The Divergence of Reinforcement Learning Algorithms with Value-Iteration and Function Approximation

Michael Fairbank, *Student Member, IEEE* and Eduardo Alonso

Cite as: Michael Fairbank and Eduardo Alonso, *The Divergence of Reinforcement Learning Algorithms with Value-Iteration and Function Approximation*, In Proceedings of the IEEE International Joint Conference on Neural Networks, June 2012, Brisbane (IEEE IJCNN 2012), pp. 3070–3077.

Errata: See footnote 2, plus note in Fig.3

***Abstract*—This paper gives specific divergence examples of value-iteration for several major Reinforcement Learning and Adaptive Dynamic Programming algorithms, when using a function approximator for the value function. These divergence examples differ from previous divergence examples in the literature, in that they are applicable for a greedy policy, i.e. in a "value iteration" scenario. Perhaps surprisingly, with a greedy policy, it is also possible to get divergence for the algorithms TD(1) and Sarsa(1). In addition to these divergences, we also achieve divergence for the Adaptive Dynamic Programming algorithms HDP, DHP and GDHP.**

***Index Terms*—Adaptive Dynamic Programming, Reinforcement Learning, Greedy Policy, Value Iteration, Divergence**

I. Introduction

Adaptive Dynamic Programming (ADP) [1] and Reinforcement Learning (RL) [2] are similar fields of study that aim to make an agent learn actions that maximise a long-term reward function. These algorithms often rely on learning a "value function" that is defined in Bellman's Principle of Optimality [3]. When an algorithm attempts to learn this value function by a general smooth function approximator, while the agent is being controlled by a "greedy policy" on that approximated value function, then ensuring convergence of the learning algorithm is difficult.

It has so far been an open question as to whether divergence can occur under these conditions and for which algorithms. In this paper we present a simple artificial test problem which we use to make many RL and ADP algorithms diverge with a greedy policy. The value function learning algorithms that we consider are Sarsa(λ) [4], TD(λ) [5], and the ADP algorithms Heuristic Dynamic Programming (HDP), Dual Heuristic Dynamic Programming (DHP), Globalized Dual Heuristic Dynamic Programming (GDHP) [6], [7], [8] and Value-Gradient Learning (VGL(λ)) [9], [10]. We prove divergence of all of these algorithms (including VGL(0), VGL(1), Sarsa(0), Sarsa(1), TD(0), TD(1), DHP and GDHP), all when operating with greedy policies, i.e. in a "value-iteration" setting.

Some of these algorithms have convergence proofs when a *fixed* policy is used. For example TD(λ) is proven to converge when $\lambda = 1$ since it is then (and only then) true gradient descent on an error function [5]. Also for $0 \leq \lambda \leq 1$, it is proven to converge by [11] when the approximate value function is linear in its weight vector and learning is "on-policy". Recent advancements in the RL literature have extended convergence conditions of variants of TD(λ) to an "off-policy" setting [12], and with non-linear function approximation of the value function [13]. However, all these proofs apply to a fixed policy instead of the greedy policy situation we consider here.

[8] show that ADP processes will converge to optimal behaviour if the value function could be perfectly learned over all of state space at each iteration. However in reality we must work with a function approximator for the value function with finite capabilities, so this assumption is not valid. Working with a general quadratic function approximator, [14] proves the general instability of DHP and GDHP. This analysis was for a fixed policy, so with a greedy policy convergence would presumably seem even less likely. This paper confirms this.

A key insight into the difficulty of understanding convergence with a greedy policy is shown by lemma 7 of [9] that the dependency of a greedy action on the approximated value function is *primarily through the value-gradient*, i.e. the gradient of the value function with respect to the state vector. We use a value-gradient analysis in this paper to understand the divergence of *all* of the algorithms being tested. [9] and [15] recently defined a value-function learning algorithm that is proven to converge under certain smoothness conditions, using a greedy policy and an arbitrary smooth approximated value function, so this contrasts greatly to the diverging algorithm examples we give here.

In the rest of this introduction (sections I-A to I-E), we state the general RL/ADP problem and give the necessary function definitions. In section II we give definitions of the algorithms that we are testing.

The approach we make to achieve divergence is to define a problem that is simple enough to analyse algebraically, but flexible enough to provide a divergence example (sections III to III-B). We then analyse a trajectory for this problem (sections III-C to III-E), so that we can write the VGL(λ) weight update as a single dynamic system and hence examine what choice of parameters could be made to force this dynamic system to diverge (section IV). The VGL(λ) weight update

M. Fairbank and E. Alonso are with the Department of Computing, School of Informatics, City University London, London, UK (e-mail: michael.fairbank.1@city.ac.uk; E.Alonso@city.ac.uk).

is easier to analyse than the TD(λ) one, since as mentioned above the greedy policy depends on the value-gradient, so in section V we just use the same learning parameters that caused divergence for VGL(λ) and find empirically that they cause the other algorithms to diverge too.

Finally, in section VI, we discuss the difficulty of ensuring value-iteration convergence but its potential advantages compared to policy-iteration.

A. RL and ADP Problem Definition and Notation

The typical RL/ADP scenario is an agent wandering around in an environment (with state space $\mathbb{S} \subset \Re^n$), such that at time t it has state vector $\vec{x}_t \in \mathbb{S}$. At each time t the agent chooses an action $\vec{a}_t$ (from an action space $\mathbb{A} \subset \Re^n$) which takes it to the next state according to the environment's model function $\vec{x}_{t+1} = f(\vec{x}_t, \vec{a}_t)$, and gives it an immediate reward, r_t, given by the function $r_t = r(\vec{x}_t, \vec{a}_t)$. In general these model functions f and r can be stochastic functions. The agent keeps moving, forming a trajectory of states $(\vec{x}_0, \vec{x}_1, \ldots)$, which terminates if and when a designated terminal state is reached. In RL/ADP, we aim to find a *policy* function, $\pi(\vec{x})$, that calculates which action $\vec{a} = \pi(\vec{x})$ to take for any given state $\vec{x}$. The objective of RL/ADP is to find a policy such that the expectation of the total discounted reward, $\langle \sum_t \gamma^t r_t \rangle$, is maximised for any trajectory. Here $\gamma \in [0, 1]$ is a constant *discount factor* that specifies the importance of long term rewards over short term ones.

