Department of
Computer Science

UNIVERSITY OF
BATH

Technical Report

A Formal Framework for Reinforcement Learning with Function Approximation in Learning Classifier Systems

Jan Drugowitsch and Alwyn Barry

Technical Report 2006-02 January 2006

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

A Formal Framework for Reinforcement Learning with Function Approximation in Learning Classifier Systems

Jan Drugowitsch
Department of Computer Science
University of Bath, UK
J.Drugowitsch@bath.ac.uk

Alwyn M Barry
Department of Computer Science
University of Bath, UK
A.M.Barry@bath.ac.uk

January 2006

Abstract

To fully understand the properties of Accuracy-based Learning Classifier Systems, we need a formal framework that captures all components of classifier systems, that is, function approximation, reinforcement learning, and classifier replacement, and permits the modelling of them separately and in their interaction. In this paper we extend our previous work on function approximation [22] to reinforcement learning and its interaction between reinforcement learning and function approximation. After giving an overview and derivations for common reinforcement learning methods from first principles, we show how they apply to Learning Classifier Systems. At the same time, we present a new algorithm that is expected to outperform all current methods, discuss the use of XCS with gradient descent and TD(λ), and given an in-depth discussion on how to study the convergence of Learning Classifier Systems with a time-invariant population.

1 Introduction

Accuracy-based Learning Classifier Systems (LCS), a Machine Learning method that combines function approximation, reinforcement learning and evolutionary computation, are capable of evolving human-readable production rules that describe the most general but still accurate representation of a solution. While featuring competitive performance in single-step tasks, such as data mining [40, 24, 4, 19, 2], they still only show limited success in other than relatively trivial delayed-reward tasks [3, 1, 21]. These limitations have stimulated research to formulate partial models of LCS [14, 16, 52]. However, even the latest theoretical developments have only produced piecemeal models that do not adequately capture the interaction between the different components of LCS.

As we have already argued in [22], to make adequate progress in the understanding of LCS we need a formal framework and model that is able to capture all components and their interactions. The framework should bridge the gap between LCS and its related Machine Learning techniques to reveal similarities and differences, and ease the translation of new developments from one field to the other. Additionally, it needs to be flexible enough to allow for the incorporation of eventual extensions to LCS.

In this paper we concentrate on investigating the reinforcement learning component of LCS, and how it interacts with its function approximation. Our study does not yet consider the replacement of classifiers and will therefore assume a time-invariant classifier population. We will build on and extend the framework that we have previously introduced to study the function approximation in LCS [22]. It is known that certain methods of reinforcement learning are not stable when used in combination with particular function approximation architectures. Q-Learning, for example, is known to diverge in some cases when used in combination with linear function approximation [12]. Hence, to guarantee stability of the application of LCS to multi-step problems, we need to study the compatibility between reinforcement learning and LCS function approximation. We will not consider the modified LCS function approximation architecture introduced in [52] for the reasons given in [22].

The first comparison between reinforcement learning and LCS was done in [20], where Dorigo and Bersini show that a Very Simple CS without generalisation and slightly modified implicit bucket brigade is equivalent to tabular Q-learning. A more general study showed how Evolutionary Computation can be used for reinforcement learning [35]. The latter investigates reinforcement learning on both the policy level and the value function level, but ignores the development of XCS [57] which moves LCS even closer to reinforcement learning, in particular Q-learning. Wilson was possibly the first to use XCS for function approximation [58]. Since then, it has been explicitly linked to reinforcement learning with function approximation in an attempt to add gradient descent to the Q-Learning update of XCS [17, 18], which was criticised by Wada et al. [51], and is commented on in Section 4.2.4. Recent developments that improved the performance of LCS in multi-step problems were the extension of the function approximation architecture for single classifiers [59, 33], the introduction of the Recursive Least Squares algorithm to improve approximation speed and accuracy [32], and our use of the Kalman filter to provide more accurate error estimation for classifiers [22]. Simultaneously, Booker has developed a hyper-plane coding scheme for classifiers [8], related to CMAC's of reinforcement learning. Similarly to [52] it forms its approximation by aggregating the approximation of classifier, which is why we will not consider it in our framework.

Due to LCS's reliance on reinforcement learning methods to solve multi-step problems, we will use studies of the latter to guide our investigations. They originate in Dynamic Programming (DP) and Temporal-Difference Learning [53], where the theoretical properties of DP are usually at the heart of answering questions of the stability of various reinforcement learning methods. Therefore we have chosen to first introduce common methods in DP and then to show how reinforcement learning builds on them.

Firstly, in section 2 we introduce how problems can be formulated in the reinforcement learning framework, and the approach that is taken by DP to solve such problems. Furthermore, we describe the function approximation architecture that we will discuss in combination with reinforcement learning, and how to express everything in the more lucid matrix notation.

Based on that framework, in Section 3 we will introduce common methods in reinforcement learning by firstly describing how the problems are approached by DP. Furthermore, we will discuss how to reduce the spatial an computational requirements of the different DP approaches by the use of function approximation, and how that influences their stability. By introducing and discussing Temporal Difference Learning, we show how DP methods can be efficiently approximated while lowering the computational costs. We conclude this section by showing how to combine them with function approximation and how to use them without a model of the problem.

In Section 4, we will firstly introduce the structure of the LCS function approximation based on our work in [22]. Applying our previous description of reinforcement learning, we will derive from first principles how to combine the LCS approximation architecture with reinforcement learning to provide several model-based and model-free methods. For Q-Learning with LCS we will, in addition, give details about two possible implementations, one based on the Least Mean Square (LMS) algorithm, and the other based on the Kalman filter. As a final step, we will give an overview in Section 4.4 of how to study the convergence of reinforcement learning with LCS function approximation by looking at the properties of a DP iteration. Note that the convergence of the LCS reinforcement learning is still an open question, which our framework might help to answer.

2 The Reinforcement Learning Framework

This section gives an overview to the type of problems that we deal with, and how a method called *Dynamic Programming (DP)* can be used to approach such problems. Most of that section can be found in more detail in [6]. The notation that is used is a blend of [6] and [47], and allows integration into the LCS function approximation framework introduced in [22].

2.1 Problem Formulation

We will concentrate on problems that are solvable by reinforcement learning and are therefore expressible as *Markov Decision Processes (MDPs)*: Let S be the set of states of the problem domain, which we will assume to be of finite[1] size N, and will hence map to the set of natural numbers $\mathbb{N}$. In every state $i \in S$ we can perform an action a out of a finite set A that leads us to the next state j. The probability of transition $p_{ij}(a)$ from state i to state j upon performing action a is given by the transition function $p : S \times S \times A \to \mathbb{R}$. Every such transition is mediated by a scalar reward $r_{ij}(a)$, defined through the reward function $r : S \times S \times A \to \mathbb{R}$. The positive *discount factor* $\gamma \in \mathbb{R}$ with $0 < \gamma \leq 1$ determines the preference of immediate reward over future reward. Therefore, the MDP that describes the problem is defined by the quintuple $\{S, A, p, r, \gamma\}$.

[1] A finite state space is assumed to simplify analysis. It might be possible to extend our analysis to continuous state spaces, but that might require significantly more technical work. For examples of an analysis of reinforcement learning in continuous state spaces see [29, 38].

The aim is for every state to pick the action that maximises reward in the long run, where future rewards are possibly valued less that immediate rewards. A possible solution is represented by a *policy* $\mu : S \to A$, which returns the chosen action $\mu(i)$ for any state $i \in S$. Thus, when fixing a policy μ, the MDP is reduced to a Markov Chain with transition probabilities $p^\mu : S \times S \to R$, where the transition probability from state i to state j is given by $p^\mu_{ij} = p_{ij}(\mu(i))$, with a reward $r^\mu : S \times S \to \mathbb{R}$ of $r^\mu_{ij} = r_{ij}(\mu(i))$. In such cases we will usually operate with the expected reward $r^\mu_i : S \to \mathbb{R}$ given some state i, which is

$$r^\mu_i = \sum_{j \in S} p^\mu_{ij} r^\mu_{ij} = \sum_{j \in S} p_{ij}(\mu(i)) r_{ij}(\mu(i)). \tag{1}$$

This reward expresses what we would expect to receive when choosing an action according to policy μ in state i.

2.2 Dynamic Programming Approach

An approach that is taken by DP is to define a value function $V : S \to \mathbb{R}$ that expresses for each state in the state space how much reward we can expect to receive in the long run. Let $\mu = \{\mu_0, \mu_1, \dots\}$ be a sequence of policies where we are operating according to policy μ_t at time t, starting at time $t = 0$. Then the reward that is accumulated after n steps starting at state i, called the *n-step return* V^μ_n for state i, can be given by

$$V^\mu_n(i) = \mathbb{E}\left(\gamma^n R(i_n) + \sum_{t=0}^{n-1} \gamma^t r^{\mu_t}_{i_t i_{t+1}} | i_0 = i\right),$$

where $\{i_0, i_1, \dots\}$ is the sequence of states, and $R(i_n)$ is the expected return that we will receive when starting from state i_n. The discount factor γ is part of the problem formulation and determines how much we value future reward when compared to immediate reward[2]. The optimal expected n-step return starting from state i, denoted by $V^*_n(i)$, is the one that chooses a policy that maximises that return,

$$V^*_n(i) = \max_\mu V^\mu_n(i).$$

Finite-step cases can be seen as a special case of *infinite-horizon problems* that are guaranteed to end in a reward-free terminal state at latest after n actions. Hence, we can concentrate on infinite-horizon problems, for which the expected return when starting at state i is given by

$$V^\mu(i) = \lim_{n \to \infty} \mathbb{E}\left(\sum_{t=0}^{n-1} \gamma^t r^{\mu_k}_{i_t i_{t+1}} | i_0 = i\right). \tag{2}$$

The optimum V^* is again given by following the policy that maximises the expected return, that is

$$V^*(i) = \max_\mu V^\mu(i).$$

The policies associated with the optimal values form the solution to our problem. Fortunately, those policies are typically stationary, that is $\mu_t = \mu_0$ for all $t = 0, 1, \dots$. We will denote a stationary policy by μ.

Given that we know the optimal value function V^*, the optimal policy μ^* is one that performs the action that leads us to the highest-valued states out of all states that we can reach for the current state, that is

$$\mu^*(i) = \operatorname*{argmax}_{a \in A} \mathbb{E}\left(r_{ij}(a) + \gamma V^*(j) | i, a\right).$$

Hence, once we know the optimal value function V^*, we also know an optimal policy $\mu*$ and have solved the problem.

2.3 Optimal Control and Belman's Equation

In some cases we do not have a model of the problem but can only explore it by trial-and-error or simulation. That might, for example, be the case when $\mathbb{E}(r_{i,j}(a) + \gamma V^*(j)|i, a)$ cannot be evaluated. In such cases we can resort to storing values for state-action pairs rather than only for states. Let

[2] Note that the difference between reward and return is that return implicitly considers future reward, whereas reward does not.

$Q : S \times A \rightarrow \mathbb{R}$ be the function that gives the expected return $Q(i, a)$ when taking action a in state i, that is, for some policy μ,

$$Q^\mu(i,a) = \lim_{n \to \infty} \mathbb{E}\left(r_{i_0 i_1}(a) + \gamma \sum_{t=1}^{n-1} \gamma^t r^\mu_{i_t i_t+1} | i_0 = i, a \right) = \mathbb{E}\left(r_{ij}(a) + \gamma V^\mu(j) | i, a \right),$$

which is the expected return when taking a in state i and then following policy μ. Equally, the value function can be expressed as the Q-value of that state when following the current policy μ, that is

$$V^\mu(i) = Q^\mu(i, \mu(i)).$$

Given that the policy μ is optimal, the optimal action in state i is the one with the highest Q-value. Hence, knowing the optimal Q^*-values, we can derive the optimal policy be evaluating

$$\mu^*(i) = \operatorname*{argmax}_{a \in A} Q^*(i, a).$$

This allows us to express the optimal value function using that policy by

$$V^*(i) = Q^*(i, \operatorname*{argmax}_{a \in A} Q^*(i, a)) = \max_{a \in A} Q^*(i, a).$$

Combining that with the definition of the Q-values gives us some form of *Bellman's Equation*

$$V^*(i) = \max_{a \in A} \mathbb{E}\left(r_{ij}(a) + \gamma V^*(j) | i, a \right), \tag{3}$$

which relates the optimal values of different states by defining them as the maximum sum of expected reward and value of the next state. Finding a solution to that equation forms the core of most DP methods.

We can derive a similar form of equation for a stationary policy μ. Then, a value of state i is defined according to Eq. (2), which can be rewritten as

$$V^\mu(i) = \lim_{n \to \infty} \mathbb{E}\left(r^\mu_{i_0 i_1} + \gamma \sum_{t=1}^{n-1} \gamma^t r^\mu_{i_t i_t+1} | i_0 = i \right).$$

The sum in the expectation is by definition the value of state i_1. Hence, above is equal to

$$V^\mu(i) = \mathbb{E}\left(r^\mu_{ij} + \gamma V^\mu(j) | i \right), \tag{4}$$

which is *Bellman's Equation for a fixed policy μ.*

2.4 Problem Types

The three basic classes of infinite-horizon problems are:

Stochastic shortest path problems These problems are undiscounted, i.e. $\gamma = 1$, with a reward-free terminal state 0, and require finding the sequence of actions that maximise the overall reward and lead to that terminal state. With the assumption that the terminal state is always reachable, these problems are in effect finite-horizon problems, but the distance to the horizon may be random.

Discounted problems This set of problems have $\gamma < 1$ and a bounded reward function to make the value $V^\mu(i)$ well defined. Discounted problems are similar to stochastic shortest path problems as for every discounted problem we can generate an equivalent stochastic shortest path problem that leads to the same optimal value function [6, Ch. 2.3].

Average reward per step problems In some cases, the total return is $V^\mu(i) = -\infty$ for every policy μ and initial state i. In many such problems, however, the average reward per step is well defined in its limit, and finite. We will not consider this set of problems any further.

Note that not all policies in the stochastic shortest path problem will lead to the terminal state. Hence, in analysis we would have to restrict ourself to so-called *proper* policies that are guaranteed to reach the terminal state. Besides that, its analysis is very similar to the one of discounted problems, which is why we will only consider the case of the latter.

2.5 Linear Approximation Architecture

Even though the set of states S is finite, it can be very large. Therefore, operating on the value function V would be spatially prohibitive. A common approach is to not store the function V itself, but only an approximation $\tilde{V}$ of it. The function approximation architecture that is currently known to work best in combination with reinforcement learning is a linear architecture, including "[...] state aggregation methods, CMACs, polynomial or wavelet regression techniques, radial basis function networks with fixed bases, and finite-element methods" [36]. In [22] we describe how LCS deviate from that linear architecture, but let us for now ignore that deviation and assume a simple linear architecture. We will analyse how the LCS architecture operates within different reinforcement learning methods in Section 4.

Let $\{\phi_1, \dots \phi_L\}$ be a set of L basis functions $\phi_l : S \rightarrow \mathbb{R}$ that return different features of a state. The collection of all features for some state i form its feature vector $\phi : S \rightarrow \mathbb{R}^L$, given by $\phi(i) = (\phi_1(i), \dots, \phi_L(i))'$, where $\cdot'$ denotes the transpose and indicates that the vector is a column vector. Additionally, let $w \in \mathbb{R}^L$ denote the adjustable parameter vector of our approximation, called the *weight vector*. Then, the approximation $\tilde{V}$ of V for some state i is given by the dot product of the feature vector of that state and the weight vector, that is

$$\tilde{V}(i) = w'\phi(i).$$

The independence between the weight vector and the current state is the defining characteristic of a linear approximation architecture.

