Homework - Reinforcement Learning - Part A (40/100 points)

by *Todd Gureckis* and *Brenden Lake*
Computational Cognitive Modeling
NYU class webpage: https://brendenlake.github.io/CCM-site/ email to course instructors: instructors-ccm-spring2024@googlegroups.com

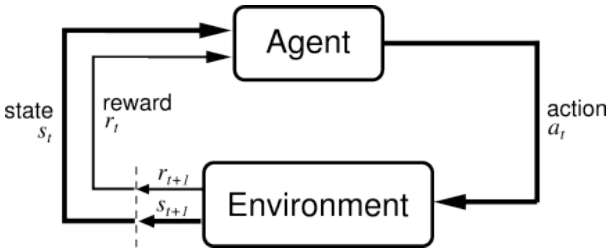
This homework is due before midnight on March 7, 2024.

Reinforcement Learning

Reinforcement learning (RL) is a topic in machine learning and psychology/neuroscience which considers how an emboddied agent should learn to make decisions in an environment in order to maximize reward. You could definitely do worse things in life than to read the classic text on RL by Sutton and Barto:

- Sutton, R.S. and Barto, A.G. (2017) Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA. [available online for free!]

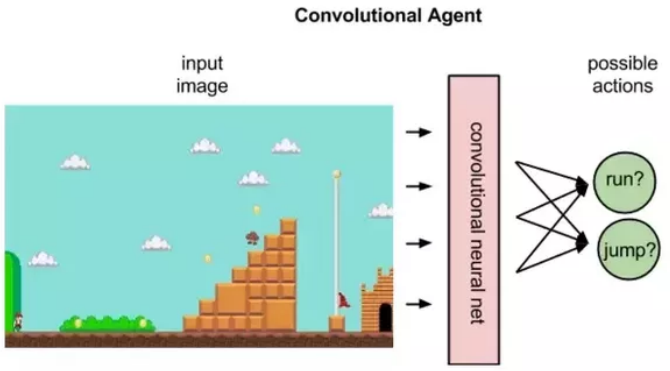
The standard definition of the RL problem can be summarized with this figure:



The agent at each time point chooses an action which influences the state of the world according to the rules of the environment (e.g., spatial layout of a building or the very nature of physics). This results in a new state (s_{t+1}) and possibly a reward (r_{t+1}). The agent then receives the new state and the reward signal and updates in order to choose the next action. The goal of the agent is to maximize the reward received over the long run. In effect this approach treats life like an optimal control problem (one where the goal is to determine the best actions to take for each possible state).

The simplicity and power of this framework has made it very influential in recent years in psychology and computer science. Recently more advanced techniques for solving RL problems have been scaled to show impressive performance on complex, real-world tasks. For example, so called "deep RL" system which combine elements

of deep convolutional nets and reinforcement learning algorithms can learn to play classic video games at near-human performance, simply by aiming to earn points in the game:



- Mnih, V. et al. (2015) Human-level control through deep reinforcement learning. *Nature*, 518, 529. [pdf]

In this homework we will explore some of the underlying principles which support these advances.

The homework is divided into two parts:

1. The first part (this notebook) explores different solution methods to the problem of behaving optimally in a *known* environment.
2. The second part explores some basic issues in learning to choose effectively in an *unknown* enviroment.

References:

- Sutton, R.S. and Barto, A.G. (2017) Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA.
- Gureckis, T.M. and Love, B.C. (2015) Reinforcement learning: A computational perspective. Oxford Handbook of Computational and Mathematical Psychology, Edited by Busemeyer, J.R., Townsend, J., Zheng, W., and Eidels, A., Oxford University Press, New York, NY.
- Daw, N.S. (2013) "Advanced Reinforcement Learning" Chapter in Neuroeconomics: Decision making and the brain, 2nd edition
- Niv, Y. and Schoenbaum, G. (2008) "Dialogues on prediction errors" *Trends in Cognitive Science*, 12(7), 265-72.
- Nathaniel D. Daw, John P. O'Doherty, Peter Dayan, Ben Seymour & Raymond J. Dolan (2006). Cortical substrates for exploratory decisions in humans. *Nature*, 441, 876-879.

Learning and deciding in a known world

Reinforcement learning is a collection of methods and techniques for learning to make good or optimal sequential decisions. As described in the lecture, the basic

definition of the RL problem (see above) is quite general and therefore there is more than one way to solve an RL problem (and even multiple ways to define what the RL problem is).

In this homework we are going to take one simple RL problem: navigation in a grid-world maze, and explore two different ways of solving this decision problem.

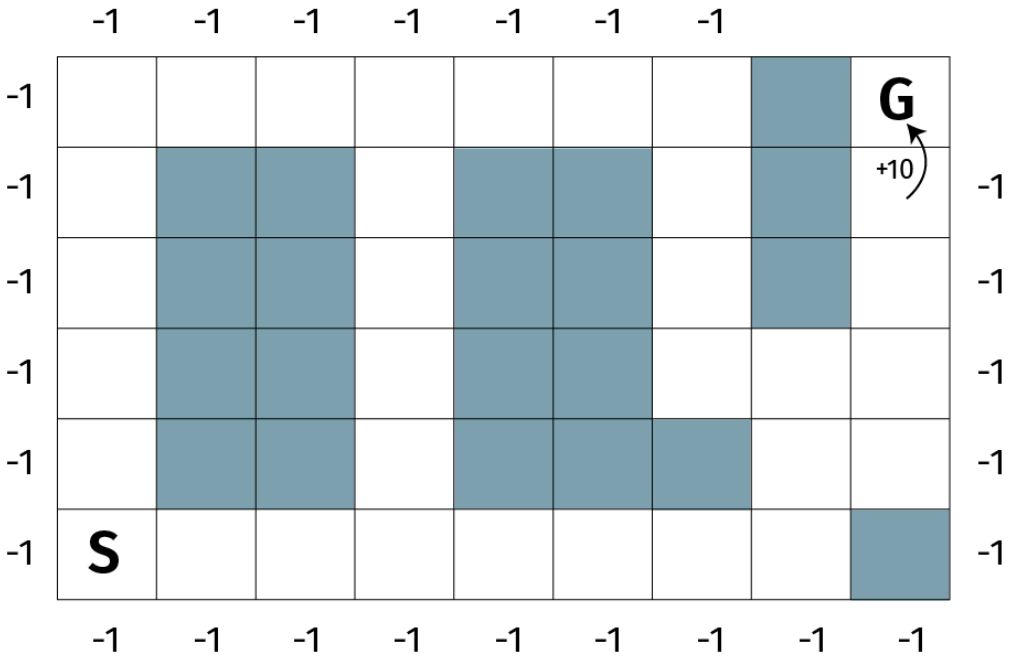
- The first method is going to be policy-iteration or dynamic programming.
- The second method is going to be monte-carlo simulation.

By seeing the same problem solved multiple ways, it helps to reinforce the differences between these different approaches and the features of the algorithms that are interesting from the perspective of human decision making.

The problem defintion

The problem we will consider is a grid world task. The grid is a collection of rooms. Within each room there are four possible actions (move up, down, left, right). There are also walls in the maze that the agent cannot move through (indicated in blue-grey below). There are two special states, S which is the start state, and G which is the goal state. The agent will start at location S and aims to arrive at G. When the agents moves into the G state they earn a reward of +10. If they walk off the edge of the maze, they receive a -1 reward and are returned to the S state. G is an absorbing state in the sense that you can think of the agent as never leaving that state once they arrive there.

The specific gridworld we will consider looks like this:



The goal of the agent to determine the optimal sequential decision making policy to arrive at state G.

To help you with this task we provide a simple `GridWorld()` class that makes it easy to specify parts of the gridworld environment and provides access to some of the variables you will need in constructing your solutions to the homework. In order to run the gridworld task you need to first execute the following cell:

> **Warning!** Before running other cells in this notebook you must first successfully execute the following cell which includes some libraries.

In [ ]:
```
# import the gridworld library
import math
import random
import statistics
import numpy as np
from copy import deepcopy
from gridworld import GridWorld, random_policy
from IPython.display import display, Markdown, Latex, HTML
```

The following cell sets up the grid world defined above including the spatial layout and then a python dictionary called `rewards` that determines which transitions between states result in a reward of a given magnitude.

In [ ]:
```
gridworld = [
    ['o', 'o', 'o', 'o', 'o', 'o', 'o', 'x', 'g'],
    ['o', 'x', 'x', 'o', 'x', 'x', 'o', 'x', 'o'],
    ['o', 'x', 'x', 'o', 'x', 'x', 'o', 'x', 'o'],
    ['o', 'x', 'x', 'o', 'x', 'x', 'o', 'o', 'o'],
    ['o', 'x', 'x', 'o', 'x', 'x', 'x', 'o', 'o'],
    ['s', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 'x'],
]  # the problem described above, 'x' is a wall, 's' is start, 'g' is goal, and

mygrid = GridWorld(gridworld)
mygrid.raw_print()    # print out the grid world
mygrid.index_print()  # print out the indicies of each state
mygrid.coord_print()  # print out the coordinates of each state (helpful in your

# define the rewards as a hash table
rewards = {}

# mygrid.transitions contains all the pairwise state-state transitions allowed i
# for each state transition intialize the reward to zero
for start_state in mygrid.transitions:
    for action in mygrid.transitions[start_state].keys():
        next_state = mygrid.transitions[start_state][action]
        rewards[str([start_state, action, next_state])] = 0.0

# now set the reward for moving up into state 8 (the goal state) to +10
rewards[str([17, "up", 8])] = 10

# now set the penalty for walking off the edge of the grid and returning to stat
for i in [0, 1, 2, 3, 4, 5, 6, 7]:
    rewards[str([i, "up", 45])] = -1
for i in [0, 9, 18, 27, 36, 45]:
    rewards[str([i, "left", 45])] = -1
for i in [45, 46, 47, 48, 49, 50, 51, 52, 53]:
    rewards[str([i, "down", 45])] = -1
```

```
for i in [8, 17, 26, 35, 44, 53]:
    rewards[str([i, "right", 45])] = -1
```

Welcome to your new Grid World!

Raw World Layout

o	o	o	o	o	o	o	x	g
o	x	x	o	x	x	o	x	o
o	x	x	o	x	x	o	x	o
o	x	x	o	x	x	o	o	o
o	x	x	o	x	x	x	o	o
s	o	o	o	o	o	o	o	x

Indexes of each grid location as an id number

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53

Indexes of each grid location as a tuple

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	(0,8)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)

Notice that the above printouts show the grid but also an array of the indexes and coordinated of each location on the grid. You will need these to help you analyze your solution to the homework so it can be frequently helpful to refer back to these outputs.

In order to solve this problem using dynamic programming the agent needs to maintain two key representations. One is the value of each state under the current policy, V^{π}, and the other is the policy $\pi(s, a)$. The following cell initializes a new value table and a new random policy and uses functions provided in `GridWorld` to print these out in the notebook in a friendly way.

In [ ]:
```
value_table = np.zeros((mygrid.nrows, mygrid.ncols))
display(Markdown("**Current value table for each state**"))
mygrid.pretty_print_table(value_table)

policy_table = [
    [random_policy() for i in range(mygrid.ncols)] for j in range(mygrid.nrows)
]
display(Markdown("**Current (randomized) policy**"))
mygrid.pretty_print_policy_table(policy_table)
```

Current value table for each state

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Current (randomized) policy

→	↓	←	←	→	→	↓	▉	↓
↓	▉	▉	↑	▉	▉	←	▉	←
↑	▉	▉	→	▉	▉	↓	▉	→
↓	▉	▉	→	▉	▉	→	→	↓
↓	▉	▉	→	▉	▉	▉	←	←
→	↑	→	→	↑	↓	←	↑	▉

Note how the current policy is random with the different arrows within each state pointing in different, sometimes opposing directions. Your goal is to solve for the best way to orient those arrows.

