RAMAKRISHNA MISSION VIVEKANANDA
EDUCATIONAL AND RESEARCH INSTITUTE

ARTIFICIAL INTELLIGENCE PROJECT 2

Reinforcement Learning

Authors:
Jimut Bahan Pal
Debadri Chatterjee
Sounak Modak

Supervisor:
Tamal Maharaj

Submitted to the Department of Computer Science in partial fulfilment of the requirements for the degree of M.Sc.

By

Dynammic Duo

October 26, 2019

Contents

List of Figures

Chapter 1

Introduction

Reinforcement Learning (RL) is one of the model free machine learning algorithms where the agent learns its behaviours from the environment by actually interacting with it. This is better than the offline planner because the agent actually interacts with the environment to learn its behaviours because it is almost impossible to simulate a real world in a computer. By using the reinforcement learning, the agent learns those extra features which can only be learned in an real world environment hence giving it a learning capability like living organisms because in a real world there are certain parameters which cannot be simulated by a computer. Since the reinforcement learning agent gets its feedback from the environment, it allows the agent to automatically determine its behaviours that are considered ideal within a specified context. Reinforcement learning is deemed important in the field of artificial intelligence as it starts to make breakthrough and benchmarks in various industrial applications. Previously we have analysed the pacman game where the pacman agent is a reflex agent, here, we are trying to make the pacman agent more smarter by applying RL techniques, i.e, Q-learning successfully. The components of a Reinforcement learning system is shown in Figure 1.1.

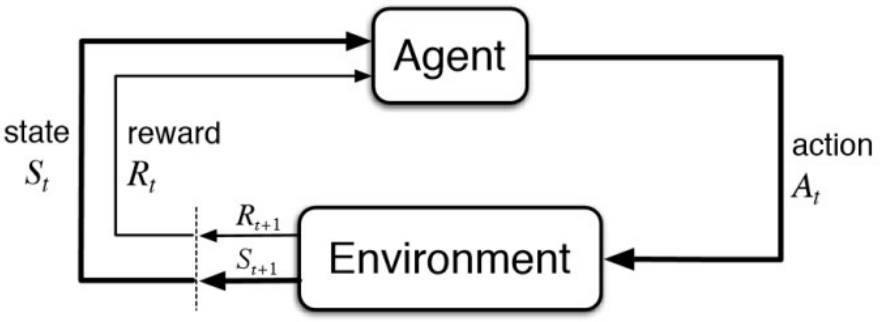

FIGURE 1.1: The general structure of Reinforcement Learning system

We have used the skeleton code from UC Berkeley CS188 Intro to AI [1], which was specially designed for this course.

Chapter 2

Value Iteration

In this part batch version of Value Iteration is implemented. Value iteration is a method for computing optimal MDP policy and value [2]. Value iteration stats from the end and then works backward. Since there is no end to value iteration so we approximate it to some k state when the value converges [3]. Value iteration starts with and arbitrary function V_0 and uses the following equations for computing the k^{th} stage.

$$V_{K+1}(s) = \max_a \sum_{s'} T(s,a,s') \Big[R(s,a,s') + \gamma V(s') \Big] \tag{2.1}$$

The values of states in previous iteration is used for updating values of states in next iteration. The utility of a state is the expected utility of that state sequence encountered when an optimal policy has been executed, starting in that state. The value iteration algorithm for solving MDP's works by iteratively solving the equations relating to the utility of each state to those of its neighbours. The following algorithm is used to calculate the states values of the value iteration in our project.

Algorithm 1 Value Iteration: Learn function $J : \mathcal{X} \to \mathbb{R}$

Require:
Sates $\mathcal{X} = \{1, \ldots, n_x\}$
Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Cost function $g : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Transition probabilities $f_{xy}(a) = \mathbb{P}(y|x,a)$
Discounting factor $\alpha \in (0,1)$, typically $\alpha = 0.9$
procedure VALUEITERATION($\mathcal{X}$, A, g, f, α)
  Initialize $J, J' : \mathcal{X} \to \mathbb{R}_0^+$ arbitrarily
  while J is not converged **do**
    $J' \leftarrow J$
    for $x \in \mathcal{X}$ **do**
      for $a \in A(x)$ **do**
        $Q(x,a) \leftarrow g(x,a) + \alpha \sum_{j=1}^{n_x} f_{xj}(a) \cdot J'(j)$
    for $x \in \mathcal{X}$ **do**
      $J(x) \leftarrow \min_a \{Q(x,a)\}$
  return J

After applying the value iteration on our grid world, we get the following values for each state as shown in Figures 2.1, 2.2, 2.3 and 2.4. We can also see that the policy converges faster than the values.

