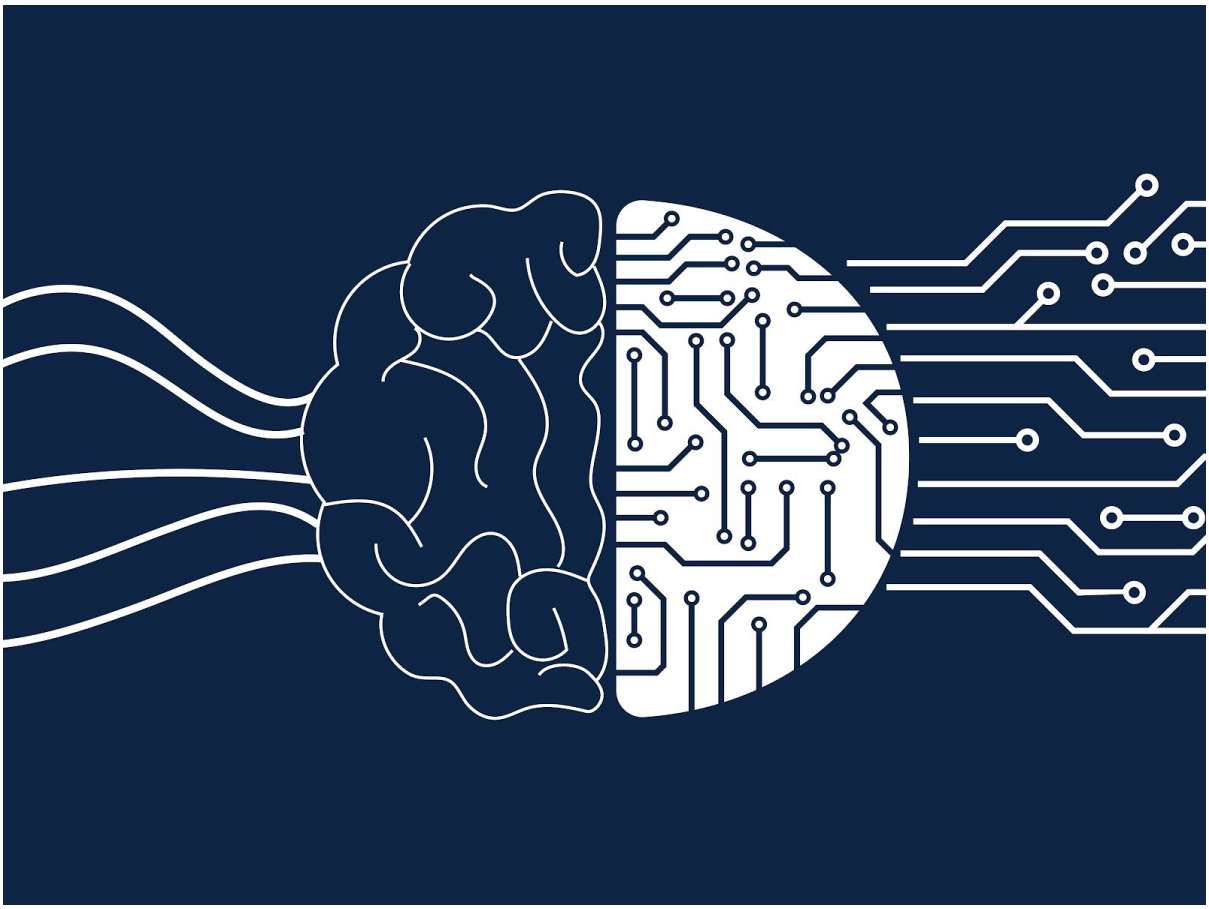
NUSISS - ISY5005 - SELF-LEARNING SYSTEMS PROJECT

REINFORCEMENT/EVOLUTIONARY LEARNING SYSTEMS FOR MSPACMAN

A CASE STUDY OF TWO APPROACHES TO AN OPENAI ATARI GAME



Name	Student ID	Email
LIM LI WEI	A0087855L	E0319479@u.nus.edu
YONG QUAN ZHI, TOMMY	A0195353Y	E0384984@u.nus.edu
PREM S/O PIRAPALA CHANDRAN	A0195324A	E0384955@u.nus.edu

1 Introduction (Project Overview)

The application for Artificial Intelligence (AI) game environments serves as an infinite supply of useful data for machine learning algorithms making them a desired domain for AI research. AI in these game environments has been helping us understand how games are played in terms of the strategies devised, the efficacy of the different algorithms used in play and also how game design may be enhanced.

These aspects pose three fundamental challenges with its own proposed approaches to solving them. The first challenge is that the state space in strategic games is very large , but this has been modelled with the neural networks using approaches such as deep Q learning. Secondly, the learning of proper policies for decision making in dynamic unknown environments is difficult, but the use of data driven methodology such as reinforcement learning (RL) provides feasible solutions. Lastly , as most AI is developed for a specified virtual environment, this poses the question of how best to transfer AI's capability generic enough to make it adaptable to other systems.[1]. In addition, these game environments provide infinite supply of useful data for machine learning algorithms, and they are much faster than real-time. These characteristics make games the unique and favorite domain for AI research. On the other side, AI has been helping games to become better in the way we play, understand and design them [1].

This project is an attempt to apply Reinforcement and Evolutionary Learning techniques to the gameplay environment of MsPacman. It would be prudent to discuss the nature of the game environment before deciding and explaining on the choice of the appropriate algorithms of the learning models to apply

2 Problem statement

Our motivation behind making the choice of game to test our learning agents for the different self-learning system methods was based on our history of games we have played, the challenge of the game environment and the time for yield of results to analyse. Ideally it would be a game we were familiar with which would pose enough of a challenge for the learning agents to play in terms of the rewards structure yet be able to produce results within a reasonable time frame for comparison of performance.

Based on the afore-mentioned criteria, we surveyed online games and discussed them before deciding on Pacman. The game environment tested is linked here (click https://gym.openai.com/envs/MsPacman-v0/) [2]. As observed from the game environment, it is underlying real-world applications and analogous to common strategy scenarios such as robot navigation in a maze environment and decision making with conflicting rewards (eating immediate pellets versus aiming for isolated fruits which leaves the agent vulnerable)

For this project, the "problem" we are going to solve, is **how to win at "Ms-Pacman**"
Winning is defined to be getting the highest score possible based on points.

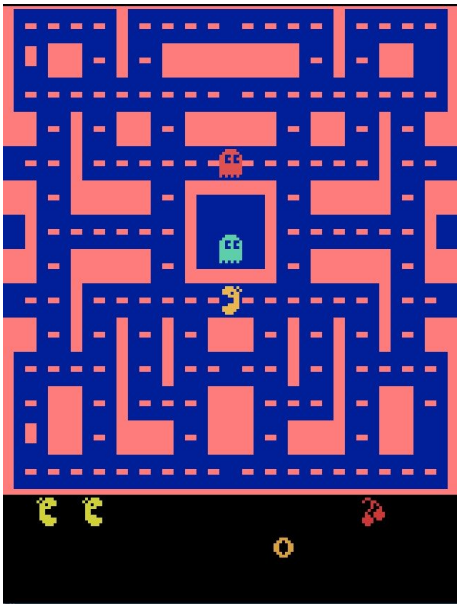


Fig. 2.1 Screenshot of Game Environment of MS-Pacman

Referring to the game environment as shown in the Fig. 2.1 above

The ground rules of the game are as follows (Human Point of View):

1. Pac man can take any of **4 directions** (up,down,left,right)
2. Every small pellet it eats is **10 points**
3. Every big pellet it eats (to convert ghost to food) is **50 points**
4. Every ghost it eats after eating the big pallet is **50 points**
5. Every cherry it east is **100 points**
6. Objective is to **clear all pellets** in the stage

However, when it comes to reinforcement learning, the machine has a different point of view. This is due to the resources that is available to the agent via OpenAI gym, which are:

1. INPUT: Actions, which are the 9 standard actions for atari games in gym which consist of **No Operation, Up, Right, Left, Down, Up-Right, Up-Left, Down-Right, Down-Left**
2. OUTPUT: Reward, which is the **direct output** from the **score engine** within the game
3. OBJECTIVE: Since the reward engine in OpenAI gym does not penalize deaths and game overs, the objective is to **score as high as possible** before all lives are exhausted, resulting in a game over.

