Introduction to Reinforcement Learning

INTELLIGENT SYSTEMS FOR PATTERN RECOGNITION (ISPR)
DAVIDE BACCIU – DIPARTIMENTO DI INFORMATICA - UNIVERSITA’ DI PISA
DAVIDE.BACCIU@UNIPI.IT
Preliminaries
Introduction to the RL Module

- RL Fundamentals
- Value Function Methods
- Policy Gradient Methods
- Exploration and Exploitation
- Deep reinforcement learning

A NOTE - Much of the content of this course and its slides are heavily based on the masterpiece course by David Silver

https://www.davidsilver.uk/teaching/
Introduction & Fundamental Concepts
What characterizes Reinforcement Learning (vs other ML tasks)?

- No supervisor: only a *reward* signal
- Delayed asynchronous feedback
- Time matters (sequential data, continual learning)
- Agent’s actions affect the subsequent data it receives (inherent non-stationarity)
Rewards

- A reward $R_t$ is a scalar feedback signal
- Indicates how well agent is doing at step $t$
- The agent’s job is to maximise cumulative reward
- Reinforcement learning is based on the reward hypothesis
- All goals can be described by the maximisation of expected cumulative reward
Sequential Decision Making

- Goal: select actions to maximise total future reward
- Actions may have long term consequences
- Reward may be delayed
- It may be better to sacrifice immediate reward to gain more long-term reward
- Examples:
  - A financial investment (may take months to mature)
  - Refuelling a helicopter (might prevent a crash in several hours)
  - Blocking opponent moves (might help winning chances many moves from now)
Agents and Environments

- $S_t^e$ is the environment $e$ private representation at time $t$
- $S_t^a$ the internal representation owned by agent $a$
- Full observability $\implies$ Agent directly observes the environment state $O_t = S_t^a = S_t^e$
- Formally this is a Markov Decision Process (MDP)
Partially Observable Environment

- Partial observability $\Rightarrow$ Agent indirectly observes the environment
  - A robot with camera vision only may not know absolute location
  - A trading agent only observes current prices
  - A poker player only observes public cards

- Formally $S^a_t \neq S^e_t$ and the problem is a Partially Observable Markov Decision Process (POMDP)

- The agent needs to build its own state representation $S^a_t$
  - History: $S^a_t = H_t$
  - Beliefs on environment state: $S^a_t = [P(S^e_t = s^1) \ldots P(S^e_t = s^N)]$
  - A dynamic memory (RNN): $S^a_t = \sigma(W_s S^a_{t-1} + W_o O_t)$
Components of a Reinforcement Learning Agent
Policy

- A policy $\pi$ is the agent’s behaviour
- It is a map from state $s$ to action $a$
  - Deterministic policy: $a = \pi(s)$
  - Stochastic policy: $\pi(a|s) = P(A_t = a|S_t = s)$
- A policy $\pi$ is a distribution over actions $a$ given states
Value Function

- The value function $v$ is a predictor of future reward.
- Used to evaluate the goodness/badness of states.
- And therefore to select between actions, e.g.

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | S_t = s]$$

Expected (discounted) future reward following policy $\pi$ from state $s$. 
Model

- A model predicts what the environment will do next
- Predict next state $s'$ following an action $a$
  \[ P_{ss'}^a = P(S_{t+1} = s' | S_t = s, A_t = a) \]
- Predict next reward
  \[ R_s^a = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] \]
A Forever Classic - The Maze Example

- **Rewards**: -1 per time-step
- **Actions**: N, E, S, W
- **States**: Agent location

Start

Goal
Maze Example (Policy)

Arrows represent policy $\pi(s)$ for each state $s$
Maze Example (Value Function)

Numbers denote the value $v_{\pi}(s)$ for each $s$

Expected time to reach the goal
Maze Example (Model)

- Agent may have an internal (imperfect) model of the environment
  - How actions change the state
  - How much reward from each state
- Grid Layout: transition model \( P_{ss'}^a \)
- Numbers: immediate reward model \( R_s^a \)
Learning Vs Planning

Two fundamental problems in sequential decision making

- Reinforcement Learning
  - The environment is initially unknown
  - The agent interacts with the environment
  - The agent improves its policy

- Planning (reasoning, introspection, search,...)
  - A model of the environment is known
  - The agent performs computations with its model (no external interaction)
  - The agent improves its policy
Markov Decision Processes
Introduction to MDPs

- Markov decision processes formally describe an environment for reinforcement learning
  - Environment is fully observable
  - i.e. The current state completely characterises the process
- Almost all RL problems can be formalised as MDPs, e.g.
  - Optimal control primarily deals with continuous MDPs
  - Partially observable problems can be converted into MDPs
  - Bandits are MDPs with one state
A Markov Decision Process (MDP) is a Markov chain with rewards and actions. It is an environment in which all states are Markov.