B. Approximate Value Function (Critic) and its Gradient

We define $\widetilde{V}(\vec{x}, \vec{w})$ to be the real-valued scalar output of a smooth function approximator with weight vector $\vec{w}$ and input vector $\vec{x}$. This is the "approximate value function", or "critic". We define $\widetilde{G}(\vec{x}, \vec{w})$ as the "approximate value gradient", or "critic gradient", to be $\widetilde{G}(\vec{x}, \vec{w}) \equiv \frac{\partial \widetilde{V}(\vec{x}, \vec{w})}{\partial \vec{x}}$.

Here and throughout this paper, a convention is used that all defined vector quantities are columns, whether they are coordinates, or derivatives with respect to coordinates. So, for example, $\widetilde{G}$, $\frac{\partial \widetilde{V}}{\partial \vec{x}}$ and $\frac{\partial \widetilde{V}}{\partial \vec{w}}$ are all columns.

C. Greedy Policy

The greedy policy is the function that always chooses actions as follows:

$$\vec{a} = \arg\max_{\vec{a} \in \mathbb{A}} (\widetilde{Q}(\vec{x}, \vec{a}, \vec{w})) \quad \forall \vec{x} \tag{1}$$

where we define the approximate Q Value function as

$$\widetilde{Q}(\vec{x}, \vec{a}, \vec{w}) = r(\vec{x}, \vec{a}) + \gamma \widetilde{V}(f(\vec{x}, \vec{a}), \vec{w}) \tag{2}$$

D. Actor-critic architectures

If a non-greedy policy is used, then a separate policy function would be used. This could be represented by a second function approximator, known as the actor (the first function approximator being the critic). The actor and the critic together are known as an actor-critic architecture.

Training of the actor and critic would take place iteratively and in alternating phases. Policy iteration is the situation where the critic is trained to completion in between every actor update. Value iteration is the situation where the actor is trained to completion in between each critic update.

The intention of the actor's training weight update is to make the actor behave more like a greedy policy. Hence value iteration is very much like using a greedy policy, since the *objective* of training an actor to completion is to make the actor behave just like a greedy policy. Hence the divergence results we derive in this paper for a greedy policy are applicable to an actor-critic architecture with value-iteration, assuming the function approximator of the actor has sufficient flexibility to learn the greedy policy accurately enough (which is true for the actor we define in section III-B).

E. Trajectory Shorthand Notation

Throughout this paper, all subscripted indices are what we call trajectory shorthand notation. These refer to the time step of a trajectory and provide corresponding arguments $\vec{x}_t$ and $\vec{a}_t$ where appropriate; so that for example $\widetilde{V}_{t+1} \equiv \widetilde{V}(\vec{x}_{t+1}, \vec{w})$; $\widetilde{Q}_{t+1} \equiv \widetilde{Q}(\vec{x}_{t+1}, \vec{a}_{t+1}, \vec{w})$; $\left(\frac{\partial \widetilde{Q}}{\partial \vec{a}}\right)_t$ is shorthand for $\left.\frac{\partial \widetilde{Q}(\vec{x}, \vec{a}, \vec{w})}{\partial \vec{a}}\right|_{(\vec{x}_t, \vec{a}_t, \vec{w})}$ and $\left(\frac{\partial \widetilde{V}}{\partial \vec{w}}\right)_t$ is shorthand for $\left.\frac{\partial \widetilde{V}(\vec{x}, \vec{w})}{\partial \vec{w}}\right|_{(\vec{x}_t, \vec{w})}$.

II. Learning Algorithms and Definitions

A. TD(λ) Learning

The TD(λ) algorithm [5] can be defined in batch mode by the following weight update applied to an entire trajectory:

$$\Delta \vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{V}}{\partial \vec{w}}\right)_t ({R^\lambda}_t - \widetilde{V}_t) \tag{3}$$

where $\lambda \in [0, 1]$, and $\alpha > 0$ are fixed constants. R^λ is the (moving) target for this weight update. It is known as the "λ-Return", as defined by [16]. For a given trajectory, this can be written concisely using trajectory shorthand notation by the recursion

$${R^\lambda}_t = r_t + \gamma(\lambda {R^\lambda}_{t+1} + (1 - \lambda)\widetilde{V}_{t+1}) \tag{4}$$

with ${R^\lambda}_t = 0$ at any terminal state, as proven in Appendix A of [9]. This equation introduces the dependency on λ into eq. 3. Using the λ-Return enables us to write TD(λ) in this very concise way, known as the "forwards view of TD(λ)" by [2], however the traditional way to implement the algorithm is using "eligibility traces", as described by [5].

TD(λ) is defined for the task of *policy evaluation*, i.e. it is defined just for the task of learning the approximated value function for a *fixed* policy. It is not usually used with a greedy policy, which is the circumstance in which we consider it in this paper. However we show in section V-A that the TD(λ) weight update can be equivalent in some circumstances to the Sarsa(λ) weight update, which *is* defined for a greedy policy. Another reason to consider TD(λ) with a greedy policy is that TD(λ) can be used in an actor-critic architecture as part of a value-iteration scheme, which, as we described in section I-D, is very similar to using a greedy policy.

B. Sarsa(λ) Algorithm

Sarsa(λ) is an algorithm for control problems that learns to approximate the $\widetilde{Q}(\vec{x},\vec{a},\vec{w})$ function [4]. It is designed for policies that are dependent on the $\widetilde{Q}(\vec{x},\vec{a},\vec{w})$ function (e.g. the greedy policy or a greedy policy with added stochastic noise), where $Q(\vec{x},\vec{a},\vec{w})$ here is defined to be the output of a given function approximator.

The Sarsa(λ) algorithm is defined for trajectories where all actions after the first are found by the given policy; the first action $\vec{a}_0$ can be arbitrary. The function-approximator update is defined to be:

$$\Delta\vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{Q}}{\partial \vec{w}}\right)_t ({Q^\lambda}_t - \widetilde{Q}_t) \tag{5}$$

where Q^λ is the target for this weight update. This is analogous to the λ-return, but uses the function approximator $\widetilde{Q}$ in place of $\widetilde{V}$. We can define Q^λ recursively in trajectory shorthand notation by

$${Q^\lambda}_t = r_t + \gamma(\lambda {Q^\lambda}_{t+1} + (1-\lambda)\widetilde{Q}_{t+1}) \tag{6}$$

with ${Q^\lambda}_t = 0$ at any terminal state.

