CMU 10-715: Homework 8 Report

Online Learning and Q Learning

Abishek Sridhar (Andrew Id: abisheks)

1 Online Learning

1.1 Results

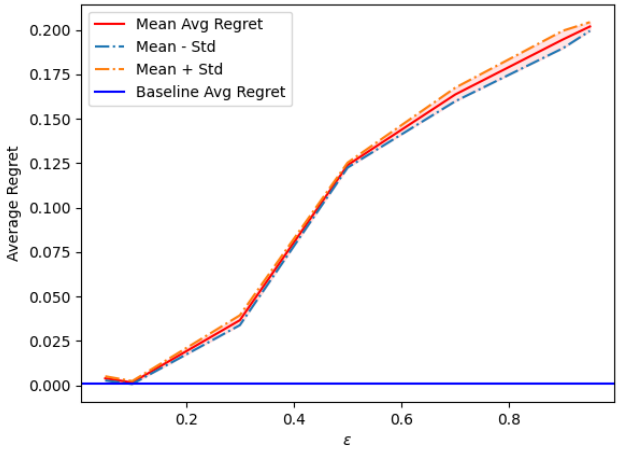
a I coded the weighted majority algorithm for online learning and attached the code in 3.

b The best baseline expert so far is gotten by predicting the same label as the expert with minimum regret till the given time instant **t**. Ties are broken by choosing the minimum expert index. Also, I run the online algorithm with 6 different seeds (0, 100, 200, 300, 400, 500) for every $\epsilon \in \varepsilon$ to get slightly better mean and standard deviation estimates. The plots are obtained in figure 1, where the plot on the right is plotted with a log scale y-axis to see the lines clearly for low values of ϵ. (For $\epsilon > 0.5$, the weights were under-flowing and in that case I randomly sample the expert to predict)

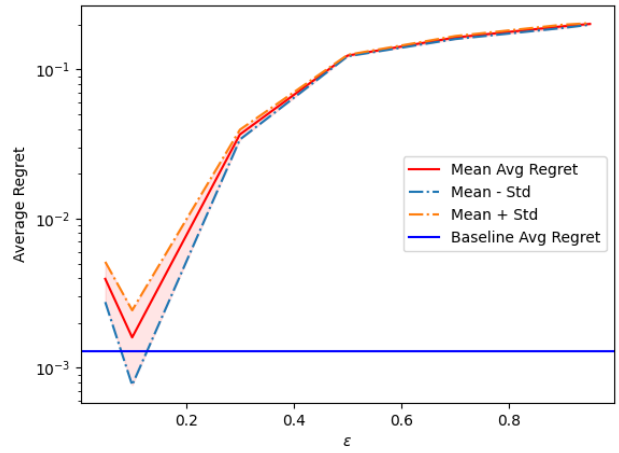
Analysis: The mean average regret of the randomized weighted majority algorithm reduces initially and rises with further increase in ϵ and the best performance (lowest regret) is achieved at $\epsilon = 0.1$, where the mean - std curve goes below the greedy baseline approach. Hence, there is a trade-off of regret wrt ϵ running at the back. This trend of average regret of the algorithm intuitively might be because in a setting where no expert is individually too good at predictions (which is usually the case of online algorithm with adversarially chosen truth labels; here, the average regret of the best individual expert for the entire time period = 0.251), higher values of ϵ will quickly bring down the trust on one expert (say it gives wrong predictions in the beginning and gradually gives more accurate predictions as we proceed) with a few wrong predictions, which need to be compensated by other experts achieving the same number of wrong predictions before its probability reaches a level where the expert might be chosen again $\rightarrow$ can be thought of over-reliance on exploitation. Additionally, we can observe high standard deviations for high values of ϵ in figure 1 and 2.

c I modified the algorithm by normalizing the weights at the beginning of each iteration to solve the values underflowing problem and obtain the plot 3.

Here, we observe that the average regret reduces with increase in ϵ, and kind of saturates with minor variations in the region of high values of ϵ



(a) Average Regret vs ϵ



(b) Average Regret vs ϵ with Y-axis in log scale

Figure 1: Variation of average regret of randomized weighted majority algorithm with ϵ and comparison with a greedy best expert so far baseline (with no modification to the algorithm)

(probably due to randomness). This can be explained by noting that we don't penalize the wrong experts much for small values of ϵ, hence the

```
For eps:  0.05 | mean-regret:   0.0039 | std-regret:   0.0012 | baseline regret:   0.0013
For eps:   0.1 | mean-regret:   0.0016 | std-regret:   0.0008 | baseline regret:   0.0013
For eps:   0.3 | mean-regret:   0.0367 | std-regret:   0.0028 | baseline regret:   0.0013
For eps:   0.5 | mean-regret:   0.1240 | std-regret:   0.0014 | baseline regret:   0.0013
For eps:   0.7 | mean-regret:   0.1637 | std-regret:   0.0038 | baseline regret:   0.0013
For eps:   0.9 | mean-regret:   0.1947 | std-regret:   0.0049 | baseline regret:   0.0013
For eps:  0.95 | mean-regret:   0.2019 | std-regret:   0.0025 | baseline regret:   0.0013
```

