UNIVERSITY OF MARYLAND

MSML610: Advanced Machine Learning

Probabilistic Reinforcement Learning

Instructor: Dr. GP Saggese - gsaggese@umd.edu

References:

- AIMA Chap 17: Making complex decisions
- AIMA Chap 22: Reinforcement Learning

Sequential Decision Problems

- **Sequential Decision Problems**
  - Utilities Over Time
  - Algorithms for MDPs
- Reinforcement Learning

Sequential Decision Problems

- **Agents need to make decisions in the real world**:
  - In a stochastic environment (randomness, unpredictability)
    - E.g., weather conditions affecting a delivery route
  - Where utility depends on a sequence of decisions
    - E.g., planning a multi-step journey where each step influences the next
- **What is involved**
  - Uncertainty
    - Represent the lack of certainty in outcomes, modeled using probabilities
    - E.g., weather forecasts often include uncertainty (70% chance of rain)
  - Utility functions
    - Measure the desirability of outcomes by quantifying preferences
    - E.g., assign higher values to outcomes with more profit and lower risk
  - Rewards
    - Yielded by the environment as feedback for actions taken
    - E.g., receive points in a game for completing a level
  - Sensing
    - Gather information about the environment using sensors
    - E.g., a robot using a camera to detect obstacles in its path
  - Search and planning
    - Find a sequence of actions to achieve a goal
    - E.g., a GPS system planning the shortest route to a destination

Markov Decision Process

- **MDPs** are a formal model for sequential decision making
- **Assumptions**
  - Fully observable but stochastic environment
    - Sensors give agent the complete state of the environment $\forall t$
    - Next state is not completely determined by current state and agent's action
  - Initial state s_0
  - An agent takes action $a \in Actions(s)$ in each state s
  - Transition model
    - $\Pr(s'|s,a)$ is probability of reaching state s', if action a is done in state s
    - Markov assumption: probability depends on (s,a), not on history
  - Reward function
    - For every transition $s \to s'$ via a the agent receives a reward $R(s,a,s')$
    - Total reward depends on sequence of states and actions, i.e., environment history
  - Goal states
- E.g., a robot navigating a slippery surface
  - It knows its exact location and the map (fully observable)
  - Its wheels may slip unpredictably (stochastic outcome)

MDP: Solution

- The solution of an MDP is a **policy** *"in state s take action a"*

$$\pi(s) : s \rightarrow a \in Actions(s)$$

  - Any execution of the policy leads to a different **environment history** because of the stochastic nature of the environment
  - Environment history is a sequence of states and actions $(s_0, a_0) \rightarrow (s_1, a_1) \rightarrow \ldots \rightarrow (s_i, a_i) \rightarrow \ldots$
- A policy is measured by the **expected utility** of the environment history

$$U(\pi(s)) = \mathbb{E}[f(s_0, \pi(s), R(s, a, s'))]$$

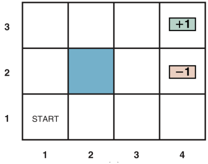
- The **optimal policy** $\pi^*(s)$ yields the highest expected utility

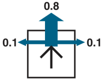
$$\pi^*(s) = \text{argmax}_\pi \, \mathbb{E}[f(s_0, \pi(s), R(s, a, s'))]$$

  - Note: the optimal policy is a function of the reward function
- MDP is often solved with **dynamic programming**
  1. Break the problem in smaller pieces recursively
  2. Solve the sub-problems
  3. Remember solutions of the pieces

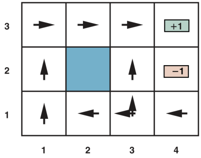
MDP: 4x3 Environment Example

- **Environment**
  - A 4 x 3 grid world
  - Fully observable: the agent always knows its location
  - Non-deterministic: actions are not reliable
    - Pr(intended action) $= 0.8$
    - Pr(move right/left angle) $= 0.1$
- **Agent**
  - Begin at the `START` cell
  - Choose actions `Up`, `Down`, `Left`, `Right` at each step
  - Aim to reach goal states marked `+1` or `-1`
- **Transition Model**
  - Result of each action in each state $\Pr(s'|s,a)$
- **Utility Function**
  - The reward for each state transition $s \to s'$ via action a is $R(s,a,s')$
    - -0.04 for all transitions (to encourage reaching terminal states swiftly)
    - $+1$ or -1 upon reaching terminal states
  - Total utility is the sum of all received rewards





Valid actions



Example of optimal policy

Utilities Over Time

- Sequential Decision Problems
  - **Utilities Over Time**
  - Algorithms for MDPs
- Reinforcement Learning

Utility Function

- The **utility function** for environment histories (finite or infinite) is expressed as:

$$U_h([s_0, a_0, s_1, a_1, ..., s_n, ...])$$

- A **finite horizon** indicates a fixed time N after which nothing matters:

$$U_h([s_0, a_0, s_1, ..., s_N, ...s_{N+k}]) = U_h([s_0, a_0, s_1, ..., s_N]) \, \forall k > 0$$

