Navigating Reward Structures: A Comparison of Deep Q-Learning and Genetic Algorithms in Diverse Environments

Daniel Gordin and **Ryan Heminway**
Northeastern University

Abstract

In this work, we evaluate the performance of two popular optimization algorithms, Deep Q-Learning and Genetic Algorithms, on training an agent's policy network in both dense and sparse reward structures. The experiments were conducted on two classic reinforcement learning environments, namely the Lunar Lander and Mountain Car scenarios. We found that Deep Q-Learning is quite fast and effective at optimizing policies in dense reward environments, but struggle with sparse ones. The Genetic Algorithm however was able to successfully train the networks in the sparse environments albeit at the cost of a much larger number of training episodes.

1 Introduction & Related Work

Reinforcement learning (RL) is a subfield of machine learning that focuses on developing intelligent agents that learn to make optimal decisions in complex and uncertain environments. These agents interacts with an environment by taking actions, receiving feedback in the form of rewards, and observing the resulting state of the environment. The goal of the agent is to determine a policy that maximizes the expected cumulative reward over time.

In the past few years, RL has gained considerable attention in a number of domains due to its flexible ability to learn from trial and error simulations(Mnih et al., 2013). The field of robotics in particular has benefited from these techniques in a multitude of applications including robotic grasping, autonomous driving, robot locomotion, robot navigation, and control policy optimization. (Kober et al., 2013)

The performance of RL algorithms heavily depends on the characteristics of the task and the environment. In some environments, the rewards are dense and well-defined, providing feedback on every action taken by the agent. In other environments, the rewards may be sparse and noisy, making it difficult for the agent to learn a good policy.

In this paper, we focus on comparing the performance of two prominent network optimization techniques, Deep Q-learning (DQL) and Genetic Algorithms (GA), in two different types of environments - dense reward and sparse reward environments. We investigate which technique is better suited for each reward structure through simulations of the Lunar Lander and Mountain Car scenarios from OpenAI gym. .

Several other studies have also investigated evolutionary algorithms (EA) or as a solution for network optimization in certain tasks (Andersen et al., 2002), (Miikkulainen et al., 2017). (Yao, 1999) used EAs to evolve neural network connection weights, architectures, learning rules, and input features. (Such et al., 2017) showed that GAs are a competitive alternative for training deep neural networks for reinforcement learning. (Mnih et al., 2016) explored the usage of asynchronous gradient descent for optimization of deep neural network controllers.

2 Background

The primary objective of a reinforcement learning agent is to learn a policy that maximizes the expected cumulative reward over time. A policy is a function that maps the current state of an agent to the action that the agent should take in that state. For this work, we represent our policy function as a neural network and train this "policy network" using two different algorithms: Deep Q-Learning and Genetic Algorithm.

2.1 Deep Q-Learning

Q-learning is a model-free algorithm in reinforcement learning that is used to learn the optimal policy for an agent in a given environment. Since this approach is model-free, learning is done through trial and error interactions with the environment

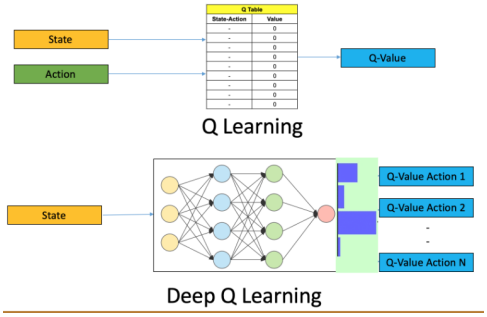


Figure 1: Q-Learning vs Deep Q-Learning

where feedback is received in the form of rewards or penalties.

The Q-learning algorithm is based on the Bellman equation, which says that the value of a state is equal to the immediate reward plus the discounted value of the expected future rewards:

$$Q(s,a) = r(s,a) + \max_{a'} Q(s',a')$$

The algorithm works by iteratively updating a lookup table (called a Q-table) that stores the expected reward for each action in each state (Figure 1). During training, the agent explores the environment and uses the Q-table to select the action with the highest expected reward at each state (epsilon-greedy). After each exploration, the current states Q-value is updated in the Q-table using Temporal Difference (TD):

$$TD(a,s) = r(s,a) + \max_{a'} Q(s',a') - Q(s,a)$$

$$Q(s,a) = Q(s,a) + TD(a,s)$$

In many real-word problems, the state-action space is too large or continuous and thus cannot be feasibility enumerated and stored in a Q-table. Deep Q-learning addresses this scalability problem by replacing the Q-table with a Deep Q Neural Network (DQN) that can approximate the action values for any state-action pair. Another advantage of deep Q-learning is its ability to generalize to unseen states and actions. This makes the algorithm more robust to changes in the environment and allows it to transfer its knowledge to new situations.

