Reinforcement Learning

2019-11-13

Announcements

- HW7 done; HW 8 is posted (due Dec 2, 23:55)
  - Grad/Ugrad both RL

Poll

- Who is familiar with reinforcement learning?
- Who feels able to implement Q learning?

https://www.technologyreview.com/s/603501/10-breakthrough-technologies-2017-reinforcement-learning/
https://www.youtube.com/watch?v=V1eYniJ0Rnk

Reinforcement Learning: An introduction.
Richard Sutton and Andrew Bartow, 2nd ed, MIT Press, 2017.
http://incompleteideas.net/book/the-book-2nd.html

REINFORCEMENT LEARNING

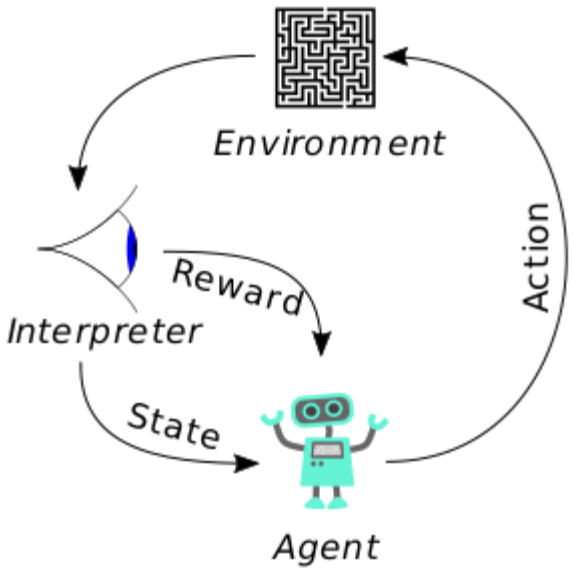
Situating this Lecture

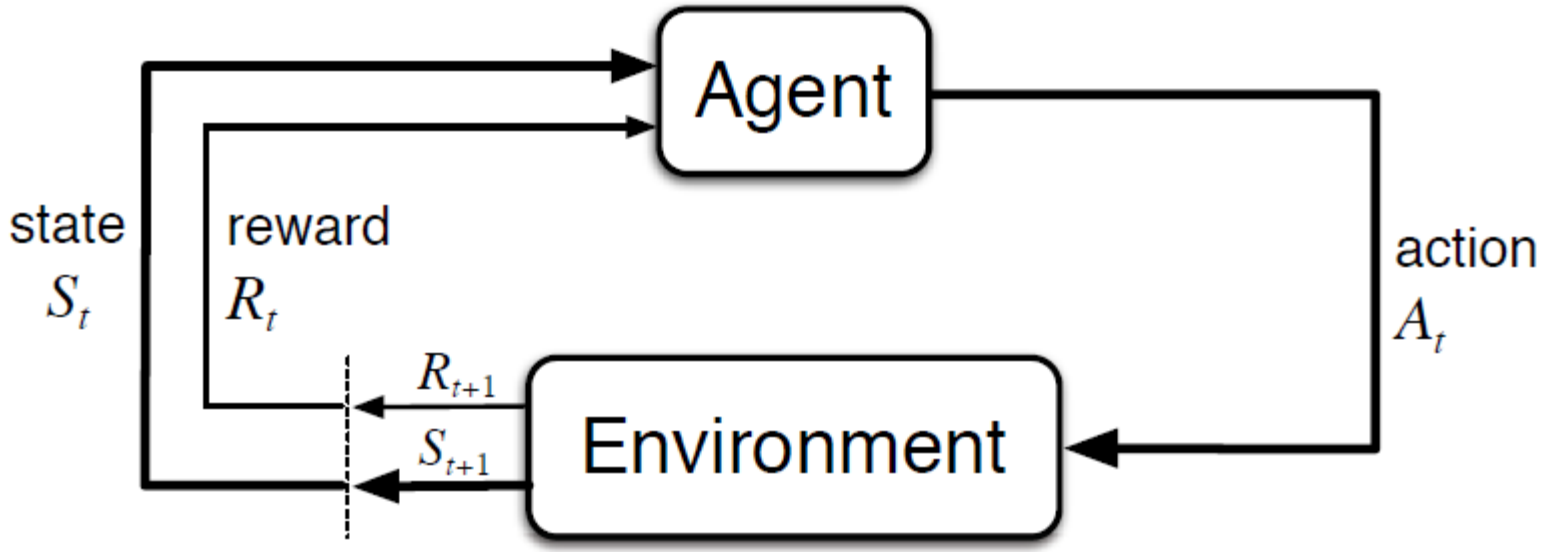
- At the beginning of the semester we drew a hard line between AI and Game AI
  - AAA
  - Indie
  - Academic

- Besides some occasional weird stuff (Black & White), we are now totally in the realm of academic Game AI.
  - Or are we?

Situating RL

- The BLUF: RL is about learning from interaction about how to map situations to actions to maximize reward. Key elements:
  - Trial-and-error search: must discover which actions yield most reward by trying them
  - Delayed reward: actions may affect both immediate reward and also next situation and all subsequent rewards
- Different from:
  - Supervised learning: training set of labeled examples
  - Unsupervised learning: finding latent structure in unlabeled data
- RL: maximize reward signal rather than discover hidden structure





By Megajuice - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=57895741

Sutton & Bartow, Fig 3.1

Fundamental Classes of Methods

- Dynamic programming
  - Mathematically well understood but require a complete and accurate model of the environment
- Monte Carlo methods
  - Model free and conceptually simple, but not well suited for step-by-step incremental computation
- Temporal-difference learning
  - Model free and fully incremental, but difficult to analyze

