# Developing Controllers for GVGAI

Paul Baird-Smith, Aastha Tripathi, and Tiffany Yang

December 20, 2017

**Abstract**

General intelligence has taken large steps in recent years, and now much research in the field occurs in video games. GVGAI offers a perfect platform for testing potentially general intelligences in video games. To this point, most approaches in developing agents that perform well in the GVGAI competition have used strategies that are effective in narrow intelligence problems (like winning chess or go), but many of these techniques have yet to be tested in this domain. In this project, we introduce controllers using 3 different techniques: neuroevolution, a critic variant of a Monte Carlo tree search algorithm, and Q-learning. We explore the benefits and drawbacks of each, and discuss which games are played better by which agents.

## 1 Introduction

Games appear to be an ideal domain for realizing several long-standing goals of AI, including affective computing, computational creativity and ultimately general intelligence. However, previous research in the field has mostly focused on performing at or better than human level for a single task. For example, *Deep Blue*, which was developed years ago specifically to play Chess; *AlphaGo*, which was developed to play Go; or Watson, which plays Jeopardy. However, these are all focused on mastering a single task, so the question arises: how does this advance the scientific study of artificial intelligence? Recently, there have been several advancements in the study of general AI through games. This movement began with General Game Playing that focused on board games and later the General Video Game Playing AI (GVGAI) competition, which centers on arcade video Games [1].

In this contest, an applicant develops an *agent*, which is capable of playing all games in the library. The score of the agent is dependent on its performance across a certain subset of games, and across different levels in each of these games. Unlike other video game playing tasks, agents are expected to play video games without any prior knowledge about them, and the only way the agent can learn information about the game dynamics is to play. Agents and controllers will be used interchangeably for the remainder of the paper.

This competition has highlighted important discrepancies that lend credibility to the core artificial general intelligence hypothesis proposed by Goertzel in [2] (Goertzel, 2014), which essentially states that developing generally intelligent agents requires fundamentally different techniques from those used to develop narrowly intelligent ones. Notably, up to this point, mostly narrow AI strategies have been applied to create agents for this game library with measurable but limited success. Furthermore, there are now new techniques that have yet to be implemented in an agent for GVGAI.

In this paper we develop controllers for the GVGAI competition. We focus on developing controllers using three different techniques and two different tracks (i.e. the planning and learning tracks, as provided by GVGAI). The three techniques discussed in this paper are neuroevolution, a critic Monte Carlo tree search variant, and Q-Learning. Also, the controllers are focused to improve performance on certain games offered by GVGAI like Frogs, Portals and Sokoban, on which the previous explored approaches have not performed well, while at the same time maintaining standard in other games so that the "generality" is intact.

In Section 2, we discuss previous work research that has explored these three techniques. Section 3 defines each approach, and explains some implementation details for each. The results of our experiments with each agent are detailed in Section 4 with the performances, advantages, and drawbacks of the agents. Section 5 summarizes the findings of this paper and the potential future work.

All code for this project was written in `Python` and `Java`, and is available as a `.zip` file.

## 2 Background and Related Work

### 2.1 GVGAI

GVGAI was created to address the over-specialization inherent in AI created to play specific games [3]. Unlike most gaming environments, while GVGAI does have the option to provide environment information as an image, agents also have the option to receive information as an object called a state observation that the agent can query to learn about the environment. The information provided in a state observation is relatively limited. For example, a state observation contains information about the location and placement of different types of objects around the screen, but no information about how interacting with those objects will affect the agent. The state observation also has information about the winner, score, and player state.

Several sets of training games are available in the GVGAI to aid in developing agents, but agents submitted to the competition are validated on a set of 10 unseen games. The GVGAI framework is written in `Java` [4] and it interfaces with games written in the `Video Game Description Language (VGDL)`, developed by Tom Schaul[3]. Up until 2017, GVGAI only accepted `Java` controllers, but now accepts Agents in both `Java` and `Python` to accommodate the new single-player learning track.

