**A blue shield with a cross and a book

Description automatically generated**

**Project 6 For Diploma in AI and its application in Business:**

**Reinforcement Learning with Genetic Algorithm**

*Evolutionary Computing to Solve Reinforcement Learning Environments*

Prepared by: Chen Yijie

Supervisor: Zhou Changxin

Table of Contents

[Project Overview: 3](#_Toc164344905)

[Covering Concepts: 3](#_Toc164344906)

[Potential Use Case: 5](#_Toc164344907)

[Environment Used: 7](#_Toc164344908)

[Reinforcement Learning Environment Used 8](#_Toc164344909)

[Reinforcement Learning Environment Understanding 9](#_Toc164344910)

[Evolutionary Computing Concepts: 13](#_Toc164344911)

[Individual (Model) and Population Implementation: 13](#_Toc164344912)

[Get Action and Individual Play Test 16](#_Toc164344913)

[Rewards Modification & Survival 17](#_Toc164344914)

[New Population Generation & Mutation 19](#_Toc164344915)

[Training Hyperparameters & Combined Evolution 21](#_Toc164344916)

[Final Result 23](#_Toc164344917)

[Potential Improvements 24](#_Toc164344918)

[Conclusion: 25](#_Toc164344919)

**Project Overview:**

The project aims to tackle reinforcement learning environments using genetic algorithms, a powerful optimization technique inspired by natural selection. Genetic algorithms operate by evolving a population of candidate solutions over multiple generations, iteratively improving their fitness through selection, crossover, and mutation operations. By applying genetic algorithms to reinforcement learning problems, the project seeks to optimize agent policies and strategies for maximizing rewards in complex and dynamic environments. With potential applications in robotics, gaming, and autonomous systems, the project endeavors to develop robust and adaptable solutions that can effectively navigate diverse challenges and scenarios. Through this interdisciplinary approach, the project aims to advance the field of reinforcement learning and contribute to the development of intelligent and adaptive systems capable of addressing real-world problems.

## **Covering Concepts:**

1. **Reinforcement Learning (RL):** Reinforcement learning is a machine learning paradigm where an agent learns to make decisions by interacting with an environment. It receives feedback in the form of rewards or penalties based on its actions, aiming to maximize cumulative rewards over time.
2. **Genetic Algorithm (GA):** Genetic algorithms are optimization techniques inspired by the process of natural selection and genetics. They use principles such as selection, crossover, and mutation to iteratively evolve a population of candidate solutions towards optimal or near-optimal solutions.
3. **Evolutionary Computing:** Evolutionary computing encompasses optimization algorithms inspired by biological evolution, including genetic algorithms. These algorithms iteratively improve candidate solutions by simulating processes such as selection, reproduction, and mutation.
4. **Population:** In genetic algorithms, a population refers to a collection of candidate solutions (individuals) representing possible solutions to the optimization problem. The population evolves over multiple generations, with each individual encoded as a set of parameters or genes.
5. **Generation:** A generation in genetic algorithms refers to a single iteration of the optimization process. During each generation, the population undergoes selection, reproduction (crossover and mutation), and evaluation to produce a new generation of individuals.
6. **Natural Selection:** Natural selection is a key component of genetic algorithms, mimicking the process of survival of the fittest in biological evolution. Individuals with higher fitness (better solutions) have a higher probability of being selected for reproduction, passing their genes to the next generation.
7. **Crossover:** Crossover is an operator in genetic algorithms where pairs of parent solutions exchange genetic information to produce offspring solutions. It promotes exploration of the solution space by combining beneficial traits from different parents.
8. **Mutation:** Mutation is a genetic operator that introduces random changes to individual solutions, allowing for exploration of new regions of the solution space. It helps maintain diversity in the population and prevent premature convergence to suboptimal solutions.
9. **Migration:** Migration is a mechanism in genetic algorithms where individuals migrate between populations, promoting information exchange and diversity. It can prevent populations from getting stuck in local optima and facilitate global exploration of the solution space.
10. **Reinforcement Learning with Genetic Algorithm:** Combining reinforcement learning with genetic algorithms involves using genetic algorithms to optimize the parameters or policies of a reinforcement learning agent. Genetic algorithms guide the exploration of the agent's policy space, enhancing its ability to learn effective strategies in complex environments.
11. **Integration of RL and GA**: The integration of reinforcement learning and genetic algorithms involves designing a framework where genetic algorithms optimize the parameters or policies of a reinforcement learning agent. This integration enables the agent to adapt and improve its behavior over successive generations through evolutionary principles.
12. **Evaluation Metrics:** Evaluation metrics such as average reward, convergence speed, and solution quality are used to assess the performance of reinforcement learning with genetic algorithm approaches. These metrics measure the effectiveness and efficiency of the optimized policies learned by the agent through the evolutionary process.