Also differ in efficiency/speed of convergence

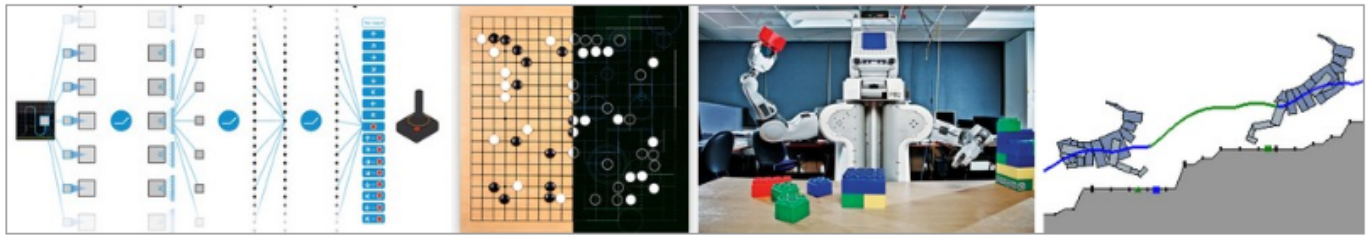
RL Has Revolutionized AI Gameplay

- Reinforcement Learning has been shown to be able to master many computer games
  - most Atari Games, Backgammon, Go, DOTA2, and StarCraft II.
- Reinforcement Learning (RL) is a class of algorithms that solve a Markov Decision Process. That is, the agent is given:
  - S: A set of all states the agent could encounter. Games: S is all configurations of pixels that the game engine can render
  - A: A set of all actions. Games: all player commands
  - T: A transition function that indicates the probability of transitioning from one state to another state if a particular action is chosen. We will assume that T is unknown.
  - R: A reward function that computes a score for every state in S. Games: reward given either based on in-game score, or for finishing a level, or achieving an objective.

Deep Reinforcement Learning: Pong from Pixels

May 31, 2016

This is a long overdue blog post on Reinforcement Learning (RL). RL is hot! You may have noticed that computers can now automatically learn to play ATARI games (from raw game pixels!), they are beating world champions at Go, simulated quadrupeds are learning to run and leap, and robots are learning how to perform complex manipulation tasks that defy explicit programming. It turns out that all of these advances fall under the umbrella of RL research. I also became interested in RL myself over the last ~year: I worked through Richard Sutton's book, read through David Silver's course, watched John Schulmann's lectures, wrote an RL library in Javascript, over the summer interned at DeepMind working in the DeepRL group, and most recently pitched in a little with the design/development of OpenAI Gym, a new RL benchmarking toolkit. So I've certainly been on this funwagon for at least a year but until now I haven't gotten around to writing up a short post on why RL is a big deal, what it's about, how it all developed and where it might be going.



Examples of RL in the wild. **From left to right**: Deep Q Learning network playing ATARI, AlphaGo, Berkeley robot stacking Legos, physically-simulated quadruped leaping over terrain.

It's interesting to reflect on the nature of recent progress in RL. I broadly like to think about four separate factors that hold back AI:

1. Compute (the obvious one: Moore's Law, GPUs, ASICs),
2. Data (in a nice form, not just out there somewhere on the internet - e.g. ImageNet),
3. Algorithms (research and ideas, e.g. backprop, CNN, LSTM), and
4. Infrastructure (software under you - Linux, TCP/IP, Git, ROS, PR2, AWS, AMT, TensorFlow, etc.).

http://karpathy.github.io/2016/05/31/rl/

Components of the AlphaGo

The core parts of the Alpha Go comprise of:

- **Monte Carlo Tree Search:** AI chooses its next move using MCTS
- Residual **CNNs (Convolutional Neural Networks):** AI assesses new positions using these networks
- **Reinforcement learning:** Trains the AI by using the current best agent to play against itself

In this blog, we will **focus on the working of Monte Carlo Tree Search** only. This helps AlphaGo and AlphaGo Zero smartly explore and reach interesting/good states in a finite time period which in turn helps the AI reach human level performance.

It's application extends beyond games. MCTS can theoretically be applied to any domain that can be described in terms of {*state*, *action*} pairs and simulation used to forecast outcomes. Don't worry if this sounds too complex right now, we'll break down all these concepts in this article.

https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/

https://www.youtube.com/watch?v=0g9SlVdv1PY

Proximal Policy Optimization

We're releasing a new class of reinforcement learning algorithms, Proximal Policy Optimization (PPO), which perform comparably or better than state-of-the-art approaches while being much simpler to implement and tune. PPO has become the default reinforcement learning algorithm at OpenAI because of its ease of use and good performance.

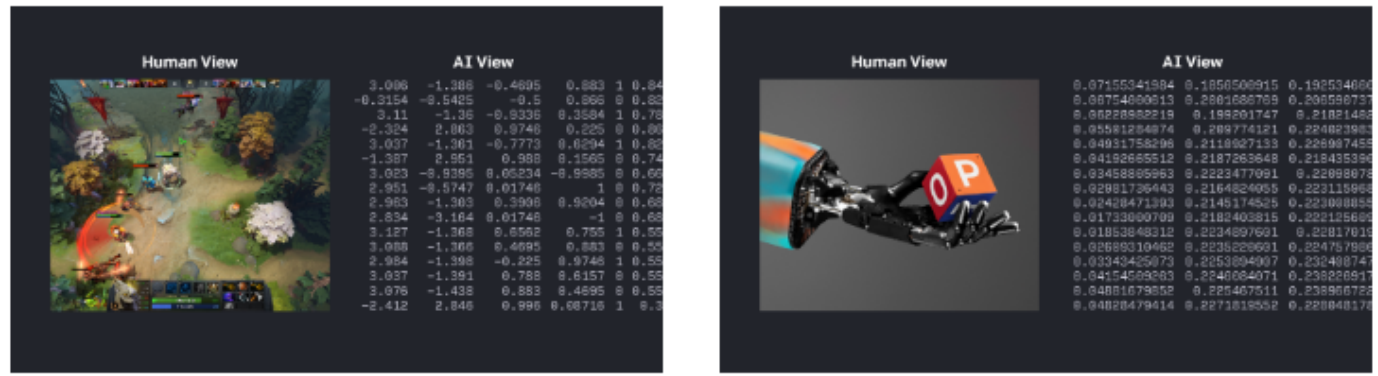
PPO lets us train AI policies in challenging environments, like the Roboschool one shown above where an agent tries to reach a target (the pink sphere), learning to walk, run, turn, use its momentum to recover from minor hits, and how to stand up from the ground when it is knocked over.

