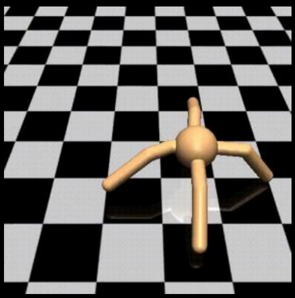
Reinforcement Learning

OpenAI Gym



- OpenAI: Nonprofit founded by Elon Musk (among others) and now largely funded by Microsoft
- OpenAI Gym is an important toolkit allowing researchers to compare RL algorithms and techniques objectively.
- Above we see example of the Mujoco physics simulator. Virtual robots learn to move around.
- https://gym.openai.com

Dota 2

- A Multiplayer Online Battle Arena game, taken on as a challenge by OpenAI
- OpenAI Five uses the Proximal Policy Optimization algorithm
- Crushed pro gaming team OG on April 14, 2019, in 2-0 matchup.
- Humans lasted 40 minutes in the first round, only 20 minutes in the second round.

https://www.youtube.com/watch?v=tfb6aEUMC04

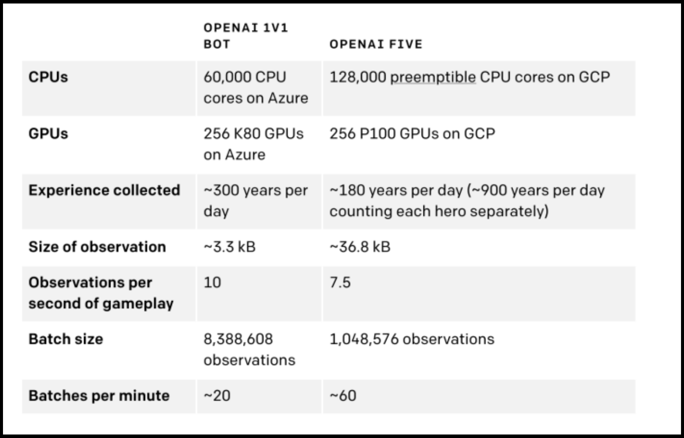
https://openai.com/blog/openai-five-defeats-dota-2-world-champions/

Why is this impressive?

- Dota 2 is a very, very high-dimensional game
- State vector: 20,000 floating point numbers
  - 70 for chess and 400 for Go
- 170,000 possible actions per step
  - Avg. of 35 in chess, 150 in Go
- 80,000 steps per game on average
  - Avg. 40 steps in Chess, 150 in Go

https://openai.com/blog/openai-five/

Technical Details



	OPENAI 1V1 BOT	OPENAI FIVE
CPUs	60,000 CPU cores on Azure	128,000 preemptible CPU cores on GCP
GPUs	256 K80 GPUs on Azure	256 P100 GPUs on GCP
Experience collected	~300 years per day	~180 years per day (~900 years per day counting each hero separately)
Size of observation	~3.3 kB	~36.8 kB
Observations per second of gameplay	10	7.5
Batch size	8,388,608 observations	1,048,576 observations
Batches per minute	~20	~60

- Champion agent has equivalent to 45,000 years of human play!

Atari Games

- An important benchmark in RL research was learning how to play Atari games using nothing but pixel inputs.
- Notable advance is the Deep Q-Network algorithm by DeepMind (a British research lab acquired by Google.)
- You will use this for PA3!
- https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

Atari Games

https://www.youtube.com/watch?v=tfb6aEUMC04

IRL Reinforcement

- Various RL techniques demonstrated on self-made real "cartpole," presented at CeBIT 2018.

https://www.youtube.com/watch?v=0wSAGwgzrp8

IRL Reinforcement

- Autonomous car by Wayve.ai

https://www.youtube.com/watch?v=eRwTbRtnT1I

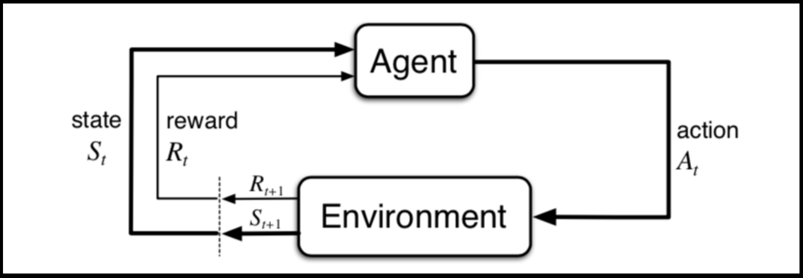
Commercial Uses

- Many theoretical advances in RL have come from game-playing
- But RL is already being applied in various fields:
  - Industrial robotics
  - Supply chain optimization
  - Advertising and recommendation engines
  - Self-driving cars (in combination with other techniques)

RL Characteristics

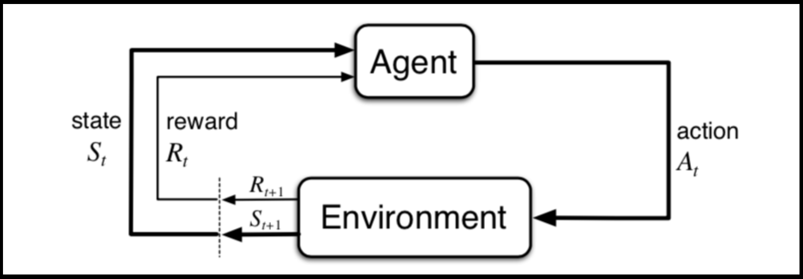
- Involves a complete, goal-seeking ***agent*** acting within an ***environment***
- The "goal" of the agent is typically to maximize reward over time
- Assumes significant uncertainty about the environment.
- Involves a tradeoff between exploration and exploitation
  - We wish explore the space of possible states to find actions that lead to high rewards. However, this will lead to mistakes!

Markov Decision Process



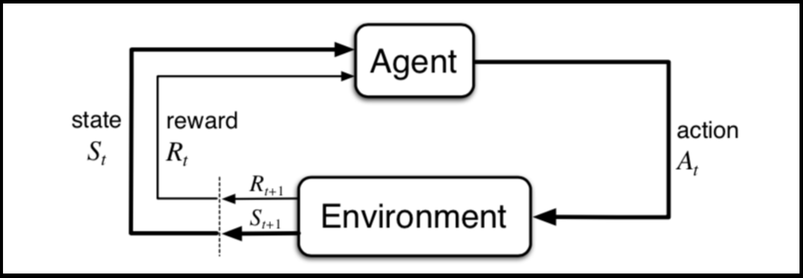
- **Agent**: The learner or decision-maker to be trained.
- **Environment**: Everything outside the agent.
- **Time**: Sequence of discrete time steps, $t = 0, 1, 2, 3 \ldots T$
- **State**: A "situation" that the agent may find itself in, $S_t \in \mathcal{S}$
- **Action**: Selected by the agent at a time step, $A_t \in \mathcal{A}$

Markov Decision Process



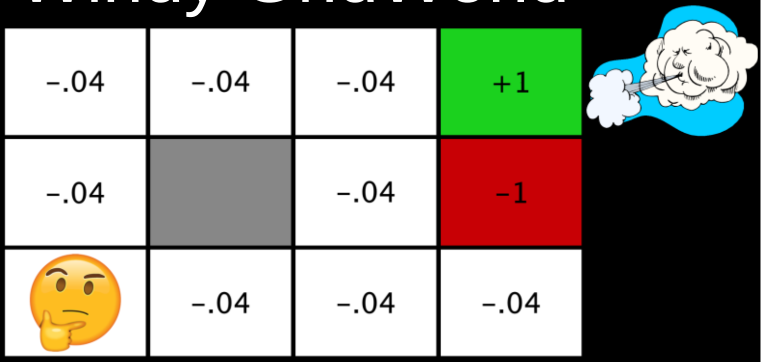
- After each action, the environment returns:
  - A **reward**: a numerical value, $R_{t+1} \in \mathcal{R} \subseteq \mathbb{R}$
  - A new state, $S_{t+1} \in \mathcal{S}$
- Interaction over time leads to a *trajectory* of the following form:
  - $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3 \ldots$

Markov Property



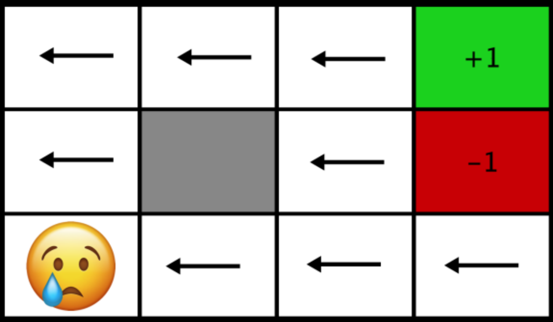
- The **Markov Property**: Each possible value for S_t and R_t depend *only* on the immediately preceding state and action, S_{t-1}, R_{t-1}, and not on any other earlier states or actions.
- “Given the present, the future does not depend on the past”
- MDP Dynamics:
  - $p(s', r \mid s, a) = \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$

Windy GridWorld



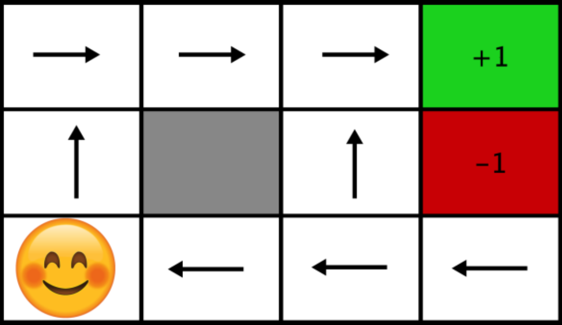
- $\mathcal{A}$ = {up, left, down right}
- 0.8 chance of making chosen action
- 0.1, 0.1 chance of making a right-angle move instead
- -0.4 reward until the agent reaches the goal.

