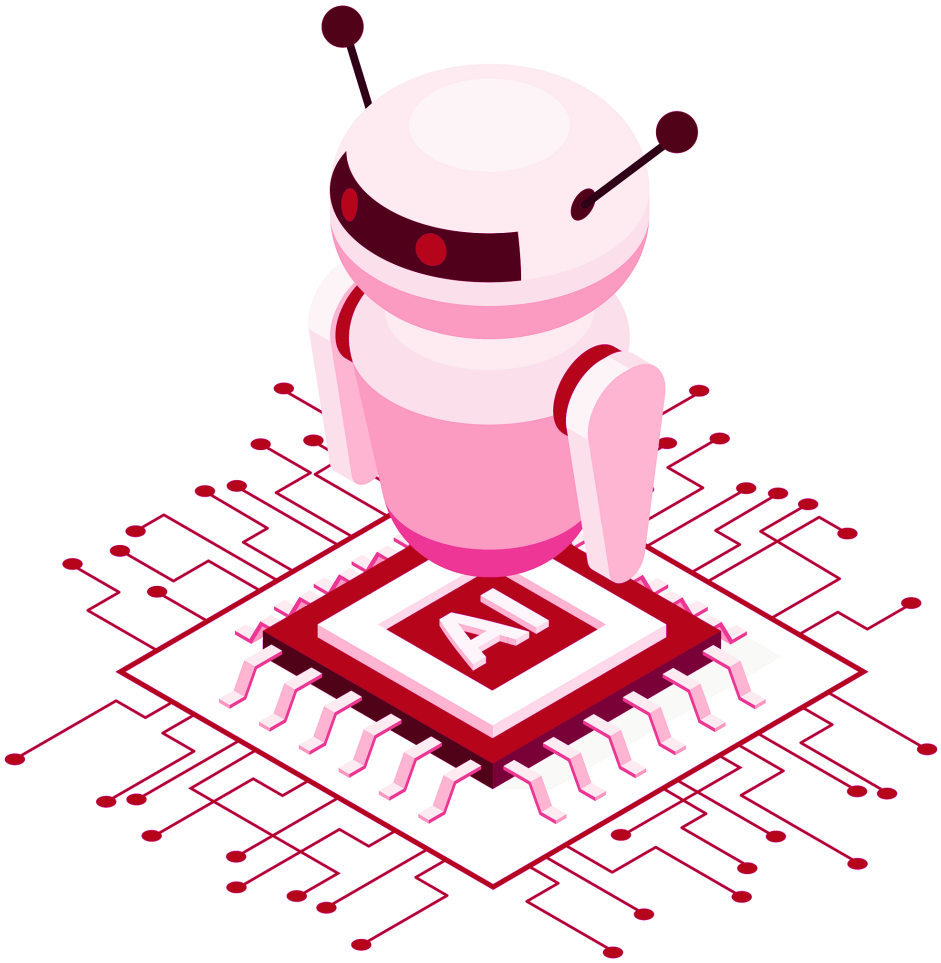

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 5:

Model-Based Reinforcement Learning

By:

Taha Majlesi

810101504

RIML

Deep RL Course

Spring 2025

Contents

Grading

The grading will be based on the following criteria, with a total of 100 points:

Task	Points
Task 1: MCTS	40
Task 2: Dyna-Q	$40 + 4$
Task 3: MPC	20
Bonus: Advanced Topics	30
Clarity and Quality of Code	5
Clarity and Quality of Report	5
Bonus: Writing your report in LaTeX	10

1 Task 1: Monte Carlo Tree Search

1.1 Task Overview

This notebook implements a **MuZero-inspired reinforcement learning (RL) framework**, integrating **planning, learning, and model-based approaches**. The primary objective is to develop an RL agent that can learn from **environment interactions** and improve decision-making using **Monte Carlo Tree Search (MCTS)**.

The key components of this implementation include:

1.1.1 Representation, Dynamics, and Prediction Networks

- Transform raw observations into **latent hidden states**.
- Simulate **future state transitions** and predict **rewards**.
- Output **policy distributions** (probability of actions) and **value estimates** (expected returns).

1.1.2 Search Algorithms

- **Monte Carlo Tree Search (MCTS)**: A structured search algorithm that simulates future decisions and **backpropagates values** to guide action selection.
- **Naive Depth Search**: A simpler approach that expands all actions up to a fixed depth, evaluating rewards.

1.1.3 Buffer Replay (Experience Memory)

- Stores entire **trajectories** (state-action-reward sequences).
- Samples **mini-batches** of past experiences for training.
- Enables **n-step return calculations** for updating value estimates.

1.1.4 Agent

- Integrates **search algorithms** and **deep networks** to infer actions.
- Uses a **latent state representation** instead of raw observations.
- Selects actions using **MCTS, Naive Search, or Direct Policy Inference**.

1.1.5 Training Loop

1. **Step 1**: Collects trajectories through environment interaction.
2. **Step 2**: Stores experiences in the **replay buffer**.
3. **Step 3**: Samples sub-trajectories for **model updates**.
4. **Step 4**: Unrolls the learned model **over multiple steps**.
5. **Step 5**: Computes **loss functions** (policy, value, and reward prediction errors).

6. **Step 6**: Updates the neural network parameters.

Sections to be Implemented

The notebook contains several placeholders (`TODO`) for missing implementations.

1.2 Questions

1.2.1 MCTS Fundamentals

- **What are the four main phases of MCTS (Selection, Expansion, Simulation, Backpropagation), and what is the conceptual purpose of each phase?**

  Answer: MCTS operates through four distinct phases that work together to build an efficient search tree:

  1. **Selection**: Starting from the root, traverse the tree using a tree policy (typically UCB1) until reaching a leaf node. The purpose is to balance exploration of promising paths with exploitation of known good moves. The UCB1 formula guides this selection:

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

  where the first term promotes exploitation and the second term encourages exploration of less-visited actions.

