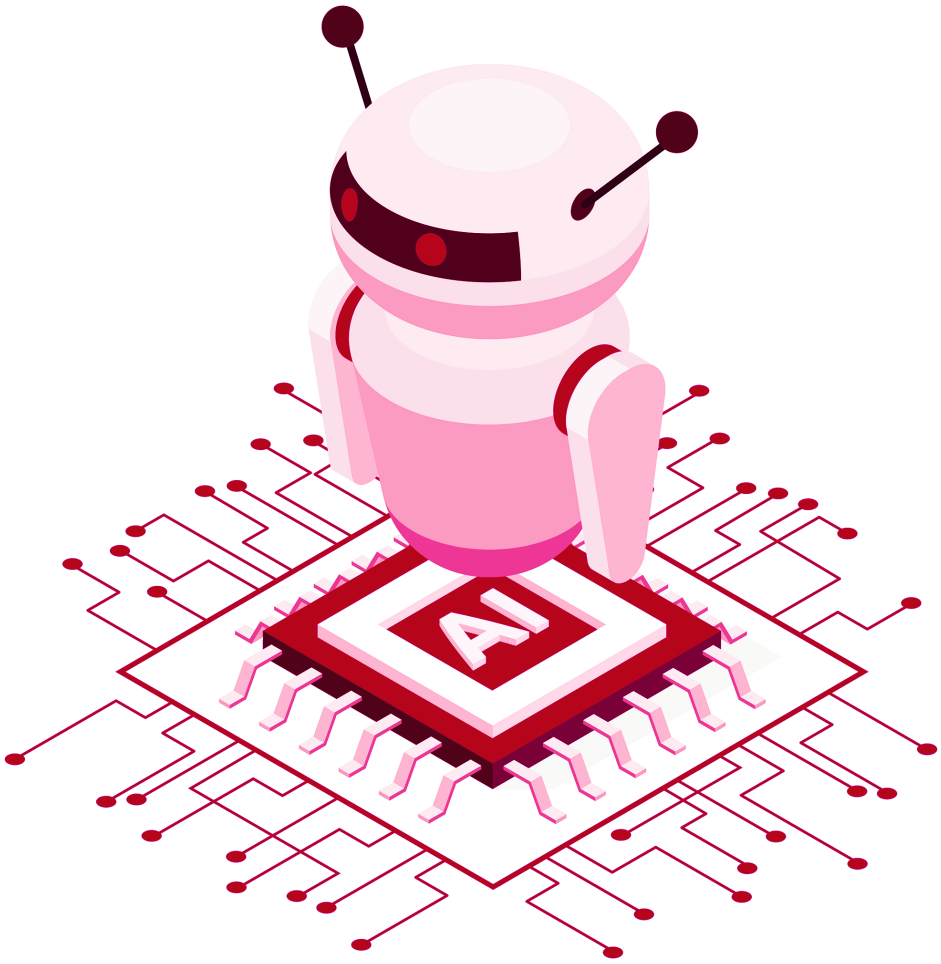

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 5:

Model-Based Reinforcement Learning

By:

Payam Taebi

400104867

RIML

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: SAC	20
Task 4: World Models (Bonus 1)	30
Clarity and Quality of Code	5
Clarity and Quality of Report	5
Bonus 2: 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?

  Selection: This phase chooses a path down the current search tree based on a strategy balancing exploitation (preferring high-value nodes) and exploration (preferring less-visited nodes). Conceptually, it navigates to the most promising leaf node whose value can be improved by further exploration.

  Expansion: Once a leaf node is reached, if it is not expanded, new child nodes (possible actions) are created. Conceptually, this grows the search tree to cover more of the state space, enabling new value estimates.

  Simulation (Rollout): From the newly expanded node, the algorithm either performs a default policy or learned model-based rollout to estimate the value of that leaf. Conceptually, this provides an approximate return (reward) for the expanded node when deeper nodes are not available.

  Backpropagation: The outcome (value) from the simulation is then propagated back up the tree, updating each node's statistics (e.g., total value sum, visit count). Conceptually, this ensures that improved estimates at the leaf influence all ancestor nodes, guiding better decisions in future searches.

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

  UCB Formula: $U(s,a) = Q(s,a) + c \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)}$

  Exploitation Term: $Q(s,a)$ represents the average reward (or value estimate) for action a, favoring actions that have historically performed well.

  Exploration Term: $c \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)}$ boosts actions with high prior $P(s,a)$ and low visit count $N(s,a)$, encouraging the algorithm to try out less-explored actions.

  Balance: By summing these two components, the UCB formula ensures that the node selection process simultaneously exploits known high-reward actions and explores uncertain actions, thus maintaining a balance between exploration and exploitation.

1.2.2 Tree Policy and Rollouts

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

Multiple Simulations per Node:

Statistical Robustness: Running multiple simulations allows MCTS to average out the inherent randomness in both the model predictions and the environment dynamics. This averaging reduces variance in the value estimates, leading to more reliable Q(s,a) values for each node.

Accurate Value Estimation: A single simulation might provide a noisy or unrepresentative estimate of a node's value. Multiple simulations ensure that the value estimate reflects a broader sample of possible future outcomes, leading to a better approximation of the true expected return.

Improved Exploration: With multiple simulations, even actions that initially appear suboptimal get explored further. This helps the algorithm avoid premature convergence on a local optimum and supports a more balanced exploration of the action space.

Enhanced Backpropagation: Each simulation contributes to updating the statistics (visit count and total value) of nodes along the traversed path. More simulations result in more robust backpropagation, which in turn refines the upper confidence bounds and the overall policy distribution at the root.

