# Neural Network Modeling of Embodied LEGO Robots

By: Greg Knoblauch
University of Alberta
PSYCO 457
Dr. Dawson
December 6th 2016

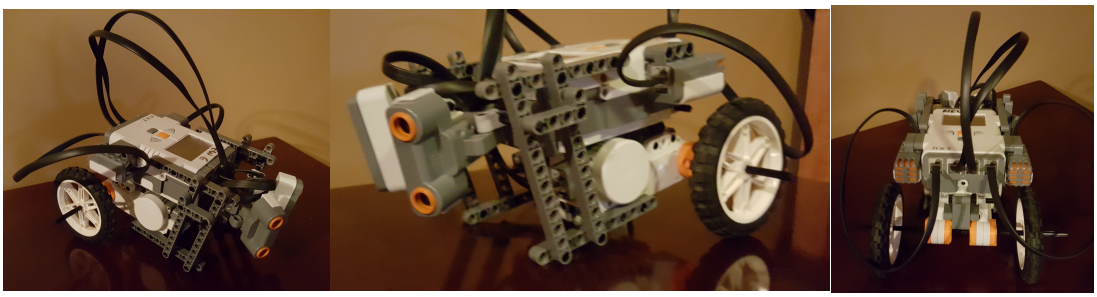
# 0 Introduction

From simple robotics emerges complicated behaviour due to a robot's embodiment and the nature of its surroundings. Embodied robots have always been marveled for their simple design and complex behaviour. Throughout the discipline, emphasis is placed on sensory and motor function for successful interaction with the environment (Wilson, 2002). Lego Mindstorm robots have been a popular medium across computer science and psychology disciples to provide a hands-on experience with simple robotics. To understand in detail embodied cognitive science, it is important to experience constructing systems from interesting components and then observing the actions to see what behaviour is elicited (Dawson, 2010). Shifting the frame of reference assigns more credit to the environment and less to the being itself. Constructing these robots from a bag of parts is called bricolage, an approach that is useful for describing the resulting behavior of robots because we know how it was built.

There is often opposing views to embodied cognition, one prominent one is connectionism. Connectionism aims to develop biologically plausible systems for information processing (Dawson, 2013). Models of connectionism often consist of neural networks, which are a large connection of neural units connected by axons. Neural networks have been described as universal approximators that can learn any task (Dawson, 2013). It is this claim which sparked my interest in applying this perspective to an embodied cognitive science domain. If a neural network can theoretically solve anything, the how does it solve and describe sense-act Lego Robots? I set out to explore if connectionisms' neural networks can explain a robot's behaviour with greater detail than an embodied perspective.

A single layer perceptron will be constructed and trained from a robot that is built using a synthetic approach. Differences in behaviour and situatedness with be evaluated and compared across the two disciplines of cognitive science. It is at this point where embodied cognitive science shows its true benefit in describing complex systems. From building from the ground up, we can build a behavior-based robot that is built upon simple mechanisms that connect perception and action directly with no representation intermediates (Shapiro, 2011). Overall, the purpose of this paper is to examine the complexity, benefits, drawbacks and implications of using a neural network to describe the behavior of a LEGO robot.

# 1 Design and Construction of Dexter



## Overall Design

When setting out to construct my robot, I set out to build something with an industrial design, bulky appearance and a lower center of gravity. What I ended up with is shown above, Dexter. One of the first things apparent with Dexter is that he has no front wheels. These were removed to enhance the robot's ability to turn and navigate around obstacles. The wheels in the front often limited its ability to react swiftly to an obstacle and thus, it ran into walls and object consistently. While no front wheel poses a problem for rough terrain, its intended environment is smooth floors such as hardwood, cement or linoleum. Dexter embodiment handcuffs him to an

environment with smooth floors. His two large wheels on the back are there to provide grip and stability and are driven individually by two motors. Throughout the construction of the robot, several types of wheels were tried. None provided the traction or precise turning radius as well as the ones shown above. Dexter has four sensors and two high torque motors. The motors turn each wheel individually giving it the ability adjust its direction by simply moving one motor faster than the other. The motors are incorporated into Dexter in a backwards position, so all inputs sent to it but be negative. To account for this, all inputs send to the motors were multiplied by negative one. The NXT brick is the largest component of Dexter and the brains to make him move. The Lego NXT Brick is best described as, "a battery-powered module based on a 32-bit ARM micro-controller with an LCD display, four input ports for sensors, and three input/output servo-motor ports. In addition, communication with a computer is possible through Bluetooth or USB cable" (Gomez-de-Gabriel, 2011). The brick is also the heaviest component of the robot, so it is positioned over the rear axle to provide traction. Overall, the construction of Dexter was messy, sporadic and unplanned. There are several things about Dexter that could be altered to improve him. Long axles, wobbly mounts and messy wires are some items in a long list of improvements, but that is part of bricolage and the synthetic approach.

