Technical Reports:

Linear Function Approximators for Reinforcement Learning

**INDEX**

[1. Markov Decision Processes 2](#_Toc520804755)

[2. Dynamic Programming 4](#_Toc520804756)

[Algorithm-1: Policy Iteration 5](#_Toc520804757)

[Algorithm-2: Value Iteration 6](#_Toc520804758)

[3. Approximate Dynamic Programming 7](#_Toc520804759)

[Algorithm-3: Trajectory Based Value Iteration 8](#_Toc520804760)

[4. Projected Bellman Residual Minimization 9](#_Toc520804761)

[Algorithm-4: Trajectory Based Policy Iteration 12](#_Toc520804762)

[5. Complexity Comparison of Model-based Algorithms 13](#_Toc520804763)

[6. Reinforcement Learning 14](#_Toc520804764)

[Algorithm-5: Q-Learning 14](#_Toc520804765)

[Algorithm-6: State-Action-Reward-State-Action(SARSA) 15](#_Toc520804766)

[Algorithm-7: LSTDQ 16](#_Toc520804767)

[Algorithm-8: Least-Squares Policy Iteration (LSPI) 17](#_Toc520804768)

[7. Complexity Comparison of Model-free Algorithms 17](#_Toc520804769)

[8. Big Picture for Algorithms Improvement 18](#_Toc520804770)

[9. Linear Representation of Value Function 19](#_Toc520804771)

[References 20](#_Toc520804772)

# Markov Decision Processes

A MDP is a tuple where S is a set of states, is a set of actions, s and s’ ,

and . (Barto., 1998)

* is a probability distribution over next states if action is executed at state s.
* of getting to state s’ by taking action a in state s.
* is the reward the agent receives current and future rewards, when the sequence (s, a, s’) occurs.

A trajectory is a sequence , each action in the trajectory is chosen according to the policy , and the Given a policy , the state-action value function, denotes the expected sum of the discounted rewards for an agent starting at state s, taking action a:

(1.1)

The factor penalize the summation of the future rewards. Similarly, the state value function, for a given policy is defined as:

(1.2)

In all possible state s , there exists an optimal policy that maximizes the expected cumulative discounted rewards:

(1.3)

And the optimal value function is defined with :

(1.4)

The optimal value function satisfies the Bellman Equation:

(1.5)

Calculating the right hand side of the Bellman equation for MDPs with infinite state and action space is challenging because:

* a unique value has to be stored for each state
* the maximization in the above equation is over all actions

For the rest of this paper, first we focus our attention on MDPs with finite state and actions spaces, and then we introduces approximations techniques that can be used for MDPs with continuous state spaces. From the definition, we know are both of the essential elements of MDPs, which the MDPs model consists of. The solving algorithms of MDPs are categorized either as:

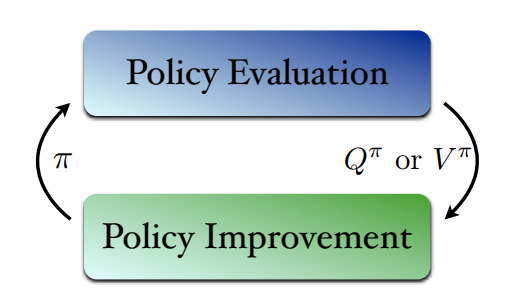
* Planning model-based MDPs: the MDPs model (i.e.  is known

We have known the MDPs model, that is to say, we can get the value of through sampling with random access. Standard MDPs planning problems can be solved directly by Dynamic Programming(DP).

* Learning model-free MDPs: the MDPs model is unknown.

In practical applications, are not available, hence we usually solve MDPs with unknown models through interaction with the environment. Exploiting ApproximateDynamicProgramming (ADP), we can make DP algorithms scalable and build Model-free algorithms for learning. [[1]](#footnote-1)

The common solvers are based on using a value function , we refer to them as value-based solvers. Value-based solvers tackle an MDP in two phases:

* Policy evaluation: the solver calculates the value function for some or all states given the fixed policy.
* Policy improvement: the solver improves the previous policy based on values obtained in the policy evaluation.

