Survey on Reinforcement Learning Techniques

Siddhi Desai[*], Kavita Joshi[**], Bhavik Desai[**]

[*] Asst. Prof, Electronics & Communication Dept., Laxmi Institute Of Technology, Sarigam
[**] Asst. Prof. Computer Department, Laxmi Institute of technology, Sarigam
**Asst. Prof. Mechanical Department, Laxmi Institute of technology, Sarigam

Abstract- There are many unsolved problems that computers could solve if the appropriate software existed. Many real time problems are currently unsolvable such as Flight control systems for aircraft, automated manufacturing systems, and sophisticated avionics systems, not because current computers are too slow or have too little memory, but simply because it is too difficult to determine what the program should do. If a computer could learn to solve the problems through trial and error, that would be of great practical value. Reinforcement Learning is an approach to machine intelligence that combines two disciplines: Dynamic Programming and supervised learning to successfully solve problems that neither discipline can address individually. Encouraged by this emerging technique, this document briefly reviews the basic study of reinforcement learning, its various techniques and applications of it in various fields.

Index Terms- Dynamic Programming, Monte carlo, Temporal difference

I. Introduction

Reinforcement learning is one of the more recent fields in artificial intelligence. Reinforcement learning is an area of machine learning in which agent (learner) learns by interaction with the environment [2]. In this, reward signal is provided to the agents so it can understand and update performance accordingly. While there are a few different types of learning, reinforcement learning normally helps adjust our physical actions and motor skills. The actions an organism perform result in feedback, which in turn is translated into a negative or positive reward for that action. Reinforcement learning is learning what to do--how to map situations to actions--so as to maximize a numerical reward signal [1]. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them [1]. Reinforcement learning is used in areas like Robot navigation, Adaptive control, combinatorial optimization, Game playing, Computational Neuroscience [4].

In Section I, basics related to reinforcement learning is introduced. In section II, various techniques of reinforcement learning with its drawback is explained. Tic -Tac

II.REINFORCEMENT LEARNING TECHNIQUES

There are many different techniques for solving problems using reinforcement learning.

A. Dynamic Programming (DP)

Dynamic programming is a technique for computing optimal policies. An optimal policy can, by definition, be used to maximize reward, so DP can be very useful under the right circumstances.

Drawbacks of Dynamic Programming
 It requires a perfect model of the environment
 It requires a considerable amount of computation.

B. Policy Improvement:

One of the most basic DP methods is to compute the exact value function for a given policy, and then use that value function to produce a new policy. If we assume that we have a good value function that assigns a value to each state, a sensible policy is often to simply move to the adjacent state with the highest value. ("Adjacent state" in this sense is defined as any state that can be reached with a single action.) In this way, the policy can be updated using the improved value function. This technique is called policy improvement.

Drawback of Policy Improvement:

- The problem with policy improvement is that computing the exact value function can take a considerably long time, since it requires iterating many times over every possible state.
- Even simple problems can have environments with a number of states so large that such iteration is realistically impossible.

C. Monte Carlo (MC)

Monte Carlo techniques, on the other hand, require no knowledge of the environment at all. They are instead based on accumulated experience with the problem. As the name might suggest, MC is often used to solve problems, such as gambling games that have large numbers of random elements.
Like DP, MC centres on learning the value function so that the policy can be improved. The simplest way of doing this is to average each reward that a given state (or state-action pair, if that is the type of value function being used) results in. For example, in a game of poker, there are a finite number of states (based on the perceptions of the player) that exist. An MC technique would be to keep track of the rewards received after each state, and then make the value of each state equal to the average of all the rewards (money won or lost) encountered following that state (in that particular game). Assuming an "agent has Royal Flush" state had been encountered at all, the value for that state would

probably be very high. On the other hand, an "agent has a worthless hand" state would probably have a very low value. Obviously, accumulating useful value data for states requires many repeat plays of the game.

Drawback of Monte Carlo:

- Since MC learns with experience, many repetittions of problems are required.
- It is possible to "solved" problems using MC by exploring every possibility and then generating an optimal policy. However, this can take a long time, (the number of variables in a human poker game make the number of states huge) and for many problems (like black jack), the randomness is so great that, a "solution" doesn't result in winning even half the time.

D. Temporal Difference (TD)

One of the problems common to both dynamic programming and Monte Carlo is that the two techniques often don't produce information that's useful until a huge number of possible states have been encountered multiple times. It is possible to get over this problem with some DP methods by localizing updates, but in that case, the problem remains that DP requires a perfect model of the environment.

