

# Value-Gradient Learning

Michael Fairbank  
Department of Computing,  
School of Informatics,  
City University London,  
London, UK

Email: michael.fairbank.1@city.ac.uk

Eduardo Alonso  
Department of Computing,  
School of Informatics,  
City University London,  
London, UK

Email: E.Alonso@city.ac.uk

**Abstract**—We describe an Adaptive Dynamic Programming algorithm VGL( $\lambda$ ) for learning a critic function over a large continuous state space. The algorithm, which requires a learned model of the environment, extends Dual Heuristic Dynamic Programming to include a bootstrapping parameter analogous to that used in the reinforcement learning algorithm TD( $\lambda$ ). We provide on-line and batch mode implementations of the algorithm, and summarise the theoretical relationships and motivations of using this method over its precursor algorithms Dual Heuristic Dynamic Programming and TD( $\lambda$ ). Experiments for control problems using a neural network and greedy policy are provided.

## I. INTRODUCTION

Adaptive Dynamic Programming (ADP, [1]) is the study of how an agent can learn to choose actions that minimise a total long-term cost. For example a typical scenario is an agent wandering around in an state space  $\mathbb{S} \subset \mathbb{R}^n$ , such that at time  $t$  it has state vector  $\vec{x}_t \in \mathbb{S}$ . At each time  $t$  the agent chooses an action  $\vec{u}_t$  (from an action space  $\vec{u}_t \in \mathbb{A}$ ) which takes it to the next state according to the environment's model function  $\vec{x}_{t+1} = f(\vec{x}_t, \vec{u}_t)$ , and gives it an immediate scalar cost  $U_t$ , given by the function  $U_t = U(\vec{x}_t, \vec{u}_t)$ . The agent keeps moving, forming a trajectory of states  $(\vec{x}_0, \vec{x}_1, \dots)$ , which terminates if and when a state from the set of terminal states  $\mathbb{T} \subset \mathbb{S}$  is reached. If a terminal state  $\vec{x}_t \in \mathbb{T}$  is reached then a final instantaneous cost  $U_t = U(\vec{x}_t)$  is given which is independent of any action.

An *action network* or *actor* or *policy function*,  $A(\vec{x}, \vec{z})$ , is a neural network (or more generally, a function approximator) with parameter vector  $\vec{z}$ , which specifies which action  $\vec{u} = A(\vec{x}, \vec{z})$  to take for any given state  $\vec{x}$ . From any given state  $\vec{x}$ , the expectation of the total future long term cost encountered when following actions chosen by an action network  $A(\vec{x}, \vec{z})$ , is given by the function  $J(\vec{x}, \vec{z}) = \langle \sum_t \gamma^t U_t \rangle$ , where  $\langle \cdot \rangle$  denotes expectation. This is the *cost-to-go* function for the given action network, also known as the *value function* from dynamic programming [2] and reinforcement learning (RL, [3]). Here  $\gamma \in [0, 1]$  is a constant *discount factor* that specifies the importance of long term costs over short term ones. The objective of ADP and RL is to train the action network to choose actions that minimise the total cost-to-go function from any state  $\vec{x}$ .

ADP also uses a second neural network (or function approximator),  $\tilde{J}(\vec{x}, \vec{w}) \in \mathbb{R}$ , with weight vector  $\vec{w}$ , known as

the *critic* or *approximate value function*. The intermediate objective of ADP is to train the critic to approximate the cost-to-go function, so that  $\tilde{J}(\vec{x}, \vec{w}) \approx J(\vec{x}, \vec{z})$  for all  $\vec{x} \in \mathbb{S}$ . If this is achieved perfectly, and if *simultaneously* the action network always chooses actions according to:

$$\vec{u} = \arg \min_{\vec{u} \in \mathbb{A}} \left\langle U(\vec{x}, \vec{u}) + \gamma \tilde{J}(f(\vec{x}, \vec{u}), \vec{w}) \right\rangle \quad \forall \vec{x} \quad (1)$$

then Bellman's Optimality Condition [2] shows the trajectories produced will be optimal, and the action network is optimal.

The method of choosing actions purely by equation 1 is called the *greedy policy on  $\tilde{J}$*  since it chooses actions that the critic rates as best.

The ADP method Heuristic Dual Programming (HDP), and the RL methods TD( $\lambda$ ) and Q-learning [4], [5], are all critic learning methods that sample trajectories and update the critic values encountered along the trajectory. We call these methods *value learning* (VL) methods since they learn the values of  $\tilde{J}$  along the trajectory. Variants of these methods have produced successes in control problems [6], [7], but they can be very slow since Bellman's condition needs meeting over the entire continuous state space for optimality. Even if Bellman's condition is perfectly satisfied along a single trajectory, performance can be extremely far from optimal. Bellman's condition must be satisfied at least on the neighbouring trajectories too for local optimality. Hence VL methods must be supplemented by exploration of the environment just to attain local optimality. This exploration could be provided by stochastic model functions, a stochastic policy, or a stochastic start point for each trajectory. The methods we follow significantly reduce the need for this explicit exploration.

In this paper, we follow the ADP methods of Dual Heuristic Dynamic Programming (DHP) and Globalized DHP (GDHP) [8], [9], [10], [1]. DHP and GDHP work by explicitly learning the *gradient* of the value function with respect to the state vector, i.e. they learn  $\frac{\partial J}{\partial \vec{x}}$  instead of  $J$  directly. We refer to these methods collectively as *value gradient learning* (VGL) methods, to distinguish them from VL methods. The main reason to use VGL methods over VL methods is that whereas VL methods need to learn the critic values all along the current trajectory *and all neighbouring trajectories* to achieve local optimality, VGL methods only need learn the value-gradient along the single trajectory to achieve the same assurance

of local optimality. This motivation is discussed further in section I-A. Also, VGL methods can work very well in control problems in continuous state spaces.

We extend the VGL methods from DHP to include a bootstrapping parameter  $\lambda \in [0, 1]$  (just as  $\text{TD}(\lambda)$  is an extension of  $\text{TD}(0)$ ), to give the algorithm we call  $\text{VGL}(\lambda)$ , which we describe in this paper. For  $\lambda = 0$ ,  $\text{VGL}(\lambda)$  is equivalent to DHP, but for  $\lambda > 0$ ,  $\text{VGL}(\lambda)$  is a new algorithm. The motivations for this extension are that using  $\lambda > 0$  can increase the stability of learning and sometimes increase learning speed, as discussed further in section I-B.

VGL methods (including DHP) are model-based methods that require a learned differentiable model of the environment and cost functions,  $f(\vec{x}, \vec{u})$  and  $U(\vec{x}, \vec{u})$ . We discuss this more in section I-C.