The two single-player tracks are planning and learning. In the planning track, agents have the ability to revisit history, or events and collisions, that have occurred during the game. Planning agents can also access a forward model of the game that allows them to simulate the next possible state after choosing an action[3]. Given the state observation and these resources, agents are required to select an action within 40ms. During validation, the agent plays each of the 5 levels of 10 games in succession.

Agents in the learning track are also only allotted 40ms to select an action, but unlike the planning track, agents in the learning track cannot retrieve their history or any forward model by the game. To restrict the information available to learning agents, the agents interface with the game through a server rather than directly [4] The agents are given 5 minutes total to train on the first 3 levels of a new game, and then evaluated based on their performance playing each of the last two levels 10 times [4].

Both tracks present different sets of challenges that we endeavored to address below.

### 2.2 Neuroevolution

The idea of neuroevolution is to apply evolutionary algorithms to neural networks. This has previously been implemented in a real-time manner in [5] (Stanley et al., (2015)), in the NERO environment. To combine the ideas of networks and evolutionary algorithms, we must redefine how our networks are trained and organized.

Neuroevolution begins by initializing a fixed-size generation of neural networks. As explained in [5], it is important that these networks start small and sparsely connected, as we expect them to learn increasingly complicated behaviors as they develop. The networks in the first generation are then tested for fitness at performing a task, using a heuristic set by the developer, and ranked according to their performance. In general, the highest-performing networks are selected to pass into the next generation, and each of these networks is mutated using crossover and random mutation.

These mutations occur in the form of adding or deleting a neuron, or a connection between 2 neurons, and can also update the weights in the neural network. This means neuroevolution may take some time to learn complicated behaviors, but has some important advantages.

These benefits are mentioned in [5]. Firstly, the authors note that for a large state and action space, NEAT selects actions much more quickly than Reinforcement Learning, a technique it is often compared to. Secondly, NEAT potentially develops a variety of advantageous behaviors by virtue of its evolutionary algorithm. A third benefit over reinforcement learning is a smaller dependence on taking random actions for exploration. Each neuroevolution agent will perform consistently, even though this performance may be suboptimal for a small part of the agents. Neuroevolution also allows for a tuning between learning quickly and learning sophisticated behaviors, according to the task. Finally, each agent developed by this technique has a sense of "memory", as it is developed over several iterations of the task.

With respect to GVGAI, [6] (Samothrakis et al., 2015) has previously explored the possibility of applying neuroevolution to the game library. This paper explores using combinations of a linear and non-linear function approximator and an $\epsilon$-greedy and softmax policy to create neuroevolution agents for the game library.

The experiments performed in this paper suggest that using an $\epsilon$-greedy policy and linear function approximator most surely achieves a reasonably good result in performance.

### 2.3 Critic

Monte Carlo Tree Search Algorithm (MCTS) is a tree search algorithm that has impact on AI games since it has been introduced in 2006. This tree search technique is different from other tree search algorithms such as Minmax as it does not require game specific knowledge and does not require much heuristics, and works reasonably well even in its vanilla form i.e general MCTS without any certain modifications. The MCTS constructs an asymmetric tree where each node of the tree has certain statistics such that *N(s)* which represents number of times this node is reached, *N(s,a)* which represents how often an action a is played from that state and an average reward i.e. *Q(s,a)* which is obtained after playing an action a from state s. The MCTS algorithm starts with a root node and further can be divided into 4 phases:

- Selection: Start from root R and select successive child nodes down to a leaf node L. The child nodes is chosen such that it lets the game tree expand towards most promising moves, which is the essence of Monte Carlo tree search. The policy on which a node is selected is called *treePolicy*. The child is selected which has not been expanded i.e. not explored yet or which has given better results in the past. The tree selection policy maintains a balance between exploration and exploitation. The general criteria for to select an action or node is based on the given formula

  $$a^* = \underset{a \in A(s)}{\operatorname{argmax}} \left\{ Q(s,a) + C\sqrt{\frac{lnN(s)}{N(s,a)}} \right\}$$

  Here the value of C is the tweak point which maintains the balance between exploration and exploitation. If the value of C is lower then chosen action is biased towards Q(s,a) which would be exploitation and if C is higher we are preferring nodes or actions which have lower N(s) i.e. which has not been explored.

