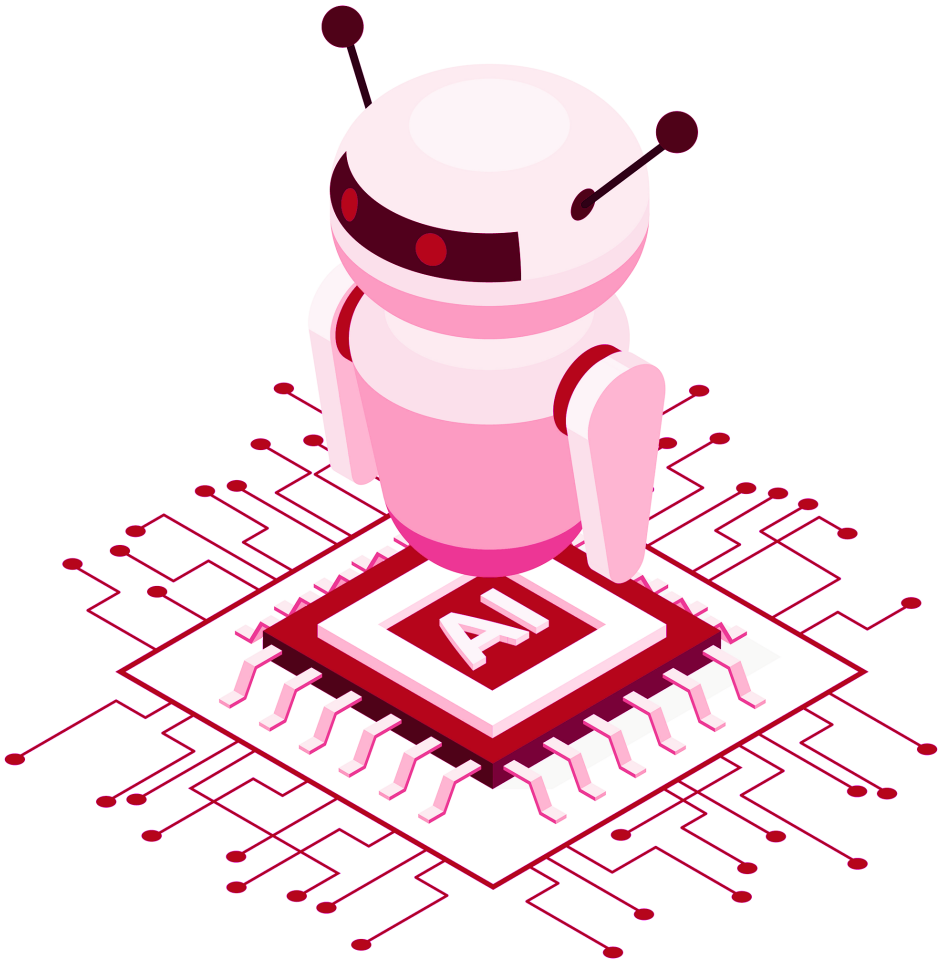

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 5:

Model-Based Reinforcement Learning

By:

Ali Najar

401102701

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?
  - **Selection:**

    starting from the root node, MCTS recursively selects child nodes according to a defined policy (using UCB formula) until reaching a node that has unexpanded children or is a terminal node. In the case of MuZero the UCB formula is like below:

    $$Q(s,a) + P(s,a) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)} \left(c_1 + \log \left(\frac{N(s) + c_2 + 1}{c_2} \right) \right)$$

    where:

    * $N(s)$ is the visit count of parent.
    * $N(s,a)$ is the visit count of child.
    * c_1, c_2 are constants controlling exploration-exploitation trade-off dynamics.

    Conceptual Purpose: To focus computational effort on the most promising nodes according to current knowledge (balancing known rewards and uncertainty).

  - **Expansion:**

    Once it reaches a node that's not fully explored (meaning it has children not yet created), MCTS expands this node by generating one or more new child nodes.

    Conceptual Purpose: This step increases the breadth of the search tree, giving the algorithm new possibilities to explore.

  - **Simulation:**

    From the newly created node, MCTS runs a simulation or "rollout" by playing randomly (or using a simple heuristic) to the end of the game or until a certain depth is reached.

