哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Big data technology and Practice

李春山

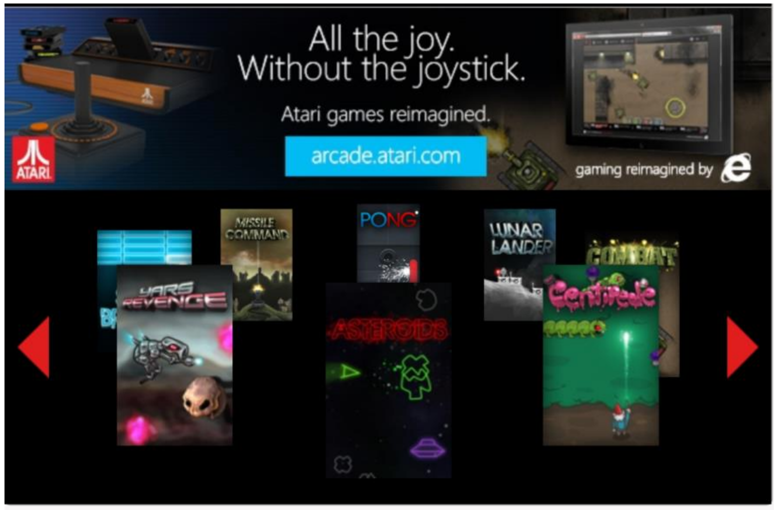
2020年5月14日

内容提要

Chapter 17: Reinforcement Learning

Chapter 17: Reinforcement Learning

- Reinforcement Learning (RL) is one of the most exciting fields of Machine Learning today.
- A revolution took place in 2013, when DeepMind demonstrated a system that could learn to play just about any Atari game from scratch.
- DeepMind was bought by Google for over 500 million dollars in 2014.
- A series of amazing feats, culminating in March 2016 with the victory of their system AlphaGo against Lee Sedol, the world champion of the game of Go.

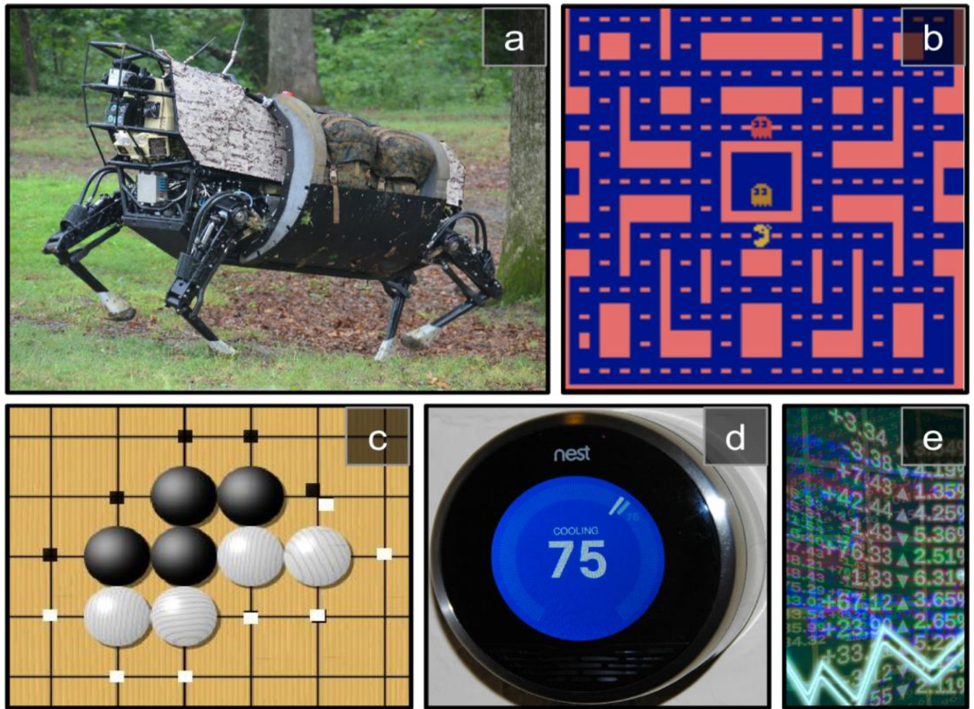


1. Basic principle of RL

- In Reinforcement Learning, a software agent
  - makes observations and
  - takes actions within an environment, and
  - in return, it receives rewards.
  - Its objective is to learn to act in a way that will maximize its
  - expected long-term rewards.
- Say, assuming “positive rewards as pleasure, and negative rewards as pain” or any other definition of “reward”.
- In short, the agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

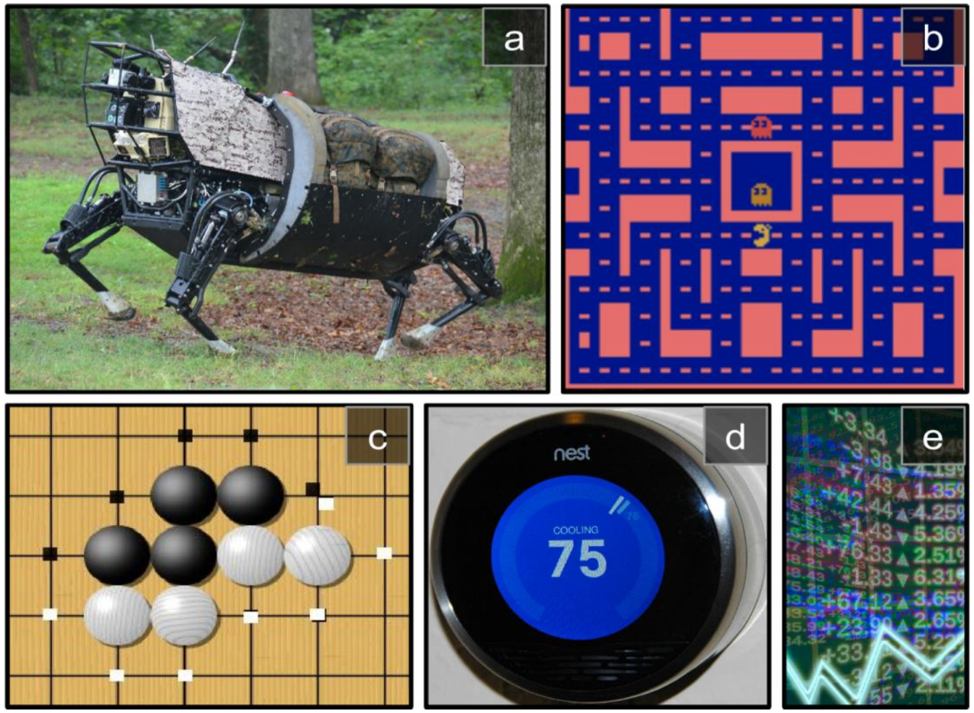
1. Basic principle of RL

- a. The agent controls a robot. It observes through sensors, and its actions consist of sending signals to activate motors. It gets positive rewards whenever it approaches the target destination, and negative rewards whenever it wastes time, goes in the wrong directidirection, or falls down.
- b. The agent can be the program controlling Pac-Man.on, or falls down.
- c. The agent can be the program playing a board game such as the game of Go.