For control problems, rather than using the value function V we operate on the Q-value function. That function can be approximated by a linear architecture in the same way. Let $w \in \mathbb{R}^L$ again be the weight vector. Then the approximation $\tilde{Q}$ of Q for some state i is given by

$$\tilde{Q}(i) = w'\phi(i).$$

The aim of the approximation is to minimise the weighted mean-squared error between the value function V and its approximation $\tilde{V}$, that is to find the weight vector w for which

$$\min_w \sum_{i \in S} \pi(i)(V(i) - w'\phi(i))^2,$$

where $\pi(i) \in \mathbb{R}$ is the weight assigned to state $i \in S$, with $\pi(i) > 0$ for any $i \in S$, and $\sum_{i \in S} \pi(i) = 1$. As that function is convex, we can find its unique minimum by setting its first derivative w.r.t. w to zero. The same applies to approximating the Q-value function. For more details on linear function approximation in general and w.r.t. LCS see [22].

As by the definition of the mean-squared error, the error weights $\pi(i)$ play a significant role in the approximation process, and are determined by the state sampling distribution. If there is a generating process that allows creating arbitrary state transitions, then those weights can be chosen freely. On the other hand, if we only have a set of sample transitions, or only can perform transitions according to the underlying Markov Process, then those error weights are determined by the sampling frequencies or steady-state distribution of the Markov Chain respectively. As we will see later, having a good set of transition samples available is important when approximating the value function.

2.6 Matrix Notation

As our state and action space are finite, it is convenient to apply matrix notation to ease readability. For policy μ, let $P^\mu = (p^\mu_{ij})$ be the $N \times N$ transition matrix of the Markov Chain for that policy. For that same policy, let r^μ be the N-sized vector that holds as its ith element the expected reward when following that policy from state i, that is $r^\mu = (r^\mu_1, \dots, r^\mu_N)'$, where r^μ_i is the expected reward for following policy μ in state i according to Eq. (1).

Let V be the N-sized value vector $V = (V(1), \dots, V(N))'$, where $V(i)$ gives the value of state i. Then, Bellman's Equation for a fixed policy μ (Eq. (4)) becomes

$$V^\mu = r^\mu + \gamma P^\mu V^\mu,$$

where V^μ is the value vector for policy μ. This form shows clearly that the value of a state is the sum of the expected reward from that state and the expected discounted value of the state after one transition. In future discussions we will use both *value function* and *value vector* to refer to the same concept.

To discuss linear function approximation, let Φ be the $N \times L$ matrix that combines the features of all states, that is

$$\Phi = \begin{pmatrix} - & \phi(1)' & - \\ & \dots & \\ - & \phi(N)' & - \end{pmatrix}.$$

That allows us to define the approximation parameterised by the weight vector w as $\tilde{V} = \Phi w$. Let D be the $N \times N$ diagonal matrix with the sampling distribution $\pi(1), \dots \pi(N)$ along its diagonal. The approximation aims at minimising the weighted distance between the value vector V and its approximation $\tilde{V}$, given by $\|V - \tilde{V}\|_D$, where $\| \cdot \|_D$ denotes the weighted norm, given for any $V \in \mathbb{R}^N$ by $\|V\|_D^2 = \sum_{i \in S} \pi(i) V(i)^2$. We can find this approximation by orthogonally projecting the value vector into the approximation subspace $\{\sqrt{D}\Phi w : w \in \mathbb{R}^L\}$, spanned by the column vectors of $\sqrt{D}\Phi$, and given by

$$\tilde{V} = \Pi_D V,$$

where Π_D is the projection matrix

$$\Pi_D = \Phi(\Phi' D \Phi)^{-1} \Phi' D. \tag{5}$$

The $L \times L$ matrix $\Phi' D \Phi$ is invertible if the basis functions $\phi_1, \dots, \phi_L$ are linearly independent and if there are at least as many states as there are features, that is $N \leq L$.

3 Common Methods in Reinforcement Learning

Using the described framework, we will discuss some methods that can be used to solve Bellman's Equation or an approximation of it. Whereas DP requires a complete model of the problem, Temporal-Difference learning approximates its solution by iterative updates based on simulated state trajectories and is therefore the more adequate method for the simulation-based approach and adaptive control.

3.1 Dynamic Programming Methods

Bellman's Equation is a set of linear equations that can in theory be evaluated directly, given that all problem parameters are known. However, even then the evaluation might be tedious and not very efficient. Fortunately, several methods have been developed that make solving that equation easier. In this section we will introduce some of those methods, about which more information can be found in [6].

3.1.1 The Dynamic Programming Operators T and T_μ

The core of the DP methods is formed by the two DP updates, given by the mapping operators T and T_μ. In this section we will define those operators and give a short description of their properties.

For any value vector V, we define the vector TV as the result of applying an update related to Bellman's Equation to it once, giving its components

$$(TV)(i) = \max_{a \in A} \sum_{j \in S} p_{ij}(a)(r_{ij}(a) + \gamma V(j)). \tag{6}$$

Similarly, for any stationary policy μ, we define the vector $T_\mu V$ as a result of applying an update related to Bellman's Equation for a fixed policy, giving its components

$$(T_\mu V)(i) = \sum_{j \in S} p_{ij}^\mu (r_{ij}^\mu + \gamma V(j)),$$

which in matrix notation can be written as

$$T_\mu V = r^\mu + \gamma P^\mu V.$$

We will write $T^n V$ for applying T to V, n times. Similarly, $T_\mu^n V$ means the application of T_μ to V, n times.

One elementary property of the mapping operators T and T_μ is that they both define a contraction mapping. That is, given any value vectors V and $\bar{V}$ and any policy μ,

$$\|TV - T\bar{V}\|_\infty \leq \gamma \|V - \bar{V}\|_\infty,$$
$$\|T_\mu V - T_\mu \bar{V}\|_\infty \leq \gamma \|V - \bar{V}\|_\infty,$$

where $\| \cdot \|_\infty$ is the maximum norm, defined by $\|V\|_\infty = \max_i |V(i)|$. That means that when applying the same operator to two different value vectors, they will move closer together (as $\gamma < 1$). Applying them repeatedly will therefore lead us to some fixed point of that update. This property of the DP operators is at the core of all of the methods.

Using those operators, we can state the main results of their analysis, as listed in [6]. Due to the contraction property of T, the optimal value vector V^* is the unique vector that satisfies $TV^* = V^*$, which is Bellman's Equation (Eq. (3)) in operator notation. Furthermore, repeatedly applying T to any initial value vector V will result in the optimal value vector V^*, that is $\lim_{n\to\infty} T^n V = V^*$. Similarly, repeatedly applying T_μ to any initial value vector V with any fixed policy μ will give us the unique solution to the Bellman Equation for fixed policy μ (Eq. (4)), that is $\lim_{n\to\infty} T_\mu^n V = V^\mu$. However, this policy μ is only optimal if and only if $T_\mu V^* = TV^*$. Note that it is possible to have several optimal policies.

3.1.2 Standard and Asynchronous Value Iteration

Value iteration is a method that follows directly from the results of the last section. It is defined by repeatedly applying T to the current value vector V. According to [6, Prop. 2.3], this method is guaranteed to converge to the optimal value vector V^* for any initial vector V. However, we cannot guarantee convergence before an infinite number of iterations.

Asynchronous Value Iteration is a variant to Value Iteration that does not update the values of all states synchronously, but only updates one state per update. We will not give any formal definition of the method here but will only state that, as long as every state is updated infinitely often, the method converges to the optimal value vector V^* for any initial vector V [6, Prop. 2.5].

3.1.3 Standard and Modified Policy Iteration

As an alternative to Value Iteration, Policy Iteration will always terminate after a finite number of iterations, and is based on alternating *policy evaluation* and *policy improvement.* In the policy evaluation step at time t, we compute the values V^{μ_t} for the policy μ_t as the solution to the system of equations given by Eq. (4). Subsequently, we improve the current policy by

$$\mu_{t+1}(i) = \underset{a \in A}{\operatorname{argmax}} \sum_{j \in S} p_{ij}(a) \left(r_{ij}(a) + \gamma V^{\mu_t}(j) \right),$$

which in operator notation is

$$T_{\mu_{t+1}} V^{\mu_t} = TV^{\mu_t}.$$

The sequence of policies $\{\mu_0, \mu_1, \dots\}$ generated by that procedure is monotonically improving and is guaranteed to terminate with an optimal policy [6, Prop. 2.4].

If the number of states is large, the policy evaluation step of Policy Iteration might be computationally prohibitive. One way to get around this is to approximate the value function V^{μ_t} by using a limited number of Value Iteration updates. The idea behind this method, called Modified Policy Iteration, is that value iteration involving a single policy (evaluating $T_\mu V$) is much less expensive than an iteration involving all policies (evaluating TV).

3.1.4 Asynchronous Policy Iteration

Asynchronous Policy Iteration allows for even more freedom than Modified Policy Iteration by mixing Asynchronous Value Iteration with Policy Iteration. At each step, we can either i) update some states of the value vector by Asynchronous Value Iteration, or ii) improve the policy of some set of states by policy improvement. Hence, Asynchronous Policy Iteration is a generalisation over all previously discussed methods. However, convergence can only be guaranteed if all states are updated infinitely often, and for the initial policy μ_0 and initial value vector V_0 we have $T_{\mu_0} V_0 \leq V_0$ [6, Prop. 2.5]. This initial condition can be satisfied by selecting a proper initial policy μ_0 and setting the initial value vector such that $V_0 = V^{\mu_0}$.

3.2 Approximate Dynamic Programming

Approximate DP applies the DP methods to an approximation $\tilde{V}$ of the value function rather than on the value function V itself. That this change also modifies the convergence behaviour will be discussed in the next two sections.

3.2.1 Approximate Value Iteration

Approximate Value Iteration is based on the Value Iteration update $V_{t+1} = TV_t$ performed on the approximation $\tilde{V}$. Hence, the update can be defined as

$$\tilde{V}_{t+1} = \operatorname*{argmin}_{\tilde{V}} \|T\tilde{V}_t - \tilde{V}\|,$$

which minimises the squared error of approximating the Value Iteration update $T\tilde{V}_t$. As demonstrated by Boyan and Moore [11], that method might diverge when used with even the most common function approximation architectures, like linear or quadratic regression, local weighted regression, or neural networks. The identified problem was that even though the approximation is able to adequately represent the optimal value function, it fails to approximate the immediate steps of the Value Iteration.

An option to avoid divergent behaviour of the method is to only use approximation architectures that by themselves feature non-expansion to the maximum norm, as discussed by Gordon in [23]. A non-expansion is similar to a contraction (see Section 3.1.1), but it does not necessarily have to reduce the norm, as long as it does not expand it. As the DP operator T causes a contraction to the maximum norm, applying a non-expansion to the same norm results in an overall contraction. This is sufficient to state that, by the Contraction Mapping Theorem, the update converges to the unique fixed point of the update procedure, given by the solution to

$$\tilde{V}^* = \operatorname*{argmin}_{\tilde{V}} \|T\tilde{V}^* - \tilde{V}\|.$$

As for any approximation, the values that $\tilde{V}$ can take are restricted to the approximation space defined by the approximation architecture.

A class of approximation architectures that fulfils the above requirement is the class of *averagers* [23]. This class is characterised by having the approximation of a set of observations bounded from below and above by the range of those observations, that is, the approximation can never exceed the highest observed value. That class, for example, contains the methods of "[...] local weighted averaging, k-nearest neighbour, Bézier patches, linear interpolation, bilinear interpolation on a square (or cubical, etc.) mesh, as well as simpler methods like grids and other state aggregation." [23]. The linear architecture, as described before, is not necessarily an averager and thus might diverge when used for Approximate Value Iteration[3].

3.2.2 Approximate Policy Iteration

Approximate Policy Iteration performs the policy evaluation step of Policy Iteration by generating an approximation $\tilde{V}^{\mu_t}$ of the value function V^{μ_t} [6]. The policy improvement step generates a new policy based on the approximated value function. This method is proven to be significantly more stable (in the sense that it cooperates with a higher variety of function approximation architectures) than Approximate Value Iteration, but has the disadvantage of having to store the policy while evaluating it. An alternative is to base the policy on the approximated value function of a partial evaluation of the previous policy, which at worst means to directly derive the policy from the current value function approximation at every step. We will discuss the impact of such a change in the next section, and will for now assume that the policy is fully evaluated before it is improved.

As for the function approximation, we again assume a linear architecture and want to minimise the mean-squared error $\|V^\mu - \tilde{V}\|_D$ for a policy μ. There are several approaches to that [36], of which the optimal solutions are different [41]:

Optimal approximate solution, which is to find the minimum of $\|V^\mu - \tilde{V}\|_D$, i.e. the orthogonal projection $\tilde{V}^\mu = \Pi_D V^\mu$ onto the approximation subspace w.r.t. $\|\cdot\|_D$. As we do not know V^μ, we can estimate its value by Monte-Carlo simulations, which makes the method computationally expensive.

Minimal Quadratic Residual (QR) solution, which is to find the function $\tilde{V}^\mu$ that minimises the Bellman residual $\|T_\mu \tilde{V} - \tilde{V}\|_D$. As this Bellman residual is related to the change caused by the DP update for a constant policy, minimising this residual is equivalent to finding the solution for which its change is minimised. Fortunately, for linear approximation architectures, finding the QR solution is reduced to resolving a linear system of size K that can always be solved. Another

[3] To be more specific, the linear function approximation architecture is an averager as long as the features are state-independent, e.g. if $\phi(i) = (1)$ for all $i \in S$.

advantage of this method is that its stability is relatively insensitive to the sampling distribution given by D, particularly when compared to the method that will be presented next [36]. A major disadvantage is that finding the QR solution either requires a full model of the system, or at least a generative model that allows as to produce sample trajectories. It cannot be applied to problems where we only have a fixed set of trajectories [31].

Temporal-Difference (TD) solution, which aims at finding the fixed point $\tilde{V}^\mu = \Pi_D T_\mu \tilde{V}^\mu$ of the update $\Pi_D T_\mu$, giving a projection of the DP update for a fixed policy into the approximation subspace. Due to its use in LCS, we will discuss this method at length in a later section. For the sake of comparison, let us only mention that this method is significantly more sensitive to the sampling distribution given by D, but can be applied to problems where no model exists.

Probably the most general approach to the analysis of policy evaluation with linear function approximation, as introduced in [42], is to reduce the algorithms to matrix iteration of the form

$$w_{t+1} = Aw_t + b,$$

where w_t is the weight vector after iteration t, A is an $L \times L$ matrix, and b is a vector of size L. To study convergence of such an iteration, we need to know the spectral radius $\rho(A)$, i.e. the eigenvalue with the maximal absolute value $\rho(A) = \max\{|\lambda| : \lambda \in \sigma(A)\}$, where $\sigma(A)$ is the spectrum of A, that is the set of its eigenvalues. The above iteration converges to its fixed point $w = (I - A)^{-1}b$ if and only if matrix A has a spectral radius $\rho(A) < 1$. This investigation is expanded on in [34], where Merke and Schoknecht show that for the case $\rho(A) = 1$ the iteration still converges under certain conditions, but the limit depends on the initial weight vector w_{-1}.

Both the QR and the TD-method can be reduced to such matrix iteration, as shown in [42]. In [34], this matrix iteration was used to demonstrate that for the QR method there exists a range of positive step-sizes α such that the method converges for every initial value w_0. The method of TD is more sensitive and might diverge if trajectory sampling does not follow the steady-state distribution of the Markov Chain, as demonstrated in [25] and analysed in [49]. Even if we sample according to the steady-state distribution, that distribution changes at the next policy improvement step, which might mislead the Policy Iteration process [27]. More positively, TD was proven to converge faster than QR under certain conditions, even in its weakest form, TD(0) [43].