Policy gradient methods are fundamental to recent breakthroughs in using deep neural networks for control, from video games, to 3D locomotion, to Go. But getting good results via policy gradient methods is challenging because they are sensitive to the choice of stepsize — too small, and progress is hopelessly slow; too large and the signal is overwhelmed by the noise, or one might see catastrophic drops in performance. They also often have very poor sample efficiency, taking millions (or billions) of timesteps to learn simple tasks.

https://openai.com/blog/openai-baselines-ppo/

Why Dota?

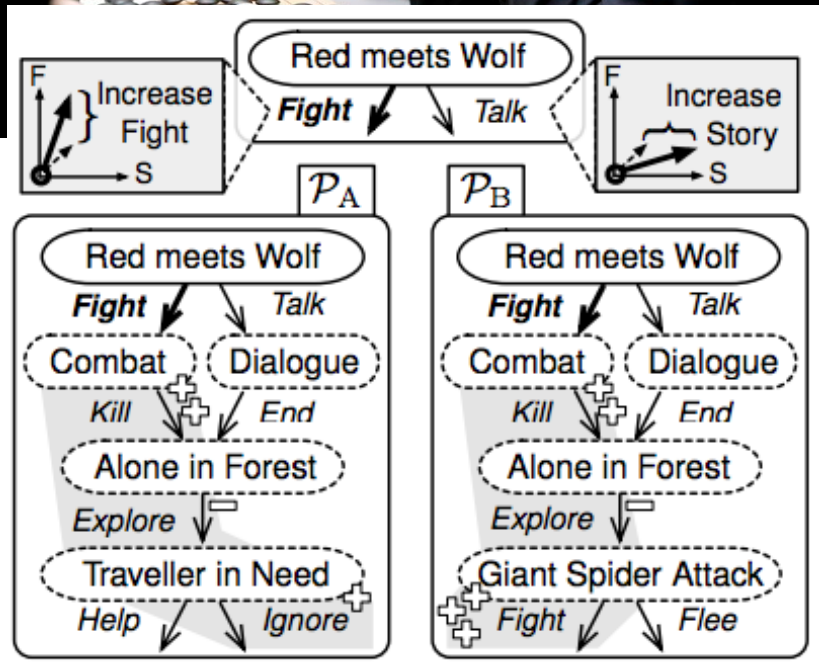
We started OpenAI Five in order to work on a problem that felt outside of the reach of existing deep reinforcement learning[1] algorithms. We hoped that by working on a problem that was unsolvable by current methods, we'd need to make a big increase in the capability of our tools. We were expecting to need sophisticated algorithmic ideas, such as hierarchical reinforcement learning, but we were surprised by what we found: the fundamental improvement we needed for this problem was scale. Achieving and utilizing that scale wasn't easy and was the bulk of our research effort!



OpenAI Five sees the world as a bunch of numbers that it must decipher. It uses the same general-purpose learning code whether those numbers represent the state of a Dota game (about 20,000 numbers) or robotic hand (about 200).

To build OpenAI Five, we created a system called Rapid which let us run PPO at previously unprecedented scale. The results exceeded our wildest expectations, and we produced a world-class Dota bot without hitting any fundamental performance limits.

Reinforcement Learning in Games



Reinforcement Learning: An Introduction

Richard S. Sutton
and Andrew G. Barto

Second Edition, in progress
MIT Press, Cambridge, MA, 2017

Online draft New Code Solutions Course Materials

Scholarly articles for **sutton and barto reinforcement learning**

Reinforcement learning: An introduction - **Sutton** - Cited by 25899

Reinforcement learning is direct adaptive optimal ... - **Sutton** - Cited by 348

http://incompleteideas.net/book/the-book-2nd.html

One-armed Bandit Problem

Scholarly articles for **sutton and barto reinforcement learning**

Reinforcement learning: An introduction - **Sutton** - Cited by 25899

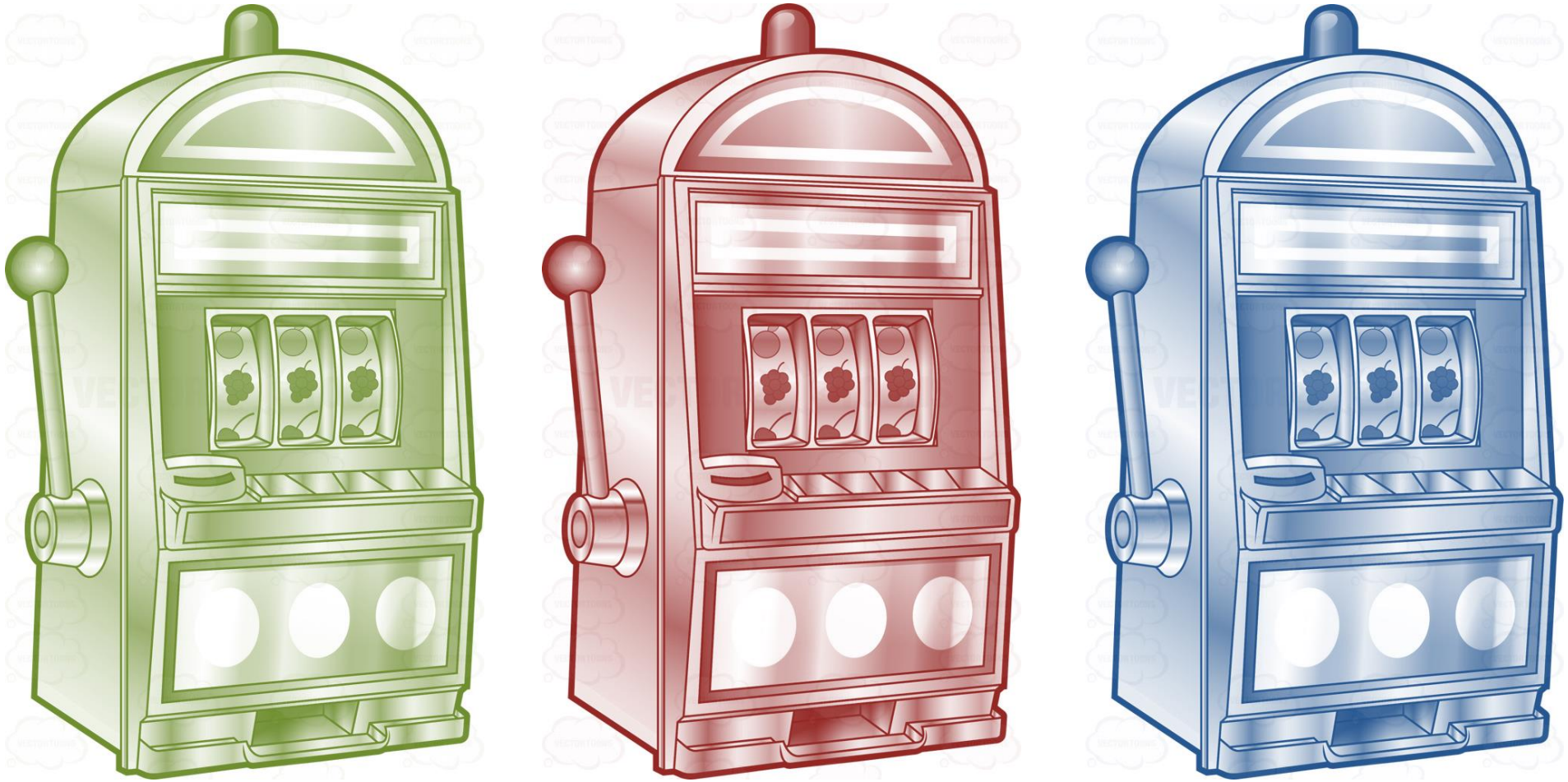
Reinforcement learning is direct adaptive optimal ... - **Sutton** - Cited by 348

