VIT®
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)
Vellore-632 014, Tamil Nadu, India.
www.vit.ac.in

CSE3013: Artificial Intelligence

J Component Report

Pac-Man Bot using Open AI

Submitted To

Prof. Annapurna Jonnalagadda
School of Computer Science and Engineering

Submitted By

Anuranjan Srivastava (15BCE0002)
Gaurav Thakur (15BCE0145)
Ratnasambhav Priyadarshi (15BCE0646)

Problem Statement

In this project we have created a artificial agent in Pac-Man game environment. The agent learns using Reinforcement learning to learn about its environment. There are different learning algorithm to make a computer play a game such as Q learning, Genetic algorithm, Minmax algorithm etc. For our implementation we have selected Genetic algorithm (GA) and Deep Q learning. Genetic Algorithm is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state. Like beam searches, GAs begin with a set of k randomly generated states, called the population. A Deep Q Network algorithm performs the best action based on the state of its environment.

Introduction

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear. The genetic algorithm can address problems of *mixed integer programming*, where some components are restricted to be integer-valued.
The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:

- *Selection rules* select the individuals, called *parents*, that contribute to the population at the next generation.
- *Crossover rules* combine two parents to form children for the next generation.
- *Mutation rules* apply random changes to individual parents to form children.

Each state, or individual, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s. Each state is rated by the objective function, or the fitness function. A fitness function should return higher values for better states. A generalized algorithm for GA can be written as:

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
    inputs: population, a set of individuals
```

```
        FITNESS-FN, a function that measures the fitness of an individual
    repeat
        new population ← empty set
        for i = 1 to SIZE(population) do
            x ← RANDOM-SELECTION(population, FITNESS-FN)
            y ← RANDOM-SELECTION(population, FITNESS-FN)
            child ←REPRODUCE(x,y)
            if (small random probability) then child ← MUTATE(child)
            add child to new population
        population←new population
    until some individual is fit enough, or enough time has elapsed
    return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
    inputs: x , y , parent individuals

    n ← LENGTH(x); c ← random number from 1 to n
    return APPEND(SUBSTRING(x, 1, c),SUBSTRING(y, c +1, n))
```

Now let's discuss about Deep Q Learning. At the end of 2013, Google introduced a new algorithm called Deep Q Network (DQN). It demonstrated how an AI agent can learn to play games by just observing the screen. The AI agent can do so without receiving any prior information about those games. Our goal is to connect a reinforcement learning algorithm to a deep neural network which operates directly on RGB images and efficiently process training data by using stochastic gradient updates. we utilize a technique known as experience replay where we store the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data-set $D = e_1 , ..., e_N$, pooled over many episodes into a replay memory. During the inner loop of the algorithm, we apply Q-learning updates, or minibatch updates, to samples of experience, $e \sim D$, drawn at random from the pool of stored samples. After performing experience replay, the agent selects and executes an action according to an ε-greedy policy. Since using histories of arbitrary length as inputs to a neural network can be difficult, our Q-function instead works on fixed length representation of histories produced by a function φ. Algorithm for Deep Q-learning with Experience Replay:

```
Initialize replay memory D to capacity N
Initialize action-value function Q with random weight
for episode = 1, M do
```

