4 - Introduction to Reinforcement Learning

Giovanni Della Lunga
giovanni.dellalunga@unibo.it

Advanced Machine Learning for Finance

Bologna - April-May, 2022

Basic Ideas

Reinforcement Learning

- Some situations by their nature involve a series of decisions rather than a single one.
- Furthermore, as the decisions are taken, the environment may be changing.
- It is then necessary to determine the best action bearing in mind that further decisions have to be taken later
- Reinforcement learning is the branch of machine learning that deals with this type of sequential decision making .
- The algorithm receives rewards when outcomes are good and incurs costs (negative rewards) when they are bad.
- The objective of the algorithm is to maximize expected future rewards possibly with discounting.

Reinforcement Learning

There are three basic concepts in reinforcement learning: **state**, **action**, and **reward**

- The **STATE** describes the current situation.
  - For a robot that is learning to walk, the state could be the position in space.
  - For a Go program, the state is the positions of all the pieces on the board.

Reinforcement Learning

There are three basic concepts in reinforcement learning: **state**, **action**, and **reward**

- **ACTION** is what an agent can do in each state.
  - Given the state, or positions, a robot can take steps within a certain distance.
  - There are typically finite (or a fixed range of) actions an agent can take.
  - For example, the robot can only move, say, 0.01 meter to 1 meter.
  - The Go program can only put down its piece in one of 19 x 19 (that is 361) positions.

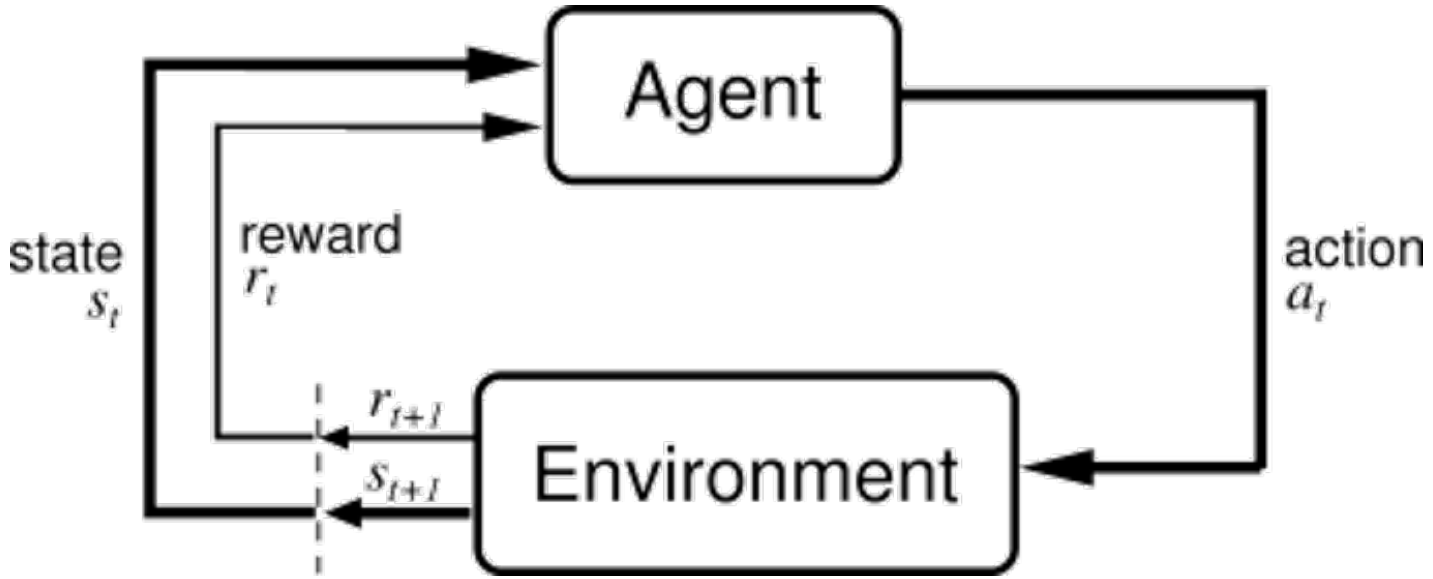
Reinforcement Learning

REWARD

- The reward signal defines the goal
- On each time step, the environment sends a single number called the reward to the reinforcement learning agent
- The agent objective is to maximize the total reward that it receives over the long run

Reinforcement Learning

- Basic idea:
- Receive feedback in the form of rewards
- Agent utility is defined by the reward function
- Must (learn to) act so as to maximize expected rewards

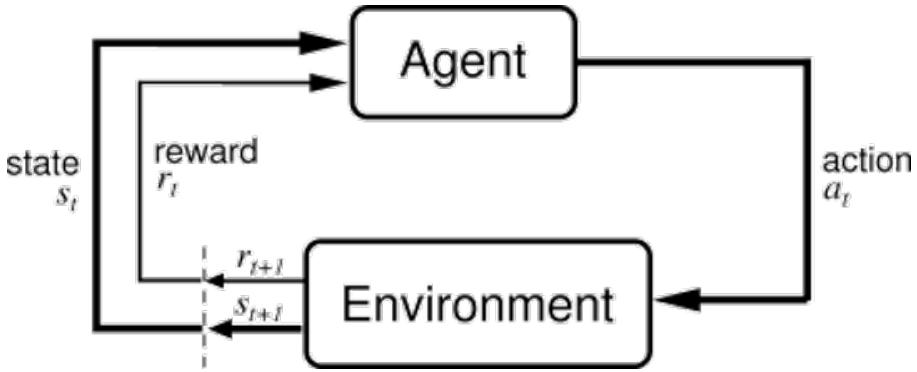


Reinforcement Learning

- **Supervised Learning**:
- Goal: $f(x) = y$
- Data: $\{(x_1, y_1); \ldots; (x_n, y_n)\}$
- **Reinforcement Learning**:
- Goal: find the sequence of actions that gives the best reward

  $$\text{Maximize} \sum_{i=1}^{\infty} \text{Reward}(\text{State}_i, \text{Action}_i)$$

