UNIVERSITY OF JEAN MONNET

COURSE: ADVANCED MACHINE LEARNING

PRACTICAL SESSION 2

ONLINE LEARNING, BANDITS, REINFORCEMENT LEARNING

Author:
Tsolakidis Dimitrios
Renner Joseph
Javaid Usama

Student Number:
16007165
16007158
xxx

Chapter 1

Online Passive-Aggressive Algorithms

In this chapter we implement the three versions of Passive Aggressive (PA) Online algorithm in the context of linear binary classification. We make use of binary classification datasets from LibSVM website and we compare the PA algorithm with LibSVM toolbox. We also introduce some noise to our dataset, in order to observe the behavior of the three different updates on PA algorithm.

For our experiment we used the splice dataset. This dataset comes originally from the UCI repository of machine learning databases and it consists of 1000 training examples and 2175 testing examples. The task in this dataset is to recognize two types of splice junctions in DNA sequences; exon/intron (EI) or intron/exon (IE) sites. A splice junction is a site in a DNA sequence at which 'superflous' DNA is removed during protein creation. 'Intron' refers to the portion of the sequence spliced out while 'exon' is the part of the sequence retained.

First, we compare the results obtained from PA algorithm with LibSVM. We set the values of C = [0.001, 0.01, 0.1, 1, 5] for the first and second relaxation in PA algorithm and for SVM we used a grid search with 10-fold cross validation on training set in order to tune the C hyper-parameter. We present the results in terms of accuracy and time in table 1.1 and 1.2 respectively. The iterations for PA algorithm are set to 5000.

Table 1.1: Accuracy using Online-PA Algorithm

C	PA-Classic	PA-First Relaxation	PA-Second Relaxation
0.001	0.714	0.642	0.733
0.01	0.714	**0.80**	0.764
0.1	0.714	0.73	0.728
1	0.714	0.714	0.714
5	0.714	0.714	0.714

The PA-classic algorithm does not depend on the C value, unlike the other two versions. Clearly, using the first update strategy yields the best performance. Setting $C \geq 1$ we observe that the performance among the three updates is the same.

Table 1.2: Execution time in seconds

C	PA-Classic	PA-First Relaxation	PA-Second Relaxation
0.001	0.006 s	0.009	0.01
0.01	0.006 s	0.008	0.009
0.1	0.006 s	0.006	0.007
1	0.006 s	0.006	0.009
5	0.006 s	0.006	0.007

Using LibSVM with linear kernel, we obtain an accuracy of 0.732 with execution time 0.063 seconds.

In the next experiment, we study the behavior of Online-PA algorithm, by setting different sizes of training sets and using the three different update rules and we plot the performance for each training set (figure 1.1. The classic PA after 200 examples performs the worst among the other two updates, while using the first and second update rule, both strategies reach the highest accuracy at 0.84 using around 900 examples.

