A Survey of Reinforcement Learning Techniques

Preprint · March 2019

DOI: 10.13140/RG.2.2.26529.15204

1 author:

Ritvik Sharma
King's College London
2 PUBLICATIONS **4** CITATIONS

SEE PROFILE

Reinforcement Learning

Ritvik Sharma
ritvik.sharma@kcl.ac.uk

March 22, 2019

Thesis Statement

Reinforcement Learning (RL) has seen many applications in the recent past where it achieves super-human performance in various activities. In this literature review, we introduce RL and explain the most popular techniques for solving control problems using RL, namely Dynamic Programming, Temporal Difference Learning, Q-Learning, and Least Squares Policy Iteration, and discuss the benefits and applications of each of these techniques.

1 Introduction

Reinforcement Learning (RL) is a type of machine learning, where the agent is situated the environment and it can only learn about the environment by trial-and-error. Just like how a baby learns to walk, by failing and falling several times. RL has a long and embedded history in psychology and mathematics. Researchers wondered how we could replicate the way humans and animals learnt behaviour from experiences of trial and error. This led to the earliest work in artificial intelligence and brought about the concept of reinforcement learning. But it wasn't until researchers started solving control problems using value functions and dynamic programming that thoughts arose about intermingling these fields together. In the late 1980s when temporal difference methods, which used concepts from both areas, surfaced, they brought about the interpretation of the field of reinforcement learning as we know today. The trial-and-error method, infused with positive and negative reinforcements, learning how to act in an environment, to maximize

some form of cumulative rewards. Making it an excellent choice to solve control and decision-making problems, this area of machine learning has been researchers in disciplines from game theory, control theory and operations research to multi-agent systems, genetic algorithms and swarm intelligence.

For example, last few decades have seen extensive applications of RL in game theory, especially because of their ability to let the agent learn optimal strategies, only by methods of trial and error, even without initially knowing the rules of the game. Software agents have been built that have defeated champions in games of Chess [Silver et al., 2017], Atari [Mnih et al., 2013], Go [Silver et al., 2007] and numerous others. RL has seen applications in computer networks, as in packet routing like [Boyan and Littman, 1993] as well as telecommunication like [Mustapha et al., 2015] and [Jiang et al., 2017]. Jiang explains how future mobile terminals can be made sophisticated so that they autonomously learn to access most efficient spectral bands by using RL.

[Abdulhai and Kattan, 2003] discussed the potential of reinforcement learning algorithms in the field of transport engineering and traffic control. They give a decent overview of reinforcement learning algorithms like Q-Learning and explain how they can be applied to create Intelligent Traffic Systems (ITS). Among the many ITS applications [Sadek and Basha, 2008] discuss how reinforcement learning can be applied for Dynamic Traffic Routing (DTR), in which drivers are recommended to route towards pathways that maximize network utilization. They created a self-learning RL agent and proved the concept using a Cell Transmission Model created at UC Berkeley and they showed that the promising results could also be scaled and implemented in road networks.

Not only this, RL is creeping towards multiple applications in the robotics sector as well. Robots, using RL, could not only learn to do simple tasks like pushing boxes [Mahadevan and Connell, 1992], but could also be trained to play complex multi-agent coordination games like soccer [Park et al., 2001]. As the fields of psychology, mathematics, physics and computer science seem to join in artificial intelligence, it can be said, without a doubt, that reinforcement learning will eventually have innumerable applications and amplify the artificial intelligence breakthroughs of this century.

To discuss the intricacies of RL, this literature survey explains concepts in reinforcement learning and points to numerous references from existing literature, hoping to familiarise the reader with RL and simulate ideas for further research. Section 2 introduces the background for reinforcement learning and the concept of Markov Decision Processes. Section 3 through section 6 dis-

cuss the various techniques to solve the reinforcement learning problems. Section 3 focusses on Dynamic Programming techniques, whereas section 4 focusses on the Temporal Difference Learning techniques. Section 5 explores the Q-Learning algorithm for solving the control problems, and Section 6 explains the recent Least Squares Policy Iteration algorithm for efficiently and approximately solving large reinforcement learning problems. Key summarising points from all sections are mentioned in the concluding section 7.

