Publications of the Astronomical Society of Australia (), 1–??
doi:

ARTICLE

Reinforcement Learning

Yashovardhan Srivastava

(Received ; revised ; accepted ; first published online)

1. INTRODUCTION

Reinforcement learning is the study in which we will see how agents can learn *what to do* in absence of labeled examples. For many problems, a supervised learning agent needs to be told exactly what to do for a particular instance of a problem. Consider the problem of playing any games such as chess - where a supervised learning agent needs to be told the correct move for each state that it encounters, but such feedback is seldom available and with problems that have large state spaces, the task of getting such data is close to impossible. In reinforcement learning, an agent is given feedback (called **reward** or **reinforcement**) which tells the agent whether something good / bad has happened or not. The main task is to use these observed rewards to learn an optimal policy[a] (or nearly optimal policy) with proficiency without any prior knowledge about both the reward and the environment.

In many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels. For a chess game -as described above, it is very difficult for humans to provide accurate and consistent evaluations of large number of positions. These evaluations are then used to train an evaluation function directly from examples. In reinforcement learning, we just tell the program when it has lost or won, and it can use this information to learn an evaluation function that gives reasonably accurate estimates.

In this chapter, we will be focusing only on simple environments and simple agent designs. The three agent design we will be discussing in detail are:

- **Utility-based Agent**: It learns a utility function on states and uses it to select actions that maximizes expected outcome utility.
- **Q-Learning Agent**: It learns an **action-utility** function giving expected utility of taking an action in a state.
- **Reflex Agent**: It learns a policy that maps directly from states to actions.

Passive Reinforcement Learning

In passive reinforcement learning, the agent's policy π is fixed: in state s it always execute $\pi(s)$. The goal is to simply learn the utility function $U^{\pi}(s)$. The passive learning task is similar to **policy evaluation**- apart from the fact that a passive learning agent does not know the transition model $P(s'|s,a)$, nor the reward function $R(s)$. The agent executes a set of trials in the environment using its policy π. In each trial, it starts with a initial state,experiences a sequence of state transitions until it reaches terminal state. The main objective is to use the information about the rewards to learn the expected utility $U^{\pi}(s)$ associated with each non-terminal state s. So, we have:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^{t} R(S_t)\right]$$

where $R(s)$ is the reward for a state, S_t is the state reached at time t when executing a policy π. γ is the discount factor.

Direct Utility Estimation

In Direct utility estimation, the utility of a state is the expected total reward from that state onward(called expected **reward-to-go**), and each trial provides a sample of this quantity for each state visited. The algorithm calculates the observed reward-to-go for each state and updates the estimated utility for the state accordingly, just by keeping a running average for each state in a trial. In the limit of infinitely many trials, the sample average will converge to the true expectation.

Direct utility estimation succeeds in reducing the reinforcement learning problem to an inductive learning problem, about which much is known. Unfortunately, it misses a very important property about utilities of state are not independent of each other - as the obey the Bellman equations.

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi(s)) U^{\pi}(s')$$

By assuming independence among states, direct utility estimation misses opportunities for learning. Direct utility estimation can be viewed as searching for U in hypothesis space that is much larger than it needs to be(as they include functions that violate Bellman equations). The algorithm often converges very slowly.

Adaptive Dynamic Programming

The ADP and TD approach are actually closely related. Both try to make local adjustments to the utility estimates in order to make the state 'agree' with its successors – with the exception that TD adjusts a state to agree with its observed successor, whereas ADP adjusts the state to agree with all of the that might occur,weighted by their probabilities. This difference disappears when the effects of TD adjustments are averaged over large number of transitions. TD can be viewed as a crude but efficient first approximation to ADP.

Active Reinforcement Learning

A passive learning agent has a fixed policy that determines its behavior. An active agent must decide what actions to take. We will study the case of an ADP agent and consider how it must be modified to handle this new freedom.

First the agent will need to learn a complete model with outcome probabilities for all actions, rather than just the model for the fixed policy. Next, we need to take into account the fact that agent has a choice of actions. The utilities it needs to learn are those defined by the optimal policy – so they must obey Bellman equations.

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s, a) U(s')$$

These equations can be solved to obtain the utility function U using value iteration or policy iteration algorithms.Having obtained the the utility function, the agent can extract an optimal action by one-step look ahead. Alternatively, if it uses policy iteration, so it can simply execute action that the optimal policy recommends. Which of these is better ?

Exploration vs Exploitation

Experiments have shown that choosing the optimal actions from the utility function can lead to sub-optimal results. These so-called **greedy agents** very seldom converges to the optimal policy. The question is why is that the case ? The answer is that the learned model and the true environment are no the same; what is optimal in the learned model might be sub-optimal in true environment. Since the agent does not know the true environment, it cannot compute optimal action for the optimal action. What, then, is to be done ?

