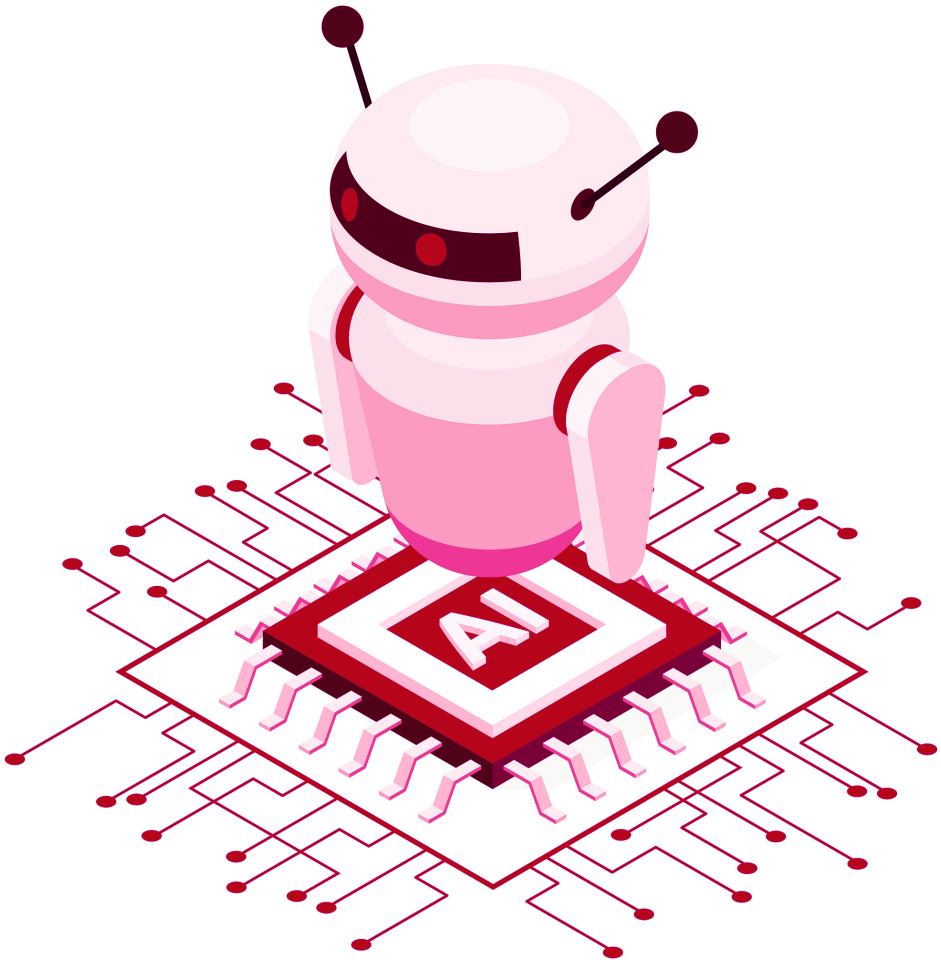

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 5:

Model-Based Reinforcement Learning

By:

Danial Parnian

401110307

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

Declaration: Some answers are written with assistance of GPT-4. However, none are directly copied (unless explicitly mentioned) and all the codes and responses are written manually.

1 Task 1: Monte Carlo Tree Search

1.1 Task Overview

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

The key components of this implementation include:

1.1.1 Representation, Dynamics, and Prediction Networks

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

1.1.2 Search Algorithms

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

1.1.3 Buffer Replay (Experience Memory)

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

1.1.4 Agent

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

1.1.5 Training Loop

1. **Step 1**: Collects trajectories through environment interaction.
2. **Step 2**: Stores experiences in the **replay buffer**.
3. **Step 3**: Samples sub-trajectories for **model updates**.

4. **Step 4**: Unrolls the learned model **over multiple steps**.
5. **Step 5**: Computes **loss functions** (policy, value, and reward prediction errors).
6. **Step 6**: Updates the neural network parameters.

Sections to be Implemented

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

1.2 Questions

1.2.1 MCTS Fundamentals

- What are the four main phases of MCTS (Selection, Expansion, Simulation, Backpropagation), and what is the conceptual purpose of each phase?
  answer: In Selection, we traverse the tree from the root to a leaf node by selecting child nodes based on a policy (e.g., UCB). The goal is to reach a node that needs further exploration. In Expansion, we expand the tree by adding that leaf's child nodes. This phase adds new possible actions and states into the tree. In Simulation, we perform a rollout from the newly expanded node to a terminal state or a predefined depth. The purpose is to simulate the outcome to estimate the value of the new node. Finally in Backpropagation, we propagate the results of the simulation back up the tree, to update the value estimates and visit counts of the nodes through the path. This refines the value estimates and improves the policy for future iterations.
- How does MCTS balance exploration and exploitation in its node selection strategy (i.e., how does the UCB formula address this balance)?
  answer: The UCB formula contains two terms: an exploitation term which is the average value of the node, which encourages selecting nodes with high value estimates, and an exploration term which is based on the visit count of the node and its parent, and this term encourages the selection of less-visited nodes to explore new possibilities. In our implementation this term is $[\log((parent.visit\_count + c2 + 1)/c2) + c1] \times \frac{\sqrt{parent.visit\_count}}{(child.visit\_count+1)}$. By adding these two terms, it balances exploration and exploitation.

