REINFORCEMENT LEARNING

REINFORCEMENT LEARNING AND DYNAMIC PROGRAMMING

(USING BELLMAN EQUATIONS TO COMPUTE VALUES AND OPTIMAL POLICIES, THUS, A FORM OF PLANNING)

AUTHORS

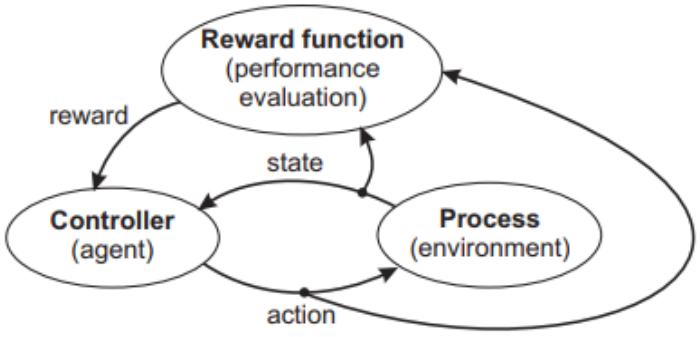
ROHITH G [CB.EN.U4AIE19026]

ABSTRACT

Dynamic programming (DP) and reinforcement learning (RL) can be used to address problems from a variety of fields, including automatic control, artificial intelligence, operations research, and economy. Many problems in these fields are described by continuous variables, whereas DP and RL can find exact solutions only in the discrete case. Therefore, the approximation is essential in practical DP and RL. This provides an in-depth review of the literature on approximate DP and RL in large or continuous-space, infinite-horizon problems. Value iteration, policy iteration, and policy search approaches are presented in turn. Model-based (DP) as well as online and batch model-free (RL) algorithms are discussed. We review theoretical guarantees on the approximate solutions produced by these algorithms. Numerical examples illustrate the behavior of several representative algorithms in practice. Techniques to automatically derive value function approximators are discussed, and a comparison between value iteration, policy iteration, and policy search is provided. We close with a discussion of open issues and promising research directions in approximate DP and RL.

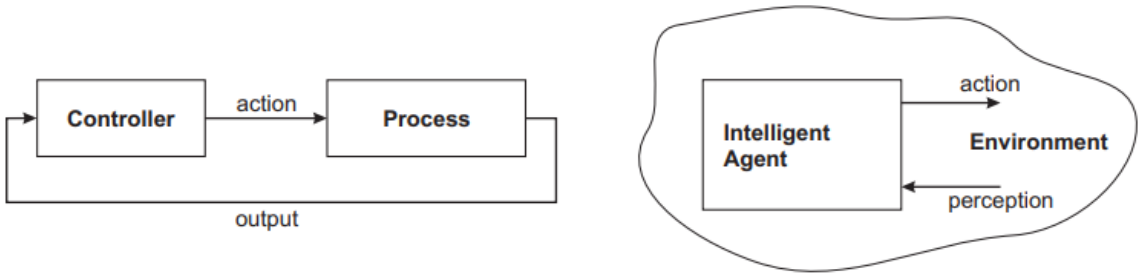
INTRODUCTION

Dynamic programming (DP) and reinforcement learning (RL) are algorithmic methods for solving problems in which actions (decisions) are applied to a system over an extended period of time, in order to achieve the desired goal. DP methods require a model of the system's behavior, whereas RL methods do not. The time variable is usually discrete and actions are taken at every discrete time step, leading to a sequential decision-making problem. The actions are taken in a closed-loop, which means that the outcome of earlier actions is monitored and taken into account when choosing new actions. Rewards are provided that evaluate the one-step decision-making performance. The goal is to optimize the long-term performance, measured by the total reward accumulated over the course of interaction.



Such decision-making problems appear in various fields, including automatic control, artificial intelligence, operations research, economics, and medicine. For instance, in automatic control, a controller receives output measurements from a process and applies actions to this

process in order to make its behavior satisfy specific requirements (Levine, 1996). In this context, DP and RL methods can be applied to solve optimal control problems, in which the behavior of the process is evaluated using a cost function that plays a similar role to the rewards. The decision-maker is the controller, and the system is the controlled process.



(a) Automatic control. (b) Artificial intelligent agents.

Two application domains for dynamic programming and reinforcement learning.

