CSC 411 Lecture 21-22: Reinforcement learning

Ethan Fetaya, James Lucas and Emad Andrews

University of Toronto

Today

- Learn to play games
- Reinforcement Learning

Playing Games: Atari

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

Playing Games: Super Mario

https://www.youtube.com/watch?v=wfL4L_l4U9A

Making Pancakes!

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

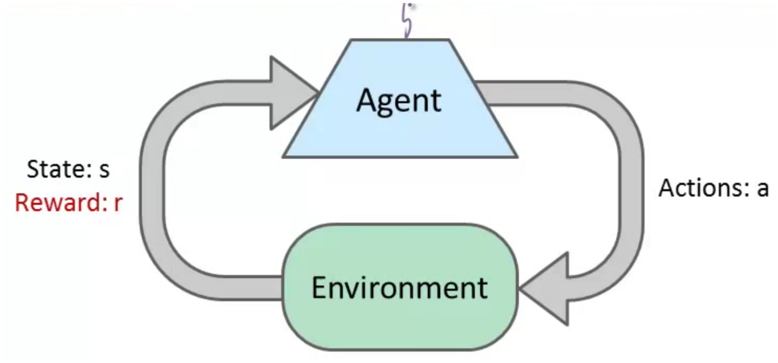
Reinforcement Learning Resources

- *Reinforcement Learning: An Introduction second edition*, Sutton & Barto Book (2016)
- Video lectures by David Silver

Reinforcement Learning

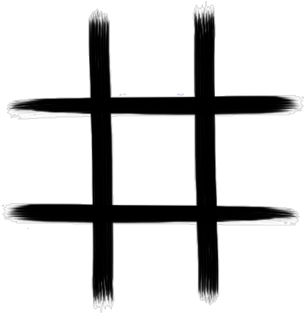
- Learning algorithms differ in the information available to learner
  - Supervised: correct outputs
  - Unsupervised: no feedback, must construct measure of good output
  - Reinforcement learning: Reward.
- More realistic learning scenario:
  - Continuous stream of input information, and actions
  - Effects of action depend on state of the world
  - Obtain reward that depends on world state and actions
    - You know the reward for your action, not other actions.
    - Could be a delay between action and reward.

Reinforcement Learning



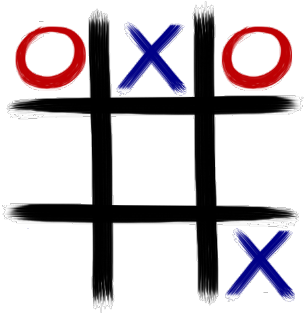
[pic from: Peter Abbeel]

Example: Tic Tac Toe, Notation



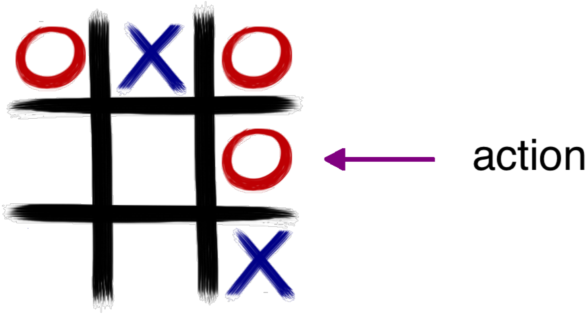
environment

Example: Tic Tac Toe, Notation

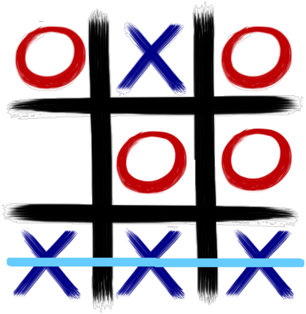


(current)
state

Example: Tic Tac Toe, Notation



Example: Tic Tac Toe, Notation



reward
(here: -1)

Formulating Reinforcement Learning

- World described by a set of states and actions
- At every time step t, we are in a state s_t, and we:
  - Take an action a_t (possibly null action)
  - Receive some reward r_{t+1}
  - Move into a new state s_{t+1}
- An RL agent may include one or more of these components:
  - Policy π: agent's behaviour function
  - Value function: how good is each state and/or action
  - Model: agent's representation of the environment

