Reinforcement learning in Pacman

Sofya Denoun & Prannoy Noel Bhumana

June 1, 2020

Abstract

We applied various reinforcement learning methods on the classical game Pacman. We study and compare Q-learning, approximate Q-learning based on the total rewards and win-rate. While Q-learning has been proved to be quite effective on smallGrid, it becomes inefficient to find the optimal policy in large grid-layouts. In approximate Q-learning, we handcraft 'intelligent' features to feed into the game. The main purpose of this project is to investigate the effectiveness of Q-learning based on the context of the Pacman game.

1 Problem definition

This project is divided into several small tasks to be completed accordingly. The first task revolves around designing Value-Iteration Agent. This agent can plan its action given knowledge about environment before even interacting with it. For this we are required to compute Q values of the actions that the agent will follow and choose best action which will return maximum Q value. After designing value iteration agent, we need to change value of discount, noise and living reward to achieve best optimal policy.

Next task is to write code for Q learning agent. This agent unlike value iteration agent needs to actively interact with the environment so that it can learn through experiences. We need to approximate the state-action pairs Q-function from the samples of Q(s, a) that we observe during interaction with the environment. After successful approximation, the agent should be able to take best action to achieve goal. Q-learning Pacman should be able to win at least 80% of time. However, as we try to use this agent in bigger layouts, – for instance, the medium Grid world – PacMan's training somehow will not work well anymore. To solve this problem, we should optimize our agent to also implement an approximate Q-learning agent that learns the weights for features of states, where many states might share the same features.

2 Infrastructure

In this final project, we are not specifically building a new simulator. Instead, we are using the existing Pac-Man game simulator from ***UC Berkeley CS188 Introduction to Artificial Intelligence***'s project code to apply reinforcement learning technique to the Pac-Man.

3 Background Review

The implementation of Q-learning has been prominent and widely used in the A.I. area. The most obvious advantage of Q-learning technique is that we do not have to have a complete model of the environment as Q-learning is a model-free technique. Also, Q-learning might suffer from the slow rate of convergence, especially the case when the discount factor is very close to 1. There are some other known reinforcement learning algorithms, such as brute-force, Monte Carlo method and direct policy search.

The brute force method involves two main steps. Firstly, for each probable policy, sample returns while following the policy. Secondly, we choose the policy with the biggest expected return. However, there is a catch in this algorithm since the number of the policies can be really large – or even infinite! – and returns variance may be high so it requires many samples to estimate the return of every policy.

Monte Carlo method is an algorithm that mimics the iteration of policy. Monte Carlo is used in one of the two steps in policy iteration (that is the policy iteration step). Given a deterministic and stationary policy π, our goal is to compute the value Q^{π} (s, a) for all pairs of (s, a). Then, we compute the estimation by getting the average value of the sample returns. This is the end of the policy iteration step. In the next policy iteration step (which is policy improvement), we compute the greedy policy with respect to Q . This, in turn, will return an action that maximizes Q(s,*). This method, however, works in small and finite MDP's only. Besides, this method might put too much time on evaluating a sub optimal policy.

The direct policy search searches 'directly' in the parts of the subset of the policy space, then the problem becomes a stochastic optimization problem. This type of search has been used in the robotics application. The downside is it may converge slowly when it's given noisy data. Additionally, it may stuck in the local optima.

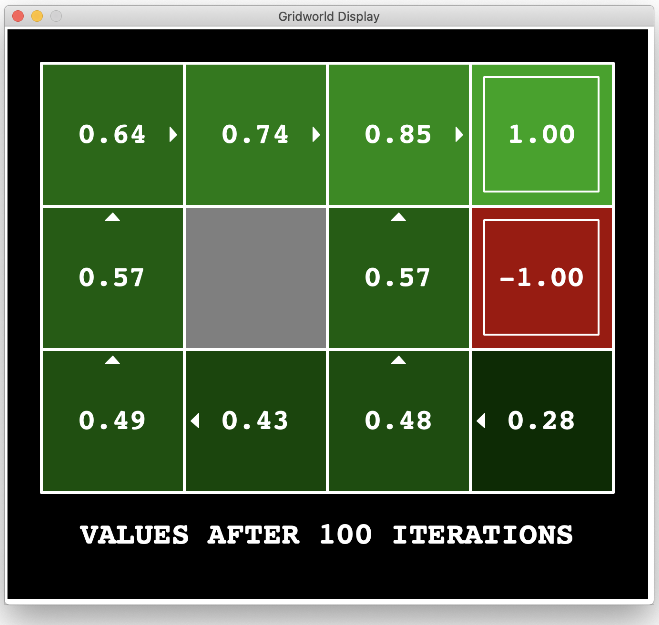
4 Methodology

For Value-iteration agent, we first calculated Q value of each state that it will follow by taking particular action and agent will choose maximum value to make optimal action. The following formula summarises the action.