The input to the DQN is the state of the environment, and the output is a vector of action-values, where each element represents the expected Q-value of taking a particular action in the current state. The loss function the network tries to minimize is squared error between the predicted and actual Q-values (Figure 2).

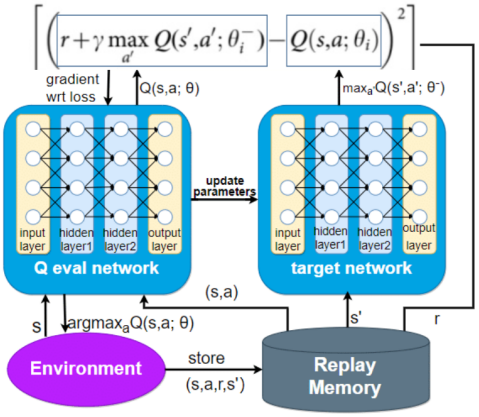


Figure 2: Deep Q-Learning Framework

However, this formulation alone can result in a large divergence between these Q-values and overall unstable learning and sub-optimal policies. One of the primary reasons for this instability is the fact that the Q-learning update rule involves bootstrapping, where the Q-value estimate is updated based on a target that is also estimated using the same neural network (since the target Q-value is a function of the predicted Q-value for the resultant state, s'). To mitigate this instability in Deep Q-Learning, there are a few common techniques employed in the field:

1. Experience Replay: Instead of updating the Q-values with the most recent experience, Deep Q-Learning stores the experiences in a replay buffer and samples a batch of experiences randomly from the buffer for each update. This can help reduce the correlation between consecutive updates and improve the stability of the learning process.

2. Target Network: A separate target network, rather than the main network, is used to calculate the target Q-values during the update process. The target network is a copy of the main network, but the weights are frozen for a certain number of iterations before being replaced with the weights of the main network. This helps to stabilize the target Q-values and prevent the Q-values from oscillating/diverging.

3. Clipping the TD Error: The TD error, which is the difference between the target Q-value and the predicted Q-value, can be clipped to a certain range. This can help prevent large

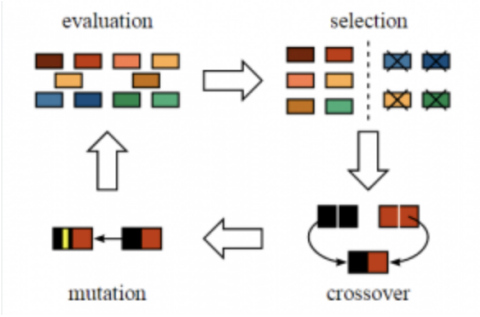


Figure 3: Genetic Algorithm Framework

updates to the Q-values, which can cause instability in the learning process.

4. Exploration Strategies: Exploring the environment in a more efficient manner can help to improve the stability of the learning process. Techniques such as epsilon-greedy exploration and Boltzmann exploration can help to encourage exploration while avoiding getting stuck in local optima.

2.2 Genetic Algorithm

The Genetic Algorithm (GA) is a population-based optimization method inspired by the processes that drive biological evolution according to Darwin's theories of natural selection. GA is a type of Evolutionary Algorithm, a larger set of methods and models inspired by natural evolution theory. The GA method is fairly simple and follows a 4-step (Figure 3) process of:

1. Evaluating individuals in the population on their fitness in the desired task.

2. Selecting individuals from the population to use for reproduction.

3. Generating offspring according to genetic operators such as crossover and mutation.

4. Replacing individuals in the population with offspring to produce the next generation's population.

The exact individual representation and fitness evaluation methodology are customized to the problem domain. A critical characteristic of this algorithm is the complete lack of gradients required to execute it. This simplicity makes it applicable to a very wide range of problems, with only a small amount of domain-specific setup. Within this algorithmic framework, though, there are many variations that are possible.