Policy

- A policy is the agent's behaviour.
- It's a selection of which action to take, based on the current state
- Deterministic policy: $a = \pi(s)$
- Stochastic policy: $\pi(a|s) = P[a_t = a|s_t = s]$

[Slide credit: D. Silver]

Value Function

- Value function is the expected future reward
- Used to evaluate the goodness/badness of states
- Our aim will be to maximize the value function (the total reward we receive over time): find the policy with the highest expected reward
- By following a policy π, the value function is defined as:

$$V^{\pi}(s_t) \quad = \quad \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots]$$

- γ is called a discount rate, and it is always $0 \leq \gamma \leq 1$
- If γ close to 1, rewards further in the future count more, and we say that the agent is "farsighted"
- γ is less than 1 because there is usually a time limit to the sequence of actions needed to solve a task (we prefer rewards sooner rather than later)

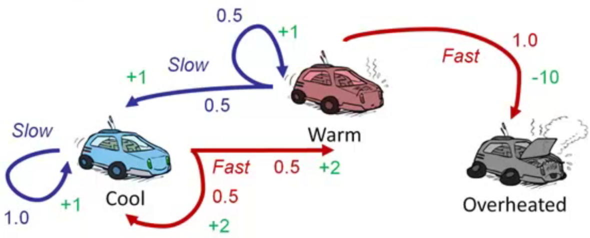
[Slide credit: D. Silver]

Model

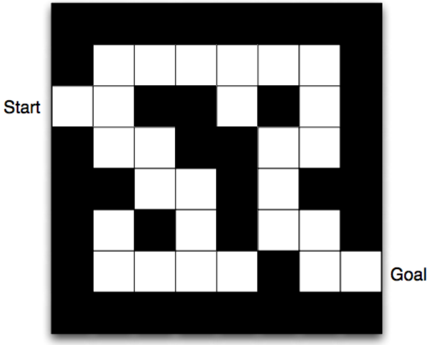
- The model describes the environment by a distribution over rewards and state transitions:

$$P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)$$

- We assume the Markov property: the future depends on the past only through the current state



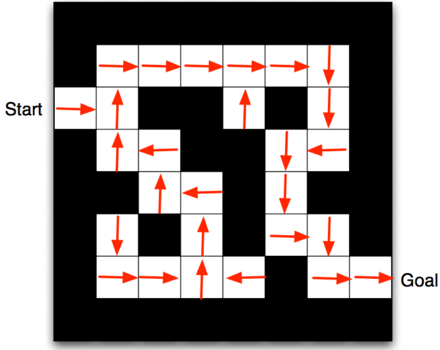
Maze Example



- Rewards: -1 per time-step
- Actions: N, E, S, W
- States: Agent's location

[Slide credit: D. Silver]

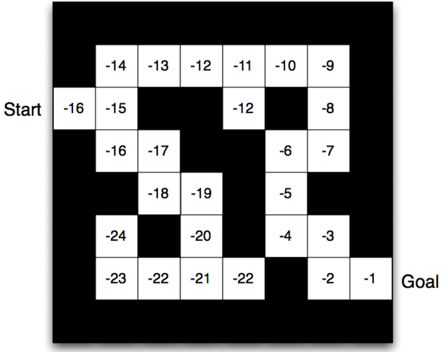
Maze Example



- Arrows represent policy $\pi(s)$ for each state s

[Slide credit: D. Silver]

Maze Example



- Numbers represent value $V^{\pi}(s)$ of each state s

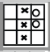
[Slide credit: D. Silver]

Example: Tic-Tac-Toe

- Consider the game tic-tac-toe:
  - reward: win/lose/tie the game $(+1/-1/0)$ [only at final move in given game]
  - state: positions of X's and O's on the board
  - policy: mapping from states to actions
    - based on rules of game: choice of one open position
  - value function: prediction of reward in future, based on current state
- In tic-tac-toe, since state space is tractable, can use a table to represent value function

