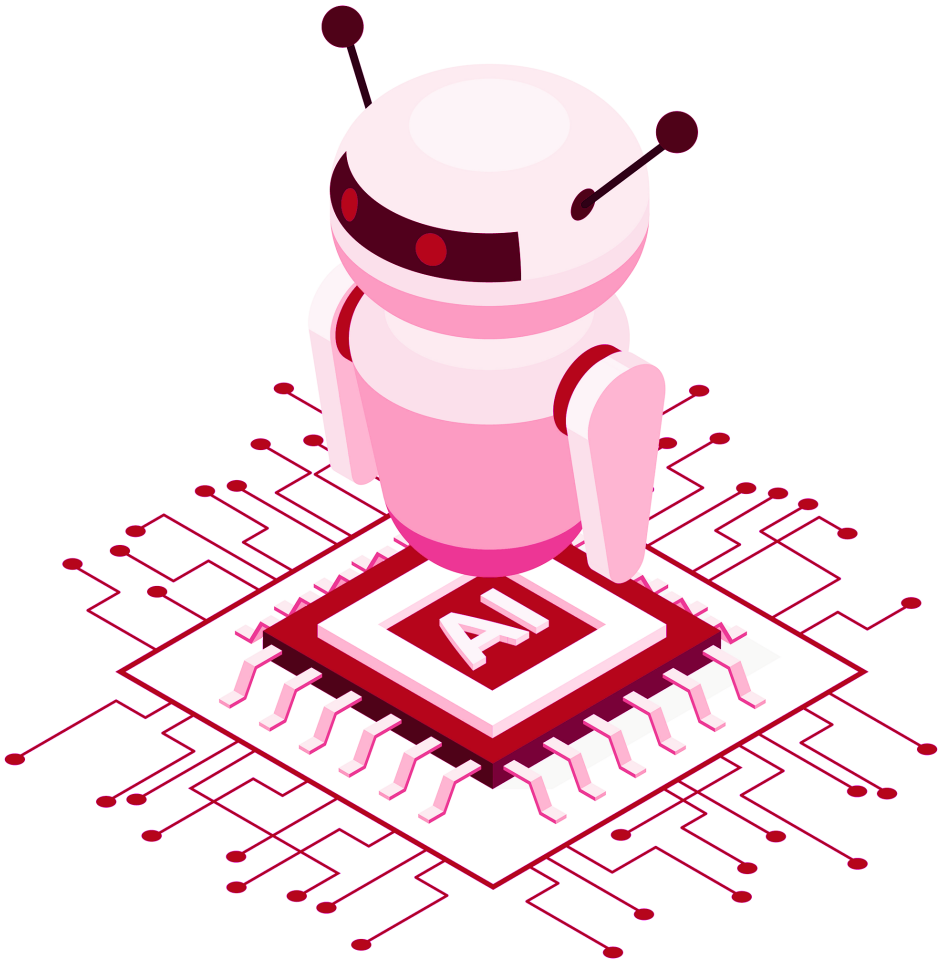

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 5:

Model-Based Reinforcement Learning

By:

Ayeen Poostforoushan

401105742

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?

  It starts with Selection, where the algorithm navigates the tree from the root to a promising leaf node. Once it reaches a leaf node, the Expansion phase starts, adding a new child node by selecting one of the possible unexplored moves. Then the rollout happens, where the algorithm simulates the dynamics of the environment using the leaned model. It starts from the new node and continues in the modeled environment to get an estimate of how good that state is. Finally, in backpropagation, the value of the rollout is passed back up the tree, updating the values of all the nodes along the path. This way, the tree gradually gets better at identifying strong moves over multiple iterations.

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

  UCB = (wi / ni) + C * sqrt(ln(N) / ni)

  where: - wi is the total reward of child node i. - ni is the number of times node i has been visited. - N is the total number of visits to the parent node. - C is a constant that controls exploration versus exploitation.

  the first term, (wi / ni), represents exploitation by favoring moves with higher average rewards. The second term, C * sqrt(ln(N) / ni), encourages exploration by giving a higher score to less-visited nodes, ensuring that the algorithm continues to try new options instead of always choosing the best-known move too early.