C. The VGL(λ) Algorithm

To define the VGL(λ) algorithm, throughout this paper we use a convention that differentiating a column vector function by a column vector causes the vector in the numerator to become transposed (becoming a row). For example $\frac{\partial f}{\partial \vec{x}}$ is a matrix with element (i,j) equal to $\frac{\partial f(\vec{x},\vec{a})^j}{\partial \vec{x}^i}$. Similarly, $\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)^{ij} = \frac{\partial \widetilde{G}^j}{\partial \vec{w}^i}$, and $\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t$ is this matrix evaluated at $(\vec{x}_t,\vec{w})$.

Using this notation and the implied matrix products, all VGL algorithms can be defined by a weight update of the form:

$$\Delta\vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t ({G'}_t - \widetilde{G}_t) \tag{7}$$

where α is a small positive constant; $\widetilde{G}_t$ is the approximate value gradient; and ${G'}_t$ is the "target value gradient" defined recursively by:

$${G'}_t = \left(\frac{Dr}{D\vec{x}}\right)_t + \gamma \left(\frac{Df}{D\vec{x}}\right)_t \left(\lambda {G'}_{t+1} + (1-\lambda)\widetilde{G}_{t+1}\right) \tag{8}$$

with ${G'}_t = \vec{0}$ at any terminal state; where Ω_t is an arbitrary positive definite matrix of dimension $(\dim\vec{x} \times \dim\vec{x})$; and where $\frac{D}{D\vec{x}}$ is shorthand for

$$\frac{D}{D\vec{x}} \equiv \frac{\partial}{\partial \vec{x}} + \frac{\partial \pi}{\partial \vec{x}}\frac{\partial}{\partial \vec{a}} \; ; \tag{9}$$

and where all of these derivatives are assumed to exist. Equations 7, 8 and 9 define the VGL(λ) algorithm. [9] and [10] give further details, and pseudocode for both on-line and batch-mode implementations.

The Ω_t matrix was introduced by Werbos for the algorithm GDHP (e.g. see [14, eq. 32]), and can be chosen freely by the experimenter, but it is in general difficult to decide how to do this; so for most purposes it is just taken to be the identity matrix. However for the special choice of

$$\Omega_t = \begin{cases} -\left(\frac{\partial f}{\partial \vec{a}}\right)^T_{t-1} \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a}\partial \vec{a}}\right)^{-1}_{t-1} \left(\frac{\partial f}{\partial \vec{a}}\right)_{t-1} & \text{for } t > 0 \\ 0 & \text{for } t = 0 \end{cases}, \tag{10}$$

the algorithm VGL(1) is proven to converge [9] when used in conjunction with a greedy policy, and under certain smoothness assumptions.

D. Definition of the ADP Algorithms HDP, DHP and GDHP

All of the ADP algorithms we will define here are particularly intended for the situation of actor-critic architectures. However for our divergence examples in this paper we are instead using the greedy policy. As detailed in section I-D, using an actor-critic architecture with value-iteration is very similar to using a greedy policy.

The three ADP algorithms we consider here can all be defined in terms of the algorithms defined so far in this paper.

- The algorithm Heuristic Dynamic Programming (HDP) uses the same weight update for its $\widetilde{V}$ function as TD(0).
- The algorithm Dual Heuristic Dynamic Programming (DHP) uses the same weight update for its $\widetilde{G}$ function as VGL(0). In DHP, the function $\widetilde{G}(\vec{x},\vec{w})$ is usually implemented as the output of a separate *vector* function approximator, but in this paper's divergence example we don't do this (instead we use $\widetilde{G} \equiv \frac{\partial \widetilde{V}}{\partial \vec{x}}$).
- Globalized Dual Heuristic Programming (GDHP) uses a linear combination of a weight update by VGL(0) and one by TD(0).

These ADP algorithms are traditionally used with a neural network to represent the critic. But this is not always necessarily the case; any differentiable structure will suffice [7]. In this paper we make use of simple quadratic functions to represent the critic.

III. Problem Definition For Divergence

We define the simple RL problem domain and function approximator suitable for providing divergence examples for the algorithms being tested.

First we define an environment with $\vec{x} \in \Re$ and $\vec{a} \in \Re$, and model functions:

$$f(x_t,t,a_t) = \begin{cases} x_t + a_t & \text{if } t \in \{0,1\} \\ x_t & \text{if } t = 2 \end{cases} \tag{11a}$$

$$r(x_t,t,a_t) = \begin{cases} -k{a_t}^2 & \text{if } t \in \{0,1\} \\ -{x_t}^2 & \text{if } t = 2 \end{cases} \tag{11b}$$

where $k > 0$ is a constant. Each trajectory is defined to terminate at time step $t = 3$, so that exactly three rewards are received by the agent (rewards are given at timings as defined in section I-A, i.e. with the final reward r_2 being received on transitioning from $t = 2$ to $t = 3$). In these model function definitions, action a_2 has no effect, so the whole trajectory is parametrised by just x_0, a_0 and a_1, and the total reward for this trajectory is $-k({a_0}^2+{a_1}^2)-(x_0+a_0+a_1)^2$. These model functions are dependent on t, which is an abuse of notation

we have adopted for brevity, but this could be legitimised by including t into $\vec{x}$.

The divergence example we derive below considers a trajectory which starts at $x_0 = 0$. From this start point, the optimal actions are $a_0 = a_1 = 0$.

A. Critic Definition

A critic function is defined using a weight vector with just four weights, $\vec{w} = (w_1, w_2, w_3, w_4)^T$:

$$\widetilde{V}(x_t, t, \vec{w}) = \begin{cases} -c_1 {x_1}^2 + w_1 x_1 + w_3 & \text{if } t = 1 \\ -c_2 {x_2}^2 + w_2 x_2 + w_4 & \text{if } t = 2 \\ 0 & \text{if } t \in \{0, 3\} \end{cases} \tag{12}$$

where c_1 and c_2 are real positive constants.

Hence the critic gradient function, $\widetilde{G} \equiv \frac{\partial \widetilde{V}}{\partial x}$, is given by:

$$\widetilde{G}(x_t, t, \vec{w}) = \begin{cases} -2c_t x_t + w_t & \text{if } t \in \{1, 2\} \\ 0 & \text{if } t \in \{0, 3\} \end{cases} \tag{13}$$

We note that this implies

$$\left(\frac{\partial \widetilde{G}}{\partial \vec{w}^k}\right)_t = \begin{cases} 1 & \text{if } t \in \{1, 2\} \text{ and } t = k \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

B. Actor Definition

In this problem it is possible to define a function approximator for the actor with sufficient flexibility to behave exactly like a greedy policy, provided the actor is trained in a value-iteration scheme. This is particularly easy to do here, since the trajectory is defined to have a fixed start point $x_0 = 0$. For example, if we define the weight vector of the actor, $\vec{z}$, to have just two components, so that $\vec{z} = (z_0, z_1)$, and then define the output of the actor to be the identity function of these two weights, so that $a_0 \equiv z_0$ and $a_1 \equiv z_1$, then training the actor to completion would be equivalent to solving the greedy policy's maximum condition. This enables the divergence results of this paper to also apply to actor-critic architectures, as discussed in section I-D.

