Reinforcement Learning (RL)

✅ Section 1 — Reinforcement Learning (RL): Introduction

1.1 What We Will Cover

- Introduction to Reinforcement Learning
- Real-world examples
- Key features of RL
- Demonstration concepts
- Directions for further reading

1.2 What is Reinforcement Learning?

Reinforcement Learning (RL) is a learning paradigm where an **agent** learns to make decisions by **interacting with an environment** in order to maximize a notion of **cumulative reward**.

Key idea:

RL is *learning by trial and error* with **delayed feedback**.

1.3 How RL Differs from Other Learning Paradigms

Supervised Learning

- Uses labeled data
- Learns hidden patterns from known input → output pairs
- Data is assumed i.i.d
- Feedback is immediate (prediction error is known instantly)

Unsupervised Learning

- Uses unlabeled data
- Finds structure/patterns
- Assumes i.i.d data
- No reward feedback loop

Reinforcement Learning

- No labels / no ground truth
- Environment provides rules and consequences
- Data is sequential and dependent
- Learns a *policy* for taking actions
- Feedback is **delayed**: reward comes after one or many steps
- Uses **trial-and-error** to discover the best actions

1.4 Example – Autonomous Helicopter

- A helicopter must figure out correct flight maneuvers through experimentation
- It receives **reward** (e.g., for maintaining balance or performing a stunt)
- Wrong actions/higher errors may make it fall → negative reward
- Shows how RL handles **continuous control**, **uncertain environment**, and **delayed outcomes**

1.5 Why RL Matters

RL is essential for any system where:

- Decisions must be made **sequentially**
- Current actions affect **future outcomes**
- Feedback is not always immediate
- The environment is **interactive**, dynamic, and uncertain

✅ Section 2 — Examples of Reinforcement Learning Applications

Reinforcement Learning appears in many complex decision-making problems. Below are key examples illustrating how RL concepts apply in the real world.

2.1 Chess

In chess, intelligent move selection combines:

- **Planning**
  - Anticipating future moves, counter-moves, and possible states
- **Intuition**
  - Evaluating desirability of positions without fully exploring the game tree

RL helps systems:

- Learn winning strategies through repeated play
- Evaluate long-term consequences of moves
- Improve policies with self-play (like AlphaZero)

2.2 Adaptive Controller in Petroleum Refinery

A refinery control system uses RL to:

- Adjust parameters in real time
- Maximize yield–cost–quality trade-offs
- Adapt based on **marginal costs**, not just fixed set points

Why RL is needed:

- Environment is **dynamic**
- Objectives can change
- Must optimize long-term performance, not instant reward

2.3 Mobile Robot Navigation

A mobile robot faces decisions like:

- Should it explore a new room for trash?
- Is it better to return to its charging station?

RL helps the robot learn:

- How battery level affects decisions
- How to balance **exploration vs. energy cost**
- How past experience of locating the charger influences future actions

---

2.4 What These Examples Teach

These scenarios highlight RL's strengths:

- Long-term planning
- Handling uncertainty
- Continual learning from interaction
- Balancing immediate vs. future rewards

✅ Section 3 — Tic-Tac-Toe as an RL Problem

Tic-Tac-Toe is simple for humans but surprisingly useful for demonstrating RL concepts.

It shows how RL can learn strategies **without explicitly solving the game mathematically**.

3.1 Why Tic-Tac-Toe is Interesting for RL

- Classical methods like **minmax or convex optimization** are not directly applied here (in the professor's framing).

- Instead, the agent must:
  - Learn from **experience (trial-and-error)**
  - Estimate **value** of board positions
  - Balance **greedy vs exploratory** moves

3.2 Key RL Components in Tic-Tac-Toe

Policy

- A rule that tells the agent what move to make from a given board state.

Reward Signal

- Immediate benefit for making a move (e.g., winning → +1, losing → 0).

Value Function

- Predicts the **long-term reward** of a position, not just the immediate outcome.
- Helps choose moves that may seem suboptimal now but lead to future wins.

3.3 Constructing the Value Table

The agent builds a table that assigns a value to **each possible board state**.

