University of Jean Monnet

Advanced Machine Learning

Practical Session 2

Online Learning, Bandits, Reinforcement Learning

Author:
Allwyn Joseph
Yevhenii Zotkin

Supervisor:
Dr. Amaury Habrard

December 7, 2018

Contents

List of Figures

1 Introduction

The principle aim of the practical session was to implement state of the art online learning and reinforcement learning algorithms and experiment on them with varying values of hyper-parmeters.

This report documents our approach towards effectuating a successful practical session. It begins with presenting a little about each algorithm, their pseudo-codes and putting together the code for the same. This is followed by testing of the algorithm based on accuracy or and time and present the results in tables and plots. All of the code for this assignment can be found on Github

2 Online Passive-Agressive Algorithms

For this question, we propose a custom implementation of SVM learning algorithm having 3 different weight update techniques. Formally, we define weight update rule for SVM classifier as $w_{t+1} = w_t + \tau_t y_t x_t$. In each of the analyzed techniques we change the definition of parameter τ. In the classical weight update rule, which enforce strict unit margin between examples of opposite class, the parameter τ defined as $\tau_t = \frac{l_t}{||x_t||^2}$, where l_t is a loss induced by the classifier at time step t. Further, we call this weight update rule *classic*. Strictly enforcing unit margin can be too aggressive in case of presence of noise in training examples(mislabeled examples, outliers, etc.). To tackle this issue, we can introduce relaxations, which relax unit margin constraint. We call the *first* relaxation the setting, where weight update parameter τ is defined as $\tau_t = \min(C, \frac{l_t}{||x_t||^2})$, where C is a hyperparameter to optimize. We call the *second* relaxation the weight update rule where τ is defined as $\tau = \frac{l_t}{||x||^2 + \frac{1}{2C}}$.

The goal of our study is to test the variations of the SVM algorithm presented above in online setting. We define the online training as a setting where training algorythm update it's weights based on only one example at the time. In our tests, we will also use Scikit-learn implementation of SVM classifier, which uses LibSVM under the hood. We perform the following experiments:

1. Execution time comparison.

2. Accuracy comparison.

3. Accuracy comparison in presence of noise.

2.1 Execution time comparison

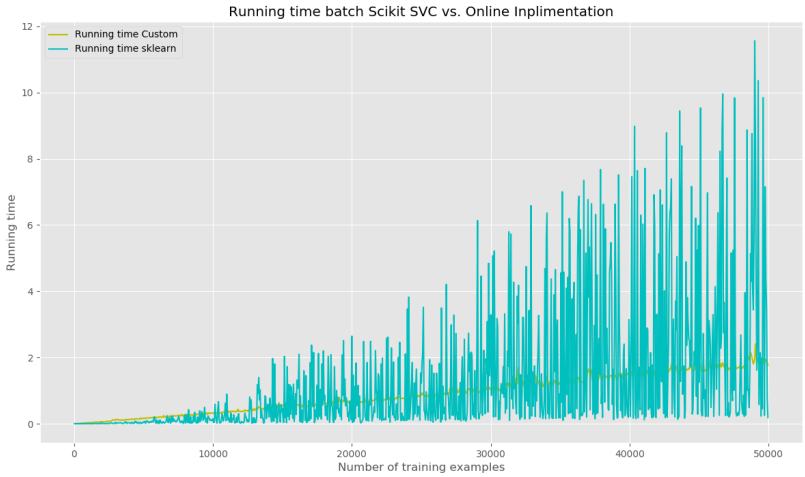

Figure 1: Execution time comparison. Y axis in seconds

In order to compare execution time we use the synthetic dataset generated by Scikit-learn datasets tools. The motivation behind this is the possibility to dynamically change the dataset size. We record execution time for each dataset size from 50 to 50000 in the interval of 50. For each dataset we record the time needed to fit the classifier on 75% of data and predict on 25%. All training is done in online setting, i.e. batch size of one. The python inspired pseudocode for comparison algorithm provided bellow in Figure 2. Execution time provided in Figure 1. As we observe, in best-case scenario scikit learn implementation is more efficient than custom python implementation. One reason for this is that scikit learn is using C++ LibSVM implementation under the hood. On the other hand, as can be observed on the figure, sklearn implementation suffers from sharp spikes in running time, the reason of which we failed to find. We test only strict version of the algorithm, since changing the constraint does not significantly impact the running time.

```
execution_times =[]
for i in dataset_size(from 50, to 50000, in_step 50)
    X, y = generate_dataset(i)
    exec_time_custom, exec_time_sklearn = train_and_test(X,y)
    execution_times.add(exec_time_custom, exec_time_sklearn)
return execution_times
```

