2017

Imitating the Brain: Autonomous Robots Harnessing the Power of Artificial Neural Networks

Mohammad Khan
Connecticut College, mkhan4@conncoll.edu

Imitating the Brain: Autonomous Robots Harnessing the Power of Artificial Neural Networks

Presented as an Honors Thesis for the

Connecticut College Computer Science Department

Mohammad Owais Khan, advised by Professor Gary Parker

Mohammad.Khan@conncoll.edu | Connecticut College | Class of 2017

Abstract:

Artificial Neural Networks (ANNs) imitate biological neural networks, which can have billions of neurons with trillions of interconnections. The first half of this paper focuses on fully-connected ANNs and hardware neural networks. The latter half of this paper focuses on Deep Learning, a strategy in Artificial Intelligence based on massive ANN architectures. We focus on Deep Convolutional Neural Networks, some of which are capable of differentiating between thousands of objects by self-learning from millions of images. We complete research in two areas of focus within the field of ANNs, and we provide ongoing work for and recommend two more areas of research in the future.

A hardware neural network was built from inexpensive microprocessors with the capability of not only solving logic operations but to also autonomously drive a model car without hitting any obstacles. We also presented a strategic approach to using the power of Deep Learning to abstract a control program for a mobile robot. The robot successfully learned to avoid obstacles based only on raw RGB images not only in its original area of training, but also in three other environments it had never been exposed to before.

Lastly, we contribute work to and recommended two applications of Deep Learning to a robotic platform. One application would be able to recognize and assist individuals based solely on facial recognition and scheduling. A system like this can serve as a personable, non-intrusive reminder system for patients with dementia or Alzheimer's. The other recommended application would allow the capability of identifying various objects in rooms and pin pointing them with coordinates based on a map.

Table of Contents

Acknowledgements:

I would like to thank my mother and father for giving me the opportunity to have a formal education even though they themselves were never able to graduate high school. I feel privileged to have been exposed to their genius as they are some of the most knowledgeable individuals that I know. Without my parents, I literally would not exist. This Honors Thesis is also dedicated to my brothers who worked long hours while attending high school when we first moved to this country. With their hard work, I was privileged enough to be able to focus solely on my studies, and I am now in college because of them. I also want to thank my sister-in-law for always reminding me the value of compassion through her actions, and my baby nephew Haseeb who has taught me both patience and the concept of genuine happiness.

I want to extend my gratitude to Professor Gary Parker for allowing me the opportunity to conduct research ever since my first year in college, and for providing me with valuable resources so that I could grow my passions. I want to thank Professor Ozgur Izmirli for his mentorship on the value of academic excellence and for motivating me to do research. I am thankful to Professor James Lee for answering my many questions about academic, personal, and professional life after college. I am also thankful to Professor Christine Chung for having shown me the value of discipline and effective teaching through her actions. I am very grateful for Professor Leslie Brown's generosity, kindness, availability, mentorship, and family-like communication that kept me going time after time.

Lastly, I am incredibly humbled by the ongoing support of my best friend, the always inspiring, compassionate, and brilliant Lamiya Khandaker.

Introduction:

An Artificial Neural Network (ANN) seeks to imitate a biological neural network, which can have billions of neurons with trillions of interconnections. While modeling the sheer size of the neural networks that exist within our bodies might be a long shot for the moment being, research of recent from both academia and the industry has shown an incredible amount of progress towards developing highly intelligent programs. Modern shows like *Humans, Westworld, and Black Mirror* and movies like *Ex Machina* have established a niche in the market by creating hysteria around the concept of advanced Artificial Intelligence and the usage of ANNs. While the worry about the usage of such technology is valid, it is important to understand how far we have come in terms of progress. With the right context, we can understand the implications of such systems. There are strides being made to move research towards the path of using Artificial Intelligence for good [38]. Use cases such as building models to understand global warming and systems that can detect cancer and suggest treatment are certainly promising green areas. It is inspiring to see an incredible amount of funding headed towards research that seeks to follow this venture.

The field of ANNs has been surging in recent years. We interact with ANNs on a daily basis whether we are aware of it or not. Apple's iPhone Siri is powered by ANNs, and actually improves through training [39]. Companies like Facebook are working on using large ANNs to allow users to search through and find specific images using text [40]. There are two particular areas of interest within ANNs for this Honors Thesis: hardware ANNs and an advanced ANN topic called Deep Learning.

The first half of this paper will discuss advances in hardware neural networks. Hardware ANNs are attractive because they seem to promise development outside the restraints of the parallel nature of software within computers. Biological neurons are inherently, and massively, parallel systems. By building microprocessors that can simulate, in tandem, this parallel nature, we can work towards building more efficient models. In this paper, we present a unique, accessible approach to building a hardware neural network that is not only capable of learning logic operations, but also capable of learning how to drive a model car by itself without hitting obstacles. The network first learns through human supervision and by "seeing" through ultrasonic sensors, and then it becomes capable of driving itself with a very low error rate. We develop a distributed neural network such that it is built up by single microprocessors, each of which imitates a neuron. Each processor only costs about \$3 to \$10 and we use only 3 neurons for our research as a presentation of a proof of concept.

The field of Deep Learning consists of algorithms that learn using massive ANN architectures. Most Deep Learning models are built with the ability to process images in mind. Some of these architectures are capable of outperforming humans in tasks like classifying objects – which simply means differentiating one object from other objects (dog vs. wolf, etc.). Figure 1 below demonstrates the power of such algorithms. This network can identify, with levels of certainty, which objects exist within an image. It can identify specific species of animals, which is a skill that even many humans struggle with. In this paper, we present an application of Deep Learning to the concept of autonomous driving for a robot within a tight classroom/laboratory setting based strictly on images. Not only is the robot able to successfully and autonomously drive without hitting obstacles within the original room, but it is also able to avoid obstacles in other environments such as: the hallways of a building, a completely different room with a different set up (different design of chairs, wall colors, etc.), and a blocked off carpeted area with wooden blocks it has never seen before. Lastly, the robot also abstracted the concept of being able to navigate around human legs. This work will be recommended for submission this upcoming July of 2017 into the IEEE International Conference on Intelligent Robots and Systems (IROS) in Vancouver, Canada.

Apart from the main areas of focus above, we also started progress on two additional application of Deep Learning. One application is to assist hospital patients or residents of an old age home by the way of face recognition. The robot would be able to drive around, identify individuals, and assist them based on the current time and the individual's schedule. Groups of these robots can serve as escorts and as personable reminder system. Patients or residents that do have access to smart devices or smart phones, and patients or residents that suffer from the symptoms of Alzheimer's and/or dementia may be able to benefit from a system like this. This research is still not mature, and more research needs to be done insofar as the specific needs of patients and residents within those environments. The other area of focus suggests and attempts to use Deep Learning in order to pinpoint the location of specific objects or items within a room. The idea is that the robot would be able to drive around, identify objects, and specify the coordinates of those objects in some room. One would be able to use a system like this to see if a lost item has been found. On a foundational level, many objects in a room can be mapped with specific coordinates in a database. A large dataset can be built to further other types of research on object placement, human-object interaction, and the like.

Note that it is recommended to start reading this paper with the "Artificial Neural Networks" section. After that section, the two main research sections can be accessible to the reader in either order, although some of the advanced concepts in the section "A Focus on Deep

Learning" may build off of concepts established in the section "A Focus on Hardware Neural Networks".

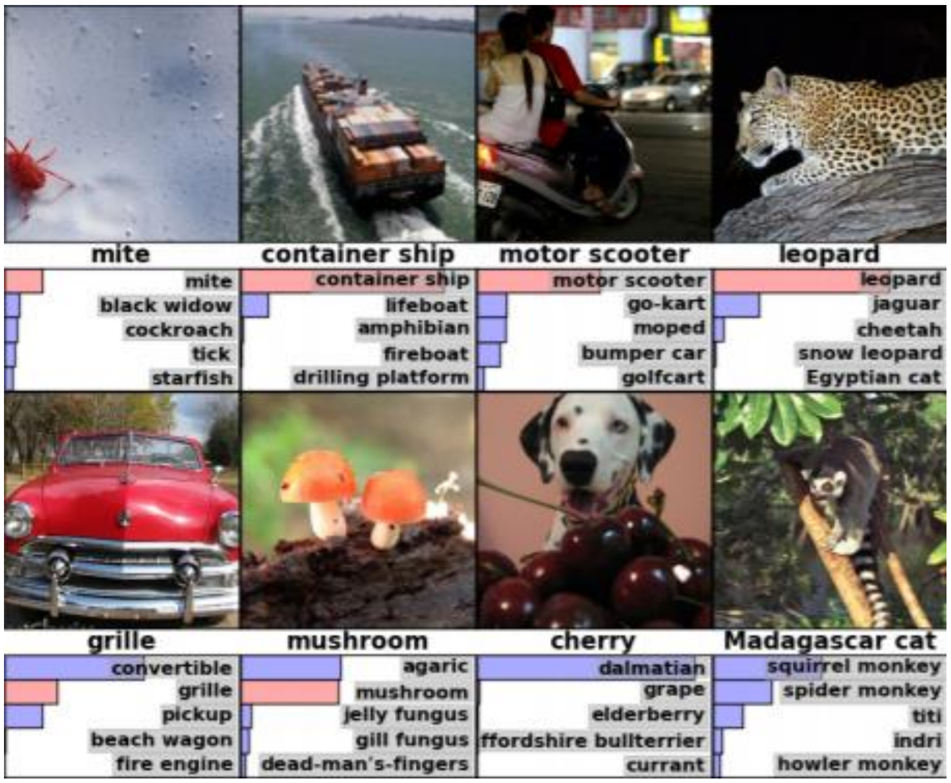


Figure 1. Objects in images being predicted by a deep neural network along with certainty values for 5 different objects per image.

Source: This is from a novel Deep Learning algorithm paper by Krizhevsky et al. (*ImageNet Classification with Deep Convolutional Neural Networks).* Please refer to endnotes.

Artificial Neural Networks:

Artificial Neural Networks (ANNs) are interconnections of single neurons linked together by weighted associations through which information is received, transformed, and transferred. The way neurons are modeled, or mimicked, in the field of Artificial Intelligence is through defining them as objects, also called nodes, which receive a set of inputs with corresponding weights and produce only a single output. Nodes are also referred to as perceptrons. For this reason, throughout the text, neurons, perceptrons, and nodes will be used synonymously. The parallel between the artificial neuron and the biological neuron is demonstrated below in Figure 2 and Figure 3. Dendrites effectively become the inputs (x_1, $x_{2, \ldots}$ x_n) to our artificial neuron, and the axons become the output signals (y). Within the cell body we may transform these inputs with some sort of mathematical function, which effectively emulates the function of the nucleus.

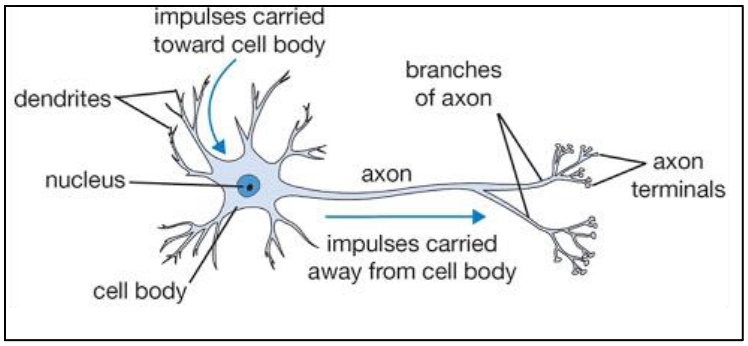

impulses carried toward cell body
branches of axon
dendrites
axon terminals
nucleus
axon
impulses carried away from cell body
cell body

Figure 2. Visual representation of a biological neuron

Source: https://wpclipart.com/medical/anatomy/nervous_system/neuron/neuron.png.html

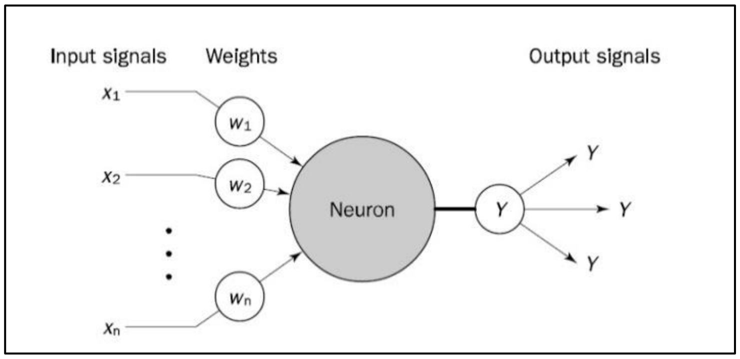


Figure 3. Visual representation of an artificial neuron

Source: *Artificial Intelligence: A Guide to Intelligent Systems* by Negnevitsky, page 167 [1]

Networks are modeled such that these neurons are stacked in multiple layers where different layers of neurons serve different functions. The output of one neuron becomes the input for neurons in a subsequent layer. For example, Figure 4 includes an input layer (which consists of no neurons – it just provides input), a hidden layer with two neurons, and an output layer with one neuron. In fully connected models, each neuron in the previous layer is connected to each neuron in the next layer. Later, through exploring Deep Learning methodologies we will cover how other methods apart from fully connected networks might be preferred. A threshold value (θ) symbolically represents the activation potential of a biological neuron. This threshold can allude to how a neuron fires after reaching some saturation point. A weighted sum of all the inputs is compared to this value to determine an output. To simplify setting up systems of equations, we treat the threshold as a constant input value of -1 and give it a corresponding weight. A linear combination, or weighted sum, (denoted as *X*) of the vector of all inputs and the corresponding vector of all weights is fed into an activation function to determine an output (*Y*). The production of an output is called activation.

For this paper, we employ the sigmoid activation function as such:

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

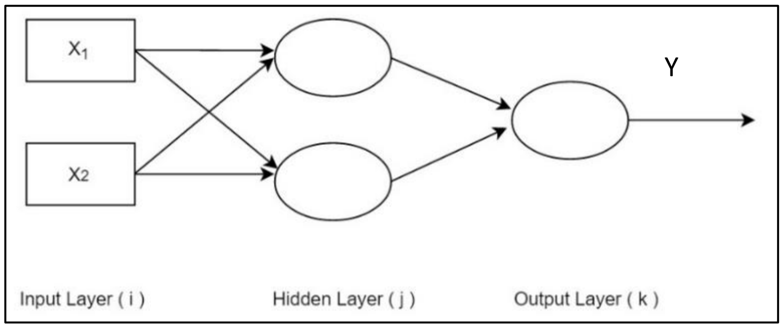


Figure 4. Visual representation of a multiple layered ANN with two inputs, two hidden layer neurons, and an output layer neuron.

What Does Learning Mean in Artificial Intelligence?

The term "learning" in the field of Artificial Intelligence and Machine Learning refers to how computers can develop and learn programs, models, and/or functions without being explicitly programmed. Generally, a dataset is given to a learning algorithm and based on that dataset knowledge of certain patterns is acquired in order to come to more refined conclusions. For example, a dataset can consist of images of fruits that are labeled with their corresponding names. Over time, the algorithm analyzes each image and recognizes specific features that differentiate one fruit from another. Pointedly, a feature may be the color of the fruit, the texture, or even the shape. Once these features are learned, the algorithm can pick up on patterns it has seen before while it is exposed to new images it has never seen. This is the advantage of

Machine Learning over general purpose programming since intelligent algorithms can set and define rules that were not explicitly programmed. It is much harder to come up with all the edge cases for a specific domain versus having an algorithm learn them. For example, this makes sense for complex situations like real world autonomous driving. One would need to program decision making for every single case that the car “sees” – a couple of cases may be “there is an object 5 feet away on the left hand side, turn right” or “there is an object 2 inches away, stop immediately”. There may be thousands of similar situations to program by hand.

How Do Artificial Neural Networks Learn?

The standard learning mechanism for feed-forward ANNs consists of two stages, forward propagation and “backpropagation”. Learning happens when these stages are recurrently repeated with a set of desired input/output pairs. In simpler terms, classification labels like “apple” or “orange” are known as the desired output or ground truth. This is opposed to the concept of an actual output, which is an output that the network guesses. The learning process starts by initially setting all of the weights in each neuron to random values within a small range. The input layer provides the input to each neuron in the hidden layer. The hidden layer neurons produce an activation, which is passed to the next layer. Activations continue to be outputted in each subsequent layer until the output layer neurons are reached. An actual output is produced after activation at the output layer, and forward propagation ends. This value is compared to the desired output corresponding to the current input and an error value is produced as the difference between the two values. Using this error, the process of backpropagation begins as described below. Error gradients are used to adjust weights in the case of multi-layered ANNs because hidden layers do not have a clear desired output value, and thus cannot have an error by the definition above. 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 [1]. The backpropagation algorithm was utilized for the learning portion of this ANN. The algorithm follows as such:

1) **The error gradient (δ) for the neuron in the output layer (k) at the current iteration (t) is calculated:**

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

Where Y_k is the actual output at the output layer and

e_k is the error at the output layer neuron such that:

$$e_k(t) = Y_{d\_k}(t) - Y_k(t)$$

Where Y_{d_k} is the 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) using this error gradient:**

$$W_{j\_k}(t+1) = W_{j\_k}(t) + [\, \alpha \times Y_j(t) \times \delta_k(t) \,]$$

Where Y_j is the actual output at the hidden layer and α is the constant learning rate

3) **The error gradient (δ) for the neurons in the hidden layer (j) is calculated:**

$$\delta_j(t) = Y_j(t) \times [\, 1 - Y_j(t) \,] \times [\, \delta_k(t) \times W_{j\_k}(t) \,]$$

This is the case when there is a single output neuron.

4) **The weights (W_{i_j}) between the input layer (i) and hidden layer (j) are updated using this error gradient:**

$$W_{i\_j}(t+1) = W_{i\_j}(t) + [\ \alpha \times x_i(t) \times \delta_j(t)\ ]$$

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

5) **Once the weights are updated, a new iteration runs through forward propagation and backpropagation. This process loops continuously.**

A Focus on Hardware Neural Networks:

Most ANNs are implemented solely in software simulations, and those implemented in hardware usually have the whole ANN loaded on one microchip. This research takes a step towards a new ANN implemented by connecting a number of microchips such that each microchip represents a single neuron. While the commercial market does have a host of hardware ANNs, no general use, unspecialized, and inexpensive model exists such that has a clear one microchip to one neuron representation. Generally, extra modules and processing parts are added to have the microchip function as a single neuron and more commonly multiple neurons are implemented on one microchip. Under-the-hood, these representations also generally require a good grasp of electrical engineering to fully understand how the inputs and outputs are mapped as they use voltage, current, and similar elements to model data. In order to manipulate these elements, specialized and expensive parts are often needed. Moreover, many of the architectures are built for specific tasks, and they are not easily incorporated into bigger systems, and even fewer have learning capabilities. This paper presents a more accessible approach for creating general hardware ANNs capable of learning.

In this research we develop a hardware ANN utilizing Arduino Pro Mini (APM) microchips, which are popular, low-priced microcontrollers, coupled with open-source software, to function as neurons that are able to learn using the backpropagation algorithm. In this implementation, each of the APM microchips act as single neurons with learning capabilities, which is a step towards a more genuine imitation of a biological neural network. The architecture of the ANN that we developed for this initial trial is such that one APM microchip handles the input, two APM microchips are each neurons in the hidden layer taking input from the previous APM microchip, and one APM microchip is a neuron in the output layer. The communication between the microchips is supported by the I2C (Inter-Integrated Circuit) capability of Arduino which allows a Sender-Receiver (also referred to as Master-Slave; this paper introduces the former terminology) connection between the output microchip, the hidden layer microchips, and the input microchip. This works into the theme of a more genuine model because the output neuron fires only when it receives input from the hidden layer neurons. This design was implemented on a breadboard to show the dynamic nature of the system whereby microchips/neurons can be moved to grow or shrink the ANN. The finished network has the capability of learning four different logic operations: AND, OR, XOR, and XNOR. Learning happens on the hardware APMs themselves without a data connection to a computer. The key to the implementation was an understanding of the intricacies of transferring data across I2C and modularizing the overall ANN across multiple Arduinos. A protocol for transferring the information from microchip to microchip was developed to allow backpropagation in the learning phase.

Later, this hardware Arduino ANN was used to learn the real world application of control for autonomous car navigation. A small car was made to incorporate 3 ultrasonic ping sensors as input and two full-rotation Servos to facilitate output. 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 8 x 8 foot colony space. Transitions between testing and training were made to incrementally record the progress of the car's learning. The results showed 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.

Related Works in Hardware Neural Networks:

Most commonly ANNs are implemented as software and trained on computers. This technique is inherently non-parallel due to the sequential von-Neumann architecture of computers. In contrast, the past two decades have shown strides being made in the development of dedicated hardware to develop faster and more genuine networks that are processed in parallel [2]. However, there is much room for improvement. Dias, Antunes, and Mota note in a review of commercial ANN hardware that much of this hardware is specialized and expensive in development time and resources, and not much is known about its commercial implementations [3]. Goser notes in a paper on limitations of hardware ANN that one problem in developing hardware models of ANNs is the need for dedicated, complex wiring due to specialized hardware [4]. It is important to consider how the architecture of an ANN is represented in hardware. Generally, multiple neurons are implemented on one chip and often the full ANN is put on one chip. Liao notes in a survey of Neural Networks hardware that almost all hardware neuron designs are such that they include an activation block, weights block, transfer function block, and other processing elements [5]. The activation block is always on the chip, but the other blocks need not be. In other words, much of these implementations do not focus on building a hardware neuron in a single chip. There have also been highly specialized products that explore learning through implementation of the backpropagation algorithm on several different hardware structures [6]. These products, however, deal more with hardware manipulation in parallel computers than representation on the microcontroller level.

One method for creating hardware ANNs is the use of Field Programmable Gate Arrays (FPGAs). FPGAs are integrated circuits that are physically configurable and serve as programmable logic. Significant work has been done in using FPGAs for ANN development [7, 9, 10]. Sahin, Becerikli and Yazici implemented a multiple layer ANN such that an FPGA was configured to link neurons together with multipliers and adders were representative of connections between neurons [8]. This framework allowed for manipulation of the ANN architecture by having a dynamic number of layers and neurons. The weights were learned beforehand and mapped onto the FPGA. Although no clear learning method was explored on the hardware itself, this work took an accessible product commonly used for prototyping to develop a hardware ANN. It is important to note that even though the ANN architecture can be changed in these frameworks, it does require physical configuration and rededication of parts within the hardware itself.

