Reinforcement Learning

Business Analytics Practice
Winter Term 2015/16

Nicolas Pröllochs and Stefan Feuerriegel

Today's Lecture

Objectives

1. Grasp an understanding of Markov decision processes
2. Understand the concept of reinforcement learning
3. Apply reinforcement learning in R
4. Distinguish pros/cons of different reinforcement learning algorithms

Outline

Outline

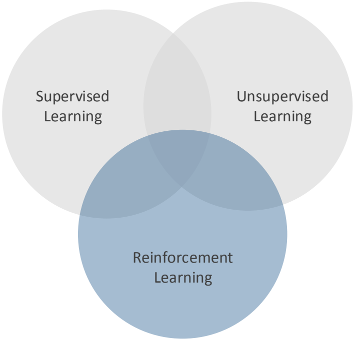
Branches of Machine Learning

Supervised Learning

- Learns from pairs of input and desired outcome (i. e. labels)

Unsupervised Learning

- Tries to find hidden structure in unlabeled data

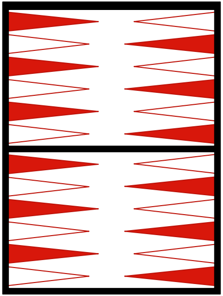


Reinforcement Learning

- Learning from interacting with the environment
- No need for pairs of input and correct outcome
- Feedback restricted to a reward signal
- Mimics human-like learning in actual environments

Example: Backgammon

Reinforcement learning can reach a level similar to the top three human players in backgammon



Learning task

- Select best move at arbitrary board states
  → i. e. with highest probability to win

Training signal

- Win or loss of overall game

Training

- 300,000 games played against the system itself

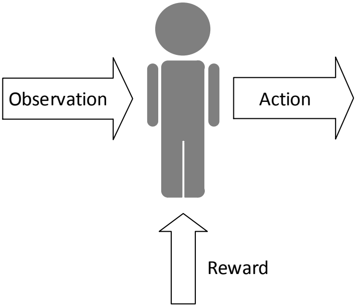
Algorithm

- Reinforcement learning (plus neural network)

→ Tesauro (1995): Temporal Difference Learning and TD-Gammon. In: Comm. of the ACM, 38:3, pp. 58–68

Reinforcement Learning

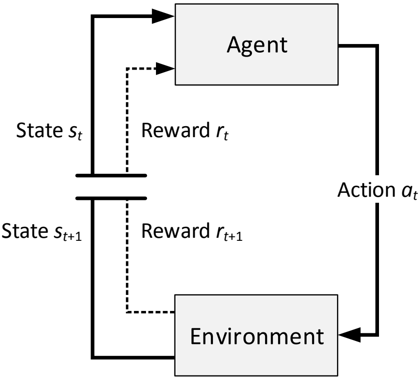
- An agent interacts with its environment
- Agent takes actions that affect the state of the environment
- Feedback is limited to a reward signal that indicates how well the agent is performing
- Goal: improve the behavior given only this limited feedback



Examples

- Defeat the world champions at backgammon or Go
- Manage an investment portfolio
- Make a humanoid robot walk

Agent and Environment



At each step t, the agent:

- Executes action a_t
- Receives observation s_t
- Receives scalar reward r_t

The environment:

- Changes upon action a_t
- Emits observation s_{t+1}
- Emits scalar reward r_{t+1}

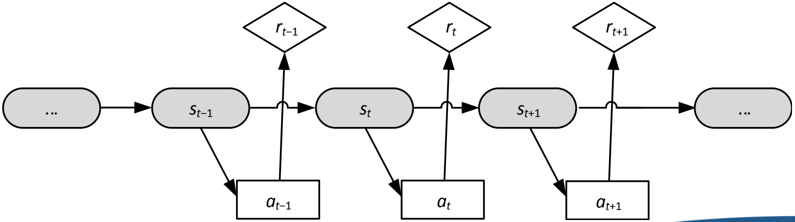
- Time step t is incremented after each iteration

Agent and Environment

Example

①	ENVIRONMENT	▸ You are in state 3 with 4 possible actions
②	AGENT	▸ I'll take action 2
③	ENVIRONMENT	▸ You received a reward of 5 units ▸ You are in state 1 with 2 possible actions
⋮	⋮	⋮

Formalization



Reinforcement Learning Problem

Finding an optimal behavior

- Learn optimal behavior π based on past actions
- Maximize the expected cumulative reward over time

Challenges

- Feedback is delayed, not instantaneous
- Agent must reason about the long-term consequences of its actions

Illustration

- In order to maximize one's future income, one has to study now
- However, the immediate monetary reward from this might be negative

$\Rightarrow$ How do we learn optimal behavior?