- Expansion: In this phase a new node is added to the selected node unless the node selected is ending the game.

- Simulation: A playout from the expanded node is simulated in this phase. In the playout the actions are chosen either randomly or strategically, the policy used for playout is called *defaultPolicy*. At the end of the playout also known as rollout the state is analyzed and a reward is given.

- Backpropagation: At this phase, the reward from the rollout are backpropagated uptil the root node.

The promising fact of MCTS is, it can be stopped anytime and can still give good results thus known as anytime algorithm. Though it provides decent performance in general game playing. There are few drawbacks to the method. Due to the time limitation the the nodes are not fully roll out in the simulation process which inhibits the process to know the winning states and biased actions on that basis. It does not utilise information that is provided in the games, i.e. actions which resulted in score gain in the past. It does not able to explore all the events, even those events which are necessary to win the game. For example in the game portals in GVGAI, one has to get to a exit point for to win the game but if the MCTS is not able to explore an action which will lead to the exit point it will never be able to win the game.

People have experimented with MCTS and modified the MCTS to able to cope with these issues. One method that was coined by [7] is Fast Evolutionary MCTS where they utilized Evolutionary strategy. They proposed the approach to have a weight vector associated with the features that are given in the game for ex- positions of sprites, position of the agent (avatar), position of non-player character etc. Every roll-out evaluates a single individual of the evolutionary algorithm, providing as fitness the reward calculated at the end of the roll-out. They used (1+1) Evolutionary strategy to come up with the weight vector and used standard *treePolicy* for selection and standard (random) *defaultPolicy* for simulation. The results from the method were promising. But it still lacked in the games which involved planning and where meaningful reward in the middle of the game were absent. We took forward this approach and meddled with it by introducing a global critic which would be able to biased towards action which worked better in the past but balancing it with exploring new events. This global critic would help to a certain limit in planning games.

## 2.4 Q-Learning

Problems that can be modeled as a set of states, *S*, and actions, *A*, lend themselves well to being solved using reinforcement learning. Q-learning is a form of value-function based reinforcement learning that assigns a reward to each specific state-action pair, *Q($s_t$,$a_t$)*, where $s_t \in S$ and $a_t \in A$, off of which a policy can be formulated. The process begins by arbitrarily initializing a Q-value table for each state achievable by the agent, executing an action, and then observing the resulting reward, *r*, and subsequent state, $s_{t+1}$, achieved. Given these new data, *Q(s,a)* is updated, and then $s_{t+1}$ becomes the new state from which actions must be chosen. This process is repeated until a terminal state is reached. Q is updated according to the equation

$$Q(s_t,a_t) \leftarrow (1-\alpha)\cdot Q(s_t,a_t)+\alpha\cdot\left(r_t+\gamma\cdot\max_a Q(s_{t+1},a)\right)$$

where $\alpha$ is the learning rate, $r_t$ is the observed reward, and $\gamma$ is the discount factor. Using this information, the agent selects the action at each state that will result in the largest expected reward that can be achieved from the current state, $s_t$.

Given the ability to visit any state in the state space unlimited times, for finite Markov decision processes, Q-learning has been shown to converge to an optimal policy

[9]. Q-learning is especially well-suited toward game playing because value can be assigned to each state without prior information about the optimal solution to the problem. Still, as state and action spaces grow along with problem complexity, visiting and maintaining a value for each state-action pair becomes untenable. Artificial neural networks (ANNs) have the ability to overcome this problem through by learning to approximate the action-value function Q. More recently, the Deep Q-Networks algorithm (DQN) has emerged as an especially powerful tool for video game playing because it takes in information about each state as raw pixels rather than hand-engineered features[10]. The same architecture was trained on seven different ATARI games using the same hyperparameters, and DQN was able to achieve better performance than other benchmark network on all games except one; in three games, DQN even outperformed a human expert player [10]. DQN represents an exciting step forward in general video game playing because it does not rely on an architecture with extensive tuning toward one task at the expense of generality.

