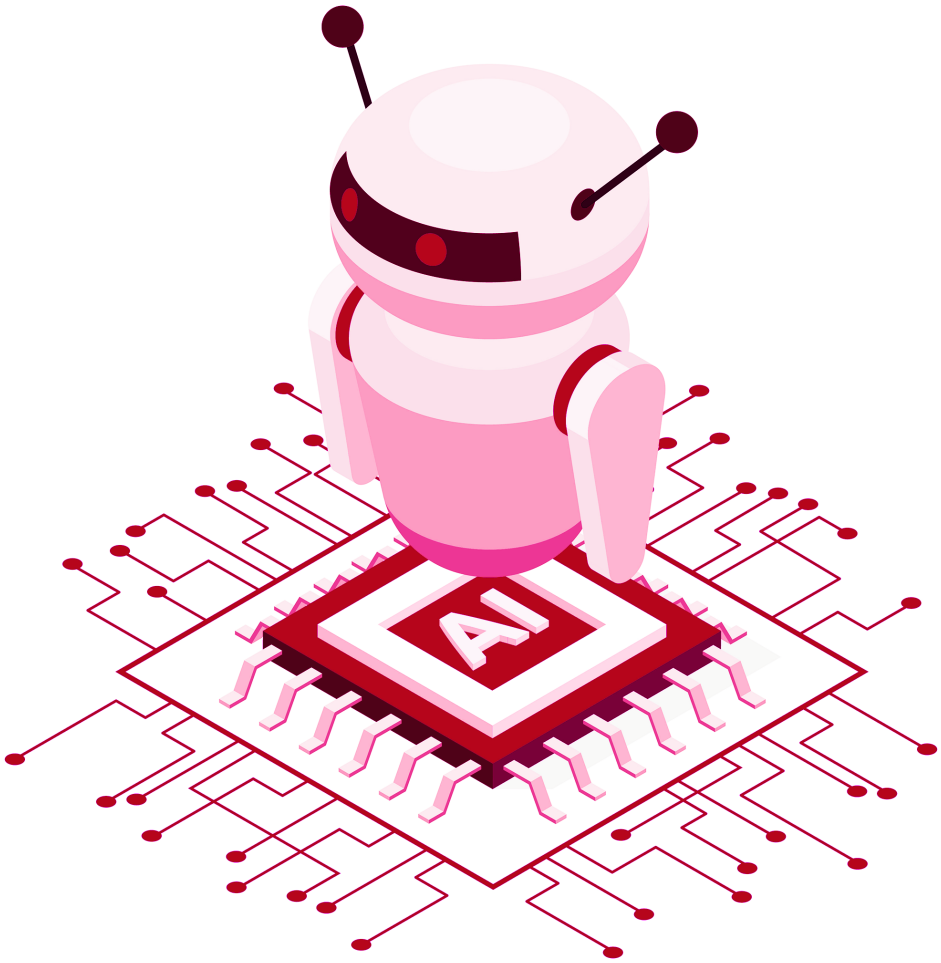

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 5:

Model-Based Reinforcement Learning

By:

Asemaneh Nafe

400105285

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: 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 The algorithm starts at the root node and selects child nodes recursively based on a strategy (usually UCB1). The goal here is to find the most promising node that balances exploration and exploitation.

  Expansion Once an under-explored node is found, the algorithm expands it by adding one or more child nodes (representing possible next states). This helps explore new areas of the search space.

  Simulation (Rollout) A simulation (also called a "playout") is performed from the newly expanded node, where the algorithm plays random moves until the game ends. This gives an estimate of how good that node is.

  Backpropagation The result of the simulation is propagated back up the tree, updating the values of all nodes along the path. This helps refine future decisions.

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

  MCTS uses Upper Confidence Bound (UCB1) to decide which nodes to explore. The formula is:

$$UCB1 = \frac{Q_i}{N_i} + C\sqrt{\frac{\ln N}{N_i}}$$

  Where:

  - Q_i = total reward of node i
  - N_i = number of times node i was visited
  - N = total visits to the parent node
  - C = exploration constant (controls trade-off between exploration & exploitation)

  Exploration vs. Exploitation:

  - **First term** $\left(\frac{Q_i}{N_i}\right)$ Represents exploitation (favoring nodes with high average reward).
  - **Second term** $\left(C\sqrt{\frac{\ln N}{N_i}}\right)$ Represents exploration (favoring nodes that havent been visited much).