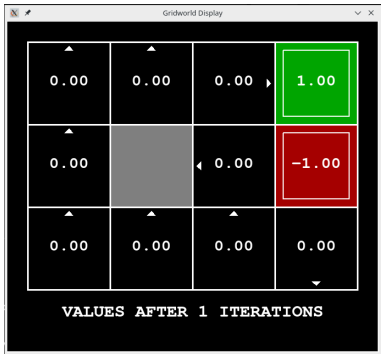

FIGURE 2.1: Values after 1 iterations

FIGURE 2.2: Values after 5 iterations

FIGURE 2.3: Values after 20 iterations

FIGURE 2.4: Values after 100 iterations

Chapter 3

Bridge Crossing Analysis

We have implemented the bridge cross analysis by the help of *BridgeGrid* which is a grid world map with a low reward terminal state and a high reward terminal state separated by a narrow bridge. The agent starts near the low reward state and have to cross the bridge to get to a high reward state. By this process the agents falls into the low reward terminal state several times and learns that - those paths are bad for it and tries new options and chooses the best path, i.e, the path through which the agent gets the highest reward. We have used a default discount of 0.9 and a default noise of 0.2 and by using this the optimal policy does not cross the bridge. We change only one of the discount and noise parameters so that the optimal policy causes the agent to cross the bridge. The noise refers to the randomness of the agent for going into a successor state. The bridge world that we have used and the values for each state and the policies that the agent have learnt is shown in Figure 3.1.

FIGURE 3.1: Values obtained after 100 iterations for the bridge crossing problem

Chapter 4

Policies

There are two terminal states with a positive payoff of +1 and another payoff of +10 as shown in Figure 4.1. The starting state is the yellow square and we distinguish between two types of paths, i.e, one which risks the cliff and travel near the bottom edge of the grid and the one which avoids the cliff and travels along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. These paths are shown by green arrow in the figure. Here we choose the setting of the discount, noise and living reward parameters for this MDP to produce optimal policies. Our agent followed its optimal policy without being subject to any noise and exhibited the given behaviour of choosing the best path.

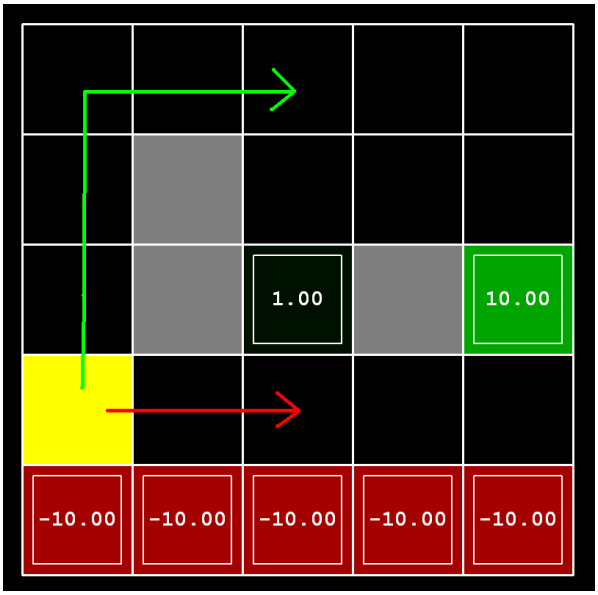

FIGURE 4.1: The policies taken by the agent

The value of and optimal policy can be defined in the following way:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)\Big[R(s,a,s') + \gamma V^*(s')\Big] \tag{4.1}$$

$$V^*(s) = \max_a Q^*(s,a) \tag{4.2}$$

$$\pi^*(s) = arg\max_a Q^*(s,a) \tag{4.3}$$

We define $Q^*(s,a)$ to be the expected value of performing action a in state s and then following the optimal policy. We define $V^*(s)$ to be the expected value of following the optimal policy from state s. $V^*(s)$ is obtained by performing the action that gives the best value in each state, similarly the optimal policy π^* is the one that gives the best value for each state. The algorithm for selecting policies is as follows:

Algorithm 2 Policy Iteration: Learning a policy $\pi : \mathcal{X} \to \mathcal{A}$

