Online Learning, Bandits, Reinforcement Learning

Advanced Machine Learning Lab Session

BLANCHARD Jérémie, BOULDJEDRI Oussama, SCHWINN Austin

Online Passive-Aggressive Algorithms

Without Noise

We implemented this algorithm in Python and we decided to use the classification generator from sickit learn to create a dataset that we are going to use to train our SVM, and another to test it. To update our weights, we had to implement three different methods to update them and compare our results.

We obtained the following results:

- With the classic update:

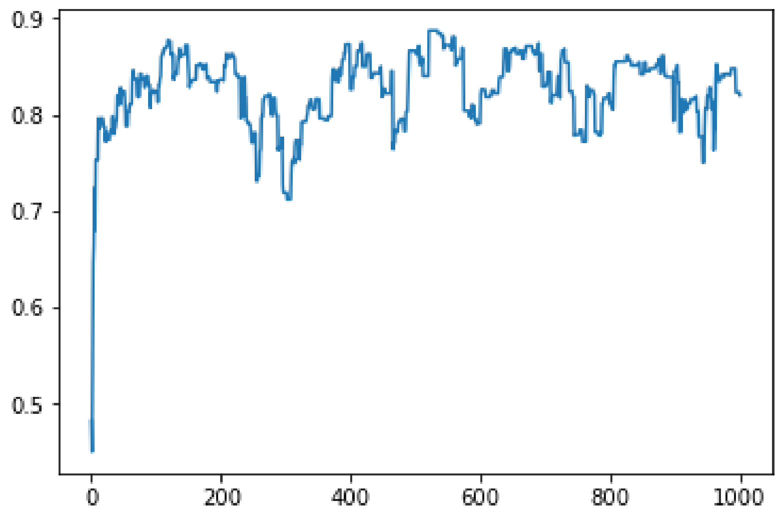


Figure 1- Accuracy in % / Number of iterations

Final Accuracy: 82 %

Total Computational Time: 0.05 s

- With a first relaxation with C=0.1:

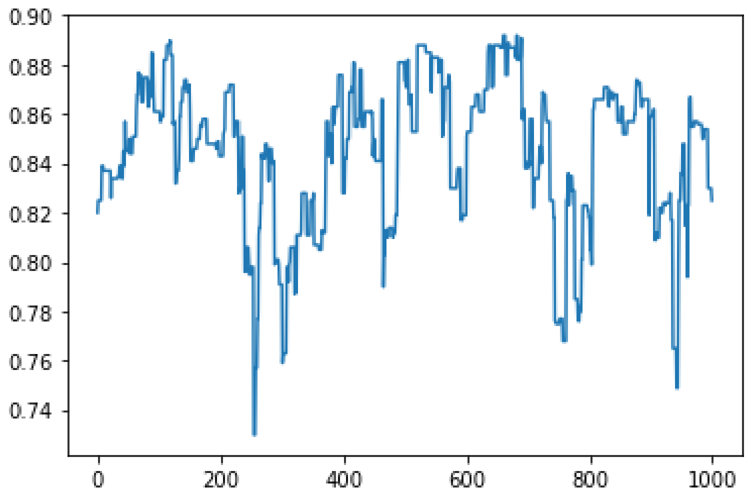


Figure 2- Accuracy in % / Number of iterations

Final Accuracy: 82.5%

Total Computational Time: 0.11s

- *With a second relaxation with C=0.1:*

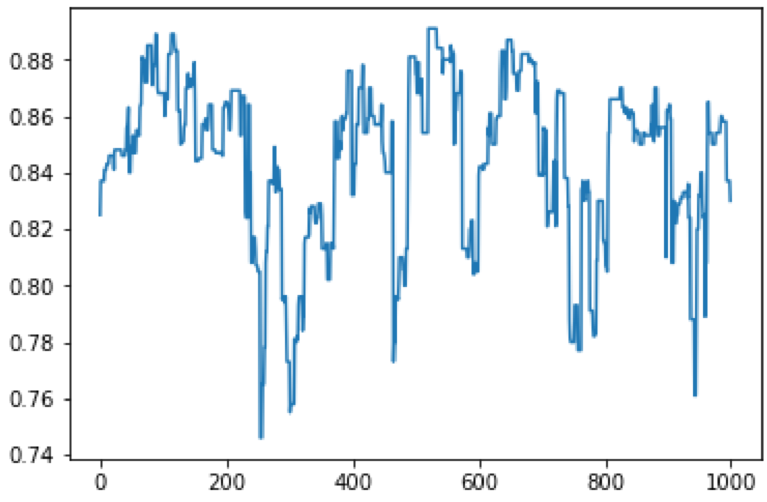


Figure 3-Accuracy in % / Number of iterations

Final Accuracy: 83 %

Total Computational Time: 0.15s

- With LibSVM we obtained:

Final Accuracy: 61.4 %

Total Computational Time: 0.001s

Conclusion:

It seems that the different Thetas help to improve the accuracy but increase a lot the computational time. With LibSVM we have the same correlation and we have a bad final accuracy but a fast-computational time.

With 5% Noise

Now Let's see how they behave when we add *5 percent* of noise.

We obtained the following results:

- With the classic update:

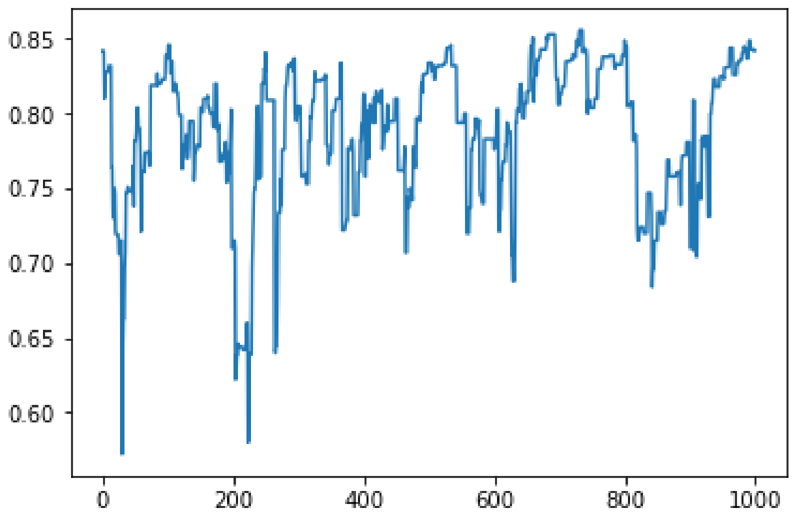


Figure 4- Accuracy in % / Number of iterations

Final Accuracy: 84.2 %

Total Computational Time: 0.08 s

- With a first relaxation with C=0.1:

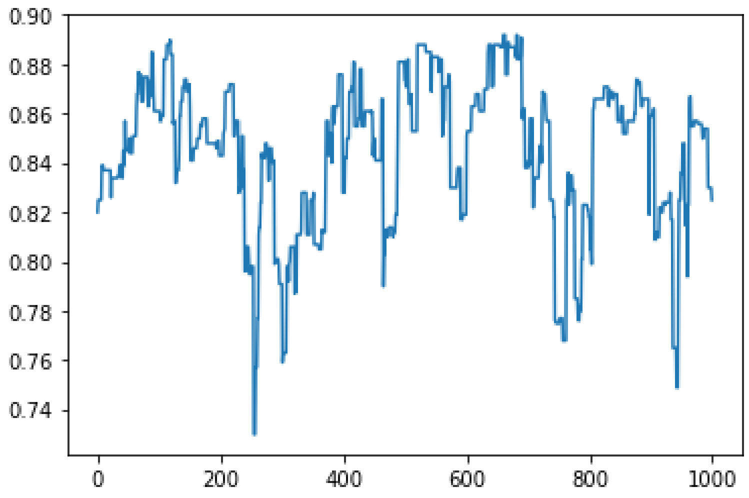


Figure 5- Accuracy in % / Number of iterations

Final Accuracy: 85.1%

Total Computational Time: 0.15s

- *With a second relaxation with C=0.1:*

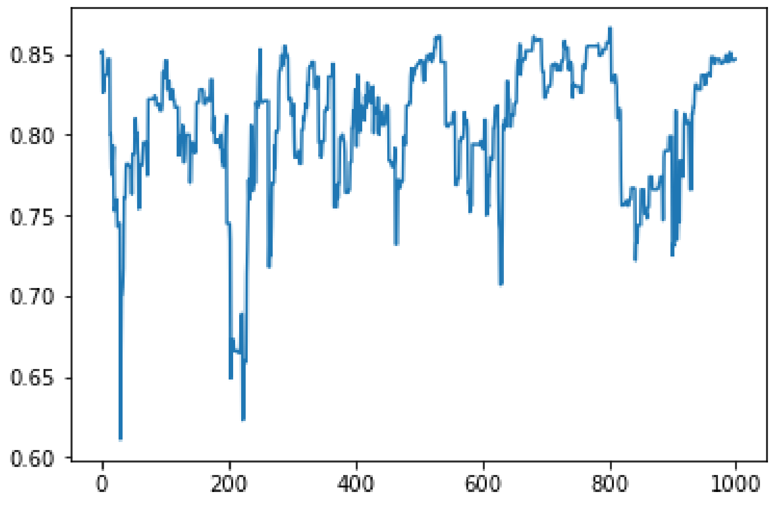


Figure 6-Accuracy in % / Number of iterations

Final Accuracy: 84.7 %

Total Computational Time: 0.21s

- With LibSVM we obtained:

Final Accuracy: 55.5 %

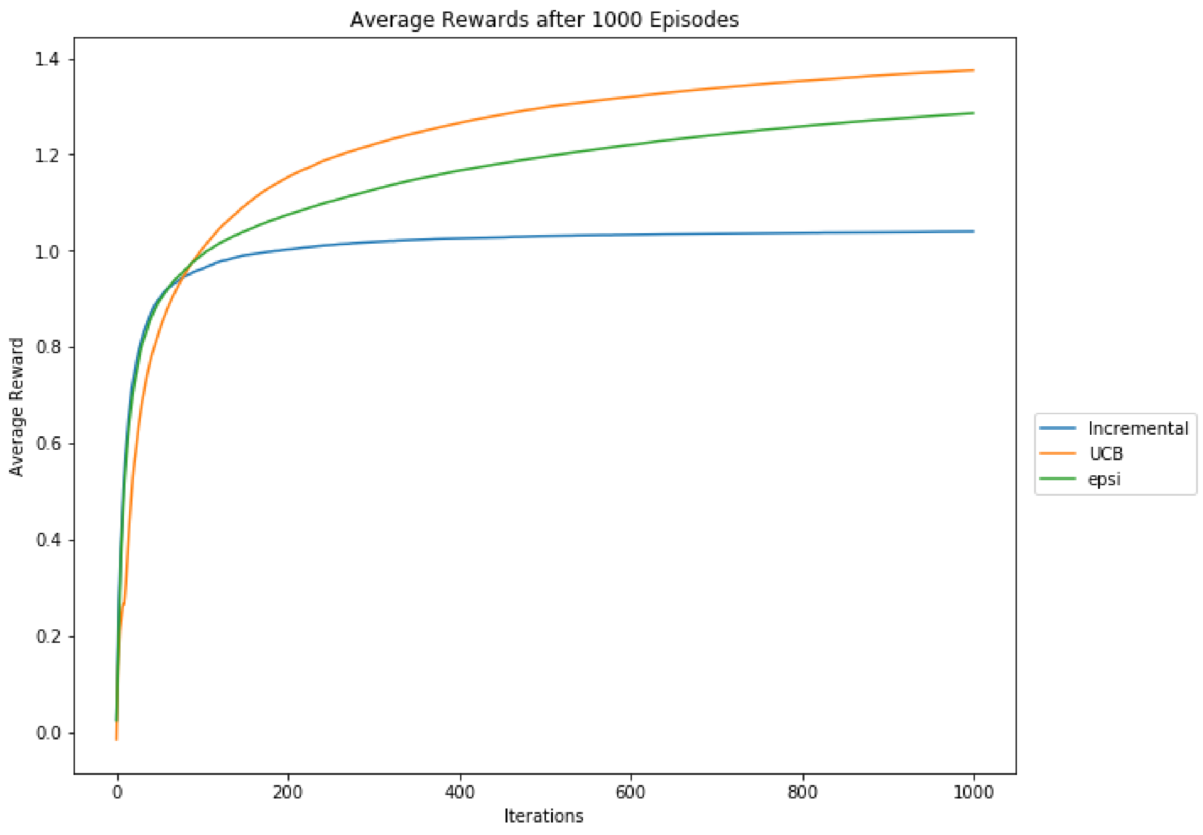
Total Computational Time: 0.002s

Conclusion:

We can observe the same things as before except that this time we can clearly see that Teta 2 and 3 avoid the model to lose too many accuracies at some steps. Here again LibSVM does not perform well and has a final accuracy of only 55.5 % but it has also a small computational time.

Bandit Algorithm

To compare these three different approaches, we decided to create a Python class where we implemented these algorithms. Then we are launching these approaches in parallel and see the results on a graph.



We can see that every algorithm has a specific shape and evolves differently. For example, it's clear that the "Incremental Uniform" algorithm stops "learning" quickly while the UCB and Epsi algorithm keep learning even after a lot of iterations. It also seems that the average reward for the epsi algorithm is going to catch up with the UCB one after some more iterations.

Reinforcement Learning

Question 1: Value Iteration

The first part of this assignment is to build a value iteration agent. This agent calculates and updates the expected value of each state for the optimal action according to each state. The expected values are based on a Markov Decision Process that is updated through a specified number of iterations. There are two separate functions used to select the best action for a current state. The iteration agent finds the expected value with the following algorithm:

```
For k in interation:
    states = possibleStates()
    For state in states:
        actions = possibleActions(state)
        bestValue = -1
        For action in actions:
            transitions,probabilities = possibleTransitions(state,action)
            transitionSum = 0
            For transition,probability in transitions,probabilities:
                reward = calculateReward(state, action, transition)
                transitionSum += probability*(reward+discountRate*transtition)
            bestValue = max(bestValue,transitionSum)
        expectedStateValue = bestValue
```

By iteration this and storing the best expected value for each state, we can then use the other two algorithms that we are asked to implement for this question to find the optimal path. The first is computeQValueFromValues. This returns the Q-value given the value of the function. The Q-value for a given state and action is computed with the following algorithm:

```
computeQValueFromValues(state,action):
    transitions,probabilities = possibleTransitions(state,action)
    transitionSum = 0
    for transition,probability in transitions,probabilities:
        reward = calculateReward(state, action, transition)
        transitionSum += probability*(reward+discountRate*transtition)
    return transitionSum
```

The second algorithm selects an action based on the best value:

```
computeActionFromValues(state):
    actions = possibleActions(state)
    bestValue = -1000
    bestAction = NONE
    for action in actions:
        q = computeQValueFromValues(state,action)
        if q >= bestValue:
            bestValue = q
            bestAction = action
    return bestAction
```

Question 2) Bridge Crossing Analysis

This question has a “bridge” between a low-reward state and a high-reward state with the initial state being the low reward. On either side of this bridge, are very high negative rewards. With the initial parameters of the discount being set to .9 and the noise being set to .2, the agent will move on square and then return to the low-reward state because it still has a higher expected value than moving towards the high reward state on the other end of the bridge. The question has us alter the parameters to reach the other end. In this instance, the discount rate being high is valuable because we want to keep strong consideration of actions and transitions further away from the initial state because that is where the high value reward is. So instead, we want to adjust the noise parameter which controls the likelihood the agent with do an unwanted action into an unspecified state (which have the high value negative rewards). We adjusted the noise from .2 to .01 and the expected values altered to our desired path and the agent reached the high value state supporting our hypothesis.

