Pacman Using Reinforcement Learning
Project Report

Submitted in partial fulfilment of the requirements for the course of

ENPM808F – Robot Learning

By

Adheesh Chatterjee
NITHISH Kumar S

Course Faculty
Prof Donald Sofge

UNIVERSITY OF MARYLAND 18 56

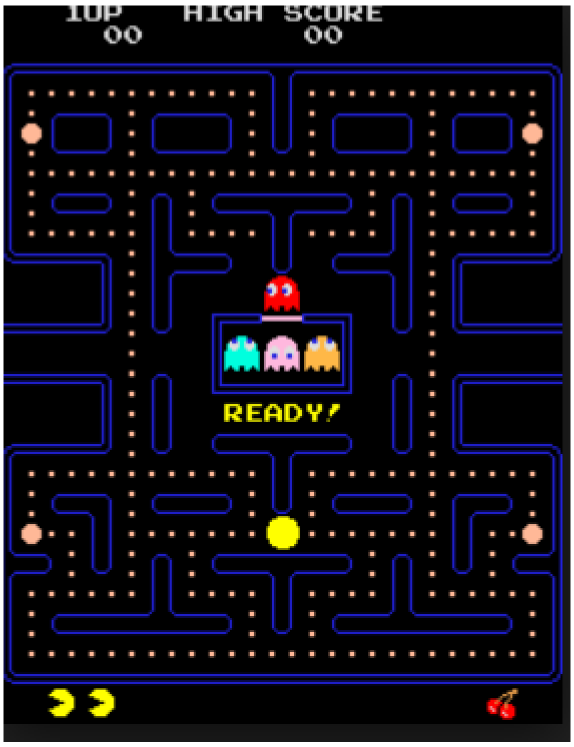
A. JAMES CLARK
SCHOOL OF ENGINEERING

ABSTRACT

This project illustrates how reinforcement learning is applied to the Pacman game to make it play itself. In order to do so, various approaches including Q Learning, Approximate Q Learning and Deep Q Learning are used to train the game and the algorithms are further explained. Furthermore, the game is tested on various grid sizes with changed parameters and the results are explained. The win rate is calculated and plotted. Final conclusions are drawn from the plots and the optimal learning algorithm is highlighted.

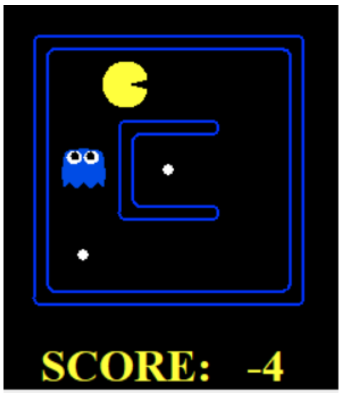
Keywords – Pacman, Q Learning, Approximate Q Learning, Deep Q Learning

INTRODUCTION

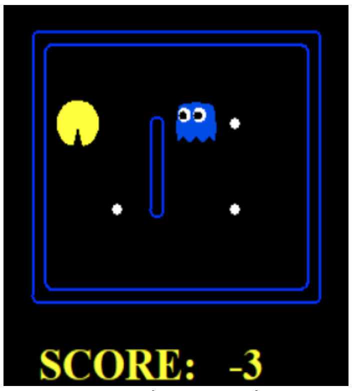


Pacman is a game that takes place on a single screen. The player controls the Pacman character and can move him around the maze. The main goal of the game is to collect all of the pellets in the maze. Once the player has collected all of the pellets, the level is complete, and the game advances to the next level. There are 4 ghosts that will make it difficult for the player to collect all of the pellets. They will chase the player in various ways and if the player comes into contact with any of the ghosts, the player will die. When the player loses all of his lives, the game is over. There are 4 power pellets and if the player eats any of these power pellets, then the player can eat the ghosts. When the ghosts get eaten they will respawn.

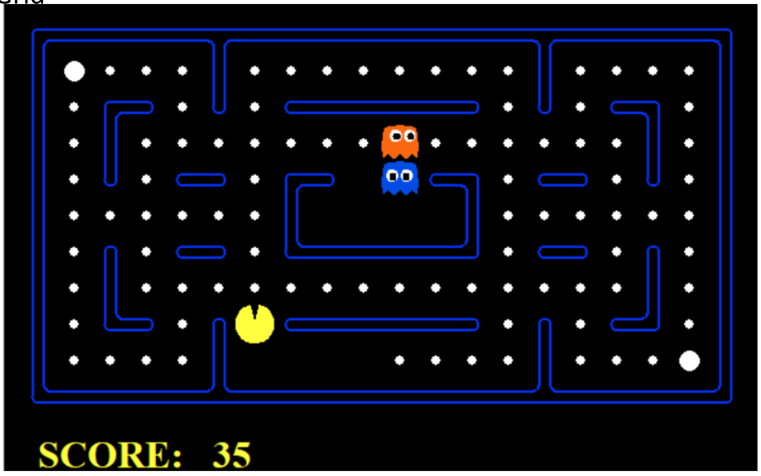
The goal of this project is to train the Pacman agent to avoid the ghosts and eat the food to increase the score and eventually win the game. We train the Pacman agent on different using different size grids. All the reinforcement methods we implement in this project are based on the code that runs the emulator for the Pacman game [A].