Require:
Sates $\mathcal{X} = \{1, \ldots, n_x\}$
Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Cost function $g : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Transition probabilities f, F
$\alpha \in (0, 1)$
procedure POLICYITERATION($\mathcal{X}$, A, g, f, F, α)
  Initialize π arbitrarily
  while π is not converged **do**
    $J \leftarrow$ solve system of linear equations $(I - \alpha \cdot F(\pi)) \cdot J = g(\pi)$
    for $x \in \mathcal{X}$ **do**
      for $a \in A(x)$ **do**
        $Q(x,a) \leftarrow g(x,a) + \alpha \sum_{j=1}^{n_x} f_{xj}(a) \cdot J(j)$
    for $x \in \mathcal{X}$ **do**
      $\pi(x) \leftarrow \arg\min_a \{Q(x,a)\}$
  return π

Here we see that the agent learns the policy of taking the path that "avoids the cliff" as shown by the policies obtained in Figure 4.2 and the Q values of each state learned by the agent as shown in Figure 4.3.

So, we find that the optimal policy is found by the agent, and the agent acts accordingly when placed in any grid of the gridworld, finding its path towards the goal.

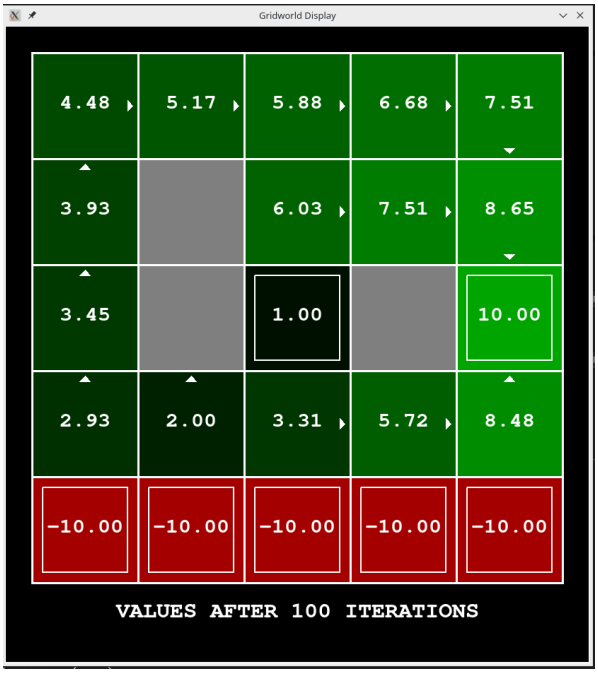

FIGURE 4.2: Values of each state and Policies obtained from the Discount grid layout after 100 iterations

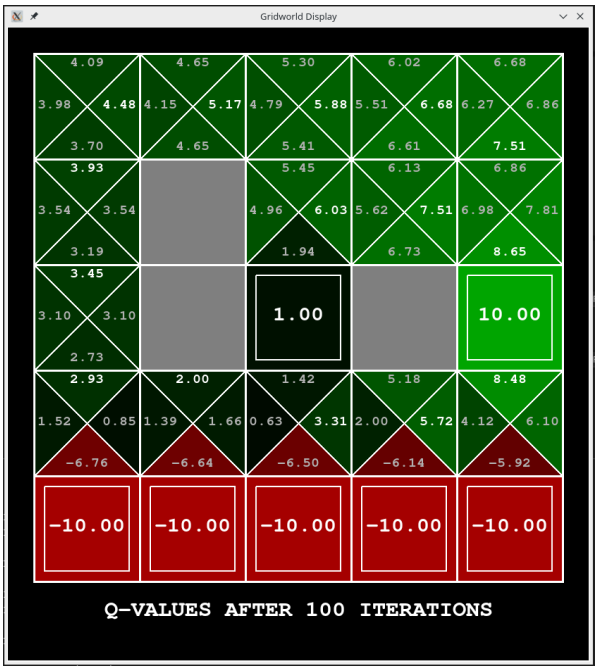

FIGURE 4.3: *Q* values of each state obtained from the Discount grid layout after 100 iterations

Chapter 5

Asynchronous Value Iteration

A common refinement of the value iteration algorithm is the asynchronous value iteration. It updates the state one at a time rather than sweeping through the states to create a new value function and stores the values in a single array. The value iteration can store either the Q(s,a) or V(s) array. It converges faster than value iteration and uses less space forming the basis for some of the algorithms for reinforcement learning. Termination can be difficult to determine as the agent must guarantee a particular error unless the agent is careful about how the action and the states are selected. This procedure often runs indefinitely and is prepared to give its best estimate of optimal action for any state when asked.