## Sensors

Two ultrasonic sensors are mounted on the front of the vehicle. These sensors provide Dexter with the ability to sense his surroundings. Ultrasonic sensors project high frequency sound waves outwards, wait for them to return and based on the frequency and time in which they return, evaluate how far away objects are. Dexter uses this information to turn away from objects that are close to him and move fast when no objects are around. The orientation of the

ultrasonic sensors is pointed in a direction so his field of view is limited to about 30 degrees in each direction from the front of the vehicle. The ultrasonic sensor values range from 0-255. These values are representative of the distance, in centimeters, away from an object, with precision of about 3 cm (CEEO, 2013). The input is read by Dexter and his motor speeds are adjusted to turn away from objects that are close to him. Obstacle avoidance is a caused by this ultrasonic embodiment.

In addition, there are two microphones mounted on the rear of Dexter. These sensors are used to determine the decibel level of the environment. The NXT sensors give you the option to measure sound in decibels (dB) or adjusted decibels (dBA). Adjusted decibels are adapted to the sensitively of the human ear, while decibels are not (CEEO, 2013). I chose to measure sound in raw decibels, because I did not want any filtering from the sensor. The output of these sensors is normalized in percentage form, 100 being the loudest the microphone can register, or about 90 dB, and 0 being no input to the microphone at all (CEEO, 2013). The use of two microphones allows the vehicle to spatially recognize where the sound is coming from. In Dexter's case, he is wired to fear loud noises. If the sound input is louder on one side than the other, he will adjust his motor speed accordingly to turn away from the loud noise.

## NXC Code

The code designed to run Dexter was originally from Anti-Slam, a robot depicted in the book, *Bricks to Brains* by Dawson, Wilson and Dupius. However much of the complexity that Anti-Slam elicited was removed so it would be easier for a neural network to learn the behaviour. The code that drove Dexter can be found in the appendix. Dexter's code was designed in a way that 40 percent of the power to the motor was dedicated to the microphones. This means

that at full volume the microphones could only drive the motors at 40 percent of their capacity. The ultrasonic sensors accounted for the other 60 percent of the motor capacity. If there are not obstacles within 255 cm of the sensor, Dexter motors would drive at 60 recent capacity. The two inputs are added together to calculate the motor speed. This means that the motors would operate at full speed if the environment was approximately 90 decibels and there are not objects anywhere near the vehicle. In contrast the motors would not spin at all if there were objects immediately in front of Dexter in a silent room.

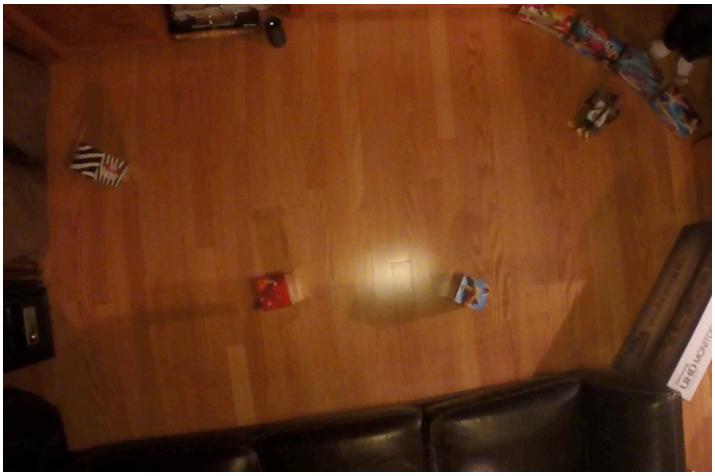
$$left_{speed} = \left(\frac{left_{ultrasonic}}{255}\right) * 0.6 + \frac{left_{microphone}}{100}$$

$$right_{speed} = \left(\frac{right_{ultrasonic}}{255}\right) * 0.4 + \frac{right_{microphone}}{100}$$

In the past, avoidance procedures have been implanted to spin and reorient the robot if the vehicle had not moved in a while. This was removed from Dexter for simplicity. Additional functionality was added to the code that is not apparent in the vehicles behaviour. All 4 sensor readings are recorded along with the motor speed signal. These numbers are logged every 2.5 seconds onto a file in the NXT brick and extracted later to be used in neural network learning.

