Module 4: Reinforcement Learning

Learning from experience — trial, error, and reward

PECST522 — Artificial Intelligence | B.Tech CSE S5 (2024 Scheme)

Module 4 — Roadmap

10 contact hours | Maps to CO4 | Difficulty: ☆☆☆

4.1	**The Setting**	Learning from rewards — motivation, credit assignment
4.2	**MDPs**	States, actions, transitions, rewards, discount factor
4.3	**Passive RL**	Evaluate a fixed policy: DUE, ADP, TD learning
4.4	**Active RL**	Choose actions — exploration vs exploitation, Q-learning
4.5	**Generalization**	Function approximation, deep Q-networks
4.6	**Policy Search**	Directly optimise the policy — policy gradient
4.7	**Inverse RL**	Learn the reward from an expert demonstration
4.8	**Applications**	AlphaGo, DQN, robotics, ChatGPT / RLHF

4.1

The Setting: Learning from Rewards

No teacher, no labels — just trial, error, and a reward signal

Why Do We Need Reinforcement Learning?

🔍 Module 2 — Search

Needed to KNOW the state space in advance

🧠 Module 3 — Reasoning

Needed KNOWLEDGE in the Knowledge Base

🎮 Module 4 — RL

Agent knows NEITHER — must figure things out by trying

💡 Analogy

You're placed in a kitchen in a foreign country. No recipe book, no labels on jars, no one to ask. You try cooking with different ingredients, taste the result (reward), and over time figure out what makes good food. That's RL.

No teacher, no correct answers — just a REWARD signal after you act, and you must figure out what worked.

Three Types of Machine Learning

Type	Teacher?	Feedback	Example
Supervised	Yes — gives correct answers	"This image = cat" (labelled data)	Photo recognition, spam filter
Unsupervised	No teacher, no labels	None — find patterns yourself	Clustering customers, topic discovery
Reinforcement	No correct answers	Delayed REWARD signal (good / bad)	Game playing, robot walking

Key difference:
In supervised learning, someone tells you the right answer for each input. In RL, nobody tells you — you just get a score (reward) AFTER you act, and must figure out which actions deserved the credit.

 Supervised = teacher grading each answer. RL = getting your semester GPA and figuring out which study habits worked.

The Credit-Assignment Problem

When you get a reward AFTER many actions, which action actually EARNED it?

Example — a sequence of moves in a game:

? Which move deserved the credit? Move C? Move A (set it up)? All of them? How do we split the reward?

Why is this hard?

Delayed reward	Outcome comes many steps AFTER the actions that caused it
No labelled data	Nobody tells you "this action was good, that one was bad"
Stochastic environment	Same action can lead to different outcomes — luck vs skill?

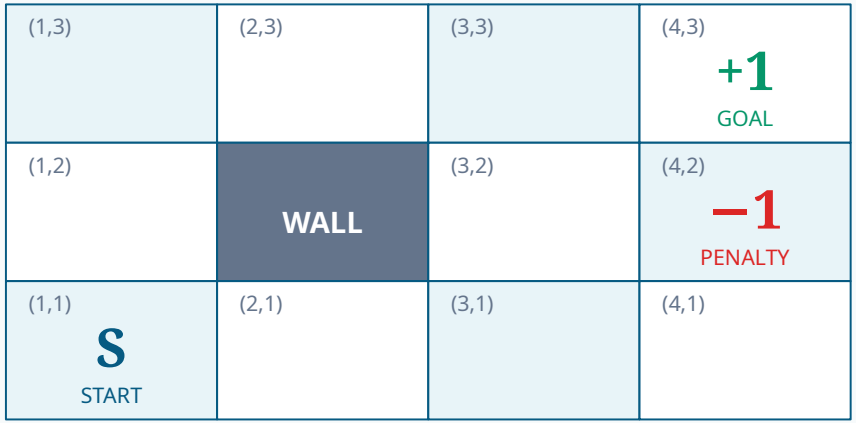
4.2

Markov Decision Processes

The mathematical framework that RL operates in

The Gridworld — Our Running Example

A 4×3 grid with stochastic actions



Rules:

