Outline

• The reinforcement learning problem
• What to learn: policies and value functions
• Monte Carlo estimation for value functions
• Markov Decision Processes
• Dynamic programming methods
• Temporal-difference learning methods
• Learning optimal control
The General Problem: Control Learning

Consider *learning to choose actions*, e.g.,

- Robot learning to dock on battery charger
- Choosing actions to optimize factory output
- Playing Backgammon, Go, Poker, ...
- Choosing medical tests and treatments for a patient with a chronic illness
- Conversation
- Portofolio management
- Flying a helicopter
- Queue / router control

*All of these are sequential decision making problems*
Reinforcement Learning Problem

- At each discrete time $t$, the agent (learning system) observes state $s_t \in S$ and chooses action $a_t \in A$
- Then it receives an immediate reward $r_{t+1}$ and the state changes to $s_{t+1}$
Example: Backgammon (Tesauro, 1992-1995)

- The states are board positions in which the agent can move
- The actions are the possible moves
- Reward is 0 until the end of the game, when it is ±1 depending on whether the agent wins or loses
Supervised Learning

Training Info: Desired (target) Output

Error = (target output - actual output)
Reinforcement Learning (RL)

Training Info: Evaluations (rewards/penalties)

Inputs

Reinforcement Learning

Outputs: actions

Objective: Get as much reward as possible
Key Features of RL

- The learner is not told what actions to take, instead it finds out what to do by trial-and-error search.
- The environment is stochastic.
- The reward may be delayed, so the learner may need to sacrifice short-term gains for greater long-term gains.
- The learner has to balance the need to explore its environment and the need to exploit its current knowledge.
The Power of Learning from Experience

- Expert examples are expensive and scarce
- Experience is cheap and plentiful!

---

TD-Gammon self-play
Neurogammon same network, but trained from 15,000 expert-labeled examples

Tesauro, 1992
Agent’s Learning Task

Execute actions in environment, observe results, and learn policy (strategy, way of behaving) \( \pi : S \times A \rightarrow [0, 1] \),

\[
\pi(s, a) = P(a_t = a \mid s_t = s)
\]

If the policy is deterministic, we will write it more simply as \( \pi : S \rightarrow A \), with \( \pi(s) = a \) giving the action chosen in state \( s \).

- Note that the target function is \( \pi : S \rightarrow A \) but we have no training examples of form \( \langle s, a \rangle \)
  - Training examples are of form \( \langle \langle s, a \rangle, r, s', \ldots \rangle \)
- Reinforcement learning methods specify how the agent should change the policy as a function of the rewards received over time
The Objective: Maximize Long-Term Return

Suppose the sequence of rewards received after time step \( t \) is \( r_{t+1}, r_{t+2} \ldots \). We want to maximize the expected return \( E\{R_t\} \) for every time step \( t \)

- **Episodic tasks**: the interaction with the environment takes place in episodes (e.g. games, trips through a maze etc)

\[
R_t = r_{t+1} + r_{t+2} + \cdots + r_T
\]

where \( T \) is the time when a terminal state is reached
The Objective: Maximize Long-Term Return

Suppose the sequence of rewards received after time step $t$ is $r_{t+1}, r_{t+2} \ldots$. We want to maximize the expected return $E\{R_t\}$ for every time step $t$.

- **Discounted continuing tasks**:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=1}^{\infty} \gamma^{t+k-1} r_{t+k}$$

where $\gamma$ is a *discount factor* for later rewards (between 0 and 1, usually close to 1).

The discount factor is sometimes viewed as an "inflation rate" or "probability of dying"
The Objective: Maximize Long-Term Return

Suppose the sequence of rewards received after time step $t$ is $r_{t+1}, r_{t+2}, \ldots$. We want to maximize the expected return $E\{R_t\}$ for every time step $t$.

- **Average-reward tasks**:

\[
R_t = \lim_{T \to \infty} \frac{1}{T} (r_{t+1} + r_{t+2} + \cdots + r_T)
\]
Example: Mountain-Car

- States: position and velocity
- Actions: accelerate forward, accelerate backward, coast
- Two reward formulations:
  - reward = \(-1\) for every time step, until car reaches the top
  - reward = 1 at the top, 0 otherwise \(\gamma < 1\)
- In both cases, the return is maximized by minimizing the number of steps to the top of the hill
Example: Pole Balancing

Avoid failure: pole falling beyond a given angle, or cart hitting the end of the track
Example: Pole Balancing

Avoid failure: pole falling beyond a given angle, or cart hitting the end of the track