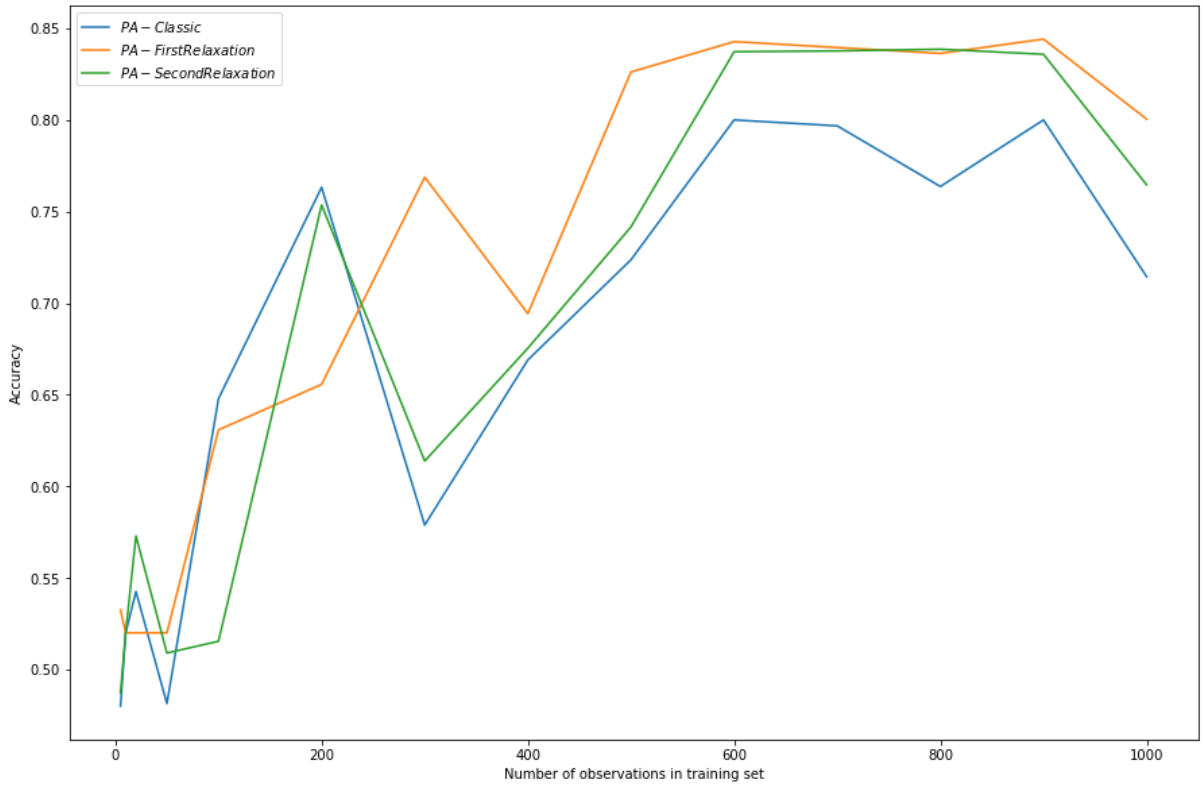


Figure 1.1: PA algorithm behavior

The PA algorithm employs an aggressive update strategy by modifying the weight vector by as much as needed to satisfy the constraint imposed by the current example. In certain real-life situations this strategy may also result in undesirable consequences. Consider for instance the common phenomenon of label noise. A mislabeled example may cause the PA algorithm to drastically change its weight vector in the wrong direction. A single mislabeled example can lead to several prediction mistakes on subsequent rounds. But to cope with this, we can make

use of the two update strategies that take into consideration the C parameter, which controls the influence of the slack term on the objective function.

In the next experiment we flipped some training labels using different percentages [5%, 10%, 20%, 30%, 40%, 50%] of the total number of labels in order to see the influence in the classification accuracy of the different update strategies. For this experiment we set C to 0.01.