The process of iteratively evaluating and improving continues until either the policy remains unchanged, a time limit has been reached, or the change to the value function is below a certain threshold.

# Dynamic Programming

In computer science, a problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems. Problems having optimal substructure can be solved by greedy searching, but dynamic programming (DP) algorithms exploit overlapping solutions to reduce computational cost.

* **Policy evaluation**

The value function can be derived recursively as:

(2.1)

For a finite state MDP with , the vector represents the value function of the policy . Let matrixbe defined using and be defined using , thus we have the form:

(2.2)

Define an operator as the Bellman operator *(*) applied to the value function:

(2.3)

The problem of evaluating policy translates into finding the fixed-point of operator , which has a closed form solution:

(2.4)

Expanding the right hand side using the Neumann series, we obtain:

(2.5)

What guarantees the convergence of the series is that, where , which refers to the operator norm of matrix . Iterative process using (2.1) is common with the computation in 2.5.

* **Policy improvement**

As for policy improvement, the new policy is updated by selecting the action that is “greedy” with respect to the calculated values:

(2.6)

Putting (2.4) and (2.6) together, we arrive at the policy iteration algorithm (Howard., 1960).

* **Computational complexity**

Policy iteration is guaranteed to stop in a polynomial number of iterations |S|×|A| for a fixed value of gamma (Ye., 2011) and reach the optimal solution (Howard., 1960). From a practical standpoint, this algorithm is not scalable because storingrequires memory and solving (2.4) takes

[[2]](#footnote-2)time.

Hence the exact policy evaluation step in Algorithm-1 is often done iteratively within a given threshold . The smaller , the better approximation at the cost of more computation. Algorithm-1 only requires memory and computation per iteration.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm-1: Policy Iteration | | | | | Time | Memory |
| **Input**: MDP,  **Output**: | | | | | | |
| 1 |  | | | | | |
| 2 |  | | | | | |
| 3 | Changed **True** | | | | | |
| 4 | **While** changed **do** | | | | | |
| 5 |  |  | | | | |
| 6 |  | **repeat** | | | | |
| 7 |  |  | **for** | | | |
| 8 |  |  |  |  | | |
| 9 | Bellman backup | | |  |  |  |
| 10 |  |  |  |  | | |
| 11 |  | **Unti** | | | | |
| 12 |  | **for** | | | | |
| 13 |  |  |  | |  |  |
| 14 |  | Changed | | | | |
| 15 |  |  | | | | |
| 16 | return | | | | | |

Line-9 of Algorithm-1 is called “Bellman backup”. Another idea to do the loop of switching between policy evaluation and policy improvement: update the policy after every single “Bellman backup”. In Algorithm-2, “Bellman backup” should use the best possible action for backup rather than a fixed policy (Bellman., 1957):

(2.7)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm-2: Value Iteration | | | | Time | Memory |
| **Input**: MDP,  **Output**: | | | | | |
| 1 |  | | | | |
| 2 | **repeat** | | | | |
| 3 |  | ***for*** | | | |
| 4 |  |  |  | | |
| 5 |  |  |  |  |  |
| 6 |  |  |  |  |  |
| 7 |  |  |  | | |
| 8 | **Unti** | | | | |
| 9 | return | | | | |

This equation unifies equation 2.1 and 2.6 and gives the value iteration algorithm (Bertsekas D. P., 1976)[[3]](#footnote-3).

Obviously, value iteration improves the policy much more frequently than policy iteration and reduces the main loop complexity from to per iteration. The memory requirement of value iteration is the same as policy iteration .

In theory value iteration may require more iterations than policy iteration (Ye., 2011), in practice, value iteration requires less compared to policy iteration.

