Reinforcement Learning with a value function

1. formalising the problem
2. recap: the Bellman equation and Value Iteration
3. examples of reinforcement learning
4. learning the value function from experience

reinforcement learning

Unsupervised learners look at patterns and try to represent them in better ways.

Supervised learners get told what they should do, and try to do it.

Reinforcement learners get “rewards”, but not targets:

- agent generates actions, in response to the current state
- the new state leads to a “reinforcement” signal (+/-)
- unlike supervised learning, there no direct guidance about *desirable* actions - all one can do is try things out and try to learn from the consequences.

The learning task is to adapt subsequent behaviour, so that successful actions are made more likely next time.

Simplest example: two-armed bandit problem.

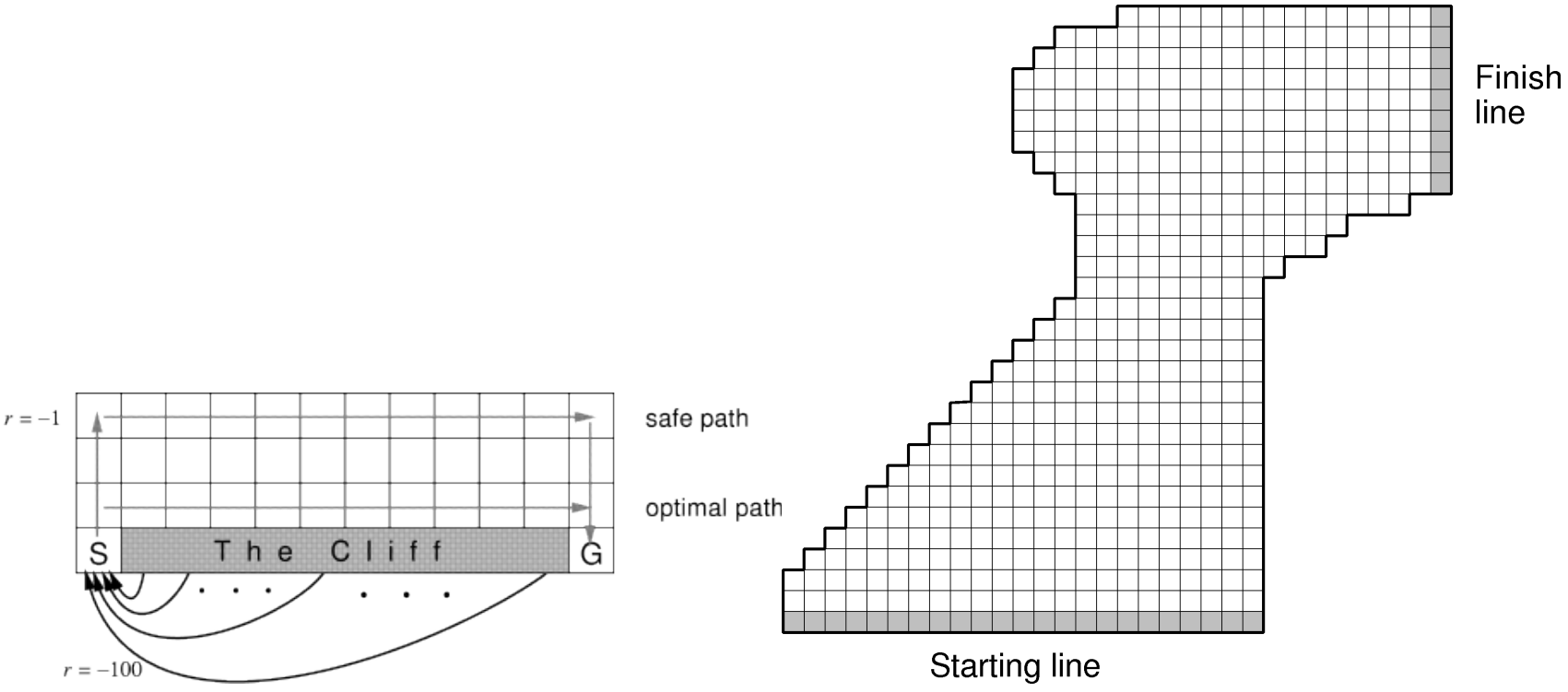
what he said...

Rich Sutton “wrote the book” on Reinforcement Learning:

> *“I am seeking to identify general computational principles underlying what we mean by intelligence and goal-directed behavior. I start with the interaction between the intelligent agent and its environment. Goals, choices, and sources of information are all defined in terms of this interaction. In some sense it is the only thing that is real, and from it all our sense of the world is created. How is this done? How can interaction lead to better behavior, better perception, better models of the world? What are the computational issues in doing this efficiently and in realtime? These are the sort of questions that I ask in trying to understand what it means to be intelligent, to predict and influence the world, to learn, perceive, act, and think.”*

http://www.incompleteideas.net/sutton/book/the-book.html

toy problems: cliffs and race-tracks



You can make the racetrack more interesting by giving the car momentum.