Policies



- A **policy, π,** is a function that takes a state and returns the probabilities of performing each a ∈ $\mathcal{A}$.
- Above is a suboptimal policy: π(s) = [0, 1, 0, 0] for all s ∈ $\mathcal{S}$ with $\mathcal{A}$ = {up, left, down right}
- The optimal policy, **π***, is the policy that maximizes expected return.

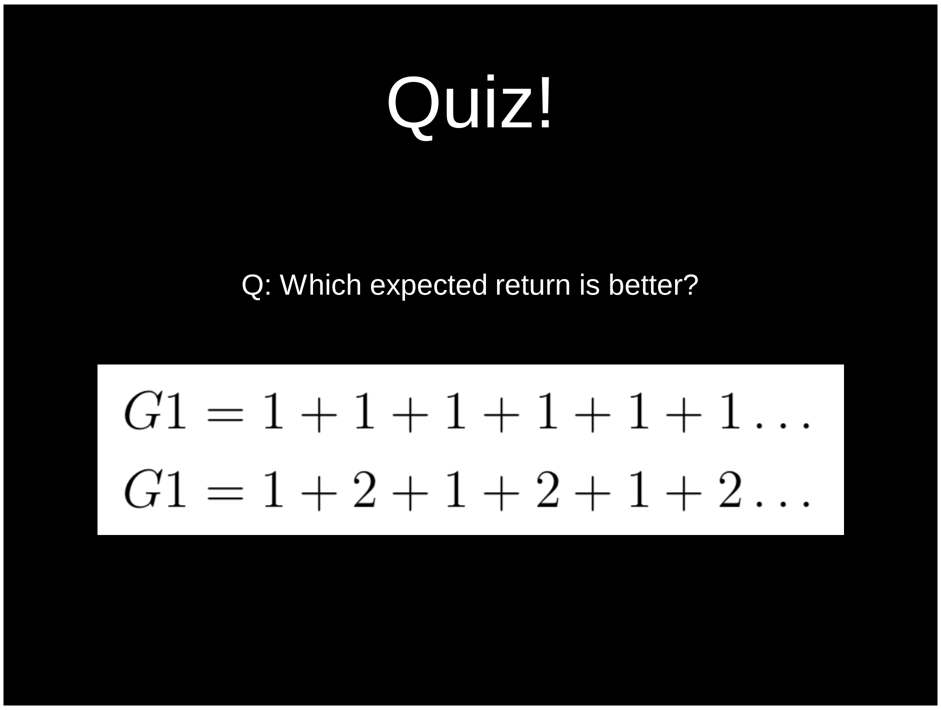
Policies



- These arrows show the optimal policy, **π***
- The agent prefers taking the “long way around,” to avoid the chance of falling into the bad terminal state.

Expected Return

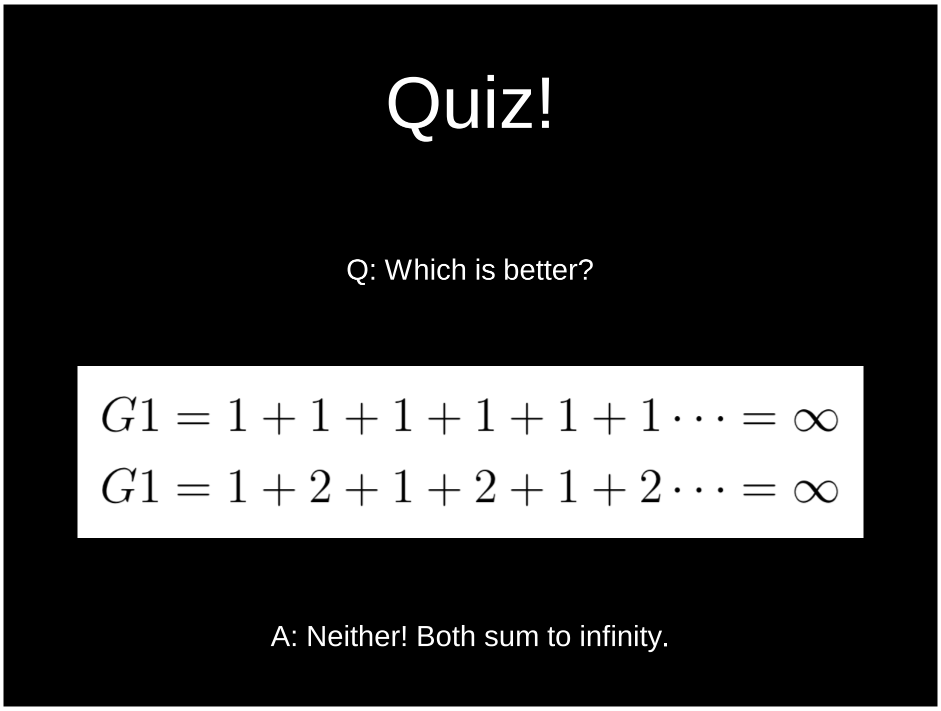
- We wish to maximize the **expected return**, which *could* be denoted as the sum of rewards across a trajectory:
  - $G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$
- **Episodic tasks**: May end in a special terminal state, and reset to a standard starting state
- **Continuing tasks**: No terminal state, i.e., $T = \infty$



Quiz!

Q: Which expected return is better?

$$G1 = 1 + 1 + 1 + 1 + 1 + 1 \ldots$$
$$G1 = 1 + 2 + 1 + 2 + 1 + 2 \ldots$$



Quiz!

Q: Which is better?

$$G1 = 1 + 1 + 1 + 1 + 1 + 1 \cdots = \infty$$
$$G1 = 1 + 2 + 1 + 2 + 1 + 2 \cdots = \infty$$

A: Neither! Both sum to infinity.

Discount Rates

$$U(S_0, S_1, S_2, \dots) = \sum_{t=0}^{\infty} \gamma^t R(S_t), \text{ where } 0 \le \gamma \le 1$$
$$\le \sum_{t=0}^{\infty} \gamma^t R_{\max} = \frac{R_{\max}}{1-\gamma}$$

- To fix this, we introduce the discount rate, γ.
  - Turns the sequence of rewards into a geometric series
  - Guarantees a bounded, finite result
  - Rewards that happen later matter less than more recent rewards
  - Analogous to “long-term” vs “short-term” thinking

Discount Rates

$$\gamma = 0.9$$

$$\begin{aligned} G1 &= 1\gamma^0 + 1\gamma^1 + 1\gamma^2 + 1\gamma^3 + 1\gamma^4 + 1\gamma^5 \dots \\ &= 1(.9)^0 + 1(.9)^1 + 1(.9)^2 + 1(.9)^3 + 1(.9)^4 + 1(.9)^5 \dots \\ &= 1 + .9 + .81 + .729 + .6561 + .5905 \dots \\ &\approx 10 \end{aligned}$$

$$\begin{aligned} G2 &= 1\gamma^0 + 1\gamma^1 + 2\gamma^2 + 1\gamma^3 + 1\gamma^4 + 2\gamma^5 \dots \\ &= 1(.9)^0 + 1(.9)^1 + 2(.9)^2 + 1(.9)^3 + 1(.9)^4 + 2(.9)^5 \dots \\ &= 1 + .9 + 1.62 + .729 + .6561 + 1.1809 \dots \\ &\approx 13 \end{aligned}$$

$$G1 < G2$$

Value Function

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S},$$

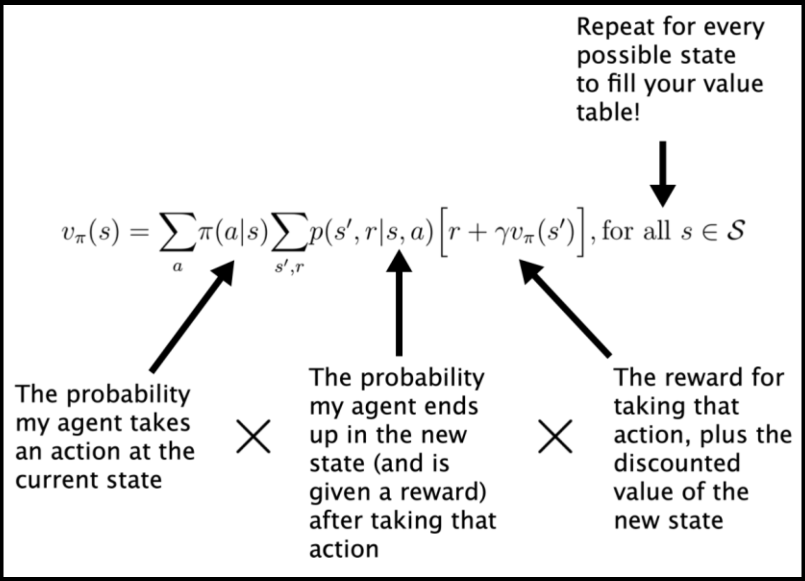
- **Value Function**: A function of "how good" it is for an agent to be in a certain state when following a given policy.
- Equivalent to finding the discounted expected return from that state.
- The table made by your "explore" in MP2 is an example of a value function! It could be filled in using techniques shown here.

