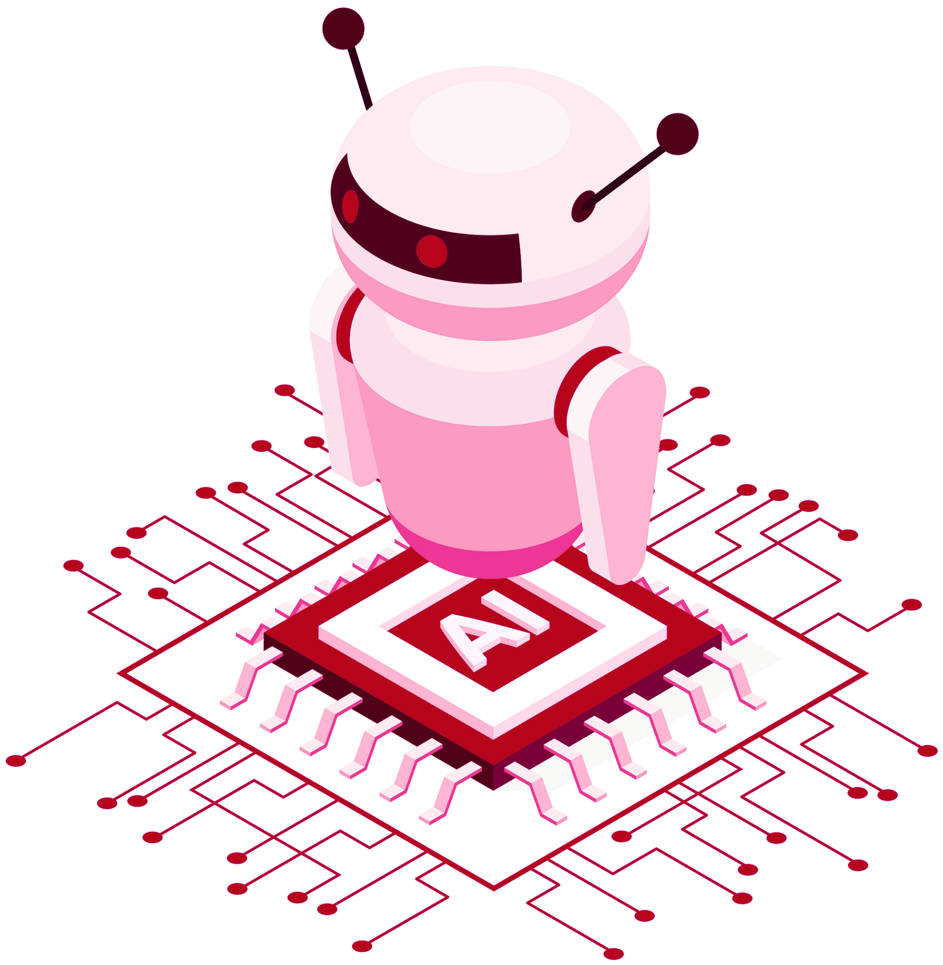

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 5:

Model-Based Reinforcement Learning

By:

Amir Kooshan Fattah Hesari

[401102191]

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? The four main phases of Monte Carlo Tree Search (MCTS) are Selection, Expansion, Simulation, and Backpropagation. The Selection phase starts from the root node of the search tree and recursively traverses down the tree, choosing child nodes based on a selection policy (often balancing exploration and exploitation) until it reaches a node that has been visited but not fully expanded. The conceptual purpose of selection is to efficiently navigate the already explored parts of the search space to identify promising areas for further investigation.

  Once a suitable node is selected, the Expansion phase takes place. In this phase, one or more child nodes representing the possible next actions from the selected node are added to the search tree. Typically, if the selected node represents a non-terminal state, a new child node corresponding to an unvisited action is created. Following expansion, the Simulation phase (also known as rollout) begins from the newly added child node. In this phase, the game or process is played out until a terminal state or a predefined limit is reached, with actions chosen according to a rollout policy (often a simple or random policy for efficiency). The outcome of this simulation provides an estimate of the value of the expanded node. Finally, the Backpropagation phase updates the information stored in the nodes along the path traversed from the newly simulated node back to the root of the tree. This typically involves updating statistics such as the number of visits and the win/reward rate for each node, allowing the search to learn and refine its understanding of the value of different actions and states over multiple iterations.