Agent starts at (1,1)

Actions: Up, Down, Left, Right

Actions are STOCHASTIC:

- 0.8 — go where intended
- 0.1 — slip left (perpendicular)
- 0.1 — slip right (perpendicular)

Hit a wall → stay in place

(4,3) and (4,2) are TERMINAL states

All other squares: reward = −0.04 / step

⚡ EXAM ALERT: Transition model — 0.8 intended direction, 0.1 each perpendicular. This stochasticity is what makes RL interesting!

Markov Decision Process — Formal Definition

An MDP = ⟨ S, A, P(s'|s,a), R(s), γ ⟩

Component	Symbol	Gridworld Example
States	S	All 11 non-wall grid squares
Actions	A	{ Up, Down, Left, Right }
Transition model	P(s'|s,a)	80% intended direction, 10% each side
Reward function	R(s)	+1 at (4,3), −1 at (4,2), −0.04 elsewhere
Discount factor	γ	e.g., 0.9

Markov Property

The next state depends ONLY on the current state and action — NOT on how you got here. History doesn't matter; only where you are NOW.

 Like a board game — it doesn't matter how pieces got to their positions. What matters is where they ARE now and what you do next.

Key Quantities — Policy, Utility, Discount Factor

⚡ EXAM ALERT — the three quantities you must define

1. Policy π(s)

A rule that tells the agent what action to take in each state. Example: π(1,1) = Right, π(3,1) = Up.

The GOAL of RL is to find the optimal policy π* — maximises total expected reward.

2. Utility U(s) (also called Value V(s))

Expected total FUTURE reward starting from state s, following a particular policy.

High U(s) = great to be in state s. Low U(s) = bad place to be.

3. Discount factor γ (0 ≤ γ ≤ 1)

Makes future rewards worth LESS than immediate rewards. Total = $R_0 + \gamma R_1 + \gamma^2 R_2 + \gamma^3 R_3 + \ldots$

What γ controls: patience. Higher γ → more far-sighted. Lower γ → more greedy / short-sighted.

γ value	Agent's attitude	Analogy
γ → 1	Very far-sighted (future matters a lot)	Long-term investor
γ → 0	Very short-sighted (only NOW matters)	Live for today
γ = 0	Completely greedy	Only cares about next reward

 ₹100 today vs ₹100 next year? TODAY — because of inflation, risk, opportunity cost. That's what γ does: future rewards are worth less.

4.2 (contd.)

The Bellman Equation

The foundation of all reinforcement learning — relates the value of a state to the values of its neighbours

The Bellman Equation

The foundation of all reinforcement learning

$$U(s) = R(s) + \gamma \cdot \max_a \Sigma_{s'} P(s'|s,a) \cdot U(s')$$

In English: "The value of HERE = the reward I get NOW + the discounted value of the BEST future I could reach."

Symbol	Meaning	Gridworld Example
U(s)	Utility (value) of state s — expected total future reward from here	How good is it to be at (3,1)?
R(s)	Immediate reward received at state s	−0.04 for most squares (cost of living)
γ	Discount factor (0 ≤ γ ≤ 1) — patience; how much future matters	γ = 0.9 → future matters a lot
max_a	Choose the action a that maximises the expected outcome	At (3,1): try Up, Down, Left, Right → pick best
Σ_s' P(s'|s,a)·U(s')	Expected value of next state, averaged over all possible outcomes	0.8×U(intended) + 0.1×U(slip left) + 0.1×U(slip right)

⚡ EXAM ALERT: The Bellman equation is the foundation of ALL RL algorithms. Memorise it — Part B questions ask you to write and explain it.

Bellman Equation — Worked Example

Numeric backup on the 4×3 gridworld

Setup: Agent at (3,1). Actions are stochastic: 0.8 intended, 0.1 each perpendicular. γ = 0.9. R(s) = −0.04 for all non-terminal states.

