City Research Online

City, University of London Institutional Repository

Citation: Fairbank, M., Prokhorov, D. & Alonso, E. (2013). Approximating Optimal Control with Value Gradient Learning. In: Lewis, F. & Liu, D. (Eds.), Reinforcement Learning and Approximate Dynamic Programming for Feedback Control. (pp. 142-161). Hoboken, NJ, USA: Wiley-IEEE Press. ISBN 111810420X doi: 10.1002/9781118453988

This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/5192/

Link to published version: https://doi.org/10.1002/9781118453988

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

APPROXIMATING OPTIMAL CONTROL WITH VALUE GRADIENT LEARNING

Michael Fairbank[1], Danil Prokhorov[2] and Eduardo Alonso[1]

[1]City University London, London, UK

[2]Toyota Research Institute NA, Ann Arbor, Michigan

Abstract

In this chapter we extend the ADP algorithm, Dual Heuristic Programming (DHP), to include a "bootstrapping" parameter λ, analogous to that used in the Reinforcement Learning algorithm TD(λ). The resulting algorithm, which we call VGL(λ) for value-gradient learning, is proven to produce a weight update that can be equivalent to backpropagation through time (BPTT) applied to a greedy policy on a critic-function. This provides a surprising connection between the two alternative methods of BPTT and DHP. Under certain smoothness conditions, VGL($\lambda = 1$) with a greedy policy acquires strong convergence conditions of BPTT, while using a general function

approximator for the critic. We show that this can lead to increased stability in the learning of control problems by a neural network.

7.1 INTRODUCTION

Adaptive Dynamic Programming (ADP) is the study of how an agent can learn actions that maximise a long-term reward [13]. For example a typical 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 $\vec{a}_t$ which takes it to the next state according to the environment's (possible stochastic) model function $\vec{x}_{t+1} = f(\vec{x}_t, \vec{a}_t)$, and gives it an immediate scalar reward r_t, given by the reward function $r_t = r(\vec{x}_t, \vec{a}_t)$. 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. The ADP problem is for the agent to learn how to choose actions so as to maximise the expectation of the total reward received, $\langle \sum_t \gamma^t r_t \rangle$, where $\gamma \in [0, 1]$ is a constant *discount factor* that specifies the importance of long term rewards over short term ones. Specifically, the problem is to find a policy function $\pi(\vec{x}, \vec{z})$, where $\vec{z}$ is the parameter vector of a function approximator, that calculates which action $\vec{a} = \pi(\vec{x}, \vec{z})$ to take for any given state $\vec{x}$, such that this total reward sum is maximised.

In this chapter we concentrate on the sub-problem of when the model functions $f(\vec{x}, \vec{a})$ and $r(\vec{x}, \vec{a})$ are comprised of a twice-differentiable deterministic part, plus optionally some additive noise. We assume that these functions are unchanging and either already known analytically, or already learned beforehand by a system identification process, for example by using a neural network as described by [17].

One paradigm of ADP is to learn a *value function*, $V(\vec{x})$, from Bellman's Optimality Principle [1], and then choose a policy function that is "greedy" on that value function. A greedy policy is one that always chooses actions which lead to states with the highest V value (whilst also taking into account the immediate short-term reward in getting there). These methods use an approximated function (e.g. the output of a neural network) to represent the learned V function, and this approximated function is called the *critic*. We call this paradigm *critic-learning*. These critic-learning methods include Heuristic Dual Programming, Dual Heuristic Programming (DHP) and Globalized Dual Heuristic Programming (GDHP) from the ADP literature [13, 16, 9], and TD(λ) and Q-learning from the reinforcement learning (RL) literature [11, 14].

Critic learning methods can be very effective and computationally efficient, but proving their convergence when a general function approximator is used for V is challenging, and even more challenging when combined with a greedy policy.

One reason that convergence analysis is difficult with a greedy policy is that in the Bellman condition, V depends on the policy π, which, being greedy, depends on V; so making progress in learning one of them can undo progress in learning the other. We make an important insight into this difficulty by showing (in Lemma 4) that the dependency of a greedy policy on a value function is through what we call the *value-gradient*, which we define to be $\frac{\partial V}{\partial \vec{x}}$. Hence a value-gradient analysis seems necessary at some level to provide a theoretical gateway to analysing the convergence properties of any critic weight update that uses a greedy policy in a continuous state space. We refer to algorithms that specifically aim to learn the value-gradient $\frac{\partial V}{\partial \vec{x}}$ as "value-gradient learning" (VGL) algorithms. The existing ADP algorithms DHP and GDHP are examples of value-gradient learning algorithms.

We extend the DHP and GDHP algorithms into a new algorithm that we call VGL(λ). This extension algorithm includes a constant parameter $\lambda \in [0, 1]$ analogous to that used in TD(λ), such that VGL(0) is equivalent to DHP. The motivation for doing this extension is similar for the motivation for using TD(λ) in preference to TD(0): By choosing λ carefully we might get faster and more stable learning.

An alternative paradigm of ADP is to use backpropagation through time (BPTT, [15]), which is gradient ascent on the total long term reward with respect to the policy's parameter vector $\vec{z}$. Since this is gradient ascent on a function that is bound above, proving convergence is more straightforward.

In this chapter we demonstrate a theoretical connection between these two paradigms of ADP (i.e. between critic learning and BPTT). We prove that, under certain strict conditions, doing BPTT on a greedy policy function is equivalent to VGL(1) with a carefully chosen learning parameter, "Ω_t", described below. This proof of equivalence is interesting because it provides a way to ensure the same convergence guarantees of BPTT will also apply to this critic learning method, with a greedy policy, and a general function approximator for V. Hence we provide a convergence proof for this instance of the algorithm in this chapter.

The convergence proof requires a particular choice of a learning parameter matrix, Ω_t. This was defined by Werbos for the algorithm GDHP (e.g. see [18, eq. 32]). Previously there was little guidance on how to set this matrix, but our proof and discussions (in sections 7.3.4 and 7.4.3) cast some new light on this problem.

Other convergence proofs do exist for critic learning methods, but they do not generally apply to a greedy policy or a critic function implemented by a general function approximator. For example the RL algorithm TD(λ) [11] has been proven to converge [12] provided the function approximator for V is linear in its parameter vector, and the policy is fixed (i.e. that excludes the greedy policy). This particular algorithm is not proven to converge when a general function approximator is used to represent the critic (e.g. a neural network) or when a greedy policy is used. Divergence examples exist for a non-linear function approximator [12], and where the function approximator for V is linear in its weight vector but where a greedy policy is used (diverging for both $\lambda = 0$ and $\lambda = 1$; see [6]). A convergence proof for general critic learning was given by [8], although this assumed the critic function being used could always be made to exactly learn any given target values, and so this is not realistic for a general function approximator. In the case of a general function approximator and a greedy policy, popular ADP methods such as DHP and GDHP can also be forced to diverge [6].

The structure of this chapter is as follows: We define the VGL(λ) and BPTT algorithms in section 7.2. We prove their equivalence in section 7.3, and discuss the conditions for convergence in section 7.3.3, and insights into the Ω_t matrix in sections 7.3.4 and 7.4.3. In section 7.4 we define a simple computer experiment and demonstrate the improved convergence properties of the VGL algorithm over DHP. We also define an efficient form for the greedy policy and Ω_t matrix in section 7.4.2. Finally, in section 7.5, we give conclusions.