- How does MCTS balance exploration and exploitation in its node selection strategy (i.e., how does the UCB formula address this balance)? The Upper Confidence Bound (UCB) formula, commonly used in the Selection phase of MCTS, explicitly balances exploration and exploitation through its two main components. The formula typically used is:

  $$UCB_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln(N_i)}{n_i}}$$

  where:

  - UCB_i is the UCB value for the i-th child node.
  - $\frac{w_i}{n_i}$ represents the exploitation component, which is the current win rate (or average reward) of the i-th child node based on the simulations performed so far. This term encourages the selection of nodes that have led to favorable outcomes in the past.
  - $c\sqrt{\frac{\ln(N_i)}{n_i}}$ represents the exploration component. Here, c is a positive exploration constant, N_i is the number of times the parent node has been visited, and n_i is the number of times the i-th child node has been visited. The natural logarithm of the parent's visit count ($\ln(N_i)$) ensures that as the parent is visited more often, the exploration bonus decreases gradually.

The term is inversely proportional to the number of times the child node has been visited (n_i), meaning that nodes that have been visited less often will have a larger exploration bonus. This encourages the algorithm to explore less visited parts of the search tree, potentially discovering better moves that haven't been thoroughly evaluated yet. The constant c controls the strength of the exploration; a higher c favors more exploration, while a lower c favors more exploitation.

By combining these two terms, the UCB formula guides the Selection phase to choose child nodes that either have a high win rate (exploitation) or haven't been visited enough times to confidently assess their potential (exploration).

1.2.2 Tree Policy and Rollouts

- Why do we run multiple simulations from each node rather than a single simulation? We run multiple simulations from each node in Monte Carlo Tree Search (MCTS) to obtain a more reliable estimate of the node's value. A single simulation, especially if the rollout policy is simple or random, can be heavily influenced by chance and might not accurately reflect the true potential of the state represented by that node.

  By performing multiple rollouts from the same node, we are essentially sampling the possible outcomes reachable from that state. The average outcome across these simulations provides a much more robust and statistically sound estimate of the node's value. This aggregation of results helps to reduce the variance inherent in a single random simulation and allows the MCTS algorithm to make more informed decisions during the Selection and Backpropagation phases, ultimately leading to better overall performance.

- What role do random rollouts (or simulated playouts) play in estimating the value of a position? Random rollouts, or simulated playouts, play a crucial role in Monte Carlo Tree Search (MCTS) by providing a computationally inexpensive way to estimate the value of a position. Starting from a leaf node in the search tree (or a newly expanded node), a rollout involves simulating the game or process until a terminal state is reached or a predefined limit is met. During the rollout, actions are typically chosen randomly or according to a simple, fast policy.

  The outcome of each random rollout (e.g., a win, loss, or a numerical score) serves as a sample of the potential value of the position from which the rollout began. By performing many such random rollouts from a given node and averaging their results, MCTS can approximate the expected value of that position. This estimated value is then backpropagated up the search tree, updating the statistics of the nodes along the path. These statistics, such as the number of wins and visits, are used in the Selection phase to guide the search towards more promising areas of the search space. Thus, random rollouts provide a fundamental mechanism for evaluating the potential of unexplored or less explored states, enabling MCTS to make informed decisions without relying on complex, domain-specific evaluation functions.

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?
  In Neural MCTS, as exemplified by AlphaGo-style approaches, policy networks and value networks trained through deep learning are deeply integrated into the traditional MCTS procedure to significantly enhance its performance.

  The policy network plays a crucial role in both the Selection and Expansion phases. During Selection, the policy network provides prior probabilities for each possible action from a given state. This prior probability is often incorporated into the UCB formula, biasing the search towards actions that the policy network deems more promising. This helps to guide the exploration towards high-potential moves from the very beginning. In the Expansion phase, when a leaf node is reached, the policy network can suggest which actions are most likely to be good candidates for expansion, allowing the tree to grow more strategically. Furthermore, in some variations, the policy network might also inform the rollout policy, making the simulated playouts more intelligent and relevant.