- If X wins:

  Value = 1
- If O wins:

  Value = 0
- Otherwise:

  Value = 0.5 (neutral start)

This table approximates:

P(win | current state) = how likely the agent is to eventually win from that board.

This table = **Value Function**.

3.4 Learning Through Playing Many Games

The agent improves by:

- Playing numerous games
- Observing transitions from one state to another
- Updating values along the trajectory

Two types of moves during learning:

1. **Greedy moves** – choose the best known action
2. **Exploratory moves** – try new actions to gain information

Learning occurs by updating state values based on game outcomes:

- Winning states reinforce preceding good states
- Losing states reduce value of harmful moves

3.5 Sequence of Actions (Trajectory)

During a single game:

- Some moves are **selected** (taken actions)
- Some moves are **not selected** (available but ignored)
- Some are **exploratory**
- Sequence helps the agent learn which trajectories lead to winning outcomes

The RL algorithm updates values using this full trajectory.

✅ Section 4 — RL Learning Method & Reward Hypothesis

This section introduces the **core objective** of RL: maximizing cumulative reward.

It also explains the nature of rewards and why RL uses them.

4.1 Reward Rt: The Basic Feedback Signal

In RL, the agent receives a **scalar reward** at each time step *t*.

Reward Rt tells the agent:

- How good or bad the last action was
- Whether it is making progress toward the goal

Important characteristics:

- Reward is **instantaneous** feedback
- It does **not** directly tell the agent what the optimal action is
- The agent must learn this over time

4.2 Goal of the Agent

The agent's goal is:

Maximize the cumulative reward over time.

Formally:

$\text{Maximize } \mathbb{E}\left[\sum_t R_t\right]$

This ensures RL focuses on:

- Long-term success
- Not just immediate wins or short-term gains

4.3 Reward Hypothesis

A fundamental principle of RL:

> All goals can be expressed as maximizing expected cumulative reward.

Implications:

- Any task (game-playing, robotics, control) can be encoded as a reward system
- Rewards must be designed carefully (reward shaping)
- The agent's behaviour emerges from the reward structure

4.4 Examples Illustrating Reward Structures

Problem	Reward (Good)	Penalty (Bad)
Fly stunt maneuvers	Following desired trajectory	Crash or wrong maneuver
Chess game	Win	Lose
Investment management	Profit	Loss
Refinery operations	Desired output, safety	Poor quality, unsafe states
Robot walking	Balanced motion	Falling over

These examples demonstrate:

- Rewards guide *desired behaviour*
- Penalties discourage dangerous or poor behaviour
- Proper reward design is crucial to successful learning

4.5 Why Rewards Matter

Rewards influence:

- How aggressively the agent explores
- How patient it is (long-term vs short-term gains)
- How well it avoids costly mistakes
- What strategies emerge during learning

A poorly designed reward can:

- Misguide the agent
- Cause unintended behaviour
- Slow down or prevent learning

✅ Section 5 — Sequential Decision Making

Reinforcement Learning problems are fundamentally **sequential**.

Each action not only affects the immediate reward but also **influences the future**.

5.1 Goal in Sequential Decision Making

The primary objective is:

Select actions that maximize total future reward.

This is different from greedy strategies that only optimize immediate steps.

5.2 Key Characteristics

1. Actions Have Long-Term Consequences

- Choosing poorly now may cause future penalties
- Choosing wisely may give delayed but larger benefits

2. Rewards May Be Delayed

Examples:

- In chess: blocking a move now may pay off many turns later
- In investments: long-term growth may require short-term sacrifices

3. Sacrificing Short-Term Reward Can Be Optimal

Agents must learn:

- Sometimes taking a small loss now can lead to bigger rewards later
- Example: fueling a helicopter or charging a robot — short-term cost, long-term benefit

---

5.3 Illustrative Examples

Financial Investment

- Might delay reward for long-term return

- Good decisions compound benefits over time

Refueling a Helicopter or Charging a Robot

- Takes time now → loses immediate progress
- But prevents catastrophic failures later

Blocking Opponent in a Game

- No instant gain
- But prevents large future losses

5.4 Why Sequential Nature Makes RL Hard

RL needs to:

- Predict long-term outcomes
- Handle uncertainty
- Explore enough to discover better future strategies
- Balance exploitation vs exploration

This makes RL different from static classification or regression problems.

