CS660: Sequential Decision Making – HW2: Frozen Lake

Michael Probst

mppr222@uky.edu

Abstract

This assignment implements four techniques for both model-based and model-free reinforcement learning to solve the frozen lake environment provided by the Open AI Gym. These techniques include: Dyna-Q, Q-Learning, SARSA, and a Neural Network. The performance for each of these implementations was determined by intermittently running tests of the agent’s policy during training to calculate the average reward. Over 50,000 training episodes, the average rewards converged to approximately: 0.23, 0.75, 0.69, and 0.33, respectively.

Background

Each agent’s implementation utilized similar strategies which are described here for conciseness.

ε-Greedy

The epsilon greedy method was used by each agent to determine whether it explore (pick a random action) or exploit (pick current best action). The value of epsilon decays by some factor each time the agent reaches the goal state in training. Doing so guaranteed that the agent would observe the goal state a specific number of times before it would primarily exploit its best actions. This was done because in this particular environment, the goal state needs to found once in order for the rewards of the other states to be non-zero because all states, excluding the goal state, yield a reward value of 0. If the agent exploits its best action before it has found any reward, then its best action still has a predicted reward of 0, which could result in the agent consistently taking an action that leads to a terminal state that is not the goal. Also, the goal state will have the greatest influence on updating the policy, so guaranteeing it has been reached a certain number of times will have a greater impact on the policy than if epsilon were decayed after every action. The rate at which epsilon decays differs for each agent because each agent required a different number of successes to ensure a good policy.

Q-Table

In tabular model-based reinforcement learning, a q table is used to describe the policy that the agent uses to determine which action it should take. The Q-Table is stored in a data structure that describes every combination of potential state and action that contains an estimated reward for if the agent were to take some action in a given state.

Q-Update Equation

Model-based agents require a function to update their q-tables, which is satisfied by the Q-Update equation as follows:

*s* represents the current state, *a* is the action taken in state *s,*  *s’* is the resulting state from taking action *a* in state *s*, and *a’* is the best action the agent could take in *s’* based on the Q-Table. *Q(s,a)* represents querying the Q-Table for the estimated reward given a state and an action. *α* is the learning rate, and *γ* is the discount factor. Lastly, *R(s’)* is the immediate reward for taking action *a* in state *s* that is returned by the environment.

It should be noted that SARSA uses a slightly modified version of this equation, and the Neural Network uses parts of this equation in its loss function, which will be discussed later.

Agent Implementations*[[1]](#footnote-1)*

In this experiment, four agents were implemented to solve the frozen lake environment: Dyna-Q, Q-Learning, SARSA, and a Neural Network. Each agent was designed such that it would suggest a move to make and update its policy based on the observation of the environment after taking an action. This made it simple for a controller class to construct a given agent, then query the agent for a move, then pass the observed environment back to the agent. This modular design resulted in simplified code and allows each agent’s implementation to be more digestible. This design was inspired from Bernoulli Bandits controller class in the first homework assignment.

Q-Learning Agent

The Q-Learing agent is constructed simply by using the techniques described earlier, ε-greedy and the Q-Update equation. The main loop of simulation queries the agent for an action, and the agent returns an action based on the ε-greedy policy. This agent implements a different value for ε based on the following equation:

Which ensures the agent has seen about 600 successes before it completely exploits its Q-Table.

Dyna-Q Agent

Dyna-Q is nearly identical to Q-Learning, however, when updating the Q-Table, Dyna-Q implements Q-Planning in addition to the Q-Update equation.

Q-Planning requires two additional data structures: a transition model and a history buffer. The transition model is similar to a Q-Table, but it stores the likelihood to transition to a particular state given that it was in some other state and took some action. After each action, these probabilities were updated using the maximum likelihood estimation:

*T(s,a,s’)* represents an index of the transition model. This equation represents the ratio of times taking action *a* in state *a* resulted in state *s’* to the times action *a* was taken in state *s*. These calculations take advantage of the history buffer so that the transition model can easily gather the information necessary by looping through the history buffer once.

