Learning Live Autonomous Navigation: A Model Car with Hardware Arduino Neurons

Mohammad O. Khan and Gary Parker
Department of Computer Science
Connecticut College
New London, CT, USA
parker@conncoll.edu, mkhan4@conncoll.edu

***Abstract* — Previously we developed an implementation for an easily expandable hardware Artificial Neural Network (ANN) capable of learning using inexpensive, off-the-shelf Arduino Pro Mini microprocessors. This ANN system is unique, general use, unspecialized, and inexpensive. The implementation involves a one neuron per microchip representation, a ratio which allows for computational parallelism and ANN architecture flexibility inherent to biological neural networks. Learning happens completely on hardware via backpropagation without the need for communication with a computer. Tests showed successful, dynamic learning of the logical operations OR, AND, XOR, and XNOR. In this paper, we demonstrate the usage and strength of this implementation by applying the same framework to learn live obstacle avoidance and autonomous navigation for a 1:24 scale model car equipped with ultrasonic distance sensors. This test of the application involved a user who supervised the learning and a method to easily transition between testing and training the ANN on the car via Bluetooth. Results show that the hardware ANN consistently learns to navigate the car through an obstacle course from entrance to exit and vice versa with no collisions.**

Keywords—Artificial Neural Networks; hardware; Arduino; backpropagation; dynamic learning; autonomous navigation; car

I. INTRODUCTION

A biological neural network consists of billions of interconnected neurons. An Artificial Neural Network (ANN) attempts to model such a network for biologically inspired control applications with learning capabilities. Most ANNs are implemented in simulations and hardware implementations usually have the entire ANN on one chip. This research involves a unique representation of an ANN whereby each microchip represents a single neuron. The commercial market has many hardware ANNs, but no general use, unspecialized, and inexpensive models exist with such representation. Generally, extra modules and processing parts allow the chip to function as a single neuron. More commonly, multiple neurons are implemented on one chip. These designs generally require a good grasp of electrical engineering to fully understand the data being mapped as voltage, current, and similar elements are used to model data. Specialized and expensive parts are often needed to manipulate these features. Furthermore, many of the designs are specifically built for certain tasks, cannot be added to bigger systems easily, and do not have learning capabilities.

In previous work we presented a more manageable way to create general hardware ANNs with learning capabilities [1]. A hardware ANN was built using Arduino Pro Mini (APM) microchips, which are popular, low-priced microcontrollers, coupled with open-source software. Each APM chip acts as a single neuron with learning capabilities to permit a more genuine imitation of a biological neural network. The architecture of the ANN consists of one APM chip handling the input and three other APM chips, two as neurons in the hidden layer and one as a neuron in the output layer. The communication in the ANN is supported by the I2C (Inter-Integrated Circuit) of Arduino which creates a Master-Slave relationship between the output chip, the hidden layer chips, and the input chip. The finished network learned four different logic operations successfully: AND, OR, XOR, and XNOR.

Researchers have explored many ANN architectures to solve complex problems. We chose to build this proof of concept based on the notion of solving more fundamental logic operators. It is commonly known that to solve the XOR logic operator a hidden layer is needed for the ANN. By building this functionality we can validate that we have the performance of a full ANN and not that of a single Perceptron.

This paper discusses the real-world application of learning control for autonomous car navigation using the previously developed hardware Arduino ANN. A small car was constructed to incorporate 3 ultrasonic ping sensors as input with the output going to two full-rotation servos for locomotion. The same ANN design was used as the previous research except the input element was replaced with direct data input to the hidden layer from the sensors. The car was driven wirelessly by a user via an application written in the Processing prototyping language, which delivered the desired output via Bluetooth at the live moment to the car as it maneuvered around blocks in an 8x8 foot testing space. Transitions between testing and training were made to incrementally record the progress of the car's learning. This paper shows that the framework allows for a highly flexible testing/training platform and that autonomous car navigation can be successfully learned in a multi-path obstacle colony space.

II. Related Works

A. Hardware Neural Networks

Significant development of dedicated hardware to develop faster and more genuine networks that are processed in parallel has been done in the past two decades [2]. Even then, there is room for improvement. Dias, Antunes, and Mota note in a review of commercial ANN hardware that such hardware is specialized and expensive in regards to manufacturing time and resources, and not much is known about its commercial implications [3]. Goser notes one limitation as dedicated, complex wiring due to specialized hardware [4]. Generally, multiple neurons exist on one chip and often a full ANN is put on one chip. Liao notes in a survey of hardware that the majority of designs include an activation block, weights block, transfer function block, and other processing elements [5]. The activation block is always on board as opposed to the other blocks. Thus, these implementations do not focus on building a hardware neuron in a single chip. Moreover, other products that explore learning through the backpropagation algorithm, but they deal more hardware on the parallel machine level than the microcontroller level [6].