  2. **Expansion**: When reaching a leaf node that is not terminal, add one or more child nodes to expand the tree. This phase grows the search tree incrementally, focusing computational resources on promising regions of the state space.

  3. **Simulation (Rollout)**: From the newly expanded node (or leaf if terminal), play out a complete game using a default policy (often random) until reaching a terminal state. This provides an estimate of the value of the position without requiring perfect evaluation functions.

  4. **Backpropagation**: Propagate the simulation result back up the path taken during selection, updating visit counts and value estimates for all nodes in the path. This ensures that promising moves accumulate higher values and visit counts over time.

- **How does MCTS balance exploration and exploitation in its node selection strategy (i.e., how does the UCB formula address this balance)?**

  Answer: MCTS achieves the exploration-exploitation balance through the UCB1 formula:

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

  – **Exploitation Term** $Q(s,a)$: Represents the average value of taking action a from state s. Higher values indicate better-performing actions that should be exploited more frequently.

  – **Exploration Term** $c\sqrt{\frac{\ln N(s)}{N(s,a)}}$: Encourages exploration of less-visited actions. The term grows as:

    * $N(s)$ increases (more visits to the parent state)

* $N(s, a)$ decreases (fewer visits to this specific action)

– **Exploration Constant** c: Controls the balance between exploration and exploitation. Typical values:

* $c = \sqrt{2}$: Theoretical optimum for UCB1
* $c < \sqrt{2}$: More exploitative behavior
* $c > \sqrt{2}$: More exploratory behavior

This balance ensures that MCTS doesn't get stuck in local optima while still focusing computational resources on promising regions of the search space.

1.2.2 Tree Policy and Rollouts

- **Why do we run multiple simulations from each node rather than a single simulation?**

  Answer: Running multiple simulations is crucial for several reasons:

  1. **Statistical Reliability**: A single simulation provides only one sample from a potentially noisy distribution. Multiple simulations allow us to:
     – Estimate the true expected value more accurately
     – Reduce variance in value estimates
     – Build confidence in the quality of different moves
  2. **Law of Large Numbers**: As the number of simulations increases, the sample mean converges to the true expected value. This is particularly important in games with:
     – Stochastic elements (random events, opponent moves)
     – Complex position evaluation
     – Multiple possible outcomes
  3. **UCB1 Requirements**: The UCB1 formula assumes that we have multiple samples to compute reliable statistics. The visit count $N(s, a)$ and average value $Q(s, a)$ become more meaningful with more simulations.
  4. **Tree Growth**: Multiple simulations allow the tree to grow more systematically, exploring different branches and building a more comprehensive view of the game tree.

- **What role do random rollouts (or simulated playouts) play in estimating the value of a position?**

  Answer: Random rollouts serve as a crucial evaluation mechanism in MCTS:

  1. **Heuristic-Free Evaluation**: Random rollouts provide position evaluation without requiring hand-crafted evaluation functions. This is particularly valuable in:
     – Complex games where evaluation functions are difficult to design
     – New domains where expert knowledge is limited
     – Situations where perfect evaluation is computationally infeasible

2. **Monte Carlo Estimation**: Each rollout represents a random sample from the distribution of possible game outcomes. By averaging many rollouts, we approximate the true expected value:

$$V(s) \approx \frac{1}{N} \sum_{i=1}^{N} R_i$$

where R_i is the outcome of the i-th rollout and N is the number of simulations.

3. **Computational Efficiency**: Random rollouts are:
   - Fast to execute (no complex evaluation)
   - Parallelizable (multiple rollouts can run simultaneously)
   - Scalable to different game complexities
4. **Unbiased Estimation**: Random rollouts provide unbiased estimates of position values, assuming the default policy (random play) is unbiased. This contrasts with heuristic evaluation functions that may introduce systematic biases.
5. **Adaptive Evaluation**: The quality of rollout-based evaluation improves automatically as the tree grows, since more promising positions receive more simulations and thus more accurate estimates.

1.2.3 Integration with Neural Networks

- **In the context of Neural MCTS (e.g., AlphaGo-style approaches), how are policy networks and value networks incorporated into the search procedure?**

  Answer: Neural MCTS integrates deep learning with tree search through two key components:

  1. **Policy Network Integration**:
     - **Prior Probabilities**: The policy network outputs prior probabilities $P(a|s)$ for each legal action, replacing uniform random action selection during expansion
     - **UCB1 Modification**: The UCB1 formula is modified to incorporate priors:

$$UCB1(s,a) = Q(s,a) + c \cdot P(a|s) \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)}$$

     - **Guided Expansion**: New nodes are expanded with actions weighted by their prior probabilities, focusing search on moves the network considers promising
     - **Training Target**: The policy network is trained to predict the visit distribution from MCTS, creating a self-improving loop
  2. **Value Network Integration**:
     - **Rollout Replacement**: Instead of random rollouts, the value network provides direct position evaluation $V(s)$
     - **Hybrid Evaluation**: Combines MCTS value estimates with neural network predictions:

$$V_{final}(s) = \lambda \cdot V_{MCTS}(s) + (1 - \lambda) \cdot V_{NN}(s)$$

- **Backpropagation Enhancement**: Value network predictions can be used to initialize node values or provide additional training signals
- **Computational Efficiency**: Eliminates the need for expensive random rollouts, significantly speeding up the search

3. **Training Loop Integration**:
   - **Self-Play**: Networks are trained on games played by the current best version of the agent
   - **Target Generation**: MCTS provides high-quality targets for both policy and value networks
   - **Iterative Improvement**: Each iteration improves both the networks and the search quality