A general theme with development of hardware ANNs has been an architectural representation such that multiple neurons exist on one chip. Ienne and Kuhn give an analysis of multiple commercial chips [11]. For example, the Philips' Lneuro-1 consists of 16 neurons while the Philips' Lneuro-2.3 consists of 12 neurons. The Ricoh RN-200 also consists of a 16 neurons in a multiple layered ANN and includes backpropagation learning. Some implementations add extra processing parts to a chip to make it a faster system. For example, Ienne and Kuhn also describe the Intel’s Ni1000 which consists of three parts where the microcontroller is combined with a classifier and an external interface that does conversion calculation to decrease the load on the chip itself.

Other specialized designs have been considered, and many intricate methods for modeling neurons have been explored. Joubert, Belhadj, Temam, and Héliot give an analysis of the Leaky Integrate-and-Fire neuron which although is a single neuron representation, has a specific architecture split among three main parts: the synapse, the neuron core, and the comparator [12]. A number of these implementations are based on electrical properties such as current, capacitance, voltage, and similar elements. Although this may be a more genuine representation of a biological neuron it does take a good amount of electrical engineering expertise to implement and reproduce, and the parts are specialized and expensive.

In summary, there has been much research in the development of novel hardware ANNs that are efficient and fast. However, most of these hardware implementations are built with specialized parts, the resources are expensive, and there does not exist a clear general use, unspecialized and inexpensive “one neuron to one chip” implementation. Hence, the ultimate goal of the research reported in this paper is to provide an alternative, to develop a hardware ANN system where each neuron is implemented on a single inexpensive, readily available chip. In addition, each neuron/chip will have learning capability (initially through backpropagation) and the implementation will be accessible to typical users. In this research we used the APM microchip because it is inexpensive, widely available, small enough for quick prototyping of ANN architectures down to the level of the neuron, and allows for a communication framework for learning through backpropagation.

Related Works in Car Navigation Using Artificial 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 the 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 corresponding algorithms implemented within the system. This low processing overhead significantly improves time spent for the ANN in learning [15]. 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 paper. 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.

Implementation of Hardware Neural Network:

The Arduino Pro Mini (APM) was selected for this implementation of a hardware ANN because it is widely available, small enough for quick prototyping, and inexpensive (the pricing for the chip runs $3 to $10 depending on distributer). The 5V and 16MHz model was chosen for this research. The APM has 14 digital input/output pins. A 6 pin header connects via an FTDI cable to the computer to upload programs or provide USB power. The size of the microchip (.7'' x 1.3'') allows linkage of many microchips together without taking up much space.

The Inter-Integrated Circuit (I2C) bus, which is a built in part of the APM, was used to control communication of data between neurons. Each neuron was given its own memory address through assignments within its program. The I2C works such that a single bus allows communication via a Serial Clock Line and a Serial Data Line through Sender-Receiver relationships between APM microchips. The Sender microchip initiates the clock and demands communication from the Receiver microchips. The Receiver microchips synchronize with and respond to Sender commands. On the APM, pin A4 provides access to the Data Line and pin A5 provides access to the Clock Line. Thus, these two lines come together in simplified wiring along one simple I2C bus that connects all APMs within the hardware ANN.

In order to embed an ANN framework within the APMs using the I2C bus, the Arduino Wire library was needed. This library comes with the Arduino Integrated Development Environment and is preset with functions that allow communication between microchips connected together with the I2C bus. The two main commands used were read() and write() which allow data flow to happen in two directions. Other commands were also used, such as:

requestFrom() which allows a microchip to ask for data from another microchip and available() which is a Boolean functions that returns true if data is available from a microchip.

I2C Data Transfer:

The transfer of data in the Arduino I2C is limited to bytes or characters. This means that a double precision value - which consists of 4 bytes in Arduino - cannot easily be transferred over wire. Precision is vital for the weights learning portion of the Neural Network. We worked around this limitation by using a Union - a data type in the C programming language that allows storing of different data types in the same memory location:

```
union T { byte b[4]; double d; }  T;
```

Here the same memory location refers to a double and a byte. Hence, a double can be encoded as an array of bytes, packaged and sent over Wire to an APM neuron address:

```
int dataSize = 4;
byte packet[dataSize];
for (byte i = 0; i < dataSize; i++)
      packet[i] = T.b[i];
Wire.write(packet, dataSize);
```

The receiving end utilizes another Union that reverses this masking to obtain the double precision and store it into a variable:

```
for (byte i = 0; i < 4; i++)
        T.b[i] = Wire.read();
double value = T.d;
```

Configuration of Individual Microchips:

Each microchip was configured with a program written in the Arduino language which is based on C/C++ functions. A single neuron class was developed for a hidden layer neuron and an output layer neuron. Similarly, a class was developed for the input element. Within the context of the I2C bus framework, each APM program included a memory address to be used while probing and communicating through the bus. This is very beneficial to making the hardware ANN a dynamic system whereby the architecture can be changed easily with minimal adjustment. Weights and inputs are each stored in different arrays of type double within each hidden and output neuron. The learning rate is set at .35 for all neurons - this can be easily adjusted based on observation of learning performance. Below, software components of each type of microchip are outlined.

a) **Input Element Microchip (Receiver)**

A desired output set is defined as a 2D array matrix such that each element is a chain of desired outputs to the corresponding logic operation input values, ordered as XOR, OR, AND, and XNOR:

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

Similarly a 2D array matrix defines the training set of inputs:

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

A requestEvent() function waits continuously for a request from the output APM Sender neuron for a desired output value. Then, this value is packaged via the I2C process above and sent over wire. Simultaneously, the next input pair is chosen to be sent to each hidden neuron for the next forward propagation iteration.

b) Hidden Neuron Microchip (Receiver)

There are two main functions in the Receiver hidden neuron class: requestEvent() and receiveEvent(). Function requestEvent() waits continuously for a request from the output APM Sender neuron, and upon triggering the function, an output value after activation is packaged and sent over wire as part of the forward propagation phase. Function receiveEvent() is triggered upon the arrival of data from the output neuron in the backpropagation phase. If the correct amount of data, 4 bytes, arrives then this means that the output neuron sent an error gradient value back for learning. Using this value, the weights are updated.

c) Output Neuron Microchip (Sender)

There are two main functions that allow data to flow into the Sender output neuron: readHidden() and readOutput(). Function readHidden() requests a neuron output from each hidden layer neuron via I2C and reads these as inputs for the output neuron itself. Once the inputs are received, an actual output is produced for forward propagation. Function readOutput() requests a desired output value for the current logic operation row from the input element via I2C and stores this value to be compared to the actual output. This comparison allows the calculation of an error. Using this new value backpropagation is initiated. The weights are updated, and an error gradient is calculated and sent back to each hidden layer neuron via I2C.

Neural Network Circuit Design:

For this proof-of-concept there are 4 APM microchips. As shown on Figure 5, there is one input element, two hidden layer neurons, and one output layer neuron. This ANN architecture mirrors Figure 4 except a single APM input element provides both inputs. The communication functions such that the output neuron in the ANN is the labeled Sender APM and the hidden neurons and input layer in the ANN are the labeled Receiver APMs. The I2C bus connects the A4 and A5 pins of all the Receiver APMs to the A4 and A5 pins of the Sender APM. Two wires from each APM, one for Data and one for Clock, allow this connection. Meanwhile, the APM input element provides constant logic operation row input through two wires (can be established via any input/output Arduino pins) to the two respective pins on the hidden APM neurons as values of 0 or 1 - respectively LOW (0V) and HIGH (5V). Two physical switches are connected to the input layer element to allow the user to dynamically change between logic operations to be learned; a switch state of 00 learns XOR, 01 learns AND, 10 learns OR, and 11 learns XOR. Moreover, the current desired output for the corresponding logic operation row at the output neuron is obtained through an I2C request to the input layer element. Power is provided through the VCC pin for each APM, and all APMs are grounded through the GND pin.

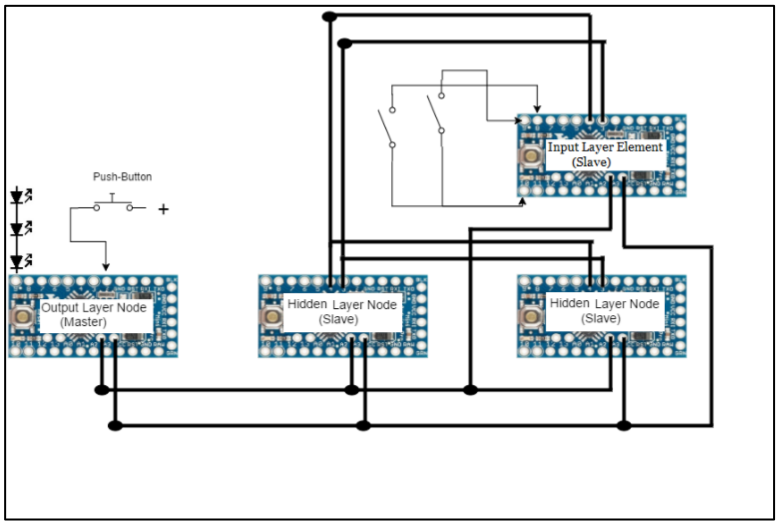

Push-Button
+
Input Layer Element (Slave)
Output Layer Node (Master)
Hidden Layer Node (Slave)
Hidden Layer Node (Slave)

Figure 5. A schematic of the APM circuit is shown. Each neuron is labeled as a node. The two switches are attached to the input element. A pushbutton to toggle between learning and testing, and 3 LEDs to visualize output are attached to the output layer neuron.

For presentation purposes, 2 LEDs are added to two input layer element output pins to help visualize the current inputs (OFF and ON correlate to an input of 0 and 1 respectively). An array of 3 LEDs is provided at the output neuron to visualize the current output. Thus, the spectrum of LED states provide a visualization of the range of outputs (see Table 1). A button is also connected to a pin on the output APM neuron to provide a means to switch between learning and testing. Upon testing (button is pressed down), the backpropagation stops and there is a 1.5 second delay at each APM data transfer to slow down changes between the different inputs to allow the visualization of the LED brightness states. Refer to Figure 6 for visual of finished prototype on breadboard.

TABLE I. THE NUMBER OF LEDS ILLUMINATED SHOWS THE RANGE OF THE OUTPUT. IN THE THREE DIGIT NUMBER, 0 AND 1 CORRESPOND TO OFF AND ON, RESPECTIVELY, FOR EACH OF THE 3 DIODS.

OUTPUT LED STATE	OUTPUT RANGE
000	0 - .3
001	.31 - .5
011	.51 - .7
111	.71 - 1

Algorithm:

The following algorithm works with the forward propagation phase and the backpropagation phase to allow for data transfer between APM chips utilizing the I2C Sender-Receiver framework:

1) Upon being powered, all APM neurons have a threshold input variable set to -1 within their programs.

2) Initially, random weights between -1 and 1 are set in arrays within each hidden and output neuron program.

3) Initiation of the forward propagation process starts at the output (Sender) neuron. It requests 4 bytes from each hidden node. The hidden neurons each forward propagate using inputs from the input APM element and their corresponding weights, and send an output to the Sender neuron upon request via I2C. The output neuron takes these values as inputs.

4) The Sender neuron sends a request to the input APM element to send a 4 byte desired output double value.

5) The input APM element sends the desired output to the Sender neuron via I2C.

6) Activation happens at the output layer, and the Sender neuron produces an actual output.

7) The backpropagation algorithm starts here:

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

9) This hidden layer error gradient is sent to each hidden layer neuron.

10) The weights of the output layer are updated using the output layer error gradient.

11) The weights of the hidden layer are updated using the hidden layer error gradient.

12) Loop back to step 3. This time, the input element changes the current input/desired output pair to the next row in the current logical operation.

When the testing button is pressed, the backpropagation steps 7-11 are skipped and the data flow is slowed down by 1.5 seconds to visually show via LEDs the corresponding output to the current input.

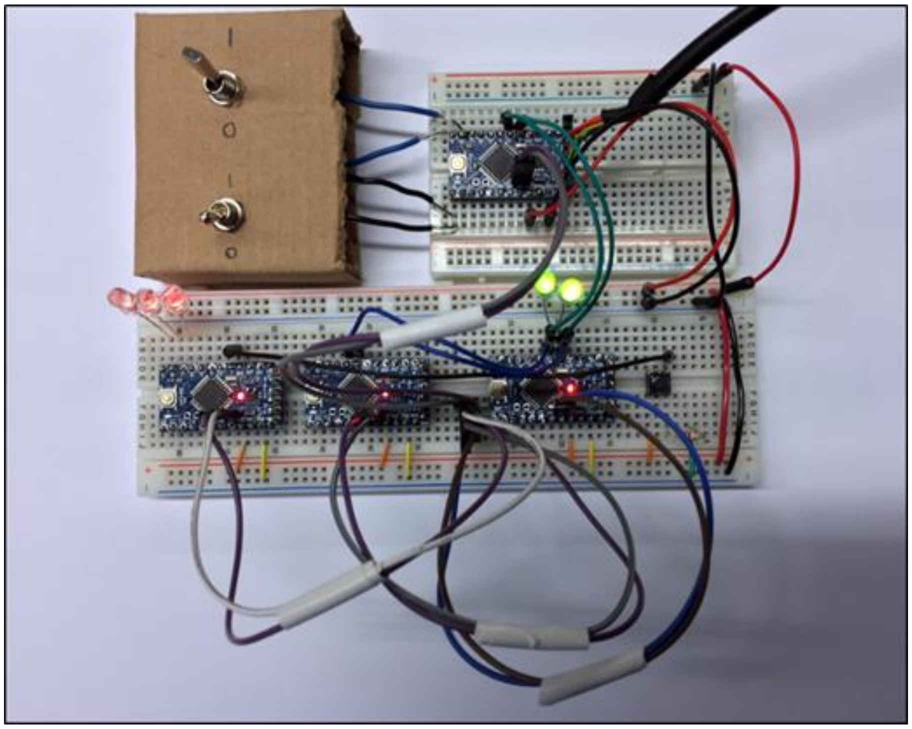


Figure 6. This is a photo of the implemented circuit design (compare to Figure 5). Here the green LEDs show the current input, and the red LEDs show the corresponding output.

Results for Preliminary Hardware Neural Network Design:

The hardware ANN successfully learned the logic operations OR, AND, XOR, and XNOR. Refer to Table 2 for the results of 5 trials. The times and average times it takes to learn each logic operation within 49% and 30% error are shown. These benchmarks mean, for example, that if a desired output is 1 and an error of 49% is reached, then the actual output of the ANN would be .51, and if the desired output is 0 and an error of 49% is reached then the actual output of the ANN would be .49. This error benchmark was selected because at this point the ANN would always yield the correct output as long as the actual output was rounded to the nearest integer (0 or 1). A similar test was done at 30% error. At this point all outputs definitely correspond to the desired output with the actual output more fine-tuned. Recorded trials were run 5 times from a random weights start, and time was taken when the error was reached for the two benchmarks. In context of the 30% benchmark, XOR took the least time to learn while AND took the most, respectively around 43 seconds and 2 minutes, 36 seconds. With further training all logical operations were learned with 2% error or less for each corresponding input/output. The Neural Network was also able to learn dynamically upon switching to other logic operations. This means, for example, that after having learned a logical operation such as XOR to within a low percentage error, the switches were used to change to another logical operation such as AND. After the switch, the AND logical operation was also learned within a low percentage error. Every combination of change between logic operations successfully worked to learn the logical operation changed to within 30% error. Tests of changing the architecture - adding more hidden layer nodes - were also done with positive results.

TABLE II. FIVE TRIALS FROM RANDOM START WEIGHTS WERE RUN. THE TIMES BELOW ARE ROUNDED IN SECONDS FOR THE CORRESPONDING ERROR BENCHMARKS.

	Trial 1		**Trial 2**		**Trial 3**		**Trial 4**		**Trial 5**		**Average**	
Logical Operation	49%	30%	49%	30%	49%	30%	49%	30%	49%	30%	**49%**	**30%**
OR	1:23	2:17	1:24	2:16	1:23	2:16	1:23	2:16	1:24	2:16	**1:23**	**2:16**
AND	1:56	2:37	1:53	2:35	1:54	2:36	1:53	2:35	1:53	2:35	**1:54**	**2:36**
XOR	0:02	0:44	0:02	0:43	0:02	0:43	0:02	0:43	0:02	0:43	**0:02**	**0:43**
XNOR	13:45	1:24	15:16	1:27	14:93	1:25	16:38	1:27	16:78	1:27	**0:15**	**1:26**

Autonomous Car Driving Using the Hardware Neural Network Design:

Car navigation was seen as a good gateway to testing the quality of the network. A car was designed as such in the car design section below. A navigation space and task were developed along with an algorithm for the learning and testing process. Results supported the usefulness of the hardware APM Neural Network.

Car Design:

A foot long wooden chassis was cut long enough to hold the hardware Arduino ANN (Figure 7). Three Parallax Ultrasonic Ping sensors were placed on a cardboard frame and attached to the edge of the chassis 40° apart (Figure 7). 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 input from the user in the queue of the HC-05 Bluetooth module. 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.

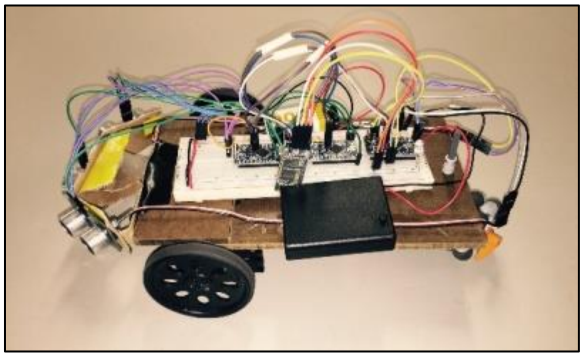
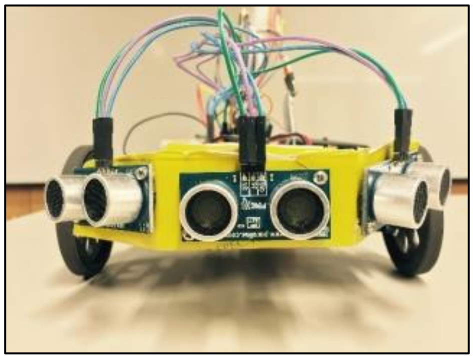



Figure 7. The car constructed for this research along with its various elements and a view of the placement of the ultrasonic sensors on the front.

Adjusted Hardware Neural Network Design:

The hardware network was adjusted such that it now takes 3 inputs corresponding to the ultrasonic sensors into each neuron of the hidden layer. Furthermore, the output of the network determines the direction that the car turns. 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 the 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. A structure of the network along with a visual of the APM model can be seen in Figures 9 and 10, respectively.

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 8 from both directions. A full training and testing iteration is counted as the summed distance of both paths taken (384 inches).

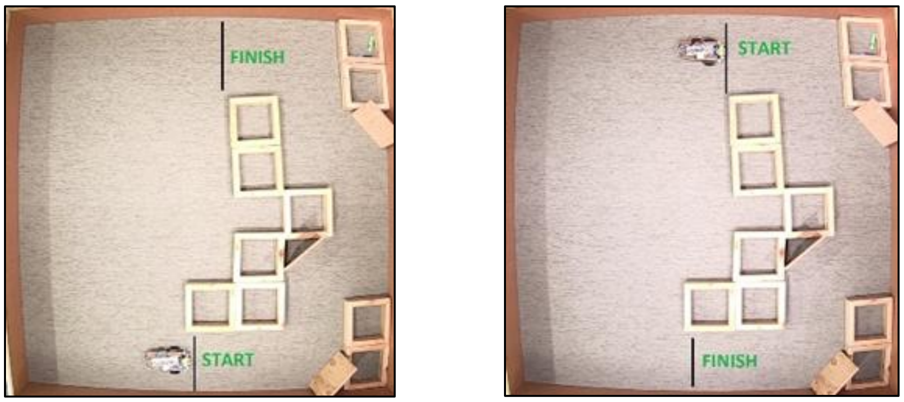


Figure 8. The car learns to complete the path from both directions.

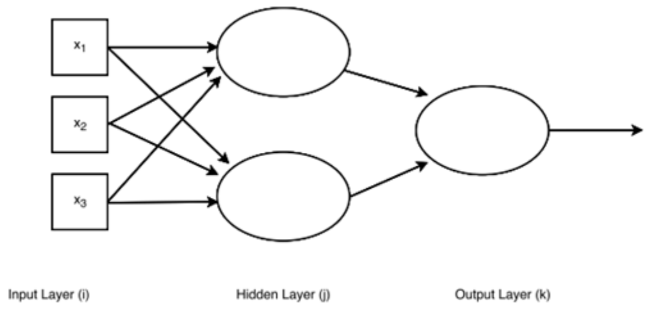


Figure 9. Visual representation of hardware ANN used for autonomous car task.

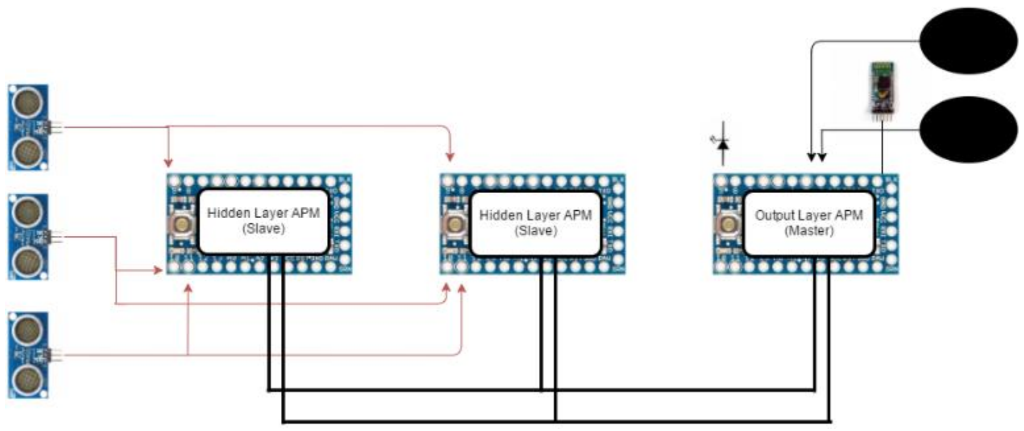


Figure 10. A schematic of the APM circuit is shown. Each neuron is labeled. Input comes in from 3 ultrasonic sensors to each hidden layer neuron. An LED is attached to the output layer neuron to show testing mode in action. A Bluetooth module (rectangle) and two full rotation servos (black ovals) are also attached to the output layer neuron.