## **Potential Use Case:**

Reinforcement learning (RL) has a wide range of potential use cases across various domains due to its ability to learn optimal behaviors through trial and errors. The potential usages are as such:

1. **Autonomous Robotics**: RL can be used to train robots to perform complex tasks such as navigation, manipulation of objects, or assembly line operations. Robots can learn from their interactions with the environment to optimize their actions and adapt to dynamic situations.
2. **Game Playing**: RL algorithms have demonstrated impressive performance in playing complex strategy games like chess, Go, and video games. These algorithms can learn effective strategies and tactics by playing against opponents or exploring different game environments.
3. **Recommendation Systems**: RL can be used to personalize recommendations in various domains such as e-commerce, content streaming, and online advertising. By learning from user interactions, RL algorithms can optimize recommendations to maximize user engagement or satisfaction.
4. **Finance and Trading**: RL algorithms can be applied to optimize trading strategies in financial markets by learning from historical data and market dynamics. They can adapt to changing market conditions and optimize portfolio management to maximize returns while managing risk.
5. **Supply Chain Management**: RL can optimize inventory management, logistics, and routing decisions in supply chain operations. By learning from historical data and real-time information, RL algorithms can optimize decision-making to reduce costs, minimize delays, and improve overall efficiency.
6. **Healthcare**: RL can be used for personalized treatment planning, drug discovery, and medical diagnosis. RL algorithms can learn optimal treatment policies by analyzing patient data and treatment outcomes, leading to more effective and personalized healthcare interventions.
7. **Energy Management**: RL can optimize energy consumption, production, and distribution in smart grid systems. RL algorithms can learn to control energy generation and storage systems to maximize efficiency, reduce costs, and minimize environmental impact.
8. **Advertising and Marketing**: RL can optimize online advertising campaigns by learning to allocate resources effectively across different channels and targeting strategies. RL algorithms can adapt to changing user preferences and market trends to maximize advertising ROI.
9. **Automated Decision Making**: RL can automate decision-making processes in various domains such as customer service, resource allocation, and scheduling. RL algorithms can learn to make optimal decisions in complex and uncertain environments, reducing the need for human intervention.
10. **Drug Discovery and Development**: RL can be used to optimize drug discovery pipelines by guiding the selection and optimization of candidate compounds. RL algorithms can learn to design and test molecules in silico, accelerating the drug discovery process and reducing the cost of bringing new drugs to market.

## **Environment Used:**

Python programming language with:

1. gym = ^0.26.2

2. swig = ^4.1.1

3. box2d-py = ^2.3.8

4. pygame =^ 2.5.2

6. tensorflow = ^2.15.0

7. keras = ^2.15.0

8. numpy = ^1.26.3

## **Reinforcement Learning Environment Used**

The Box2D car racing environment in Gymnasium is a dynamic simulation designed for training reinforcement learning agents in the domain of autonomous driving. With a 2D physics engine, this environment features a customizable race track with various terrains, curves, and obstacles. Agents control virtual cars to navigate the track, adjusting throttle, steering, and braking to optimize lap times while avoiding collisions. Observations provided to the agents is a top-down image of the car and race track, enabling them to make informed decisions based on environmental cues. The environment has a reward system incentivizing efficient racing and strategic decision-making. The Box2D car racing environment offers a challenging yet accessible platform for developing autonomous driving capabilities.

A video game screen with a red car on it

Description automatically generated

Fig 1. Sample Game Image

## **Reinforcement Learning Environment Understanding**

Necessary steps and documentation researching are needed in order to fully understand and solve the environment.

1. **Rendering the Game Window:** Rendering the game window involves visualizing the car racing environment to the agent, enabling observation and interaction. This step encompasses rendering graphical elements, providing essential information for users to understand the environment.