Several improvements have been made to DQN to make it even more effective for video game playing. Double DQN, which separates the action selector network from the evaluation network, combats the overvaluation of certain state-action pairs that commonly occurs with Q-learning [11]. Another extension called Dueling DQN divides reward, $Q(s, a)$ into a separate value and advantage function, with the value function, $V(s)$ corresponding to the value of a state, and the advantage function, $A(a)$, corresponding to the benefit that would come from choosing one action compared to others [12]. This decoupling improves performance because there are many situations in which the specific action at a given state is not as critical as the state itself, thus the separated network can give a more accurate value-action representation of the system sooner by modifying the value of the entire state with each update, rather than the specific state-action pair alone [12]. These extensions, along with prioritized experience replay, multi-step targets, distributional Q-learning, and noisy DQN, were combined to create one integrated architecture, called Rainbow, for agents to play 57 different ATARI games, achieving a median performance of 153% compared to each extension individually[13]. In order to gain a better understanding of the contributions of each of the improvements to the integrated agent, the effect of removing each extension on game performance was also studied [13]. Although removing either dueling DQN or double DQN resulted in little change to the median performance over games, doing so improved performance over certain games, but performance in games like Montezuma's Revenge, a planning-type game, degraded [13].//

Despite the seemingly natural application of DQN to general video game playing, none of the ranked controllers in GVGAI seem to make use of the algorithm. In fact, ANNs are absent from all available controllers for the learning track of GVGAI. One likely reason is the feasibility of training an ANN in the constraints laid out by the competition. Compared to the 108K frame limit placed on one training episode for Rainbow, corresponding to about 30 minutes, each training episode for an agent in GVGAI is capped at 2000 frames. Thus, developing an agent that utilizes deep Q-learning to perform within the limits of deep Q-learning shows promise, but is relatively unexplored. We will also explore an attempt to improve the performance of traditional Q-learning with feature extraction.

## 3 Approaches and Architectures

### 3.1 Neuroevolution

To implement neuroevolution, we start with an already-implemented genetic algorithm controller, originally written by Spyridon Samothrakis, and modified to use neuroevolution.

As it is not possible to import the existing Python version of the NEAT algorithm into the GVGAI server, we decided to implement our own version of neuroevolution in Java, and took the opportunity to tweak the algorithm to see how this would affect performance. These adjustments can be seen in the `NeuralNet.java` and `Neuron.java` files in our repository.

Each neural network consists of a set of neurons, some of which are labeled as input neurons, some of which are labeled as output neurons, and some of which are neither input nor output. The network can be constructed using an input size and an output size, determining how many of these neurons are present in the network. When initialized, the network will also generate random connections between the input and output neurons.

When a network is mutated, there is a random chance that a neuron is added, with new directed connections from a source neuron and to a destination neuron. Some random connections may also be added, connecting 2 previously unconnected neurons. It is important that in this mutation step, the graph does not develop any cycles, which would make activation of the network impossible. We implement this cyclic check in the network file, making our network essentially a directed acyclic graph (DAG). An example of what one of the networks might look like is shown in Figure .

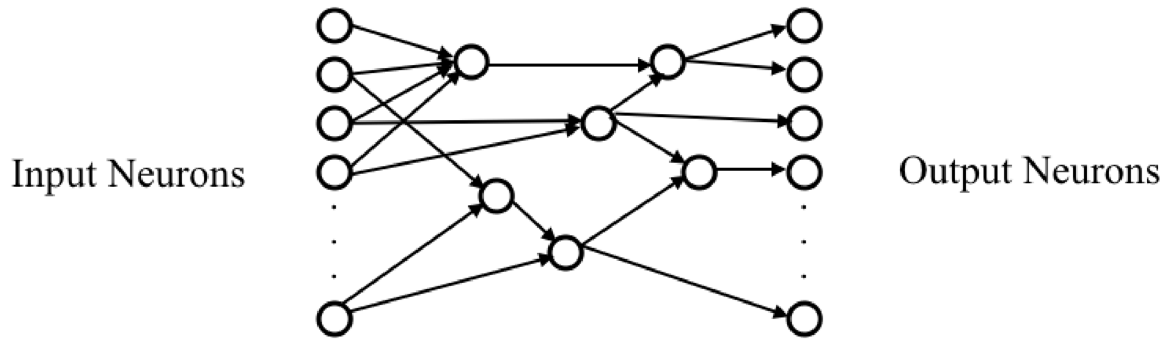

Figure 1: This neural net architecture abandons the notion of "layers", and can be thought of as a DAG. Each connection is directed according to the input/output relationship of the adjacent neurons. Mutation occurs by adding a neuron and/or connections.