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 8.

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 8. 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.

Results for Autonomous Driving Experiment:

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 only 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 11). 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.

Conclusions for Autonomous Driving:

The 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.

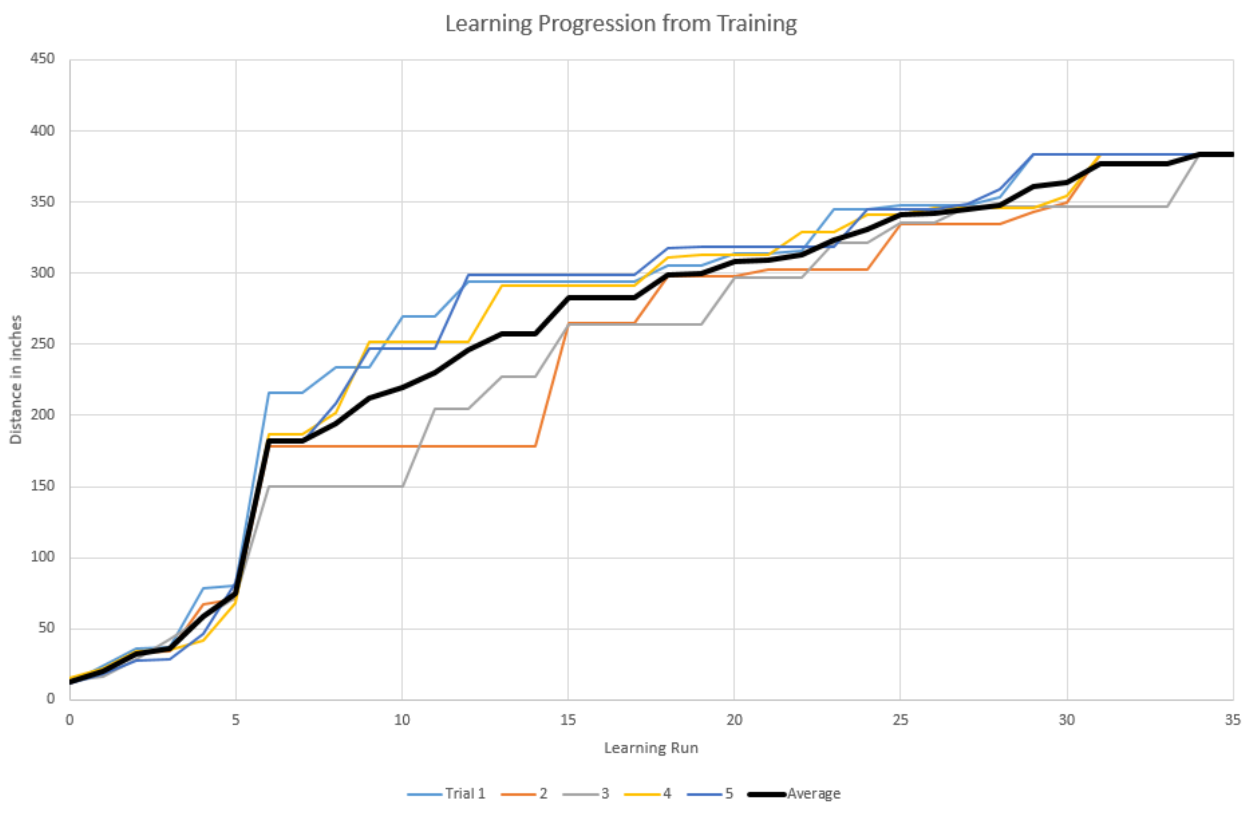


Figure 11. Distance the car covered in the path is mapped against the number of training iterations.

A Focus on Deep Learning:

Introduction to Deep Learning:

In the earlier introduction to ANNs, we covered the concept of fully connected layers. To reiterate this approach we can simply state that every single neuron in the previous layer of a network is connected to every single neuron in the next layer. This mode of architecture is inherently limiting and serves as one of the main motivations for Deep Learning. To further highlight the specific limitation we can consider an anecdote. Take a 100 x 100 pixel image. Suppose that we have 3 color channels. Immediately, we have 100 x 100 x 3 or 30,000 input vectors heading our network’s way. We choose 3 color channels as our input field, but for more complex problems an image can be paired with many descriptive features. Suppose we choose 100 other features along with our picture. We would then have 30,000 x 100 or 3 million input vectors for each neuron in our first layer! Deep Learning directly addresses this limitation.

Convolution:

Instead of sending all input values from layer to layer, deep networks are designed to take regions or subsamples of inputs. Pointedly, for images this means that instead of sending all pixels in the entire image as inputs, different neurons will only take regions of the image as inputs. Essentially, full connectivity is reduced to local connectivity. Figure 12 visualizes this concept. We take an image of the Shain library courtyard and extract local regions of depth 3 for the color channels along with their respective pixel values and input them into a neuron.

Supposing that our local receptive fields are of size 5 x 5, this neuron takes in an input of dimensions 5 x 5 x 3 for that particular portion of the 3 color channel image.

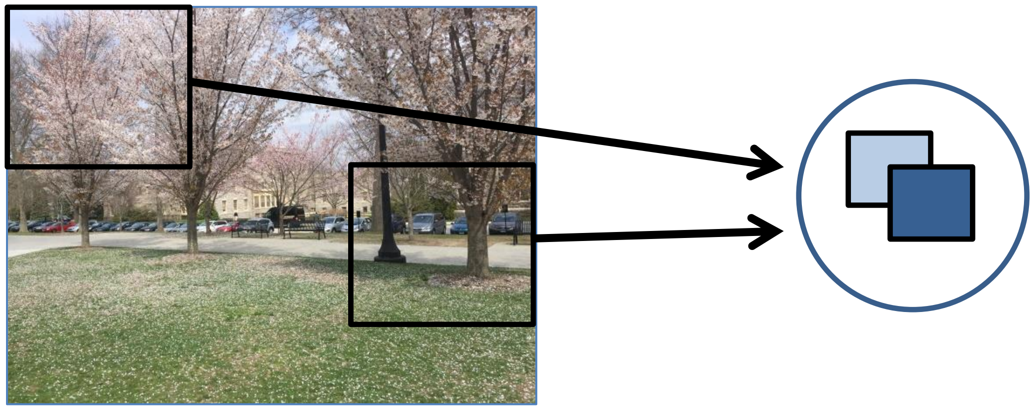


Figure 12. Examples of two local receptive fields being input into a neuron.

It turns out that this approach is directly modeled after the biological visual cortex. Neurons in the visual cortex are activated by stimuli in a specific location based manner. This means that neurons that are linked as direct neighbors share the saturation from an impulse. This analogy is explained by professor Andrew Ng et al. of Stanford (adjunct, as of Spring 2017) in their Unsupervised Feature Learning and Deep Learning (UFLDL) notes [17]. We also find this analogy in the syllabus for the popular Stanford course run by professor Fei-Fei Li and researcher Andrej Karpathy *CS231n: Convolutional Neural Networks for Visual Recognition* specified under "the brain view" section [18]. This method of subsampling is defined as *convolution*.

The local receptive fields can be seen as small windows that slide over our image, where the number of panes on the window is predefined. These panes help determine what features under the window we want to extract, and over time these features are better refined. As an analogy, the tint of the window panes may change over time to allow a better view of the right features. These tints can be seen as the weights of the network, and by extension these local receptive fields define the features of the network. The weighted windows are commonly called *kernels*. In Figure 13, we can see that the kernel defines how the image gets subsampled. The matrix operations use the weights of the kernel (red font) and the values of the image (black font) to produce parts of a newer, smaller image by placing the product of the operation in a correlated region in the new image. Simply, the yellow square regions define the new image by producing the red regions in the new image. In this way, the network picks up relevant features for learning from the original raw image.

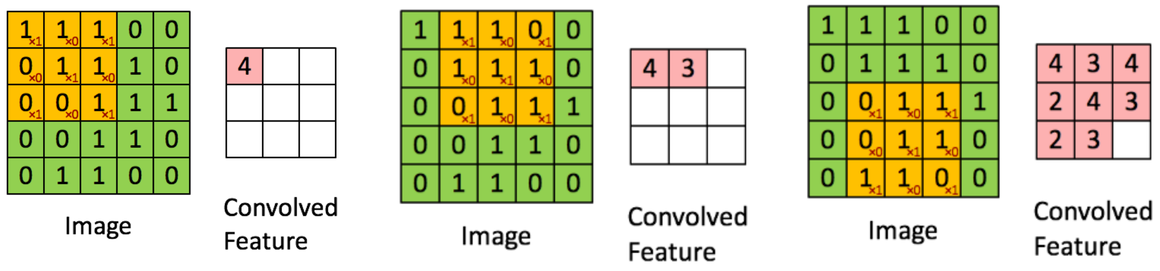


Figure 13. Different stages of convolution of the same image are shown.

Source: http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution/

Depending on the type of kernel, different features of the image may be highlighted. In the Figure 14, different kernels produce different results, such as blurring and sharpening. In this way, networks can develop identification of complex patterns in datasets just by applying kernel filters like this. One astounding fact about deep networks is that they develop these kernels through training without being explicitly programmed to do so. The only supervision is from a loss function in the output layer denoting how close the network’s prediction was to the actual value of the image. Through training, these kernels become more fine-grained to reduce the loss function’s output.

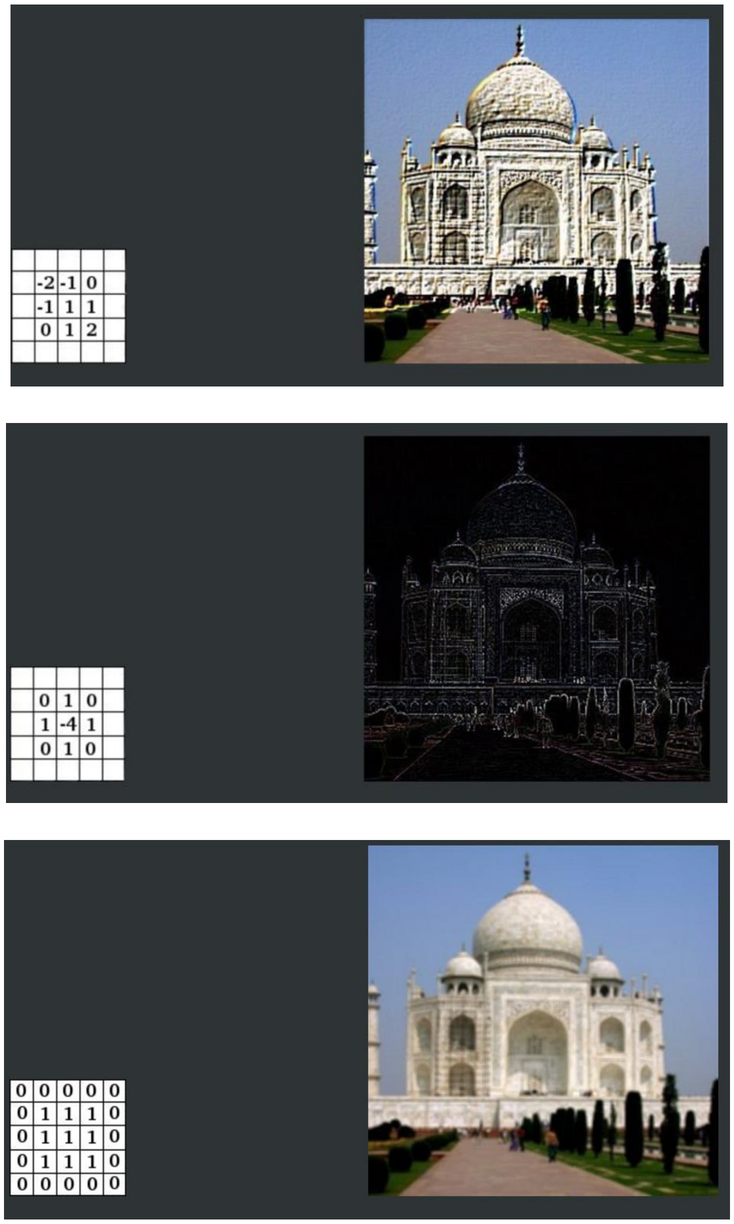


Figure 14. Three different types of convolution kernels are shown (from top to bottom, respectively: emboss, edge detect, blur) with their effects on the same image.

Source: https://docs.gimp.org/en/plug-in-convmatrix.html

Pooling:

Deep networks are stacked in such a way as to include many different types of layers. A general strategy is to follow a convolution layer with a type of layer called a *pooling* layer. The convolution layer is responsible for learning the lower level features of an image. These might be edges and the like. The pooling layer seeks to detect higher level features. Generally, these could be something like corners where two or more edges meet. Pooling is also good for building translational invariance. This means that even if the object moves to a different part of an image frame (“bottom left”, “top right”, etc.) we are still able to detect it. In order to have translational invariance we can pluck a sample from a whole region of pixels. In this way, dominant features can move further through the network. Along with this benefit, the image is also reduced dramatically because it is downsampled in one of three ways:

1) *Max pooling* – The maximum pixel value is chosen out of a rectangular region of pixels.
2) *Min pooling* – The minimum pixel value is chosen out of a rectangular region of pixels.
3) *Average pooling* – The average pixel value is chosen out of a rectangular region of pixels.

Reducing the size of the image dramatically cuts down on the amount of processing needed to train the higher level features of the network. In terms of processing, the idea is similar to convolution as we still pass a window over our image. Figure 15 visually demonstrates this procedure.

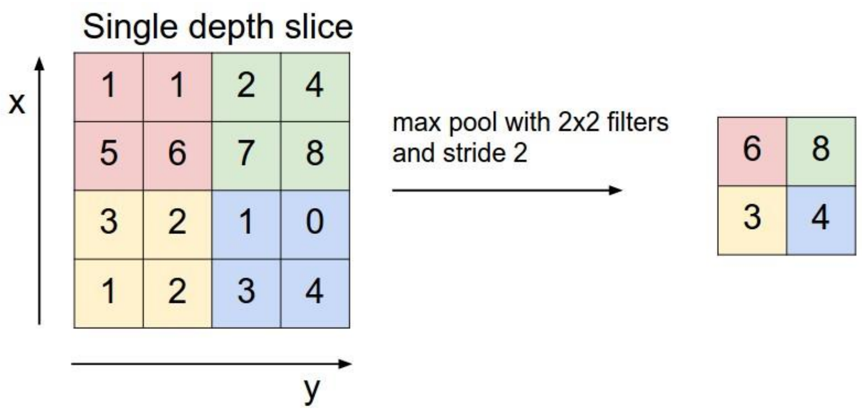


Figure 15. A 4 x 4 pooling window is passed over this image, where the *maximum* pixel value is chosen each time in order to downsample the image.

Source: http://cs231n.github.io/convolutional-networks/ (max pooling section)

Recall that the convolution layer passes convolution windows over the image to produce new images that are smaller. The number of images produced can be specified by the programmer. Of course each new image will be accompanied by a convolution kernel signifying the weights. Pooling is applied to each one of these new images. In Figure 16 below, pooling is applied to all 64 images that come out of the previous convolution layer. Here each image is of dimension 224 x 244. The output of this layer is 64 other images that are of dimension 112 x 112, effectively dividing the dimensions of the image in half.

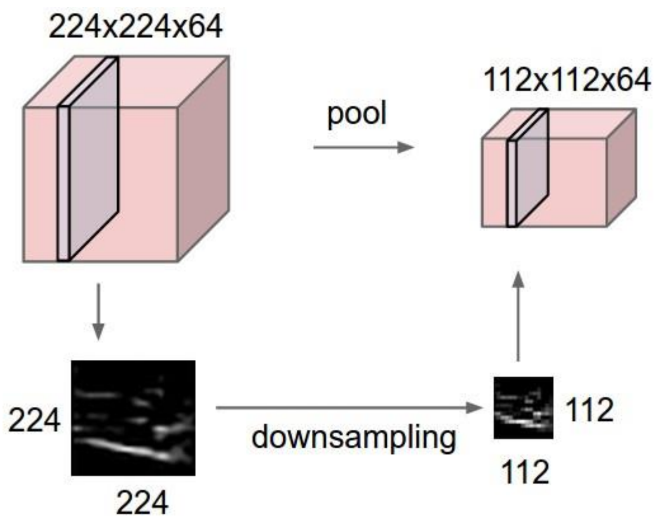


Figure 16. An input volume of 64 images of dimension 224 x 224 is pooled to produce an output volume of 64 other images of dimension 112 x 112.

Source: http://cs231n.github.io/convolutional-networks/ (max pooling / downsampling section)

Convolution and pooling dominate the discussion about types of network layers. However, there are a few other types of layers that were used in this research. The next few subsections give a briefing on this.

Rectified Linear Unit:

Of recent, the Rectified Linear Unit (RLU) layer has grown in popularity. Many researchers consider this over using the sigmoid activation function. In fact, Alex Krizhevsky et al. were able to accelerate convergence in their training by a factor of 6 times in relation to the sigmoid activation function using this function [19]. This is a fairly straightforward operation: the function takes a numerical input X and returns it if it is positive, otherwise it returns -1 * X. This effectively eliminates negative inputs and boosts computation time as no exponentiation and computations of the like are needed.

Local Response Normalization:

The Local Response Normalization layer imitates biological lateral inhibition. This concept underlines the fact that excited neurons have the capability of subduing neighbor neurons [20]. In essence, a “message” is amplified and focused on by this differential in neuron excitement. In simpler terms, these layers allow neuron’s with large activation values to be much more influential than other neurons. Following the pattern of feature recognition in every layer, these layers allow significant features to “survive” deeper into the network.

Fully Connected Layer:

The fully connected layer, namely, is like any regular Multi-Layered Perceptron. Generally, this is the final layer of the network. The outputs of the neurons in this layer are the actual outputs of the network. Connected to this layer is the loss layer whereby the network compares desired outputs to actual outputs, and the learning is initiated here in terms of gradient descent updates. We have covered this layer in our discussion about regular ANNs.

Relevant Works in Deep Learning:

In order to better understand the role and use of deep neural networks for our own research purposes, it is important to appreciate the progress made recently. For years, object recognition tasks have been a hallmark of the Machine Learning community. Deep Learning has outperformed many other algorithms in this arena. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has attracted and pushed many researchers to develop advanced algorithms. As we previously hinted at this in the introduction of the thesis, this challenge includes 1000 different categories for images to be classified into. The training set by itself is about 1.2 million images. There are 50,000 images for validation (images that the network will never have access to throughout the learning), and 150,000 testing images (images that the network can be benchmarked against during the training) [21].

Krizhevsky, Sutskever, and Hinton put forth a foundational paper in regards to Deep Learning in 2012 [19]. They developed a neural network with 60 million parameters and 650, 000 neurons. This network had 5 convolutional layers alongside a few pooling layers and 3 fully connected layers including a final output layer of 1000 outputs. At the time, they achieved a top-5 classification (of the 1000 classes) error rate of only 15.3% compared to a much higher second-place error rate of 26.2%. This paper contributed to the discussion of the importance of depth in neural networks by noting that removal of a single hidden layer dropped the top-1 classification error rate by 2%. From our earlier discussion of convolution kernels, we can appreciate, in Figure 17 below, a visual representation of the kernels of such a network that can handle thousands of classes. Some of these kernels are clearly edge detectors while others may apply some form of blur to the image to highlight particular features.

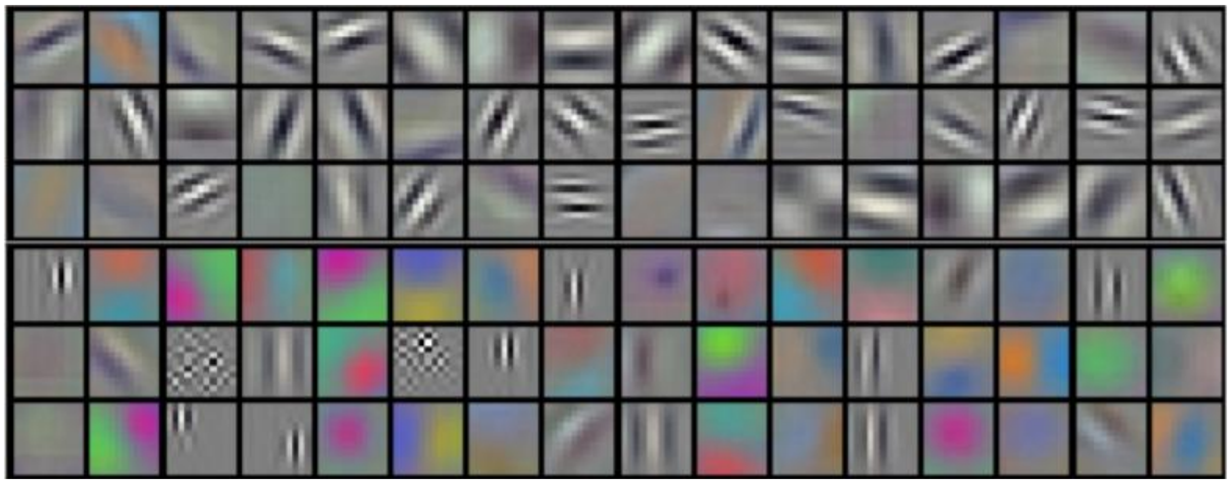


Figure 17. The convolution kernels of the first convolution layer of the network detailed in the discussion above.

Source: Paper by Krizhevsky et al. (*ImageNet Classification with Deep Convolutional Neural Networks)*. See end notes.

