

# Value Gradient Learning

## VGL( $\lambda$ )

**Michael Fairbank**

Eduardo Alonso

# What we do

- Optimize control problems in a **large and continuous** state space using **model** functions of the environment
- Learn the **gradient of the value function**
- We use a **greedy policy**
- We consider **deterministic** environments
- Extend **DHP** with a **bootstrapping** parameter – like in TD( $\lambda$ )

# Outline

- Context
- From Bellman to Pontryagin
- VGL( $\lambda$ ): The algorithm
- Optimality
- Convergence
- Empirical results
- Summary and conclusions

# Adaptive Dynamic Programming

model

yes

(G)DHP

.

P

G

L

.

no

TD(0)=Q

... TD( $\lambda$ ) ...

TD(1)=MC

$\lambda=0$

$0 < \lambda < 1$

$\lambda=1$

boots

## Reinforcement Learning

# Adaptive Dynamic Programming

model

yes

(G)DHP    ... VGL( $\lambda$ ) ...    VGL(1)=BPTT

.

P  
G  
L

.

no

TD(0)=Q    ... TD( $\lambda$ ) ...    TD(1)=MC

$\lambda=0$

$0 < \lambda < 1$

$\lambda=1$

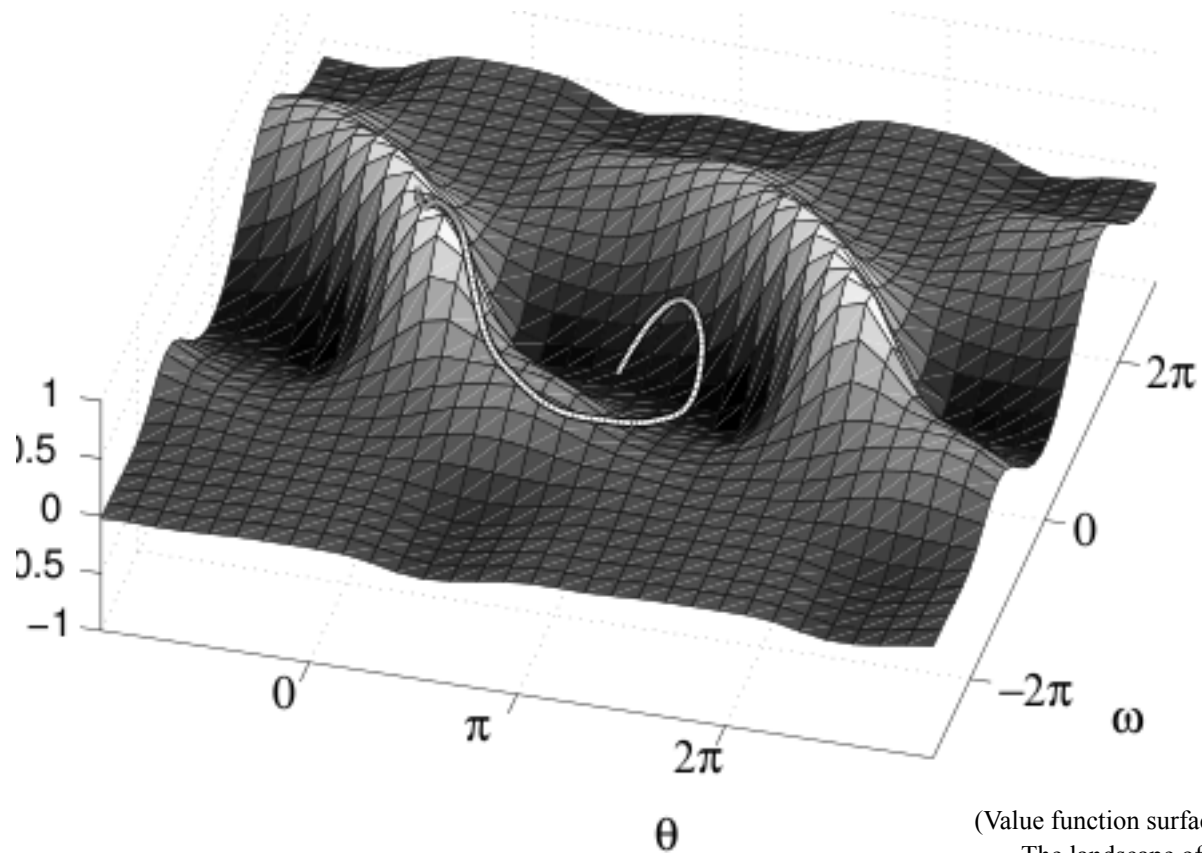
boots

## Reinforcement Learning

# What we mean by gradient-based

	What is being learned	Gradient used in weight update
TD	$V$	when $\lambda=1$ and policy fixed $\frac{\partial E}{\partial \vec{w}}$
VGL	$\frac{\partial V}{\partial \vec{x}}$	when $\lambda=1$ and policy greedy $\frac{\partial R}{\partial \vec{w}}$

# An Optimal Trajectory



(Value function surface taken from Doya 2000)

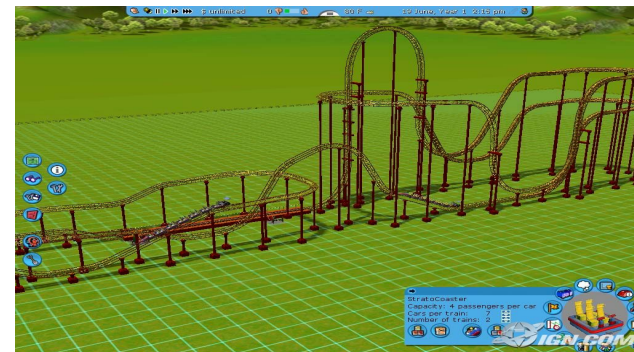