terms and notation

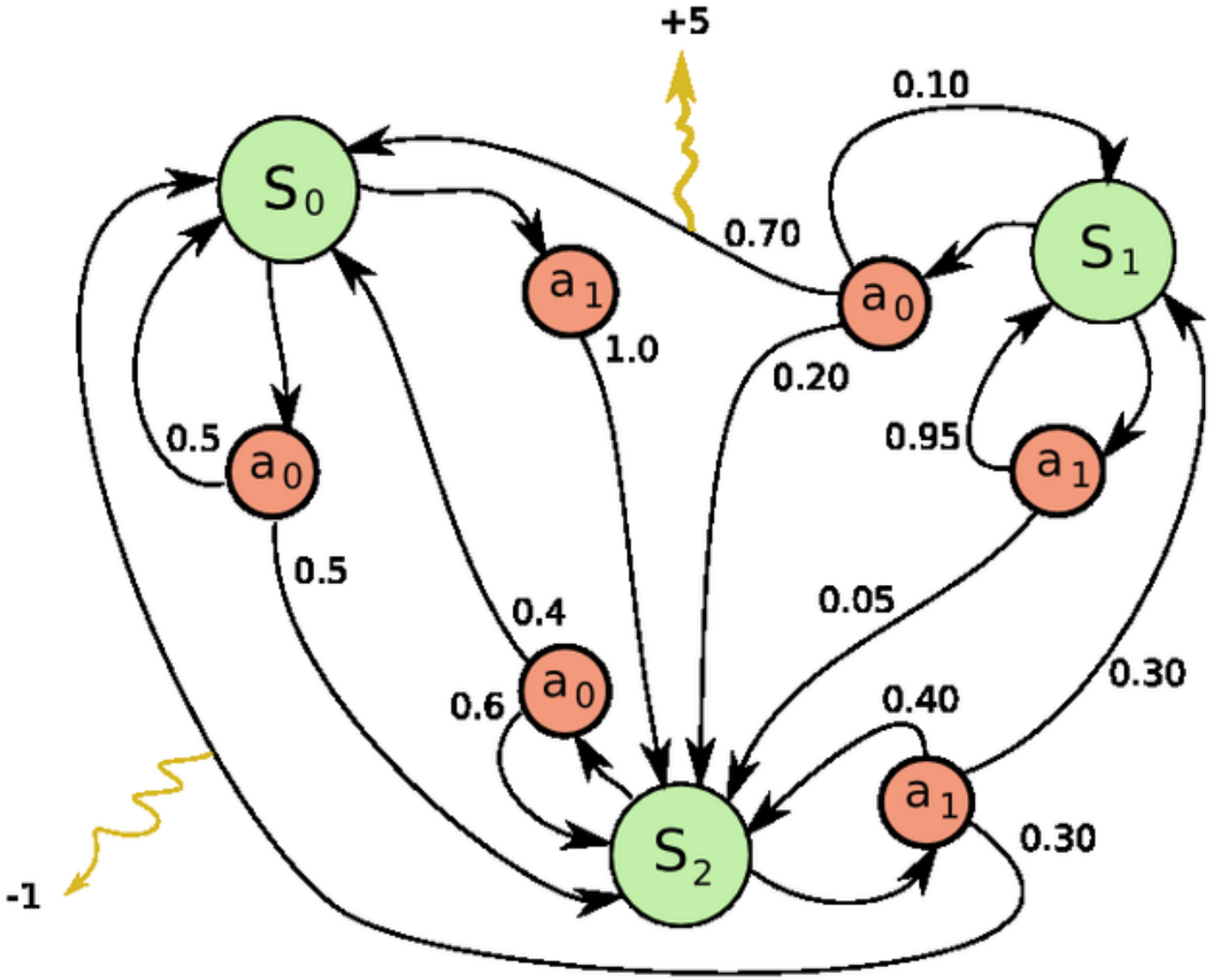
- $\mathbf{s} \in \{S\}$ is state of the world.
- $a \in \{A\}$ is an action carried out by the agent.
- $P_{\mathbf{s}'|\mathbf{s},a}$ is probability that a in $\mathbf{s}$ takes agent to $\mathbf{s}'$
- r_s is a scalar reward obtained in s. Could be stochastic.
- $R_{\mathbf{s}}$ is $\mathbb{E}[r_s]$
- $\pi_{a|s}$ is the policy, eg. a look-up table (if s, a discrete).

MDP *Markov Decision Process*:
the agent has complete knowledge of state.

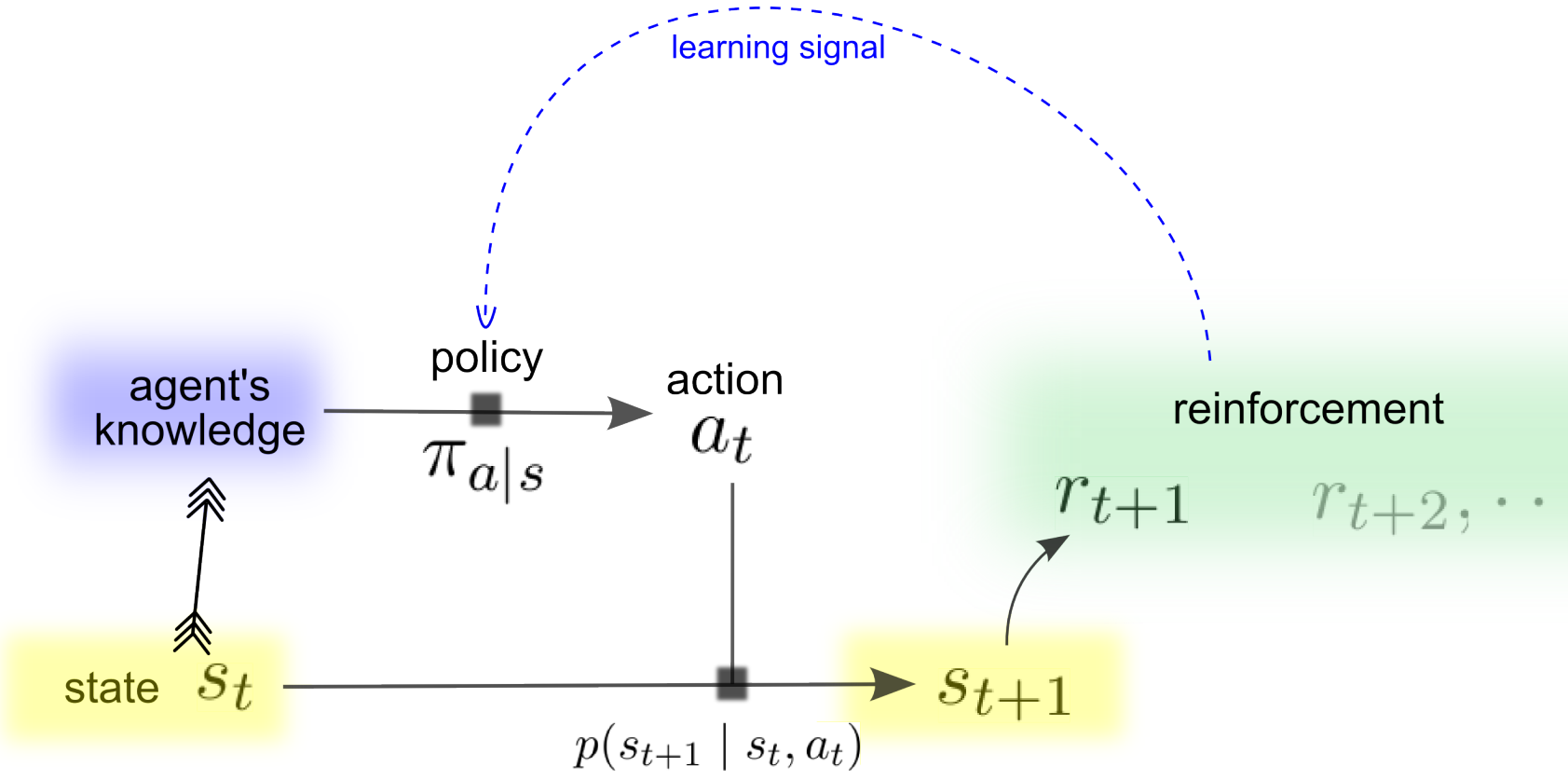
POMDP *Partially Observable Markov Decision Process*:
the agent is uncertain of true state. Possible "perceptual aliasing" - two states might look identical at a given moment.