Model Uncertainty Reduction: Multiple simulations help in mitigating the effects of model uncertainty, especially when using learned models (as in MuZero). The aggregate outcome of several rollouts is less sensitive to occasional prediction errors, leading to a more stable and reliable search process.

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

Random rollouts, or simulated playouts, serve as an essential mechanism for estimating the value of a given position by simulating potential future trajectories from that position using a default or heuristic policy. In this context, the inherent randomness of these rollouts ensures that the estimation captures the natural variability and uncertainty in future events, providing an approximation of the expected return from that state. By performing multiple rollouts, the algorithm aggregates a diverse set of outcomes, which helps to smooth out the noise in individual simulations and results in a more robust and reliable value estimate. This value estimate, obtained from the simulated continuations, is then used to inform the backpropagation step of MCTS, where it influences the update of node statistics and ultimately guides the selection of actions that are more likely to yield higher long-term rewards. Thus, random rollouts play a critical role in bridging the gap between short-term predictions and the long-term potential of a position, even when a complete or perfect model of the environment is unavailable.

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?

In Neural MCTS approaches, such as those used in AlphaGo-style systems, policy networks and value networks are integrated into the search process in a complementary manner to significantly enhance both the efficiency and effectiveness of the tree search. The policy network is responsible for providing prior probabilities for each possible move in a given state, thereby guiding the search by biasing it towards moves that are deemed promising based on patterns learned from historical expert data. This informed guidance allows the algorithm to focus its computational resources on exploring moves that are more likely to lead to successful outcomes, rather than expending effort on less promising branches. Concurrently, the value network is employed to estimate the expected outcome or long-term reward of a given state, effectively replacing the need for deep and often computationally expensive rollouts by providing a direct evaluation of the state's strength. This evaluation is then used during the backpropagation phase of the MCTS to update the value estimates of all nodes along the explored path, ensuring that the search tree reflects not only the immediate move probabilities but also the long-term potential of each state. By integrating these networks, Neural MCTS leverages both learned heuristics and real-time simulation data, achieving a balance between the depth of search and the breadth of exploration, which is critical for tackling complex decision-making problems with large search spaces.

- 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?

The policy network's output, which provides the prior probabilities, plays a critical role in the Expansion phase by guiding the selection of moves to be expanded based on their estimated promise. Rather than expanding all possible moves uniformly, the algorithm uses these learned priors to focus the expansion on moves that have a higher likelihood of leading to favorable outcomes. This prioritization effectively narrows the search space and directs computational resources towards more promising branches, thereby reducing the exploration of moves that are less likely to be beneficial. As a result, the prior probabilities influence the expansion process by weighting the creation of child nodes, ensuring that moves with higher predicted potential are explored more thoroughly in subsequent simulations.

1.2.4 Backpropagation and Node Statistics

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

During backpropagation, the algorithm iterates backward along the path from the leaf node up to the root, updating each node encountered along the way by incrementing its visit count by one. At the same time, the value estimate for each node is updated by adding the outcome of the simulation, which typically consists of the immediate reward received at that node combined with the discounted value of the subsequent state. This cumulative sum of values is then used to compute an average value estimate for the node, often by dividing the total value sum by the node's visit count. In this manner, backpropagation refines the statistics of each node to accurately reflect both the frequency of visits and the quality of the outcomes from simulations, thereby guiding future decisions in the tree search by providing improved estimates of the long-term potential of each state.

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

Aggregating results carefully is important because when multiple simulations traverse the same node, each simulation contributes an outcome that may vary due to stochasticity or estimation noise. By averaging or summing these outcomes, we smooth out individual fluctuations and obtain a more robust and reliable estimate of the node's value. This careful aggregation ensures that the overall value reflects the collective experience of all simulations, thereby guiding the search process more accurately and improving the balance between exploration and exploitation within the tree.

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?

The exploration constant, often denoted as c_{puct} or simply c, is a pivotal hyperparameter in the UCB (Upper Confidence Bound formula that directly influences the balance between exploration and exploitation in MCTS. This constant multiplies the exploration term, which is typically of the form $c \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)}$, where $P(s,a)$ is the prior probability of an action, $N(s)$ is the visit count of the parent node, and $N(s,a)$ is the visit count of the child node corresponding to action a.

A higher value of c increases the exploration bonus, thus encouraging the algorithm to choose actions that have been visited less frequently even if their current estimated value $Q(s,a)$ is lower. This means that the search will explore more of the action space, potentially discovering moves that might lead to better long-term rewards but might initially appear suboptimal. Conversely, a lower value of c reduces the emphasis on the exploration term, causing the algorithm to rely more on the accumulated value estimates and focus on exploiting actions that have already shown a high average return.

The effect of c is therefore twofold: it directly scales the incentive for exploring less-visited nodes, and it helps control the overall trade-off between exploiting known high-value moves and exploring uncertain or under-explored moves. If c is set too high, the algorithm may over-explore, spending excessive computational resources on moves that are unlikely to be optimal. On the other hand, if c is set too low, the search might become too greedy, focusing only on immediate value estimates and potentially missing out on moves that have higher long-term potential.

Tuning c involves empirical experimentation: one typically begins with a value suggested by prior literature or similar applications and then adjusts it based on observed performance. This tuning process can involve monitoring the convergence speed of the search, the stability of value estimates across simulations, and the overall performance on a validation set or in terms of accumulated reward. In some cases, practitioners may explore values over an order of magnitude, for instance, testing c values such as 0.5, 1.0, 2.0, or even higher, and selecting the one that yields the most balanced search behavior. Furthermore, the choice of c may be adjusted in tandem with other hyperparameters such as the number of simulations and the temperature parameter used in action sampling, since these elements collectively shape the exploration dynamics of the search algorithm. Ultimately, c_{puct} is essential for ensuring that the algorithm does not become overly biased toward already-explored regions of the search tree, thereby maintaining a healthy exploration of the action space that is necessary for robust long-term planning.