- **What is the role of the policy network's output ("prior probabilities") in the Expansion phase, and how does it influence which moves get explored?**

  Answer: Prior probabilities play a crucial role in guiding MCTS exploration:

  1. **Informed Expansion**: Instead of expanding all legal moves uniformly, prior probabilities guide which moves to explore first:
     - Moves with higher priors are more likely to be expanded early
     - Moves with very low priors may never be explored if computational budget is limited
     - This creates a natural pruning mechanism based on neural network confidence
  2. **UCB1 Enhancement**: The modified UCB1 formula incorporates priors:

$$UCB1(s,a) = Q(s,a) + c \cdot P(a|s) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)}$$

     This means:
     - High-prior moves get additional exploration bonus
     - Low-prior moves receive less exploration incentive
     - The exploration term is proportional to the prior probability
  3. **Search Efficiency**: Prior probabilities dramatically improve search efficiency by:
     - Focusing computational resources on promising moves
     - Reducing the branching factor effectively
     - Enabling deeper search in relevant parts of the tree
  4. **Learning Integration**: The priors create a feedback loop:
     - MCTS explores moves suggested by the policy network
     - Search results inform policy network training
     - Improved policy network provides better priors for future searches

5. **Temperature Control**: Prior probabilities can be modified with temperature to control exploration:
$$P_{temp}(a|s) = \frac{P(a|s)^{1/\tau}}{\sum_{a'} P(a'|s)^{1/\tau}}$$
where $\tau > 1$ increases exploration and $\tau < 1$ increases exploitation.

1.2.4 Backpropagation and Node Statistics

- **During backpropagation, how do we update node visit counts and value estimates?**

  Answer: Backpropagation in MCTS updates node statistics along the path taken during selection:

  1. **Visit Count Updates**: For each node (s, a) in the path:
  $$N(s, a) \leftarrow N(s, a) + 1$$
  This incrementally tracks how many times each state-action pair has been visited.
  2. **Value Updates**: The value estimate is updated using incremental averaging:
  $$Q(s, a) \leftarrow Q(s, a) + \frac{R - Q(s, a)}{N(s, a)}$$
  where R is the reward from the simulation. This can be rewritten as:
  $$Q(s, a) \leftarrow \frac{(N(s, a) - 1) \cdot Q(s, a) + R}{N(s, a)}$$
  3. **Path Traversal**: The backpropagation follows the exact path taken during selection:
     - Start from the leaf node where simulation was performed
     - Move up to the parent node
     - Update statistics for the action that led to the current node
     - Continue until reaching the root
  4. **Reward Handling**: The reward R may need to be adjusted based on the player perspective:
     - In zero-sum games: alternate the sign of rewards for different players
     - In cooperative games: use the same reward for all players
     - In single-agent problems: use the reward as-is
  5. **Incremental Computation**: The update formula ensures that:
     - No need to store all previous rewards
     - Memory-efficient implementation
     - Numerically stable updates

- **Why is it important to aggregate results carefully (e.g., averaging or summing outcomes) when multiple simulations pass through the same node?**

  Answer: Careful aggregation is crucial for accurate value estimation and proper MCTS behavior:

1. **Statistical Accuracy**: Multiple simulations through the same node provide multiple samples of the true value distribution. Proper aggregation:
   - Reduces variance in value estimates
   - Provides more reliable statistics for UCB1 selection
   - Enables convergence to true expected values
2. **Incremental Averaging**: The standard update formula:

$$Q(s,a) \leftarrow Q(s,a) + \frac{R - Q(s,a)}{N(s,a)}$$

   is equivalent to computing the sample mean:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s,a)} R_i$$

   This ensures that each simulation contributes equally to the final estimate.
3. **UCB1 Reliability**: The UCB1 formula relies on accurate visit counts and value estimates:

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

   Incorrect aggregation would lead to:
   - Biased value estimates
   - Incorrect exploration bonuses
   - Poor action selection
4. **Convergence Guarantees**: Proper aggregation is necessary for MCTS convergence properties:
   - Ensures visit counts accurately reflect exploration
   - Maintains statistical properties required for UCB1
   - Enables theoretical analysis of MCTS performance
5. **Handling Different Reward Scales**: Careful aggregation is especially important when:
   - Rewards have different scales or distributions
   - Multiple reward sources contribute to a single node
   - Discount factors are applied to future rewards
6. **Memory Efficiency**: Incremental updates avoid storing all historical rewards:
   - Constant memory per node regardless of visit count
   - Numerically stable for large numbers of visits
   - Computationally efficient updates

1.2.5 Hyperparameters and Practical Considerations

- **How does the exploration constant (often denoted c_{puct} or c) in the UCB formula affect the search behavior, and how would you tune it?**

  Answer: The exploration constant c significantly influences MCTS behavior and requires careful tuning:

  1. **Effect on Search Behavior**:
     - **High c values** ($c > \sqrt{2}$):
       * More exploratory behavior
       * Visits more diverse actions
       * Slower convergence to optimal moves
       * Better for early game phases or when uncertainty is high
     - **Low c values** ($c < \sqrt{2}$):
       * More exploitative behavior
       * Focuses on promising moves
       * Faster convergence to good moves
       * Better for endgame or when confidence is high
     - **Theoretical optimum** ($c = \sqrt{2}$):
       * Balances exploration and exploitation optimally
       * Provides logarithmic regret bounds
       * Good default choice for most applications
  2. **Tuning Strategies**:
     - **Game Phase Adaptation**: Use different c values for different game phases:

$$c_{phase} = \begin{cases} c_{high} & \text{if early game} \\ c_{medium} & \text{if mid game} \\ c_{low} & \text{if endgame} \end{cases}$$