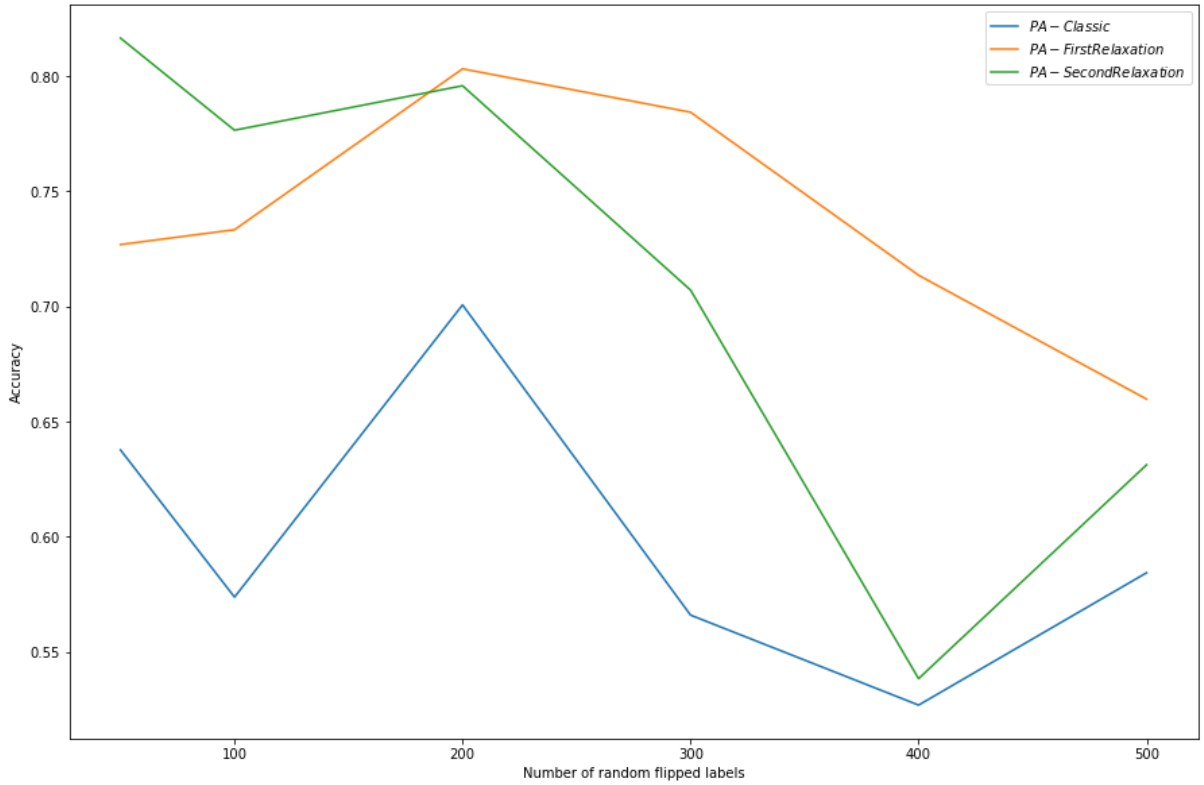


Figure 1.2: Accuracy on three updates of PA algorithm using flipped training labels as noise.

Chapter 2

Bandit Algorithm

We have run 3 bandit algorithms by multi agent bandit. Here is the comparisons on all three algorithms.To have a little insight on algorithms, here are some graphs on choosing the best arm. The average reward increase over the number of episodes.

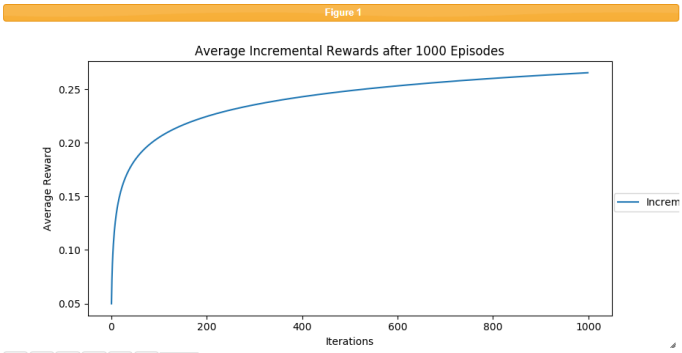


Figure 2.1: Average Incremental Reward

Here is a depiction on UCB. We choose the value of c=2. There are some hiccups but then it

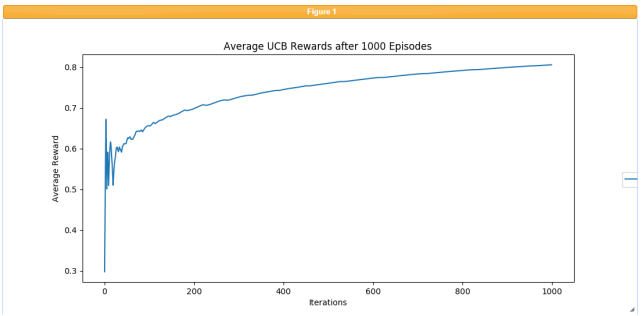


Figure 2.2: UCB with c=2

converged to best arm.

For greedy bandit algorithm, we choose the value 0.1, because in experiments it performed better than 0 and 0.01. By taking value 0.1, our algorithm choose the best arm for 80 percent of time. So we decided to carry on greedy with 0.1 value. Regarding the experimental study, we

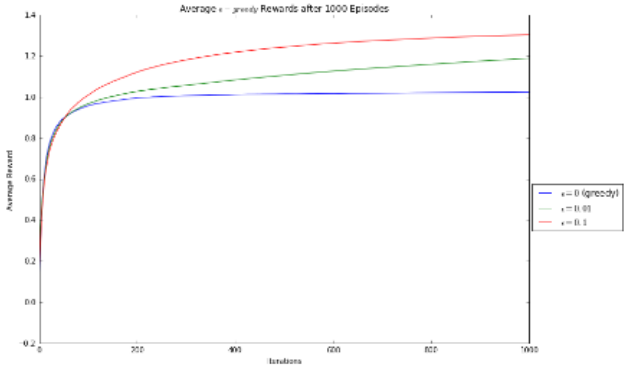


Figure 2.3: Greedy with different values

have studied the algorithm with the performance parameter of reward. We also performed the experiments with regret performance measure. Here is the comparison between the three: So we