**Definition (Markov Decision Process)**

- A Markov Decision Process is a tuple \( \langle S, A, P, R, \gamma \rangle \)
  - \( S \) is a finite set of states
  - \( A \) is a finite set of actions \( a \)
  - \( P \) is a state transition matrix, s.t. \( P_{ss'}^a = P(S_{t+1} = s' | S_t = s, A_t = a) \)
  - \( R \) is a reward function, s.t. \( R_s^a = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] \)
  - \( \gamma \) is a discount factor, \( \gamma \in [0,1] \)
Return

Definition (Return)

The return $G_t$ is the total discounted reward from time-step $t$

\[ G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \]

- The value of receiving reward $R$ after $k + 1$ timesteps is $\gamma^k R$
- $\gamma$ values immediate reward Vs delayed reward
  - $\gamma \approx 0$ leads to "myopic" evaluation
  - $\gamma \approx 1$ leads to "far-sighted" evaluation
Bellman Equation for MDPs

- The state-value function $v(s)$ of a Markov Decision Process is the expected return starting from state $s$
  $$v(s) = \mathbb{E}[G_t | S_t = s]$$

- The value function $v(S_t)$ can be decomposed into two parts
  - Immediate reward $R_{t+1}$
  - Discounted value of successor state $\gamma v(S_{t+1})$

$$v(s) = \mathbb{E}[G_t | S_t = s] = \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t \right]$$

$$= \mathbb{E}[R_{t+1} | S_t = s] + \gamma \mathbb{E}[v(S_{t+1}) | S_t = s]$$

$$= \mathcal{R}_s + \gamma \sum_{s'} P_{ss'} v(s')$$
Value Function (with policy)

**Definition (Value Function)**

The state-value function $v_\pi(s)$ of an MDP is the expected return starting from state $s$ and following policy $\pi$

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

**Definition (Action-Value Function)**

The action-value function $q_\pi(s, a)$ is the expected return starting from state $s$, taking action $a$, and then following policy $\pi$

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$
Bellman Expectation Equation – Value and Action-Value Functions

The state-value function can again be decomposed into immediate reward plus discounted value of successor state

\[ v_\pi(s) = \mathbb{E}_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] = \sum_{a \in \mathcal{A}} \pi(a | s) q_\pi(s, a) \]

Similarly, we can decompose the action-value function

\[ q_\pi(s, a) = \mathbb{E}_\pi [R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_\pi(s') \]
Value and Action-Value Functions – One More Step of Nesting

\[ v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left( R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right) \]

The expected return of being in a state reachable from \( s \) through action \( a \) and then continue following policy

\[ q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a') \]

The expected return of any action \( a' \) taken from states reachable from \( s \) through action \( a \) (and then follow policy)
Finding an Optimal Policy

An optimal policy can be found by maximising over $q_*(s, a)$

$$
\pi_*(a|s) = \begin{cases} 
1 & \text{if } a = \arg \max_{a \in \mathcal{A}} q_*(s, a) \\
0 & \text{otherwise}
\end{cases}
$$

- There is always a deterministic optimal policy for any MDP
- If we know $q_*(s, a)$, we straightforwardly find the optimal policy
Bellman Optimality Equations

Optimal value functions are **recursively related** Bellman-style

\[
v_*(s) = \max_{a \in \mathcal{A}} q_*(s, a) = \max_{a \in \mathcal{A}} R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_*(s')
\]

\[
q_*(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_*(s') = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \max_{a' \in \mathcal{A}} q_*(s', a')
\]
Solving the Bellman Optimality Equation

- Bellman Optimality Equation is non-linear
- No closed form solution (in general)
- Many iterative solution methods
  - Value Iteration
  - Policy Iteration
  - Q-learning
  - SARSA
Model-Based RL
Iterative Policy Evaluation

- **Problem**: evaluate a given policy $\pi$
- **Solution**: iterative application of Bellman expectation backup

$$v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{\pi}$$

- **Using synchronous backups**
  1. At each iteration $k + 1$
  2. For all states $s \in S$
  3. Update $v_{k+1}(s)$ from $v_k(s')$ where $s'$ is a successor state of $s$
Iterative Policy Evaluation (Dynamic Programming)

\[ v_{k+1}(s) = \sum_{a \in A} \pi(a|s) \left( R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \right) \]

\[ v_{k+1} = R^\pi + \gamma P^\pi v_k \]
Evaluating a Random Policy in the Small Gridworld

