Lindsey Sample

GridBot: a comparison of Evolutionary Algorithms in Dynamic and Static Environments

GitHub Repository: <<https://github.com/lpsample/Senior-Thesis>>

Table of Contents

Acknowledgements

Introduction

Methods

Results

Discussion

References

Table of Contents

**Acknowledgements**

**Chapter 1:** Introduction

**Chapter 2:** Methods

**Chapter 3:** Results

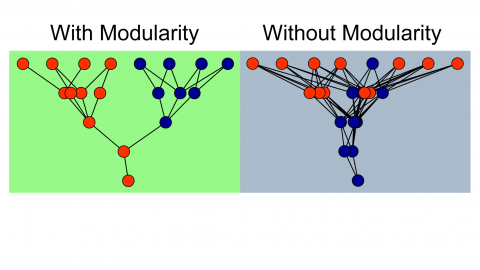
**Chapter 4:** References

Chapter 1: Introduction

Is there an optimal architecture for neuro-controllers? How do brain structures evolve with respect to the behaviors that they support? What are the tradeoffs between efficiency and full access to input information? Why are there different looking brains across species? What made them evolve that way? These are central questions in both Cognitive Science and Neuroscience, but it is difficult to address them given the complexity of animals. We can use models to examine hypotheses regarding brain architectures and the effect on fitness that variations can have. **Presently, we are interested in examining the differences in neural architectures that evolve in dynamic versus static environments in evolved fixed-topology artificial neural networks (FTANNs), compared to networks evolved using Neuroevolution of Augmenting Topologies (NEAT)** (Stanley & Miikkulainen, 2002)**.** Thus, there are two main topics of this study: the environment’s impact on evolution, and the method of modeling chosen.

1. Modularity and Sparsity: Adapting to Environments

Often, Cognitive Scientists have used neural networks to model the brain of biological organisms. The standard neural net features a suitable architecture of nodes, and requires the development of appropriate weights between nodes that yield optimal output or performance. When exploring architectures of neurocontroller, a common method of describing and analyzing architecture is to examine modularity and sparsity (Cappelle, Bernatskiy, Livingston, Livingston, & Bongard, 2016). This attempts to quantify the degree to which the system is divided into modules, or is more fully-connected. Clune, Mouret, & Lipson (2013), define modular networks as those which contain highly connected clusters of nodes that are sparsely connected to nodes in other clusters. Another way to conceptualize modularity is to examine how discrete or integrated communication is within the network between input (sensors) and output (motors).



**Figure 1.** Depiction of ANNs with modularity and without modularity(Huizinga, Mouret, & Clune, 2014).

Modularity is believed to evolve from selection pressure **to reduce connection costs** (Clune et al., 2013). It is important to note the distinction that modularity does not evolve from selection pressure on performance alone, but specifically on pressure to reduce connection costs (Clune et al., 2013; Livingston et al., 2016). This can often occur in environments that are changing and dynamic, which it is not surprising to require dynamic change of the agent, in turn. The modularity of nervous systems is hypothesized to enhance evolutionary adaptation of a population by allowing selection to target regions separately. Previous research suggests modular networks are favorable for handling evolutionary goals. Livingston et al. (2016), suggests modularity can allow rapid response to environmental change, specialization without loss of useful sub-functions, and avoidance of “catastrophic forgetting.”

Kashtan and Alon (2005) found that the evolutionary emergence of modularity was related to the environment faced and evolutionary goal experienced. They found that when goals were repeatedly switched (with common subgoals), the networks rapidly evolved to satisfy the different goals with only a few rewiring changes. They claim that these evolutionary forces favor modularity for its structural simplicity and ability to rapidly adapt (Kashtan and Alton, 2005). They found that when the varying goals contained no common sub-goals, modular structures did not evolve and that adaptation was very slow (since evolution was essentially starting from scratch each time there was a change).

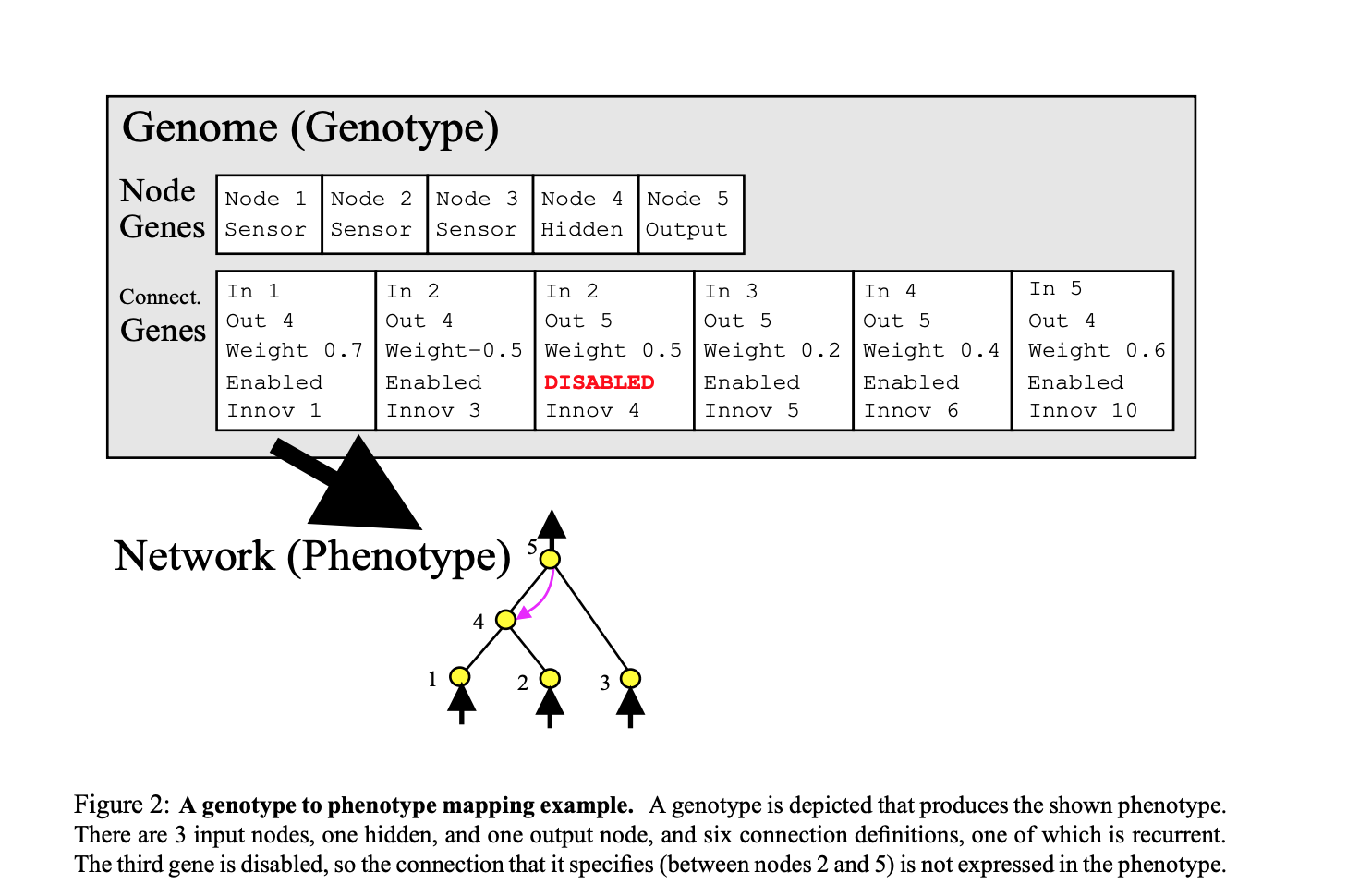
The majority of research on this topic points to the claim that cognitive architectures develop with respect to their specific environments, or the tasks required of them. Modularity evolves with respect to pressure to reduce connection costs in environments, and is a strategy for easily evolving to changing environments. This leads us to the hypothesis that modularity will be favored in dynamic environments, more-so than in static environments. We have developed a model to test this hypothesis.