MDP example

(from wikipedia)



MDP drawn as a graphical model



We want to use experience of s and r to improve π about a.

values and discounting

Useful quantity: $V_{\mathbf{s}}^{\pi}$ denotes the **long-term value** of being in state $\mathbf{s}$ (and following policy π thereafter).

In an "episodic" game (e.g. backgammon) $V_{\mathbf{s}}^{\pi}$ is the expected sum of reinforcement until the end of the game, using policy π:

$$V_{\mathbf{s}}^{\pi} = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\text{end}} r_{t+k+1} \quad | \, \mathbf{s}_t = \mathbf{s}\right]$$

But in situations where there's no obvious "end" to the game we *discount* future rewards by a parameter γ, giving

$$V_{\mathbf{s}}^{\pi} = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1} \quad | \, \mathbf{s}_t = \mathbf{s}\right] \tag{1}$$

$\gamma = 1$ is no discounting. If $\gamma \to 0$, V reduces to just the *immediate* reward.

the Bellman equation

Value functions having this form satisfy the *Bellman Equation*:

$$V_{\mathbf{s}}^{\pi} = \sum_{a} \pi_{a|\mathbf{s}} \sum_{\mathbf{s}'} P_{\mathbf{s}'|\mathbf{s},a} \left[R_{\mathbf{s}'} + \gamma V_{\mathbf{s}'}^{\pi} \right] \tag{2}$$

This expresses V's in terms of other V's, and is true for any stochastic policy π.

A policy π' is better than π if $V_{\mathbf{s}}^{\pi'} > V_{\mathbf{s}}^{\pi}$ (all else being equal). So there's at least one *optimal* policy (and could be several).

Denote the optimal policy $\pi^{\star}$. It has the best possible value function, which *is* unique:

$$V_{\mathbf{s}}^{\star} = \max_{\pi} V_{\mathbf{s}}^{\pi}$$

Bellman equation for an optimal policy

The optimal policy is *greedy* with respect to the optimal value function. This gives the Bellman equation for the optimal policy a particularly simple form:

$$V^\star_{\mathbf{s}} = \max_a \sum_{\mathbf{s}'} P_{\mathbf{s}'|\mathbf{s},a} \left[R_{\mathbf{s}'} + \gamma V^\star_{\mathbf{s}'} \right] \tag{3}$$

Ah ha! - the "policy" π is reduced to just the $\max$ operation here.

properties of $V^\star$, the optimal value function

- optimal policy is to be greedy w.r.t. $V^\star$
- only requires one step of look-ahead to decide what to do
- put a vector $V^\star$ in, and get the same vector out
  (like an eigen-equation)

So if we could just solve this equation, the policy would come for free: be greedy w.r.t. $V^\star$.

value iteration (solving Bellman for $V^\star$)

If you've *got* the optimal value function $V^\star$, you can shove it through equation 3 and the same thing pops out. But if you've got something else, it comes out different.

One can prove that what comes out is closer to $V^\star$ that what went in. This means you can take an initial *guess* for the value function and *iterate*:

$$V_{\mathbf{s}}^{k+1} = \max_a \sum_{\mathbf{s}'} P_{\mathbf{s}'|\mathbf{s},a} \left[R_{\mathbf{s}'} + \gamma V_{\mathbf{s}'}^k \right] \tag{4}$$

It can be shown that this converges to $V^\star$ from arbitrary initial V.

Value Iteration comes down to doing monster-sized matrix-multiplications, and max-ing.

It gives The Answer: $V^\star$ (and therefore $\pi^\star$).

It all sounds good... Too good...

recap: value iteration (solving Bellman for $V^\star$)

If you put in a *guess* value function V^k for the r.h.s., the l.h.s. gives you a V^{k+1} that is closer to $V^\star$, the optimal one:

$$V_{\mathbf{s}}^{k+1} = \max_a \sum_{\mathbf{s}'} P_{\mathbf{s}'|\mathbf{s},a} \left[R_{\mathbf{s}'} + \gamma V_{\mathbf{s}'}^k \right]$$

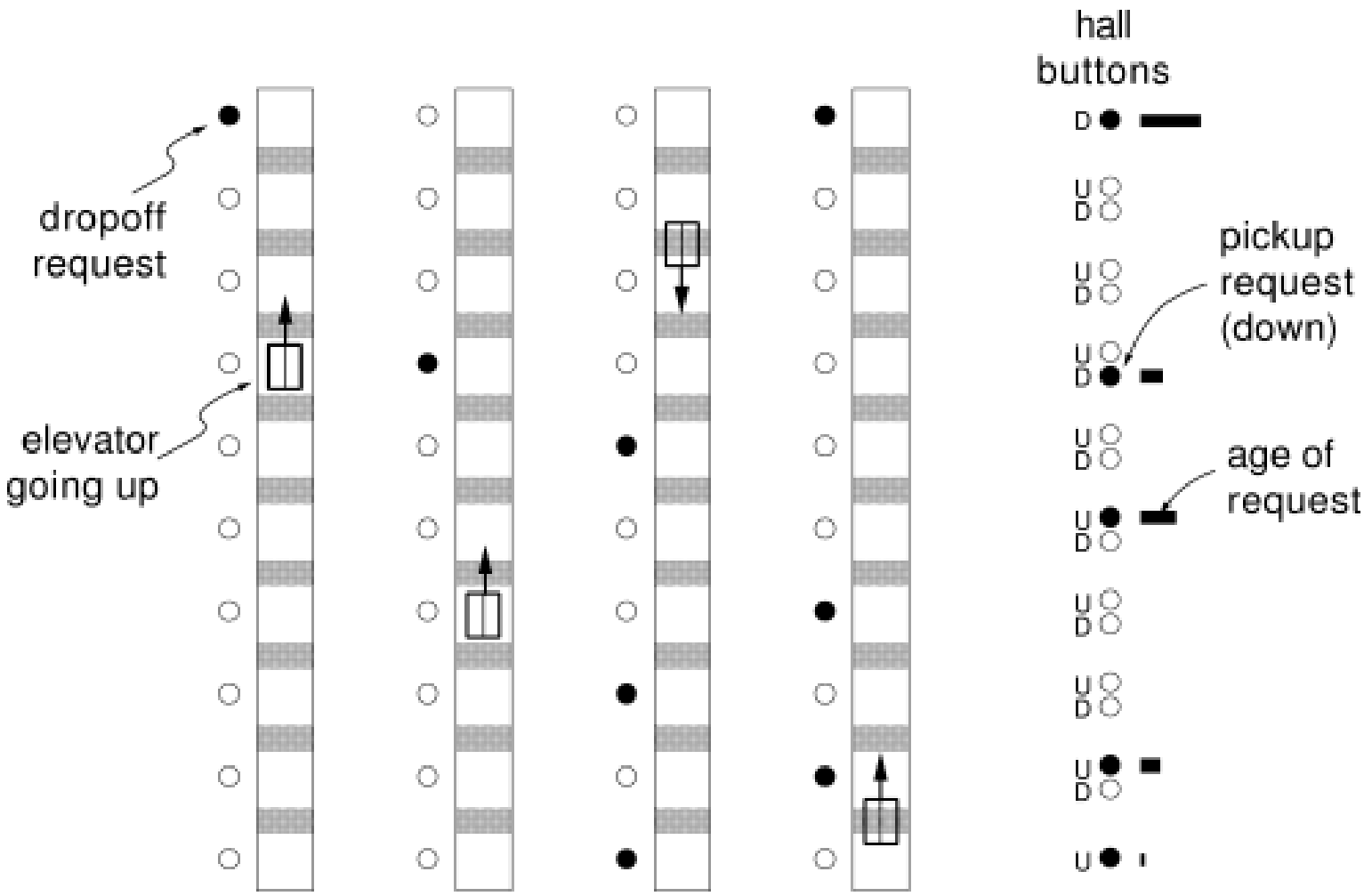
It can be shown that this converges to $V^\star$ from arbitrary initial V.
(nice demo: http://www.cs.ubc.ca/∼poole/demos/rl/q.html)

