Reinforcement Learning Part I

Volker Tresp
2021

Introduction

- Some researchers distinguish between supervised learning and unsupervised learning
- Supervised learning: We typically have an input $\mathbf{x}$ and an output y and in the training data both are known; in the test data, only $\mathbf{x}$ is known
- In unsupervised learning, only $\mathbf{x}$ is known in training and testing. Maybe a better definition, which would include Bayesian networks, is to say that one does not distinguish between input and output. One might be interested in understanding structure in the data (as in clustering); another case is that in test data different variables can be inputs (known) or output (variable of interest to predict)

Introduction (cont'd)

- But in both cases, the learner is not really smart in the sense that the agent can only learn to describe what is in the data
- In a clinical setting, we might learn to imitate the doctor, but how can we become smarter than the doctor? Reinforcement learning is a learning-based optimization approach, which tries to achieve that
- In reinforcement learning, an agent acts in an environment and receives rewards and punishments as feedback and as a result of its actions
- The agent should learn, based on interactions with the environment, and from rewards (or punishments) to optimize its behavior
- Reward can be delayed (end of a board game)
- Actions may have long term consequences

Introduction (cont'd)

- Parents reward their child for doing something well; the child learns to repeat what is well done (for more rewards to obtain)
- Small children reward and punish their parents with a smile or by uncontrolled screaming; parents learn to optimize their behavior so that rewards are maximized and punishments are minimized; here it is obvious that children only give unspecific signals, and parents are trained, to find out which action is optimal (change diaper, feed, put to sleep, carry around ...)
- Technically we are considering Markov decision processes (MDPs) and we are solving multistage optimization problems
- The first part of the lecture is about the derivation of optimal actions with perfect knowledge about the system one is trying to optimize (keywords are: model-based, planning, dynamic programming)

Introduction (cont'd)

- The second part is about the derivation of optimal actions; you derive the policy by observing the system or a simulation (off-policy) or by experimenting with the system or a simulation (on-policy) or by learning a model from observed data
- In the third part we discuss the role of function approximation (e.g., neural networks) and a number of advanced topics

Literature

- An excellent book and the standard is: An Introduction to Reinforcement Learning, Sutton and Barto (SB)
- Excellent slide set and video: https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver (S); this lecture uses the structure and content of this slide set

Preliminary: Expected Values are Additive

- Consider random variables X and Y with a joint $P(X, Y)$
- We have $\mathbb{E}(X) = \sum_x xP(x)$ and $\mathbb{E}(Y) = \sum_y yP(y)$
- Let $Z = X + Y$; then

$$\mathbb{E}(Z) = \sum_{x,y}(x+y)P(x,y) = \sum_{x,y} xP(y|x)P(x)$$

$$+\sum_{x,y} yP(x|y)P(y) = \mathbb{E}(X) + \mathbb{E}(Y)$$

- So even when X and Y are not independent, expected values are additive; also,

$$\mathbb{E}(X) \approx \frac{1}{N_s}\sum_{i=1}^{N_s} x_i$$

with N_s samples drawn either from $P(X)$ or from $P(X, Y)$

Preliminary: An Important Identity used in Policy Gradient

- We know

$$\frac{\partial \log f_w(x)}{\partial w} = \frac{1}{f_w(x)} \frac{\partial f_w(x)}{\partial w}$$

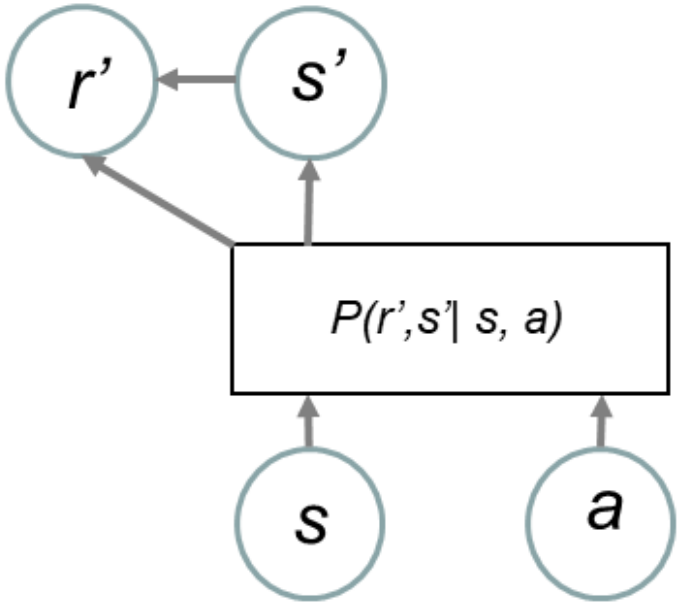
- This implies

$$\frac{\partial f_w(x)}{\partial w} = f_w(x) \frac{\partial \log f_w(x)}{\partial w}$$

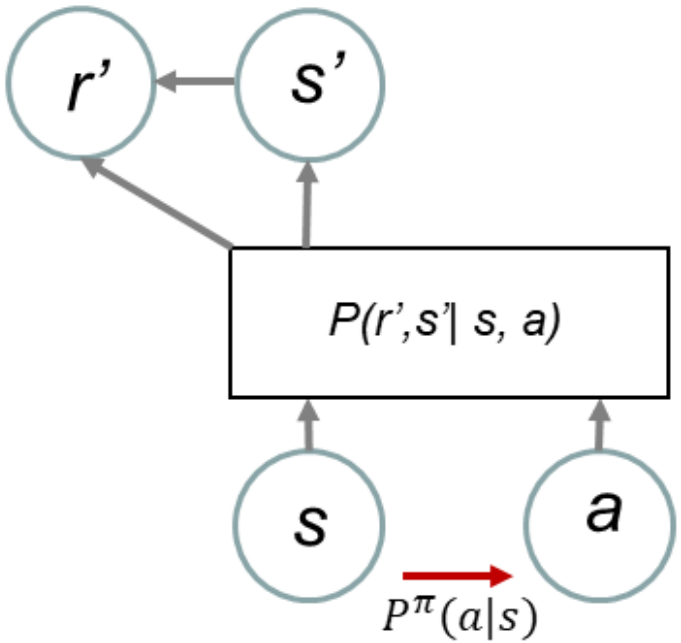
One Stage Problem

- We begin very simple: there is only one action at one instance
- The next figure shows the random variables and their dependencies as a Bayes net
- $S = s$ means that the world is in state s; the state can also represent the history of what had happened to the agent
- $A = a$ means that action a is executed
- $S' = s'$ is the state of the world in the next instance
- $R' = r'$ means that the agent obtained reward r' (e.g., $r' = 1$, or $r' = 0$) "at" s'; the state is labelled by the actual reward value

- Structure relevant for calculating the Q-function (action-value function)



- Structure relevant for calculating the state-value function
- We assume a policy $P^{\pi}(a|s)$



One Stage: Dependencies and Probabilities for Action-Value Setting

- Left: The probability distribution, conditioned on s and a, is

$$P(r', s'|s, a) = P(s'|a, s)P(r'|s', s, a)$$

- $P(s'|s, a)$ is the probability of the next state, given the current state and given that action a is executed; it can be represented by a 3-way array of size $|S| \times |S| \times |A|$

- $P(r'|s', s, a)$ is the probability for reward r' when we observe a transition from s to s' under a; note that we include in our model the possibility, that reward is probabilistic, even when s', s, a is given! it can be represented by a 4-way array of size $|S| \times |S| \times |S| \times |A|$

One Stage: Dependencies and Probabilities for Value Setting

- Right: The probability distribution, conditioned on s, is

$$P^{\pi}(r', s', a|s) = P^{\pi}(a|s)P(s'|a, s)P(r'|s', s, a)$$

- $P^{\pi}(a|s)$ is called the **policy** π (also called control law); a policy can be represented as an array of size $|A| \times |S|$
- If we have a deterministic policy, we write $a = \pi(s)$; an optimal policy is indicated by an asterisk π^*; in fully observed MDPs, optimal policies are deterministic!

One Stage: Reward Function and Q-Function

- We define the **reward function**

$$\mathcal{R}(s,a) = \sum_{s'} \sum_{r'} r' P(r'|s,s',a) P(s'|s,a)$$

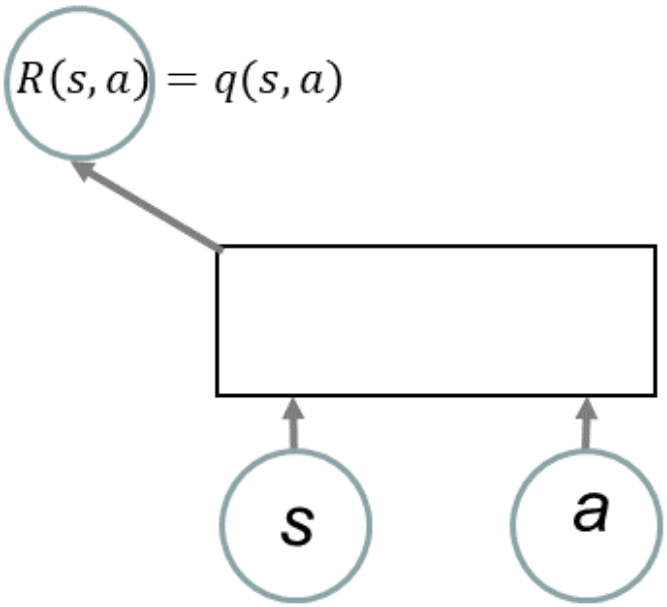
  This is the expected instantaneous reward, when action a is executed in state s; $\mathcal{R}(s,a)$ can be represented by a 2-way array of size $|S| \times |A|$; note that the expected reward at s' is associated with state s

- The **action-value function** (also called **Q-function**) in this case is defined as

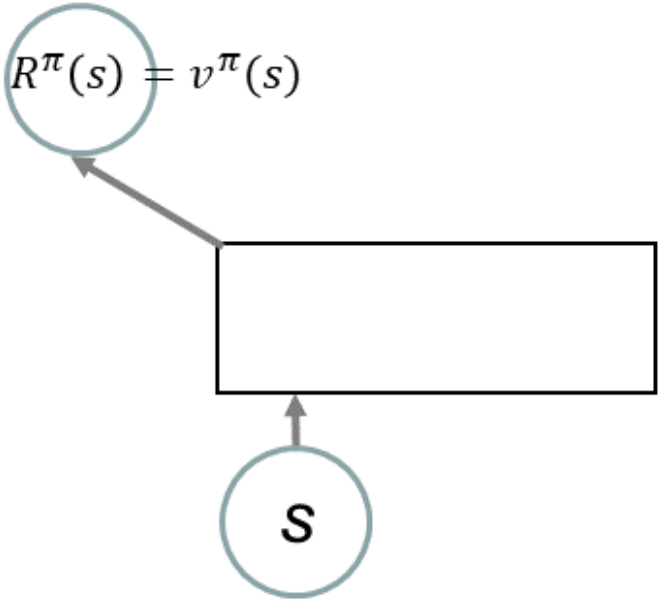
$$q(s,a) \doteq \mathbb{E}(r'|s,a) = \sum_{s'} \sum_{r'} r' P(r'|s,s',a) P(s'|s,a)$$

