Lecture 1: Foundations — From Dynamic Programming to Deep Q-Learning

Wouter van Heeswijk

University of Twente
w.j.a.vanheeswijk@utwente.nl

22 July 2025

Outline

What is Reinforcement Learning?

- Reinforcement Learning (RL) is about learning to make decisions through **interaction with an environment**.
- An agent observes the state, takes an action, receives a reward, and transitions to a new state.
- Objective: learn a policy to maximize long-term reward.

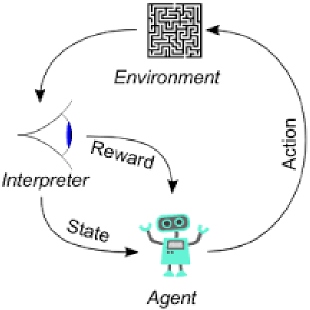


Figure: https://commons.wikimedia.org/wiki/File:Reinforcement_learning_diagram.svg

Key Ingredients of RL

- **Agent**: Learns and acts
- **Environment**: Dynamics + rewards
- **Policy** $\pi(a|s)$: Maps state to action
- **Reward function** $r(s, a)$: Defines goal
- **Value function** $V(s), Q(s, a)$: Expected downstream reward
- **Model (optional)**: Estimates $P(s'|s, a)$

Classification of RL algorithms:

- Model-free vs. model-based
- Value-based vs. policy-based

Reinforcement Learning in Optimization

Why RL in optimization?

- Many OR problems involve sequential decision-making under uncertainty
- Traditional OR uses mathematical programming (e.g., LP, MIP, DP), which struggles with:
  - High-dimensional or partially observed systems
  - Nonstationary dynamics and learning from interaction
  - Data-driven, model-free environments

RL can enhance OR by:

- Learning policies for complex systems without full models
- Combining simulation, optimization, and data in one loop
- Tackling combinatorial and structured problems (e.g., routing, scheduling, pricing)

Supervised vs. Reinforcement Learning

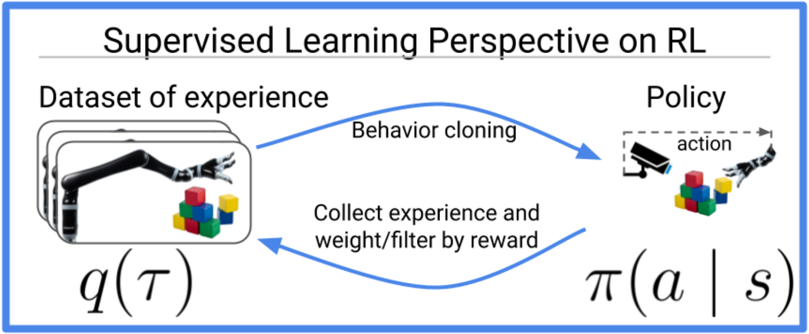


Figure: https://bair.berkeley.edu/static/blog/supervised_rl/supervised_perspective.png

Markov Decision Processes and Dynamic Programming

Markov Decision Processes

Markov Decision Processes (MDPs)

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- $\mathcal{S}$: state space
- $\mathcal{A}$: action space
- $P(s'|s, a)$: transition probability
- $r(s, a)$: immediate reward
- $\gamma \in [0, 1)$: discount factor

Objective: Maximize the expected discounted return:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right]$$

State Space: $\mathcal{S}$

The state $s_t \in \mathcal{S}$ captures all information necessary to make future decisions, compute rewards and guide transitions.

Types of state (Powell, 2022):

- **Physical state:** Position, inventory, battery level, queue length
- **Information state:** History, signals, auxiliary variables (e.g., demand forecasts)
- **Belief state:** Probability distribution over hidden states (used in POMDPs)

Note: State representation strongly influences tractability and learning success.

Action Space: $\mathcal{A}$

Actions $a_t \in \mathcal{A}(s_t)$ represent decisions taken at time t.

Key characteristics:

- Action set may depend on the state: $\mathcal{A}(s_t)$
- Actions can be discrete (e.g., `accept/reject`, `route A/B`) or continuous (e.g., `order quantity`, `price level`)
- Actions may be bounded or constrained (e.g., capacity limits)

Example: In inventory management,

$$\mathcal{A}(s) = \{a \in \mathbb{Z}_+ : s + a \leq \text{capacity}\}$$

Reward Function: $r(s, a)$

The reward captures the immediate performance of a decision.

General form:

$$r(s_t, a_t, \omega_t) \quad \text{where } \omega_t \text{ is exogenous information}$$

Possible dependencies:

- **State only:** Penalty for being in a bad state
- **Action only:** Cost of applying control
- **State-action pair:** Joint impact (e.g., ordering cost depends on current stock)
- **Randomness:** Realized revenue, stochastic cost

Transition Function: $P(s' \mid s, a)$

Describes how the system evolves after action a is taken in state s.