The transition model is used to sample *s’* given a random *s* and *a* from the history buffer, which simply contains pairs of actions taken in particular states. Using the sampled *s’*, *a’* can be obtained from the Q-Table and *s,a,s’,*and *a’* can be used in the Q-Update equation to further update the model.

The history buffer has a limited size of 10,000 so that only recent actions are taken into account. Additionally, before sampling the history buffer for a random observation, duplicate pairs are removed to make the sampling unbiased to selecting moves that occur more frequently. For example, taking an action while the agent is in the corner or on the edge of the environment is prone to taking an action into the boundary causing the agent to end up where it started that turn. This can happen multiple times in a row making these actions not helpful for determining the best action for the agent to take since the move is not productive.

This agent does not decay ε using an exponential function, rather, its decay function is linear such that ε is initialized with value 1 and when the agent reaches a goal state, ε is subtracted by 0.0005, buy will never be less than 0.01.

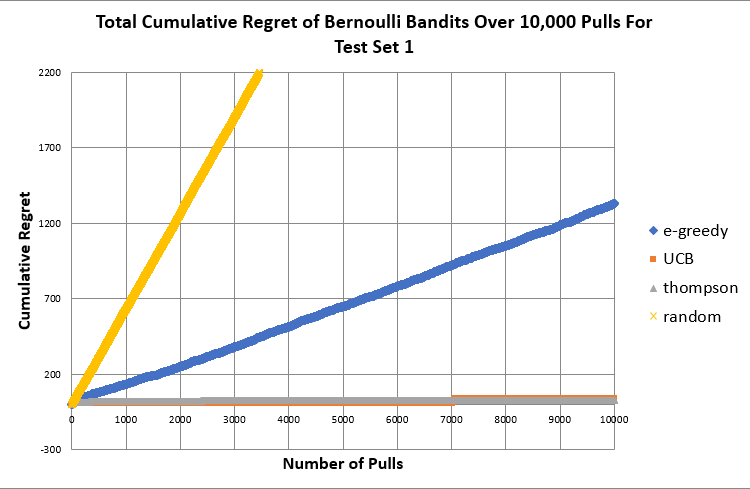
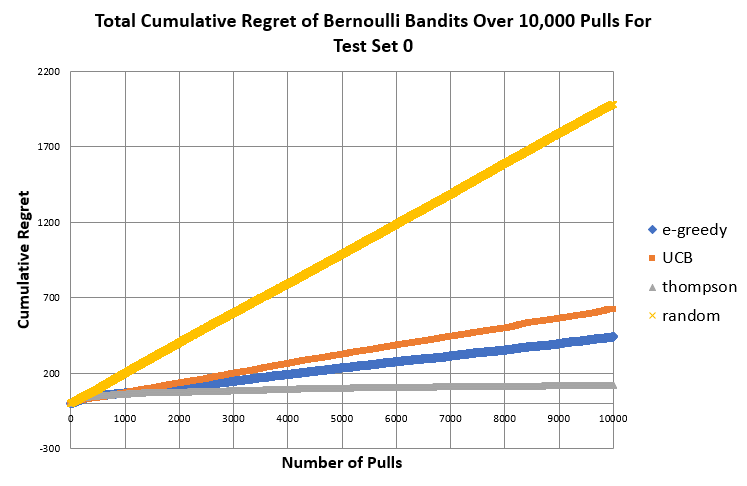
SARSA Agent

The SARSA algorithm is very similar to Q-Learning, however, it modifies the Q-Update equation such that the next observation of the environment is used rather than the predicted max according to the Q-Table. This was achieved by delaying the Q-Table update such that the *s* and *a* in the Q-Update equation are stored until the next action is taken when they are then used to update the Q-Table at that *s,a* index. At the time of updating, the true current state and action act as the future state and action for the previous state and action.

Neural Network Agent

The previous agents solved the 4-by-4 sized frozen lake because the state space is relatively small and a Q-Table will not be too large. If an environment’s state space becomes large, then these agents will struggle to keep their Q-Tables because they may run out of memory. A counter to this issue is to use a neural network where the policy can be determined by function approximation.