  - Actions are chosen based on the current state and remaining steps
  - Lead to non-stationary policies

- **Infinite horizon**
  - No fixed time limit, i.e., the process continues indefinitely
  - Utility is often defined using a discount factor $\gamma < 1$ for convergence
  - The optimal policy can be **stationary**
    - Policies do not depend on the specific time step
    - Same action is chosen whenever the agent visits the same state

Additive Rewards

- **Additive rewards**:
  - Rewards for each transition $s_i \xrightarrow{a_i} s_{i+1}$ are summed:

$$U_h([s_0, a_0, s_1, a_1, \ldots]) = \sum_{i=0} R(s_i, a_i, s_{i+1})$$

- **Additive discounted rewards**:
  - Include a discount factor $\gamma \in [0, 1]$:

$$U_h([s_0, a_0, s_1, a_1, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots$$
$$= \sum_{i=0} \gamma^i R(s_i, a_i, s_{i+1})$$

  where:
    - $\gamma = 1$: purely additive rewards
    - $\gamma \to 0$: future rewards are negligible
    - $\gamma \to 1$: future rewards significant
- **Pros of discounted rewards**:
  - Reflect human tendency to prioritize near-term rewards
  - In economics, early rewards can be reinvested, compounding further rewards
  - Support infinite horizons, preventing unbounded rewards

Expected Utility of a Policy

- We said that a policy leads to different environment histories
- The expected **utility of executing policy** π from state s:

$$U^{\pi}(s) = \mathbb{E}[\sum_{i=0}^{\infty} \gamma^t R(S_i, \pi(S_i), S_{i+1})]$$

  where the expectation $\mathbb{E}[\cdot]$ is:
  - Over state sequences determined by s, $\pi(s)$
  - The environment's transition model $\Pr(s'|s, a)$
- The agent should choose the **optimal policy**:

$$\pi^*_s = \text{argmax}_{\pi} U^{\pi}(s)$$

  - With discounted utilities and infinite horizons, the optimal policy is independent of the starting state: $\pi^*_s = \pi^*$
  - This is not true for finite-horizon policies or other reward combinations

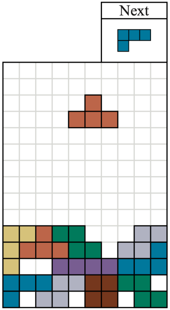
Principle of Maximum Expected Utility (MEU)

- MEU posits: *"A rational agent should choose the action that maximizes its expected utility based on its beliefs"*
- **Formal Definition**
  - Possible actions: $a \in A$
  - Possible outcomes: s'
  - Probability distribution: $\Pr(s'|a)$ for each action
  - Utility function: assign a numerical value $U(s')$ to each outcome
  - The expected utility of action a is (recursive):

$$EU(a) = \mathbb{E}[U(a)] = \sum_{s'} U(s') \Pr(s'|a)$$

  - Choose the action $a^* = \text{argmax}_{a \in A} \sum_{s'} U(s') \Pr(s'|a)$
- **Example**:
  - E.g., an agent must choose between:
    - `Action A`: 80% chance of reward 10; 20% chance of reward 0
    - `Action B`: 100% chance of reward 6
  - By MEU, choose `Action A`, since $EU(A) = 0.8 \cdot 10 + 0.2 \cdot 0 = 8 > EU(B) = 1.0 \cdot 6 = 6$

MDP: Tetris Example



- **States** S
  - Current board configuration and falling piece
- **Actions** A
  - Valid final placements of the piece
    - Rotation (0–3 positions)
    - Horizontal movement (left, right)
    - Hard drop (instant placement)
- **Transition Model** $T(s, a, s')$
  - Deterministic or stochastic based on next piece modeling
  - Piece generation often random (uniform or "bag" system)
- **Reward** $R(s, a, s')$
  - $+1$ for each cleared line
  - Negative reward for new block addition or height increase
  - Game over may have large negative reward
- **Discount Factor** γ
  - Close to 1 for valuing long-term survival and line-clearing

Utility of a State

- The utility of a state s, $U(s)$, reflects the long-term desirability of a state under optimal behavior
  - I.e., $U(s) = U^{\pi^*}(s)$, the expected sum of discounted rewards under an optimal policy from s
  - To remove the dependency from the policy, use the optimal policy
  - Calculated based on the expected rewards and the discount factor
- **Example**:
  - In a 4x3 environment, the utility of a state is:
    - Higher closer to the $+1$ state, as fewer steps are needed to reach it
    - Lower for the one close to the -1 state, since the agent needs to go around it
  - E.g., if the agent is two steps away from the $+1$ state, the utility will be higher compared to being four steps away
  - This assumes certain reward (e.g., $\gamma = 1$ and $r = -0.04$ for non-terminal transitions)