2 Background

Reinforcement Learning is applied in situations where we need to decide what actions we should make to maximise some notion of rewards. This kind of problem where we need to control behaviour of an agent is termed as a control problem. The agent is *situated* in an environment, placed in some state S_t at time t where it takes an action A_t, which affects the environment is some way, causing the state of the agent to shift to S_{t+1} in that time step $(t+1)$. The agent also receives a reward at this state, termed as R_{t+1}. This is shown in the figure 1 taken from [Sutton and Barto, 1998].

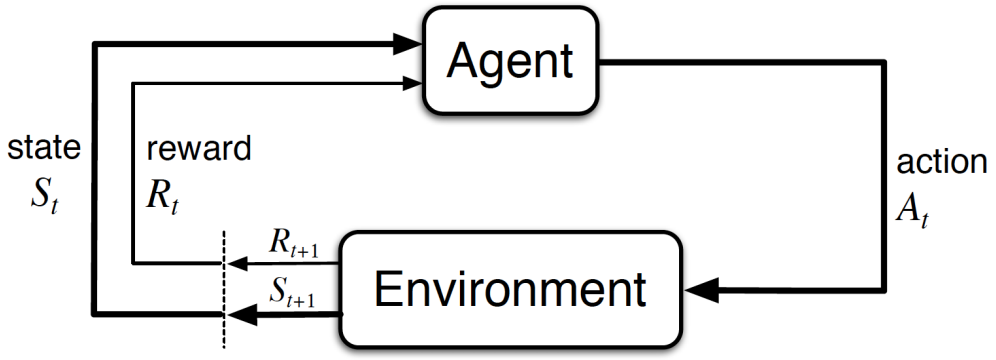

Figure 1: Interaction between the agent and the environment

Now this agent-environment interaction in a control problem can be mathematically formulated into something called as a Markov Decision Process(MDP). An MDP is a 4-tuple (S, A, P, R) where:

- S is the set of states, with an initial state at time t as S_t or s_0.

- $A(s)$ is the set of actions possible in a state s.
- $P(s'|s,a)$ is a transition probability model that gives the probability of leading to state s' from state s, when an action a is performed.
- $R(s)$ is the immediate reward received on reaching a state s.

Markov decision processes are based on an underlying Markov Chain (MC), where the system(environment) moves from one state to another state only in discrete time steps and that the probability of transition to the next state only depends on the action chosen on the current state and not an action chosen earlier (first-order MC). Although this makes you ponder that 'how will we attribute the sequential journey of the agent?' because often in life, an action taken earlier, decides our choices of actions in the future (like studying sciences in college eliminates your possibility to be a commerce professor), but the concrete idea behind the MDP is that what state you were in, decided your current state, and hence your future (next) state, can then be attributed only to your current state. That is why, although an assumption, first-order Markov chains hold good for many situations we see in the environment.

Now, given an MDP, when the agent is situated in a certain state s, the task of the agent is to choose an action a by which it can receive the *maximum cumulative reward*. The action is deemed as the optimal action in that state. Note that, cumulative in maximum cumulative reward means that the action chosen need not be the action giving the maximum immediate reward, but the action which will eventually make the agent earn the highest reward at the end of its *run*. A *policy* $\pi : S \to A$, maps all states to actions and defines the action to be taken when in each state. Hence the task is to find the set of all optimal actions in all possible states, which is termed as finding the *optimal policy* π^*.

Now the question arises that at any state s_t, "how does one choose the maximum *cumulative* rewarding action?" To formalise this, we define the cumulative value $V^\pi(s_t)$ achieved by following an arbitrary policy π when at an arbitrary initial state s_t as below:

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} .... = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

The r_t value is the reward received at time step t and γ, the *discount factor*, determines how important a future reward is in comparison with the

immediate reward. Value of γ ranges between 0 and 1. When the weightage of a future reward is the same as the immediate reward, γ is 1 but as in many cases, we prefer to obtain the reward sooner rather than later, hence here we restrict γ as $0 \leq \gamma < 1$. This helps in proving the convergence of the algorithms we describe in the next sections.

