A Comparison of Learning Speed and Ability to Cope Without Exploration between DHP and TD(0)

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

Cite as: Michael Fairbank and Eduardo 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, June 2012, Brisbane (IEEE IJCNN 2012), pp. 1478–1485.

Errata: See footnote 1

***Abstract*—This paper demonstrates the principal motivations for Dual Heuristic Dynamic Programming (DHP) learning methods for use in Adaptive Dynamic Programming and Reinforcement Learning, in continuous state spaces: that of automatic local exploration, improved learning speed and the ability to work without stochastic exploration in deterministic environments. In a simple experiment, the learning speed of DHP is shown to be around 1700 times faster than TD(0). DHP solves the problem without any exploration, whereas TD(0) cannot solve it without explicit exploration.**

DHP requires knowledge of, and differentiability of, the environment's model functions. This paper aims to illustrate the advantages of DHP when these two requirements are satisfied.

***Index Terms*—Dual Heuristic Dynamic Programming, DHP, Adaptive Dynamic Programming, Reinforcement Learning**

I. Introduction

Adaptive Dynamic Programming (ADP) [1] and Reinforcement Learning (RL) [2] are similar fields of study that aim to make an agent learn actions that maximise a long-term reward function. These algorithms rely on learning a *value function*, V, that is defined in Bellman's Principle of Optimality [3].

Successful algorithms exist in the RL and ADP literature that learn the values of the value-function directly by sampling trajectories. These algorithms include the RL algorithms TD(λ), Sarsa(λ) and Q(λ) [4], [5], [6]. We refer to these algorithms collectively as "value learning" (VL) algorithms.

ADP contains similar and sometimes equivalent algorithms to RL, but in ADP there is another category of value-function learning algorithms that aim to learn the *value-gradient*, which we define to be $\frac{\partial V}{\partial \vec{x}}$, where $\vec{x}$ is the state vector of the agent in state-space. These algorithms include Dual Heuristic Dynamic Programming (DHP) and Value Gradient Learning (VGL(λ)) [7], [8], [9], [10], [11]. We refer to these algorithms collectively as "DHP based" algorithms.

Implementing DHP-based algorithms is a bit more difficult than the VL algorithms, and they require knowledge of the environment's model functions and their differentiability. However DHP methods have the big advantage that local exploration occurs automatically for them. This is the main motivation for DHP methods, and we elaborate on what this means in more detail in the rest of the paper. There are two main useful consequences of this fact:

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

- DHP methods do not require explicit local exploration of the state space. This means they can work reasonably well in deterministic environments without stochastic exploration, but VL methods cannot.
- DHP based methods can have much improved learning speeds compared to VL methods.

In this paper we aim to present clearly this motivation for DHP methods, plus the two consequences, by presenting a simple problem for comparison. We are providing this experiment to motivate DHP usage which is not currently a well known algorithm within the RL community, but potentially has these significant benefits which we aim to highlight here.

We also highlight important and relevant caveats for using the DHP algorithm: DHP assumes the model functions of the environment can be learned, and are differentiable, and that the cost of model learning is relatively low compared to the cost of learning optimal behaviour with a model-free algorithm such as TD(0). We show that when these conditions are satisfied, then the use of DHP could be extremely beneficial.

In section II we define the functions and concepts necessary for RL and ADP problems, and define the algorithms. In section III we give the motivations for designing and choosing the DHP based algorithms, making reference to how a greedy policy relates to the value-gradient. In section IV we give an experiment that confirms the motivations succinctly and convincingly, and in section V we give conclusions and summary.

II. Problem and Algorithm Definitions

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

the importance of long term rewards over short term ones, and $\mathbb{E}\,()$ denotes expectation.

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

A. Approximate Value Function

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

B. Greedy Policy

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

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

where we define the function $\widetilde{Q}$ as

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

C. Action Network

An *action network*, or *actor*, is a separate function approximator, $\pi(\vec{x},\vec{z})$, (with a weight vector $\vec{z}$) designed to represent the policy function. The action network can be trained by one of several possible methods, e.g. see [2, ch. 6.6] or [8, eq.10]. The objective of any actor training method is to make the actor behave as closely as possible to equation 1. If an actor is fully trained to behave exactly like equation 1 then we say it is *behaving greedily*.

D. Bellman's Optimality Principle

Bellman's Optimality Principle [3] asserts that there exists an optimal value function $V^*(\vec{x})$ which satisfies

$$V^*(\vec{x}) = \max_{\vec{a}\in\mathbb{A}}\left(\mathbb{E}\left(r(\vec{x},\vec{a}) + \gamma V^*(f(\vec{x},\vec{a}))\right)\right) \quad \forall\vec{x}\in\mathbb{S},$$

and if this optimal value function can be found, then optimal behaviour is determined by following a greedy policy on V^*.