The approximated value function $\tilde{V}^\mu$ will most likely never exactly represent V^μ. Therefore, when alternating approximate policy evaluation and greedy policy improvement we might improve the policy rapidly in the first few iterations and then oscillate around the optimal policy. This behaviour is due to the approximation error in comparison to the set of value functions that produce optimal policies. At some point in the iteration we will not be able to get any closer to the optimal value function V^* and the policy improvement step will therefore fail to be efficient. Hence, the algorithm does not converge, but due to the closeness of the approximate value function to the optimal value function, we can expect to reach good final policies [36]. Error bounds for sub-optimal policies can be found as functions of the maximum norm in [6, Ch. 6.2], and as functions of the quadratic norm in [36].

3.2.3 Optimistic Policy Iteration

Optimistic Policy Iteration, like Policy Iteration, consists of a policy evaluation and a policy improvement step. However, in contrast to Policy Iteration, the policy improvement step is based on an incomplete evaluation of the policy. The method is in many respects similar to Asynchronous Policy Iteration introduced in Section 3.1.4 [6, Ch. 5.4].

By the use of a Value Iteration-like iterative update for state transition i_t, i_{t+1} at time $t + 1$, given by a variant of $V_{t+1}(i_t) = (T_\mu V_t)(i_t)$, we get a monotonically improving sequence of value functions with the value function V^μ for policy μ as its limit, given that each state is visited an infinite number of times. Hence, we can perform policy improvement based on an intermediate step rather than the limit. Optimistic Policy Iteration improves the policy after each partial policy evaluation step. As such, it does not need to store the policy separately but can it derive at each step from the current value function. Partially Optimistic Policy Iteration is a variant that performs several policy evaluation steps before improving the policy and therefore needs to store the policy separately.

For the case without value function approximation, Tsitsiklis has shown that Optimistic Policy Iteration with a synchronous value function update of

$$V_{t+1} = (1 - \alpha_t)V_t + \alpha_t T_{\mu_t} V_t,$$

where μ_t is the policy at time t, converges to the optimal value function V^* with probability 1, given that the scalar step-size α behaves according to stochastic approximation theory [50]. Similarly, convergence can be guaranteed if the value function update is only performed for one state at a time, given that the states are sampled uniformly over the state space. In the same paper, Tsitsiklis also proves convergence for the TD-method and a variant that can be applied to control problems, both for the case of synchronous value function update. For an asynchronous update, however, when state trajectories are observed or generated with a nonuniform distribution, the same methods are known to be non-convergent in some cases.

What happens if we work with an approximated value function rather than a tabular representation is still an partially open question, but in the light of results presented in this section, the outlook is rather dim. Still, Konda and Tsitsiklis prove in [29] that some form of step-wise TD-update on an approximate value function in combination with an approximated policy based on the same features shows convergent behaviour, even for a special case of continuous state and action spaces. As the result relies heavily on a linear approximation architecture, it is unclear if similar analysis can be performed for the nonlinear function approximation architecture of LCS.

3.3 Temporal-Difference Learning

TD-Learning is a method for policy evaluation that can be used as part of (Optimistic and/or Approximate) Policy Iteration. It is actually a family of algorithms TD(λ) that is parameterised by the scalar λ, with $0 \leq \lambda \leq 1$.

At its core is a sequence of temporally related events with associated predictions, of which the predictions are updated in a step-wise fashion by the temporal difference between the old prediction and the updated prediction. It originates from a reformulation of the Widrow-Hoff rule [55] for multi-step sequences, resulting in TD(1), and is then generalised to TD(λ). From the reinforcement learning perspective, it acts as a multi-step backup operator, in contrast to the single-step backup T_μ at the core of most DP methods. The next sections discuss the TD-method from various different viewpoints, starting with its origin, then on to its application in reinforcement learning, and finishing on how to improve reinforcement learning with TD by using least-squared methods.

3.3.1 The Origins of TD(λ)

In his original paper [45], Sutton introduced TD-Learning as a method to update predictions on events that are temporally related. It is derived by a rewrite of the Widrow-Hoff rule [55], which performs gradient descent on a local approximation of the gradient. Given a state trajectory $\{i_0, i_1, \dots\}$ due to following policy μ, the sequence of rewards $\{r^\mu_{i_0 i_1}, r^\mu_{i_1 i_2}, \dots\}$ and the value function $V_t(i)$ at time t, we use the updated prediction of the value of state i_t, given by $r^\mu_{i_t i_{t+1}} + \gamma V_t(i_{t+1})$, to perform gradient descent on the resulting local approximation error for state i_t,

$$(r^\mu_{i_t i_{t+1}} + \gamma V_t(i_{t+1}) - V(i_t))^2.$$

Following the gradient of the error w.r.t. $V(i_t)$ results in the Widrow-Hoff weight update

$$V_{t+1}(i_t) = V_t(i_t) + \alpha_t (r^\mu_{i_t i_{t+1}} + \gamma V_t(i_{t+1}) - V_t(i_t)), \tag{7}$$

where α_t is the positive scalar step-size at time t. This update modifies the value for the current state i_t based on the current value of the next state i_{t+1}. Since the transition from i_{t+1} to i_{t+2} will update the value for state i_{t+1}, we can also use this knowledge to update the value for state i_t. Performing such a back-propagated update at time $t+1$ for the values of the states $i_0, \dots, i_t$ is the basis of TD-learning.

Given the policy μ, the value function V_t at time t, and the Temporal Difference $d_t(i,j)$ at time t for performing a transition from state i to j,

$$d_t(i,j) = r^\mu_{ij} + \gamma V_t(j) - V_t(i),$$

the TD(λ) update is defined as

$$\begin{aligned} V_{t+1}(i) &= V_t(i) + \alpha_t d_t(i_t, i_{t+1}) e_{t+1}(i_t), \qquad \forall \quad i \in S, && (8) \\ e_{t+1}(i) &= \begin{cases} \lambda\gamma e_t(i) + 1 & \text{if } i = i_t, \\ \lambda\gamma e_t(i) & \text{otherwise,} \end{cases} && (9) \end{aligned}$$

where $e_t \in \mathbb{R}^N$ is the *eligibility trace* vector at time t, of which component $e_t(i)$ gives the eligibility trace for state $i \in S$ at time t. Sutton has shown that for $\lambda = 1$, the above method is equivalent to performing a Widrow-Hoff update on the current and all past states, even if combined with linear function approximation architectures [45]. On the other hand, setting $\lambda = 0$ causes TD-Learning to update only the current state and is therefore equivalent to the local Widrow-Hoff update of Eq. (7). Note that the interpretation of TD-Learning presented so far is called the *Backward View* as it treats TD(λ) as looking backwards in time to update the prediction of all states that it has already visited.

3.3.2 Bias-Variance Tradeoff

A different, but mathematically equivalent interpretation for TD-Learning is the *Forward View*, which treats TD(λ) as looking forward in time and founding the prediction of any state on the observation of all future rewards. Given policy μ and the infinite state trajectory $\{i_0, i_1, \dots\}$, the new value of state i at time t is according to $TD(\lambda)$ estimated by

$$(1 - \lambda) \sum_{m=1}^{\infty} \lambda^{m-1} R_t^{(n)},$$

where $R_t^{(n)}$ is the n-step return at time t, given by

$$R_t^{(n)} = r^{\mu}_{i_t i_{t+1}} + \gamma r^{\mu}_{i_{t+1} i_{t+2}} + \gamma^2 r^{\mu}_{i_{t+2} i_{t+3}} + \dots + \gamma^n V_t(i_{t+n}).$$

Hence, TD(λ) mixes returns of different lengths to generate a new estimate for the current state [47, Ch. 7]. The closer λ is set to 1, the more future reward influences that estimate. A low λ, on the other hand, will cause TD(λ) to rely mainly on the existing estimate V_t of the value of future states.

For $\lambda = 1$, the expected return is the unbiased Monte-Carlo return, which might have a high variance, as it is based on a long stochastic sequence of rewards. $\lambda = 0$ only considers the current reward and the current value estimate of the next state, causing the new estimate to have lower variance (being based on less samples) but introduces a bias by the potential inaccuracy of the current estimate [10]. Hence, the parameter λ controls the *Bias-Variance Tradeoff* of TD(λ). Several empirical studies have demonstrated that intermediate values for λ give the best performance [45, 47].

3.3.3 The Temporal-Difference Operator $T_\mu^{(\lambda)}$

Similar to the DP update operators T and $T\mu$ (Section 3.1.1) for Dynamic Programming, we can introduce an update operator for TD(λ), that we will denote by $T_\mu^{(\lambda)}$, indicating value update by TD(λ) according to policy μ. Given a value vector V, and the state sequence $\{i_0, i_1, \dots\}$ from following policy μ, $T_\mu^{(\lambda)}$ is according to [49] defined by

$$(T_\mu^{(\lambda)} V)(i) = (1 - \lambda) \sum_{m=0}^{\infty} \lambda^m \mathbb{E} \left(\sum_{t=0}^{m} \gamma^t r^{\mu}_{i_t i_{t+1}} + \gamma^{m+1} V(i_{m+1}) | i_0 = i \right),$$

for $\lambda < 1$, and

$$(T^{(1)} V)(i) = \mathbb{E} \left(\sum_{t=0}^{\infty} \gamma^t r^{\mu}_{i_t i_t + 1} | i_0 = i \right) = V^{\mu}(i),$$

for $\lambda = 1$, so that $\lim_{\lambda \uparrow 1} (T^{(\lambda)} V)(i) = (T^{(1)} V)(i)$ (under some technical conditions).

For $\lambda < 1$, the expectation is equivalent to the n-step return V_n^μ, as defined in Section 2.2, and is approximated by the trajectory-based n-step return $R^{(n)}$ of the last section. This shows again that the λ parameter controls the mixing weights for returns of different lengths. If λ is set to 0, the $T_\mu^{(0)}$ is equivalent to the DP operator T_μ for a fixed policy.

Regarding the properties of that operator, it was shown in [49, 6] that $T_m^{(\lambda)} u$ describes a contraction mapping w.r.t. the steady-state distribution due to policy μ; that is, for any $\lambda \in [0, 1]$, and any $V, \bar{V} \in \mathbb{R}^N$,

$$\|T_\mu^{(\lambda)} V - T_\mu^{(\lambda)} \bar{V}\|_D \leq \frac{\gamma(1 - \lambda)}{1 - \gamma\lambda} \|V - \bar{V}\|_D \leq \gamma \|V - \bar{V}\|_D,$$

where D determines the steady-state distribution due to policy μ. Hence, repeatedly applying that operator to a value vector makes it converge to the fixed point of the update, independent of its initial value. Additionally, Berstekas and Tsitsiklis show in [6, Ch. 2] that the $T_\mu^{(\mu)}$ forms a contraction mapping to the maximum norm with a contraction modulus[4] of $\gamma\lambda$. When comparing that to the DP

[4] The contraction modulus determines the strength of the contraction. Given the contracting function f, causing the contraction $\|f(a) - f(b)\| \leq c\|a - b\|$, its contraction modulus is the scalar c.

update T_μ, which has a contraction modulus of γ, we can see that TD-Learning performs at least as much contraction as the DP update, which is particularly helpful as the parameter λ is controllable by the learning system.

3.3.4 Convergence of TD(λ)

The discussion of the Forward View as well as the operator description of TD(λ) both rely on looking into the future for an infinite number of steps, and hence prohibit implementation. However, with the help of eligibility traces, we can use the mathematically equivalent Backward View to describe an implementable algorithms.

Following the update described by Eq. (8) and (9), we perform a step-wise approximation to the update as given by the $T_\mu^{(\lambda)}$ operator. As the state transitions follow the Markov Chain determined by the policy μ, the approximation will asymptotically converge to the iteration

$$V_{t+1} = V_t + \alpha_t D(T^{(\lambda)} V_t - V_t),$$

which is equivalent to the steady-state distribution weighted Widrow-Hoff update for the new estimate $T^{(\lambda)} V_t$.

That observation allows linking of TD(λ) to stochastic approximation theory, as first done in [26]. Given the mapping $H : \mathbb{R}^N \to \mathbb{R}^N$, and some parameter $V \in \mathbb{R}^N$, the *Robbin-Monro* stochastic approximation algorithm

$$V_{t+1} = (1 - \alpha_t) V_{t+1} + \alpha_t H V_t$$

is known to converge to its fixed point $V = HV$, given that the step-size α_t fulfils some stochastic approximation assumptions. In its stationary form, TD(λ) can be described by such an update equation. Its initial deviation from the stationary form can be added as update noise that asymptotically converges to zero. This path was taken in [6, Ch. 5] to prove convergence of TD(λ) with probability 1 to the value function V^μ of the followed policy μ.

Even though our discussion has been kept rather informal, it captures the core of the convergence proofs of most step-wise approximations to DP updates: Firstly, it is shown that the method converges for the synchronous case, that is when all states are updated in the same iteration. For DP and TD-Learning this is ensured by the contraction mapping formed by their update operators. As a second step, the step-wise approximation is modelled as a deviation from the synchronous case that asymptotically converges to zero. The same approach has been used to show convergence of TD(λ) with function approximation [49], and for Least-Squared Policy Evaluation (LSPE) [37].

3.3.5 Approximate Temporal-Difference Learning

So far, we have only discussed TD-Learning with a full representation of the value function in form of a value vector V. What happens to its properties if we replace that vector by its linear approximation $\tilde{V}(i) = w'\phi(i)$ for any $i \in S$?

Firstly, we need to adapt the TD(λ) update in terms of the function approximation used, as was already done when TD-Learning was first proposed [45]. We will use the description of [49], which gives the temporal difference d_t at time t for policy μ and the state sequence $\{i_0, i_1, \dots\}$ by

$$d_t = r_{i_t i_{t+1}^\mu} + \gamma w_t' \phi(i_{t+1}) - w_t' \phi(i_t),$$

where w_t is the approximation's weight vector at time t. That weight vector is updated according to TD(λ) by

$$w_{t+1} = w_t + \alpha_t d_t \sum_{m=0}^{t} (\gamma\lambda)^{t-m} \phi(i_m).$$

As that would require remembering past states to evaluate $\phi(i_m)$, we can again use the eligibility trace vector $e_t \in \mathbb{R}^L$ to rewrite the update as

$$\begin{aligned} w_{t+1} &= w_t + \alpha_t d_t e_t, \\ e_{t+1} &= \sum_{m=0}^{t+1} (\gamma\lambda)^{t+1-m} \phi(i_m) = \gamma\lambda e_t + \phi(i_{t+1}), \end{aligned}$$

initialised with $e_{-1} = 0$. Due to the linear architecture's separation of state-dependent features and their mixing weights, most of the state-dependencies are moved to the trace vector e_t. This separation allows is to update values of past states without remembering the whole trajectory. In TD-Learning without value function approximation this is only possible by updating all states at once at the end of the trajectory (called *off-line* TD-Learning). As we are dealing with discounted problems without a terminal state, there is no end to the trajectory, and we have to update the state values while passing by. Even though that method is still convergent, it is only the case for decreasing step-sizes α_t, as that also reduces the noise that is introduced by the *on-line* update. Since for linear architectures we can produce accumulated state values of past states with the help of eligibility traces, this noise does not occur.