3 Dynamic Programming

The term dynamic programming (DP) was coined by Richard Bellman in the early 1950s and refers to a collection of algorithms that are used to compute optimal policies, given a perfect model of the environment as a Markov decision process. The key idea behind DP is that we simplify a complicated problem by breaking it down into simpler sub-problems, in a recursive fashion such that the simpler sub-problems become much easier to solve than the original complex problem. As [Sutton and Barto, 1998] exclaims, classical DP algorithms are not of much practical use in reinforcement learning because not only is their assumption of a perfect model usually incorrect, but also they are of great computational expense for solving the complexity of problems RL deals with. Yet, they are still important theoretically and for this literature survey because DP provides an essential foundation for understanding the other more complex methods presented later in this paper. Not only that, all of the other methods can be seen as attempts to achieve approximately the same effect as DP, only with lesser computation and without assuming a perfect model of the environment.

Because we assume that environment is a finite MDP, we assume that its action, state, and reward sets, A, S, and R, are finite, and that its state transition probabilities are given by $p(s', r|s, a)$, for all $s \epsilon S$, $a \epsilon A(s)$, $r \epsilon R$, and $s' \epsilon S$. Now, given this MDP, the utility value of each state is given by the Bellman equation [Bellman, 1954] as follows:

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

For getting an optimal policy of an MDP with n states, we have n Bellman equations and although n linear equations with n unknowns can be solved using simultaneous equations, but because of the max operation, the equations are non-linear. Hence, we use the *Value Iteration* algorithm to iteratively solve the MDP for utility values of all the states. The Value Iteration

algorithm is given below:

Algorithm 1 The Value Iteration algorithm

Input:
S is the set of states, with an initial state at time t as S_t or s_0.
$A(s)$ is the set of actions possible in a state s.
$P(s'|s,a)$ is a transition probability model.
$R(s)$ is the immediate reward received on reaching a state s.
$U(s)$ is the utility value of the state s.

1: $\mathbf{U}(s) \leftarrow \mathbf{0}$
2: **repeat**
3: $\quad U' \leftarrow U$
4: $\quad$ **for** s in S **do**
5: $\quad\quad U(s) = R(s) + \gamma\ max_{a \epsilon A(s)} \sum_{s'} P(s'|s,a) U'(s')$
6: **until** $U \approx U'$

This iterative procedure provides a neat way to obtain the final utility values for all the states in the MDP. The optimum policy is then computed as below for each of the states:

$$\pi^*(s) = arg\ max_{a \epsilon A(s)} \sum_{s'} P(s'|s,a) U^{\pi^*}(s')$$

In [Howard, 1960] invented the policy iteration method for MDPs which instead of computing optimal utility values and then calculating optimal policy, it looks through the space of possible policies and tries to find the ideal policy. The Policy Iteration algorithm is as given below:

Algorithm 2 The Policy Iteration algorithm

Input:
S is the set of states, with an initial state at time t as S_t or s_0.
$A(s)$ is the set of actions possible in a state s.
$P(s'|s,a)$ is a transition probability model.
$R(s)$ is the immediate reward received on reaching a state s.
π is a policy, mapping from $S \rightarrow A$
$U(s)$ is the utility value of the state s.

1: $\pi \leftarrow \pi_0$
2: **repeat**
3: //Policy Evaluation
4: **for** every state **do**
5: $U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s,a) U_i(s')$
//Policy Improvement
6: **for** every state **do**
7: $\pi_{i+1}(s) = arg\ max_{a \epsilon A(s)} \sum_{s'} P(s'|s,a) U_i(s')$
8: **until** $\pi_i \approx \pi_{i+1}$

Both these algorithms get exact optimal policy results but as you can observe, the nested loops for each state causes numerous iterations and hence these algorithms become computationally expensive for large MDPs. Hence, we move towards a more suitable solution in the next section on Temporal Difference Learning.

