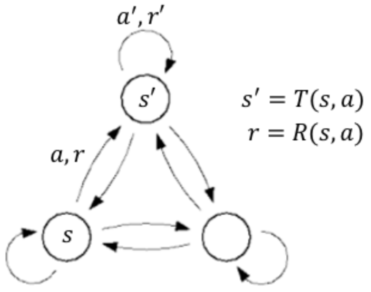
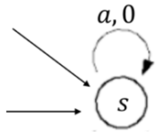


图 12.1　马尔可夫模型的状态图



$T(s,a) = s, \forall a \in A$

$R(s,a) = 0, \forall a \in A$

图 12.2　终止状态示意图

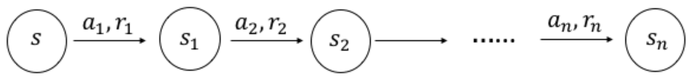


图 12.3　长度为 n 的状态路径

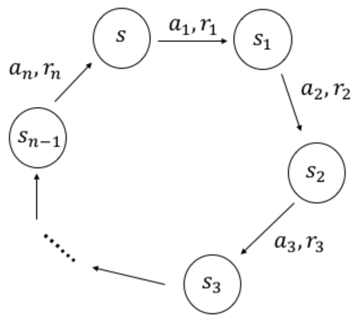


图 12.4　形成一个圈的状态路径

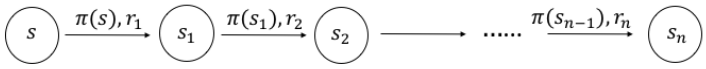


图 12.5　策略 π 从状态 s 发出的 n 个行动的状态图

值迭代算法

```
Compute _environment_Q _value( S, A, T, R, γ):
    For all s ∈ S: V_0(s) = 0
    n = 0
    While True:
        n ← n + 1
        For all s ∈ S:
            For all a ∈ A:
                Q_n(s, a) = R(s, a) + γV_{n-1}(T(s, a))
        For all s ∈ S:
            V_n(s) = max_{a∈A} Q_n(s, a)
        If for all s ∈ S : V_n(s) = V_{n-1}(s) :
            Break
    Return Q_n

Value_Iteration ( S, A, T, R, γ):
    Q = Compute_environment_Q_value ( S, A, T, R, γ)
    For all s ∈ S:
        π(s) = argmax_{a∈A} Q(s, a)
    Return π
```

图 12.6　值迭代算法描述

machine_learning.reinforcement_learning.environment

```python
import numpy as np

class Environment:
    S = []
    A = []
    T = []
    R = []
    S_end = {}

    def reset(self):
        s_start = np.random.randint(0, len(self.S))
        return s_start
```

图 12.7　Envinronment 类

machine_learning.reinforcement_learning.value_iteration

```
import numpy as np

def compute_environment_Q_values(S, A, T, R, gamma):
    V = np.zeros(len(S))
    Q = np.zeros((len(S), len(A)))
    while True:
        for s in S:
            for a in A:
                Q[s][a] = R[s][a] + gamma * V[int(T[s][a])]
        converge = True
        for s in S:
            if np.max(Q[s]) != V[s]:
                converge = False
            V[s] = np.max(Q[s])
        if converge:
            break
    return Q

def value_iteration(env, gamma):
    S,A,T,R = env.S, env.A, env.T, env.R
    Q = compute_environment_Q_values(S, A, T, R, gamma)
    pi = np.argmax(Q, axis = 1)
    return pi
```

图 12.8　值迭代算法

图 12.9　地雷与宝藏游戏

machine_learning.reinforcement_learning.landmine_and_treasure

```
import numpy as np
from machine_learning.reinforcement_learning.environment import Environment

class LandmineAndTreasure(Environment):
    def __init__(self, m, d):
        n_states = m
        n_actions = 2
        S = range(n_states)
        S_end = {0, d, n_states - 1}
        A = range(n_actions)
        R = np.zeros((n_states, n_actions))
        R[n_states - 2][1] = 1000
        R[1][0] = 1000
        R[d + 1][0] = -1000
        R[d - 1][1] = -1000
        T = np.zeros((n_states, n_actions))
        for s in S:
            T[s][0] = s - 1
            T[s][1] = s + 1
            if s in S_end:
                T[s][0] = s
                T[s][1] = s
        self.S, self.A, self.T, self.R, self.S_end = S, A, T, R, S_end
```