- Data: Reward_i,State_{i+1},State_i, Action_i

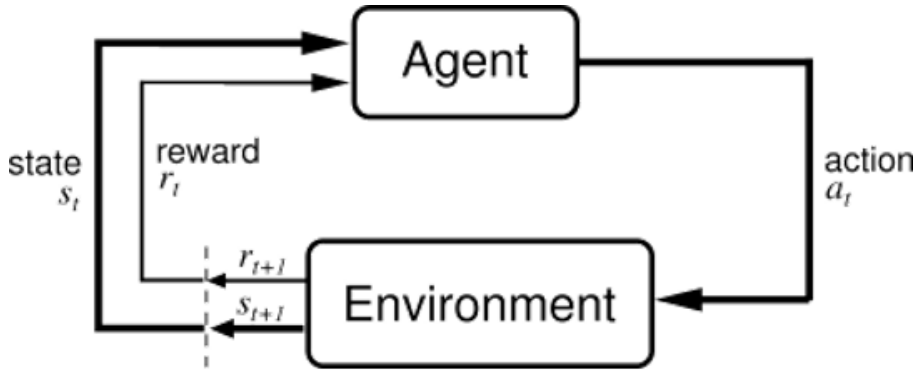


Reinforcement Learning

- **Supervised Learning**:
- Goal: $f(x) = y$
- Data: $\{(x_1, y_1); \ldots; (x_n, y_n)\}$
- **Reinforcement Learning**:
- Goal: find the sequence of actions that gives the best reward

$$\text{Maximize} \sum_{i=1}^{\infty} \text{Reward}(\text{State}_i, \text{Action}_i)$$

- Data: Reward_i,State_{i+1},State_i, Action_i



By trying different actions, and accumulationg the rewards, the agent can find the best action for each state. In this way, with the reinforcing reward values, the optimal policy is learned from repeated interaction with the environment and the problem is "solved"

Policy or Strategy

- A policy is a mapping from the perceived states of the environment to actions to be taken when in those states
- A reinforcement learning agent uses a policy to select actions given the current environment state

Policy or Strategy

- The policy answer the question how the different actions a at state s should be chosen
- The policy could be deterministic or stochastic

The Multi-armed Bandit Problem

A Simple example: K-armed bandits

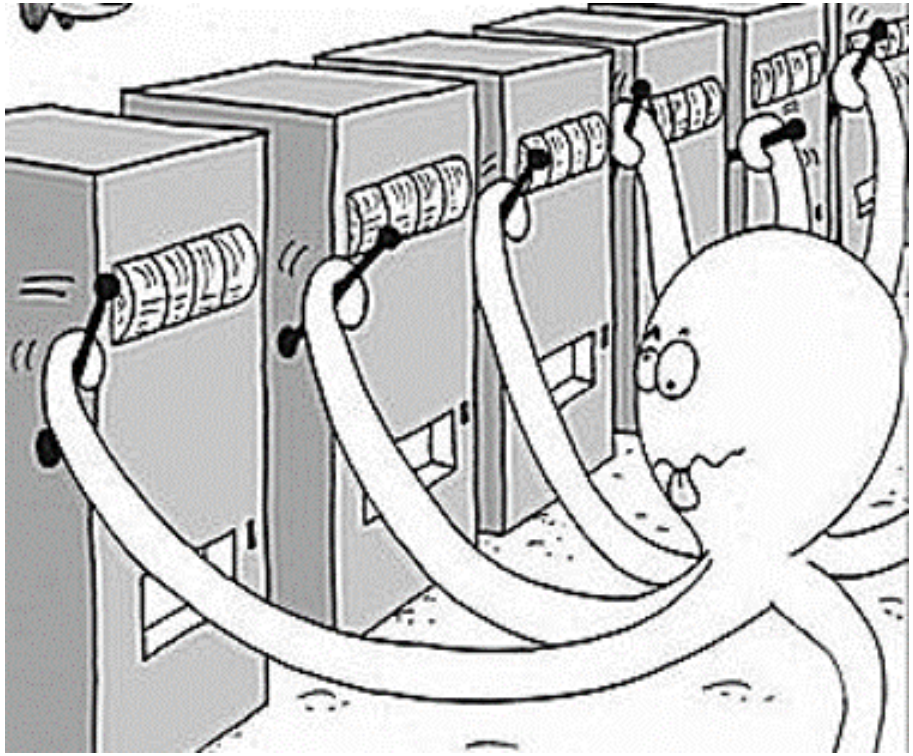
- This is like a one-armed bandit except that you have to choose between K levers.
- Lever k provides a return from a normal distribution with mean m_k and standard deviation 1
- Objective is to maximize return over a large number of trials

A Simple example: K-armed bandits

- One single state, the problem is a state-less one;
- Stochastic Reward
- Action depends only on strategy and not on state;
- Exploitation Vs Exploration