- Episodic task formulation: reward = +1 for each step before failure
  \[ \Rightarrow \text{return} = \text{number of steps before failure} \]

- Discounted continuing task formulation: reward = -1 upon failure, 0 otherwise, \( \gamma < 1 \)
  \[ \Rightarrow \text{return} = -\gamma^k \text{ if there are } k \text{ steps before failure} \]
What is the best policy?
Finding a good policy

- The problem seems difficult to solve even for toy examples
- Since we do not have expert-labeled examples, ideas for supervised learning do not apply immediately.
- One way to address the problem is to use search for a good policy, in the space of all possible policies
- To do this, we need a measure of the quality of a policy
State Value Function

- The **value of a state** \( s \) under policy \( \pi \) is the expected return when starting from \( s \) and choosing actions according to \( \pi \):

\[
V^\pi(s) = E_\pi\{R_0 \mid s_0 = s\} = E_\pi\left\{ \sum_{k=1}^{\infty} \gamma^{k-1} r_k \mid s_0 = s \right\}
\]

- If the state space is finite, the collection of values of all states, \( V^\pi \), can be represented as a vector of size equal to the number of states.

- This vector is called the **state-value function**
State-action value function

• Analogously, the value of taking action \( a \) in state \( s \) under policy \( \pi \) is:

\[
Q^\pi(s, a) = E_\pi \left\{ \sum_{k=1}^{\infty} \gamma^{k-1} r_k \mid s_0 = s, a_0 = a \right\}
\]

• \( Q^\pi \) can be represented as a matrix of size \(|S| \times |A|\); this is called the action-value function.
Policies and value functions

• Value functions define a partial order over policies:

\[ \pi_1 \geq \pi_2 \text{ if and only if } V^{\pi_1}(s) \geq V^{\pi_2}(s) \forall s \in S \]

• So a policy is “better” than another policy if and only if it generates at least the same amount of return at all states.

• If \( \pi_1 \) has higher value than \( \pi_2 \) at some states and lower value at other, the two policies are not comparable.

• Computing the value of a policy will be helpful in searching for it.
Monte Carlo Methods

• Suppose we have an episodic task
• The agent behaves according to some policy $\pi$ for a while, generating several trajectories.
• Compute $V^\pi(s)$ by averaging the observed returns after $s$ on the trajectories in which $s$ was visited.
• Two main approaches:
  – *Every-visit*: average returns for every time a state is visited in an episode
  – *First-visit*: average returns only for the first time a state is visited in an episode
Implementation of Monte Carlo Policy Evaluation

Suppose that we have \( n + 1 \) returns from state \( s \)

\[
V^{n+1}(s) = \frac{1}{n + 1} \sum_{i=1}^{n+1} R^i(s) = \frac{1}{n + 1} \left( \sum_{i=1}^{n} R^i(s) + R^{n+1}(s) \right)
\]

\[
= \frac{n}{n + 1} \frac{1}{n} \sum_{i=1}^{n} R^i(s) + \frac{1}{n + 1} R^{n+1}(s)
\]

\[
= \frac{n}{n + 1} V^n(s) + \frac{1}{n + 1} R^{n+1}(s)
\]

\[
= V^n(s) + \frac{1}{n + 1} \left( R^{n+1}(s) - V^n(s) \right)
\]

If we do not want to keep counts of how many times states have been visited, we can use a learning rate version:

\[
V(s_t) \leftarrow V(s_t) + \alpha_t (R_t - V(s_t))
\]
Monte Carlo estimation of action values

- We use the same idea: $Q^\pi(s, a)$ is the average of the returns obtained by starting in state $s$, doing action $a$ and then choosing actions according to $\pi$

- Like the state-value version, it converges asymptotically 
  
  \textit{if every state-action pair is visited}

- But $\pi$ might not choose every action in every state!

- \textit{Exploring starts:} Every state-action pair has a non-zero probability of being the starting pair
Representing value functions

- If the state space is finite, $V^\pi$ can be represented as an array with one entry for every state.
- If the state space is infinite, use your favorite function approximator that can represent real-values functions:
  - Linear function approximator, with non-linear basis functions
  - Nearest neighbor
  - Neural networks
  - Locally weighted regression
  - Regression trees
  - ...
- Some choices are better than others, theoretically and in practice.
Sparse, coarse coding

Main idea: we want linear function approximators (because they have good convergence guarantees, as we will see later) but with lots of features, so they can represent complex functions.

- **Coarse** means that the receptive fields are typically large.
- **Sparse** means that just a few units are active at any given time.

