Value Gradient Learning: VGL($\lambda$)
An extension to Dual Heuristic Programming.

M. Fairbank$^1$ E. Alonso$^1$

$^1$City University London
London

These slides accompany the paper “Value-Gradient Learning”, presented at
The International Joint Conference on Neural Networks (IJCNN),
Brisbane, June 2012

Last modified date: September 2012
We propose a new algorithm VGL(\(\lambda\)), which is the same as DHP but with a parameter \(\lambda\) included, where \(\lambda\) is analogous to the parameter \(\lambda\) used in TD(\(\lambda\)).
This presentation will cover...

- (The new algorithm is the one in the bottom right)

- **HDP is**
  \[ \Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \vec{w}} \right)_t (J'_t - \tilde{J}_t) \]
  with \( J'_t \) defined by
  \[ J'_t = U_t + \gamma \tilde{J}_{t+1} \]

- **DHP is**
  \[ \Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{G}}{\partial \vec{w}} \right)_t \Omega_t (G'_t - \tilde{G}_t) \]
  with \( G'_t \) defined by
  \[ G'_t = \left( \frac{DU}{D\vec{x}} \right)_t + \gamma \left( \frac{Df}{D\vec{x}} \right)_t \tilde{G}_{t+1} \]

- **TD(\( \lambda \)) is**
  \[ \Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \vec{w}} \right)_t (J'_t - \tilde{J}_t) \]
  with \( J'_t \) defined by
  \[ J'_t = U_t + \gamma \left( \lambda J'_{t+1} + (1 - \lambda) \tilde{J}_{t+1} \right) \]

- **VGL(\( \lambda \)) is**
  \[ \Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{G}}{\partial \vec{w}} \right)_t \Omega_t (G'_t - \tilde{G}_t) \]
  with \( G'_t \) defined by
  \[ G'_t = \left( \frac{DU}{D\vec{x}} \right)_t + \gamma \left( \frac{Df}{D\vec{x}} \right)_t (\lambda G'_{t+1} + (1 - \lambda) \tilde{G}_{t+1}) \]

This presentation will discuss the motivations for the 3 prior algorithms first.
Then the new algorithm VGL(\( \lambda \)). Plus a very surprising result!
1. Review of ADP problem and notation

2. Review of Existing ADP Algorithms, and their Motivations
   - The HDP Algorithm
   - The TD($\lambda$) Algorithm
   - What are the problems with TD($\lambda$)?
   - The DHP Algorithm

3. The New Algorithm: VGL($\lambda$)
   - What is VGL($\lambda$)?
   - How does VGL($\lambda$)/DHP compare to TD($\lambda$)/HDP

4. Experimental Results

5. The relationship of VGL(1) to Backpropagation Through Time
Note: 49 slides in 15 minutes, so delivery will be brisk!

- Important equations will be repeated over consecutive slides.
- Slides will be made available for download/review.
At each discrete timestep \( t \), agent has state vector \( \vec{x}_t \in \mathbb{R}^n \)

**Action vectors:** \( \vec{u}_t \in \mathbb{R}^m \)

**State transition function:** \( \vec{x}_{t+1} = f(\vec{x}_t, \vec{u}_t) \)

**State transition cost function:** \( U_t = U(\vec{x}_t, \vec{u}_t) \).

**Action network chooses actions:** \( \vec{u}_t = A(\vec{x}_t, \vec{z}) \)

\( \vec{z} \) is weight vector.

**Total cost-to-go** \( J(\vec{x}_0, \vec{z}) = \langle \sum_t \gamma^t U_t \rangle \)

**Objective:** Choose \( \vec{z} \) so that total cost \( J \) is minimised from any start point
A spacecraft is constrained to move in a vertical line under gravity $k_g$

State vector, $\vec{x} = (\text{height}(h), \text{velocity}(v))$

Action $u \in \mathbb{R}$ produces upward accelerations, with immediate "fuel usage" cost $U(\vec{x}, u) = kf_u$

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

On landing a final cost is given: $\frac{1}{2}mv^2$

So there are two objectives: To conserve fuel and land slowly.
Example ADP problem - spacecraft vertical lander

- The trajectory can be shown much more efficiently using a state space view.
- In the state space diagram, axes show velocity and height.
- **Blue curve shows an actual trajectory**, and **green curve shows the theoretical optimal trajectory**.
- In this example, the **actual trajectory** shows the velocity becoming more and more negative until the height reaches zero. This example shows the spacecraft crashing badly.
- In this example, the **optimal trajectory** shows the spacecraft freefalling for an initial time, and then using the thruster to slow the spacecraft down to land gently.
- Notice that **optimal behaviour** is to descend freely and then brake as hard and as late as possible; hovering down slowly would be wasteful of fuel.
Using a critic network

We consider ADP with a critic network.