Similar but not equal to TD-Learning without function approximation, approximate TD-Learning performs a step-wise approximation of the steady-state iteration

$$w_{t+1} = w_t + \alpha_t \Phi' D(T_\mu^{(\lambda)}(\Phi w_t) - \Phi w_t).$$

As analysed in [49], for the case of $\lambda = 1$ the iteration describes a steepest descent along the gradient of

$$\sum_{i \in S} \pi(i)(V^\mu(i) - w'\phi(i))^2,$$

which is known to converge for adequate step-size settings. For $\lambda < 1$, above iteration follows the steepest descent of the time-variant function

$$\sum_{i \in S} \pi(i) \left((T_\mu^{(\lambda)}(\Phi w_t))(i) - w'\phi(i) \right)^2,$$

which makes sense if we see $T_\mu^{(\lambda)}(\Phi w_t)$ as an approximation to V^μ.

Both versions aim to minimise a convex function, of which the optimum can be found by orthogonal projection into the approximation subspace, given by the projection matrix Π_D (Eq. (5)). Hence, the steepest descent at time t aims to find $\Pi_D T_\mu^{(\lambda)} \tilde{V}_t$, where we use $\tilde{V}_t = \Phi w_t$. That lets us introduce a replacement algorithm of the form

$$\tilde{V}_{t+1} = \Pi_D T_\mu^{(\lambda)} \tilde{V}_t,$$

which gives the optimal approximation at each iteration. We already know that $T_\mu^{(\lambda)}$ describes a contraction mapping w.r.t. D, the steady-state distribution of the Markov Chain due to policy μ. As shown in [49], the projection matrix Π_D causes a non-expansion on that same norm. Hence, both in combination give a contraction w.r.t. $\| \cdot \|_D$, and the iteration converges to the fixed point of its update $\tilde{V}^\mu = \Pi_D T_\mu^{(\lambda)} \tilde{V}^\mu$, which is different for different settings of λ. The implemented algorithm is a step-wise approximation to the described iteration. As the difference between the iteration and its approximation decreases asymptotically, the algorithm converges under some realistic assumption with probability 1 [49].

An important finding of the above is that Π_D only causes a non-expansion on $\| \cdot \|_D$ if the states are sampled according to steady-state distribution. As this distribution is usually not known beforehand, we have to follow the state trajectory as it would occur by following the state transitions of the problem Markov Decision Process. If the states are sampled according to another distribution, the non-expansion w.r.t. $\| \cdot \|_D$ cannot be guaranteed anymore and divergence can occur, as demonstrated by counter-examples in [25, 11, 23, 48]. That the condition of on-line sampling is sufficient but not necessary for the convergence of approximate synchronous TD-Learning is shown in [42], where they reduce the algorithm to a form of matrix iteration. We will later demonstrate that LCS with TD-Learning can also violate that condition but still converge.

3.3.6 Least-Squares Methods

With the better understanding of TD-Learning, two variants of TD(λ) emerged that feature significantly better convergence rates by replacing the local gradient descent by a direct evaluation of the minimum approximation error.

The first method, called Least-Squares TD-Learning (LSTD(λ)), works from the convergence point backwards and introduces a new algorithm that directly approximates that convergence point. The method was introduced for $\lambda = 0$ by Bradtke and Barto [13], and later extended to $\lambda \in [0, 1]$ by Boyan [9, 10]. It uses the solution to the fixed point $\tilde{V}^\mu = \Pi_D T_\mu^{(\lambda)} \tilde{V}^\mu$, which is also the solution to $Aw + b = 0$, where

$$A = \sum_{t=0}^{\infty} e_t(\phi(i_t) - \phi(i_{t+1}))', \qquad b = \sum_{t=0}^{\infty} e_t r^\mu_{i_t i_{t=1}},$$

and e_t is the eligibility trace vector, given by

$$e_t = \sum_{m=0}^{t} (\gamma\lambda)^{t-m} \phi(i_m).$$

Matrix A and vector b can be incrementally updated, giving A_t and b_t at time t. Hence, the value function approximation a time t is the solution to $A_t w_t + b_t = 0$, given by $w_t = A_t^{-1} b_t$. To avoid taking the inverse of A_t at each step, we can directly update the inverse by use of the Sherman-Morrison formula [37]. Either way, the incremental update of both A_t and b_t converges to A and b, and therefore LSTD(λ) as a whole converges with probability 1 [37]. Due to the change of algorithm, the requirement of TD(λ) for sampling by the steady-state distribution is not significant anymore. Instead, an arbitrary sampling distribution will still lead to convergence, as long as every state is visited infinitely often. [41].

An interesting observation is that LSTD(λ) has the same structure as an approach that builds an observation-based model of the environment and then uses that model to derive the approximate value function $\tilde{V}^\mu$ [10]. The vector b is responsible for storing an approximation of the expected return for each state. An approximation of the observed state transitions are captured by the matrix A. If Φ is an $N \times N$ identity matrix, that is if the approach is tabular, then LSTD(0) is equivalent to learning an exact model of the environment. For any form of function approximation, LSTD(λ) creates a compressed model in correspondence with the feature vectors. As a side-note, LSTD(1) is also mathematically equivalent to linear regression without the same excessive use of resources [10].

The other recently introduced Least-Squares method is Least-Squares Policy Evaluation (LSPE) [37, 5], a method that closely follows the TD(λ) update. Indeed, at every time t it aims at finding the $\bar{w}_t$ that minimises

$$\sum_{m=0}^{t} \left(\bar{w}_t' \phi(i_m) - w_t' \phi(i_m) - \sum_{n=m}^{t} (\gamma\lambda)^{n-m} d_t(i_n, i_{n+1}) \right)^2,$$

where $d_t(i,j)$ is the temporal difference, given by

$$d_t(i_n, i_{n+1}) = r^\mu_{i_n, i_{n+1}} + \gamma w_t' \phi(i_{n+1}) - w_t' \phi(i_n).$$

While TD(λ) performs local gradient descent on above function, LSPE computes the minimum of the above function by an iterative matrix update. The resulting $\bar{w}$ is used to update the approximation weights by

$$w_{t+1} = w_t + \alpha(\bar{w}_t - w_t),$$

where α is the scalar step-size. Thus, rather than strictly following the optimal approximation, which would be the case for $\alpha = 1$, the algorithm also allows for more gradual weight updates. According to [5], that is an advantage that LSPE has over LSTD(λ), as it allows LSPE to be used with Optimistic Policy Iteration where a small step-size is essential for good overall performance. Otherwise, LSPE and LSTD(λ) converge to each other faster than they converge to the optimal solution, given that $\alpha = 1$. What is not documented is that LSPE is computationally and spatially more expensive as it needs to maintain one additional $L \times L$ matrix.

With respect to convergence, LSPE can be reduced to a step-wise approximation to a matrix iteration. As the difference between the iteration and its approximation converges to zero with infinity, the method converges if the matrix iteration converges. This is shown to be the case if the step-size is within a particular range that always contains 1 [5]. Due to its similarity to TD(λ), the proof for LSPE is based on the assumption that the state transitions are distributed according to the steady-state distribution for the current policy. That requirement is another drawback of LSPE when compared to LSTD(λ).

3.4 Optimal Control and Q-Learning

All above methods require some model of the problem to create policies based on the current value function. However, as already shown in Section 2.3, we can use the Q-value function rather than the value function to improve the policy without having a model of the problem. In addition to that, we need to use some step-wise update on that Q-value function as full update of all states also requires a model.

In this section we will introduce SARSA(λ) and Q-Learning. The first performs Policy Iteration and uses TD-Learning to update the Q-value function. Q-Learning is a step-wise approximation to Value Iteration.

Both methods require visiting all states an infinite number of times for convergence. However, policies that always select the best action (so-called *greedy* policies) might not cover the whole state space. Hence, those methods need to implement some form of a soft policy, like ϵ-greedy or a softmax policy, which sometimes also choose sub-optimal actions. We will not discuss details about those policies here, but the interested reader is referred to [47] for more information.

3.4.1 SARSA(λ)

SARSA stands for State-Action-Reward-State-Action, as SARSA(0) requires only information on the current and next state-action pair and the reward that was received for the transition. The name was coined by Sutton [46] for an algorithm developed by Rummery and Niranjan [39] in its approximate form, which is very similar to Wilson's ZCS [56], as noted by Kovacs [30].

It performs Optimistic Policy Iteration on a Q-value function that is updated by TD(λ). As the value update is based on the state trajectory of the current policy, this method is an *on-policy* method. Due to its use of Optimistic Policy Iteration, the convergence properties discussed in Section 3.2.3 apply. An additional investigation that shows convergence of SARSA(0) under certain policies is available in [44]. For a description of how to implement SARSA(λ), the interested reader is referred to [47]. Using linear function approximation on the Q-value function has the same effect as using approximate TD-learning, which was discussed in Section 3.3.5. The requirement of on-line sampling is always fulfilled as the sequence of observations is the only information that is used.

3.4.2 Q-Learning

The much-celebrated Q-Learning was developed by Watkins [53] as the result of combining TD-Learning and DP methods. It is similar to SARSA(0), but rather than using the Q-value of the next state-action pair to update the Q-value of the last state-action pair, it uses the Q-value that would result from following a greedy policy, even though that is not necessarily the case. Hence, Q-Learning is called an *off-policy* method.

For the sequence of states $\{i_0, i_1, \dots\}$ and the corresponding sequence of actions $\{a_0, a_1, \dots\}$, the Q-values are updated by

$$Q_{t+1}(i_t, a_t) = Q_t(i_t, a_t) + \alpha_t \left(r_{i_t, i_{t+1}}(a_t) + \gamma \max_{a \in A} Q_t(i_{t+1}, a) - Q_t(i_t, a_t) \right).$$

Hence, the estimate for $Q(i_t, a_t)$ is updated by $r_{i_t i_{t+1}}(a_t) + \gamma V_t^*(i_{t+1})$, where $V_t^*(i_{t+1}) = \max_{a \in A} Q_t(i_{t+1}, a)$ is the current estimate for the next state i_{t+1} when following a greedy policy. This shows that Q-Learning is an approximation to Asynchronous Value Iteration that performs the update with the actual reward rather than its expectation. Consequently, Q-Learning is guaranteed to converge to the optimal Q^*-values, given that all state-action pairs are visited an infinite number of times [54].

A variant of Q-Learning, called Q(λ) is an extension that uses eligibility traces like TD(λ) as long as it performs on-policy actions [54]. With the choice of an off-policy action, all traces are reset to zero, as the off-policy action breaks the temporal sequence of predictions. Hence, the performance increase due to traces depends significantly on the policy that is used, but is usually marginal.

As Q-Learning is a step-wise approximation of Asynchronous Value Iteration, function approximation architectures for which the latter diverges will very likely not work with Q-Learning (see Section 3.2.1). This also applies for linear approximation architectures, for which Q-Learning was demonstrated to diverge in some cases [12].

4 Reinforcement Learning with LCS

In this section we will show how to construct Learning Classifier Systems based on reinforcement learning that uses an LCS function approximation architecture introduced in [22]. For now, we will restrict ourselves to a time-invariant population of classifiers, but investigations on how such a system reacts to the replacement of classifiers in the population is the next logical step of our research.

We will firstly give a short overview of the LCS function approximation architecture and how it can be related to reinforcement learning methods. Subsequently, we will show how it can be applied to model-based and model-free Value Iteration and Policy Iteration. Finally, convergence of one type of such a system is discussed, followed by an outline of possible further work.

4.1 LCS Function Approximation Architecture

We have introduced a formal framework and extensions to the LCS function approximation architecture in [22]. Here we will show how it can be applied to reinforcement learning.

4.1.1 The Framework

LCS utilise a finite set of K classifiers to approximate the value function. We will enumerate the classifiers with $1, \dots, K$, and denote a classifier parameter of classifier k by the subscript $\cdot_k$.

Each classifier k *matches* a particular subset $S_k \subseteq S$ of the state space S, which we have called the *match set*. The aim of classifier k is to find the optimal approximation in the mean-squared sense of the parts of the value function that it matches. To ease notation, we use the indicator function $I_{S_k} : S \to 0, 1$ that returns $I_{S_k}(i) = 1$ if $i \in S_k$ and $I_{S_k}(i) = 0$ otherwise. The approximation of classifier k is determined by its weight vector $w_k \in \mathbb{R}^L$, which is used to approximate the value for state i by $\tilde{V}_k(i) = w_k' \phi(i)$. Additionally, each classifier keeps track of its own approximation error, that we denote by ε_k.

To recover an approximation $\tilde{V} : S \to \mathbb{R}$ over the whole state space, the classifier's individual approximation is mixed by

$$\tilde{V}(i) = \sum_{k=1}^{K} \psi_k(i) \tilde{V}_k(i), \tag{10}$$

where $\psi_k : S \to [0, 1]$ is the mixing weight for classifier k and is given by

$$\psi_k(i) = \frac{I_{S_k}(i) \varepsilon_k^{-\nu}}{\sum_{p=1}^{K} I_{S_p}(i) \varepsilon_k^{-\nu}}. \tag{11}$$

ν is a positive constant that allows additional control over the mixing weights. Hence, classifiers are weighted by an inverse of their estimates approximation error, and only contribute to the approximation if they match the current state. The mixing weights are undefined for states that no classifier matches. We will assume that for each state there exists at least one classifier that matches that state, to avoid that problem. For demonstrations of how to use this framework and a detailed discussion about the optimality of a classifier see [22].

In matrix notation, the approximated value vector $\tilde{V}_k$ of classifier k is given by $\tilde{V}_k = \Phi w_k$. Matching of the same classifier is expressed through the $N \times N$ diagonal matching matrix I_{S_k} with $I_{S_k}(1), \dots, I_{S_k}(N)$ along its diagonal. Note that due to binary matching, $(I_{S_k})^a = I_{S_k}$ for all $a \in \mathbb{R}_{\neq 0}$. The sampling distribution w.r.t. classifier k is given by the sampling matrix $D_k = I_{S_k} D$. The mixing weights are represented by the $N \times N$ diagonal mixing matrix Ψ_k with diagonal entries $\psi_k(1), \dots, \psi_k(N)$. Due to our definition of the mixing weights, for any classifier k, $\Psi_k = I_{S_k} \Psi_k$, and $\sum_{k=1}^{K} \Psi_k = I$. The combined approximation $\tilde{V}$ is given by

$$\tilde{V} = \sum_{k=1}^{K} \Psi_k \tilde{V}_k = \sum_{k=1}^{K} \Psi_k \Phi w_k.$$

This approximation is a result of the approximation of all classifiers, and should *not* be optimised as a whole as that would distort the approximation of the separate classifiers [22].

4.1.2 Relating States

Any reinforcement learning method presented here is based on relating the value of the current state to the value of one or more following states. The values of the states cannot be directly observed but are only an artifact of the DP solution to an MDP problem, emerging through the reward function and the relation between states.

In LCS, each classifier approximates its own part S_k of the state space S, but might have many states that it does not match. Let us consider a transition from state i_t to state i_{t+1} by performing action a_t, where classifier k matches the first state but not the second, that is $i_t \in S_k$ and $i_{t+1} \notin S_k$. That implies that the classifier provides an approximation $\tilde{V}_k(i_t)$ for the value of the first state i_t, but its approximation $\tilde{V}_k(i_{t+1})$ for the second state i_{t+1} is unreliable as the classifier does not aim at approximating it. Hence, to update $\tilde{V}_k(i_t)$ the classifier has to rely on another approximation than its own. For that purpose we will use the combined approximation $\tilde{V}(i_{t+1})$ that reflects our best estimate of the value approximation of that state. Hence, the new estimate for $\tilde{V}_k(i_t)$ will be the reward for the transition and the discounted value of the next state, that is $r^{\mu}_{i_t i_{t+1}} + \gamma \tilde{V}(i_{t+1})$, given that we are following policy μ.