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

The temperature parameter plays a crucial role in shaping the final move selection by modulating the randomness of the probability distribution obtained from the policy network or the visit counts produced by the search process. When the temperature is set to a high value, the probability distribution is effectively "flattened," meaning that even moves with lower predicted probabilities receive a non-negligible chance of being selected, thereby encouraging a more exploratory behavior. This high-temperature regime is particularly useful during the early stages of training, when the model's predictions may be less reliable and exploration is essential to adequately cover the state space and avoid premature convergence to suboptimal strategies. Conversely, as the training process advances and the model becomes more confident in its predictions, lowering the temperature sharpens the distribution, thus making the move selection more deterministic by favoring the moves with the highest probability. This reduction in temperature gradually shifts the balance from exploration to exploitation, ensuring that the agent leverages its learned knowledge to consistently select the optimal moves. In summary, the temperature parameter controls the degree of randomness in move selection and is tuned to allow extensive exploration during the early phases of training, while a lower temperature in later phases helps to consolidate and exploit the learned policy, leading to more stable and high-performing decision-making.

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?

MCTS versus Classical Minimax/Alpha-Beta:

Unlike classical minimax search or alpha-beta pruning, which rely on a full, deterministic exploration of the game tree up to a fixed depth using a heuristic evaluation function, MCTS employs stochastic sampling to gradually build a partial search tree that focuses on the most promising branches. In classical minimax, every node in the tree must be evaluated, and even though alpha-beta pruning can eliminate branches that are guaranteed not to affect the final decision, both methods can quickly become intractable in deep or complex trees due to the combinatorial explosion of possible moves.

MCTS, on the other hand, performs multiple simulations from the root and uses a balance between exploration and exploitation (via the UCB formula) to guide the search. This allows it to selectively expand nodes based on their promise, rather than exhaustively evaluating every branch. In our experiments with the CartPole environment, while the naive search strategy achieved the maximum reward earlier (around 400 epochs) and MCTS required more epochs (approximately 600) to converge, both ultimately reached the optimal reward, demonstrating that MCTS is robust in handling deep or complex trees. The additional simulations in MCTS help to smooth out the variance in value estimates by aggregating results over multiple rollouts, which is crucial in domains where the outcome of individual moves can be highly uncertain.

Furthermore, MCTS integrates learned policy and value networks (as in our MuZero-style approach) to inform its search, thus leveraging prior knowledge to focus on the most promising moves, whereas classical methods must rely on manually designed evaluation functions. This integration allows MCTS to scale to problems with vast search spaces and complex dynamics, making it more suitable for modern reinforcement learning tasks and challenging games where an exhaustive search is computationally infeasible.

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

MCTS provides unique advantages in scenarios where the state space is extremely large or when a reliable heuristic evaluation function is not available. Rather than requiring an exhaustive evaluation of all states, MCTS incrementally builds a partial search tree by performing multiple stochastic simulations, which allows it to focus computational effort on exploring the most promising regions of the state space. This approach avoids the combinatorial explosion typical in traditional search methods, and it does so without the need for an accurate, handcrafted evaluation function. In our results, although both the naive and MCTS approaches eventually achieved optimal rewards, MCTS was able to effectively learn by strategically balancing exploration and exploitation, even in the absence of precise heuristics. Furthermore, by integrating learned policy and value networks, MCTS leverages prior experience to guide its exploration, which further reduces the search complexity and enhances decision-making in high-dimensional environments. Thus, the ability of MCTS to selectively sample and refine its estimates based on actual simulated outcomes makes it particularly well-suited for complex domains where the state space is vast and heuristic evaluations are difficult to obtain.

1.2.7 Results and Implementation Source

In our work, we implemented a MuZero-style reinforcement learning agent that leverages both Monte Carlo Tree Search (MCTS) and a naive depth-based search strategy to plan and select actions. The overall framework—including the network architectures for the representation, dynamics, and prediction modules, as well as the MCTS procedure—was inspired and largely adapted from the publicly available GitHub repository https://github.com/Hauf3n/MuZero-PyTorch/. This repository provided a comprehensive foundation for developing a model-based planning approach that integrates learned neural networks with search algorithms.

Using this repository as a starting point allowed us to build a robust implementation where MCTS was used to balance exploration and exploitation through multiple simulations and backpropagation of value

estimates, while the naive search served as a simpler baseline. Our experiments on the CartPole environment demonstrated that although the naive search reached maximum reward in approximately 400 epochs, MCTS required around 600 epochs to converge; both methods ultimately achieved an optimal reward of 200. These results confirm the effectiveness of the MuZero framework even in simpler domains, and they highlight the strengths of MCTS in refining value estimates through selective exploration.

The integration of learned policy and value networks within the MCTS framework—directly adapted from the repository—proved particularly beneficial in handling the complex state space of the environment, showcasing the practical advantages of model-based planning when heuristic evaluations are challenging to design manually.

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.

