Reinforcement Learning: A Protocol for Identification

Ryan T. Cullen

In Collaboration with Dr. Sarah Marzen

Pitzer College, Keck Science Institute

Pitzer College, 1050 N Mills Ave, Claremont CA 91711

rcullen@students.pitzer.edu

Abstract

The goal of this project is to attempt to develop a taxonomy for identifying whether or not an agent (or organism) is a reinforcement learner. A reinforcement learner is an agent that seeks to maximize its total reward by altering its action policy. However, simple observation of an agent acting to maximize its potential future reward is not enough to say that it has the ability to learn. The justification for this claim is that the population as a whole could be doing the learning over evolutionary timescales, as opposed to the individual agent learning over its lifetime. This paper makes an attempt at establishing a protocol for distinguishing between these two strategies. Reinforcement learning algorithms such as Policy Iteration and Q-learning are tested as a means of solving/learning to solve MDPs, alongside various detection methods to identify potential learners. Outlier detection proved to be sub-optimal overall under most conditions, but certain environment configurations yielded extremely successful trials. Other methods such as signal detection were explored as a means of improving the protocol. Also discussed is the intriguing fact that an organism's ability to learn had to have itself been learned by the population first.

1 Introduction

Grouping species based on shared characteristics has long been a tool used in biology to gain a broader understanding of the different evolutionary strategies that organisms employ. One characteristic for which attempts to create a taxonomy have been scarce is an organism's individual ability to learn. This is a difficult thing to measure because one thing that is shared by *all* populations is that they learn as a whole to increase their local growth rate over evolutionary timescales.[2]

This begs the question: who is doing the learning, the population or the organism? On one end of the spectrum, the population might be equipped with a reinforcement learning strategy that programs each organism with a static action policy that is optimized through natural selection; on the other end, each organism is a reinforcement learner that evaluates and optimizes its action policy dynamically in response to changing environment conditions. All biological life it would seem can be placed on or between these two limits; oftentimes mixing adaptations programmed by the population with those individually learned over a lifetime. Regardless of which strategy an organism employs, they all are designed to maximize its total reward in *previously seen* environments.

However, what happens when you place each aforementioned type of organism into a *new* environment with different conditions? This paper proposes a protocol for identifying individual reinforcement learners by doing exactly that. Broadly, the method follows the following path. First, we solve for the optimal policy function of an initial environment. This policy is given to a set of agents (some of which are learners, but others are not and will not alter the policy). The

agents are then placed into a *new* environment and allowed to act over many episodes. Theoretically, the agents who are equipped with reinforcement learning algorithms will learn to increase their average reward by altering their action policy in a way that is measurably greater than the reward gained by the agents with static policies. Tactics such as outlier detection and signal detection were explored as possible ways to distinguish between the two. Outlier detection proved to be suboptimal overall, but it has potential for high success rates under certain conditions.

This goal of this paper is to further our understanding of how we can test and measure an agent's ability to learn. Reinforcement learning as a whole is discussed in detail along with its various forms and underlying logic. Heavily utilized was Barto and Sutton's *Reinforcement Learning: An Introduction (2018)*[1] to gain an understanding of the fundamental concepts and algorithms that characterize the field. Several methods for classification are proposed, along with graphical analysis of the results. Also discussed is the intriguing fact that an organism's ability to learn had to have itself been learned by the population first.

2 Background

At the intersection of computer science and neuroscience lies the realm of Reinforcement Learning (RL). Generally, it can be understood as the science of decision making. It is a powerful framework that allows one to approach a problem in terms of cost vs. reward, and can thus be generalized for a variety of applications.

Before continuing onto a precise characterization of RL, and what it means to be a reinforcement learner, it will be useful to first define a few important terms. The paradigm revolves around the construction of an agent that takes actions in an environment, and is then rewarded (positively or negatively) for taking each action based on the current state. Let us better understand this statement by examining each of these terms individually, in reference to a hypothetical scenario.

For this example, let us say that we have a robot that has the ability to walk around a room. Here, the room is our ***environment***, and the robot is our ***agent***. An example of the set of ***actions*** that the robot might be able to take could be "go right", "go left", "go forward", and "go backward". The ***state*** describes the relevant information about a particular section of the environment; each state in this example would describe the characteristics of one subsection of the room, such as its relative position or the presence of an obstacle. Additionally, every state contains a set of scalar values that represents the reward that our agent would receive for taking each action in that state. This number is added to the ***total cumulative reward*** that our agent has received up until that point. Finally, our agent has an ***action policy*** (or ***policy function***) that dictates the action it will take based on the state in which it currently exists.

We can now define precisely a ***reinforcement learning agent*** as an entity that takes actions in an environment, and seeks to maximize its total cumulative reward over time through manipulation of its action policy.[1] It should be noted, however, that while this example characterizes our agent as a physical entity (like an organism), RL agents can take on many abstract forms such as managing databases or stock portfolios.

Statistics and Probability

A brief mathematical aside is necessary for a full understanding of the material ahead. The word ***stochastic*** means "pertaining to chance". In an RL context, an environment can be thought of as stochastic if an agent's actions do not have a fully deterministic relationship with the transition between states. For example, if our robot from the above example was walking on an ice rink, there might be a chance that when it decides to "go forward" that it instead slips and moves to the right. This means that our environment has a certain ***probability*** associated with the transition from one state to another. Probability can be defined as the likelihood of an occurrence, and can be represented mathematically using the following notation:

$$\mathbb{P}[x] = \text{probability of event } x \text{ occurring}$$

However, in an RL context, even when an environment is stochastic it will still have some correlation with the actions taken by the agent. To express this mathematically, we use what is called a ***conditional probability*** which can be represented with the following notation:

$$\mathbb{P}[x \mid y] = \text{probability of event } x \text{ occurring, given } y$$

or

$$\mathbb{P}[x \mid y, z] = \text{probability of event } x \text{ occurring, given } y \text{ and } z$$

Thus, it is a probability that is *conditioned* on the fact that something else has already occurred or been observed. This is necessary in RL because the transition between states is conditioned on the actions taken by the agent.

With this, we can now take a deeper look into the types of problems to which reinforcement learning algorithms can be applied to solve.

What constitutes a reinforcement learning problem?

A reinforcement learning problem takes the form of a set of states (the environment) and an agent. At each timestep *t,* the environment shows the agent the current state and doles out a reward for the previous action taken. The agent then takes a new action, and we repeat the process. Below (Fig. 1) is a simple schematic depicting the aforementioned loop.

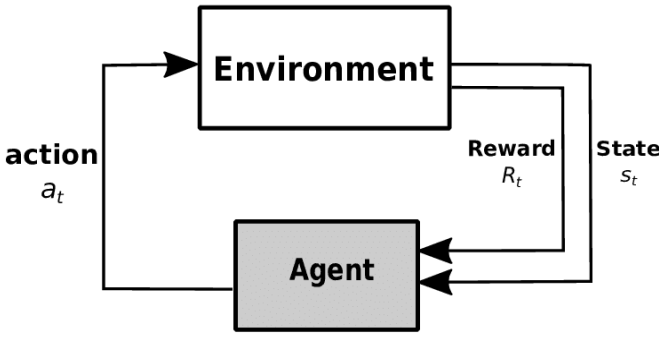


Fig. 1[1]

To understand how we can apply RL to solve a problem we must begin by precisely defining the environments that we wish to model. Reinforcement learning algorithms are typically trained on environments that are characterized by the following five (5) parameters:[9][1]

(1) A finite set of states $s \in \mathcal{S}$ for which each state obeys the *Markov Property*. This property is defined as follows:

Definition
A state s_t is *Markov* if and only if : $$\mathbb{P}[\, s_{t+1} \mid s_t \,] = \mathbb{P}[\, s_{t+1} \mid s_1, \ldots, s_t \,]$$

This means that the conditional probability that our agent will end up in the next state s_{t+1} depends only on our current state s_t and not any other previous state that our agent has visited. In other words, the future is independent of the past given the present.[1]

(2) In each state s_t our agent will be allowed to take an action $a \in \mathcal{A}$ where $\mathcal{A}$ is a finite *action matrix*.

(3) A *state transition probability matrix* $\mathcal{P}$ defines how our agent progresses between states; it is constructed from the following expression corresponding to each action taken at each state:

$$\mathcal{P}(s' \mid s, a) = \mathbb{P}[\, s_{t+1} = s' \mid s_t = s, \mathcal{A}_t = a].$$

In words, this is the conditional probability that our agent will end up in state s' if it is currently in state s and takes action a.

(4) A *reward matrix* $\mathcal{R}(s, a)$ determines the reward r_t that our agent receives if it is currently in state s and takes action a.

(5) A *discount factor*, γ, is used to limit the value of future rewards. The *return,* G_t*,* describes the total discounted reward from timestep t and can be expressed in the following way:

$$G_t = \mathcal{R}_{t+1} + \gamma \mathcal{R}_{t+2} + \ldots = \sum_{k=0}^{\infty} {}^{k}_{t+k+1}$$

Note: These expressions can be easily generalized to stochastic reward functions

If all of these five parameters are met , then this environment satisfies the conditions for what is called a Markov Decision Process (MDP). Thus, an MDP can be expressed as the following tuple:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$$

This is useful because almost all reinforcement learning problems can be formalised as MDPs. Thus, if we are given an environment and we formulate a way to accurately model it in terms of $\mathcal{M}$, then we can apply any of our RL algorithms to *solve* the MDP. Solving an environment amounts to calculating the optimal action policy, denoted $\boldsymbol{\pi}$, that will cause our agent to act in a

way that will maximize its total reward. This is what evolution does,[6] and this is what we humans learn to do over our lifetimes.[7] In fact, there is significant evidence that the dopamine system in the human brain acts similarly to an algorithm that was implemented and tested for this project, called Temporal Difference (TD) learning.[8] A coded implementation of this algorithm for the purposes of this project can be found in Appendix E.

How does an agent solve an MDP?

There are a multitude of methods for solving MDPs. The problem can be approached from many different angles, which gives rise to a variety of strategies. However, there is one commonality across RL algorithms in that they *all* utilize some form of the Bellman Equation. The Bellman Equation is a formula for calculating how much potential reward that an agent might be able to acquire in the future. In other words, it is a way for the algorithm to gauge "how good it is" to be in the current state. It exploits the sequential nature of MDPs by dividing the environment into a series of simpler subproblems. A derivation of the bellman equation for a particular policy $\boldsymbol{\pi}$ can be found in Appendix A.

Some algorithms assume that the agent has prior information regarding the MDP model, such as the transition probability matrix and the reward matrix. Therefore, they can be used by the agent to find the optimal policy before interacting with the environment. This is called *offline* learning. This is in contrast to *online* learning, in which the agent learns directly from experience (by taking actions in the environment) while the algorithm actively updates the policy function depending on the rewards that the agent receives.

