INFO8003-1: Reinforcement Learning in a continuous domain

†**Guillaume Delporte**[1] and †**Andréas Coco**[2]

[1] *guillaume.delporte@student.uliege.be (s191981)*
[2] *andreas.coco@student.uliege.be (s2302246)*

† These authors contributed equally to this work.

Course taught by [1]**Damien Ernst**

[1] *dernst@uliege.be*

CONTENTS

I. SECTION 1: IMPLEMENTATION OF THE DOMAIN

For the first part of the project, we had to implement the components of the domain that was provided in the instructions, exploiting the Euler integration method. This domain then had to be used to simulate the results obtained from a rule-based policy. We decided to simulate a policy that dynamically adjusts the direction of an agent's movement based on its current speed, specifically to reverse direction when the speed is nearly zero. This behavior is encapsulated in the `MomentumAgent` class of our implementation. The policy is given by the following rule:

$$u_{t+1} = \begin{cases} -u_t & \text{if } |s_t| \leq 0.01 \\ u_t & \text{otherwise} \end{cases}$$

Where u_t represents the action taken by the agent at time t. The first action u_0 is set to 4 by default and switches to -4 whenever the agent's speed (s) is close to 0, within a tolerance of 0.01.

The obtained trajectory is displayed in TABLE I. In this table, the first 10 steps and the last 10 steps before the terminal state is reached are reported. As will be further illustrated in the next sections, this policy is not optimal. In this simulated example, the car luckily arrived on top of the hill but this will not always be the case with this policy.

TABLE I: Detailed Trajectory Steps with Actions, Rewards, and States, Here, only the first ten steps and the last steps before the terminal state is reached are reported. States are displayed as (p, s)

Step	State	Action	Reward	Next State
1	(-0.0072, 0)	4	0	(-0.0214, -0.2883)
2	(-0.0214, -0.2883)	4	0	(-0.0646, -0.5793)
3	(-0.0646, -0.5793)	4	0	(-0.1369, -0.8660)
4	(-0.1369, -0.8660)	4	0	(-0.2362, -1.1098)
5	(-0.2362, -1.1098)	4	0	(-0.3538, -1.2076)
6	(-0.3538, -1.2076)	4	0	(-0.4684, -1.0322)
7	(-0.4684, -1.0322)	4	0	(-0.5521, -0.6055)
8	(-0.5521, -0.6055)	4	0	(-0.5865, -0.0693)
9	(-0.5865, -0.0693)	4	0	(-0.5661, 0.4771)
10	(-0.5661, 0.4771)	4	0	(-0.4940, 0.9470)
...	...	...	...	...
147	(-0.2594, 2.0842)	4	0	(-0.0741, 1.5998)
148	(-0.0741, 1.5998)	4	0	(0.0642, 1.2269)
149	(0.0642, 1.2269)	4	0	(0.1778, 1.0562)
150	(0.1778, 1.0562)	4	0	(0.2777, 0.9513)
151	(0.2777, 0.9513)	4	0	(0.3705, 0.9176)
152	(0.3705, 0.9176)	4	0	(0.4641, 0.9695)
153	(0.4641, 0.9695)	4	0	(0.5672, 1.1080)
154	(0.5672, 1.1080)	4	0	(0.6880, 1.3232)
155	(0.6880, 1.3232)	4	0	(0.8336, 1.5993)
156	(0.8336, 1.5993)	4	1	(1.0091, 1.9194)

II. SECTION 2: EXPECTED RETURN OF A POLICY

The next part of this project consists in estimating the expected return of the policy implemented in section 1 using Monte Carlo simulation. To achieve this, we sampled 50 initial states from the initial state distribution, which is given by Equation 1. Then, as we have access to the domain's dynamics, simulation is easy. For each of the initial states sampled, we simulate a trajectory of length $N = 300$ and computed the total discounted reward obtained over that trajectory. The maximum trajectory length was set to 300 because in the problem at hand, the only non-zero rewards that can be obtained are 1 and -1, when a terminal state is reached. This implies that when simulating the i^{th} step of the trajectory, the absolute change in the estimation of the expected return is bounded by γ^i, where $\gamma = 0.95$ in our domain. Moreover, as $0.95^{300} \approx 1e^{-7}$ is negligible, considering larger N values would not significantly impact the estimated expected return.

$$p_0 \sim \mathcal{U}([-0.1, 0.1]), s_0 = 0 \tag{1}$$

FIG. 1 displays how the expected return estimates evolve as N increases. The plot clearly shows that the policy we designed has a negative expected return. The number of time steps needed to reach a terminal state obviously depends on the starting state. It is not clear from the plot but 4 of the 50 trajectories reached a winning terminal state within 300 steps while 35 of them ended in a losing state.

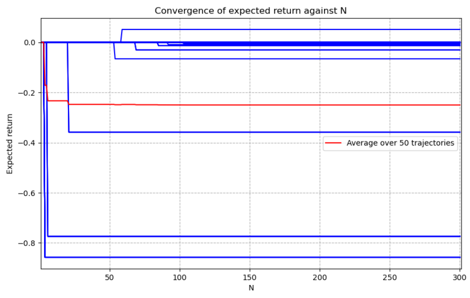


FIG. 1: Evolution of the expected return estimates as N increases. The blue lines represent the expected return for a given initial state while the red line is the average expected return over the 50 initial states.

III. SECTION 3: VISUALIZATION

In the third section, our aim was to produce a GIF illustrating the results of the policy we defined in Section 1 when starting at the initial state $p_0 = s_0 = 0$. The resulting GIF file can be found in our deliverable. We decided to use PIL as our GIF creation library.

We used a GIF speed of 10 frames per second. The trajectory obtained by the `MomentumAgent` when starting at $p_0 = s_0 = 0$ is 53 steps long. Consequently, the resulting GIF lasts 5.3 seconds.

