ML Term Project Final Report: AlphaGame

Anran Du, Jien Li, Hao Niu, Yicheng Shen, Qinzi Zhang

Abstract

This paper presents AlphaGame, a project that utilizes reinforcement learning algorithms as cores of AI to play simple games. We first discuss how the theories of Deep Q-Learning and genetic algorithm relate to our design choices and implementation. Then, we explain methods which include our model architectures and real-time data collection methods, followed by a discussion of our results. We evaluate the performance of two AI on three games, Snake, Pong, and Flappy Bird.

1 Introduction

In recent years, machine learning is widely used for a variety of applications, including building AI that mimics real human beings in making the most favorable decision. One prominent achievement in the field is the famous AlphaGo AI defeating a human professional player. The algorithm of AlphaGo [2] achieved high winning rate by training on deep neural networks and utilizing reinforcement learning. Amazed by the great potential of AI, we hope to research about and implement algorithms that learn to perform decision making in simple games.

Our project titled **AlphaGame** is an *application focused* project, which aims to build AI to play Snake, Pong, and Flappy Bird. According to Kaelbling et al.'s survey of reinforcement learning [1], there are two general approaches of *reinforcement learning*. The first approach aims to estimate the utility (reward function) of taking actions in states of the world and the second approach aims to find one behavior that performs well in the environment.

In our project, we investigate both approaches. We implement the first approach as a *Deep Q-Learning* algorithm, which is relatively new but widely used in training AI for games. We build a specific deep neural network, named Deep Q-Network (DQN), which learns the reward function. The second approach is implemented as a *genetic algorithm*, which is a traditional way of implementing AI. This algorithm selects the best players in every generation to let the AI evolve. We compare the performance of two approaches and study which one performs better on each of the three games.

The following section briefly discusses the theories behind our design and implementation.

Deep Q-Learning A reinforcement learning algorithm trains an *agent* to perform a specific task in an *environment*. Without knowing how the task is to be achieved, the algorithm trains the agent by rewarding and punishing its actions [1]. In a typical reinforcement learning problem, there are three main components.

- State space $\mathcal{S}$.

  Each element $s \in \mathcal{S}$ represents a specific state of the environment. For example, in the Snake game, s includes the essential information of the game such as the snake's configuration and the apple's position.

- Action space $\mathcal{A}$.

  The action space consists of all actions of the agent. For example, in the Snake game, each action $a \in \mathcal{A}$ corresponds to one action of going up, down, left, and right.

- Reward function $\mathcal{R}$.

  There is a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that maps a tuple of a state and an action to a real value, denoted as the reward of the action.

The goal of a reinforcement learning algorithm is thus to maximize the expected total rewards in all rounds:

$$\mathbb{E}\left(\sum_{k=1}^{\infty} \gamma^t r_t\right), \quad r_t = \mathcal{R}(s_t, a_t), \tag{1}$$

and $\gamma \in [0, 1]$ is the discount factor of future rewards.

Genetic Algorithm Genetic algorithm, also called evolutionary algorithms, is inspired by the process of natural selection, where the fittest individuals of a population are selected to produce offspring. These offspring inherit traits from their parents and produce offspring of their own. The process repeats until a generation with the fittest individual is found.

Six phases are considered in a genetic algorithm.

- Individual

  An individual is the *agent* we are training. Each individual is characterized by its set of *genes*. *Genes* are joined to form a *chromosome*. An individual can perform action according to the outputs of its *chromosome*.

- Population

  Each generation consists of a population made up of a fixed number of individuals.

- Fitness Function

  The fitness function determines how fit an individual is. Each individual has a fitness score determined by the fitness function.

- Selection

  The goal of selection is to select the fittest individuals from the population of a generation as candidates to reproduce the population for the next generation. The fitness of an individual is determined by its fitness score. There are many methods in selecting, but, generally, an individual with a higher fitness score is more likely to be selected.

- Crossover

  Crossover is used to produce the chromosome for individuals of the next generation, where one individual selected will have a section of its chromosomes replaced with the same section from another selected individual.

- Mutation

  After crossover, some genes of an individual can be subjected to a mutation with low random probability. This is to ensure diversity in the gene pool within the population.

Challenges One key challenge in the Deep Q-Learning approach is the definitions of the state and reward spaces. Incorrect definitions will lead the AI to learn undesired behaviors. Furthermore, for different games, these definitions vary. For each game, we need to carefully classify the actions that we want to reward and the actions that we want to penalize and assign proper weights to the functions.