7.2 VALUE GRADIENT LEARNING AND BPTT ALGORITHMS

First we give some preliminary definitions and notation, and then follow with the algorithm definitions.

7.2.1 Preliminary Definitions

Critic function: We define $\widetilde{V}(\vec{x},\vec{w})$ to be the scalar output of a smooth function approximator with weight vector $\vec{w}$. This can also be known as the *approximate value function*.

Critic gradient function: We define the critic gradient function to be $\widetilde{G}(\vec{x},\vec{w}) \equiv \frac{\partial \widetilde{V}(\vec{x},\vec{w})}{\partial \vec{x}}$. The VGL and DHP algorithms attempt to learn this quantity. The critic gradient can also be known as the *approximate value gradient*.

Smooth policy function: We define a general smooth policy function $\pi(\vec{x},\vec{z})$ to be the output of a smooth function approximator with output vector of dimension $\dim(\vec{a})$, and with parameter vector $\vec{z}$. This is also referred to in the literature as an "action network" or "actor".

Approximate Q function: We define the function $\widetilde{Q}(\vec{x},\vec{a},\vec{w})$ to be

$$\widetilde{Q}(\vec{x},\vec{a},\vec{w}) = r(\vec{x},\vec{a}) + \gamma \widetilde{V}(f(\vec{x},\vec{a}),\vec{w}) \tag{7.1}$$

This function comes up often in this chapter in the greedy policy and its related lemmas. It is related to, but not identical to, the Q-function defined by [14].

Matrix and vector notation used: We make a notational convention that all vectors are columns, and differentiation of a scalar by a vector gives a column vector. So for example $\widetilde{G}$, $\vec{x}$ and $\frac{\partial \widetilde{V}}{\partial \vec{x}}$ are all columns. We define differentiation of a vector function by a vector as follows:[1] $\frac{\partial \widetilde{G}}{\partial \vec{w}}$ is a matrix with element (i,j) equal to $\frac{\partial \widetilde{G}(\vec{x},\vec{w})^j}{\partial \vec{w}^i}$. Similarly, $\frac{\partial f}{\partial \vec{x}}$ is the matrix with element $\left(\frac{\partial f}{\partial \vec{x}}\right)^{ij} = \frac{\partial f(\vec{x},\vec{a})^j}{\partial \vec{x}^i}$.

Trajectory Shorthand Notation: All subscripted "*t*" indices 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 $\widetilde{V}_{t+1} \equiv \widetilde{V}(\vec{x}_{t+1},\vec{w})$, $r_t \equiv r(\vec{x}_t,\vec{a}_t)$, $\widetilde{G}_t \equiv \widetilde{G}(\vec{x}_t,\vec{w})$ and $\left(\frac{\partial \widetilde{Q}}{\partial \vec{a}}\right)_t$

[1] This is the transpose of a common convention.

is shorthand for the function $\frac{\partial \widetilde{Q}(\vec{x},\vec{a},\vec{w})}{\partial \vec{a}}$ evaluated at $(\vec{x}_t, \vec{a}_t, \vec{w})$. Similarly $\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \equiv \left.\frac{\partial \widetilde{G}}{\partial \vec{w}}\right|_{(\vec{x}_t,\vec{w})}$.

7.2.2 VGL(λ) Algorithm

The VGL algorithm is an extension of the DHP algorithm [16] to include a constant parameter λ analogous to that used in the algorithm TD(λ). The algorithm requires the derivatives of $f(\vec{x},\vec{a})$, $r(\vec{x},\vec{a})$, $\pi(\vec{x},\vec{z})$ and $\widetilde{V}(\vec{x},\vec{w})$ all to exist at every time-step along a trajectory.

Using the notation conventions of section 7.2.1, and the implied matrix products, the VGL(λ) algorithm is defined to be a critic weight update of the form:

$$\Delta\vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t (G'_t - \widetilde{G}_t) \tag{7.2}$$

where α is a small positive constant; $\Omega_t \in \Re^{\dim(\vec{x})\times\dim(\vec{x})}$ is an arbitrary positive definite matrix described further below; $\widetilde{G}_t$ is the critic gradient; and G'_t is the "target value gradient" defined recursively by:

$$G'_t = \left(\frac{Dr}{D\vec{x}}\right)_t + \gamma \left(\frac{Df}{D\vec{x}}\right)_t \left(\lambda G'_{t+1} + (1-\lambda)\widetilde{G}_{t+1}\right) \tag{7.3}$$

with $G'_t = \vec{0}$ at any terminal state, and where $\lambda \in [0,1]$ is a fixed constant; and where $\frac{D}{D\vec{x}}$ is shorthand for

$$\frac{D}{D\vec{x}} \equiv \frac{\partial}{\partial \vec{x}} + \frac{\partial \pi}{\partial \vec{x}} \frac{\partial}{\partial \vec{a}} ; \tag{7.4}$$

and where all of these derivatives are assumed to exist. We ensure the recursion in equation 7.3 converges by requiring that either $\gamma\lambda < 1$, or the environment is such that the agent is guaranteed to reach a terminal state at some finite time (i.e. the environment is "episodic").

Equations 7.2, 7.3 and 7.4 define the VGL(λ) algorithm. These equations can be implemented by unrolling a whole trajectory and then working backwards along it applying the recursion of equation 7.3, as demonstrated in pseudocode of Algorithm 7.1, which runs in time $O(\dim(\vec{w}))$ per trajectory step. Alternatively, it is possible to apply the weight update in an on-line manner as described by [7, Appendix B], but this runs in a longer time of $O(\dim(\vec{w})\dim(\vec{x})^2)$ per trajectory step.

When we choose the parameter $\lambda = 0$, then we get the VGL(0) algorithm which is equivalent to DHP. Like DHP, the objective of the VGL(λ) weight update is to make the values $\widetilde{G}_t$ move towards the target values ${G'}_t$.

Ω_t was introduced by [18, eq. 32] for the GDHP algorithm, and is included in our weight update for full generality. This positive definite matrix can be set as required by the experimenter since its presence ensures every component of $\widetilde{G}_t$ will move towards the corresponding component of ${G'}_t$ (in any basis). However choosing what value to use for Ω_t is not obvious, so it is often just taken to be the identity matrix for all t. We make an insight into how to choose Ω_t by finding an explicit formula for it in section 7.3.2.

Algorithm 7.1

```
VGL($\lambda$) algorithm {
  $t \leftarrow 0$, $\Delta\vec{w} \leftarrow \vec{0}$
  while not terminated($\vec{x}_t$)
      $\vec{a}_t \leftarrow \pi(\vec{x}_t, \vec{z})$
      $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{a}_t)$
      $t \leftarrow t + 1$
  end while
  $F \leftarrow t$
  $\vec{p} \leftarrow \vec{0}$
  for $t = F - 1$ to 0 step $-1$
      ${G'}_t \leftarrow \left(\frac{Dr}{D\vec{x}}\right)_t + \gamma \left(\frac{Df}{D\vec{x}}\right)_t \vec{p}$
      $\Delta\vec{w} \leftarrow \Delta\vec{w} + \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left({G'}_t - \widetilde{G}_t\right)$
      $\vec{p} \leftarrow \lambda {G'}_t + (1 - \lambda)\widetilde{G}_t$
  end for
  $\vec{w} \leftarrow \vec{w} + \alpha\Delta\vec{w}$
}
```

The policy function $\pi(\vec{x}, \vec{z})$ needs training concurrently with the VGL algorithm. This could be achieved by the same weight update as used by the DHP algorithm's policy function weight update, i.e.

$$\Delta\vec{z} = \beta \sum_t \left(\frac{\partial \pi}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial r}{\partial \vec{a}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{a}}\right)_t \widetilde{G}_{t+1}\right). \tag{7.5}$$

where β is a separate learning rate for the policy function.