✅ Section 6 — Agent and Environment

Reinforcement Learning centers around the interaction between two entities:

The Agent and **the Environment**.

Understanding this interaction is fundamental to everything in RL.

6.1 Agent

The **agent** is the decision-maker.

The agent:

- Observes the environment

- Chooses an action
- Learns from rewards
- Updates its policy to act better in the future

Examples:

- A robot
- A game-playing program
- A control system

6.2 Environment

The **environment** represents everything outside the agent.

The environment:

- Presents observations
- Responds to actions
- Provides rewards
- Evolves according to its internal rules

Examples:

- The game board in tic-tac-toe
- Physical world for a robot
- Market conditions for an investment agent

6.3 Agent–Environment Interaction Cycle

This cycle repeats at every time step *t*:

1. Agent executes an action

A_t

2. Environment responds with:

- **Observation for the next state**

  O_{t+1}

- **Reward**

  R_{t+1}

3. **Time index increments**, and the loop continues.

6.4 Example: Car Driving Scenario

- **Action (At):** turn, slow down, wipers on
- **Observation (Ot):** rain, speed limit, cars ahead
- **Reward (Rt):** points for safe or correct maneuver

This shows how real-world environments provide *implicit* feedback.

6.5 Why This Framework Matters

It enables RL algorithms to:

- Model dynamic systems
- Consider time and sequence
- Integrate trial-and-error learning
- Learn policies directly from interaction

This is the foundation for all RL formulations like MDPs, Q-learning, and policy gradients.

✅ Section 7 — History and State

In RL, the agent must make decisions based on **what it knows**.

This knowledge comes from the **history** of interactions and is often summarized into a **state**.

7.1 History (H_t)

The **history** is the complete record of everything the agent has experienced up to time *t*.

Formally:

$H_t = (O_1, R_1, A_1, \ldots, A_{t-1}, O_t, R_t)$

It includes:

- All **observations**
- All **actions**
- All **rewards**

Importance:

- Contains everything the agent has seen
- Too large and impractical to use directly
- Not used as-is in most RL algorithms

7.2 State (S_t)

A **state** is a compressed, useful summary of the history.

Formally:

$S_t = f(H_t)$

The state should contain:

- All the information needed to predict what happens next
- NO unnecessary or irrelevant detail

The goal is to construct a “good” state that gives the agent exactly the information it needs.

7.3 Types of State

1. Environment State (S_t^e)

- The actual internal state of the environment
- Not always fully visible to the agent
- May include hidden or irrelevant details
- Example: hidden cards in a game, exact wind speed affecting a robot

2. Agent State (S_t^a)

- Agent's internal representation
- Used to choose actions
- Based on observations/history
- What RL algorithms operate on

7.4 Why State Matters

A good state enables the agent to:

- Predict future rewards
- Choose effective actions
- Plan ahead
- Learn efficiently

A poorly designed state can:

- Confuse the agent
- Slow down learning
- Lead to suboptimal behavior

✅ Section 8 — Information State (Markov Property)

A central idea in Reinforcement Learning is the **Markov Property**, which defines when a state contains all necessary information about the past.

8.1 What is an Information State?

An **information state** (or *Markov state*) is a state representation that contains **all useful information** from the history needed to predict the future.

In other words:

> If the state is Markov, the future depends only on the current state, not the full history.

8.2 Markov Property (Formal Definition)

State (S_t) is **Markov** if:

$$P(S_{t+1} \mid S_t) = P(S_{t+1} \mid S_1, S_2, \ldots, S_t)$$

Meaning:

- Given the present state, the past does **not** provide extra information
- The state (S_t) is a complete summary of history relevant for prediction

---

8.3 Interpretation

"**The future is independent of the past given the present.**"

This simplifies RL tremendously because:

- The agent doesn't need the full history
- Algorithms can be built on transitions between states
- Many RL models (e.g., Q-learning) assume a Markov structure

---

8.4 Examples of Markov and Non-Markov Representations

Markov Examples

- Complete game board in Chess/Tic-tac-toe
- Robot position + velocity

- Market state with relevant indicators

These contain enough information to determine what happens next.