  It is the expected reward, given that we are in state s and perform action a; in the one-step case, $q(s,a) = \mathcal{R}(s,a)$

- Here the reward function is the Q-function
- We integrate out *s'*



- For a given policy, the reward function is the value-function
- We integrate out *a* and *s'*



One Stage: Value Function

- In addition, we can integrate out the action. We define,

$$\mathcal{R}^{\pi}(s) = \sum_{a} P^{\pi}(a|s) \sum_{s'} \sum_{r'} r' P(r'|s, s', a) P(s'|s, a)$$

  This is the expected instantaneous reward, under policy π; $\mathcal{R}^{\pi}(:)$ can be represented as a 1-way array of size $|S|$; note that $\mathcal{R}^{\pi}(s)$ can be considered a function of s (and not s')

- We define the **state-value function** (also simply called **value function**) as

$$v^{\pi}(s) \doteq \mathbb{E}^{\pi}(r'|s) = \sum_{a} P^{\pi}(a|s) \sum_{s'} \sum_{r'} r' P(r'|s, s', a) P(s'|s, a)$$

  It is the expected reward, given that we are in state s and follow policy π; $v^{\pi}(s)$ can be represented as a 1-way array of size $|S|$; in the one-step case, $v^{\pi}(s) = \mathcal{R}^{\pi}(s)$

One Stage: Relationship Between Action-Value Function and Value Funciton

- We can calculate the value function from the Q-function as

$$v^{\pi}(s) = \sum_{a} P^{\pi}(a|s) q(s,a)$$

One Stage: Optimal Control

- We are now interested to perform the action that gives us the maximum expected reward
- The optimal action for state s is

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

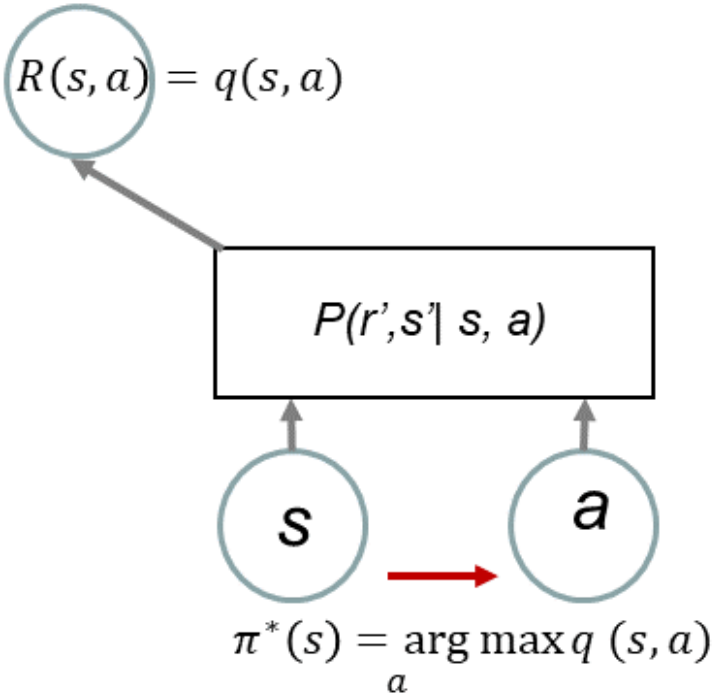
- Thus the **optimal policy** is deterministic with

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

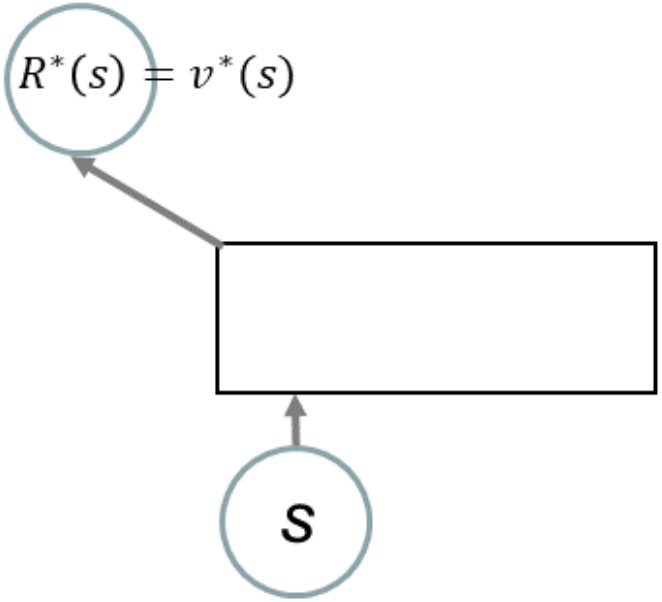
- The **value of the optimal policy** is

$$v^*(s) = \max_a q(s, a)$$

- Optimal policy



- The value function for the optimal policy



Optimizing an Input

- The optimal policy is found by optimizing the Q-function with respect to an input (the action)

One Stage: Contextual Bandits

- One stage reinforcement learning problems (i.e., with unknown models) are called contextual multi-armed bandit problems and are special cases of learning automata
- Contextual: s is observed
- Las Vegas-style slot machines are sometimes called "one-armed bandits"

Two Stages: Dependencies and Probabilities for Action-Value Setting

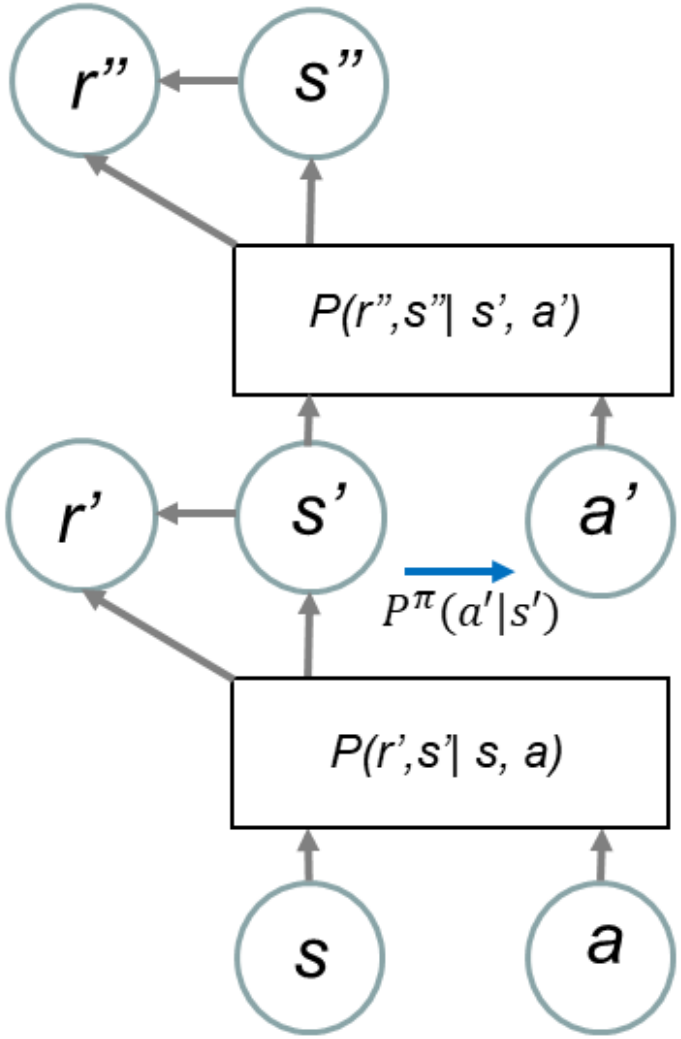
- We now make everything slightly more complex by considering two steps (see figure)
- Left: The probability distribution, conditioned on s and a, is

$$P^{\pi}(r'', r', s'', s', a'|s, a) = P(s'|s, a)P(r'|s', s, a)$$

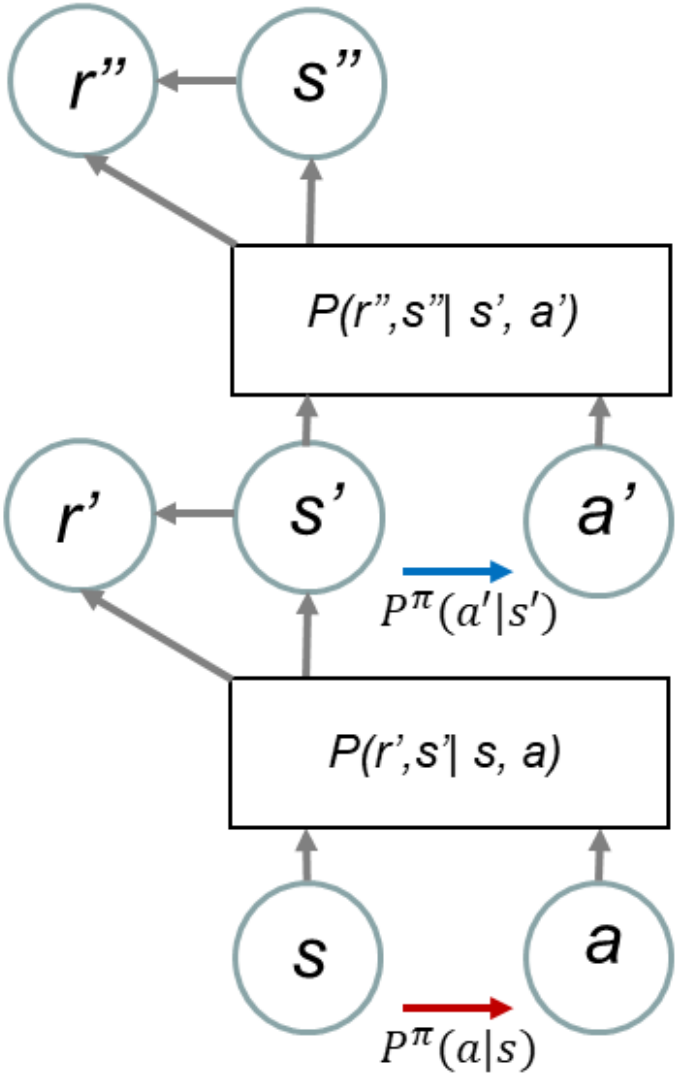
$$\times P^{\pi}(a'|s')P(s''|s', a')P(r''|s'', s', a')$$

- Note that we have a $P^{\pi}(a'|s')$ but NOT a $P^{\pi}(a|s)$

- Structure relevant for calculating the Q-function (action-value function)
- Note: at the second stage a policy is applied



- Structure relevant for calculating the state-value function
- We assume a policy $P^{\pi}(a|s)$



Two Stages: Dependencies and Probabilities for Value Setting

- Right: The probability distribution, conditioned on s, is

$$P^{\pi}(r'', r', s'', s', a', a|s) =$$

$$P^{\pi}(a|s)P(s'|s,a)P(r'|s',s,a)$$

$$\times P^{\pi}(a'|s')P(s''|s',a')P(r''|s'',s',a')$$

- Note that we both a $P^{\pi}(a'|s')$ and a $P^{\pi}(a|s)$

Two Stages: Calculating the Q-Function

- Consider we are in state s'; the Q-function concerns only reward r'' and is

$$q'(s', a') = \mathbb{E}(r''|s', a') = \sum_{s''} \sum_{r''} r'' P(r''|s', s'', a') P(s''|s', a') = \mathcal{R}(s', a')$$

