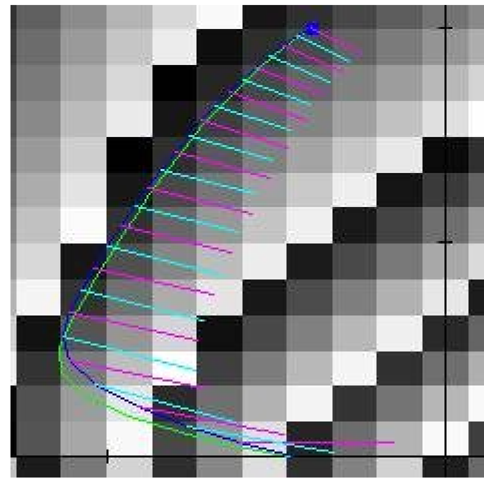
Reinforcement Learning by Value Gradients



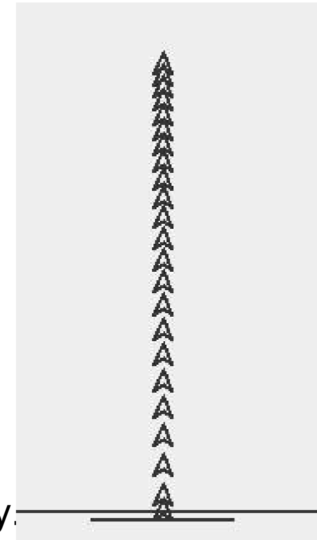
Michael Fairbank

November 18th 2010

michael.fairbank.1@city.ac.uk

Reinforcement Learning Problem and Notation

For any trajectory, at each time step t:
- State vector x_t
- Action vector a_t
- Instantaneous reward r_t



For the physics of the environment there is a

Model function f: $x_{t+1}=f(x_t, a_t)$

And a reward function r: $r_t= r(x_t, a_t)$

State space is Markovian: x_t determines everything you need to know to fly.

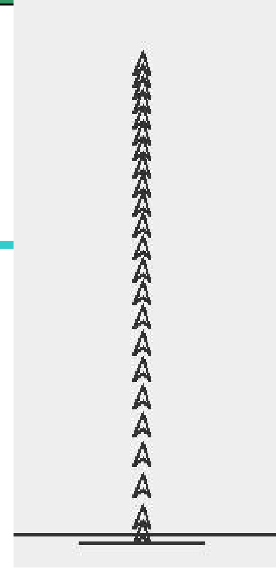
i.e. no need to remember the past.

The RL problem is to figure out what actions maximise the total reward $R=\Sigma r_t$

Specifically we want a policy function $\pi(x,w)$ such that always choosing $a_t=\pi(x_t,w)$ produces optimal behaviour (i.e. always maximises R)

The motivation for RL is that all useful problems that require intelligence to solve could be stated as trying to maximise some reward function R. E.g. life's reward function in the long term is presumably reproductive success. In shorter term it might be getting the next meal.

RL example problem: 1D Lunar Lander



Total reward, R=-(K_1*final_velocity^2+K_2*fuel_consumed)