A similar method to this research is the use of Field Programmable Gate Arrays (FPGAs). FPGAs are integrated circuits that can be configured physically. Significant work has been done with FPGAs and ANNs [7, 9, 10]. Sahin, Becerikli and Yazici implemented a multiple layer ANN by linking neurons together with multipliers and adders using an FPGA [8]. This framework makes the architecture of the ANN dynamic. The weights were learned offline, and no clear learning was done on the hardware. Importantly, this work used an accessible, prototyping product to develop an ANN. However, even though the ANN architecture is adjustable, it requires physical configuration and rededication of parts within the hardware.

A common theme with hardware ANNs is having multiple neurons exist on one chip. Ienne and Kuhn analyze multiple commercial chips [11]. Such as the Philips' Lneuro-1, which consists of 16 neurons. Or the Philips' Lneuro-2.3, which consists of 12 neurons. The Ricoh RN-200 also consists of 16 neurons in a multi-layered ANN with backpropagation learning. Other implementations add extra parts to speed up the chip. For example, the Intel's Ni1000 consists of three parts where the microcontroller is joined with a classifier and an external interface for conversion calculation to decrease overhead on the chip.

Other intricate methods for designing neurons have been explored. Joubert, Belhadj, Temam, and Héliot analyze the Leaky Integrate-and-Fire neuron. It is a single neuron representation, but it has a special architecture split among three parts: the synapse, the neuron core, and the comparator [12]. Several of these implementations utilize electrical properties such as current, capacitance, and voltage. Although this is a more genuine imitation of a biological neuron, it requires electrical engineering expertise to design and reproduce, and the parts are usually specialized and expensive.

In summary, much research exists in the development of hardware ANNs. However, such hardware implementations are generally built with specific parts and expensive resources – a clear general, unspecialized, and inexpensive "one neuron to one chip" implementation with learning capabilities does not exist. We provide an alternative; a hardware ANN capable of learning through the backpropagation algorithm with a one neuron to one chip ratio using inexpensive, readily available APM chips. The implementation is also accessible to typical users.

B. Autonomous Car Navigation using Neural Networks

Much work has been done in autonomous car navigation, and most research is done with full sized cars as opposed to small model cars such as in this paper. Pomerleau contributed to the roots of this research in 1989 with the development of ALVINN (An Autonomous Land Vehicle in a Neural Network) whereby the ANN was trained offline using simulated road images, and testing was done on a truck like vehicle [13]. Two years later, he improved this design by allowing "on the fly" training by permitting human driving to be desired output in the ANN [14]. Jonathan, Chandrasekhar, and Srinivasan developed an innovative approach by building a sensor driven network such as the research in this paper versus relying on image processing overhead. Different parts of the decision making such as turning and overtaking had their own algorithms implemented within the system. This low processing overhead significantly improves time spent for the ANN in learning. A more similar work was done on a miniature car which was capable of navigation in unknown environments by Farooq, Amar, Asad, Hanif, and Saleh [16]. The training, however, happened off-line and the learned weights were then put on a controller.

None of these designs utilize a distributed hardware ANN such as in this research. Also, an improvement in the transition between training and testing is presented. On board transitioning is done easily and quickly with one simple application command as opposed to waiting for training to be finished before testing. For example, Pomerleau's systems need human driver training for five minutes followed by ten minutes of backpropagation before testing can be done.