- The Critic is a neural network $\tilde{J}(\vec{x}, \vec{w})$ with single output node.
- $\vec{w}$ is the weight vector
- ADP trains the critic to approximate the cost-to-go function:
  $\tilde{J}(\vec{x}, \vec{w}) \approx J(\vec{x}, \vec{z}) \quad \forall \vec{x}$

Once we have a critic defined:

- We train the actor to be greedy with respect to the critic:
  $A(\vec{x}, \vec{z}) = \arg \min_{\vec{u}} \left\{ U(\vec{x}, \vec{u}) + \gamma \tilde{J}(f(\vec{x}, \vec{u}), \vec{w}) \right\} \quad \forall \vec{x}$
- If the above two objectives met simultaneously then Bellman’s Optimality Condition is met: Optimal behaviour
Critic Training with Heuristic Dynamic Programming (HDP)

HDP is
\[
\Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \vec{w}} \right)_t (J'_t - \tilde{J}_t)
\]
with \(J'_t\) defined by
\[
J'_t = U_t + \gamma \tilde{J}_{t+1}
\]

- HDP = “TD(0)” (with a neural network for \(\tilde{J}(\vec{x}, \vec{w})\))
- Aims to train critic, i.e. achieve \(\tilde{J}(\vec{x}, \vec{w}) \equiv < J' > \ \forall \vec{x}\)
  \[\implies \tilde{J}_t = < U_t + \gamma \tilde{J}_{t+1} > \ \forall \vec{x}_t\]
  \[\implies \tilde{J}_t = < \sum_t \gamma^t U_t > \ \forall \vec{x}_t\]
  \[\implies \tilde{J}(\vec{x}, \vec{w}) \equiv J(\vec{x}, \vec{z}) \ \forall \vec{x} \text{ (Bellman satisfied)}\]
- We still need this to apply \(\forall \vec{x}\), and with a greedy policy, for Bellman’s Optimality condition to apply.
Critic Training with Hueristic Dynamic Programming (HDP)

\[ \Delta \tilde{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \tilde{w}} \right)_t \left( J'_t - \tilde{J}_t \right) \]

with \( J'_t \) defined by
\[ J'_t = U_t + \gamma \tilde{J}_{t+1} \]

- \( J' \) is the target for the weight update.
- It is not the final target: it is a *moving target* (since it is a function of \( \tilde{J} \) and \( A(\tilde{x}, \tilde{z}) \)).
- We want the \( \tilde{J} \) values to “chase” the \( J' \) values.
- So making \( \tilde{J} \) catch \( J' \) can be elusive (and proving convergence with neural network architectures is hard).
- But thinking in terms of targets \( J' \) is useful (for me, at least).
Extending HDP into TD(\(\lambda\))

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

with \( J'_t \) defined by

\[
J'_t = U_t + \gamma \tilde{J}_{t+1}
\]

insert \( \lambda \)

\[
\text{TD}(\lambda) \text{ is}\]

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

with \( J'_t \) defined by

\[
J'_t = U_t + \gamma \left( \lambda J'_{t+1} + (1 - \lambda) \tilde{J}_{t+1} \right)
\]
Extending HDP into TD(\(\lambda\))

\[
\Delta \tilde{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \tilde{w}} \right)_t (J'_t - \tilde{J}_t)
\]

with \(J'_t\) defined by

\[
J'_t = U_t + \gamma \tilde{J}_{t+1}
\]

**HDP is**

**TD(\(\lambda\)) is**

\[
\Delta \tilde{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \tilde{w}} \right)_t (J'_t - \tilde{J}_t)
\]

with \(J'_t\) defined by

\[
J'_t = U_t + \gamma (\lambda J'_{t+1} + (1 - \lambda) \tilde{J}_{t+1})
\]

- TD(\(\lambda\)) is an extension to HDP/TD(0), by Richard Sutton\(^a\)
- \(\lambda \in [0, 1]\) is constant.

