Reinforcement Learning Notes

Lecture 1: Introduction to Reinforcement Learning

1.1 What is AI?

- AI: Can machine M autonomously solve a problem in environment E?
- M takes actions based on state S to maximize reward R.
- If E is known: use classical CS / control theory.
- If E is unknown: use Reinforcement Learning (RL).

1.2 Adaptive Systems

- Adaptive systems update internal parameters (e.g., weights).
- Examples: Adaptive equalizers, filters.
- RL = adaptive system + decision-making over time to reach goals.

1.3 Reinforcement Learning Formulation

- Agent M interacts with environment E over time.
- At each step t:

$$S_t \xrightarrow{A_t} R_{t+1}, S_{t+1}$$

- Objective: learn a policy π that maximizes return G_t.

1.4 Definitions

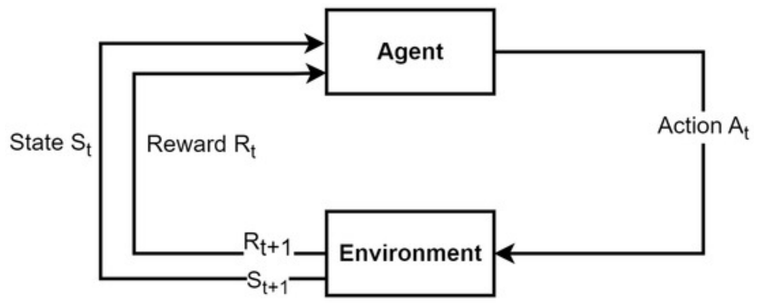

Figure 1: Agent-Environment interaction in reinforcement learning.

- M: Agent (machine/robot)
- E: Environment
- S: State (feedback from E)
- A: Action (chosen by M)
- R: Reward (goal indicator)

Agent acts on E via A, and receives R, S as feedback.

1.5 Example: Tic Tac Toe (TTT)

- S: board state
- A: available moves (1–9)
- R: reward
  - Win: +100
  - Draw: -50 or 0
  - Lose: -100
- Goal of agent: maximize R

1.6 Return Function G_t

- Finite horizon:
$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T$$
- Infinite horizon (discounted):
$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$
- $\gamma \in [0, 1]$: discount factor.

1.7 Markov Property

- Environment is Markovian if:
$$\Pr(S_{t+1}, R_{t+1} | S_t, A_t)$$
- Current state S_t contains all necessary information.
- Action selection only depends on S_t, not full history.

1.8 Policy

- Policy $\pi(a|s)$: probability of choosing a in state s.
- Can be deterministic or stochastic.

1.9 Action Selection

- The agent must select A_t that maximizes future rewards.
- In ideal case, if agent could foresee the future, it would:
$$A_t = \arg\max_a \mathbb{E}[G_t | S_t, A_t = a]$$
- In practice, agent must learn from experience.

1.10 Reward Design

- Reward function must reflect the actual goal.
- Poorly designed reward signals can mislead the agent.
- Example (chess):
  - Goal = win (checkmate): assign +100 for win.
  - Don't reward piece captures if that's not the real goal.
- Reinforcement learning requires engineer-defined reward signals aligned with the goal.

1.11 Observable State and Markov Assumption

- RL assumes Markovian environments.
- In fully observable environments (e.g., Tic Tac Toe): S_t is sufficient.
- In partially observable settings (e.g., Poker): S_t may not be sufficient.
- In this course, we focus on Markov Decision Processes (MDPs).

Lecture 2: Introduction to Reinforcement Learning

2.0 Episode Definition

- A sequence of agent-environment interactions is called an **episode**.
- Also referred to as a *run*, *trace*, or *play-out*.
- Episode example: $S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_T$

2.1 RL as Sequential Decision Making

- An RL agent (controller M) interacts with the environment E.
- At each time step t:

$$S_t \xrightarrow{A_t} R_{t+1}, S_{t+1}$$

- Goal: learn a policy to maximize cumulative reward G_t over time.
- The agent observes state $S_t \in \mathcal{S}$ and selects action $A_t \in \mathcal{A}$.

2.2 Markov Property and Policies

- The environment satisfies the **Markov property**:

$$\Pr(S_{t+1}, R_{t+1} | S_t, A_t)$$

- This implies S_t is a sufficient statistic—no need to consider history.
- The policy π maps from states to action distributions:

$$\pi(a|s) = \Pr(A_t = a | S_t = s)$$

- Initially, policies are often fully exploratory (e.g., ε-greedy with $\varepsilon = 1$).

2.3 Objective: Maximizing Return

- Define return from time t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

- Discount factor $\gamma \in [0, 1]$ controls horizon:
  - $\gamma \approx 0$: short-term focus
  - $\gamma \approx 1$: long-term accumulation
- $\gamma < 1$ ensures convergence of the infinite sum.
- The return is the foundation for evaluating how good actions are.