- The expected reward contributed by r'', given initial state s and initial action a is

$$\mathbb{E}^{\pi}(r''|s, a) = \sum_{r''} \sum_{s''} \sum_{a'} \sum_{s'} r'' P(r''|s', s'', a') P(s''|s', a') P^{\pi}(a'|s') P(s'|s, a)$$

$$= \sum_{s'} P(s'|s, a) \sum_{a'} P^{\pi}(a'|s') \; q'(s', a')$$

- Note that this is a backpropagation of the Q-of s' to s

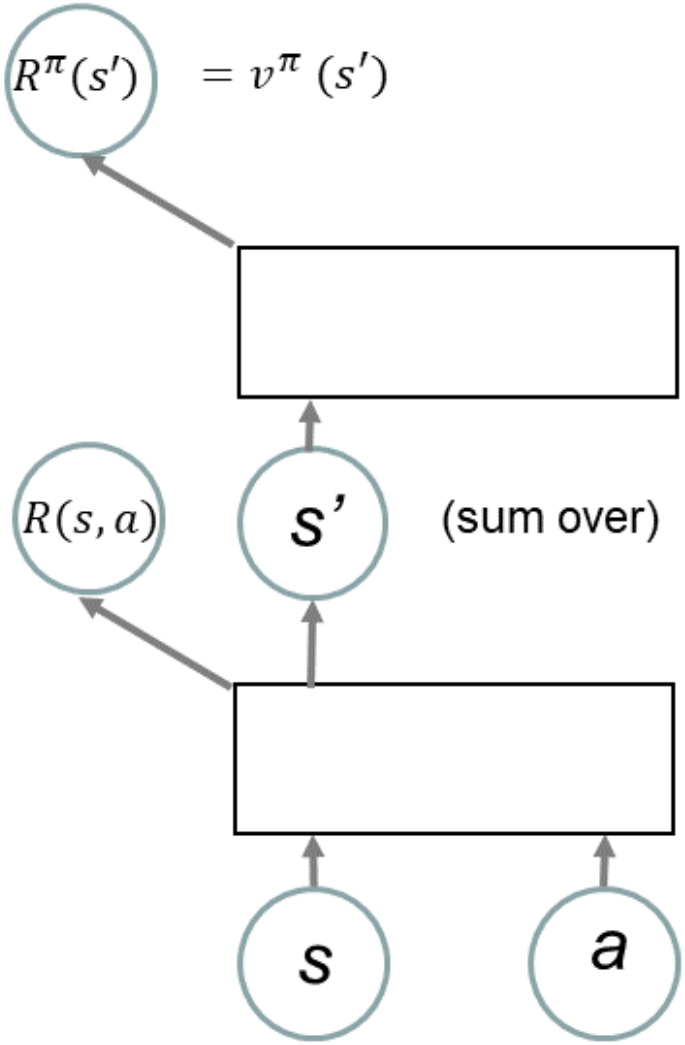
(Q-PE) Two Stages: Policy Evaluation using Action-Value Function

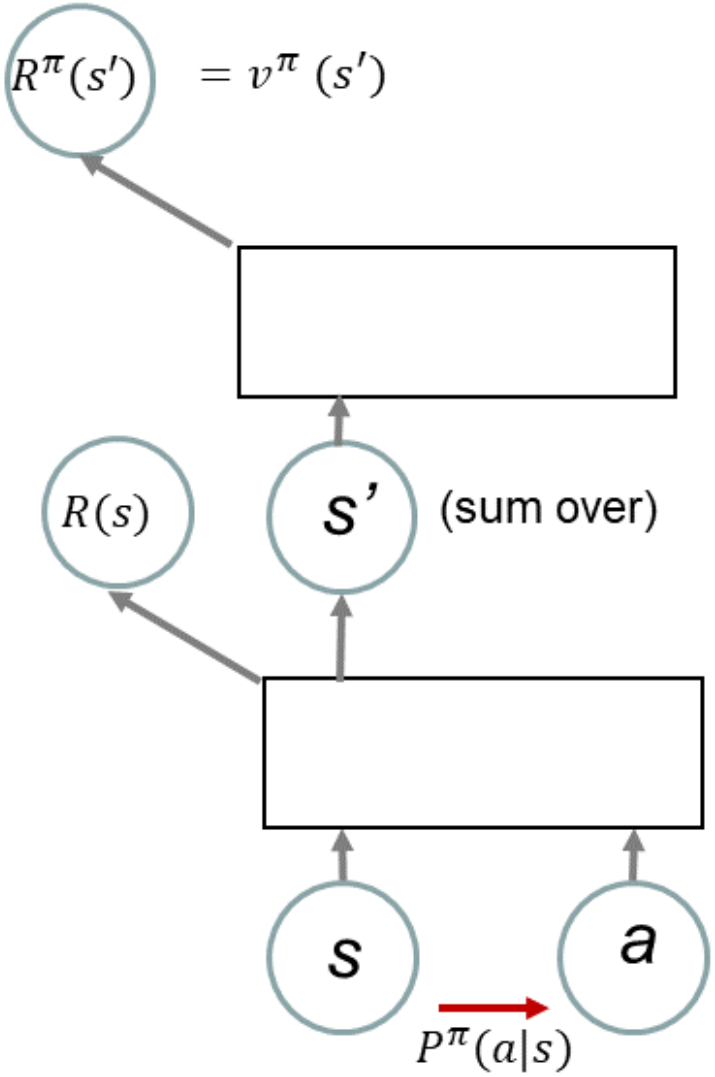
- We are interested in the action a that maximizes the sum of the rewards, and also considers expected reward in the future!
- Q-evaluation 2-step: Finally the **Q-function** at s with contributions from r' and r'' is

$$q^\pi(s,a) = \mathbb{E}^\pi(r' + r''|s,a) = \mathbb{E}^\pi(r'|s,a) + \mathbb{E}^\pi(r''|s,a)$$

$$= \mathcal{R}(s,a) + \sum_{s'} P(s'|s,a) \sum_{a'} P^\pi(a'|s') q'(s',a')$$

  This includes the backpropagation of $q'(s',a')$
- Note that a can be freely chosen, but a' is selected according to policy $P^\pi(a'|s')$


$R^{\pi}(s')$
$= v^{\pi}(s')$
$R(s,a)$
s'
(sum over)
s
a


$R^{\pi}(s')$
$= v^{\pi}(s')$
$R(s)$
s'
(sum over)
s
a
$P^{\pi}(a|s)$

(PE) Two Stages: Policy Evaluation

- The **value function** at s is

$$v^\pi(s) = \sum_a P^\pi(a|s) q^\pi(s,a) = \mathcal{R}^\pi(s) + \sum_{s'} P^\pi(s'|s)\, v'^\pi(s')$$

  Here we have used that

$$P^\pi(s'|s) = \sum_a P^\pi(a|s) P(s'|s,a)$$

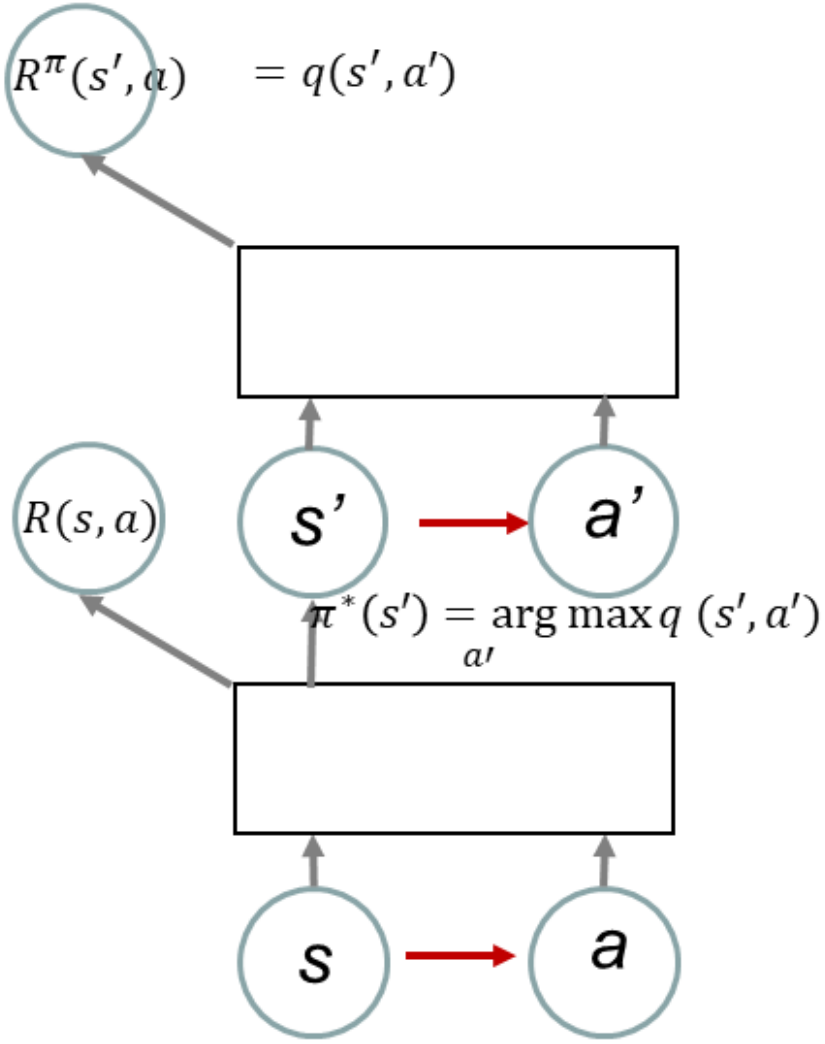
Two Stage: Relationship Between Action-Value Function and Value Function