III. Artificial Neural Networks

Every neuron in an Artificial Neural Network (ANN) receives inputs with corresponding weights and produces only one output. One neuron's output is an input for another neuron in a subsequent layer. For example, Figure 1 has an input layer (no neurons, just three inputs), a two neuron hidden layer, and single neuron output layer. A threshold constant (θ) represents the activation of a biological neuron, and usually a weighted sum of the inputs is matched to this value to produce an output. The threshold is set to -1 as an input with a corresponding weight in this paper. The weighted sum (X) of the inputs goes into an activation function to produce an output (Y). This process is called activation. This paper uses the sigmoid function:

$$Y = \frac{1}{1 + e^{-X}} \tag{1}$$

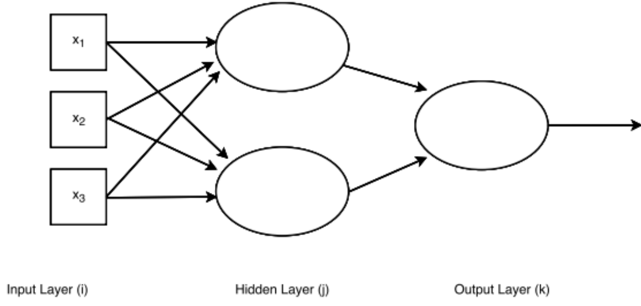

Figure 1. Visual representation of a multi-layered ANN with three inputs, two hidden layer neurons, and an output layer neuron.

The combination of forward propagation and backpropagation offer a common technique for learning in ANNs when these two stages are repeated continually on input/output pairs. Initially, the weights in each neuron are set to random values. The input layer gives the input to each hidden layer neuron. The hidden layer neurons give their outputs to the next layer neurons. Such activations continue to be forward propagated until the output layer neurons are reached. An actual output is produced at the output layer. This value is compared to a desired output for the current input. An error value is calculated as the difference between the two values. Backpropagation begins with this value by calculating error gradients. These values adjust weights for multi-layered ANNs since hidden layers do not have clear desired output values. Below, the equation formatting is similar to that which is used in the Neural Networks chapter of Artificial Intelligence: A Guide to Intelligent Systems by Negnevitsky [12]. Several different activation functions exist in ANN learning, and we employ the sigmoid activation function for this research. The application of the backpropagation algorithm follows as such:

1) The output layer (k) neuron error gradient (δ) at the current iteration (t) is calculated.

$$\delta_k(t) = Y_k(t) \times [1 - Y_k(t)] \times e_k(t) \qquad (2)$$

Y_k = actual output for output layer
e_k = error at output layer neuron such that:

$$e_k(t) = Y_{d_k}(t) - Y_k(t) \qquad (3)$$

Y_{d_k} = desired output at the output layer neuron.

2) The weights (W_{j_k}) between the hidden layer (j) and output layer (k) are updated for the next iteration (t+1).

$$W_{j_k}(t+1) = W_{j_k}(t) + [\, \alpha \times Y_j(t) \times \delta_k(t) \,] \qquad (4)$$

Y_j = actual output at the hidden layer
α = constant learning rate

3) The error gradient (δ) for the neurons in the hidden layer (j) is calculated. *This is the case when there is a single output neuron.*

$$\delta_j(t) = Y_j(t) \times [\, 1 - Y_j(t) \,] \times [\, \delta_k(t) \times W_{j_k}(t) \,] \qquad (5)$$

4) The weights (W_{i_j}) between the input layer (i) and hidden layer (j) are updated.

$$W_{i_j}(t+1) = W_{i_j}(t) + [\, \alpha \times x_i(t) \times \delta_j(t) \,] \qquad (6)$$

$x_i(t)$ = input to the hidden layer neuron at iteration t

5) Continuously loop through the forward propagation and backpropagation cycle.

IV. Implementation of Hardware Neural Network

The 5V and 16MHz Arduino Pro Mini (APM) was selected for this research because it is widely available, small enough (.7’’ x 1.3’’) for quick prototyping of ANN architectures, and inexpensive ($3 to $10 depending on distributer). The APM has 14 digital input/output pins.

Communication between neurons was controlled via the Inter-Integrated Circuit (I2C) bus, a built-in part of the APM. Each neuron has its own memory address assigned within the circuit. The I2C is a single bus which permits communication through a Serial Clock Line and a Serial Data Line. This creates Master-Slave relationships between APMs. The Master chip starts the clock and requests data from the Slave chips. The Slave chips only respond to Master requests. Pin A4 accesses the Data Line and pin A5 accesses the Clock Line. These two pins allow simple wiring for communication.
To manipulate the I2C, the Arduino Wire library was needed. It comes with the Arduino Integrated Development Environment. The two primary commands used were read() and write(), thus data flows in two directions. Commands such as requestFrom() and available() were also used. The former allows a Master to ask for data, and the latter returns true if data is available from a Slave.