Figure 2: Time comparison algorithm

2.2 Accuracy

We test the accuracy of our algorithm on Wisconsin Breast Cancer Dataset []. We use this dataset since it is a binary classification dataset, and can be tested on both Scikit Learn and our implementations. All the results were obtained using 20 folds cross validation. We use 2 different experimental settings to asses the performance. In first, we employ pure online learning strategy for our algorithms, i.e. for some number of examples n we compute accuracy during training on the unseen examples, then we update the classifier with this example. In the second case, we train our classifier using one example at the time, but computing accuracy on separate test set (in cross-validation setting). The results are provided in Table 2.2 . As for noise testing, we provide the results for different noise levels, where .1, .2, .5 fractions of training set labels were randomly flipped.

	SVC	Classic	First	Second
Pure Online	-	80%	76%	78%
On test set	98 %	76 %	98%	96%
Noise.1	93%	78%	82%	73%
Noise.2	94%	79%	79%	73%
Noise.5	62%	57%	56%	53%
C Value	0.76	-	0.06	0.03

Table 1: Accuracy on Boston Breast Cancer Dataset

3 Bandit Algorithm

We consider an 8 arm bandit in our experiment. We define one arm as some random variable X which follows some distribution D. We assume that arms does not have the same probability $P(X = x)$ and there exist an arm with highest probability, yielding lowest regret. In order to define the notion of regret, following the course we are going to define the reward μ_i^t as a value $X = x$ that arm i produce at timestep t. We then define μ^* as the best reward achieved by any arm until current timestep t. Finally, we define regret as $r_t = \mu^* - \frac{1}{T}\Sigma\mu_t$, the difference between the best reward obtained so far and average reward obtained.

3.1 Experimental Setup

In our experiments we use test 3 different distributions for X, Normal, Uniform with range being variable for each arm (from 0 to 1), so that there must be the best arm (with biggest dynamic range), Binomial with different probability p for each arm. We fix parameters of those distribution independently by sampling from Uniform distribution and eliminating the possibility for multiple maximum values.

For each of the proposed algorithms we repeat our experiment for some k times (usually $k = 1000$).

3.2 Incremental Uniform

In this algorithm we use pure exploration mechanism, where we pull each arm equal amount of times and select the best arm based on average reward for each arm. The pseudocode for the algorithm is provided in Figure 3.

3.3 UCB

In UCB variation, we decide to which arm to pull next based on Upper Confidence Bound (thus UCB). Following course we define ucb as $UCB_i(t) = \hat{\mu} - \sqrt{\frac{2\log(t)}{N_i(t)}}$. At each point of time t we pull the arm with highest UCB. In theory, this algorithm should find optimal arm is fastest among other algrythm. One of the drawbacks thought, is that UCB is susceptible to stuck with sub-optimal arm if this arm randomly gives good results during the early iterations of the algorithm.

```
require number_of_experiments;
initialize Arms;

for each experiment:
    for each arm:
        arm.pull().save_reward();
rewards = [];
for each arm:
    avg_reward = arm.compute_average()
    rewards.add(avg_reward)

return argmax(rewards)
```

Figure 3: Incremental Uniform algorithm.

3.4 $\epsilon - greedy$

This variation attempts to find the optimal tradeoff between exploration and exploitation by introducing probability ϵ, with which we are going to exploit, otherwise explore. The prseudocode sketch is presented in Figure 4.

```
require number_of_trials;
require epsilon;

for each trial:
    let x = random_probability;
    if x< epsilon:
        pull_best_arm()
    else:
        pull_random_arm()
return best_arm
```

Figure 4: ϵ -greedy algorithm

3.5 Results

The results of the conducted experiment can be seen on Figure 5. As expected, we can observe that all algorithms converge to optima with different speed depending on the parameters in case of epsilon-greedy.