The value network is primarily used in place of or to augment the Simulation phase. Instead of performing a random or simple rollout until the end of the game, the value network directly estimates the value of the state reached after the Expansion phase. This value represents the predicted probability of winning (or the expected outcome) from that state. This direct evaluation by the value network is much more efficient and often more accurate than relying on the outcome of a full random rollout. The value predicted by the value network is then used in the Backpropagation phase to update the win/loss statistics of the nodes along the path from the expanded node back to the root. This allows the MCTS to learn from the neural network's understanding of the game's value function, leading to a much stronger search performance compared to traditional MCTS.

- 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?
  In the Expansion phase of Neural MCTS, the policy network's output, which consists of prior probabilities for each possible action from the current state, serves as a crucial guide for determining which moves to explore further. Instead of blindly expanding all possible next states, the prior probabilities provided by the policy network offer a heuristic indicating the likelihood of each move being good or promising.

  Specifically, these prior probabilities influence which moves get explored by:

  1. **Prioritizing Expansion:** MCTS often prioritizes the expansion of nodes corresponding to actions with higher prior probabilities assigned by the policy network. This means that the algorithm is more likely to create new branches in the search tree for moves that the policy network deems more promising based on its training.
  2. **Focusing Search:** By focusing the expansion on moves with higher prior probabilities, the search tree is more likely to grow in directions that are considered strategically relevant. This can significantly improve the efficiency of the search by concentrating computational resources on exploring potentially better moves earlier in the process.
  3. **Informing UCB:** The prior probabilities are often incorporated into the UCB formula used during the Selection phase. This means that even for newly expanded nodes with few or no simulations, the initial UCB value will be influenced by the policy network's prior probability, making it more likely that these initially promising moves will be selected for further simulation.

  In essence, the policy network's prior probabilities act as an informed prior, guiding the Expansion phase to strategically grow the search tree towards moves that are deemed more likely to be beneficial, based on the knowledge learned by the neural network. This allows Neural MCTS to explore the search space more effectively compared to traditional MCTS, which might expand nodes more uniformly or based solely on visit counts.

1.2.4 Backpropagation and Node Statistics

- During backpropagation, how do we update node visit counts and value estimates?
During the Backpropagation phase of Monte Carlo Tree Search (MCTS), the results of the simulation (or the value estimate from a value network in Neural MCTS) are propagated back up the search tree from the newly expanded node to the root node. This involves updating the statistics of each node along the path traversed.

  Here's how node visit counts and value estimates are typically updated:

  1. **Visit Counts:** For every node along the path from the expanded node back to the root, the visit count is incremented by one. This count keeps track of how many times each node has been part of a simulation. It reflects how much the algorithm has explored the subtree rooted at that node.
  2. **Value Estimates:** The update of the value estimate depends on the outcome of the simulation or the prediction from a value network.
     - **For games with win/loss outcomes:** Each node usually stores a win count (or a total reward). If the simulation from the expanded node results in a win for the player whose turn it was at that node, the win count (or reward) of that node and all its ancestors along the path is incremented. The value estimate of a node is often calculated as the ratio of wins to visits (win rate).
     - **For problems with numerical rewards:** If the simulations yield numerical rewards, each node typically stores the sum of the rewards obtained from all simulations that have passed through it. The value estimate of a node is then the average reward, calculated by dividing the sum of rewards by the number of visits:

$$\text{Value Estimate} = \frac{\text{Sum of Rewards}}{\text{Number of Visits}}$$

  The backpropagation process ensures that the information gained from exploring a particular path in the search tree is reflected in the statistics of all the nodes along that path, from the point of expansion back to the root. This allows the MCTS algorithm to gradually learn the value of different actions and states as it performs more simulations. Nodes that lead to better outcomes (higher win rates or rewards) will tend to have higher value estimates and will be more likely to be selected for further exploration in subsequent iterations of the MCTS algorithm.