A. I2C Data Communication

Arduino limits data transfer in the I2C to bytes or characters. Thus, a double precision value (4 bytes) cannot be transferred simply over wire. Precise weight values are vital for learning portion of the ANN. A Union data type was used to work around this limitation: *union T {byte b[4]; double d;} T;* Hence, different data types can be stored in the same memory location. Here the same memory location includes a double and a byte. A double is encoded as an array of bytes. It is then sent over the Wire interface to another APM neuron. The receiving end reverses this masking to obtain the double precision value. While this seems like a trivial note, it is important to clarify that this is a mandatory implementation step in order for the network to be able to transfer data with double precision and to update in an effective manner within the constraints of the I2C framework.

B. Programming of Individual Microchips

Each microchip was programmed using the C based Arduino language. Different neuron classes were developed for hidden layer and output layer neurons. For the I2C bus framework, each APM class included a memory address. Arrays of type double

within each hidden and output neuron represent inputs and weights. The learning rate is set at .35.

a) Hidden Neuron Microchip (Slave)

A requestEvent() function waits for a request from the output APM Master neuron. Once triggered, an output value for the neuron is produced, packaged, and sent over wire in forward propagation. A receiveEvent() function is triggered with any data from the output neuron in backpropagation. This neuron receives a 4 byte error gradient from the output neuron, with which the weights are updated. Meanwhile, three ultrasonic sensors input data to the neuron.

b) Output Neuron Microchip (Master)

A readHidden() function requests an output from each hidden layer neuron. An actual output is produced once the inputs are received. The neuron continuously accepts data packets via an HC-05 Bluetooth module from a Processing application running on a remote laptop. Depending on the data, a desired output is determined [(0), (.5), (1)] respectively based on the turn [(left), (forward), (right)]. Using the desired output and a calculated output an error is calculated, and backpropagation starts. Also received from the application is data that triggers training mode to be initiated.

C. *Neural Network Circuit*

For this application there are 3 APM microchips. Each microchip is a single artificial neuron. As shown in Figure 2, there are 3 inputs coming in ultrasonic sensors into two hidden layer neurons in the same hidden layer, and the output of those neurons is input into one output layer neuron. This ANN architecture mirrors Figure 1. The communication works in a way such that the output neuron is the Master APM and the hidden neurons are the Slave APMs. The I2C bus is interlaced using the A4 and A5 pins of all the APMs. A wire for Data and a wire for Clock allows this connection. Power is provided through the VCC pin for each APM, and all APMs are grounded through the GND pin. Four AAA batteries were used to power the servos, and a single 9V battery was used to power all the APM chips, through which the ultrasonic sensors were also powered. For presentation purposes, an LED is attached to the output layer neuron to visualize the toggling between testing and training.

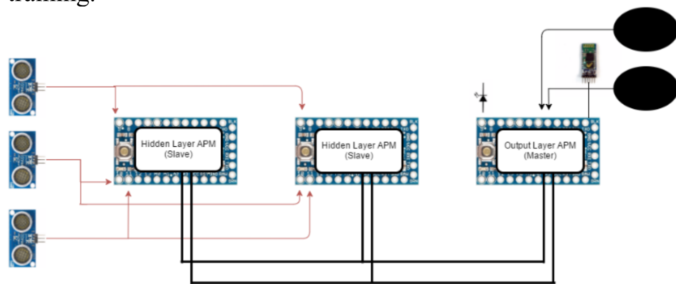

Figure 2. A schematic of the APM circuit is shown. Each neuron is labeled. Input comes in from 3 ultrasonic sensors to each hidden layer neuron. An LED is attached to the output layer neuron to show testing mode in action. A Bluetooth module (rectangle) and two full rotation servos (black ovals) are also attached to the output layer neuron.

D. *Car Design*

A foot long wooden chassis was cut long enough to hold the hardware Arduino ANN (Figure 3). Three Parallax Ultrasonic Ping))) sensors were placed on a cardboard frame and attached to the edge of the chassis 40° apart (Figure 3). The range of the sensors is between 2cm to 300cm. However, the inverse of each sensor value is taken and multiplied by 100 to bias the learning towards closer obstacles by producing larger numbers. Thus, the range lies between the value of .333 for far objects, and 50 for close objects. Two full rotation Parallax servos were attached at the end of the chassis to hold two 2" diameter plastic wheels. The car moves forward at a constant rate of about 6 inches per second when there is no user input. Both right and left turns are approximately the same angle and speed, which is defined by a signal to the appropriate servo telling it to turn at half speed. This signal is sent to the servo every 20 milliseconds while the turn command is active. Different amounts of turn are achieved by running the function again and again over a certain period of time. This equates to the driver pressing and holding a key. Two omni-directional wheels are attached to the back end of the chassis with an axle to allow for smooth turning.