Similar to definition of reward in Deep Q-Learning, setting a proper fitness function is crucial to ensure good convergence. An unfit fitness function may lead to immature convergence in local minimum. To combat this, we took into multiple considerations when setting the fitness function and compared several configurations for the most optimal result.

2 Methods

In this section, we review our methodologies and design choices for both Deep Q-Learning and genetic algorithms. The Deep Q-Learning algorithm is explained in greater details because of its novelty and the genetic algorithm is briefly covered since it is a classic approach.

2.1 Environment and Agent

Before we train our reinforcement learning algorithms, we need to simulate the interaction between the environment and the agent as a mimic of the actual learning task (i.e., playing a game in our project). Figure 1 is a schematic diagram of this interaction. In each round, the agent receives the state s from the environment and makes an action a in the action space. The environment then updates the state to s' and sends the agent the reward $r = \mathcal{R}(s, a)$.

We implement the environment and the agent that will be fed into the reinforcement learning algorithms as follows.

Environment Model The environment model simulates the game environment, i.e., how a game responds to the player's actions. We implement different environments for each specific game we want to train. The environment models contain the following functions:

- reset(): The environment initializes with a random initial state.
- forward(a): Upon receiving an action $a \in \mathcal{A}$ from the agent, the environment returns: (1) the new state s after action a; (2) the reward r corresponding to a.

Agent Model The agent model is equipped with a deep learning model (which we will discuss in the next part). It will make an action based on the current state during the simulation. It also learns from the collected data at the end of each game. The agent model contains the following functions:

- act(s): Given state $s \in \mathcal{S}$, the agent returns an action a.
- train(D): The agent trains its learning model with a mini-batch $\xi \sim D$.

2.2 Deep Q-Network

A Q-learning algorithm predicts the transition function and the reward function of the model and uses this information to predict an optimal action [1]. Moreover, recent experiments and studies have shown that deep neural networks have high generalization performance [4]. Therefore, we adopted the Deep Q-Network (DQN) that combines the deep network with the Q-learning algorithm.

Q-Learning under MDP In our model, the action of the agent determines the next state. Moreover, the reward function depends not only on the state and action in the current round, but also the state in the next round. For example, we punish the snake agent for moving away from the apple. To check whether the snake moves away, both states are necessary. This kind of reinforcement learning problem is categorized as *delayed reinforcement*, and is often modelled as *Markov Decision Processes* (MDP) [1].

Denote $Q^*(s.a)$ be the expected optimal cumulative reward (also denoted as expected reinforcement) after taking action a in state s and then taking an optimal action in the new state s'. Then the sequence $\{Q_0^*, Q_1^*, \ldots\}$ is a Markov chain of expected reinforcements in each

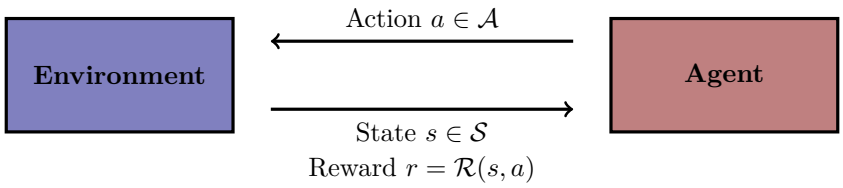

Figure 1: Schematic diagram of environment-agent interaction

round, where $Q_t^* = Q^*(s_t, a_t)$. Given the transition function $T(s, a, s')$ such that

$$T(s, a, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a),$$

the expected reinforcement can be written recursively as

$$Q^*(s, a) = \mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') \max_{a' \in \mathcal{A}} Q^*(s', a'). \quad (2)$$

The goal of the algorithm is thus to train the agent to learn the target function $Q^*(s, a)$, while T and $\mathcal{R}$ are unknown a priori. The Q^* function is essential to learn the task because it defines the optimal policy function $\Pi : \mathcal{S} \to \mathcal{A}$ as follows.

$$\Pi(s) = \operatorname*{argmax}_{a \in \mathcal{A}} Q^*(s, a). \quad (3)$$

In the real world, while T and $\mathcal{R}$ remain unknown, a Q-learning algorithm predicts the optimal Q^* function by training on the *experience tuple* (s, a, r, s'), where s, a are the state and action in round t, r is the immediate reward from the environment, and s' is the next state in round $t+1$. The traditional Q-learning algorithm approximates Q^* with the following updating rule:

$$Q(s, a) \leftarrow Q(s, a) + \eta(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s, a)). \quad (4)$$

Tsitsiklis and Van Roy [3] have proved that under certain diminishing step size η, this function $Q(s, a)$ converges to the optimal reinforcement Q^* defined in Equation (2) with probability 1. However, in our model, we instead use a deep neural network to accomplish the learning objective.