RL & Tic-Tac-Toe

- Each board position (taking into account symmetry) has some probability

State	Probability of a win (Computer plays "o")
	0.5
	0.5
	1.0
	0.0
	0.5
etc	

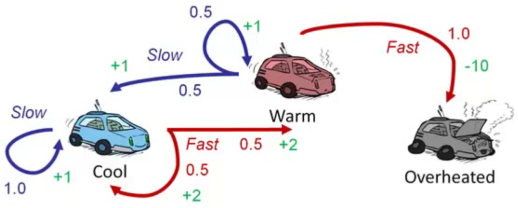
- Simple learning process:
  - start with all values $= 0.5$
  - policy: choose move with highest probability of winning given current legal moves from current state
  - update entries in table based on outcome of each game
  - After many games value function will represent true probability of winning from each state
- Can try alternative policy: sometimes select moves randomly (exploration)

MDP

- Markov Decision Problem (MDP): tuple (S, A, P, γ) where P is

$$P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)$$

- Main assumption: Markovian dynamics and reward.
- Standard MDP problems:
  1. Planning: given complete Markov decision problem as input, compute policy with optimal expected return



[Pic: P. Abbeel]

Basic Problems

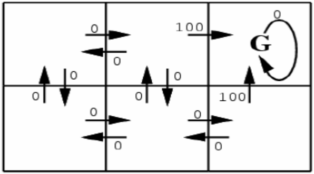
- Markov Decision Problem (MDP): tuple (S, A, P, γ) where P is

$$P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)$$

- Standard MDP problems:
  1. Planning: given complete Markov decision problem as input, compute policy with optimal expected return
  2. Learning: We don't know which states are good or what the actions do. We must try out the actions and states to learn what to do

[P. Abbeel]

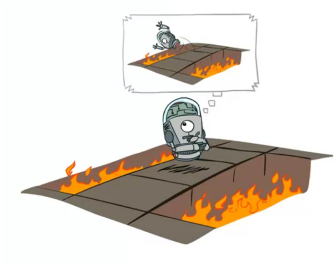
Example of Standard MDP Problem

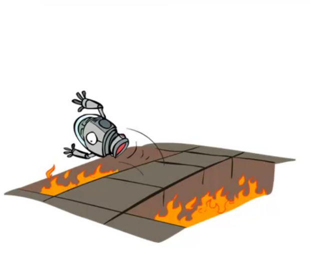


$r(s, a)$ (immediate reward)

1. Planning: given complete Markov decision problem as input, compute policy with optimal expected return
2. Learning: Only have access to experience in the MDP, learn a near-optimal strategy

Example of Standard MDP Problem





1. Planning: given complete Markov decision problem as input, compute policy with optimal expected return
2. Learning: Only have access to experience in the MDP, learn a near-optimal strategy

We will focus on learning, but discuss planning along the way

Exploration vs. Exploitation

- If we knew how the world works (embodied in P), then the policy should be deterministic
  - just select optimal action in each state
- Reinforcement learning is like trial-and-error learning
- The agent should discover a good policy from its experiences of the environment
- Without losing too much reward along the way
- Since we do not have complete knowledge of the world, taking what appears to be the optimal action may prevent us from finding better states/actions
- Interesting trade-off:
  - immediate reward (exploitation) vs. gaining knowledge that might enable higher future reward (exploration)

Examples

- Restaurant Selection
  - Exploitation: Go to your favourite restaurant
  - Exploration: Try a new restaurant
- Online Banner Advertisements
  - Exploitation: Show the most successful advert
  - Exploration: Show a different advert
- Oil Drilling
  - Exploitation: Drill at the best known location
  - Exploration: Drill at a new location
- Game Playing
  - Exploitation: Play the move you believe is best
  - Exploration: Play an experimental move

[Slide credit: D. Silver]

Value function

- The value function $V^\pi(s)$ assigns each state the expected reward

$$V^\pi(s) = \mathop{\mathbb{E}}_{a_t, a_{t+i}, s_{t+i}} \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s \right]$$

- Usually not informative enough to make decisions.
- The Q-value $Q^\pi(s, a)$ is the expected reward of taking action a in state s and then continuing according to π.

