Ian Spetnagel

University of Bridgeport - Bridgeport, Connecticut

MS Artificial Intelligence Capstone Research Paper

Fall 2023

School of Engineering - RISC Laboratory

Advised by Professor Sarosh Patel

**Abstract**

An exploration of neural network based obstacle avoidance research conducted throughout the Fall of 2023 for the benefit of a final capstone research paper. This study aimed to assess the feasibility of using back-propagation algorithms with an Arduino uno microcontroller board. Further context regarding the research methods and sources will be discussed throughout the report.

**Introduction and Robot Platform**

Neural networks provide the foundation for machine learning and artificial intelligence decision making and have been vital for the growth and increased sophistication of expert computer systems. As we move closer to an autonomous world with greater reliance on non-human decision making, neural network capability and integration have continued to grow exponentially. In the human brian, neurons act as the messenger between different areas of the brain and one’s entire body. Neural networks are no different connecting several neurons to one another and to different layers for use in a program.

The three types of layers can be classified as input, output and hidden layers, which are where much of the computational function occurs. While a neural network can have multiple hidden layers, generally, they only operate with a single input and single output layer. Furthemore, the number of neurons included will depend on the specific purpose of a project and its respective requirements for processing and subsequent decision making. Neural networks provide the foundation for the integration of AI in robotics including obstacle and collision avoidance, path planning and navigation, motor and manipulator control and many other applications.

Initially the aim of this project was to duplicate the efforts of Tim Zimmerman (University of Hartford, West Hartford, CT) is his 2014 work with developing obstacle avoidance using simulated sensor data in a neural network. His study was characterized by attaching a hobby servo to an infrared sensor to a small mobile robot in order to have 180º of sensor visibility. Using LabVIEW, an engineering workbench software, he could control the movements of the robot after training. Essentially, the sensor would retrieve data from a full range of motion and send information to the neural network for processing. It would then use the trained network for decision making on which direction to move in next.

Similarly this report will explore the use of backpropagation neural networks and their implementation with the Arduino Uno microcontroller. The robotic chassis was the KEYESTUDIO Mini Tank Robot V2 [Figure 1] which was used in a previous obstacle avoidance robot. Its current hardware makes use of an ultrasonic distance sensor, coupled with a servo motor for the robots ability to scan left and right of its viewpath thus not having to alter any of the hardware for the integration of the neural network. The controller for robotic operation was completed in the Arduino IDE and can be viewed in the index of this document. Furthermore, the configuration of the neural network and structure of nodes and layers was also completed in the Arduino IDE.

| **Figure 1 - KEYESTUDIO Mini Tank Robot V2** | **Research Methodologies : Neural Network Framework and Implementation**  Though several facets of the project were vital in its development the most important was the integration and configuration of the neural network. In order for the robot to correctly maneuver its way through the environment with obstacle avoidance, the IR sensor must use the trained neural network for its decision making. Touched on above, neural networks can be configured with different numbers of hidden layers within the input and output layers. Their functionality is used in decision making by taking different inputs in order to produce some sort of conclusion in the output layer. |
| --- | --- |

In addition to this, each of the neuron layers (nodes) are connected. There are weights assigned to each node that determine their strength and direction. As data passes through the network, each input is manipulated by the weights in each layer, and passed through an activation function for the calculation of the output. For this project, each of these weights was initially randomized but then adjusted in the process of learning. Zimmerman explains it similarly for his work as follows:

“Through each synapse the data is modified by a weight value, which is determined during network learning. All of the previous layer’s neuron outputs are connected to each neuron in the subsequent layer, where they are summed and applied to an activation function. This value is then output to each neuron in the subsequent layer, where the process is repeated until it reaches the output layer.” [Zimmerman, 2014]