2.4 Value Functions

- **State-value function:**

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

- **Action-value function:**

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

- These estimate the expected return under policy π.
- Provide a basis for improving decisions over time.

2.5 Experience and Learning

- Learning occurs from sequences of experience:

$$\{S_0, A_0, R_1, S_1, A_1, R_2, \dots\}$$

- Goal is to estimate $q_\pi(s, a)$ from sampled episodes.
- These estimates are then used to improve the policy π.
- In many real environments, the model $p(s', r | s, a)$ is unknown—learning is essential.

2.6 Tabular Representation of Value Functions

- When $|\mathcal{S}|$ and $|\mathcal{A}|$ are small, value functions can be stored as tables.
- Each table entry corresponds to an estimate of $q_\pi(s,a)$.
- These tables are used to guide action selection and update the policy.

2.7 Modeling Known Environments

- If the transition dynamics $p(s',r|s,a)$ are known, planning algorithms can be applied.
- Use the Bellman expectation equation:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a') \right]$$

- This allows for synthetic experience and learning without sampling from the environment.

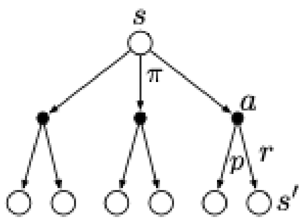

Figure 2: planning via Bellman backups

2.8 Initial Policy and Learning Motivation

- The agent begins with no knowledge; action choices are random or uniform.
- Over time, experience is used to improve estimates and refine the policy.
- The learning process turns randomness into purposeful behavior.

Lecture 3: Model-based RL: Dynamic Programming

3.1 From Goals to Action Selection

- Agent is in state S_t; must choose action $A_t \in \mathcal{A}$
- Policy $\pi(a|s)$: probability of choosing action a given state s
- Return: $G = \sum_{t=1}^{\infty} \gamma^{t-1} R_t$
- Goal: maximize expected return $\mathbb{E}_\pi[G]$

3.2 Action-Value Function

- $Q_\pi(s,a) = \mathbb{E}[G_t \mid S_t = s, A_t = a]$
- Represents expected return starting from s, taking a, and following π
- Use $Q_\pi(s,a)$ to solve action selection: choose $a = \arg\max Q_\pi(s,a)$
- Table structure: $|\mathcal{S}| \times |\mathcal{A}|$

3.3 Bellman Expectation Equation (Model-Based)

- Assume transition model $p(s', r \mid s, a)$ is known
- Recursive equation:

$$Q_\pi(s,a) = \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s', a') \right]$$

- Computes expected value via weighted sum over next states and rewards

3.4 Deterministic vs Stochastic Models

- Deterministic: use differential/difference equations (control theory)
- Stochastic: use probabilities from experience (RL context)
- Our course focuses on stochastic models

3.5 Solving Bellman Equations

- Closed-form yields $N \times K$ equations for $Q_\pi(s,a)$
- Solvable via linear algebra, but impractical due to:
  - Numerical instability
  - Memory/compute cost
- Instead: use successive approximation

3.6 Successive Approximation (Fixed-Point Iteration)

- Initialize $Q_\pi(s,a)$ arbitrarily
- Loop until convergence:

$$Q_\pi(s,a) \leftarrow \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s', a') \right]$$

- Repeat until change $< \theta$ for all (s,a)
- Guaranteed to converge if $\gamma < 1$ or $T < \infty$

3.7 Policy Evaluation Algorithm

- Step 1: Initialize $Q_\pi(s,a)$ arbitrarily
- Step 2: Repeat

  - For all (s, a), update using Bellman expectation
  - If change $< \theta$, stop
  - Else, copy updated values back
- Computes Q_π for a given policy π
- *Note: Q_π is tentative — based on current π and not guaranteed to be optimal yet.*

3.8 Policy Improvement

- Given $Q_\pi(s, a)$, improve π:
$$\pi'(s) = \arg\max_a Q_\pi(s, a)$$
- If multiple actions tie: distribute probability equally
- Upgrade π slightly: amplify high-value actions, attenuate others

3.9 Generalized Policy Iteration (GPI)

- Alternate between:
  - **Evaluation:** Compute Q_π
  - **Improvement:** Update π using Q_π
- Repeat until π converges (i.e., $\pi' = \pi$)
- This process converges to the optimal policy π^*

3.10 Computational Considerations

- Policy evaluation is asymptotic (runs until fixed point)
- Full GPI structure:
$$\pi_0 \xrightarrow{\text{evaluate fully}} Q_{\pi_0} \xrightarrow{\text{improve}} \pi_1 \xrightarrow{\text{evaluate fully}} \cdots \Rightarrow \pi^*$$
- **Truncated GPI:**
$$\pi_0 \xrightarrow{\text{1-step or few-step eval}} \tilde{Q}_{\pi_0} \xrightarrow{\text{improve}} \pi_1 \xrightarrow{\text{quick eval}} \cdots \Rightarrow \pi^*$$
- Truncated evaluation still converges to π^* but allows for faster incremental improvements
- *Motivation: More responsive, better-engineered systems with continuous improvement*