Question: What is U(3,1) if we choose action Up? Compute the Bellman backup.

Step 1: Identify transition outcomes for action Up from (3,1)

Direction	Prob	Destination	Note
Up (intended)	**0.8**	(3,2)	Agent goes up as planned
Left (slip)	**0.1**	(2,1)	Agent slips left
Right (slip)	**0.1**	(4,1)	Agent slips right → terminal! R = −1

Step 2: Plug into Bellman equation for action Up

$$Q(\text{Up}) = R(3,1) + \gamma \times [\, 0.8 \cdot U(3,2) + 0.1 \cdot U(2,1) + 0.1 \cdot U(4,1) \,]$$
$$= -0.04 + 0.9 \times [\, 0.8 \cdot U(3,2) + 0.1 \cdot U(2,1) + 0.1 \cdot (-1) \,]$$

✓ **Key Insight:** The Bellman backup for ONE action gives the value of choosing that action. The agent picks the action with the MAX value. To get the full U(3,1), we'd compute this for all 4 actions and take the max. The actual U values are found by solving all equations simultaneously (value iteration).

⚡ **EXAM ALERT: You may be asked to write the Bellman backup for a specific state-action pair — show the weighted sum over all transition outcomes.**

4.3

Passive Reinforcement Learning

The agent is given a fixed policy π — its only job is to evaluate how good that policy is (learn U^π)

Passive RL — Setup & Overview

Given a fixed policy π, learn the utility function U^π(s)

Setup: Someone GIVES the agent a fixed policy π (a fixed way to behave). The agent's ONLY job is to figure out how GOOD that policy is — learn $U_\pi(s)$ for each state.

Analogy: *Your boss gives you a fixed delivery route. You don't change the route — you just figure out "how good IS this route?" by driving it many times.*

Three methods to learn U^π — compared:

Method	Model needed?	Updates when?	Speed	Memory
DUE (Direct Utility Estimation)	No	After full episode	Slow (ignores structure)	Low
ADP (Adaptive Dynamic Programming)	Yes (learns P and R)	Each step, solves equations	Fast learning, expensive compute	High
TD (Temporal-Difference)	No	Each step, simple update	Good balance	Low

Analogy: DUE = rate the restaurant after eating the full meal (waits for the end). ADP = learn the recipe and predict how it'll taste (builds a model). TD = adjust your expectations after every bite (updates incrementally).

Direct Utility Estimation (DUE)

Run the policy many times; average the observed returns

How DUE works:

1. Follow policy π for many episodes (start → terminal state)
2. For each state visited, record the total discounted reward from that state to the end
3. Average these values across many episodes → that's U(s)

Example — one episode in gridworld:

Episode: (1,1)→(1,2)→(1,3)→(2,3)→(3,3)→(4,3) [+1]
Rewards: −0.04, −0.04, −0.04, −0.04, −0.04, +1

With γ = 1 (no discount, for simplicity):
U(1,1) sample = −0.04×5 + 1 = +0.80
U(1,2) sample = −0.04×4 + 1 = +0.84

Pros & Cons:

✓ Pros
- Very simple to implement
- No model of the environment needed
- Guaranteed to converge with enough episodes

✗ Cons
- SLOW — needs many episodes to converge
- Ignores MDP structure: treats each state independently, even though neighbouring states' utilities are RELATED

Key weakness: DUE computes each U(s) independently from raw experience. It doesn't exploit the fact that U(1,1) and U(1,2) are related through the MDP transitions.

⚡ DUE in one line: Follow the policy, record what happens, average the returns. Simple but slow — ignores the structure of the MDP.