In sections I-A and I-B, we expand on the motivations for VGL methods and the  $\lambda$  parameter, and in section I-C we discuss the applicability of VGL methods and the necessary learning of the model functions. In the rest of the paper, we define the  $\text{VGL}(\lambda)$  algorithm in section II, and state its relationship to  $\text{TD}(\lambda)$  and DHP in subsections II-C and II-D. In section III, we give experimental results, and finally, in section IV we give conclusions.

#### A. Motivations for DHP and VGL Methods

The VGL (and hence DHP) methods address the issue of the Bellman equation needing to be solved over the whole of state space, in that it turns out to be only necessary to fully learn the value gradient along a *single trajectory*, under a greedy policy, for it to be locally extremal, and often locally optimal. This is proven by [11], and is closely related to Pontryagin’s Minimum Principle [12]. This optimality condition contrasts strongly with the VL methods which need to learn the value function over all immediately neighbouring trajectories too in order to achieve the same level of guarantee for being locally extremal/optimal trajectories. So this is a significant efficiency gain for the VGL methods, and is their principal motivation.

This implies that VGL methods have a much lesser requirement for exploration than VL methods do, since the *local* part of exploration comes for free by using VGL methods. What we mean by this is that provided the VGL learning algorithm makes progress in learning the value gradient all along a trajectory, while following a greedy policy, then the trajectory will automatically make progress in bending itself towards a locally optimal shape. This will happen without the need for any stochastic exploration. In effect, by using VGL methods, local exploration is automatic; or put another way the traditional RL dilemma of “exploration versus exploitation” transforms into “exploration *and* exploitation”, locally at least.

This leads to greater efficiency for VGL compared to VL. In comparison the failure of VL without any exploration in a deterministic environment is dramatic and common, even when the value-function is *perfectly learned* along a single trajectory. The experiment in section III-A confirms this.

Both VL and VGL methods have the same requirement for global optimality, that is if the value function (or its gradient)

is exactly learned over all of state space, with a greedy policy, then Bellman’s condition assures global optimality.

For a fuller discussion of the motivations to use VGL methods see [13].

#### B. Motivations for Introducing the $\lambda$ Parameter into DHP

The motivations for using the  $\lambda$  parameter are that choosing  $\lambda$  carefully can increase the stability of learning and sometimes increase learning speed.

For example, when  $\lambda = 1$ , the weight update used by  $\text{TD}(\lambda)$  for a fixed action network is true gradient descent on an error function and so is guaranteed to converge [4]. However when  $\lambda = 0$ ,  $\text{TD}(\lambda)$  is not true gradient descent on any error function, and this weight update can diverge when the approximator for  $\tilde{J}(\vec{x}, \vec{w})$  is non-linear in  $\vec{w}$  [14]. Similarly, DHP is not true gradient descent on any function, and general instability in this case has been proven [15, sec. 7.7-7.8]. However  $\text{VGL}(1)$  is true gradient descent on an error function, for a fixed action network, and so will converge.

Furthermore, when a greedy policy is used, all of  $\text{TD}(1)$ ,  $\text{TD}(0)$  and DHP can be made to diverge [16]. However with carefully chosen learning parameters and smoothness assumptions,  $\text{VGL}(1)$  can be guaranteed to converge with a greedy policy, as discussed further in section II. In this case  $\text{VGL}(1)$  becomes identical to backpropagation through time [17] acting directly on the greedy policy, as proven by [11].

Choosing  $\lambda$  carefully can affect the learning speed of critic learning algorithms. Having a high value of  $\lambda$  makes the target in the critic weight update use a longer “look-ahead” along the trajectory, and so this can improve learning speed. However, conversely, having too large a value of  $\lambda$  can increase the variance of the sampled target  $J$  value in a stochastic environment, which can slow down learning. Hence in  $\text{TD}(\lambda)$ , often the optimal value of  $\lambda$  for learning speed is somewhere in the middle range of  $\lambda \in [0, 1]$ , as demonstrated by [3].

#### C. Applicability of VGL methods and System Identification

The VGL methods are strictly model-based methods, and this is largely where the extra efficiency comes from. We assume the model functions can be learned by a separate “system identification” learning process, for example as described by [18]. This system identification process could have taken place prior to the main learning process (e.g. like [19]), or concurrently with it, and results in learned model functions  $f(\vec{x}, \vec{u})$  and  $U(\vec{x}, \vec{u})$ . Alternatively, there is an extremely fast on-line model-learning method by [20] which could be used. We do not describe this model-learning stage of the process any further in this paper.

The VGL methods work naturally with continuous space problems. They are also limited to situations to where the model functions and policy are once-differentiable, comprising of a deterministic part plus optionally some additive noise. In many situations we can force smoothness onto the model functions by using a smooth deterministic function approximator to represent them.

DHP successes include autopilot landing [10], power system control [21] and many others [1]. However some traditional RL problem domains with discrete spaces would not be readily solvable by VGL methods, such as a “grid world” problem, backgammon, or a k-armed bandit problem. Also, step cost functions would not be applicable, thus excluding problems such balancing a pole where the total cost is a function of the *integer* number of time steps that the pole is balanced for. However the pole balancing problem can be solved by DHP if a smoothed out cost function is used [22]. As a rule of thumb, if a problem is suitable for smooth gradient descent on  $J$  with respect to  $\vec{w}$ , then it will be suitable to work on VGL methods.

## II. THE VGL( $\lambda$ ) ALGORITHM

In this section we define the VGL( $\lambda$ ) Algorithm and state how it relates to the algorithms TD( $\lambda$ ) and Dual Heuristic Dynamic Programming.

To define the VGL( $\lambda$ ) algorithm, we require that  $\tilde{J}(\vec{x}, \vec{w})$  is defined by a *smooth* scalar function approximator, e.g. a neural network with weight vector  $\vec{w}$ . This enables us to define the “critic gradient”, or “approximate value gradient”, as  $\tilde{G}(\vec{x}, \vec{w}) \equiv \frac{\partial \tilde{J}(\vec{x}, \vec{w})}{\partial \vec{x}}$ .

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,  $\tilde{G}$  and  $\frac{\partial \tilde{J}}{\partial \vec{x}}$  are columns. Also, 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  $\vec{u}_t$  where appropriate; so that for example  $\tilde{J}_{t+1} \equiv \tilde{J}(\vec{x}_{t+1}, \vec{w})$  and  $\tilde{G}_t \equiv \tilde{G}(\vec{x}_t, \vec{w})$ .

