15015762_RL_hw1

February 20, 2018

1 RL homework 1

Due date: 19 February 2017, 11:55pm (just before mid-night!)

1.1 Name: Yifan Shen

1.2 ID: 15015762

1.3 How to submit

When you have completed the exercises and everything has finsihed running, click on 'File' in the menu-bar and then 'Download .ipynb'. This file must be submitted to Moodle named as studentnumber_DL_hw2.ipynb before the deadline above.

Also send a sharable link to the notebook at the following email: ucl.coursework.submit@gmail.com. You can also make it sharable via link to everyone, up to you.

1.4 Context

In this assignment, we will take a first look at learning decisions from data. For this, we will use the multi-armed bandit framework.

1.5 Background reading

- Sutton and Barto (2018), Chapters 1 and 2

2 The Assignment

2.0.1 Objectives

You will use Python to implement several bandit algorithms.

You will then run these algorithms on a multi-armed Bernoulli bandit problem, to understand the issue of balancing exploration and exploitation.

3 Setup

3.0.1 Import Useful Libraries

```
In [0]: import tensorflow as tf
        import matplotlib.pyplot as plt
        import numpy as np
        from collections import namedtuple
```

3.0.2 Set options

```
In [0]: np.set_printoptions(precision=3, suppress=1)
        plt.style.use('seaborn-notebook')
```

3.0.3 A generic multi-armed bandit class, with Bernoulli rewards

```
In [0]: class BernoulliBandit(object):
          """A stationary multi-armed Bernoulli bandit."""

          def __init__(self, success_probabilities, success_reward=1., fail_reward=0.):
            """Constructor of a stationary Bernoulli bandit.

            Args:
              success_probabilities: A list or numpy array containing the probabilities,
                  for each of the arms, of providing a success reward.
              success_reward: The reward on success (default: 1.)
              fail_reward: The reward on failure (default: 0.)
            """
            self._probs = success_probabilities
            self._number_of_arms = len(self._probs)
            self._s = success_reward
            self._f = fail_reward

          def step(self, action):
            """The step function.

            Args:
              action: An integer or tf.int32 that specifies which arm to pull.

            Returns:
              A sampled reward according to the success probability of the selected arm.

            Raises:
              ValueError: when the provided action is out of bounds.
            """
            if action < 0 or action >= self._number_of_arms:
              raise ValueError('Action {} is out of bounds for a '
                               '{}-armed bandit'.format(action, self._number_of_arms))
```

```
        success = bool(np.random.random() < self._probs[action])
        reward = success * self._s + (not success) * self._f
        return reward
```

