Kevin McCarville
CS - 7641 - Machine Learning
3/29/2024
Assignment 4

The focus of this assignment is on reinforcement learning. I will solve two Markov Decision Processes using three different algorithms: Value Iteration, Policy Iteration and Q-Learning.

Algorithms:

- Value Iteration - Value iteration is a dynamic programming algorithm used to find the optimal value function for a Markov decision process (MDP). It iteratively updates the value function by applying the Bellman optimality equation until convergence, often using a specified threshold. By computing the value function, it enables determining the optimal policy by selecting actions that maximize the expected return.
- Policy Iteration - Policy iteration is an iterative algorithm used to find the optimal policy for a Markov decision process (MDP). It alternates between two steps: policy evaluation and policy improvement. In the policy evaluation step, it computes the value function for a given policy, while in the policy improvement step, it updates the policy by selecting actions that maximize the expected return according to the current value function. This process continues until the policy converges to the optimal policy.
- Q-Learning - Q-learning is a model-free reinforcement learning algorithm used to find the optimal action-value function for an agent interacting with an environment. It updates Q-values based on observed transitions and rewards, using the Bellman equation to estimate the expected return of taking an action in a given state. By iteratively updating Q-values through exploration and exploitation, it enables the agent to learn the optimal policy without requiring knowledge of the environment's dynamics.

Problems:

The two problems I will be solving are Black Jack and Frozen Lake.

- The "Blackjack" problem from OpenAI Gym is a classic reinforcement learning problem designed to teach agents how to play the card game Blackjack. In this environment, the agent plays against a dealer, aiming to achieve a hand value as close to 21 as possible without exceeding it, or to force the dealer to bust (exceed 21).
    - State Space: The state is represented by a tuple containing three elements: the player's current sum of cards (an integer from 4 to 31), the dealer's one showing card (an integer from 1 to 10), and whether or not the player holds a usable ace (a binary value).
    - Action Space: The agent can take one of two actions at each state: 'hit' (request another card) or 'stick' (end the turn).
    - Rewards: The agent receives a reward of +1 if it wins, -1 if it loses, and 0 for a tie.
    - Game Termination: The game terminates when the player's sum exceeds 21 (player busts), or the player decides to 'stick.' Then, the dealer plays according to a fixed policy (draws until their sum is 17 or more). The game ends with a win/loss/tie outcome.
    - The goal of the agent is to learn a policy that maximizes its expected cumulative reward over time. This typically involves learning when to 'hit' and when to 'stick' based on the current state and potentially past experiences. Reinforcement learning algorithms like Q-learning or deep Q-networks can be applied to train an agent to play Blackjack effectively in this environment.
- The "FrozenLake" problem from OpenAI Gym is another classic reinforcement learning environment. In this problem, the agent navigates a grid world represented as a frozen lake, trying to reach a goal tile while avoiding holes in the ice. Here's an overview:
    - State Space: The grid world is represented as a 20x20 grid, where each cell can be one of four types: start (S), frozen surface (F), hole (H), or goal (G). The agent starts at the start cell (S) and aims to reach the goal cell (G) while avoiding holes.
    - Action Space: The agent can take one of four actions in each state: move up, down, left, or right. However, due to the slippery nature of the frozen lake, the agent may end up moving in a different direction with a small probability (0.33) than the one it intended.
    - Rewards: The agent receives a reward of 1 if it reaches the goal cell, and 0 otherwise (including falling into a hole).
    - Game Termination: The game terminates when the agent reaches the goal cell or falls into a hole.

- The FrozenLake environment is stochastic and has a high chance of slipping on the icy surface. This makes it a challenging environment for reinforcement learning agents, as they need to learn to navigate the grid world effectively despite the uncertain outcomes of their actions.

These two games are interesting as they contrast each other in a number of ways which should provide for fruitful analysis. Frozen Lake is grid world, stochastic (in terms of action space), and large (state space = 400). Blackjack is non-grid world, deterministic, and slightly smaller (state space ~ 290). The state space difference is even bigger when you factor in that blackjack has two actions and frozen lake has 4 and that the number of steps of blackjack is rarely more than 4-5 while for frozen lake it can be in the thousands.

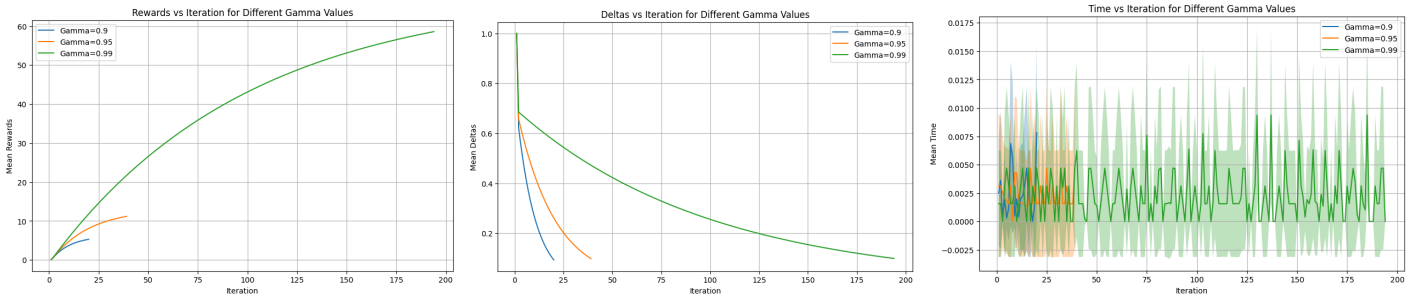
I experimented with a number of other potential environments but had to switch for logistical reasons, however exploring these environments was still valuable for understanding the assignment. I needed a transition and reward matrix for a non-grid world problem for value iteration and policy iteration but those were largely not available in the gym api which limited my ability to use the library. This put an end to my exploration of lunar-lander, cart pole, mountain car and space invaders. I tried to engineer this for cart pole and mountain car but for both I had to discretize the continuous state space. I was mildly successful in doing this but the state space exploded and the algorithm had a very difficult time learning. I believe I could have successfully trained in these environments but for the sake of the assignment, I switched to blackjack for my non-grid world assignment. I was still fortunate to gain a better understanding of these algorithms and environments in the process and am better for the journey. After choosing blackjack, I considered the 290 states as a small state environment in terms of RL size and thus expanded Frozen lake from the 8x8 to 20x20 to 400 states to represent my large environment.