In 2014, Szegedy et al. entered the ILSVRC challenge with a 22 layer deep network nicknamed GoogLeNet – in part because most of the engineers and research scientists on the team worked for Google at the time. Astoundingly, the team won the competition with 12 times fewer parameters than Krizhevsky's deep network noted above [22]. The team obtained an impressive 6.66% error rate for top-5 classification. To reiterate the difficulty of building such an effective network, we can look at the complexity needed to differentiate between the two classes of animals in Figure 18. The dataset included distinctions that were as hard as this.

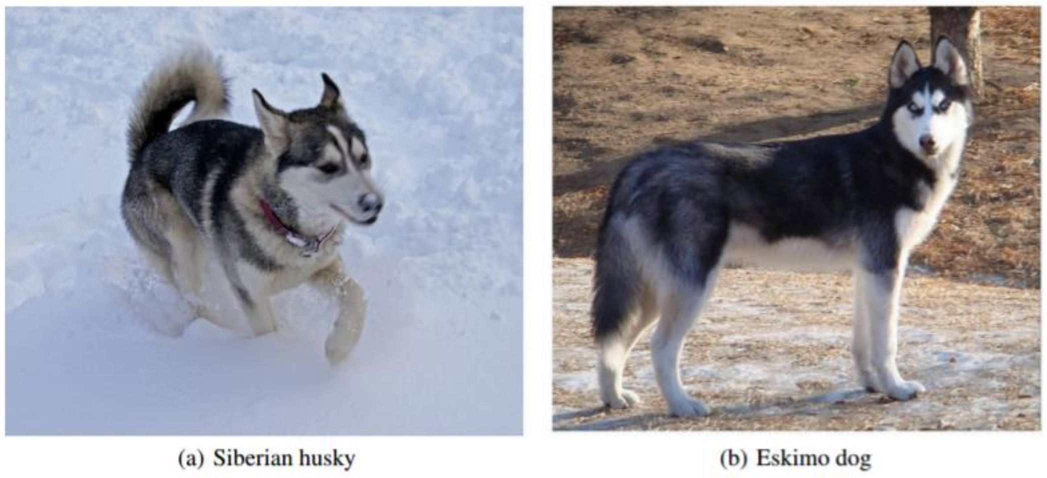


Figure 18. Two different classes from the ILSVRC challenge in 2014. We can see here how similar these two classes look, even to the naked human eye.

Source: From a paper by Szegedy et al. (*Going deeper with convolutions*). Refer to end notes.

Following the pattern of improvements, another team in 2015 yet surpassed the previous research team. He, Zhang, Ren, and Sun of Microsoft Research used a 19 layer deep neural network for the task and obtained an accuracy of 4.94% for top-5 classification [23]. This is a landmark accomplishment as it is purported to be the first to beat human level performance for the ImageNet dataset. Human level performance is noted as 5.1% for the dataset. The details of the paper are not easily described as the improvements are mathematically advanced. In general, newer layer activation functions were introduced to boost learning.

The most relevant dataset to our research is that of CIFAR10 from the Canadian Institute for Advanced Research [26]. Alex Krizhevsky outlined the use of this dataset when he developed it in 2009 for his Master's Thesis during his time at the University of Toronto [24]. Prior to this, tiny images on the scale of 32 x 32 were not easily labeled for classification tasks in regards to algorithms like deep learning. The CIFAR10 dataset includes 10 different classes, noted as such:

airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The classes are set up in a way to be mutually exclusive. For example, automobile and truck are completely different categories. Krizhevsky developed different deep neural network models in 2010 to run training with the dataset. At the time he obtained the highest accuracy using this dataset as his best model classified objects correctly with a success rate of 78.9% [25]. Since then, Mishkin and Matas have obtained 94.16% accuracy on the CIFAR10 dataset [28]. Whereas, Springenberg et al. have obtained 95.59% accuracy [27], and the current best performance is by Graham with an accuracy of 96.53% using max pooling [29].

Tools and Platforms Used for Research:

Most of the heavy lifting Deep Learning work for this research was done with the Caffe framework. Implementations were ported with the Python API. To interface with the physical world and to develop an experimental scheme, the TurtleBot framework was used. This allowed us to combine both robotics and Deep Learning. Underneath the hood, functionality of the Robot Operating System (ROS) was used to control the robot. These tools and platforms are described below.

Caffe:

Caffe is a popular open-source Deep Learning framework. It has been marketed as one of the fastest implementations available [30]. Within the Caffe community and online repositories, several different models are available to be experimented with and many are based on popular papers from both academia and the industry. Jia choose to use the Google Protocol Buffer format as an interface with Caffe to improve readability and to have efficient serialization. Caffe allows users to establish hyper parameters in a simple manner. In one file, users can specify:

- the base learning rate
- the learning rate policy (step: dropping the learning rate, fixed: keeping the learning rate fixed, etc.)
- the gamma value (how much the learning rate will be dropped by)
- the step size (choosing the frequency of the learning rate drops)
- momentum
- the number of maximum iterations for the network to learn

There are several dependencies that are needed to run algorithms using Caffe, including, but not limited to: protobuf, leveldb, snappy, opencv, hdf5, lmdb, boost, blas, cuda, and python or MATLAB. A complete rundown of the functionality of Caffe is provided at the website link under the tutorial tab. This includes a description about data structures unique to the Caffe library and how to pass data to those structures.

Caffe follows a stochastic gradient descent learning model. However, since most deep networks ingest massive amount of data with large datasets, running gradient descent for each image would be very costly. Hence, the loss function in Caffe happens to work through batches of data. This “batch gradient descent” method averages the loss function over a whole batch of images and then updates the weights accordingly.

While Caffe is very modular and there are many models available to work with, creating one’s own experiment is not necessarily trivial. Each training/testing experiment requires several different files, including:

- a training and a testing text file delineating labeled images and their locations (such as: “imgs/run_im1 0” and “imgs/run_im2 1” where the first part of each quote before the space is the image location and the second part is the label)
- a mean file used to average images across a whole dataset
- a training and testing dataset stored in LMDB (Lightning Memory-Mapped Database) format; there are other types of formats available, but this serves most experiments
- a solver protobuffer file delineating the training hyper parameters
- a training and testing protobuffer file actually defining the layers of the network
- a protobuffer file defining the network without a loss layer – this is only for forward propagation purposes

We have provided further documentation and scripts and stored them on the TurtleBot to simplify the training and testing pipeline in order to make the ramp up period easier for new users and future researchers.

TurtleBot:

While the robot used for this research was not marketed as one that is based on the TurtleBot framework, its functionality is essentially equivalent to that of the TurtleBot platform. The research robot was marketed as a “Deep Learning Robot” from a robotics company, Autonomous AI [36]. The robot includes an Asus Xtion Pro 3D Depth Camera, a microphone embedded in the camera, and a speaker. A Kobuki mobile base allows it to rotate and move in any direction on the ground plane. Most importantly, it is equipped with an Nvidia Tegra TK1, which allows us to carry out Deep Learning computations on a GPU instead of having to resort to extremely long wait times for training with a CPU. This is its main differentiator from a regular TurtleBot. While the Tegra TK1 is marketed as the world’s most powerful mobile processor, it only has 2GB of memory. This is problematic for training very deep networks and holding too many parameters in memory causes the robot to crash. While training, the robot is unstable because of this limited memory so running multiple programs at the same time is to be avoided.

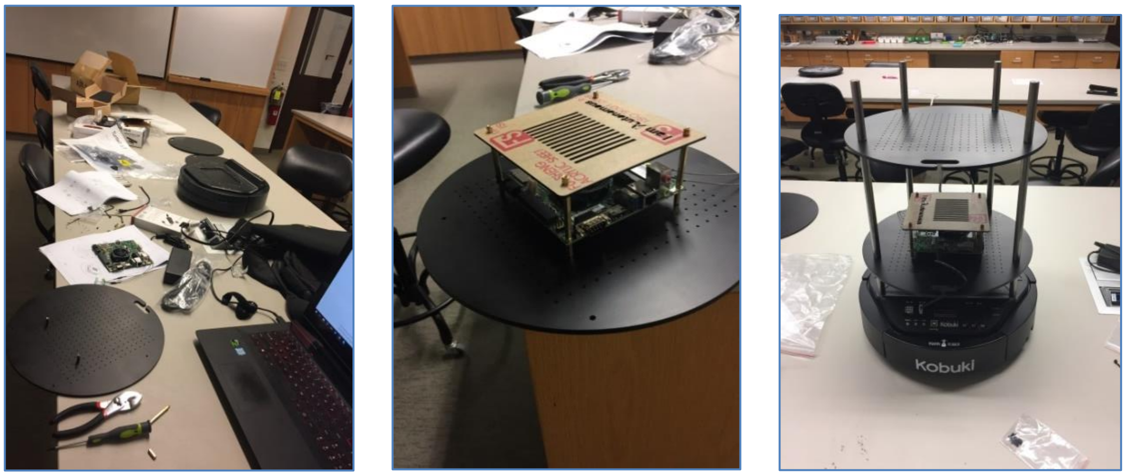


Figure 19. A sampling of pictures from the building process of the DL robot.

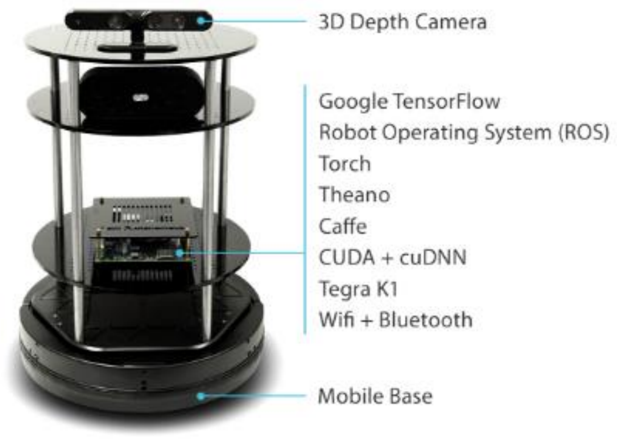


Figure 20. An official marketing image of the DL robot.

Source: https://www.autonomous.ai/deep-learning-robot

Google TensorFlow, Torch, Theano, and Caffe are all Deep Learning frameworks. CUDA and cuDNN are needed for implementing Deep Learning on GPUs and for speeding up that computation. This robot is virtually a computer in itself, and it allows us to treat it as such as it is very compatible with Ubuntu 14.04. Wi-Fi and Bluetooth give us the flexibility of setting up network connections and connecting devices for communication which we may consider in the future. In fact, the speaker that comes along with the robot is a Bluetooth speaker. The TurtleBot framework works hand in hand with the Robot Operating System (ROS).

Robot Operating System (ROS):

ROS is used to control the robot and to have access to all information coming from any of the robot's sensors [37]. ROS is an "open-source, meta-operating system" which allows hardware abstraction, low-level control and message passing between different modules/processes. ROS also allows us to package different modules in an efficient way in order to develop modular programs. ROS can also be described as a distributed framework of modules called *nodes* (note that this is much different than an ANN node) which enables us to execute different processes at the same time. Basically, any form of computation can be packaged into a node. For example, finding range from a wall using a laser, controlling the wheels using the motor, and performing face detection can all be separate nodes. We can have nodes run on different machines that have ROS installed. These nodes are able to find each other and exchange information through the *ROS Master*.

In a way, ROS has its own network set up. This is very similar to how the Domain Name System (DNS) works and ROS indeed uses the standard Transmission Control Protocol/ Internet Protocol (TCP/IP) sockets for communication. Nodes are directly connected to other nodes while the ROS Master provides a naming service. For our research, we can communicate from a remote laptop to the robot and vice versa. We can run a node on either the laptop or robot, terminate a node on either, and even send information from one node on one device to a node on the other device. *Messages* in ROS lingo are data structures that can have standard primitive types along with arrays in order to package information in an organized manner. Lastly, ROS hinges on a publisher-subscriber model of communication. This means that one node can send out a message by *publishing* it to a *topic* (think of this as a virtual boxed location). If another node needs this message, it can *subscribe* to the same topic to obtain that information. An accessible example in terms of mobile robots with cameras can be provided here. A robot's control program may have a motor node which publishes to a topic named velocity or the like, and our image processing node can then subscribe to the velocity node in order to label images with values of velocity. Figure 21 outlines a sample ROS network.

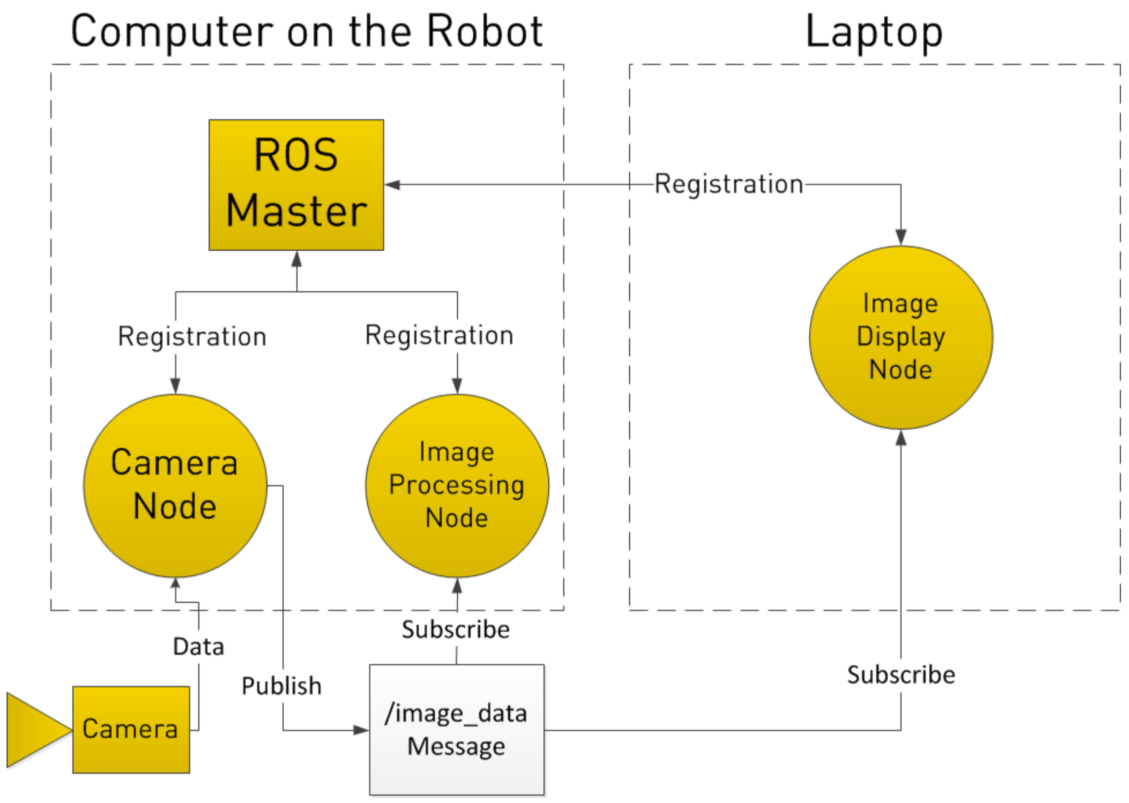


Figure 21. A graphical representation of the ROS Node network is shown. The concepts of subscribing and publishing to a topic are simplified with this graphic as one can see that the "/image_data" topic is available both to the remote laptop and the robot.

Source: http://www.clearpathrobotics.com/guides/ros/_images/ros101three.png

To put a few ideas together, the TurtleBot can use information from its sensors about the environment and publish them to topics. The robot is capable of SLAM (Simultaneous Localization And Mapping) and so it is able to build a map of the environment. ROS in the backend would handle bringing the robot up on the network. ROS would also handle all communication from the depth camera, and ROS would also be able to interface with a visualization program solely developed for mapping. The user would then be able to see the map of the room on their own laptop while the robot is driving without any data tether.

We have provided a document that is saved on the robots memory with an ordered list of commands for new users to follow in order to see different experimental setups and common TurtleBot use cases. This includes building a map of the room and programs that enable the robot to follow a human using a depth image.

Relevant Works using the TurtleBot Platform:

A significant amount of research has gone into applications of TurtleBot in terms of human interactivity. Hotze used the TurtleBot platform to develop a robotic framework for identifying anomalies [31]. These anomalies are based on the detection of unconscious humans. Mannequins were used in substitution of humans to test the system and the robot approached fallen mannequins from different position in a mapped environment. “Humans” were detected based on temperature, breath, and face detection. Furthermore, significant additions to the TurtleBot platform were made. For example, an Arduino, two Adafruit motor shields, a robot arm, a webcam, a temperature sensor, and a breath sensor were attached to the body of the TurtleBot.

While Correa et al. did not use the TurtleBot platform, it is important to note their paper because significant work was done in developing a system for human detection and recognition with a robot. Thermal images were used to identify humans along with raw visual images of faces. The approach used here was a nearest neighbor classifier that utilized similarity measurements like histogram intersection [32]. The research environment developed for this paper is perfect for the Turlebot platform. Specifically, at one point the robot was used to detect particular humans in a waiting room. The TurtleBot is a small, non-intrusive robot that is capable of navigating environments like this.

Moreover, detecting humans is not a trivial task for a robot close to the ground. Either the camera needs to be tilted upward or a camera needs to be mounted on a rod. Gritti et al. developed a Kinect camera based approach to detecting and tracking humans [43]. The focus here was on identifying "leg-like" objects that protruded from the ground. This is a natural approach to take considering that the TurtleBot is a short robot with a limited view from the ground. They developed a statistical classifier to differentiate between legs and other objects protruding from the ground plane.

There has been strong interest in using the TurtleBot platform for obstacle detection and avoidance. Boucher used the Point Cloud Library and depth information along with plane detection algorithms to build methods of obstacle avoidance [33]. High curvature edge detection was used to locate boundaries between the ground and objects that rest on the ground. Furthermore, researchers have considered the use of Deep Learning for the purpose of obstacle avoidance using the TurtleBot platform.

Tai, Li, and Liu used depth images as the only input into the deep network for training purposes [34]. They discretized control commands with outputs such as: “go-straightforward”, “turning-half-right”, “turning-full-right”, etc. The depth image is from a Kinect camera with dimensions of 640 x 480. This image was downsampled to 160 x 120. Three stages of processing are done where the layering is ordered as such: convolution, activation, pooling. The first convolution layer uses 32 convolution kernels, each of size 5 x 5. The final layer includes a fully-connected layer with outputs for each discretized movement decision. In all trials, the robot never collided with obstacles, and the accuracy obtained after training in relation to the testing set was 80.2%. Their network was trained only on 1104 depth images. This hints at the idea that maybe many edge cases are missing or that this dataset is specific to only one environment. The environment used in this dataset seems fairly straightforward – meaning that the only “obstacles” seems to be walls or pillars. The environment does not seem dynamic. A sample image from the paper is shown below. Tai and Liu produced another paper related to the previous paper [35]. Instead of a real-world environment, this was tested in a simulated environment provided by the TurtleBot platform, called Gazebo. Different types of corridor environments were tested and learned. A reinforcement learning technique called Q-learning was paired with the power of Deep Learning. The robot, once again, used depth images and the training was done using Caffe.

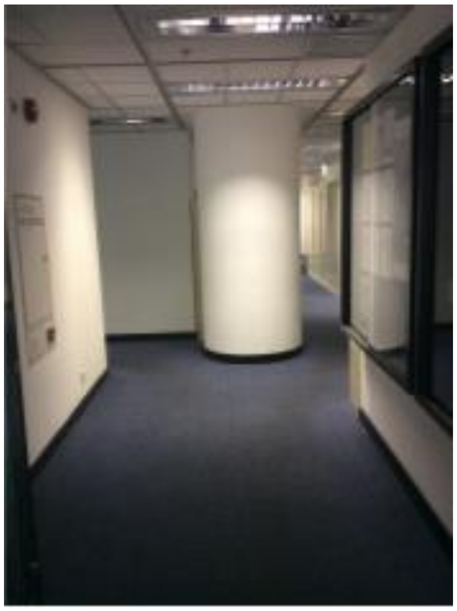


Figure 22. A sample image for an obstacle based environment from paper described above

Source: From a paper by Tai et al. (*A Deep-Network Solution Towards Model-less Obstacle Avoidance*). Refer to endnotes.

This Honors Thesis paper provides a distinctive approach in comparison to the three papers on obstacle avoidance listed above. Research like Boucher's does not consider higher level learning, but instead builds upon advanced expert systems, the likes of which can detect differentials in the ground plane. By focusing on Deep Learning, this thesis allows a pattern based learning approach that is more general and one which does not need to be explicitly programmed. The logic structure for obstacle avoidance can be fairly complex depending on the variability in the environment. While Tai et al. used Deep Learning, their dataset was limited with just over 1100 images. We built our own dataset to have over 30,000 images, increasing the size of the effective dataset by about 28 times. The environment for this thesis is much more complex than just the flat surfaces of walls and columns (covered in detail in next section). This thesis also developed a dataset that was based on raw RGB images, opening the door to further research with cameras that do not have depth. Moreover, the sizes of the images used in this thesis were dramatically smaller, which also opens up the door for faster training and a speed up in forward propagation. Lastly, the results of this thesis are solely tested in the real world and a simulated environment is not used.

Real-time Autonomous Driving and Obstacle Avoidance:

The TurtleBot platform was used to demonstrate the powerful learning capabilities of Deep Learning algorithms. An intelligent control program was abstracted from the learning of a network without explicit programming of specific scenarios. A large dataset of Asus Xtion Pro raw RGB images was developed to diminish the possibility of over fitting and to allow a broader pattern to be developed. The real world scenario of obstacle avoidance towards building an autonomous driving vehicle was addressed. An approach of “fine-tuning” a large network was taken to mitigate the constraints of computation time taken to learn the problem scenario. The TurtleBot successfully managed to learn a control program in order to drive autonomously, follow a path, and simultaneously avoid obstacles.

Task:

The problem scenario is that of training a deep neural network to learn autonomous driving of a vehicle in a tight, chaotic room/office environment. To test the functionality and success of the program, the performance of the robot was compared to the end goals. The end goals are primarily that the TurtleBot should autonomously follow an approximately rectangular path in a tight environment without colliding into obstacles. A description of this environment is provided below.