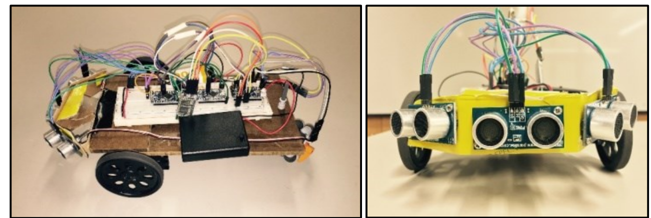

Figure 3. The car used in this research with its various elements and a view of the placement of the ultrasonic sensors on the front.

E. *Navigation Space and Task*

The task of the car is to learn to complete the path in the 8x8 foot colony space with 1x1 foot block obstacles as shown in Figure 4 from both directions. A full training and testing iteration is counted as the summed distance of both paths taken (384 inches).

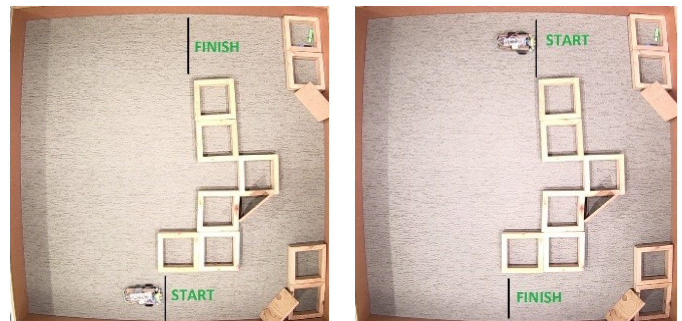

Figure 4.The car learns to complete the path from both directions.

F. Algorithm

The user constantly supervises the car; driving it by sending control commands as desired output through a Processing application. Bluetooth delivers computer keyboard inputs: 4 as left, 5 as right, enter as start testing, backspace to end testing, otherwise move the car forward using the servos. Note that although the human driver has full control of the car during the training phase, he or she does not send error signals to the ANN, instead the error is calculated automatically within the APM chips by comparing the ANN's output direction with the direction input by the user as he/she continuously navigates the course with the Processing application.

The following algorithm demonstrates the learning process for the hardware Arduino ANN through car movement:

1) Upon receiving power, all APM neurons are inputted a threshold variable of -1 within their programs, and random weights are set between -1 and 1 within each neuron program.

2) While the steps below are happening, the user is navigating the course wirelessly through keyboard input on the computer to drive the car in such a way that is avoids any collisions with the obstacles.

3) The car is placed in the START position as in the left image in Figure 4.

4) Forward propagation starts at the output (Master) neuron when the user sends a driving command to the hardware Arduino ANN. It requests 4 bytes from each hidden node. Meanwhile, hidden layer neurons are constantly getting updated input about any obstacles from the ultrasonic sensors.

5) Each hidden layer neuron forward propagates, and sends an output to the Master neuron.

6) Activation happens at the output layer, and the Master neuron produces an actual output. The Master also registers the current data from the driver as the desired output.

7) Using the actual output and the desired output, an error is calculated at the Master neuron. Using this, an output layer error gradient is also calculated. This value is used to calculate a hidden layer error gradient.

8) This hidden layer error gradient is sent to each hidden layer neuron. The weights of the output layer are updated using the output layer error gradient, and the weights of the hidden layer are updated using the hidden layer error gradient.

9) Steps 4 through 8 are repeated until the FINISH line is reached. The car is put into test mode, and reversed to face the new path, then put into train mode and Steps 3-9 are repeated for the backwards path as in the right image in Figure 4. This marks the end of one iteration.

10) Testing mode is initiated. The car is placed as in step 3, put into test mode via the Processing application, and allowed to autonomously navigate. The distance where the first collision occurs is recorded along with how many iterations have elapsed. We loop back to step 3 for further training.

