (frequencyToggle)
                                {
                                    dataOut[(i+8)] = FilterBeta(dataOut[i],
i);

                                }
                                dataOut[i] = Filter(dataOut[i], i);
                                loggingData[i] = dataOut[i];
                                if (lastDataOut.Count > 5 && dataOut[i] ==
lastDataOut.First()[i])
                                {
                                    dataOut[i] = double.NaN;
                                    Console.WriteLine("Node " + (i+1) +" is
not connected");
                                }
                                else if (i == 7 && lastDataOut.Count > 5) {

                                    Console.WriteLine("Node " + (i + 1) + "
is connected with value " + dataOut[i]);
```

```
                                lastDataOut.RemoveAt(0);
                            }
                            else
                            {
                                Console.WriteLine("Node " + (i + 1) + "
is connected with value " + dataOut[i]);
                            }

                        }

                        lastDataOut.Add(loggingData);
                    }

                    frequencyToggle = !frequencyToggle;
                }



                bciDataLock.ReleaseMutex();
            }
        }

        //Provided by OpenBCI
        public int interpret24bitAsInt32(byte byte1, byte byte2, byte byte3)
        {
            int newInt = (
                ((0xFF & byte1) << 16) |
                ((0xFF & byte2) << 8) |
                (0xFF & byte3)
              );
            if ((newInt & 0x00800000) > 0)
            {
                newInt = (int)((uint)newInt | (uint)0xFF000000);
            }
            else
            {
                newInt = (int)((uint)newInt & (uint)0x00FFFFFF);
            }
            return (newInt);
        }

        //Filtering function for OpenBCI Nodes
        //Adapted from nekrodezynfekator's OpenBCI_GUI repository
        private double Filter(double inputVal, int i)
        {
            double returnVal = 0;
            var b = new double[5] { 0.1173510367246093, 0, -
0.2347020734492186, 0, 0.1173510367246093 };
            var a = new double[5] { 1, -2.137430180172061, 2.038578008108517,
-1.070144399200925, 0.2946365275879138 };
            var b2 = new double[5] { 0.9650809863447347, -0.2424683201757643,
1.945391494128786, -0.2424683201757643, 0.9650809863447347 };
            var a2 = new double[5] { 1, -0.2467782611297853,
1.944171784691352, -0.2381583792217435, 0.9313816821269039 };

            for (int j = 4; j > 0; j--)
                {
                    prev_x_notch[i, j] = prev_x_notch[i, j - 1];
                    prev_y_notch[i, j] = prev_y_notch[i, j - 1];
                    prev_x_standard[i, j] = prev_x_standard[i, j - 1];
                    prev_y_standard[i, j] = prev_y_standard[i, j - 1];
                }

                prev_x_notch[i, 0] = inputVal;

                double score = 0;

                for (int j = 0; j < 5; j++)
                {
                    score += b2[j]*prev_x_notch[i, j];
                    if (j > 0)
                    {
                        score -= a2[j]*prev_y_notch[i, j];
                    }
                }

                prev_y_notch[i, 0] = score;
                prev_x_standard[i, 0] = score;
                score = 0;
                for (int j = 0; j < 5; j++)
                {
                    score += b[j]*prev_x_standard[i, j];
                    if (j > 0)
                    {
                        score -= a[j]*prev_y_standard[i, j];
                    }
                }

                prev_y_standard[i, 0] = score;
                returnVal = score;

            return returnVal;
        }

        //Filter modified for Beta waves while recording at higher
frequencies
        private double FilterBeta(double inputVal, int i)
        {
            double returnVal = 0;
            var b = new double[5] { 0.1173510367246093, 0, -
0.2347020734492186, 0, 0.1173510367246093 };
            var a = new double[5] { 1, -2.137430180172061, 2.038578008108517,
-1.070144399200925, 0.2946365275879138 };
            var b2 = new double[5] { 0.96508099, -1.19328255, 2.29902305, -
1.19328255, 0.96508099 };
            var a2 = new double[5] { 1, -1.21449347931898, 2.29780334191380,
-1.17207162934772, 0.931381682126902 };

            for (int j = 4; j > 0; j--)
            {
                prev_x_notchBeta[i, j] = prev_x_notchBeta[i, j - 1];
                prev_y_notchBeta[i, j] = prev_y_notchBeta[i, j - 1];
                prev_x_standardBeta[i, j] = prev_x_standardBeta[i, j - 1];
                prev_y_standardBeta[i, j] = prev_y_standardBeta[i, j - 1];
            }

