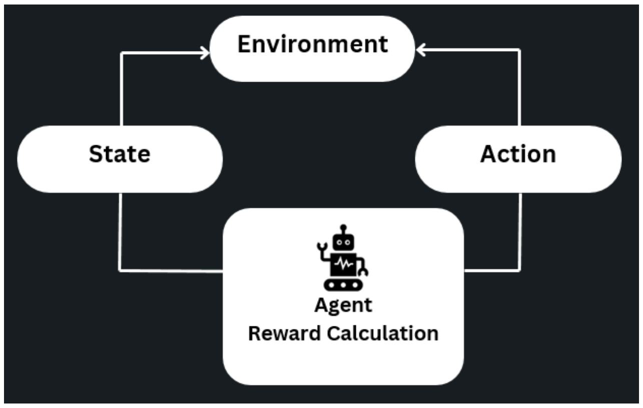
## 1- Reinforcement Learning (Review the slides, they are enough)

Reinforcement Learning is a branch of machine learning where an agent learns to make decisions by interacting with an environment. In an RL scenario, the agent observes the state of the environment, makes decisions (actions), and receives rewards (or penalties) based on the outcomes of its actions. The objective of the agent is to learn a policy (a mapping from states to actions) that maximizes the cumulative reward over time. This framework is powerful for solving a wide range of decision-making problems, from playing games to autonomous vehicle control.



## 2- Dynamic Programming (Review the slides as well)

Dynamic Programming is a method for solving complex problems by breaking them down into simpler sub-problems. In the context of RL, DP is used to solve the Bellman equation, which represents a recursive relationship between the value of a state and the values of the next states.

The **Bellman equation** forms the cornerstone of many RL algorithms. It provides a way to recursively decompose the value of a state into the immediate reward plus the discounted value of the next state.

$$V(s_t) = \max_a [r_t + \gamma V(s_{t+1}) \mid s_t = s]$$

**Episodic Tasks** are tasks that have a clear endpoint, such as reaching a goal or terminal state. Once an episode ends, the agent typically starts over from a beginning state.

**Continuing tasks** are those that do not have a defined endpoint and continue indefinitely, requiring ongoing decision making (action).

**We solved multiple examples about policy and value iteration in class, photocopy them from your friends who wrote during the session.**

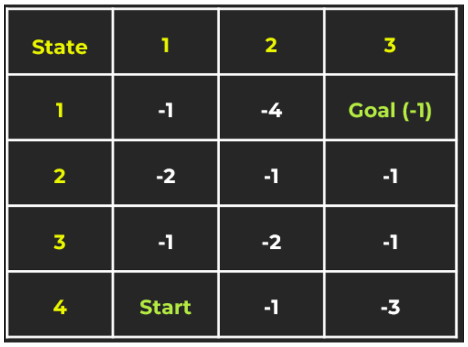
**Policy Iteration**: This method iterates between evaluating a policy (calculating the value of being in each state under this policy) and improving it by choosing actions that lead to higher-value states (going into the state that has a higher value). It alternates between two main steps: **policy evaluation** and **policy improvement**.

1- Given a policy π, policy evaluation computes its value function $\boldsymbol{V_\pi(s)}$**,** which is the expected return when starting in state s and following π. Then, the value function for policy π is obtained by solving the Bellman equation for π:

$$V_\pi(s_t) = \sum_a \pi(a \mid s_t) \sum_{s' \cdot r} p(s', r \mid s_t, a)[r_t + \gamma V_\pi(s_{t+1})]$$

$\pi(a \mid s_t)$**: Probability of taking action a from state** $s_t$

$p(s', r \mid s_t, a)$**: Probability of transitioning from state s' to state s after taking action a**



| State | 1 | 2 | 3 |
|---|---|---|---|
| 1 | -1 | -4 | Goal (-1) |
| 2 | -2 | -1 | -1 |
| 3 | -1 | -2 | -1 |
| 4 | Start | -1 | -3 |

2- Policy improvement aims to improves the policy by making it greedy with respect to the value function (select the action that gives the max expected reward), for each state s, the improved policy is given by:

$$\pi'(s) = argmax_a \sum_{s', r} p(s', r \mid s, a)[r + \gamma V_\pi(s')]$$

**Value Iteration**: A simpler and often faster alternative to policy iteration, it directly computes the optimal value function without requiring a separate policy evaluation step.