In artificial intelligence, DP and RL are useful to obtain optimal behavior for intelligent agents, which, as shown in Figure (b), monitor their environment through perceptions and influence it by applying actions (Russell and Norvig, 2003). The decision-maker is now the agent, and the system is the agent's environment. If a model of the system is available, DP methods can be applied. A key benefit of DP methods is that they make few assumptions about the system, which can generally be nonlinear and stochastic (Bertsekas, 2005a, 2007). This is in contrast to, e.g., classical techniques from automatic control, many of which require restrictive assumptions on the system, such as linearity or determinism. Moreover, many DP methods do not require an analytical expression of the model but are able to work with a simulation model instead. Constructing a simulation model is often easier than deriving an analytical model, especially when the system behavior is stochastic. However, sometimes a model of the system cannot be obtained at all, e.g., because the system is not fully known beforehand, is insufficiently understood, or obtaining a model is too costly. RL methods are helpful in this case since they work using only data obtained from the system, without requiring a model of its behavior (Sutton and Barto, 1998). Offline RL methods are applicable if data can be obtained in advance. Online RL algorithms learn a solution by interacting with the system and can therefore be applied even when data is not available in advance. For instance, intelligent agents are often placed in environments that are not fully known beforehand, which makes it impossible to obtain data in advance. Note that RL methods can, of course, also be applied when a model is available, simply by using the model instead of the real system to generate data. Here, we primarily adopt a control-theoretic point of view, and hence employ control-theoretical notation and terminology, and choose control systems as examples to illustrate the behavior of DP and RL algorithms. We nevertheless also exploit results from other fields, in particular the strong body of RL research from the field of artificial intelligence. Moreover, the methodology we describe is applicable to sequential decision problems in many other fields.

METHODOLOGY

Dynamic programming can be used to solve reinforcement learning problems when someone tells us the structure of the MDP (i.e when we know the transition structure, reward structure, etc.).

Therefore dynamic programming is used for the *planning* in an MDP either to solve:

1. **Prediction problem *(Policy Evaluation):***

Given an MDP $<S, A, P, R, \gamma>$ and a policy π. Find the value function v_π (which tells you how much reward you are going to get in each state). i.e the goal is to find out how good a policy π is.

Prediction problems can be solved using the Bellman Expectation Equation Iteratively (Policy Evaluation).

Policy Evaluation:

Problem: Evaluate a given policy π and MDP. (Find out how good a policy π is)

***Solution*:** Iterative application of Bellman Expectation backup.

Approach:

Start with the initial value function v_1 (a value of all states in the MDP). E.g. start with a value of 0. Therefore there is no reward. Then use the Bellman Expectation Equation to compute v_2 and repeat many times, which will eventually converge to v_π.

$$v_1 \rightarrow v_2 \rightarrow v_3 \ldots \rightarrow v_\pi$$

One method to achieve this convergence is to use *synchronous backups* where we consider all states at every step.

1. At each iteration k+1, for all states $s \in S$

2. Update $v_{k+1}(s)$ from $v_k(s')$, where s' is a successor state of s

$v_k(s')$ is updated using the Bellman Expectation Equation

How to Improve a Policy?

In the above approach, we have evaluated a given policy but have not found the best policy (actions to take) in our environment. In order to improve a given policy, we can *evaluate* a given policy π and improve the policy by *acting greedily* with respect to v_π. This can be done using

Policy Iteration.

2. Control Problem (*Find the best thing to do in MDP*):

Given an MDP *<S, A, P, R, γ>*. Find the optimal value function *v_π and* the optimal policy *π**. i.e the goal is to find the policy which gives you the most reward you can receive with the best action to choose.

Control problems can be solved using Bellman Expectation Equation Policy Iteration & Greedy (Policy Improvement) or the Bellman Optimality Equation (Value Iteration).

Policy Iteration

Problem: Find the best policy *π** for a given MDP.
***Solution*:** Bellman Expectation Equation Policy Iteration and Acting Greedy.

Approach:

Start with a given policy π

1. Evaluate the policy π with Policy Evaluation
2. Improve the policy π by acting greedily with respect to v_π with Policy Improvement to get a new policy π'.
3. Repeat until the new policy π' converges to the optimal policy π*.