We note that we can break down Bellman's Optimality Principle into two necessary conditions which are more relevant to the learning algorithms used in this paper: For any given approximate value function $\widetilde{V}(\vec{x},\vec{w})$ and policy function $\pi(\vec{x},\vec{z})$, if

1) $\widetilde{V}(\vec{x},\vec{w}) = \mathbb{E}\left(r(\vec{x},\pi(\vec{x},\vec{z})) + \gamma\widetilde{V}(f(\vec{x},\pi(\vec{x},\vec{z})),\vec{w})\right)$ for all $\vec{x}\in\mathbb{S}$, and simultaneously,
2) the policy $\pi(\vec{x},\vec{z})$ is behaving greedily on $\widetilde{V}$ for all $\vec{x}\in\mathbb{S}$,

then Bellman's Optimality Principle is satisfied, and $\pi(\vec{x},\vec{z})$ is an optimal policy.

E. TD(0) Learning

Here and throughout this paper, a convention is used that all defined vector quantities are columns, whether they are coordinates, or derivatives with respect to coordinates. For example, $\vec{x}_t$ and $\frac{\partial\widetilde{V}}{\partial\vec{x}}$ are both column vectors. Also, subscripted "t" indices are what we call *trajectory shorthand* notation. These refer to the time step of a trajectory and provide corresponding arguments $\vec{x}_t$ and $\vec{a}_t$ where appropriate; so that for example $\widetilde{V}_{t+1} \equiv \widetilde{V}(\vec{x}_{t+1},\vec{w})$, and $\left(\frac{\partial\widetilde{V}}{\partial\vec{w}}\right)_t$ is shorthand for the column vector function $\frac{\partial\widetilde{V}(\vec{x},\vec{w})}{\partial\vec{w}}$ evaluated at $(\vec{x}_t,\vec{w})$.

Using this notation, the TD(0) algorithm [4], applied in batch mode to a whole trajectory, can be defined succinctly by the following weight update:

$$\Delta\vec{w} = \alpha\sum_t\left(\frac{\partial\widetilde{V}}{\partial\vec{w}}\right)_t(r_t + \gamma\widetilde{V}_{t+1} - \widetilde{V}_t) \tag{3}$$

where $\alpha > 0$ is the learning rate. Pseudocode for the TD(0) algorithm is given in Algorithm 1.

The quantity

$$\delta_t = r_t + \gamma\widetilde{V}_{t+1} - \widetilde{V}_t \tag{4}$$

is called the "TD error". The TD algorithm aims to set this to zero all over state space. If this can be achieved exactly, whilst simultaneously making the policy behave greedily, then Bellman's Optimality Principle will be satisfied.

Algorithm 1 On-line implementation of TD(0) algorithm.

```
t ← 0
while not terminated(x_t) do
    a_t ← π(x_t)
    x_{t+1} ← f(x_t, a_t)
    δ ← r(x_t, a_t) + γṼ_{t+1} − Ṽ_t
    w ← w + α (∂Ṽ/∂w)_t δ
    t ← t + 1
end while
```

F. The DHP Algorithm

We define the *approximate value gradient*, or *critic gradient*, to be the column vector $\widetilde{G}(\vec{x},\vec{w}) \equiv \frac{\partial\widetilde{V}(\vec{x},\vec{w})}{\partial\vec{x}}$.

We define vector by vector differentiation by example. We define $\frac{\partial f}{\partial\vec{x}}$ to be a matrix with element (i,j) equal to $\frac{\partial f(\vec{x},\vec{a})^j}{\partial\vec{x}^i}$. Similarly, $\left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)^{ij} = \frac{\partial\widetilde{G}^j}{\partial\vec{w}^i}$; and combining with trajectory shorthand notation, $\left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_t$ is this matrix evaluated at $(\vec{x}_t,\vec{w})$.

Using this notation, and the implied matrix-vector products, the DHP algorithm is defined by a weight update of the form:

$$\Delta\vec{w} = \alpha\sum_t\left(\frac{\partial\widetilde{G}}{\partial\vec{w}}\right)_t\left(\left(\frac{Dr}{D\vec{x}}\right)_t + \gamma\left(\frac{Df}{D\vec{x}}\right)_t\widetilde{G}_{t+1} - \widetilde{G}_t\right) \tag{5}$$

where $\alpha > 0$ is a small learning rate constant; $\widetilde{G}_t$ is the approximate value gradient; and where $\frac{D}{D\vec{x}}$ is shorthand for

$$\frac{D}{D\vec{x}} \equiv \frac{\partial}{\partial\vec{x}} + \frac{\partial\pi}{\partial\vec{x}}\frac{\partial}{\partial\vec{a}}\ ; \tag{6}$$

and where all of these derivatives are assumed to exist. Equations 5 and 6 define the DHP algorithm.