Asynchronous value iteration when implemented while storing just the V(s) array carries the following update:

$$V(s) = \max_a \sum_{s'} P(s'|s,a) \Big[R(s,a,s') + \gamma V(s') \Big] \tag{5.1}$$

Though the variant stores less information. It is more difficult to extract the policy. It requires one extra backup to determine which action a results in the maximum value, which is given by the following equation:

$$\pi(s) = arg \max_a \sum_{s'} P(s'|s,a) \Big[R(s,a,s') + \gamma V(s') \Big] \tag{5.2}$$

Our value iteration agent is an offline planner and not a reinforcement learning agent (007) so the relevant training option is the number of iteration of value iteration it should run in its initial planning phase. The agent takes an MDP on construction and runs cyclic value iteration for the specified number of iteration. Our agent is asynchronous value iteration agent because it updates only one state in each iteration as opposed to batch style update. The working of the cyclic value iteration are as follows:

- The first iteration only updates the value of the first state in the states list.
- In the second iteration it only updates the value of the second state
- This procedure continues until the agent has updated the value of each state once and then it starts back from the first state for the subsequent iteration.

Chapter 6

Q-learning

The value iteration agent does not actually learn from experience. It ponders its MDP model to arrive at a complete policy before interacting with the real environment. So, it becomes a reflex agent when it follows the pre-computed policy when interacting with a real environment [4]. This distinction is subtle in a simulated environment like GridWorld but is very important in the real world because the real MDP is not available. The work of the two components of adaptive heuristic critic can be accomplished by Watkin's Q-learning algorithm. Q-learning is typically easier to implement and we can do that by the following equations:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \epsilon S} T(s,a,s') \max_{a'} Q^*(s',a') \tag{6.1}$$

where $Q^*(s,a)$ is the expected discounted reinforcement for taking action a in state s and then choosing the actions optimally. $V^*(s)$ is the value of s after assuming that the best action is taken initially so $V^*(s) = \max_a Q^*(s,a)$. We also see that since $V^*(s) = \max_a Q^*(s,a)$, we have $\pi^*(s) = arg\max_a Q^*(s,a)$ as an optimal policy. The Q-learning equation is:

$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha \left[R(s,a) + \gamma \max_{a'} Q(s',a') - Q_{t-1}(s,a) \right] \tag{6.2}$$

Where $\langle$ s, a, r, s' $\rangle$ is an experienced tuple. The Q values will converge with probability 1 to Q^* as each action is executed in each state an infinite number of times and the α is decayed appropriately. When the Q values nearly converge to their optimal value the agent can act greedily by selecting the highest Q value of a particular state. There is a difficult exploitation vs exploration trade-off during learning. The details of the exploration strategy will not effect the convergence of the learning algorithm, so Q-learning is the most popular and seems to be the most efficient model free algorithm for learning for delayed reinforcement. So, it may converge quite slowly to a good policy. We have implemented a Q-learning agent which did very little on construction but instead learned by trial and error from interaction with the environment. The algorithm for Q-learning is as follows:

Algorithm 3 Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
Sates $\mathcal{X} = \{1, \ldots, n_x\}$
Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
Discounting factor $\gamma \in [0, 1]$
procedure QLEARNING($\mathcal{X}$, A, R, T, α, γ)
  Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
  while Q is not converged **do**
    Start in state $s \in \mathcal{X}$
    while s is not terminal **do**
      Calculate π according to Q and exploration strategy (e.g. $\pi(x) \leftarrow_a Q(x, a)$)
      $a \leftarrow \pi(s)$
      $r \leftarrow R(s, a)$ ▷ Receive the reward
      $s' \leftarrow T(s, a)$ ▷ Receive the new state
      $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
      $s \leftarrow s'$
  return Q

The Q vaules obtained for each state after the 100 iterations in grid world is shown in Figure 6.1. Similarly the Q values obtained for the "bridge crossing" problem after 100 iterations is shown in Figure 6.2.