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

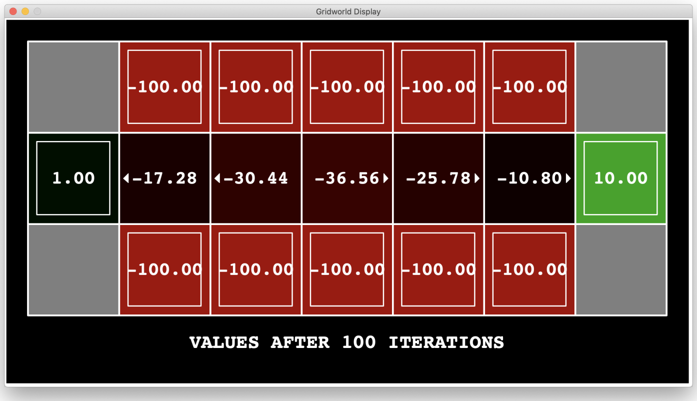
The following image shows the output after value iteration agent has computed policy and computed it 100 times.
Command for the execution :

```
$ python gridworld.py -a value -i 100 -k 10
```



We are testing it on BridgeGrid environment. We changed the default value of discount and noise to 0.9 and 0.2 respectively to change optimal policy that enable the agent to cross bridge. Command for the execution :

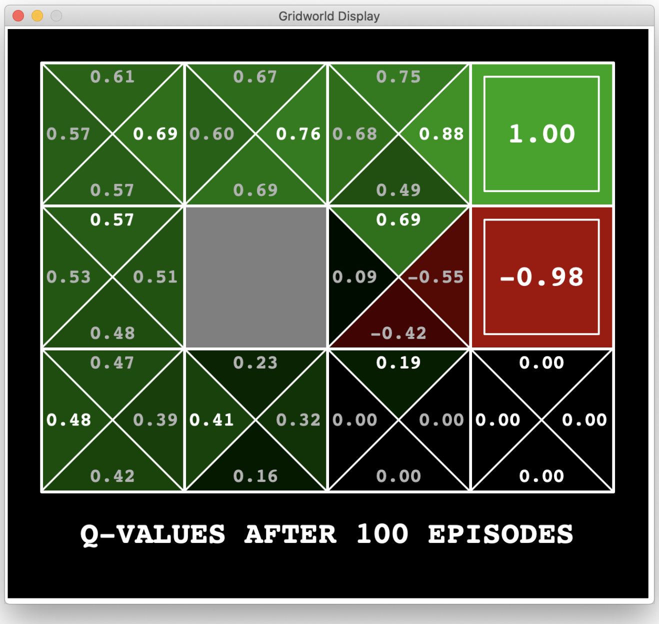
```
$ python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9
--noise 0.2
```



For Q learning agent, first we need to obtain q value and then compute value from given all q Values.Then we proceed to choose optimal action by obtaining max value. We can also make the agent explore MDP by making it choose action randomly. The agent need to update q value by using old q value.

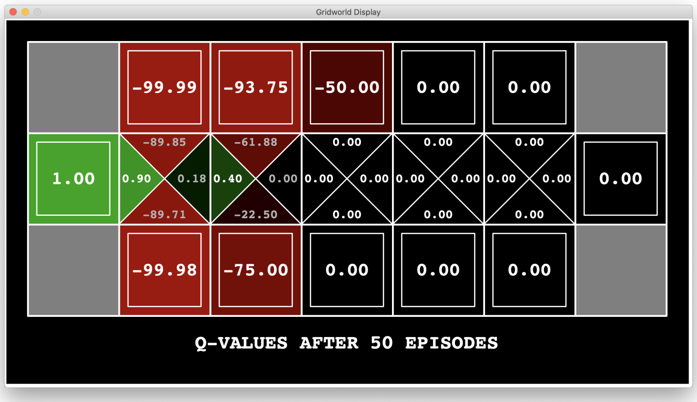
When agent follow random action , it produces following q values after 100 episodes.

```
$ python gridworld.py -a q -k 100
```