Hypothesis:

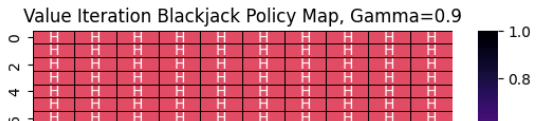
First, I think the Frozen Lake is going to be considerably harder to train than blackjack. It has more states, it has more actions, and it has a stochastic action space as the agent can “slip on the ice”. Blackjack also has an advantage around reward structure in that it gets its feedback usually after 1 or 2 steps or at most 4-5, while Frozen lake could take hundreds of steps before it finds the reward. I also predict that value iteration and policy iteration will outperform Q-Learning, though this is a safe assumption as Q-Learning is utilizing considerably less info than the other two. I think that value iteration will run the quickest, then policy iteration, and then Q-Learning, as policy iteration does more computationally intensive work each iteration but policy iteration will run with less iterations as it's doing potentially larger updates per iteration. Q-Learning will be considerably slower on both fronts as it needs to organically explore the environment. Finally, around exploration, I predict that it won’t be that crucial for these environments. Blackjack only has 2 actions, so it shouldn't take too much exploring to look under every proverbial rock for potential unexpected rewards. For Frozen Lake, the stochastic nature of the action space does a pseudo natural exploration into new states for the agent, thus taking suboptimal actions are less likely to be fruitful. I read a handful of papers on this, suggesting, “don’t explore just to explore if the action doesn’t make sense”[5]. I predict this will be the case for Frozen Lake.

Value Iteration:

First we will look at Value Iteration for Blackjack. I ran Value Iteration on Blackjack with varying gamma values: 0.9, 0.95, 0.99. Here are its time performance, convergence speed and reward per iteration:

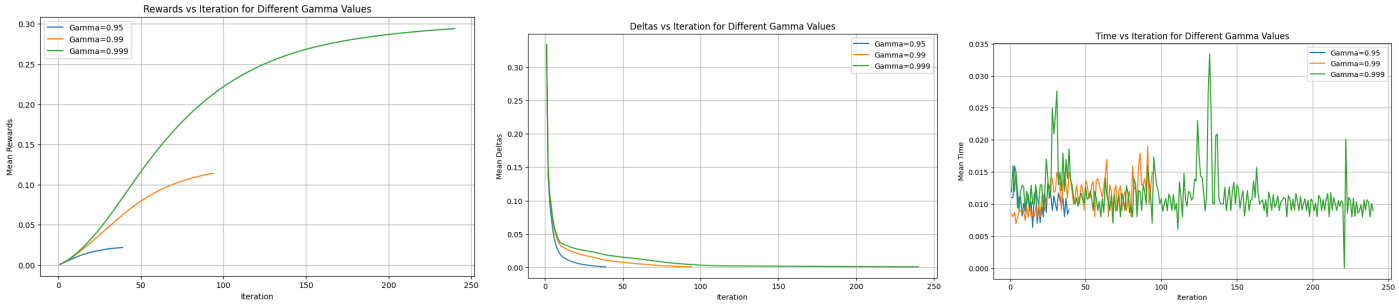


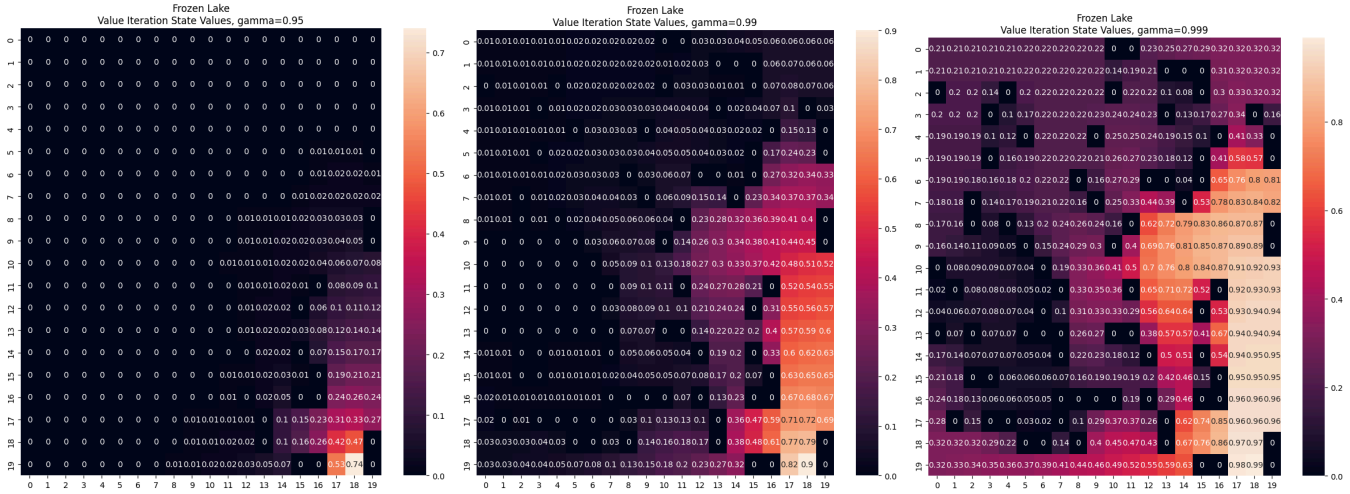
I defined convergence as when the update, delta, per iteration, decreased below a predefined epsilon level. I used epsilon as 0.1 which we can see each delta line converging to in the middle chart. I experimented with varying epsilon but all it did was extend training time. The resulting policy was identical, so for this exercise we will keep it static at 0.1. Also of note is that the rewards have no standard deviation. Value iteration is a deterministic function, so each experiment will yield identical results. The 0.9 gamma converges after just 20 iterations, the 0.95 after 39 iterations and the 0.99 after 194. The rewards chart looks like it keeps growing. This is because the rewards per iteration are still outpacing the original V value after 194 iterations. It hasn’t reached an equilibrium yet. I ended training before this as the policy isn’t changing so the V(s) is really just a number that won’t affect our actions. The value iteration process actually converges after a single iteration if I define it based on convergence to its final policy map (below on the right). Again this is due to the deterministic nature of the environment. It simply has all the information it needs