1.2.2 Tree Policy and Rollouts

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

  To get a better result with less randomness. Because every planning step that we do just expands one leaf. We need to run the leaf selection and the next part of the algorithm a lot of time to get a good estimate of the rewards of each state in the MC tree.

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

  They assume that the model of the env is real and by doing a lot of number of rollouts the values converge to the low variance estimations of the actual value of the position in the real environment (if the model is accurate). They continue the leaf state with random actions to return a value for that state because it is not possible to reach the termination at every iteration.

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 guides move selection in the tree nodes by predicting probabilities for each action, replacing random exploration with a more informed search. The value network replaces rollouts by estimating the win probability of a position, making simulations faster and more efficient. Instead of running full random games, MCTS directly uses this estimate during backpropagation to update node values.

- 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?

  In the Expansion phase, the policy network's output provides prior probabilities for each possible move, helping MCTS focus on the most promising actions. When a node is expanded, its child nodes are initialized with these probabilities, which influence their exploration priority. Instead of treating all moves equally, MCTS uses these priors to bias the search toward moves that the policy network considers strong. This reduces unnecessary exploration of weak moves and speeds up learning by directing simulations toward better decisions.

1.2.4 Backpropagation and Node Statistics

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

  We simply add one to the visit counts of all the nodes from the root to the leaf. Also we calculate the discounted rewards that start from the rollout and add it by discounting it more when moving towards the root.

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

  Because the aggregated outcomes directly affect the selection probability of that node in the next simulations too. So if you wrongly aggregate it you ruin the next simulations of the planning too.

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?

  It directly affects the entire model's decisions exploration. If the constant is low, the action sequences that initially have higher returns get selected more in the tree search and be exploited. If we see unexplored paths that are good we might need to increase this constant.

- 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 influences how deterministic or random the final move selection is. A higher temperature smooths out the probabilities, making the moves more evenly likely and encouraging exploration, while a lower temperature sharpens the distribution, making it more likely that the move with the highest probability is chosen. As training progresses, lowering the temperature can help the agent focus on exploiting the best-known strategies rather than exploring too many less optimal moves.

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 and classical minimax search handle deep game trees very differently. MCTS uses random simulations to explore only the best moves without examining every possibility, which makes it more flexible for complex or very deep trees. In contrast, minimax (even with alpha-beta pruning) looks at every move up to a fixed depth and relies on pruning to cut off branches, which can become slow or less effective when the tree is huge.

- 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 offers unique benefits in extremely large state spaces or when no accurate heuristic is available. It doesn't rely on a handcrafted evaluation function but instead runs many simulations to assess moves. This allows the algorithm to concentrate on parts of the game tree that are likely to yield good results without having to explore every possibility in detail, making it well-suited for complex or hard to model environments.

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.

Answers are in the notebook.

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?
- What would happen if we trained on the slippery version of the environment, assuming we **didn't** change the *deterministic* nature of our algorithm?
- Does planning even help for this specific environment? How so? (Hint: analyze the reward signal)
- 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.

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.
- Changing the value of ε over time or using a policy other than the ε-greedy policy.
- Changing the number of planning steps n over time.
- Modifying the reward function.
- Altering the planning function to prioritize some state–action pairs over others. (Hint: explain how **Prioritized Sweeping** helps)

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.

Answers are in the notebook.

3.2.1 Analyze the Results

Answer the following questions after running your experiments:

- How does the number of LQR iterations affect the MPC?
- What if we didn't have access to the model dynamics? Could we still use MPC?
- Do `TIMESTEPS` or `N_BATCH` matter here? Explain.
- Why do you think we chose to set the initial state of the environment to the downward position?
- As time progresses (later iterations), what happens to the actions and rewards? Why