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

Michael Fairbank\textsuperscript{1} and Eduardo Alonso\textsuperscript{1}
\textsuperscript{1}Department of Computing, School of Informatics, City University London, London, UK

\textbf{Keywords:} Value Iteration, Reinforcement Learning, Adaptive Dynamic Programming

\section*{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.

\section{Introduction}

Adaptive Dynamic Programming (ADP) (Wang et al. 2009) and Reinforcement Learning (RL) (Sutton & Barto 1998) 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 (Bellman 1957). 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(\(\lambda\)) (Rummery & Niranjan 1994), TD(\(\lambda\)) (Sutton 1988), and the ADP algorithms Heuristic Dual Programming (HDP), Dual Heuristic Dynamic Programming (DHP), Globalized Dual Heuristic Dynamic Programming (GDHP) (Werbos 1992, Prokhorov & Wunsch 1997, Ferrari & Stengel 2004) and Value-Gradient Learning (VGL(\(\lambda\)), Fairbank & Alonso 2011). We prove divergence of all of these algorithms (including VGL(0), VGL(1), Sarsa(0), Sarsa(1), TD(0) and TD(1)), 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($\lambda$) is proven to converge when $\lambda = 1$ since it is then (and only then) true gradient descent on an error function (Sutton 1988). Also for $0 \leq \lambda \leq 1$, it is proven to converge by Tsitsiklis & Van Roy (1996a) when the approximate value function is linear in its weight vector and learning is “on-policy”. However these convergence proofs do not apply to a greedy policy that we consider in this paper.

Ferrari & Stengel (2004) show the 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, (Werbos 1998 sections 7.7-7.8) 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 (Fairbank & Alonso, 2011, Lemma 7) 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. Fairbank & Alonso (2011) and Fairbank (2008) 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 1.1-1.4), we state the general RL/ADP problem and give the necessary function definitions. In section 2 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 3.1-3.3). We then analyse a trajectory for this problem (sections 3.2-3.4), so that we can write the VGL($\lambda$) 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 4). The VGL($\lambda$) weight update is easier to analyse than the TD($\lambda$) one, since as mentioned above the greedy policy depends on the value-gradient, so in section 5 we just use the same learning parameters that caused divergence for VGL($\lambda$) and find empirically that they cause the other algorithms to diverge too.

Finally in section 6 we give conclusions and discuss the difficulty of ensuring value-iteration convergence but its potential advantages compared to policy-iteration.

1.1 RL and ADP Problem Definition and Notation

The typical RL/ADP scenario is an agent wandering around in an environment, such that at time $t$ it has state vector $\vec{x}_t$. At each time $t$ the agent chooses
an action $\bar{a}_t$ which takes it to the next state according to the environment’s model function $\bar{x}_{t+1} = f(\bar{x}_t, \bar{a}_t)$, and gives it an immediate reward, $r_t$, given by the function $r_t = r(\bar{x}_t, \bar{a}_t)$. In general these model functions $f$ and $r$ can be stochastic functions. The agent keeps moving, forming a trajectory of states $(\bar{x}_0, \bar{x}_1, \ldots)$, which terminates if and when a designated terminal state is reached.

In RL/ADP, we aim to find a policy function, $\pi(\bar{x})$, that calculates which action $\bar{a} = \pi(\bar{x})$ to take for any given state $\bar{x}$. 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.

There are only minor differences between the ADP and RL learning methods that we know of: one difference is that RL methods commonly place more emphasis on model-free learning than ADP methods do, whereas ADP methods often assume the model functions are already known and therefore can be made use of during learning.

1.2 Approximate Value Function (Critic) and its Gradient

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

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, $\tilde{G}$, $\frac{\partial \tilde{V}}{\partial x}$ and $\frac{\partial \tilde{V}}{\partial w}$ are all columns.

1.3 Greedy Policy

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

$$\bar{a} = \arg \max_{\bar{a} \in \mathbb{R}^n} (\tilde{Q}(\bar{x}, \bar{a}, \bar{w})) \quad \forall \bar{x}$$

where we define the approximate Q Value function [Watkins, 1989] as

$$\tilde{Q}(\bar{x}, \bar{a}, \bar{w}) = r(\bar{x}, \bar{a}) + \gamma \tilde{V}(f(\bar{x}, \bar{a}), \bar{w})$$

1.4 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 $\bar{x}_t$ and $\bar{a}_t$ where appropriate; so that for example $\tilde{V}_{t+1} \equiv \tilde{V}(\bar{x}_{t+1}, \bar{w})$; $\left( \frac{\partial \tilde{Q}}{\partial \bar{w}} \right)_t$ is shorthand for $\frac{\partial \tilde{Q}(\bar{x}_t, \bar{a}_t, \bar{w})}{\partial \bar{w}}$ and $\left( \frac{\partial \tilde{V}}{\partial \bar{w}} \right)_t$ is shorthand for $\frac{\partial \tilde{V}(\bar{x}_t, \bar{w})}{\partial \bar{w}}$. 