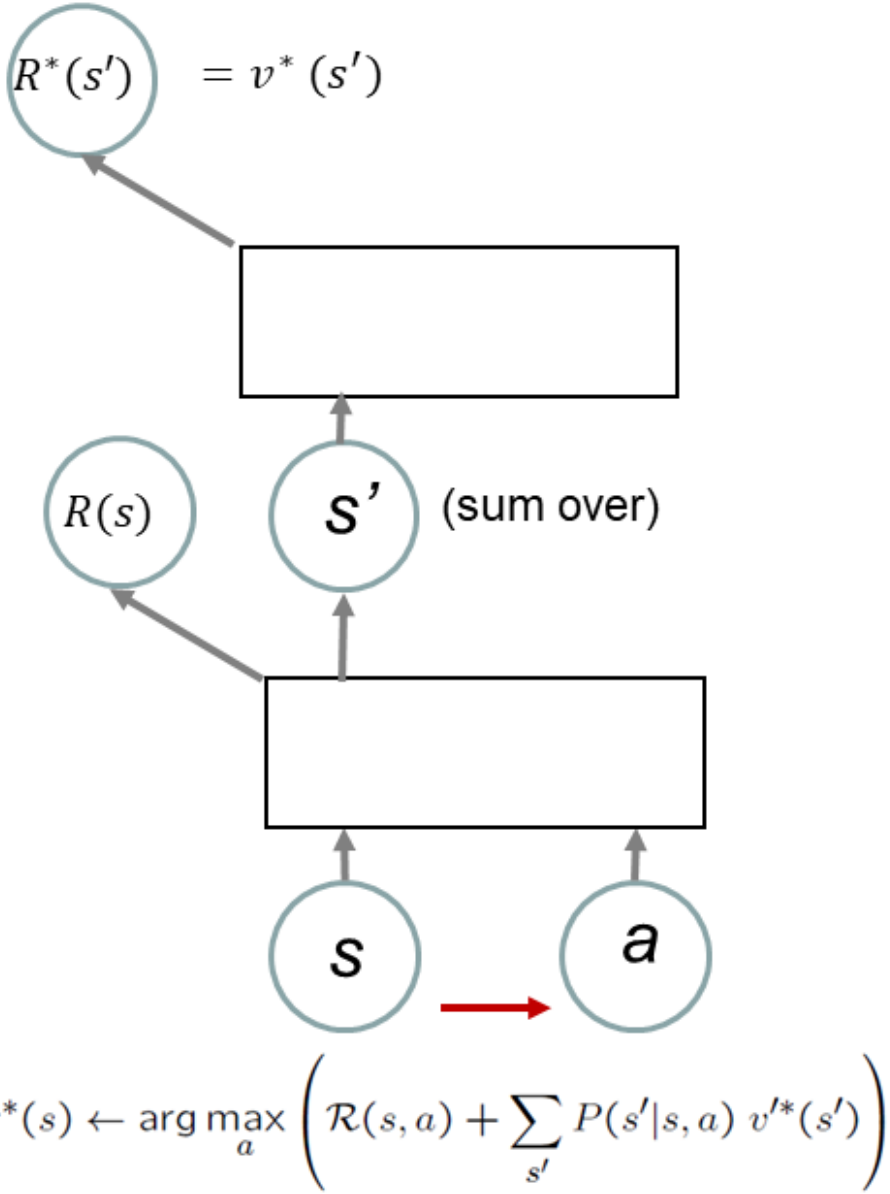
- We can calculate the value function from the Q-function as

$$v^{\pi}(s) = \sum_{a} P^{\pi}(a|s) q^{\pi}(s,a)$$

- We can calculate the value function from the Q-function as

$$q^{\pi}(s,a) = \mathcal{R}(s,a) + \sum_{s'} P(s'|s,a)\, v'^{\pi}(s')$$


$R^{\pi}(s', a)$
$= q(s', a')$
$R(s, a)$
s'
a'
$\pi^*(s') = \arg\max_{a'} q(s', a')$
s
a


$R^*(s')$
$= v^*(s')$
$R(s)$
s'
(sum over)
s
a
$\pi^*(s) \leftarrow \arg\max_a \left(\mathcal{R}(s, a) + \sum_{s'} P(s'|s, a)\, v'^*(s') \right)$

(PI) Two Stages: Policy Iteration to Obtain An Optimal Policy

- The value function of the optimal policy at the second step s' is

$$v'^*(s') = \max_{a'} q(s', a')$$

  Selecting any other action can only decrease the total value

- The optimal policy thus must be,

$$\pi^*(s) \leftarrow \arg\max_a \left(\mathcal{R}(s, a) + \sum_{s'} P(s'|s, a)\ v'^*(s') \right)$$

(VI) Two Stages: Value Iteration to Obtain An Optimal Policy

- The value function of the optimal policy at initial step s is

$$v^*(s) = \max_a \left(\mathcal{R}(s,a) + \sum_{s'} P(s'|s,a)\, v'^*(s') \right)$$

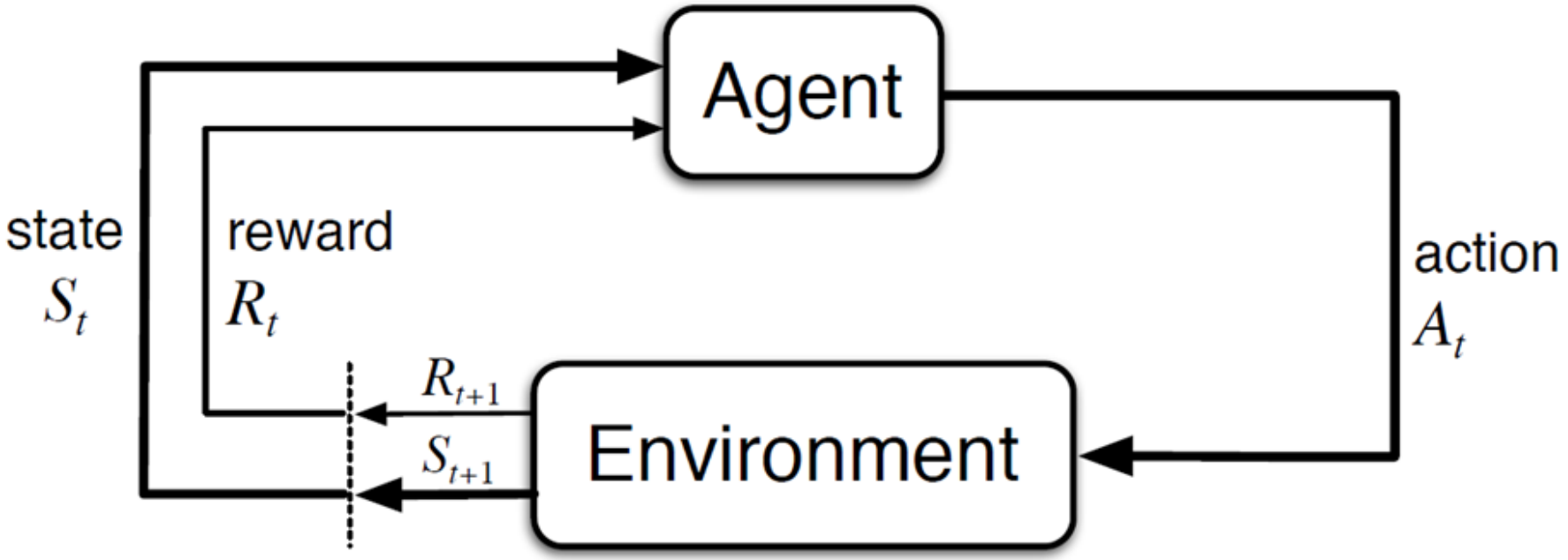
(Q-VI) Two Stages: Action-Value Iteration to Obtain An Optimal Policy

- We also get

$$q^*(s,a) = \mathcal{R}(s,a) + \sum_{s'} P(s'|s,a) \max_{a'} q'(s',a')$$

Infinite Horizon

- Note that we essentially obtained a structure suitable for Dynamic Programming; to make a decision *now*, we need information from the *future*!
- We are interested in infinite horizon problems
- We are looking for a time-invariant solution; then the value function and the Q-function are independent of the instance (in particular $v(\cdot) = v'(\cdot)$ and $q(\cdot) = q'(\cdot)$)
- We introduce discount factor γ to reduce the influence of events far in the future $0 \leq \gamma \leq 1$; in other words, the agent's decision now is a bit more optimizing immediate reward than a reward far in the future


Agent
state
S_t
reward
R_t
action
A_t
R_{t+1}
S_{t+1}
Environment

Bellman Equation

- Bellman equation for the value function is

$$v^{\pi}(s) = \mathcal{R}^{\pi}(s) + \gamma \sum_{s'} P^{\pi}(s'|s) v^{\pi}(s')$$

- In vector notation

$$\mathbf{v}^{\pi} = \mathbf{r}^{\pi} + \gamma P^{\pi} \mathbf{v}^{\pi}$$

- The leads to the explicit solution

$$\mathbf{v}^{\pi} = (I - \gamma P^{\pi})^{-1} \mathbf{r}^{\pi}$$

- Direct inversion is almost never used; computational complexity scales as $O(|S|^3)$; consider that $|S|$ can be quite large!

Bellman Equation (cont'd)

- To some degree, the remaining part of the lecture is about methods which calculate the value function and the Q-function more efficiently:
  - Iterative solutions to the Bellman equation (in the remaining of Part I)
  - Replacing the model probabilities by simulation (Part II)
  - Approximating the value function, the Q-function and the policy by function approximators (e.g., neural networks) (Part III)

(PE) Policy Evaluation

- As an immediate generalization of the 2-stage case, we get with fixed policy π

$$v^{\pi}(s) \leftarrow \mathcal{R}^{\pi}(s) + \gamma \sum_{s'} P^{\pi}(s'|s)\, v^{\pi}(s')$$

- Note that this is an update equation: pick any s and update $v^{\pi}(s)$ using this equation! One should be smart about with s to pick in which order!
- In the Appendix we provide some background on-policy evaluation

(Q-PE) Policy Evaluation using Action-Value Function

- As a generalization to the 2-stage case, we can update the Q-function

$$q^\pi(s,a) \leftarrow \mathcal{R}(s,a) + \sum_{s'} P(s'|s,a) \sum_{a'} P^\pi(a'|s')\, q^\pi(s',a')$$

- As a reminder: we continue to assume that the system $P(s'|s,a)$ is known and does not need to be learned!
- Note that this is an update equation: pick any s, a and update $q^\pi(s,a)$ using this equation! One should be smart about with s, a to pick in which order!

Relationship Between Action-Value Function and Value Function

- We can calculate the value function from the Q-function as

$$v^{\pi}(s) = \sum_{a} P^{\pi}(a|s) q^{\pi}(s,a)$$

- We can calculate the value function from the Q-function as

$$q^{\pi}(s,a) = \mathcal{R}(s,a) + \sum_{s'} P(s'|s,a)\ v^{\pi}(s')$$

(PI) Policy Iteration to Obtain An Optimal Policy

- Policy iteration
  - We calculate $v^{\pi}(s)$ with policy evaluation (as discussed) until some form of convergence is reached
  - Then we calculate a new deterministic policy as

$$\pi_{new}(s) \leftarrow \arg\max_{a} \left(\mathcal{R}(s,a) + \gamma \sum_{s'} P(s'|s,a)\, v^{\pi}(s') \right)$$

- The two steps are repeated until convergence
- Note that in the second step, we only go through all states s once

(Q-PI) Q-Policy Iteration to Obtain An Optimal Policy

- Policy iteration
  - We calculate $q^{\pi}(s, a)$ with Q-PE(as discussed) until some form of convergence is reached
  - Then we calculate a new deterministic policy as

$$\pi_{new}(s) \leftarrow \arg\max_{a} \left(q^{\pi}(s, a)\right)$$

- The two steps are repeated until convergence
- Note that in the second step, we only go through all states s once

(VI) Value Iteration to Obtain An Optimal Policy

- In generalization of the two-stage case, we get

$$v(s) \leftarrow \max_a \left(\mathcal{R}(s,a) + \gamma \sum_{s'} P(s'|s,a) v(s') \right)$$

  We iterate repeatedly over all s in some order until convergence.

- No policy needs to be stored; convergence can be faster than in (PI)

(Q-VI): Action-Value Iteration to Obtain An Optimal Policy

- We also get

$$q(s,a) \leftarrow \mathcal{R}(s,a) + \sum_{s'} P(s'|s,a) \max_{a'} q(s',a')$$

  We iterate repeatedly over all s, a in some order until convergence.