Small Grid



Medium Grid



BACKGROUND/RELATED WORK

A lot of the literature related to reinforcement learning on games deals with Pong and other Atari games in general. We got an understanding of the basic Q-learning from [1] We use [2] and [3] to understand the Deep Q learning algorithm and how it is implemented. There has also been a lot of research on Deep Recursive Q Networks [4] where a deep Q network was implemented with a LSTM which better handled information loss and on Dual Q Networks [5] where 2 DQN's are used with 2 different weights 2 Q values are used to estimate an action, this helps to avoid exploitation. Even though all these alternatives seem to make the model more robust, we decided to implement a DQN and proposed all this related work as future work that could be added to our implementation of the Pacman.

APPROACH

1) Q Learning

Initially, we employed the use of Q Learning to train the model. Q Learning is a method of determining the value of state-action pairs. The Q Value, is represented by Q(S,a), where S is the State and a is the action. This value is generated from the reward received for taking action a.

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t,a_t)}_{\text{old value}} \right)$$

From the above formula we see that we have to define certain parameters for the algorithm – the learning rate and the discount factor. The learning rate defines the degree to which new information will be considered by the agent. A high learning rate, changes the true values of the agent very quickly, while a lower one can mean adjustments will be slow. We use a lower learning rate so that we have convergence at the end. The discount factor controls how much the agent knows about the future states when making a choice. A high discount factor means the future state will matter as much as the current state while a lower one means, the decision is made in the moment without any considerations. For this project, we used a learning rate of 0.2 and a discount factor of 0.8.

Epsilon Greedy Strategy

We use Epsilon Greedy strategy for exploration. Essentially, it chooses random actions an epsilon fraction of the time and follows its current best Q-values otherwise.

We do this to balance the exploitation-exploration trade-off. Although, this could sometimes under explore the variant space before exploiting what it estimates to be the strongest variant. A possible alternative to epsilon-greedy could be a future scope for this project. We use an epsilon value of 0.1

2) Approximate Q Learning

Approximate Q Learning assumes the existence of a feature function $f(s,a)$ over the state and action pairs which will yield a series of vectors $f_1(s,a), f_2(s,a) \ldots f_n(s,a)$. In approximate q learning, the Q function is defined as –

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i$$

Where each weight w_i is associated with a particular feature $f_i(s,a)$

The weights get updated according the equation –

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$
$$difference = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a)$$

The idea here is that the Q value is represented as weighted linear sum of the feature values for the state. After taking an action from a state we go to new state and get a reward. Based on the features, the reward and the next state we update the weights

3) Deep Q Learning

The general Q-Learning uses a table to store the Q values for each corresponding state-action pair. But due to the enormous size of the state space in the pacman, the required memory and number of iterations makes the convergence practically impossible. Approximate Q learning would give a reasonable performance, but humans don't look for features while playing the game. We look at the whole grid to understand and make decisions. Deep Q networks do the same. Also, it interpolates the Q-values for each state-action pair using a CNN rather than a table.

In Approximate Q Learning, since the features are hand picked, it is possible to miss out some significant features. By using a CNN that extracts all the features and outputs the Q Values for a given state and action, it is optimal.

We utilize a Q-network and a target network. Q- network is updated using back propagation and target network is where the training happens. It is an earlier copy of the Q-Network. Also, experience replay helps us update the Q-network properly.

IMPLEMENTATION

Pacman Environment:

To implement Q learning approaches we need an game environment that gave easy access to states, rewards. In order to run the networks for a number of iterations the game simulation had to happen in real time. It allowed us to visualize the results of the techniques with ease.The Pacman implementation by the Berkeley AI fit our needs and it really suits the requirements for RL research. [A]

This environment provided us python class implementation for the grid world environment, feature extraction, reinforcement learning, visualization and its own data structures for efficient implementation.

We were able to easily add new Q-Learning agent, approximate-Q agent and a Deep-Q agent.