Adaptive Dynamic Programming (ADP)

Learn the model P and R from observations, then solve the Bellman equations

How ADP works:

1. While following policy π, COUNT transitions: "from state s with action a, I ended up in s' this many times"
2. Estimate P(s'|s,a) = (count of s→s' with a) / (total times did a in s)
3. Estimate R(s) from observed rewards
4. Solve the Bellman equations (system of linear equations) for U(s)

Learning the transition model — example:

From (3,1), did action Up 100 times:

→ ended at (3,2): 78 times → P = 0.78
→ ended at (2,1): 12 times → P = 0.12
→ ended at (4,1): 10 times → P = 0.10

Pros & Cons:

✓ Pros

- Makes FULL use of MDP structure
- Learns fast (fewer episodes needed)
- Uses transition relationships between states
- Solves for ALL U(s) at once (exact solution)

✗ Cons

- Computationally EXPENSIVE for large state spaces (must solve system of equations)
- Requires learning AND storing the full transition model P(s'|s,a)

ADP vs DUE: ADP exploits the MDP structure to learn faster, but pays for it in computation. DUE is simpler but wastes structural information.

⚡ ADP in one line: Learn the model (P and R) from experience, then solve the Bellman equations exactly. Fast learning but expensive computation.

⚡ Temporal-Difference (TD) Learning

THE central idea of reinforcement learning — update after every step, no model needed

$$U(s) \leftarrow U(s) + \alpha \cdot [\, R(s) + \gamma \cdot U(s') - U(s) \,]$$

TD error (δ) — **the SURPRISE**

Symbol	Meaning	Analogy
$U(s)$	Current estimate of state s's value	What I THINK this state is worth
α	Learning rate ($0 < \alpha \leq 1$) — how much to trust new experience	How much I adjust my belief
$R(s)$	Reward received at state s	Cash I just got
γ	Discount factor — patience	How much I care about the future
$U(s')$	Current estimate of NEXT state's value	What I THINK the next state is worth
$R(s) + \gamma \cdot U(s')$	TD target — what experience suggests s is worth	Observed evidence about s

What does the TD error (δ) tell us?

- **$\delta > 0$** → Reality was BETTER than expected → increase U(s)
- **$\delta < 0$** → Reality was WORSE than expected → decrease U(s)
- **$\delta = 0$** → No surprise — no update needed

In plain English: "Adjust my estimate of HERE to look more like (what I got) + (discounted value of where I ended up)."

⚡ TD Learning — Worked Example

Numeric update on the 4×3 gridworld

Scenario: Agent at (3,1), takes action Up → arrives at (3,2).

γ = 0.9, α = 0.1

Current estimates: **U(3,1) = 0.5, U(3,2) = 0.7** Reward: **R(3,1) = −0.04**

Step-by-step trace:

Step 1: Compute the TD Target

TD Target = R(s) + γ · U(s')

= −0.04 + 0.9 × 0.7

= −0.04 + 0.63 = 0.59

Step 2: Compute the TD Error (δ)

δ = TD Target − U(s)

= 0.59 − 0.5

= 0.09 (reality slightly better than expected)

Step 3: Update U(3,1)

U(3,1) ← U(3,1) + α · δ

= 0.5 + 0.1 × 0.09

= 0.5 + 0.009 = 0.509

RESULT: New U(3,1) = 0.509 (slightly increased — that state is a bit better than we thought)

⚡ EXAM ALERT: TD learning is THE central idea of RL. The update rule and a numeric trace like this are guaranteed Part B questions.

DUE vs ADP vs TD — Passive RL Comparison

Three ways to evaluate a fixed policy — when to use each?

Method	Model needed?	Updates	Speed	Memory
DUE	No	After full episode	Slow (ignores structure)	Low
ADP	Yes (learns it)	Each step, solves equations	Fast learning, expensive compute	High
TD	No	Each step, simple update	Good balance	Low

🍽 Restaurant Analogy

DUE = Rate the restaurant after eating the FULL meal (wait for the end).
ADP = Learn the recipe and predict how it will taste (build a model).
TD = Adjust your expectations after EVERY bite (updates incrementally).

🔑 Key Takeaway

TD is the most popular passive method — no model needed, updates online after each step, and converges to the true utility. It is the foundation on which Q-learning (active RL) is built.

4.4

Active Reinforcement Learning

The agent must now CHOOSE actions — not just evaluate a given policy.