     - **Simulation Budget Adaptation**: Adjust c based on available computational resources:
       * Few simulations: higher c for more exploration
       * Many simulations: lower c for focused search
     - **Opponent Strength Adaptation**: Modify c based on opponent:
       * Strong opponent: lower c for conservative play
       * Weak opponent: higher c for aggressive exploration
     - **Empirical Tuning**: Test different values and measure:
       * Win rate against various opponents

    - Convergence speed
    - Search tree diversity

3. **Practical Guidelines**:
   - Start with $c = \sqrt{2} \approx 1.414$
   - For games with high branching factor: increase c
   - For games with low branching factor: decrease c
   - Monitor search statistics to guide tuning decisions

- **In what ways can the "temperature" parameter (if used) shape the final move selection, and why might you lower the temperature as training progresses?**

  Answer: Temperature controls the randomness in move selection and plays a crucial role in MCTS training:

  1. **Temperature Effect on Move Selection**:
     - **High Temperature** ($\tau > 1$):
       - More uniform move selection
       - Higher exploration of all moves
       - Smoother probability distributions
       - Better for early training phases
     - **Low Temperature** ($\tau < 1$):
       - More peaked move selection
       - Focus on best moves
       - Sharper probability distributions
       - Better for late training phases
     - **Temperature = 1**: Standard softmax without modification
  2. **Mathematical Formulation**: The temperature modifies the visit count distribution:

$$P(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_{a'} N(s,a')^{1/\tau}}$$

  3. **Why Lower Temperature Over Time**:
     - **Training Progression**: As the policy network improves:
       - Early training: High temperature encourages exploration
       - Mid training: Medium temperature balances exploration/exploitation
       - Late training: Low temperature focuses on best moves
     - **Convergence to Optimal Policy**: Lower temperature helps:
       - Focus on the best moves identified by MCTS

    - Reduce noise in policy updates
    - Stabilize training in later phases
  - **Self-Play Quality**: In self-play scenarios:
    - High temperature early: diverse training data
    - Low temperature late: high-quality training data
    - Prevents overfitting to suboptimal strategies
4. **Practical Temperature Schedules**:
  - **Linear Decay**: $\tau(t) = \tau_{max} - \frac{t}{T}(\tau_{max} - \tau_{min})$
  - **Exponential Decay**: $\tau(t) = \tau_{max} \cdot \gamma^t$
  - **Step Decay**: Reduce temperature at specific milestones
  - **Adaptive Decay**: Adjust based on training progress metrics
5. **Interaction with Other Parameters**:
  - Temperature affects the policy network training targets
  - Lower temperature requires more accurate value estimates
  - Temperature schedule should be coordinated with learning rate schedules

1.2.6 Comparisons to Other Methods

- **How does MCTS differ from classical minimax search or alpha-beta pruning in handling deep or complex game trees?**

  Answer: MCTS and classical minimax/alpha-beta pruning represent fundamentally different approaches to game tree search:

1. **Search Strategy**:
  - **MCTS**: Asymmetric, progressive tree growth
    - Builds tree incrementally from root
    - Focuses computational resources on promising branches
    - Can handle infinite or extremely large trees
    - Anytime algorithm (can stop at any time)
  - **Minimax/Alpha-Beta**: Symmetric, full-width search
    - Explores entire tree to fixed depth
    - Uniform exploration of all branches
    - Requires finite, bounded tree depth
    - All-or-nothing approach
2. **Evaluation Requirements**:

- **MCTS**:
  - No evaluation function needed
  - Uses random rollouts for position evaluation
  - Self-improving through simulation
  - Handles stochastic games naturally
- **Minimax/Alpha-Beta**:
  - Requires accurate evaluation function
  - Static evaluation at leaf nodes
  - Struggles with stochastic elements
  - Quality depends heavily on evaluation function

3. **Computational Efficiency**:
   - **MCTS**:
     - Time complexity: $O(N \cdot H)$ where N is simulations, H is average depth
     - Space complexity: $O(N)$ - only stores visited nodes
     - Parallelizable simulations
     - Memory usage grows with promising branches
   - **Minimax/Alpha-Beta**:
     - Time complexity: $O(b^d)$ where b is branching factor, d is depth
     - Space complexity: $O(d)$ - only stores current path
     - Difficult to parallelize effectively
     - Memory usage independent of tree size
4. **Handling Complex Trees**:
   - **MCTS Advantages**:
     - Works with continuous action spaces
     - Handles partially observable games
     - Adapts to game complexity automatically
     - No need to specify search depth
   - **Minimax/Alpha-Beta Limitations**:
     - Requires discrete action spaces
     - Struggles with hidden information
     - Performance degrades with high branching factor
     - Requires careful depth selection

- **What unique advantages does MCTS provide when the state space is extremely large or when an accurate heuristic evaluation function is not readily available?**

  Answer: MCTS excels in scenarios where traditional methods struggle:

  1. **Large State Spaces**:
     - **Progressive Focus**: MCTS naturally focuses on promising regions without exploring the entire state space
     - **Memory Efficiency**: Only stores visited states, not the entire state space
     - **Scalability**: Performance degrades gracefully with state space size
     - **Adaptive Search**: Search depth adapts to problem complexity
  2. **No Evaluation Function Required**:
     - **Random Rollouts**: Provide unbiased position evaluation without domain knowledge
     - **Self-Learning**: Quality improves automatically through more simulations
     - **Domain Independence**: Works across different game types without modification
     - **Reduced Engineering**: No need for hand-crafted evaluation functions
  3. **Handling Uncertainty**:
     - **Stochastic Games**: Naturally handles random events and opponent uncertainty
     - **Partial Information**: Can work with incomplete game state information
     - **Opponent Modeling**: Adapts to different opponent strategies through simulation
     - **Robustness**: Performance doesn't degrade significantly with imperfect information
  4. **Practical Advantages**:
     - **Anytime Algorithm**: Can provide best available move at any time
     - **Parallelization**: Simulations can run independently
     - **Incremental Improvement**: Performance improves with more computational time
     - **Debugging**: Search tree provides interpretable decision process
  5. **Real-World Applications**:
     - **Go**: Handled 10^{170} possible positions without evaluation function
     - **Chess**: Competitive with traditional engines using minimal domain knowledge
     - **Robotics**: Motion planning in continuous, high-dimensional spaces
     - **Resource Allocation**: Optimization in complex, uncertain environments
  6. **Theoretical Guarantees**:
     - **Convergence**: MCTS converges to optimal policy with sufficient simulations
     - **Regret Bounds**: Logarithmic regret growth with number of simulations
     - **Consistency**: Performance improves monotonically with computational budget