Split view (common in OR):

- **Deterministic component:** Post-decision state

$$s^a = S^M(s, a)$$

- **Stochastic component:** Exogenous information $\omega \sim \mathbb{P}$

$$s_{t+1} = S^M(s^{\text{post}}, \omega_t)$$

Example: Inventory dynamics

$$\underbrace{s^a = s + a}_{\text{replenish}} \qquad \underbrace{s' = \max(0, s^a - \text{demand}_t)}_{\text{realized demand}}$$

OR Example: Vehicle Routing as an MDP

State s_t:

- Current location and load of the vehicle
- Set of unvisited customers with attributes (location, demand, time windows)

Action a_t:

- Select sequence of customers to visit

Transition:

- Deterministic: vehicle moves to selected node, updates time and load

Reward:

- Negative travel cost (or penalty for lateness/violation)

Challenge:

- Action space is combinatorial (permutations of visit sequences)

OR Example: Inventory Management as an MDP

State s_t:

- Current stock level
- Pending orders and lead times

Action a_t:

- Order quantity (discrete or continuous)

Transition:

- Deterministic stock update + stochastic demand

Reward:

- Revenue − ordering cost − holding cost − stockout penalty

Challenge:

- Long planning horizon, delayed effects, partial observability

Dynamic Programming

What is Dynamic Programming?

- A method for solving multi-stage decision problems by breaking them into subproblems.
- Solves problems with the **principle of optimality**:

  > *"An optimal policy has the property that, whatever the initial state and initial decision, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." — Bellman, 1957*

- Used when transition probabilities and rewards are known.
- Core idea: **recursive value computation**.

Value Functions and Bellman Equations

State-Value Function:

$$V^\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \mid s_0 = s\right]$$

Action-Value Function:

$$Q^\pi(s, a) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \mid s_0 = s, a_0 = a\right]$$

Bellman Expectation Equation:

$$V^\pi(s) = \sum_a \pi(a|s)\left[r(s, a) + \gamma \sum_{s'} P(s'|s, a) V^\pi(s')\right]$$

Dynamic Programming in Shortest Path Problem

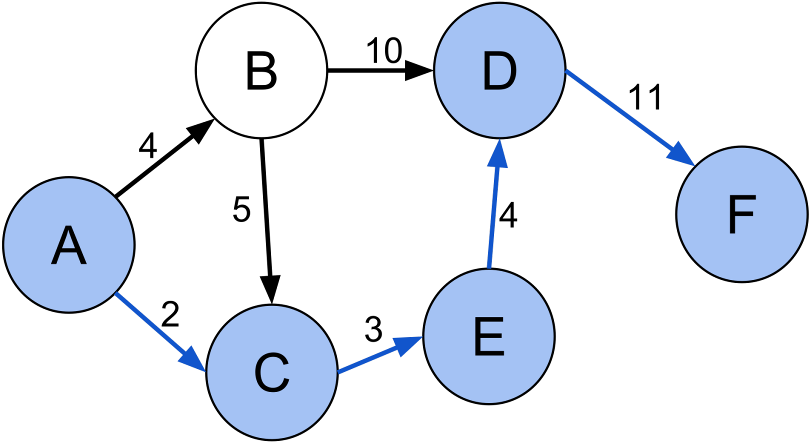


Figure:
https://www.cis.jhu.edu/~shanest/project/metricpatterntheory/additional/dynamic/dynamic2.html

Bellman Optimality Equations

Value function:

$$V^*(s) = \max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right]$$

Action-value function:

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a')$$

Greedy policy: $\pi^*(s) = \arg\max_a Q^*(s,a)$

Bellman Operator

Bellman Optimality Operator:

$$(TV)(s) = \max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) V(s') \right]$$

Key properties:

- T is a contraction mapping w.r.t. the sup norm
- T has a unique fixed point V^*, i.e., $TV^* = V^*$

Value Iteration: Iteratively apply T

$$V_{k+1} = TV_k$$

Value Iteration

Single-step update:

$$V_{k+1}(s) = \max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) V_k(s') \right]$$

Repeat until convergence:

$$\|V_{k+1} - V_k\| < \epsilon$$

Extract greedy policy:

$$\pi^*(s) = \arg\max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right]$$

Guarantees:

- Converges to V^* in finite MDPs
- Monotonic improvement toward optimal value

Policy Iteration

Two steps:

1. **Policy Evaluation:** Compute V^{π} under current policy π
2. **Policy Improvement:** $\pi'(s) = \arg\max_a Q^{\pi}(s, a)$

Repeat until convergence.

Guarantees:

- Finite convergence in finite MDPs
- Each iteration strictly improves or maintains performance

Linear Programming Formulation

Primal:

$$\min_V \sum_s \mu(s)V(s) \quad \text{s.t.} \quad V(s) \geq r(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s'), \quad \forall s, a$$

Dual:

$$\max_x \sum_{s,a} r(s,a)x(s,a) \quad \text{s.t.} \quad \begin{cases} \sum_a x(s,a) - \gamma \sum_{s',a'} x(s',a')P(s|s',a') \\ = \mu(s) \quad \forall s \\ x(s,a) \geq 0 \end{cases}$$

Interpretation: $x(s,a)$ is the **expected discounted visitation frequency** (occupation measure)

Foundations of Reinforcement Learning

Curse of Dimensionality: Outcome Space

Problem: The number of possible realizations of randomness (next states) becomes very large.

Examples:

- Demand in each of n regions: 2^n demand vectors
- Weather patterns, traffic, prices $\rightarrow$ continuous distributions

Implications:

- Exact expectation $\sum_{s'} P(s'|s,a)V(s')$ is intractable
- Requires sampling-based methods (Monte Carlo, TD)
- Drives the need for simulation and model-free learning

Curse of Dimensionality: State Space

Problem: The state space grows exponentially with the number of features or system components.