## Testing Environment

Dexter was placed in three different types of testing environments. All conditions were on clean, smooth hardwood floors. The area was enclosed with various objects including pillows, boxes and furniture. Obstacles and sound stimuli were added to the environment at different times so the correct stimuli could be associated to a behaviour. Dexter was run for approximately three minutes in each location, or until an accurate representation of the behaviour elicited could be established.



| Environment | Obstacles? | Sound? |
|---|---|---|
| # 1 | No | Yes |
| # 2 | Yes | No |
| # 3 | Yes | Yes |

## 2. Observations of Dexter

When observing Dexter in action three noteworthy observations were can be made. First off, he does a good job at avoiding obstacles, rarely running into objects. If Dexter happens to run into an obstacle it seems to be under two different conditions: the object is directly in front of him or it is a soft object. After several times of driving straight into an object it was evident that Dexter does not correctly sense what is directly in front of him. Pillows were a common object in his environment and it was apparent very quickly that he was not sensing the softer objects correctly. But overall, on hard surfaces, Dexter quickly navigates through mazes of tissue boxes and walls with no problems.

Secondly, Dexter also seemed to respond more to obstacle avoidance than loud sound avoidance. There were times in a tight area in front of a speaker and beside an obstacle where Dexter would avoid the obstacle by going close to the speaker. Dexter felt more comfortable with loud noises than hitting objects.

Thirdly, if a loud noise is present on only one side of the vehicle, Dexter appears startled and rapidly turns away from the noise. One interesting observation was the magnitude of the

sound need to scare Dexter. He seemed to have a tolerance for medium loud noises, which would not result in a change in motor speed. However, when a loud sound was picked up by the microphone he would adjust his motor speeds accordingly. In conclusion, Dexter behaved almost exactly the way I assumed he would since I built and programmed him.

## 3. Interpretation of Dexter's Behaviour

The first behaviour of interest is the seemingly lack in soft object detection. This can easily be explained after examining the means in which the ultrasonic sensor calculates the distance of an object. Pillows and pant legs absorb the high frequency sound waves the sensor emits resulting in the sensor receiving very few waves back, which is a similar result when there are no obstacles. This is a consequence of using this type of sensor.

The second behaviour of interest is that objects directly in front of Dexter also seemed to be ignored. The orientation of the ultrasonic sensor is to blame for this, as they don't point to detect objects in front. A change in embodiment could resolve this, like an ultrasonic sensor or a wider field of view.

Dexter seems to respond more to obstacles that sound. This behaviour can be described by the percentage of motor speed that is assigned to each sensor. Ultrasonic sensor utilizes 60 percent of the motor speed, while the microphone the remaining 40. In some cases, the ultrasonic can overpower or neutralize sound input because it is responsible for a larger chunk of the motor input. Lastly, interpreting Dexter's tendency to be unaffected by medium ambient noise in a room seemed unexplainable by the code. However, if we examine the location of the microphones, it becomes quite clear why Dexter is unaffected by medium ambient noise. Dexter's microphones are located directly above his motors. Sound from the motor spinning was

being picked up and transmitted into input. This explains why, the ignorance to lower noises levels. His microphones were consistently reading 34 percent as a direct result from the motor noise. This was a behaviour that was not intentionally programmed but we got it "for free".

It is also important to interpret the behaviour as a naïve observer; Dexter seems to have a personality of enjoying music and eliciting excitement over ambient music but frightened by loud sounds in one "ear". It is easy to make a comparison to Dexter's behaviour to an individual at a concert. Overall an individual is excited at a concert, the loud music elicits rapid body movements such as dancing. However, if an individual is placed directly in front of a concert speaker they often will turn away and find another place to listen. If we personify Dexter's behaviour it is easy to imagine the incentive for his behaviour. This is often what happens when an observer is not aware of how it works, its behaviours are interpreted as more complex than they are.

## 4. Training a Neural Network to Replicate Dexter's Behaviour

After building, programming and recording Dexter's behavior I set out to train a neural network to replicate Dexter's actions. This reincarnation of Dexter will be referred to as "TF. Dexter" because he runs on a machine learning framework called Tensorflow.

### Framework

There are tons of open source machine learning frameworks published such as Tensorflow, Theano, CNTK, Torch and Caffe (Tran, 2016). These frameworks have the capability and capacity to train a network as simple as the one needed to learn Dexter's behavior. Tensorflow has more recently been in the spotlight for machine learning. Google's AlphaGo had

a recent success in defeating the grandmaster Lee Sodel in the game of go (Borowiec, 2016). It is unclear if the open source Tensorflow was used In AlphaGo, nevertheless, Google built the Tensorflow interface and the interface implementation (Abadi, 2016). Therefore, I chose to use Tensorflow because of its popularity, recent success, its python interface and easy model deployment.