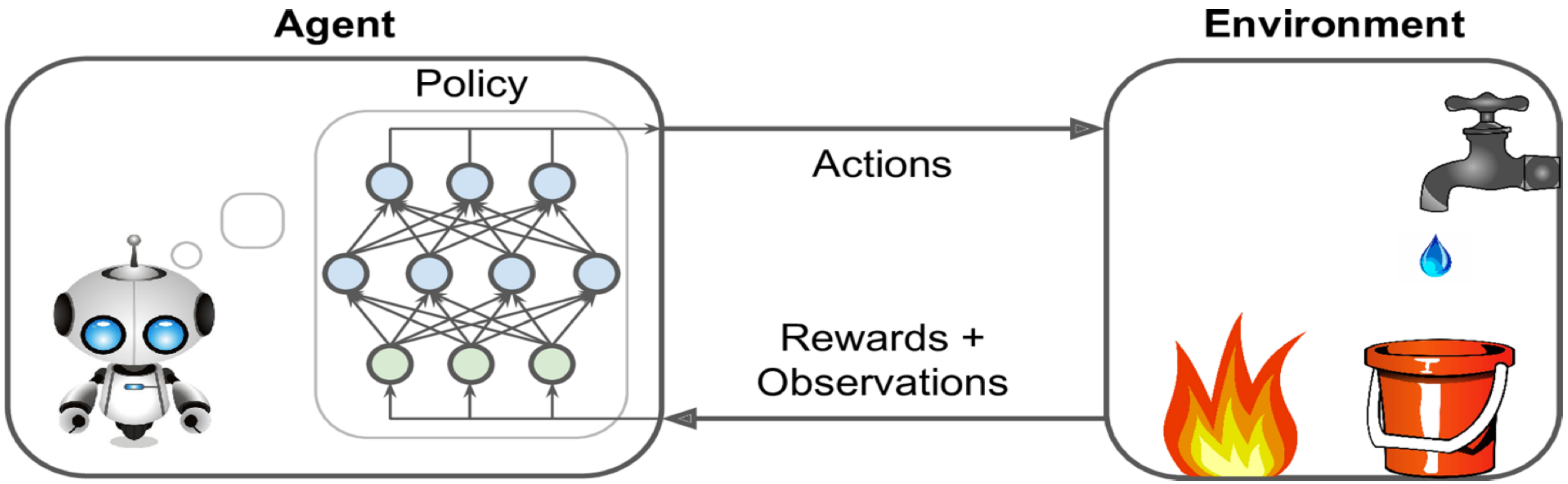
1. Basic principle of RL

- d. A smart thermostat, getting rewards whenever it is close to the target temperature and saves energy, and negative rewards when humans need to tweak the temperature, so the agent must learn to anticipate human needs.
- e. The agent (automatic trader) can observe stock market prices and decide how much to buy or sell every second. Rewards are obviously the monetary gains and losses.



2. **Policy Search** (Algorithm Training)

- The algorithm used by the software agent to determine its actions is called its policy. For example, the policy could be a neural network taking observations as inputs and outputting the action to take:



- The policy can be any algorithm you can think of, and it does not even have to be deterministic. It can be trained.

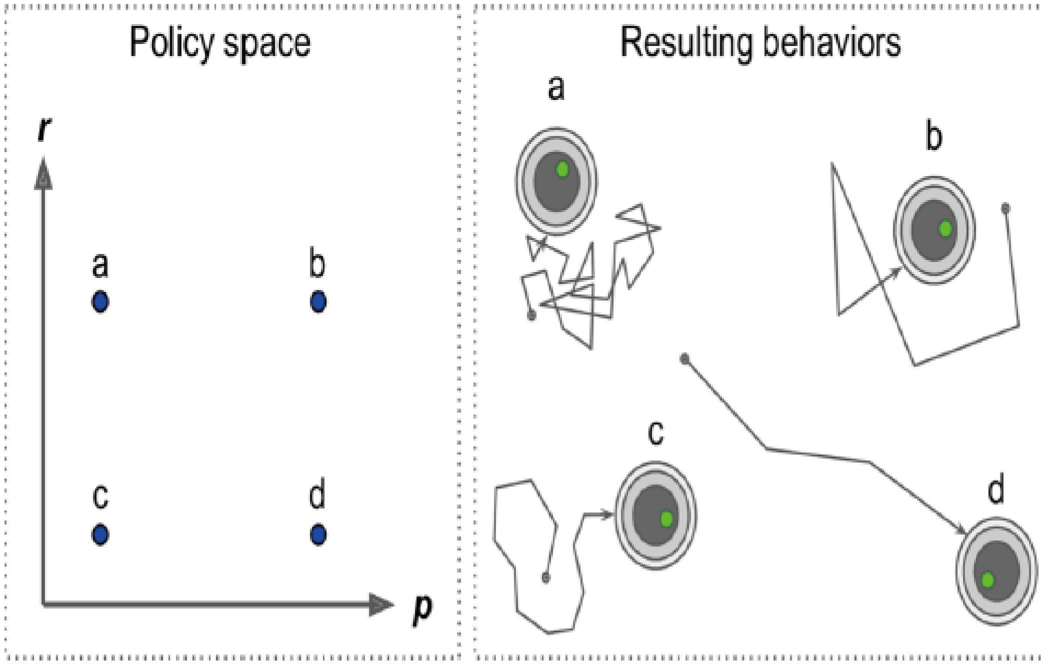
2. **Policy Search** (Algorithm Training)

- For examples:
- A vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes.
- Its policy could be to move forward with some probability p every second, or randomly rotate left or right with probability 1 – p . The rotation angle would be a random angle between –r and +r . It is called a stochastic policy (or algorithm).