1.2.2 Tree Policy and Rollouts

- Why do we run multiple simulations from each node rather than a single simulation?
  answer: It helps to reduce variance and a more reliable estimate of the node's value by averaging out the randomness. This leads to more robust and consistent training.
- What role do random rollouts (or simulated playouts) play in estimating the value of a position?
  answer: Random rollouts estimate the value of a node by simulating future actions until a terminal state or a fixed depth is reached. The accumulated rewards from these rollouts provide an approximation of the expected return, which improves value estimates.

1.2.3 Integration with Neural Networks

- In the context of Neural MCTS (e.g., AlphaGo-style approaches), how are policy networks and value networks incorporated into the search procedure?
  answer: The policy network provides prior probabilities for each action at a given state, which guide the selection process instead of using only the UCB formula. The value network estimates the value

of a state, which is used to evaluate leaf nodes in the search tree, which is used instead of the rollout mechanism.

- What is the role of the policy network's output ("prior probabilities") in the Expansion phase, and how does it influence which moves get explored?
  answer: The policy network's prior probabilities are incorporated into the UCB-based selection formula used in MCTS. Specifically, in AlphaGo, the PUCT formula is used: $Q(s,a)+cP(s,a)\frac{\sqrt{N(s)}}{1+N(s,a)}$, where $Q(s,a)$ is the estimated values, $P(s,a)$ is the prior probability from the policy network, $N(s)$ is the total visit count of the parent node, and $N(s,a)$ is the visit count of action a.

1.2.4 Backpropagation and Node Statistics

- During backpropagation, how do we update node visit counts and value estimates?
  answer: We increment the visit count of each node along the path to the root. We update the value estimate by averaging the returned evaluation (from the value network or simulation) with previous values, which improves the node's estimated return.
- Why is it important to aggregate results carefully (e.g., averaging or summing outcomes) when multiple simulations pass through the same node?
  answer: Careful aggregation ensures that value estimates remain stable and representative of actual returns. Averaging helps smooth out noise from individual simulations, preventing biased updates, while proper summation of visit counts maintains accurate exploration statistics, ensuring balanced action selection in future searches. [1]

1.2.5 Hyperparameters and Practical Considerations

- How does the exploration constant (often denoted c_{puct} or c) in the UCB formula affect the search behavior, and how would you tune it?
  answer: The exploration constant c in the UCB formula balances exploration and exploitation during the search. A higher c encourages more exploration by increasing the weight of the exploration term. In our implementation, we had two parameters c1 and c2. I tried values similar to the MuZero-like approaches [2] and also experimented different values.
- In what ways can the "temperature" parameter (if used) shape the final move selection, and why might you lower the temperature as training progresses?
  answer: The temperature parameter controls exploration in action selection by adjusting the probability distribution over actions. A higher temperature results in more exploration, while a low temperature increases exploitation. Lowering the temperature as training progresses helps the agent transition from exploration to more deterministic and optimal action selection. The effect is similar to epsilon-decay mechanism in epsilon greedy policy.

1.2.6 Comparisons to Other Methods

- How does MCTS differ from classical minimax search or alpha-beta pruning in handling deep or complex game trees?
  answer: It differs from classical methods by using random simulations (rollouts) to estimate the value of a position, rather than searching the entire game tree (or use a heuristic evaluation after reaching depth limit). This makes MCTS more suitable for handling deep or complex game trees. Also the action selection mechanism is smarter and focuses on exploring more promising actions to

find better estimate of them, whereas in classical methods we tried to find a fixed value for each node, and focused on all nodes equivalently.

- What unique advantages does MCTS provide when the state space is extremely large or when an accurate heuristic evaluation function is not readily available?
  answer: As mentioned above, using rollouts, we dont need an accurate heuristic evaluation. Additionally, when state space is extremely large, classical methods can inspect a very limited depth, but in MCTS we prioritize important states and find good evaluation for them. MCTS balances exploration and exploitation better.