C. Unrolling a greedy trajectory

Substituting the model functions (eq. 11) and the critic definition (eq. 12) into the $\widetilde{Q}$ function definition (eq. 2) gives, with $\gamma = 1$,

$$\widetilde{Q}(x_t, t, a_t, \vec{w}) = \begin{cases} -k(a_0)^2 - c_1(x_0 + a_0)^2 + w_1(x_0 + a_0) + w_3 & \text{if } t = 0 \\ -k(a_1)^2 - c_2(x_1 + a_1)^2 + w_2(x_1 + a_1) + w_4 & \text{if } t = 1 \end{cases}$$

In order to maximise this with respect to a_t and get greedy actions, we first differentiate to get,

$$\begin{aligned} \left(\frac{\partial \widetilde{Q}}{\partial a}\right)_t &= -2ka_t - 2c_{t+1}(x_t + a_t) + w_{t+1} \qquad \text{for } t \in \{0, 1\} \\ &= -2a_t(c_{t+1} + k) + w_{t+1} - 2c_{t+1}x_t \quad \text{for } t \in \{0, 1\} \end{aligned} \tag{15}$$

Hence the greedy actions are given by

$$a_0 \equiv \frac{w_1 - 2c_1 x_0}{2(c_1 + k)} \tag{16}$$

$$a_1 \equiv \frac{w_2 - 2c_2 x_1}{2(c_2 + k)} \tag{17}$$

Following these actions along a trajectory starting at $x_0 = 0$, and using the recursion $x_{t+1} = f(x_t, a_t)$ with the model functions (eq. 11) gives

$$x_1 = a_0 = \frac{w_1}{2(c_1 + k)} \tag{18}$$

$$\text{and } x_2 = x_1 + a_1 = \frac{w_2(c_1 + k) + kw_1}{2(c_2 + k)(c_1 + k)} \tag{19}$$

Substituting x_1 (eq. 18) back into the equation for a_1 (eq. 17) gives a_1 purely in terms of the weights and constants:[1]

$$a_1 \equiv \frac{w_2(c_1 + k) - c_2 w_1}{2(c_2 + k)(c_1 + k)} \tag{20}$$

D. Evaluation of value-gradients along the greedy trajectory

We can now evaluate the $\widetilde{G}$ values by substituting the greedy trajectory's state vectors (eqs. 18-19) into eq. 13, giving:

$$\widetilde{G}_1 = -\frac{c_1 w_1}{(c_1 + k)} + w_1 = \frac{w_1 k}{(c_1 + k)} \tag{21}$$

$$\begin{aligned} \text{and } \widetilde{G}_2 &= -\frac{w_2(c_1 + k)c_2 + kw_1 c_2}{(c_2 + k)(c_1 + k)} + w_2 \\ &= \frac{w_2 k(c_1 + k) - kw_1 c_2}{(c_2 + k)(c_1 + k)} \end{aligned} \tag{22}$$

The greedy actions in equations 16 and 17 both satisfy

$$\left(\frac{\partial \pi}{\partial x}\right)_t = \begin{cases} \frac{-c_{t+1}}{c_{t+1} + k} & \text{for } t \in \{0, 1\} \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

Substituting eqs. 23 and 11 into $\frac{Df}{Dx} = \frac{\partial f}{\partial x} + \frac{\partial \pi}{\partial x}\frac{\partial f}{\partial a}$ gives

$$\left(\frac{Df}{Dx}\right)_t = \begin{cases} 1 - \frac{c_{t+1}}{c_{t+1} + k} = \frac{k}{c_{t+1} + k} & \text{if } t \in \{0, 1\} \\ 1 & \text{if } t = 2 \end{cases} \tag{24}$$

Similarly, substituting them into $\frac{Dr}{Dx} = \frac{\partial r}{\partial x} + \frac{\partial \pi}{\partial x}\frac{\partial r}{\partial a}$ gives

$$\left(\frac{Dr}{Dx}\right)_t = \begin{cases} 0 - \frac{c_{t+1}}{c_{t+1} + k}(-2ka_t) = \frac{2kc_{t+1}a_t}{c_{t+1} + k} & \text{if } t \in \{0, 1\} \\ -2x_t & \text{if } t = 2 \end{cases} \tag{25}$$

E. Backwards pass along trajectory

We do a backwards pass along the trajectory calculating the target gradients using eq. 8 with $\gamma = 1$, and starting with $\widetilde{G}_3 = 0$ (by eq. 13) and $G'_3 = 0$ (since G'_3 is at a terminal state):

$$\begin{aligned} G'_2 &= \left(\frac{Dr}{Dx}\right)_2 && \text{by eq. 8 and } G'_3 = \widetilde{G}_3 = 0 \\ &= -2x_2 && \text{by eq. 25} \\ &= -\frac{w_2(c_1 + k) + kw_1}{(c_2 + k)(c_1 + k)} && \text{by eq. 19} \end{aligned} \tag{26}$$

[1] We emphasise that we are doing this step for the divergence analysis, and that this is *not* the way that VGL is meant to be implemented in practice.

Similarly,

$$\begin{aligned}
G'_1 &= \left(\frac{Dr}{Dx}\right)_1 + \left(\frac{Df}{Dx}\right)_1 \left(\lambda G'_2 + (1-\lambda)\widetilde{G}_2\right) && \text{by eq. 8}\\
&= \frac{2kc_2a_1}{c_2+k} + \frac{k}{c_2+k}\left(\lambda G'_2 + (1-\lambda)\widetilde{G}_2\right) && \text{by eqs. 25,24}\\
&= \frac{kc_2(w_2(c_1+k)-c_2w_1)}{(c_1+k)(c_2+k)^2}\\
&\quad + \frac{k}{c_2+k}\left(-\lambda\frac{w_2(c_1+k)+kw_1}{(c_2+k)(c_1+k)}\right.\\
&\quad \left. +(1-\lambda)\frac{w_2k(c_1+k)-kw_1c_2}{(c_2+k)(c_1+k)}\right) && \text{by eqs.20,22,26}\\
&= \frac{w_2k(c_2-\lambda+k(1-\lambda))}{(c_2+k)^2}\\
&\quad - \frac{w_1k(k\lambda+(c_2)^2+k(1-\lambda)c_2)}{(c_1+k)(c_2+k)^2} && (27)
\end{aligned}$$

IV. Divergence Examples for VGL and DHP Algorithms

We now have the whole trajectory and the terms $\widetilde{G}$ and G' written algebraically, so that we can next analyse the VGL(λ) weight update for divergence.