Active vs Passive Reinforcement Learning

From evaluating a fixed policy to finding the best policy

Passive RL

- Agent is GIVEN a fixed policy π
- Task: evaluate how good π is
- Learn U^π(s) for each state
- Agent has NO control over actions
- Like: driving a fixed delivery route and rating how good it is

Active RL

- Agent must CHOOSE actions itself
- Task: FIND the best policy π*
- Must balance learning vs earning
- Agent controls which states it visits
- Like: exploring a city to find the best restaurant on your own

⚡ The New Challenge

In passive RL, the policy decides which states you visit. In active RL, YOUR actions decide which states you visit. This creates a fundamental tension: do you do what you THINK is best, or try something new to LEARN what's best?

Coming up...

1. Exploration vs Exploitation — the core dilemma of active RL
2. ε-Greedy — a simple strategy to balance both
3. Q-Learning — the flagship algorithm that solves this (model-free, off-policy)

Exploration vs Exploitation

⚡ The fundamental dilemma of active reinforcement learning

	Exploitation	Exploration
What	Do the action currently believed to be best	Try something NEW
Benefit	Cash in on what you know	Discover possibly better actions
Risk	Miss better options forever	Waste time on bad actions

🍽 The Restaurant Problem

You know a restaurant you rate 7/10. A new restaurant opened next door.

- Exploit → go to your known 7/10 (safe, but you might miss a 9/10)
- Explore → try the new one (risky — could be 3/10 or 9/10)

If you always exploit, you never discover the 9/10 restaurant.
If you always explore, you eat at terrible places half the time.

✅ Solution: ε-Greedy — exploit most of the time, explore occasionally (next slide →)

ε-Greedy Exploration Strategy

A simple way to balance exploration and exploitation

The ε-Greedy Rule

With probability (1 − ε): do the BEST known action (exploit)
With probability ε: do a RANDOM action (explore)

Choosing ε

- ε = 0.1 → explore 10%, exploit 90%
- ε = 0.5 → explore 50%, exploit 50%
- ε = 1.0 → explore 100% (pure random)
- ε → 0 → pure exploitation

Decaying ε over time

- Start with high ε (lots of exploration)
- Gradually decrease ε over episodes
- Early: explore broadly to gather info
- Later: exploit what you've learned
- GLIE: ε → 0 slowly enough that every state-action is visited infinitely often

⚡ Why not always explore?

Pure exploration never converges to a good policy — you keep trying random things without using what you've learned. Pure exploitation can get stuck on a suboptimal action. ε-greedy balances both, and decaying ε guarantees convergence to the optimal policy (GLIE condition).

⚡ EXAM ALERT — "Explain exploration vs exploitation" and "What is ε-greedy?" are frequent Part A questions.

Q-Learning — Why Q(s,a) Instead of U(s)?

⚡⚡⚡ The flagship algorithm — model-free action selection

	U(s) — State Value	Q(s,a) — Action Value
What it stores	Value of a STATE	Value of a STATE-ACTION pair
To pick an action	Need transition model P(s'|s,a) to know which action leads where	Just pick argmax_a Q(s,a) — NO model needed!
Model-free?	NO — needs P(s'|s,a)	YES — learns from experience alone

The Key Insight

With U(s), to choose the best action you need to know the transition model P(s'|s,a) — you must simulate "if I take action a, where do I end up, and how good is that state?"

With Q(s,a), you already know the value of EACH action — just pick the one with the highest Q. No model needed. This is what makes Q-learning MODEL-FREE.

⚡ EXAM ALERT

"Why is Q(s,a) more useful than U(s) for choosing actions?" — frequent Part B question.
Answer: Q(s,a) lets you pick the best action directly (argmax) without needing the transition model P(s'|s,a).