Question 3) Policies

This question uses the discount, noise, and living reward parameters to change the policy of what type of optimal path is created. It is an extension of the previous lab but on the DiscountGrid layout. Listed are the policies with their parameter values:

a. Prefer the close exit (+1), risking the cliff (-10)
    - Discount=.5
    - Noise =.01
    - Living reward =-5
b. Prefer the close exit (+1), but avoiding the cliff (-10)
    - Discount=.2
    - Noise =.2
    - Living reward=2
c. Prefer the distant exit (+10), risking the cliff (-10)
    - Discount=.9
    - Noise =.01
    - Living reward = .01
d. Prefer the distant exit (+10), avoiding the cliff (-10)
    - Discount=.9
    - Noise =.2
    - Living reward =.01
e. Avoid both exits and the cliff (so an episode should never terminate)
    - Discount=0
    - Noise =.2
    - Living reward = .01

Question 4) Q-Learner

In our previous implementations, we have the MDP predefined for us. At this point we do not, we must update the true Q-learner, which we use to converge on Bellman’s equation. The Q-learner update is based on the following algorithm:

```
updateQValue(state,action,nextState,reward):
        QValue(state,action)=QValue(state,action)+alpha*(reward+(QValue(Sate+1,possibleAction
s)
```

In this way, the Qvalues are stored as they are updated. To compute the QValue, a function would just return the maxium QValue for all possible actions for the current state. It is as follows:

```
computeActionFromQValues(state):
        actions = possibleActions(state)
        bestValue = -1000
```

```
    bestAction = NONE
    for action in actions:
        qValue = getQValue(state,action)
        if qValue > bestValue:
            bestValue = qValue
            bestAction = action
    return bestAction
```

Question 5) Epsilon Greedy

The Epsilon value in the QValue interjects random actions for the agent to perform. These random moves may be sub-optimal but it allows the agent to explore more actions throughout its iterations than if it only went to optimal actions. This lab recommends using the util.flipcoin function in python for the following getAction algorithm:

```
actions = possibleActions(state)
if util.flipCoin(Epsilon):
    return random.choice(states)
else:
    return computeActionFromQValue(state)
```

Question 6) Bridge Crossing Revisited

For this question, we use the Q-learner with no noise and see if we can find the optimal policy by adjusting the learning rate and epsilon values with less than 50 iterations. We successfully found the optimal path in less than 50 itesations with the following parameters:

- Learning Rate: 1
- Epsilon: 0

Question 7) Q-Learning and Pacman

We now test our QLearningAgent in the context of the Pacman game. It is set on a smallGrid layout with 2000 training iterations and 10 test iterations. With the default PacMan agent, it uses the QLearning agent with the predefined parameters:

- Learning Rate: 1
- Epsilon: .05
- Discount Rate: .8

These parameters gave us very good results. The agent won all 10 of the test rounds. Here are our full results:

- Average Score: 499.4
- Scores: 503.0, 495.0, 503.0, 495.0, 503.0, 495.0, 503.0, 495.0, 503.0, 499.0
- Win Rate: 10/10 (1.00)
- Record: Win, Win, Win, Win, Win, Win, Win, Win, Win, Win

