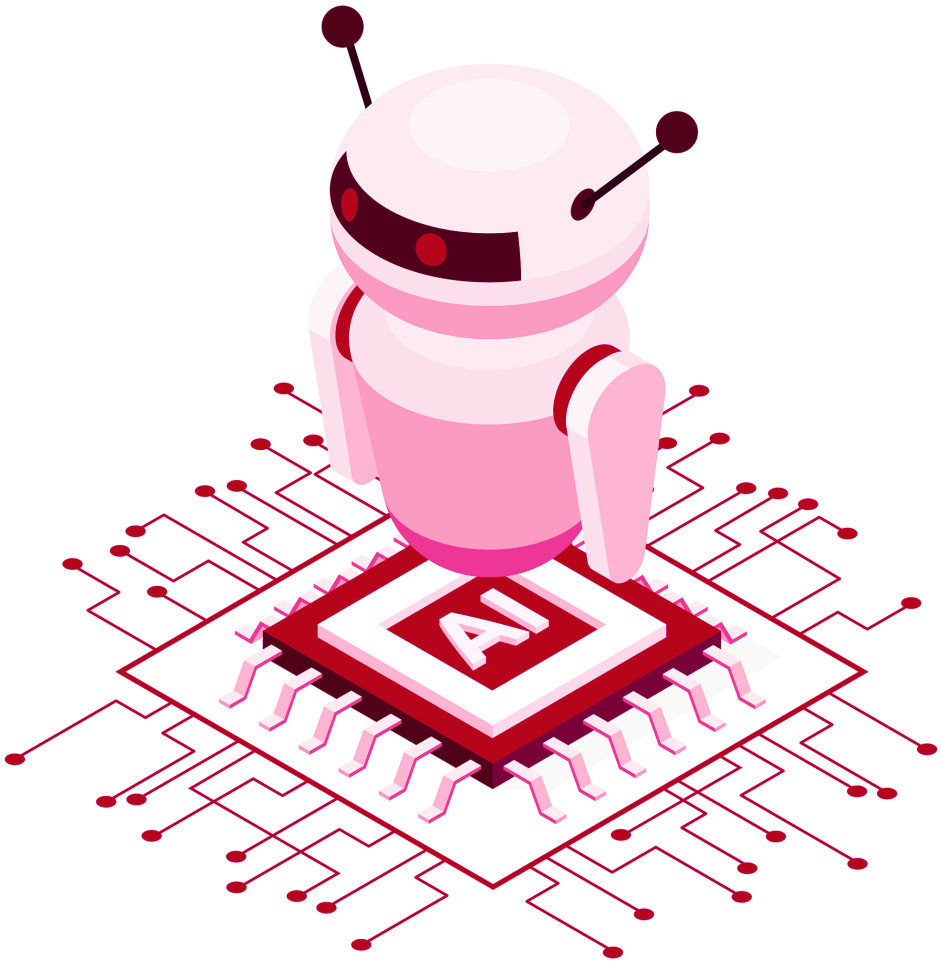

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 5:

Model-Based Reinforcement Learning

By:

Ashkan Majidi

400109984

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?
- 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

- **Four main phases of MCTS:**
  - **Selection**: Traverse the tree from the root node to a leaf node using a tree policy (e.g., UCB) to balance exploration and exploitation.
  - **Expansion**: Grow the tree by adding a new child node if the current node is expandable (i.e., not fully expanded).
  - **Simulation (Rollout)**: Perform a playout from the newly expanded node to a terminal state using a default policy (e.g., random actions).
  - **Backpropagation**: Update the statistics (e.g., visit count, value estimate) of all nodes along the traversed path based on the simulation outcome.
- **UCB Formula:** The Upper Confidence Bound (UCB) formula balances exploration and exploitation by considering both:
  - The average reward (exploitation term).
  - An exploration bonus proportional to $\sqrt{\frac{\ln N}{n}}$, where N is the parent's visit count and n is the child's visit count.

  The exploration constant c scales the exploration term.

1.3.2 Tree Policy and Rollouts

- **Multiple Simulations:** Running multiple simulations reduces variance in value estimates by averaging outcomes, leading to more reliable decisions.
- **Random Rollouts:** Random rollouts provide a fast but approximate estimate of a node's value when no prior knowledge (e.g., a learned value function) is available.

1.3.3 Integration with Neural Networks

- **Neural MCTS:**
  - A **policy network** guides the search by providing prior probabilities for actions during expansion.
  - A **value network** replaces rollouts by directly estimating the expected return from a given state.
- **Prior Probabilities:** The policy network's output biases the search toward promising actions, making MCTS more efficient by prioritizing high-value branches.

1.3.4 Backpropagation and Node Statistics

- **Updating Node Statistics:**
  - Increment the **visit count** for each traversed node.

  - Update the **value estimate** by averaging (or summing) the backpropagated returns.

