Deep Reinforcement Learning

Deep Learning: Bryan Pardo, Northwestern University, Fall 2020

Learning Types

- Supervised learning:
  - (Input, output) pairs of the function to be learned are given (e.g. image labeling)
- Unsupervised Learning:
  - No human labels provided (e.g. language modeling, image reconstruction)
- Reinforcement learning:
  - Reward or punishment for actions (winning or losing a game)

Reinforcement Learning

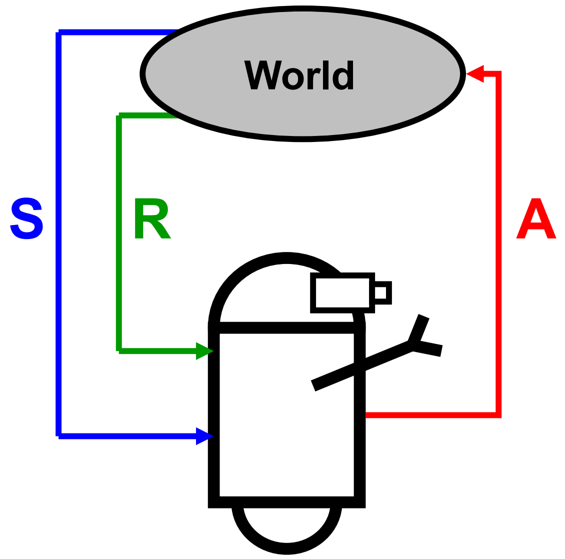
- Task
  - Learn how to behave to achieve a goal
  - Learn through experience from trial and error

- Examples
  - Game playing: The agent knows when it wins, but doesn't know the appropriate action in each state along the way

  - Control: a traffic system can measure the delay of cars, but not know how to decrease it.

Basic RL Model

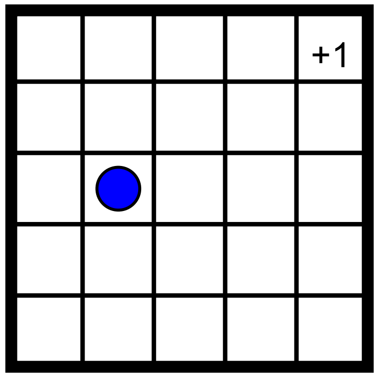
1. Observe state, s_t
2. Decide on an action, a_t
3. Perform action
4. Observe new state, s_{t+1}
5. Observe reward, r_{t+1}
6. Learn from experience
7. Repeat



Goal: Find a control policy that will maximize the observed rewards over the lifetime of the agent

A Cannonical Example: Gridworld

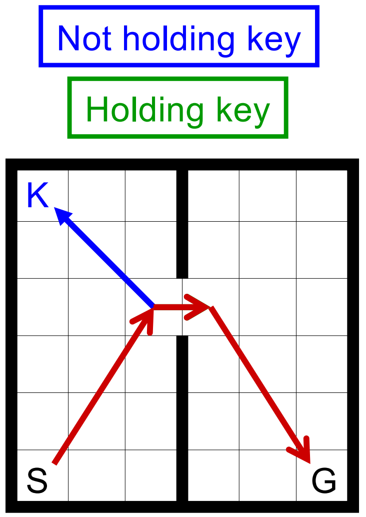
- States are grid cells
- 4 actions: N, S, E, W
- Reward for entering top right cell
- -0.01 for every other move



The Markov Property

- RL needs a set of states that are Markov
  - Everything you need to know to make a decision is included in the state
  - Not allowed to consult the past

- Rule-of-thumb
  - If you can calculate the reward function from the state without any additional information, you're OK

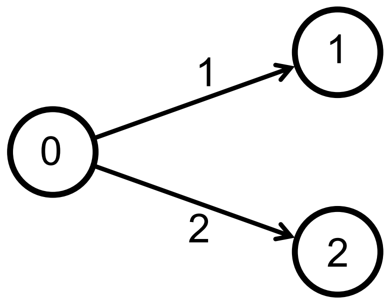


We need some background

- Simple decision theory
- Markov Decision Processes
- Value functions
- Dynamic programming

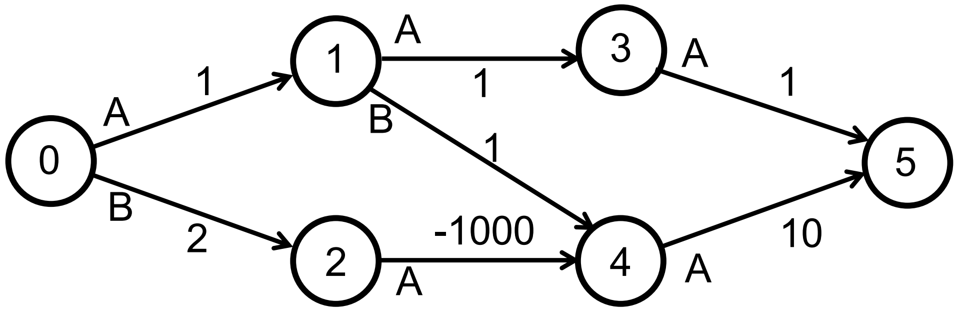
Making Single Decisions

- Single decision to be made
  - Multiple discrete actions
  - Each action has an associated reward
- Goal is to maximize reward
  - Just pick the action with the largest reward
- State 0 has a value of 2
  - Reward from taking the best action



Markov Decision Processes

- We can generalize the previous example to multiple sequential decisions
    - Each decision affects subsequent decisions
- This is formally modeled by a Markov Decision Process (MDP)



Northwestern University, EECS 349, 2017

Markov Decision Processes

- Formally, a MDP is
  - A set of states, $S = \{s_1, s_2, \ldots, s_n\}$
  - A set of actions, $A = \{a_1, a_2, \ldots, a_m\}$
  - A reward function, $R: S \times A \times S \rightarrow \Re$
  - A transition function, $P_{ij}^{a} = P(s_{t+1} = j \mid s_t = i, a_t = a)$
    - Sometimes $T: S \times A \rightarrow S$

- We want to learn a policy, $\pi: S \rightarrow A$
  - Maximize sum of rewards we see over our lifetime

Policies

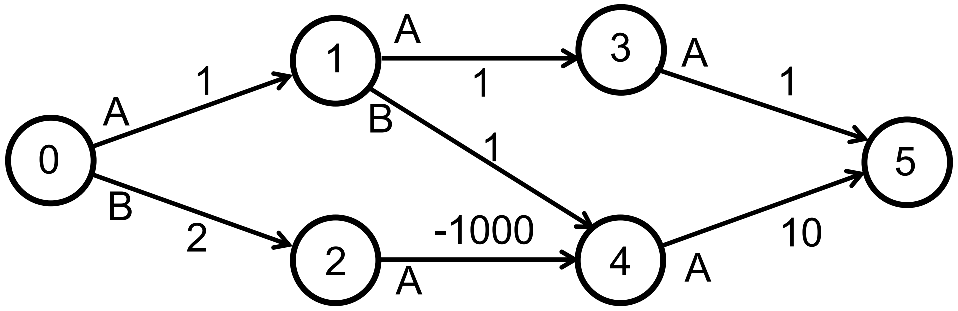
- A policy $\pi(s)$ returns the action to take in state s.

- There are 3 policies for this MDP

Policy 1: 0 →1 →3 →5

Policy 2: 0 →1 →4 →5

Policy 3: 0 →2 →4 →5



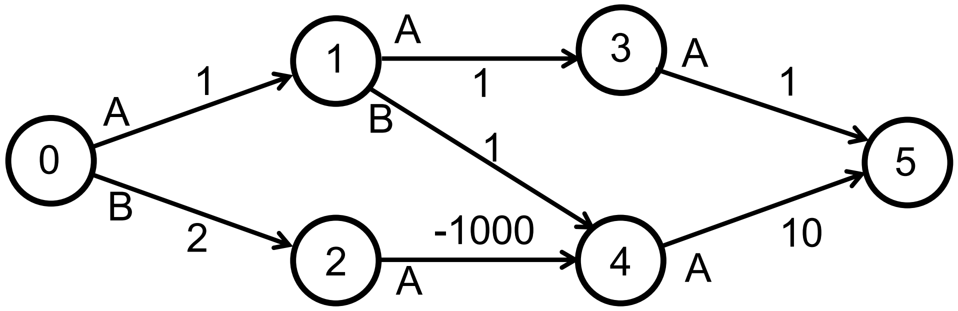
Comparing Policies

- Which policy is best?
- Order them by how much reward they see

Policy 1: 0 →1 →3 →5 = 1 + 1 + 1 = 3

Policy 2: 0 →1 →4 →5 = 1 + 1 + 10 = 12

Policy 3: 0 →2 →4 →5 = 2 – 1000 + 10 = -988

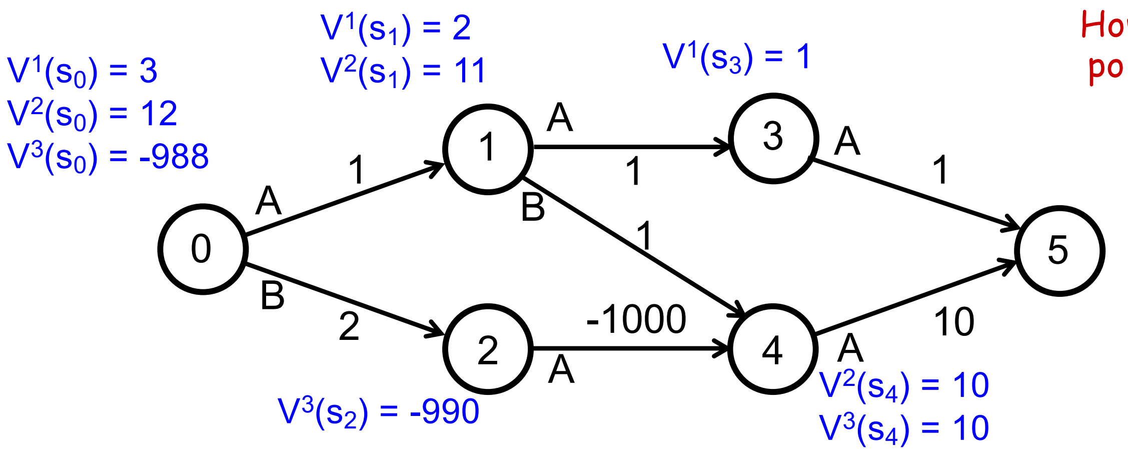


Value Functions

s' is the next state

a' is the next action

- For a given policy, we can associate a value with each state
  - How good is it to run policy π from that state s?
  - This is the state value function, V



How do you tell which policy to follow from each state?