We will call the quantity

$$\vec{e}_t = \left(\frac{Dr}{D\vec{x}}\right)_t + \gamma \left(\frac{Df}{D\vec{x}}\right)_t \widetilde{G}_{t+1} - \widetilde{G}_t \tag{7}$$

the "DHP error". The DHP algorithm aims to make this equal to zero whilst also under a policy that is behaving greedily. Achieving these two conditions simultaneously all over state space would recover Bellman's Optimality Principle.

DHP is traditionally defined with the function $\widetilde{G}(\vec{x}, \vec{w})$ directly implemented as the output of a smooth *vector* function approximator, where the output vector dimension is the same as that of $\vec{x}$ [7], [8]. However it can alternatively be implemented as the actual gradient $\frac{\partial \widetilde{V}(\vec{x},\vec{w})}{\partial \vec{x}}$, where $\widetilde{V}$ is the usual *scalar* approximate value function. Either option is possible. We use the second option in this paper's experiment, so that a greedy policy can be described more easily, and so that the relationship of DHP to TD(0) is more apparent.

Pseudocode for the DHP algorithm is given in Algorithm 2.[1] The most computationally expensive line of this implementation is the matrix-vector product in line 6. If this product is evaluated using methods analogous to those used by [12] then it can be evaluated in O($\dim(\vec{w})$) operations, which is thus the overall asymptotic running time for the whole algorithm, per trajectory time step. This is the same asymptotic running time as the TD(0) algorithm, so when comparing the two algorithms in experiments, we can just consider the number of iterations required until convergence.

Algorithm 2 On-line implementation of DHP algorithm.

1: $t \leftarrow 0$
2: **while** not terminated($\vec{x}_t$) **do**
3: $\quad \vec{a}_t \leftarrow \pi(\vec{x}_t)$
4: $\quad \vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{a}_t)$
5: $\quad \vec{e} \leftarrow \left(\frac{\partial r}{\partial \vec{x}}\right)_t + \left(\frac{\partial \pi}{\partial \vec{x}}\right)_t \left(\frac{\partial r}{\partial \vec{a}}\right)_t + \gamma \left(\left(\frac{\partial f}{\partial \vec{x}}\right)_t + \left(\frac{\partial \pi}{\partial \vec{x}}\right)_t \left(\frac{\partial f}{\partial \vec{a}}\right)_t \right) \widetilde{G}_{t+1} - \widetilde{G}_t$
6: $\quad \vec{w} \leftarrow \vec{w} + \alpha \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_t \vec{e}$
7: $\quad t \leftarrow t + 1$
8: **end while**

1) Applicability of the DHP algorithm: Due to the appearance of the derivatives of f and r in the DHP weight update, the DHP algorithm is model-based. We assume these model functions can be learned by a separate "system identification" learning process, for example as described by [13]. This system identification process could have taken place prior to the main learning process (which is a recognised strategy for RL, as was used in the successful flight of an inverted helicopter by [14]), or concurrently with it; and results in learned model functions $f(\vec{x}, \vec{a})$ and $r(\vec{x}, \vec{a})$. Alternatively, in deterministic continuous time situations, there is an on-line method by [15] which is capable of learning the necessary derivatives of f and r virtually instantaneously. We do not describe this model-learning stage of the process any further in this paper.

Furthermore, these functions must be differentiable. In many situations we can *force* smoothness onto the model functions by using a smooth deterministic function approximator to represent them. However it be noted that the DHP algorithm will be optimising performance with respect to the *learned* model functions, as opposed to the true model functions.

DHP also requires the differentiability of the policy function $\pi(\vec{x}, \vec{z})$. This can be achieved by implementing the actor by a smooth function approximator such as a neural network. However when a greedy policy is used, which often results in bang-bang behaviour, it can be harder to assure differentiability; except in simple analytical situations like the experiment of this paper, and in the continuous time scenario [16] where a closed form for the greedy policy based upon the value-gradient can be used.

The DHP algorithm is designed specifically for continuous space environments problems, so would not do as well at many traditional RL discrete state-space domains (such as board games or grid worlds).

III. Motivation for DHP based Algorithms by Consideration of the Greedy Policy

A strong motivation for using DHP based algorithms in continuous state spaces can be understood by considering a first-order Taylor series expansion of the greedy policy function (eq. 1):

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

Hence we see the greedy policy is dependent on $\frac{\partial \widetilde{V}}{\partial \vec{x}}$ and *not* on $\widetilde{V}$. We can see from this that changing $\frac{\partial \widetilde{V}}{\partial \vec{x}}$ will immediately affect the greedy policy. We can also see that if the greedy policy is to deliver an optimal trajectory, then learning this value-gradient is necessary, and that is why the DHP algorithm tries to learn it directly (note that $\frac{\partial \widetilde{V}}{\partial \vec{x}}$ is named $\widetilde{G}$ in Algorithm 2).

