ESCOLA
POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA

Intelligent Systems (IS)

Reinforcement Learning

Josep Pon

[Slides adapted from Dan Klein and Pieter Abbeel (`ai.berkeley.edu`)]

1. Preliminaries
   - Markov Decision Process (MDP)
   - Policies
   - Utilities
   - Solving MDPs
   - Value Iteration
   - Policy Evaluation
   - Policy Extraction
   - Policy Iteration

2. Reinforcement Learning

3. Reinforcement Learning Algorithms

4. Extras

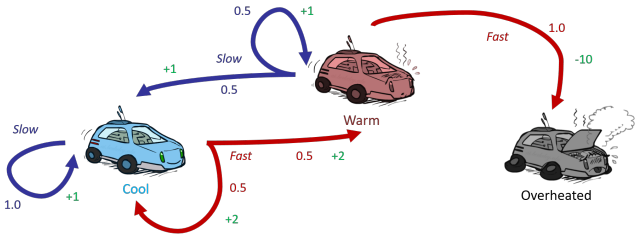
5. References

Markov Decision Process (MDP)

Modeling decision making

Describes a series of time discrete events under the control of a decision maker.

- A set of states $s \in S$
- A set of actions $a \in A$
- A transition function T(s, a, s')
- A reward function R(s, a, s')
- A start state



Markov Decision Process (MDP)

Markov property

The transition function is memory less

$$P(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, \ldots, S_0 = s_0)$$
$$=$$
$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

The probability of arriving to a state s' at time $t+1$ depends only on the state s and action a taken at time t.

The **state** is not limited to sensory information, it can contain highly processed versions of original sensations and complex structures built up over time.

Policies

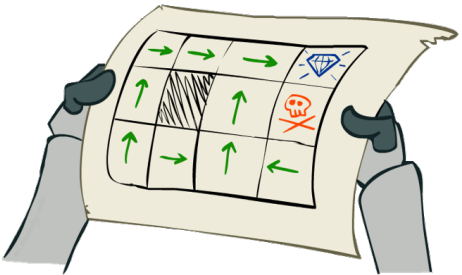
What should I do next?

The environment is stochastic, and bound to change.

- It is hard to devise a sequence of optimal actions beforehand.
- We need a policy that tells us how to act on any situation.

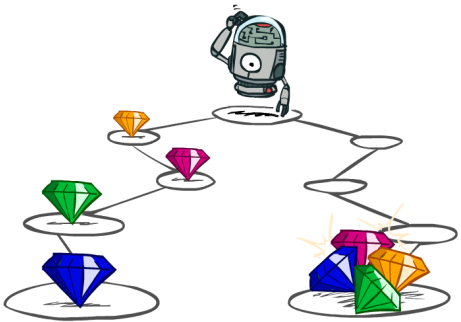
A policy $\pi* : S \rightarrow A$

- Gives us an action for each state.
- Is optimal if it maximizes the expected *utility*.



Utilities
Sequences

- More or less? $[1, 2, 2]$ or $[2, 3, 4]$
- Now or later? $[0, 0, 1]$ or $[1, 0, 0]$
- What about infinite utilities?



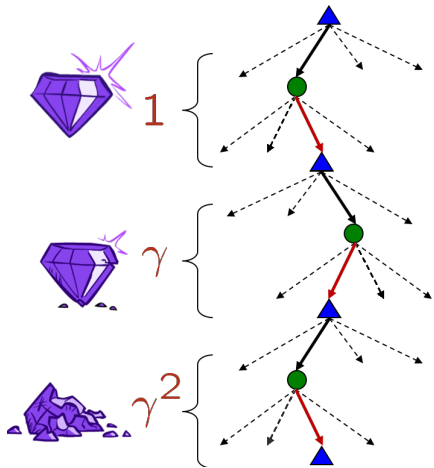
Utilities
Discounting

It is reasonable to:

- Maximize the sum of rewards
- Prefer rewards now to rewards later

How?

- Multiply each reward by a discount factor $0 < \gamma < 1$



Utilities
Stationary Preferences

- If we assume stationary preferences [1]:

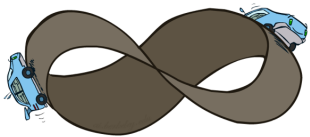
$$[a_1, a_2, \dots] \succ [b_1, b_2, \dots]$$
$$\Updownarrow$$
$$[r, a_1, a_2, \dots] \succ [r, b_1, b_2, \dots]$$

- Then there are only two ways to define utilities
  - Additive: $U([r_0, r_1, r_2, \dots]) = r_0 + r_1 + r_2 + \dots$
  - Discounted: $U([r_0, r_1, r_2, \dots]) = r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$

[1] If a future starting tomorrow is preferred another future, then starting the former future today does not change the preference

