HW #4: Value/Policy Iteration, Discretization

Name: Gyanig Kumar

Deliverable: PDF write-up and zip file of project directory. Your PDF should be generated by replacing the placeholder images in this LaTeX document with the appropriate solution images for each question. Submit your PDF and code via the course Canvas. The scripts will automatically generate the required images; recompile this LaTeX document to populate it with content. If you prefer not to install LaTeX locally, use an online platform like Overleaf.

Graduate Students: Complete the entire assignment.
Undergraduate Students: Complete only questions without **(GRAD)** labels (i.e., skip max-ent Value Iteration and look-ahead policies).

You will need to install matplotlib and the Gymnasium Environment for Python:

```
pip install gymnasium
pip install matplotlib
```

or

```
conda install gymnasium
conda install matplotlib
```

1 Gridworld Environment

1.1 Value Iteration [20 points]

First, you will implement the value iteration algorithm for the tabular case. You must fill the code in code/tabular solution.py below the lines if self.policy type == 'deterministic vi'. Run the script for the two gridworld domains and report the heatmap of the converged values.

1.2 Policy Iteration [20 points]

Next, you will implement the policy iteration algorithm for the tabular case. You will need to fill the code in code/tabular solution.py below the lines if self.policy type == 'deterministic pi'. Run the script for the first gridworld domain and turn in a graph with policy value (accumulated reward) on the vertical axis and iteration number on the horizontal axis.

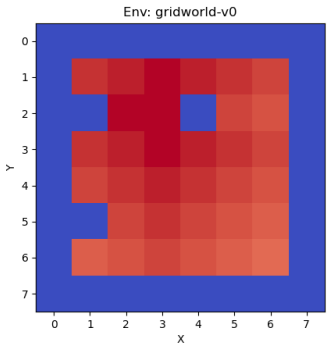


Figure 1: Value function heatmap for Gridworld (v0) using Deterministic Value Iteration

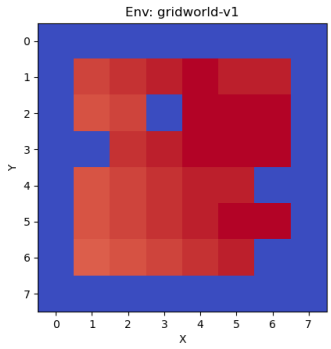


Figure 2: Value function heatmap for Gridworld (v1) using Deterministic Value Iteration

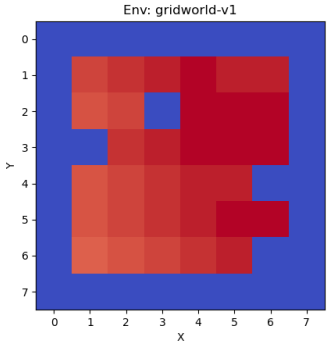


Figure 3: Value function heatmap for Gridworld (v1) using Stochastic Policy Iteration

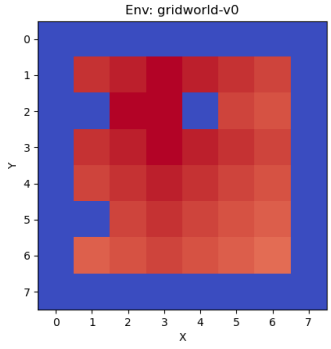


Figure 4: Value function heatmap for Gridworld (v0) using Stochastic Policy Iteration

2 Mountain Car Environment

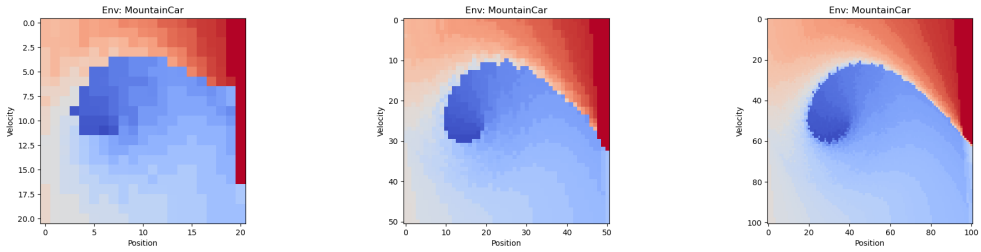
2.1 Nearest-Neighbor Interpolation [20 points]