- No policy needs to be stored; convergence can be faster than in (Q-PI)

Optimal Policy

- The optimal policy is deterministic, with

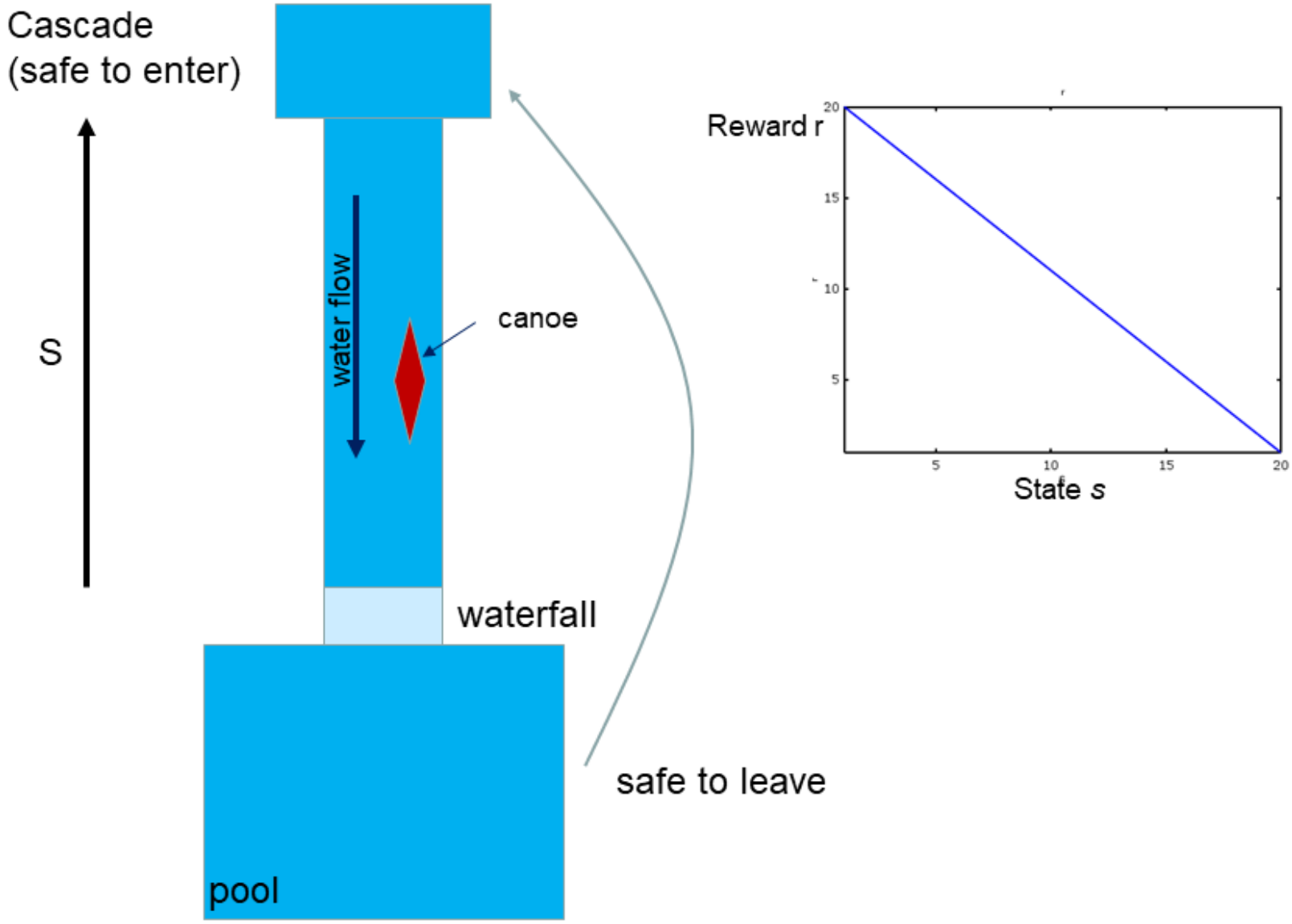
$$\pi^*(s) \leftarrow \arg\max_a q^*(s,a) = \arg\max_a \left(\mathcal{R}(s,a) + \gamma \sum_{s'} P(s'|s,a) v^*(s') \right)$$

A Wet Game of Chicken (Tresp, 1994)

- The agent is a paddler in a canoe
- The agent can just float in which case in **drifts** towards the waterfall ($a = -1$); the agent can try to **balance** the drift ($a = 0$); the agent can **paddle** upstream ($a = 1$)
- The agent get's a larger reward closer to the waterfall
- If the agent's goes down the waterfall it has to get up to the cascade and can again enter the river
- We have

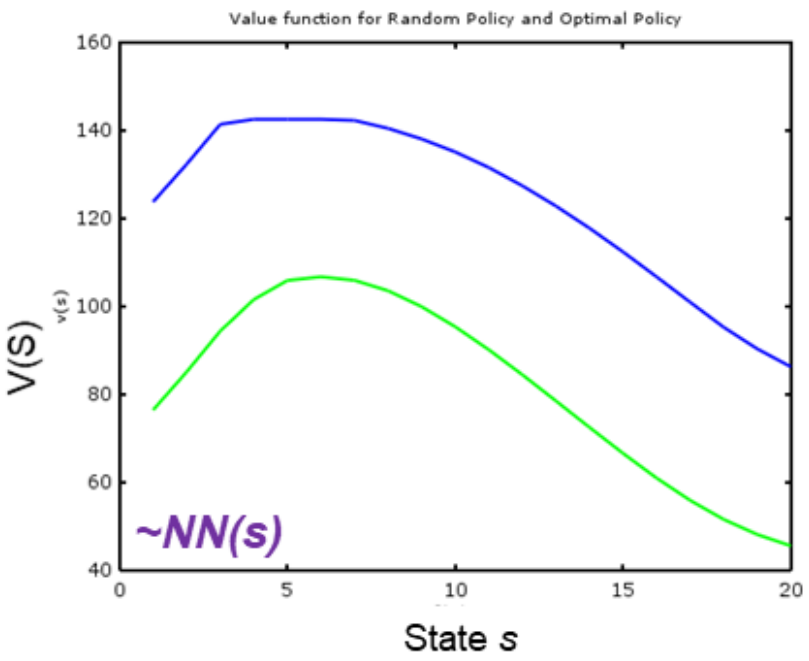
$$s_{t+1} = s_t + round(a_t + \epsilon_t)$$

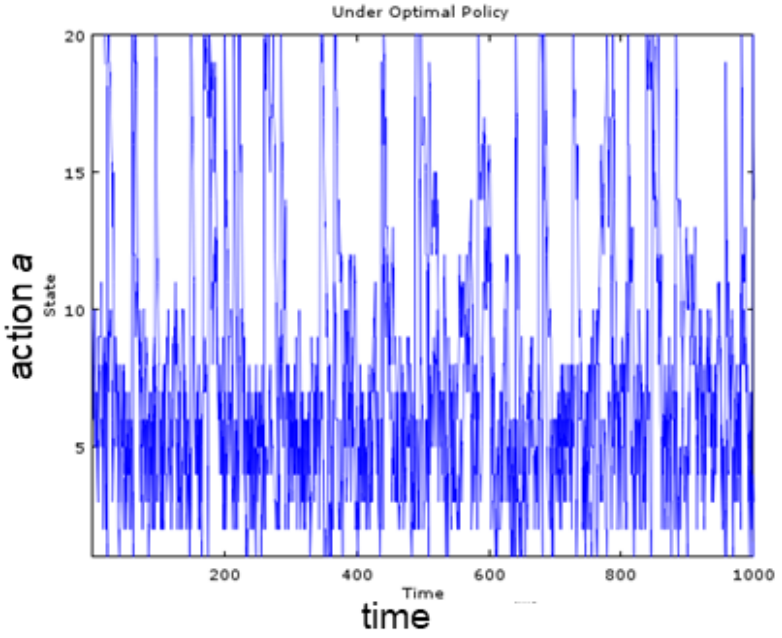
If $s_t \leq 1$, then $s_t = 20$; ϵ is independent Gaussian noise

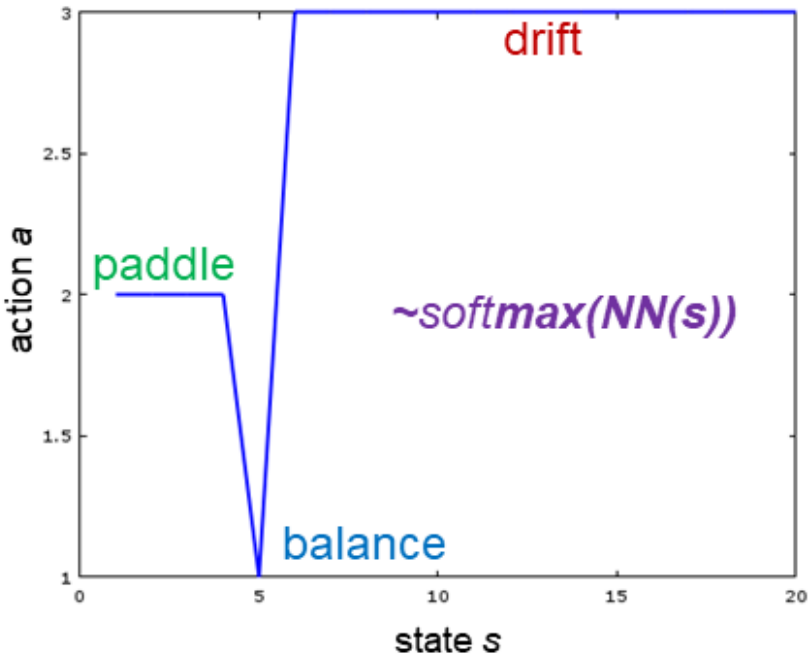

Cascade
(safe to enter)
S
water flow
canoe
waterfall
pool
safe to leave
Reward r
State s
20
15
10
5
5
10
15
20

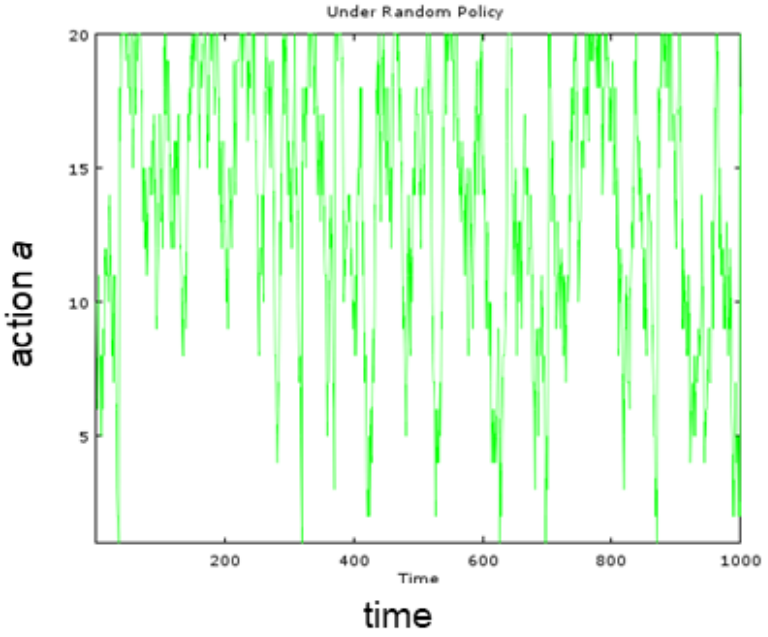
A Wet Game of Chicken (cont'd)

