Lindsey Sample

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

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

Name and Page #

Table of Contents

Acknowledgements

Introduction

Methods

Results

Discussion

References

Table of Contents

**Acknowledgements**

**Abstract**

**Chapter 1:** Introduction

**Chapter 2:** Methods

**Chapter 3:** Results and Discussion

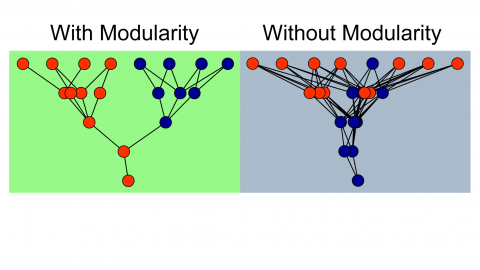
**Chapter 4:** References

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).

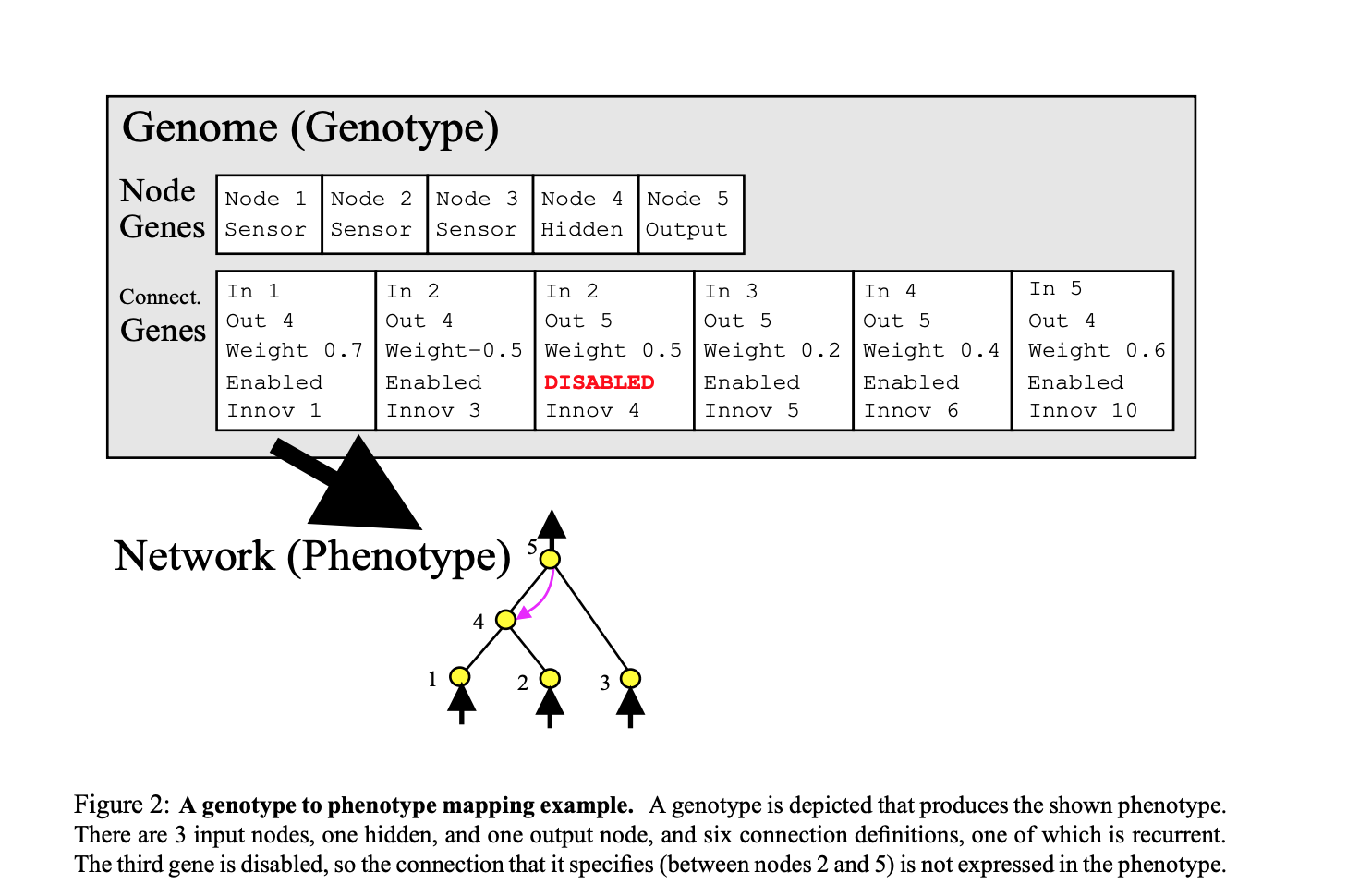
The majority of research on this topic points to the claim that cognitive architectures develop with respect to their specific environments, or the tasks required of them. Modularity evolves with respect to pressure to reduce connection costs in environments, and is a strategy for easily evolving to 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.

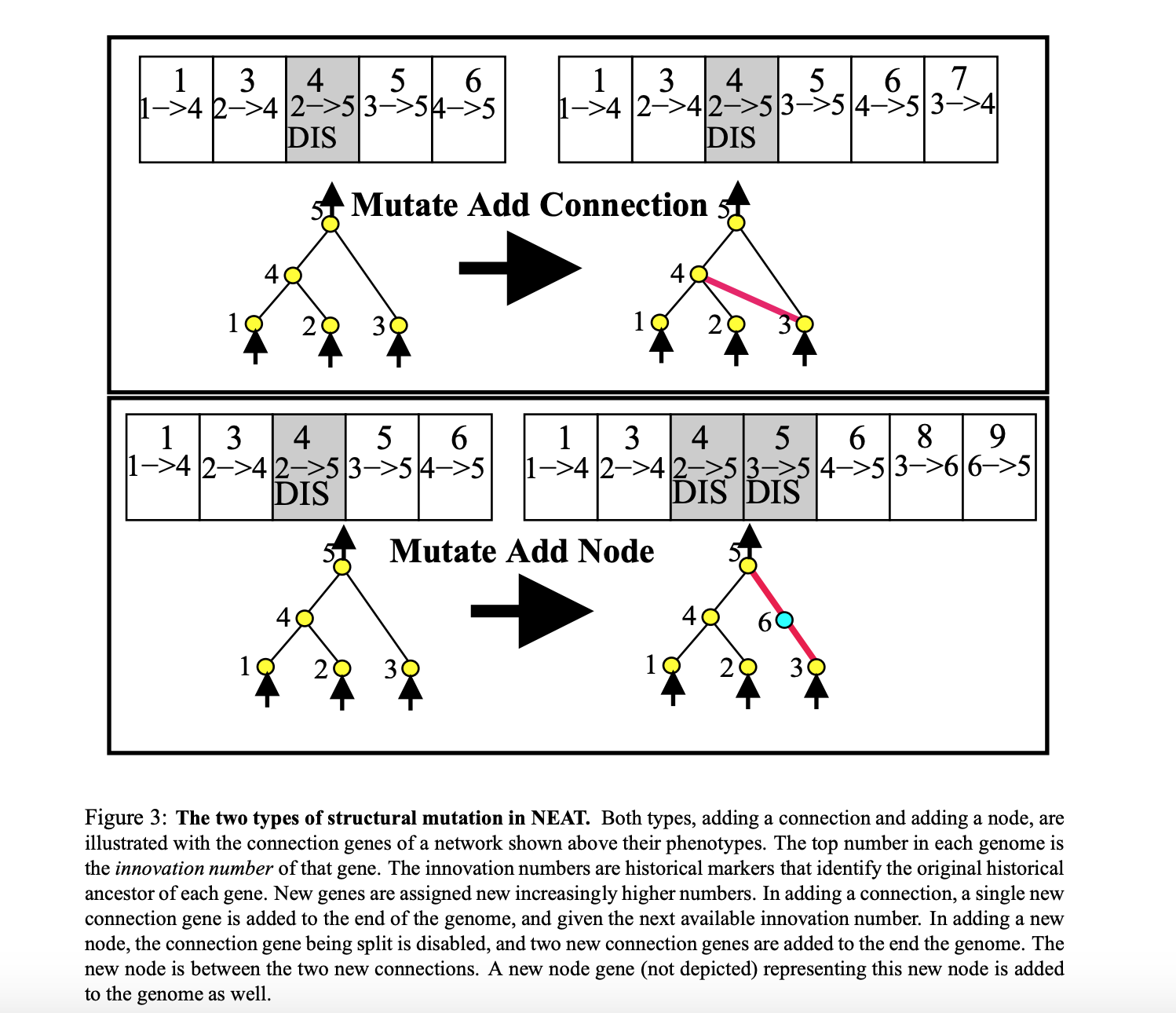
The NEAT algorithm begins with an initialization that minimizes the number of nodes. The network is evaluated based on a fitness function, and then, is mutated. This repeats for a specified number of iterations. The network needs to be penalized for excess complexity that doesn’t add functional benefit, 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).



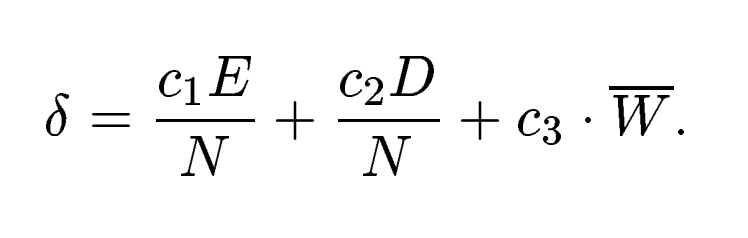
**Figure 2.** Genotype to Phenotype mapping. The genome, featuring the node gene and connection gene sets (top), and a depiction of the network (bottom) (Stanley & Miikkulainen, 2002).

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.

