Brief Papers

Simple and Fast Calculation of the Second-Order Gradients for Globalized Dual Heuristic Dynamic Programming in Neural Networks

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

***Abstract*— We derive an algorithm to exactly calculate the mixed second-order derivatives of a neural network's output with respect to its input vector and weight vector. This is necessary for the adaptive dynamic programming (ADP) algorithms globalized dual heuristic programming (GDHP) and value-gradient learning. The algorithm calculates the inner product of this second-order matrix with a given fixed vector in a time that is linear in the number of weights in the neural network. We use a "forward accumulation" of the derivative calculations which produces a much more elegant and easy-to-implement solution than has previously been published for this task. In doing so, the algorithm makes GDHP simple to implement and efficient, bridging the gap between the widely used DHP and GDHP ADP methods.**

***Index Terms*— Adaptive dynamic programming, dual heuristic programming, neural networks, value-gradient learning.**

I. Introduction

ADAPTIVE DYNAMIC PROGRAMMING (ADP) [1] is an important methodology closely related to reinforcement learning, which is successfully used for solving control problems in industry and research, including recent work by [2]–[5]. Dual heuristic dynamic programming (DHP) and globalized DHP (GDHP) [6]–[8] are two important algorithms used in ADP.

Out of these two algorithms, DHP has been used far more often than GDHP in successful applications. For example, DHP has been used in control problems [7], [9], power grid control [10], [11], and many other applications [1]. One of the reasons for the comparatively low take-up of GDHP is that it is more challenging to implement than DHP. The difficult step of implementation is to correctly and efficiently calculate the second-order mixed partial derivatives of a neural network's scalar output $y = y(\vec{x}, \vec{w})$ with respect to its input vector $\vec{x}$ and weight vector $\vec{w}$, i.e., the matrix $(\partial^2 y/\partial\vec{w}\partial\vec{x})$ (where we define this second derivative notation such that this matrix has the element at (i, j) equal to $(\partial^2 y/\partial\vec{w}^j\partial\vec{x}^i)$). This matrix is related to, but slightly different from, the usual Hessian matrix $(\partial^2 y/\partial\vec{w}\partial\vec{w})$ described in the neural network literature (e.g., [12, Sec. 4.10]).

The GDHP algorithm, which we describe in detail in Section IV, only requires this second derivative matrix $(\partial^2 y/\partial\vec{w}\partial\vec{x})$ as an inner product $(\partial^2 y/\partial\vec{w}\partial\vec{x})\vec{k}$, where $\vec{k}$ is a column vector constant of dimension $\dim(\vec{x})$. To form this inner product by matrix-vector multiplication would take time $O(\dim(\vec{w})\dim(\vec{x})^2)$. In this brief, we provide a very clear and straightforward algorithm to calculate this inner product directly and exactly in an asymptotically faster time of $O(\dim(\vec{w}))$.

Existing literature does briefly outline an equally efficient method to calculate the required inner product, but this outline is only in the form of schematic diagrams [7] or a high-level description of generic backpropagation [13], [14]. Simple pseudocode applicable for a generic feed-forward neural network is not available there.

All of these existing descriptions calculate the required second derivatives by applying the mathematical technique of backpropagation twice to the neural network's feed-forward equations [15]. Using the terminology of "automatic differentiation" [16], [17], this is called a reverse accumulation of the derivatives. But it is not a trivial task to create a correct implementation of this for the given error function. In fact, this difficulty is thought to be one of the reasons why the equations for the required second-order derivatives have not been published earlier for a generic neural network.

Our method is to do a forward accumulation of the derivatives. This is much easier to derive and implement, and equally efficient. Our method follows the technique and terminology of [18], which is used to calculate an inner product of the Hessian matrix of a neural network in a fast and exact manner, without explicitly finding the full Hessian matrix itself.

The difference in simplicity in derivation between the forward and backward accumulation methods is very significant, as is illustrated by the way the two techniques were used to find fast products with the Hessian matrix in the neural network literature. Here, the backward accumulation is described by [19], and the derivation takes several pages (dense with lemmas and equations). The forward accumulation is described by [18], and the derivation takes just one page to define a differential operator ("*R*"), which then is used to produce the five lines of the algorithm instantly. Despite the technically demanding accomplishment of [19], it is the vastly simpler technique of [18] that receives nearly all of the citations in the literature.

After making some necessary modifications to the technique of [18], we achieve an algorithm that is almost trivial to derive and easy to state in pseudocode. By stating this forward accumulation method for GDHP clearly and giving the pseudocode for the resulting algorithm, we intend to solve the problem of GDHP being more difficult to implement than DHP.

Manuscript received August 5, 2011; revised May 8, 2012; accepted June 7, 2012. Date of publication July 13, 2012; date of current version September 10, 2012.

M. Fairbank and E. Alonso are with the Department of Computing, School of Informatics, City University London, London EC1V OHB, U.K. (e-mail: michael.fairbank@virgin.net; E.Alonso@city.ac.uk).