    Conceptual Purpose: To estimate the potential reward of the expanded node by quickly approximating outcomes from that node onward.

  - **Backpropagation:**

    After the simulation phase, the result of the simulation is propagated back up through the tree, updating statistics (like total rewards and visit counts) at each visited node.

    Conceptual Purpose: To refine and update node evaluation metrics, allowing future selections to be informed by the latest gathered information.

Here's a pseudocode of the classical MCTS algorithm.

Algorithm 1 Monte Carlo Tree Search (MCTS)

procedure MCTS($rootNode$, $iterations$)
  for $i = 1, 2, \ldots, iterations$ **do**
    $node \leftarrow rootNode$
    ▷ Selection
    while node is fully expanded and not terminal **do**
      $node \leftarrow$ SELECTCHILD($node$)
    end while
    ▷ Expansion
    if node is not terminal **then**
      $node \leftarrow$ EXPAND($node$)
    end if
    ▷ Simulation (Rollout)
    $reward \leftarrow$ SIMULATE($node$)
    ▷ Backpropagation
    BACKPROPAGATE($node$, $reward$)
  end for
  return BESTCHILD($rootNode$)
end procedure
procedure SELECTCHILD($node$)
  return $\arg\max_{child \in children(node)}$

$$Q(child) + c\sqrt{\frac{\ln N(node)}{N(child)}}$$

end procedure
procedure EXPAND($node$)
  Choose an unvisited child $newNode$ of $node$
  Add $newNode$ to the tree
  return $newNode$
end procedure
procedure SIMULATE($node$)
  Perform random playout from $node$ to terminal state
  return reward/outcome of the simulation
end procedure
procedure BACKPROPAGATE($node$, $reward$)
  while $node \neq$ null **do**
    Increment visit count $N(node) \leftarrow N(node) + 1$
    Update total reward $Q(node) \leftarrow Q(node) + reward$
    $node \leftarrow$ parent of $node$
  end while
end procedure

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

- **Exploitation (the first term $Q(child)$)**:

  This term ensures the algorithm favors nodes that have produced good results historically. Nodes with higher win rates (or rewards) are more likely to be selected.

- **Exploration (the second term)**:

  This part of the formula gives nodes with fewer visits a higher value, ensuring that the algorithm continues to explore new or less-explored nodes. The term grows larger when a node is visited fewer times, thus pushing MCTS to explore unknown or less-known branches to potentially discover better options.

1.2.2 Tree Policy and Rollouts

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

  Because one simulation is not a good estimation of the average reward from that node. Running multiple simulations provides a more reliable, statistically significant estimate of a node's value. Since individual simulations include randomness, relying on only a single simulation can lead to misleading results due to random fluctuations or outliers. Multiple simulations help smooth out this randomness. Note that we want the expected value over all possible outcomes not just one of them.

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

  Random rollouts help approximate how favorable or unfavorable a given state might be by exploring possible future outcomes from that state. Although each individual rollout is typically random, aggregating results from many rollouts provides an effective estimate of how promising a node might be in terms of long-term success or failure. In other words, random rollouts give MCTS a practical way to evaluate positions without exhaustively searching the entire game tree.

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?

  How Policy and Value Networks Are Incorporated:

  - **Policy Network (prior probabilities):**

    Provides an initial estimate of move probabilities before exploration. This is known as the prior ($P(s, a)$). It also guides which nodes are expanded first during the Expansion phase of the search. Moreover, It reduces the breadth of exploration by biasing search toward promising moves, significantly improving computational efficiency.

  - **Value Network:**

    Estimates the expected outcome from a given state without needing extensive simulation. It also replaces traditional rollout steps, allowing evaluation of positions without full-depth simulations, thereby saving considerable computation.