Q-Learning — The Update Rule

⚡⚡⚡ MEMORISE THIS — the most important formula in Module 4

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

TD error for Q

Term	Meaning
Q(s,a)	Current estimate: "how good is doing action a in state s?"
α (learning rate)	How much to trust new experience vs old belief (0 < α ≤ 1)
R(s)	Reward just received at state s
γ · max_a' Q(s',a')	Discounted value of the BEST action from next state s'
max_a' Q(s',a')	Best achievable utility from the next state — this is what makes Q-learning learn the OPTIMAL policy even while exploring
[R(s) + γ·max Q − Q(s,a)]	The TD error δ — the SURPRISE: was reality better or worse than expected?

In Plain English

"Adjust my estimate of (state, action) to look more like: (what I just got) + (discounted value of the BEST I could do next)." The max operator is what makes Q-learning converge to the OPTIMAL policy — it learns what's best, not what you actually did.

Q-Learning — Worked Example

⚡ Watch the reward propagate backward through the chain!

Setup

A → B → C (goal, reward +10)
All other rewards = 0
γ = 0.9, α = 0.5
All Q-values initialized to 0.

Episode 1

Step 1: A → B, R = 0
$Q(A,R) \leftarrow 0 + 0.5 \times [0 + 0.9 \times 0 - 0]$
= 0 (nothing learned yet)

Step 2: B → C, R = +10

Episode 2

Step 1: A → B, R = 0
$Q(A,R) \leftarrow 0 + 0.5 \times [0 + 0.9 \times \max Q(B,*) - 0]$
max Q(B,*) = Q(B,R) = 5.0 (from Episode 1!)
= 0 + 0.5 × [0 + 0.9 × 5.0] = 0.5 × 4.5 = **2.25** ✓ A now knows it leads to good things!

Step 2: B → C, R = +10
$Q(B,R) \leftarrow 5.0 + 0.5 \times [10 + 0.9 \times 0 - 5.0]$
= 5.0 + 0.5 × 5.0 = **7.5** (closer to true value)

🔑 See What Happened?

The reward at C propagated BACKWARD through the chain:
• Episode 1: B learned it was good (Q = 5.0)
• Episode 2: A learned it was good (Q = 2.25) because A leads to B

This is how RL solves the credit-assignment problem — value backs up one step at a time.

Q-Learning — Key Properties

⚡⚡⚡ Why Q-learning became the most popular RL algorithm

Model-Free

Does NOT need to know P(s'|s,a) or R(s) in advance.

Learns everything from raw experience: just (s, a, r, s') tuples.

Off-Policy

Can learn the OPTIMAL policy even while BEHAVING RANDOMLY (exploring).

The max_a' in the update uses the best action, not the one actually taken.

Converges to Q*

Given enough exploration (GLIE), Q-learning converges to the optimal Q* table.

The optimal policy is then: $\pi^*(s) = \text{argmax}_a Q^*(s,a)$.

The Table Problem

Q-learning stores a Q-value for EVERY (state, action) pair. This works for tiny worlds:

- Gridworld (4×3): 11 states × 4 actions = 44 entries — easy ✓
- Backgammon: ~10^{20} states — impossible to store or visit them all ✗
- Self-driving: continuous (infinite) state space — completely infeasible ✗

What's the Solution? → Section 4.5: Generalization

Instead of a table, approximate Q(s,a) with a FUNCTION (linear or neural network).
Use FEATURES of the state so the agent can generalise to states it has NEVER visited.
This is the idea behind Deep Q-Networks (DQN) and modern Deep RL.

4.5

Generalization in RL

From tables to functions — scaling RL to real-world problems

Why Generalization?

The Q-table blows up — you cannot visit every state

State-Space Explosion

Domain	States	Q-table feasible?
Gridworld (4×3)	11	Yes ✓
Backgammon	~10^{20}	**No way ✗**
Chess	~10^{40}	**Absolutely not ✗**
Self-driving car	Continuous (infinite)	**Impossible ✗**

⚠ The Core Problem

- Even if you COULD store 10^{20} entries, you would need to VISIT every state to fill the table.
- With 10^{20} states, that's impossible in a lifetime.
- You must generalise from states you HAVE visited to states you HAVE NOT.

