An Introduction to Temporal Difference Learning

Florian Kunz
Seminar on Autonomous Learning Systems
Department of Computer Science
TU Darmstadt
fkunz@sim.tu-darmstadt.de

Abstract

Temporal Difference learning is one of the most used approaches for policy evaluation. It is a central part of solving reinforcement learning tasks. For deriving optimal control, policies have to be evaluated. This task requires value function approximation. At this point TD methods find application. The use of eligibility traces for backpropagation of updates as well as the bootstrapping of the prediction for every update state make these methods so powerful. This paper gives an introduction to reinforcement learning for a novice to understand the TD(λ) algorithm as presented by R. Sutton. The TD methods are the center of this paper, and hence, each step for deriving the update function is treated. Starting with value function approximation followed by the Bellman Equation and ending with eligibility traces. The further enhancement of TD in form of linear-squared temporal difference methods is treated. Both methods are compared in respect to their computational cost and learning rate. In the end an outlook towards application in control is given.

1 Introduction

This article addresses the machine learning approach of temporal difference. TD can be classified as an incremental method specialized in predicting future values for a partially unknown system.

Temporal difference learning is declared to be a reinforcement learning method. This area of machine learning covers the problem of finding a perfect solution in an unknown environment. To be able to do so, a representation is needed to define which action yields the highest rewards. Introducing value functions gives the ability to represent rewards of actions and their distribution in the future. As machine learning is likely to be applied on huge systems the value functions have to be represented by an approximation. The use of linear functions for this task has the advantage that the resulting systems can be solved more easily.

TD methods calculate value functions, or approximations of these. The methods differ from other approaches as they try to minimize the error of temporal consecutive predictions instead of an overall prediction error. A core mechanism to achieve this is to rewrite the value function update in form of a Bellman Equation. Allowing the approach to enhance the prediction by bootstrapping. This effect reduces the variance of the prediction in each update step. Giving the methods a property, important for value approximation. Another trick of TD methods to achieve backpropagation of updates, while saving memory, is the application of a eligibility vector. The sample trajectories are utilized more efficiently, resulting in good learning rates.

LSTD methods are an enhancement of the original methods, using the mean squared error instead of a gradient descent approach to minimize the error of the prediction. the LSTD algorithm eliminates the need for a step size factor, while at the same time increasing the learning rate. However the number of computations needed is significantly higher compared to the TD method.

The article is organized as follows: the second section gives an overview of reinforcement learning pointing out the problem for tasks of this character. To establish a common ground for handling TD methods, basic definitions of this field are given. Section 3 treats temporal difference methods for prediction learning, beginning with the representation of value functions and ending with an example for an TD(λ) algorithm in pseudo code. Section 4 introduces an extended form of the TD method the least-squares temporal difference learning. Ending with Section 5 by taking an outlook how TD methods can be used in control problems.

2 Introduction to Reinforcement Learning

The basic idea of reinforcement learning is to utilize the huge information gain of learning in a previously unknown area. The application of supervised learning methods is limited to systems that can be interpreted clearly using samples evaluated by an external supervisor. Reinforcement learning does not depend on preprocessed data, it derives knowledge from its own experience.

Reinforcement learning deals with the problem of mapping states of a system to actions that will maximize a numerical reward for the agent. Every action effects the future rewards an agent can achieve, rendering reinforcement learning such a complex problem.

A main characteristic of reinforcement learning is the trade-off between exploration and exploitation. On the one hand the learner wants to reliably maximize the reward of future actions, which is done by picking a known action with the highest reward so far. On the other hand the agent should explore new actions in order to find the action resulting in the highest reward. The unique feature of reinforcement learning is the principle of trail and error in order to discover the best actions. Another feature is the handling of delayed rewards. An action taken by the learner may result not only in an immediate reward, but also have effect on all future rewards. This effect results in a very complex dynamical system.

The core problem of reinforcement learning is evolving a mapping of states to actions yielding the highest reward. The policy π of an agent describes the mapping from states to actions as a set of probabilities. For each state s it holds the probabilities $\pi(a|s)$ for selecting a possible action a .