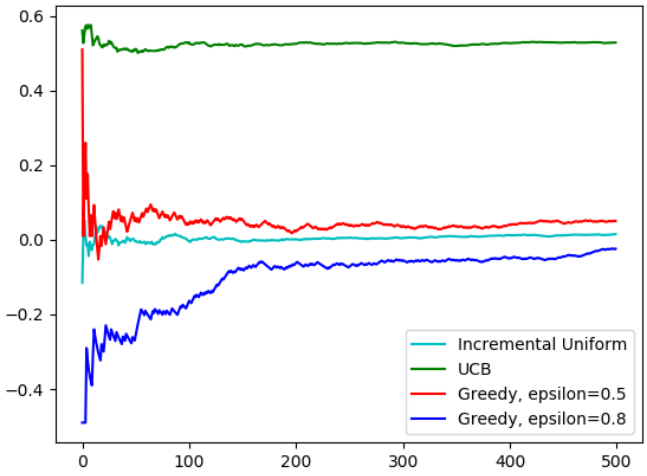

Figure 5: Average Regret for different bandits implementation

The arms are set to draw random values of Bernoulli trial with some predefined probability p. For each set of trials there exist the best arm.

4 Reinforcement Learning [2]

In this question we tackle the problem of 'Model Free Prediction' or in other words a problem in which the dynamics of the Markovian model is unkown. Methods such as Policy Iteration and Value Iteration are often used to solve a know MDP (Markov Decision Process) where the actions, rewards and transition probabilities are already known. These methods unfortunately can't be used for 'Model Free Prediction' and one would have to resort to Temporal Difference (TD) or Monte Carlo (MC) methods. Q-learning is a subset of the TD method and we'll be exploring it in further detail in this section.

4.1 The Environment

A basic text based environment class was defined for visual purposes - display.py. The class displays the movements of the agent at a defined frequency, along with information on the policy and state-values after every episode. The image below is a sample output from the environment class . The grid world and the matrices appear separately during execution, but have been shown together here for illustrative purposes only).

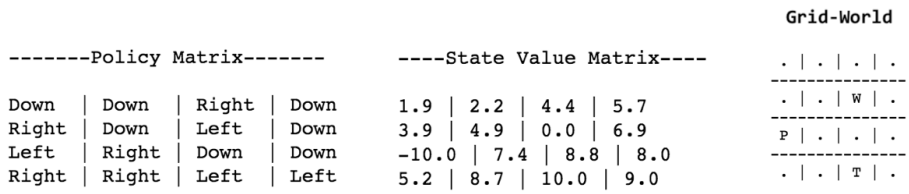

Figure 6: Output from the Environment class

4.2 Q-Learning

As mentioned earlier, the Q-learning algorithm is a subset of the TD methods. Unlike the MC methods, the algorithm effectuates updates to the action value function every iteration (or few iterations in special cases). And since the Bellman Action value function (1) doesn't need the transition probabilities and the state values it makes it an ideal candidate for a function with which the model can learn when the dynamics of the environment is unknown.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r' + \gamma Q(s',a') - Q(s,a)) \quad (1)$$

where

$Q(s,a)$: Action-value at current time step
α : learning rate
r' : reward on taking an action a
γ : discount factor
$Q(s',a')$: Action-value at next time step

The following is the pseudo-code used to construct the Q-learning Algorithm [1]:

$Q \leftarrow Q_0$ (initialization, *e.g.* to 0);
for $t \leftarrow 0$ *to* T **do**
  $s \leftarrow$ SelectState();
  for *each step of epoch* t **do**
    $a \leftarrow$ SelectAction(s, π) - from policy π deduced from Q with ϵ-greedy strategy
    $r' \leftarrow$ Reward(s, a);
    $s' \leftarrow$ NextState(s, a);
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r' + \gamma \max_{a'} Q(s', a') - Q(s, a)]$;
    $s \leftarrow s'$;
return Q;

Once the Q-table is learnt the policy and the state values were predicted using equations 2 and 3 below:

$$V(s, a) = max_{a \in A} \quad Q(s, a) \quad (2) \quad | \quad \pi'(s) = argmax_{a \in A} \quad Q(s, a) \quad (3)$$

Once the algorithm was put together (code can be found on github) the following hyper-parameters were tuned to better understand its effects on the accuracy of the learnt value function :

- ϵ (epsilon) : the probability with which the model decides not to follow the greedy policy.
- γ (gamma) : discount factor -importance attributed to current and future rewards (values between 0 and 1).
- α (alpha): the learning rate
- Agent_random - the probability with which the agent moves randomly (at test time).