E.g., CMACs, sparse distributed memories etc.
Markov Decision Processes

• A general framework for non-linear optimal control, extensively studied since the 1950s

• In optimal control
  – Specializes to Ricati equations for linear systems
  – Hamilton-Jacobi-Bellman equations for continuous-time

• In operations research
  – Planning, scheduling, logistics, inventory control
  – Sequential design of experiments
  – Finance, marketing, queuing and telecommunications

• In artificial intelligence (last 15 years)
  – Probabilistic planning
Markov Decision Processes (MDPs)

- Set of states $S$
- Set of actions $A(s)$ available in each state $s$
- **Markov assumption**: $s_{t+1}$ and $r_{t+1}$ depend only on $s_t$, $a_t$ and not on anything that happened before $t$
- **Rewards**: 
  $$r_s^a = E\{r_{t+1}|s_t = s, a_t = a\}$$
- **Transition probabilities**
  $$p_{ss'}^a = P(s_{t+1} = s'|s_t = s, a_t = a)$$
- Rewards and transition probabilities form the **model** of the MDP
Optimal Policies and Optimal Value Functions

- In an MDP, there is a unique optimal value function:

\[ V^*(s) = \max_\pi V^\pi(s) \]

This result was proved by Bellman in the 1950s

- There is also at least one deterministic optimal policy:

\[ \pi^* = \arg \max_\pi V^\pi \]

It is obtained by greedily choosing the action with the best value at each state

- Note that value functions are measures of long-term performance, so the greedy choice is not myopic
Bellman Equations

Values can be written in terms of successor values

E.g. \( V^\pi(s) = E_\pi \{ r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \mid s_t = s \} \)

\[
= E_\pi \{ r_{t+1} + \gamma V(s_{t+1}) \mid s_t = s \}
\]

\[
= \sum_{a \in A} \pi(s, a) \left( r^a_s + \gamma \sum_{s' \in S} p^a_{ss'} V^\pi(s') \right)
\]

This is a system of linear equations whose unique solution is \( V^\pi \).

Bellman optimality equations for the value of the optimal policy:

\[
V^*(s) = \max_{a \in A} \left( r^a_s + \gamma \sum_{s' \in S} p^a_{ss'} V^*(s') \right)
\]

This produces a nonlinear system, but still with a unique solution.
Dynamic Programming

Main idea: turn Bellman equations into an update rules.

For instance, value iteration approximates the optimal value function by doing repeated sweeps through the states:

1. Start with some initial guess, e.g. $V_0$

2. Repeat:

$$V_{k+1}(s) \leftarrow \max_{a \in A} \left( r_s^a + \gamma \sum_{s' \in S} p_{ss'}^a V_k(s') \right)$$

3. Stop when the maximum change between two iterations is smaller than a desired threshold (the values stop changing)

In the limit of $k \to \infty$, $V_k \to V^*$, and any of the maximizing actions will be optimal.
Illustration: Rooms Example

Four actions, fail 30% of the time
No rewards until the goal is reached, $\gamma = 0.9$. 
Policy Iteration

1. Start with an initial policy \( \pi_0 \)

2. Repeat:
   (a) Compute \( V^\pi_i \) using policy evaluation
   (b) Compute a new policy \( \pi_{i+1} \) that is greedy with respect to \( V^\pi_i \)

until \( V^\pi_i = V^\pi_{i+1} \)
Generalized Policy Iteration

Any combination of policy evaluation and policy improvement steps, even if they are not complete.
Model-Based Reinforcement Learning

- Usually, the model of the environment (rewards and transition probabilities) is unknown.
- Instead, the learner observes transitions in the environment and learns an \( \hat{r}_s^a, \hat{p}_{ss'}^a \).

Note that this is a classical machine learning problem!

- Pretend the approximate model is correct and use it to compute the value function as above.
- Very useful approach if the models have intrinsic value, can be applied to new tasks (e.g. in robotics).
Asynchronous Dynamic Programming

- Updating all states in every sweep may be infeasible for very large environments
- Some states might be more important than others
- A more efficient idea: repeatedly pick states at random, and apply a backup, until some convergence criterion is met
- Often states are selected along trajectories experienced by the agent
- This procedure will naturally emphasize states that are visited more often, and hence are more important
Dynamic Programming Summary

- In the worst case, scales polynomially in $|S|$ and $|A|$.
- Linear programming solution methods for MDPs also exist, and have better worst-case bounds, but usually scale worse in practice.
- Dynamic programming is routinely applied to problems with millions of states.
- However, if the model of the environment is unknown, computing it based on simulations may be difficult.
The Curse of Dimensionality