Environment:

Attention was paid to what obstacles were placed in the room and the placement of such obstacles in order to build a complex environment. The approximate rectangular path that was provided was the perimeter of a long lab table. This table only had 3 planes of support on the underside; otherwise there were gaps underneath the table. Figure 23 demonstrates this approximate environment set up. The white circles with dark borders are signified as chairs. White rectangles with dark borders are lab tables where the left and right side of them have gaps. The gap size is large enough for the robot to be able to drive through, and for that reason chairs were placed in those locations. The radii of the chairs are larger than the circles may demonstrate because the feet of the chairs extend further such that there is no gap for the robot to move in between neighboring chairs (in most cases). The dark brown rectangle is a boxed off area of the lab that may be used for other experiments, but there are borders that the robot would need to avoid hitting. The golden rectangles denote cabinets which the robot must avoid also. The red rectangle in the middle of the figure shows the path that the robot must follow or the general path it needs to go in on its way as it avoids chairs, tables, boxes, etc. This path must be completed in both clockwise and counterclockwise directions.

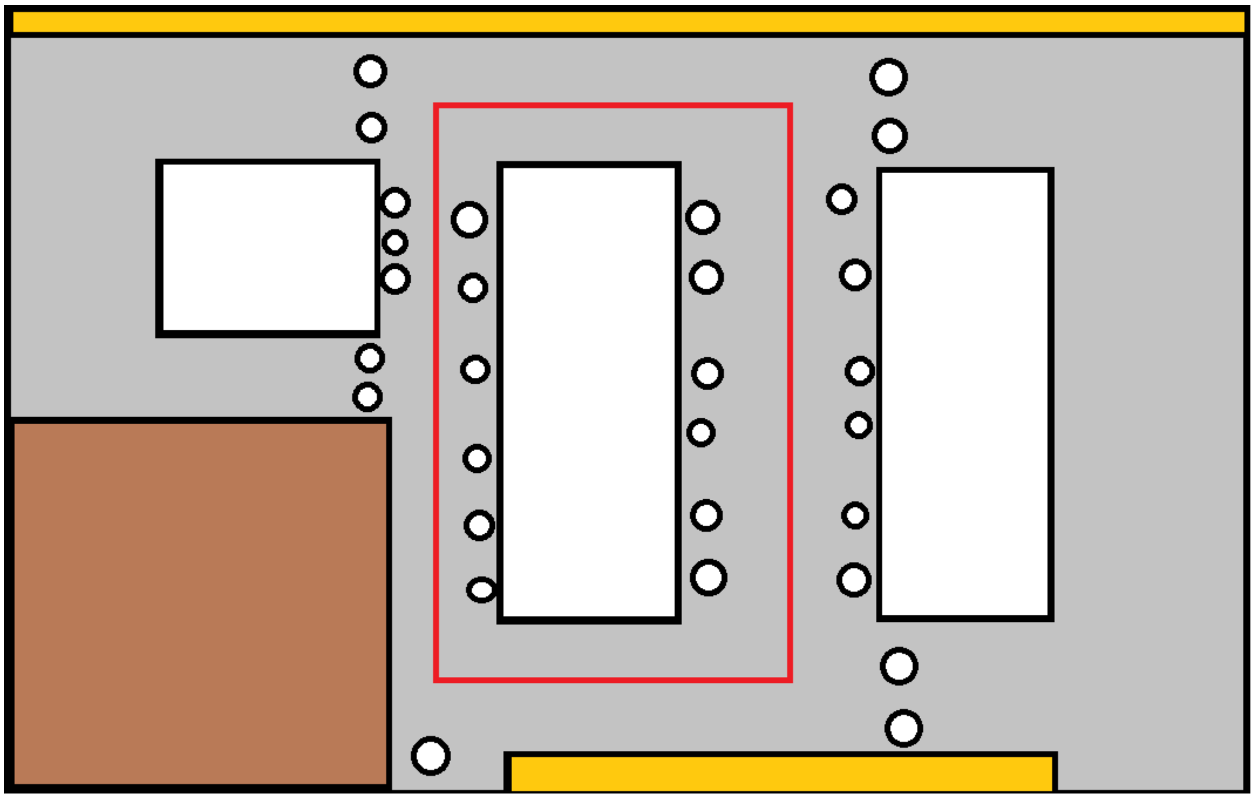


Figure 23. A visual of the environment with lab tables, chairs, and cabinets. Images are provided below to help understand this environment even more.

One can see from Figure 24 below that the gaps were closed with adjustable round chairs. Each chair has 5 rounded legs and a circular stump. The chair heights can be adjusted and the orientation can change 360 degrees for both the base and the actual seating. Sample images are provided below in Figure 24 to visualize different possible orientations for the chairs. These were chosen as the main objects of interest because there is clearly a good amount of gap area in between the object – specifically the legs. This allows for complexity in defining what an obstacle is and what and obstacle is not. The robot must not simply learn to follow the color of the carpet because even the gaps reveal the carpet.

The camera for the robot faces down at about 40 degrees from the vertical position, so it is important to design an environment that is complex enough, in terms of objects close to the ground, to be a problem of interest. To highlight the point of this experiment, if the environment was built only using cardboard or other flat material as the main obstacle in the environment, then it would be a fairly straightforward experiment. There would not be much variety, apart from lighting conditions, as to what material needed to be avoided. The chairs were moved by students overnight. While they might be in the same relative location, the orientations were completely different each time. This is a hard problem because it is not easy for a pattern to be developed since the orientation keeps changing. This does mean that for obstacle avoidance to be successful that the deep neural network necessarily needs to develop an understanding that chairs are to be avoided. With enough gaps in between chairs and the legs of the chairs having significant gaps, the robot will still see the carpeted area. Thus, it cannot just develop a control program to follow a carpeted area, but instead needs a more complex pattern to be recognized from the dataset.

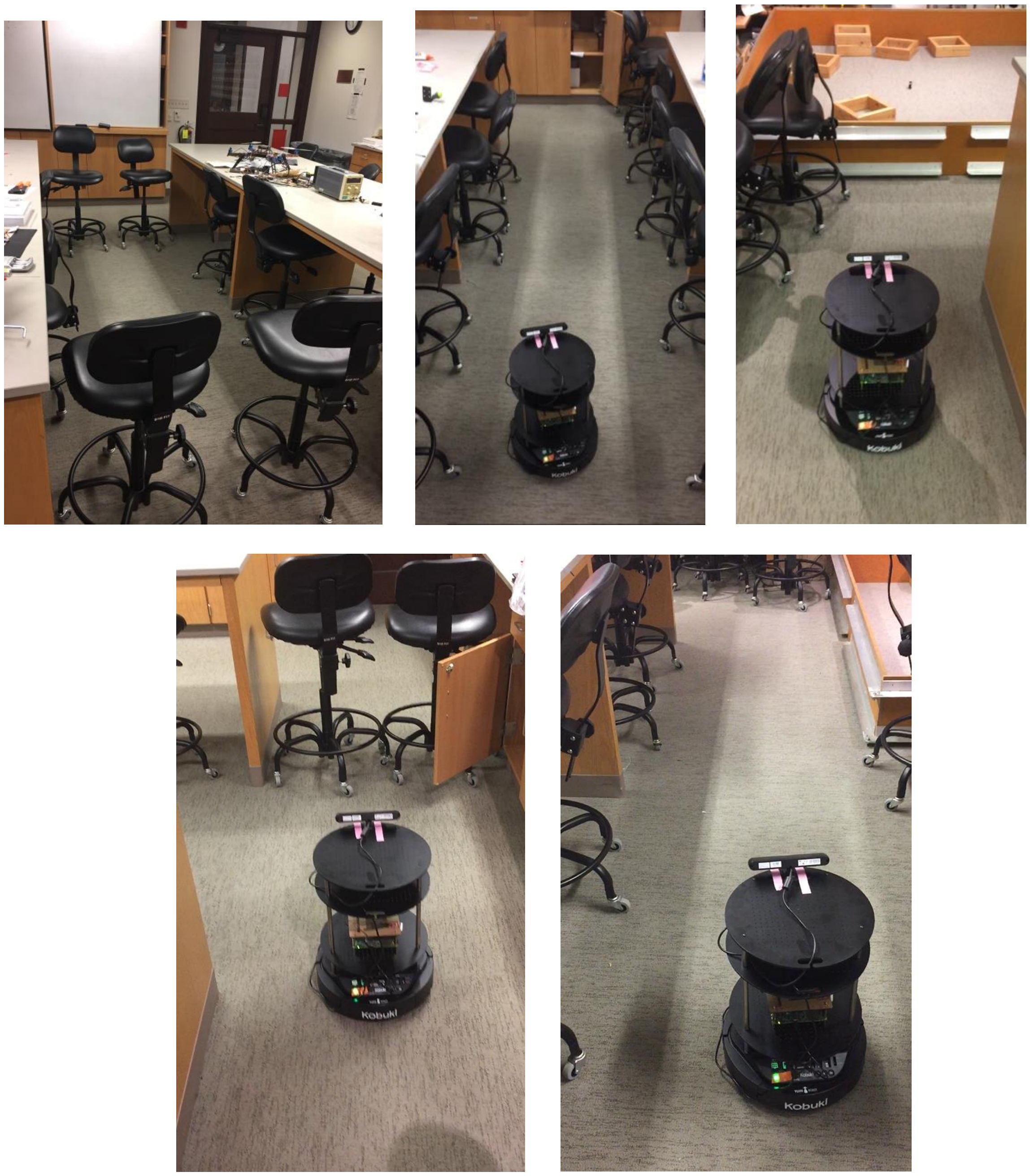


Figure 24. All the images above demonstrate various obstacle avoidance scenarios

It is important to establish robust guidelines as far as environmental set up because there may be scenarios that are impossible for the robot to solve. In the first of the three images in Figure 25, we can see that ideally we wanted the robot to avoid getting near the chair, but it turns out that there was enough of a gap between the chairs that the robot made the decision to go straight instead of turning away from the chairs. In the second of the three images, the robot is facing a cabinet directly head on. Even for a human with limited peripheral vision, it would be impossible to know which direction to turn. There is no way to have metaknowledge about which direction contains an obstacle and which does not. This is not a fair scenario to include in the dataset. To solve the former of the two issues, the enviroment included chairs that were placed close enough to have a small enough gap that the robot would not be able to fit through. To solve the latter of the two scenarios, cabineted area included an open cabinet that swiveled to a direction the robot was supposed to avoid. This is demonstrated in the third image. Not only does this add more choas to the environment (there are various different items in the cabinets which adds to the complexity of developing a pattern), but it also establishes rough guidelines as to the path to be followed.

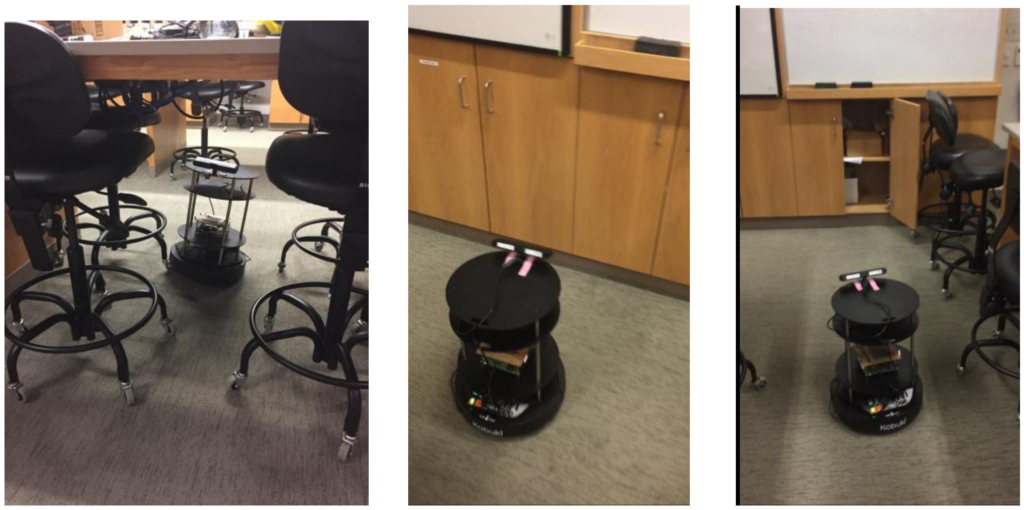


Figure 25. The two scenarios on the left were ommited from training. We can see that after some training, the robot managed to decide to go under the table in between the chairs. Hence, care was taken to put chairs closer together in the future to address the first scenario. The third scenario includes an open cabinet to address the second scenario.

Chairs, cabinets, and tables were not the only obstacles to avoid. A few images in the dataset included small cardboard boxes. A good amount of the dataset included the borders of a colony space environment (as shown in Figure 26 below). It was important to include obstacles like this in order to confirm that the concept of obstacle avoidance was being abstracted instead of the robot only avoiding black colored objects (the black chairs). It is also significant to note that students used the lab throughout the day and night, so conditions of the carpet changed while the dataset was being developed. For example, coins were found laid out on the ground near a turn in the path on one day. On another day, shreds of paper were at different locations on the path. We decided not to remove some of these items while building the dataset because it only adds to the diversity in what we might consider edge cases.

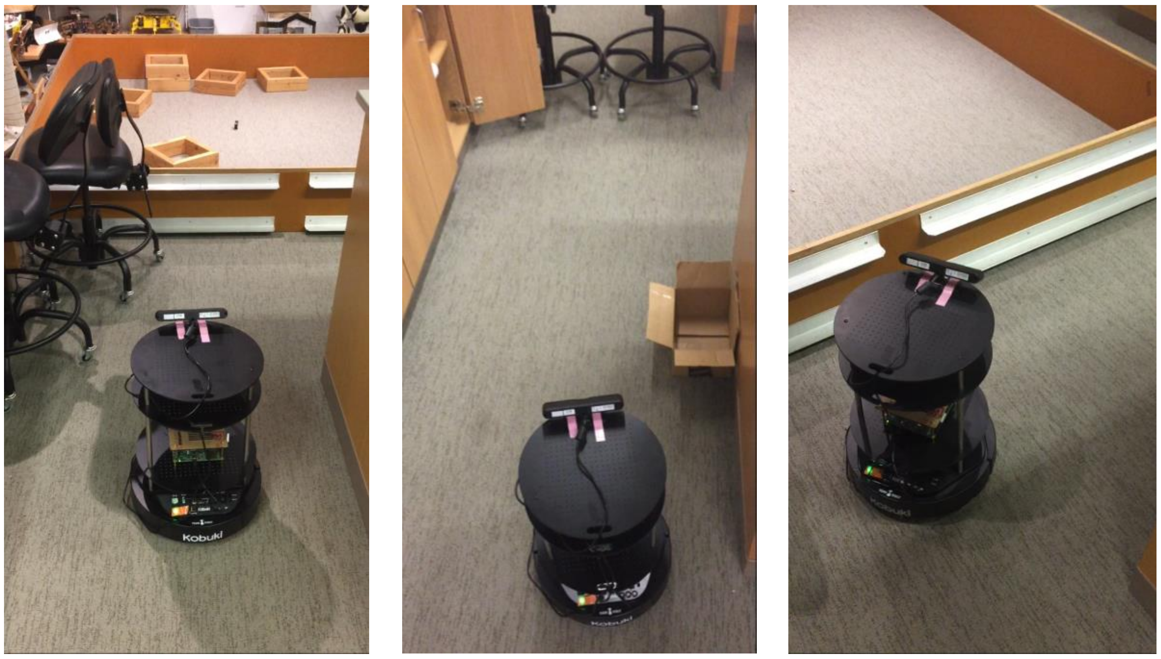


Figure 26. Some more examples of possible obstacles for the robot.

Dataset Collection:

We developed a script that was motivated by the keyboard teleoperation module of TurtleBot to label images. It is important to note that control of the robot happened remotely. From a laptop the *bringup* node was launched. This node is responsible for establishing a foundation for the communication to and from the robot. It sets up the Master Node to act as a name service to all other nodes. Next the *3dsensor* node was launched to bring the camera up for use. We have created bash aliases for the two actions above. Normally the user would need to type "roslaunch turtlebot_bringup minimal.launch" and "roslaunch turtlebot_bringup 3dsensor.launch", but with the aliases the user can simply complete the former command with "rosl" and the latter command with "rosc". Note that these two commands need to happen on two *separate* Terminal window instances. This is usually done remotely from a laptop through using a secure socket shell (ssh).

The control script written for labeling is interfaced with OpenCV to allow efficient image manipulation. Thus, the script (named as *pic_save.py*) on the TurtleBot subscribes to the camera node, extracts a message and converts this message into an RGB image using the OpenCV bridge. While this is happening, the script also publishes commands to the robot velocity / motor command node. The user is able to teleoperate the robot using keyboard keys that correspond to forward, left, and right. Every single time the use hits a key, the script saves the full size image in a folder with its name such that it is labeled with the corresponding action. For example, "run34_2.jpg" means that the robot took a left turn where "2" denotes left. Similarly, the value "0" denotes "go straight" and the value "1" denotes "turn right". In this way, the TurtleBot was driven around the lab following the path in both directions. To increase the diversity of the dataset, different starting points were chosen and hard scenarios such as being close to walls were considered. Overall, 30, 754 images were collected.

The script processes about 10 images per second, but not every image is saved. While no time record was kept, an estimated 1.5 – 2 hours were spent on trial runs and collections. In initial testing conditions we found that there were edge cases that were missing. More data was added over time. By default the images form the Asus Xtion Pro are of dimension 640 x 480. While this would a great amount of granularity to train on, it would take an incredible amount of processing power and time to train to a significant accuracy. For our deep network we downsample this image to the dimensions of 64 x 64. Saving the images full size also gives us the flexibility of experimenting with strategies based on a downsampling pyramid should we have the resources to process larger images in the future. Intuitively, larger, more detailed, images should boost learning since more information about the environment is available. To build more insight on the dataset, Figure 27 provides some examples of these images.

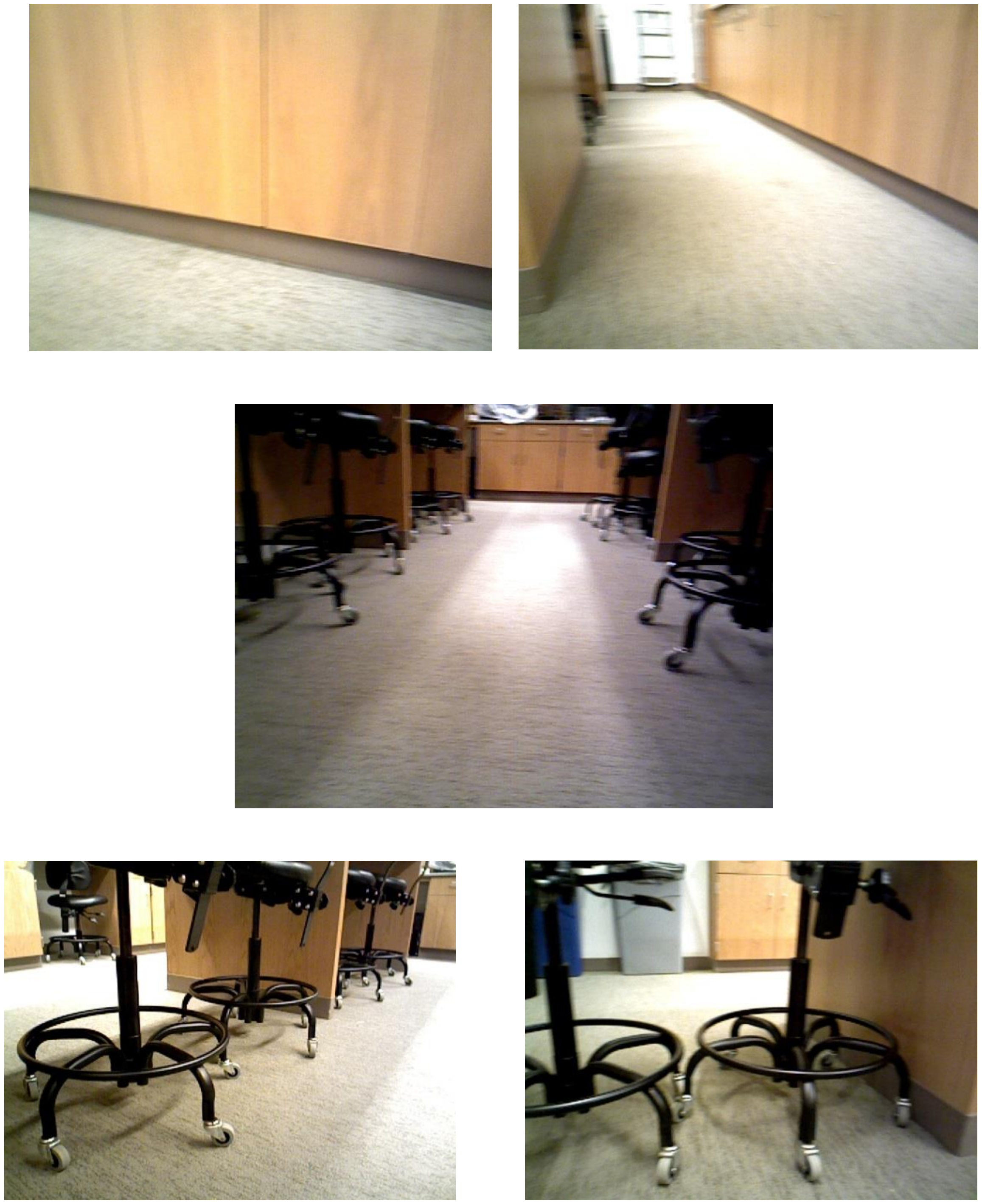


Figure 27. A sampling of the images in the dataset is provided. This is directly from the point of view of the TurtleBot.