There are a variety of schemes to do this concurrently with the critic training. For example, the policy could be trained to completion in between every critic function weight update (a process known as value-iteration), or the vice-versa arrangement could be applied (a process known as policy-iteration). Or a simple concurrent scheme of doing alternating iterations of each could be applied. A final option is to use a greedy policy function, as described in section 7.3.1, which obviates the need for training the policy, and can be viewed as an extreme form of value-iteration.

In Algorithm 7.1, the critic function $\widetilde{G}(\vec{x},\vec{w})$ can be interpreted as *either* a synonym for $\frac{\partial \widetilde{V}(\vec{x},\vec{w})}{\partial \vec{x}}$, which implies that $\frac{\partial \widetilde{G}}{\partial \vec{w}} \equiv \frac{\partial^2 \widetilde{V}}{\partial \vec{w} \partial \vec{x}}$, or alternatively $\widetilde{G}(\vec{x},\vec{w})$ could be implemented more simply as the output of a smooth vector function approximator of dimension $\dim(\vec{x})$. We call the first of these two options a "GDHP-style critic" and the second a "DHP-style critic". A GDHP-style critic is the harder of the two options to implement, since it requires second order backpropagation to find $\frac{\partial \widetilde{G}}{\partial \vec{w}}$, while a DHP-style critic only requires first-order backpropagation for this quantity. In the experiments of section 7.4.4 we use a DHP-style critic, for simplicity.

7.2.3 BPTT Algorithm

We define $R(\vec{x}_0,\vec{z})$ to be the total discounted reward encountered by an agent starting at state $\vec{x}_0$ and then following a policy $\pi(\vec{x},\vec{z})$ until termination, so that $R(\vec{x}_0,\vec{z}) = \sum_t \gamma^t r_t$. This function can be witten recursively as:

$$R(\vec{x},\vec{z}) = r(\vec{x},\pi(\vec{x},\vec{z})) + \gamma R(f(\vec{x},\pi(\vec{x},\vec{z})),\vec{z}) \tag{7.6}$$

with $R(\vec{x},\vec{z}) = 0$ at any terminal state.

BPTT is gradient ascent on $R(\vec{x}_0,\vec{z})$ with respect to $\vec{z}$, i.e. $\Delta\vec{z} = \beta\left(\frac{\partial R}{\partial \vec{z}}\right)_0$ for some small positive constant β. Expanding the term $\left(\frac{\partial R}{\partial \vec{z}}\right)_t$ gives:

$$\left(\frac{\partial R}{\partial \vec{z}}\right)_t = \left(\frac{\partial}{\partial \vec{z}}(r(\vec{x},\pi(\vec{x},\vec{z})) + \gamma R(f(\vec{x},\pi(\vec{x},\vec{z})),\vec{z}))\right)_t \qquad \text{by eq. 7.6}$$

$$= \left(\frac{\partial \pi}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial r}{\partial \vec{a}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{a}}\right)_t \left(\frac{\partial R}{\partial \vec{x}}\right)_{t+1}\right) + \gamma \left(\frac{\partial R}{\partial \vec{z}}\right)_{t+1}$$

where we used the chain rule, trajectory shorthand notation, $\vec{a}_t = \pi(\vec{x}_t, \vec{z})$ and $\vec{x}_{t+1} = f(\vec{x}_t, \vec{a}_t)$. Expanding this recursion gives:

$$\left(\frac{\partial R}{\partial \vec{z}}\right)_0 = \sum_{t \geq 0} \gamma^t \left(\frac{\partial \pi}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial r}{\partial \vec{a}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{a}}\right)_t \left(\frac{\partial R}{\partial \vec{x}}\right)_{t+1}\right)$$

It is common practice to drop the γ^t factor in this equation. Combining this with the gradient ascent equation, $\Delta \vec{z} = \beta \left(\frac{\partial R}{\partial \vec{z}}\right)_0$, gives the BPTT weight update:

$$\Delta \vec{z} = \beta \sum_{t \geq 0} \left(\frac{\partial \pi}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial r}{\partial \vec{a}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{a}}\right)_t \left(\frac{\partial R}{\partial \vec{x}}\right)_{t+1}\right) \tag{7.7}$$

This equation is the BPTT weight update. It refers to the quantity $\frac{\partial R}{\partial \vec{x}}$ which can be found recursively by differentiating equation 7.6 and using the chain rule, giving

$$\left(\frac{\partial R}{\partial \vec{x}}\right)_t = \left(\frac{Dr}{D\vec{x}}\right)_t + \gamma \left(\frac{Df}{D\vec{x}}\right)_t \left(\frac{\partial R}{\partial \vec{x}}\right)_{t+1} \tag{7.8}$$

with $\frac{\partial R}{\partial \vec{x}} = \vec{0}$ at any terminal state.

Equation 7.8 can be understood to be backpropagating the quantity $\left(\frac{\partial R}{\partial \vec{x}}\right)_{t+1}$ through the actor network, model and reward functions to obtain $\left(\frac{\partial R}{\partial \vec{x}}\right)_t$, and giving the algorithm its name.

7.3 A CONVERGENCE PROOF FOR VGL(1) FOR CONTROL WITH FUNCTION APPROXIMATION

In this section we define an equivalence proof between VGL(1) and BPTT applied to a greedy policy. Since BPTT is gradient ascent on a function that is bound above, it has relatively good convergence guarantees, and hence the equivalence proof can be used to make a convergence guarantee for VGL(1).

First we describe the greedy policy with some useful lemmas, then we prove the equivalence of BPTT to VGL(1), and then discuss the convergence conditions. For the equivalence and convergence to hold we require a specifically chosen time-dependent Ω_t matrix, which is discussed in section 7.3.4.

7.3.1 Using a Greedy Policy with a Critic function

The greedy policy is defined to choose actions as follows:

$$\vec{a} = \arg\max_{\vec{a} \in \Re^n} (\widetilde{Q}(\vec{x}, \vec{a}, \vec{w})) \quad \forall \vec{x} \tag{7.9}$$

where $\widetilde{Q}(\vec{x}, \vec{a}, \vec{w})$ is defined in equation 7.1.

For VGL, we required all of the constituent functions of equation 7.1 to be smooth, therefore under this requirement, $\widetilde{Q}$ is a smooth function with respect to all of its parameters.

The greedy policy depends on $\widetilde{V}(\vec{x}, \vec{w})$, i.e. uses the weight vector $\vec{w}$ instead of $\vec{z}$. This means the greedy policy is a function $\pi(\vec{x}, \vec{w})$ (instead of the usual policy dependency $\pi(\vec{x}, \vec{z})$). Hence from now on, when considering the greedy policy, it is the *same* weight vector $\vec{w}$ that controls the policy as is used for the critic function, and so we will write $\pi(\vec{x}, \vec{w})$ to always specifically mean the *greedy* policy. Any change to the critic function will immediately affect the greedy policy and move trajectories, and we need to take this into account when proving convergence.