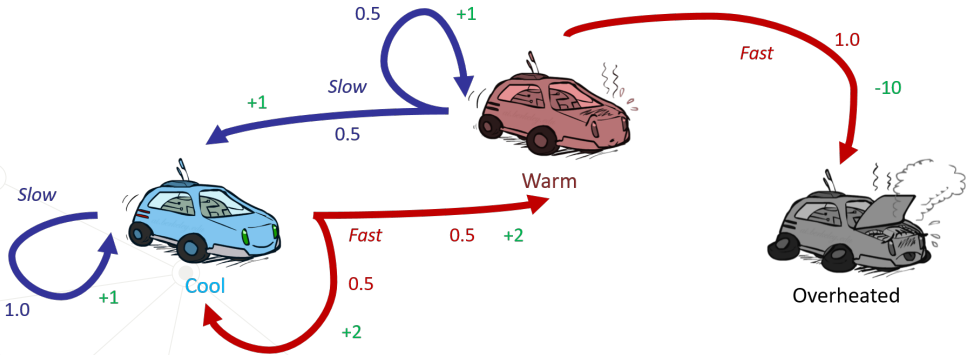
Utilities
Infinite sequences

- What if the game or environment lasts forever?
- Possible solutions:
  - Finite horizon
    - Terminate after a fixed number of time steps.
    - Results in non-stationary policies.
  - Discounting with $0 < \gamma < 1$
    - $U[r_0, \ldots, r_\infty] = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{max}/(1-\gamma)$
  - Absorbing state
    - Guarantee that for every policy, a terminal state will eventually be reached.



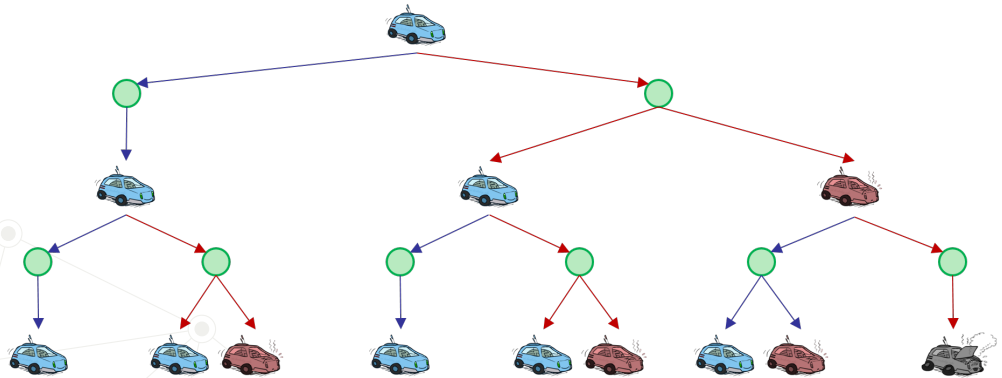
Solving MDPs
The example

- States: Cool, Warm, Overheated
- Actions: Slow, Fast

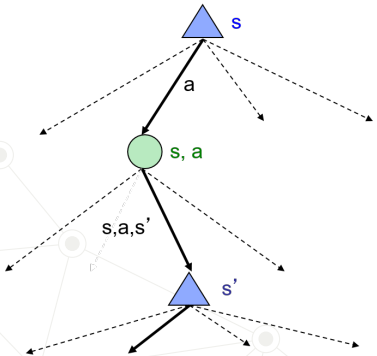


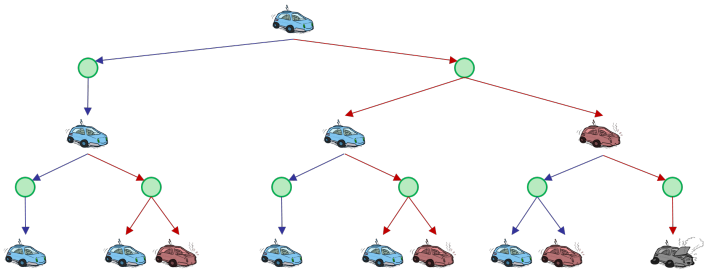
Solving MDPs

Example search tree



Solving MDPs
Example search tree



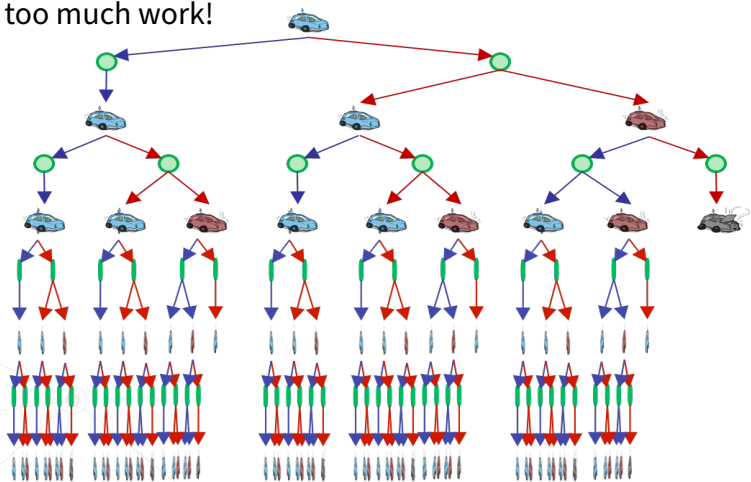


Looks like *Expectimax*!

You can already solve MDPs (not in the most efficient way)

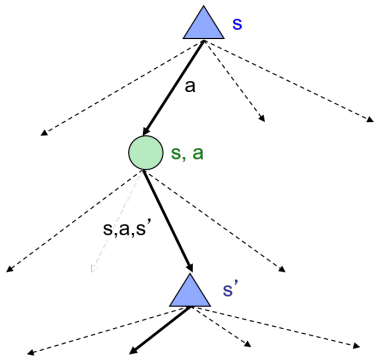
Solving MDPs

We are doing too much work!