- Why is it important to aggregate results carefully (e.g., averaging or summing outcomes) when multiple simulations pass through the same node?
It is crucial to aggregate results carefully (e.g., averaging or summing outcomes) when multiple simulations pass through the same node in Monte Carlo Tree Search (MCTS) because this aggregation forms the basis for estimating the value of that state. Here's why this careful aggregation is important:

  1. **Obtaining a Reliable Value Estimate:** Each simulation provides a single sample of the potential outcome from a given node. However, a single outcome can be noisy or heavily influenced by the randomness of the rollout policy. By aggregating the results of multiple simulations, we obtain a more robust and statistically reliable estimate of the expected value of that node. For example, averaging win/loss outcomes gives us an estimate of the win probability from that state.

2. **Reducing Variance:** The outcomes of individual simulations can vary significantly. By averaging or summing over many simulations, we reduce the impact of these random fluctuations, providing a more stable and accurate representation of the underlying value of the state. This reduction in variance is essential for making informed decisions during the Selection phase of MCTS.

3. **Informing Node Selection:** The aggregated value estimates are used to guide the exploration-exploitation balance in the Selection phase. For instance, in the UCB formula, the average reward or win rate of a node is a key component. If the aggregation is not done correctly, the value estimates will be misleading, potentially causing the algorithm to explore suboptimal branches or prematurely exploit seemingly good but ultimately less promising moves.

4. **Convergence Towards True Value:** As more simulations pass through a node, the aggregated value should, according to the Law of Large Numbers, converge towards the true expected value of that state. Careful aggregation ensures that this convergence is accurate and reflects the actual potential of the position.

In summary, careful aggregation of simulation results is fundamental to the accuracy and reliability of the value estimates in MCTS. These estimates are the cornerstone of the algorithm's decision-making process, guiding both exploration and exploitation to find optimal strategies or solutions.

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 (often denoted c_{puct} or c) in the UCB formula,

$$UCB_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln(N_i)}{n_i}},$$

plays a critical role in balancing the trade-off between exploitation and exploration during the node selection phase of Monte Carlo Tree Search.

A **higher value** of the exploration constant c increases the weight of the exploration term (the second term in the formula). This encourages the MCTS algorithm to select nodes that have been visited less frequently, even if their current win rate ($\frac{w_i}{n_i}$) is lower than that of more frequently visited nodes. This promotes a broader search of the state space, increasing the chance of discovering potentially better moves that haven't been thoroughly explored yet. Conversely, a **lower value** of c reduces the influence of the exploration term, making the algorithm more likely to choose nodes with a higher current win rate, thus favoring exploitation of the knowledge already gained.

Tuning the exploration constant is crucial for optimal performance and often depends on the specific problem domain. A common approach is to perform a parameter sweep or grid search, trying different values of c and evaluating the performance of the MCTS agent (e.g., by playing against a fixed opponent or a baseline). One might start with a range of values (e.g., from 0.1 to 10) and systematically test each. More sophisticated methods like Bayesian optimization can also be employed to more efficiently find a good value for c. The optimal value often represents a balance: too low, and the search might get stuck in a local optimum; too high, and the search might spend too much time exploring unpromising branches. In AlphaGo and its successors, the exploration constant (often within the Puct formula, which is a variant of UCB) was carefully tuned through extensive self-play and evaluation to achieve the best balance between exploration and exploitation for the game of Go.

- In what ways can the “temperature” parameter (if used) shape the final move selection, and why might you lower the temperature as training progresses?
  The "temperature" parameter, often used in the final move selection stage of MCTS, shapes the probability distribution over the available moves based on their visit counts. A **higher temperature** makes the probabilities of selecting different moves more uniform, even if their visit counts are significantly different. This encourages more exploration in the final move selection, as less visited but potentially promising moves have a higher chance of being chosen. Conversely, a **lower temperature** makes the probability distribution sharper, concentrating the probability mass on the move with the highest visit count. As the temperature approaches zero, the move selection becomes deterministic, always picking the move that MCTS has explored the most and deems the best.