Data Collection Similar to a typical reinforcement learning algorithm, our algorithm collects necessary data from the simulation of the games, i.e., the interaction between the environment and the agent described in Section 2.1. More specifically, in each round of the interaction, the agent stores the experience tuple (s, a, r, s') in its memory set $\mathcal{M}$. The following algorithm illustrate this data collection process.

Algorithm 1 Deep Q-Network: data collection

```
for epoch from 1 to N do
    Initialization: s ← env.reset();
    while game does not end do
        a ← agent.action(s);
        s', r ← env.forward(a);
        agent.memorize(s, a, r, s');
    end while
end for
```

The training set and the target set are induced from the stored experience tuples. Recall the learning objective $Q^*(s, a)$ defined in Equation (2). To simplify the computation, we convert the target functions $Q^*(s, a)$ of different $a \in \mathcal{A}$ into a vector form such that

$$\overline{Q}^*(s) = [Q^*(s, a_1), \ldots, Q^*(s, a_{|\mathcal{A}|})]^T. \quad (5)$$

Therefore, the training set is the set of all states in the current round, i.e.,

$$\mathcal{X} = \{s : (s, a, r, s') \in \mathcal{M}\}. \quad (6)$$

The target set is more difficult to obtain. It is hard to compute an explicit value of Q^* from its recursive definition in Equation (2). Instead, we assume that the DQN algorithm has learned an approximate function that is close enough to Q^*, i.e., $\mathrm{DQN}(s) \approx \overline{Q}^*(s)$. Therefore, according to Equation (2), the target is computed as $y(s, a, r, s') = [y_1, \ldots, y_{|\mathcal{A}|}]^T$, where

$$y_k = \begin{cases} \mathrm{DQN}(s)[k] & \text{if } \mathcal{A}[k] \neq a, \\ r + \gamma \max(\mathrm{DQN}(s')) & \text{if } \mathcal{A}[k] = a. \end{cases} \quad (7)$$

Note that target set is *dependent* of the DQN model. Therefore, in practice, we first choose a random batch $\xi \subset \mathcal{M}$, and then train this model with one experience tuple at a time. In other words, each time we pick an experience tuple $(s, a, r, s') \in \xi$, and then fit our model with input $= s$ and label $= y(s, a, r, s')$ defined above.

DQN architecture Figure 2 is a schematic diagram of our DQN network architecture. The input layer has size $|\mathcal{S}|$. The hidden layers consist of three dense layers, each of size 128. Each dense layer has a relu activation function. The output layer is a fully connected dense layer of size $|\mathcal{A}|$. In other words, given a state $s \in \mathcal{S}$, this

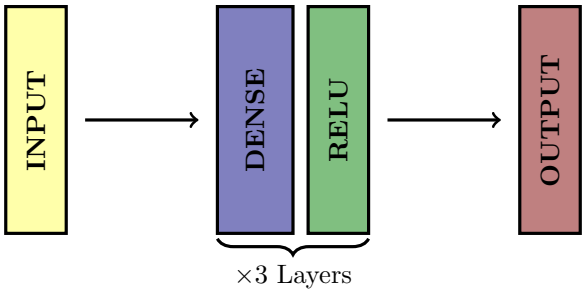

Figure 2: Schematic diagram of DQN architecture

DQN network takes s as an input and outputs a vector $v \in \mathbb{R}^{|\mathcal{A}|}$, where each element is an estimate of $Q^*(s, a)$.

The loss function of the DQN network is chosen to be MSE because the output and the target are both vectors of real numbers. We also choose Adam as the optimizer, which has the best performance among all the optimizer we have tested.