Bellman Equation

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big], \text{for all } s \in \mathcal{S}$$

- **Bellman Equation:** Expresses the relationship between the value of a state and values of its successor states.
- Averages over all the possibilities, weighting each by its probability of occurring.
- The value of the start state equals the (discounted) value of the expected next state, plus the reward expected along the way.

Bellman Equation



Repeat for every possible state to fill your value table!

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big], \text{ for all } s \in \mathcal{S}$$

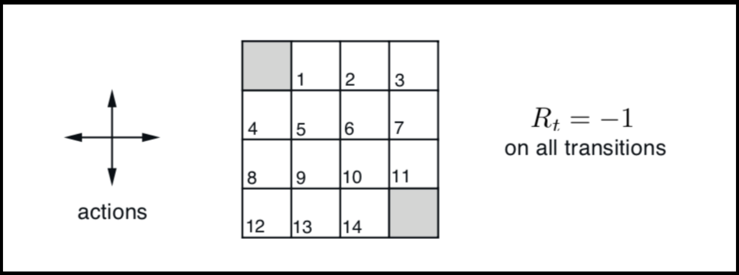
The probability my agent takes an action at the current state $\times$ The probability my agent ends up in the new state (and is given a reward) after taking that action $\times$ The reward for taking that action, plus the discounted value of the new state

Optimality Equation

$$v_*(s) = \max_a \sum_{s',r} p(s', r, |s, a)\Big[r + \gamma v_*(s')\Big]$$

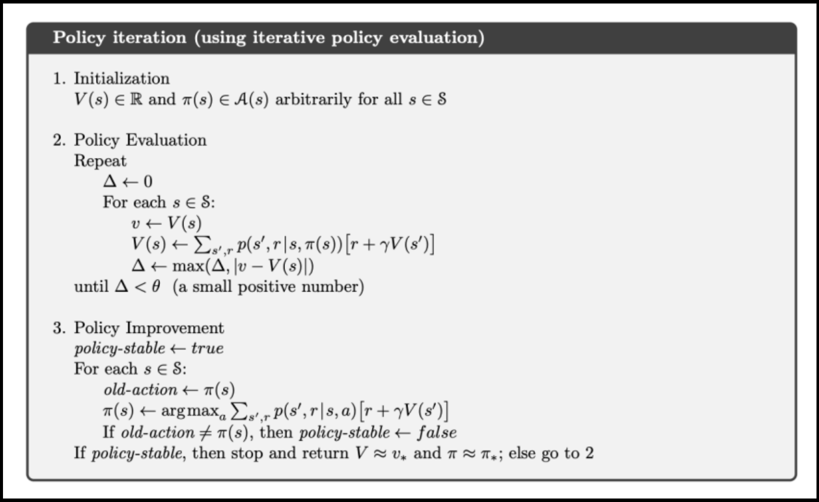
- The value of a state under an optimal policy must equal the expected return for the best action taken from that state.
- (A policy that chooses bad actions cannot be optimal.)

Solving Gridworlds



- Given the GridWorld above, assume an initial policy where in each state {1 … 14}, the agent makes a move at random.
- The terminal states are in the top left and bottom right
- How do we improve this policy?