Value Iteration can only work when the state and action spaces are discrete and finite. If these assumptions do not hold, we can approximate the problem domain by coming up with a discretization such that the previous algorithm is still valid. The MountainCar domain, as described in class, has a continuous state space that will prevent us from using value or policy iteration to solve it directly. One of the solutions that we came up with for this was nearest-neighbor interpolation, where we discretize the actual state space S into finitely many states = 1, 2, ..., n, and act as if we are in the nearest to our actual state s. Implement Value Iteration with nearest-neighbor interpolation on the MountainCar domain. You will need to add code in code/continuous solution.py below the lines if self. mode == 'nn'. Run the script and report the state value heatmap for MountainCar, discretizing each dimension of the state space (position and velocity) into 21, 51, and 101 bins.

2.1.1 Deterministic Value Iteration

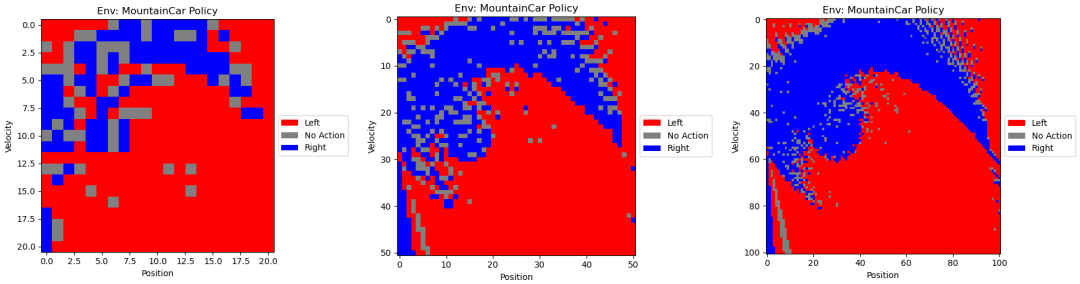
2.1.2 Deterministic Policy Iteration

2.1.3 Stochastic Policy Iteration



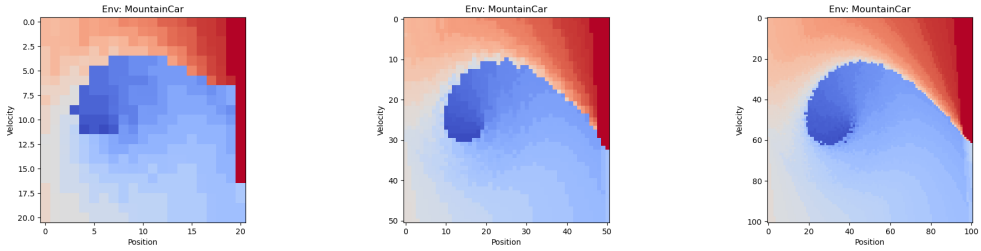
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 5: State value heatmaps for MountainCar using Deterministic VI with nearest-neighbor interpolation



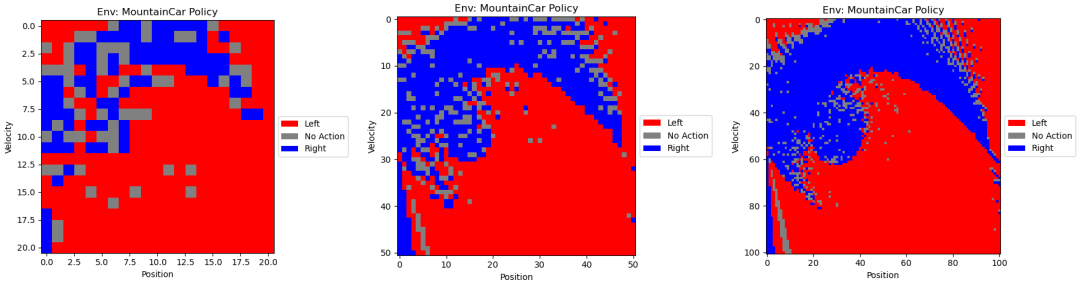
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 6: Policy heatmaps for MountainCar using Deterministic VI with nearest-neighbor interpolation