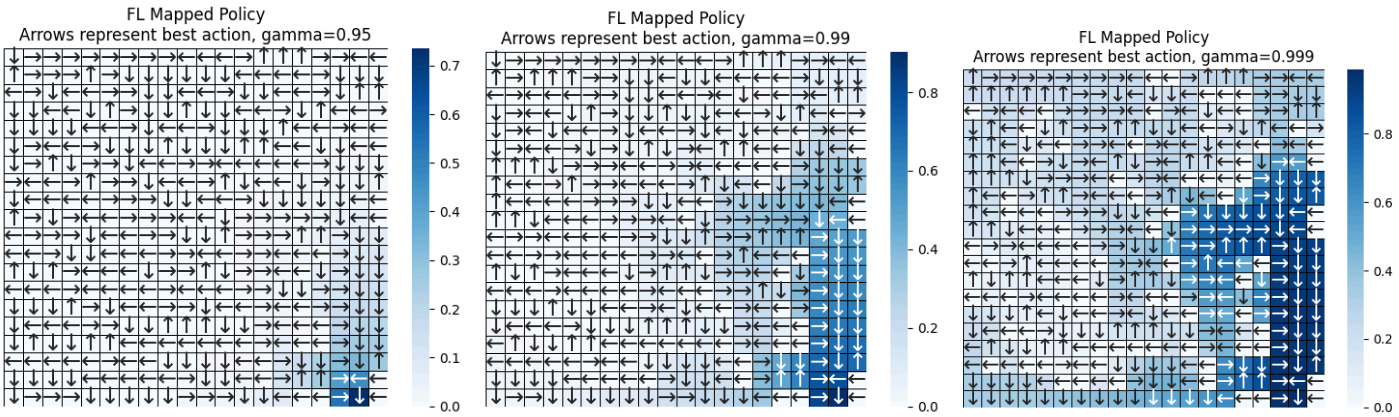
with a full transition and reward matrix. The policy is the optimal blackjack policy given our environment. We can see it making all the logical plays like hitting on anything below a 10, and staying on everything 17 and above. We can also see the increased value in the dealer having a 15 or 16 on the x axis and the increase in value of us holding 19-21. The time per iteration was pretty stable throughout. Value iteration runs very quickly. The total time run for all three experiments is under 1 second. The state space is not very big and the deterministic nature of the problem leads to very quick convergence.

Now we look at Frozen Lake. Frozen lake is a stochastic environment, however for value iteration (and later policy iteration), we know the transition and reward matrices so the process is actually deterministic. So again, we won't see any standard deviation amongst experiments. For this I needed to tweak epsilon down to 1e-3. This is really just hyperparameter tuning of trying to define convergence based on delta to line up roughly with when optimal policy is reached. 1e-3 gave me the best results here. For gamma we will use 0.95, 0.99, 0.999. These yielded the most interesting comparative results for this project as we will see below. A single run can have a lot of steps so its important to keep gamma high so rewards flow through to previous steps. Finally, the 20x20 environment is a custom environment so I could ramp up the state size. Where the holes in the lake are will be fairly obvious from the state-value plots.