The two fundamental versions of a GA are Steady State and Generational. Steady State is an incremental version where reproduction happens a single time, replacing only one or two individuals each generation. Generational GA is a version where each generation produces an entirely new population, with possibly little resemblance of the previous. An important feature of Generational GA is called "elitism" where an implementation can elect to maintain a certain number of top-performing individuals in every generation. This avoids the chance of progressing to a new generation and losing the best solution found so far.

Furthermore, there are multiple ways to implement evolutionary operators such as selection, crossover, and mutation. For example, the mechanism used for selection is important and varies based on the implementation. Popular methods include "truncated selection", where parents are chosen randomly from the set of top performing individuals, "tournament selection" where a random group of individuals is chosen and then the parent is taken as the fittest one of the group, or "roulette wheel selection" where individuals are weighted according to their fitness and then randomly selected.

There are also multiple ways to implement mutation and crossover operations. "Gaussian mutation", for example, refers to adding Gaussian noise to all values in an individual while "replacement mutation" refers to replacing a single or multiple values in an individual with new randomly sampled values. "Single point crossover" refers to exchanging portions of two individual solutions' vectors divided at one location, while "multi point crossover" refers to the same process with multiple dividing positions. All of these variations can be viewed as ways to alter the exploration and exploitation tendencies of the algorithm, a classic problem in reinforcement learning.

Overall, the power of GA comes from its simplicity and quality of being a gradient-free optimization method. This is especially helpful when a gradient evaluation is costly or infeasible. Additionally, the Generational GA that we will be implementing has the benefit of being highly parallelizable which helps drastically reduces the wall-time required to run the algorithm.

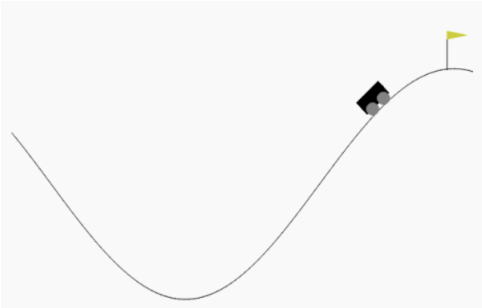


Figure 4: Mountain Car Environment

3 Developer Toolkit/Simulator

All the simulators used in this project originate from OpenAI Gym[1], an open-source toolkit for developing and comparing reinforcement learning algorithms. Gym offers a set of environments, including classic control problems and Atari games, as well as a standard interface for communicating with these environments.

3.1 Mountain Car

The Mountain Car environment (Figure 4) is a classic control problem where the goal is to teach an agent to drive a car up a steep mountain slope. The agent must learn to drive the car back and forth to gain enough momentum to climb the hills and reach the goal position located at the top of the right hill. The Mountain Car problem is challenging because the agent must learn to balance the trade-off between accelerating to gain momentum and decelerating to avoid overshooting the goal position. Mountain Car has a state dimension of 2, an action dimension of 3, a very repetitive reward structure in -1 per time-step, and we considered solved if the episode score is greater than -200.

3.2 Lunar Lander

In this environment an agent is tasked with controlling a lunar lander to land on a designated landing pad. The goal of the agent is to land the lander safely on the landing pad while using the least amount of fuel possible. Lunar Lander has a state dimension of 8, an action dimension of 4, a very rich reward structure, and we consider solved if the episode score is greater than 200. Due to the larger state space and more stringent win conditions, lunar lander is considered a more difficult problem to solve.

[1] https://www.gymlibrary.dev/

4 Methods

We wrote all of our implementation code in Python 3.6 using the following main libraries: PyTorch, PyGAD, Numpy, Pandas, and Gymnasium. For every episode of a Gymnasium environment, a random seed was used to ensure our methods were producing general learners rather than overfit solutions to a specific initialization.

In order to be able to compare results between models, we made sure the architecture of the policy network was kept the same. The architecture was a multilayered perceptron with 1 fully connected hidden layer of dimension 64 and a linear layer output since we are predicting Q-values which can take on any real number.