\(^a\)“Learning to Predict by the Methods of Temporal Differences,” Machine Learning, 1988
TD(\(\lambda\)) is
\[
\Delta \tilde{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \tilde{w}} \right)_t (J'_t - \tilde{J}_t)
\]
with \(J'_t\) defined by
\[
J'_t = U_t + \gamma (\lambda J'_{t+1} + (1 - \lambda) \tilde{J}_{t+1})
\]

- This is a concise way of writing the TD(\(\lambda\)) algorithm.
- Equivalence of the \(J'\) recursion to the conventional implementation of TD(\(\lambda\)) is proven by Fairbank.\(^1\)
- \(J'\) is equivalent to the \(\lambda\)-return described by Chris Watkins\(^2\)
- Writing \(J'\) in our recursive form was key to deriving the VGL(\(\lambda\)) algorithm - because \(J'\) is relatively easy to differentiate.

\(^1\)“Reinforcement Learning by Value Gradients”, 2008, eprint arXiv:0803.3539
\(^2\)“Learning from Delayed Rewards”, PhD thesis, Cambridge University, 1989
What are motivations for TD(\(\lambda\))? 

\[
\text{TD}(\lambda) \text{ is } \\
\Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \vec{w}} \right)_t (J'_t - \tilde{J}_t) \\
\text{with } J'_t \text{ defined by } \\
J'_t = U_t + \gamma (\lambda J'_{t+1} + (1 - \lambda) \tilde{J}_{t+1})
\]

- When \(\lambda\) is large, the “look ahead” in the recursion for \(J'\) is larger, enabling faster learning (especially if there is a delayed significant cost arriving after many time steps).
- When \(\lambda\) is small, it reduces the variance in the sampled \(J'\) value, thus improving learning in stochastic environments.
- When \(\lambda = 1\), learning can be more likely to converge. For example when the action network is fixed, TD(1) is true gradient descent on 
\[
E = \frac{1}{2} \sum_t (J'_t - \tilde{J}_t)^2.
\]
- In contrast, TD(0) and HDP are not gradient descent on any analytical function.
What are the problems with TD(\(\lambda\))?

Actual behaviour of TD(\(\lambda\)) in a deterministic problem

- Here the HDP error is virtually zero \(\left(\sum_t (J'_t - \tilde{J}_t)^2 = 7.37 \times 10^{-5}\right)\), and this was using a greedy policy, but the actual trajectory is nowhere near the optimal trajectory.

- Values have been near-perfectly learned all along the trajectory, but greedy policy takes no notice.

- TD(\(\lambda\)) has converged to a suboptimal trajectory!
What are the problems with TD(\(\lambda\))?

Actual behaviour of TD(\(\lambda\)) in a deterministic problem

- What is the reason for this?
- There is nothing wrong with the programming here.
- If we used more weights in the critic, would that fix it?
- No. The problem is not because of poor function approximation - even if the HDP error was exactly zero this problem would still happen.
What are the problems with TD(\(\lambda\))?

- Actual behaviour of TD(\(\lambda\)) in a deterministic problem

- Is the actor sufficiently trained?

- This example was created with the greedy policy, so that is not relevant.
What are the problems with TD(\(\lambda\))?

- What about different values of \(\lambda\)?
- You get the same behaviour with \(\lambda = 0\) as \(\lambda = 1\): both show convergence to a sub-optimal trajectory.
- So this problem affects HDP as much as it does TD(\(\lambda\)).
The values $\tilde{J}_t = J'_t$ are perfectly learned for all $t$ along this trajectory, under a greedy policy, so why doesn’t Bellman’s Optimality Condition apply then?

Bellman’s Optimality Condition only applies when the critic is perfectly learned over the whole state space.

Here we’ve only explored one trajectory.

This negative result should not be surprising - it’s just the classic exploration versus exploitation dilemma.
Specifically, the problem is that the greedy policy is concerned with whether possible adjacent trajectories are better, not changes in $J$ along this particular trajectory.