Dynamic Programming via Policy Iteration

Problem 1 (15 points)

Remember that the Bellman equation that recursively relates the value of any state to any other state is like this:

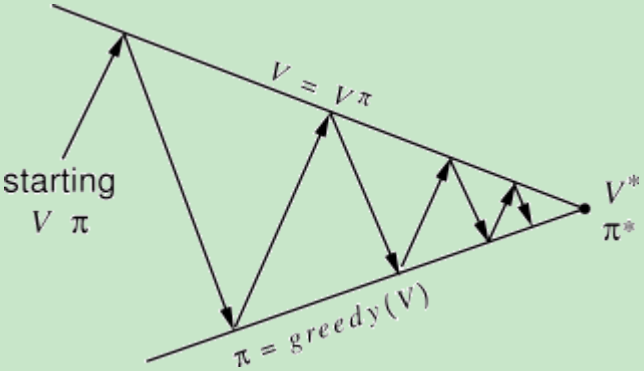
$$V^{\pi}(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')]$$

Your job in this first exercise is to set up a dynamic programming solution to the provided gridworld problem. Your should implement two steps. The first is policy evaluation which means given a policy (in `policy_table`) update the `value_table` to be consistent with that policy. Your algorithm should do this by

visiting each state in a random order and updating it's value in the `value_table` (this is known as asychronous update since you are changing the values in-place).

The next step is policy improvement where you change the policy to maximize expected long-run reward in each state by adjusting which actions you should take (this means changing the values in `policy_table`). We will only consider deterministic policies in this case. Thus your algorithm should always choose one action to take in each state even if two actions are similarly valued.

The algorithm you write should iterate sequentially between policy evaluate and (greedy) policy improvement for at least 2000 iterations.



(Figure from Sutton and Barto text)

To gain some intuition about how preferences for the future impact the resulting policies, run your algorithm twice, once with γ set to zero (as in lecture) and another with γ set to 0.9 and output the resulting policy and value table using `mygrid.pretty_print_policy_table()` and `mygrid.pretty_print_table()`.

Info
The `GridWorld` class provides some helpful functions that you will need in your solution. The following code describes these features

Only some states are "valid" i.e., are not walls. `mygrid.valid_states` is a python dictionary containing those states. The keys of this dictionary are id numbers for each state (see the output of `mygrid.index_print()`) and the values are coordinates (see the output of `mygrid.coord_print()`). Your algorithm will want to iterate over this list to update the value of each "valid" state.

In [ ]:
```
mygrid.valid_states  # output the indexes and coordinates of the valid states
```

Out[ ]:
```
{0: (0, 0),
 1: (0, 1),
 2: (0, 2),
 3: (0, 3),
 4: (0, 4),
 5: (0, 5),
 6: (0, 6),
 8: (0, 8),
 9: (1, 0),
 12: (1, 3),
 15: (1, 6),
 17: (1, 8),
 18: (2, 0),
 21: (2, 3),
 24: (2, 6),
 26: (2, 8),
 27: (3, 0),
 30: (3, 3),
 33: (3, 6),
 34: (3, 7),
 35: (3, 8),
 36: (4, 0),
 39: (4, 3),
 43: (4, 7),
 44: (4, 8),
 45: (5, 0),
 46: (5, 1),
 47: (5, 2),
 48: (5, 3),
 49: (5, 4),
 50: (5, 5),
 51: (5, 6),
 52: (5, 7)}
```

As the previous command makes clear, there are two ways of referencing a state: by its id number or by its coordinates. Two functions let you swap between those:

- `mygrid.index_to_coord(index)` converts a index (e.g., 1-100) to a coordinate (i,j)
- `mygrid.coord_to_index(coord)` takes a tuples representing the coordinate (i,j) and return the index (e.g., 1-100)

Both the value table (`value_table`) and policy table (`policy_table`) are indexed using coordinates.

A key variable for your algorithm is $\mathcal{P}^a_{ss'}$ which is the probability of reaching state s' when in state s and taking action a. We assume that the world is deterministic here so these probabilities are always 1.0. However, some states do not lead to immediately adjacent cells but instead return to the start state (e.g., walking off the edge of the grid). `mygrid.transitions` contains a nested hash table that contains this information for your gridworld. For example consider state 2:

In [ ]:
```
state = 2
mygrid.transitions[state]
```

Out[ ]:
```
{'up': 45, 'right': 3, 'down': 2, 'left': 1}
```

The output of the above command is a python dictionary showing what next state you will arrive at if you chose the given actions. Thus `mygrid.transitions[2]['down']` would return state id 2 because you will hit the wall and thus not change state. Whereas `mygrid.transitions[2]['left']` will move to state 1. The `mygrid.transitions` dictionary thus provides all the information necessary to represent $P^a_{ss'}$. The world is deterministic so taking an action in a given state will always move the agent to the next corresponding state with probability 1.

The next variable you will need is the reward function. Rewards are delivered anytime the agent makes a transition from one state to another using a particular action. Thus this variable is written $\mathcal{R}^a_{ss'}$ in the equation above. You can access this programmatically using the python dictionary `rewards` which we ourselves defined above. The `rewards` dictionary defines the reward for taking a particular action in a particular state and then arriving at a new state s'. To look up the reward for a particular $< s, a, s' >$ triplet you create a list with these variables in index format, convert it to a string, and look it up in the dictionary. For example the reward for being in state 17, choosing up, and then arriving in state 8 is:

In [ ]:
```
state = 17
next_state = 8
action = "up"
rewards[str([state, action, next_state])]
```

Out[ ]:
```
10
```

This should be the required ingredients to solve both the policy evaluation and policy improvement functions that you will need to write. If you need further information you can read the `GridWorld` class directly in gridworld.py.

Your solution:

Implement the two major steps of your algorithm as the following two functions. Then write code that iterates between them for the specified number of steps and inspect the final solution. **Some scaffolding code has been provided for you so all you have to implement is the sections noted in the comments**

In [ ]:
```
def policy_evaluate(mygrid: GridWorld, value_table, policy_table, GAMMA):
    valid_states = list(mygrid.valid_states.keys())
    random.shuffle(valid_states)
```

```
    for state in valid_states:
        sx, sy = mygrid.index_to_coord(state)
        new_value = 0.0
        for action in mygrid.transitions[state].keys():
            next_state = mygrid.transitions[state][action]
            next_sx, next_sy = mygrid.index_to_coord(next_state)
            new_value += policy_table[sx][sy][action] * (
                rewards[str([ state, action, next_state ])] +
                GAMMA * value_table[next_sx][next_sy]
            )
        value_table[sx][sy] = new_value


# this is a helper function that will take a set of q-values and convert them in
def be_greedy(q_values):
    if len(q_values) == 0:
        return {}

    keys = list(q_values.keys())
    vals = [q_values[i] for i in keys]
    maxqs = [i for i, x in enumerate(vals) if x == max(vals)]
    if len(maxqs) > 1:
        pos = random.choice(maxqs)
    else:
        pos = maxqs[0]
    policy = deepcopy(q_values)
    for i in policy.keys():
        policy[i] = 0.0
    policy[keys[pos]] = 1.0
    return policy


def policy_improve(mygrid, value_table, policy_table, GAMMA):
    # for each state
    valid_states = list(mygrid.valid_states.keys())

    for state in valid_states:
        # compute the Q-values for each action
        q_values = {}
        for action in mygrid.transitions[state].keys():
            next_state = mygrid.transitions[state][action]
            next_sx, next_sy = mygrid.index_to_coord(next_state)
            qval = rewards[str([ state, action, next_state ])] + \
                   GAMMA * value_table[next_sx][next_sy]
            q_values[action] = qval
        newpol = be_greedy(
            q_values
        )  # take this dictionary and convert into a greedy policy
        # then update the policy table printing to allow more complex policies
        sx, sy = mygrid.index_to_coord(state)
        for action in mygrid.transitions[state].keys():
            policy_table[sx][sy][action] = newpol[action]
```

The following code actually runs the policy iteration algorithm cycles. You should play with the parameters of this simulation until you are sure that your algorithm has converged and that you understand how the various parameters influence the obtained solutions.

In [ ]:
```
mygrid.pretty_print_table(value_table)
mygrid.pretty_print_policy_table(policy_table)

GAMMA = 0.9  # run your algorithm from
# above with different settings of GAMMA
# (Specifically 0 and 0.9 to see how the resulting value function and policy cha
for i in range(2000):
    policy_evaluate(mygrid, value_table, policy_table, GAMMA)
    policy_improve(mygrid, value_table, policy_table, GAMMA)

mygrid.pretty_print_table(value_table)
mygrid.pretty_print_policy_table(policy_table)
```

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

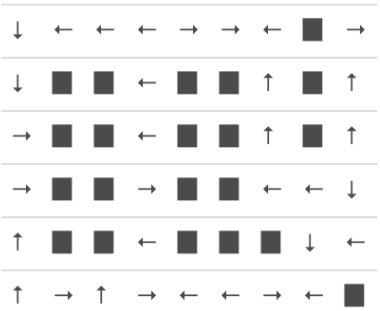
→	↓	←	←	→	→	↓	▉	↓
↓	▉	▉	↑	▉	▉	←	▉	←
↑	▉	▉	→	▉	▉	↓	▉	→
↓	▉	▉	→	▉	▉	→	→	↓
↓	▉	▉	→	▉	▉	▉	←	←
→	↑	→	→	↑	↓	←	↑	▉

2.54187	2.8243	3.13811	3.48678	3.8742	4.30467	4.78297	0	0
2.28768	0	0	3.13811	0	0	5.31441	0	10
2.05891	0	0	2.8243	0	0	5.9049	0	9
2.28768	0	0	3.13811	0	0	6.561	7.29	8.1
2.54187	0	0	3.48678	0	0	0	6.561	7.29
2.8243	3.13811	3.48678	3.8742	4.30467	4.78297	5.31441	5.9049	0

→	→	→	→	→	→	↓	▉	↓
↑	▉	▉	↑	▉	▉	↓	▉	↑
↓	▉	▉	↓	▉	▉	↓	▉	↑
↓	▉	▉	↓	▉	▉	→	→	↑
↓	▉	▉	↓	▉	▉	▉	↑	↑
→	→	→	→	→	→	→	↑	▉

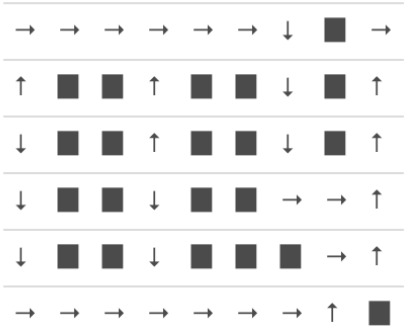
Your final policy should look something like this for $\gamma = 0.0$:

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	10
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0



and like this for $\gamma = 0.9$

2.54187	2.8243	3.13811	3.48678	3.8742	4.30467	4.78297	0	0
2.28768	0	0	3.13811	0	0	5.31441	0	10
2.05891	0	0	2.8243	0	0	5.9049	0	9
2.28768	0	0	3.13811	0	0	6.561	7.29	8.1
2.54187	0	0	3.48678	0	0	0	6.561	7.29
2.8243	3.13811	3.48678	3.8742	4.30467	4.78297	5.31441	5.9049	0



Although note that your solution may not be identical because we are doing greedy action selection and randomly choosing one perferred action in the case

that there are ties (partly because it is harder to display stochastic policies as a grid). However, if you consider the structure of this particular gridworld there is always one best move.

First Visit Monte-Carlo

In the previous exercise you solved the sequential decision making problem using policy iteration. However, you relied heavily on the information provided by the `GridWorld()` class, especially $\mathcal{P}^a_{ss'}$ (`mygrid.transitions`) and $\mathcal{R}^a_{ss'}$ (`rewards`). These values are not commonly known when an agent faces an environment. In this step of the homework you will solve the same grid world problem this time using Monte-Carlo.

