

# 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-spring2019@nyucll.org](mailto:instructors-ccm-spring2019@nyucll.org)

This homework is due before midnight on March 23, 2020.

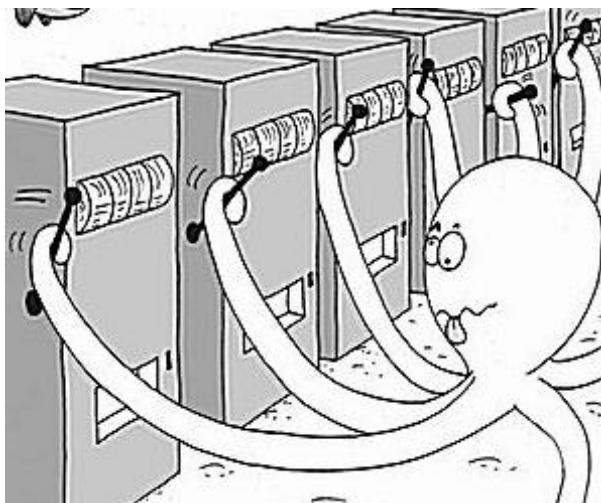
## Learning and deciding in an unknown world

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 an 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 restaurant 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 numpy as np
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import random
import math
import seaborn as sns
```

## 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]),
```

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 opt
```

Out[ ]: (14.054668904656618, 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 [ ]: def plot_results(results_df, window=25):
    # set up figure
    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='r')
    rew_ax.legend(loc="upper right")

    sns.lineplot(x='timepoint', y='opt', data=smooth_df, ax=opt_ax, label='p(optimal)')
    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

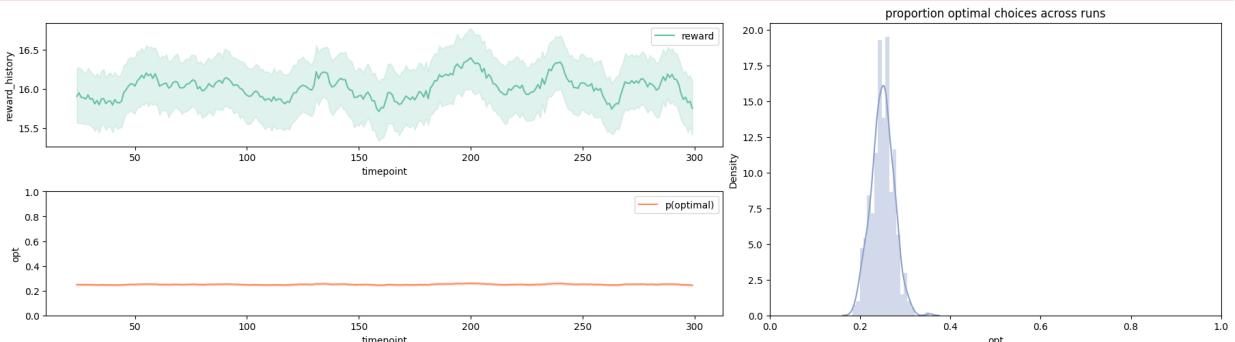
reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs): # repeat a given number random repetitions of the experim
    agent = RandomAgent(n_bandits)
    np.random.shuffle(bandit_means) # randomize location of "best"
    bandit = KArmBanditStationary(bandit_means,bandit_sds) # create a new bandit
    for i 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(i)

# 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)

```

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).  
 warnings.warn(msg, FutureWarning)



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 [ ]: class EpsilonGreedyAgentIncremental():
    def __init__(self, k, epsilon):
        self.num_actions = k
        self.epsilon = epsilon
        self.reward = np.zeros(k)
        self.action = np.zeros(k)
        # you could add parameters to your agent here
        pass

    def choose(self):
        if np.random.rand() < self.epsilon or np.sum(self.action) == 0:
            return np.random.randint(self.num_actions)
        else:
            return np.argmax(self.reward/self.action)

    def learn(self, reward, action):
        self.action[action] += 1
        self.reward[action] += (reward-self.reward[action])/self.action[action]
        pass
```

```
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
epsilons = [0,0.01,0.1,0.5]
```

```

for epsilon in epsilons:

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

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

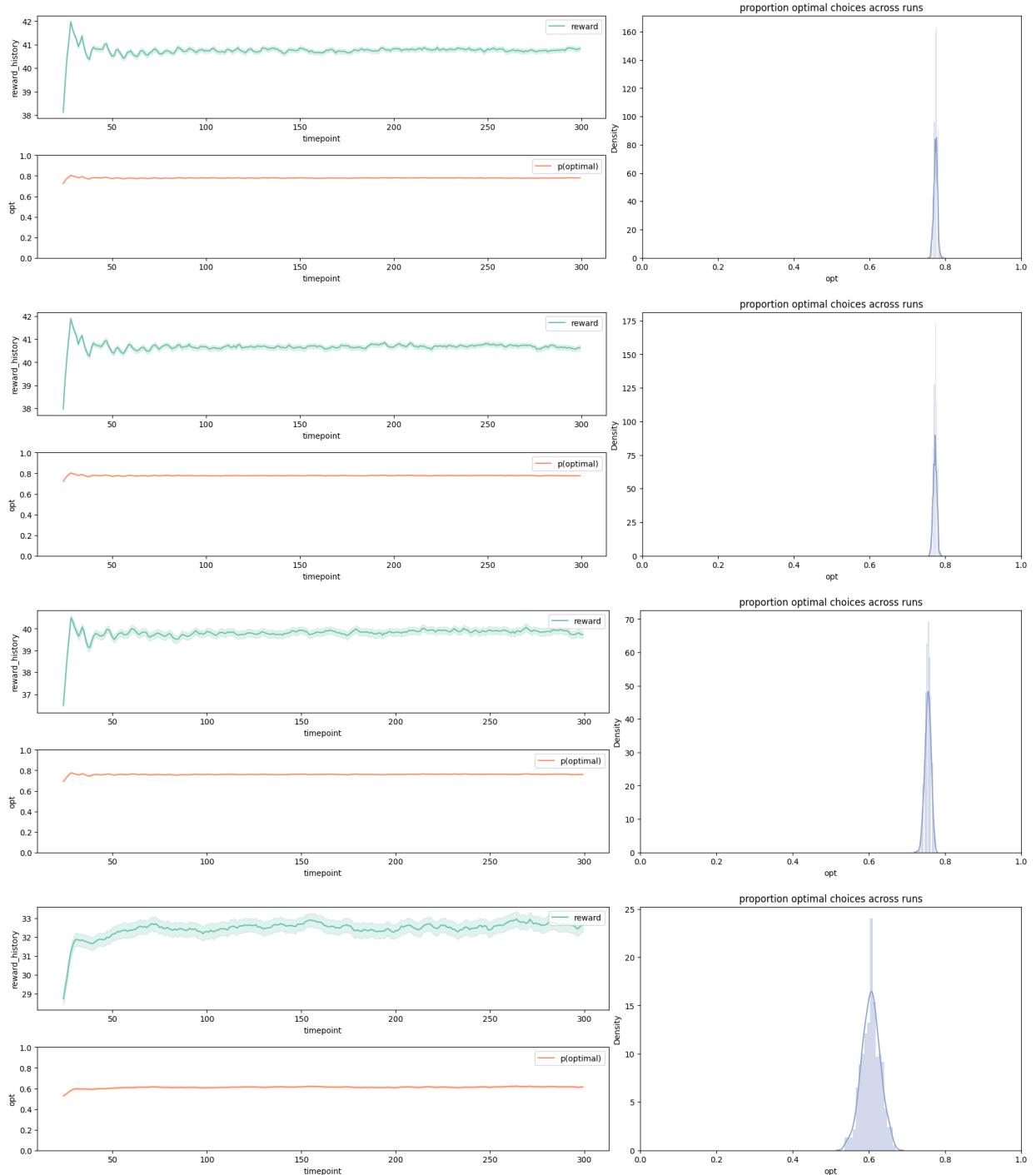
    # plot the results
    sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep, "reward_hi
    plot_results(sim_result_df)

```

```

C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\3807414818.py:14: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\3807414818.py:14: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\3807414818.py:14: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\3807414818.py:14: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)

```



The best performance for the epsilons I tested was at  $\epsilon = 0.01$ , with it being closely followed by  $\epsilon = 0$ . It seems the epsilon with no randomness did very well, but the slight bit of randomness did improve performance. For the greater epsilon  $.1$  and  $.5$  the performance dropped significantly, in both density and proportion of optimal choices. This may be only because of how few bandits there are and there being mostly non-zero rewards. I feel like higher epsilon would perform better for problems with more bandits.

Regardless, all performed better than the RandomAgent. Unsurprisingly, it would choose the optimal choice about  $.25$  of the time, which makes sense as the optimal choice is  $1/4$  chance of getting it randomly. The Epsilon Agent did better by learning where the best rewards are and being more likely to stay where it is optimal, moreso for lower 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 algorithm is enough. In all cases run your agent for 300 time steps and average over 500 runs.