We tried the algorithms on three different grids namely small grid (3x3), medium grid(4x4) and a classic grid (16x9).

The rewards chosen were

+1	To collect a food particle.
-1	For each step taken without collecting food
+500	Game completion
-500	Killed by a ghost

Q-Learning Implementation:

Q learning was implemented a class with the Q-table defined as a dictionary. The following parameters gave us the best results.

learning rate : 0.2 , discount rate = 0.8, epsilon = 0.1

The Q agent is an object that takes in the parameters and updates the Q-table until the maximum number of iterations. Testing was done for 100, 1000 and 2000 iterations for the small grid, the medium grid and the classic grids

Approximate Q-Learning Implementation:

The following features were hand picked for this implementation.

- Distance to closest food
- Number of ghosts 1 step away
- Distance to closest capsule
- Activation of capsule when ghost is nearby
- Scared Ghost nearby

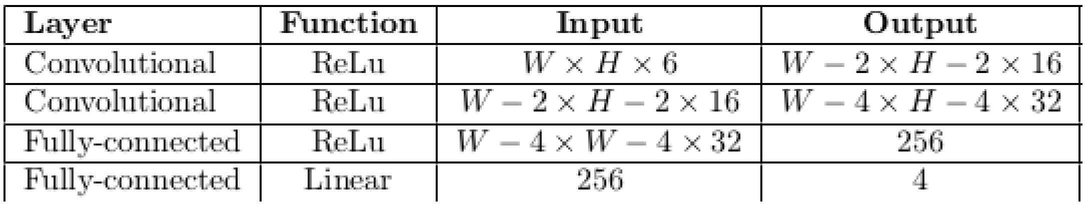
Out of these the distance to closest food and number of ghosts one step away are most important ones. If removed from the feature space, the network couldn’t converge.

The update function updates the tensor of n weights (n-1 features, 1 bias element). The features are expressed as a count of the number of grids from the current position of the pacman. A dot product of the weight tensor and the feature vector provides us with the Q-value for the state.

Deep Q-Learning Implementation:

We used the tensorflow based DQN that was originally developed to play Atari games but we make modification to the layers, optimizer and the activation function. Max pooling layer was removed.

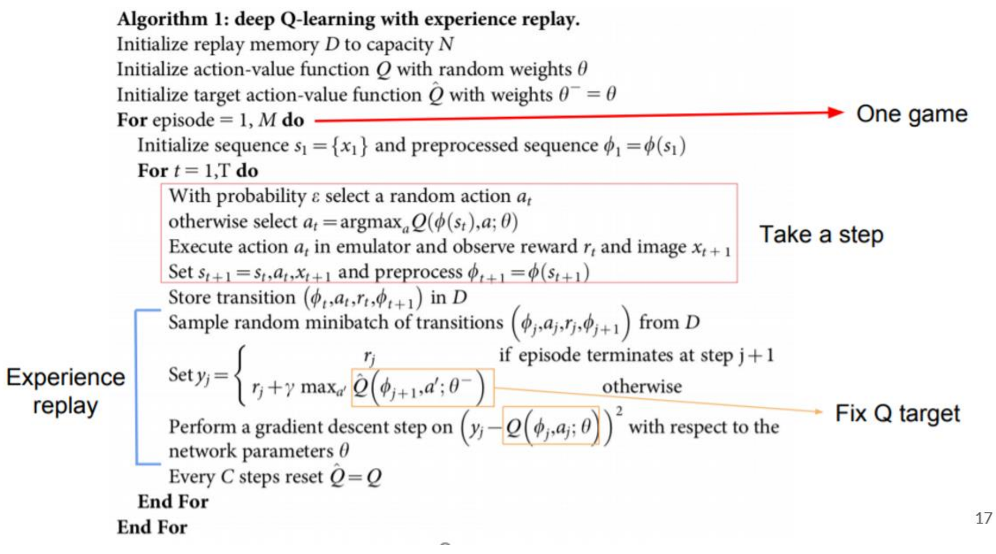
A Convolutional Neural Network (CNN) was used which is a variation of a regular feed-forward neural network. The architecture of a CNN is designed to exploit the local-connectivity typically found in high dimensional data such as images. The CNN is composed of 2 convolutional layers, one dense layer and one output layer. We use the ReLu activation function for the Convolutional layers and the dense layer, and for the final layer we use a linear activation function.The CNN model, described above is tabulated in further detail as follows:



Layer	**Function**	**Input**	**Output**
Convolutional	ReLu	$W \times H \times 6$	$W - 2 \times H - 2 \times 16$
Convolutional	ReLu	$W - 2 \times H - 2 \times 16$	$W - 4 \times H - 4 \times 32$
Fully-connected	ReLu	$W - 4 \times W - 4 \times 32$	256
Fully-connected	Linear	256	4

In the deep Q learning algorithm we employ, we use deep networks: the Q Network and the Target network. Every episode, the Q net is updated by back propagation and the update function just as the normal Q learning algorithm. The target Q network is just an earlier copy of the Q net. Once, every 1000 iterations, we substitute the target net by the updated Q net. As per [8], when the most up-to-date Q-network is not directly used to calculate Q-values, improves the performance of the Deep Q-learning algorithm significantly.