1. Methods of Modeling Neuro-controllers

Typically, network topology features a single hidden layer of neurons which connect to an input and an output layer node. These networks are essentially searched by the means of evolution to find optimal or suitable connection weights, allowing high performing networks to arise. This is the goal in a fixed-topology network (Stanley & Miikkulainen, 2002). But, what is often overlooked is that the topology itself, or structure, of the network is another factor that affects functionality, not weights alone. In traditional fixed-topology ANNs, this is overlooked. This idea has been around for many years (Gruau, 1995), and there are several possible advantages. Evolving structures can save time, compared to fixed-topology systems that require a trial-and-error process to determine the optimal amount of hidden nodes. Gomez & Miikkulainen (1999) showed that a pole balancing task problem could be solved 5 times faster using an algorithm that spawned a random number of hidden layer neurons when it became stagnant. Stanley & Miikkulainen (2002) showed that not only can non-fixed-topology networks improve speed, but also provide an overall more efficient and higher performing alternative by taking advantage of structure as the means to minimize the search space of connection weights. This is achieved by minimizing and incrementally growing the topology, which minimizes excess burden on the network throughout the evolution process, rather than at the end. In this manner, structures become more and more complex as they are optimized, more accurately mirroring and reflecting genetic algorithms and natural evolution.

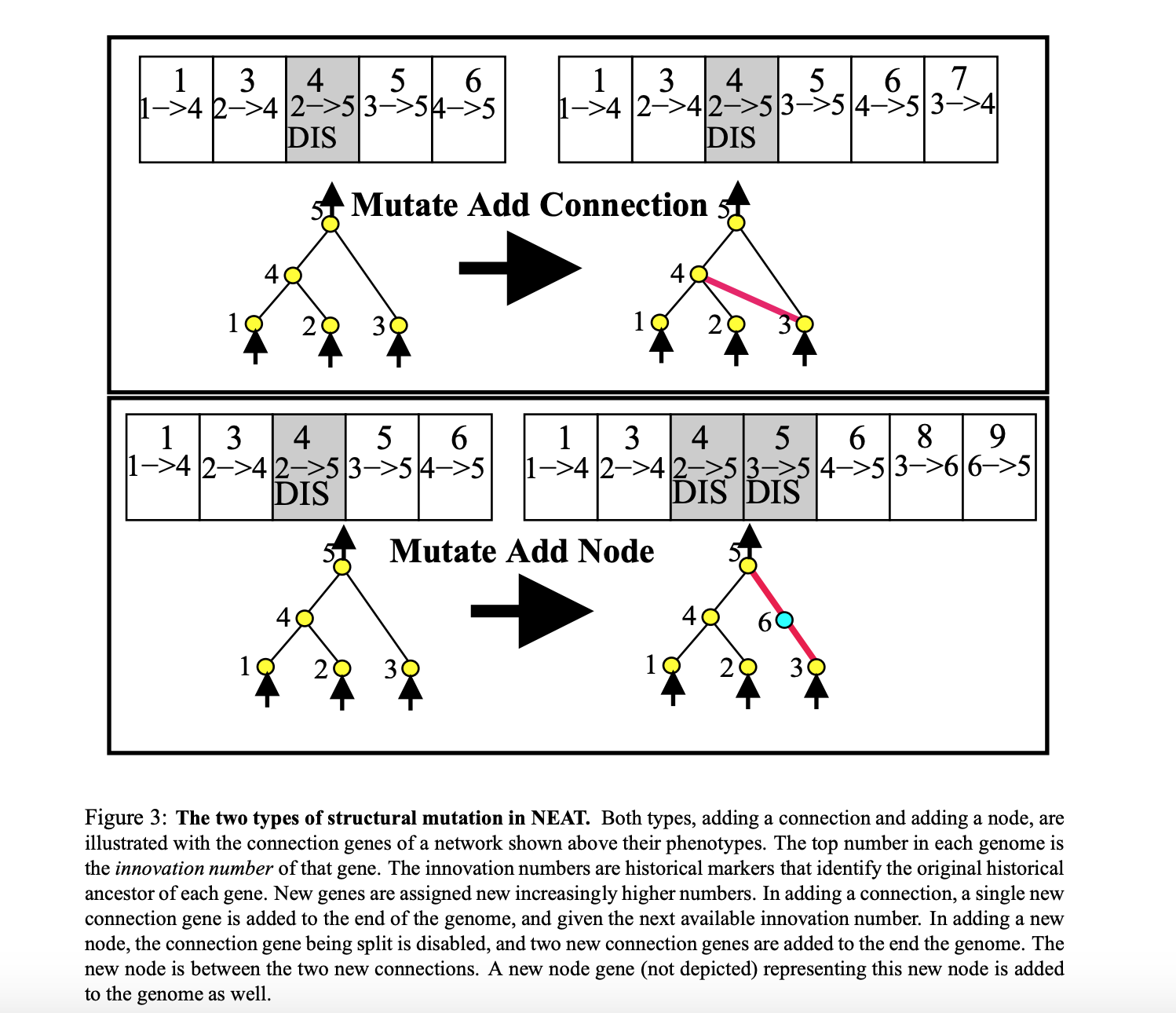
The NEAT algorithm begins with an initialization that minimizes the number of nodes. The network is evaluated based on a fitness function, and then, is mutated. This repeats for a specified number of iterations. The network needs to be penalized for excess complexity that doesn’t add functional benefit, but not so heavily penalized that new complexities cannot be added that will aid the system, but are not weighted appropriately yet, therefore, too heavily restricting evolution. The NEAT algorithm calls upon many biological ideas, such as genetic encoding, historically marking genetic crossover (and emulating this crossover, which is challenging in computational neuroscience), genotype/ phenotype distinction, and speciation.