This algorithm assumes that we

1. **know** the expected rewards, $R_{\mathbf{s}}$ of all states,
2. **know** the "physics": $P_{\mathbf{s}'|\mathbf{s},a}$ of all $(\mathbf{s}', \mathbf{s}, a)$ tuples,
3. **can enumerate** the states in the first place, and
4. **know for sure** what current state is (so can choose action).

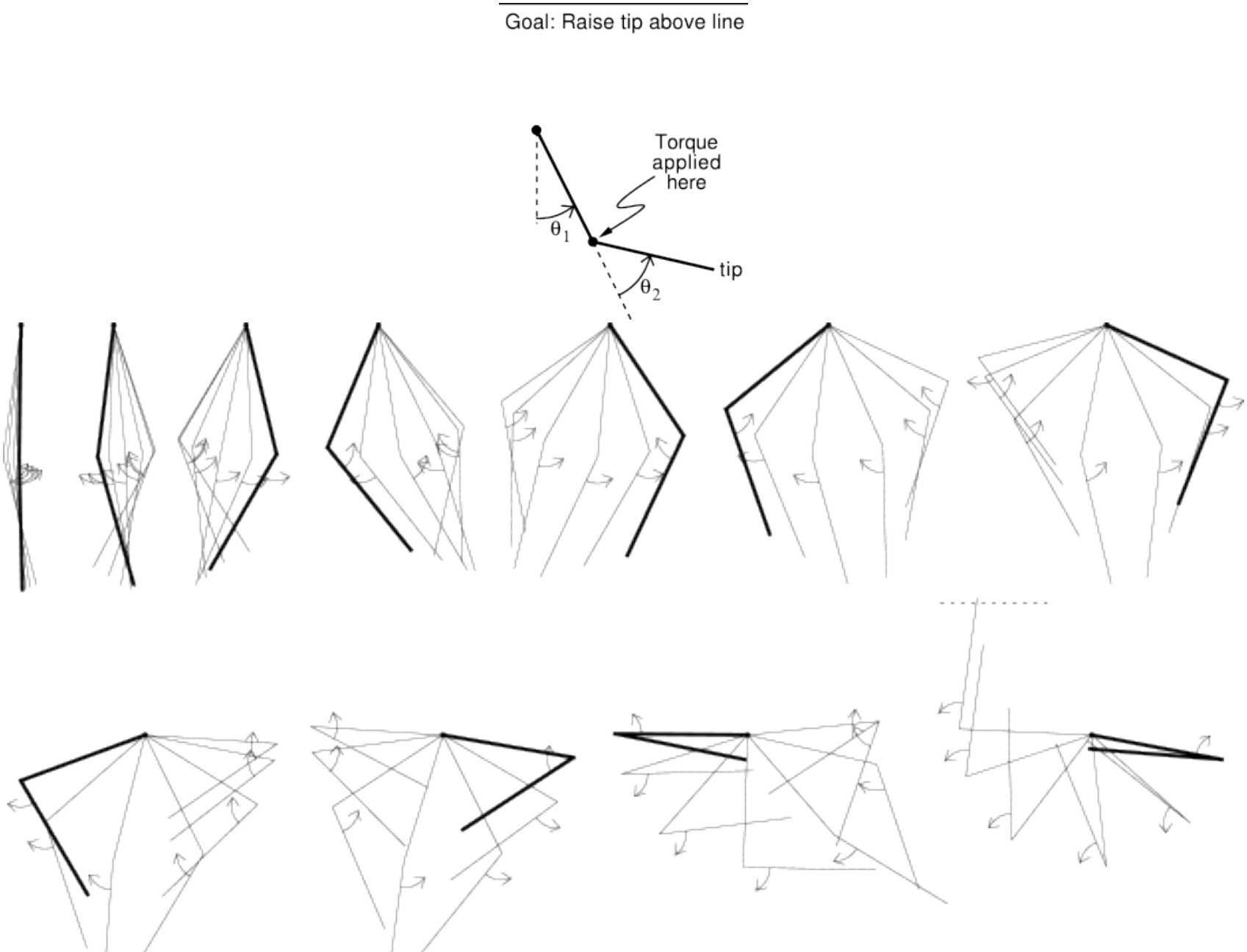
All of these are problematic for a real learner.