The update rule at each iteration k:

$$V_{k+1}(s) = \max_a \sum_{s', r} p(s', r \mid s, a)[r + \gamma V_k(s')]$$

This update rule directly computes the value of the best action to take in each state, without explicitly calculating the policy. Once the value function has converged to $\boldsymbol{V^*}$ **(when the value function of all the states becomes stable):**

$$\boldsymbol{\pi^*(s) = argmax_a \sum_{s',r} p(s',r \mid s,a)[r + \gamma V^*(s')]}$$

Value Iteration is efficient and often preferred in practice due to its simplicity and the fact that it directly converges to the optimal value function and policy without the need for separate evaluation and improvement steps.

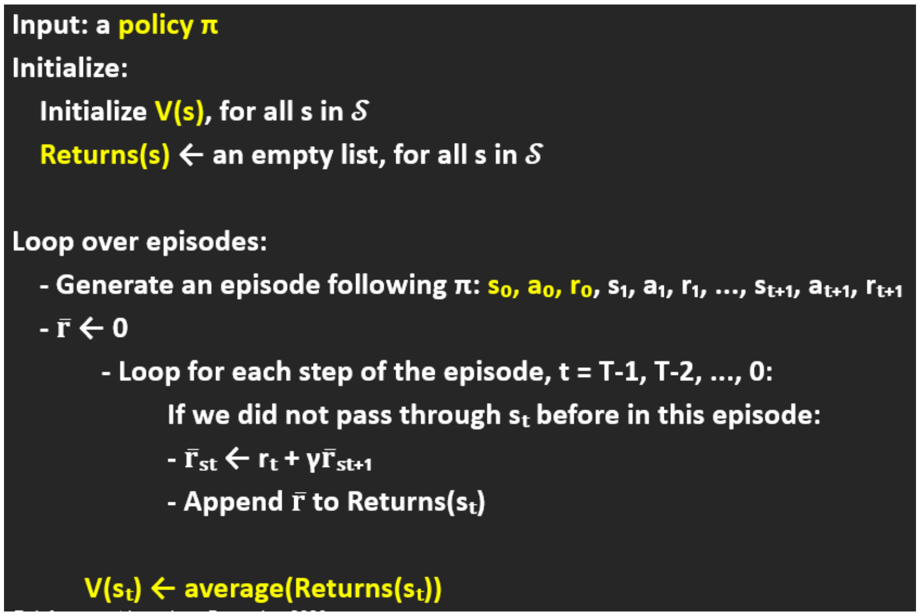
## 3- Monte Carlo Methods (Refer to the slides and the examples we solved in class)

Monte Carlo methods are used for estimating value functions and improving policies based on the average returns of episodes.

**We solved multiple examples about MC prediction and control in class, photocopy them from your friends who wrote during the session.**

**MC Prediction**: This involves estimating the value function of a policy by averaging the returns received after visiting a state across multiple episodes. Techniques include **first-visit MC** (only the first visit to a state is considered in each episode) and **every-visit MC** (all visits are considered).

- **First visit MC method (ensures that a state is only considered for update if it is visited for the first time.)**

