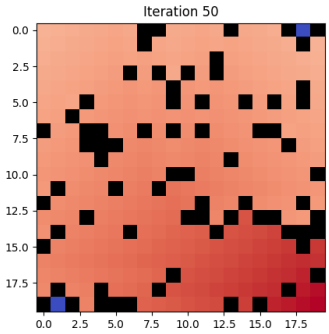
Homework #1: Value Iteration, Discretization

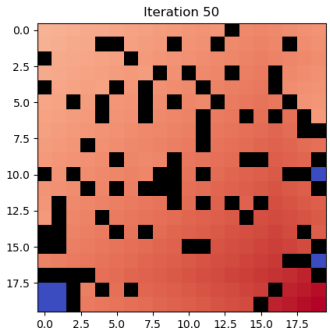
Deliverable: PDF write-up by **Wednesday September 18th, 23:59pm**. Your PDF should be generated by simply replacing the placeholder images of this LaTeX document with the appropriate solution images that will be generated automatically when solving each question. Your PDF is to be submitted into www.gradescope.com. Look at the `README.md` file for running each question and modifying the arguments. This PDF already contains a few solution images. These images will allow you to check your own solution to ensure correctness. If any parameters are not specified explicitly, for reporting purposes make sure to use the default settings in the environments.

1. Value Iteration

(a) **[15pt] Value iteration.** First, you will implement the value iteration algorithm for the tabular case. You will need to fill the code in `part1/tabular_value_iteration.py` below the lines `if self.policy_type == 'deterministic'`. Run part 1's run script and report the heatmap of the converged values.



(a) Heatmap of the final values on the `GridWorld(0)` environment.



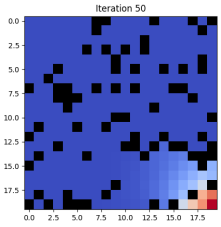
(b) Heatmap of the final values on the `GridWorld(1)` environment. My Solutions

(b) **[15pth] Maxent value iteration.** In cases where the environment might change, a deterministic policy would mostly likely fail. In this scenarios a stochastic policy results more robust. This stochastic policy can be obtained by solving the maximum entropy value iteration. You will need to fill the code in `part1/tabular_value_iteration.py` below the lines `if self.policy_type == 'max_ent'`. Run part 1's run script with the maximum entropy policy and report the heatmap of the converged values for the temperature values 1, 1e-2, 1e-5. In order to stabilize the training: 1) Make sure to add `self.eps` to the policy's probabilities, and 2) compute the exponentiated values after first subtracting the maximum value across all actions for each state (a standard way to stabilize softmax calculations):

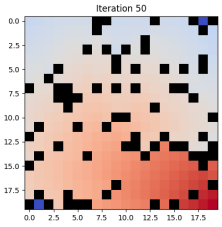
$$\pi_k(a|s) = \frac{1}{Z}\left(\exp\left(\frac{1}{\beta}(Q(s,a) - \max_a Q(s,a))\right) + \epsilon\right)$$

$$V_k(s) = \beta \log\left(\sum_a \exp\left(\frac{1}{\beta}(Q(s,a) - \max_a Q(s,a))\right)\right) + \max_a Q(s,a)$$

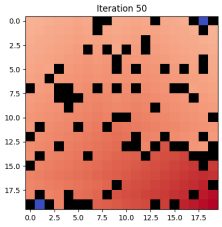
The ϵ value prevents the probabilities to be 0, or close to it, stabilizing the $\log(\pi_k(a|s))$ term of the entropy. Subtracting $\max_a Q(s,a)$ forces all the values in the exponential to be negative preventing large values on the exponential term which would raise numerical problems. Make sure to compute the normalization constant after doing this processing so the probabilities add up to 1.



(a) Heatmap of the final values on the `GridWorld(0)` environment using a maximum entropy policy with temperature equal to 1.



(b) Heatmap of the final values on the `GridWorld(0)` environment using a maximum entropy policy with temperature equal to 1e-2.