By moving $\frac{\partial \widetilde{V}}{\partial \vec{x}}$ towards its correct target we will steer the trajectory in the correct direction. That's all we have to do: Learn the $\frac{\partial \widetilde{V}}{\partial \vec{x}}$ gradients and the trajectory will bend itself into the correct (optimal) shape. Hence *local exploration is automatic* for DHP based methods, and this makes DHP methods extremely efficient.

Furthermore, it can be proven that if the value gradient is perfectly known along a greedy trajectory, i.e. if the DHP error $\vec{e}_t = \vec{0}$ at all time steps along a trajectory generated by a greedy policy, then that trajectory will be locally extremal, and often locally optimal. The proof for this is given by [10], and

[1] This contains a fix to the version that was published in IJCNN12. In line 6 of Algorithm 2, the learning rate α was missing in the published version.

the proof is closely related to Pontryagin's Maximum Principle [17].

The first order Taylor series approximation used here becomes better and better as the time step for sampling the underlying physical system gets smaller. It becomes perfect in continuous time, and this is what the value gradient policy of [16] uses.

This section was intended to show the motivation for DHP when using a greedy policy. The motivation for using it with a general action network is similar, because the objective of any actor training algorithm is to make the actor behave greedily, and greediness of the actor is a necessary condition for Bellman's Optimality Principle to apply. We demonstrate an actor-critic experiment in section IV-H.

Hence whether by a greedy policy, or by an actor-critic architecture, greedy behaviour needs to be learned eventually if Bellman's Optimality Principle is to be satisfied. And if greedy behaviour is to be learned in a continuous state space, then the value gradients need to be learned. In the case of DHP-based methods, these value-gradients are learned directly and without the need for explicit local exploration, using model-based formulae. In the case of VL methods, learning can be model-free, but the value-gradients must be learned indirectly, for example by learning the values on all of a group of neighbouring trajectories through explicit local exploration.

IV. Experiments

We now provide an experiment with a simple quadratic reward function to confirm the efficiency and exploration claims for DHP. We describe the experiment first for TD(0) and DHP using a greedy policy (in sections IV-A to IV-D). We also provide theoretical analysis in section IV-E of the two weight updates in this situation, to demonstrate further how TD(0) requires stochastic exploration to generate local exploration, but DHP does not. We then show, in sections IV-F and IV-G, that similar results would occur if TD(0) was replaced by the Sarsa or TD(λ) algorithms. And finally, in section IV-H, we give another version of the experiment using a neural network based actor critic architecture, and show that similar results happen in that situation too.

A. Problem Definition

We define an environment with $\vec{x} \in \Re$ and $\vec{a} \in \Re$. This is a simple quadratic optimisation problem. The model functions that we use to define the problem are:

$$f(x_t, t, a_t) = \begin{cases} x_t + a_t & \text{if } t = 0 \\ x_t & \text{if } t = 1 \end{cases} \tag{8a}$$

$$r(x_t, t, a_t) = \begin{cases} 0 & \text{if } t = 0 \\ -(x_t)^2 & \text{if } t = 1 \end{cases} \tag{8b}$$

Each trajectory is defined to terminate at time step $t = 2$, so that exactly two rewards are received by the agent (i.e. with the final reward being received on transitioning from $t = 1$ to $t = 2$). In these model function definitions, action a_1 has no effect, so the whole trajectory is parametrised by just x_0 and a_0. The total reward for this trajectory is $-(x_0 + a_0)^2$, so the optimal action to choose is $a_0 = -x_0$. These model functions are dependent on t, which is an abuse of notation we have adopted for brevity, but this could be legitimised by including t into $\vec{x}$.

In all the experiments we performed in this paper, to make exploration slightly harder, we forced the agent to always start at $x_0 = 0$. From this start point the optimal action is to do nothing (i.e. $a_0 = 0$).

B. Approximate Value Function Definition

We define a very simple approximate value function as follows. It uses weight vector $\vec{w} = (w_1, w_2)^T$:

$$\widetilde{V}(x_t, t, \vec{w}) = \begin{cases} -(x_1)^2 + w_1 x_1 + w_2 & \text{if } t = 1 \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

This function is capable of representing the optimal value function, which for this problem is $V^*(x_t, t) = -(x_t)^2$ when $t = 1$.

