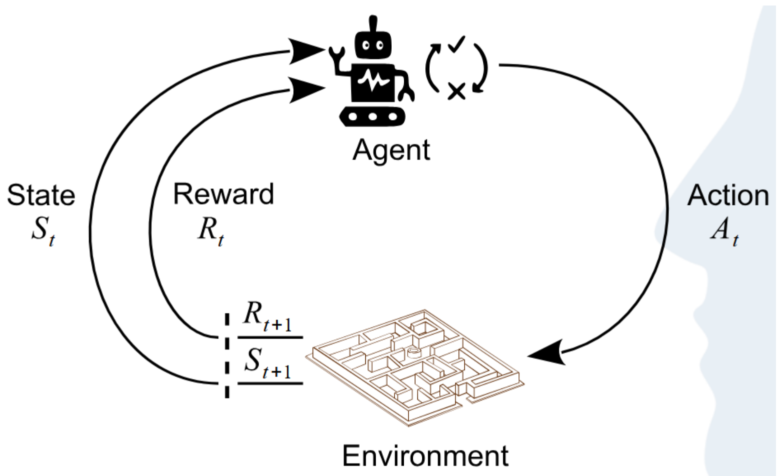
1 Reinforcement Learning

In Reinforcement Learning we assign each action a **positive or negative reward** depending on what we want the agent to do.

Reinforcement Learning algorithms attempt to find a policy (what action to do in which state) for **maximizing the cumulative reward** for the agent.



Interaction loop between agent and environment

Mathematically this is typically represented by a **Markov Decision Process (MDP)**.

Reinforcement learning differs from supervised learning, that we never explicitly show a desired outcome, we merely give feedback. Additionally we never correct sub-optimal actions.

One of the biggest problems in RL is how we assign rewards to actions. A lot of the time we do not assign a reward to just one action, as that would be very inefficient, but rather to a sequence of actions.

- Given a sequence of states and actions how do we define which actions were effective and ineffective?
- Actions might not immeadiately lead to a reward, they might have an effect that's only apparent later on
- Not every environment is deterministic, the same action might have different results
- Agents are not perfect - an action might not always be performed correctly

1.1 Markov Decision Process (MDP)

An MDP is defined by a 4D-tuple (S, A, T_a, R_a), where:

- S**:** A set of states $s \in S$
- A**:** A set of actions $a \in A$
- T_a**:** Transition function $T(s, a, s')$
  - Probability that a from s leads to s'
  - i.e. P(s' | s, a) also called the model
- R_a**:** Reward function $R(s, a, s')$
  - Sometimes just $R(s)$ or $R(s')$

Has a **start state** (or distribution), an optional **terminal state** and a **discount factor** γ.

$$\begin{tikzcd} s_0 \arrow[r, "a_0", "r_0"'] & s_1 \arrow[r, "a_1", "r_1"'] & s_2 \arrow[r, "a_2", "r_2"'] & s_3 \end{tikzcd}$$

Markov Property

The probability of moving to the next state depends only on the current state and action, **not on past states and actions**.

$$\boxed{P(S_{t+1} = s' | S_t = s_t, A_t = a_t)}$$

The future is independent of the past, given the present

1.1.1 Solve MDPs

To solve an MDP we want to compute an **optimal policy** $\pi^* : S \rightarrow A$.

- A policy gives an action for each state
- An optimal policy maximizes expected utility
- Defines a reflex agent

Deterministic Policy

$$a_t = \pi(s_t)$$

Stochastic Policy

$$a_t \sim \pi(\cdot|s_t)$$

At any time t, the agent tries to select an action to maximize the sum G_t of discounted rewards received in the future:

$$\boxed{G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}}$$

We discount **future rewards** with the discount factor γ to account for uncertainty in the future. $\gamma \in [0, 1]$ is often set to 0.9.

For each state we calculate a **utility value** equal to the sum of future rewards.

1.1.2 Optimal Utility

Optimal utilities define optimal policies:

- **Value function:**
  - Defines the value of a state s:
  - $V^*(s)$ = expected utility starting in s and thereafter acting optimally
- **Action-Value Function:**
  - Defines the value of a q-state (s,a):
  - $A^*(s, a)$ = expected utility starting in s, taking action a and thereafter acting optimally
- Define **optimal policy** π^***:**
  - $\pi^*(s)$ = optimal action from state s

A policy is **better than or equal** to another policy if its expected return is greater than or equal to the other policy **for all states**.

There is always at least one optimal policy π^* that is better than or equal to all other policies.