Figure 2: Exact numbers for the online algorithm with different ϵ's for Online Algorithm with no modification

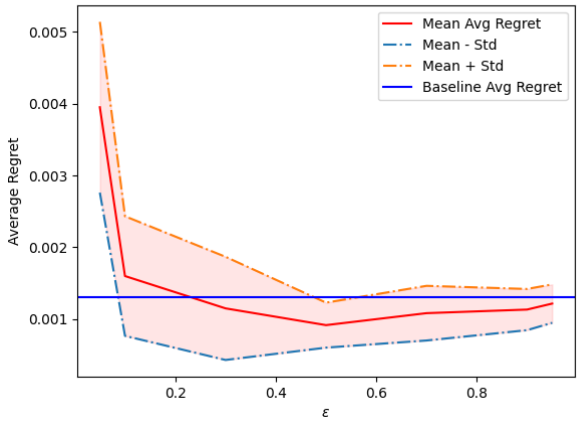


Figure 3: Variation of average regret of randomized weighted majority algorithm with ϵ and comparison with a greedy best expert so far baseline (with normalization of weights at the beginning of each iteration in the algorithm)

regret of the algorithm stays above the baseline. But for higher values of epsilon the wrong experts are penalized enough to cause the algorithm to choose the best experts only at each iteration and be better than the baseline.

2 Q Learning

2.1 Results

a I ran the Q Learning algorithm with ϵ-greedy sampling for the default hyperparameters ($\alpha = 0.8$, $\gamma = 0.9$, $\epsilon = 0.7$) and added the code in 3.

b An episode of the game for each $\epsilon \in \{0, 0.25, 0.5\}$ for the same Q values learned from the algorithm is shown in figures 4, 5, and 6 respectively.

The episode for $\epsilon = 0$ indicates the agent reaching the goal in minimal steps via a neat, direct path and achieving a discounted total reward of

0.5905. This suggests that the Q Learning algorithm has indeed learned the optimal policy quite well, because with a different seed while running Q Learning algorithm, we also get an episode like 7. Figure 7 is actually truncated because the agent is stuck at the first cell, trying to move left (leads to out of the grid), and this continues indefinitely since there is no exploration in this case $\implies$ total discounted reward = 0. Though this issue will be resolved if the Q Learning algorithm learned from a lot more trajectories, and in fact even for the given number of iterations in the problem, the Q Learning does learn better actions depending on the random seed, this episode points at an important shortcoming of choosing $\epsilon = 0$: a sub-optimal/wrong policy for any state might result in no hope for the agent to recover and reach the goal since there is no exploration part.

The episode for $\epsilon = 0.25$ shows the agent taking reaching the goal in 7 steps (suboptimal number of steps compared to $\epsilon = 0$) and obtaining a discounted rewards of 0.5314. The agent first tries to move up from the first cell (leads to out of grid), accounting for the additional step in this case, and then proceeds to take the same route as with the $\epsilon = 0$ case to reach the final goal. Though the agent does take more steps to reach the final goal, a good thing with a small exploration is that the agent will be able to recover from an indefinite state if the optimal policy is not learned. Ofcourse as a trade-off, there is also a small chance for the agent to take a further de-tour or fall into the water as part of the exploration. But by keeping ϵ small, we would be able to reach the goal in minimal or sub-minimal steps with a high probability and also deter the effect of a wrong policy learned for any state.

The episode for $\epsilon = 0.5$ shows the agent taking a long de-tour, visiting the same cells at times, seeming to progress towards the goal after the de-tour, but suddenly falling into water and obtaining reward = 0. The same con as mentioned for $\epsilon = 0.25$ applies here, but the difference is that due to the high weightage on exploration, the reliance on the policy learned from the algorithm comes down, and there is high chance for the agent to not reach the end goal (or fail). There is greater chance to escape from wrong learned policy, but it is severely outweighed by the con of exploration in this setting. This explains the episode: the agent tried to go up like in $\epsilon = 0.25$, then seems to come down based on the policy learned, again explores and takes a de-tour to cell at row 1 and column 2, then seems to proceed towards the goal by taking the easier route from that cell, but falls into water due to one more exploration.