## Data

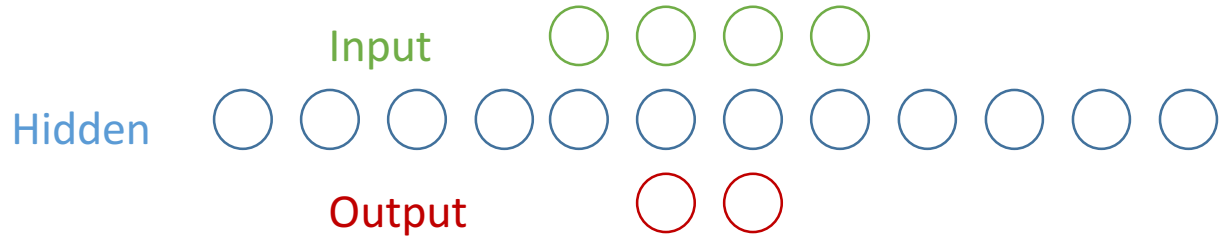
The data used to train the network was collected during the observation of Dexter's behavior. Collection of these log files was a tedious task due to the to 256kb of memory storage on the NXT brick. Logs files had to be extracted every minute, or the file would become too large to collect any more data. Over the course of many log dumps, I collected 1150 log entries of sensor inputs and motor outputs. The format given to Tensorflow was a csv file and can be found in my Github repository.

| Sensor Inputs | | | | Motor Outputs | |
|---|---|---|---|---|---|
| Left Microphone | Right Microphone | Left Ultrasonic | Right Ultrasonic | Left Motor Speed | Right Motor Speed |

## Network Construction

Relative to the problems that the application of Tensorflow can solve, my problem is trivial. Which is why the approach I took to construct my network was as simple as possible. Many problems are solved by using deep neural networks or "Deep learning". The network used to train TF. Dexter is far from deep, it is only three layers. A single layer perceptron seemed appropriate for this learning task as Dr. Dawson states, "There is no restrictions on the number of hidden units in the network and if there are not restrictions of the connection weights, then in principle a network can be created to approximate –over a finite interval – any continuous

mathematic function to an arbitrary degree of precision" (Dawson, 2005). This type of network has been used in robotics research in the past. Single-layer feedback neural networks are suitable for processing path control level of robots (King, 1989). After multiple configurations, the appropriate number of hidden units seem be 10. Successful learning was possible with less units, however, 10 consistently got good results.




Durability and replicability was favored over tedious and time consuming learning of a smaller network. Overall, this architecture is proven to be effective at solving these types of problems, and as we will see, successfully learned Dexter's behavior.

## Training

Training was conducted on all 1000 log entries and validated on 150 in batches of 16. A batch size of 16 was used to stabilize learning and more accurately estimate the gradient. Loss was calculated using a function called l2_loss. This function take the expected output subtracts if from the networks prediction, raises that result to the power of 2 and divides by 2. L2_loss is expressed in the equation below:

$$loss = \frac{\left(expected_{output} - predicted_{output}\right)^2}{2}$$

The Adam algorithm was used to optimize the network. Adam is best described by the author, "A method for efficient stochastic optimization that only requires first order gradients" (Kingma, 2015). Combing through Tensorflow documentation and examples shows that this is a

very popular optimizer and was effective in learning Dexter's behavior. There are many parameters to the Adam optimizer, but they were all left at their default values.

A learning rate of 0.001 was used over 100000 iterations to successfully train the network resulting in loss of approximately 0.5. The weights of the network were then saved to a file so that they could be loaded into a network later. Below are the key metrics used to train the network:

- Loss Function = L2 Loss
- Random initialization with mean of 0.1
- Number of iterations = 100000
- Learning rate = 0.001
- No direct connections between input and output units
- Number of input units = 4
- Number of hidden units = 10
- Number of output units = 2
- Optimizer = Adam

Using this configuration, the network would converge and successfully learn Dexter's behavior!

## 5. Running a Neural Network on a NXT Brick

One thing most neural network frameworks have in common, they all interface with python. The hardware on the NXT brick is limited to 256kb of memory and 32kb of ram (Wikipedia, 2016) and it is simply not capable of running a python interpreter or compute any sort of complex computation. A solution to this problem was to run Tensorflow on my MacBook and interact with the NXT brick via a python package called nxt-python via Bluetooth. This seemed like an elegant solution, but there were compatibility issues with MacOS Sierra and the Bluetooth drivers needs to control Dexter. A Windows 10 computer was placed in between my MacBook and the NXT brick to eliminate the Bluetooth connectivity issues. At the time of


experimentation, it was not possible to install Tensorflow on a Windows therefore the resulting infrastructure is illustrated to the left.