3.0.4 Helper functions

```
In [0]: def smooth(array, smoothing_horizon=100., initial_value=0.):
          """smoothing function for plotting"""
          smoothed_array = []
          value = initial_value
          b = 1./smoothing_horizon
          m = 1.
          for x in array:
            m *= 1. - b
            lr = b/(1 - m)
            value += lr*(x - value)
            smoothed_array.append(value)
          return np.array(smoothed_array)

        def one_hot(array, depth):
          """Multi-dimensional one-hot"""
          a = np.array(array)
          x = a.flatten()
          b = np.eye(depth)[x, :depth]
          return b.reshape(a.shape + (depth,))

        def plot(algs, plot_data, optimal_value, repetitions=30):
          """Plot results of a bandit experiment."""
          algs_per_row = 3
          n_algs = len(algs)
          n_rows = (n_algs - 2)//algs_per_row + 1
          fig = plt.figure(figsize=(15, 5*n_rows))

          for i, p in enumerate(plot_data):
            for c in range(n_rows):
              ax = fig.add_subplot(n_rows, len(plot_data), i + 1 + c*len(plot_data))
              ax.grid(0)
              ax.set_axis_bgcolor('white')

              current_algs = [algs[0]] + algs[c*algs_per_row + 1:(c + 1)*algs_per_row + 1]
              for alg in current_algs:
                data = p.data[alg.name]
                m = smooth(np.mean(data, axis=0))
                s = np.std(smooth(data.T).T, axis=0)/np.sqrt(repetitions)
                if p.log_plot:
                  line = plt.semilogy(m, alpha=0.6, label=alg.name)[0]
                else:
                  line = plt.plot(m, alpha=0.6, label=alg.name)[0]
```

```
            plt.fill_between(range(number_of_steps), m + s, m - s,
                             color=line.get_color(), alpha=0.2)
        if not p.log_plot:
          plt.plot([0, number_of_steps], [optimal_value]*2, '--k', label='optimal')

        plt.ylim(p.ylim)
        plt.title(p.title)
        if i == len(plot_data) - 1:
          plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

def run_experiment(bandit, algs, repetitions):
  """Run multiple repetitions of a bandit experiment."""
  reward_dict = {}
  action_dict = {}

  for alg in algs:
    reward_dict[alg.name] = []
    action_dict[alg.name] = []

    for _ in range(repetitions):
      alg.reset()
      reward_dict[alg.name].append([])
      action_dict[alg.name].append([])
      action = None
      reward = None
      for i in range(number_of_steps):
        try:
          action = alg.step(action, reward)
        except:
          print(alg, action, reward)
          aoushd()
        reward = bandit.step(action)
        reward_dict[alg.name][-1].append(reward)
        action_dict[alg.name][-1].append(action)

  return reward_dict, action_dict

def train_agents(agents, number_of_arms, number_of_steps, repetitions=30,
                 success_reward=1., fail_reward=0.):

  success_probabilities = np.arange(0.25, 0.75 + 1e-6, 0.5/(number_of_arms - 1))
  bandit = BernoulliBandit(success_probabilities, success_reward, fail_reward)

  max_p = np.max(success_probabilities)
  min_p = np.min(success_probabilities)
  ylim_max = max_p*success_reward + (1 - max_p)*fail_reward
  ylim_min = min_p*success_reward + (1 - min_p)*fail_reward
  dif = ylim_max - ylim_min
```

```
ylim = (ylim_min - 0.1*dif, ylim_max + 0.1*dif)

max_prob = np.max(success_probabilities)
optimal_value = max_prob*success_reward + (1 - max_prob)*fail_reward

reward_dict, action_dict = run_experiment(bandit, agents, repetitions)

smoothed_rewards = {}
expected_rewards = {}
regrets = {}
for agent, rewards in reward_dict.items():
  smoothed_rewards[agent] = np.array(rewards)
for agent, actions in action_dict.items():
  p_success = one_hot(actions, number_of_arms).dot(success_probabilities)
  expected_rewards[agent] = p_success*success_reward + (1 - p_success)*fail_reward
  regrets[agent] = optimal_value - expected_rewards[agent]

PlotData = namedtuple('PlotData', ['title', 'data', 'log_plot', 'ylim'])
plot_data = [
    PlotData(title='Smoohted rewards', data=smoothed_rewards,
             log_plot=False, ylim=ylim),
    PlotData(title='Expected rewards', data=expected_rewards,
             log_plot=False, ylim=ylim),
    PlotData(title='Current Regret', data=regrets,
             log_plot=True, ylim=(1e-2, 1)),
    PlotData(title='Total Regret',
             data=dict([(k, np.cumsum(v, axis=1)) for k, v in regrets.items()]),
             log_plot=False, ylim=(1e-0, 5e2)),
]

plot(agents, plot_data, optimal_value)
```

3.0.5 Random agent

```
In [0]: class Random(object):
          """A random agent.

          This agent returns an action between 0 and 'number_of_arms',
          uniformly at random. The 'previous_action' argument of 'step'
          is ignored.
          """

          def __init__(self, number_of_arms):
            self._number_of_arms = number_of_arms
            self.name = 'random'
            self.reset()

          def step(self, previous_action, reward):
```

```
#     optimal = 4
#     return int(optimal)
    return np.random.randint(self._number_of_arms)

  def reset(self):
    pass
```

4 A1: Implement agents

Each agent, should implement the following methods:

4.0.1 `step(self, previous_action, reward):`

should update the statistics by updating the value for the previous_action towards the observed reward.

(Note: make sure this can handle the case that previous_action=None, in which case no statistics should be updated.)

(Hint: you can split this into two steps: 1. update values, 2. get new action. Make sure you update the values before selecting a new action.)

4.0.2 `reset(self):`

resets statistics (should be equivalent to constructing a new agent from scratch). Make sure that the initial values (after a reset) are all zero.

4.0.3 `__init__(self):`

The `__init__` can be *the same* as for the random agent above (with the exception of ϵ-greedy---see below), except for the name, which should be unique (e.g., 'greedy', 'ucb', etc.)

All agents should be in pure Python - so you cannot use TensorFlow to, e.g., compute gradients. Using `numpy` is fine.