3	0.8516	0.9078	0.9578	+1
2	0.8016		0.7003	–1
1	0.7453	0.6953	0.6514	0.4279
	1	2	3	4

Bellman Equation

- The **utility of a state** s is the expected reward for the next transition plus the discounted utility of the next state, assuming the agent chooses the optimal action:

$$U(s) = \max_{a \in A(s)} \sum_{s'} \Pr(s'|s, a)[R(s, a, s') + \gamma U(s')]$$

where:
  - $A(s)$: set of actions available in state s
  - $\Pr(s'|s, a)$: probability of transitioning to state s' from state s by action a
  - $R(s, a, s')$: reward after transitioning from state s to s' using a
  - γ: discount factor, where $0 \leq \gamma < 1$
- Writing Bellman equations for all states gives a system of equations
  - Each state has its own equation based on its possible actions and transitions
  - Each equation is recursive: utility of s depends on utilities of its successor states
- Under certain conditions (e.g., finite state/action spaces, $\gamma < 1$):
  - This system has a unique solution
  - The utility function $U(s)$ is well-defined
  - E.g., in a grid world with a finite number of cells and actions

Bellman Equation: Intuition

- The **Bellman Equation:** says

  "Utility of a state = Best immediate action + Future potential"

  - Balances short-term gain and long-term value when outcomes are partly under the control of a decision-maker and partly random
- E.g., to find the fastest path to the goal in a maze, the Bellman equation prescribes:
  - *"Your current position is only as valuable as the best path out of it"*
  - Best path combines current proximity (reward now) and future position quality (reward later)
  - Value backs up from future to present, similar to tracing a route from finish to start
- E.g., in a chess game, the optimal strategy involves making the best move at each turn while considering future moves and potential outcomes

Q-Function

- Aka "Action-utility function"
- The Q-function $Q(s, a)$ is the expected utility of taking an *action* in *a given state*
  - Gives the expected value of choosing action a in state s, and then acting optimally afterward
- Utility of actions $Q(s, a)$ is the "dual" view of utility of states $U(s)$
  - Express the utility of a state in terms of utility of actions:

$$U(s) = \max_a Q(s, a)$$

  - Bellman equation for Q-functions

$$Q(s, a) = \sum_{s'} \Pr(s'|s, a)[R(s, a, s') + \gamma \max_{a'} Q(s', a')]$$

  - An optimal policy picks the "best" action:

$$\pi^*(s) = \operatorname{argmax}_a Q(s, a)$$

Shaping Theorem

- For discounted sums of rewards, the **scale of utilities** is arbitrary:
  - An affine transformation $U'(s) = m \cdot U(s) + b$ does not change the optimal policy $\pi^*(s)$
  - What matters for decision-making is the relative ordering of utilities (which is preserved)
- More generally, a **potential-based reward shaping**, i.e., using a function $\Phi(s)$ of the state s doesn't change the optimal policy:

$$R'(s, a, s') = R(s, a, s') + \gamma \Phi(s') - \Phi(s)$$

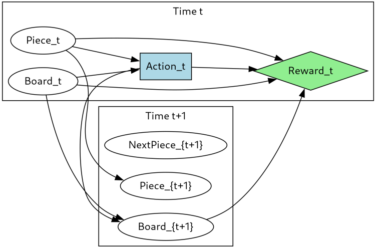
  - It ensures the difference in value between states remains consistent
- **Pros**
  - Speed: Can significantly speed up learning by guiding the agent
    - E.g., adding a potential function that increases with proximity to a goal can encourage faster convergence
    - E.g., animal trainers provide a small treat to the animal for each step in the target sequence
  - Safety: Prevents misleading the agent into a suboptimal policy
    - E.g., an agent might prioritize short-term rewards over long-term gains

Representing MDP

- The transition model $\Pr(s'|s,a)$ and the reward function $R(s,a,a')$ can be represented with:
  - Three-dimensional tables of size $|S|^2 \cdot |A|$
  - For sparse MDPs (i.e., each s transitions to only a few states s'), the table size is $O(|S| \cdot |A|)$
- MDPs can be represented using **Dynamic Decision Networks** (DDNs):
  - DDNs are a type of probabilistic graphical model extending Bayesian networks for sequential decision problems
  - DDNs offer a factored representation, compactly encoding state variables and dependencies
  - They are more scalable and expressive than atomic (flat) representations
    - E.g., in a large MDP with many states, a DDN can efficiently represent the problem without explicitly listing every possible state transition

Dynamic Decision Networks: Tetris Example

- DDN models Tetris in terms of time slices with the game's state, actions, and rewards
  - **State variables**:
    - $Board_t$: grid configuration at time t
    - $Piece_t$: current piece falling
    - $NextPiece_t$: upcoming piece
  - **Decision variable**:
    - $Action_t$: placement of $Piece_t$ (rotation and position)
  - **Chance nodes** (transition):
    - $Board_{t+1}$: board after action
    - $Piece_{t+1}$: next piece, depending on $NextPiece_t$ or random selection
  - **Utility node**:
    - $Reward_t$: derived from $Board_{t+1}$ (e.g., lines cleared, holes created)