Reward Function An important task of reinforcement learning is to define a reasonable reward function $\mathcal{R}(s, a)$ which will guide the agent to learn the task, and a poorly designed reward function will likely make the agent learn bad behaviors. We define the reward function for each game as follows:

Pong Reward definition for Pong is straightforward. The intuition is to encourage the agent (the paddle) to chase the ball. Therefore, we reward the action of moving closer (horizontally) to the ball by 10 and punish the action of moving away by -10.

Snake Initially, we only defined non-trivial rewards for two events: eating an apple is rewarded by 10 and dying is rewarded by -100. In other cases, the reward is 0. However, as we trained the model, we observed that the snake continuously made random actions. Although it learned to avoid the walls, it failed to learn to eat the apples. To encourage the agent to eat the apples, we add a pair of rewards. We reward the action of moving closer to the apply by 1 and punish the action of moving away by -1. This new reward function effectively encouraged the snake to act smarter.

Flappy Bird The initial reward definition is straightforward. The bird staying alive is reward by 1 and dying is rewarded by -100. To further improve performance, we added an additional penalty when the bird collides with the upper tube which indicates a mistaken jump.

Exploration v.s. Exploitation One important feature of the DQN algorithm is the trade-off between exploration and exploitation. The agent *explores* by taking a random action in the action space, and it *exploits* by predicting an optimal action that maximizes the reward. As [1] points out, if the agent always exploits, then this greedy strategy might lead to a sub-optimal solution. In order to solve this issue, one common strategy is to introduce randomness in the training process.

More specifically, we denote $\epsilon \in [0, 1]$ as the exploration constant, which is the probability that the agent explores in a round. ϵ initialized as 1 at the beginning of the training process. As the training proceeds, ϵ decreases in an exponential speed, i.e., $\epsilon_{t+1} = \eta_\epsilon \epsilon_t$, where η_ϵ is the decay constant of ϵ. We also set a lower bound of ϵ to be 0.01 so that there is always a small chance that the agent takes a random step.

2.3 Genetic Algorithm

In addition to DQN, we also want to explore the domain of evolutionary algorithm, in particular genetic algorithm. The learning process of genetic algorithm differs from that of DQN in that the actions performed by the agents are random. To explain why this is, we will build on the six building blocks of genetic algorithm we discussed in the previous section.

Individual What makes our algorithm unique from the traditional genetic algorithm is that the chromosome of each individual is a fully connected neural network. This implementation utilizes the advantages of machine learning and makes each individual's decision space more complex and the total gene pool more diverse. The input to the neural network is some states in the game that the individual is observing, and output of the neural network is the action the individual should take.

In the initial population, the weights and biases of each individual is random-uniformly chosen to be a number between -1 and 1. This is the building block of the training process–the more random and spread out our initial population is, the more likely we will find a trace that will lead us to a good model.

Fitness Function How our fitness function is defined and how the fitness is defined varies from game to game, but they all share the same goal–to find the most favorable candidates out of the population to reproduce.

What we mean by most favorable is the individual with the most likely chance to do well in the game or beat the game.

Snake For the snake game, in training a genetic algorithm to play snake, our fitness function is defined as follow:

$$\text{fitness} = m + (2^{\lambda} + 500\lambda^{2.1}) - (0.25m^{1.3}\lambda^{1/2}),$$

where m is the total steps the snake took during the generation and λ is the total number of apples the snake ate during the generation (game score). The goal for this fitness function is to 1) reward snakes early on for exploration; 2) have an increasing reward for snakes as they find more apples; and 3) penalize snakes for taking a lot of steps.

Pong In similar fashion, the fitness function for Pong is defined as follow:

$$\begin{aligned}\text{fitness} = {} & 200(2^{\lambda} + \lambda^{2.1}) + 400(1 - \min\{d_p/d_b, 1\}) \\ & + 0.5(w - \delta),\end{aligned}$$

where λ is the number of times the paddle return the ball, d_b is the total distance in the x-direction travelled by the ball, d_p is the total distance in the x-direction travelled by the paddle, w is the width of the screen, and δ is the distance between the ball and the paddle when the paddle did not catch the ball. This setup encourages the paddle to catch the ball while taking as few steps as possible.