Conclusion: The exploration part is essential during initial stages of training the Q Learning algorithm, to visit new cells and attempt learning the optimal policy for them. But during prediction, the ϵ should be kept

low (lower with increasing number of iterations the algorithm is trained for) to trust the policy learned from the algorithm more and not result in unnecessary de-tours or failures. But maybe an $\epsilon = 0$ is a bad choice because we can't train till convergence in practice, and need some chance to get out of indefinite wrong actions.

(a) State 0

(b) State 1

(c) State 2

(d) State 3

(e) State 4

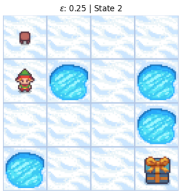
(f) State 5

(g) State 6

Figure 4: State transitions of one episode of the game with $\epsilon = 0$

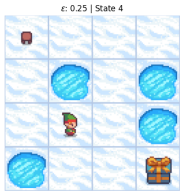
(a) State 0

(b) State 1

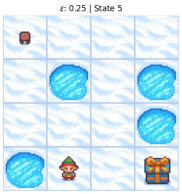


(c) State 2

(d) State 3



(e) State 4



(f) State 5

(g) State 6

(h) State 7

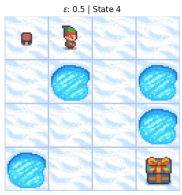
Figure 5: State transitions of one episode of the game with $\epsilon = 0.25$

(a) State 0

(b) State 1

(c) State 2

(d) State 3



(e) State 4

(f) State 5

(g) State 6

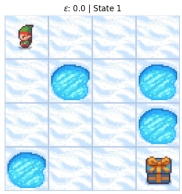
(h) State 7

(i) State 8

(j) State 9

Figure 6: State transitions of one episode of the game with $\epsilon = 0.5$

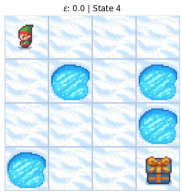
(a) State 0



(b) State 1

(c) State 2

(d) State 3



(e) State 4

(f) State 5

(g) State 6

(h) State 7

Figure 7: State transitions of a failed episode of the game with $\epsilon = 0$

3 Code

online_learning.py

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

class OnlineLearning:
    def __init__(self, eps, d, seed=0):
        self._seed = seed
        self.eps = eps
        self._d = d
        self.w = np.ones((d,))

    def fit(self, x, y_true):
        pred = np.random.choice(x, p=self.w/np.sum(self.w) if np.sum(self.w) > 0 else None)
        self.w -= self.eps * np.not_equal(x, y_true) * self.w
        return pred

    def fit_baseline(self, X_prev, Y_prev, x_cur):
        if len(X_prev) == 0:
            loss = np.zeros((self._d,))
        else:
            loss = np.sum(X_prev != Y_prev.reshape(len(Y_prev), 1), axis=0)
        return x_cur[np.argmin(loss)]

    def fit_horizon(self, X, Y_true, baseline=False):
        if isinstance(X, pd.DataFrame):
            X = X.to_numpy()
            Y_true = Y_true.to_numpy()
        T = len(X)
        self.predictions = []
        self.baseline_preds = []
        np.random.seed(self._seed)
        for i in range(T):
            self.w /= np.sum(self.w) # Normalizing (comment if not needed)
            self.predictions.append(self.fit(X[i], Y_true[i]))
            if baseline:
                self.baseline_preds.append(self.fit_baseline(X[:i], Y_true[:i], X[i]))

        self.predictions = np.array(self.predictions)
        self.baseline_preds = np.array(self.baseline_preds)
```

```
        self.calc_regret(X, Y_true, baseline)
        return self.predictions

    def calc_regret(self, X, Y_true, baseline):
        self.regret = np.mean(np.not_equal(self.predictions, Y_true))
        best_expert_regret = np.min(np.mean(X != Y_true.reshape((len(Y_true),
        ↪  1)), axis=0))
        self.regret -= best_expert_regret
        if baseline:
            self.baseline_regret = np.mean(np.not_equal(self.baseline_preds,
            ↪  Y_true))
            self.baseline_regret -= best_expert_regret

def plot(eps_set, mean, std, best_expert_regret):
    plt.figure(0)
    plt.plot(eps_set, mean, c='r', label='Mean Avg Regret')
    plt.plot(eps_set, np.maximum(mean-std, 0), linestyle='-.', label='Mean -
    ↪  Std')
    plt.plot(eps_set, mean+std, linestyle='-.', label='Mean + Std')
    plt.fill_between(eps_set, np.maximum(mean-std, 0), mean+std, color='r',
    ↪  alpha=.1)
    plt.axhline(y=best_expert_regret, color='b', linestyle='-',
    ↪  label='Baseline Avg Regret')
    plt.xlabel(r'$\epsilon$')
    plt.ylabel('Average Regret')
    plt.legend(loc='best')
    plt.savefig('regret.png')
    plt.show()
    plt.close()