- Undiscounted episodic MPD ($\gamma = 1$)
- Nonterminal states 1, ..., 14
- One terminal state (shown twice as shaded squares)
- Actions leading out of the grid leave state unchanged
- Reward is $-1$ until the terminal state is reached
- Agent follows uniform random policy
  - $\pi(n \mid \cdot) = \pi(s \mid \cdot) = \pi(e \mid \cdot) = \pi(w \mid \cdot) = 0.25$
Iterative Policy Evaluation on Small Gridworld (I)

$k = 0$

$\nu_k$

<table>
<thead>
<tr>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Greedy policy on $\nu_k$

$k = 1$

<table>
<thead>
<tr>
<th>0.0</th>
<th>-1.0</th>
<th>-1.0</th>
<th>-1.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>-1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

$k = 2$

<table>
<thead>
<tr>
<th>0.0</th>
<th>-1.7</th>
<th>-2.0</th>
<th>-2.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.7</td>
<td>-2.0</td>
<td>-2.0</td>
<td>-2.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.0</td>
<td>-2.0</td>
<td>-2.0</td>
<td>-1.7</td>
<td>0.0</td>
</tr>
<tr>
<td>-2.0</td>
<td>-2.0</td>
<td>-1.7</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

random policy
Iterative Policy Evaluation on Small Gridworld (II)

$k = 3$

<table>
<thead>
<tr>
<th></th>
<th>-2.4</th>
<th>-2.9</th>
<th>-3.0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>-2.4</td>
<td>-2.9</td>
<td>-3.0</td>
<td>-2.9</td>
<td></td>
</tr>
<tr>
<td>-2.9</td>
<td>-3.0</td>
<td>-2.9</td>
<td>-2.4</td>
<td></td>
</tr>
<tr>
<td>-3.0</td>
<td>-2.9</td>
<td>-2.4</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

$k = 10$

<table>
<thead>
<tr>
<th></th>
<th>-2.4</th>
<th>-2.9</th>
<th>-3.0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>-2.4</td>
<td>-2.9</td>
<td>-3.0</td>
<td>-2.9</td>
<td></td>
</tr>
<tr>
<td>-2.9</td>
<td>-3.0</td>
<td>-2.9</td>
<td>-2.4</td>
<td></td>
</tr>
<tr>
<td>-3.0</td>
<td>-2.9</td>
<td>-2.4</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

$k = \infty$

<table>
<thead>
<tr>
<th></th>
<th>-2.4</th>
<th>-2.9</th>
<th>-3.0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>-2.4</td>
<td>-2.9</td>
<td>-3.0</td>
<td>-2.9</td>
<td></td>
</tr>
<tr>
<td>-2.9</td>
<td>-3.0</td>
<td>-2.9</td>
<td>-2.4</td>
<td></td>
</tr>
<tr>
<td>-3.0</td>
<td>-2.9</td>
<td>-2.4</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

optimal policy
How to Improve a Policy

- Given policy $\pi$
  - Evaluate the policy $\pi$
    $$v_\pi(s) = \mathbb{E}_\pi [R_{t+1} + \gamma R_{t+2} + \cdots | S_t = s]$$
  - Improve the policy by acting greedily with respect to $v_\pi$
    $$\pi' = greedy(\pi) \Rightarrow \pi'(s) = \arg\max_{a \in A} q_\pi(s, a)$$

- In Small Gridworld improved policy was optimal, $\pi' = \pi^*$
- In general, need more iterations of improvement / evaluation
- But this process of policy iteration always converges to $\pi^*$
Policy Iteration

✓ Policy evaluation - Estimate $v_{\pi}$
✓ Iterative policy evaluation

✓ Policy improvement - Generate $\pi' \geq \pi$
✓ Greedy policy improvement
Modified Policy Improvement

- Does policy evaluation need to converge to $v_{\pi^*}$?
  - Introduce a stopping condition, e.g. $\epsilon$-convergence of value function
  - Stop after $k$ iterations of iterative policy evaluation, e.g. $k=3$ was sufficient in small gridworld

- Why update policy every iteration?
  - Stop after $k = 1$
  - This is equivalent to value iteration (coming up)
Generalized Policy Iteration

✓ Policy evaluation - Estimate $v_\pi$
✓ Any policy evaluation

✓ Policy improvement - Generate $\pi' \geq \pi$
✓ Any policy improvement algorithm
Value Iteration

✓ **Problem**: find optimal policy $\pi$

✓ **Solution**: iterative application of Bellman optimality backup

$$v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_\pi$$

✓ Using *synchronous backups*

  i. At each iteration $k + 1$
  ii. For all states $s \in S$
  iii. Update $v_{k+1}(s)$ from $v_k(s')$