Each genome for each individual evolved using NEAT contains a node gene set and a connection gene set. The node gene set specifies if a node is a sensor node, a hidden layer node, or an output node. The connection gene set specifies which node the connection originates from (“in”), where the connection leads (“out”), the weight, if the node is enabled or disabled (active or not), and the innovation number (used for crossover).



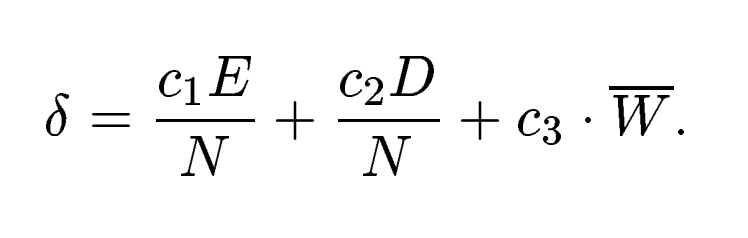
**Figure 2.** Genotype to Phenotype mapping. The genome, featuring the node gene and connection gene sets (top), and a depiction of the network (bottom) (Stanley & Miikkulainen, 2002).

During the mutation step, following evaluation, the network can either add a connection or add a node. When the algorithm adds a node or connection, this step is given a number, called the innovation number (see top of boxes in **Figure 3** below).



**Figure 3.** Mutation by adding a connection (top) and a node (bottom). The top number in each genome specifies the innovation number of that gene, or the chronological order of steps mutations made. These are historical markers that serve to identify ancestors of genes. When adding a connection, a new connection gene is added to the end of the genome and given the next successive available innovation number. When a new node is added, the connection gene formerly in place is disabled, and two new connection genes are added to the end of the genome, along with a new node gene (not shown) (Stanley & Miikkulainen, 2002).

In order to protect new complexities evolving from being penalized, NEAT uses speciation. This allows similar networks to compete in evaluation, while allowing new innovations to develop without competing by dividing them into different species.



This speciation function depends on: E, the number of excess genes, D, the number of disjoint genes, and W, the average weight differences of matching genes, between two different samples of the population. If delta exceeds a certain threshold, it will be grouped into a new species. The fitness function incorporates the number of members in a species, so fewer individuals in a species result in higher fitness for a given individual in that species.

1. Our Model

Presently, we are interested in examining FTANNs and NEAT evolved NNs in both static and dynamic environments to compare and contrast the resulting networks in terms of modularity and fitness.

We will examine these using a model featuring a simulated robot called GridBot. This model will compare the evolution of FT and NEAT networks in a dynamic environment, and a static environment. GridBot‘s specific task is to traverse a 100 square GridWorld (starting from the bottom center of the world) that features a light source at the top of the and 5 obstacles. In the static condition, the GridWorld is always a 10x10 matrix with the 5 obstacles in the shape of a plus in the center of the grid. In the dynamic condition, the grid can be 5x20, 4x25, 20x5, 25x4, or 10x10, and the 5 obstacles are randomly placed throughout the grid. There are also boundaries surrounding the edge of the grid in both conditions that GridBot cannot occupy. GridBot features simulated versions of two photoresistor sensors (LDRs), two infrared sensors (IRs), and one bumper. GridBot navigates GridWorld autonomously, making movement decisions based on the values inputted and computed in its neural network. We will analyze and compare the best performing emergent architectures in the static and dynamic conditions for both the FT and NEAT bots.