Differentiating a column vector function by a column vector causes the vector in the numerator to become transposed (becoming a row).<sup>1</sup> For example  $\frac{\partial f}{\partial \vec{x}}$  is a matrix with element  $(i, j)$  equal to  $\frac{\partial f(\vec{x}, \vec{u})^j}{\partial \vec{x}^i}$ . Similarly,  $\left(\frac{\partial \tilde{G}}{\partial \vec{w}}\right)^{ij} = \frac{\partial \tilde{G}^j}{\partial \vec{w}^i}$ , and  $\left(\frac{\partial \tilde{G}}{\partial \vec{w}}\right)_t$  is this matrix evaluated at  $(\vec{x}_t, \vec{w})$ .

Then, using this notation and the implied matrix products, the VGL( $\lambda$ ) algorithm can be defined as a weight update of the form:

$$\Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{G}}{\partial \vec{w}} \right)_t \Omega_t (G'_t - \tilde{G}_t) \quad (2)$$

where  $\alpha$  is a small positive learning rate;  $\tilde{G}_t$  is the critic gradient; and  $G'_t$  is the “target value gradient” defined recursively by:

$$G'_t = \left( \frac{DU}{D\vec{x}} \right)_t + \gamma \left( \frac{Df}{D\vec{x}} \right)_t \left( \lambda G'_{t+1} + (1 - \lambda) \tilde{G}_{t+1} \right) \quad (3)$$

with  $G'_t = \left( \frac{\partial U}{\partial \vec{x}} \right)_t$  at any terminal state; where  $\lambda \in [0, 1]$  is a constant; where  $\Omega_t$  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

$$\frac{D}{D\vec{x}} \equiv \frac{\partial}{\partial \vec{x}} + \frac{\partial A}{\partial \vec{x}} \frac{\partial}{\partial \vec{u}}; \quad (4)$$

<sup>1</sup>This is the opposite of a common convention.

and where all of these derivatives are assumed to exist. We ensure the recursion in eq. 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 2, 3 and 4 define the VGL( $\lambda$ ) algorithm. Section II-B gives further implementation details and pseudocode.

The target value-gradients,  $G'_t$ , are so called because the VGL objective is to achieve  $\tilde{G}_t = G'_t$  for all  $t$  along a trajectory. This objective ensures a locally extremal, and often locally optimal, trajectory (as proven by [11]), when combined with a greedy policy. It should be noted that this objective is not straightforward to achieve since the targets  $G'_t$  are moving ones and are highly dependent on  $\vec{w}$  (especially when a greedy policy is being used, so that then the policy is also *indirectly* dependent on  $\vec{w}$ ). Hence we must use the weight update to *slowly* move the approximated gradients towards their targets.

The  $\Omega_t$  matrix was introduced by Werbos for the algorithm GDHP (e.g. see [15, eq. 32]), which is very closely related to VGL( $\lambda$ ). It is a free parameter, included for generality, since the presence of any positive definite matrix here in equation 2 will force every component of  $\tilde{G}_t$  to move towards the corresponding component of  $G'_t$  (in any basis). It is often just taken to be the identity matrix for simplicity. However for the special choice of

$$\Omega_t = \begin{cases} \left( \frac{\partial f}{\partial \vec{u}} \right)_{t-1}^T \left( \frac{\partial^2 \tilde{Q}}{\partial \vec{u} \partial \vec{u}} \right)_{t-1}^{-1} \left( \frac{\partial f}{\partial \vec{u}} \right)_{t-1} & \text{for } t > 0 \\ 0 & \text{for } t = 0 \end{cases}, \quad (5)$$

the algorithm VGL(1) is proven to converge for a sufficiently small learning rate and when used in conjunction with a greedy policy, under certain smoothness assumptions [11]. Here  $\tilde{Q}$  is the approximate Q Value function defined by

$$\tilde{Q}(\vec{x}, \vec{u}, \vec{w}) = U(\vec{x}, \vec{u}) + \gamma \tilde{J}(f(\vec{x}, \vec{u}), \vec{w}). \quad (6)$$

### A. Action Network Training Algorithm and Greedy Policy

The VGL( $\lambda$ ) algorithm is for training a critic function. To make the agent learn to behave optimally, the action network function  $A(\vec{x}, \vec{z})$  also needs training. We follow the method of [10], which uses the following weight update for the action network at each time step  $t$ :

$$\Delta \vec{z} = -\beta \left( \frac{\partial A}{\partial \vec{z}} \right)_t \left( \left( \frac{\partial U}{\partial \vec{u}} \right)_t + \gamma \left( \frac{\partial f}{\partial \vec{u}} \right)_t \tilde{G}_{t+1} \right) \quad (7)$$

where  $\beta$  is a separate learning rate for the action network. The multiplication by  $\frac{\partial A}{\partial \vec{z}}$  can be done quickly and exactly by ordinary backpropagation. This weight update can be done concurrently with the critic weight update, or by iteratively doing several critic weight updates followed by several action network weight updates.

The above weight update is equivalent to  $\Delta \vec{z} = -\beta \left( \frac{\partial A}{\partial \vec{z}} \right)_t \left( \frac{\partial \tilde{Q}}{\partial \vec{u}} \right)_t$ , which is direct gradient descent on the function  $\tilde{Q}(\vec{x}_t, A(\vec{x}_t, \vec{z}), \vec{w})$  with respect to  $\vec{z}$ , where  $\tilde{Q}$  is defined by eq. 6. Consequently the objective of the above weight update is to achieve

$$A(\vec{x}, \vec{z}) = \arg \min_{\vec{u} \in \mathbb{A}} (\tilde{Q}(\vec{x}, \vec{u}, \vec{w})) \quad \forall \vec{x}. \quad (8)$$

In some circumstances we can omit the action network altogether and just use the right hand side of equation 8 directly, forming the *greedy policy* (as in equation 1). This saves the difficulty of having to simultaneously train the critic and action networks, which may interfere with each other in unpredictable ways. Instead it makes the action network appear to be always fully trained. A greedy policy is only possible when the right hand side of equation 8 is efficient to compute, which is common in the continuous time situations described by [7] or [23, section 2.2]. In this case the greedy policy often reduces to

$$(\vec{u}_t)^i \equiv g \left( - \left( \frac{\partial U}{\partial \vec{u}^i} \right)_t - \gamma \left( \frac{\partial f}{\partial \vec{u}^i} \right)_t \tilde{G}_t \right), \quad (9)$$

where  $g(x)$  is a chosen sigmoid function, for example a hyperbolic tangent or logistic function. This greedy policy equation produces actions that are bound to the range of  $g(x)$ , and it is conveniently efficient and differentiable, so is applicable for use in the VGL( $\lambda$ ) algorithm. We give an example of this kind of greedy policy in section III-C.