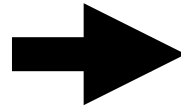
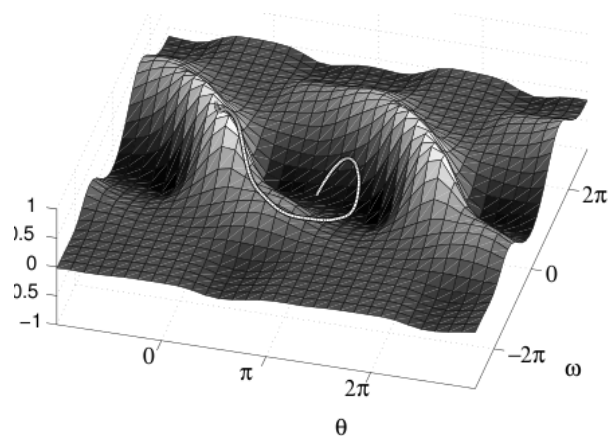
The landscape of the value function for the pendulum swing-up task. The white line shows an example of a swing-up trajectory. The state space was a cylinder. The axis labelled theta is the angle of the pendulum, and the axis labelled omega is the angular velocity

# Bellman's Optimality Principle

- To find this optimal trajectory, Bellman's Optimality Principle requires that we learn **the whole of the value function surface** correctly, and, at the same time, the trajectory is greedy with respect to that value function
- Was learning that whole surface **really necessary for just this one single trajectory?**
- **Can we make this process more efficient?**
- So **how little of the surface** could we really get away with learning?
- Pontryagin's maximum principle (PMP) answers this question

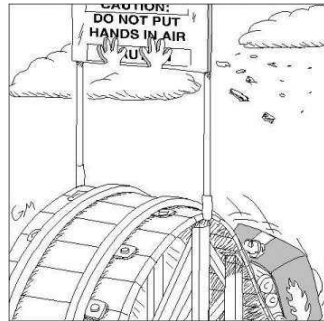
# A Visual Explanation of Pontryagin's maximum principle

- Can we **just** learn **the portion** of the value function surface that is **directly under the trajectory**?
- Instead of a whole surface this would leave us a rollercoaster shaped object



# A Visual Explanation of PMP

- This is **not** quite **enough**
- We need **shear** adding to the rollercoaster track. That then gives us **the least possible portion of the value function surface** in order to decide whether our trajectory is optimal or not. This is PMP!