Following this the learning rates would be revised. In regard to this project, as different data was passed through the network, the values were examined and compared to the expected outputs from the training data. Initially, training data was fed to the neural network, along with what decisions could be ‘expected’ for it to produce. Digit representations were used for the inputs, and paired with target outputs for each input pattern. The training data is shown with an array in the Arduino code as *const byte Input* and the expected output array *const byte Target*. The data would train the network until a low error was achieved. To be certain the training of the network was functional, the input patterns, target outputs and current network output were all displayed in the console every thousand iterations. Moreover, each training cycle would run until its error rate fell below the success threshold which was set to have a value of 4/10000 and illustrated in the network’s code as *const float Success = 0.0004*. Subsequently, the succeeding array (inputs and outputs) would run any number of iterations until again falling below the success threshold. Each iteration showing a reduction in error. This is illustrated below in Figure 2.

| **Figure 2 - Training output in Serial Monitor Console of Ardunio** | ← Input and Target Patterns with Network Output  ← Expectation NOT met (Error = 0.00048 > 0.0004)  ← Expectation met  (Error < 0.0004)  ← Training set complete |
| --- | --- |

The process of training the network began with randomizing the weights in the hidden and output layers. Similarly, the training patterns were shuffled (randomly) in order to improve the learning of the network. The forward pass portion of the training generated the activations for the hidden and output layer along with error computation; this was done using the sigmoid function. Subsequently, these error values were used in backpropagation which then adjusted the weights in both layers. Though the network used was a simple feedforward process with a single hidden layer, it was ideal for simple pattern recognition in obstacle avoidance. As was hard-coded in the previous obstacle avoidance project, the network aided in manageable if this, then that decision making.

To help define the architecture of the network, the number of input, output, and hidden nodes were introduced at the top of the code before use. The input layer consisted of 7 nodes, hidden layer 8, and the output layer 4, each interconnected with one another. This is visualized in Figure 3.

| **Figure 3 - Illustration of Neural Network** | The learning rate of the network was set to 0.3 having an update of 30% in the gradient descent algorithm at each step through the network. This felt optimal as it maintained a reasonable training rate and limited training error.  Additionally, the initial weight maximum was calibrated as 0.5 which again was ideal for the network’s learning parameters, but allowed for changes throughout dependent on the results from the sigmoid function. Both of these values were also defined prior to training.  The training of the neural network was certainly the most challenging and time consuming aspect of the project. |
| --- | --- |

**Analysis and Robot Navigation**

Zimmerman’s project was outlined by the method of having a simulated target or destination point given to the robot prior to its obstacle avoidance decision making. For the majority of his experiment, the objective was relatively limitless as much of his work used simulated sensor data in the LabVIEW environment, as opposed to physical testing. This project however did not use any simulated data outside of what was fed to the neural network as training data. Thus the decision making revolved solely around the robot's interactions in its environment.

Previously, programming directives would dictate the robots behavior in and around its environment from predefined, hard-coded conditions; measurements from the ultrasonic distance sensor would guide its decision making on when to stop and afterwards turn. Zimmerman’s work was similar but allowed the robot to have a map of its area. Using vision sensors and different colored (pink and green) post-it notes, allowed the robot's neural network to understand its position and heading within the local environment. LabVIEW would present the robot with a target point in the given x,y plane, then use the network to map a path to it.

As the robot traversed its way to its destination, it would map its environment and the obstacles encountered. When close to obtaining the objective point (within 10 cm) the program would generate a new target point and path to success.

Although similar, not using sensor data or target points, offered a much different approach for this project. The neural network implemented was strictly called on when the robot faced an interaction with its environment. Each time it encountered an obstacle it would adjust its internal parameters and desired outcomes; respectively, these were IR sensor readings modifying the weights in order to accomplish the task of obstacle avoidance. This forced the robot to adapt to its surroundings but offered more nuanced decision making capabilities for subsequent interactions.

However, the decision making came at a much slower rate than the previously programmed obstacle avoidance. Due to the computational demands of input processing through multiple layers of the network, and calculating output the decision for turning and stopping was much slower and less accurate. At times, obstacle recognition was late and caused minor collisions with obstacles around the robot’s environment. Even with the simple architecture and parameters of the neural network, the time taken for forward propagation is inherently longer than the execution of direct control commands.