The Solution: Function Approximation

Instead of a table, represent Q(s,a) as a parameterised function of features.
Learn the parameters from experience — then the function can estimate values for unseen states.

Function Approximation

Q(s,a) ≈ f(features, weights) instead of a table entry

Linear Approximation

$$Q(s, a) \approx w_1 \cdot f_1(s,a) + w_2 \cdot f_2(s,a) + \ldots + w_n \cdot f_n(s,a)$$

f_1, f_2, … = FEATURES of the state (things you can measure)
w_1, w_2, … = WEIGHTS you learn from experience

Example: Pac-Man Features

f_1	distance to nearest ghost	w = +2
f_2	distance to nearest food	w = −1
f_3	ghosts within 3 squares	w = −3

Learned weights make the function generalise to new mazes.

Neural Network Approximation

- Replace linear function with a neural network
- Input = state features
- Output = Q-values for all actions
- This is Deep RL: deep neural nets + RL
- DQN (Deep Q-Network) — played Atari games (2015, DeepMind)

One-liner

Generalization = use a function (linear or neural net) instead of a table, so the agent can make good guesses about states it has NEVER visited.

Deep Q-Network (DQN)

Neural net as function approximator — the algorithm that played Atari

What is DQN?

- A Q-learning agent where the Q-table is replaced by a deep neural network
- Input: raw state (e.g. pixel frames); Output: Q-value for each action
- Trained with the Q-learning update rule, using gradient descent on the TD error
- Result (2015, DeepMind): superhuman performance on many Atari games

① Experience Replay

- Store transitions (s, a, r, s') in a replay buffer
- Sample random mini-batches for training
- Breaks correlation between consecutive samples
- Makes training stable (otherwise the net chases its own moving predictions)

② Target Network

- Use a separate (frozen) copy of the network to compute the TD target
- Update the target network only periodically
- Without it: the target shifts with every update → training oscillates and diverges
- With it: stable convergence to good Q-values

Connection to Q-learning

DQN still uses the Q-learning update — but instead of looking up Q(s,a) in a table, it queries the neural network. The "deep" refers to the deep neural net; the RL idea is the same Q-learning.

4.6

Policy Search

Skip values — directly optimise the policy itself

Policy Search

Directly learn π — optimise the policy, not the value function

The Idea

1. Represent the policy as a parameterised function: π_θ(s) = action (θ = learnable parameters)
2. Try the policy in the environment, observe total reward
3. Adjust θ in the direction that INCREASES total reward
4. This is called policy gradient — literally gradient ascent on total reward

When Policy Search Wins

- Continuous actions (robot arm angles — can't "max over all actions" with infinite choices)
- Value function is complex but policy is simple
- Stochastic policies (sometimes randomness helps)

Actor-Critic Methods

Combine both approaches:
- Actor = the policy (chooses actions)
- Critic = the value function (evaluates)
- Critic evaluates, actor improves
- Best of both worlds

Analogy

Value-based = learn which restaurants are good (values), then always go to the best one (derive policy).
Policy search = directly learn the rule "always pick the one with the shortest queue" (learn the policy directly, never actually rating restaurants).

Inverse Reinforcement Learning (IRL)

Rewards are hard to hand-specify — learn the reward function by observing an expert

⚠ The Problem: Rewards Are Hard to Define

- How do you define a reward for "drive safely and comfortably"?
- How do you reward "write a good essay" or "be a helpful assistant"?
- For many real tasks, writing a reward function is extremely difficult.

Apprenticeship / Imitation Learning

- Watch an expert performing the task
- Learn to COPY their behaviour
- Simple but limited:
  - Cannot do better than the expert
  - Cannot adapt to new situations

Inverse RL (IRL)

- Watch the expert
- Infer the REWARD FUNCTION the expert seems to be optimising
- Then use standard RL with that reward
- Much more powerful: once you know the reward, you can do BETTER than the expert

Analogy

Imitation = watch a chef cook and copy every move exactly.
IRL = watch the chef, figure out "oh, they're optimising for FLAVOUR, not presentation" (recover the reward), then optimise for flavour yourself — perhaps finding even better recipes.