Offline algorithms are generally more computationally efficient than those that employ online learning, simply due to having access to more initial information. However, these algorithms are not good representations of individual biological learners because they do not learn through direct experience, and they have a sort of omniscience about their environment prior to planning an action policy. By contrast, online methods are much more reminiscent of an organism that is actively learning over its lifetime through interaction with its environment. Thus, we used online algorithms to represent the learners that we were looking to identify, while offline algorithms were used to efficiently solve for an optimal policy when needed. A control group was designed with a static, pre-programmed action policy.

The offline algorithms that I implemented in this project are Policy Iteration and Value iteration, and the online methods include Q-learning and TD-learning. Policy Iteration and Q-learning proved to be the most performant in their respective categories, and were thus chosen for acquiring experimental data. The logical explanations behind all of the aforementioned algorithms, as well as my coded implementations of each one, can be found in the Appendix.

How can we identify a learner?

All organisms act in a way so as to maximize their reward in previously seen environments. However, due to the presence of evolution, the population as a whole could be considered a reinforcement learner rather than the individual. Take bacteria, for example. While you could not teach a single bacterium to associate a behavior with a reward, as you can do with dogs, both dogs and bacteria over many lifetimes learn to avoid the hazards present in their natural habitats; i.e. both species have some form of pre-programmed action policy that is optimized for their usual environment.

As such, it follows that simply observing an organism in its natural habitat acting to maximize its reward is not enough to say that it is learning on an individual level. In order to identify the dog as a reinforcement learner we would have to place both organisms into *new* habitats that they have never seen before. In these new conditions, the reinforcement learner would evaluate and update its action policy based on new observations, while the organism with a static policy would act in the same way as it did before the switch. Theoretically, this would lead to an average reward gained by the learner that is measurably greater than that of the non learner.

The question then becomes, how can we design a framework to identify when an agent's average reward increases due to reinforcement learning, versus when it increases randomly due to noise? Tactics that we explored to tackle this objective are ***outlier detection*** and ***signal detection***.[3] These methods stem from Detection Theory, a field of study revolving around designing ways to differentiate between information-bearing patterns.

Detection Methods

1. Outlier Detection Method:
    a. First, an MDP is solved for an optimal policy using Policy Iteration. This policy is given to a set of learners (those who will change the policy over time) and non learners (those who will not).
    b. The learners are equipped with a Q-learning algorithm with a learning rate of 1. Then, both sets of agents are placed into a new environment with different conditions and are allowed to act over many episodes.
    c. The average reward gained by each agent is then calculated.

d. Subsequently, the reward density functions (how many of each agent received close to the same reward, within some small range) for each of the two sets of agents are graphically overlayed onto one another.

e. A threshold value for the reward is chosen and it is declared that all agents who received an average reward that is above the threshold will be classified as reinforcement learners.

The logic behind this method is that there should be a measurable difference in the average reward between any given learner and non-learner in the new environment. However, due to noise, this is not always the case and there will often be some overlap between the two density functions. The following (Fig. 2) is an example of the result of this process.

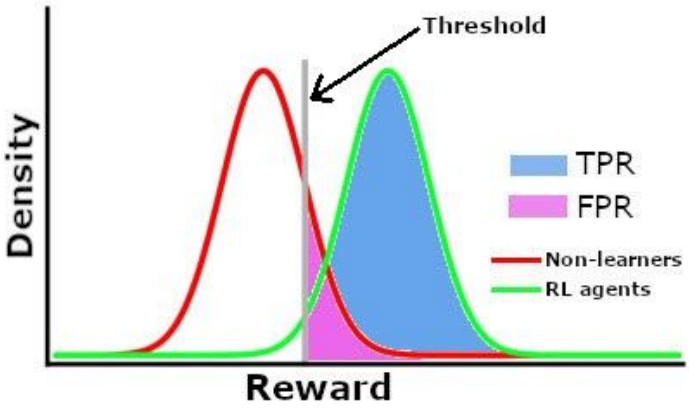


Fig. 2

2. Signal Detection Method: This method is based on the concept of a ***hypothesis test***.[3] The solution to a hypothesis test is in terms of a ***decision rule***.

a. In this framework, each of the possible outcomes corresponds to a hypothesis. We denote the set of N possible hypotheses as

$$H = \{H_0, H_1, \ldots, H_{N-1}\}$$

b. For each of the possible hypotheses, different data is observed, and this is what we will use to distinguish between them, denoted by the k-dimensional vector:

$$y = \{y_0, ..., y_k\}, \quad y_k = s_k + w_k$$

Here, s is our signal that we wish to detect, and w represents noise.

c. We associate *a priori* probabilities for each hypothesis

$$P_N = \mathbb{P}[\, H = H_N \,]$$

d. It is required that we have a well-defined objective function corresponding to a measure of performance. For this we use an objective function taking the form of an expected cost function. Specifically, we use

$$\tilde{C}(H_j, H_i) = C_{ij}$$

To denote the "cost" deciding that the hypothesis is H_i when the correct hypothesis is H_j. We can customize this function based on the particular parameters of the problem.

e. An average cost function (also called the *Bayes Risk*) is defined by:

$$J(f) = E[\tilde{C}(H, f(y))]$$

f. Finally, an optimal decision rule is calculated by minimizing J.

Time constraints prohibited the implementation and testing of this method. However, for future research we believe that this approach could yield better results than outlier detection.

Analyzing our detection methods

After we applied the above methods, the next step was to analyze and measure how successful each approach was at correctly classifying reinforcement learners. To do this, we utilized what is called a ***receiver operating characteristic*** (ROC) curve.

ROC Curves

An ROC curve is a graph which plots the ***true positive rate*** (TPR) on the y-axis against the ***false positive rate*** (FPR) on the x-axis. TPR refers to the number of times that our detection method correctly labeled a reinforcement learner. FPR refers to the number of times that our detection method incorrectly labeled a non-learner as a reinforcement learner. In the case of outlier detection, the way we acquired a spectrum of values for the above quantities is by sweeping the threshold value over the set of average rewards gained by the agents, from zero to the maximum averaged reward gained by a single agent. At each iteration of the sweep, we say that all agents who accumulated an average reward that is above the threshold are RL agents, and those below the threshold are non-learners. Finally, we check our classifications against the actual set of agents to see how well our method performed.

The ideal case here is that the Area Under the Curve (AUC) of the ROC graphs is equal to 1. This would indicate that, at a certain threshold, our TPR is 1 (100%) and our FPR is 0 (0%). In other words, AUC = 1 indicates that there is a value threshold for which ALL agents that received an average reward above that threshold are indeed RL agents, and ALL agents that received an average reward below the threshold are non-learners. Let us consider the following (Fig. 3) example:

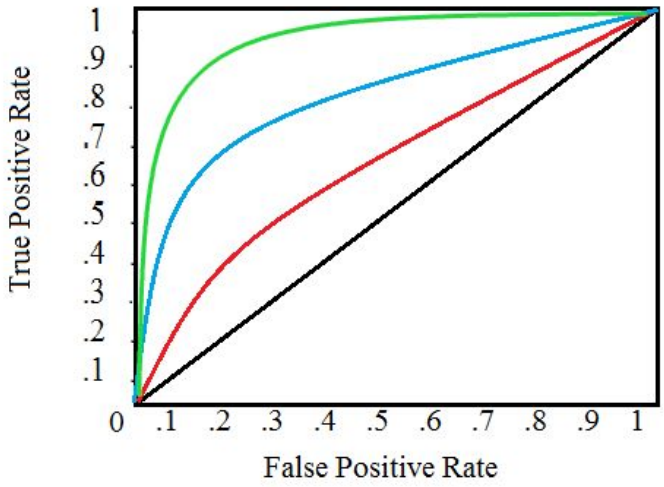


Fig. 3 *https://a8h2w5y7.rocketcdn.me/wp-content/uploads/2016/08/ROC-curve.png*

Let us say that the red, blue, and green lines correspond to, Detection Method A, Detection Method B and Detection Method C, respectively. From this graph, we can readily see that the AUC of Detection Method C is *greater* than the AUC of A and B. Thus, we can conclude that Detection Method C is the most successful at correctly distinguishing data from background noise. This is due to the fact that for any given point on the green line (corresponding to a threshold), our FPR is large and our TPR is small with respect to the other two curves.

3 Results

The environments on which we tested our agents come from OpenAI's Gym toolkit[5], a package for designing and testing RL algorithms in python. It comes with many built in environments such as Atari games, Physics simulations, and command-line text representations. The particular environment for which results are shown below is a modified version of a text-based environment called "Frozen Lake".

This environment consists of a 2-D grid of states. Every state has random reward values for taking each possible action, ranging from -10 to 10. There are a random number of terminal states (holes in the ice) that, if entered, will cause the episode to end. There are four actions that our agents can take: move up[↑], move down[↓], move left[←], and move right[→]. A representation of what a 5x5 version of the environment might look like is pictured below (Fig. 4). Here, the H's represent holes (terminal states) and the F's represent frozen sections of the lake (actionable states).

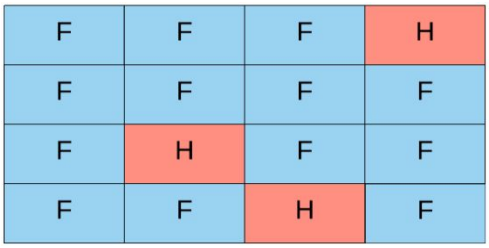


Fig. 4

For every trial, the initial environment is generated with the parameters outlined above, using the MDPparams() method shown in the *GymRL.py* script in Appendix F. Using the SolveMDP() method from Appendix B, this initial environment is solved for an optimal policy using Policy Iteration. It has been proven that, for a finite MDP, this method will *always* converge to an optimal policy.[1] Then, a new environment is again generated using MDPparams(), in which we allow our two sets of agents to act and acquire our data.

For the following section, State Space (SS) refers to the number of states in the environment. For the above example, SS = 25. The agents were tested on Frozen Lakes with SS ranging from 16 to 2500.

Outlier Detection Results

To display the results, the resulting reward probability distributions for the two sets of agents are overlaid onto one another for many separate trials, with varying values of SS. The bars are reward ranges, and thus the area of each bar describes the probability that an agent of that group would receive an average reward that is within the width of the bar. I have superimposed approximate normalized probability distributions over the bar graphs. Accompanying the distributions are the corresponding ROC curves for each trial, placed to the right of the respective distribution.

Note: the following data (Fig. 5) was collected using the original configuration of the environment. Data was *also* collected using an alternate configuration (Fig. 6), in which the scattered terminal states were removed.