2.1 The challenge

The challenge lies in learning a policy efficiently. A policy is defined in this context to be a set of strategies to earn the highest score. In the DQN paper[3], efficient learning is (somewhat arbitrarily) defined as at most 800 million game interactions. This might seem like a large number (it corresponds with close to 4000 hours of real-time game-play), compared to the number of possible game states that can be encountered—up to 10^77 for Ms. Pac-Man—it is an extremely tiny amount. This necessarily means that strong generalization is required to learn an effective policy.

As an off-policy model-based reinforcement learning, and dynamic programming similar to Bellman's equation, DQN does not take any external inputs and learns gameplay entirely based-on the environment itself. As such, it may not be suitable for a number of games where rewards are ultimately delayed and requires multiple objectives to achieve, for example, in the game 'Montezuma's Revenge'.

$$V(s) = max_a(R(s,a) + \gamma V(s'))$$

Fig. 2.1.1 : Fundamental Bellman Equation

Q-learning involves doing value iteration to update Q-values in a Q-table, which will reflect what action the agent will take. Hence, training will take very long as the agent is processing different past experiences based on many different scenarios, based on a sizable amount of pixels which can have multitude combinations. It is also not easy to find a global optimum to converge to, especially when it is exploiting too much and not exploring enough. One problem with this is that Pacman will get stuck in a wall, unable to move on.

Compared to other games, Pacman is one of the more suitable candidates for Q-learning due to the immediate reward (points) it will receive for quite a number of actions, such as consuming pallets, taking fruits, defeating ghosts etc, which is why we selected it as the game of choice.

There have been prior implementations of RL [3] and EL [4] learning techniques to play Pacman. Our project is aimed to combine these techniques, experiment with different algorithms, hyperparameters and network architectures to arrive at a state of consistency so as to compare their outcome performance and arrive at some conclusions of our own to build on our understanding of RL and EL.

In the face of unlimited resources, namely computation time, memory and game interaction, policy learning becomes easy with the storing of exact values for each individual frame and using RL (e.g Q-Learning). The challenge for RL and EL techniques for playing Pacman lies in the learning of a policy efficiently [5]

3 Reinforcement Learning / Evolutionary Learning Systems Methods

In this section, the two different approaches to the applied to playing MSPacman are discussed. A brief introduction is firstly given about each approach and its strengths and limitations and then a detailed explanation is given to its implementation for playing the game.

3.1 Reinforcement Learning

Reinforcement Learning, being a subfield of machine learning beside supervised and unsupervised learning, is concerned with the learning process of an arbitrary being, formally known as an *Agent*, in the world surrounding it, known as the *Environment*. The Agent seeks to maximize the *rewards* it receives from the Environment, after performing different *actions* in order to learn how the Environment responds and gain more rewards [6].One of the greatest challenges of RL tasks is to associate actions with postponed rewards — which are rewards received by the Agent long after the reward-generating action was made. [7] A generic representation of a RL scenario is shown in Fig 3.2 below [8].

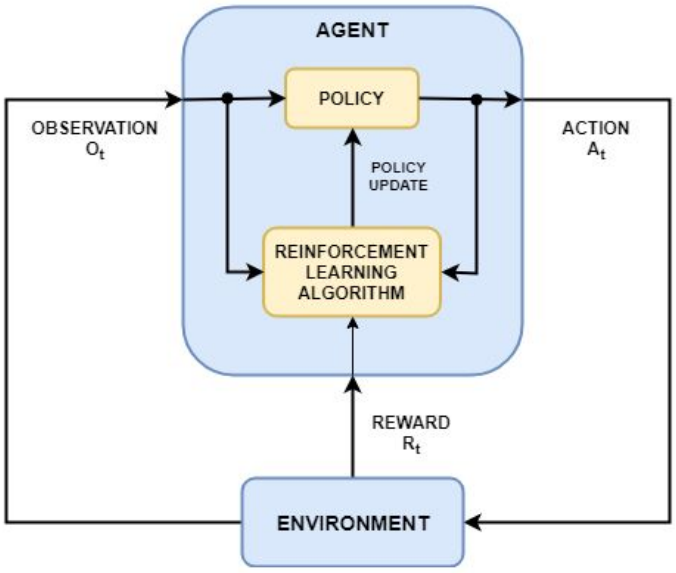


Fig 3.1.1 : General representation of a reinforcement learning scenario.
https://www.mathworks.com/help/reinforcement-learning/ug/what-is-reinforcement-learning.html

The key distinguishing factor of reinforcement learning is how the agent is trained. Instead of inspecting the data provided, the model learns from the environment by interacting with it seeking ways to maximize the reward. In the case of deep reinforcement learning, a neural network is in charge of storing the experiences in terms of tuning the weights assigned to the neurons and thus improves the way the task is performed.

In this project, the method of reinforcement learning we are using is the **Deep Q Network (DQN)**, a model-based, off-policy method employing Methods similar to Q-Learning

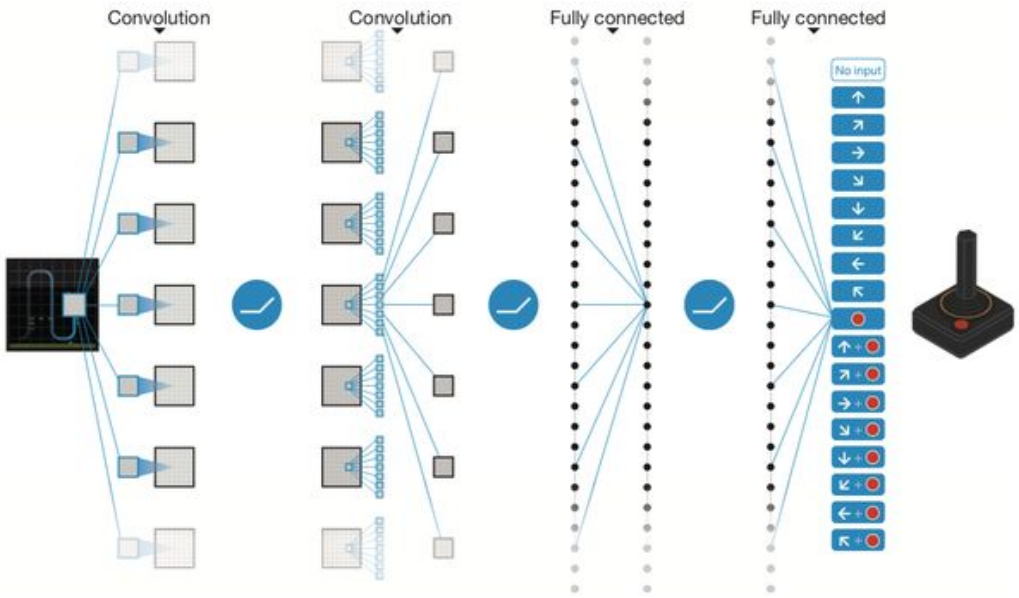


Fig 3.1.2 : DQN Architecture [9]

In Q-learning, the game is framed mathematically in **Markov Decision Process (MDP)** style, where every immediate State **S** is observed in the environment after initialization or taking a step, The Agent is then able to perform a fixed set of actions defined by the rules of the environment, to obtain a certain Reward **R.** By using value iteration with the Q function **Q(s,a)** to allow the agent to move to the next state using the best resulting **Q-Value**, and with the *epsilon-greedy* policy to balance between exploration and exploitation, the ultimate objective is to optimize long term reward.