```
In [ ]: class EpsilonGreedyAgentConstant():
    def __init__(self, k, epsilon, alpha):
        self.num_actions = k
        self.epsilon = epsilon
        self.alpha = alpha
        self.reward = np.zeros(k)
        self.action = np.zeros(k)
        # you could add parameters to your agent here
        pass

    def choose(self):
        if np.random.rand() < self.epsilon or np.sum(self.action) == 0:
            return np.random.randint(self.num_actions)
        else:
            return np.argmax(self.reward/self.action)

    def learn(self, reward, action):
        self.action[action] += 1
        self.reward[action] += (reward-self.reward[action])*self.alpha
        pass
```

```
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
alphas = [.01,.1,.25]
epsilons = [0,0.01,0.1]

for epsilon in epsilons:
    for alpha in alphas:
        reward_history = []
        opt_history = []
        run_history = []
        timestep = []

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

        # plot the results
        sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep, "reward": reward_history, "opt": opt_history})
        plot_results(sim_result_df)
        print("epsilon: ", epsilon, "alpha: ", alpha)
        #This doesn't print with graphs, but tells me the order at Least
    
```

```

C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning:
`distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0 alpha:  0.01
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning:
`distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0 alpha:  0.1

```

```
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0 alpha:  0.25

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0.01 alpha:  0.01

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0.01 alpha:  0.1

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0.01 alpha:  0.25

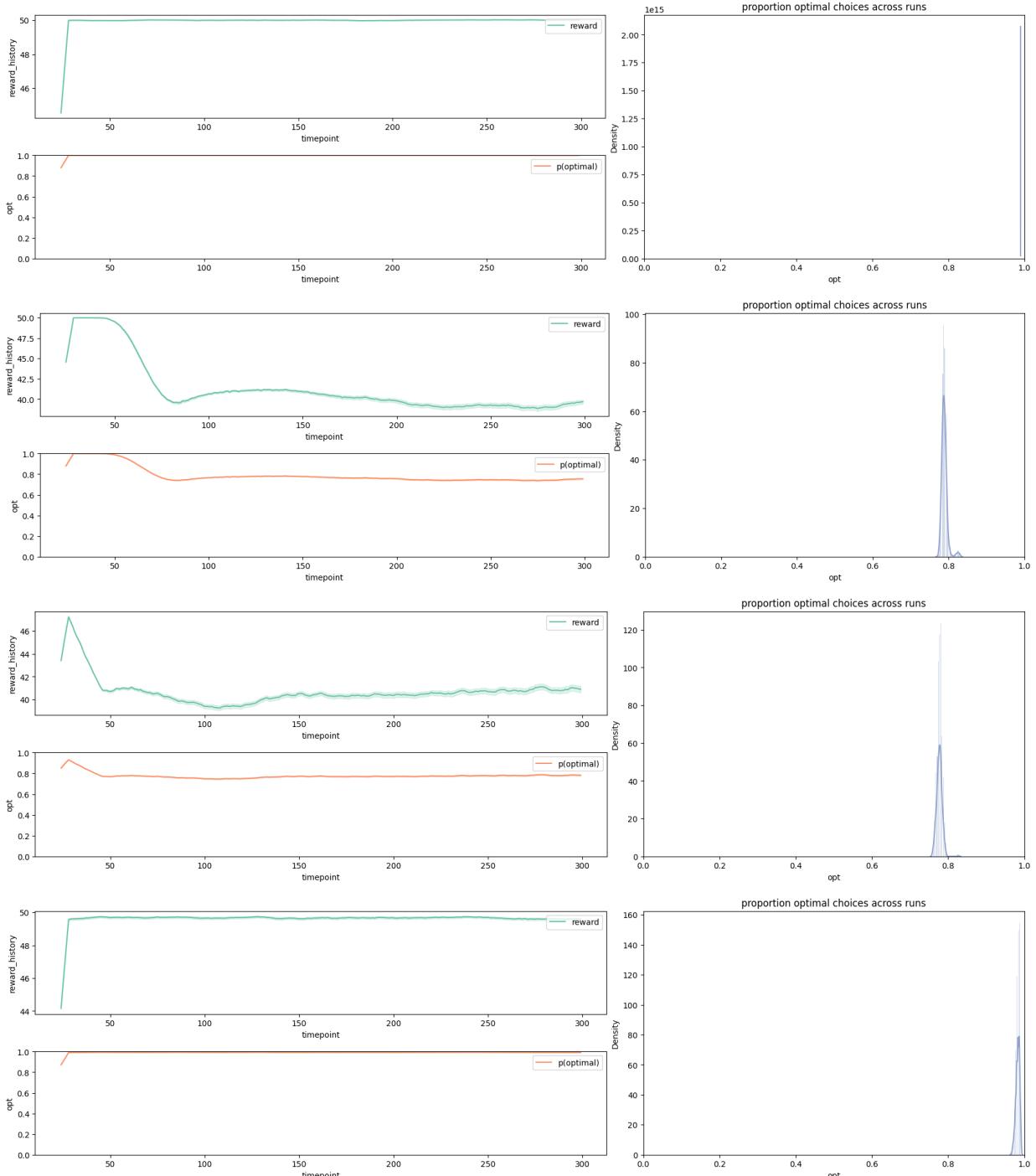
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0.1 alpha:  0.01

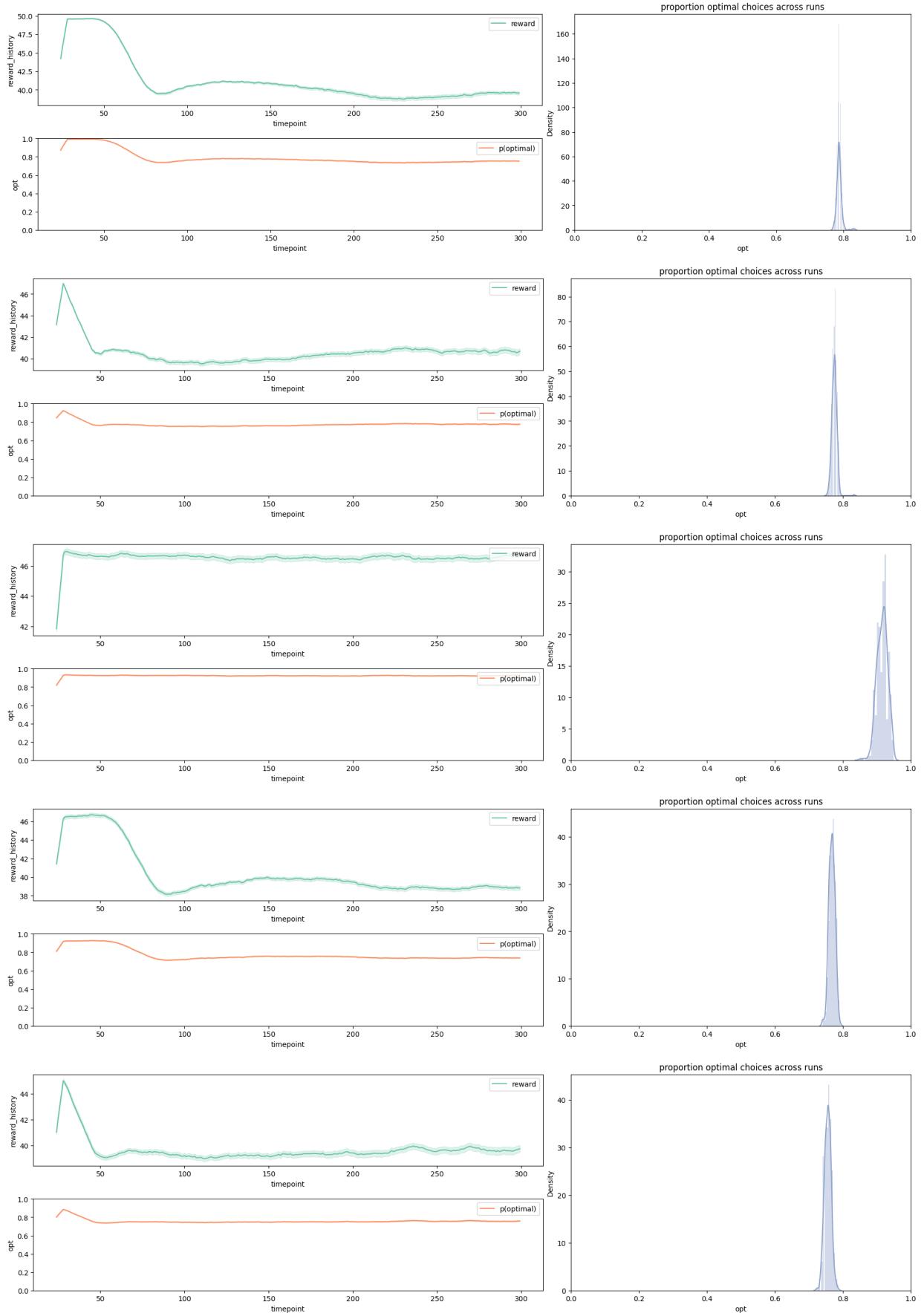
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0.1 alpha:  0.1
```