Generally, we will use that new estimate for all updates, regardless of whether the classifier matches the next state or not. This is justified by observing that the overall approximation is on average more accurate than the approximation of a single classifier. Given, for example, that we have a classifier that matches a large area of the state space, then this classifier will without doubt have a larger approximation error than a classifier that matches a subset of that space. Hence, the mixed approximation for the states where both classifiers match will be more accurate than the approximation of the first classifier. It will only differ slightly from the approximation of the second classifier, as the approximation error determines the mixing weights.

4.2 Value Iteration

The method of Value Iteration is based on repeatedly performing the DP update T on the current value function estimate. If used without approximation, it is guaranteed to converge to the optimal value function V^*. In the next few sections we will develop some variants of Value Iteration in combination with LCS function approximation, and will discuss their likelihood of convergence.

4.2.1 LCS Value Iteration

In the case of LCS, each classifier approximates the result of one Value Iteration update $T\tilde{V}_t$ based on the overall value function approximation $\tilde{V}_t$. Hence, we want find $\bar{V}_k$ for which

$$\sum_{i \in S_k} \left((T\tilde{V}_t)(i) - \bar{V}_k(i) \right)^2 = \| T\tilde{V}_t - \bar{V}_k \|^2_{I_{S_k}}$$

is minimal. We can compute that minimum by performing an orthogonal projection into the approximation subspace $\{I_{S_k}\Phi w : w \in \mathbb{R}^L\}$ of classifier k, given by the projection matrix $\Pi_{I_{S_k}}$ (Eq. (5)). Hence, one Value Iteration update becomes

$$\tilde{V}_{k,t+1} = \Pi_{I_{S_k}} T\tilde{V}_t, \qquad k = 1, \dots, K,$$

which results in the weight update

$$\begin{aligned} w_{k,t+1} &= \left(\sum_{i \in S_k} \phi(i)\phi(i)' \right)^{-1} \sum_{i \in S_k} \phi(i)(T\tilde{V}_t)(i) \\ &= \left(\sum_{i \in S_k} \phi(i)\phi(i)' \right)^{-1} \sum_{i \in S_k} \phi(i) \max_{a \in A} \sum_{j \in S} p_{ij}(a)(r_{ij}(a) + \gamma \tilde{V}_t(j)) \\ &= \left(\sum_{i \in S_k} \phi(i)\phi(i)' \right)^{-1} \sum_{i \in S_k} \phi(i) \max_{a \in A} \sum_{j \in S} p_{ij}(a) \left(r_{ij}(a) + \gamma \phi(j)' \sum_{k=1}^{K} \psi_{k,t}(j) w_{k,t} \right), \end{aligned}$$

where we minimise above approximation error w.r.t. w_k and substitute for the DP update T (Eq. (6)) and the overall approximation $\tilde{V}$ (Eq. (10)). The mixing weights $\psi_{k,t}$ are given by Eq. (11) and are based on the approximation error $\varepsilon_{k,t}$, which is

$$\varepsilon_{k,t} = \frac{1}{|S_k|} \sum_{i \in S_k} \left((T\tilde{V}_{t-1})(i) - w'_{k,t}\phi(i) \right)^2,$$

where $|S_k|$ returns the number of elements in S_k which is the number of states that classifier k matches. Note that for the update at time t we have to use the approximation error from time $t-1$, as we cannot evaluate the error at the same time as using it for the mixing weight to assemble the overall approximation $\tilde{V}$. The error can only be updated once the mixing weights are known and therefore always lags one step behind. Furthermore, we should not rely on the current overall approximation $\tilde{V}_t$ to calculate the error, as that approximation is less accurate than the DP update $T\tilde{V}_{t-1}$ based on the previous approximation. Overall, it might be most efficient to update the error at the same time as updating the weight vector (using the mixing weights based on the previous error) so that we do not need to store information to recover the previous overall approximation $\tilde{V}_{t-1}$.

As already discussed in Section 3.2.1, Approximate Value Iteration might diverge if used in combination with linear approximation architectures. Hence, it might only be safe to apply if we use the features $\phi(i) = (1)$ for all $i \in S$. This makes the classifiers to be averagers, for which Approximate Value Iteration is known to converge [23]. By averaging over the classifier's approximation to form the overall value approximation, it makes it very likely that the whole function approximation architecture acts as an averager and allows us to guarantee convergence. On the other hand, using other features might cause the method to diverge. Further work on that topic will allow us to give more definite statements about the convergence behaviour of LCS Value Iteration.

4.2.2 Asynchronous LCS Value Iteration

Rather than updating the value function of all states at once, Asynchronous LCS Value Iteration only updates the value function of a subset of all states. We will develop the method as updating only one state at a time, which we consider to be state i_t at time t. The new value estimate for that state is given by the DP update $(T\tilde{V}_t)(i_t)$ and concerns only classifiers that match that state.

In contrast to completely reevaluating the approximation of each classifier at each iteration, as done in LCS Value Iteration, we now only update the value estimate for one state and therefore have to perform an iterative update of the function approximation without discarding past information. As the estimate at time t is given by $(T\tilde{V}_t)(i_t)$ and classifier k only performs updates for the states that it matches, its approximation at time t aims at minimising[5]

$$\sum_{m=0}^{t} I_{S_k}(i_m)((T\tilde{V}_m)(i_m) - w'_{k,t}\phi(i_m))^2.$$

Consequently, the minimisation is dependent on the distribution of states that we update. In the long run that causes the approximation costs to be weighted by the state distribution D, that is

$$\sum_{i \in Sk} \pi(i)((T\tilde{V})(i) - w'_k\phi(i))^2 = \|T\tilde{V} - \Phi w_k\|^2_{D_k}.$$

Hence, minimising that cost gives a step-wise approximation to the iteration

$$\tilde{V}_{k,t+1} = \Pi_{D_k} T\tilde{V}_t,$$

which differs from LCS Value Iteration by the distribution weighting. In terms of the overall approximation, the iteration becomes

$$\tilde{V}_{t+1} = \sum_{k=1}^{K} \Psi_{k,t+1} \Pi_{D_k} T\tilde{V}_t,$$

which is a weighted mixture of the orthogonal projection of the DP update into the approximation subspaces of the classifiers.

Implementation possibilities are to use the LMS algorithm to perform local gradient descent on the current error $I_{S_k}(i_t)((T\tilde{V}_t)(i_t) - w'_{k,t}\phi(i_t))^2$ and track the approximation error, or to use a Kalman-filter based update to accurately track both the optimal approximation and its approximation error. Both algorithms are described in [22], and their application will be demonstrated in the next section.

With respect to the method's convergence properties, we expect the difference between Asynchronous LCS Value Iteration and LCS Value Iteration to asymptotically converge to zero (ignoring the difference in sampling distribution). Hence, the discussion of the convergence of LCS Value Iteration should also apply to the asynchronous variant.

4.2.3 LCS Q-Learning with Implementations

Even though the previous method only updates one state at a time, it still required the evaluation of the DP update $(T\tilde{V})(i_t)$ at each step. However, if we choose our actions according to a greedy policy and follow the transitions of the Markov Chain, the received rewards will in the long run be similar to the ones that correspond to the DP update T (Eq. (6). Hence, we can replace the DP update $T\tilde{V}_t(i_t)$ of the previous method by

$$r_{i_t i_{t+1}}(a_t) + \gamma \max_{a \in A} \sum_{j \in S} p_{i_{t+1} j}(a)\tilde{V}_t(j),$$

[5] Even though it would be better to use $\tilde{V}_t(i_m)$ rather than $\tilde{V}_m(i_m)$, we cannot separate the state information from the overall approximation as the mixing weights might change over time. Hence, to use $\tilde{V}_t(i_m)$ would require the performance of the complete minimisation at every step and does not allow for an iterative update.

where a_t is chosen in accordance with the greedy policy. That still requires consideration of all transitions from i_{t+1} to compute the value of the second term. We can avoid that by operating with Q-values rather than the value function itself, and reduce above to

$$r_{i_t i_{t+1}}(a_t) + \gamma \max_{a \in A} \tilde{Q}_t(i_{t+1}, a),$$

which essentially gives Q-Learning. Even though it increases the spatial requirements, because we need to store one value function per possible action, it does not require a model of the problem. For the sake of this discussion we will assume that one classifier only matches one action, as is usually the case, which is why it is sufficient to keep the classifier approximation $\tilde{V}_k$ action-independent. Hence, we will only modify the matching indicator function to $I_{S_k} : S \times A \rightarrow \{0, 1\}$, returning only 1 for the actions that the classifier matches, the mixing weights $\psi_k : S \times A \rightarrow [0, 1]$ to also consider the actions, and will define the overall Q-value approximation by

$$\tilde{Q}(i, a) = \sum_{k=1}^{K} \psi_k(i, a) \tilde{V}_k(i),$$

with the mixing weights

$$\psi_k(i, a) = \frac{I_{S_k}(i, a) \varepsilon_k^{-\nu}}{\sum_{p=1}^{K} I_{S_p}(i, a) \varepsilon_k^{-\nu}}.$$

An extension to this would be to allow a classifier to approximate values for several actions, made possible by introducing action-dependent feature vectors.

The error we want to minimise is the sequence of temporal differences

$$\sum_{m=0}^{t} I_{S_k}(i_m, a_m) \left(r_{i_m i_{m+1}}(a_m) + \gamma \max_{a \in A} \tilde{Q}_m(i_{m+1}, a) - w'_{k,t} \phi(i_m) \right)^2 . \tag{12}$$

To avoid having to store the sequence of past states, we will employ an iterative update procedure.

Using the normalised LMS algorithm, we will perform local gradient descent w.r.t. the current error[6]. That gives the weight update

$$w_{k,t+1} = w_{k,t} + \alpha_t I_{S_k}(i_t, a_t) \frac{\phi(i_t)}{\|\phi(i_t)\|^2} \left(r_{i_t i_{t+1}}(a_t) + \gamma \max_{a \in A} \tilde{Q}_t(i_{t+1}, a) - w'_{k,t} \phi(i_t) \right), \tag{13}$$

where α_t is the step-size at time t. Hence, only the matching classifiers are updated. Besides the difference in the mixing weight computation, the algorithm is equivalent to the one used in XCSF [59].

The approximation error can be updated by the same LMS algorithm, performing gradient descent on the local approximation error

$$I_{S_k}(i_t, a_t) \left(\left(r_{i_t i_{t+1}}(a_t) + \gamma \max_{a \in A} \tilde{Q}_t(i_{t+1}, a) - w'_{k,t} \phi(i_t) \right)^2 - \varepsilon_{k,t} \right)^2$$

to get the error update

$$\varepsilon_{k,t+1} = \varepsilon_{k,t} + \alpha_t I_{S_k}(i_t, a_t) \left(\left(r_{i_t i_{t+1}}(a_t) + \gamma \max_{a \in A} \tilde{Q}_t(i_{t+1}, a) - w'_{k,t} \phi(i_t) \right)^2 - \varepsilon_{k,t} \right).$$

That completes the algorithmic description for LCS Q-Learning with the LMS algorithm.

A more powerful alternative is to use the Kalman filter to track both the optimal approximation and the approximation error. Minimising the temporal difference sequence, given by Eq. (12), reveals that the optimal weight vector $w_{k,t+1}$ for classifier k after the transition $i_t \rightarrow^{a_t} i_{t+1}$ satisfies

$$\left(\sum_{m=0}^{t} I_{S_k}(i_m, a_m) \phi(i_m) \phi(i_m)' \right) w_{k,t+1} = \sum_{m=0}^{t} I_{S_k}(i_m, a_m) \phi(i_m) \left(r_{i_m, i_{m+1}}(a_m) + \gamma \max_{a \in A} \tilde{Q}_m(i_{m+1}, a) \right).$$

Of the several possible algorithmic forms of tracking this optimum, we will use the one described in [22, Sec. 4.3.6]. The approach is to observe that above optimality condition is of the form $A_{k,t} w_{k,t+1} = b_{k,t}$,

[6] For more information on the use of the normalised LMS algorithm in LCS, see [22].

where $A_{k,t}$ is an $L \times L$ matrix, and $b_{k,t}$ is a vector of size L. Hence, if we have knowledge of $A_{k,t}$ and $b_{k,t}$, we can recover $w_{k,t+1}$ by

$$w_{k,t+1} = A^{-1}_{k,t} b_{k,t}.$$

$b_{k,t}$ can be iteratively updated by

$$b_{k,t} = b_{k,t-1} + I_{S_k}(i_t, a_t)\phi(i_t)\bar{Q}_t(i_t),$$

initialised with $b_{k,-1} = 0$, where $\bar{Q}_t(i_t)$ is the expected return for state i_t, given by

$$\bar{Q}_t(i_t) = r_{i_t,i_{t+1}}(a_t) + \gamma \max_{a \in A} \tilde{Q}_t(i_{t+1}, a).$$

To avoid inversion of $A_{k,t}$ at each step, we can apply the Sherman-Morrison formula to directly operate on the inverse, that is

$$A^{-1}_{k,t} = A^{-1}_{k,t-1} - I_{S_k}(i_t, a_t) \frac{A^{-1}_{k,t-1}\phi(i_t)\phi(i_t)' A^{-1}_{k,t-1}}{1 + \phi(i_t)' A^{-1}_{k,t-1}\phi(i_t)},$$

where $A^{-1}_{k,-1}$ is initialised to δI, with δ being a small constant.

The approximation error can be tracked according to [22, Th. 4.1] by

$$(c_{k,t+1} - 1)\varepsilon_{k,t+1} = (c_{k,t} - 1)\varepsilon_{k,t} + I_{S_k}(i_t, a_t) \left(\bar{Q}_t(i_t) - w'_{k,t}\phi(i_t)\right) \left(\bar{Q}_t(i_t) - w'_{k,t+1}\phi(i_t)\right),$$

with $\varepsilon_{k,-1} = 0$, where $c_{k,t}$ is the match count for classifier k, defined as

$$c_{k,t} = \sum_{m=0}^{t} I_{S_k}(i_m, a_m).$$

This completes the algorithmic description for LCS Q-Learning with the Kalman filter. A mathematically similar weight-update has already been used in [32], but in that XCS variant the error was approximated by the LMS algorithm. The presented algorithm tracks the exact mean-squared error and can therefore be expected to give a quicker and more accurate error approximation.

Both algorithms describe an approximation to LCS Value Iteration. Hence, we can assume that the same convergence constraints that apply to Value Iteration also apply to those algorithms. Additionally, they require investigation of whether the step-wise approximation is in conformity with LCS Value Iteration in order for their difference to converges to zero.

Even though LCS are mostly applied to complete state trajectories, a set of independent state transitions is sufficient for using this algorithm. Examples of how this can be done for a Least-Squares reinforcement learning method can be found in [31].

4.2.4 XCS with Gradient Descent?

Inspired by [47, Ch. 8.2], Butz, Goldberg and Lanzi attempt in [18] to add a gradient-descent like update to the Q-Learning of XCS by multiplying the residual term of the update by the derivate of the Q-value function w.r.t. the weight vector of the corresponding classifier. What they do not consider is that in XCS each classifier approximates its value function independently. Hence, the derivative of Q is to be taken of the classifier's approximation $\tilde{Q}_{k,t}$ rather than the combined approximation $\tilde{Q}_t$ of all classifiers. The derivative of $\tilde{Q}_{k,t}$ is $\phi(i_t)$ at time t, and is therefore 1 for XCS's feature vector of $\phi(i) = (1)$, leaving the update equation unchanged.

In Butz, Goldberg and Lanzi's derivation, they add a factor inversely proportional to the approximation error to the update equation. Surprisingly, this factor improves XCS performance significantly. Our intuitive explanation for the observed effect is that the changed update strongly supports over-specific classifiers and does not allow for sufficiently general classifiers. As more specific classifiers have a lower approximation error, the additional update factor will have a higher value than for more general classifiers, hence supporting the Q-value update of more specific classifiers. As a result, more general classifiers will have an overly high error, as their approximation is very slowly updated. Consequently, those classifiers are easily removed from the population, and the over-specific classifiers are replicated. What follows is an accurate approximation due to many specific classifiers, but very little generalisation. As no population analysis was published in [18], we cannot check the validity of our argument.