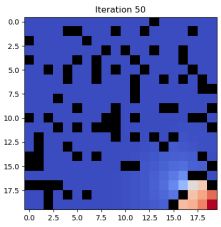


(c) Heatmap of the final values on the `GridWorld(0)` environment using a maximum entropy policy with temperature equal to 1e-5.

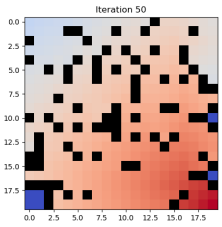
I want to try with naive formula without subtracting $\max_a Q(s,a)$ or adding ϵ

$$\pi_k(a|s) = \frac{1}{Z}\exp\left(\frac{1}{\beta}Q(s,a)\right)$$

$$V_k(s) = \beta \log\left(\sum_a \exp\left(\frac{1}{\beta}Q_k(s,a)\right)\right)$$



(a) Heatmap of the final values on the `GridWorld(1)` environment using a maximum entropy policy with temperature equal to 1. My Solutions



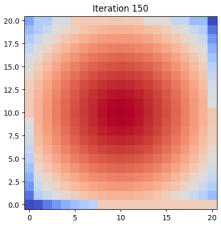
(b) Heatmap of the final values on the `GridWorld(1)` environment using a maximum entropy policy with temperature equal to 1e-2. My Solutions

Placeholder

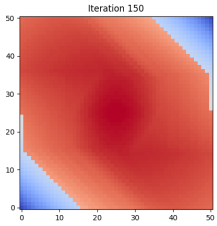
(c) Heatmap of the final values on the `GridWorld(1)` environment using a maximum entropy policy with temperature equal to 1e-5. Error because nan in solutions, which is due to unstable in exp

2. Discretization

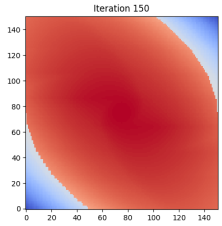
(a) **[10pt] Nearest-neighbor interpolation.** The value iteration algorithm that you implemented in the last part is just valid when the state space and the action space is discrete. What do we do when we are posed with a problem that has the state space and/or the action space continuous. One solution is to discretize such spaces so the previous algorithm is still valid. In this question, you are asked to implement the more naive discretization scheme: nearest-neighbor interpolation. You will need to fill the code in `part2/discretize.py` below the lines `if self.mode == 'nn'`. Run part 2's run script and report the heatmap for `MountainCar` discretizing each dimension of the state space into 21, 51, and 151 bins.



(a) Heatmap of the values after 150 iterations of the value iteration algorithm on the `DoubleIntegrator` environment. The state space is discretized in 21 points and the action space into 5 points using the nearest-neighbor interpolation.



(b) Heatmap of the values after 150 iterations of the value iteration algorithm on the `DoubleIntegrator` environment. The state space is discretized in 51 points and the action space into 5 points using the nearest-neighbor interpolation.



(c) Heatmap of the values after 150 iterations of the value iteration algorithm on the `DoubleIntegrator` environment. The state space is discretized in 151 points and the action space into 5 points using the nearest-neighbor interpolation.

Placeholder

(d) Heatmap of the values after 150 iterations of the value iteration algorithm on the `MountainCar` environment. The state space is discretized in 21 points using the nearest-neighbor interpolation.

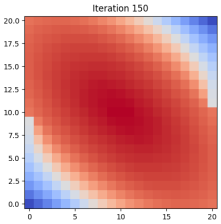
Placeholder

(e) Heatmap of the values after 150 iterations of the value iteration algorithm on the `MountainCar` environment. The state space is discretized in 51 points using the nearest-neighbor interpolation.

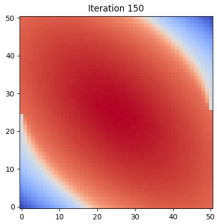
Placeholder

(f) Heatmap of the values after 150 iterations of the value iteration algorithm on the `MountainCar` environment. The state space is discretized in 151 points using the nearest-neighbor interpolation.