3
2 Learning Algorithms and Definitions

2.1 TD(\(\lambda\)) Learning

The TD(\(\lambda\)) algorithm (Sutton, 1988) can be defined in batch mode by the following weight update applied to an entire trajectory:

\[
\Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{V}}{\partial \vec{w}} \right)_t (R^\lambda_t - \tilde{V}_t) \tag{3}
\]

where \(\lambda \in [0, 1]\), and \(\alpha > 0\) are fixed constants. \(R^\lambda\) is the (moving) target for this weight update. It is known as the “\(\lambda\)-Return”, as defined by Watkins (1989). For a given trajectory, this can be written concisely using trajectory shorthand notation by the recursion

\[
R^\lambda_t = r_t + \gamma (\lambda R^\lambda_{t+1} + (1 - \lambda) \tilde{V}_{t+1}) \tag{4}
\]

with \(R^\lambda_t = 0\) at any terminal state, as proven by Fairbank & Alonso (2011, Appendix A). This equation introduces the dependency on \(\lambda\) into eq. 3. Using the \(\lambda\)-Return enables us to write TD(\(\lambda\)) in this very concise way, known as the “forwards view of TD(\(\lambda\))” (Sutton & Barto, 1998), however the traditional way to implement the algorithm is using “eligibility traces”, as described by Sutton (1988).

2.2 Sarsa(\(\lambda\)) Algorithm

Sarsa(\(\lambda\)) is an algorithm for control problems that learns to approximate the \(\tilde{Q}(\vec{x}, \vec{a}, \vec{w})\) function (Rummery & Niranjan, 1994). It is designed for policies that are dependent on the \(\tilde{Q}(\vec{x}, \vec{a}, \vec{w})\) function (e.g. the greedy policy or a greedy policy with added stochastic noise), where \(\tilde{Q}(\vec{x}, \vec{a}, \vec{w})\) here is defined to be the output of a given function approximator.

The Sarsa(\(\lambda\)) 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:

\[
\Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{Q}}{\partial \vec{w}} \right)_t (Q^\lambda_t - \tilde{Q}_t) \tag{5}
\]

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

\[
Q^\lambda_t = r_t + \gamma (\lambda Q^\lambda_{t+1} + (1 - \lambda) \tilde{Q}_{t+1}) \tag{6}
\]

with \(Q^\lambda_t = 0\) at any terminal state.
2.3 The VGL($\lambda$) Algorithm

To define the VGL($\lambda$) 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 $\frac{\partial f}{\partial \vec{x}}(\vec{x}, \vec{a})_j \frac{\partial \vec{x}}{\partial i}$. Similarly, $\left(\frac{\partial \vec{G}}{\partial \vec{w}}\right)^T_j = \frac{\partial \vec{G}}{\partial \vec{w}}_j$, and $\left(\frac{\partial \vec{G}}{\partial \vec{w}}\right)_t$ 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:

$$\Delta \vec{w} = \alpha \sum_t \left(\frac{\partial \vec{G}}{\partial \vec{w}}\right)_t \Omega_t (G'_t - \vec{G}_t)$$  \hspace{1cm} (7)

where $\alpha$ is a small positive constant; $\vec{G}_t$ is the approximate value gradient; and $G'_t$ is the “target value gradient” defined recursively by:

$$G'_t = \left(\frac{D \gamma}{D \vec{x}}\right)_t + \gamma \left(\frac{D f}{D \vec{x}}\right)_t \left(\lambda G'_{t+1} + (1 - \lambda) \vec{G}_{t+1}\right)$$  \hspace{1cm} (8)

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

$$\frac{D}{D \vec{x}} = \frac{\partial}{\partial \vec{x}} + \frac{\partial \pi}{\partial \vec{x}} \frac{\partial}{\partial \vec{a}};$$  \hspace{1cm} (9)

and where all of these derivatives are assumed to exist. Equations 7, 8 and 9 define the VGL($\lambda$) algorithm. Fairbank & Alonso (2011) give further details, and pseudocode for both on-line and batch-mode implementations.