### B. Implementation of VGL( $\lambda$ )

We now give pseudocode for two different ways of implementing the VGL( $\lambda$ ) algorithm - one is for on-line learning which can be continually applied as trajectories are expanded, and one is a batch mode implementation which is slightly more efficient but is only applicable to completed trajectories.

When implementing VGL( $\lambda$ ), the function  $\tilde{G}(\vec{x}, \vec{w})$  can be implemented in two different possible ways. Since  $\tilde{G}(\vec{x}, \vec{w})$  is defined to be  $\frac{\partial \tilde{J}(\vec{x}, \vec{w})}{\partial \vec{x}}$ , the appearance of the term  $\frac{\partial \tilde{G}}{\partial \vec{w}}$  in equation 2 is defined to mean  $\frac{\partial^2 \tilde{J}}{\partial \vec{w} \partial \vec{x}}$ . We refer to this method of implementation as using a *scalar critic function*,  $\tilde{J}(\vec{x}, \vec{w})$ , or a *GDHP style critic*. However an alternative, easier, method is to implement  $\tilde{G}(\vec{x}, \vec{w})$  directly as the output of a smooth vector function approximator of output dimension  $\dim(\vec{x})$ . In this case the scalar function  $\tilde{J}(\vec{x}, \vec{w})$  is never actually needed in VGL( $\lambda$ ). We refer to this alternative method of implementation as using a *vector critic function*,  $\tilde{G}(\vec{x}, \vec{w})$ , or as using a *DHP style critic*. Either of these two approaches is a valid way to implement the VGL( $\lambda$ ) algorithm.

Algorithm 1 makes a direct implementation of VGL( $\lambda$ ) for episodic environments. It makes a forward pass through the trajectory, storing all states and actions, followed by a backward pass through the trajectory accumulating  $G'_t$  by the recursion in Eq. 3.

In this implementation, if a neural network is used to output the function  $\tilde{G}(\vec{x}, \vec{w})$ , then the matrix-vector products involving  $\frac{\partial \tilde{G}}{\partial \vec{w}}$  can be calculated in time  $O(\dim(\vec{w}))$ , by using backpropagation. Similarly, if  $\tilde{G} \equiv \frac{\partial \tilde{J}}{\partial \vec{w}}$ , where  $\tilde{J}$  is the scalar output of a neural network, then the matrix product involving second order derivatives  $\frac{\partial \tilde{G}}{\partial \vec{w}}$  can still be evaluated in the same time, by using methods analogous to those of

---

**Algorithm 1** VGL( $\lambda$ ). Batch-mode implementation for episodic environments.

---

```

1:  $t \leftarrow 0$ 
2: {Unroll trajectory...}
3: while not terminated( $\vec{x}_t$ ) do
4:    $\vec{u}_t \leftarrow A(\vec{x}_t, \vec{z})$ 
5:    $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t)$ 
6:    $t \leftarrow t + 1$ 
7: end while
8:  $F \leftarrow t$ 
9:  $\vec{p} \leftarrow \left( \frac{\partial U}{\partial \vec{x}} \right)_t$ ,  $\Delta \vec{w} \leftarrow \vec{0}$ ,  $\Delta \vec{z} \leftarrow \vec{0}$ 
10: {Backwards pass...}
11: for  $t = F - 1$  to 0 step -1 do
12:    $G'_t \leftarrow \left( \frac{\partial U}{\partial \vec{x}} \right)_t + \gamma \left( \frac{\partial f}{\partial \vec{x}} \right)_t \vec{p}$ 
         $+ \left( \frac{\partial A}{\partial \vec{x}} \right)_t \left( \left( \frac{\partial U}{\partial \vec{u}} \right)_t + \gamma \left( \frac{\partial f}{\partial \vec{u}} \right)_t \vec{p} \right)$ 
13:    $\Delta \vec{w} \leftarrow \Delta \vec{w} + \left( \frac{\partial \tilde{G}}{\partial \vec{w}} \right)_t \Omega_t (G'_t - \tilde{G}_t)$ 
14:    $\Delta \vec{z} \leftarrow \Delta \vec{z} - \left( \frac{\partial A}{\partial \vec{z}} \right)_t \left( \left( \frac{\partial U}{\partial \vec{u}} \right)_t + \gamma \left( \frac{\partial f}{\partial \vec{u}} \right)_t \tilde{G}_{t+1} \right)$ 
15:    $\vec{p} \leftarrow \lambda G'_t + (1 - \lambda) \tilde{G}_t$ 
16: end for
17:  $\vec{w} \leftarrow \vec{w} + \alpha \Delta \vec{w}$ 
18:  $\vec{z} \leftarrow \vec{z} + \beta \Delta \vec{z}$ 

```

---

[24]. The matrix products involving  $\frac{\partial A}{\partial \vec{x}}$  can also be evaluated quickly using backpropagation. Hence the whole algorithm takes  $O(n)$  operations per time step of the trajectory, where  $n = \max(\dim(\vec{w}), \dim(\vec{z}))$ .

For non-episodic environments, Algorithm 2 gives an on-line implementation of VGL( $\lambda$ ). Unlike the previous algorithm, this one does not require that the trajectory reaches a terminal state before the weight update can be applied. This algorithm runs in a slower time of  $O(\dim(\vec{w}) \dim(\vec{x})^2)$  operations per time step of the trajectory, and its derivation is given by [11, Appendix B]. In the case of  $\lambda = 0$ , the algorithm can be optimised to remove the variable  $E$ , and then the algorithmic complexity drops to be the same as that of Algorithm 1.

Neither algorithm attempts to learn the value gradient at the final time-step of a trajectory since it is prior knowledge that the target value gradient is always  $\frac{\partial U}{\partial \vec{x}}$  at any terminal state. Hence we assume the function approximator for  $\tilde{G}(\vec{x}, \vec{w})$  has been designed to explicitly return  $\frac{\partial U}{\partial \vec{x}}$  for all terminal states  $\vec{x}$ .

Both algorithms incorporate the action network weight update of equation 7. If a different actor weight update scheme was needed then these lines could be moved or replaced, and similarly if a greedy policy was used then these lines would be removed.