Reinforcement learning describes a class of problems, it is not characterized as a class of learning methods. A method for supervised regression learning may also fit the requirements of reinforcement learning. In order to be able to apply these machine learning algorithms to systems of states, actions and rewards, these have to be modeled. Typically a Markov Decision Process is used for this purpose.

2.1 Markov Decision Process

A process with the Markov Property is called memoryless as the transition probabilities for future states depends solely on the current state of the system. None of the preceding states will give an advantage in form of further knowledge when predicting the next state and the corresponding reward. In **[SuttonBarto98]** this fact is illustrated with a game of checkers. In this case the state of the process can be modeled by the configuration of all pieces. Further Knowledge about the sequence of positions which led up to this configuration do not influence the future of the game.

Formally the Markov Property can be displayed under the assumption of a process with finite states, actions and rewards. While in the general case the probability distribution of future states is a function of all past states

$$\mathcal{P}_G(s_{t+1} = s', r_{t+1} = r \,|\, s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \ldots, s_0, a_0),$$

in a process having the Markov Property it is a function only of the current state

$$\mathcal{P}_M(s_{t+1} = s', r_{t+1} = r \,|\, s_t, a_t),$$

for all s', r, s_t, a_t and all possible $s_{t-1}, a_{t-1}, r_{t-1}, \ldots, s_0, a_0$. A process has the Markov Property, if $\mathcal{P}_G$ equals $\mathcal{P}_M$ for all possible states and past states.

A reinforcement learning task can be modeled as a Markov Decision Process (MDP), which is a stochastic process satisfying the Markov Property. The MDP is composed of state and action spaces. A finite Markov Decision Process only holds a countable number of states and actions.

Whether a problem runs infinitely long or is guaranteed to terminate in an finite amount of time affects the choice of MDP. The model can be episodic, meaning it will at some point enter a terminal state and never leave it again. The sequence of states leading up to entering the terminal state is called episode. In case the terminal state will be reached after a *fixed* number of states, this is called a *finite-horizon* task. For a *indefinite-horizon* task the length of the episode is not limited by a *fixed* number. Problems that could run for infinite time without termination are called *infinite-horizon* tasks and need to be captured in a ergodic model. In contrast to episodic MDPs there is the possibility that the system will never enter a terminal state. This circumstance demands that a subsection of the model has the characteristic that each state can be reached from any other state.

This article is restrained to the definition of a finite MDP. It is formally defined as $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, consisting of sets of states $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ and actions $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$. $\gamma \in [0, 1]$ denotes a discount factor, for discounting future rewards. The set

$$\mathcal{P}^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$$

holds the probability for a transition from s to s' when taking action a. Similarly

$$\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

holds the rewards for the transition $(s \xrightarrow{a} s')$. $\mathcal{P}$ is also called a Markovian transition model. For a given state-action pair the expected reward can be calculated to

$$\mathcal{R}(s, a) = \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} \tag{1}$$

2.2 Value Functions

In reinforcement learning, the learner needs information about the quality of a state in order to find an optimal policy. A value function V^π represents the expected future reward for a current state, if the agent follows a policy π. Based on the introduced MDP the state-value function can be formally defined as

$$V^\pi(s) = E_\pi \{R_t | s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, s_t = s \right\}. \tag{2}$$

More specific, the action-value function describes the expected reward for taking action a in state s and then following policy π

$$Q^\pi(s, a) = E_\pi \{R_t | s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, s_t = s, a_t = a \right\}. \tag{3}$$

While the state-value function calculates the sum of all possible actions multiplied by their probability, the action-value function calculates the reward for an explicit action a at time step t in state s. The action-value function is based on a proper policy of state-action pairs. Policy evaluation is based on this calculation.

The connection between the state-value function and the action-value function is displayed by

$$V^\pi(s) = \sum_a \pi(s, a) Q^\pi(s, a). \tag{4}$$

V^π denotes a mean expectation of the future reward over all possible actions a, weighted by the probability distribution $\pi(a|s)$. Q^π represents an actual reward for a fixed choice of actions.