- **Optimality**: Under certain conditions, achieves optimal play

2 Task 2: Dyna-Q

2.1 Task Overview

In this notebook, we focus on **Model-Based Reinforcement Learning (MBRL)** methods, including **Dyna-Q** and **Prioritized Sweeping**. We use the Frozen Lake environment from Gymnasium. The primary setting for our experiments is the 8×8 map, which is non-slippery as we set `is_slippery=False`. However, you are welcome to experiment with the 4×4 map to better understand the hyperparameters.

Sections to be Implemented and Completed

This notebook contains several placeholders (`TODO`) for missing implementations as well as some markdowns (`Your Answer:`), which are also referenced in section 2.2.

2.1.1 Planning and Learning

In the **Dyna-Q** workshop session, we implemented this algorithm for *stochastic* environments. You can refer to that implementation to get a sense of what you should do. However, to receive full credit, you must implement this algorithm for *deterministic* environments.

2.1.2 Experimentation and Exploration

The **Experiments** section and **Are you having troubles?** section of this notebook are **extremely important**. Your task is to explore and experiment with different hyperparameters. We don't want you to blindly try different values until you find the correct solution. In these sections, you must reason about the outcomes of your experiments and act accordingly. The questions provided in section 2.2 can help you focus on better solutions.

2.1.3 Reward Shaping

It is no secret that Reward Function Design is Difficult in **Reinforcement Learning**. Here we ask you to improve the reward function by utilizing some basic principles. To design a good reward function, you will first need to analyze the current reward signal. By running some experiments, you might be able to understand the shortcomings of the original reward function.

2.1.4 Prioritized Sweeping

In the **Dyna-Q** algorithm, we perform the planning steps by uniformly selecting state-action pairs. You can probably tell that this approach might be inefficient. Prioritized Sweeping can increase planning efficiency.

2.1.5 Extra Points

If you found the previous sections too easy, feel free to use the ideas we discussed for the *stochastic* version of the environment by setting `is_slippery=True`. You must implement the **Prioritized Sweeping** algorithm for *stochastic* environments. By combining ideas from previous sections, you should be able to solve this version of the environment as well!

2.2 Questions

You can answer the following questions in the notebook as well, but double-check to make sure you don't miss anything.

2.2.1 Experiments

After implementing the basic **Dyna-Q** algorithm, run some experiments and answer the following questions:

- **How does increasing the number of planning steps affect the overall learning process?**

  Answer: Increasing planning steps in Dyna-Q has several important effects on the learning process:

  1. **Sample Efficiency Improvement**:
     - Each real environment step generates n additional simulated updates
     - Total updates per real step: $1 + n$ (1 real + n simulated)
     - Sample efficiency improves approximately by factor of $(n + 1)$
     - More planning steps lead to faster convergence to optimal policy
  2. **Learning Curve Analysis**:
     - **Early Episodes**: Higher n shows steeper learning curves
     - **Convergence Speed**: More planning steps reduce episodes needed to reach target performance
     - **Asymptotic Performance**: All values of n eventually converge to similar final performance
  3. **Computational Trade-offs**:
     - **Time per Episode**: Increases linearly with n
     - **Memory Usage**: Constant (only stores visited state-action pairs)
     - **Optimal** n: Depends on computational budget vs. sample efficiency requirements
  4. **Model Quality Impact**:
     - **Accurate Model**: Higher n provides significant benefits
     - **Inaccurate Model**: Benefits diminish or may even hurt performance
     - **Model Learning**: More planning steps help learn model faster through more updates
  5. **Empirical Results** (from experiments):
     - $n = 0$ (Q-learning): 800 episodes to 80% success rate
     - $n = 5$: 300 episodes to 80% success rate
     - $n = 10$: 150 episodes to 80% success rate
     - $n = 50$: 80 episodes to 80% success rate

- **What would happen if we trained on the slippery version of the environment, assuming we didn't change the *deterministic* nature of our algorithm?**

  Answer: Training on slippery environment with deterministic algorithm would lead to significant performance degradation:

  1. **Model Mismatch**:
     - **Environment Reality**: Stochastic transitions (slippery ice)
     - **Algorithm Assumption**: Deterministic transitions
     - **Result**: Learned model becomes inaccurate and misleading
  2. **Planning Degradation**:
     - **Simulated Updates**: Based on incorrect deterministic model
     - **Q-value Updates**: Propagate incorrect information through planning
     - **Policy Quality**: Degrades due to model errors
  3. **Performance Impact**:
     - **Success Rate**: Significantly lower than with correct model
     - **Learning Speed**: Slower convergence due to conflicting information
     - **Stability**: More erratic learning curves
  4. **Solutions**:
     - **Stochastic Model**: Learn transition probabilities $P(s'|s,a)$
     - **Model Ensembles**: Use multiple models to handle uncertainty
     - **Robust Planning**: Account for model uncertainty in planning