```
        Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequence $\Phi_1 = (s_1)$
        for $t = 1, T$ do
            With probability $\varepsilon$ select a random action $a_t$
            otherwise select $a_t = max_a\ Q^*(\varphi(st), a; \theta)$
            Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
            Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\varphi_{t+1} = \varphi(s_{t+1})$
            Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ in $D$
            Sample random minibatch of transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from $D$
            if terminal $\varphi_{j+1}$ set $y_j = r_j$
            else if $y_j = r_j + \gamma\ max_{a'}\ Q(\varphi_{j+1}, a'; \theta)$
            Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
        end for
end for
```

Each step of experience is potentially used in many weight updates, which allows for greater data efficiency. Learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates. When learning on-policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically. By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. Note that when learning by experience replay, it is necessary to learn off-policy (because our current parameters are different to those used to generate the sample), which motivates the choice of Q-learning.

We have used OpenAI library and Python to implement this project. OpenAI is a non-profit artificial intelligence (AI) research company that aims to promote and develop friendly AI in such a way as to benefit humanity as a whole. The organization aims to "freely collaborate" with other institutions and researchers by making its patents and research open to the public. OpenAI's mission is to build safe AGI, and ensure AGI's benefits are as widely and evenly distributed as possible. OpenAI focus on long-term research, working on problems that demands to make fundamental advances in AI capabilities.

Pac-Man (originally known as Puck Man in Japan) is the initial game in the Pac-Man series. It was first created by Toru Iwatani and released by Namco/Midway. Released in 1980, the game would go on to become a cultural icon worldwide. The player controls a circular character which has a pie wedge shaped mouth to eat pellets through a maze, eating Pac-Dots. When all dots are eaten, Pac-Man is taken to the next stage. Four ghosts, Blinky, Pinky, Inky and Clyde roam the maze, trying to catch Pac-Man. If a ghost chomps (touches) Pac-Man, a life is lost. When all lives have been lost, the game ends. Pac-Man is awarded a single bonus life at 10,000 points by default–DIP switches inside the machine can change the required points or disable the bonus life altogether.

Near the corners of the maze are four larger, flashing dots known as Power Pellets, provide Pac-Man with the temporary ability to eat the ghosts. The ghosts turn deep blue, reverse direction, and move slower when Pac-Man eats one. When a ghost is eaten, its eyes return to the ghost home where it is regenerated in its normal color. Blue ghosts flash white before they become dangerous again and the amount of time the ghosts remain vulnerable varies from one round to the next, but the time period generally becomes shorter as the game progresses. In later stages, the ghosts don't change colors at all, but still reverse direction when a power pellet is eaten.

In addition to Pac-Dots and Power Pellets, bonus items, usually referred to as fruits (though not all items are fruits) appear near the center of the maze. These items score extra bonus points when eaten. The items change and bonus values increase throughout the game. Sometimes, items can appear several times. Also, a series of intermissions play after certain levels toward the beginning of the game, showing a humorous set of interactions (the first being after level 2, which has a giant Pac-Man chasing a blue Blinky) between Pac-Man and Blinky. Intermissions: #1: Giant Pac-Man vs Blue Blinky. #2: Blinky chases Pac-Man, but gets cut on a white stick. #3: Number 3 has Blinky, whose wound has been repaired, chase Pac-Man, but his entire slime (body) falls off! Intermission 3 is used as all other intermissions after.