A Windows 10 computer reads sensor input from TF. Dexter and sends an http request to a python Flask server running on the MacBook. The MacBook receives the inputs, runs them through the Tensorflow network and replies the results to the Windows computer, which then send the motor speed over Bluetooth to TF. Dexter. The total travel time of the signal is approximately 3 seconds, with the Bluetooth connection taking up most that trip time. This is not an ideal setup however, there are no published works out there that have used Tensorflow on a Lego Robot, so this was an improvisation.

## 6. Observation of TF. Dexter

Environments remained consistent across observations of both iterations of Dexter. Initial observations of TF. Dexter are one thing: slow. Observations of the robot took much longer, as they cover less ground in a certain time frame than the predecessor. Nevertheless, TF. Dexter was observed as learning the same behaviour as Dexter. He navigates through the same environment that Dexter did with ease. Dexter's idiosyncrasies also appeared in the TF. Dexter. The lack of detection of soft objects, poor avoidance to objects in front of him and favoring ultrasonic input over sound input are all apparent in the neural network controlled version of Dexter. TF. Dexter's reaction to sound seems similar in magnitude to before. When a speaker is placed in his environment he avoids the loud noise but overall his motor speed increases just like before. Overall, it was quite impressive how good TF. Dexter replicated the behaviour.

There seemed to be one minor difference and that was in the speed of the motor rotation. With no obstacles in front of him, in a silent room, TF. Dexter seemed to move slower than he predecessor. In conclusion, there were not visible differences in the robot that was controlled by a neural network, other than the lack of speed in the transmission of data and the slower movements in certain cases.

## 7. Interpretation of TF. Dexter's Behaviour

Before observing TF. Dexter, it was apparent there was going to be a delay between sensor input and motor input because was being run on a server rather than locally on the NXT Brick. The signal must be encoded and decoded several times and travel across different means, Bluetooth and http. The resulting choppy behavior resembles the Shakey Robot built in 1971 by Artificial Intelligence Center at SRI (Nilsson, 1984). Shakey had no real on-board computing so he sent signal to a larger computer to do computations much like TF. Dexter does. Infrastructure causes a delay in movement.

Another behavior that didn't quite match with Dexter's was that the actions TF. Dexter took were, on average of less intensity. Reading the neural network output, I determined that this was not a connection issue. This was not a mistake on learning either, the network was accurately predicting the motor speed of the robot, however sound input was on average lower for TF. Dexter than its predecessor. This is curious as this change in behaviors has nothing to do with learning at all, a change in the environment must have occurred for the sensor to be pickup up less input. Yet another disadvantage of slow communication; lower microphone input is because the motors are not spinning when the sensors take a reading. The implementation of TF. Dexter measures the inputs, send that information to the computers, receives motor speed back,

turns the motors then repeats. There is no overlap in time between when the motors are spinning and when the sensors are taking a reading. A lack in overlap results in a change in TF. Dexter's environment that changes his behavior. Previous feedback loops are no longer present due to a change in infrastructure and not embodiment.

## 8. Conclusions

Overall, TF. Dexter and Dexter, elicited the same behavior in the same environments. So, does a robot that has learned behavior and is guided by a neural network provide a greater understanding of how Dexter situatedness in his environment? To analyze the differences between the two underlying structures it is important to look at different levels of analysis.

First off, the inputs and outputs between the two robots are almost identical. There are a few slight differences, but those were caused due to a difference in infrastructure. The similarity in behavior and numerical output was astonishing. If it was possible to run the neural net locally on the NXT brick, the robot's behavior would be indistinguishable. From a purely behavior standpoint there is no differences between connectionism and embodied cognitive science, therefore we can proceed with the analysis of how they both construct this behavior.

One step further down we look at how both vehicles algorithmically, determining the relation between input and output. Dexter, the strictly embodied cognitive science robot, using a basic math function, calculated the result of his motor speed. The approach TF. Dexter used on the other hand is more difficult to determine. It is safe to assume that trained into the weights of the network is the same formula used in Dexter. While this may be decipherable by looking at the weight, it is more difficult to deduce than Dexter's code. It is at this point we already see connectionism becoming increasingly complicated to describe the behavior.

Diving deeper into analysis we look at the basic architecture of embodied verses connectionism. At the root, embodied cognitive science uses a sense-act relationship while connectionism uses sense-think-act. The comparison of these two cycles is quite easy, the sense-act cycle is a neural network without the hidden units, typically referred to as a perceptron. Perceptron's have direct connections from input to output units, "Output units in a perceptron calculate their net input by summing the signal sent to by each input units after the signal has been scaled down by a connection weight" (Dawson, 2005). For a fair comparison, we can infer how a network would look for Dexter, and compare it. Because TF. Dexter used a multilayer perceptron and Dexter used a perceptron, we can draw the conclusion that TF. Dexter is unnecessarily more complex.