Furthermore, it is vital to consider the reliability of the neural networks decision-making. Having direct programming rules for IR sensor readings offers predictable and consistent responses to obstacle interaction around the environment. The performance of the neural network is contingent on the quality and extent of its training. Inadequately trained networks will exhibit less optimal and likely slower decision making than rule based logic.

**Conclusion**

The exploration of neural network driven decision making presents an intriguing idea at what is to come in the future of autonomous systems. Though this project supports the idea that training a back propagation neural network for obstacle avoidance is possible, it is not perfect. Generally, the robot did very well to avoid obstacles but was not without fail. High computational demands with distance sensor reading and implementation into the network while moving, caused some minor collisions and errors in decision making.

Using the KEYESTUDIO Mini Tank Robot V2 and Arduino Uno board provided a functional platform for the implementation of the network with the minor concern of having to retrain the network upon restart or power down. Moreover, each time it would be powered on it would behave slightly differently due to changes in the weights between each node.

In principle, the robot would be able to use its network for prediction of all obstacles or interactions it may encounter, by mapping itself within its environment. However, as the robot relies on continuous adjustment to its network and changes in its surroundings, the network performed very well for obstacle avoidance and provides an excellent example as to what autonomous obstacle avoidance systems may look like in the future.

**Works Cited**

Heymsfeld, Ralph. “A Neural Network for Ardunio - .” Hobbizine Robotics, robotics.hobbizine.com/arduinoann.html. Accessed Dec. 2023.

Zimmerman, Timothy A. West Hartford, Connecticut, 2014, pp. 1–8, Neural Network Based Obstacle Avoidance Using Simulated Sensor Data.

**Index**

**Figure 4 - Arduino Code for Neural Network**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* ArduinoANN - An artificial neural network for the Arduino**

**\* All basic settings can be controlled via the Network Configuration**

**\* section.**

**\* See robotics.hobbizine.com/arduinoann.html for details.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**#include <math.h>**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Network Configuration - customized per network**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**const int PatternCount = 10;**

**const int InputNodes = 7;**

**const int HiddenNodes = 8;**

**const int OutputNodes = 4;**

**const float LearningRate = 0.3;**

**const float Momentum = 0.9;**

**const float InitialWeightMax = 0.5;**

**const float Success = 0.0004;**

**const byte Input[PatternCount][InputNodes] = {**

**{ 1, 1, 1, 1, 1, 1, 0 }, // 0**

**{ 0, 1, 1, 0, 0, 0, 0 }, // 1**

**{ 1, 1, 0, 1, 1, 0, 1 }, // 2**

**{ 1, 1, 1, 1, 0, 0, 1 }, // 3**

**{ 0, 1, 1, 0, 0, 1, 1 }, // 4**

**{ 1, 0, 1, 1, 0, 1, 1 }, // 5**

**{ 0, 0, 1, 1, 1, 1, 1 }, // 6**

**{ 1, 1, 1, 0, 0, 0, 0 }, // 7**

**{ 1, 1, 1, 1, 1, 1, 1 }, // 8**

**{ 1, 1, 1, 0, 0, 1, 1 } // 9**

**};**

**const byte Target[PatternCount][OutputNodes] = {**

**{ 0, 0, 0, 0 },**

**{ 0, 0, 0, 1 },**

**{ 0, 0, 1, 0 },**

**{ 0, 0, 1, 1 },**

**{ 0, 1, 0, 0 },**

**{ 0, 1, 0, 1 },**

**{ 0, 1, 1, 0 },**

**{ 0, 1, 1, 1 },**

**{ 1, 0, 0, 0 },**

**{ 1, 0, 0, 1 }**

**};**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* End Network Configuration**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**int i, j, p, q, r;**