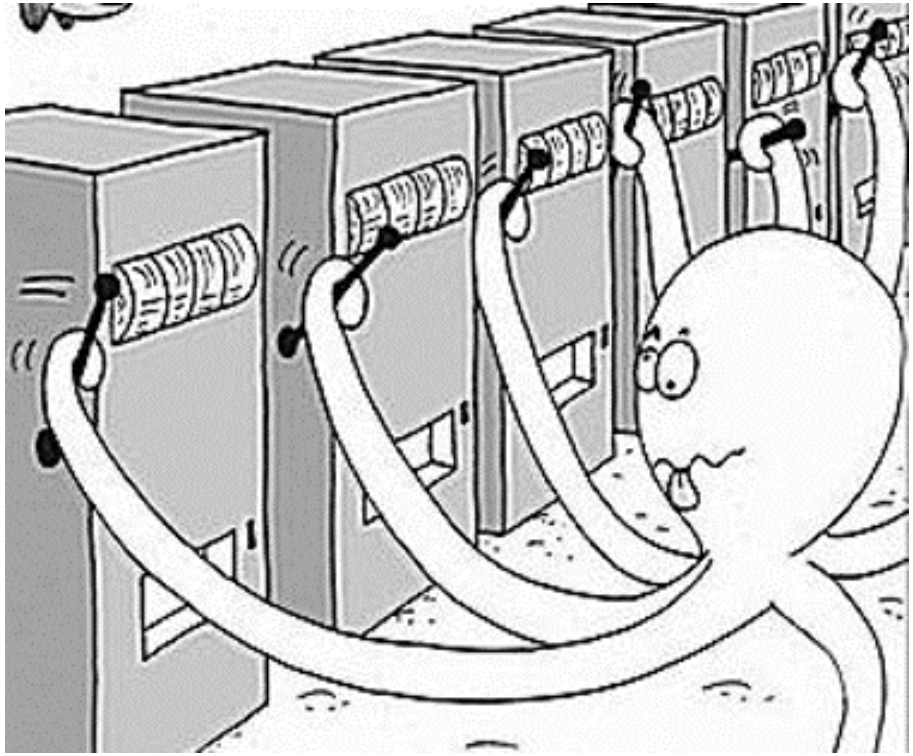
Strategy

- You keep records of the average return from choosing each lever
- At each turn you have to decide between
- Choose the lever that has given you the best average return so far (the *greedy action*)
- Try out a new action



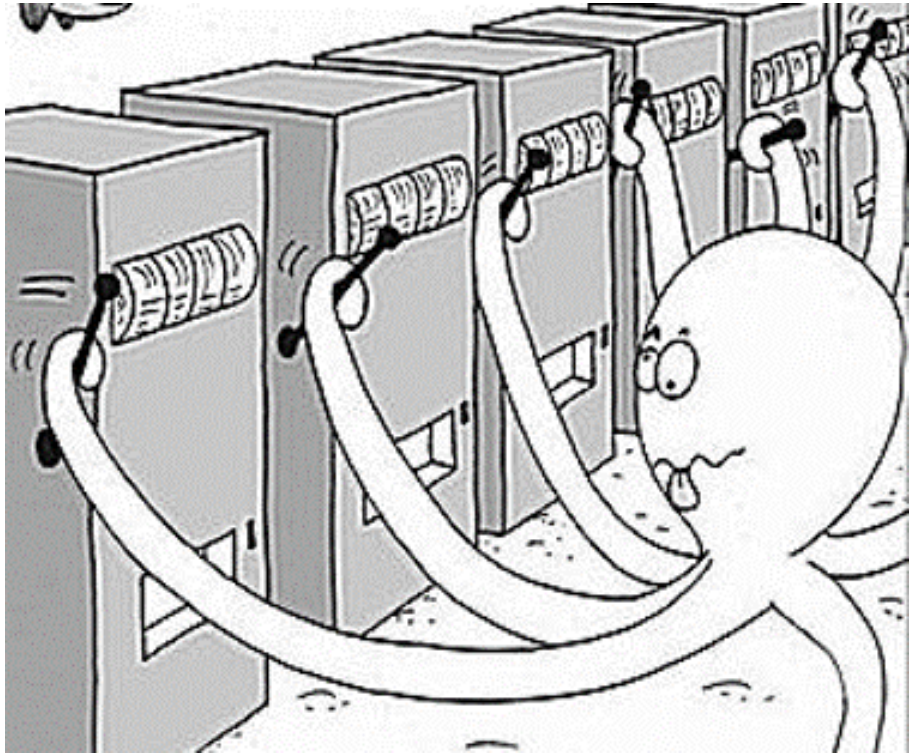
Strategy

- The first choice is **exploitation**; the second is **exploration**.
- Exploitation maximizes the immediate expected return but exploration may do better in the long run