  Lowering the temperature as training progresses in systems like AlphaGo serves several important purposes. In the early stages of training, the agent's knowledge (represented by the policy and value networks) is limited. A higher temperature encourages more exploration during self-play games, allowing the agent to discover a wider variety of states and moves. This diverse experience is crucial for learning a robust and general policy. As training progresses, the agent becomes more confident in its evaluations. Lowering the temperature gradually shifts the focus from exploration to exploitation in the self-play games. By selecting moves more deterministically (based on the higher visit counts resulting from the improved policy), the agent reinforces the strategies it has learned and generates higher-quality training data for further improvement. This transition from exploration to exploitation is essential for the agent to converge to a strong playing policy.

1.2.6 Comparisons to Other Methods

- How does MCTS differ from classical minimax search or alpha-beta pruning in handling deep or complex game trees? Monte Carlo Tree Search (MCTS) and classical minimax search (with alpha-beta pruning) differ significantly in their approach to handling deep or complex game trees.

  Classical minimax search with alpha-beta pruning aims to explore the game tree up to a certain depth, evaluating terminal states or using a heuristic evaluation function for non-terminal states at the search horizon. It operates by exhaustively considering all possible moves and their consequences within the search depth, pruning branches that are guaranteed to be suboptimal. This method suffers from the "state-space explosion" problem in deep or complex games with high branching factors, as the number of nodes to explore grows exponentially with the search depth. While alpha-beta pruning helps to mitigate this by reducing the number of nodes evaluated, it still struggles with the sheer size of the game tree in games like Go or complex real-time strategy games. Furthermore, it heavily relies on a well-crafted static evaluation function to estimate the value of non-terminal states, which can be challenging to design effectively for complex games.

- What unique advantages does MCTS provide when the state space is extremely large or when an accurate heuristic evaluation function is not readily available?
  Monte Carlo Tree Search (MCTS) offers unique advantages when dealing with extremely large state spaces or when an accurate heuristic evaluation function is not readily available due to its fundamental differences from traditional search algorithms like minimax.

  When the **state space is extremely large**, MCTS excels because it employs a selective exploration strategy. Unlike algorithms that attempt to explore the entire game tree or rely on a fixed depth limit, MCTS iteratively builds a partial tree by focusing its simulations on the most promising regions of the search space. This is achieved through the UCB (or similar) selection policy, which balances exploration of less visited nodes with exploitation of nodes that have shown good results so far. This adaptive and focused exploration allows MCTS to find good moves even in vast state spaces where a full or deep search is computationally infeasible. Furthermore, MCTS is an anytime algorithm, meaning it can be stopped at any point and will return the best move found so far, with the quality of the move typically improving with more computation time.

When an **accurate heuristic evaluation function is not readily available**, MCTS provides a significant advantage by estimating the value of a state through simulation (rollout). Instead of relying on a handcrafted heuristic to evaluate non-terminal states, MCTS performs random playouts from a given state until a terminal state is reached, and the outcome of these simulations (e.g., win or loss) is used to update the value of the nodes in the search tree. This approach is particularly powerful in domains where defining a good heuristic is challenging or requires extensive domain expertise. In more advanced implementations like AlphaGo, MCTS is combined with deep neural networks that learn a policy and a value function through self-play, further reducing the reliance on manually designed heuristics. This allows MCTS to be applied effectively to complex games and problems where traditional evaluation functions have struggled.

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 **??**.

2.1.1 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 **??** 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?
  After implementing the basic Dyna-Q algorithm, experiments would likely reveal the following effects of increasing the number of planning steps on the overall learning process:

  Increasing the number of planning steps in Dyna-Q allows the agent to more extensively utilize its learned model of the environment. Each planning step involves randomly sampling a previously experienced state-action pair from the model and updating the Q-values as if that transition had occurred in reality. Consequently, a higher number of planning steps per real interaction can lead to **faster learning**. The agent can propagate the rewards and learned values more quickly through the state space via these simulated experiences, potentially reaching an optimal or near-optimal policy in fewer actual interactions with the environment. This is particularly beneficial in scenarios where real-world interactions are costly or time-consuming. Furthermore, in complex environments with delayed rewards or intricate state transitions, more planning can help the agent to better anticipate the long-term consequences of its actions, leading to the development of a more sophisticated and effective policy.