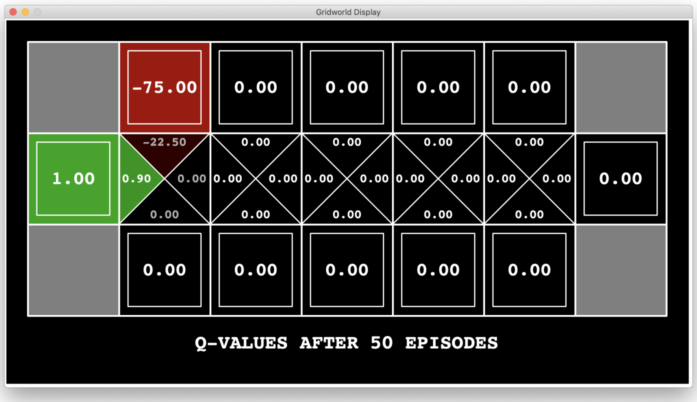
Before pacman enter game, we need to analyse by training a completely random Q-learner with the default learning rate on the noiseless BridgeGrid for 50 episodes and observe whether it finds the optimal policy.The following output with exploration rate of 1.

```
$ python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1
```



With epsilon of 0 , the agent does not explore and always choose optimal path.

```
$ python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 0
```



For approximate q learning agent,we represent state by collection features like number of ghost remaining, distance to ghost,etc.Then we calculate Q value by adding weighted features.

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

The agent will make move by considering fact that similar states have same value.We need to update weighted vector just like q value.

$$\begin{aligned} w_i &\leftarrow w_i + \alpha[correction] f_i(s,a) \\ correction &= (R(s,a) + \gamma V(s')) - Q(s,a) \end{aligned}$$

Smaller difference means smaller change to be made to the Q value.

5 Results

Now, we put our 'smart' Pac-Man agent to test! The scenarios are as following:

1. Train the Pacman agent 2000 times without approximate Q-learning. After the training is done, we test the agent out ten times to determine the winning rate.

```
$ python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l
[layout]
```

2. Train the Pacman agent 50 times with approximate Q-learning. After the training is done, we test the agent out ten times to determine the winning rate.

```
$ python pacman.py -p ApproximateQAgent -a
extractor=SimpleExtractor -x 50 -n 60 -l [layout]
```

Table 1: Winning rate

Layout	With Q learning	With Approx. Q learning
capsuleClassic	0/10 with average score -452.8	4/10 with average score 52.6
contestClassic	0/10 with average score -457.8	8/10 with average score 825.9
mediumClassic	0/10 with average score -359.9	10/10 with average score 1324.9
mediumGrid	1/10 with average score -406.7	10/10 with average score 527.2
minimaxClassic	9/10 with average score 415.6	6/10 with average score 111.3
smallClassic	0/10 with average score -363.7	6/10 with average score 481.8
smallGrid	10/10 with average score 502.2	8/10 with average score 302.6
testClassic	0/10 with average score -473.9	10/10 with average score 560.8
trappedClassic	5/10 with average score -502	7/10 with average score 224.8
trickyClassic	0/10 with average score -447.7	6/10 with average score 936.2

We can see that the former Q-learning technique implemented on the Pac-man agent is not smart enough to beat in the more difficult layout like, capsuleClassic, mediumGrid, etc. Even sometimes, it takes painful amount of time to train the agent. This problem can be solved with the approximate Q-learning. This makes the agent much smarter than previously – even though with only 50 times of training session. There's one weird behavior of the Pac-man that we notice. That is when it eats the Power Pellet, it somehow doesn't know that it can actually eat the ghosts.

6 Conclusion

In this project , we used value iteration for the agent that will choose action that will maximize expected utility.With value iteration , always calculate state's true value until it converges.The state's value is the optimal policy. Q learning agent does not need to have prior information about model of the environment as it learn by process of trial and error.We also implemented approximate q learning agent by using weighted feature of the states. By pacman game, we found that approximate q learning agent has higher rate of winning game than normal q learning agent.

References

[1] UC Berkeley CS188 Introduction to Artificial Intelligence Pac-Man project: `http://ai.berkeley.edu/reinforcement.html`

[2] Csaba Szepesvari. The asymptotic convergence-rate of Q-learning. A *Advances in Neural Information Processing Systems 10, Denver, Colorado, USA, 1997.*