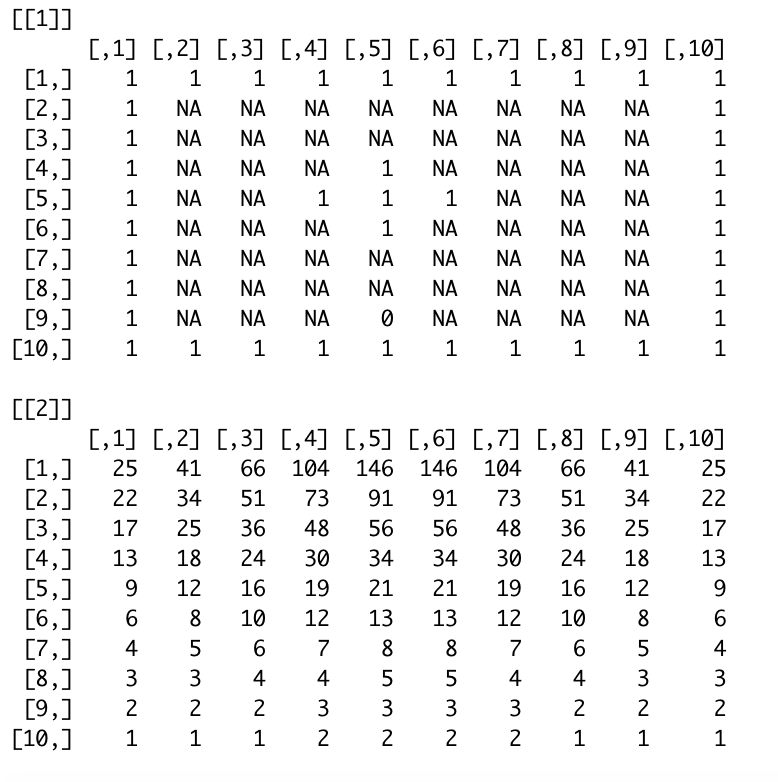
Chapter 2: Methods

1. Basic Description of the Model

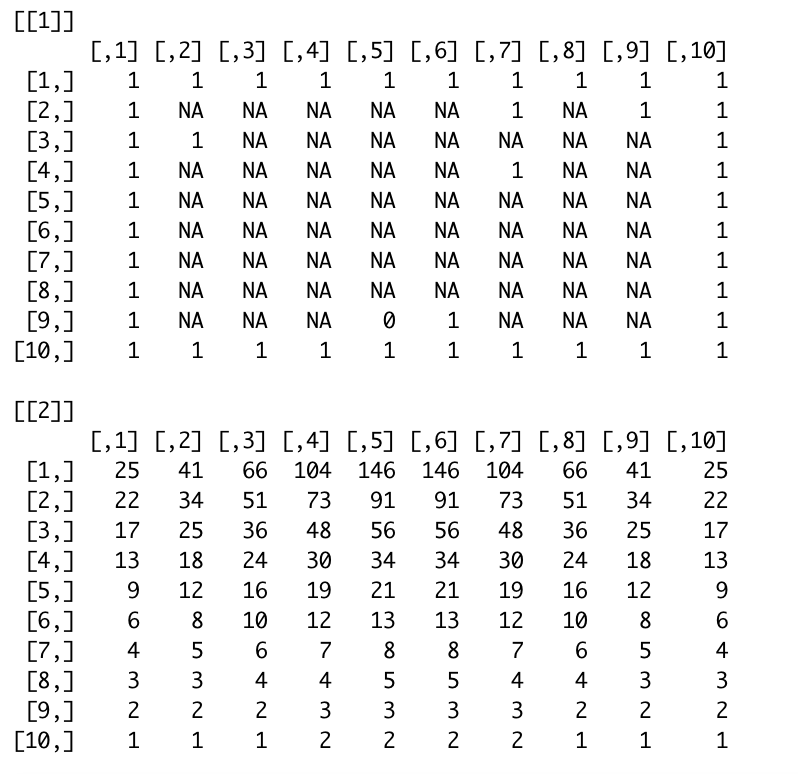
*GridBot and GridWorld.* Presently, we develop a simulated world to emulate a toy world for a simulated bot to traverse. This bot, called GridBot, has two light dependent resistors (LDRs), two infared sensors (IRs), and one bumper on its front. It can traverse a grid world by moving forward or backward, 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.

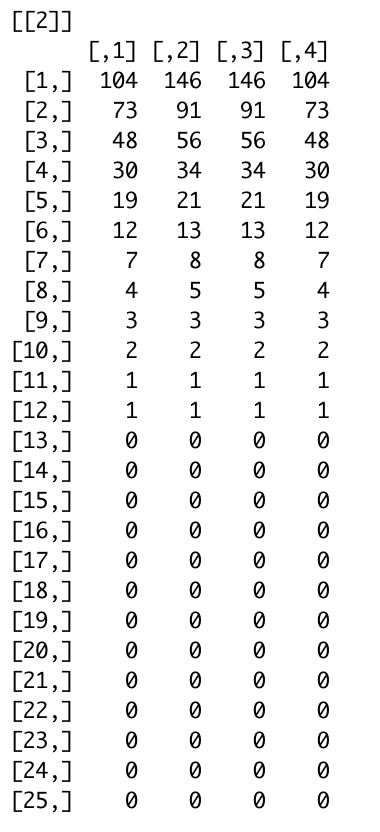
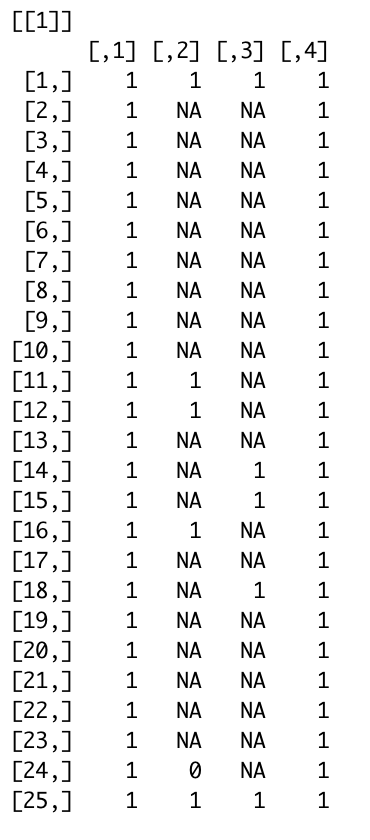
Light values are highest near the source of the light at the top center of the grid, and fall off in a gradient (255\*e^(-distance\*.5)). The light grids depict the amount of light represented in each square numerically and the obstacle grids depict blank spaces as ‘*NULL*,’ the bot as ‘0,’ and obstacles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



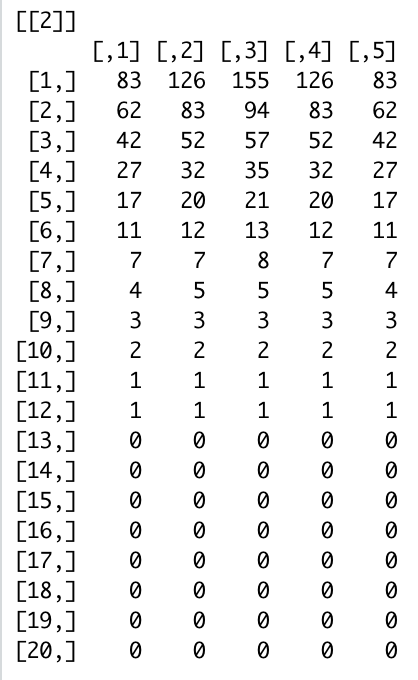
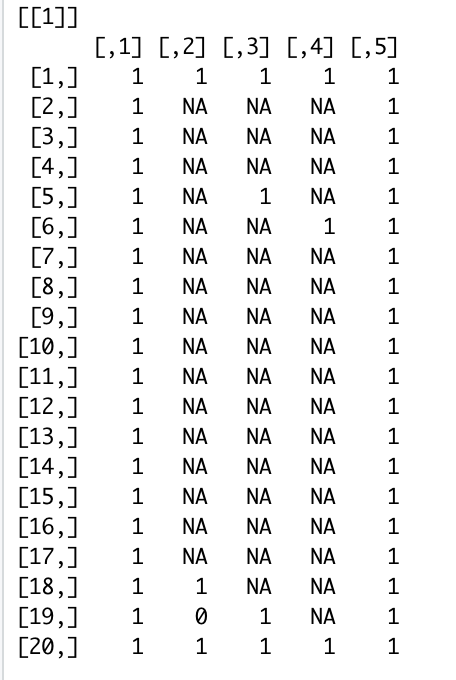
**Figure 4.** 10x10 Grid with “plus” shape obstacles (for all static condition runs). Top: obstacle grid, bottom: light grid.



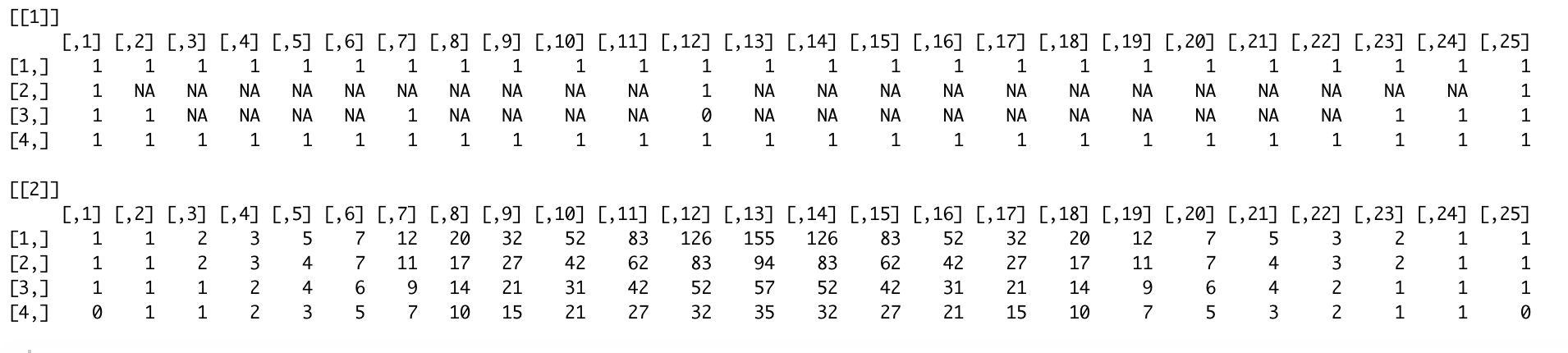
**Figure 5.** 10x10 Grid with randomly placed obstacles (example of 10x10 dynamic condition) Top: obstacle grid, bottom: light grid.