  However, increasing the number of planning steps also comes with potential drawbacks. The most immediate is the **increased computational cost** per step of interaction with the real environment. Each planning step requires computational resources, and a very high number of planning steps might make the learning process slower in terms of wall-clock time, even if it requires fewer real interactions. Additionally, the effectiveness of planning is directly tied to the **accuracy of the learned model**. If the model is flawed or incomplete, performing a large number of planning steps based on this inaccurate model could lead the agent to converge to a suboptimal policy. There is also a risk of **overfitting to the model**, where the agent becomes too reliant on the simulated experiences and fails to adequately explore the real environment or adapt to changes not captured by the model. Ultimately, the optimal number of planning steps often represents a trade-off between the benefits of leveraging the learned model for faster learning and the costs associated with computation time, model accuracy, and the need for continued real-world exploration.

- 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)
  If we trained a basic Dyna-Q algorithm, which assumes a deterministic environment in its model learning and planning phases, on a slippery version of the environment (where transitions are stochastic), several issues would likely arise:

  1. **Inaccurate Model Learning:** The core of Dyna-Q involves learning a model of the environment, typically represented as $Model(s, a) \rightarrow (s', r)$. If the environment is slippery, meaning that taking action a in state s can lead to different next states s' with certain probabilities, a deterministic algorithm would struggle to learn an accurate model. It might learn a model

that predicts the most frequent next state or an average outcome, but this would not capture the true stochastic nature of the environment.

2. **Suboptimal Planning:** The planning phase of Dyna-Q relies on the learned model to simulate experiences and update Q-values. If the model is deterministic but the environment is not, the simulated experiences will not accurately reflect the actual possibilities. The agent might plan based on transitions that are not guaranteed to occur, leading to a suboptimal policy. For instance, it might assume it can reliably reach a certain state by taking a specific sequence of actions, only to find that the slipperiness of the environment often leads to unexpected outcomes.

3. **Inefficient Exploration:** A deterministic policy derived from a deterministic model might lead to inefficient exploration. In a slippery environment, sometimes seemingly suboptimal actions might need to be taken to navigate the stochasticity or to reach certain states. A deterministic agent might not discover these necessary strategies if its model doesn't account for the probabilistic transitions.

In conclusion, training a deterministic Dyna-Q algorithm on a slippery environment without modifications to handle stochasticity would likely result in a poor performance. The learned model would be inaccurate, leading to flawed planning and an inability of the agent to develop an optimal policy that accounts for the probabilistic nature of the environment. To effectively learn in a slippery environment, the algorithm would need to be adapted to handle stochastic transitions, for example, by learning a probabilistic model or by using reinforcement learning algorithms inherently designed for stochastic environments.

- 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.

  - **Small n (close to 0):** When the number of planning steps is very small or zero, Dyna-Q behaves similarly to a purely model-free reinforcement learning algorithm like Q-learning. The agent primarily learns from direct interactions with the environment. It will take a certain number of episodes (N_1) to randomly stumble upon the goal and receive the reward. Learning will be slow as Q-values are mainly updated based on actual experiences.

  - **Intermediate n:** With a moderate number of planning steps, the agent starts to leverage its learned model of the environment. After each real interaction, it performs n simulated experiences using the model. These simulated experiences allow the reward signal from reaching the goal to propagate backwards through the state space more rapidly than with just direct experience. Consequently, the agent is likely to find the goal in fewer episodes (N_1 will be smaller) compared to the case with small n. The planning helps in exploring potential paths to the goal without needing to physically traverse them in the environment.

  - **Large n:** If the number of planning steps is large, the agent spends a significant amount of computation time planning based on its current model after each real interaction. If the model is reasonably accurate, this can dramatically speed up the initial learning process, resulting in a significantly smaller N_1. The agent can effectively "think" through many possible action sequences using its model and potentially identify a path to the goal much faster. However, there might be diminishing returns beyond a certain point, and if the model is initially inaccurate, excessive planning could lead the agent astray.