The deep Q learning algorithm employed is described below –


One game
Take a step
Experience replay
Fix Q target
17

```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t$ = 1,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    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 $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

The inputs to the CNN are as a series of logical matrices defining the position of our features in the environment. The output is the Q value based on the 5 possible actions.

The state input to the network is as follows:

$$\underset{walls}{\begin{pmatrix} 1&1&1&1&1 \\ 1&0&0&0&1 \\ 1&0&1&1&1 \\ 1&0&0&0&1 \\ 1&1&1&1&1 \end{pmatrix}}, \underset{Pac-man}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{dot}{\begin{pmatrix} 0&0&0&0&0 \\ 0&1&1&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{capsules}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}}, \underset{ghosts}{\begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&0 \end{pmatrix}}$$

A combination of the current and the past state is the input to the network as it gives us the temporal information. So each is a tuple of past and current state { S2, S1}.

The improvement in the learning output is enabled by using randomly sampled Experience replay. During each iteration of the Qnet the tuple of

{ Current state, Action, Next state, Reward } is added to the Experience list until it maximum capacity of 1000 is reached. After each 1000 iterations, the target net is trained with a randomly sampled batch of size 32 from the Experience list. Tuples are popped when max limit is reached to pave way for addition of new elements.

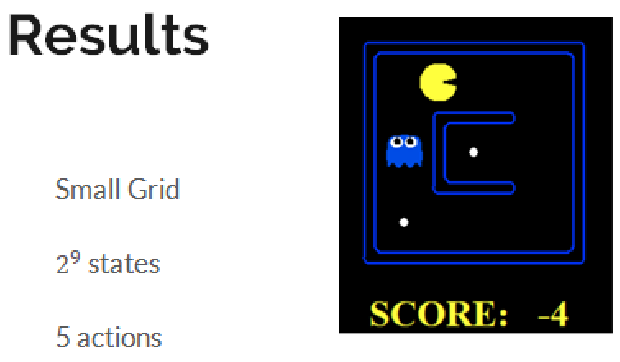
We finally simulated the trained algorithms on the environment for 10 games to check the results. The win % and loss % was stored for the entire training and we plotted the results to show improvement in performance over various iterations.

RESULTS

Q Learning

Q learning was implemented on the small grid and the results for 100, 1000 and 10000 iterations were plotted.