  If a node is highly rewarding, it gets chosen more often (exploitation). But if a node is underexplored the second term ensures it still gets considered (exploration).

  This balance allows MCTS to avoid local optima and discover the best strategy over time.

1.2.2 Tree Policy and Rollouts

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

  Running just one simulation can be misleading because the outcome depends on random factors that may not reflect the true strength of a position. Instead, running multiple simulations helps in several ways:

  Reduces randomness: A single rollout could be an outlier, leading to bad decisions. More simulations smooth out extreme results.

  Gives a more reliable estimate: By averaging results, we get a better approximation of a positions true value.

  Improves decision confidence: If one action consistently leads to better outcomes over many simulations, its a stronger indicator of a good move.

  Handles uncertainty better: Some positions might look bad at first but turn out favorable in deeper play. More rollouts help uncover this.

  In short, multiple simulations prevent MCTS from making decisions based on pure luck and ensure more stable and reliable choices.

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

  Random rollouts help MCTS estimate how good or bad a position is without requiring an explicit evaluation function. Heres why they are useful:

  They approximate future rewards: Since MCTS doesnt calculate exact position values, rollouts simulate future play to get an idea of long-term outcomes.

  They guide exploration: If rollouts from a particular move often result in wins, MCTS learns to favor that move.

  They prevent short-sighted decisions: Some actions may look good immediately but lead to bad outcomes later. Rollouts capture long-term potential.

  They work in complex games: In games without a clear evaluation function (like Go or general board games), rollouts act as a simple way to estimate a moves strength.

  Even though they are random, rollouts provide valuable hints that help MCTS make smarter decisions over time.

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, the policy network and value network work together to improve search efficiency and decision-making. Instead of purely relying on simulations, the policy network suggests likely good moves, while the value network estimates the strength of a given board position.

  Policy Network: Instead of expanding all possible moves, the policy network gives a probability distribution over actions, helping MCTS focus on promising ones.

Value Network: Traditionally, MCTS uses rollouts (simulated random plays) to estimate how good a position is. Instead, the value network directly predicts the long-term outcome, reducing the need for full rollouts.

Search Efficiency: The policy network guides the selection of moves, reducing unnecessary exploration, while the value network speeds up backpropagation by providing an immediate evaluation rather than relying on simulated results.

Training: These networks are trained using self-play, where MCTS searches refine the policy, and game outcomes refine the value network, making them progressively stronger.

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

  During Expansion, MCTS adds new nodes to the search tree, representing new game states after taking an action. The policy network influences this step by assigning prior probabilities to moves, ensuring that the tree grows in more relevant directions.

  Focus on Strong Moves: Instead of randomly picking a move to expand, MCTS prioritizes moves with higher prior probabilities, which makes the search more efficient.

  Reduces Computational Waste: Instead of exploring every possibility equally, the policy network helps MCTS focus on moves that are more likely to lead to a win.

  Guides UCB Calculation: These priors are combined with visit counts in the Upper Confidence Bound (UCB) formula, balancing exploration (trying new moves) and exploitation (favoring moves that already seem strong).

  Improves Learning Over Time: Since these priors come from a trained policy network, the system improves as it plays more games, refining its search efficiency over time.

  By integrating the policy network into Expansion, Neural MCTS becomes much faster and smarter than traditional search methods.

1.2.4 Backpropagation and Node Statistics

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

  Backpropagation in MCTS updates the search tree after a simulation finishes, ensuring that past decisions are informed by new results. The updates happen as we move back up the tree from the simulated games outcome to the root node.

  Visit Count Update: Each node along the simulation path increments its visit count. This tracks how often a state has been explored, which helps balance exploration and exploitation.

  Value Update: The estimated value of a node is updated using the outcome of the simulation. Typically, the node's value is updated as the average of all previous values observed in simulations passing through it.

  Discounting (Optional): If needed, the reward from deeper nodes can be discounted to prioritize short-term results.