Each neuron in a network consists of a value, a bias, inputs, and outputs. To compute its value, a neuron computes the sum of all of the values of its inputs recursively, weighted by their biases. The input neurons are assigned the values of the input passed to the neural network. Thus, the value, $v$, of a neuron with inputs $\{1, 2, ..., n\}$ would be

$$v = \sum_{i=1}^{n} v_i b_i,$$

where $v_i$ is the value of neuron $i$ and $b_i$ is the bias of neuron $i$.

To find its next action, our agent samples several neural networks from the current generation and uses it to find the next 30 actions to take. The network that performs the best is used to compute the agent's next move, and the others are mutated. This occurs every time the agent acts.

## 3.2 Critic

To implement a critic, we built it on top of a controller written by Diego Liebana, based on Fast Evolutionary MCTS, explained in 2.3. We maintain global information for all the actions which have performed better in the past. To quantify the quality of an action we used two metrics - first, score gain when that particular action was performed, and second, if it triggered exploration of new events. These two metrics help to bias the actions, thereby increasing the score, which serves as global information to improve performance on planning games, and concurrently, preferring actions that will explore when rewards are sparse. This controller is available in `FastEvoMCTS.zip` This combination should eventually help in the games that do not provide an intermediate reward but that require a particular event to happen in order to win. This global information is called the *critic*, which will guide the selection of node in the MCTS selection phase and will choose action among the actions that have been explored up to this point.

In the selection phase, the critic will do a "coin toss" to decide whether the agent should explore or exploit. If the probability of coin toss comes out to be less than 0.3, then it will explore; otherwise, it will exploit. Let $p$ be the probability of the coin toss. Then in the selection phase we will now choose an action according to the following formula:

$$a^* = \underset{a \in A(s)}{\text{argmax}} \left\{ \left\{ Q(s,a) + p * f * critic\_value[a] \right\} + C \left\{ \sqrt{\frac{lnN(s)}{N(s,a)}} + (1-p) * (1-f) \right\} \right\}$$

where $critic\_value[a]$ is the value of the action $a$ that quantifies how it has performed in the past, and $f$ is the factor that quantifies the critic's effect, i.e. if $p$ is greater than 0.3, then we want to exploit and increase $f$, which reduces the factor of exploring, and vice versa in the case when $p$ is less than 0.3. While choosing the action finally to return among the actions that have been explored, we favor those actions according to the critic by multiplying the value $critic\_value[a]$ for a particular action $a$ directly by the value of the node corresponding to $a$.

## 3.3 Q-Learning

Two controllers were implemented to examine how well Q-learning seems to work as a tool for learning in general video game playing. The first implementation uses traditional Q-learning algorithm in the form of tabulation. The controller has been modified from one written in `Java` by Kamolwan Kunanusont to provide the agent with better information about each state by including information about the agent's surroundings and the sprites present in each game rather than only information about the agent alone. The action selection policy was also updated to encourage more exploration at the onset of training with an $\epsilon$ that decreases exponentially. Last, given the emphasis on win percentage over game score, the reward function was updated to decrease the reward by half when the player does not win.

The feature extraction for the states was normalized so that information about the number of each type of sprite is preserved between levels in the Q-table, which allows for a more accurate comparison between states than if the representation of different types of sprites were to change for different levels. The first three episodes of training during the learning track always levels 0, 1, and 2 of the game; by exploring the level during this phase and tallying up the different types of objects encountered at each level, the agent can initialize a feature vector that can fit information for all three stages.