Small Grid

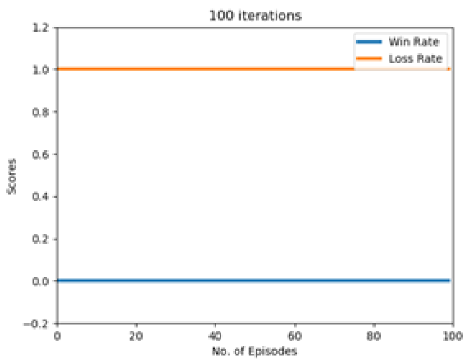


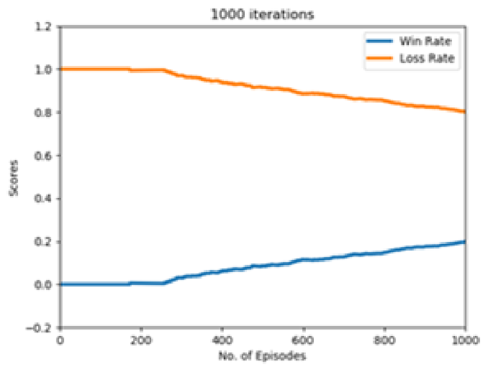
Results

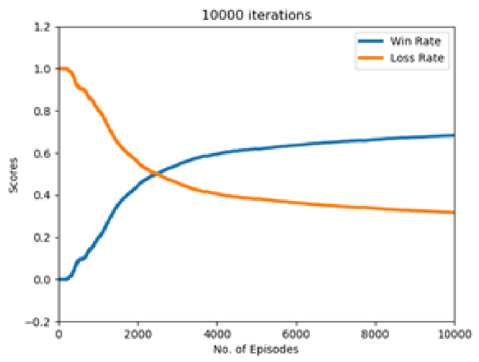
Small Grid

2^9 states

5 actions







Medium Grid

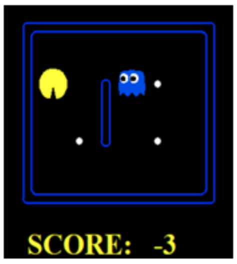
Q learning was implemented on the medium grid and the results for 100, 1000 and 10000 iterations were plotted.

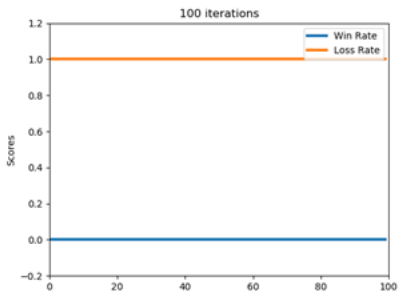
Results

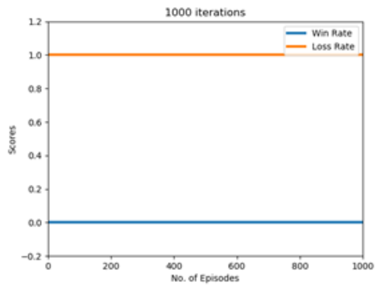
Medium Grid

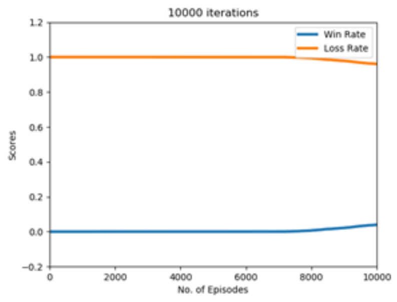
2^{16} states

5 actions









8

Classic Grid

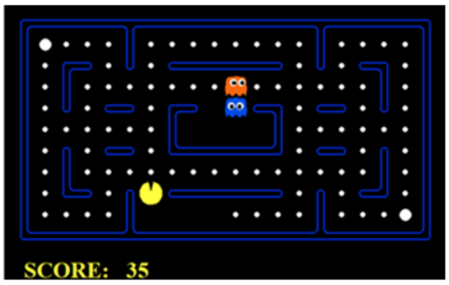
Q learning was implemented on the classic grid and the results for 100, 2000 and 4000 iterations were plotted.

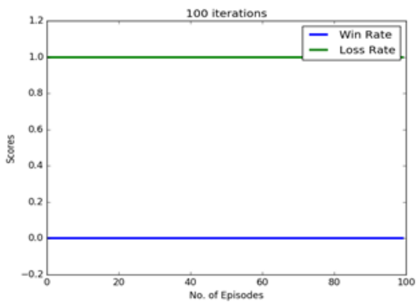
Results

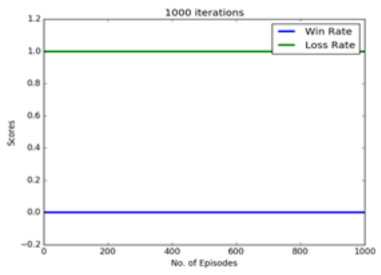
Classic Grid

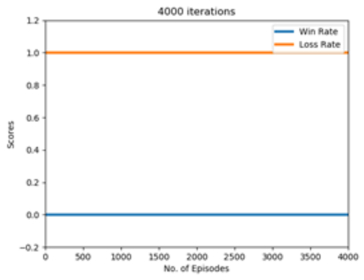
2^{144} states

5 actions







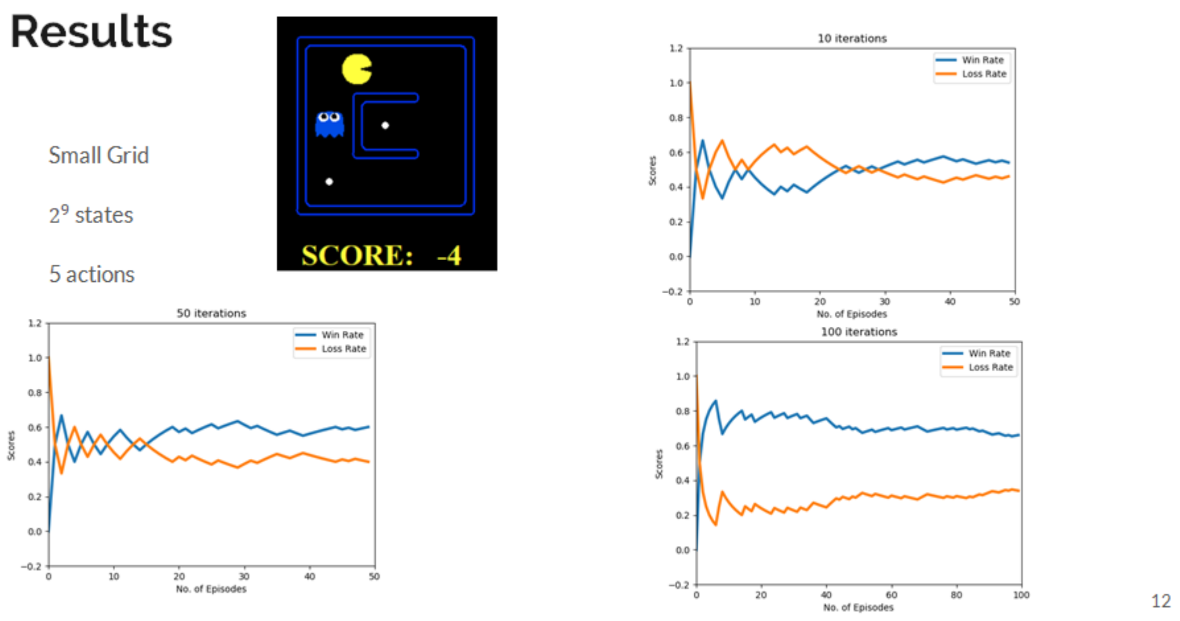


9

Approximate Q Learning

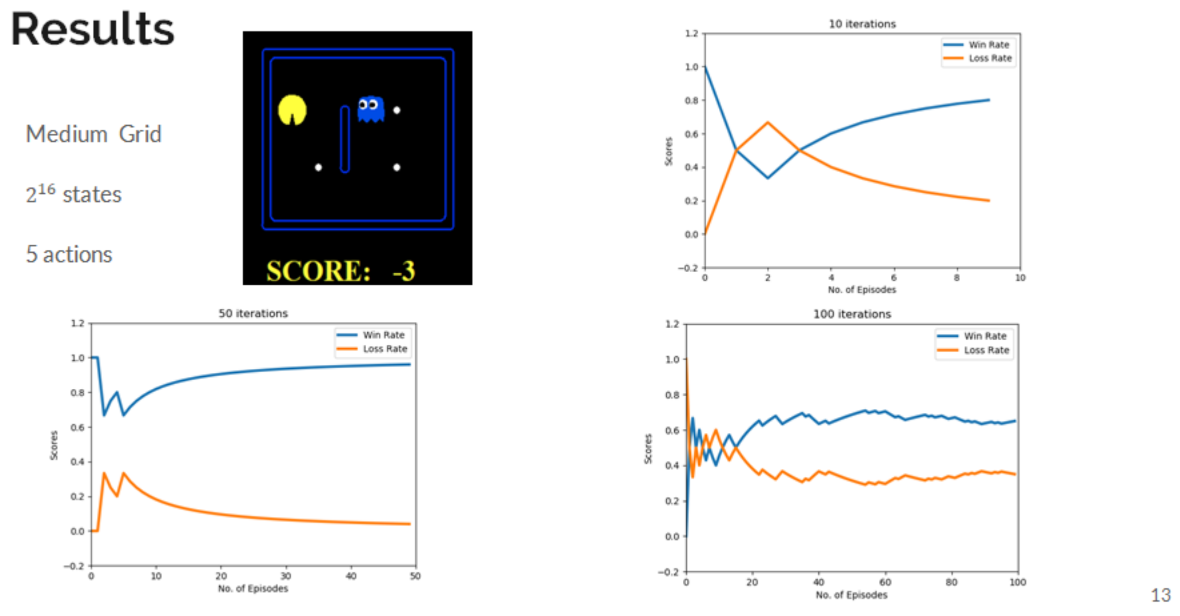
Approximate Q learning was implemented on the small grid and the results for 10, 50 and 100 iterations were plotted.

Small Grid



Approximate Q learning was implemented on the small grid and the results for 10, 50 and 100 iterations were plotted.

Medium Grid



Approximate Q learning was implemented on the Classic grid and the results for 10, 50 and 100 iterations were plotted.

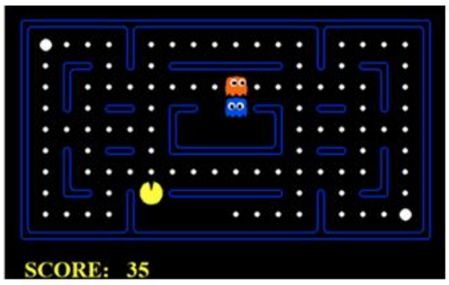
Classic Grid

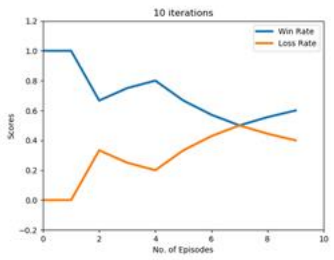
Results

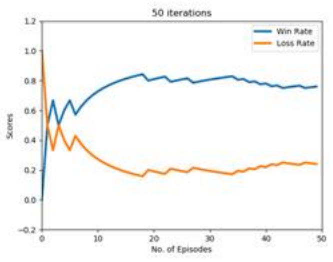
Classic Grid

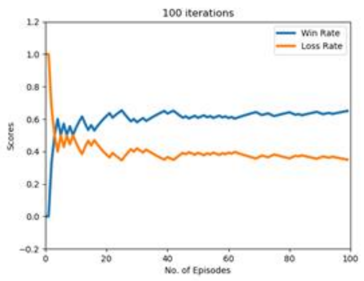
2^{144} states

5 actions









14

Deep Q Learning

Deep Q learning was implemented on the smallgrid and the results for 2000, 5000 and 10000 iterations were plotted.

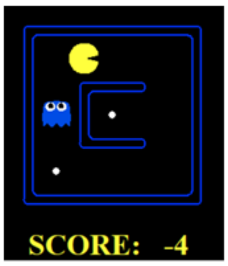
Small Grid

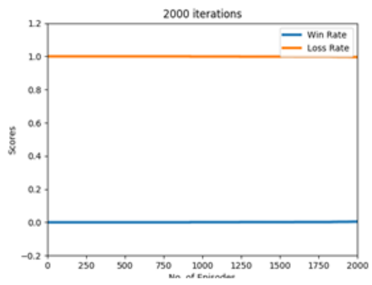
Results

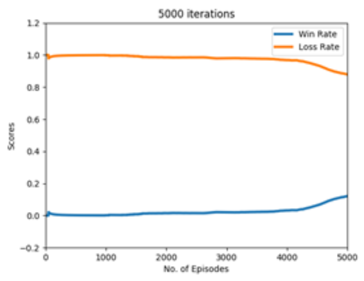
Small Grid

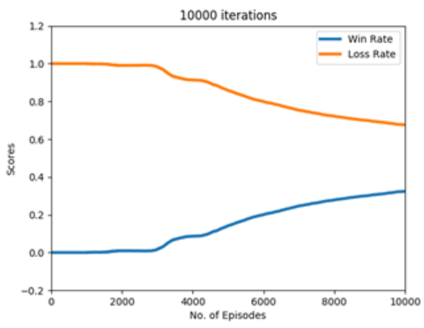
2^9 states

5 actions









19