            prev_x_notchBeta[i, 0] = inputVal;

            double score = 0;

            for (int j = 0; j < 5; j++)
            {
                score += b2[j] * prev_x_notchBeta[i, j];
                if (j > 0)
                {
                    score -= a2[j] * prev_y_notchBeta[i, j];
                }
            }

            prev_y_notchBeta[i, 0] = score;
            prev_x_standardBeta[i, 0] = score;
            score = 0;
            for (int j = 0; j < 5; j++)
            {
                score += b[j] * prev_x_standardBeta[i, j];
                if (j > 0)
                {
                    score -= a[j] * prev_y_standardBeta[i, j];
                }
            }

            prev_y_standardBeta[i, 0] = score;
            returnVal = score;

            return returnVal;
        }
    }
}
```

SetPoint.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Sender
{
    public class SetPoint
    {
        public float T1Position = 0.0f;
        public float T2Position = 0.0f;
        public float A1Position = 0.0f;
        public float A2Position = 0.0f;
        public float A3Position = 0.0f;
        public float B1Position = 0.0f;
        public float B2Position = 0.0f;
        public float B3Position = 0.0f;
        public float C1Position = 0.0f;
        public float C2Position = 0.0f;
        public float C3Position = 0.0f;
        public float D1Position = 0.0f;
        public float D2Position = 0.0f;
        public float D3Position = 0.0f;

        public SetPoint()
        {
            this.T1Position = 0.0f;
            this.T2Position = 0.0f;
            this.A1Position = 0.0f;
            this.A2Position = 0.0f;
            this.A3Position = 0.0f;
            this.B1Position = 0.0f;
            this.B2Position = 0.0f;
            this.B3Position = 0.0f;
            this.C1Position = 0.0f;
            this.C2Position = 0.0f;
            this.C3Position = 0.0f;
            this.D1Position = 0.0f;
            this.D2Position = 0.0f;
            this.D3Position = 0.0f;
        }

        public SetPoint(float T1, float T2, float A1, float A2, float A3,
float B1, float B2, float B3, float C1, float C2, float C3, float D1, float
D2, float D3)
        {
            this.A1Position = A1;
            this.A2Position = A2;
            this.A3Position = A3;
            this.B1Position = B1;
            this.B2Position = B2;
            this.B3Position = B3;
            this.C1Position = C1;
            this.C2Position = C2;
            this.C3Position = C3;
            this.D1Position = D1;
            this.D2Position = D2;
            this.D3Position = D3;
            this.T1Position = T1;
            this.T2Position = T2;
        }

        public double[] ConvertToDoubles()
        {
            var returnArray = new double[14];
            returnArray[0] = A1Position;
            returnArray[1] = A2Position;
            returnArray[2] = A3Position;
            returnArray[3] = B1Position;
            returnArray[4] = B2Position;
            returnArray[5] = B3Position;
            returnArray[6] = C1Position;
            returnArray[7] = C2Position;
            returnArray[8] = C3Position;
            returnArray[9] = D1Position;
            returnArray[10] = D2Position;
            returnArray[11] = D3Position;
            returnArray[12] = T1Position;
            returnArray[13] = T2Position;

            return returnArray;
        }
    }
}
```

Threshold.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Sender
{
    public class Threshhold
    {

        public double min;
        public double max;

        public Threshhold(double min, double max)
        {
            this.min = min;
            this.max = max;
        }