Real-world use cases:

Self-driving cars (learn from human driving data) • Robots (learn tasks by watching demonstrations) • ChatGPT (RLHF — humans rate outputs, system learns a reward model)

Applications of Reinforcement Learning

From games to robotics to language models

Application	What RL does	Key technique
AlphaGo (2016)	Beat world Go champion	Deep RL + MCTS
Atari games (DQN, 2015)	Learned to play from raw pixels	Deep Q-Network
Robot locomotion	Learned to walk, run, jump	Policy gradient
Data center cooling (Google)	Reduced cooling energy by 40%	Model-based RL
ChatGPT / LLMs	Aligned with human preferences	RLHF
Autonomous driving	Lane keeping, decision making	Deep RL + simulation
Recommendation systems	Netflix, YouTube, Amazon	Contextual bandits
Drug discovery	Molecule design	RL + graph neural nets
Finance	Portfolio management, trading	Q-learning + function approx.

 AlphaGo combines tree search (Module 2) with deep RL (Module 4) — the course concepts connect to real breakthroughs.

Summary & Exam Cheat Sheet

Compact master reference — formulas + comparison table

⚡ Three Must-Memorise Formulas

Bellman: $U(s) = R(s) + \gamma \cdot \max_a \Sigma_{s'} P(s'|s,a) \cdot U(s')$

TD: $U(s) \leftarrow U(s) + \alpha\,[\,R(s) + \gamma\,U(s') - U(s)\,]$

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

γ = discount factor α = learning rate R(s) = reward U(s) = utility of state Q(s,a) = value of action a in state s

Master Comparison: Passive vs Active × Model-free vs Model-based

Method	Passive/Active	Model-free/based	Key Idea
DUE	Passive	Model-free	Average returns over full episodes
ADP	Passive	Model-based	Learn P and R, solve Bellman exactly
TD	Passive	Model-free	Update after each step: U ← U + α·δ
Q-learning	Active	Model-free	Learn Q(s,a); pick argmax_a Q(s,a)

Key Definitions

- Policy π(s): rule telling agent what action to take in each state. Goal: find optimal policy π*.
- Utility U(s): expected total future reward from state s. High U = great state to be in.
- Discount factor γ (0≤γ≤1): how much future rewards matter vs immediate rewards. γ→1 = far-sighted; γ→0 = greedy.
- TD error δ = R(s) + γ·U(s') − U(s): the SURPRISE — was reality better or worse than expected?
- Exploration vs Exploitation: try new actions (explore) vs do best known action (exploit). ε-greedy balances both.

The Complete Picture

Module 1: DEFINE the problem → Module 2: SEARCH for solutions → Module 3: REASON with knowledge → Module 4: LEARN from experience

Self-Check Questions

⚡ = Exam hot topics (Part A: 3 marks | Part B: 9 marks)

1. Contrast supervised, unsupervised, and reinforcement learning with one example each.
2. What is the credit-assignment problem? Why does delayed reward cause it?
3. **Define: MDP, policy, utility, discount factor. What does γ control?** ⚡
4. **Explain the Bellman equation in your own words.** ⚡
5. What does "passive" RL mean? Compare DUE, ADP, TD in a table.
6. **Write the TD update rule. Explain each term. What does a positive TD error mean?** ⚡
7. **TD update: U(s)=2.0, R=1, U(s')=3.0, γ=0.9, α=0.1 → target 3.7, error 1.7, U ← 2.17** ⚡
8. **Explain exploration vs exploitation with an analogy.** ⚡
9. **Write the Q-learning update rule. Why is Q(s,a) more useful than U(s) for choosing actions?** ⚡
10. Why do we need generalization? What is function approximation?
11. How does policy search differ from value-based RL?
12. What is IRL? When is it more useful than standard RL?
13. Name 3 real-world applications of RL and the technique used.

Module 4 Complete — You now have the full AI toolkit: Search + Reason + Learn