Deep Q learning was implemented on the medium grid and the results for 200, 4000 iterations were plotted.

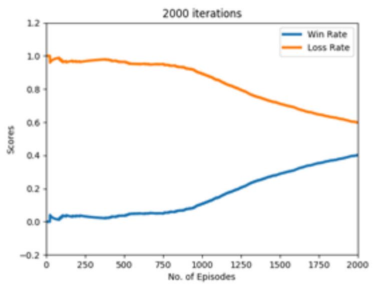
Medium Grid

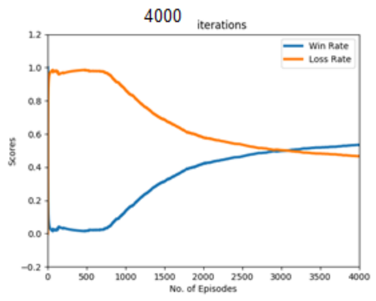
Results

Medium Grid

2^{16} states

5 actions





20

Deep Q learning was implemented on the Classic grid and the results for 2000 iterations were plotted

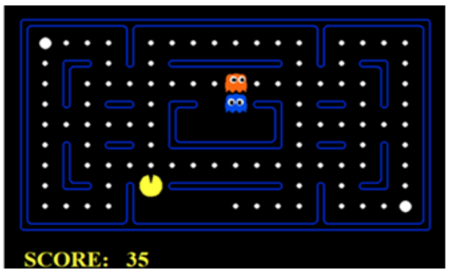
Classic Grid

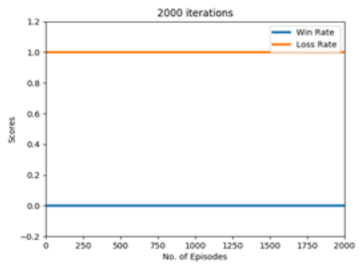
Results

Classic Grid

2^{144} states

5 actions





21

ANALYSIS

Q learning

For the small grid, we notice that for 100 iterations, we still have no wins. Only at 1000 iterations do we see the win % start to increase. When we run the code for about 10000 iterations, we observe that the win rate starts to overcome the loss rate at around 2000 or so iterations, after which it becomes steady at around 70-80%.

For the medium grid, for 100 as well as for 1000 iterations, the game is not able to register a single win. Only when we iterate over 10000 times, do we see the win% start to increase. When we ran it for around 10000 iterations, we observed that the win percent was just around 1%. This showed that the agent takes too long to train and hence it should take a larger number of games to train it on the Classic grid. So the Q Learning algorithm is not scalable and should essentially fail for the classic grid.

When we ran it for our classic grid, our hypothesis was proven right. Even for 10000 iterations, the Q learning algorithm did not produce a single win.

Approximate Q learning

The performance of the Approximate Q learning algorithm was significantly better than the simple Q learning algorithm.

For the small grid, we notice that for a mere 50 iterations, we get a 60% win rate. The medium grid gives us almost a 100% win rate and the classic grid gives us a 60% win rate.