Algorithm 2 Neural MCTS in AlphaGo-style approaches

procedure NEURALMCTS(s_0)
  for each simulation **do**
    $s \leftarrow s_0$
    Search(s)
  end for
  return action with highest visit count from root
end procedure

procedure SEARCH(s)
  if s is terminal **then**
    return outcome of s
  end if
  if s is not in Tree **then**
    $(\mathbf{P}, V) \leftarrow$ NeuralNetworkEvaluate(s)
    add s to Tree with priors $P(s, a) = \mathbf{P}_a$
    initialize visit count $N(s, a) = 0$, and value estimate $Q(s, a) = 0$ for all a
    return V
  end if
  Select action maximizing UCT-style formula:

$$a \leftarrow \arg\max_a \left[Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \right]$$

  $s' \leftarrow$ NextState(s, a)
  $v \leftarrow$ Search(s')
  Update statistics:

$$Q(s,a) \leftarrow \frac{N(s,a) \cdot Q(s,a) + v}{N(s,a) + 1}$$
$$N(s,a) \leftarrow N(s,a) + 1$$

  return v
end 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?

  As it can be seen in (2), when a new node (representing a state) is expanded, each possible move a from that node is assigned an initial prior probability from the policy network's output. These priors heavily influence the move selection in the MCTS via the PUCT (Predictor + Upper Confidence bound applied to Trees) formula. This way, moves with higher prior probabilities $P(s, a)$ receive preferential exploration early on. Also focusing computational resources on likely good moves is another advantage.

1.2.4 Backpropagation and Node Statistics

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

Suppose we traversed a path:

$$_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{L-1}} s_L$$

and suppose v is the outcome of simulation (typically the output of value network at leaf node.).

Then, backpropagation updates each node along this path in reverse (from leaf node s_L back to root s_0) as mentioned in **line 21 of algorithm** (2).

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

  It's mainly due to these reasons:

  - **Accurate Estimation of Action Value ($Q(s,a)$):**

    $Q(s,a)$ is meant to estimate the expected value of choosing action a at state s. Each simulation gives a noisy or partial estimate of this value (since it samples a path). To get a reliable and stable estimate, we aggregate the simulation results:

    $$Q(s,a) \leftarrow \frac{1}{N(s,a)} \sum_{i=1}^{N(s,a)} v_i$$

    This ensures that with more simulations, $Q(s,a)$ converges to the true expected outcome from taking action a in state s, assuming good value estimates from the leaf nodes.

  - **Avoiding Bias from Early or Noisy Simulations:**

    Early simulations may pass through a node with random or biased outcomes. If we just replace the Q value (instead of averaging), we'd end up overfitting to the most recent simulation. Averaging ensures stability and robustness, especially important when value estimates have variance (as is common with neural value functions).

  - **Enabling Effective Selection in UCT/PUCT:**

    The PUCT selection formula combines:

    $$Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

    If $Q(s,a)$ isn't accurate (due to poor aggregation), the exploitation term misleads the search. Careful aggregation ensures $Q(s,a)$ truly reflects the average return and helps balance exploration vs. exploitation properly.

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?

  The exploration constant c_{puct} scales the exploration term. If it is too high search becomes too exploratory. If c_{puct} is too low search becomes too greedy.

  For example, in MuZero paper, a method was given in order to tune it properly:

  $$c_{puct} = \left(c_1 + \log \left(\frac{\sum_b N(s,b) + c_2 + 1}{c_2} \right) \right)$$

The constants c_1 and c_2 are used to control the influence of the prior $P(s,a)$ relative to the value $Q(s,a)$ as nodes are visited more often. In their paper, $c_1 = 1.25$ and $c_2 = 19652$.

As before it has some properies. the term $1 + N(s,a)$ in the denominator allows the agent to explore unseen nodes more. As it can be seen at the initial steps it acts like the original MCTS's UCB formulas. As the number of visits of a node s increases, the term $Q(s,a)$ might get us stuck in choosing an action that may not be the best. So we add a logarithmic term that increases with respect to $N(s)$. without this adjustment, the exploration bonus might decay too quickly relative to the overall accumulated knowledge. By increasing the bonus as the tree deepens, the algorithm avoids over-committing to a branch that might not be the best in the long run.

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

  In many neural MCTS implementations, after the tree search is done, we select a move to actually play (or train from) using a distribution over visit counts:

$$\pi(a|s) = \frac{N(s,a)^{\frac{1}{\tau}}}{\sum_b N(s,b)^{\frac{1}{\tau}}}$$

  The effect of high and low τ are mentioned below:

  - **High Temperature (e.g., $\tau \gg 1$):**
    * Flattens the distribution.
    * More exploratory: even rarely visited actions have a decent chance of being picked.
    * Good for diversity in training.
    * Used early in training or early in games.
  - **Low Temperature (e.g., $\tau \ll 1$)**:
    * Sharper distribution, approaches an argmax.
    * The most visited move is chosen almost deterministically.
    * Promotes exploitation of current knowledge.
    * Used later in training or in competitive settings.