图 12.10　生成 LandmineAndTreasure 子环境的算法

```python
from machine_learning.reinforcement_learning.value_iteration import value_iteration
import machine_learning.reinforcement_learning.landmine_and_treasure as game

env = game.LandmineAndTreasure(m = 100, d = 30)
pi = value_iteration(env, 0.95)
print(pi)
```

图 12.11　小机器人的最优行走策略的值迭代算法

策略迭代算法

Compute_policy_Q_value (S, A, T, R, γ, π):

For all $s \in S$: $V_0^\pi(s) = 0$

$n = 1$

While True:

$n \leftarrow n + 1$

For all $s \in S$:

For all $a \in A$:

$Q_n^\pi(s, a) = R(s, a) + \gamma V_{n-1}^\pi\big(T(s, a)\big)$

For all $s \in S$:

$V_n^\pi(s) = Q_n^\pi(s, \pi(s))$

If $V_n^\pi = V_{n-1}^\pi$:

Break

Return Q_n^π

Policy_Iteration(S, A, T, R, γ):

For all $s \in S$:

$\pi_0(s) = \text{argmax}_{a \in A} R(s, a)$

$n = 0$

While True:

$Q^{\pi_n} =$ Compute_policy_Q_value ($S, A, T, R, \gamma, \pi_n$)

For all $s \in S$:

$\pi_{n+1}(s) = \text{argmax}_{a \in A} Q^{\pi_n}(s, a)$

If $\pi_{n+1} = \pi_n$:

Break

$n \leftarrow n + 1$

Return π_n

图 12.12 策略迭代算法描述

machine_learning.reinforcement_learning.policy_iteration

```python
import numpy as np

def compute_policy_Q_values(S, A, T, R, gamma, pi):
    V = np.zeros(len(S))
    Q = np.zeros((len(S), len(A)))
    while True:
        for s in S:
            for a in A:
                Q[s][a] = R[s][a] + gamma * V[int(T[s][a])]
        converge = True
        for s in S:
            if Q[s][pi[s]] != V[s]:
                converge = False
            V[s] = Q[s][pi[s]]
        if converge:
            break
    return Q

def policy_iteration(env, gamma):
    S,A,T,R = env.S, env.A, env.T, env.R
    pi = np.argmax(R, axis = 1)
    while True:
        Q = compute_policy_Q_values(S, A, T, R, gamma, pi)
        converge = True
        for s in S:
            if np.argmax(Q[s]) != pi[s]:
                converge = False
            pi[s] = np.argmax(Q[s])
        if converge:
            break
    return pi
```

图 12.13　策略迭代算法

ϵ 贪心探索策略

Epsilon_greedy ($\tilde{Q}, s, A, \epsilon$):

r = random number $\in [0,1)$

If $r < \epsilon$:

Return random $a \in A$

Else:

Return $\text{argmax}_{a\in A}\tilde{Q}(s,a)$

图 12.14 ϵ 贪心探索策略描述

Sarsa 算法

Sarsa(S, A, T, R, N, γ, ϵ, η):

For all $s \in S, a \in A$: $\tilde{Q}(s,a) = 0$

For $i = 1,2,\dots,N$:

s_{cur} = initial state in S

While $s_{\text{cur}} \notin$ {terminal states of S}:

a_{cur} = Epsilon_greedy($\tilde{Q}$, s_{cur}, A, ϵ)

$s_{\text{next}} = T(s_{\text{cur}}, a_{\text{cur}})$

a_{next} = Epsilon_greedy($\tilde{Q}$, s_{next}, A, ϵ)

$\tilde{Q}(s_{\text{cur}}, a_{\text{cur}}) \leftarrow (1-\eta)\tilde{Q}(s_{\text{cur}}, a_{\text{cur}}) + \eta\left(R(s_{\text{cur}}, a_{\text{cur}}) + \gamma\tilde{Q}(s_{\text{next}}, a_{\text{next}})\right)$