IV. SECTION 4: FITTED-Q-ITERATION

The fourth step of the project consisted in estimating $\hat{Q}_N$ for $N = 1, 2, 3, ...$ using Fitted-Q-Iteration. Three supervised learning techniques had to be used, namely linear regression, extremely randomized trees and neural networks.

The linear regression model we used was the `LinearRegression` from scikit-learn with the default parameters. Regarding the extremely randomized trees, the hyperparameters were chosen to be the same as those introduced in [1] as they were proven to lead to a winning policy. That is, we chose the number of trees estimated to be $M = 50$ as it is sufficiently big to make sure the models' accuracy could not be improved significantly by increasing M. The number of candidate tests at each node $K = 3$ was set equal to the input dimension (as it is a good default practice) and we set the minimal leaf size to $n_{min} = 2$. The neural network architecture we designed consists in:

1. Layer Composition: The network starts with an input layer that matches the `input_size = 3`, followed by layers that progressively increase the number of neurons from 8 to 64 and then decrease back to the `output_size = 1`. This sequence of layers is 8, 16, 32, 64, 64, 32, 16, 8, and finally to the output layer.

2. Activation Function: Between each layer, a Tanh (hyperbolic tangent) activation function is applied. Unlike ReLU and sigmoid, Tanh was chosen empirically as it is the one that gave the most best result even if, as will be observed, the results are not as good as with the trees algorithm.

Furthermore, we had to create two strategies to generate the set of one-step system transitions used by the algorithms. The first method we developed was strongly inspired from the one described in [1]. We generated the transitions by simulating 1,000 episodes which we all stopped once a terminal state was reached. These episodes were simulated using a random agent (i.e., at every step, the action was chosen with equal probability between $u = -4$ and $u = 4$). The only difference from the paper's approach is that we drew the starting state uniformly like in the domain description to comply to the 'realistic' setting. This provided us with a set of 59,530 one-step system transitions. Among the 1000 episodes, only 16 of them ended in a winning state whereas the 984 others reached a losing terminal state. The second technique we used to generate transitions consists in sampling uniformly from the state space (only non-terminal states), using $p \sim \mathcal{U}([-1, 1]), s \sim \mathcal{U}([-3, 3])$ and simulating an action at random from that state. We decided to simulate 100,000 transitions using this technique in order to cover the state-action space as well as possible and to have enough transitions to train our models. This simulation method is not realistic as in regular experiments it is not possible to just sample from the state space and take an action from there. However, we still used this as we thought that having a sample that covered more of the state-action space would help our estimation, notably by enabling faster convergence. This is investigated further in the report, when the estimation results are presented. FIG.2 shows that, as expected, the method sampling uniformly from the state space spans the state space very well, which is not the case of the episodic method.

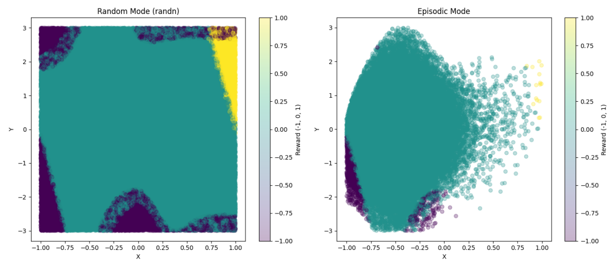


FIG. 2: This plot illustrates how the two transitions generation methods cover the state space. The blue points represent states where the reward obtained is 0, the yellow ones represent states where the reward obtained is 1 and the purple ones represent states where the reward obtained is -1. It can clearly be observed that the left plot shows a dense cloud of points effectively covering the space. This generation technique allows for a more complete exploration of the state space although it is not very realistic. On the right, the plot illustrates the episodic approach to exploration, with points spread out in a less uniform manner. This method is more representative of realistic sequences of state-actions but shows a more sparse and potentially challenging coverage of the state space.

The last aspect of Fitted-Q-Iteration that had to be set was the stopping criterion. The first stopping criterion we used is to just iterate until a fixed N value. We set $N = 50$ for the linear regression and extremely randomized trees as this value was used in [1]. The authors showed that this was sufficiently big to reach convergence, at least with the extremely randomized trees algorithm. However, for the neural network, we decided to use $N = 200$ iterations so that it is better able to learn. The second stopping criterion we used consists in stopping the learning if the mean squared error (MSE) between two consecutive estimations of the Q-function is under $3 * 10^{-4}$, as we thought this threshold was sufficiently low to allow us to stop the learning at that point. We used the MSE because it is the divergence measure used in [1]. If this threshold is not reached by that time, we stop iterating when the N values defined above are reached. The efficiency of these criteria will be further discussed later in the report.

A. Discussion of the results

We now investigate the impact on the results of the supervised learning algorithms, the stopping rules and the one-step generation strategies. To achieve this, the evolution of the MSE over the iterations, the final $\hat{Q}_N$ functions and the final derived policies are respectively illustrated in FIG. 9, FIG. 10 and FIG. 7 in the Appendix.

Concerning the linear regression, FIG. 7a shows that the final policy obtained when using linear regression always consists in taking the same action: decelerating. This was to be expected given the nature of linear regression. The linear equation we obtain is given by Equation 2. Moreover, the resulting policy consists in choosing $\hat{\mu}^*_N(p, s) = \underset{u}{\text{argmax}} \hat{Q}_N(p, s, u)$ in state (p, s). However, the only difference between $\hat{Q}_N(p, s, 4)$ and $\hat{Q}_N(p, s, -4)$ comes from the value of $\hat{\alpha_3}$. In other words, if $\hat{\alpha_3} > 0$ then the policy will consist in always choosing $u = 4$ while if $\hat{\alpha_3} < 0$, the policy will consist in always choosing $u = -4$.

$$\hat{Q}_N(p, s, u) = \hat{\alpha_0} + \hat{\alpha_1} p + \hat{\alpha_2} s + \hat{\alpha_3} u \tag{2}$$

Thus, no other parameters such as the transition generation strategy or the stopping criterion have an influence on the resulting policy and this methods proves to be too simple and inefficient for this type of problem.