  So it is good to lower the temperature as time goes by to exploit more.

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?

Feature	MCTS	Minimax / Alpha-Beta
Search Strategy	Stochastic, selective sampling of game tree paths	Deterministic, systematic traversal of the tree
Node Expansion	Expands nodes on-demand (only when needed)	Expands nodes to a fixed depth or all nodes within a subtree
Evaluation	Uses simulations or neural value functions to estimate outcomes	Uses handcrafted or static evaluation functions at leaf nodes
Memory Usage	Low — only stores visited nodes	High — may require large memory to store subtrees
Handling Branching Factor	Good — can scale to very high branching factors (e.g., Go)	Poor — struggles as branching factor increases
Best Suited For	Large, uncertain, or probabilistic domains (e.g., Go, RTS games)	Smaller, deterministic, perfect information games (e.g., Tic-Tac-Toe, Checkers)
Depth Control	Adaptive — goes deeper in promising paths	Fixed — goes to a specific depth regardless of promise
Exploration vs. Exploitation	Balances both dynamically using UCT/PUCT	No built-in exploration; fully deterministic

Table 1: Comparison between MCTS and classical minimax/alpha-beta search.

Table (1) gives a comprehensive comparison of the two mentioned methods. It can be seen that MCTS proves to be more effective than Minimax/ Alpha-Beta algorithms. Minimax is exhaustive and optimal (in theory), but MCTS is selective and scalable (in practice), especially when combined with learned models. Also MCTS can be adapted to both deterministic scenarios whereas the latter algorithm can only be used for deterministic environment.

- What unique advantages does MCTS provide when the state space is extremely large or when an accurate heuristic evaluation function is not readily available?
  - **Searches Without a Heuristic:**

    Classical algorithms (like minimax) require a hand-crafted evaluation function to score leaf nodes. MCTS can operate without a heuristic by using:

    * Rollouts (simulated playouts to a terminal state).
    * A learned value function (like in AlphaZero, MuZero).

    Advantage: You don't need domain knowledge to write an evaluation function.
  - **Selective, Asymmetric Tree Growth:**

    MCTS focuses its search on promising branches rather than expanding the tree uniformly. It goes deeper in good regions and shallower in weak ones.

    Advantage: Makes it feasible to explore more state/action spaces by investing effort only where it matters.
  - **Stochastic Sampling Allows Scalable Approximation:**

    Instead of evaluating every possible state (infeasible in large spaces), MCTS uses randomized simulations to approximate the value of moves. Over time, more simulations lead to increasingly

accurate value estimates.

Advantage: You get a good estimate without exhaustive search.

- **Using Neural Networks:**

  In neural MCTS (e.g., MuZero), you can learn a value function to replace rollouts and learn a policy prior to guide tree expansion.

  Advantage: You don't need handcrafted heuristics, the system learns them automatically via self-play and search.

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?
  - **Faster Propagation of Information:** With more planning steps, the agent uses its model to simulate additional transitions. This means that information about rewards and state transitions can be spread more quickly across the state space, leading to faster convergence of the value function or policy.
  - **Improved Sample Efficiency:** By leveraging simulated experiences, the agent can learn effectively even with limited real-world interactions. More planning steps amplify this benefit by generating a richer dataset from the model.
  - **Computational Trade-Off:** While more planning steps can speed up learning, they also increase the computational burden per update. In environments where computational resources or time are limited, there is a trade-off between learning speed and computation cost.
  - **Impact of Model Accuracy:** The effectiveness of planning steps relies on the accuracy of the learned model. If the model is well-calibrated, additional planning can lead to significant improvements. However, if the model is flawed, excessive planning might propagate errors, potentially leading to suboptimal learning.
- What would happen if we trained on the slippery version of the environment, assuming we **didn't** change the *deterministic* nature of our algorithm?
  - **Inaccurate Model Predictions:** The deterministic model will assume a fixed outcome for each action, even though the actual environment has multiple possible outcomes. This discrepancy means that planning steps will simulate transitions that rarely match real experiences.
  - **Suboptimal Policy Learning:** Since the planning phase propagates errors from the model, the value estimates may become skewed. The agent might overcommit to actions that look good under the deterministic assumption, even if they're less reliable in a stochastic setting.
  - **Slower Convergence or Divergence:** The mismatch between planned transitions and actual outcomes can slow down learning. In some cases, if the planning significantly misguides the value function updates, it may even lead to unstable or divergent behavior.
  - **Reduced Robustness:** A policy learned under deterministic assumptions may perform poorly when faced with the variability of a slippery environment, as it doesn't properly account for the uncertainty in state transitions.
- Does planning even help for this specific environment? How so? (Hint: analyze the reward signal)
  - **Amplifying Sparse Rewards:** In Frozen Lake, you usually only get a reward when you reach the goal, and zero otherwise. Planning allows the agent to simulate additional transitions, which can help spread the goal's reward value backwards through the state space. This means