In order to act greedily, we use a one-step look ahead to determine the action which gives us the maximum *action-value function.*

$$\pi'(s) = \arg\min_{a \in A} q_\pi(s,a)$$

The action-value function has the following equations:

$$q_\pi(s,a) = E_\pi[R_{t+1} + \Upsilon q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

$$q_\pi(s,a) = R_s^a + \Upsilon \sum_{s' \in S} P_{ss'}^a v_\pi(s')$$

$$q_\pi(s,a) = R_s^a + \Upsilon \sum_{s' \in S} P_{ss'}^a \sum_{a \in A} \pi(a' \mid s') q_\pi(s', a')$$

Action-Value Function

Value Iteration

An alternative approach to control problems is with value iteration using the *Bellman Optimality Equation*. First, we need to define how we can divide an optimal policy into its components using the ***Principle of Optimality.***

Principle of Optimality:

Any optimal policy can be subdivided into two components that make the overall behavior optimal:

->An optimal first action $A*$

->Followed by an optimal policy from successor state S

Principle of Optimality Theorem:

A policy $\pi(a|s)$ achieves the optimal value from state s, $v\_\pi(s)=v*(s)$, if and only if:

->For any state s' reachable from *s*

->π achieves the optimal value from state s', $v\_\pi(s')=v*(s')$

Now coming to the Problem and Solution of the Value Iteration.

Problem: Find the optimal policy π^* for a given MDP.
Solution: Iterative application of Bellman Optimality backup.

Approach:

Using synchronous backups updates the value function until the optimal value function is computed without computing the action-value function.

1. At each iteration k+1, for all states $s \in S$
2. Update $v_{k+1}(s)$ from $v_k(s')$, where s' is a successor state of s

$v_k(s')$ is updated using the Bellman Optimality Equation

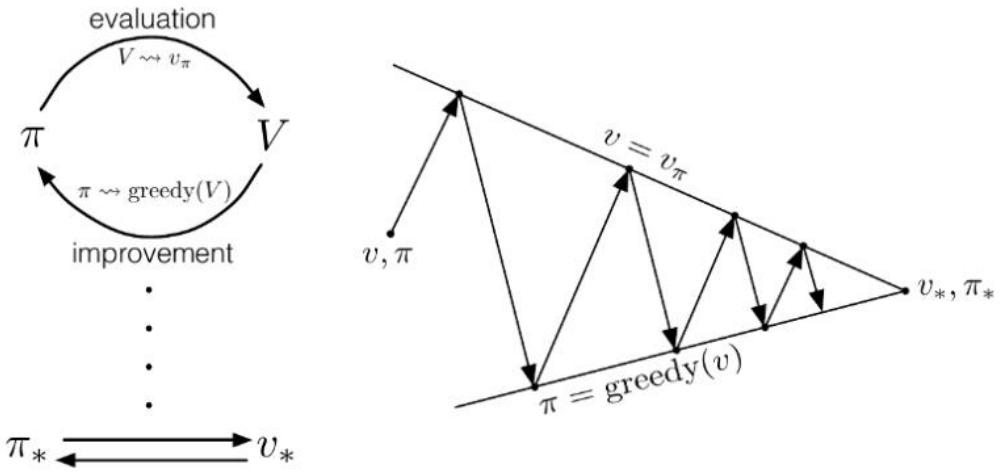
$$v_{k+1}(s) = \max_{a \in A}(R_a^s + \Upsilon \sum_{s' \in S} P_{ss'}^a v_k(s'))$$

Bellman Optimality Equation

By repeating the above process, it will eventually converge to v^*. Note this process differs to policy iteration in that intermediate value functions may not correspond to any policy.

1. Jack's Car Rental Problem

We'll try to **elucidate the policy iteration algorithm** in Reinforcement Learning by using it to solve Jack's Car Rental Problem.



Jack manages two locations for a nationwide car rental company. Each day, a number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited $10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for rent the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of $2 per car moved.

We assume that the number of cars requested and returned at each location is a Poisson random variable. Recall that if X is a Poisson random variable, then

$$\mathrm{P}(X = n) = \frac{\lambda^n e^{-\lambda}}{n!}$$

Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns.

To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate γ to be 0.9 and formulate this as a continuing finite Markov Decision Process, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations

overnight.