The feature extraction of the DQN controller is similar to the standard Q-learning controller, but it provides additional spatial information. The state observation returned by the game has an observationGrid, which is a 3-dimensional array that holds a list of observations containing the ID type for all sprites located in each 2-dimensional region indexed by the first two dimensions of the observationGrid. The maximum dimensions of the three levels played by the agent during the training were combined with the total number of sprites to create a 3-dimensional feature space, with the first two dimensions being the height and width of the level broken into even blocks according to the dimensions of the smallest sprite, and the third dimension being the total number of different sprites observed during exploration.

The DQN architecture takes in one 84 x 84 x 4 pixel image, which is then processed through 2 convolutional layers and 1 fully-connected layer before reaching an output layer with one output node for each valid game action[10]. The architecture implemented in the DQN agent is modified to better fit the input dimensions, with smaller filters that are proportional to the grid being fed in. Rather than using a traditional DQN architecture with data feeding directly from convolution to a fully connected layer, a dueling DQN architecture was implemented for the potential benefit it could bring to planning games. After the second convolution, the outputs were Likewise, that architecture was used to create a target Q-net and a prediction Q-net. Each experience, consisting of the previous state, action, reward, and current state were meant to be saved in an experience buffer so that the networks would be able to train.

## 4 Experiments and Results

The controllers developed are for two different tracks provided by GVGAI i.e planning track and learning track. The learning track is newly developed and became part of the competition this year 2017. Learning track is different from the planning tracks as no forward model is given to the agent, thus, no simulation of games is possible. The agent though, have access to the current game state (objects in the current game state), as in the planning tracks. We tried to explore methods for both the tracks i.e with and without simulation. Namely, neuroevolution and critic are for planning track and Q-Learning controller is for the learning track.

The performance of the controllers was tested using Training Set 1 for the single-player competitions, which consists of Aliens, Boulderdash, Butterflies, Chase, Frogs, Missile Commands, Portals, Sokoban, Survive Zombies and Zelda. The games are tested over different levels and each level is played multiple times, with the average score and average percentage of victory recorded. This set was chosen because the performance of the winning controllers over this set is readily available. The 10 games are distinct in terms of layout, mechanics, and win criteria.

### 4.1 Neuroevolution

The results from the agent using neuroevolution are promising. Notably, we see that it wins on 2 out of the 5 levels it plays on in Sokoban, a result that betters the one from [6]. This was one of the main goals of this agent, as Sokoban proved to be a difficult game to play in past research. Furthermore, this agent wins every level in Aliens and Butterflies. This is most likely due to the fact that the exploration phase during the Aliens phase is quickly solved by the agent, as it discovers that moving all the way to the left and then shooting will optimize performance (Figure ).

A generally interesting point to make about this agent is that it seems to perform relatively well at binary score games (games where the player either wins or loses and gets no score in between). This is the case for Frogs and Portals, where the agent managed to win of the levels.

Shortcomings of this agent occurred in the games Boulderdash and Chase. This could be because of the variety of objects in both these games. In Chase, the agent must distinguish between angry and scared goats, and in Boulderdash, it must distinguish between valuable resources, such as diamonds, enemies, and rocks. It is possible that this is what

| Controller | Games | Aliens | Boulderdash | Butterflies | Chase | Frogs | Missile Command | Portals | Sokoban | Survive Zombies | Zelda |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Neuro | Victory (%) | 100 | 0 | 100 | 0 | 20 | 80 | 20 | 40 | 60 | 20 |
| | Avg. Score | 64.6 | 6.0 | 38.8 | 0.8 | 0.2 | 5.2 | 0.2 | 1.4 | 4.2 | 7.4 |
| Critic | Victory (%) | 100 | 0 | 0 | 0 | 20 | 20 | 40 | 20 | 20 | 20 |
| | Avg. Score | 43.2 | 3.6 | 20.4 | 1.4 | 0.2 | 43.2 | 3.6 | 20.4 | 1.4 | 0.2 |
| Champion | Victory (%) | 100 | 80 | 100 | 40 | 100 | 100 | 80 | 100 | 20 | |
| | Avg. Score | 70.8 | 19 | 30.4 | 5.6 | 1 | 25 | 12 | 3 | -9 | 4.4 |

Table 1: Performance of planning-track controllers compared to top controller for GVGAI Training Set 1