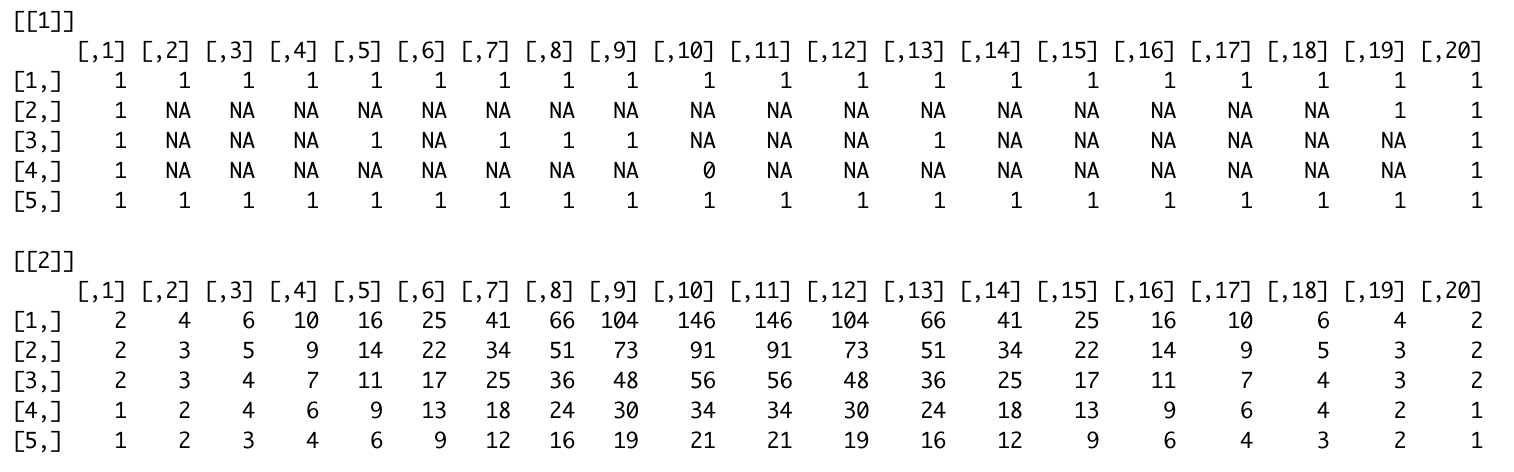
**Figure 6.** 4x25 Grid with randomly placed obstacles (example of 4x25 dynamic condition). Left: obstacle grid, right: light grid.



**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.

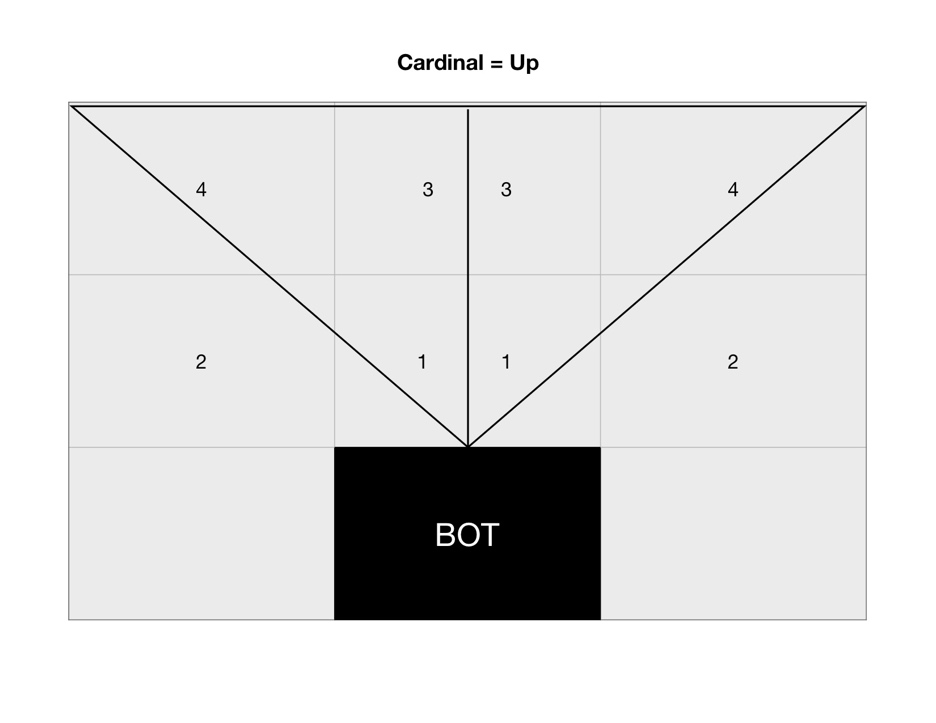


**Figure 9**. 20x5 Grid with randomly placed obstacles (example of 20x5 dynamic condition). top: obstacle grid, bottom: light grid.

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).

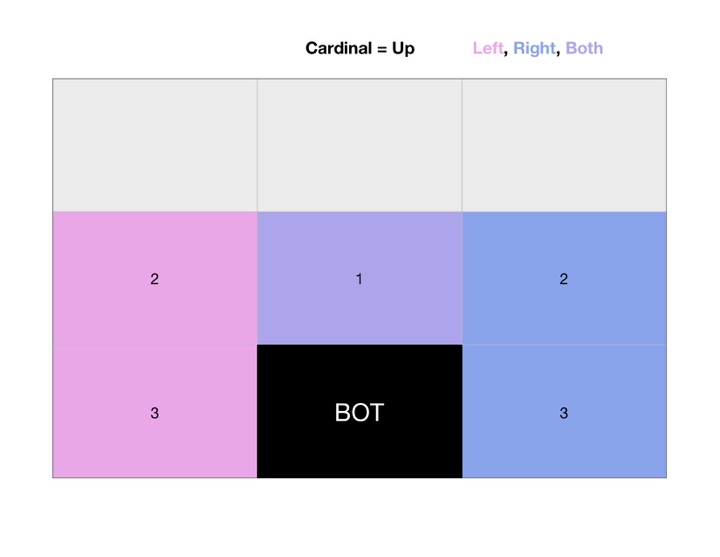
1. Sensors

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*.

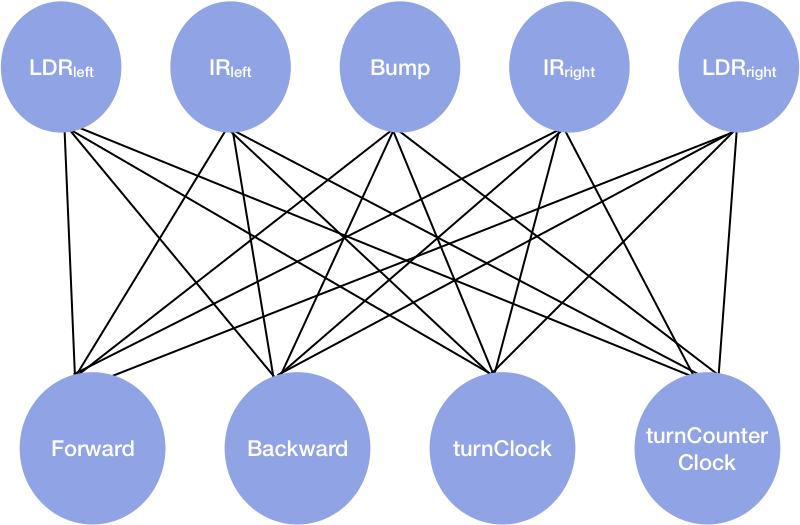


**Figure 11.** Specifications for LDR sensors.

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.

1. 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.

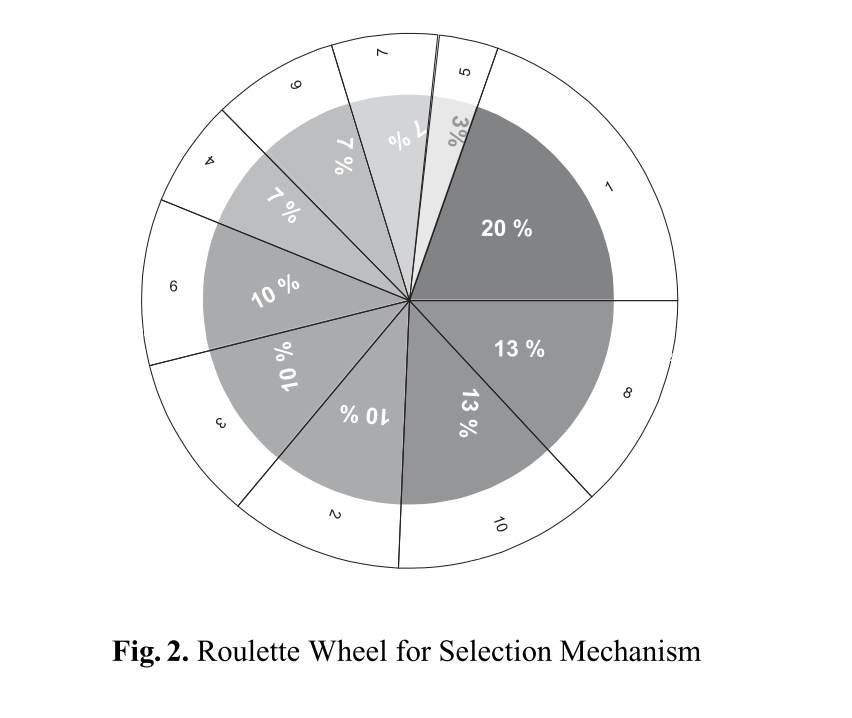
1. NEAT Network

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).

1. 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.

Chapter 3: Results and Discussion

1. Comparison of Networks

We found four distinct networks for our best performers in the static and dynamic conditions in both FT and NEAT networks.

*Fixed-topology Network Basic Analysis.* Our FT results yielded the weight matrices depicted below in **Tables 1 and 2**, and visualized in **Figure 12**.