Algorithms for MDPs

- Sequential Decision Problems
  - Utilities Over Time
  - **Algorithms for MDPs**
- Reinforcement Learning

Value Iteration (1/2)

- **Value iteration** solves MDPs using 2 steps:
  - Compute optimal utility for each state $U(s)$
  - Extract optimal policy π^* from utilities $U(s)$
- **Step 1**: compute optimal utility for each state
  - There are n possible states, so n Bellman equations, one per state

$$U(s) = \max_{a \in A(s)} \sum_{s'} \Pr(s'|s, a)[R(s, a, s') + \gamma U(s')]$$

  - Each equation relates the utility of a state to the utilities of its successors
  - The state utilities $U(s)$ are n unknowns
  - Solve these equations n equations with n unknowns simultaneously
    - Problem: equations are non-linear due to max operator
    - Solution: use an iterative approach

Value Iteration (2/2)

- **Solve system of Bellman equations**
  - Start with arbitrary values for utilities $U(s) = 0$
  - Perform Bellman updates:

$$U_{i+1}(s) \leftarrow \max_a \sum_{s'} \Pr(s'|s,a)[R(s,a,s') + \gamma U_i(s')]$$

  - Calculate the right-hand side and plug it into the left-hand side
  - No strict update order required for convergence, but intelligent ordering can improve speed, especially in large or structured MDPs
  - Repeat until equilibrium or close to convergence $||U_{i+1} - U_i|| < \epsilon$
  - Guaranteed to converge to the unique fixed point (optimal policy) for additive discounted rewards and $\gamma < 1$
- **Step 2**: compute optimal policy
  - Derive optimal policy by choosing action a that maximizes expected utility for each state s:

$$\pi^*(s) = \text{argmax}_a \sum_{s'} \Pr(s'|s,a)[R(s,a,s') + \gamma U(s')]$$

Policy Iteration

- **Policy iteration** solves MDPs by iteratively improving a policy
  - Alternates between evaluating the current policy and improving it
  - Uses the simplified Bellman equation with a fixed action per state
- **Algorithm steps**
  - Start with an initial (random) policy π
  - Policy Evaluation: compute $U^\pi(s)$ by solving:

$$U^\pi(s) = \sum_{s'} \Pr(s'|s, \pi(s))[R(s, \pi(s), s') + \gamma U^\pi(s')]$$

  - Policy Improvement: for each state, find:

$$\pi'(s) = \text{argmax}_a \sum_{s'} \Pr(s'|s, a)[R(s, a, s') + \gamma U^\pi(s')]$$

  - Repeat until policy is unchanged or close to convergence
- **Convergence Guarantee**
  - Each iteration strictly improves or maintains policy performance
  - Guaranteed to terminate with an optimal policy for finite MDPs
- **Efficiency Considerations**
  - Policy evaluation involves solving linear equations
  - Typically converges in fewer iterations than value iteration

Off-Line vs On-Line Solution of MDPS

- **Offline methods** (e.g., value iteration, policy iteration) precompute full solutions
  - Pros:
    - Compute the entire optimal policy $\pi^* \ \forall s$ before taking any action
  - Cons:
    - Assumes full knowledge of transition probabilities $\Pr(s'|s, a)$ and reward function $R(s, a, s')$
    - Not feasible for large MDPs (e.g., Tetris with 10^{62} states)
- **Online methods** compute actions at runtime, using only reachable parts of the state space
  - Interleave planning and acting
  - Agent explores the environment and updates estimates (e.g., Q-learning)
  - Pros:
    - Focuses computation only on relevant parts of the state space
    - Scales to large problems with appropriate heuristics and approximations
    - Allows adaptive, real-time decision-making
    - No need for full model of the MDP
  - Cons
    - Requires fast and accurate state evaluation functions
    - May require significant computation at each decision point
    - Needs exploration and careful tradeoff with exploitation
    - Sensitive to model accuracy and search depth

The n-Bandit Problem

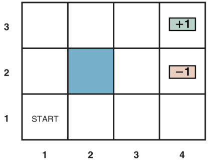
- A simplified reinforcement learning scenario
  - There are n different actions (arms)
  - Each arm a_i yields a reward drawn from an unknown probability distribution R_i
  - At each timestep t, agent selects an arm a_t and receives reward $r_t \sim R_{a_t}$
  - No state transitions: the environment is static and memoryless
  - Goal: maximize total reward over a sequence of pulls
- **Exploration vs. Exploitation**
  - Exploration: try different arms to learn their rewards
  - Exploitation: choose the best-known arm to maximize immediate reward
- **Applications**
  - Online advertising (choosing ads to show)
  - Clinical trials (testing treatments)
  - A/B testing in web development