Lecture 4: Model-free RL: Monte Carlo Methods

4.1 Recall Definition of Action Value

- Action-value function: $q_\pi(s, a)$ is the expected return after taking action a in state s under policy π
- Estimate from many sampled trajectories: $q_\pi(s, a) \approx \frac{1}{N} \sum_{i=1}^{N} G_i$
- Averaging returns gives an unbiased estimate of $q_\pi(s, a)$
- Does not require model P; derived from direct interaction with the environment

4.2 Use Model to Calculate $q_\pi(s, a)$

- Use Bellman expectation equation:

$$q_\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a]$$

- Evaluate expectation by expanding over next state-action pairs using known transition probabilities P
- Practical only when model P is known

4.3 Generalized Policy Iteration (GPI)

- Evaluate q_π with respect to π
- Improve: $\pi'(s) = \arg\max_a q_\pi(s, a)$
- Repeat: Evaluate $\rightarrow$ Improve
- Fixed point: π^* is optimal w.r.t. q_{π^*}
- Approximate value functions can be used (e.g., truncated evaluation)

4.4 Expectation by Empirical Sampling

- Unknown expectations are estimated through experience
- Empirical estimation:
  - Observe values $x_1, x_2, \dots, x_n$
  - Compute mean: $\frac{1}{n} \sum_{i=1}^{n} x_i$
- In the limit $n \to \infty$, sampling approximates the true expected value

4.5 Experience and Return Estimation

- Agent interacts with the environment: generates $(S_0, A_0, R_1, S_1, A_1, \dots)$
- For each (s, a), record returns:

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots$$

- Average these returns to estimate $q_\pi(s, a)$
- Requires multiple episodes to ensure sufficient data coverage

4.6 Experience Quality

- Low-quality: sparse (s, a) coverage
- High-quality: rich, diverse coverage of all (s, a) pairs
- Aim: fully populate the return table to enable meaningful policy improvement

4.7 Exploration Strategies

(i) Exploratory Starts (ES)

- Force exploration by starting in every possible (s, a) pair
- Only feasible in simulators or controlled environments

(ii) ϵ-Soft Policies

- With probability ϵ, take a random action; otherwise, act greedily

- Ensures every action has non-zero probability
- Eventually explores the entire state-action space (law of large numbers)

4.8 Monte Carlo GPI

- Evaluate q_π using Monte Carlo sampling
- Repeat: Evaluate $\rightarrow$ Improve
- Each step combines MC evaluation with greedy policy improvement
- Requires full-episode sampling; slower than DP with truncated evaluation

4.9 GPI Conflict with ϵ-Softness

- ϵ-softness ensures exploration but conflicts with greedy improvement
- Optimal policies are often deterministic (assign probability 1 to best action)
- GPI with fixed ϵ-soft policies cannot converge to true optimal policy

4.10 On-Policy vs Off-Policy MC

	On-Policy	**Off-Policy**
Optimal Policy (GPI)	Yes (limited by exploration strategy)	Yes
Exploratory Policy (Eval)	Same as target policy	Separate behavior policy

4.11 Importance Sampling Ratio (ISR)

- **Goal:** Estimate $q_\pi(s, a)$ using data collected from a different policy μ
- Condition: $\mu(a|s) > 0$ whenever $\pi(a|s) > 0$ (i.e., μ covers π)
- Correct returns using:

$$\text{ISR} = \prod_{t=0}^{T-1} \frac{\pi(A_t \mid S_t)}{\mu(A_t \mid S_t)}$$

- Apply ISR to reweight MC returns from μ to estimate values under π

Lecture 5: Model-free RL: Temporal-Difference Learning

5.1 From Monte Carlo to TD Learning

- **Monte Carlo (MC):** Estimate $Q(s,a)$ via complete returns G_t after an episode ends.
- **TD(0):** Update $Q(s,a)$ after each step using:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R + \gamma Q(s',a') - Q(s,a)\right)$$

- TD bootstraps from the current estimate of the next state-action value.