Trial-and-Error Learning

The agent should discover optimal behavior via trial-and-error learning

1. **Exploration**
   - Try new or non-optimal actions to learn their reward
   - Gain a better understanding of the environment
2. **Exploitation**
   - Use current knowledge
   - This might not be optimal yet, but should deviate only slightly

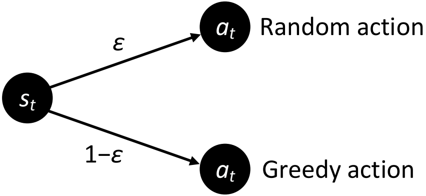
Examples

1. Restaurant selection
   - **Exploitation:** go to your favorite restaurant
   - **Exploration:** try a new restaurant
2. Game playing
   - **Exploitation:** play the move you believe is best
   - **Exploration:** play an experimental move

ε-Greedy Action Selection

Idea

- Provide a simple heuristic to choose between exploitation and exploration
- Implemented via a random number $0 \leq \varepsilon \leq 1$
  - With probability ε, try a random action
  - With probability $1 - \varepsilon$, choose the current best



- Typical choice is e. g. $\varepsilon = 0.1$
- Other variants decrease this value over time
  $\rightarrow$ i. e. agent gains confidence and thus needs less exploration

Outline

Markov Decision Process

- A Markov decision process (MDP) specifies a setup for reinforcement learning
- MDPs allow to model decision making in situations where outcomes are partly random and partly under the control of a decision maker

Definition

1. A Markov Decision Process is a 4-tuple (S, A, R, T) with
   - A set of possible world states S
   - A set of possible actions A
   - A real-valued reward function R
   - Transition probabilities T
2. A MDP must fulfill the so-called Markov property
   - The effects of an action taken in a state depend only on that state and not on the prior history

Markov Decision Process

State

- A state s_t is a representation of the environment at time step t
- Can be directly observable to the agent or hidden

Actions

- At each state, the agent is able to perform an action a_t that affects the subsequent state of the environment s_{t+1}
- Actions can be any decisions which one wants to learn

Transition probabilities

- Given a current state s, a possible subsequent state s' and an action a
- The transition probability $T^a_{ss'}$ from s to s' is defined by

$$T^a_{ss'} = P\left[s_{t+1} = s' \mid s_t = s, a_t = a\right]$$

Rewards

- A reward r_{t+1} is a scalar feedback signal emitted by the environment
- Indicates how well agent is performing when reaching step $t+1$
- The expected reward $R^a_{ss'}$ when moving from state s to s' via action a is given by

$$R^a_{ss'} = E\left[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right]$$

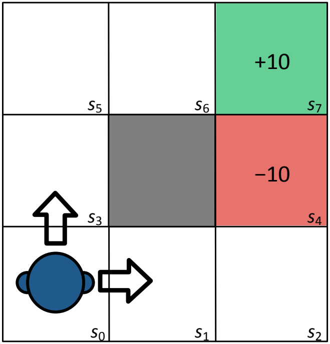
Examples

1. Playing backgammon or Go
   - Zero reward after each move
   - A positive/negative reward for winning/losing a game
2. Managing an investment portfolio
   - A positive reward for each dollar left in the bank

Goal: maximize the expected cumulative reward over time

Markov Decision Process

Example: Moving a pawn to a destination on a grid



- States $S = \{s_0, s_1, \ldots, s_7\}$
- Actions $A = \{\text{up}, \text{down}, \text{left}, \text{right}\}$
- Transition probabilities
  - $T^{\text{up}}_{s_0,s_3} = 0.9$
  - $T^{\text{right}}_{s_0,s_1} = 0.1$
  - $\ldots$
- Rewards
  - $R^{\text{right}}_{s_6,s_7} = +10$
  - $R^{\text{up}}_{s_2,s_4} = -10$
  - Otherwise $R = 0$
- Start in s_0
- Game over when reaching s_7