V. Results

The vehicle successfully completed navigation of the course in 5 different trials. Without any training, the car is able to navigate to about 13 inches into the path before a collision. After one iteration of training, the distance traveled is about 8 inches greater than before. After five iterations of training, the average distance traveled is around 75 inches. At six iterations, a major jump of distance traveled occurs on average of 182 inches traveled – just 10 inches shy of completing half the course. Furthermore, the least number of iterations it took to complete the full 384 inch course for any trial was 29 iterations, while the most was 34 iterations (Figure 5). This range of performance is fairly consistent, but is also heavily dependent on how consistently the driver is able to navigate the path. In the initial runs, the car was observed to develop bad "habits" from being supervised by the user. This included taking an unorthodox path by staying close to the right side of the path. In cases like these, only one or two sensor inputs were being mapped and learned. However, once consistency was developed by the driver to keep the car in the middle "lane" of the path, the ANN was able to learn obstacle avoidance on all sensors and sides and stay more in the middle of the path.

One issue with the car was that its forward motion was not always consistently straight throughout the course. This probably added to the time spent learning for the ANN since the car swerved to a side even though it was executing a forward movement command, which lead to a misinterpretation of what the best ground truth movement should be. This could be due to several factors. For example, the plastic wheels were wrapped with rubber bands which were not completely symmetrically placed, the course was on a carpeted area with small variations of texture patches, and the motors wore out over iterations. In a more controlled environment, these contributions to the error could have been minimized before each run with adjustments. Another note is that the ultrasonic sensors returned erratic values at some intervals of testing, albeit few. This also contributed to incorrect learning of ground truth since a close distance might have been mapped as a far distance and vice versa during the iterations that the sensor was incorrect. While this commonly happens with certain ping sensors, in our experiments this could possibly also happen when the program gets interrupted by user input before the distance calculation is fully completed by the microchips controlling the sensors. One way to handle this may be to only give commands to the car in a synchronous fashion. However, this would lead to a less genuine model for learning in a real time environment. Despite the issues with non-exact forward motion and occasional erratic sensor information, the system was very effective at learning the proper control signals.

It's also important to note that this framework is not a completely parallel system because the communication between the APM chips happens through a bus. Nevertheless, it is also worth noting that this is a *more* parallel system than a sequential software computation model or a single microprocessor loop implementation since the internal computations of the neurons still can happen in parallel. In the future, this would be an interesting comparative study to undertake in terms of noting performance and training time between the different models for a similar task.

VI. CONCLUSION

We have shown that a hardware Arduino ANN with single neurons on each chip can be used for practical applications such as learning control for obstacle avoidance and autonomous navigation of a test path. The training framework provided through the communication between the Arduino and a Processing application, plus the capability to easily switch from training to testing, proved to be effective in gathering information after each training iteration in regards to understanding the performance of the ANN. Future research will include the expansion of the system to test its robustness and ability to take on more difficult tasks.

VII. REFERENCES

[1] G. Parker and M. Khan, "Distributed Neural Network: Dynamic Learning via Backpropagation with Hardware Neurons using Arduino Chips," International Joint Conference on Neural Networks , 2016.

[2] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," Neurocomputing, 74(1-3), pp. 239-255, 2010, doi:10.1016/j.neucom.2010.03.021.

[3] F. M. Dias, A. Antunes, and A. M. Mota, "Artificial neural networks: A review of commercial hardware," Engineering Applications of Artificial Intelligence, IFAC, 17(8), pp. 945-952, 2004.

[4] K. Goser, "Implementation of artificial neural networks into hardware: Concepts and limitations," Mathematics and Computers in Simulation, 41(1-2), pp. 161-171, 1996.

[5] Y. Liao, "Neural networks in hardware: A survey," Davis, CA: Department of Computer Science, University of California, Davis, 2001.

[6] V. Kumar, S. Shekhar, and M. B. Amin, "A scalable parallel formulation of the backpropagation algorithm for hypercubes and related architectures," IEEE Transactions on Parallel and Distributed Systems, 4(10), pp. 1073-1090, 1994.

[7] A. R. Omondi and J. C. Rajapakse (Eds.), "FPGA implementations of neural networks," Springer, Netherlands, 2006.

[8] S. Sahin, Y. Becerikli, and S. Yazici, "Neural network implementation in hardware using FPGAs," NIP, Neural Information Processing, 4234(3), pp. 1105–1112, 2006.

[9] L. P. Maguire, T. M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin, "Challenges for large-scale implementations of spiking neural networks on FPGAs," Neurocomputing, 71(1-3), pp. 13-29, 2007, doi:10.1016/j.neucom.2006.11.029.