Results



The charts above show the cumulative regret for each agent for 10,000 arm pulls for two different test sets. Test set 1 is designed to be more difficult for the agents to determine the best arm because the differences between the arm values are 0 except for one arm. Test set 0 is easier because the differences between the arms and best arm are large compared to the gaps in test set 0. Therefore, it is expected that the agents would require more arm pulls in test set 0 to determine the best arm because the regret for each pull will be relatively small. The results above show exactly that because the UCB and Thompson agents quickly determine the best arm. The ε-greedy agent still exhibits a large amount of cumulative regret in test 1 which is presumed to be due to this agent still exploring a random arm 20% of the time and when it explores it will attribute a large amount of regret whereas the UCB and Thompson agents will not explore other arms once the best one has been found. In test 0, the UCB and ε-greedy exhibit clear linear regret growth, but the Thompson agent appears to have a somewhat sub-linear regret growth. It is expected that Thompson sampling should perform better than the other agents, but all agents were expected to exhibit linear regret growth. Another note of unexpected behavior is the UCB agent’s performance being worse than that of the ε-greedy agent’s, but this could be due to the small gaps between the arms resulting in UCB determining the best arm to be one that is close to the best arm, but not quite. This would lead to UCB consistently choosing a sub-optimal arm and attributing more regret than the other two agents.

Minimax Search

An agent using minimax search was used to play two games: connect 4 and breakthrough. Minimax searches through potential game states using a game tree where each layer in the tree alternates describing the agent’s and opponent’s turns consisting of max and min nodes, respectively. Max nodes take on the largest value of its children, and min nodes take on the smallest value of its children. These values are propagated up to the root, being the current state of the game and the agent’s turn. The value attributed with the root and the child which provided the value determines the action that the agent should take.

This agent was implemented by creating the game tree using recursion and depth-first search. It’s tree structure also takes advantage of a custom Node class that contains the game state in which the node exists, its current value, which child its value originates, the action that led to its state, and a list of children based on the moves the agent could make in this game state. Those variables assist the agent in backpropagating values. In an effort of saving computation time, this tree was limited to 10,000 nodes which was checked in tandem with the check if the current node was a terminal state or not so that a leaf node would receive the same treatment as a terminal state. It should be noted that the value of a terminal state is dependent on the current turn of the game. The heuristic value generated is ignorant of who has the turn of the game, so this value could appear very good to the agent, when it is in fact the opponent’s turn which would result in the agent picking moves that were good for it’s opponent. This was handled by swapping the sign of the value if the node was a min node.

Results

The Minimax agent behaves drastically differently depending on whether it moves first or second, so the results will be split into two sections.

When moving first

The Minimax agent performed in very well at connect 4 against an opponent that made moves at random, winning 8 of 10 times. The agent also performed very well against the same agent in a game of breakthrough, winning 10 of 10 games.

The agent’s observed behavior while playing connect 4 was that it would attempt to fill the left-most column and when that column was full, it would fill the next available column.

While playing breakthrough, the agent would advance its top left pawn as far forward as it could until that pawn was taken by the opponent or it could no longer move, then it would advance the next pawn in the front row in a similar fashion.

When moving second

The Minimax agent performed in very well at connect 4 against an opponent that made moves at random, winning 8 of 10 times. However, the agent performed very poorly at breakthrough against the same agent, winning 0 of 10 games.

When playing connect 4, the agent behaved the same as it was when it moves first. This is likely because the game of connect 4 is relatively simple and the heuristic in this case was to make moves on columns with a large amount of space, which will be true for every column because there can only be one spot filled in any column after the opponent’s first turn.