D. Prokhorov is with Toyota Research Institute NA, Ann Arbor, MI 48104 USA (e-mail: dvprokhorov@gmail.com).

Digital Object Identifier 10.1109/TNNLS.2012.2205268

Besides being useful for GDHP, the quantity $\vec{k}^T(\partial^2 y/\partial\vec{w}\partial\vec{x})$ is also useful in the general circumstance of trying to adjust the weights of a neural network so as to force the gradient $(\partial y/\partial\vec{x})$ to equal a given target value at a given $\vec{x}$. To achieve this, we would do gradient descent on an error function $E = (1/2)|(\partial y/\partial\vec{x}) - \vec{t}|^2$, where $\vec{t}$ is the "target" gradient. In this case, we would choose the constant vector $\vec{k}$ by

$$\vec{k} = \frac{\partial E}{\partial(\partial y/\partial\vec{x})} = \frac{\partial y}{\partial\vec{x}} - \vec{t}. \tag{1}$$

We give an example of an application like this in Section III-B.

An extension algorithm to GDHP is value-gradient learning [20]–[22], and this can require a further inner product $\vec{k}^T(\partial^2 y/\partial\vec{x}\partial\vec{x})$, which is also produced by the algorithm.

In the rest of this brief, in Section II we define the neural network and gradient-finding algorithms. In Section III, we give experimental results for a neural network problem, and in Section IV we define ADP and show how our algorithm can be used with GDHP. Finally, in Section V we give conclusions.

II. Algorithms

In this section, we describe the algorithm to find the second-order gradients that this brief requires. We define a very general feed-forward neural network architecture in Section II-A, and derive the second derivatives for it in Section II-B. We describe how the method can be extended to find the full second derivative matrices in Section II-C.

The way we derive the second-derivative matrices for this neural network is a general technique that could be applied to any existing feed-forward network structure (or even a recurrent neural network that has been unrolled using back-propagation through time), and we describe how we would validate any algorithm's correctness in Section III-A.

A. Feed-Forward Neural Network Architecture and Backpropagation

Lines 2 to 8 of Algorithm 1 implement a general neural network $y(\vec{x}, \vec{w})$, which has a single input layer, and is fully connected with all shortcut connections. The algorithm takes an input vector $\vec{x}$ and weight vector $\vec{w}$, and, for the purposes of this brief, produces a scalar output y. Pruned or layered network architectures are possible by fixing specific weights to zero. Fig. 1 illustrates an example network attainable by the algorithm.

In the algorithm, superscripts on variable names indicate the node number, for $j = 0, \ldots, n$, where n is the final node in the network. The variables a^j represent the firing activations of node j, and the final node's activation a^n gives the network's output y. Node 0 is a dedicated "bias" node, with a fixed value of $a^0 = 1$. $\vec{x} = (x^1, x^2, \ldots, x^m)$ is the external input vector to the whole network, with dimension $m = \dim(\vec{x})$. We use the notation $w^{k,j}$ to indicate the weight within $\vec{w}$ that connects node k to node j. s^j, δa^j, and δs^j are workspace scalars for each node.

$g(x) : \Re \to \Re$ is the "activation function," which is often taken to be $g(x) = \tanh(x)$, or the logistic function $g(x) = (1/1 + e^{-x})$. $g'(x)$ and $g''(x)$ are its first and second derivatives.

Algorithm 1 Feed-Forward Dynamics of a Neural Network Followed by First-Order Error Backpropagation to Calculate $(\partial y/\partial\vec{x})$

```
1:  {Feed-forward input vector $\vec{x}$ through network...}
2:  $a^0 \leftarrow 1$ {Bias node}
3:  $\forall j : 1 \le j \le \dim(\vec{x}),\ a^j \leftarrow x^j$ {Input vector, $\vec{x}$}
4:  for $j = \dim(\vec{x}) + 1$ to $n$ do
5:      $s^j \leftarrow \sum_{m=0}^{j-1} w^{m,j} a^m$
6:      $a^j \leftarrow g(s^j)$
7:  end for
8:  $y \leftarrow a^n$ {network output}
9:  {Backpropagation loop...}
10: for $j = n$ to 1 step $-1$ do
11:     $\delta a^j \leftarrow \begin{cases} 1, & \text{if } j = n \\ \sum_{m=j+1}^{n} (w^{j,m})(\delta s^m), & \text{otherwise} \end{cases}$
12:     $\delta s^j \leftarrow (\delta a^j) g'(s^j)$
13:     $\forall m : 0 \le m < j,\ (\partial y/\partial w^{m,j}) \leftarrow (\delta s^j) a^m$
14: end for
15: $\forall j : 1 \le j \le \dim(\vec{x}),\ (\partial y/\partial x^j) \leftarrow \delta a^j$ {output vector}
```

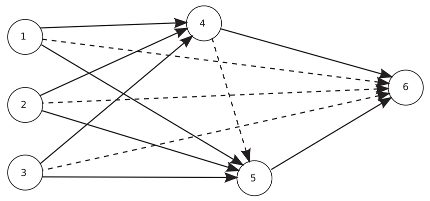

Fig. 1. Example neural network architecture obtainable by Algorithm 1, with three input nodes and one output node. When all graph edges are included, we have a fully connected feed-forward neural network, with all shortcut connections. If the dotted edges are removed (for example by clamping those weights to zero), then a more traditional layered network is obtained, containing a single hidden layer of two nodes. For the purpose of this brief, we always require a single output node, which in this example is node 6.