Policy Iteration



Policy iteration (using iterative policy evaluation)

```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
       $\Delta \leftarrow 0$
       For each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$  (a small positive number)

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
       $old\text{-}action \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
       If $old\text{-}action \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

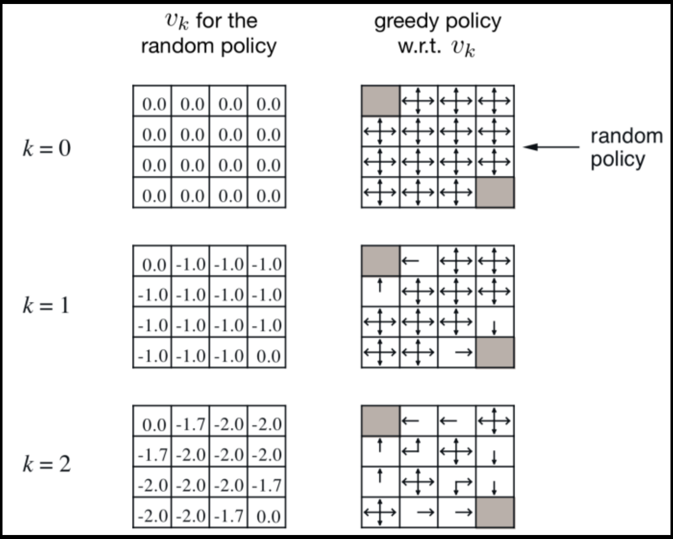
Policy Iteration

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

Follows a cycle of evaluation and improvement

1. Following your existing policy, select the “best” action for each state greedily.
2. Update the previous policy with those state-action pairs.
3. Repeat until nothing changes

Policy Iteration



$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

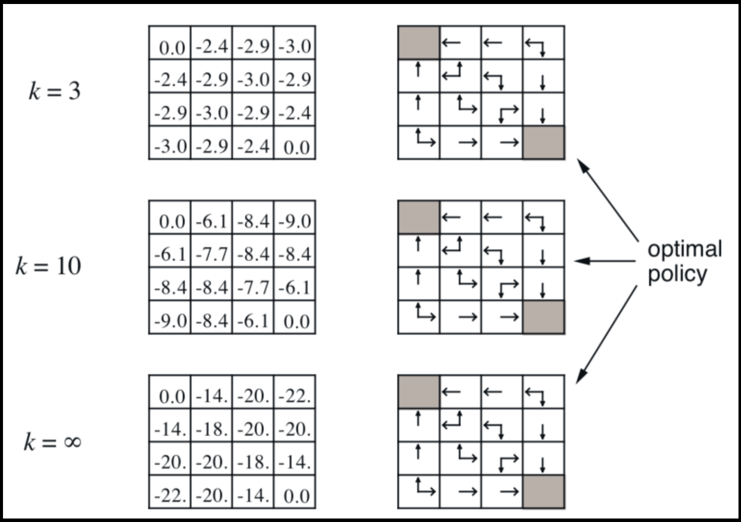
$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

Policy Iteration



$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

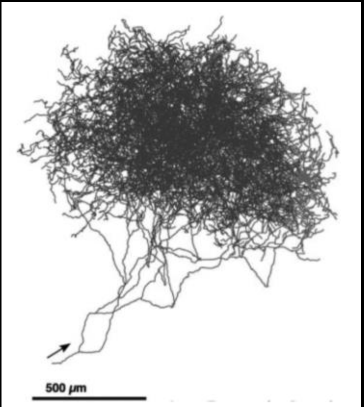
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Problems

- In theory, we can solve almost any MDP using a Bellman algorithm. That's amazing! However, we need:
  - Complete knowledge of the environment dynamics
  - Space to store the entire value function in a tabular format
- As the number of states increase, classical Bellman iteration become increasingly costly to compute
- Could we do this with chess? Or Dota 2?

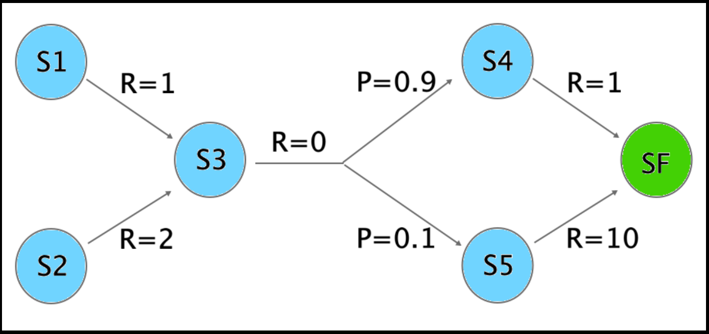
Temporal Difference

- Temporal Difference is a method of learning how to predict future rewards, and choose better actions, over time in an MDP.
- TD does not assume full knowledge of the MDP's dynamics. The agent improves its policy over time using data gathered by interacting with the environment.
- Correspondences have been found between TD and the action of the neurotransmitter dopamine.



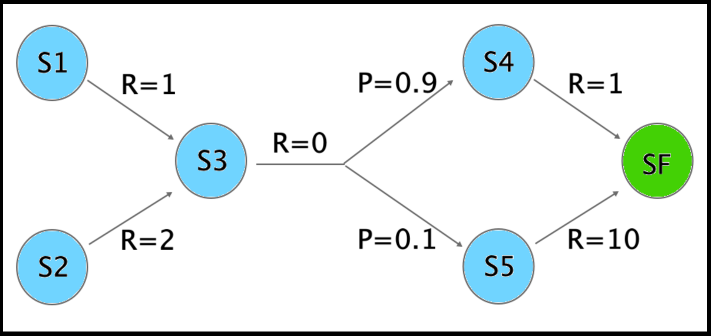
Axonal arbor of a single neuron producing dopamine as a neurotransmitter. These axons make synaptic contacts with a huge number of dendrites of neurons in targeted brain areas.

Value Computation



- Consider the simple system above.
- Quiz! What is the value of S3?

Value Computation



- Quiz! What is the value of S3? (A: 1.9)

Estimating from Data

1. $S_1 \xrightarrow{+1} S_3 \xrightarrow{+0} S_4 \xrightarrow{+1} S_F = +2$

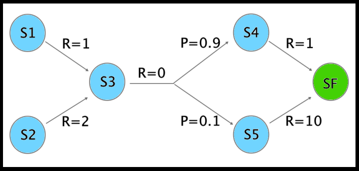
2. $S_1 \xrightarrow{+1} S_3 \xrightarrow{+0} S_5 \xrightarrow{+10} S_F = +11$

3. $S_1 \xrightarrow{+1} S_3 \xrightarrow{+0} S_4 \xrightarrow{+1} S_F = +2$

4. $S_1 \xrightarrow{+1} S_3 \xrightarrow{+0} S_4 \xrightarrow{+1} S_F = +2$

5. $S_2 \xrightarrow{+2} S_3 \xrightarrow{+0} S_5 \xrightarrow{+10} S_F = +12$

True value of $S_1 = 2.9$
After three episodes: $V(S_1) = \frac{15}{3} = 5$
After four episodes: $V(S_1) = \frac{17}{4} = 4.25$



What if we don’t know the dynamics beforehand?

Run episodes and gather samples

The approach to the left shows how we might estimate the value of S1 using data.

Learning Rates

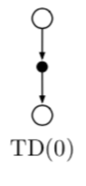
True value of $S_1 = 2.9$
After three episodes: $V(S_1) = \frac{15}{3} = 5$
After four episodes: $V(S_1) = \frac{17}{4} = 4.25$

$$\begin{aligned} V_T(S_1) &= \frac{(T-1)V_{T-1}(S_1) + R_T(S_1)}{T} \\ &= \frac{(T-1)}{T} V_{T-1}(S_1) + \frac{1}{T} R_T(S1) \\ &= V_{T-1}(S_1) + \alpha_T (R_T(S_1) - V_{T-1}(S_1)) \\ &\text{where } \alpha_T = \frac{1}{T} \end{aligned}$$

- The learning rate, α, reflects this averaging approach to estimation.
- α decreases as the number of episodes increases
- Proven to converge (slowly). In practice, usually set to a constant value 0 < α <= 1

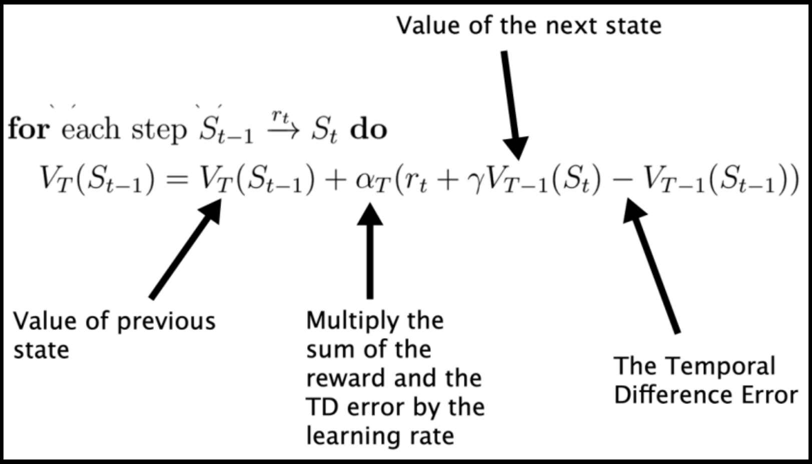
TD(0)