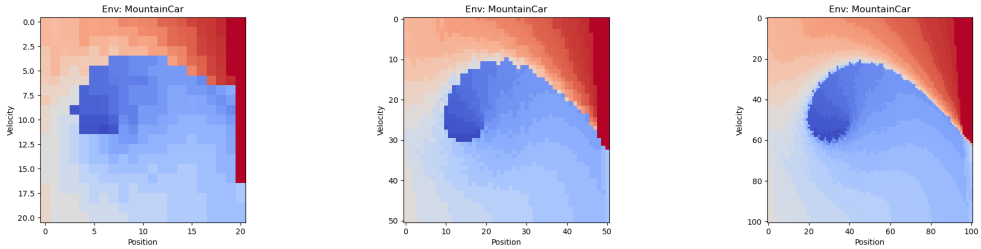
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 7: State value heatmaps for MountainCar using Deterministic PI with nearest-neighbor interpolation



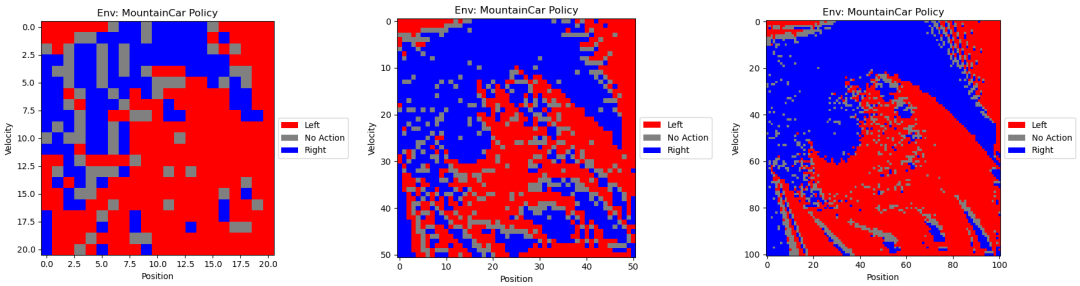
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 8: Policy heatmaps for MountainCar using Deterministic PI with nearest-neighbor interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 9: State value heatmaps for MountainCar using Stochastic PI with nearest-neighbor interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 10: Policy heatmaps for MountainCar using Stochastic PI with nearest-neighbor interpolation

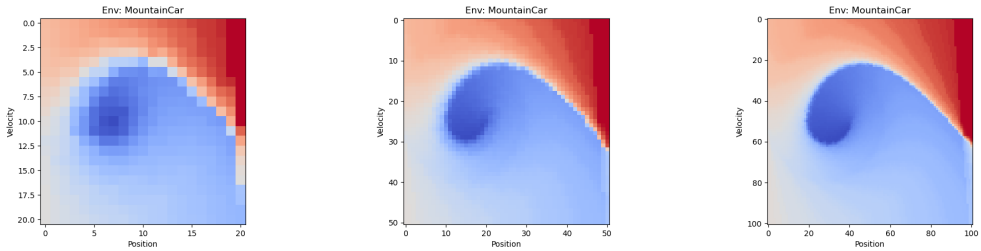
2.2 Linear Interpolation

Nearest-neighbor interpolation is able to approach the optimal solution if you use a fine-grained approximation, but doesn't scale well as the dimensionality of your problem increases. A more powerful discretization scheme that we discussed in class is n-linear interpolation, an n- dimensional analogue of linear interpolation. Add your code within code/continuous solution.py below the line if self. mode == 'linear'. Just as before, report the state value heatmap for MountainCar with discretization resolutions of 21, 51, and 101 points per dimension.