### C. Relationship to TD( $\lambda$ )

TD( $\lambda$ ), and its related algorithms Sarsa( $\lambda$ ) and Q( $\lambda$ ), are important learning algorithms of RL [25], [5]. VGL( $\lambda$ ) was adapted from these algorithms to improve efficiency of learning of control problems in continuous spaces, where the model functions are known, and where a critic function must be

---

**Algorithm 2** VGL( $\lambda$ ). On-line implementation.

---

```
1:  $E \leftarrow 0$  { $E \in \mathfrak{R}^{\dim(\vec{w}) \times \dim(\vec{x})}$  is an “eligibility trace”  
   workspace matrix.}  
2:  $t \leftarrow 0$   
3: while not terminated( $\vec{x}_t$ ) do  
4:    $\vec{u}_t \leftarrow A(\vec{x}_t, \vec{z})$   
5:    $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{u}_t)$   
6:    $\vec{\delta} \leftarrow \left(\frac{\partial U}{\partial \vec{x}}\right)_t + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\frac{\partial U}{\partial \vec{u}}\right)_t$   
      $+ \gamma \left( \left(\frac{\partial f}{\partial \vec{x}}\right)_t + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\frac{\partial f}{\partial \vec{u}}\right)_t \right) \tilde{G}_{t+1} - \tilde{G}_t$   
7:    $E \leftarrow E + \left(\frac{\partial \tilde{G}}{\partial \vec{w}}\right)_t \Omega_t$   
8:    $\vec{w} \leftarrow \vec{w} + \alpha E \vec{\delta}$   
9:    $E \leftarrow \lambda \gamma E \left( \left(\frac{\partial f}{\partial \vec{x}}\right)_t + \left(\frac{\partial A}{\partial \vec{x}}\right)_t \left(\frac{\partial f}{\partial \vec{u}}\right)_t \right)$   
10:   $\vec{z} \leftarrow \vec{z} - \beta \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left( \left(\frac{\partial U}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{u}}\right)_t \tilde{G}_{t+1} \right)$   
11:   $t \leftarrow t + 1$   
12: end while
```

---

learned by a neural network. Hence the VGL( $\lambda$ ) algorithm has a very similar form to that of TD( $\lambda$ ).

For a given trajectory, the following critic weight update is equivalent to the TD( $\lambda$ ) algorithm:

$$\Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \vec{w}} \right)_t (J'_t - \tilde{J}_t) \quad (10)$$

where  $J'$  is defined recursively by

$$J'_t = U_t + \gamma \left( \lambda J'_{t+1} + (1 - \lambda) \tilde{J}_{t+1} \right) \quad (11)$$

with  $J'_t = U_t$  at any terminal state, and where  $\lambda \in [0, 1]$  is a fixed global constant. For convergence of this recursion we require that either  $\gamma\lambda < 1$ , or the environment is episodic, just like the recursion in equation 3. To understand the equivalence of this weight update to the TD( $\lambda$ ) algorithm originally described by [4], first we note that  $J'$  as defined by equation 11 is identical to the “ $\lambda$ -return” of [5], as proven by [11, Appendix A]. Hence equation 10 is an application of what [3] describe as the “forwards view of TD( $\lambda$ )”, and hence is equivalent to the TD( $\lambda$ ) weight update (see [11, Appendix A] for further details).

Since  $J'_t$  is defined for an arbitrary trajectory, we can rewrite its recursive definition as

$$J'(\vec{x}_t, \vec{w}, \vec{z}) = U(\vec{x}_t, A(\vec{x}_t, \vec{z})) + \gamma \left( \lambda J'(f(\vec{x}_t, A(\vec{x}_t, \vec{z})), \vec{w}) \right. \\ \left. + (1 - \lambda) \tilde{J}(f(\vec{x}_t, A(\vec{x}_t, \vec{z})), \vec{w}) \right) \quad (12)$$

We can now obtain the relationship of VGL( $\lambda$ ) to TD( $\lambda$ ). Differentiating equation 12 fully with respect to  $\vec{x}_t$  (and applying the chain rule,  $\vec{u}_t = A(\vec{x}_t, \vec{z})$ ,  $\vec{x}_{t+1} = f(\vec{x}_t, \vec{u}_t)$ ,  $\tilde{G} \equiv \frac{\partial \tilde{J}}{\partial \vec{x}}$  and trajectory shorthand notation) produces the same recursion as equation 3. This proves that  $G' \equiv \frac{\partial J'}{\partial \vec{x}}$ . Furthermore, replacing all of the terms  $\tilde{J}$  and  $J'$  in the TD( $\lambda$ ) weight update by their derivatives with respect to  $\vec{x}$  produces the VGL( $\lambda$ ) weight update (with  $\Omega_t \equiv I$ ). Hence VGL( $\lambda$ ) is a differentiated form of TD( $\lambda$ ); whereas TD( $\lambda$ ) attempts

to learn *values*, VGL( $\lambda$ ) attempts to learn *value gradients*. The reason VGL( $\lambda$ ) does this is because value-gradients are what the greedy policy uses to decide which actions to take (for example see equation 9, or see section I-A for a further discussion).

#### D. Relationship to Dual Heuristic Dynamic Programming

The algorithm Dual Heuristic Dynamic Programming (DHP) by [8] and described more recently by [10], [1] is identical to VGL(0) with a vector critic. The DHP algorithm has been designed with similar motivations as ours, and pre-dates our work. Our implementation VGL( $\lambda$ ) generalises it to include a bootstrapping parameter  $\lambda$  analogous to that used in TD( $\lambda$ ). The relationship of DHP to VGL( $\lambda$ ) is identical to the relationship of TD(0) to TD( $\lambda$ ).

The algorithm Globalized Dual Heuristic Dynamic Programming (GDHP) is identical to a weight update that is a linear combination of VGL(0) with a scalar critic and TD(0).

### III. EMPIRICAL RESULTS

We describe neural network experiments for two control problems: a simple quadratic optimisation problem and a vertically moving spacecraft simulation. We show the performance of VL versus VGL methods, and also investigate the effects of varying the  $\lambda$  parameter and  $\Omega_t$  matrix.

#### A. Quadratic Optimisation Problem

We now describe a quadratic optimisation experiment using an actor-critic architecture, and compare the effectiveness of VGL( $\lambda$ ) to TD( $\lambda$ ) in the situation of both noisy and deterministic policies. This shows the motivation for VGL based methods in that they work very quickly and that they can do local exploration even while only following deterministic trajectories.