5.2 Averaging and Incremental Updates

- **Averaging:** $\bar{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i$
- **Incremental form:**

$$\bar{x}_n = \bar{x}_{n-1} + \frac{1}{n}(x_n - \bar{x}_{n-1})$$

- Define learning rate α as:

$$\alpha = \frac{1}{n} \quad \text{(or fixed value for constant step size)}$$

- **General TD update:**

$$Q_n(s,a) = Q_{n-1}(s,a) + \alpha \left(\text{target} - Q_{n-1}(s,a)\right)$$

5.3 TD(0) and SARSA

- TD(0) with policy π is an on-policy method.
- Update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R + \gamma Q(s',a') - Q(s,a)\right)$$

- Called **SARSA**: State, Action, Reward, next State, next Action.
- Learning happens during the episode using transitions:

$$(s,a,r,s',a') \sim \pi$$

- Policy used to gather data is also used for updates.

5.4 Expected SARSA

- Use expectation over π instead of sampled a':

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R + \gamma \sum_{a'} \pi(a'|s') Q(s',a') - Q(s,a)\right)$$

- Reduces variance in updates by averaging over all possible actions at s'.
- More stable and faster-converging than regular SARSA.

5.5 Q-Learning: An Off-Policy TD Method

- Greedy update w.r.t. current Q:

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

- Off-policy: follows behavior policy b, learns greedy target policy $\pi = \arg\max Q$.
- Learns optimal value function Q^* regardless of behavior policy.

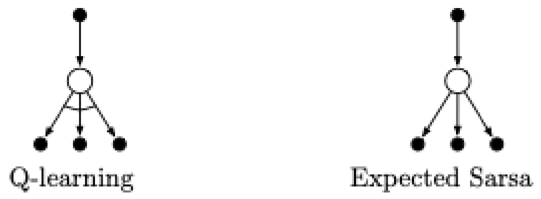

Figure 3: The backup diagrams for Q-learning and Expected Sarsa.

5.6 SARSA vs Q-Learning

- **SARSA:** On-policy, learns value of the actual policy being followed.
- **Q-Learning:** Off-policy, learns value of the greedy policy while following a potentially exploratory behavior.
- Q-learning improves even while using ϵ-greedy behavior.

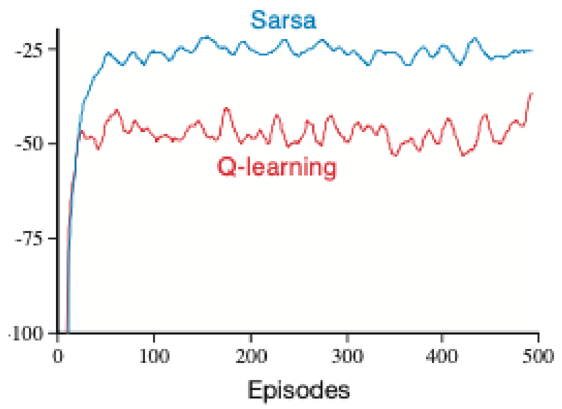

Figure 4: Performance comparison of SARSA and Q-learning over episodes.

5.7 Monte Carlo vs TD

- **Monte Carlo:**
  - Learn from full returns G_t after episode ends.
  - Unbiased but high variance.
- **TD:**
  - Learn from partial returns (bootstrapping).
  - Biased but lower variance.

5.8 On-policy vs Off-policy Summary

- **On-policy:** Learn value of the policy being used to interact with the environment.
- **Off-policy:** Learn value of a separate, target policy.
- Examples:
  - TD(0), SARSA, Expected SARSA: On-policy
  - Q-learning: Off-policy

5.9 Importance Sampling (IS)

- Required for off-policy Monte Carlo learning.

- Corrects mismatch between target policy π and behavior policy b:

$$\frac{\pi(a|s)}{b(a|s)}$$

- Allows unbiased estimation using samples from b while learning about π.

5.10 Summary of TD Variants

- **TD(0) / SARSA:** On-policy, one-step bootstrapping.
- **Expected SARSA:** Lower-variance on-policy TD method.
- **Q-Learning:** Off-policy, uses max operator for optimal learning.
- **All:** Use TD targets for sample-efficient and online learning.

Lecture 6: Function Approximation and Neural Networks

6.1 Motivation for Function Approximation

- **GPI with Approximate Value Functions:**
  - Approximate $q_\pi(s, a)$ using experience, not a model.
  - Tabulated values summarize experience.
  - GPI still applicable: Approximate $\rightarrow$ Improve $\rightarrow$ Repeat.
- **TD(0):** Makes updates per step via bootstrapping from current value estimates.
- **Monte Carlo:** Requires full episodes; slower updates.
- TD(0) allows quicker and more frequent policy improvement.

6.2 Limitations of Tabular Representation

- Tables don't scale to large or continuous state/action spaces.
- Fine discretization $\Rightarrow$ Huge tables.
- Tables fail with continuous inputs unless discretized.
- Goal: find a more scalable function representation.