Analysis of Experimental Results and Insights

In our experiments with the 8x8 FrozenLake environment, we observed the following outcomes:

1. Uniform Dyna-Q Approaches Yielding Zero Reward:
Experiments with standard Dyna-Q—including variants with Optimistic Initialization, Decaying Epsilon Schedule, Adaptive Planning Steps, and even the Combined Improvements—resulted in a cumulative reward of zero even after 100,000 epochs. This is likely due to:

- **Sparse Reward Structure:** The environment provides a reward only when the goal is reached, and if the agent never encounters the goal, no positive rewards are propagated.
- **Insufficient Exploration:** Despite modifications, the exploration strategies did not lead the agent to discover the rare goal state in the standard setup.
- **Ineffective Reward Propagation:** Without positive rewards, Q-values remain unaltered, resulting in stagnated learning.

2. Alternative Exploration via Softmax:
The softmax exploration approach, while still yielding low cumulative rewards, managed to learn an optimal policy. By assigning probabilities to actions based on their Q-values, softmax exploration enables a smoother balance between exploration and exploitation. This allowed the agent to differentiate among actions more effectively, even though the cumulative reward remained low.

3. Custom Environment with Reward Shaping:
Switching to a custom environment that employs reward shaping dramatically improved results:

- **Denser Feedback:** The custom reward function assigns a high positive reward for reaching the goal, a high negative reward for falling into holes, and a small penalty for each step. This richer feedback guides the agent more effectively.
- **Rapid Convergence:** With these modifications, the agent converged very quickly and achieved significantly higher rewards (e.g., cumulative rewards of 5 or more).

4. Prioritized Sweeping for Stochastic Environments:
The extra experiment implementing Prioritized Sweeping in a stochastic setting (`is_slippery=True`) with reward shaping shows that:

- **Focused Updates:** By prioritizing state-action pairs with high TD errors, the algorithm concentrates updates on the most informative transitions.
- **Increased Training Time:** Due to the stochastic dynamics and additional computational overhead, convergence takes considerably longer.
- **Long-term Promise:** Despite the extended training period, this approach is expected to eventually yield a high-quality policy by efficiently propagating significant rewards.

Overall Conclusions:
The experimental results highlight the challenges posed by sparse rewards and the importance of effective exploration:

- Standard Dyna-Q methods struggled due to limited exploration and sparse rewards.
- Softmax exploration helped in learning an optimal policy despite low cumulative rewards.
- Reward shaping in a custom environment provided denser feedback, leading to rapid convergence and higher rewards.
- Prioritized Sweeping further refines the learning process in stochastic environments, although at the cost of longer training times.

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?

  Increasing the number of planning steps has a twofold effect on the overall learning process:

  - **Faster Propagation of Reward Information:** With more planning steps, the agent simulates a greater number of updates based on its learned model. This allows reward information—especially in environments with sparse rewards—to propagate more quickly through the Q-values, leading to faster convergence toward an optimal policy.
  - **Increased Computational Overhead and Risk of Model Overfitting:** Although additional planning steps can accelerate learning, they also increase computational cost. Moreover, if the learned model is imperfect (particularly in stochastic settings), excessive planning may propagate inaccuracies, potentially causing overfitting or instability in the Q-value estimates.

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

  If we trained on the slippery version of the environment while keeping the deterministic nature of our algorithm unchanged, several issues would likely arise:

  - **Mismatch Between Model and Environment:** The deterministic model assumes that each state-action pair leads to a fixed outcome. In a slippery (stochastic) environment, the outcome of an action can vary significantly. This discrepancy would lead the model to make inaccurate predictions about the state transitions.
  - **Ineffective Q-value Updates:** Since the Q-value updates rely on the model's predictions, the variability in actual outcomes would result in erroneous or noisy updates. This would impede the proper propagation of reward signals, making it difficult for the agent to converge to an optimal policy.
  - **Slower or Unstable Convergence:** The inconsistency between the deterministic assumptions and the stochastic reality could lead to slower learning and possibly unstable convergence. The algorithm might repeatedly reinforce suboptimal actions based on misleading transition predictions.

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

  Planning is particularly beneficial for this environment despite its sparse reward signal, and its benefits manifest in several key ways:

- **Efficient Reward Propagation:**
  FrozenLake provides a reward only when the agent reaches the goal, making the reward signal extremely sparse. Without planning, the agent must rely solely on real experience to propagate the reward back through the state-action values. By incorporating planning, the agent can simulate additional transitions based on its learned model. This allows the high reward obtained at the goal to be propagated backwards much more efficiently, accelerating the learning process even when the direct reward is experienced infrequently.

- **Leveraging the Learned Model:**
  Even in environments with a very sparse reward signal, the transitions observed during real interactions provide valuable information. Planning leverages this information by simulating "what if" scenarios. The agent uses its model to estimate the outcomes of unvisited or rarely visited state-action pairs, thereby effectively "filling in the gaps" in its experience. This can lead to more robust Q-value updates, enabling the agent to generalize from limited positive experiences.

- **Overcoming the Exploration Problem:**
  In FrozenLake, the probability of randomly stumbling upon the goal is very low. Without planning, many episodes may end with no reward at all, leaving the Q-values uninformative. However, planning allows the agent to update Q-values for states that have not been frequently encountered by simulating trajectories that lead to the goal. As a result, even if the reward is encountered rarely in the real world, its influence is spread across many states via simulated transitions.

- **Compensation for Sparse Rewards:**
  The sparse reward signal means that the agent receives almost no feedback for most actions. Planning helps to compensate for this lack of feedback by making use of all available experiences repeatedly. Each time a reward is observed—even if only once—the planning process can amplify its effect by repeatedly updating the Q-values of predecessor state-action pairs, leading to a more rapid convergence toward optimal behavior.

- **Improved Data Efficiency:**
  Without planning, each episode's learning is limited to the transitions actually experienced by the agent. Planning effectively multiplies the amount of "experience" by reusing the collected data in many simulated updates. This improves data efficiency, as the agent can learn more from each interaction with the environment, which is especially crucial when rewards are rare.