The sparse reward versions of the Lunar Lander and Mountain Car environments were achieved by modifying the original reward functions to assign a reward of of 0 for every step, except at the termination of the episode, where the total accumulated reward was provided instead. This resulted in a total of four environments for each model to be evaluated. For the GA model, it is important to highlight that the sparse versions of the environments operate identically, as the fitness function of the GA only reports the reward after an episode is completed. Effectively, the GA is forced to operate on a sparse reward function at all times.

4.1 Deep Q-Learning

The Deep Q-learning implementation was made up of two major classes: a Memory Replay class which stored the agent's experiences in a deque and the Agent class that stored the policy and target networks.

The Memory Replay had a maximum size of 100,000, a minimum size of 1000 (the amount of experiences necessary before sampling and updating can occur), and a sampling batch size of 64.

The network training hyperparameters included a learning rate of 5e-4, an episode length limit of 1500, a gamma value of 0.99, and an update rate of 50 (rate of overwriting the target network).

Finally, agent action selection utilized an Epsilon-greedy approach with an initial epsilon value of 1, a per-episode epsilon decay rate of 0.9999, and a minimum epsilon value of 0.01.

4.2 Genetic Algorithm

We implemented a Generational GA with tournament selection, single point crossover, replacement

mutation, and elitism. The implementation was done using the PyGAD library. The individual representation is a one dimensional vector of floating point values. Each value corresponds to a weight or bias in the network, and has a value in the range [-1, 1]. For initialization of the population, each individual is assigned a vector sampled uniformly from the same range. Crossover was executed with a probability of 0.6, while mutation was applied to 10% of the values in an individual. The size of the tournament used in selection was 10 individuals and the number of elites maintained at each generation was also 10. Finally, the fitness of an individual was represented by the average total reward collected across 10 episodes in an environment. Notably, these episodes used potentially different random seeds to aid in the agent's ability to generalize.

For all experiments, the GA was trained for a total of 100 generations. With a population of 100 individuals, each evaluated on 10 episodes for their fitness, this equates to running a total of 100,000 episodes in the environment, and a total of 10,000 different solutions evaluated.

5 Results

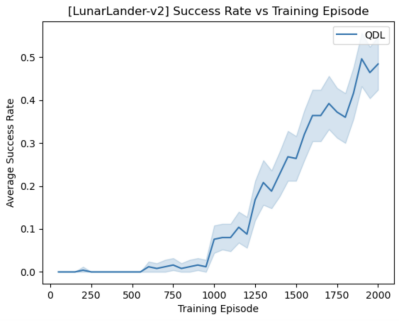
To execute our experiments, we first conducted simulations of each model in all four environments. To track the results throughout training, the models were periodically subjected to evaluation. In this evaluation, the model trained up to that point was tested by running it in the environment for a total of 25 episodes. For each episode, we collected the metrics of 'episode', 'total reward', 'number of steps', and 'success' where success is a Boolean determined by the final score of the episode. This evaluation period was executed after every generation for the GA models, and every 50 episodes for the DQN models. To ensure statistical robustness, all experiments were repeated 10 times for each unique set of parameters. Considering the 25 episodes of each evaluation, and the 10 repetitions of each run, all reported results represent an average of 250 data points.

Once experiments and data collection were complete, we created plots of the resulting performance metrics. In Figure 5, the average success of an agent is plotted against training episode for the DQN model trained in the Lunar Lander environments. In Figure 6, the average success of an agent is plotted against training episode for the DQN model trained in the Mountain Car environments. In Figure 7, the average success of an agent is plotted against training episode for the GA model trained in both the Lunar Lander and Mountain Car environments. From this set of plots, it can be observed that the DQN models were trained for 2000 episodes, while the GA models were trained for 100 generations totalling to 100,000 episodes evaluated. Finally, in Figure 8, the average success of an agent is plotted against training episode and number of weight updates respectively. Figure 8 details results from both models trained in the Lunar Lander environment. The visible discrepancy in training episodes and number of weight updates between the two models is further discussed in the following sections. Finally, we created a table consolidating the results of all our experiments which can be seen in Figure 9. For additional plots of the model performances, refer to the Appendix.