The approximate value-gradient is found by differentiating equation 9 partially with respect to x, to give

$$\widetilde{G}(x_t, t, \vec{w}) = \begin{cases} -2x_1 + w_1 & \text{if } t = 1 \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

C. Greedy Policy with Added Noise

For this approximate value function, the greedy policy can be solved analytically. First we expand the approximate Q value function for this model and value function:

$$\widetilde{Q}(x_0, a_0, \vec{w}) = -(x_0 + a_0)^2 + w_1(x_0 + a_0) + w_2$$

Then the greedy policy requires that we maximise $\widetilde{Q}$ with respect to the action, giving:

$$\pi(x_0, \vec{w}) = w_1/2 - x_0 + X_\sigma \tag{11}$$

Here X_σ is a normally distributed random variable with mean zero and standard deviation σ, that we are deliberately adding to the greedy policy to force it to explore more. If $\sigma = 0$ then no noise will be added and the greedy policy will not explore at all.

We emphasise that the greedy policy function makes use of the *same* weight vector $\vec{w}$ in both the policy, π, and the critic, $\widetilde{V}$.

By considering what weights would force the greedy policy to produce the optimal action $a_0 = -x_0$ (in the absence of noise), we see that the optimal weights are given by ${w_1}^* = 0$.

D. Results for DHP and TD(0) with a Greedy Policy

Even though both DHP and TD(0) are traditionally defined for an actor critic scenario, in this initial experiment we use them both with a greedy policy. We do this because in this problem the greedy policy is a very simple analytical function, and we can therefore analyse the learning algorithms in greater detail (in section IV-E). In a later version of the experiment, in

section IV-H, we repeat the experiment with a full actor-critic architecture.

The results for each combination of learning parameters and learning algorithm were collated from 1000 trials. At the start of each trial, all weight components were initialised with a uniform random distribution over a range from -10 to $+10$. In each trial, the learning algorithm was applied iteratively, with the following stopping condition: A trial was considered a success when $|w_1 - {w_1}^*| < 10^{-7}$. A trial was considered a failure if $|\vec{w}| > 10^4$, or when the number of iterations exceeded 10^7. All algorithms used discount factor $\gamma = 1$.

Results for these experiments using the TD(0) and DHP algorithms are shown in Table I. A summary of these results is that:

- Out of the experiments that converged with 100% reliability, DHP had a convergence time of just one iteration but TD(0) had a convergence time of approximately 1700 iterations, making DHP faster by three orders of magnitude, in this experiment.
- TD(0) has 0% success rate when the policy exploration rate is set to zero. Hence TD(0) *requires* explicit exploration to solve this problem, and fails in this deterministic setting without stochastic exploration.
- DHP achieves 100% convergence even when the policy is deterministic.
- DHP is three orders of magnitude faster than TD(0) even when the DHP policy is stochastic.

E. Analysis of Behaviour of DHP and TD(0) Under a Greedy Policy

Due to the simplicity of the greedy policy used, we can find the analytical forms for the DHP and TD(0) weight updates. In this section we do this, to try to gain some insights into why the DHP method so clearly outperforms the TD(0) method in this problem, with respect to exploration requirements and speed.

When the greedy policy equation (eq. 11) is used to choose the action, and when following the model function of equation 8a (i.e. $x_1 = x_0 + a_0$) from a start state x_0, the agent will pass through a next state of

$$x_1 = w_1/2 + X_\sigma. \tag{12}$$

To analyse the DHP and TD(0) weight updates for this problem, first we note that the approximate value function (eq. 9) is only learnable at time step $t = 1$, hence the summations in both the DHP and TD(0) weight updates only apply at $t = 1$. For example, the DHP weight update (eq. 5) reduces in this case to:

$$\begin{aligned}
\Delta\vec{w} =& \alpha \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_1 \left(\left(\frac{Dr}{D\vec{x}}\right)_1 + \gamma \left(\frac{Df}{D\vec{x}}\right)_1 \widetilde{G}_2 - \widetilde{G}_1\right) \\
=& \alpha \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_1 \left(\left(\frac{Dr}{D\vec{x}}\right)_1 + 2x_1 - w_1\right) && \text{by eq. 10} \\
=& \alpha \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_1 \left(\left(\frac{\partial r}{\partial \vec{x}}\right)_1 \left(\frac{\partial r}{\partial a}\right)_1 + 2x_1 - w_1\right) && \text{by eq. 6} \\
=& \alpha \left(\frac{\partial \widetilde{G}}{\partial \vec{w}}\right)_1 (-2x_1 + 2x_1 - w_1) && \text{by eq. 8b} \\
\implies & \begin{pmatrix}\Delta w_1 \\ \Delta w_2\end{pmatrix} = \begin{pmatrix}-\alpha w_1 \\ 0\end{pmatrix} && \text{by eq. 10}
\end{aligned} \tag{13}$$

The above equation shows that for any learning rate $0 \leq \alpha \leq 1$, DHP will make w_1 travel directly in a straight line through weight space towards the optimal weight ${w_1}^* = 0$. This shows how DHP solves this problem so quickly, and regardless of the level of stochastic exploration.