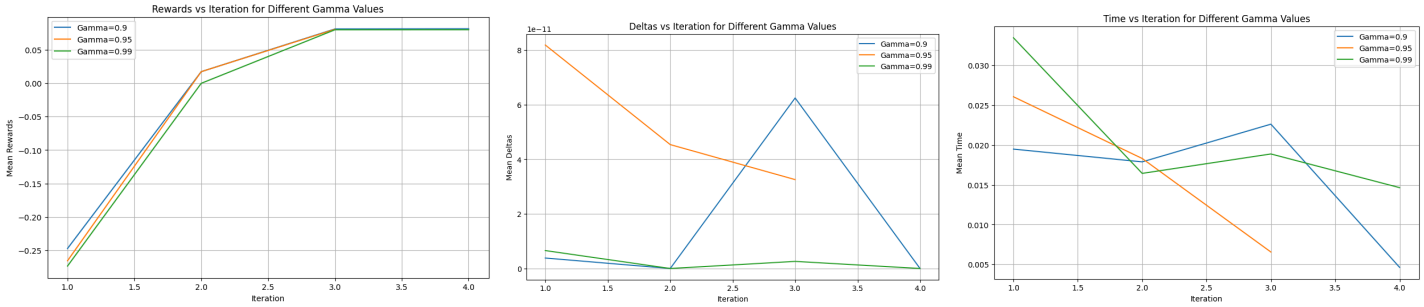
For the reward plot, unlike blackjack, we start to see some equilibrium beginning to be reached. Again, there is no need to focus on the different levels as they are explicitly determined by the gamma variable. For the delta convergence plot, we again see fairly rapid convergence despite the larger state space. The gamma values 0.95, 0.99, and 0.999 converge in 39, 94, and 240 iterations respectively. The time chart doesn't tell us too much. All three iterations took 3.4 seconds to run with the breakdown for each being: 0.34s, 0.81s and finally 2.24s for the 0.999 convergence. Significantly slower than black jack, mostly due to the increased state size, but still fast. Each iteration is the same run time regardless of gamma which makes sense as that is just an input into the same calculation. The policy maps above are difficult to read and due to the increased state space, it's hard to decipher how far or near we are from optimal. The stochastic action space is also convoluting results. For example, the agent may not want to go to a goal state right next to it if that risks slipping into a hole on the other side of the agent's location. We do see the arrows increasingly pointing away from the holes which is logically the right maneuver to avoid "slipping" left or right into them.



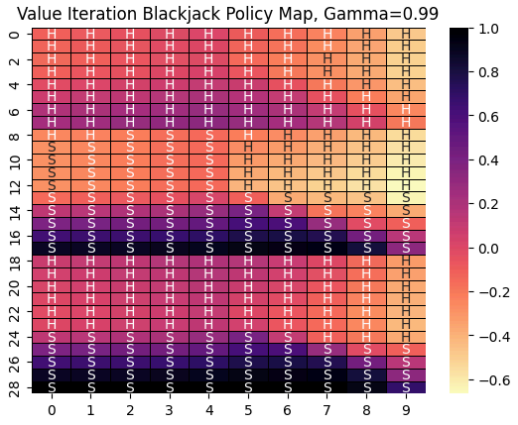
I did try to run with the algorithms defined as “converged” when an iteration produced no policy changes, however this led to early stopping for some gamma values in the 40s where the policy was far from optimal and needed a few more iterations to update state values and eventually, in turn, policy. Using epsilon thresholds instead produced better policy outcomes. The heatmaps are more informative. The policy’s understanding of the environment improves dramatically with the higher gamma value. We can more clearly see the holes and the ways the policy is avoiding them. We can also see the policy logically applying very small weights to the left and lower side of the grid where it is far from the goal with many holes in its way. The difference between the 0.95 and 0.999 heat maps is stark as there is just not enough information leaked to the left side of the grid due to the large state space. Compare this to the smaller state space of blackjack where 0.9 gamma still gave us perfect policy after just 1 iteration.

Policy Iteration:

I will now run policy iteration on our MDPs and compare the results between them and to value iteration. Again we started with blackjack.

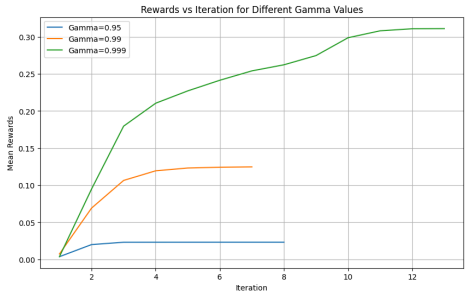


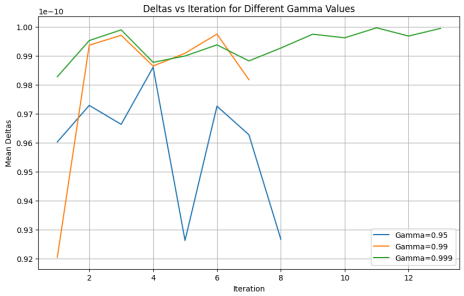
We stuck with the same gamma values as before of 0.9, 0.95, and 0.99. There was minimal difference in performance in terms of rewards based on gamma. With policy iteration I now defined convergence as no change to the preferred policy. Now, the gamma value isn’t as essential for convergence as it is with value iteration where it is part of the equation that determines if we are below our delta value. The algorithm converges after only 3-4 iterations. Oddly the middle gamma of 0.95 was the quickest to converge in just 3 iterations, but I believe that is just a strange edge case of the environment. We note that they converge at different values for delta. Delta in policy iteration is used during the policy evaluation step. It determines if we are under our theta threshold (1e-10) to move on to our policy improvement step. Thus it does not need to be 0 for the policy to converge. Our times got quicker as the runs went on, my guess is this is due to less updates to policy. They were all very fast though so I would not read too much into this. The value iteration ran for all three gammas in 0.2 seconds. Extremely fast and faster than value iteration which is surprising as traditionally policy iteration is a little slower as it makes more updates. My guess here is that the nature of the problem with only two actions and a small number of states can lend itself to faster convergence to Policy Iteration. Policy Iteration converged in considerably less iterations than value iteration but it's an apples to oranges comparison as the nature of their iterations are fundamentally different. I give the win here to policy iteration though, based on its faster clock time. On the right is the policy model. This was identical for all gammas and I had to move gamma all the way down to 0.1 to get anything different.