$s_{\text{cur}} = s_{\text{next}}$

For all $s \in S$:

$\pi(s) = \text{argmax}_{a \in A}\tilde{Q}(s,a)$

Return π

图 12.15　Sarsa 算法描述

machine_learning.reinforcement_learning.epsilon_greedy

```python
import numpy as np

def epsilon_greedy(Q_s, n_actions, epsilon):
    if np.random.rand() < epsilon:
        return np.random.randint(n_actions)
    else:
        return np.argmax(Q_s)
```

图 12.16　ϵ 贪心探索策略

machine_learning.reinforcement_learning.sarsa

```python
import numpy as np
from machine_learning.reinforcement_learning.epsilon_greedy import epsilon_greedy

def sarsa(env, N, gamma, epsilon, eta):
    S, A, T, R = env.S, env.A, env.T, env.R
    n_states, n_actions = len(S), len(A)
    Q = np.zeros((n_states, n_actions))
    for iter in range(N):
        env.reset()
        s_cur = env.s_start
        while s_cur not in env.S_end:
            a_cur = epsilon_greedy(Q[s_cur], n_actions, epsilon)
            s_next = int(T[s_cur][a_cur])
            a_next = epsilon_greedy(Q[s_next], n_actions, epsilon)
            q = (1 - eta) * Q[s_cur][a_cur] + eta * (R[s_cur][a_cur] + gamma * Q[s_next][a_next])
            Q[s_cur][a_cur] = q
            s_cur = s_next
    return np.argmax(Q, axis = 1)
```

图 12.17　Sarsa 算法

Q 学习算法

Q_learning (S, A, T, R, N, γ, ϵ, η):

For all $s \in S, a \in A$: $\tilde{Q}(s,a) = 0$

For $i = 1,2,\ldots,N$:

s_{cur} = initial state in S

While $s_{\text{cur}} \notin \{\text{terminal states of } S\}$:

a_{cur} = Epsilon_greedy($\tilde{Q}$, s_{cur}, A, ϵ)

$s_{\text{next}} = T(s_{\text{cur}}, a_{\text{cur}})$

$a_{\text{next}} = \text{argmax}_{a \in A} \tilde{Q}(s_{\text{next}}, a)$

$\tilde{Q}(s_{\text{cur}}, a_{\text{cur}}) \leftarrow (1-\eta)\tilde{Q}(s_{\text{cur}}, a_{\text{cur}}) + \eta\left(R(s_{\text{cur}}, a_{\text{cur}}) + \gamma\tilde{Q}(s_{\text{next}}, a_{\text{next}})\right)$

$s_{\text{cur}} = s_{\text{next}}$

For all $s \in S$:

$\pi(s) = \text{argmax}_{a \in A} \tilde{Q}(s,a)$

Return π

图 12.18 Q 学习算法描述

machine_learning.reinforcement_learning.q_learning

```
import numpy as np
from machine_learning.reinforcement_learning.epsilon_greedy import epsilon_greedy

def q_learning(env, N, gamma, epsilon, eta):
    S, A, T, R = env.S, env.A, env.T, env.R
    n_states, n_actions = len(S), len(A)
    Q = np.zeros((n_states, n_actions))
    for iter in range(N):
        env.reset()
        s_cur = env.s_start
        while s_cur not in env.S_end:
            a_cur = epsilon_greedy(Q[s_cur], n_actions, epsilon)
            s_next = int(T[s_cur][a_cur])
            a_next = np.argmax(Q[s_next])
            q = (1 - eta) * Q[s_cur][a_cur] + eta * (R[s_cur][a_cur] + gamma * Q[s_next][a_next])
            Q[s_cur][a_cur] = q
            s_cur = s_next
    return np.argmax(Q, axis = 1)
```