SS = 16

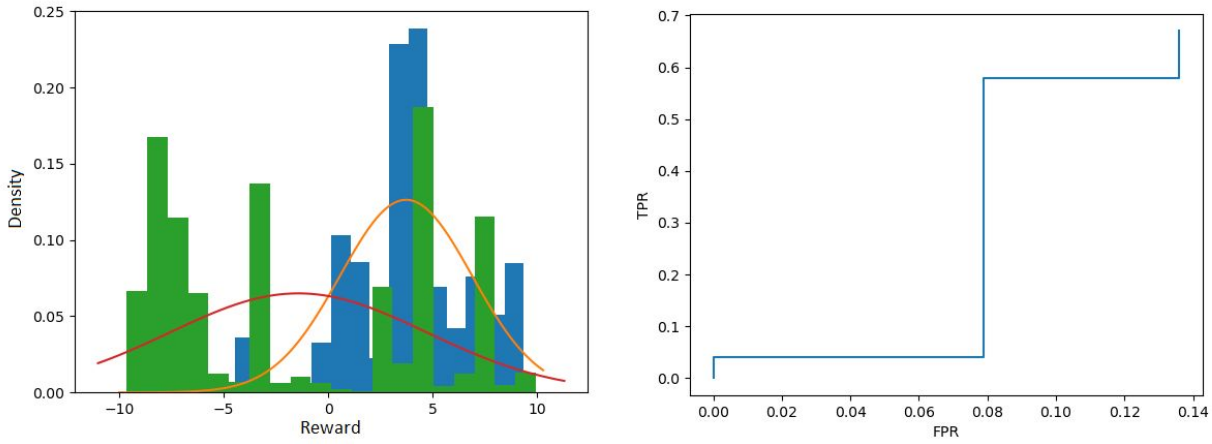
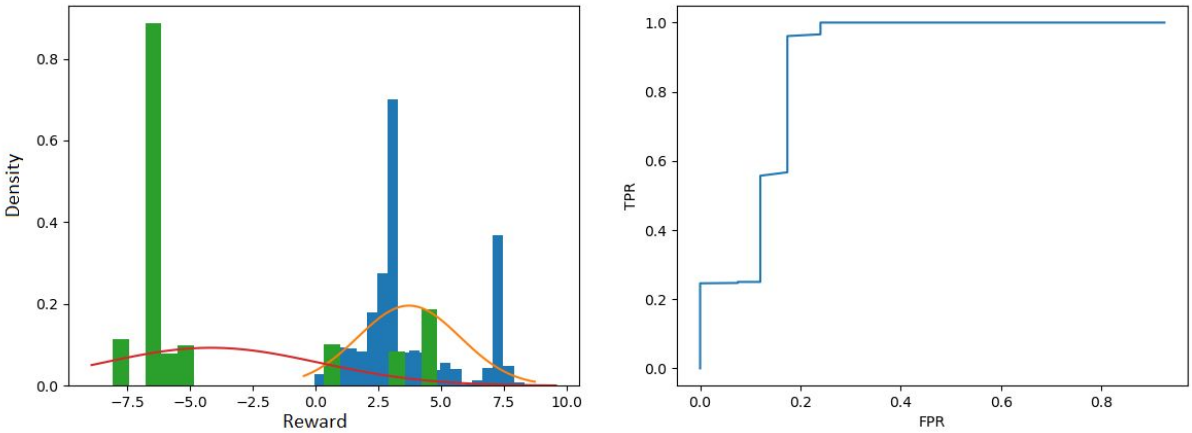


Fig. 5 (cont. on next page)



SS = 256

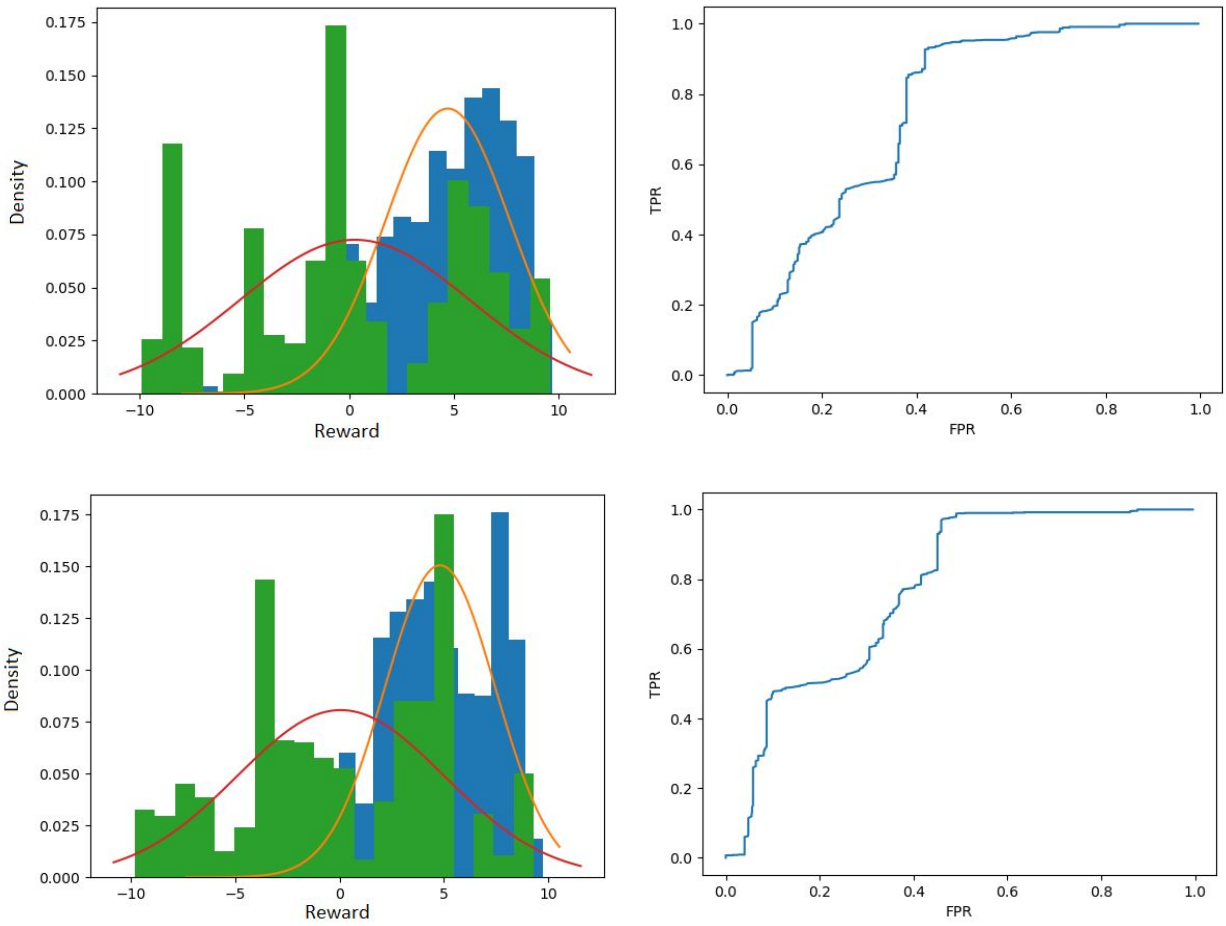


Fig. 5 (cont. on next page)

SS = 400

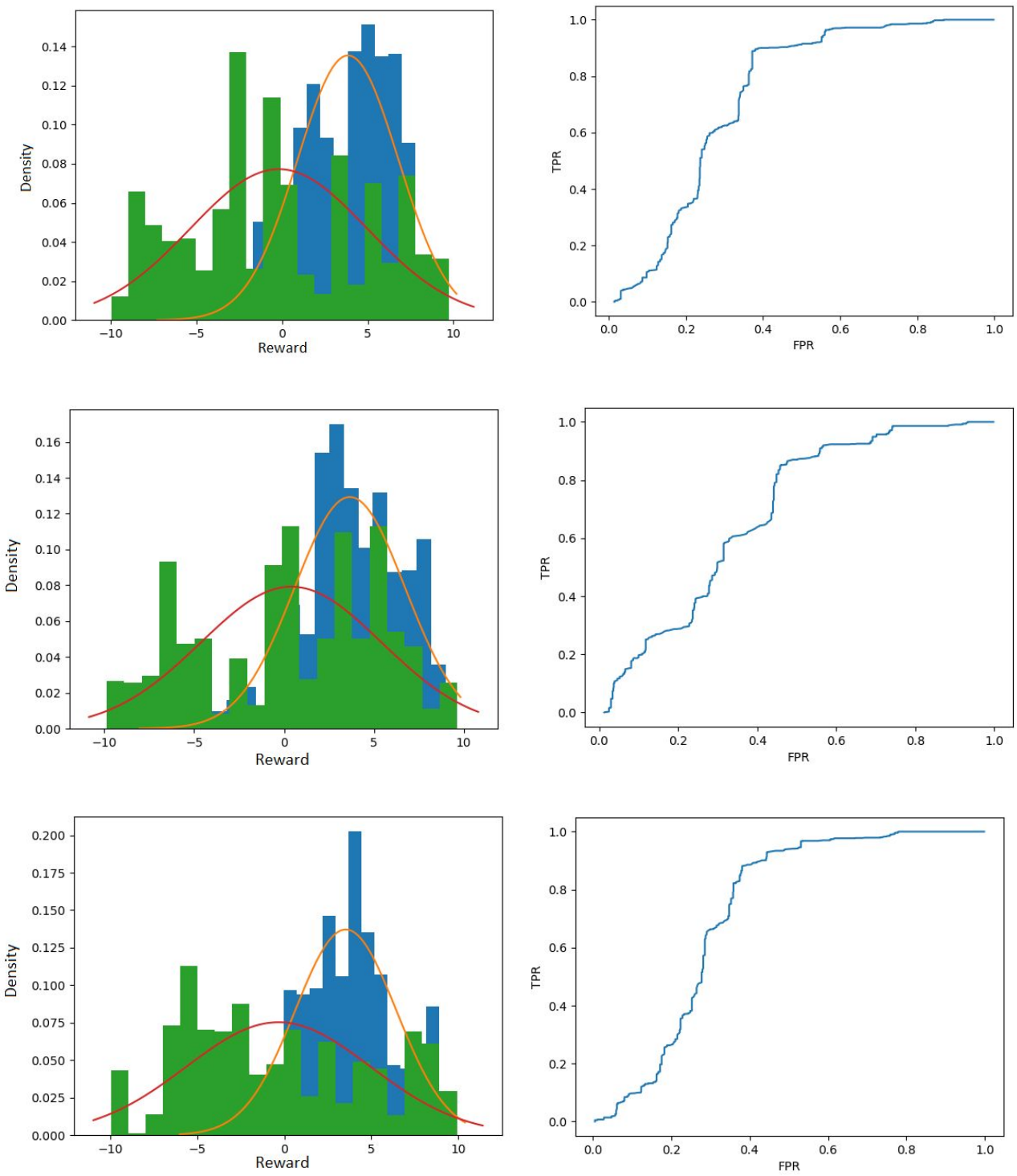


Fig. 5 (cont. on next page)