Monte Carlo methods are ones where stochastic samples are drawn from a problem space and aggregated to estimate quantities of interest. In this case we want to average the expected rewards available from a state going forward. Thus, we will use Monte Carlo methods to estimate the value of particular actions or states.

The specific Monte-Carlo algorithm you should use is known as First-Visit Monte Carlo (described in lecture). According to this algorithm, each time you first visit a state (or state-action pair) you record the rewards received until the end of an episode. You do this many times and then average together the rewards received to estimate the value of the state or action.

Then, as you did in problem 1, you adjust your policy to become greedy with respect to the values you have estimated.

There are two significant conceptual changes in applying Monte-Carlo to the gridworld problem. First is that rather than estimate the value of each state $V^\pi(s)$ under the current policy π, it makes more sense to estimate the value of each state-action pair, $Q^\pi(s,a)$, directly. The reason is that in your previous solution, in order to determine the optimal policy, you likely had to know $\mathcal{P}^a_{ss'}$ to determine which action to perform and which state it would lead to. Since we are trying to avoid accessing any explicit knowledge about the probabilities and rewards we cannot use this variable in our solution. Thus, average the returns following the first visit to a particular action.

The second is what policy we should use for running our Monte Carlo updates. If we randomly initialize the policy as we did above and then run it forward it is very easy for the runs to get caught in cycles and loops that never visit many of the states or ever encounters any rewards. Thus, we will want to include some randomness in our simulations so that they have a non-zero probability of choosing different actions. We will consider this issue in more detail in Part B of the homework. For now use the ϵ-greedy algorithm which chooses the currently "best" action with probability $1-\epsilon$ and otherwise chooses randomly.

In addition, we will utilize the concept of **exploring starts**. Even though we designated one state as the "Start state" it can help the monte carlo algorithm explore more efficiently if we start the episodes from random starting locations. The reason is that early on the policy might have loops and other inconistencies which mean some states are rarely sampled, if at all. Exploring starts (when feasible) can help the algorithm avoid these local minima.

Problem 2 (15 points)

In this exercise you should solve the problem introduced at the start of this notebook using Monte Carlo methods. The pseudo code for your algorithm is described here:

```
Initialize, for all $s \in S$, $a \in A(s)$:
    $Q(s,a)$ <- arbitrary
    $\pi(s)$ <- arbitrary
    $Returns(s,a)$ <- empty list

Repeat many times:
    a) Generate an episode using $\pi$ with $\epislon$ probability
of choosing an action at random
    b) For each pair $s,a$ appearing in the episode
        R <- return following the first occurence of $s,a$
        Append R to $Returns(s,a)$
        $Q(s,a)$ <- discounted_average(Returns(s,a))
    c) For eash $s$ in episode:
        $\pi(s)$ <- arg max_a Q(s,a)
```

When you compute the average returns you should weight them by them by γ so that they reflect the discount rates described above. Run your algorithm for both $\gamma = 0.0$ and $\gamma = 0.9$ and compare the resulting `policy_table` to the one you obtained in Problem 1. They should work out to the same optimal policies, obtained using a quite different method, and one that in particular doesn't need an explicit model of the environment.

Keep in mind that in cases where there are two equally good actions which one is selected and shown in your policy table is arbitrary. If correctly implemeneted the dynamic programming soluion then you should be aware of when these cases happen. It is thus fine if the policies you get from monte-carlo and dynamic programming are not **identical** but are still **correct**.

There are a couple of hints that you will need to implement your solution which are provided by the `GridWorld` class. The first is that you will still need to use the `rewards` dictionary from your solution to Problem 1 to compute when the rewards are delievered. However instead of consulting this function arbitrarily you are using it just to sample the rewards when the correct event happens in your Monte Carlo simulation.

Second, you will need to find out what state you are in after taking an action in a given state. The one-step transition dynamics of the gridworld can be simulated from the GridWorld class. For example, to determine the state you would be in if you were in state 45 (the start state) and chose the action "up", "down", "left", or "right" is given by:

In [ ]:
```
[mygrid.up(45), mygrid.down(45), mygrid.left(45), mygrid.right(45)]
```

Out[ ]:
```
[36, 45, 45, 46]
```

Note that in this example, down and left walk off the edge of the environment and thus return the agent to the start state.

The following two functions implement the epsilon-greedy Monte Carlo sample from your gridworld task using a recursive function. Although this is provided to you for free, you should try to understand the logic of these functions.

In [ ]:
```
def epsilon_greedy(actions, epsilon):
    if random.random() < epsilon:
        return random.choice(list(actions.keys()))
    else:
        if actions["up"] == 1.0:
            return "up"
        elif actions["right"] == 1.0:
            return "right"
        elif actions["down"] == 1.0:
            return "down"
        elif actions["left"] == 1.0:
            return "left"


# recursively sample state-action transitions using epsilon greedy algorithm wit
def mc_episode(
    current_state, epsilon, goal_state, policy_table, depth=0, max_depth=100
):
    if current_state != goal_state and depth < max_depth:
        sx, sy = mygrid.index_to_coord(current_state)
        action = epsilon_greedy(policy_table[sx][sy], epsilon)
        if action == "up":
            new_state = mygrid.up(current_state)
        elif action == "right":
            new_state = mygrid.right(current_state)
        elif action == "down":
            new_state = mygrid.down(current_state)
        elif action == "left":
            new_state = mygrid.left(current_state)
        r = rewards[str([current_state, action, new_state])]
        return [[r, current_state, action]] + mc_episode(
            new_state,
            epsilon,
            goal_state,
            policy_table,
            depth=depth + 1,
```

```
            max_depth=max_depth,
        )
    else:
        return []
```

Some initial data structures for managing the q-values, policy, and returns have been defined for you here:

In [ ]:
```
starting_state = 45
goal_state = 8  # terminate the MC roll out when you get to this state
GAMMA = 0.9
EPSILON = 0.2  # more exploration is often better
ITERATIONS = 200000  # this may need to be 100,000 or more!
PRINT_EVERY = 40000  # how often to print out our progress
random.seed(5000)  # try multiple random seed to verify your code works


# set up initial data strucutres that might be useful for you
# q(s,a)
def init_q_values(init):
    qvals = {"up": init, "right": init, "down": init, "left": init}
    return qvals


INIT_VAL = (
    -99999.0
)  # initialize unsampled q values to a very small number (pessimitic intiaizlia
q_value_table = [
    [init_q_values(INIT_VAL) for i in range(mygrid.ncols)] for j in range(mygrid
]

# pi
policy_table = [
    [random_policy() for i in range(mygrid.ncols)] for j in range(mygrid.nrows)
]
display(Markdown("**Initial (randomized) policy**"))
mygrid.pretty_print_policy_table(policy_table)

# dictionary for returns, can be filled in as more info is encountered
returns = {}

for i in range(
    ITERATIONS
):  # you probably need to take many, many steps here and it make take some time
    # instead of always starting at the start state, this algorithm will use the
    # "exploring start" so that it starts in a random valid state
    # this can help a lot
    ss = random.choice(
        list(mygrid.valid_states.keys())
    )  # select and exploring start state
    episode = mc_episode(ss, EPSILON, goal_state, policy_table)

    visited = {}
    for idx in range(len(episode)):
        item = episode[idx]
        qkey = str((item[1], item[2]))
        if qkey not in visited:
            visited[qkey] = []
```

```
            for qk in visited.keys():
                visited[qk].append(item[0])
        for qk in visited.keys():
            if qk not in returns:
                returns[qk] = [ 0, 0 ]
            returns[qk][0] += sum([ v*(GAMMA**vi) for vi, v in enumerate(visited[qk]
            returns[qk][1] += 1

        # update q-value-table
        for ret in returns.keys():
            s, a = eval(ret)
            sx, sy = mygrid.index_to_coord(s)
            q_value_table[sx][sy][a] = returns[ret][0] / returns[ret][1]

        # improve policy
        for sx in range(len(q_value_table)):
            for sy in range(len(q_value_table[sx])):
                policy_table[sx][sy] = be_greedy(q_value_table[sx][sy])

        if i % PRINT_EVERY == 0:
            display(Markdown(f"**Improved policy interation {i}**"))
            mygrid.pretty_print_policy_table(policy_table)

display(Markdown("**Improved policy**"))
mygrid.pretty_print_policy_table(policy_table)
```

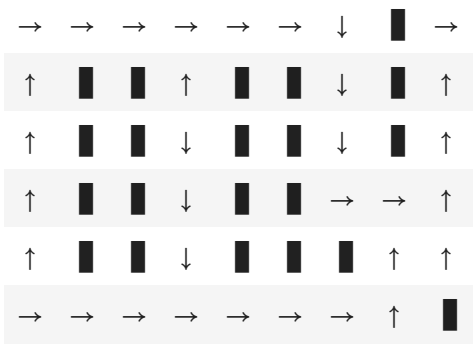
Initial (randomized) policy

→	↓	←	←	↑	←	↑		←
↓			←			→		↓
←			↑			→		→
↑			↑			↓	←	←
←			→				↑	↓
↑	↓	↓	↑	←	←	↓	↓	

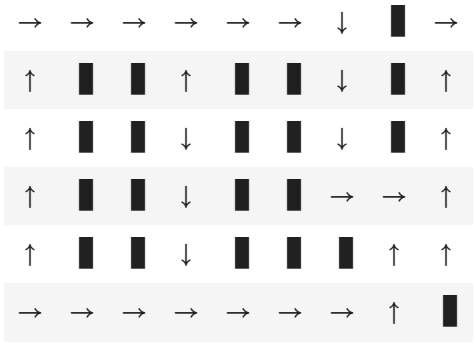
Improved policy interation 0

→	→	→	↑	↓	↓	↓		←
↑			←			↑		←
←			←			→		→
↑			↑			↑	↑	←
↓			↑				↓	↓
←	↓	↓	↓	↑	↑	↓	↓	

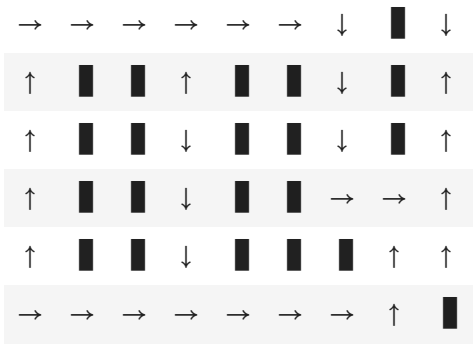
Improved policy interation 40000



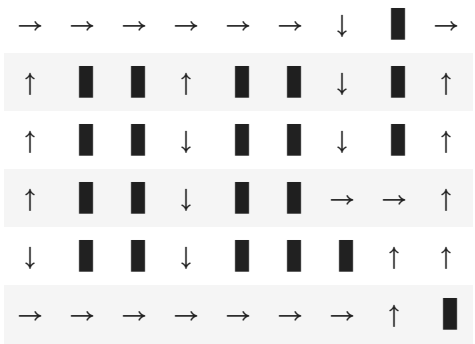
Improved policy interation 80000



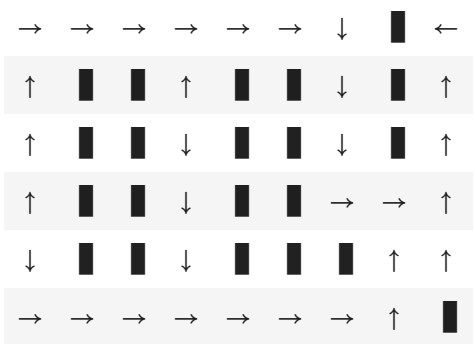
Improved policy interation 120000



Improved policy interation 160000



Improved policy



Problem 3 (10 points)

The two solution methods we just considered have different strengths and weaknesses. Describe in your own words the things that make these solutions methods better or worse. Your response should be 2-3 sentences and address both the computational efficiency of the algoriths, the amount of assumed knowledge of the environment, and the relationship between these methods to how humans might solve similar sequential decision making problems. Are either algorithm plausible models of human cognition?