One-armed Bandit Process

Pull #	Response	Believed Probability of Jackpot
1	WIN (1.0)	1.0
2	LOSS (0.0)	0.5
3	LOSS (0.0)	0.33…
4	LOSS (0.0)	0.25
…	…	…
N.	LOSS (0.0)	0.1

Unknown True Probability

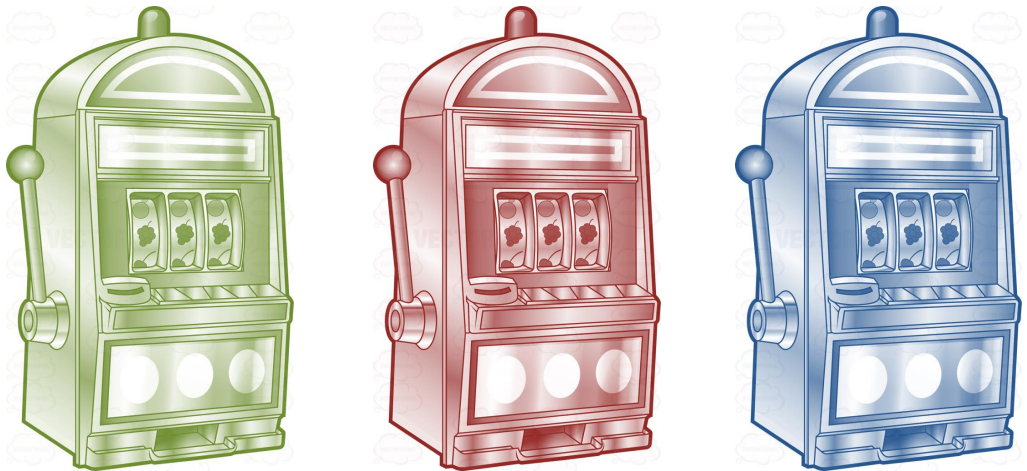
Multi-armed Bandit Problem



“Through repeated action selections you are to maximize your winnings by concentrating your actions on the best levers.” – Sutton & Barto

Now what?

- We can’t just keep playing one bandit to figure out its potential for reward (money)
- Want to maximize reward across all bandits
- We need to trade off making money with current knowledge and gaining knowledge
  - *Exploitation vs. Exploration*

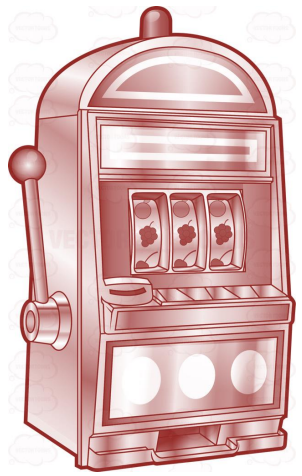


Just Exploitation

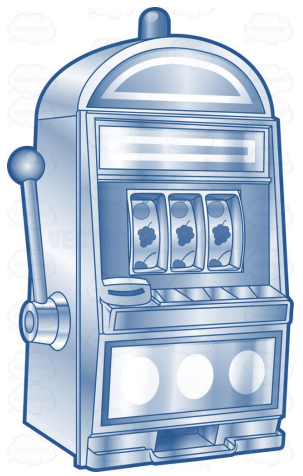
Greedy Strategy where we always pick the machine we think will give us the best reward (reminder: machines give rewards according to a probability)

Machine	**Response**	**Belief (G,R,B)**
B	LOSS	(-,-,0.0)
R	WIN	(0,1,0)
G	LOSS	(0,1,0)
R	LOSS	(0,0.5,0)
…	…	…

50%



25%



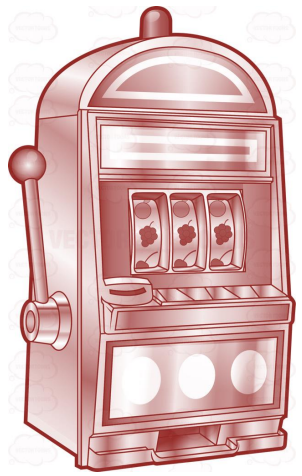
10%

Just Exploration

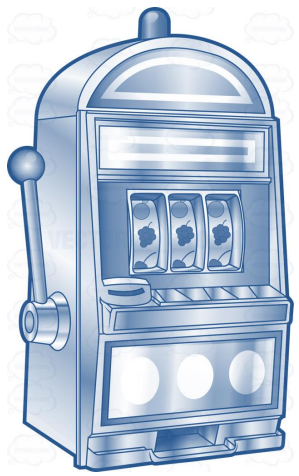
Always just pick a random machine, no matter what we believe.