For the TD(0) weight update (eq. 3), we also only need consider the $t = 1$ term of the sum, giving:

$$\begin{aligned}
\Delta\vec{w} =& \alpha \left(\frac{\partial \widetilde{V}}{\partial \vec{w}}\right)_1 (r_1 + \gamma \widetilde{V}_2 - \widetilde{V}_1) \\
=& \alpha \left(\frac{\partial \widetilde{V}}{\partial \vec{w}}\right)_t (r_1 + (x_1)^2 - w_1 x_1 - w_2) && \text{by eq. 9} \\
=& \alpha \left(\frac{\partial \widetilde{V}}{\partial \vec{w}}\right)_t (-w_1 x_1 - w_2) && \text{by eq. 8b} \\
\implies & \begin{pmatrix}\Delta w_1 \\ \Delta w_2\end{pmatrix} = \alpha \begin{pmatrix}x_1 \\ 1\end{pmatrix} (-w_1 x_1 - w_2) && \text{by eq. 9} \\
=& \alpha \begin{pmatrix}-w_1 (x_1)^2 - w_2 x_1 \\ -w_1 x_1 - w_2\end{pmatrix} \\
=& \alpha \begin{pmatrix}-w_1 (w_1/2 + X_\sigma)^2 - w_2 (w_1/2 + X_\sigma) \\ -w_1 (w_1/2 + X_\sigma) - w_2\end{pmatrix} && \text{by eq. 12}
\end{aligned}$$

This weight update is dependent on a random variable, X_σ. But we can average out the stochastic effects of X_σ to calculate the *expectation* of the above weight update, and hence calculate the overall drift of the weight vector as the learning algorithm makes progress. Here we use $\mathbb{E}(X_\sigma) = 0$ and $\mathbb{E}\left((X_\sigma)^2\right) = \sigma^2$ and the linear rule for expectations, i.e. $\mathbb{E}(aX_\sigma + b) = a\mathbb{E}(X_\sigma) + b$, to get

$$\mathbb{E}\left(\begin{pmatrix}\Delta w_1 \\ \Delta w_2\end{pmatrix}\right) = \alpha \begin{pmatrix}-(w_1)^3/4 - w_1\sigma^2 - w_1 w_2/2 \\ -(w_1)^2/2 - w_2\end{pmatrix}$$

This averaged weight update for TD(0) is a more complex dynamical system for the weight vector than the corresponding DHP weight update (eq. 13), and it is more difficult to solve and find the trajectory through weight space. But the Δw_2 part of the equation is continually aiming to achieve a fixed point of $w_2 = -(w_1)^2/2$, and when this is attained the averaged weight update for w_1 simplifies to

$$\mathbb{E}(\Delta w_1) = -\alpha w_1 \sigma^2.$$

This is the almost the same as the DHP equation (eq. 13), except the learning rate is proportional to σ^2. This shows nicely how when the exploration rate σ drops to zero, the TD(0) weight update fails to emulate DHP and fails to learn the optimal weights ${w_1}^* = 0$. This confirms the main points that this paper is trying to show, i.e. that DHP is direct, fast

Policy noise (σ)	$\alpha = 0.01$			$\alpha = 0.1$			$\alpha = 1.0$		
	Success rate	Iterations (Mean)	Iterations ($\pm$ s.e.)	Success rate	Iterations (Mean)	Iterations ($\pm$ s.e.)	Success rate	Iterations (Mean)	Iterations ($\pm$ s.e.)
Results for algorithm TD(0)									
10	68.5%	1081.4	11	0.0%			0.0%		
1	**100.0%**	**1697.8**	11.1	85.9%	162.4	1.0	3.0%	189.6	19.3
0.1	100.0%	174234		90.4%	17223	104	15.5%	1519.1	9.8
0	**0.0%**			**0.0%**			**0.0%**		
Results for algorithm DHP									
10	100.0%	1734.8	3.3	100.0%	166.1	0.3	100.0%	1	0
0	100.0%	1738.2	3.0	100.0%	166.1	0.3	**100.0%**	**1**	0

TABLE I
RESULTS FOR DHP AND TD(0) WITH A GREEDY POLICY, AS DESCRIBED IN SECTION IV-D. (S.E.=STANDARD ERROR)

and does not need to explicitly perform local exploration to succeed; but when stochastic exploration is set to zero ($\sigma = 0$), TD(0) learning will fail to make any progress towards its intended goal.

F. Use of the Sarsa Algorithm

In the previous experiment, the use of a greedy policy with TD(0) may have been unconventional. However we can repeat the above experiment with the Sarsa algorithm [5] and obtain identical results, as we prove here.