The parameter GAMMA (γ) is important for both method to gain the ability to look at the future, without it (setting it to 0) will make both method almost loss their functions.

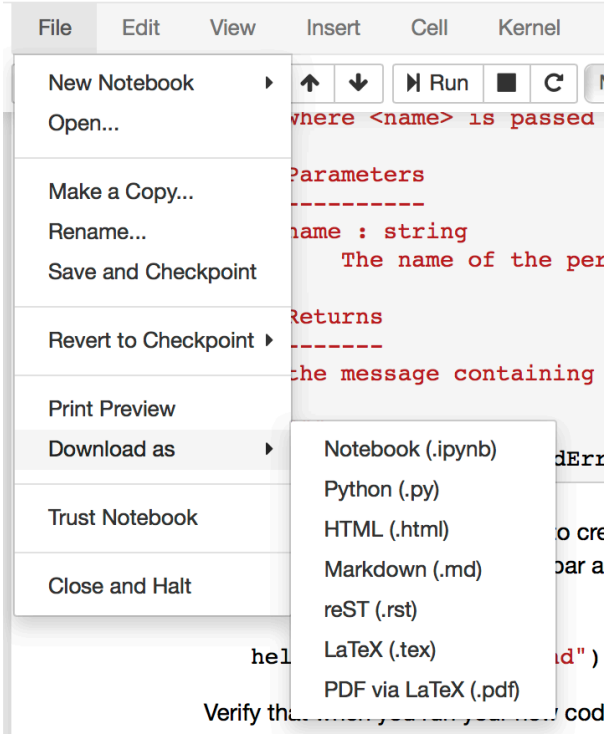
For the first method, the weakness is that we have to know the environment more at the beginning, but it doesn't need to expand the route to the goal.
However, when there are many action options but less possible state, the first method might be less efficient.
The computational time is $O(isa)$, where i is the number of iterations, s is the number of states, and a is the the number of actions.

For the second method (MC), since it is more like DFS, the weakness is that some states might never or rarely be explore.
More, the upstream of possible routes are always less explored that the downstream.
So the upstream parts are trained slower than the downsteam.
The computational time is $O(is)$, where i is the number of iterations, and s is the number of states.
While it seems like more efficient only looking on the denotion, the i here is larger than the i in the first method.

The first method could be used in the case like humans playing chess.
While we do try to predict few moves later, we can never think to the end of the game to decide our move.
The second method is suitable for the case like finding the fastest way to a location.
After few times of attempt, we are able to summarize the key factor to make our trip most efficient.

Turning in homework

When you are finished with this notebook. Save your work in order to turn it in. To do this select *File->Download As...->HTML*.



You can turn in your assignments using Gradescope. **Make sure you complete all parts (A and B) of this homework.**

Next steps...

So, far, so good... Now move on to Homework Part B

Homework - Reinforcement Learning - Part B (60/100 points)

by *Todd Gureckis* and *Brenden Lake*
Computational Cognitive Modeling
NYU class webpage: https://brendenlake.github.io/CCM-site/
email to course instructors: instructors-ccm-spring2024@nyuccl.org

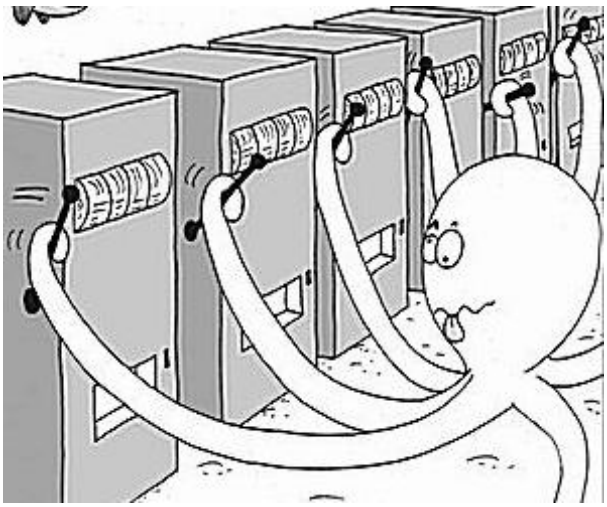
This homework is due before midnight on March 7, 2024.

Learning and deciding in an unknown world

Part A of the homework explored various solution to a sequential decision making problem in a case where considerable information about the environment was known or was provided (e.g., the probabilities of transitioning between different states and the magnitude of rewards available in particular states). However, if reinforcement learning could only be applied to cases with full, explicit knowledge than it would be much less of a compelling model of human learning. In many cases, humans and other animals learn even when there is much more ambiguity, and as a result, a good model of experiential learning for humans would apply to cases where less is known a-priori about the world.

In this part of the homework, we will shift to think about learning and deciding in a unknown environment. This is a fairly complex topic with many different solutions and types of problems. However, we will focus on one particular problem class known as the **n-armed bandit**. N-armed bandits are optimization problems that mimic many real-world problems faced by humans, organizations, and machine learning agents. The term "bandit" comes from the name of the casino games where

you pull a lever to enter a lottery. The bandits have one arm (the arm you pull down) and they steal your money (see above).



An N-armed bandit is a problem where a decision maker is presented with a bandit with n arms instead of just one (see Octopus cartoon). The task for the agent is, on each trial or moment in time, to choose bandits that are good while avoiding those that are less good. Since nothing may be known about the bandits a-priori, the problem is difficult and requires a balance of *exploration* (trying new things in order to learn) and *exploitation* (choosing options known to be good).

If each bandit paid out a fixed amount every time it was selected, then the problem would be solved with very simple exhaustive search process (visit each bandit once and then select the best one for the remaining time). However, the sequential search strategy just described doesn't capture the *opportunity cost* of exploration. For example, imagine that there is 100 armed bandits. Further assume that you know that 98 give zero reward, one gives a reward of 10, and one gives a reward of 20. If on the first pull you receive 10 units of reward then you are lucky and landed on a good one. However, is it worth going searching for the 20 point bandit? Given that you will have to pull a lot of zero reward bandits, it might actually be more rewarding over a finite period to continue to pull the 10 point bandit arm. Thus, exploration and exploitation act more like a tradeoff depending on the structure of the problem.

In addition, when the reward received from each bandit is probabilistic or stochastic, and furthermore the quality of the bandits might change over time, the problem becomes much more difficult. These cases require the agent to learn from the past but also be willing to adjust their beliefs based on more recent information.

Bandit tasks come up in many areas of cognitive science and machine learning. For example, there is a way to view A/B testing on websites as a particular type of bandit problem (your goal is to ensure conversions or purchases on your website, and your bandit arms are the different web designs you might try out). Similarly, the very real human problem of deciding where to eat lunch is a bit like a bandit problem -- should you return to your favorite restuarant or try a new one? Is the exploration worth giving up a reliably good meal?

In this part of the homework you will explore different simple algorithms for bandit problems.

Starter code

Warning! Before running other cells in this notebook you must first successfully execute the following cell which includes some libraries.

In [ ]:
```
# The typical imports
import math
import random
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

A simple bandit environment

The first class provided here creates a set of simple, stationary multi-arm bandits. The bandits are stateless in that the reward from choose each action is simply a probabilistic function of the bandit itself, and there are no other cues you can use to decide which action to take. The parameters to the constructor of the bandit environment are:

- `mus` : the mean of the distribution from which each bandit is drawn from. This should have k numbers (for the k-armed bandit)
- `sds` : the standard deviation of the distribution from which the bandit means are drawn from (also k numbers for k bandits)

In [ ]:
```
class KArmBanditStationary:
    def __init__(self, mus, sds):
        self.action_means = mus
        self.action_sds = sds
        self.optimal = np.argmax(self.action_means)
        self.k = len(mus)

    def step(self, action):
        return (
            np.random.normal(self.action_means[action], self.action_sds[action]),
            action == self.optimal,
        )
```

Your job in this first exercise is to write a simple RL agent which samples from these bandits and attempts to earn as much reward as possible. The following cell gives an example of how to initialize the bandit and how to draw from it

In [ ]:
```
bandit = KArmBanditStationary([0, 50, 10, 4], [10, 10, 10, 10])
action = 0  # pull the 0th bandit
bandit.step(
```

```
    action
)  # return values are the reward, and if the action is actually optimal or not
```

Out[ ]: (5.7525900438823605, False)

When we initialize the `KArmBanditStationary` we in some sense know exactly which arm is optimal (the one with the higher mean), and also how hard the problem is (the standard deviation of the rewards on each arm determines the difficulty... low SD generally is a easier task due to signal-noise relationship).

However, we are going to be implementing agents that do not have access to this information. For example, this first cell implements a simple random agent. You will want to modify this class to create an agent that can learn.

In [ ]:
```
class RandomAgent:
    def __init__(self, k):
        self.num_actions = k
        # you could add parameters to your agent here
        pass

    def choose(self):
        return np.random.randint(self.num_actions)

    def learn(self, reward, action):  # this agent doesn't learn
        pass
```

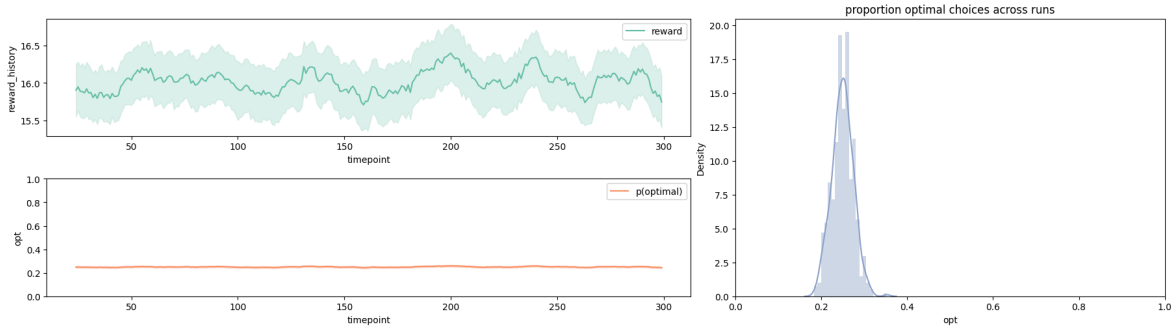
This cell helps you plot the reward history including a smoothed average reward earned by the agents over the last 30 trials

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

def plot_results(results_df, window=25):
    # set up figiure
    palette = iter(sns.color_palette("Set2"))
    fig = plt.figure(constrained_layout=True, figsize=(18, 5))
    gs = fig.add_gridspec(2, 5)

    # add three axes
    rew_ax = fig.add_subplot(gs[0, :-2])
    opt_ax = fig.add_subplot(gs[1, :-2])
    runs_ax = fig.add_subplot(gs[:, -2:])

    # fig, (rew_ax,opt_ax) = plt.subplots(nrows=2,ncols=1,figsize=(18,8))
    smooth_df = results_df.groupby("run").rolling(window, on="timepoint").mean()
    sns.lineplot(
        x="timepoint",
        y="reward_history",
        data=smooth_df,
        ax=rew_ax,
        label="reward",
        color=next(palette),
    )
    rew_ax.legend(loc="upper right")

    sns.lineplot(
```

```
        x="timepoint",
        y="opt",
        data=smooth_df,
        ax=opt_ax,
        label="p(optimal)",
        color=next(palette),
    )
    opt_ax.set_ylim(0, 1)
    opt_ax.legend(loc="upper right")

    sns.distplot(
        results_df.groupby("run")["opt"].mean(), ax=runs_ax, color=next(palette)
    )
    runs_ax.set_title("proportion optimal choices across runs")
    runs_ax.set_xlim(0, 1)
```

Finally, this is an example of the random agent's performance in the environment.