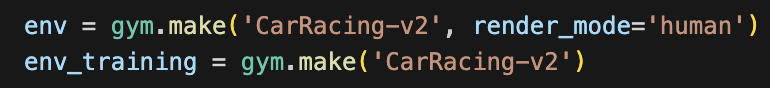
****

Fig 2. Code Implementation to Render Game Window

A screenshot of a video game

Description automatically generated

Fig 2. Sample Game Window

1. **Action Space Determination:** Finding out the action space involves identifying all possible actions the agent can take while navigating the racetrack. In the Gymnasium car racing environment, the action space consists of a set of 2 continuous actions ranging from 0 to 1, being acceleration and braking and 1 continuous action ranging from -1 to 1, which stands for the steering adjustments done to the car.

**A screenshot of a computer

Description automatically generated**

Fig 3. Code Implementation to Find out Action Space

**A black background with white text

Description automatically generated**

Fig 4. Gymnasium’s Documentation on Car Racing Environment’s Action Space

1. **Observation Space Exploration:** Exploring the observation space entails understanding the range of information available to the agent about its environment. In the car racing environment, observations are a top-down 96x96 RGB image of the car and racetrack, making the dimension of the observation (96,96,3).

**A computer screen with white text

Description automatically generated**

Fig 5. Code Implementation to Find out Observation Space.

**A black and white text

Description automatically generated**

Fig 6. Gymnasium’s Documentation on Car Racing Environment’s Observation Space

1. **Reward Definition:** Defining the game reward involves specifying the feedback provided to the agent after each action, reflecting the consequences of its decisions during the race. The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track. For example, if you have finished in 732 frames, your reward is 1000 – 0.1\*732 = 926.8 points.

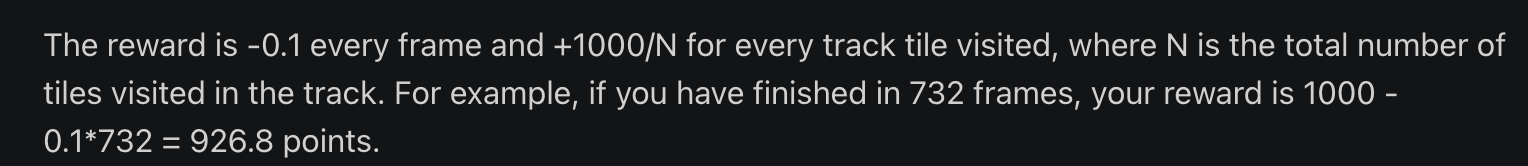


Fig 7. Gymnasium’s Documentation on Car Racing Environment’s Rewards

1. **Termination Conditions Establishment:** Establishing termination conditions determines when an episode or race ends, signifying the completion of a racing task or achieving specific objectives. In the Gymnasium car racing environment, termination occur when the car visits all the tiles or after 1000 steps in game environment.

**A computer screen with colorful text

Description automatically generated**

A screenshot of a computer

Description automatically generated

Fig 8&9. Code Implementation and Result to Find out Truncated Situation

## **Evolutionary Computing Concepts:**

Evolutionary computing is a computational paradigm inspired by the principles of biological evolution and natural selection. It encompasses a family of optimization algorithms that simulate evolutionary processes to solve complex problems. At its core, evolutionary computing involves the iterative evolution of a population of candidate solutions, where each solution represents a potential solution to the problem at hand. Through a process of selection, reproduction (crossover and mutation), and evaluation, better solutions are progressively generated and refined over successive generations.

## **Individual (Model) and Population Implementation:**

To implement evolutionary computing effectively, the initial step involves identifying suitable candidates to operate within the environment. In the realm of reinforcement learning, we conceptualize a neural network model as an individual, with its observations serving as input and its resulting actions as output within the environment.

Given that the environment's observations consist of RGB images, and the task requires three continuous actions as output, the model architecture is thus crafted to fulfill this objective. Each individual (representing a model) requires an input of dimention (96,96,3) cporresponding to a RGB image input, it also comprises two convolutional layers with depths of 32 and 64, strategically designed to extract pertinent information and generate a cohesive context vector. Subsequently, this unified context vector is channeled into three distinct dense layers, tailored to produce predictions for the actions to undertake. The dense layer corresponding to steering is activated by the hyperbolic tangent (tanh) function, confining the output within the range of -1 to 1 to meet the required specifications. Conversely, the dense layers associated with acceleration and braking are activated by the sigmoid function, shaping the output to fall between 0 and 1, aligning with the anticipated action range.