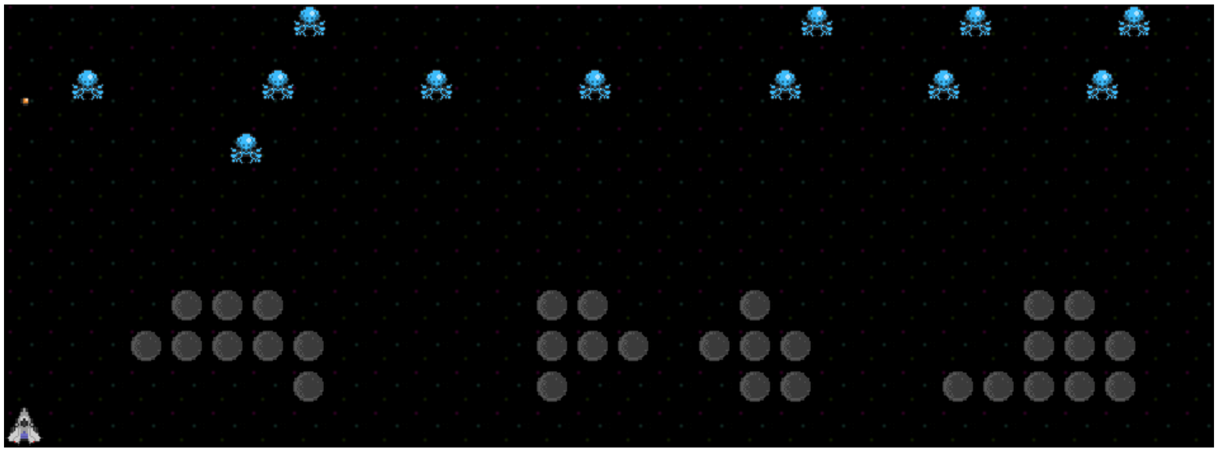

Figure 2: The neuroevolution agent quickly discovers that moving to the left and firing consistently is an optimal strategy to win the level in Aliens.

confuses the agent.

### 4.2 Critic

As explained in section 3.2 we are trying to explore and exploit to overcome the issues of MCTS and perform better in games such as Sokoban, Portals for which there are no good controllers. The method was successfull as it wins a significant percentage in games like Frogs, Portals, Sokoban as compared to previous controllers like vanilla MCTS and Fast Evolutionary MCTS which did not perform good enough in this games as mentioned in [8]. They got benefit from the exploration and exploitation, as now the algorithm was able to explore new events and gained score which was further used to prefer these actions as in exploitation. Though the performance got better in these games, some games at the same time suffered, such as butterflies, chase which would have performed better if the exploration factor was not there.

### 4.3 Q-Learning

Unfortunately, even though TensorFlow was stated to be compatible with the learning track of GVGAI, it seemed as though the socket built to communicate between the `Java` game and the Python agent broke down whenever the TensorFlow session was initialized. The code to create the DQN agent can be seen in `DQN.zip` Thus, the only results available for Q-learning are from the standard tabulated function. Q-learning seems to work well for games like aliens, which has a very limited state space, as the agent can only travel along the x-axis of the screen, and the action space only consists of 3 actions. For planning games, the simple Q agent is ineffective. This agent can be seen in `simpleQ1.zip`.

## 5 Conclusion and Future Work

Neuroevolution proves to be a promising approach towards developing agents for GVGAI. By making adjustments to the existing neuroevolution algorithm, we see that performance improves measurably in Sokoban. We also note that the agent performs well in binary score games such as Frogs and Portals. An interesting extension to the neuroevolution agent could be to tune the hyperparameters of the algorithm to attempt to optimize agent performance.