The greedy agent has overlooked one important aspect of actions that they provide something more than reward to the current learned model, they also contribute to the learning the true model by affecting the precepts that are received. By improving the model, the agent will receive greater rewards in the future.

This trade off is often being dubbed as **Exploration vs. Exploitation**. An agent must make a trade off between exploitation to maximize its reward and exploration to maximize its long term well being. Pure exploitation risks getting stuck in a rut. Pure exploration to improves one's knowledge is of no use if one never puts that knowledge into practice[b].

The next question that arises is whether their is an optimal exploration policy ? This question has been studied in depth in the sub-field of statistical decision theory that deals with **bandit problems**. Bandit problems are extremely difficult to solve exactly. Nonetheless, it is possible to come up with reasonable schemes that will eventually lead to optimal policy. Any such scheme needs to be greedy in limit of infinite exploration – or **GLIE**.

GLIE Schemes

GLIE schemes, as described above are reasonable schemes that eventually lead to optimal behavior by the agent. There are several GLIE schemes, the simplest of which is to have the agent choose a ransom action $1/t$ of the time and to follow greedy policy otherwise. While this does eventually converge to an optimal policy, it is extremely slow. A more sensible approach (an optimistic one) is to give weights to actions that the agent has not tried enough, while tending to avoid actions that are *believed* to be of low utility. This causes the agent to behave initially as if there are wonderful rewards all over the place – which can be explored. We can re-write the update equation to account for this.

$$U^{+} \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s'|s, a) U^{+}(s'), N(s, a)\right)$$

[b]Parallels are often drawn with the real world – one has to constantly decide whether to continue in a comfortable existence or to seek out in the unknown in the hope of discovering a new and better life.

Here, $f(u, n)$ is the exploration function. It determines how **greed**(preference for higher utility values) is traded off against **curiosity**(exploring actions that have not been tried often and have low utility). The function should be increasing in u and decreasing in n. One particularly simple definition of exploration function is:

$$f(u, n) = \{ R^+ \text{ if } n \leq N_e 0\text{otherwise}$$

where R^+ is an optimistic estimate of the best possible reward obtainable in any state. N_e is a fixed parameter which has the effect on agent to try each action-state pair at least N_e times.

One important point to note is that in the update equation U^+ appears instead of U. As exploration proceeds, the states and actions near the start state might well be tried a large number of times. If we used U(the more pessimistic estimate), then the agent would soon become disinclined to explore further afield. The use of U^+ means that actions that lead towards unexplored regions are weighted more highly, rather than actions that are themselves unfamiliar. This has the effect that explorations policies shows rapid convergence.

Action-Utility Functions

This section focuses on constructing an active temporal difference learning agent. The obvious change from the passive case is that the agent is no longer fixated towards a single policy - so if it learns a utility function U, it will need to learn a model in order to be able to choose an action based on the utility function via one step look ahead. The model acquisition for the TD agent is identical to that for the ADP agent. The update rule is also similar. The TD algorithm will converge to the same values as ADP as the number of training examples tends to infinity.

There is an alternative TD method, called **Q-Learning**, which learns an action-utility representation instead of learning utilities. Q-values are directly related to utility values as follows.

$$U(s) = \max_a Q(s, a)$$

Q-functions have a very important property: *a TD agent that learns a Q-function does not need a model of the form $P(s'|s, a)$, either for learning or action selection* - it is therefore called **model-free** method. For utilities, we can write constraint equations that must hold at equilibrium(when Q-values are correct) :

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a')$$

We can use this constraint equation similarly to how we used with an ADP agent - that is to use it as an iteration process that calculates exact Q-values, given an estimated model. This. however means that we are required to learn the model(as we have a term of state transition). Temporal difference approach, on the other hand requires only Q-values. The update equation for TD Q-Learning is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

or

$$Q(s, a) \leftarrow (1 - \alpha)\, Q(s, a) + \alpha \left(R(s) + \gamma \max_{a'} Q(s', a') \right)$$

which can be seen as diffusing current Q-value with the one calculated from utility equation. It is calculated whenever action a is executed in state s leading to state s'.

