Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Solution for Homework 5:

Model-Based Reinforcement Learning

Designed By:

Naser Kazemi

naserkazemi2002@gmail.com

Ramtin Moslemi

ramtin4moslemi@gmail.com

RIML

Spring 2025

Contents

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. The four main phases are:

  1. **Selection:** From the root node, iteratively select child nodes according to a tree policy (often using a UCB-based formula) until reaching a leaf node.
  2. **Expansion:** If the current leaf node is not terminal, generate one or more children (possible next states) to expand the tree.
  3. **Simulation (or Rollout):** From the expanded node, run a (random or policy-guided) simulation until a terminal state to estimate the value of the position.
  4. **Backpropagation:** Propagate the simulation result back up the tree, updating statistics such as visit counts and value estimates for each node along the path.

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

  Answer. MCTS commonly uses a UCB-based formula to select child nodes. One variant is given by:

  $$\text{UCB} = Q_i + c \times P_i \times \frac{\sqrt{\sum_j N_j}}{1 + N_i},$$

  where Q_i is the estimated value of node i, N_i is the visit count of node i, $\sum_j N_j$ is the total visits of all children, and c is an exploration constant. This approach adds a bonus for less-explored actions, thus balancing exploitation (high Q_i) with exploration (large uncertainty or high prior P_i).

1.2.2 Tree Policy and Rollouts

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

  Answer. Multiple simulations reduce the variance of value estimates for each node. A single simulation can be unrepresentative, while repeated rollouts refine the estimated outcome of taking a particular action, leading to more reliable value assessments over time.

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

  Answer. Random rollouts approximate the eventual outcome from a given state when no strong heuristic or neural network is available. By quickly simulating to a terminal state, they provide an empirical estimate of the position's goodness, which is then used to update the node's value during backpropagation.

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.

  – A **policy network** provides prior probabilities for possible actions at a given state. These priors guide the Expansion step by focusing on likely strong moves.

  – A **value network** estimates the expected outcome (e.g., win probability or reward) from a state. This value can replace or supplement random rollouts, providing a more informed evaluation than pure chance.

- 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. During Expansion, the policy network assigns a prior probability P_i to each possible move. In the UCB-based selection formula, moves with higher P_i receive a larger exploration bonus, thus being explored earlier or more frequently. This steers MCTS toward moves that the policy network predicts will be promising.

1.2.4 Backpropagation and Node Statistics

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

  Answer. Every time a simulation passes through a node:

$$N_i \leftarrow N_i + 1, \quad W_i \leftarrow W_i + (\text{simulation result}), \quad Q_i \leftarrow \frac{W_i}{N_i},$$

  where N_i is the number of visits, W_i is the total accumulated reward, and Q_i is the average value estimate of node i.

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

  Answer. Multiple simulations provide multiple data points for a node's outcome. Proper aggregation (summing or averaging) ensures the value estimate reflects all historical outcomes. Careless aggregation could bias the node's value, while correct aggregation reduces variance and improves accuracy over many simulations.

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.

  – A higher c emphasizes exploration, encouraging the search to try less-visited moves.

  – A lower c emphasizes exploitation, focusing on moves that currently appear most promising.

  – Tuning c typically involves experimentation, searching for a balance that avoids excessive exploration or over-commitment to a small set of moves.

- 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.

  – Temperature controls how peaked or uniform the final move distribution is. A high temperature produces more randomness in move selection, while a low temperature is more deterministic.

  – Lowering the temperature as training progresses reduces exploration, allowing the agent to converge on stronger moves rather than sampling broadly.

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. Minimax search with alpha-beta pruning systematically explores the game tree (up to a certain depth) and relies on a heuristic evaluation at leaf nodes. This becomes infeasible in extremely large state spaces. MCTS, on the other hand, incrementally refines estimates in promising parts of the tree and does not require a full-depth search or an expert heuristic at each node, making it well-suited for very large or complex problems.

- 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.

  – MCTS can operate with minimal domain knowledge, relying primarily on simulations to evaluate positions.

  – It directs computational resources to the most promising actions (based on visits and priors), while still exploring less-visited actions to avoid missing potentially strong alternatives.

  – It is an anytime algorithm, meaning it can be stopped at any point and still produce a valid policy based on the simulations done so far.

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.

Answer.