Flappy Bird The Flappy Bird fitness function follows a similar structure in the sense that the fasted growing component is exponential. The function is as follows:

$$\text{fitness} = 200(2^{\lambda} + 2.1\lambda) - 1.2(x + |y|) + 1000,$$

where λ is the raw score proportional to the number of pipes passed. This is a float that increases each frame and would increase by 1 between consecutive pipes. The game score is then a integer calculated from this raw score to represent the number of pipes passed. x and y are the horizontal and vertical distance, in terms of pixels between the bird and the center of the next opening between the pair of pipes. The raw score is used in an exponential term to better differentiate between birds that died at the entry of the opening and birds that died deeper into the openings. Dying further from the pipe opening is penalized by the x, y term. The constant of 1000 is used to keep the fitness value positive, since x, y can induce rather large penalties.

Selection In the selection of the fittest individual, we used the classic method called Roulette-wheel Selection. It randomly picks an individual out of the population; however, the individual with higher fitness score are given more weights and are more likely to be selected.

Algorithm 2 Selection: Roulette-wheel

```
Initialization wheel ← sum(individual.fitness for individual in population);
Initialization pick ← random.uniform(0, wheel);
Initialization current ← 0;
for individual in population do
    current ← current + individual.fitness;
    if current > pick then return individual;
    end if
end for
```

Crossover Crossover is the process of reproduction of selected individuals. Simulated binary crossover (SBX) is used here to generate offspring symmetrically around the parents. This symmetry ensures the reproduction only induces variation and not bias. The degree of variation between the parents and the offspring follows a standard Gaussian distribution and can can be controlled by a hyper-parameter η. A higher η would result offspring that are closer to the parent chromosome.

Two parents $x_1(t)$, $x_2(t)$ are used to produce two offspring $\tilde{x}_1(t)$, $\tilde{x}_2(t)$, by composing each gene j according to the following equations:

$$\tilde{x}_{1j}(t) = 0.5\left[(1+\gamma_j)x_{1j}(t) + (1-\gamma_j)x_{2j}(t)\right] \quad (8)$$

$$\tilde{x}_{2j}(t) = 0.5\left[(1-\gamma_j)x_{1j}(t) + (1+\gamma_j)x_{2j}(t)\right] \quad (9)$$

where

$$\gamma_j = \begin{cases} (2r_j)^{\frac{1}{\eta+1}} & \text{if } r_j \leq 0.5, \\ \left(\frac{1}{2(1-r_j)}\right)^{\frac{1}{\eta+1}} & \text{otherwise.} \end{cases} \quad (10)$$

and r_j follows the uniform distribution $U(0, 1)$

Mutation Mutation introduces variation that are not found in parents. This is usually done by adding a variation term to the new genes along side the crossover process. There are various methods to implement the variation term. Here a scaled normal distribution is added to selected terms in selected individuals. The selection process is random among the offspring with a probability that is set with a hyper-parameter. One can switch from exploration to exploitation by gradually decreasing the mutation probability over generations.

2.4 Evaluation Metric

Unlike typical supervised learning algorithms, our reinforcement learning algorithms can be evaluated by straight-forward metrics. Since the learning task of the algorithms is to play a game, a natural evaluation metric is the score achieved in the game. The higher the score is, the better the algorithm performs. To be more specific, we define the scores as follows.

- In the Snake game, the score increases by one when the snake eats an apple.

- In the Pong game, the score increases by one when the paddle hits the ball.
- In the Flappy Bird game, the score increases by one when the bird passes one tube.

An alternative evaluation metric is the cumulative reward (for Deep Q-Learning) and the fitness function (for genetic algorithm). In the Deep Q-Learning algorithm, our algorithm assigns one reward value to each action taken by the agent. The cumulative reward of the agent in a single game therefore reflects its performance in that game. The same is true for the fitness function. Both game score and cumulative reward (fitness) are proper evaluation metrics. They are in fact approximately equivalent. The intuition is that our carefully chosen reward function (fitness function) well-reflects the performance of an action and is closely related to the final score. In other words, the reward function guides the agent to perform better. In later experiment, We will see that the cumulative rewards are linearly proportional to the game scores.

2.5 Model Selection and Hyper-parameter Optimization

Deep Q-Learning In Deep Q-Learning, we tune a number of hyper-parameters to achieve decent performance. Our implementation customizes a neural network using the Keras library, so we need to set the number of layers, the size of each layer, the activation functions, the loss function, and the optimizer. Along with these setups, we find optimal batch size and learning rate to improve training speed and score.

For model selection, among a number of models from different epochs or generations, we aim to find the best performing model. Our approach is to save the model weights for every epoch during training and run tests on the weights. Since performance of the agent can mostly be evaluated by the score it achieved, we select the model weights of the epoch that reports the highest mean score during testing.