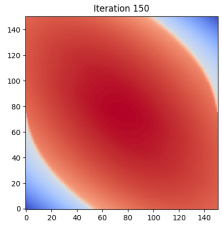
(b) **[10pt] n-linear interpolation.** Discretization using nearest-neighbors is able to recover the optimal solution if your discretization is fine enough. However, the number of points increase exponentially with the dimensionality of your problem. The naive scheme of nearest-neighbors is not computationally tractable for higher dimensional problems. In this question, you will implement a better discretization scheme: n-linear interpolation, the analogous of linear interpolation in n dimensions. To do so, you will need to fill the code in `part2/discretize.py` below the lines `if self.mode == 'linear'`. Run part 2's run script and report the heatmap for `MountainCar` for different discretization resolution: 21, 51, and 151 points per dimension.



(a) Heatmap of the values after 150 iterations of the value iteration algorithm on the `DoubleIntegrator` environment. The state space is discretized in 21 points and the action space into 5 points using the n-linear interpolation.



(b) Heatmap of the values after 150 iterations of the value iteration algorithm on the `DoubleIntegrator` environment. The state space is discretized in 51 points and the action space into 5 points using the n-linear interpolation.



(c) Heatmap of the values after 150 iterations of the value iteration algorithm on the `DoubleIntegrator` environment. The state space is discretized in 151 points and the action space into 5 points using the n-linear interpolation.

Placeholder

(d) Heatmap of the values after 150 iterations of the value iteration algorithm on the `MountainCar` environment. The state space is discretized in 21 points using the n-linear interpolation.

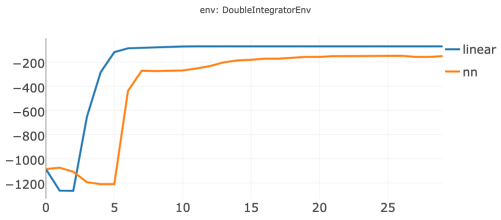
Placeholder

(e) Heatmap of the values after 150 iterations of the value iteration algorithm on the `MountainCar` environment. The state space is discretized in 51 points using the n-linear interpolation.

Placeholder

(f) Heatmap of the values after 150 iterations of the value iteration algorithm on the `MountainCar` environment. The state space is discretized in 151 points using the n-linear interpolation.

(c) **[10pt] Nearest-neighbor vs. n-linear interpolation.** Here we will properly compare the performance of both discretization schemes. Specifically, we will compare the average return across iterations for both methods using the same number of points, 21 in this question. You will report the learning curve for the environments `MountainCar`, `CartPole`, and `SwingUp`. As in the example, create different plots for each environment and split the learning curves by discretization scheme. See the `README.md` file for how to do that.



(a) Learning curve of nearest-neighbor and n-linear interpolation in the `DoubleIntegrator` environment. The state space is discretized in 151 points and the action space using 5 points.

Placeholder

(b) Learning curve of nearest-neighbor and n-linear interpolation in the `MountainCar` environment. The state space is discretized in 151 points.

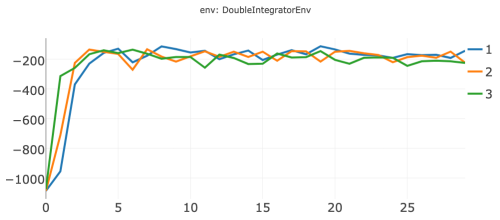
Placeholder

(c) Learning curve of nearest-neighbor and n-linear interpolation in the `CartPole` environment. The state space is discretized in 21 points.

Placeholder

(d) Learning curve of nearest-neighbor and n-linear interpolation in the `SwingUp` environment. The state space is discretized in 21 points.

(d) **[15pt] Look-ahead policies.** Value iteration computes the optimal returns for every position in the state space, however it is often the case that we will just need a good policy in a smaller region of the state space. In such case, running value iteration until convergence might seem a waste of resources. Instead, what we can do is to is to run value iteration with a coarser discretization or not until convergence, and then compensate for that by using look-ahead. In look-ahead we optimize over the first k actions, where we optimize for the sum of near-term reward plus the value achieved in the state achieved after k steps. In this case, the value achieved will be approximated by the discretization. After having found the sequence of k actions, the first one is executed, and then the process is repeated. In this question, we ask you to implement a look-ahead policy. To do so, you will need to fill the code in `part2/look_ahead_policy.py`. Report the learning curves for the values of look ahead horizon equals to 1, 2, and 3 for the `MountainCar`, `CartPole`, and `SwingUp` environment. The results should be split by environment.