The $\Omega_t$ matrix was introduced by Werbos (1998), and can be chosen freely by the experimenter, but it is 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

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

the algorithm VGL(1) is proven to converge Fairbank & Alonso (2011) when used in conjunction with a greedy policy, and under certain smoothness assumptions.

2.4 Definition of ADP Algorithms (HDP, DHP and GDHP)

All of the ADP algorithms we will define here are particularly intended for the situation where $\vec{V}$ is implemented as the output of a neural network, and the policy function is implemented as the output of a second neural network. However for our divergence examples in this paper we are instead using the greedy policy. Excluding this difference, 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 $\tilde{V}$ function as TD(0).

• The algorithm Dual Heuristic Dynamic Programming (DHP) uses the same weight update for its $\tilde{G}$ function as VGL(0). The function $\tilde{G}(\vec{x}, \vec{w})$ is generally implemented as the output of a vector function approximator, i.e. without it having to explicitly be the gradient $\frac{\partial \tilde{V}}{\partial \vec{x}}$.

• Globalized Dual Heuristic Programming (GDHP) uses a linear combination of a weight update by VGL(0) and one by TD(0).

3 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 $\vec{x} \in \mathbb{R}$ and $\vec{a} \in \mathbb{R}$, and model functions:

\[
\begin{align*}
    f(x_t, t, a_t) &= \begin{cases} 
        x_t + a_t & \text{if } t \in \{0, 1\} \\
        x_t & \text{if } t = 2
    \end{cases} \\
    r(x_t, t, a_t) &= \begin{cases} 
        -ka_t^2 & \text{if } t \in \{0, 1\} \\
        -x_t^2 & \text{if } t = 2
    \end{cases}
\end{align*}
\]

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 1.1, 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 $\vec{x}$.

3.1 Critic Definition

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

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

where $c_1$ and $c_2$ are real positive constants.

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

\[
\tilde{G}(x_t, t, \vec{w}) = \begin{cases} 
    -2c_1 x_t + w_t & \text{if } t \in \{1, 2\} \\
    0 & \text{if } t \in \{0, 3\}
\end{cases}
\]
We note that this implies
\[
\left( \frac{\partial \tilde{G}}{\partial \vec{w}} \right)_t = \begin{cases} 
1 & \text{if } t \in \{1, 2\} \text{ and } t = k \\
0 & \text{otherwise}
\end{cases}
\] (14)

### 3.2 Unrolling a greedy trajectory

Substituting the model functions (eq. 11) and the critic definition (eq. 12) into the \( \tilde{Q} \) function definition (eq. 2) gives, with \( \gamma = 1 \),
\[
\tilde{Q}(x_t, t, a_t, \vec{w}) = \begin{cases} 
-k(a_0)^2 - c_1(x_0 + a_0)^2 + w_1(x_0 + a_0) + w_3 & \text{if } t = 0 \\
-k(a_1)^2 - c_2(x_1 + a_1)^2 + w_2(x_1 + a_1) + w_4 & \text{if } t = 1
\end{cases}
\]

In order to maximise this with respect to \( a_t \) and get greedy actions, we first differentiate to get,
\[
\left( \frac{\partial \tilde{Q}}{\partial a} \right)_t = \begin{cases} 
-2ka_t - 2c_{t+1}(x_t + a_t) + w_{t+1} & \text{for } t \in \{0, 1\} \\
-2a_t(c_{t+1} + k) + w_{t+1} - 2c_{t+1}x_t & \text{for } t \in \{0, 1\}
\end{cases}
\]

Hence the greedy actions are given by
\[
a_0 \equiv \frac{w_1 - 2c_1x_0}{2(c_1 + k)} \\
a_1 \equiv \frac{w_2 - 2c_2x_1}{2(c_2 + k)}
\] (15) (16)

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
\[
x_1 = a_0 = \frac{w_1}{2(c_1 + k)} \\
\text{and } x_2 = x_1 + a_1 = \frac{w_2(c_1 + k) + kw_1}{2(c_2 + k)(c_1 + k)}
\] (17) (18)

Substituting \( x_1 \) (eq. 17) back into the equation for \( a_1 \) (eq. 16) gives \( a_1 \) purely in terms of the weights and constants.\footnote{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.}
\[
a_1 \equiv \frac{w_2(c_1 + k) - c_2w_1}{2(c_2 + k)(c_1 + k)}
\] (19)
3.3 Evaluation of value-gradients along the greedy trajectory