Examples:

- Multi-product inventory: $s = (s_1, s_2, \ldots, s_n)$
- Robot with d sensors: state in $\mathbb{R}^d$
- Time, location, demand forecasts $\rightarrow$ combinatorial blow-up

Implications:

- Value functions cannot be stored or computed exactly
- Tabular methods become infeasible
- Requires function approximation ($\rightarrow$ ADP, RL)

Curse of Dimensionality: Action Space

Problem: The number of possible actions grows rapidly with control granularity or system size.

Examples:

- Vehicle routing: $n!$ possible permutations
- Portfolio optimization: continuous vector in $[0, 1]^n$
- Multi-agent control: joint action space = product of agents' actions

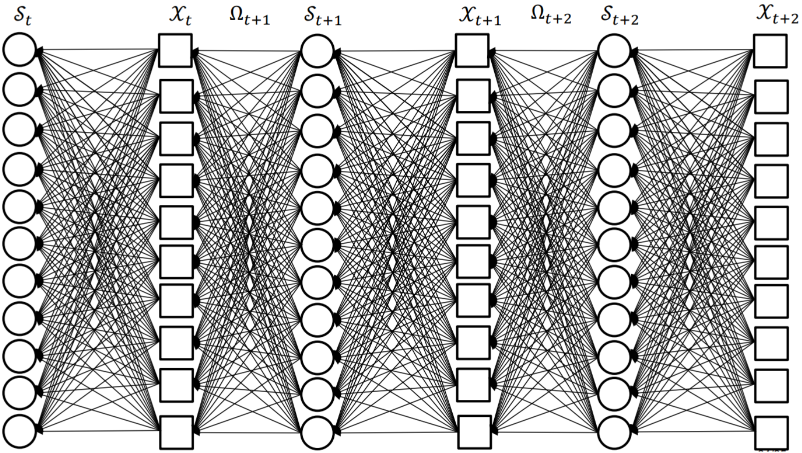
Implications:

- $\arg\max_a Q(s, a)$ becomes computationally expensive
- Discrete optimization or sampling required
- Leads to policy-based methods or parameterized actions

The Three Curses of Dimensionality

1. **Outcome Space Explosion:** Cannot compute expectations over all possible futures
   Solution: Monte Carlo sampling (Law of Large Numbers)
2. **State Space Explosion:** Cannot represent or learn value functions for all states
   Solution: Latent state representation (feature design)
3. **Action Space Explosion:** Finding or learning optimal actions becomes intractable
   Solution: Mathematical programming or sampling

Curses of Dimensionality Visualized



Approximate Dynamic Programming (ADP)

Why? Traditional DP is intractable for large-scale problems.

- Instead of computing the full Bellman expectation:

$$\sum_{s'} P(s'|s,a)\left[r(s,a)+\gamma V(s')\right]$$

  ADP uses a sample transition:

$$\hat{Q}(s,a) = r(s,a) + \gamma \hat{V}(s')$$

- Replace exact value functions with approximations:

$$V(s) \approx \hat{V}(s;\theta)$$

- Replace full state sweeps with sampling (simulation-based learning)
- Use projected Bellman updates or stochastic approximations

→ Sampled Q-values replace expectation over next states

Running Example: Cliff Walking

Figure: https://towardsdatascience.com/walking-off-the-cliff-with-off-policy-reinforcement-learning-7fdbcdfe31ff/

Monte Carlo Learning

Monte Carlo Estimation

Goal: Estimate $V^\pi(s)$ using sample rollouts.
First-Visit Monte Carlo:

$$V(s) \leftarrow \text{average return of first visits to } s$$

Properties:

- No need to know transitions
- High variance, but unbiased
- Only works for episodic problems

Exploration vs. Exploitation

Problem: Always choosing highest-valued state leads to suboptimal convergence
Solution: Insert random component in action selection, e.g., by randomly choosing an action with a probability ϵ
Challenge: Balance exploration (try new actions) and exploitation (use known good actions)

Monte Carlo Trajectory [1/2]

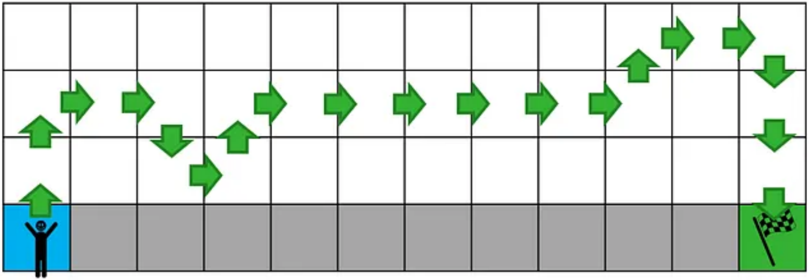


Figure: https://medium.com/data-science/cliff-walking-with-monte-carlo-reinforcement-learning-587e9d3bc4e7

Monte Carlo Trajectory [2/2]