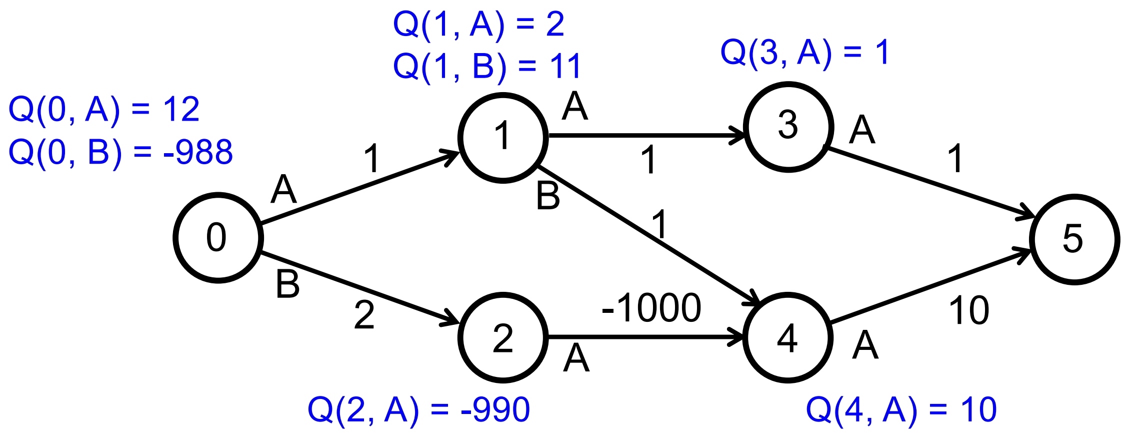
$$V^{\pi}(s) = R(s, \pi(s), s') + V^{\pi}(s')$$

Q Functions

s' is the next state

a' is the next action

- Define value without specifying the policy
  - It is the value of taking action A from state S and then performing optimally, thereafter



$$Q(s, a) = R(s, a, s') + \max_{a'} Q(s', a')$$

Value Functions

- These can be extended to probabilistic actions
  - (for when the results of an action are not certain, or when a policy is probabilistic)

$$V^{\pi}(s) = \sum_{s'} P(s' | s, \pi(s)) \left(R(s, \pi(s), s') + V^{\pi}(s') \right)$$

$$Q(s,a) = \sum_{s'} P(s' | s, a) \left(R(s, a, s') + \max_{a'} Q(s', a') \right)$$

Getting the Policy

- If we have the value function, then finding the optimal policy, $\pi^*(s)$, is easy…just find the policy that maximized value

$$\pi^*(s) = \arg\max_a (R(s, a, s') + V^\pi(s'))$$

$$\pi^*(s) = \arg\max_a Q(s, a)$$

Learning Policies Directly

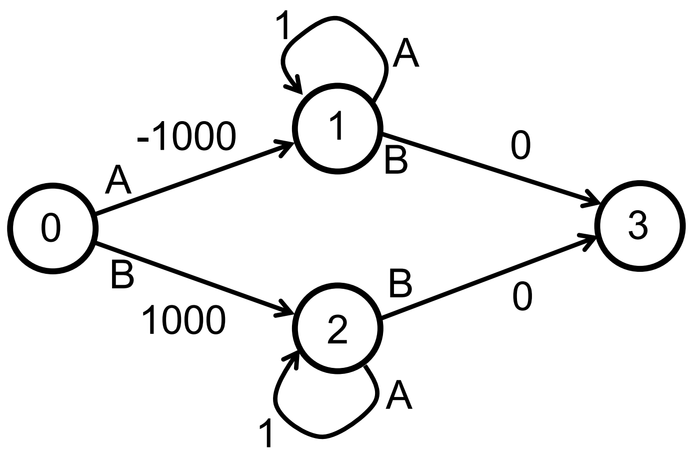
- Run whole policy, then receive a single reward
- Reward measures success of the whole policy
- If there are a small number of policies, we can exhaustively try them all
- This is not possible in most interesting problems

Problems with Our Functions

- Consider this MDP
  - Number of steps is now unlimited because of loops
  - Value of states 1 and 2 is infinite for some policies

$$\begin{aligned} Q(1, A) &= 1 + Q(1, A) \\ &= 1 + 1 + Q(1, A) \\ &= 1 + 1 + 1 + Q(1, A) \\ &= \ldots \end{aligned}$$

- This is bad
  - All policies with a non-zero reward cycle have infinite value



Better Value Functions

- Introduce the ***discount factor*** γ, to get around the problem of infinite value
  - Three interpretations
    - Probability of living to see the next time step
    - Measure of the uncertainty inherent in the world
    - Makes the mathematics work out nicely

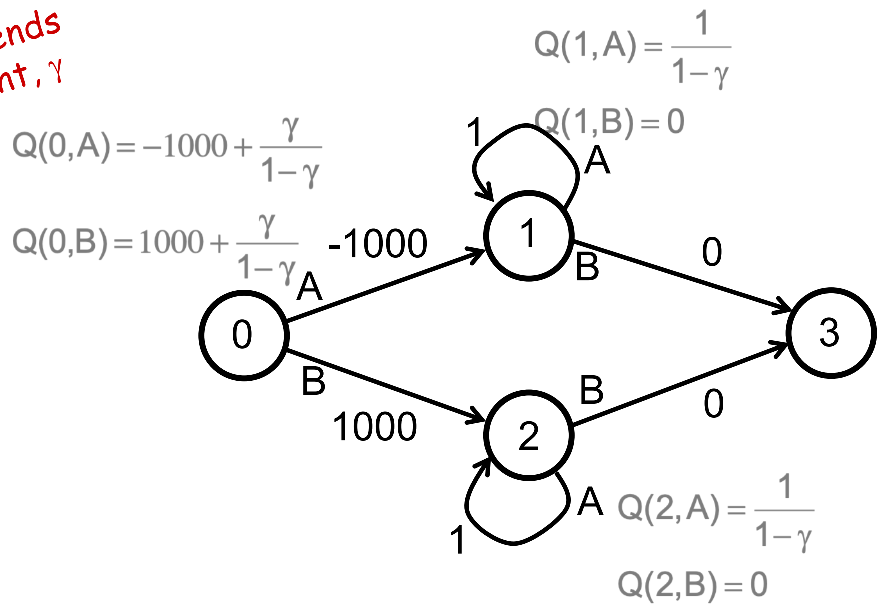
Assume $0 \leq \gamma \leq 1$

$$V^{\pi}(s) = R(s, \pi(s), s') + \gamma V^{\pi}(s')$$

$$Q(s, a) = R(s, a, s') + \gamma \max_{a'} Q(s', a')$$

Better Value Functions

Value now depends on the discount, γ

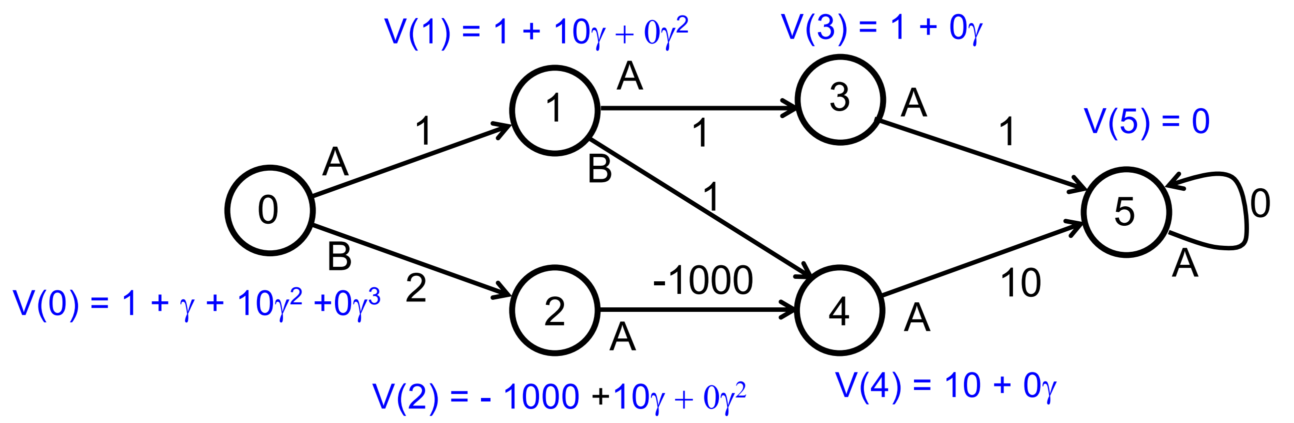


$Q(0,A) = -1000 + \frac{\gamma}{1-\gamma}$
$Q(0,B) = 1000 + \frac{\gamma}{1-\gamma}$
$Q(1,A) = \frac{1}{1-\gamma}$
$Q(1,B) = 0$
$Q(2,A) = \frac{1}{1-\gamma}$
$Q(2,B) = 0$
0
1
2
3
A
B
-1000
1000
1
0

- Optimal Policy:
  - $\pi(0)$ = B
  - $\pi(1)$ = A
  - $\pi(2)$ = A

Dynamic Programming

- Given the complete MDP model, we can compute the optimal value function directly



[Bertsekas, 87, 95a, 95b]

Reinforcement Learning

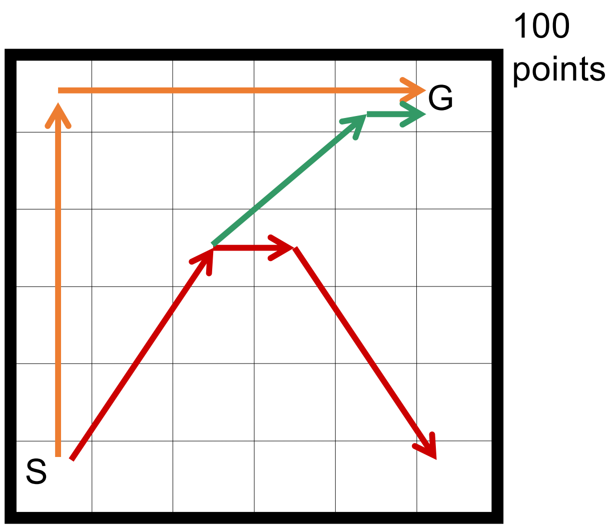
- What happens if we don' t have the whole MDP?
    - We know the states and actions
    - We don't have the system model (transition function) or reward function
- We' re only allowed to sample from the MDP
    - Can observe experiences (s, a, r, s')
    - Need to perform actions to generate new experiences
- This is Reinforcement Learning (RL)
    - Sometimes called Approximate Dynamic Programming (ADP)

Learning Value Functions

- We still want to learn a value function
  - We're forced to approximate it iteratively
  - Based on direct experience of the world

- Four main algorithms
  - Certainty equivalence
  - TD λ learning
  - Q-learning
  - SARSA

How are we going to do this?



