Assignment 1: Bandits and Exploration/Exploitation

Welcome to Assignment 1. This notebook will:

- Help you create your first bandit algorithm
- Help you understand the effect of epsilon on exploration and learn about the exploration/exploitation tradeoff
- Introduce you to some of the reinforcement learning software we are going to use for this specialization

This class uses RL-Glue to implement most of our experiments. It was originally designed by Adam White, Brian Tanner, and Rich Sutton. This library will give you a solid framework to understand how reinforcement learning experiments work and how to run your own. If it feels a little confusing at first, don't worry - we are going to walk you through it slowly and introduce you to more and more parts as you progress through the specialization.

We are assuming that you have used a Jupyter notebook before. But if not, it is quite simple. Simply press the run button, or shift+enter to run each of the cells. The places in the code that you need to fill in will be clearly marked for you.

Section 0: Preliminaries

```
In [ ]:  # Import necessary libraries
         %matplotlib inline
         import numpy as np
         import matplotlib.pyplot as plt
         from tqdm import tqdm
         import time

         from rlglue.rl_glue import RLGlue
         import main_agent
         import ten_arm_env
         import test_env
```

In the above cell, we import the libraries we need for this assignment. We use numpy throughout the course and occasionally provide hints for which methods to use in numpy. Other than that we mostly use vanilla python and the occasional other library, such as matplotlib for making plots.

You might have noticed that we import ten_arm_env. This is the **10-armed Testbed** introduced in section 2.3 of the textbook. We use this throughout this notebook to test our bandit agents. It has 10 arms, which are the actions the agent can take. Pulling an arm generates a stochastic reward from a Gaussian distribution with unit-variance. For each action, the expected value of that action is randomly sampled from a normal distribution, at

the start of each run. If you are unfamiliar with the 10-armed Testbed please review it in the textbook before continuing.

DO NOT IMPORT OTHER LIBRARIES as this will break the autograder.

DO NOT SET A RANDOM SEED as this will break the autograder.

Please **do not** duplicate cells. This will put your notebook into a bad state and break Cousera's autograder.

Before you submit, please click "Kernel" -> "Restart and Run All" and make sure all cells pass.

Section 1: Greedy Agent

We want to create an agent that will find the action with the highest expected reward. One way an agent could operate is to always choose the action with the highest value based on the agent's current estimates. This is called a greedy agent as it greedily chooses the action that it thinks has the highest value. Let's look at what happens in this case.

First we are going to implement the argmax function, which takes in a list of action values and returns an action with the highest value. Why are we implementing our own instead of using the argmax function that numpy uses? Numpy's argmax function returns the first instance of the highest value. We do not want that to happen as it biases the agent to choose a specific action in the case of ties. Instead we want to break ties between the highest values randomly. So we are going to implement our own argmax function. You may want to look at np.random.choice to randomly select from a list of values.

```python
In [ ]:  # -----------
         # Graded Cell
         # -----------
         def argmax(q_values):
             """
             Takes in a list of q_values and returns the index of the item
             with the highest value. Breaks ties randomly.
             returns: int - the index of the highest value in q_values
             """
             top_value = float("-inf")
             ties = []

             for i in range(len(q_values)):
                 # if a value in q_values is greater than the highest value update top a
                 # if a value is equal to top value add the index to ties
                 # return a random selection from ties.
                 # YOUR CODE HERE
         #        raise NotImplementedError()
                 if q_values[i] > top_value:
                     top_value = q_values[i]
                     ties = [i]
                 elif q_values[i] == top_value:
                     ties.append(i)
```

```
#            raise NotImplementedError()
      return np.random.choice(ties)
```

In [ ]:
```
# --------------
# Debugging Cell
# --------------
# Feel free to make any changes to this cell to debug your code

test_array = [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
# print(f"the highest number is {argmax(test_array)}")
assert argmax(test_array) == 8, "Check your argmax implementation returns the i

# make sure np.random.choice is called correctly
np.random.seed(0)
test_array = [1, 0, 0, 1]

assert argmax(test_array) == 0
```

In [ ]:
```
# -----------
# Tested Cell
# -----------
# The contents of the cell will be tested by the autograder.
# If they do not pass here, they will not pass there.

test_array = [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
assert argmax(test_array) == 8, "Check your argmax implementation returns the i

# set random seed so results are deterministic
np.random.seed(0)
test_array = [1, 0, 0, 1]

counts = [0, 0, 0, 0]
for _ in range(100):
    a = argmax(test_array)
    counts[a] += 1

# make sure argmax does not always choose first entry
assert counts[0] != 100, "Make sure your argmax implementation randomly chooose

# make sure argmax does not always choose last entry
assert counts[3] != 100, "Make sure your argmax implementation randomly chooose

# make sure the random number generator is called exactly once whenver `argmax`
expected = [44, 0, 0, 56] # <-- notice not perfectly uniform due to randomness
assert counts == expected
```

Now we introduce the first part of an RL-Glue agent that you will implement. Here we are going to create a GreedyAgent and implement the agent_step method. This method gets called each time the agent takes a step. The method has to return the action selected by the agent. This method also ensures the agent's estimates are updated based on the signals it gets from the environment.

Fill in the code below to implement a greedy agent.

In [ ]:
```
# -----------
# Graded Cell
# -----------
```

```python
class GreedyAgent(main_agent.Agent):
    def agent_step(self, reward, observation=None):
        """
        Takes one step for the agent. It takes in a reward and observation and
        returns the action the agent chooses at that time step.

        Arguments:
        reward -- float, the reward the agent recieved from the environment aft
        observation -- float, the observed state the agent is in. Do not worry
                        until future lessons
        Returns:
        current_action -- int, the action chosen by the agent at the current ti
        """
        ### Useful Class Variables ###
        # self.q_values : An array with what the agent believes each of the val
        # self.arm_count : An array with a count of the number of times each ar
        # self.last_action : The action that the agent took on the previous tim
        ####################

        # Update Q values Hint: Look at the algorithm in section 2.4 of the tex
        # increment the counter in self.arm_count for the action from the previ
        # update the step size using self.arm_count
        # update self.q_values for the action from the previous time step

        # YOUR CODE HERE
#           raise NotImplementedError()

        self.arm_count[self.last_action] = self.arm_count[self.last_action] + 1

        self.q_values[self.last_action] = self.q_values[self.last_action] + (1


        # current action = ? # Use the argmax function you created above
        # YOUR CODE HERE
#           raise NotImplementedError()
        current_action = argmax(self.q_values)

        self.last_action = current_action

        return current_action
```

```python
In [ ]:  # --------------
         # Debugging Cell
         # --------------
         # Feel free to make any changes to this cell to debug your code

         # build a fake agent for testing and set some initial conditions
         np.random.seed(1)
         greedy_agent = GreedyAgent()
         greedy_agent.q_values = [0, 0, 0.5, 0, 0]
         greedy_agent.arm_count = [0, 1, 0, 0, 0]
         greedy_agent.last_action = 1

         action = greedy_agent.agent_step(reward=1)

         # make sure the q_values were updated correctly
         assert greedy_agent.q_values == [0, 0.5, 0.5, 0, 0]
```

```
    # make sure the agent is using the argmax that breaks ties randomly
    assert action == 2
```