Q Learning is able to accomodate for the complexity in modeling scenarios where the action space is discrete and small such as the Pacman game environment. However it cannot handle large and/or continuous action space. Its flexibility is that its policy is deterministically computed from the Q function by maximizing the reward,hence it cannot learn stochastic processes.

In this game, our states are defined by the pixels at any frame instant. In an attempt to reduce dimensionality, the pixel data are then cropped, gray-scaled, and resized to (88,80) to be prepared for processing by our deep learning network.

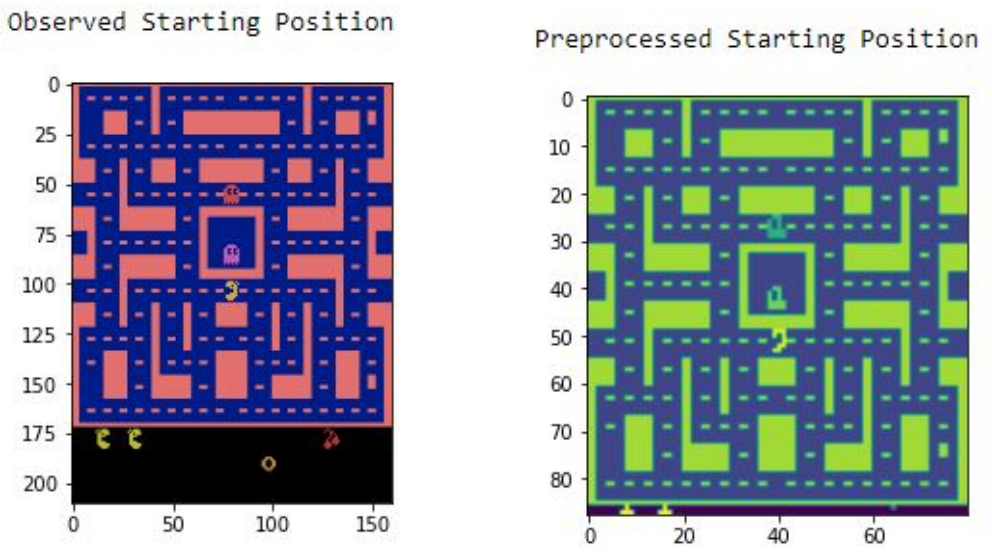


Figure 3.1.3: Initial Position of PacMan

Before feeding into the CNN, we have to prepare the datasets fed from the game itself. Our datasets consist of windowed data of 4 sets of frames back to back from the game. The CNN will then perform feature extraction for the states and generate the Q values to decide and output an action step into the environment.

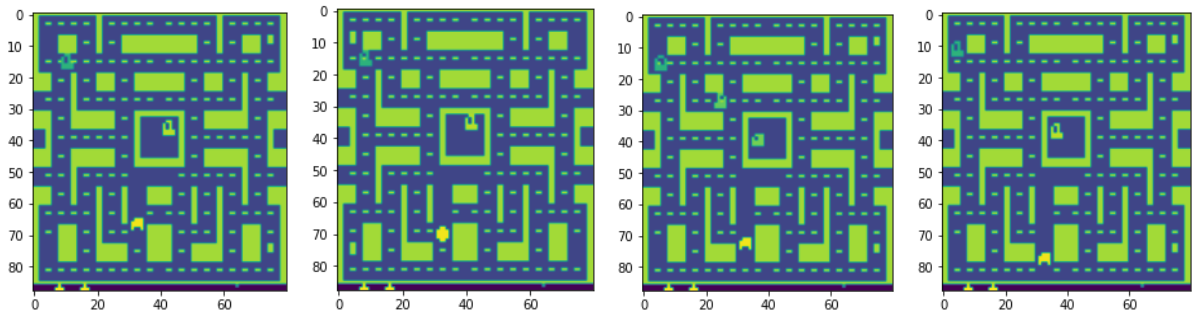


Figure 3.1.4: A Single Datapoint input to the CNN (4 Frames)

We will be using a similar model as proposed by Mnih et al. (2015) in his paper [10] as shown below:

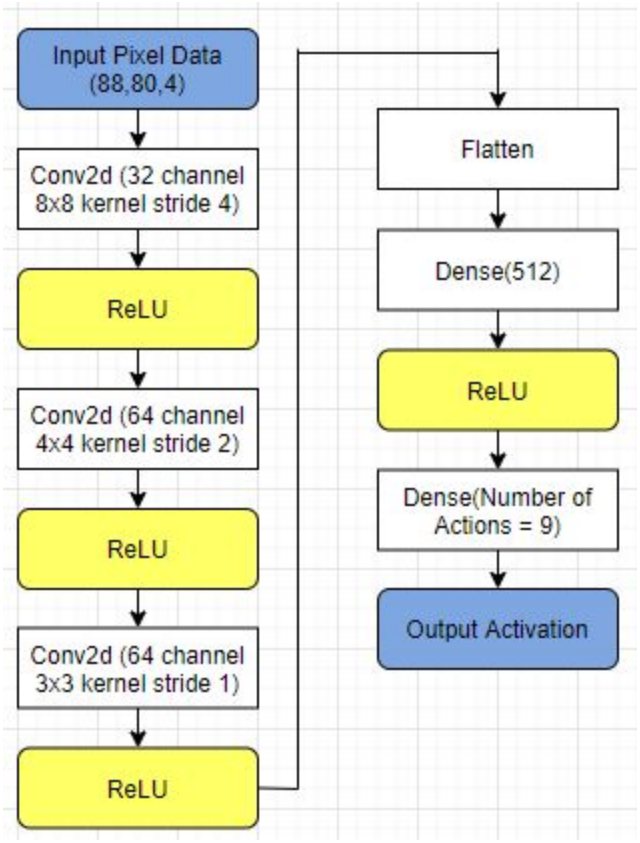


Figure 3.1.5 : CNN Model as Proposed by Mnih et al. (2015).

This same model is popular with DQN reinforcement learning in the community and it functions well to output the actions given the input set of 4 frames of data input. Instead of doing the Bellman's equation manually to update the q-values, we use a deep neural network to approximate the q-values for the next action taken. The output will be one of the 9 actions as prescribed in the gym environment.