# Approximate Dynamic Programming

We focus on MDPs with large, yet finite, state spaces with a small number of actions (. This assumption is often met in practice. For example in 9×9 Go, and . Approximate dynamic programming aims on reducing memory sizes and computations. There are four scaling problems in Algorithm-2.

* **Problem-1**: stores an action for each state (line 6)
* **Problem-2**: There is a loop over all possible states (line 3)
* **Problem-3**: Both the bellman backup (line 5) and the policy update consider all possible next states, which in the worst case can be .
* **Problem-4**: stores a unique parameter for every state of the MDP (line 5).

**Problem-1** can be solved by storing the policy implicitly through using an action-value function. Hence, we have greedy policy:

(3.1)

Note that switching from *V* to *Q* increases the memory requirement for storing the value function by factor of , which is assumed to be small. As for **Problem-2**, we use trajectory-based sampling to execute Bellman backups on states that are visited under good policies. (A. Barto, 1995) We adopt -greedy policy to generate trajectories instead of sweeping through the whole state space, where 2.1 holds for all states asymptotically.

(3.2)

Locality property inspires us to exploit empirical estimation to reduce computation of **Problem-3**, as the estimate becomes exact with probability one:

(3.3)

*Q* function holds sufficient information for deriving a policy without needing extra storage. The savings of Problem-1 come from a parametric representation of *Q* requiring less number of parameters in storing and updating than the number of states. A classical example of *Q* is linear representation:

(3.4)

Where *(s, a)* denotes the value of feature for state-action pair . The feature function maps each state-action pair to a vector of feature value and To avoid the shortage of **Problem-4**, we update each timeto complete “bellman backup” 3.3 with linear function *Q* :

(3.5)

Let , thus we have:

(3.6)

Hence, the update rule takes the following form:

(3.7)

Where is the step size parameter and . Combining the above approach and the solutions to Problem1-4, we have an algorithm called trajectory based value iteration (TBVI) shown in Alogrithm-3.

* Complexity analysis.

TBVI has an memory requirement and iteration complexity, where T is the

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm-3: Trajectory Based Value Iteration | | | | Time | Memory |
| **Input**: MD*P,*  **Output**: | | | | | |
| 1 |  | | | | |
| 2 | **while** *time left* **do** | | | | |
| 3 |  | ***for*** | | | |
| 4 |  |  | Create *N* samples: | | |
| 5 |  |  |  |  |  |
| 6 |  |  |  |  |  |
| 7 |  |  |  | | |
| 8 | return greedy with respect to *Q* | | | | |

maximum length of a trajectory and n refers to the dimension of .

# Projected Bellman Residual Minimization

In this section, we introduce a new algorithm with more compact memory and computation in evaluating the policy. Similar to the linear function approximation in Algorithm-3(TBVI), we now assume value function has a form of linear function:

(4.1)

Define as an approximation of:

(4.2)

Best approximate value function in the space spanned by the base functions(i.e. column space of ) would be solved by finding the solution to the following minimization problem:

(4.3)

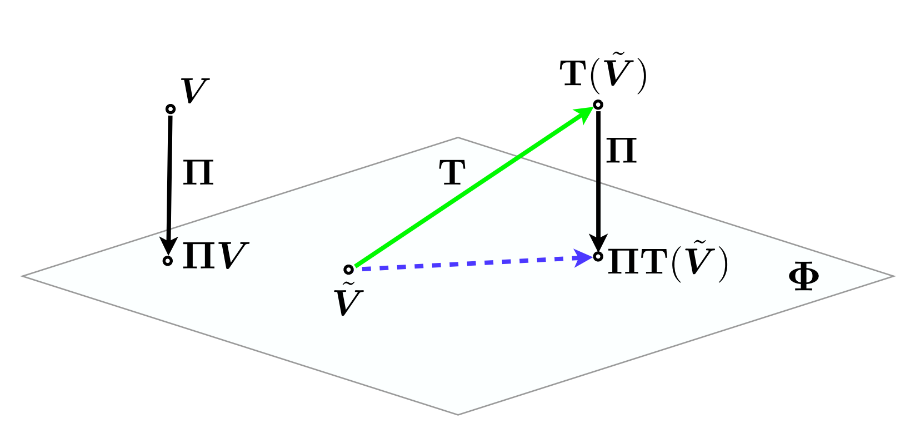
Where is the element of vector and is a non-negative weight vector specifying the importance of each state. The solution of 4.3 has an analytic form via manual deduction:

(4.4)

Where defined as a matrix, with on its diagonal . Intuitively states that are visited more often should have higher weights, penalizing the error correspondingly. If we have a state transition matrix binding to an fixed policy, satisfies the following conditions:

(4.5)

Calculating steady probabilistic transition matrix can be challenging, we will use a more practical weighting scheme.

In general, bellman operator improves the approximation through moving the vector out of the span of , whileplays the role to project the vector back to the space spanned by . Correspondingly, there are two major metrics used in the literature to define the best approximated value function: 1) the projected Bellman error(blue line); 2) the Bellman error(green line).

Our goal is to find the approximation that minimizes the norm of the blue dashed line in the above figure formulated as:

(4.6)

The minimizer can be found by forcing the two points and to be equal:

(4.8)

(4.7)

The solution of projected Bellman residual minimization, also known as the Least Squares Temporal Difference solution(LSTD) (Barto, 1996). The calculated above is known as the LSTD solution and can be far from, however the approximation error is bounded (Roy., 1999) (Bertsekas H. Y., 2010):

(4.9)

When is rank deficient, regularization techniques are used (Ng, 2009). In the tabular case where , that is, features simply indicate which state the system is in, 4.7 is identical to 2.4.

Similar to the derivation of 2.1, can be written recursively as:

(4.10)

As for a fixed policy, removing notation and the above equation can be written as:

(4.11)

where,

(4.16)

(4.15)

(4.14)

(4.13)

(4.12)

becomes with overloaded on its diagonal. Consequently all derivations of LSTD remain intact by calculatinginstead of.

Due to the same reason of exceeded memory requirement, it’s not practical for LSTD algorithm to use with rows, we can collect samples by following a fixed policy , that is, a sampling trajectory where is the initial state and and for *.*

(4.17)

The weight vector can be estimated as follows:

(4.22)

(4.20)

(4.21)

(4.19)

(4.18)

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm-4: Trajectory Based Policy Iteration | | | Time |
| **Input**: MD*P,*  **Output**: | | | |
| 1 |  | | |
| 2 | **while** *time left* **do** | | |
| 3 |  | Create | |
| 4 |  | Calculate andusing equation 4.21 and 4.22 |  |
| 5 |  | (Use regularization if ill-posed) |  |
| 8 | return greedy with respect to *Q* | | |

* Table of computational cost

|  |  |  |  |
| --- | --- | --- | --- |
| Calculation | Time | Memory | Equation |
|  |  |  | 4.1 |
|  |  |  | 3.1 |
|  |  |  | 4.18 |
|  |  |  | 4.17 |
|  |  |  | 4.21 |

We have. For regularization, is added to the matrixbefore the inversion, where is a small scalar. Some reasons for divergence can be found in J. Boyan. (Moore, 1995)