  Propagation to Root: These updates continue recursively up to the root, refining the decision-making process for future searches.

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

  Since MCTS runs many simulations, we must combine results properly to prevent misleading statistics.

  Avoid Overweighting a Single Outcome: If we simply replace values instead of averaging, a single lucky or unlucky simulation could skew the entire trees evaluation.

  Smooth Learning: Averaging the values across visits ensures stability, preventing wild swings in decision-making.

  Better Long-Term Predictions: When multiple simulations pass through the same node, summing rewards directly could inflate values, making nodes look artificially strong. Using an average helps the system converge toward a realistic estimate.

  Balanced Exploration vs. Exploitation: Correct aggregation ensures the UCB formula correctly prioritizes which nodes to explore next, improving overall search quality.

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, c, controls the balance between trying new moves (exploration) and choosing moves that already seem strong (exploitation).

  Higher c: Encourages more exploration, meaning the search tries a wider variety of moves before committing to a strong choice. This is useful when the model is uncertain or when facing a complex position.

  Lower c: Focuses more on known good moves, making the search more greedy and efficient but potentially missing better alternatives.

  Tuning the Exploration Constant: If the AI plays too randomly, lower c to make it focus on the best-known moves.

  If the AI keeps repeating the same mistakes or missing creative plays, increase c to encourage more exploration.

  Often tuned experimentally, but values around 1.0 to 2.5 work well in many cases.

  In the CartPole environment, tuning c depends on balancing stability vs. discoverya lower c helps refine balancing strategies, while a higher c ensures diverse experiences, improving robustness to different conditions.

  In CartPole, a moderate to low exploration constant (c) generally works better. Since the environment has simple dynamics and a clear optimal policy (keeping the pole balanced), excessive exploration can lead to unnecessary random actions that destabilize learning.

  A low c helps the agent refine precise balancing strategies, while a high c might be useful early on to explore diverse trajectories but should be reduced over time for stability.

- 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 controls how deterministic or random the AI's final move selection is.

High temperature: Makes move selection more random, giving lower-ranked moves a better chance of being chosen. Useful for early training to encourage diverse play and learning.

Low temperature: Moves are chosen more strictly based on their visit count, making the AI act more decisively.

Why lower temperature as training progresses? Early on, randomness helps explore different strategies.

Later, as the AI improves, a lower temperature helps exploit learned knowledge and play more reliably.

In competitive settings, temperature is often set very low or zero so the AI plays its absolute best move.

By adjusting these parameters carefully, we can balance learning, creativity, and strong decision-making in an AI system.

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?

  Minimax with alpha-beta pruning searches systematically, evaluating every possible move down to a certain depth and using a heuristic function to estimate outcomes. This works well in structured games like chess but struggles in extremely deep or complex game trees.

  MCTS, on the other hand, doesnt require full-depth exploration. Instead, it randomly samples promising moves, focusing more on rewarding paths instead of blindly searching everything. This makes MCTS more flexible and adaptive, especially when the game tree is too large to fully analyze.

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

  No need for a perfect evaluation function Unlike minimax, which relies heavily on a handcrafted heuristic, MCTS learns good moves on the fly by simulating gameplay.

  Handles large, complex states efficiently In games with massive possibilities (e.g., Go), MCTS selectively prioritizes the best-looking moves rather than evaluating everything.

  Adapts dynamically Instead of committing to a fixed depth like minimax, MCTS keeps improving the search as long as time allows, meaning it can work well under limited computation resources.

  Exploration-exploitation balance Using UCB (Upper Confidence Bound), MCTS balances trying new moves with refining known strong moves, making it better suited for games without clear-cut strategies.

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?

  Increasing the number of planning steps in the Dyna-Q algorithm significantly affects the learning process by improving efficiency and convergence speed. Heres how:

  Faster Convergence More planning steps mean the agent updates its Q-values using stored experiences rather than just real interactions. This allows it to generalize faster and learn optimal policies with fewer actual environment interactions.

  Better Sample Efficiency Instead of relying solely on new experiences, the agent revisits past experiences and learns from them repeatedly. This is especially useful in environments where collecting new samples is costly or slow.

  Improved Stability With more planning steps, the Q-values stabilize faster because the agent reinforces its learning with simulated updates, reducing variance in policy improvement.

