Week 11&12: Reinforcement_Learning

1. Introduction

In this Week, we focus on how agents can learn what to do in the absence of labeled examples of what to do.

1.1. FrameWork

- we know the state space, the action set
- we don not know the transition model and reward mechanism
- agents regards the reward as part of the input percept

- an agent must experience as much as possible of its environment in order to learn how to behave in it (exploration)

1.2. Assumption

- a fully observable environment

1.3. Basic Methods

1. Utility-based agent

learns a **utility function** on states and uses it to select actions that maximize the expected outcome utility

- utility-based agent must also have a model of the environment in order to make decisions

2. Q-learning

learns an action-utility function, or **Q-function**, giving the expected utility of taking a given action in a given state.

- Q-learning agents cannot look ahead
- Q-learning agents do not need to have a model of the environment

3. reflex agent

2. Model-Free Reinforcement Learning

2.1. Passive Reinforcement Learning

Given a policy, try to learn how good the policy is—that is, to learn the utility function

2.1.1. Model

- transition model and reward function is unknown
- In this part, our agent execute fixed policy many times and record the reward and state.
- In this part, we assume we receive reward after each step

2.1.2. Direct Utility Estimation (Monte Carlo Estimation)

the idea is that the utility of a state is the expected total reward from that state onward (called the **expected reward-to-go**), and each trial provides a sample of this quantity for each state visited.

That means:

1. calculate the utility of each state from the end of each trial just using simple addition
2. calculate average if a state appear several times in different trials\

Convergence

It is **unbiased** → will converge to right solution if you have **enough samples**

Flaw

- it misses a very important source of information, namely, the fact that **the utilities of states are not independent**!
  - The utility of each state equals its own reward plus the expected utility of its successor states
  - So the simple addition is not sufficient to calculate the Utility
  - Or we can say, it does not exploit any knowledge of the Bellman equation

2.1.3. Temporal-Difference learning

Background

Direct Utility method like Monte Carlo Estimation

- unbiased will converge to right solution
- high variance need many samples in practice

Idea

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha\left(R(s) + \gamma U^{\pi}\left(s'\right) - U^{\pi}(s)\right)$$

where:

$$\alpha \rightarrow \text{learning rate}$$

- TD does not need a transition model to perform its update
- The second part is the difference between prediction after observing the next reward and state

So this form use the information of Bellman equation

And is also called **“stochastic gradient descent” (SGD)**

It is actually **"Bootstrapping the target"**

Implementation

function PASSIVE-TD-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    U, a table of utilities, initially empty
    N_s, a table of frequencies for states, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if s' is new **then** $U[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma\, U[s'] - U[s])$
  if s'.TERMINAL? **then** $s, a, r \leftarrow$ null **else** $s, a, r \leftarrow s', \pi[s'], r'$
  return a

Figure 21.4 A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function $\alpha(n)$ is chosen to ensure convergence, as described in the text.

Properties

- Don't need to wait until end of episode to update values
- **biased**

2.2. Active Reinforcement Learning

2.2.1. Target

An active agent must decide what actions to take

- First, the agent will need to learn a complete model with outcome probabilities for all actions, rather than just the model for the fixed policy PASSIVE_ADP_AGENT
- Second, the agent should has some choice of actions

2.2.2. Active ADP Agent

2.2.3. Q-learning

Idea

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

Implementation

function Q-LEARNING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r'
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if TERMINAL?(s) **then** $Q[s, None] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$
    $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
  $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$
  return a

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s, a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

Property

off-policy: Q-learning uses the best Q-value, it pays no attention to the actual policy being followed

2.2.4. SARSA

Idea

$$\mathrm{Q(s,a)} := \mathrm{Q(s,a)} + \mathrm{a}\left[\mathrm{r} + \mathrm{yQ}\left(\mathrm{s}', \mathrm{a}'\right) - \mathrm{Q(s,a)}\right]$$

Property

on-policy: it pays atention on actual policy being followed

2.2.5. SARSA vs Q-learning

- Q-learning is **more flexible** than SARSA, in the sense that a Q-learning agent can learn how to behave well even **when guided by a random or adversarial exploration policy**.
- SARSA is **more realistic**: for example, if the overall policy is even partly **controlled by other agents**, it is better to learn a Q-function for what will actually happen rather than what the agent would like to happen.