```
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
```

```
warnings.warn(msg, FutureWarning)
```

epsilon: 0.1 alpha: 0.25





The best models seem to have even lower alpha, with the best being at  $\epsilon = 0$  and  $\alpha = .01$ , getting the optimal 100% of the time somehow. Regardless, all did at least a bit better than Epsilon Incremental with the same  $\epsilon$ , and waaaay better than RandomAgent.

All reward histories for when alpha = .1 and .25 have a spike up before going back down, whereas the alpha=.01 graphs spike up and stay up, along with having a very large proportion of optimal choices, which makes sense due to step size.

## 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 intiallly 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 [ ]: # initialize this bandit like this
n_bandits = 4
bandit = KArmBanditRestless(n_bandits)
```

### RANDOM AGENT

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

# parameters of simulation
n_timesteps = 300
n_runs = 500
```

```

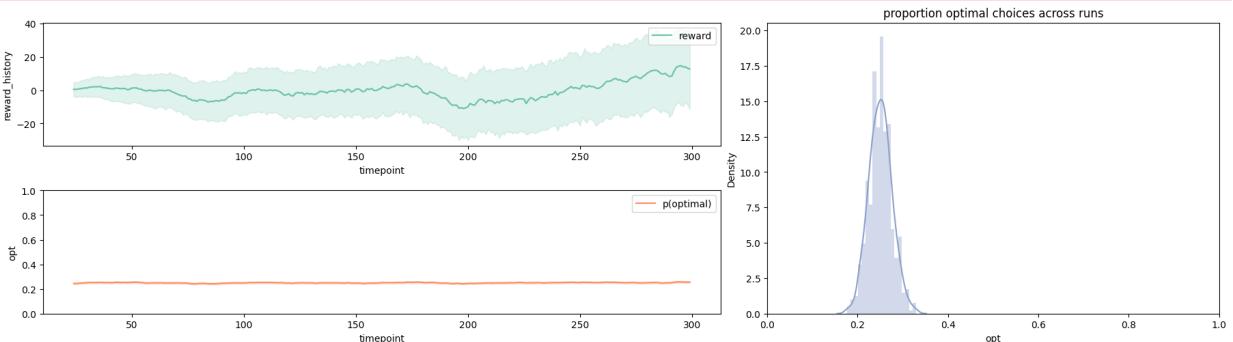
reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs): # repeat a given number random repetitions of the experim
    agent = RandomAgent(n_bandits)
    np.random.shuffle(bandit_means) # randomize location of "best"
    bandit = KArmBanditRestless(n_bandits) # create a new bandit
    for i 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(i)

# 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)

```

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).  
 warnings.warn(msg, FutureWarning)



Though hard to judge based on reward history since the scale is different, it seems the bandit being restless does not impact the RandomAgent too much, but does seem to decrease performance slightly when looking at density and the spread of the reward history. This insignificant change at best is a result of the same issue as before changing the bandits: the Agent doesn't learn it just picks randomly which makes it a little impervious to the bandit changes.

## EPSILON INCREMENTAL

```

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

# parameters of simulation
n_timesteps = 300
n_runs = 500
epsilons = [0, 0.01, 0.1, 0.5]

for epsilon in epsilons:

```

```

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

for run in range(n_runs): # repeat a given number random repetitions of the exper
    agent = EpsilonGreedyAgentIncremental(n_bandits,epsilon)
    np.random.shuffle(bandit_means) # randomize location of "best"
    bandit = KArmBanditRestless(n_bandits) # create a new bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward,choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

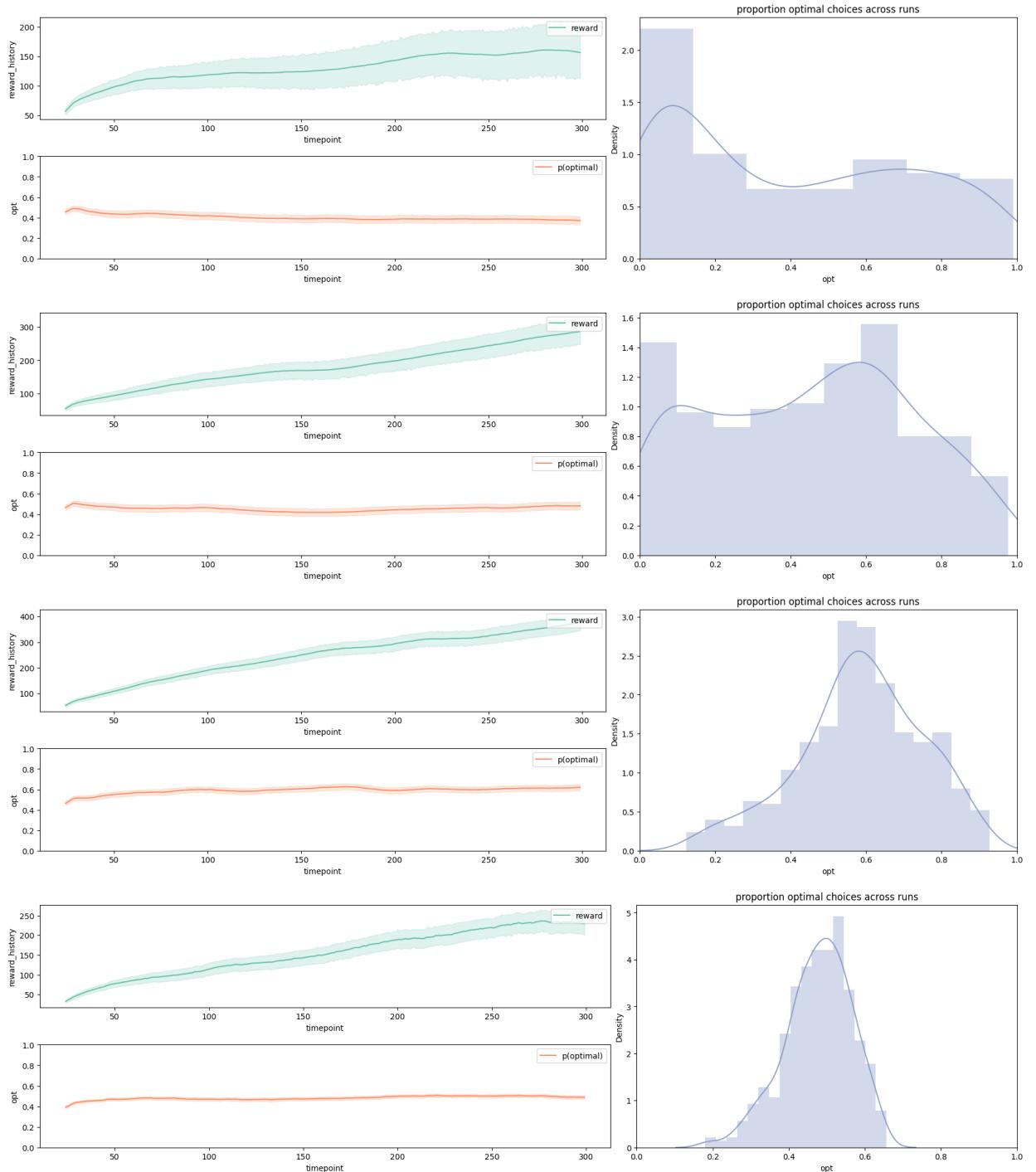
# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep, "reward_hi
plot_results(sim_result_df)

```

