NeuroEvolution of Augmenting Topology for Capture the Flag Pacman:

AI Final Project Report

Edward Xie, Jackson Neal
Khoury College of Computer Sciences
Northeastern University
Boston, USA
xie.e@northeastern.edu, neal.ja@northeastern.edu

***Abstract*—This report covers the motivation, implementation, and analysis of the final project for CS 5100 Foundations of Artificial Intelligence. The purpose of the paper is to share the process and results of our project and comment on further related exploration. Our project concerns the development of a competitive agent that plays the Capture the Flag variant of the Berkeley Pacman program. The agent utilizes a neural net to evaluate the game state and choose actions. We improve our agent through genetic optimization and employ NeuroEvolution of Augmenting Topology to evolve the neural net of the agent. This paper is intended to provide an overview of our chosen task, an in-depth look at the application and implementation to solve this problem, and discussion on the results of our implementation.**

Keywords: NeuroEvolution of Augmenting Topology, Pacman, Genetic Optimization, Neural Net, Competitive Agent.

I. Introduction

We were initially motivated to develop an agent that could participate in a competitive game. Our intent was to train agents against each other and hopefully discover complex behaviors through this process. After exploring various combative and competitive options, we decided to pursue an agent capable of playing Capture the Flag Pacman. This game is a variation of the Berkeley Pacman program and has the same basic structure. The first key difference in this program variant is that the game involves two teams of two agents. The second difference is that the game requires a Pacman to travel to the enemy team's territory, eat food, and then return to its side to "capture" the food in order to score. The teams compete against each other, trying to capture the most food. Agents within their own territory can eat members of the opposing team. Agents that have eaten a capsule in the offensive territory can eat enemy ghosts in the enemy territory. In the sample Figure 1, the red Pacman must eat the blue dots and then return to the red side.

This problem choice was ideal for several reasons. Importantly, this program was open source. Although we may have liked to pose a problem of greater personal interest, such as the development of a Mortal Kombat player agent, the accessibility of the Pacman program was undeniable. Furthermore, we were already familiar with the Berkeley Pacman program and understood how to develop simple agents for the program. This prior knowledge of the task allowed us to dedicate most of our time to exploring a novel algorithm and agent development instead of spending time with program familiarization and technical setup. We also believed that this variant of Pacman provided enough complexity to pose a challenging agent development. The team aspect and scoring by capture transforms Pacman into a complex, competitive, cooperative, and strategic undertaking. We were excited by the prospects of developing advanced behaviors by training and testing our agents in this environment.

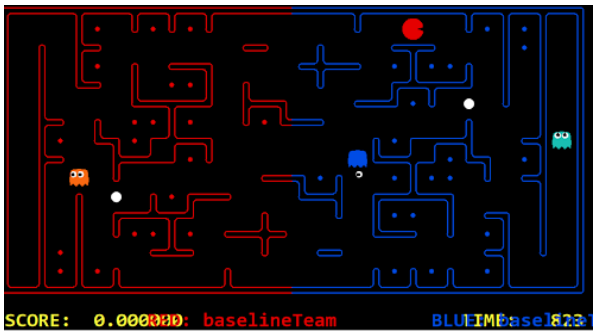

Figure 1: Capture the Flag Game Board

After selecting our problem of Capture the Flag Pacman, we defined the technical aspects of our task. We decided that our agent would utilize a neural net to evaluate the board state and play the game by outputting Pacman actions. We researched methods for developing and improving neural nets and decided to implement NeuroEvolution of Augmenting Topolgy (NEAT) to evolve our agent. This algorithm poses an innovative method of neural net development that we were eager to explore. The subsequent sections detail the implementation of our agent, the application of NEAT to improve our agents, and discussion and analysis on our findings.

II. Implementation

A. Agent Inputs and Outputs

Our agent is designed as a neural net that takes the game state as input and chooses a Pacman action. When choosing inputs to our neural net we aimed to encode all relevant information from the visible board as well as information that

may be necessary for strategic capture the flag play. We first encode the board locations as input nodes with values that indicate if that board location is either blank, a wall, has food, or has a capsule. The capture the flag board dimensions are 16x32, so the basic board state lends 512 input nodes for our neural net agent. We then add an additional 8 input nodes corresponding to the x and y coordinates of the four agents playing the game. Over the course of training agents, we added one additional input node corresponding to the number of food pieces currently being carried but not yet captured by the agent. This input value could theoretically be learned using recurrent connections, but we wanted it readily available as a strategic agent should behave differently if it is carrying food with intent to capture. We added a final input node with a value encoding whether the opposing team is scared, indicating that the agent's team has eaten a capsule and can eat the other team (again theoretically learnable). This totaled to 522 input nodes for our neural net agent. The output nodes were derived from the Pacman program action space. Pacman may take movement in any of four directions or stop. This action space naturally defined five output nodes for our neural net agent. After feeding information through our neural net - which we will cover in more depth in the following section - our agent executes the legal action that has the highest corresponding output node value.