Machine	Response	Belief (G,R,B)
B	LOSS	(-,-,0.0)
R	WIN	(0,1,0)
G	LOSS	(0,1,0)
B	**LOSS**	**(0,1,0)**
…	…	…

50%



25%



10%

Epsilon (ϵ) Greedy
(ϵ-Greedy)

(ϵ=0.1)

- 90% of the time try the best machine according to our current beliefs
- 10% of the time take random action

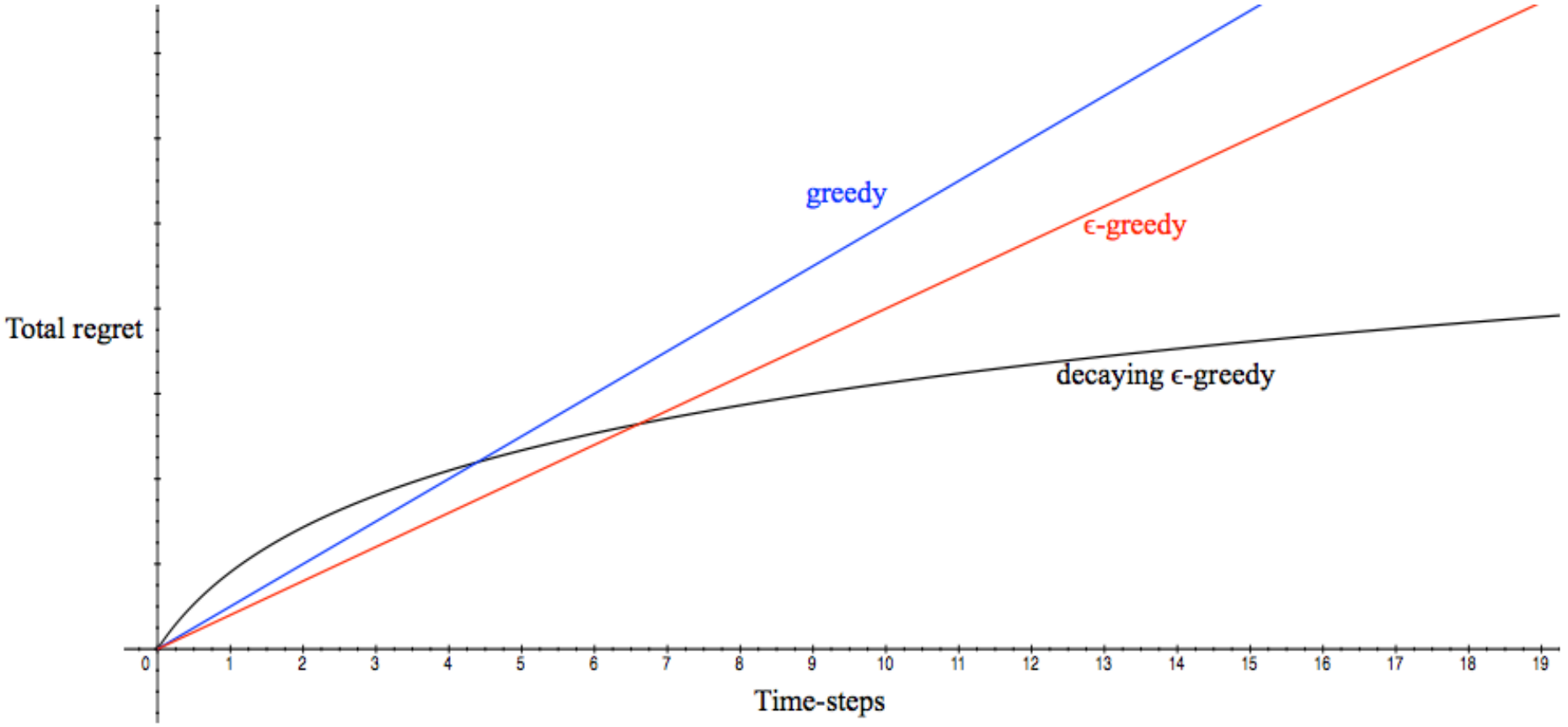
Now we can “escape” from early greedy mistakes!

Regret

- Imagine a perfect agent, who always takes the best action
- We can compare that agent's payout with our agent's payout for each action to get the *regret* for that step
- Summing regret across all time steps gives us the *total regret*

- Maximize total reward == minimize total regret

Different Strategies and Regret



Decaying ϵ-Greedy

- Drop ϵ lower and lower according to some schedule
- Now: logarithmic asymptotic regret