The VGL(λ) weight update (eq. 7) combined with Ω_t=1 gives

$$\begin{aligned}
\Delta w_i &= \alpha \sum_t \left(\frac{\partial \widetilde{G}}{\partial w_i}\right)_t (G'_t - \widetilde{G}_t) && (28a)\\
&= \alpha(G'_i - \widetilde{G}_i) && (\text{for } i \in \{1,2\}, \text{ by eq. 14})\\
\Rightarrow \begin{pmatrix}\Delta w_1\\ \Delta w_2\end{pmatrix} &= \alpha \begin{pmatrix}G'_1 - \widetilde{G}_1\\ G'_2 - \widetilde{G}_2\end{pmatrix}\\
&= \alpha A \begin{pmatrix}w_1\\ w_2\end{pmatrix} && (28b)
\end{aligned}$$

where A is a 2×2 matrix with elements found by subtracting equations 21 and 22 from equations 27 and 26, respectively, giving,

$$A = \begin{pmatrix} -\frac{k(k\lambda+(c_2)^2+k(1-\lambda)c_2)}{(c_1+k)(c_2+k)^2} - \frac{k}{(c_1+k)} & \frac{k(c_2+k-\lambda(k+1))}{(c_2+k)^2} \\ \frac{k(c_2-1)}{(c_2+k)(c_1+k)} & \frac{-1-k}{(c_2+k)} \end{pmatrix} \quad (29)$$

Equation 28b is the VGL(λ) weight update written as a single dynamic system of just *two* variables, i.e. a shortened weight vector, $\vec{w} = (w_1, w_2)^T$. For this shortened weight vector, $\vec{w}$, by looking at the right-hand sides of the sequence of equations from eq. 28a to eq. 28b, we can conclude that

$$\sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t (G'_t - \widetilde{G}_t) = A\vec{w} \quad (30)$$

To add further complexity to the system, in order to achieve the desired divergence, we next define these two weights to be a linear function of two *other* weights, $\vec{p} = (p_1, p_2)^T$, such that the shortened weight vector is given by $\vec{w} = F\vec{p}$, where F is a 2×2 constant real matrix. The VGL(λ) weight update equation can now be recalculated for these new weights, as follows:

$$\begin{aligned}
\Delta\vec{p} &= \alpha \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{p}}\right)_t (G'_t - \widetilde{G}_t) && \text{by eq. 7 and } \Omega_t\text{=1}\\
&= \alpha \sum_t \frac{\partial \vec{w}}{\partial \vec{p}}\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t (G'_t - \widetilde{G}_t) && \text{by chain rule}\\
&= \alpha \frac{\partial \vec{w}}{\partial \vec{p}} \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t (G'_t - \widetilde{G}_t) && \text{since independent of } t\\
&= \alpha \frac{\partial \vec{w}}{\partial \vec{p}} A\vec{w} && \text{by eq. 30}\\
&= \alpha(F^T A F)\vec{p}. && \text{by } \vec{w} = F\vec{p} \text{ and } \frac{\partial \vec{w}}{\partial \vec{p}} = \frac{\partial(F\vec{p})}{\partial \vec{p}} = F^T
\end{aligned} \quad (31)$$

The optimal actions $a_0 = a_1 = 0$ would be achieved by $\vec{p} = \vec{0}$. To produce a divergence example, we want to ensure that $\vec{p}$ does *not* converge to $\vec{0}$.

Taking $\alpha > 0$ to be sufficiently small, then the weight vector $\vec{p}$ evolves according to a continuous-time linear dynamic system given by eq. 31, and this system is stable if and only if the matrix product $F^T AF$ is "stable" (i.e. if the real part of every eigenvalue of this matrix product is negative).

Choosing $\lambda = 0$ and $c_1 = c_2 = k = 0.01$ gives $A = \begin{pmatrix}-0.75 & 0.5\\ -24.75 & -50.5\end{pmatrix}$ (by equation 29). Choosing $F = \begin{pmatrix}10 & 1\\ -1 & -1\end{pmatrix}$ makes $F^T AF = \begin{pmatrix}117.0 & -38.25\\ 189.0 & -27.0\end{pmatrix}$ which has eigenvalues $45 \pm 45.22i$. Since the real parts of these eigenvalues are positive, eq. 31 will diverge for VGL(0) (i.e. DHP). In an extended analysis, we found that these parameters also cause VGL(0) to diverge when the Ω_t matrices are included according to equation 10 (see the Appendix for further details).

Since GDHP is a linear combination of DHP, which we have proven to diverge, and TD(0) (which we prove to diverge below), it follows that GDHP can diverge with a greedy policy too.

Also, perhaps surprisingly, it is possible to get instability with VGL(1). Choosing $c_2 = k = 0.01$, $c_1 = 0.99$ gives $A = \begin{pmatrix}-0.2625 & -24.75\\ -0.495 & -50.5\end{pmatrix}$. Choosing $F = \begin{pmatrix}-1 & -1\\ .2 & .02\end{pmatrix}$ makes $F^T AF = \begin{pmatrix}2.7665 & 0.1295\\ 4.4954 & 0.2222\end{pmatrix}$ which has two positive real eigenvalues. Therefore this VGL(1) system diverges.

Diverging weights are shown for the VGL(0) and VGL(1) algorithms in Figure 1, with a learning rate of $\alpha = 10^{-6}$. Both experiments (and all subsequent experiments in this paper) used a starting weight vector of $(p_1, p_2, w_3, w_4) = (5.23{*}10^{-5}, 8.53{*}10^{-5}, 0, 0)$, which is based upon a principal eigenvector of the $F^T AF$ matrix found to make VGL(1) diverge.

The divergence result for VGL(1) does not affect the convergence result by [9] which is for VGL(1) but with the special choice of Ω_t given by eq. 10, which we will refer to as VGLΩ(1). It was not possible to make VGLΩ(1) diverge with the methods of this paper (see Appendix for futher details). Figure 2 shows VGLΩ(1) converging using the same learning parameters that made VGL(1) diverge.