  Diminishing Returns While adding more planning steps accelerates learning initially, after a certain point, the improvements become marginal. The agent may spend too much time refining existing knowledge rather than exploring new states.

  Potential Overfitting If the model is inaccurate, excessive planning steps can reinforce incorrect assumptions, leading to suboptimal decisions. This is more likely in stochastic environments where transitions are uncertain.

- 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 without adjusting our deterministic algorithm, we would face several challenges:

  Poor Policy Performance The algorithm assumes that taking an action leads to a specific outcome, but in a slippery environment, the result is unpredictable. This mismatch causes the agent to make incorrect decisions.

  Increased Variance Since actions have uncertain outcomes, the same action in the same state may lead to different results. The agent might struggle to learn a stable policy as Q-values fluctuate.

  Slower Convergence The agent will take longer to learn because it needs more samples to correctly estimate the value of each action in a stochastic setting. It may repeatedly learn incorrect values due to randomness.

  Suboptimal Path Selection In a deterministic setting, the agent learns the shortest or most efficient path. However, in a slippery environment, blindly following that path could lead to falls or unintended movements, making the learned policy unreliable.

Possible Policy Collapse If the agent overly trusts its Q-values (which are based on deterministic updates), it may repeatedly attempt actions that dont work in practice, leading to cycles of failure.

What Can Be Done? To handle slipperiness, we should modify our approach:

Introduce exploration strategies like ϵ greedy to adapt to uncertainty.

Use a stochastic model that accounts for different outcomes of the same action.

Adjust learning parameters (e.g., slower learning rate) to allow more stable updates.

Without these changes, the agent will likely struggle in a slippery environment and fail to find an optimal policy.

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

  Planning definitely helps in this environment, but its not a magic fixit depends on how predictable the world is. Imagine learning to walk across a frozen lake. If you could sit and mentally rehearse every step before actually moving, youd learn faster. Thats what planning doesit lets the agent practice in its head without physically interacting with the environment every time.

  Since the reward in this game (reaching the goal) is rare, planning helps spread that reward information back to earlier steps, so the agent doesnt need to stumble around as much. It also helps reinforce good decisions, even if they happen by chance at first.

  But theres a catch. If the environment is too unpredictable (like a really slippery lake), the agent might build a misleading model of how things work. If its plan assumes it has perfect control, but in reality, it keeps slipping in random directions, all that mental rehearsal wont help much.

  So, does planning help? Yes, because it speeds up learning and helps the agent make smarter choices earlier. But if the environment is too chaotic, the agent might end up overconfident in a strategy that doesnt actually work.

- 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.

  Effect on N_1 (First Time Reaching the Goal)
  Before reaching the goal for the first time (N_1), the agent mostly experiences bad triestaking suboptimal paths, falling into traps, or failing to reach the goal efficiently. A higher number of planning steps (n) helps in the following ways:

  Learning from Mistakes Faster

  Every failed attempt updates the model with bad transitions (low reward or falling into a hole).

  With high n, these bad experiences get replayed multiple times, reinforcing the idea that those actions should be avoided.

  Propagating Knowledge Backward

  When the agent stumbles upon a good action, high n ensures that its benefits are learned quickly, even for earlier states.

  This means the agent wont need to repeat as many bad tries before finding a better path.

  Smarter Exploration

Since bad experiences get reinforced more often, the agent is less likely to retry poor decisions, allowing it to discover the optimal route faster.

At the start, the agent is clueless. It has to explore a lot, trying different moves and sometimes failing. A higher n means the agent spends more time thinking about what it has already seen, reinforcing useful experiences. This can reduce N_1, since the agent learns faster from its limited experience. But if n is too low, it relies mostly on trial-and-error, making N_1 larger.

Effect on N_2 (Second Time Reaching the Goal)
Once the agent reaches the goal for the first time, things change. It now has some idea of what works, and planning helps refine that knowledge. A higher n makes sure the agent updates its Q-values more efficiently, meaning it can reuse past knowledge better. This significantly reduces N_2, making the second successful attempt much faster than the first.