Non-Markov Examples

- Only the last camera frame of a robot
- Only the last stock price without trend history
- Partially observed environments

These **lack information**, so the agent cannot predict the future well.

8.5 Markov Decision Process (MDP)

If the environment is **fully observable**, meaning the agent directly observes the true environment state:

$$O_t = S_t^a = S_t^e$$

Then the problem becomes a **Markov Decision Process (MDP)**.

This is the foundation for:

- Value iteration
- Policy iteration
- Q-learning
- Many modern RL algorithms

8.6 Important Notes

- **Environment state (S_t^e)** is always Markov by definition
- **History (H_t)** is also Markov (since it contains everything)
- The goal is to find an efficient **agent state (S_t^a)** that is also Markov

---

✅ Section 9 — Partially Observable Environments (POMDP)

Not all environments allow the agent to fully see the true state.

Most real-world RL problems fall under **Partial Observability**.

9.1 What Does Partial Observability Mean?

In many environments:

- The agent cannot directly observe the true environment state
- Observations provide only **partial, noisy, or incomplete** information

Examples:

- A robot sees the world only through a camera
- An autonomous car cannot see beyond its sensors
- A trading agent cannot observe all market forces

Thus:

$O_t \neq S_t^e$

9.2 POMDP: Partially Observable Markov Decision Process

When observations are incomplete:

Agent State ≠ Environment State

The agent must **construct** its own internal representation of the world.

In a POMDP, the agent must infer:

- What state the environment *might* be in
- Based on all past observations and actions

This increases complexity dramatically.

9.3 How the Agent Handles Partial Observability

There are three common approaches:

(A) Maintain the Complete History

In theory, the history:

$$H_t = (O_1, R_1, A_1, \ldots, O_t, R_t)$$

contains all useful information.

But:

- It grows over time
- Cannot be used directly
- Computationally expensive

Still, this idea motivates memory-based models.

(B) Belief State Representation

The agent maintains a **probability distribution** over possible environment states.

For example:

- A robot doesn't know its exact position
- But it maintains probabilities over multiple likely locations

This is common in:

- Robotics (localization)
- Navigation
- Tracking problems

(C) Recurrent Neural Networks (RNNs)

RL agents often use RNNs to build internal memory.

The agent constructs its own state:

$$S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o)$$

Meaning:

- The RNN combines the past internal state + the new observation
- Learns to store only what is necessary
- Builds its own Markovian representation of the environment

This is used in:

- Atari games
- Autonomous driving
- Complex POMDP environments

9.4 Why POMDPs Are Hard

Partial observability leads to:

- Uncertainty
- Need for memory
- More exploration
- Higher sample complexity
- More complicated value/policy computations

Despite this, they reflect **real-world RL problems**, which are rarely fully observable.

✅ Section 10 — Components of Reinforcement Learning

Every RL system is built around **four fundamental components**:

Policy, Reward, Value Function, and Model.

These define how an agent learns, evaluates, and interacts with the environment.

10.1 Policy (π) — The Agent's Behavior

A **policy** tells the agent *what action to take* in each state.

Types of Policies:

1. Deterministic Policy

$a = \pi(s)$

- For each state, the action is fixed
- Example: “If battery < 20%, go to charger”

2. Stochastic Policy

$\pi(a|s) = P(A_t = a \mid S_t = s)$

- Specifies a probability distribution over actions
- Useful in exploration, games, and uncertain environments

10.2 Reward Function

Rewards measure **immediate gain or loss** from taking an action.

Characteristics:

- Simple scalar value
- Drives short-term behavior
- Does *not* encode long-term outcomes
- Must be well-designed to shape correct behavior

Examples:

- +1 for winning a game
- –1 for hitting a wall
- +0.1 for moving closer to the goal

10.3 Value Function (V or Q) — Predicting Future Reward

Value functions estimate **how good** it is to be in a certain state or state–action pair.

Two main types:

1. State Value Function

$$V^{\pi}(s) = \mathbb{E}[\text{future rewards} \mid S_t = s]$$

Measured under policy π.