This is a track with  
no shear



This is a track with  
shear

# Pontryagin's maximum principle for dummies

If we have the shear of the rollercoaster track "correct", and the rollercoaster track was found by a greedy policy, then that trajectory is locally extremal (and usually locally optimal)

# What do we mean by the shear of the track is “correct”

- Bellman's optimality principle requires that  $V = R^\lambda$  (Watkins'  $\lambda$ -return) everywhere in state space
- For the track shear to be correct we just require that  $\frac{\partial V}{\partial \vec{x}} = \frac{\partial R^\lambda}{\partial \vec{x}}$  all along the trajectory
- Note the difference: Bellman's must apply over all of state space. It is concerned with just the heights of the value function surface (heights of all of the possible rollercoaster tracks). PMP only requires something to be true over the single trajectory: the shear to be correct along one single rollercoaster track

# VGL( $\lambda$ ) is based on PMP

- Our learning method (“value-gradient learning”) is to just learn the track **shear** (i.e. the value-**gradient**)
- Since we have to learn a much lesser area of the value function VGL can increase efficiency and reduce the need for exploration
- Our paper defines a learning algorithm to do this, and extends PMP to work with a bootstrapping parameter, lambda, analogous to TD( $\lambda$ )
- When  $\lambda=0$ , our work becomes equivalent to Werbos’ algorithm Dual Heuristic Programming (DHP)

# The intimate relationship of the greedy policy to the value-gradient

- Using a first order Taylor series expansion of the greedy policy gives:

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

- Hence we see that the **greedy policy depends on the value-gradient, but *not* on the values themselves**
- This Taylor series approximation becomes better and better as the time step used in integrating the underlying continuous time system gets smaller. It becomes perfect in continuous time, and this is what Doya (2000)'s value-gradient policy uses

# Local Exploration Comes for Free, with VGL!

- Changing  $\partial\tilde{V}/\partial\vec{x}$  will immediately affect the greedy policy
- By moving  $\partial\tilde{V}/\partial\vec{x}$  towards its correct target we will steer the trajectory in the correct direction
- That's all we have to do: Learn the  $\partial V/\partial\vec{x}$  gradients and the trajectory will bend itself into the correct (locally optimal) shape

# In search for a good motto

- So “Exploration **versus** exploitation”
- Becomes
- “Exploration **and** exploitation”
- “for free ...”
- Just remember to be “greedy on the gradient”

# Comparison algorithm: TD( $\lambda$ )

- TD ( $\lambda$ ) learning is similar to MC but uses a modified target

$$\Delta \vec{w} = \alpha \sum \left( \frac{\partial \tilde{V}_t}{\partial \vec{w}} \right) (R^\lambda - \tilde{V}_t)$$

- $R^\lambda$  is Watkin's  $\lambda$ -Return
- $\lambda$  is a (global constant) blending parameter with  $0 \leq \lambda \leq 1$
- $\lambda = 0$  : full “bootstrapping” (the estimate  $\tilde{V}_t$  is being updated towards another estimate,  $\tilde{V}_{t+1}$ )
- $\lambda = 1$  : Monte Carlo (get same recurrence relationship as for  $R_t$ )
- TD( $\lambda$ ) is traditionally described using “eligibility traces” but we use the “forward view of TD( $\lambda$ )” here

# TD( $\lambda$ )

---

**Algorithm 2** TD( $\lambda$ ). This achieves  $\Delta \vec{w} = \alpha \sum_t \frac{\partial \tilde{V}}{\partial \vec{w}} (R^\lambda_t - \tilde{V}_t)$ .