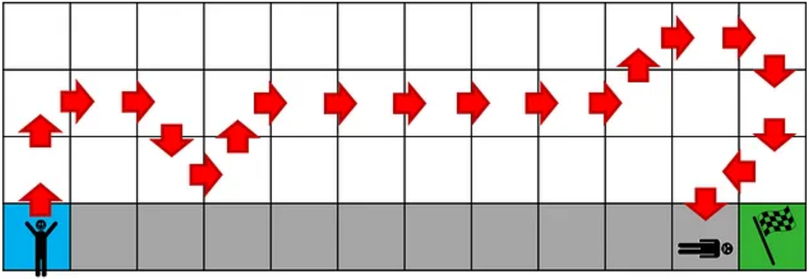


Figure: https://medium.com/data-science/cliff-walking-with-monte-carlo-reinforcement-learning-587e9d3bc4e7

Temporal Difference Learning

Temporal Difference Learning

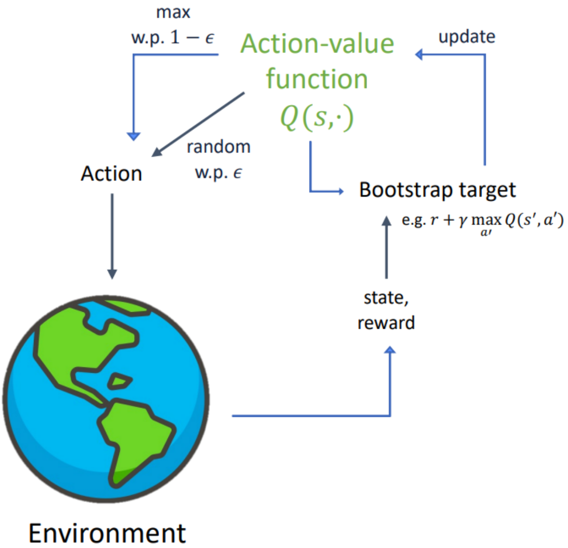
TD(0) update rule:

$$V(s) \leftarrow V(s) + \alpha \underbrace{\left[\underbrace{r}_{\text{reward}} + \gamma \underbrace{V(s')}_{\text{predicted value}} - \underbrace{V(s)}_{\text{current prediction}} \right]}_{\textbf{TD error}}$$

What happens:

- **Observed:** reward r and next state s' come from the environment
- **Predicted:** $V(s)$ and $V(s')$ are outputs of the current value function

Value-based learning



Advantages:

- Bootstraps from next state
- Lower variance than Monte Carlo
- Online and incremental

Downsides:

- Introduces bias (since it uses predictions to update predictions)
- Backpropagation of reward signals may take longer

Mapping RL Algorithms to DP Concepts

Conceptual analogies:

- **Q-learning:** Approximate value iteration (off-policy)
- **SARSA:** Generalized policy iteration (on-policy TD evaluation + improvement)
- **DQN:** Deep Approximate Value Iteration using neural networks + stabilizers

Note: These mappings are conceptual — convergence, stability, and approximation dynamics differ from exact DP.

SARSA

SARSA (on-policy):

$$Q(s,a) \leftarrow Q(s,a) + \alpha \underbrace{\left[r + \gamma Q(s',a') - Q(s,a)\right]}_{\text{TD error}}$$

- **Observed:** reward r, next state s', next action $a' \sim \pi$
- **Predicted:** Q-values come from current Q-function
- **Follows the action actually taken by the current policy**

Q-learning

Q-learning (off-policy):

$$Q(s,a) \leftarrow Q(s,a) + \alpha \underbrace{\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]}_{\text{TD error}}$$

- **Observed:** reward r, next state s'
- **Predicted:** greedy Q-value for next state
- **Uses the best predicted action, not necessarily the one taken**

SARSA vs. Q-learning: On-policy vs. Off-policy

SARSA (On-policy)

- Learns from the action actually taken by the current policy
- Updates using: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
- Follows the policy's own exploration
- More conservative: safer in stochastic or risky environments

Evaluates what you *did*

Q-learning (Off-policy)

- Learns from the greedy action regardless of current policy
- Updates using: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
- Partially ignores exploration during learning
- Can learn optimal policies even from suboptimal behavior

Evaluates what you *should* have done

Sample trajectory SARSA

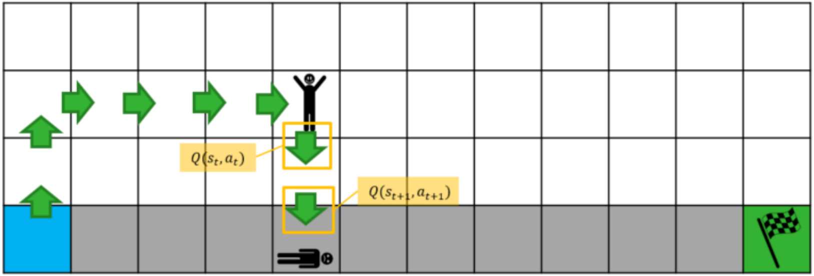


Figure: https://towardsdatascience.com/walking-off-the-cliff-with-off-policy-reinforcement-learning-7fdbcdfe31ff/

Sample trajectory Q-learning

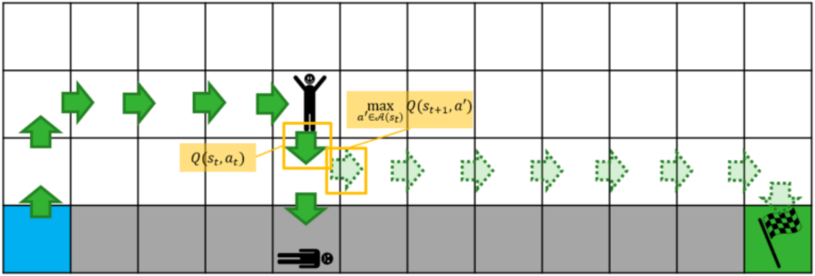


Figure: https://towardsdatascience.com/walking-off-the-cliff-with-off-policy-reinforcement-learning-7fdbcdfe31ff/