2. Action Value Function (Q-function)

$$Q^{\pi}(s, a) = \mathbb{E}[\text{future rewards} \mid S_t = s, A_t = a]$$

Purpose:

- Evaluate states and actions
- Compare alternatives
- Guide improvements to the policy

Value functions capture long-term outcomes, unlike rewards.

10.4 Model of the Environment

A **model** predicts what will happen next:

1. Transition Model

$P(s' \mid s, a)$

Predicts the next state.

2. Reward Model

$R(s, a)$

Predicts the immediate reward.

Why a model is useful:

- Enables **planning**
- Let's agent simulate the environment internally
- Reduces need for real-world interaction

But:

- Models may be incorrect
- Hard to build in complex environments

10.5 Summary of the RL Components

Component	Role	Short vs Long-Term
Policy	Chooses actions	Action-level
Reward	Immediate feedback	Short-term
Value Function	Predicts future return	Long-term
Model	Predicts transitions + reward	Helps planning

These four components form the foundation of all RL algorithms.

✅ Section 11 — RL Example: Maze

The maze example illustrates how RL concepts—state, action, reward, value, and policy—work together in a simple environment.

11.1 Problem Setup

Environment

- A grid-based maze with **passages** and **blockages**

Agent

- Starts somewhere in the maze
- Tries to reach the goal efficiently

Actions

- **N** (North)
- **E** (East)

- **W** (West)
- **S** (South)

States

- Each grid cell = one state (agent's location)

Rewards

- **–1 per time step** (penalty for taking too long)
- Encourages the agent to **reach the goal faster**

11.2 Policy Representation

A **policy** π(s) assigns an action to each state.

In the visualization:

- Arrows (↑, →, ↓, ←) represent the agent's preferred action in each cell
- Shows how the agent intends to move at each location

This is the **current best strategy**, not necessarily optimal yet.

11.3 Value Function Representation

Each state has a **value** vπ(s):

- Represents the **expected total future reward**
- Higher value = better state (closer to goal)
- Lower value = worse or far away

Example numbers from the slide:

- States near the goal have values like –1, –2 (less penalty)
- Far states have large negative values like –16 or –17

  → means many steps still needed → more cumulative penalty

Thus:

$$v_\pi(s) = \text{expected future reward under policy } \pi$$

11.4 Reward Model Representation

The agent may learn an internal reward model R(s):

- Predicts the immediate reward at each state
- In the maze, this is always 1
- But the agent may estimate or approximate it

A model is **not required** for all RL methods (model-free methods skip this).

11.5 Key Takeaways From the Maze Example

- **States** are agent positions
- **Actions** move in four directions
- **Reward** penalizes time wasting
- **Policy** suggests a direction to move
- **Value Function** tells how good each state is

This example shows how RL solves navigation/trajectory problems efficiently.

✅ Section 12 — Types of RL Agents

RL agents can be categorized based on whether they use a **policy**, a **value function**, a **model of the environment**, or combinations of these.

This taxonomy helps understand the different RL methods.

12.1 Value-Based Agents

These agents **learn a value function** but do **not** explicitly represent a policy.

Examples:

- Q-learning

- SARSA

How they choose actions:

- Use the value function indirectly (e.g., take the action with highest Q-value)

Characteristics:

- Policy is derived from value estimates
- No explicit policy model

12.2 Policy-Based Agents

These agents learn the **policy directly**.

Examples:

- REINFORCE
- Some Policy Gradient methods

Characteristics:

- No value function is learned
- Useful for continuous action spaces
- Policy often represented as a probability distribution (stochastic)

12.3 Actor–Critic Agents

These agents combine the best of both worlds:

- **Actor**: learns the policy
- **Critic**: learns the value function
- Policy improves using feedback from the critic's value estimates

Examples:

- A2C (Advantage Actor-Critic)
- A3C
- PPO

- DDPG

Characteristics:

- Stable learning
- Handles large or continuous action spaces

12.4 Model-Free Agents

These agents **do not learn a model** of the environment.