图 12.19　Q 学习算法

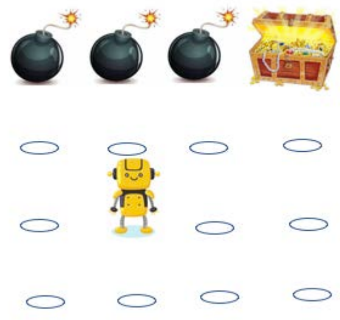


图 12.20　埋藏着宝藏的沼泽地

machine_learning.reinforcement_learning.landmine_and_treasure_2d

```python
import numpy as np
from machine_learning.reinforcement_learning.environment import Environment

class LandmineAndTreasure2d(Environment):
    def __init__(self, m, n):
            n_states = m * n
            n_actions = 4
            S = range(n_states)
            S_end = range(n)
            A = range(n_actions)
            R = np.zeros((n_states, n_actions))
            for j in range(n-1):
                    R[n + j][0] = -1000
            R[2 * n - 1][0] = 100
            T = np.zeros((n_states, n_actions))
            for i in range(m):
                    for j in range(n):
                            s = n * i + j
                            T[s][0] = s - n if i > 0 else s
                            T[s][1] = s + 1 if j < n - 1 else s
                            T[s][2] = s + n if i < m - 1 else s
                            T[s][3] = s - 1 if j > 0 else s
            for s in S_end:
                    for a in A:
                            T[s][a] = s
            self.S, self.A, self.T, self.R, self.S_end = S, A, T, R, S_end
```

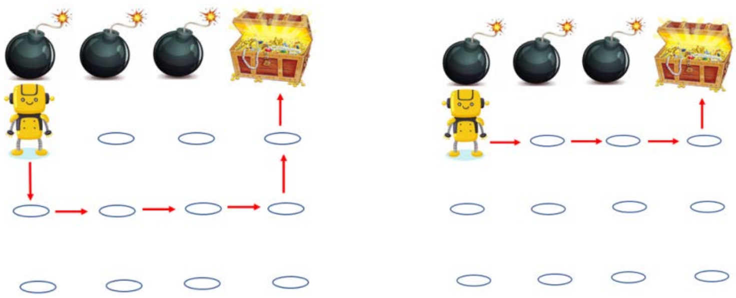
图 12.21　地雷与宝藏游戏的环境模型

```python
from machine_learning.reinforcement_learning.td.sarsa import sarsa
from machine_learning.reinforcement_learning.td.q_learning import q_learning
import machine_learning.reinforcement_learning.landmine_and_treasure_2d as game

m = 4
n = 4
env = game.LandmineAndTreasure2d(m, n)
pi_sarsa = sarsa(env, 1000, 0.95, 0.2, 0.1)
pi_q = q_learning(env, 1000, 0.95, 0.2, 0.1)

map = {0: "U", 1: "R", 2: "D", 3: "L"}
for i in range(m):
    print([map[pi_sarsa[s]] for s in range(i * n, (i + 1) * n)])
for i in range(m):
    print([map[pi_q[s]] for s in range(i * n, (i + 1) * n)])
```

图 12.22　Sarsa 算法和 Q 学习算法求解寻宝最优策略



(a)　Sarsa 算法　　　　　(b)　Q 学习算法

图 12.23　两个算法的寻宝路线图

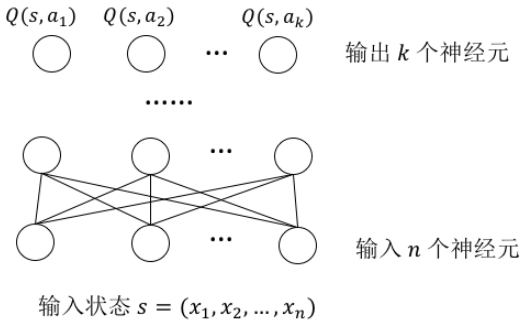


图 12.24　DQN 模型

DQN 算法

Deep_Q_Network (S, A, T, R, N, γ, ϵ):

    $DQN =$ initial DQN model

    For $i = 1,2,\dots,N$:

        $s_{\mathrm{cur}} =$ initial state of S

        While $s_{\mathrm{cur}} \notin$ {terminal states of S} :

            a_{cur} = Epsilon_greedy(DQN, s_{cur}, A, ϵ)