We define an environment with  $\vec{x} = x \in \mathfrak{R}$  and  $\vec{u} = u \in \mathfrak{R}$ , and model and cost functions:

$$f(x, t, u) = x + u \\ U(x, t, u) = (u)^2$$

Each trajectory is defined to terminate on arriving at time step  $t = 2$ , and on termination a final instantaneous cost of  $U(\vec{x}) = (x)^2$  is given. The only actions used in the trajectory are  $u_0$  and  $u_1$ ; the total cost for this trajectory is  $(u_0)^2 + (u_1)^2 + (x_0 + u_0 + u_1)^2$ , and the theoretical *optimal* total cost for the whole trajectory is  $J^* = (x_0)^2/3$ .

The action network was a multi-layer perceptron (MLP, see [26] for details) with two inputs, one output and one hidden layer of 4 nodes, shortcut connections from the input layer to the output layer, and with activation function  $g(x) = \tanh(x)$  at all nodes. The weights  $\vec{z}$  were initially randomised uniformly from the range [-0.1, 0.1]. The critic network  $\tilde{J}(\vec{x}, \vec{w})$  was a *scalar critic*, so that in this implementation, VGL(0) is equivalent to GDHP. The critic was identically dimensioned to the action network, with a weight vector  $\vec{w}$  randomised initially in the same way. The activation function used for the critic was  $g(x) = \tanh(x)$  at all nodes except for the output

node, which used  $g(x) = x$ . The input vector to each neural network was  $(x, t)$ .

Each trajectory was made to start at  $x_0 = 0.8$ . Learning rates for the critic and actor were both  $\alpha = 0.1$  and  $\beta = 0.1$ , respectively, with discount factor  $\gamma = 1$ . To provide the facility for exploration, Gaussian noise with mean zero and variance  $\sigma^2$  was added to the output of the action network to form the policy function,  $A(\vec{x}, \vec{z})$ .

The critic learning algorithms tested were TD(1), TD(0), VGL(1) and VGL(0). The action network’s weight update was done by equation 7 in all experiments. We repeated the experiments with noise ( $\sigma = 0.01$ ) and without noise ( $\sigma = 0$ ). Results averaged from 40 trials for each algorithm are shown in figure 1. The results show that the value-learning method (TD( $\lambda$ )) could not cope without some random exploration, but the VGL based methods (GDHP and VGL( $\lambda$ )) work successfully for both  $\sigma$  values used. Also, in comparison to the value learning methods, VGL methods are very fast. On the other-hand VGL methods require knowledge of the model functions (in order to use their derivatives), whereas VL methods do not. But when the model functions *are* available, the increased speed and automatic local exploration provides a strong motivation to use VGL methods.

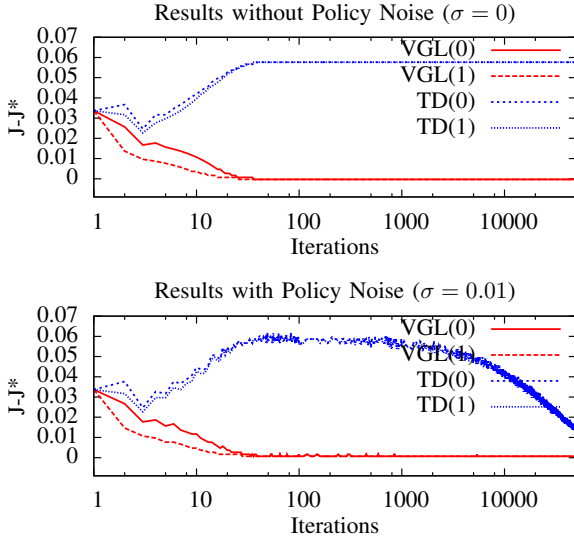


Fig. 1. Algorithm performances for the quadratic optimisation problem of section III-A, both with and without policy noise. The  $y$  axis shows  $J - J^*$ , where  $J^*$  is the optimal trajectory cost. Compared to the VL method (TD( $\lambda$ )), the VGL method works well in the absence of stochastic exploration, and quickly attains  $J = J^*$ . The VL method fails without stochastic exploration here (i.e. it converges to a suboptimal policy), but does learn slowly and successfully in the presence of policy noise.

## B. Vertical Spacecraft Problem

In this section we consider a neural network controlled spacecraft.

A spacecraft of mass  $m$  is dropped in a uniform gravitational field. The spacecraft is constrained to move in a vertical line, and a single thruster is available to make upward accelerations. The state vector of the spacecraft is  $\vec{x} = (h, v, t)^T$  and

has three components: height ( $h$ ), velocity ( $v$ ) and time step ( $t$ ). The action vector is one-dimensional (so that  $\vec{u} \equiv u \in \mathbb{R}$ ) producing accelerations  $u \in [0, 1]$ . The Euler method with time-step  $\Delta t$  is used to integrate the equation of motion, giving the model function:

$$f((h, v, t)^T, u) = (h + v\Delta t, v + (u - k_g)\Delta t, t + 1)^T$$

Here,  $k_g = 0.2$  is a constant giving the acceleration due to gravity (which is less than the range of  $u$ ; so the spacecraft can overcome gravity easily).  $\Delta t$  was chosen to be 0.4.

A trajectory is defined to last exactly 200 time steps. A final impulse of cost equal to

$$U(\vec{x}) = \frac{1}{2}mv^2 + m(k_g)h \quad (14)$$

is given on completion of the trajectory. This cost penalises the total kinetic and potential energy that the spacecraft has at the end of the trajectory. This means the task is for the spacecraft to lose as much mechanical energy as possible throughout the duration of the trajectory, to prepare for a gentle landing. The optimal strategy for this task is to leave the thruster switched off for as long as possible in the early stages of the journey, so as to gain as much downward speed as possible and hence lose as much potential energy as possible, and at the end of the journey produce a burst of continuous maximum thrust to reduce the kinetic energy as much as possible.

In addition to the cost received at termination by equation 14, a cost is also given for each non-terminal step. This cost is

$$U(\vec{x}, u) = c \left( \frac{1}{2} \ln(2 - 2u) - u \operatorname{arctanh}(1 - 2u) \right) \Delta t \quad (15)$$

where  $c = 0.02$  is constant. This cost function is designed to ensure that the actions chosen will satisfy  $u \in [0, 1]$ , even if a greedy policy is used. We explain how this cost function was derived, and how it can be used in a greedy policy, in section III-C, but first we describe experiments that did not use a greedy policy.

A DHP-style critic,  $\tilde{G}(\vec{x}, \vec{v})$ , was provided by a fully connected MLP with 3 input units, 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 the identity function in the output layer. The input to the MLP was  $(h/1600, v/40, t/200)^T$  and the output gave  $\tilde{G}$  directly.

The action network was identical in design to the critic, except there was only one output node, and this had a logistic sigmoid function as its activation function. The output of the action network gave the spacecraft’s acceleration  $u$  directly.