Lines 10–15 of Algorithm 1 do a backpropagation calculation, which calculates the gradients $(\partial y/\partial\vec{x})$ and $(\partial y/\partial\vec{w})$. This is the standard backpropagation algorithm for neural networks [15], [23] but modified, following [24], so that the "error function" used is actually the network output y, and also so that the backward pass continues until it has fully generated the quantity $(\partial y/\partial\vec{x})$, as required.

B. Finding Second Derivatives Using the R Operator

To find the second-order derivatives that this brief is aiming to produce, first we note that we can swap the order of differentiation. Hence, our desired second derivatives can be written as $\vec{k}^T(\partial/\partial\vec{x})(\partial y/\partial\vec{w})$ and $\vec{k}^T(\partial/\partial\vec{x})(\partial y/\partial\vec{x})$, respectively. This trick makes it easy to define a differential operator R, analogous to that used by [18].

Hence we define $R(.)$ as the differential operator

$$R = \sum_i k^i \frac{\partial}{\partial x^i} \tag{2}$$

where k^i is the ith component of $\vec{k}$.

Using this operator, the two second derivatives that we seek can now be written as $R(\partial y/\partial\vec{w})$ and $R(\partial y/\partial\vec{x})$, respectively.

Algorithm 2 Calculation of the Second-Order Derivatives

```
Require: $s^j$, $a^j$, $\delta s^j$ and $\delta a^j$ values calculated according to
    Algorithm 1 for all nodes $j$, and vector $\vec{k}$ calculated by an
    appropriate equation [e.g., (1)]
1: {Forward pass...}
2: $R(a^0) \leftarrow 0$
3: $\forall j : 1 \le j \le \dim(\vec{x}),\ R(a^j) \leftarrow k^j$
4: for $j = \dim(\vec{x}) + 1$ to $n$ do
5:     $R(s^j) \leftarrow \sum_{m=0}^{j-1} w^{m,j} R(a^m)$
6:     $R(a^j) \leftarrow g'(s^j) R(s^j)$
7: end for
8: {Backward pass...}
9: for $j = n$ to 1 step $-1$ do
10:    $R(\delta a^j) \leftarrow \begin{cases} 0, & \text{if } j = n \\ \sum_{m=j+1}^{n} (w^{j,m}) R(\delta s^m), & \text{otherwise} \end{cases}$
11:    $R(\delta s^j) \leftarrow R(\delta a^j) g'(s^j) + (\delta a^j) g''(s^j) R(s^j)$
12:    $\forall m : 0 \le m < j,\ R(\partial y/\partial w^{m,j}) \leftarrow R(\delta s^j)(a^m) + (\delta s^j) R(a^m)$ {Output vector 1: $\vec{k}^T(\partial^2 y/\partial \vec{w} \partial \vec{x})$}
13: end for
14: $\forall j : 1 \le j \le \dim(\vec{x}),\ R(\partial y/\partial x^j) \leftarrow R(\delta a^j)$ {Output
    vector 2: $\vec{k}^T(\partial^2 y/\partial \vec{x} \partial \vec{x})$}
```

Also, applying this R operator to $\vec{x}$ gives

$$R(\vec{x}) = \sum_i k^i \frac{\partial}{\partial x^i} \vec{x} = \vec{k}. \tag{3}$$

As detailed by [18], R obeys the usual rules for a differential operator, i.e., it obeys the product rule, the chain rule, the sum rule for derivatives, and differentiating a constant gives zero. For example, the weight vector $\vec{w}$ is a constant with respect to the R operator; thus $R(\vec{w}) = \vec{0}$. Using these rules, and (3), we apply the R operator to each side of each line of Algorithm 1 to obtain each line of Algorithm 2, respectively. We emphasize that applying the R operator to each line of the algorithm, while obeying the correct rules for differential operators, is calculating the exact derivatives that we are seeking. See [18, Sec. 3] for further explanation of the exactness of the R method.

Hence, Algorithm 2 exactly calculates the quantities $R(\partial y/\partial \vec{w})$ and $R(\partial y/\partial \vec{x})$, which are the two second-derivative inner products that we seek.

When implementing the algorithm, $R(a^j)$, $R(s^j)$, $R(\delta a^j)$, and $R(\delta s^j)$ are workspace scalars for each node j.