        public bool Contains(double input)
        {
            if(input >= min && input < max)
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        public void Set(double INmin, double INmax)
        {
            this.min = INmin;
            this.max = INmax;
        }
    }
}
```

UnityCommunicationHub.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Sender
{
    public static class UnityCommunicationHub
    {
        private static string directoryPath = @"c:\BCIDataDirectory";
        private static string filePath =
@"c:\BCIDataDirectory\transferInfo.txt";
        private static string mutexFileTurn =
@"c:\BCIDataDirectory\WFATurn.mutex";
        private static string mutexUnityTurn =
@"c:\BCIDataDirectory\UnityTurn.mutex";
        private static string unityReadyToGo =
@"c:\BCIDataDirectory\UnityReady.txt";
        private static string WFAReadyToGo =
@"c:\BCIDataDirectory\WFAReady.txt";

        private static bool initialized = false;

        public static bool connected = false;

        //initializes file communication system with Unity. Should ONLY be
called once at program start.
        public static bool InitializeUnityCommunication()
        {
            if (initialized)
            {
                Console.WriteLine("ERROR: TRYING TO INITIALIZE AFTER
INITIALIZATION SUCCESSFUL");
                return true;
            }
            // Determine whether the directory exists.
            if (Directory.Exists(directoryPath))
            {
                Console.WriteLine("Directory path exists already. Proceeding
as normal.");
            }
            else
            {
                // Try to create the directory.
                Console.WriteLine("Creating directory path.");
                DirectoryInfo di = Directory.CreateDirectory(directoryPath);
            }

            //make sure to clean up any excess data from last run
            if (File.Exists(filePath))
            {
                File.Delete(filePath);
            }
            if (File.Exists(mutexFileTurn))
            {
                File.Delete(mutexFileTurn);
            }
            if (File.Exists(mutexUnityTurn))
            {
                File.Delete(mutexUnityTurn);
            }
            using (StreamWriter sw = new StreamWriter(mutexFileTurn))
            {
                sw.WriteLine("g");
            }
            using (StreamWriter sw = new StreamWriter(WFAReadyToGo))
            {
                sw.WriteLine("g");
            }
            using (StreamWriter sw = new StreamWriter(filePath))
            {
                sw.WriteLine("START");
            }
            //wait for confirmation step
            Stopwatch watch = Stopwatch.StartNew();
            watch.Start();
            while (!File.Exists(unityReadyToGo))
            {
                if(watch.ElapsedMilliseconds > Globals.TimeToConnectToUnity)
                {
                    Console.WriteLine("Unity connection attempt stopped");
                    return false;
                }
            }
            initialized = true;
            connected = true;
            return true;
        }

        //performs both the read and write
        public static bool TwoWayTransmission()
        {
            if (!ReadData(false))
            {
                Console.WriteLine("ERROR IN READING FROM UNITY");
                return false;
            }
            if (!WriteData(false))
            {
                Console.WriteLine("ERROR IN WRITING TO FILE TO TRANSMIT TO
UNITY");
                return false;
            }
            switchToUnity();
            return true;
        }

        //returns true if it was able to aquire the file to read and then
write to the file, false otherwise
        //input true unless using as an intermediate step
        public static bool ReadData(bool turnOverToUnityAfter = true)
        {
            if (File.Exists(mutexFileTurn))
            {
                //first get the position from the hand
                string line = "";
                using (StreamReader sr = new StreamReader(filePath))
                {
                    if((line = sr.ReadLine()) != null){
                        //make sure we're the recipient
                        if(line.Equals("TO WFA"))
                        {
                            //get data if we're the recipient
                            while ((line = sr.ReadLine()) != null)
                            {
                                switch (line.Substring(0, 2))
                                {
                                    case "T1":
                                        Globals.T1ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "T2":
                                        Globals.T2ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "A1":
                                        Globals.A1ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "A2":
                                        Globals.A2ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "A3":
                                        Globals.A3ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "B1":
                                        Globals.B1ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "B2":
                                        Globals.B2ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "B3":
                                        Globals.B3ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "C1":
                                        Globals.C1ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "C2":
                                        Globals.C2ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "C3":
                                        Globals.C3ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "D1":
                                        Globals.D1ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                        break;
                                    case "D2":
```

```
                                    Globals.D2ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                    break;
                                case "D3":
                                    Globals.D3ActualPosition =
(float)System.Convert.ToDouble(line.Substring(2));
                                    break;
                            }

                        }
                    }
                }

            }

            if (turnOverToUnityAfter)
            {
                switchToUnity();
            }
            return true;
        }
        else
        {
            return false;
        }
    }

    //write the global variables to the file to transmit to unity.
automatically switches to unity reading after
    public static bool WriteData(bool turnOverToUnityAfter)
    {
        //get the most recent data
        switchToUnity();
        Stopwatch timer = new Stopwatch();
        timer.Start();
        while (!File.Exists(mutexFileTurn))
        {
            if(timer.ElapsedMilliseconds > 5000)
            {
                Console.WriteLine("Could not read from unity");
                throw new Exception();
            }
        }
        //and now write your own data to the file
        File.Delete(filePath);
        using (StreamWriter sw = new StreamWriter(filePath))
        {
            sw.WriteLine("TO UNITY");
            sw.WriteLine("T1" + Globals.T1DesiredPosition);
            sw.WriteLine("T2" + Globals.T2DesiredPosition);
            sw.WriteLine("A1" + Globals.A1DesiredPosition);
            sw.WriteLine("A2" + Globals.A2DesiredPosition);
            sw.WriteLine("A3" + Globals.A3DesiredPosition);
            sw.WriteLine("B1" + Globals.B1DesiredPosition);
            sw.WriteLine("B2" + Globals.B2DesiredPosition);
            sw.WriteLine("B3" + Globals.B3DesiredPosition);
            sw.WriteLine("C1" + Globals.C1DesiredPosition);
```

```
                sw.WriteLine("C2" + Globals.C2DesiredPosition);
                sw.WriteLine("C3" + Globals.C3DesiredPosition);
                //sw.WriteLine("D1" + Globals.D1DesiredPosition);
                sw.WriteLine("D2" + Globals.D2DesiredPosition);
                sw.WriteLine("D3" + Globals.D3DesiredPosition);
                sw.WriteLine("From sender");
            }
            if (turnOverToUnityAfter)
            {
                switchToUnity();
            }
            return true;
        }

        public static void switchToUnity()
        {
            bool ready = false;
            while (!ready)
            {
                try
                {
                    File.Delete(mutexFileTurn);
                    ready = true;
                }
                catch (Exception e)
                {

                }
            }
            using (StreamWriter sw = new StreamWriter(mutexUnityTurn))
            {
                sw.WriteLine("G");
            }
        }

        private static bool isMyTurn()
        {
            if (File.Exists(mutexFileTurn))
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        //deletes all files from transmission directory. Should ONLY be
called upon program exit.
        public static void PurgeFileSystem()
        {
            File.Delete(unityReadyToGo);
            File.Delete(WFAReadyToGo);
            File.Delete(filePath);
            File.Delete(mutexFileTurn);
            File.Delete(mutexUnityTurn);
            connected = false;
            initialized = false;
        }

    }
}
```

UnsureNetworkForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Controls;
using System.Windows.Forms;
using System.Windows.Media.Converters;
using Accord.Math;
using Accord.Neuro.Learning;

namespace Sender
{
    public partial class UnsureNetworkForm : Form
    {
        private NeuralNet net;
        private SerialReader serial;
        private int currentHandPosition;
        private List<double[]> inputTrainingData;
        private List<double[]> outputTrainingData;
        private object dataLock = new object();
        private Dictionary<string, int> indexList;
        private Dictionary<int, string> reverseIndexList;
        private Dictionary<int, SetPoint> setPointList;

        Stopwatch timer = new Stopwatch();
        private List<Threshhold> ranges;
        PatsControlScheme controls;
        int rate;

        private string ANNfilename = Globals.NeuralNetSaveLocation;
        private string KFoldFilename = Globals.KFoldDataSaveLocation;

        public UnsureNetworkForm()
        {
            InitializeComponent();

            NodeSavingReading reader = new NodeSavingReading();

            net = new NeuralNet(7, 7, ANNfilename, KFoldFilename);
            inputTrainingData =
reader.GetStoredDataFromFile(Globals.inputDataStorage);
            outputTrainingData =
reader.GetStoredDataFromFile(Globals.outputDataStorage);
```

```
            //inputTrainingData = new List<double[]>();
            //outputTrainingData = new List<double[]>();

            controls = new PatsControlScheme();
            controls.Initialize();
            controls.timeNeededForChange = 7000;

            serial = new SerialReader();
            serial.Read();

            UnityCommunicationHub.InitializeUnityCommunication();
            UnityCommunicationHub.TwoWayTransmission();


            indexList = Globals.GetBasicValues();
            reverseIndexList = Globals.GetBasicValuesReversed();
            setPointList = Globals.GetBasicPositions();

            foreach (KeyValuePair<string, int> position in indexList)
            {
                DefaultPositionsBox.Items.Add(position.Key);
            }