Conversely, extra trees, capture much more complex decision boundaries. The $\hat{Q}_N$ plots in FIG. 10 show that the extra trees learn very similar functions, no matter the stopping criterion. Still, the transitions generation strategy appears to make a difference as the MSE threshold is not reached before $N = 50$ iterations. The uniform sampling method leads to a smoother estimation of $\hat{Q}_N$. This is reflected in the derived policies as those obtained from the episodic generation technique seem to know less about some specific regions of the state space.

Even though they provide more meaningful results than linear regressions, the neural networks also seem unable to properly capture the structure of the Q-function. As explained in [1], this surely stems from the fact that the parametric supervised learning methods, in contrast to the non-parametric ones, do not perform well mostly because it is difficult to select the shape of the parametric approximation architecture that could provide good results in advance. The boundaries that can be observed in FIG. 10 are much simpler than those from the trees algorithm. As a result, the policies obtained all differ based on the learning setting. In particular, the policy learned using the episodic generation strategy with early stopping leads to an always-decelerating strategy, similarly to the linear regression. This is not surprising as FIG. 9 shows that the MSE threshold was reached after only 3 iterations. The model clearly did not have time to learn the function effectively.

Overall, the extra trees clearly offer better results than the two other learning algorithms as exhibited in FIG. 3. The plot reveals that the extra trees algorithm is the only one that leads to a positive expected return. The uniform generation strategy with no early stopping appears to lead to an even slightly higher expected return that the other settings but this might be an estimation approximation. Most of the other methods lead to a null expected return while the neural network model combined with the episodic transitions generation and no early stopping gives a strongly negative expected return, showing it led to a poorly-designed policy. The separate expected return plots for all parameters combinations are displayed in FIG. 12 in the Appendix.

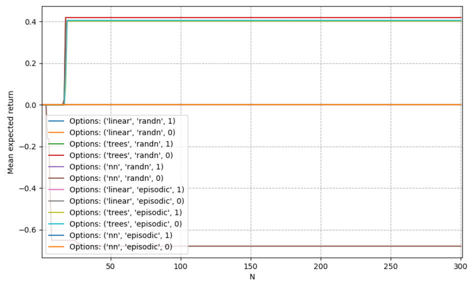


FIG. 3: Expected return for each transitions generation strategy, each learning algorithm and each stopping criterion. The lines were obtained by simulating from 50 initial states and averaging the estimated expected return based on the policy obtained using Fitted-Q-Iteration. The plot shows that the extra trees is the only algorithm leading to a positive expected return, no matter the rest of the setting.

The generation strategies did not make a major difference in the results although as already explained above focusing on the extra trees as the other methods did not yield good results, the uniform strategy led to a more accurate estimation of the Q-function and a policy that was more precise in some areas of the state space. The main difference the stopping criterion caused is with the neural network in the episodic setting. It clearly stopped too early (after only 3 iterations) and led to the same policy as the linear regression, suggesting this stopping criterion was not ideal. Apart from this, the stopping criterion did not make a big difference in the results.

V. PARAMETRIC Q-LEARNING

In the final section of this project, we had to implement the Parametric Q-Learning (PQL) algorithm to estimate the Q-function and compare the results with those obtained by Fitted-Q-Iteration. The approximation architecture that had to be used is a neural network. Rather than implementing basic PQL, we decided to apply some concepts from approximate Q-learning to enhance the performance of our estimation technique. Consequently, we used the structure from the Deep Q-Network (DQN) method [2]. Hence, we first used a target network to solve the non-stationarity issue of Parametric Q-Learning. We decided to update this target network every 1000 epochs. Additionally, we used a replay buffer that aims at tackling the non-iid samples issue of Q-Learning. We used a replay buffer containing $10,000$ transitions. When sampling from it, the probability of a transition (x, u, r, x') to get picked is proportional to the error $(r + \gamma \max_{u' \in U} Q_{\theta'}(x, u') - Q_{\theta_{k-1}}(x, u))^2$, where $Q_{\theta'}$ denotes the target network and $Q_{\theta_{k-1}}$ denotes the current Q-network. As new transitions are added to the buffer, the oldest ones get removed from it. We also used a label named `done`, which is a boolean set to true when we are in a terminal state. We use this label when computing the expected Q-values as because we reached a terminal state, all future reward will be equal to zero and thus the reward for this terminal state is the ground truth. Furthermore, we used mini-batches instead of pure stochastic gradient descent as this decreases the variance of the gradient estimates. However, large batches make the problem computationally expensive. Resultingly, it is common practice to use small batch sizes. We opted for a size of 4. Finally, instead of using random sampling, we implemented an epsilon-greedy policy with a special configuration of epsilon (ϵ) parameters to try and achieve a trade off between exploration and efficacy. The parameters were set as follows: an initial epsilon (ϵ_{start}) of 1, meaning that actions are initially drawn randomly, aiming towards exploration. As learning progresses, our goal is to gradually shift the balance towards exploitation, leveraging the knowledge the agent has accumulated to sample more desirable states that make the learning quicker. To facilitate this transition, we set a minimum epsilon (ϵ_{end}) value of 0.1, indicating that even at its most knowledgeable state, the agent retains a 10% chance to explore the state-space. We also set the decay rate to .8, so the value of epsilon is exponentially decaying but not too fast as the state space is pretty large and needs a bit of time to be explored enough (otherwise our model did not have time to reach a positive terminal state and led to poor results).