λ-Return and TD(λ): Bridging MC and TD

Idea: Combine short-term bootstrapping (TD) and long-term returns (MC) via trace decay parameter $\lambda \in [0, 1]$

- **n-step return:**

$$G_t^{(n)} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n})$$

- **λ-return:**

$$G_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

- **TD(λ) update:**

$$V(s_t) \leftarrow V(s_t) + \alpha \big(G_t^{\lambda} - V(s_t) \big)$$

Interpretation:

- $\lambda = 0$: standard TD(0) (Q-learning, Sarsa)
- $\lambda = 1$: standard TD(1) (Monte Carlo)

TD(λ) Visualized

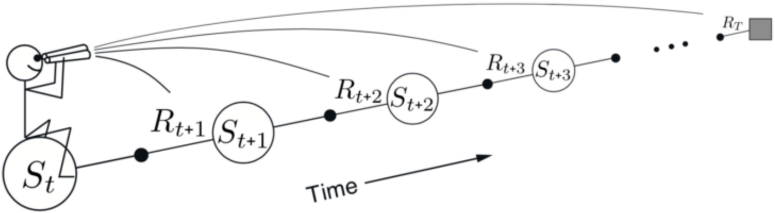
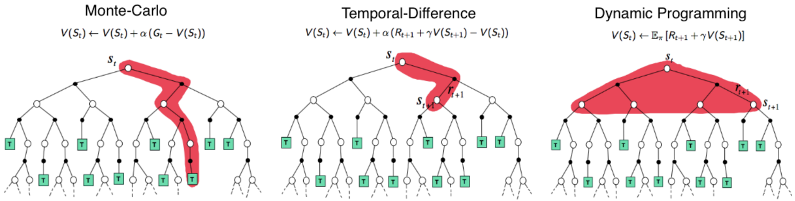


Figure: Sutton, R. S., & Barto, A. G. (1998). Reinforcement Learning: An Introduction (Vol. 1, No. 1, pp. 9-11). Cambridge: MIT Press.

DP, MC and TD



https://davidstarsilver.wordpress.com/wp-content/uploads/2025/04/lecture-4-model-free-prediction-.pdf

Exploration strategies

Epsilon-Greedy Selection

ϵ-greedy:

- With probability ϵ, choose a random action
- With probability $1 - \epsilon$, choose: $\arg\max_a Q(s, a)$
- Simple and effective; ϵ often decayed over time

Boltzmann Exploration [1/2]

Softmax / Boltzmann exploration:

- Choose action a with probability:

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}$$

- Temperature τ controls exploration: high $\tau \to$ uniform, low $\tau \to$ greedy

Boltzmann Exploration [2/2]

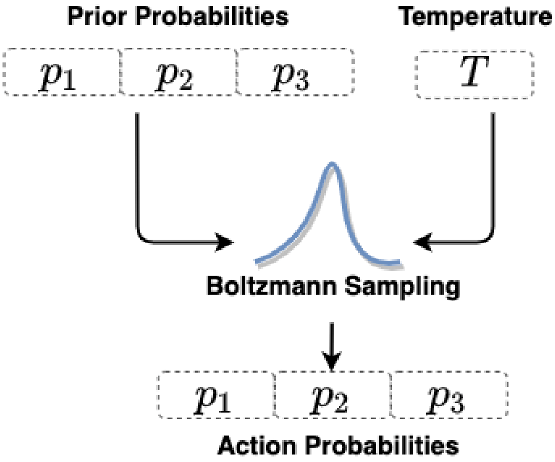


Figure: `https://medium.com/the-modern-scientist/mastering-the-unknown-leveraging-softmax-and-boltzmann-exploration-for-optimal-decision-making-in-cccd3`

Upper Confidence Bounds

- **Upper Confidence Bound (UCB):**
  - Inspired by bandits; selects:

$$a_t = \arg\max_a \left[Q(s,a) + c \cdot \sqrt{\frac{\log t}{N(s,a)}} \right]$$

  - Encourages exploration of less-visited actions
  - Requires tracking visit counts $N(s,a)$
- **Trade-off:** Too little exploration $\rightarrow$ premature convergence; too much $\rightarrow$ instability and noise

Upper Confidence Bound [2/2]

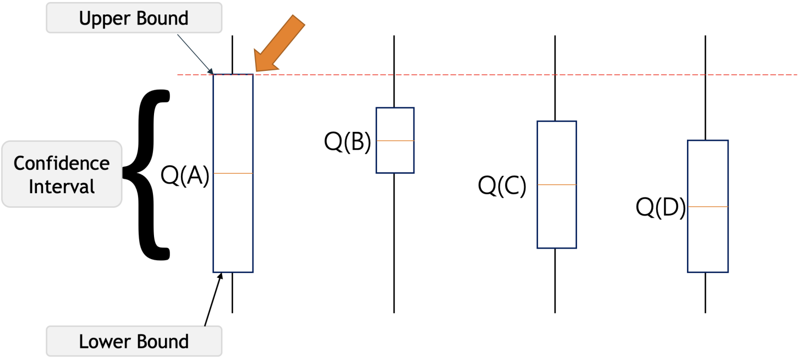


Figure: https://www.geeksforgeeks.org/machine-learning/upper-confidence-bound-algorithm-in-reinforcement-learning/

Learning rates

Learning Rate: Intuition

What is it?