C. Generating the Full Second-Derivative Matrices

The above algorithm generates the inner products of two second-order derivative matrices with a constant vector. If instead of this inner product, the full second-order derivative matrices are required, then these can be constructed one column at a time. Execution of the above algorithms with $\vec{k}$ equal to the jth Euclidean standard basis vector of dimension $\dim(\vec{x})$ will calculate the jth column of each matrix. Thus accumulating the full second-order matrices, column by column, would take $O(\dim(\vec{w}) \dim(\vec{x}))$ time.

An alternative algorithm to find the full matrix $(\partial^2 y/\partial \vec{w} \partial \vec{x})$ is given for a neural network with just one hidden layer by [7, eqs. (14) and (15)], and for a general network by [25, Appendix A.4]. These also have asymptotic timings of $O(\dim(\vec{w}) \dim(\vec{x}))$.

III. Experimental Results

We describe how we validated the correctness of the algorithm in Section III-A, and describe a simple experiment that shows how a neural network can be forced to learn a target quantity $(\partial y/\partial \vec{x})$ in Section III-B.

A. Numerical Verification of the Algorithm

To validate that the algorithm was calculating the correct second-order gradients, we used numerical differentiation. This method provides a flexible and powerful check of the correctness of the program. Since the R method could be applied to other neural network architectures, it would be advisable to verify any other implementation in a similar manner.

To do the numerical confirmation, we first verified the formation of $(\partial y/\partial \vec{x})$ and $(\partial y/\partial \vec{w})$ taking place in lines 10–15 of Algorithm 1, by differentiating y numerically with respect to both $\vec{x}$ and $\vec{w}$, respectively. For example, to verify the first of these, we used a central-differences numerical derivative

$$\frac{\partial y}{\partial x^i} = \frac{y(\vec{x} + \epsilon \vec{e}_i, \vec{w}) - y(\vec{x} - \epsilon \vec{e}_i, \vec{w})}{2\epsilon} + O\left(\epsilon^2\right)$$

where ϵ is a small positive constant, and $\vec{e}_i$ is the ith Euclidean standard basis vector.

We next checked the second derivatives found by Algorithm 2 against a first-order numerical differentiation of the (already tested) first-order analytical derivatives found by Algorithm 1. For example

$$\vec{k}^T \frac{\partial}{\partial \vec{x}} \left(\frac{\partial y}{\partial \vec{w}} \right) = \frac{\left.\frac{\partial y}{\partial \vec{w}}\right|_{(\vec{x} + \epsilon \vec{k}, \vec{w})} - \left.\frac{\partial y}{\partial \vec{w}}\right|_{(\vec{x} - \epsilon \vec{k}, \vec{w})}}{2\epsilon} + O\left(\epsilon^2\right) \tag{4}$$

where $(\partial y/\partial \vec{w})$ is calculated by Algorithm 1 each time it is required in the right-hand side of this equation. This check was used to successfully confirm the correctness of Algorithm 2. For example, using a layered neural network with four inputs, three hidden layers of four units each, one output layer with one unit, hyperbolic tangent activation functions at all nodes, and all components of $\vec{w}$, $\vec{k}$, and $\vec{x}$ randomized uniformly in $[-1, 1]$, we found the average magnitude of the vector $\vec{k}^T(\partial^2 y/\partial \vec{w} \partial \vec{x})$ was 1.4, and the average magnitude of the error in this vector [between its value calculated by Algorithm 2 and its value calculated by (4)] was 6.4×10^{-10}, when $\epsilon = 10^{-5}$.

B. Wave Learning Experiment

In this experiment, we provide an example of how the algorithms of this brief can be used to adjust the weights of a neural network so as to force the gradient $(\partial y/\partial \vec{x})$ to equal a given target value at a given $\vec{x}$.

The objective here is to make a neural network with one input and one output learn the training data given in Table I.

TABLE I
TRAINING DATA FOR EXPERIMENT 1

x_p (network input)	s_p (target for y)	t_p [target for $(\partial y/\partial \vec{x})$]
0.00	0	20
0.25	0	−20
0.50	0	20
0.75	0	−20
1.00	0	20

In this training data, each row of the table is a different "pattern" p. Each pattern consists of an input value for the neural network (x_p), a target output value (s_p), and a target for the gradient $(\partial y/\partial \vec{x})$ (t_p). This experiment is aiming to make a neural network learn two complete periods of a sine wave from just five training patterns positioned along the x-axis.

Learning took place by minimizing the error function given in (5)

$$E=\frac{1}{2}\sum_{p=1}^{5}\left[\eta_1\left(y(x_p,\vec{w})-s_p\right)^2+\eta_2\left(\left.\frac{\partial y}{\partial \vec{x}}\right|_{(x_p,\vec{w})}-t_p\right)^2\right]. \quad (5)$$

η_1 and η_2 are real constants to weight the relative significance of the two terms in this equation. The first term is an ordinary sum-of-squares error for making the network output $y(x,\vec{w})$ match the target output for each pattern. The gradient of this part of the error function (with respect to $\vec{w}$) would be found by ordinary backpropagation. The second term of E is a sum-of-squares error term for making the gradient $(\partial y/\partial \vec{x})$ match the target gradient t_p for each pattern p. Hence this second term would form the vector $\vec{k}$ described in (1) by

$$\vec{k}=\left.\frac{\partial y}{\partial \vec{x}}\right|_{(x_p,\vec{w})}-t_p$$

for each pattern p, and then the required gradient for learning can be found by Algorithm 2.