Greedy Actions: A greedy action is one that satisfies equation 7.9.

Since a greedy action selects a *maximum* with respect to $\vec{a}$ of the smooth function $\widetilde{Q}(\vec{x}, \vec{a}, \vec{w})$, the following two consequences hold:

Lemma 1 *For a greedy action $\vec{a}$, $\frac{\partial \widetilde{Q}}{\partial \vec{a}} = \vec{0}$.*

Lemma 2 *For a greedy action $\vec{a}$, $\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}$ is a negative semi-definite matrix.*

Furthermore, we prove three less obvious lemmas about greedy actions and a greedy policy:

Lemma 3 *The greedy policy implies $\left(\frac{\partial r}{\partial \vec{a}}\right)_t = -\gamma \left(\frac{\partial f}{\partial \vec{a}}\right)_t \widetilde{G}_{t+1}$.*

Proof: First, we note that differentiating equation 7.1 gives

$$\left(\frac{\partial \widetilde{Q}}{\partial \vec{a}}\right)_t = \left(\frac{\partial r}{\partial \vec{a}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{a}}\right)_t \widetilde{G}_{t+1} \tag{7.10}$$

Substituting this into Lemma 1 and solving for $\frac{\partial r}{\partial \vec{a}}$ completes the proof. ■

Lemma 4 *When* $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ *and* $\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t^{-1}$ *exist for an action* $\vec{a}_t$*, the greedy policy implies*

$$\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t = -\gamma \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_{t+1} \left(\frac{\partial f}{\partial \vec{a}}\right)_t^T \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t^{-1}$$

Proof: We use implicit differentiation. The dependency of $\vec{a}_t = \pi(\vec{x}_t, \vec{w})$ on $\vec{w}$ must be such that Lemma 1 is always satisfied, since the policy is greedy. This means that $\left(\frac{\partial \widetilde{Q}}{\partial \vec{a}}\right)_t \equiv \vec{0}$, both before and after any infinitesimal change to $\vec{w}$. Therefore the function $\pi(\vec{x}_t, \vec{w})$ must be such that,

$$\begin{aligned}
\vec{0} =& \frac{\partial}{\partial \vec{w}} \left(\frac{\partial \widetilde{Q}(\vec{x}_t, \pi(\vec{x}_t, \vec{w}), \vec{w})}{\partial \vec{a}_t} \right) \\
=& \frac{\partial}{\partial \vec{w}} \left(\frac{\partial \widetilde{Q}(\vec{x}_t, \vec{a}_t, \vec{w})}{\partial \vec{a}_t} \right) + \left(\frac{\partial \pi}{\partial \vec{w}}\right)_t \frac{\partial}{\partial \vec{a}_t} \left(\frac{\partial \widetilde{Q}(\vec{x}_t, \vec{a}_t, \vec{w})}{\partial \vec{a}_t} \right) \\
=& \frac{\partial}{\partial \vec{w}} \left(\left(\frac{\partial r}{\partial \vec{a}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{a}}\right)_t \widetilde{G}_{t+1} \right) + \left(\frac{\partial \pi}{\partial \vec{w}}\right)_t \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t \\
=& \frac{\partial}{\partial \vec{w}} \left(\left(\frac{\partial r}{\partial \vec{a}}\right)_t + \gamma \sum_i \left(\frac{\partial (f)^i}{\partial \vec{a}}\right)_t (\widetilde{G}_{t+1})^i \right) + \left(\frac{\partial \pi}{\partial \vec{w}}\right)_t \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t \\
=& \gamma \sum_i \left(\frac{\partial (f)^i}{\partial \vec{a}}\right)_t \frac{\partial (\widetilde{G}_{t+1})^i}{\partial \vec{w}} + \left(\frac{\partial \pi}{\partial \vec{w}}\right)_t \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t \\
=& \gamma \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_{t+1} \left(\frac{\partial f}{\partial \vec{a}}\right)_t^T + \left(\frac{\partial \pi}{\partial \vec{w}}\right)_t \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t
\end{aligned}$$

In the above six lines of algebra, line 2 is by the chain rule and substitution of $\vec{a}_t = \pi(\vec{x}_t, \vec{w})$; line 3 is by equation 7.10; line 4 just expands an inner product; line 5 follows since $\frac{\partial r}{\partial \vec{a}}$ and $\frac{\partial f}{\partial \vec{a}}$ are not functions of $\vec{w}$; and line 6 just forms an inner product.

Then solving the final line for $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ proves the lemma. ■

Lemma 5 *When* $\left(\frac{\partial \pi}{\partial \vec{x}}\right)_t$ *and* $\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t^{-1}$ *exist for an action* $\vec{a}_t$*, the greedy policy implies*

$$\left(\frac{\partial \pi}{\partial \vec{x}}\right)_t = -\gamma \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{x} \partial \vec{a}}\right)_t \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t^{-1}$$

This lemma is useful because it provides the quantity $\left(\frac{\partial\pi}{\partial\vec{x}}\right)_t$ which is used in the VGL(λ) algorithm definition, in equation 7.4. This enables us to use the VGL(λ) algorithm with a *greedy* policy.

Proof: The proof is virtually the same as that of Lemma 4, but with the start point changed to $\vec{0} = \frac{\partial}{\partial\vec{x}}\left(\frac{\partial\widetilde{Q}(\vec{x}_t,\pi(\vec{x}_t,\vec{w}),\vec{w})}{\partial\vec{a}_t}\right)$. ■

7.3.2 The Equivalence of VGL(1) to BPTT

BPTT can be defined on *any* smooth policy function $\pi(\vec{x},\vec{z})$. A greedy policy, $\pi(\vec{x},\vec{w})$, can be defined on *any* critic function $\widetilde{G}(\vec{x},\vec{w})$. In this section we apply BPTT to the *greedy* policy function $\pi(\vec{x},\vec{w})$, and we observe the resulting combined weight update that emerges. Surprisingly, we find that this weight update is identical to the VGL(1) weight update, provided the Ω_t matrix is chosen carefully. This proves that the VGL(λ) weight update of equation 7.2, with $\lambda = 1$ and a carefully chosen Ω_t matrix, is equivalent to BPTT on a greedy policy.