3.2 Evolutionary Learning

Evolutionary algorithms are an approach to optimization problems where assumptions about value functions and the unique characteristics of elements within an environment are eliminated and the only methods used are those that utilize the fitness function [B2]. This concept is taken from biological evolution where only the fittest survive [B1]. In recent years, there have been research works and code competitions which have shown the ability of evolutionary techniques to train computer agents to play video games that rival the competencies of some human players [B4]. The following section will outline one of such evolutionary techniques that will be used to play the game 'MsPacman'.

3.2.1 Genetic algorithm

Genetic algorithms(GA) were inspired by biology much like neural networks were and follows the Darwinian theory of evolution in its design as shown in figure 3.1. GA first creates a certain number of participants also known as "genes", these groups of genes are established as a "population". It will set a task for these genes and will be evaluated against a "fitness function". This function effectively determines how well a gene has completed a specific task. If a hard constraint has been fulfilled, GA ends the cycle. If not, it will proceed to select the best performing genes also known as the "fittest survivors" and move them to the next phase. It is important to note that each phase is considered to be a "generation". As only the fittest few are selected, it can be anywhere from 5 - 20% of the initial population, it will need to be repopulated. There are different ways to execute repopulation, one way is to copy the genes of the fittest parents and conduct a "crossover" to make a new gene. Occasionally, a gene can be "mutated" to create some difference from other genes as a way to explore other possible make up of genes that may create stronger genes. This cycle will continue until a certain hard constraint has been fulfilled.

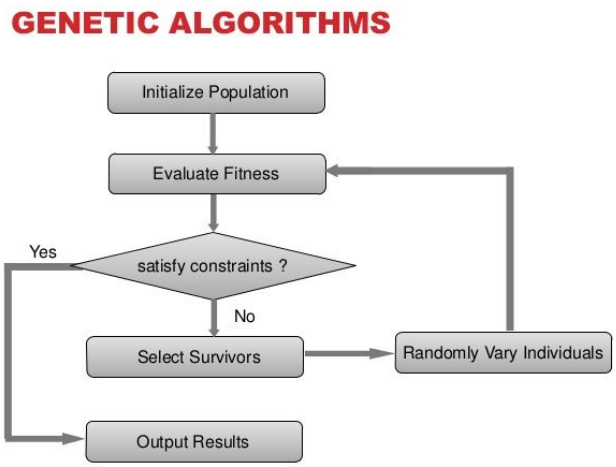


Figure 3.2.1: GA Workflow [B6]

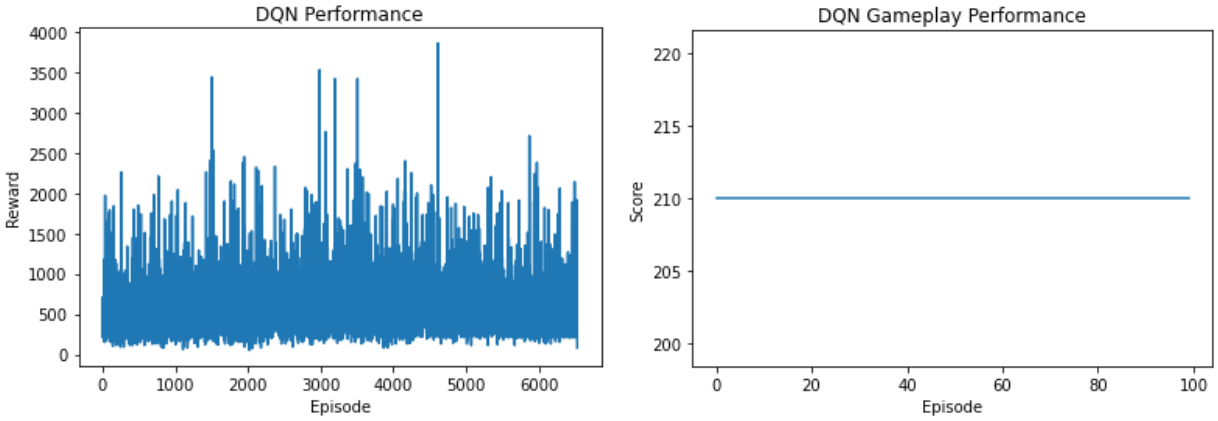
For our implementation, we have adopted the same logic as outlined above. For our initial population, we will create a number of policy nets, which are deep neural networks, and the fitness function will be based on rewards earned. For each re-population phase, a random parent will be selected and duplicated with a noise applied to it's weights. In section 4.2 we will look at the results obtained with our GA implementation.

4 Results and Findings

4.1 Reinforcement Learning

In this project, we are doing DQN with a Greedy-Epsilon approach, using the linear annealed policy by decaying Episilon over X steps. As such, the primary hyperparameters are the learning rate alpha, discount factor gamma, and epsilon (Min,Max,Decay Length). To obtain a good AI, hyperparameter tuning is needed, especially epsilon, which defines how much exploration/exploitation ratio the agent training will take.

Steps	Learning Rate (Alpha)	Discount Factor (Gamma)	Epsilon (Min)	Epsilon (Max)	Epsilon Decay Length	Weight Update Batch
5,000,000	0.001	0.99	0.1	0.65	4,500,000	25,000

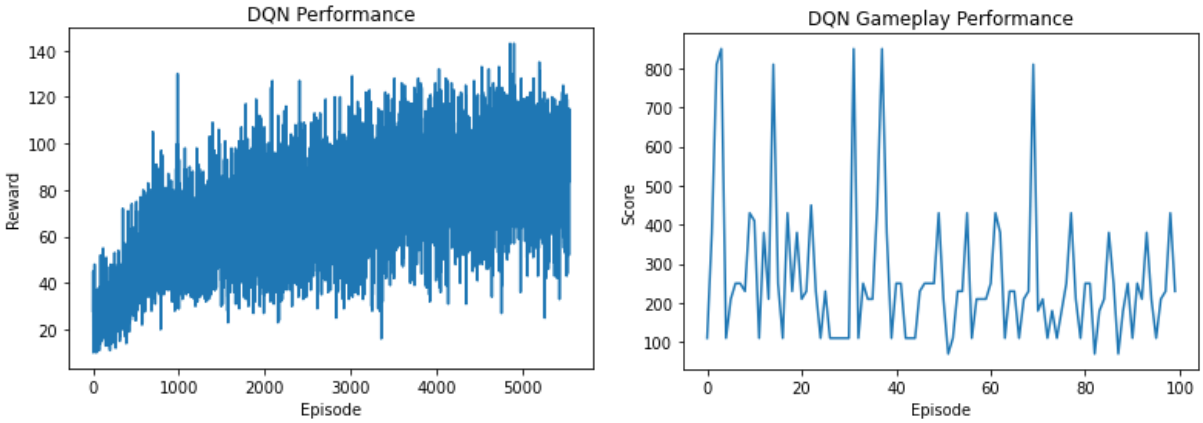


Average Gameplay Score: 210.0

Figure 4.1.1: First Training Batch with Training Plot (Left) and Gameplay Plot (Right)

In this setup, we did not clip the values of the rewards, and somehow it is not converging well. Referring to section 2 of this report, we see that you can get varying amounts of reward every time a successful scoring action is taken. This may cause the learning rate to be unstabilized as taking a huge reward in a short amount of time alters how the agent learns and may converge into a sub-optimal result, as shown in this run.