$\rightarrow$ available actions $A(s)$ depend on current state s

Policy

Learning task of an agent

- Execute actions in the environment and observe results, i. e. rewards
- Learn a policy $\pi : S \to A$ that works as a selection function of choosing an action given a state
- A policy fully defines the behavior of an agent, i. e. its actions
- MDP policies depend only on the current state and not its history
- Policies are stationary (i. e. time-independent)

Objective

- Maximize the expected cumulative reward over time
- The expected cumulative reward from an initial state s with policy π is

$$J_\pi(s) = \sum_t R^{a_t}_{s_t, s_{t+1}} = E_\pi \left[\sum_t r_t \mid s_0 = s \right]$$

Value Functions

Definition

- The state-value function $V_\pi(s)$ of an MDP is the expected reward starting from state s, and then following once policy π
- $V_\pi(s) = E_\pi\left[J_\pi(s_t) \mid s_t = s\right]$
- Quantifies how good is it to be in a particular state s

Definition

- The state-action value function $Q_\pi(s,a)$ is the expected reward starting from state s, taking action a, and then following policy π
- $Q_\pi(s,a) = E_\pi\left[J_\pi(s_t) \mid s_t = s, a_t = a\right]$
- Quantifies how good is it to be in a particular state s and apply action a, and afterwards follow policy π

Now, we can formalize the policy definition (with discount factor γ) via

$$\pi(s) = \arg\max_a \sum_{s'} T^a_{ss'}\left(R^a_{ss'} + \gamma V_\pi(s')\right)$$

Optimal Value Functions

- While π can be any policy, π^* denotes the optimal one with the highest expected cumulative reward
- The optimal value functions specify the best possible policy
- A MDP is solved when the optimal value functions are known

Definitions

1. The optimal state-value function $V_{\pi^*}(s)$ maximizes the expected reward over all policies

$$V_{\pi^*}(s) = \max_{\pi} V_{\pi}(s)$$

2. The optimal action-value function $Q_{\pi^*}(s,a)$ maximizes the action-value function over all policies

$$Q_{\pi^*}(s,a) = \max_{\pi} Q_{\pi}(s,a)$$

Markov Decision Processes in R

- Load R library `MDPtoolbox`

```
library(MDPtoolbox)
```

- Create transition matrix for two states and two actions

```
T <- array(0, c(2, 2, 2))
T[,,1] <- matrix(c(0, 1, 0.8, 0.2), nrow=2, ncol=2, byrow=TRUE)
T[,,2] <- matrix(c(0.5, 0.5, 0.1, 0.9), nrow=2, ncol=2, byrow=TRUE)
```

$\rightarrow$ Dimensions are #states $\times$ #states $\times$ #actions

- Create reward matrix (of dimensions #states $\times$ #actions)

```
R <- matrix(c(10, 10, 1, -5), nrow=2, ncol=2, byrow=TRUE)
```

- Check whether the given `T` and `R` represent a well-defined MDP

```
mdp_check(T, R)

## [1] ""
```

$\rightarrow$ Returns an empty string if the MDP is valid

Outline

Types of Learning Algorithms

Aim: find optimal policy and value functions

Model-based learning

- Aim: find optimal policy and value functions
- Model of the environment is as MDP with transition probabilities
- Approach: learn the MDP model or an approximation of it

Model-free learning

- Explicit model of the environment model is not available
  $\rightarrow$ i. e. transition probabilities are unknown
- Approach: derive the optimal policy without explicitly formalizing the model

Outline

3 Learning Algorithms
- Model-Based Learning
- Model-Free Learning

Model-Based Learning: Policy Iteration

Approach via policy iteration

- Given an initial policy π_0
- Evaluate policy π_i to find the corresponding value function V_{π_i}
- Improve policy over V_π via greedy exploration
- Policy iteration always converges to optimal policy π^*

Illustration

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \cdots \xrightarrow{E} V_{\pi^*} \xrightarrow{I} \pi^*$$

with

- E: policy evaluation
- I: policy improvement

Policy Evaluation

- Computes the state-value function V_π for an arbitrary policy π via

$$
\begin{aligned}
V_{\pi}(s) &= E_\pi \left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t-3} + \cdots \mid s_t = s \right] \\
&= E_\pi \left[r_{t+1} + \gamma V_{\pi}(s+1) \mid s_t = s \right] \\
&= \sum_a \pi(s,a) \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma V_{\pi}(s') \right]
\end{aligned}
$$