4 Temporal-Difference Learning

Temporal Difference (TD) Learning is one of the most novel ideas in reinforcement learning. Based on the earlier work by Arthur Samuel [Samuel, 1959], the invention of TD Learning is attributed to Richard S. Sutton for his contribution during his Ph.D. dissertation [Sutton, 1984] and publication [Sutton, 1988]. Learning with temporal differences uses techniques from Monte Carlo (MC) methods, where it directly learns from the ongoing occurrences of events, rather than creating a model of the environment. It also uses ideas from dynamic programming (DP), where it bases its current update on an existing estimate of utilities (bootstrapping), and not waiting for an outcome to determine utility values.

The simplest temporal difference learning algorithm is the TD(0) algorithm [Sutton and Barto, 1998] as shown below:

Algorithm 3 TD(0) Learning for estimating U^{π}

Input: the policy π to be evaluated
Parameter: learning rate or step size $\alpha \in (0, 1]$
Initialize U(s), for all s ϵ S^{+} arbitrarily, except U(terminal) = 0

```
1: for each episode do
2:     Initialize S
3:     for each step of episode, until S is terminal do
4:         A ← action given by π for S
5:         Take action A, observe R, S'
6:         U(S) ← U(S) + α [R + γ U(S') − U(s)]
7:         S ← S'
```

TD Learning is based on the theme that we strengthen an action by increasing its utility value if we receive a positive feedback, and decreasing, if we receive a negative feedback. When the feedback is only from one of the state transition of the Markov Chain, we call it as TD(0) or one-step TD method, and when the feedback involves inputs from multiple transitions, it is called a TD(λ) or n-step TD method. Note that TD Learning is model-free, i.e. there is no transition model needed to get the utility estimates, making TD Learning easy to apply. The name *temporal difference* comes from the fact that the iterative update in the algorithm is based on the differences between consecutive states i.e. temporal. The convergence of TD methods was proven by [Sutton, 1988, Watkins, 1989, Dayan, 1992] successfully and the methods have been widely used since, especially because of their property of bootstrapping.

The TD algorithms have specifically been applied in many gaming applications like in playing games such as Checkers [Samuel, 1959], Backgammon [Tesauro, 1994], Chess [Baxter et al., 1999] and Shogi [Beal and Smith, 2001]. TD algorithms also set the groundwork for the most famous algorithm in reinforcement learning, called Q-Learning, which we will cover in the next section.

5 Q-Learning

Q-Learning is amongst the most researched and applied method of reinforcement learning. It is a model-free technique that was derived from the TD(0) algorithm and from the definition of Q-values. Q-learning [Watkins, 1989] finds the optimal policy by implicitly representing the policy as a state-action value function Q, instead of just the utility value function $U(s)$ that we used in VI and PI algorithms and then we iterate as below:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Q-learning can be viewed as way of asynchronous dynamic programming too. Unlike in classical VI, where all values are updated in each iteration, in Q-learning, we update only those values asynchronously that lead to the optimal result. Although it may not seem intuitive, it has been proved in [Borkar, 1998] that asynchronous transformations can also lead to optimal solutions.

The convergence of Q-learning was proved in [Watkins and Dayan, 1992] and since then, Q-learning has boomed, especially because it was able to directly approximate the optimal state-action function Q^*, independent of any policy [Sutton and Barto, 1998]. This dramatically reduced the computation time and made convergence faster. Because it is considerably easier to simulate a complex system, rather than to estimate its transition probability model, reinforcement learning methods like Q-learning avoid the curse of modelling. It is to note that Q-learning is an *off-policy* algorithm, i.e. it learns the Q-values from the actions yielding the maximum values, rather than following any current policy.

[Boyan and Littman, 1993] show how Q-learning can be applied in computer networks to network packet routing. They show that without knowing anything about the traffic patterns or the topology of the centralized routing system, the Q-routing algorithm. is able determine efficient routing policies in the dynamic changing network.

Q-learning is applied to manufacturing systems in [Wang and Usher, 2005]. They discuss its potential in the production scheduling sector where a single agent can potentially learn the commonly accepted dispatching rules using the method of Q-learning.

Another interesting application of Q-learning in the sector of Cooperative Multiagent Systems as explained in [Claus and Boutilier, 1998]. They focus

on creating strategies that converge to the Nash equilibria in the system. [Park et al., 2001] further show how such systems could be implemented as a soccer game between robots. They implemented a modular Q-learning technique to make the robots cooperate and coordinate to play soccer with each other. Then, [Matignon et al., 2007] proposed a variation of Q-learning called hysteretic Q-learning where more efficient coordination is achieved in a similar cooperative multi-agent environment.