In [ ]:
```
np.random.seed(100)  # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
bandit_means = [0, 50, 10, 4]
bandit_sds = [1] * n_bandits

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of the exp
    agent = RandomAgent(n_bandits)
    np.random.shuffle(bandit_means)  # randomize location of "best"
    bandit = KArmBanditStationary(bandit_means, bandit_sds)  # create a new bana
    for sx in range(n_timesteps):  # for a certain number of time steps
        choice = agent.choose()  # let the agent choose
        reward, opt = bandit.step(choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(sx)

# plot the results
sim_result_df = pd.DataFrame(
    {
        "run": run_history,
        "timepoint": timestep,
        "reward_history": reward_history,
        "opt": opt_history,
    }
)
plot_results(sim_result_df)
```



Here the plot shows the average reward earned for each trial fo the task across the repeated runs/experiments. The lower panel shows the proportion of optimal choices made by the agent across trials. The error bars on both of these plots are bootstrapped confidence intervals across runs of the agent. Finally, the right panel shows a histogram of the proportion of optimal choices made across the runs.

Everything looks as expected here. The random agent makes about the average reward of the task and chooses the optimal choice about 25% of the time.

Can you use what we have learned in lecture to do better?

Problem 4 (15 points)

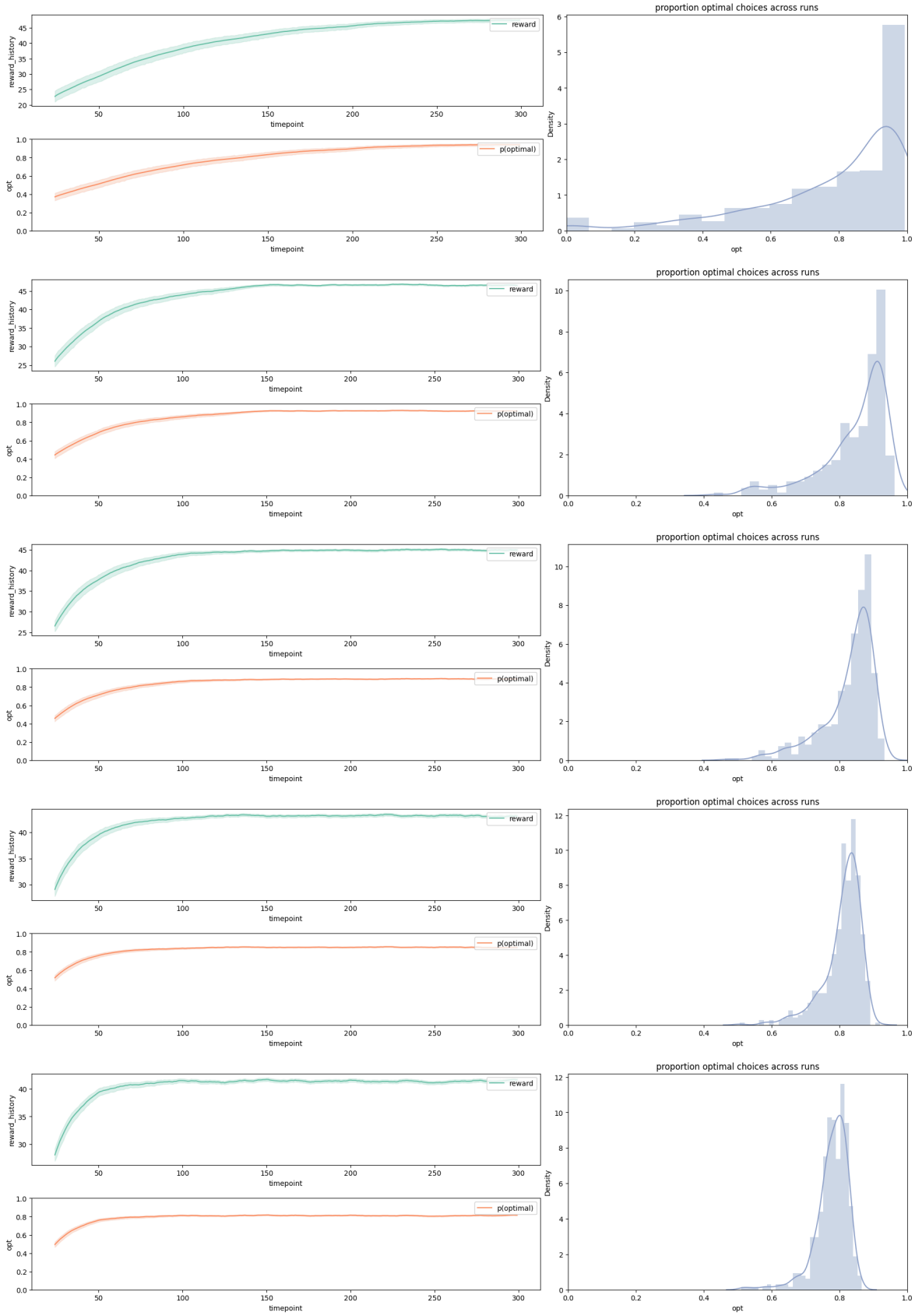
Create a new agent class based on `RandomAgent` called `EpsilonGreedyAgentIncremental()` which keeps track of the average reward earned from each draw of the bandit. This agent will include a parameter `epsilon` which will determine the probability of choosing a random action, otherwise it should choose the best so far. To update the value of each bandit use the incremental implementation of mean-tracking which was first introduced on the slides blending monte carlo methods (this is a incremental approach to calculating the mean as a new observation arrive). Make a plot similar to the ones above showing the performance of the agent on the three measures we have considered when the environment is initialized in the same way (i.e., means are a random shuffling of [0.,50.,10.,4.] with sd=1). Next, show with a couple examples how changes in epsilon determine the shape of that plot. You should show the final code for your agent and the plots along with a single markdown cell describing your solution (1-2 paragraphs). In your answer be sure to explain if your new agent does better than the random agent and why you think that is the case. In all cases run your agent for 300 time steps and average over 500 runs.

In [ ]:
```
from typing import List

class RewardRecords():
    def __init__(self) -> None:
        self.records = []
        self.rec_num = 0
```

```
    @property
    def mean(self) -> float:
        return sum(self.records) / self.rec_num \
            if self.rec_num != 0 else 0.0

    def append(self, reward) -> None:
        self.records.append(reward)
        self.rec_num += 1

class EpsilonGreedyAgentIncremental(RandomAgent):
    def __init__(self, k, epsilon) -> None:
        super().__init__(k)
        self.rewards: List[RewardRecords] = \
            [ RewardRecords() for _ in range(k) ]
        self.epsilon = epsilon

    def choose(self) -> int:
        if np.random.ranf() < self.epsilon:
            return np.random.randint(self.num_actions)
        else:
            return np.argmax([ r.mean for r in self.rewards ])

    def learn(self, reward, action) -> None:
        self.rewards[action].append(reward)
```

In [ ]:

```
np.random.seed(100)
n_timesteps = 300
n_runs = 500

n_bandits = 4
bandit_means = [0, 50, 10, 4]
bandit_sds = [1] * n_bandits

for epsilon in [ 0.05, 0.10, 0.15, 0.2, 0.25 ]:
    run_history, timestep, reward_history, opt_history = [], [], [], []
    for run in range(n_runs):
        agent = EpsilonGreedyAgentIncremental(n_bandits, epsilon)
        np.random.shuffle(bandit_means)
        bandit = KArmBanditStationary(bandit_means, bandit_sds)
        for sx in range(n_timesteps):
            choice = agent.choose()
            reward, opt = bandit.step(choice)
            run_history.append(run)
            timestep.append(sx)
            reward_history.append(reward)
            opt_history.append(opt)
            agent.learn(reward, choice)
    sim_result_df = pd.DataFrame({
        "run": run_history,
        "timepoint": timestep,
        "reward_history": reward_history,
        "opt": opt_history,
    })
    plot_results(sim_result_df)
```



My Answer:

By selecting the optimal action and sometimes exploring new actions, the new agent is able to find the best action as time goes by.

In the shown cases, we can see that the larger we set the parameter *epsilon*, the faster we can make sure which action is the best.

However, because our *epsilon* does not decay as time goes by, the agent still tries to explore other actions, which leads to the result that the average rewards are lower than we got from smaller *epsilon*.

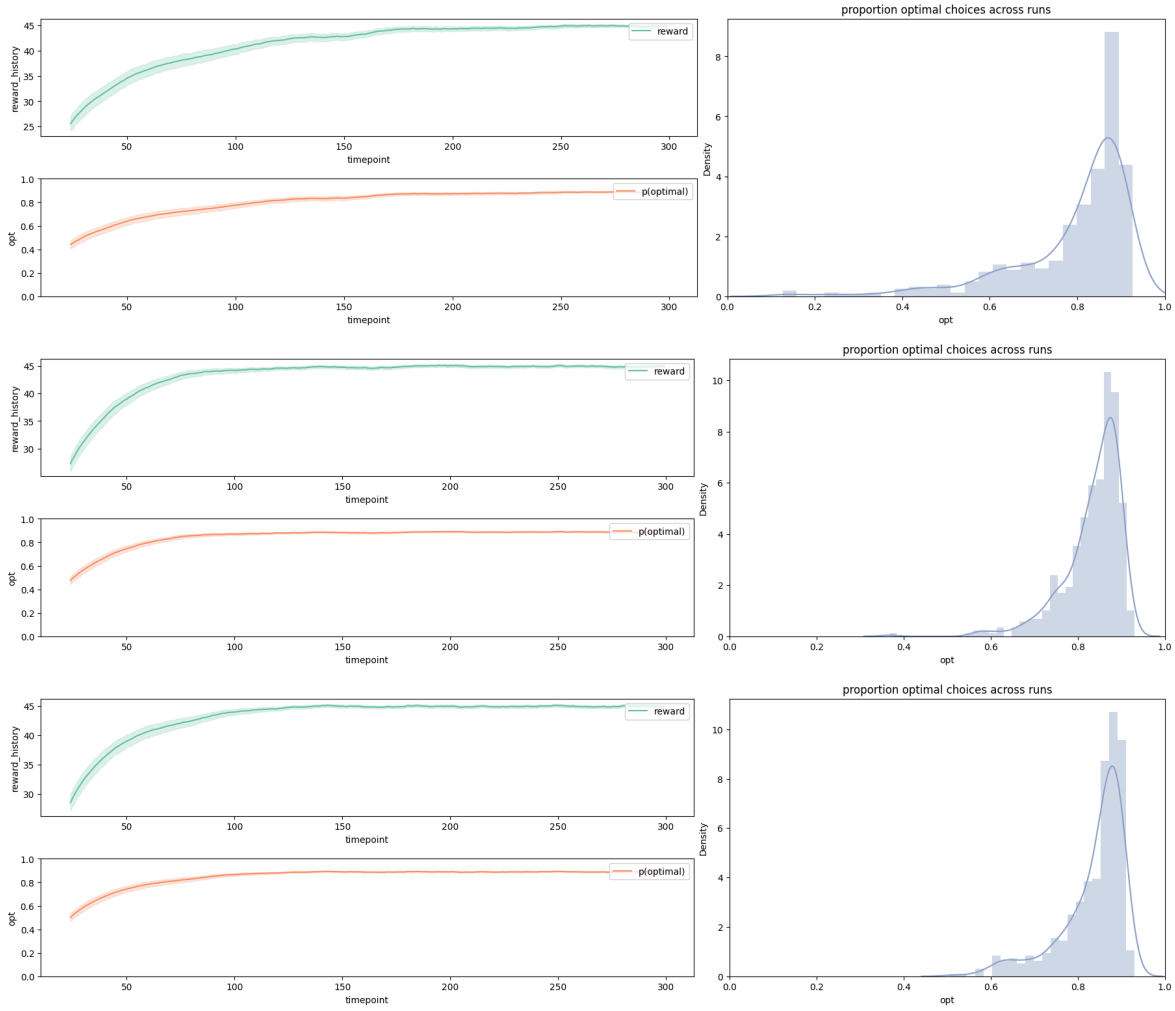
Problem 5 (15 points)