Once the agent has reached the goal for the first time, the reward information has been received and propagated to some extent through both real experience and the planning process. The learned model and Q-values will now contain information about a path to the goal.

- **Small n:** For the second time, the agent will still primarily rely on direct experience. However, having reached the goal once, the Q-values for the states leading to it should have been updated. Therefore, it will likely take fewer episodes ($N_2 < N_1$) to reach the goal again compared to the very first time, but the number will still be relatively high as the reinforcement of the goal path is slow.
- **Intermediate n:** With a moderate number of planning steps after the first goal achievement, the reward information will be propagated much more effectively through the simulated experiences. The agent's model and Q-values will be more strongly informed about the path to the goal. As a result, the agent should be able to reach the goal for the second time much faster (N_2 will be significantly smaller than N_1 and also smaller than the case with small n). The planning allows the agent to quickly reinforce the learned path and potentially discover more efficient routes.
- **Large n:** A large number of planning steps after the first goal achievement will likely lead to a very rapid re-discovery of the goal (N_2 will be very small, potentially even 1 if the policy has become deterministic and directly leads to the goal). The agent will extensively simulate experiences based on its now more informed model, quickly solidifying a policy that leads to the goal. Similar to the case of N_1, excessively large n might not provide significant additional benefit after the goal has been reached once and a good model has been learned.

In conclusion, increasing the number of planning steps n generally leads to a decrease in both N_1 and N_2. The more the agent plans, the faster it can learn and reinforce a policy that leads to the goal, both initially and on subsequent attempts. However, the optimal value of n involves a trade-off with computational cost and the accuracy of the learned model.

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)

Here's how each of the mentioned methods might help in solving the Frozen Lake environment using Dyna-Q:

- **Adding a baseline to the Q-values:** Adding a baseline to the Q-values can influence the exploration behavior. A positive baseline might encourage the agent to explore more by making the initial estimates of all actions more optimistic. In Frozen Lake, where the goal provides a positive reward, a positive baseline could motivate the agent to seek out this reward rather than getting stuck in local optima with zero or negative expected returns. The baseline can help in overcoming initial timidity to try new paths, especially in a slippery environment where the immediate outcome of an action might be unpredictable. The optimal value of the baseline would likely need to be tuned through experimentation.

- **Changing the value of ε over time or using a policy other than the ε-greedy policy:** The ε-greedy policy balances exploration and exploitation.
  - **Changing ε over time (e.g., annealing):** Starting with a high ε (more exploration) and gradually decreasing it over time (more exploitation) is often beneficial. In Frozen Lake, especially the slippery version, a high initial exploration rate is crucial to discover the goal and understand the stochastic transitions. As the agent learns more about the environment, reducing ε allows it to exploit the learned policy, hopefully leading to the goal more consistently.
  - **Using a policy other than ε-greedy (e.g., softmax/Boltzmann exploration):** Policies like softmax choose actions based on a probability distribution derived from the Q-values. This can provide a more nuanced exploration strategy compared to the purely random exploration of ε-greedy. Actions with higher Q-values are more likely to be chosen, but less optimal actions still have a non-zero probability, allowing for continued exploration of potentially better paths. In Frozen Lake, this might help in navigating the slippery conditions by occasionally trying less obvious moves that could lead to the goal.
- **Changing the number of planning steps n over time:** The number of planning steps in Dyna-Q determines how much the agent simulates experiences using its learned model.
  - **Increasing n over time:** Initially, the learned model might be inaccurate due to limited experience in Frozen Lake. Starting with a smaller n could prevent the agent from over-relying on a flawed model. As the agent explores more and the model becomes more accurate, gradually increasing n can allow for more efficient learning by propagating the reward from the goal (once found) more rapidly through simulated experiences, thus speeding up policy improvement.
  - **Decreasing n over time:** One might also consider starting with a higher n to quickly propagate the reward signal once the goal is reached for the first time, and then decreasing n to focus more on real-time interactions and adapting to the stochasticity of the slippery environment. The optimal schedule for changing n would likely depend on the specific characteristics of the Frozen Lake environment (e.g., its size and slipperiness).
- **Modifying the reward function:** The reward function shapes the agent's learning goals. In the standard Frozen Lake, the agent gets a reward only at the goal.
  - **Reward shaping:** We could introduce intermediate rewards to guide the agent. For example, a small positive reward for moving closer to the goal (perhaps based on Manhattan distance) or a small negative reward for moving towards a hole. This could provide more frequent feedback and accelerate learning, especially in the slippery environment where reaching the terminal states might be infrequent initially.
  - **Penalties:** Increasing the penalty for falling into a hole might discourage the agent from taking risky actions and encourage it to find safer, albeit possibly longer, paths to the goal.
- **Altering the planning function to prioritize some state–action pairs over others (Prioritized Sweeping):** The basic Dyna-Q updates state-action pairs for planning uniformly at random. Prioritized Sweeping is an enhancement that focuses planning on state-action pairs whose updates are likely to have a significant impact on the Q-values of other states.