- System of $|S|$ linear equations with $|S|$ unknowns
- Solvable but computational expensive if $|S|$ is large
- Advanced methods are available, e. g. iterative policy evaluation

Discount factor

- If $0 < \gamma < 1$, makes cumulative reward finite
- Necessary for setups with infinite time horizons
- Puts more importance on first learning steps, but less on later ones

Iterative Policy Evaluation

- Iterative policy evaluation uses dynamic programming
- Iteratively approximate V_π
- Choose V_0 arbitrarily
- Then use Bellman equation as an update rule

$$
\begin{aligned}
V_{k+1}(s) &= E_\pi \left[r_{t+1} + \gamma V_k(s+1) \mid s_t = s \right] \\
&= \sum_a \pi(s,a) \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right]
\end{aligned}
$$

- Sequence $V_k, V_{k+1}, \ldots$ converges to V_π as $k \to \infty$

Policy Improvement

- Policy evaluation determines the value function V_π for a policy π
- The alternative step exploits this knowledge to select the optimal action in each state
- For that, policy improvement searches policy π' that is as good as or better than π
- Remedy is to use state-action value function via

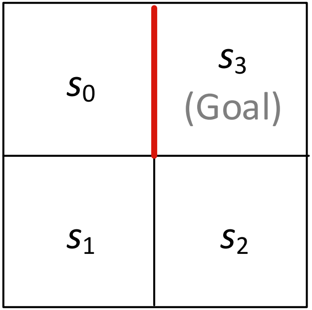
$$\begin{aligned}
\pi'(s) &= \arg\max_a Q_\pi(s,a) \\
&= \arg\max_a E\left[r_{t+1} + \gamma V_k(s+1) \mid s_t = s\right] \\
&= \arg\max_a \sum_{s'} T^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s')\right]
\end{aligned}$$

- Afterwards, continue with policy evaluation and policy improvement until a desired convergence criterion is reached

Policy Iteration

Example

- Learning an agent traveling through a 2×2 grid (i. e. 4 states)



- Wall (red line) prevents direct moves from s_0 to s_3
- Reward favors shorter routes
  - Visiting each square/state gives a reward of -1
  - Reaching the goal gives a reward of 10
- Actions: move left, right, up or down
- Transition probabilities are < 1 $\rightarrow$ i. e. allows erroneous moves

Policy Iteration in R

Example

- Design an MDP that finds the optimal policy to that problem
- Create individual matrices with pre-specified (random) transition probabilities for each action

```
up <- matrix(c(  1,   0,   0,   0,
               0.7, 0.2, 0.1,   0,
                 0, 0.1, 0.2, 0.7,
                 0,   0,   0,   1),
             nrow=4, ncol=4, byrow=TRUE)

left <- matrix(c(0.9, 0.1,   0,   0,
                 0.1, 0.9,   0,   0,
                   0, 0.7, 0.2, 0.1,
                   0,   0, 0.1, 0.9),
               nrow=4, ncol=4, byrow=TRUE)
```

Policy Iteration in R

- Second chunk of matrices

```
down <- matrix(c(0.3, 0.7,   0,   0,
                   0, 0.9, 0.1,   0,
                   0, 0.1, 0.9,   0,
                   0,   0, 0.7, 0.3),
               nrow=4, ncol=4, byrow=TRUE)

right <- matrix(c(0.9, 0.1,   0,   0,
                  0.1, 0.2, 0.7,   0,
                    0,   0, 0.9, 0.1,
                    0,   0, 0.1, 0.9),
                nrow=4, ncol=4, byrow=TRUE)
```

- Aggregate previous matrices to create transition probabilities in `T`

```
T <- list(up=up, left=left,
          down=down, right=right)
```

Policy Iteration in R

- Create matrix with rewards

```r
R <- matrix(c(-1, -1, -1, -1,
              -1, -1, -1, -1,
              -1, -1, -1, -1,
              10, 10, 10, 10),
              nrow=4, ncol=4, byrow=TRUE)
```

- Check if this provides a well-defined MDP

```r
mdp_check(T, R) # empty string => ok

## [1] ""
```