Sarsa is an algorithm for control problems that learns to approximate the $\widetilde{Q}(\vec{x}, \vec{a}, \vec{w})$ function. The policy used is intended to be dependent on the $\widetilde{Q}(\vec{x}, \vec{a}, \vec{w})$ function (e.g. the greedy policy or a greedy policy with added stochastic noise), so Sarsa is ideal for control problems. The Sarsa weight update is defined to be:

$$\Delta\vec{w} = \alpha \sum_t \left(\frac{\partial \widetilde{Q}}{\partial \vec{w}}\right)_t (r_t + \gamma \widetilde{Q}_{t+1} - \widetilde{Q}_t) \qquad (14)$$

Sarsa is designed with $\widetilde{Q}$ represented by an arbitrary function approximator. We choose to define our $\widetilde{Q}$ function exactly by equation 2. Substituting this into the Sarsa weight update (eq. 14), and simplifying, gives

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

which is identical to TD(0) but with summation over t now excluding $t = 0$, and with an extra constant factor, γ^2.

The experiment described above in this paper used $\gamma = 1$, and has no weight update term for $t = 0$, so the results apply to both TD(0) *and* Sarsa (provided the $\widetilde{Q}$ function used by Sarsa is given exactly by eq. 2).

We stress that we did not disadvantage Sarsa by giving it $\widetilde{Q}$ defined exactly by eq. 2 using the known model functions, as this is only giving it extra correct model information, which should be beneficial, not detrimental.

G. Using Eligibility Traces

The RL community is familiar with extensions to TD(0) and Sarsa, which use "eligibility traces". These are the well known algorithms TD(λ) and Sarsa(λ) (see [4], [5] for details). Similarly VGL(λ) is an extension to DHP that uses eligibility traces. [10], [11] give further details, and pseudocode for both on-line and batch-mode implementations.

However using eligibility traces would not make any difference to the results of this particular experiment. This is because the approximate value function defined in equation 9 is only learnable at time step $t = 1$; and it is perfectly known at all time steps after that (i.e. it is perfectly known at time step $t = 2$). Hence the total *actual* future reward after $t = 1$ is identical to the total *approximated* future reward after $t = 1$. This choice between "actual" and "approximated" is what λ is designed to distinguish between. Hence the introduction of an eligibility trace learning constant λ will have no effect. This proves that the above results for TD(0), Sarsa and DHP are identical to the results we would get for TD(λ), Sarsa(λ) and VGL(λ) respectively.

H. Using a Neural-Network Based Actor-Critic Architecture

The results and conclusions of this paper are also applicable to actor-critic architectures, as we argued at the end of section III. To demonstrate this, we describe a version of the same TD(0)/DHP experiment (as defined in section IV-A) using one multi-layer perceptron (MLP, [18]) neural network for the actor and another MLP for the critic.

Both networks used a fully connected layered architecture with 2 inputs, 4 hidden units, and 1 output unit, and shortcut connections from the input layer to the output layer. The weights for both networks were initially randomised in [-1,1], and all activation units were hyperbolic tangent functions, except for the critic's output node which used an identity function. The input vector for each network was the full state vector, i.e. (x, t). The output of the critic network gave $\widetilde{V}$ directly, and the output of the action network plus noise X_σ gave the policy function.

The critic was trained by either the DHP or TD(0) weight update, with a learning rate of $\alpha = 0.1$. The actor was trained concurrently in each case by a common ADP weight update (e.g. [8, eq. 10]),

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

with a learning rate $\beta = 0.1$. This is an efficient model-based weight update that is unusual to be used in conjunction with TD(0), but again we stress that it is not disadvantaging TD(0) to be given access to this extra model information.

Neither critic learning algorithm attempted to learn the critic (or its gradient) at the final time-step of a trajectory ($t = 2$) since it is prior knowledge that the target critic value (or its gradient) is always 0 at any terminal state. Hence, in the TD(0) and DHP algorithms, we used $\widetilde{V}_2 \equiv 0$ and $\widetilde{G}_2 \equiv 0$, and in the actor weight update, we used $\left(\frac{\partial \widetilde{V}}{\partial \vec{x}}\right)_2 \equiv 0$.

Results showing learning performance for the two critic learning algorithms, both with policy noise and without policy noise, are given in Figure 1. The results were averaged over 100 experimental trials. The conclusions of these graphs are the same as before, i.e. that DHP can cope without stochastic exploration easily but TD(0) cannot; and DHP is many times faster.