- With a sufficiently large model that encapsulates different states and actions, we can rely on very few episodes and instead learn the environment via planning steps. Since the environment is deterministic, there aren't many problems with increasing the number of planning steps and the only downside is the computational cost. For a stochastic environment however, the planning steps would be useful when our model closely corresponds to the real distributions of the environment.
- While our model of the environment would be wrong, it would still be possible for us to learn the environment. However this time increasing the number of planning steps would yield biased updates resulting in worse performance.
- Due to the sparse nature of the reward signal, finding the right trajectory could become very difficult. The problem is that the q-values only get updated when we reach the goal or a state where have previously reached the goal from. As a result the updates must propagate from the goal to the starting position. Planning steps don't require reaching these special states which help us update the q-values, instead because of their stochastic nature (sampling state-action pair at random), they help update the necessary q-values. As a result, planning steps help us speed-up convergence. Check out this example for more explanation.
- N_1 depends on the exploration policy (or exploration rate) alone whereas $N2$ is also affected by the convergence rate. When there are no planning steps ($n = 0$), upon reaching the goal for the first time, we only update the the q-value of the state-action pair which led us to the goal. If we increase n however, using the model we have, we can also update neighboring states without needing to reach them once more. In conclusion in large environments (with many states) with sparse rewards, it is fair to say $N_1 \approx N_2$ but if we use model-based learning with planning steps or n-step bootstrapping, we can expect $N_2 << N_1$.

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)

Answer.

- While there are many benefits to adding a baseline, including variance reduction and improved learning efficiency, in this example the baseline helps guide the exploration. Introducing a proper baseline will modify the exploration strategy by emphasizing actions that outperform the it. This mitigates the over-exploration of suboptimal or random actions.
- It makes sense to explore more in the beginning when your knowledge about the environment is more limited. But when you become more confident about your predictions, it makes sense to choose the best actions and reduce exploration. In short ε should decay overtime. Given that we're working with a sparse reward signal here, we can also exclusively explore until we reach the goal for the first time since it is not possible to exploit!
- As we collect more experience, our model of the environment becomes more complete. Planning steps help us most when we have a reliable model of the environment. Once again, since we are working with a sparse reward signal, performing planning steps doesn't help us because the q-values don't change.
- By using dense rewards (as opposed to sparse rewards) we can make use of the frequent updates to speed-up convergence. A common approach is to add a living reward of -1. This encourages the agent to find the shortest path to the goal.
- By working backward from states with significant value changes, prioritized sweeping ensures that updates propagate effectively through the model. This avoids unnecessary computations on less relevant transitions. It accelerates learning by concentrating on areas where updates will have the most influence, reducing the time needed to achieve optimal policies. In environments with large state spaces, prioritized sweeping allocates computational resources to the most critical areas, making it more scalable.

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

Answer.

- Linear-Quadratic Regulator (LQR) iterations are crucial for finding the optimal control sequence during each MPC update. Increasing the number of LQR iterations allows the algorithm to converge better on the optimal solution by iteratively improving the policy within the planning horizon. However, more iterations mean higher computational cost. If you limit the iterations, the control might be less optimal, resulting in reduced accuracy of predictions and actions taken by MPC.
- Classic MPC relies on knowing the dynamics of the system to predict future states. Without access to the model dynamics, you'd need to shift toward model-free approaches, such as reinforcement learning algorithms or use data-driven techniques like system identification (learning dynamics from observed data). Alternatively, you could implement approximate dynamics or hybrid approaches combining learned and analytical models to simulate MPC-like behavior.

- The `TIMESTEPS` control how far into the future MPC plans. A larger number gives the controller more time to stabilize and optimize its trajectory, but it also increases computational complexity. `N_BATCH` refers to the batch size in a sampling-based context (often used for environments that rely on stochastic processes or ensemble methods). If used, this value could influence how reliably MPC captures variability in system dynamics.
- Setting the initial state to the downward position adds a challenge: it requires the pendulum to move to the upright position, which involves overcoming gravity. This mimics real-world control problems where the system starts in a low-energy state and needs to be driven to a high-energy configuration. Additionally, it provides a measurable goal for the controller to optimize—the upright position is inherently unstable, making it an ideal target for testing MPC's performance.
- In later iterations:
  - **Actions** tend to stabilize because MPC refines its control strategy, minimizing unnecessary corrections as it learns the optimal behavior for the task.
  - **Rewards** generally increase as the pendulum achieves or maintains its upright position, reflecting better performance. This happens because MPC focuses on minimizing cost (e.g., energy usage and deviations from the desired trajectory).

  As the controller becomes more confident and precise over iterations, rewards plateau at optimal levels unless external disturbances challenge the system.