6.3 Functions: A Mathematical Refresher

- Functions map domain to range: $f : D \rightarrow \mathbb{R}$
- Example: $q_\pi(s, a)$ maps from (s, a) to expected return.
- Three representations:
  1. **Table of Values** (Finite domain only)
  2. **Closed-form Expressions** (e.g., $f(x) = ax + b$)
  3. **Function Approximators** (e.g., Linear Regression, Neural Networks)
- *Triangle View:* These three forms can express similar concepts — all represent functions but differ in generalization and flexibility.

6.4 Why Closed-form Isn't Enough

- Closed-form impractical for complex or unknown mappings (e.g., q_π for chess).
- Need generalization from limited samples.
- Approximators offer compact and general representations.

6.5 Function Approximation via Data

- Inspired by physics/science: approximate f from data.
- Use a model (e.g., linear regression) to learn function $f(x)$ from data $\{(x_i, t_i)\}$.
- General goal: Minimize prediction error between $f(x)$ and t.

6.6 Linear Regression

- Hypothesis: $y = mx + b$ fits data best.
- Define loss:

$$E = \frac{1}{M} \sum_{i=1}^{M} \left(y(x^{(i)}) - t^{(i)} \right)^2$$

- Minimize $E(m, b)$ using gradient descent:

$$m, b \leftarrow m, b - \alpha \nabla E$$

- Stop when gradient ≈ 0 or E is below threshold.

6.7 Vector Extensions

- Input vector $x \in \mathbb{R}^n$, output scalar t:
$$y = w^T x + b$$
- Error:
$$E(w, b) = \frac{1}{M} \sum_{i=1}^{M} \left(w^T x^{(i)} + b - t^{(i)} \right)^2$$
- Gradient descent over all w_j and b

6.8 Multidimensional Outputs

- If $t \in \mathbb{R}^K$, learn K independent regressors (or matrix form):
$$y = Wx + b, \quad E = \frac{1}{M} \sum_i \|Wx^{(i)} + b - t^{(i)}\|^2$$

6.9 Need for Nonlinear Approximators

- Linear regression fails on nonlinear mappings.
- Motivation for Neural Networks: represent arbitrary functions.
- Idea: Approximate complex functions using piecewise linear segments.
- **Kolmogorov Approximation Theorem:** Any continuous function on a compact domain can be approximated arbitrarily well by a neural network with enough hidden units.

6.10 Neural Network Architecture

- Each unit: linear regressor + nonlinearity (e.g., ReLU).
- Layers:
  - **Input layer:** raw input vector x
  - **Hidden layers:** $h = \phi(Wx + b)$
  - **Output layer:** linear combination of hidden units
- Activation function (nonlinearity): typically ReLU:
$$\phi(z) = \max(0, z)$$
- **ReLU:** $\phi(z) = \max(0, z)$ — sparse activation, fast, but risk of "dying neurons".
- **Leaky ReLU:** $\phi(z) = \max(az, z)$ where $a \in (0, 1)$ — avoids zero gradients.

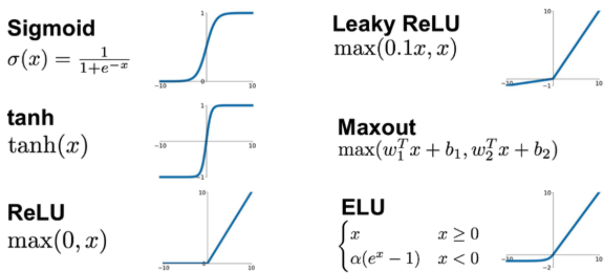

Figure 5: Activation functions used in neural networks, including Sigmoid, Tanh, ReLU, Leaky ReLU, Maxout, and ELU.

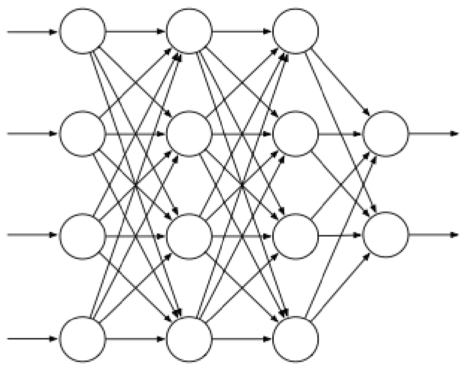

Figure 6: A generic feedforward ANN with four input units, two output units, and two hidden layers.

6.11 Training Neural Networks

- Objective:

$$E(\theta) = \frac{1}{M}\sum_{i=1}^{M}(f_\theta(x^{(i)}) - t^{(i)})^2$$

- θ includes all weights and biases across layers.
- Use gradient descent:

$$\theta \leftarrow \theta - \alpha \nabla_\theta E(\theta)$$

- Use autodiff libraries (e.g., PyTorch, TensorFlow) to compute $\nabla_\theta E$

6.12 Neural Network Properties

- Approximate nonlinear mappings.
- Require tuning of:
  - Number of layers
  - Number of units per layer
- Non-quadratic loss surface: many local minima.
- Training may converge to suboptimal solution depending on initialization.
- **Architecture Refinement Insight:** Increase layers/units until error drops below threshold; balance capacity and overfitting.