```
Algorithm $TD(0)$ for estimating $v_\pi$
  Input: a policy $\pi$
  Algorithm parameter: learning rate $\alpha \in (0, 1]$
  Initialize $V_0(s) = 0$ for all $s \in \mathcal{S}$
  Generate episodes following $\pi$
  **for** each episode $T$ **do**
    $V_T(s) = V_{T-1}(s)$ for all $s \in \mathcal{S}$
    **for** each step $S_{t-1} \xrightarrow{r_t} S_t$ **do**
      $V_T(S_{t-1}) = V_T(S_{t-1}) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))$
  **return** $V_T$
```



- Makes a sample update of a single state, using the temporal difference at each time step.

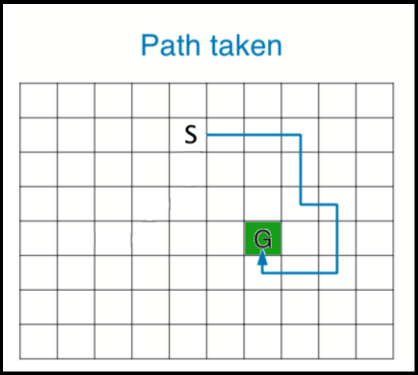
TD(0)


for each step $S_{t-1} \xrightarrow{r_t} S_t$ **do**

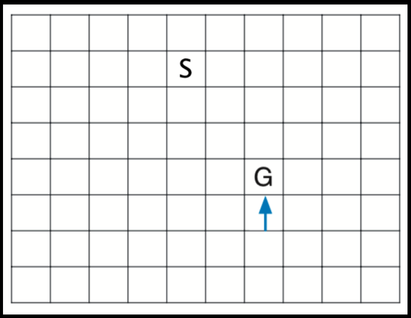
$$V_T(S_{t-1}) = V_T(S_{t-1}) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))$$

Value of the next state
Value of previous state
Multiply the sum of the reward and the TD error by the learning rate
The Temporal Difference Error

TD(0) - GridWorld

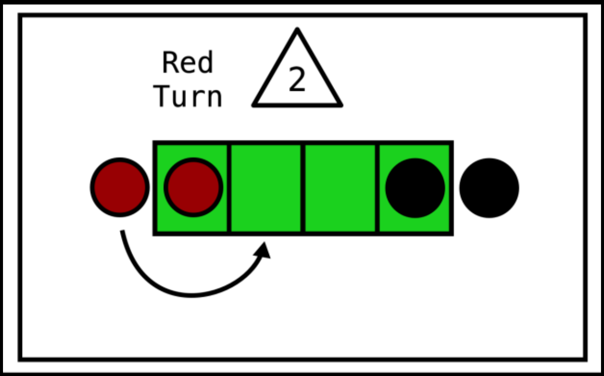


Update after 1 episode of TD(0)



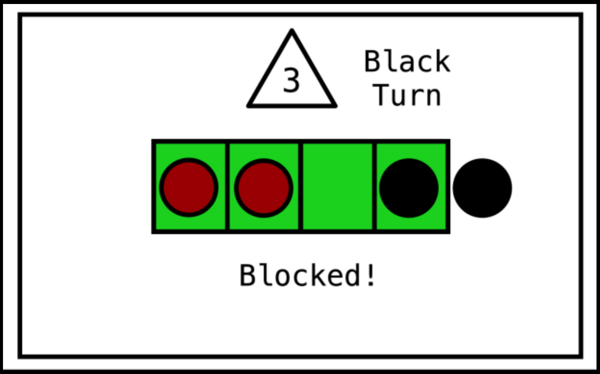
- Reward = 1 at goal. All values initialized at 0.
- Only the very last state is updated!

Mini-Nannon



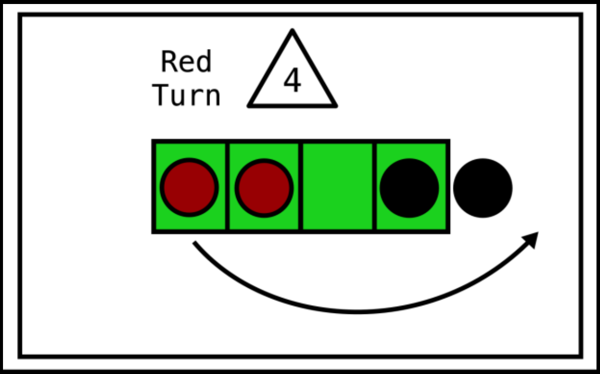
- Beginning of a {2, 4, 4} game of Nannon. Red goes first and has rolled a 2.

Mini-Nannon



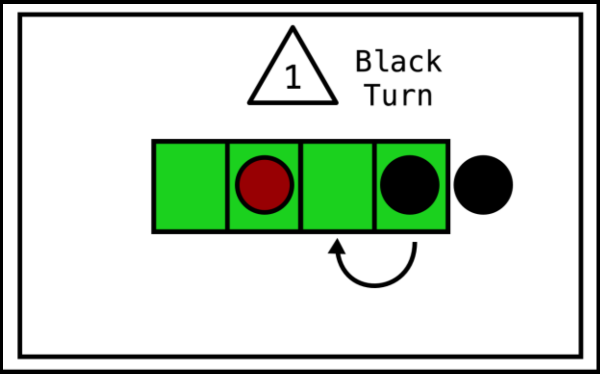
- Black rolls a 3 and is blocked.

Mini-Nannon



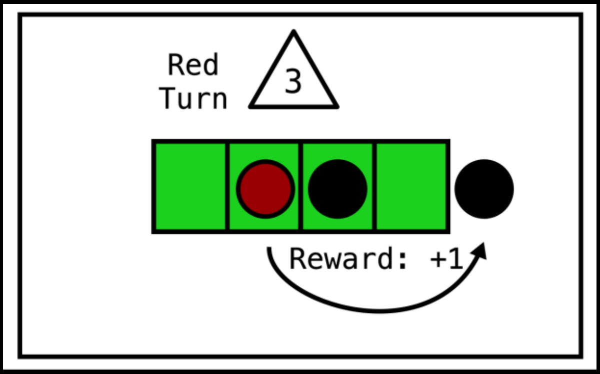
- Red rolls a 4 and moves its rear piece to goal.

Mini-Nannon



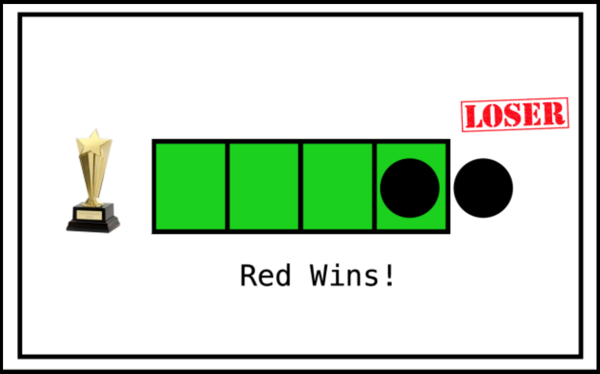
- Black rolls a 1 and must move its front piece.

Mini-Nannon



- Red rolls a 3 and moves its remaining piece to goal.
- Winning the game returns a reward of 1.

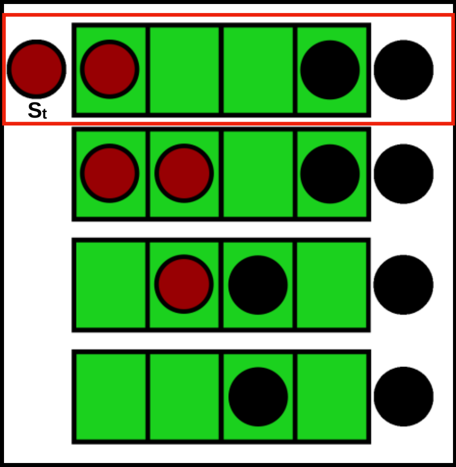
Mini-Nannon



```
$V_T(s) = V_{T-1}(s)$ for all $s \in \mathcal{S}$
**for** each step $S_{t-1} \xrightarrow{r_t} S_t$ **do**
    $V_T(S_{t-1}) = V_T(S_{t-1}) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))$
```

TD(0)

γ = 0.9, α = 1.0

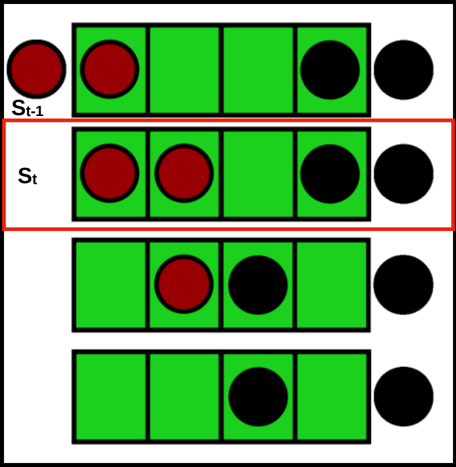


- V(s_1) = 0.0
- V(s_2) = 0.0
- V(s_3) = 0.0
- V(s_T) = 0.0

```
$V_T(s) = V_{T-1}(s)$ for all $s \in \mathcal{S}$
**for** each step $S_{t-1} \xrightarrow{r_t} S_t$ **do**
    $V_T(S_{t-1}) = V_T(S_{t-1}) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))$
```

TD(0)

γ = 0.9, α = 1.0

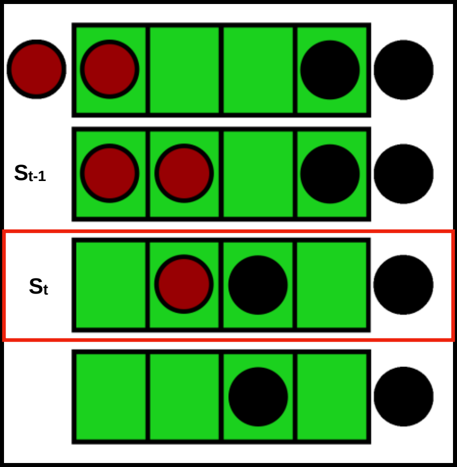


- V(s_1) = 0.0 + 0.0 = 0.0
- V(s_2) = 0.0
- V(s_3) = 0.0
- V(s_T) = 0.0

```
$V_T(s) = V_{T-1}(s)$ for all $s \in \mathcal{S}$
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $V_T(S_{t-1}) = V_T(S_{t-1}) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))$
```

TD(0)

γ = 0.9, α = 1.0

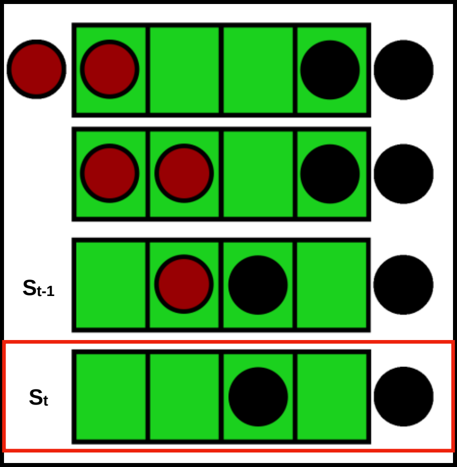


- $V(s_1)$ = 0.0
- $V(s_2)$ = 0.0 + 0.0 = 0.0
- $V(s_3)$ = 0.0
- $V(s_T)$ = 0.0

```
$V_T(s) = V_{T-1}(s)$ for all $s \in \mathcal{S}$
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $V_T(S_{t-1}) = V_T(S_{t-1}) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))$
```

TD(0)

γ = 0.9, α = 1.0

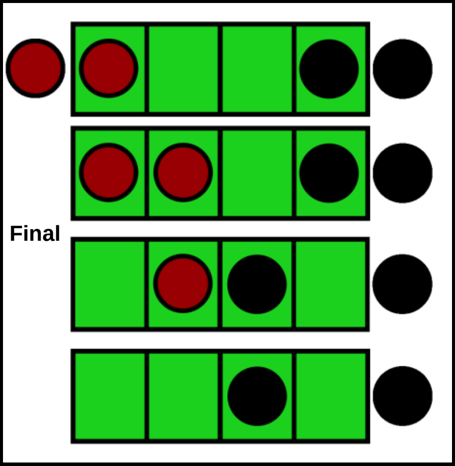


- $V(s_1)$ = 0.0
- $V(s_2)$ = 0.0
- $V(s_3)$ = 0.0 + 1.0 = 1.0
- $V(s_T)$ = 0.0