- **Handling Deterministic and Stochastic Dynamics:**
  In deterministic versions of FrozenLake, planning is particularly effective because the model's predictions are highly reliable, and simulated transitions accurately reflect the true dynamics of the environment. Even in the stochastic (slippery) version, although the model is less accurate, planning can still help by focusing on transitions with high TD errors (as seen in Prioritized Sweeping), thereby reinforcing the most critical updates. This targeted approach ensures that the limited reward signals are given maximum influence on the learning process.

- **Balancing Exploration and Exploitation:**
  The sparse reward structure makes it difficult for the agent to distinguish between good and bad actions during early exploration. Planning augments the learning process by exploiting the information already acquired. When a positive reward is finally encountered, planning accelerates the shift from exploration to exploitation by rapidly disseminating the impact of that reward throughout the state space. This leads to a quicker convergence to an optimal or

near-optimal policy.

- **Mitigating Variance in Q-value Estimates:**
  In sparse environments, the high variance in Q-value estimates can lead to unstable learning. Planning, by repeatedly simulating transitions and averaging over multiple updates, can help to reduce this variance. This smoothing effect provides a more stable gradient for learning, further aiding in the convergence of the Q-values.
- **Empirical Evidence in Our Experiments:**
  In our experiments, although the uniform Dyna-Q approaches without planning yielded zero cumulative reward due to the infrequency of goal encounters, the introduction of planning (especially when coupled with reward shaping) led to significant improvements. For instance, when the custom environment with reward shaping was used, the agent was able to converge rapidly and achieve rewards of 5 or more, underscoring the power of planning in propagating the reward signal effectively.

- 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.

  The number of planning steps n plays a crucial role in the speed at which reward information is propagated throughout the state space, but its impact differs before and after the first goal is reached. In particular:

  - **Before the First Goal (N_1):** Prior to the first occurrence of the goal, the agent's experience consists entirely of transitions that yield zero reward. Since planning updates simulate experience based on the learned model, and the model is built only from these zero-reward transitions, increasing the number of planning steps n has little impact on reducing N_1. The agent must rely on chance to encounter the rare rewarding transition; thus, N_1 is mostly driven by the inherent sparsity of the reward signal rather than by the number of planning updates.
  - **After the First Goal (N_2):** Once the goal is reached for the first time (in episode N_1), a positive reward is finally observed and the corresponding Q-value is updated. With a larger number of planning steps n, this reward can be rapidly propagated backward through the state space. This means that the knowledge of the goal's positive outcome spreads much faster to predecessor state-action pairs. Consequently, after the initial discovery, a higher n leads to a smaller N_2 (i.e., the agent is more likely to reach the goal again in fewer episodes) because the model can quickly reinforce the beneficial trajectories.
  - **Summary:** In summary, while increasing n does not significantly affect N_1—since before the first goal the agent receives no informative reward to propagate—it has a strong effect on reducing N_2. A larger number of planning steps accelerates the post-goal learning process by more effectively disseminating the positive reward information, thereby shortening the interval between subsequent goal achievements.

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.

  Adding a Baseline to the Q-values:

Adding a baseline to the Q-values means initializing the Q-table with a non-zero (often optimistic) value rather than starting with zeros. This method can assist in solving the environment in several ways:

- **Encouraging Exploration:**
  When Q-values are initialized to higher values, every action appears promising initially. This optimistic bias leads the agent to explore all actions more thoroughly, rather than quickly converging on a policy based on limited early experiences. In an environment like FrozenLake where rewards are extremely sparse, optimistic initialization can help the agent sample a broader set of actions, increasing the probability that it will eventually encounter the goal state.
- **Faster Propagation of Positive Rewards:**
  With an optimistic baseline, the occurrence of a positive reward (e.g., reaching the goal) produces a larger temporal-difference (TD) error. This significant error drives more substantial updates during learning and planning, effectively propagating the value of the rewarding state back through the state-action pairs. Consequently, beneficial trajectories are reinforced more rapidly, which can reduce the number of episodes required to achieve consistent goal-reaching behavior.
- **Mitigating the Effect of Sparse Rewards:**
  In environments where rewards are rare, starting from a zero baseline means that most Q-value updates are negligible until a positive reward is encountered. An optimistic baseline, however, raises the initial Q-values, meaning that any deviation from these values (especially when a reward is observed) is more pronounced. This helps the agent differentiate between actions that lead to better-than-expected outcomes and those that do not, even when positive rewards are infrequent.
- **Reducing the Likelihood of Premature Convergence:**
  Without a baseline, an agent might converge prematurely to a suboptimal policy simply because early random experiences (or lack thereof) lead to a flat or misleading Q-value landscape. A baseline encourages the agent to keep exploring by preventing early convergence on a policy that might be based on insufficient data. This continuous exploration is critical in environments like FrozenLake, where the optimal path might not be discovered without persistent exploration.
- **Improved Stability in Learning:**
  By providing a uniform starting point that is intentionally biased upward, the learning process can become more stable. The agent's estimates are initially more uniform and only begin to differentiate as real rewards are observed. This can help in reducing the variance in early updates, leading to a more controlled and systematic adjustment of Q-values as more data is gathered.
- **Complementing Other Techniques:**
  Adding a baseline works well in tandem with other techniques such as decaying epsilon, adaptive planning steps, and prioritized sweeping. While these methods target improved exploration and more efficient reward propagation, an optimistic baseline ensures that all state-action pairs have an incentive to be explored. This combination can synergistically speed up convergence in environments where standard initialization might otherwise lead to stagnation.