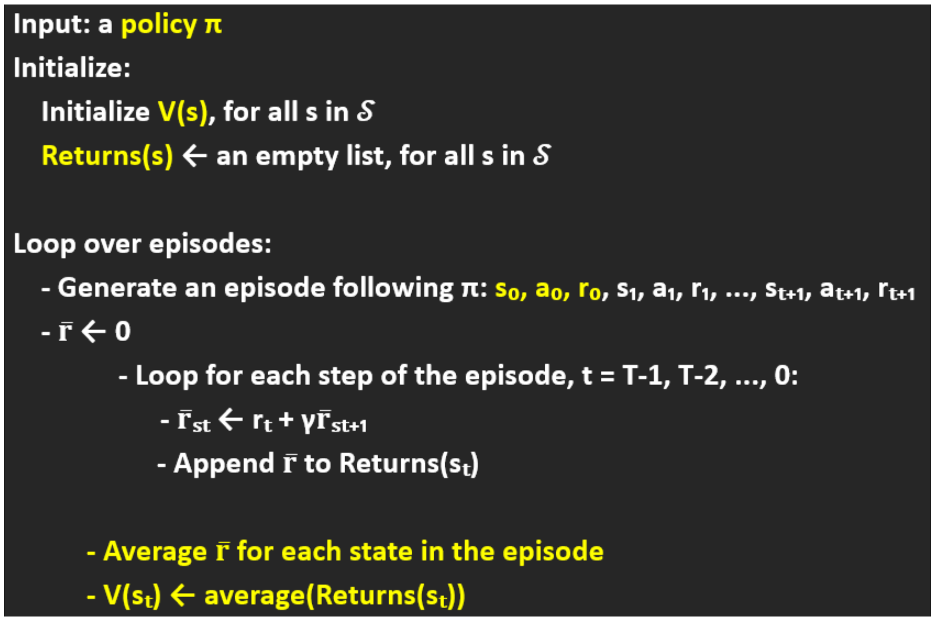


```
Input: a policy $\pi$
Initialize:
  Initialize $V(s)$, for all s in $\mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all s in $\mathcal{S}$

Loop over episodes:
  - Generate an episode following $\pi$: $s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{t+1}, a_{t+1}, r_{t+1}$
  - $\bar{r} \leftarrow 0$
      - Loop for each step of the episode, t = T-1, T-2, ..., 0:
          If we did not pass through $s_t$ before in this episode:
          - $\bar{r}_{st} \leftarrow r_t + \gamma\bar{r}_{st+1}$
          - Append $\bar{r}$ to $Returns(s_t)$

     $V(s_t) \leftarrow average(Returns(s_t))$
```

- **Every-visit MC method (updates the value of a state-action pair every time it is encountered during an episode, then averages them)**



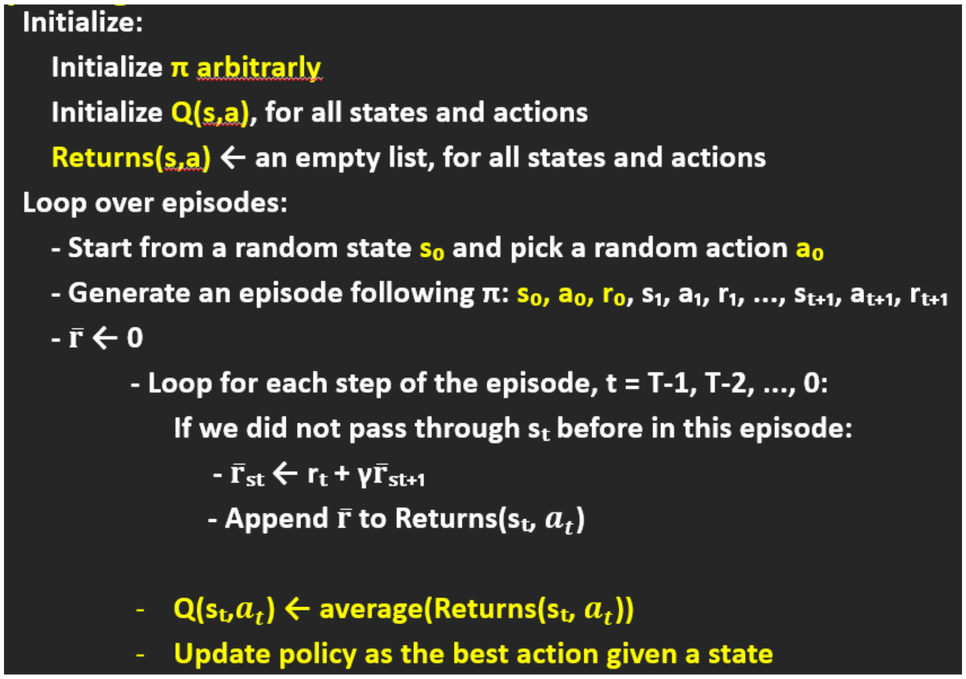
```
Input: a policy $\pi$
Initialize:
  Initialize $V(s)$, for all s in $\mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all s in $\mathcal{S}$

Loop over episodes:
  - Generate an episode following $\pi$: $s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{t+1}, a_{t+1}, r_{t+1}$
  - $\bar{r} \leftarrow 0$
      - Loop for each step of the episode, t = T-1, T-2, ..., 0:
         - $\bar{r}_{st} \leftarrow r_t + \gamma\bar{r}_{st+1}$
         - Append $\bar{r}$ to $Returns(s_t)$

     - Average $\bar{r}$ for each state in the episode
     - $V(s_t) \leftarrow average(Returns(s_t))$
```

**MC Control**: Monte Carlo control algorithms seek to directly find the optimal policy. They explore the environment by sampling and then refine the policy based on the sampled experience.