```
$V_T(s) = V_{T-1}(s)$ for all $s \in \mathcal{S}$
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $V_T(S_{t-1}) = V_T(S_{t-1}) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))$
```

TD(0)

γ = 0.9, α = 1.0



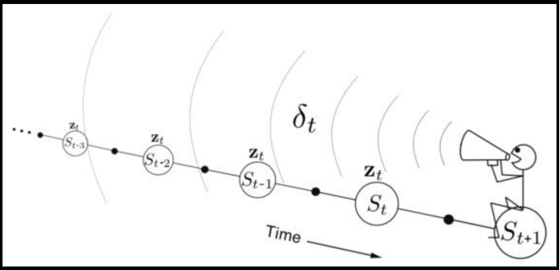
- V(s_1) = 0.0
- V(s_2) = 0.0
- V(s_3) = 1.0
- V(s_T) = 0.0

TD(1)

```
Algorithm $TD(1)$ for estimating $v_\pi$
  Input: a policy $\pi$
  Algorithm parameter: learning rate $\alpha \in (0,1]$
  Initialize $V_0(s) = 0$ for all $s \in \mathcal{S}$
  Generate episodes following $\pi$
  for each episode $T$ do
    $V_T(s) = V_{T-1}(s)$ for all $s \in \mathcal{S}$
    Initialize an eligibility function $E(s) = 0$ for all $s \in \mathcal{S}$
    for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
      $E(S_{t-1}) = E(S_{t-1}) + 1$
      for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \gamma E(s)$
  return $V_T$
```

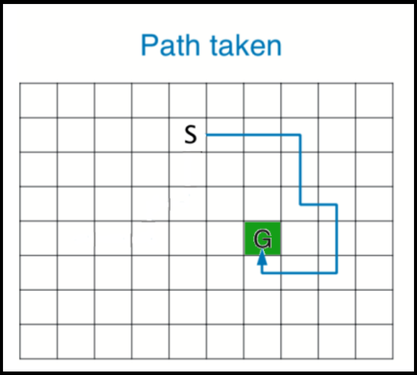
- Introduces an **eligibility trace, E,** which keeps track of events that occurred, as well as how much future rewards should be discounted.

Eligibility Trace

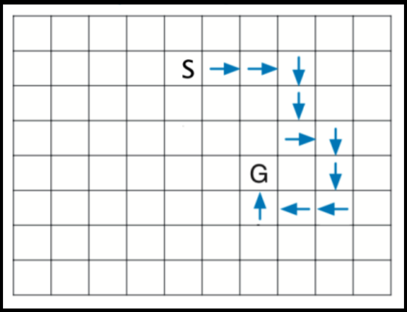


- E makes a temporary record that an event occurred, such as visiting a state or taking an action.
- Keeps track of both how recent and how frequent that event is.
- Marks states or actions as eligible for undergoing learning changes. When a TD error occurs, only the eligible states or actions are assigned credit or blame for the error.

TD(1) - GridWorld



Update after 1 episode of TD(1)

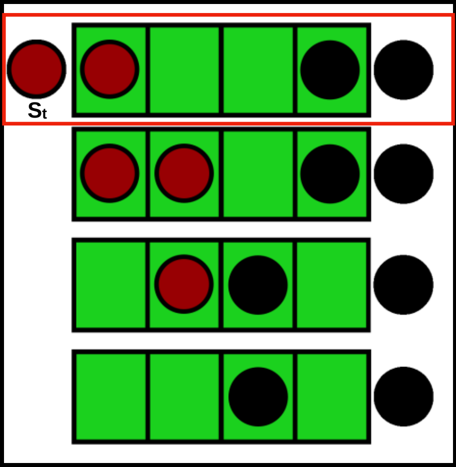


- Assume γ to be 1
- All state values are equally updated!

```
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $E(S_{t-1}) = E(S_{t-1}) + 1$
    for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \gamma E(s)$
```

TD(1)

γ = 0.9, α = 1.0

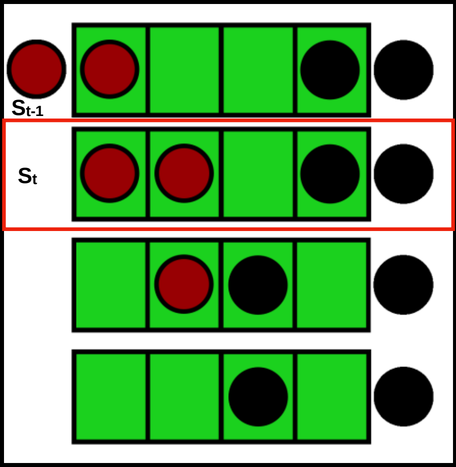


- $V(s_1)$ = 0.0
- $E(s_1)$ = 0.0
- $V(s_2)$ = 0.0
- $E(s_2)$ = 0.0
- $V(s_3)$ = 0.0
- $E(s_3)$ = 0.0

```
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $E(S_{t-1}) = E(S_{t-1}) + 1$
    for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \gamma E(s)$
```

TD(1)

γ = 0.9, α = 1.0

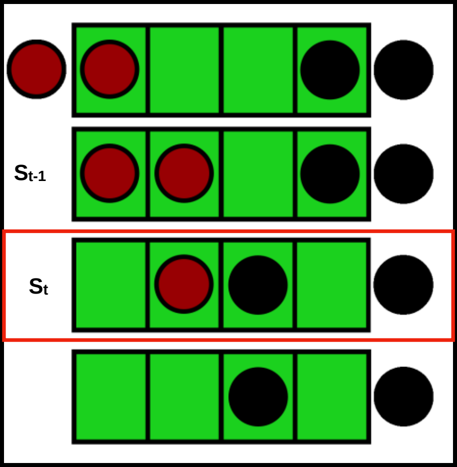


- $V(s_1)$ = 0.0 + 1 * 0.0 = **0.0**
- $E(s_1)$ = **1.0**
- $V(s_2)$ = 0.0
- $E(s_2)$ = 0.0
- $V(s_3)$ = 0.0
- $E(s_3)$ = 0.0

```
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $E(S_{t-1}) = E(S_{t-1}) + 1$
    for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \gamma E(s)$
```

TD(1)

γ = 0.9, α = 1.0

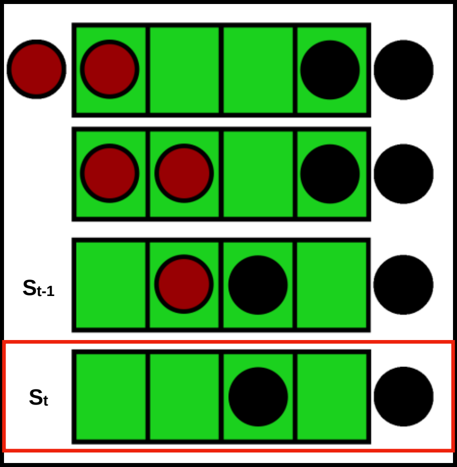


- $V(s_1) = 0.0 + \gamma * 0.0 =$ **0.0**
- $E(s_1) = \gamma$
- $V(s_2) = 0.0 + 1 * 0.0 =$ **0.0**
- $E(s_2) = 1$
- $V(s_3) = 0.0$
- $E(s_3) = 0.0$

```
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $E(S_{t-1}) = E(S_{t-1}) + 1$
    for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \gamma E(s)$
```

TD(1)

γ = 0.9, α = 1.0

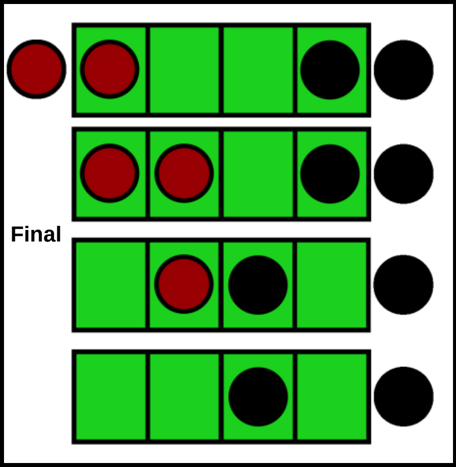


- $V(s_1) = 0.0 + \gamma^2 * 1.0 =$ **0.81**
- $E(s_1) = \gamma^2$
- $V(s_2) = 0.0 + \gamma * 1.0 =$ **0.9**
- $E(s_2) = \gamma$
- $V(s_3) = 0.0 + 1.0 * 1.0 =$ **1.0**
- $E(s_3) = 1$