**Table 1.** Best performingfixed-topology network’s connection matrix for the static condition.

Static Condition

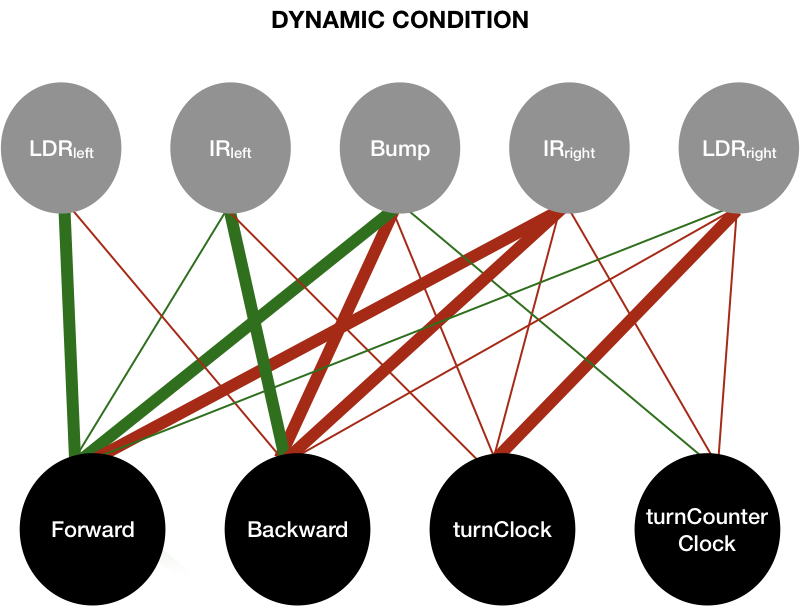
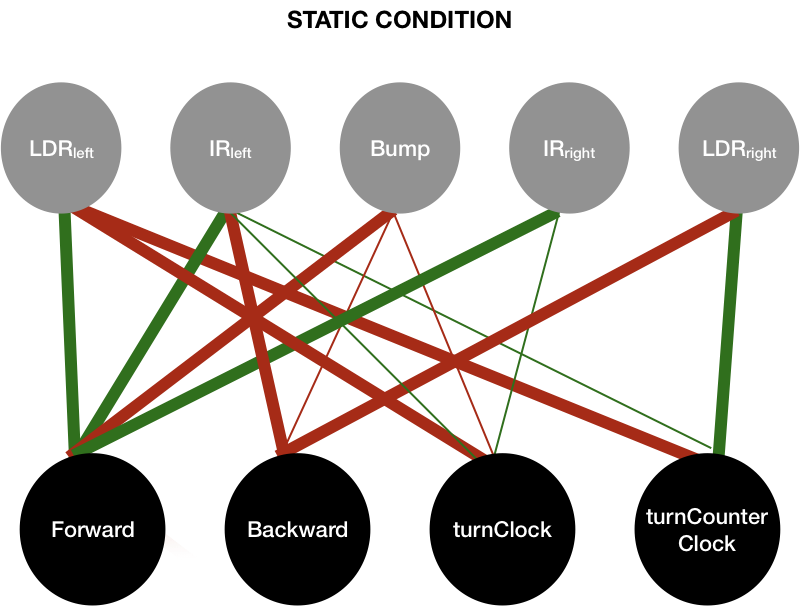
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Forward | Backwards | TurnClock | TurnCounter |
| leftLDR | 1 | 0 | -1 | -1 |
| leftIR | 1 | -1 | 0.5 | 0.5 |
| Bumper | -1 | -0.5 | -0.5 | 0 |
| rightIR | 1 | 0 | 0.5 | 0 |
| rightLDR | 0 | -1 | 0 | 1 |

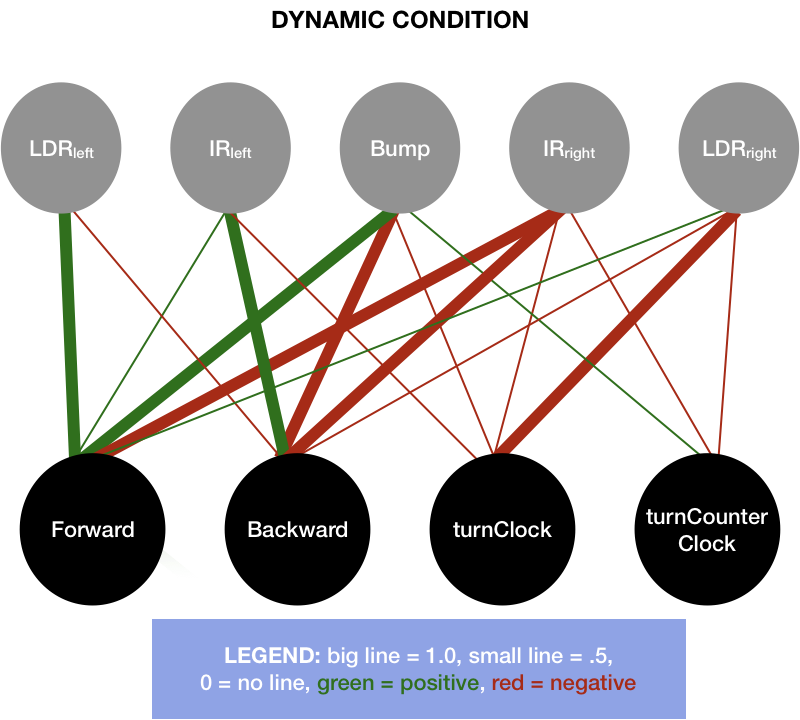
**Table 2.** Best performingfixed-topology network’s connection matrix for the dynamic condition.

Dynamic Condition

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Forward | Backwards | TurnClock | TurnCounter |
| leftLDR | 1 | -0.5 | 0 | 0 |
| leftIR | 0.5 | 1 | -0.5 | 0 |
| Bumper | 1 | -1 | -0.5 | 0.5 |
| rightIR | -1 | -1 | -0.5 | -0.5 |
| rightLDR | 0.5 | -0.5 | -1 | -0.5 |

**Best Performing Fixed Topology Neural Networks**

****

****

**Figure 12.** Visualization of fixed-topology GridBot’s evolved, best performing static and dynamic networks

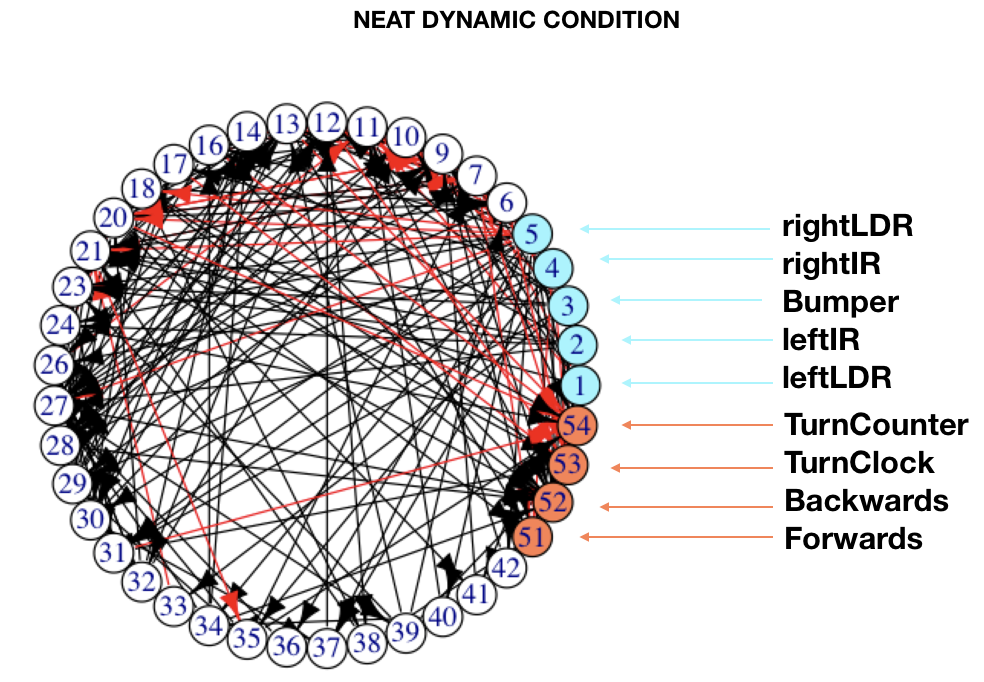
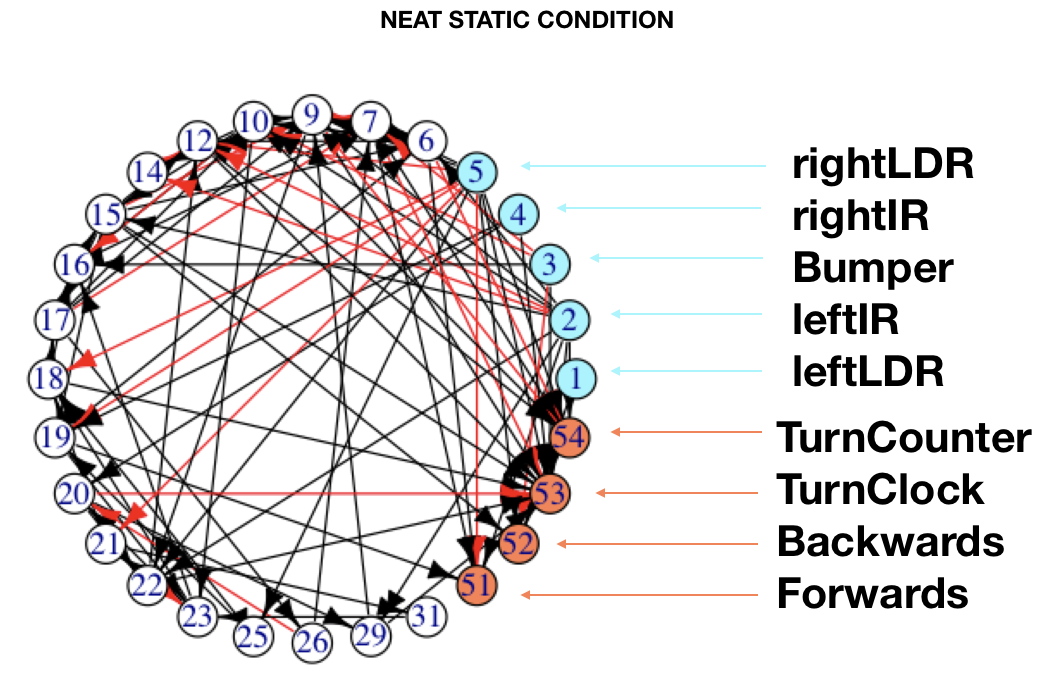
Both networks feature similar numbers of strong, excitatory, and zero connections (between 5-10% difference), but a 25% difference in weak and inhibitory connections (with the static condition having 25% less of each of these connection types compared to the dynamic condition’s network). See **Table 3.**

**Table 3.** Descriptive statistics of connection types in FTNN static and dynamic conditions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Strong (+/-1.0) | Weak (+/-0.5) | Excitatory (+) | Inhibitory (-) | Zero (0.0) |
| Static | 9 | 5 | 7 | 6 | 5 |
| Dynamic | 7 | 10 | 6 | 11 | 3 |

These networks have few things in common, and distinct features, as well. Both networks feature strong excitatory connections from the left LDR to the forward node, and excitatory connections between the left IR and the forward node (though the networks vary in connection strength). Additionally, both right LDRs inhibit the backwards node, which we would predict to be advantageous to the bot. Both feature varying strengths of inhibitory connection between the bumper and the backward node, which is counter intuitive, where we might expect to see the bumper exciting the backwards, or turning nodes. The bumper only excites the turn-counter-clockwise node in the dynamic condition, yet also excites the forward node. This would only be evolutionarily beneficial for a bot running into the barrier at the top center of the grid. The LDRs do not seem to excite their respective turn nodes (counter-clockwise for left and clockwise for right), which we might have expected to see. Instead, the right LDR in the static condition excites the turn-counter-clockwise node, which would raise the chances of the bot turning away from the light. There are many more interesting comparisons and features of these networks that are worth exploring, despite having only chosen a few noteworthy points to focus on here.