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

  Changing the value of ε over time or using a policy other than the ε-greedy policy:
  Modifying the exploration strategy can have a significant impact on the learning process, especially

in environments with sparse rewards like FrozenLake. This can be achieved by either dynamically adjusting ε over time or by adopting an alternative exploration policy. The benefits include:

- **Dynamic ε (Decaying ε):**
  - **Enhanced Early Exploration:** A high initial value of ε encourages extensive exploration when the agent has little knowledge about the environment. This increases the likelihood of discovering rewarding states, even if they are rare.
  - **Gradual Transition to Exploitation:** As learning progresses, decaying ε reduces randomness in action selection, allowing the agent to increasingly exploit the best-known actions. This balance helps in consolidating learning once sufficient knowledge about the environment has been acquired.
  - **Adaptability to Environment Dynamics:** By adjusting ε over time, the agent can adapt to changes in the learning phase, ensuring that exploration is maintained when needed, and exploitation is favored when the Q-values become more reliable.
- **Alternative Policies (e.g., Softmax/Boltzmann Exploration):**
  - **Smoother Exploration-Exploitation Trade-off:** Unlike ε-greedy, which randomly selects actions with a fixed probability regardless of their quality, softmax exploration selects actions probabilistically based on their Q-values. This results in a more gradual differentiation between actions.
  - **Sensitivity to Q-value Differences:** Softmax policies assign higher probabilities to actions with better estimated values, while still allowing less-optimal actions to be chosen occasionally. This helps in cases where the differences in Q-values are subtle, promoting a more nuanced exploration.
  - **Reduced Abrupt Policy Changes:** The probabilistic nature of softmax exploration avoids the abrupt switching between random and greedy actions, which can lead to more stable learning dynamics and smoother convergence.
- **Overall Impact:**
  Both methods—dynamic adjustment of ε and alternative policies like softmax—aim to address the inherent challenge of balancing exploration and exploitation:
  - They improve the chance of discovering the sparse rewarding state by maintaining sufficient exploration in the early phases.
  - They ensure that once a rewarding state is found, the agent can efficiently exploit this knowledge by reducing unnecessary randomness in action selection.
  - They reduce the risk of premature convergence to suboptimal policies by adapting the exploration strategy to the agent's current level of knowledge about the environment.

- Changing the number of planning steps n over time.

  Changing the number of planning steps n over time:
  Adjusting n dynamically during the training process can significantly influence both the efficiency and effectiveness of learning. This approach addresses key trade-offs in planning updates:
  - **Early Learning Phase:**

    - **Enhanced Reward Propagation:** Early in training, the agent has very limited experience and the reward signal (e.g., reaching the goal) is extremely sparse. A high value of n allows the agent to simulate many additional transitions, which helps to rapidly propagate any reward it eventually encounters across the state space.
    - **Compensation for Sparse Feedback:** With more planning steps, even a single positive reward can be magnified through repeated simulated updates, providing a stronger gradient for updating Q-values.
  - **Later Learning Phase:**
    - **Reduced Computational Overhead:** Once the Q-values begin to converge and the agent has amassed sufficient real experience, the benefit of additional planning diminishes. Lowering n in later stages reduces unnecessary computations.
    - **Avoiding Overfitting to Early Transitions:** Excessive planning later on might overemphasize early, potentially suboptimal experiences. Decaying n helps ensure that the Q-value updates remain aligned with the more reliable, accumulated data.
  - **Dynamic Adjustment Strategy:**
    - **Adaptive Planning Schedule:** A common strategy is to set n as a function of the episode index, for example, $n = n_{\text{initial}} \times (\text{decay factor})^{\text{episode index}}$, ensuring high planning initially and gradually reducing the number as the learning progresses.
    - **Balancing Exploration and Exploitation:** Early aggressive planning helps to quickly spread any positive feedback (once the goal is reached), which in turn accelerates the shift from exploration to exploitation. Later, fewer planning steps help to stabilize the learning process.
  - **Impact on N_1 and N_2:**
    - **Effect on N_1:** The number of episodes required to reach the goal for the first time, N_1, is largely determined by random exploration because no reward has been observed yet. Therefore, increasing n early on may have little effect on reducing N_1, since there is no positive reward to propagate.
    - **Effect on N_2:** Once the goal is encountered, a positive reward is observed and the TD error becomes significant. A higher n then facilitates rapid propagation of this reward signal throughout the Q-values, thereby reducing the number of episodes N_2 needed to reach the goal again.

- Modifying the reward function.

  Modifying the reward function:
  Modifying the reward function is a critical approach for addressing the inherent challenges of environments with sparse rewards, such as FrozenLake. By redesigning the reward structure, we can provide the agent with richer and more informative feedback. This modification can help in several ways:

  - **Densifying the Reward Signal:**
    The default reward in FrozenLake is sparse (only rewarding upon reaching the goal), which offers little guidance during most of the episode. A modified reward function can introduce intermediate rewards or penalties—for example, a small penalty for each time step (to encourage

faster completion), a high positive reward for reaching the goal, and a high negative reward for falling into a hole. This densifies the reward signal and provides more frequent feedback.