- **Exploring starts method (the agent initially explores various actions –basically it starts from a random state and selects a random action only in the first step- to learn about the environment before refining its policy)**



```
Initialize:
   Initialize $\pi$ arbitrarly
   Initialize Q(s,a), for all states and actions
   Returns(s,a) $\leftarrow$ an empty list, for all states and actions
Loop over episodes:
   - Start from a random state $s_0$ and pick a random action $a_0$
   - Generate an episode following $\pi$: $s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{t+1}, a_{t+1}, r_{t+1}$
   - $\bar{r} \leftarrow 0$
           - Loop for each step of the episode, t = T-1, T-2, ..., 0:
                If we did not pass through $s_t$ before in this episode:
                    - $\bar{r}_{st} \leftarrow r_t + \gamma \bar{r}_{st+1}$
                    - Append $\bar{r}$ to Returns($s_t$, $a_t$)

           -    Q($s_t$,$a_t$) $\leftarrow$ average(Returns($s_t$, $a_t$))
           -    Update policy as the best action given a state
```

- **Constant-α Monte Carlo methods: introduces the step size (learning rate) parameter to the equation of the value function. We wait until the end of the episode to update our value function by:**

$$\boldsymbol{V(S_t) \leftarrow V(S_t) + \alpha(\bar{r}_t - V(S_t))}$$

## 4- Temporal Difference Learning (Review the exercises we solved in class)

Temporal Difference learning is a reinforcement learning method that updates value estimates based on the difference between the value of the current state and the value of the next state. We do not wait until the end of the episode to update our value function like in MC methods.

### Simplest form

$$V(S_t) \leftarrow V(S_t) + \alpha(r_t + \gamma V(S_{t+1}) - V(S_t))$$

**NewEstimate = oldEstimate + stepSize(target – oldEstimate)**

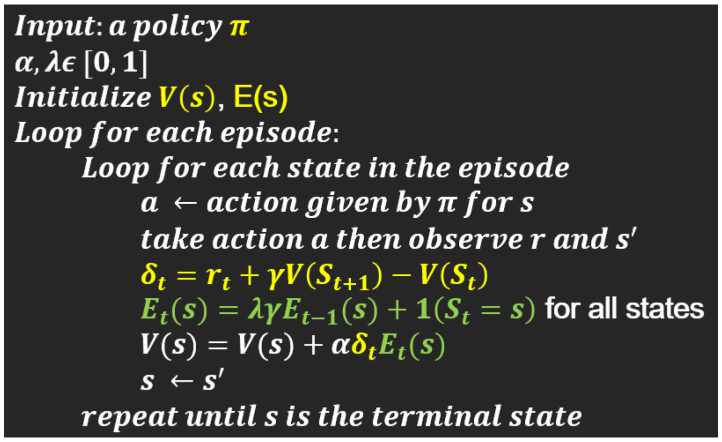
#### TD(0) and TD(λ)

TD(0) updates the value function based on the difference between the estimated values of consecutive states. TD(λ) extends this by considering a weighted average of multi-step returns, providing a smoother update mechanism.

- **TD(0) (updates are made based solely on the immediate next state)**

```
Input: a policy $\pi$
Step size: $\alpha \epsilon (0, 1]$
Initialize $V(s)$
Loop for each episode:
    $s = s_0$: first state
    Loop for each state in the episode
        $a \leftarrow$ action given by $\pi$ for $s$
        take action $a$ then observe $r$ and $s'$
        $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$
        $s \leftarrow s'$
    repeat until $s$ is the terminal state
```

- **TD(λ) incorporates a parameter λ to balance between bootstrapping from multiple future states and the immediate next state.**