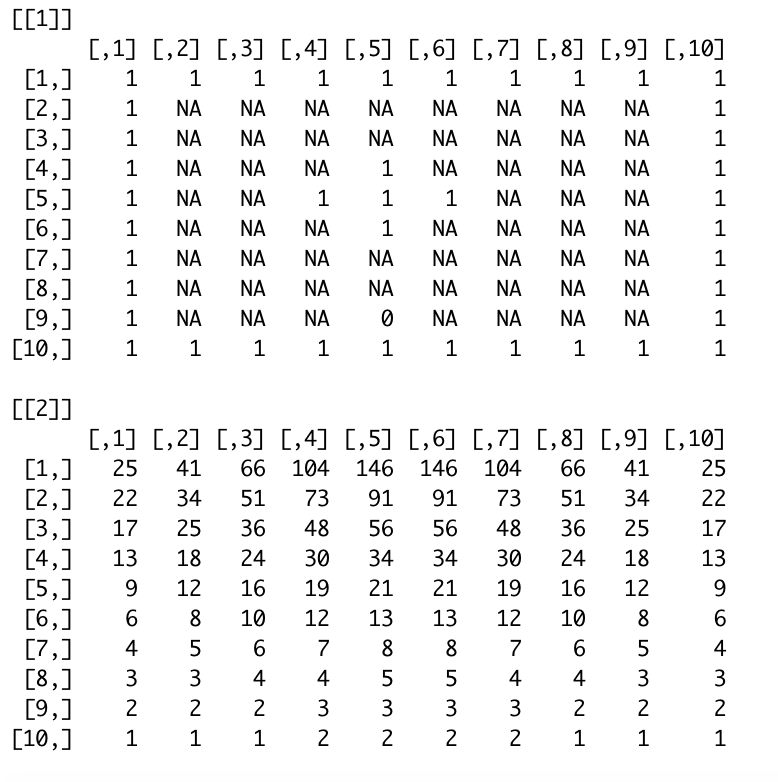
Chapter 2: Methods

1. Description of the Model

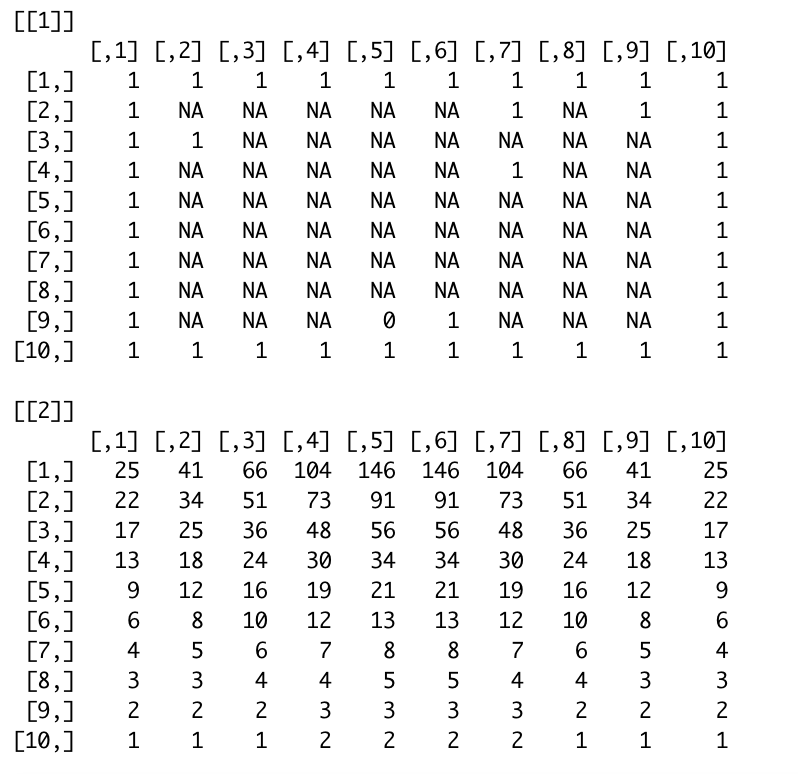
*GridBot and GridWorld.* Presently, we develop a simulated world to emulate a toy world for a simulated bot to traverse. This bot, called GridBot, has two light dependent resistors (LDRs), two infared sensors (IRs), and one bumper on its front. It can traverse a grid world by moving forward or backward, or by turning to the left or right. Each step, GridBot can make one move: stepping, or turning. GridBot can make 100 moves in a trial, and move backwards, move forwards, turn clockwise, or turn counterclockwise on any given move. Each move is determined probabilistically based on its NN featuring 5 input nodes corresponding to its 5 sensors, and 4 output nodes, corresponding to its 4 move options. In the FT NN, there are no hidden layer nodes, and in the NEAT NN, hidden layer nodes and connections are mutated throughout the simulation, resulting in any number of hidden layer nodes under the maximum, 45.

GridWorld is a grid, composed of a 10x10 matrix (100 squares) in the static condition, and a randomly determined NxN matrix (also equal to 100 squares) in the dynamic condition (4x25, 25x4, 10x10, 20x5, or 5x20). Every square within the grid contains a value of light that can be collected by entering that square, and a binary value that indicates if there is an obstacle in that square. In the static condition, there are 5 obstacles that form a plus in the center of the GridWorld. In the dynamic condition, there are 5 randomly placed obstacle squares. There are boundaries that line the border of GridWorld that function as obstacles as well. GridBot cannot step into a square that an obstacle inhabits, though it can sense the light in that square if it is facing or next to the square. This will be elaborated upon in the explanation of the functional LDRs.

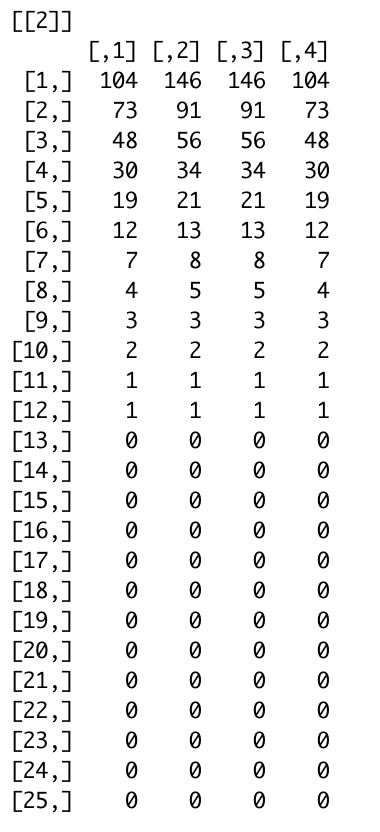
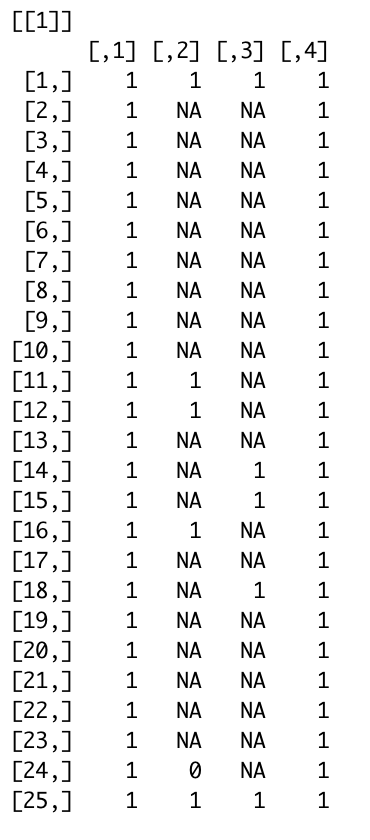
Light values are highest near the source of the light at the top center of the grid, and fall off in a gradient (255\*e^(-distance\*.5)). The light grids depict the amount of light represented in each square numerically and the obstacle grids depict blank spaces as ‘*NULL*,’ the bot as ‘0,’ and obstacles as ‘1.’ **Figures 4-9** below show all possible grid dimensions with examples of randomly placed obstacles in the dynamic condition grids.