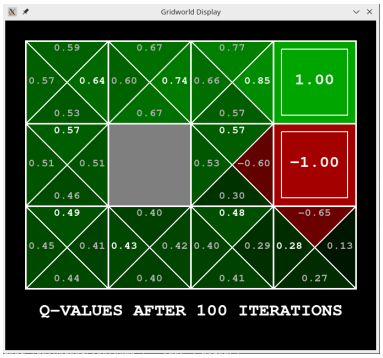

FIGURE 6.1: Q values of each state obtained from the grid world after 100 iterations

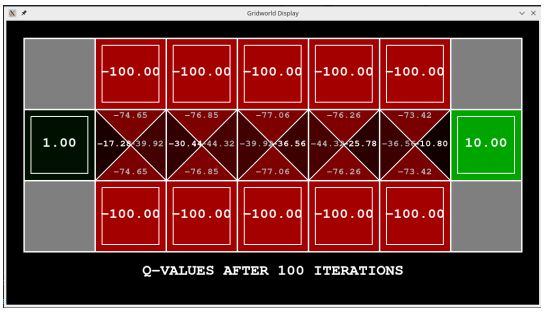

FIGURE 6.2: Q values of each state obtained from the grid world after 100 iterations

Chapter 7

ϵ-Greedy

To balance exploration and exploitation trade off we need ϵ Greedy strategy. The algorithm after a certain period of exploration exploits the best option k greedily ϵ% of the time. For example if we set ϵ as 0.04 then the algorithm will exploit the best option k, 96% of the time, exploring random alternatives 4% of the time. This is actually quite effective but there is a drawback i.e., the agent can sometimes under explore the variant space before exploiting the strongest variant k. This drawback makes the ϵ-Greedy algorithm prone to getting stuck while exploiting a sub-optimal variant.

We have built our Q-learning agent by implementing ϵ-greedy action using getAction method. It chooses random action an ϵ fraction of the time and follows [5] its current best Q-values otherwise. We have found that choosing a random action may result in choosing the best action which may not be found by choosing a sub-optimal action, rather can be found by choosing a random legal action. We have simulated a binary variable having probability p of success which returns true and false just like a coin flip. Our final Q-values resembled those of the value iteration agent specially along optimal paths. However our average returns was lower than the Q-values predicted due to the random actions taken in the initial learning phase. We have also implemented a crawler robot using our Q-learning class and we have tweaked certain learning parameters to study the agents policies and actions as shown in Figure 7.1. We have noticed that the step delay is a parameter of the simulation whereas the learning rate and ϵ are parameters of our learning algorithm, whereas discount factor is the property of the environment.

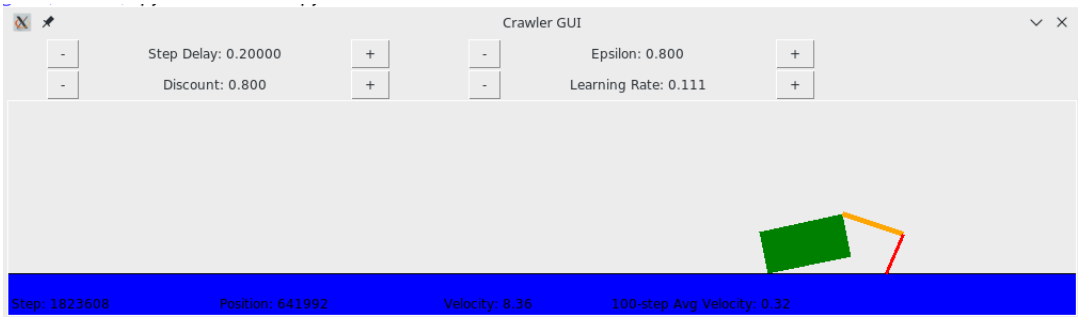

FIGURE 7.1: The crawler robot in action

Chapter 8

Bridge Crossing Revisited

We have trained a completely random Q-learner with a default learning rate on the noiseless BridgeGrid for 50 episodes and observed whether it finds an optimal policy. We have noticed that the learning agent finds the optimal path for one crosssing and couldn't find the other part of the bridge because it was busy exploring and not exploiting what it has learned as shown in Figure 8.1. Now we have tried the same experiment with an ϵ of 0 and noticed that after finding an optimal path, it only follows that path reflexively and does not explore any other path as shown in Figure 8.2.