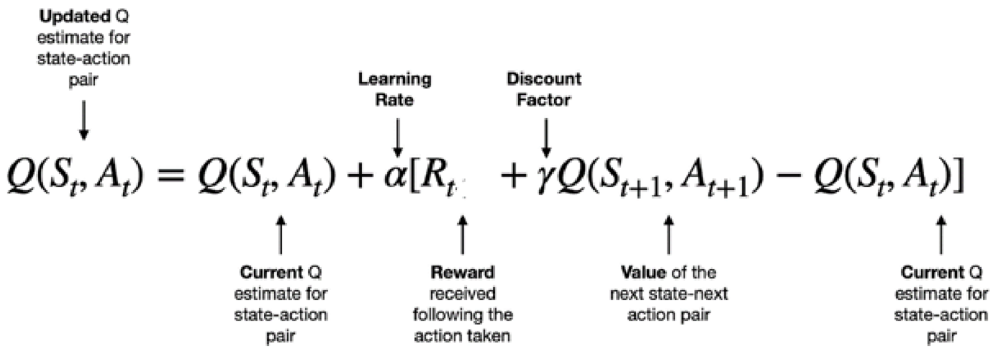


```
Input: a policy $\pi$
$\alpha, \lambda \epsilon [0,1]$
Initialize $V(s)$, E(s)
Loop for each episode:
    Loop for each state in the episode
        $a \leftarrow$ action given by $\pi$ for $s$
        take action $a$ then observe $r$ and $s'$
        $\delta_t = r_t + \gamma V(S_{t+1}) - V(S_t)$
        $E_t(s) = \lambda\gamma E_{t-1}(s) + 1(S_t = s)$ for all states
        $V(s) = V(s) + \alpha\delta_t E_t(s)$
        $s \leftarrow s'$
    repeat until s is the terminal state
```

Eligibility traces in TD(λ) are temporary records or traces of the state-action pairs that have been encountered during an agent's trajectory, which are used to update the value function in a way that incorporates credit assignment over multiple time steps. These traces decay over time (**λ** as decay parameter) and help to attribute credit to states and actions that contribute to the observed rewards, allowing for more efficient learning across a range of time scales.

**SARSA**: State-Action-Reward-State-Action, SARSA is an **on-policy** TD learning algorithm that updates the **action-value** function based on the current state, action taken, reward received, and the next state-action pair.

$$\boldsymbol{Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))}$$


Updated Q estimate for state-action pair
Learning Rate
Discount Factor
$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
Current Q estimate for state-action pair
Reward received following the action taken
Value of the next state-next action pair
Current Q estimate for state-action pair

**Q-Learning**: An **off-policy** algorithm that learns the optimal action-value function independently of the policy being followed by the agent. It updates its estimates based on the **maximum reward** that can be obtained from the next state (meaning it do not follow the policy in the update), promoting an exploration of the action space.

$$\boldsymbol{Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)}$$

## 5- Deep Reinforcement Learning

Deep Reinforcement Learning combines the decision-making framework of reinforcement learning with the powerful representation capabilities of deep neural networks (DL + RL = DRL). This integration allows agents to learn optimal policies directly from high-dimensional, raw sensory inputs, such as images or audio, which are typically challenging for traditional RL techniques to handle due to the curse of dimensionality.

**Why DRL?**

DRL addresses several challenges that are difficult for classical RL approaches, mainly due to the latter's limitations in handling high-dimensional state or action spaces. Classical RL methods, such as those based on tabular approaches or linear function approximators, struggle with the vastness and complexity of the state spaces encountered in real-world applications. Deep neural networks, by contrast, can efficiently represent complex functions and extract relevant features from raw sensory data, making them highly effective at approximating value functions, policies, and models of the environment.

**Some examples of where standard RL has been used in practice**

- Teaching robots to walk or move in a certain way
- Training game-playing AI agents to play simple games like tic-tac-toe

**Some examples of where DRL has been used in practice**

- Training game-playing AI agents to play more complex games like Go or chess
- Developing intelligent agents for robotic manipulation and control
- Training self-driving cars to navigate in complex environments
- Teaching robots to understand and respond to natural language

## 6- Deep Reinforcement Learning Algorithms

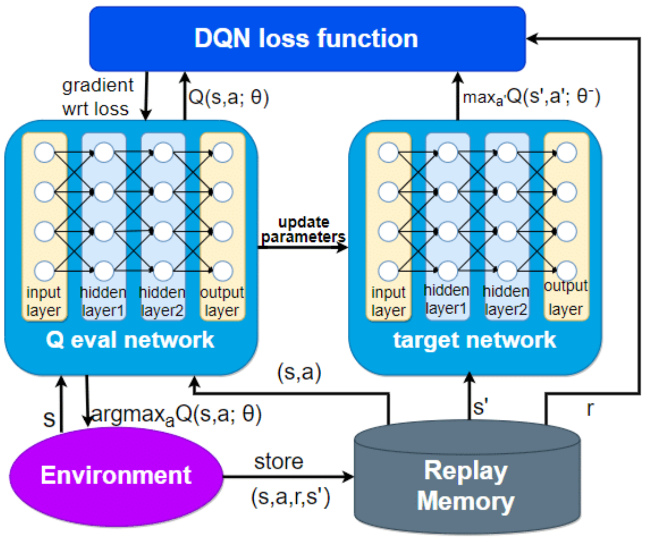
These algorithms have achieved remarkable success in various domains, from playing complex video games at a superhuman level to advancing robotics and autonomous systems. Below are some of the most popular DRL algorithms