**int ReportEvery1000;**

**int RandomizedIndex[PatternCount];**

**long TrainingCycle;**

**float Rando;**

**float Error;**

**float Accum;**

**float Hidden[HiddenNodes];**

**float Output[OutputNodes];**

**float HiddenWeights[InputNodes+1][HiddenNodes];**

**float OutputWeights[HiddenNodes+1][OutputNodes];**

**float HiddenDelta[HiddenNodes];**

**float OutputDelta[OutputNodes];**

**float ChangeHiddenWeights[InputNodes+1][HiddenNodes];**

**float ChangeOutputWeights[HiddenNodes+1][OutputNodes];**

**void setup(){**

**Serial.begin(9600);**

**randomSeed(analogRead(3));**

**ReportEvery1000 = 1;**

**for( p = 0 ; p < PatternCount ; p++ ) {**

**RandomizedIndex[p] = p ;**

**}**

**}**

**void loop (){**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Initialize HiddenWeights and ChangeHiddenWeights**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( i = 0 ; i < HiddenNodes ; i++ ) {**

**for( j = 0 ; j <= InputNodes ; j++ ) {**

**ChangeHiddenWeights[j][i] = 0.0 ;**

**Rando = float(random(100))/100;**

**HiddenWeights[j][i] = 2.0 \* ( Rando - 0.5 ) \* InitialWeightMax ;**

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Initialize OutputWeights and ChangeOutputWeights**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( i = 0 ; i < OutputNodes ; i ++ ) {**

**for( j = 0 ; j <= HiddenNodes ; j++ ) {**

**ChangeOutputWeights[j][i] = 0.0 ;**

**Rando = float(random(100))/100;**

**OutputWeights[j][i] = 2.0 \* ( Rando - 0.5 ) \* InitialWeightMax ;**

**}**

**}**

**Serial.println("Initial/Untrained Outputs: ");**

**toTerminal();**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Begin training**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( TrainingCycle = 1 ; TrainingCycle < 2147483647 ; TrainingCycle++) {**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Randomize order of training patterns**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( p = 0 ; p < PatternCount ; p++) {**

**q = random(PatternCount);**

**r = RandomizedIndex[p] ;**

**RandomizedIndex[p] = RandomizedIndex[q] ;**

**RandomizedIndex[q] = r ;**

**}**

**Error = 0.0 ;**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Cycle through each training pattern in the randomized order**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( q = 0 ; q < PatternCount ; q++ ) {**

**p = RandomizedIndex[q];**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Compute hidden layer activations**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( i = 0 ; i < HiddenNodes ; i++ ) {**

**Accum = HiddenWeights[InputNodes][i] ;**

**for( j = 0 ; j < InputNodes ; j++ ) {**

**Accum += Input[p][j] \* HiddenWeights[j][i] ;**

**}**

**Hidden[i] = 1.0/(1.0 + exp(-Accum)) ;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Compute output layer activations and calculate errors**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( i = 0 ; i < OutputNodes ; i++ ) {**

**Accum = OutputWeights[HiddenNodes][i] ;**

**for( j = 0 ; j < HiddenNodes ; j++ ) {**

**Accum += Hidden[j] \* OutputWeights[j][i] ;**

**}**

**Output[i] = 1.0/(1.0 + exp(-Accum)) ;**

**OutputDelta[i] = (Target[p][i] - Output[i]) \* Output[i] \* (1.0 - Output[i]) ;**

**Error += 0.5 \* (Target[p][i] - Output[i]) \* (Target[p][i] - Output[i]) ;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Backpropagate errors to hidden layer**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( i = 0 ; i < HiddenNodes ; i++ ) {**

**Accum = 0.0 ;**

**for( j = 0 ; j < OutputNodes ; j++ ) {**

**Accum += OutputWeights[i][j] \* OutputDelta[j] ;**

**}**

**HiddenDelta[i] = Accum \* Hidden[i] \* (1.0 - Hidden[i]) ;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Update Inner-->Hidden Weights**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( i = 0 ; i < HiddenNodes ; i++ ) {**

**ChangeHiddenWeights[InputNodes][i] = LearningRate \* HiddenDelta[i] + Momentum \* ChangeHiddenWeights[InputNodes][i] ;**