In [ ]:
```
# ------------
# Tested Cell
# ------------
# The contents of the cell will be tested by the autograder.
# If they do not pass here, they will not pass there.

# build a fake agent for testing and set some initial conditions
np.random.seed(1)
greedy_agent = GreedyAgent()
greedy_agent.q_values = [0, 0, 1.0, 0, 0]
greedy_agent.arm_count = [0, 1, 0, 0, 0]
greedy_agent.last_action = 1

# take a fake agent step
action = greedy_agent.agent_step(reward=1)

# make sure agent took greedy action
assert action == 2

# make sure q_values were updated correctly
assert greedy_agent.q_values == [0, 0.5, 1.0, 0, 0]

# take another step
action = greedy_agent.agent_step(reward=2)
assert action == 2
assert greedy_agent.q_values == [0, 0.5, 2.0, 0, 0]
```

Let's visualize the result. Here we run an experiment using RL-Glue to test our agent. For now, we will set up the experiment code; in future lessons, we will walk you through running experiments so that you can create your own.

In [ ]:
```
# ----------------
# Discussion Cell
# ----------------

num_runs = 200                          # The number of times we run the experiment
num_steps = 1000                        # The number of pulls of each arm the agent t
env = ten_arm_env.Environment          # We set what environment we want to use to t
agent = GreedyAgent                    # We choose what agent we want to use
agent_info = {"num_actions": 10}       # We pass the agent the information it needs.
env_info = {}                          # We pass the environment the information it

rewards = np.zeros((num_runs, num_steps))
average_best = 0
for run in tqdm(range(num_runs)):            # tqdm is what creates the progress
    np.random.seed(run)

    rl_glue = RLGlue(env, agent)           # Creates a new RLGlue experiment wit
    rl_glue.rl_init(agent_info, env_info)  # We pass RLGlue what it needs to ini
    rl_glue.rl_start()                     # We start the experiment

    average_best += np.max(rl_glue.environment.arms)

    for i in range(num_steps):
        reward, _, action, _ = rl_glue.rl_step()  # The environment and agent ta
```

```
                                                        # the reward, and action taken
        rewards[run, i] = reward

greedy_scores = np.mean(rewards, axis=0)
plt.figure(figsize=(15, 5), dpi= 80, facecolor='w', edgecolor='k')
plt.plot([average_best / num_runs for _ in range(num_steps)], linestyle="--")
plt.plot(greedy_scores)
plt.legend(["Best Possible", "Greedy"])
plt.title("Average Reward of Greedy Agent")
plt.xlabel("Steps")
plt.ylabel("Average reward")
plt.show()
```

```
100%|███████████| 200/200 [00:02<00:00, 68.73it/s]
```
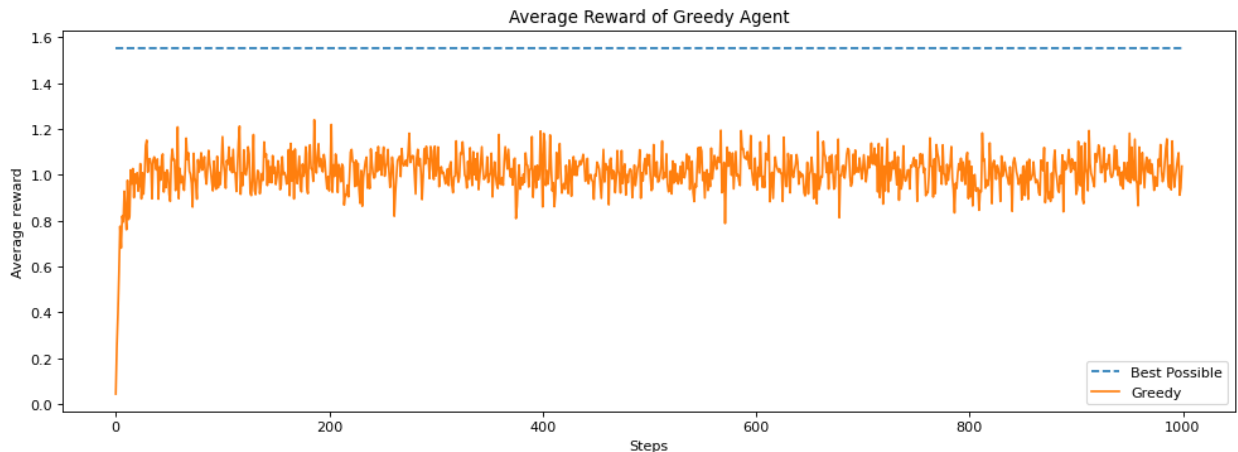

How did our agent do? Is it possible for it to do better?

Section 2: Epsilon-Greedy Agent

We learned about another way for an agent to operate, where it does not always take the greedy action. Instead, sometimes it takes an exploratory action. It does this so that it can find out what the best action really is. If we always choose what we think is the current best action is, we may miss out on taking the true best action, because we haven't explored enough times to find that best action.

Implement an epsilon-greedy agent below. Hint: we are implementing the algorithm from section 2.4 of the textbook. You may want to use your greedy code from above and look at np.random.random, as well as np.random.randint, to help you select random actions.