✓ Unlike policy iteration, there is *no explicit policy*

✓ Intermediate value functions *may not correspond to any policy*
Value Iteration - Formally

\[ v_{k+1}(s) = \max_{a \in \mathcal{A}} \left( \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_k(s') \right) \]

\[ v_{k+1} = \max_{a \in \mathcal{A}} (\mathcal{R}^a + \gamma P^a v_k) \]
Wrap-up
Model-based Reinforcement Learning

<table>
<thead>
<tr>
<th>Problem</th>
<th>Bellman Equation</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td>Bellman Expectation Equation</td>
<td>Iterative Policy Evaluation</td>
</tr>
<tr>
<td>Control</td>
<td>Bellman Expectation Equation + Greedy Policy Improvement</td>
<td>Policy Iteration</td>
</tr>
<tr>
<td>Control</td>
<td>Bellman Optimality Equation</td>
<td>Value Iteration</td>
</tr>
</tbody>
</table>

- Algorithms are based on **state-value function** $v_\pi(s)$ or $v_*(s)$
  - Complexity is $O(mn^2)$ per iteration ($m = |A|$ and $n = |S|$)

- Could also apply to **action-value function** $q_\pi(s, a)$ or $q_*(s, a)$
  - Complexity is $O(m^2n^2)$ per iteration
Take home messages

- Reinforcement learning is a general-purpose framework for decision-making.
- Markov decision processes are a formalism to describe a fully-observable environment for reinforcement learning.
  - Can be relaxed to infinite and continuous actions/state and partially observable environments.
- Value functions have a recursive formulation using Bellman equations.
  - Any MDP allows for an optimal policy.
- Policy iteration - Re-define the policy at each step and compute the value according to this new policy until the policy converges.
- Value iteration - Computes the optimal state value function by iteratively improving the estimate of V(s).
- Policy vs Value iteration
  - Policy can converge quicker (agent is interested in optimal policy).
  - Value iteration is computationally cheaper (per iteration).
Next Lecture

✓ **Model-free prediction** - Estimate the value function of an unknown MDP
  o Monte-Carlo approaches
  o Temporal-Difference learning
  o TD($\lambda$)

✓ **Model-free control** - Optimise the value function of an unknown MDP
  o On-policy Vs Off-policy
  o SARSA($\lambda$)
  o Q-learning
Addons
DP Example

https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html
OpenAI Gym

A toolkit for developing and comparing reinforcement learning algorithms
✓ Implementation of the interaction environment
✓ Plug-in your agent with integration of main DL frameworks

```python
import gym

# create the environment
env = gym.make("FrozenLake-v0")
# reset the environment before starting
env.reset()

# loop 10 times
for i in range(10):
    # take a random action
    env.step(env.action_space.sample())
    # render the game
    env.render()

# close the environment
env.close()
```
Step 1 – Prepare a main learning loop

```
# spaces dimension
nA = env.action_space.n
nS = env.observation_space.n

# initializing value function and policy
V = np.zeros(nS)
policy = np.zeros(nS)

# some useful variable
policy_stable = False
it = 0

while not policy_stable:
    policy_evaluation(V, policy)
    policy_stable = policy_improvement(V, policy)
    it += 1

# Learning converged
run_episodes(env, policy)
```

Value function evaluation
Policy improvement on value function

Full code here
def policy_evaluation(V, policy, eps=0.0001):
    '''
    Policy evaluation. Update the value function until it reach a steady state
    '''
    while True:
        delta = 0
        # loop over all states
        for s in range(nS):
            old_v = V[s]
            # update V[s] using the Bellman equation
            V[s] = eval_state_action(V, s, policy[s])
            delta = max(delta, np.abs(old_v - V[s]))
        if delta < eps:
            break

def eval_state_action(V, s, a, gamma=0.99):
    return np.sum([p * (rew + gamma*V[next_s]) for p, next_s, rew, _ in env.P[s][a]])
Policy Update

```python
def policy_improvement(V, policy):
    
    Policy improvement. Update the policy based on the value function
    
    policy_stable = True
    for s in range(nS):
        old_a = policy[s]
        # update the policy with the action that bring to the highest state value
        policy[s] = np.argmax([eval_state_action(V, s, a) for a in range(nA)])
        for a in range(nA)]
        if old_a != policy[s]:
            policy_stable = False
    return policy_stable
```

\[ \pi'(s) = \arg \max_{a \in \mathcal{A}} q_{\pi}(s, a) \]
Value Iteration

Full Code Here

\[ v_{k+1}(s) = \max_{a \in A} \left( R_s^a + \gamma \sum_{s' \in S} p_{s,s'}^a v_k(s') \right) \]