The mass of the spacecraft used was  $m = 0.02$ . This kept the magnitude of  $J$  quite small so that no rescaling was needed on the critic’s output. In all of the experiments we made the trajectory always start from  $h = 1600$ ,  $v = -2$ , and used discount factor  $\gamma = 1$ .

Results using the actor-critic architecture and Algorithm 1 are given in figure 2 comparing the performance of VGL(1) and VGL(0) (DHP). Each curve shows algorithm performance averaged over 40 trials.

The graphs show that the VGL(1) algorithm produces a lower total cost  $J$  than the VGL(0) algorithm does, and does it faster. It is thought that this is because in this problem the major part of the total cost comes as a final impulse, so it is advantageous to have a long look-ahead (i.e. a high  $\lambda$  value) for fast and stable learning.

For the actor-critic learning we chose the learning rate of the actor to be high compared to the learning rate for the critic (i.e.  $\beta > \alpha$ ). This was to make the results comparable to those of a greedy policy which we try in the next section.

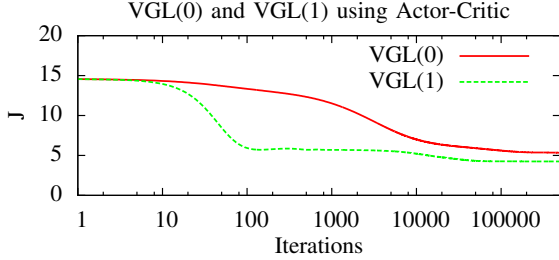


Fig. 2. VGL(0) (i.e. DHP) and VGL(1), with Actor-Critic, using learning rates  $\alpha = 10^{-6}$  and  $\beta = 0.01$ .

### C. Vertical Spacecraft Problem with Greedy Policy

The cost function of equation 15 was derived to form an efficient greedy policy, by following the method of [7]. First we chose the sigmoid function  $g(x)$  that we would like the greedy policy of equation 9 to use. This was chosen to be

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

The choice of  $c$  affects the sharpness of this sigmoid function. Using this chosen sigmoid function, the cost function based on [7] is defined to be

$$U(\vec{x}, u) = \Delta t \int g^{-1}(u) du. \quad (16)$$

Note that solving this integral gives equation 15. Then to derive the greedy policy for this cost function, we make a first order Taylor series expansion of the  $\tilde{Q}(\vec{x}, \vec{u}, \vec{w})$  function (eq. 6) about the point  $\vec{x}$ :

$$\begin{aligned} \tilde{Q}(\vec{x}, \vec{u}, \vec{w}) &\approx U(\vec{x}, \vec{u}) + \gamma \left( \left( \frac{\partial \tilde{J}}{\partial \vec{x}} \right)^T (f(\vec{x}, \vec{u}) - \vec{x}) + \tilde{J}(\vec{x}, \vec{w}) \right) \\ &= U(\vec{x}, \vec{u}) + \gamma \left( \tilde{G}(\vec{x}, \vec{w}) \right)^T (f(\vec{x}, \vec{u}) - \vec{x}) + \gamma \tilde{J}(\vec{x}, \vec{w}) \end{aligned} \quad (17)$$

This approximation becomes exact in continuous time, i.e. in the limit as  $\Delta t \rightarrow 0$ . The greedy policy must minimise  $\tilde{Q}$ , hence we differentiate equation 17 to get

$$\left( \frac{\partial \tilde{Q}}{\partial u} \right)_t = \left( \frac{\partial U}{\partial u} \right)_t + \gamma \left( \frac{\partial f}{\partial u} \right)_t \tilde{G}_t \quad \text{by eq. 17}$$

$$= g^{-1}(u_t) \Delta t + \gamma \left( \frac{\partial f}{\partial u} \right)_t \tilde{G}_t \quad \text{by eq. 16}$$

For a minimum, we must have  $\frac{\partial \tilde{Q}}{\partial u} = 0$ , which, since  $\frac{\partial f}{\partial u}$  is independent of  $u$ , gives  $u_t = g \left( -\frac{\gamma}{\Delta t} \left( \frac{\partial f}{\partial u} \right)_t \tilde{G}_t \right)$ . This is a variation on equation 9, and we used it as the greedy policy for the experiments in Figure 3. The results show similar *relative* performance of VGL(1) versus VGL(0) as in the actor-critic experiments, and both algorithms are faster than when an actor was used. This indicates that in this experiment, the greedy policy derived can successfully replace the action network, raising efficiency, and without any apparent detriment.

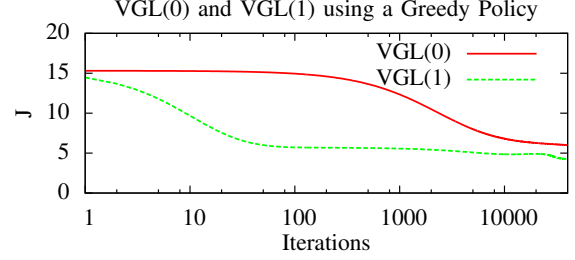


Fig. 3. VGL(0) (i.e. DHP) and VGL(1), with a greedy policy, using a learning rate  $\alpha = 10^{-6}$ .

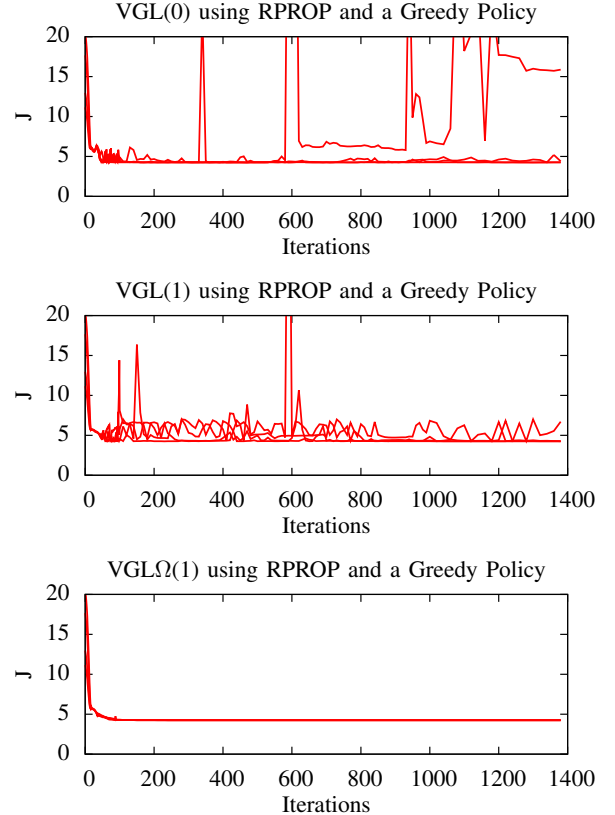


Fig. 4. VGL(0), VGL(1) and VGL $\Omega$ (1), with a greedy policy, using RPROP. Each graph shows the performance of a learning algorithm for each of five different weight initialisations; hence the ensemble of curves in each graph gives some idea of an algorithm's reliability and volatility.