In [ ]:
```
# -----------
# Graded Cell
# -----------
class EpsilonGreedyAgent(main_agent.Agent):
    def agent_step(self, reward, observation):
        """
        Takes one step for the agent. It takes in a reward and observation and
        returns the action the agent chooses at that time step.

        Arguments:
        reward -- float, the reward the agent recieved from the environment aft
```

```
            observation -- float, the observed state the agent is in. Do not worry
                            until future lessons
        Returns:
        current_action -- int, the action chosen by the agent at the current ti
        """


        ### Useful Class Variables ###
        # self.q_values : An array with what the agent believes each of the va
        # self.arm_count : An array with a count of the number of times each a
        # self.last_action : The action that the agent took on the previous tim
        # self.epsilon : The probability an epsilon greedy agent will explore (
        #####################

        # Update Q values - this should be the same update as your greedy agent
        # YOUR CODE HERE
#           raise NotImplementedError()
        self.arm_count[self.last_action] += 1
        self.q_values[self.last_action] = self.q_values[self.last_action] + (1

        # Choose action using epsilon greedy
        # Randomly choose a number between 0 and 1 and see if it's less than se
        # (hint: look at np.random.random()). If it is, set current_action to a
        # otherwise choose current_action greedily as you did above.
        # YOUR CODE HERE
#           raise NotImplementedError()
        rand_num = np.random.random()
        if self.epsilon > rand_num:
            current_action = np.random.choice(range(0, len(self.q_values)), 1)[
        else:
            current_action = argmax(self.q_values)

        self.last_action = current_action

        return current_action
```

In [ ]:
```
# --------------
# Debugging Cell
# --------------
# Feel free to make any changes to this cell to debug your code

# build a fake agent for testing and set some initial conditions
np.random.seed(0)
e_greedy_agent = EpsilonGreedyAgent()
e_greedy_agent.q_values = [0, 0.0, 0.5, 0, 0]
e_greedy_agent.arm_count = [0, 1, 0, 0, 0]
e_greedy_agent.num_actions = 5
e_greedy_agent.last_action = 1
e_greedy_agent.epsilon = 0.5

# given this random seed, we should see a greedy action (action 2) here
action = e_greedy_agent.agent_step(reward=1, observation=0)

# ------------------------------------------------
# we'll try to guess a few of the trickier places
# ------------------------------------------------

# make sure to update for the *last_action* not the current action
assert e_greedy_agent.q_values != [0, 0.5, 1.0, 0, 0], "A"
```

```
    # make sure the stepsize is based on the *last_action* not the current action
    assert e_greedy_agent.q_values != [0, 1, 0.5, 0, 0], "B"

    # make sure the agent is using the argmax that breaks ties randomly
    assert action == 2, "C"


    # ------------------------------------------------

    # let's see what happens for another action
    np.random.seed(1)
    e_greedy_agent = EpsilonGreedyAgent()
    e_greedy_agent.q_values = [0, 0.5, 0.5, 0, 0]
    e_greedy_agent.arm_count = [0, 1, 0, 0, 0]
    e_greedy_agent.num_actions = 5
    e_greedy_agent.last_action = 1
    e_greedy_agent.epsilon = 0.5

    # given this random seed, we should see a random action (action 4) here
    action = e_greedy_agent.agent_step(reward=1, observation=0)

    # The agent saw a reward of 1, so should increase the value for *last_action*
    assert e_greedy_agent.q_values == [0, 0.75, 0.5, 0, 0], "D"

    # the agent should have picked a random action for this particular random seed
    assert action == 4, "E"
```

```
In [ ]:  # -----------
         # Tested Cell
         # -----------
         # The contents of the cell will be tested by the autograder.
         # If they do not pass here, they will not pass there.

         np.random.seed(0)
         e_greedy_agent = EpsilonGreedyAgent()
         e_greedy_agent.q_values = [0, 0, 1.0, 0, 0]
         e_greedy_agent.arm_count = [0, 1, 0, 0, 0]
         e_greedy_agent.num_actions = 5
         e_greedy_agent.last_action = 1
         e_greedy_agent.epsilon = 0.5
         action = e_greedy_agent.agent_step(reward=1, observation=0)

         assert e_greedy_agent.q_values == [0, 0.5, 1.0, 0, 0]

         # manipulate the random seed so the agent takes a random action
         np.random.seed(1)
         action = e_greedy_agent.agent_step(reward=0, observation=0)

         assert action == 4

         # check to make sure we update value for action 4
         action = e_greedy_agent.agent_step(reward=1, observation=0)
         assert e_greedy_agent.q_values == [0, 0.5, 0.0, 0, 1.0]
```

Now that we have our epsilon greedy agent created. Let's compare it against the greedy agent with epsilon of 0.1.

```
In [ ]:  # ----------------
         # Discussion Cell
```

```python
# ----------------

# Plot Epsilon greedy results and greedy results
num_runs = 200
num_steps = 1000
epsilon = 0.1
agent = EpsilonGreedyAgent
env = ten_arm_env.Environment
agent_info = {"num_actions": 10, "epsilon": epsilon}
env_info = {}
all_rewards = np.zeros((num_runs, num_steps))

for run in tqdm(range(num_runs)):
    np.random.seed(run)

    rl_glue = RLGlue(env, agent)
    rl_glue.rl_init(agent_info, env_info)
    rl_glue.rl_start()

    for i in range(num_steps):
        reward, _, action, _ = rl_glue.rl_step() # The environment and agent ta
                                                 # the reward, and action taken

        all_rewards[run, i] = reward

# take the mean over runs
scores = np.mean(all_rewards, axis=0)
plt.figure(figsize=(15, 5), dpi= 80, facecolor='w', edgecolor='k')
plt.plot([1.55 for _ in range(num_steps)], linestyle="--")
plt.plot(greedy_scores)
plt.title("Average Reward of Greedy Agent vs. E-Greedy Agent")
plt.plot(scores)
plt.legend(("Best Possible", "Greedy", "Epsilon: 0.1"))
plt.xlabel("Steps")
plt.ylabel("Average reward")
plt.show()
```
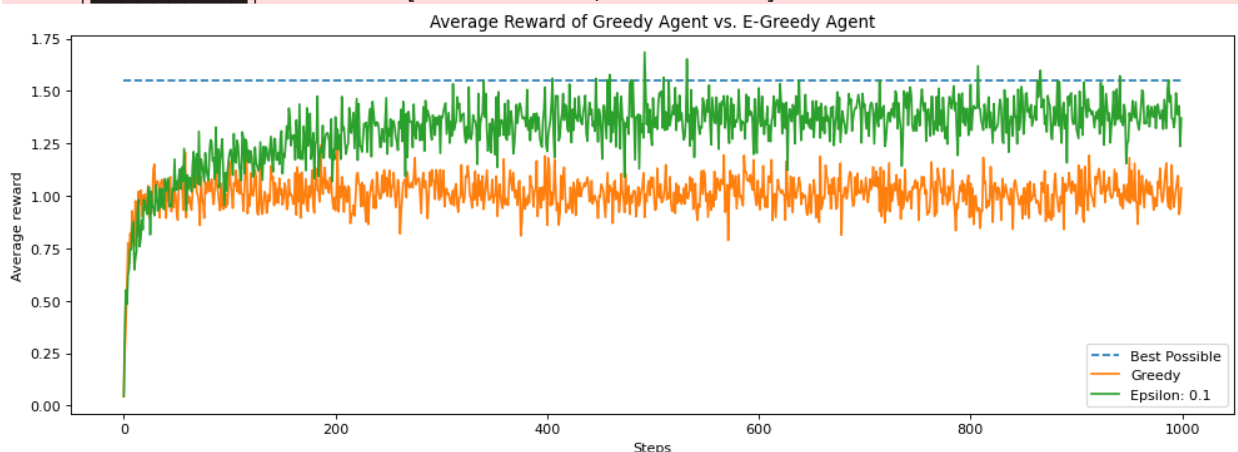
```
100%|████████████| 200/200 [00:03<00:00, 52.75it/s]
```


Notice how much better the epsilon-greedy agent did. Because we occasionally choose a random action we were able to find a better long term policy. By acting greedily before our value estimates are accurate, we risk settling on a suboptimal action.