---

```
1:  $\vec{e} \leftarrow 0$ 
2:  $t \leftarrow 0$ 
3: while not terminated( $\vec{x}_t$ ) do
4:    $\vec{a}_t \leftarrow \pi(\vec{x}_t, \vec{w})$ 
5:    $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{a}_t)$ 
6:    $\delta \leftarrow r(\vec{x}_t, \vec{a}_t) - \tilde{V}_t$ 
7:   if not terminated( $\vec{x}_{t+1}$ ) then
8:      $\delta \leftarrow \delta + \gamma \tilde{V}_{t+1}$ 
9:   end if
10:   $\vec{e} \leftarrow \lambda \gamma \vec{e} + \left( \frac{\partial \tilde{V}}{\partial \vec{w}} \right)_t$ 
11:   $\vec{w} \leftarrow \vec{w} + \alpha \vec{e} \delta$ 
12:   $t \leftarrow t + 1$ 
13: end while
```

---

# Our algorithm: VGL( $\lambda$ )

- We change this 
$$\Delta \vec{w} = \alpha \sum \left( \frac{\partial \tilde{V}_t}{\partial \vec{w}} \right) (R^\lambda - \tilde{V}_t)$$
- to this 
$$\Delta \vec{w} = \alpha \sum \left( \frac{\partial \tilde{G}_t}{\partial \vec{w}} \right) (G' - \tilde{G}_t)$$

where  $\tilde{G}_t = \frac{\partial \tilde{V}}{\partial \vec{x}}$  and  $G' = \frac{\partial R^\lambda}{\partial \vec{x}}$

- $\frac{\partial \tilde{V}}{\partial \vec{x}}$  can be found by ordinary backpropagation
- $\frac{\partial \tilde{G}}{\partial \vec{w}} = \frac{\partial^2 \tilde{V}}{\partial \vec{w} \partial \vec{x}}$  can be found by second order backpropagation
- $G' = \frac{\partial R^\lambda}{\partial \vec{x}}$  is found by full differentiation of the definition of  $R^\lambda$ , as detailed by Fairbank and Alonso 2011

# VGL( $\lambda$ )

---

**Algorithm 1** VGL( $\lambda$ ). This achieves  $\Delta\vec{w} = \alpha \sum_t \frac{\partial \tilde{G}}{\partial \vec{w}} \Omega_t (G'_t - \tilde{G}_t)$ .

---

1: $E \leftarrow 0$	8: $\vec{\delta} \leftarrow \vec{\delta} + \gamma \left( \frac{Df}{D\vec{x}} \right)_t \tilde{G}_{t+1}$
2: $t \leftarrow 0$	9: <b>end if</b>
3: <b>while</b> not terminated( $\vec{x}_t$ ) <b>do</b>	10: $E \leftarrow E + \left( \frac{\partial \tilde{G}}{\partial \vec{w}} \right)_t \Omega_t$
4: $\vec{a}_t \leftarrow \pi(\vec{x}_t, \vec{w})$	11: $\vec{w} \leftarrow \vec{w} + \alpha E \vec{\delta}$
5: $\vec{x}_{t+1} \leftarrow f(\vec{x}_t, \vec{a}_t)$	12: $E \leftarrow \lambda \gamma E \left( \frac{Df}{D\vec{x}} \right)_t$
6: $\vec{\delta} \leftarrow \left( \frac{Dr}{D\vec{x}} \right)_t - \tilde{G}_t$	13: $t \leftarrow t + 1$
7: <b>if</b> not terminated( $\vec{x}_{t+1}$ ) <b>then</b>	14: <b>end while</b>