One solution to these problems lies in the method of temporal difference (TD), which combines many of the elements of DP and MC.

Like MC, TD uses experience to update an estimate of the value function over time. Like MC, after a visit to a state or state-action pair, TD will update the value function based on what happened. However, MC only updates after the run-through of the problem, or episode, has been completed. It is at that point that MC goes back and updates the value averages for all the states visited, based on the reward received at the end of the episode.

TD (0):

TD, on the other hand, updates after every single step taken. The general methodology for basic TD, sometimes called TD (0), is to choose an action based on the policy, and then update the value of the current state based on the sum of the reward given by the following state and the difference in values between the current and following state. This sum is often multiplied times a constant called a step-size parameter. This technique of updating to new estimates based partly on current estimates is called bootstrapping.

TD works well because it allows the agent to explore the environment and modify its value function while it's working on the current problem. This means that it can be a much better choice than MC is for problems that have a large number of steps in a given episode, since MC only updates after the episode is completed. Also, if the policy depends partly on the value function, the behaviour of the agent should become more effective at maximizing reward as updating continues. This is called using TD for control (as opposed to simply predicting what future value), and there are a number of well-known algorithms, such as Sarsa and Q-Learning, that do it.

TD is often used with state-action pair values rather than simply state values. Since the value of a given state-action pair is an estimation of the value of the next state, TD is considered to predict the next value.

TD (λ):

TD (0) predicts ahead one step. There is a more generalized form of TD prediction called n-Step TD Prediction, characterized by TD (λ). This uses a mechanism called an eligibility trace that keeps track of which states (or state-action pairs) leading upto the current state are responsible for the current state, and then updates the values of those states to reflect the extent to which they made a difference. As Sutton & Barton (1998) point out, the generalized MC method can be considered a form of TD(λ) that tracks the entire sequence of actions and then updates all the visited states (without using a discount factor) once a reward has been reached. MC, in other words, can be considered a form of TD that is on the opposite end of the spectrum from TD (0). TD (0) only predicts one state, but MC "predicts" every state (though in this sense "prediction" refers to learning about past events).

Using TD (λ) in this way results in some of the same problems that MC has, in that some information isn't learned about a state until well after that state has been encountered. However, if multiple episodes are expected in the same environment, the information learned during one episode will become useful in the next episode. Also, if the same state is visited twice, the information will be immediately useful. [2]

IV. Tic-Tac-Toe Game

Consider the familiar child's game of tic-tac-toe. Two players take turns playing on a three-by-three board. One player plays Xs and the other Os until one player wins by placing three marks in a row, horizontally, vertically, or diagonally, as the X player has in this game:

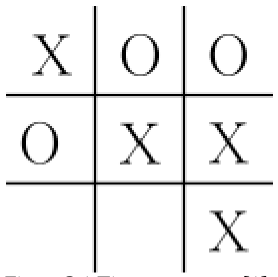

Figure 3.1 Tic-tac-toe game[1]

If the board fills up with neither player getting three in a row, the game is a draw. Because a skilled player can play so as never to lose, let us assume that we are playing against an imperfect player, one whose play is sometimes incorrect and allows us to win. For the moment, in fact, let us consider draws and losses to be equally bad for us. Here is how the tic-tac-toe problem would be approached using reinforcement learning and approximate value functions. First we set up a table of numbers, one for each possible state of the game. Each number will be the latest estimate of the probability of our winning from that state. We treat this estimate as the state's value, and the whole table is the learned value function. State A has higher value than state B, or is considered "better" than state B, if the current estimate of the probability of our winning from A is higher than it is from B. Assuming we always play X s, then for all states with three Xs in a row the probability of winning is 1, because we have already won. Similarly, for all states with three Os in a row, or that are "filled up," the correct probability is 0, as we cannot win from them. We set the initial values of all the other states to 0.5, representing a guess that we have a 50% chance of winning. We play many games against the opponent. To select our moves we examine the states that would result from each of our possible moves (one for each blank space on the board) and look up their current values in the table. Most of the time we move greedily, selecting the move that leads to the state with greatest value, that is, with the highest estimated probability of winning. Occasionally, however, we select randomly from among the other moves instead. These are called exploratory moves because they cause us to experience states that we might otherwise never see. A sequence of moves made and considered during a game can be diagrammed as in Figure 3.2.