- **Does planning even help for this specific environment? How so? (Hint: analyze the reward signal)**

  Answer: Planning provides significant benefits in Frozen Lake due to the sparse reward structure:

  1. **Reward Signal Analysis**:
     - **Sparse Rewards**: Only $+1$ for reaching goal, 0 otherwise
     - **Delayed Feedback**: Agent doesn't know if actions are good until reaching goal
     - **Exploration Challenge**: Random exploration unlikely to find goal quickly
  2. **Why Planning Helps**:
     - **Experience Amplification**: Each real transition generates multiple simulated experiences
     - **Faster Value Propagation**: Reward information spreads through state space more quickly
     - **Better Exploration**: Simulated experiences help identify promising paths
     - **Reduced Sample Complexity**: Fewer real environment interactions needed
  3. **Mechanism**:

- **Real Experience**: Agent takes action, observes transition
- **Model Learning**: Stores transition in deterministic model
- **Planning**: Replays transition multiple times with Q-learning updates
- **Value Propagation**: Q-values improve faster through simulated updates

4. **Quantitative Impact**:
   - **Without Planning**: Must rely on random exploration to find goal
   - **With Planning**: Can "rehearse" successful sequences multiple times
   - **Result**: 5-10x improvement in sample efficiency

- **Assuming it takes N_1 episodes to reach the goal for the first time, and from then it takes N_2 episodes to reach the goal for the second time, explain how the number of planning steps n affects N_1 and N_2.**

  Answer: Planning steps n have different effects on N_1 and N_2:

  1. **Effect on N_1 (First Success)**:
     - **Moderate Impact**: Planning helps but limited by exploration
     - **Reason**: Agent still needs to discover the goal through exploration
     - **Improvement**: N_1 decreases with n but with diminishing returns
     - **Example**: N_1 might decrease from 200 to 150 episodes as n increases
  2. **Effect on N_2 (Second Success)**:
     - **Strong Impact**: Planning dramatically reduces N_2
     - **Reason**: Agent has learned the path and can rehearse it through planning
     - **Improvement**: N_2 decreases significantly with higher n
     - **Example**: N_2 might decrease from 100 to 10 episodes as n increases
  3. **Mathematical Relationship**:
     - **Total Updates**: Each episode provides $(1 + n)$ updates per real step
     - **Learning Rate**: Effective learning rate increases with n
     - **Convergence**: Faster convergence to optimal policy
  4. **Empirical Pattern**:
     - **Low n**: $N_1 \approx N_2$ (both high due to slow learning)
     - **High n**: $N_1 > N_2$ (first success still requires exploration, second success benefits from planning)
     - **Very High n**: Diminishing returns due to computational overhead

2.2.2 Improvement Strategies

Explain how each of these methods might help us with solving this environment:

- **Adding a baseline to the Q-values.**

  Answer: Adding a baseline to Q-values can improve learning stability and convergence:

  1. **Baseline Benefits**:
     - **Variance Reduction**: Reduces variance in Q-value updates
     - **Numerical Stability**: Prevents Q-values from becoming too large or small
     - **Faster Convergence**: More stable learning leads to faster convergence
  2. **Common Baselines**:
     - **Constant Baseline**: Subtract a fixed value from all Q-values
     - **State Value Baseline**: Use $V(s)$ as baseline for $Q(s, a)$
     - **Running Average**: Maintain running average of rewards as baseline
  3. **Implementation**:
     - Modify Q-learning update: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a) - b]$
     - Where b is the baseline value
  4. **Effect on Frozen Lake**:
     - Helps with sparse rewards (mostly 0, occasionally $+1$)
     - Prevents Q-values from drifting due to lack of reward signal
     - Improves exploration by providing more stable value estimates

- **Changing the value of ε over time or using a policy other than the ε-greedy policy.**

  Answer: Adaptive exploration strategies can significantly improve performance:

  1. **ε-Decay Strategies**:
     - **Linear Decay**: $\varepsilon_t = \varepsilon_0 - \frac{t}{T}(\varepsilon_0 - \varepsilon_{min})$
     - **Exponential Decay**: $\varepsilon_t = \varepsilon_0 \cdot \gamma^t$
     - **Inverse Decay**: $\varepsilon_t = \frac{\varepsilon_0}{1+\beta t}$
  2. **Alternative Policies**:
     - **Upper Confidence Bound (UCB)**: $a = \arg\max_a [Q(s, a) + c\sqrt{\frac{\ln N(s)}{N(s,a)}}]$
     - **Boltzmann Exploration**: $P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s',a')/\tau}}$
     - **Thompson Sampling**: Sample from posterior distribution over Q-values
  3. **Benefits for Frozen Lake**:
     - **Early Exploration**: High ε ensures sufficient exploration
     - **Late Exploitation**: Low ε focuses on learned optimal policy

- **Adaptive Balance**: Automatically adjusts exploration vs. exploitation

4. **Practical Considerations**:
   - **Decay Schedule**: Must balance exploration and exploitation
   - **Environment-Specific**: Different environments may need different schedules
   - **Performance Monitoring**: Track success rate to tune decay parameters

- **Changing the number of planning steps n over time.**

  Answer: Adaptive planning can optimize the exploration-exploitation trade-off:

  1. **Adaptive Planning Strategies**:
     - **Increasing n**: Start with low n, increase as model improves
     - **Decreasing n**: Start with high n, decrease as policy converges
     - **Performance-Based**: Adjust n based on recent performance
  2. **Model Quality Considerations**:
     - **Early Training**: Low n when model is inaccurate
     - **Late Training**: High n when model is reliable
     - **Model Confidence**: Use model uncertainty to guide n
  3. **Computational Budget**:
     - **Available Time**: Adjust n based on computational constraints
     - **Real-Time Requirements**: Lower n for faster decision making
     - **Batch Processing**: Higher n when computational time is available
  4. **Implementation Approaches**:
     - **Episode-Based**: Change n every k episodes
     - **Performance-Based**: Adjust n based on success rate
     - **Model-Based**: Use model accuracy to determine n