H-infinity techniques are used in control theory for making controllers reach stabilization with good guaranteed performance. [Al-Tamimi et al., 2007] show that linear, discrete-time, zero-sum games can have Q-learning implemented to get optimal strategies. They also show the immense impact of this method by performing an H-infinity control autopilot design for an F-16 aircraft.

Q-learning techniques, along with Ant System [Dorigo, 1992] gave rise to a family of algorithms called Ant-Q [Gambardella and Dorigo, 1995] which gave very good results on trying to solve the travelling salesman problem. They show that Ant-Q performed better than the usual Ant System technique and the results obtained were similar to those obtained only by very sophisticated algorithms.

It can be observed that Q-learning works very well when the state-action space is small, like a small grid. But as the state-action space increases, like a game of Atari, where each frame in the game is treated as a single state, we end up having millions of states. Hence, instead of storing and computing millions of records, we use a function approximator to estimate the value of Q. This is what we will discuss in the next section where methods like least-squares temporal difference and least-squares policy iteration will be used to approximate Q. We could also use a supervised learning technique for a reinforcement learning problem. This gave rise to a new technique in RL called as Deep Q-Learning, where the function approximator used is a neural network (deep learning). DeepMind in [Mnih et al., 2013] created a deep learning model to estimate and learn control policies by using a deep convolutional neural network, referred to as a Deep Q-Network (DQN). This was a breakthrough in high-dimensionality estimation and solved the problems with Q-learning that arose because of very large environments, leading to additional applications in behavioural learning of robots [Gu et al., 2017] and learning in biological domains like bioimaging [Mahmud et al., 2018].

6 Least-Squares Policy Iteration

Approximate reinforcement learning deals with the essential problem of applying reinforcement learning in large and continuous state-action spaces, by using function approximators to represent the solution. Least-Squares Policy Iteration (LSPI) is a reinforcement learning algorithm that learns the decision policy from samples using an approximate policy iteration method. Proposed in [Lagoudakis and Parr, 2003], it uses the idea from Q-learning, that even though we want a policy as a solution to the RL problem, we can represent the policy as a *state-action* value function Q and then use the approximate policy iteration algorithm to eventually determine the policy. This is why the policy is computed on demand, i.e. in any state s if you want to find its optimal policy direction, you can access the value function in that state to see all possible actions and then choose the maximum rewarding action (greedy choice) at that moment.

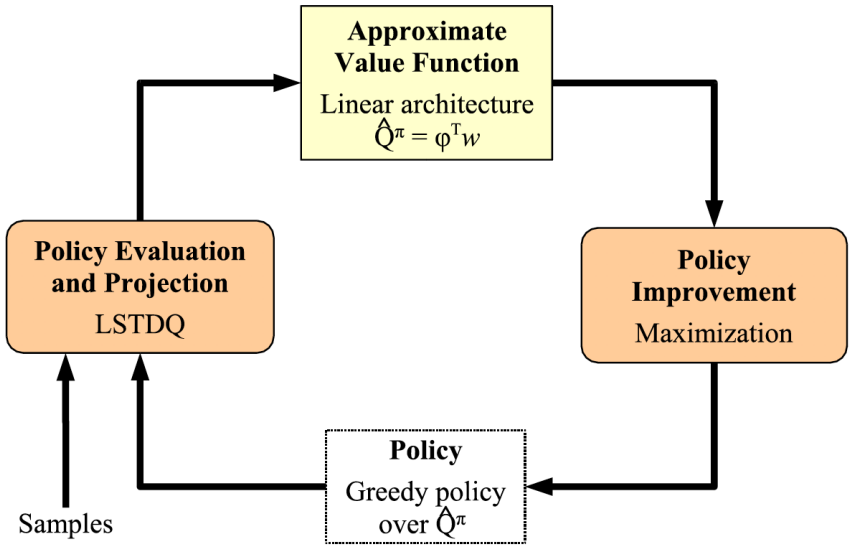

Figure 2: Least Squares Policy Iteration

We have seen in section 3 that policy iteration is composed of two steps, policy evaluation and policy improvement. As shown in figure 2, [Lagoudakis and Parr, 2003]

explains that LSPI uses a method called LSTDQ, which is similar to a method called Least Squares Temporal Difference (LSTD) [Bradtke and Barto, 1996]. Here, LSTDQ learns the approximate state-action value function $\hat{Q}^{\pi}$ of a policy π when the approximation architecture used is a linear architecture.

