An Equivalence between Adaptive Dynamic Programming with a Critic and Backpropagation Through Time

Michael Fairbank, *Student Member, IEEE*, Eduardo Alonso and Danil Prokhorov, *Senior Member, IEEE*

(To appear in *IEEE Transactions on Neural Networks and Learning Systems*, Vol.??, Issue ??, October 201?, pages ?–?. Copyright the authors and IEEE, http://ieeexplore.ieee.org.)

***Abstract*—We consider the adaptive dynamic programming technique called Dual Heuristic Programming (DHP), which is designed to learn a critic function, when using learned model functions of the environment. DHP is designed for optimising control problems in large and continuous state spaces.**

We extend DHP into a new algorithm that we call VGL(λ), and prove equivalence of an instance of the new algorithm to backpropagation through time for control with a greedy policy. Not only does this equivalence provide a link between these two different approaches, but it also enables our variant of DHP to have guaranteed convergence, under certain smoothness conditions and a greedy policy, when using a general smooth non-linear function approximator for the critic.

We consider several experimental scenarios including some which prove divergence of DHP under a greedy policy, which contrasts against our proven-convergent algorithm.

***Index Terms*—Adaptive Dynamic Programming, Dual Heuristic Programming, Value-Gradient Learning, Backpropagation Through Time, Neural Networks**

I. Introduction

ADAPTIVE Dynamic Programming (ADP) [1] is the study of how an agent can learn actions that minimise a long-term cost function. For example a typical scenario is an agent moving in a state space, $\mathbb{S} \subset \mathbb{R}^n$, such that at time t it has state vector $\vec{x}_t$. At each time $t \in \mathbb{Z}^+$ the agent chooses an action $\vec{u}_t$ from an action space $\mathbb{A}$, which takes it to the next state according to the environment's model function

$$\vec{x}_{t+1} = f(\vec{x}_t, \vec{u}_t), \tag{1}$$

and gives it an immediate *cost* or *utility*, U_t, given by the function $U_t = U(\vec{x}_t, \vec{u}_t)$. The agent keeps moving, forming a trajectory of states $(\vec{x}_0, \vec{x}_1, \ldots)$, 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*, $A(\vec{x}, \vec{z})$, is the output of a smooth approximated function, e.g. the output of a neural network with parameter vector $\vec{z}$. The action network, which is also known as the *actor* or *policy*, assigns an action

$$\vec{u}_t = A(\vec{x}_t, \vec{z}) \tag{2}$$

to take for any given state $\vec{x}_t$.

For a trajectory starting from $\vec{x}_0$ derived by following (1) and (2), the total trajectory cost is given by the *cost-to-go* function, or *value function*, which is

$$J(\vec{x}_0, \vec{z}) = \left\langle \sum_t \gamma^t U(\vec{x}_t, \vec{u}_t) \right\rangle .$$

Here $\langle\rangle$ denotes expectation, and $\gamma \in [0, 1]$ is a constant *discount factor* that specifies the importance of long term costs over short term ones. The objective of ADP is to train the action network such that $J(\vec{x}_0, \vec{z})$ is minimised from any $\vec{x}_0$.

ADP also uses a second approximated function, the scalar valued *critic network*, $\widetilde{J}(\vec{x}, \vec{w})$. This is the output of a smooth general scalar function approximator, e.g. a neural network with a single output node and weight vector $\vec{w}$. The objective of training the critic network is for it act as a good approximation of the cost-to-go function, i.e. so that $\widetilde{J}(\vec{x}, \vec{w}) \approx J(\vec{x}, \vec{z})$ for all states $\vec{x}$.

For any given critic network, the *greedy policy* is a policy which always chooses actions that lead to states that the critic function rates as best (whilst also taking into account the immediate short term utility in getting there), i.e. a greedy policy chooses actions according to

$$\vec{u} = \arg\min_{\vec{u} \in \mathbb{A}} \left\langle U(\vec{x}, \vec{u}) + \gamma \widetilde{J}(f(\vec{x}, \vec{u}), \vec{w}) \right\rangle \quad \forall \vec{x}. \tag{3}$$

When a critic and action network are used together, the objective is to train the action network to be greedy with respect to the critic (i.e. the action network must choose actions $\vec{u}_t = A(\vec{x}_t, \vec{z})$ that satisfy (3)), and to train the critic to approximate the cost-to-go function for the current action network. If these two objectives can be met simultaneously, and exactly, for all states, then Bellman's Optimality Condition [2] will be satisfied, and the ADP objective of optimising the cost-to-go function from any start state will be achieved.

We follow the methods of Dual Heuristic Programming (DHP) and Globalized DHP (GDHP) [1], [3]–[6]. DHP and GDHP work by explicitly learning the gradient of the cost-to-go 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), to distinguish them from the usual direct updates to the values of the cost-to-go function, which we refer to as *value-learning* methods.

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

D. Prokhorov is with Toyota Research Institute NA, Ann Arbor, Michigan (e-mail: dvprokhorov@gmail.com).

We extend the VGL methods to include a bootstrapping parameter λ to give the algorithm we call VGL(λ) [7], [8]. This is directly analogous to how the reinforcement learning (RL) algorithm TD(λ) is extended from TD(0) [9]. This extension was desirable because, for example, choosing $\lambda = 1$ can lead to increased stability in learning, and guaranteeing stably convergent algorithms is a serious issue for all critic learning algorithms. Also, setting a high value of λ can increase the "look-ahead" of the training process for the critic, which can lead to faster learning in long trajectories.

The VGL methods work very efficiently in continuous domains such as autopilot landing [4], power system control [10], simple control benchmark problems such as "pole balancing" [11], and many others [1]. It turns out to be only necessary to learn the value gradient along a single trajectory, while also satisfying the greedy policy condition, for the trajectory to be locally extremal, and often locally optimal. This is for reasons closely related to Pontryagin's minimum principle [12], as proven by [13]. This is a significant efficiency gain for the VGL methods and is their principal motivation. In contrast, value-learning methods must learn the values all along the trajectory and over all neighbouring trajectories too to achieve the same level of assurance of local optimality [14].

All VGL methods are model-based methods. We assume the functions f and U each consist of a differentiable part plus, optionally, an additive noise part, and we assume they can be learned by a separate "system identification" learning process, for example as described by [15]. This system identification process could have taken place prior to the main learning process, or concurrently with it. From now on in this paper, we assume $f(\vec{x},\vec{u})$ and $U(\vec{x},\vec{u})$ refer to the learned, differentiable model functions; and it is with respect to these two learned functions that we are seeking to find optimal solutions. Using a neural network to learn these two functions would enforce the required smoothness conditions onto these two functions.

Proving convergence of ADP algorithms is a challenging problem. References [5], [16], [17] show the ADP process will converge to optimal behaviour if the critic could be perfectly learned over all of state space at each iteration. However in reality we must work with a function approximator for the critic with finite capabilities, so this assumption is not valid. Variants of DHP are proven to converge by [18], [19], the first of which uses a linear function approximator for the critic which can be fully trained in a single step, and the second of which is based on the Galerkin-based form of DHP [20], [21]. We do not consider the Galerkin-based methods (which are also known as *residual-gradient* methods by [21]) for reasons given by [22] and [7, sec 2.3]. Working with a general quadratic function approximator, [20, sec.7.7-7.8] proves the general instability of DHP and GDHP. This analysis was for a fixed action network, so with a greedy policy convergence would presumably seem even less likely. In this paper, we show a specific divergence example for DHP with a greedy policy in Section IV.

One reason that the convergence of these methods is difficult to assure is that in the Bellman condition, there is an interdependence between $J(\vec{x},\vec{z})$, $A(\vec{x},\vec{z})$ and $\widetilde{J}(\vec{x},\vec{w})$. We make an important insight into this difficulty by showing (in Lemma 4) that the dependency of a greedy policy on the critic is primarily through the value-gradient.

In this paper, using a method first described in our earlier technical reports [7], [13], we show the VGL(λ) weight update, with $\lambda = 1$ and some learning constants chosen carefully, is identical to the application of backpropagation through time (BPTT) [23] to a greedy policy. This makes a theoretical connection between two seemingly different learning paradigms, and provides a convergence proof for a critic learning ADP method, with a general smooth function approximator and a greedy policy.

In the rest of this paper, in Section II we define the VGL(λ) algorithm, state its relationship to DHP, and give pseudocode for it. Section III is the main contribution of this paper, in which we describe BPTT and the circumstances of its equivalence to VGL(λ) with a greedy policy, and we describe how this equivalence forms a convergence proof for VGL(λ) under these circumstances. In Section IV, we provide an example ADP problem which we use to make a confirmation of the equivalence proof, and which we also use to demonstrate how critic learning methods with a greedy policy can be made to diverge; in contrast to the proven convergent algorithm. In Section V we describe two neural-network based experiments, and give conclusions in Section VI.

II. The VGL(λ) and DHP Algorithms

All VGL algorithms, including DHP, GDHP and VGL(λ), attempt to learn the value-gradient, $\frac{\partial J}{\partial \vec{x}}$. This gradient is learned by a vector critic function $\widetilde{G}(\vec{x},\vec{w})$ which has the same dimension as $\dim(\vec{x})$. In the case of DHP this function is implemented by the output of a smooth vector function approximator, and in GDHP it is implemented as $\widetilde{G}(\vec{x},\vec{w}) \equiv \frac{\partial \widetilde{J}(\vec{x},\vec{w})}{\partial \vec{x}}$, i.e. the actual gradient of the smooth approximated scalar function $\widetilde{J}(\vec{x},\vec{w})$. For the VGL($\lambda$) algorithm, we can use either of these two representations for $\widetilde{G}$.