We can now evaluate the $\tilde{G}$ values by substituting the greedy trajectory’s state vectors (eqs. 17-18) into eq. 13, giving:

$$\tilde{G}_1 = -\frac{c_1w_1}{(c_1+k)} + w_1 = \frac{w_1k}{(c_1+k)} \quad (20)$$

and

$$\tilde{G}_2 = -\frac{w_2(c_1+k)c_2 + kw_1c_2}{(c_2+k)(c_1+k)} + w_2 = \frac{w_2k(c_1+k) - kw_1c_2}{(c_2+k)(c_1+k)} \quad (21)$$

The greedy actions in equations 15 and 16 both satisfy

$$\left(\frac{\partial \pi}{\partial x}\right)_t = \begin{cases} 
-\frac{c_{t+1}+k}{c_{t+1}+k} & \text{for } t \in \{0,1\} \\
0 & \text{otherwise} 
\end{cases} \quad (22)$$

Substituting eqs. 22 and 11 into $Df/Dx = \frac{\partial f}{\partial x} + \frac{\partial \pi}{\partial x} \frac{\partial f}{\partial a}$ gives

$$\left(\frac{Df}{Dx}\right)_t = \begin{cases} 
0 - \frac{c_{t+1}+k}{c_{t+1}+k}(-2ka_t) = \frac{2k(c_{t+1}+k)}{c_{t+1}+k} & \text{if } t \in \{0,1\} \\
-2c_1 & \text{if } t = 2 
\end{cases} \quad (23)$$

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

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

3.4 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 $\tilde{G}_3 = 0$ (by eq. 13) and $G'_3 = 0$ (since $G'_3$ is at a terminal state):

$$G'_2 = \left(\frac{Dr}{Dx}\right)_2 \quad \text{by eq. 8 and } G'_3 = \tilde{G}_3 = 0$$

$$= -2x_2 \quad \text{by eq. 24}$$

$$= -\frac{w_2(c_1+k) + kw_1}{(c_2+k)(c_1+k)} \quad \text{by eq. 18} \quad (25)$$
Similarly,
\[
G'_1 = \left( \frac{Dr}{Dx} \right)_1 + \left( \frac{Df}{Dx} \right)_1 \left( \lambda G'_2 + (1 - \lambda) \tilde{G}_2 \right) \quad \text{by eq. 5}
\]
\[
= \frac{2kc_2a_1}{c_2 + k} + \frac{k}{c_2 + k} \left( \lambda G'_2 + (1 - \lambda) \tilde{G}_2 \right) \quad \text{by eqs. 23, 24}
\]
\[
= \frac{kc_2(w_2(c_1 + k) - c_2w_1)}{(c_1 + k)(c_2 + k)^2}
\]
\[
+ \frac{k}{c_2 + k} \left( -\lambda \frac{w_2(c_1 + k) + kw_1}{(c_2 + k)(c_1 + k)} \right)
\]
\[
+ (1 - \lambda) \frac{w_2k(c_1 + k) - kw_1c_2}{(c_2 + k)(c_1 + k)} \quad \text{by eqs. 19, 21, 25}
\]
\[
= \frac{w_2k(c_2 - \lambda + k(1 - \lambda))}{(c_2 + k)^2}
\]
\[
- \frac{w_1k(k\lambda + (c_2)^2 + k(1 - \lambda)c_2)}{(c_1 + k)(c_2 + k)^2}
\] (26)

4 Divergence Examples for VGL and DHP Algorithms

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

The VGL(\( \lambda \)) weight update (eq. 7) combined with \( \Omega_t = 1 \) gives
\[
\Delta \vec{w}_i = \alpha \sum_t \left( \frac{\partial G'}{\partial \vec{w}_i} \right)_t (G'_t - \tilde{G}_t)
\]
\[
= \alpha (G'_i - \tilde{G}_i) \quad \text{for } i \in \{1, 2\}, \text{ by eq. 14}
\]
\[
\Rightarrow \begin{pmatrix} \Delta w_1 \\ \Delta w_2 \end{pmatrix} = \alpha \begin{pmatrix} G'_1 - \tilde{G}_1 \\ G'_2 - \tilde{G}_2 \end{pmatrix}
\]
\[
= \alpha A \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}
\] (27)