4.2.1 Experimentation

The experimentation phase has two main parts to it, training and testing. To begin with, the agent was first put into an unknown environment with educated guess for hyper-parameters gamma, alpha and Agent_random and varying values for epsilon. This was repeated over 100 training episodes for different values of epsilon. Once the agent was done training, the models generated as result of varying values of epsilon were put to test for 1000

episodes - within the same environment the agent had trained in. With the conclusion of the tests, we were able to decide on a suitable value for epsilon. The entire process was then repeated for varying values of gamma, alpha and finally agent_random picking the most suitable values for each hyper-parameter after the tests.

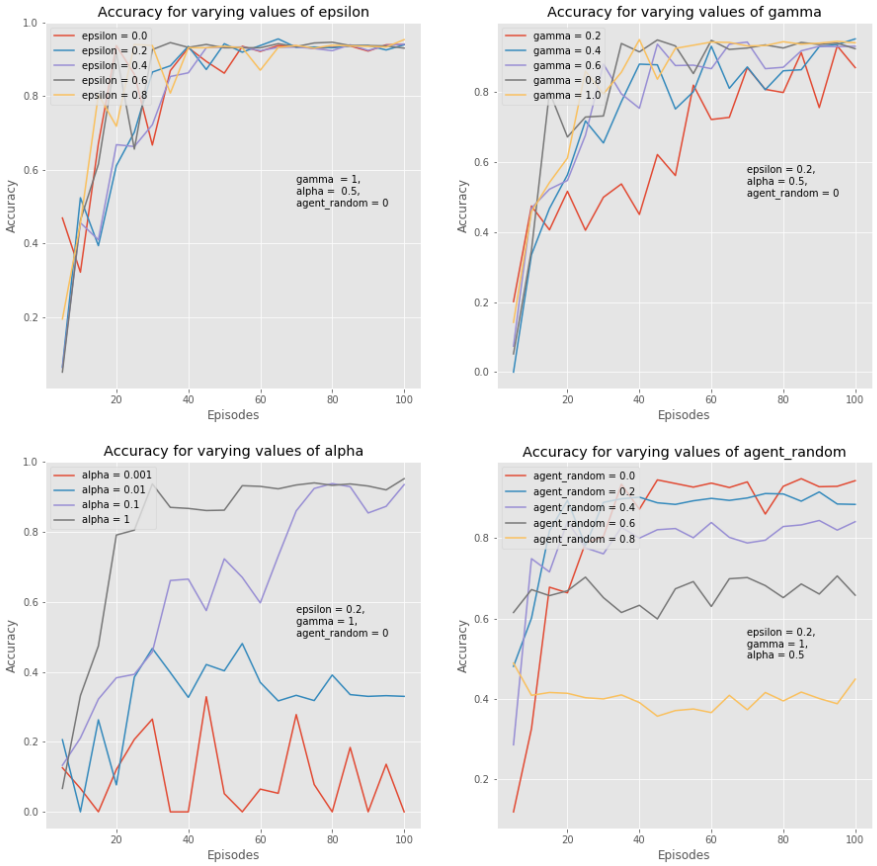

Figure 7: Accuracy plots for varying hyper-parameters - Q-learning

As explained above, and can be seen from figure 7 the ideal value for epsilon was found to be 0.2. In the following iteration, the value of epsilon was fixed to 0.2 while varying the value for gamma. And the plot goes to show that when gammaa = 1, the model performs best during test. Finally, a suitable value for alpha was chosen too, keeping epsilon = 0.2 and gamma = 1 for varying values for alpha.

Epsilon value of 0.2 seems ideal, as while the agent follows a greedy policy, it also from time to time tries and explores other actions. This in theory leads to robust models and also goes to show that as epsilon increases the model takes longer to reach higher accuracies. As for value of gamma, when it equalled 1 the agent performed best. Within the grid world in order for the agent to learn fast and smart a high weight has to be attributed to the action-values of the next step. The model seemed to perform best with alpha = 1, but to avoid missing the minimum due to large step sizes alpha = 0.5 was chosen. And finally, after having learnt a perfect policy and testing the agent on the environment from which the policy was learnt we see that higher the value of agent_random lower is its accuracy which is logical.