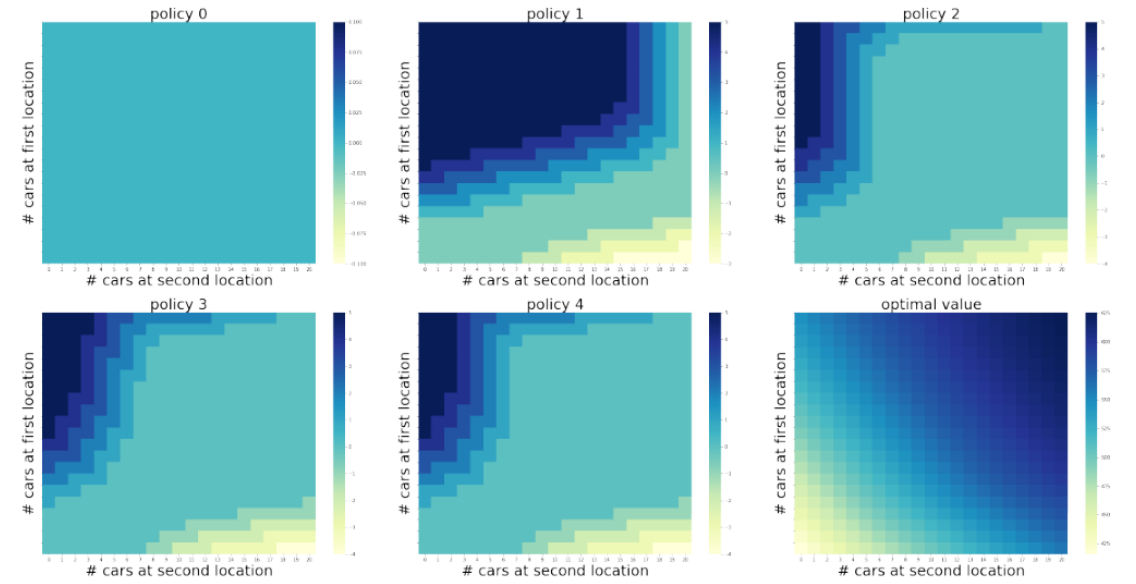
But what does it mean to solve this problem?

Solving this problem means solving two things, firstly how many cars should Jack move, overnight, between each location, to maximize his total expected reward, i.e., what should be his strategy (policy) given a situation (state), and secondly, if Jack knows this strategy, he compare which situations are better than others (value).

EXPERIMENTAL CODE 1

https://drive.google.com/file/d/1PDwH-7UObCTxDgJ8rHf44s-XOXOy9KML/view?usp=sharing

OUTPUT:



---

2. Gambler's problem

Our agent is playing a game where they can place a bet on whether a coin flip will show heads. If it is headed, then they win the same amount they bet, otherwise, the opposite happens and they lose all the money they bet. The coin lands on heads 40% of the time. The game continues over and over again until the player either has 0 capital (loss) or 100+ capital (win).

Key points of the problem:

1. Undiscounted, finite, episodic MDP: Therefore, gamma is 1, and there is a terminal state.

2. The state is the gambler's capital.
3. Policy is the mapping of levels of capital to how much the agent should bet
4. Terminal states 0 capital and 100+ capital.
5. Reward of 0 at all states except the 100+ state.

Solving the Problem:

we're expected to solve this using Value Iteration. One issue with Policy Iteration is that it required fully fitting a Value function with a particular policy. As, it can be a protracted procedure, whereby the convergence to v_pi occurs only in the limit. I suppose it wouldn't be as problematic if we didn't also switch to updating the policy (which then changes the value function) or if we knew that a value function that fully fits a policy resulted in a more efficient update to the policy than if we stopped the iterations earlier.

Instead of looping through a state space with a value function until it converges to v_pi, we can loop just once (updating the values on the way), during which we take the max value from the set of values and assign it to the state being investigated. Thus, every time we sweep through the states, we look at all the places we can go from that position, and simply indicate that the max reward from that position is the maximal value one could expect from that state.

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_k(s')\Big]$$

Value function for Value Iteration, where the equation is applied iteratively.

One way to think about this formula is that from a given state, there exist many actions. So we accumulate the values for how much each action would get in terms of expected value, and simply assign the value of the state to that of the action which returned the highest value. This formula is similar to the Policy Evaluation update except for the important fact of taking the maximum overall actions.

Another way to think about is that we've taken the Bellman Optimality equation and turned it into an update rule.

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_*(s')\Big]$$

Bellman Optimality Equation for the State Value Function.

Unlike Policy Iteration, we don't use or build an explicit policy at each step, exclusively working within the value space. We go from a value function, to value function, to a value function. At the end of the process, we are guaranteed to have the value function for the optimal policy. It's akin to a Modified Policy Iteration with k = 1 since we are taking the max over the next step.