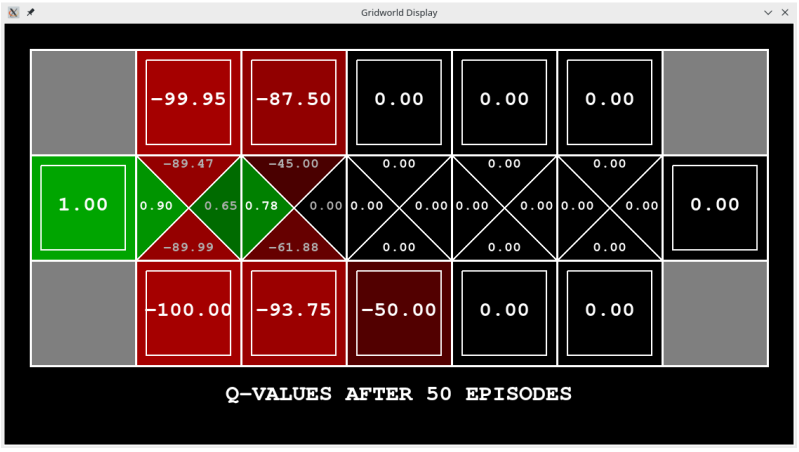

FIGURE 8.1: After training a random Q-learner with default learning rate on the noiseless BridgeGrid for 50 episodes and epsilon of 1

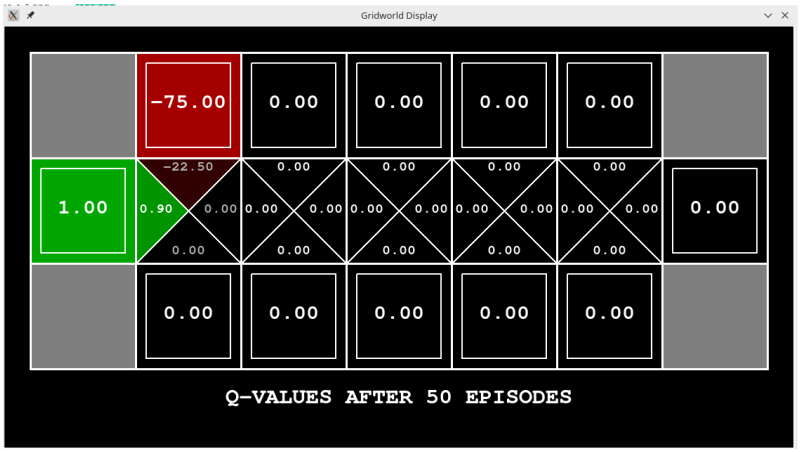

FIGURE 8.2: After training a random Q-learner with default learning rate on the noiseless BridgeGrid for 50 episodes and epsilon of 0

Chapter 9

Q-Learning And Pacman

We have trained a pacman which began to learn about the Q-values of positions and actions and their consequences. It takes a long time to learn the accurate Q-values even for small grids. The pacman's training mode runs without GUI so that the pacman agent can learn as soon as possible [6]. Once the pacman's training is complete, it enters into the testing mode. When testing, pacmans' exploration policies and learning rate will be set to 0 which effectively stops Q-learning and disables the exploration so that the pacman is able to exploit its learned policies. The testing games are shown in the GUI by default and we found that the pacman avoids the ghost and successfully eats the cakes which results in its winning every game and celebrating. The default learning parameters which were effective for this problem are ϵ=0.05, α=0.2, γ=0.8. Our Q-learning agent works for GridWorld, Crawler and pacman since it learns good policy for every world because our agent considers unseen actions and every case properly. Unseen actions have a Q-value of 0. If all the actions that have been seen, have negative Q-values then an unseen action may be optimal. We have played a total of 2010 games of which the first 2000 games were not displayed since they were used in training the pacman and the last 10 games were successfully won by the pacman which shows that the pacman learned the optimal policies for each state in the pacman world as shown in Figure 9.1.

FIGURE 9.1: Pacman as a *Q*-learning agent, successfully avoiding ghosts and eating food pallets

During training we see an output of the performance of pacman after every 100

games. ϵ is kept positive during training so the pacman learns a good policy but still tries to explore new policy, so it results in its poor performance. This is because it occasionally makes a random exploratory move into a ghost. As a benchmark, it should take between 1000 and 1400 games before pacmans reward for a 100 episode segment becomes positive, which reflects that it started winning more than losing. By the end of the training pacman rewards should remain positive and fairly high.

The MDP state is the exact board configuration facing pacman, with the now complex transition describing an entire ply of change to that state. The intermediate game configuration in which pacman has moved but the ghost has not replied are not MDP states but are bundled into the transition. We have seen that once the pacman has completed its training it should win in the test game 90% of the times since its exploiting its learned policy as shown in Figure 9.1.