In [51], Wada et al. investigate the gradient term introduced by Butz, Goldberg and Lanzi, and argue that the term is not valid as it refers to the approximation error which is a function of the approximation itself. What Wada et al. ignore is that it is completely valid to use a local and temporary approximation of the gradient, as used in the well known Widrow-Hoff rule [55], also known as the LMS algorithm. They proceed by investigating XCS with different combinations of standard and residual gradient descent, but derive their gradients from the combined approximation rather than from the classifier's approximation, which is incorrect for the reasons discussed above. Our update Equation (13) for Q-Learning in XCS is derived from first principles and uses a normalised form (by the additional factor $\|\phi(i_t)\|^{-2}$) of local gradient descent. This demonstrates that no additional factor is required to make Q-Learning in XCS conform to a gradient descent update.

4.2.5 Directly solving Bellman's Equation

Particularly for testing new algorithms, it is useful to directly find the solution to Bellman's Equation (3). As in combination with function approximation this solution depends on the function approximation architecture, we have to solve it by including the LCS architecture.

As previously described, the value estimates of an individual classifier $\tilde{V}_k$ are backed up by the reward and the value estimate of the overall approximation $\tilde{V}$. That gives for Bellman's Equation with LCS function approximation

$$\tilde{V}_k^*(i) = \max_{a \in A} \mathbb{E}\left(r_{ij}(a) + \gamma \tilde{V}^*(j) | i, a\right) = \max_{a \in A} \sum_{j \in S} p_{ij}(a) \left(r_{ij}(a) + \gamma \sum_{p=1}^{K} \psi_p(j) \tilde{V}_p^*(j)\right).$$

The mixing weights ψ_k are some normalised inverse of the approximation error ε_k, which can be given by

$$\varepsilon_k = \frac{1}{|S_k|} \sum_{i \in S_k} \left(\max_{a \in A} \sum_{j \in S} p_{ij}(a) \left(r_{ij}(a) + \gamma \sum_{p=1}^{K} \psi_p(j) \tilde{V}_p^*(j) \right) - \tilde{V}_k^*(i) \right)^2 .$$

Therefore, the mixing weights are nonlinearly related to the classifier's approximation, which makes the whole Bellman Equation nonlinear and not directly solvable.

Although this is a problem, we might get around it with an iterative procedure. Given that the classifier errors are held fixed, the Bellman Equation with LCS function approximation is linear and can be solved. Therefore, we can alternate between solving the Bellman Equation for fixed error values and updating the error values. Due to the increasingly more accurate approximation error estimate we can expect that iterative update to converge.

An alternative approach to solving the Bellman Equation is to use the iteration derived for LCS Value Iteration, which can be written as

$$\tilde{V}_{t+1} = \sum_{k=1}^{K} \Psi_{k,t} \Pi_k T \tilde{V}_t.$$

The approximation error $\varepsilon_{k,t}$ to compute the mixing weights $\Psi_{k,t}$ can be evaluated by

$$\varepsilon_{k,t} = |S_k|^{-1} \|T\tilde{V}_t - \tilde{V}_{k,t}\|^2_{I_{S_k}} = |S_k|^{-1} \|T\tilde{V}_t - \Pi_{I_{S_k}} \tilde{V}_t\|^2_{I_{S_k}}.$$

That gives an iterative update procedure on the overall approximation $\tilde{V}$ equal to LCS Value Iteration. The approximation of individual classifiers if given at any time by $\tilde{V}_{k,t} = \Pi_{I_{S_k}} \tilde{V}_t$. Due to its relation to Value Iteration it is questionable if the method converges for anything else than simple averaging classifiers (though not even that is currently guaranteed). If in doubt, we recommend using the iteration that is based on fixed errors rather than the one derived from Value Iteration.

4.3 Policy Iteration

Due to the fragility of Value Iteration w.r.t. some function approximation architectures, we will also investigate how LCS function approximation can be applied to the policy evaluation step of Policy Iteration. That step aims at finding the value function V^μ for a fixed policy μ. Throughout the rest of the section we will consider policy μ as being fixed, and will discuss several possibilities of how to find its value function when using LCS function approximation architectures. For a discussion on the consequences of improving the policy before that policy is fully evaluated see Section 3.2.3.

At its core, policy evaluation facilitates the DP update T_μ for policy μ. Repeatedly applying that update to the current estimate of the value function guarantees convergence to the optimal value function V^μ for a fixed policy μ. When applying function approximation to the value function approximate, our goal becomes to minimise the difference $\|V^\mu - \tilde{V}^\mu\|$ between the optimal value function V^μ and its approximation $\tilde{V}^\mu$. Section 3.2.2 outlines common methods to achieve this.

4.3.1 Model-based LCS Policy Evaluation

Similarly to LCS Value Iteration, we want each classifier to approximate the result of the update $T_\mu \tilde{V}_t$ for the states that it matches. Hence, we want to find $\bar{V}_k$ for classifier k that minimises

$$\sum_{i \in S_k} \left((T_\mu \tilde{V}_t)(i) - \bar{V}_k(i) \right)^2 = \| T_\mu \tilde{V}_t - \bar{V}_k \|^2_{I_{S_k}}. \tag{14}$$

This minimum is given by the orthogonal projection $\Pi_{I_{S_k}}$ (Eq. 5) into the approximation subspace of classifier k, and hence the update becomes

$$\tilde{V}_{k,t+1} = \Pi_{I_{S_k}} T_\mu \tilde{V}_t, \qquad k = 1, \dots, K.$$

Deriving the weight update and updating the classifier error is similar to that for LCS Value Iteration and does not require repetition.

If we perform our approximation on generated samples rather than iterating though all the problem states, the update gets weighted by the sampling distribution D, and gives the iteration

$$\tilde{V}_{k,t+1} = \Pi_{D_k} T_\mu \tilde{V}_t, \qquad k = 1, \dots, K.$$

In terms of the overall value approximation this iteration can be written as

$$\tilde{V}_{t+1} = \sum_{k=1}^{K} \Psi_k \Pi_{D_k} T_\mu \tilde{V}_t.$$

As for Asynchronous LCS Value Iteration, this iteration can be approximated by a step-wise procedure that, at time t, minimises

$$\sum_{m=0}^{t} I_{S_k}(i_m) \left((T_\mu \tilde{V}_m)(i_m) - w'_{k,t} \phi(i_m) \right)^2,$$

with respect to $w_{k,t}$. Possible candidates for an iterative update are the LMS algorithm or a Kalman filter-based approach [22].

Due to the higher stability of Policy Iteration, we could expect the outlined algorithms to be more likely to converge than LCS Value Iteration. We will give more details about the convergence of synchronous policy evaluation in Section 4.4, and will note here that the presented analysis gives the first partial results on the convergence of LCS with such a function approximation architecture.

4.3.2 Step-wise LCS Policy Evaluation

By following the transitions of the Markov Chain due to policy μ, we can approximate the expected return $\mathbb{E}(r^\mu_{ij} + \gamma \tilde{V}_t(j)|i)$, required by the operator T_μ, by the state transitions $i_t \to i_{t+1}$, giving $r^\mu_{i_t i_{t+1}} + \gamma \tilde{V}_t(i_{t+1})$. Hence, for the state sequence $\{i_0, i_1, \dots\}$ we can approximate LCS Policy Evaluation by minimising for classifier k,

$$\sum_{m=0}^{t} I_{S_k}(i_m) \left(r^\mu_{i_m i_{m+1}} + \gamma \tilde{V}_m(i_{m+1}) - w'_{k,t} \phi(i_m) \right)^2,$$

with respect to $w_{k,t}$ at time t. To additionally remove the requirement for a model of the problem, we can use Q-values instead of the value function. Using the same notation as in Section 4.2.3, our minimisation goal becomes

$$\sum_{m=0}^{t} I_{S_k}(i_m, a_m) \left(r_{i_m i_{m+1}}(a_m) + \gamma \tilde{Q}_m(i_{m+1}, a_{m+1}) - w'_{k,t} \phi(i_m) \right)^2,$$

where $a_m = \mu(i_m)$ is chosen according to policy μ.

Applying the LMS algorithm to above minimisation gives SARSA(0) with LCS function approximation. Applying the Kalman filter gives an algorithm similar to LSPE with $\lambda = 0$ and $\alpha = 1$. As the derivation and results are almost equal to the ones in Section 4.2.3 we will not discuss them here.

4.3.3 What about TD(λ)?

In [21] we have empirically tested the effect of introducing eligibility traces to LCS. Our conclusion was that the performance loss due to traces was caused by classifier replacement and the introduction of over-general classifiers. Here we present an additional reason why introducing eligibility traces in LCS can degrade performance.

To perform TD(λ) with linear function approximation we calculate the expected return for state i_m after following the state trajectory $\{i_m, \dots i_t\}$ by

$$\begin{aligned}
&\tilde{V}_t(i_m) + \sum_{l=m}^{t} (\gamma\lambda)^{l-m} \left(r^\mu_{i_l i_{l+1}} + \gamma \tilde{V}_t(i_{m+1}) - \tilde{V}_t(i_m) \right) \\
&\quad = w_t' \phi(i_m) + \sum_{l=m}^{t} (\gamma\lambda)^{l-m} \left(r^\mu_{i_l i_{l+1}} + \gamma w_t' \phi(i_{m+1}) - w_t' \phi(i_m) \right) \\
&\quad = w_t' \phi(i_m) + w_t' \sum_{l=m}^{t} (\gamma\lambda)^{l-m} \left(\gamma \phi(i_{m+1}) - \phi(i_m) \right) + \sum_{l=m}^{t} (\gamma\lambda)^{l-m} r^\mu_{i_l i_{l+1}}.
\end{aligned}$$

That shows how to separate the approximation parameter w_t from the state-dependent values $\phi(i_m)$ and $r^\mu_{i_l i_{l+1}}$, and allows us to use w_t to calculate the approximation for previous states, effectively reevaluating their values given the current knowledge.

In LCS, the overall approximation of a state value is the mixed approximation of all matching classifiers. At time t, the values for $\tilde{V}_t$ are given by the values of $\tilde{V}_{k,t}$ for all matching classifiers. An update of those approximations concerns not only the classifiers but also reevaluates the mixing weights for the overall approximations. Even though we are able to calculate all state values using the current approximation, there is no known efficient implementation that allows us to update the state values of past states without having to store the state trajectory. Our previous implementation, as described in [21], does not honour the change of mixing weights and therefore introduces additional errors. We believe that it is not possible to find such an implementation, because we cannot make predictions about the mixing weight changes and therefore are unable to separate the state-dependent and the state-independent part of the approximation. That is not only a problem for LCS but for any non-linear function approximation architecture.

An additional effect of the non-separation of approximation parameters and state-dependent values is that for TD(0) we have to minimise Eq. (14), using a previous approximation $\tilde{V}_m(i_{m+1})$ for the expected return of state i_m rather than projecting it onto the current approximation $\tilde{V}_t(i_{m+1})$, which would be to minimise

$$\sum_{m=0}^{t} I_{S_k}(i_m) \left(r^\mu_{i_m i_{m+1}} + \gamma \tilde{V}_t(i_{m+1}) - \tilde{V}_t(i_m) \right)^2 .$$

For a linear architecture that gives

$$\sum_{m=0}^{t} I_{S_k}(i_m) \left(r^\mu_{i_m i_{m+1}} + w_{t+1}' \left(\gamma \phi(i_{m+1}) - \phi(i_m) \right) \right)^2 ,$$

which allows separation of state-dependent and state-independent values. Hence, with linear architectures we can minimise the difference between the current approximation and the expected return, given the current approximation, for all states ever visited. For non-linear architectures we are forced to accept that we cannot project expected returns for past states onto the current value approximation and are therefore bound to use the past approximations for minimisation which results in a slower rate of convergence.

4.4 Convergence Investigations

Convergence properties of a system give important information about its long-term behaviour. Even though it is usually impossible for stochastic systems to give convergence guarantees within finite time, even convergence after an infinite number of steps can tell us how the solution evolves over a finite number of time steps.

In this section we will investigate the behaviour of the LCS policy evaluation update

$$\tilde{V}_{k,t+1} = \Pi_{D_k} T_\mu \tilde{V}_t.$$

This is the first step to investigating the properties of TD(0) and SARSA(0) with LCS function approximation, as both performs a step-wise approximation of this iteration.

For the sake of this discussion, let us assume that the mixing weights are time-invariant, given by Ψ_k for classifier k and all t. By using the definition of T_μ and the overall approximation $\tilde{V}$, we can reformulate above iteration as

$$\begin{aligned} \tilde{V}_{t+1} &= \sum_{k=1}^{K} \Psi_k \Pi_{D_k} T_\mu \tilde{V}_t \\ &= \sum_{k=1}^{K} \Psi_k \Pi_{D_k} r^\mu + \gamma \sum_{k=1}^{K} \Psi_k \Pi_{D_k} P^\mu \tilde{V}_t \end{aligned}$$

We can see that this is a matrix iteration of the form

$$\tilde{V}_{t+1} = A\tilde{V}_t + b, \tag{15}$$

where A and b are given by

$$\begin{aligned} A &= \gamma \sum_{k=1}^{K} \Psi_k \Pi_{D_k} \Pi^\mu \\ b &= \sum_{k=1}^{K} \Psi_k \Pi_{D_k} r^\mu. \end{aligned}$$

We will use this observation in later to determine the convergence of this iteration.

4.4.1 Optimal Approximation

Let us give a short overview of how the iteration comes about: We assume that we have a model of the problem and therefore know the transition probabilities p_{ij}^μ and expected rewards r_{ij}^μ. Hence, the expected return for state i based on the overall value function $\tilde{V}_t$ is given by

$$\sum_{j \in S} p_{ij}^\mu \left(r_{ij}^\mu + \gamma \tilde{V}_t(j) \right) = \left(r^\mu + \gamma P^\mu \tilde{V}_t \right)(i).$$

Classifier k aims at minimising the distribution-weighted difference between expected returns and their approximation for all states that it matches, which is to minimise

$$\sum_{i \in S_k} \pi(i) \left(\sum_{j \in S} p_{ij}^\mu \left(r_{ij}^\mu + \gamma \tilde{V}_t(j) \right) - w'_{k,t+1} \phi(i) \right)^2,$$

which is equivalent to

$$\| r^\mu + \gamma P^\mu \tilde{V}_t - \Phi w_{k,t+1} \|_{D_k}^2.$$

Minimising that w.r.t. $w_{k,t+1}$ gives the condition

$$\left(\sum_{i \in S_k} \pi(i) \phi(i) \phi(i)' \right) w_{k,t+1} = \sum_{i \in S_k} \pi(i) \phi(i) \sum_{j \in S_k} p_{ij}^\mu \left(r_{ij}^\mu + \gamma \tilde{V}_t(j) \right),$$

which, in matrix notation, is

$$(\Phi' D_k \Phi)\, w_{k,t+1} = \Phi' D_k \left(r^\mu + \gamma P^\mu \tilde{V}_t \right).$$

Pre-multiplying by $\Phi \left(\Phi' D_k \Phi \right)^{-1}$, and using Eq. (5), results in

$$\Phi w_{k,t+1} = \Phi \left(\Phi' D_k \Phi \right)^{-1} \Phi' D_k \left(r^\mu + \gamma P^\mu \tilde{V}_t \right) = \Pi_{D_k} \left(r^\mu + \gamma P^\mu \tilde{V}_t \right) = \Pi_{D_k} T_\mu \tilde{V}_t.$$

That demonstrates that the iteration $\tilde{V}_{k,t+1} = \Pi_{D_k} T_\mu \tilde{V}_t$ does indeed give the optimal approximation for classifier k.