Section 2.1 Averaging Multiple Runs

Did you notice that we averaged over 200 runs? Why did we do that?

To get some insight, let's look at the results of two individual runs by the same agent.

```python
In [ ]:  # ----------------
         # Discussion Cell
         # ----------------

         # Plot runs of e-greedy agent
         agent = EpsilonGreedyAgent
         env = ten_arm_env.Environment
         agent_info = {"num_actions": 10, "epsilon": 0.1}
         env_info = {}
         all_averages = []
         plt.figure(figsize=(15, 5), dpi= 80, facecolor='w', edgecolor='k')
         num_steps = 1000

         for run in (0, 1):
             np.random.seed(run) # Here we set the seed so that we can compare two diffe
             averages = []
             rl_glue = RLGlue(env, agent)
             rl_glue.rl_init(agent_info, env_info)
             rl_glue.rl_start()

             scores = [0]
             for i in range(num_steps):
                 reward, state, action, is_terminal = rl_glue.rl_step()
                 scores.append(scores[-1] + reward)
                 averages.append(scores[-1] / (i + 1))

             plt.plot(averages)

         plt.title("Comparing two independent runs")
         plt.xlabel("Steps")
         plt.ylabel("Average reward")
         plt.show()
```
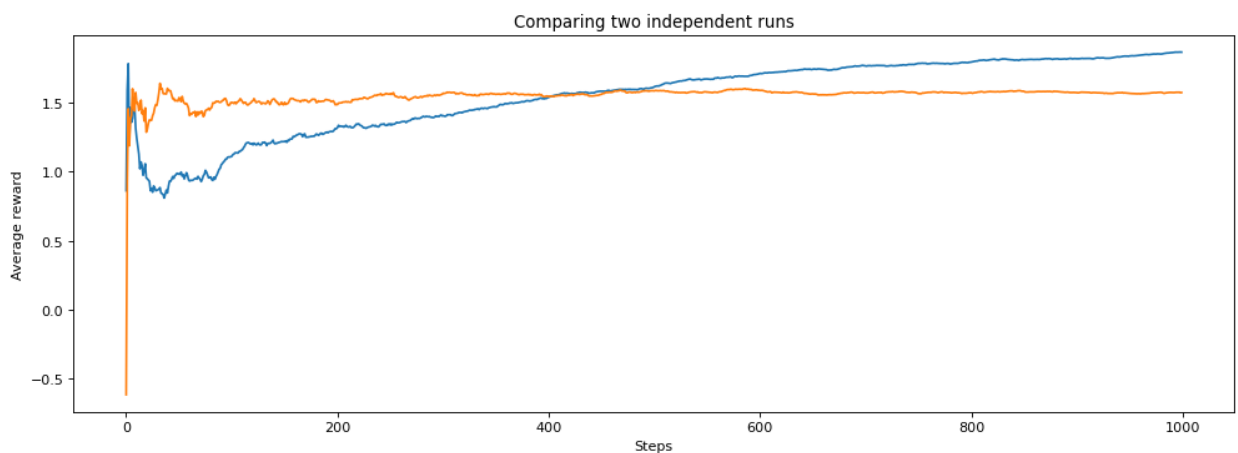

Notice how the two runs were different? But, if this is the exact same algorithm, why does it behave differently in these two runs?

The answer is that it is due to randomness in the environment and in the agent. Depending on what action the agent randomly starts with, or when it randomly chooses to explore, it

can change the results of the runs. And even if the agent chooses the same action, the reward from the environment is randomly sampled from a Gaussian. The agent could get lucky, and see larger rewards for the best action early on and so settle on the best action faster. Or, it could get unlucky and see smaller rewards for best action early on and so take longer to recognize that it is in fact the best action.

To be more concrete, let's look at how many times an exploratory action is taken, for different seeds.

```python
# ----------------
# Discussion Cell
# ----------------
print("Random Seed 1")
np.random.seed(1)
for _ in range(15):
    if np.random.random() < 0.1:
        print("Exploratory Action")


print()
print()

print("Random Seed 2")
np.random.seed(2)
for _ in range(15):
    if np.random.random() < 0.1:
        print("Exploratory Action")
```

```
Random Seed 1
Exploratory Action
Exploratory Action
Exploratory Action


Random Seed 2
Exploratory Action
```

With the first seed, we take an exploratory action three times out of 15, but with the second, we only take an exploratory action once. This can significantly affect the performance of our agent because the amount of exploration has changed significantly.

To compare algorithms, we therefore report performance averaged across many runs. We do this to ensure that we are not simply reporting a result that is due to stochasticity, as explained in the lectures. Rather, we want statistically significant outcomes. We will not use statistical significance tests in this course. Instead, because we have access to simulators for our experiments, we use the simpler strategy of running for a large number of runs and ensuring that the confidence intervals do not overlap.

Section 3: Comparing values of epsilon

Can we do better than an epsilon of 0.1? Let's try several different values for epsilon and see how they perform. We try different settings of key performance parameters to understand

how the agent might perform under different conditions.