4.3 Deep Q-Learning

The 4x4 grid world is weak imitation of real-life MDP scenarios with numerous actions and possible states. So when the MDP gets bigger using the normal Q-learning algorithm to build q-tables is going to be in-feasible taking into consideration the plethora of states and possible actions. In such scenarios we make use of Deep Q Learning, which is simply a neural network assisted Q-learning.

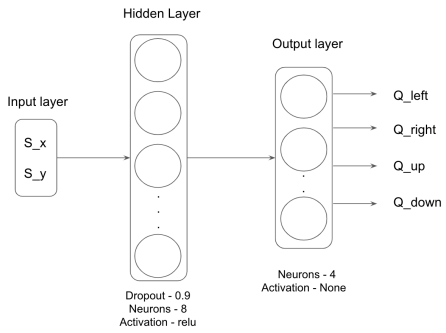

Figure 8: The Nerual-Network used for Deep Q-learning

The peculiarity here would be, at every state the action values are predicted using a neural network seen in figure 8 and an action is taken based on the $\epsilon-$greed approach. The transitions of the agent $(s_t, a_t, r_{t+1}, s_{t+1})$ are then stored in a replay memory D. On fixed episodes the neural network is

trained on a random batch of these transitions and the weights of the network is updated. As the neural-network is trained over episodes it gets better at predicting the state action-values.
[] The loss function L_i used for updating the weight of the neural network is as follows:

$$L_i = (r + \gamma max_{a'} Q(s', a') - Q(s, a))^2 \quad (4)$$

Figure 8 dwells into the neural-net used. As is noticed, the network has only one hidden layer with 8 units (with Relu activation and dropout = 0.9) and an output layer with 4 units (without activation) corresponding to the Q-values for the possible actions (left, right, up and down). The network accepts the x and y coordinates of the current position of the agent in order to predict the Q-values. The inputs to the neural network were fed in batches of 32 and the optimizer used was Adam

The pseudo-code for the Deep Q-learning algorithm is as follows [3]:

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation **4**
  end for
end for

Just like in the previous section tests were conducted on hyper-parameters γ, α, ϵ and Agent_random_test.

4.3.1 Experimentation

The experimentation setup was exactly as defined in the previous section with the only difference that the agent was trained on the environment in episodes of 150 instead of 100 for varying values of hyper-parameters. Since

at each episode the weights of the neural network were updated using only 32 instance of agent transitions, more number of episodes were required to arrive at similar performances as that of the basic Q-learning algorithm.