Value Iteration
Definitions

- $V^*(s)$: Expected utility starting in s and acting optimally.
- $Q^*(s, a)$: Expected utility starting on s taking action a and thereafter acting optimally.
- $\pi^*(s)$: Optimal action from state s.
- $V_k(s)$: Optimal $utility$ of s if the game/environment ends in k time steps.

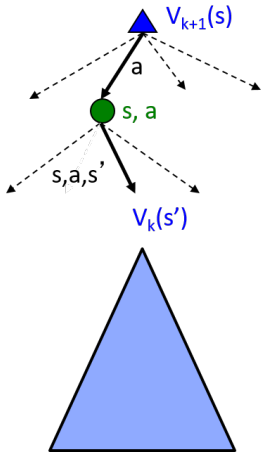


Value Iteration

- Start with $V_0(s) \leftarrow 0$.
- Given V_k do one ply of expectimax:

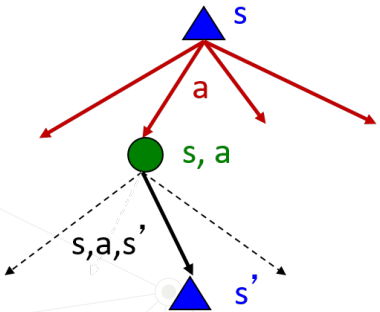
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

- Complexity of each iteration $O(S^2A)$
- It will converge as approximations get refined to optimal values.
- Policy may converge before values do.

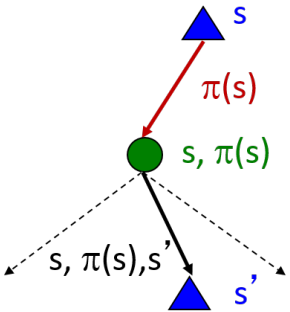


Policy Evaluation
Fixed Policy

Optimal action



Do what π says to do



- With a fixed policy the tree is simpler

Policy Evaluation

How do we calculate the V's?

- $V^{\pi}(s)$: Expected utility starting in s and following π.
- Recursive relation

$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

- Turn the recursive equations into updates like value iteration:

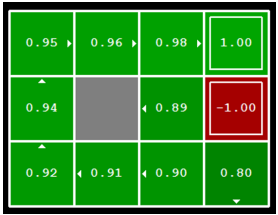
$$V_0^{\pi}(s) \leftarrow 0$$

$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

- Complexity of each iteration: $O(S^2)$

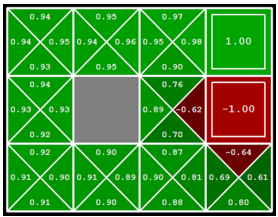
Policy Extraction

We have $V^*(s)$, now what?



- With $V^*(s)$ we have to run one ply to determine the best action:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

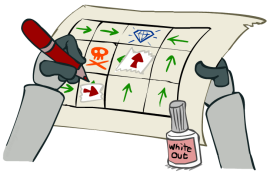


- With $Q^*(s,a)$ is trivial to decide

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

Policy Iteration

- Value Iteration problems
  - It is slow $O(S^2A)$.
  - The *max* of a state rarely changes.
  - The policy often converges before values do.
- Alternative approach *policy iteration*
  - *Policy evaluation:* Compute utilities for a fixed π until convergence.
  - *Policy improvement:* Update π using one-step look-ahead with converged utilities as future values.
  - Repeat until π converges.
- It is still optimal and can converge faster.



Policy Iteration

- Evaluation step (iterate k until it converges):

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s')]$$

- Improvement step (one-step look-ahead policy extraction):

$$\pi_{i+1} \leftarrow \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^{\pi_i}(s')]$$

- Stop when $\pi_{i+1} = \pi_i$.

Comparison

Value Iteration & Policy Iteration

- Both compute optimal values.
- Value iteration:
  - Iterations update the values and implicitly the policy.
  - We do not track the policy, taking the max over actions recomputes it.
  - Each iteration is “slow”.
- Policy iteration:
  - Iteratively updates utilities with a fixed policy (fast)
  - After computing utilities, a new policy is chosen (slow)
  - The new policy is either better or we are done.
- Both are dynamic programs for solving MDPs.

1. Preliminaries

2. Reinforcement Learning
   - Comparison with MDP
   - Model-based learning
   - Model-free learning
   - Temporal Difference Learning