SS = 900

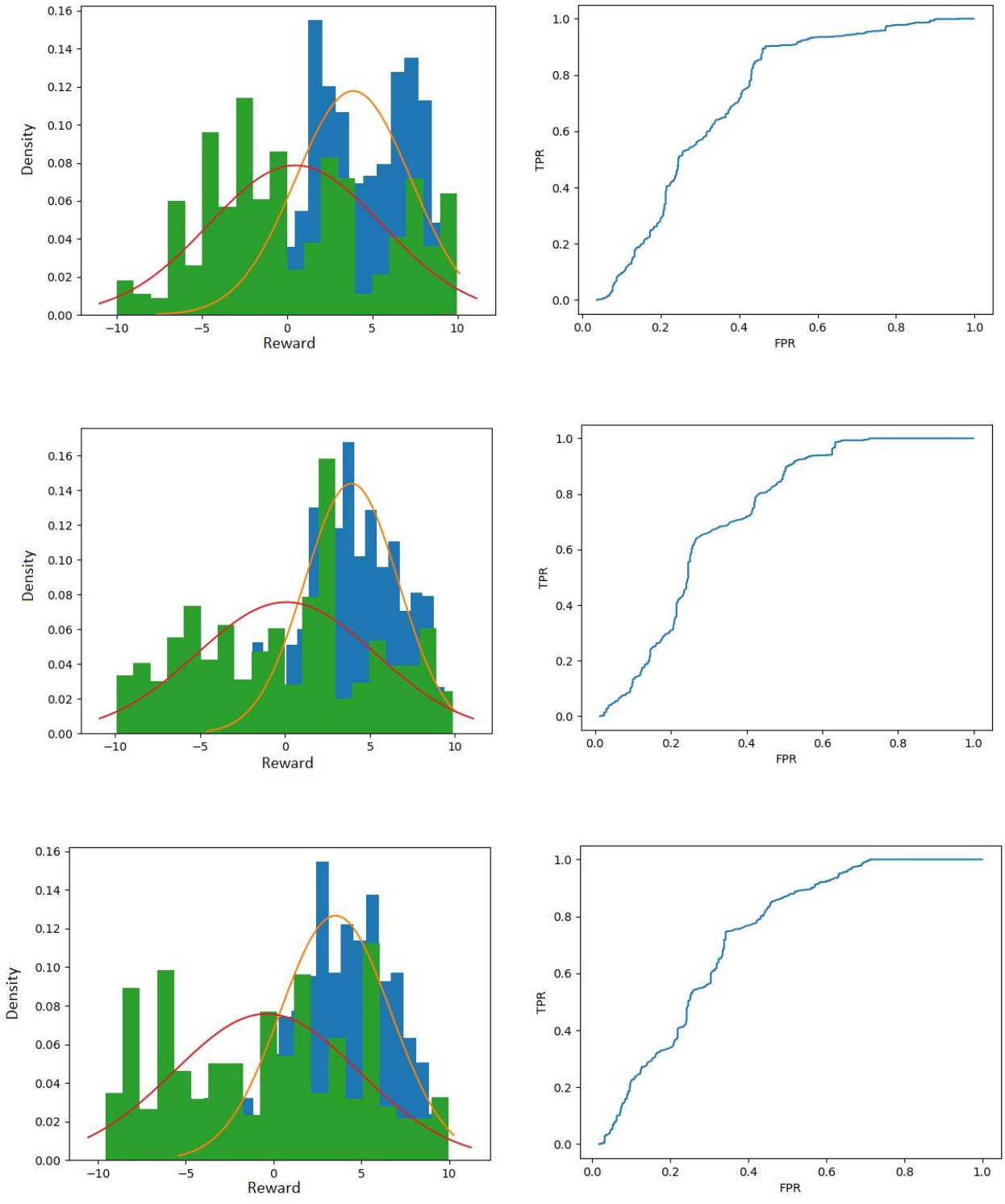


Fig. 5 (cont. on next page)

SS = 2500

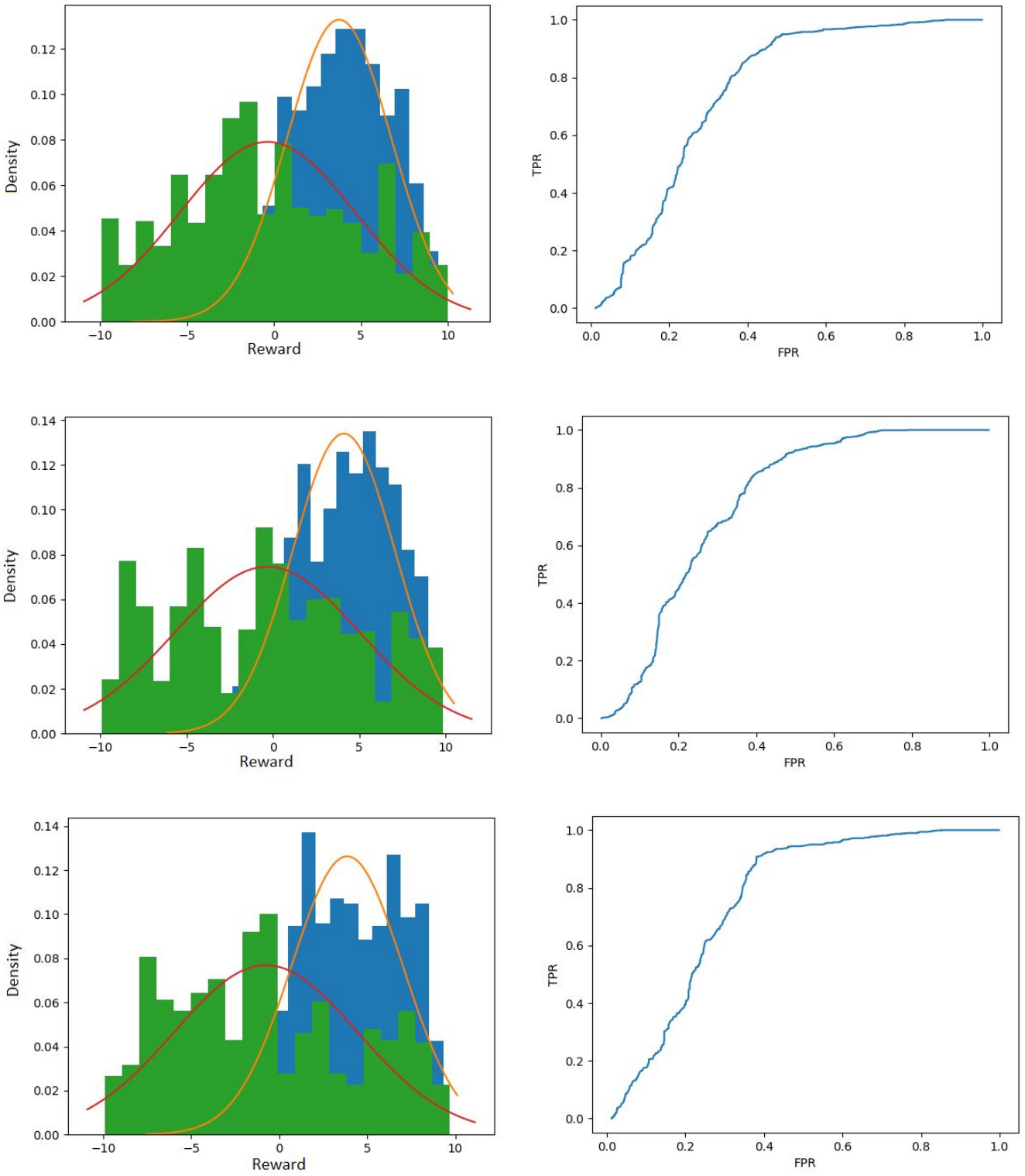

Density
Reward
TPR
FPR
Density
Reward
TPR
FPR
Density
Reward
TPR
FPR

Fig. 5

The next set of data (Fig. 6) was collected using the alternate configuration. All of the "holes in the ice" (i.e. the scattered terminal states) were removed, and thus all states are actionable. Probability distributions and the corresponding ROC curves are displayed in the same manner as above. At low values of SS, the data is extremely volatile and the results from the two configurations are indistinguishable. Thus, only graphs for SS = 256 and above are shown for this set.

SS = 256

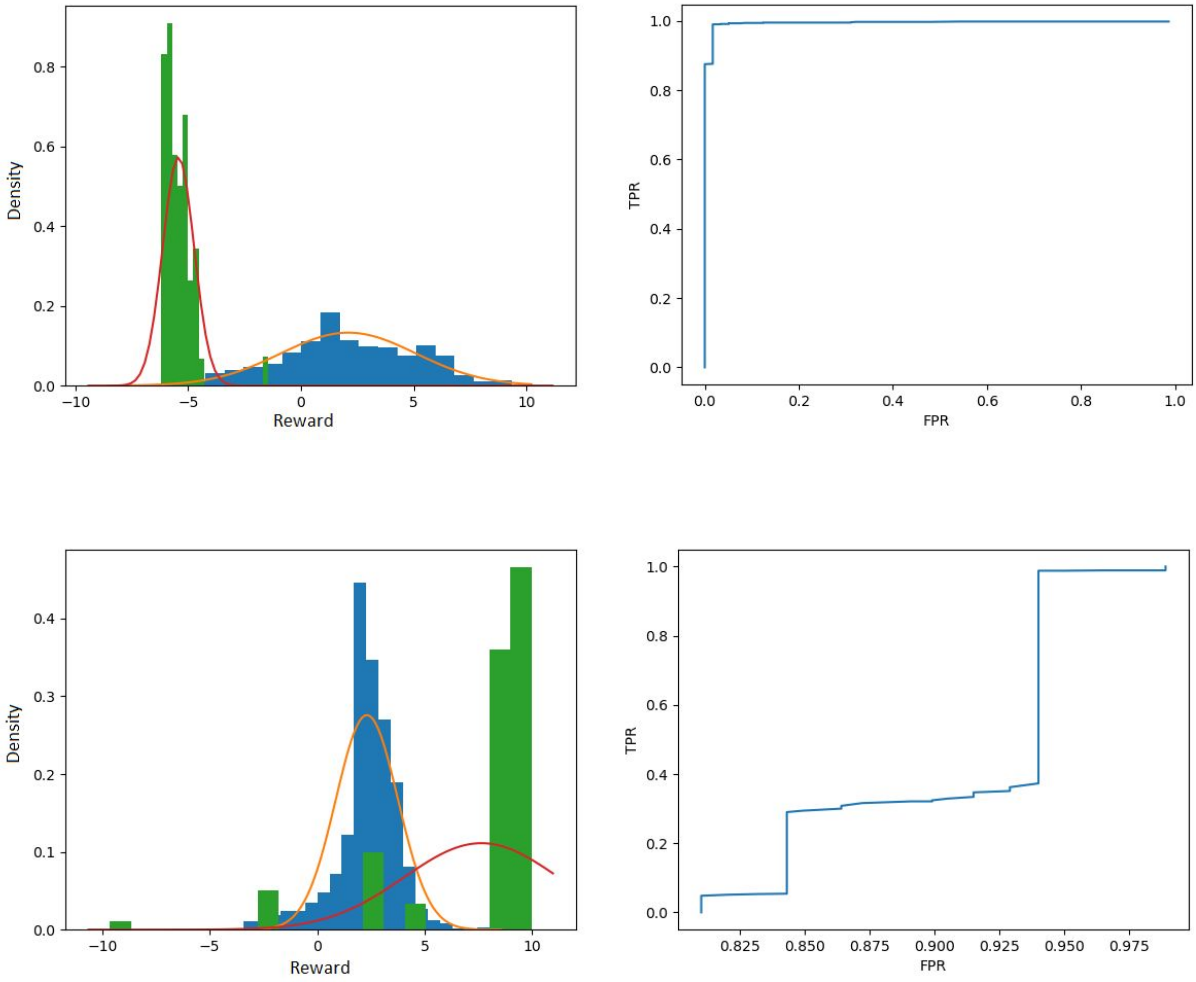


Fig. 6 (cont. on next page)