Development of Deep Neural Network Architecture:

In the initiation of the research, a few different deep architectures were experimented with. Not all of these experiments were explored deeply as far as tweaking hyper parameters of the network since they did not seem promising to begin with. The initial tests were with only 1 convolution layer and 1 pooling layer. The idea here was to start with a simple model to allow several iterations on different strategies. After a few unpromising experiments with the shallow architecture, it was determined that a couple of steps should be considered. Firstly, the size of the dataset needed to be increased. At the time there were only 1600 images in the dataset. Secondly, to augment the training from where it was left off, the learning rate could be reduced. The dataset size was increased dramatically to the full size as listed before. With some tweaks to the learning rate, the accuracy did not show drastic change. The next change to consider was to build a deeper network. The reason this strategy was not chosen at the start is because the computations are expensive and tests need to be run over night. At the end of the semester, there was not much time to experiment with a large network built from scratch because of the sheer amount of training needed. Thus, there were two main issues: in comparison to most popular papers on deep architectures, the network initially chosen for this research was not large enough, and there was not enough time to train a large network from scratch.

After researching the capability of some other networks, we noted that Alex Krizhevsky's deep network architecture to solve the CIFAR10 dataset was not extremely large. Consideration to imitate the architecture of this network was taken. Even though it is not large in relation to other networks like GoogLeNet, this is still a large enough network where many experiments would be time consuming to carry out. In any case, we trained an imitation of this network with *the original CIFAR10 dataset*. We obtained about 74% accuracy for that dataset. The proposition here was that we might be able to augment this network with our own dataset. In other words, we can take the weights of the network from it having learned the CIFAR10 data, and then fine tune it for our own purpose – namely, obstacle avoidance and autonomous driving.

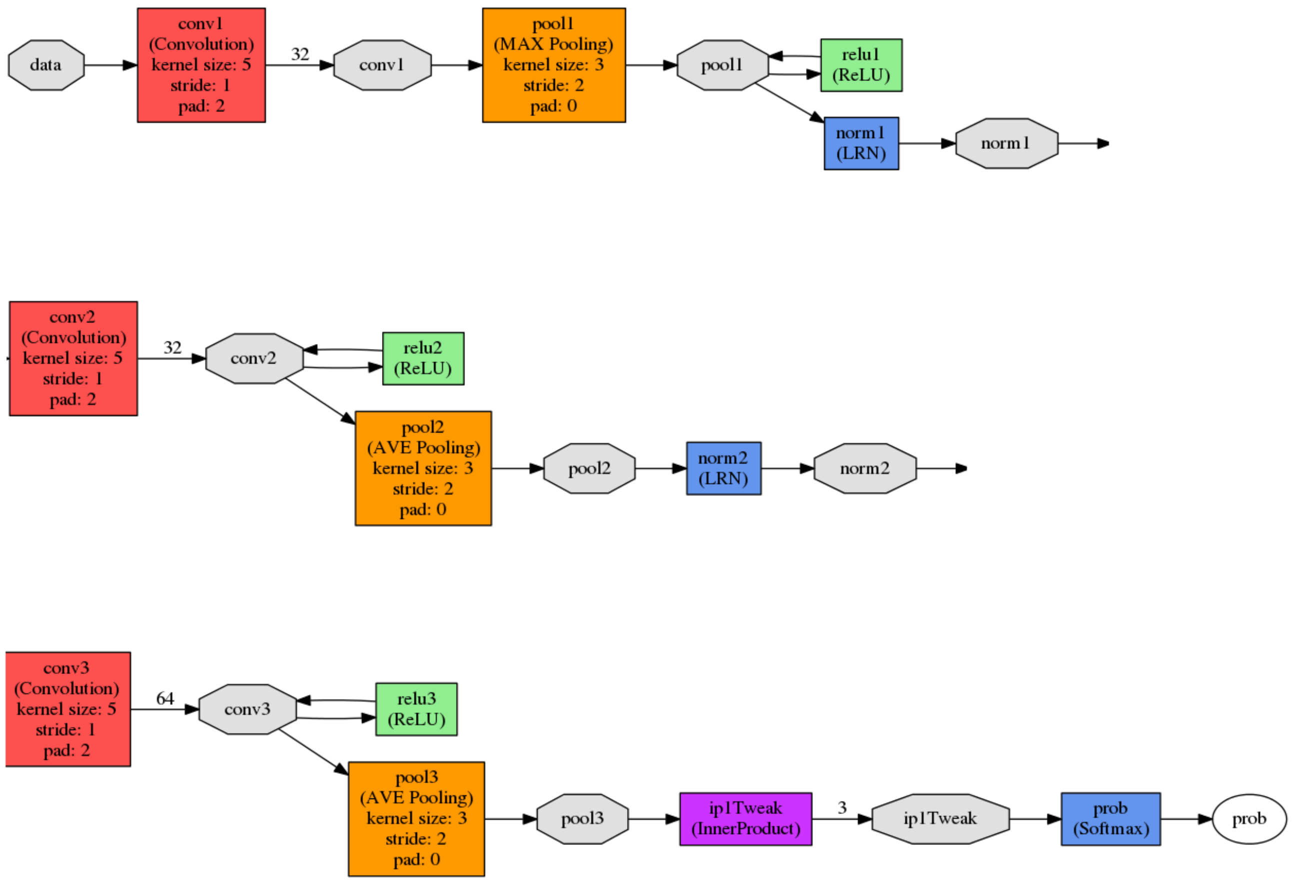
Recall that the CIFAR10 dataset includes 10 different classes. The thought for fine-tuning is inspired from the notion that the lower level features detected by the network are general enough to be applied to the problem of obstacle detection. Intuitively, there is a large difference between detecting an airplane and detecting a dog or a cat. However, Krizhevsky's network is capable of differentiating between the two based on the same kernel weights. That seems to be a large area of coverage for the type of data to be provided. The other thought here is that Krizhevsky's network was trained on 32 x 32 dimension images. Since our images will be 64 x 64 pixels we may expect that there will be a boost in accuracy. In the results section we will see that this approach proved to be fruitful.

The final network used for this research is shown below in Figure 28. It is split into three lines to ease the visualization. We can see that there are 3 iterations of the layer combinations of convolution, pooling, and normalization. Note that the fine-tuning of the network is evident from the visual. The layer "ip1Tweak" is labeled as such because the final layer of Krizhevsky's network was ripped out, and replaced with an inner product ("ip"), or also considered as fully connected, layer that only had 3 outputs. This is signified by the value 3 above the ip1Tweak layer in the visual. The 3 outputs correspond to the decision making of the TurtleBot in terms of autonomous driving directions. The original network included 32 convolution kernels for the first convolution, 32 convolution kernels for the second convolution, and 64 convolution kernels for the last one. We can also see how each convolution layer is immediately followed by a pooling layer. Every convolution layer also includes a rectified linear unit attached to it. Local Response Normalization also appears to be an effective addition to this network, as it augments the outputs of 2 of the 3 pooling layers. Later, we will see into each layer and visualize exactly how the weights and activations of the network look.

The dataset was split as such for the final network: 7,689 images for testing and 23,065 images for training. This is based on a 75% training split of the entire dataset. The hyperparameters were defined as such:

- testing iterations: 100; basically how many forward passes the test will carry out
- batch size: 77; this is for batch gradient descent – notice that batch size * testing iterations will cover the entire testing dataset
- testing interval: 150; testing will be carried out every 150 training iterations
- base learning rate: .001
- momentum .9
- weight decay: .004
- learning rate policy: fixed
- maximum iterations: 15, 000

These hyperparameters were determined through several experiments in order to find the desired level of accuracy and performance. Some of these parameters are surely subjective. For example, we considered testing interval to be much less than it usually is for large networks (on the order of 1000). The reason for making this a small value is so that we can analyze shifts in learning in a decent amount of time instead of having to wait for over half an hour. The number of maximum iterations was chosen as an estimation of the number of epochs the network may have needed to stabilize. The batch size of the training data is 77 images, thus we would need about 300 iterations to cover the whole training dataset. Hence, the number for maximum iterations was established as 15, 000 in order for the network to go through about 50 epochs.



data
conv1 (Convolution) kernel size: 5 stride: 1 pad: 2
32
conv1
pool1 (MAX Pooling) kernel size: 3 stride: 2 pad: 0
pool1
relu1 (ReLU)
norm1 (LRN)
norm1
conv2 (Convolution) kernel size: 5 stride: 1 pad: 2
32
conv2
relu2 (ReLU)
pool2 (AVE Pooling) kernel size: 3 stride: 2 pad: 0
pool2
norm2 (LRN)
norm2
conv3 (Convolution) kernel size: 5 stride: 1 pad: 2
64
conv3
relu3 (ReLU)
pool3 (AVE Pooling) kernel size: 3 stride: 2 pad: 0
pool3
ip1Tweak (InnerProduct)
3
ip1Tweak
prob (Softmax)
prob

Figure 28. The final architecture for the deep network. This is inspired by the architecture for solving the CIFAR10 dataset.

Results from Using Deep Learning for Autonomous Driving:

The initial shallow network (1 convolution and 1 pooling layer), was capable of attaining an accuracy of only around 50%. Note that this accuracy refers to performance on the testing dataset, which is data that the network is not trained on. For shallower networks, it is intuitive that one will attain lower accuracies since the function being developed is not as complex. A few tests showed promise when the learning rate was reduced from .001 to .0001. Accuracy of around 52% was obtained with this change. A hypothesis was proposed noting that if the size of the dataset was increased, then the accuracy would have a significant increase. Even with the full scale dataset, the learning still hovered around the same accuracy level.

As described above, the best and final approach was the fine-tuning of Krizhevsky's network. At first, we fine-tuned the network with only 20,000 images. The network accuracy plateaued at around the 10,000 iteration mark, and it took about 2.2 hours for the network to stabilize around the value of about 94% accuracy. The graphs are labeled with "khan_roam" to signify that we tested the network in the real world by letting the robot roam. While there was great accuracy on paper, it turned out that this network had trouble with making premature turns. For example, even though there are two chairs to the right of the robot 4 feet in front of it, the network decides to turn instead of moving straight a little longer. One interesting observation to make is that in the training performance, the network was immediately able to reach an accuracy of around 90% in about 200 iterations. It then takes a while to hit the peak of 94% accuracy. Since these experiments were conducted overnight, these extra iterations were uncaught. This does, however, provide valuable insight into how a large network like this may handle other data similar to our obstacle avoidance dataset.

From here on, the full dataset was trained with the model. This is about 10,000 more images than the previous experiment. Intuitively, we can expect that more edge cases will be considered. Notice from the graphs of the training that the network was able to obtain an accuracy of about 92% in about 4 hours. It took the network about 200 iterations to get to the 84% accuracy mark and around 2000 iterations to achieve an accuracy of about 90%. This is certainly a drop in accuracy on the dataset compared to the previous experiment. However, it important to reiterate that this is a much larger dataset. As far as real world accuracy, this robot rarely collided with any objects. 10 different test runs were completed where the robot was reversed after a completion of a lap in order to complete the lap in the opposite direction. The robot did, although rarely, slightly graze against the leg of a chair or a cardboard box. However, this did not change the trajectory of the robot and it was still able to complete its course. For this reason, these rare occurrences are not considered as major events for hitting an obstacle.

Even then, one may argue that the turning angle for the robot is the only issue here since this is such a tight environment. Though the network made the right decision, the movement of the physical robot may have been slightly too much. This can be corrected with very small tweaks in the values of turning radii for the different decisions, however this does not reflect or add to the discussion about the performance of the deep network in itself.

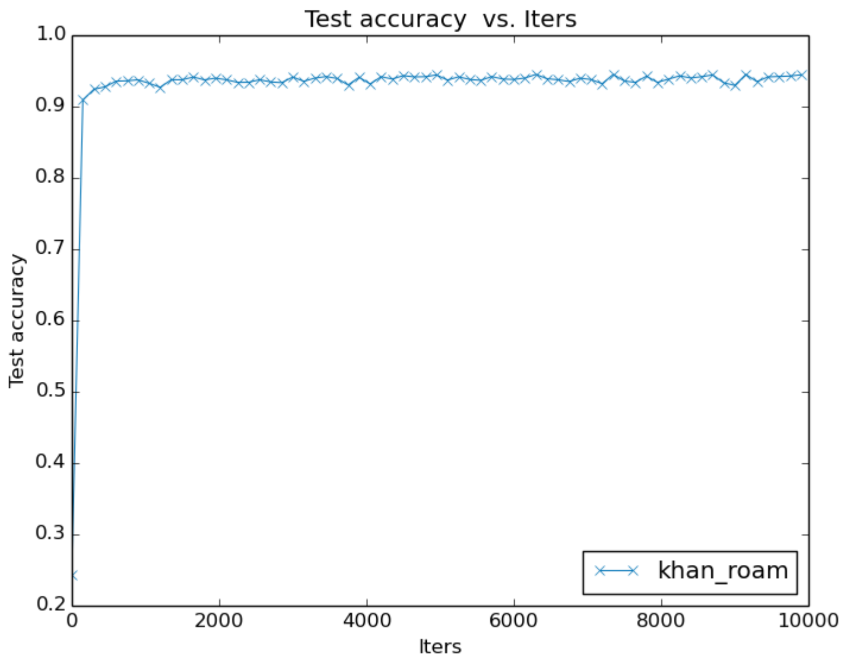


Figure 29. The performance of the network in relation to iterations for the fine-tuned Krizhevsky network trained with 20,000 images.

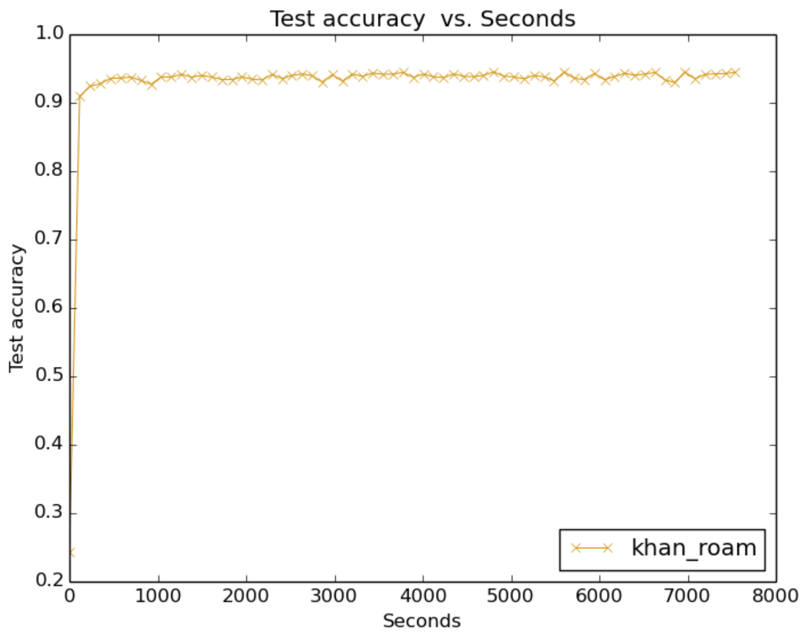


Figure 30. The performance of the network in relation to time for the fine-tuned Krizhevsky network trained with 20,000 images.

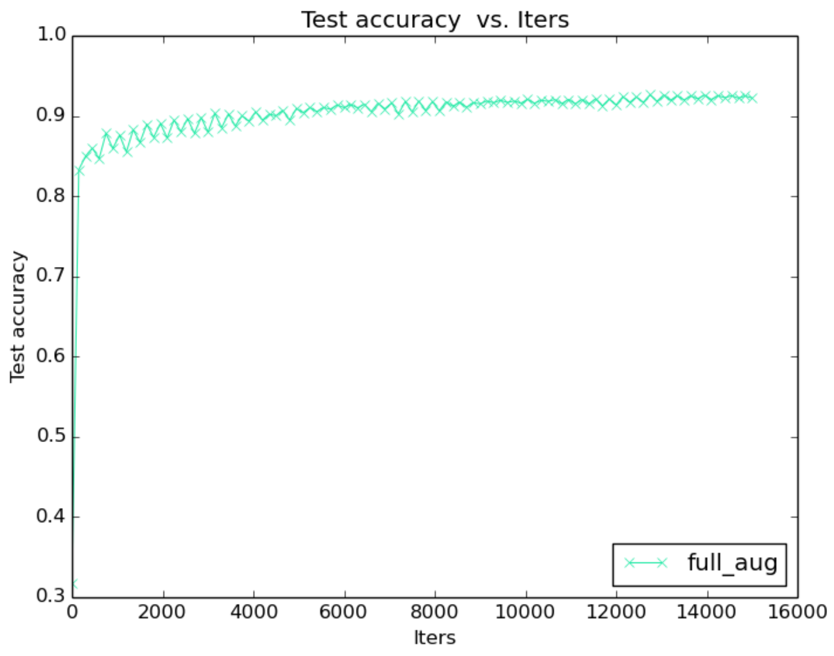


Figure 31. The performance of the network in relation to iterations for the fine-tuned Krizhevsky network trained with over 30,000 images.

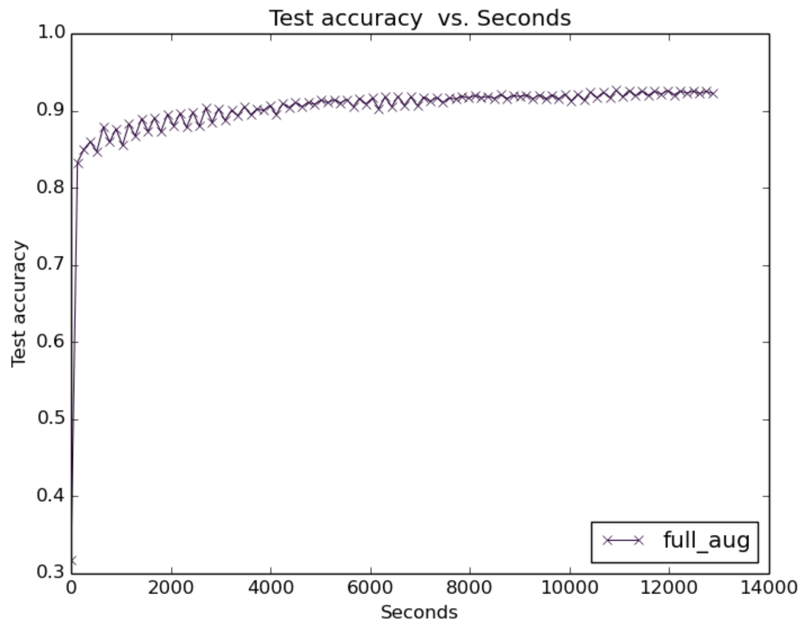


Figure 32. The performance of the network in relation to time for the fine-tuned Krizhevsky network trained with over 30,000 images.

Visual Analysis of Results:

We can visualize particular situations of interest and acknowledge that the robot performed the correct action. Various progressions shots of before and after are provided from videos to show the robot's decision making process. In Figure 33, we can see that the scenario of the open cabinet did not particularly challenge the robot. As previously mentioned, this helped augment the robots path learning. In Figure 34, we can see that the robot was successfully able to navigate the tight corridor and move away from the border of the colony space. This goes to show that the robot learned to avoid more than just the chairs. To provide more insight, Figure 35 shows the robot has clearly learned to avoid chair obstacles. Although, the cardboard box was seldom included in the original dataset, the robot clearly has pattern recognition broad enough to be able to avoid it – refer to Figure 36.

Results from Testing in New Environments:

To fully understand the extent of the learning we tested the robot in a real world scenario, the robot was tested in the original environment that the dataset was developed in. This is a fair assessment to make, and a true "validation dataset" of a sort since the robot has not seen these exact orientations as every run is different. To take this testing a step further, we explored testing the robot in three different environments.

The first approach was to see if the robot would be successful in avoiding bumping into walls in a hallway. For the most part the robot was able to turn away from walls in the hallway. This is especially surprising since there were very few images in the original dataset that had white walls to turn away from. Perhaps the robot is instead turning away from the dark brown bottom lining of the white walls as that color is close to the color of the chairs. The robot also successfully managed to turn away from doors in the hallway. However, a head on approach to a door is problematic and the robot has trouble navigating its way out of that situation. This makes sense as we can liken this to the issue with head on approaches of cabinets in the original environment that the robot was trained it. Some diagonal approaches are also problematic. Since the robot does not make enough of a wide turn, it does not see the edge of a wood lining for a door and bumps into it. As in a previous discussion, this can be mitigated by tweaking the turning angle so that smaller turns are made.

Another environment that was tested out was the colony space in the laboratory. This is an 8 x 8 foot area where wooden blocks of 1 x 1 foot dimension can be rearranged to create paths. The robot was generally successful in avoiding these blocks, but *only* if the blocks were doubly stacked on each other or if the blocks were placed horizontally. One hypothesis for this behavior is that most of the avoidance of similar colored and textured objects from the original dataset had objects that were large (cabinets) or high enough for the robot to detect ahead of time (border of colony space).

The third environment that the robot was introduced into was a completely different room. This room does not have cabinets like the original room and the chairs are different. The robot successfully avoided all chairs and tables. It was also able to go under tables and avoid objects there, where lighting conditions were much darker than in any image in the dataset. The fact that the robot was able to avoid chairs with different designs in the new room is yet another confirmation that the deep neural network has abstracted the concept of obstacle avoidance. Perhaps what is more important for the network is the concept of going to an open area, which might focus on the color and texture of the carpet. The carpets in the two rooms are very similar.

Another interesting scenario presented to the robot was that of obstacle avoidance with humans, or specifically human legs. Recall that Gritti et al. approached this same exact problem scenario, or at least to the point of detecting legs. We have shown that Deep Learning can be used to have a robot navigate around human legs. This is surely a fascinating finding since nowhere in the original dataset were jeans or human legs introduced. One hypothesis is that the network correlates dark objects with obstacles and the individual in the experiment wore dark pants. However, this is very hard to prove since the inner workings of the network do not easily reveal that information.