The neural network used was a multilayer perceptron, with two hidden layers of eight nodes each, and all shortcut connections present between all pairs of nonadjacent layers. The activation function used for all nodes was $g(x)=1/(1+e^{-x})$, except for the output node which used $g(x)=x$. Training used gradient descent on (5), with $\eta_1=\eta_2=1$, i.e., with the same significance attached to each component of the error function. Initial network weights were randomized uniformly from $[-0.1, 0.1]$.

When training was accelerated through conjugate gradients, with a full line search, learning produced a 100% convergence rate over 100 trials, where convergence was defined as E going below 10^{-5} within 20 000 iterations. The output of five typical neural networks trained in this way is shown in Fig. 2.

The results show that the problem has been solved correctly, and that it has been possible to make a neural network learn specified target gradients $(\partial y/\partial \vec{x})$ at given values of $\vec{x}$. However, when experimental parameters were changed, we observed that the success rate could drop significantly. For example if RPROP was used, and the activation functions used were swapped to $g(x)=\tanh(x)$, then the success rate dropped from 100% to 65%. It seems that it is harder to train

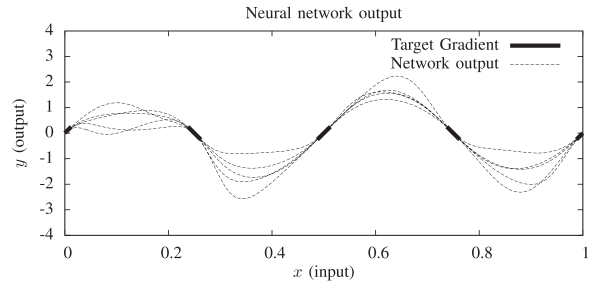

Fig. 2. Outputs from a sample of five neural networks created in the experiment of Section III-B. Each dotted curve shows the output from a neural network produced in a different trial. The solid thick lines on the x-axis are designed to give a visual indication of the training point and target gradient. The objective of training is to make the dotted lines run parallel and through the thick lines, which has been achieved well in all of the five network outputs in this figure.

a neural network to learn target gradients $(\partial y/\partial \vec{x})$ than the target values. By analogy, we can imagine that in trying to make the flexible curves of Fig. 2 bend into the shape of a sine wave, it is harder to do so by just twisting the curve at certain points along the x-axis than it is by stretching the curve to pass through given target points.

IV. ADP

In this section, we describe ADP, and in Sections IV-A and IV-B we define the algorithms GDHP and DHP, showing how the methods of this brief can be applied to GDHP. We describe a simple GDHP experiment in Section IV-C. However, we point out that, since the methods of this brief calculate $\vec{k}^T(\partial^2\tilde{J}/\partial\vec{w}\partial\vec{x})$ exactly, the empirical results in using GDHP with it will be identical to any other exact method.

In ADP, an agent moves in an environment such that at time t it has state vector $\vec{x}_t$. At each time the agent chooses an action $\vec{u}_t$ which takes it to the next state according to the environment's model function $\vec{x}_{t+1}=f(\vec{x}_t,\vec{u}_t)$, thus the agent passes through a trajectory of states $(\vec{x}_0,\vec{x}_1,\vec{x}_2,\ldots)$. On transitioning from each state $\vec{x}_t$ to the next, the agent receives an immediate scalar cost U_t from the environment according to the function $U_t=U(\vec{x}_t,\vec{u}_t)$.[1]

The ADP problem is for the agent to learn how to choose actions so as to minimize the expectation of the total long-term cost received, i.e., $J(\vec{x}_0)=\left\langle\sum_t \gamma^t U_t\right\rangle$, from any given start state $\vec{x}_0$, where $\gamma\in[0,1]$ is a constant discount factor that specifies the importance of long-term costs over short-term ones.[2] Specifically, the problem is to find an action network $A(\vec{x},\vec{z})$, where $\vec{z}$ is the parameter vector of a function approximator that calculates which action $\vec{u}=A(\vec{x},\vec{z})$ to take for any given state $\vec{x}$, such that this total long-term cost is minimized.

To aid training the action network, ADP algorithms make an intermediate goal to first learn a scalar "critic" function $\tilde{J}(\vec{x},\vec{w})$ over the state space. This function is the scalar output of a function approximator (e.g., a neural network with one output node) with parameter vector $\vec{w}$, and the objective of a

[1]Throughout this section, subscripts indicate the time step of a trajectory, and superscripts indicate a component of a vector.