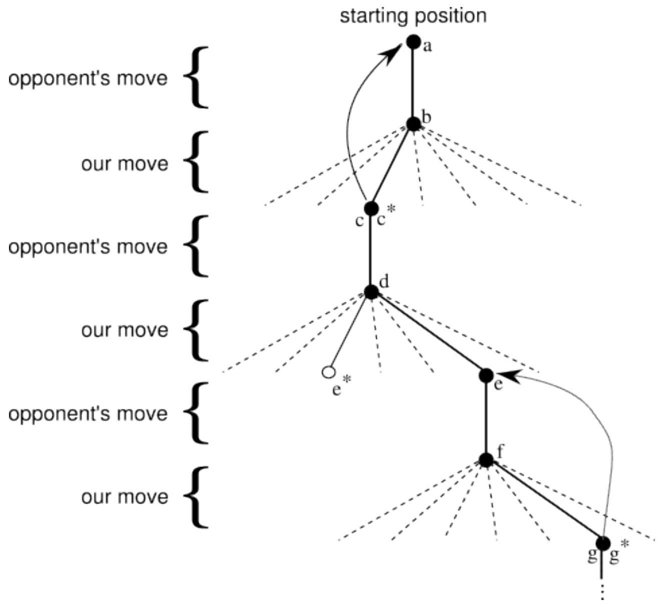

Figure 3.2 A sequence of tic-tac-toe moves [1].

Here, The solid lines represent the moves taken during a game; the dashed lines represent moves that we (our reinforcement learning player) considered but did not make.

While we are playing, we change the values of the states in which we find ourselves during the game. We attempt to make them more accurate estimates of the probabilities of winning. To do this, we "back up" the value of the state after each greedy move to the state before the move, as suggested by the arrows in Figure 3.2 More precisely; the current value of the earlier state is adjusted to be closer to the value of the later state. This can be done by moving the earlier state's value a fraction of the way toward the value of the later state. If we let s denote the state before the greedy move, and s' the state after the move, then the update to the estimated value of s, denoted $v(s)$, can be written as

$$V(s) \leftarrow V(s) + \alpha[V(s') - V(s)]$$
(3.1)

Where α is a small positive fraction called the step-size parameter, which influences the rate of learning. This update rule is an example of a temporal-difference learning method, so called because its changes are based on a difference, $V(s') - V(s)$, between estimates at two different times.

The method described above performs quite well on this task. For example, if the step-size parameter is reduced properly over time, this method converges, for any fixed opponent, to the true probabilities of winning from each state given optimal play by our player. Furthermore, the moves then taken (except on exploratory moves) are in fact the optimal moves against the opponent. In other words, the method converges to an optimal policy for playing the game. If the step-size parameter is not reduced all the way to zero over time, then this player also plays well against opponents that slowly change their way of playing.

Although tic-tac-toe is a two-person game, reinforcement learning also applies in the case in which there is no external adversary, that is, in the case of a "game against nature." Reinforcement learning also is not restricted to problems in which behaviour breaks down into separate episodes, like the separate games of tic-tac-toe, with reward only at the end of each episode. It is just as applicable when behaviour continues indefinitely and when rewards of various magnitudes can be received at any time [1].

CONCLUSION

The Reinforcement learning can efficiently solve many real time applications. Unlike supervised learning, reinforcement learning systems do not require explicit input-output pairs for training. It can be efficiently used in various applications like robot navigation, Game playing and for dynamic allocation of channels. Reinforcement learning is an extension of classical dynamic programming in that it greatly enlarges the set of problems that can practically be solved.

REFERENCES

[1] Sutton, R.S. and Barto, A. G. Reinforcement Learning: An Introduction. MIT Press. 1998.
[2] Albert Robinson.CS242FinalProject: Reinforcement Learning. May 7, 2002.pp: 2-6
[3] Mance E. Harmon and Stephanie S. Harmon. Reinforcement Learning: A Tutorial. pp:2
[4] B. Ravindran. Reinforcement Learning: Learning from Interaction, Winter School on Machine Learning and Vision, 2010. www.findthatpowerpoint.com/search-31849170hDOC/download-documents ravindran-tutorial.ppt.htm
[5] S. Rajasekaran and G.A. Vijayalakshmi Pai. Neural Networks, Fuzzy Logic and Generic Algorithms synthesis and applications, PHI Learning private limited,Newdelhi,2011.pp:19

AUTHORS

First Author – Siddhi Desai, M.Tech E&C.(Gold medalist), Siddhi_desai@ymail.com
Second Author – Kavita Joshi, M.tech-Computer Engineering,joshikavita171@gmail.com
Third Author – Bhavik Desai, M.Tech-CAD/CAM, bhavikdesaivapi@gmail.com