Below we run an experiment where we sweep over different values for epsilon:

In [ ]:
```python
# ---------------
# Discussion Cell
# ---------------

# Experiment code for different e-greedy
epsilons = [0.0, 0.01, 0.1, 0.4]

plt.figure(figsize=(15, 5), dpi= 80, facecolor='w', edgecolor='k')
plt.plot([1.55 for _ in range(num_steps)], linestyle="--")

n_q_values = []
n_averages = []
n_best_actions = []

num_runs = 200

for epsilon in epsilons:
    all_averages = []
    for run in tqdm(range(num_runs)):
        agent = EpsilonGreedyAgent
        agent_info = {"num_actions": 10, "epsilon": epsilon}
        env_info = {"random_seed": run}

        rl_glue = RLGlue(env, agent)
        rl_glue.rl_init(agent_info, env_info)
        rl_glue.rl_start()

        best_arm = np.argmax(rl_glue.environment.arms)

        scores = [0]
        averages = []
        best_action_chosen = []

        for i in range(num_steps):
            reward, state, action, is_terminal = rl_glue.rl_step()
            scores.append(scores[-1] + reward)
            averages.append(scores[-1] / (i + 1))
            if action == best_arm:
                best_action_chosen.append(1)
            else:
                best_action_chosen.append(0)
            if epsilon == 0.1 and run == 0:
                n_q_values.append(np.copy(rl_glue.agent.q_values))
        if epsilon == 0.1:
            n_averages.append(averages)
            n_best_actions.append(best_action_chosen)
        all_averages.append(averages)

    plt.plot(np.mean(all_averages, axis=0))

plt.legend(["Best Possible"] + epsilons)
plt.xlabel("Steps")
plt.ylabel("Average reward")
plt.show()
```
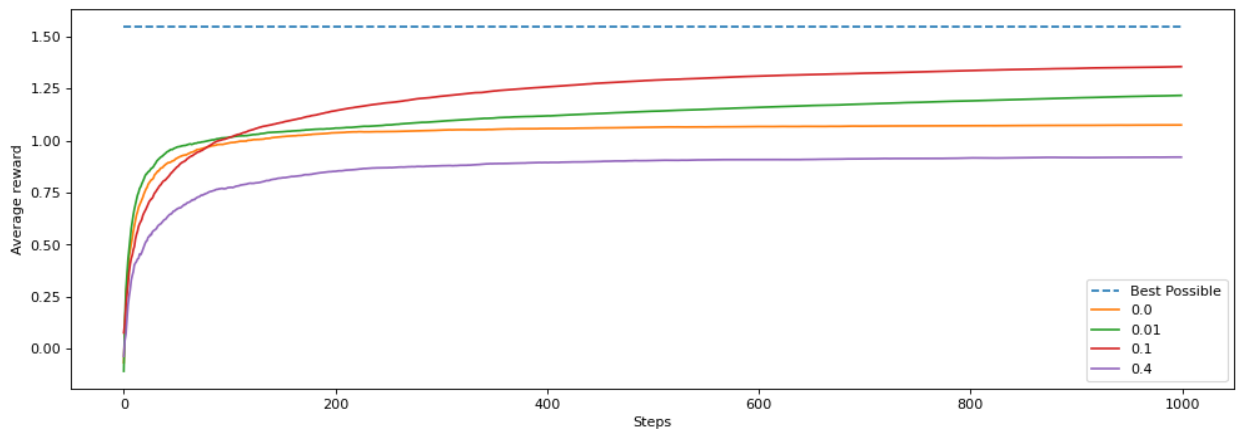
Why did 0.1 perform better than 0.01?

If exploration helps why did 0.4 perform worse that 0.0 (the greedy agent)?

Think about these and how you would answer these questions. They are questions in the practice quiz. If you still have questions about it, retake the practice quiz.

Section 4: The Effect of Step Size

In Section 1 of this assignment, we decayed the step size over time based on action-selection counts. The step-size was 1/N(A), where N(A) is the number of times action A was selected. This is the same as computing a sample average. We could also set the step size to be a constant value, such as 0.1. What would be the effect of doing that? And is it better to use a constant or the sample average method?

To investigate this question, let's start by creating a new agent that has a constant step size. This will be nearly identical to the agent created above. You will use the same code to select the epsilon-greedy action. You will change the update to have a constant step size instead of using the 1/N(A) update.

```python
In [ ]:  # -----------
         # Graded Cell
         # -----------
         class EpsilonGreedyAgentConstantStepsize(main_agent.Agent):
             def agent_step(self, reward, observation):
                 """
                 Takes one step for the agent. It takes in a reward and observation and
                 returns the action the agent chooses at that time step.

                 Arguments:
                 reward -- float, the reward the agent recieved from the environment aft
                 observation -- float, the observed state the agent is in. Do not worry
                                 until future lessons
                 Returns:
                 current_action -- int, the action chosen by the agent at the current ti
```

```
        """

        ### Useful Class Variables ###
        # self.q_values : An array with what the agent believes each of the va]
        # self.arm_count : An array with a count of the number of times each a]
        # self.last_action : An int of the action that the agent took on the p]
        # self.step_size : A float which is the current step size for the agent
        # self.epsilon : The probability an epsilon greedy agent will explore (
        #######################

        # Update q_values for action taken at previous time step
        # using self.step_size intead of using self.arm_count
        # YOUR CODE HERE
#            raise NotImplementedError()
        self.q_values[self.last_action] = self.q_values[self.last_action] + sel

        # Choose action using epsilon greedy. This is the same as you implement
        # YOUR CODE HERE
#            raise NotImplementedError()
        rand_num = np.random.random()
        if self.epsilon > rand_num:
            current_action = np.random.choice(range(0, len(self.q_values)), 1)[
        else:
            current_action = argmax(self.q_values)

        self.last_action = current_action

        return current_action
```

```
In [ ]: # --------------
        # Debugging Cell
        # --------------
        # Feel free to make any changes to this cell to debug your code

        for step_size in [0.01, 0.1, 0.5, 1.0]:
            e_greedy_agent = EpsilonGreedyAgentConstantStepsize()
            e_greedy_agent.q_values = [0, 0, 1.0, 0, 0]
            e_greedy_agent.num_actions = 5
            e_greedy_agent.last_action = 1
            e_greedy_agent.epsilon = 0.0
            e_greedy_agent.step_size = step_size
            action = e_greedy_agent.agent_step(1, 0)
            assert e_greedy_agent.q_values == [0, step_size, 1.0, 0, 0], "Check that yo
```

```
In [ ]: # -----------
        # Tested Cell
        # -----------
        # The contents of the cell will be tested by the autograder.
        # If they do not pass here, they will not pass there.

        np.random.seed(0)
        # Check Epsilon Greedy with Different Constant Stepsizes
        for step_size in [0.01, 0.1, 0.5, 1.0]:
            e_greedy_agent = EpsilonGreedyAgentConstantStepsize()
            e_greedy_agent.q_values = [0, 0, 1.0, 0, 0]
            e_greedy_agent.num_actions = 5
            e_greedy_agent.last_action = 1
            e_greedy_agent.epsilon = 0.0
            e_greedy_agent.step_size = step_size
```

```
        action = e_greedy_agent.agent_step(1, 0)

        assert e_greedy_agent.q_values == [0, step_size, 1.0, 0, 0]
```