First we note that by comparing equations 7.3 and 7.8, we see that

$$G'_t \equiv \left(\frac{\partial R}{\partial\vec{x}}\right)_t \qquad \text{when } \lambda = 1 \tag{7.11}$$

The BPTT gradient ascent weight update, following on from equation 7.7, but now using a greedy policy $\pi(\vec{x},\vec{w})$ instead of a general policy function $\pi(\vec{x},\vec{z})$, is

$$\begin{aligned}
\Delta\vec{w} =& \beta\sum_{t\geq 0}\left(\frac{\partial\pi}{\partial\vec{w}}\right)_t\left(\left(\frac{\partial r}{\partial\vec{a}}\right)_t + \gamma\left(\frac{\partial f}{\partial\vec{a}}\right)_t\left(\frac{\partial R}{\partial\vec{x}}\right)_{t+1}\right) && \text{(eq. 7.7)}\\
=& \beta\sum_{t\geq 0}\left(\frac{\partial\pi}{\partial\vec{w}}\right)_t\gamma\left(\frac{\partial f}{\partial\vec{a}}\right)_t\left(-\widetilde{G}_{t+1} + \left(\frac{\partial R}{\partial\vec{x}}\right)_{t+1}\right) && \text{by Lemma 3}\\
=& \beta\sum_{t\geq 0}\gamma\left(\frac{\partial\pi}{\partial\vec{w}}\right)_t\left(\frac{\partial f}{\partial\vec{a}}\right)_t\left(G'_{t+1} - \widetilde{G}_{t+1}\right) && \text{by eq. 7.11}\\
=& \beta\sum_{t\geq 0}-\gamma^2\left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_{t+1}\left(\frac{\partial f}{\partial\vec{a}}\right)_t^T\left(\frac{\partial^2\widetilde{Q}}{\partial\vec{a}\partial\vec{a}}\right)_t^{-1}\left(\frac{\partial f}{\partial\vec{a}}\right)_t(G'_{t+1} - \widetilde{G}_{t+1}) && \text{by Lemma 4}\\
=& \beta\sum_{t\geq 0}\gamma^2\left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_t\Omega_t(G'_t - \widetilde{G}_t) && (7.12)
\end{aligned}$$

where

$$\Omega_t = \begin{cases} -\left(\frac{\partial f}{\partial \vec{a}}\right)^T_{t-1} \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)^{-1}_{t-1} \left(\frac{\partial f}{\partial \vec{a}}\right)_{t-1} & \text{if } t > 0 \\ 0 & \text{if } t = 0 \end{cases}, \tag{7.13}$$

and is positive semi-definite, by the greedy policy (Lemma 2).

Equation 7.12 is identical to a VGL weight update equation (eq. 7.2), with a carefully chosen matrix for Ω_t, and $\lambda = 1$, provided $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ and $\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)^{-1}_t$ exist for all t. If $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ does not exist, then $\frac{\partial R}{\partial \vec{w}}$ is not defined either.

This completes the demonstration of the equivalence of a critic learning algorithm (VGL(1), with the conditions stated above) to BPTT on a greedy policy $\pi(\vec{x}, \vec{w})$ (where $\vec{w}$ is the weight vector of a critic function $\widetilde{G}(\vec{x}, \vec{w})$ defined for Algorithm 7.1), when $\frac{\partial R}{\partial \vec{w}}$ exists.

7.3.3 Convergence conditions

Good convergence conditions exist for BPTT since it is gradient ascent on a function that is bound above, and therefore convergence is guaranteed if that surface is smooth and the learning step size is sufficiently small. If the ADP problem is such that $\frac{\partial \pi}{\partial \vec{w}}$ always exists, and we choose Ω_t by equation 7.13, then the above equivalence proof shows that the good convergence guarantees of BPTT will apply to VGL(1). Significantly $\frac{\partial \pi}{\partial \vec{w}}$ always does exist in a continuous time setting when a value-gradient policy using the technique of section 7.4.2 is used.

In addition to smoothness of the policy, we also require smoothness of the functions, r and f, for VGL to be defined; and also for the convergence of BPTT that we have proved equivalence to, the weight vector for the policy must only traverse smooth regions of the surface of $R(\vec{x}, \vec{w})$.

If all of these conditions are satisfied, then this approximated-critic value-iteration scheme will converge.

This has been a non-trivial accomplishment in proving convergence for a smoothly approximated critic function with a greedy policy, even though it is only proven for $\lambda = 1$. Other related algorithms with $\lambda = 1$, such as TD(1) and Sarsa(1), and VGL(1) *without* the specially chosen Ω_t matrix, can all be made to diverge under the same conditions when a greedy policy is used [6]. Algorithms with $\lambda = 0$, such as TD(0), Sarsa(0), DHP and GDHP are also shown to diverge with a greedy policy by [6].

While the smoothness of all functions is required for provable convergence, in practice a sufficient condition appears to be piece-wise continuity, as BPTT has been applied successfully to systems with friction and dead-zones.

7.3.4 Notes on the Ω_t matrix

The Ω_t matrix that we derived in equation 7.13 differs from the previous instances of its use in the literature (e.g. [18, eq. 32]):

- Firstly, our Ω_t matrix is time dependent, whereas previous usages of it have not used a t subscript.

- Secondly, we have found an exact equation on how to choose it (i.e. equation 7.13). Previous guidance on how to choose it has been only intuitive.

- Thirdly, equation 7.13 often only produces a positive *indefinite* matrix which is problematic for the case of $\lambda < 1$. If we have $\dim(\vec{x}) > \dim(\vec{a})$ then the matrix $\frac{\partial f}{\partial \vec{a}}$ will be wider than it is tall, and so the matrix product in equation 7.13 will yield an Ω_t matrix that is rank deficient (i.e. positive indefinite). It seems that it is not a problem to have a rank-deficient Ω_t matrix when $\lambda = 1$ (as section 7.3.2 effectively proves), but it is a problem when $\lambda < 1$. A rank-deficient Ω_t matrix will have some zero eigenvalues, and the components of $\widetilde{G}$ corresponding to these missing eigenvalues will not be learned at all by equation 7.2. However in the case of $\lambda < 1$, the definition of G'_t in equation 7.3 depends upon potentially *all* of the components of $\widetilde{G}_{t+1}$ via the multiplication in equation 7.3 by $\left(\frac{Df}{D\vec{x}}\right)_t$. So if some of the components of $\widetilde{G}_{t+1}$ are missing, then the target gradients G'_t will be wrong, and so the VGL(λ) algorithm will be badly defined.

  This view that it is necessary for Ω_t to be full-rank for $\lambda < 1$ is consistent with the original positive-definite requirement made by Werbos for GDHP, which is a $\lambda = 0$ algorithm [18].

Consequently, our choice of Ω_t matrix is best used for the situation of $\lambda = 1$ and a greedy policy. But we feel it may provide some guidance in how to choose Ω_t in other situations, especially if working in a problem where $\dim(\vec{x}) \leq \dim(\vec{a})$. And even if the policy is not greedy, then equation 7.13 might still be a useful *guiding* choice for Ω_t, since it is the objective of the training algorithm for the actor network to always try to make the policy greedy.

The Ω_t matrix definition in equation 7.13 requires an inverse of the following rather cumbersome looking matrix:

$$\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t = \frac{\partial}{\partial \vec{a}} \left(\left(\frac{\partial r}{\partial \vec{a}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{a}}\right)_t \widetilde{G}_{t+1} \right) \qquad \text{by eq. 7.10}$$

$$\Rightarrow \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a}^i \partial \vec{a}^j}\right)_t = \left(\frac{\partial^2 r}{\partial \vec{a}^i \partial \vec{a}^j}\right)_t + \gamma \left(\frac{\partial^2 f}{\partial \vec{a}^i \partial \vec{a}^j}\right)_t \widetilde{G}_{t+1} + \gamma \left(\frac{\partial f}{\partial \vec{a}^i}\right)_t \left(\frac{\partial \widetilde{G}}{\partial \vec{x}}\right)_{t+1} \left(\frac{\partial f}{\partial \vec{a}^j}\right)_t^T \tag{7.14}$$

Hence to evaluate the Ω_t matrix, we could require knowledge of the functions, f and r, so that the first and second order derivatives in equations 7.13 and 7.14 could be manually computed. Computing equation 7.13 is no more challenging to implement than computing $\frac{\partial \pi}{\partial \vec{x}}$ by Lemma 5, which is a necessary step to implement the VGL(λ) algorithm with a greedy policy. In many cases, such as in section 7.4.2, both of these computations simplify considerably, for example if the functions are linear in $\vec{a}$, or in a continuous time situation. Alternatively, if a neural network is used to represent the functions f and r, then we would require first and second order backpropagation through the neural network work to find these necessary derivatives.