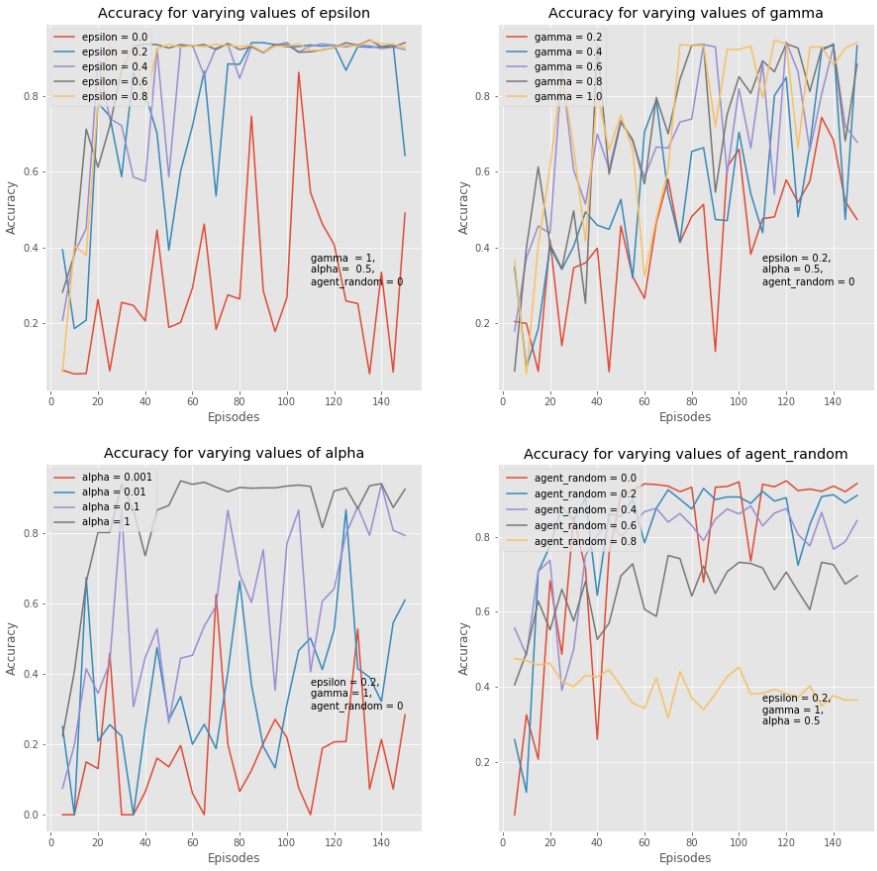

Figure 9: Accuracy plots for varying hyper-parameters - Deep Q-learning

Again, behaviours very similar to Q-learning were observed in figure 9. The only noticable difference are in the accuracy curves which were more sporadic here, this can be attributed to the random manner with which batches are pulled from set D for the neural network to train on. However these steep accents and descents diminishes with time as the the network gets to train on more recent data is available. And again as expected the accuracy of the agent falls as the probability of its movements shifts from greedy to random.

4.4 Learning from Random Environments

Until now the environment on which the Q-learning or Deep Q-learning algorithms were run, was the same during train and test time. In this section we'll present results for when the algorithms are run on different environments at every episode during train time and then tested on a new environment during test time.

4.4.1 Q-learning Experimentation

Here a similar training and testing structure as in the case with fixed environment Q-learning was employed.