Taking a closer look at the state-value function it is not only possible to calculate the expected reward for a state at a given time and future policy, but also to do this incremental, basing each calculation on the result of the preceding update.

$$\begin{aligned} V^\pi(s) &= E_\pi \{r_t + \gamma V^\pi(s_{t+1}) \,|\, s_t = s\} \\ &= \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s') \right] \end{aligned} \tag{5}$$

Equation (5) shows the Bellman Equation. Solving the linear system returns the explicit values for V^π. However for large trajectories the computation is expensive and requires knowledge of the complete trajectory to the last state.

2.3 Linear Function Approximation

To be able to apply reinforcement learning on real problems there is need for a representation that is capable of holding all possible states in an efficient way. Until now, V was assumed to be represented as a lookup table without awareness for use of memory. For tasks with a huge number of states a representation is needed that generalizes well, while being memory efficient.

In machine learning, the standard approach for representing values is a linear function approximation. Introducing a feature vector $\phi(s)$, holding linear basis functions, and a parameter vector θ, used to align the functions, the value function can be approximated as

$$V(s) \approx V_\theta(s) = \theta^T \phi(s) \tag{6}$$

The dimensionality of the parameter vector $\theta \in \mathbb{R}^K$ is equal to the number of features K of the function $\phi(s)$. For most tasks the features are hand picked, using expert knowledge. The major advantage using linear function approximation is the dimensionality reduction from the number of states to a fixed number of basis functions K. However the dimensionality reduction also causes an approximation error. $\phi(x)$ can only capture a finite number K of linear functions, resulting in a limited precision and hence an error in state representation.

3 Temporal Difference in prediction learning

Temporal Difference methods find application in reinforcement learning tasks. In **[SuttonBarto98]**, three classes are listed to solve these tasks: *Dynamic Programming*, *Monte Carlo methods* and *Temporal Difference Learning*, of which the latter is called the most central and novel idea in reinforcement learning.

Dynamic Programming is based on the Bellman Equation and breaks down a problem into subproblems. Dividing a big task into smaller steps, this approach is depending on a perfect model of the environment.

Monte Carlo methods do not need a model of the learning environment. From experience in form of sequences of state-action-reward-samples they can approximate future rewards. However the methods only update after a complete sequence, when the final state is reached. The Markov Return is defined as:

$$\mathcal{R}_t = \sum_{i=t}^{T-1} \gamma^{i-t} R_{i+1} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T. \tag{7}$$

Temporal Difference methods combine both procedures - there is no need for a model of the learning environment and updates are available at each state of the incremental procedure. The method learns directly from the raw experience in a partially unknown system with each recorded sample.

An often used example is the weather forecast for a future day, lets say Saturday. In **[Sutton88]** it is pointed out that learning to predict Saturday's weather from a earlier day by evaluating the prediction using the actual outcome is a supervised learning approach. In this scenario the change of weather over the course of time leading up to Saturday is ignored, only the forecast from the time the prediction was made is compared to the weather at Saturday. Taking the approach of Temporal Difference methods all days from the time of prediction up to Saturday are taken into account. In the process of learning, this means that the prediction of Saturday's weather at one day is compared to the succeeding prediction and an increment is calculated to adjust the prediction. TD methods are called bootstrapping methods, as they do not learn by the difference to the final outcome but the difference between each update step. Instead of a single update, TD methods calculate $T-1$ updates for a episode of T time steps.

The TD method aims to achieve a approximation V_θ^π as close to the value function V^π as possible. The error of the approximation can be measured by the mean squared error function

$$\mathrm{MSE}(\theta) = \frac{1}{n} \sum_{i=1}^{n} (V_\theta^\pi(s_i) - V^\pi(s_i))^2. \tag{8}$$

By minimizing the mean squared error the approximation of the value function can be optimized. As $V^\pi(s)$ is unknown, it is estimated by applying Equation (5) on the current approximation V_θ^π

$$V^\pi(s_t) \approx E\left\{r_t + \gamma V_\theta^\pi(s_{t+1})\right\}. \tag{9}$$