- **Aggregating Results:** Careful aggregation ensures that nodes reflect accurate long-term value estimates, preventing overfitting to noisy simulation outcomes.

1.3.5 Hyperparameters and Practical Considerations

- **Exploration Constant (c_{puct}):**
  - A higher c encourages more exploration.
  - Tuning involves balancing between discovering new actions and exploiting known high-reward paths.
- **Temperature Parameter:**
  - High temperature flattens action probabilities, encouraging exploration early in training.
  - Lowering temperature sharpens the policy, favoring the highest-value actions as training progresses.

1.3.6 Comparisons to Other Methods

- **MCTS vs. Minimax/Alpha-Beta:**
  - MCTS does not require a full-depth search or heuristic evaluation function.
  - It handles large state spaces by focusing on promising subtrees.
- **Unique Advantages of MCTS:**
  - Scalability in large/complex trees due to asymmetric growth.
  - Robustness when no heuristic is available, relying instead on sampling.

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 Experiments

- **Increasing Planning Steps:**
  - In our training procedure it doesn't affect much because we are working with *FrozenLake* environment which has a sparse reward system and our model didn't reach even for one time in both cases so doesn't matter how we set the number of planning steps. We should do something that helps us with this sparse nature.
  - But generally (not in our case) more planning steps improve learning efficiency by better propagating value estimates through the model. However, excessive planning can lead to diminishing returns or overfitting to an imperfect model.
- **Slippery Environment with Deterministic Algorithm:**

  - Performance would degrade significantly because the deterministic model would fail to account for stochastic transitions.
  - The agent would make suboptimal decisions by assuming perfect control over movement.

- **Usefulness of Planning:**

  Note: because our pure Dyna-Q does not reach the goal in practice we don't see even on time reaching the goal and followng data are based on improvement strategies for dyna-Q

  - Planning helps in this environment by propagating sparse reward signals (only at the goal) to intermediate states.
  - Without planning, the agent would require many more episodes to discover the goal through random exploration.

- **Effect on N_1 and N_2:**
- **Usefulness of Planning: Note:** because our pure Dyna-Q does not reach the goal in practice we don't see even on time reaching the goal and followng data are based on improvement strategies for dyna-Q
  - Higher n reduces N_1 by accelerating initial learning through simulated experience.
  - N_2 decreases further as planning reinforces successful paths after the first goal discovery.

2.3.2 Improvement Strategies

- **Adding a Baseline to Q-values:**
  - Reduces variance in updates and stabilizes learning, especially useful in FrozenLake sparse-reward setting but if was not enough for our problem. as you see in the notebook it helped the Q values to have more realistic look but it suffers from lack of adjusting exploration rates.
- **Adaptive ε or Policy:**
  - Decaying ε balances exploration (early) and exploitation (late). Alternative policies (e.g., Boltzmann) can provide smoother action selection.
  - in my experiments it was not enou
- **Dynamic Planning Steps (n):**
  - Increasing n over time focuses computation where the model is more accurate. Early low n avoids wasteful planning with poor initial estimates.
  - by implementing this we see that no change happened but adding baseline and epsilon decay together with this will help to reach good results.
- **Reward Function Modification:**
  - Shaping rewards (e.g., penalties for holes, proximity-to-goal bonuses) guides exploration. Mitigates credit assignment problems in sparse-reward environments.
  - Agent learned optimal policy by doing this in the experiments
- **Prioritized Sweeping:**

- Focuses planning on state-action pairs with large Bellman errors, accelerating value propagation.
- More efficient than uniform sampling, especially in large state spaces.
- Agent learned optimal policy by doing this in the experiments

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

implementation of these parts are in notebook. please check the notebook for more experimental information.

- **LQR Iterations Impact:**
  - As you see in the notebook more iterations improve control policy convergence at the cost of computation time
  - Too few iterations may lead to suboptimal control actions
  - The sweet spot balances solution quality with real-time requirements

- **Unknown Model Dynamics:**
  - MPC requires some model approximation - could use:
    - Learned dynamics models (neural networks)
    - System identification techniques
    - Gaussian processes for uncertainty modeling
  - Pure model-free approaches would lose MPC's predictive advantages
- `TIMESTEPS` **and** `N_BATCH` **Significance:**
  - `TIMESTEPS`:
    - Longer horizons improve stability but increase computation
    - Must cover the pendulum's swing-up transient dynamics
    - As you can see in the notebook it took much more time for the learning procedure by increasing `TIMESTEPS`
  - `N_BATCH`:
    - Enables parallel solving of multiple trajectories
    - Larger batches improve GPU utilization but require more memory
- **Initial Downward Position:**
  - Represents the most challenging starting condition
  - Tests the controller's ability to handle maximum potential energy
  - Verifies robustness against worst-case initialization
- **Time Progression Effects:**
  - Actions become smaller and more precise as the pendulum stabilizes
  - Rewards increase then stabilize near the upright equilibrium
  - MPC continuously adjusts for small disturbances (noise/friction)