example: elevator dispatching

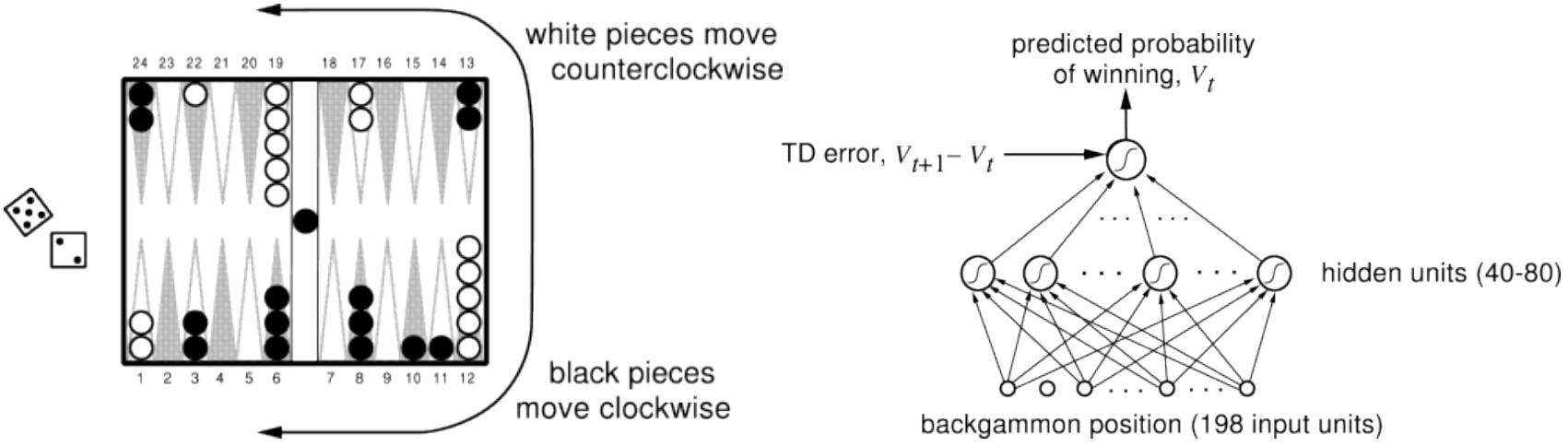


There are about 10^{22} states.

example: the acrobot



example: TD-gammon



Trained only on 300,000 games *against itself*, TD-gammon played as well as the best (dedicated) computer programs.

Went on to play at the level of the best human players in the world.

But see the 2015 *Nature paper on Atari games* for an even more impressive application.

values and policies

One approach to RL is to approximate Value Iteration by substituting real experience of states and rewards for the idealised knowledge:

$$V_{\mathbf{s}}^{k+1} = \max_a \sum_{\mathbf{s}'} P_{\mathbf{s}'|\mathbf{s},a} \left[R_{\mathbf{s}'} + \gamma V_{\mathbf{s}'}^k \right]$$

Draw inspiration from Value Iteration: *even though* the `max` operation was applied to *guessed* values for V, $V_{\mathbf{s}}^{k+1}$ moved closer to the optimal than $V_{\mathbf{s}}^k$ was.

In other words, this procedure converges despite assuming greedy actions w.r.t. the current (flawed) value function.

IDEA: let's do approximate value iteration, learning better and better V-values from experience as we go along, while using the existing V *fairly greedily*. $V_{\mathbf{s}}$ is a great big look-up table whose entries we're required to fill in. So how can we learn the table entries from experience?

learning better V, from experience

Suppose your guesses $V_{\mathbf{s}}$ started off random.

Starting from a state $\mathbf{s}$, by acting according to the current policy and keeping track of reinforcement receives, you could generate a sample estimate of the "true" $V_{\mathbf{s}_t}$:

$$v_{\mathbf{s}_t} = \sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1} \tag{5}$$

You could pull your current estimate for $V_{\mathbf{s}_t}$ towards that estimate:

$$\Delta V_{\mathbf{s}_t} = \eta \, \left[v_{\mathbf{s}_t} - V_{\mathbf{s}_t} \right] \tag{6}$$

where η is a learning rate.

Here $v_{\mathbf{s}_t}$ is just like a "target" in supervised learning.

learning from temporal differences (the TD trick)

To do the above we have to **wait** until the end of time... But there is a neat trick for avoiding this, called the **TD trick**, which stands for "Temporal Difference".

The idea is to *truncate* the series for $v_{\mathbf{s}_t}$ in equation 8 at some finite time, and *replace* the bit we've chopped off with our current estimate for it, V. Consider this hierarchy of approximations to $v_{\mathbf{s}_t}$:

$$
\begin{aligned}
v_{\mathbf{s}_t}^{(\infty)} &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \\
&\vdots \\
v_{\mathbf{s}_t}^{(2)} &= r_{t+1} + \gamma r_{t+2} + \gamma^2 V_{\mathbf{s}_{t+2}} \\
v_{\mathbf{s}_t}^{(1)} &= r_{t+1} + \gamma V_{\mathbf{s}_{t+1}} \\
v_{\mathbf{s}_t}^{(0)} &= V_{\mathbf{s}_t}
\end{aligned}
$$

the TD update equation for V

- 1st is correct but requires we waiting
- last is so severely truncated it never sees the world!
- use $v_{\mathbf{s}_t}^{(1)}$: it includes the true immediate reward r_{t+1}, but from then on it treats the stored V as if it were correct.

Equation 9 now becomes:

TD update equation for V

$$\Delta V_{\mathbf{s}_t} = \eta \left[r_{t+1} + \gamma V_{\mathbf{s}_{t+1}} - V_{\mathbf{s}_t} \right] \tag{7}$$

This learning rule converges on $V^\star$ provided every action can be tried in each state an arbitrary number of times (and η decays towards zero slowly enough). Pretty optimistic!!

aside: action selection

i.m.h.o. There's no really satisfactory solution to this. In practice ϵ**-greedy** action selection is often used: with some low probability ϵ, choose a *random* action. Another idea is to use something like the **softmax** function, favouring actions that lead to higher V.

the explore-exploit dilemma

Actions for reinforcement often conflict with actions for learning i.e. Choosing actions "greedily" isn't actually a good idea if you have the wrong value function, *e.g.* there's a risk we'll never even check out the non-greedy actions. In Value Iteration this wasn't an issue, because we simply assumed R was known already.

Note: choosing good actions from V depends on knowing "the physics", $P(\mathbf{s}' \mid \mathbf{s}, a)$

Issue 1: knowledge of $P(\mathbf{s}' \mid \mathbf{s}, a)$

There's no point in knowing that some state $\mathbf{s}'$ is great if you have no idea what action to take to get yourself there, so on the face of it knowing $V_{\mathbf{s}}$ is not going to help much.

We've used $P(\mathbf{s}' \mid \mathbf{s}, a)$ - knowledge of what states are accessible from the current one, and what action to do to get there.

There are variants of TD learning that are "model-free", meaning they can work even without $P(\mathbf{s}' \mid \mathbf{s}, a)$.

Consider a more detailed value function, $Q^{\pi}_{\mathbf{s},a}$: the expected long-term discounted value of carrying out a in state $\mathbf{s}$, and using policy π thereafter.

$Q_{\mathbf{s},a}$ is a look-up table: how can we learn its entries from experience?