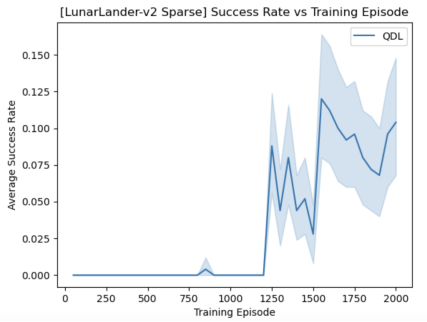
Figure 10 is a nice demonstration of how the Lunar Lander agent learns after a certain amount of training episodes. In the first 100 episodes, the agent is lost and keeps crashing. After 200 episodes it realizes that staying in the air indefinitely is better than crashing, which poses an infinite horizon problem if the training is never truncated. At around 400 episodes the agent finally learns the benefits of descending and by 1500 episodes they are consistently landing on the target.

6 Discussion

The results of our model evaluations and simulations offer several valuable insights. Firstly, Deep Q-Learning is an effective reinforcement learning algorithm for training policy networks in environments with dense reward structures, as demonstrated by its high success rates in the regular Lunar Lander and Mountain Car scenarios. As shown in Figure 9, the DQN models achieve average success rates of 80% and 50% in the dense Mountain Car and Lunar Lander environments, respectively. Interestingly, the DQN outperforms the GA model on the Lunar Lander environment, which achieves an average success rate of only 35%, but only matches the performance of the GA model on the Mountain Car environment. We hypothesize that this inability to outperform the simpler GA model is due to the DQN's high dependency on a rich reward structure from the environment. Even in the default Mountain Car environment, the reward is identical for all time steps. The reward is non-zero, so it is not considered a sparse environment, but we still be-

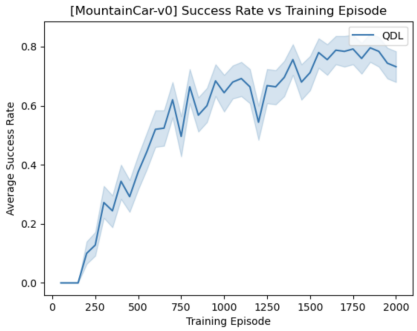


(a) Dense Reward

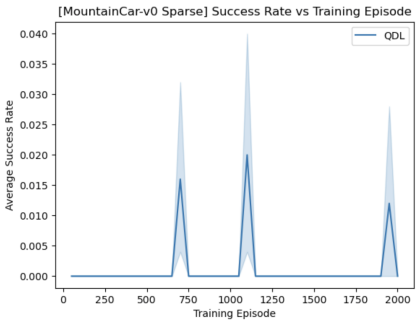


(b) Sparse Reward

Figure 5: Average success rate plotted against training episode for the QDL model trained in the Lunar Lander environments.

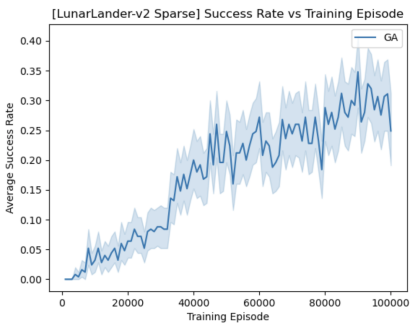


(a) Dense Reward

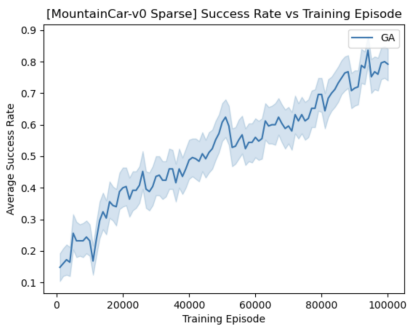


(b) Sparse Reward

Figure 6: Average success rate plotted against training episode for the QDL model trained in the Mountain Car environments.