(a) Learning curve for different look-ahead horizons in the `DoubleIntegrator` environment.

Placeholder

(b) Learning curve for different look-ahead horizons in the `MountainCar` environment.

Placeholder

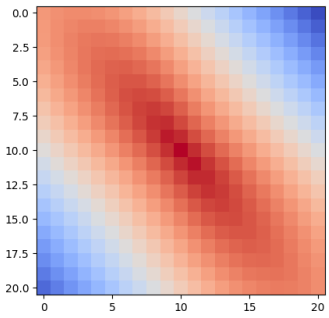
(c) Learning curve for different look-ahead horizons in the `CartPole` environment.

Placeholder

(d) Learning curve for different look-ahead horizons in the `SwingUp` environment.

3. Value Iteration & Function Approximation

(a) **[15pt] Value iteration for continuous state spaces.** In higher dimensional domains discretization its unfeasible, since you end up with an exponential number of points. For instance, consider an environment with 10 state dimensions and 5 action dimensions; if we were to discretize each dimension in just 10 points our transition matrix wold ocupy over 1 million GB!! A solution to this problem is to use function approximators: a function parametrize with some set of parameters θ that maps from states to values. In this excersise we will use a neural network as function approximator (but you can treat it as a black box function). Here, we ask you to implement the value iteration algorithm for function approximators. You will need to fill the code in `part3/continuous_value_iteration.py` and `part3/look_ahead_policy.py`. You will implement a policy that will choose actions by maximizing the value function over a random sample of possible actions. You will need to do it for continuous and discrete actions. Run part 3's run script and report the heatmap for `MountainCar`.

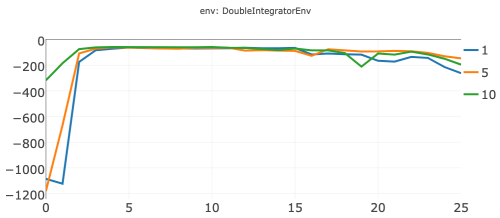


(a) Double Integrator

Placeholder

(b) Double Integrator

(b) **[10pt] Look-ahead with cross-entropy for continuous and discrete action spaces.** A better version of the policy implemented before is using look-ahead with cross-entropy method, specially in larger action spaces. You will need to fill the code in `part3/look_ahead_policy.py`. Report the learning curves for look ahead horizon equals to 1, 5, and 10 for all the environments using the cross-entropy method and horizon equals 1 with random shooting. The results should be split by environment.



(a) Learning curve for different horizons of look ahead on the `DoubleIntegrator` environment using cross-entropy method for action selection with value function approximation.

Placeholder

(b) Learning curve for different horizons of look ahead on the `MountainCar` environment using cross-entropy method for action selection with value function approximation.

Placeholder

(c) Learning curve for different horizons of look ahead on the `CartPole` environment. using cross-entropy method for action selection with value function approximation.

Placeholder

(d) Learning curve for different horizons of look ahead on the `SwingUp` environment. using cross-entropy method for action selection with value function approximation.

4. [20pt] Extra Credit: Vectorized Discretization

As you might have experienced in the `CartPole` or `SwingUp` environments, discretization can be pretty slow and just discretizing 21 points might not be enough to achieve learning even with n-linear interpolation (as in the case of `SwingUp`). The current implementation uses `for` loops to compute the transition and reward matrix and the discretization the discretization. As extra credit you are asked to implement a vectorized version of part 2.a and achieve maximum performance on `CartPole` and `SwingUp` using n-linear interpolation and dicretizing each dimension of the state space in 51 points.

(a) Learning curve for the `CartPole` environment.

(b) Learning curve on the `SwingUp` environment.