The neural network we designed starts with an input layer that matches the input size: 3, followed by layers that progressively increase the number of neurons from 8 to 32 and then decrease back to the output size: 1. The sequence of layer sizes is thus 3, 8, 16, 32, 16, 8 and 1. All these are fully connected layers. We did not use wider/deeper networks because given the high number of transitions considered, updating them would have been too time consuming. We also did not use a simpler network because by trying simpler architectures, we realized they were not able to capture the Q-function structure sufficiently well. Furthermore, a tanh (hyperbolic tangent) activation function is applied between all pairs of layers. Tanh was chosen as we also tried using ReLU and Sigmoid but Tanh was the function giving the best results. We used AdamW as optimizer as Adam is the most commonly-used optimizer and we used the default weight decay of 0.01. After trying several values, we set our learning rate to $3e - 4$ as it was the one providing us with the best results. The loss function we optimize is the mean squared error (`MSELoss`) to remain coherent as it was used to assess convergence in the previous section. Finally, the number of epochs, which in this case corresponds to the number of one-step system transitions generated, was set to $400,000$. We decided to use a high number to try ensuring convergence. The impact of this hyperparameter is analyzed at the end of this section.

Afterwards, we derived the policy $\hat{\mu}_*$ from $\hat{Q}$. The result is displayed in FIG. 4.

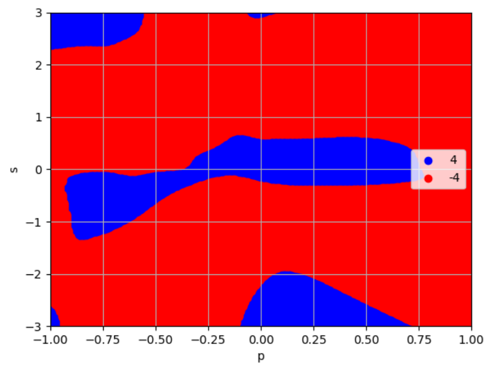


FIG. 4: Estimated Optimal Policy for PQL (Parametric Q Learning). Clearly, the policy captured is different from the one obtained in FQI with extra trees.

The policy is somewhat different from what we obtained using Fitted-Q-Iteration. As a first tool to compare them, we constructed FIG. 5 that shows the expected return of the derived policy $\hat{\mu}_*$. Similarly to what was obtained with FQI, the policy always leads the agent to a winning terminal state. However, this policy never provides a return as high as the one obtained by FQI and the return obtained depends much more on the initial state than it was the case in the previous section. The estimated expected return we get is only 0.29073.

Ultimately, we created a plot showing how the expected return of the policies derived with Fitted-Q-Iteration and Parametric Q-Learning compare as the number of transitions used to estimate them increases. For both algorithms, the transitions were generated by simulating episodes. In order to cover more of the state space and make our algorithms more efficient, the starting state of the episodes was sampled uniformly using $p_0 \sim \mathcal{U}([-1, 1])$ and $s_0 \sim \mathcal{U}([-3, 3])$ even though this can lead to non-valid starting states of the domain. Then, to obtain

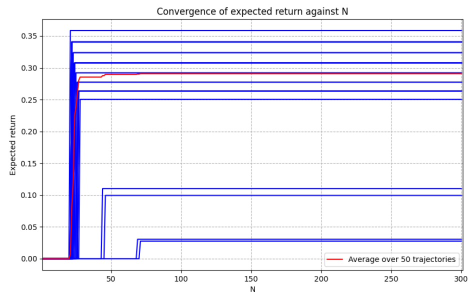


FIG. 5: Expected return of the policy derived from final PQL (Parametric Q Learning) model. This policy underperforms the policy obtained using extremely randomized trees with FQI.

the Fitted-Q-Iteration curve in this plot, we sequentially increased the number of episodes and applied the algorithm. We did this using extra trees and no early stopping as it led to the best results in Section 4. Moreover, it is essential to note that one-step system transitions are note used as many times to train the model in the case of PQL than in the case of FQI. In Parametric Q-Learning, at each epoch, a mini-batch of transitions is used to update the model. That is, effectively, `batch_size*n_transitions` transitions are used to improve the model. On the other hand, in Fitted-Q-Iteration, the entire set of transitions is used every time the supervised learning algorithm is trained. Thus, effectively, `N*n_transitions` transitions are used to improve the model. Consequently, rather than plotting expected return against the actual number of transitions, we decided to plot the expected return of the derived policy against the effective number of transitions used, to allow for a fairer comparison. The results are shown in FIG. 6.

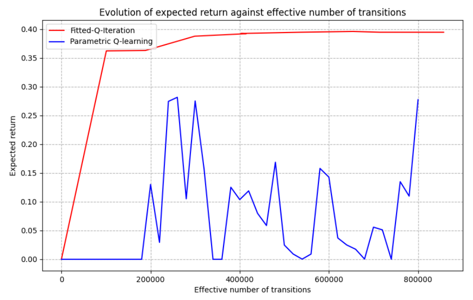


FIG. 6: Expected return comparing PQL (blue) and FQI (red). Expected return of the policy derived from final PQL (Parametric Q Learning) model. This policy underperforms the policy obtained using extremely randomized trees with FQI.

