Reinforcement Learning II

Addison Phelps

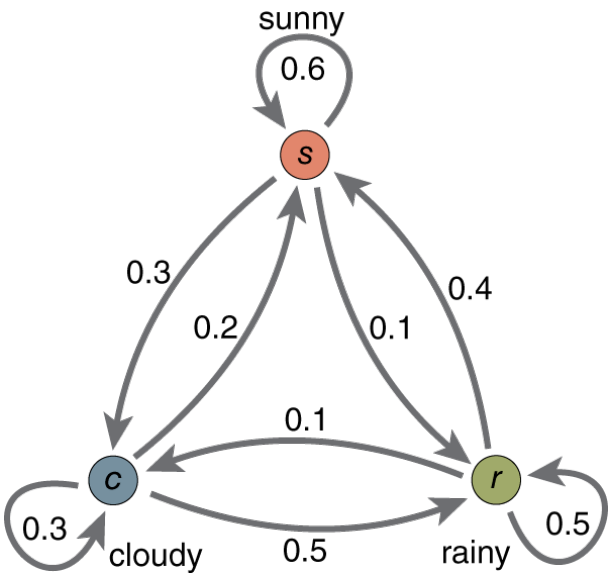
May 8, 2018

1 Introduction

While Reinforcement Learning has been around since the 1950s, it has gained headlines in recent years with the developments made by DeepMind. DeepMind developed software able to play any Atari game and AlphaGo which defeated the Go champion Ke Jie in Go.

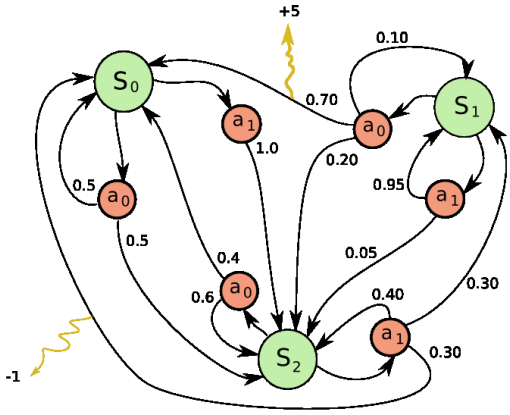
1.1 RL Review

Reinforcement Learning is based upon the software taking note of its environment and making decisions accordingly based on it. Based on the actions the system makes, it receives positive or negative rewards. Its goal is to optimize its total long term rewards usually by trial and error until it maximizes its positive reward. How the software makes it actions is called its policy. The policy can be almost any algorithm. Some policy functions involve a certain amount of randomness. Say for example you were to program Pac-Man to move by itself and try to run from the ghosts. At any intersection the probability that it changes its line of motion is p and which direction it turns is based on another probability q. These two probabilities would be called the policy parameters. One way to find the optimal policy parameters is to simply try many, many different combinations and use which ever one works best. However, this takes way too much time and is inefficient. The process by which the policy parameters are determined is called the policy search. One method to search for good policy parameter is using genetic algorithms. Another process to determine a good policy is using Q-Values which I will talk about later.



1.2 Markov Decision Processes

At the basis of Markov Decision Processes are Markov Chains. Developed by Andrey Markov, Markov Chains are a mathematical process with no memory that describe movement of a system from state to state with a certain amount of randomness. If the current state is defined as s and the next state is defined as s2. With a certain probability s will evolve to s2, then this process will repeat with s2 and s3 until the Markov Chain reaches its terminal state, where it does not evolve from its current state. A Markov Decision Process (MDP) utilizes this underlying process except at each transition between states the system has a decision to make depending on the current state. MDPs have 4 primary components. The first being the states that the system transitions between defined as S. The second being the set of actions the system makes in transitions between states defined as A. The next is are the transition probabilities which will be defined at P. The final component is the reward function defined as R. One more component to keep in mind is γ. γ is the discount applied to the reward that is anywhere from 0 to 1. What does this mean? The discount is used to evaluate how much we care about the reward at each state. A discount close to 0 means that we prefer a reward in the short term (called myopic) and a discount of 1 means we prefer a reward in the long term (called far-sighted). Moving between states yields a reward for the system and the system's goal is to maximize that reward in the long term. As I said earlier, a policy characterizes the actions the system can make transitioning between states. Policies are defined as π. Since the system is trying to maximize is long term rewards, is there an approach we can use to gain it?



1.3 Dynamic Programming

There is an approach we can use to maximize long term rewards called the Bellman Optimality Equation. This equation combines all of the components discussed in MDPs. This approach makes use of dynamic programming to solve the MDPs. In short, if we can solve (maximize the rewards) from a certain state onward, we can simply store that instead of recomputing it when we solve the MDP from our starting state. This is incredibly helpful because when MDPs are too large, it is extremely slow and space inefficient to recompute the values of each state. Our long term cumulative rewards are characterized by the value function for a given policy.

$$V^*(s) = max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma.V^*(s')] \tag{1}$$

T(s, a, s') is our transition probability if the agent chose a as its action. R(s, a, s') is our reward from the agent choosing action a. As I said earlier γ is the discount. Why would we need a discount? Remember that by choosing action a, it is not guaranteed that the agent actually transitions between state s and s' and gets the reward. When we focus on reaching a reward far in the future, the uncertainty that we actually get that reward increases. Now using this equation, we can estimate the maximum reward the agent can gain from state s assuming that it acts optimally. To implement this, you first set all the state value to 0 then update them using the Value-Iteration Algorithm (its very similar to the Bellman Optimality Equation).

$$V_{k+1} = max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma.V_k(s')] \tag{2}$$

Now using these, we yield the optimal state values, but there is one problem: this algorithm does not tell us how to gain the optimal state values.