In short, increasing n helps the agent get smarter between actual interactions. It speeds up learning and makes the second success come much sooner than the first. But if n is too low, progress stays slow, and the agent has to rediscover good moves repeatedly.

2.2.2 Improvement Strategies

Explain how each of these methods might help us with solving this environment: I tried all the softmax policy worked well for me as it is available in my note book.

- Adding a baseline to the Q-values.

  Adding a baseline to the Q-values means initializing them with a higher value rather than zero. This encourages the agent to explore more because initially, every action seems promising. In the Frozen Lake environment, where some paths lead to holes (failure), optimistic initialization helps the agent quickly learn which actions are bad by trying them and getting negative feedback. This speeds up the learning process since the agent moves away from bad choices faster. Instead of getting stuck in loops of suboptimal moves, it explores better routes early on, leading to faster convergence to the optimal policy.

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

  Gradually reducing epsilon over time allows the agent to explore more in the beginning and then exploit what it has learned later. This helps the agent avoid unnecessary exploration when it already has a good understanding of the environment. Alternative policies, like softmax or UCB, allow smarter exploration by favoring more promising actions rather than random choices, making learning more efficient.Using alternative policies like softmax or UCB can balance exploration and exploitation more effectively, improving learning efficiency.

- Changing the number of planning steps n over time.

  Increasing n as the agent learns can speed up convergence. Early on, low n allows quick updates without overcommitting to uncertain knowledge. As the model becomes more accurate, higher n ensures deeper planning, refining decisions based on a better understanding of the environment. This balances exploration and exploitation effectively.

- Modifying the reward function.

  Changing the reward function is a powerful way to influence an agents learning process and improve its efficiency in solving an environment. In standard reinforcement learning setups like Frozen Lake,

the agent typically receives a reward of +1 for reaching the goal and 0 otherwise. However, this sparse reward structure can make learning slow, as the agent might struggle to discover successful paths purely through random exploration.

One approach to improving learning is reward shaping, where additional rewards or penalties are introduced to guide the agents exploration. For example, adding a small negative reward (e.g., -0.01) for each step taken encourages the agent to reach the goal in fewer steps. This helps prevent excessive wandering and accelerates convergence. Conversely, increasing the penalty for falling into holes (e.g., from 0 to -1) teaches the agent to avoid dangerous states more effectively.

Another method is progress-based rewards, where the agent receives intermediate rewards for moving closer to the goal. This can be done by assigning higher values to states that are part of the optimal path or rewarding transitions that reduce the estimated distance to the goal. Such modifications make the learning process smoother and reduce reliance on random exploration.

However, modifying the reward function must be done carefully. If the added rewards introduce unintended incentives, the agent may develop counterproductive behaviors, such as looping in high-reward areas rather than completing the task efficiently. Proper tuning and testing are essential to ensure the modified rewards align with the desired policy.

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

  In traditional Dyna-Q, planning updates are performed uniformly across all state-action pairs, regardless of their importance or relevance to the current learning process. This approach can be inefficient, especially in environments where certain state-action pairs have a more significant impact on learning than others. Prioritized Sweeping is an improvement over this basic approach that allows the planning function to prioritize the most important state-action pairs, thus making the planning phase more efficient and faster in convergence.

  Prioritized Sweeping works by identifying and updating the state-action pairs that are expected to have the largest impact on the value function, thereby focusing on the most crucial parts of the state space first. The key idea is that the state-action pairs that are likely to experience the greatest changes in their Q-values should be updated more frequently, while less significant updates can be deferred. This prioritization can be particularly beneficial in sparse-reward environments, where the agent might spend a lot of time exploring areas that dont significantly affect the goal.

  The method uses a priority queue (often a max-heap) to store state-action pairs, where each pair is associated with a priority score. The priority is typically determined based on how much the Q-value of a state-action pair is likely to change given the most recent experience. State-action pairs with high priority are updated more often in the planning phase, and as a result, the agent is more likely to quickly learn the most important aspects of the environment.

  How Prioritized Sweeping Helps:

  Faster Convergence: By focusing on state-action pairs that are expected to change significantly, the agent can quickly correct any poor decisions made in these pairs. This leads to faster learning, as the most crucial parts of the policy are refined sooner. For example, if the agent encounters a state where it often fails, prioritizing this state allows it to learn more effectively and avoid repeating the same mistakes.