We make further observations on the role of the Ω_t matrix in section 7.4.3.

7.4 VERTICAL LANDER EXPERIMENT

We describe a simple computer experiment which shows VGL learning with a greedy policy and demonstrates increased learning stability of VGL(1) compared to DHP (VGL(0)). We also demonstrate the value of using the Ω_t matrix as defined by equation 7.13 which can make learning progress achieve consistent convergence to local optimality with $\lambda = 1$.

After defining the problem in section 7.4.1, we derive an efficient formula for the greedy policy and Ω_t matrix (section 7.4.2), which provides some further insights into the purpose of Ω_t (section 7.4.3), before giving the experimental results in section 7.4.4.

7.4.1 Problem Definition

A spacecraft is dropped in a uniform gravitational field, and its objective is to make a fuel-efficient gentle landing. The spacecraft is constrained to move in a vertical line, and a single thruster is available to make upward accelerations. The state vector $\vec{x} = (h, v, u)^T$ has three components: height (h), velocity (v), and fuel remaining (u). The action vector is one-dimensional (so that $\vec{a} \equiv a \in \Re$) producing accelerations $a \in [0, 1]$. The Euler method with time-step Δt is used to integrate the motion, giving functions:

$$\begin{aligned} f((h,v,u)^T, a) =& (h + v\Delta t, v + (a - k_g)\Delta t, u - (k_u)a\Delta t)^T \\ r((h,v,u)^T, a) =& -(k_f)a\Delta t + r^c(a)\Delta t \end{aligned} \tag{7.15}$$

Here, $k_g = 0.2$ is a constant giving the acceleration due to gravity; the spacecraft can produce greater acceleration than that due to gravity. $k_f = 1$ is a constant giving fuel penalty. $k_u = 1$ is a unit conversion constant. Δt was chosen to be 1. $r^c(a)$ is an "action cost" function described further below that ensures the greedy policy function chooses actions satisfying $a \in [0, 1]$.

Trajectories terminate as soon as the spacecraft hits the ground ($h = 0$) or runs out of fuel ($u = 0$). For correct gradient calculations, clipping is needed at the terminal time-step, and differentiation of the functions needs to take account of this clipping. Further details of clipping are given by [5, Appendix E.1].

In addition to the reward function $r(\vec{x}, a)$ defined above, a final impulse of reward equal to $-\frac{1}{2}mv^2 - m(k_g)h$ is given as soon as the lander reaches a terminal state, where $m = 2$ is the mass of the spacecraft. The terms in this final reward are cost terms for the kinetic and potential energy respectively. The first cost term penalises landing too quickly. The second term is a cost term equivalent to the kinetic energy that the spacecraft would acquire by crashing to the ground under freefall.

A sample of ten optimal trajectories in state space is shown in figure 7.1.

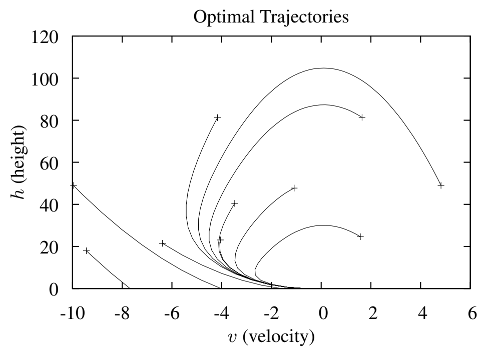

Figure 7.1 State space view of a sample of optimal trajectories in the Vertical Lander problem. Each trajectory starts at the cross symbol, and ends at $h = 0$. The u-dimension (fuel) of state space is not shown.

For the action cost function, we follow the method of [3], and choose

$$r^c(a) = -\int g^{-1}(a)da \tag{7.16}$$

where $g(x)$ is a chosen sigmoid function, as this will force a to be bound to the range of the chosen function $g(x)$, as illustrated in the following subsection. Hence to ensure $a \in [0, 1]$, we use

$$g(x) = \frac{1}{2}(\tanh(x/c) + 1) \tag{7.17}$$

where $c = 0.2$ is a sharpness constant, and therefore

$$r^c(a) = c\left(a \operatorname{arctanh}(1 - 2a) - \frac{1}{2}\ln(2 - 2a)\right).$$

7.4.2 Efficient evaluation of the greedy policy

The greedy policy $\pi(\vec{x}, \vec{w})$ is defined to choose the maximum with respect to $\vec{a}$ of the $\widetilde{Q}(\vec{x}, \vec{a}, \vec{w})$ function. This function has been defined to be smooth, so a numerical solver could be used to maximise this function, while introducing some inefficiency. A technical difficulty is that there might be multiple local maxima, and this means that as $\vec{w}$ or $\vec{x}$ change, the global maximum could hop from one local maximum to another, meaning the derivatives $\frac{\partial \pi}{\partial \vec{x}}$ and $\frac{\partial \pi}{\partial \vec{w}}$ would not be defined in these instances.

We can get around these problems, and derive a closed form solution to the greedy policy, by following the method of [3]. This leads to a very efficient and practical solution to using a greedy policy, and avoids the need to use an actor network altogether. To achieve this though, we do have to transfer to a continuous time analysis, i.e. we consider the case in the limit of $\Delta t \to 0$. The most important benefit that this delivers is that it forces the greedy policy function to be always differentiable, and hence for the VGL(λ) algorithm to be always defined.

We make a first order Taylor series expansion of the $\widetilde{Q}(\vec{x},\vec{a},\vec{w})$ function (eq. 7.1) about the point $\vec{x}$:

$$\begin{aligned}\widetilde{Q}(\vec{x},\vec{a},\vec{w}) &\approx r(\vec{x},\vec{a}) + \gamma\left(\left(\frac{\partial \widetilde{V}}{\partial \vec{x}}\right)^T (f(\vec{x},\vec{a})-\vec{x}) + \widetilde{V}(\vec{x},\vec{w})\right)\\ &= r(\vec{x},\vec{a}) + \gamma\left(\widetilde{G}(\vec{x},\vec{w})\right)^T (f(\vec{x},\vec{a})-\vec{x}) + \gamma\widetilde{V}(\vec{x},\vec{w}) \qquad (7.18)\end{aligned}$$

This approximation becomes exact in continuous time. We next define a greedy policy that maximises equation 7.18. Differentiating equation 7.18 gives

$$\begin{aligned}\left(\frac{\partial \widetilde{Q}}{\partial a}\right)_t &= \left(\frac{\partial r}{\partial a}\right)_t + \gamma\left(\frac{\partial f}{\partial a}\right)_t \widetilde{G}_t && \text{by eq. 7.18}\\ &= -(k_f)\Delta t + \left(\frac{\partial r^c(a)\Delta t}{\partial a}\right)_t + \gamma\Delta t(0,1,-1)\widetilde{G}_t && \text{by eq. 7.15}\\ &= \left(-(k_f) - g^{-1}(a_t) + \gamma(0,1,-1)\widetilde{G}_t\right)\Delta t && \text{by eq. 7.16} \quad (7.19)\end{aligned}$$