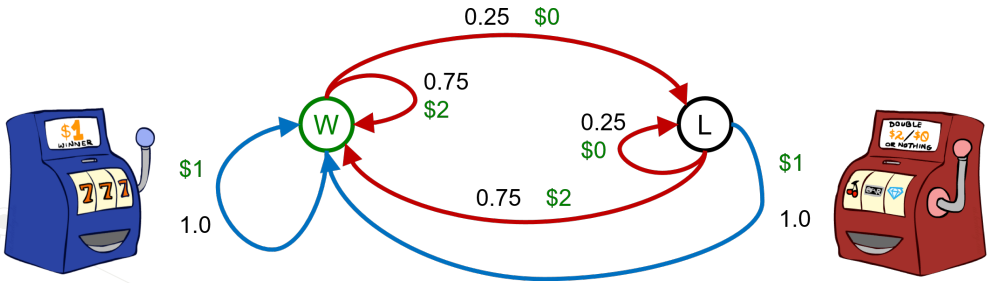
3. Reinforcement Learning Algorithms

4. Extras

5. References

Comparison with MDP

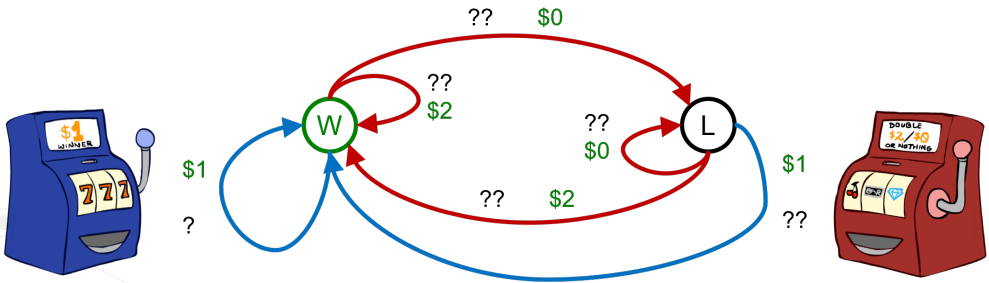
MPD is offline planning



- Solving an MDP is offline planning.
- You need to know the details of the MDP.
- You do not actually interact with the environment.

Comparison with MDP

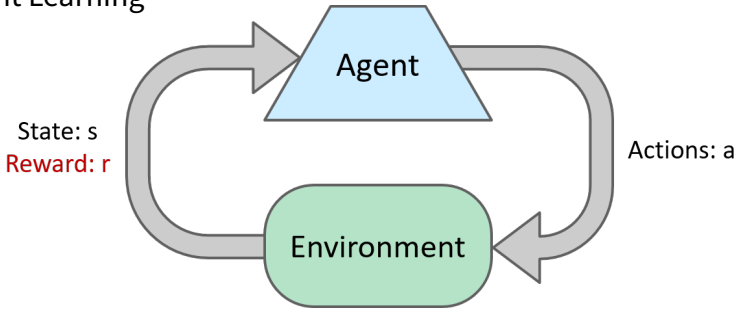
Reinforcement Learning is online planning



- There is an MDP but it has missing data.
- You need to interact with the environment.
- You have to balance Exploration and Exploitation.

Comparison with MDP

Reinforcement Learning



- Receive feedback in the form or rewards.
- Agent's utility is defined by the reward function.
- Agent learns to act in a way that maximizes the expected reward.
- Learning requires interacting with the environment.

Model-based learning

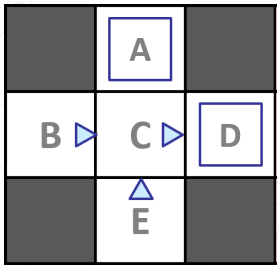
Learn the MDP

- Idea:
  - Learn an approximate model (transitions & rewards).
  - Solve the learned MDP as if the model was correct.
- Learn the empirical MDP model:
  - Register outcomes s' for each s and a.
  - Normalize to give an estimate of $\hat{T}(s, a, s')$.
  - Register $\hat{R}(s, a, s')$ for every experience (s, a, s').
- Solve the empirical MDP:
  - Value Iteration.
  - Policy Iteration.
  - . . .

Model-based learning

Example

Input Policy π



$\gamma = 1$

Observed Episodes

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, -, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, -, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, -, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, -, -10

Learned Model

$\hat{T}(s, a, s')$

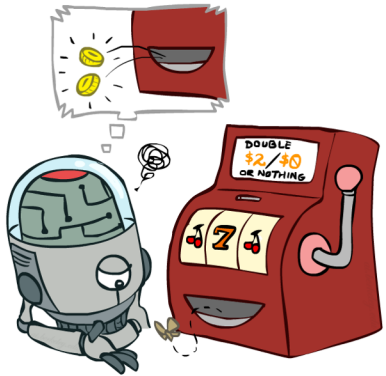
$$T(B, east, C) = 1.0$$
$$T(C, east, D) = 0.75$$
$$T(D, exit, -) = 0.25$$
$$\ldots$$

$\hat{R}(s, a, s')$

$$R(B, east, C) = -1$$
$$R(C, east, D) = -1$$
$$R(D, exit, -) = +10$$
$$\ldots$$

Model-free learning
Direct Evaluation

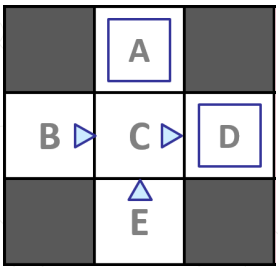
- Given a fixed policy π
  - We do not need $T(s, a, s')$ and $R(s, a, s')$.
  - We can learn $V^{\pi}(s)$.
- Learn by exploring and normalizing.
  - From a state s, acting according to π, register the sum of discounted rewards.
  - Repeat multiple times from each state s and average the samples.