```

C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\3807414818.py:14: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning:
`distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\3807414818.py:14: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning:
`distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\3807414818.py:14: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning:
`distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\3807414818.py:14: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning:
`distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)

```



Many more interesting changes happened here. To start, the cases where epsilon is low seem to take on a uniform distribution shape a bit more than higher epsilon which seem more Gaussian, an effect of the Restless Bandit. With the new bandit, it seems the reward history is monotonic compared to previously, getting much higher rewards as well. The proportion of optimal choices is much more spread as well compared to the steady peaks of the Stationary Bandit.

Finally, it seems that instead of epsilon = .01 or 0 being way better Agents, now it seems the epsilon=.1 is performing much better in both reward and distribution, though at lower densities due to spread compared to Stationary Bandits.

## EPSILON CONSTANT

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

# parameters of simulation
n_timesteps = 300
n_runs = 500
alphas = [.01,.1,.25]
epsilons = [0,0.01,0.1,0.5]

for epsilon in epsilons:
    for alpha in alphas:
        reward_history = []
        opt_history = []
        run_history = []
        timestep = []

        for run in range(n_runs): # repeat a given number random repetitions of the experiment
            agent = EpsilonGreedyAgentConstant(n_bandits,epsilon, alpha)
            np.random.shuffle(bandit_means) # randomize Location of "best"
            bandit = KArmBanditRestless(n_bandits) # create a new bandit
            for i in range(n_timesteps): # for a certain number of time steps
                choice = agent.choose() # let the agent choose
                reward, opt = bandit.step(choice)
                agent.learn(reward,choice)
                reward_history.append(reward)
                opt_history.append(opt)
                run_history.append(run)
                timestep.append(i)

        # plot the results
        sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep, "reward": reward_history, "opt": opt_history})
        plot_results(sim_result_df)
        print("epsilon: ", epsilon, "alpha: ", alpha)
        #This doesn't print with graphs, but tells me the order at least
```

```
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning:
`distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0 alpha:  0.01
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning:
`distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning:
invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0 alpha:  0.1
```

```
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0 alpha:  0.25

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0.01 alpha:  0.01

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0.01 alpha:  0.1

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0.01 alpha:  0.25

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0.1 alpha:  0.01

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon:  0.1 alpha:  0.1
```

```

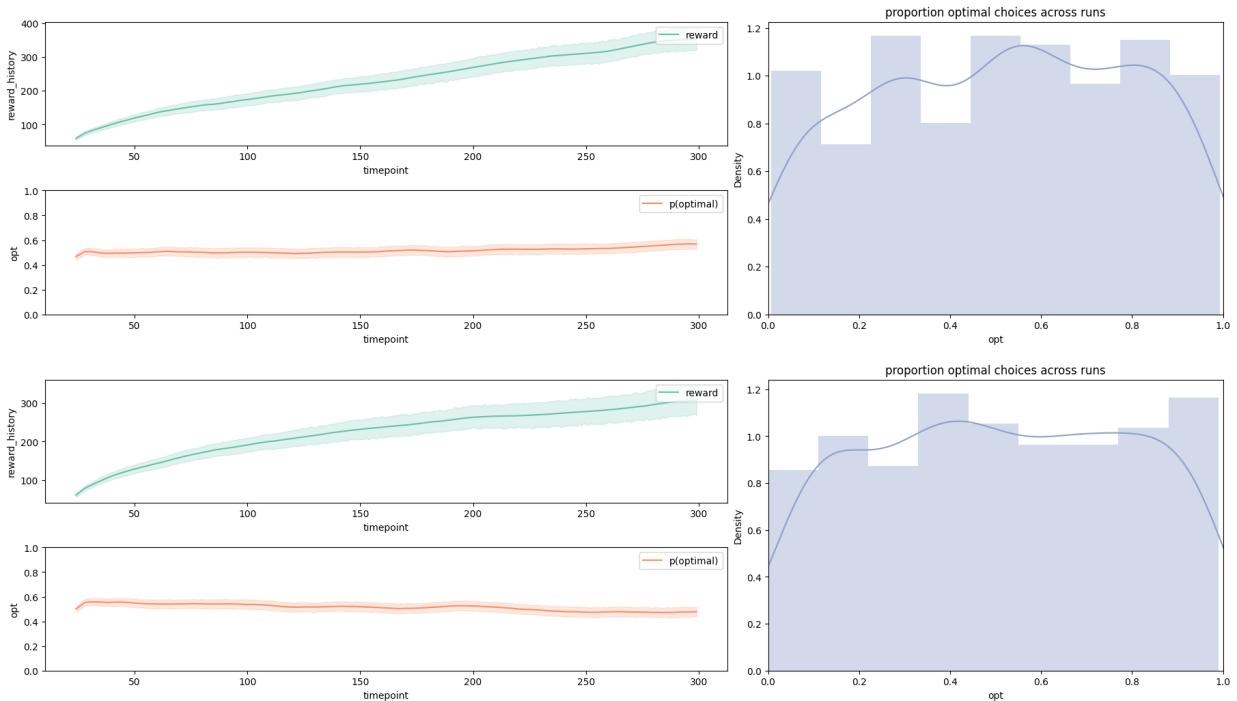
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon: 0.1 alpha: 0.25

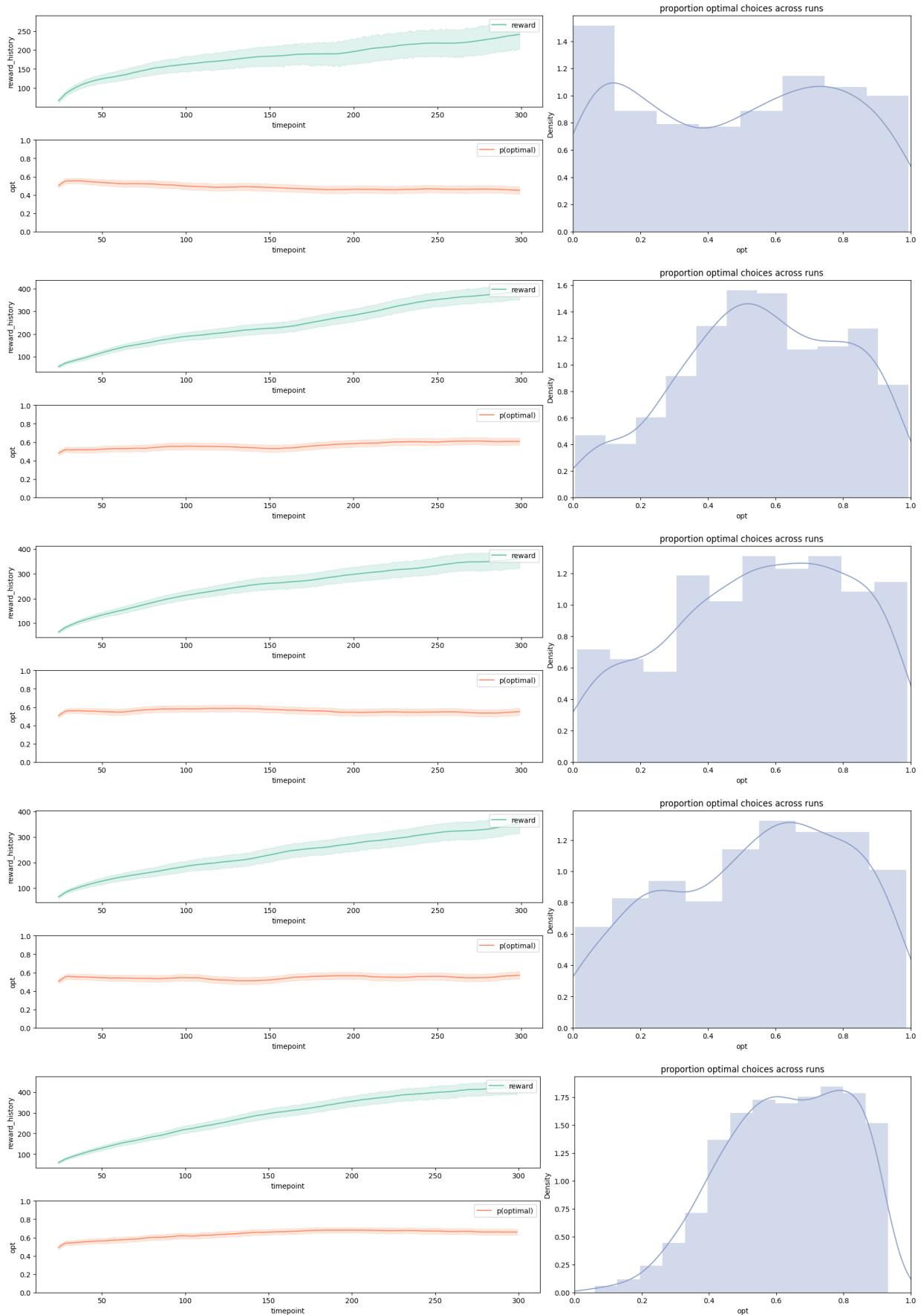
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon: 0.5 alpha: 0.01

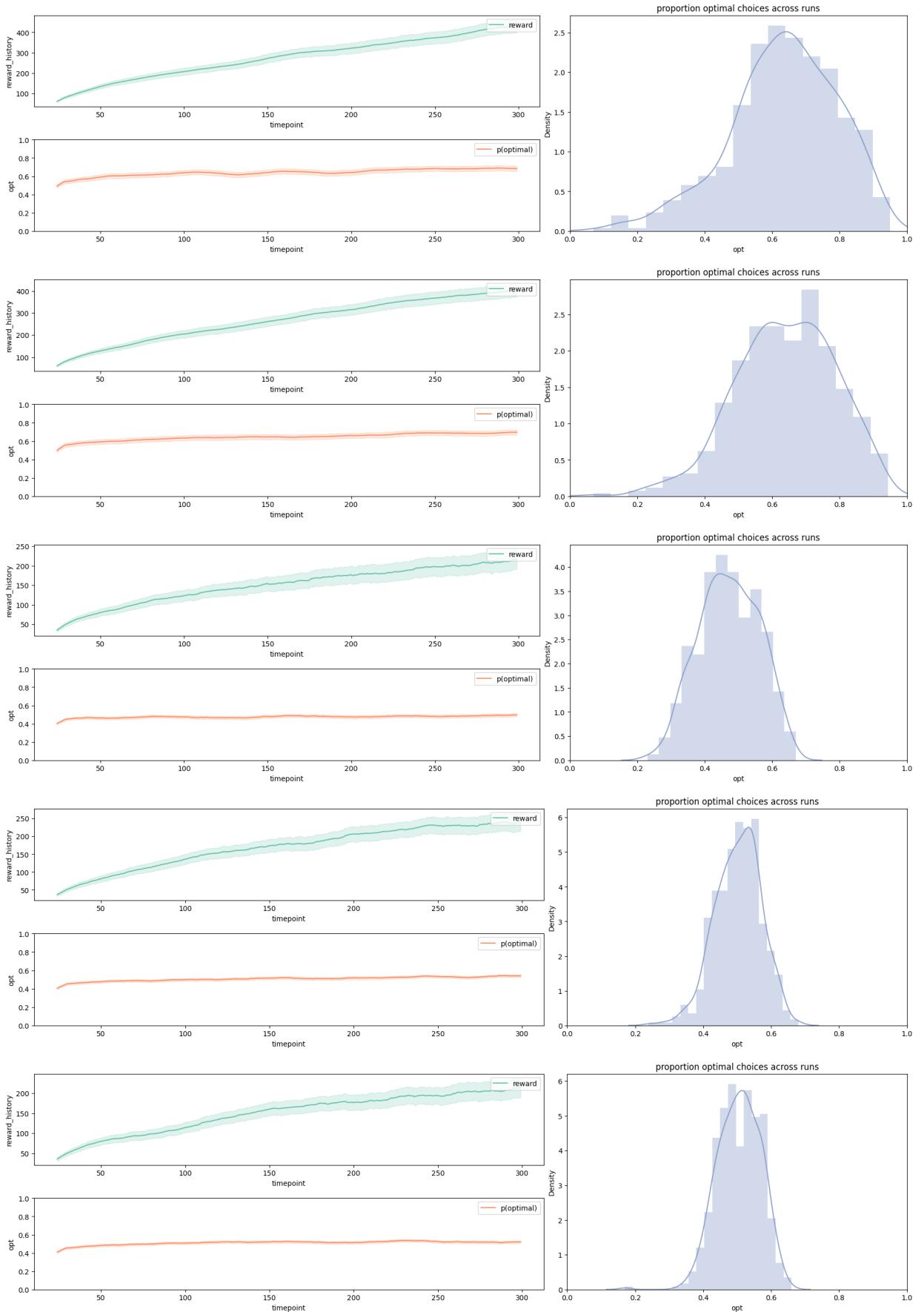
c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
C:\Users\Michael\AppData\Local\Temp\ipykernel_6360\1737093098.py:15: RuntimeWarning: invalid value encountered in divide
    return np.argmax(self.reward/self.action)
epsilon: 0.5 alpha: 0.1

c:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
    warnings.warn(msg, FutureWarning)
epsilon: 0.5 alpha: 0.25