A computer screen shot of a program code

Description automatically generated

Fig 10. Code Implementation for Make Individual Function

A diagram of a computer program

Description automatically generated with medium confidence

Fig 11. Individual Model Structure Visualization

Individual models are not able to evolve, only a population of them can. Therefore, a Get population function is then defined to return an array of individuals by repeatedly calls the make individual function until a given size is reached.

A screen shot of a computer code

Description automatically generated

Fig 12. Code Implementation to Define Get Population Function

## **Get Action and Individual Play Test**

After individuals are defined, it is important to define a get action function in order to call it recursively during a game, allowing individuals to perform continuously in the environment. As neural network defined using Keras API requires batch data, the observation is first expanded using Numpy’s expand dimension function. Afterwards, the observation is fed into individual to produce 3 actions, these actions are in the form of array and thus requires indexing to access.

A computer screen with text

Description automatically generated

Fig 13. Code Implementation to Define Get Action Function

The process of running individuals by recursively calling the "get action" function is encapsulated within a larger function called "Run individual." In this function, each individual is configured to participate in the game a few times with a fresh environment every time to reduce the likelihood of a model excelling at a specific racetrack, thereby ensuring they remain generalized. The final rewards are determined by averaging the model's performance across the game sessions.

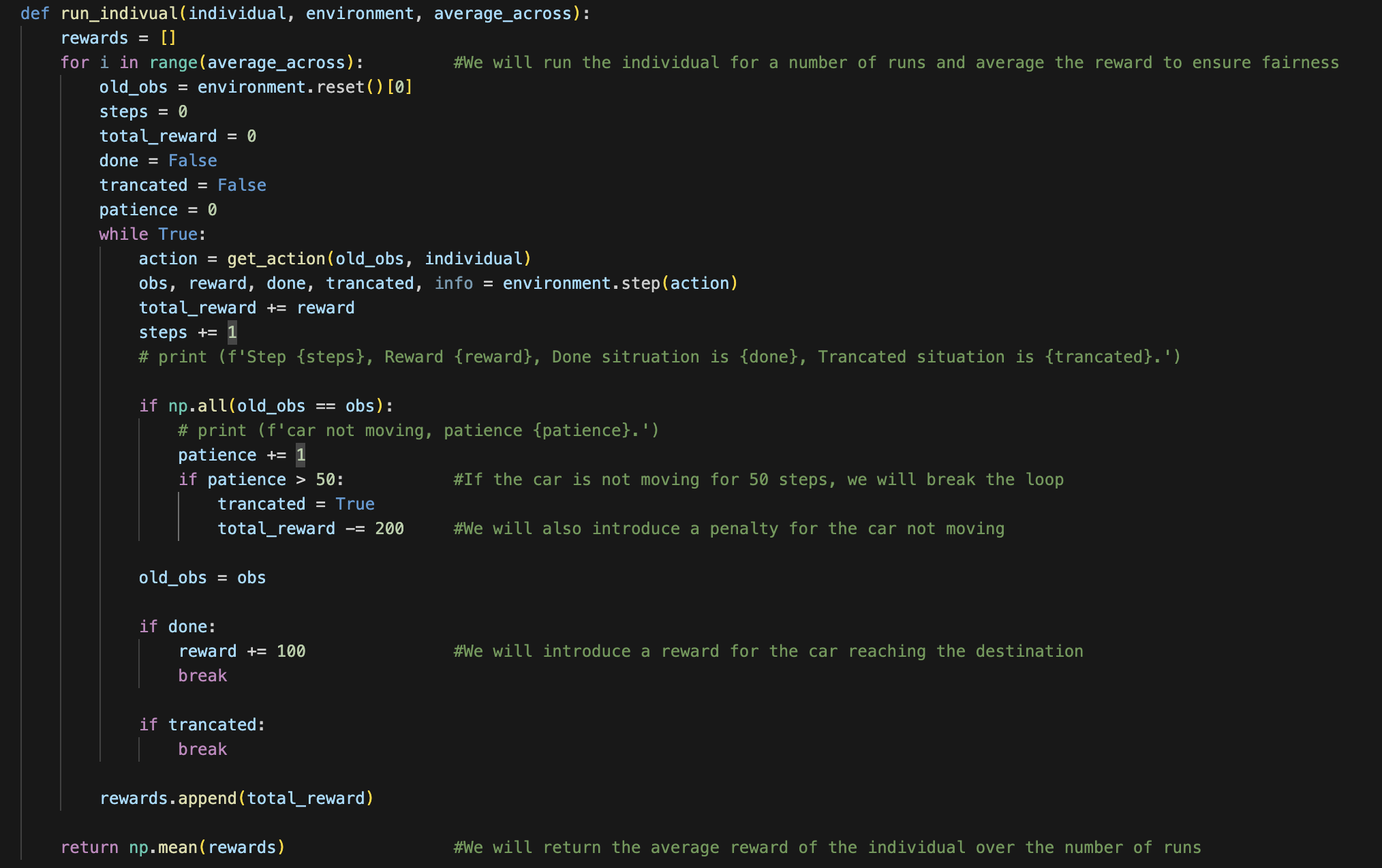


Fig 14. Code Implementation to Define Run Individual Function

## **Rewards Modification & Survival**

The first few training episodes are rendered to visualize any potential problems that requires attention. One interesting discovery was that there are a lot of individuals just don’t move at all. This is because the action representing braking is constantly outputting 1, which mask the other 2 actions. As the car is not moving, the observation acquired by the individual will stay the same, resulting the same output being produced and individual will never move as a result, until it use up all the actions and terminates the game.