[10] J. Zhu, and P. Sutton, "FPGA implementations of neural networks–a survey of a decade of progress," In Field Programmable Logic and Application, Springer Berlin Heidelberg, pp. 1062-1066, 2003.

[11] P. Ienne, and G. Kuhn, "Digital systems for neural networks," Digital Signal Processing Technology, 57, pp. 311-45, 1995.

[12] A. Joubert, B. Belhadj, O. Temam, and R. Héliot, "Hardware spiking neurons design: Analog or digital?" International Joint Conference on Neural Networks, pp. 1-5, June 2012.

[13] D. A. Pomerleau, "ALVINN: An Autonomous Land Vehicle in a Neural Network," Report CMU-CS-89-107, Carnegie Mellon University, 1989.

[14] D. A. Pomerleau, "Rapidly adapting neural networks for autonomous navigation" , Advances in Neural Information Processing Systems, Morgan Kaufmann, San. Mateo, 3, pp. 429–435, 1991.

[15] J. B. S. Jonathan, A. Chandrasekhar, and T. Srinivasan, "Sentient Autonomous Vehicle Using Advanced Neural Net Technology," Department of Computer Science and Engg, Sri Venkateswara College of Engineering, Sriperumbudur, India, 2004.

[16] U. Farooq, M. Amar, M. U. Asad, A. Hanif, and S. O. Saleh, "Design and Implementation of Neural Network Based Controller for Mobile Robot Navigation in Unknown Environments," International Journal of Computer and Electrical Engineering, 6(2), 2014.

[17] M. Negnevitsky, Artificial intelligence: A guide to intelligent systems, 2nd ed., England: Pearson Education Limited, 2005, pp. 165-216.

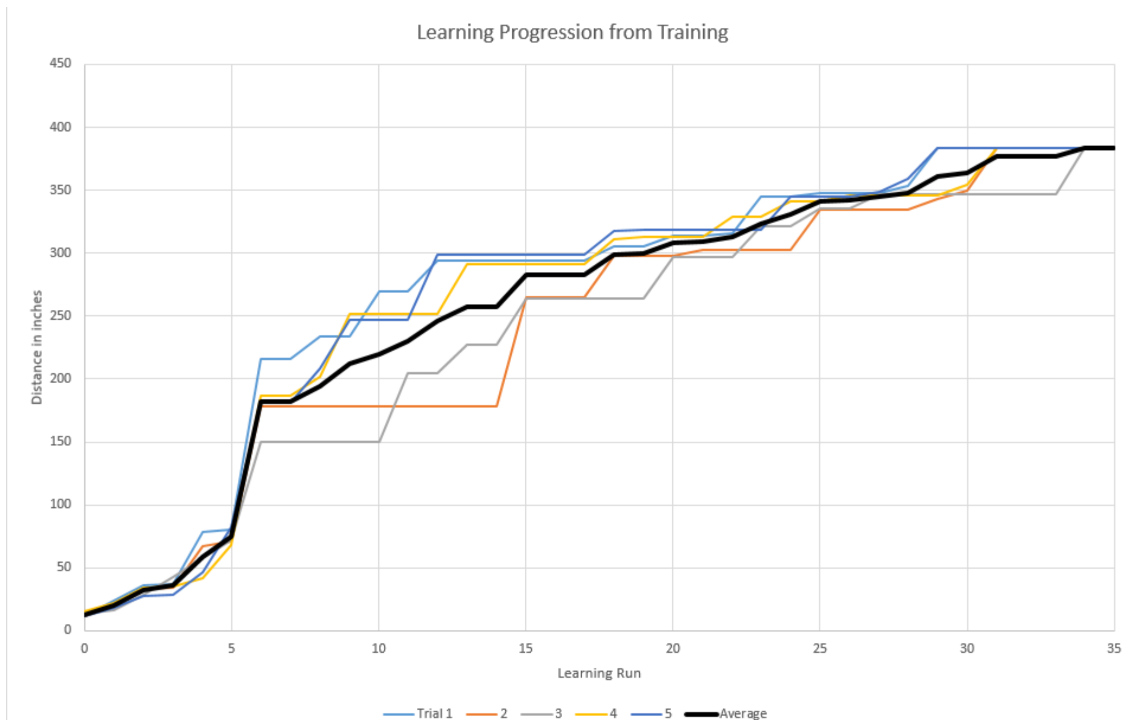

Figure 5. Distance the car covered in the path is mapped against the number of training iterations.