When playing breakthrough, a drastically different behavior is observed than that of when the agent moves first. The agent would make moves that advanced its pawns one row at a time, meaning no pawn would advance to the next row until the forward-most row was filled with pawns. This leads to a situation where the Minimax agent has progressed a line of pawns into its opponent’s half of the board and eliminating many of the opponent’s pawns. This leaves the opponent with few pawns to choose from, so the few remaining opponent pawns quickly advance to the agent’s side and win the game as the agent slowly advances its row of pawns. This explains why the agent lost, but it does not explain why the strategy was different than when it moves first. It can only be speculated that the explanation for this change in strategy is due to the agent reacting to the moves of the opponent and when the opponent has moved first, the state space is completely different than what it would be if the agent first sees the initial game state. This could lead to different outcomes from the agent’s perspective causing it to change its strategy.

Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is an algorithm for game playing agents that does not rely on human-created heuristics. Instead, it determines what move to take by taking actions that lead to a higher probability of winning. This is done by expanding a tree all potential game states and simulating random moves until a terminal state is reached. The outcome of that terminal state is back propagated, and the parent nodes of the tree update their probability of simulated wins to losses.

This implementation of MCTS limited the tree size to 10,000 nodes which was tricky to terminate the search when the game had been played for a long time and there were less than 10,000 nodes generated from a game state. This was handled by checking the number of nodes generated from one loop of the search to the next. During each loop of the search, a node should be generated, so if the number of nodes is the same from one loop to the next, then no more nodes are going to be generated so the search is terminated. A custom MCNode class was used that stored the current turn of the tree which was inverted on each turn. This value was compared to the turn value of the first layer and if they were the same, then the MCTS agent had simulated to a winning state. Initially, one would expect that the turn value in the simulated terminal state would need to be equal to the root node’s turn value as that would indicate that it is the same turn in the terminal state as it is in the initial state. However, this is not correct This MCNode class also consisted of a copy of the current game state, its parent node, a list of its children, the number of times a node leads to victory, the number of times the node was visited, and the action that led to this node. The number of times a node leads to victory and the number of times it was visited is simply a sum of the number of those same values of its children and those values originate from the simulations. These values are used to determine the probability of the action leading to a victory.

Results

The MCTS agent played both games against an agent that picks moves at random and the Minimax agent. The MCTS agent won 9 of 10 games of connect 4 and 10 of 10 games of breakthrough against the random agent. It won 9 of 10 games of connect 4 and 10 of 10 games of breakthrough against the Minimax agent.

It is expected that MCTS should outperform the Minimax agent in both games because the Minimax makes decisions that are only as good as its heuristic; and in this project, the heuristics are designed to be very poor. MCTS is used in situations where there is not access to a good heuristic so that a good strategy can still be devised, so it only makes sense that the MCTS agent would dominate the Minimax agent in both games.

Alpha-Beta Pruning

Alpha-Beta pruning is a technique used in Minimax search that removes branches from the game tree that will never be reached assuming the opponent is rational. This is done simply by recognizing when a child’s value will not be propagated because it will be limited by one of its parent’s siblings.

The implementation of this Alpha-Beta agent starts by copying the Minimax agent, then some minor modifications to account for the pruning.

Results

When the Alpha-Beta agent plays the Minimax agent in a game of connect 4, whichever agent moves first will always win. This is due to observing that the agents will calculate the exact same move but whichever goes first gets to place their piece first. This results in the first 3 columns filling up with alternating pieces and the first agent to place their piece in the fourth row will win, which will always the be the agent that moved first. This is potentially due to the bad heuristic of this game which results in the agent attempting to fill one column at a time. However, the Alpha-Beta agent consistently had a tree height of 19 when playing connect 4 against the Minimax agent which had an average tree height of 41.

When the two agents play breakthrough, the Alpha-Beta agent generates a tree of height 11 while Minimax generates a height of 107. Similarly to connect 4, whichever agent moves first will win the game. This is due to the agent creating different strategies based on what turn it moves on as described earlier. This still means that the Alpha-Beta agent is able to win with a tree that is nearly 10 times less the height of a Minimax agent.

This shows that the Alpha-Beta displays the same behavior as Minimax, but does it much more efficiently by generating a tree that is significantly smaller than Minimax, making the runtime much quicker. For an increased speedup with no loss in performance, the Alpha-Beta pruning is clearly an improvement to Minimax.

1. Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved. [↑](#footnote-ref-1)