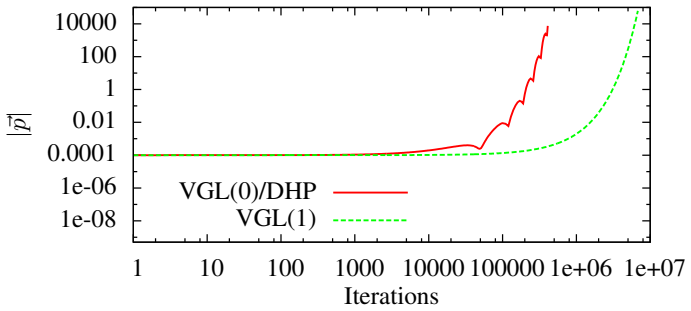

Fig. 1. Divergence for VGL(0) (i.e. DHP) and VGL(1) using the learning parameters described in section IV and a learning rate of $\alpha = 10^{-6}$.

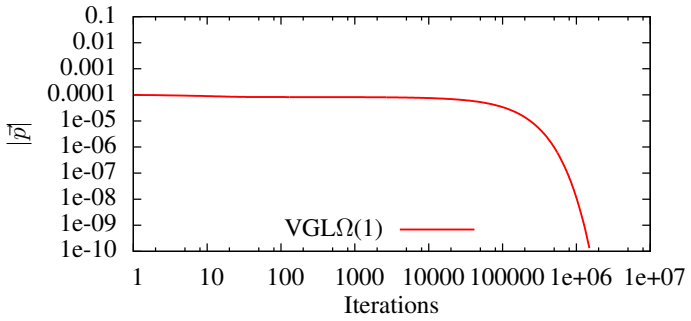

Fig. 2. Convergence for VGLΩ(1) using the same parameters that caused VGL(1) to diverge, and $\alpha = 10^{-3}$. This algorithm demonstrates that it *is* possible to have proven convergence for a critic learning algorithm with a greedy policy and general function approximation.

V. Divergence results for TD(λ), Sarsa(λ) and HDP

To satisfy the requirement for exploration in TD(λ)-based algorithms, we supplemented the greedy policies (eqs. 16 & 17) with a small amount of stochastic Gaussian noise with zero mean and variance 0.0001. This Gaussian noise was necessary, since it is well known that these classic RL algorithms must be supplemented with some form of exploration. This is the classic "exploration versus exploitation" dilemma. Without exploration, these algorithms do not converge to an optimal policy, in general. Specific examples of converging to the wrong policy without exploration are given by [17, sec. IV.H] and [15, appendix B].

To achieve divergence of these algorithms with the noisy greedy policy, we used exactly the same learning and environment constants as used for the VGL(0) and VGL(1) divergence experiments. These choices of parameters, with the stochastic noise added to the greedy policy, made TD(0) and TD(1) diverge respectively, as shown in figures 3 and 4. Hence HDP diverges too, since this is equivalent to TD(0) with the given policy.

Although these divergence results for the TD(λ) based algorithms were only found empirically, as opposed to the results for the previous sections which were first found analytically, these results do still have value. Firstly, source code for the empirical experiments used here is provided by [18, in ancillary files], so the empirical results should be entirely replicable. Secondly, an insight into why the divergence parameters for VGL were sufficient to make the TD(λ) based algorithms diverge too is because TD with stochastic exploration can be understood to be an approximation to a stochastic version of VGL(λ), so we would *expect* a divergence example for VGL to cause divergence for TD(λ) too.

Without the stochastic noise added to the greedy policy, these examples would not diverge, but instead converge to a sub-optimal policy, which is also considered a failure.

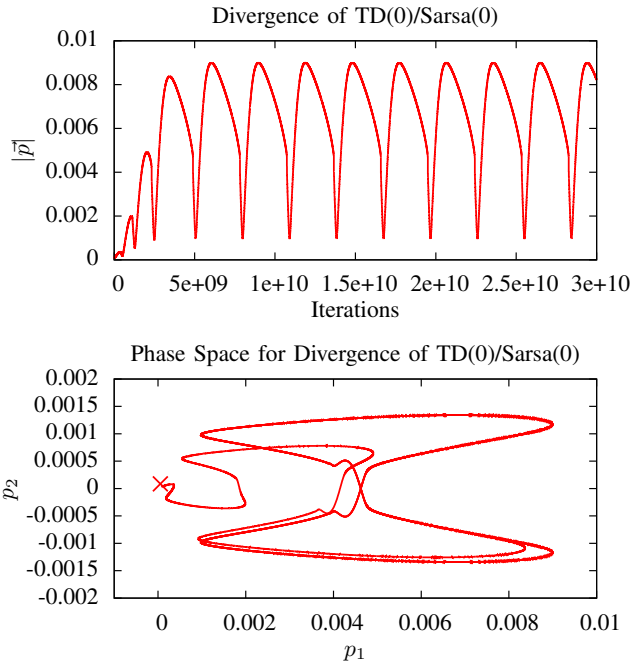

Fig. 3. Divergence for TD(0) and Sarsa(0) generated with the diverging parameters described in section V, and a learning rate of $\alpha = 10^{-6}$. The upper graph shows progress of $|\vec{p}|$ versus iterations. The lower graph shows the corresponding evolution of the weight vector components (p_1, p_2) in phase space. This phase curve *starts* close to the origin (at the 'X'), and finishes off in a limit cycle. **(Errata, 2015-09): The phase space graph is plotted incorrectly. See Fig. 9.2 of first author's phd thesis for a correction.**

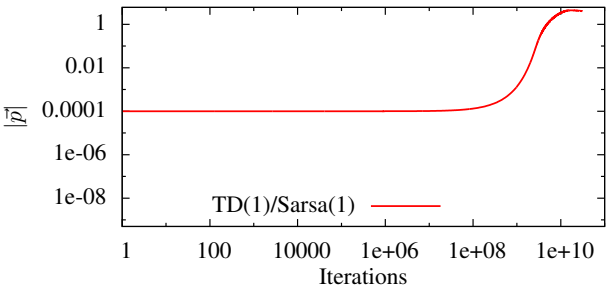

Fig. 4. Divergence for TD(1) and Sarsa(1) generated with the diverging parameters described in section V, and a learning rate of $\alpha = 10^{-6}$.

A. Divergence results for Sarsa(λ)

We next prove divergence for Sarsa(λ) by choosing a function approximator for $\widetilde{Q}$ that makes the Sarsa(λ) weight update equivalent to the TD(λ) weight update, so that the divergence result for TD(λ) carries over to Sarsa(λ).