x_t= {height, vertical velocity, fuel consumed}

a_t= {thrust}

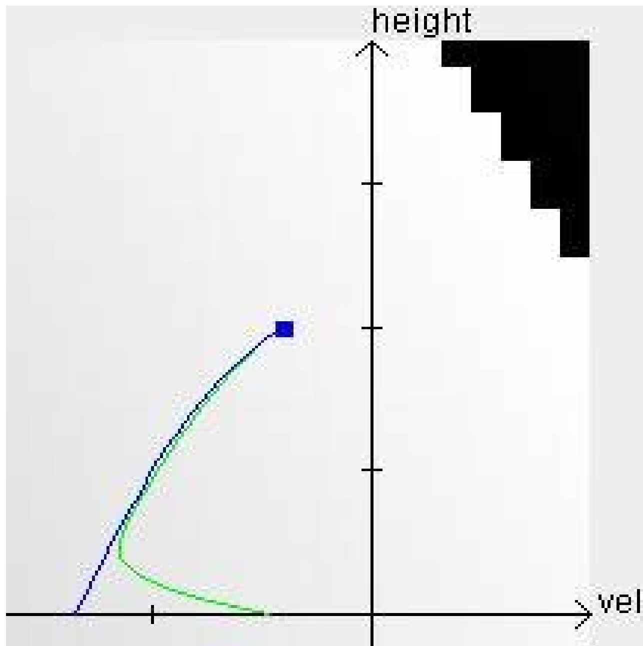
$$x_{t+1} = f(x_t, a_t) \quad \text{Such that} \quad \begin{cases} \text{vertical velocity} \leftarrow \text{vertical velocity} + (\text{-g} + a_t)\Delta t \\ \text{height} \leftarrow \text{height} + (\text{vertical velocity})\Delta t \\ \text{fuel consumed} \leftarrow \text{fuel consumed} + a_t \Delta t \end{cases}$$

$$r(x_t, a_t) = -K_2 * a_t \Delta t + \begin{cases} -K_1(\text{vertical velocity})^2 & \text{if new state is terminal} \\ 0 & \text{otherwise} \end{cases}$$

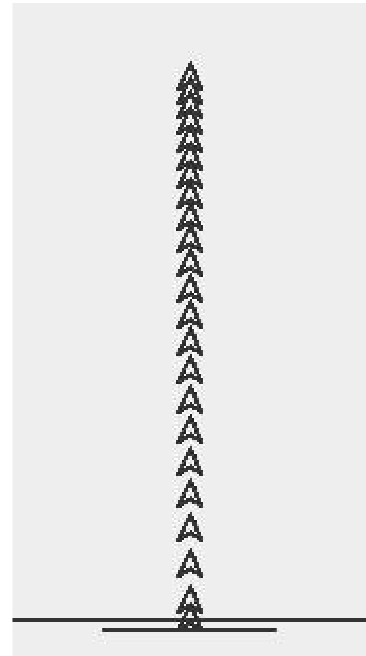
- Actions sometimes good, sometimes bad
- Functions are smooth

State-space diagrams show entire trajectories (spacecraft plummeting like a stone)

State-space shows height on the y-axis and velocity on the x-axis.

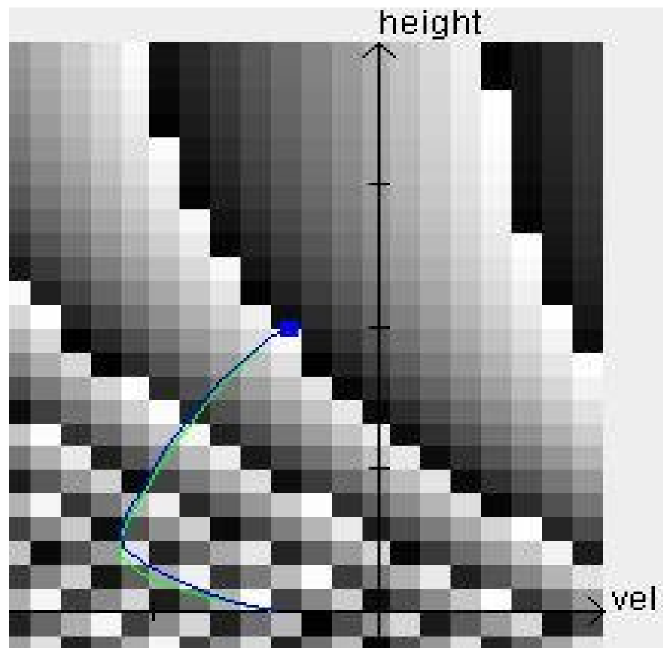


State-space view of a flight. Blue line shows flight.