How to train such a robot?

- There are just two policy parameters: the probability p and the angle range r .
- Way 1:
- One possible learning algorithm： try out many different values for these parameters, and pick the combination that performs best (to achieve max reward for a log run).
- However, when the policy space is too large, finding a good set of parameters is too cost in computing.

- Say case a: *r* is larger and *p* is smaller



How to train such a robot?

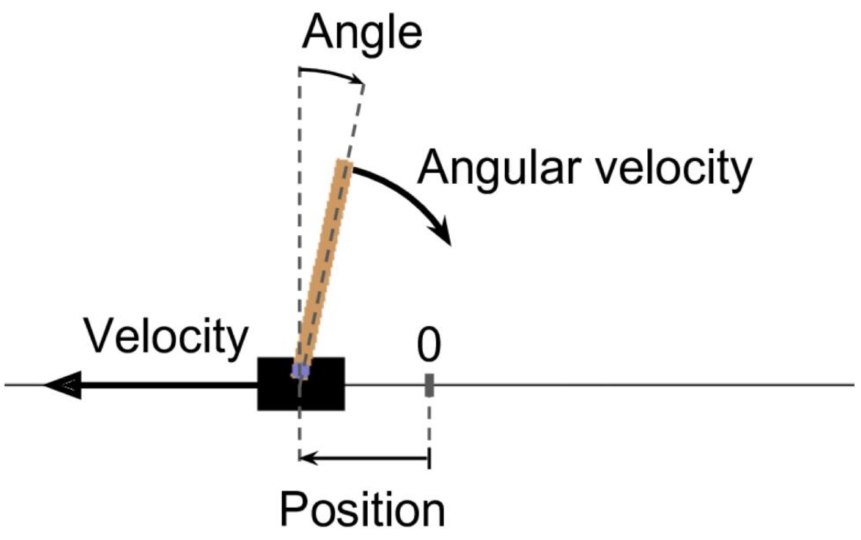
- Way 2:
- The genetic algorithm is a method for solving both constrained and unconstrained optimization problems.
- It is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions.
  - https://www.obitko.com/tutorials/genetic-algorithms/ga-basic-description.php
- Way 3:
- By evaluating the gradients of the rewards with regards to the policy parameters, (for example, parameters p and r with the robot) then tweaking these parameters by following the gradient toward higher rewards (gradient ascent). But, how to express or calculate reward? (later)
- This approach is called Policy Gradients (PG), more detail later.

3. Introduction to OpenAI Gym

- One of the challenges of Reinforcement Learning is that in order to train an agent, you first need to have a working environment.
- If you learn to play an Atari game, you will need an Atari game simulator.
- If you want to program a walking robot, then the environment is the real world and you can directly train your robot in that environment, but this has its limits: if the robot falls off a cliff, you can't just click "undo."
- OpenAI gym is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new RL algorithms.
- Let's install OpenAI gym and have a practice with it in Lab class.

4. Neural network Policies

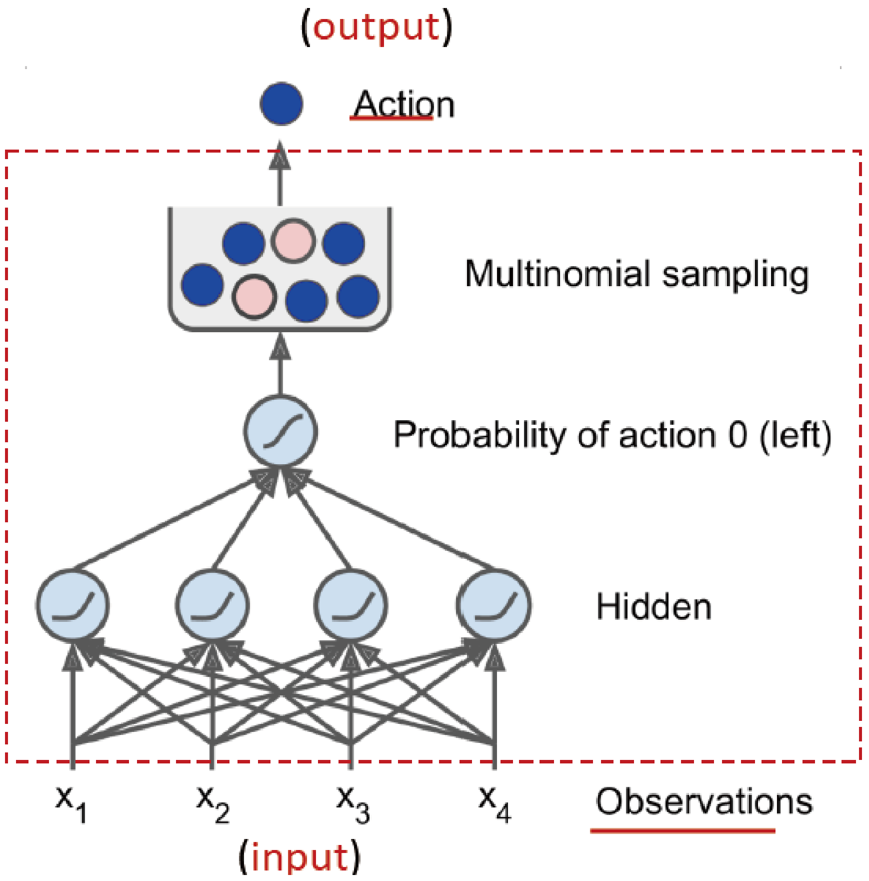
- By using an example in the CartPole environment, where each observation deals with four features:
- The cart's horizontal position (0.0 = center), its velocity, the angle of the pole (0.0 = vertical), and its angular velocity， as indicated.



- Let's create a neural network policy or algorithm.
- This neural network, as agent, will take an observation as input with four features, and it will output the action to be executed in this example.

4. Neural network Policies

- Specifically, it will estimate a probability for each action, and then we will select an action randomly according to the estimated probabilities (see the diagram).
- In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron.



- It will output the probability p of action 0 (left), and the probability of action 1 (right) will be 1 – p .
- For example, if it outputs 0.7, then we will pick action 0 with 70% probability, and action 1 with 30% probability.

The discussion about two “Why”?