- We see the optimal policy: drift if the agent is far away from the waterfall, paddle, if $s < 5$ and try to balance when $s = 5$
- If compared to a random policy, the value function of the optimal policy is much larger and the agent spends more time close to the waterfall to collect more reward


Value function for Random Policy and Optimal Policy
V(S)
v(s)
State s
~NN(s)

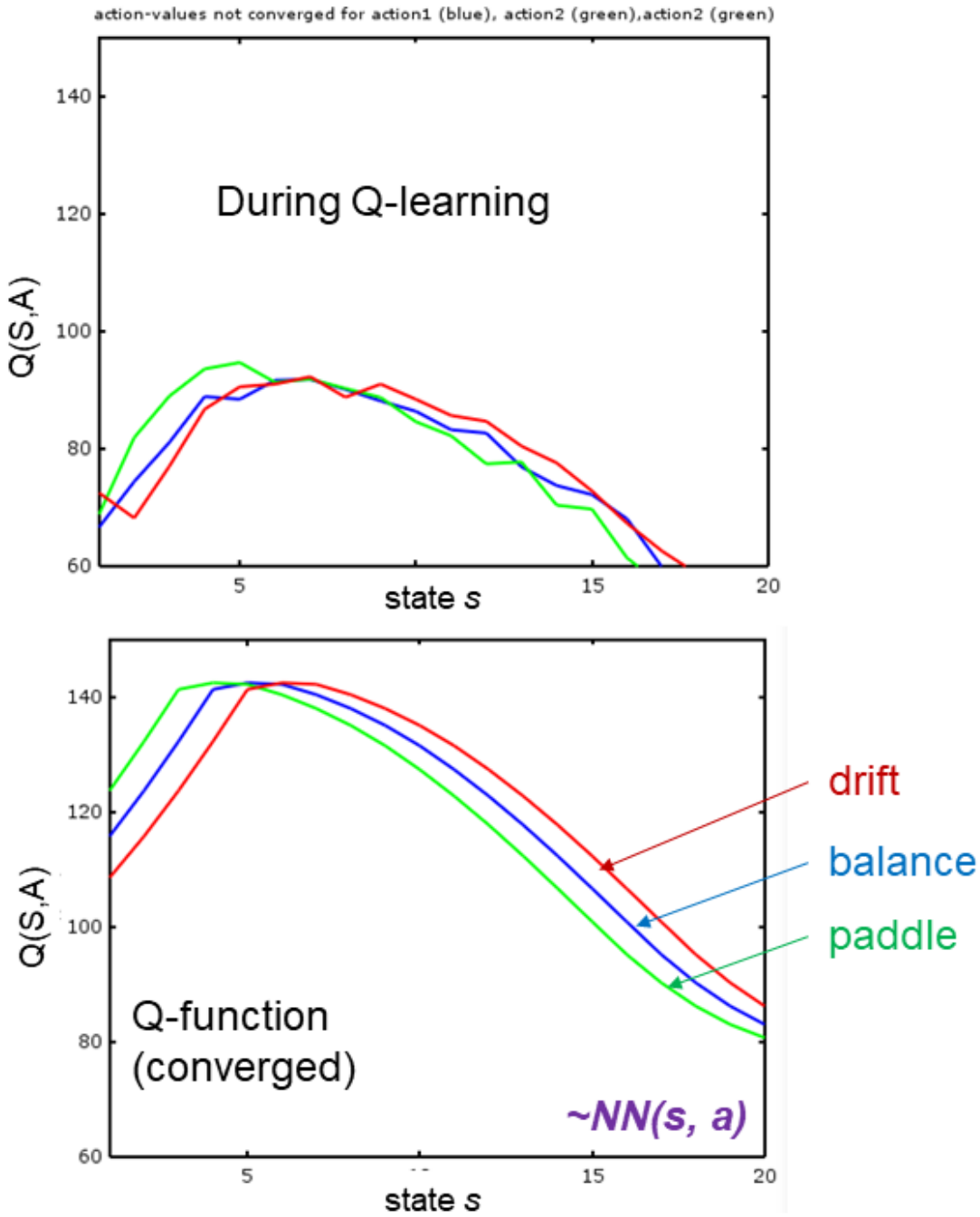

Under Optimal Policy
action a
State
Time
time


drift
paddle
~softmax(NN(s))
balance
action a
state s


Under Random Policy
action a
Time
time

A Wet Game of Chicken (cont'd)

- The Q-function during iterations and after convergence
- Paddling leads to a higher value when close to the waterfall


action-values not converged for action1 (blue), action2 (green),action2 (green)
During Q-learning
Q(S,A)
state s
Q-function
(converged)
~NN(s, a)
drift
balance
paddle

Summary

- Evaluation/prediction
  - (PE) Policy evaluation with value function or Q-function
- Optimal control
  - (PI) Policy iteration with value function or Q-function
  - (VI) Value iteration with value function or Q-function

Discussion

- So far, we covered S, lectures 1-3
- The discusses approaches were **model based**: we assumed that models $P(s'|s,a)$ and $R(\int, a)$ for the system are known; recall that $P(s'|s,a)$ is one $|S| \times |S|$ matrix for each action a and that $R(\int, a)$ is one vector for each a
- Classically the algorithms were derived in the field of optimal control and **dynamic programming**; Silver calls this also **planning**
- It might be unusual to consider an infinite future; consider a finite board game; after the end of the game, formally the states simply do not change anymore

Discussion (cont'd)

- For these algorithms to work, $v(s)$ or $q(s, a)$ must be well estimated for all relevant states s; there are different strategies:
- Which states are updated in which order? Here algorithms differ: there are **synchronous** updates (iterate through all s) or **asynchronous** updates; a special case is **prioritized sweeping** which prioritizes states with large error; other approaches are **real-time dynamic programming** and full width backup
- The iterations are related to the Jacobi method and the Gauss–Seidel algorithm for solving systems of linear equations (see Appendix)
- In **approximate dynamic programming** the value function is approximated by function approximation (more on this later)
- The iterative algorithms discussed so far used **bootstrapping:** using the current value function (resp., Q-function) estimate instead of the true ones on the right side of the updates

- We now discuss model-free approaches where the models $P(s'|s,a)$ and $\mathcal{R}(s,a)$ are not given; we start with model-free evaluation (prediction) and then discuss model-free control

Reinforcement Learning Part II

Volker Tresp
2021

Model Free Prediction/Evaluation

- In many important applications, a probabilistic model is not available but we can generate data from the real system (real world) or a simulation of the real world; this is the **model-free** case
- Also, some of the discussed algorithms involve the multiplication of a vector with a transition probability matrix, which is costly, and should be avoided by only exploring relevant regions in state space
- We consider Monte Carlo (MC) methods and temporal difference (TD) learning
- This is covered in S, lecture 4

Strategies/Overview

- **Evaluation**: Estimate the value function or the Q-function from a simulation ("experience") (TD(0), SARSA-PE)
- **Optimization:** learn an optimal policy (SARSA-algorithm, Q-learning)
- In part II we focus on table representations; in part III we will consider function approximation (neural networks)
- **Model the system** and then apply previous approaches (covered in part III) (e.g., Dyna)
- Directly optimize the policy (controller): **direct policy search**; e.g., policy gradient (in part III); in combination with Q-learning: Actor-Critic algorithm
- **Learn from an existing controller** (human): imitation learning and inverse reinforcement learning

Off-policy and On-policy

- Behavior policy: the policy actually used in the simulation (SB, p. 103); states and rewards are observed by using the behavior policy
- Target policy: the policy we are optimizing; this is the policy we are improving which might not be the one actually used in the simulation
- On-policy: target policy = behavior policy; TD(0) and SARSA-PE and the SARSA-algorithm are on-policy methods
- Off-policy: target policy $\neq$ behavior policy; Q-learning is off-policy; importance sampling allows other approaches to be used off-policy as well (see Appendix)

Offline and Online

- Offline means: Collect data in a simulation and then do the modelling and optimization offline; policy evaluation and Q-learning can, in principle, be used offline (biological aspect: some argue that an episodic memory stores data for offline learning (during sleep?)
- Online means: The updates use the data from the actual simulation without storing the latter; online approaches are often considered biologically more plausible
- Learning from episodes: An episode might be one game of chess; often several episodes are available for training. In an episode, the actual sequence must be stored in order. In online learning one actually follows the sequence; in offline learning, one might randomize the order and apply SGD; if one uses recurrent neural networks or eligibility traces, also learning needs to follow the temporal sequences of the episodes

(MC-PE) Monte Carlo for PE

- Goal: Estimate the value function of an unknown MDP (under a fixed policy π)
- In its simplest form, we simulate the system and we register when state s is observed and calculate the discounted reward T steps in the future (or until the end of a game); let's assume that state s was observed at time t; we define the **return** (see S) as

$$G_t(s) = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{T-1} r_T$$

$G_t(s)$ is the actual discounted reward, when we start from state s. Recall, that the value function is the expected return

$$v(s) = \mathbb{E}(G_t(s))$$

(MC-PE) Monte Carlo for PE (cont'd)

- We then update

$$S(s) \leftarrow S(s) + G_t(s) \text{ and } N(s) \leftarrow N(s) + 1$$

and eventually

$$V^\pi(s) \approx \frac{S(s)}{N(s)}$$

$N(s)$ is the number of times that state s was visited

- We follow S and use capital letters $V^\pi(s)$ and $Q^\pi(s)$ for quantities derived from data and lowercase letters $v^\pi(s)$ and $q^\pi(s)$ for quantities derived from the (true) model

(MC-PE) Monte Carlo for PE (cont'd)

- Note that r_{t+1} is the actual observation of the reward variable at time $t+1$ (see our early figures) and is not an expected value
- MC needs episodic data (data from observed sequences)
- MC methods have zero bias but high variance; thus convergence might be slow; it does not exploit bootstraping
- Robustness: Also works when Markov property is violated
- *The simulation must be on-policy (i.e., with a policy π) but can be offline (with one fixed huge simulation)*

(TD-PE) Temporal Difference

- If at time t, the system was in state s in the simulation, and in time $t+1$ in state s':

$$V(s) \leftarrow \frac{N(s)}{N(s)+1}V(s)+\frac{1}{N(s)+1}G(s') = V(s)+\frac{1}{N(s)+1}(G(s')-V(s))$$

- Now we can approximate

$$G(s') \approx r' + \gamma V(s')$$

which gives us the TD(0) update

$$V(s) \leftarrow V(s) + \eta([r' + \gamma V(s')] - V(s))$$

with learning rate η

(TD-PE) Temporal Difference (cont'd)

- We can also start from our earlier derivation
- Temporal-difference learning (Sutton, 1988)
- Recall that policy evaluation uses

$$v(s) \leftarrow \mathcal{R}^{\pi}(s) + \gamma \sum_{s'} P^{\pi}(s'|s)\, v(s')$$

- The right side can be written as

$$\sum_{a} P^{\pi}(a|s) \sum_{s'} P(s'|s,a) \sum_{r'} P(r'|s,s',a) \left[r' + \gamma v(s')\right]$$