- Reward whole policies?
  - That could be a pain
- What about incremental rewards?
  - Everything has a reward of 0 except for the goal
- Now what???

Exploration vs. Exploitation

- We want to pick good actions most of the time, but also do some exploration
- Exploring means we can learn better policies
- But, we want to balance known good actions with exploratory ones
- This is the **exploration/exploitation** problem

On-Policy vs. Off Policy

- On-policy algorithms
  - Final policy is influenced by the exploration policy
  - Generally, the exploration policy needs to be “close” to the final policy
  - Can get stuck in local maxima

Given enough experience

- Off-policy algorithms
  - Final policy is independent of exploration policy
  - Can use arbitrary exploration policies
  - Will not get stuck in local maxima

Picking Actions

ε-greedy
- Pick best (greedy) action with probability 1 - ε
- Otherwise, pick a random action

- Boltzmann (Soft-Max)
  - Pick an action based on its Q-value

$$P(a|s) = \frac{e^{\left(\frac{Q(s,a)}{\tau}\right)}}{\sum_{a'} e^{\left(\frac{Q(s,a')}{\tau}\right)}}$$

…where τ is the "temperature"

TD(l)

- TD-learning estimates the value function directly
  - Don't try to learn the underlying MDP

- Keep an estimate of $V^{\pi}(s)$ in a table
  - Update these estimates as we gather more experience
  - Estimates depend on exploration policy, π
  - TD is an on-policy method

[Sutton, 88]

What are we learning here?

Approach 1: Run the policy until you see the final outcome, then update your value V(S). If G is the final outcome…then

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big]$$

Approach 2: Update as you go, using your existing estimate of state values as your proxy for the final reward

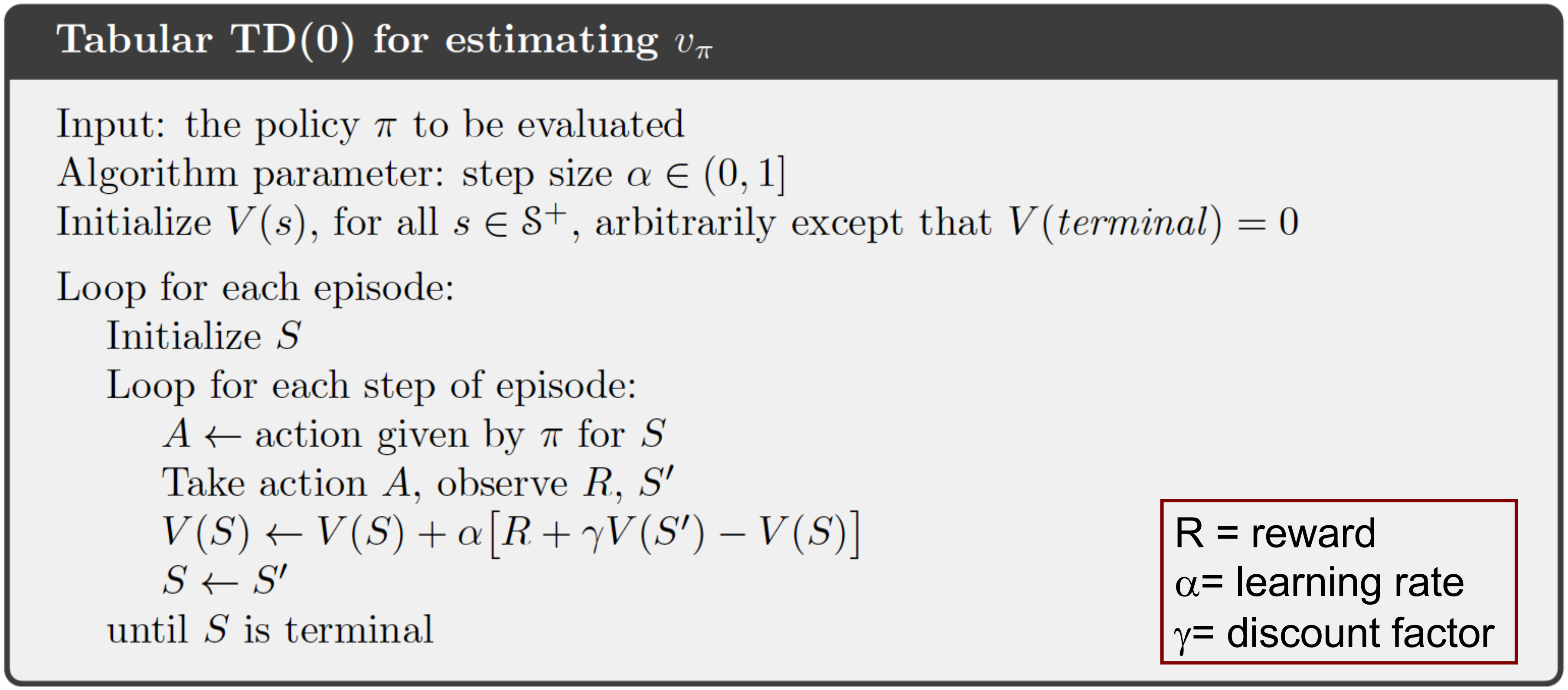
$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$$

The temporal difference error: δ_t

This is the difference between our current estimate of this state’s value and our bootstrapped estimate of the value, now that we’ve taken an action according to a policy.