Scoring System in PacMan:

- Pac-Dot - 10 points.
- Power Pellet - 50 points.
- Vulnerable Ghosts:
  1 in succession - 200 points.
  2 in succession - 400 points.
  3 in succession - 800 points.
  4 in succession - 1600 points.
- Fruit:
  Cherry: 100 points.
  Strawberry: 300 points.
  Orange: 500 points.
  Apple: 700 points.

Melon: 1000 points.
Galaxian Boss: 2000 points.
Bell: 3000 points.
Key: 5000 points.

The four ghosts, formerly known as "monsters", are the enemies in the original arcade game. They cycle through different "modes" of behavior, colloquially known as "scatter"–where they retreat to the four corners of the maze–and "chase"–where their A.I. kicks in. Each ghost has unique A.I., programmed so that the game would not get impossibly difficult or boring. There are certain one-way areas on the maze–namely, the two "T" formations located directly above and below the Ghost Home –that the ghosts can travel down through, but can't go up through. There are also two entrances to a tunnel on either side of the maze that Pac-Man can travel through and come out the opposite side of the screen on, which will slow the ghosts down if they enter it. Yet another advantage Pac-Man has over the quartet is that he can turn slightly faster.

Literature Survey:

- **John Koza et al (1992)** While Pac-Man research began in earnest in the early 2000's, work by John Koza (Koza, 1992) discussed how Pac-Man provides an interesting domain for genetic programming; a form of evolutionary algorithm that learns to generate basic programs. The idea behind Koza's work and later that of (Rosca, 1996) was to highlight how Pac-Man provides an interesting problem for task-prioritisation. This is quite relevant given that we are often trying to balance the need to consume pills, all the while avoiding ghosts or – when the opportunity presents itself – eating them.
- **Kalyanpur and Simon et al (2001)** explored how evolutionary learning algorithms could be used to improve strategies for the ghosts. In time it was evident that the use of crossover and mutation – which are key elements of most evolutionary based approaches was effective in improving the overall behaviour. However it's important to note that they themselves acknowledge their work uses a problem domain similar to Pac-Man and not the actual game.
- **Gallagher and Ryan et al (2003)** uses a slightly more accurate representation of the original game. There actual implementation only used one ghost rather than the original four. In this research the team used an incremental learning algorithm that tailored a series of rules for the player that dictate how Pac-Man is controlled using a Finite State Machine. This proved highly effective in the simplified version they were playing.

- **Lucas et al (2005)** Two notable publications on Pac-Man are (Lucas, 2005), which attempted to create a 'move evaluation function' for Pac-Man based on data scraped from the screen and processed as features (e.g. distance to closest ghost), while (Gallagher and Ledwich, 2007) attempted to learn from raw, unprocessed information.
- **Simon and Lucas et al (2011)** Simon Lucas in conjunction with Philipp Rohlfshagen and David Robles created the Ms Pac-Man vs Ghosts competition. In this iteration, the 'screen scraping' approach had been replaced with a Java implementation of the original game. This provided an API to develop your own bot for competitions. This iteration ran at four conferences between 2011 and 2012.
- **Hado van Hasselt and David Silver et al (2016)** showed that the recent DQN algorithm, which combines Q-learning with a deep neural network, suffers from substantial overestimations in some games in the Atari 2600 domain. They demonstrated the idea behind the Double Q-learning algorithm, which was introduced in a tabular setting, can be generalized to work with large-scale function approximation. They proposed a specific adaptation to the DQN algorithm and show that the resulting algorithm not only reduces the observed overestimations, as hypothesized, but that this also leads to much better performance on several games.