"Connectionist approaches to control are instance of memory-intensive approaches" is an example of another disadvantage of using neural networks in this application (Miller, 1990). Memory requirements for a neural network are much larger than an embodied approach. This resulted in a change of infrastructure to accommodate the computationally demanding needs of a neural network. Connectionist takes the approach of placing the complexity of behavior into the mind of the robot, where embodied cognitive science takes that complexity and places it into the environment. This leaky mind hypothesis make behavior analysis easier.

Advantages of controlling a robot like Dexter with a neural network are few and far between. However, if the complexity of Dexter was increased neural networks would be beneficial. First off, networks can pre-compute results and store them in network that allows fast retrieval (Berkley, 2012). This is useful for complex equations that normally take time to compute. Secondly, a neural network can learn a task more generally that a hardcoded embodied machine (Berkley, 2012). Machine and robotic learning is becoming increasingly popular because we are

constantly pushing the domain for what robot can and cannot do. The ability for a machine to learn, accelerates the performance of machines much further than embodied cognitive science can.

Finally, in an examination of two versions of a NXT robot name Dexter, connectionism provide no great insights to describing his behavior. While connectionism has validity and credibility for more complex problems, it does not help describe behavior of a robot that was built using the synthetic approach. Embodied cognition is the best approach taken when working with NXT Lego robots. Its emphasis on the embodiment and environment can explain behaviors better than any other disciple of cognitive science.

## 9. Future Studies and Improvements

I have often brought up the point that the neural network used to control TF. Dexter is more complex than Dexter. While this hold true, it is possible to train Dexter without hidden units. If I were to hand wire a network with four inputs and two inputs to replicate Dexter's behavior, it is achievable as Dexter's actions are a linear relationship to his inputs. However, the industry of machine learning often does not consider the smaller networks that can do the same task. Results, trainability and replicability are often more important than fewer neuron connections. Therefore, I took a more brute approach to solving the problem, but this is like the way they are solved in industry.

In a future project, I would like to used something that a better hardware than the NXT Bricks, its outdate Bluetooth protocol, limited storage and poor CPU made it unusable to run anything other than NXC code on it. A Raspberry Pi or equivalent could be a good replacement for an NXT brick.

# Works Cited

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Wilson, M., & Dupuis, B. (2010). *From bricks to brains: The embodied cognitive science of LEGO robots*. Athabasca University Press

King, S., & Hwang, J. (1989). Neural network architectures for robotic applications. *IEEE Transactions on Robotics and Automation, 5*(5), 641-657. doi:10.1109/70.88082

Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*

Dawson, M. R. (2005). *Connectionism: A hands-on approach*. Oxford, UK: Blackwell Pub.

Nilsson, N. J. (1984). *Shakey the robot*. Menlo Park, CA: SRI International.

Miller, W. Thomas, Sutton, R. S, & Werbos, P. J. (1990). *Neural networks for control.* Cambridge, Mass.: MIT Press.

Gomez-de-Gabriel, J. M., Mandow, A., Fernandez-Lozano, J., & Garcia-Cerezo, A. (2011). Using LEGO NXT Mobile Robots with LabVIEW for Undergraduate Courses on Mechatronics. *IEEE Transactions On Education*, *54*(1), 41-47.

Bekey, G., & Goldberg, K. Y. (Eds.). (2012). *Neural Networks in robotics* (Vol. 202). Springer Science & Business Media**.**

Borowiec, S. (2016). AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol. Retrieved December 06, 2016, from https://www.theguardian.com/technology/2016/mar/15/googles-alphago-seals-4-1-victory-over-grandmaster-lee-sedol

CEEO. (2013, April 12). NXT sensors. Retrieved December 06, 2016, from http://www.legoengineering.com/nxt-sensors/

Tran, K ., (2016). Deepframeworks. Github Repository. Retrieved December 6, 2016, from https://github.com/zer0n/deepframeworks/blob/master/README.md

Dawson, M. R. (2013). *Mind, body, world: Foundations of cognitive science*. Edmonton, AB: AU Press.