Examples:

- Q-learning
- Deep Q Networks (DQN)
- Monte Carlo RL

Characteristics:

- Use trial-and-error
- Learn from direct interaction
- Cannot simulate future scenarios internally

12.5 Model-Based Agents

These agents learn a **model** of the environment:

- Transition model: what happens after an action
- Reward model: expected reward

They can **plan** using the learned model.

Examples:

- Dyna-Q
- Model-based planning algorithms
- Integrated architectures like MuZero

Characteristics:

- More sample-efficient
- Harder to implement in complex environments

12.6 Summary Table

Agent Type	Policy	Value Function	Model
Value-Based	Derived indirectly	Yes	No
Policy-Based	Yes	No	No
Actor–Critic	Yes (Actor)	Yes (Critic)	No
Model-Free	Optional	Optional	No
Model-Based	Optional	Optional	Yes

✅ Section 13 — Learning vs Planning

Reinforcement Learning involves two major problem types:

Learning (from experience) and **Planning** (using a model).

Both aim to improve decision-making policies, but they operate differently.

13.1 Learning

Learning happens when:

- The **environment is initially unknown**
- The agent must **interact** with the environment to gather experience
- The agent improves its policy based on actual feedback (rewards, transitions)

Key Characteristics:

- Trial-and-error based
- Relies on real experience
- Often slower but does not require prior knowledge
- Used in model-free RL

Example:

An Atari agent starts with no knowledge of the game and learns only by playing.

13.2 Planning

Planning happens when:

- A **model of the environment is available**
- The agent can **simulate** how actions affect states and rewards
- No real-world interaction is necessary to update the policy

Key Characteristics:

- Uses internal computations
- Agent queries the model:
  - "If I take action *a* from state *s*, what will happen?"
- Faster and more efficient
- Used in model-based RL

Planning Example:

Chess engines simulate thousands of hypothetical moves ahead to choose the best action.

13.3 Learning vs Planning: Key Differences

Feature	Learning	Planning
Requires environment model?	❌ No	✔ Yes
Requires real-world interaction?	✔ Yes	❌ No (simulated)
Data source	Experience	Model queries
Sample efficiency	Lower	Higher
Flexibility in unknown environments	High	Low unless model is built

---

13.4 How They Work Together

Modern RL systems often combine both:

(A) Learn the model

→ From interaction data

(B) Plan using the model

→ To reduce interaction cost

This hybrid approach leads to methods like:

- **Dyna-Q**
- **MuZero** (learns representation + dynamics + value)

13.5 Summary

- **Learning**: improves policy using experience
- **Planning**: improves policy using simulated experience
- RL agents may use either or both depending on the problem

✅ Section 14 — RL with Unknown Environment & Exploration

When the environment is **unknown**, the agent must learn entirely through interaction.

This section focuses on how agents handle uncertainty and discover good strategies.

14.1 Unknown Environment

In many real-world RL tasks:

- The **rules** of the environment are not known beforehand
- The agent must learn purely from **experience**

- It receives only observations, actions, and rewards

Examples:

- Video games (pixels + score only)
- Robot navigation in a new building
- Stock market trading

The agent must figure out:

- Which actions cause which outcomes
- Which trajectories lead to higher reward
- How to balance exploration vs exploitation

14.2 Interactive Gameplay

Learning directly from gameplay involves:

- **Picking actions** using buttons or controls (N, E, W, S, Fire, Jump, etc.)
- Receiving **pixel observations** as visual input
- Getting **scores or rewards**

The agent gradually builds:

- A **value function** for good/bad states
- A **policy** for choosing actions

This is how classic Deep RL systems like **DQN** learned Atari.

14.3 Exploration is Necessary

In unknown environments, the agent must sometimes:

- Try new actions
- Venture into unseen states
- Risk lower rewards temporarily to discover better strategies

If the agent never explores:

- It may get stuck in **suboptimal** behavior
- It never discovers better paths

If it explores too much:

- It wastes time and rewards
- Learning becomes unstable

Thus, RL requires a **balance**.

14.4 Why Unknown Environments Are Challenging

- No prior knowledge means **everything** must be learned
- Rewards may be delayed
- Explorations may be costly
- Harder to establish which actions truly cause success

Example:

A robot exploring a new room may waste battery or even crash before learning.