6.13 Practicalities

- Use libraries (PyTorch, TensorFlow) to:
  - Define network architecture
  - Compute gradients
  - Automate training
- Nonlinearity: use ReLU (or leaky ReLU) throughout.
- Empirically adjust size/depth to control approximation quality.

Lecture 7: F.Approx (Neural Networks) and RL; DQNs

7.1 Motivation

- In classical control (SCT), dynamic programming uses known model p to compute π^*.
- In RL, we use methods like Monte Carlo (MC) and TD(0).
- Goal: learn $q_\pi(s, a)$ using experience.
- MC: Use full episode returns to estimate $q_\pi(s, a)$.
- TD(0): Bootstraps from current q_π estimates.

7.2 Q as a Function

- Think of $q_\pi(s, a)$ as a function mapping from (s, a) to $\mathbb{R}$.
- Three function representations:
  1. Table
  2. Closed-form expression
  3. Function approximator (e.g., Neural Network)
- Tables work if domain is small.
- For large/continuous spaces, use function approximation.

7.3 Approximation via Data

- Get training data (x, t) from experience.
- Approximate function f from this data.
- Train via regression:

$$E = \frac{1}{M}\sum_{i=1}^{M}(f(x^{(i)}) - t^{(i)})^2$$

- Use linear regression or neural networks.

7.4 Gradient Descent

- Use gradient descent to minimize E:

$$\theta \leftarrow \theta - \alpha \nabla E$$

- Stop when $\nabla E \approx 0$ or $E < \epsilon$.

7.5 Vector Regression and Neural Networks

- Input: $x \in \mathbb{R}^n$, Output: $t \in \mathbb{R}^K$
- Linear model:

$$y = Wx + b, \quad E = \frac{1}{M}\sum_i \|Wx^{(i)} + b - t^{(i)}\|^2$$

- Neural Networks:
  - Add hidden layers with nonlinear activations (e.g., ReLU).
  - Stack layers to increase representational power.

7.6 NN and Value Function Approximation

- Define neural network $Q_\theta(s, a)$ to approximate the action-value function.
- Get training targets $G(s, a)$ from experience.
- Learn Q_θ by minimizing error:

$$E = \frac{1}{M}\sum_i (Q_\theta(s, a) - G(s, a))^2$$

7.7 TD(0) vs MC Targets

- MC: Use full return: $G = R + \gamma R' + \gamma^2 R'' + \dots$
- TD(0): Use one-step return:
$$G = R + \gamma Q_\theta(s', a')$$
- TD target bootstraps from next value estimate.

7.8 Semi-Gradient Update

- Compute gradient w.r.t. prediction $Q_\theta(s, a)$:
$$\nabla_\theta E = 2(Q_\theta(s, a) - G(s, a))\nabla_\theta Q_\theta(s, a)$$
- In TD(0), $G(s, a)$ includes $Q_\theta(s', a')$, but we do not backpropagate through s'.
- This is called a semi-gradient update — improves stability.

7.9 Deep Q-Learning (DQN)

- Use two networks:
  - Q_θ: Online network (updated frequently)
  - Q_{target}: Target network (frozen, updated periodically)
- Define target:
$$G = R + \gamma \max_{a'} Q_{\text{target}}(s', a')$$
- Minimize TD error:
$$E = (Q_\theta(s, a) - G)^2$$
- Periodically sync weights: $\theta_{\text{target}} \leftarrow \theta$
- *Reason: Freezing target network stabilizes learning by decoupling prediction and bootstrap source.*

7.10 Experience Replay

- Store transitions (s, a, r, s') in a replay buffer.
- Sample mini-batches randomly for training.
- Advantages:
  - Breaks correlation in sequential data.
  - Reuses past data for improved sample efficiency.
  - Enables mini-batch SGD training.

7.11 DQN Enhancements (Summary)

- **Target Network:** Use separate Q_{target} to compute target values, update periodically.
- **Experience Replay:** Buffer stores past transitions, samples mini-batches to stabilize and decorrelate training.
- **Gradient Clipping:** Clip large TD errors to avoid destabilizing weight updates.
- **Markov Approximation via History Stacking:**
  - In environments like Atari, stack k frames to create approximate Markovian state input.