The application of the Bellman Equation is the core idea of Temporal Difference Learning and allows to calculate the error denoted by Equation (8). However analytic computation of the minimum of the error is not possible for systems with huge state spaces. Instead, a local minimum is searched numerically by *Stochastic Gradient Descent*. The method calculates new search positions θ' by following a approximation of the gradient of the error function. A learning rate factor α is used to adjust the step size of the SGD method and prevent overshooting.

$$\begin{aligned}\theta' &= \theta - \alpha\nabla\mathrm{MSE}(\theta)\\ &= \theta - \alpha\left[V_\theta^\pi(s_i) - V^\pi(s_i)\right]\nabla_{\theta_t} V_{\theta_t}(s_t).\end{aligned} \tag{10}$$

The value function approximation applied on Equation (10) results in a sum, calculating the gradient. To reduce the number of computations needed, the gradient is approximated by $\phi(s_t)$. The update function of the TD learning method can be displayed as:

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha\delta_t e_t\\ \delta_t &= r_{t+1} + \gamma V_{\theta_t}(s_{t+1}) - V_{\theta_t}(s_t)\\ e_t &= \phi(s_t).\end{aligned} \tag{11}$$

The vector δ_t denotes the temporal difference. By comparing the prediction at the current state $V_{\theta_t}(s_t)$ to the prediction of the next state $V_{\theta_t}(s_{t+1})$ the temporal difference δ_t is used for adapting the prediction itself. This behavior is called bootstrapping. The variance of the approximation is limited at each update by this correction.

The vector e_t denotes the approximation of the gradient $\nabla_{\theta_t} V_{\theta_t}$. It can be pictured as the algebraic link along which the update is propagated. For this formulation the update resulting from the TD error only effects the current state s_t. Learning by this equation takes long, as the rewards only propagate one state with each update.

To speed up the process of reward propagation, eligibility traces are introduced. This vector allows the method to carry rewards backward over the sampled trajectory without the need to store the trajectory itself. The reach of this effect is depending on the factor $\lambda \in [0, 1]$. The eligibility traces replace the approximation of the gradient in Equation (11) with

$$e_t = \sum_{k=t_0}^{t} \lambda^{t-k} V_{\theta_{\theta_t}}(s_k).$$

The backpropagation of rewards using eligibility traces is a basic mechanism of TD methods. The factor λ determines the degree to which extend the changes are propagated. For application on real tasks the value of λ is of such importance that the algorithm is named TD(λ). The algorithm is displayed as pseudo code in *Algorithm 1* following **[Boyan2002]** for the example of linear approximation of the undiscounted value function of a fixed proper policy.

The TD method is modified by λ so much that for λ=1 the method yields the same results as supervised linear regression learning on Monte Carlo returns, while a λ=0 results in a one-step lookahead. The behavior for λ=1 is problematic for value function approximation. Monte Carlo returns represent all states traversed by looking at the complete trajectory from t to T, as shown in Equation (7). A large variance results from observing the long stochastic sequence of all future states. On the opposite the TD(0) algorithm has low variance, but uses samples inefficiently, as with each update the reward only propagates to the next state.

The TD(λ) algorithm has to be adapted for each task by tuning the step size parameter α to achieve a low error. Furthermore the choice of the λ value has influence on the error, as well as the efficiency of the use of samples. The step size factor is also a source of error. In a worst case scenario a poorly chosen α factor corrupts the minimum search by SGD. In the next chapter the dependency on a step size factor is eliminated by the introduction of linear least-squares temporal difference learning.

Algorithm 1: TD(λ) for approximate policy approximation:

Data: a simulation model for a proper policy π in MDP $\mathcal{M}$.
a feature function $\boldsymbol{\phi} : \mathcal{S} \rightarrow \mathbb{R}^K$, mapping states to feature vectors, $\boldsymbol{\phi}(T) \stackrel{\text{def}}{=} 0$;
a parameter $\lambda \in [0, 1]$; and
a sequence of step sizes $\alpha_1, \alpha_2, \ldots$ for incremental coefficient updating.

Output: a coefficient vector $\boldsymbol{\theta}$ for which $V^{\pi}(s) \approx \boldsymbol{\theta}^T \boldsymbol{\phi}(s)$.

