Distinction Task 12: Reinforcement Learning

School of Information Technology, Deakin University

Overview of Learning (Module Summary)

GitHub Link
ChatGPT

Task 1a: Code a soluton to solve the MDP using value-iteraton algorithm

Problem Statement

If we view the game of Tic-Tac-Toe as a Markov Decision Process (MDP), how would we use the value-iteration algorithm to find the optimal policy for the game?

Assumptions

We do not have access to transition probabilities, so we will assume that the transition probabilities are 1.0 for all actions. We will also use a discount factor is 0.9. A graph will be used to represent the each board state. The nodes in the graph will be the board states and the edges will be the actions.

Solution

In this task, I employed the Value Iteration algorithm to derive the optimal policy for playing Tic-Tac-Toe, conceptualizing it as a Markov Decision Process (MDP). I designed a ValueIteration class that inherits from a generic Algorithm class, responsible for initializing essential elements like the value function, policy, and a graph to encapsulate all conceivable game states. This design strategy offered a modular and organized framework to address the problem.

State Space and Initialization

A considerable amount of time was invested in constructing the state space within the graph. To achieve this, I utilized a breadth-first search (BFS) strategy in the initialize method. This method explored all potential states and transitions, including terminal states, to ensure a comprehensive representation of the game's dynamics. The BFS technique was instrumental in systematically traversing the state space, a critical factor for the algorithm's precision.

Iterative Value Updates and Convergence
I implemented the valueIteration method to iteratively refine the value function and policy. Although time constraints prevented me from running episodes to validate the optimal policy, I noticed that the algorithm converged after just six iterations. This quick convergence was an encouraging sign of the algorithm's effectiveness.

Action Selection
The bestMove method leverages the converged policy to identify the most advantageous move for a given board state, with the objective of maximizing the expected reward.

Key Take aways

In summary, my design choices aimed to create a robust and accurate model of Tic-Tac-Toe as an MDP. While I couldn't fully test the optimal policy due to time limitations, the algorithm's rapid convergence and the meticulous construction of the state space suggest a promising approach to solving the game. The solution to value iteration is not overly complex but took much time to implement. The most time-consuming part of the project was deciding on how to creating the state space in the graph. Value iteration was only suitable for small tic-tac-toe boards, as the states explodes exponentially as the board size increases.

Task 1b: Devise and code a soluton to tic-tac-toe using Q-Learning algorithm.

Problem Statement

Once again use the game of Tic-Tac-Toe as a Markov Decision Process (MDP), how would we use the Q-Learning algorithm to find the optimal policy for the game?

Assumptions

This time I did not use a graph to represent the board states. Instead I used a dictionary to store the Q-values for each board state. The keys in the dictionary are the board states and the values are the Q-values for each action. The Q-values are updated using the Q-Learning algorithm.

Solution

In my project, I implemented the Q-Learning algorithm to solve the Tic-Tac-Toe game, treating it as a Markov Decision Process (MDP). The `QLearning` class inherits from an abstract `Algorithm` class and is responsible for initializing various parameters like the exploration rate, learning rate, and discount factor. This design choice allows for a modular and structured approach to solving the problem.

State Space and Training
Creating the state space was a significant part of the project, similar to the Value Iteration approach. I trained the model using the `train` method, which calls the `qLearning` method for a specified number of episodes. To speed up testing times, I kept the number of training iterations low. After training, I saved the policies to disk using Python's `pickle` library, allowing me to reuse them later without retraining. This was particularly useful as the policy was also reused in the Monte Carlo Tree Search (MCTS) algorithm.

Iterative Coding Style
I adopted an iterative coding style, where I would code one method and then immediately test it. This approach was invaluable for understanding the code and debugging issues along the way.

Action Selection
The `chooseAction` method uses an epsilon-greedy strategy to balance exploration and exploitation. However, due to time constraints, I didn't fine-tune the exploration rate, learning rate, or discount factor. These parameters were set to default values, as experimenting with them would have been hit-or-miss given the limited time.

Randomization in Best Move
In the `bestMove` method, I introduced randomization when two Q-values were the same. This adds an element of unpredictability and exploration, which can be beneficial in certain game scenarios.

Policy Update and Convergence
The `updatePolicy` method updates the policy based on the learned Q-values. Although I didn't implement a method to check for convergence explicitly, the training process seemed to stabilize the Q-values, indicating some level of convergence.

Model Management
I used a `ModelManager` class to manage the loading and saving of trained models. This made it easier to reuse policies and facilitated the integration of the Q-Learning algorithm into other parts of the project, like the MCTS algorithm.

Key Take aways

Overall, the Q-Learning implementation was as complex as the Value Iteration approach but offered the flexibility of being more easily integrated with other algorithms. The time constraints did limit the extent to which I could fine-tune the model, but the iterative coding style and modular design made the development process more manageable.

Task 2: Code Monte-Carlo Tree Search algorithm and integrate it with Q-Learning algorithm

Problem Statement

Solutions to larger tic-tac-toe boards are not feasible using the value-iteration algorithm. How would we use the Monte-Carlo Tree Search algorithm to find the optimal policy for the game?

Assumptions

We can use the policy found by the Q-Learning algorithm to guide the Monte-Carlo Tree Search algorithm. The Monte-Carlo Tree Search algorithm will be used to find the optimal policy for larger tic-tac-toe boards.

Solution

In this task, I've designed a Monte Carlo Tree Search (MCTS) algorithm integrated with Q-Learning to play Tic-Tac-Toe. The architecture consists of several classes, each serving a distinct purpose.

Node and Tree Structure

The TreeNode class serves as the building block for the MCTS tree. Each node keeps track of its state, parent, visit count, value, and possible actions. The Tree class, on the other hand, manages these nodes, providing methods to set the root, add children, and retrieve various node attributes.

MCTS Algorithm

The MonteCarloTreeSearch class inherits from a generic Algorithm class and is the core of the MCTS implementation. It uses a tree to represent the state space and employs a Q-Learning policy for simulations. The algorithm consists of four main steps: selection, expansion, simulation, and backpropagation. The bestMove method uses the Q-Learning policy to determine the best move for the current board state.

Integration and Policy

The Integration class serves as the bridge between MCTS and Q-Learning. It uses Q-Learning to select actions but falls back to MCTS for states where Q-Learning is uncertain. The action_selection method encapsulates this logic. The class also provides placeholders for future improvements, such as policy updates and efficiency guides.

Challenges and Limitations

1. Player Handling in Simulation: I encountered issues with the simulate method, particularly in managing the current player. Each time a simulation would end, I had to reset the player back to 'X'.
2. Unfinished Functions: Due to time constraints, I couldn't complete the last two functions in the Integration class, which are intended for policy updates and efficiency guides.
3. Policy Optimality: While the Q-Learning policy is used, its optimality is not guaranteed. I can beat the AI most times on smaller boards, indicating room for improvement.
4. Tuning: The exploration constant and other hyperparameters were kept constant throughout the project, as I didn't have time to fine-tune them.

Key Take aways

In summary, the project aims to create a robust Tic-Tac-Toe AI using MCTS and Q-Learning. Although it's a work in progress, the architecture is modular and extensible, providing a solid foundation for future improvements. With more time I would have researched more to gain a deeper understanding of the MCTS algorithm. I would have also liked to have implemented the policy updates and efficiency guides.