Steps	Learning Rate (Alpha)	Discount Factor (Gamma)	Epsilon (Min)	Epsilon (Max)	Epsilon Decay	Weight Update Batch
5,000,000	0.00025	0.99	0.1	1.0	500,000	20,000

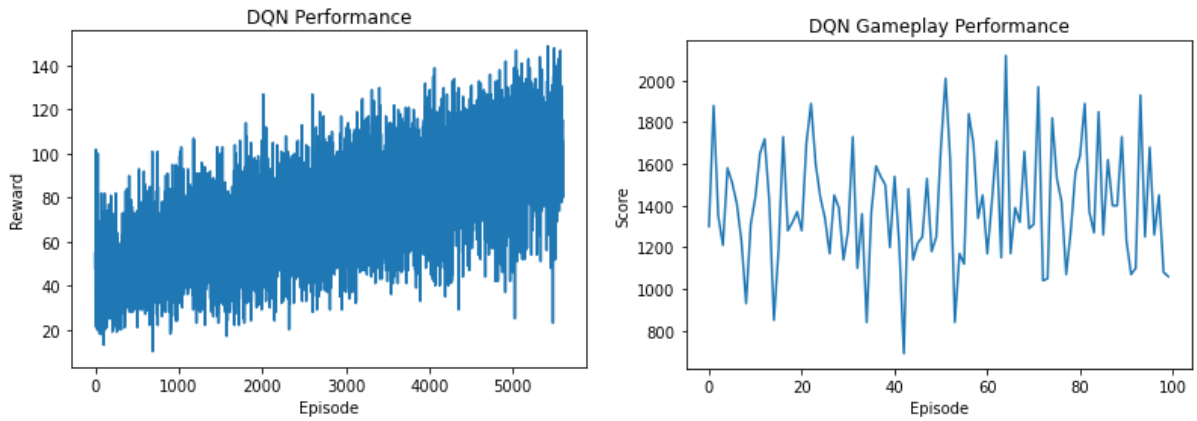


Average Gameplay Score: 260.6

Figure 4.1.2: Second Training Batch with Training Plot (Left) and Gameplay Plot (Right)

In this next setup, reward clipping is implemented: reward is scaled to -1 to 1 every time it is obtained. As a result, the magnitude given in the training plot is not the real score. From here we can see that the training is properly converging. However, we allowed epsilon to decay too fast, from 1.0 to 0.1 in 500k steps, making epsilon during training 0.1 for the next 4,500,0000 steps. This results in not enough exploration, again, causing a sub-optimal solution. After viewing a sample run, we discovered that the agent has found a strategy to eat the nearest big pellet to convert the ghost to food, then wait for the ghost to come find it. Hence it keeps getting stuck to the corner and not progressing.

Steps	Learning Rate (Alpha)	Discount Factor (Gamma)	Epsilon (Min)	Epsilon (Max)	Epsilon Decay	Weight Update Batch
5,000,000	0.0002	0.99	0.1	0.5	4,500,000	25,000

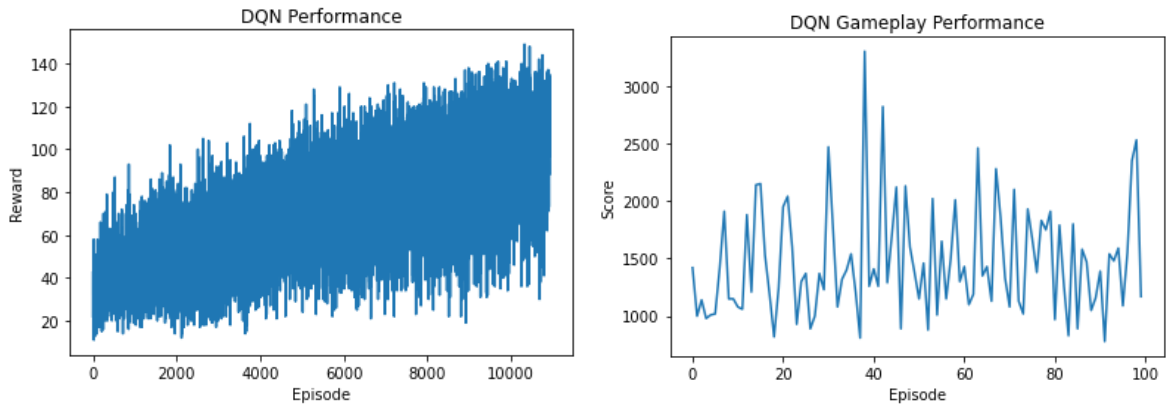


Average Gameplay Score: 1398.5

Figure 4.1.3: Third Training Batch with Training Plot (Left) and Gameplay Plot (Right)

In this run, we improve the exploration step by decaying it over a longer range, from step 1 to step 4,500,000. This allows ample exploration to jump out of any bad convergence, if any. As we can see from the training plot to the left, there is a gradual steady increase in gameplay performance as time passes. After viewing a sample run from this trained model, we can see it performs a lot better than the previous model, now being able to develop an elementary strategy to eat the pallet in a particular pattern from the start point; and this path is also less likely to encounter ghosts along the way. However as gameplay goes over time when the ghosts have a different combination of appearance, the agent still appears unable to handle them and dies to the ghosts.

Steps	Learning Rate (Alpha)	Discount Factor (Gamma)	Epsilon (Min)	Epsilon (Max)	Epsilon Decay	Weight Update Batch
10,000,000	0.0002	0.99	0.1	0.65	9,500,000	25,000



Average Gameplay Score: 1467.0

Figure 4.1.4: Fourth and Last Training Batch with Training Plot (Left) and Gameplay Plot (Right)

In this final run, we made a final tweak to increase its exploration by a bit, starting from 0.65 instead of 0.5, and decaying to 0.1 over 9,500,000 steps, in a 10,000,000 step run. However, the performance is still about the same as the previous run where it was trained for 5,000,000 steps. Perhaps this is as good as the agent can get using just simple DQN for training. However, we do see an improvement in the average gameplay score and the max score (Over 3600!).

4.1.1 Final thoughts and future improvements

DQN is definitely suitable for this game. However, by training in a gym environment without modifications, there seems to be a few disadvantages. Firstly, this game only needs 4 actions, however we are forced to output 9 actions. These may cause many redundancy as those actions might be mapped to one of the undesired 4 actions. The reward engine is also fixed, we cannot alter it. As such, there is no penalty when pac man dies. The agent will keep outputting actions until all 3 lives are exhausted. This will cause the agent to think that it is ok to die to the ghosts as long as it continues to consume pellets.

Secondly, as DQN is a off-policy method of training, it is very difficult to come up with a optimal solution for this game just by using DQN alone. To transit from one frame to another, perhaps might be possible in the first few frames. However as gameplay gets longer, making the same move will not allow transition to the exact same state. As such, the agent will need to experience all kinds of permutations of gameplay experiences to be able

to perform perfectly. The generated Q-value will still help to make the best decision, however.

Having read up multiple papers, this paper [a] suggested a better architecture where a duelling DQN is used to improve the results. Perhaps the next step is to explore another algorithm to improve on the agent gameplay. Also as seen in [Z], Microsoft has developed an agent that can beat the game. They do not just develop one simple DQN agent to solve it, however. Multiple agent have to be developed, in a divide and conquer strategy to beat the game, as Pac-man, although a simple classic, is actually quite complex to compute given its rules and the unpredictable nature of the ghosts.

4.2 Evolutionary Learning

The number of hyperparameters involved in our GA implementation is far lesser than the DQN method. It only involved population size, parents count and a noise value. Initially, we imposed a hard constraint that if a generation could earn a reward mean of 2000 points, the training would end, but this was increased to 10000 points in order to allow the agents a higher ceiling to improve itself.