The critic was able to perform better in games which we planned to make our controller better for, such as frogs, portals and Sokoban and played decent enough in other games, even acheived 100% victory rate on Aliens. It was able to resolve some of the issues faced by Vanilla MCTS and

| Controller | Games | Aliens | Boulderdash | Butterflies | Chase | Frogs | Missile Command | Portals | Sokoban | Survive Zombies | Zelda |
|---|---|---|---|---|---|---|---|---|---|---|---|
| simpleQ | Victory (%) | 45.0 | 0 | 10 | 0 | 0 | 50 | 0 | 0 | 0 | 0 |
|  | Avg. Score | 35.0 | 0 | 8.0 | 0 | 0 | -3.0 | 0 | 0 | 0 | 0 |
| Champion | Victory (%) | 45 | 0 | 90 | 0 | 0 | 50 | 0 | 0 | 0 | 0 |
|  | Avg. Score | 37.25 | 1.3 | 18.6 | 0.65 | 0 | -0.1 | 0 | 0.7 | 0.05 | 0.05 |

Table 2: Performance of simple Q-learning controller compared to top controller on GVGAI Training Set 1

even Fast Evolutionary MCTS. However, there are still lot of spaces for improvement in the current implementation. In this implementation, there are many variables whose values are decided empirically through testing over games after playing it certain limited number of times, the method might improve more if the values could be learnt over time so that it will be able to adapt when to explore more or when to exploit more. The performance in the targeted games is slightly better than previous but still less in absolute manner which could be improved by adding more features correspondingly but with the overhead of large search space as discussed in [8].

The simple Q-learning agent acted as expected, with performance degrading for games that require more complex behavior. Still, compared to the existing learning agents, it performs about on par. Clearly, much more work can be done in terms of the learning track of GVGAI. One option to test the effectiveness of the DQN agent that was created is the Arcade Learning Environment, but that environment, along with other video game options for testing AI, do not provide the same object-oriented feedback as GVGAI. Our initial assumption that accounted for the lack of controllers utilizing ANNs due to not being able to effectively train in the given time constraints has been replaced with the possibility that competitors did not implement ANNs using TensorFlow because they were not compatible with the system.

## 6 Acknowledgements

Our work would not have been possible without building off of work and code written by Tom Schaul in February 2014 (https://github.com/schaul/py-vgdl), Diego Perez-Liebana, and Kamolwan Kunanusont. Their code for sample controllers and feature extraction were used in this project.

## References

[1] Togelius, J., and Yannakakis, G. N. (2016). General general game AI. In 2016 IEEE Conference on Computational Intelligence and Games (CIG) (pp. 1-8).

[2] Goertzel, B. (2014). Artificial General Intelligence: Concept, State of the Art, and Future Prospects. Journal of Artificial General Intelligence, 5, 1-48.

[3] Perez-Liebana, D., Samothrakis, S., Togelius, J., Lucas, S. M., and Schaul, T. (2016). General video game AI: Competition, challenges, and opportunities. Presented at the 30th AAAI Conference on Artificial Intelligence, AAAI 2016, AAAI press.

[4] Liu, J. (2017). The Single-Player GVGAI Learning Framework Technical Manual.

[5] Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2005). Real-time neuroevolution in the NERO video game. IEEE Transactions on Evolutionary Computation, 9(6), 653-668.

[6] Samothrakis, S., Perez-Liebana, D., Lucas, S. M., and Fasli, M. (2015). Neuroevolution for General Video Game Playing. In 2015 IEEE Conference on Computational Intelligence and Games (CIG) (pp. 200-207).

[7] Lucas, S. M., Samothrakis, S., and PÃľrez, D. (2014). Fast Evolutionary Adaptation for Monte Carlo Tree Search. In Applications of Evolutionary Computation (pp. 349-360). Springer, Berlin, Heidelberg.

[8] Perez, D., Samothrakis, S., and Lucas, S. (2014). Knowledge-based fast evolutionary MCTS for general video game playing. In 2014 IEEE Conference on Computational Intelligence and Games (pp. 1-8).

[9] Watkins, C. J. C. H., and Dayan, P. (1992). "Q-learning." In Machine Learning (pp. 279-292).

[10] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602 [cs].

[11] Van Hasselt, H., Guez, A., and Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. arXiv:1509.06461 [cs].

[12] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. (2015). Dueling Network Architectures for Deep Reinforcement Learning. arXiv:1511.06581 [cs].

[13] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., âĂę Silver, D. (2017). Rainbow: Combining Improvements in Deep Reinforcement Learning. arXiv:1710.02298 [cs].