It is notable that LSPI uses a linear architecture as the approximating architecture and LSTDQ as the policy evaluation and projection technique, but practically, any other combinations of methods may also be used, and they would result is decent learning algorithms.

LSPI uses a linear architecture because not only are they easy to implement, and use, they are also quite transparent from a feature engineering and analysis perspective. The box at the top in figure 2 shows the linear architecture used, which formulates as follows:

$$\hat{Q}(s, a; w) = \sum_{i=1}^{k} \phi_i(s, a) w_i = \phi(s, a)^T w$$

where $\phi_i(s, a)$ are the arbitrary but fixed basis functions of state s and action a, and w_i are the parameters. The $\hat{Q}$ values are estimated by a linear combination of the k basis functions (features). Features are independent of each other, and k is usually very small in comparison with the number of states $\times$ actions. Hence, the storage requirements using a linear architecture are much smaller than tabular representations, like in value iteration algorithm where we make a table of all states.

As in the bottom box of the figure 2, this approximate value function $\hat{Q}$ is then used to find the greedy policy π, by finding the action that maximises the value over all possible actions:

$$\pi(s) = arg\, max_{a \epsilon A} \hat{Q}(s, a) = arg\, max_{a \epsilon A} \phi(s, a)^T w$$

This is straightforward when the action space is finite, but if the action space is very large or continuous, maximising over all possible actions may be costly, hence in such scenarios, some global optimisation over the entire space could be preformed, like in [Bradtke, 1993].

Algorithm 4 The LSTDQ algorithm

LSTDQ $(D, k, \phi, \gamma, \pi)$

Inputs:
D : Source of samples (s, a, r, s')
k : Number of basis functions
ϕ : Basis functions
γ : Discount factor
π : Policy whose value function is sought

1: $\tilde{\mathbf{A}} \leftarrow \mathbf{0}$ // $(k \times k)$ matrix
2: $\tilde{b} \leftarrow \mathbf{0}$ // $(k \times 1)$ vector
3: **for** each $(s, a, r, s') \epsilon\ D$ **do**
4: $\quad \tilde{\mathbf{A}} \leftarrow \tilde{\mathbf{A}} + \phi(s, a)(\phi(s, a) - \gamma\phi(s', \pi(s')))^T$
5: $\quad \tilde{b} \leftarrow \tilde{b} + \phi(s, a)r$
6: $\tilde{w}^\pi \leftarrow \tilde{\mathbf{A}}^{-1}\tilde{b}$
7: **return** $\tilde{w}^\pi$

Finally, in the policy evaluation and projection part, the policy π, which is represented by the parameters w and basis functions ϕ, is fed into LSTDQ for evaluation, together with a set of samples. LSTDQ then determines a policy π for each sample (s, a, r, s') by performing maximization and then it returns the new parameters w^π of the approximate value function of policy π. Iterations are performed in the same manner until consecutive generated policies are the same, i.e. LSPI converges. More details can be found in [Lagoudakis and Parr, 2003]. Algorithms 4 and 5 show the LSTDQ and the LSPI algorithms respectively.

Algorithm 5 The LSPI algorithm

LSPI $(D, k, \phi, \gamma, \epsilon, \pi_0)$ // Learns a policy from samples.