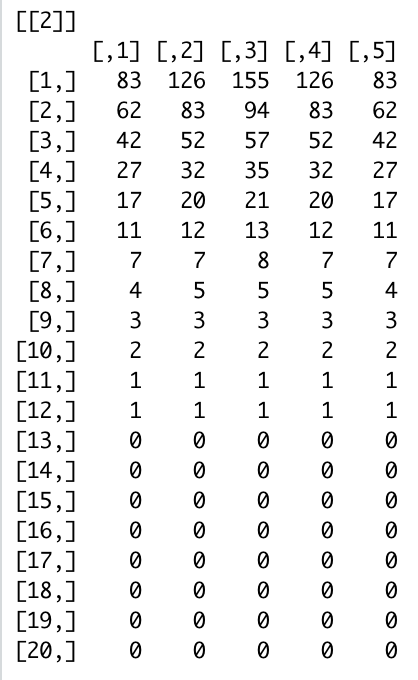
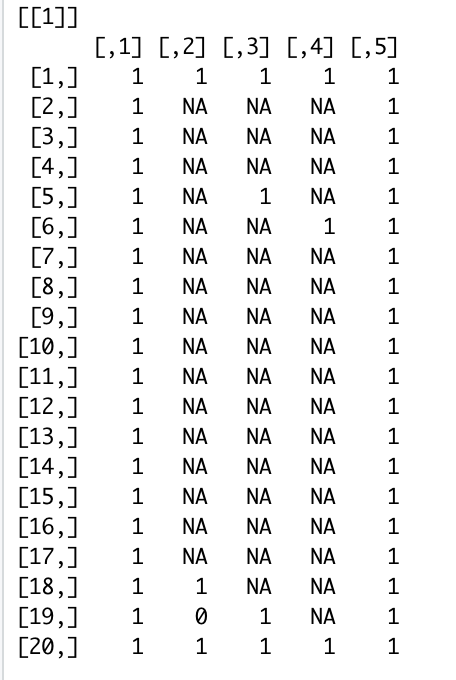
**Figure 4.** 10x10 Grid with “plus” shape obstacles (for all static condition runs). Top: obstacle grid, bottom: light grid.



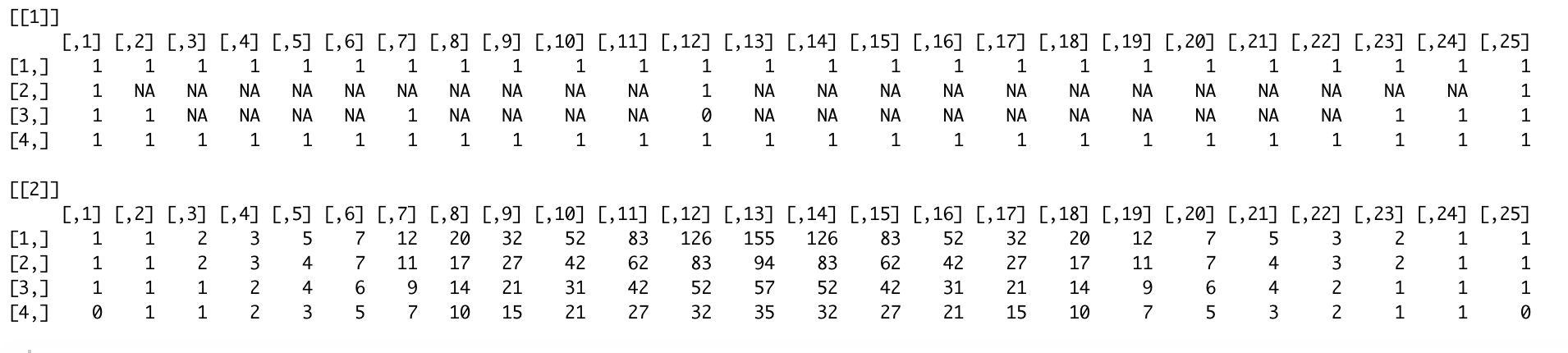
**Figure 5.** 10x10 Grid with randomly placed obstacles (example of 10x10 dynamic condition) Top: obstacle grid, bottom: light grid.



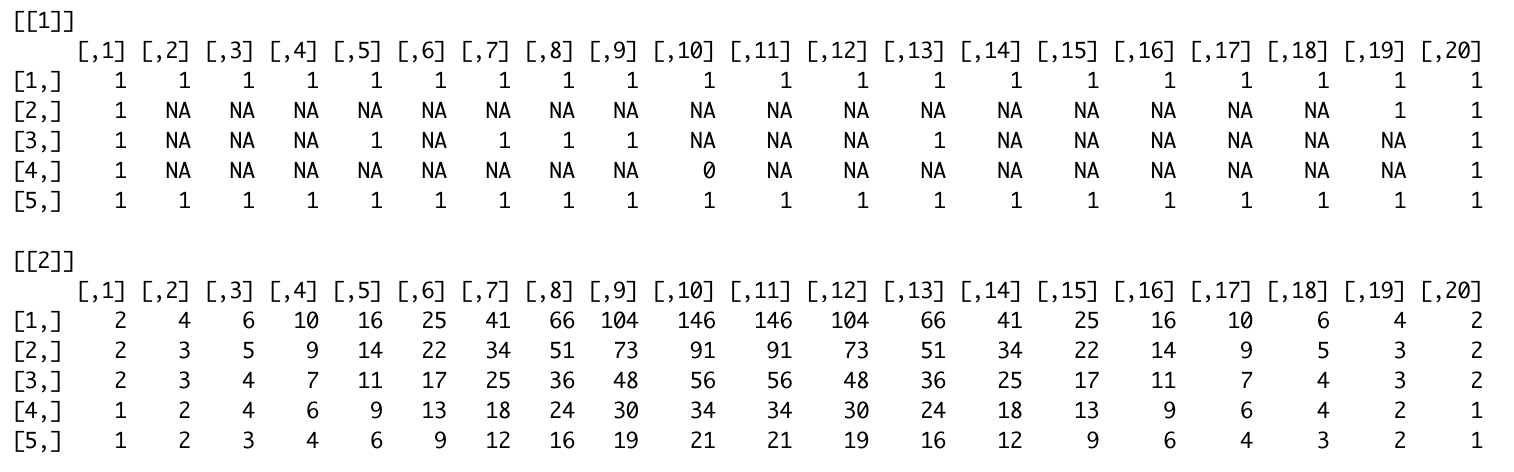
**Figure 6.** 4x25 Grid with randomly placed obstacles (example of 4x25 dynamic condition). Left: obstacle grid, right: light grid.