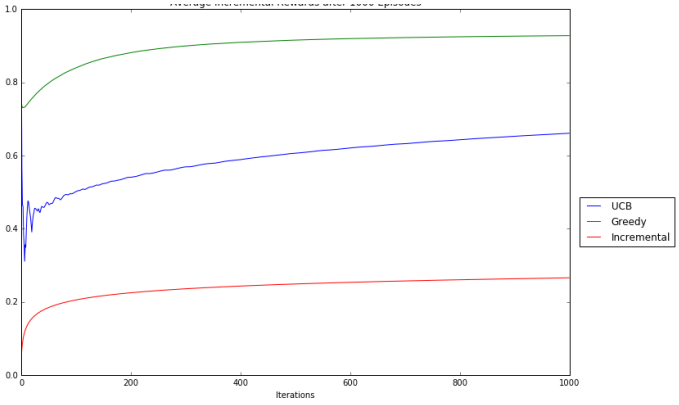


Figure 2.4: Comparison between three algorithms

can see clearly, greedy algorithm is outperforming both and by a margin, Here the incremental algorithm under performing because we are here considering the average reward with all the arms. But still greedy is outperforming both.
Here is comparison on regret. By formula of regret, we minus the reward of an arm from the best arm that round. So because of this we do not have regret for UCB algorithm. It is due to the reason, arm pulled in UCB is due to a optimal criteria which considers the only best arm pulled. For greed, we only have the regret only when we take a random decision because then we are ignoring the optimal decision. Here is graph of the two algorithms. We taken average regret for incremental approach. It is somewhat giving negative regrets, may be some problem in our code. The line for greedy is almost on 0 but it is not exactly zero. There is very small regret, close to zero.

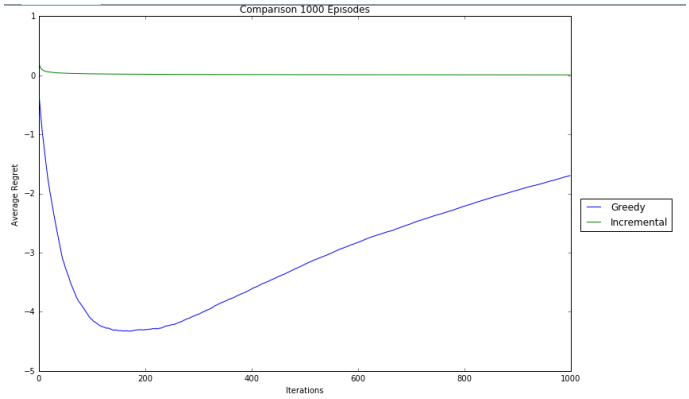


Figure 2.5: Average Incremental Reward

Here are a few comments about the code. It is messy. We are sorry for that. For incremental algorithm, we can get the average reward for best arm by print the parameter of incremental object, $k_(reward)$. If we call numpy function argmax on this array we get the arm which has best reward and later we can print that too. $k_(regret)$ is attribute for average regret of each arm. The regret() function in the code which is outside the class, is to calculate the regret and the simple script, at the very end of the code is to calculate rewards and plot it. The code is run for 1000 iteration and 1000 episodes. We used all the experimentation on normal distribution. We tried to keep the names of variables very meaningful. The code prints arm with best average reward at the end.

Chapter 3

Reinforcement Learning and PACMAN

3.1 Project Answers

3.1.1 Question 1: Value Iteration

Question 1 asks the student to implement a value iteration algorithm to compute the values of the states in the Gridworld. The Value Iteration algorithm updates the value for a state by finding the optimal action when in this state, as shown in the following equation:

$$V_{t+1} = max_{a \epsilon A}[\rho(s,a) + \gamma E_{s' \ \tau(s,a)} V_t(s')]$$

where s is the state, a is the action, ρ is the probability of taking action a in state s multiplied by the expected reward of taking this action in this situation, γ is the discount factor, and it is multiplied by the value of the new state after taking action a. The algorithm works by iterating over all states and updating their values using this equation, then repeating this for a specified number of times.