All optimal policies share the **same optimal state-value function** v^*, which gives the maximum expected return for any state s over all possible policies.

All optimal policies share the **same optimal action-value function** q^*, which gives the maximum expected return for any state-action pair (s,a) over all possible policies.

Formalization

$$\boxed{V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s)) V^\pi(s')}$$

- V^π: Value of state s when acting accoring to π
- $R(s, \pi(s))$: Immediate reward
- γ: Discount factor
- $T(s'|s, \pi(s))$: Transition probability to arrive in s'

1.2 Reinforcement Learning

Reinforcement learning assumes an MDP:

- **Set of states** $s \in S$
- **Set of actions** $a \in A$
- **Model** $T(s, a, s')$
- **Reward function** $R(s, a, s')$
- **Discount factor** γ

The twist: We don't know the model and the reward function. We only have access to the environment. So we must actually try out actions and states to learn.

Model-based Approach

Learn (or use) the model and use it to derive the optimal policy

Model-free Approach

Derive the optimal policy without learning the model

On-Policy

An on-policy agent only learns about the policy it is executing

Off-Policy

An off-policy agent learns about a policy different from the one it is executing. Can learn the optimal policy regardless of the policy it follows during learning

Passive Learning

The agent simply watches the world and tries to learn the utilities of the states and actions. The goal is to execute a fixed policy (sequence of actions) and evaluate it

Active Learning

The agent also acts. The goal is to act and learn an optimal policy.

Sample-Based Policy Evaluation

Improve estimate of V by computing rather than a single action:

$$\boxed{V_{k+1}^{\pi} = \frac{1}{n}\sum_i \text{sample}_i}$$

Temporal-Difference (TD) Learning

Instead of waiting for full episode completion, we update the value function $V(s)$ each time s occurs.

$$\boxed{V^{\pi}(s) = (1-\alpha)V^{\pi}(s) + \alpha(sample)}$$

Q-Learning

Q-Learning keeps track of samples and their Q-values, so that in the future it knows better which action to take in which situation.

$$\boxed{Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]}$$

Q-Learning yields great results if enough exploration is done and if the learning rate is small enough. It basically doesn't matter how actions are selected.
However, Q-Learning is very space-inefficient as it keeps a table of all q-values. Additionally in most realistic situations it is impossible to learn about every single state - Too many states to visit and too many states to hold in memory.

1.2.1 Exploration

How do we choose which action to take next?

- Simplest: random actions (ε greedy)
  - Every time step, "flip a coin"
  - With probability ε, act randomly
  - With probability $1-\varepsilon$, act according to current policy
- Problem with random action:
  - Random movement might not yield ideal results
  - One solution: lower ε over time
  - Another solution: exploration functions

1.2.2 Deep Q-Networks (DQN)

Similar to Q-Learning, but instead of using a Q-table to keep track of actions we train a deep neural network to estimate the best action.

Uses ε-Greedy policy to select actions.

The learning process is similar to training a neural network using **backpropagation**.

1.2.3 Policy Search

Often feature-based policies that work well aren't the ones that approximate V / Q best.

To remedy this, we can try to learn the policy that maximizes the rewards rather than the value that predicts rewards.

Simple Policy Search:

- Start with an initial value function or q-function
- Nudge each feature weight up and down and see if the policy improves

Problems:

- How do we tell if a policy got better?
- Many sample episodes needed
- A lot of features make this harder

1.3 Deep Neural Networks (AlphaZero)

AlphaZero combines Neural Networks with Tree Search.

Example, Monte Carlo Tree Search (MCTS):
MCTS is used to guide move selection by searching through possible future moves before making a decision:

1. Selection: Starting from root (current state), moves are selected based on prior visit counts and a balance between exploration and exploitation
2. Expansion: When an unexplored move is encountered, a new node is added to the tree
3. Evalutation: Neural network predicts the policy (move probabilities) and value (expected outcome) for the new node
4. Backpropagation: The value is propagated back up the tree to update the estimated win probabilities.

After multiple simulation, AlphaZero picks the move with the highest visit count, rather than the highest policy probability.

Hereby the search depth and breadth might be limited using the policy prediction.

1.3.1 Training Objective: Loss Formulization

$$\boxed{l = \underbrace{\alpha(z-v)^2}_{\text{Mean Squared Error}} - \underbrace{\pi^T \log p}_{\text{Cross Entropy}} + \underbrace{||\theta||^2}_{Regularizer}}$$

- α: value loss factor
- z: target value
- v: predicted value
- π^T: MCTS simulation distribution
- p: policy head output
- c: L_2 regularization constant