One thing important to note is that the interim value functions between the initial value function and the optimal value function may not correspond to any policy. This is different than Policy Iteration where at every step value function is calculated for a particular policy.

$$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow \ldots \rightarrow v_*$$

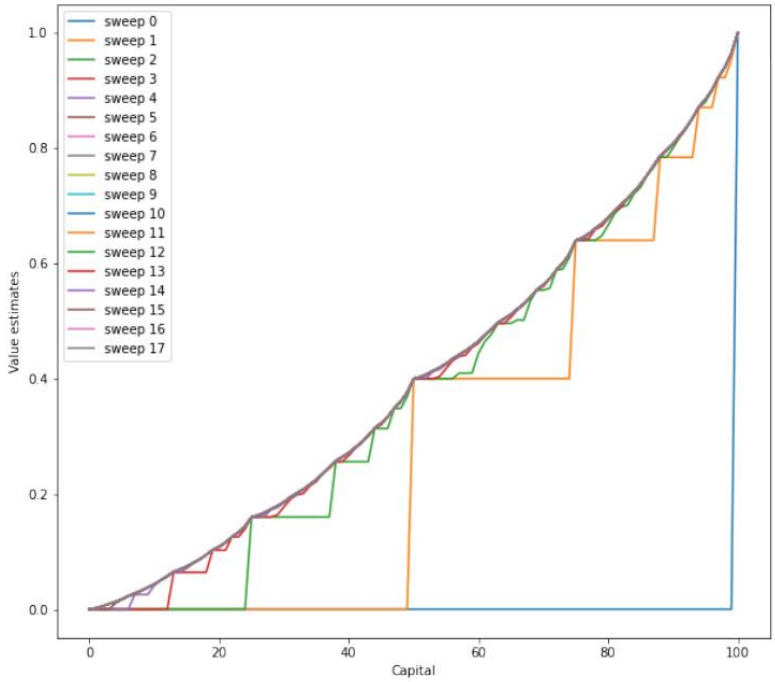
In essence, it combines one step of policy evaluation with policy improvement.

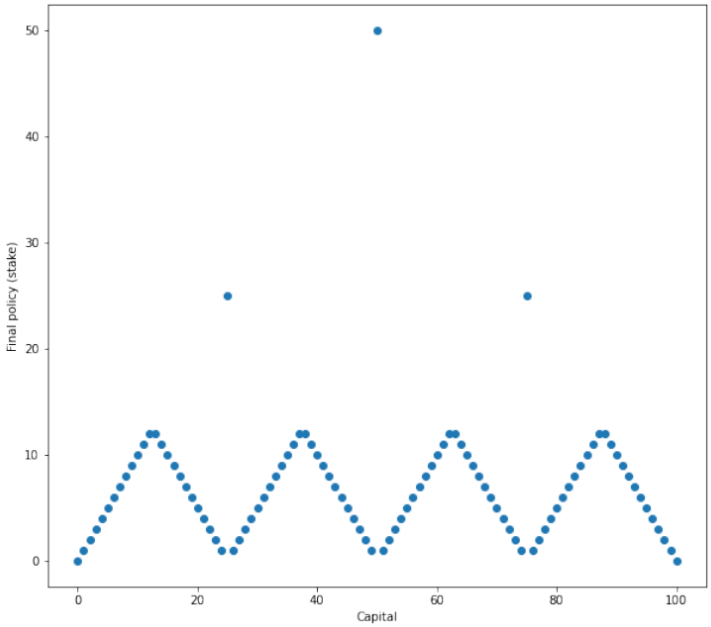
The Gambler's Ruin Problem is a great example of how you can take a complex situation and derive a simple general structure from it using statistical tools. It might be difficult to believe that, given a fair game, the probability that someone will win enough games to claim the total wealth of both players is determined by their initial wealth and the total wealth. This is known not only at the beginning of the sequence but also at each step. Using Markov chains, we can determine the same probabilities between any sequences of games using the transition matrix and the probability vector at the initial state. Consider this, the conclusion that we came to in the first section of this post was enhanced by the use of an additional concept. Applying different perspectives to the same problem can open the door to insightful analysis. This is the power of theoretical statistical thinking.