SARSA

Following exactly the same argument as for V leads to a Q-value equivalent of equation 10:

SARSA

'On-policy' TD learning of Q values:

$$\Delta Q_{\mathbf{s}_t,a_t} = \eta \left[r_{t+1} + \gamma Q_{\mathbf{s}_{t+1},a_{t+1}} - Q_{\mathbf{s}_t,a_t} \right]$$

- called SARSA, because it uses $\mathbf{s}_t$, a_t, r_{t+1}, $\mathbf{s}_{t+1}$ and a_{t+1}!
- fine if all actions are chosen greedily
- but then no exploration takes place (so use ϵ-greedy)
- if we do other actions, SARSA pulls Q in wrong direction
- called an 'on-policy' algorithm, since if 'off-policy' actions are taken the algorithm makes no sense.

Q-learning

Q learning

$$\Delta Q_{\mathbf{s}_t,a_t} = \eta \left[r_{t+1} + \gamma \max_{a'} Q_{\mathbf{s}_{t+1},a'} - Q_{\mathbf{s}_t,a_t} \right]$$

- Q is pulled toward the immediate reward r plus the discounted return that *would* follow if the greedy action were taken next.
- we are free to explore other 'off-policy' actions while still learning the correct Q-values.
- have an "exploratory" policy for purposes of learning the exploitative one...
- proven to converge to the correct optimal values (provided the exploration policy carries out every action in every state an arbitrary number of times...!)

(a demo: http://www.cs.ubc.ca/∼poole/demos/rl/q.html)

Issue 2: too many states

Huge state spaces have two problems: the memory involved in storing the lookup tables, and learning their entries: cramming that much experience into one life, no matter how artificial, simply isn't feasible.

eg: Backgammon

you have P (it's deterministic) and R, the game is Markovian, and is perfectly observable. But there are $\sim 10^{20}$ states, so it would take millions of years to solve for $V^\star$.

But it may be that the entire perfect table of 1,000,000 Q-values for some problem might be able to be captured by a much smaller number of free parameters, say 10. And in that case the learning task takes place in a 10 dimensional arena instead of the million dimensional one.

the original (lookup table) TD as gradient descent

Let's say we want to minimize the 'cost'

$$C = \frac{1}{2}(v_{\mathbf{s}}^{(1)} - V_{\mathbf{s}})^2$$

where $v_{\mathbf{s}}^{(1)} = r + \gamma V_{\mathbf{s}'}$ as described earlier. Gradient descent:

$$\Delta V_{\mathbf{s}_t} \;=\; -\eta \frac{\partial C}{\partial V_{\mathbf{s}_t}} \;=\; \eta\,(v_{\mathbf{s}_t}^{(1)} - V_{\mathbf{s}_t})$$

where η is some suitably small learning rate. This just gives the TD update, equation 10.

learning with function approximation

Now we're going to capture the V function with some parameters θ of a function approximator instead of using a lookup table:

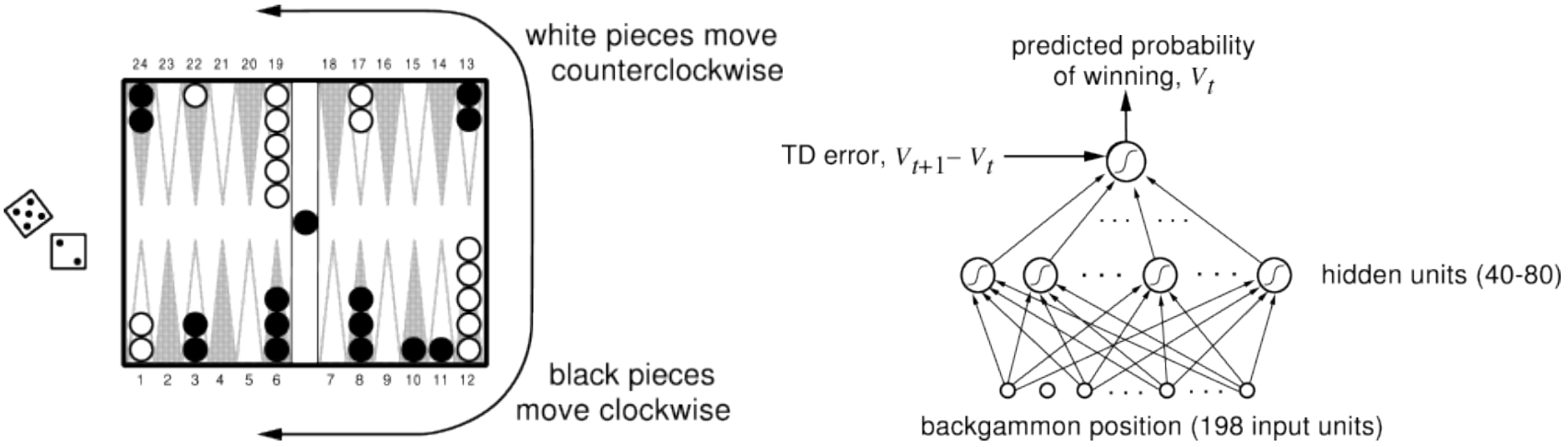
$$\Delta\theta_i \;=\; \eta \frac{\partial C}{\partial \theta_i} \;=\; \eta \left(v_{\mathbf{s}}^{(1)} - V_{\mathbf{s}}\right) \frac{\partial V_{\mathbf{s}}}{\partial \theta_i}$$

So for example if the function approximator was just a linear 'neuron' with output $V_{\mathbf{s}} = \sum_i w_i s_i$ we would have

$$\Delta w_i \;=\; \eta \left(r + \gamma V_{\mathbf{s}'} - V_{\mathbf{s}}\right) s_i$$

Hopefully you can see that this could easily be extended to BackPropagation through whole networks, or any parameterized function approximator. We can also represent and learn Q values this way.

example: TD-gammon



- the world's best backgammon player? (cf. no hand-coded program has ever got better than weak human performance)
- From a given position, the program goes through each available action, and reads off the neural network's estimate for the V-value of the resulting board state.
- ~ 100 weights take the place of a LUT with $\sim 10^{40}$ rows.
- reinforcement only supplied at the conclusion of a game.
- learned by playing 10^6 games against variants of itself.

Top human players now use openings 'invented' by TD-gammon.

the problem with function approximation

Replacing the LUT for V or Q with a function approximator destroys the convergence guarantee that the look-up table version has.

- learning can easily be made unstable, with wild variations in behaviour and no apparent improvement over time.
- in a LUT all the values can vary independently. But with an approximator they all potentially interact with one another. So for example a large reward that improves the $V_{\mathbf{s}}$ estimate might also make some other unrelated estimate, say $V_{\mathbf{s}'}$, much worse.