        }


        private void Run()
        {
            //while (true)
            {
                lock (dataLock)
                {

                    var input = serial.GetData();
                    double[] inputData = new double[7];
                    for (int j = 0; j < 8; j++)
                    {
                        if (j < 1) inputData[j] = input[j];
                        else if (j > 1) inputData[j - 1] = input[j];
                    }
                    var percievedPositionArray = net.Think(inputData);

                    double bestVal = 0;
                    SetPoint bestSetPoint = new SetPoint();
                    bool goodToMove = false;

                    while (!goodToMove)
                    {
                        var index = 0;
                        for (int i = 0; i < percievedPositionArray.Length;
i++)
                        {
                            if (percievedPositionArray[i] > bestVal)
                            {
                                bestVal = percievedPositionArray[i];
```

```
                                    index = i;
                                    bestSetPoint = setPointList[i];
                                }
                            }

                            if (bestVal > 0.95)
                            {
                                goodToMove = true;
                            }
                            else
                            {
                                PositionTextBox.Text = reverseIndexList[index];
                                goodToMove = RunFocus();
                                if (!goodToMove)
                                {
                                    percievedPositionArray[index] = 0;
                                    bestVal = 0;
                                }
                            }
                        }

                        var percievedPosition = bestSetPoint;

                        Globals.T1DesiredPosition = percievedPosition.T1Position;
                        Globals.T2DesiredPosition = percievedPosition.T2Position;
                        Globals.A1DesiredPosition = percievedPosition.A1Position;
                        Globals.A2DesiredPosition = percievedPosition.A2Position;
                        Globals.A3DesiredPosition = percievedPosition.A3Position;
                        Globals.B1DesiredPosition = percievedPosition.B1Position;
                        Globals.B2DesiredPosition = percievedPosition.B2Position;
                        Globals.B3DesiredPosition = percievedPosition.B3Position;
                        Globals.C1DesiredPosition = percievedPosition.C1Position;
                        Globals.C2DesiredPosition = percievedPosition.C2Position;
                        Globals.C3DesiredPosition = percievedPosition.C3Position;
                        Globals.D1DesiredPosition = percievedPosition.D1Position;
                        Globals.D2DesiredPosition = percievedPosition.D2Position;
                        Globals.D3DesiredPosition = percievedPosition.D3Position;

                        UnityCommunicationHub.WriteData(true);
                    }

                }
            }

            private void Train()
            {
                lock (dataLock)
                {
                    var networkTrainingInput = new
double[inputTrainingData.Count][];
                    var networkTrainingOutput = new
double[outputTrainingData.Count][];
                    for (int i = 0; i < inputTrainingData.Count; i++)
                    {
                        networkTrainingInput[i] = inputTrainingData[i];
                        networkTrainingOutput[i] = outputTrainingData[i];
                    }
                    net.Train(networkTrainingInput, networkTrainingOutput, 100,
    .10f);
                }
            }

            private void DefaultPositionsBox_SelectedIndexChanged(object sender,
    EventArgs e)
            {

                var inputItemName = (System.Windows.Forms.ListBox)sender;
                currentHandPosition =
    indexList[(string)inputItemName.SelectedItem];
            }



            private void SaveButton_Click(object sender, EventArgs e)
            {
                NodeSavingReading reader = new NodeSavingReading();
                net.Save();
                reader.pushDataToFile(Globals.inputDataStorage,
    inputTrainingData);
                reader.pushDataToFile(Globals.outputDataStorage,
    outputTrainingData);
            }

            private void logButton_Click(object sender, EventArgs e)
            {
                Thread.Sleep(200);
                for (int i = 0; i < 50; i++)
                {
                    double[] inData = serial.GetData();
                    double[] inputData = new double[7];
                    for (int j = 0; j < 8; j++)
                    {
                        if (j < 1) inputData[j] = inData[j];
                        else if (j > 1) inputData[j - 1] = inData[j];
                    }
                    inputTrainingData.Add(inputData);
                    double[] outputData = new double[7];
                    outputData[currentHandPosition] = 1;
                    outputTrainingData.Add(outputData);
                    Thread.Sleep(1);
                }
            }

            private void TrainButton_Click(object sender, EventArgs e)
            {
                Thread trainingThread = new Thread(Train);
                trainingThread.Start();
            }

            private void TestButton_Click(object sender, EventArgs e)
            {
                Thread.Sleep(200);
                //Thread testThread = new Thread(Run);
                //testThread.Start();
                Run();
            }


