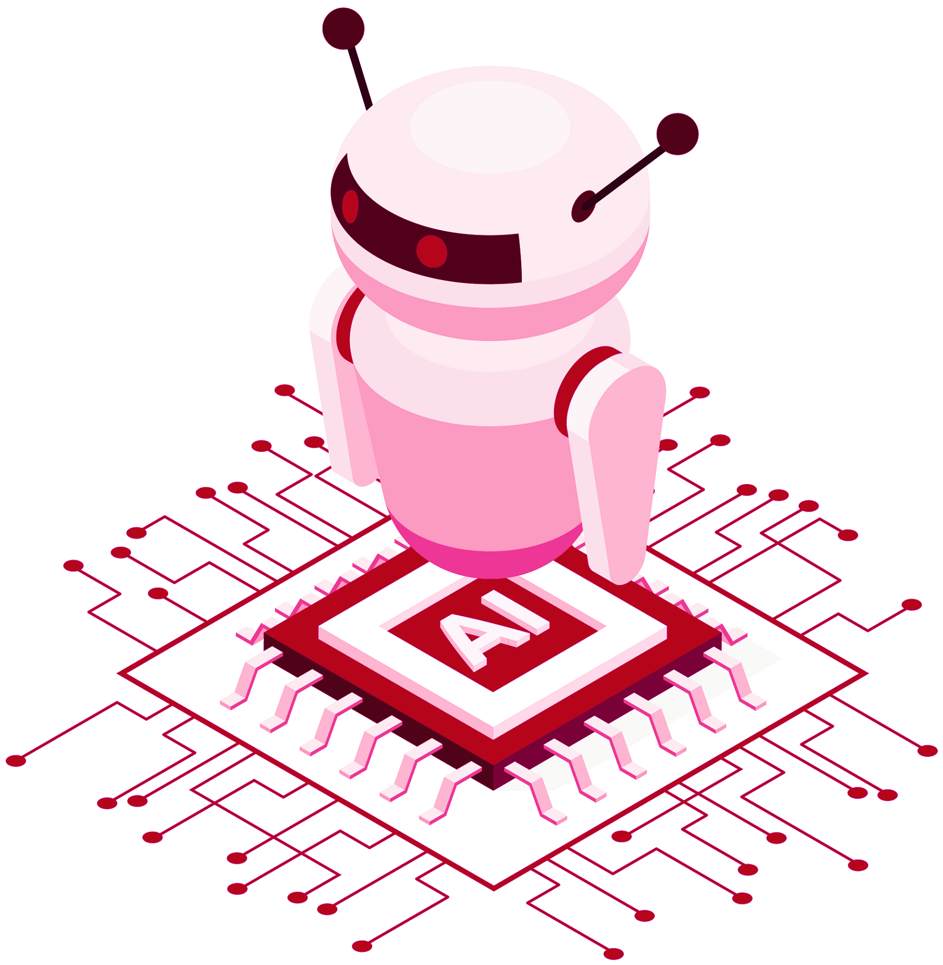

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 5:

Model-Based Reinforcement Learning

By:

Ali Ghasemzadeh

401106339

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

1.2.2 Tree Policy and Rollouts

- Why do we run multiple simulations from each node rather than a single simulation?
- What role do random rollouts (or simulated playouts) play in estimating the value of a position?

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?
- 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?

1.2.4 Backpropagation and Node Statistics

- During backpropagation, how do we update node visit counts and value estimates?
- Why is it important to aggregate results carefully (e.g., averaging or summing outcomes) when multiple simulations pass through the same node?

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

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

1.3 Answers

1.3.1 MCTS Fundamentals

Selection : In this phase you traverse the tree by picking child nodes accorfing to the rule that weighs both how promising a move looks and how little you've tried it, this means exploration and exploitation. Here the UCB fomula help to decide which branch to explore next.
Expansion : when reaching a node that hasn't been explored much, you expand the tree by adding one or more new nodes.
Simulation : From the newly added node we perform a rollout which is a simulation where we play the game randomly or with a simple policy until the end. This give us an approximation idea of the outcome if we follow that branch of moves, and because the perfect evaluation is often too expensive these rollouts proved a quick estimate of how good that state might be.
Backpropagation : At the end we have result from the simulation and update nodes along the path we we took. each node's statisitics is the many time it was visited and cumulative reward are updated. In this way future decisions are informed by these outcomes and gradually refine the estimated values for each move.
Balancing Exploration and Exploitation with UCB : The UCB formula is :

$$\text{Value} = Q(s,a) + c \times \sqrt{\frac{\ln(N)}{n}}$$

Q(s,a) is the average reward and c is a constant to determine how much to explore and the second term boosts moves that haven't been tried much.

1.3.2 Tree Policy and Rollouts

Multiple Simulations per Node : Running several simulations from each node instead of running just one time makes the decision better and reduce the misleading result due to randomness and gives us a more reliable estimate of how good a position really is.
Role of Random Rollouts : instead of calculating the exact value of every state, rollouts help approxmate that value. They are especially useful in environments where calculating a perfect evalutation is either too complex or simply unavailable.

1.3.3 Integration with Neural Networks

In Neural MCTS :
Policy Networks : provide a prior probabilities for each possible move. during the expansion phase, instead of randomly picking a move to to expand, we use these priors to focus on moves that the network believes are good.
Value Networks : instead of relying just on random rollouts to evaluate a position, the value network gives an estimate of the state's quality directly. This speeds up the evaluation since not needing to simulate all the way every time.
Role of Prior Probabilites : When you expand the tree, the policy network's output tells us which moves are more likely better using the biased selection towards moves that have higher prior probabilites.