TD($\lambda$) has perfectly learned the values of $J$ along this particular trajectory (and that’s not much use to the greedy policy).

This is the problem that DHP and value-gradient methods are designed to address efficiently.

Our other paper in this conference is dedicated to emphasising this point.\(^a\)

\(^a\)Fairbank and Alonso. *A Comparison of Learning Speed and Without Exploration between DHP and TD(0).* IJCNN 2012.
Considering the previous problem, we observe that value-gradients, \( \frac{\partial \tilde{J}}{\partial \tilde{x}} \), are what the greedy policy needs to decide whether an adjacent trajectory is better than the current one.

Hence DHP learns value-gradients, \( \frac{\partial \tilde{J}}{\partial \tilde{x}} \), instead of values, \( \tilde{J} \).

Similar reasons given by Werbos when GDHP was first defined.\(^3\)

Learning the value-gradients along a trajectory makes the trajectory bend itself into a locally optimal shape, automatically and without the need for local exploration, under a greedy policy.

What is DHP?

- DHP learns *value-gradients*, \( \frac{\partial \tilde{J}}{\partial \vec{x}} \), instead of values, \( \tilde{J} \)

HDP is
\[
\Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \vec{w}} \right)_t (J'_t - \tilde{J}_t)
\]
with \( J'_t \) defined by
\[
J'_t = U_t + \gamma \tilde{J}_{t+1}
\]

DHP is
\[
\Delta \vec{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \vec{w}} \right)_t \Omega_t \left( \left( \frac{\partial J'}{\partial \vec{x}} \right)_t - \left( \frac{\partial \tilde{J}}{\partial \vec{x}} \right)_t \right)
\]

Here the subscripted \( t \) means a quantity is evaluated at time step \( t \) of a trajectory, e.g. \( \left( \frac{\partial \tilde{J}}{\partial \vec{x}} \right)_t \equiv \left. \frac{\partial \tilde{J}}{\partial \vec{x}} \right|_{(\vec{x}_t, \vec{w})} \)

\( \Omega_t \) is a \( \text{dim}(\vec{x}) \times \text{dim}(\vec{x}) \) positive definite matrix. Included for generality here, but most DHP practitioners omit it. It was introduced by Werbos when GDHP was first defined.

In the above weight update, we treat matrix-vector product notation as defined in the paper - \( \left( \frac{\partial \tilde{J}}{\partial \vec{w}} \frac{\partial \tilde{J}}{\partial \vec{x}} \right) \) is a matrix with dimensions such that the matrix-matrix-vector product is a valid one!
What is DHP?

\[ \Delta \tilde{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \tilde{w}} \right)_t (J'_t - \tilde{J}_t) \]

with \( J'_t \) defined by

\[ J'_t = U_t + \gamma \tilde{J}_{t+1} \]

\[ \Delta \tilde{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \tilde{w}} \right)_t \Omega_t \left( \left( \frac{\partial J'}{\partial \bar{x}} \right)_t - \left( \frac{\partial \tilde{J}}{\partial \bar{x}} \right)_t \right) \]

- To differentiate \( \frac{\partial J'}{\partial \bar{x}} \), we need to define \( J' \) as a function of \( \bar{x} \):

\[ J'(\bar{x}, \bar{w}, \bar{z}) \equiv U(\bar{x}, A(\bar{x}, \bar{z})) + \gamma \tilde{J}(f(\bar{x}, A(\bar{x}, \bar{z})), \bar{w}) \]

- Fully differentiating this w.r.t. \( \bar{x} \) using the chain rule gives a recursion:

\[ \left( \frac{\partial J'}{\partial \bar{x}} \right)_t = \left( \frac{\partial U}{\partial \bar{x}} \right)_t + \left( \frac{\partial A}{\partial \bar{x}} \right)_t \left( \frac{\partial U}{\partial \bar{u}} \right)_t + \gamma \left( \left( \frac{\partial f}{\partial \bar{x}} \right)_t + \left( \frac{\partial A}{\partial \bar{x}} \right)_t \left( \frac{\partial f}{\partial \bar{u}} \right)_t \right) \left( \frac{\partial \tilde{J}}{\partial \bar{x}} \right)_{t+1} \]

\[ = \left( \frac{DU}{D\bar{x}} \right)_t + \gamma \left( \frac{Df}{D\bar{x}} \right)_t \left( \frac{\partial \tilde{J}}{\partial \bar{x}} \right)_{t+1} \]

where \( \frac{D}{D\bar{x}} \equiv \frac{\partial}{\partial \bar{x}} + \frac{\partial A}{\partial \bar{x}} \frac{\partial}{\partial \bar{u}} \)
What is DHP?