- Why are we picking a random action based on the probability given by the neural network, rather than just picking the action with the highest score?
- Also note that in this particular environment, the past actions and observations can safely be ignored, why?

5. Evaluating Actions

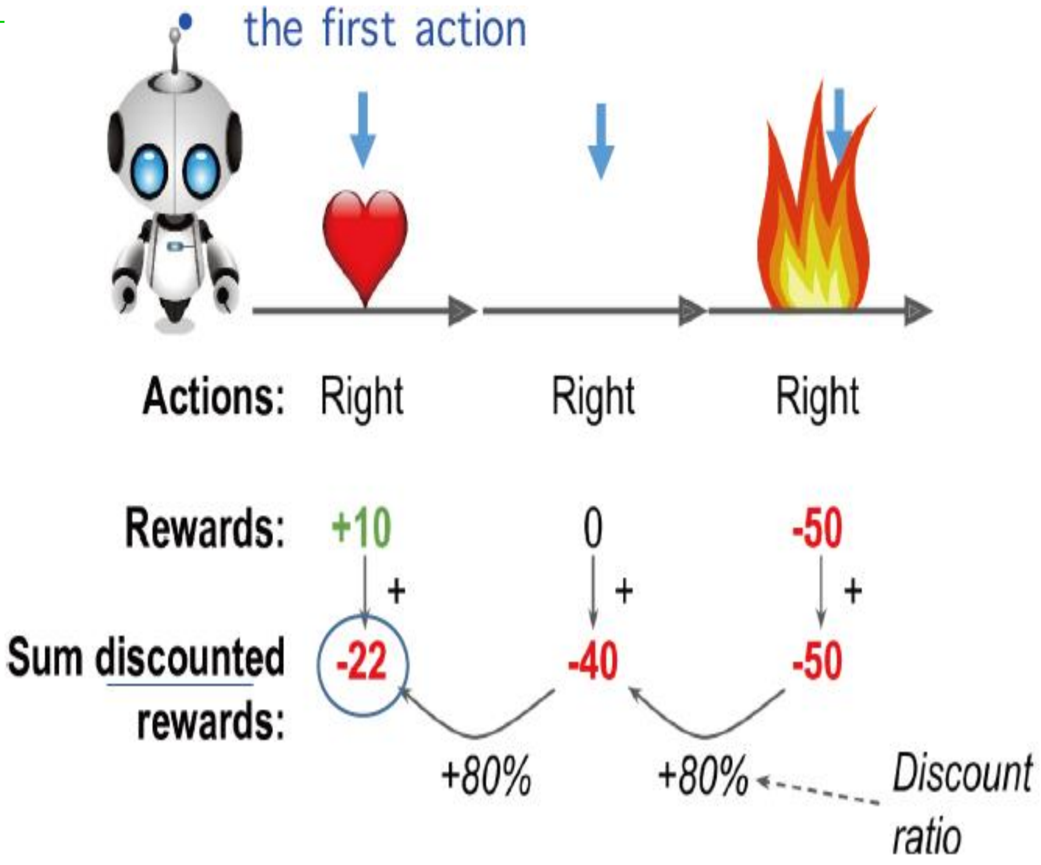
- If we knew what the best action was at each step, we could train the neural network as a regular supervised learning. However, in Reinforcement Learning the only guidance the agent gets is through rewards.
- For example, if the agent manages to balance the pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? Also, you cannot say the last action is entirely responsible for the pole fell after it. This is called the credit assignment problem.
- The solution is to evaluate an action based on the sum of all the rewards that come after it. Once we can get the sum at each step, we can set the action at each step.
- How to calculate the sum of all the rewards? usually applying a discount rate r at each step.

5. Evaluating Actions

- if an agent decides to go right three times in a row and gets +10, 0, and -50 reward, then assuming we use a discount rate r = 0.8, the first action will have a total score (or reward) of

$r^0 \times 10 + r^1 . 0 + r^2 (-50) = -22.$

- Typical discount rates are 0.95 or 0.99.
- If the discount rate is close to 0, then future rewards won't count for much compared to immediate rewards.



5. Evaluating Actions

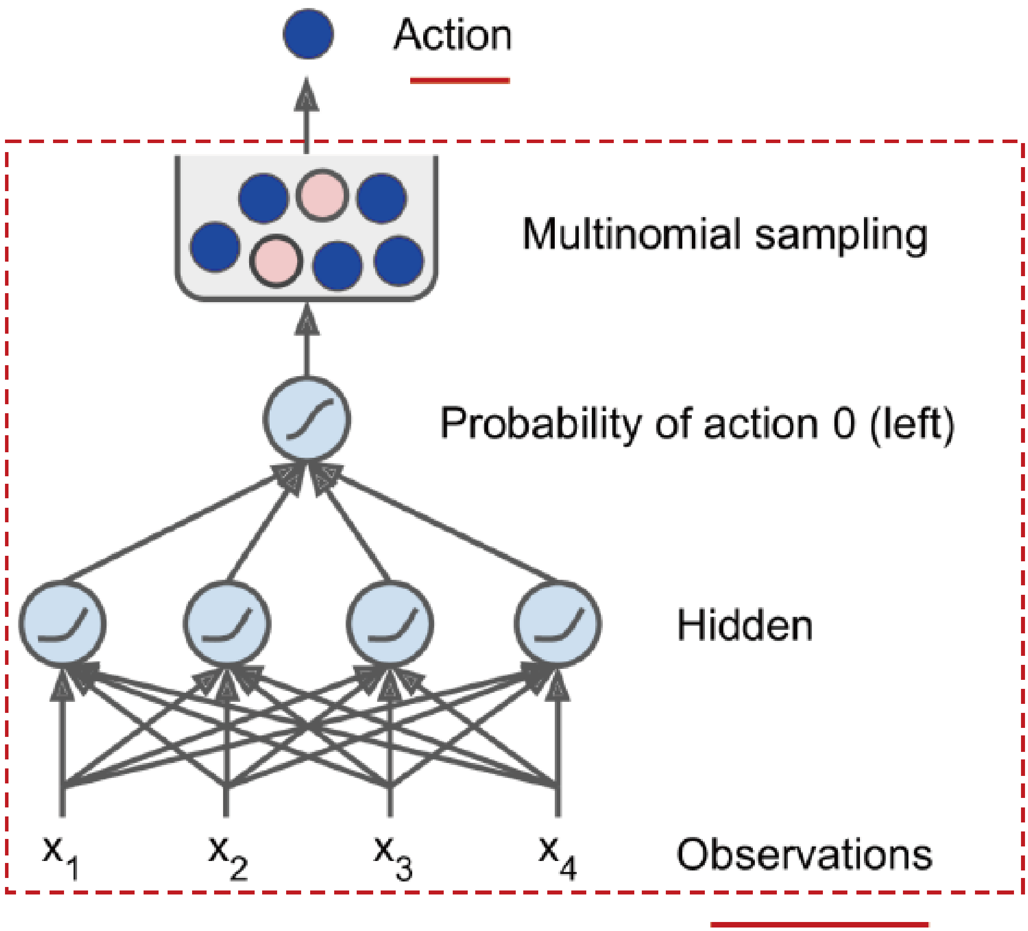
- In the CartPole environment, actions have fairly short-term effects, so choosing a discount rate of 0.95 seems reasonable.
- After that, we can reasonably assume that actions with a negative score were bad while actions with a positive score were good.
- Now, we have a way to evaluate each action, we are ready to train our first agent using policy gradients. Let’s see how.