*Neuroevolution of Augmenting Topologies Neural Network Basic Analysis.* Our results yielded two distinct networks evolved using NEAT. The static condition’s best performer had a 4x26x5 network with 35 total nodes, and the dynamic condition’s best performer had a 4x37x5 with 46 total nodes. The weight matrices can be found in the ‘*neat\_static\_weights\_4\_15’* and ‘*neat\_dynamic\_weights\_4\_15’* files within the *finalists* folder of this project’s GitHub repository. The are visualized below in **Figure 13.**



**Figure 13.** Visualization of Best Performing NEAT Networks (utilizing visualization from Hunter, 2016)

**Table 4** shows descriptive statistics of output nodes of the networks above. These were calculated in the ‘*Plots.Rmd’* notebook of this project’s GitHub repository.

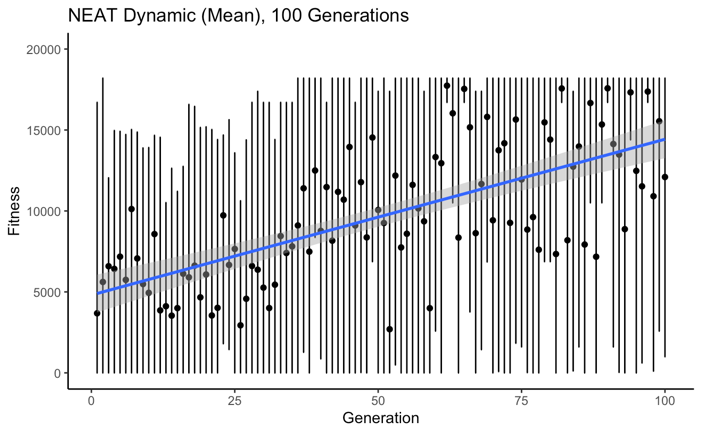
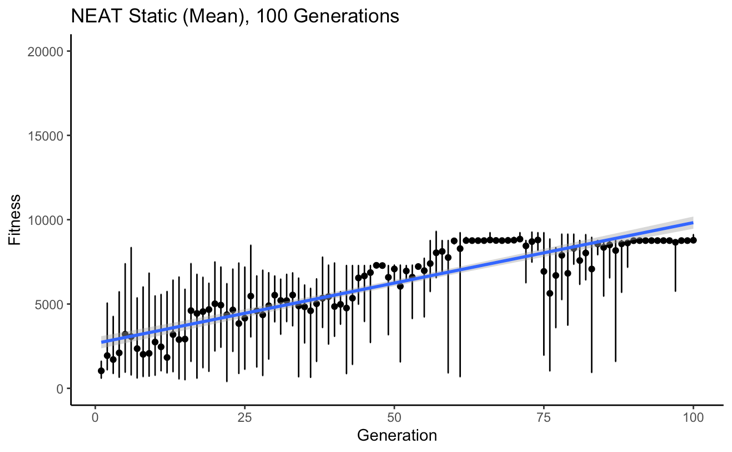
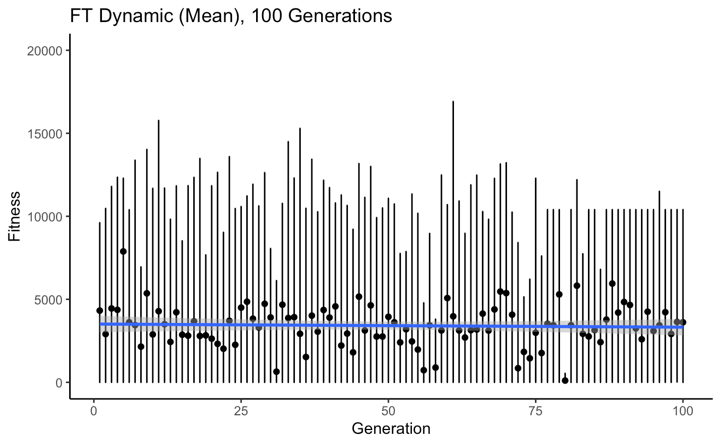
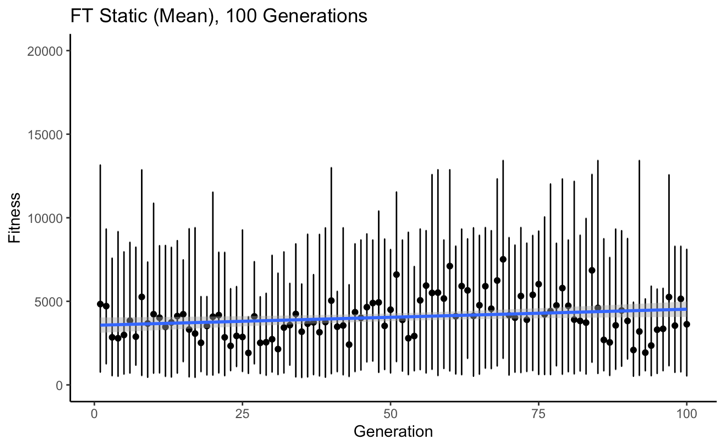
**Table 4.** Descriptive Statistics of NEAT NN Output Nodes

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Non-Zero | Strong (+/-1.0) | Excitatory (+) | Inhibitory (-) | Zero (0.0) |
| Static | 28 | 5 | 11 | 17 | 188 |
| Dynamic | 41 | 8 | 22 | 19 | 175 |

The total possible connections for the static network is 520 (4x26x5), whereas the total possible number of connections for the dynamic network is 740 (4x37x5). Both networks feature 216 possible connections to output nodes, which makes up 41% of the static condition’s total connections, and 29% of the dynamic condition’s total connections.

The percentages of non-zero connections to output nodes in the static and dynamic networks are around 5% of total connections for both networks, while no-connection makes up 36% of the static network and ~24% of the dynamic network. Both networks feature ~1% of connections as strong connections (1 or greater), ~2% of connections are positive, and ~3% of connections are negative.

1. Evolutionary Fitness



**Figure 14.** Fixed-topology and NEAT evolution across 100 generations. Each black dot represents the mean of that generation’s fitness, and each bar shows the max and min fitness for each generation. The blue line shows a best fit y~x regression.

The following analyses can be found in the *‘Plots.Rmd’* notebook of this project’s GitHub repository.

*FTNN Evolution Analysis.* We found that there was a significant difference between the FT static and dynamic mean fitness using a two-tailed Welch Two Sample t-test (t197.65 = -3.7093, p = 0.00027), and further investigated to find that the dynamic condition’s mean fitness was significantly less than that of the static condition (t197.65 = -3.7093, p = 0.00135). Additionally, we found a significant difference between the ranges, where the FT dynamic network’s ranges were significantly greater than the static network’s ranges (t195.8 = 7.9351, p = 7.937x10-14). We found that the fixed-topology network did not show improvement with respect to evolved fitness over time, actually showing a decrease (correlation = -0.044), in the dynamic condition, and only a slight increase in the static condition with a correlation of 0.235.

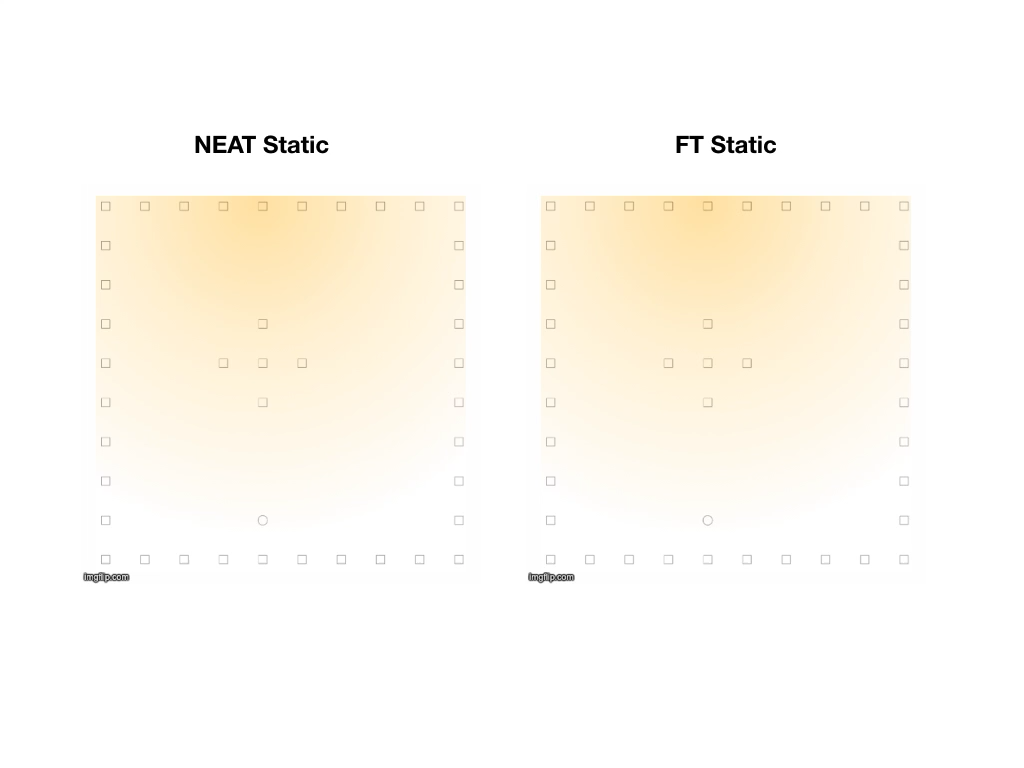
*NEAT Evolution Analysis.* We found that the mean fitness achieved by the NEAT dynamic network was significantly greater than that of the static network (t155.25=7.272, p = 8.171x10-12) using a one-tailed Welch’s Two Sample t-test. We also examined the range (max-min fitness) of the networks after seeing such stark differences between the static and dynamic condition’s ranges, and found that the NEAT dynamic condition’s range was significantly greater than the static condition’s (t158.58=23.76, p < 2.2x10-16), also using a one-tailed Welch’s Two Sample t-test. We also found that the correlation of the NEAT dynamic condition’s correlation between generation and mean fitness was 0.69, and the static condition’s was a striking 0.92.