For the greedy policy to satisfy equation 7.9, we must have $\frac{\partial \widetilde{Q}}{\partial a} = 0$. Therefore,

$$\begin{aligned}0 &= -(k_f) - g^{-1}(a_t) + \gamma(0,1,-1)\widetilde{G}_t && \text{by eq. 7.19}\\ \Rightarrow a_t &= g\left(-(k_f) + \gamma(0,1,-1)\widetilde{G}_t\right) && (7.20)\end{aligned}$$

This closed form greedy policy is efficient to calculate, bound to $[0,1]$, and most importantly, always differentiable. This has achieved the objectives we aimed for by moving to continuous time. Furthermore, we get a simplified expression for the Ω_t matrix in continuous time:

Since $\dim(\vec{a}) = 1$, $\frac{\partial^2 \widetilde{Q}}{\partial a \partial a}$ is a scalar:

$$\begin{aligned}
\left(\frac{\partial^2 \widetilde{Q}}{\partial a \partial a}\right)_t &= \frac{\partial\Big(-(k_f) - g^{-1}(a_t) + \gamma(0,1,-1)\widetilde{G}_t\Big)}{\partial a_t}\Delta t && \text{by eq. 7.19} \\
&= -\frac{\partial g^{-1}(a_t)}{\partial a_t}\Delta t \\
&= -\frac{1}{g'\left(g^{-1}(a_t)\right)}\Delta t && \text{differentiating an inverse} \\
&= -\frac{1}{g'\Big(-(k_f) + \gamma(0,1,-1)\widetilde{G}_t\Big)}\Delta t && \text{by eq. 7.20}
\end{aligned} \tag{7.21}$$

Here g' is the derivative of the sigmoidal function g given by equation 7.17. Substituting equation 7.21 into equation 7.13 gives,

$$\Omega_t = (0,1,-1)^T g'\left(-(k_f) + \gamma(0,1,-1)\widetilde{G}_t\right)(0,1,-1)\Delta t \tag{7.22}$$

This is a much simpler version of the Ω_t matrix than that described by equations 7.13 and 7.14. The simplicity arose because of the linearity with respect to a of the function $f(x,a)$ and because of the change to continuous time.

Since we have moved to continuous time for the sake of deriving this efficient and always differentiable greedy policy, there are some consequential minor changes that we should make to the VGL algorithm. Firstly, if we were to re-derive lemmas 3, 4 and 5 using the $\widetilde{Q}$ function of equation 7.18, then the references in the lemmas to $\widetilde{G}_{t+1}$ would change to $\widetilde{G}_t$. For example, Lemma 4 would change to:

$$\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t = -\gamma \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \left(\frac{\partial f}{\partial \vec{a}}\right)_t^T \left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{a} \partial \vec{a}}\right)_t^{-1} \tag{7.23}$$

The VGL(λ) weight update would be the same as equation 7.12, but we would use Ω_t as given by equation 7.22, and the greedy policy given by equation 7.20. Also $\frac{\partial \pi}{\partial \vec{x}}$ (which is needed in the VGL(λ) algorithm in equation 7.4) is found most easily by differentiating equation 7.20, as opposed to using Lemma 5.

We note that equation 7.23, when combined with equation 7.21, is consistent with what is obtained by differentiating equation 7.20 directly.

7.4.3 Observations on the purpose of Ω_t

Now that we have a simple expression for Ω_t, we can make some observations on its purpose. Substituting Ω_t of equation 7.22 into the VGL weight update (eq. 7.2) gives:

$$\Delta\vec{w} = \alpha\gamma^2(\Delta t)\sum_{t\geq 0}\left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_t (0,1,-1)^T g'\left(-(k_f)+\gamma(0,1,-1)\widetilde{G}_t\right)(0,1,-1)({G'}_t-\widetilde{G}_t) \tag{7.24}$$

This has similarities in form to a weight update for the supervised learning neural network problem. Consider a neural network output function $y = g(s(\vec{x},\vec{w}))$ with sigmoidal activation function g, summation function $s(\vec{x},\vec{w})$, input vector $\vec{x}$ and weight vector $\vec{w}$. To make the neural network learn targets t_p for input vectors $\vec{x}_p$ (where p is a “pattern” index), the gradient descent weight update would be:

$$\begin{aligned}\Delta\vec{w} &= \alpha\sum_p \frac{\partial y_p}{\partial\vec{w}}(t_p - y_p)\\ &= \alpha\sum_p \frac{\partial s(\vec{x}_p,\vec{w})}{\partial\vec{w}} g'\left(s(\vec{x}_p,\vec{w})\right)(t_p-y_p)\end{aligned} \tag{7.25}$$

The similarities in equations 7.24 and 7.25 give hints at the purpose of Ω_t, since it is the Ω_t matrix that introduces the g' term into the VGL weight update equation (eq. 7.24). In neural network training, we would not omit the g' term from a weight update, and we would not treat it as a constant; and likewise we deduce that we should not omit the Ω_t matrix or treat it as fixed in the VGL learning algorithm. In neural network training, some algorithms choose to give the g' term an artificial boost to help escape plateaus of the error surface in weight space (e.g. Fahlman's method [4] which replaces g' by $g'+k$, for a small constant k), but this comes at the expense of the learning algorithm no longer being true gradient descent, and hence it not being as stable. Choosing to set $\Omega_t \equiv I$, the identity matrix, is like doing an extreme version of Fahlman's method on the VGL algorithm. This can help the learning algorithm escape plateaus of the R surface very effectively, but may lead to divergence. Plateaus are a severe problem when c is small in equation 7.17, since then $g' \approx 0$ which will make learning by equation 7.24 grind to a halt.

Having made these deductions about the role of Ω_t in VGL, we should make the caveat that these deductions only strictly apply to VGL(1) with a greedy policy, as that is the algorithm that Ω_t was derived for.

7.4.4 Experimental Results for Vertical Lander Problem

A DHP-style critic, $\widetilde{G}(\vec{x}, \vec{w})$, was provided by a fully connected multi-layer perceptron (MLP) (see [2] for details). The MLP had 3 inputs, two hidden layers of 6 units each, and 3 units in the output layer. Additional short-cut connections were present fully connecting all pairs of layers. The weights were initially randomised uniformly in the range $[-1, 1]$. The activation functions were logistic sigmoid functions in the hidden layers, and a linear function with slope 0.1 in the output layer. The input to the MLP was a rescaling of the state vector, given by $D(h, u, v)^T$, where $D = diag(0.01, 0.1, 0.02)$, and the output of the MLP gave $\widetilde{G}$ directly. In our implementation, we also defined the function $f(\vec{x}, \vec{a})$ to input and output coordinates rescaled by D, the intention being to ensure that the value gradients would be more appropriately scaled too.