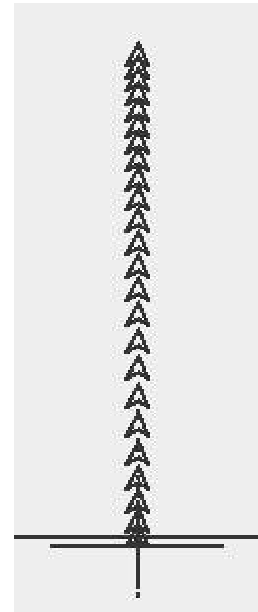


The corresponding actual flight: Entire animation is summarised by a single "long exposure photograph".

State-space diagrams (second example – braking as late as possible is optimal)

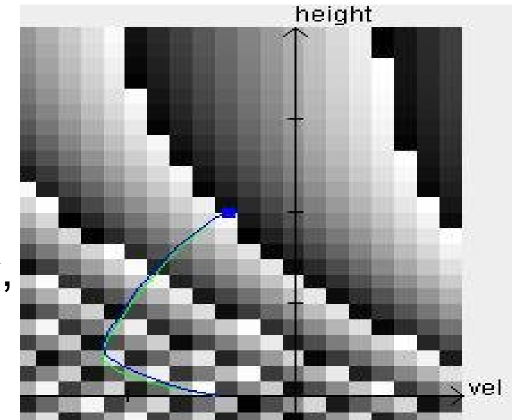


State-space view. Blue line shows flight. Green is theoretical optimal behaviour.



The corresponding actual flight: braking nicely to slow down as it approaches ground.

Optimal trajectories for 1DLL problem



- The green curve shows the theoretical optimal trajectory, calculated using Pontryagin's maximum principle. This calculation is detailed in the appendix of Fairbank 2008.
- The optimal trajectory is useful to judge learning algorithms by.
- It is quite a tricky problem to find the optimal trajectory - Pontryagin's maximum principle is the most efficient method to solve a problem of this kind AFAIK.
- Notice the solution doesn't land at zero velocity- that would be wasteful of fuel.
- Notice the solution is to brake as late as possible – it would be wasteful of fuel to hover down slowly at a constant speed.

Making a neural network solve a RL problem directly ("policy gradient learning")

One method to solve a RL problem is Policy Gradient Learning (PGL):

- Neural network takes in input vector x_t and produces output vector a_t.
- Thus neural network is the function approximator for the policy function $\pi(x_t,w)$
- Train neural network by gradient ascent on R: $\Delta w=\alpha(dR/dw)$
- dR/dw can be found by "backpropagation through time" (BPTT)
- This works fine, but is not the method used in most RL literature. They prefer to use a "value function", which is the method we will consider from now on:

Using a value function (The optimal value function)

- Assign a score to every point of state space *V*(x)* that equals the best possible total reward that could be achieved if starting from *x* and flying optimally until termination.
- This is called the optimal value function.
- If this was perfectly known then optimal policy would be easy: at any instant consider all possible actions available and always choose the one that leads to the best valued state as rated by V*, whilst also taking into account the immediate short-term reward in getting there.
- This policy is called the greedy policy on V*:
  - $\pi(x_t,w)=\text{argmax_a} \; [r(x_t,a)+V^*(f(x_t,a))]$

Using a neural network to represent the value function

- Represent our approximation to $V^*(x)$ by the neural network output $\tilde{V}(x,w)$, with weight vector w.
- Fly spacecraft by following greedy policy on $\tilde{V}$
- Initially w is random, so greedy policy produces inefficient behaviour.
- Learning is meant to do weight updates so as to gradually move $\tilde{V}$ towards V^*.
- Bellman's Optimality Principle proves that if for every x, if the total reward encountered ($R(x)$) while following the greedy policy on $\tilde{V}$ is such that $R(x)=\tilde{V}(x,w)$, then $\tilde{V}(x,w)\equiv V^*(x)$ and hence the greedy policy on $\tilde{V}$ is optimal.