2.2.1 Deterministic Value Iteration

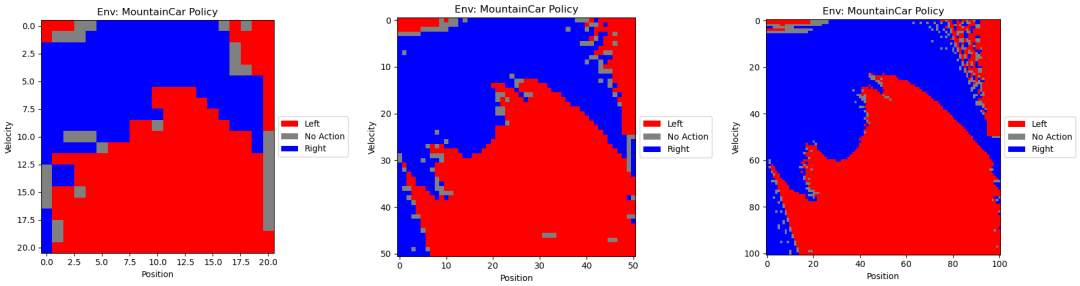
2.2.2 Deterministic Policy Iteration

2.2.3 Stochastic Policy Iteration



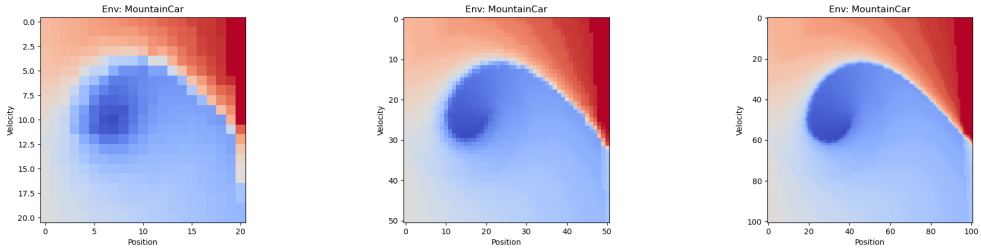
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 11: State value heatmaps for MountainCar using Deterministic VI with linear interpolation



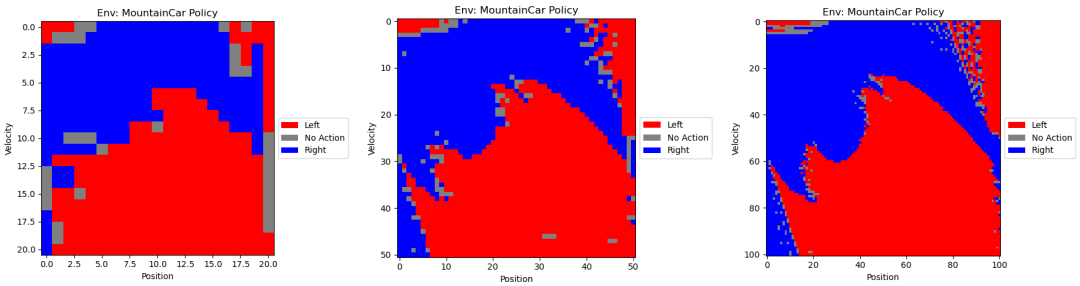
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 12: Policy heatmaps for MountainCar using Deterministic VI with linear interpolation



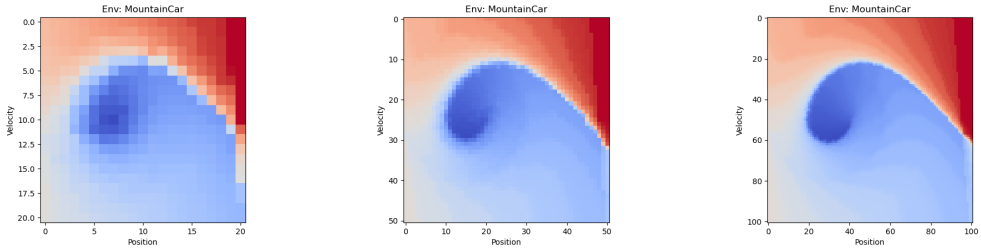
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 13: State value heatmaps for MountainCar using Deterministic PI with linear interpolation