Q-learning has a close relative called **SARSA**(for State-Action-Reward-State-Action). The update rule of SARSA is very similar to that of an TD agent :

$$Q(s, a) \leftarrow (1 - \alpha)\, Q(s, a) + \alpha \left(R(s) + \gamma Q(s', a') \right)$$

where a' is the action *actually* taken in state s'. The difference between them is quite subtle : while Q-learning backs up the *best* Q-value from the state reached, SARSA waits until an action is actually taken and backs up the Q-value for that action. During the exploration phase, they differ significantly. Because Q-learning uses the best Q-value, it pays no attention to the actual policy being followed(it is therefore called **off-policy** learning algorithm). SARSA on the other hand, is an **on-policy** learning algorithm. Q-learning is more flexible than SARSA in the sense that it can behave well even when guided by stochastic exploration policy. SARSA, on the other hand is more realistic than Q-learning.

Analysis of both Q-learning and SARSA raise a general question: *Is it better to learn a model and a utility function or to learn an action-utility function with no model ?* or to put precisely, *what is the best way to represent an agent function ?* While many claim that the availability of model-free methods such as Q-learning means that model-based approach are unnecessary, others rely on the intuition that as environment becomes complex, knowledge based approach become more apparent.

Generalization in Reinforcement Learning

So far, our assumption that utility functions and Q-functions are represented in tabular form is reasonable. However, as the size of state spaces increases, they become difficult to retain. One way to handle such problem is to use a **function approximation**, which means using any sort of representation for the Q-learning function other than a lookup table. Function approximation makes it practical to represent utility function for very large state space. Apart from this, *the compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited.* This implies that function approximation allows for inductive generalization over input states.

On the flip side, there is the problem that there isn't any function in the hypothesis space that fails to approximate the true utility function. As in all inductive learning - there is a trade-off between size of the hypothesis space and the time it takes to learn the function.

Let us begin with a simple case: which is direct utility estimation.With function approximation, this is an instance of supervised learning.For example, if we represent utilities using a simple linear function and given a set of sample values, we can always find the best fit.

For reinforcement learning it makes more sense to use an online algorithm that updates the parameters after each trial. Similar to neural network learning, we can write an error function and compute its gradient with respect to the parameters, then we move the parameter in the direction of decreasing the error.This is called the **Widrow-Hoff** rule or the **delta** rule, for online least squares.

$$\theta_i \leftarrow \theta_i - \alpha \left(\frac{\partial E}{\partial \theta_i} \right)$$

We can apply these ideas equally well to TD learners. All we need to do is adjust the parameters to try to reduce the temporal difference between successive states. The new versions of the TD and Q-learning equations are given by

$$\theta_i \leftarrow \theta_i + \alpha \left(R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s) \right) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha \left(R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a) \right) \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

for Q-values. For passive TD learning, the update rule can be shown to converge to the closest approximation when the function approximator is *linear* in the parameters. With active learning and neural networks, all bets are off -anything is possible.

Policy Search

The final approach for reinforcement learning problems is **policy search**. The idea is to keep twiddling the policy as long as the policy improves, and then stop. We will focus primarily on parameterized representation of policy that have far fewer parameters than there are states. For example: we could represent policy π by a collection of parameterized Q-functions, one for each action and take the one with highest predicted value.

$$\pi(s) = \max_a \hat{Q}_\theta(s, a)$$

Q-function could be linear or non-linear in θ. Policy search will then adjust the parameters θ to improve the policy. If the policy is represented by Q-functions, then policy search results in a process that learns Q-functions [c].

One problem,however with policy representations of this kind is that the policy is *discontinuous* function of parameters, when the actions are discrete - so even a small change in parameter causes change of policy. For this reason, policy search methods often

[c]This is not same as Q-learning. Policy search adjusts the parameters so as to improve performance, which might not be same as finding the optimal Q-function.

use a stochastic policy representation which specifies the probability of selecting an action *a* in state *s*. One popular representation is the **softmax function**

$$\pi_\theta(s, a) = \frac{e^{\hat{Q}_\theta(s,a)}}{\sum_{a'} e^{\hat{Q}_\theta(s,a')}}$$

For improving a deterministic policy in a deterministic environment, let $\rho(\theta)$ be the **policy value**. If we can derive an expression for $\rho(\theta)$ in closed form, we can optimize it using standard optimization techniques. We can follow the policy gradient vector $\nabla_\theta \rho(\theta)$ provided $\rho(\theta)$ is differentiable. If policy is not available in closed form, we can simply evaluate accumulated reward by executing the policy. Otherwise, we can follow the empirical gradient by hill climbing.

When the environment or policy is stochastic, things get more difficult. The problem is that total reward on each trial may vary widely. One solution is to run a lot of trials to get a reliable indication of direction of improvement - which is impractical for many real-life problems.

References

[1] Russel,Stuart and Norvig,Peter.*Artificial Intelligence-A Modern Approach*(3rd ed.).Pearson Education,2010