Monte Carlo learning

Update the weight vector, *w*, with Monte Carlo backups:

- For each trajectory encountered while following the greedy policy on $\tilde{V}$, evaluate the total reward $R(x_t)$ for each x_t along the trajectory, and make $\tilde{V}(x,w)$ move towards these R values for each x_t with a neural network weight update:

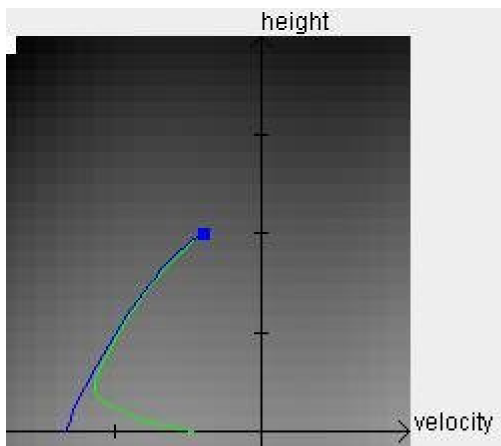
  i.e. For each timestep t of each trajectory, use:

  $\Delta w = \alpha \Sigma (d\tilde{V}(x_t,w)/dw)(R(x_t)-\tilde{V}(x_t,w))$

- This gradient is found by ordinary neural network backpropagation, but the error signal is $\delta = R(x_t) - \tilde{V}(x_t,w)$, which is analogous to the usual δ=(target-actual)
- Problem: Unlike ordinary NN learning, convergence is not assured, because $R(x_t)$ depends on *w* (since R depends on the trajectory, and that depends on the greedy policy, which in turn depends on w). $R(x_t)$ is a moving target! Weight update is not proven to converge.
- Advantage: Compared to policy gradient learning, we learn something interesting (the values) about every point on the trajectory.

Monte Carlo Learning in Practice

- Does it work? See demo.
- Add exploration. Does it work? See demo.
- In demo, grey-scale indicates the value-function. The whiter the “better”.
- (But in the demo, the colours just “wrap around” when the whiteness exceeds RGB (255,255,255)!)

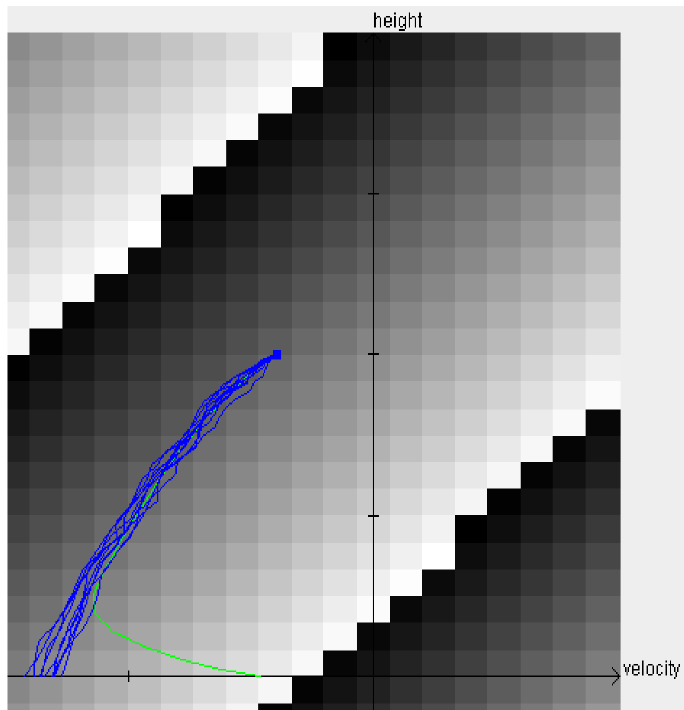


ErrorMeasures (averaged over all trajectories)