The hyperparameters and first training and testing information are outlined below.

First training

Population size	Parents count	Noise value	No of generations
50	10	0.01	1000

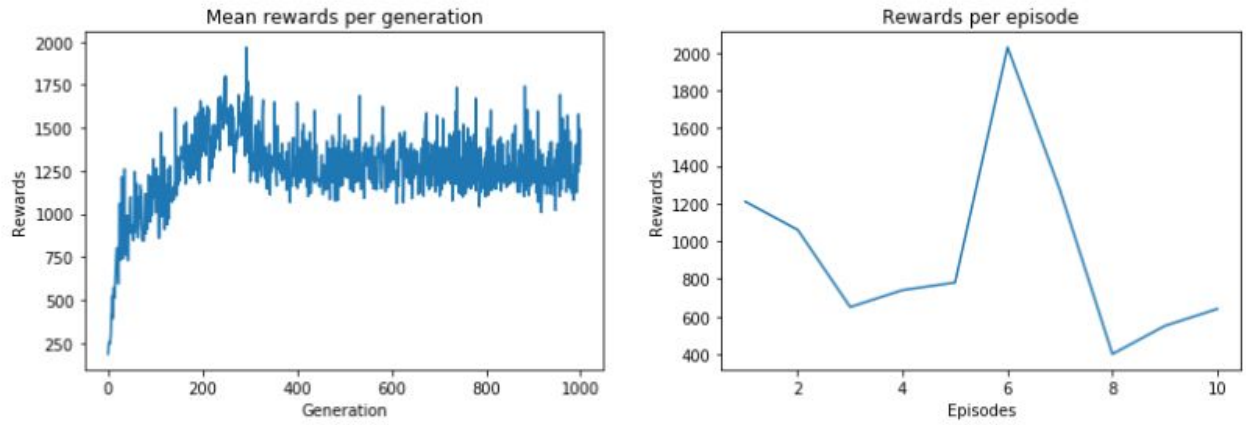


Figure 4.2.1: First training results over 1000 generations(Left). First Test results (Right)

The first attempt at training shows that each generation is able to achieve a steady improvement in mean rewards and being able to peak at close to 2000 points. However, after 400 generations it starts to fluctuate between the 1100 to 1600 range. However, the testing of the fittest agent does not show that the agent is able to achieve a decent consistency with rewards gained between 400 to 1900. The next round of training will seek to adjust the noise value.

Second Training

Population size	Parents count	Noise value	No of generations
50	10	0.001	300

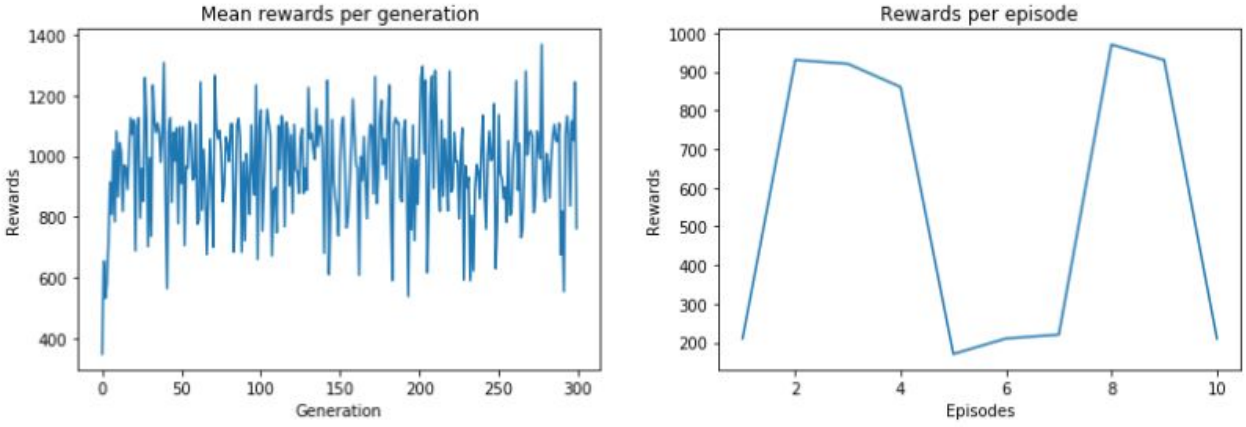


Figure 4.2.2: Second training results over 300 generations(Left). Second Test results (Right)

The noise value has been decreased to experiment if it has a positive or negative effect on the reward mean. Based on the left plot in figure 4.2.2, it shows that the reward mean fluctuates wildly between 600 - 1300 points after about 10 generations. It was obvious to see that the reduction of noise has a negative effect on GA training, this is further illustrated in the right plot of figure 4.2.2 when testing of the fittest agent is done. The score never breaches the 1000 mark and achieves a range of only between 150 to 950.

Third Training

Population size	Parents count	Noise value	No generations
50	5	0.1	500

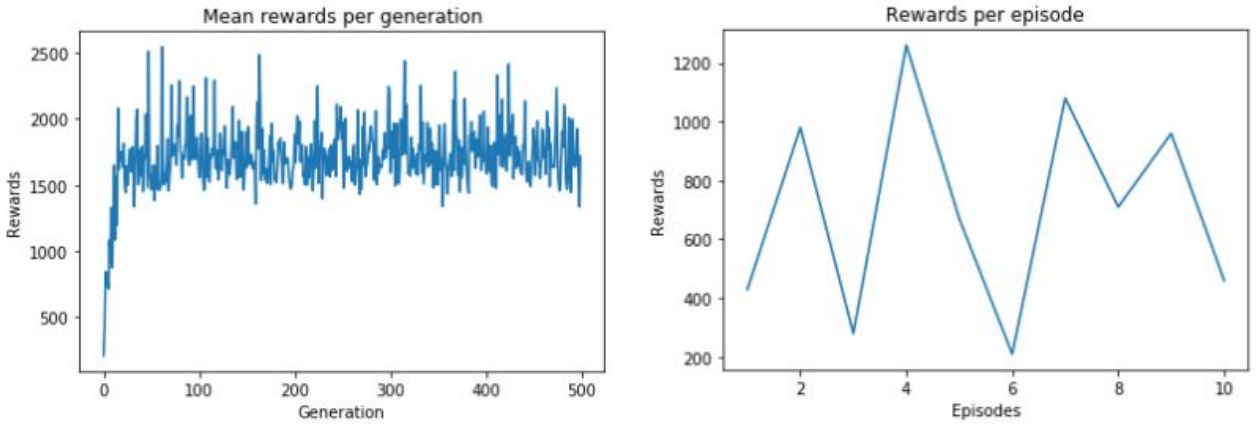


Figure 4.2.3: Third training results over 500 generations(Left). Third Test results (Right)

Upon increasing the noise value, training stopped at 60 generations. This was due to the hard constraint of 2000 mean rewards which will cease training. This was when we increased the cap to 10000 points and fixed the number of generations to 500 as it was observed that no much improvement in training occurs after 500 generations. The left plot of figure 4.2.3 shows that the mean reward is able to fluctuate at about 1400 to 2500 points which is a good improvement over the previous training iteration. However, the testing of the fittest agent from the right plot of figure 4.2.3 does not reflect this consistency as the agent achieves scores between 200 to 1200 which is not a satisfactory performance.