B. NEAT Algorithm[1]

NEAT is a framework for developing neural networks through genetic optimization [1]. In this algorithm, as in typical genetic algorithms, a population of individuals is evolved over many generations. Each individual in the population contains genes that correspond to a neural net. In each generation, every individual is scored by a fitness function. The more fit individuals are then chosen to contribute offspring to the next generation population. When generating the next generation of individuals, there is also a series of crossover and mutation operations. Crossover produces novel individuals by combining features of two individuals. Mutation introduces novel individuals by administering random alterations to an individual's genes. NEAT provides several innovations in addition to the typical genetic optimization approach that allow genetic optimization to function with a neural net gene representation which we will explore below.

One important feature of NEAT is that it optimizes for minimal neural nets. This is accomplished by initializing our population of individuals with genes that represent a bare neural net of only input nodes and output nodes. This means that each individual of the initial population will have no hidden nodes or connections. Connections and hidden nodes can only be added to the gene topology through mutations. We expand on the exact implementation of mutations below. Through this initial population, we prevent network bloating by only maintaining growth that outperforms smaller networks.

An integral feature of the NEAT evolution process is the concept of speciation. Speciation separates NEAT from other standard genetic algorithms and protects innovation to avoid early population convergence and stagnation. In order to divide our population into species, NEAT defines a compatibility function (details on this later) to measure the similarity between genes. We can then define a hyperparameter, such that when genes are not sufficiently compatible, they should be placed in separate species. Each generation, every pre-existing species will have a random representative selected. The species is then emptied and prepared to receive new offspring. Each offspring is placed into the first species whose representative is sufficiently compatible with the offspring. If no representative is compatible, a new species will be created for that offspring.

After species are created, their development is protected through several steps. The first step is fitness sharing. Each individual's fitness score is adjusted by dividing it by the size of the species of the individual. Species then produce a number of offspring proportional to the sum of the species' individual's fitness scores over the total fitness sum of the population. These steps prevent any single species from becoming too large and dominating the entire population.

Within NEAT, a gene is represented as a set of nodes and a set of connections between these nodes (Figure 2). A node is either an input, an output, or a hidden node. Nodes are labelled by index, the first node is a bias node (an input node whose value in always 1), followed by the state input nodes, the output nodes, and the hidden nodes. A connection is defined as a tuple of the incoming node index, the outgoing node index, the weight of the connection, whether the connection gene is enabled, and the innovation number that produced the gene (more on innovation number soon). The outgoing node cannot be an input node. The connections of the genes define a neural network.

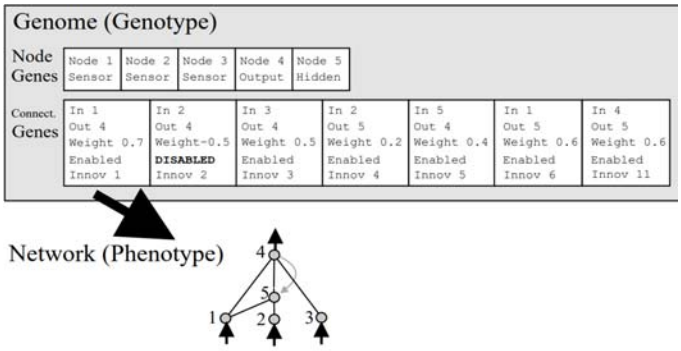

Figure 2: Example genome and corresponding network [1]. Note the recurrent connection from node 4 to node 5.

Data is fed through the network first by copying the input values into the input nodes. For the non-input nodes, its value is set to tanh of the sum of the incoming connected nodes' values multiplied by the connections' weights. This feeding of

[1] Our implementation is available at https://github.com/edwardx999/PacmanAI.

values is done first for the hidden nodes, and then the output nodes, in order of node index. The original NEAT algorithm uses a sigmoid activation function; we chose to employ tanh after observing faster convergence in simpler tests. The values in the neurons after feeding are maintained in order to facilitate recurrent connections, though such connections may be disabled (an extension we made to the original NEAT).