- An unbiased sample from $P^{\pi}(a|s)P(s'|s,a)P(r'|s,s',a)$ leads to the the so-called TD target, $[r' + \gamma V^{\pi}(s')]$

TD(0)

- TD(0) (as an approximation to (PE)) becomes

$$V(s) \leftarrow V(s) + \eta([r' + \gamma V(s')] - V(s))$$

- The term $([r' + \gamma V(s')] - V(s))$ is called the TD error; η is a learning rate
- The observed data from the simulation can be presented in random order (SGD); a data point is the triple $s_t = s, s_{t+1} = s', r_{t+1} = r'$
- TD(0) is used on-policy and can be used offline
- TD(0) can also be used online, where updates are done following the order in the simulation: this is considered biologically more plausible

TD(0) (cont'd)

- TD uses bootstrapping
- Has low variance but some bias
- "Backup": denotes how far the future is explored (TD(0):1, MC: T)
- The simple TD approach generalizes to TD(λ), with $0 \leq \lambda \leq 1$; our discussed algorithm corresponds to TD(0), whereas TD(1) is similar to MC; TD(λ) includes eligibility traces (not covered in the lecture)

(Q-PE) Sarsa Policy Evaluation

- SARSA policy evaluation uses the same idea, but estimates the Q-function (Q-PE)
- The SARSA-PE target is $[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})]$
- Updating Action-Value Functions with Sarsa-PE gives

$$Q(s,a) \leftarrow Q(s,a) + \eta([r' + \gamma Q(s',a')] - Q(s,a))$$

- The observed data from the simulation can be presented in random order (SGD); a data point is the quintuple $s_t = s, a_t = a, s_{t+1} = s', a_{t+1} = a', r_{t+1} = r'$
- SARSA-PE is on-policy and can be used offline

Model Free Control

- This is covered in S, lecture 5

(VI) Value Iteration on the Value Function

- Recall that value iteration uses

$$v(s) \leftarrow \max_a \left(\mathcal{R}(s,a) + \gamma \sum_{s'} P(s'|s,a) v(s') \right)$$

  MC is not a good estimator of a maximum, so to apply this formula, we cannot do MC but we would need $P(s'|s,a)$ explicitly, which is not available in model-free approaches

- How can you maximize over actions, when only a specific action was observed?

- Only with certain model structures, TD is applicable to VI (see Appendix, Afterstate)

(Q-PI) ϵ-greedy exploration defines policy π^ϵ

- Recall that in policy iteration, we select a policy that is optimal (greedy) under the current Q-function
- Here, we need exploration: with a given deterministic policy, $a = \pi(s)$, the action space is not sufficiently explored
- Given a Q-function $Q(s, a)$ (obtained in the next step)
- ϵ-greedy exploration (π^ϵ) is defined as:
  - With probably $1-\epsilon$, chose $a^* = \arg\max_a Q(s, a)$ (a policy which is greedy on the Q- function)
  - With probability ϵ choose a random action
- One can show that $v^{\pi^\epsilon}(s) \geq v^\pi(s)$, where $Q(s, a)$ is the Q-function corresponding to $v^\pi(s)$
- ϵ-greedy exploration defines a policy π^ϵ!

(Q-PI) SARSA Algorithm

- ϵ-greedy exploration leads to a policy π^ϵ
- We can then use SARSA-PE (or MC) to estimate the Q-function of π^ϵ
- Iterating the two steps (SARSA-PE to obtain the Q-function of the new policy, ϵ-greedy exploration to define a policy update) is called the **SARSA algorithm**
- Since we have to simulate a new policy π^ϵ after a SARSA-PE update, the SARSA algorithm is on-policy!

(Q-VI) Q-Learning

- Recall Q-iteration,

$$q(s,a) \leftarrow \mathcal{R}(s,a) + \sum_{s'} P(s'|s,a) \max_{a'} q(s',a')$$

- The right side can be written as

$$\sum_{s'} \sum_{r'} P(r'|s,s',a) P(s'|s,a) [r' + \max_{a'} q_k(s',a')]$$

- Thus, $[r' + \gamma \max_{a'} Q(s',a')]$ is a one-sample approximation and serves as Q-target

(Q-VI) Q-Learning (cont'd)

- Q-learning (Chris Watkins, 1989) uses

$$Q(s,a) \leftarrow Q(s,a) + \eta \left([r' + \gamma \max_{a'} Q(s',a')] - Q(s,a) \right)$$

- The observed data from the simulation can be presented in random order (SGD); a data point is the quadruple $s_t = s, a_t = a, s_{t+1} = s', r_{t+1} = r'$
- The simulation can be off-policy and offline (with one fixed huge simulation); the reason is that we sample from $P(r'|s,s',a)P(s'|s,a)$ which is independent of the applied policy
- This can be relevant in safety critical applications where one is only able to observe the system under control: in theory, we can do Q-learning with one historic data set (in theory)!

(Q-VI) Q-Learning (cont'd)

- "SARSA and Q-Learning are both reinforcement learning algorithms that work in a similar way. The most striking difference is that SARSA is on-policy while Q Learning is off-policy."
- But: The maximization step $max_{a'}Q(s', a')$ might concern actions that were never observed, which can be dangerous!
- Thus, in actual Q-learning, one does new simulations with an updated behavior policy, which explores relevant space (actual Q-learning)

Q-Learning with Improvements on the Behavioral Policy

- The **behavior policy** (during learning) is ϵ-greedy w.r.t. the current $Q(s, a)$
- In turn: (i) update behavior policy using the current Q-function and perform a simulation using this updated policy and (ii) perform Q-learning
- (After convergence) the **target policy** is greedy

$$\pi(s) = \arg\max_{a'} Q(s, a')$$

Reinforcement Learning Part III

Volker Tresp
2021

From Tables to Function Approximation

- $V(s)$, $Q(s,a)$ and $\pi(s)$ can be high-dimensional
- So far, we assumed that these are represented as table lookup (vector, array)
- Convergence results exist for table-lookup representations; table-lookup becomes infeasible when the state space becomes large (the board game Go has 10^{170} states)
- s can be represented by a number of input variables, e.g., the positions of specific robot parts, velocities, angles, ...; in the notation we keep using s, but consider

$$V(s) \equiv V_{\mathbf{w}}(s) \equiv V_{\mathbf{w}}(x_1, \ldots, x_M)$$

- $(x_1, \ldots, x_M)$ are the features
- A bit advanced: if $x_1, \ldots, x_M$ are M one-hot vectors, then s can also be a one-hot vector defined as $x_1 \otimes x_2, \ldots, \otimes x_M$, where $\otimes$ is the tensor product

- Now we consider that $V_{\mathbf{w}}(s)$ is represented as basis functions (or feature functions), or with a neural networks
- This is covered in S, lecture 6

Basis / Feature Functions (Linear)

- As before, we can map the M inputs to M_ϕ basis/feature functions, and then one only need to learn the map from basis functions to value
- Basis functions here are real world features, that describe the state (e.g., distance of robot from a number of landmarks)
- To maintain the Markov property it should be possible (in principle) to restore the state from its features
- The parameters enter linearly in the model

Neural Networks (Nonlinear)

- Neural networks replace the basis functions (feature) mappings

TD(0) with a Neural Network

- As an example, in TD(0), assume that $V_{\mathbf{w}}(s)$ is an approximation by a neural network
- Then we have the update for parameter w_k,

$$w_k \leftarrow w_k + \eta([r' + \gamma V_{\mathbf{w}^-}(s')] - V_{\mathbf{w}}(s))\frac{\partial V_{\mathbf{w}}(s)}{\partial w_k}$$

- Here, $[r' + \gamma V_{\mathbf{w}^-}(s')]$ is the TD target with the current parameter estimates (I use the notation $V_{\mathbf{w}^-}(s')$); the gradient is only calculated w.r.t $V_{\mathbf{w}}(s)$
- This generalizes, e.g., to SARSA-PE and Q-learning

Getting it to work: Experience Replay / Batch Reinforcement Learning

- Q-learning as described has convergence problems, in particular in connection with nonlinear approximators, neural networks, and even more specifically with deep neural networks (deep Q-networks, DQNs)
- An **experience** is a (random) subset of the simulation (Schaul, John Quan, Antonoglou, Silver, 2016)
- First, compute Q-learning targets using the current, fixed parameters $\mathbf{w}^-$; the fixed Q-targets are $[r' + \gamma \max_{a'} Q_{\mathbf{w}^-}(s', a')]$
- Repeatedly sample random mini-batches; calculate the updated weights $\mathbf{w} \leftarrow$ using least squares where the contribution to the cost function of a data point sample is

$$([r' + \gamma \max_{a'} Q_{\mathbf{w}^-}(s', a')] - Q_{\mathbf{w}}(s, a))^2$$

- This is called **experience replay with fixed Q-targets**

Experience Replay / Batch Reinforcement Learning (cont'd)

- Since the cost function minimizes the squared error, this is called **Least Squares Q-learning**
- Sampling random subsets is important to reduce correlation!
- Occasionally, $Q_{\mathbf{w}^-}(s,a) \leftarrow Q_{\mathbf{w}}(s,a)$ (there are different mechanisms)
- Note: with fixed basis functions/features, the update can be solved by simple linear regression!
- The **behavior policy** (during learning) is ϵ-greedy w.r.t. the current $Q(s,a)$

Dyna: Learning a Model

- Recall that in part I, we assumed that a model of the system was available
- It might be possible to learn an (inaccurate) model first and then do the optimization using the equations in part I. (One problem is that a real system is typically being run under a control law and its might be challenging to identify the model)
  - Learn model $P(s'|s,a)$, which predicts a probability distribution
  - Learn the reward function $\mathcal{R}(s,a)$ as a regression problem
- General problem: is a good policy for the learned model (simulated experience) also good for the true system (environment)?
- $P(s'|s,a)$: if a table representation is used, this is a 3-way array
- Modelling $P(s'|s,a)$ when s, s' are represented as features is non-trivial

- (i) One can use graphical models or generalized linear models, ...
- (ii) One makes strong model assumptions, like a joint Gaussian distribution
- (iii) In cases one only needs to generate samples (next) from the distribution: One can use GAN models

Dyna (cont'd)

- Approach 1: Planning / Dynamic programming using the model, using favourite planning algorithm: Value iteration, Policy iteration, ...
- Approach 2: Generate samples from the model; then apply model-free RL to samples, e.g.: Monte-Carlo control, Sarsa, Q-learning
- Data from the true system/envorinment is called data from the **real experience**
- Data from the learned model is called data from the **simulated experience**
- Approach 3, Dyna-Q (Sutton 1990): Q-learning with data both from the real experience and the simulated experience; popular in real-world control applications
- This is covered in S, lecture 8

Imitation Learning and Inverse RL

- Copying an existing controller is called **imitation learning** or **apprentice learning**
- Instead of copying a (human) controller (i.e., teacher) directly, one can try to learn the reward function $\mathcal{R}(s)$ if the teacher by observing the teacher's actions; then one derives the control based on that that reward function; some approaches also use a model of the system (Pieter Abbeel, Andrew Y. Ng, 2004)
- This approach is used in the spectacular control of a toy helicopter and is called **inverse reinforcement learning**