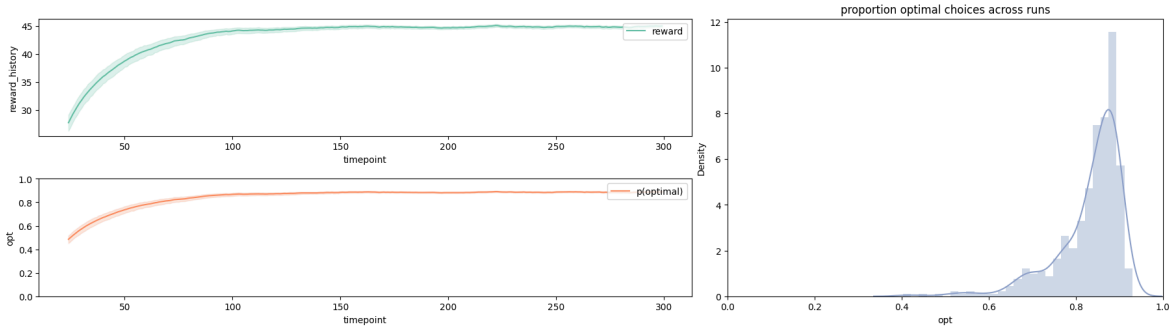
Create a new agent class based on `RandomAgent` called `EpsilonGreedyAgentConstant()` which keeps track of the average reward earned from each draw of the bandit. This agent will be nearly identical to `EpsilonGreedyIncremental()`. However, in addition to the parameter `epsilon` which will determine the probability of choosing a random action, this agent should use the "constant step size" update rule related to temporal-different learning to update the value of each action. The step size parameter (`alpha`) should be added as a new input parameter to your agent (hint: small values of this parameter are often better). Make a plot similar to the ones above showing the performance of the agent on the three measures we have considered. Is the performance of this agent the same or different than the previous agent you coded for this environment? Try a few parameter combinations with your agent and in your response show 1-2 examples to help make your point. You should show the final code for your agent and the plots along with a single markdown cell describing your solution (1-2 paragraphs). Be sure that your answer includes the answer to this key question: Does the constant agent out perform the incremental agent? And does it do better than the random agent? You don't need to do statistics but just a general visual comparison of the quality of the algorithsm is enough. In all cases run your agent for 300 time steps and average over 500 runs.

In [ ]:
```
class EpsilonGreedyAgentConstant(RandomAgent):
    def __init__(self, k, epsilon, alpha) -> None:
        super().__init__(k)
        self.rewards = [ 0.0 ] * k
        self.epsilon = epsilon
        self.alpha   = alpha

    def choose(self) -> int:
        if np.random.ranf() < self.epsilon:
            return np.random.randint(self.num_actions)
        else:
            return np.argmax(self.rewards)

    def learn(self, reward, action) -> None:
        self.rewards[action] += \
            self.alpha * (reward - self.rewards[action])
```

In [ ]:
```
np.random.seed(100)
n_timesteps = 300
n_runs = 500

n_bandits = 4
```

```
bandit_means = [0, 50, 10, 4]
bandit_sds = [1] * n_bandits
epsilon = 0.15

for alpha in [ 0.1, 0.25, 0.4, 0.65 ]:
    run_history, timestep, reward_history, opt_history = [], [], [], []
    for run in range(n_runs):
        agent = EpsilonGreedyAgentConstant(n_bandits, epsilon, alpha)
        np.random.shuffle(bandit_means)
        bandit = KArmBanditStationary(bandit_means, bandit_sds)
        for sx in range(n_timesteps):
            choice = agent.choose()
            reward, opt = bandit.step(choice)
            run_history.append(run)
            timestep.append(sx)
            reward_history.append(reward)
            opt_history.append(opt)
            agent.learn(reward, choice)
    sim_result_df = pd.DataFrame({
        "run": run_history,
        "timepoint": timestep,
        "reward_history": reward_history,
        "opt": opt_history,
    })
    plot_results(sim_result_df)
```





My Answer:

If we set *alpha* to a smaller value, the rewards are updated slower.
While if we set *alpha* to a larger value, the average rewards will be impacted by the exploration even if the agent has already found the best action.

But it depends on the scale or the parameter *epsilon*.
If the exploration rate is low, then even we set a large *alpha*, the average rewards will not be impacted significantly.

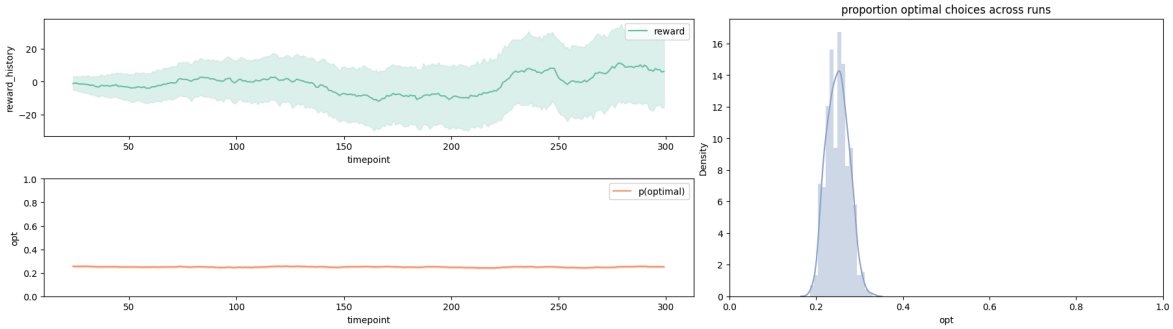
Problem 6 (15 points)

Below is a new type of bandit environment based on `KArmBanditStationary` class where the reward probabilities of each bandit change over time. This is sometimes known as a "restless bandit" (see the Daw et al. 2013 paper on explore exploit mentioned in Lecture). The idea is that on each time step the mean reward of each action should be modified up or down by a sample from a Gaussian distribution (e.g., $\mu_{t+1} = \mu_t + \mathcal{N}(0, 20)$). I have called the new class `KArmBanditRestless`. The mean of the arms is itself drawn intially from a random normal distribution as well. Using this environment (with the number of arms set to 4), test the `RandomAgent()`, `EpsilonGreedyAgentConstant()` and `EpsilonGreedyAgentIncremental()` agents. You may want to play with the alpha parameter of the incremental agent to see if you can find a particularly good setting. Show the final code for your agent, plots showing the average reward the agent earns over time, along with a markdown cell describing your solution in 1-2 paragraphs. Which agent performs better in this environment? Is this different than the conclusion you made from the previous environment? Be sure to answer these two questions in your response. In all cases run your agent for 300 time steps and average over 500 runs.

In [ ]:
```
class KArmBanditRestless:
    def __init__(self, k, mu=0, sigma=2, sd=2, walk_sd=30):
        self.k = k
        self.action_means = np.random.normal(mu, sigma, k)
        self.action_sds = sd
        self.walk_sd = walk_sd
        self.optimal = np.argmax(self.action_means)
```

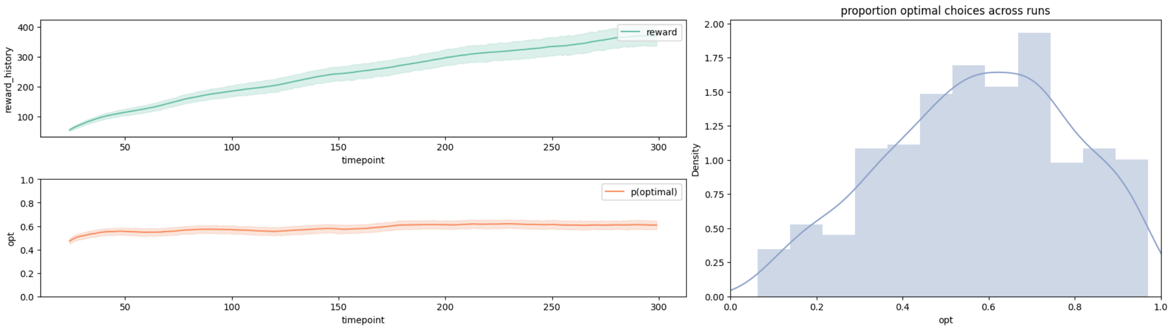
```
    def step(self, action):
        out = (
            np.random.normal(self.action_means[action], self.action_sds),
            action == self.optimal,
        )
        self.action_means = self.action_means + np.random.normal(
            0.0, self.walk_sd, self.k
        )
        self.optimal = np.argmax(self.action_means)
        return out
```

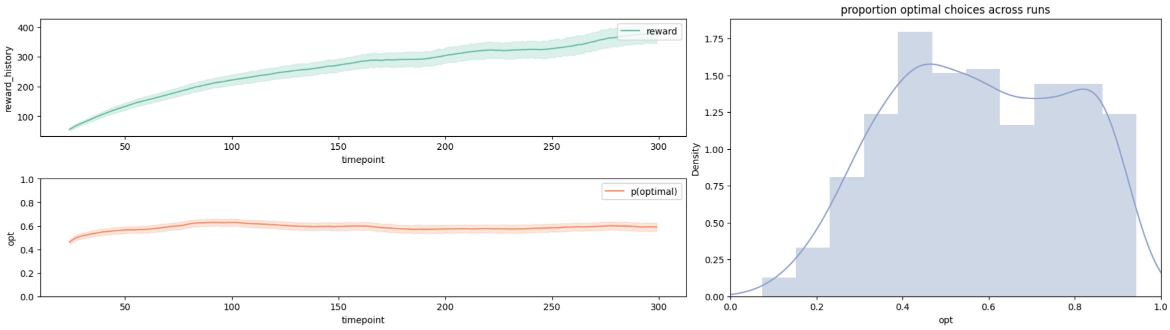
In [ ]:
```
# RandomAgent
n_bandits = 4
bandit = KArmBanditRestless(n_bandits)

np.random.seed(100)
n_timesteps = 300
n_runs = 500

run_history, timestep, reward_history, opt_history = [], [], [], []
for run in range(n_runs):
    agent = RandomAgent(n_bandits)
    bandit = KArmBanditRestless(n_bandits)
    for sx in range(n_timesteps):
        choice = agent.choose()
        reward, opt = bandit.step(choice)
        run_history.append(run)
        timestep.append(sx)
        reward_history.append(reward)
        opt_history.append(opt)
sim_result_df = pd.DataFrame({
    "run": run_history,
    "timepoint": timestep,
    "reward_history": reward_history,
    "opt": opt_history,
})
plot_results(sim_result_df)
```