Question 1 also asks to code two functions: One computes the best action according to the value function, the other returns the Q-value for a (state, action) pair according to the value function. For computing the best action, the value of the above equation is computed for each possible action in the state, and the action with the maximal value is returned. In the case where there are no possible actions (such as the terminal state), None is returned. For computing the Q-value, one must take the sum over all possible transitions of the probability of the transition multiplied by the reward of this transition, plus the value of the new state after the transition. This can be expressed in the following equation:

$$Q(s) = \sum_{a \epsilon A} \rho(s,a) + E_{s' \ \tau(s,a)} V_t(s')$$

3.1.2 Question 2: Bridge Crossing Analysis

Question 2 asks to change either a noise parameter or a discount parameter in a bridge Gridworld set up. In the setup, there is a low-reward terminal state and a high-reward terminal state. There is a bridge between the two, on either side of which is high-negative reward states. The agent starts near the low-reward terminal state. The goal is to change either the discount parameter or the noise parameter so that the agent will cross the bridge to the high-reward state. The discount parameter controls how much of the next state's value should be taken into account when computing the value of a state, while the noise parameter controls how much of the time the agent will move into an unintended state when performing an action. The current parameter values (discount = 0.9, noise = 0.2) cause the agent's path to be towards the low-reward terminal state.

I decided to change the noise parameter and make it extremely low. This means that the agent will very rarely move into a state that it does not intend to (i.e. the high-negative reward states on either side of the bridge). Setting the noise parameter to 0.002 made the bridge have a much higher reward, making the agent want to go to the high-reward terminal state. In the original parameter setting (noise = 0.2), the possibility of unintentionally moving into one of the high-negative reward states was driving the value of the bridge down. But with a lower noise parameter, the bridge had higher value and the agent finds its way across the bridge to the high-reward terminal state.

3.1.3 Question 3: Policies

Question 3 is similar to question 2 in that it asks to change parameter settings to get a desired optimal policy in the DiscountGrid layout. The parameters to change are the discount, noise, and living reward.

3.1.3.1 Question 3a

This policy is to prefer the less rewarding close terminal state, while risking going by the cliff of negative terminal states. The parameter settings to achieve this policy are discount = 0.9, noise = 0.2, living reward = -3. By setting the living reward to negative, the optimal policy is to just get to get to the less rewarding terminal state as quickly as possible.

3.1.3.2 Question 3b

The policy should prefer the less rewarding terminal state, but avoid the cliff and go the long way to get to it. The parameter settings to achieve this are discount = 0.2, noise = 0.2, living reward = -2. The discount is reduced so that when the first step will be away from the cliff because the value of going next to the cliff is reduced. The living reward is negative so that the policy will favor going to the closer terminal state and not the higher-reward farther state.

3.1.3.3 Question 3c

This policy is to prefer the farther high-reward state by way of the cliff. The parameter settings to achieve this are discount = 0.9, noise = 0.002, and living reward = 0.002. The noise is made small so the risk of going off the cliff is reduced, thus making the value of the desired path higher. The living reward is made smaller as well to make the policy prefer going the short way next to the cliff instead of the long way away from the cliff.

3.1.3.4 Question 3d

The policy shold prefer the high-reward farther terminal state and should go the long way avoiding the cliff. The parameter settings to achieve this are discount = 0.85, noise = 0.2, living reward 0.01. With the noise parameter set to 0.2, there is enough of a chance of ending in a negative reward terminal state, so the path will go away from the cliff. Since the living reward is positive, the path has no reason not to go to the farther more-rewarding terminal state.

3.1.3.5 Question 3e

The optimal policy should avoid the exits and cliff and should never terminate. The parameter settings to achieve this are discount = 0, noise = 0.15, living reward = 3. With discount set to 0, the policy does not take into account the resulting state's value, making the path less likely to find a terminal state. The living reward being high gives the policy incentive to not terminate.

3.1.4 Question 4: Q-Learner

Question 4 asks to complete functions for a Q-Learning agent, specifically how to update the learner, how to compute the value from the Q-values, how to get a specific Q-value for a state and action, and how to compute which action to take from the Q-values.