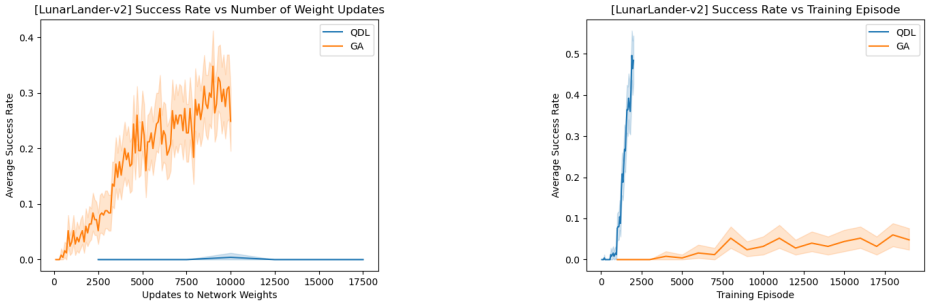


(a) Lunar Lander



(b) Mountain Car

Figure 7: Average success rate plotted against training episode for the GA model trained in both environments.



(a) (b)

Figure 8: Success rate plotted against number of network weight updates (a) and number of training episodes (b). Depicts results of both models trained on the Lunar Lander environment. X-axis is constrained to a maximum of 20,000 even though results continue after this value.

Environment	Model	Mean Success Rate	Mean Reward	Mean Steps
Lunar Lander	DQL	50%	180	610
Mountain Car	DQL	80%	-138	138
Lunar Lander (Sparse)	DQL	12%	-325	450
Mountain Car (Sparse)	DQL	0%	-198	198
Lunar Lander (Sparse)	GA	35%	80	320
Mountain Car (Sparse)	GA	83%	-137	137

Figure 9: Table of results across all environments and models. All results report the best average value achieved during training, except for "Mean Steps" which reports the value at the very end of training

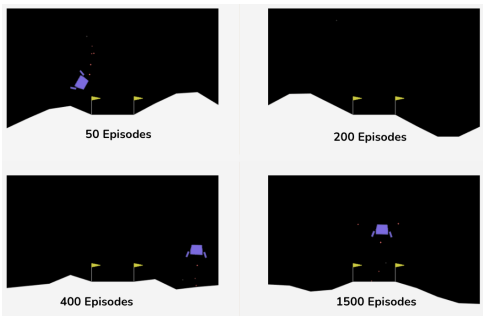


Figure 10: Lunar Lander Performance at Various Stages of Training

lieve this poses problems for the DQN which relies heavily on rewards at each time step to guide its learning trajectory. In contrast, the default Lunar Lander environment has a diverse set of reward signals that continuously guide the agent towards the goal at each time step. This trend of reward structure dependency is further amplified by the results obtained in the sparse environments.

In the sparse environments, the GA models achieve identical results as in the dense environments, while the DQN models struggle to learn useful behavior. From Figure 9, we can see that the DQN model achieves average success rates of only 12% and 0% on the sparse versions of the Lunar Lander and Mountain Car environments, respectively. These results indicate a strong reliance on consistent reward signals for the DQN to learn a general policy. This is likely due to the zero rewards at almost every time step causing gradient updates which have little to no effect on the model weights during training. In this same vein, we conclude that the GA models are more resilient to variations in environment reward structures since they are never dependent on rewards at a per time step basis. This experiment really highlights the strength of GA being a gradient-free optimizer since gradient calculations in sparse environments are generally untenable.

There is still a trade off in performance with these models, even considering the overall acceptable performance of the GA results. That trade off is one of training episodes vs number of unique solutions (weight updates). As detailed in the Methods section, the GA model necessitates a total of 100,000 episodes in the environment to evaluate a total of 10,000 unique solutions, a ratio that can be adjusted based on the formulation of the fitness function in a given implementation. On the other hand, the DQN model exhibits variability in the number of updates across experiments. The training period comprises 2,000 episodes in total, but a weight update occurs every 4 steps within an episode. This ratio, too, can be adjusted within an implementation. Based on our findings, it is reasonable to estimate that the DQN model executes

approximately 50 weight updates per episode, totaling to 100,000 unique solutions tested during the training period. There is a noticeable disparity here, with the DQN model requiring 50 times fewer episodes for training but executing 10 times as many weight updates compared to the GA model. As a demonstration of this trade off, Figure 8 depicts the average success of both models run in the dense Lunar Lander environment plotted against (a) number of weight updates and (b) number of training episodes. Both plots show the full training period for only one of the models, as the X-axis was clamped to a maximum value of 20,000 to demonstrate the severe imbalance in training requirements for each model. Our conclusion from this result is that the GA model is highly episode dependent, which lends itself well to problems where gradient updates may be computationally expensive but episode evaluations are cheap. On the other side, DQN may be computationally efficient for problems where gradient updates are cheap but episode evaluations are expensive. This trade off is not one we intended to explore, but poses an interesting direction for future work. Despite this trade off, both models demanded a comparable amount of wall time for their training. Although resource dominance was not the primary focus of our work, it is noteworthy that one algorithm did not exhibit excessive resource requirements. Specifically, the GA model, as implemented in our study, took longer to run but offers the advantage of being parallelizable for faster run times, a benefit that the DQN model lacks.