Shapiro, L. A. (2011). *Embodied cognition*. New York: Routledge.
Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*

Lego Mindstorms NXT. (n.d.). Retrieved December 06, 2016, from https://en.wikipedia.org/wiki/Lego_Mindstorms_NXT

## NXC Code

```
//Authour: Greg Knoblauch
//NXC Code for the training of a neural network

#define LeftMotor OUT_B
#define RightMotor OUT_C
#define LeftUltra S2
#define RightUltra S3
#define LeftMic S1
#define RightMic S4

int Sensitivity = 1; //Level 0 connection: Ultrasonic sensitivity. Default 1.
int LMic = 0; //Impact of the left microphone
int RMic = 0; //Impact of the right microhone
int ultraSensitivity = 60; // Pecentage of ultra reading that affects the motor speed
int micSensitivity = 40; // Pecentage of mic reading that affects the motor speed
int RightSpeed = 0; // Initalization of value for the speed of the right motor
int LeftSpeed = 0;// Initalization of value for the speed of the left motor

/*=====Level -1": Integration=============================================
Each of the terms (ultraSensitivity, mic) is part of a later
level's connection to the motors. See the main task to see their defaults.
On its own, this task does nothing. However, it will function at every level
without modification.
*/

task Drive(){
  while(true){
    //"Hearing/255" converts from responsive raw ultrasonic to % motor speed.d
    RightSpeed = ((SensorUS(RightUltra)*(ultraSensitivity)/255)+RMic) * (-1);
    LeftSpeed = ((SensorUS(LeftUltra)*(ultraSensitivity)/255)+LMic) * (-1);
  }
}

/*=====Level 0: Drive=====================================================
Feed the distance from each ultrasonic sensor to the motor. The robot is wired
contralaterally, and thus avoids all walls equally. As a result, when it reaches
a corner, it slows down and ends up stopping in the corner "for free".
*/

task DriveRight(){
  while(true){
    OnFwd(RightMotor, RightSpeed);
  }
}

task DriveLeft(){
  while(true){
    OnFwd(LeftMotor, LeftSpeed);
  }
}

/*=====Level 1: Hear======================================================
Reads the decible level from both microphones. Decible devel are read in values 1-100.
They are multiplied by the mic sensitivity value then divided by 100 to get a vale from 0-1.
*/

task Hear(){
  while(true){
    LMic = Sensor(LeftMic)*micSensitivity/100;
    RMic = Sensor(RightMic)*micSensitivity/100;
  }
}

/*=====Level 2: Logging===================================================
Simple Logginf Function to write all inputs and outputs to file.
*/
```

```
byte handle;
long fileSize = 10000;
short bytesWritten;
string write;
string comma = ",";

task Logger(){
  DeleteFile("Example.txt");
  CreateFile("Example.txt", fileSize, handle);
  while(true){
    Wait(250);
    int left_mic = Sensor(LeftMic);
    int right_mic = Sensor(RightMic);
    int left_ultra = SensorUS(LeftUltra);
    int right_ultra = SensorUS(RightUltra);


    string left_eye_str = NumToStr(left_mic);
    string right_eye_str = NumToStr(right_mic);
    string left_ear_str = NumToStr(left_ultra);
    string right_ear_str = NumToStr(right_ultra);
    string left_motor_str = NumToStr(LeftSpeed);
    string right_motor_str = NumToStr(RightSpeed);

    write = StrCat(left_eye_str, comma, right_eye_str, comma, left_ear_str, comma, right_ear_str, comma, left_motor_str, comma,
right_motor_str);
    WriteLnString(handle, write, bytesWritten);
  }
  CloseFile(handle);
}


//=====Main Task=================================================================

task main(){
  //Set up ultrasonic sensors and speed calculation weights.
  SetSensorLowspeed(LeftUltra);
  SetSensorMode(LeftUltra, SENSOR_MODE_RAW);

  SetSensorLowspeed(RightUltra);
  SetSensorMode(RightUltra, SENSOR_MODE_RAW);

  SetSensorType(LeftMic, SENSOR_TYPE_SOUND_DB);
  SetSensorMode(LeftMic, SENSOR_MODE_PERCENT);

  SetSensorType(RightMic, SENSOR_TYPE_SOUND_DB);
  SetSensorMode(RightMic, SENSOR_MODE_PERCENT);

  start Drive;
  start DriveRight;
  start DriveLeft;
  start Hear;
  start Logger;
}
```

## Python Client Code

```
import nxt.locator
from nxt.sensor import *
from nxt.motor import *
import requests
from threading import Thread
import time

b = nxt.locator.find_one_brick()
motor1 = Motor(b, PORT_B)
motor2 = Motor(b, PORT_C)
sound1 = Sound(b, PORT_1)
sound2 = Sound(b, PORT_4)
ultra1 = Ultrasonic(b, PORT_2)
ultra2 = Ultrasonic(b, PORT_3)

def run_motor1(speed):
motor1.turn(float(speed) * -1, 270)

def run_motor2(speed):
motor2.turn(float(speed) * -1, 270)

while True:
time.sleep(3)
data0 = str(sound1.get_sample())
data1 = str(sound2.get_sample())
data2 = str(ultra1.get_sample())
data3 = str(ultra2.get_sample())

all_data = data0 + " " + data1 + " " + data2 + " " + data3
try:
result = requests.post('http://192.168.1.79:3000/tf', data = all_data)
speeds = result.text.split(" ")
print speeds
Thread(target=run_motor1, args=(speeds[0], )).start()
Thread(target=run_motor2, args=(speeds[1], )).start()

except StandardError as e:
print "No connection"
print e
```