Inputs:
D : Source of samples (s, a, r, s')
k : Number of basis functions
ϕ : Basis functions
γ : Discount factor
ϵ : Stopping criterion
π_0 : Initial policy, given as w_0 (default: $w_0 = 0$)

1: $\pi' \leftarrow \pi_0$ // $w' \leftarrow w_0$
2: **repeat**
3: $\quad \pi \leftarrow \pi'$ // $w \leftarrow w'$
4: $\quad \pi' \leftarrow$ **LSTD**Q $(D, k, \phi, \gamma, \pi)$ // $w' \leftarrow$ **LSTD**Q $(D, k, \phi, \gamma, \pi)$
5: **until** $(\pi \approx \pi')$ // **until**$(||w - w'|| < \epsilon)$
6: **return** π // **return** w

The approximate policy-iteration algorithm LSPI we discussed is best considered as an *off-line*, *off-policy* reinforcement learning algorithm because the samples are collected arbitrarily and its learning is separated from the execution. Although, minor modifications like in [Buşoniu et al., 2010] allow LSPI to be made *on-line* and *on-policy*. LSPI has much reduced ambiguity and possibility of errors in comparison with other approximate algorithms, mainly because of its sole focus on the representation of the state-action value function, rather than the actor parts in the actor-critic framework [Lagoudakis and Parr, 2003]. Because of these positives, LSPI has enhanced applications where initial function approximation was erroneous and noisy.

7 Conclusion

Machine Learning is a beautiful field of *artificial intelligence*, where reinforcement learning is quite the new phenomenon. Combining trial-and-error psychology with advances in mathematics gave birth to this field. Here we have seen that what we need to do is interpret any control problem as a goal-oriented problem, such that we can enumerate the value of actions into certain numeric rewards, and then carry out learning by finding actions that

maximise achieving of these rewards. We saw in section 2 that the environment setup is called a Markov Decision Process and the solution for this is an optimum policy.

In section 3 we saw how we can use the Bellman equation to find the optimum policy and used forms of the Bellman equation in the Value Iteration and Policy Iteration algorithms. It was noted that dynamic programming like these provided exact solutions for policies, but took a lot of computational space and time for processing them. In a practical situation, the MDP size is often large, hence we needed to come up with faster approximate solutions to the get the policies. That is when we started our discussion on Temporal Difference Learning Algorithms and Q-Learning algorithms in sections 4 and 5 respectively and we discussed how we can use these techniques for function approximation to eventually get the policies. Then, because of large errors in approximation using simple TD and Q-Learning, we introduced 6 and mentioned how lately it is revolutionising and improving efficiency of applications in RL.

Although, relatively the newest branch of machine learning, a lot of work has already been done in the algorithmic development, but a lot of work is still pending to convert the applications in games and software agents to applications in day-to-day physical robots that have a lot of more uncertainty than software agents. Spaces like Deep-Q-Learning, [Mnih et al., 2013], Partially Observed MDPs, Hierarchical RL [Panigrahi and Bhatnagar, 2005], and others, continues to attract research attention. Because of its trial-and-error ability like humans, it is very likely that reinforcement learning will explode in the near future and be able to solve problems that were considered unsolvable before.

References

[Abdulhai and Kattan, 2003] Abdulhai, B. and Kattan, L. (2003). Reinforcement learning: Introduction to theory and potential for transport applications. *Canadian Journal of Civil Engineering*, 30(6):981–991.

[Al-Tamimi et al., 2007] Al-Tamimi, A., Lewis, F. L., and Abu-Khalaf, M. (2007). Model-free q-learning designs for linear discrete-time zero-sum games with application to h-infinity control. *Automatica*, 43(3):473–481.

[Baxter et al., 1999] Baxter, J., Tridgell, A., and Weaver, L. (1999). Knightcap: A chess program that learns by combining td(lambda) with game-tree search. *CoRR*, cs.LG/9901002.

[Beal and Smith, 2001] Beal, D. F. and Smith, M. C. (2001). Temporal difference learning applied to game playing and the results of application to shogi. *Theor. Comput. Sci.*, 252(1-2):105–119.

[Bellman, 1954] Bellman, R. (1954). The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60(6):503–515.

[Borkar, 1998] Borkar, V. (1998). Asynchronous stochastic approximations. *SIAM Journal on Control and Optimization*, 36(3):840–851.