```
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $E(S_{t-1}) = E(S_{t-1}) + 1$
    for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \gamma E(s)$
```

TD(1)

γ = 0.9, α = 1.0

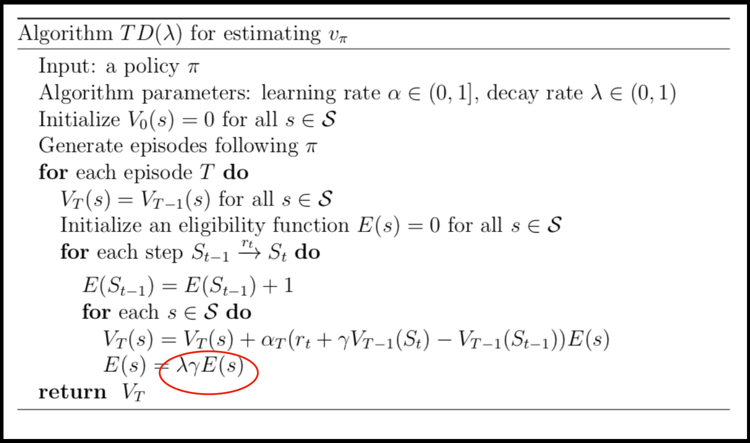


- **V(s_1) = 0.81**
- **V(s_2) = 0.9**
- **V(s_3) = 1.0**

TD(λ)

- Unifies the previous approaches, TD(0) and TD(1), under a single algorithm.
- TD(0) is TD(λ) where λ = 0
- TD(1) is TD(λ) where λ = 1
- λ is the **decay rate**, i.e., the rate at which eligibility traces decay over time.

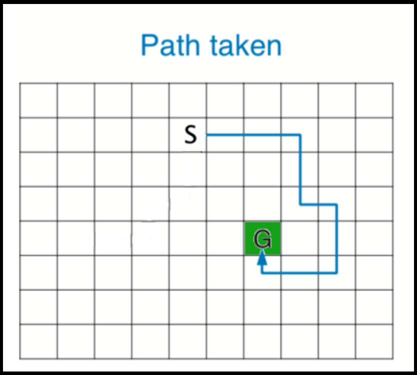
TD(λ)

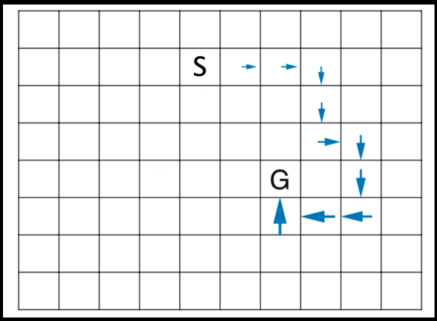


```
Algorithm $TD(\lambda)$ for estimating $v_\pi$
  Input: a policy $\pi$
  Algorithm parameters: learning rate $\alpha \in (0,1]$, decay rate $\lambda \in (0,1)$
  Initialize $V_0(s) = 0$ for all $s \in \mathcal{S}$
  Generate episodes following $\pi$
  for each episode $T$ do
    $V_T(s) = V_{T-1}(s)$ for all $s \in \mathcal{S}$
    Initialize an eligibility function $E(s) = 0$ for all $s \in \mathcal{S}$
    for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
      $E(S_{t-1}) = E(S_{t-1}) + 1$
      for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \lambda\gamma E(s)$
  return $V_T$
```

- Multiply the eligibility by the decay rate

TD(λ)



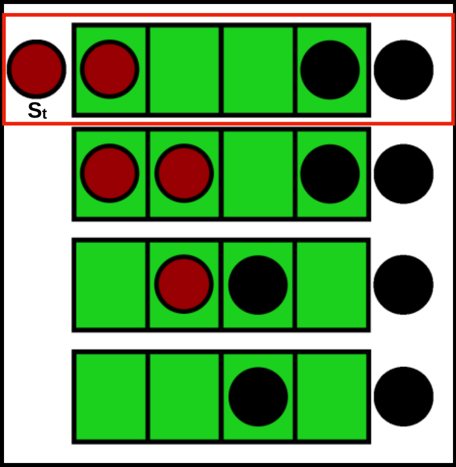


- Assume γ = to be close to 1 and λ = 0.9
- The updates decrease as we go further back in time from reaching goal

```
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $E(S_{t-1}) = E(S_{t-1}) + 1$
    for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \lambda\gamma E(s)$
```

TD(λ)

γ = 0.9
α = 1.0
λ = 0.5

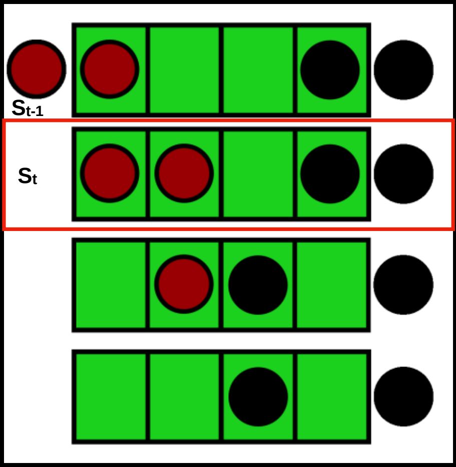


- $V(s_1) = 0.0$
- $E(s_1) = 0.0$
- $V(s_2) = 0.0$
- $E(s_2) = 0.0$
- $V(s_3) = 0.0$
- $E(s_3) = 0.0$

```
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $E(S_{t-1}) = E(S_{t-1}) + 1$
    for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \lambda\gamma E(s)$
```

TD(λ)

γ = 0.9
α = 1.0
λ = 0.5

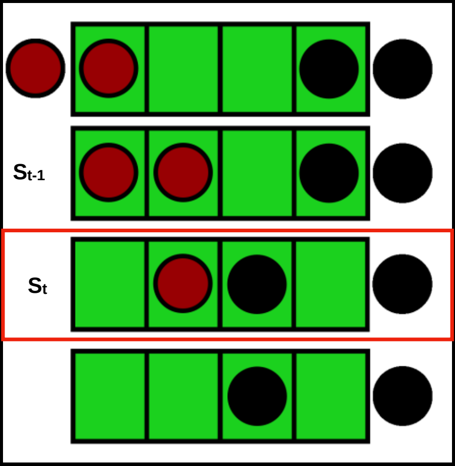


- $V(s_1) = 0.0 + 1 * 0.0 =$ **0.0**
- $E(s_1) = 1.0$
- $V(s_2) = 0.0$
- $E(s_2) = 0.0$
- $V(s_3) = 0.0$
- $E(s_3) = 0.0$

```
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $E(S_{t-1}) = E(S_{t-1}) + 1$
    for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \lambda\gamma E(s)$
```

TD(λ)

γ = 0.9
α = 1.0
λ = 0.5



- $V(s_1) = 0.0 + \lambda\gamma * 0.0 =$ **0.0**
- $E(s_1) = \lambda\gamma$

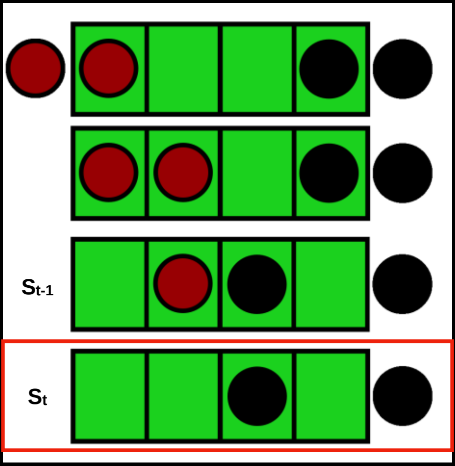
- $V(s_2) = 0.0 + 1 * 0.0 =$ **0.0**
- $E(s_2) = 1$

- $V(s_3) = 0.0$
- $E(s_3) = 0.0$

```
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $E(S_{t-1}) = E(S_{t-1}) + 1$
    for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \lambda\gamma E(s)$
```

TD(λ)

γ = 0.9
α = 1.0
λ = 0.5



- $V(s_1) = 0.0 + (\lambda\gamma)^2 * 1.0 =$ **0.2**
- $E(s_1) = (\lambda\gamma)^2$

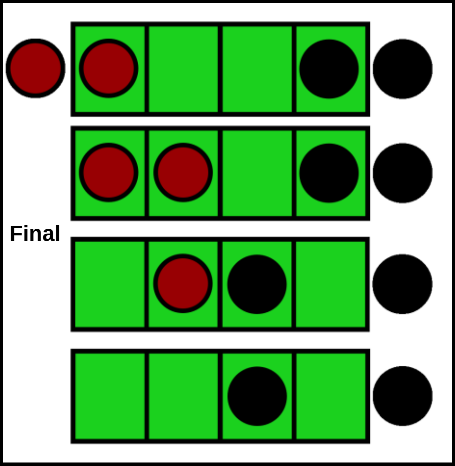
- $V(s_2) = 0.0 + \lambda\gamma * 1.0 =$ **0.45**
- $E(s_2) = \lambda\gamma$

- $V(s_3) = 0.0 + 1.0 * 1.0 =$ **1.0**
- $E(s_3) = 1$