Such a drastic improvement in performance shows the advantages of defining explicit features. This is due to the fact that states can share many features, which allows generalization to unvisited states and makes behaviour more robust: making similar decisions in similar states; otherwise, the agent needs to have explored each unique game state during the training phase before it can perform well in the test phase. As the state space increases exponentially in terms of the complexity of the game, the size of the training set also has to grow exponentially before the agent starts learning to play.

Deep Q learning

The performance of the Deep Q learning algorithm was lower than we expected. Where only after 10000 iterations does it start to converge on a small grid, it requires 4000 iterations to start winning on a medium grid and we believe it should converge at around 3000 iterations for a classic grid.

However, a Deep Q Learning network takes too much processing power and is not a feasible option for small grids. Since in Deep Q Learning, we are not handpicking the features, it should essentially run for more iterations, but it doesn’t perform as well as we’d hoped. As per the logic of neural networks though, it should give us a very good output, which leads us to believe that the features may have not been fit properly or may have been over fit as the Pacman agent loses track very easily. But it is definite that by training it for a greater number of episodes, it will converge.

Endgame Issue - While simulating the Pacman using DQN we noticed that towards the end of the game, when the Pacman has just a few food tokens to collect and they are at the opposite end of the grid, it either chooses to become static or it keeps revolving around the same spots due to the ghosts. We believe the reason for this is because the negative reward to reach the last food tokens could seem larger than the reward it gets by collecting the food tokens. So rewards will have to be adjusted accordingly during the last phases of game or we could force it to exploit rather than explore.

CONCLUSIONS

From the results obtained, we can conclude that the Q learning Algorithm performs poorly for a medium grid and a classic Pacman grid, even though it works well for a small Grid.