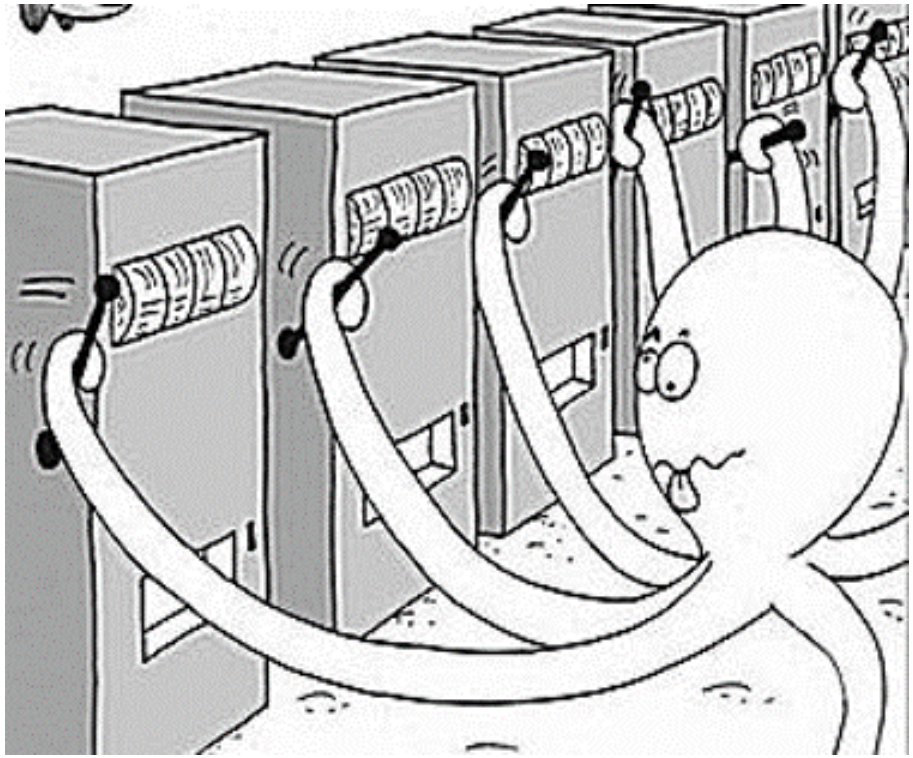
Strategy

- A strategy for the gambler is to:
- Randomly choose a lever with probability ϵ
- Choose the lever that has given the best average payoff so far with probability $1 - \epsilon$



Strategy

- We can implement this strategy by sampling a random number between 0 and 1. If it is less than ϵ the lever is chosen randomly; otherwise, the lever with the best average payoff so far is chosen.
- We might choose a value of ϵ equal to one initially and then slowly reduce it to zero as data on the payoffs is obtained



The Math

- Suppose that lever k has been chosen $n-1$ times and the total reward on the jth time it is chosen is R_j
- Expected reward is

$$Q_k^{old} = \frac{1}{n-1} \sum_{j=1}^{n-1} R_j \tag{1}$$

- If kth lever is chosen for the nth time and produces a reward R_n

$$Q_k^{new} = \frac{1}{n} \sum_{i=1}^{n} R_i = Q_k^{old} + \frac{1}{n} \left(R_n - Q_k^{old} \right) \tag{2}$$

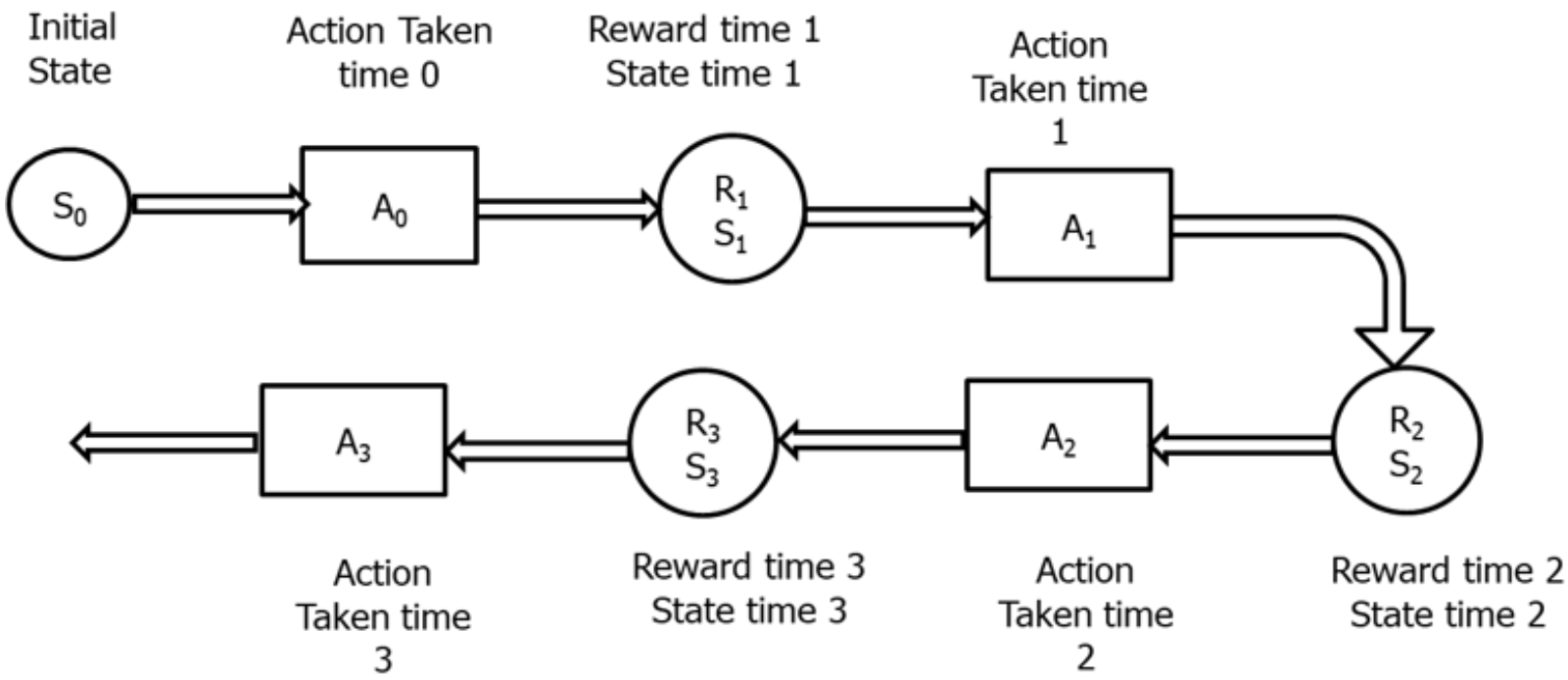
The exploration parameter ϵ and initial values