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

TD(0)-Learning Algorithm



Tabular TD(0) for estimating v_π

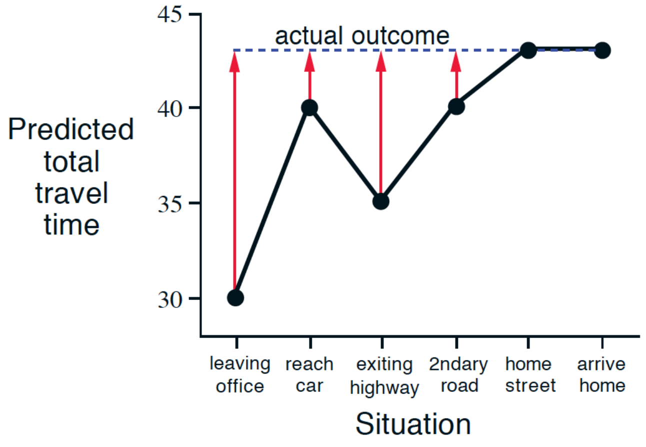
```
Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

R = reward
α= learning rate
γ= discount factor

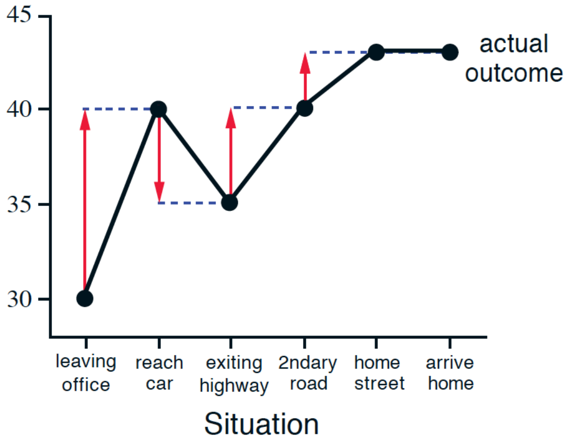
this formulation is from Sutton & Barto's "Reinforcement Learning"

Driving example

(MC) Waiting till end

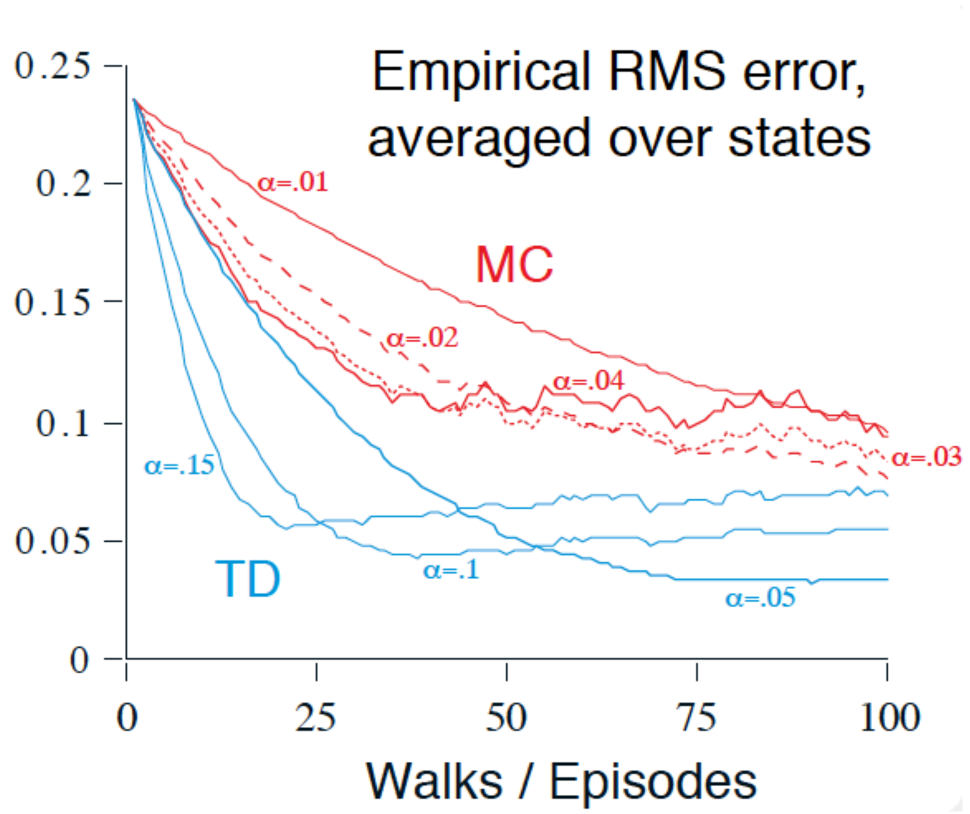


(TD) Updating incrementally



State	*Elapsed Time (minutes)*	*Predicted Time to Go*	*Predicted Total Time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

Why not wait till the end to do our update?



TD-Learning

- $V^{\pi}(s)$ is guaranteed to converge to $V^{*}(s)$
    - After an infinite number of experiences
    - If we decay the learning rate

$$\alpha_t = \frac{c}{c+t}$$ will work, where c is a constant and t is the step index

- In practice, we often don't need value convergence
    - Policy convergence generally happens sooner

What if we want to predict the action, not the state?

Replace our existing update as you go learning rule:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big]$$

With one that updates the Q table instead of the V table:

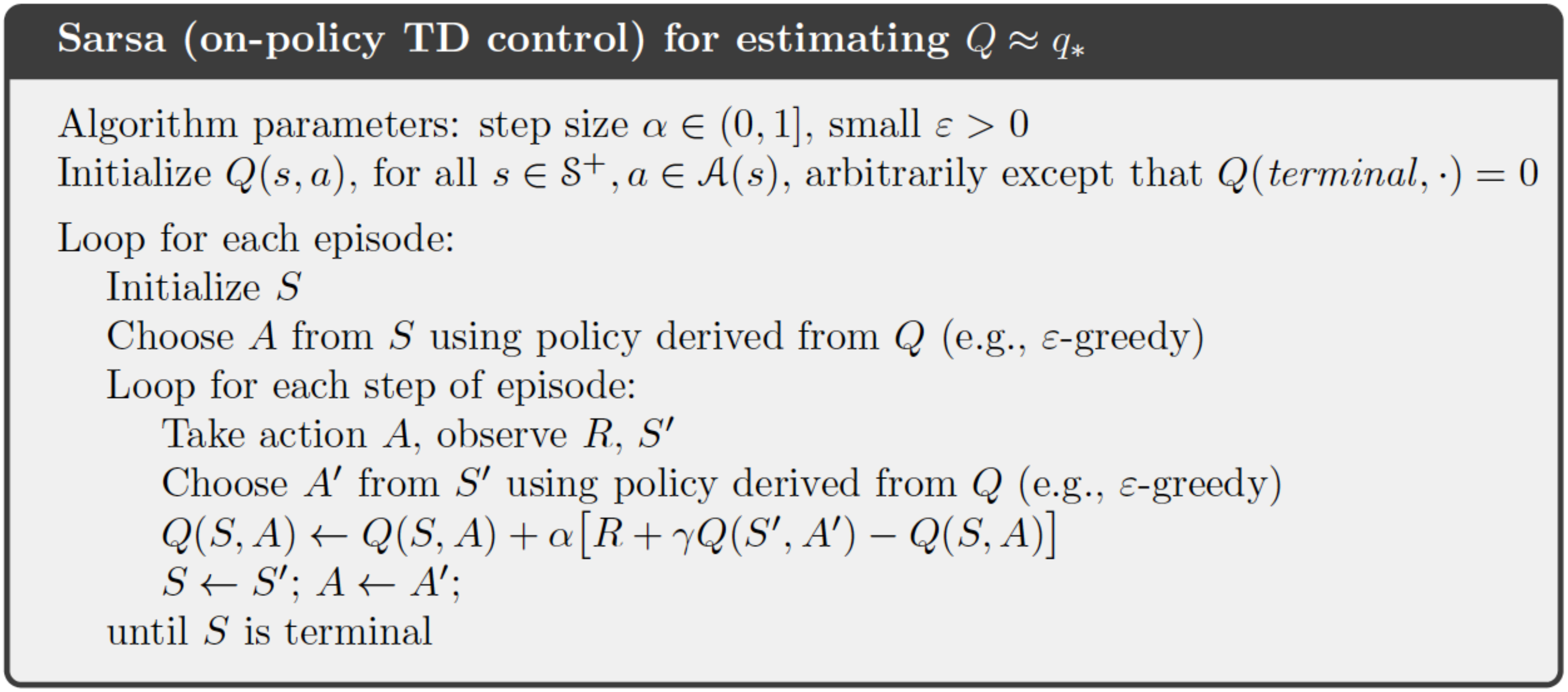
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$$

SARSA

- SARSA iteratively approximates the state-action value function, Q
  - L, SARSA learns the policy and the value function simultaneously

- Keep an estimate of Q(s, a) in a table
  - Update these estimates based on experiences
  - Estimates depend on the exploration policy
  - SARSA is an on-policy method
  - Policy is derived from current value estimates

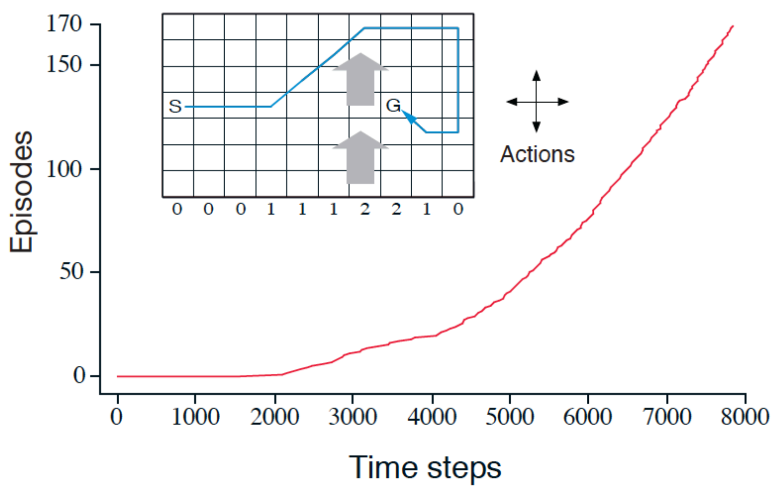
SARSA: The algorithm



Sarsa (on-policy TD control) for estimating $Q \approx q_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

Windy Grid World



- There is wind where there are arrows
- Wind moves you 1 step up at each turn
- This means there are some policies that will never terminate
- Therefore you must learn the policy on-the-fly
- You can’t just try all policies and wait till you see the final result

What if we want to learn the best action, regardless of current policy?

Replace our existing update to the Q table:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$$

....with this:

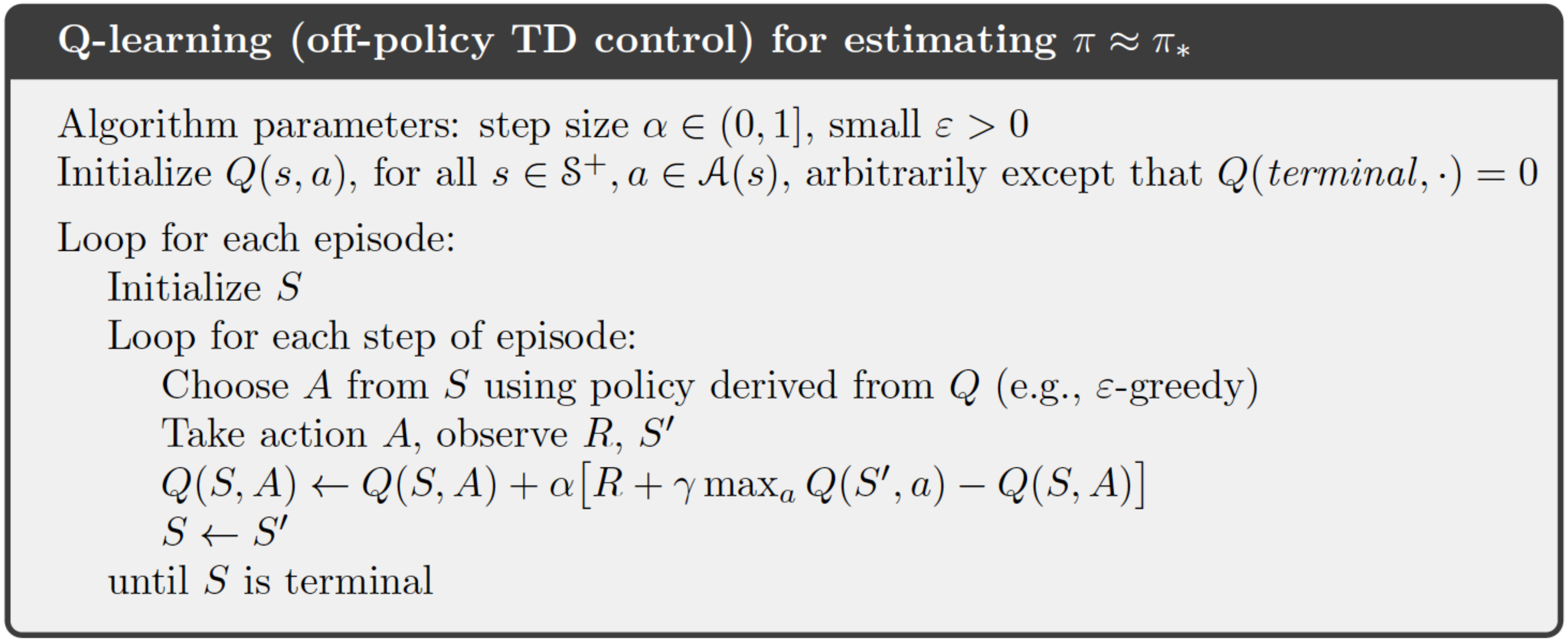
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

Q-Learning

[Watkins & Dayan, 92]

- Q-learning iteratively approximates the state-action value function, Q
  - We won't estimate the MDP directly
  - Learns the value function and policy simultaneously
- Keep an estimate of Q(s, a) in a table
  - Update these estimates as we gather more experience
  - Estimates do not depend on exploration policy
  - Q-learning is an off-policy method

Q-Learning Algorithm



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

$0 \leq \alpha \leq 1$ is the learning rate & we should decay α, just like in TD

Note: this formulation is from Sutton & Barto's "Reinforcement Learning"

Breaking apart that update formula

$Q(s, a) \leftarrow Q(s, a) + \alpha(R + \gamma \max_{a'} Q(s', a') - Q(s, a))$

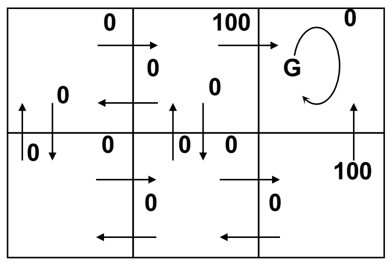
This can be written another way…

$Q(s, a) \leftarrow (1- \alpha)Q(s, a) + \alpha(R + \gamma \max_{a'} Q(s', a'))$

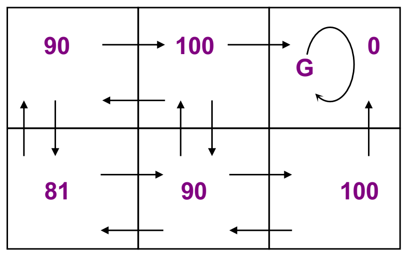