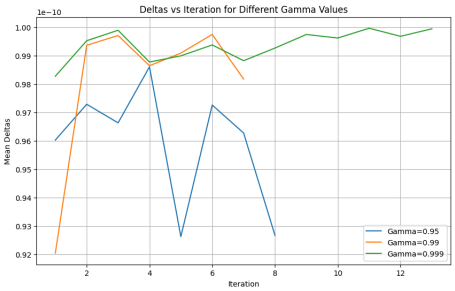


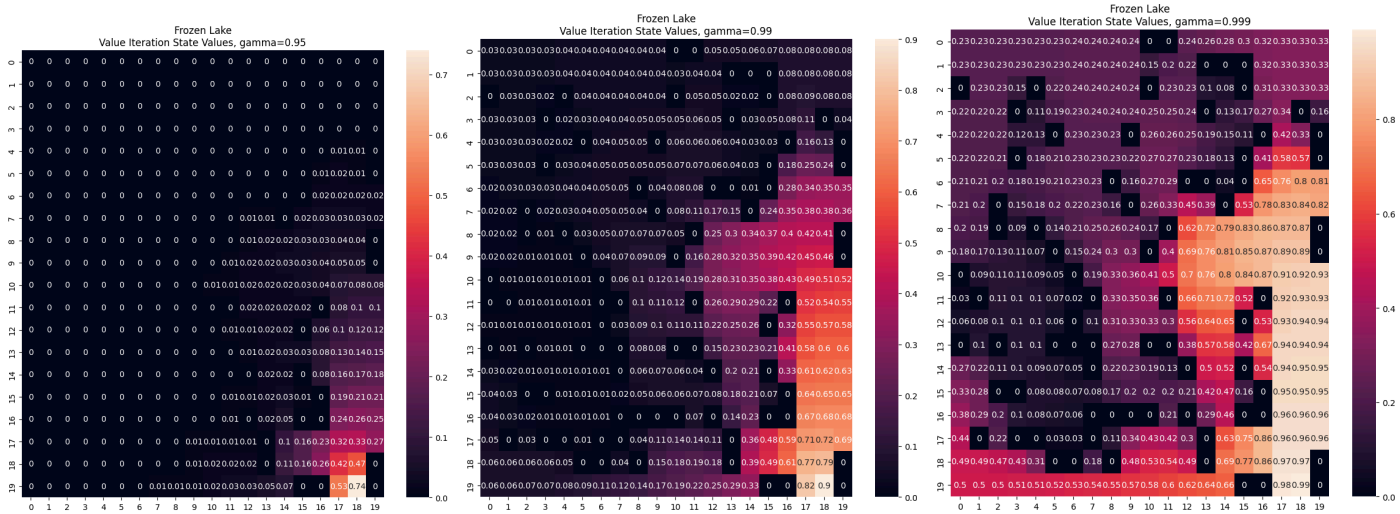
Moving around theta made no difference as well. It's identical to the policy of value iteration. This is simply a small state and action space with no strange edge cases which lends itself to fast convergence to a consistent optimal policy.

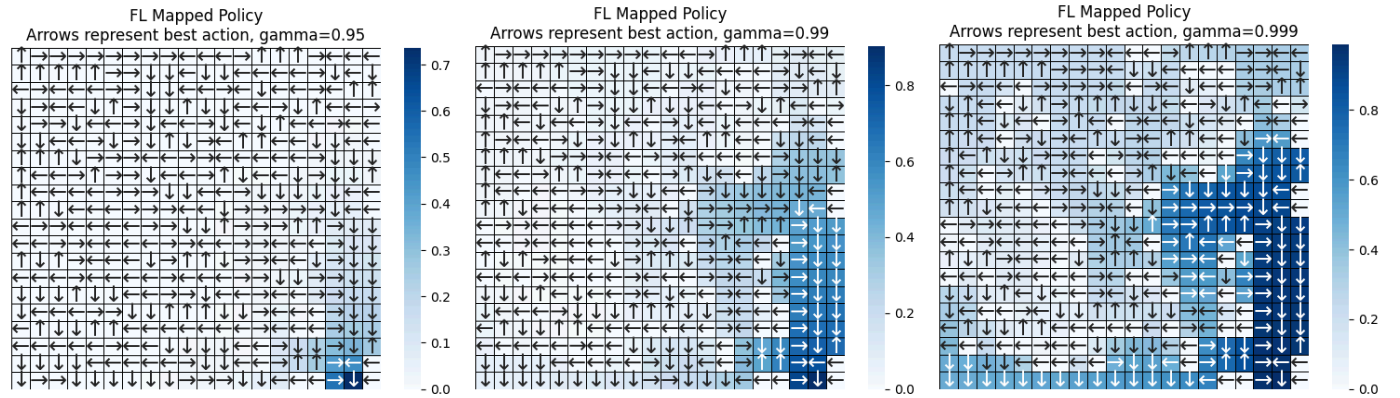
I will now repeat the process for Frozen Lake.