  Improved Exploration: Prioritized Sweeping does not ignore the exploration of other state-action pairs but instead ensures that those state-action pairs that are most likely to lead to useful discoveries

or corrections are explored first. This prevents unnecessary exploration of less useful or non-impactful state-action pairs, saving computational resources.

Better Handling of Large State Spaces: In environments with large state spaces, randomly updating all state-action pairs can be inefficient and time-consuming. Prioritized Sweeping allows for a more targeted approach, focusing updates on areas of the state space that matter most, which is particularly useful in complex environments with many irrelevant or less critical states.

Reducing Redundant Updates: Without Prioritized Sweeping, the agent might repeatedly update state-action pairs that have already converged, wasting time and computational effort. By assigning priorities based on how much a state-action pairs value is likely to change, Prioritized Sweeping minimizes redundant updates, improving the efficiency of planning.

Adapting to Dynamic Environments: In environments where the dynamics can change over time, Prioritized Sweeping helps the agent adapt quickly to new conditions by prioritizing the most relevant state-action pairs that are affected by these changes. This is particularly useful in non-stationary environments, where the agent needs to continuously adapt its policy based on new experiences.

How it Works: When the agent experiences a transition (i.e., a state-action pair, the next state, and the reward), the difference between the current Q-value and the expected Q-value is used to update the priority of the corresponding state-action pair. If a significant change is expected (due to a high TD error), that pairs priority is increased. The priority queue then ensures that these high-priority pairs are updated more often during the planning phase.

Example of Prioritized Sweeping in Practice: Consider a scenario where an agent is navigating a grid world. It may initially focus on a few critical states (e.g., near obstacles or near the goal) that are known to have high impact on the learning process. As the agent learns, the priority queue helps it focus updates on the state-action pairs that lead to major policy improvements, avoiding excessive updates to areas of the state space that are less important at that stage.

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 more iterations the LQR algorithm performs, the more accurate the solution will be. In the context of MPC, where you are constantly re-optimizing the control inputs based on updated system states, a higher number of iterations could help improve the precision of the control inputs.

  However, there's a trade-off: increasing the number of iterations can result in better accuracy but also takes more time to compute. So, if you have many iterations, the optimization might take longer to solve, potentially slowing down the control process.

  MPC requires solving an optimization problem at each time step, and if you're using LQR as the solver, the number of iterations required to reach the optimal solution can impact how fast the controller can react. More iterations typically mean more computational time, and this could be a problem in systems that need real-time responses or have limited computational resources.

  If you're running multiple iterations to achieve a highly accurate solution, there may not be enough time to update the control signal before the next iteration.

  Fewer LQR iterations can make the optimization process faster, which is useful when the system needs to make quick decisions. However, it might lead to less accurate solutions, potentially making the MPC less effective at controlling the system in a highly optimized way.

On the other hand, more iterations can produce more robust and refined control signals, leading to better long-term performance. But this also means the system might react slower, which could be undesirable for highly dynamic systems.

as you can see in the bellow code by decresing the number of itteration the pendluem the number of pisode to find the optimal policy and reaching the zero reward increases and it converges to zero later.

- What if we didn't have access to the model dynamics? Could we still use MPC? If we didn't have access to the model dynamics, traditional Model Predictive Control (MPC) would face challenges because it typically relies on a mathematical model of the system's behavior to predict future states and optimize control inputs. However, there are still ways to use MPC without direct access to the model:

  Data-driven MPC:

  In this approach, we use historical data or real-time measurements to approximate the system's dynamics. Techniques like system identification can be applied to learn the model from the data over time. This learned model can then be used in an MPC framework to make predictions and control decisions.

  Learning-based MPC:

  You can use machine learning models (like neural networks) to learn the dynamics of the system. These models, once trained, can approximate the system's behavior and serve as a substitute for traditional physical models in the MPC optimization process.

  Nonlinear MPC (NMPC):

  If the system is nonlinear and we lack an exact model, NMPC can still be applied with an approximate model or even a learned surrogate model. In this case, the optimization would still proceed, but it might be based on simpler approximations or black-box models that don't directly reflect the exact dynamics.