Looked at this way, it is more obvious that α controls whether we value past experience more or new experience more.

Q-learning

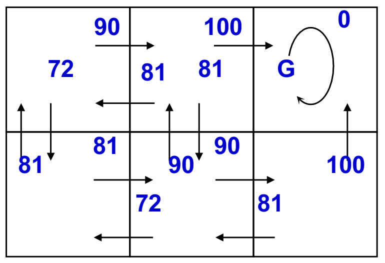
- Q-learning, learns the expected utility of taking a particular action **a** in state **s**



***r*(*state, action*) immediate reward values**



***V**(*state*) values**



***Q*(*state, action*) values**

Convergence Guarantees

- The convergence guarantees for RL are “in the limit”
  - The word “infinite” crops up several times
- Don’t let this put you off
  - Value convergence is different than policy convergence
  - We’re more interested in policy convergence
  - If one action is significantly better than the others, policy convergence will happen relatively quickly

Rewards

- Rewards measure how well the policy is doing
  - Often correspond to events in the world
    - Current load on a machine
    - Reaching the coffee machine
    - Program crashing
  - Everything else gets a 0 reward

These are sparse rewards

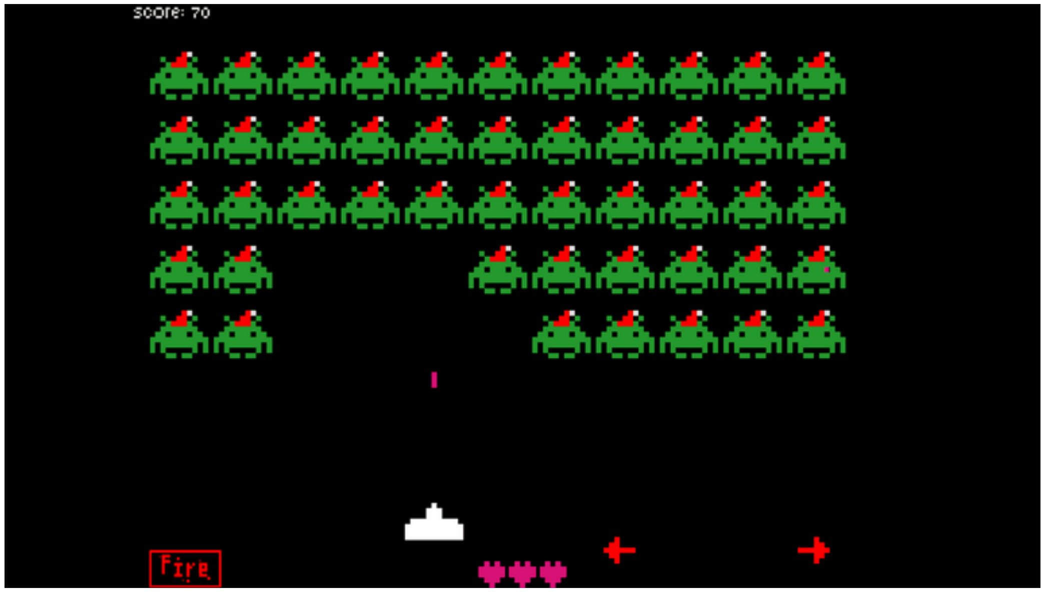
- Things work better if the rewards are incremental
  - For example, distance to goal at each step
  - These reward functions are often hard to design

These are dense rewards

Let’s talk state space & combinatorics

- The idea is to learn a probability distribution over the set of actions possible at each state
- We’ve assumed that there is a table of states and actions
- How big could such a table get?

Playing Video Games



Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

We need to replace the table with….

- …a parameterized function that can output the policy
- Historically, this could be any function
- These days it means…you guessed it…a deep net
- ...and this also means we need differentiable policy gradient

Policy Gradient Methods

- Assume that our policy, p, has a set of n real-valued parameters, q = {q_1, q_2, q_3, ... , q_n }
  - Running the policy with a particular q results in a reward, r_q
  - Estimate the reward gradient, $\frac{\partial R}{\partial \theta_i}$, for each q_i

$$\theta_i \leftarrow \theta_i + \underbrace{\alpha}_{\text{This is another learning rate}} \frac{\partial R}{\partial \theta_i}$$

Policy Gradient Methods

- This results in hill-climbing in policy space
  - So, it's subject to all the problems of hill-climbing
  - But…we can also use tricks from search, like random restarts and momentum terms
- This is a good approach if you have a parameterized policy
  - Typically faster than value-based methods
  - "Safe" exploration, if you have a good policy
  - Learns locally-best parameters *for that policy*

Going to a parameterized Q model

Take the standard Bellman equation for estimating the Q function:

$$Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') \Big| s,a\right] \tag{1}$$

Take a loss function for a parameterized function , where y_i is the target value and θ_i are the parameters :

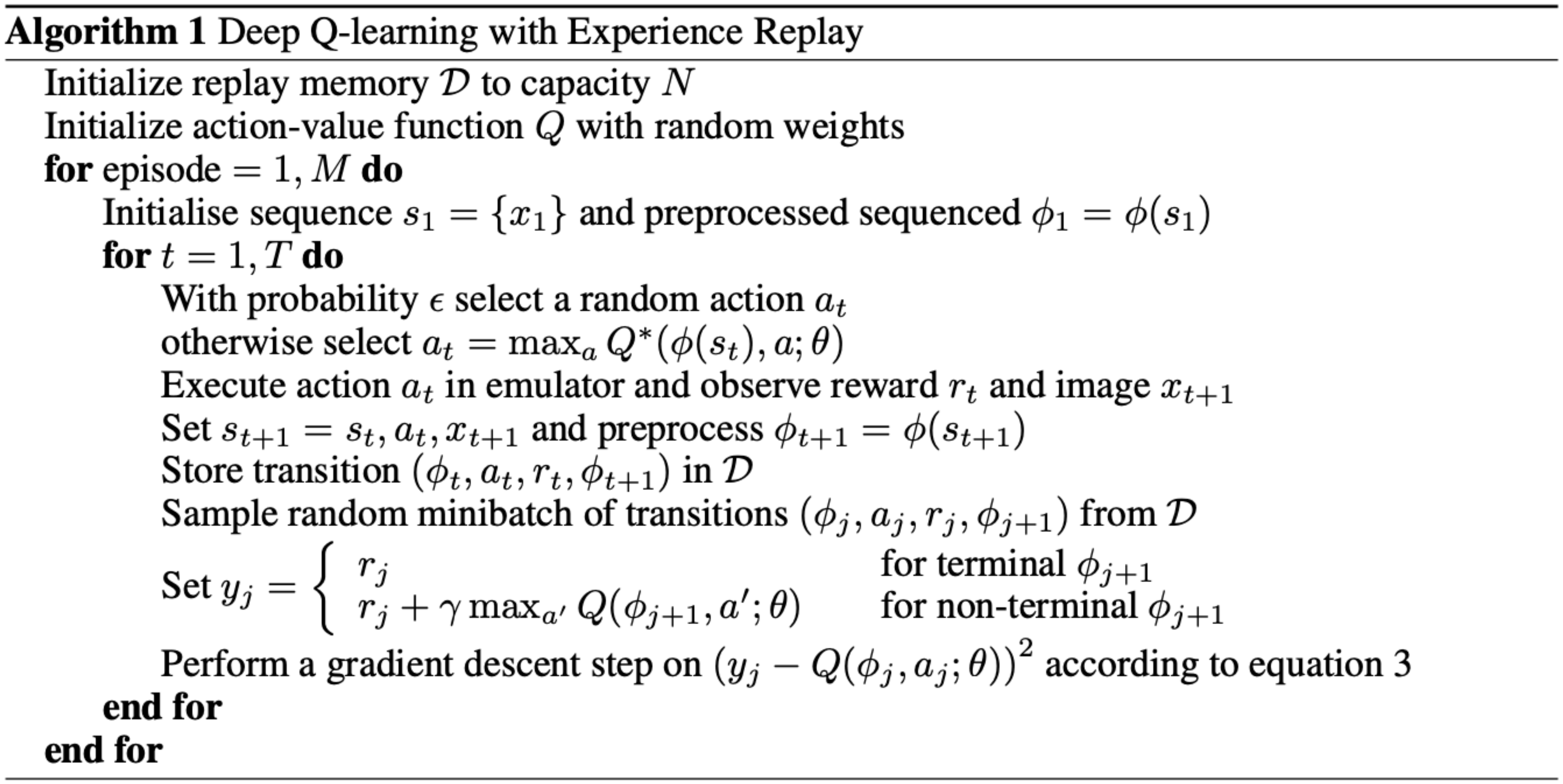
$$L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q\left(s,a;\theta_i\right)\right)^2\right], \tag{2}$$

Take the gradient:

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]. \tag{3}$$

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv:1312.5602*.

Deep Q-learning

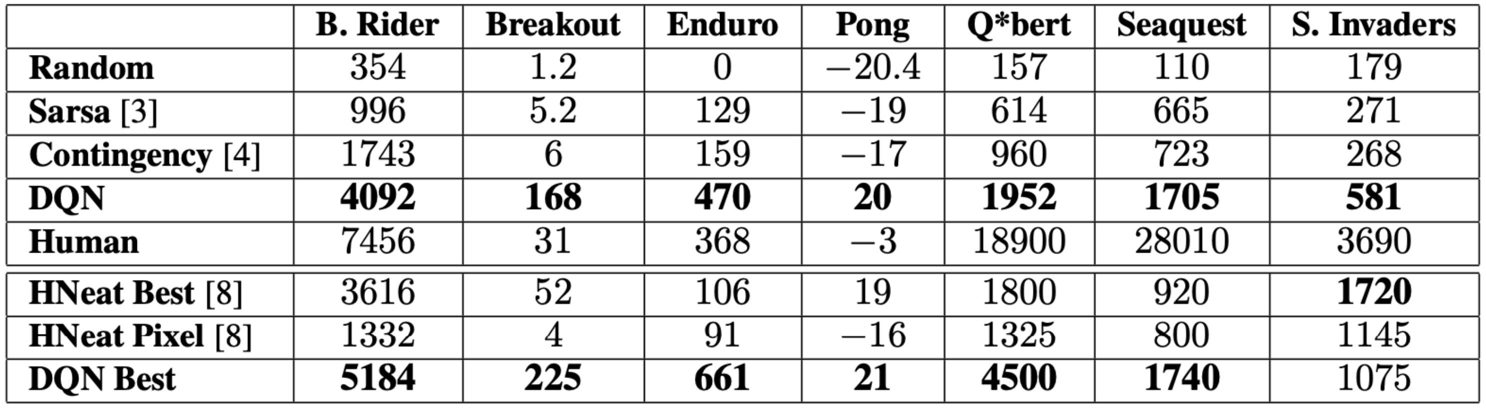


Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    **for** $t = 1, T$ **do**
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    **end for**
**end for**
```