**Figure 7.** 5x20 Grid with randomly placed obstacles (example of 5x10 dynamic condition). Left: obstacle grid, right: light grid.



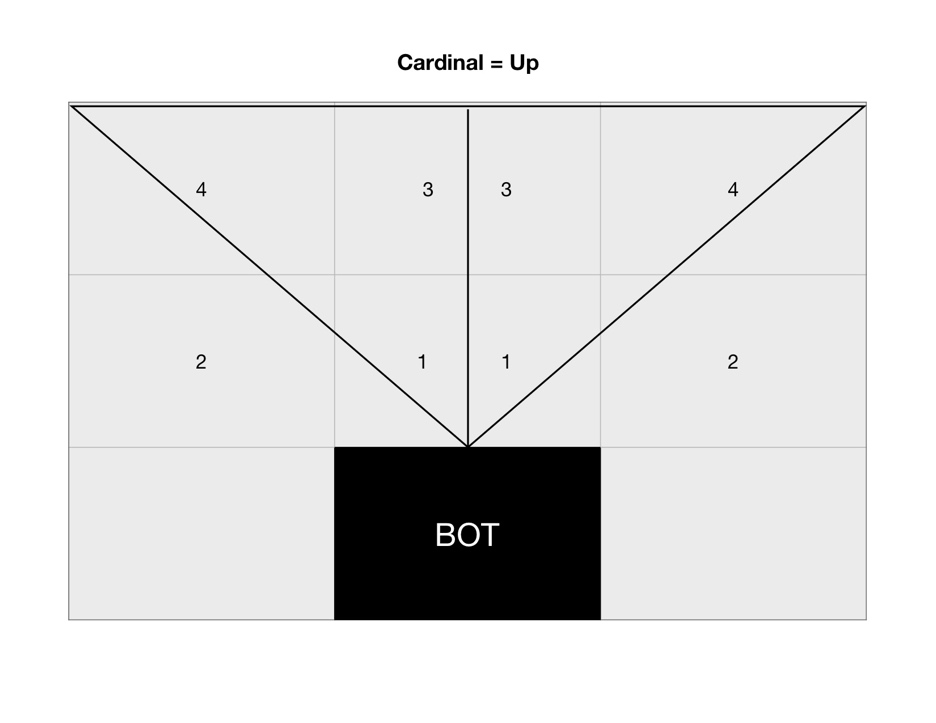
**Figure 8.** 25x4 Grid with randomly placed obstacles (example of 25x4 dynamic condition). top: obstacle grid, bottom: light grid.



**Figure 9**. 20x5 Grid with randomly placed obstacles (example of 20x5 dynamic condition). top: obstacle grid, bottom: light grid.

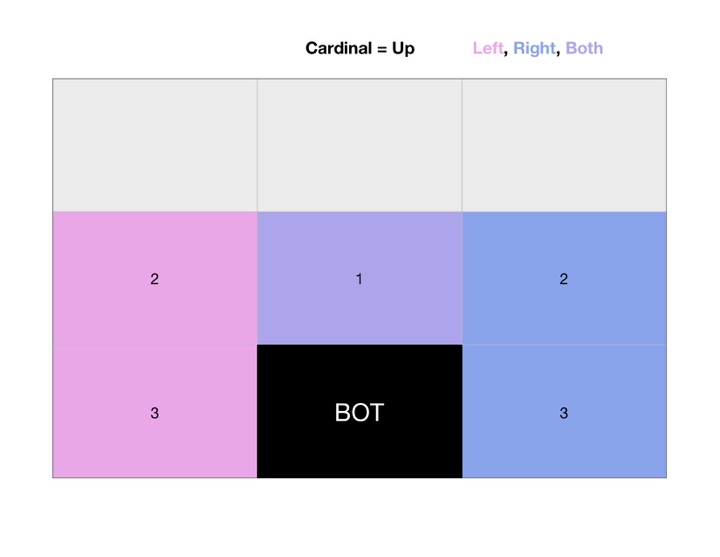
GridWorld. In the static condition, the light source resides at the top of the world (5,5). In the dynamic condition, the light source is in the center top as well, but the exact source location depends on the grid’s dimensions. Both the bot and the light is placed using the built in *round()* function on *numCols/2*, to place the light and the bot in the center. The code stipulating these functions is found in the file “*GridBotv2.0.Rmd*,” within the [GitHub repository](https://github.com/lpsample/Senior-Thesis). See functions *dynamicNums* (provides dimensions for dynamic condition), *makeLightGrid*, and *makeGrids,* and *setUp* (takes in a 1 for a static condition setup and a 2 for a dynamic condition setup).

*Sensors.* IRs are calculated by searching the four spaces in front of the bot as depicted in **Figure 10**. The spaces are searched in order, and if an object is detected in the space, the IR value becomes the distance between the bot’s current location in the grid and the obstacle. The code stipulating this functionality is found in the file “*GridBotv2.0.Rmd*.” See functions *getRightIR,*and *getLeftIR*.