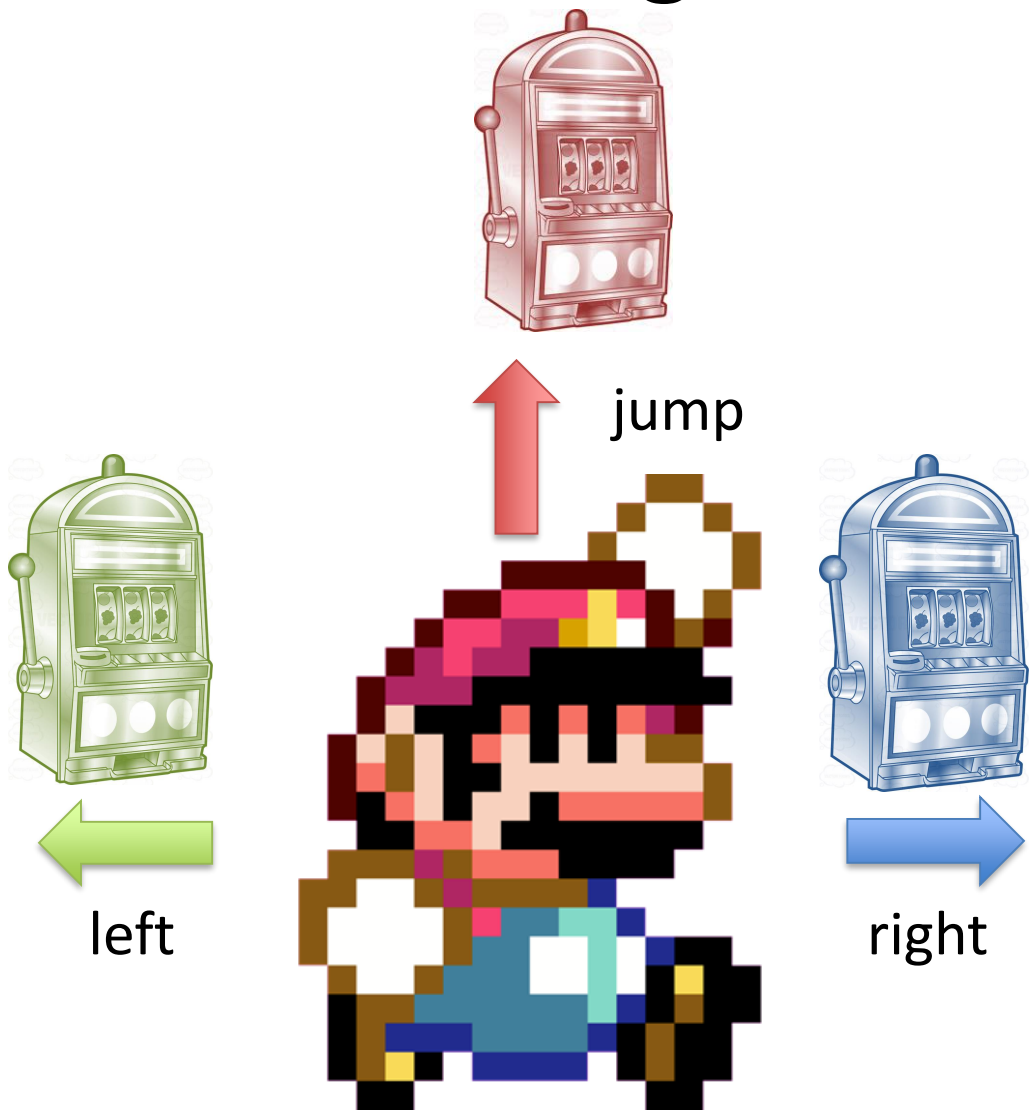
- Downside: we need to have enough domain knowledge to come up with a schedule beforehand

What does this have to do with games?

We can imagine mapping this onto a video game domain

Each distinct machine is now a distinct action

Reward: Not dying, end of level, etc.



But these values will probably differ based on context…

Value of going left versus right will be different if an enemy is to the left or the right.

State

- Multi-armed bandit problem is a “single state” or “stateless” **Markov Decision Process**