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv:1312.5602*.

This was a breakthrough



	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	-20.4	157	110	179
Sarsa [3]	996	5.2	129	-19	614	665	271
Contingency [4]	1743	6	159	-17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	-3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	-16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv:1312.5602*.

Let’s watch!

https://www.youtube.com/watch?v=V1eYniJ0Rnk

The game of GO

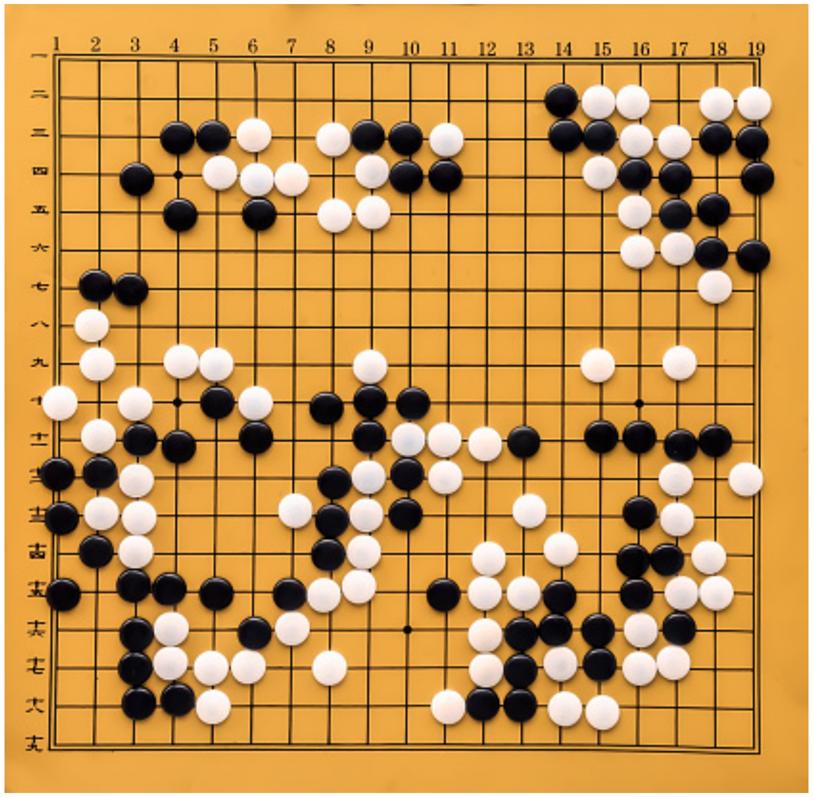


Image Creator: Zozulya | Credit: Getty Images/iStockphoto

- 19 by 19 board
- Each position is either empty, white, or black
- At each turn you can place a stone of your color on any empty position
- How big would the Q table be for this world?

AlphaGO

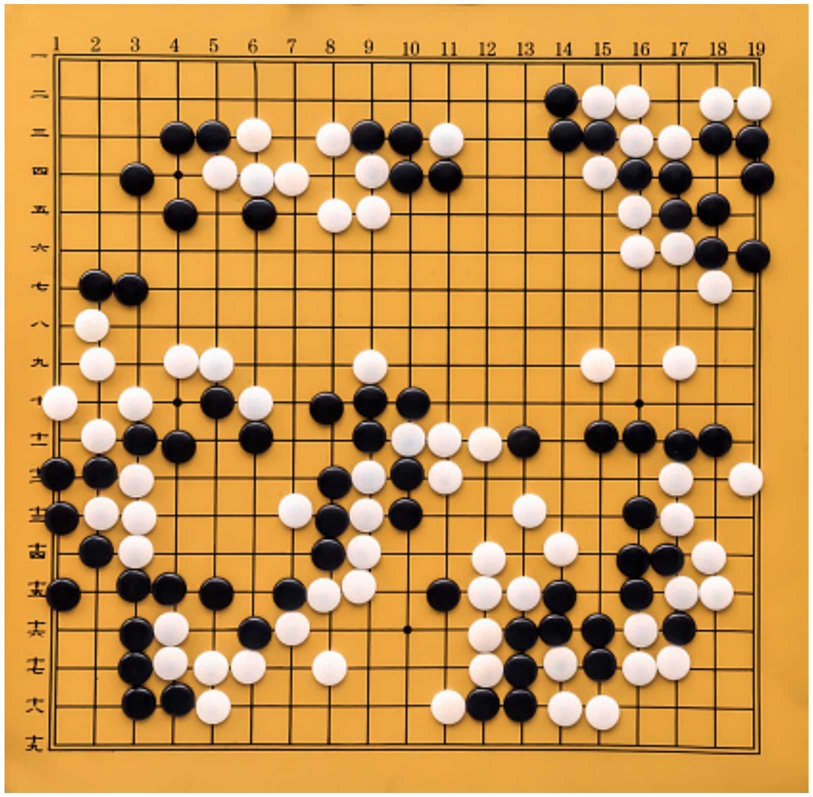
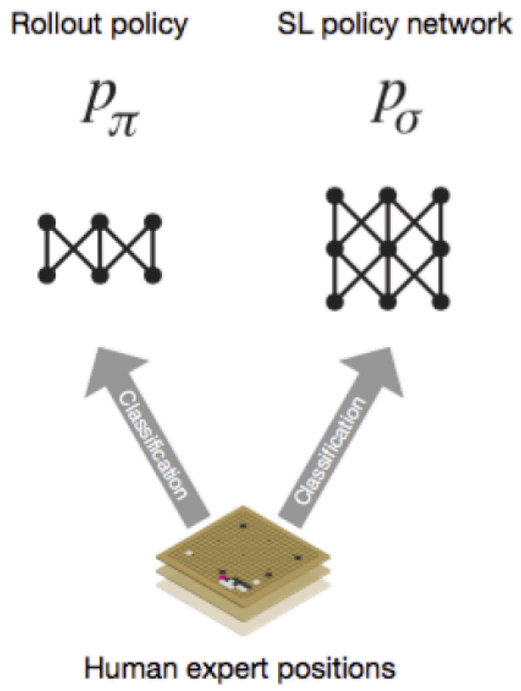


Image Creator: Zozulya | Credit: Getty Images/iStockphoto

- 2015: First program to beat a professional Go master with no handicap
- 2017: Beat Ke Jie, the number one ranked player in the world at the time

Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *nature* 529.7587 (2016): 484-489

The parts of Alpha Go



Train the policy network with supervised learning, based on prior human games

σ are the network weights

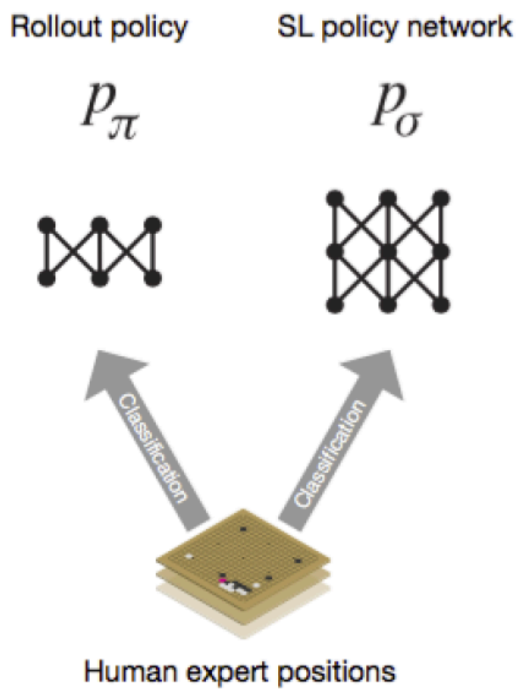
$p_\sigma(a|s)$ is the distribution over the next actions output by the model.

$p_\sigma(a|s)$ is compared to true action y.

After training this predicted the true human move 57% of the time

Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *nature* 529.7587 (2016): 484-489

The parts of Alpha Go



They also trained a small fast “rollout” policy network to predict actions.