Lecture 8:Model learning and Planning; MCTS; Alpha-Zero; Mu-Zero

8.1 Model Components and Goals

- **Model-Based RL** learns a model of the environment:

$$\hat{P}(s', r \mid s, a)$$

- Break into two learning tasks:
  1. Predict next state: $s, a \to s'$
  2. Predict reward: $s, a \to r$
- Learns from transitions: (s, a, r, s')
- *Motivation: Model enables simulation and planning, reducing reliance on real environment interactions.*

8.2 Recall: Function Approximation with NN

- Neural networks used for both regression and classification.
- Regression: Predict real values (e.g., reward).
- Classification: Predict discrete next states.
- Training via gradient descent on:

$$E = (f_\theta(x) - t)^2 \quad \text{or} \quad E = -\sum t_i \log(y_i)$$

8.3 Cross-Entropy for Classification

- Softmax output: $\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$
- Loss:

$$E = -\sum_{i=1}^{N} t_i \log(\hat{y}_i)$$

- Used to train networks predicting categorical s' given (s, a)

8.4 Output Representations

- Discrete s' represented as one-hot vector in $\{0, 1\}^N$
- NN learns to predict softmax distribution over s'
- Training label is the observed s' one-hot vector

8.5 Reuse of Experience

- Store transitions (s, a, r, s') in replay memory.
- Enables reuse of real experience for training.
- Important when experience is expensive or limited.
- *Model-based learning allows for experience efficiency.*

8.6 Role of the Model

- Once trained, the model can:
  1. Perform planning (e.g., Dynamic Programming with $\hat{P}$)
  2. Generate simulated rollouts to train a value function
- Model-based RL reduces reliance on real environment interactions.

8.7 Motivation for Simulation

- Real-world interaction is slow and costly.
- Use model $\hat{P}$ to simulate the consequences of actions.
- Enables more efficient policy improvement and planning.

8.8 Monte Carlo Tree Search (MCTS)

- Combines:
  - Monte Carlo estimation
  - Tree search over action sequences
- Each node: represents a state
- Each edge: represents an action and transition
- Nodes store:
  - Visit count $N(s, a)$
  - Average return $\hat{Q}(s, a)$

8.9 Selection Strategy: UCB1

- **UCB1 metric:**

$$\text{UCB1}(s, a) = \hat{Q}(s, a) + c \cdot \sqrt{\frac{\ln N(s)}{N(s, a)}}$$

- Balances exploration (uncertainty) and exploitation (value).
- c controls exploration strength.

8.10 MCTS Simulation and Backpropagation

- **Simulation (Rollout):**
  - Roll forward using current policy or random rollout.
  - Use model $\hat{P}$ to generate next state and reward.
- **Backpropagation:**
  - At end of rollout, compute total return G
  - Propagate G back up the tree to update:
    - $\hat{Q}(s, a) \leftarrow$ average of returns
    - $N(s, a) \leftarrow$ increment visit count

8.11 Final Action Selection

- After many simulations:
  - Choose action a with the highest visit count from the root
- $a^* = \arg\max_a N(s_0, a)$

8.12 Summary of MCTS Advantages

- Uses a learned model to simulate efficiently.
- Trades real environment interaction for computational planning.
- Effective in domains like Go, Chess, and LLMs (e.g., MuZero).

Lecture 9: Policy Gradient: REINFORCE and Actor-Critic

9.1 From MDP to Non-MDP

- **Standard MDP episode:** $s_0, a_0, r_1, s_1, a_1, r_2, \ldots$
- **Non-Markovian setting:**
  - Observations are insufficient to identify the full state
  - Use a representation network: $h : o \to \hat{s}$
  - Compute latent state: $\hat{s}_0 = h(o_0)$

A sequence of observations can help disambiguate hidden state — a key idea in non-Markovian RL.

9.2 U_0 Architecture (MuZero-inspired)

- **Representation network** h: maps $o \to \hat{s}$
- **Dynamics network** g: $(\hat{s}, a) \to (\hat{s}', \hat{r})$
- **Prediction network** f: $\hat{s} \to (\hat{\pi}, \hat{v})$
- These networks replace traditional simulators with learnable components

9.3 U_0 Tree Search (MCTS)

- Expand in latent space:
  - Use g to simulate latent transitions
  - Use f to predict $\hat{v}, \hat{\pi}$
- Guided by Upper Confidence Bound (UCB1):

$$a^* = \arg\max \left(\hat{Q}(s, a) + \text{bonus from } \hat{\pi} \right)$$

- Monte Carlo Tree Search is efficient — no simulator needed

9.4 Replay Buffer and Learning

- Store sequences: $o_0, \pi_0, a_0, r_1, o_1, \pi_1, \ldots$
- Sample trajectory segments of length L
- Roll out in latent space:

$$\hat{s}_0 \xrightarrow{a_0} \hat{s}_1 \xrightarrow{a_1} \hat{s}_2 \ldots$$

- Compare predictions vs targets:
  - $\hat{r}_t$ vs r_t for reward loss
  - $\hat{v}(\hat{s}_t)$ vs actual return
  - $\hat{\pi}(\hat{s}_t)$ vs stored π_t from MCTS
- Learning involves unrolling and backpropagating across time