# Complexity Comparison of Model-based Algorithms

* Per iteration cost

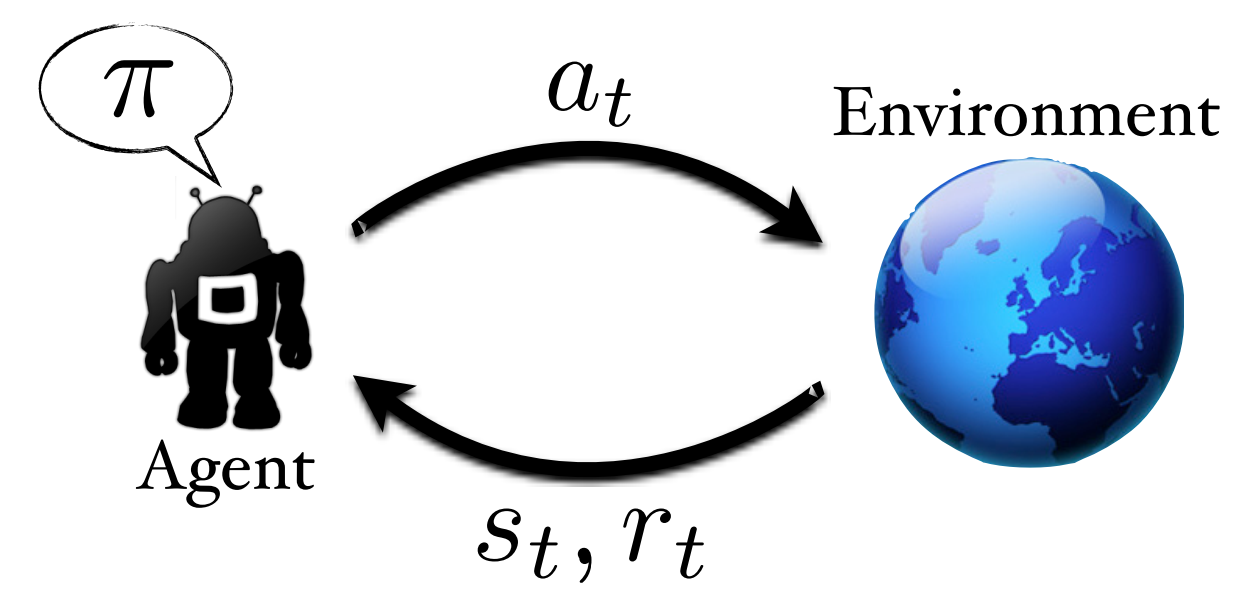
|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Iteration Complexity | Memory | Algorithm |
| Policy Iteration |  |  | 1 |
| Value Iteration |  |  | 2 |
| TBVI |  |  | 3 |
| TBPI |  |  | 4 |

Often, the size of the state space is very large for real-world problems, which eliminates methods with computational complexity or memory dependent on , such as Algorithm-1 and Algorithm-2. While often we have , for some domains with a large number of continuous actions, the methods having a dependency on are not sustainable either, such as Algorithm-3 and Algorithm-4. We need further approximation to eliminate (A. Antos, 2007)

The cost table does not say anything about the number of iterations required to obtain a reasonable policy, nor about their behavior if they diverge. For example, some methods might require more time to finish one iteration, but they may require fewer iterations in total to find good policies.

# Reinforcement Learning

The goal of RL methods is to achieve high reward, usually by solving MDPs with unknown models (i.e. , solely through interaction with the environment. Many RL techniques can be seen as dynamic programming methods where samples are obtained through sequential interaction.



In RL, the agent does not have random access to all states at once, for example, in a navigation task, sampling from the environment is only available through interaction. Interacting with the environment is time sensitive, the per-time-step complexity plays a critical role. Since the task is also sensitive to the observing samples, a trade-off between exploration and exploitation should also be met (L. Li, Online exploration in least-squares policy, 733–739).

We describe two classical RL methods (Q-learning and SARSA) derived from TBVI and two batch RL algorithms (LSTDQ and LSPI) derived from TBPI.