6. Policy Gradients (PG)

- For example, the policy could be an algorithm of a neural network taking observations as inputs and outputting the action to take.
- How to train this policy? There are different ways. One of ways is to evaluate the gradients of the rewards with regards to the policy parameters,
- PG algorithms optimize the parameters of a policy by following the gradients toward higher rewards. (This is opposite to the minimum of cost function in ML discussed before).
- One popular class of PG algorithms, called REINFORCE algorithm (next slide)
- This algorithm is implemented by using TensorFlow. We will see its codes in the notebook, Reinforcement_learning.ipynb, given in …/Lab_10/Lab10_tasks.

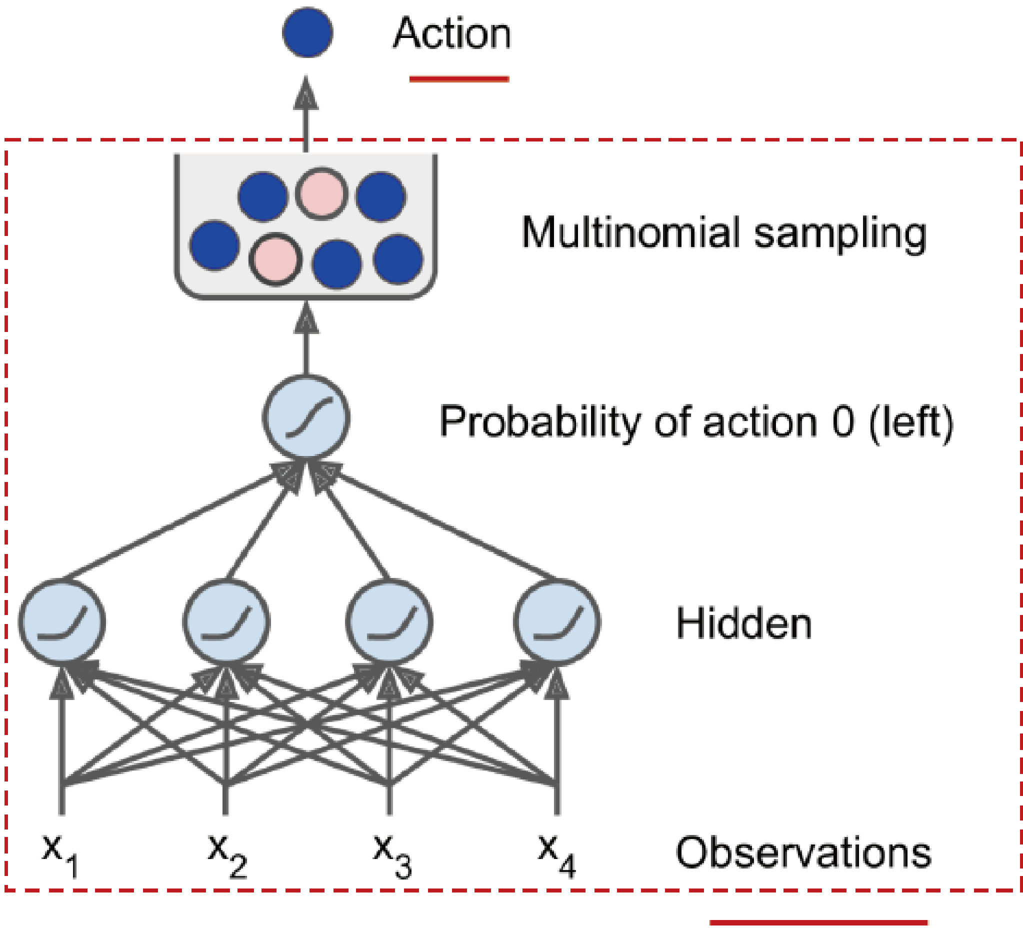
REINFORCE algorithm

- It gives a way to compute the gradients for carrying out a Gradient Descent step for training the neural network. The steps:
- 1. let the neural network policy play the game several times and at each step compute the gradients, based on the chosen actions even more likely, but don't apply these gradients yet. (Similar to random initialization)
- 2. Once you have run several episodes, compute each action's reward (similar to calculate cost function, such MSE), using the method just discussed.