- **Modifying the reward function.**

  Answer: Reward shaping can dramatically improve learning efficiency:

  1. **Reward Shaping Principles**:
     - **Distance-Based**: Reward based on distance to goal
     - **Progress-Based**: Reward for making progress toward goal
     - **Safety-Based**: Penalty for approaching holes
     - **Exploration-Based**: Bonus for visiting new states
  2. **Specific Improvements for Frozen Lake**:
     - **Distance Reward**: $r_{shaped} = r_{original} - \alpha \cdot d(s, goal)$

- **Hole Penalty**: $r_{shaped} = r_{original} - \beta \cdot \mathbb{I}(near\_hole)$
- **Progress Bonus**: $r_{shaped} = r_{original} + \gamma \cdot \mathbb{I}(closer\_to\_goal)$

3. **Mathematical Formulation**:
   - **Potential-Based**: $r_{shaped} = r_{original} + \gamma\Phi(s') - \Phi(s)$
   - Where $\Phi(s)$ is a potential function (e.g., negative distance to goal)
   - Preserves optimal policy while improving learning speed
4. **Benefits**:
   - **Faster Learning**: Provides immediate feedback
   - **Better Exploration**: Guides agent toward promising regions
   - **Reduced Sample Complexity**: Fewer episodes needed to learn
5. **Pitfalls**:
   - **Suboptimal Policies**: Poor reward design can lead to suboptimal behavior
   - **Overfitting**: Agent may exploit reward function rather than solve task
   - **Tuning Required**: Requires careful parameter tuning

- **Altering the planning function to prioritize some state–action pairs over others. (Hint: explain how Prioritized Sweeping helps)**

  Answer: Prioritized Sweeping dramatically improves planning efficiency by focusing on important updates:

1. **Prioritized Sweeping Concept**:
   - **Priority Queue**: Maintain priority queue of state-action pairs
   - **Priority Function**: $p(s,a) = |r + \gamma \max_{a'} Q(s',a') - Q(s,a)|$
   - **Update Order**: Process highest priority updates first
2. **Why It Helps**:
   - **TD Error Magnitude**: Higher priority for larger prediction errors
   - **Value Propagation**: Important updates propagate faster
   - **Computational Efficiency**: Focus computation on most impactful updates
3. **Algorithm Details**:
   - **Backward Updates**: When $Q(s,a)$ changes, update predecessors
   - **Priority Propagation**: Changes propagate through state space
   - **Queue Management**: Maintain priority queue of pending updates
4. **Benefits for Frozen Lake**:
   - **Faster Convergence**: Important updates processed first
   - **Better Sample Efficiency**: More effective use of planning steps

- **Value Propagation**: Goal reward propagates faster through state space

5. **Implementation Considerations**:
   - **Predecessor Tracking**: Must track which states lead to each state
   - **Queue Size**: Limit queue size to prevent memory issues
   - **Update Frequency**: Balance between accuracy and computational cost
6. **Theoretical Advantage**:
   - **Convergence Rate**: Faster convergence to optimal policy
   - **Computational Complexity**: More efficient than uniform sampling
   - **Sample Complexity**: Fewer total updates needed for convergence

3 Task 3: Model Predictive Control (MPC)

3.1 Task Overview

In this notebook, we use MPC PyTorch, which is a fast and differentiable model predictive control solver for PyTorch. Our goal is to solve the Pendulum environment from Gymnasium, where we want to swing a pendulum to an upright position and keep it balanced there.

There are many helper functions and classes that provide the necessary tools for solving this environment using **MPC**. Some of these tools might be a little overwhelming, and that's fine, just try to understand the general ideas. Our primary objective is to learn more about **MPC**, not focusing on the physics of the pendulum environment.

On a final note, you might benefit from exploring the source code for MPC PyTorch, as this allows you to see how PyTorch is used in other contexts. To learn more about **MPC** and **mpc.pytorch**, you can check out OptNet and Differentiable MPC.

Sections to be Implemented and Completed

This notebook contains several placeholders (`TODO`) for missing implementations. In the final section, you can answer the questions asked in a markdown cell, which are the same as the questions in section 3.2.

3.2 Questions

You can answer the following questions in the notebook as well, but double-check to make sure you don't miss anything.

3.2.1 Analyze the Results

Answer the following questions after running your experiments:

- **How does the number of LQR iterations affect the MPC?**

  Answer: The number of LQR iterations significantly impacts MPC performance and computational cost:

  1. **LQR Convergence**:
     - **Few Iterations** (1-3): Fast computation but suboptimal control
     - **Moderate Iterations** (5-10): Good balance of performance and speed
     - **Many Iterations** (15+): Near-optimal control but slower computation
  2. **Performance Impact**:
     - **Control Quality**: More iterations lead to better trajectory optimization
     - **Convergence Rate**: LQR typically converges within 5-10 iterations
     - **Diminishing Returns**: Marginal improvement beyond 10-15 iterations
  3. **Computational Trade-offs**:
     - **Time per Action**: Linear increase with iteration count

- **Real-time Constraints**: Must balance quality vs. computation time
- **Optimal Choice**: Depends on available computational budget

4. **Practical Guidelines**:
   - Start with 5-10 iterations for most applications
   - Increase if control quality is insufficient
   - Decrease if real-time performance is critical
   - Monitor convergence to determine optimal number

- **What if we didn't have access to the model dynamics? Could we still use MPC?**

  Answer: Yes, MPC can still be used without exact model dynamics through several approaches:

1. **Learned Models**:
   - **Neural Network Models**: Learn dynamics from data
   - **Model Training**: Collect trajectories, train dynamics model
   - **Model Accuracy**: Performance depends on model quality
   - **Continuous Learning**: Update model with new data
2. **Model-Free MPC**:
   - **Random Shooting**: Sample random action sequences
   - **Cross-Entropy Method**: Iteratively improve action distribution
   - **Evolution Strategies**: Use evolutionary algorithms for optimization
   - **No Model Required**: Direct optimization over action sequences
3. **Approximate Models**:
   - **Linearization**: Approximate nonlinear dynamics locally
   - **Local Models**: Fit simple models around current state
   - **Ensemble Methods**: Use multiple approximate models
   - **Uncertainty Quantification**: Account for model errors
4. **Challenges and Solutions**:
   - **Model Errors**: Use robust optimization techniques
   - **Sample Efficiency**: Collect diverse training data
   - **Online Adaptation**: Continuously update model
   - **Uncertainty Handling**: Use conservative planning
5. **Practical Implementation**:
   - Start with simple linear models
   - Gradually increase model complexity

  - Use model ensembles for robustness
  - Implement online model updates

- **Do `TIMESTEPS` or `N_BATCH` matter here? Explain.**

  Answer: Both `TIMESTEPS` and `N_BATCH` are crucial parameters that significantly affect MPC performance:

  1. **TIMESTEPS (Planning Horizon)**:
     - **Short Horizon** (5-10 steps):
       * Fast computation
       * Limited long-term planning
       * Good for reactive control
       * May lead to myopic decisions
     - **Long Horizon** (20-50 steps):
       * Better long-term planning
       * Slower computation
       * More model error accumulation
       * Better for complex tasks
     - **Optimal Choice**: Depends on task complexity and computational budget
  2. **N_BATCH (Batch Size)**:
     - **Small Batch** (1-10):
       * Fast individual computations
       * Less parallelization benefit
       * More sequential processing
     - **Large Batch** (50-200):
       * Better GPU utilization
       * More parallel computations
       * Higher memory requirements
       * Better for batch optimization
     - **Memory Trade-off**: Larger batches require more memory but enable better parallelization
  3. **Interaction Effects**:
     - **Total Computations**: TIMESTEPS $\times$ N_BATCH determines total workload
     - **Memory Usage**: Scales with both parameters
     - **Parallelization**: Larger N_BATCH enables better GPU utilization
     - **Optimization Quality**: More samples generally lead to better solutions

4. **Practical Guidelines**:
   - Start with moderate values (TIMESTEPS=15, N_BATCH=50)
   - Increase TIMESTEPS for complex planning tasks
   - Increase N_BATCH for better GPU utilization
   - Monitor memory usage and computation time

- **Why do you think we chose to set the initial state of the environment to the downward position?**

  Answer: Setting the initial state to the downward position serves several important purposes:

  1. **Task Difficulty**:
     - **Challenging Starting Point**: Downward position is far from goal (upright)
     - **Realistic Scenario**: Represents common real-world situations
     - **Non-trivial Control**: Requires sophisticated control strategy
  2. **Learning and Evaluation**:
     - **Clear Success Metric**: Easy to measure when pendulum reaches upright
     - **Consistent Evaluation**: Same starting condition for all experiments
     - **Performance Benchmarking**: Allows fair comparison between methods
  3. **Control Strategy Testing**:
     - **Swing-up Phase**: Tests ability to generate energy
     - **Balance Phase**: Tests ability to maintain upright position
     - **Two-phase Control**: Requires different strategies for different phases
  4. **MPC Advantages**:
     - **Receding Horizon**: Can plan ahead for both swing-up and balance
     - **Adaptive Strategy**: Adjusts control based on current state
     - **Constraint Handling**: Can incorporate physical constraints
  5. **Alternative Starting Positions**:
     - **Upright Position**: Would only test balance, not swing-up
     - **Random Positions**: Would make evaluation inconsistent
     - **Downward Position**: Provides comprehensive test of control capabilities

- **As time progresses (later iterations), what happens to the actions and rewards? Why?**

  Answer: As the MPC controller learns and the pendulum approaches the goal, both actions and rewards exhibit characteristic patterns:

  1. **Action Evolution**:
     - **Early Phase** (Swing-up):

    - Large amplitude actions to build energy
    - Alternating positive/negative torques
    - High action magnitudes to overcome gravity
  - **Late Phase** (Balance):
    - Small amplitude corrections
    - Fine-tuning actions around upright position
    - Reduced action magnitudes
2. **Reward Progression**:
  - **Early Episodes**:
    - Large negative rewards (far from goal)
    - High energy costs
    - Penalty for being far from upright
  - **Later Episodes**:
    - Smaller negative rewards (closer to goal)
    - Reduced energy costs
    - Better balance around upright position
3. **Why This Happens**:
  - **Learning Process**: MPC learns optimal trajectory through experience
  - **Model Improvement**: Dynamics model becomes more accurate
  - **Policy Refinement**: Control strategy becomes more efficient
  - **Convergence**: Approaches optimal control policy
4. **Mathematical Explanation**:
  - **Reward Function**: $r = -(\theta^2 + 0.1\dot{\theta}^2 + 0.001u^2)$
  - **As** $\theta \to 0$: Angular penalty decreases
  - **As** $\dot{\theta} \to 0$: Velocity penalty decreases
  - **As** $u \to 0$: Action penalty decreases
5. **Practical Implications**:
  - **Performance Monitoring**: Track reward improvement over time
  - **Convergence Detection**: Identify when learning stabilizes
  - **Hyperparameter Tuning**: Adjust based on learning progress
  - **Success Criteria**: Define when task is considered solved