Partially Observable MDPs (POMDPs)

- **Motivation**
  - Traditional MDPs assume full observability of the environment
  - The agent knows in which state it is in
  - In real-world situations, agents often lack precise knowledge of the current state
  - POMDPs (read "pom-dee-pees") extend MDPs to handle uncertainty in state perception
- **Definition**
  - A POMDP is defined by:
    - States S
    - Actions A
    - Transition model $\Pr(s'|s, a)$
    - Reward function $R(s, a, s')$
    - Sensor model $\Pr(e|s)$: probability of observing evidence e in state s
- **Belief States**
  - A belief state $b(s)$ is a probability distribution over possible actual states s (i.e., the probability of being in s)
  - The agent maintains $b(s)$ as its internal representation of the environment
  - Optimal policies depend on belief states: $\pi^*(b)$

POMDP: 4x3 World with Noisy Four-Bit Sensor

- The world is the 4x3 grid with partial and probabilistic information about the environment
- Use a noisy four-bit sensor, instead of knowing where the agent is
  - Detect obstacles in four directions: North, East, South, West
  - Produces a four-bit string (e.g., `1010`), each bit indicating presence (`1`) or absence (`0`) of a wall in one direction
- **Error Model**
  - Each bit is correct with probability $1 - \epsilon$, incorrect with probability ϵ
  - Errors are assumed to be independent across bits
  - Example: true config is `1100`, observed is `1110`
- **Localization Rule**
  - Helps infer the robot's position by comparing sensor output with map-based expectations (integrated into belief state updates)
  - Localization is achievable with high error rate by aggregating observations over time
  - E.g., if the robot believes to be in `(3, 2)`, moves left



Belief State Transitions and Value of Information

- **Belief Update**
  - After action a and observation e, belief state b is updated:

$$b'(s') = \alpha \Pr(e|s') \sum_s \Pr(s'|s,a)b(s)$$

    where α normalizes the distribution
  - Same equation as the filtering task to calculate the new belief state $b'(s)$ from the previous belief state $b(s)$ and the new evidence e
- **Belief space**
  - Everything (policy, transition and reward models) is now function of belief state
  - It can't be function of the actual state the agent is in, since the agent doesn't know the actual state
  - Intermediate belief states have lower utility due to uncertainty
  - Information-gathering actions can improve future decision quality
- **Transition and Reward Models in Belief Space**
  - Transition: $\Pr(b'|b,a)$ defined using:

$$\Pr(b'|b,a) = \sum_e \Pr(b'|e,a,b) \Pr(e|a,b)$$

  - Expected reward in belief state:

Solving POMDPs

- **Observable MDP over Belief Space**
  - A POMDP on an actual state space can be converted into an MDP on the belief space
- **Value Iteration for POMDPs**
  - Maintains a set of conditional plans p with associated utility vectors α_p
  - Expected utility of a plan in belief state b is $b \cdot \alpha_p$
  - Optimal utility is piecewise linear and convex over belief space
- **Recursive Plan Evaluation**

$$\alpha_p(s) = \sum_{s'} \Pr(s'|s,a) \left[R(s,a,s') + \gamma \sum_{e} \Pr(e|s') \alpha_{p.e}(s') \right]$$

- **Challenges**
  - Number of plans grows exponentially with depth
  - Even small problems generate many plans (e.g., 2^{255} plans for a two-state POMDP at depth 8)
  - Approximation Techniques

Reinforcement Learning

- Sequential Decision Problems
- **Reinforcement Learning**
  - Passive Reinforcement Learning
  - Active Reinforcement Learning
  - Generalization in Reinforcement Learning
  - Policy Search

Problem with Supervised Learning

- **In supervised learning**
  - An agent learns by observing examples of input / outputs
  - It's hard to find labeled data for all situations
- E.g., apply supervised learning to play chess
  - Take a board position as input $\underline{\boldsymbol{x}}$ and return a move m
  - Build a DB of grandmaster games with positions and winner (assuming moves by winner are good)
  - Problems
    - In a new game, positions differ from DB, as we have few examples compared to possible positions (10^{40})
    - The agent doesn't understand the game's goal (i.e., checkmate) or valid moves of each piece
- *"The AI revolution will not be supervised"* (Yann LeCun)

Reinforcement Learning

- **Reinforcement Learning (RL) Paradigm**
  - Agent learns from direct interaction with the environment
  - Periodically receives reward signals indicating success or failure ("reinforcements")
  - Learns a policy to maximize cumulative future rewards
  - Goal: maximize expected sum of rewards
- **RL vs supervised learning**
  - Providing a reward signal to the agent is easier than providing inputs / outputs
  - RL is active since the agent explores the environment and learn from actions and consequences
- **RL vs MDP**
  - The goal of both is to maximize the expected sum of rewards
  - In RL the agent:
    - Doesn't know the transition model or the reward function (doesn't know the rules)
    - Needs to act to learn more