- The number of states grows \textit{exponentially} with the number of state variables (the dimensionality of the problem)
- To solve large problems:
  - We need to \textit{sample} the states
  - Values have to be \textit{generalized} to unseen states using function approximation
Reinforcement Learning: Using Experience instead of Dynamics

Consider a trajectory, with actions selected according to policy $\pi$:

$$\cdots \rightarrow s_t \xrightarrow{a_t} s_{t+1} \xrightarrow{a_{t+1}} s_{t+2} \xrightarrow{a_{t+2}} s_{t+3} \xrightarrow{a_{t+3}} \cdots$$

The Bellman equation is: $V^\pi(s_t) = E_\pi [r_{t+1} + \gamma V^\pi(s_{t+1})|s_t]$ which suggests the dynamic programming update:

$$V(s_t) \leftarrow E_\pi [r_{t+1} + \gamma V(s_{t+1})|s_t]$$

In general, we do not know this expected value. But, by choosing an action according to $\pi$, we obtain an unbiased sample of it, $r_{t+1} + \gamma V(s_{t+1})$

In RL, we make an update towards the sample value, e.g. half-way

$$V(s_t) \leftarrow \frac{1}{2} V(s_t) + \frac{1}{2} (r_{t+1} + \gamma V(s_{t+1}))$$
Temporal-Difference (TD) Learning (Sutton, 1988)

We want to update the prediction for the value function based on its change from one moment to the next, called temporal difference

- **Tabular TD(0):**

  \[
  V(s_t) \leftarrow V(s_t) + \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \quad \forall t = 0, 1, 2, \ldots
  \]

  where \( \alpha \in (0, 1) \) is a step-size or learning rate parameter

- **Gradient-descent TD(0):**

  If \( V \) is represented using a parametric function approximator, e.g. a neural network, with parameter \( \theta \):

  \[
  \theta \leftarrow \theta + \alpha (r_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)) \nabla_\theta V_\theta(s_t), \forall t = 0, 1, 2, \ldots
  \]
Eligibility Traces (TD(\(\lambda\)))

- On every time step \(t\), we compute the TD error:

\[
\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)
\]

- Shout \(\delta_t\) backwards to past states
- The strength of your voice decreases with temporal distance by \(\gamma \lambda\), where \(\lambda \in [0, 1]\) is a parameter
Example: TD-Gammon

- Start with random network
- Play millions of games \textit{against itself}
- Value function is learned from this experience using TD learning
- This approach obtained the \textbf{best} player among people and computers
- Note that classical dynamic programming is not feasible for this problem!
RL Algorithms for Control

- TD-learning (as above) is used to compute values for a *given* policy \( \pi \)
- *Control methods* aim to find the *optimal policy*
- In this case, the behavior policy will have to balance two important tasks:
  - *Explore* the environment in order to get information
  - *Exploit* the existing knowledge, by taking the action that currently seems best
Exploration

• In order to obtain the optimal solution, the agent must try all actions

• $\epsilon$-soft policies ensure that each action has at least probability $\epsilon$ of being tried at every step

• Softmax exploration makes action probabilities conditional on the values of different actions

• More sophisticated methods offer exploration bonuses, in order to make the data acquisition more efficient

• This is an area of ongoing research...
A Spectrum of Solution Methods

- **Value-based RL**: use a function approximator to represent the value function, then use a policy that is based on the current values
  - Sarsa: incremental version of generalized policy iteration
  - Q-learning: incremental version of value iteration

- **Actor-critic methods**: use a function approximator for the value function *and* a function approximator to represent the policy
  - The value function is the **critic**, which computes the TD error signal
  - The policy is the **actor**; its parameters are updated *directly* based on the feedback from the critic.

E.g., policy gradient methods
Continual, on-line learning

Many RL methods can be understood as trying to solve the Bellman optimality equations in an approximate way.
Success Stories

- TD-Gammon (Tesauro, 1992)
- Elevator dispatching (Crites and Barto, 1995): better than industry standard
- Inventory management (Van Roy et. al): 10-15% improvement over industry standards
- Job-shop scheduling for NASA space missions (Zhang and Dietterich, 1997)
- Dynamic channel assignment in cellular phones (Singh and Bertsekas, 1994)
- Robotic soccer (Stone et al, Riedmiller et al...)
- Helicopter control (Ng, 2003)
- Modelling neural reward systems (Schultz, Dayan and Montague, 1997)
Reference books

- For RL: Sutton & Barto, Reinforcement learning: An introduction
- For MDPs: Puterman, Markov Decision Processes
- For theory on RL with function approximation: Bertsekas & Tsitsiklis, Neuro-dynamic programming