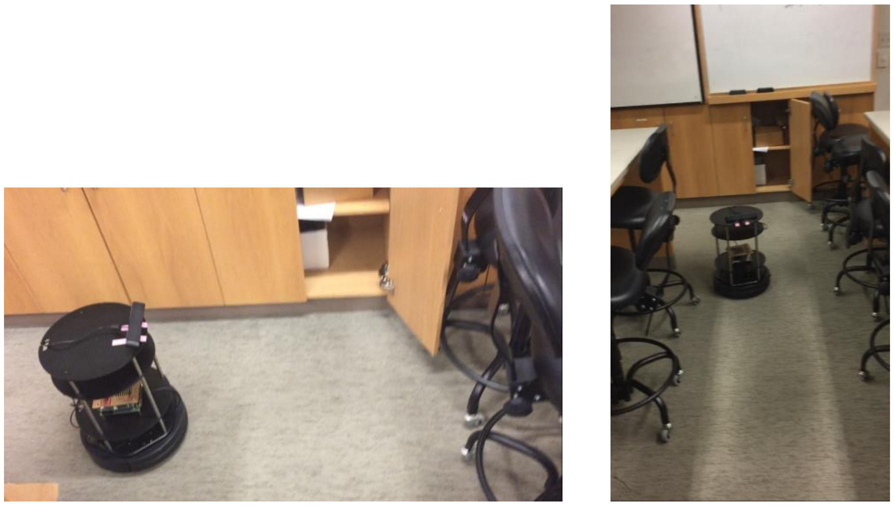


Figure 33. Scenario: detecting an open cabinet; turn right.

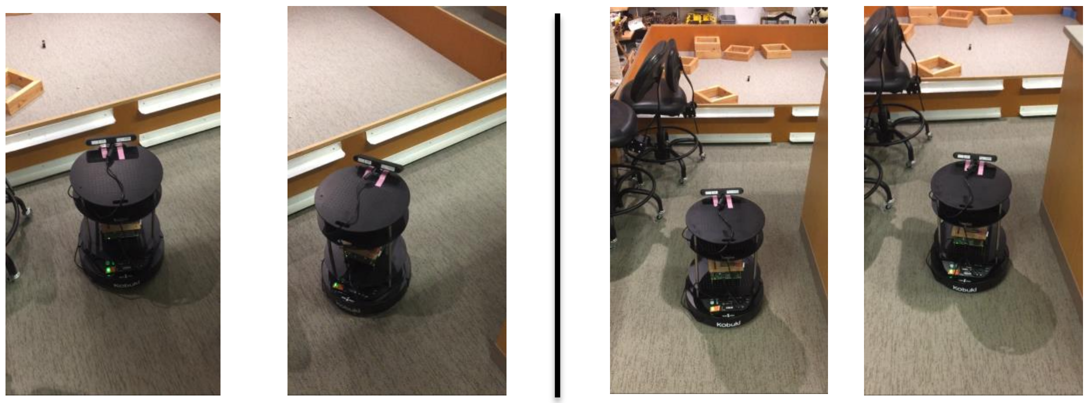


Figure 34. Scenario: detecting border of test colony space.

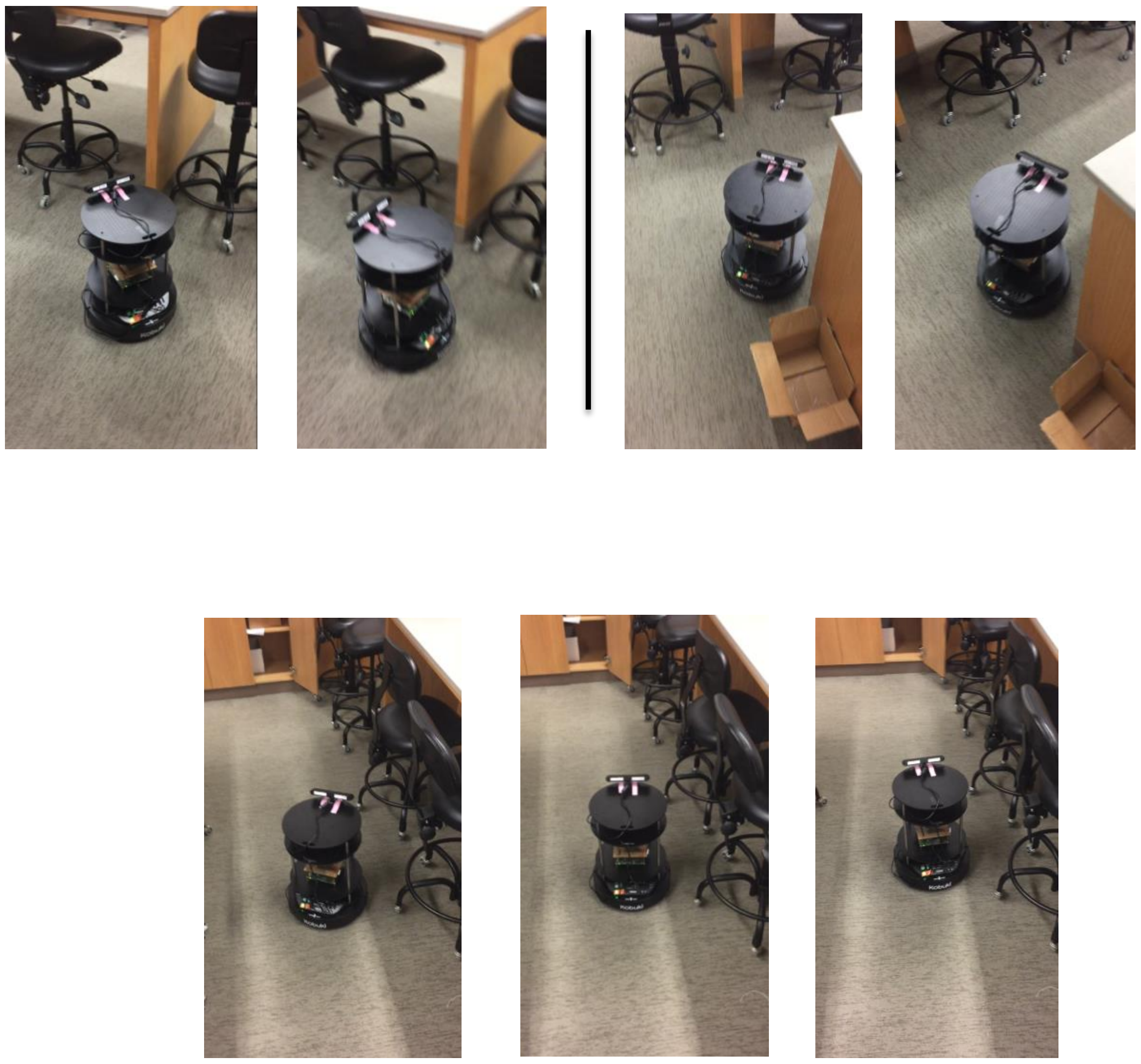


Figure 35. Scenario: chair avoidance

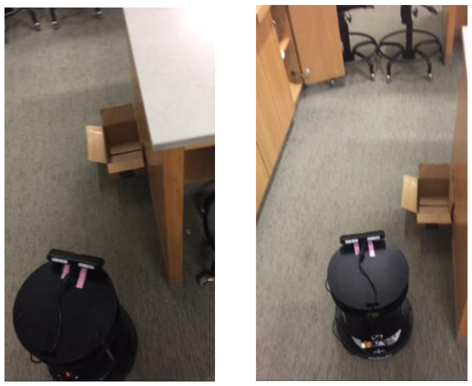


Figure 36. Scenario: cardboard box avoidance

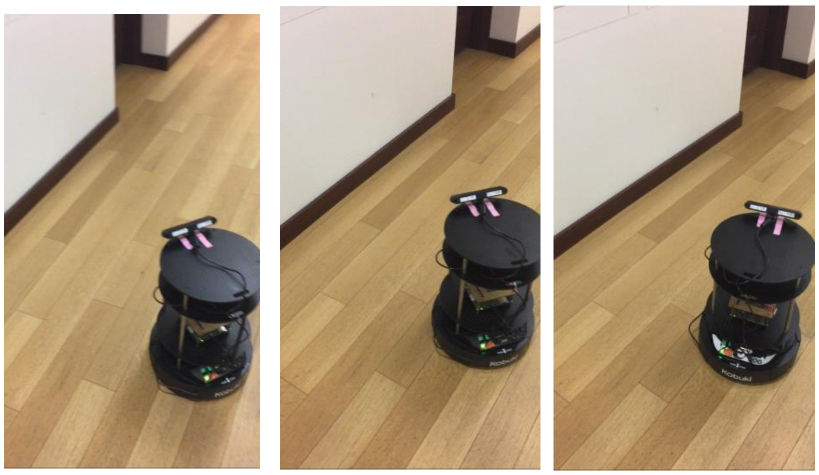


Figure 37. Scenario: new environment; walls in a hallway

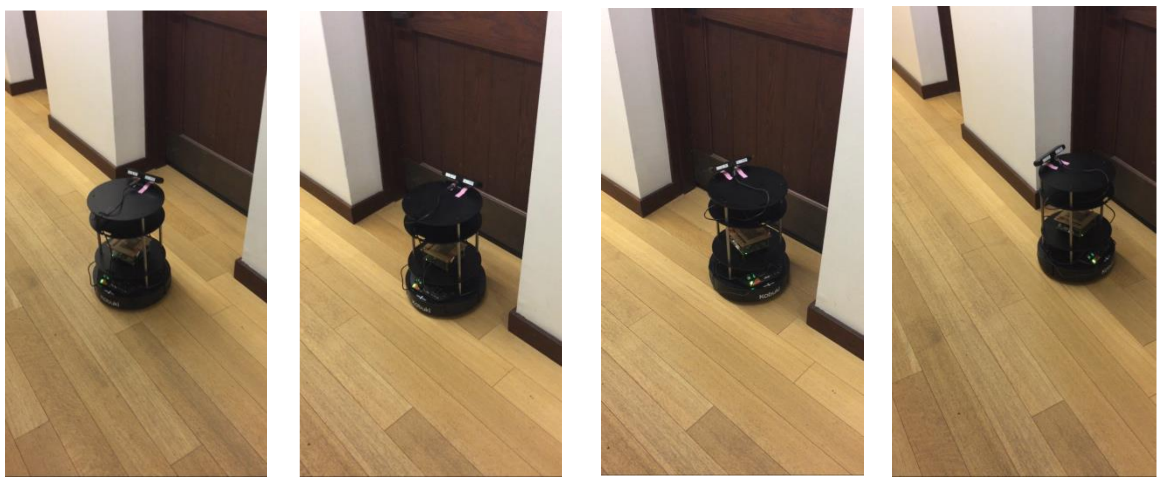
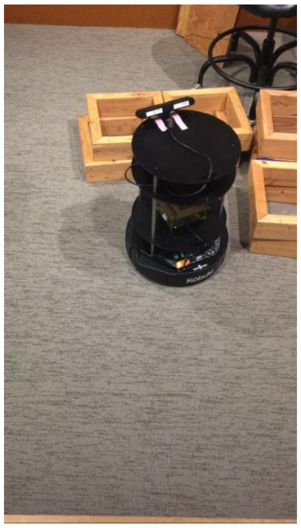
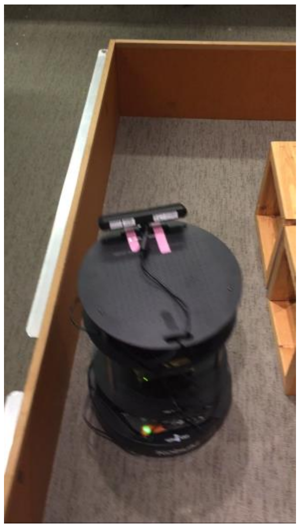


Figure 38. Scenario: new environment; doors in a hallway





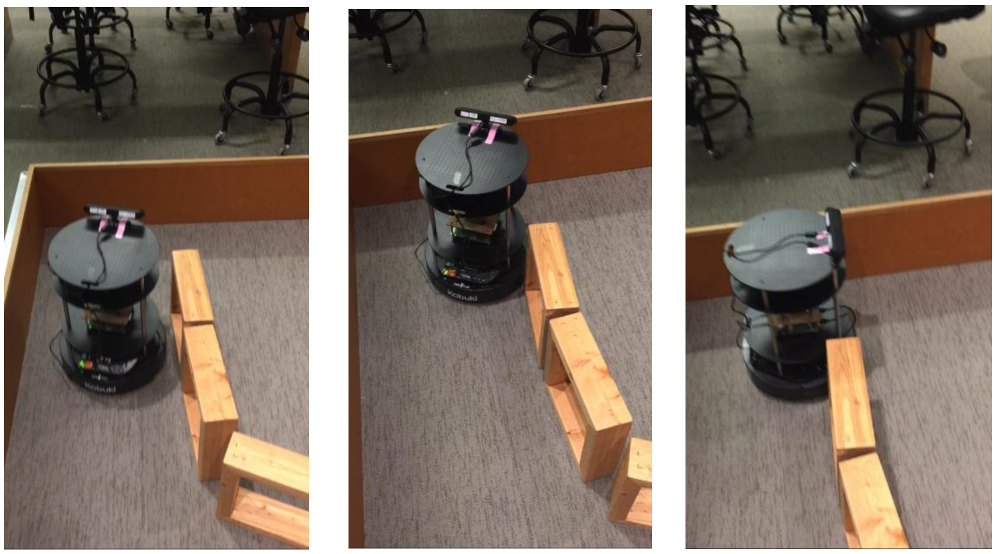


Figure 39. Scenario: new environment; obstacle avoidance with wooden blocks

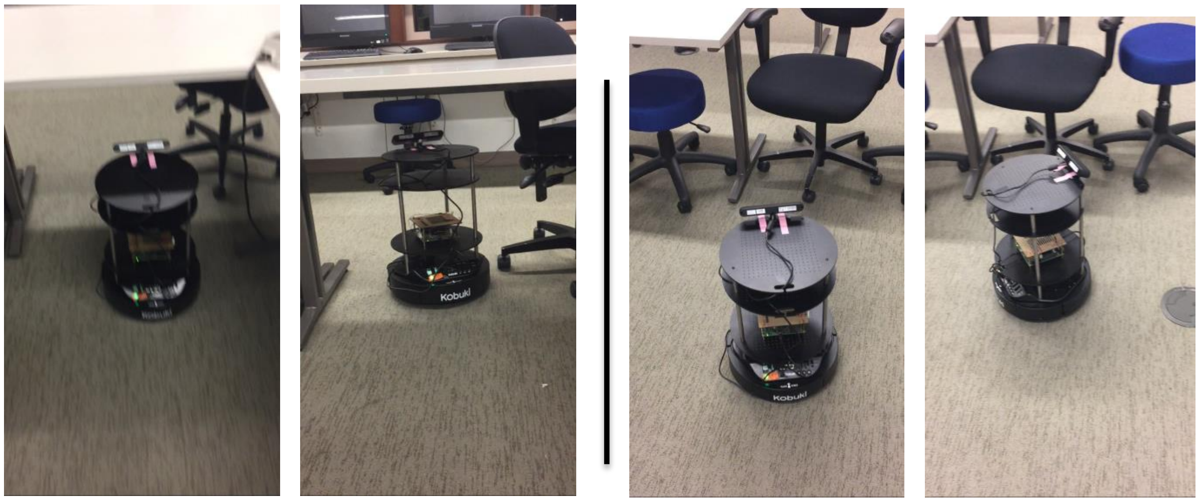


Figure 40. Scenario: new environment; obstacle avoidance with chairs and tables in new room

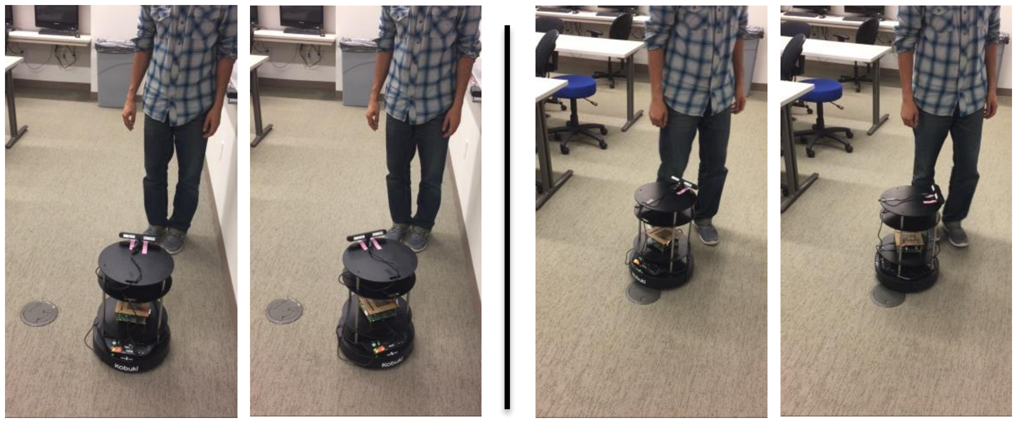


Figure 41. Scenario: obstacle avoidance with human legs as obstacles

Examining Network Weights and Activations:

While it is not trivial to understand what type of logic or pattern the network has learned, we can surely peek into the network and see how the weights and activations look. We might be able to pick out a few features that seem important to the development for the network that can not only differentiate in its lower levels between 10 different categories of animals and objects, but also differentiate between obstacle and non-obstacle. The first convolution layer kernels are shown in Figure 42 below, we can immediately see that some of these kernels are edge detectors while others resemble some form of blurring or sharpening. It is harder to discern exactly what the kernels of the second and third convolution layers are doing in Figure 43 and 44 (respectively), but nonetheless it is important to realize that the deeper the layers get into the network the more complex the overall function becomes for differentiating between objects. This complexity allows for the general pattern recognition model that the network builds through training.

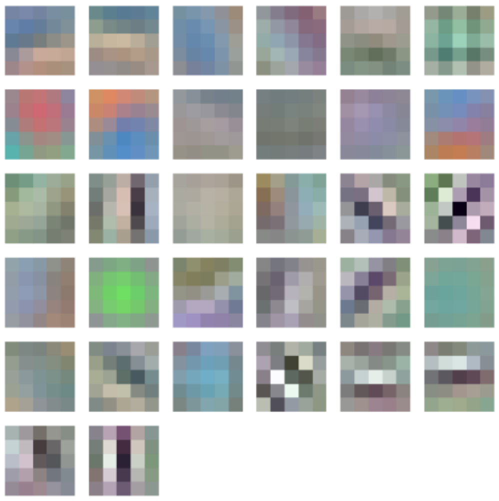


Figure 42. First convolution layer kernels

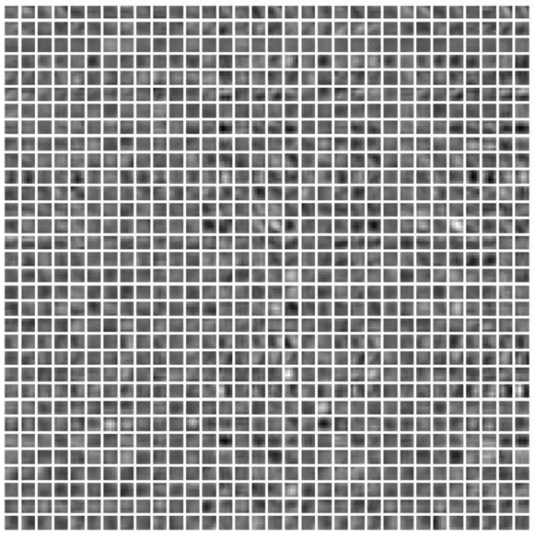


Figure 43. Second convolution layer kernels

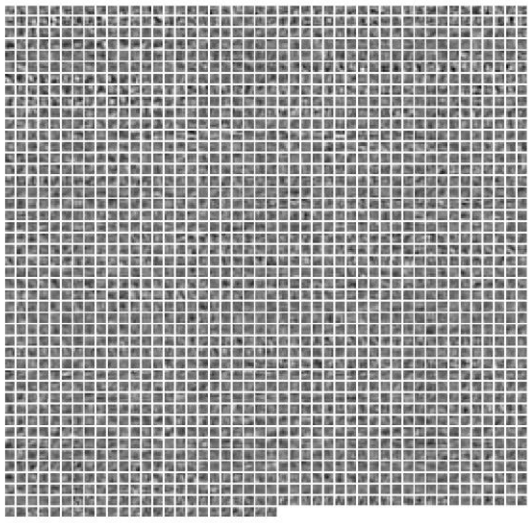


Figure 44. Third convolution layer kernels

The activations of the network with these kernels are also of particular interest to us. Below we can see the image activations in all the different layers of the network (Figures 46, 47, 48). The validation image provided to the network in this case is with a chair to the right of the robot and a cabinet straight ahead.

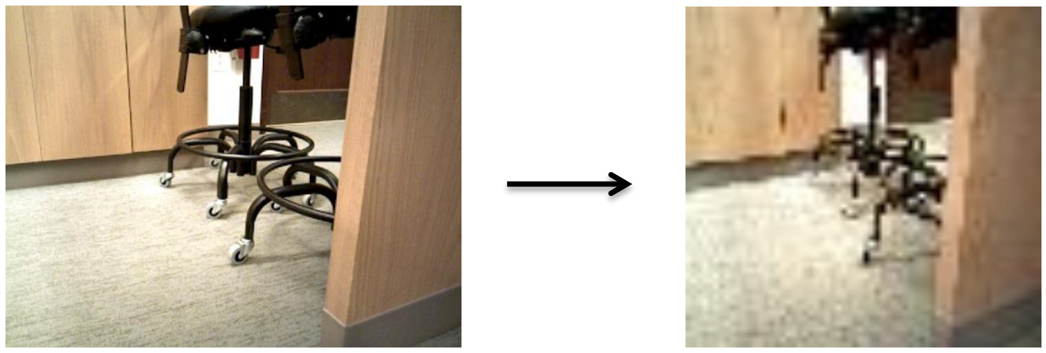


Figure 45. Image provided to network to demonstrate activation values. Recall that the network takes 64 x 64 images. An enlarged version of the downsampled image is shown.