4.1 Agent 1: greedy

[10 pts] You should **implement the greedy** agent, that

1. Estimates the average reward for each action that was selected so far, and
2. Always selects the highest-valued action.

```
In [0]: class Greedy(object):

          def __init__(self, number_of_arms):
            self._number_of_arms = number_of_arms
            self.name = 'greedy'
            self.reset()

          def step(self, previous_action, reward):
            # previous_action: an integer from 0 to 4
```

```
    # reward: previous_reward: 0 or 1

    if previous_action == None:
      current_action = np.random.randint(self._number_of_arms)
      return current_action
    else:
      self.Nt[previous_action] += 1
      self.value_store[previous_action] += reward
      self.prob_store[previous_action] = \
            self.value_store[previous_action]/self.Nt[previous_action]
      current_action = np.argmax(self.prob_store)

      return current_action

  def reset(self):
    # stores the information from past for future decisions
    self.value_store = np.zeros(self._number_of_arms)
    self.prob_store = np.zeros(self._number_of_arms)
    self.Nt = np.zeros(self._number_of_arms) + 0.001
```

4.2 Agent 2: ϵ-greedy

[15 pts] You should **implement an ϵ-greedy** agent, that selects the highest-valued action with probability $1 - \epsilon$, and otherwise selects an action at random.

The exploration parameter ϵ should be given in the `__init__`, as indicated in the code below.

```
In [0]: class EpsilonGreedy(object):

  def __init__(self, number_of_arms, epsilon=0.1):
    self._number_of_arms = number_of_arms
    self._epsilon = epsilon
    self.name = 'epsilon-greedy epsilon:{}'.format(epsilon)
    self.reset()

  def step(self, previous_action, reward):
    # previous_action: an integer from 0 to 4
    # reward: previous_reward: 0 or 1

    if previous_action == None:
      return np.random.randint(self._number_of_arms)
    else:

      prob = np.random.uniform(0,1)

      if prob < self._epsilon:
        return np.random.randint(self._number_of_arms)
      else:
        self.Nt[previous_action] += 1
```

```
            self.value_store[previous_action] += reward
            self.prob_store[previous_action] = \
                 self.value_store[previous_action]/self.Nt[previous_action]
            current_action = np.argmax(self.prob_store)
            return current_action



        def reset(self):
          # stores the information from past for future decisions
          self.value_store = np.zeros(self._number_of_arms)
          self.prob_store = np.zeros(self._number_of_arms)
          self.Nt = np.zeros(self._number_of_arms) + 0.001
```

4.3 Agent 3: UCB

[15 pt] You should **implement** an agent that **explores with UCB**.

```
In [0]: class UCB(object):

        def __init__(self, number_of_arms):
          self._number_of_arms = number_of_arms
          self.name = 'ucb'
          self.reset()

        def step(self, previous_action, reward):

          if previous_action == None:
            return np.random.randint(self._number_of_arms)
          else:
            self.value_store[previous_action] += reward
            self.Nt[previous_action] += 1
            self.prob_store[previous_action] = \
                      self.value_store[previous_action]/self.Nt[previous_action]

            current_action = np.argmax(self.prob_store +
                                       np.sqrt(np.log(np.sum(self.Nt))/self.Nt))

            return current_action

        def reset(self):
          # stores the information from past for future decisions
          self.value_store = np.zeros(self._number_of_arms)
          self.prob_store = np.zeros(self._number_of_arms)
          self.Nt = np.zeros(self._number_of_arms) + 0.001
```

4.4 Agent 4: REINFORCE agents

You should implement agents that implement REINFORCE policy-gradient methods

The policy should be a softmax on action preferences:

$$\pi(a) = \frac{\exp(p(a))}{\sum_b \exp(p(b))}\,.$$

The action preferences are stored separately, so that for each action a the preference $p(a)$ is a single value that you directly update.

4.5 Assignment 4a:

In the next text field, write down the update function to the preferences for all actions $\{a_1, \ldots, a_n\}$ if you selected a specific action $A_t = a_i$ and received a reward of R_t.

In other words, complete:

$$\begin{aligned} p_{t+1}(a) &= \ldots && \text{for } a = A_t \\ p_{t+1}(b) &= \ldots && \text{for all } b \neq A_t \end{aligned}$$

[10 pts] Instructions: please provide answer in markdown below.

$$\begin{aligned} p_{t+1}(a) &= p_t(a) + \alpha R_t\big(1 - \pi_t(a)\big) && \text{for } a = A_t \\ p_{t+1}(b) &= p_t(b) - \alpha R_t \pi_t(a) && \text{for all } b \neq A_t \end{aligned}$$