We assume the action network $A(\vec{x},\vec{z})$ is differentiable with respect to all of its arguments, and similar differentiability conditions apply to the model and cost functions.

Throughout this paper, subscripted t variables attached to a function name indicates that all arguments of the function are to be evaluated at time step t of a trajectory. This is what we call *trajectory shorthand notation*. For example $\widetilde{J}_{t+1} \equiv \widetilde{J}(\vec{x}_{t+1},\vec{w})$, $U_t \equiv U(\vec{x}_t,\vec{u}_t)$ and $\widetilde{G}_t \equiv \widetilde{G}(\vec{x}_t,\vec{w})$.

A convention is also used that all defined vector quantities are columns, whether they are coordinates, or derivatives with respect to coordinates. For example, both $\vec{x}_t$ and $\frac{\partial \widetilde{J}}{\partial \vec{x}}$ are columns. Also any vector function becomes transposed (becoming a row) if it appears in the numerator of a differential. For example $\frac{\partial f}{\partial \vec{x}}$ is a matrix with element (i,j) equal to $\frac{\partial f(\vec{x},\vec{u})^j}{\partial \vec{x}^i}$; $\frac{\partial \widetilde{G}}{\partial \vec{w}}$ is a matrix with element (i,j) equal to $\frac{\partial \widetilde{G}^j}{\partial \vec{w}^i}$; and $\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t$ is the matrix $\frac{\partial \widetilde{G}}{\partial \vec{w}}$ evaluated at $(\vec{x}_t,\vec{w})$. This is the transpose of the common convention for Jacobians.

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

$$\Delta\vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t ({G'}_t - \widetilde{G}_t) \tag{4}$$

where α is a small positive learning rate; Ω_t is an arbitrary positive definite matrix described further below; $\widetilde{G}_t$ is the output of the vector critic at time step t; and ${G'}_t$ is the "target value gradient" defined recursively by:

$$
{G'}_t = \begin{cases} \left(\frac{DU}{D\vec{x}}\right)_t + \gamma \left(\frac{Df}{D\vec{x}}\right)_t \left(\lambda {G'}_{t+1} + (1-\lambda)\widetilde{G}_{t+1}\right) & \text{if } \vec{x}_t \notin \mathbb{T} \\ \left(\frac{\partial U}{\partial \vec{x}}\right)_t & \text{if } \vec{x}_t \in \mathbb{T} \end{cases} \tag{5}
$$

where $\lambda \in [0,1]$ is a fixed constant, analogous to the λ used in TD(λ); and where $\frac{D}{D\vec{x}}$ is shorthand for

$$
\frac{D}{D\vec{x}} \equiv \frac{\partial}{\partial \vec{x}} + \frac{\partial A}{\partial \vec{x}} \frac{\partial}{\partial \vec{u}}; \tag{6}
$$

and where all of these derivatives are assumed to exist. All terms of (5) are obtainable from knowledge of the model functions and the action network. We ensure the recursion in (5) converges by requiring that either $\gamma < 1$, or $\lambda < 1$, 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").

The objective of the weight update is to make the values $\widetilde{G}_t$ move towards the target values ${G'}_t$. This weight update needs to be done slowly because the targets G' are heavily dependent on $\vec{w}$, so are moving targets. This dependency on $\vec{w}$ is especially great if the policy is greedy or if the action network is concurrently trained to try to keep the trajectory greedy, so that the policy is then also indirectly dependent on $\vec{w}$. If the weight update succeeds in achieving $\widetilde{G}_t = {G'}_t$, and the greedy action condition (3), for all t along a trajectory, then this will ensure the trajectory is locally extremal, and often locally optimal (as proven by [13]).

The matrix Ω_t was introduced by Werbos in the GDHP algorithm (e.g. [20, eq.32]), 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). For simplicity Ω_t is often just taken to be the identity matrix for all t, so effectively has an optional presence. One use for making Ω_t arbitrary could be for the experimenter to be able to compensate explicitly for any rescalings of the state space axes. We make use of it in Section III, when proving how VGL(λ) can be equivalent to BPTT.

Equations (4), (5) and (6) define the VGL(λ) algorithm. Algorithm 1 gives pseudocode for it in a form that can be applied to complete trajectories. This algorithm runs in time $O(\dim(\vec{w}))$ per time step of the trajectory. An on-line version is also possible that can be applied to incomplete trajectories [8].

The algorithm does not attempt 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 $\widetilde{G}(\vec{x}, \vec{w})$ has been designed to explicitly return $\frac{\partial U}{\partial \vec{x}}$ for all terminal states $\vec{x} \in \mathbb{T}$.

A. Relationship of the VGL(λ) Algorithm to DHP and GDHP

The VGL(λ) algorithm is an extension of the DHP algorithm, made by introducing a λ parameter as defined in