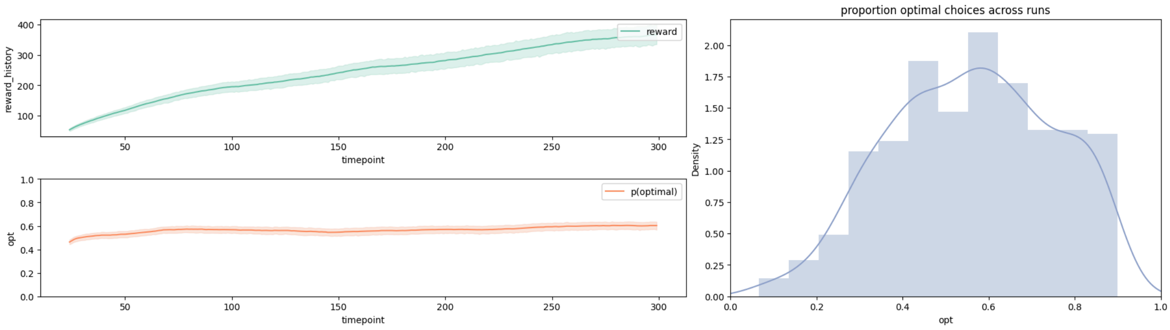


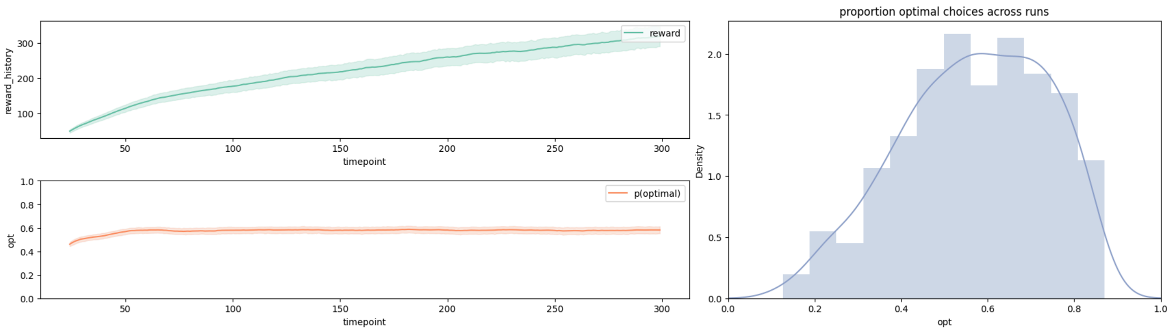
In [ ]:
```
# EpsilonGreedyAgentIncremental
n_bandits = 4
bandit = KArmBanditRestless(n_bandits)

np.random.seed(100)
n_timesteps = 300
n_runs = 500

for epsilon in [ 0.05, 0.10, 0.15, 0.2, 0.25 ]:
    run_history, timestep, reward_history, opt_history = [], [], [], []
    for run in range(n_runs):
```

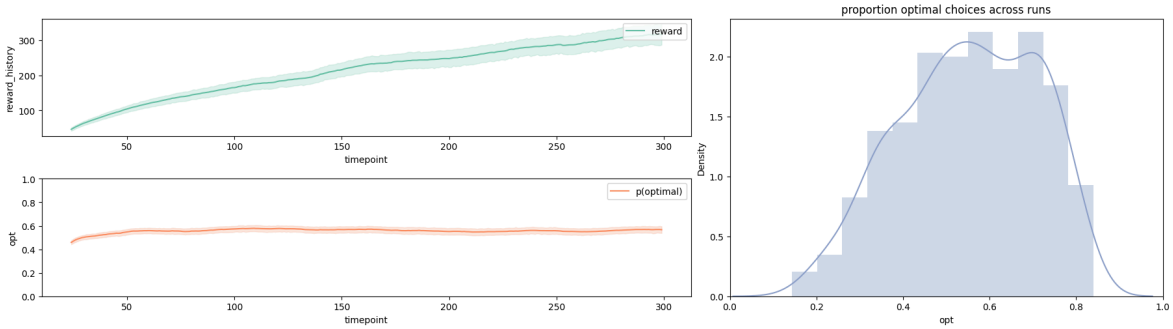
```
        agent = EpsilonGreedyAgentIncremental(n_bandits, epsilon)
        bandit = KArmBanditRestless(n_bandits)
        for sx in range(n_timesteps):
            choice = agent.choose()
            reward, opt = bandit.step(choice)
            run_history.append(run)
            timestep.append(sx)
            reward_history.append(reward)
            opt_history.append(opt)
            agent.learn(reward, choice)
    sim_result_df = pd.DataFrame({
        "run": run_history,
        "timepoint": timestep,
        "reward_history": reward_history,
        "opt": opt_history,
    })
    plot_results(sim_result_df)
```



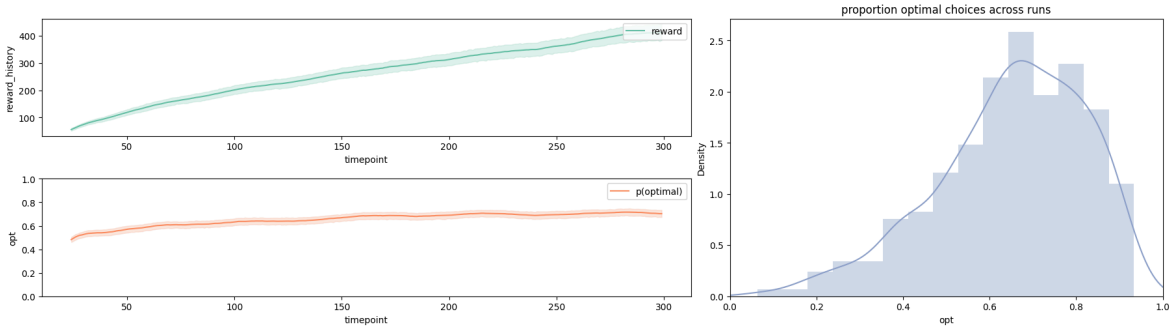


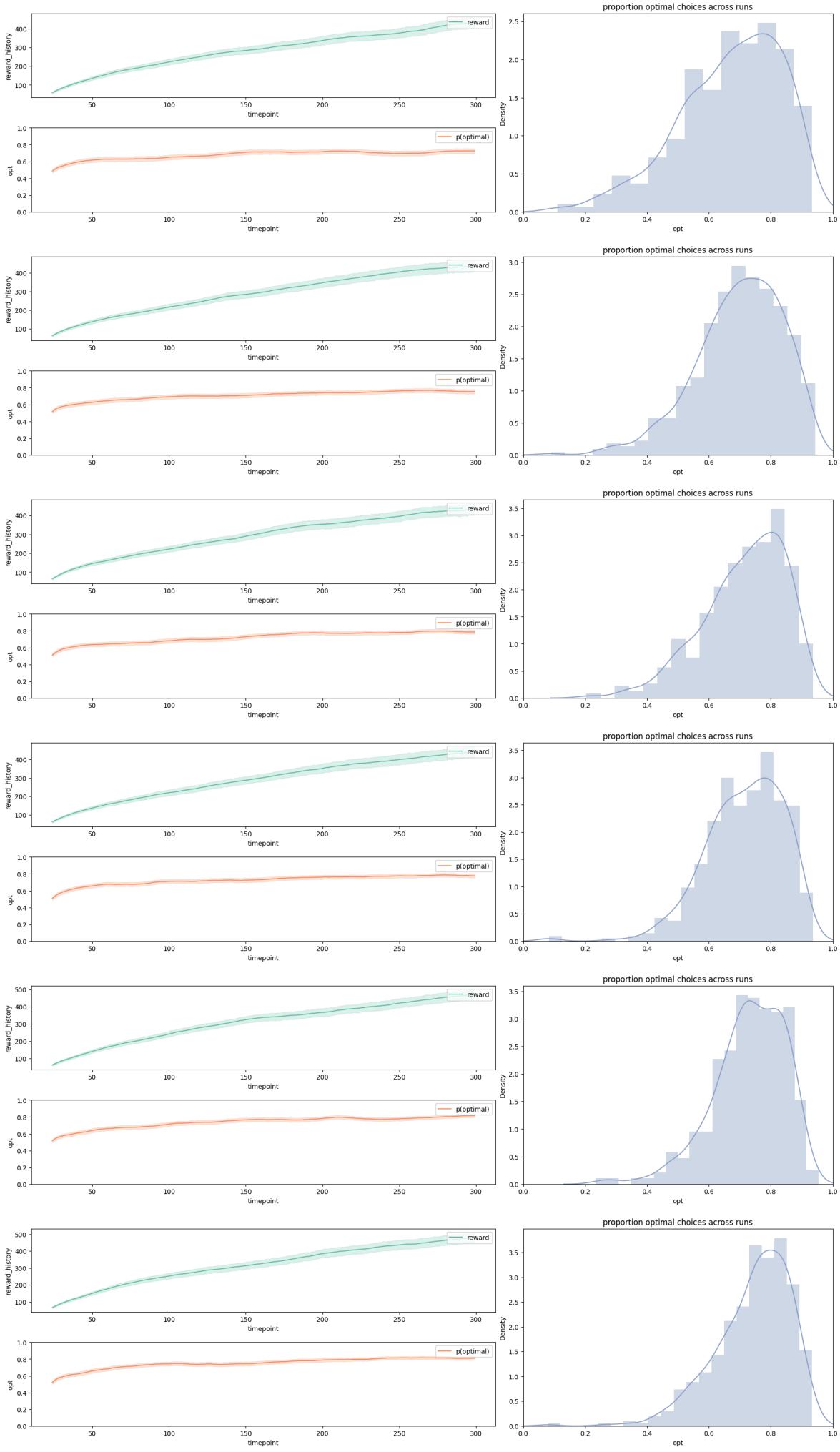


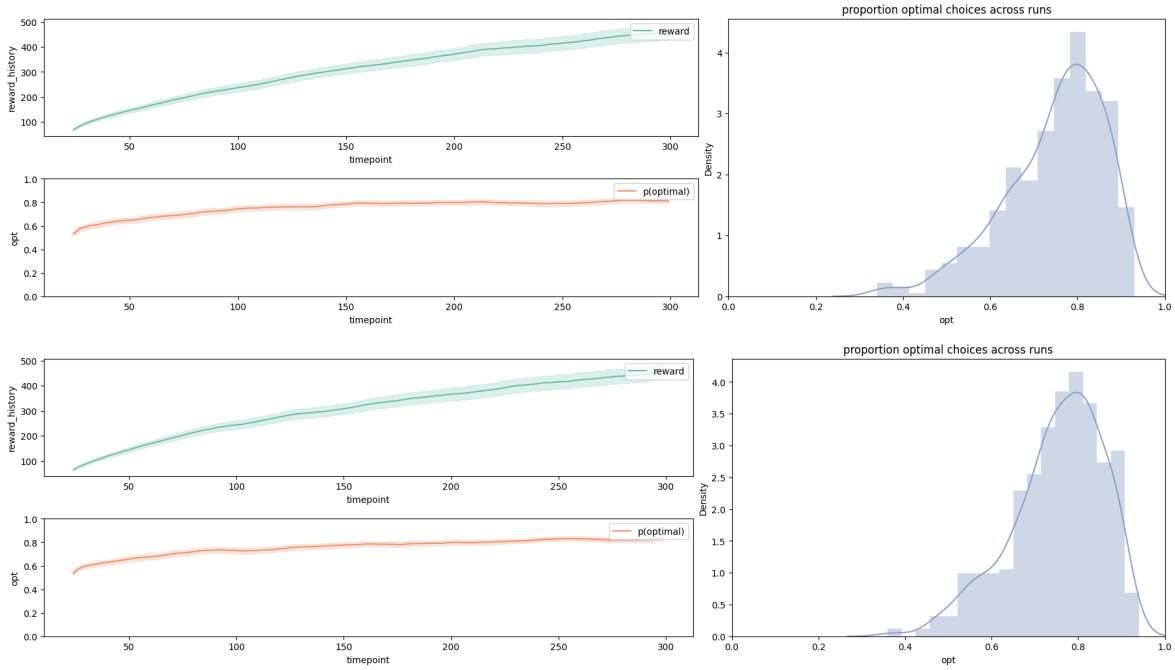




In [ ]:
```
# EpsilonGreedyAgentConstant
n_bandits = 4
bandit = KArmBanditRestless(n_bandits)

np.random.seed(100)
n_timesteps = 300
n_runs = 500
epsilon = 0.1

for alpha in [ 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 ]:
    run_history, timestep, reward_history, opt_history = [], [], [], []
    for run in range(n_runs):
        agent = EpsilonGreedyAgentConstant(n_bandits, epsilon, alpha)
        bandit = KArmBanditRestless(n_bandits)
        for sx in range(n_timesteps):
            choice = agent.choose()
            reward, opt = bandit.step(choice)
            run_history.append(run)
            timestep.append(sx)
            reward_history.append(reward)
            opt_history.append(opt)
            agent.learn(reward, choice)
    sim_result_df = pd.DataFrame({
        "run": run_history,
        "timepoint": timestep,
        "reward_history": reward_history,
        "opt": opt_history,
    })
    plot_results(sim_result_df)
```




reward
p(optimal)
reward_history
opt
timepoint
proportion optimal choices across runs
Density



My Answer:

From the `EpsilonGreedyAgentIncremental()` experiment, we can see that *epsilon*=0.1 works best when facing restless bandits.
So I set *epsilon*=0.1 to find the best *alpha* of `EpsilonGreedyAgentConstant()`.