Genetic Algorithm In genetic algorithm, in addition to tuning the hyper-parameters of the neural network similar to that of Deep Q-Learning, parameters such as the mutation rate, mutation type, parent size, and offspring size. The mutation rate controls the probability of mutation happening within an individual. The mutation type determines whether the mutation rate decreases as generation increases. The parent and offspring size specifies the size of parents and children for the next generation.

In selecting the model, because of the randomness associated with genetic algorithm, the best model is not guaranteed to be from the later generations, nor is it guaranteed from a single run of the algorithm. As can be seen from 3, after running genetic algorithm on the Snake game a hundred times, each with a cutoff at 100 generations, only about 20% of the runs generated a model that could achieved a score of 20+, and a select few achieved a score of 40+. For the majority of runs, our model is stuck at a local minimum and does not lead to convergence. Therefore, we keep tract of the fittest individuals from each generation and different runs of the game in order to select the one best individual from all. We then record the neural network of the selected individual as our model.

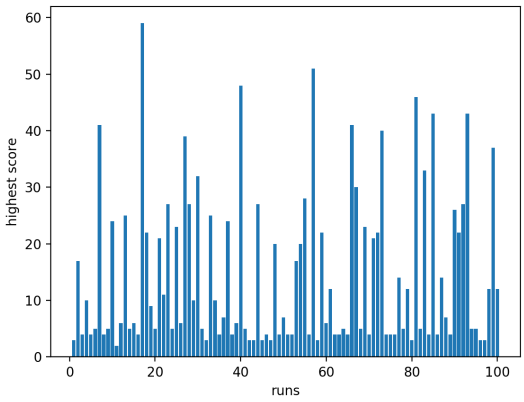

Figure 3: Bar chart of best scores in 100 generations among 100 runs of genetic algorithm in Snake

2.6 Experiment Setup

We have done several experiments in order to compare the performance of DQN algorithm and genetic algorithm. The experiment setup and the hyper-parameters are described in the following tables.

Attribute	Description
DQN model	3 Dense layers of size 128 (Figure 2)
Batch size	Snake and Bird: 512, Pong: 8
# of epochs	Snake and Bird: 100, Pong: 50
# of tests	100 tests for each weight

Table 1: Experiment setup of DQN model

Attribute	Description
NN model	3 Hidden layers of size 12,20,12
Population size	Snake, Bird, Pong: 150
# of generations	Snake, Bird, Pong: 100
Selection Method	Roulette Wheel
Mutation Rate	0.05
Mutation Type	Static

Table 2: Experiment setup of genetic algorithm

Additional Notes We control other variables of the games in order to make a fair comparison. For Snake game, we fix the width and height to 20 grids, and each grid is a 20×20 pixel square. For Pong game, we fix the screen size to 600×400 pixels, the paddle length to 100 pixels, and the ball speed to 15 pixels per second. For Flappy Bird game, we fix the bird speed to 10.

In some experiments, our trained models are able to play the game without dying. Therefore, we set a maximum score threshold so that the game stops when the score reaches the threshold. This prevents the testing trial from running forever. We set the value of this threshold to 13 for Pong and 250 for Flappy birds. For Snake, we set a maximum threshold of number of *moves* instead of the game score. The reason is that the board size of Snake is finite, so the game always ends unless the snake is looping in a cycle. In the case when the snake loops, the score won't increase, so a threshold on number of moves is more reasonable for Snake game.

Moreover, in the context of reinforcement learning, there are no test sets. In each test trial, the agent performs a task in a randomized environment, and its score completely reflects its performance. Therefore, we do not need to perform cross validation in our experiments, and we can directly compare the average test performances of each configuration.

3 Results and Discussion

In this section, we present and compare the performance of the Deep Q-Learning algorithm and the genetic algorithm for each of the 3 games, Snake, Pong, and Flappy Bird. As mentioned in 2.4, there are two equivalent evaluation metrics, the game score and the cumulative reward (fitness function). In the following presentation, we choose to use game score as the final evaluation metric for the following two reasons.