SS = 400

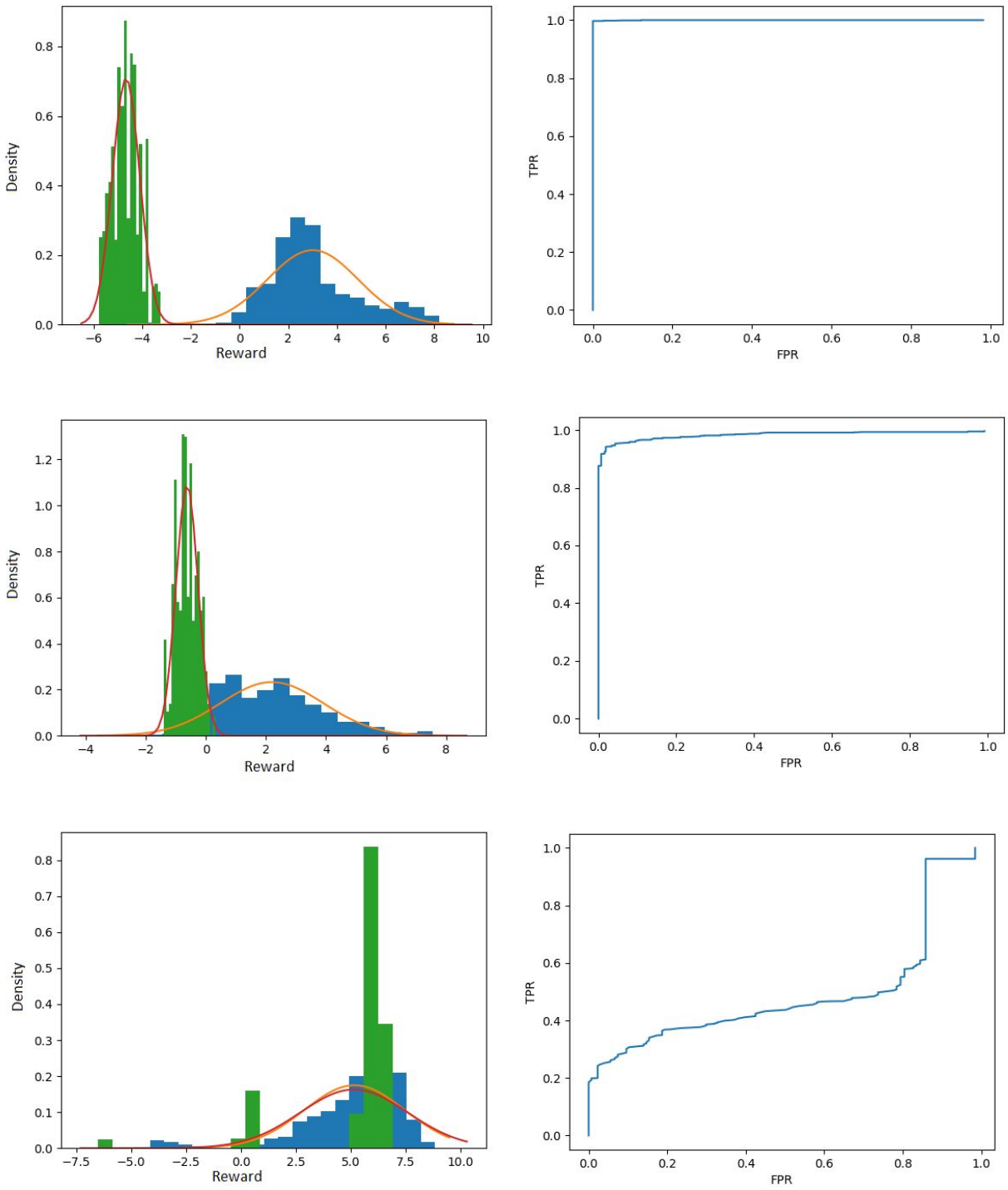


Fig. 6 (cont. on next page)

SS = 900

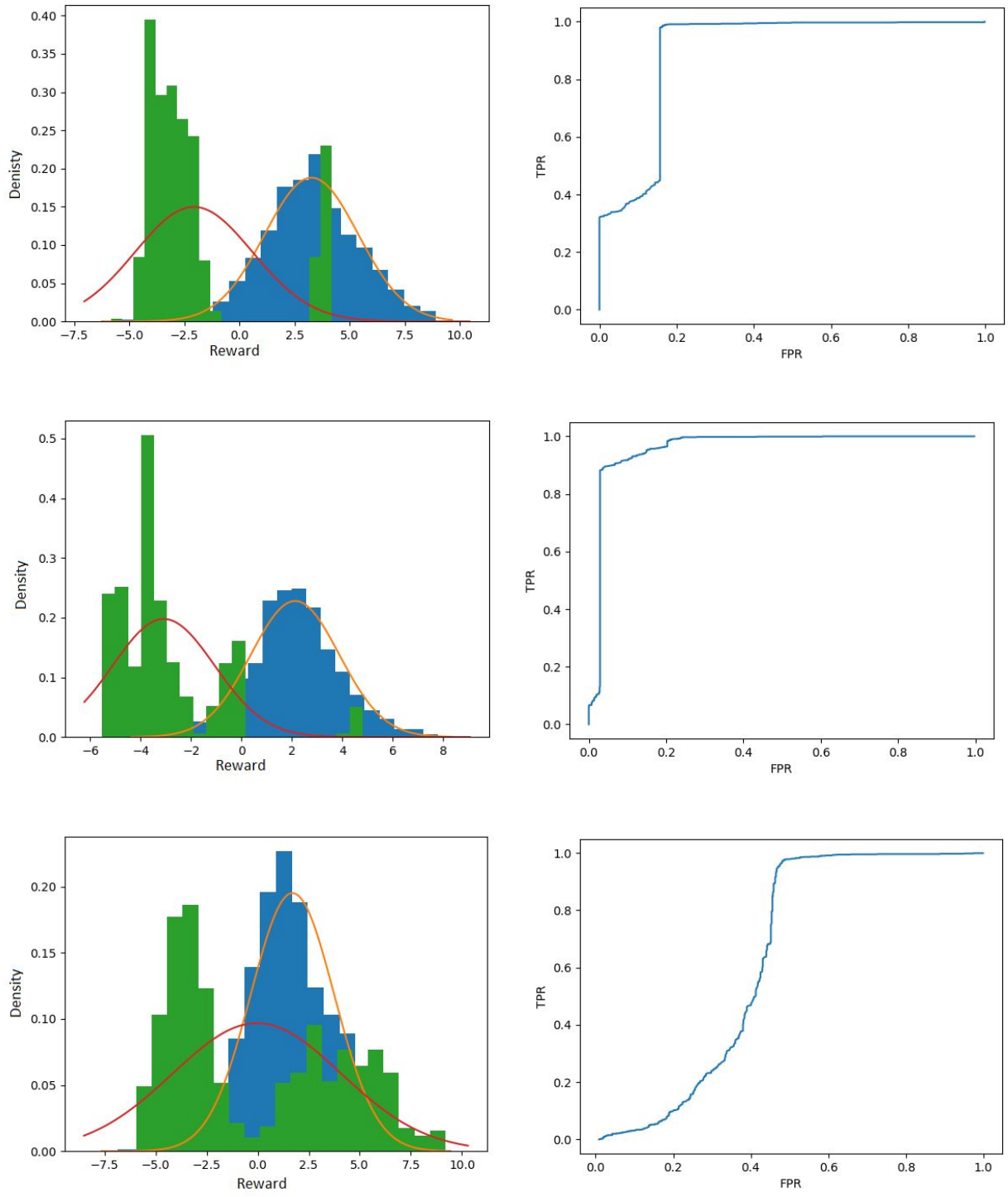


Fig. 6 (cont. on next page)