## Python Neural Network Training Code:

```
import tensorflow as tf
import numpy as np
import sys
import random
import time

dataFile = "good_data_pos.txt"
df = open(dataFile)

examples = []

for line in df:
    data = line.split(",")
    mic = data[0:2]
    ultra = data[2:4]
    output = data[4:6]
    for i in range(0, len(mic)):
        mic[i] = float(mic[i]) / 100.0
    for i in range(0, len(mic)):
        ultra[i] = float(ultra[i]) / 255.0
    for i in range(0, len(output)):
        output[i] = float(output[i]) / 100.0
    examples.append((mic + ultra, output))


batchSize = 16
hiddenUnits = 10

inputX = tf.placeholder(tf.float32, (batchSize, 4))
outputY = tf.placeholder(tf.float32, (batchSize, 2))

weight = tf.get_variable("mweight", (4, hiddenUnits), initializer=tf.random_normal_initializer(mean=0.1))
bias = tf.get_variable("mbias", (hiddenUnits, ), initializer=tf.random_normal_initializer(mean=0.1))

weight2 = tf.get_variable("mweight2", (hiddenUnits, 2), initializer=tf.random_normal_initializer(mean=0.1))
bias2 = tf.get_variable("mbias2", (2, ), initializer=tf.random_normal_initializer(mean=0.1))

hidden = tf.nn.relu(tf.matmul(inputX, weight) + bias)
output = tf.nn.relu(tf.matmul(hidden, weight2) + bias2)

#loss = tf.reduce_sum(tf.abs(outputY - output)) # works okay
loss = tf.nn.l2_loss(outputY - output)

optimize = tf.train.AdamOptimizer(0.001).minimize(loss)


iterations = 100000
k=0
init = tf.initialize_all_variables()
saver = tf.train.Saver()

with tf.Session() as sess:

    sess.run(init)
    for k in range(iterations):
        batchX = []
        batchY = []
        for i in range(0, batchSize):
            rX, rY = random.choice(examples)
            batchX.append(rX)
            batchY.append(rY)

        rLoss, _ =  sess.run([loss, optimize], feed_dict={inputX: batchX, outputY: batchY})
        print "Loss was {}".format(rLoss)

        if k % 5000 == 0:
            save_path = saver.save(sess, "model.ckpt")
```

## Python Server Code:

```
from flask import Flask, request
import tensorflow as tf

batchSize = 1
inputX = tf.placeholder(tf.float32, (batchSize, 4))
weight = tf.get_variable("mweight", (4, 10), initializer=tf.random_normal_initializer(mean=0.01))
bias = tf.get_variable("mbias", (10, ), initializer=tf.random_normal_initializer(mean=0.01))

weight2 = tf.get_variable("mweight2", (10, 2), initializer=tf.random_normal_initializer(mean=0.01))
bias2 = tf.get_variable("mbias2", (2, ), initializer=tf.random_normal_initializer(mean=0.01))

hidden = tf.nn.relu(tf.matmul(inputX, weight) + bias)
output = tf.nn.relu(tf.matmul(hidden, weight2) + bias2)

saver = tf.train.Saver()

def create_app():
    app = Flask(__name__)
    app.route("/tf", methods=["post"])(handle_request)
    return app

def handle_request():
    inputs = request.get_data()
    inputs = inputs.split(' ')
    mic = inputs[0:2]
    ultra = inputs[2:4]
    for i in range(0, len(mic)):
        mic[i] = float(mic[i]) / 100.0
    for i in range(0, len(mic)):
        ultra[i] = float(ultra[i]) / 255.0
    scaled_inputs = mic + ultra

    with tf.Session() as sess:
        saver.restore(sess, "video_weights.ckpt")
        x = sess.run(output, feed_dict={inputX: [scaled_inputs]})

    result = x.tolist()
    motor1 = format_motor_values(result[0][0])
    motor2 = format_motor_values(result[0][1])

    print "Motor1: " + motor1
    print "Motor2: " + motor2

    return motor1 + " " + motor2

def format_motor_values(x):
    x = x * 100
    if x > 127:
        x = 127
    if x <= 0:
        x = 1
    return str(x)

server = create_app()
server.run(host='0.0.0.0', port=3000)
```