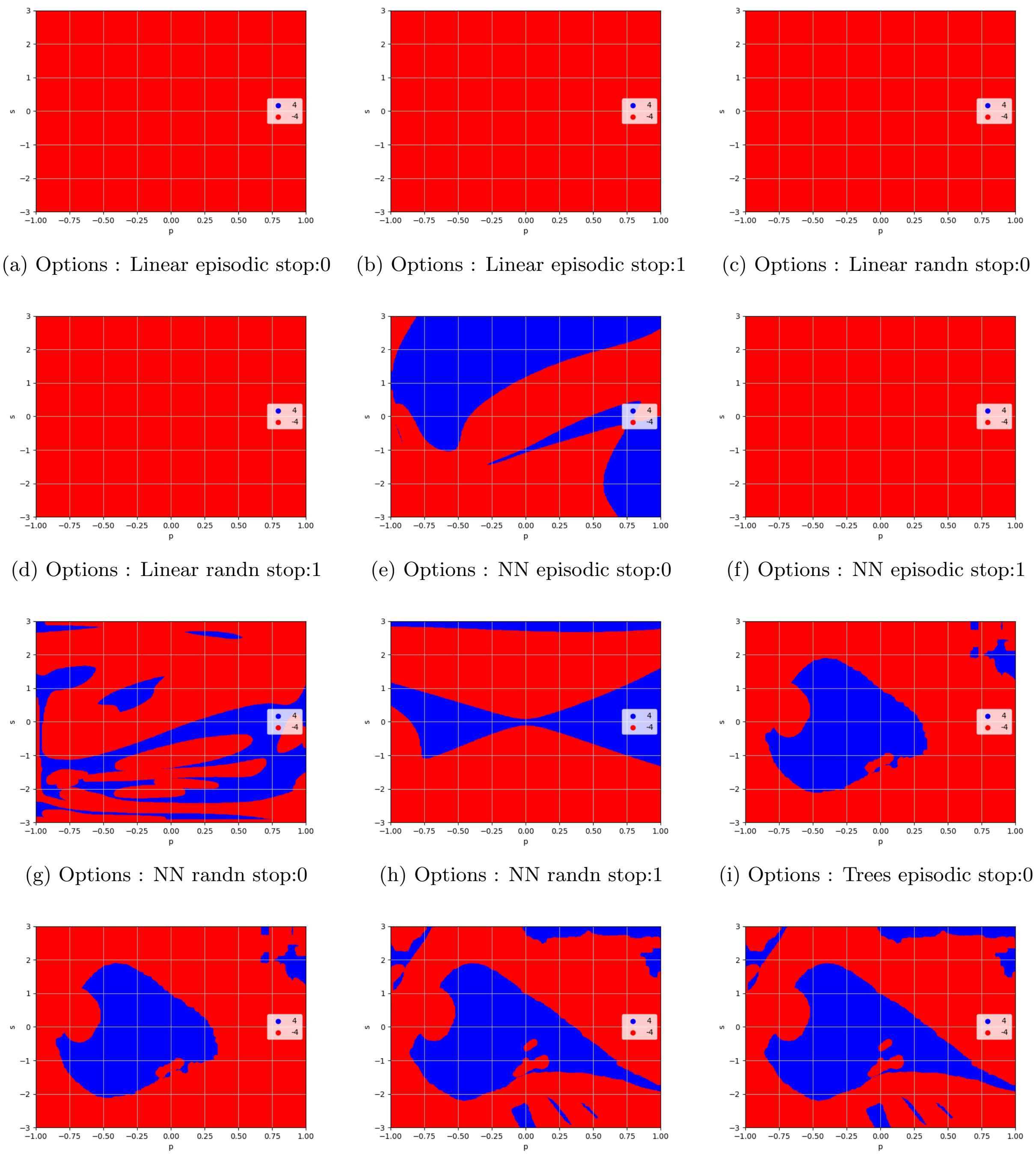
Despite using effective number of transitions used, Fitted-Q-Iteration strongly dominates Parametric-Q-Learning. Fitted-Q-Iteration converges to its best policy quite fast and then stays at that policy. PQL, on the other hand, does not converge and leads to much more volatile results. Also, at no point does it attain the return obtained by FQI. In other words, for the same number of one-step system transitions, Fitted-Q-Iteration evidently outperforms Parametric-Q-Learning. This suggests that for the problem at hand, Fitted-Q-Iteration is much more powerful than Parametric Q-Learning. This might also be because the neural network architecture we used for PQL was not optimal. Still, the main advantage of PQL against FQI is that it uses online-learning and it can improve its policy while simulating, which is good when simulating in real environments and making mistakes is costly.

[1] D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6, 2005.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

APPENDIX

A. Figures



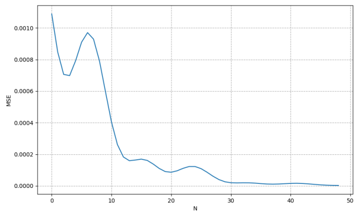
(a) Options : Linear episodic stop:0 (b) Options : Linear episodic stop:1 (c) Options : Linear randn stop:0

(d) Options : Linear randn stop:1 (e) Options : NN episodic stop:0 (f) Options : NN episodic stop:1

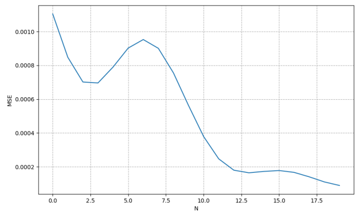
(g) Options : NN randn stop:0 (h) Options : NN randn stop:1 (i) Options : Trees episodic stop:0

(j) Options : Trees episodic stop:1 (k) Options : Trees randn stop:0 (l) Options : Trees randn stop:1

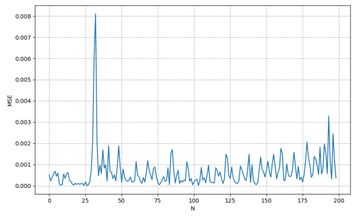
FIG. 7: Plot of the different 'optimal' policies found by our algorithm. Episodic indicates that the 1, 000 episodes were used to generate the transitions set whereas randn indicates that the the uniform sampling method was used to generate the transitions. Stop:0 indicates that we iterated up to horizon N whereas stop:1 indicates that we stopped iterating when the MSE fell below the threshold we defined.