SS = 2500

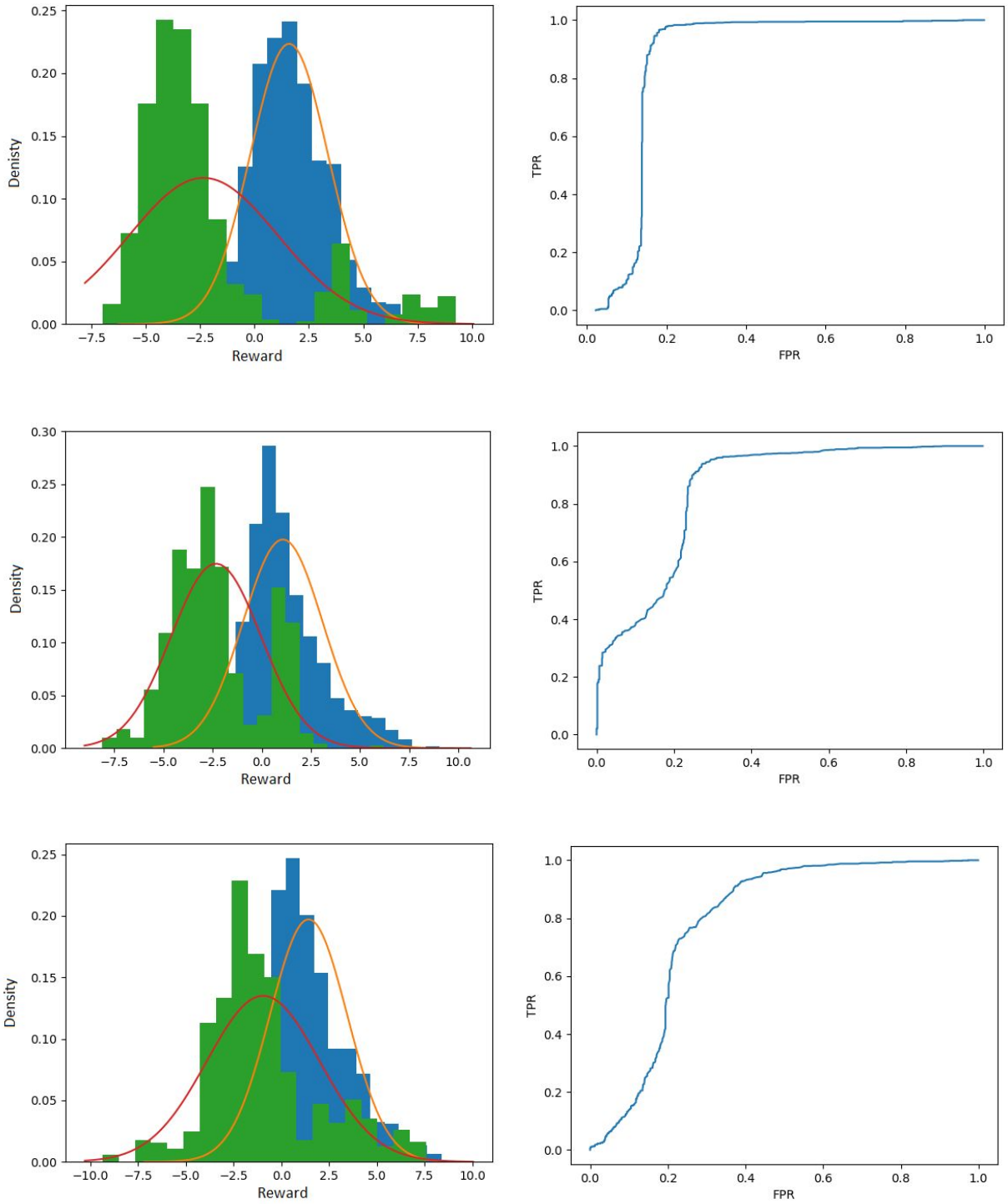

Denisty
Reward
TPR
FPR
Density
Reward
TPR
FPR
Density
Reward
TPR
FPR

Fig. 6

To gain insight into the consistency of outlier detection as a way of identifying RL agents, the next step was to plot AUC vs. SS for each of the different configurations. The results of this are shown below, with error bars describing the standard deviation for each data point.

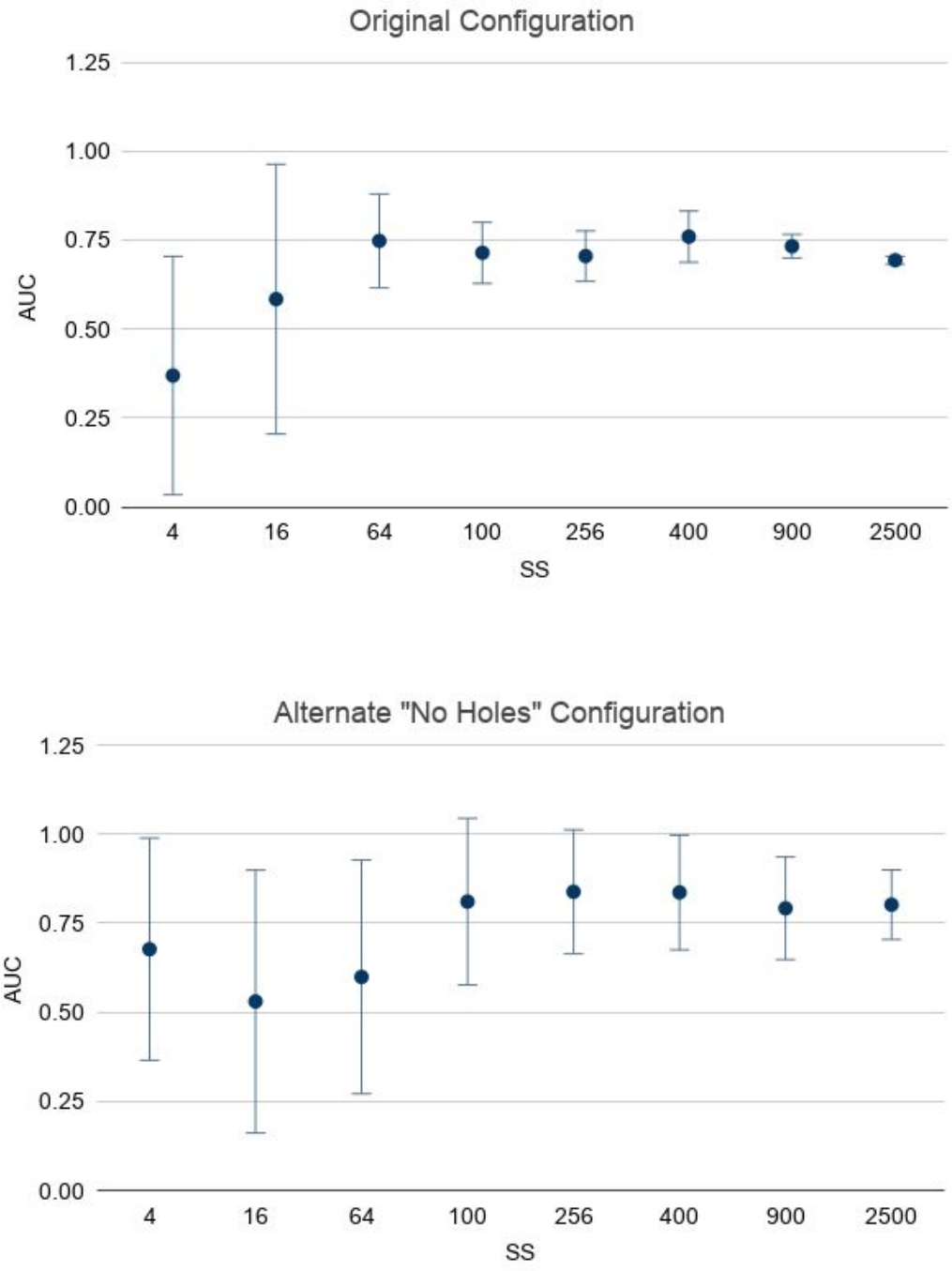


Fig. 7

4 Discussion

The results for outlier detection have proved to be largely suboptimal under the original configuration. However, under the altercate conditions there is potential for extremely high performance in correctly identifying learners; unfortunately, this is achieved at the cost of consistency over multiple trials.

Original Configuration

Let us first consider the original configuration, in which there are terminal states scattered throughout the environment. As shown in **Figure 5**, the distribution for the non learners is somewhat randomly spread out across the range of rewards, regardless of the magnitude of SS. However, as SS increases it is evident that it looks more and more like a normal distribution. On the other hand, the reward distribution for the RL agents remains fairly consistent, and the peak of the average reward hovers around 3 to 5 for just about every single trial. So while the results are not optimal, this method yields very consistent results in terms of its success rate in correctly identifying learners, *especially* for higher values of SS.

This fact is described in the ROC curves that accompany the distributions. Evidently, the curves are of similar shape when trials for the same value of SS are considered. The exceptions are for trials with very low values of SS. Intuition for this could come from the fact that when there are fewer states, there are fewer possible policies; thus there is a higher chance for the non learners to “accidentally” have a really good (or really bad) policy for the new environment. This causes high volatility in the performance of outlier detection. However, it can be seen that the ROC curves for SS = 2500 are very consistent in both shape and AUC.

This is further proven through the SS vs AUC graph in **Figure 7**. The error bars on the data points show a decreasing standard deviation for AUC (across 5 trials) as SS increases. At SS = 2500, the standard deviation is less than 0.02. However, the AUC only averages around 0.73-0.75 for high magnitudes of SS; far from our desired value of 1. Thus, it can be concluded that, while precise and consistent for higher magnitudes of SS, this method is overall inaccurate in correctly labeling the agents under this environment configuration.

Alternate "No Holes" Configuration

The results for the alternate configuration in which the scattered terminal states were removed are somewhat surprising. As shown in **Figure 6**, the distribution of rewards for the non-learners (often, not always) becomes much steeper under these conditions. Intuition for this could come from the fact that under the original configuration, the episodes are cut short by the scattered terminal states. This in turn limits the amount of actions that the agents can take, which widens the possible distribution of rewards. For example, a non learner might "accidentally" acquire a lot of reward in its first two or three actions in the new environment, before reaching a terminal state and ending the episode. If all agents are allowed to take a similar number of actions, as is the case with this alternate configuration, then the average reward of each agent will converge to a tighter range of values. Again, this is portrayed by the steep peaks of the distributions for the non-learners.

The tight reward range for the non learners has a couple important consequences. **Figure 6** shows a much smaller overlap in the reward distributions for the two sets of agents, which means distinguishing between the two becomes much easier. Much of the time, this leads to an *extremely* successful performance by outlier detection in correctly labeling the agents. This is

shown in **Figure 6**, where many ROC curves have AUC values that are much closer to 1 than those from the original configuration.

However, because the ranges are tighter, there is a possibility that the non-learners can outperform most of the RL agents if their policy is coincidentally well optimized for the new environment. This is generally unlikely and was only observed for around 1 out of every 5-10 trials, but when it does happen it causes the performance of outlier detection to drop *drastically*. As shown in **Figure 6**, there are a few ROC curves with dismal AUC values, sometimes even lower than 0.5, which means that the non-learners did unusually well and ruined any chances of correctly labeling the agents.

These facts are further illustrated by the AUC vs SS graph in **Figure 7**. The average AUC values for the alternate configuration are higher for large values of SS as opposed to the original configuration. But they would be even higher, much closer to our target of 1, if there were no outlying data points that severely pull down the averages caused by the random occurrences of non learners performing really well. This is described by the significant error bars for each data point, showing that there is a significant standard deviation even for higher values of SS. Thus, it can be concluded that outlier detection has the potential to perform extremely well under this new configuration, but at a significant cost of consistency. In other words, the method is more accurate under these conditions, but less precise.

One conclusion that results seem to point to is that one must carefully design their detection method based on the conditions of the MPD. It may be the case that there does not exist a singular best method of identifying learners across a broad range of environments.

"Meta Learning"

A final thought that arose in my ventures came from the relationship between individual versus evolutionary learning. It seems to be evident, that, if the individual organisms of a population are equipped with the ability to learn over a single lifetime, then this ability must have *first* been learned by the population itself through evolution. This would seem to be a sort of "meta learning", or "learning to learn".