            $s_{\mathrm{next}} = T(s_{\mathrm{cur}}, a_{\mathrm{cur}})$

            $a_{\mathrm{next}} = \mathrm{argmax}_{a\in A} DQN(s_{\mathrm{next}}, a)$

            $\mathrm{loss} = \left(R(s_{\mathrm{cur}}, a_{\mathrm{cur}}) + \gamma DQN(\mathrm{s}_{\mathrm{next}}, a_{\mathrm{next}}) - DQN(s_{\mathrm{cur}}, a_{\mathrm{cur}})\right)^2$

            BackProp(DQN, loss)

            $s_{\mathrm{cur}} = s_{\mathrm{next}}$

    For all $s \in S$:

        $\pi(s) = \mathrm{argmax}_{a\in A} DQN(s, a)$

Return π

图 12.25　DQN 算法描述

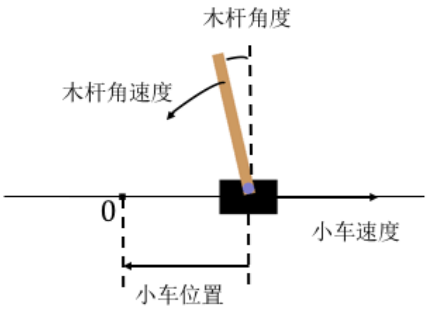


图 12.26 木杆平衡的环境

```python
import gym

env = gym.make("CartPole-v0")
state = env.reset()
action = 1
while True:
    state, reward, done, info = env.step(action)
    env.render(mode = "rgb_array")
    if done:
        break
```

图 12.27　OpenAI 中的木杆平衡虚拟环境

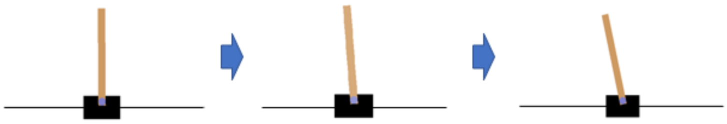


图 12.28　木杆状态变化过程

```python
import numpy as np
import gym
import tensorflow as tf
import machine_learning.reinforcement_learning.epsilon_greedy as eg

state_size = 4
n_hidden1 = 24
n_hidden2 = 24
n_actions = 2
State = tf.placeholder(tf.float32, shape=[None, state_size])
hidden1 = tf.layers.dense(State, n_hidden1, activation = tf.nn.relu)
hidden2 = tf.layers.dense(hidden1, n_hidden2, activation = tf.nn.relu)
Q_values = tf.layers.dense(hidden2, n_actions)

Target = tf.placeholder(tf.float32)
Action = tf.placeholder(tf.int32)
Q_value = tf.reduce_sum(Q_values * tf.one_hot(Action, n_actions))
loss = tf.reduce_mean(tf.square(Target - Q_value))
optimizer = tf.train.AdamOptimizer(learning_rate = 0.001)
training_op = optimizer.minimize(loss)

env = gym.make("CartPole-v0")
gamma = 0.95
n_iterations = 1000
epsilon_max = 1.0
epsilon_min = 0.01
epsilon_decay = 0.99
stop_penalty = -100
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    epsilon = epsilon_max
    for iteration in range(n_iterations):
        state = env.reset()
        done = False
        steps = 0
        while not done:
            steps += 1
            s_cur = state.reshape(1, state_size)
```

```
            Q_s_cur = Q_values.eval(feed_dict = {State: s_cur})
            a_cur = eg.epsilon_greedy(Q_s_cur, n_actions, epsilon)
            if epsilon > epsilon_min:
                epsilon *= epsilon_decay
            state, reward, done, info = env.step(action)
            s_next = state.reshape(1, state_size)
            if done:
                target = stop_penalty
                feed_dict = {State: s_cur, Action: a_cur, Target: target}
                sess.run(training_op, feed_dict = feed_dict)
                print(steps)
                    break
            else:
                Q_s_next = Q_values.eval(feed_dict = {State: s_next})
                target = reward + gamma * np.max(Q_s_next, axis=1)
                feed_dict = {State: s_cur, Action: a_cur, Target: target}
                sess.run(training_op, feed_dict = feed_dict)
env.close()
```