### DQN (Deep Q-Networks)

Deep Q-Networks (DQN) combines deep neural networks with Q-learning. At its core, DQN employs a deep neural network to approximate the action-value function, Q(s, a), which estimates the expected return an agent will receive by taking action a in state s. The network takes states as input and outputs the estimated Q-values for each action. During training, DQN utilizes experience replay and target networks to stabilize learning. Experience replay randomly samples transitions from a replay buffer. Target networks provide a stable target for Q-value updates improving its policy.

- **Experience Replay**: This technique stores the agent's experiences at each time step in a replay buffer and randomly samples mini-batches from this buffer to update the network. This approach breaks the correlation between consecutive learning samples, leading to more stable and efficient learning.
- **Fixed Q-Targets**: DQN uses a separate network to generate the Q-value targets for each update. This target network's weights are fixed and only updated with the main

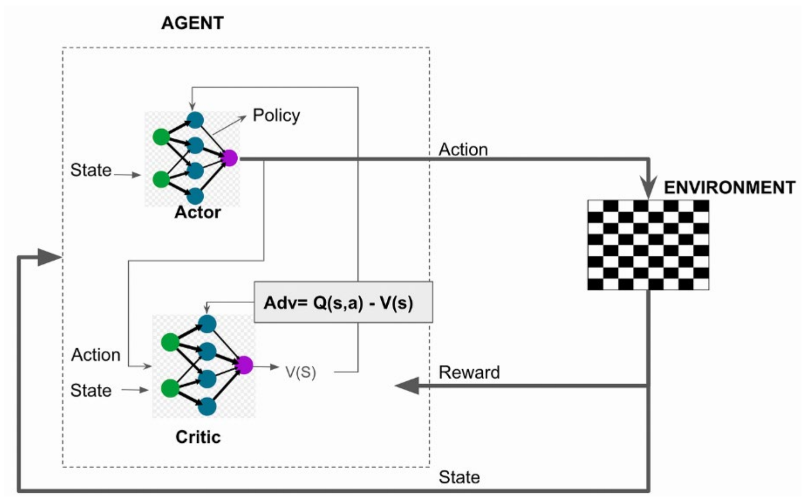
network's weights every few steps. This separation reduces the moving target problem, where updates rely on a constantly shifting set of Q-values.

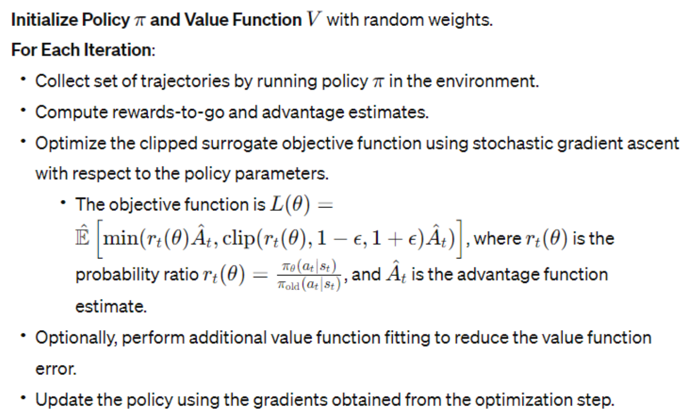


```
**Initialize Replay Memory D** with a fixed capacity.
**Initialize the Q-network Q** with random weights.
**Copy Q-network to create Target Q-network** $\hat{Q}$ with the same weights.
**For Each Episode**:
  • Observe initial state $s$.
  • **For Each Time Step**:
      • Select action $a$ using an $\epsilon$-greedy policy based on $Q$.
      • Execute action $a$, observe reward $r$ and next state $s'$.
      • Store transition $(s, a, r, s')$ in Replay Memory D.
      • Sample random mini-batch from D.
      • For each mini-batch sample, calculate target $Q$-value:
          • If $s'$ is terminal, $y = r$.
          • Otherwise, $y = r + \gamma \max_{a'} \hat{Q}(s', a')$.
      • Update $Q$ by minimizing $(y - Q(s, a))^2$.
      • Every $N$ steps, update $\hat{Q}$ with weights from $Q$.
```