[Boyan and Littman, 1993] Boyan, J. A. and Littman, M. L. (1993). Packet routing in dynamically changing networks: A reinforcement learning approach. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS'93, pages 671–678, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Bradtke, 1993] Bradtke, S. J. (1993). Reinforcement learning applied to linear quadratic regulation. In *Proceedings of the 5th International Conference on Neural Information Processing Systems*, NIPS'92, pages 295–302, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Bradtke and Barto, 1996] Bradtke, S. J. and Barto, A. G. (1996). Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22(1):33–57.

[Buşoniu et al., 2010] Buşoniu, L., Ernst, D., De Schutter, B., and Babuška, R. (2010). Online least-squares policy iteration for reinforcement learning control. In *Proceedings of the 2010 American Control Conference*, pages 486–491.

[Claus and Boutilier, 1998] Claus, C. and Boutilier, C. (1998). The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 746–752, Menlo Park, CA, USA. American Association for Artificial Intelligence.

[Dayan, 1992] Dayan, P. (1992). The convergence of td(λ) for general λ. *Machine Learning*, 8(3):341–362.

[Dorigo, 1992] Dorigo, M. (1992). *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy.

[Gambardella and Dorigo, 1995] Gambardella, L. M. and Dorigo, M. (1995). Ant-q: A reinforcement learning approach to the traveling salesman problem. pages 252–260. Morgan Kaufmann.

[Gu et al., 2017] Gu, S., Holly, E., Lillicrap, T., and Levine, S. (2017). Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3389–3396.

[Howard, 1960] Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA.

[Jiang et al., 2017] Jiang, C., Zhang, H., Ren, Y., Han, Z., Chen, K., and Hanzo, L. (2017). Machine learning paradigms for next-generation wireless networks. *IEEE Wireless Communications*, 24(2):98–105.

[Lagoudakis and Parr, 2003] Lagoudakis, M. G. and Parr, R. (2003). Least-squares policy iteration. *J. Mach. Learn. Res.*, 4:1107–1149.

[Mahadevan and Connell, 1992] Mahadevan, S. and Connell, J. (1992). Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55(2):311 – 365.

[Mahmud et al., 2018] Mahmud, M., Kaiser, M. S., Hussain, A., and Vassanelli, S. (2018). Applications of deep learning and reinforcement learning to biological data. *IEEE Transactions on Neural Networks and Learning Systems*, 29(6):2063–2079.

[Matignon et al., 2007] Matignon, L., Laurent, G. J., and Fort-Piat, N. L. (2007). Hysteretic q-learning : an algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 64–69.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

[Mustapha et al., 2015] Mustapha, I., Ali, B. M., Rasid, M. F. A., Sali, A., and Mohamad, H. (2015). An energy-efficient spectrum-aware reinforcement learning-based clustering algorithm for cognitive radio sensor networks. *Sensors*, 15(8):19783–19818.

[Panigrahi and Bhatnagar, 2005] Panigrahi, J. and Bhatnagar, S. (2005). Hierarchical decision making in semiconductor fabs using multi-time scale markov decision processes. pages 4387 – 4392 Vol.4.

[Park et al., 2001] Park, K., Kim, Y., and Kim, J. (2001). Modular q-learning based multi-agent cooperation for robot soccer. *Robotics and Autonomous Systems*, 35(2):109–122.

[Sadek and Basha, 2008] Sadek, A. and Basha, N. (2008). Self-learning intelligent agents for dynamic traffic routing on transportation networks. In Minai, A., Braha, D., and Bar-Yam, Y., editors, *Unifying Themes in Complex Systems*, pages 503–510, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Samuel, 1959] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.

[Silver et al., 2017] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.

[Silver et al., 2007] Silver, D., Sutton, R., and Müller, M. (2007). Reinforcement learning of local shape in the game of Go. In *IJCAI 2007*.

[Sutton, 1984] Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis. AAI8410337.

[Sutton, 1988] Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition.

[Tesauro, 1994] Tesauro, G. (1994). Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219.

[Wang and Usher, 2005] Wang, Y.-C. and Usher, J. M. (2005). Application of reinforcement learning for agent-based production scheduling. *Eng. Appl. Artif. Intell.*, 18(1):73–82.

[Watkins, 1989] Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK.

[Watkins and Dayan, 1992] Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.