```
for each step $S_{t-1} \xrightarrow{r_t} S_t$ do
    $E(S_{t-1}) = E(S_{t-1}) + 1$
    for each $s \in \mathcal{S}$ do
        $V_T(s) = V_T(s) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))E(s)$
        $E(s) = \lambda\gamma E(s)$
```

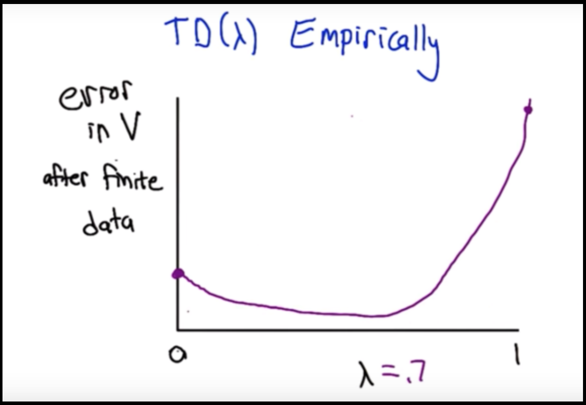
TD(λ)

γ = 0.9
α = **1.0**
λ = 0.5

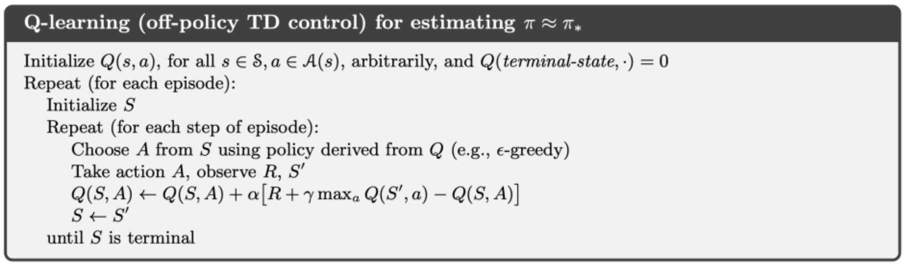


- **V(s_1) = 0.2**
- **V(s_2) = 0.45**
- **V(s_3) = 1.0**

TD(λ)



Q-Learning

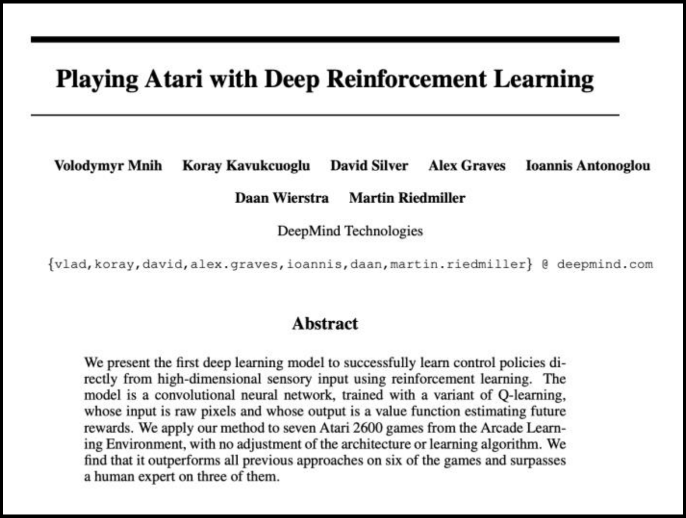


Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
  Initialize $S$
  Repeat (for each step of episode):
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-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
```

- A Q table uses state-action pairs as its keys, rather than just states.
- Q-learning is analogous to TD(0).
- It is “off-policy” in the sense that updates do not necessarily depend on the action picked. This allows for more exploration!
- “Epsilon-greedy” exploration chooses a random move with some probability (say 5%,); otherwise the best move is picked.

Deep Q-Network: DQN



Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou
Daan Wierstra Martin Riedmiller
DeepMind Technologies
{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

(2013) https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

Deep Q-Network: DQN

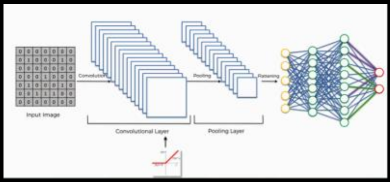
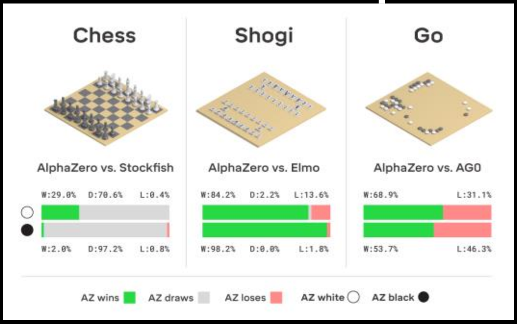


Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

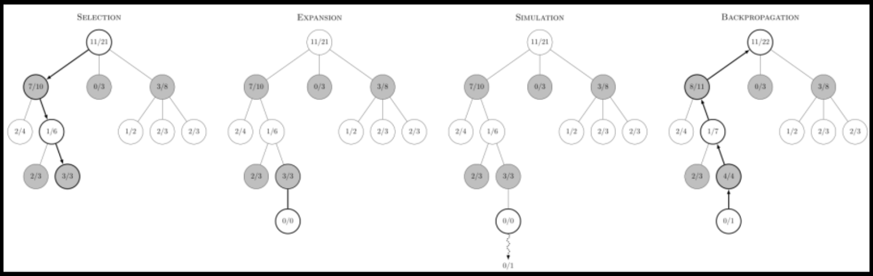
- Progress in RL stalled after TD-Gammon in the 1990s. However, the combination of Q-Learning with deep learning techniques and increased computation proved to be powerful.
- DQN achieved good performance on Atari games solely through simulated play. DQN uses a convolutional neural network as a function approximator (instead of a table).
- DQN achieves better stability through using an "experience replay buffer," allowing the network to train using past data.

AlphaGo and AlphaZero



- Success of DQN led to rapid development of new techniques for increasing performance and stability
- AlphaGo (2015) and AlphaZero (2017) attained superhuman performance in Go gameplay.
- AlphaGo was tailored to work with Go and trained using human expert play in addition to self-play. AlphaZero used self-play only, and was general enough to be applied to chess and Shogi.
- The AlphaZero approach combines elements of RL and DQN with Monte Carlo Tree Search, (MCTS) a powerful tree search technique.
- MCTS is related to alpha-beta pruning, but uses statistical sampling to discover promising moves in large state spaces.

MCTS



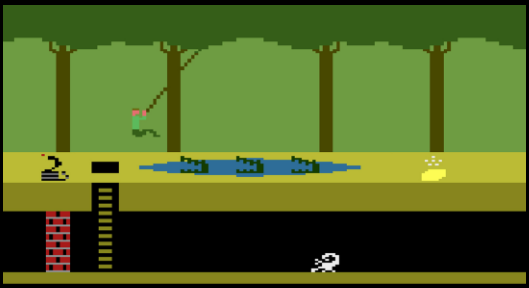
This graph shows the steps involved in one decision, with each node showing the ratio of wins to total playouts from that point in the game tree for the player that the node represents.[36] In the Selection diagram, black is about to move. The root node shows there are 11 wins out of 21 playouts for white from this position so far. It complements the total of 10/21 black wins shown along the three black nodes under it, each of which represents a possible black move.

If white loses the simulation, all nodes along the selection incremented their simulation count (the denominator), but among them only the black nodes were credited with wins (the numerator). If instead white wins, all nodes along the selection would still increment their simulation count, but among them only the white nodes would be credited with wins. In games where draws are possible, a draw causes the numerator for both black and white to be incremented by 0.5 and the denominator by 1. This ensures that during selection, each player's choices expand towards the most promising moves for that player, which mirrors the goal of each player to maximize the value of their move.

RL Pitfalls

- “Catastrophic forgetting”: Most RL algorithms produce agents that can perform well on one task only. If trained on a different task, their ability to perform the previous task is lost.
- Brittleness: Even slight changes to an environment can flummox trained agents. Famously, if you change the colors of the pixels or the position of the paddle, an expert Breakout agent can fail spectacularly.
- Reward shaping and sparsity: RL algorithms can achieve drastically different results depending on how rewards are shaped and how frequently they occur in a problem.
- Sim2Real: There continue to be real challenges in translating results in-silico to the real world

RL Pitfalls



- OpenAI Five achieved spectacular success with Dota 2. But it used a hand-tailored, information-rich reward function: https://gist.github.com/dfarhi/66ec9d760ae0c49a5c492c9fae93984a/
- Meanwhile, until recently, RL algorithms scored less than 0 points on the Atari game Pitfall: https://www.youtube.com/watch?v=aAJzamWAFOE
- It took much research effort and innovation to produce a superior Pitfall player! https://arxiv.org/pdf/1901.10995.pdf

Future Areas

- Meta-Learning: A subfield dedicated to improving transfer learning and enabling RL agents to perform better on related tasks
- Neuroevolution: Insights from evolutionary computation are being applied to reinforcement learning problems
- Multi-Agent RL: Increasingly sophisticated algorithms are being developed to handle the challenges of coordinating action between