```







Here there are a couple of differences to consider. Just based off reward history,  $\epsilon$  = .1 and either  $\alpha$  = .1 or .25 seems to be the best, which is already different from using the Stationary Bandit where  $\epsilon$ =0 and  $\alpha$ =.01 was the best. The behaviors also seem to vary

between different epsilon: when starting with epsilon=0 the affects were similar to the Stationary Bandit where the lower alphas were better, but as epsilon increased this rule changed as well. Furthermore, the lower epsilon weren't necessarily better than higher epsilon unlike in Stationary.

## 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 to the 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 update 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  $s \in S, a \in A(s)$ :  $Q(s, a) \leftarrow$  arbitrary Repeat many times (for each episode): a) Initialize  $s$  at the start state b) Repeat 1. Choose action  $a$  from  $s$  using policy derived 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  $s'$   $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max Q(s', a') - Q(s, a)]$   $s \leftarrow s'$  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 'o

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 code)

# define the rewards as a hash table
rewards=[]

# mygrid.transitions contains all the pairwise state-state transitions allowed in the
# for each state transition initialize 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 state 45 (
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)

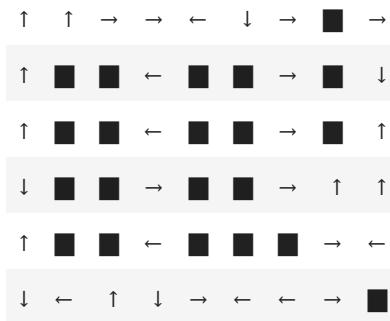
The following code sets up the major things you need to track. Note that unlike in the Monte Carlo solution you do not need a separate accounting of the returns as you are updating those to the Q-values directly. Also you shouldn't need to update the policy table until you have run many episodes through the maze. The final update to the `policy\_table` should just 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 structures 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 [ ]: # this is a helper function that will take a set of q-values and convert them into a greedy policy
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

```

```

In [ ]: ALPHA = .1
ITERATIONS = 500000 # this may need to be 100,000 or more!
PRINT_EVERY = 10000 # how often to print out our progress
INIT_VAL = -99999.0 #initialize unsampled q values to a very small number (pessimistic)
random.seed(5000) # try multiple random seed to verify your code works

for i in range(ITERATIONS):
    current_state = starting_state

    while current_state != goal_state:
        sx, sy = mygrid.index_to_coord(current_state)
        best_value = INIT_VAL
        best_action = None
        if random.random() < EPSILON:
            best_action = random.choice(list(mygrid.transitions[current_state].keys()))
        else:
            for action in mygrid.transitions[current_state].keys():
                qval = q_value_table[sx][sy][action]
                if qval > best_value:
                    best_value = qval
                    best_action = action

        next_state = mygrid.transitions[current_state][best_action]
        reward = rewards[str([current_state, best_action, next_state])]
        nsx, nsy = mygrid.index_to_coord(next_state)
        best_next_value = INIT_VAL
        for action in mygrid.transitions[next_state].keys():
            qval = q_value_table[nsx][nsy][action]
            if qval > best_next_value:
                best_next_value = qval
                best_next_action = action

        # update q-value-table
        q_value_table[sx][sy][best_action] += ALPHA * (reward + GAMMA * q_value_table[nsx][nsy][best_next_action])
        current_state = next_state

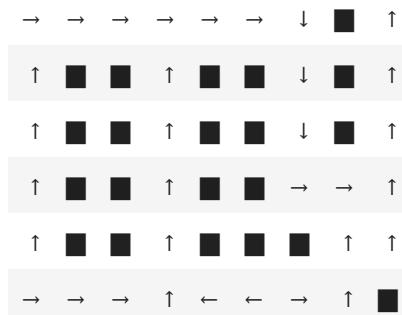
    # improve policy
    for sx in range(len(q_value_table)):
        for sy in range(len(q_value_table[sx])):
            policy_table[sx][sy] = be_greedy(q_value_table[sx][sy])