- It makes sense to reduce ϵ over time. For example we can let it decline exponentially
- The initial Q-values make a difference. For example if they all set equal to 2 instead of 0 there would be early exploration
- If the standard deviation of the payoff is increased, a higher value of ϵ would be appropriate

Changing Environment

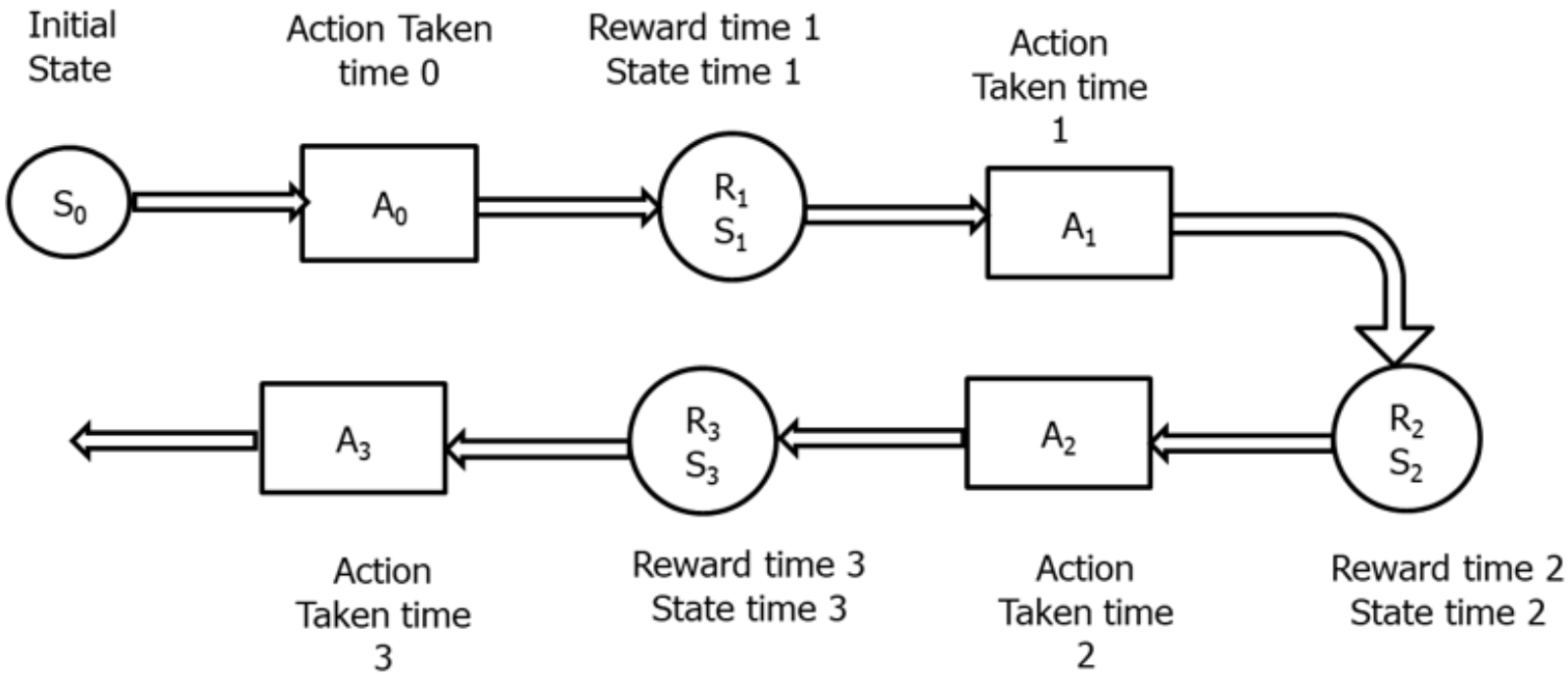
The More General Model: Actions and States

- The multi-armed bandit problem provides a simple example of reinforcement learning.
- **The environment does not change** and so the Q values are a function only of the action (i.e., the lever chosen).
- In a more general situation, there are a number of states and a number of possible actions and we are dealing with a **changing environment**;



The More General Model: Actions and States

- The decision maker takes an action, A_0, at time zero when the state S_0 is known. This results in a reward, R_1, at time 1 and a new state, S_1, is encountered.
- The decision maker then takes another action at time 1 which results in a reward, R_2, at time 2 and a new state, S_2; and so on.
- In this more general situation, the $Q - value$ **is a function of both the state and the action taken**.