1.3.4 Backpropagation and Node Statistics

Updating Node Statistics : During the backprop each node's visit count is increased and the outcome of the simulation is added to its cumulative score, during the time we average these outcomes ti get a

reliable estimate of node's value.
Why Aggregate is important : When several simulation pass through the same node, we want to ensure the statistics that accurately represent the overall performace of the state. Aggregating results helps smooth the noise from any individual simulation, and make our estimation more robust.

1.3.5 Hyperparameters and Practical Considerations

Exploration Constant c : This constant controls how mcuh you prioritize exploring the nodes that are less visited. A higher constant means you want to explore more which can be useful when you are still unsure about the best moves, but it is a trade-off and choosing it much big may cause waisting time on moves that unlikely to be good.
Temperature Parameter : the temperature parameter affects how deterministic our final move selection is, if it is high the choice is more random and at first it is good but after a part of time when our model learns it is better for it to be small.

1.3.6 Comparisons to Other Methods

Classical methods like minimax or alpha-beta pruning that we became familiar with in AI course at first require evaluating the entire game tree of a big part of it which is computationly heavy both for time and both for memory especially for complex games and MCTS builds the tree gradually and focuses on good moves, it can deal with huge state space.
Unique Advantages of MCTS :
Handling Leage State Space : because it builds the tree gradually and focuses on promising moves, so it can deal with enormous state spaces without needing a perfect heuristic.
No Need for an Accurate Heuristic : when you don't have a reliable evaluation function, MCTS can still work well by sing random rollouts or learned estimate of values from your neural network.
Flexibility : the algorithm adapts as more simulations are run and improves the estimates continuously and don't need to evaluate the whole space again.

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?
- 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)

2.3 Answers

2.3.1 Effect of Increasing the Number of Planning Steps

increasing the number of planning steps n improves efficiently of learning because the Q-values are updated by agent's direct experience and sinmulated experience, but increasing it too much lead to diminishing returns or may cause overfitting in inaccurate models.

2.3.2 Training on the Slippery Version Without Changing the Deterministic Algorithm

The standard Dyna-Q assumes a deterministic model this means it simulates transitions exactly as observed and in the slippery version, state transitions are stochastic, so with a deterministic model we will have inaccurate simulation.

2.3.3 Does Planning Help in Frozen Lake

It works if we reach the goal for this environment because it is sparse reward but the exploration space is big and for the 8x8 table it doesn't converge wit planning and it needs a large number of exploration, but for 4x4 it converges and solve the game.

2.3.4 Effect of Planning Steps on N_1 and N_2

N_1 is the episodes to first reach the goal, with more planning steps, N_1 decrease because the agent updates more states per real-world transition, and it spreads useful information faster.
N_2 is the episodes between first and second success and if planning is effective then N_2 also decreases.

1. Adding a Baseline to the Q-values
This can reduces variance and speed up convergence in learning but after we reach the goal at least one time, but it can be useful.

2. Changing ϵ Over Time or Using a Different Policy
The ϵ-greedy over time allows the agent to explore initially and gradually shift towards exploitation and make the learning process better and I use Boltzman exploration in my code to converge better.

3. Changing the Number of Planning Steps n Over Time
Using a higher n helps to spread reward faster across states, and reducing it when the agent has reasonable model avoids overfitting so we can use large n at first and gradually decrease it for better stablizing.

4. Modifying the Reward Function
Since the frozen lake is sparse reward environment, small change in the reward function can improve learning: shaping rewards that give small reward is a method, also we can penalizing falls and give negative reward for stopping into a hole, after all I give negative reward for holes and a small negative reward for each action to prevent the agent from staying in one place.

2.3.5 Prioritized Sweeping

standard Dyna-Q selects random past experience for planning and it is not efficient so we prioritize important state-action pairs with leage Q-value updates. This speeds up learning by focusing on states where agent's knowledge is changing rapidly, in the frozen lake it avoids the wasting updates or unimportant transitions.

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?
- 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

3.3 Answers

3.3.1 Effect of the Number of LQR Iterations on MPC

When I increase the number of LQR iterations it generally leads to better local approximation of the optimal control law. more iterations mean that the MPC can refine the estimate of how the system will evolve under control, which usually results in smoother and more accurate actions. But after a point additional iterations not improve the returns and increase the computation time.

3.3.2 Using MPC Without Direct Access to the Model Dynamics

MPC fundamentally relies on a model to predict future states and optimize the actions. If we don't have access to the true dynamics, we can't run a perfect MPC. However we could still use MPC if we can construct a surrogate model and use techniques ro learn dynamics using data. So also the performace might decrease we can learn a model that can still enable us to use an MPC framework.

3.3.3 Role of TIMESTEPS and N-BATCH

TIMESTEPS : It is the prediction horizon and the more it is the MPC can anticipate future behavior. a short horizon may miss longer-term effects, while an overly long horizon increases computational load without a good benefit.
N-BATCH : It relates to how many trajectories pr samples we process in parallel during optimization. It doesn't in general change the quality of the control nut it can improve the computational efficiency and robustness of the optimization process, especially when using parallel computations.
In my ovservation TIMESTEPS has more effect than N-BATCH.

3.3.4 Initial State Set to the Downward Position

Starting of the pendulum is in the downward position and it is a classical challenge. The downward position is a challenge because we want ti swing it up and stabilize it in the upward position, and this cause it to be a nontrivial task.

3.3.5 Changes in Actions and Rewards Over Time

As the MPC iterates over time, we observe that the control actions become more refined and less aggresive. The controller might use larger or more erratic actions as it figures out the system's behavior. with time, as it gathers more feedback and optimizes the predictions, the actions settle into a smoother pattern that reliably brings the pendulum closer to the desired state (rewards are measure of how far the system is from its target) improve as the system stabilizes means that w esee higher rewards in later iterations and this improvement reflects the controller's convergence towards a more optimal control strategy.