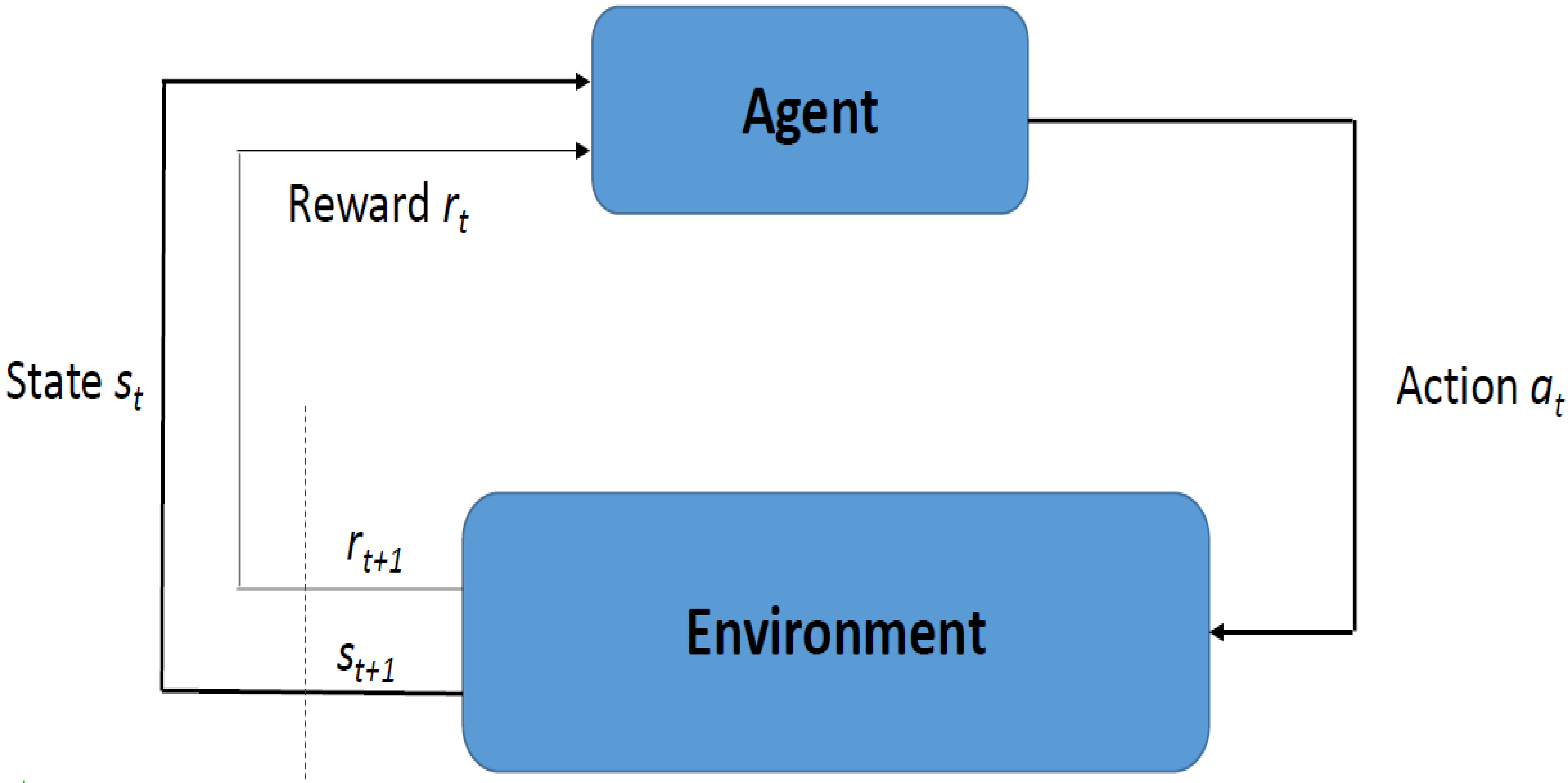
REINFORCE algorithm

- If an action's reward is positive ("good for action"), it means that the gradients are more likely to be chosen in the future, otherwise, less likely applied in the future. The solution (for reflecting such fact) is simply to multiply each gradient victor by the corresponding action's reward.
- Finally, compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step for training the neural network.



Another popular family of algorithms

- PG algorithms directly try to optimize the policy to increase rewards, the algorithms we will look at now are less direct:
- The agent learns to estimate the expected sum of discounted future rewards
  - for each state, or
  - for each action in each state.
- then uses this knowledge to decide how to act.
- To understand these algorithms, we must first introduce Markov decision processes (MDP).