After each species provides its proportional number of offspring for the next generation, the offspring then have a probability of going through mutation and crossover before being placed in the population. NEAT also defines methods for executing crossover and mutation on the previously explored gene representation. As stated earlier, these mutation and crossover procedures are the essential methods of generating new genes. Three gene relations are defined for these operations: shared, disjoint, and excess. If a gene is selected for crossover, we will select a mate from the same species to serve as the alternate parent. During crossover, a shared gene is a gene with the same innovation number and present in both parents. A disjoint gene is a gene found in only one parent that has a lower innovation number than the maximum innovation number found in the genes of the alternate parent. An excess gene is defined as having a higher innovation number that the maximum innovation number of the alternate parent. Disjoint and excess genes are always inherited from the more fit of the two parents. If the parents are equally fit, each has a 50% likelihood of contributing that gene to the offspring. Shared genes are randomly inherited from either of the parents. The crossover process gives inherited disabled genes a chance of being re-enabled.

The gene structure also allows us to define the compatibility function to determine speciation. The compatibility function is comprised of three parts: number of disjoint genes between individuals divided by total number of genes, number of excess genes between individuals divided by total number of genes, and the average weight difference between shared genes of the individuals. These terms are each given a hyperparameter cofactor and then summed to construct the compatibility function.

There are several types of mutations in the NEAT algorithm. The probability of each mutation is independently configured by a hyperparameter. A new connection mutation connects two nodes with a random weight. A perturbation mutation modifies connections weights by selecting from a normal distribution around the current connection weight or resetting the weight completely to a random new value. A new node mutation splits an existing connection, such that a connection leads from the old incoming node to the new node, and another connection leads from the new node to the old outgoing node. The old connection becomes disabled during this mutation type. The final mutation type is to disable or enable a gene. We also chose to add an additional hyperparameter to control whether mutations are allowed to generate recurrent connections, which would theoretically improve the likelihood of a beneficial mutation when output should only be dependent on the current input. Whenever a mutation occurs that produces new connections, we track that gene with a global innovation number so that we can identify the gene in other individuals during crossover and speciation compatibility operations.

NEAT also offers a few supplementary evolutionary strategies that we chose to implement. For instance, to protect against stagnation, a non-improving species is not allowed to produce offspring after a certain number of generations configured by a hyperparameter. Additionally, there is a hyperparameter providing a small chance of interspecies crossover to produce extra novel offspring. NEAT protects against deterioration by automatically copying the best individual from sufficiently large species into the next generation.

C. Fitness Function

Given that our end goal is to have an agent that is a competitive Pacman Capture the Flag player, our fitness function is derived from the game score. We adjust the game score to provide fractional points for eating food, eating other ghosts, and eating capsules. These additions promote more simplistic behaviors than the highly complex capturing of food but are given very little weight in the fitness function so that our agent can primarily learn to win the game. Our final fitness function was the square of the sum of the number of food pieces captured, number of food pieces eaten times 0.1, the number of capsules eaten times 0.15, and the number of ghosts eaten times 0.05. This fitness function was reached through experimentation with the idea that rewarding simple behaviors could build complex ones, while providing substantially greater reward for scoring so that individuals that score are prioritized when creating offspring in the next generation. Our fitness function development was continuous throughout our research process, taking many small adjustments to properly promote these simple behaviors (see analysis). To get a fitness score, each generation, every individual plays a game against the best individual from the preceding generation (the remaining two players are controlled by defensive reflex agents provided by the Pacman game in order to simplify training). We mandated that every agent play against the same opponent to have consistency of scoring.

D. Technical Implementation

After the exploration of NEAT, we took on the task of implementing the algorithm and evolving an agent population. The open source Capture the Flag Program was written in Python, so we continued to develop in that language. After having suspicions on the performance of the code, we then employed Cython and saw dramatic improvements in speed. We also implemented our algorithm to allow for parallel processing of individuals during fitness evaluation which was completely necessary for successful evolution in a practical time period.

