

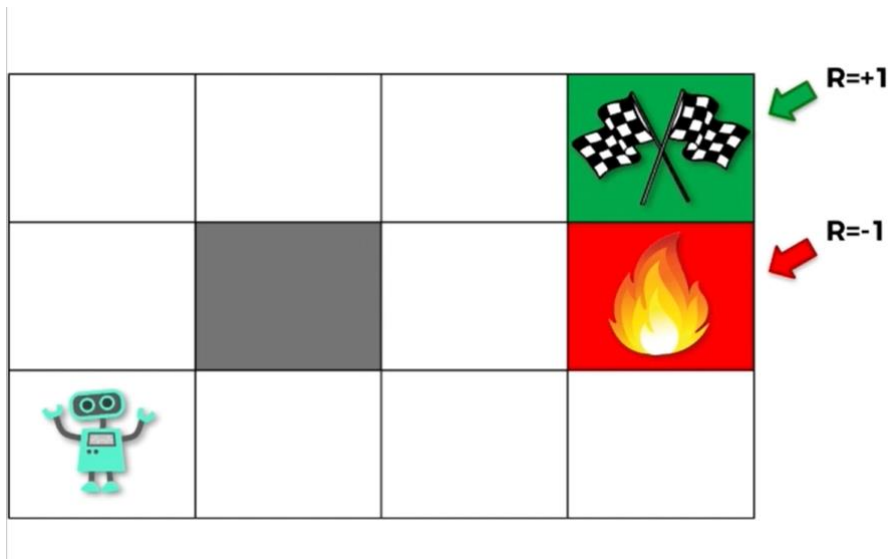
Deep Reinforcement Learning

1. Fundamentals of Reinforcement Learning

Process: Agent 在当前时刻 t 位于状态 s_t , 作出一个动作 a_t , 到达一个新的状态 s_{t+1} , 并获得奖赏 $R_{(s_t, a_t)}$, 如果游戏还没有结束, 可执行下一个动作 a' , 以此类推。

1.1 State Value Function, V 函数方法

通过比较 V 值, 来选择接下来要进行的动作。





对于这个环境, 我们如何指导 agent 到达终点

Bellman equation:

$$V(s) = \max_a (R(s, a) + \gamma V(s'))$$

γ 可认为是折扣因子 discount, 属于(0,1)。

V=0.81	V=0.9	V=1		<div>$\gamma = 0.9$</div>
V=0.73		V=0.9		
V=0.66	V=0.73	V=0.81	V=0.73	

缺点: 未能考虑掉入火坑的风险, 未考虑随机性。

改进：引入 Markov Decision Process (MDP) 的 Bellman equation

$$V(s) = \max_a (R(s, a) + \gamma \sum_{s'} P(s, a, s') V(s'))$$

下一个动作是 s' 的概率仅与 s, a 有关——马尔可夫性质。

比如，Agent 希望向上走，但是由于外界因素，实际有10%的概率往左或往右走。

V=0.71	V=0.74	V=0.86	
V=0.63		V=0.39	
V=0.55	V=0.46	V=0.36	V=0.22

0.9 变成了 0.39

若初始位置在 0.39，agent 宁可选择绕长路，以避免掉入火坑的危险。

Living penalty: 除了终点的 1 与 -1，在每个位置增加 R ， $R < 0$ ，以督促 agent 走更短的路径。

1.2 State-Action Value Function，Q 函数方法

$$V(s) = \max_a (R(s, a) + \gamma \sum_{s'} P(s, a, s') V(s'))$$

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} P(s, a, s') V(s')$$

可见

$$V(s) = \max_a Q(s, a)$$

V : 状态的质量

Q : 动作的质量

易得迭代方程:

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} P(s, a, s') \max_a Q(s', a')$$

一种更新 Q-value 的方法:

Temporal difference 时序差分

在实际中, 很难一次性算得某个动作的 Q-value, 比如有些 Q-value 的计算需要用到其他 Q-value。因此需要用新信息更新旧结果。

为简单起见, 这里先不考虑马尔可夫性质。

$$Q(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a')$$

旧结果: $Q(s, a)$

新结果: $R(s, a) + \gamma \max_{a'} Q(s', a')$

$$TD(a, s) = R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

用新样本改进先验的思想

$$Q_t(s, a) = Q_{t-1}(s, a) + \alpha TD(a, s)$$

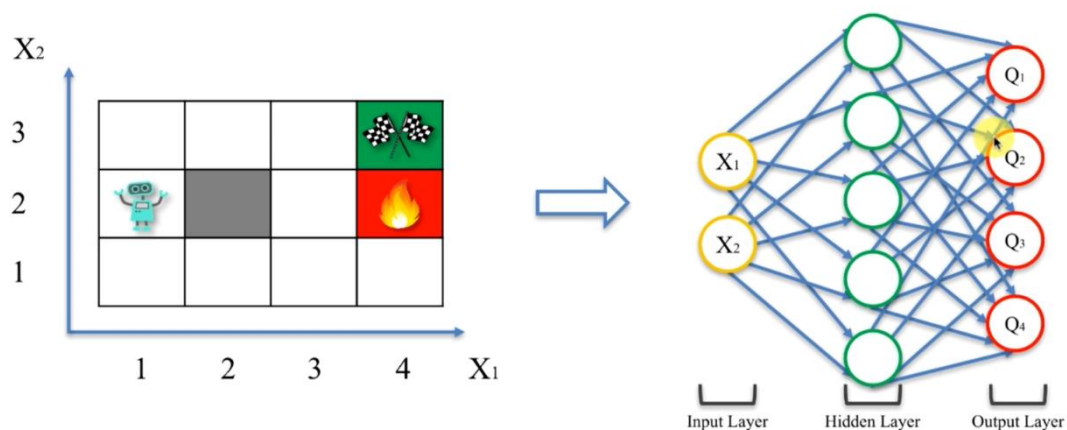
消去 $TD(a, s)$

$$Q_t(s, a) = Q_{t-1}(s, a) + \alpha (R(s, a) + \gamma \max_{a'} Q(s', a') - Q_{t-1}(s, a))$$

$$0 < \alpha < 1$$

1.3 Deep Q-learning

Intuition:



把每个方格用坐标表示, 即可引入神经网络

x_1, x_2 : state

Simple Q-learning 与 Deep Q-learning 的对比

Simple Q-learning:

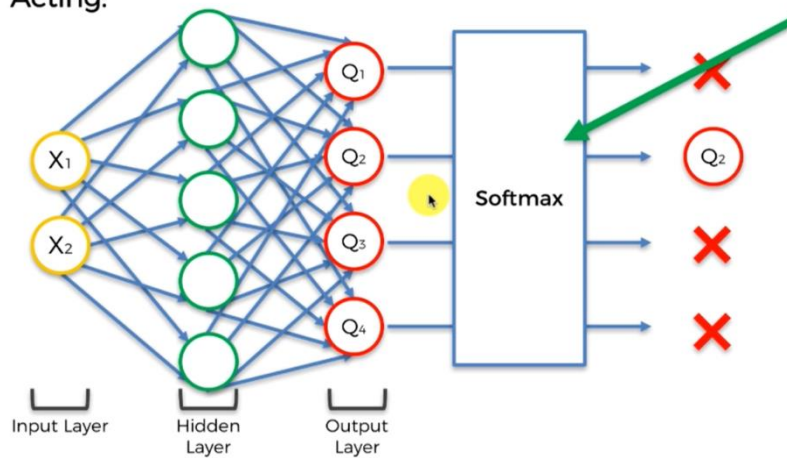
$$TD(a, s) = R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

Deep Q-learning

$$L = \sum (Q_{Target} - Q_{pred})^2$$

Action Selection Policies

Acting:

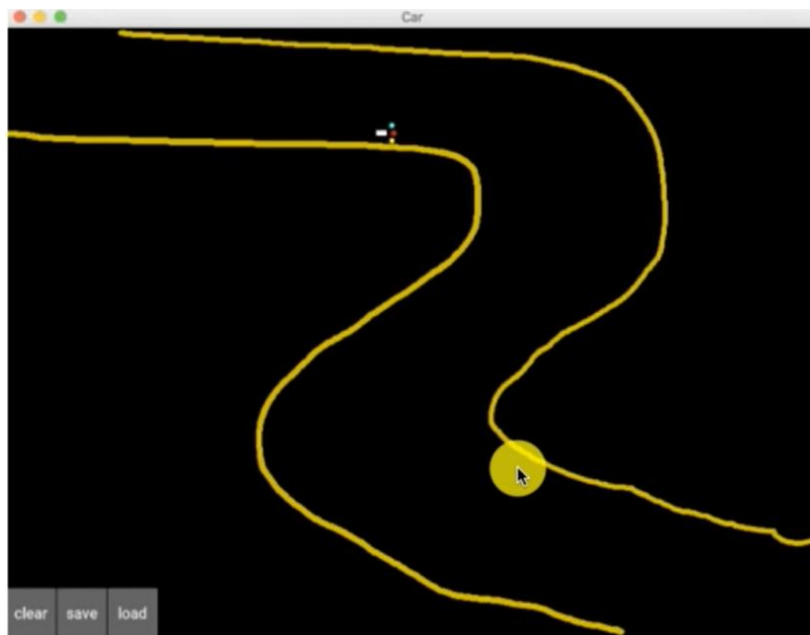


Action Selection:

- ϵ -greedy
- ϵ -soft ($1-\epsilon$)
- Softmax

Exploration
vs
Exploitation

Experience Replay



储存足够多的 experience, $\{(s_1, s'_1, a_1, r_1), (s_2, s'_2, a_2, r_2) \dots\}$, 再按照均匀分布抽样, 训练。

Trick1: 连续的状态通常是相关的, 会给神经网络带来偏差 bias。

Trick2: Experience Replay 有利于让稀有经验多次被学习。

2. Q-learning

Initialization (First iteration):

For all couples of states s and actions a , the Q-values are initialized to 0.