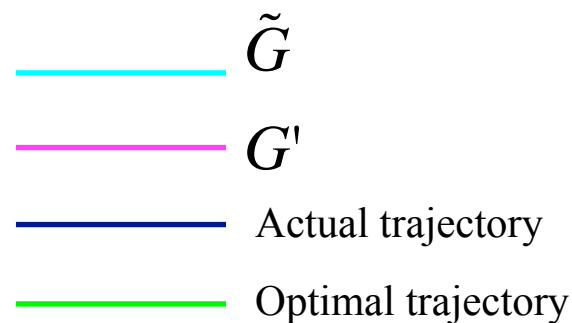
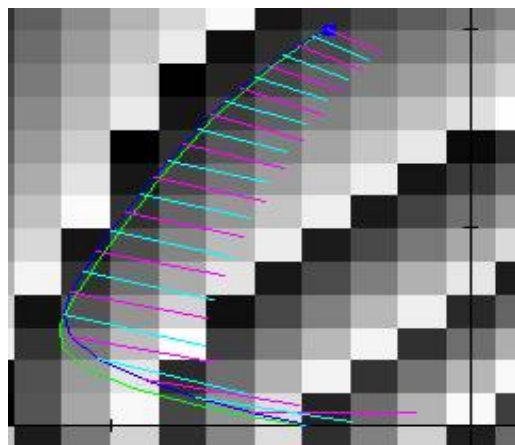
---

# VGL( $\lambda$ ) – algorithm notes

- This algorithm is derived in (Fairbank & Alonso, 2011)
- $E \in \mathfrak{R}^{\dim(\vec{w}) \times \dim(\vec{x})}$  is an “eligibility trace” workspace matrix
- Here we use a shorthand notation as  $\frac{D}{D\vec{x}} \equiv \frac{\partial}{\partial \vec{x}} + \frac{\partial \pi}{\partial \vec{x}} \frac{\partial}{\partial \vec{a}}$
- $\Omega_t \in \mathfrak{R}^{\dim(\vec{x}) \times \dim(\vec{x})}$  is an arbitrary positive definite matrix that the experimenter is free to choose, often just taken to be the identity matrix
- The running time is  $O(\dim(\vec{w})\dim(\vec{x}))$  per trajectory time-step
- This is the on-line implementation of the VGL( $\lambda$ ) algorithm. It is also possible to implement it in batch mode for episodic trajectories in a faster running time of  $O(\dim(\vec{w}))$  per trajectory time-step
- Jacobian notation: We use the transpose of what is usual
- Subscripted  $t$  indicate the timestep that a function is being evaluated at

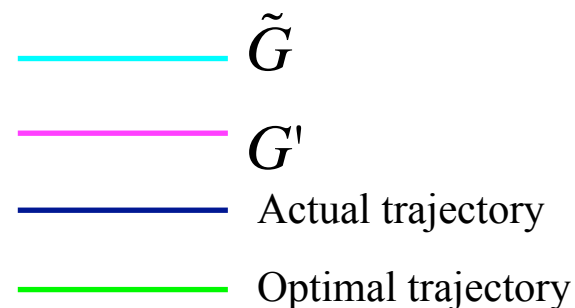
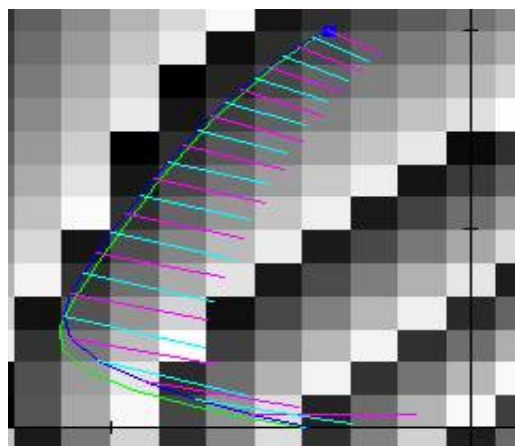
# Theoretical result: VGL Trajectory Optimality Condition