that even though the reward is rare, its influence is amplified over many simulated steps, accelerating learning.

  - **Faster Credit Assignment:** Since real interactions provide limited feedback, planning can help assign credit to intermediate states more quickly. By using a model (even if it's imperfect due to the stochasticity), the agent can update the value estimates of many states that might eventually lead to the goal, guiding it towards more promising paths.

- 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.

  - **Before the First Goal (N1):** Initially, when the agent hasn't experienced the goal reward yet, planning is based on transitions that all have zero reward. Even if you increase n, the agent isn't propagating any useful reward signal—it's just reinforcing the fact that nothing special happens. This means that a higher n doesn't reduce N1 by much because the model's simulated experiences aren't yet providing any information about where the rewarding states are.

  - **After the First Goal (N2):** Once the agent finally reaches the goal and experiences the positive reward, planning becomes much more effective. With more planning steps, the reward received at the goal is propagated more rapidly backward through the state space. This accelerated credit assignment means that, in subsequent episodes, the value estimates for states leading to the goal are improved faster. Consequently, a higher n significantly reduces N2, as the agent quickly learns to follow paths that reliably lead to the reward.

2.2.2 Improvement Strategies

Answers to "Are you having any trouble?" part.

Explain how each of these methods might help us with solving this environment:

- Adding a baseline to the Q-values.

  Adding a baseline to the Q-values can stabilize and normalize the learning process, especially in an environment like Frozen Lake 8x8 where rewards are sparse. Since the agent typically receives a reward only when it reaches the goal, raw Q-values can vary significantly, which might lead to unstable or slow convergence. By incorporating a baseline, the agent reduces the variance in its estimates, allowing subtle differences between actions to be more apparent. This adjustment helps prevent negative biases that could occur if the Q-values were initialized too low, ensuring that the agent doesn't prematurely dismiss actions that could be valuable in the long run.

- Changing the value of ε over time or using a policy other than the ε-greedy policy.

  Changing the value of ε over time, or even using a different policy than the standard ε-greedy approach, helps balance the trade-off between exploration and exploitation. In the early stages of training, a higher ε encourages the agent to explore a wider range of actions and states, which is crucial in the unpredictable, slippery conditions of Frozen Lake. As the agent gains experience and the model begins to reflect more accurate estimates, reducing ε gradually allows the agent to shift focus from exploration to exploiting the most rewarding actions it has discovered. Alternative policies like softmax, which select actions based on a probability distribution derived from the Q-values, can offer a more nuanced approach to this balance, often leading to more consistent learning

by smoothing the transition between exploratory and exploitative behavior.

- Changing the number of planning steps n over time.

  Adjusting the number of planning steps n over time can further enhance the learning process by aligning the intensity of simulated updates with the stage of training. Early on, when the agent has not yet encountered the rewarding goal state, the model is largely built on transitions that yield no reward. In this phase, a high number of planning steps may only reinforce uninformative data and consume unnecessary computational resources. However, once the agent experiences the goal reward and the model begins to capture valuable reward signals, increasing the number of planning steps can significantly accelerate the propagation of this reward information throughout the state space. This dynamic adjustment allows the agent to leverage simulated experiences more effectively, ultimately speeding up convergence and improving performance in the stochastic and reward-sparse environment of Frozen Lake.