**Figure 10**. Specifications for IR sensors (right and left).

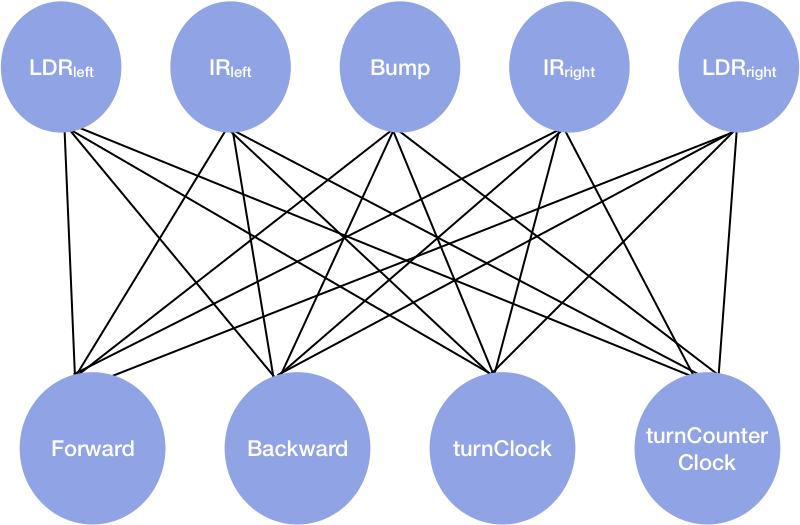
LDRs are calculated by taking the average of the space the bot is currently in and the 3 adjacent spaces (above, aside, and diagonal, depicted in **Figure 11**). The light it collects in a given move is the sum of the right and left LDRs. Each LDR assumes the light value of the space it is sensing according to these specifications. The code stipulating this functionality is found in the file “*GridBotv2.0.Rmd*.” See functions *getRightLight,*and *getLeftLight*.



**Figure 11.** Specifications for LDR sensors.

GridBot also features a bumper, that returns 1 when there is an object directly in front of where GridBot is facing, and 0 when the space is unoccupied. See function *getBump* in the “*GridBotv2.0.Rmd*” file.

*Fixed-topology Neural Network Structure.* Each GridBot has a weight matrix, where weights can take on values of -1, -.5, 0, .5, and 1. On each turn, each output receives a value that is the dot product of the weight matrix (or the bot’s genome), and the input. The outputs are also scaled using a sigmoid function (1 / (1 + x^(-sum(input\*weight[,i])), where *i* corresponds to the index of the node, and divided by the sum of outputs (to appropriately scale them for probabilistically choosing between them). Then, an output node is probabilistically chosen, which represents the move for that turn (moveForward, moveBackward, turnClock, turnCounterClock). This action is then taken, sensors are updated, and the process repeats for 100 moves. At the end of a trial, a GridBot’s fitness is equal to the amount of light it has accumulated in that 100 moves. See **Figure 10** for a visual representation of GridBot’s FTNN structure. The code stipulating this approach is implemented in “*GridBotv2.0.Rmd*,” using the functions *updateInput* (using the respecting ‘get’ functions for leftIR, rightIR, leftLight, rightLight and bumper), *makeMove* (updating the state, which is a list holding the light grid obstacle grid, and cardinal direction that the bot is facing, by choosing an action and updating the obstacle grid).