4.6 Assignment 4b:

[15 + 5 pts] You should **implement a vanilla REINFORCE agent with** and **without a baseline**. Whether or not a baseline is used should be a boolean constructor argument.

```
In [0]: class REINFORCE(object):

          def __init__(self, number_of_arms, step_size=0.1, baseline=False):
            self._number_of_arms = number_of_arms
            self._lr = step_size
            self.name = 'reinforce, baseline: {}'.format(baseline)
            self._baseline = baseline
            self.reset()

          def step(self, previous_action, reward):

            if previous_action == None:
              return np.random.randint(self._number_of_arms)
            else:
              self.N += 1
              self.value_sum += reward

              b = 0
              if self._baseline:
                b = self.value_sum / self.N
```

```
        # read previous pi from previous_action
        pi = self.pi_store[previous_action]

        # update action preference
        self.action_preference[previous_action] += self._lr *\
                                                (reward - b) * (1 - pi)
        for i in range(self._number_of_arms):
          if i != previous_action:
            self.action_preference[i] -= self._lr *(reward - b) *self.pi_store[i]

        # calculate new pi with softmax function
        exp_preference = np.exp(self.action_preference)
        self.pi_store = exp_preference / np.sum(exp_preference)

        current_action = np.random.choice(self._number_of_arms,p = self.pi_store)

        return current_action

    def reset(self):
      # stores the information from past for future decisions
      self.action_preference = np.zeros(self._number_of_arms)
      self.pi_store = np.zeros(self._number_of_arms)
      self.value_sum = 0
      self.N = 0
```

5 Assignment 5: Analyse Results

5.0.1 Run the cell below to train the agents and generate the plots for the first experiment.

Trains the agents on a Bernoulli bandit problem with 5 arms, with a reward on success of 1, and a reward on failure of 0.

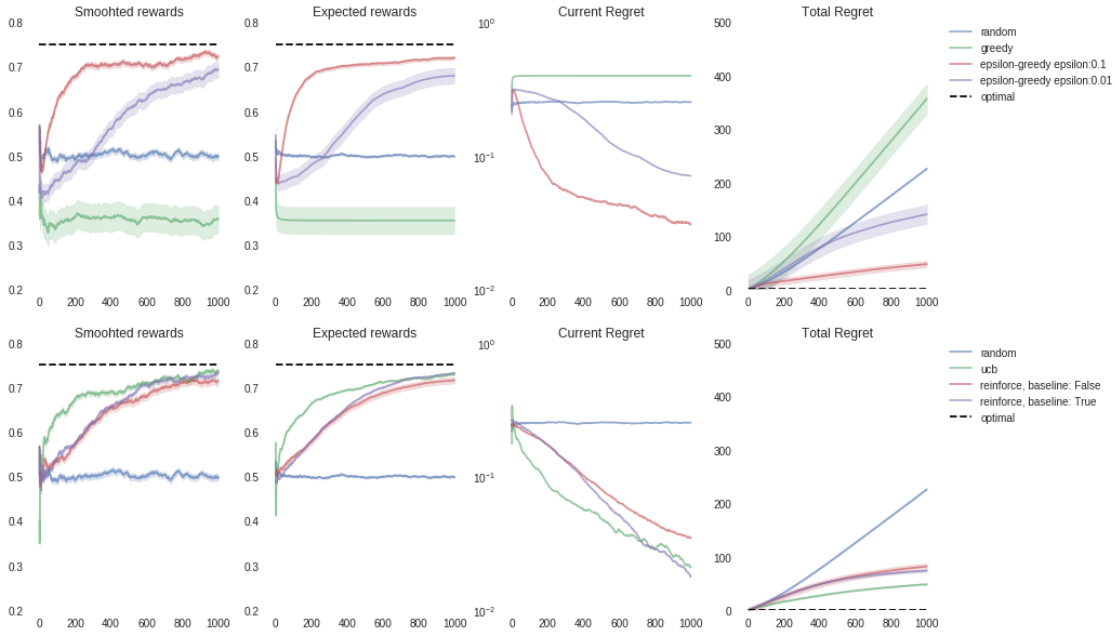
In [27]:
```
#@title Experiment 1: Bernoulli bandit

number_of_arms = 5
number_of_steps = 1000

agents = [
    Random(number_of_arms),
    Greedy(number_of_arms),
    EpsilonGreedy(number_of_arms, 0.1),
    EpsilonGreedy(number_of_arms, 0.01),
    UCB(number_of_arms),
    REINFORCE(number_of_arms),
    REINFORCE(number_of_arms, baseline=True),
]
```

```
train_agents(agents, number_of_arms, number_of_steps)
```

```
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:32: MatplotlibDeprecationWarning:
```



5.1 Assignment 5 a.

(Answer inline in the markdown below each question.)

[5pts] Name the best and worst algorithms, and explain (with one or two sentences each) why these are best and worst.