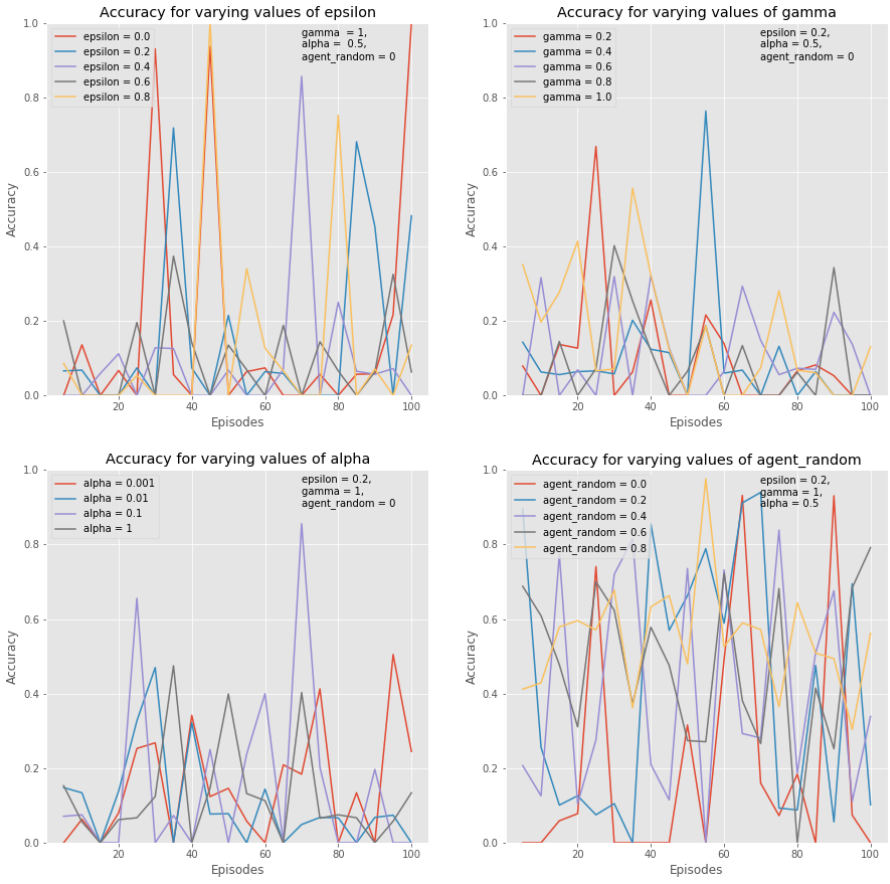

Figure 10: Accuracy plots for varying hyper-parameters - Random Q-learning

As expected, the results are sporadic and poor. The poor results can be attributed to the fact that the agent trains on different environments during each episode. The changing environments as a consequence doesn't allow for the convergent of the Q-table values. And at the end of the training session we are left with a fruitless Q-table which is not a general representation of all possible environments. This goes to show in the fourth plot (agent_random) when the agent accuracy is considerably higher when it is allowed to move randomly with a higher probability.

4.4.2 Deep Q-learning Experimentation

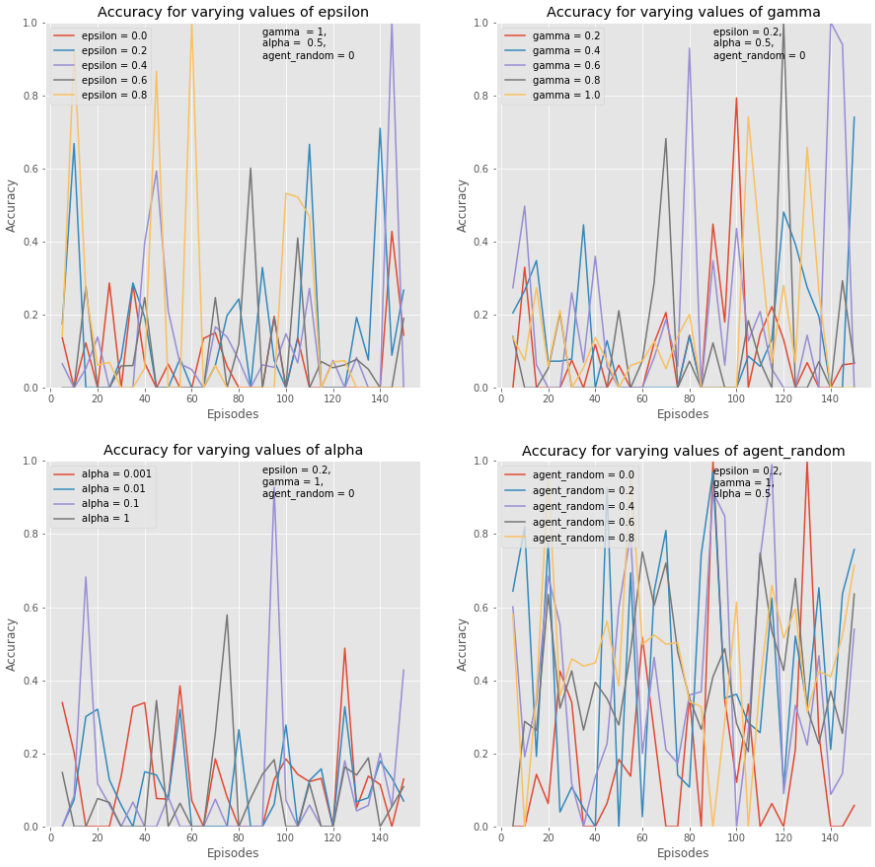

Figure 11: Accuracy plots for varying hyper-parameters - Random DQ-learning

Here too a similar training and testing structure as in the case with fixed environment Deep Q-learning was employed. And as expected the results are poor as the algorithm is not able to generalize well to any given environment.

Unfortunately both Q-learning and Deep Q-learning aren't able to generalize well to changing environments and a different approach or algorithm would have to be used to remedy this.

5 Conclusions

With the second lab session we got to explore two new and highly sought after sub fields within the machine learning space namely, Online Learning and Reinforcement Learning. The assignment got us to fiddle around with these new topics and understand them at their very roots. All in all, yet another enlightening experience, the fruits of which we hope to use to further our research, studies and contributions within the Machine Learning space.

References

[1] Habrard Amaury. Reinforcement learning bandits. 2018.

[2] Silver David. Reinforcement learning. 2015.

[3] Li Fei-Fei. Reinforcement learning. 2017.