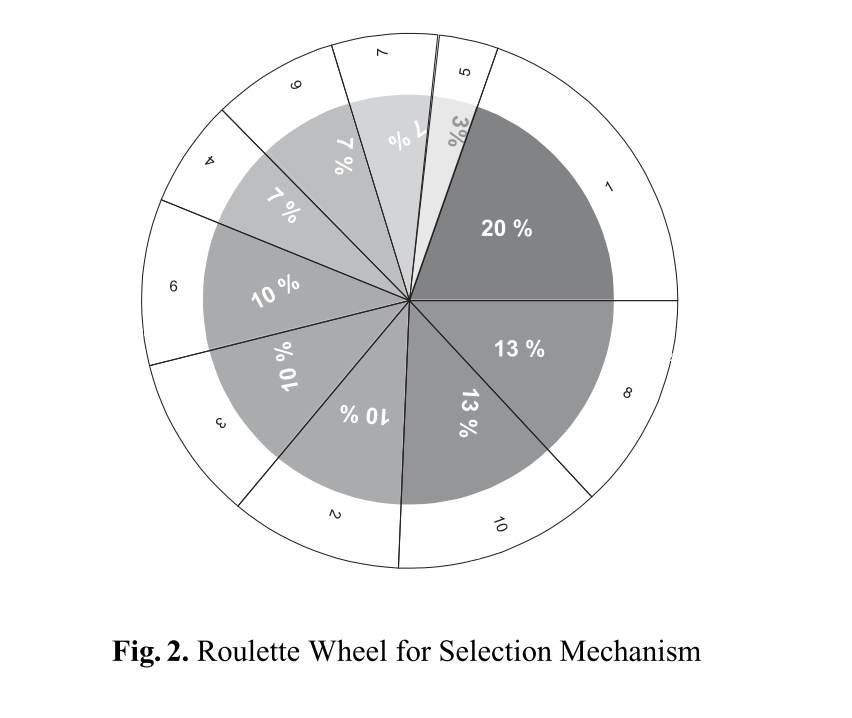


**Figure 10.** Depiction of GridBot’s FTNN. In this example, the network is fully connected.

*NEAT Network.* We implemented the existing NEAT framework using Andrew Hunter’s (2016) RNeat GitHub Repository: <<https://github.com/ahunteruk/RNeat>>. The static and dynamic conditions for NEAT GridBot are implemented in the ‘*NEATGridBot.Static.R’* and ‘*NEATGridBot.Dynamic.R’* files within this project’s [GitHub repository](https://github.com/lpsample/Senior-Thesis), respectively, in which, there is a large degree of overlap. These files source from Hunter’s *‘neat.R,,’ ‘neatCharting.R,’* and *‘neatFormula.R’* scripts, and implement several functions for the use of our model. *gridBot.InitialState* creates the grids using *makeGrids*, and returns *state,* containing the light and obstacle grids, as well as the cardinal direction of the bot. This is the only difference between the static and dynamic scripts, where in the static script, *makeGrids(1)* is called, and in the dynamic script, *makeGrids(2)* is called. *gridBot.ConvertStatetoNeuralNetInputs* takes in the current state, updates the sensors, and returns these values as *neuralNetInputs. gridBot.UpdateState* takes in the current state and *neuralNetOutputs*, makes a move based on the possible actions, and returns the updated state. Additionally, a plot of the obstacle grid is saved at this step if *plotState* is set to *true*, which we used to visualize our best preforming bots’ paths. *gridBot.UpdateFitness* takes in the old state, the updated state, and the old fitness, and returns the new fitness, or in our case, total light. *gridBot*.*CheckForTermination* takes in the frame number, old state, updated state, old fitness, and the new fitness, and returns *false* until the frame number exceeds the number of moves (in our case, 100). Each of these functions are taken in as inputs into the *newNEATSimulation* function from Hunter (2016), along with a *newConfigNEAT* object, which serves as a configuration for the number of inputs (5), outputs (4), maximum number of nodes (we decided arbitrarily upon 50), and the species population (10).

*Selection.* GridBot survives by harvesting light. Each move, GridBot’s light counter is updated by adding the light it has collected by moving into a new square, and taking the mean of the light sensed in the square it is in, averaged with the 3 adjacent squares to the right (for the right sensor) and left (for the left sensor). The total light collected over a trial (100 moves) will serve as that bot’s fitness, where the most fit bots collect the most light. Light is intended to represent a food source.

For the FTNN, the evolutionary algorithm uses a roulette style wheel algorithm to select the next generation of GridBots based on relative fitness of the previous generation. This evolutionary algorithm is taken from Haddow & Tufte (1999), and depicted in **Figure 11.** ﻿ In this example, individual number 1 has 20% percent of the roulette wheel whereas individual number 4 has only 7% percent of the wheel. As such, individual 1 is more likely to be selected than individual number 7. The selection mechanism “spins the wheel” or in our case, uses the built-in sample function (with replacement) to select the next generation’s genomes. The wheel has to be spun 10 times to select 10 individuals to retain the size of the population in the new generation.



**Figure 11.** Roulette style wheel for selection, based on relative fitness (Haddow & Tufte, 1999).

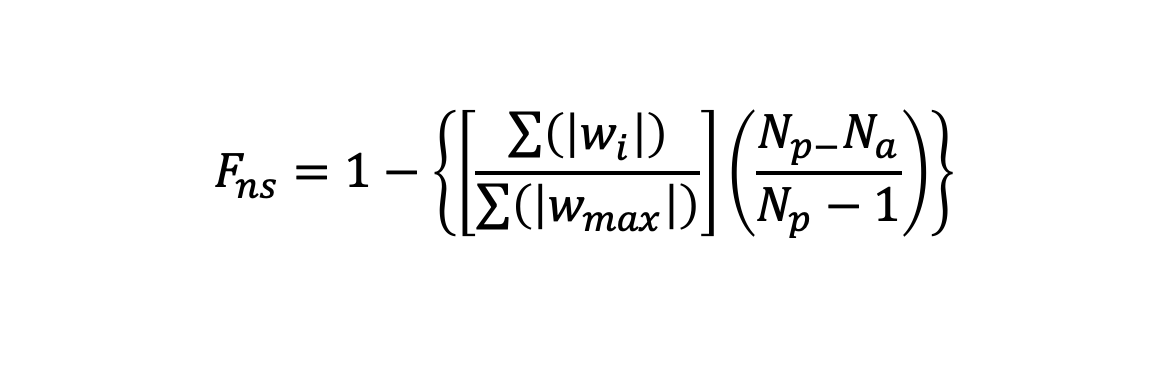
The corresponding bot numbers are selected, and each bot’s weight matrix is mutated at a rate of 5%. This means 1/20 weights are changed at random (from the list of possible values: -1, -.5, 0, .5, and 1). The experiment is run for 100 generations, with 10 bots per generation, and 100 moves per bot, per trial. The experiment is run in the dynamic and simulated condition. The code for this feature is found in ‘*GridBotv2.0.Rmd’* in the *next.genomes* function. It is also important to note that each individual’s genome in the first generation for the FTNN bots is composed of random weights, found in the *createGeneration* and *makeRandWeights* functions.

NEAT handles selection using functions within Hunter (2016)’s ‘*neat.R’* script. Many of these functions alter the pool, which contains all individuals, their genotypes, and their phenotypes. These functions are: *cullSpecies , rankGlobally, removeStaleSpecies , rankGlobally* (again), *calculateAverageFitness, removeWeakSpecies,and totalAverageFitness,* which are all called in the *newGeneration* function, in that order. A list of children is created, new children are bred, species are culled, and the generation number is incremented by 1. The starting population is initialized using *initializePool*, which takes in a NEAT configuration object and creates a basic genome based on the inputs and outputs specified, which are mutated using the *mutateGenome* function, that works as described in our earlier description of mutation.

1. Functional Modularity

*Functional Modularity.* Functional Modularity (FM) is calculated to compare networks in terms of how fully connected or modular their structures are. FM, for our purposes, is determined as any of the sets of the minimum number of edges needed to connect one sensor input to one motor output. For the common three-layer NN, the total number of possible circuits in the NN is the product of the number of nodes in each layer. The following approach was adapted with the guidance of John Long and Ken Livingston in 2018.

Our measurement for node-specific functional modularity, Fns, is for each node, and is calculated as follows:



where, for all circuits in which the node is involved *wi* is the weight of each edge; *wmax* is the maximum possible weight of each of those edges; *Np* is the number of possible circuits for that node; and *Na* is the actual number of circuits for that node. For a three-layer neural network, the *Np* for any input node is the product of the number of hidden layer nodes and the number of output layer nodes; for any output node the *Np* is the product of the number of hidden layer nodes and the number of output layer nodes. A circuit is defined as a path from an input node to an output node.

For the purposes of our model, *Np* = 20 for both FT conditions, and *Np* = 104 and 144 for the NEAT static and dynamic conditions, respectively. See *‘w\_sum.xlsx*’ for the calculation of *Fns*.

Functional modularity is one of many measures by which we can examine the differences between networks, and while we are using this measure for this project due to time constraint, we hope to expand this analysis in the future.