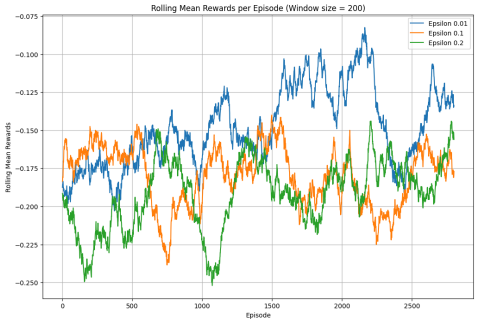


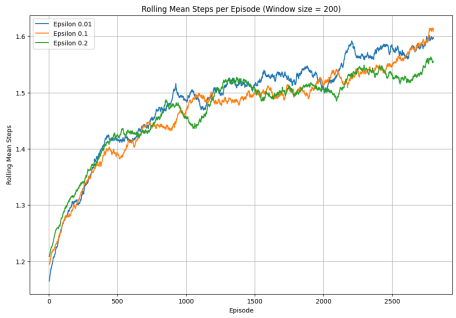
Again we see more of a plateau around rewards at the end of the iterations than in value iteration. This makes sense as our convergence definition is when the policy is static and thus so are rewards. Again we see deltas not at 0 when the policy converges which was not the case with value iteration. The algorithms converged after 8, 7, and 13 iterations respective to each gamma value. This is significantly more than the blackjack iterations due to the larger state space. Note that because we set the initial actions randomly, each run will have some variability which can lead to odd results like a higher gamma converging quicker but it shouldn't affect the final convergence. The times were also much longer for policy iteration than value iteration for frozen lake. The run time total was 54 seconds, compared to just 3.4 seconds for value iteration. We also see the average run times vary by gamma value. My theory is that the higher gamma creates more intermediate policy changes that lengthen the policy evaluation step. Above is the policy heat map for policy iteration. The results here are extremely similar across both algorithms. In terms of the resulting policy, they are nearly identical, with a few arrows flipped here and there but those are exclusively, far away from the goal with low state values where any action's superiority is probably negligible. The 0.999 gamma also tends to have higher state values than the value iteration map, most notably on the bottom row of the grid. The general heatmaps areas of strengths and weaknesses are nearly identical across the two algorithms which is to be expected after we discovered how similar

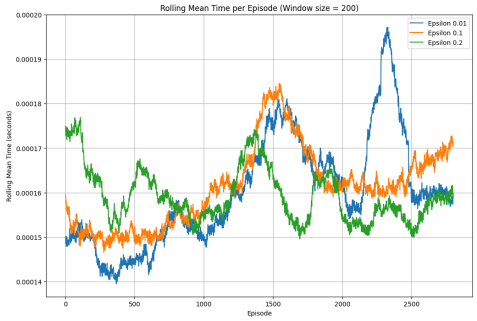
the resulting policies are. For both of these environments, Value Iteration and Policy Iteration both create similar and very strong robust policies.

Q-Learning

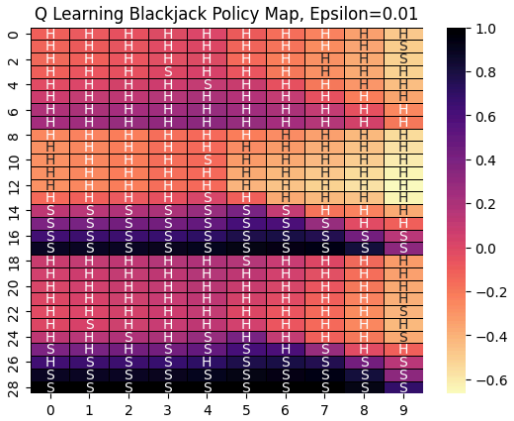
Now I will implement Q-Learning. Value Iteration and Policy Iteration are great algorithms, however in the real world we often won't have a reward or transition matrix. Q-Learning solves this problem through the exploration and exploitation tradeoff. First we will look at blackjack. Here were the results for 3 different epsilon values for exploration = 0.01, 0.1, 0.2. I ran each epoch for 3,000 iterations and then repeated it 10 times to average the results. I did not include the standard deviation in the charts below as it was too large and dominated the graph. First to touch on the run times, the algorithm ran very quickly. All 10 trials of 3,000 iterations each took only 15 seconds total to run. It's interesting that the number of steps was higher for the higher epsilon scores. That leads me to believe that the exploration is causing more hits when the player should stay thus extending the game than vice versa. Finally to look at the rewards chart, we see a general increase in performance and outperformance by the epsilon=0.01 value. This makes sense as the state space is simply very small. There are only 290 states and 2 actions to take. The Q-Learning algorithm can traverse the space very quickly. After that point when the optimal policy is found, any exploration will be a net negative to performance, which is what we see in the rewards chart.







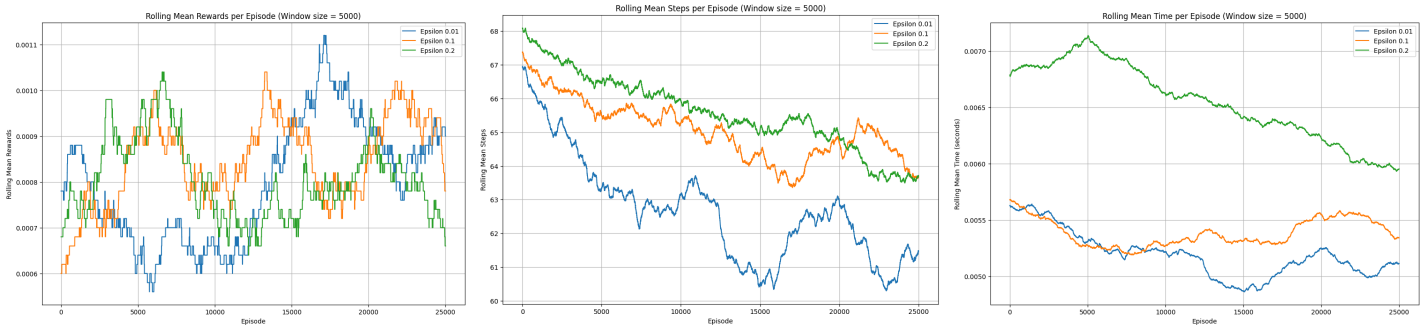
I also tried tuning the learning rate, alpha, and the discount factor, gamma, but they had minimal impact on performance. Finally we look at the policy map and we see that it gets close to the optimal policy but does not get all the way there. We see some hits where we should stay, mostly in areas where the dealer has a high card and the player is in bad shape either way. Also some stays on the lower hands which could be just the player getting lucky with the dealer busting when the player stayed when he shouldn't have. With this in mind, I would say the algorithm hasn't fully converged. For Q-Learning, I will define convergence as when the mean reward levels off and that has not happened here. It's a subjective definition and not a perfect one but with Q-Learning it's difficult to define when the work is done and the optimal or at least sufficient policy has been reached. We could keep training and get to the optimal policy. It's noteworthy the model has done as well as it did with the limited iterations. This is certainly a product of the small state space of blackjack.