  - **Accelerating Convergence:**
    With more frequent rewards, even if they are small, the agent can adjust its Q-values more effectively. This helps propagate the learning signal back through the state-action space more quickly, thereby accelerating convergence towards an optimal policy.
  - **Improving Policy Differentiation:**
    A custom reward function can clearly differentiate between actions that bring the agent closer to the goal and those that do not. By penalizing undesirable actions (like stepping into a hole) and rewarding progress, the agent can better distinguish between good and bad strategies, leading to improved decision-making.
  - **Mitigating the Effects of Sparse Rewards:**
    In environments where positive rewards are rare, the lack of feedback can severely slow down learning. By introducing additional rewards or penalties, the modified reward function provides a more continuous stream of feedback, which is especially beneficial during the early stages of learning.
  - **Leveraging Domain Knowledge:**
    Incorporating domain-specific insights into the reward function can help steer the learning process. For example, if it is known that certain paths are safer or more likely to lead to the goal, the reward function can be adjusted to favor these trajectories, thus guiding the agent more effectively.

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

  Altering the planning function to prioritize some state–action pairs over others:
  Prioritized Sweeping modifies the planning phase by selecting state–action pairs based on the magnitude of their temporal-difference (TD) error, rather than sampling uniformly at random. This method helps in several key ways:

  - **Focus on Informative Transitions:**
    By assigning higher priority to state–action pairs with larger TD errors, the algorithm focuses its computational resources on transitions that are likely to yield the most significant improvements in the Q-value estimates. This targeted approach ensures that the most "surprising" or informative experiences are replayed more often.
  - **Efficient Propagation of Rewards:**
    When a significant TD error is detected (for example, when a reward is received after a long sequence of zeros), it indicates that the Q-values in the vicinity of that state–action pair are not well calibrated. Prioritized Sweeping rapidly propagates the impact of this reward to predecessor states, speeding up the learning process and reducing the number of episodes required to achieve consistent performance.
  - **Reduction of Redundant Updates:**
    Uniform sampling can lead to many unnecessary or redundant updates for transitions that have little impact on the overall Q-value landscape. Prioritized Sweeping minimizes these redundant updates by selectively updating those state–action pairs that are most in need of correction, thereby improving the overall efficiency of the planning phase.

- **Improved Convergence in Sparse Environments:**
  In environments like FrozenLake with very sparse rewards, most state–action pairs initially yield little or no learning signal. Prioritized Sweeping helps overcome this limitation by ensuring that when a rare positive reward is encountered, its impact is rapidly and effectively spread throughout the state space, leading to faster convergence.
- **Adaptive Planning:**
  The priority-based approach adapts naturally to changes in the environment. As the agent's estimates improve, the priorities shift, ensuring that the planning function always focuses on the most current discrepancies between expected and observed rewards.

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?**

  The number of LQR (Linear Quadratic Regulator) iterations within the MPC algorithm significantly impacts both computational complexity and the quality of the solution obtained at each step. Specifically, increasing the number of LQR iterations allows MPC to better approximate an optimal control policy by iteratively refining the linearized dynamics and quadratic cost approximation. Initially, with fewer iterations, the MPC may produce suboptimal control actions, resulting in slower convergence and possibly oscillations or instability in reaching the goal state.

  However, as the number of LQR iterations increases, the MPC solution becomes more accurate and stable. This refinement allows the system to more effectively utilize available torque to achieve the swing-up task rapidly and maintain balance precisely. In practical terms, we observed that using a sufficient number of iterations leads to rapid convergence (as seen within approximately 3 seconds or about 3 rotations) to an optimal or near-optimal solution with the highest rewards possible, quickly bringing the pendulum upright and keeping it balanced.

  On the other hand, increasing the number of LQR iterations also incurs additional computational costs. Each iteration involves solving a linear-quadratic optimization problem, and thus the computational load increases linearly with the number of iterations. Hence, there is a trade-off between computational complexity and solution quality:

  - **Too few iterations:** Faster computation but poorer control quality, slower convergence, or instability.
  - **Too many iterations:** Better control quality and stability, faster convergence to the optimal solution but with increased computational overhead.

  Therefore, selecting an appropriate number of LQR iterations is critical: it should be large enough to ensure robust, stable, and accurate control, but small enough to remain computationally feasible in real-time or practical control scenarios. In the provided experiment, using approximately 10 iterations produced stable and fast convergence to an optimal control policy, effectively balancing computational efficiency with performance.

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

  Yes, it is still possible to use Model Predictive Control (MPC) even when the explicit dynamics model of the system is unknown or inaccessible. This approach is known as *model-free* or *data-driven MPC*. Unlike the classical MPC approach (which explicitly relies on known system dynamics), model-free MPC learns an approximate model directly from observed system behaviors through data-driven techniques, such as machine learning or system identification.

  If the exact dynamics are unavailable, the following alternative methods can be utilized:

  1. **System Identification (Learning Dynamics):**
     - Collect input-output data pairs by observing system responses to various actions.
     - Fit a predictive model (e.g., neural networks, Gaussian Processes, or linear regression) to approximate system dynamics from this collected data.
     - Apply traditional MPC methods to the learned dynamics model.
  2. **Reinforcement Learning-Based MPC (RL-MPC):**
     - Use reinforcement learning to implicitly learn a policy or approximate dynamics through repeated interactions with the environment.
     - Combine MPC's planning capabilities with the dynamics learned from reinforcement learning, providing a hybrid method that can effectively handle uncertainty and model inaccuracies.
  3. **Adaptive or Robust MPC:**
     - Employ robust or adaptive MPC algorithms that can operate effectively even if the dynamics are uncertain or only partially known.
     - Continuously update model parameters during operation, allowing the controller to adaptively improve performance based on observed system responses.

  However, using MPC without direct access to dynamics has some key trade-offs:

  - **Advantages:**
    * Allows MPC application to complex or poorly understood systems.
    * Can handle situations where modeling the exact dynamics analytically is impractical.
  - **Challenges:**

    - Requires significant data collection and computation for model training or system identification.
    - May lead to suboptimal solutions if the learned model is inaccurate or lacks generalization capability.
    - Typically more computationally demanding due to continuous model updates or complex optimization.