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{a_{t+i}, s_{t+i}} \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s, a_t = a \right]$$

Bellman equations

- The foundation of many RL algorithms

$$V^\pi(s) = \mathop{\mathbb{E}}_{a_t, a_{t+i}, s_{t+i}} \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s \right]$$

$$= \mathop{\mathbb{E}}_{a_t} [r_t | s_t = s] + \gamma \mathop{\mathbb{E}}_{a_t, a_{t+i}, s_{t+i}} \left[\sum_{i=1}^{\infty} \gamma^i r_{t+i+1} | s_t = s \right]$$

$$= \mathop{\mathbb{E}}_{a_t} [r_t | s_t = s] + \gamma \mathop{\mathbb{E}}_{s_{t+1}} [V^\pi(s_{t+1}) | s_t = s]$$

$$= \sum_{a,r} P^\pi(a|s_t) p(r|a, s_t) \cdot r + \gamma \sum_{a,s'} P^\pi(a|s_t) p(s'|a, s_t) \cdot V^\pi(s')$$

- Similar equation holds for Q

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{a_{t+i}, s_{t+i}} \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s, a_t = a \right]$$

$$= \sum_{r} p(r|a, s_t) \cdot r + \gamma \sum_{s'} p(s'|a, s_t) \cdot V^\pi(s')$$

$$= \sum_{r} p(r|a, s_t) \cdot r + \gamma \sum_{a', s'} p(s'|a, s_t) p(a'|s') \cdot Q^\pi(s', a')$$

Solving Bellman equations

- The Bellman equations are a set of linear equations with a unique solution.
- Can solve fast(er) because the linear mapping is a contractive mapping.
- This lets you know the quality of each state/action under your policy - policy evaluation.
- You can improve by picking $\pi'(s) = \max_a Q^\pi(s, a)$ - policy improvement.
- Can show the iterative policy evaluation and improvement converges to the optimal policy.
- Are we done? Why isn't this enough?
  - Need to know the model! Usually isn't known.
  - Number of states is usually huge (how many unique states does a chess game have?)

Optimal Bellman equations

- First step is understand the Bellman equation for the optimal policy π^*
- Under this policy $V^*(s) = \max_a Q^*(s,a)$

$$V^*(s) = \max_a \left[\mathbb{E}\left[r_{t+1}|s_t = s, a_t = a\right] + \gamma \mathop{\mathbb{E}}_{s_{t+1}}\left[V^*(s_{t+1})|s_t = s, a_t = a\right]\right]$$

$$= \max_a \left[\sum_r p(r|a,s_t) \cdot r + \gamma \sum_{s'} p(s'|a,s_t) \cdot V^*(s') \right]$$

$$Q^*(s,a) = \mathbb{E}\left[r_{t+1}|s_t = s, a_t = a\right] + \gamma \mathop{\mathbb{E}}_{s_{t+1}} \left[\max_{a'} Q^*(s_{t+1},a')|s_t = s, a_t = a \right]$$

$$= \sum_r p(r|a,s_t) \cdot r + \gamma \sum_{s'} p(s'|a,s_t) \cdot \max_{a'} Q^*(s',a')$$

- Set on nonlinear equations.
- Same issues as before.

Q-learning intuition

- Q-learning is a simple algorithm to find the optimal policy without knowing the model.
- Q^* is the unique solution to the optimal Bellman equation.

$$Q^*(s,a) = \mathbb{E}\left[r_{t+1}|s_t = s, a_t = a\right] + \gamma \mathop{\mathbb{E}}_{s_{t+1}}\left[\max_{a'} Q^*(s_{t+1}, a')|s_t = s, a_t = a\right]$$

- We don't know the model and don't want to update all states simultaneously.
- Solution - given sample $s_t, a_t, r_{t+1}, s_{t+1}$ from the environment update your Q-values so they are closer to satisfying the bellman equation.
  - off-policy method: Samples don't have to be from the optimal policy.
- Samples need to be diverse enough to see everything - exploration.

Exploration vs exploitation

- Given Q-value the best thing we can do (given our limited knowledge) is to take $a = \arg\max_{a'} Q(s, a')$ - exploitation
- How do we balance exploration with exploitation?
- Simplest solution: ϵ-greedy.
  - With probability $1 - \epsilon$ pick $a = \arg\max_{a'} Q(s, a')$ (i.e. greedy)
  - With probability ϵ pick any other action uniformly.
- Another idea - softmax using Q values
  - With probability $1 - \epsilon$ pick $a = \arg\max_{a'} Q(s, a')$ (i.e. greedy)
  - With probability ϵ pick any other action with probability $\propto \exp(\beta Q(s, a))$.
- Other fancier solutions exist, many leading methods use simple ϵ-greedy sampling.