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

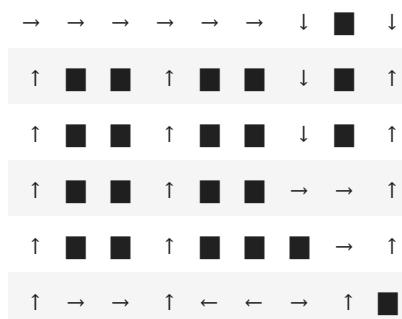
```

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

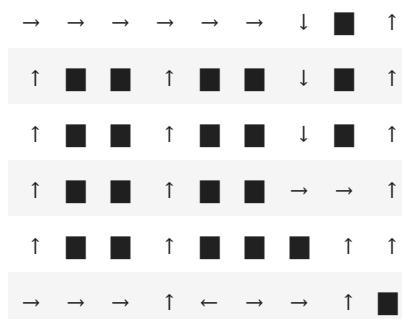
### Improved policy interation 0



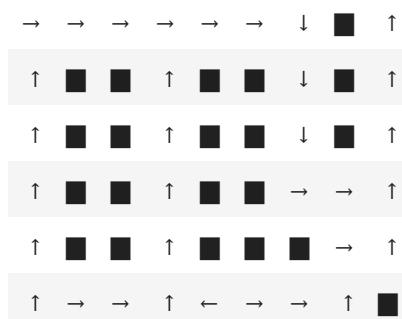
### Improved policy interation 10000



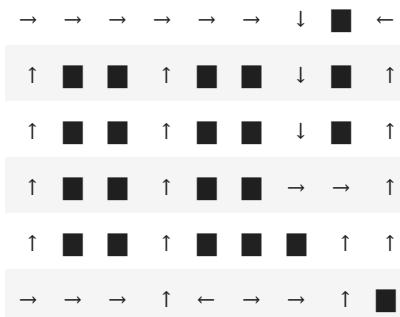
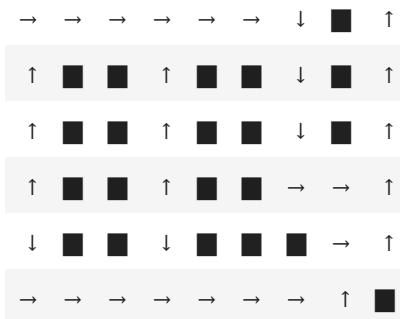
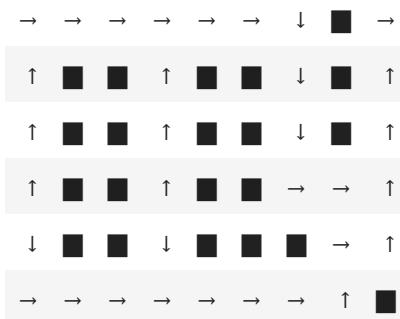
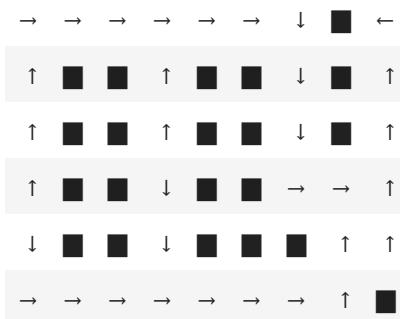
### Improved policy interation 20000

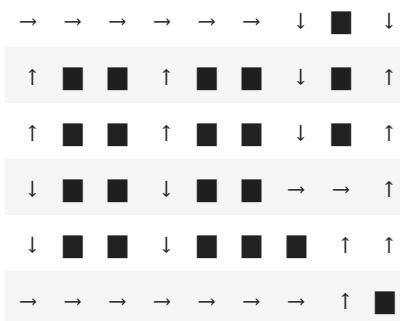
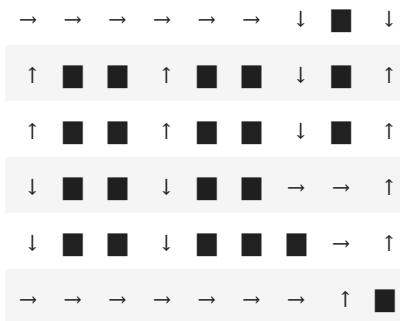
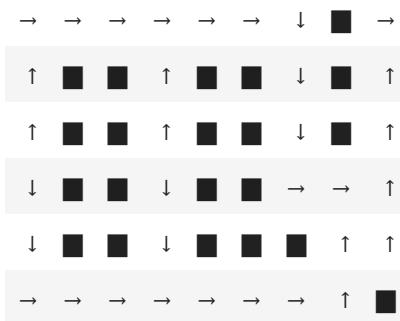
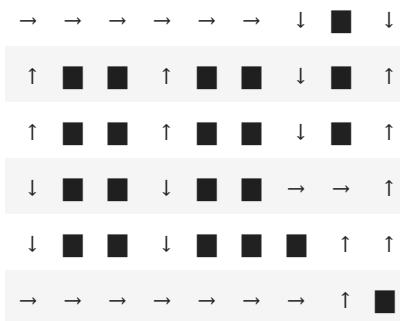


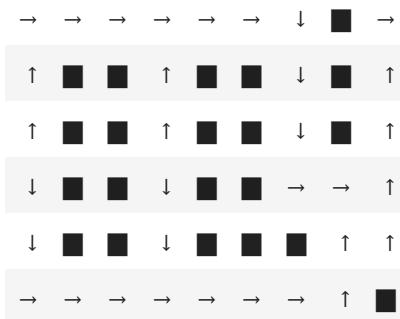
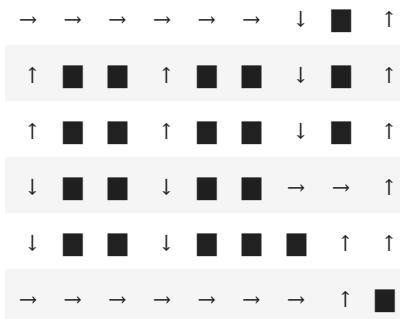
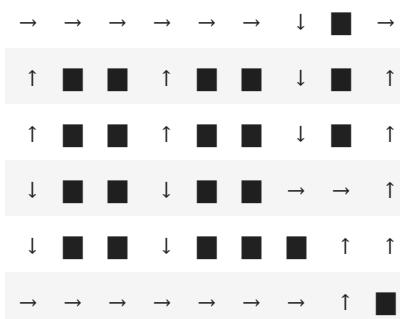
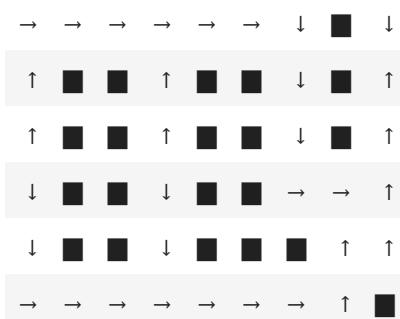
### Improved policy interation 30000

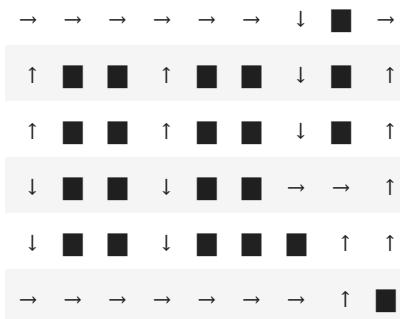


### Improved policy interation 40000

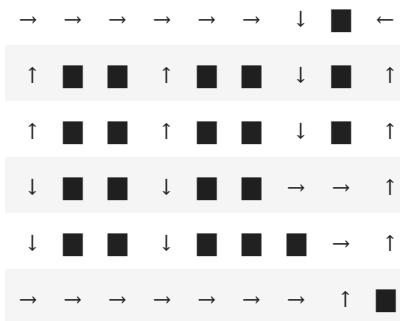
**Improved policy interation 50000****Improved policy interation 60000****Improved policy interation 70000****Improved policy interation 80000**

**Improved policy interation 90000****Improved policy interation 100000****Improved policy interation 110000****Improved policy interation 120000**

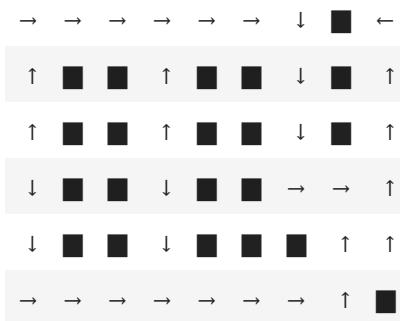
**Improved policy interation 130000****Improved policy interation 140000****Improved policy interation 150000****Improved policy interation 160000**



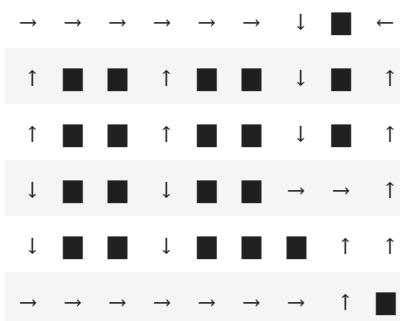
### Improved policy interation 170000



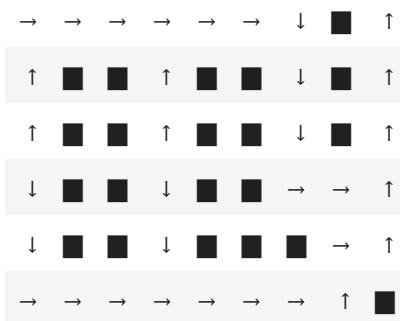