The More General Model: Actions and States

Agent-Environment Interaction

- Set of states $\{S\}$
- Set of actions $\{A\}$
- In the general case we are facing with a changing environment. We assume that we can define state transition probabilities $p(s'|s, a)$. This is the probability distribution over the state space given we take action a in state s
- Reward function $R : S \times A \to \mathbb{R}$
- For simplicity, assume discrete rewards
- Finite Markov Decison Process if both S and A are finite

The More General Model: Actions and States

Agent-Environment Interaction

- At each time step, the agent implements a stochastic policy or mapping from states to probabilities of selecting each possible action
- The policy is denoted π_t where $\pi_t(a|s)$ is the probability of taking action a when in state s
- A policy is a stochastic rule by which the agent selects actions as a function of states
- Reinforcement learning methods specify how the agent changes its policy using its experience

The More General Model: Actions and States

State Transition Probabilities

- Suppose the reward function is discrete and maps from $S \times A \rightarrow W$
- The state transition probability or probability of transitioning to state s' given current state s and action a in that state is given by:

$$p\left(s'|s,a\right) = \sum_{r \in W} p\left(s', r|s, a\right) \tag{3}$$

MDP Dynamics

- Given current state s and action a in that state, the probability of the next state s' and the next reward r is given by:

$$p\left(s', r|s, a\right) = \Pr\left(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\right) \tag{4}$$

Value Function

- Objective could be to maximize the expected value of rewards

$$G = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_{t+T}$$

- In some cases, particularly when there is a long or infinite horizon, the objective is to maximize the **expected value of discounted rewards**:

$$G = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{T-t+1} \gamma^k R_{t+k+1} \qquad (5)$$

  where γ is a discount factor ≤ 1
- For each state we have to calculate an utility value equal to the sum of future discounted rewards.

Value Function

- The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting in that state
- Compute the value using the states that are likely to follow the current state and the rewards available in those states
- Use the values to make and evaluate decisions
- Action choices are made based on value judgements

Value Function

- Prefer actions that bring about states of **highest value instead of highest reward**
- Rewards are given directly by the environment
- Values must continually be re-estimated from the sequence of observations that an agent makes over its lifetime

The reward signal indicates what is good in the short run while the value function indicates what is good in the long run

State-Value Function

- Value functions are defined with respect to particular policies because future rewards depend on the agent actions
- Value functions give the expected return of a particular policy
- The value $v_\pi(s)$ of a state s under a policy π is the expected return when starting in state s and following the policy π from that state onwards

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} | S_t = s \right] \end{aligned} \tag{6}$$

- This is called also **state-value function**

Action-Value Function

- The value $q_\pi(s, a)$ of taking action a in state s under a policy π is defined as the expected return starting from s, taking the action a and thereafter following policy π
- $q_\pi(s, a)$ is the **action-value function** for policy π

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \end{aligned} \tag{7}$$

Bellman Equation

For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\
&= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right] \\
&= \mathbb{E}_\pi \left[R_{t+1} + \gamma \sum_{n=0}^{\infty} \gamma^n R_{t+n+2} | S_t = s \right] \\
&= \sum_a \pi(a, s) \sum_{s' r} p(s', r | s, a) \left[r + \gamma \mathbb{E}_\pi \left[\sum_{n=0}^{\infty} \gamma^n R_{t+n+2} | S_{t+1} = s' \right] \right] \\
&= \sum_a \pi(a, s) \sum_{s' r} p(s', r | s, a) \left[r + \gamma v_\pi(s') \right]
\end{aligned}
$$

Temporal difference learning

- So $r + \gamma v_\pi(s')$ is an unbiased estimate for $v_p i(s)$.
- This observation motivates the following algorithm for estimating $v_\pi(s)$
- The algorithm starts by initializing a table $V(s)$ arbitrarily, with one value for each state of the MDP. A positive learning rate α is chosen.
- We then repeatedly evaluate the policy π , obtain a reward r and update the value function for the old state using the rule:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right] \qquad (8)$$

Temporal difference learning

- How does an agent know what the desired long-term payoff should be?
- The secret lies in a Q-table (or Q function).
- It's a lookup table for rewards associated with every state-action pair.
- Each cell in this table records a value called a Q-value.
- It is a representation of the long-term reward an agent would receive when taking this action at this particular state, followed by taking the best path possible afterward.

Temporal difference learning

The procedure for Temporal difference learning is:

- In the beginning, the agent initializes Q-values to 0 for every state-action pair.
- More precisely, $Q(s, a) = 0$ for all states s and actions a.
- This is essentially saying we have no information on long-term reward for each state-action pair.
- After the agent starts learning, it takes an action a in state s and receives reward r. It also observes that the state has changed to a new state s'.

Temporal difference learning

The agent will update $Q(s, a)$ with the following formula:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}^{\text{temporal difference}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

The learning rate is a number between 0 and 1. The new long-term reward is the current reward, r, plus all future rewards in the next state, s', and later states, assuming this agent always takes its best actions in the future..

A Worked Example: Hedging Portfolio

RL Application: Hedging

- Suppose we do not know how to calculate delta
- Can reinforcement hedging find a good hedging strategy?

The set up

- We want to hedge the sale of 10 call options where $S = 100, K = 100, r = 0, q = 0, \sigma = .2$, and $T = 10$ days or 0.04 years (we assume 250 trading days in a year).
- The hedge is accomplished by owning $0, 1, 2, \ldots, 10$ shares (i.e., **there are 11 possible positions**).
- If the hedger knew that the option would be exercised because the final price of the stock was certain to be greater than 100, it would be optimal for her to buy 10 shares. Similarly, if the hedger knew that the option would be not be exercised because the final price of the stock was certain to be less than 100, it would be optimal for the hedger to hold no shares.
- In practice of course, the outcome is uncertain and, when each hedging decision is made, the hedger **must choose a position between 0 and 10 shares**.