- The learning rate α determines how much new information overrides old estimates.
- Balances **stability vs. plasticity**.

Effect of α:

- **Small** α:
  - Slow learning
  - More stable updates
  - Relies heavily on past experience
- **Large** α:
  - Fast adaptation
  - More volatile and noisy estimates

Analogy: Learning rate as a memory decay parameter

Learning Rate in Practice

Tabular Q-learning:

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

Choices of α:

- **Constant:** $\alpha = 0.1$ (common in early experiments)
- **Decaying:** $\alpha_t = \frac{1}{1+t}$ or $\alpha_t = \frac{c}{c+t}$ for some $c > 0$

In deep RL:

- α is optimizer step size (e.g., Adam, RMSProp)
- Typically: $\alpha \in [10^{-5}, 10^{-3}]$ and tuned per architecture

Convergence Conditions on Learning Rate

For convergence of stochastic approximation, learning rate must satisfy Robbins–Monroe conditions:

$$\sum_{t=0}^{\infty} \alpha_t = \infty \qquad \text{and} \qquad \sum_{t=0}^{\infty} \alpha_t^2 < \infty$$

Interpretation:

- Learning must continue forever (nonzero updates)
- But updates must shrink fast enough to stabilize

Typical choice:

$$\alpha_t = \frac{1}{t+1} \quad \text{or} \quad \alpha_t = \frac{c}{c+t}$$

In deep RL:

- No convergence guarantees (non-stationary, non-convex)
- Empirical tuning of α is critical

Value Function Approximation

Value Function Approximation in RL

Why? Too many states (and thus state values) to store in a table.
Key idea: Use parameterized function $Q(s, a; \theta)$
Common choices:

- Linear approximators
- Decision trees, kernel methods
- Neural networks

Linear Function Approximation

Goal: Approximate value function or Q-function:

$$\hat{V}(s) = \phi(s)^\top \theta \quad \text{or} \quad \hat{Q}(s, a) = \phi(s, a)^\top \theta$$

Where:

- $\phi(s)$: feature vector (manually designed or derived)
- θ: learned weights

Advantages:

- Simple, interpretable, computationally efficient
- Enables generalization across states

Limitations:

- Cannot represent nonlinear relationships
- Often underfits real-world problems

Nonstationary Regression in RL

Key difference from supervised learning:

- In RL, the targets (e.g., returns or TD targets) depend on the function we are learning

Example (TD learning):

$$y_t = r_t + \gamma \hat{V}(s_{t+1}) \quad \text{depends on } \hat{V}$$

Implications:

- Targets change as the model changes
- Regression problem is **nonstationary** and bootstrapped
- Can cause instability and divergence

Contrast: Supervised learning assumes fixed input-output pairs

Polynomial Approximators

Idea: Extend linear approximation using polynomial basis:

$$\hat{V}(s) = \sum_{\alpha \in \mathbb{N}^n,\ |\alpha| \leq d} \theta_\alpha \cdot \phi^\alpha$$

Properties:

- More expressive than linear functions
- Still analytically tractable
- Sensitive to overfitting and poor generalization in high dimensions

Use case: Low-dimensional problems or as a benchmark

Deep Q-Learning

Deep Q-Learning: Motivation and Overview

Problem: Tabular Q-learning fails in large or continuous state spaces.

Solution: Use a neural network to approximate the Q-function:

$$Q(s, a; \theta) \approx Q^*(s, a)$$

- States and actions are mapped to Q-values via a deep network.
- Trained using temporal-difference targets:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$$

- Loss: $\mathcal{L}(\theta) = \big(y - Q(s, a; \theta)\big)^2$

Challenge: Prone to overfitting.

Deep Q-Learning Network (DQN) Architecture

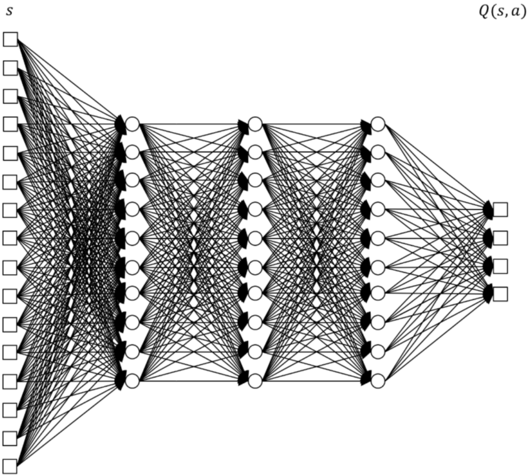


Figure: https://towardsdatascience.com/a-minimal-working-example-for-deep-q-learning-in-tensorflow-2-0-e0ca8a944d5e/

Deep Q-Network (DQN)

Update target:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$$

Loss:

$$\mathcal{L}(\theta) = (y - Q(s, a; \theta))^2$$

Key components:

- Experience replay buffer
- Mini-batch gradient descent
- Target network θ^-

Replay Buffer: Breaking Sequential Correlation

Problem: Transitions observed in sequence are highly correlated $\rightarrow$ violates i.i.d. assumption, harms learning stability.