Thus, we saw that NEAT yielded major improvement in GridBot’s fitness over time, compared to the slight and lack of improvement shown in the Fixed-topology network.

*Comparison.* We used a one-tailed Welch Two Sample t-test to examine the static condition in the FT versus the NEAT evolved network, and found that the mean fitnesses evolved were significantly greater in the NEAT version compared to the FT version (t149.54 = 8.687, p = 2.999x10-15). As for the dynamic condition, the same result was found, where the NEAT version’s mean fitnesses exceeded those of the FT version (t117.27= 14.702, p < 2.2x10-16). From this, we conclude empirically that our NEAT algorithm far exceeded the FT version in terms of mean fitness achieved over time.

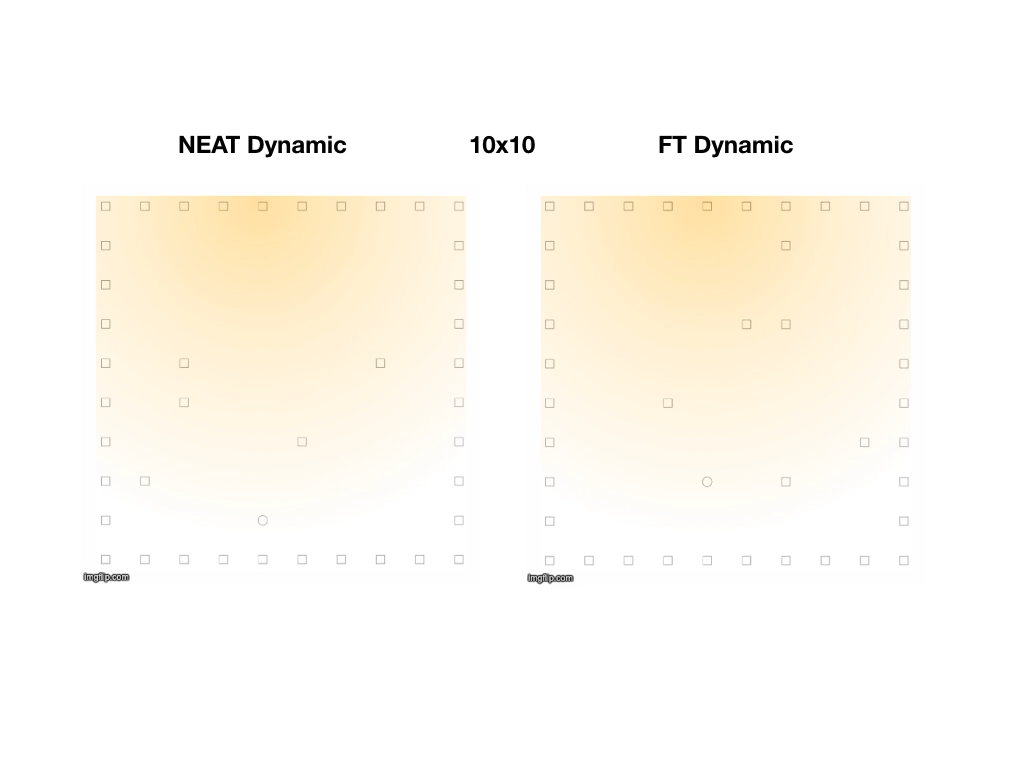
1. Paths

The following are visualizations of paths taken by the best performing individuals of the specified conditions. The overlaid light is meant to reflect the light gradient, but is not exactly to scale. Double click the video to play. The NEAT bot will go first and the FT bot will begin after a short delay. The labeled plain GIF files are located in the *final vis* folder of the GitHub repository, along with the labeled videos.



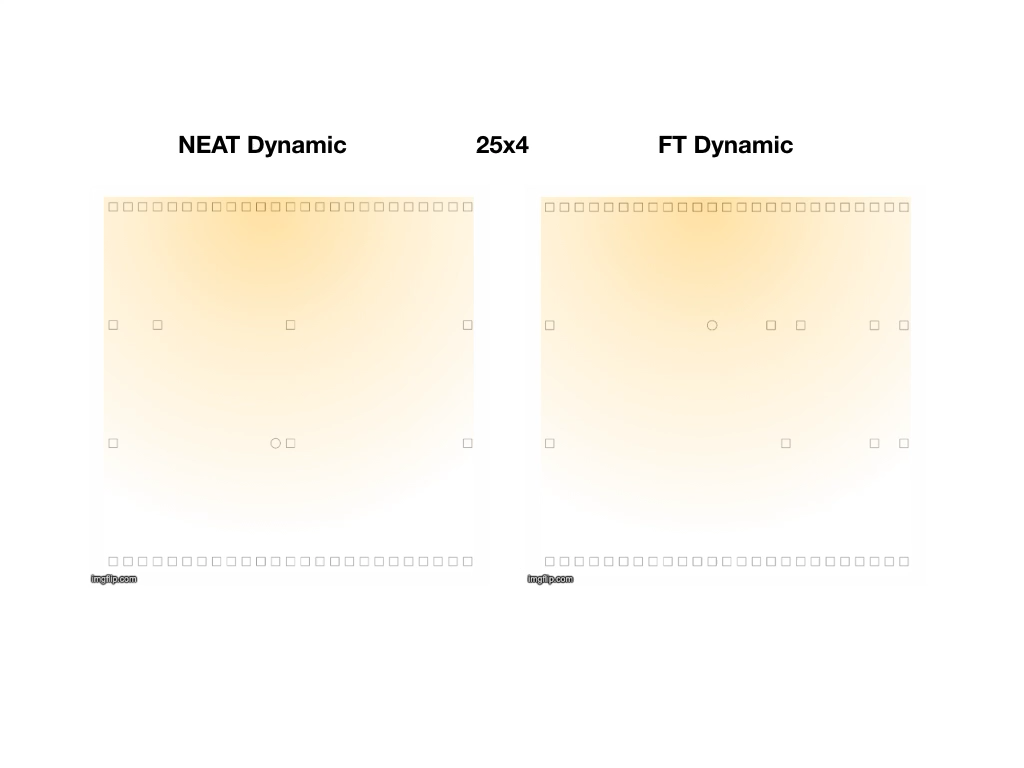
**Figure 15.** Visualization of paths taken by the best performing networks in the NEAT and FT static condition.

In **Figure 15** above, we see the NEAT bot is able to reach a zone of higher light value than the FT bot, which aligns with the generational and final results shown in **Figure 14.**We see the FT bot struggles to find a high light zone, getting stuck in corners, boundaries, and obstacles.



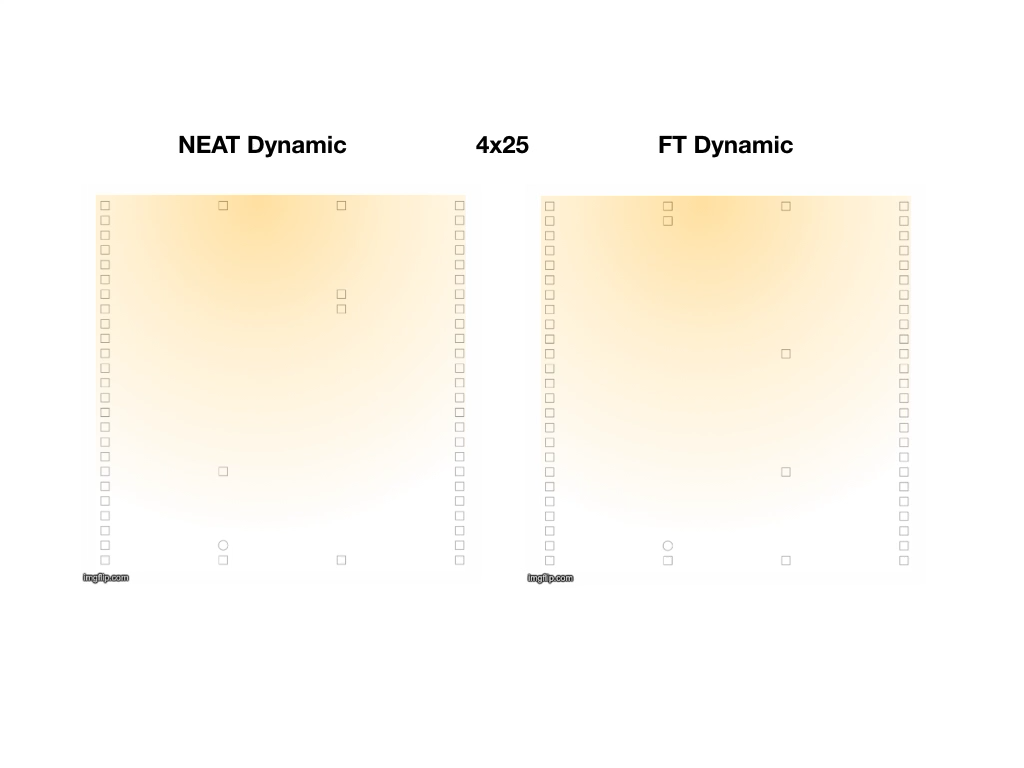
**Figure 16.** Visualization of paths taken by the best performing networks in the NEAT and FT dynamic 10x10 condition.