            private double[] ScaleOutputStorageData(double[] inputData)
            {
                var returnData = new double[inputData.Length];
                for (int i = 0; i < inputData.Length; i++)
                {
                    returnData[i] = inputData[i] / 90;
                }

                return returnData;
            }

            private float[] ScaleOutputData(double[] inputData)
            {
                var returnData = new float[inputData.Length];
                for (int i = 0; i < inputData.Length; i++)
                {
                    returnData[i] = (float)inputData[i] * (90);
                }

                return returnData;
            }

            private void FocusButton_Click(object sender, EventArgs e)
            {
                    int desiredMillisecondDelay = controls.timeNeededForChange;
                    int arraySize = (desiredMillisecondDelay / 1000) * rate;
                    serial.Read();
                    rate = serial.getRate();

                    double lowConcentration = 0;
                    double highConcentration = 0;

                    //first get threshholds
                    bool done = false;
                    while (!done)
                    {

                        int reads = 0;
                        decimal allReads = 0;
                        MessageBox.Show("First try to let your mind wander until
    the next popup appears. Hit OK when ready.", string.Empty,
    MessageBoxButtons.OK);
                        timer.Start();
                        while (timer.ElapsedMilliseconds <
    Globals.threshholdAquisitionTime)
                        {
                            decimal currentIn =
    (decimal)Math.Abs(serial.GetData()[Globals.inputNode]);
                            allReads += currentIn;
                            reads++;
                        }
                        timer.Reset();
                        lowConcentration = (double)(allReads / reads);
                        Console.WriteLine("Low concentration was: " +
    lowConcentration);

                        reads = 0;
                        allReads = 0;

                        MessageBox.Show("Next try to focus as hard as possible
    something. Hit OK when ready.", string.Empty, MessageBoxButtons.OK);

                        timer.Start();
                        while (timer.ElapsedMilliseconds <
    Globals.threshholdAquisitionTime)
                        {
                            decimal currentIn =
    (decimal)Math.Abs(serial.GetData()[Globals.inputNode]);
                            allReads += currentIn;
                            reads++;
                        }
                        timer.Reset();

                        highConcentration = (double)(allReads / reads);

                        Console.WriteLine("High concentration was: " +
    highConcentration);

                        if (highConcentration > lowConcentration)
                        {
                            done = true;
                        }
                    }

                    double differenceInConcentrations = highConcentration -
    lowConcentration;
                    double deltaBetweenThreshholds = differenceInConcentrations /
    2;

                    //make ranges for this run
                    ranges = new List<Threshhold>(2);


                            //make sure all reads work for it
                            ranges.Add(new Threshhold(Double.MinValue,
    lowConcentration + (deltaBetweenThreshholds)));

                            ranges.Add(new Threshhold(lowConcentration +
    (deltaBetweenThreshholds), Double.MaxValue));


                    MessageBox.Show("Ready to control hand. Press OK when
    ready.", string.Empty, MessageBoxButtons.OK);
            }

            private bool RunFocus()
            {
```

```
            int desiredMillisecondDelay = controls.timeNeededForChange;
            timer = new Stopwatch();

                //get the inputs and average them for the desired output
                double averageInput = 0;
                Random rand = new Random();
                var numInputs = 0;
                decimal accruedValues = 0;
                timer.Start();
                decimal currentIn;
                while (timer.ElapsedMilliseconds < desiredMillisecondDelay)
                {
                    currentIn =
(decimal)Math.Abs(serial.GetData()[Globals.inputNode]);
                    accruedValues += currentIn;
                    numInputs++;
                    double currentVal = (double)(accruedValues / numInputs);
                    if (currentVal > ranges[0].max)
                    {
                        FocusTextBox.Text = "Yes, " + currentVal;
                    }
                    else
                    {
                        FocusTextBox.Text = "No, " + currentVal;
                    }
                }
                averageInput = (double)(accruedValues / numInputs);

                if (averageInput > ranges[0].max)
                {
                    return true;
                }
                else
                {
                    return false;
                }