III. Results

A. Warm-up Exercises

In order to confirm the soundness of our implementation, we set to testing our optimizer on several simpler problems: xor, cartpole, and mountain car. Xor was demonstrated in the NEAT paper and the latter two exercises come from the OpenAI gym[3]. Starting with no connections and a population of size 150, we can see xor converge in around 40 generations[2], i.e. reaches a fitness above 5.95, where fitness is defined as six minus the total squared error. Regarding cartpole, a near perfect agent is found within 10 generations. Fitness was time balanced (environment reward) minus average deviation from center. With the mountain car problem, the solution is reached in under 10 generations. Fitness was environment reward plus maximum height reached. (See analysis for why environment reward purely could not be used.) Additionally, these problems are typically solved using a minimal network, for xor: 1 hidden node, and for mountain car and cartpole: no hidden nodes.

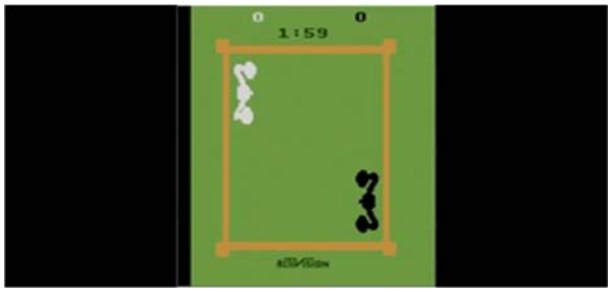

Figure 4: Atari Boxing Exercise

The algorithm was additionally applied to two Atari games also from the OpenAI gym, where the input was the 128 bytes of RAM and the output was an action in the game – to be acted upon for 2, 3, or 4 frames randomly. In the boxing video game (Figure 4), the agent quickly learns to win the game and reaches a peak fitness at around 80 generations of optimization (Figure 5), with the most fit individual having 15 hidden nodes and 58 connections. In the game of Asteroids, a peak fitness was reached in approximately 150 generations (Figure 6), with the most fit individual having 171 hidden nodes and 642 connections. Fitness was game score in both instances.

These exercises demonstrated the ability of NEAT to solve simple problems and provided promising results for the ability of NEAT to solve more complex environments than had been demonstrated in the original paper.

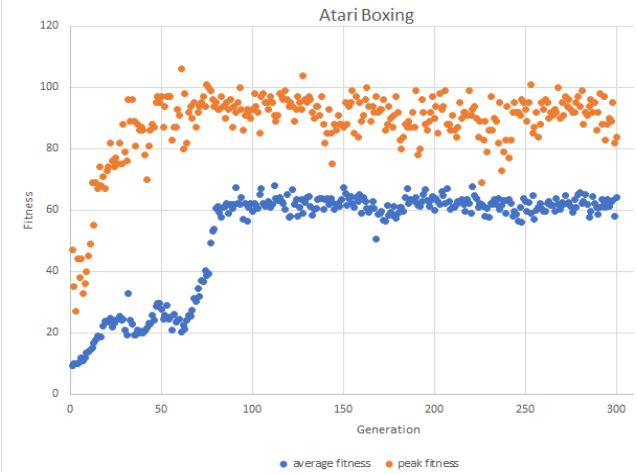

Figure 5: Atari Boxing, peaks at ~80 Generations. A random agent typically scores around 20.

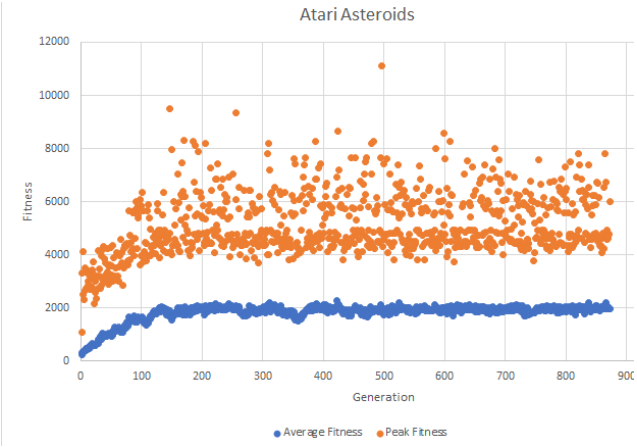

Figure 6: Atari Asteroids, peaks at ~150 Generations. A random agent typically scores around 1000.