2 Q-Values

Fortunately, Bellman created a method on which we can base our actions to act optimally called state-action values or Q-values. The Q-value for any state-action (defined as Q*(s, a)) is the sum of the rewards that the agent can expect to receive if it makes action a at state s (note that these rewards are discounted) this is only if the agent acts optimally after choosing action a. Again to do this, we must first set all the Q-values for every state-action pair to 0, and then implement the Q-Value Iteration Algorithm.

$$Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma.max'_a Q_k(s',a')] \tag{3}$$

After finding the optimal Q-values, making an optimal policy (π*(s)) is easy. In every state the agent simply chooses the action with the highest Q-value.

3 Temporal Difference Learning and Q-learning

Q-Value Iteration and the Bellman Optimality Equation apply when the agent has full knowledge of its environment, but sometimes in Markov Decision Processes, the agent does not know exactly what the transition probabilities are or the rewards for that matter. In order to figure those out, it much visit each state and learn those parameters often multiple times. The algorithm for Temporal Difference Learning is quite similar to the Value-Iteration except for the fact that again the agent only partially knows the environment (only the states and the actions). In order to visit each state, the agent uses an exploration policy and at each step the TD algorithm updates the estimated values based on the agents observations (essentially the rewards and transition probabilities.

$$V_{k+1}(s) = (1-\alpha)V_k(s) + \alpha(r + \gamma.V_k(s')) \tag{4}$$

(α is the learning rate)

At each state the agent stores the average rewards it gets just from transitioning from that state and the average rewards it can expect in the future. Another method of doing this is the Q-Learning algorithm. Again the actual Q-learning algorithm is quite similar to the Q-Value Iteration algorithm except again in this case the agent does not actually know the rewards and transition probabilities at each step.

$$Q_{k+1}(s,a) = (1-\alpha)Q_k(s,a) + (r + \gamma \cdot max_(a')Q_k(s',a')) \tag{5}$$

The agent in this case stores the average rewards from transitioning from state s after making action a and the rewards it can expect later. After running this algorithm enough times, it would eventually converge to the optimal Q-value estimates and we would take the maximum Q-Values because they specify a policy which the agent can use to act optimally.

4 Exploration Policies

Q-Learning only works if the agent visits each state enough to make accurate estimates of the Q-values at each state. Using a random policy, in which the agent simply hops around each state is often used, but there are two problems with this. The first being that by moving randomly there is no guarantee that it visits each state. The second is that the point at which we are almost certain that the agent has visited each state a sufficient number of times often can take a very, very long time. In order to improve the time efficiency and yield better results, we can implement the ϵ-greedy policy. Essentially what the policy does is with a probability 1-ϵ it chooses the action with the highest Q-value (or acts greedily), and with a probability ϵ it acts randomly to explore new parts of the MDP. The advantage of this is not only is this more efficient than a simply random policy, but it also alternates between exploiting and exploring the MDP. Usually, we use a high ϵ value then decrease it as we explore more of the MBP because there are less unknown regions. One other way to visit each state is to add a bonus to the Q-value estimates to incentivize the agent to chose actions it has not chosen before.

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma.max_(\alpha')f(Q(s',a'),N(s',a'))) \quad (6)$$

N(s', a') is the conunter from how many times the agent choses action a upon reaching state s f(q, n) is the exploration function (f(q, n) = q + K/(1+n)). K is the curiosity value, it dictates how much the agent is willing to explore unknown regions of the MDP.

5 Approximate Q-Learning and Deep Q-learning

One of the major issues with Q-Learning is that it is still not efficient on very large, complex MDPs. Our solution is to approximate the Q-values with a function Q_θ(s, a). Instead of using every available parameter, we use a more manageable amount defined by the vector θ. How do we decide on which parameters to use? DeepMind found that using a Deep Neural Network (DNN) works nicely on complex problems. When a DNN is used to approximate Q-Values it's called a Deep Q-Network (DQN) and using a DQN for Q-learning is called Deep Q-Learning.

How do we implement the DQN? We simply calculate the target Q-Value which is based upon the fact that we wish to make the approximated Q-Value as close to the reward received from making action a plus the discounted reward from future actions. We then run the DQN for future actions and pick the one with the highest approximate Q-value, discount it and sum it with the reward r which yields the target Q-value.

$$y(s,a) = r + \gamma.max_(a')Q_(\theta)(s',a') \quad (7)$$

Using this we try to minimize the difference between our estimated Q-values and our target Q-value.

6 Conclusion

Reinforcement Learning is a very powerful tool, but it's main limitation being that it is incredibly computationally expensive. However, it has lead to great products and applications, and I hope this lecture helps you make use of this incredibly amazing technique.