- **Result of our experiments:**

  In the deterministic version of Frozen Lake, the agent originally never reached the goal because the standard learning setup was unable to break out of the zero-reward loop. With no reward signal propagating through the state space, the Q-values remained low and uniform, meaning that no particular sequence of actions stood out as beneficial. This situation made it almost impossible for the agent to discover the path to the goal, so it effectively remained stuck in a cycle of exploring paths that all led to zero rewards.

  Adding a baseline shifts the Q-values upward. This makes any nonzero reward stand out. Previously, all rewards were zero, so the agent never learned which actions were better. With the baseline, even a single reward makes a difference.

  Switching from an ε-greedy policy to a softmax policy improved exploration. Softmax uses the small differences in Q-values to choose actions, rather than picking randomly. This helped the agent favor actions that led to the goal once a reward was observed.

  Using an adaptive number of planning steps helped target the learning process. Early on, when no rewards were seen, fewer planning steps avoided reinforcing useless transitions. Once a reward was encountered, increasing the planning steps quickly spread the reward information across the state space.

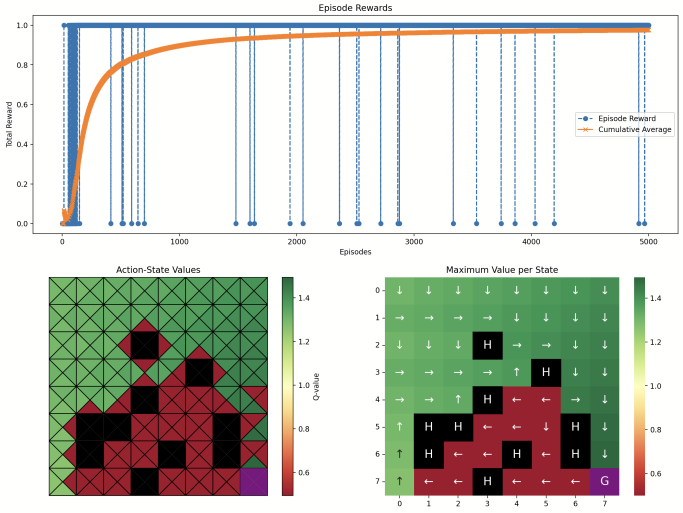

Figure 1: Effect of softmax, adding baseline, and adaptive planning

Reward Shaping Experience

Explain how this method might help us with solving this environment:

- Modifying the reward function.
  - **Shaping Behavior:** Changing the reward function can provide more informative feedback to the agent. For example, adding intermediate rewards helps the agent recognize partially successful steps, which guides it toward the goal.
  - **Encouraging Exploration:** A modified reward function can incentivize the agent to explore specific regions of the state space. This targeted feedback can help overcome the challenge of sparse rewards by highlighting beneficial behaviors.
  - **Accelerating Learning:** More frequent or graded rewards make it easier for the agent to learn which actions are valuable. This often leads to faster convergence because the agent receives clearer signals about what works.
- **Result of our experiments:**

  The episode rewards (blue dots) start off relatively low but quickly rise, and the cumulative average (orange line) stabilizes at a high level. This indicates that the agent learns a more effective policy much faster than if it relied solely on a single reward at the goal. The incremental distance-based reward provides ongoing feedback, making it easier for the agent to distinguish good moves from bad ones.

  In the bottom-left action-state values plot, you can see higher Q-values (lighter colors) along states that move closer to the goal, while states leading to holes remain low-valued. This reflects the shaping reward's influence: the agent is consistently nudged toward the goal rather than wandering aimlessly. In the bottom-right heatmap, the maximum Q-values for each state form a clear gradient guiding the agent toward the goal state ("G"), where it earns the large positive reward of 10.