To address the issue of not moving, a patience threshold of 50 is set in the play test function. The patience limit will receive a +1 for every same observations observed by the individual and once the threshold is reached, the game will be terminated early and individual will receive a reward of -200, heavily penalizing their performance. This way, computational resources will be saved and model is almost guaranteed to be ranked as the last few in terms of performance will not survive to the natural selection implemented.

A screen shot of a computer

Description automatically generated

Fig 15. Code Implementation to Modify the Rewards Given

A bigger function called the “Run Population” defined encapsulating the “Run Individual” function. It calls the run individual function for every individual in a given population and returns a dictionary of model and rewards pairs arranged according to their rewards.

A computer screen with text

Description automatically generated

Fig 16. Code Implementation Define Run Population Function

The “survival of the fittest” function is then defined to take in the arranged dictionary and returns a percentage of the dictionary from the top, they are considered as the survivals of this natural selection.

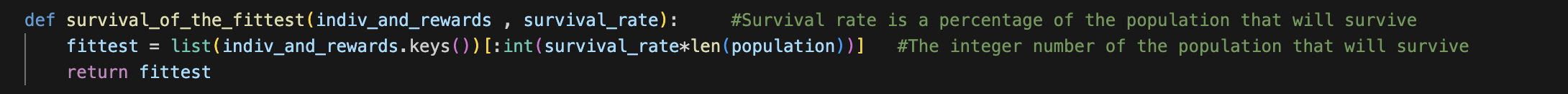


Fig 17. Code Implementation Define Survival of the Fittest Function

## **New Population Generation & Mutation**

The new survivals will then be used as the source to generate new population. If every neural network model is seen as individual, then the layers inside the model can be seen as genes and the weights and bias in the layers can be seen as DNA.

A function called the “gene crossover” is defined to return a gene which is an offspring of the 2-input gene. The new gene is produced by taking a random length of the first gene and added the remaining of the second gene. To mimic real life situation and to attained the mentioned benefits, the produced gene are also put under a small chance of mutation. Mutated gene will be a layer of randomly generated weights and bias to simulate the devastating effect of mutation.

A screen shot of a computer program

Description automatically generated

Fig 18. Code Implementation Define Gene Crossover Function

Another wrapper function called the “reproduction” is defined to generate a new population base on the given survivals and mutation rate. The functionality is achieved by first randomly select a pair of survivals and call the “Gene Crossover” Function on every single layer of the 2 gene to produce a new offspring. The process is done iteratively until a set number of new population is generated.



Fig 19. Code Implementation Define Reproduction Function

To introduce some variance that can potentially get population out of local minimal, immigrants are also introduced. A small percentage of the population will be individuals that are randomly generated, not an offspring of any of the survivals.