The set up

Dynamics

- We assume that the behavior of the stock price is that the underlying process follows the Black-Scholes-Merton model.
- We assume that the hedge position (i.e., number of shares owned) can be changed once a day so that **a total of 10 decisions have to be made**;
- Both State and Reward are stochastic;

The set up

State

The states for the reinforcement learning algorithm at the time hedging decision is made on a day and are defined by:

- **Number of Shares Currently Held** (i.e., the position taken on the previous day)
- **The stock price**
- **Days to Maturity**

The set up

Actions

- As mentioned there are 11 possible values for the number of shares currently held.
- A total of 15 possible price states are considered:

$$\leq 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, \geq 107$$

- To simplify matters we assume that **at most five shares can be bought and five shares can be sold**.
- For example, if the current holding is 7, possible actions (i.e., changes in the position) are -5, -4, -3, -2, -1, 0, +1, +2, and +3. Similarly, if the current holding is 2, possible actions are -2, -1, 0, +1, +2, +3, +4, and +5.

The set up

Objective

- The objective function is to **minimize the variance of the hedging cost**. Since $r = 0$ this is equivalent to minimize

$$\sum_i H_i^2$$

- Hedging cost on day i is

$$H_i = N_i(S_{i+1} - S_i) - 10 \cdot (c_{i+1} - c_i)$$

where N_i is the number of shares held, S_i is the share price at the beginning of Day i and c_i is the BSM call option price at the beginning of Day i

The Training Algorithm

Goal:

- Build a Q-Table with all state-actions pairs

How:

- Generating artificial scenarios via Montecarlo simulation

```
# Initiate Q Table with Q-value = 0 for all state-action pairs
# Option Day to Maturity - TIME DIMENSION
M = 10
# Action [from -5 to 5]  - ACTION DIMENSION
N_ACTION   = 11
# Number of possible positions [from 0 to 10] - STATE DIMENSION
N_POSITION = 11
# Set number of state for stock price.         - STATE DIMENSION
# Prices are round to the nearest integer. Prices >= 107 are assumed to be
STOCK_PRICE_STATE = 15
q = np.zeros((s.M+1,s.STOCK_PRICE_STATE,s.N_POSITION,s.N_ACTION))
```

The Training Algorithm

Montecarlo Simulation

- TRAINING_SAMPLE is the number of simulated path
- Each path contains M time step (days to option maturity)
- Remember, we allow 1 Action per Day

Output:

- price_table (2D array) - asset price for each date for each simulation;
- call_price_table (2D array) - call option price for each date for each simulation
- delta_table - not used in this phase

The Training Algorithm

- Brownian Simulation Function

```
def brownian_sim(num_path, num_period, mu, std, init_p, dt):
    z = np.random.normal(size=(num_path, num_period))

    a_price = np.zeros((num_path, num_period))
    a_price[:, 0] = init_p

    for t in range(num_period - 1):
        #print(t)
        a_price[:, t+1] = a_price[:, t] * np.exp(
            (mu - (std ** 2) / 2) * dt + std * np.sqrt(dt) * z[:, t]
        )
    return a_price
```

The Training Algorithm

- Option Pricing Function

```
# Function for Black-Scholes Call Option Pricing & Black-Scholes Delta calculation
# T is time to maturity in year
def bs_call(v, T, S, K, r, q):
    np.seterr(divide='ignore', invalid='ignore')
    d1 = (np.log(S / K) + (r - q + v * v / 2) * T) / (v * np.sqrt(T))
    d2 = d1 - v * np.sqrt(T)
    bs_price = S * np.exp(-q * T) * norm.cdf(d1)
             - K * np.exp(-r * T) * norm.cdf(d2)
    bs_delta = np.exp(-q * T) * norm.cdf(d1)
    return bs_price, bs_delta
```

The Training Algorithm

Pseudocode for training phase

```
epsilon = 1
for k in range(TRAINING_SAMPLE):
        for t in range(DAYS_TO_MATURITY):
                if random <= epsilon:
                        # EXPLORATION
                        -> choose a random action
                else:
                    # EXPLOITATION
                    -> choose action corresponding to the higher expected value
                    which in this example is the minimum variance of pnl
                -> Compute Reward - Reward is calculated as expected variance
                        of total pnl from Hedging
    -> Update Q table according to the Temporal Difference Learning Rule
    Q_NEW = R * Q[t+1]
    Q     = Q + ALPHA*[Q_NEW - Q]
    -> Decrease epsilon to make the agente more likely to exploit
```

The Training Algorithm

- We generated three million paths for the stock price for the training set (the fact that three million paths are necessary for such a small problem emphasizes the point that reinforcement learning is *data hungry*)
- We then used a further 100,000 paths for the test set.
- Initially the **probability of exploration** was 100%.
- The probability of exploration decreases as the algorithm proceeds so that the probability of exploration during a trial is 0.999999 times its value at the preceding trial.
- The minimum value for the probability of exploration was 5%.