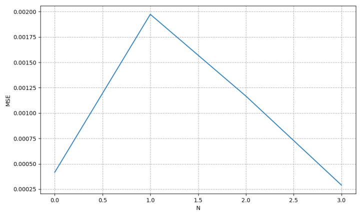
(a) Episodic linear 0



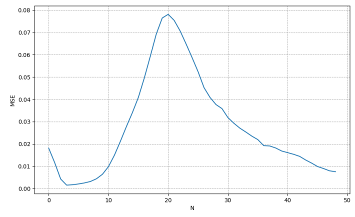
(b) Episodic linear 1



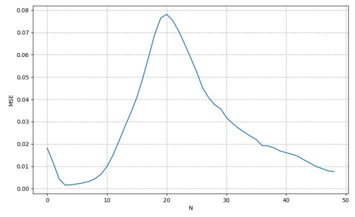
(c) Episodic nn 0



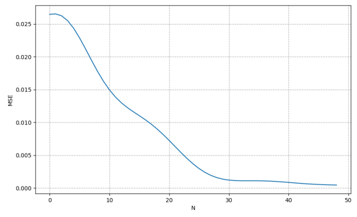
(d) Episodic nn 1



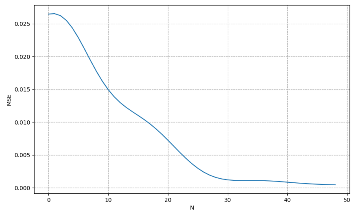
(e) Episodic trees 0



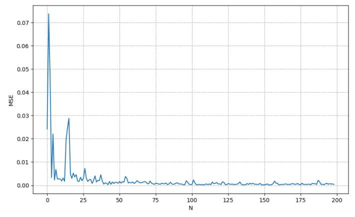
(f) Episodic trees 1



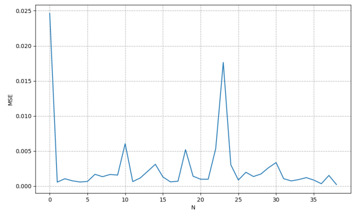
(g) Randn linear 0



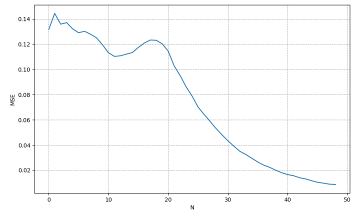
(h) Randn linear 1



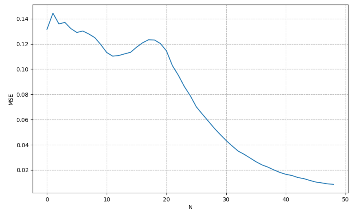
(i) Randn nn 0



(j) Randn nn 1

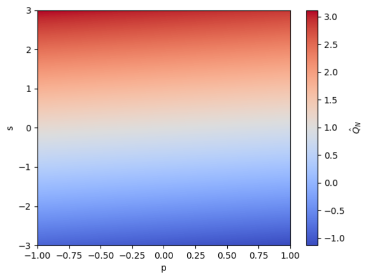


(k) Randn trees 0

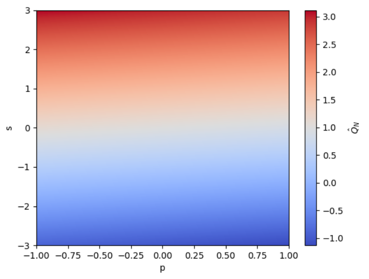


(l) Randn trees 1

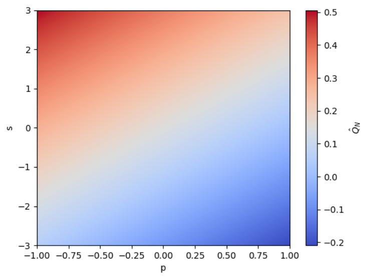
FIG. 8: Mean Squared Error (MSE) for different methods and settings.