- **off-policy learning has great promise**: learn about optimal values following any policy
- but when using **'function approximation' SARSA** tends to be more stable

2.3. Passive vs Active

- Passive learning $\rightarrow$the agent's policy is fixed which means that it is told what to do, the goal of the agent is to evaluate how good an policy is
- Active learning $\rightarrow$ an agent needs to decide what to do as there's no fixed policy that it can act on. learn **which actions** to take

2.4. Policy Research

Basic Idea

- not bother with value functions Q/V
- **directly parametrize policy** $\pi(a|s;w)$
- update these parameters based on the returns u(s) observed

2.4.1. Policy Gradient Methods

Basic Process

1. Start out with an arbitrary random policy
2. **Sample some actions** in the environment
3. If rewards are better than expected, increase probability of taking those actions, vice versa.

$$\mathrm{w}_{\mathrm{t}+1} := \mathrm{w}_{\mathrm{t}} + \mathrm{a} \cdot \mathrm{u}\left(\mathrm{S}_{\mathrm{t}}\right) \cdot \nabla \log \pi\left(\mathrm{a}_{\mathrm{t}} \mid \mathrm{s}_{\mathrm{t}}; \mathrm{w}_{\mathrm{t}}\right)$$

Why log

- Probability when multiplied repeatedly can approach zero
- Logarithms are more numerically stable, perform better

2.5. Actor-Critic Methods

Idea

Combined policy gradient methods with estimated Q-value function

- policy: actor $\rightarrow$ tries to take good actions

- value function: critic → gives feedback to policy

Property

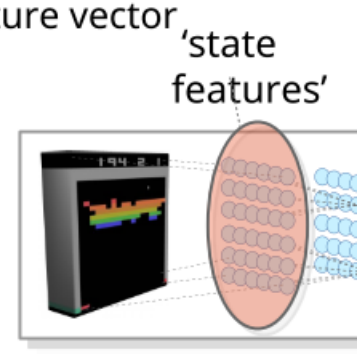
addresses the high variance that Policy Gradient methods (working directly on returns) otherwise have.

Intuition

Actor看到游戏目前的state，做出一个action。
Critic根据state和action两者，对actor刚才的表现打一个分数。
Actor依据critic（评委）的打分，调整自己的策略（actor神经网络参数），争取下次做得更好。
Critic根据系统给出的reward（相当于ground truth）和其他评委的打分（critic target）来调整自己的打分策略（critic神经网络参数）。
一开始actor随机表演，critic随机打分。但是由于reward的存在，critic评分越来越准，actor表现越来越好。

2.6. Function Approximation

- For simplicity: policy evaluation
  - we want to find $V_\pi(s;w)$
  - w is a vector of parameters
- Using features:
  - $x(s)=(x_1(s),...,x_d(s))$ is a d-dimensional feature vector
  - $V_\pi(s;w) = f(x(s); w)$
- For instance...
  - f can be a neural network
  - or a linear function $V_\pi(s;w) = w^T x(s)$



- F.A. in TD
  - target is $T(s) = r + \gamma V_k(s')$
  - we minimize the squared TD error:

$$w_{k+1} := w_k - \alpha \nabla [T(s) - V(s;w_k)]^2 / 2$$
$$= w_k + \alpha [r + \gamma V(s';w_k) - V(s;w_k)] \nabla V(s;w_k)$$

- E.g., for linear function approximation:
  - $\nabla V(s;w_k) = \nabla w^T x(s) = x(s)$
  - $w_{k+1} := w_k + \alpha[r + \gamma V(s';w_k) - V(s;w_k)] \; x(s)$

- Q-learning is adapted similarly

- What is optimal under function approximation?
  - 'best' value function in class
  - under some metric...

- Few of the nice properties remain...
  - SARSA with linear function approximation might **'chatter'** - it can cycle around the 'optimal' solution
  - Q-learning with non-linear function approximation: **can diverge**

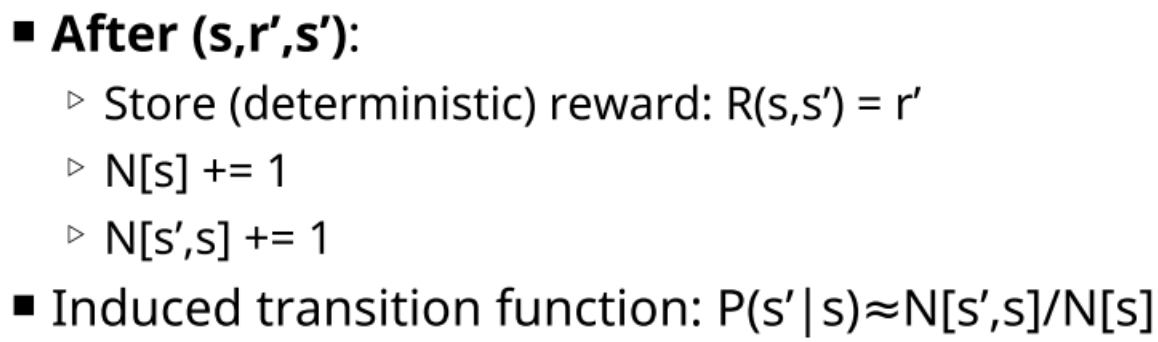
- For **passive TD learning**, the update rule can be shown to **converge** to the closest possible approximation to the true function when the function approximator is **linear** in the parameters.
- With **active learning** and **nonlinear functions** such as neural networks, all bets are off: There are some very simple cases in which the parameters can **go off to infinity** even though there are good solutions in the hypothesis space

3. Model-Based Reinforcement Learning

We maintain a belief over augmented states formulated as a triplet: state, transition counter and observation counter.