Sparse vs Immediate Rewards

- Sparse rewards = in the vast majority of states the agent is not given informative reward
  - E.g., win/lose at the end of a chess game
  - The agent must explore many states to find the few that provide rewards
  - Often requires more sophisticated exploration strategies
- Immediate / intermediate rewards help guide learning
  - E.g.,
    - In tennis, you can get rewards for every point scored
    - Learning to crawl, any forward motion is a reward
    - In a video game, collecting coins or power-ups can serve as intermediate rewards
  - Provides continuous feedback to the agent

Applications of Reinforcement Learning

- **Games and Simulations**
  - RL has achieved superhuman performance in games like Go, Chess, and Dota2
  - Algorithms learn strategies through self-play and reward-driven improvement
- **Robotics**
  - RL enables learning of complex control policies for walking, grasping, and manipulation
  - Applications include robotic arms, quadrupeds, and autonomous drones
- **Autonomous Vehicles**
  - RL used for decision-making and control in self-driving cars
  - Handles tasks like lane merging, navigation, and obstacle avoidance
- **Recommendation Systems**
  - Adaptive recommendation based on user interactions (e.g., Netflix, YouTube) to optimize long-term engagement and satisfaction
- **Finance and Trading**
  - Portfolio management and trading strategies learned through market simulations
  - Agents aim to maximize returns under uncertainty and risk constraints
- **Healthcare**
  - Personalized treatment policies learned from patient data

Model-Based Reinforcement Learning

- **Definition**
  - Learns an explicit model of the environment's dynamics and uses it to make a decision about how to act
  - **Transition model**: estimates $\Pr(s'|s, a)$, i.e., probability of reaching state s' from s after action a
  - **Reward model**: estimates $R(s, a)$, i.e., expected reward after taking action a in state s
  - Intuition: learn to drive by studying the manual and physics
- **Learning Process**
  - Collects experience tuples (s, a, r, s')
  - Updates the model of the environment (transition and reward)
  - Plans using the model to improve policy (e.g., via value iteration or policy iteration)
  - Dyna-Q algorithm: combines model-free updates with simulated planning steps
- **Advantages**
  - Efficient sample usage: fewer real-world interactions required
  - Enables planning by simulating outcomes
- **Disadvantages**
  - Learning an accurate model is challenging
  - Errors in the model can propagate and lead to poor decisions

Model-Free Reinforcement Learning

- **Definition**
  - Learns directly from interactions with the environment without building a model of dynamics
  - Agent observes (s, a, r, s') and updates value or policy estimates based on observed outcomes
  - No attempt to predict $P(s'|s, a)$ or $R(s, a)$
  - Intuition: learn to drive by trial and error
- **Learning Process**
  - Value-based methods: Learn state or state-action values (e.g., $Q(s, a)$)
    - E.g., Q-learning
  - Policy-based methods: Learn the policy directly
    - E.g., REINFORCE, actor-critic
- **Advantages**
  - Simpler to implement when environment model is unknown or too complex
  - Robust to model inaccuracies since no model is used
- **Disadvantages**
  - Requires more environment interactions (sample inefficient)
  - Harder to incorporate planning or long-term reasoning

Model-Based vs Model-Free Reinforcement Learning

- **Core Distinction**
  - Model-Based RL: Learns a model of environment dynamics $P(s'|s,a)$ and $R(s,a)$ and uses it for planning
  - Model-Free RL: Learns value functions $Q(s,a)$ or policies $\pi(a|s)$ directly from experience
- **Sample Efficiency**
  - Model-Based: Generally more sample efficient due to simulated planning
  - Model-Free: Typically needs more environment interactions
- **Computation**
  - Model-Based: Higher planning overhead; simulations required
  - Model-Free: Simpler computations per step; often more scalable
- **Flexibility and Robustness**
  - Model-Based: Sensitive to model inaccuracies
  - Model-Free: More robust to model errors (since it doesn't learn one)
- **Typical Use Cases**
  - Model-Based: Robotics, planning tasks, known environments
  - Model-Free: Games, large-scale unknown or stochastic environments
- **Examples**
  - Model-Based: Dyna-Q, PILCO
  - Model-Free: Q-learning, Deep Q-Networks (DQN), REINFORCE