B. Empirical Results

Regarding the agent development in the Capture the Flag Pacman game, we were able to see the development of several interesting behaviors. An optimized agent was expected to perform better than a random agent, which usually scores 0, but we did not predict the ultimate extent of its ability. In our most advanced evolution trials, we ran a population of 1,000 agents for 3,000 generations, involving several changes to the fitness function (see analysis). At the end of this optimization, one of the most fit agents has 156 hidden nodes and 628 connections. The most fit agents consistently ate between 4 and 8 food from the enemy territory and typically captured 4 pieces of food. Agents consistently learned the ability to smoothly navigate the maze and reach enemy territory. The most evolved agents

[2] The hyperparameters are too numerous to list here but are visible in the repository in the respective files for each test.

learned the behavior of evading ghosts. When deep into the enemy team's territory, our agent can evade in any direction to avoid being eaten. The agent also learned to attack with caution – avoiding crossing into enemy territory when a ghost in in range, trying new points of attack, and waiting until enemies have left the area before making a move. After 3,000 generations of evolution, we see the Pacman agents stagnate. At this point, the best individual from each generation scores between 4 and 8 points, but, and does not continue to improve on a generation to generation basis.

C. Analysis of Empirical Results

We believe there are clear explanations for the behaviors we see and those that we never saw developed. It is evident that our agents learned to travel to the opposing side of the map. This is important because nearly all scoring opportunities require action on the opposing side, and we want to develop highly competitive agents. When eating food, our agent consistently ate the same few pieces. Although eating these pieces was promising, it was not enough to guarantee a win. It is likely that the large number of inputs and large networks did not allow the agent to learn to eat food in all positions. Further and potentially more careful evolution would be necessary to develop this more generic behavior.

Disjoint Gene Cofactor	1
Excess Gene Cofactor	1
Weight Difference Cofactor	1
Compatibility Threshold	4
Perturbation Chance	0.8
Perturbation St. Dev.	0.1
Reset Weight Chance	0.1
New Connection Chance	0.3
Force Bias Connection Chance	0.01
New Connection Weight St. Dev.	1
New Node Chance	0.03
Disable Mutation Chance	0.1
Enable Mutation Chance	0.25
Allow Recurrent Connections	True
Intra-Species Crossover Chance	0.70
Inter-Species Crossover Chance	0.05
Species Stagnation Limit	15

Figure 7: Initial Set of Hyperparameters for Pacman.

The agent's most impressive learned behavior was that of avoiding ghosts. This behavior was likely learned because being eaten resulted in a penalty and being sent back to the start position which hinder further scoring and result in a lower fitness. The behavior of avoiding ghosts led our agent to develop a hesitant but consistent behavior. When playing our agent against an offensive reflex agent (provided by the game), this behavior is highlighted. The offensive agent plays aggressively and attempts to eat as much food as possible, often resulting in being eaten by ghosts many times. Since our agent consistently scored a few points through capture and very rarely was eaten, our agent can occasionally beat an offensive reflex agent.

Our agents did not learn to eat other ghosts after eating a capsule. This behavior is rather complex as it involves a series of moves and requires the disregard of the agent's learned avoid ghost behavior. Perhaps it is not advisable to have the agent attempt to learn to avoid and eat ghosts simultaneously. It may have been more productive to optimize agents to develop one behavior and then attempt to learn the next. The agent's skill in capturing food stagnated and did not appear to be fully intelligent. Although we saw our best agents score every game, they often only retreated to their side when under pressure from enemy ghosts. Again, this concept of returning to their side to score is complex and counterintuitive as all the food pellets are in enemy territory. We attempted various levels of speciation, mutation, and exploitation but were unable to evolve past stagnation and achieve these complex behaviors. They may have been attainable, but time and resource constraints prevented us from fully exploring what we would have liked.

It was found that developing a continuous fitness function is essential. In our initial attempts, the fitness function was purely game score. Thus, in the beginning, when agents are very simple, it was impossible to determine which agents were better (i.e. nearer capable of scoring). This was true for the cartpole and mountain car problems as well. In order to encourage movement to the other side, fitness was given to agents who were able to move away from the start. However, this initially only created agents that ventured far to the opposite corner. The jump could not be made to eat food and then return to the other side to score. Further enhancements were made to the fitness function to encourage picking up food, eating ghosts, and eating capsules. These enhancements finally enabled our agents to score in the game (although not to eat ghosts). However, such micro-enhancements to the fitness function are hard to develop and so a game requiring such complex behavior as Pacman may be unsuitable. If the fitness function is not continuous, the mutations necessary for a network to learn complex behaviors are not encouraged and too rare. A more advanced and granular fitness function may be needed to gradually optimize for the complex behaviors. The fitness function should provide greater differentiation between agents by encoding the quality of gameplay and move sequences taken during the match, instead of only evaluating final score.