        }

        private void FocusTextBox_TextChanged(object sender, EventArgs e)
        {

        }
    }
}
```

WelcomeScreen.cs

```
namespace Sender
{
    partial class WelcomeScreen
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be
disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.NeuralTreeButton = new System.Windows.Forms.Button();
            this.NeuralNetButton = new System.Windows.Forms.Button();
            this.BasicFunctionalityButton = new
System.Windows.Forms.Button();
            this.clearDataButton = new System.Windows.Forms.Button();
            this.UnsureNetworkButton = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // NeuralTreeButton
            //
            this.NeuralTreeButton.Location = new System.Drawing.Point(32,
11);
            this.NeuralTreeButton.Margin = new
System.Windows.Forms.Padding(2);
            this.NeuralTreeButton.Name = "NeuralTreeButton";
            this.NeuralTreeButton.Size = new System.Drawing.Size(108, 77);
            this.NeuralTreeButton.TabIndex = 0;
            this.NeuralTreeButton.Text = "Neural Tree";
            this.NeuralTreeButton.UseVisualStyleBackColor = true;
            this.NeuralTreeButton.Click += new
System.EventHandler(this.NeuralTreeButton_Click);
            //
            // NeuralNetButton
            //
            this.NeuralNetButton.Location = new System.Drawing.Point(224,
11);
            this.NeuralNetButton.Margin = new
System.Windows.Forms.Padding(2);
            this.NeuralNetButton.Name = "NeuralNetButton";
            this.NeuralNetButton.Size = new System.Drawing.Size(108, 77);
            this.NeuralNetButton.TabIndex = 1;
            this.NeuralNetButton.Text = "Neural Net";
            this.NeuralNetButton.UseVisualStyleBackColor = true;
            this.NeuralNetButton.Click += new
System.EventHandler(this.NeuralNetButton_Click);
            //
            // BasicFunctionalityButton
            //
            this.BasicFunctionalityButton.Location = new
System.Drawing.Point(32, 100);
            this.BasicFunctionalityButton.Margin = new
System.Windows.Forms.Padding(2);
            this.BasicFunctionalityButton.Name = "BasicFunctionalityButton";
            this.BasicFunctionalityButton.Size = new System.Drawing.Size(108,
77);
            this.BasicFunctionalityButton.TabIndex = 2;
            this.BasicFunctionalityButton.Text = "Continuous Neural Net";
            this.BasicFunctionalityButton.UseVisualStyleBackColor = true;
            this.BasicFunctionalityButton.Click += new
System.EventHandler(this.BasicFunctionalityButton_Click);
            //
            // clearDataButton
            //
            this.clearDataButton.Location = new System.Drawing.Point(147,
218);
            this.clearDataButton.Margin = new
System.Windows.Forms.Padding(2);
            this.clearDataButton.Name = "clearDataButton";
            this.clearDataButton.Size = new System.Drawing.Size(108, 77);
            this.clearDataButton.TabIndex = 3;
            this.clearDataButton.Text = "Clear All Data";
            this.clearDataButton.UseVisualStyleBackColor = true;
            //
            // UnsureNetworkButton
            //
            this.UnsureNetworkButton.Location = new System.Drawing.Point(224,
100);
            this.UnsureNetworkButton.Margin = new
System.Windows.Forms.Padding(2);
            this.UnsureNetworkButton.Name = "UnsureNetworkButton";
            this.UnsureNetworkButton.Size = new System.Drawing.Size(108, 77);
            this.UnsureNetworkButton.TabIndex = 4;
            this.UnsureNetworkButton.Text = "Unsure Network";
            this.UnsureNetworkButton.UseVisualStyleBackColor = true;
            this.UnsureNetworkButton.Click += new
System.EventHandler(this.UnsureNetworkButton_Click);
            //
            // WelcomeScreen
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(400, 335);
            this.Controls.Add(this.UnsureNetworkButton);
            this.Controls.Add(this.clearDataButton);
            this.Controls.Add(this.BasicFunctionalityButton);
            this.Controls.Add(this.NeuralNetButton);
            this.Controls.Add(this.NeuralTreeButton);
            this.Margin = new System.Windows.Forms.Padding(2);
            this.Name = "WelcomeScreen";
            this.Text = "WelcomeScreen";
            this.ResumeLayout(false);

        }

        #endregion

        private System.Windows.Forms.Button NeuralTreeButton;
        private System.Windows.Forms.Button NeuralNetButton;
        private System.Windows.Forms.Button BasicFunctionalityButton;
        private System.Windows.Forms.Button clearDataButton;
        private System.Windows.Forms.Button UnsureNetworkButton;
    }
}
```