Policy Iteration in R

- Run policy iteration with discount factor $\gamma = 0.9$

```
m <- mdp_policy_iteration(P=T, R=R, discount=0.9)
```

- Display optimal policy π^*

```
m$policy

## [1] 3 4 1 1

names(T)[m$policy]

## [1] "down"  "right" "up"    "up"
```

- Display value function V_{π^*}

```
m$V

## [1]  58.25663  69.09102  83.19292 100.00000
```

Outline

3 Learning Algorithms
- Model-Based Learning
- Model-Free Learning

Model-Free Learning

Drawbacks of model-based learning

- Requires MDP, i. e. explicit model of the dynamics in the environment
- Transition probabilities are often not available or difficult to define
- Model-based learning is thus often intractable even in "simple" cases

Model-free learning

- Idea: learn directly from interactions with the environment
- Only use experience from the sequences of states, action, and rewards

Common approaches

1. **Monte Carlo methods** are simple but has slow convergence
2. **Q-learning** is more efficient due to off-policy learning

Monte Carlo Method

- Monte Carlo methods require no knowledge of transition as in MDPs
- Perform reinforcement learning from a sequence of interactions
- Mimic policy iteration to find optimal policy
- Estimate the value of each action $Q(s,a)$ instead of $V(s)$
- Store average rewards in state-action table

Example

- State-action table

State	Actions		Optimal Policy
	a_1	a_2	
s_1	2	1	a_1
s_2	1	3	a_2
s_3	2	4	a_2

Monte Carlo Method

Algorithm

1. Start with an arbitrary state-action table (and corresponding policies)
   $\rightarrow$ Often all rewards are initially set to zero
2. Observe first state
3. Choose an action according to ε-greedy action selection, i. e.
   - With probability ε, pick a random action
   - Otherwise, take action with highest expected reward
4. Update state-action table with new reward (averaging)
5. Observe new state
6. Go to step 3

Disadvantage

- High computational time and thus slow convergence
  $\rightarrow$ Method must frequently evaluate a suboptimal policy

Q-Learning

- One of the most important breakthroughs in reinforcement learning
- Off-policy learning concept
  - Explore the environment and at the same time exploit the current knowledge
- In each step, take a look forward to the next state and observe the maximum possible reward for all available actions in that state
- Use this knowledge to update the action-value of the corresponding action in the current state
- Apply update rule with learning rate α $(0 < \alpha \leq 1)$

$$Q(s,a) \leftarrow \underbrace{Q(s,a)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \left[\underbrace{r'}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_{a'} Q(s',a')}_{\text{expected optimal value}} - \underbrace{Q(s,a)}_{\text{old value}} \right]$$

- Q-learning is repeated for different episodes (e. g. games, trials, etc.)

Q-Learning

Algorithm

1. Initialize the table $Q(s,a)$ to zero for all state-action pairs (s,a)
2. Observe the current state s
3. Repeat until convergence
   - Select an action a and apply it
   - Receive immediate reward r
   - Observe the new state s'
   - Update the table entry for $Q(s,a)$ according to

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

   - Move to next state, i. e. $s \leftarrow s'$

Outline

Q-Learning in R

- Unfortunately, R has no dedicated library for model-free reinforcement learning yet
- Alternative implementations are often available in other programming languages
- Possible remedy: write your own implementation
  $\rightarrow$ Not too difficult with the building blocks on the next slides

Example

- Learning an agent finding a destination in a 2×2 grid with a wall
- Initialize 4 states and 4 actions

```
actions <- c("up", "left", "down", "right")
states <- c("s0", "s1", "s2", "s3")
```

- Note: real applications (such as in robotics) are prone to disturbances

Q-Learning in R

Building blocks

1. Adding a function that mimics the environment

```
simulateEnvironment <- function(state, action) {
  ...
}
```

2. Add a Q-learning function that performs a given number `n` of episodes

```
Qlearning <- function(n, s_0, s_terminal,
                      epsilon, learning_rate) {
  ...
}
```

3. Call Q-learning with an initial state `s_0`, a final state `s_terminal` and desired parameters to search a policy

```
Qlearning(n, s_0, s_terminal, epsilon, learning_rate)
```