- But in games (and most cases we care about) there is more than one state

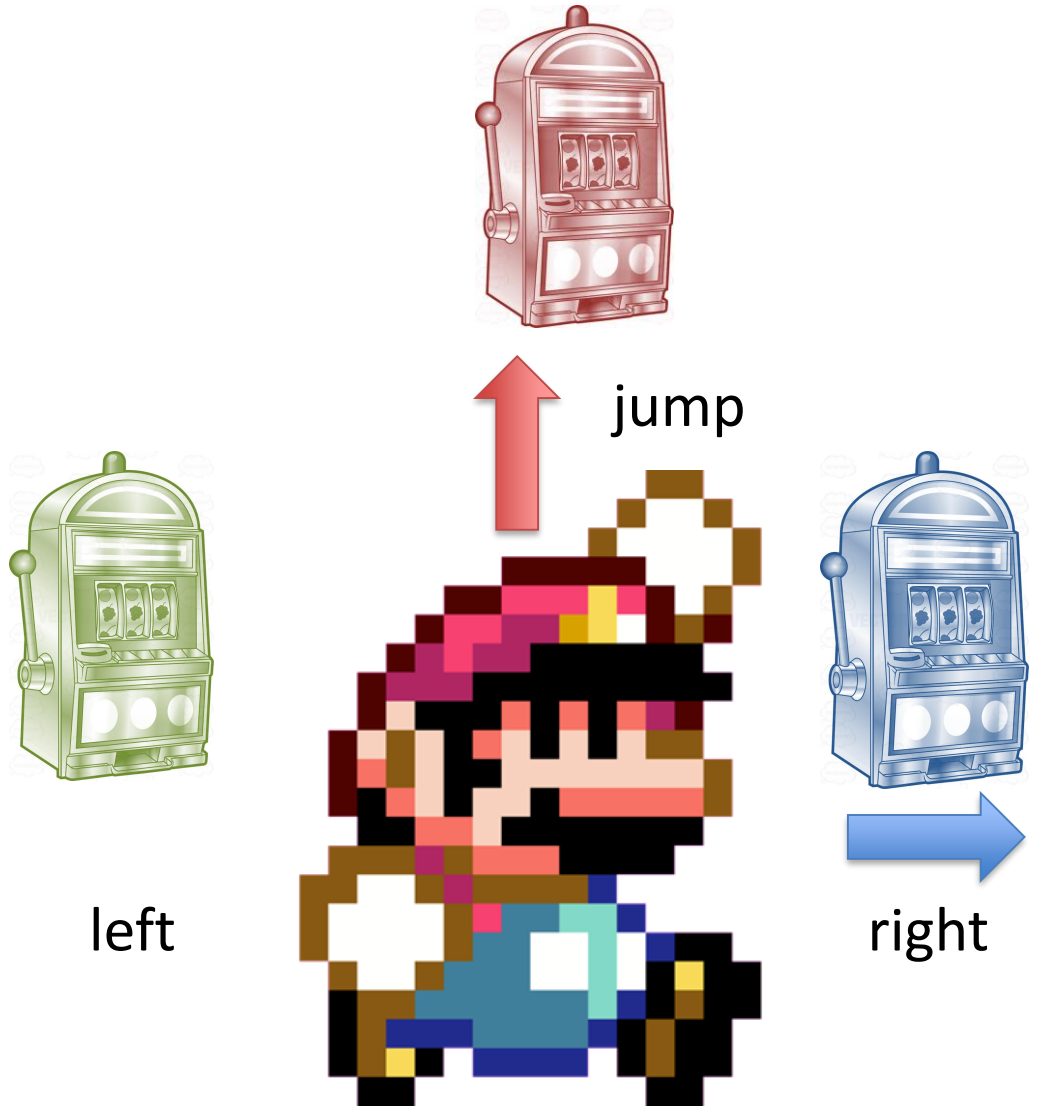
Andrey Markov

Markov Decision Process

- *S*: finite set of states
- *A*: finite set of actions
- $P(s,s_1)$: Probability that action a takes us from state s to state s_1
- $R(s,s_1)$: Reward for transitioning from state s to state s_1
- *y:* Discount factor ([0-1]). Demonstrates the difference in importance between future awards and present awards

High-level Idea

If the multi-armed bandit problem was a single state MDP, we can think of learning a strategy to play a game as solving this problem for *every state of the game*



Markovian State

- We call a state representation *Markovian* if it has all the information we need to make an optimal decision

State Representation Examples

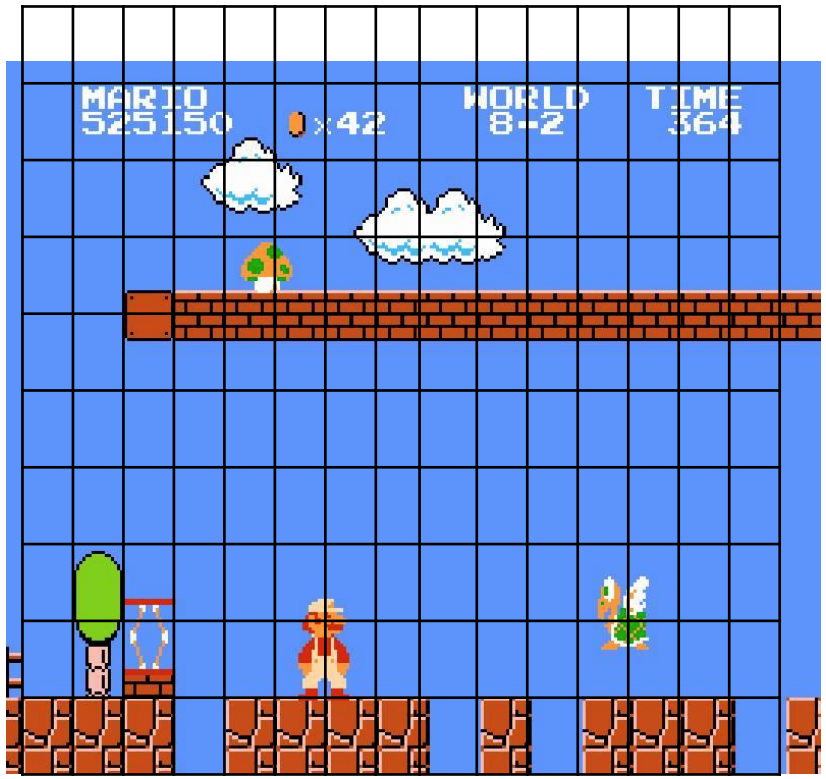
List of facts

- -enemy to right
- -powerup above
- -fire mario
- -on ground
- -etc…

State Representation Examples

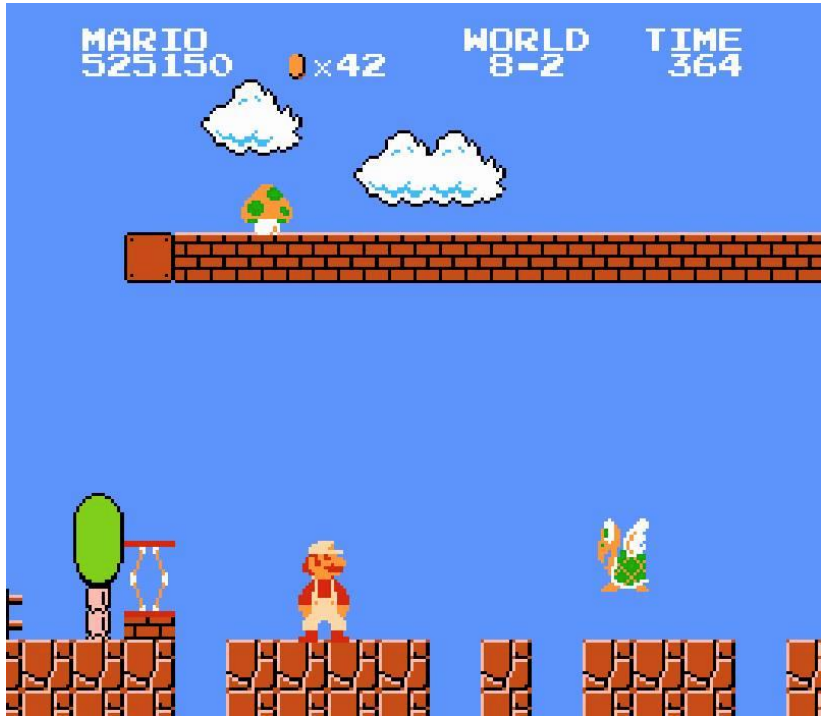
Grid Representations

- Segment all locations to tiles
- All enemies represented individually? Or just as “enemy”
- How much of the level to include in the state? Just screen? Smaller? Larger?



State Representation Examples

What pixels are present on the screen at this time?



State Representation Tradeoffs

- More complex state representations ensure that an agent has all info needed
- Less complex state representations speed up training (more states “look” the same)

- Goal: find the middle ground. Simplest state representation that still allows for optimal performance

Question

What state representations would you use for the following? Why?