Active vs Passive Reinforcement Learning

- **Basic Distinction**
  - Passive RL: Learns value of a fixed policy; does not choose actions
  - Active RL: Learns both the value function and the optimal policy through exploration
- **Policy Handling**
  - Passive: Follows a given policy $\pi(s)$ and estimates $V^\pi(s)$ or $Q^\pi(s, a)$
  - Active: Improves policy over time, aiming for $\pi^*(s)$ that maximizes reward
- **Exploration**
  - Passive: No exploration — strictly evaluates the given policy
  - Active: Explores actions to improve the policy (e.g., ϵ-greedy, softmax)
- **Learning Goal**
  - Passive: Accurate value function for a known policy
  - Active: Optimal policy and value function via interaction
- **Algorithms**
  - Passive: Temporal Difference Learning (TD), Adaptive Dynamic Programming for a fixed policy
  - Active: Q-learning, SARSA, policy iteration methods
- **Use Cases**
  - Passive: Evaluation of policies from human demonstrations or expert systems
  - Active: Autonomous agents discovering optimal strategies from scratch

Passive Reinforcement Learning

- Sequential Decision Problems
- Reinforcement Learning
    - **Passive Reinforcement Learning**
    - Active Reinforcement Learning
    - Generalization in Reinforcement Learning
    - Policy Search

Passive Learning Agent

- Consider a fully observable environment with a small number of actions and states
- **The agent:**
  - Has a fixed policy $\pi(s)$ to determine its action
  - Needs to learn $U^{\pi}(s)$, the expected discounted reward if policy π is executed starting in state s
  - Doesn't know the transition model $\Pr(s'|s, a)$ and the reward function $R(s, a, s')$
- The agent executes a set of trials using the policy π:
  - Starts from an initial state and experiences state transitions until reaching terminal states
  - Stores actions and rewards at each state $(s_0, a_0, r_1, s_1, ..., s_n)$
  - Estimates:

$$U^{\pi}(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^{t} R(S_t, \pi(S_t), S_{t+1})]$$

- Direct utility estimation
  - For each state s, average the returns from all episodes in which s was visited:

Adaptive Dynamic Programming

- **Objective**
  - Learn utility estimates $U^{\pi}(s)$ for a fixed policy π using an estimated model of the environment
- **Key Components**
  - Model learning: Estimate transition probabilities $\Pr(s'|s,a)$ and reward function $R(s,a)$ from experience
  - Utility update: Solve the Bellman equations for the fixed policy:

$$U^{\pi}(s) = R(s,\pi(s)) + \gamma \sum_{s'} \Pr(s'|s,\pi(s)) U^{\pi}(s')$$

- **Learning Process**
  - Collect transitions $(s,\pi(s),r,s')$ during execution
  - Update model estimates:
    - $\Pr(s'|s,a) \approx$ empirical frequency
    - $R(s,a) \approx$ average observed reward
  - Use dynamic programming to compute $U^{\pi}(s)$
- **Advantages**
  - More sample-efficient than direct utility estimation
  - Leverages structure of the MDP to generalize better
- **Limitations**
  - Requires accurate model estimation
  - Computational cost of solving Bellman equations repeatedly

Temporal-Difference Learning

- **Objective**
  - Estimate utility values $U^\pi(s)$ for a fixed policy π using experience without a model
- **Key Idea**
  - Combine benefits of Monte Carlo methods and Dynamic Programming
  - Update estimates after every transition using bootstrapping
- **TD(0) Update Rule**
  - When a transition occurs from state s to state s' via action $\pi(s)$, we apply the update:

    $$U^\pi(s) \leftarrow U^\pi(s) + \alpha[r + \gamma U^\pi(s') - U^\pi(s)]$$

    where:
    - s is the current state
    - r is the immediate reward
    - s' is the next state
    - α is the learning rate
    - γ is the discount factor
- **Characteristics**
  - Online and incremental: updates occur after each step
  - Does not require knowledge of model $P(s'|s, a)$ or $R(s, a)$
- **Advantages**
  - More efficient and lower variance than Monte Carlo methods

Active Reinforcement Learning

- Sequential Decision Problems
- Reinforcement Learning
  - Passive Reinforcement Learning
  - **Active Reinforcement Learning**
  - Generalization in Reinforcement Learning
  - Policy Search

Active Reinforcement Learning

- Passive RL assumes agent has a fixed policy and passively receives reward signals
  - In many real-world cases, agent needs to decide what actions to take and rewards must be actively sought or queried
- Active RL includes cost-sensitive decisions about when to query for rewards
  - Useful when querying is expensive or limited (e.g., human feedback)
- Key problem: balancing cost of querying against benefit of accurate reward
- Formal model:
  - Agent observes state s and selects action a
  - Decides whether to query for reward r
  - Cost c incurred if query is made
- Objective:
  - Maximize cumulative reward minus query costs
  - $\sum(r_t - c_t)$ where $c_t = c$ if query made, 0 otherwise
- Optimal policy needs to learn both:
  - What actions to take
  - When it is worth querying for reward
- Applications:
  - Robotics with costly sensors

Greedy Agent in Reinforcement Learning

- A greedy agent always selects the action with the highest estimated value based on current knowledge or Q-values

$$a = \text{argmax}_a Q(s, a)$$

  for state s

  - No exploration: purely exploits known information $$

- An agent must make a tradeoff between
  - Exploitation of current best action to maximize its short-term reward
  - Exploration of unknown states to gain information that can lead to a change in policy (and greater rewards in the future)
  - E.g., in life you need to decide continuing a comfortable existence, or try something unknown in the hopes of a better life