- **Sutton and Barto et al (1998)** aimed to learn good policies for sequential decision problems, by optimizing a cumulative future reward signal. Q-learning is one of the most popular reinforcement learning algorithms, but it is known to sometimes learn unrealistically high action values because it includes a maximization step over estimated action values, which tends to prefer overestimated to underestimated values**.**
- **Thrun and Schwartz et al (1993)** unified the views and show overestimations can occur when the action values are inaccurate, irrespective of the source of approximation error. Of course, imprecise value estimates are the norm during learning, which indicates that overestimations may be much more common than previously appreciated.
- **Kaelbling et al (1996)** Overoptimistic value estimates are not necessarily a problem in and of themselves. If all values would be uniformly higher then the relative action preferences are preserved and we would not expect the resulting policy to be any worse. Furthermore, it is known that sometimes it is good to be optimistic: optimism in the face of uncertainty is a well-known exploration technique.
- **Mnih et al (2015)** To test whether overestimations occur in practice and at scale, they investigate the performance of the recent DQN algorithm. DQN combines Q-learning with a flexible deep neural network and was tested on a varied and large set of deterministic Atari 2600 games, reaching human-level performance on many games.

Genetic Algorithm Implementation

PACMAN is a pareto-optimization problem. PACMAN can either be concerned about his safety from ghosts, or concerned about collecting the maximal number of pellets. He can't do both, because over-collecting exposes him to ghosts. While we are on the subject, you could evolve all sorts of pathological PACMEN. Such as a PACMAN who is primarily concerned with killing ghosts.

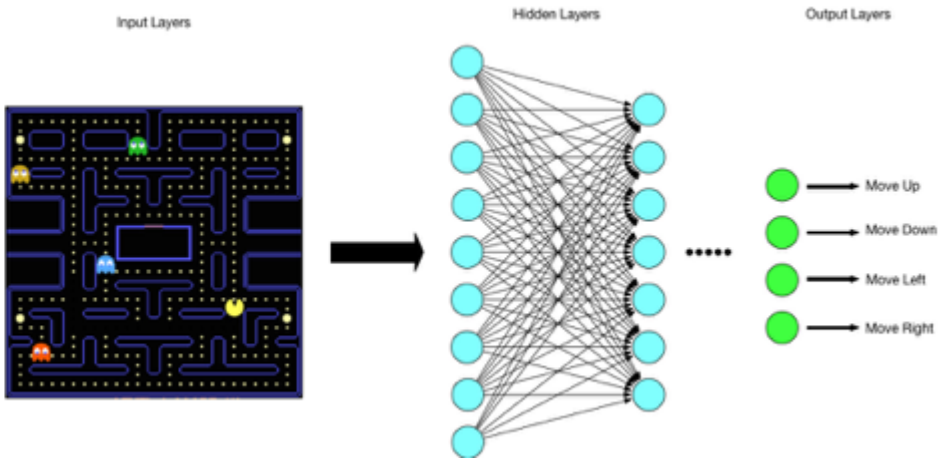
Our PACMAN agent is not going to evolve a mere feed-forward mapping of environment states to actions. Instead he is going to collect a laundry list of every object/sprite that exists in the entire universe of the PACMAN game, which are:

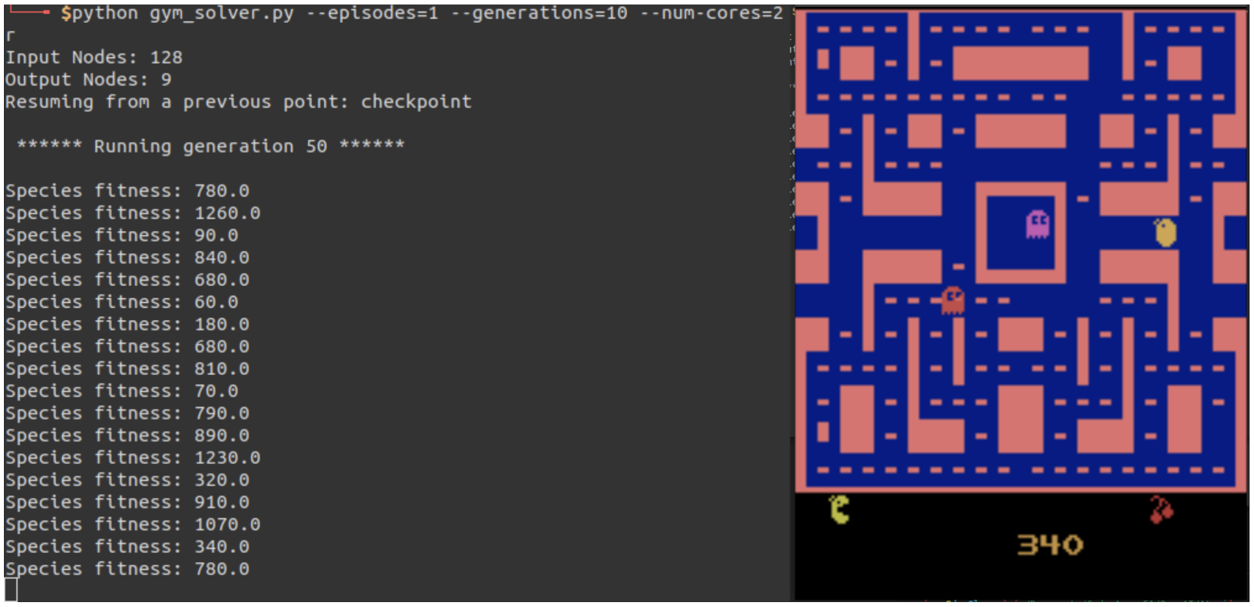
- PACMAN himself
- Ghosts
- Pellets
- Super pills
- Cherries
- Walls
- Corners

For our project we are using NeuroEvolution of Augmenting Topologies (NEAT) genetic algorithm. NEAT algorithm is a genetic algorithm (GA) for the generation of evolving artificial neural networks developed by Ken Stanley in 2002 while at The University of Texas at Austin. It alters both the weighting parameters and structures of networks, attempting to find a balance between the fitness of evolved solutions and their diversity. It is based on applying three key techniques: tracking genes with history markers to allow crossover among topologies, applying speciation (the evolution of species) to preserve innovations, and developing topologies incrementally from simple initial structures.

On simple control tasks, the NEAT algorithm often arrives at effective networks more quickly than other contemporary neuro-evolutionary techniques and reinforcement learning methods.

NEAT attempts to simultaneously learn weight values and an appropriate topology for a neural network. The NEAT approach begins with a perceptron-like feed-forward network of only input neurons and output neurons. As evolution progresses through discrete steps, the complexity of the network's topology may grow, either by inserting a new neuron into a connection path, or by creating a new connection between (formerly unconnected) neurons. We have used NEAT-python package provides us with functions to create a neural network, calculate statistics for different generations, the fitness function and the method to mutate the genome, which in our case is the weights of the neural network. Hence we can easily prototype models for our algorithm.





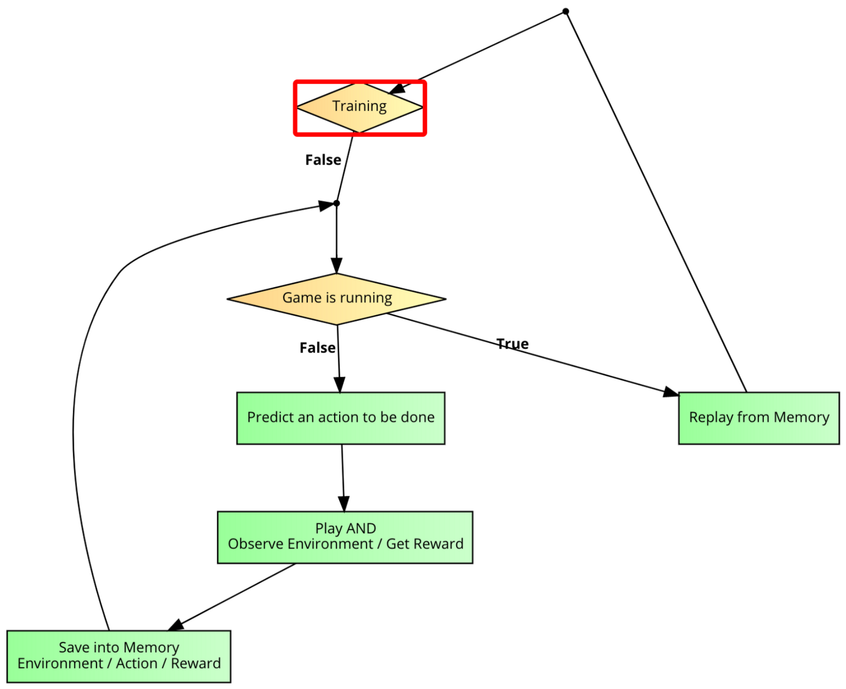
```
└── $python gym_solver.py --episodes=1 --generations=10 --num-cores=2
r
Input Nodes: 128
Output Nodes: 9
Resuming from a previous point: checkpoint