Set $\boldsymbol{\theta}$:= 0 (or an arbitrary initial state), t := 0.
for n := *1,2,...* **do**
  Set $\boldsymbol{\delta}$:= 0.
  Choose a start state $s_t \in S$.
  Set $\boldsymbol{e}_t := \boldsymbol{\phi}(s_t)$.
  while $s_t \neq T$ **do**
    Simulate one step of the process, producing a reward r_t and next state s_{t+1}.
    Set $\boldsymbol{\delta} := \boldsymbol{\delta} + \boldsymbol{e}_t(r_t + (\boldsymbol{\phi}(s_{t+1}) - \boldsymbol{\phi}(s_t))^T \boldsymbol{\theta})$.
    Set $\boldsymbol{e}_{t+1} := \lambda \boldsymbol{e}_t + \boldsymbol{\phi}(s_t)$.
    Set $t := t + 1$.
  end
  Set $\boldsymbol{\theta} := \boldsymbol{\theta} + \alpha_n \boldsymbol{\delta}$.
end

4 Least-Squares Temporal Difference Learning

The LSTD algorithm introduced by **[BradtkeBarto96]** eliminates the need of adapting a step size factor α. Furthermore, it improves the learning speed compared to TD by utilizing samples more efficiently. The function approximation introduced in Equation 4 limits the value function representation to linear functions, satisfying the limitation for the application of LSTD. In **[BradtkeBarto96]** the LSTD algorithm is introduced with the limitation of λ=0. An extended algorithm LSTD(λ) is presented in **[Boyan2002]**.

Applying the limitation to linear functions $\phi(s)$ the TD learning rule in Equation (11), for convergence analysis can be rewritten in the form $\theta := \theta + \alpha_n(\mathbf{d} + \mathbf{C}\theta + \omega)$, resulting in:

$$\mathbf{d} = E\left\{\sum_{i=0}^{T} \mathbf{e_i} r_i\right\}; \qquad \mathbf{C} = E\left\{\sum_{i=0}^{T} \mathbf{e_i}(\phi(s_{i+1}) - \phi(s_i))^T\right\}, \tag{12}$$

where $\mathbf{d} \in \mathbb{R}^K$ and $\dim(\mathbf{C}) = K \times K$. The vector ω denotes a zero-mean noise vector of dimension K. Proof for the convergence of θ to a fixed value θ_λ, satisfying $\mathbf{d} + \mathbf{C}\theta_\lambda = \mathbf{0}$ is given in **[BertsekasTsitsiklis96]**. In the course, it is shown that $\mathbf{C}$ is negative definite and that ω has only small variance from its zero-mean, which combined with the requirement for a decreasing step size α_n results in the above stated. The TD algorithm does not store sampled trajectories, and hence, wastes data which results in a low learning speed.

The LSTD(λ) algorithm does not perform gradient descent, but builds explicit estimates of $\mathbf{C}$ and $\mathbf{b}$ and stores them between trajectories. The algorithm directly solves the equation $\mathbf{d} + \mathbf{C}\theta_\lambda = \mathbf{0}$. The estimates calculated are denoted as:

$$\mathbf{b} = \sum_{i=0}^{t} \mathbf{e_i} r_i; \qquad \mathbf{A} = \sum_{i=0}^{t} \mathbf{e_i}(\phi(s_i) - \phi(s_{i+1}))^T. \tag{13}$$

The algorithm converges after n independent sample trajectories to unbiased explicit estimates

$$\mathbf{b} = n\mathbf{d}; \qquad \mathbf{A} = -n\mathbf{C}.$$

Using *Singular Value Decomposition* the inverse of $\mathbf{A}$ can be computed in order to solve the equation

$$\theta_\lambda = \mathbf{A}^{-1}\mathbf{b}.$$

In **[Boyan2002]**, it is stated that the LSTD(λ) algorithm for λ=1 produces the same results as the supervised linear regression method, trained on Monte Carlo results. For λ=0 the algorithm performs as the LSTD algorithm presented by **[BradtkeBarto96]**. The LSTD(λ) algorithm is shown in pseudocode in Algorithm 2, to illustrate the update steps.