How this might be implemented practically is a very complicated question. But it is certainly an intriguing prospect that we might be able to design a population of agents that as a whole can learn to equip each individual agent with an RL algorithm. Not only would this allow us to more accurately model the progression of learning ability in a population, but it seems to me that this could be a way of generating completely new algorithms. The population could learn to develop a strategy for each individual that humans haven't considered yet. It is my belief that a large application of AI (artificial intelligence), for which RL is a subfield, will be in designing agents that can write code for us. This might be one way of approaching that problem.

5 Appendix

All the code in this appendix is also available for viewing or download on my github, here is a link to the repository: https://github.com/rc805/Reinforcement-Learning-Identifying-Learners

A. Bellman Equation

The Bellman equation is used (in some form) in all RL algorithms. Is defined by the following:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] \Big] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S}, \end{aligned}$$

Fig. 6[1]

Here, $\boldsymbol{v_\pi}(s)$ is the *value function* corresponding to the policy $\boldsymbol{\pi}$, and *s'* is the next possible state. All other variables are defined in Section 2. This equation is used to evaluate the value (potential future reward) of being in the current state.

B. Policy Iteration

This is one of the two offline algorithms that I implemented to efficiently solve the initial environments. It has been proven that, for a finite MDP, this method will *always* converge to an optimal policy.[1] This is due to the fact that there are a finite number of total possible policies.[1] The algorithm follows two steps: *policy evaluation* and *policy improvement*. Essentially, at each timestep ***t*** we evaluate our current policy using the Bellman Equation (Apdx. A). Then, we

perform a one-step lookahead and apply the Bellman Equation to each of the next possible states. Finally, we use this information to improve our current policy by taking the argmax of the value function for each of these states. The action that would give us the highest valued next state is added to our current policy function. A pseudocode outline of this process is shown below:

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Fig. 7[1]

Here, $\boldsymbol{\pi_*}$ represents an optimal policy. The following is a script that I wrote which implements and tests the efficiency of Policy Iteration in the Frozen Lake environment.

FL-PolicyIteration.py

```
import gym
import numpy as np
import math
import timeit
import numpy.linalg as LA
```

```
env = gym.make('FrozenLake-v0')

S = env.observation_space.n
A = env.action_space.n

def MDPparams(S, A):

    trans_p = {}
    r = np.zeros([S,A])

    for i in range (A):

        M = np.zeros([S,S])

        for j in range(S):

            #set up transition matrix
            lis = [0]*S
            lis[env.P[j][i][0][1]] = 1

            M[:,j] = lis
            trans_p[str(i)] = M
            #set up reward matrix
            r[j][i] = env.P[j][i][0][2]

    return r, trans_p

def solveMDP(r,trans_p):
    # start with random initial policy
    # only have to search over deterministic policies.

    gamma = 0.99
    pi_old = ['0']*S
    dpi = True

    while dpi:

        # calculate V of pi
```

```
        B = np.zeros([S,S])
        for i in range(S):
            B[i,:] = trans_p[pi_old[i]][:,i]

        r2 = np.zeros(S)
        for i in range(S):
            r2[i] = r[i,int(pi_old[i])]

        V = np.dot(LA.inv(np.identity(S)-gamma*B),r2)


        # calculate Q
        Q = np.zeros([S,A])
        for i in range(A):
            Q[:,i] = r[:,i]+gamma*np.dot(trans_p[str(i)].T,V)


        # improve the policy
        dpi = False
        for i in range(S):

            new = str(np.argmax(Q[i,:]))
            if pi_old[i]!=new:
                dpi = True

            pi_old[i] = str(np.argmax(Q[i,:]))


    # calculate average reward
    M = np.zeros([S,S])
    for j in range(S):

        M[:,j] = trans_p[pi_old[j]][:,j]

    w, v = LA.eig(M)
    ind = np.argmin(np.abs(w-1))
    stationary_dist = v[:,ind]; stationary_dist /= np.sum(stationary_dist)
    avg_r = np.dot(V,stationary_dist)*(1-gamma)
```

```
    reshapedPolicy = ReshapePolicy(S, pi_old)
    print("Policy Map (0=left, 1=down, 2=right, 3=up):")
    print(reshapedPolicy)
    print("")
    print("Environment:")
    env.render()

    return reshapedPolicy, avg_r

#turn policy function from a list into a matrix
def ReshapePolicy(S,pi):
    rows = int(math.sqrt(S))
    cols = int(math.sqrt(S))
    reshapedPolicy = np.zeros([rows,cols])
    counter = 0
    for i in range(rows):
        for j in range (cols):
            reshapedPolicy[i][j] = int(pi[counter])
            counter += 1
    return reshapedPolicy

#initialize reward and transition matrices
rewards, transProbs = MDPparams(env.nS, env.nA)

#wrap function so we cant time it
def functionWrapper(f, *args, **kwargs):
    def wrappedFunction():
        return f(*args, **kwargs)
    return wrappedFunction

wrapped = functionWrapper(solveMDP, rewards, transProbs)
print("Execution Time:", timeit.timeit(wrapped, number = 1))
```

C. Q-learning (Watkins, 1989)[4]

This is one of the two online algorithms that I implemented with which the RL agents were equipped. Q-learning utilizes what's called an ***action value function***, which represents the value of being in a state *s* and taking action *a.* The core idea of Q-learning is to approximate the value of state-action pairs, i.e. the Q-function, from the samples of Q(s, a) that we observe during interaction with the environment. Then, the agent then acts "greedily" to maximize its reward for the next step, or it acts randomly to explore the environment. A pseudocode outline of the algorithm is shown below. Here, α represents the learning rate.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Fig. 8[1]

The following is a portion of the script from Appendix F showing the implementation of the Q-learning algorithm to our problem.

From *RLtest.py*

```
elif algorithm=='Q-learning':

        # Run with Q learning
        s0 = np.random.choice(S)
        gamma = 0.99
```

```
        a0 = '0'
        R = r[s0,int(a0)]

        # Initialize Action-Policy function
        Q_Q = 10*np.mean(r)*np.ones(shape=[S,A])
        alpha = 0.1

        for i in range(train_time-1):

            # Sample the environment
            s1 = np.random.choice(a=S,p=trans_p[a0][:,s0])

            # Update Action-Policy Function
            dQ_Q = r[s0,int(a0)]+gamma*np.max(Q_Q[s1,:])-Q_Q[s0,int(a0)]
            Q_Q[s0,int(a0)] += alpha*dQ_Q/(i+1)

            # Explore vs. exploit
            u = np.random.uniform()
            epsilon = 0.1
            if u<1-epsilon:
                a0 = str(np.argmax(Q_Q[s1,:]))
            else:
                a0 = str(np.random.choice(A))

            # Update R
            R += r[s0,int(a0)]
            a0 = str(a0)
            s0 = s1
        R /= train_time
```

The “Explore vs. Exploit” section assigns a probability, defined by 1 minus epsilon, that our agent might take a random action instead of an optimal one so as to “explore” the environment. This allows our agent to experience more of the environment and possibly discover very valuable states.

D. Value Iteration

This is the second offline algorithm that I implemented as a means of solving MDPs. This algorithm revolves around computing a value function for an initial policy, and then improving that policy by calculating the value function if the agent were to take a particular action. The maximum potential value function for the possible set of actions is used to inform the best action for each state. A pseudocode outline of this process is shown below.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

Fig. 9[1]

When I tested this algorithm, I had not yet utilized OpenAI's gym package[5]. I instead implemented it to solve an environment called Gridworld, taken from Chapter 4 of Sutton and Barto's *Reinforcement learning: an introduction*.[1] Gridworld is very similar to Frozen Lake, in that it is a 2-D grid of states. However, it is different in that every state has a reward of -1 except for two terminal states that have a reward of 0. Thus, solving the MDP amounts to finding the shortest path from the starting position to one of the terminal states. The following is a script I wrote to solve the Gridworld using Value Iteration.

Value-Iteration.py

```
import numpy as np
import sys
import gym.spaces
import timeit
if "../" not in sys.path:
  sys.path.append("../")
from gridworld import GridworldEnv

environment = GridworldEnv()
def value_iteration(environment, discountFactor=0.9, minError=0.1):

    def lookahead(V, a, s):

        [(next_state, reward, done)] = environment.P[s][a]
        # Bellman equation
        value = (reward + discountFactor * V[next_state])
        return value

    # Initial value function and policy
    V = np.zeros(environment.nS)
    policy = np.zeros([environment.nS, environment.nA])

    while True:

        error = 0

        # Loop over states
        for s in range(environment.nS):

            actions_values = np.zeros(environment.nA)

            for a in range(environment.nA):

                # Apply Bellman eqn
                actions_values[a] = lookahead(V, a, s)
```

```
            # Update value function and error
            best_action_value = max(actions_values)
            error = max(error, abs(best_action_value - V[s]))
            V[s] = best_action_value

            # Update policy function
            best_action = np.argmax(actions_values)
            policy[s] = np.eye(environment.nA)[best_action]


        # Break if close enough to optimal value function
        if(error < minError):
            break

    print("")
    print("Grid Policy (0=up, 1=right, 2=down, 3=left):")
    print(np.reshape(np.argmax(policy, axis=1), environment.shape))
    print("")

    #print("Grid Value Function:")
    #print(V.reshape(environment.shape))
    #print("")

    return policy, V

# used to time the script
def functionWrapper(f, *args):
    def wrappedFunction():
        return f(*args)
    return wrappedFunction

wrapped = functionWrapper(value_iteration, environment)
print("Execution Time:", timeit.timeit(wrapped, number = 1))
```

E. TD Learning - TD(0)

This is the second online algorithm that I implemented as a means of representing our RL agents. TD learning is a set of algorithms that utilize a tactic called ***bootstrapping***; this means that we can learn from incomplete episodes by means of substituting running through to the end of an episode with an estimation of our future reward. I implemented the simplest TD algorithm, TD(0) which estimates the value function based on a prediction of the value function of the next state *s'* if we are in stats *s* and take action *a*. TD(0) is special *one-step* case of the more general algorithm, TD(γ). A pseudocode outline of TD(0) is shown below.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Fig. 10[1]

As with my implementation of Value Iteration, my script for testing TD(0) was also implemented for the Gridworld environment.