  Reinforcement Learning (RL):

  RL-based control methods, which share similarities with MPC, can be used to control systems without requiring explicit dynamics. Here, an agent learns the control policy by interacting with the environment, learning from the outcomes rather than using a fixed model.

- Do `TIMESTEPS` or `N_BATCH` matter here? Explain. Yes, both TIMESTEPS and N_BATCH can have an impact on how MPC performs, especially in the context of learning-based or data-driven MPC approaches.

  TIMESTEPS:

  TIMESTEPS refer to how far ahead the controller looks in the future to predict and optimize the system's behavior. A larger number of timesteps means that the controller considers a longer horizon when making decisions. This can improve the performance, especially in systems with long-term dynamics, but it also increases the computational load.

  If the timesteps are too large, the predictions may become less accurate, especially if the system dynamics are uncertain or not fully understood. On the other hand, too few timesteps might lead to short-sighted decisions that don't account for the broader context of the system. as you can see in bellow code both very short and long time steps wiill lead to bad learning.

N_BATCH:

N_BATCH refers to the number of parallel computations or simulations performed during the optimization process. In the context of data-driven MPC or machine learning models, having a larger batch size can help the model generalize better by using more data points to make decisions. However, larger batches require more memory and computational resources.

In reinforcement learning, for example, a larger batch size might help the model learn faster, but it can also lead to slower convergence or overfitting if not tuned properly. Smaller batch sizes, while requiring less memory, might result in less stable training or slower learning.

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

  Stability: The downward position is typically an unstable equilibrium point. By starting the system there, we test how well the controller or algorithm can stabilize the system. If it can recover from the downward position, its likely to be able to handle other scenarios with less instability.

  Challenge for the Controller: The downward position provides a "worst-case" starting point, meaning that the controller has to work harder to bring the system into a stable state (like the upright position). This is a good way to test the robustness and effectiveness of the control strategy, especially in systems that involve balancing or stabilization.

  Learning and Performance: For learning algorithms, like reinforcement learning or model predictive control (MPC), starting from an unstable position forces the algorithm to explore more of the state space. This encourages the algorithm to learn how to perform well under challenging conditions, improving its generalizability.

  Physical Relevance: In many real-world applications, the system may naturally be in the downward position, especially when starting from a resting state. For example, a pendulum left to hang will start in the downward position, making it a natural choice for many experiments or simulations.

- As time progresses (later iterations), what happens to the actions and rewards? Why As time progresses in later iterations, the actions and rewards typically show a pattern of improvement due to the learning and adaptation of the system, especially in reinforcement learning or model-based control like MPC.

  Actions: Early on, the actions might be more exploratory or random, especially if the model hasnt learned the dynamics well. Over time, as the system collects more data, the actions become more refined and optimal. The model begins to make decisions that are more targeted at achieving the desired outcome, such as balancing the pendulum upright or stabilizing the system. So, the actions should converge to a more efficient policy as the model learns.

  Rewards: Initially, rewards might be low because the system is far from the optimal state and may make inefficient decisions. However, as the system learns and adapts to its environment, it should start taking actions that bring it closer to the goal (e.g., balancing a pendulum), resulting in higher rewards. The rewards are a reflection of how well the system is performing based on its actions, so as time progresses, we generally see the rewards increase.

  This improvement is due to better control policies being learned, where the system increasingly takes actions that bring it closer to the desired goal, which in turn results in higher rewards. The periodic behavior in the reward at the beginning is likely due to the pendulum oscillating back and forth before stabilizing. Heres why:

  Initial Exploration and High Energy Swings

At the start, the controller applies forces to swing the pendulum up, causing oscillatory motion.

These swings lead to fluctuating rewards because the reward function is based on how well the pendulum aligns with the upright position and minimizes control effort.

Underactuated System Dynamics

The pendulum initially lacks enough control authority to reach the goal smoothly.

It follows a natural oscillation pattern as the MPC refines the control actions over time.

MPC Adjusting Actions

Since Model Predictive Control (MPC) optimizes actions over a horizon, the controller starts with suboptimal moves, causing oscillations.

As iterations progress, MPC learns better actions, and the oscillations dampen, leading to more stable rewards.