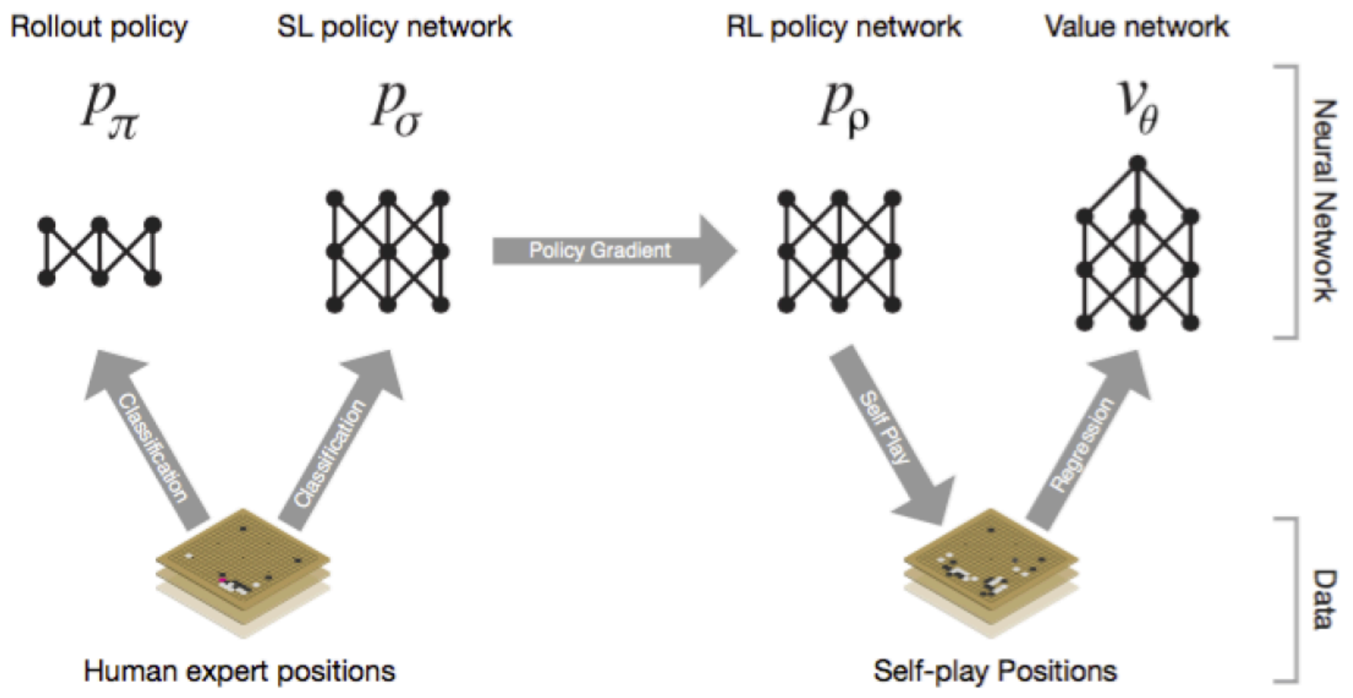
$p_\pi(a|s)$ is the distribution over the next actions output by fast model.

After training this predicted the true human move 24% of the time.

…but it took 2 microseconds to predict the next move, which was useful for rollouts to play out games.

Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *nature* 529.7587 (2016): 484-489

The parts of Alpha Go



Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *nature* 529.7587 (2016): 484-489

The parts of Alpha Go

Once they had a trained policy network, they engaged in self-play.

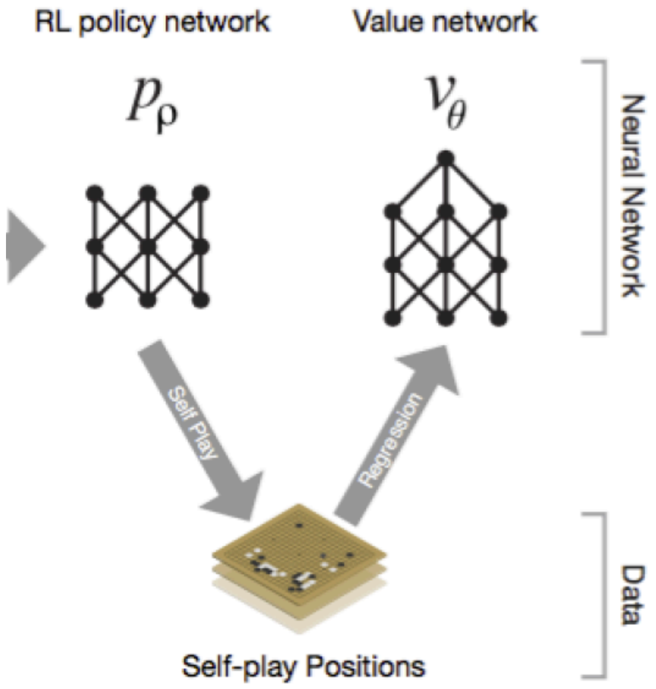
Make 2 copies of the network.

They play.

Apply reinforcement learning to the winner.

Thereafter, always play vs a previous version of the RL network.

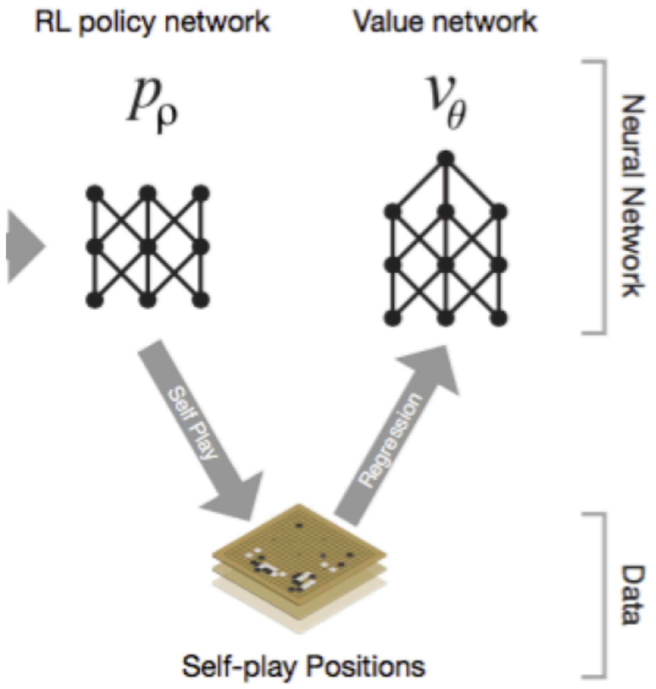
Learn as you go…



Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *nature* 529.7587 (2016): 484-489

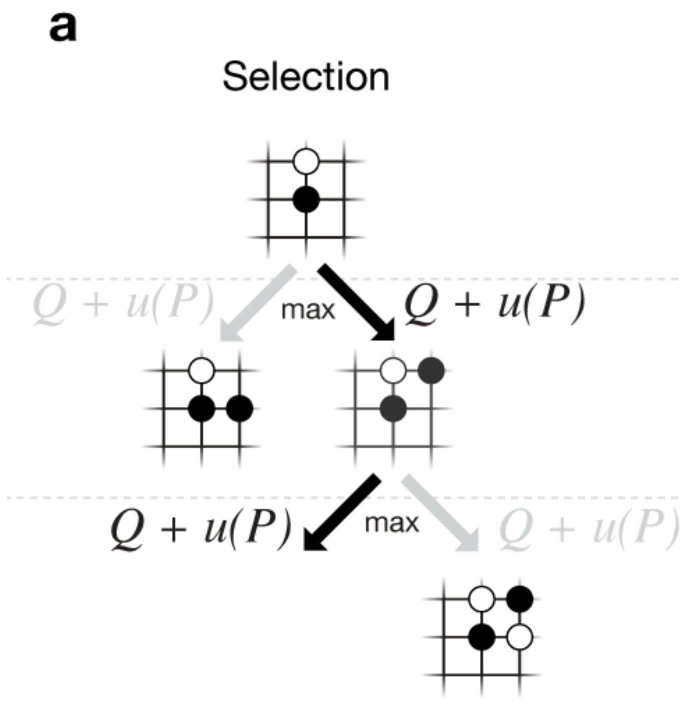
The parts of Alpha Go

Then….they generated 30 million game states from self play and taught a Value network to estimate the value of every one of those states, based on the outcome of the game the state came from.



Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *nature* 529.7587 (2016): 484-489

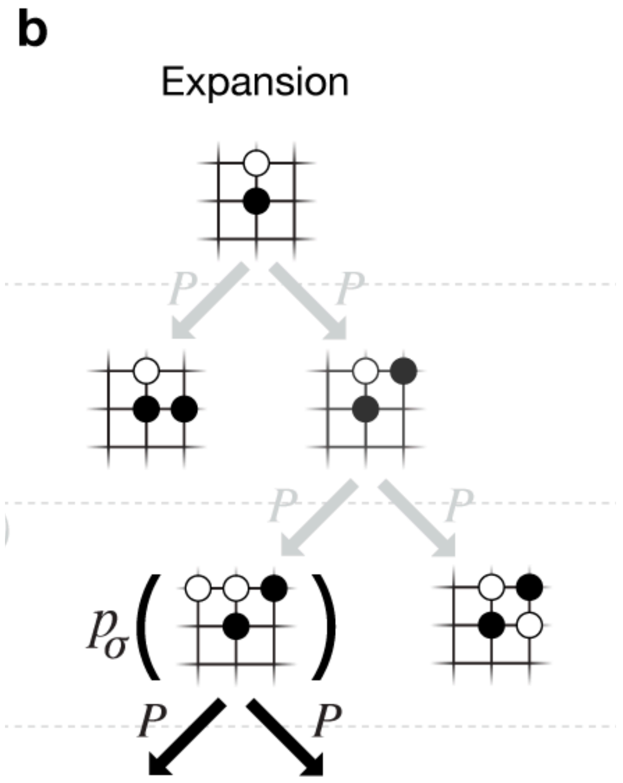
Monte Carlo Tree Search (MCTS)



Select the edge with maximum action-value Q , plus a bonus u(P) that depends on a stored prior probability P for that edge

Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *nature* 529.7587 (2016): 484-489