Fourth Training

Population size	Parents count	Noise value	No generations
50	5	0.5	500

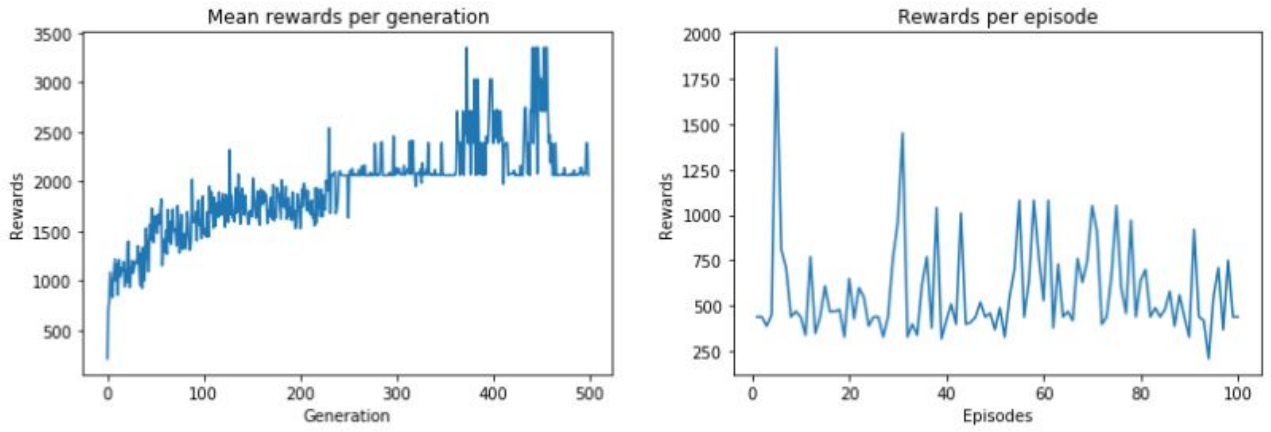


Figure 4.2.4: Fourth training results over 500 generations(Left). Fourth Test results (Right)

For the fourth training iteration, we increased the noise value and observed a drastic difference in training performance after 200 generations as seen in the left plot of figure 4.2.4. The reward mean appears to stay stable at 2000 points and sometimes even increases to 3400 points. It was expected that the agent would be able to get more consistent results, at least being about to obtain more than 1000 points for each play through. The testing as shown in the right plot of figure 4.2.4 shows that this again may not be the case. Testing was done over 100 episodes of gameplay to get a better idea of its long term performance. Even though there are occasions where it can get above 1000 points per playthrough, it's average performance seems to stay around 350 to 600.

It is at this point that we studied the training outputs in Spyder(Anaconda) closer. The first key observation we found was that the top 10 agents of the last generation were able to obtain a score of more than a 1000 and we only took the top 5 fittest. Therefore, we decided to increase the number of parents to select to try and include more variations of "fit" agents. This may help GA explore more possibilities of ideal candidates to tackle the MsPacman game. The second key observation was that after each generation the standard deviation was quite high, often around 700 - 900 points. This helped to explain that maybe the agents had not truly developed some consistent strategy and most were "lucky" to obtain a high

score. The next final training iteration aims to change the hyperparameters with the aim to increase the mean reward capability and reduce the standard deviation for each generation.

Fifth training

Population size	Parents count	Noise value	No generations
100	15	0.9	500

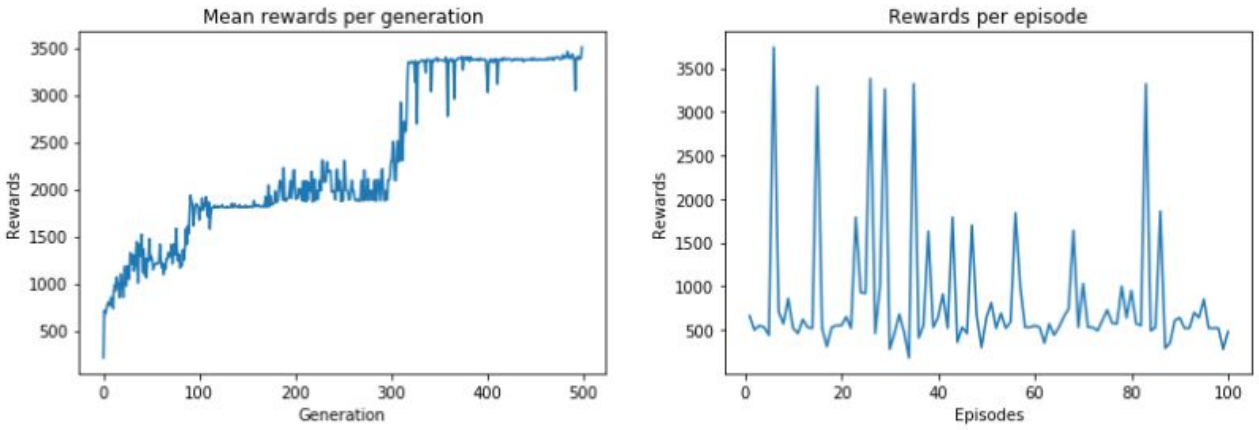


Figure 4.2.5: Final training results over 500 generations(Left). Final Test results (Right)

Final generation statistics:

Reward mean	**Reward STD**	**Reward max**
3505.3333333333335	414.06709064541167	4890.0

For the final training iteration, we have included some statistics on agent performances. With the increased population size, parent count and noise value we can see a far different training outcome than in all previous attempts. The training starts to perform consistently past 300 generations with a mean score of about 3300. At the same time, the standard deviation has fallen in value from 900 to about 414, though it is important to note the generation just before this had achieved about 180. It is thought that, at this point, the agent has found an optimal solution. The test results show that it was the most optimal solution that worked for it based on the fitness function but it was not what we were looking for in terms of human-like playing performances.

By looking at the right plot of figure 4.2.5, it shows the ability of the fittest agent to hit highs of 3600 score or close to that for a number of episodes but it is unable to maintain a decent score consistently with the score achieved ranging from 250 to 3600. As seen in the plot, most episodes end with less than 1000 points. However, this performance capacity is indeed an improvement over past agents. Interestingly, when reviewing the way the agent plays, it appears that it has found the strategy of eating a magic pellet and waiting in one corner for the ghosts to go to it. It is during the right circumstances that the score can shoot upwards to 3300+ points. This shows that the agent has found one unique exploitation strategy and is continually using it as the GA method is not able to improve on this without any exploration-exploitation strategy or backpropagation to adjust its policy net.

4.2.1 Final thoughts and future improvements

GA is a technique which finds the optimal solution for a given problem, however, for the case of MsPacman where there are countless scenarios that an agent can find itself in, 1 optimal solution may not be sufficient. At the same time, the fitness function was based on rewards earned but it did not motivate the agent to complete the game by eating all the pellets which was the main point. Moving forward, it may be better to change the fitness function to be based on pellets eaten rather than rewards earned. Also, it would be good to find a way to integrate an exploration-exploitation technique to encourage the agent to explore more instead of being stuck in a wall. Alternatively, the repopulation technique can be changed to either add or subtract the weights value by a range of possible values instead of just adding a constant to the randomly selected fittest parents.