First, these two evaluation metrics are equivalent. Figure 4 shows a scatter plot of cumulative reward v.s. game score tested on 10^4 Snake games (100 epochs and 100 tests for each epoch). Most points fall in a linear region, indicating that the cumulative reward metric is approximately equivalent to the game score metric. We can also notice that some points have relatively high cumulative rewards but low game scores. This is due to the fact that the reward function is not always a perfect estimator of the game score. In some rare cases, the agent learns a weird pattern, which has high rewards but low game score.

Second, game score is a more general metric in order to compare both DQN algorithm and genetic algorithm. Cumulative reward and fitness function are defined differently. Although both reflect the performance of the agent, they are not on the same scale.

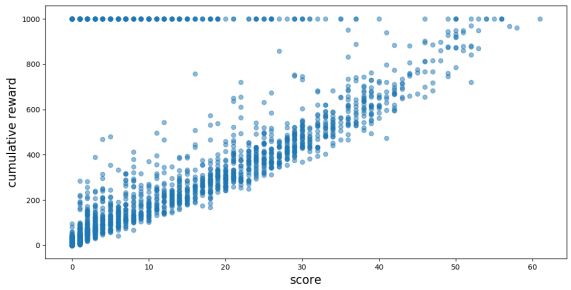

Figure 4: Scatter plot of reward v.s. score tested for Snake game

3.1 Snake

The plots of Snake in Figure 5 (row 1) show that the performances of two algorithms are close. Our DQN model (row 1, left) reaches the highest mean score, which is about 32, in epoch 45, and our genetic algorithm (row 1, right) reaches the highest mean score 22 in generation 90. Although DQN slight outperforms genetic algorithm, genetic algorithm has lower time complexity. It took us about 6 hours to train 100 epochs of our DQN model, while genetic algorithm took less then one hour (around 20 minutes in most cases).

The Deep Q-Learning algorithm achieves the higher score in early epochs, but its performance is not stable throughout different epochs. As the figure shows, The performance begins to fluctuate dramatically from epoch 80 to epoch 100. The main reason is that as the agent is trained to behave better, it less frequently receives the punishment. As a result, it falsely learns that eating an apple has higher priority then not dying. This makes the agent easier to die, and thus it receives more punishment and learns that not dying has higher priority. This process might repeatedly happen, causing the performance in later epochs to fluctuate.

The genetic algorithm is relatively more stable compared to DQN. It shows a slow trend of learning. As generation number increases, the performance gradually improves and converges to a stable level after 90 generations.

3.2 Pong

The plots of Pong in Figure 5 (row 2) show that the Deep Q-Learning algorithm (row 2, left) wins the competition of learning the Pong game (row 2, right). We set the maximum score, which is the highest possible score that the agent can achieve, to be 13. After about 15 epochs, the Deep Q-Learning algorithm achieves the maximum score almost every time.

In comparison, the genetic algorithm (row 2, right) seems to have fallen into a local minimum and have not