where \( A \) is a 2 \( \times \) 2 matrix with elements which were found by subtracting equations 20 and 21 from equations 26 and 25, respectively, giving,
\[
A_{00} = -\frac{k(k\lambda + (c_2)^2 + k(1 - \lambda)c_2)}{(c_1 + k)(c_2 + k)^2} - \frac{k}{(c_1 + k)}
\]
\[
A_{01} = \frac{k(c_2 + k - \lambda(k + 1))}{(c_2 + k)^2}
\]
\[
A_{10} = \frac{k(c_2 - 1)}{(c_2 + k)(c_1 + k)} \quad A_{11} = \frac{-1 - k}{(c_2 + k)}
\]
Equation 27 is the VGL(\(\lambda\)) weight update written as a single dynamic system of just two variables, i.e. a shortened weight vector, \(\vec{w} = (w_1, w_2)^T\). 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 \(\vec{w} = F\vec{p}\), where \(F\) is a 2 x 2 constant real matrix. The VGL(\(\lambda\)) weight update equation can now be recalculated for these new weights, as follows:

\[
\Delta \vec{p} = \alpha \sum_t \left( \frac{\partial \tilde{G}}{\partial \vec{p}} \right) \left( G'_t - \tilde{G}_t \right)
\]

by eq. 7 and \(\Omega_t = 1\)

\[
= \alpha \sum_t \frac{\partial \vec{w}}{\partial \vec{p}} \left( \frac{\partial \tilde{G}}{\partial \vec{w}} \right) \left( G'_t - \tilde{G}_t \right)
\]

by chain rule

\[
= \alpha \frac{\partial \vec{w}}{\partial \vec{p}} \vec{w}
\]

by eq. 27

\[
= \alpha (F^T AF)\vec{p}.
\]

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. 28, 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 \(\lambda = 0\), with \(c_1 = c_2 = k = 0.01\) gives \(A = \begin{pmatrix} -0.75 & 0.5 \\ -24.75 & -50.5 \end{pmatrix}\).

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. 28 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 \(\Omega_t\) matrices are included according to equation 10.

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 \(F = \begin{pmatrix} -1 & -1 \\ 0.2 & 0.02 \end{pmatrix}\) 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.

The divergence result for VGL(1) does not affect the convergence result by Fairbank & Alonso (2011) which is for VGL(1) but with the special choice of \(\Omega_t\) given by eq. 10. It was not possible to make this algorithm diverge with the methods of this paper.
5 Divergence results for TD(\(\lambda\)) and Sarsa(\(\lambda\)) and HDP

To satisfy the exploration requirement for exploration in TD(\(\lambda\))-based algorithms, we supplemented the greedy policies (eqs. 15 & 16) with a small amount of stochastic Gaussian noise with zero mean. (We had to add this noise, 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 [Fairbank, 2008, 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, in empirical tests. Source code for this is provided. Hence HDP diverges too, since this is equivalent to TD(0) with the given policy.

An insight into why the divergence parameters for VGL were sufficient to make the TD(\(\lambda\)) based algorithms diverge too is because TD with stochastic exploration can be understood to be an approximation to a stochastic version of VGL(\(\lambda\)), so we would expect a divergence example for VGL to cause divergence for TD(\(\lambda\)) 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.

5.1 Divergence results for Sarsa(\(\lambda\))

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

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

\[
\left(\frac{Q^\lambda_t - r_t}{\gamma}\right) = \lambda Q^\lambda_{t+1} + (1 - \lambda)\tilde{Q}_{t+1}
\]

\[
= \lambda Q^\lambda_{t+1} + (1 - \lambda)(r_{t+1} + \gamma\tilde{V}_{t+2})
\]

\[
= r_{t+1} + \lambda(Q^\lambda_{t+1} - r_{t+1}) + (1 - \lambda)(\gamma\tilde{V}_{t+2})
\]

\[
= r_{t+1} + \gamma\left(\lambda\left(\frac{Q^\lambda_{t+1} - r_{t+1}}{\gamma}\right) + (1 - \lambda)\tilde{V}_{t+2}\right)
\]

From this we can see that \(\left(\frac{Q^\lambda_t - r_t}{\gamma}\right)\) obeys the same recursion equation as \(R^\lambda\), 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 2 and 4) that

\[
\left( \frac{Q^\lambda_t - r_t}{\gamma} \right) \equiv R^\lambda_{t+1}
\]

\[
\Rightarrow Q^\lambda_t = r_t + \gamma R^\lambda_{t+1}
\]