图 12.29 木杆平衡环境的 DQN 算法

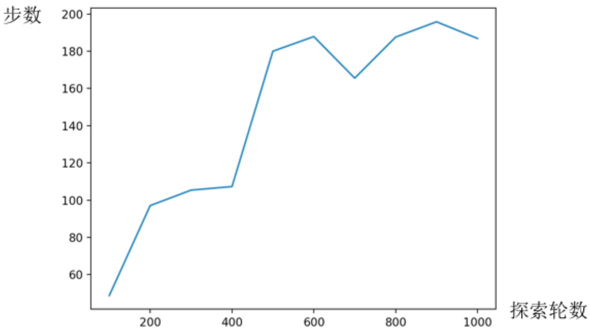


图 12.30　DQN 算法环境探索轮数与木杆平衡步数的关系

REINFORCE 算法

REINFORCE (S, A, T, R, N, γ, η):

$\boldsymbol{W}$ = random initial model parameters

For $i = 1,2,\ldots,N$:

$\boldsymbol{s}_{\mathrm{cur}}$ = initial state in S

$t = 0$

While $\boldsymbol{s}_{\mathrm{cur}} \notin$ {terminal states of S} :

$a_{\mathrm{cur}} \sim h_{\boldsymbol{W}}(\boldsymbol{s}_{cur})$

$\boldsymbol{s}_{\mathrm{next}} = T(\boldsymbol{s}_{\mathrm{cur}}, a_{\mathrm{cur}})$

$\boldsymbol{W} \leftarrow \boldsymbol{W} + \eta \cdot \gamma^{t} \cdot R(\boldsymbol{s}_{\mathrm{cur}}, a_{\mathrm{cur}}) \nabla \log h_{\boldsymbol{W}}(\boldsymbol{s}_{\mathrm{cur}}, a_{\mathrm{cur}})$

$\boldsymbol{s}_{\mathrm{cur}} = \boldsymbol{s}_{\mathrm{next}}$

$t \leftarrow t + 1$

Return $h_{\boldsymbol{W}}$

图 12.31　REINFORCE 算法描述

```python
import numpy as np
import gym

def softmax(scores):
    e = np.exp(scores)
    s = e.sum()
    return e / s

def REINFORCE(env, state_size, n_actions, n_iter, gamma, eta):
    W = np.random.rand(state_size, n_actions)
    for iter in range(n_iter):
        state = env.reset()
        done = False
        discount = 1
        steps = 0
        while not done:
            steps += 1
            s = state.reshape(1, state_size)
            probs = softmax(s.dot(W))
            a = np.random.choice(n_actions, p = probs.reshape(-1))
            state, reward, done, info = env.step(a)
            y = np.zeros(n_actions)
            y[a] = 1
            y = y.reshape(1, n_actions)
            gradient = -s.T.dot(probs - y)
            if done:
                reward = -100
                print("iteration {} lasts for {} steps".format(iter, steps))
            W = W + eta * discount * reward * gradient
            discount *= gamma

env = gym.make("CartPole-v0")
REINFORCE(env, 4, 2, 3000, 0.95, 0.1)
```

图 12.32 木杆平衡问题的 REINFORCE 算法

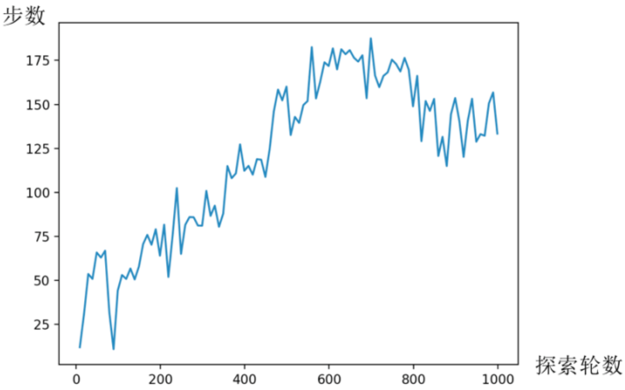


图 12.33　REINFORCE 算法探索轮数与木杆平衡步数的关系

Actor-Critic 算法

Actor_Critic (S, A, T, R, N, γ, $\eta_{\boldsymbol{u}}, \eta_{\boldsymbol{W}}$):