**HiddenWeights[InputNodes][i] += ChangeHiddenWeights[InputNodes][i] ;**

**for( j = 0 ; j < InputNodes ; j++ ) {**

**ChangeHiddenWeights[j][i] = LearningRate \* Input[p][j] \* HiddenDelta[i] + Momentum \* ChangeHiddenWeights[j][i];**

**HiddenWeights[j][i] += ChangeHiddenWeights[j][i] ;**

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Update Hidden-->Output Weights**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( i = 0 ; i < OutputNodes ; i ++ ) {**

**ChangeOutputWeights[HiddenNodes][i] = LearningRate \* OutputDelta[i] + Momentum \* ChangeOutputWeights[HiddenNodes][i] ;**

**OutputWeights[HiddenNodes][i] += ChangeOutputWeights[HiddenNodes][i] ;**

**for( j = 0 ; j < HiddenNodes ; j++ ) {**

**ChangeOutputWeights[j][i] = LearningRate \* Hidden[j] \* OutputDelta[i] + Momentum \* ChangeOutputWeights[j][i] ;**

**OutputWeights[j][i] += ChangeOutputWeights[j][i] ;**

**}**

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Every 1000 cycles send data to terminal for display**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**ReportEvery1000 = ReportEvery1000 - 1;**

**if (ReportEvery1000 == 0)**

**{**

**Serial.println();**

**Serial.println();**

**Serial.print ("TrainingCycle: ");**

**Serial.print (TrainingCycle);**

**Serial.print (" Error = ");**

**Serial.println (Error, 5);**

**toTerminal();**

**if (TrainingCycle==1)**

**{**

**ReportEvery1000 = 999;**

**}**

**else**

**{**

**ReportEvery1000 = 1000;**

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* If error rate is less than pre-determined threshold then end**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**if( Error < Success ) break ;**

**}**

**Serial.println ();**

**Serial.println();**

**Serial.print ("TrainingCycle: ");**

**Serial.print (TrainingCycle);**

**Serial.print (" Error = ");**

**Serial.println (Error, 5);**

**toTerminal();**

**Serial.println ();**

**Serial.println ();**

**Serial.println ("Training Set Solved! ");**

**Serial.println ("--------");**

**Serial.println ();**

**Serial.println ();**

**ReportEvery1000 = 1;**

**}**

**void toTerminal()**

**{**

**for( p = 0 ; p < PatternCount ; p++ ) {**

**Serial.println();**

**Serial.print (" Training Pattern: ");**

**Serial.println (p);**

**Serial.print (" Input ");**

**for( i = 0 ; i < InputNodes ; i++ ) {**

**Serial.print (Input[p][i], DEC);**

**Serial.print (" ");**

**}**

**Serial.print (" Target ");**

**for( i = 0 ; i < OutputNodes ; i++ ) {**

**Serial.print (Target[p][i], DEC);**

**Serial.print (" ");**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Compute hidden layer activations**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( i = 0 ; i < HiddenNodes ; i++ ) {**

**Accum = HiddenWeights[InputNodes][i] ;**

**for( j = 0 ; j < InputNodes ; j++ ) {**

**Accum += Input[p][j] \* HiddenWeights[j][i] ;**

**}**

**Hidden[i] = 1.0/(1.0 + exp(-Accum)) ;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* Compute output layer activations and calculate errors**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**for( i = 0 ; i < OutputNodes ; i++ ) {**

**Accum = OutputWeights[HiddenNodes][i] ;**

**for( j = 0 ; j < HiddenNodes ; j++ ) {**

**Accum += Hidden[j] \* OutputWeights[j][i] ;**

**}**

**Output[i] = 1.0/(1.0 + exp(-Accum)) ;**

**}**

**Serial.print (" Output ");**

**for( i = 0 ; i < OutputNodes ; i++ ) {**

**Serial.print (Output[i], 5);**

**Serial.print (" ");**

**}**

**}**

**}**