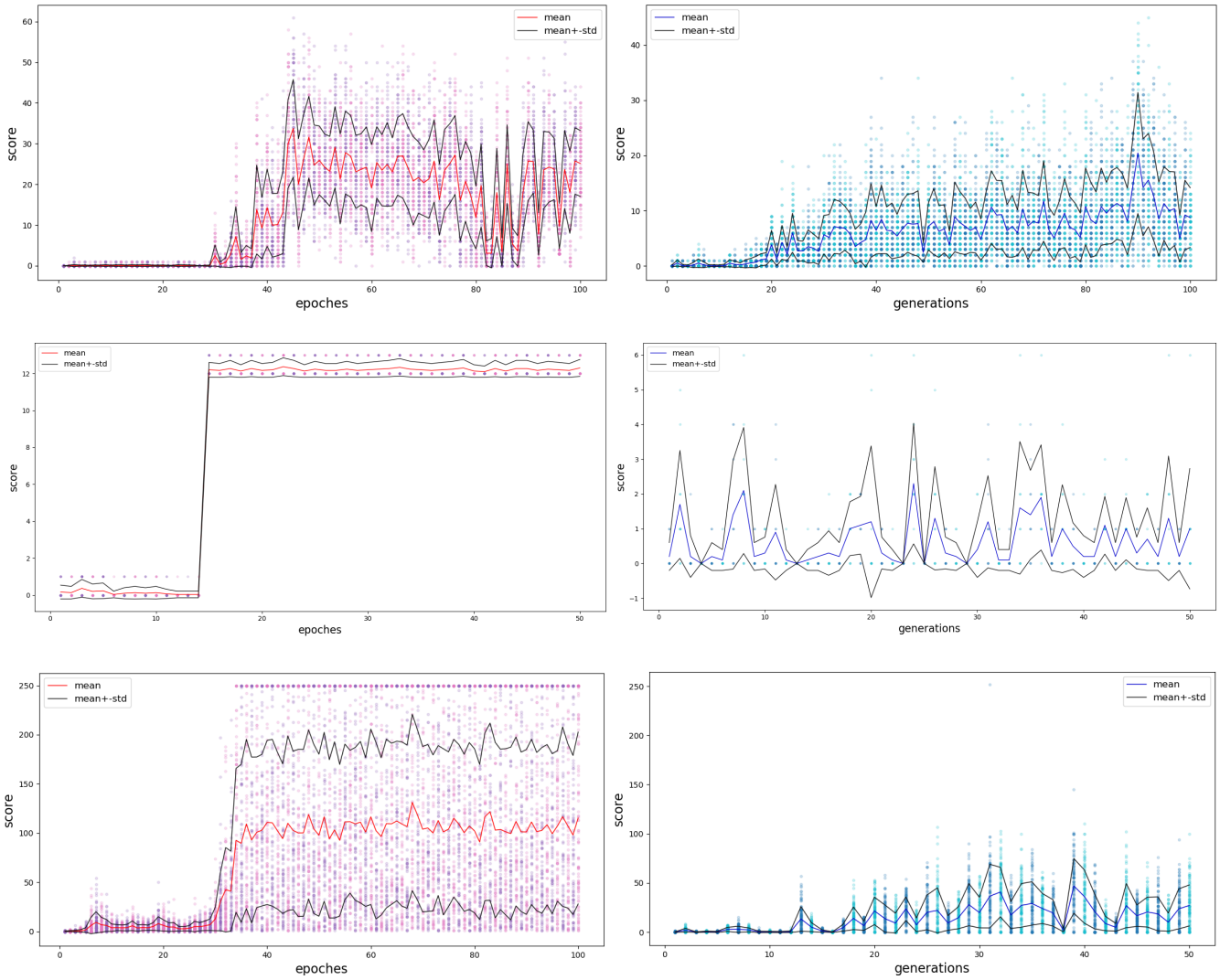

Figure 5: Learning curves of deep Q-learning (left) and genetic algorithm (right) for Snake (row 1), Pong (row 2), and Flappy Birds (row 3). Each dot represents a test score, and there are 100 tests in each epoch / generation.

learned the game by generation 50. Its performance fluctuates dramatically. The highest score that the genetic algorithm achieves in the first 50 generations is 6.

3.3 Flappy Bird

The plots of Flappy Bird in Figure 5 (row 3) show that the Deep Q-Learning algorithm (row 3, left) performs better than the genetic algorithm (row 3, right). We set the maximum score to be 250. The Deep Q-Learning algorithm reaches the maximum score consistently after about 30 epochs. DQN went from average score of around 10 to around 100 within 8 epochs, which demonstrate impressive exploration ability.

The genetic algorithm (row 3, right) successfully reached a score of around 100 after 30 generations but no further advancement was observed. Unlike the DQN which had a steep improvement at around 30 epochs, the genetic algorithm had a long and gradual increase of score as generation count increases. The variation also seemed to be rather large in the genetic algorithm, indicating a potential problematic emphasis on exploration rather than exploitation.

4 Conclusions

In this project, we design reinforcement learning algorithms that learn to play simple games, including Pong, Snake, and Flappy Bird, from two different approaches. The Deep Q-Learning algorithm achieves good performance on all 3 games and the genetic algorithm achieves good performance on Snake and Flappy Bird.

Future Works The reward function is one of the essential factors that determines the performance of a deep Q-learning algorithm. One of the main focuses of our project was to define a suitable reward function. We improved the naive reward functions by adding more events to it. An interesting question is whether we can design a more effective reward function that learns the game rules more comprehensively? For example, we observed that the trained DQN model for Snake game sometimes enclosed itself. In this case, the snake would always die in a few moves despite its actions. This event occurred frequently when the snake body was long, and this is the main factor that prevents our model from performing better.

References

[1] Leslie Kaelbling, Michael Littman, and Andrew Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 04 1996.

[2] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[3] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[4] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *CoRR*, abs/1611.03530, 2016.