Both bots in this condition do very well, taking a direct path to the area of highest light concentration. In this trial, there happened to be no obstacles blocking the bots from taking direct paths forward, which works well for both of them. We note that the FT bot’s starting position is not where it should be (one space below where it is), and upon investigation, cannot pinpoint why this is occurring. This is an error we would examine in further research with this project.



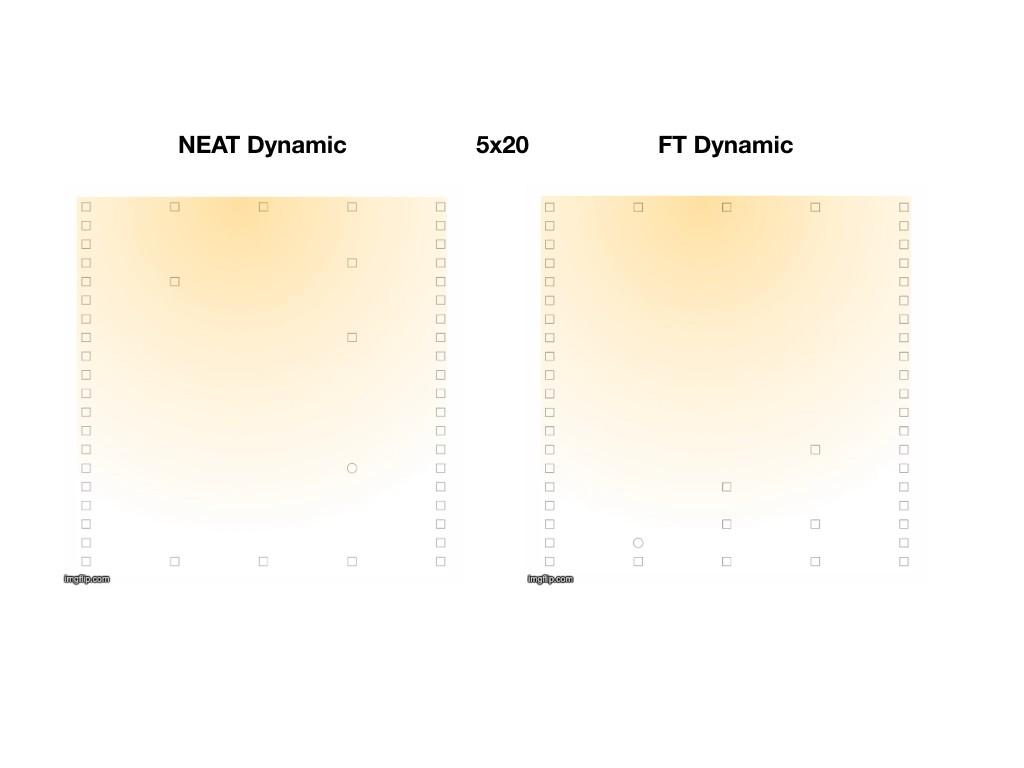
**Figure 17.** Visualization of paths taken by the best performing networks in the NEAT and FT dynamic 25x4 condition.

In the 25x4 board, the bots have an extreme advantage in many regards, as they only need to move forward one space to be in the optimal light space. However, if there is an obstacle there, they will need to move around it. Here, we again see an error in the FT bot’s starting place, which is again cause for concern. It’s starting position is the best position on the grid, which puts it at a major advantage. The nature of having access to such an “easy” board could explain the why NEAT dynamic condition bots were able to achieve such higher fitness than the static condition bots.



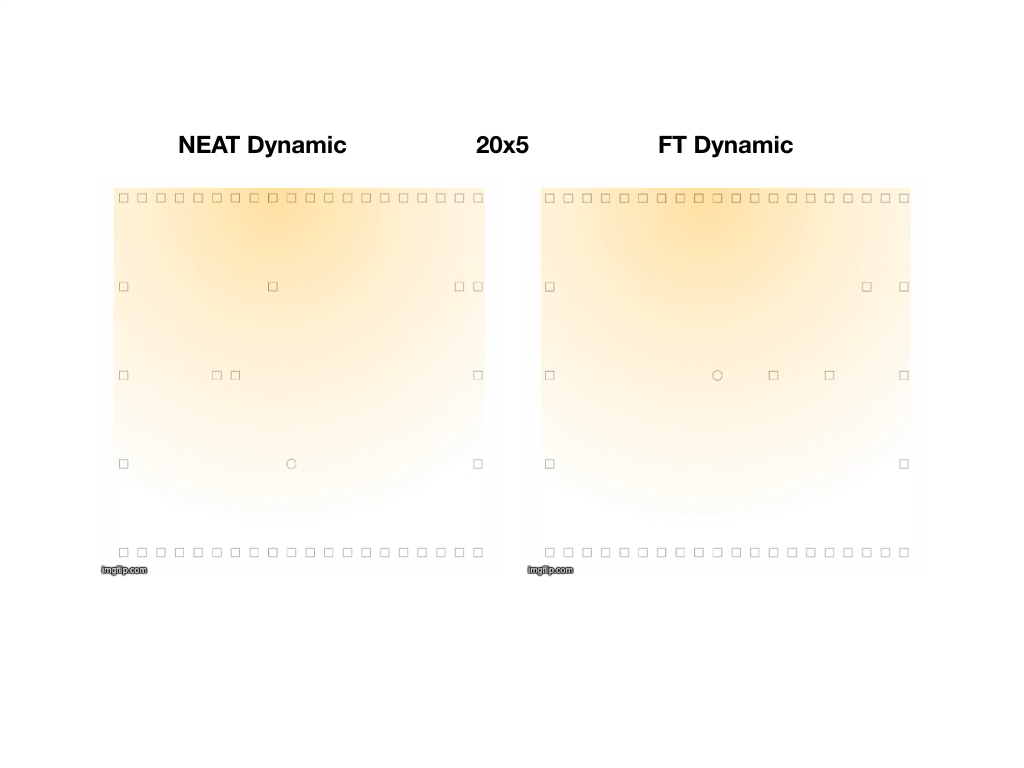
**Figure 18.** Visualization of paths taken by the best performing networks in the NEAT and FT dynamic 4x25 condition.

Both bots in this condition seem to struggle and are unable to reach an area of high light concentration. This board is challenging in requiring 20+ steps forward to reach high light concentration, especially with only the ability to move laterally only one space. Obstacles in this condition can create a very challenging task for the bot. This, perhaps, explains the occurrence of many low fitness bots occurring in the NEAT dynamic condition, compared to the static condition which did not have as many low scoring bots as evolution progressed.



**Figure 19.** Visualization of paths taken by the best performing networks in the NEAT and FT dynamic 5x20 condition.

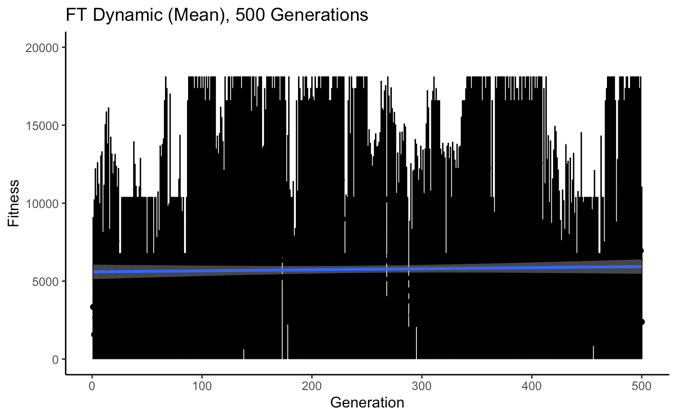
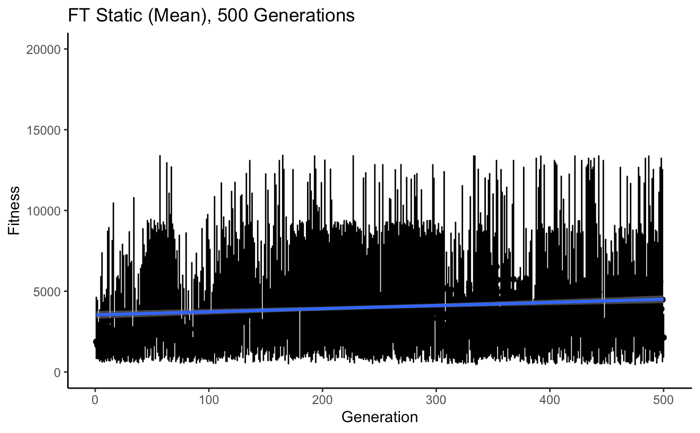
In this run, the FT bot appears to outperform the NEAT bot, finding the light source and staying near it. It is unclear without further analysis of the NEAT bot’s network why it acts in this way. The FT bot is impressive in its ability to get out of the obstacle trap, and to continue to the light source without shying away from it. We see a strong activation between the bumper and the forward node in the FT bot’s dynamic network, which could explain its end behavior. This shows us a good example of the challenging task set forth that both bots must face: using the bumper to avoid obstacles yet continue running into the wall when near the light source.



**Figure 20.** Visualization of paths taken by the best performing networks in the NEAT and FT dynamic 20x5 condition.

This is another example of an easier grid, as, if there are no obstacles, only a few steps forward are required to be in an optimal space. We see another erroneous placement of the FT bot here, yet in this case, it does move forward to the optimal spot. Both bots perform well in this case. The NEAT bot performs exceedingly well– it moves so quickly to the optimal spot, you might miss it!