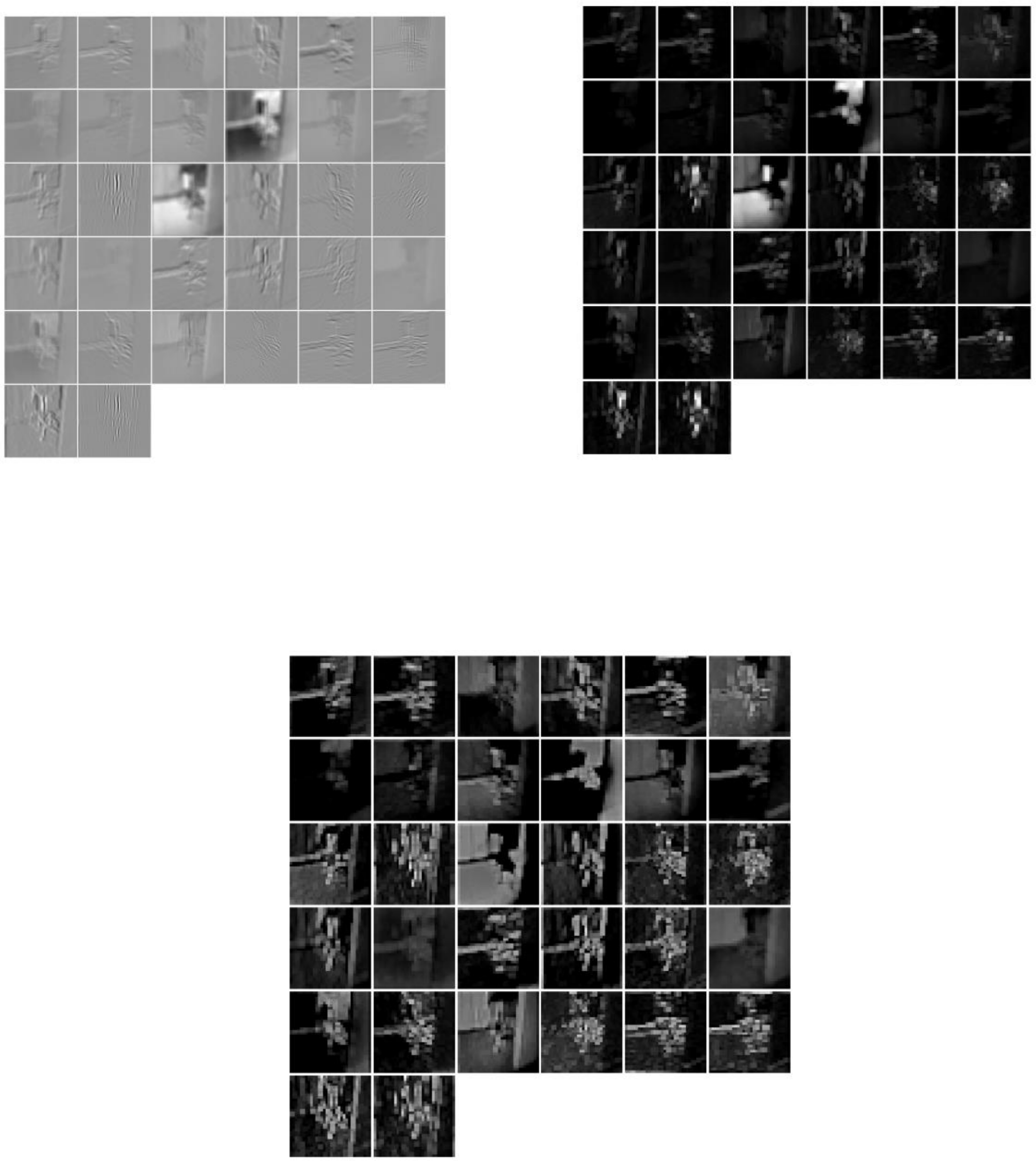


Figure 46. Activations are shown here, respectively, for the first convolution layer, the first pooling layer, and the first normalization layer

We immediately identify that the chair is highlighted in the pooling layer, and the normalization layer augments its location even more. Another observation is that the carpet and chair are shown as contrastive in a couple of the kernels. The cabinet does not light up too much, however there are several kernels where it is next in line after the chair in terms of brightness level.

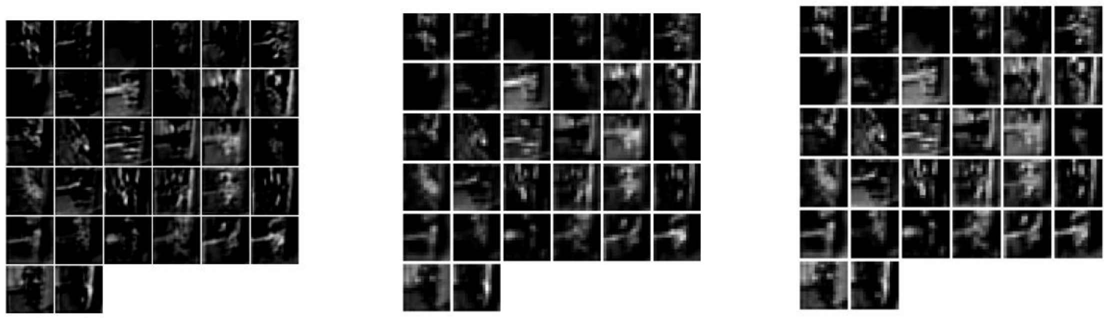


Figure 47. Activations are shown here, respectively, for the second convolution layer, the second pooling layer, and the second normalization layer

The second iteration of the same types of layers does not readily reveal much information about obstacle detection. It now seems that different parts of the image light up. This might mean that the network expects the object to show up in different locations of the image. In a way this makes the network translation invariant. The bottom edge of the cabinet and the edge of the table to the right of the chair also seems to be highlighted even more. At this point it seems as if the network is focusing more on objects in the vicinity of the chair than the chair itself. In fact, in some images it looks as if the chair and the bottom edge of the cabinet are linked together.

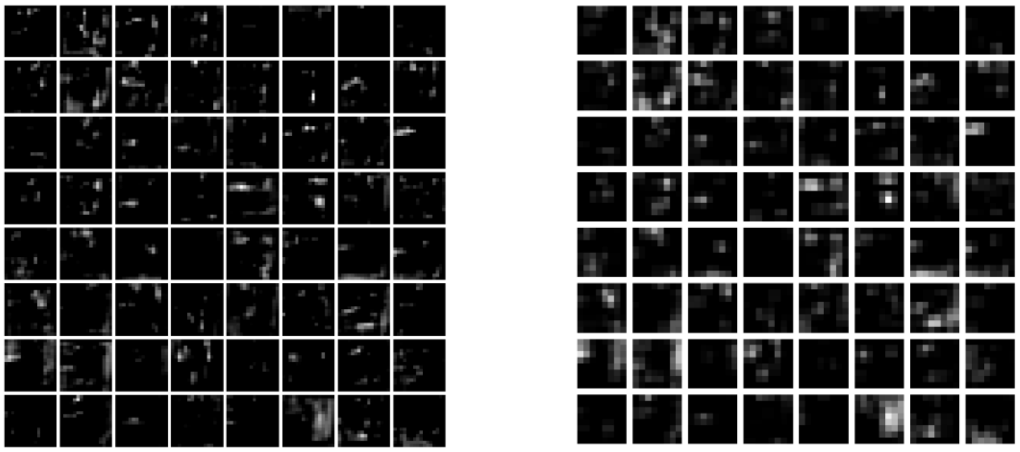


Figure 48. Activations are shown here, respectively, for the third and final convolution layer, and the third and final pooling layer

Similarly to the previous findings, there is not much readily available to notice here. There are some fairly high level features that the network picked up. It is hard to understand what exactly these features may be. It also does seem like almost all parts of the image light up, this may mean that the network is able to focus not on a single location in the image but more so the whole image in working towards developing a holistic understanding of the class of object. The last layer of the network is harder to visualize as it is a fully connected neural network, but we can certainly see the probabilities of the activation for each "class" or driving decision the robot could make. We see below in Figure 49 that the network decides that this particular scenario is one where it is about 30% confident the robot should go straight and about 70% confident that it should turn left (recall that 0 is straight, 1 is right, and 2 is left).

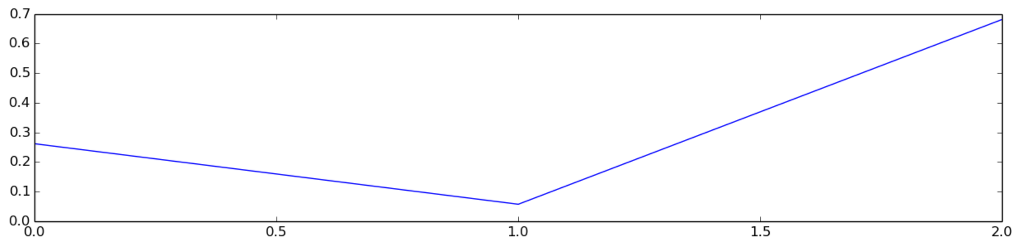
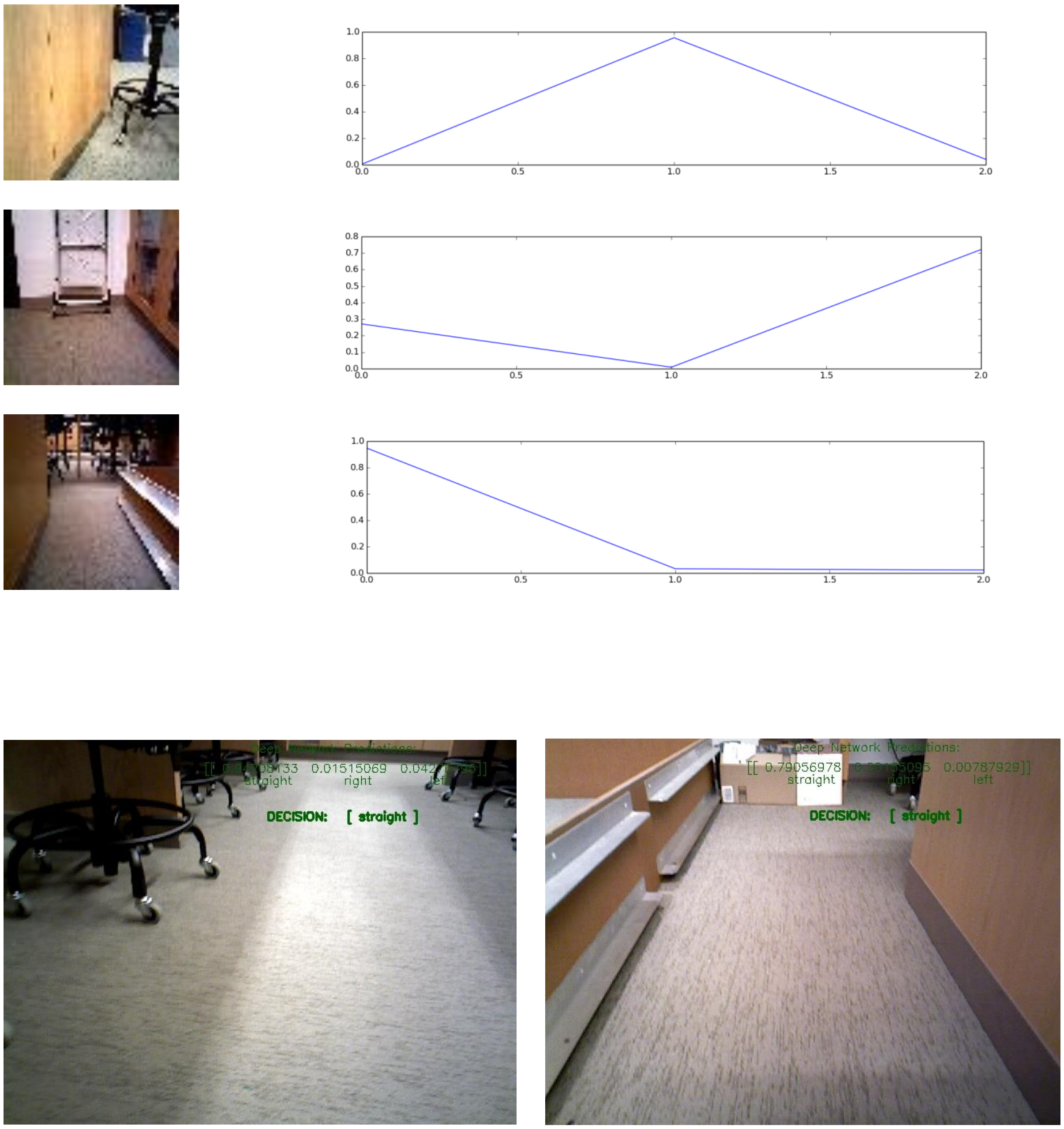


Figure 49. Final layer output predictions and probabilities for each class

While it is still hard to determine what exact features the network is picking up, we have hypothesized about the meaning of some of the graphics from looking into the activation of the network. We know for sure that there is some contrastive definition that is built between the chair and the carpet and also between the cabinet and the carpet.

A Sampling of Live Prediction Making:

Below we show the live decision that the neural network made along with certainty values for each decision in Figure 50. It is a solidifying notion in terms of research results to see that in cases where a human may consider multiple decisions, with a significant probability of trying each decision, that the deep network does also.


```
Deep Network Predictions:
[[ 0.[illegible]08133   0.01515069   0.042[illegible]]]
   straight          right          left
DECISION:   [ straight ]
```

```
Deep Network Predictions:
[[ 0.79056978   0.[illegible]5096   0.00787929]]
   straight          right          left
DECISION:   [ straight ]
```

Figure 50. A sampling of scenarios where the neural network made live decisions and the probabilities of each possible decision. Large images are shown for the second half of the examples in order to make the numbers easier to see.

Conclusions for Deep Driving:

The approach of fine-tuning Krizhevsky's network that solved the CIFAR10 dataset was greatly successful. The robot effectively avoided obstacles in the original room that the dataset was collected in. The robot also avoided colliding into other obstacles that were not part of the dataset. This was proven by placing the robot in 3 new environments. This goes to show that this robot has perhaps abstracted the idea of obstacle avoidance, and that the network has not overfit on the original dataset. In simpler terms, the deep network did not solely focus on chairs and cabinets as the only obstacles to avoid. Although, accuracy wise, this approach seems more successful than the approaches that utilized depth in the relevant papers section, it might be worthwhile to understand how the robot behaves with different types of images. In the future, different dimensions may be considered. It would be valuable to potentially find a definable relationship between the image dimension and network accuracy.

Ongoing / Future Work:

While the primary focus of this paper was autonomous driving, we also explored possible applications of Deep Learning with face detection and object mapping. Brief notes are left below for each research area, and it is recommended that further research is done along with more information from other relevant papers.

Real-time Face Recognition:

While there are several algorithms that can detect faces, facial recognition is a harder problem because specific features must be mapped to specific individuals. In a way, a pattern of remembering the individual must be developed. In a simplistic way, it seems like Deep Learning would be able to handle “classifying” a few different human faces based on images. We initially attempted using the same fine-tuning approach for this problem as the one used for obstacle avoidance. While we obtained an accuracy of 100% for 21 different individuals with about 25 to 30 images each, the network was very sensitive to even the slightest shift in the person’s face. All of the images that the network was trained on were strictly frontal images, and included about the same lighting.

In moving away from the previous approach, we found an excellent article by Adam Geitgey. He uses a face recognition library called OpenFace. In the backend, his use of the library utilized Deep Learning [41]. We were able to integrate this library and pair it with images from our TurtleBot. The robot was able to recognize us with a very high probability. However, the probability dipped dramatically on some occasions with the movement of the robot. With time constraints, we were unable to explore more into this. One way to make the system more robust would be to take an average of prediction probabilities over several frames to build a more confident prediction. In terms of training, more images of ourselves could be added. We only trained the network with about 20 images of each person. Our personal images for this research also spanned over 5 years, which makes the problems more complex – in some images the face has a beard, in another a moustache, and in others the face is clean shaved.

One application of a system like this for TurtleBot would be for escorting services for hospital patients or residents of an old age home. Many patients do not have access to handheld, smart devices. A robot like this would be able to detect an individual and guide them to their location based on the individual's schedule and the current time. For example, if a patient is due to pick up medicine in an in-house clinic, the robot can serve as a nurse aide. The robot will be able to remind the person using a personable sound message.

Real-time Object Detection, Discovery, and Mapping:

As we have seen from our previous discussions, large networks are capable of learning to differentiate between thousands of objects. We were able to take a pretrained model that was based on a deep neural network which solved the ImageNet dataset (the first example in related works for Deep Learning) and load it onto the TurtleBot. The model is named CaffeNet after AlexNet and can be found on the Caffe website [42]. While we were unable to experiment in depth due to time constraints, the idea here is that the TurtleBot can drive around, detect objects, and localize itself with some mapping coordinates within a room. Using these coordinates, the detected object can be pinpointed to a specific location in a room. Over time, the TurtleBot can build a relational dataset of images of items detected (only with high prediction probability) along with their specific coordinates, and maybe even a time stamp. Not only would one be able to track the movement of specific items in a room over time, but one can also look into the database, which would supposedly get updated often, to find a specific item. An application would be to see if a lost item was found by the TurtleBot. In our test runs, the network successfully detected many objects in the laboratory including, but not limited to shoes, chairs, window shades, computer monitors, and machines.

Conclusion:

We presented two areas of research within the field of ANNs. A hardware neural network was built with the capability of solving logic operations and the capability of self-driving a model car. The communication system developed is capable of expanding as far as the number of neurons in the hidden layer. In the future, a framework like this would be able to expand into multiple hidden layers. We also presented a strategic approach to using the power of Deep Learning to abstract a control program for a mobile robot. The robot successfully learned to avoid obstacles not only in its original area of training, but also in 3 other environments it had never been exposed to before. In looking forward at more modes of research, we recommended two different applications of Deep Learning to the TurtleBot platform. One application would be able to recognize and assist individuals based solely on facial recognition and scheduling. The other application would allow the capability of identifying various objects in rooms and pin pointing them with coordinates based on a map. These coordinates and identifications of the objects would be useful to users who are interested in the positions of objects in their homes and offices.

References:

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

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

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

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

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

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

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

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

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

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

[11] Ienne, P., & Kuhn, G. (1995). Digital systems for neural networks. Digital Signal Processing Technology, 57, 311-45.

[12] Joubert, A., Belhadj, B., Temam, O., & Héliot, R. (2012, June). Hardware spiking neurons design: Analog or digital?. In Neural Networks (IJCNN), The 2012 International Joint Conference on (pp. 1-5). IEEE.

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

[14] Pomerleau, D. A. (1991). Rapidly adapting neural networks for autonomous navigation. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, Advances in Neural Information Processing Systems 3, pages 429–435, San Mateo. Morgan Kaufmann.

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

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

[17] Ng, Andrew et al. Unsupervised Feature Learning and Deep Learning (UFLDL) online tutorial and notes. Computer Science Department, Stanford University. http://ufldl.stanford.edu/tutorial

[18] Li, Fei-Fei and Andrej, Karpathy. (2017). CS231n: Convolutional Neural Networks for Visual Recognition. Online course notes and syllabus. Computer Science Department, Stanford University. http://cs231n.github.io/

[19] Krizhevsky, A, Sutskever, I. & Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. Neural Information Processing Systems (NIPS).

[20] Joshi, P. V. (2016). Perpetual enigma: perennial fascination with all things tech. Online blog. https://prateekvjoshi.com/2016/04/05/what-is-local-response-normalization-in-convolutional-neural-networks/

[21] Russakovsky, O., Deng, J., Su, H. et al. (2015). ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision. 115: 211. doi:10.1007/s11263-015-0816-y

[22] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. CoRR, abs/1409.4842

[23] He, K. , Zhang, X., Ren, S. & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In International Conference on Computer Vision.

[24] Krizhevsky, A. (2009). Learning multiple layers of features from Tiny Images. Master's thesis, Department of Computer Science, University of Toronto.

[25] Krizhevsky, A. (2010). Convolutional Deep Belief Networks on CIFAR-10. Unpublished manuscript.

[26] Krizhevsky, A. CIFAR10 Dataset Project Page. https://www.cs.toronto.edu/~kriz/cifar.html

[27] Springenberg, J.T., Dosovitskiy, A., Brox, T., Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. arXiv:1412.6806

[28] Mishkin, D. & Matas, J. (2016). All you need is a good init. International Conference on Learning Representations.

[29] Graham, B. (2014). Fractional max-pooling. CoRR, abs/1412.6071. http://arxiv.org/abs/1412.6071.

[30] Jia, Y. (2013). Caffe: An open source convolutional architecture for fast feature embedding. Berkeley Artificial Intelligence Research Lab. University of California, Berkeley. http://caffe.berkeleyvision.org/.

[31] Hotze, Wolfgang. (2016). Robotic First Aid: Using a mobile robot to localise and visualise points of interest for first aid. Master's Thesis. Embedded and Intelligent Systems, Hamstad University.

[32] Correa, M. , Hermosilla, G., Verschae, R. & Ruiz-del-Solar, J. (2012). Human Detection and Identification by Robots Using Thermal and Visual Information in Domestic Environments. Journal of Intelligent Robot Systems. 66:223–243. DOI 10.1007/s10846-011-9612-2

[33] Boucher, S. (2012). Obstacle Detection and Avoidance Using TurtleBot Platform and XBox Kinect. Research Assistantship Report. Department of Computer Science, Rochester Institute of Technology.

[34] Tai, L., Li, S. & Liu, M. (2016). A deep-network solution towards model-less obstacle avoidence. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).

[35] Tai, L. & Liu, M. (2016). A robot exploration strategy based on q-learning network. IEEE International Conference on Real-time Computing and Robotics (RCAR)

[36] Deep Learning Robot. Produced and sold by Autonomous AI. https://www.autonomous.ai/deep-learning-robot

[37] ROS, Robot Operating System. Ros.org. http://wiki.ros.org/ROS/Introduction

[38] OpenAI. https://openai.com/about/

[39] Leswing, Kif. (2016). "Apple completely changed how Siri works and almost nobody noticed". Business Insider. http://www.businessinsider.com/apples-siri-using-neural-networks-2016-8

[40] Mannes, John. (2017). "Facebook's AI unlocks the ability to search photos by what's in them". https://techcrunch.com/2017/02/02/facebooks-ai-unlocks-the-ability-to-search-photos-by-whats-in-them/

[41] Geitgey, Adam. (2016). "Machine Learning is Fun! Part 4: Modern Face Recognition with Deep Learning". Medium. https://medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-face-recognition-with-deep-learning-c3cffc121d78

[42] CaffeNet – an imitation of AlexNet. Caffe. https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet

[43] Gritti, A. P., Tarabini, O., Guzzi, J., Caro, G. A. D., Cagliotti, V., Gambardella, L. M. & Giusti, A. (2014). Kinect-based people detection and tracking from small-footprint ground robot. International Conference on Intelligent Robots and Systems (IROS).