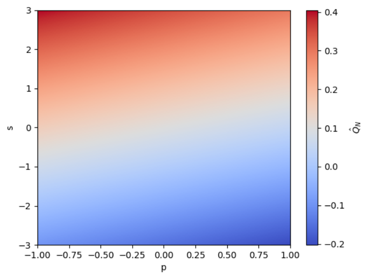
(a) Linear randn stop:0



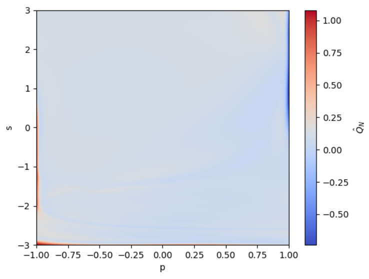
(b) Linear randn stop:1



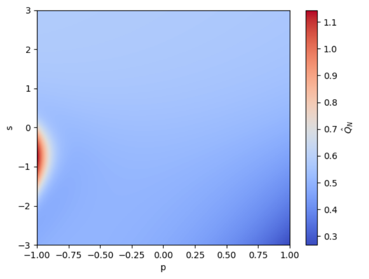
(c) Linear episodic stop:0



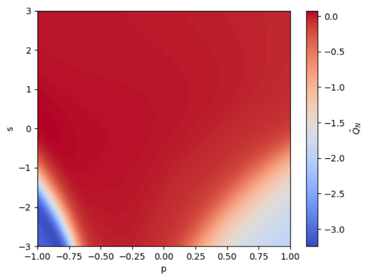
(d) Linear episodic stop:1



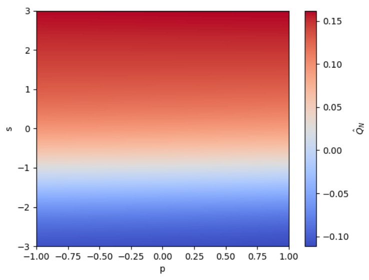
(e) NN randn stop:0



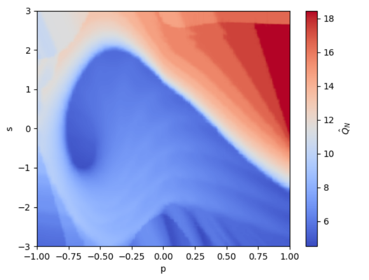
(f) NN randn stop:1



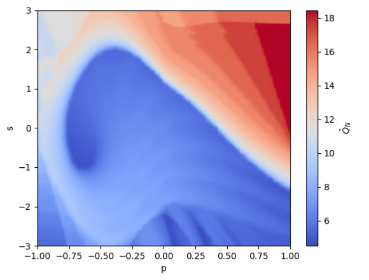
(g) NN episodic stop:0



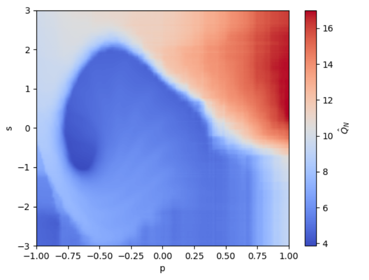
(h) NN episodic stop:1



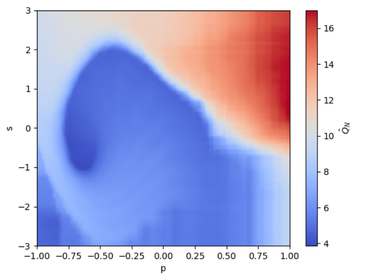
(i) Trees randn stop:0



(j) Trees randn stop:1

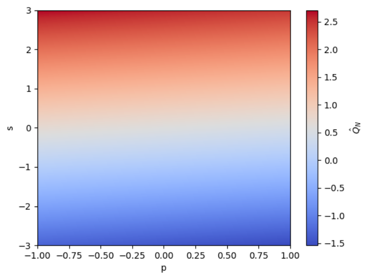


(k) Trees episodic stop:0

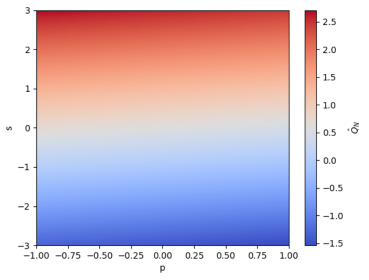


(l) Trees episodic stop:1

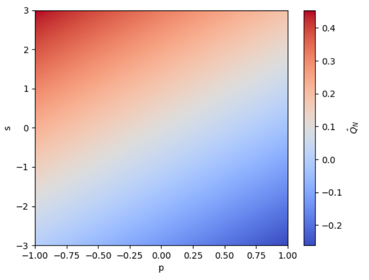
FIG. 9: Plots of the action value function Q_N for action 4 across different models and modes. Episodic indicates that the $1,000$ episodes were used to generate the transitions set whereas randn indicates that the the uniform sampling method was used to generate the transitions. The value 0 indicates that we iterated up to horizon N whereas 1 indicates that we stopped iterating when the MSE fell below the threshold we defined.