2 Task 2: Dyna-Q

2.1 Task Overview

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

Sections to be Implemented and Completed

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

2.1.1 Planning and Learning

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

2.1.2 Experimentation and Exploration

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

2.1.3 Reward Shaping

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

2.1.4 Prioritized Sweeping

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

2.1.5 Extra Points

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

2.2 Questions

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

2.2.1 Experiments

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

- How does increasing the number of planning steps affect the overall learning process?
  answer: Increasing the number of planning steps improves the learning process by refining the model of environment with better estimates, which leads to a more precise model, and therefore faster convergence to an optimal policy.
- What would happen if we trained on the slippery version of the environment, assuming we **didn't** change the *deterministic* nature of our algorithm?
  answer: Setting is_slippery=True introduces stochasticity into the environment. If the algorithm remains unchanged, it leads to suboptimal performance, since we rely on the model for q-updates in planning steps but the model would predict wrong deterministic next_action and rewards.
- Does planning even help for this specific environment? How so? (Hint: analyze the reward signal)
  answer: Planning helps in this environment by allowing the agent to simulate and evaluate potential future states and actions without actual interaction. This makes the learning more efficient. The higher the planning steps are, the more quickly model finds optimal policy.
- Assuming it takes N_1 episodes to reach the goal for the first time, and from then it takes N_2 episodes to reach the goal for the second time, explain how the number of planning steps n affects N_1 and N_2.
  answer: The number of planning steps does not affect N_1, since we have no information about q-values before reaching the goal for the first time, and planning updates are meaningless before that. However, after that planning steps help us find the optimal path more quickly, and higher n can significantly reduce 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.
  answer: it encourages exploring unseen states since they have higher Q-values. This helps the agent to reach the goal more quickly, which leads to faster convergence. However, it doesn't help much here. when we set a positive baseline, since the q-values for frozen cells dont change, they always stay positive and model might tend to go towards them and get stuck. when we set baseline a negative number, first episodes take too long and it's not more efficient than the normal case.
- Changing the value of ε over time or using a policy other than the ε-greedy policy.
  answer: epsilon decay allows the agent to explore more in the beginning and exploit more as it learns, leading to faster convergence. I use a linear decay for ε from 1.0 to 0.1. However, in this problem it doesn't affect much. because the hard part is reaching the goal for the first time where all the q-values are 0, and with changes I made to greedy_policy (pick random among all maximums), the epsilon does not affect the problem before finding goal for the first time, as policy is basically

random. After reaching goal for the first time, we quickly find optimal policy so epsilon decay has little effect (it just helps to explore different states and find optimal policy at more states, which could be done by setting higher epsilon value in the normal case).
Also I used a softmax policy instead of ε-greedy. This allows for more exploration of actions with higher Q-values while still allowing for some exploration of lower Q-value actions. As you can see in reward plot, it speeds up the convergence and helps significantly.

- Changing the number of planning steps n over time.
answer: Changing the number of planning steps over time can help by allowing the agent to initially focus on learning from real experiences and gradually increasing the amount of planning as it gains more knowledge about the environment. This can lead to more efficient learning and faster convergence.

- Modifying the reward function.
answer: I provided a custom reward function that penalizes each step taken a small amount. This encourages the agent to find the shortest path to the goal. Additionally, I provided a larger reward for reaching the goal and a penalty for falling into a hole. This helps the agent learn faster using more informative rewards.

- Altering the planning function to prioritize some state–action pairs over others. (Hint: explain how **Prioritized Sweeping** helps)
answer: I used Prioritized sweeping with a threshold of 0.1 and added important state/action pairs to priorities for planning on them, which helped focus on important updates. This led to faster learning. As you can see, the comulatve average grows faster. Prioritized sweeping allowed the agent to concentrate on the most significant and effective state-action pairs, making learning more efficient.

3 Task 3: Model Predictive Control (MPC)

3.1 Task Overview

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

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

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

Sections to be Implemented and Completed

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

3.2 Questions

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

3.2.1 Analyze the Results

Answer the following questions after running your experiments:

- How does the number of LQR iterations affect the MPC?
  answer: The number of LQR iterations affects the convergence speed and accuracy of the MPC. More iterations generally lead to better control performance, but also increase computation time. Here if we set it too low, some examples might not converge. Additionally increasing it more than 10 doesn't help much.
- What if we didn't have access to the model dynamics? Could we still use MPC?
  answer: f we didn't have access to the model dynamics, we could not use traditional MPC directly. However, we could learn the dynamics model from data and use it for mpc.
- Do `TIMESTEPS` or `N_BATCH` matter here? Explain.
  answer: Yes, they matter. TIMESTEPS determines how many future steps the MPC looks ahead, and when we set it too low, the agent might not be able to see the future and plan accordingly. According to my experiments, if we set it less than 8 here, the agent won't be able to learn anything. Also N_BATCH determines how many parallel simulations we run, which can speed up the optimization process. However, to use higher N_BATCH, we should change the code a bit to support parallelization (define N different environments and run them in parallel, which I didn't do here).
- Why do you think we chose to set the initial state of the environment to the downward position?
  answer: Setting it to the downward position makes the task more challenging, as the agent needs

to learn to swing up from a stable position. This shows the effectiveness of the MPC in a more difficult scenario.

- As time progresses (later iterations), what happens to the actions and rewards? Why?
  answer: As time progresses, the actions and rewards stabilize. The agent learns to apply the right torque to keep the pendulum upright, and then actions and rewards converge to 0, as you can see in the total reward plot.

References

[1] This paragraph is generated by ChatGPT.

[2] Schrittwieser et al., (2019). Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. Available: https://arxiv.org/abs/1911.08265.