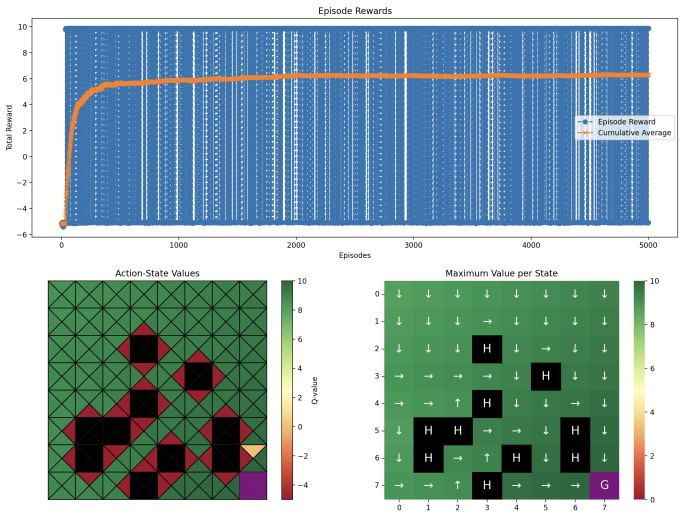

Figure 2: Reward Shaping

In our custom `FrozenLakeEnv` environment, we modify the reward function to provide informative feedback based on the Manhattan distance to the goal.

$$\text{goal\_state} = 64 - 1 = 63.$$

For any given state (represented as a single integer), we compute the Manhattan distance d to the goal state using:

$$d = |\text{current\_col} - \text{goal\_col}| + |\text{current\_row} - \text{goal\_row}|,$$

The reward function is defined piecewise as follows:

- **Goal reached:** If the current observation equals the goal state, then

$$r = 10.0.$$

- **Episode terminated without reaching the goal:** If the episode is done (e.g., the agent falls into a hole), then

$$r = -5.0.$$

- **Otherwise:** The reward is based on the distance to the goal:

$$r = \left(\frac{1.0}{d+1} - 1\right) \times 0.01.$$

This design provides a strong positive reward when the agent reaches the goal and a negative penalty when it fails. For intermediate steps, the reward function encourages the agent to decrease its distance to the goal. As the Manhattan distance d decreases, the term $\frac{1.0}{d+1}$ increases, leading to a less negative (or higher) reward. This shaped reward function helps guide the agent toward the goal by providing incremental feedback even in states that would otherwise yield a zero reward.

Prioritized Sweeping

Explain how this method might help us with solving this environment:

- Altering the planning function to prioritize some state–action pairs over others. (Hint: explain how **Prioritized Sweeping** helps)

  By prioritizing updates, the agent can focus its planning efforts on the state–action pairs that are most likely to influence the overall policy. In particular, when a significant change in the value function is detected—such as when a nonzero reward is finally encountered—the affected state–action pairs are given higher priority. This allows the algorithm to propagate this important information more rapidly through the state space, rather than spending time updating transitions that have little impact on the final outcome.

  Moreover, this selective update mechanism leads to a more efficient use of computational resources. Instead of performing uniform updates across all state–action pairs, the agent concentrates on those areas where changes will most improve its performance. As a result, learning converges faster and the policy becomes more robust, helping the agent navigate the environment more effectively.

- **Result of our experiments:**

  When using Prioritized Sweeping, the learning curve (left plot) rises faster and stabilizes more quickly compared to methods without priority-based planning. This is because updates focus on the state–action pairs most likely to affect the policy, rather than uniformly sweeping through all possibilities. Once the agent discovers a useful transition—such as one that leads closer to the goal—the associated updates receive higher priority, allowing the reward information to propagate more quickly across the state space.

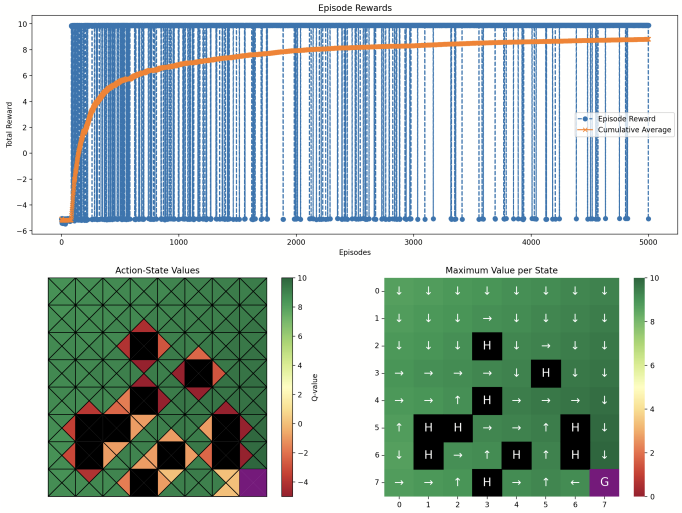

Figure 3: Prioritized Sweeping

Extra

In the slippery version of Frozen Lake, reward shaping gives the agent useful intermediate feedback that helps it learn to move toward the goal. Even with an ε-greedy policy, which introduces randomness in action selection, the shaped reward creates a clearer gradient. This is reflected in the improved running average of episode rewards, even though individual episodes still vary due to stochastic transitions. However, due to the slippery nature of the environment, even good actions can result in unintended moves, so the agent's performance remains inconsistent on a per-episode basis.