Algorithm 2: LSTD(λ) for approximate policy approximation:

Data: a simulation model for a proper policy π in MDP $\mathcal{M}$;
a feature function $\boldsymbol{\phi} : \mathcal{S} \to \mathbb{R}^K$, mapping states to feature vectors, $\boldsymbol{\phi}(T) \stackrel{\text{def}}{=} 0$;
a parameter $\lambda \in [0, 1]$; and
a sequence of step sizes $\alpha_1, \alpha_2, \ldots$ for incremental coefficient updating.

Output: a coefficient vector $\boldsymbol{\theta}$ for which $V^\pi(s) \approx \boldsymbol{\theta}^T \cdot \boldsymbol{\phi}(s)$.

Set $\boldsymbol{A} := 0$, $\boldsymbol{b} := 0$, $t := 0$.
for *n := 1,2,...* **do**
  Choose a start state $s_t \in S$.
  Set $\boldsymbol{e}_t := \boldsymbol{\phi}(s_t)$.
  while $s_t \neq T$ **do**
    Simulate one step of the chain, producing a reward r_t and next state s_{t+1}.
    Set $\boldsymbol{A} := \boldsymbol{A} + \boldsymbol{e}_t(\boldsymbol{\phi}(s_t) - \boldsymbol{\phi}(s_{t+1}))^T$.
    Set $\boldsymbol{b} := \boldsymbol{b} + \boldsymbol{e}_t r_t$.
    Set $\boldsymbol{e}_{t+1} := \lambda \boldsymbol{e}_t + \boldsymbol{\phi}(s_{t+1})$.
    Set $t := t + 1$.
  end
  Whenever update coefficients are desired: Set $\boldsymbol{\theta} := \boldsymbol{A}^{-1}\boldsymbol{b}$.
end

The LSTD(λ) algorithm calculates a matrix inversion at the cost of $O(K^3)$ for every update of θ_λ. The update is most likely needed for every sampled trajectory. Furthermore the computation costs per time step are of $O(K^2)$. Compared to the TD(λ) algorithm the amount of computation is much higher, as the TD(λ) algorithm updates its coefficients at a linear cost of $O(K)$. For larger numbers of features K the LSTD(λ) algorithm causes significantly more computational load. Compensating this disadvantage, **[BradtkeBarto96]** lists the significant advantages. The TD(λ) algorithm can reuse trajectories for learning to compensate for the inefficient use of the samples. LSTD(λ) is data efficient, hence there is no need for reusing samples. Using less samples the LSTD(λ) algorithm converges faster than TD(λ). Another advantage is the omission of the step size parameter. On the one hand an advantage in usability is achieved by eliminating the need for tuning the parameter. On the other hand the LSTD(λ) algorithm can not be slowed down by a bad choice of parameter.

5 Application to control problems

LSTD makes efficient use of sample data and converges faster than other temporal difference methods. However it is limited to prediction learning problems, as stated in **[LagoudakisParr2003]**. LSTD can be used for approximating the state-value function, however with out a model this knowledge can not be utilized for policy search. The method of policy iteration is an approach to the control problem in reinforcement learning. It is based on a loop of policy evaluation and policy improvement. To apply the idea of TD methods in policy evaluation, the LSTDQ algorithm is introduced. The algorithm learns an approximation for the action-value function, where as the TD(λ) algorithm approximates the state-value function. For the computation of V^π sample trajectories of a policy where given. The LSTDQ algorithm is given a source of sample sets of the form (s, a, r, s') from which it picks training samples arbitrarily. By this method the action-value function of different policies can be approximated using the same source of samples.