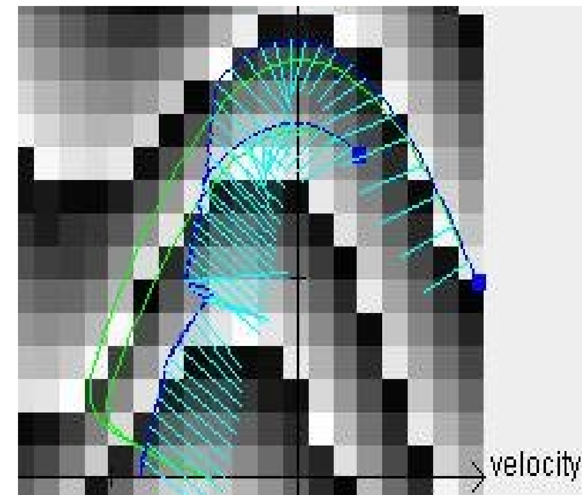
Total reward	-46.5941684766864
VL error (=Sum (V'_t-V_t)^2)	7.379849561915806E-5

Monte Carlo learning with exploration

With some randomness added to greedy policy – does not solve it:



With “exploring starts” - an improvement



TD (λ)

Remember, Monte Carlo learning was $\Delta w=\alpha\Sigma(d\tilde{V}_t/dw)(R_t-\tilde{V}_t)$

The "target" for this neural network update is R_t

Notice here $R_t=\Sigma_{k\geq t} r_k$ satisfies the recurrence relation $R_t=r_t+R_{t+1}$

TD (λ) learning is similar but uses a modified target:

$\Delta w=\alpha\Sigma(d\tilde{V}_t/dw)(V'_t-\tilde{V}_t)$, where $V'_t=r_t+\lambda V'_{t+1}+(1-\lambda)\tilde{V}_{t+1}$

λ is a (global constant) blending parameter with $0\leq\lambda\leq 1$

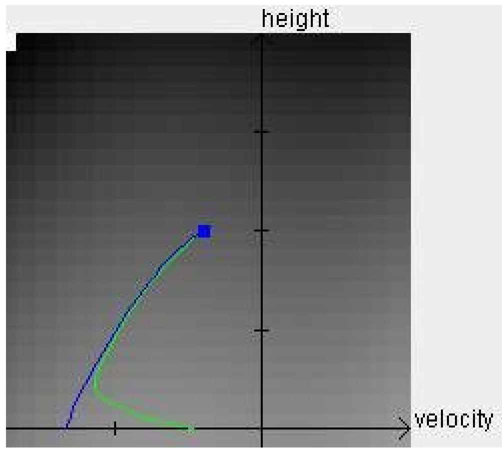
$\lambda=0$:full "bootstrapping" (the estimate $\tilde{V}_t$ is being updated towards another estimate, $\tilde{V}_{t+1}$)

$\lambda=1$:Monte Carlo (get same recurrence relationship as for R_t)

TD (λ) - issues

- The above slide is an extremely concise definition of the TD (λ) algorithm: it is traditionally described using "eligibility traces" (Sutton 1988). The two are mathematically equivalent (proof in appendix A of Fairbank/Alonso 2010).
- TD(λ) is potentially faster because the estimate $\tilde{V}_{t+1}$ is potentially "better" than R_{t+1}.
- TD(λ) is potentially slower because the estimate $\tilde{V}_{t+1}$ is potentially "worse" than R_{t+1}.
- In my experience, TD(λ) is even less stable at converging than Monte Carlo learning because of positive feedback from updating estimates towards estimates.
- In my experience, TD(λ) is pretty useless for deterministic environments like this. Its real value may be for stochastic systems, since $\tilde{V}_{t+1}$ may have lower variance than R_{t+1}.
- It does not address the reason why the above Monte Carlo experiment isn't working.
- It is not proven to converge. (However there is one notable partial convergence result: Tsitsiklis and Van Roy, 1996, proved it will converge if the function approximator is linear in w, i.e. not a neural network, and if the policy is fixed, i.e. not a greedy policy. Neither of these simplifications is relevant here.)

Fixing the problem



ErrorMeasures (averaged over all trajectories)

Total reward	-46.5941684766864
VL error (=Sum (V'_t-V_t)^2)	7.379849561915806E-5

This picture shows the spacecraft plummeting like a stone, even though the approximated value function equals its target value almost everywhere (to within a total sum of squared error 7e-5).