Using a greedy policy, there are no longer two mutually



interacting neural networks whose training could be interfering with each other. With the simpler architecture of just one neural network (the critic) to contend with, we attempt to speed up learning using RPROP [27]. Results as shown in figure 4. It seems the aggressive acceleration by RPROP can cause large instability in the VGL(1) and DHP algorithms. This is because neither of these two algorithms is true gradient descent when used with a greedy policy [16]. However when the  $\Omega_t$  matrix defined by equation 5 is used with  $\lambda = 1$ , giving the algorithm that we will refer to as VGL $\Omega$ (1), the resulting algorithm is true gradient descent. It is gradient descent on  $J$ , as proven by [11]. The performance of this algorithm is shown in the bottom graph of Figure 4, and this shows the minimum being reached stably and many times quicker than in the actor-critic or non-RPROP case.

This represents a significant breakthrough in making learning with a greedy policy exhibit reliable and monotonic progress. The  $\Omega_t$  equation that achieves this is only proven to work for VGL(1), and counterexamples exist for its use with DHP [16].

#### IV. CONCLUSIONS

We have defined the VGL( $\lambda$ ) algorithm and explained its relationship to its precursor algorithms DHP and TD( $\lambda$ ). VGL( $\lambda$ ) can be viewed as a differentiated form of TD( $\lambda$ ); whereas TD methods learn values, VGL methods learn value-gradients. VGL( $\lambda$ ) extends the DHP algorithm by introducing a bootstrapping parameter,  $\lambda$ , which can affect learning speed and stability.

We have described the motivations for using VGL based methods (including DHP) in comparison to VL methods. These are that local exploration is automatic; VGL methods can be many times faster than VL methods; and VGL methods work naturally in continuous state spaces. The experiments confirmed these motivations, showing success for VGL in environments when no exploration is used and where VL methods fail. The experiments also demonstrate that VGL methods can be many times faster than VL methods. However, unlike VL methods, VGL methods require that the model functions are differentiable, and known or learnable.

Learning value-gradients is theoretically motivated since value-gradients can drive a greedy policy (e.g. as in equation 9), and the greedy policy equation (in the form of equation 1) must be satisfied if Bellman's Optimality Principle is to apply; so explicitly learning value-gradients is a very direct way to achieve optimal trajectories, without the need for local exploration.

The experiments demonstrated that when the special  $\Omega_t$  matrix of equation 5 is used, then the VGL $\Omega$ (1) algorithm can produce very stable learning with a greedy policy. This is a proven convergent critic learning algorithm, under certain smoothness assumptions, with a general function approximator and a greedy policy.

#### REFERENCES

- [1] F.-Y. Wang, H. Zhang, and D. Liu, "Adaptive dynamic programming: An introduction," *IEEE Computational Intelligence Magazine*, pp. 39–47, 2009.
- [2] R. E. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1957.
- [3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts, USA: The MIT Press, 1998.
- [4] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, 1988.
- [5] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, Cambridge University, 1989.
- [6] C. Kwok and D. Fox, "Reinforcement learning for sensing strategies," in *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, 2004.
- [7] K. Doya, "Reinforcement learning in continuous time and space," *Neural Computation*, vol. 12, no. 1, pp. 219–245, 2000.
- [8] P. J. Werbos, "Approximating dynamic programming for real-time control and neural modeling," *Handbook of Intelligent Control, editors White and Sofge, Chapter 13*, pp. 493–525, 1992.
- [9] S. Ferrari and R. F. Stengel, "Model-based adaptive critic designs," *Handbook of learning and approximate dynamic programming, editors Jennie Si et al.*, pp. 65–96, 2004.
- [10] D. Prokhorov and D. Wunsch, "Adaptive critic designs," *IEEE Transactions on Neural Networks*, vol. September, pp. 997–1007, 1997.
- [11] 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.
- [12] L. S. Pontryagin, V. G. Boltayanskii, R. V. Gamkrelidze, and E. F. Mishchenko, *The Mathematical Theory of Optimal Processes (Translated from Russian)*. Wiley, 1962, vol. 4.
- [13] 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, 2012.
- [14] J. N. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control*, Tech. Rep., 1996.
- [15] P. J. Werbos, "Stable adaptive control using new critic designs," *eprint arXiv:adapt-org/9810001*, 1998.
- [16] M. Fairbank and E. Alonso, "The divergence of reinforcement learning algorithms with value-iteration and function approximation," in *Proceedings of the IEEE International Joint Conference on Neural Networks 2012 (IJCNN'12)*. IEEE Press, 2012.
- [17] P. J. Werbos, "Backpropagation through time: What it does and how to do it," in *Proceedings of the IEEE*, vol. 78, No. 10, 1990, pp. 1550–1560.
- [18] —, "Neural networks, system identification, and control in the chemical process industries," *Handbook of Intelligent Control, editors White and Sofge, Chapter 10*, pp. 283–356, 1992.
- [19] A. Y. Ng, H. J. Kim, M. I. Jordan, and S. Sastry, "Inverted autonomous helicopter flight via reinforcement learning," in *International Symposium on Experimental Robotics*. MIT Press, 2004.
- [20] R. Munos, "Policy gradient in continuous time," *Journal of Machine Learning Research*, vol. 7, pp. 413–427, 2006.
- [21] G. K. Venayagamoorthy and D. C. Wunsch, "Dual heuristic programming excitation neurocontrol for generators in a multimachine power system," *IEEE Transactions on Industry Applications*, vol. 39, pp. 382–394, 2003.
- [22] G. G. Lendaris and C. Paintz, "Training strategies for critic and action neural networks in dual heuristic programming method," in *Proceedings of International Conference on Neural Networks, Houston, 1997*.
- [23] M. Fairbank, "Reinforcement learning by value gradients," *eprint arXiv:0803.3539*, 2008.
- [24] B. A. Pearlmutter, "Fast exact multiplication by the Hessian," *Neural Computation*, vol. 6, no. 1, pp. 147–160, 1994.
- [25] 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.
- [26] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.
- [27] 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*, San Francisco, CA, 1993, pp. 586–591.