7 Future Work

Our study focused on comparing the performance of Deep Q Learning and Genetic Algorithms in dense and sparse reward environments. However, there are other ways to modify the environment that could be explored in future research. For example, in some scenarios, the number of actions that an agent can take may not be constant for every state. Comparing the performance of different RL algorithms in such environments could be a fruitful research direction.

Additionally, while our study specifically analysed Deep Q Learning and Genetic Algorithms, future work could explore extensions to these algorithms, such as Double Q-Learning or Dueling Q-Learning. Moreover, we may also try entirely different types of RL algorithms such a Actor-Critic algorithms, Proximal Policy Optimization, or Trust Region Policy Optimization in dense and sparse reward environments to ultimately identify the most suitable algorithm for a given environments characteristics.

8 Code

Please find our code at this[2] repo.

References

Timothy Andersen, Kenneth Stanley, and Risto Miikkulainen. 2002. Neuro-evolution through augmenting topologies applied to evolving neural networks to play othello.

Jens Kober, J. Bagnell, and Jan Peters. 2013. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32:1238–1274.

Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. 2017. Evolving deep neural networks.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA. PMLR.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. 2017. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning.

Xin Yao. 1999. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.

[2] https://github.com/ryanheminway/LearningInTheGym

Appendix

A. Lunar Lander Performance at Various Stages of Training

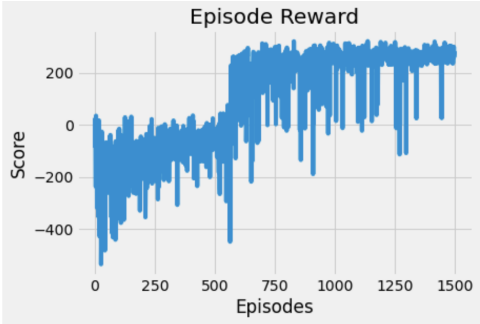


Figure 11: Scores

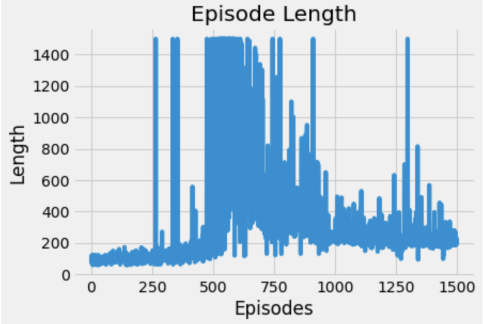


Figure 12: Episode Lengths

B. Mountain Car Performance at Various Stages of Training

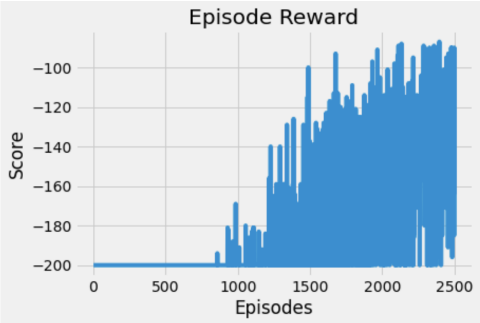


Figure 13: Scores

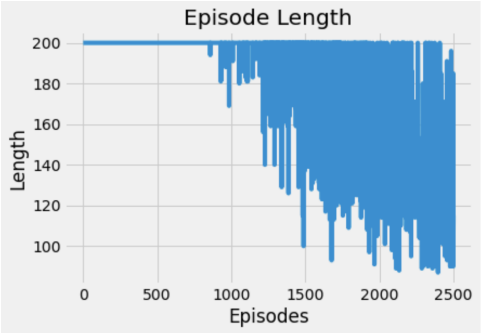


Figure 14: Episode Lengths