- This is our application of PMP:
- If the VGL weight update is at a fixed point at every time step along a trajectory generated by a greedy policy for any lambda (i.e. if the learning objective  $G' = \tilde{G}$  is met for all t along the trajectory), then that trajectory is locally extremal, and often locally optimal (Proof: section 3 of Fairbank & Alonso, 2011)



# Theoretical result: VGL Trajectory Optimality Condition

- Provided the VGL algorithm makes progress towards achieving  $G' = \tilde{G}$  all along a greedy trajectory, then provided the trajectory remains greedy, it will make progress in **bending itself** towards a locally optimal shape, and this will happen without the need for any stochastic exploration



# Theoretical result: VGL convergence

- In the weight update  $\Delta \vec{w} = \alpha \sum \left( \frac{\partial \tilde{G}_t}{\partial \vec{w}} \right) (G' - \tilde{G}_t)$

$G'$  is a moving target, so why should it converge?

- It is proven (section 4 of Fairbank & Alonso, 2011) that if you modify the weight update to

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

where  $\Omega_t$  is a specially chosen symmetric positive semi-definite matrix, and use  $\lambda = 1$ , then  $\Delta \vec{w} = \alpha \left( \frac{\partial R}{\partial \vec{w}} \right)$ , so it is equivalent to PGL and to BPTT

# Theoretical result: VGL convergence

- Therefore in this case it is gradient ascent on  $R$ , a function that is bound above, so the algorithm must converge in this case (subject to the error surface for  $R$  being smooth with respect to  $w$ , and subject to the greedy policy being differentiable (e.g. like Doya's efficient value-gradient greedy policy))
- This is a convergence proof for a value-function learning algorithm, using a **general smooth function approximator, with a greedy policy, but only for  $\lambda=1$**
- This is a step forward, because with a greedy policy,  $TD(\lambda)$  can diverge when  $\lambda=1$  (we have found simple divergence examples)

# Empirical results

- Experiments are given in (Fairbank, 2008)
- For the sake of simplicity, let's present a very simple quadratic optimisation control problem:
- The robot only takes one action,  $a_0$ , in a trajectory starting from state  $x_0$
- The total reward for that trajectory is  $-(x_0 + a_0)^2$
- So the optimal action to choose is  $a_0 = -x_0$  before the trajectory terminates
- When starting from  $x_0=0$ , the robot simply has to learn to choose a zero action

# Details of this experiment

Model functions used:

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

The trajectory terminates as soon as  $t = 2$  is reached. We make robot always start from  $x_0 = 0$ .

Function approximator used:

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

Greedy policy (with added noise)  
(solving exactly):

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

where  $X_\sigma$  is a normally distributed random variable with mean zero and standard deviation  $\sigma$ , that we are adding to the greedy policy to force it to explore

Optimal weights:

$$w_1^* = 0$$

Starting from randomised weights,  
stopping condition is:

$$|w_1 - w_1^*| < 10^{-7}$$

Policy noise ( $\sigma$ )	$\alpha = 0.01$			$\alpha = 0.1$			$\alpha = 1.0$		
	Success rate	Iterations		Success rate	Iterations		Success rate	Iterations	
		(Mean)	(s.d.)		(Mean)	(s.d.)		(Mean)	(s.d.)
<b>Results for algorithm VL(<math>\lambda</math>)</b>									
10	66.4%	1075.1	293.35	0.0%			0.0%		
1	<b>100.0%</b>	<b>1715.8</b>	343.31	87.6%	163.52	31.948	3.8%	134.86	59.643
0.1	100.0%	172445	31007	89.5%	17160	3033.6	16.5%	1527.6	118.39
0	<b>0.0%</b>			<b>0.0%</b>			<b>0.0%</b>		
<b>Results for algorithm VGL(<math>\lambda</math>)</b>									
<b>0</b>	100.0%	1728.2	112.05	100.0%	166.15	11.481	<b>100.0%</b>	<b>1</b>	<b>0</b>