.

However when we have trained the agent on a seemingly simple mediumGrid as shown in Figure 9.2, it does not perform well. In our implementation the pacman's average training reward remains negative throughout training. At test time it looses its games because it couldnot explore all the states. Training also took long time despite its ineffectiveness. The pacman fails to win on a larger layout because each board configuration is a separate state with separate state values. It has no way to generalise that running into a ghost is bad for all positions and this approach doesn't result is scaling so this is a bad approach for solving pacman using Q-learning for each state.

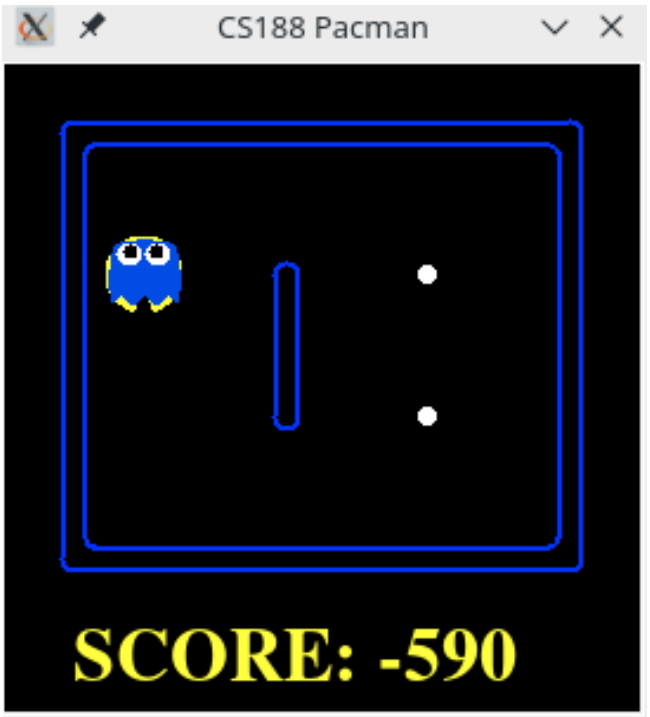

FIGURE 9.2: Pacman as a *Q*-learning agent on a medium grid, unsuccessful at eating food pallets

Chapter 10

Approximate Q-Learning

We have implemented an approximate Q-learning agent that learns weights for features of states where many state may share the same feature. Approximate Q-learning assumes the existence of a feature function $f(s,a)$ over state and action pairs which yields a vector $f_1(s,a)..f_i(s,a)..f_n(s,a)$ of feature values. Feature vectors are like a dictionary of objects containing the non zero pair of features and values, where omitted features have a value of zero.

The approximate Q-learning function takes the following form:

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i \tag{10.1}$$

where each weight w_i is associated with a particular feature $f_i(s,a)$. We implemented the weight vectors as dictionary mapping features to weight value which were returned by the feature extractors. We updated weight vectors similar to how we updated Q-values by applying the following equations:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a) \tag{10.2}$$

$$difference = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a) \tag{10.3}$$

The $difference$ term is the same as used in normal Q-learning whereas ours is the experience reward. By default approximate Q-agent uses the *IdentityExtractor* which assigns a single feature to every $\langle$ *state*, *action* $\rangle$ pair. With this feature extractor, our approximate Q-learning agent worked identically to PacmanQAgent as shown in Figure 10.1.

Approximate Q-agent shares several methods which are common to Q-learning agent because it is a subclass of the later. We have implemented a Q-learning agent to call the getQValue instead of accessing Q-values directly, so that when we override getQValue in our agent, the new approximate Q-values are useful in computing actions. We have also trained our approximate Q-agent is a large layout which took few minutes to train and our agent won almost every time with these simple features, even with only 50 training games as shown in Figure 10.2.