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

\[
\Delta \vec{w} = \alpha \sum_t \left( \frac{\partial (r_t + \gamma \tilde{V}(\vec{x}_{t+1}, \vec{w}))}{\partial \vec{w}} \right)_t \left( r_t + \gamma R^\lambda_{t+1} - (r_t + \gamma \tilde{V}_{t+1}) \right)
\]

\[
= \alpha \sum_t \gamma \left( \frac{\partial \tilde{V}}{\partial \vec{w}} \right)_{t+1} \gamma (R^\lambda_{t+1} - \tilde{V}_{t+1})
\]

\[
= \alpha \gamma^2 \sum_{t>0} \left( \frac{\partial \tilde{V}}{\partial \vec{w}} \right)_t (R^\lambda_t - \tilde{V}_t)
\]

which is identical to TD(\(\lambda\)) but with summation over \(t\) now excluding \(t = 0\), and with an extra constant factor, \(\gamma^2\). 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 \(\tilde{Q}\) and problem definition causes divergence for Sarsa(\(\lambda\)) (with both \(\lambda = 1\) and \(\lambda = 0\)).

6 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(1) with \(\Omega_t\) as defined by eq. 10 which is proven to converge by Fairbank & Alonso (2011) and Fairbank (2008) 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 (Baird, 1995; Tsitsiklis & Van Roy, 1996a,b). The divergences we achieved for TD(1) and Sarsa(1) were only possible because of the use of a greedy policy.

It is hoped that these specific examples of divergence of value-iteration will provide a better understanding of how it can happen, and help motivate research to understand and prevent it.

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 that we summarise here:
Sutton et al. (2000) describe conditions under which policy iteration provably converges. These conditions are thought to apply only when the function approximator for $\tilde{V}$ is linear in the same features of the state vector that the function approximator for the policy uses as input (see footnote 1 of Sutton et al. (2000)). Also policy iteration in general has an inner loop of training the value-function to completion, over the whole of state space, for the current fixed policy, which is an extremely computationally intensive process (taking theoretically an infinite time to complete). And this inner loop is combined in an outer loop that, for provable convergence, must train the policy function at a learning rate that tends to zero; so policy-iteration is prohibitively computationally expensive in comparison to value-iteration.

References

Baird, Leemon C. Residual algorithms: Reinforcement learning with function approximation. In International Conference on Machine Learning, pp. 30–37, 1995.

Bellman, R. Dynamic Programming. Princeton University Press, Princeton, NJ, USA, 1957.

Fairbank, Michael. Reinforcement learning by value gradients. eprint arXiv:0803.3539, 2008.

Fairbank, Michael and Alonso, Eduardo. The local optimality of reinforcement learning by value gradients and its relationship to policy gradient learning. eprint arXiv:1101.0428, 2011.

Ferrari, Silvia and Stengel, Robert F. Model-based adaptive critic designs. Handbook of learning and approximate dynamic programming, editors Jennie Si et al., pp. 65–96, 2004.

Prokhorov, Danil and Wunsch, Don. Adaptive critic designs. IEEE Transactions on Neural Networks, September:997–1007, 1997.

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

Sutton, Richard S. Learning to predict by the methods of temporal differences. Machine Learning, 3:9–44, 1988.

Sutton, Richard S. and Barto, Andrew G. Reinforcement Learning: An Introduction. The MIT Press, Cambridge, Massachusetts, USA, 1998.

Sutton, Richard S., Mcallester, David, Singh, Satinder, and Mansour, Yishay. Policy gradient methods for reinforcement learning with function approximation. In Advances in Neural Information Processing Systems 12, volume 12, pp. 1057–1063, 2000.
Tsitsiklis, John N. and Van Roy, Benjamin. An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, 1996a.

Tsitsiklis, John N. and Van Roy, Benjamin. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22(1-3):59–94, 1996b.

Wang, Fei-Yue, Zhange, Huaguang, and Liu, Derong. Adaptive dynamic programming: An introduction. *IEEE Computational Intelligence Magazine*, pp. 39–47, 2009.

Watkins, C. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, England, 1989.

Werbos, Paul J. *Handbook of Intelligent Control, editors White and Sofge, Chapter 13*. Van Nostrand, 1992.

Werbos, Paul J. Stable adaptive control using new critic designs. *eprint arXiv:adap-org/9810001*, 1998.