$\boldsymbol{W}, \boldsymbol{u}$ = random initial model parameters

For $i = 1,2,\ldots,N$:

$\boldsymbol{s}_{\text{cur}}$ = initial state in S

$t = 0$

While $\boldsymbol{s}_{\text{cur}} \notin$ {terminal states of S} :

$a_{\text{cur}} \sim h_{\boldsymbol{W}}(\boldsymbol{s}_{\text{cur}})$

$\boldsymbol{s}_{\text{next}} = T(\boldsymbol{s}_{\text{cur}}, a_{\text{cur}})$

$\delta = R(\boldsymbol{s}_{\text{cur}}, a_{\text{cur}}) + \gamma V_{\boldsymbol{u}}(\boldsymbol{s}_{\text{next}}) - V_{\boldsymbol{u}}(\boldsymbol{s}_{\text{cur}})$

$\boldsymbol{W} \leftarrow \boldsymbol{W} + \eta_{\boldsymbol{W}} \cdot \gamma^t \cdot \delta \cdot \nabla \log h_{\boldsymbol{W}}(\boldsymbol{s}_{\text{cur}}, a_{\text{cur}})$

$\boldsymbol{u} \leftarrow \boldsymbol{u} + \eta_{\boldsymbol{u}} \cdot \gamma^t \cdot \delta \cdot \nabla V_{\boldsymbol{u}}(\boldsymbol{s}_{\text{cur}})$

$\boldsymbol{s}_{\text{cur}} = \boldsymbol{s}_{\text{next}}$

$t \leftarrow t + 1$

图 12.34　Actor-Critic 算法描述

```python
import numpy as np
import gym

def softmax(scores):
    e = np.exp(scores)
    s = e.sum()
    return e / s

def actor_critic(env, state_size, n_actions, n_iter, gamma, eta_u, eta_W):
    W = np.random.rand(state_size, n_actions)
    u = np.random.rand(state_size, 1)
    for iter in range(n_iter):
        state = env.reset()
        done = False
        discount = 1
        steps = 0
        while not done:
            steps += 1
            s_cur = state.reshape(1, state_size)
            probs = softmax(s_cur.dot(W))
            a_cur = np.random.choice(n_actions, p = probs.reshape(-1))
            state, reward, done, info = env.step(a_cur)
            if done:
                print("iteration {} lasts for {} steps".format(iter, steps))
                reward = -100
                delta = reward - s_cur.dot(u)
            else:
                s_next = state.reshape(1, state_size)
                delta = reward + gamma * s_next.dot(u) - s_cur.dot(u)
            y = np.zeros(n_actions)
            y[a_cur] = 1
            y = y.reshape(1, n_actions)
            gradient_W = s_cur.T.dot(probs - y)
            W = W - eta_W * discount * delta * gradient_W
            gradient_u = s_cur.T
            u = u + eta_u * discount * delta * gradient_u
            discount *= gamma

env = gym.make("CartPole-v0")
actor_critic(env, 4, 2, 1000, 0.95, 0.1, 0.1)
```

图 12.35 木杆平衡问题的 Actor-Critic 算法

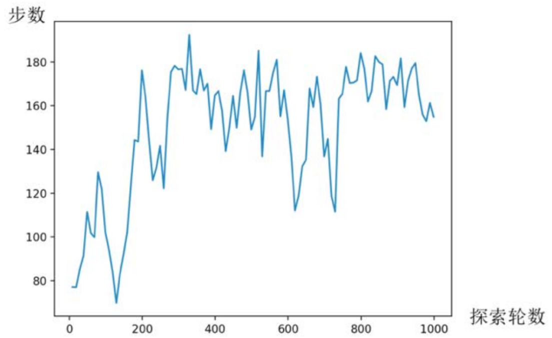


图 12.36　Actor-Critic 算法探索轮数与木杆平衡步数的关系