The result is not surprising.
Since the means of each bandits change as time goes by, the importance of new rewards are higher than the past, so the the agent performs better when we set *alpha* larger.

Problem 7 (15 points)

For this problem, we return to the grid world task we considered in Part A. Specifically, you should combine the ideas about explore-exploit and incremental learning of q-values to implement a temporal-difference solution the to grid world problem you explored in Part A of the homework. You can reuse the code from that notebook in building your solution. This solution should also obtain similar results to the policy-iteration and monte-carlo versions you explored, but is learned more incrementally and online.

The basic setup of the GridWorld environment is provided again for you below. Your solution to this problem should involve modifications to the solution to the Monte-Carlo problems in Part A. In particular, instead of waiting until a particular episode ends to update the values of the Q-values, use the Q-learning equation to incrementally updates these values as an episode unfolds. To balance exploration and exploitation try any of the methods you developed in the earlier parts of this assignment.

As a reminder the question for updating the Q values in Q-learning is as follows:

$Q(s,a) = Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$

The pseudo code for this algorithm is:

```
Initialize, for all $s \in S$, $a \in A(s)$:
    $Q(s,a)$ <- arbitrary

Repeat many times (for each episode):
    a) Initialize $s$ at the start state
    b) Repeat
        1. Choose action $a$ from $s$ using policy dervived from Q
values in that state (e.g., SoftMax)
        2. Take action $a$, observe $r$, $s'$
        3. Update $Q(s,a)$
           Find max Q(s',a') over all action a' in state s'
           $Q(s,a) = Q(s,a) + \alpha [r + \gamma max Q(s',a') -
Q(s,a)]$
           $s$ <- $s'$
       Until $s$ is the goal state
```

In [ ]:
```
# import the gridworld library
import numpy as np
import random
import math
import statistics
from copy import deepcopy
from IPython.display import display, Markdown, Latex, HTML
from gridworld import GridWorld, random_policy
```

In [ ]:
```
gridworld = [
    ["o", "o", "o", "o", "o", "o", "o", "x", "g"],
    ["o", "x", "x", "o", "x", "x", "o", "x", "o"],
    ["o", "x", "x", "o", "x", "x", "o", "x", "o"],
    ["o", "x", "x", "o", "x", "x", "o", "o", "o"],
    ["o", "x", "x", "o", "x", "x", "x", "o", "o"],
    ["s", "o", "o", "o", "o", "o", "o", "o", "x"],
]  # the problem described above, 'x' is a wall, 's' is start, 'g' is goal, and


mygrid = GridWorld(gridworld)
mygrid.raw_print()  # print out the grid world
mygrid.index_print()  # print out the indicies of each state
mygrid.coord_print()  # print out the coordinates of each state (helpful in your

# define the rewards as a hash table
rewards = {}

# mygrid.transitions contains all the pairwise state-state transitions allowed i
# for each state transition intialize the reward to zero
for start_state in mygrid.transitions:
    for action in mygrid.transitions[start_state].keys():
        next_state = mygrid.transitions[start_state][action]
        rewards[str([start_state, action, next_state])] = 0.0

# now set the reward for moving up into state 8 (the goal state) to +10
rewards[str([17, "up", 8])] = 10
```

```
# now set the penalty for walking off the edge of the grid and returning to stat
for sx in [0, 1, 2, 3, 4, 5, 6, 7]:
    rewards[str([sx, "up", 45])] = -1
for sx in [0, 9, 18, 27, 36, 45]:
    rewards[str([sx, "left", 45])] = -1
for sx in [45, 46, 47, 48, 49, 50, 51, 52, 53]:
    rewards[str([sx, "down", 45])] = -1
for sx in [8, 17, 26, 35, 44, 53]:
    rewards[str([sx, "right", 45])] = -1
```

Welcome to your new Grid World!

Raw World Layout

o	o	o	o	o	o	o	x	g
o	x	x	o	x	x	o	x	o
o	x	x	o	x	x	o	x	o
o	x	x	o	x	x	o	o	o
o	x	x	o	x	x	x	o	o
s	o	o	o	o	o	o	o	x

Indexes of each grid location as an id number

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53

Indexes of each grid location as a tuple

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	(0,8)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)

The following code sets up the major things you need to track. Note that unlike in the Monte Carlo solution you do not need a separate accounting of the returns as you are updating those to the Q-values directly.

Also you shouldn't need to update the policy table until you have run many episodes through the maze. The final update to the `policy_table` should just

to be to print out your final greedy solution and compare it to the solutions you obtained in Part A.

In [ ]:
```
starting_state = 45
goal_state = 8  # terminate the MC roll out when you get to this state
GAMMA = 0.9
EPSILON = 0.2


# set up initial data strucutres that might be useful for you
# q(s,a) - the q-values for each action in each state
def zero_q_values():
    qvals = {"up": 0.0, "right": 0.0, "down": 0.0, "left": 0.0}
    return qvals


q_value_table = [
    [zero_q_values() for i in range(mygrid.ncols)] for j in range(mygrid.nrows)
]

# pi - the policy table
policy_table = [
    [random_policy() for i in range(mygrid.ncols)] for j in range(mygrid.nrows)
]
display(Markdown("**Initial (randomized) policy**"))
mygrid.pretty_print_policy_table(policy_table)
```

Initial (randomized) policy

→	↓	→	↑	←	←	←	▉	↓
→	▉	▉	←	▉	▉	→	▉	→
↓	▉	▉	←	▉	▉	←	▉	↓
↑	▉	▉	↑	▉	▉	→	←	→
↓	▉	▉	→	▉	▉	▉	→	→
←	↓	↑	↑	←	←	→	↑	▉

In [ ]:
```
ALPHA = 0.6
ITERATIONS = 50000
PRINT_EVERY = 12500

for i in range(ITERATIONS):
    state = starting_state
    while state != goal_state:
        sx, sy = mygrid.index_to_coord(state)

        if random.random() < EPSILON:
            action = random.choice(list(q_value_table[sx][sy].keys()))
        else:
            action = max(q_value_table[sx][sy].items(), key=lambda i: i[1])[0]
        next_state = mygrid.transitions[state][action]

        r = rewards[str([state, action, next_state])]
        sxn, syn = mygrid.index_to_coord(next_state)
        max_q_next_state = max(q_value_table[sxn][syn].values())
```

```
        q_value_table[sx][sy][action] += \
            ALPHA * (r + GAMMA*max_q_next_state - q_value_table[sx][sy][action])
        state = next_state

    if i % PRINT_EVERY == 0:
        for sy in range(mygrid.ncols):
            for sx in range(mygrid.nrows):
                policy_table[sx][sy] = {"up": 0.0, "right": 0.0, "down": 0.0, "l
                best_action = max(q_value_table[sx][sy].items(), key=lambda q: q
                policy_table[sx][sy][best_action] = 1.0
        display(Markdown(f"**Improved policy interation {i}**"))
        mygrid.pretty_print_policy_table(policy_table)

for sy in range(mygrid.ncols):
    for sx in range(mygrid.nrows):
        policy_table[sx][sy] = {"up": 0.0, "right": 0.0, "down": 0.0, "left": 0.
        best_action = max(q_value_table[sx][sy].items(), key=lambda q: q[1])[0]
        policy_table[sx][sy][best_action] = 1.0
display(Markdown("**Improved policy**"))
mygrid.pretty_print_policy_table(policy_table)
```

Improved policy interation 0

→	→	→	→	→	→	→	▉	↑
↑	▉	▉	↑	▉	▉	↑	▉	↑
↑	▉	▉	↑	▉	▉	↑	▉	↑
↑	▉	▉	↑	▉	▉	↑	↑	↑
↑	▉	▉	↑	▉	▉	▉	↑	↑
↑	↑	↑	↑	↑	↑	↑	↑	▉

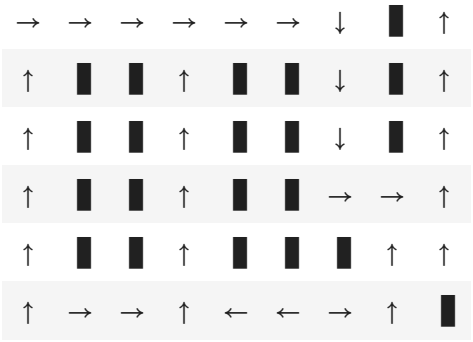
Improved policy interation 12500

→	→	→	→	→	→	↓	▉	↑
↑	▉	▉	↑	▉	▉	↓	▉	↑
↑	▉	▉	↑	▉	▉	↓	▉	↑
↑	▉	▉	↑	▉	▉	→	→	↑
↑	▉	▉	↑	▉	▉	▉	↑	↑
↑	←	←	↑	↑	↑	→	↑	▉

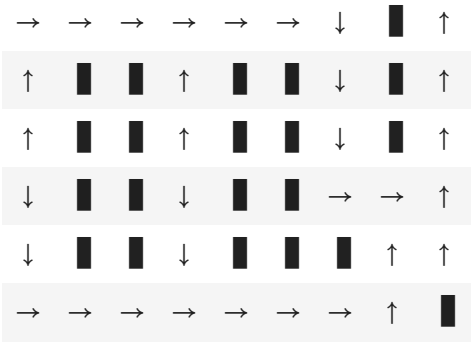
Improved policy interation 25000

→	→	→	→	→	→	↓	▉	↑
↑	▉	▉	↑	▉	▉	↓	▉	↑
↑	▉	▉	↑	▉	▉	↓	▉	↑
↑	▉	▉	↑	▉	▉	→	→	↑
↑	▉	▉	↑	▉	▉	▉	↑	↑
↑	→	→	↑	←	←	→	↑	▉

Improved policy interation 37500

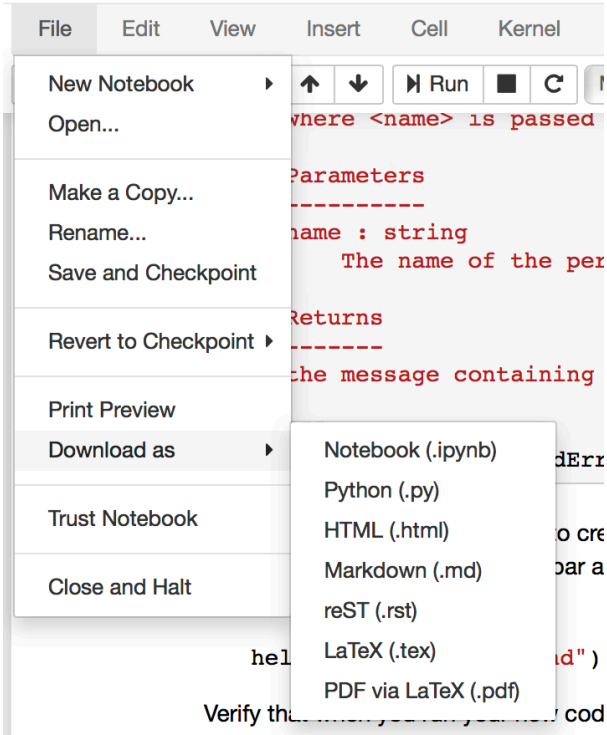


Improved policy



Turning in homework

When you are finished with this notebook. Save your work in order to turn it in. To do this select *File->Download As...->HTML*.



You can turn in your assignments using Gradescope for the course (available on https://home.nyu.edu). **Make sure you complete all parts (A and B) of this homework.**