To recap, we have some policy π and learn an approximation $\tilde{V}^{\pi}$ to its value function. We then have a new policy π' defined by being greedy with respect to those values.

But it can be shown that unless our approximation $\tilde{V}$ is in fact *exact* (such as a look-up table), π' can be *arbitrarily worse* than π. That's pretty bad!

blah blah blah

recap: value iteration (solving Bellman for $V^\star$)

If you put in a *guess* value function V^k for the r.h.s., the l.h.s. gives you a V^{k+1} that is closer to $V^\star$, the optimal one:

$$V_{\mathbf{s}}^{k+1} = \max_a \sum_{\mathbf{s}'} P_{\mathbf{s}'|\mathbf{s},a} \left[R_{\mathbf{s}'} + \gamma V_{\mathbf{s}'}^k \right]$$

It can be shown that this converges to $V^\star$ from arbitrary initial V.
(nice demo: http://www.cs.ubc.ca/∼poole/demos/rl/q.html)

This algorithm assumes that we

1. **know** the expected rewards, $R_{\mathbf{s}}$ of all states,
2. **know** the "physics": $P_{\mathbf{s}'|\mathbf{s},a}$ of all $(\mathbf{s}', \mathbf{s}, a)$ tuples,
3. **can enumerate** the states in the first place, and
4. **know for sure** what current state is (so can choose action).

All of these are problematic for a real learner.

values and policies

One approach to RL is to approximate Value Iteration by substituting real experience of states and rewards for the idealised knowledge:

$$V_{\mathbf{s}}^{k+1} = \max_a \sum_{\mathbf{s}'} P_{\mathbf{s}'|\mathbf{s},a} \left[R_{\mathbf{s}'} + \gamma V_{\mathbf{s}'}^{k} \right]$$

Even though it used a `max` operation over the *guessed* values for V, $V_{\mathbf{s}}^{k+1}$ is closer to the optimal policy than $V_{\mathbf{s}}^{k}$ was.

In other words, this procedure converges despite assuming greedy actions w.r.t. the current value function.

We're going to do approximate value iteration, learning better and better V-values as we go along, but meanwhile using the existing ones *fairly greedily*. $V_{\mathbf{s}}$ is a great big look-up table whose entries we're required to fill in. So how can we learn the table entries from experience?

learning better V, from experience

Suppose your guesses $V_{\mathbf{s}}$ started off random.

Starting from a state $\mathbf{s}$, by acting according to the current policy and keeping track of reinforcement receives, you could generate a sample estimate of the "true" $V_{\mathbf{s}_t}$:

$$v_{\mathbf{s}_t} = \sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1} \tag{8}$$

You could pull your current estimate for $V_{\mathbf{s}_t}$ towards that estimate:

$$\Delta V_{\mathbf{s}_t} = \eta \, \left[v_{\mathbf{s}_t} - V_{\mathbf{s}_t} \right] \tag{9}$$

where η is a learning rate.

Here $v_{\mathbf{s}_t}$ is just like a "target" in supervised learning.

learning from temporal differences (the TD trick)

To do the above we have to **wait** until the end of time... But there is a neat trick for avoiding this, called the **TD trick**, which stands for "Temporal Difference".

The idea is to *truncate* the series for $v_{\mathbf{s}_t}$ in equation 8 at some finite time, and *replace* the bit we've chopped off with our current estimate for it, V. Consider this hierarchy of approximations to $v_{\mathbf{s}_t}$:

$$
\begin{aligned}
v_{\mathbf{s}_t}^{(\infty)} &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \\
&\vdots \\
v_{\mathbf{s}_t}^{(2)} &= r_{t+1} + \gamma r_{t+2} + \gamma^2 V_{\mathbf{s}_{t+2}} \\
v_{\mathbf{s}_t}^{(1)} &= r_{t+1} + \gamma V_{\mathbf{s}_{t+1}} \\
v_{\mathbf{s}_t}^{(0)} &= V_{\mathbf{s}_t}
\end{aligned}
$$

the TD update equation for V

- 1st is correct but requires we waiting
- last is so severely truncated it never sees the world!
- use $v_{\mathbf{s}_t}^{(1)}$: it includes the true immediate reward r_{t+1}, but from then on it treats the stored V as if it were correct.

Equation 9 now becomes:

TD update equation for V

$$\Delta V_{\mathbf{s}_t} = \eta \left[r_{t+1} + \gamma V_{\mathbf{s}_{t+1}} - V_{\mathbf{s}_t} \right] \tag{10}$$

This learning rule converges on $V^\star$ provided every action can be tried in each state an arbitrary number of times (and η decays towards zero slowly enough). Pretty optimistic!!

aside: action selection

i.m.h.o. There's no really satisfactory solution to this. In practice ϵ**-greedy** action selection is often used: with some low probability ϵ, choose a *random* action. Another idea is to use something like the **softmax** function, favouring actions that lead to higher V.

the explore-exploit dilemma

Actions for reinforcement often conflict with actions for learning i.e. Choosing actions "greedily" isn't actually a good idea if you have the wrong value function, *e.g.* there's a risk we'll never even check out the non-greedy actions. In Value Iteration this wasn't an issue, because we simply assumed R was known already.

Note: choosing good actions from V depends on knowing "the physics", $P(\mathbf{s}' \mid \mathbf{s}, a)$

Issue 2: knowledge of $P(\mathbf{s}' \mid \mathbf{s}, a)$

There's no point in knowing that some state $\mathbf{s}'$ is great if you have no idea what action to take to get yourself there, so on the face of it knowing $V_{\mathbf{s}}$ is not going to help much.

We've used $P(\mathbf{s}' \mid \mathbf{s}, a)$ - knowledge of what states are accessible from the current one, and what action to do to get there.

There are variants of TD learning that are "model-free", meaning they can work even without $P(\mathbf{s}' \mid \mathbf{s}, a)$.

Consider a more detailed value function, $Q^{\pi}_{\mathbf{s},a}$: the expected long-term discounted value of carrying out a in state $\mathbf{s}$, and using policy π thereafter.

$Q_{\mathbf{s},a}$ is a look-up table: how can we learn its entries from experience?

SARSA

Following exactly the same argument as for V leads to a Q-value equivalent of equation 10:

SARSA

'On-policy' TD learning of Q values:

$$\Delta Q_{\mathbf{s}_t,a_t} = \eta \left[r_{t+1} + \gamma Q_{\mathbf{s}_{t+1},a_{t+1}} - Q_{\mathbf{s}_t,a_t} \right]$$

- called SARSA, because it uses $\mathbf{s}_t$, a_t, r_{t+1}, $\mathbf{s}_{t+1}$ and a_{t+1}!
- fine if all actions are chosen greedily
- but then no exploration takes place (so use ϵ-greedy)
- if we do other actions, SARSA pulls Q in wrong direction
- called an 'on-policy' algorithm, since if 'off-policy' actions are taken the algorithm makes no sense.