Next iterations:

At each iteration $t \geq 1$, we repeat the following steps:

1. We select a random state s_t from the possible states.
2. From that state, we play a random action a_t .
3. We reach the next state s_{t+1} and we get the reward $R(s_t, a_t)$.
4. We compute the Temporal Difference $TD_t(s_t, a_t)$:

$$TD_t(s_t, a_t) = R(s_t, a_t) + \gamma \max_a (Q(s_{t+1}, a)) - Q(s_t, a_t)$$

5. We update the Q-value by applying the Bellman equation:

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha TD_t(s_t, a_t)$$

训练完后，agent 选择当前状态下获得最高 Q-value 的动作。

3. Deep Q-learning (DQN)

Initialization:

1. The memory of the Experience Replay is initialized to an empty list M.
2. We choose a maximum size of the memory.

At each time t, we repeat the following process, until the end of the epoch:

1. We predict the Q-values of the current state s_t .
2. We play the action that has the highest Q-value: $a_t = \operatorname{argmax}_a \{Q(s_t, a)\}$
3. We get the reward $R(s_t, a_t)$.
4. We reach the next state s_{t+1} .
5. We append the transition (s_t, a_t, r_t, s_{t+1}) in the memory M.
6. We take a random batch $B \subset M$ of transitions. For all the transitions $(s_{t_B}, a_{t_B}, r_{t_B}, s_{t_B+1})$ of the random batch B:

We get the predictions: $Q(s_{t_B}, a_{t_B})$

We get the targets: $R(s_{t_B}, a_{t_B}) + \gamma \max_a (Q(s_{t_B+1}, a))$

We compute the loss between the predictions and the targets over the whole batch B:

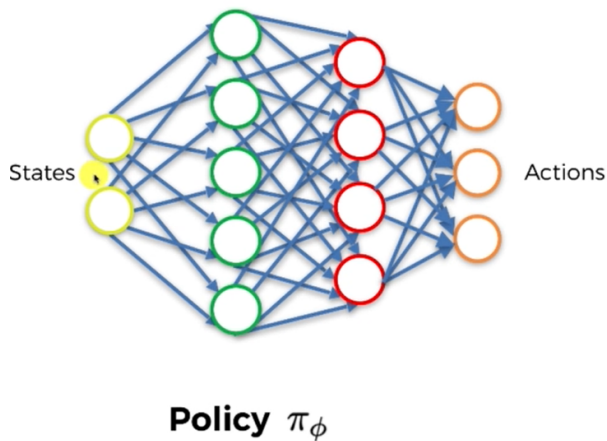
$$\text{Loss} = \frac{1}{2} \sum_B \left(R(s_{t_B}, a_{t_B}) + \gamma \max_a (Q(s_{t_B+1}, a)) - Q(s_{t_B}, a_{t_B}) \right)^2 = \frac{1}{2} \sum_B TD_{t_B}(s_{t_B}, a_{t_B})^2$$

We backpropagate this loss error back into the neural network, and through stochastic gradient descent we update the weights according to how much they contributed to the loss error.

缺点: $\gamma \max_a Q(s_{t_B+1}, a)$ 使得 DQN 难以处理连续的动作。

4. Actor-Critic (AC) 演员-评论家模型

4.1 Policy Gradient



Return: $R_t = \sum_{i=t}^T \gamma^{i-t} r(s_i, a_i)$

Goal: Maximize the expected return $J(\phi) = \mathbb{E}_{s_i \sim p_\pi, a_i \sim \pi} [R_0]$

Policy Gradient:

- We compute the gradient of the expected return with respect to the parameters ϕ

$$\nabla_\phi J(\phi)$$

- We update the parameters through gradient ascent:

$$\phi_{t+1} = \phi_t + \alpha \nabla_\phi J(\pi_\phi)|_{\phi_t}$$

容易证明，通过梯度上升，神经网络将输出能够获得更大总回报的 action。

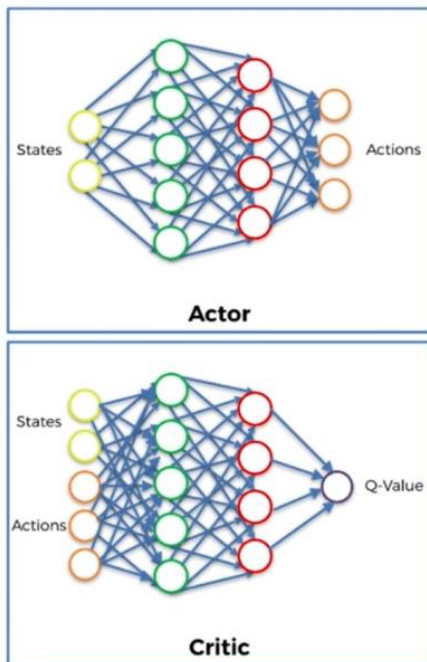
$$J(\pi_\phi) = \int \pi_\phi(\tau) R(\tau) d\tau$$

$$\frac{dJ(\pi_\phi)}{d\phi} = E_{\tau \sim \pi_\phi}(\tau) \left[\frac{\partial \log \pi_\phi(\tau)}{\partial \phi} R(\tau) \right]$$

其中 τ 是一个完整的轨迹，即 $s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_T$ 。 $R(\tau)$ 是一次轨迹 τ 的累计奖励。

可见当 $R(\tau) > 0$ 时， $\frac{dJ(\pi_\phi)}{d\phi}$ 与 $\frac{\partial \log \pi_\phi(\tau)}{\partial \phi}$ 同向， J 变大则 $\log \pi_\phi(\tau)$ 变大，即鼓励产生更多这样的 τ ，即证。

4.2 Actor-Critic (AC)



Return: $R_t = \sum_{i=t}^T \gamma^{i-t} r(s_i, a_i)$

Goal: Maximize the expected return $J(\phi) = \mathbb{E}_{s_i \sim p_\pi, a_i \sim \pi} [R_0]$

Deterministic Policy Gradient:

- In actor-critic methods, the policy, known as the actor, can be updated through the deterministic policy gradient algorithm:

$$\nabla_\phi J(\phi) = \mathbb{E}_{s \sim p_\pi} [\nabla_a Q^\pi(s, a)|_{a=\pi(s)} \nabla_\phi \pi_\phi(s)]$$

- We update the policy parameters through gradient ascent:

$$\phi_{t+1} = \phi_t + \alpha \nabla_\phi J(\pi_\phi)|_{\phi_t}$$

5. Twin Delayed DDPG (TD3)

解决了连续动作空间的问题

Initialization:

- Step 1: We initialize the Experience Replay memory, with a size of 20000. We will populate it with each new transition.
 Step 2: We build one neural network for the Actor model and one neural network for the Actor target.
 Step 3: We build two neural networks for the two Critic models and two neural networks for the two Critic targets.

Training Process - We run a full episode with first 10,000 actions played randomly, and then with actions played by the Actor model. Then we repeat the following steps:

- Step 4: We sample a batch of transitions (s, s', a, r) from the memory. Then for each element of the batch:
 Step 5: From the next state s' , the Actor target plays the next action a' .
 Step 6: We add Gaussian noise to this next action a' and we clamp it in a range of values supported by the environment.
 Step 7: The two Critic targets take each the couple (s', a') as input and return two Q-values $Q_{t1}(s', a')$ and $Q_{t2}(s', a')$ as outputs.
 Step 8: We keep the minimum of these two Q-values: $\min(Q_{t1}, Q_{t2})$. It represents the approximated value of the next state.
 Step 9: We get the final target of the two Critic models, which is: $Q_t = r + \gamma * \min(Q_{t1}, Q_{t2})$, where γ is the discount factor.
 Step 10: The two Critic models take each the couple (s, a) as input and return two Q-values $Q_1(s, a)$ and $Q_2(s, a)$ as outputs.
 Step 11: We compute the loss coming from the two Critic models: Critic Loss = $\text{MSE_Loss}(Q_1(s, a), Q_t) + \text{MSE_Loss}(Q_2(s, a), Q_t)$.
 Step 12: We backpropagate this Critic loss and update the parameters of the two Critic models with a SGD optimizer.
 Step 13: Once every two iterations, we update our Actor model by performing gradient ascent on the output of the first Critic model: $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$, where ϕ and θ_1 are resp. the weights of the Actor and the Critic.
 Step 14: Still once every two iterations, we update the weights of the Actor target by polyak averaging: $\theta'_i \leftarrow \tau \theta_i + (1 - \tau) \theta'_i$.
 Step 15: Still once every two iterations, we update the weights of the Critic target by polyak averaging: $\phi' \leftarrow \tau \phi + (1 - \tau) \phi'$.