* Q-learning

In RL literature, only one next state can be sampled from each state. Hence must be 1 to allow for learning from trajectory. By setting to be 1 in line-5 of Algorithm-3, we get an online algorithm known as Q-learning.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm-5: Q-Learning | | | Time | Memory |
| **Input**: MD*P,*  **Output**: | | | | |
| 1 |  | | | |
| 2 |  | | | |
| 3 | **while** *time left* **do** | | | |
| 4 |  | Take action and receive reward *r* and next state *s’* | | |
| 5 |  |  |  |  |
| 6 |  |  | | |
| 7 |  |  |  |  |
| 8 |  |  |  |  |
| 9 | return greedy with respect to *Q* | | | |

The calculated on line-6 of Algorithm-5 highlights the difference between the better estimate of the Q function based on a single interaction*,* , and the current estimate . This quantity is called the temporal difference (TD) error in the literature. Under certain conditions, Q-Learning has been shown to converge with function approximation. (F. S. Melo, 2008)

* off-policy and on-policy algorithm

Q-Learning is an off-policy algorithm, that is, the current policy (i.e. greedy searching on the value

function) is not identical to the policy , which generates the samples for Q-Learning to update the value function.

* SARSA

Compared to Algorithm-5 Q-Learning, SARSA makes both the sampling and learning polices identical, that is, is calculated based on rather than . SARSA is an on-policy algorithm learns the value function of the policy it is currently implementing. It has the same per-time-step computational complexity as Q-Learning, but has convergence guarantees under milder conditions when used the linear value function approximation. (F. S. Melo, 2008)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm-6: State-Action-Reward-State-Action(SARSA) | | | Time | Memory |
| **Input**: MD*P,*  **Output**: | | | | |
| 1 |  | | | |
| 2 |  | | | |
| 3 | **while** *time left* **do** | | | |
| 4 |  | Take action and receive reward *r* and next state *s’* | | |
| 5 |  |  |  |  |
| 6 |  |  |  |  |
| 7 |  |  | | |
| 8 |  |  |  |  |
| 9 |  |  |  |  |
| 10 | return greedy with respect to *Q* | | | |

* LSTDQ

Because samples can be generated only by following a trajectory, we set in 4.18 and 4.19. Then, can be calculated as:

(6.1)

The number of samples collected, , controls the accuracy of estimates for and. (L. Li, Online exploration in least-squares policy, 2009a). The complexity of memory and time requirement per iteration is identical to the Algorithm-4 TBPI.

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm-7: LSTDQ | | | Time |
| **Input**: MD*P,*  **Output**: | | | |
| 1 |  | | |
| 2 | **while** *time left* **do** | | |
| 3 |  | Create |  |
| 4 |  | Calculate andusing equation 4.21, 4.22 and 6.1 |  |
| 5 |  | (Use regularization if ill-posed) |  |
| 6 | return greedy with respect to *Q* | | |

An important fact about this method is that samples generated to estimate andare discarded after each cycle, making this algorithm inefficient for domains with an expensive cost of sample acquisition.

* LSPI

Least-squares policy iteration(LSPI) selects samples to evaluate the new policy by switching to , where is greedy with respect to the most recent Q values. The per-iteration complexity kept unchanged. Notice that given a fixed set samples, LSPI does not necessarily improve the policy between each iteration, hence it lacks convergence guarantee, but it will not diverge either as in the worst case it will switch between polices. (L. Li, Online exploration in least-squares policy, 2009a) (Schaal, March 2008)