In [ ]:
```python
# ---------------
# Discussion Cell
# ---------------

# Experiment code for different step sizes
step_sizes = [0.01, 0.1, 0.5, 1.0, '1/N(A)']

epsilon = 0.1
num_steps = 1000
num_runs = 200

fig, ax = plt.subplots(figsize=(15, 5), dpi= 80, facecolor='w', edgecolor='k')

q_values = {step_size: [] for step_size in step_sizes}
true_values = {step_size: None for step_size in step_sizes}
best_actions = {step_size: [] for step_size in step_sizes}

for step_size in step_sizes:
    all_averages = []
    for run in tqdm(range(num_runs)):
        np.random.seed(run)
        agent = EpsilonGreedyAgentConstantStepsize if step_size != '1/N(A)' els
        agent_info = {"num_actions": 10, "epsilon": epsilon, "step_size": step_
        env_info = {}

        rl_glue = RLGlue(env, agent)
        rl_glue.rl_init(agent_info, env_info)
        rl_glue.rl_start()

        best_arm = np.argmax(rl_glue.environment.arms)

        if run == 0:
            true_values[step_size] = np.copy(rl_glue.environment.arms)

        best_action_chosen = []
        for i in range(num_steps):
            reward, state, action, is_terminal = rl_glue.rl_step()
            if action == best_arm:
                best_action_chosen.append(1)
            else:
                best_action_chosen.append(0)
            if run == 0:
                q_values[step_size].append(np.copy(rl_glue.agent.q_values))
        best_actions[step_size].append(best_action_chosen)
    ax.plot(np.mean(best_actions[step_size], axis=0))

plt.legend(step_sizes)
plt.title("% Best Arm Pulled")
plt.xlabel("Steps")
plt.ylabel("% Best Arm Pulled")
vals = ax.get_yticks()
ax.set_yticklabels(['{:,.2%}'.format(x) for x in vals])
plt.show()
```
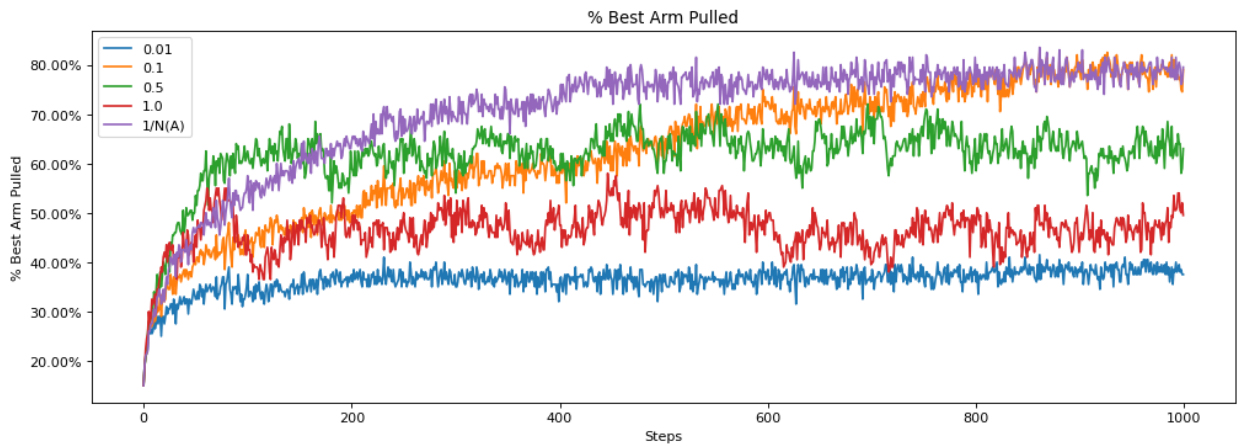
Notice first that we are now plotting the amount of time that the best action is taken rather than the average reward. To better understand the performance of an agent, it can be useful to measure specific behaviors, beyond just how much reward is accumulated. This measure indicates how close the agent's behaviour is to optimal.

It seems as though 1/N(A) performed better than the others, in that it reaches a solution where it takes the best action most frequently. Now why might this be? Why did a step size of 0.5 start out better but end up performing worse? Why did a step size of 0.01 perform so poorly?