3.1.4.1 Update

The goal of the Q-learner update is to converge to Bellman's equation. When given a Q_t, a current state s_t, a action a_t, the next state s_{t+1}, the transition reward r_t, the discount γ, and the learning rate η, one can compute the update of the Q-value for a specific state and action using the following equation:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + (r_t + \gamma max_{a'} Q_(s_{t+1}, a'))$$

3.1.4.2 Other Funtions

The other functions are relatively simple to implement. To compute the value for a state from the Q-values, you take the maximum Q-value for all possible actions in the state. Getting the Q-values is simple depending on how they are store in the code. In this case it was in a python dictionary so simply indexing by the (state, action) tuple returned the associated Q-value. To compute the action from the Q-values, you take the action with the maximum Q-value.

3.1.5 Question 5: Epsilon Greedy

The goal of question 5 is to implement a epsilon greedy approach to choosing an action. This means that ϵ fraction of the time, it will choose a random action, and otherwise it will choose the action with the highest Q-value. To implement this, one needs a random number generator or something to similate a binary variable with a positive output ϵ fraction of the time. If it is positive, a random action will be chosen, otherwise it will choose the highest Q-value action.

3.1.6 Question 6: Bridge Crossing Revisited

In Question 6, the Q-learner implemented above is tested on the bridge scenario from Question 2, with noise equal to 0. The question is, is there an ϵ value and learning rate η value for which the optimal policy (crossing the bridge) will be learned with high probability after 50 iterations. After trying multiple ϵ values and multiple η values, I came to the conclution that it is not possible to learn this policy in 50 iterations. The learner needs more iterations to fully explore the space.

3.1.7 Question 7: Q-Learning and Pacman

Question 7 is simply to make sure the implemented Q-learning agent works on the Pacman game. The Pacman learner is a wrapper around the implemented Q-learning agent with specific learning parameters that are better for the Pacman application, specifically ϵ=0.05, α=0.2, γ=0.8. The small ϵ value means it will rarely choose a random action and usually choose the action with the highest Q-value. The lower α value gives more weight to the current Q-value when updating. With the discount γ being relatively high, the learner will consider the Q-value of the following state more than if the discount was less.

3.1.8 Question 8: Approximate Q-Learning

In question 8, an approximate Q-Learning algorithm is to be implemented. In approximate Q-Learners, there is a function $f(s, a)$ which will take a state s and an action a and return a feature vector, which can then be multiplied by learned weights to approximate the Q-value of a state-action pair. Thus, to compute the Q-value of a state action pair, first the feature vector has to be obtained from the function, then the dot product must be taken between the feature vector and the weights. This is shown as:

$$Q(s, a) = \sum_{i=1}^{n} f_i(s, a) w_i$$

The weights will be updated by a process similar to updating the Q-values. At each step, for each weight, the following update will be performed:

$$difference = (r + \gamma max_{a'} Q(s', a')) - Q(s, a)$$

$$w_i = w_i + \alpha \cdot difference \cdot f_i(s, a)$$

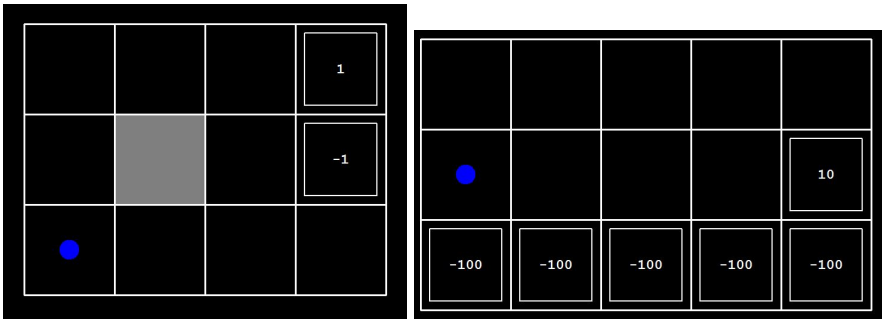
3.2 Experimental Study

As we have seen, the main difference between Value Iteration and Q-Learning is that the Q-Learning agent does not know the transition rewards and reward models apriori. Since Value Iteration has access to knowledge that Q-Learning doesn't, it makes sense to assume a Value Iteraton agent will surpass the performance of a Q-Learning agent when computed on the same grid. We will explore this question as well as analyze the behavior of the two algorithms in the following Experimental Study.