[2]If the problem is such that the trajectory is not guaranteed to terminate, then to prevent an infinite total cost, γ should also satisfy $\gamma<1$.

critic-learning algorithm is for the critic function to be trained to equal the cost-to-go function $J(\vec{x})$ over all of the state space. The function J is also known as the value function from dynamic programming [26].

A. GDHP Algorithm

The GDHP algorithm is a critic learning algorithm that attempts to learn the function $\widetilde{J}(\vec{x},\vec{w})$ by making its gradient $(\partial\widetilde{J}/\partial\vec{x})$ explicitly match the gradient of the cost-to-go function $(\partial J/\partial\vec{x})$. For each time step t of a trajectory $(\vec{x}_0,\vec{x}_1,\vec{x}_2,\ldots)$, the GDHP algorithm is defined to be the following critic weight update:

$$\Delta\vec{w}^i = \eta_1\left(\frac{\partial\widetilde{J}}{\partial\vec{w}^i}\right)_t (U_t+\gamma\widetilde{J}_{t+1}-\widetilde{J}_t) + \eta_2\sum_j\left(\frac{\partial^2\widetilde{J}}{\partial\vec{w}^i\partial\vec{x}^j}\right)_t\left(\vec{E}_t\right)^j \tag{6}$$

where η_1 and η_2 are small positive learning rates corresponding to a learning rate for the values of J and a learning rate for the gradients $(\partial J/\partial\vec{x})$, respectively, where $\widetilde{J}(\vec{x},\vec{w})$ is the critic function, and $\vec{E}_t\in\mathbb{R}^{\dim(\vec{x})}$ is a vector defined to be

$$\vec{E}_t = \left(\tfrac{DU}{D\vec{x}}\right)_t + \gamma\sum_j\left(\tfrac{Df^j}{D\vec{x}}\right)_t\left(\tfrac{\partial\widetilde{J}}{\partial\vec{x}^j}\right)_{t+1} - \left(\tfrac{\partial\widetilde{J}}{\partial\vec{x}}\right)_t \tag{7}$$

and $(D/D\vec{x})$ is shorthand for

$$\frac{D}{D\vec{x}} \equiv \frac{\partial}{\partial\vec{x}} + \sum_j\frac{\partial A^j}{\partial\vec{x}}\frac{\partial}{\partial\vec{u}^j} \tag{8}$$

where all of these derivatives are assumed to exist. j is a free variable to indicate a vector component, so that for example A^j refers to the jth component of the output of the action network; the subscripted t indicates that a function is to be evaluated at the state $\vec{x}_t$ (and action $\vec{u}_t$, where relevant), so that, for example, $\widetilde{J}_{t+1}\equiv\widetilde{J}(\vec{x}_{t+1},\vec{w})$ and $(Df/Dx)_t$ is the derivative $(Df(\vec{x},\vec{u})/D\vec{x})$ evaluated at $(\vec{x}_t,\vec{u}_t)$.

Equations (6)–(8) define the GDHP algorithm. See [6] and [7] for further details.

In GDHP, the function $\widetilde{J}(\vec{x},\vec{w})$ would be the output of a neural network with one single output node, i.e., the function $\widetilde{J}$ takes the role of y in the descriptions of the rest of this brief. Consequently, the appearance of $(\partial^2\widetilde{J}/\partial\vec{w}\partial\vec{x})$ in (6) is a requirement for a calculation of the second-order gradients described by this brief. To use our method for GDHP, we would run Algorithm 2 with $\vec{k}=\vec{E}_t$ according to (7).

The other gradients of $\widetilde{J}$ appearing in (6) and (7), i.e., $(\partial\widetilde{J}/\partial\vec{w})$ and $(\partial\widetilde{J}/\partial\vec{x})$, can be found by Algorithm 1 (in lines 13 and 15, respectively). The derivatives of $f(\vec{x},\vec{u})$ and $U(\vec{x},\vec{u})$ should be available from the environment, since in the GDHP method we assume these functions are known and differentiable.

Once the critic function has been learned (or during concurrent learning of both the critic and the action network), the weight vector for the action network $A(\vec{x},\vec{z})$ can be updated by the following weight update:

$$\Delta\vec{z}^i = -\beta\sum_j\left(\tfrac{\partial A^j}{\partial\vec{z}^i}\right)_t\left(\left(\tfrac{\partial U}{\partial\vec{u}^j}\right)_t + \gamma\sum_k\left(\tfrac{\partial f^k}{\partial\vec{u}^j}\right)_t\left(\tfrac{\partial\widetilde{J}}{\partial\vec{x}^k}\right)_{t+1}\right) \tag{9}$$

where β is a separate learning rate for the action network. The multiplications by $(\partial A/\partial\vec{z})$ and $(\partial A/\partial\vec{x})$ in (8) and (9), respectively, can be done quickly and exactly by ordinary backpropagation.