Sarsa(λ) is designed to work with an arbitrary function approximator for $\widetilde{Q}(\vec{x}, \vec{a}, \vec{w})$. We will define our $\widetilde{Q}$ function exactly by Eq. 2. Rearranging eq. 6 gives

$$\begin{aligned}\left(\frac{{Q^\lambda}_t - r_t}{\gamma}\right) &= \lambda {Q^\lambda}_{t+1} + (1-\lambda)\widetilde{Q}_{t+1} \\ &= \lambda {Q^\lambda}_{t+1} + (1-\lambda)(r_{t+1} + \gamma \widetilde{V}_{t+2}) \qquad \text{by eq. 2} \\ &= r_{t+1} + \lambda({Q^\lambda}_{t+1} - r_{t+1}) + (1-\lambda)(\gamma \widetilde{V}_{t+2}) \\ &= r_{t+1} + \gamma\left(\lambda\left(\frac{{Q^\lambda}_{t+1} - r_{t+1}}{\gamma}\right) + (1-\lambda)\widetilde{V}_{t+2}\right)\end{aligned} \tag{32}$$

From this we can see that $\left(\frac{{Q^\lambda}_t - r_t}{\gamma}\right)$ obeys the same recursion equation as R^λ, and they have the same endpoint (since both are zero at a terminal state), from which we can conclude (e.g. by comparing recursion equations 32 and 4) that

$$\begin{aligned}\left(\frac{{Q^\lambda}_t - r_t}{\gamma}\right) &\equiv {R^\lambda}_{t+1} \\ \Rightarrow {Q^\lambda}_t &= r_t + \gamma {R^\lambda}_{t+1}\end{aligned}$$

Substituting this into the Sarsa(λ) weight update (eq. 5), with eq. 2, and simplifying gives

$$\begin{aligned}\Delta\vec{w} =& \alpha \sum_t \left(\frac{\partial(r_t + \gamma \widetilde{V}(\vec{x}_{t+1}, \vec{w}))}{\partial \vec{w}}\right)_t (r_t \\ &+ \gamma {R^\lambda}_{t+1} - (r_t + \gamma \widetilde{V}_{t+1})\Big) \\ =& \alpha \sum_t \gamma \left(\frac{\partial \widetilde{V}}{\partial \vec{w}}\right)_{t+1} \gamma(R^\lambda{}_{t+1} - \widetilde{V}_{t+1}) \\ =& \alpha\gamma^2 \sum_{t>0} \left(\frac{\partial \widetilde{V}}{\partial \vec{w}}\right)_t (R^\lambda{}_t - \widetilde{V}_t)\end{aligned}$$

which is identical to TD(λ) but with summation over t now excluding $t = 0$, and with an extra constant factor, γ^2. The divergence example we derived above used $\gamma = 1$, and had no weight update term for $t = 0$, so uses an identical weight update. Therefore this particular choice of function approximator for $\widetilde{Q}$ and problem definition causes divergence for Sarsa(λ) (with both $\lambda = 1$ and $\lambda = 0$).

VI. Conclusions

We have shown that under a value-iteration scheme, i.e. using a greedy policy, all of the RL algorithms have been made to diverge, and all but one of the VGL algorithms have been made to diverge. The algorithm we found that didn't diverge was VGL$\Omega(1)$ with Ω_t as defined by eq. 10, which is proven to converge by [9] and [15] under these conditions.

These are new divergence results for TD(0), Sarsa(0), TD(1) and Sarsa(1), in that previous examples of divergence have only been for TD(0) and for non-greedy policies [19], [20], [11]. The divergences we achieved for TD(1) and Sarsa(1) were only possible because of the use of a greedy policy (or equivalently, value-iteration).

A conclusion of this work is that the diverging algorithms considered cannot currently be reliably used for value-iteration, and instead can only be used under some form of policy iteration if provable convergence is required. However there are some distinct advantages of value-iteration over policy-iteration. Value-iteration using a greedy policy can be faster than using an actor-critic architecture. Also policy iteration does provably converge in some cases [21], but the necessary conditions are thought to apply only when the function approximator for $\widetilde{V}$ is *linear* in the same features of the state vector that the function approximator for the policy uses as input (see footnote 1 of [21]).

The divergence results of this paper were derived for quadratic critic functions, as this was the situation that allowed for easiest analysis to derive concrete divergence examples. We assume that similar divergence results will exist for neural network based critic functions, since neural networks are more complex structures that should allow for more possibilities for divergence situations similar to our simple example here. In our experience, divergence often does occur when using a greedy policy with a neural network critic, but these situations are harder to analyse and make replicable. In this situation, we speculate that a second order Taylor series expansion of the neural network could be made about the fixed point of the learning process, and locally this approximation could be behaving very similarly to the quadratic functions we have used in this paper.

It is hoped that the specific divergence examples of this paper will provide a better understanding of how value-iteration can diverge, and help motivate research to understand and prevent it. We believe that the value-gradient analysis that produced the converging algorithm of Figure 2 by [9] could be helpful for reinforcement learning research, since this is a critic learning algorithm that does have convergence guarantees under a greedy policy with general function approximation.

Appendix

In this appendix we give the extension analysis that was used to determine that VGL$\Omega(0)$ (i.e. VGL(0) with the Ω_t matrix of eq. 10) could be made to diverge. We also include an analysis that shows VGL$\Omega(1)$ will converge in the experiment of this paper for any choice of experimental constants.

To construct the Ω_t matrix of eq. 10, first we note that differentiating equation 11a gives $\left(\frac{\partial f}{\partial a}\right)_t = 1$, for $t \in \{0, 1\}$. And differentiating equation 15 gives

$$\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t = -2(c_{t+1} + k) \qquad \text{for } t \in \{0, 1\}.$$

Hence, by equation 10,

$$\Omega_t = \begin{cases} 1/(2(c_t + k)) & \text{for } t \in \{1, 2\} \\ 0 & \text{for } t = 0. \end{cases}$$

The VGL(λ) weight update can be re-derived using this new Ω_t matrix. Following the method that was used to derive equations 28a to 30, but starting with this new Ω_t matrix, gives

$$\sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t(G'_t - \widetilde{G}_t) = \alpha D A \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$$

where $D = \begin{pmatrix} \frac{1}{2(c_1+k)} & 0 \\ 0 & \frac{1}{2(c_2+k)} \end{pmatrix}$ and A is given by equation 29. Then, defining $\vec{w} = F\vec{p}$ for a constant matrix F (as done in section IV), and following the method that was used to derive eq. 31, we would derive the VGL(λ) weight update for the weight vector $\vec{p}$ as

$$\Delta\vec{p} = \alpha(F^T DAF)\vec{p}.$$

As before, this system will converge for sufficiently small α if and only if the product $F^T DAF$ is "stable", i.e. if the real parts of the eigenvalues are negative.