- The greedy policy depends upon the relative values of $\tilde{V}$.
- Hence the greedy policy is only concerned with the gradient of $\tilde{V}$, not its magnitude.
- In the above, the weight updates for Monte Carlo learning / TD(lambda) do not do anything, since the target values are already met. But they haven't learned what is necessary to fly well: they have not learned the gradient. They learn the magnitude of $\tilde{V}$, not its gradient.
- The solution (surely?) is to learn the gradient!

Value Gradient Learning

We change this:

$$\Delta w=\alpha\Sigma(d\tilde{V}_t/dw)(V'_t-\tilde{V}_t), \text{ where } V'=r_t+\lambda V'_{t+1}+(1-\lambda)\tilde{V}_{t+1}$$

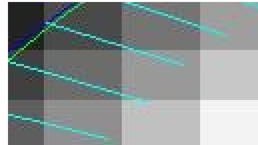
to this:

$$\Delta w=\alpha\Sigma(d\tilde{G}_t/dw)(G'_t-\tilde{G}_t), \text{ where } \tilde{G}=d\tilde{V}/dx \text{ and } G'=dV'/dx$$

- Notes:
  $d\tilde{V}/dx$ can be found by ordinary backpropagation.
  $d\tilde{G}/dw=d^2\tilde{V}/dwdx$ can be found by second order backpropagation.
  $G'=dV'/dx$ is found by full differentiation of the definition of V', giving:
  $G'=dr/dx+(df/dx)(\lambda G'_{t+1}+(1-\lambda)\tilde{G}_{t+1})$ and where here d/dx means full differentiation through the policy function by the chain rule (since $a_t=\pi(x_t,w)$, $G'_{t+1}=G'_{t+1}(x_{t+1},w)$, $\tilde{G}_{t+1}=\tilde{G}_{t+1}(x_{t+1},w)$, $r_t=r(x_t,a_t)$ and $x_{t+1}=f(x_t,a_t)$; see Eq. 7 of Fairbank/Alonso 2010 for all of the details fully worked out).
- With $\lambda=0$, this algorithm is equivalent to “Dual Heuristic Programming” (DHP) by Werbos (Handbook of Intelligent Control, eds. White and Sofge, 1992). Our contribution extends it to $0\leq\lambda\leq1$ and adds two nice theoretical results.

Getting familiar with value-gradients

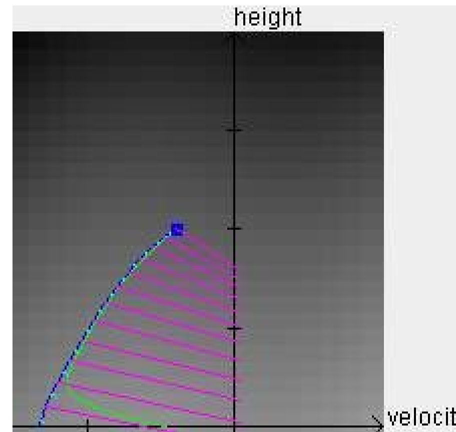
- We defined G=dV/dx. This is the gradient of the value function (the grey-scale) with respect to the state vector.
- This is a vector indicating which direction the value function increases in the most (in the grey-scale this means the direction of maximum increasing whiteness)



- The demo has a mode in which the value gradient can be viewed more easily by the drawing of cyan coloured lines.

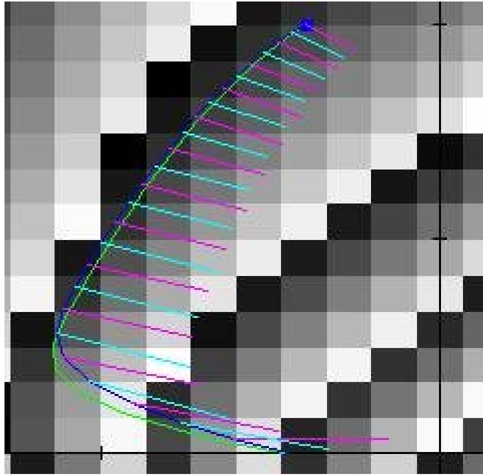
The target value gradient: G'