B. DHP Algorithm and Its Relationship to GDHP

The DHP algorithm is almost identical to GDHP, but the scalar critic $\widetilde{J}(\vec{x},\vec{w})$ is removed, and instead a new neural network of input and output dimension $\dim(\vec{x})$ is used, which we call the vector critic, $\vec{\lambda}(\vec{x},\vec{w})$. The purpose of this vector critic is to simulate the gradient $(\partial\widetilde{J}/\partial\vec{x})$. Hence, the DHP weight update is similar to (6), but all occurrences of $(\partial\widetilde{J}/\partial\vec{x})$ are replaced by $\vec{\lambda}$, and the constant η_1 must be fixed at zero (since $\widetilde{J}_t$ is not defined). Hence the DHP critic weight update is

$$\Delta\vec{w}^i = \eta_2\sum_j\left(\tfrac{\partial\vec{\lambda}^j}{\partial\vec{w}^i}\right)_t\left(\vec{E}_t\right)^j \tag{10}$$

where both instances of $(\partial\widetilde{J}/\partial\vec{x})$ in (7) are replaced by $\vec{\lambda}$. A similar replacement is done for the DHP actor weight update (9).

The DHP weight update (10) is a much simpler weight update than the GDHP one of (6), since there is no requirement for second-order differentiation. This is why DHP has so far always been simpler to program than GDHP. Another advantage of DHP over GDHP is that training a neural network to learn target values can be easier than training one to learn target gradients (as discussed at the end of Section III-B). However, GDHP is possibly advantageous over DHP in that its critic function is a true scalar field, just like its intended target, the cost-to-go function $J(\vec{x})$, although it remains to be seen whether GDHP is advantageous in practice.

C. GDHP Experiment

We now provide an extremely simple quadratic optimization experiment using GDHP.

We define an environment with $\vec{x}\equiv x\in\Re$ and $\vec{u}\equiv u\in\Re$, and model and cost functions

$$f(x,t,u) = \begin{cases} x+u, & \text{if } t=0\\ x, & \text{if } t=1\end{cases}$$

$$U(x,t,u) = \begin{cases} 0, & \text{if } t=0\\ (x)^2, & \text{if } t=1.\end{cases}$$

Each trajectory is defined to terminate at time step $t=2$, so that exactly two costs are received by the agent (i.e., with the final cost being received on transitioning from $t=1$ to $t=2$). In these model function definitions, action u_1 has no effect, so the whole trajectory is parameterized by just x_0 and u_0. The total cost for this trajectory is $(x_0+u_0)^2$, so the optimal action to choose is $u_0=-x_0$.

The action network $A(\vec{x},\vec{z})$ was a layered neural network with two inputs, one output and one hidden layer of four nodes, shortcut connections from the input layer to the output layer, and with activation function $g(x)=\tanh(x)$ on all nodes. The weights $\vec{z}$ were initially randomly chosen from $[-0.1,0.1]$

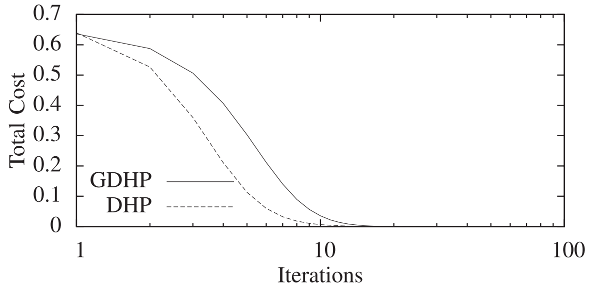

Fig. 3. Results for GDHP and DHP in solving the problem described in Section IV-C. Both GDHP and DHP manage to reduce the total cost to almost zero within 100 iterations.

uniformly. The critic network $\tilde{J}(\vec{x}, \vec{w})$ was identically dimensioned to the action network, with a weight vector $\vec{w}$ randomized initially in the same way. The activation function used for the critic was $g(x) = \tanh(x)$ on all nodes except for the output node, which used $g(x) = x$. The input vector to each neural network was (x, t), since in this problem the optimal cost-to-go function depends on both of these inputs.

We applied the GDHP and action network learning equations (6) and (9) concurrently, using learning rate constants $\eta_1 = 0$, $\eta_2 = 0.1$, and $\beta = 0.1$, and discount factor $\gamma = 1$. Each iteration of training consisted of the application of these two weight updates accumulated over all nonterminal time steps of the trajectory $t \in \{0, 1\}$. Each trajectory started from $x = 0.8$. Experimental results, for both GDHP and DHP, averaged over 10 trials are shown in Fig. 3. The graph shows that both algorithms can solve this problem in similar time. DHP is slightly faster, but, with this being such a simple test problem, the slight difference between the two algorithms is not significant.

V. Conclusion

We have presented a very clear and straightforward algorithm to find the product $\vec{k}^T(\partial^2 y/\partial\vec{w}\partial\vec{x})$ exactly in time $O(\dim(\vec{w}))$. We have found that using a forward accumulation of the derivatives leads to an extremely easy way to derive the algorithm, in comparison to a backward accumulation that GDHP practitioners have relied upon until now.