Algorithm 1 VGL(λ). 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: $t_f \leftarrow t$
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 $t = t_f - 1$ 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:     $\Delta\vec{w} \leftarrow \Delta\vec{w} + \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t \left({G'}_t - \widetilde{G}_t\right)$
14:     $\Delta\vec{z} \leftarrow \Delta\vec{z} - \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial U}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}\right)$
15:     $\vec{p} \leftarrow \lambda {G'}_t + (1-\lambda)\widetilde{G}_t$
16: end for
17: $\vec{w} \leftarrow \vec{w} + \alpha\Delta\vec{w}$
18: $\vec{z} \leftarrow \vec{z} + \beta\Delta\vec{z}$
```

(5). This λ parameter affects the exponential decay rate of the look-ahead used in (5). In this equation, when $\lambda = 0$, the target value gradient G' becomes equivalent to the target used in DHP. Hence when $\lambda = 0$, and when $\widetilde{G}(\vec{x}, \vec{w})$ is implemented as the vector output of a function approximator, VGL(λ) becomes identical to DHP.

When VGL(λ) is implemented with $\lambda = 0$ and $\widetilde{G}(\vec{x}, \vec{w})$ is defined to be $\frac{\partial \widetilde{J}}{\partial \vec{x}}$, for a scalar function approximator $\widetilde{J}(\vec{x}, \vec{w})$, it becomes equivalent to an instance of GDHP. GDHP is more general than VGL(0) since its weight update is defined to also include a value-learning component too, i.e. GDHP is a linear combination of the VGL(0) weight update plus a Heuristic Dynamic Programming ([3], [4]) weight update.

B. Action Network Weight Update

To solve the ADP problem, the action network also needs training. The objective of the action network's weight update is to make the actor more greedy, i.e. to behave more closely to the following objective:

$$
\vec{u} = \arg\min_{\vec{u} \in \mathbb{A}} (\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})) \quad \forall \vec{x} \tag{7}
$$

where we define the approximate Q Value function as

$$
\widetilde{Q}(\vec{x}, \vec{u}, \vec{w}) = U(\vec{x}, \vec{u}) + \gamma \widetilde{J}(f(\vec{x}, \vec{u}), \vec{w}), \tag{8}
$$

and which is consistent with (3).

Hence the actor weight update used most commonly in ADP is gradient descent on the $\widetilde{Q}$ function with respect to $\vec{z}$. This is implemented by lines 14 and 18 of Algorithm 1. Here $\beta > 0$ is a separate learning rate for the action network.

In practice the actor weight update can be done concurrently with the critic weight update, as in Algorithm 1; or learning can consist of alternating phases of one or more actor weight updates followed by one or more critic weight updates.

C. Greedy Policy

In some implementations the action network can be efficiently replaced by a greedy policy, which is a function that directly solves (7). Since the greedy policy (7) is dependent on the weight vector $\vec{w}$ of the critic function, we will denote it as $\pi(\vec{x}, \vec{w})$ to distinguish it from a general action network $A(\vec{x}, \vec{z})$. When a greedy policy is used, all occurrences of $A(\vec{x}, \vec{z})$ in the VGL(λ) algorithm would be replaced by $\pi(\vec{x}, \vec{w})$, and the actor weight update equation lines 14 and 18 of Algorithm 1 would be removed.

A greedy policy is only possible when the right-hand side of (7) is efficient to compute, which is possible in the continuous time situations described by [24] or [7, section 2.2], and in the "single network adaptive critic" described by [18]. We give an example of this kind of greedy policy in Section V-B.

D. The Relationship of an Action Network to a Greedy Policy

The main results of this paper apply to a greedy policy: convergence is proven for a greedy policy (in Section III), and so is the divergence of other algorithms (in Section IV). However in certain circumstances, these results can partially apply to an action network too. Since the action network's weight update is gradient descent on (8), the intention of it is to make the action network behave more like a greedy policy. Hence when the action network is trained to completion in between every single critic weight update (a situation known as *value-iteration*), then the action network will be behaving very much like a greedy policy. Hence, if the action network has sufficient flexibility to learn the greedy policy accurately enough, then the convergence/divergence results of this paper would apply to it.

III. The Relationship of VGL to BPTT

We now prove that the VGL(λ) weight update of (4), with $\lambda = 1$ and a carefully chosen Ω_t matrix, is equivalent to backpropagation through time (BPTT) on a greedy policy. First we derive the equations for BPTT (in Section III-A), then we describe some lemmas about a greedy policy (in Section III-B), and then we demonstrate that when BPTT is applied to a greedy policy, the weight update obtained is an instance of VGL(λ) (in Section III-C). Finally, in Section III-D, we discuss the consequences of the results and the convergence properties.

A. Backpropagation Through Time for Control Problems

BPTT can be used to calculate the derivatives of any differentiable scalar function with respect the weights of a neural network. To apply BPTT in solving a control problem, it can be used to find the derivatives of $J(\vec{x}_0, \vec{z})$ with respect to $\vec{z}$, so as to enable gradient descent on J.

Hence the gradient-descent weight update is $\Delta\vec{z} = -\alpha \left(\frac{\partial J}{\partial \vec{z}}\right)_0$ for some small positive learning rate α. Gradient descent will naturally find local minima of $J(\vec{x}_0, \vec{z})$, and has good convergence properties when the surface $J(\vec{x}_0, \vec{z})$ is smooth with respect to $\vec{z}$.

The total discounted cost for a trajectory $J(\vec{x}_0, \vec{z}) = \sum_t \gamma^t U_t$ can be written recursively as

$$J(\vec{x}, \vec{z}) = U(\vec{x}, A(\vec{x}, \vec{z})) + \gamma J(f(\vec{x}, A(\vec{x}, \vec{z})), \vec{z}) \tag{9}$$

with $J(\vec{x}, \vec{z}) = U(\vec{x})$ at any terminal state $\vec{x} \in \mathbb{T}$.

To calculate the gradient of (9) with respect to $\vec{z}$, we differentiate using the chain rule, and substitute (1) and (2):

$$\left(\frac{\partial J}{\partial \vec{z}}\right)_t = \left(\frac{\partial}{\partial \vec{z}}(U(\vec{x}, A(\vec{x}, \vec{z})) + \gamma J(f(\vec{x}, A(\vec{x}, \vec{z})), \vec{z}))\right)_t$$

$$= \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial U}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{u}}\right)_t \left(\frac{\partial J}{\partial \vec{x}}\right)_{t+1}\right) + \gamma \left(\frac{\partial J}{\partial \vec{z}}\right)_{t+1}$$

Expanding this recursion and substituting it into the gradient-descent equation gives,

$$\Delta\vec{z} = -\alpha \sum_{t \geq 0} \gamma^t \left(\frac{\partial A}{\partial \vec{z}}\right)_t \left(\left(\frac{\partial U}{\partial \vec{u}}\right)_t + \gamma \left(\frac{\partial f}{\partial \vec{u}}\right)_t \left(\frac{\partial J}{\partial \vec{x}}\right)_{t+1}\right) \tag{10}$$

This weight update is BPTT with gradient-descent to minimise $J(\vec{x}_0, \vec{z})$ with respect to the weight vector $\vec{z}$ of an action network $A(\vec{x}, \vec{z})$. It refers to the quantity $\frac{\partial J}{\partial \vec{x}}$ which can be found recursively by differentiating (9) and using the chain rule, giving

$$\left(\frac{\partial J}{\partial \vec{x}}\right)_t = \begin{cases} \left(\frac{DU}{D\vec{x}}\right)_t + \gamma \left(\frac{Df}{D\vec{x}}\right)_t \left(\frac{\partial J}{\partial \vec{x}}\right)_{t+1} & \text{if } \vec{x}_t \notin \mathbb{T} \\ \left(\frac{\partial U}{\partial \vec{x}}\right)_t & \text{if } \vec{x}_t \in \mathbb{T}. \end{cases} \tag{11}$$

Equation (11) can be understood to be backpropagating the quantity $\left(\frac{\partial J}{\partial \vec{x}}\right)_{t+1}$ through the action network, model and cost functions to obtain $\left(\frac{\partial J}{\partial \vec{x}}\right)_t$, and giving the BPTT algorithm its name.

By comparing (11) with (5), we note that

$$G' \equiv \frac{\partial J}{\partial \vec{x}}, \qquad \text{when } \lambda = 1. \tag{12}$$

B. Lemmas about a Greedy Policy and Greedy Actions

To prepare for the later analysis of BPTT applied to a greedy policy, first we prove some lemmas about the greedy policy. These lemmas apply when the action space, $\mathbb{A}$, is equal to $\mathbb{R}^{\dim(\vec{u})}$, which we will denote as $\mathbb{A}^*$.

A greedy policy $\pi(\vec{x}, \vec{w})$ is a policy that always selects actions $\vec{u}$ that are the minimum of the smooth function $\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})$ defined by (8). These minimising actions are what we call *greedy actions*. In this case, since the minimum of a smooth function is found from an unbound domain, $\vec{u} \in \mathbb{R}^{\dim(\vec{u})}$, the following two consequences hold:

Lemma 1: For a greedy action $\vec{u}$ chosen from $\mathbb{A}^*$, we have $\frac{\partial \widetilde{Q}}{\partial \vec{u}} = \vec{0}$.

Lemma 2: For a greedy action $\vec{u}$ chosen from $\mathbb{A}^*$, $\frac{\partial^2 \widetilde{Q}}{\partial \vec{u} \partial \vec{u}}$ is a positive semi-definite matrix.