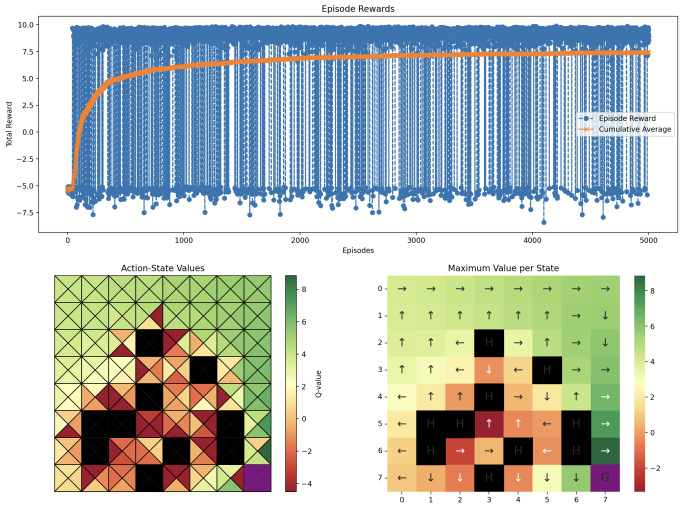

Figure 4: Slippery environment

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?

  Increasing the number of LQR iterations refines the solution within each MPC step. More iterations lead to more accurate control inputs and smoother trajectories by better solving the local linear-quadratic approximation. However, this comes at the cost of additional computation. Conversely, using too few iterations may yield suboptimal or noisier control signals, especially in a nonlinear setting.

- What if we didn't have access to the model dynamics? Could we still use MPC?

  Classical MPC relies on a predictive model of the system dynamics to forecast future states and optimize control actions. Without access to the true dynamics, we cannot directly use standard MPC. However, an alternative is to learn or estimate the dynamics (for example, through system identification or a neural network) and then use the learned model within the MPC framework. This shifts the problem towards model-based reinforcement learning.

- Do `TIMESTEPS` or `N_BATCH` matter here? Explain.

  In the context of MPC, each control action is computed by solving an optimization problem over a finite horizon at every timestep. As a result, parameters like `TIMESTEPS` or `N_BATCH`, which are important in training loops of RL algorithms, have less impact here. The MPC controller recalculates

the optimal trajectory at each timestep, so while simulation length (`TIMESTEPS`) may affect overall performance evaluation, it does not directly influence the control quality in the same way as batch size does in offline learning.

- Why do you think we chose to set the initial state of the environment to the downward position?

  The initial state is chosen to be the downward position to present a challenging control problem. Starting from this state requires the controller to generate sufficient torque to swing the pendulum upward and balance it near the upright position. This choice effectively tests the controller's ability to handle a full swing-up maneuver rather than simply stabilizing an already upright pendulum.

- As time progresses (later iterations), what happens to the actions and rewards? Why

  As time progresses, the actions computed by the MPC become more refined. In the early iterations, the controller makes larger corrective moves to achieve the swing-up. Once the pendulum nears the upright position, the required actions become smaller and more precise, resulting in smoother control. Consequently, rewards improve over time because the controller maintains the pendulum closer to the target state with reduced error, demonstrating increased stability and performance.

Figure 5: Pendulum-v1 MPC

Early on in figure (5), the reward becomes more negative as the controller attempts to swing the pendulum upward from the downward position, causing large and sometimes erratic actions. As time progresses, the controller refines its behavior and the pendulum's motion stabilizes, reflected by the orange line leveling off near zero. Consequently, the reward rises (becoming less negative) and approaches a steady value, indicating that the controller has succeeded in reducing the pendulum's deviation from the upright position.