9.5 Why Learn Policies?

- Classification network approximates policy:

$$s \to \pi_\theta(a|s)$$

- Output is softmax over actions
- Motivation:
  - Handles non-Markovianity
  - Needed in MuZero to train $\hat{\pi}$
  - Basis for LLMs (e.g. DeepSeek-VL, DeepSeek-RL)

9.6 Policy Gradient Motivation

- Value functions fail when state information is incomplete
- Stochastic policies can outperform deterministic ones
- Example: A state aliasing setup where optimal behavior is to go right 60% and left 40%

9.7 Policy Gradient Framework

$$J(\theta) = V^{\pi_\theta}(s_0) \quad \text{(performance measure)}$$
$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \quad \text{(gradient ascent)}$$

9.8 Policy Gradient Theorem

$$\nabla J(\theta) \propto \mathbb{E}\left[G_t \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

No need for true value function. Gradient driven by return-weighted log-likelihood.

9.9 REINFORCE Algorithm (Monte Carlo PG)

1. Initialize policy network π_θ
2. For each episode:
    - Collect full trajectory: $s_0, a_0, r_1, s_1, \ldots$
    - For each t:
        - Compute return $G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$
        - Update parameters:

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

9.10 Probability Distribution View

$$\nabla J(\theta) = \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi_\theta(a|s)$$

Improves actions proportionally to their expected value under visitation frequency.

Lecture 10: RL and LLMs: RLHF; RL-only LLMs (e.g., Deepseek-R1-zero)

10.1 Policy Gradient Recap

- Policy is a neural network: $\pi_\theta(a|s)$
- Define objective: $J(\theta) = V^{\pi_\theta}(s_0)$
- Gradient:

$$\nabla J(\theta) \propto \mathbb{E}_\pi \left[G_t \cdot \nabla_\theta \log \pi_\theta(a_t|s_t) \right]$$

- Gradient ascent:

$$\theta \leftarrow \theta + \alpha \nabla J(\theta)$$

10.2 LLMs as Agents

- LLM = agent
- Query $\rightarrow$ initial state s_0
- Token generation = sequential actions a_t
- Environment provides next state: query + generated tokens
- Reward $R_T \equiv G$ only at the end (episodic reward)

10.3 LLM Architecture

- Input: tokenized sequence $t_1, \ldots, t_N$
- Embedding: $t_i \in \mathbb{R}^e$
- Transformer NN outputs logits $\rightarrow$ softmax over vocabulary
- Sample next token $\sim \pi_\theta$; repeat until END or context limit

10.4 Training Phases

I. Unsupervised Pretraining

- Predict next token using cross-entropy:

$$-\sum y_i \log \hat{y}_i$$

- No human labels needed

II. Supervised Fine-Tuning (SFT)

- Human-labeled (query, response) pairs
- Use teacher forcing for loss minimization

10.5 RL-Based Fine-Tuning

I. RLHF (Reinforcement Learning with Human Feedback)

- Train reward model ρ (neural net) using human preferences
- Fine-tune LLM using:
  - REINFORCE or PPO
  - Optionally train value function $V(s)$ (actor-critic)
  - Advantage: $A(s, a) = Q(s, a) - V(s)$
- *Training $V(s)$ requires a large additional NN, often same size as the LLM*

II. DeepSeek / GRPO (Group Relative Policy Optimization)

- No human-labeled data or reward model NN
- No value network $V(s)$
- Rule-based reward model (e.g., correctness, formatting)

10.6 GRPO Mechanism

- For each query:
  - Sample G responses: $a^{(1)}, \ldots, a^{(G)}$
  - Compute rule-based rewards $r_1, \ldots, r_G$
  - Define **pseudo-advantage**:

$$A_i = \frac{r_i - \text{mean}(\vec{r})}{\text{std}(\vec{r})}$$

- Use PPO-style objective:

$$J(\theta) = \frac{1}{G} \sum_i \min\left(r_i A_i,\ \text{clip}(r_i, 1-\epsilon, 1+\epsilon) A_i\right)$$

- *Normalized advantage replaces the need for $V(s)$ or $Q(s,a)$ estimation*

10.7 Combined PPO + Actor-Critic Update (RLHF)

- Define:

$$r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, \quad A_t = Q(s_t, a_t) - V(s_t)$$

- Final PPO update:

$$\theta \leftarrow \theta + \alpha \cdot \min\left(r_t A_t,\ \text{clip}(r_t, 1-\epsilon, 1+\epsilon) A_t\right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

- Clipping stabilizes updates to avoid destroying pretrained policy structure

10.8 Comparison Table

	RLHF	GRPO
Reward	Human + Neural Net	Rule-based algorithm
Value Network $V(s)$	Optional (large)	None
Stability	PPO	PPO
Advantage	$Q - V$ (critic)	Normalized reward