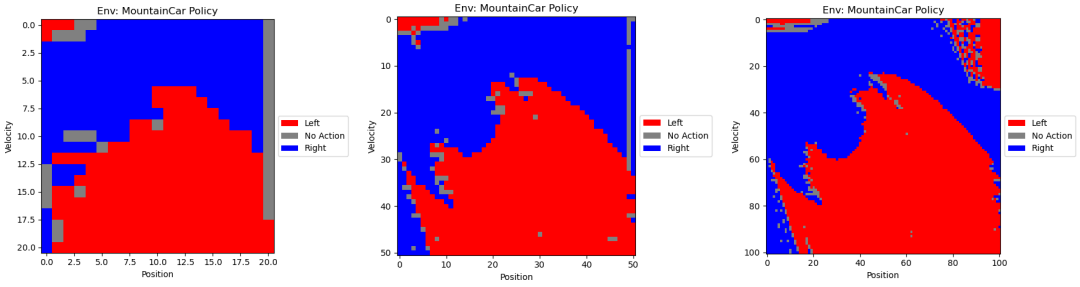
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 14: Policy heatmaps for MountainCar using Deterministic PI with linear interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 15: State value heatmaps for MountainCar using Stochastic PI with linear interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 16: Policy heatmaps for MountainCar using Stochastic PI with linear interpolation

2.3 Stochastic Policy Iteration (GRAD Only)

Grad students, implement stochastic policy iteration for the Mountian Car problem. This should be a small update to your existing policy iteration implementation.

a) You should not be considered if your stochastic policy iteration does not work as well as your deterministic implementation, this is expected. Why might this be? Explain your hypothesis...

Answer: I think stochastic policy iteration might not do as well as deterministic policy iteration because the whole random action steps creates random solutions which are not optimal always. Like, with stochastic policies, you're picking actions based on probabilities, so sometimes you end up trying stuff that's not the best move, which could mean taking longer to get to the good solution. It's is completely random, and that randomness might make it explore paths that aren't super helpful(like difficulty to find the global minima). Meanwhile, deterministic policy iteration just looks at the value function and picks the action that's obviously the best right then and there—it's super straightforward and keeps improving step by step without messing around.

For something like Gridworld, where the state space is discrete and everything's laid out in a neat grid, deterministic policies feel like they'd work better. You've got a set number of spots, and you can just figure out the value for each one and always go with the top choice. It's like following a treasure map where you know exactly where the X's are, so being greedy makes sense and gets you there fast. But with stochastic policies in that same setup, the randomness might make it overcomplicate things—like, why roll the dice when you can just see the best path?

Now, in a continuous space like MountainCar, where position and velocity can be any value and we're splitting it into bins, it's a different scenario. Deterministic policies still try to pick the best action based on those bins, but since it's an approximation, they might miss some details between the lines. Stochastic policies, with their random vibe, could maybe stumble onto something better by not always sticking to one choice—they're more chill about exploring. But even then, I'd guess they could underperform because that randomness might keep them from locking in on the optimal policy as fast as deterministic ones do. So, my hypothesis is that stochastic policy iteration struggles more because the randomness slows it down, especially in discrete spaces where you don't need to guess, and even in continuous spaces, it might not focus as well as deterministic's straight-to-the-point approach.

b) The idea of a stationary policy is more difficult to determine in the stochastic case. We have provided an implementation of KL-divergence for you in the codebase, you may use this to determine if your policy is approximately stationary. What is KL-divergence, and why might it be a good way of determining if your policy is stable? If you chose to use something other than KL-divergence, briefly explain...