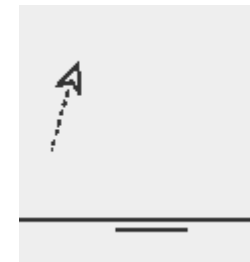
# Conclusions of the Previous Experiment

These concur succinctly with our motivations for VGL:

1. Without exploration value-learning methods fail
2. VGL is potentially more efficient than value-learning methods (in this experiment by a factor of  $\sim 1700$ )

# Does it scale up to larger problems?

- Our work: More experiments given in (Fairbank, 2008)
- Full neural network control problems with “lunar lander” spacecraft



- Others' work: DHP has successful applications in industrial control, e.g:
- “Dual Heuristic Programming Excitation Neurocontrol for Generators in a Multimachine Power System” (Venayagamoorthy and Wunsch, 2003).
- See “ADP: An introduction” (Wang, Zhange and Liu, 2009) for more applications of DHP

# Suitable domains

- The problem works well where the model functions are smooth, almost everywhere
- For example, good problems for our method are control system, robot navigation, continuous reward functions
- But it would **not** be **good** for some traditional RL environments, e.g. Backgammon, Step function reward functions, because these are **discrete / discontinuous** almost everywhere

Value Gradient Learning	Value Learning (TD( $\lambda$ ))	MSPBE (GQ-learning)
$\Delta \vec{w} = \alpha \sum \left( \frac{\partial \tilde{G}_t}{\partial \vec{w}} \right) \Omega_t (G^* - \tilde{G}_t)$	$\Delta \vec{w} = \alpha \sum \left( \frac{\partial \tilde{V}_t}{\partial \vec{w}} \right) (R^\lambda - \tilde{V}_t)$	$\Delta w = -\alpha \frac{\partial E}{\partial \vec{w}}$
Requires $G^* = G$ for all t along a <b>single</b> trajectory for local optimality	Requires $V^* = V$ over <b>all neighboring</b> trajectories for local optimality	Same as VL
Optimality proof relies on <b>Pontryagin's</b> maximum principle	Optimality proof requires <b>Bellman's</b> Optimality principle	Same as VL
Local <b>exploration</b> comes from <b>free</b>	Local exploration does <b>not</b> come from free	Same as VL
Works with <b>deterministic</b> systems <b>and stochastic</b> systems	Works with <b>stochastic</b> exploration <b>only</b>	Same as VL
Requires knowledge of the <b>model</b> functions	Does <b>not</b> require knowledge of the <b>model</b> functions	Same as VL
Has convergence proof for a <b>greedy policy and general function approximator</b> to represent the value function, <b>when <math>\lambda = 1</math></b>	Has <b>no</b> convergence proof for a greedy policy or a general function approximator to represent the value function	Has convergence proof for a <b>fixed policy and a general function approximator for <math>0 \leq \lambda \leq 1</math></b>

# Conclusions

- We have introduced a new learning algorithm based on value gradients, proved its optimality and convergence –and its equivalence to PGL and BPTT
- Further work: **Investigate VGL-MSPBE relations**
  - MSPBE may help VGL find a proof for  $\lambda < 1$
  - VGL may help MSPBE with greedy policies

# References

- Detailed proofs, Fairbank & Alonso 2011, *The local optimality of value-gradient learning, and its relationship to Policy Gradient Learning*, <http://arxiv.org/abs/1101.0428>
- Empirical results, Fairbank 2008, *Reinforcement Learning by Value Gradients*, <http://arxiv.org/pdf/0803.3539>
- Demos at <http://vega.soi.city.ac.uk/~abdy934/>
- Questions, [michael.fairbank.1@city.ac.uk](mailto:michael.fairbank.1@city.ac.uk)