- Goal: efficient learning with minimal queries to maximize information gain per unit cost

- Strategies include:
  - Random follow greedy policy or explore
  - Cost-aware exploration: modify exploration bonus based on query cost
  - Confidence-based querying: only query when uncertain about reward

Safe Exploration in Reinforcement Learning

- In idealized settings, agents can explore freely and learn from negative outcomes (e.g., losing in chess or simulations)
  - E.g., a self-driving car in simulation can crash without consequences
- In the real world, exploration has risks:
  - Irreversible actions may lead to states that cannot be recovered from
  - Agents can enter “absorbing states” where no further rewards or actions are possible
  - E.g., a crash that destroys a self-driving car permanently limits its future learning
- Safer Policy Approaches
  - **Bayesian Reinforcement Learning**: Maintain a probability distribution over possible models
    - Compute a policy that maximizes expected utility across all plausible models
    - In complex cases, leads to an “exploration POMDP” which is computationally intractable but conceptually useful
  - **Robust Control Theory**: Optimize for the worst-case scenario among all plausible models
    - Resulting policies are conservative but safe
    - E.g., agent avoids any action that could possibly lead to death
  - Impose constraints to prevent the agent from taking dangerous actions
    - E.g., safety controllers can intervene in risky states for autonomous helicopters

Temporal-Difference Q-Learning

- Q-learning is a model-free reinforcement learning algorithm
  - Learns the value of taking an action in a given state, denoted $Q(s, a)$
  - Does not require a model of the environment
- Temporal-difference (TD) learning updates estimates based on other learned estimates
  - Unlike Monte Carlo methods, it updates after every step using bootstrapping
- **Q-learning update rule:**

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

  - α: learning rate
  - r: reward received after action a
  - γ: discount factor for future rewards
  - s': next state
  - a': next action
- The update aims to reduce "the TD error" $r + \gamma \max_{a'} Q(s', a') - Q(s, a)$, i.e., the difference between current estimate and observed return

Generalization in Reinforcement Learning

- Sequential Decision Problems
- Reinforcement Learning
  - Passive Reinforcement Learning
  - Active Reinforcement Learning
  - **Generalization in Reinforcement Learning**
  - Policy Search

Generalization in Reinforcement Learning (1/2)

- Tabular representations become infeasible for large state spaces
  - Real-world problems often have millions or more distinct states
  - Example: Backgammon has $\sim 10^{20}$ states, but successful agents visit only a small fraction
- Function approximation enables scalability and generalization
  - Replace large tables with parameterized functions: $\hat{U}_\theta(s)$ or $\hat{Q}_\theta(s, a)$
  - Linear example: $\hat{U}_\theta(s) = \theta_1 f_1(s) + \cdots + \theta_n f_n(s)$
- Benefit: Generalizes from visited states to unvisited ones
  - Allows efficient learning with fewer examples
- Temporal-Difference (TD) and Q-learning adapt to function approximation
  - TD update:

$$\theta_i \leftarrow \theta_i + \alpha[r + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s)] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

  - Q-learning update:

$$\theta_i \leftarrow \theta_i + \alpha[r + \gamma \max_{a'} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)] \frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i}$$

  - Issues and solutions:
    - **Divergence**: parameters can grow uncontrollably
    - **Catastrophic forgetting**: important knowledge can be lost
    - **Solution**: experience replay reuses old data to stabilize learning

Policy Search

- Sequential Decision Problems
- Reinforcement Learning
  - Passive Reinforcement Learning
  - Active Reinforcement Learning
  - Generalization in Reinforcement Learning
  - **Policy Search**

Policy Search in Reinforcement Learning

- A policy $\pi(s)$ maps states to actions
  - Use a parameterized representation with fewer parameters than states (e.g., linear, deep neural network): $\pi_\theta(s)$
  - Directly optimizes parameters θ of the policy $\pi_\theta(s)$ rather than value functions
  - Pick the value with highest predicted value

$$\pi_\theta(s) = \text{argmax}_a \hat{Q}_\theta(s, a)$$

  - Useful in high-dimensional or continuous action spaces
- Even if learning a function replaces the Q-function, it is not an approximation of Q-function (i.e., Q_learning)
  - Seek a function that gives good performance and might differ from the optimal Q-function Q^*
- To avoid jittery policy for discrete actions, use stochastic policies for smoother optimization:
  - E.g., softmax over Q-values

$$\pi_\theta(s, a) = \frac{e^{\beta \hat{Q}_\theta(s,a)}}{\sum_{a'} e^{\beta \hat{Q}_\theta(s,a')}}$$

    where β controls exploration vs exploitation
- If everything is continuous and differentiable, use gradient descent to find