Answer: So, KL-divergence—or Kullback-Leibler divergence, is this process that tells you how different two probability distributions are from each other. In this homework, it's like a way to check how much the policy changes between iterations when you're doing stochastic policy iteration. If the KL-divergence is super low, it means the policy isn't shifting/changing around much, which kinda hints that it's settling down and getting stable—or, like, "approximately stationary," as they call it. I think it's pretty handy for stochastic policies because those are all about probabilities for actions, not just picking one thing every time like deterministic policies do. So, KL-divergence helps you see if those probabilities are staying consistent, which matters a lot when your actions aren't set in stone.

For something discrete like Gridworld, where the states are all fixed and you've got a clear grid, KL-divergence could still work to check stability. Since stochastic policies give you a spread of

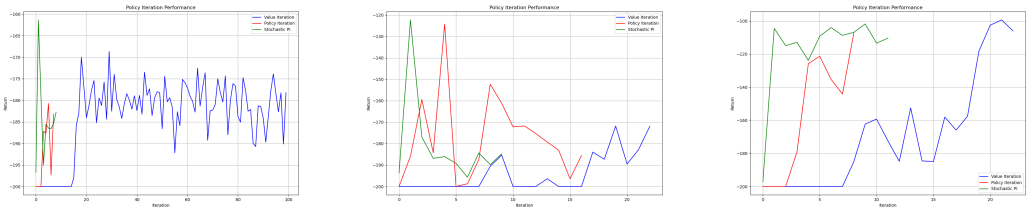
possible actions at each spot, you can use it to see if that spread stops changing much—like, are you still flipping between options a lot, or are you chilling with the same odds? In a continuous space like MountainCar, though, it feels even more useful. The state space is all over the place with position and velocity, and we're just approximating it with bins, so the policy's probabilities might bounce around more. KL-divergence can tell you if those probabilities are finally lining up iteration after iteration, even with all that no order, because it's good at comparing how similar those distributions are.

I didn't use anything else instead of KL-divergence since it's already in the code they gave us, and it seems to fit the job. It's all about measuring how close two sets of probabilities are, which makes sense for stochastic stuff where everything's a chance game. If I had to pick something different, something like the average reward, but that wouldn't really tell me if the policy itself is stable—just how well it's doing. So, yeah, KL-divergence looks straightforward as a right tool here because it's laser-focused on the probability shifts, which is what you need to figure out if a stochastic policy's is working. On another note, the original diffiusion model paper on image generation performs denoising step which could be used to determine the stability of a stochastic policy. It works by iteratively refining a noisy version of the data until it converges to a stable distribution. This process is similar to how KL-divergence measures the difference between two distributions, as both methods aim to find a stable representation of the data. However, KL-divergence is more direct in quantifying the difference between two distributions, while diffusion models focus on the iterative refinement process. I am not sure exactly how but there are some implementation of the KL divergence also a part of the newer/improvements of the diffusion models. So, I think it could be used to determine the stability of a stochastic policy.

Report learning curves for both interpolation methods:

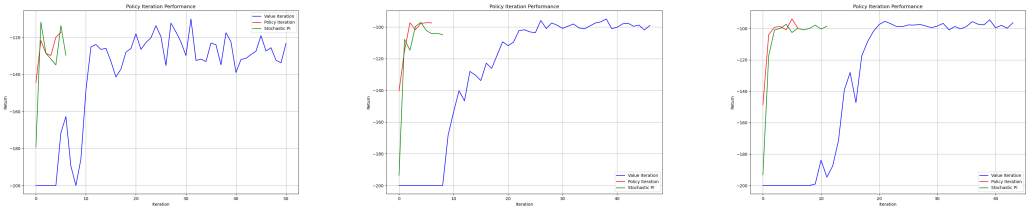
2.3.1 Nearest-Neighbor Interpolation

2.3.2 Linear Interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 17: Learning curves for MountainCar with nearest-neighbor interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 18: Learning curves for MountainCar with linear interpolation