Defining $\tilde{G} \equiv \frac{\partial \tilde{J}}{\partial \tilde{x}}$ and $G' \equiv \frac{\partial J'}{\partial \tilde{x}}$ gives

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

with $G'_t$ defined by

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

and $\frac{D}{D\tilde{x}} \equiv \frac{\partial}{\partial \tilde{x}} + \frac{\partial A}{\partial \tilde{x}} \frac{\partial}{\partial \tilde{u}}$
Why learn value-gradients? Why use DHP? (2)

- Can be much faster
- Note: being model based is an issue, i.e. we need to know the functions $f(\vec{x}, \vec{u})$ and $U(\vec{x}, \vec{u})$ and their derivatives.
- ‘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.’
- See our other paper at this conference, *A Comparison of Learning Speed and Ability to Cope Without Exploration between DHP and TD(0).*

---

4 George G. Lendaris and James C. Neidhoefer, *Handbook of learning and ADP*, chapter 4, 2004
5 Paper 233
Where does VGL(\(\lambda\)) fit in?

**HDP is**
\[
\Delta \tilde{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \tilde{w}} \right)_t (J'_t - \tilde{J}_t)
\]
with \(J'_t\) defined by
\[
J'_t = U_t + \gamma \tilde{J}_{t+1}
\]

**DHP is**
\[
\Delta \tilde{w} = \alpha \sum_t \left( \frac{\partial \tilde{G}}{\partial \tilde{w}} \right)_t \Omega_t (G'_t - \tilde{G}_t)
\]
with \(G'_t\) defined by
\[
G'_t = \left( \frac{DU}{D\tilde{x}} \right)_t + \gamma \left( \frac{Df}{D\tilde{x}} \right)_t \tilde{G}_{t+1}
\]

**TD(\(\lambda\)) is**
\[
\Delta \tilde{w} = \alpha \sum_t \left( \frac{\partial \tilde{J}}{\partial \tilde{w}} \right)_t (J'_t - \tilde{J}_t)
\]
with \(J'_t\) defined by
\[
J'_t = U_t + \gamma (\lambda J'_{t+1} + (1-\lambda) \tilde{J}_{t+1})
\]

**VGL(\(\lambda\)) is**
\[
\Delta \tilde{w} = \alpha \sum_t \left( \frac{\partial \tilde{G}}{\partial \tilde{w}} \right)_t \Omega_t (G'_t - \tilde{G}_t)
\]
with \(G'_t\) defined by
\[
G'_t = \left( \frac{DU}{D\tilde{x}} \right)_t + \gamma \left( \frac{Df}{D\tilde{x}} \right)_t (\lambda G'_{t+1} + (1-\lambda) \tilde{G}_{t+1})
\]
The New Algorithm: VGL($\lambda$)

Why insert $\lambda$ into DHP, to get VGL($\lambda$)?

VGL($\lambda$) is

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

with $G'_t$ defined by

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

Introducing $\lambda$ into DHP to get VGL($\lambda$) yields the same advantages as it did for TD($\lambda$):

- More stable learning.
- Easier to prove convergence.
- Faster learning in long chains with delayed rewards.
- Good on finite horizon problems.
The New Algorithm: VGL(\(\lambda\))

What is VGL(\(\lambda\))?