Q-Learning in R

- Function returns a list with two entries: the next state and the corresponding reward given the current state and an intended action

```
simulateEnvironment <- function(state, action) {
  # Calculate next state (according to sample grid with wall)
  # Default: remain in a state if action tries to leave grid
  next_state <- state
  if (state == "s0" && action == "down")  next_state <- "s1"
  if (state == "s1" && action == "up")    next_state <- "s0"
  if (state == "s1" && action == "right") next_state <- "s2"
  if (state == "s2" && action == "left")  next_state <- "s1"
  if (state == "s2" && action == "up")    next_state <- "s3"
  if (state == "s3" && action == "down")  next_state <- "s2"

  # Calculate reward
  if (next_state == "s3") {
    reward <- 10
  } else {
    reward <- -1
  }

  return(list(state=next_state, reward=reward))
}
```

Q-Learning in R

- Function applies Q-learning for a given number `n` of episodes

```
Qlearning <- function(n, s_0, s_terminal,
                      epsilon, learning_rate) {
  # Initialize state-action function Q to zero
  Q <- matrix(0, nrow=length(states), ncol=length(actions),
              dimnames=list(states, actions))

  # Perform n episodes/iterations of Q-learning
  for (i in 1:n) {
    Q <- learnEpisode(s_0, s_terminal,
                      epsilon, learning_rate, Q)
  }

  return(Q)
}
```

- Returns state-action function Q

Q-Learning in R

```
learnEpisode <- function(s_0, s_terminal, epsilon, learning_rate, Q) {
  state <- s_0 # set cursor to initial state

  while (state != s_terminal) {
    # epsilon-greedy action selection
    if (runif(1) <= epsilon) {
      action <- sample(actions, 1)      # pick random action
    } else {
      action <- which.max(Q[state, ])   # pick first best action
    }

    # get next state and reward from environment
    response <- simulateEnvironment(state, action)

    # update rule for Q-learning
    Q[state, action] <- Q[state, action] + learning_rate *
      (response$reward + max(Q[response$state, ]) - Q[state, action])

    state <- response$state  # move to next state
  }

  return(Q)
}
```

Q-Learning in R

- Choose learning parameters

```
epsilon <- 0.1
learning_rate <- 0.1
```

- Calculate state-action function Q after 1000 episodes

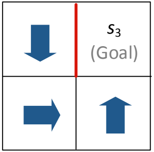
```
set.seed(0)
Q <- Qlearning(1000, "s0", "s3", epsilon, learning_rate)
Q

##             up      left      down     right
## s0 -79.962619 -81.15445 -68.39532 -79.34825
## s1 -73.891963 -52.43183 -52.67565 -47.91828
## s2  -8.784844 -46.32207 -17.97360 -20.29088
## s3   0.000000   0.00000   0.00000   0.00000
```

- Optimal policy

```
# note: problematic for states with ties
actions[max.col(Q)]

## [1] "down"  "right" "up"    "up"
```



- Agent chooses optimal action in all states

Outline

1 Reinforcement Learning

2 Markov Decision Process

3 Learning Algorithms

4 Q-Learning in R

5 Wrap-Up

Wrap-Up

Summary

- Reinforcement learning learns through trial-and-error from interactions
- The reward indicates the performance of the agent
  $\rightarrow$ But without showing how to improve its behavior
- Learning is grouped into model-based and model-free strategies
- A common and efficient model-free variant is Q-learning
- Similar to human-like learning in real-world environments
- Common for trade-offs between long-term vs. short-term benefits

Drawbacks

- Can be computational expensive when state-action space is large
- No R library is yet available for model-free learning

Wrap-Up

Commands inside MDPtoolbox

`mdp_example_rand()`	Generate a random MDP
`mdp_check(T, R)`	Check whether the given *T* and *R* represent a well-defined MDP
`mdp_value_iteration(...)`	Run value iteration to find best policy
`mdp_policy_iteration(...)`	Run policy iteration to find best policy

Further readings

- Sutton & Barto (1998). Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA. Also available online: `https://webdocs.cs.ualberta.ca/~sutton/book/the-book.html`
- Slides by Watkins: `http://webdav.tuebingen.mpg.de/mlss2013/2015/speakers.html`
- Slides by Littman: `http://mlg.eng.cam.ac.uk/mlss09/mlss_slides/Littman_1.pdf`
- Vignette for MDPtoolbox: `https://cran.r-project.org/web/packages/MDPtoolbox/MDPtoolbox.pdf`