## PPO (Proximal Policy Optimization)

Proximal Policy Optimization (PPO) addresses the challenge of stable and efficient policy optimization. PPO operates by iteratively updating a parameterized policy in a way that maximizes the expected cumulative rewards while ensuring that the policy changes are not too drastic. It achieves this through a combination of two key techniques: clipping and a surrogate objective. Clipping constrains the policy update to a region around the current policy, preventing large changes that could destabilize learning. The surrogate objective, derived from the ratio of new and old policy probabilities weighted by advantages, guides the policy updates in a direction that improves performance.




**Initialize Policy $\pi$ and Value Function $V$** with random weights.

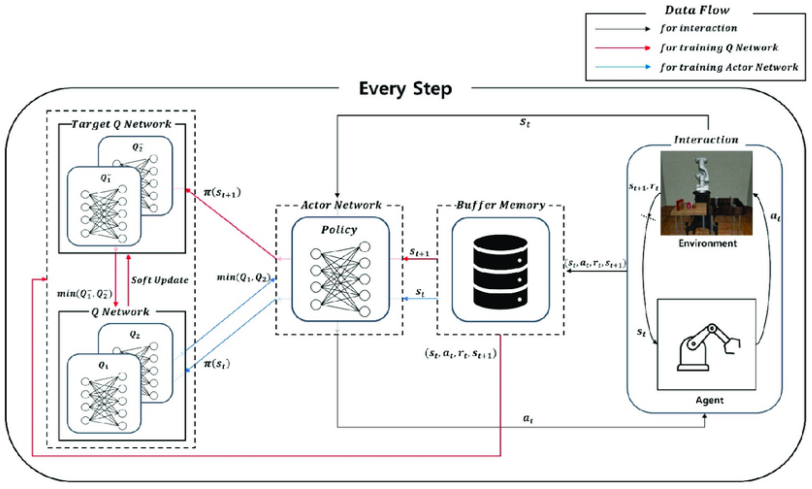
**For Each Iteration**:

- Collect set of trajectories by running policy $\pi$ in the environment.
- Compute rewards-to-go and advantage estimates.
- Optimize the clipped surrogate objective function using stochastic gradient ascent with respect to the policy parameters.
  - The objective function is $L(\theta) = \hat{\mathbb{E}}\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$, where $r_t(\theta)$ is the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$, and $\hat{A}_t$ is the advantage function estimate.
- Optionally, perform additional value function fitting to reduce the value function error.
- Update the policy using the gradients obtained from the optimization step.

## SAC (Soft Actor-Critic)

Soft Actor-Critic (SAC) is an off-policy algorithm that incorporates entropy into the reward structure, encouraging the agent to explore more diverse strategies. This approach is particularly effective in continuous action spaces and complex environments.

- **Entropy-Regularized Reinforcement Learning**: SAC adds an entropy bonus to the reward, incentivizing the agent not only to maximize expected returns but also to act as randomly as possible. This balance between exploration and exploitation results in more robust learning.
- **Actor-Critic Framework**: SAC uses a twin critic network to estimate the value function and a separate actor network to approximate the policy. This separation allows for more accurate value estimation and policy improvement.



**Initialize Policy Network $\pi$, two Q-Networks $Q_1, Q_2$, and Value Network $V$** with random weights.

**Initialize Target Value Network $V'$** with the same weights as $V$.

**For Each Episode**:

- Observe initial state $s$.
- **For Each Time Step**:
  - Select action $a$ according to the current policy $\pi(s)$.
  - Execute action $a$, observe reward $r$ and next state $s'$.
  - Store transition $(s, a, r, s')$ in the replay buffer.
  - Sample random mini-batch from the replay buffer.
  - Update critic networks $Q_1, Q_2$ by minimizing the Bellman error.
  - Update the value network $V$ by minimizing the value error.
  - Update the policy network $\pi$ by maximizing the expected return and entropy.
  - Update the target value network $V'$ with soft updates.

## 7- Trends in RL (The slides are enough)