Fig 20. Code Implementation that sets Immigration Number

## **Training Hyperparameters & Combined Evolution**

With all the functions defined, evolution of the population can be simulated by combining all the functions. During the training, the following hyperparameters are used:

1. **POPULATION\_SIZE (100):** The population size determines the number of individuals (models) present in each generation. A larger population size allows for greater exploration of the solution space and increases the diversity of solutions. With a population size of 100, there are enough individuals to sample a wide range of possible strategies while still being computationally feasible.
2. **GENERATIONS (40):** The number of generations defines how many iterations of the genetic algorithm will be executed. Each generation represents a cycle of selection, reproduction, and evaluation. A larger number of generations allows for more opportunities for individuals to evolve and improve their performance over time.
3. **STARTING\_GENERATION (5):** Starting the genetic algorithm from the fifth generation instead of the first can be beneficial for several reasons. It allows for the initial population to be generated using more sophisticated techniques or pre-trained models, potentially accelerating the convergence towards optimal solutions. Additionally, starting from a later generation reduces the computational overhead of the early generations, which may be less effective due to random initialization.
4. **AVERAGE\_ACROSS\_GAME\_SESSION (2):** Averaging the performance of each individual across multiple game sessions helps reduce the variability in performance and provides a more reliable estimate of their capabilities. By averaging across two game sessions, a more stable measure of an individual's performance can be obtained while still keeping the computational overhead manageable.
5. **SURVIVAL\_RATE (0.2):** The survival rate determines the proportion of individuals that survive to the next generation based on their fitness scores. A survival rate of 0.2 means that only the top 20% of individuals, in this case 20 individuals with the highest fitness scores are selected to reproduce and pass on their genetic information to the next generation. This ensures that the fittest individuals are preserved while allowing for some diversity in the population through selective breeding.
6. **MUTATION\_RATE (0.01):** The mutation rate controls the probability of introducing random changes (mutations) to the genetic information of individuals during reproduction. A low mutation rate of 0.01 ensures that mutations occur infrequently, preventing excessive disruption of successful traits while still allowing for exploration of new genetic combinations.
7. **IMMIGRATION\_RATE (0.1):** The immigration rate determines the proportion of individuals in each generation that are replaced by new randomly generated individuals. Introducing new individuals through immigration helps maintain genetic diversity in the population and prevents premature convergence to suboptimal solutions by introducing fresh genetic material from outside sources. A rate of 0.1 ensures a moderate level of immigration while still predominantly relying on evolution through selection and reproduction.

A screen shot of a black screen

Description automatically generated

Fig 21. Training Hyperparameters

A screen shot of a computer program

Description automatically generated

Fig 22. Code Implementation that Defines Evolution Function

## **Final Result**

After 40 generation of evolving, the best individual from generation is able to achieve a reward of 294.2 from the initial -38.14. The generational average also increases from -167 to 88.54. The fitted individual is able to perform in a fresh environment set up by following the track and thus we can determine the success of Genetic algorithm as a optimization technique for reinforcement learning problem.

A graph of blue and orange lines

Description automatically generated

Fig 23. Training History Visualization

## **Potential Improvements**

Some potential improvements that can be implemented includes:

* 1. Further finetune the model by introducing mutation rate decay and changing other parameters.
  2. Further finetune the model by introducing immigration rate decay and changing other parameters.
  3. Further finetune the model by training more episodes until generation converges.

**Conclusion:**  
In conclusion, this project has delved into the realm of evolutionary computing to tackle reinforcement learning problems, offering valuable insights and practical solutions. Through the systematic application of genetic algorithms, the project has demonstrated the efficacy of evolutionary techniques in optimizing neural network architectures for navigating complex environments. By carefully fine-tuning parameters such as population size, mutation rate, and immigration rate, the project has effectively balanced exploration and exploitation to evolve robust and adaptable reinforcement learning agents.

The project's exploration of evolutionary computing has shed light on its potential to address challenges such as overfitting, computational complexity, and the need for generalization in reinforcement learning tasks. By iteratively refining candidate solutions over multiple generations, the project has showcased the power of evolutionary algorithms to discover novel strategies and adapt to dynamic environments.

Furthermore, the project's findings underscore the importance of thoughtful parameter selection and experimental design in maximizing the effectiveness of evolutionary computing techniques. Through rigorous experimentation and performance evaluation, the project has provided valuable insights into the trade-offs involved in designing evolutionary algorithms for reinforcement learning applications.

In summary, this project's utilization of evolutionary computing represents a significant step forward in the quest to develop robust and scalable solutions for reinforcement learning problems. By leveraging the principles of natural selection and genetic optimization, the project has laid the groundwork for future advancements in evolutionary reinforcement learning methodologies, paving the way for more efficient and adaptable autonomous systems in various domains.