Q-learning

Q learning

$$\Delta Q_{\mathbf{s}_t,a_t} = \eta \left[r_{t+1} + \gamma \max_{a'} Q_{\mathbf{s}_{t+1},a'} - Q_{\mathbf{s}_t,a_t} \right]$$

- Q is pulled toward the immediate reward r plus the discounted return that *would* follow if the greedy action were taken next.
- we are free to explore other 'off-policy' actions while still learning the correct Q-values.
- have an "exploratory" policy for purposes of learning the exploitative one...
- proven to converge to the correct optimal values (provided the exploration policy carries out every action in every state an arbitrary number of times...!)

(a demo: http://www.cs.ubc.ca/∼poole/demos/rl/q.html)

Issue 3: too many states

Huge state spaces have two problems: the memory involved in storing the lookup tables, and learning their entries: cramming that much experience into one life, no matter how artificial, simply isn't feasible.

eg: Backgammon

you have P (it's deterministic) and R, the game is Markovian, and is perfectly observable. But there are $\sim 10^{20}$ states, so it would take millions of years to solve for $V^\star$.

But it may be that the entire perfect table of 1,000,000 Q-values for some problem might be able to be captured by a much smaller number of free parameters, say 10. And in that case the learning task takes place in a 10 dimensional arena instead of the million dimensional one.

the original (lookup table) TD as gradient descent

Let's say we want to minimize the 'cost'

$$C = \frac{1}{2}(v_{\mathbf{s}}^{(1)} - V_{\mathbf{s}})^2$$

where $v_{\mathbf{s}}^{(1)} = r + \gamma V_{\mathbf{s}'}$ as described earlier. Gradient descent:

$$\Delta V_{\mathbf{s}_t} \;=\; -\eta \frac{\partial C}{\partial V_{\mathbf{s}_t}} \;=\; \eta\,(v_{\mathbf{s}_t}^{(1)} - V_{\mathbf{s}_t})$$

where η is some suitably small learning rate. This just gives the TD update, equation 10.

learning with function approximation

Now we're going to capture the V function with some parameters θ of a function approximator instead of using a lookup table:

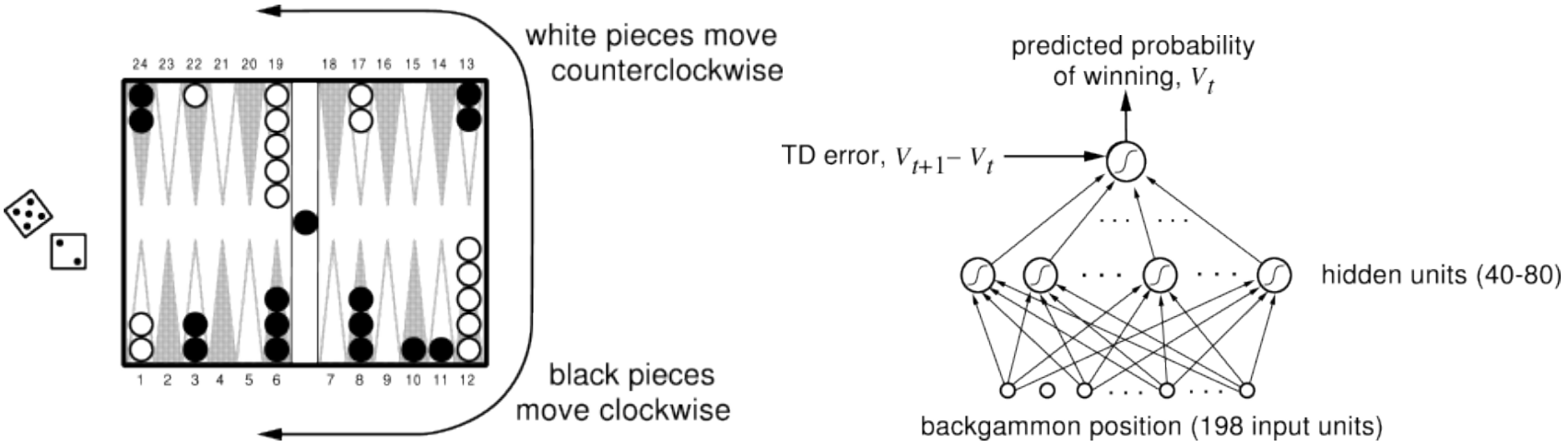
$$\Delta\theta_i \;=\; \eta \frac{\partial C}{\partial \theta_i} \;=\; \eta \left(v_{\mathbf{s}}^{(1)} - V_{\mathbf{s}}\right) \frac{\partial V_{\mathbf{s}}}{\partial \theta_i}$$

So for example if the function approximator was just a linear 'neuron' with output $V_{\mathbf{s}} = \sum_i w_i s_i$ we would have

$$\Delta w_i \;=\; \eta \left(r + \gamma V_{\mathbf{s}'} - V_{\mathbf{s}}\right) s_i$$

Hopefully you can see that this could easily be extended to BackPropagation through whole networks, or any parameterized function approximator. We can also represent and learn Q values this way.

example: TD-gammon



- the world's best backgammon player? (cf. no hand-coded program has ever got better than weak human performance)
- From a given position, the program goes through each available action, and reads off the neural network's estimate for the V-value of the resulting board state.
- ~ 100 weights take the place of a LUT with $\sim 10^{40}$ rows.
- reinforcement only supplied at the conclusion of a game.
- learned by playing 10^6 games against variants of itself.

Top human players now use openings 'invented' by TD-gammon.

the problem with function approximation

Replacing the LUT for V or Q with a function approximator destroys the convergence guarantee that the look-up table version has.

- learning can easily be made unstable, with wild variations in behaviour and no apparent improvement over time.
- in a LUT all the values can vary independently. But with an approximator they all potentially interact with one another. So for example a large reward that improves the $V_{\mathbf{s}}$ estimate might also make some other unrelated estimate, say $V_{\mathbf{s}'}$, much worse.

To recap, we have some policy π and learn an approximation $\tilde{V}^{\pi}$ to its value function. We then have a new policy π' defined by being greedy with respect to those values.

But it can be shown that unless our approximation $\tilde{V}$ is in fact *exact* (such as a look-up table), π' can be *arbitrarily worse* than π. That's pretty bad!