Solution: Store experience in a **replay buffer** $\mathcal{D}$ and sample randomly:

$$(s_t, a_t, r_t, s_{t+1}) \in \mathcal{D}$$

Why it helps:

- Breaks temporal correlations
- Improves sample efficiency via re-use
- Enables mini-batch training

Batch (Offline) Reinforcement Learning

Scenario: Learn entirely from a fixed dataset $\mathcal{D}$, without further interaction with the environment.

Training loop:

- Sample batch $\{(s_i, a_i, r_i, s_i')\}_{i=1}^{B} \sim \mathcal{D}$
- Compute targets (e.g., $y_i = r_i + \gamma \max_{a'} Q(s_i', a')$)
- Update network via stochastic gradient descent

Advantages:

- Safe and sample-efficient, useful in robotics/health/finance
- Experience reuse without further environment calls

Challenges:

- Distributional shift from behavior policy μ to target policy π
- Bootstrapping errors can accumulate over time

Mini-Batch Learning: Smooth and Stable Updates

Problem: Single-sample updates are noisy; full-batch is inefficient and slow.

Solution: Sample small batches (e.g., $B = 32$–512) from replay buffer:

- Reduces gradient variance
- Improves numerical stability
- Prevents overfitting to individual outliers

Used in: Most deep RL algorithms (DQN, DDPG, SAC, etc.)

Target Network: Decoupling Targets for Stability

Problem: Using the same network for both prediction and target leads to instability (due to strong correlation between s_t and s_{t+1} and moving targets).

Solution: Maintain a separate **target network** with parameters θ^-:

$$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-)$$

$$\text{Loss: } \left(y_t - Q(s_t, a_t; \theta)\right)^2$$

Update rule:

- Every K steps: $\theta^- \leftarrow \theta$
- Or: Soft updates (Polyak averaging): $\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$

Stabilizes learning by decoupling prediction and target values

Limitations of Deep Q-Learning

Challenges:

- Instability and divergence with function approximation
- Overestimation bias (fixed in Double Q-learning)
- Not well-suited for continuous action spaces

Solutions:

- Double DQN
- Dueling DQN
- Actor-critic methods (covered in Lecture 2)

Advanced DQN

Double Deep Q-Learning

Problem in standard DQN: Q-values are overestimated due to the max operator in the target:

$$y_t^{\mathrm{DQN}} = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-)$$

Fix: Use **Double Q-learning**:

$$y_t^{\text{Double DQN}} = r_t + \gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta); \theta^-)$$

Key idea:

- Use online network θ to select the action
- Use target network θ^- to evaluate it

Effect: Reduces overestimation bias, improves stability.

Advantage Function: Decomposing Q-values

Definition:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

Interpretation:

- Measures how much better (or worse) action a is in state s, **compared to the average action**
- If $A^\pi(s,a) > 0$, then action a is better than the expected value of the state

Why this matters in value-based RL:

- Q-values mix both state value and action-specific advantage
- Separating $V(s)$ and $A(s,a)$ leads to better generalization across actions

Dueling Deep Q-Networks

Observation: In some states, the choice of action has little effect on value.

Solution: Decompose Q-value:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

Network architecture:

- Shared convolutional layers
- Two output heads:
  - Value stream $V(s)$
  - Advantage stream $A(s,a)$

Effect: Learns state values more efficiently when many actions yield similar outcomes.

Dueling Deep Q-Learning Architecture

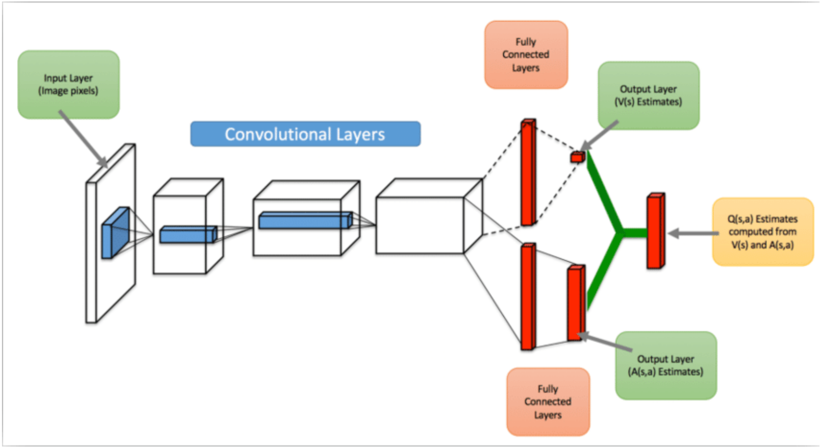


Figure: Sewak, Mohit. (2019). Deep Q Network (DQN), Double DQN, and Dueling DQN: A Step Towards General Artificial Intelligence.

Multi-Step Learning

Standard Q-learning uses 1-step bootstrapping:

$$y_t^{(1)} = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$$

Multi-step target:

$$y_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \max_{a'} Q(s_{t+n}, a')$$

Advantages:

- Faster credit assignment over delayed rewards
- Combines benefits of Monte Carlo (low bias) and Temporal Difference (low variance)

Trade-off: Larger n reduces bias, but increases variance.