4.4.2 Contraction of T_μ

As we are interested in the effects of the operator conjunction $\Pi_{D_k} T_\mu$, let us first investigate the effects of T_μ. We can use the equivalence of T_μ and $T_\mu^{(0)}$ and the knowledge that $T_\mu^{(\lambda)}$ performs a contraction w.r.t. $\|\cdot\|_D$ [49], where D is the steady-state distribution for policy μ, to see that T_μ gives a contraction to the same norm. As this is an essential property of T_μ, we will give a short derivation.

For that derivation we will use Lemma 2.1 from [5], that states:

Lemma 4.1. *For all $z \in \mathbb{C}^N$, we have $\|P^\mu z\|_D \leq \|z\|_D$.*

That allows us to show the contraction mapping of T_μ:

Lemma 4.2. *For all $V, \bar{V} \in \mathbb{C}^N$, we have*

$$\|T_\mu V - T_\mu \bar{V}\|_D \leq \gamma \|V - \bar{V}\|.$$

Proof. We will use Lemma 4.1, the definition of T_μ, and the fact that $\gamma \geq 0$ to show that

$$\begin{aligned} \|T_\mu V - T_\mu \bar{V}\|_D &= \|r^\mu + \gamma P^\mu V - r^\mu - \gamma P^\mu \bar{V}\|_D \\ &= \|\gamma P^\mu (V - \bar{V})\|_D \\ &= \gamma \|P^\mu (V - \bar{V})\|_D \\ &\leq \gamma \|V - \bar{V}\|_D. \end{aligned}$$

□

The dependency of the contraction of T_μ on the steady-state distribution of the transition matrix P^μ is introduced by the relation of the expected return to the next state, which is determined by that matrix.

4.4.3 Approximation Properties

Having clarified the contraction of T_μ, we will now show the non-expansion of Π_{D_k} for any $|S_k| > 1$:

Lemma 4.3. *For all $V, \bar{V} \in \mathbb{C}^N$, we have*

$$\|\Pi_{D_k} V - \Pi_{D_k} \bar{V}\|_D \leq \|V - \bar{V}\|_{D_k} \leq \|V - \bar{V}\|_D \leq \|V - \bar{V}\|.$$

Proof. It is well known that for an orthogonal projection matrix Π, $\|\Pi\| \leq 1$. Additionally, for all $z \in \mathbb{C}^N$, the weighted matrix norm can be rewritten as $\|z\|_D = \|\sqrt{D} z\|$. Furthermore, by the definition of the projection matrix Π_{D_k} (Eq. (5)), its hermitian property, and the fact that $(I_{S_k})^a = I_{S_k}$ for all $a \in \mathbb{R}_{\neq 0}$, we have

$$\sqrt{D}\Pi_{D_k} = \sqrt{D}\Phi(\Phi' D_k \Phi)\Phi' D_k = \sqrt{I_{S_k} D}\Phi(\Phi' D_k \Phi)\Phi' \sqrt{D_k}\sqrt{D_k} = \Pi_{D_k}\sqrt{D_k}.$$

Overall, that gives

$$\begin{aligned} \|\Pi_{D_k} V - \Pi_{D_k} \bar{V}\|_D &= \|\sqrt{D}\Pi_{D_k}(V - \bar{V})\| \\ &= \|\Pi_{D_k}\sqrt{D_k}(V - \bar{V})\| \\ &\leq \|\Pi_{D_k}\| \|\sqrt{D_k}(V - \bar{V})\| \\ &\leq \|V - \bar{V}\|_{D_k}. \end{aligned}$$

The second and third inequality stem from the observation that the norm of a diagonal matrix is equal to its largest element along the diagonal, which implies $\|I_{S_k}\| = 1$ for any $|S_k| > 0$, and $\|\sqrt{D}\| \leq 1$ as all diagonal elements are positive and smaller than 1. From that follows that for all $z \in \mathbb{C}^N$,

$$\|z\|_{D_k} = \|I_{S_k}\sqrt{D} z\| \leq \|I_{S_k}\| \|z\|_D = \|z\|_D,$$

and

$$\|z\|_D = \|\sqrt{D} z\| \leq \|\sqrt{D}\| \|z\| \leq \|z\|,$$

which completes the proof. □

4.4.4 Single Classifier Approximation

Let us assume that we have one single classifier k, and that this classifier matches all states of the state space, that is $S_k = S$. Then we have an approximation architecture equivalent to a linear architecture, and the overall approximation is equivalent to the classifier's approximation, i.e. $\tilde{V}_t = \tilde{V}_{k,t}$. Consequently, we can reduce the LCS policy evaluation to

$$\tilde{V}_{k,t+1} = \Pi_D T_\mu \tilde{V}_{k,t}, \tag{16}$$

of which we can prove convergence, given the following theorem (see, for example, [28]):

Theorem 4.4 (Contraction Mapping). *Let S_f be a complete vector space with norm $\|\cdot\|$. Suppose f is a contraction mapping on S_f with contraction factor α. Then f has exactly one fixed point x^* in S_f. For any initial point x_0 in S_f, the sequence $x_0, f(x_0), f(f(x_0)), \dots$ converges to x^*; the rate of convergence of the above sequence in the norm $\|\cdot\|$ is at least α.*

Then, together with our knowledge of the properties of T_μ and Π_{D_k}, we can state the following:

Theorem 4.5. *Given a single Classifier k, with $S_k = S$, then for all initial $\tilde{V}_{k,-1} \in \mathbb{R}^N$, the iteration given by Eq. (16) converges to the unique fixed point of that iteration, given by*

$$\tilde{V}_k^\mu = (I - \gamma \Pi_D P^\mu)^{-1} \Pi_D r^\mu$$

Proof. Applying Lemma 4.2 and 4.3, we can show for the operator conjunction $\Pi_{D_k} T_\mu$ and any two $V, \bar{V} \in \mathbb{R}^N$:

$$\begin{aligned} \|\Pi_{D_k} T_\mu V - \Pi_{D_k} T_\mu \bar{V}\|_D &= \|\Pi_{D_k} T_\mu (V - \bar{V})\|_D \\ &\leq \|T_\mu (V - \bar{V})\|_D \\ &\leq \gamma \|V - \bar{V}\|. \end{aligned}$$

Hence, $\Pi_{D_k} T_\mu$ describes a contraction mapping in the inner product space defined by $< \cdot, D \cdot >$ with contraction factor γ. Thus, Theorem 4.4 applies and the sequence $\tilde{V}_{k,-1}, \tilde{V}_{k,0}, \tilde{V}_{k,1}, \dots$ converges to the unique fixed point of the iteration. The fixed point is derived by using $D_k = D$ due to $I_{S_k} = I$, and the definition of T_μ:

$$\tilde{V}_k^\mu = \Pi_D r^\mu + \gamma \Pi_D P^\mu \tilde{V}_k^\mu,$$

giving

$$\Pi_D r^\mu = \tilde{V}_k^\mu - \gamma \Pi_D P^\mu \tilde{V}_k^\mu = (I - \gamma \Pi_D P^\mu) \tilde{V}_k^\mu.$$

□

That result is already well known in reinforcement learning, and was first derived in [49]. Naturally, having a single classifier never applies to LCS, but the theorem shows how to combine approximation and DP update.

4.4.5 Special Classifier Arrangements

For an arbitrary number of classifiers K, let us consider the case for which each classifier k has a constant mixing weight ψ_k over all its matching states, giving its mixing matrix $\Psi_k = \psi_k I$. Naturally, the condition $\sum_{k=1}^K \Psi_k = I$ has to hold to ensure averaged mixing over all matching classifier. A special case of such a classifier arrangement is to have a disjoint set of classifiers, that is $\sum_{k=1}^K I_{S_k} = I$, with $\psi_k = 1$ for all classifiers.

Even though that setting of classifiers is very artificial, it is currently the only combination of classifiers that we know to form a non-expansion. That lets us state the following result:

Theorem 4.6. *Given a set of K classifiers, each with a mixing matrix $\Psi_k = \psi_k I_{S_k}$, where ψ_k is a constant that satisfies $0 \leq \psi_k \leq 1$ and $\sum_{k=1}^K \Psi_k = I$, the iteration*

$$\tilde{V}_{t+1} = \sum_{k=1}^K \Psi_k \Pi_{D_k} T_\mu \tilde{V}_t$$

converges to the unique fixed point

$$\tilde{V}^\mu = \left(I - \gamma \sum_{k=1}^K \Psi_k \Pi_{D_k} P^\mu \right)^{-1} \sum_{k=1}^K \Psi_k \Pi_{D_k} r^\mu$$

Proof. We will first show that the mixed projection for all classifiers is a non-expansion on $\| \cdot \|_D$, which is satisfied if for all $z \in \mathbb{R}^N$, $\| \sum_{k=1}^K \Psi_k \Pi_{D_k} V \|_D \leq \|V\|_D$:

$$\begin{aligned}
\| \sum_{k=1}^K \Psi_k \Pi_{D_k} V \|_D^2 &= \sum_{i \in S} \pi(i) \left(\sum_{k=1}^K I_{S_k}(i) \psi_k (\Pi_{D_k} V)(i) \right)^2 \\
&\leq \sum_{i \in S} \pi(i) \sum_{k=1}^K I_{S_k}(i) \psi_k (\Pi_{D_k} V)(i)^2 \\
&= \sum_{i \in S} \pi(i) \sum_{k=1}^K \psi_k (\Pi_{D_k} V)(i)^2 \\
&= \sum_{k=1}^K \psi_k \sum_{i \in S} \pi(i) (\Pi_{D_k} V)(i)^2 \\
&\leq \sum_{k=1}^K \psi_k \sum_{i \in S} \pi(i) I_{S_k}(i) V(i)^2 \\
&= \sum_{i \in S} \pi(i) V(i)^2 \sum_{k=1}^K \psi_k I_{S_k}(i) \\
&= \sum_{i \in S} \pi(i) V(i)^2 \\
&= \|V\|_D^2.
\end{aligned}$$

The first inequality is due to Jensen's Inequality. The following equality uses $I_{S_k} \Pi_{D_k} = \Pi_{D_k}$, and the second inequality is based on $\|\Pi_{D_k} V\|_D \leq \|V\|_{D_k}$, as given by Lemma 4.3. The equality after that is based on our initial assumption that $\sum_{k=1}^K \Psi_k = I$.

Above non-expansion in combination with Lemma 4.2 lets us derive for all $V, \bar{V} \in \mathbb{R}^N$:

$$\begin{aligned}
\| \sum_{k=1}^K \Psi_k \Pi_{D_k} T^\mu V - \sum_{k=1}^K \Psi_k \Pi_{D_k} T^\mu \bar{V} \|_D &= \| \sum_{k=1}^K \Psi_k \Pi_{D_k} T^\mu (V - \bar{V}) \|_D \\
&\leq \|T^\mu (V - \bar{V})\|_D \\
&\leq \gamma \|V - \bar{V}\|_D.
\end{aligned}$$

Hence, the iteration describes a contraction mapping on the inner product space $< \cdot, D \cdot >$, and Theorem 4.4 applies, proving convergence to the unique fixed point of the iteration.

By using the definition of T^μ, we can write

$$\tilde{V}^\mu = \sum_{k=1}^K \Psi_k \Pi_{D_k} r^\mu + \gamma \sum_{k=1}^K \Psi_k \Pi_{D_k} P^\mu \tilde{V}^\mu,$$

from which the fixed point follows from solving above for $\tilde{V}^\mu$. □

4.4.6 Arbitrary Classifier Arrangements

Let us consider a simple problem which we will investigate: Let the transition matrix P^μ for our current policy μ be given by

$$P^\mu = \begin{pmatrix} \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \end{pmatrix},$$

which has a uniform steady state distribution $\pi(1) = \pi(2) = \pi(3) = \frac{1}{2}$, giving the diagonal distribution matrix $D = \frac{1}{2} I$. We will use two classifiers to approximate the value function, where the first matches all states, and the second only the first two, that is $S_1 = \{1, 2, 3\}$ and $S_2 = \{1, 2\}$. Their mixing is determined by the mixing parameter ψ, and the diagonal mixing matrices $\Psi_1 = \text{diag}(1-\psi, 1-\psi, 1)$ and $\Psi_2 = \text{diag}(\psi, \psi, 0)$. We will be using averaging classifiers, which gives the feature matrix $\Phi = (1, 1, 1)'$. For a value vector $(a, b, c)'$, that gives the overall approximation as:

$$\sum_{k=1}^{2} \Psi_k \Pi_k \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \frac{1}{6} \begin{pmatrix} (2+\psi)(a+b) + 2(1-\psi)c \\ (2+\psi)(a+b) + 2(1-\psi)c \\ 2(a+b+c) \end{pmatrix}.$$

As we can see, classifier 1 averages over all states, and classifier 2 averages over the first two states. Hence, setting $\psi = 1$ will assign the first two states of the overall approximation the values of classifier 2, whereas $\psi = 0$ gives all states the values of classifier 1.

Let us now consider the value function $V = (2, 2, 1)'$, and approximations $\tilde{V}^{\psi=0} = (\frac{5}{3}, \frac{5}{3}, \frac{5}{3})'$ and $\tilde{V}^{\psi=1} = (2, 2, \frac{5}{3})'$, and their norms

$$\|V\|_D = \sqrt{\frac{81}{9}}, \qquad \|\tilde{V}^{\psi=0}\|_D = \sqrt{\frac{75}{9}} < \|V\|_D, \qquad \|\tilde{V}^{\psi=1}\|_D = \sqrt{\frac{97}{9}} > \|V\|_D.$$

Those values can be seen as the result of $\|\Pi V - \Pi\bar{V}\|_D$, where Π is the overall approximation, and $\bar{V} = (0, 0, 0)'$ is the null vector with its approximation $\Pi\bar{V} = (0, 0, 0)'$. Hence, given that $\psi = 0$, the overall approximation forms a contraction. However, $\psi = 1$ features a lower approximation error $\|V - \tilde{V}^{\psi=1}\|_D$ and performs an expansion. That demonstrates that even with a fixed mixing weight the overall approximation is not necessarily a non-expansion. Hence, we cannot guarantee that this approximation in combination with the DP update will converge.

An alternative approach to answering the question of convergence is to consider the LCS policy evaluation iteration as a matrix iteration of the form of Eq. (15). As we have already discussed in Section 3.2.2, this iteration converges if and only if the matrix A has a spectral radius of $\rho(A) < 1$. In the above example, A is given by

$$A = \gamma \sum_{k=1}^{2} \Psi_k \Pi_k P^{\mu} = \frac{\gamma}{3} \begin{pmatrix} \frac{1}{2}(\psi+2) & \frac{1}{4}(\psi-4) & \frac{1}{4}(\psi-4) \\ \frac{1}{2}(\psi+2) & \frac{1}{4}(\psi-4) & \frac{1}{4}(\psi-4) \\ 1 & 1 & 1 \end{pmatrix},$$

which has a spectrum of $\sigma(A) = \{0, \gamma, \frac{\gamma\psi}{12}\}$. Hence, $\rho(A) < 1$, and the iteration converges. That shows that the requirement of having an approximation that forms a non-expansion is sufficient for convergence, but not necessary. In the case of LCS that requirement is not always fulfilled, and therefore we need to concentrate on studying the eigenvalues of the matrix A. So far, we can give neither positive nor negative results on their investigation.