A. Divergence of VGLΩ(0)

Choosing the same parameters that made VGL(0) diverge, i.e. $c_1 = c_2 = k = 0.01$, gives $D = \begin{pmatrix} 25 & 0 \\ 0 & 25 \end{pmatrix}$. Since D is a positive multiple of the identity matrix, its presence will not affect the stability of the product $F^T DAF$, so the system for $\vec{p}$ will still be unstable, and diverge, just as it did for VGL(0).

B. Convergence of VGLΩ(1)

When VGLΩ(1) is used, convergence can be proven for any choice of parameters as follows: When $\lambda = 1$, the A matrix of eq. 29 reduces to[2]

$$\begin{aligned} A =& \begin{pmatrix} -\frac{k(k+(c_2)^2)}{(c_1+k)(c_2+k)^2} - \frac{k}{(c_1+k)} & \frac{k(c_2-1)}{(c_2+k)^2} \\ \frac{k(c_2-1)}{(c_2+k)(c_1+k)} & \frac{-1-k}{(c_2+k)} \end{pmatrix} \\ =& 2\begin{pmatrix} -\frac{k(k+(c_2)^2)}{(c_2+k)^2} - k & \frac{k(c_2-1)}{(c_2+k)} \\ \frac{k(c_2-1)}{(c_2+k)} & -1-k \end{pmatrix} D \\ =& 2E\begin{pmatrix} -k(k+(c_2)^2+(c_2+k)^2) & k(c_2-1) \\ k(c_2-1) & -1-k \end{pmatrix} ED \end{aligned}$$

where $E = \begin{pmatrix} \frac{1}{(c_2+k)} & 0 \\ 0 & 1 \end{pmatrix}$. Hence the matrix product $F^T DAF$ can now be written as $2F^T DEBEDF$ where $B = \begin{pmatrix} -k(k+(c_2)^2+(c_2+k)^2) & k(c_2-1) \\ k(c_2-1) & -1-k \end{pmatrix}$. This new product is real and symmetrical (as we would expect it to be for true gradient descent), hence it has real eigenvalues. For any $c_2 > 0$ and $k > 0$, the central matrix B has a negative trace, and a determinant equal to $k(k+2)(k+c_2)^2$, which is positive. Hence B has two negative real eigenvalues. Therefore, assuming F is a full-rank matrix, the matrix product $2F^T DEBEDF$ must be negative definite, and therefore stable, and thus the dynamic system for $\vec{p}$ will converge.

REFERENCES

[1] F.-Y. Wang, H. Zhang, and D. Liu, "Adaptive dynamic programming: An introduction," *IEEE Computational Intelligence Magazine*, vol. 4, no. 2, pp. 39–47, 2009.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachussetts, USA: The MIT Press, 1998.

[3] R. E. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1957.

[4] G. Rummery and M. Niranjan, "On-line q-learning using connectionist systems," *Tech. Rep. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department*, 1994.

[5] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, 1988.

[6] P. J. Werbos, "Approximating dynamic programming for real-time control and neural modeling." in *Handbook of Intelligent Control*, D. A. White and D. A. Sofge, Eds. New York: Van Nostrand Reinhold, 1992, ch. 13, pp. 493–525.

[7] D. Prokhorov and D. Wunsch, "Adaptive critic designs," *IEEE Transactions on Neural Networks*, vol. 8, no. 5, pp. 997–1007, 1997.

[8] S. Ferrari and R. F. Stengel, "Model-based adaptive critic designs," in *Handbook of learning and approximate dynamic programming*, J. Si, A. Barto, W. Powell, and D. Wunsch, Eds. New York: Wiley-IEEE Press, 2004, pp. 65–96.

[9] M. Fairbank and E. Alonso, "The local optimality of reinforcement learning by value gradients, and its relationship to policy gradient learning," *CoRR*, vol. abs/1101.0428, 2011. [Online]. Available: http://arxiv.org/abs/1101.0428

[10] ——, "Value-gradient learning," in *Proceedings of the IEEE International Joint Conference on Neural Networks 2012 (IJCNN'12)*. IEEE Press, June 2012, pp. 3062–3069.

[11] J. N. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," IEEE Transactions on Automatic Control, Tech. Rep., 1996.

[12] R. S. Sutton, H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári, and E. Wiewiora, "Fast gradient-descent methods for temporal-difference learning with linear function approximation," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. New York, NY, USA: ACM, 2009, pp. 993–1000.

[13] H. Maei, C. Szepesvári, S. Bhatnager, D. Precup, D. Silver, and R. Sutton, "Convergent temporal-difference learning with arbitrary smooth function approximation," in *Advances in Neural Information Processing Systems (NIPS'09)*. MIT Press, 2009.

[14] P. J. Werbos, "Stable adaptive control using new critic designs," *eprint arXiv:adap-org/9810001*, 1998.

[15] M. Fairbank, "Reinforcement learning by value gradients," *CoRR*, vol. abs/0803.3539, 2008. [Online]. Available: http://arxiv.org/abs/0803.3539

[16] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, Cambridge University, 1989.

[17] M. Fairbank and E. Alonso, "A comparison of learning speed and ability to cope without exploration between DHP and TD(0)," in *Proceedings of the IEEE International Joint Conference on Neural Networks 2012 (IJCNN'12)*. IEEE Press, June 2012, pp. 1478–1485.

[18] ——, "The divergence of reinforcement learning algorithms with value-iteration and function approximation," *eprint arXiv:1107.4606*, 2011.

[19] L. C. Baird, "Residual algorithms: Reinforcement learning with function approximation," in *International Conference on Machine Learning*, 1995, pp. 30–37.

[20] J. N. Tsitsiklis and B. Van Roy, "Feature-based methods for large scale dynamic programming," *Machine Learning*, vol. 22, no. 1-3, pp. 59–94, 1996.

[21] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems 12*, vol. 12, 2000, pp. 1057–1063.

[2] This version of this document contains a fix to the following equations - the constant factor 2 was missing from the version published in the proceedings of IJCNN12.