Let's dig into this further below. Let's plot how well each agent tracks the true value, where each agent has a different step size method. You do not have to enter any code here, just follow along.

```
In [ ]:   # ---------------
          # Discussion Cell
          # ---------------
          largest = 0
          num_steps = 1000
          for step_size in step_sizes:
              plt.figure(figsize=(15, 5), dpi= 80, facecolor='w', edgecolor='k')
              largest = np.argmax(true_values[step_size])
              plt.plot([true_values[step_size][largest] for _ in range(num_steps)], lines
              plt.title("Step Size: {}".format(step_size))
              plt.plot(np.array(q_values[step_size])[:, largest])
              plt.legend(["True Expected Value", "Estimated Value"])
              plt.xlabel("Steps")
              plt.ylabel("Value")
              plt.show()
```
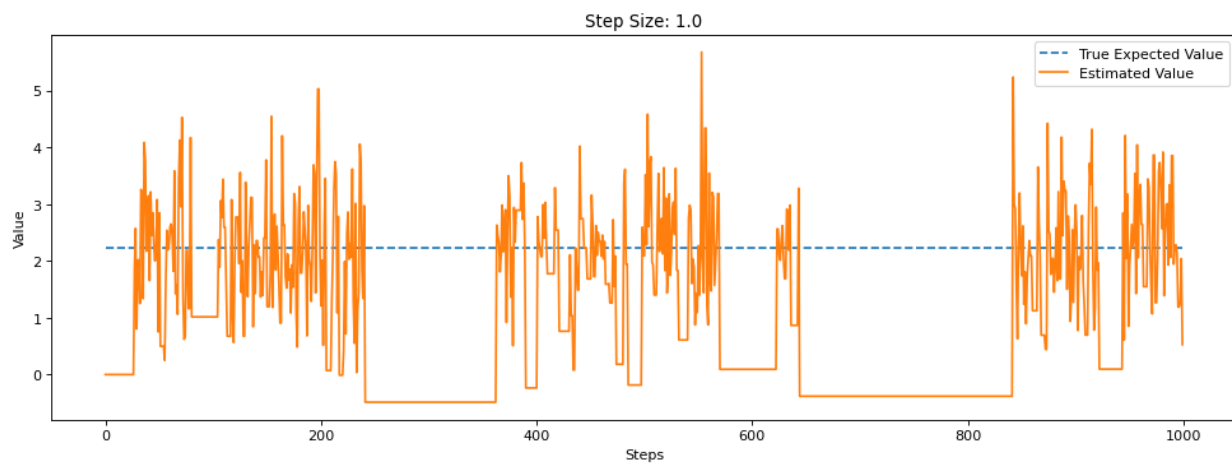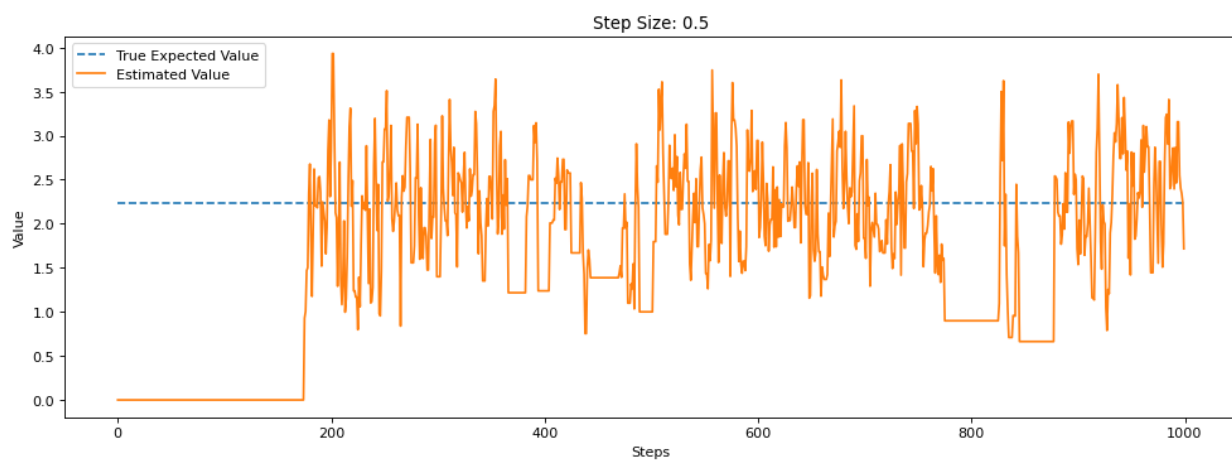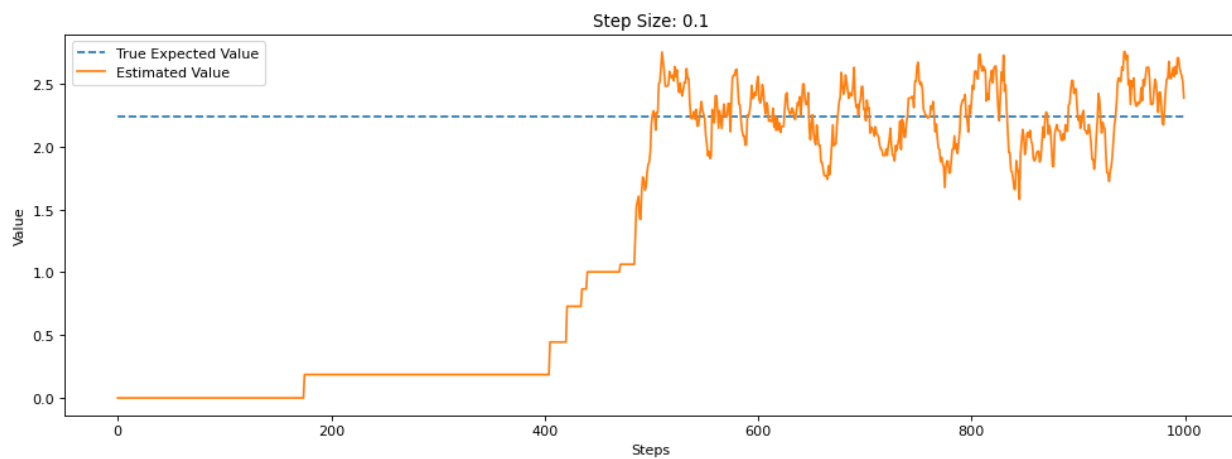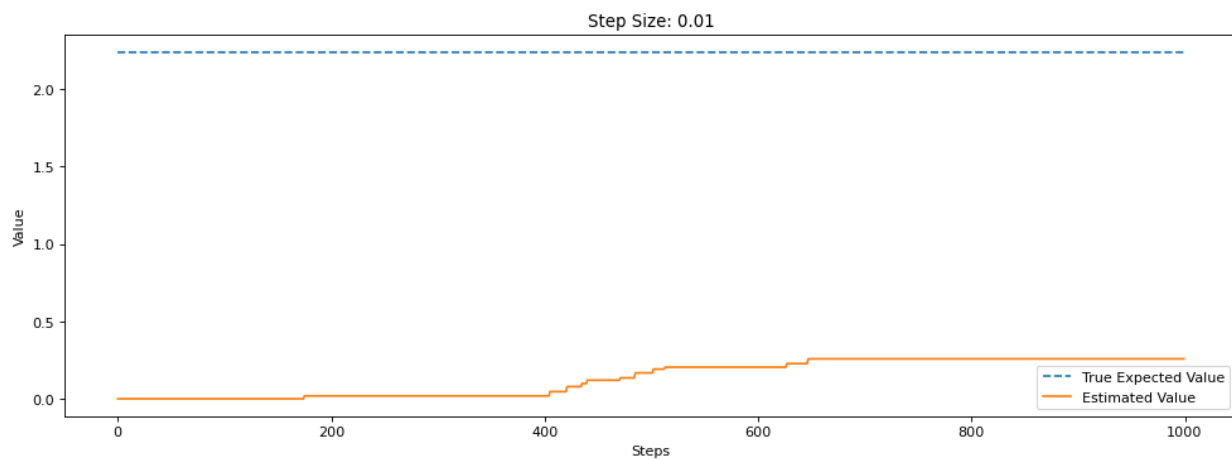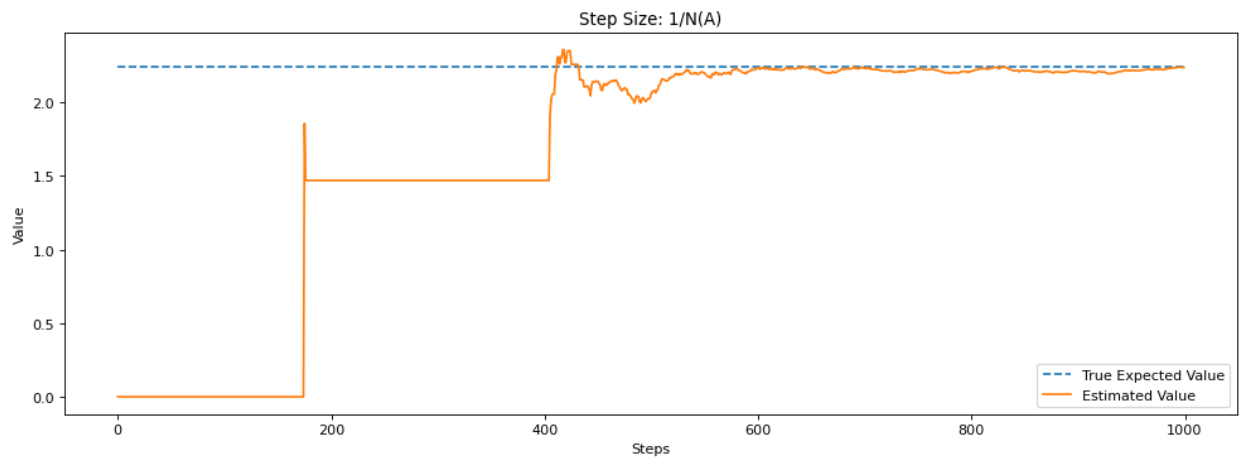
Step Size: 1/N(A)