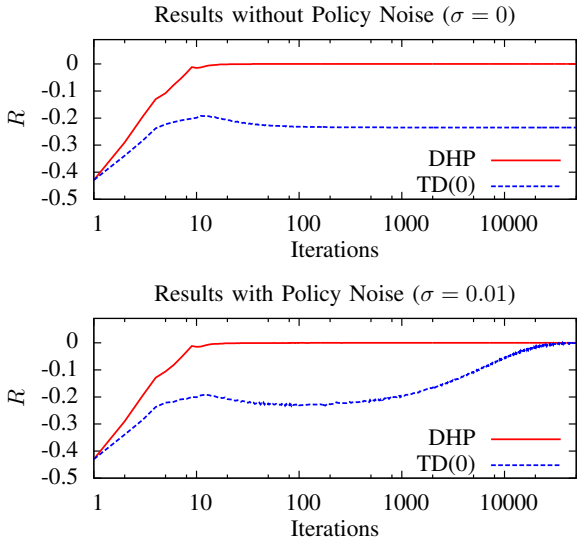

Fig. 1. Algorithm performances on the test problem using a neural network based actor critic architecture, as described in section IV-H; both with and without policy noise. The y axis shows the total total discounted long term reward, R. Compared to TD(0), the DHP method works well in the absence of stochastic exploration, and quickly attains optimal behaviour ($R = 0$). The TD(0) method fails without stochastic exploration here (in fact *it converges to a sub-optimal policy*), but does learn slowly and successfully in the presence of policy noise.

V. Conclusions

In this simple experiment, it was shown that DHP could successfully find an optimal solution without the explicit need for exploration, but the VL algorithms could not. It was also shown that when the VL algorithms were allowed to do their necessary exploration, in this case stochastically, they were slower by a factor of approximately 1700 when the quadratic function approximator was used, and by a similar factor when the actor-critic neural network architecture was used.

This is the motivation for DHP-based methods, and shows how they differ significantly from other RL methods. With DHP-based methods:

1) Local exploration comes automatically. "Exploration versus exploitation" becomes "exploration *and* exploitation" when following a greedy policy, locally at least.
2) We can get potentially much greater efficiency.
3) When DHP makes progress in learning the value gradient along a single trajectory, the trajectory will start to automatically bend itself towards a locally optimal shape, when used with a greedy policy. When TD(0) learns just the values along a single trajectory, the trajectory will not necessarily bend itself towards a locally optimal shape.

The results also show DHP-based methods out-performing regular RL methods both with and without stochastic policies.

Despite it possibly having been a foregone conclusion that a model-based algorithm (DHP) would outperform a model-free algorithm (TD(0)), it does *not* seem so obvious that items 1 and 3 in the above list would be the case. It is these two items that we particularly want to highlight as lesser known motivations for DHP.

This experiment was designed to highlight the key benefits of choosing DHP, and this was a simple problem in that there were no local optima to become trapped in, and all functions were differentiable. We emphasise that in any more complex differentiable environment, DHP will successfully find locally optimal trajectories without explicit exploration, i.e. in situations where TD(0) will fail. We also point out that DHP has also been used in much more complex problems: DHP successes include autopilot landing [8], power system control [19], simple control benchmark problems such as "pole balancing" [20], and many others [1].

The important caveats to be made about DHP are that:

- It assumes the cost of model learning is low compared to the cost of value-function learning.
- It assumes the functions $f(\vec{x}, \vec{a})$, $r(\vec{x}, \vec{a})$ and $\pi(\vec{x}, \vec{w})$ are differentiable.

These caveats apply to DHP but not to TD(0). Therefore it could have been unfair to have made the experimental comparison between them. But the experiment is intended to show that when these conditions *are* satisfied, and when working in continuous spaces, then the use of DHP could be extremely beneficial.

On the issue of whether it is worth learning the model in order to use DHP, we agree with a quote by [21, sec 4.3] concerning the use of DHP: "We mention that some view this model dependence to be an unnecessary 'expense'. The position of the authors, however, is that the expense is in many contexts more than compensated for by the additional information available to the learning/optimization process."

Acknowledgments

The authors would like to thank Danil Prokhorov and Peter Dayan for their comments on this manuscript.

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

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

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

[5] 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.

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

[7] 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.

[8] D. Prokhorov and D. Wunsch, "Adaptive critic designs," *IEEE Transactions on Neural Networks*, vol. September, pp. 997–1007, 1997.

[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] 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

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

[12] B. A. Pearlmutter, "Fast exact multiplication by the Hessian," *Neural Computation*, vol. 6, no. 1, pp. 147–160, 1994.

[13] P. J. Werbos, "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.

[14] 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.

[15] R. Munos, "Policy gradient in continuous time," *Journal of Machine Learning Research*, vol. 7, pp. 413–427, 2006.

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

[17] I. N. Bronshtein and K. A. Semendyayev, *Handbook of Mathematics*, 3rd ed. Van Nostrand Reinhold Company, 1985, ch. 3.2.2, pp. 372–382.

[18] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[19] 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.

[20] 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.

[21] G. G. Lendaris and J. C. Neidhoefer, "Guidance in the use of adaptive critics for control," *Handbook of learning and approximate dynamic programming, editors Jennie Si et al.*, pp. 97–124, 2004.