Test Set Results

- The objective was **to minimize the variance of total hedging costs**
- Our results were close to those given by delta hedging
- The mean absolute difference between positions taken by the algorithm and those that would be taken with delta hedging when averaged across all hedging days was 0.33

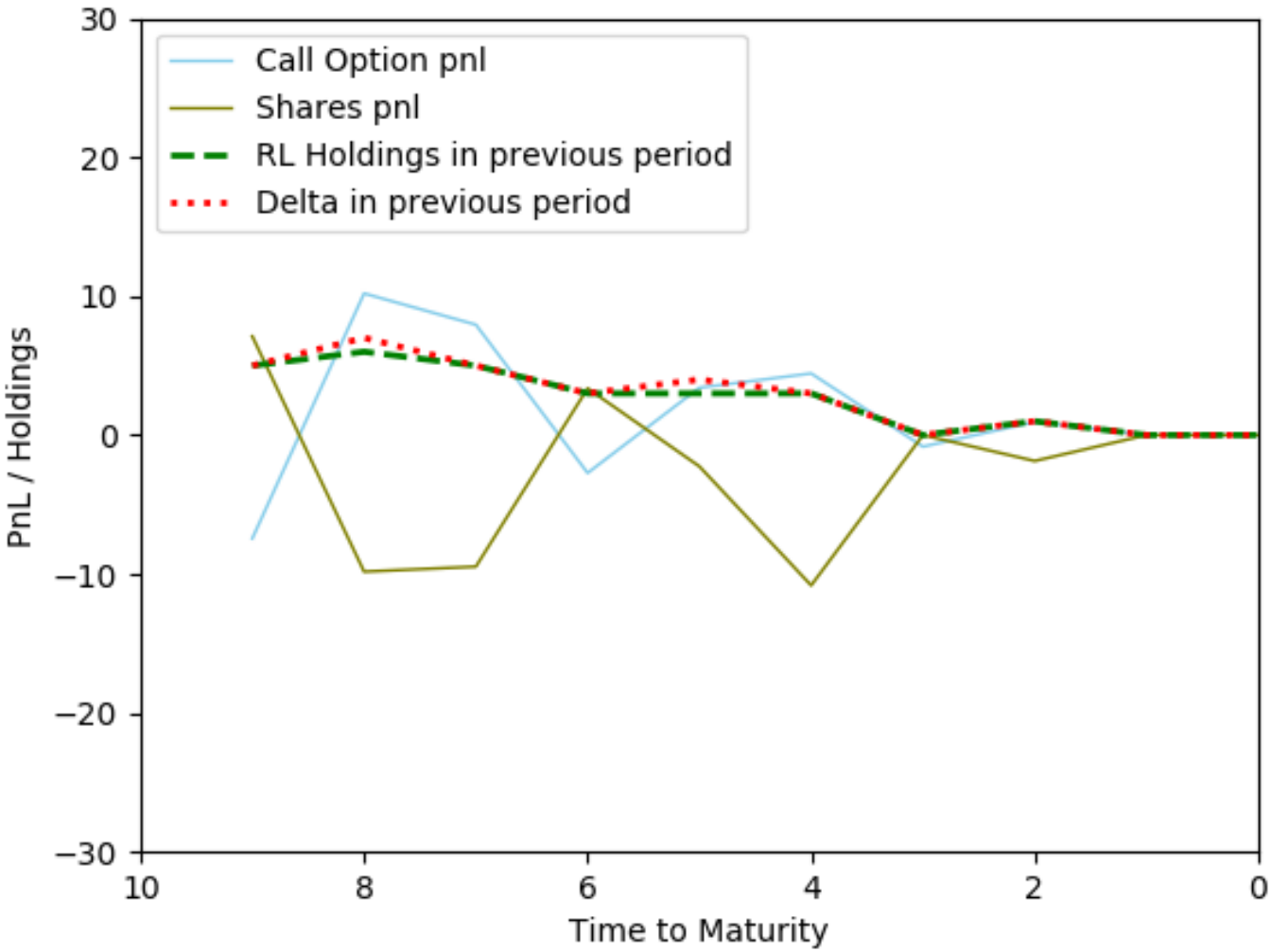
Test Set Results

- Pseudocode for Test Program

```
# Load the Q table from training
q = np.load('Q_RL_Hedging.npy')
# Asset paths simulation and BS-Parameters calculation for test phase
price_table = brownian_sim(...)
call_price_table, delta_table = bs_call(...)

for i in range(len(price_table)):
    for t in range(DAYS_TO_MATURITY):
        # Position to hold if the agent were to apply delta hedging
        new_dh_position = bsdelta[t]
        # Get action by finding the minimum non_zero value from the look up table
        rl_action = np.where(q==min) - 5
        new_rl_position = rl_action + rl_position
        rl_reward = (asset[t+1] - asset[t])*new_rl_position
        - 10*(call[t+1] - call[t])
        dh_reward = (asset[t+1] - asset[t])*new_dh_position
        - 10*(call[t+1] - call[t])
```

Test Set Results



Extension

- In practice, a trader faces bid-ask spreads, i.e., the ask price at which she or he can buy an instrument is generally higher than the bid price at which she or he can sell the instrument. If bid-ask spreads are not negligible, the trader wants to use a strategy where the cost of buying and selling is weighed against the reduction in risk.
- The objective function can then be

$$\sum_i \left(C_i + \alpha H_i^2\right)$$

where C_i is the transaction cost (arising from the bid-ask spread) incurred in period i and H_i is as before the change in the value of the hedger position on day i with the summation being taken over all hedging periods from the current one onward.

- The parameter α defines the trade-off between expected costs and variance of costs.

Extension

- Delta hedging is not optimal when transaction costs are considered and the problem is then a genuine multi-period problem that requires reinforcement learning.
- The code which accompanies the analysis can be extended to consider this problem