14.5 Key Idea

RL agents in unknown environments:

- Learn **directly from experience**
- Must manage **trial-and-error**
- Improve the policy gradually
- Often rely on stochastic strategies (ε-greedy, softmax, etc.)

This is the basis of **model-free RL**, which dominates practical applications like games and robotics.

✅ Section 15 — Planning (Rules Known)

Planning is the counterpart to learning in RL.

Here, the agent **already knows the environment's rules**, so it can think ahead without real-world interaction.

15.1 When Planning is Possible

Planning is used when:

- The **transition rules** (what happens after each action) are known
- The **reward structure** is known
- The agent can simulate outcomes internally

Examples:

- Chess, checkers, Go
- Grid-world problems with full knowledge
- Robotics simulators
- Planning with a learned model (e.g., MuZero)

15.2 How Planning Works

The agent builds an **internal emulator** of the environment.

It can then ask:

- "If I take action a from state s, what will be the next state s'?"
- "What reward will I get?"

This lets the agent evaluate many possibilities **without actually performing them**.

15.3 Benefits of Planning

1. **No Real-World Interaction Required**

- Cheaper
- Faster
- No risk (unlike crashing a real robot)

2. **Search-Based Optimization**

- Explore many trajectories in simulation
- Identify the best action sequence
- Works well with tree search (e.g., Minimax, MCTS)

3. **Highly Sample-Efficient**

- One model can generate unlimited “imagined” data
- Useful when real interaction is expensive

15.4 Example: Chess Engine

A chess engine with known rules can:

- Simulate moves several steps ahead
- Evaluate board positions using heuristics or value functions
- Choose actions purely via internal computation

No learning from environment interaction is needed.

15.5 Planning vs Learning

Planning uses:

- **Predictions** from a model
- **Simulations** to improve strategy
- No real-time experience needed

Learning uses:

- Interactions and rewards from actual environment
- Trial-and-error
- Experience to improve the policy

Modern RL systems often combine them:

- Learn a model

- Use the model to plan
- Update the policy using simulated data

This hybrid approach is extremely powerful (e.g., AlphaZero, MuZero).

✅ Section 16 — Exploration vs Exploitation

A core challenge in RL is balancing **exploration** (trying new actions) with **exploitation** (using known good actions).

This is one of the most important concepts in reinforcement learning.

16.1 Why Exploration is Needed

If the agent only exploits (uses known best actions):

- It may miss better strategies
- It may get stuck in **local optima**
- Its behaviour may remain suboptimal forever

Example:

A robot knows one safe route but could discover a shorter path by exploring.

16.2 Why Exploitation is Needed

If the agent only explores:

- It wastes time and energy
- Rewards will be low
- Behaviour becomes random and unstable

Example:

A robot may endlessly try new routes and never reliably reach the goal.

16.3 The Trade-Off

The agent must balance:

- **Exploitation** → maximize reward **now**
- **Exploration** → gather information to maximize reward **later**

This is the **exploration–exploitation dilemma**.

16.4 How RL Handles This Trade-Off

1. ε-Greedy Exploration

- With probability **1 – ε**, choose the best-known action (exploit)
- With probability **ε**, choose a random action (explore)

2. Softmax/Boltzmann Exploration

- Assign probabilities to actions based on their estimated value
- Higher-value actions chosen more often, but lower-value actions still tried

3. Decaying Exploration

Start with high exploration, gradually reduce it as the agent learns more.

4. Intrinsic Motivation / Curiosity

Give a small "bonus" reward for visiting new or surprising states.

5. Upper Confidence Bound (UCB)

Pick actions that balance:

- Estimated reward
- Uncertainty in the estimate

16.5 Real-World Example

Robot Walking:

- **Exploitation**: Use known safe paths to conserve energy
- **Exploration**: Try new routes that might be faster or safer long-term

Without exploration → stuck in old habits

Without exploitation → never stabilize behaviour

16.6 Key Insight

Exploration leads to knowledge.
Exploitation uses that knowledge.

Both are necessary for successful RL.