We have made the appropriate modifications to the R method of [18], enabling us to derive the algorithm quickly. We have provided an empirical demonstration of the problem of learning target gradients in a neural network, and we have placed emphasis on the applicability to GDHP, giving an example. It is the intention of this brief that implementing GDHP should be as straightforward as implementing DHP while having an equivalent $(O(\dim(\vec{w})))$ running time, too.

References

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

[2] F.-Y. Wang, N. Jin, D. Liu, and Q. Wei, "Adaptive dynamic programming for finite-horizon optimal control of discrete-time nonlinear systems with ϵ-error bound," *IEEE Trans. Neural Netw.*, vol. 22, no. 1, pp. 24–36, Jan. 2011.

[3] J. Fu, H. He, and X. Zhou, "Adaptive learning and control for MIMO system based on adaptive dynamic programming," *IEEE Trans. Neural Netw.*, vol. 22, no. 7, pp. 1133–1148, Jul. 2011.

[4] K. M. Iftekharuddin, "Transformation invariant on-line target recognition," *IEEE Trans. Neural Netw.*, vol. 22, no. 6, pp. 906–918, Jun. 2011.

[5] Y. Jiang and Z.-P. Jiang, "Approximate dynamic programming for optimal stationary control with control-dependent noise," *IEEE Trans. Neural Netw.*, vol. 22, no. 12, pp. 2392–2398, Dec. 2011.

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

[7] D. Prokhorov and D. Wunsch, "Adaptive critic designs," *IEEE Trans. Neural Netw.*, vol. 8, no. 5, pp. 997–1007, Sep. 1997.

[8] S. Ferrari and R. F. Stengel, "Model-based adaptive critic designs," in *Handbook of Learning and Approximate Dynamic Programming*, S. Jennie, B. Andy, P. Warren, and W. Donald, Eds. New York: Wiley, 2004, pp. 65–96.

[9] G. G. Lendaris and C. Paintz, "Training strategies for critic and action neural networks in dual heuristic programming method," in *Proc. Int. Conf. Neural Netw.*, Houston, TX, Jun. 1997, pp. 712–717.

[10] G. K. Venayagamoorthy and D. C. Wunsch, "Dual heuristic programming excitation neurocontrol for generators in a multimachine power system," *IEEE Trans. Ind. Appl.*, vol. 39, no. 2, pp. 382–394, Mar.–Apr. 2003.

[11] G. K. Venayagamoorthy, R. G. Harley, and D. C. Wunsch, "Comparison of heuristic dynamic programming and dual heuristic programming adaptive critics for neurocontrol of a turbogenerator," *IEEE Trans. Neural Netw.*, vol. 13, no. 3, pp. 764–773, May 2002.

[12] C. M. Bishop, *Neural Networks for Pattern Recognition*. London, U.K.: Oxford Univ. Press, 1995.

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

[14] P. J. Werbos, "Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research," *IEEE Trans. Syst. Man Cybern.*, vol. 17, no. 1, pp. 7–20, Jan. 1987.

[15] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Dept. Appl. Math., Harvard Univ., Cambridge, MA, 1974.

[16] L. B. Rall, *Automatic Differentiation: Techniques and Applications* (Lecture Notes in Computer Science), vol. 120. New York: Springer-Verlag, 1981.

[17] P. J. Werbos, "Backwards differentiation in AD and neural nets: Past links and new opportunities," in *Automatic Differentiation: Applications, Theory, and Implementations* (Lecture Notes in Computational Science and Engineering), H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, Eds. New York: Springer-Verlag, 2005, pp. 15–34.

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

[19] M. Møller, "Exact calculation of the product of the Hessian matrix of feed-forward network error functions and a vector in $O(N)$ time," Dept. Comput. Sci., Aarhus Univ., Århus, Denmark, Tech. Rep. DAIMI PB-432, 1993.

[20] M. Fairbank and E. Alonso, "Value-gradient learning," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, Jun. 2012, pp. 3062–3069.

[21] M. Fairbank. (2008). Reinforcement learning by value gradients. *CoRR* [Online]. Available: http://arxiv.org/abs/0803.3539

[22] M. Fairbank and E. Alonso. (2011). The local optimality of reinforcement learning by value gradients, and its relationship to policy gradient learning. *CoRR* [Online]. Available: http://arxiv.org/abs/1101.0428

[23] D. Rumelhart, G. Hinton, and R. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, Oct. 1986.

[24] J. Kindermann and A. Linden, "Inversion of neural networks by gradient descent," *Parallel Comput.*, vol. 14, no. 3, pp. 277–286, 1990.

[25] R. Coulom, "Reinforcement learning using neural networks, with applications to motor control," Ph.D. dissertation, Inst. National Polytechnique de Grenoble, Grenoble, France, 2002.

[26] R. E. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton Univ. Press, 1957.