### Improved policy interation 180000



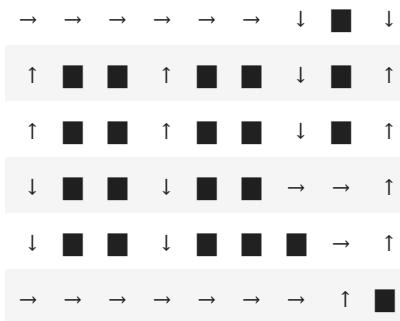
### Improved policy interation 190000



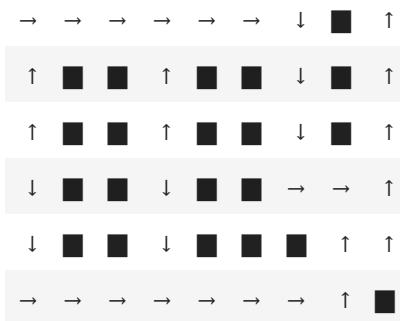
### Improved policy interation 200000



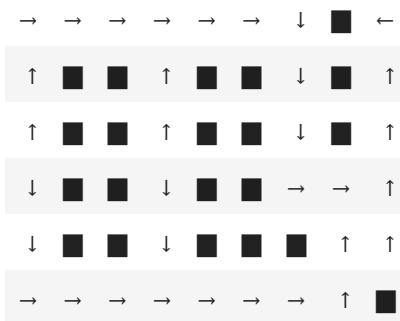
### Improved policy iteration 210000



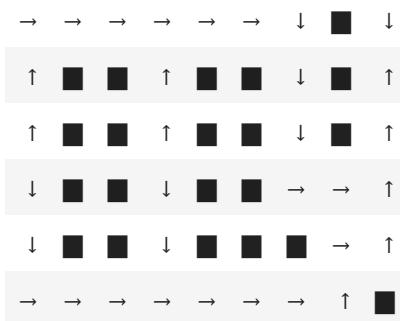
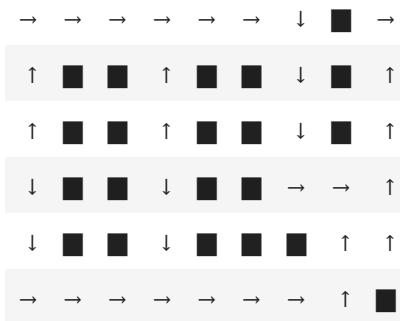
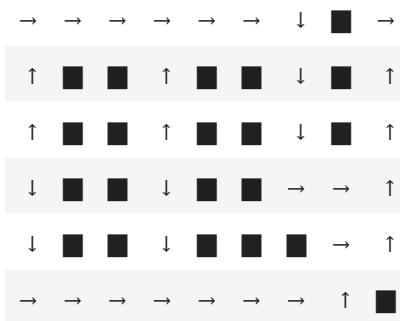
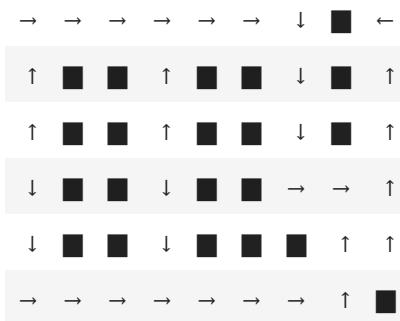
### Improved policy iteration 220000

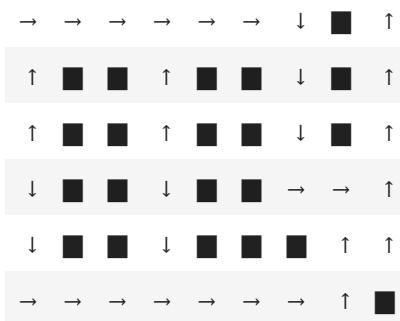
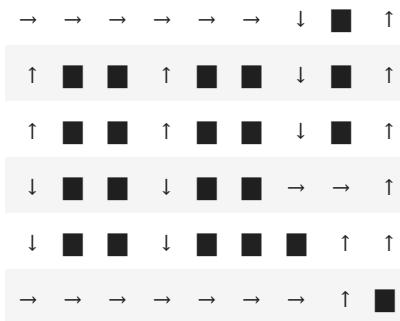
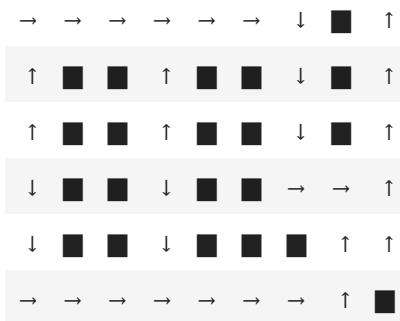
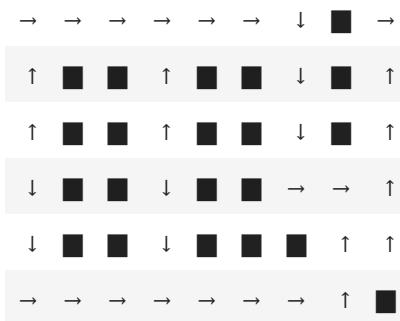


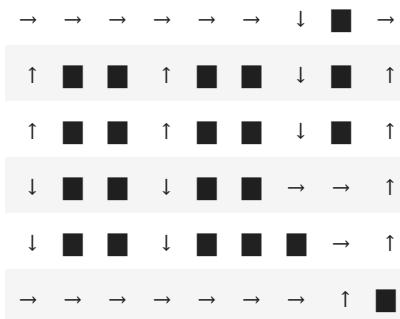
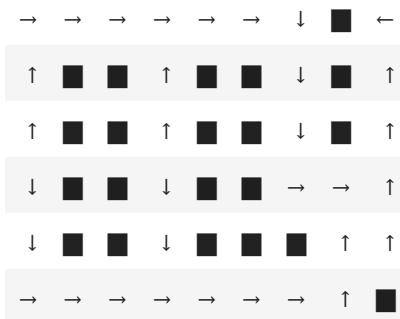
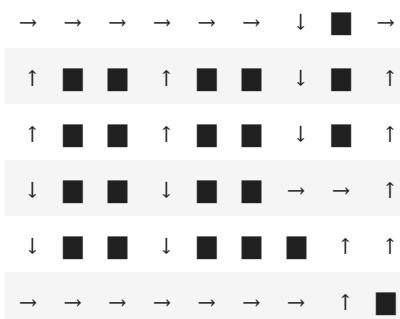
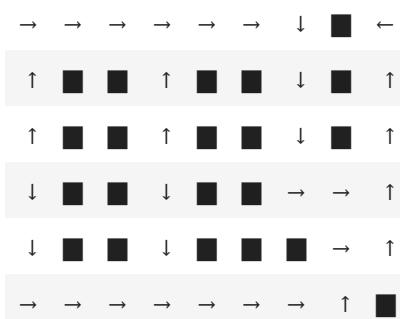
### Improved policy iteration 230000

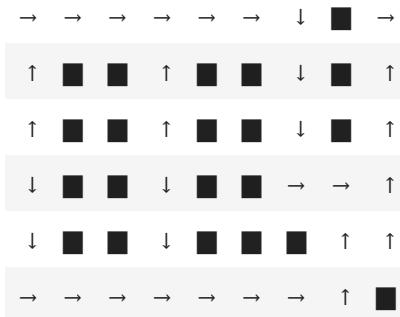
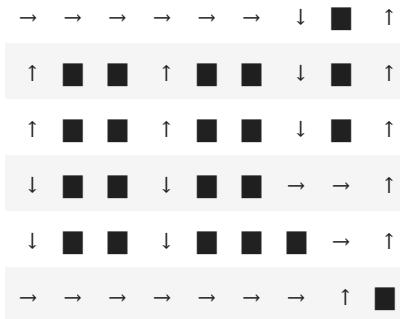
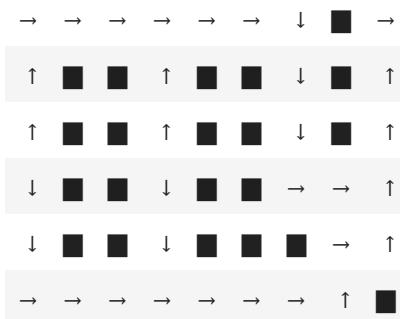
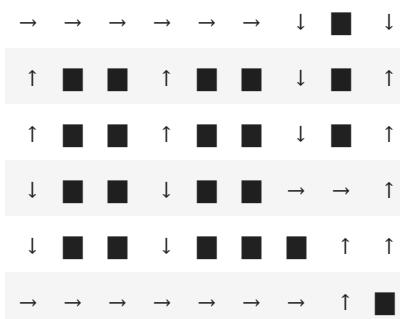


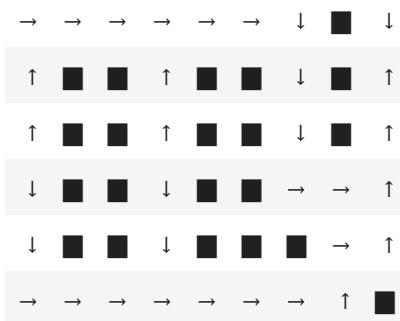
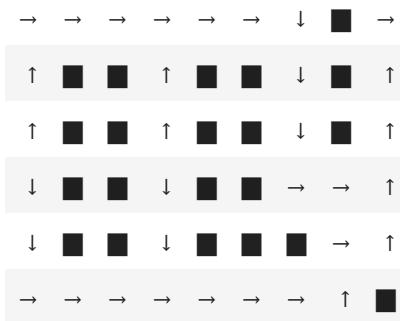
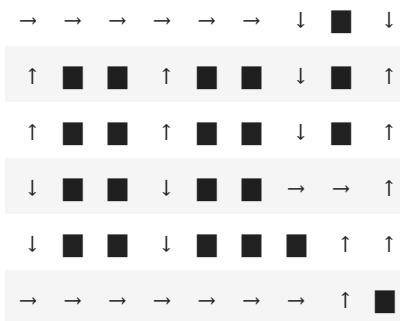
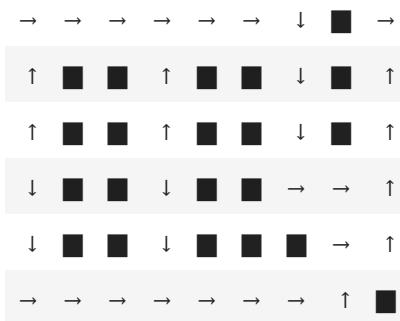
### Improved policy iteration 240000

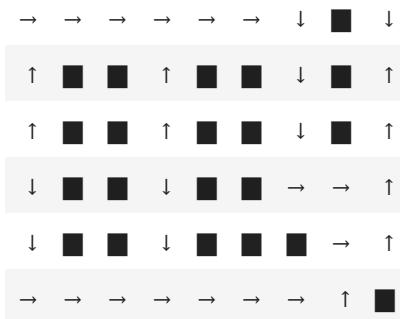
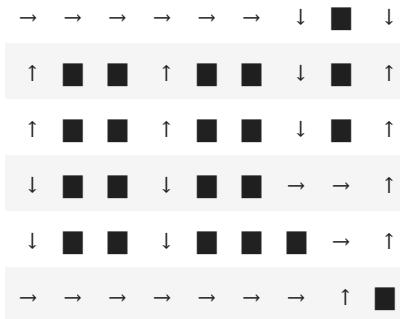
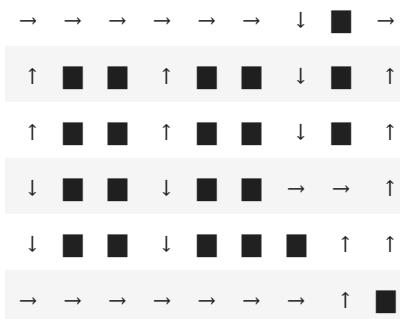
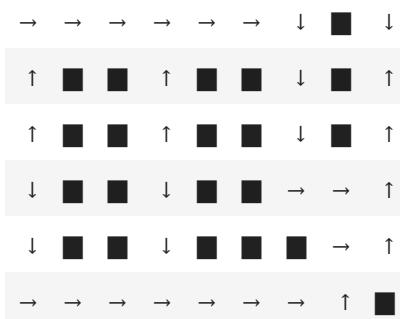
**Improved policy interation 250000****Improved policy interation 260000****Improved policy interation 270000****Improved policy interation 280000**

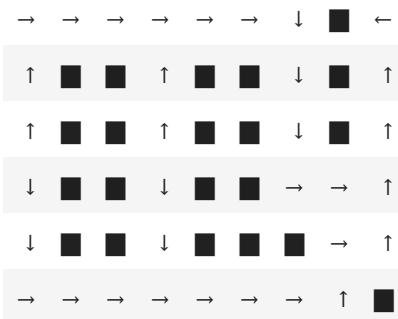