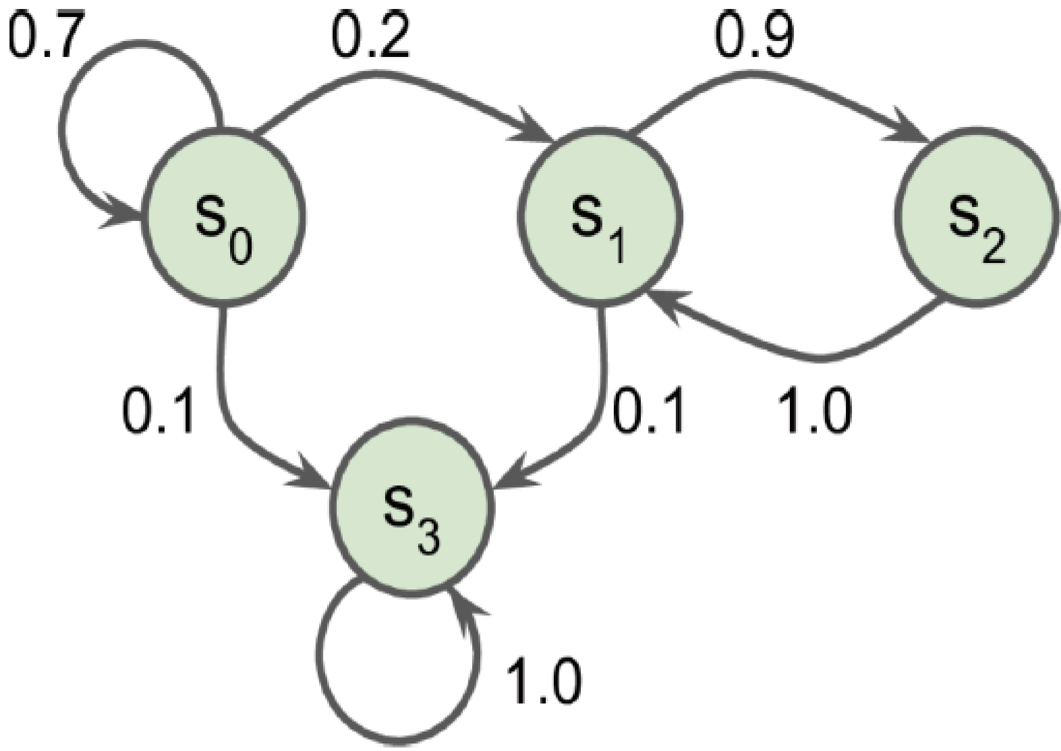


- **References:**
- https://skymind.ai/wiki/deep-reinforcement-learning

7. Markov Decision Processes

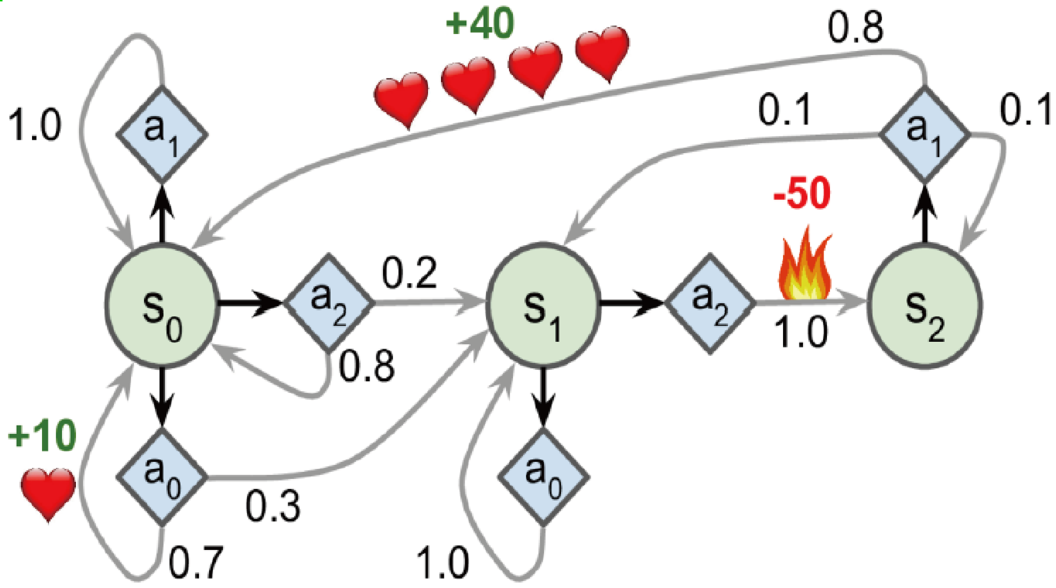
- In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called Markov chains.
- Such a process has a fixed number of states, and it randomly evolves from one state to another at each step.
- The probability for it to evolve from a state s to a state s′ is fixed, and it depends only on the pair (s,s'), not on past states (the system has no memory).

a Markov chain with four states



7. Markov Decision Processes

- At each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action.
- Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize rewards over time.

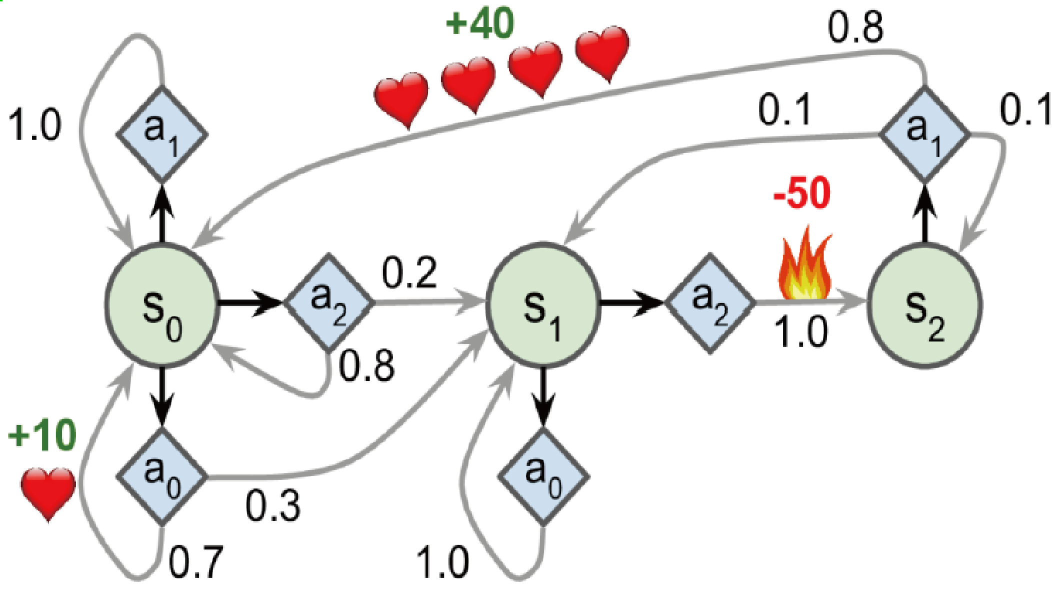


The MDP has three states and up to three possible discrete actions at each step

- If it starts in state s0, the agent can choose between actions a0, a1, or a2. If it chooses action a1, it just remains in state s0 with certainty, and without any reward. It can thus decide to stay there forever.
- But if it chooses action a0, it has a 70% probability of gaining a reward of +10, and remaining in state s0.

7. Markov Decision Processes

- It can then try again and again to gain as much reward as possible. But at one point it is going to end up instead in state s1.
- In state s1 it has only two possible actions: a0 or a1. It can choose to stay put by repeatedly choosing action a1, or it can choose to move on to state s2 and get a negative reward of –50 (ouch).

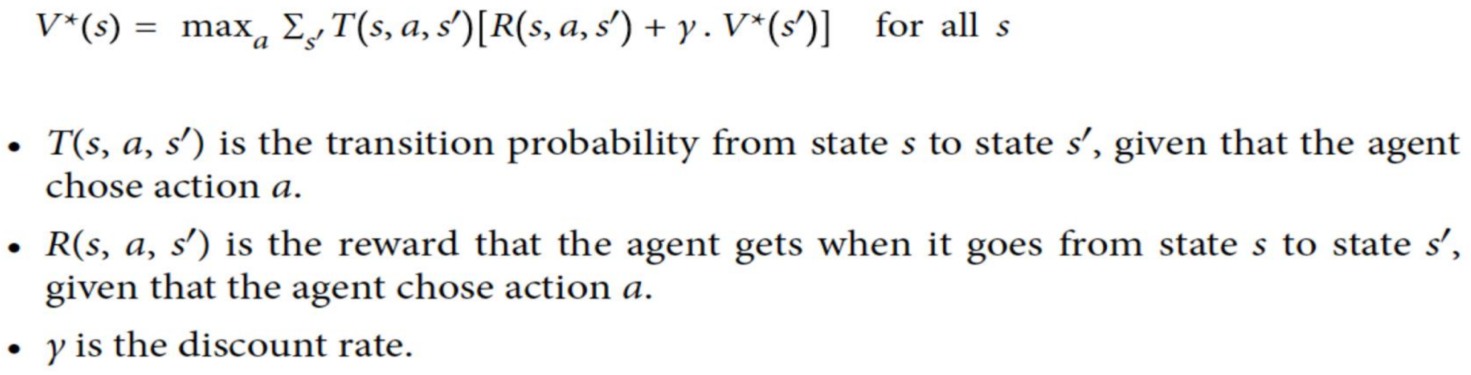


The MDP has three states and up to three possible discrete actions at each step

- In state s2 it has no other choice than to take action a1, which will most likely lead it back to state s0, gaining a reward of +40 on the way.