 ****** Running generation 50 ******

Species fitness: 780.0
Species fitness: 1260.0
Species fitness: 90.0
Species fitness: 840.0
Species fitness: 680.0
Species fitness: 60.0
Species fitness: 180.0
Species fitness: 680.0
Species fitness: 810.0
Species fitness: 70.0
Species fitness: 790.0
Species fitness: 890.0
Species fitness: 1230.0
Species fitness: 320.0
Species fitness: 910.0
Species fitness: 1070.0
Species fitness: 340.0
Species fitness: 780.0
```

We have observed that genetic algorithm doesn't give good results for a moderate amount of iterations. We could only get a maximum score of about 340 in 50 generations. Of course, if we increase the no. of generations, the score will increase, but we want to get the most optimal result. Hence, we will be switching to Q Learning and try to obtain better score.

Deep Q Learning Network Implementation


Training
False
Game is running
False
True
Predict an action to be done
Replay from Memory
Play AND
Observe Environment / Get Reward
Save into Memory
Environment / Action / Reward

We have used Keras a high-level neural networks API, written in Python and capable of running on top of either TensorFlow, CNTK or Theano. Keras makes it really simple to implement a basic neural network.

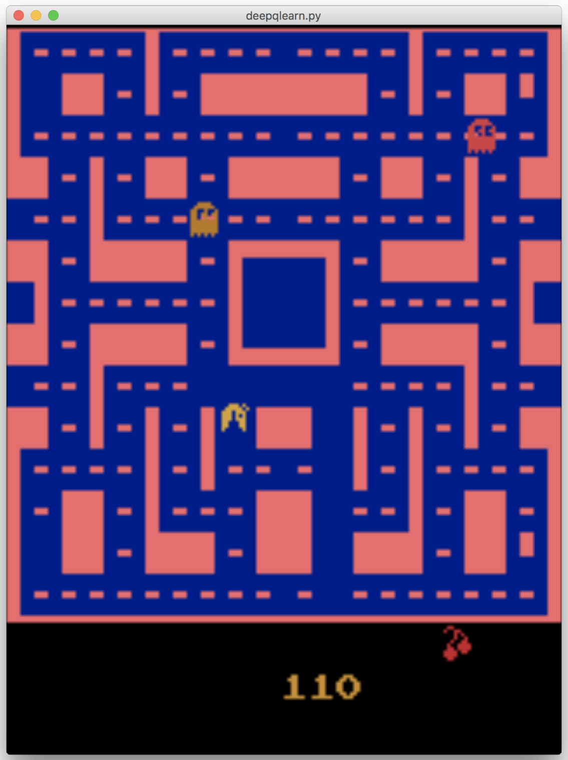
In the DQN algorithm, there are also two very important parts: the *remember* and *replay* methods. Both are pretty simple concepts and can be better explained as how we live a situation as humans: we remember what we did after performing each action and when we have enough elements, we are trying to recreate the situation in our mind. We will have some hyperparameters for our model as listed them below:

- *learning_rate* - This indicates how much neural network learns from the loss between the target and the prediction in each iteration.
- *gamma* - This is used to calculate the future discounted reward.
- *exploration_rate* - At the beginning, the lack of experience of our agent makes us choose randomly an action and when the agent gets more experienced, we let it decide which action to undertake.
- *exploration_decay* - We want to decrease the number of explorations as it gets better and better at playing games.
- *episodes* - This indicates how many games we want the agent to play in order to train itself.

The loss is a value that indicates how far our prediction is from the actual target. For example, the prediction of the model could indicate that it sees more value in pushing the left button when in fact it can gain more reward by pushing the right button. Our goal is to decrease the loss, which is the gap between the prediction and the target. We first pick randomly an action, and observe the reward. It will also result in a new State. Keras takes care of the most difficult tasks for us.

One of the most important steps in the learning process is to remember what we did in the past and how the reward was bound to that action. Therefore, we need a list of previous experiences and observations to re-train the model with those previous experiences. We will store our experiences into an array called memory and we will create a remember() function to append state, action, reward, and next state to the array memory. And the remember() function will simply store states, actions and resulting rewards into the memory.