3.2.1 Experimental Setup

We will use two grids from the Berkeley AI Project: BookGrid and CliffGrid.

Figure 3.1: BookGrid (left) and CliffGrid (right)



We will run each algorithm several times (for different number of iterations) to tune hyperparameters and accumulate results. Then we will compare the empirical average rewards for 10 runs of each resulting policy that the agent has learned. Thus, we can compare how well the agents have learned and their behavior, such as how many more iterations of learning do the Q-Learning agents need to make up for their lack of knowledge of the transition rewards and reward models. Unfortunately, the girds in the Berkeley AI project that allow both value iteration and Q-learning agents do not have an option to let the Q-agent train before getting statistics such as empirical average rewards. Thus, the metrics are not exactly fair for the Q-learning agents, as their results include training errors and the value iteration agent results are after training. But these is the

Table 3.1: Results for on BookGrid Environment

Agent	Training Iterations	Noise	Average Reward
Value	10	0.0	0.59049
Value	10	0.2	0.493173173619
Value	100	0.0	0.59049
Value	100	0.2	0.505128582548
Value	500	0.0	0.59049
Value	500	0.2	0.52668696501
Q	10	0.0	0.213981781055
Q	10	0.2	0.271650321233
Q	100	0.0	0.291498478256
Q	100	0.2	0.217350545697
Q	500	0.0	0.460252095844
Q	500	0.2	0.316614799129

Table 3.2: Results for on CliffGrid Environment

Agent	Training Iterations	Noise	Average Reward
Value	10	0.0	6.561
Value	10	0.2	4.34521940083
Value	100	0.0	6.561
Value	100	0.2	3.72444917933
Value	500	0.0	6.561
Value	500	0.2	4.06608776878
Q	10	0.0	-36.1973456558
Q	10	0.2	-27.9008334858
Q	100	0.0	-17.2162854474
Q	100	0.2	-22.8042941525
Q	500	0.0	-15.6513674395
Q	500	0.2	-11.4166133052

environments we have and the only ones in which we can run both agents, so we have to make due.

3.2.1.1 Hyperparameters

We will test multiple values of the noise (0.0 and 0.2) parameter for transitions and keep the living reward constant (set at 0.0). We will keep the discount constant at 0.9 for both algorithms, and test with the number of iterations varying from 10 and 100. Also, we will test the Q-Learning agent with a learning rate of 0.5 and with epsilon value of 0.3.

3.2.2 Results and Analysis

3.2.2.1 BookGrid

The results for the BookGrid are shown in Table 3.1. As expected, the Value iteration algorithm has much better results, as it has knowledge that the Q learner does not and it is evaluated on the learned policies as opposed to with the training. With no noise, the value agent achieves the optimal policy after just 10 training iterations. This is because the grid is relatively small. The noise has less negative effect on the reward with more training iterations. The Q-learner's results are much worse. The noise seems to help with just 10 iterations, as it is not enough to learn

good Q-values and thus the randomness helps the Q-learner. However, with more iterations, the noise hurts the reward. After 500 iterations, the average reward for the Q-learner with no noise is approaching the optimal, so it is likely that it had learned the optimal policy by then, as the early failures while learning brought down the average.

3.2.2.2 CliffGrid

The results for the CliffGrid are shown in Table 3.2. Again, the value iteration reward values are much better than that of the BookGrid. Also, since the CliffGrid has a few terminal states with a large negative reward (-100), the rewards for the Q-learner are negative, as they contain the training rewards as well. The value iteration again learns the optimal policy after just 10 iterations. In a couple of experiments, the noise seems to help the Q-learner, as shown by the higher reward than without noise. This is because the noise probably saved the Q-learner from making some mistakes and helped the learning early on.

3.2.3 Conclusion

These experiments show just how powerful the apriori knowledge of the transition rewards and reward models can be, as the value iteration agent performs much better off of less training iterations than the Q-learner. Also, we see that in some situations, higher noise can help the Q-learner and can speed up learning, while noise seems to always hinder the value iteration agent.