Note that the above two lemmas are multi-dimensional analogues of the familiar minimum conditions for a one-dimensional function $q(u) : \mathbb{R} \to \mathbb{R}$ with an unbound domain, which are $q'(u) = 0$ and $q''(u) \geq 0$, respectively.

We now prove too less obvious lemmas about a greedy policy:

Lemma 3: The greedy policy on $\mathbb{A}^*$ implies $\left(\frac{\partial U}{\partial \vec{u}}\right)_t = -\gamma\left(\frac{\partial f}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1}$.

Proof: First, we note that differentiating (8) gives

$$\left(\frac{\partial \widetilde{Q}}{\partial \vec{u}}\right)_t = \left(\frac{\partial U}{\partial \vec{u}}\right)_t + \gamma\left(\frac{\partial f}{\partial \vec{u}}\right)_t \widetilde{G}_{t+1} \tag{13}$$

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

Lemma 4: When $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ and $\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u}\partial \vec{u}}\right)_t^{-1}$ exist, the greedy policy on $\mathbb{A}^*$ implies

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

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

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

In the above six lines of algebra, the sum of the two partial derivatives in line 2 follows by the chain rule from the total derivative in line 1, since $\vec{w}$ appears twice in line 1. This step has also made use of $\vec{u}_t = \pi(\vec{x}_t, \vec{w})$. Also note that the first term in line 2 is not zero, despite the greedy policy's requirement for $\frac{\partial \widetilde{Q}}{\partial \vec{u}} \equiv 0$, since in this term the $\vec{u}$ and $\vec{w}$ are now treated as independent variables. Then in the remaining lines, line 3 is by (13); line 4 just expands an inner product; line 5 follows since $\frac{\partial U}{\partial \vec{u}}$ and $\frac{\partial f}{\partial \vec{u}}$ 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. ■

C. The equivalence of VGL(1) to BPTT

In Section III-A, the equation for gradient descent on $J(\vec{x}, \vec{z})$ was found for a general policy $A(\vec{x}, \vec{z})$, using BPTT. But BPTT can be applied to any policy, and so we now consider what would happen if BPTT is applied to the greedy policy $\pi(\vec{x}, \vec{w})$, with actions chosen from $\mathbb{A}^*$. The parameter vector for the greedy policy is $\vec{w}$. Hence we can do gradient descent with respect to $\vec{w}$ instead of $\vec{z}$ (assuming the derivatives $\frac{\partial \pi}{\partial \vec{w}}$ and $\frac{\partial J}{\partial \vec{w}}$ exist). We should emphasise that with the greedy policy, it is the same weight vector that appears in the critic, $\vec{w}$, as appears in the greedy policy $\pi(\vec{x}, \vec{w})$.

Consequently, for the gradient-descent equation in BPTT for control (10), we now change all instances of A and $\vec{z}$ to π and $\vec{w}$, respectively, giving the new weight update:

$$\Delta\vec{w} = -\alpha\sum_{t\geq 0}\gamma^t\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t\left(\left(\frac{\partial U}{\partial \vec{u}}\right)_t + \gamma\left(\frac{\partial f}{\partial \vec{u}}\right)_t\left(\frac{\partial J}{\partial \vec{x}}\right)_{t+1}\right)$$

Substituting Lemmas 3 and 4, and $\left(\frac{\partial J}{\partial \vec{x}}\right)_t \equiv G'_t$ with $\lambda = 1$ (by (12)), into this gives:

$$\begin{aligned}
\Delta\vec{w} =& -\alpha\sum_{t\geq 0}\gamma^t\Bigg[-\gamma^2\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_{t+1}\left(\frac{\partial f}{\partial \vec{u}}\right)_t^T\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u}\partial \vec{u}}\right)_t^{-1}\\
&\left(\frac{\partial f}{\partial \vec{u}}\right)_t(-\widetilde{G}_{t+1} + G'_{t+1})\Bigg]\\
=& \alpha\sum_{t\geq 0}\gamma^{t+1}\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t\Omega_t(G'_t - \widetilde{G}_t)
\end{aligned} \tag{14}$$

where

$$\Omega_t = \begin{cases}\left(\frac{\partial f}{\partial \vec{u}}\right)_{t-1}^T\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u}\partial \vec{u}}\right)_{t-1}^{-1}\left(\frac{\partial f}{\partial \vec{u}}\right)_{t-1} & \text{for } t > 0\\ 0 & \text{for } t = 0\end{cases}, \tag{15}$$

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

Equation (14) is identical to a VGL weight update equation (4), with a carefully chosen matrix for Ω_t, and $\gamma = \lambda = 1$, provided $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ and $\left(\frac{\partial^2 \widetilde{Q}}{\partial \vec{u}\partial \vec{u}}\right)_t^{-1}$ exist for all t. If $\left(\frac{\partial \pi}{\partial \vec{w}}\right)_t$ does not exist, then $\frac{\partial J}{\partial \vec{w}}$ 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 (with a greedy policy with actions chosen from $\mathbb{A}^*$, and when $\frac{\partial J}{\partial \vec{w}}$ exists).

Furthermore, the presence of the γ^t factor in (14) could be removed if we changed the BPTT gradient-descent equation by removing the γ^t factor from (10). This would make the BPTT weight update more accurately follow the spirit and intention of the on-line critic weight update; and then the equivalence of VGL(1) to BPTT would hold for any γ too.

D. Discussion

BPTT is gradient descent on a function which is bounded below. Therefore, assuming the surface of $J(\vec{x}, \vec{z})$ in $\vec{z}$-space is sufficiently smooth and the step-size for gradient descent is sufficiently small, convergence of BPTT is guaranteed.

If the ADP problem is such that $\frac{\partial \pi}{\partial \vec{w}}$ always exists for a greedy policy, then the equivalence proof above shows that the good convergence guarantees of BPTT will apply to VGL(1) (when used with the special choice of Ω_t by (15)). In this case, this particular VGL algorithm will achieve monotonic progress with respect to J, and so will have guaranteed convergence, provided it is operating within a smooth region of the surface of the function J. Significantly, the requirement for $\frac{\partial \pi}{\partial \vec{w}}$ to always exist is satisfied when a value-gradient greedy-policy, of the kind used in our experiments in Section V-B, is used. The requirement for the surface of J in $\vec{z}$-space to be sufficiently smooth cannot so easily be guaranteed, but this situation is no different than the requirement for BPTT.

This equivalence result was surprising to the authors because it was thought that the VGL weight updates (equation (4), and DHP and GDHP) were based on gradient descent on an error function $E = \sum_t (G'_t - \widetilde{G}_t)^T \Omega_t (G'_t - \widetilde{G}_t)$. But as [21] showed, the TD(λ) weight update is not true gradient descent on its intended error function, and furthermore it is not gradient descent on any error function [25]. Similarly, nor is the VGL(λ) weight update true gradient descent on E (unless both the policy is fixed and $\lambda = 1$). Our proof shows that when a greedy policy is used, VGL(1) is closer to true gradient descent on J than the gradient on E. It was also surprising to learn that BPTT and critic weight updates are not as fundamentally different to each other as we first thought.

For a fuller discussion of the Ω_t matrix defined by (15), including methods for its computation and a discussion of its purpose and effectiveness, see reference [26].

IV. Example Analytical Problem

In this section we define an ADP problem which is simple enough to analyse algebraically. We define this problem and derive the VGL(λ) weight update algebraically for it in sections IV-A to IV-F. Then in Section IV-G we show that when the Ω_t matrix of (15) is used, we do get exact equivalence of VGL(1) to BPTT in the example problem, thus confirming the theoretical result of Section III. We also use the example problem to derive divergence instances for DHP and VGL(λ) without the special Ω_t matrices (in sections IV-H to IV-I), thus emphasising the value of the BPTT equivalence proof.

A. *Environment Definition*

We define an environment with state $x \in \mathbb{R}$ and action $u \in \mathbb{R}$, and with model and cost functions:

$$f(x_t, t, u_t) = x_t + u_t \qquad \text{for } t \in \{0,1\} \tag{16a}$$

$$U(x_t, t, u_t) = k(u_t)^2 \qquad \text{for } t \in \{0,1\} \tag{16b}$$

where $k > 0$ is a constant. Each trajectory is defined to terminate immediately on arriving at time step $t = 2$, when a final terminal cost of

$$U(x_t) = (x_t)^2 \tag{17}$$

is given, so that exactly three costs are received by the agent over the full trajectory duration. The termination condition is dependent on t, so strictly speaking t should be included in the state vector, but we have omitted this for brevity.

A whole trajectory is completely parametrised by x_0, u_0 and u_1, and the total cost is

$$J = k(u_0)^2 + k(u_1)^2 + (x_0 + u_0 + u_1)^2. \tag{18}$$

The examples we derive below consider a trajectory which starts at $x_0 = 0$. From this start point, the optimal actions are those that minimise J, i.e. $u_0 = u_1 = 0$.

B. *Critic Definition*

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

$$\widetilde{J}(x_t, t, \vec{w}) = \begin{cases} 0 & \text{if } t = 0 \\ c_1(x_1)^2 - w_1 x_1 & \text{if } t = 1 \\ c_2(x_2)^2 - w_2 x_2 & \text{if } t = 2 \end{cases} \tag{19}$$