5 Summary

MsPacman is a game with a great number of possible states and it is not precisely possible to find just 1 optimal solution to play the game. Different strategies will need to be adopted based on different states being presented. Also, it is not sufficient enough for MsPacman to earn a good score consistently in order for it to be considered a good agent. It will need to finish the game as well and stay alive as long as possible. This may require experimenting with different exploration-exploitation techniques such as proximal policy optimization.

Also, this game requires multiple thought processes to complete as there are many sub-classes of problems to solve, and getting the score as high as possible is just one of the many problems we need to solve. Perhaps multiple agents need to be deployed to solve different aspects of the game, in a divide and conquer strategy. However, from this project, DQN/EL seems to be good at just trying to get the score as high as possible, but may not be able to adapt to the game ghost AI, or pinpoint that the ghost can be eaten after taking the big pellet, for example.

6 User-Guide

6.1 Reinforcement Learning (Tensorflow/Keras)

6.1.1 Prerequisite software/libraries/dependencies

Libraries: Pillow, Numpy, gym, tensorflow/keras, keras-rl, matplotlib

6.1.2 Instructions

1. Set up conda environment, open your command prompt.
    a. Type in “conda create -n pacman-keras python=3.7”
    b. Enter ‘Y’ when prompted “Proceed Y/N?”
2. Activate environment
    a. Type in “conda activate pacman-keras”
    b. You should see your env name on the left side of your command prompt “(pacman-keras) C:/”
3. Install dependencies
    a. Type in “pip3 install pillow numpy gym tensorflow==1.14.0 keras==2.2.4 keras-applications keras-preprocessing keras-rl matplotlib jupyter”
4. Open the jupyter notebook for each configuration of run
    a. Run the first 6 code blocks to initialize the environment and hyperparameters
    b. Run the 7th code block to perform training. Training can take up to 24 Hours.
    c. Run the 9th code block to perform testing. Enable visuals if needed.
    d. You can skip the training, and load the default model trained by us by skipping the 7th code block

6.2 Evolutionary Learning (Pytorch)

6.2.1 Prerequisite software/libraries/dependencies

Python: 3.6 and above

Libraries: Pytorch, Numpy, Matplotlib, gym, random, os

6.2.2 Instructions

5. Set up conda environment, open your command prompt.
    a. Type in “conda create -n myenv python=3.6”
    b. Enter ‘Y’ when prompted “Proceed Y/N?”
6. Activate environment
    a. Type in “conda activate myenv”
    b. You should see your env name on the left side of your command prompt “(myenv) C:/”
7. Install dependencies
    a. Type in “pip3 install pytorch gym Matplotlib numpy”
8. Run GA_TRAIN.py script
    a. Type “GA_TRAIN.py” or “python3 GA_TRAIN.py”. You may also use anaconda-Spyder to run the script. Training can take up to 24 hours.
9. Run GA_TEST.py script
    a. Type “GA_TEST.py” or “python3 GA_TEST.py”. You may also use anaconda-Spyder to run the script.

7 References

[1] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, 'A Survey of Deep Reinforcement Learning in Video Games', *ArXiv191210944 Cs*, Dec. 2019, Accessed: May 10, 2020. [Online]. Available: http://arxiv.org/abs/1912.10944.

[2] OpenAI, 'Gym: A toolkit for developing and comparing reinforcement learning algorithms'. https://gym.openai.com (accessed May 11, 2020)

[3] J. Grigsby, 'Advanced DQNs: Playing Pac-man with Deep Reinforcement Learning', *Medium*, Oct. 31, 2018. https://towardsdatascience.com/advanced-dqns-playing-pac-man-with-deep-reinforcement-learning-3ffbd99e0814 (accessed May 17, 2020).

[4] 'Learning to play Pac-Man: an evolutionary, rule-based approach | Request PDF', *ResearchGate*. https://www.researchgate.net/publication/4074739_Learning_to_play_Pac-Man_an_evolutionary_rule-based_approach (accessed May 11, 2020).

[5] 'Hybrid Reward Architecture (HRA) Achieving super-human performance on Ms. Pac-Man', *Microsoft Research*, Jun. 14, 2017. https://www.microsoft.com/en-us/research/blog/hybrid-reward-architecture-achieving-super-human-ms-pac-man-performance/ (accessed May 11, 2020).

[6] S. Zychlinski, 'The Complete Reinforcement Learning Dictionary', *Medium*, Nov. 24, 2019. https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e (accessed May 12, 2020).

[7] B. Osiński and K. Budek, 'What is reinforcement learning? The complete guide', *deepsense.ai*, Jul. 05, 2018. https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/ (accessed May 12, 2020).

[8] 'What Is Reinforcement Learning? - MATLAB & Simulink'. https://www.mathworks.com/help/reinforcement-learning/ug/what-is-reinforcement-learning.html (accessed May 12, 2020).

[9] V. Mnih *et al.*, 'Human-level control through deep reinforcement learning', *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: 10.1038/nature14236.

[10] V. Mnih *et al.*, 'Playing Atari with Deep Reinforcement Learning', *ArXiv13125602 Cs*, Dec. 2013, Accessed: May 11, 2020. [Online]. Available: http://arxiv.org/abs/1312.5602.

[B1] M. Roos, “Evolutionary approaches towards AI: past, present, and future,” Medium, 27-Mar-2020. [Online]. Available: https://towardsdatascience.com/evolutionary-approaches-towards-ai-past-present-and-future-b23ccb424e98. [Accessed: 17-May-2020].

[B2] MAXIM. LAPAN, DEEP REINFORCEMENT LEARNING HANDS-ON;APPLY MODERN RL METHODS TO PRACTICAL PROBLEMS OF CHATBOTS, ROBOTICS, DISCRETE OPTIMIZATION, WEB AUTOMATION. PACKT Publishing, 2020.

[B3] E. T. from the arXiv, “Evolutionary algorithm outperforms deep-learning machines at video games,” MIT Technology Review, 02-Apr-2020. [Online]. Available: https://www.technologyreview.com/2018/07/18/104191/evolutionary-algorithm-outperforms-deep-learning-machines-at-video-games/. [Accessed: 17-May-2020].

[B4] G. Surma, “Atari - Solving Games with AI (Part 2: Neuroevolution),” Medium, 17-Jan-2019. [Online]. Available: https://towardsdatascience.com/atari-solving-games-with-ai-part-2-neuroevolution-aac2ebb6c72b. [Accessed: 17-May-2020].

[B5] P. Chopra, “Reinforcement learning without gradients: evolving agents using Genetic Algorithms,” Medium, 07-Jan-2019. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-without-gradients-evolving-agents-using-genetic-algorithms-8685817d84f. [Accessed: 17-May-2020].

[B6] “Evolutionary computing and artificial intelligence,” Math Scholar. [Online]. Available: https://mathscholar.org/2018/01/evolutionary-computing-and-artificial-intelligence/. [Accessed: 17-May-2020].

[B7] “Genetic Algorithm,” Genetic Algorithm - an overview | ScienceDirect Topics. [Online]. Available: https://www.sciencedirect.com/topics/engineering/genetic-algorithm. [Accessed: 17-May-2020].

Appendix A: Project File Structure

ISA_SLS_TommyYong_PremChandran_LiWei (Root project folder)

1. **Codes**
    a. Reinforcement Learning (DQN Script in Code_DQN, use pacmanv4)
    b. Evolutionary Learning (GA scripts found here)
2. > **Report** (Project report found here)
3. > **Videos**
    a. Reinforcement Learning(3 videos of best performing moments of agent)