if __name__ == '__main__':
    df = pd.read_csv('online_data.csv', sep=', ', index_col='T')
    X_train = df.loc[:, df.columns != 'Label']
    Y_train = df['Label']
    regret_mean, regret_std, baseline_regret = [], [], None
    eps_set = [0.05, 0.1, 0.3, 0.5, 0.7, 0.9, 0.95]
    for eps in eps_set:
        regrets = []
        b_regrets = []
        for seed in [0, 100, 200, 300, 400, 500]:
            algo = OnlineLearning(eps, 10, seed)
            algo.fit_horizon(X_train, Y_train, True if baseline_regret is
            ↪  None else False)
            regrets.append(algo.regret)
```

```
            if baseline_regret is None:
                b_regrets.append(algo.baseline_regret)

        regret_mean.append(np.mean(regrets))
        regret_std.append(np.std(regrets))
        if baseline_regret is None:
            baseline_regret = np.mean(b_regrets)
        print(f'For eps: {eps: >5} | mean-regret: {regret_mean[-1]: >8.4f} | '
              f'std-regret: {regret_std[-1]: >8.4f} | baseline regret:
               ↪  {baseline_regret: >8.4f}')

    plot(eps_set, np.array(regret_mean), np.array(regret_std),
     ↪  baseline_regret)
    X = X_train.to_numpy()
    Y_true = Y_train.to_numpy()
    best_expert_regret = np.min(np.mean(X != Y_true.reshape((len(Y_true),
     ↪  1)), axis=0))
    print(f'Best Expert Average Regret (at the end): {best_expert_regret}')
```

qlearning.py

```
import gym
import numpy as np
import random
import matplotlib.pyplot as plt
import os
import sys

MAX_ITERATIONS = 500
ENV = "FrozenLake-v1"


def qlearning(env=ENV, epsilon=0.7, alpha=0.8, gamma=0.9):
    env = gym.make(env, is_slippery=False, render_mode="rgb_array")

    na = env.action_space.n
    ns = env.observation_space.n
    Q = np.zeros((ns, na))
    random.seed(42)
    for i in range(MAX_ITERATIONS):
        # TODO: write your code here to update Q
        cur_state, _ = env.reset()
        term = False
```

```
        while not term:
            # eps-greedy sampling of action
            best_action = np.argmax(Q[cur_state])
            if random.random() <= epsilon: # Exploration
                cur_act = random.randint(0, na-1)
            else: # Exploitation
                cur_act = best_action
            next_state, rew, term, _, _ = env.step(cur_act)

            # Update step
            Q[cur_state, cur_act] = (1 - alpha)*Q[cur_state, cur_act] +
            ↪  alpha*(rew + gamma*np.max(Q[next_state]))
            cur_state = next_state

    env.close()
    return Q

def plot(state_image, idx, epsilon):
    plt.figure()
    plt.imshow(state_image)
    plt.axis('off')
    plt.title(rf'$\epsilon$: {epsilon} | State {idx}')
    if not os.path.exists(f'./ql_plots/eps_{epsilon}'):
        os.makedirs(f'./ql_plots/eps_{epsilon}')
    plt.savefig(f'./ql_plots/eps_{epsilon}/state_{idx}.png')
    plt.close()

def generate_trajectory(Q, epsilon=0.7, env=ENV, gamma=0.9, seed=2022):
    env = gym.make(env, is_slippery=False, render_mode="rgb_array")
    cur_state, _ = env.reset()
    na = env.action_space.n
    idx = 0
    plot(env.render(), idx, epsilon)
    term, tot_rewards, discount = False, 0, 1
    random.seed(seed)
    while not term and idx < 10:
        # idx < 10 to prevent agent stuck scenario to run indefinitely (when
        ↪  eps = 0 mainly)
        idx += 1
        best_action = np.argmax(Q[cur_state])
        if random.random() <= epsilon:  # Exploration
            cur_act = random.randint(0, na - 1)
        else:  # Exploitation
            cur_act = best_action
```

```
65
66          next_state, rew, term, _, _ = env.step(cur_act)
67          plot(env.render(), idx, epsilon)
68          print(cur_state, cur_act, next_state, rew, term)
69
70          cur_state = next_state
71          tot_rewards += discount*rew
72          discount *= gamma
73
74      env.close()
75      print(f'Total reward for the trajectory: {tot_rewards}')
76
77  if __name__ == '__main__':
78      # Train Q with Q-learning
79      Q = qlearning()
80      # TODO: write your code here to evaluate and render environment
81      for epsilon in [0, 0.25, 0.5]:
82          generate_trajectory(Q, epsilon=epsilon, seed=67)
```