TDlearner.py

```
import numpy as np
import itertools
import pandas as pd
import sys
import timeit
```

```
from gridworld import GridworldEnv
from random import randint
from numpy import empty

env = GridworldEnv()
actionSteps = env.P

# Used to generate a "greedy" policy
def generatePolicy(Q, epsilon, nA):

    def policyFunc(observation):
        A = np.ones(nA, dtype=float) * epsilon / nA
        bestAction = np.argmax(Q[observation])
        A[bestAction] += (1.0 - epsilon)
        return A
    return policyFunc


def TDlearn(env, numEpisodes, discountFactor=0.9, alpha=0.5, epsilon=0.1):


    # Initialize action-value and policy functions
    Q = np.random.uniform(size=[env.nS,env.nA])
    policy = generatePolicy(Q, epsilon, env.nA)


    for i_episode in range(numEpisodes):
        # Track episodes in console because I am fancy and my programs are
fancy
        if (i_episode + 1) % 100 == 0:
            print("\rEpisode {}/{}.".format(i_episode + 1, numEpisodes),
end="")
            sys.stdout.flush()

        # Reset agent to random starting position
        s = randint(0, env.nS - 1)
        actionProbs = policy(s)
        a = np.random.choice(np.arange(len(actionProbs)), p=actionProbs)
```

```
        # Step through environment
        while True:

            [(nextState, reward, done)] = actionSteps[s][a]
            # Pick the next action
            nextActionProbs = policy(nextState)
            nextAction = np.random.choice(np.arange(len(nextActionProbs)),
p=nextActionProbs)

            # Update Q
            td_target = reward + discountFactor * Q[nextState][nextAction]
            td_delta = td_target - Q[s][a]
            Q[s][a] += alpha * td_delta

            if done:
                break

            a = nextAction
            s = nextState

    # Extract policy from Q
    rows = env.shape[0]
    cols = env.shape[1]
    counter = 0
    extractedPolicy = np.zeros([rows,cols])
    for i in range(rows):
        for j in range (cols):
            extractedPolicy[i][j] = int(np.argmax(Q[counter]))
            counter += 1

    print("")
    print("")
    print("Action Value Function:")
    print(Q)

    print("")
    print("")
```

```
    print("Policy Map (0=up, 1=right, 2=down, 3=left):")
    print(extractedPolicy)
    print("")

    return Q, extractedPolicy


# Used to time the execution of the script
def functionWrapper(f, *args, **kwargs):
    def wrappedFunction():
        return f(*args, **kwargs)
    return wrappedFunction

wrappedTD = functionWrapper(TDlearn, env, 10000)
print("Execution Time:", timeit.timeit(wrappedTD, number = 1))
```

F. Testing Script

This is the script, written in collaboration by both myself and Dr. Marzen, that has given us the results shown in section 3. It incorporates Policy Iteration to solve the initial environment, Q-learning to represent the RL agents, and employs the outlier detection method.

RLtest.py

```
import gym
import numpy as np
import pylab as pl
import math
import timeit
import numpy.linalg as LA
from scipy.stats import norm

# Authors: Dr. Sarah Marzen, Ryan Cullen

def MDPparams():
    env = gym.make('FrozenLake-v0')
```

```
    S = env.observation_space.n
    A = env.action_space.n
    trans_p = {}
    r = np.zeros([S,A])

    for i in range (A):
        M = np.zeros([S,S])
        for j in range(S):
            #set up transition matrix
            lis = [0]*S
            lis[env.P[j][i][0][1]] = 1
            M[:,j] = lis
            trans_p[str(i)] = M
            #set up reward matrix
            r[j][i] = np.random.uniform(-10, 10)
            #r[j][i] = env.P[j][i][0][2]
    return r, trans_p

# METHOD: Policy Iteration
def solveMDP(r,trans_p):
    # start with random initial policy
    # only have to search over deterministic policies.
    S, A = r.shape
    gamma = 0.99
    pi_old = ['0']*S
    dpi = True
    while dpi:

        # calculate V of pi_0
        B = np.zeros([S,S])
        for i in range(S):
            B[i,:] = trans_p[pi_old[i]][:,i]
        r2 = np.zeros(S)
        for i in range(S):
            r2[i] = r[i,int(pi_old[i])]
        V = np.dot(LA.inv(np.identity(S)-gamma*B),r2)
```

```
        # calculate Q
        Q = np.zeros([S,A])
        for i in range(A):
            Q[:,i] = r[:,i]+gamma*np.dot(trans_p[str(i)].T,V)

        # improve the policy
        dpi = False
        for i in range(S):
            new = str(np.argmax(Q[i,:]))
            if pi_old[i]!=new:
                dpi = True
            pi_old[i] = str(np.argmax(Q[i,:]))

    # calculate average reward
    M = np.zeros([S,S])
    for j in range(S):
        M[:,j] = trans_p[pi_old[j]][:,j]
    w, v = LA.eig(M)
    ind = np.argmin(np.abs(w-1))
    stationary_dist = v[:,ind]; stationary_dist /= np.sum(stationary_dist)
    avg_r = np.dot(V,stationary_dist)*(1-gamma)

    return pi_old, avg_r

def run_Envt(r,trans_p,pi,train_time,algorithm='None'):
    S, A = r.shape
    if algorithm=='None':

        # run it for all of train_time
        s = np.random.choice(S)
        a = np.random.choice(A, p=pi[s,:])
        R = r[s,a]

        for i in range(train_time-1):
            # sample to get s
            s = np.random.choice(S,p=trans_p[str(a)][:,s])
            a = np.random.choice(A, p=pi[s,:])
            R += r[s,a]
```

```
        R /= train_time
    elif algorithm=='Q-learning':

        # Run with Q learning
        s0 = np.random.choice(S)
        gamma = 0.99
        a0 = '0'
        R = r[s0,int(a0)]

        # Initialize Action-Policy function
        Q_Q = 10*np.mean(r)*np.ones(shape=[S,A])
        alpha = 0.1

        for i in range(train_time-1):

            # Sample the environment
            s1 = np.random.choice(a=S,p=trans_p[a0][:,s0])

            # Update Action-Policy Function
            dQ_Q = r[s0,int(a0)]+gamma*np.max(Q_Q[s1,:])-Q_Q[s0,int(a0)]
            Q_Q[s0,int(a0)] += alpha*dQ_Q/(i+1)

            # Explore vs. exploit
            u = np.random.uniform()
            epsilon = 0.1
            if u<1-epsilon:
                a0 = str(np.argmax(Q_Q[s1,:]))
            else:
                a0 = str(np.random.choice(A))

            # Update R
            R += r[s0,int(a0)]
            a0 = str(a0)
            s0 = s1
        R /= train_time
    else:
        R = 0
    return R
```

```
def ROC_curve(train_time):
    # have them acclimate to environment
    r1, trans_p1 = MDPparams()
    pi_old, r_avg = solveMDP(r1,trans_p1)
    S, A = r1.shape
    # turn the pi from solveMDP into a matrix of S by A
    pi = np.zeros([S,A])
    for i in range(S):
        pi[i,int(pi_old[i])] = 1
    # generate new environment
    r2, trans_p2 = MDPparams()
    # figure out the distribution for unchanging pi; this gives the null
hypothesis
    r_null = []
    for i in range(1000):

r_null.append(run_Envt(r2,trans_p2,pi,train_time,algorithm='None'))
    r_null = np.asarray(r_null)
    mean_r_null = np.mean(r_null)
    std_r_null = np.std(r_null)
    p_null =
np.exp(-(r_null-mean_r_null)**2/2/std_r_null**2)/np.sqrt(2*np.pi*std_r_nul
l**2)
    # then run for all the algorithms
    r_Qlearning = []
    for i in range(1000):

r_Qlearning.append(run_Envt(r2,trans_p2,pi,train_time,algorithm='Q-learnin
g'))
    r_Qlearning = np.asarray(r_Qlearning)
    mean_r_Qlearning = np.mean(r_Qlearning)
    p_Qlearning =
np.exp(-(r_Qlearning-mean_r_null)**2/2/std_r_null**2)/np.sqrt(2*np.pi*std_
r_null**2)

    # calculate FPR and TPR for each algorithm separately
    thresholds = np.linspace(np.min(r_null) ,np.max(r_Qlearning),1000)
```

```
    FPR = np.zeros(shape=[1,len(thresholds)])
    TPR = np.zeros(shape=[1,len(thresholds)])

    # look for false positive rate for Q learning methods
    for i in range(len(thresholds)):
        FPR[0,i] = np.sum((r_null>thresholds[i]))/len(r_null)
        TPR[0,i] = np.sum((r_Qlearning>thresholds[i]))/len(r_Qlearning)

    #Calculate Area Under Curve
    AUC = 0
    for i in range(len(thresholds) - 1):
        AUC -= TPR[0,i] * (FPR[0,i+1]-FPR[0,i])

    return FPR, TPR, r_Qlearning, r_null, AUC

FPR, TPR, rQ, rNull, AUC = ROC_curve(50)

print(AUC)

mean,std = norm.fit(rQ)
pl1 = pl.hist(rQ, bins=20, density=True)
xmin, xmax = pl.xlim()
x = np.linspace(xmin, xmax, 100)
y = norm.pdf(x, mean, std)
pl.plot(x, y)

mean,std = norm.fit(rNull)
pl2 = pl.hist(rNull, bins=20, density=True)
xmin, xmax = pl.xlim()
x = np.linspace(xmin, xmax, 100)
y = norm.pdf(x, mean, std)
pl.plot(x, y)
pl.show()

pl.plot(FPR.T,TPR.T)
pl.xlabel('FPR')
pl.ylabel('TPR')
pl.show()
```

Citations

1. Sutton, R. S., & Barto, A. (2018). *Reinforcement learning: an introduction*. Cambridge, MA: The MIT Press.
2. Kussell, E. (2005). Phenotypic Diversity, Population Growth, and Information in Fluctuating Environments. *Science*, *309*(5743), 2075–2078. doi: 10.1126/science.1114383
3. Willsky, A. S., Wornell, G. W., and Shapiro, J. H. (2003). *Stochastic processes: detection and estimation*. MIT 6.432
4. Watkins, C.J.C.H. (1989).*Learning from delayed rewards*. PhD Thesis, University of Cambridge, England.
5. OpenAI. (n.d.). A toolkit for developing and comparing reinforcement learning algorithms. Retrieved from https://gym.openai.com/envs/#toy_text
6. Salimans, T., Chen, X., Ho, J., Sidor, S., & Sutskever, I. (2017). Evolution Strategies as a Scalable Alternative to Reinforcement Learning. doi: 1703.03864
7. Montague, P., Dayan, P., & Sejnowski, T. (1996). A framework for mesencephalic dopamine systems based on predictive Hebbian learning. *The Journal of Neuroscience*, *16*(5), 1936–1947. doi: 10.1523/jneurosci.16-05-01936.1996
8. Potjans, W., Diesmann, M., & Morrison, A. (2011). An Imperfect Dopaminergic Error Signal Can Drive Temporal-Difference Learning. *PLoS Computational Biology*, *7*(5). doi: 10.1371/journal.pcbi.1001133
9. Silver, David. [DeepMind]. (2015). *Introduction to Reinforcement Learning with David Silver* [Video series]. Youtube. https://www.youtube.com/playlist?list=PLqYmG7hTraZBiG_XpjnPrSNw-1XQaM_gB