3.1. Adaptive Dynamic Programming (ADP)

3.1.1. Basic ADP



Property

- Need many data samples
- prone to overfitting

3.1.2. Integrated Model-Based Control

Idea

We try to optimize direct utility estimation by using Bellman Function.

So

- keep track of **how often each action outcome occurs**
- **estimate the transition probability P(s′ | s, a)** from the frequency with which s′ is reached when executing a in s

Implementation

function PASSIVE-ADP-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    mdp, an MDP with model P, rewards R, discount γ
    U, a table of utilities, initially empty
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    $N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s' is new **then** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$ and $N_{s'|sa}[s', s, a]$
    for each t such that $N_{s'|sa}[t, s, a]$ is nonzero **do**
      $P(t \mid s, a) \leftarrow N_{s'|sa}[t, s, a] \,/\, N_{sa}[s, a]$
  $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
  if s'.TERMINAL? **then** $s, a \leftarrow$ null **else** $s, a \leftarrow s', \pi[s']$
  return a

Figure 21.2 A passive reinforcement learning agent based on adaptive dynamic programming. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page 657.

Property

- may have impact of incorrect model, can lead to loss in value (**estimation error**)
- pure greedy agent might never find out that there is more reward to be found in different part of state space (**exploration**)

Potential Improvement

- Using a maximum likelihood estimate of true model**(prone to be overfitting)**
- Or we can use Bayesian estiamtion of model
- as MCTS, we can use exploration bonus

$$U(a) = Q(a) + c\sqrt{\log(N)/N_a}$$

Attention ways to try random (s,a) pairs may not as usual, because if we want to try (s,a) pairs, we need to get s first, but it cannot essentially solve the exploration problem

3.2. R-max (like) methods

- Use maintained model to plan as before… but:
  - Initialize optimistically
    - R(s,a) = Rmax
    - P(s'|s,a) = **I**{ s=s' } ← assume we stay in state s
  - Mark *(s,a)* pairs 'known' IFF taken at least *m* times

so we assume each unknown (s,a) pair corresponds to 'heaven'

- **After (s,a,r',s')**:
  - Store reward: Rset(s,a) = Rset(s,a) ∪ r'
  - Store transition: N[s',s,a] += 1, N[s,a] += 1
  - if N[s,a] == m:
    - R(s,a) := mean (Rset(s,a))
    - P(s'|s,a) := N[s',s,a] / N[s,a]
  - Plan next step with updated model

and we will plan to actually get to those unknown transitions

Algorithm 1: R-MAX

Input: S, A, γ, m, ϵ_1, $R_{\max}$

$\bar{S} \leftarrow S \cup \{z\}$, where z is an arbitrary fictitious state
foreach $(s,a) \in \bar{S} \times A$ **do**
  $n(s,a) \leftarrow 0$
  $r(s,a) \leftarrow 0$
  $\bar{Q}(s,a) \leftarrow R_{\max}/(1-\gamma)$
  $\bar{R}(s,a) \leftarrow R_{\max}$
  foreach $s' \in S$ **do**
    $n(s,a,s') \leftarrow 0$
    $\bar{T}(s,a,s') \leftarrow 0$
  end
  $n(s,a,z) \leftarrow 0$
  $\bar{T}(s,a,z) \leftarrow 1$
end
for $t = 1, 2, 3, \ldots$ **do**
  Observe current state s
  Execute action $a := \text{argmax}_{a' \in A} \bar{Q}(s,a')$
  Observe immediate reward r and next state s'
  if $n(s,a) < m$ **then**
    $n(s,a) \leftarrow n(s,a) + 1$
    $r(s,a) \leftarrow r(s,a) + r$
    $n(s,a,s') \leftarrow n(s,a,s') + 1$
    if $n(s,a) = m$ **then**
      $\bar{R}(s,a) \leftarrow r(s,a)/m$
      foreach $s'' \in \bar{S}$ **do** $\bar{T}(s,a,s'') \leftarrow n(s,a,s'')/m$
      $\bar{Q} \leftarrow$ Solve $(\bar{S}, A, \bar{R}, \bar{T}, \gamma, \epsilon_1)$ using VI
    end
  end
end

3.3. Bayesian RL