- **Mechanism:** Prioritized Sweeping uses a priority queue to keep track of state-action pairs. When a transition results in a change in the Q-value of a state s', we look at all the states s that could lead to s' and prioritize the planning for the state-action pair (s, a) that leads to s'. The priority is often based on the magnitude of the change in the Q-value.
- **How it helps in Frozen Lake:** In Frozen Lake, reaching the goal state yields a significant reward. When this happens, the Q-values of the preceding states should be updated to reflect the possibility of reaching the goal. Prioritized Sweeping ensures that planning efforts are concentrated on these relevant predecessors, allowing the reward signal to propagate backwards from the goal more efficiently. This is particularly useful in navigating the environment, as it helps the agent quickly learn the sequence of actions required to reach the goal once it has been discovered. Even in the slippery version, prioritizing updates based on significant changes can help the agent adapt to the probabilistic outcomes more effectively by focusing on reinforcing successful (or avoiding unsuccessful) transitions.

By strategically applying these methods, we can potentially improve the efficiency and effectiveness of the Dyna-Q algorithm in solving the Frozen Lake environment, especially when dealing with the challenges posed by the slippery surface. The optimal configuration of these modifications would likely require empirical evaluation.

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 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?
  The number of LQR iterations affects the accuracy of the terminal cost and constraints used in MPC. More iterations lead to a more accurate LQR solution, which can improve the stability and performance of the MPC controller. Insufficient iterations might result in suboptimal MPC behavior.
- What if we didn't have access to the model dynamics? Could we still use MPC?
  Yes, we could still use MPC by employing data-driven techniques to learn a model from input-output data or by using model-free approaches like reinforcement learning within an MPC framework.
- Do `TIMESTEPS` or `N_BATCH` matter here? Explain.
  Yes, `TIMESTEPS` (which likely refers to the prediction horizon length) matters significantly in MPC. It determines how far into the future the controller predicts the system's behavior to optimize control actions.

  No, `N_BATCH` typically does not matter in the online operation of MPC. It is a parameter used in training machine learning models and is not a direct part of the core MPC algorithm.
- Why do you think we chose to set the initial state of the environment to the downward position?
  Starting the pendulum downward (π radians) is a challenging test for MPC because it's an unstable equilibrium. The controller must stabilize it and swing it upright. This demonstrates MPC's ability to control unstable systems, reach a target state, and operate in real-time dynamic systems.

- As time progresses (later iterations), what happens to the actions and rewards? Why ?
  In the Gymnasium Pendulum environment with MPC, as time progresses:

  Actions (Torque): Initially, the MPC controller applies torques to swing the pendulum to the upright position. Later, the actions will adjust to maintain the pendulum in the upright position, typically becoming smaller if the goal is achieved and the system is stable.
  Rewards (Cost): The cost, which penalizes deviation from the upright and control effort, should decrease (become less negative or closer to zero) as the pendulum is successfully controlled and stabilized at the desired state.
  Why: MPC optimizes control actions at each step to minimize a cost function over a prediction horizon. As the pendulum approaches the desired state, the optimal actions and the resulting cost reflect this convergence.