1. Outstanding Questions and Post-Hoc Analysis
2. *Why didn’t the FT network evolve higher fitness?*

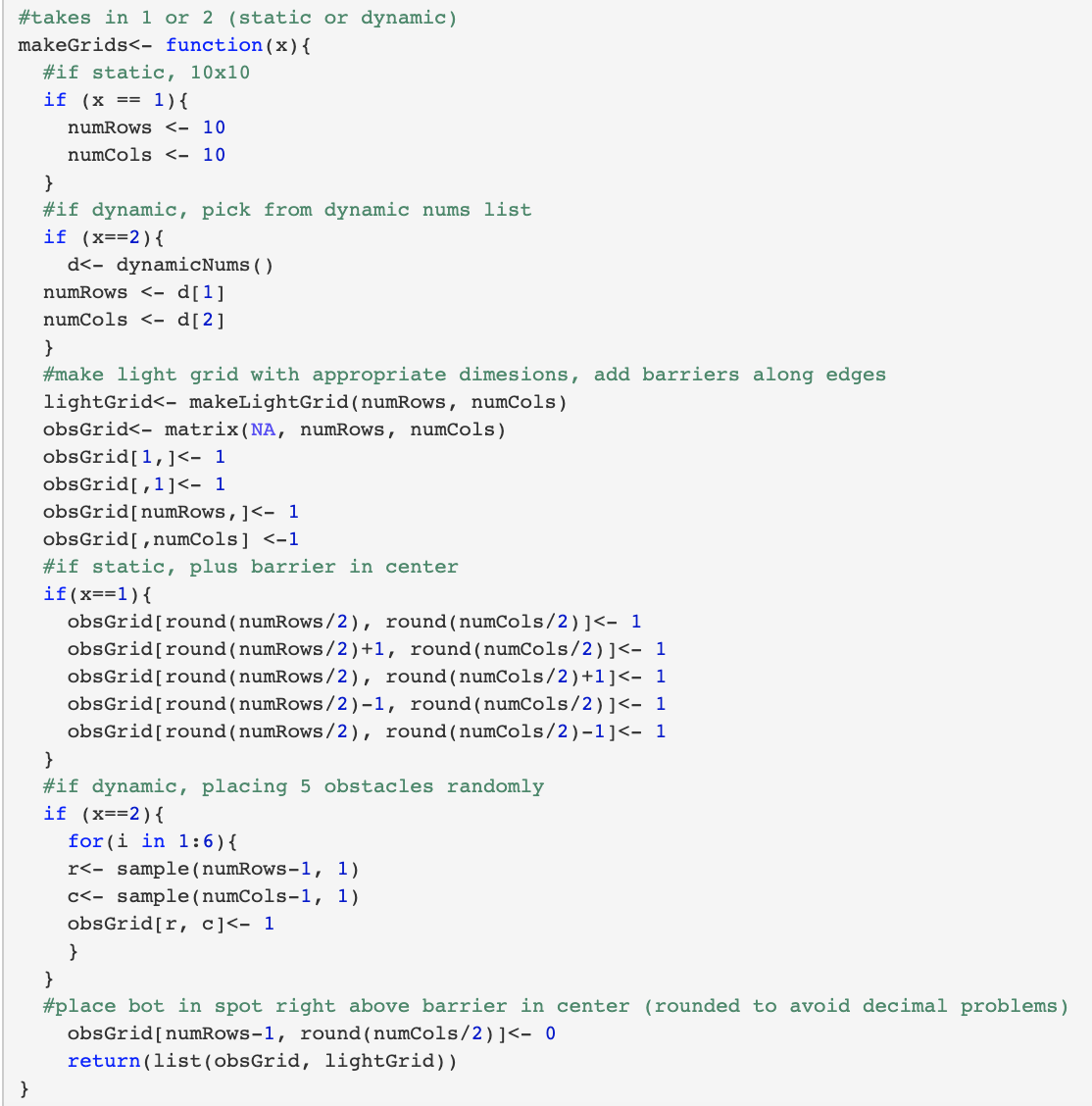


**Figure 21.** Fixed-topology evolution across 500 generations in the static and dynamic conditions. Each black dot represents the mean of that generation’s fitness, and each bar shows the max and min fitness for each generation. The blue line shows a best fit y~x regression.

We were curious to see if perhaps the reason why the FT network did not yield an improvement in fitness across conditions was that it simply did not have enough time to evolve. The NEAT algorithm allows a larger search space of possibilities than that of its counterpart. We decided to increase the number of generations from 100 to 500 to see if this would result in more fitness improvement. We found that over 500 generations, the dynamic condition yielded a correlation of 0.037, an improvement over the negative correlation in the 100 generation trial. The static condition yielded a correlation of 0.201, similar to the correlation found over 100 generations. This is conflicting evidence, because the dynamic condition showed improvement, where the static condition did not. So, we might conclude that number of generations does play a factor in the FT network’s evolution, but there are other factors involved that caused problems in our study.

1. *Why does the bot start in the wrong place in some NT dynamic grids?*

This is a major cause for concern in our study that we were unfortunately unable to pinpoint. The area of code that determines the bot’s starting position is the same for the NT and NEAT is shown below, in the *makeGrids* function of the ‘*GridBotv2.0.Rmd’* notebook (lines 37-75):



**Figure 22.** *makeGrids* function from ‘*GridBotv2.0.Rmd’* notebook.

At the end of this function, GridBot is placed, represented by a ‘0’ at *numRows*-1, *numCols/2.* This is the called in the *setUp* function, which is called in the *runBot* function in the FT bot’s code. *makeGrids* is called in *gridBot.InitialState* of the ‘*NEATGridBot.Dynamic.R’* and‘*NEATGridBot.Static.R’* scripts. So, it is very bizarre to find that only in certain FT dynamic conditions is the bot’s starting position one spot less than it should be (lower grid space index appears as one step forward visually on the grid), when all other conditions (FT static, NEAT dynamic, NEAT static) are all calling the exact same function and are all not experiencing the same error. This is the only line of code used to set up the bot’s starting position, and is used in all conditions.

It is important to make sure there is a level playing field for all individuals who are competing in order to compare their evolutionary fitness over time, and this issue directly contradicts this effort. This is cause for concern and a problem regarding our study.

1. Discussion

We examined GridBots evolved using fixed-topology networks and NEAT networks, and compared the best-preforming networks in a static and dynamic environment. We found that the NEAT networks far exceeded the FT networks in terms of mean fitness, and that the dynamic environments produced a vaster range of fitnesses (between max and minimum fitness of individuals in a given generation) than did static environments.

One possible explanation for the large ranges exhibited in the dynamic condition across FT and NEAT networks is the layout of the dynamic grids. The 20x5 and 25x4 grid require only a few steps forward to reach the best area in the grid, which could result in bots of very high fitness, compared to static condition bots who at least require double if not triple the steps forward to reach an area of comparable light. Equally, the 4x25 and 5x20 grid require double the steps of the static environment’s gird to reach the light source, which is costly in a world with a fixed number of moves allowed (100). The dynamic grids’ unique properties could also explain the higher mean fitness exhibited by the NEAT dynamic condition bots, as they allow higher fitness by requiring less moves to reach the optimal spaces of the grid, allowing more moves to sit and collect light, resulting in high fitness. Further research could improve upon our study by randomizing the start location of the bot, or by randomizing the location of the light source. This would make the task more difficult for the dynamic condition bots, which could perhaps serve as an equalizing force for the addition of easier grids that they receive the advantage of. However, as previously stated, it is of note that an equal number of boards are harder in the dynamic condition, requiring more moves to reach an optimal area of the grid. This could serve as an equalizing force in itself between the static and dynamic conditions. In comparing the resulting mean fitness achieved by each, it is inconclusive to say that it is “easier” or “harder” to evolve to succeed in the dynamic or static world, given the differences between them. Each world has its advantages, and it is clear that the environmental differences resulted in distinctions among the networks due to the different task demands of each condition.

We found structural differences between the NEAT networks, and differences in the connection strategies of the FT networks, though, without further analysis, we are unable to draw meaningful conclusions about these network’s modularity. Unfortunately, due to unforeseen project constraints imposed by quarantine, we were unable to calculate functional modularity as originally intended. This is further analysis that we encourage for future research, and have provided our data for any that wish to partake. The NEAT dynamic bot’s network appears more complex and less modular than the static bot’s, upon examining the visualization, but we would need to confirm this mathematically to definitively assert this. We would love to compare the emergent networks in terms of their modularity, and the similarities and differences in terms of which output nodes are activated or inhibited within the network, and how this translates to complex behavior, such as avoiding obstacles while still allowing the bot to run into the light source wall repeatedly. We did see this behavior emerge, which is an exciting feat.

This research confirms that we evolve with respect to the environments we are exposed to. One far reaching implication of our results could predict that in changing environments, the variability of success among individuals varies greater than in more stable environments, but, greater success is possible in these environments than in more stable environments. Of course, our GridBot exists in a simulated toy world, and fitness in this case is one simple metric. In the biological world, there is extreme complexity in what constitutes fitness and obstacles, and, there is the addition of agent interaction. Nevertheless, it is interesting to think about the broader implications of this theory, and it would be interesting to expand upon this research by examining more stable environments compared to more dynamic environments in animals or human culture.

Chapter 4: References

Cappelle, C. K., Bernatskiy, A., Livingston, K., Livingston, N., & Bongard, J. (2016). Morphological modularity can enable the evolution of robot behavior to scale linearly with the number of environmental features. *Frontiers Robotics AI*, *3*(OCT), 1–10. https://doi.org/10.3389/frobt.2016.00059

Clune, J., Mouret, J. B., & Lipson, H. (2013). Summary of the evolutionary origins of modularity. *GECCO 2013 - Proceedings of the 2013 Genetic and Evolutionary Computation Conference Companion*, 23. https://doi.org/10.1145/2464576.2464596

Gomez, F. J., & Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. *IJCAI International Joint Conference on Artificial Intelligence*, *2*, 1356–1361.

Gruau, F. (1995). Automatic Definition of Modular Neural Networks. *Adaptive Behavior*, *3*(2), 151–183. https://doi.org/10.1177/105971239400300202

Haddow, P., & Tufte, G. (1999). Evolving a robot controller in hardware. *Proc. of the Norwegian Computer Science Conference*, (November), 141–150. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.7078&amp;rep=rep1&amp;type=pdf

Huizinga, J., Mouret, J. B., & Clune, J. (2014). Evolving neural networks that are both modular and regular: Hyperneat plus the connection cost technique. *GECCO 2014 - Proceedings of the 2014 Genetic and Evolutionary Computation Conference*, 697–704. https://doi.org/10.1145/2576768.2598232

Hunter, Andrew, Neuroevolution of Augmenting Topologies – NEAT, (2016), GitHub repository, https://github.com/ahunteruk/RNeat

Livingston, N., Bernatskiy, A., Livingston, K., Smith, M. L., Schwarz, J., Bongard, J. C., … Long, J. H. (2016). Modularity and sparsity: Evolution of neural net controllers in physically embodied robots. *Frontiers Robotics AI*, *3*(DEC), 1–16. https://doi.org/10.3389/frobt.2016.00075

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99–127. https://doi.org/10.1162/106365602320169811

<https://www.youtube.com/watch?v=b3D8jPmcw-g>