Three algorithms were tested: VGL(0) with $\Omega_t = I$, the identity matrix; VGL(1) with $\Omega_t = I$; and VGL(1) with Ω_t given by equation 7.13 (denoted by throughout by "VGLΩ(1)"). Each algorithm was set the task of learning a group of 10 trajectories with randomly chosen fixed start points (the 10 start points used in all experiments are those shown in figure 7.1), and with initial fuel $u = 30$. In each iteration of the learning algorithm, the weight update was first accumulated for all 10 trajectories, and then this aggregate weight update was applied. In some experiments RPROP was used to accelerate this aggregate weight update at each iteration, with its default parameters defined by [10].

Figure 7.2 shows learning performance of the three algorithms, both with and without RPROP. These graphs show the clear stability and performance advantages of using $\lambda = 1$ and the chosen Ω_t matrix.

The VGLΩ(1) algorithm shows near-to-monotonic progress in the later stages of learning. The large kink in learning performance in the early iterations of RPROP is

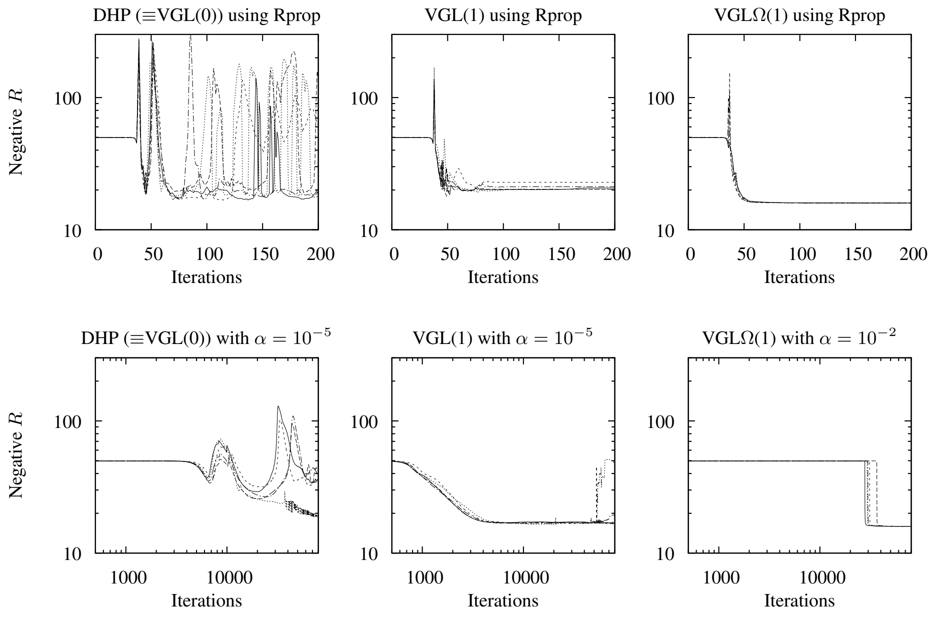

Figure 7.2 Results show learning progress for 5 typical random weight initialisations, for the problem of trying to learn 10 different trajectories. Results show increasing effectiveness (particularly in reduced volatility) for the three learning algorithms being considered, in the order that the graphs appear from left to right. The top row of graphs are all using RPROP to accelerate learning. The bottom row of graphs all use a fixed step-size parameter α.

present because RPROP causes the weight vector to traverse a significant discontinuity in the value function that exists at $h = 0, v = 0$.

VGL(0) shows very far-from-monotonic behaviour in this problem.

7.5 CONCLUSIONS

We have defined the VGL(λ) algorithm, and proven its equivalence under certain conditions to BPTT. VGL(1) with an Ω_t matrix defined by 7.13 is thus a critic learning algorithm that is proven to converge, under conditions stated in section 7.3.3, for a greedy policy and general smooth approximated critic. Although the proof does not extend to VGL(0), i.e. DHP, we hope that it might provide a pointer for research in that direction, particularly with the publication of Lemma 4. This convergence proof has also given us insights into how the Ω_t matrix can be chosen and what its purpose is, at least for the case of $\lambda = 1$ with a greedy policy, and we speculate that similar choices could be valid for $\lambda < 1$ or non-greedy policies. In our experiment, we used a simplified Ω_t matrix that was analytically derived and easy to compute; but this may not always be possible, so an approximation to equation 7.13 may be necessary.

Our experiment has been a simple one with known analytical functions, but it has demonstrated effectively the convergence properties of VGL(1) with the chosen Ω_t matrix, and the relative ease with which it can be accelerated using RPROP. In this experiment we found the convergence behaviour and optimality attained by VGL(1) with the chosen Ω_t matrix to be superior to VGL(1) with $\Omega_t = I$, which in turn has proved superior to VGL(0) (DHP) with $\Omega_t = I$. The given experiment was quite problematic for VGL(0) to learn and produce a stable solution, partly because in this deceptively simple environment the major proportion of the total reward arrives in the final time step, and partly because the low c value chosen for equation 7.17 makes the function g into approximately a step-function, which implies that the surface $R(\vec{x}, \vec{w})$ will be riddled with flat plateaus separated by steep cliffs.

It was surprising to the authors that the VGL(1) weight update has been proven to be equivalent to *gradient ascent on* R when previous research has always expected DHP (and therefore presumably its variant, VGL(1)) to be *gradient descent on* E, where E is the error function $E = \sum_t \left(G'_t - \widetilde{G}_t\right)^T \Omega_t \left(G'_t - \widetilde{G}_t\right)$.

REFERENCES

1. R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.
2. C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.
3. K. Doya. Reinforcement learning in continuous time and space. *Neural Computation*, 12(1):219–245, 2000.
4. S. E. Fahlman. Faster-learning variations on back-propagation: An empirical study. In *Proceedings of the 1988 Connectionist Summer School*, pages 38–51, San Mateo, CA, 1988. Morgan Kaufmann.
5. M. Fairbank. Reinforcement learning by value gradients. *eprint arXiv:0803.3539*, 2008.
6. M. Fairbank and E. Alonso. The divergence of reinforcement learning algorithms with value-iteration and function approximation. *eprint arXiv:1107.4606*, 2011.
7. M. Fairbank and E. Alonso. The local optimality of reinforcement learning by value gradients and its relationship to policy gradient learning. *eprint arXiv:1101.0428*, 2011.

8. S. Ferrari and R. F. Stengel. Model-based adaptive critic designs. *Handbook of learning and approximate dynamic programming, editors Jennie Si et al.*, pages 65–96, 2004.

9. D. Prokhorov and D. Wunsch. Adaptive critic designs. *IEEE Transactions on Neural Networks*, September:997–1007, 1997.

10. M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA, 1993.

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

12. J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, 1996.

13. F.-Y. Wang, H. Zhang, and D. Liu. Adaptive dynamic programming: An introduction. *IEEE Computational Intelligence Magazine*, pages 39–47, 2009.

14. C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

15. P. J. Werbos. Backpropagation through time: What it does and how to do it. In *Proceedings of the IEEE*, volume 78, No. 10, pages 1550–1560, 1990.

16. P. J. Werbos. Approximating dynamic programming for real-time control and neural modeling. *Handbook of Intelligent Control, editors White and Sofge*, pages 493–525, 1992.

17. P. J. Werbos. Neural networks, system identification, and control in the chemical process industries. *Handbook of Intelligent Control, editors White and Sofge*, pages 283–356, 1992.

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