Now that we have our past experiences in an array, we can train our neural network. Let's create a function replay(). We cannot afford to go through all our memory, it will take too many resources. Therefore, we will only take a few samples and we will just pick them randomly. To make the agent perform well in mid-term and long-term, we need to take into account not only the immediate rewards, but also the future rewards we are going to get. In order to implement that, we will use gamma. In such a way, our DQN agent will learn to maximize the discounted future reward on the given State.



Our agent will randomly select its action at first by a certain percentage, called 'exploration rate' (or 'epsilon'). At the beginning, it is better for the DQN agent to try different things before it starts to search for a pattern. When our DQN agent has enough experience, the agent will predict the reward value based on the current State. It will pick the action that will give the highest *reward.np.argmax()* is the function that returns the index of the highest value between two elements in the act_values[0].

Conclusion

This study evaluates the effectiveness of Genetic Algorithm and Deep Q-learning applied to PACMAN. This study evaluates the effectiveness of Deep Q-learning applied to PACMAN. For a game like PacMan, the input layer of neural net is 128 nodes and because of this Genetic Algorithm is inefficient as compared to Q-Learning. Deep-Q model attains more score in lesser amount of training and data. Hence we can say that Deep-Q learning is better suited for PACMAN than Genetic Algorithm. The code can be found here.

References

[1] Fernando Dominguez-Estevez, Antonio A. Sanchez-Ruiz, and Pedro Pablo Gomez-Martin, "Training Pac-Man bots using Reinforcement Learning and Case-based Reasoning", CoSECivi, 144-156,2017.

[2] Bom, L., Henken, R., Wiering, M.: Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In: ADPRL. pp. 156–163. IEEE (2013).

[3] Williams, P.R.: Ms. Pac-Man vs Ghosts AI. http://www.pacmanvghosts.co.uk, accessed: 2017-06-25.

[4] Stuart Russell and Peter Norvig, Artificial: Intelligence A Modern Approach, Third Edition, PRENTICE HALL

[5] K.F. Man, Sam Kwong and K.s. Tang "Genetic algorithms: Concepts and Applications", IEEE Transactions on Industrial Electronics 43(5):519 - 534·November 1996.

[6] J. S. De Bonet and C. P. Stauffer. Learning to play pacman using incremental reinforcement learning. Retrieved from http://www.ai.mit.edu/people/stauffer/Projects/PacMan/ (19/06/03), 1999.

[7] M. Gallagher. Multi-layer perceptron error surfaces: visualization, structure and modelling. PhD thesis, Dept. Computer Science and Electrical Engineering, University of Queensland, 2000.

[8] S. Gugler. Pac-tape. Retrieved from http://www.geocities.com/SiliconValley/Bay/4458/ (21/05/01), 1997.

[9] A. Kalyanpur and M. Simon. Pacman using genetic algorithms and neural networks. Retrieved from http://www.ece.umd.edu/ adityak/Pacman.pdf (19/06/03), 2001.

[10] J. Koza. Genetic Programming: On the Programming of Computer by Means of Natural Selection. MIT Press, 1992.

[11] S. Lawrence. Pac-man autoplayer. Retrieved from http://www.cis.rit.edu/ jerry/Software/ autopac/ (21/05/01), 1999.

[12] J. P. Rosca. Generality versus size in genetic programming. In J. Koza et. al., editor, Genetic Programming (GP96) Conference, pages 381–387. MIT Press, 1996.