Model-free learning

Direct Evaluation Example

Input Policy π



$\gamma = 1$

Observed Episodes

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, -, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, -, +10

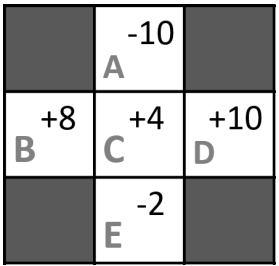
Episode 3

E, north, C, -1
C, east, D, -1
D, exit, -, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, -, -10

Output Values



Model-free learning

Direct Evaluation Pros & Cons

- Pro: Easy to understant and implement.
- Pro: It does not require any knowledge of $T(s, a, s')$ & $R(s, a, s')$.
- Pro: With enough sample transitions it will compute correct average values.
- Con: It does not exploit state connections.
- Con: Each state is learned separately.
- Con: It takes a long time to learn.

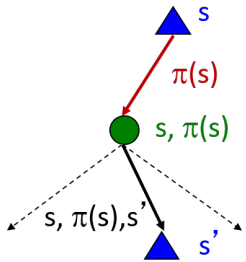
Model-free learning
Can we do better?

- Remember Policy Evaluation?

$$V_0^\pi(s) \leftarrow 0$$

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$

- It exploited connections between the states!

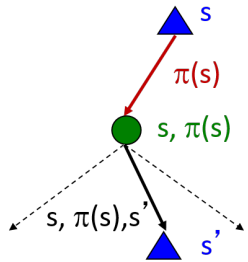


Model-free learning
Can we do better?

- Remember Policy Evaluation?

$$V_0^\pi(s) \leftarrow 0$$

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$



- It exploited connections between the states!
- But for that we need $T(s, \pi(s), s')$ & $R(s, \pi(s), s')$.
- Can we update V without knowing T and R?

Model-free learning

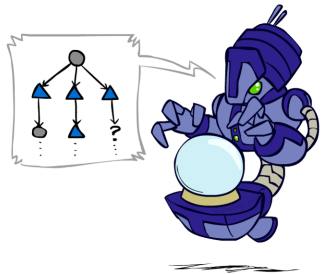
Can we do better?

- We want to improve our estimate of V with the benefits of the one-step-look-ahead strategy.

$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

- Idea: Sample outcomes and average results.

$$sample_1 = R(s, \pi(s), s_1') + \gamma V_k^{\pi}(s_1')$$
$$sample_2 = R(s, \pi(s), s_2') + \gamma V_k^{\pi}(s_1')$$
$$\cdots$$
$$sample_n = R(s, \pi(s), s_n') + \gamma V_k^{\pi}(s_1')$$
$$V_{k+1}^{\pi} \leftarrow \frac{1}{n} \sum_i sample_i$$



Model-free learning

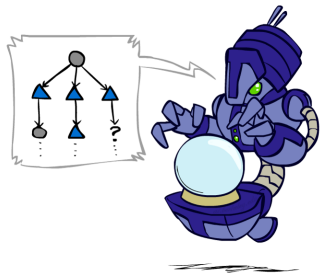
Can we do better?

- We want to improve our estimate of V with the benefits of the one-step-look-ahead strategy.

$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

- Idea: Sample outcomes and average results.

$$sample_1 = R(s, \pi(s), s_1') + \gamma V_k^{\pi}(s_1')$$
$$sample_2 = R(s, \pi(s), s_2') + \gamma V_k^{\pi}(s_1')$$
$$\cdots$$
$$sample_n = R(s, \pi(s), s_n') + \gamma V_k^{\pi}(s_1')$$
$$V_{k+1}^{\pi} \leftarrow \frac{1}{n} \sum_i sample_i$$



We cannot rewind time to get a sample after another from state s.

Temporal Difference Learning

What is it?

- Learn from every experience!
  - Update $V(s)$ each time we experience a transition (s, a, s', r).
  - Likely outcomes s' will contribute updates more often.
- Policy is still fixed, we only evaluate it.
- Move value towards whatever successor occurs: running average.

Sample $V(s)$: $\quad sample = R(s, \pi(s), s') + \gamma V_k^{\pi}(s')$

Update $V(s)$: $\quad V^{\pi} \leftarrow (1 - \alpha)V^{\pi}(s) + \alpha\, sample$

Same update: $\quad V^{\pi} \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

Temporal Difference Learning

Exponential Moving Average

- A moving average smooths out short-term fluctuations and highlights long-term trends.
- Interpolation update: $\bar{x}_n = (1-\alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n$
- Weight of older samples decrease exponentially

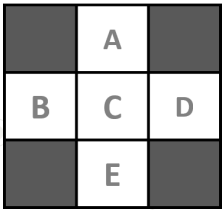
$$\bar{x}_n = \frac{x_n + (1-\alpha) \cdot x_{n-1} + (1-\alpha)^2 \cdot x_{n-2} + \dots}{1 + (1-\alpha) + (1-\alpha)^2 + \dots}$$

- Forgets about past values.
  - Good: Initial values were more likely wrong.