1. The best algorithm is the UCB algorithm. Reason: it has high rewards the lowest total regret.
2. The worest algorithm is the Greedy algorithm. Reason: it has the lowest rewards and highest regret, even worse then the random one. When running the greedy algorithm, it might stuck in the local optimum without exploring other better solutions.

[5pts] Which algorithms are guaranteed to have linear total regret?

Random and Greedy algorithm are guaranteed for linear total regret.

[5pts] Which algorithms are guaranteed to have logarithmic total regret?

UCB and both Reinforce algorithms are guaranteed for logarithmic total regret.

[5pts] Which of the ϵ-greedy algorithms performs best? Which should perform best in the long run?

In the short period run (1000 times), the 0.1 ϵ-greedy runs better. However, in the long run (say, 10000 times), the 0.01 ϵ-greedy will be better. As in the long run, 0.01 ϵ-greedy will approach the best strategy with minimum affect from exploring other random moves.

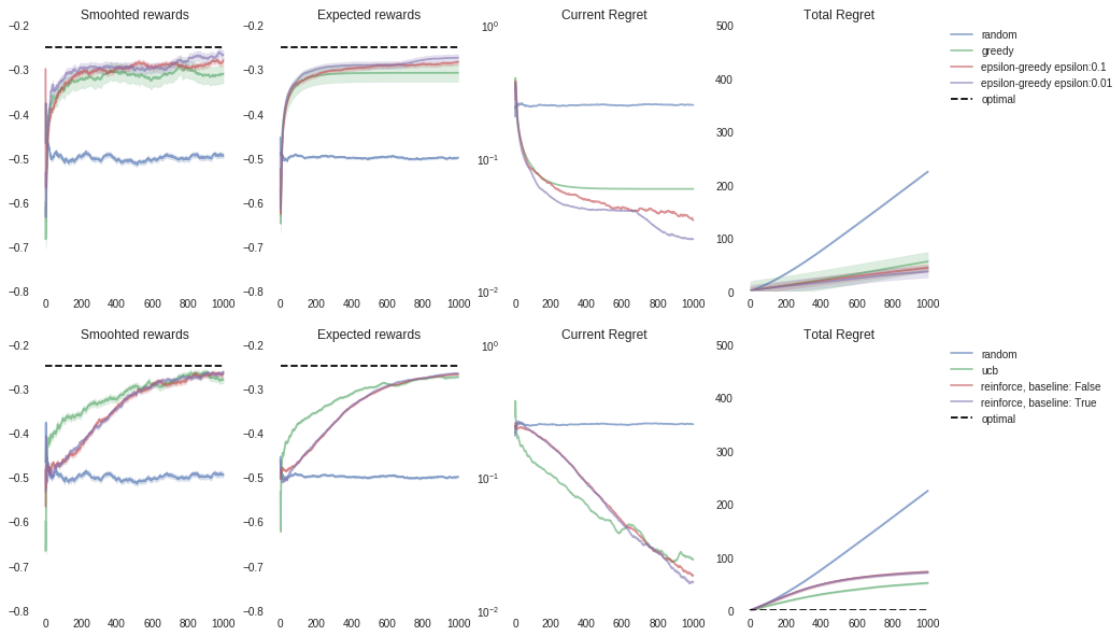
5.1.1 Run the cell below to train the agents and generate the plots for the second experiment.

Trains the agents on a bernoulli bandit problem with 5 arms, with a reward on success of 0, and a reward on failure of -1.

```
In [28]: #@title Experiment 2: R = 0 on success, R = -1 on failure.
         number_of_arms = 5
         number_of_steps = 1000

         train_agents(agents, number_of_arms, number_of_steps,
                      success_reward=0., fail_reward=-1.)
```

```
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:32: MatplotlibDeprecationWarning:
```



5.2 Assignment 5 b.

(Answer inline in the markdown.)

[10pts] Explain which algorithms improved from the changed rewards, and why.

(Use at most two sentences per algorithm and feel free to combine explanations for different algorithms where possible).

1. Greedy massively improved.

2. 0.01 ϵ-greedy improved.

Reason: As 0.01 ϵ-greedy is very close to Greedy, we analyse them together. We use np.argmax to choose the next decision, in the reward +1/0 case, we start with choosing the first arm, which

gives us some positive reward (although the worest), then the greedy algorithm stuck with the first arm. While in the 0/-1 case, the greedy algorithm starts with choosing the fisrt arm as well, which gives a quite significant negative reward, then the algorithm choose the second arm, which leads to slight better negative reward....finanlly the greedy algorithm is likely to find the optimal solution is choosing arm 5.