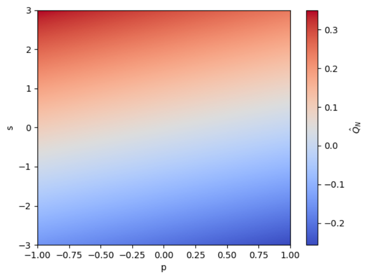
(a) Linear randn stop:0



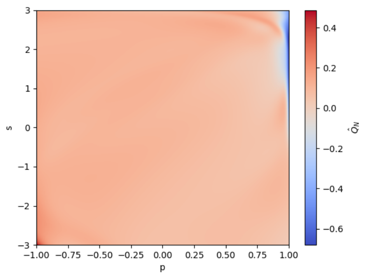
(b) Linear randn stop:1



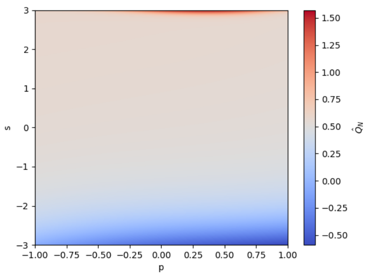
(c) Linear episodic stop:0



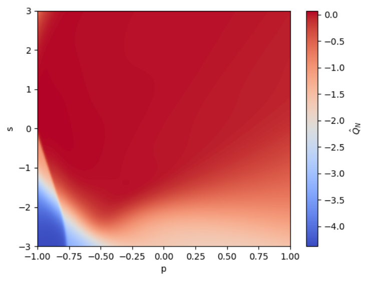
(d) Linear episodic stop:1



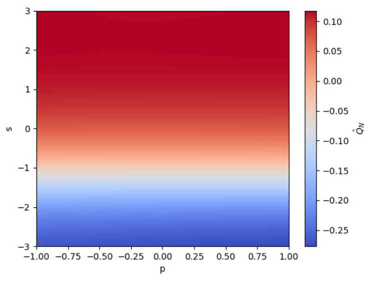
(e) NN randn stop:0



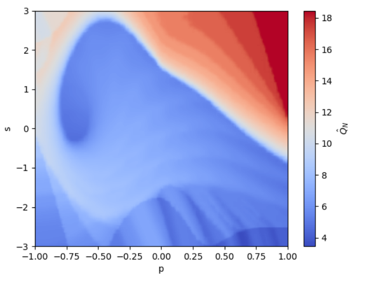
(f) NN randn stop:1



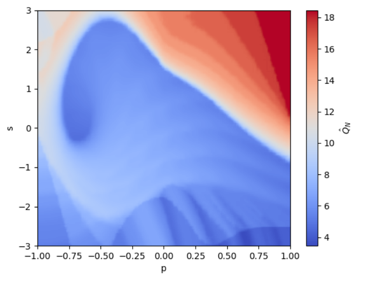
(g) NN episodic stop:0



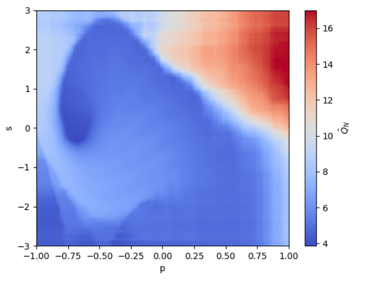
(h) NN episodic stop:1



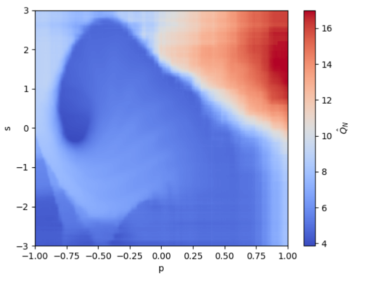
(i) Trees randn stop:0



(j) Trees randn stop:1

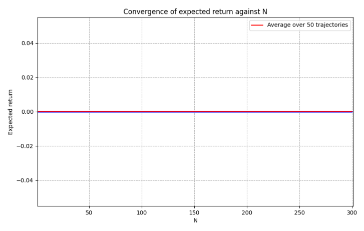


(k) Trees episodic stop:0

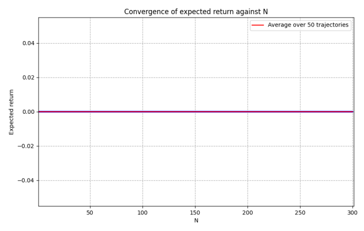


(l) Trees episodic stop:1

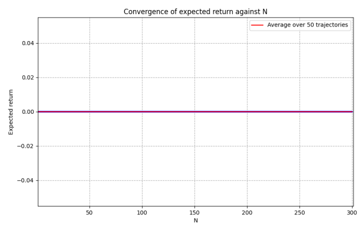
FIG. 10: Plots of the action value function Q_N for action -4 across different models and modes. Episodic indicates that the $1,000$ episodes were used to generate the transitions set whereas randn indicates that the the uniform sampling method was used to generate the transitions. Stop:0 indicates that we iterated up to horizon N whereas stop:1 indicates that we stopped iterating when the MSE fell below the threshold we defined.



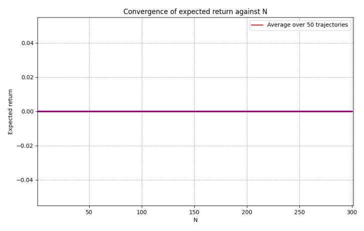
(a) Linear randn stop:0



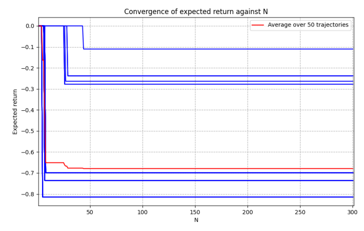
(b) Linear randn stop:1



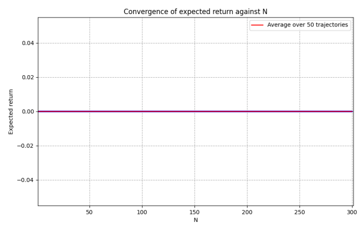
(c) Linear episodic stop:0



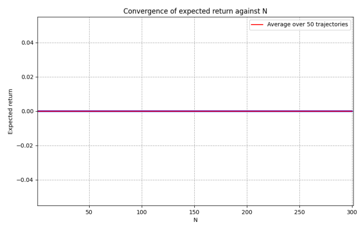
(d) Linear episodic stop:1



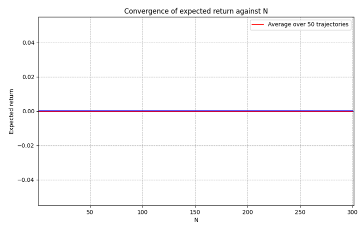
(e) NN randn stop:0



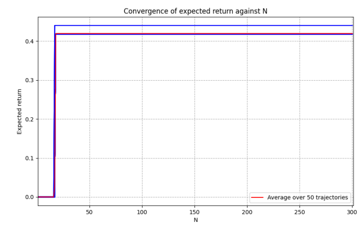
(f) NN randn stop:1



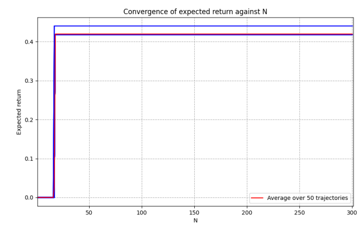
(g) NN episodic stop:0



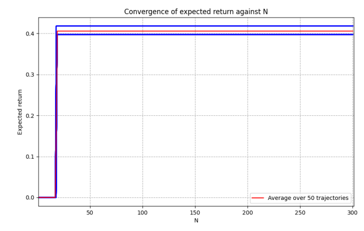
(h) NN episodic stop:1



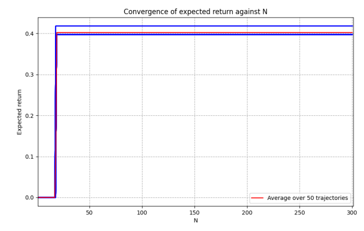
(i) Trees randn stop:0



(j) Trees randn stop:1

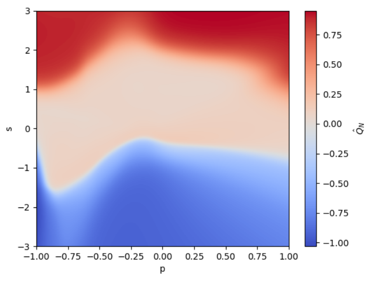


(k) Trees episodic stop:0



(l) Trees episodic stop:1

FIG. 11: Plots of the evolution of expected return across different models and modes. Like in Section 2, 50 initial states were randomly sampled to simulate from the domain using the policy obtained from Fitted-Q-Iteration. The red line represents the average over the 50 initial states. Episodic indicates that the 1,000 episodes were used to generate the transitions set whereas randn indicates that the the uniform sampling method was used to generate the transitions. Stop:0 indicates that we iterated up to horizon N whereas stop:1 indicates that we stopped iterating when the MSE fell below the threshold we defined.


(a) Q function for a=-4

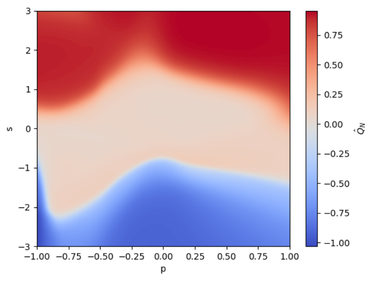

(b) Linear randn stop:1

FIG. 12: Visualization of Q-function values for actions $a = -4$ and $a = 4$ for the final Parametric Q Learning (PQL) model.