D. Related Work

We were partially inspired by an application of NEAT to play Super Mario N64 [2]. The application of NEAT to this game used the game screen as input and allowed a neural net to control Mario. The application of NEAT to Mario was quite successful. There are a few key differences between our application and that of Mario. The first is that the game of Capture the Flag Pacman is much more complex than Mario. In Mario, an agent must simply learn to go right and not die – going further and doing it faster. In Capture the Flag Pacman, there is much greater complexity in the movements required to navigate and capture food as well as to coordinate with a teammate and avoid intelligent enemies. Pacman requires global information, while Mario can generally make decisions based on only its immediate surroundings. The second key

difference is that Super Mario lends itself very naturally to a continuous and consistent fitness function. Within Super Mario it is very easy to evaluate and compare agents since a granular fitness function can be derived from the distance traveled by the Mario agent and the time taken to reach that point. As explained prior, deriving such a fitness function for Capture the Flag Pacman is not immediately clear and should be explored in the future. Our application of NEAT provides a problem of greater complexity than we had seen attempted elsewhere.

E. Future Exploration and Improvements

A severe crutch of genetic algorithms is the number of hyperparameters involved. Our algorithm is no exception. Tuning of hyperparameters is problem specific and requires time and resources for repeated evolutionary runs that we were not able to achieve. Future exploration could experiment with various rates of mutation, measurements of compatibility, and cutoffs for stagnation measurements. These, among others, contribute greatly to the success of the NEAT algorithm and make it difficult to assess the performance of the algorithm until they are analyzed.

A problem with our current algorithm is that it tends to stagnate when attempting to find complex behavior. We believe this is because, as the network grows larger, the probability of a mutation occurring that induces a correct novel behavior, or which even affects the behavior and fitness, becomes very low, particularly with a non-continuous fitness function. The current algorithm cannot associate fitness to an action in a specific state (fitness is assigned after a game is completed) and "fix" the action. In order to solve this problem, the algorithm could be combined with Q-learning. In this new algorithm, instead of evolving towards a higher game scores, the algorithm could learn to approximate a Q-function. This would provide a clearer continuous fitness function: the loss function. Alternatively, instead of the purely random mutations of NEAT, a method could be developed to produce mutations that are more likely to induce the correct behavior, perhaps based on identifying where agents are stagnating. If combining this strategy with Q-learning, the perturbation mutation could be replaced with gradient descent, since fitness would then be a differentiable loss function, in a manner similar to deep Q-learning.

A possible extension to NEAT would be adapting it to other topologies, such as convolutional networks. This would necessary for large inputs such as images. In addition, this could help with learning to generalize behavior. For example, our agents tended to navigate along the top of the map. As it is now, NEAT might be required to evolve a duplicate set of connections for the inputs in the lower part of the map in order to navigate that section.

IV. Conclusion

The overall result of the project was successful. We developed a competitive Pacman agent, explored a new algorithm, and developed an implementation that demonstrates the power of evolved neural nets while opening the door to further exploration.

Although the agent was not unbeatable, our Pacman Capture the Flag agent developed novel behaviors and some that we consider complex. The ability of the agent to avoid ghosts, smoothly navigate the maze, and patiently await moments of attack may very well exceed that of a human player. We believe that with further optimization and the steps we laid out in future explorations, the agent could develop into an unbeatable player.

The most rewarding portion of the project was the exploration into NEAT and the development of neural networks. Although genetic optimization and neural nets were familiar topics, the application of optimizing networks and constructing networks of unique topology from scratch was intriguing. As advice to future students, it is important to develop logging and saving early. It is easy to waste a lot of time training to lose your data and progress.

Acknowledgment

Thank you to Professor Platt for a great semester!

References

[1] K. Stanley, and R. Miikkulainen, "Evolving Neural Networks Through Augmenting Topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99-127, Jun. 2002.

[2] S. Bling, "MarI/O – Machine Learning for Video Games," *YouTube*, Jul. 13, 2015. Available: https://www.youtube.com/watch?v=qv6UVOQ0F44, Accessed on: Dec. 14, 2020.

[3] G. Brockman, et al. "OpenAI Gym," *arXiv preprint arXiv: 1606.01540*, Jun. 2016.