where c_1 and c_2 are positive constants. These two constants are not to be treated as weights. We included them so that we could consider a greater range of function approximators for the critic when we searched for a divergence example, as described in Section IV-H. Also to ease the finding of that divergence example, we chose this simplified critic structure (as opposed to a neural network) since it is linear in $\vec{w}$, and its weight vector has just two components.

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

$$\widetilde{G}(x_t, t, \vec{w}) = \begin{cases} 0 & \text{if } t = 0 \\ 2c_t x_t - w_t & \text{if } t \in \{1,2\} \end{cases} \tag{20}$$

We note that this implies

$$\left(\frac{\partial \widetilde{G}}{\partial w_k}\right)_t = \begin{cases} -1 & \text{if } t \in \{1,2\} \text{ and } t = k \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

C. *Unrolling a greedy trajectory*

A *greedy trajectory* is a trajectory that is found by following greedy actions only. Greedy actions are $\vec{u}$ values that minimise $\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})$.

Substituting the model functions (16) and the critic definition (19) into the $\widetilde{Q}$ function definition (8) gives, with $\gamma = 1$,

$$\begin{aligned} &\widetilde{Q}(x_t, t, u_t, \vec{w}) \\ &= U(x_t, t, u_t) + \gamma \widetilde{J}(f(x_t, t, u_t), t+1, \vec{w}) && \text{by (8)} \\ &= k(u_t)^2 + \widetilde{J}(x_t + u_t, t+1, \vec{w}), \text{ for } t \in \{0,1\} && \text{by (16)} \\ &= \begin{cases} k(u_0)^2 + c_1(x_0 + u_0)^2 - w_1(x_0 + u_0) & \text{if } t = 0 \\ k(u_1)^2 + c_2(x_1 + u_1)^2 - w_2(x_1 + u_1) & \text{if } t = 1 \end{cases} && \text{by (19)} \\ &= k(u_t)^2 + c_{t+1}(x_t + u_t)^2 - w_{t+1}(x_t + u_t), \text{ for } t \in \{0,1\}. \end{aligned}$$

In order to minimise this with respect to u_t and get greedy actions, we first differentiate to get,

$$\begin{aligned} \left(\frac{\partial \widetilde{Q}}{\partial u}\right)_t &= 2ku_t + 2c_{t+1}(x_t + u_t) - w_{t+1} && \text{for } t \in \{0,1\} \\ &= 2u_t(c_{t+1} + k) - w_{t+1} + 2c_{t+1}x_t && \text{for } t \in \{0,1\} \end{aligned} \tag{22}$$

Hence the greedy actions are found by solving $\left(\frac{\partial \widetilde{Q}}{\partial u}\right)_t = 0$, to obtain,

$$u_0 \equiv \frac{w_1 - 2c_1 x_0}{2(c_1 + k)} \tag{23}$$

$$u_1 \equiv \frac{w_2 - 2c_2x_1}{2(c_2+k)} \tag{24}$$

and these two equations define the greedy policy function $\pi(\vec{x},\vec{w})$ for this environment and critic function.

Since the optimal actions are $u_0 = u_1 = 0$ from a start state of $x_0 = 0$, the optimal weights are $w_1 = w_2 = 0$.

Following the greedy actions along a trajectory starting at $x_0 = 0$, and using the recursion $x_{t+1} = f(x_t, u_t)$ with the model functions (16) gives

$$\begin{aligned} x_1 &= x_0 + u_0 && \text{by (16a)} \\ &= \frac{w_1}{2(c_1+k)} && \text{by (23) \& } x_0 = 0, \end{aligned} \tag{25}$$

and,

$$\begin{aligned} x_2 &= x_1 + u_1 && \text{by (16a)} \\ &= \frac{w_2(c_1+k)+kw_1}{2(c_2+k)(c_1+k)}. && \text{by (24) \& (25).} \end{aligned} \tag{26}$$

Substituting x_1 (25) back into the equation for u_1 (24) gives u_1 purely in terms of the weights and constants:

$$u_1 \equiv \frac{w_2(c_1+k) - c_2w_1}{2(c_2+k)(c_1+k)} \tag{27}$$

D. Evaluation of value-gradients along the greedy trajectory

We can now evaluate the $\widetilde{G}$ values by substituting the greedy trajectory's state vectors (eqs. (25)-(26)) into (20), giving:

$$\begin{aligned} \widetilde{G}_1 &= 2c_1x_1 - w_1 && \text{by (20)} \\ &= \frac{c_1w_1}{(c_1+k)} - w_1 && \text{by (25)} \\ &= \frac{-w_1k}{(c_1+k)} \end{aligned} \tag{28}$$

and

$$\begin{aligned} \widetilde{G}_2 &= 2c_2x_2 - w_2 && \text{by (20)} \\ &= \frac{w_2(c_1+k)c_2 + kw_1c_2}{(c_2+k)(c_1+k)} - w_2 && \text{by (26)} \\ &= \frac{kw_1c_2 - w_2k(c_1+k)}{(c_2+k)(c_1+k)}. \end{aligned} \tag{29}$$

The greedy actions of equations (23) and (24) both satisfy

$$\left(\frac{\partial \pi}{\partial x}\right)_t = \frac{-c_{t+1}}{c_{t+1}+k} \qquad \text{for } t \in \{0,1\} \tag{30}$$

Substituting (30) and (16a) into the definition for $\left(\frac{Df}{Dx}\right)_t$ given by (6), gives,

$$\begin{aligned} \left(\frac{Df}{Dx}\right)_t &= \left(\frac{\partial f}{\partial x}\right)_t + \left(\frac{\partial \pi}{\partial x}\right)_t \left(\frac{\partial f}{\partial u}\right)_t && \text{by (6)} \\ &= \left(\frac{\partial x+u}{\partial x}\right)_t + \left(\frac{\partial \pi}{\partial x}\right)_t \left(\frac{\partial x+u}{\partial u}\right)_t && \text{by (16a)} \\ &= 1 - \frac{c_{t+1}}{c_{t+1}+k}, \quad \text{for } t \in \{0,1\} && \text{by (30)} \\ &= \frac{k}{c_{t+1}+k}, \quad \text{for } t \in \{0,1\}. \end{aligned} \tag{31}$$

Similarly, the expression for $\left(\frac{DU}{Dx}\right)_t$ is found by:

$$\begin{aligned} \left(\frac{DU}{Dx}\right)_t &= \left(\frac{\partial U}{\partial x}\right)_t + \left(\frac{\partial \pi}{\partial x}\right)_t \left(\frac{\partial U}{\partial u}\right)_t && \text{by (6)} \\ &= \left(\frac{\partial k(u)^2}{\partial x}\right)_t + \left(\frac{\partial \pi}{\partial x}\right)_t \left(\frac{\partial k(u)^2}{\partial u}\right)_t && \text{by (16b)} \\ &= 0 - \frac{c_{t+1}}{c_{t+1}+k}(2ku_t), \quad \text{for } t \in \{0,1\} && \text{by (30)} \\ &= \frac{-2kc_{t+1}u_t}{c_{t+1}+k}, \quad \text{for } t \in \{0,1\}. \end{aligned} \tag{32}$$

E. Backwards pass along trajectory

We do a backwards pass along the trajectory calculating the target gradients using (5) with $\gamma = 1$:

$$\begin{aligned} {G'}_2 &= \left(\frac{\partial U}{\partial x}\right)_2 && \text{by (5) with } x_2 \in \mathbb{T} \\ &= 2x_2 && \text{by (17)} \\ &= \frac{w_2(c_1+k)+kw_1}{(c_2+k)(c_1+k)} && \text{by (26)} \end{aligned} \tag{33}$$

Similarly,

$$\begin{aligned} {G'}_1 &= \left(\frac{DU}{Dx}\right)_1 + \left(\frac{Df}{Dx}\right)_1 \left(\lambda {G'}_2 + (1-\lambda)\widetilde{G}_2\right) && \text{by (5)} \\ &= \frac{-2kc_2u_1}{c_2+k} + \frac{k}{c_2+k}\left(\lambda {G'}_2 + (1-\lambda)\widetilde{G}_2\right) && \text{by (32),(31)} \\ &= \frac{-kc_2(w_2(c_1+k) - c_2w_1)}{(c_1+k)(c_2+k)^2} && \text{by (27)} \\ &\quad + \frac{k}{c_2+k}\left(\lambda \frac{w_2(c_1+k)+kw_1}{(c_2+k)(c_1+k)}\right. && \text{by (33)} \\ &\quad \left. +(1-\lambda)\frac{kw_1c_2 - w_2k(c_1+k)}{(c_2+k)(c_1+k)}\right) && \text{by (29)} \\ &= \frac{w_1k(k\lambda + (c_2)^2 + k(1-\lambda)c_2)}{(c_1+k)(c_2+k)^2} \\ &\quad - \frac{w_2k(c_2 - \lambda + k(1-\lambda))}{(c_2+k)^2} \end{aligned} \tag{34}$$

F. Analysis of weight update equation

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

The VGL(λ) weight update (4) is comprised of

$$\begin{aligned} &\sum_t \left(\frac{\partial \widetilde{G}}{\partial w_i}\right)_t \Omega_t({G'}_t - \widetilde{G}_t) \\ &\quad = -\Omega_i({G'}_i - \widetilde{G}_i) \qquad \text{(for } i \in \{1,2\}\text{, by (21)).} \end{aligned}$$

Switching to vector notation for $\vec{w}$, this is

$$\begin{aligned} \sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t({G'}_t - \widetilde{G}_t) &= -\begin{pmatrix} \Omega_1({G'}_1 - \widetilde{G}_1) \\ \Omega_2({G'}_2 - \widetilde{G}_2) \end{pmatrix} \\ &= -\begin{pmatrix} \Omega_1 & 0 \\ 0 & \Omega_2 \end{pmatrix} \begin{pmatrix} {G'}_1 - \widetilde{G}_1 \\ {G'}_2 - \widetilde{G}_2 \end{pmatrix} \end{aligned}$$

$$= DB\vec{w} \tag{35}$$

where

$$D = \begin{pmatrix} \Omega_1 & 0 \\ 0 & \Omega_2 \end{pmatrix} \tag{36}$$

and B is a 2×2 matrix with elements found by subtracting equations (28) and (29) from equations (34) and (33), respectively, giving,

$$B = -\begin{pmatrix} \frac{k(k\lambda+(c_2)^2+k(1-\lambda)c_2)}{(c_1+k)(c_2+k)^2} + \frac{k}{(c_1+k)} & \frac{-k(c_2+k-\lambda(k+1))}{(c_2+k)^2} \\ \frac{k(1-c_2)}{(c_2+k)(c_1+k)} & \frac{1+k}{(c_2+k)} \end{pmatrix} \tag{37}$$

By equations (4) and (35), $\Delta\vec{w} = \alpha DB\vec{w}$ is the VGL(λ) weight update written as a single dynamic system of $\vec{w}$.

G. Equivalence of BPTT to VGLΩ(1) for this Example Problem

We define VGL$\Omega(\lambda)$ to be the VGL(λ) algorithm combined with the Ω_t matrix of (15). We now demonstrate that VGL$\Omega(1)$ is identical to the BPTT equation derived for this example problem. The purpose of this subsection is just to provide a confirmation of the main equivalence proof of this paper (Section III), in the context of our simple example problem.

To construct the Ω_t matrix of (15), we differentiate (22) to get

$$\left(\frac{\partial^2 \widetilde{Q}}{\partial u \partial u}\right)_t = 2(c_{t+1} + k) \qquad \text{for } t \in \{0, 1\}. \tag{38}$$

and then substitute (38) and $\left(\frac{\partial f}{\partial u}\right)_t = 1$ (by (16a)) into (15), to get

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

Substituting these Ω_t matrices into the D matrix of (36) gives

$$D = \begin{pmatrix} \frac{1}{2(c_1+k)} & 0 \\ 0 & \frac{1}{2(c_2+k)} \end{pmatrix} \tag{39}$$

The VGL$\Omega(1)$ weight update, with $\alpha = 1$, is

$$\begin{aligned} \Delta\vec{w} &= DB\vec{w} && \text{by (4), (35)} \\ &= -D\begin{pmatrix} \frac{k(k+(c_2)^2)}{(c_1+k)(c_2+k)^2} + \frac{k}{(c_1+k)} & \frac{k(1-c_2)}{(c_2+k)^2} \\ \frac{k(1-c_2)}{(c_2+k)(c_1+k)} & \frac{1+k}{(c_2+k)} \end{pmatrix}\vec{w} && \text{by (37), } \lambda = 1 \\ &= -2D\begin{pmatrix} \frac{k(k+(c_2)^2)}{(c_2+k)^2} + k & \frac{k(1-c_2)}{(c_2+k)} \\ \frac{k(1-c_2)}{(c_2+k)} & 1+k \end{pmatrix} D\vec{w} && \text{by (39)} \end{aligned} \tag{40}$$

We aim to show that this equation is identical to gradient descent on J, i.e. $\Delta\vec{w} = -\frac{\partial J}{\partial \vec{w}}$. First we consider the equations that determine how the actions u_0 and u_1 depend on $\vec{w}$. By equations (25) and (27), we have