EXPERIMENTAL CODE 2

https://drive.google.com/file/d/1CkTv3w1CTOuBFWBHgVHcdIN2x2ghkSN6/view?usp=sharing

OUTPUT:

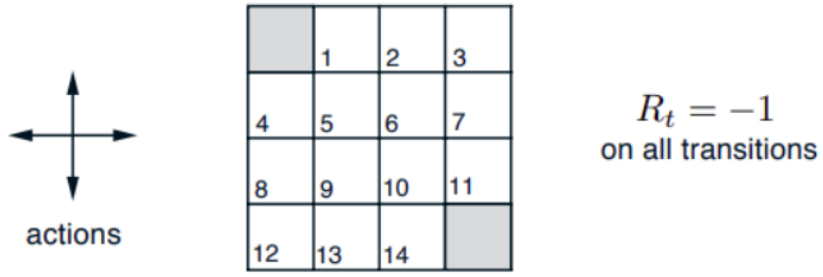




3. Convergence of Iterative Policy Evaluation on Small GridWorld

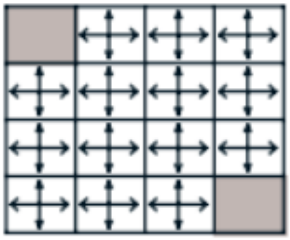
Gridworld is a common testbed environment for new RL algorithms. We consider a small Gridsworld, a 4x4 grid of cells, where the northmost-westmost cell and the southmost-eastmost cell are terminal states.

The agent can move between different cells.



We formulate the problem as an undiscounted, episodic MDP with:

- The set of states $\mathcal{S}$ = {0, 1, …, 14, 15} where 0 and 15 are absorbing states.
- The set of actions $\mathcal{A}$ = {north, south, west, east}
- The reward function $\mathcal{R}$ = {0 if s = 0 or s = 15, -1 otherwise}
- The discount factor γ = 1



Equiprobable random policy

The objective of this example is to evaluate the random equiprobable policy, i.e. for each state, all actions have equal probability: $\mathcal{P}$ = 0.25 for any state.

EXPERIMENTAL CODE 3

https://drive.google.com/file/d/1Evoq6Pdr63urD_KBwfNzAwBRAwDW5seh/view?usp=sharing

OUTPUT:

	1	2	3	4
1	0.0	-14.0	-20.0	-22.0
2	-14.0	-18.0	-20.0	-20.0
3	-20.0	-20.0	-18.0	-14.0
4	-22.0	-20.0	-14.0	0.0

Plot

CONCLUSION

We solved the above problem using Dynamic Programming. We stored the intermediate value and policy matrices and used them in our policy evaluation and improvement functions. In order to use Bellman updates though, we need to know the dynamics of the environment, just like we knew the probabilities for rewards and the next states in the rental example. If we can only sample from the underlying distribution and don't know the distribution itself, then Monte Carlo methods can be used to solve the corresponding learning problem. All the algorithms described in this post are solutions to planning problems in reinforcement learning (where we are given the MDP). These planning problems (prediction and control) can be solved using synchronous dynamic programming algorithms.

REFERENCES

[1] Sutton, R., & Barto, A. (2017). Reinforcement learning: An introduction. Cambridge, MIT Press.

[2] S. David's lecture on Planning by Dynamic Programming.

[3] Baddeley, B.: Reinforcement learning in continuous time and space: Interference and not ill-conditioning is the main problem when using distributed function approximators. IEEE Transactions on Systems, Man, and Cybernetics-Part B: Cybernetics 38(4), 950–956 (2008)

[4] Barash, D.: A genetic search in policy space for solving Markov decision processes. In: AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information. Palo Alto, US (1999)

[5] Barto, A.G., Sutton, R.S., Anderson, C.W.: Neuronlike adaptive elements that can solve difficult learning control problems. IEEE Transactions on Systems, Man, and Cybernetics 13(5), 833–846 (1983)

[6] Baxter, J., Bartlett, P.L.: Infinite-horizon policy-gradient estimation. Journal of Artificial Intelligence Research 15, 319–350 (2001)

[7] Berenji, H.R., Khedkar, P.: Learning and tuning fuzzy logic controllers through reinforcements. IEEE Transactions on Neural Networks 3(5), 724–740 (1992)

[8] Berenji, H.R., Vengerov, D.: A convergent actor-critic-based FRL algorithm with application to power management of wireless transmitters. IEEE Transactions on Fuzzy Systems 11(4), 478–485 (2003)

[9]UCL Course on RL — Lecture 3

[10]An Introduction to Reinforcement Learning, Sutton and Barto, 1998