Temporal Difference Learning

Example

States

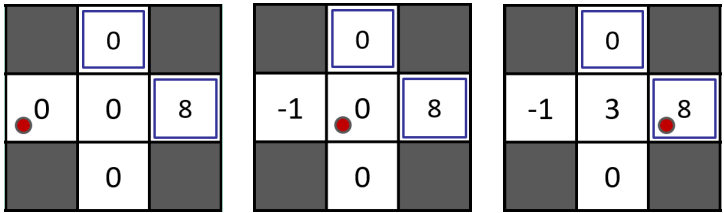


$\gamma = 1, \alpha = \frac{1}{2}$

Observed Transitions

B, east, C, -1

C, east, D, -2



$$V^{\pi} \leftarrow (1-\alpha)V^{\pi}(s) + \alpha[R(s,\pi(s),s') + \gamma V^{\pi}(s')]$$

Temporal Difference Learning

What about a new (better) policy?

- With Temporal Difference learning we got a model-free method to do Policy Evaluation.
- Now lets extract a new policy, remember Policy Extraction?

$$\pi(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Temporal Difference Learning

What about a new (better) policy?

- With Temporal Difference learning we got a model-free method to do Policy Evaluation.
- Now lets extract a new policy, remember Policy Extraction?

  $\pi(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$

  We cannot do it now!

Temporal Difference Learning

What about a new (better) policy?

- With Temporal Difference learning we got a model-free method to do Policy Evaluation.
- Now lets extract a new policy, remember Policy Extraction?

  $\pi(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$

  We cannot do it now!
- Learn Q-values, not values (Can you guess where this is going?)
- This way action selection will be model-free too!

1. Preliminaries

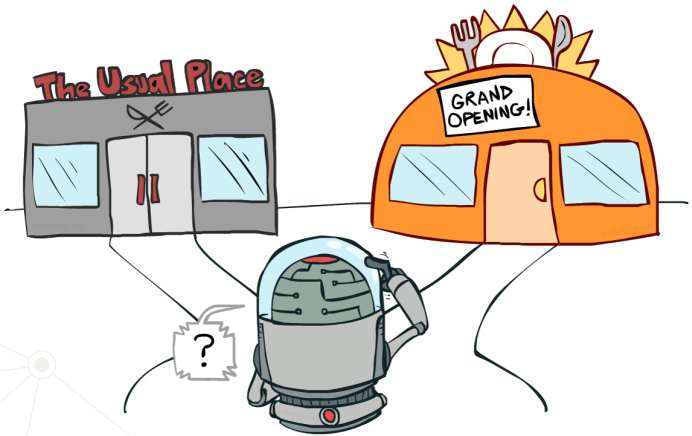
2. Reinforcement Learning

3. Reinforcement Learning Algorithms
   - Exploration vs Exploitation
   - Q-learning
   - Sarsa
   - Approximate Q-learning

4. Extras

5. References

Exploration vs Exploitation



Exploration vs Exploitation

ϵ-greedy

- Probably the simplest method to force exploration: Random Actions.
  - With a small probability ϵ, act randomly.
  - With a large probability $1 - \epsilon$, act according to the current policy π.
- Problems?
  - Eventually you explore all the space, but keep trashing around once learning is done.
  - Solution: lower ϵ over time.

Q-learning
An off-policy TD algorithm

- We would like to update Q-values of each state.

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

  - But we can not do it without knowing T and R.

- Lets do Temporal Difference learning of the Q-values.
  - Take a sample transition (s, a, r, s')
  - The sample suggest

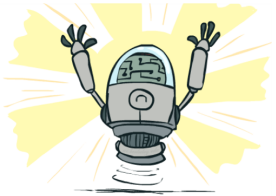
$$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a')$$

  - Compute the moving average:

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

Q-learning
Properties

- Converges to an optimal policy even if you act sub-optimally while learning!
- While learning we do not need to follow the policy, this is called off-policy learning.
  - For example, we can perform random actions and still works.
- Caveats
  - You have to explore enough!
  - Eventually you may have to make the learning rate small enough.



Q-learning

Pseudocode

Require: S set of possible states, A set of possible actions
Require: E Number of episodes, I Maximum number of iterations

Initialize $Q(s, a)\ \forall s \in S, a \in A(s)$ arbitrarily.
Set $Q(terminal, \cdot) = 0$
for $each\ \ episode$ **do**
    $i \leftarrow 0$
    $s \leftarrow initial(S)$
    while $i < I$ and $not\_terminal(s)$ **do**
        $a \leftarrow choose\_action(Q, A, s)$ ▷ e.g: $\epsilon - \text{greedy}$
        $r, s' \leftarrow take\_action(a)$
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
        $s \leftarrow s'$
        $i \leftarrow i + 1$
    end while
end for

Sarsa

An on-policy TD algorithm

- Again, we would like to update Q-values of each state.

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

  - We do not know T and R.
  - We want to avoid potentially *dangerous* optimal paths while learning.

Sarsa

An on-policy TD algorithm

- In Q-learning we use a policy for exploration (i.e ϵ-greedy) and another to update the Q values (a greedy policy).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]]$$

- In Sarsa we use the same policy for exploration and updating the Q values.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]]$$

  - Where a_{t+1} is the action that the exploration policy will take on the next time-step.

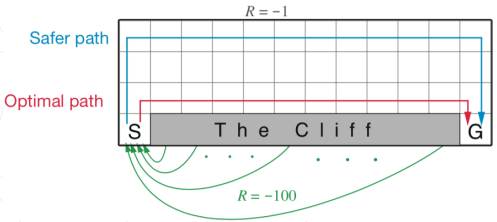
Sarsa
Pseudocode

Require: Same requirements as Q-learning.