These plots help clarify the performance differences between the different step sizes. A step size of 0.01 makes such small updates that the agent's value estimate of the best action does not get close to the actual value. Step sizes of 0.5 and 1.0 both get close to the true value quickly, but are very susceptible to stochasticity in the rewards. The updates overcorrect too much towards recent rewards, and so oscillate around the true value. This means that on many steps, the action that pulls the best arm may seem worse than it actually is. A step size of 0.1 updates fairly quickly to the true value, and does not oscillate as widely around the true values as 0.5 and 1.0. This is one of the reasons that 0.1 performs quite well. Finally we see why 1/N(A) performed well. Early on while the step size is still reasonably high it moves quickly to the true expected value, but as it gets pulled more its step size is reduced which makes it less susceptible to the stochasticity of the rewards.

Does this mean that 1/N(A) is always the best? When might it not be? One possible setting where it might not be as effective is in non-stationary problems. You learned about non-stationarity in the lessons. Non-stationarity means that the environment may change over time. This could manifest itself as continual change over time of the environment, or a sudden change in the environment.

Let's look at how a sudden change in the reward distributions affects a step size like 1/N(A). This time we will run the environment for 2000 steps, and after 1000 steps we will randomly change the expected value of all of the arms. We compare two agents, both using epsilon-greedy with epsilon = 0.1. One uses a constant step size of 0.1, the other a step size of 1/N(A) that reduces over time.

```
In [ ]:  # ---------------
         # Discussion Cell
         # ---------------
         epsilon = 0.1
         num_steps = 2000
         num_runs = 500
         step_size = 0.1

         plt.figure(figsize=(15, 5), dpi= 80, facecolor='w', edgecolor='k')
         plt.plot([1.55 for _ in range(num_steps)], linestyle="--")

         for agent in [EpsilonGreedyAgent, EpsilonGreedyAgentConstantStepsize]:
             rewards = np.zeros((num_runs, num_steps))
```

```
    for run in tqdm(range(num_runs)):
        agent_info = {"num_actions": 10, "epsilon": epsilon, "step_size": step_
        np.random.seed(run)

        rl_glue = RLGlue(env, agent)
        rl_glue.rl_init(agent_info, env_info)
        rl_glue.rl_start()

        for i in range(num_steps):
            reward, state, action, is_terminal = rl_glue.rl_step()
            rewards[run, i] = reward
            if i == 1000:
                rl_glue.environment.arms = np.random.randn(10)

    plt.plot(np.mean(rewards, axis=0))
plt.legend(["Best Possible", "1/N(A)", "0.1"])
plt.xlabel("Steps")
plt.ylabel("Average reward")
plt.show()
```
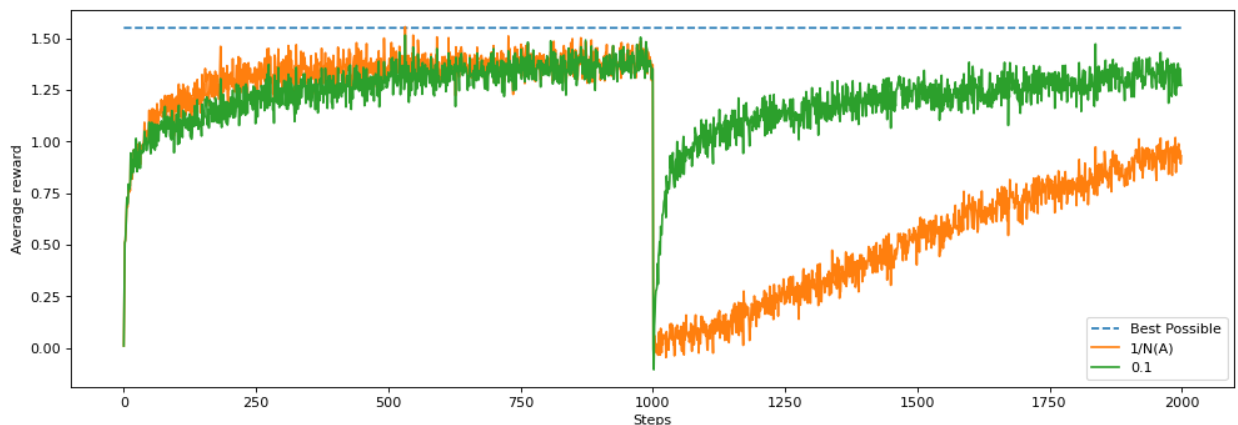
```
100%|████████████| 500/500 [00:19<00:00, 26.02it/s]
100%|████████████| 500/500 [00:18<00:00, 27.09it/s]
```


Now the agent with a step size of 1/N(A) performed better at the start but then performed worse when the environment changed! What happened?

Think about what the step size would be after 1000 steps. Let's say the best action gets chosen 500 times. That means the step size for that action is 1/500 or 0.002. At each step when we update the value of the action and the value is going to move only 0.002 * the error. That is a very tiny adjustment and it will take a long time for it to get to the true value.

The agent with step size 0.1, however, will always update in 1/10th of the direction of the error. This means that on average it will take ten steps for it to update its value to the sample mean.

These are the types of tradeoffs we have to think about in reinforcement learning. A larger step size moves us more quickly toward the true value, but can make our estimated values oscillate around the expected value. A step size that reduces over time can converge to close to the expected value, without oscillating. On the other hand, such a decaying stepsize is not able to adapt to changes in the environment. Nonstationarity---and the

related concept of partial observability---is a common feature of reinforcement learning problems and when learning online.

Section 5: Conclusion

Great work! You have:

- Implemented your first agent
- Learned about the effect of epsilon, an exploration parameter, on the performance of an agent
- Learned about the effect of step size on the performance of the agent
- Learned about a good experiment practice of averaging across multiple runs