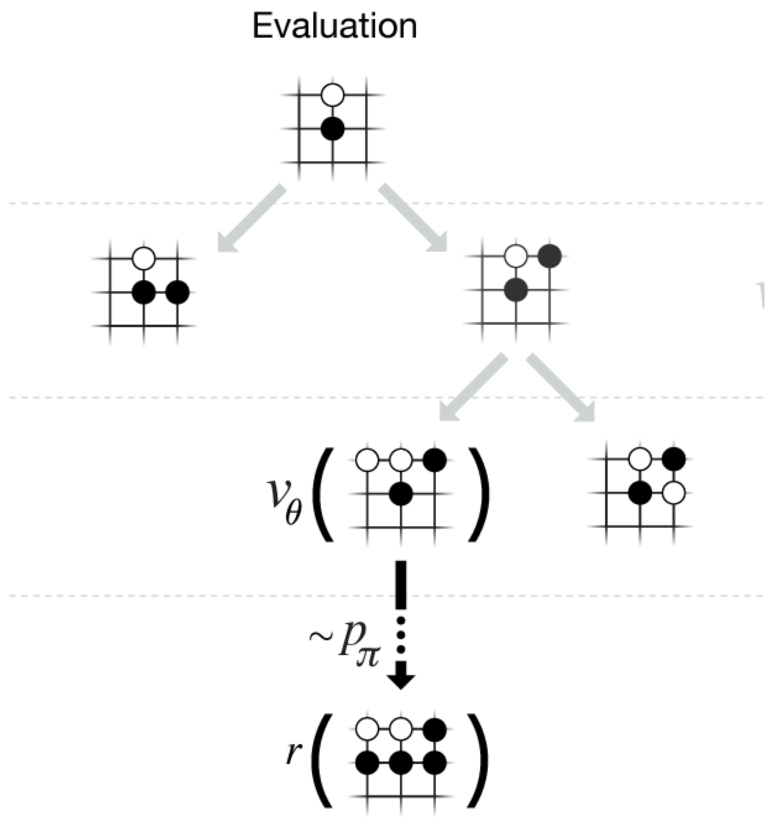
Monte Carlo Tree Search (MCTS)



- The leaf node may be expanded; the new node is processed once by the **supervised** policy network p_σ and the output probabilities are stored as prior probabilities P for each action.

Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *nature* 529.7587 (2016): 484-489

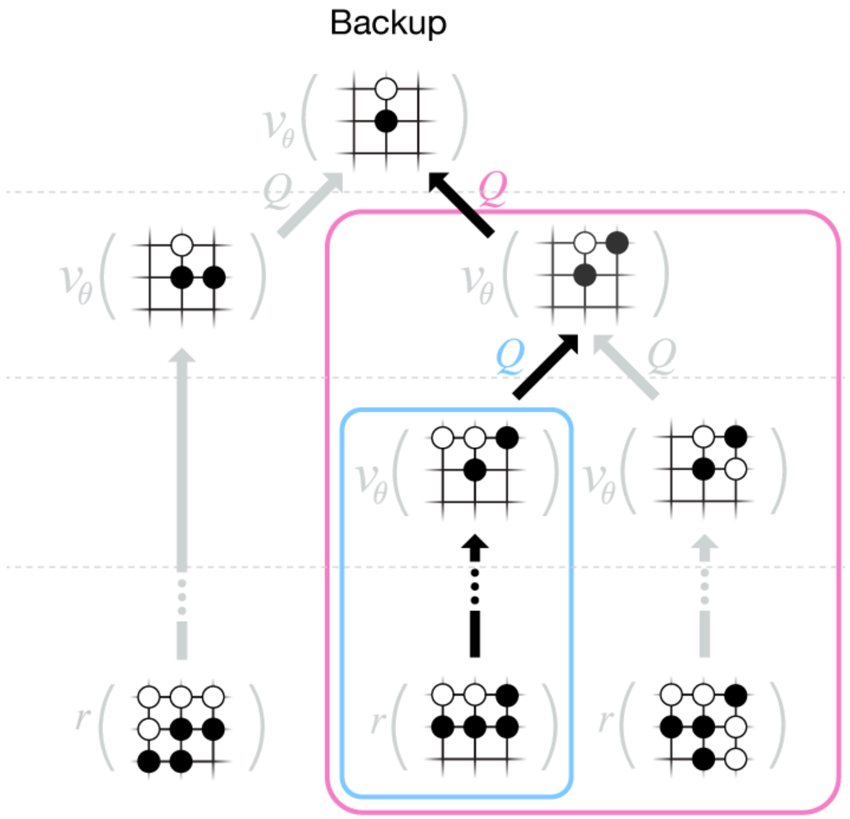
Monte Carlo Tree Search (MCTS)



- At the end of a simulation, the leaf node is evaluated in two ways:
  - using the value network v_θ
  - running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r.

Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *nature* 529.7587 (2016): 484-489

Monte Carlo Tree Search (MCTS)



- Action-values Q are updated to track the mean value of all evaluations r() and v() in the subtree below that action.

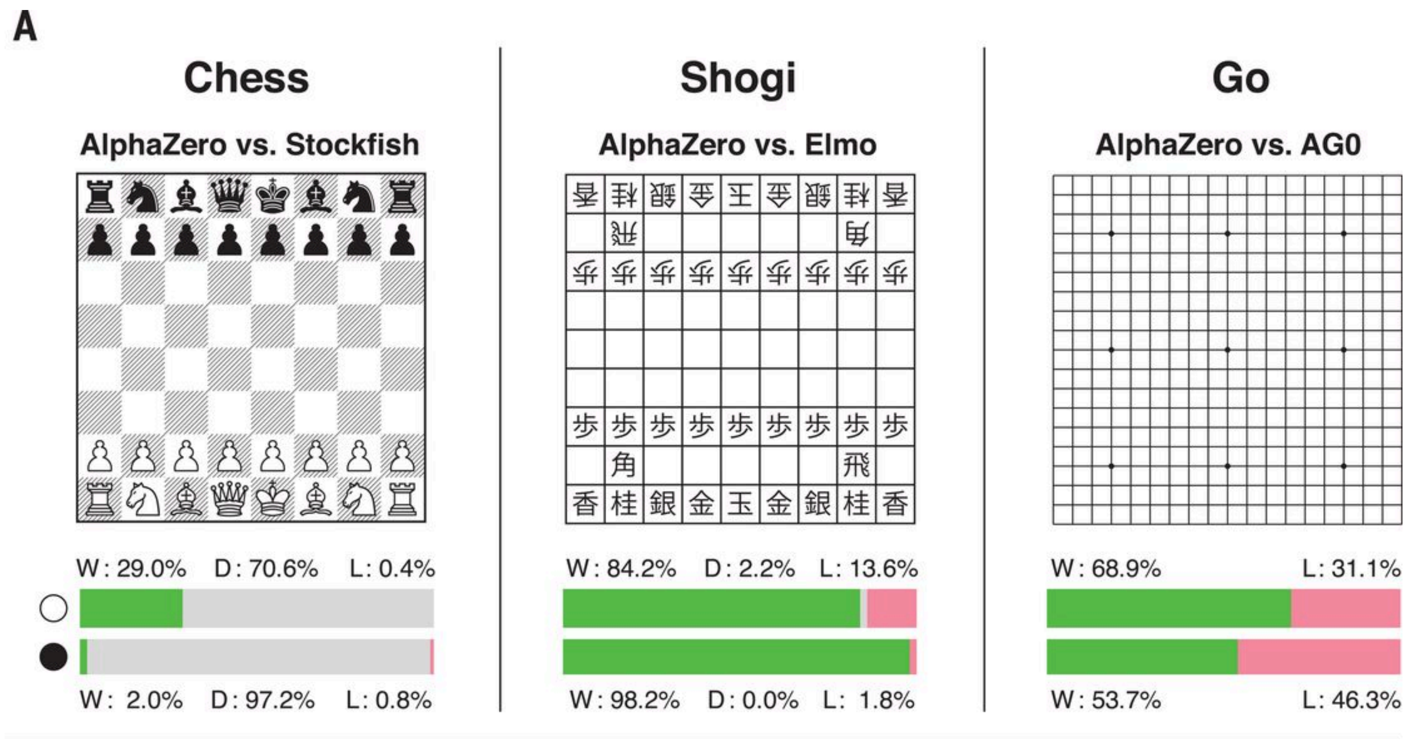
Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *nature* 529.7587 (2016): 484-489

Do they really need to learn from humans?

- Alpha0 jettisoned all the human-trained supervised learning
- It learns exclusively from self play
- How well does that work?

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Lillicrap, T. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, *362*(6419), 1140-1144.

Alpha0 is a general game winner



Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Lillicrap, T. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, *362*(6419), 1140-1144.

Learning to Walk

- RoboCup legged league
  - Walking quickly is a *big* advantage
- Robots have a parameterized gait controller
  - Multiple REAL VALUED parameters
  - Controls step length, height, etc.
- Robots walk across soccer pitch and are timed
  - Reward is a function of the time taken

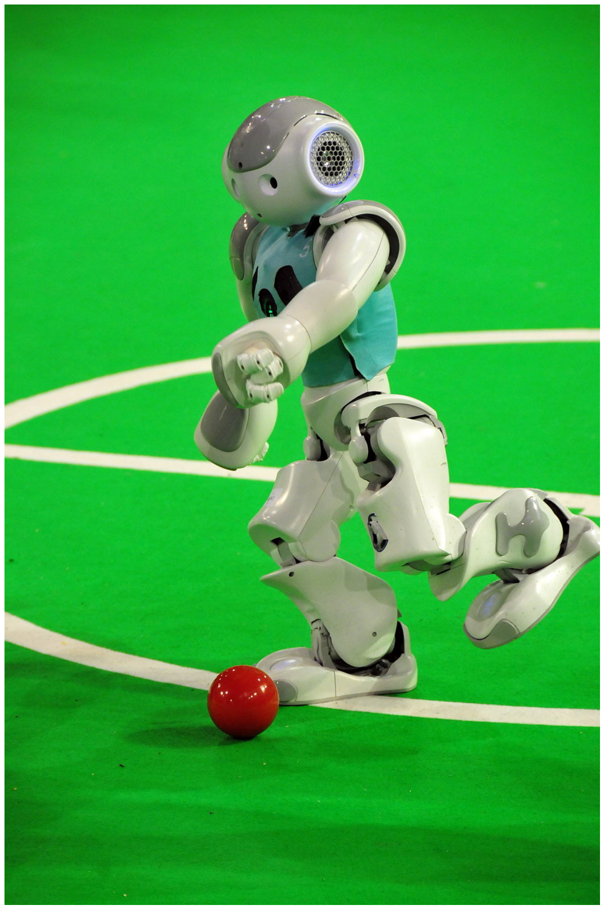


Image courtesy of: Ralf Roletschek
https://www.wikidata.org/wiki/Q15080600