To relate matrix iterations to contraction mappings, we will give one final result which shows that TD(0) with any non-expanding approximation on $\|\cdot\|_D$ results in a converging matrix iteration, which is given if the matrix A has $\rho(A) < 1$:

Theorem 4.7. *Let $\Pi : \mathbb{C}^N \to \mathbb{C}^N$ be a non-expansion on $\|\cdot\|_D$, that is for all $V, \bar{V} \in \mathbb{C}^N$,*

$$\|\Pi V - \Pi\bar{V}\|_D \leq \|V - \bar{V}\|_D.$$

Then the $N \times N$ matrix A given by

$$A = \gamma \Pi P^{\mu}$$

has eigenvalues within a circle of radius γ, that is $\rho(A) \leq \gamma$.

Proof. Let $\beta \in \mathbb{R}$ be an eigenvalue of A, and $z \in \mathbb{C}^N$ its corresponding eigenvector, that is

$$\gamma \Pi P^{\mu} z = \beta z.$$

Taking the weighted norm w.r.t. D gives

$$|\gamma| \|\Pi P^{\mu} z\|_D = |\beta| \|z\|_D.$$

Using the non-expansion of Π and Lemma 4.1 lets us derive for the left-hand side

$$|\gamma| \|\Pi P^{\mu} z\|_D \leq |\gamma| \|P^{\mu} z\|_D \leq |\gamma| \|z\|_D.$$

Comparing that to the right-hand sides lets us conclude that $|\beta| \leq |\gamma|$. Hence, every eigenvalue of A is within a circle of radius γ. □

That confirms Theorem 4.5 and 4.6, as the approximation architecture of both theorems describe a non-expansion that meets to the requirements of the last theorem.

5 Summary and Conclusion

We have introduced a framework for LCS that allows studying reinforcement learning, function approximation and their interaction. Furthermore, we have demonstrated its use by deriving both model-based and model-free reinforcement learning methods with LCS function approximation from first principles, and have elaborated on possible implementations of the use of Q-Learning in LCS. One of the two presented implementations is novel and is expected to surpass the performance of current LCS function approximation algorithms.

In more detail, we have derived how we can perform model-based Value Iteration with LCS, and how this can be approximated by a step-wise update. A further approximation led us straight to Q-Learning, for which we have shown that the Least Mean Square algorithm on Q-Learning gives the algorithm that is currently used in XCS. Based on our derivation we have analysed recent attempts and arguments about XCS with gradient descent, and have emphasised the independence of classifiers in performing the value function approximation. Based on our previous work on the function approximation in LCS [22], we have also presented an algorithm based on the Kalman filter that performs Q-Learning with the LCS function approximation architecture and accurately tracks the optimal approximation while simultaneously keeping track of the approximation error of a classifier more accurately than all current implementations. With respect to the optimal approximation, we have argued that the non-linearity of the LCS approximation architecture makes it impossible to solve the Bellman Equation directly, but have introduced two possible iteration that should lead to that optimal approximation.

Regarding Policy Iteration, we have discussed how we can use LCS for the policy evaluation step. We again discussed both the model-based and the model-free case, but have omitted the description of possible implementations due to the similarity in derivations. Regarding TD(λ), we have shown how the non-linear architecture does not allow the same efficient implementation of TD(0) as a linear approximation architecture, and how there is no known accurate implementation of TD(λ), and possibly never will be.

As the framework adapts concepts from reinforcement learning to LCS, it should make LCS more accessible to researchers of reinforcement learning, and vice versa. For that purpose, we have derived both the reinforcement learning methods and the LCS methods from first principles, using comparable derivations. As demonstrated in the previous section, theoretical investigations on the stability of LCS can now partially be answered by the using similar methods to the ones that are used in reinforcement learning.

We have demonstrated the contraction mapping of the T_μ operator and the non-expansion of the approximation of a single classifier. Both in combination gives the contraction of policy evaluation with a single classifier, and therefore its convergence to a fixed point of the update. We have also shown how a particular arrangement of classifiers, including any disjoint set of classifiers, describes a contraction mapping. In a simple example we have shown that not all combinations of classifiers form a contraction mapping, but can still converge. That convergence was established by showing that the matrix iteration that describes the LCS policy evaluation satisfies the necessary condition for convergence.

For the use of LCS for Value Iteration (including its approximations, like Q-Learning), it is know that linear approximation architectures might diverge. However, it might still be possible to show their convergence in combination with averaging classifiers, as originally used in XCS. What needs to be demonstrated is that all classifiers in combination form an averager, as defined in [23], which is quite likely, as discussed in Section 4.2.1. Once this is achieved, we additionally need to show that Q-Learning in LCS performs an approximation to Value Iteration, in which the approximation error converges to zero with time.

To clarify the theoretical properties of using a linear approximation architecture in policy evaluation, we need to analyse the matrix iteration as already outlined at the end of the previous section. Even if that matrix iteration is known to converge, it only concerns the case of fixed mixing weights. Changing the mixing weights results in a time-variance of the matrix iteration which might be captured by observing the joint spectral radius of the iteration matrix sequence. If that is shown converge, the work of Konda and Tsitsiklis [29] might give hints on how to study LCS policy evaluation when used in Optimistic Policy Iteration.

Note that all of the above only concerns LCS with a time-invariant population. How to include the replacement of classifiers is topic of further work on our framework. Given that LCS converges with a time-invariant population, we can assume that modifying the population of classifiers changes the fixed point of the update. Hence, having a convergent classifier replacement makes convergence of the whole LCS very likely. However, there is still a lot of work ahead of us before we can give definite statements.

References

[1] Alwyn Barry. Limits in long path learning with XCS. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1832–1843. Springer-Verlag, 2003.

[2] Alwyn Barry, John Holmes, and Xavier Llora. Data Mining using Learning Classifier Systems. In Larry Bull, editor, *Foundations of Learning Classifier Systems*, Berlin, 2004. Springer Verlag.

[3] Alwyn M. Barry. The stability of long action chains in XCS. *Journal of Soft Computing*, 6(3–4):183–199, 2002.

[4] Ester Bernadó, Xavier Llorà, and Josep M. Garrell. XCS and GALE: a Comparative Study of Two Learning Classifier Systems with Six Other Learning Algorithms on Classification Tasks. In *Proceedings of the 4th International Workshop on Learning Classifier Systems (IWLCS-2001)*, pages 337–341, 2001.

[5] Dimitri P. Bertsekas, Vivek S. Borkas, and Angelia Nedić. Improved Temporal Difference Methods with Linear Function Approximation. In Jennie Si, Andrew G. Barto, Warren Buckler Powell, and Don Wunsch, editors, *Handbook of Learning and Approximate Dynamic Programming*, chapter 9, pages 235–260. Wiley Publishers, August 2004.

[6] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[7] H.-G. Beyer, U.-M. O'Reilly, D.V. Arnold, W. Banzhaf, C. Blum, E.W. Bonabeau, E. Cant Paz, D. Dasgupta, K. Deb, J.A. Foste r, E.D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G.R. Raidl, T. Soule, A. Tyrrell, J.-P. Watson, and E. Zitzler, editors. *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2005*, volume 2, New York, 2005. ACM Press.

[8] Lashon B. Booker. Approximating value function in classifier systems. In Bull and Kovacs [15].

[9] Justin A. Boyan. Least-Squares Temporal Difference Learning. In *Proceedings of the 16th International Conference on Machine Learning*, pages 49–56, San Francisco, CA, USA, 1999. Morgan Kaufmann.

[10] Justin A. Boyan. Technical Update: Least-Squares Temporal Difference Learning. *Machine Learning*, 49(2-3):233–246, 2002.

[11] Justin A. Boyan and Andrew W. Moore. Generalization in Reinforcement Learning: Safely Approximating the Value Function. *Advances in Neural Information Processing Systems*, 7, 1995.

[12] Steven J. Bradtke. Reinforcement Learning Applied to Linear Quadratic Regulation. In *Advances in Neural Information Processing Systems*, volume 5. Morgan Kaufmann Publishers, 1993.

[13] Steven J. Bradtke and Andrew G. Barto. Linear Least-Squares Algorithms for Temporal Difference Learning. *Machine Learning*, 22(1–3):33–57, 1996.

[14] Larry Bull. On accuracy-based fitness. *Journal of Soft Computing*, 6(3–4):154–161, 2002.

[15] Larry Bull and Tim Kovacs, editors. *Foundations of Learning Classifier Systems*, volume 183 of *Studies in Fuzziness and Soft Computing.* Springer Verlag, Berlin, 2005.

[16] Martin Butz, Tim Kovacs, Pier Luca Lanzi, and Stewart W. Wilson. Toward a theory of generalization and learning in XCS. *IEEE Transactions on Evolutionary Computation*, 2004.

[17] Martin V. Butz, David E. Goldberg, and Pier Luca Lanzi. Gradient Descent Methods in Learning Classifier Systems: Improving XCS Performance in Multistep Problems. Technical Report 2003028, Illinois Genetic Algorithms Laboratory, December 2003.

[18] Martin V. Butz, David E. Goldberg, and Pier Luca Lanzi. Gradient Descent Methods in Learning Classifier Systems: Improving XCS Performance in Multistep Problems. *IEEE Transactions on Evolutionary Computation*, 9(5):452–473, October 2005.

[19] Phillip William Dixon, David W. Corne, and Martin John Oates. A preliminary investigation of modified XCS as a generic data mining tool. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Advances in Learning Classifier Systems*, volume 2321 of *LNAI*, pages 133–150. Springer-Verlag, Berlin, 2002.

[20] Marco Dorigo and Hugues Bersini. A Comparison of Q-Learning and Classifier Systems. In Dave Cliff, Philip Husbands, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *From Animals to Animats 3. Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB94)*, pages 248–255. A Bradford Book. MIT Press, 1994.

[21] Jan Drugowitsch and Alwyn M. Barry. XCS with Eligibility Traces. In Beyer et al. [7], pages 1851–1858.

[22] Jan Drugowitsch and Alwyn M. Barry. A Formal Framework and Extensions for Function Approximation in Learning Classifier Systems. Technical Report CSBU2006-01, Dept. Computer Science, University of Bath, January 2006. ISSN 1740-9497.

[23] Geoffrey J. Gordon. Stable Function Approximation in Dynamic Programming. In Armand Prieditis and Stuart Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261–268, San Francisco, CA, USA, 1995. Morgan Kaufmann.

[24] Andrew Greenyer. The use of a learning classifier system JXCS. In P. van der Putten and M. van Someren, editors, *CoIL Challenge 2000: The Insurance Company Case*. Leiden Institute of Advanced Computer Science, June 2000. Technical report 2000-09.

[25] Leemon C. Baird III. Residual Algorithms: Reinforcement Learning with Function Approximation. In *International Conference on Machine Learning*, pages 30–37, 1995.

[26] Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh. On the Convergence of Stochastic Iterative Dynamic Programming Algorithms. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 703–710. Morgan Kaufmann Publishers, 1994.

[27] Daphne Koller and Ronald Parr. Policy Iteration for Factored MDPs. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 326–334, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers.

[28] A. N. Kolmogorov and S. V. Fomin. *Introductory Real Analysis*. Prentice Hall, 1970. Revised English edition translated and edited by A. N. Silverman.

[29] Vijay R. Konda and John N. Tsitsiklis. On actor-critic algorithms. *SIAM Journal on Control and Optimization*, 42(4):1143–1166, 2003.

[30] Tim Kovacs. *A Comparison and Strength and Accuracy-based Fitness in Learning Classifier Systems*. PhD thesis, University of Birmingham, 2002.

[31] Michail G. Lagoudakis and Ronald Parr. Least-squares policy iteration. *Journal of Machine Learning Research*, 4:1107–1149, 2003.

[32] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Generalization in the XCSF Classifier Systems: Analysis, Improvement, and Extenstion. Technical Report 2005012, Illinois Genetic Algorithms Laboratory, March 2005.

[33] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. XCS with Computed Predictions in Multistep Environments. In Beyer et al. [7], pages 1859–1866.

[34] Autor Merke and Ralf Schoknecht. Convergence of Synchronous Reinforcement Learning with Linear Function Approximation. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 75, New York, NY, USA, 2004. ACM Press.

[35] David E. Moriarty, Alan C. Schultz, and John J. Grefenstette. Evolutionary Algorithms for Reinforcement Learning. *Journal of Artificial Intelligence Research*, 11:199–229, 1999. http://www.ib3.gmu.edu/gref/papers/moriarty-jair99.html.

[36] Remi Munos. Error Bounds for Approximate Policy Iteration. In *19th International Conference on Machine Learning*, pages 560–567, 2003.

[37] Angelia Nedić and D. P. Bertsekas. Least Squares Policy Evaluation Algorithms with Linear Function Approximation. *Discrete Event Dynamic Systems*, 13(1-2):79–110, 2003.

[38] Dirk Ormoneit and Saunak Sen. Kernel-Based Reinforcement Learning. *Machine Learning*, 49(2-3):161–178, 2002.

[39] Gavin Rummery and Mahesan Niranja. On-line Q-Learning using Connectionist Systems. Technical Report 166, Engineering Department, University of Cambridge, 1994.

[40] Shaun Saxon and Alwyn Barry. XCS and the Monk's Problems. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Learning Classifier Systems. From Foundations to Applications*, volume 1813 of *LNAI*, pages 223–242, Berlin, 2000. Springer-Verlag.

[41] Ralf Schoknecht. Optimality of Reinforcement Learning Algorithms with Linear Function Approximation. In *Proceedings of the 15th Neural Information Processing Systems conference*, pages 1555–1562, 2002.

[42] Ralf Schoknecht and Artur Merke. Convergent Combinations of Reinforcement Learning with Linear Function Approximation. In *Proceedings of the 15th Neural Information Processing Systems conference*, pages 1579–1586, 2002.

[43] Ralf Schoknecht and Artur Merke. TD(0) Converges Provably Faster than the Residual Gradient Algorithm. In *ICML '03: Proceedings of the twentieth international conference on Machine Learning*, pages 680–687, 2003.

[44] Satinder Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvari. Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. *Machine Learning*, 39:287–308, 2000.

[45] Richard S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 1988.

[46] Richard S. Sutton. Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044, Cambridge, MA, USA, 1996. MIT Press.

[47] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998. A Bradford Book.

[48] John Tsitsiklis and Benjamin Van Roy. Feature-Based Methods for Large Scale Dynamic Programming. *Machine Learning*, 22:59–94, 1996.

[49] John Tsitsiklis and Benjamin Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

[50] John N. Tsitsiklis. On the Convergence of Optimistic Policy Iteration. *Journal of Machine Learning Research*, 3:59–72, 2003.

[51] Atsushi Wada, Keiki Takadama, Katsunori Shimohara, and Osamu Katai. Is Gradient Descent Method Effective for XCS? Analysis of Reinforcement Process in XCSG? In Wolfgang Stolzmann et al., editor, *Proceedings of the Seventh International Workshop on Learning Classifier Systems, 2004*, LNAI, Seattle, WA, June 2004. Springer Verlag.

[52] Atsushi Wada, Keiki Takadama, Katsunori Shimohara, and Osamu Katai. Learning Classifier System with Convergence and Generalisation. In Bull and Kovacs [15].

[53] Christopher J.C.H. Watkins. *Learning from delayed rewards.* PhD thesis, University of Cambridge, Psychology Department, 1989.

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

[55] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *IRE WESCON Convention Revord Part IV*, pages 96–104, 1960.

[56] Stewart W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994. http://prediction-dynamics.com/.

[57] Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[58] Stewart W. Wilson. Function Approximation with a Classifier System. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 974–981. Morgan Kaufmann, 2001.

[59] Stewart W. Wilson. Classifiers that Approximate Functions. *Neural Computing*, 1(2-3):211–234, 2002.