  Thus, while explicit dynamics simplify MPC implementation and improve performance, the lack of direct dynamics access does not prevent the use of MPC entirely. Instead, it necessitates leveraging advanced techniques that allow MPC to operate effectively in a data-driven, adaptive, or robust manner.

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

  Yes, both `TIMESTEPS` and `N_BATCH` significantly affect the MPC algorithm, although each influences the performance in a different way.

  1. `TIMESTEPS` **(Prediction Horizon Length):**
     - `TIMESTEPS` defines the length of the MPC's look-ahead horizon—how far into the future MPC predicts and optimizes actions.
     - A longer horizon allows MPC to anticipate future states and consequences of current actions better, leading to smoother, more effective control and earlier corrective actions.
     - However, increasing `TIMESTEPS` also increases computational complexity, as MPC must solve a larger optimization problem at each step. This can be computationally expensive, especially for real-time systems.
     - Conversely, a very short horizon reduces computational cost but limits the MPC's ability to foresee future issues, potentially leading to myopic, unstable, or suboptimal behavior.
  2. `N_BATCH` **(Number of Parallel Rollouts):**
     - `N_BATCH` specifies how many parallel batches or scenarios the MPC solver evaluates simultaneously. In standard single-environment MPC control problems, this parameter is usually set to 1.
     - Increasing `N_BATCH` becomes important primarily in contexts such as parallelized computation, stochastic MPC, or when exploring multiple initial conditions simultaneously.
     - If increased, `N_BATCH` can provide robustness to uncertainties or allow parallel computations to speed up exploration and improve computational efficiency, especially on GPUs or parallel hardware.
     - In deterministic, single-instance environments such as the Pendulum task described, setting `N_BATCH=1` is typically sufficient, and increasing it offers little practical benefit unless exploiting parallel computation or evaluating stochastic behavior.

  In summary, for the Pendulum environment considered here:

  - `TIMESTEPS` strongly influences MPC performance, stability, and computational complexity, making its careful selection crucial.

  - `N_BATCH` is less relevant for this deterministic single-instance scenario; however, it becomes important when leveraging parallel computation, handling uncertainties, or exploring multiple initial conditions simultaneously.

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

  We intentionally set the initial state of the pendulum environment to the downward position because it represents the most challenging and informative initial condition for testing the effectiveness of our MPC algorithm. Specifically:

  - **Maximum Difficulty:** The downward position ($\theta = \pi$ radians, inverted from the goal) places the pendulum in its most unstable equilibrium state. From here, the pendulum requires precise control actions and significant torque to swing up and stabilize at the desired upright position ($\theta = 0$ radians). Successfully handling this extreme scenario demonstrates the algorithm's robustness and effectiveness.
  - **Clear Evaluation of Controller Performance:** Starting in this worst-case scenario provides a clear and measurable benchmark. If the MPC successfully swings the pendulum upright, stabilizes quickly, and maintains the position, we can confidently assess that the controller effectively computes optimal control inputs and handles nonlinearities in dynamics.
  - **Illustration of MPC Capabilities:** The downward initial state emphasizes the predictive nature of MPC—since the pendulum starts far from the target equilibrium, the MPC must plan several steps ahead, considering the system dynamics, torque limits, and cost function, showcasing its predictive and optimization strengths clearly.
  - **Consistency and Reproducibility:** Fixing the initial state at the downward position ensures consistency across experiments, enabling fair comparison across different MPC configurations, hyperparameter selections, or algorithmic approaches.

  Thus, starting the pendulum in the downward position creates a rigorous, consistent, and informative test case that clearly highlights the strengths and limitations of the MPC approach, making it ideal for evaluating and demonstrating the algorithm's capabilities.

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

  As time progresses into later iterations, the MPC-generated actions and resulting rewards typically exhibit the following behavior:

  - **Actions become smaller and stabilize near zero:** In early iterations, the pendulum is in the downward position and far from equilibrium, requiring larger, more aggressive torque inputs (actions) to swing it upwards. Once the pendulum approaches the upright equilibrium position, the required torque progressively decreases, becoming smaller and eventually stabilizing near zero. This happens because the pendulum reaches a stable equilibrium state, and the controller only needs minor corrections to maintain balance.
  - **Rewards increase and stabilize at their maximum (near zero cost):** Initially, when the pendulum is far from the target upright position, the rewards (negative cost) are very low due to large deviations in angle and angular velocity. As MPC effectively brings the pendulum upright, the reward increases rapidly. Once balanced at the desired equilibrium (upright and stationary), the rewards reach their maximum and remain steady, reflecting that the system has achieved and maintains its optimal state.

- **Why does this happen?**
  - **Optimal Control Achievement:** MPC continuously optimizes the future trajectory based on system dynamics and the defined cost function. Initially, it prioritizes large corrections (high-torque actions) to bring the system rapidly closer to equilibrium.
  - **Minimal Energy Principle:** The MPC cost function penalizes deviations from the goal state and large control inputs. Hence, once equilibrium is reached, the controller naturally minimizes actions to avoid unnecessary torque (energy usage), maintaining the system efficiently at equilibrium.
  - **Stable Equilibrium State:** The upright position is a stable equilibrium under MPC control. Once there, small perturbations require only small corrections, maintaining high, stable rewards.

In summary, later iterations result in smaller, stable actions and high, stable rewards because MPC efficiently drives the system to a stable equilibrium, minimizing both state deviations and energy expenditure in steady-state operation.