Implementation of VGL(\(\lambda\))

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

with \(G'_t\) defined by

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

- There are two ways to implement it: in batch mode (where the above recursion is unrolled going backwards in time), and in on-line mode (using matrix “eligibility traces”).
- Both implementations are given in the paper.
- The batch mode implementation has a faster running time (details given in paper).
Why learn value-gradients? Why use DHP/VGL(λ)? (3)

- This diagram shows the value gradients, $\tilde{G}$, matching their targets, $G'_t$, well (the lines $\tilde{G}$ match the lines $G'$ closely in magnitude and direction).
- The trajectory (the blue curve) matches the optimal trajectory (the green curve)
- This is a consequence of Pontryagin’s Minimum Principle. It happens for any value-gradient learning algorithm, e.g. DHP and VGL(λ).
- A proof is available on author’s webpage.²

Key:
- blue line: actual trajectory
- green line: optimal trajectory
- magenta lines: $G''$
- cyan lines: $\tilde{G}$

Why learn value-gradients? Why use DHP/VGL(\(\lambda\))? (4)

- When the VGL algorithm makes progress in making the value-gradients, \(\tilde{G}_t\), approach their targets, \(G'_t\), for all \(t\), under a greedy policy, the trajectory will make progress in bending itself into a locally optimal shape.

- Java demo is available showing this bending happening.\(^a\)

- Hence, local exploration is automatic for VGL/DHP methods.

- This contrasts greatly to the TD(\(\lambda\)) example, which converged to a suboptimal trajectory in the absence of local exploration.

---

\(^a\)see author’s webpage
Example simple quadratic problem (taken from the paper)

- Notice both VGL algorithms work well without stochastic exploration, but the TD algorithms do not.
- Of course VGL(0) is identical to DHP, so the above graph shows the DHP result doing well (but slightly slower than VGL(1))

- Notice both VGL algorithms work > 1000 times faster than corresponding TD algorithms.
BPTT is a completely different method than using a critic:

- We want to minimise $J(\vec{x}_0, \vec{z})$ from any start state $\vec{x}_0$ w.r.t. $\vec{z}$.
- We could do gradient descent $\Delta \vec{z} = -\alpha \frac{\partial J}{\partial \vec{z}}$
- The back-propagation through time algorithm does this efficiently.
- Note, BPTT is here solving the full ADP problem (even though the ADP problem is an *unsupervised learning problem* in the sense that we are never shown the optimal actions we are looking for in advance).\(^6\)
- No critic involved!

---

\(^6\)Paul Werbos: “Backpropagation through time, what it is and how to do it,” 1990
Now for the surprising result, which was mentioned in an earlier slide (slide 3):
- BPTT is a critic-free method, so is fundamentally different from VGL? Right?
- Wrong!
An amazing fact ***

- If we use a greedy policy, and choose

\[
\Omega_t = \begin{cases} 
\left( \frac{\partial f}{\partial \bar{u}} \right)^T_t \left( \frac{\partial^2 \tilde{Q}}{\partial \bar{u} \partial \bar{u}} \right)^{-1}_{t-1} \left( \frac{\partial f}{\partial \bar{u}} \right)^{-1}_{t-1} & \text{for } t > 0 \\
0 & \text{for } t = 0
\end{cases}
\]

with \( \tilde{Q}(\bar{x}, \bar{u}, \bar{w}) = U(\bar{x}, \bar{u}) + \gamma \tilde{J}(f(\bar{x}, \bar{u}), \bar{w}) \) then:

- **VGL(1) becomes equivalent to BPTT**, i.e.:\(^7\)

\[
\sum_t \left( \frac{\partial \tilde{G}}{\partial \bar{w}} \right)_t \Omega_t \left( G'_t - \tilde{G}_t \right) \equiv - \frac{\partial J}{\partial \bar{w}}
\]

- Proof: see paper by Fairbank and Alonso\(^8\) or Fairbank 2008.

- This equivalence unifies BPTT (no critic) with our critic learning algorithm

---

\(^7\)Subject to \( \frac{\partial J}{\partial \bar{w}} \) existing.

The relationship of VGL(1) to Backpropagation Through Time

The equivalence of VGLΩ(1) to BPTT ***

- This provides a powerful convergence proof for VGLΩ(1) with a neural-network critic.
- Results from spacecraft problem in paper, with a greedy policy:

![Graphs showing learning progress](image)

- Notice how the final graph shows perfectly smooth learning progress, solving the on-going instability problem that exists for critic-learning algorithms.
Unlike other convergence proofs, this is valid for a non-linear function approximator for the critic (i.e. a neural network), and for the situation of the action network and the critic network both being updated concurrently.
Thank you!
Please email me for a copy of the slides or the paper to review at a slower pace.
michael.fairbank@virgin.net
Please chat with me after the presentation.