The performance of the Approximate Q learning algorithm was significantly better than the simple Q learning algorithm. It gave very good output and the agent was trained almost perfectly

The performance of the deep Q learning was lower than we expected however. This could be improved by using powerful hardware to train for higher number of iterations until convergence is achieved.

The code can for the project can be viewed on https://github.com/adheeshc/Pacman--Reinforcement-Learning

FUTURE WORK

As discussed previously, we feel the Pacman game can be trained better using either a Deep Recurrent Q Learning Network or a Double Q Learning Network to improve the Deep Q learning algorithm and also be able to deal with the end game issue. We also feel that this should be tested for an increased number of ghosts and the difficulty level can be increased with better ghost heuristics. For Exploration/Exploitation trade-off, instead of Epsilon-Greedy, another possible algorithm like Thomas Sampling can be considered to make it more robust. Learning from demonstration can be tried to make it learn by watching a user play the game.

BIBLIOGRAPHY

[A] Berkeley Pacman Code http : //ai.berkeley.edu/projectoverview.html

[1]https://medium.freecodecamp.org/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013

[3] David Silver, Andrei A. Rusu et al. Human-level control through deep reinforcement learning. doi:10.1038/nature14236, 2015.

[4] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. arXiv preprint arXiv:1507.06527, 2015.

[5] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. arXiv preprint arXiv:1509.06461, 2015.

[6]Abeynaya Gnanasekaran, Jordi Feliu Faba, Jing A, Reinforcement Learning in Pacman , Project Thesis, Stanford University, 2017

[7] Ioannis Antonoglou Tom Schaul, John Quan and David Silver. Prioritized experience replay. arXiv preprint arXiv:1511.05952, 2016.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., Nature 2015, 518, 529–533.