- Tic-Tac-Toe
- A simple platformer
- A real time strategy game (Civilization/Starcraft)

Possible Answers

- Tic-Tac-Toe: The Gameboard
- Simple platformer: grid of screen width and height, with values for collideable elements, breakable elements, empty spaces, and enemy types (flying, etc)
- RTS: World map (?) + list of facts about currently running commands (building troops, moving troops, upgrading building, etc) (?)

Markov Decision Process

- *S*: finite set of states
- ***A*: finite set of actions**
  - **What are they in Tic-Tac-Toe? Platformers? RTS?**
- *P(s,s_1)*: Probability that action a takes us from state s to state s_1
- *R(s,s_1)*: **Reward** for transitioning from state s to state s_1
- γ*:* Discount factor ([0-1]). Demonstrates the difference in importance between future awards and present awards

What is the optimal action a *to take for every state* s?

Goal of the MDP

- Find the optimal action *a* to take for every state *s*
- We refer to this strategy as the policy

- Policy is represented as $\pi(s)$: S → A
  - What is the optimal action to take in state s?

MDP: Reward Function

- Better understood as “feedback”
  - An MDP’s reward can be positive or negative

- How to give reward? And what reward to give?

MDP: Transition function ($P(s|s_1, a)$)

- Also sometimes called a forward model
- Gives probability of transition from one state to another given a particular action
- Typically this is defined at a high level, not an individual state
  - E.g. When trying to move forward there is a 50% change of moving forward, a 25% change of moving to the left and a 25% chance of moving to the right
  - Not “when mario is at position (10,15) there is a 80% chance moving right gets mario to (11,15)

Value Iteration

Value Iteration:

- every state has a value V(s), generally represented as a table
- Known transition probabilities
- Known reward for every transition

Algorithm

- Start with arbitrary values for all V(s)
- Every iteration we update the value of every state according to the rewards we can get from nearby states
- Stop when values are no longer changing

See page 83 in S&B book linked earlier

Value Iteration Update

$$V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a) \left(R(s,a,s') + \gamma V_k(s')\right)$$

Value Iteration Update

$$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma V_k(s'))$$

$$V_k(s) = \max_a Q_k(s,a)$$

Value Iteration Update

$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a) (R(s,a,s') + \gamma V_k(s'))$

$V_k(s) = \max_a Q_k(s,a)$

^This probably looks like gibberish huh?

Value Iteration Update

$$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a) \, (R(s,a,s') + \gamma V_k(s'))$$

Translation:

The value of a particular action a in state s =

- the sum across all neighboring states of the
- the probability of going from state s to state s’ given action a multiplied by
- The reward of entering state s’ by action a from state s plus the discount factor γ multiplied by the current value of state s’

Value Iteration Update

$V_k(s) = \max_a Q_k(s,a)$

The new value of state s is then

- the max value within $Q_k(s,a)$

Value Iteration Algorithm

assign V0[S] arbitrarily

$k \leftarrow 0$

repeat

$k \leftarrow k+1$

for each state s **do**:

$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a) (R(s,a,s')+ \gamma V_k(s'))$

$V_k(s) = \max_a Q_k(s,a)$

until $V_k(s) - V_{k+1}(s) < \theta$:

Get Policy From Value Table

Remember we want the optimal action *a* for each state *s*

$\pi[s] = \text{argmax}_a \sum_{s'} P(s'|s,a) (R(s,a,s')+ \gamma V_k[s'])$

Translation:

Take the value of a that gives the max value…

- For each neighboring state and action a
  - Multiply the transition probability from state s to state s’ with action a by
  - The sum of
    - the reward for entering s’ from s with a and
    - The product of the discount value by the Value table for s’

Question

Give me all the pieces needed to run an MDP for the *simple* game of your choice

S is the set of all states

A is the set of all actions

P is state transition function specifying P(s'|s,a)

R is a reward function R(s,a,s')

Pros/Cons

Pros

- Once value iteration is done running we have an agent that can act optimally in any state very, very quickly (table lookup)

Cons:

- *Once value iteration is done running*
- **What if we don't have a transition function**? IE we don't know probability of getting from s to s'?
  - What if we have a black box that allows us to simulate it…

Seems like we have some extra stuff we don't really need huh?

$$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma V_k(s'))$$

$$V_k(s) = \max_a Q_k(s,a)$$

Seems like we have some extra stuff we don't really need huh?

$$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma \mathbf{V_k(s')})$$

$$\mathbf{V_k(s) = \max_a Q_k(s,a)}$$

Traditionally, there was a preference for keeping track of V(s) over Q(s,a) as it was smaller

Q-table Update

$$Q[s,a] \leftarrow Q[s,a] + \alpha * (r+\gamma * \max_{a'} Q[s',a']-Q[s,a])$$

A Q-table enumerates all possible states and all possible actions and gives the utility of each action in each state

States	Action 1	Action 2	Action 3	Action 4	Action 5
s1	0.1	0.5	0.9	0.4	0.0
s2	0.8	0.2	0.1	0.0	-1.0
s3	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...

See page 131 in S&B book linked earlier

Q-learning algorithm

Assign Q[S,A] arbitrarily

observe current state s

repeat

select and carry out an action a

observe reward r and state s’

$Q[s,a] \leftarrow Q[s,a] + \alpha * (r+\gamma * \max_{a'} Q[s',a']-Q[s,a])$

until termination

learning rate, α, ranges (0,1]
1 if env is deterministic
often 0.1

discount factor, γ, ranges [0,1]
0 is myopic, 1 long-term

the learned action-value function, Q, directly approximates the optimal action-value function, independent of the policy being followed

Q Learning

Pros

- We don’t need to have a forward model/transition function
- Means we can go into an environment blind

Cons

- Takes up more memory than Value Iteration as we have to hold a |S|*|A| table instead of just an |S|-sized table
  - Can be ameliorated with function approximation
- Bigger table means (at times) longer to converge

Q Learning

- The most general game playing/decision making technique we've seen so far
- Only needs a reward function, list of actions, and state representation
- But that state space sure is too massive to be useful huh?