$$\begin{aligned} \begin{pmatrix} u_0 \\ u_1 \end{pmatrix} &= \begin{pmatrix} \frac{1}{2(c_1+k)} & 0 \\ \frac{-c_2}{2(c_2+k)(c_1+k)} & \frac{1}{2(c_2+k)} \end{pmatrix}\vec{w} \\ &= ED\vec{w} \end{aligned} \tag{41}$$

where D is given by (39) and

$$E = \begin{pmatrix} 1 & 0 \\ \frac{-c_2}{c_2+k} & 1 \end{pmatrix}. \tag{42}$$

Now we can consider the gradient descent equation as follows:

$$\begin{aligned} -\frac{\partial J}{\partial \vec{w}} &= -\frac{\partial\big(k(u_0)^2 + k(u_1)^2 + (x_0 + u_0 + u_1)^2\big)}{\partial \vec{w}} && \text{by (18)} \\ &= -2ku_0\frac{\partial u_0}{\partial \vec{w}} - 2ku_1\frac{\partial u_1}{\partial \vec{w}} - 2(u_0+u_1)\left(\frac{\partial u_0}{\partial \vec{w}} + \frac{\partial u_1}{\partial \vec{w}}\right) \\ &= -2((k+1)u_0 + u_1)\frac{\partial u_0}{\partial \vec{w}} - 2(u_0 + (k+1)u_1)\frac{\partial u_1}{\partial \vec{w}} \\ &= -2\frac{\partial u_0}{\partial \vec{w}}\begin{pmatrix} k+1 & 1 \end{pmatrix}\begin{pmatrix} u_0 \\ u_1 \end{pmatrix} - 2\frac{\partial u_1}{\partial \vec{w}}\begin{pmatrix} 1 & k+1 \end{pmatrix}\begin{pmatrix} u_0 \\ u_1 \end{pmatrix} \\ &= -2\begin{pmatrix} \frac{\partial u_0}{\partial w_1} & \frac{\partial u_1}{\partial w_1} \\ \frac{\partial u_0}{\partial w_2} & \frac{\partial u_1}{\partial w_2} \end{pmatrix}\begin{pmatrix} k+1 & 1 \\ 1 & k+1 \end{pmatrix}\begin{pmatrix} u_0 \\ u_1 \end{pmatrix} \\ &= -2DE^T\begin{pmatrix} k+1 & 1 \\ 1 & k+1 \end{pmatrix}ED\vec{w} && \text{by (41)} \\ &= -2DE^T\begin{pmatrix} k+1 & 1 \\ 1 & k+1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ \frac{-c_2}{c_2+k} & 1 \end{pmatrix}D\vec{w} && \text{by (42)} \\ &= -2D\begin{pmatrix} 1 & \frac{-c_2}{c_2+k} \\ 0 & 1 \end{pmatrix}\begin{pmatrix} k + \frac{k}{c_2+k} & 1 \\ \frac{k(1-c_2)}{c_2+k} & k+1 \end{pmatrix}D\vec{w} && \text{by (42)} \\ &= -2D\begin{pmatrix} \frac{k(k+(c_2)^2)}{(c_2+k)^2} + k & \frac{k(1-c_2)}{(c_2+k)} \\ \frac{k(1-c_2)}{(c_2+k)} & k+1 \end{pmatrix}D\vec{w} \end{aligned}$$

This final line is identical to (40), which completes the proof of exact equivalence of VGL$\Omega(1)$ to BPTT for this particular problem.

H. Divergence Examples for VGL(0) and VGL(1)

We now show that unlike VGL$\Omega(1)$, the algorithms DHP and VGL(1) can both be made to diverge with a greedy policy in this problem domain.

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

$$\begin{aligned} \Delta\vec{p} &= \alpha\sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{p}}\right)_t \Omega_t({G'}_t - \widetilde{G}_t) && \text{by (4)} \\ &= \alpha\sum_t \frac{\partial \vec{w}}{\partial \vec{p}}\left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t({G'}_t - \widetilde{G}_t) && \text{by chain rule} \\ &= \alpha\frac{\partial \vec{w}}{\partial \vec{p}}\sum_t \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \Omega_t({G'}_t - \widetilde{G}_t) && \text{since independent of } t \\ &= \alpha\frac{\partial \vec{w}}{\partial \vec{p}}DB\vec{w} && \text{by (35)} \\ &= \alpha(F^T DBF)\vec{p}. && \text{by } \vec{w} = F\vec{p} \text{ and } \frac{\partial \vec{w}}{\partial \vec{p}} = \frac{\partial(F\vec{p})}{\partial \vec{p}} = F^T \end{aligned} \tag{43}$$

Equation (43) represents the whole learning system, described as a dynamical system of the weight vector $\vec{p}$.

We consider the VGL(0) and VGL(1) algorithms with the Ω_t matrix equal to the identity matrix, which implies that $D = I$, the 2×2 identity matrix, and hence we can ignore D from (43).

The optimal actions $u_0 = u_1 = 0$ would be achieved by $\vec{p} = \vec{0}$. To produce a divergence example, we want to ensure that $\vec{p}$ does not converge to $\vec{0}$.

Taking $\alpha > 0$ to be sufficiently small, then the weight vector $\vec{p}$ evolves according to a continuous-time linear dynamic system (equation (43), with D ignored), and this system is stable if and only if the matrix product $F^T BF$ is "stable" (i.e. if the real part of every eigenvalue of this matrix product is negative). The logic here is that if it is proven to diverge for a continuous time system, i.e. in the limit of an infinitesimal learning rate, then it would also diverge for any small finite learning rate too.

Choosing $\lambda = 0$, with $c_1 = c_2 = k = 0.01$ gives $B = \left(\begin{smallmatrix} -0.75 & 0.5 \\ -24.75 & -50.5 \end{smallmatrix}\right)$ (by (37)). Choosing $F = \left(\begin{smallmatrix} 10 & 1 \\ -1 & -1 \end{smallmatrix}\right)$ makes $F^T BF = \left(\begin{smallmatrix} 117.0 & -38.25 \\ 189.0 & -27.0 \end{smallmatrix}\right)$ which has eigenvalues $45 \pm 45.22i$. Since the real parts of these eigenvalues are positive, (43) will diverge for VGL(0) (i.e. DHP).

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

Fig. 1 shows the divergences obtained for VGL(0) and VGL(1) with a greedy policy.

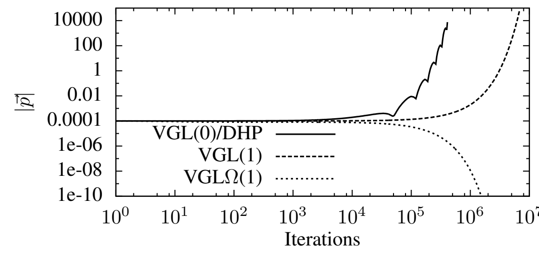

Fig. 1. Diverging behaviour for VGL(0) (i.e. DHP) and VGL(1), using the learning parameters described in Section IV-H and a learning rate of $\alpha = 10^{-6}$); and converging behaviour on the same problem for VGLΩ(1), as described in Section IV-I, with $\alpha = 10^{-3}$.

I. Results for VGLΩ(1) and VGLΩ(0)

VGL$\Omega(\lambda)$ is defined to be VGL(λ) with the Ω_t matrix defined by (15). As predicted by the convergence proof of Section III, and the explicit demonstration of Section IV-G, it was not possible to make the VGLΩ(1) weight update diverge. An example of VGLΩ(1) converging under the same conditions that caused VGL(1) to diverge is given in Fig.1.

Next, we considered VGLΩ(0). Substituting the same parameters that made VGL(0) diverge, i.e. $c_1 = c_2 = k = 0.01$, into (39) gives $D = \left(\begin{smallmatrix} 25 & 0 \\ 0 & 25 \end{smallmatrix}\right)$. Since D is a positive multiple of the identity matrix, its presence in (43) will not affect the stability of the product $F^T DBF$, so the system for $\vec{p}$ will still be unstable, and diverge, just as it did for VGL(0) (where D was taken to be the identity matrix). So unfortunately using the Ω_t matrix of (15) does not force reliable convergence for VGL(0) (i.e. DHP) with a greedy policy.

V. Neural-Network Experiments

To extend the experiments of the previous section that used a quadratic function approximator for the critic, in this section we consider two neural-network based critic experiments: a vertical-spacecraft problem and the cart-pole benchmark problem.

A. Vertical-Spacecraft Problem

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 Δt 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). Δt was chosen to be 0.4.

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

$$U(\vec{x}) = \frac{1}{2} m v^2 + m(k_g)h \tag{44}$$

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 (44), a cost is also given for each non-terminal step. This cost is

$$U(\vec{x}, u) = c\left(\ln(2 - 2u) - u \ln\left(\frac{1 - u}{u}\right)\right)\Delta t \tag{45}$$

where $c = 0.01$ 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 V-B, but first we describe experiments that did not use a greedy policy.

A DHP-style critic, $\widetilde{G}(\vec{x}, \vec{w})$, 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 shortcut 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. To ensure suitably scaled inputs for the

MLP, we used a rescaled state vector $\vec{x}'$ defined to be $\vec{x}' = (h/1600, v/40, t/200)^T$. In our implementation, we also used redefined model and cost functions that work directly with the rescaled state vectors, i.e. we rescaled them so that $\vec{x}'_{t+1} = f(\vec{x}'_t, \vec{u}_t)$ and $U_t = U(\vec{x}'_t, \vec{u}_t)$. By doing this we also ensured that the output of the neural network, $\widetilde{G}$, was also suitably scaled.

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 $m = 0.02$. In all of the experiments we made the trajectory always start from $h = 1600$, $v = -2$, and used discount factor $\gamma = 1$. The exact derivatives of the functions $f(\vec{x}, \vec{u})$ and $U(\vec{x}, \vec{u})$ were made available to the algorithms.

Results using the actor-critic architecture and Algorithm 1 are given in the left-hand graph of Fig. 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 cost comes as a final impulse, so it is advantageous to have a long look-ahead (i.e. a high λ 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.

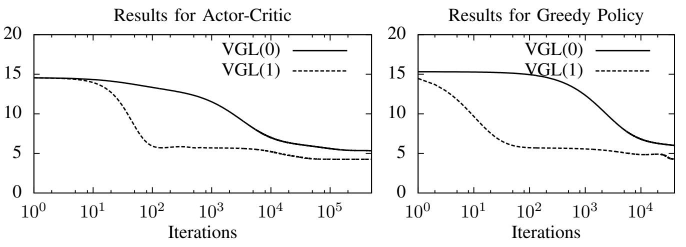

Fig. 2. VGL(0) (i.e. DHP) and VGL(1): The left-hand graph shows Actor-Critic performance, using learning rates $\alpha = 10^{-6}$ and $\beta = 0.01$ as described in Section V-A; and the right-hand graph shows performance with a greedy policy and $\alpha = 10^{-6}$ as described in Section V-B.

B. Vertical-Spacecraft Problem with Greedy Policy

The cost function of (45) was derived to form an efficient greedy policy, by following the method of [24]. This method uses a continuous time approximation that allows the greedy policy to be derived in the form of an arbitrary sigmoidal function $g(\cdot)$. To achieve a range of $u \in (0, 1)$, we chose g to be the logistic function,

$$g(x) = \frac{1}{1 + e^{-x/c}}. \tag{46}$$

The choice of c affects the sharpness of this sigmoid function. Using this chosen sigmoid function, the cost function based on [24] is defined to be

$$U(\vec{x}, u) = \Delta t \int g^{-1}(u) du. \tag{47}$$