Question 8) Approximate Q Learning

This question takes a slightly different approach and implements Approximate Q Learning. The QValue is obtain by applying a function of the state and action, which return a feature vector that is then multiplied with the dot product of a set of learned weights. The weights are learned in in the same way that classic QValues are learned but updated the weights throughout iterations. The update of the weights is implemented in the following algorithm:

```
updateWeights(state,action,nextState,reward):
    weights = getWeights()
```

```
features = getFeatures(state,action)
actions = possibleActions(nextState)
bestQValue = -1000
for action in actions:
        q = getQValue(nextState,action)
        bestQValue = max(bestQValue, q)
for feature in features:
        difference = (reward+discount*bestQ) – getQValue(state,action)
        weights = weights + alpha*difference*features
```

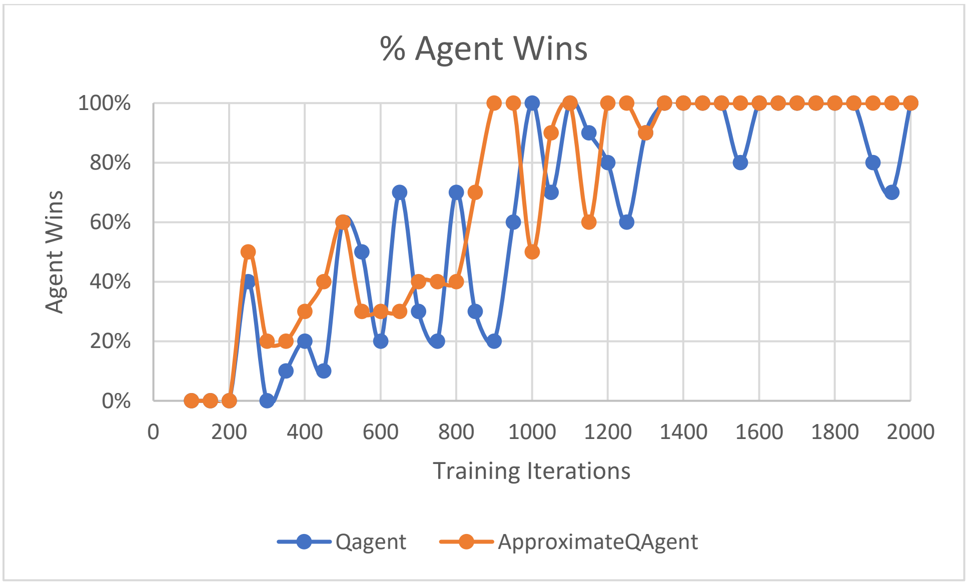
We tested the approximate Q Learning Agent on the small PacMan grid with 2000 training interations and 10 iterations. We actually received the exact same average score and number of wins as our test with the original QLearningAgent. We received the following results:

- Average Score: 499.4
- Scores: 499.0, 503.0, 495.0, 495.0, 495.0, 503.0, 495.0, 503.0, 503.0, 503.0
- Win Rate: 10/10 (1.00)
- Record: Win, Win, Win, Win, Win, Win, Win, Win, Win, Win

EXPIREMENTATION

For our experimentation, we decided to compare how quickly Q-Learning and Approximate Q-Learning learn to win in the game of Pacman. To do this, we will keep our parameters constant at the default specified by the lab and instead adjust the number of training iterations the agents learn before testing them on 10 final games. We are keeping the map constant and using the smallGrid. We tested from 100 to 2000 iterations by increments of 50.

RESULTS



We found that the approximate Q-Learner outperformed the standard Q-Learner but only slightly. Both agents were unstable in the beginning, reaching up to 40% - 50% win rates by 250 training iterations and then crashing back to low win percentages. The Approximate Q-Learner generally had better win rates until 600 iterations and then surpassed the standard Q-Learner, being the first to reach 100% win rate by 900 iterations. While both gradually grew with large variance in the win rate between training steps, both converged on optimal policies around 1400 iterations. The larger performance difference we observed, though, was that once the Approximate learner converged on an optimal policy, its variance in win rate dropped off. The standard Q-Learner, on the other hand, continued to experience strong variance in its win rate well after converging on an acceptable policy. Because of this, we believe the Approximate Q-Learner is the stronger choice.