- We defined G'=dV'/dx. This is the “target value gradient”.
- Like in the previous slide, the target value gradient, G', also can be visualised. V' is a function of (x,w), so it could be plotted as a grey-scale all over state-space, and so gradients of maximum increasing whiteness could be imagined.
- The demo doesn't show a grey-scale for V', but it does show the target gradients G' with magenta lines.
- In the above example which showed Monte-Carlo learning stuck, here is what G' and $\tilde{G}$ looked like:



- If the learning algorithm had managed to make the cyan lines point towards the pink ones then the greedy policy would also the curve bend in the correct direction.

Value-Gradients Optimality Principle



- In this picture cyan lines ($\tilde{G}$) roughly match the magenta lines (G') in orientation and length, and therefore...
- The trajectory is roughly optimal.

- If G'_t-$\tilde{G}_t$ for all t then the trajectory will be locally extremal, and often locally optimal. Proof: Fairbank/Alonso 2010, section 3.
- The target G' is dependent on the trajectory and hence w, so it is a moving target.
- However all we need to do is concentrate on achieving G'_t-$\tilde{G}_t$ for all t, and the trajectory will bend itself into a locally optimal shape.

Value gradient learning in practice

- See demos.
- Scales up to 2D problem with spaceship rotations.

A Convergence guarantee:

- In the weight update $\Delta w=\alpha\Sigma(d\tilde{G}_t/dw)(G'_t-\tilde{G}_t)$, G' is a moving target, so why should it converge?
- It is proven in the paper (section 4 of Fairbank/Alonso 2010) that if you modify the weight update to $\Delta w=\alpha\Sigma(d\tilde{G}_t/dw)\Omega_t(G'_t-\tilde{G}_t)$ where Ω_t is a specially chosen symmetric positive semi-definite matrix, and use $\lambda=1$, then $\Delta w=\alpha(dR/dw)$, so it is equivalent to PGL!
- 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).
- Interestingly, Werbos included a matrix Ω_t in his variant algorithm “Globalized Dual Heuristic Programming”, but didn't have a recipe to choose it, and only uses $\lambda=0$.

Future goals

- Try to find a similar convergence result for $\lambda<1$
- Try to see if the proven convergence result can be applied to a simple variant of Monte-Carlo learning (or TD(λ)). Is Monte Carlo learning a rough stochastic approximation to PGL? Or can it be modified to be so?

Criticisms of value-gradient learning

- It requires knowledge of the model functions (e.g. for df/da and df/dx and dr/dx etc).

Industrial applications of Dual Heuristic Programming

- Many examples: google "dual heuristic programming" returns 1500 records.
- Strangely it is not mentioned in main RL literature – largely unknown? Or rejected due to requirement for model function knowledge?

A comparison between value learning and value gradient learning

VGL	VL
$\Delta w=\alpha\Sigma(d\tilde{G}_t/dw)\Omega_t(G'_t-\tilde{G}_t)$	$\Delta w=\alpha\Sigma(d\tilde{V}_t/dw)(V'_t-\tilde{V}_t)$
Requires G'=G for all t along a single trajectory for local optimality	Requires V'=V over all neighboring trajectories for local optimality
Optimality proof relies on Pontryagin's maximum principle	Optimality proof requires Bellman's Optimality Principle
Local exploration comes for free	Local exploration does not come for free
Works with deterministic systems and stochastic systems	Works with stochastic exploration only.
Requires knowledge of the model functions	Does not require knowledge of the model functions (when using the variant algorithm, "Q-learning").
Has convergence proof for a greedy policy and general function approximator to represent the value function, when $\lambda=1$	Has no convergence proof for a greedy policy or a general function approximator to represent the value function.

References

- Fairbank/Alonso 2011: The local optimality of value-gradient learning, and its relationship to Policy Gradient Learning. http://arxiv.org/abs/1101.0428
- Fairbank 2008. Reinforcement learning by value gradients. http://arxiv.org/abs/0803.3539