Distributional Reinforcement Learning

Standard RL: Learn expected return

$$Q(s,a) = \mathbb{E}[R_t \mid s, a]$$

Distributional RL: Learn full distribution of returns

$$Z(s,a) \approx \text{distribution of } \sum_{t=0}^{\infty} \gamma^t r_t$$

Benefits:

- Captures uncertainty and risk
- More informative learning targets
- Improves stability and performance

Key Algorithms:

- **C51 (Categorical DQN)**: Discrete support + softmax
- **QR-DQN**: Quantile regression for estimating return quantiles

C51: Categorical DQN

Idea: Approximate the return distribution $Z(s, a)$ using a categorical distribution over fixed support (the **atoms**)

Support: $z_i = V_{\text{min}} + i \cdot \Delta$, for $i = 0, \ldots, N-1$
where $\Delta = \frac{V_{\text{max}} - V_{\text{min}}}{N-1}$

Representation: Learn probabilities $p_i(s, a)$ over atoms z_i:

$$Z(s, a) \approx \sum_{i=0}^{N-1} p_i(s, a) \cdot \delta_{z_i}$$

Why more informative targets?

- Bellman operator becomes a **distributional projection**
- Disambiguate actions with similar means but different risks

Loss: KL divergence between projected and predicted distribution

Why Fixed Supports in Distributional RL?

Problem with parametric distributions:

- If we model $Z(s, a)$ as a Gaussian (or other simple parametric family), Bellman updates often create shapes that are skewed, multi-peaked, or heavy-tailed.
- These shapes cannot be faithfully represented by a single Gaussian $\rightarrow$ information loss.

Fixed support / atoms solution:

- Define a fixed grid of possible returns z_i (the atoms).
- After a Bellman update, map the new distribution back onto this grid (distributional projection).
- Only the probabilities on the atoms change; the support stays fixed $\rightarrow$ stable and flexible.
- Can capture multi-modality, skew, and tail risks.

Distributional Bellman Operator

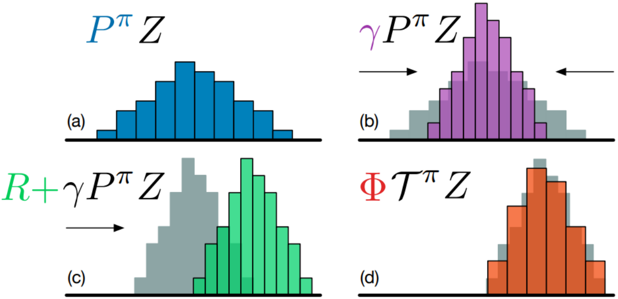


Figure 1. A distributional Bellman operator with a deterministic reward function: (a) Next state distribution under policy π, (b) Discounting shrinks the distribution towards 0, (c) The reward shifts it, and (d) Projection step (Section 4).

Figure: Bellemare, M. G., Dabney, W., & Munos, R. (2017). A distributional perspective on reinforcement learning. In ICML (pp. 449-458).

Noisy Networks (Noisy Nets)

Goal: Learn an exploratory policy via parameter noise

Replace deterministic weights with stochastic ones:

$$y = (\theta + \sigma \odot \epsilon)x + b$$

- ϵ: sampled noise
- σ: learnable noise scaling parameter

Advantages:

- Enables exploration through randomness in action preferences
- Replaces ϵ-greedy with state-dependent, learned noise
- More efficient and adaptive exploration

Rainbow DQN Architecture

Combines six DQN improvements into one architecture:

1. **Double DQN** — reduces overestimation bias
2. **Dueling networks** — separates state value from advantage
3. **Prioritized replay** — samples important transitions more often
4. **Multi-step learning** — bootstraps over n steps
5. **Distributional RL** — learns the full distribution of returns
6. **Noisy Nets** — exploration through learned parametric noise

Rainbow = stability $\oplus$ efficiency $\oplus$ exploration $\oplus$ generalization

Limitations of DQN in Operations Research

Deep Q-Learning works well for:

- Low-dimensional state/action spaces
- Immediate feedback and short horizons
- Flat, unstructured problems (e.g., Atari)

But many OR problems feature:

- **Combinatorial action spaces** (e.g., routing, scheduling)
- **Structured states** (e.g., graphs, sets, sequences)
- **Long horizons and sparse rewards** (e.g., inventory, supply chains)
- **Constraints and feasibility** (e.g., capacity, time windows)

Implication: Standard DQN fails to scale or generalize; requires structured encoders (e.g., GNNs), policy-based methods, or model-based planning.

Wrapping up

Lecture 1 Summary: Foundations of Reinforcement Learning

What we covered:

- Formulated decision-making as a Markov Decision Process
- Explored exact solution methods: Value Iteration, Policy Iteration, LP
- Introduced sample-based methods: Monte Carlo, Temporal Difference Learning
- Compared on-policy (SARSA) and off-policy (Q-learning) learning
- Motivated Value Function Approximation and Deep Q-Networks (DQN) for scaling to large state spaces problems

Takeaway: Value-based RL provides powerful tools but struggles in large, continuous, or structured spaces — motivating the shift to policy-based and function-approximation-based methods.

Contact details

Wouter van Heeswijk
Assistant Professor
Operations Research & Financial Engineering
University of Twente
w.j.a.vanheeswijk@utwente.nl
+31 53 489 8460