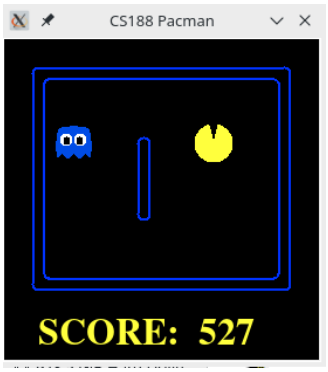

FIGURE 10.1: Pacman as an approximate *Q*-learning agent, successfully avoiding ghosts and eating food pallets in medium grid

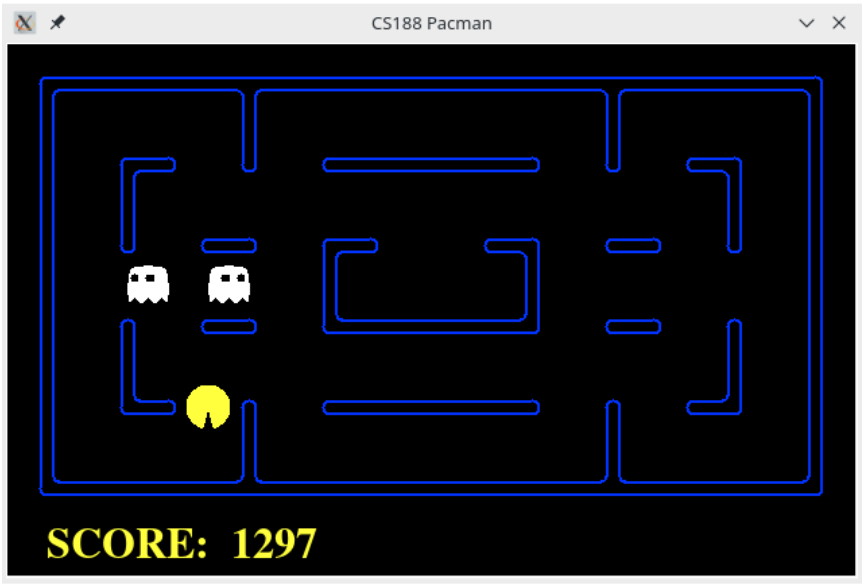

FIGURE 10.2: Pacman as an approximate *Q*-learning agent, successfully avoiding ghosts and eating food pallets even for large and complex maze

Chapter 11

Conclusion

In this project we used value iteration for the agent that will choose the action which will maximise its expected utility. We have noticed that when we use value iteration, the policy converges faster than the values so, we need to maintain a threshold after which we need to stop since the values will change very minutely without significantly affecting the policies of a given *GridWorld*. The values obtained will decide the optimal policy which will decide the agents action when it is spawned in a given state in the GridWorld. Then we implemented Bridge Crossing Analysis in which an agent has to cross the bridge by exploring and exploiting the learned values of each state. We have noticed that when the agent explores more, it finds certain paths which are not good or optimal. So, exploiting the given policies the agent learns the optimal or best path over time.

We have implemented asynchronous value iteration along with Q-learning and noticed that Q-learning agent does not have to have prior information about the environment as it learns by process of trial and error. We implemented the approximate Q-learning agent by using weighted features of states. By implementing the approximate Q-learning agent on the crawler, we found that the crawler learns about the environment and moves from left to right in a two dimensional world. We also implemented the pacman game using approximate Q-learning agent which had a higher rate of winning the game than the normal Q-learning agent. We found that the approximate Q-learning agent is practically more useful than normal Q-learning agent for any state space.

Bibliography

[1] DeNero, J., Klein, D., Abbeel, P. (2013). The Pac-Man Projects. UC Berkeley CS188 Intro to AI – Course Materials. available online at *http : //ai.berkeley.edu/project*$_o$*verview.html*, last accessed on 26th October, 2019.

[2] Russell, S., Norvig, P. (2010). Artificial Intelligence: A Modern Approach. Upper Saddle River, NJ: Prentice Hall.

[3] Kaelbling, P., K. (1996). Journal of Artificial Intelligence Research . Enhancement to Value Iteration and Policy Iteration. Available online at https://www.cs.cmu.edu/afs/cs/project/jair/pub/, last accessed on 26th October, 2019.

[4] Pal, S. (2019, May 15). An Introduction to Q-learning: Reinforcement Learning. Floydhub Blogs. Available online at https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/, last accessed on 26th October, 2019.

[5] Sita, C., Calvinthio, G. (2017). COMP3211 Final Project Report: Pacman with Reinforcement Learning. Available online at https://www.cs.cmu.edu/afs/cs/project/jair/pub/, last accessed on 26th October, 2019.

[6] Pal, J., B. (2019, August 29). DESIGNING OF SEARCH AGENTS USING PAC-MAN. Available online at https://doi.org/10.31219/osf.io/rnsy6, last accessed on 26th October, 2019.