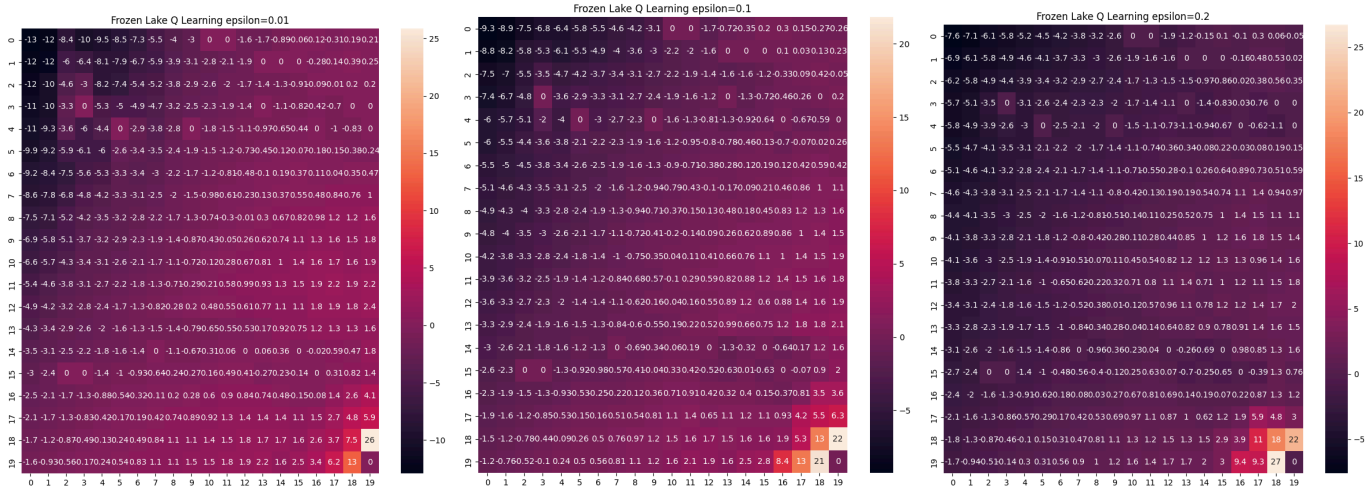


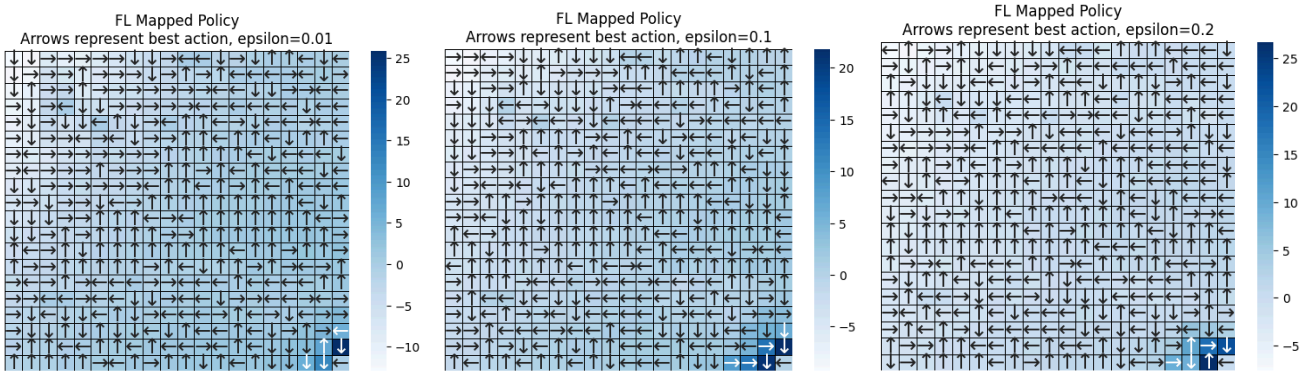
Now we run again for frozen lake. For Frozen Lake, I had to make considerable modifications to the environment. After 10 million observations and multiple runs, it never reached the reward and thus it couldn't learn anything. I then made numerous changes to the algorithm to help it along:

- I gave it intermittent rewards as it moved closer to the goal and magnified the reward when it reached the goal to ripple through nearby states.
- I also gave it a negative reward for falling in a hole to teach it to stay alive
- I added an exploration bonus for the agent to go to an area it hasn't seen as often
- I removed the 200 step maximum on the environment. The gym environment has a maximum of 200 steps per episode so if it doesn't find it in that time, there is no reward. This is actually a difficult limiting principle. The agent needs to make 38 steps in the correct direction of the goal state to get the reward. This is while 2/3rds of the time it is "slipping on the ice" and not moving in the desired direction. The stochastic nature of the action space dramatically slows down learning and the state space is just too big. I found a work around to override this restriction and allow the agent to continue running past 200 steps.
- Finally I removed most of the holes from the environment. It was just too difficult. I kept the holes in rows 1, 2, 4, 5, 15, and 16 and removed all the rest. With a little experimentation, this was the best compromise.