Q-learning algorithm

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

- Can prove convergence to the optimal Q^* under mild conditions.
- Update is equivalent to gradient descent on loss $||R + \gamma \max_a Q(S', a) - Q(s, a)||^2$.
- Why L_2 loss? Optimal solution is the mean which is what we are looking for!

Bootstrapping

- Another way to think about Q-learning.
- $Q(s, a)$ is the expected reward, can use Monte-Carlo estimation.
- Problem - you update only after the episode ends, can be very long (or infinite).
- Q-learning solution - take only 1 step forward and estimate the future using our Q value - bootstrapping.
  - "learn a guess from a guess"
- Q-learning is just one algorithm in a family of algorithms that use this idea.

Function approximation

- Q-learning still scales badly with large state spaces, how many states does a chess game have? Need to save the full table!
- Similar states, e.g. move all chess pieces two steps to the left, at treated as totally different.
- Solution: Instead of Q being a $S \times A$ table it is a parametrized function.
- Looking for function $\hat{Q}(s, a; \mathbf{w}) \approx Q^*(s, a)$
  - Linear functions $Q(s, a; \mathbf{w}) = \mathbf{w}^T \phi(s, a)$.
  - Neural network
- Hopefully can generalize to unseen states.
- Problem: Each change to parameters changes all states/actions - can lead to instability.
- For non-linear Q-learning can diverge.

Deep Q-learning

- We have a function approximator $Q(s, a; \theta)$, standard is neural net but doesn't have to be.
- What is the objective that we are optimizing?
- We want to minimize $\mathbb{E}_{\rho}[||R + \gamma \max_{a'} Q(S', a') - Q(s, a)||^2]$
  - ρ is a distribution over states, depends on θ!
- Two terms depend on Q, don't want to take gradients w.r. to $\gamma \max_a Q(S', a)$
- We want to correct our previous estimation given the new information.

online Q iteration algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

Figure: Take from:rll.berkeley.edu/deeprlcourse

- This simple approach doesn't work well as is.

Issues and solutions

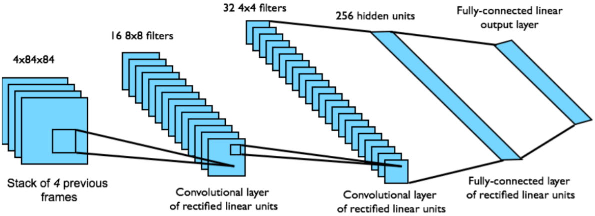
- Problem: data in the minibatch is highly correlated
  - Consecutive samples are from the same episode and probably similar states.
  - Solution: Replay memory.
  - You store a large memory buffer of previous (s, a, r, s') (notice this is all you need for Q-learning) and sample from it to get diverse minibatch.
- Problem: The data distribution keeps changing
  - Since we aren't optimizing y_i its like solving a different (but related) least squares each iteration.
  - We can stabilize by fixing a target network for a few iterations

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update ϕ': copy ϕ every N steps

Figure: Take from:rll.berkeley.edu/deeprlcourse

Example: DQN on atari

- Trained a NN from scratch on atari games



- Ablation study

	Replay Fixed-Q	Replay Q-learning	No replay Fixed-Q	No replay Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.3	831.25	141.89	29.1
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.4	822.55	1003	275.81
Space Invaders	1088.94	826.33	373.22	301.99

RL recap

- Learning from experience not from labeled examples.
- Why is RL hard?
  - Limited feedback.
  - Delayed rewards.
  - Your model effect what you see.
  - Huge state space.
- Usually solved by learning the value function or optimizing the policy (not covered)
- Model based method but less successful at the moment.
- How do you define the rewards? Can be trick.
  - Bad rewards can lead to reward hacking

Q-Learning recap

- Try to find Q that satisfies the optimal Bellman conditions
- Off-policy algorithm - Doesn't have to follow a greedy policy to evaluate it.
- Model free algorithm - Doesn't have any model for instantaneous reward or dynamics.
- Learns a separate value for each s, a pair - doesn't scale up to huge state spaces.
- Can scale using a function approximation
  - No more theoretical guarantees.
  - Can diverge.
  - Some simple tricks help a lot.