The same way the representation of state-value functions was realized by a linear function approximation it is possible to represent action-value functions as

$$Q^\pi(s, a) \approx \widehat{Q}^\pi(s, a; w).$$

where w denotes the adjustable parameter vector of the approximation function. The LSTDQ algorithm used in the policy evaluation step is based on the LSTD method presented in **[LagoudakisParr2003]**, requiring a linear representation of the value function

$$\widehat{Q}^\pi(s, a; w) = \sum_{j=1}^{k} \phi_j(s, a) w_j. \tag{14}$$

Policy improvement is not at the focus of this paper, for completeness a basic approach is presented. The improvement is achieved by computing a greedy policy π' based on Q^π for all states s:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q^\pi(s, a). \tag{15}$$

In the step of policy evaluation the LSTDQ algorithm solves the Bellman Equation for w. The parameter vector is only updated after all samples have been learned. Learning from the samples, the matrix A and the vector b are refined by the update rules

$$\mathbf{b} = \sum_{i=0}^{T} \phi(s_i, a_i) r_i; \qquad \mathbf{A} = \sum_{i=0}^{T} \phi(s_i, a_i) \Big(\phi(s_i, a_i) - \phi(s_{i+1}, \gamma\pi(s_{i+1}) \Big)^T. \tag{16}$$

After D is depleted the parameter of the linear function approximation is updated. This step requires *single value decomposition*, which comes at a high computational cost, as mentioned for the LSTD(λ) algorithm.

$$w = A^{-1} b \tag{17}$$

The updated action-value function is used in the policy improvement step, as depicted in Equation (15), for computing a new policy π', which is inserted in Equation (16), closing the loop of policy iteration.

With the LSTDQ algorithm displayed as pseudocode this paper leaves the reader to hit the ground running and get a good start at understanding reinforcement learning roughly and temporal difference methods in particular.

Algorithm 3: LSTDQ for approximate policy evaluation:

Data: a feature function $\boldsymbol{\phi} : \mathcal{S} \to \mathbb{R}^K$, mapping states to feature vectors, $\boldsymbol{\phi}(T) \stackrel{\text{def}}{=} 0$;
a discount factor $\gamma \in]0, 1]$;
a source of samples D; and
the policy π, whose value function is sought

Output: a weight vector $\boldsymbol{w}$ for which $Q^\pi \approx \boldsymbol{\phi}\boldsymbol{w}^\pi$.

Set $\boldsymbol{A} := 0$, $\boldsymbol{b} := 0$.
for $(s, a, r, s') \in D$ **do**
  Simulate one step of the policy, producing a reward r and next state s'.
  Set $\widetilde{\boldsymbol{A}} := \widetilde{\boldsymbol{A}} + \boldsymbol{\phi}(s, a) \left[\boldsymbol{\phi}(s, a) - \gamma\boldsymbol{\phi}(s', \pi(s'))\right]^T$.
  Set $\widetilde{\boldsymbol{b}} := \widetilde{\boldsymbol{b}} + \boldsymbol{\phi}(s, a) r$.
end
Set $\boldsymbol{w}^\pi := \widetilde{\boldsymbol{A}}^{-1} \widetilde{\boldsymbol{b}}$.

References

[Sutton88] Sutton, Richard S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning* **3**: 9-44. Boston: Kluwer Academic Publishers.

[LagoudakisParr2003] Lagoudakis, Michail G. & Parr, Ronald (2003). Least-Squares Policy Iteration. *Journal of Machine Learning Research* **4**: 1107-1149.

[SuttonBarto98] Sutton, Richard S. & Barto, Andrew G. (1998). *Reinforcement Learning: An Introduction*. Cambridge: MIT Press.

[BradtkeBarto96] Bradtke, Steven J. & Barto, Andrew G. (1996). Linear Least-Squares Algorithms for Temporal Difference Learning. *Machine Learning* **22**: 33-57. Boston: Kluwer Academic Publishers.

[Boyan2002] Boyan, Justin A. (2002). Technical Update: Least-Squares Temporal Difference Learning. *Machine Learning* **49**: 233-246. Boston: Kluwer Academic Publishers.

[Dann2012] Dann, Christoph (2012). *Algorithms for Fast Gradient Temporal Difference Learning*. Autonomous Learning Systems Seminar, TU Darmstadt.

[BertsekasTsitsiklis96] Bertsekas, D. & Tsitsiklis, J. (1996). *Neuro-dynamic programming*. Belmont: Athena Scientific.