The results are below. This took a heck of a lot of tuning. First for gamma, I used 0.99. The results between 0.95 and 0.9999 were mostly negligible but anything lower than that and the agent didn't really learn. It would find the reward serendipitously but the discount was too high so the finding wouldn't ripple to nearby states. For alpha I used 0.5. A very high learning rate. I needed the agent to really absorb the little feedback it got. Having the state biased towards the original value of 0 seemed illogical and the results supported that. The stochastic nature of the action space really slowed down the learning as the agent would take the correct action, slip on the ice, end up in a worse state and receive negative feedback for the action taken. This bedeviled the learning and was a true educational note for me as a developer. Below I only show the epsilon results as the various gamma and alpha values I tried had a lot of runs where the reward was never found and thus the results were boring and nonsensical. The majority of tuning I did was around the intermittent rewards. This took a lot of trial and error to find the right equilibrium. At one point the agent was getting too much reward for exploration and would go away from the reward just to see new corners of the map for no other reason throughout the entirety of the run. Other times, the intermittent rewards for movements towards the goal were so strong that the agent would keep going right into a hole in the ice because it was closer to the goal state. Some exploration policies emphatically outperformed others.







The finicky and tedious nature of this process is something I'm sure I will come across throughout my time in reinforcement learning. As for the results above, the agent typically found the reward around run 800. It started to accelerate from there but not dramatically. Removing the 200 step max was a huge boost to performance in this domain. I ran 10 trials, each with 30,000 iterations and then averaged the results. The heat map clearly shows a trend that the agent has learned something about the

environment and how to move towards the bottom right of the map. The 0.01 epsilon map shows a more consistent understanding of the state space but this did not translate to any outsized performance or improvement in policy. The returns chart is mostly horizontal though it does show an upward trend for all three epsilon values, and particularly 0.01. The policy map is naturally very jumbled. There are some logical maneuvers close to the reward and there is a general trend in the rest of the map to get there, especially in the early arrows that head mostly right and down, but the policy is chaotic. It is thus very different from the policy found by value iteration and policy iteration which reached very similar end states (and in a more complex environment). I attribute this to the stochastic nature of the action space. This is the first algorithm that has had to deal with the randomness of the environment as value iteration and policy iteration could just use the probabilities of transitions. There is just so much noise compared to signals in terms of the messages the agent is receiving. I also learned the hard way what a big state space can do to slow down training. The full run was about 90 minutes. One thing that is noteworthy is the decrease in steps amongst all epsilons as the agent starts to learn. It is clearly doing less random movements and trying to move more towards the goal which is good. Its iterations are just ending earlier as it hits a hole in the ice. This is seen in the decrease in wall clock times per run, as well as the steps decrease. It is also consistent with the epsilon values, as more exploration leads to more steps taken and longer runs.

	BlackJack	Frozen Lake
Value Iteration	0.7s	3.4s
Policy Iteration	0.2s	54s
Q-Learning	15s	84m 37s

Conclusion:

This exercise shed light on all of the concepts we have been studying in reinforcement learning:

- The MDP process of states, models, actions, and rewards used to craft a policy
- Delaying reward matters and minor changes matter (lecture 10)
- Intermediate reward states can vastly change behavior (lecture 11)
- The horizon of rewards and how they affect policy (lecture 12)
- Using the Bellman recursive equation to converge on the optimal policy (lecture 18 and 19)
- How the Bellman equation is the backbone of Policy Iteration and Value Iteration (lecture 20)
- Policy Iteration making bigger jumps and thus converging in less iterations than Value Iteration (lecture 22)
- How Q-Learning can still converge on an optimal policy without a reward and transition matrix (lecture 8, RL 2)

Throughout this assignment, we saw the effectiveness of policy iteration and value iteration but how we won't usually have the transition and reward matrices in real life and how much harder that makes the RL task. We experienced the larger state (and action) space of Frozen Lake make learning considerably more difficult than Blackjack. Frozen Lake's nature of a stochastic action space vs. the deterministic nature of Blackjack only furthered the difficulties. We also saw how there are some nuances to grid world vs. non-grid world environments. In our example, it was significantly more steps for the grid world agent. The biggest takeaway from me for this assignment is how much tuning is necessary to properly incentivize our agents in novel environments. Learning rate, discount rate, exploration rate, convergence definition, and reward structure, among a handful of other parameters to tune, make the job of the developer a formidable one. Though this also empowers us to craft the right mix of environmental incentives to train our agents. For example, the exploration reward I used for the Frozen Lake Q-Learning was directly inspired from the famous Montezuma's Revenge solution. This freedom to engineer everything about the inputs the agent receives is an exciting revelation. For better or worse, these agents will do exactly what we incentivize them to do. In order to train them to succeed in the environments we will come across, from self-driving cars to playing pong, there really are no limitations to what kind of state spaces and reward structures we can feed them in order to get the actions we desire.

References:

1. README file including for instructions for running the algorithms included
2. https://bambielli.com/posts/2018-07-22-comparison-of-four-randomized-optimization-methods/
3. https://louis-dr.github.io/machinel2.html
4. https://www.gymlibrary.dev/index.html
5. https://arxiv.org/abs/2105.09992
6. Main libraries: sklearn, keras, tensorflow, pandas, numpy, matplotlib, xgboost, bettermdptools, gym