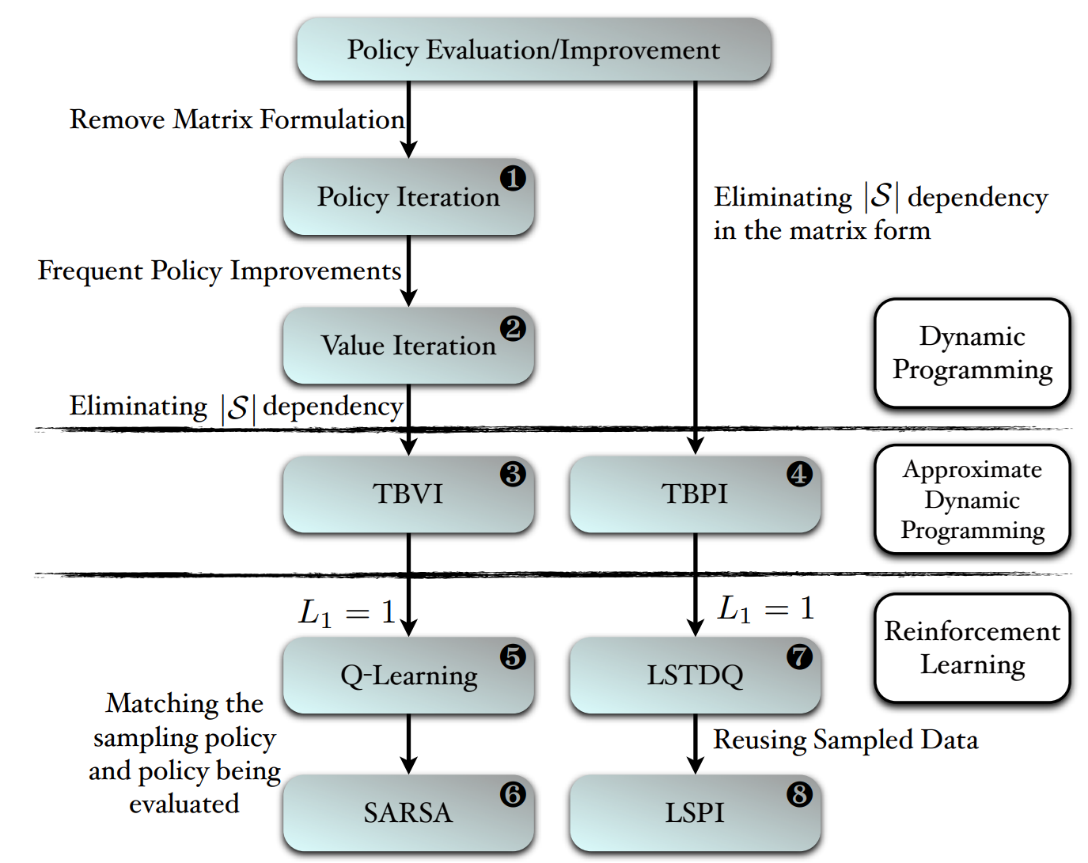
|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm-8: Least-Squares Policy Iteration (LSPI) | | | Time |
| **Input**: MD*P,*  **Output**: | | | |
| 1 |  | | |
| 2 | Create | | |
| 2 | **while** *time left* **do** | | |
| 3 |  | Calculate andusing equation 4.21, 4.22 and 6.1 |  |
| 4 |  | (Use regularization if ill-posed) |  |
| 5 |  | **For** all |  |
| 6 | return greedy with respect to *Q* | | |

# Complexity Comparison of Model-free Algorithms

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Iteration Complexity | Memory | Algorithm |
| Q-Learning |  |  | 5 |
| SARSA |  |  | 6 |
| LSTDQ |  |  | 7 |
| LSPI |  |  | 8 |

Notice that online algorithms (Q-Learning and SARSA) provide cheap complexity per iteration, however, if these methods are not paired with a powerful exploration strategy, their sample complexity may still make their total runtime prohibitive. If sample complexity is the only concern, batch algorithms (LSTDQ and LSPI) are often preferred over online methods. (Szepesvári, 2010)

# Big Picture for Algorithms Improvement



All of these algorithms we discussed including dynamic programming and reinforcement learning ones were derived from the unified idea of policy evaluation/improvement. By eliminating the matrix operator 2.4 for policy evaluation, and performing policy improvement after each full evaluation, we first introduced policy iteration. By increasing the frequency of policy improvements, we arrive at value iteration. In order to scale DP techniques to MDPs with large state-spaces, we eliminate all memory and computational dependencies, inducing trajectory based value iteration (TBVI). Using the same approximate DP techniques, we introduce trajectory based policy iteration (TBPI) with matrix operations. Under the reinforcement learning setting, we can’t be random access to the MDP model. We first showed how the Q-Learning algorithm can be derived from TBVI by simply using a single sample collected along trajectories to estimate the Bellman backup By using the same policy to generate samples and be evaluated in an iteration(on-policy), we arrive at the SARSA algorithm. By extending TBPI to estimate and with one sample along the trajectory, we obtain the LSTDQ algorithm. Finally, by changing to , we could reuse collected samples for policy iteration, arriving at the LSPI algorithm.