Note that solving this integral gives (45). Then to derive the greedy policy for this cost function, we make a first order Taylor series expansion of the $\widetilde{Q}(\vec{x}, \vec{u}, \vec{w})$ function (8) about the point $\vec{x}$:

$$\begin{aligned}\widetilde{Q}(\vec{x}, \vec{u}, \vec{w}) &\approx U(\vec{x}, \vec{u}) + \gamma \left(\left(\frac{\partial \widetilde{J}}{\partial \vec{x}} \right)^T (f(\vec{x}, \vec{u}) - \vec{x}) + \widetilde{J}(\vec{x}, \vec{w}) \right) \\ &= U(\vec{x}, \vec{u}) + \gamma \left(\widetilde{G}(\vec{x}, \vec{w}) \right)^T (f(\vec{x}, \vec{u}) - \vec{x}) + \gamma \widetilde{J}(\vec{x}, \vec{w})\end{aligned} \tag{48}$$

This approximation becomes exact in continuous time, i.e. in the limit as $\Delta t \to 0$. The greedy policy must minimise $\widetilde{Q}$, hence we differentiate (48) to get

$$\begin{aligned}\left(\frac{\partial \widetilde{Q}}{\partial u} \right)_t &= \left(\frac{\partial U}{\partial u} \right)_t + \gamma \left(\frac{\partial f}{\partial u} \right)_t \widetilde{G}_t & \text{by (48)} \\ &= g^{-1}(u_t) \Delta t + \gamma \left(\frac{\partial f}{\partial u} \right)_t \widetilde{G}_t & \text{by (47)}\end{aligned} \tag{49}$$

For a minimum, we must have $\frac{\partial \widetilde{Q}}{\partial u} = 0$, which, since $\frac{\partial f}{\partial u}$ is independent of u, gives the greedy policy as

$$\pi(\vec{x}_t, \vec{w}) = g \left(-\frac{\gamma}{\Delta t} \left(\frac{\partial f}{\partial u} \right)_t \widetilde{G}_t \right). \tag{50}$$

We note that this type of greedy policy is very similar to the Single Network Adaptive Critic (SNAC) formulation proposed by [18].

This is the sigmoidal form for the greedy policy that we sought to derive. We used this greedy policy function (50) in place of $A(\vec{x}, \vec{z})$ in lines 4 and 12 of Algorithm 1. For the occurrence of $\frac{\partial A}{\partial \vec{x}}$ in line 12, we differentiated (50) directly to obtain

$$\left(\frac{\partial \pi}{\partial \vec{x}} \right)_t = g' \left(-\frac{\gamma}{\Delta t} \left(\frac{\partial f}{\partial u} \right)_t \widetilde{G}_t \right) \left(-\frac{\gamma}{\Delta t} \left(\frac{\partial \widetilde{G}}{\partial \vec{x}} \right)_t \left(\frac{\partial f}{\partial u} \right)_t^T \right) \tag{51}$$

where $g'(x)$ is the derivative of the function $g(x)$ and where we have used the fact that for these model functions $\vec{u}$ is one-dimensional and $\frac{\partial^2 f}{\partial \vec{x} \partial u} = 0$. Lines 14 and 18 of the algorithm were not used.

The results for experiments using the greedy policy are shown in the right-hand graph of Fig. 2. Comparing the left-hand and right-hand graphs we see the relative performance between VGL(1) and VGL(0) is similar. This indicates that in this experiment, the greedy policy derived can successfully replace the action network, raising efficiency, and without any apparent detriment.

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 are shown in the two left-hand graphs of Fig. 3. It seems the aggressive acceleration by RPROP can cause large instability in the VGL(1) and DHP (VGL(0)) algorithms. This is because neither of these two algorithms is true gradient descent when used with a greedy policy (e.g. as shown in Section IV).

However when the Ω_t matrix defined by (15) is used with $\lambda = 1$, giving the algorithm VGLΩ(1), we did obtain

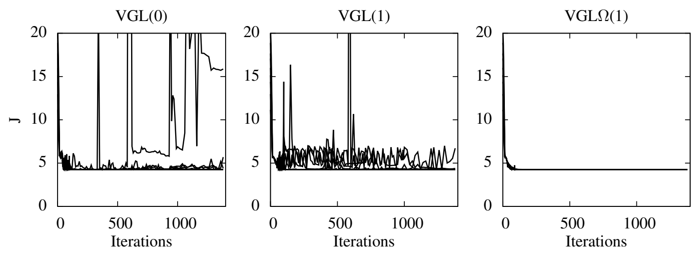

Fig. 3. Solution of the spacecraft problem by VGL(0), VGL(1) and VGLΩ(1), with a greedy policy, using RPROP. Each graph shows the performance of a learning algorithm for each of ten different weight initialisations; hence the ensemble of curves in each graph gives some idea of an algorithm's reliability and volatility.

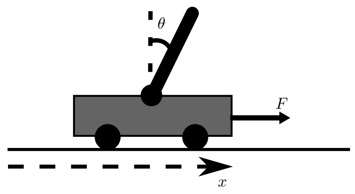

Fig. 4. Cart-pole benchmark problem. A pole with a pivot at its base is balancing on a cart. The objective is to apply a changing horizontal force F to the cart which will move the cart backwards and forwards so as to balance the pole vertically. State variables are pole angle, θ, and cart position, x, plus their derivatives with respect to time, $\dot{\theta}$ and $\dot{x}$.

monotonic progress, as shown in the right-hand graph of Fig. 3, and as explained by the equivalence proof of Section III.

In this case, due to the continuous time approximations made earlier, we modified Ω_t from (15) into

$$\Omega_t = \left(\frac{\partial f}{\partial u}\right)_t^T \left(\frac{\partial^2 \widetilde{Q}}{\partial u \partial u}\right)_t^{-1} \left(\frac{\partial f}{\partial u}\right)_t,$$

with $\frac{\partial^2 \widetilde{Q}}{\partial u \partial u}$ found by differentiating (49), giving

$$\begin{aligned}\left(\frac{\partial^2 \widetilde{Q}}{\partial u \partial u}\right)_t &= \frac{\partial\left(g^{-1}(u_t)\Delta t + \gamma \left(\frac{\partial f}{\partial u}\right)_t \widetilde{G}_t\right)}{\partial u_t} && \text{by (49)} \\ &= \frac{\partial g^{-1}(u_t)}{\partial u_t}\Delta t && \text{(since } \frac{\partial^2 f}{\partial u \partial u} = 0 \text{ and } \frac{\partial \widetilde{G}_t}{\partial u_t} = 0\text{)} \\ &= \frac{1}{g'\left(g^{-1}(u_t)\right)}\Delta t && \text{(differentiating an inverse)} \\ &= \frac{1}{g'\left(-\frac{\gamma}{\Delta t}\left(\frac{\partial f}{\partial u}\right)_t \widetilde{G}_t\right)}\Delta t && \text{by (50)}\end{aligned}$$

$$\Rightarrow \Omega_t = \frac{1}{\Delta t}\left(\frac{\partial f}{\partial u}\right)_t^T g'\left(-\frac{\gamma}{\Delta t}\left(\frac{\partial f}{\partial u}\right)_t \widetilde{G}_t\right)\left(\frac{\partial f}{\partial u}\right)_t \quad (52)$$

This equation for Ω_t is advantageous to use over (15) in that it is always defined to exist (i.e. there is no matrix to invert), it is always positive semi-definite, and it is very efficient to implement. Hence we use (52) in preference to (15) in our experiments here. However (52) is only an approximation to (15), hence the applicability of the equivalence to BPTT proof of III-C will only be approximate; but the approximation becomes exact in the limit $\Delta t \to 0$, and in practice the empirical results are good with the finite Δt value that we used, as shown in the right-hand graph of Fig. 3. This graph shows the minimum being reached stably and many times quicker than the other algorithms considered.

Unfortunately, since the value of Ω_t given by (52) is not full rank, it does not make a good candidate for use in DHP, or for any VGL(λ) algorithm with $\lambda < 1$. This is an area for future research.

C. Cart-Pole Experiment

We applied the algorithm to the well known cart-pole benchmark problem described in Fig. 4. The equation of motion for this system ([11], [28], [29]), in the absence of friction, is:

$$\ddot{\theta} = \frac{g \sin\theta - \cos\theta \left[\frac{F + ml\dot{\theta}^2 \sin\theta}{m_c + m}\right]}{l\left[\frac{4}{3} - \frac{m \cos^2\theta}{m_c + m}\right]} \quad (53)$$

$$\ddot{x} = \frac{F + ml\left[\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta\right]}{m_c + m} \quad (54)$$

where gravitational acceleration, $g = 9.8ms^{-2}$; cart's mass, $m_c = 1kg$; pole's mass, $m = 0.1kg$; half pole length, $l = 0.5m$; $F \in [-10, 10]$ is the force applied to the cart, in Newtons; and θ is the pole angle, in radians. The motion was integrated using the Euler method with a time constant $\Delta t = 0.02$, which, for a state vector $\vec{x} \equiv (x, \theta, \dot{x}, \dot{\theta})^T$, gives a model function $f(\vec{x}, \vec{u}) = \vec{x} + (\dot{x}, \dot{\theta}, \ddot{x}, \ddot{\theta})^T \Delta t$.

To achieve the objective of balancing the pole and keeping the cart close to the origin, $x = 0$, we used a cost function

$$\begin{aligned}U(\vec{x}, t, u) =& \gamma^t \left(5x^2 + 50\theta^2\right. \\ &\left. + c\left(\ln(2 - 2u) - u \ln\left(\frac{1-u}{u}\right)\right)\right)\Delta t\end{aligned} \quad (55)$$

applied at each time step, and the term with coefficient c is there to enable an efficient greedy policy as in Section V-B, but here with $c = 10$. Each trajectory was defined to last exactly 300 timesteps, i.e. 6 seconds of real time, regardless of whether the pole swung below the horizontal or not, and with no constraint on the magnitude of x. This cost function and the fixed duration trajectories is similar to that used by [11], [24], but differs to the trajectory termination criterion used by [28] which relies upon a non-differentiable step cost function, and hence is not appropriate for VGL based methods. We used $\gamma = 0.96$ as a discount factor in (55). This discount factor is placed in the definition of U so that the sharp truncation of trajectories terminating after 6 seconds is replaced by a smooth decay. This is preferable to the way that Algorithm 1 implements discount factors, which effectively treats each time step as creating a brand new cost-to-go function to minimise.

Following the same derivation as in Section V-B, a greedy policy was given by (50), which we used to map u to F by $F_t \equiv 20u_t - 10$ (to achieve the desired range $F \in [-10, 10]$ when using using $g(x)$ defined by (46)). Again, Ω_t and $\frac{\partial \pi}{\partial \vec{x}}$ were given by (52) and (51), respectively.