**Improved policy interation 290000****Improved policy interation 300000****Improved policy interation 310000****Improved policy interation 320000**

**Improved policy interation 330000****Improved policy interation 340000****Improved policy interation 350000****Improved policy interation 360000**

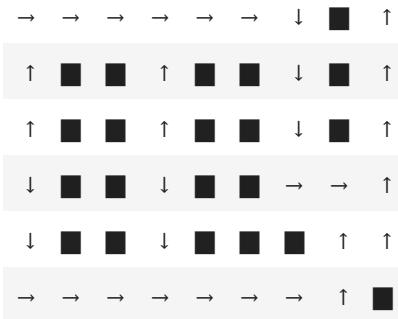
**Improved policy interation 370000****Improved policy interation 380000****Improved policy interation 390000****Improved policy interation 400000**

**Improved policy interation 410000****Improved policy interation 420000****Improved policy interation 430000****Improved policy interation 440000**

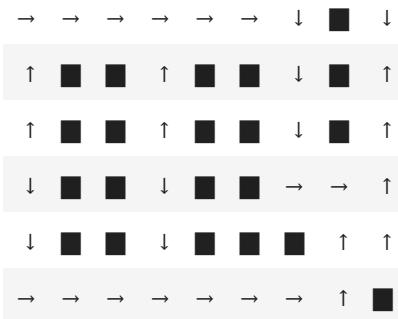
**Improved policy interation 450000****Improved policy interation 460000****Improved policy interation 470000****Improved policy interation 480000**



### Improved policy iteration 490000



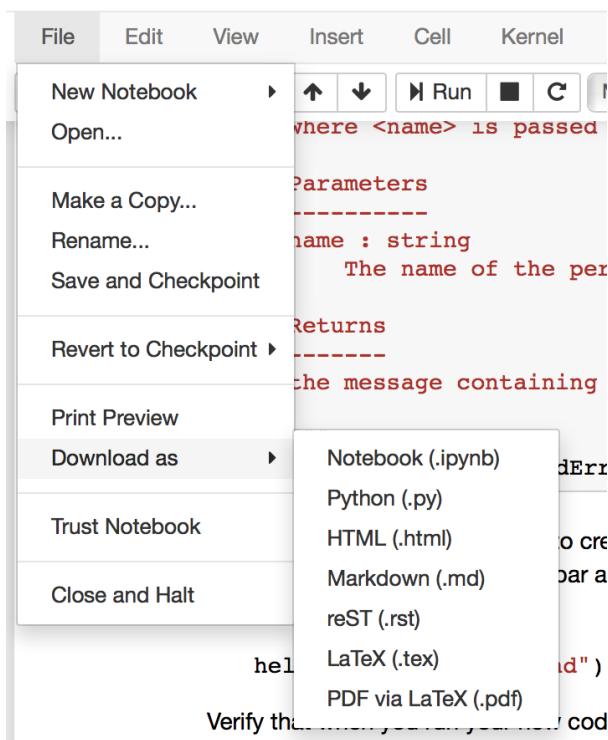
### Improved policy



The results are identical (or nearly) to the ones received in Part A, both being what I see as at least the most optimized policy that can be made given the constraints. However, Part B was notably faster, most likely from not having to do the complete Monte Carlo sims. each time. This model, furthermore, is much closer to human thinking than either Monte Carlo or Dynamic Programming in that it incorporates not knowing the entire environment unlike Dynamic Programming, while also only considering simple next steps in time and not complete simulations like Monte Carlo.

## 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 NYU Classes webpage for the course (available on <https://home.nyu.edu>). **Make sure you complete all parts (A and B) of this homework.**