Direct Policy Search and Policy Gradient

Direct Policy Search

- Pure Q-learning works well in complete discrete domains (e.g., board games), but not as well continuous (robotic) domains (e.g., control of robot arm movement)
- **This is the approach to use if you need a robust RL algorithm which you want to apply to a (novel) problem and where you do not want to deal with all the complexity of value function or Q-function estimation!**
- The idea is to directly optimize the parameterized policy/controller: $P_\theta(a|s)$; this is called **direct policy search**
- Example: $P_\theta(a|s) = \text{softmax}_a(\mathbf{f}_\theta(x_1, ..., x_M))$
- Direct policy search is covered in S, lecture 7
- Advantages: robust convergence properties; effective in high-dimensional or continuous action spaces; can learn stochastic policies

- Disadvantages: Typically converge to a local rather than global optimum; evaluating a policy is typically inefficient with high variance

Estimating the Gradient versus Gradient-free Methods

- The score is given by MC
- Some approaches are gradient free: Hill climbing; Simplex; Genetic algorithms
- One can approximate the gradient by finite differences
- We will now cover policy gradient, which is the basis approach for calculating an approximate gradient

Policy Gradient

- Consider that at $t = 0$, we are is state s_0.
- Consider the expected reward at some feature time τ,

$$\mathbb{E}(r_\tau) = \sum_r r P(r_\tau = r|s_0)$$

- Consider the gradient w.r.t a parameter in the policy,

$$\frac{\partial}{\partial\theta}\mathbb{E}(r_\tau) = \sum_r r\frac{\partial}{\partial\theta}P(r_\tau = r|s_0) = \sum_r rP(r_\tau = r|s_0)\frac{\partial}{\partial\theta}\log P(r_\tau = r|s_0)$$

Here we have used the log-trick discussed in the beginning of part I. Note that is the sum all terms which do not concern the policy disappear, since they do not depend on θ.

Policy Gradient (cont'd)

- Note that is the sum all terms which do not concern the policy disappear, since they do not depend on θ, and

$$\frac{\partial}{\partial\theta}\mathbb{E}(r_\tau) = \sum_r rP(r_\tau = r|s_0)\frac{\partial}{\partial\theta}\sum_{t=0}^{\tau-1}\log P(a_t|s_t)$$

$$\approx r_\tau \sum_{t=0}^{\tau-1}\frac{\partial}{\partial\theta}\log P(a_t|s_t)$$

- The last step is a Monte Carlo approximation; r_τ is the actually observed reward at τ and a_t, s_t are the actual actions and states in the simulation

Policy Gradient (cont'd)

- Now we consider the sum of the future rewards,

$$\frac{\partial}{\partial\theta}\mathbb{E}(r_1 + r_2 + ...) \approx \sum_{\tau=1} r_\tau \sum_{t=0}^{\tau-1} \frac{\partial}{\partial\theta} \log P(a_t|s_t)$$

$$= \sum_{t=0} \left(\sum_{\tau=t+1} r_\tau \right) \frac{\partial}{\partial\theta} \log P(a_t|s_t)$$

  The sample update of the algorithm REINFORCE (Williams 1992) is now,

$$\theta \leftarrow \theta + \eta \left(\sum_{\tau=t+1} r_\tau \right) \frac{\partial}{\partial\theta} \log P(a_t|s_t)$$

- The actual algorithm uses a discount factor γ. Note the intuitive elegance: the algorithm "pretends" that the action in the simula-

tion is the true target, but the gradient is weighted by the future success of the action

Actor-Critic

- (Barto, Sutton, and Anderson, 1983)
- In combination with Q-learning (Q-actor critic)

$$\theta \leftarrow \theta + \eta Q_{\mathbf{w}}(s_t, a_t) \frac{\partial \log P^{\pi_\theta}(a_t|s_t)}{\partial \theta}$$

- In learning, both $\mathbf{w}$ and θ are optimized
- The Q-function can be optimized with experience replay and least squares

Board Games

Monte-Carlo Search (MCS)

- Assume a (learned) model and a policy π
- The current state is s_t
- Simulate K episodes (until the end of the episode at time T)
- Estimate $Q(s, a)$ based on the mean returns
- Select the one action which corresponds to the action which maximizes $Q(s, a)$
- In many games simple Monte-Carlo search is enough (Scrabble, Backgammon)

Monte-Carlo Trees Search (MCTS)

- Coulom (2006) and Kocsis and Szepesvari (2006)
- Combination of tree policy and defaults policy
- For details: S, lecture 8
- Works for black-box models (only requires samples)
- Computationally efficient and can be parallelized
- MCTS is best performing method in many challenging games (e.g., Go)

Minimax and Nash Equilibrium

- (This is covered in S, lecture 10)
- So far, we assumed that nature was random but neutral
- In a game, one is playing against an opponent (minimax search)
- If the policy of your opponent is fixed, the problem reduces to a normal MDP; the other player becomes part of the environment
- In a Nash equilibrium, none of the (rational players) would choose to deviate from her/his policy; this is the concept behind **self-play RL** (a RL agent playing against itself)

TD-Gammon

- Predecessor: checkers-playing program (Arthur Samuel, 1959)
- TD-Gammon is a computer backgammon program developed in 1992 by Gerald Tesauro
- TD-gammon is based on TD(0)
- A neural network was used for function approximation
- Afterstate evaluation (see Appendix)

AlphaGo

- AlphaGo is a computer program that plays the board game Go
- David Silver et al., 2016
- Apprentice learning: First, the policy network π_σ is trained by copying humans (directly from expert human moves, imitation learning): bootstrapping from human gameplay expertise by supervised learning; this provides fast, efficient learning updates with immediate feedback and high quality gradient
- Policy gradient: A fast policy network π_π (reinforcement learning (RL) policy network) is trained that can rapidly sample actions during rollouts
- Experience replay with LS: Then learn a value network $v(s)$ using LS training
- MCTS: Alpha-Go uses (MCTS) Monte Carlo tree search, guided by a "value network" and a "policy network"

AlphaGo Zero

- By playing games against itself, AlphaGo Zero reached the level of AlphaGo Master in 21 days, and exceeded all the old versions in 40 days
- David Silver et al., 2017
- Does not use bootstrapping from human gameplay expertise
- The neural network initially knows nothing about Go beyond the rules: Pure self-play

AlphaZero

- AlphaZero: achieved within 24 hours a superhuman level of self-play in the games of chess, shogi, and Go
- David Silver et al., 2017

Atari

- An agent uses end-to-end reinforcement learning with convolutional neural networks (DQNs) for playing ATARI games (game screen is treated as a pixel image)
- Mnih1, Kavukcuoglu1, Silver, et al., 2015
- The basis is a DQN with experience replay and least squares Q-learning

Recent Developments

- MuZero is a computer program developed by artificial intelligence research company DeepMind to master games without knowing their rules; MuZero was trained via self-play, with no access to rules, opening books, or endgame tables. The trained algorithm used the same convolutional and residual algorithms as AlphaZero (Schrittwieser et al., 2020)
- To get serious: *Stable Baselines* is a set of improved implementations of Reinforcement Learning (RL) algorithms based on OpenAI Baselines.
- Recent advances in policy gradient, Actor-Critic: Soft Actor Critic (SAC), Proximal Policy Optimization (PPO); Deep Deterministic Policy Gradient (DDPG)

Conclusions

- Reinforcement learning permits one to act better, maybe even optimally (rationally)
- If the Markov process cannot be observed completely, we are dealing with a partially observable Markov decision process (POMDPs) (not covered in this lecture, but in S and BS)
- An ongoing topic is offline RL; examples are BCQ, BEAR, and BRAC

Appendix

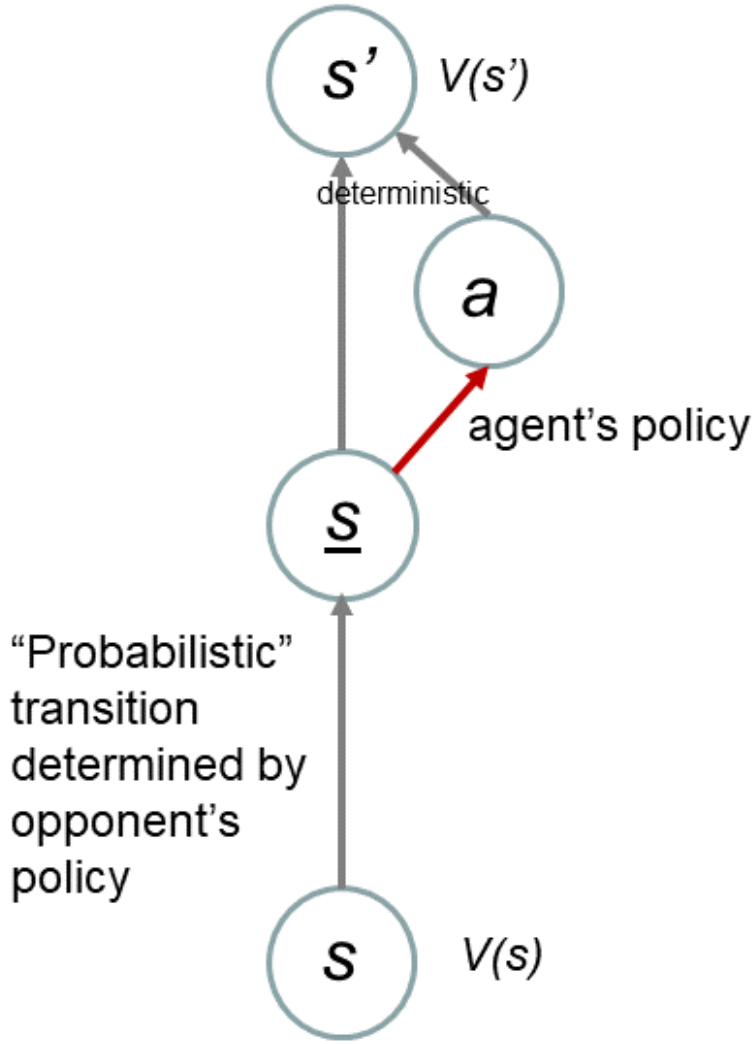
Special Model Structure: Afterstate

- Assume an intermediate state $\underline{s}$ (see figure) with

$$P(s'|s, \underline{s}, a) = P(s'|\underline{s}, a)$$

  The term on the right side is deterministic with: $s' = s'(\underline{s}, a)$; $s'(\underline{s}, a)$ is called the afterstate

- Then $q(s, a) = v(s'(\underline{s}, a))$ and a greedy policy update is $\pi(s) = \arg\max_{a'} v(s'(\underline{s}, a'))$


s'
V(s')
deterministic
a
agent's policy
s
"Probabilistic" transition determined by opponent's policy
s
V(s)

Off-Policy TD using Importance Sampling

- TD and some of the other algorithms can also be used off-policy using importance sampling
- For TD,

$$V(S_t) \leftarrow V(S_t) + \eta \left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} [R_{t+s} + \gamma V(S_{t+1})] - V(S_t) \right)$$

- $\pi(A_t|S_t)$ is the policy we want to evaluate, i.e. the target policy
- $\mu(A_t|S_t)$ is the policy in the simulation, i.e. the behavior policy
- This equation can be derived from the concept of importance sampling
- *The simulation can be off-policy and offline (with one fixed huge simulation)*