# Linear Representation of Value Function

In classical Inverted Pendulum domain problem, base features can be formed for this domain by discretizing and into a certain number of buckets separately. There are three discrete actions in this domain: 1) apply no force , 2) push the pendulum clock-wise, 3) push it counterclockwise. A negative reward is given when the pendulum hits the horizon and zero reward is given otherwise. Due to the physical characteristics can’t be formularized by a simple linear combination of the base features, a good approximation is possible by overlaying a very fine grid across the base features and using each grid cell as a feature in the domain, forming a tabular representation.

* Tabular representation

The tabular representation maintains a unique weight for each state-action pair. In discrete

domains, this is an exact representation. In continuous domains, like the Inverted Pendulum above, it

corresponds to a discretization of the state space by tiling the state space with uniform grid cells and using each cell as a feature.

* Fixed sparse representation

Consider the set , where corresponds to the possible value for the

dimension of the state space, fixed sparse representation for state are created as follows:

(9.2)

(9.1)

Which amounts to a total features per action. For each dimension, refers to the number of distinct values the state space can take.[[4]](#footnote-4) Compared to the tabular representation of the Inverted Pendulum domain, only weights need to be learned instead of .

# References

A. Antos, R. M. (2007). Fitted Q-iteration in continuous action-space. *Neural Information Processing Systems Conference*.

A. Barto, S. B. (1995). *Artificial Intelligence*, 72:81–138.

Barto, S. J. (1996). *Journal of Machine Learning Research (JMLR)*, 22:33–57.

Barto., R. S. (1998). *Reinforcement Learning: An Introduction.* MIT Press.

Bellman., R. E. (1957). *Dynamic Programming.* Prineton, NJ: Princeton University Pres.

Bertsekas, D. P. (1976). *Dynamic Programming and Optimal Control.* AP.

Bertsekas, H. Y. (2010). Error bounds for approximations from projected linear. *Math. Oper. Res*, 35(2):306–329.

F. S. Melo, S. P. (2008). An analysis of reinforcement learning. *In International Conference on Machine Learning*, 664–671.

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

L. Li, M. L. (2009a). Online exploration in least-squares policy. 733–739.

Moore, J. B. (1995). *Neural Information Processing Systems (NIPS)*, 369–376.

Ng, J. Z. (2009). Regularization and feature selection in least-squares. *In International Conference on Machine Learning*, 521–528.

Roy., J. N. (1999). Average cost temporal-difference learning. *Automatica*, 35(11):1799 – 1808.

Schaal, J. P. (March 2008). Natural actor-critic. (pp. 71:1180–1190). Neurocomputing.

Szepesvári, C. (2010). Algorithms for Reinforcement Learning.

Tsitsiklis, D. B. (1996). Neuro-Dynamic Programming. *Athena Scientific*, Belmont, MA.

Ye., Y. (2011). The simplex and policy-iteration methods are strongly polynomial for the markov decision problem with a fixed discount rate. *Math. Oper. Res.*, pp. 36(4):593–603.

1. In this paper, model-based and model-free refer to planning algorithms and learning algorithms respectively. [↑](#footnote-ref-1)
2. Advanced techniques for matrix inversion with computational complexity. [↑](#footnote-ref-2)
3. Bertsekas has shown that if the difference of the value function between two successive iterations of value iteration (the Bellman residual) is less than , then the difference between the value functions of the greedy policy with respect to the current values and the optimal policy is no more than . [↑](#footnote-ref-3)
4. continuous dimensions can be discretized into buckets. [↑](#footnote-ref-4)