Training used a DHP style critic MLP network, with 4 inputs, a single hidden layer of 12 units and 4 output nodes, with extra shortcut connections from the input layer to the output layer. The activation functions used were hyperbolic tangent functions at all nodes except for the output layer which used the identity function. Network weights were initially randomised uniformly from the range $[-0.1, 0.1]$. To ensure the state vector was suitably scaled for input to the MLP, we used rescaled state vectors $\vec{x}'$ defined by $\vec{x}' = (0.16x, 4\theta/\pi, \dot{x}, 4\dot{\theta})^T$ throughout the implementation. As noted by [11], choosing an appropriate state-space scaling was critical to success with DHP on this problem.

Learning took place on 10 trajectories with fixed start points randomly chosen with $|x| < 2.4$, $|\theta| < \pi/15$, $|\dot{x}| < 5$, $|\dot{\theta}| < 5$, which are similar to the starting conditions used by [28]. The exact derivatives of the model and cost functions were made available to the algorithms. Four algorithms were tested and their results are shown in Fig. 5. Both VGL(1) and VGL(0) performed badly when accelerated by RPROP. The results again show that VGLΩ(1) had less volatility and better performance than both VGL(1) and VGL(0), which demonstrates the effectiveness of the Ω_t matrix used. For comparison, we also show the results of an actor-only architecture (i.e. with no critic) trained entirely by BPTT and RPROP. This demonstrates that the minimum attained by the VGL algorithms is suitably low. Also we observed that when this minimum was reached, the pole was being balanced effectively with the cart remaining close to $x = 0$.

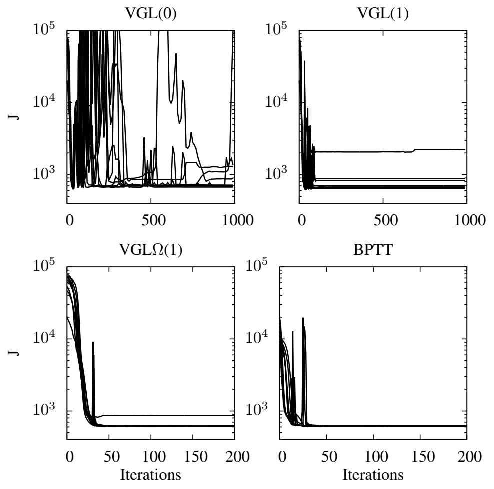

Fig. 5. Cart-pole solutions by VGL(0), VGL(1) and VGLΩ(1), with a greedy policy, plus, for comparison, BPTT. All algorithms were used in conjunction with RPROP. Each graph shows the performance of a learning algorithm for each of ten different random weight initialisations; hence each ensemble of curves gives some idea of an algorithm's reliability and volatility.

The results show the cart-pole problem being solved effectively. We have achieved largely monotonic progress (with the brief non-monotonicity down to the aggressive acceleration of RPROP and/or discontinuities in the cost-to-go function surface in weight space) for a critic learning algorithm, replicating the performance of BPTT by a critic with a greedy policy.

VI. Conclusions

We have found a strong theoretical equivalence between BPTT and an ADP critic weight update (VGLΩ(1)), two algorithms that on first sight appeared to be operating totally differently. This provides a convergence proof for this VGL algorithm under the conditions stated in Section III-D. This analysis has been successful for a VGL learning system where a greedy policy is used. Analytical and empirical confirmations of the equivalence to BPTT have been provided in sections IV-G and V, respectively. This contrasts to the demonstrated divergence in Section IV of several other critic algorithms with a greedy policy (DHP, VGL(1) and VGLΩ(0)).

In the experiments of Section V we have shown the effectiveness of the algorithm and its ability to produce approximate monotonic learning progress for a neural-network based critic with a greedy policy, even when combined with an aggressive learning accelerator such as RPROP.

References

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

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

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

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

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

[6] M. Fairbank, E. Alonso, and D. Prokhorov, "Simple and fast calculation of the second-order gradients for globalized dual heuristic dynamic programming in neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 10, pp. 1671–1678, October 2012.

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

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

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

[10] 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.

[11] 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.

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

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

[15] P. J. Werbos, T. McAvoy, and T. Su, "Neural networks, system identification, and control in the chemical process industries." in *Handbook of Intelligent Control*, White and Sofge, Eds. New York: Van Nostrand Reinhold, 1992, ch. 10, pp. 283–356.

[16] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press, 1960, ch. 4, pp. 42–43.

[17] A. Al-Tamimi, F. L. Lewis, and M. Abu-Khalaf, "Discrete-time nonlinear HJB solution using approximate dynamic programming: Convergence proof." *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 38, no. 4, pp. 943–949, 2008.

[18] A. Heydari and S. N. Balakrishnan, "Finite-horizon input-constrained nonlinear optimal control using single network adaptive critics," *American Control Conference ACC*, pp. 3047–3052, 2011.

[19] D. V. Prokhorov and D. C. Wunsch, "Convergence of critic-based training," in *in Proc. IEEE Int. Conf. Syst*, 1997, pp. 3057–3060.

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

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

[22] P. J. Werbos, "Consistency of HDP applied to a simple reinforcement learning problem," *Neural Networks*, vol. 3, pp. 179–189, 1990.

[23] ——, "Backpropagation through time: What it does and how to do it," in *Proceedings of the IEEE*, vol. 78, No. 10, 1990, pp. 1550–1560.

[24] K. Doya, "Reinforcement learning in continuous time and space," *Neural Computation*, vol. 12, no. 1, pp. 219–245, 2000.

[25] E. Barnard, "Temporal-difference methods and markov models," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 2, pp. 357–365, 1993.

[26] M. Fairbank, D. Prokhorov, and E. Alonso, "Approximating optimal control with value gradient learning," in *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*, F. Lewis and D. Liu, Eds. New York: Wiley-IEEE Press, 2012, Sections 7.3.4 and 7.4.3.

[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.

[28] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 13, pp. 834–846, 1983.

[29] R. V. Florian, "Correct equations for the dynamics of the cart-pole system," Center for Cognitive and Neural Studies (Coneural), Str. Saturn 24, 400504 Cluj-Napoca, Romania, Tech. Rep., 2007.

Michael Fairbank received his BSc degree in Mathematical Physics from Nottingham University in 1994, and MSc in Knowledge Based Systems from Edinburgh University in 1995. He has been independently researching ADPRL and neural networks since that time, while pursing careers in computer programming and mathematics teaching. He is now a PhD student at City University London.

His research has been primarily motivated by considering a simple simulated-spacecraft neurocontroller, and optimising flight through a landscape of increasingly challenging obstacles. Limitations in standard RL led him to discover the benefits of ADP methods, including DHP, VGL(λ) and BPTT. The search for a convergence proof for critic designs led to the work presented here. He is also interested in neural-network learning algorithms, especially for recurrent neural networks.

Dr. Eduardo Alonso is a Reader in Computing at City University London. His research interests include Artificial Intelligence and reinforcement learning, both in machine learning and as a computational model of associative learning in neuroscience. He co-directs the Centre for Computational and Animal Learning Research. He is contributing to "The Cambridge Handbook of Artificial Intelligence" and has edited special issues for the journals "Autonomous Agents and Multi-Agent Systems" and "Learning & Behavior", and the book "Computational Neuroscience for Advancing Artificial Intelligence: Models, Methods and Applications". He has acted as OC and PC of the International Joint Conference on Artificial Intelligence (IJCAI) and the International Conference on Autonomous Agents and Mutiagent Systems (AAMAS), served as vice-chair of The Society for the Study of Artificial Intelligence and the Simulation of Behaviour (AISB), and is a member of the UK Engineering and Physical Sciences Research Council (EPSRC) Peer Review College.

Dr. Danil Prokhorov (SM02) began his career in St. Petersburg, Russia, in 1992. He was a research engineer in St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences. He became involved in automotive research in 1995 when he was a Summer intern at Ford Scientific Research Lab in Dearborn, MI. In 1997 he became a Ford Research staff member involved in application-driven research on neural networks and other machine learning methods. While at Ford, he was involved in several production-bound projects including neural network based engine misfire detection. Since 2005 he is with Toyota Technical Center, Ann Arbor, MI. He is currently in charge of Mobility Research Department with Toyota Research Institute North America, a TTC division. He has more than 100 papers in various journals and conference proceedings, as well as several inventions, to his credit. He is honored to serve in a number of capacities including the International Neural Network Society (INNS) President, the National Science Foundation (NSF) Expert, and the Associate Editor/Program Committee member of many international journals/conferences.