Initialize $Q(s, a)\ \forall s \in S, a \in A(s)$ arbitrarily.
Set $Q(terminal, \cdot) = 0$
for $each\ episode$ **do**
    $i \leftarrow 0$
    $s \leftarrow initial(S)$
    $a \leftarrow choose\_action(Q, A, s)$ ▷ e.g: $\epsilon - \text{greedy}$
    while $i < I$ and $not\_terminal(s)$ **do**
        $r, s' \leftarrow take\_action(a)$
        $a' \leftarrow choose\_action(Q, A, s')$ ▷ e.g: $\epsilon - \text{greedy}$
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
        $s \leftarrow s' \quad ; \quad a \leftarrow a'$
        $i \leftarrow i + 1$
    end while
end for

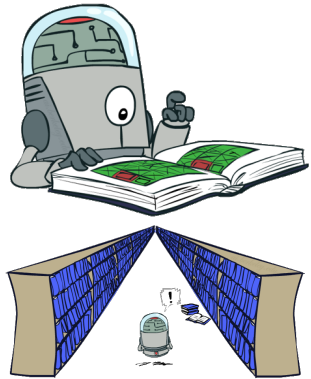
Sarsa
Why Sarsa?

- Q-learning directly learns the optimal policy, whilst SARSA, as is, is near-optimal.
- Q-learning has higher variance than SARSA, which may cause problems converging.
- Sarsa tries to avoid large negative rewards (dangerous situations) while exploring.
  - Great if mistakes are costly.



Approximate Q-learning
Generalizing across states

- Q-learning keeps a table of all Q-values!
- In real environments, we rarely can learn about every state!
  - To many states to visit them all in training.
  - To many (state, action) pairs to hold in memory.
- Let's generalize:
  - Learn with experience from some training states.
  - From this experience we want to learn values that let us make good decisions on unknown states.
  - This is a fundamental idea in machine learning!

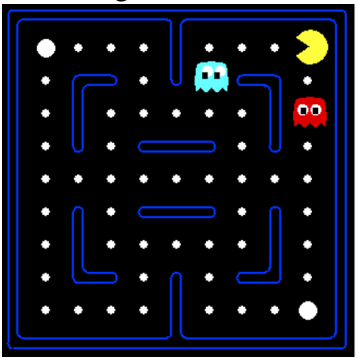


Approximate Q-learning

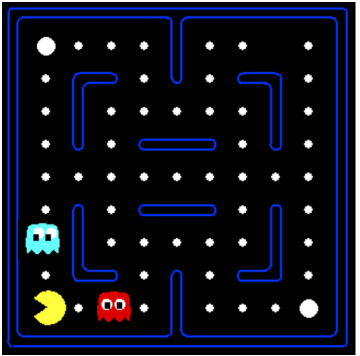
Example: Pacman

We discover that this state is bad.

In naive Q-learning we know nothing about this state:

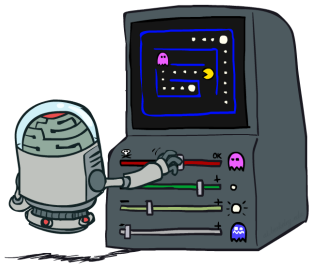


Or this one:



Approximate Q-learning
Feature-based representation

- Describe a state using a vector of features.
  - Features are functions from states to real numbers that capture important properties of the state.
  - Example features:
    - Distance to closest ghost.
    - Distance to closest dot.
    - Number of ghosts.
    - $\frac{1}{(dist\ to\ dot)^2}$
    - Is the agent in a tunnel?
    - . . .
  - We can also describe a Q-state (s, a) with features (e.g. action moves closer to food).



Approximate Q-learning

Linear value functions

- Using feature representation, we can write a Q or V function for any state using a few weights:

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \cdots + w_n f_n(s,a)$$

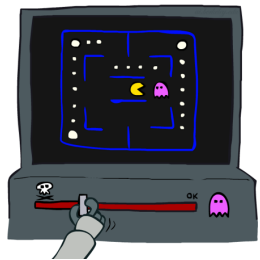
- Advantage: Our experience is summed up in a few *powerful* numbers.
- Disadvantage: States may share features but actually be very different in value!
  - Features may not be representative enough.

Approximate Q-learning

Update $Q(s, a)$ value

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \cdots + w_n f_n(s, a)$$

- Q-learning with linear Q-functions:
  - transition = (s, a, r, s')
  - difference = $[r + \gamma \max_{a'} Q(s', a')] - Q(s, a)$
  - Original Update: $Q(s, a) \leftarrow Q(s, a) + \alpha[\text{difference}]$
- Intuitive interpretation:
  - Adjust weights of active features.
  - If something good/bad happens increase/decrease the weight of that feature proportionally.
- Formal: Gradient descent with least squares error function.