7. Markov Decision Processes

- Bellman found a way to estimate the optimal state value of any state s, noted V*(s), which is the sum of all discounted future rewards the agent can expect on average after it reaches a state s, assuming it acts optimally:



$$V^*(s) = \max_a \Sigma_{s'} T(s, a, s')[R(s, a, s') + \gamma . V^*(s')] \quad \text{for all } s$$

- $T(s, a, s')$ is the transition probability from state s to state s', given that the agent chose action a.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a.
- γ is the discount rate.

- So, given T(s, a, s') , R(s, a, s') and ! ,#e can determine V*(s) in math.

7. Markov Decision Processes

- The following equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state:
- you first initialize all the state value estimates to zero, and then you iteratively update them using the Value Iteration Algorithm:

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \, . \, V_k(s')\right] \quad \text{for all } s$$

- $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

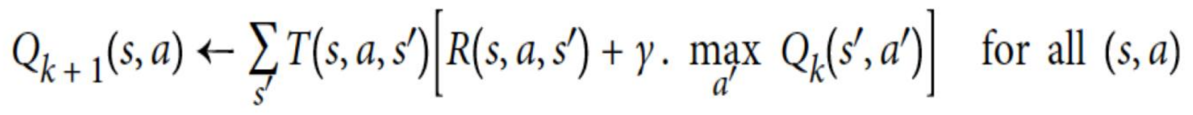
- This is a better way for computing literately

7. Markov Decision Processes

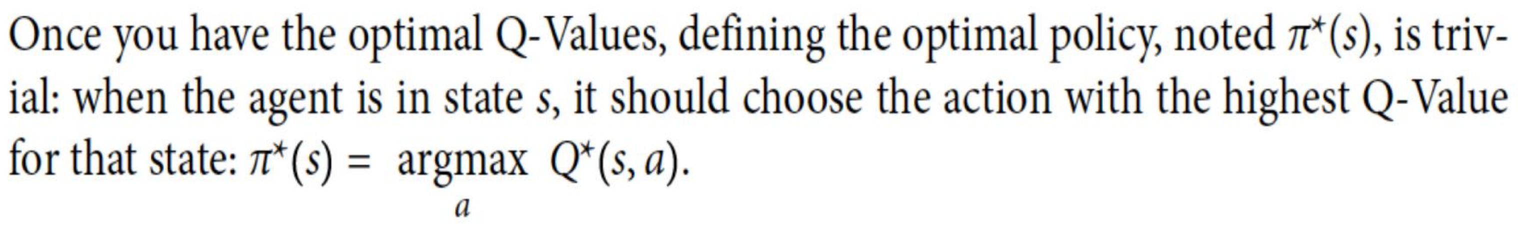
- The algorithm above does not tell the agent explicitly what to do (which action should be taken? No answer).
- Bellman found a very similar algorithm to estimate the optimal stateaction values Q*(s, a), generally called Q-Values.
- The optimal Q-Value of the state-action pair (s, a), noted Q*(s, a), is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a, but before it sees the outcome of this action, assuming it acts optimally after that action.
- The next slide shows how it works: once again, you start by initializing all the Q-Value estimates to zero, then you update them using the Qvalue Iteration algorithm :

Markov Decision Processes

- Once a neural network is trained to implement this model or algorithm, which can be used for an agent to make action at each step of observation.
- Q-Value Iteration algorithm



$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s,a)$$



Once you have the optimal Q-Values, defining the optimal policy, noted $\pi^*(s)$, is trivial: when the agent is in state s, it should choose the action with the highest Q-Value for that state: $\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s,a)$.

- The codes that apply this algorithm to the MDP in the figure in previous dlide, are given in notebook, Reinforcement_learning.ipynb, given in …/Lab_10/Lab10_tasks.
- We have a look at them in Lab class.

8. Temporal Difference Learning and Q-Learning

- Reinforcement Learning problems with Markov decision processes has an issue:
- The agent initially has no idea what the transition probabilities (it does not know T(s, a, s')), and it does not know what the rewards are going to be either (it does not know R(s, a, s')).
- It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.
- So, we need a "learning algorithm".
- The Temporal Difference Learning (TD Learning) algorithm

8. Temporal Difference Learning and Q-Learning

- In general, assuming that the agent initially knows only the possible states and actions, and nothing more.
- The agent uses an exploration policy:
- for example, a purely random policy—to explore the MDP, and as it progresses the TD Learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed:

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \,.\, V_k(s')\right] \quad \text{for all } s$$

- $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

- α is the learning rate (e.g., 0.01).

- For each state s, this algorithm simply keeps track of a running average of the immediate rewards the agent gets upon leaving that state, Vk(s), plus the rewards it expects to get later, Vk(s'), (assuming it acts optimally).
- Similarly, the Q-Learning algorithm is an adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown:

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\left(r + \gamma \cdot \max_{a'} Q_k(s',a')\right)$$

$$\underline{Q_{k+1}(s,a)} \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s,a)$$

Once you have the optimal Q-Values, defining the optimal policy, noted $\pi^*(s)$, is trivial: when the agent is in state s, it should choose the action with the highest Q-Value for that state: $\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s,a)$.

TD Learning algorithm

- **For each state-action pair (s, a), this algorithm keeps track of a running average of the rewards the agent gets upon leaving the state s with action a, plus the rewards it expects to get later.**
- Since the target policy would act optimally, we take the maximum of the Q-Value estimates for the next state.
- In Lab class, we will look at how Q-Learning is implemented in notebook, Reinforcement_learning.ipynb, given in …/Lab_10/Lab10_tasks.

About Lab 10

- Two samples of Lab10 work well with Linux Virtual Box!

- Note:
- When you run the codes in the notebooks, please take one cell by one cell slowly, don't do "restart & run all", as graphical animation takes time to run in a sequence of frames. So, you have to wait each frame display finished.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

结束

2020年5月14日