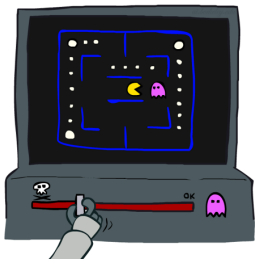


Approximate Q-learning

Update $Q(s, a)$ value

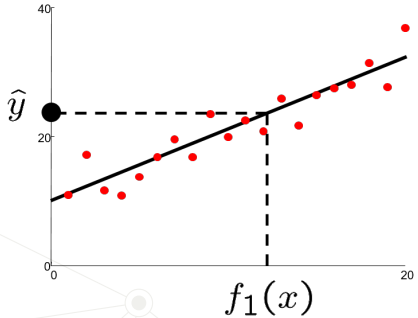
$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \cdots + w_n f_n(s, a)$$

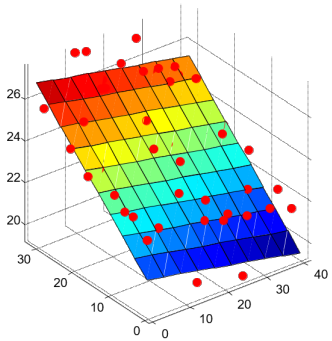
- Q-learning with linear Q-functions:
  - transition = (s, a, r, s')
  - difference = $[r + \gamma \max_{a'} Q(s', a')] - Q(s, a)$
  - New Update: $w_i \leftarrow w_i + \alpha[\text{difference}] f_i(s, a)$
- Intuitive interpretation:
  - Adjust weights of active features.
  - If something good/bad happens increase/decrease the weight of that feature proportionally.
- Formal: Gradient descent with least squares error function.



Approximate Q-learning

Linear approximation: Regression





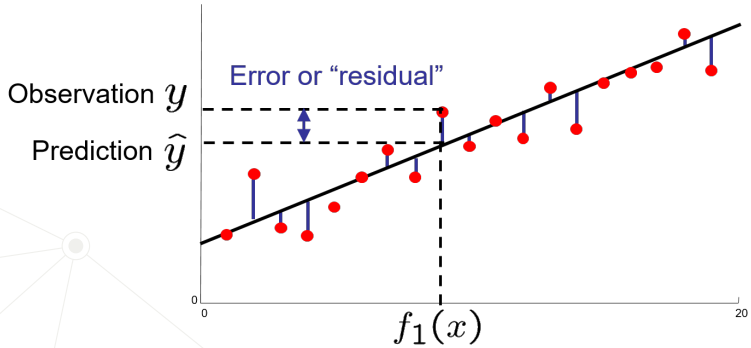
$$\hat{y} = w_0 + w_1 f_1(x)$$

$$\hat{y} = w_0 + w_1 f_1(x) + w_2 f_2(x)$$

Approximate Q-learning

Linear approximation: Optimization with Least Squares

$$\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i (y_i - \sum_k w_k f_k(x_i))^2$$



Approximate Q-learning

Linear approximation: Gradient descent

Assume we had only one point x, with features $f(x)$, target value y, and weights w:

$$\text{error}(w) = \frac{1}{2}(y - \sum_k w_k f_k(x))^2$$

$$\frac{\delta \text{error}(w)}{\delta w_m} = -(y - \sum_k w_k f_k(x))\ f_m(x)$$

$$w_m \leftarrow w_m + \alpha(y - \sum_k w_k f_k(x))\ f_m(x)$$

Replacing the error function variables we get:

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

Gym
The toolkit

Gym is a toolkit for developing and comparing reinforcement learning algorithms. That makes no assumption about the structure of your agent, and is compatible with any numerical computation library.

- It is a collection of test problems (environments).
- With a shared interface, one algorithm works on many environments.

Gym
Installation

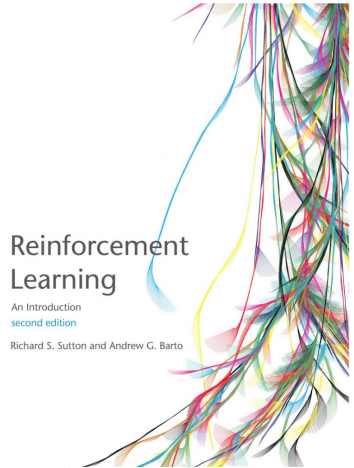
- Basic installation: `pip install gym`
- Atari environments: `pip install gym[atari]`
  - To install the ROMs follow the instructions of the atari-py library: `https://github.com/openai/atari-py`
- Box2D environments: `pip install gym[box2d]`
- MuJoCo environments: `pip install gym[mujoco]`
  - Requires the installation of the MuJoCo library.
  - Dowload it from `https://mujoco.org/download`.
  - Put it in `$HOME/.mujoco/mujoco210`.

Gym
Example

```
1  import gym
2  env = gym.make('CartPole-v0')
3  for i_episode in range(20):
4      observation = env.reset()
5      for t in range(100):
6          env.render()
7          print(observation)
8          action = env.action_space.sample()
9          observation, reward, done, info = env.step(action)
10         if done:
11             print("Episode finished after {} timesteps".format(t+1))
12             break
13 env.close()
```

1. Preliminaries
2. Reinforcement Learning
3. Reinforcement Learning Algorithms
4. Extras
5. References

References



Reinforcement Learning: An Introduction

OpenAI Spinning Up

https://spinningup.openai.com

Deep RL Bootcamp
26-27 August 2017 | Berkeley CA

https://sites.google.com/view/deep-rl-bootcamp