# Classic and Risk-Sensitive Reinforcement Learning

**Tanner Fiez**
University of Washington
fiezt@uw.edu

## 1 Introduction

Reinforcement learning (RL) is a subfield of machine learning that focuses on learning from interactions. The core idea of RL is that an agent learns through the consequences of actions instead of being explicitly taught how to act. Actions are chosen based on past experience and new choices, which can be thought of as learning through trial and error. Inherently this forces a tradeoff between exploitation and exploration that must be solved. This style of learning has often been traced to how humans learn. With each choice we make, feedback from the environment—positive or negative—provides information on how to make future choices. In computer science this abstraction is often ignored and the focus is on computational approaches to learning from interactions.

The distinguishing factors of an RL problem are a closed–loop (actions influence later feedback), indirect feedback (the agent is not explicitly told what to do), and feedback playing out over an extended period of time (feedack from an action may come well after an action is taken). The most fundamental method to model such characteristics is through a Markov Decision Process (MDP). The MDP formulation is designed to include sensing, actions, and a goal in a simple form. An algorithm that solves an MDP is considered to be an RL algorithm. We provide a description of an MDP in Section 2.

We will begin by examining the theory for solving MDPs with a focus on the distinct approaches of model based algorithms and model free algorithms in Sections 3 and 4 respectively. In this work we will only consider finite MDPs, i.e. problems with discrete state and action spaces. Note that many problems with continuous state and action spaces can be discretized so that the methods we will discuss are still applicable. The first major contribution of this work is the implementation of many classic RL algorithms in a flexible, object-oriented framework[1]. In support of the algorithms, we develop a grid-world environment that allows for unique problem specifications, rapid testing and comparison of algorithms, and visualization of results. Moreover, we have made our library compatible with OpenAI Gym[2] [1]. OpenAI Gym is a recently developed python toolkit containing a wide variety of RL environments for evaluation purposes. This was a significant step in developing benchmarks for RL because the problems do not arise as naturally as supervised learning problems. Despite this advancement, there is still no universally used python package containing RL algorithms, providing motivation for this portion of the work as we seek to begin to create our own scikit-learn like package to make RL more accessible and ubiquitous. For an overview of implementation details see Appendix Section A.

Following our work with classic RL, we delve deeper into an interesting and promising new line of work called risk-sensitive RL. RL algorithms have historically modeled agents as expected utility maximizers. This modeling paradigm thus considers agents as rational decision makers. A rational decision maker can be described as risk-neutral. However, extensive work in behavior psychology, cognitive science, and economics has shown that humans are inherently irrational decision makers acting according to both a reference point and an internal set of risk preferences. This phenomenon has revealed that humans distort event probabilities and value losses and gains asymmetrically.

[1]Implementations and many more examples of the algorithms than we had room to show using both grid-world and OpenAI Gym are available at the github repo for this project: `https://github.com/fiezt/ML-Project`.

[2]https://gym.openai.com/envs

Specifically, low probability events are overestimated and high probability events are underestimated, and losses are weighed more heavily than gains (see Appendix Section B for a concrete example). In [2] an RL framework to model risk-sensitive decision making was developed leveraging behavioral models of human decision making. The results show that many of the convergence properties and optimality conditions from classic RL still apply. We discuss the methods and implications of the paper, our implementation, and the tests we run in Section 5.

Our final contribution, detailed in Section 6, is to apply the RL methods we discuss to the New York Taxi dataset[3] [3]. In this problem we formulate an MDP for taxi drivers and pre-process the data—which includes trip times, distances, fares, and pick-up and drop-off locations—accordingly to create the environment. The goal in this problem is to find the optimal policy for a driver, i.e. to find where a driver should go to look for a new ride following dropping off passengers to maximize their earning rate. Because we have access to the data, we can find the empirical policy of a driver and compare this to the optimal policy.

## 2 Markov Decision Process

The key requirement of the state and environment in an MDP is that they obey the Markov property. The Markov property refers to the memoryless property of a stochastic process. In plain language, the Markov property says that given the present, the future is independent of the past. In the most general case the dynamics of a process are defined by the joint probability distribution

$$\mathcal{P}(S_{t+1}=s', R_{t+1}=r'|S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t). \tag{1}$$

In the case where the Markov property is satisfied, the dynamics can equivalently be represented by the following:

$$p(s', r'|s, a) \triangleq \mathcal{P}(S_{t+1}=s', R_{t+1}=r'|S_t=s, A_t=a). \tag{2}$$

The Markov property is fundamental in reinforcement learning because the dynamics of one transition allow for prediction of the next state and the expected reward given only the current state and action. In RL problems, even when the state does not obey the Markov property, it is often still thought of as at least approximating it [4].

Given the dynamics specified by the Markov property in (2) all quantities of interest with respect to the environment can be computed. Specifically, we can determine the state-transition probabilities and the expected rewards of state-action-state triples. The state-transition probabilities are obtained by marginalizing out the rewards.

$$p(s'|s, a) \triangleq \mathcal{P}(S_{t+1}=s'|S_t=s, A_t=a) = \sum_{r'\in\mathcal{R}} p(s', r'|s, a). \tag{3}$$

The expected rewards are obtained by using the definition of expectation.

$$r(s, a, s') \triangleq \mathbb{E}[R_{t+1}|S_t=s, A_t=a, S_{t+1}=s'] = \frac{\sum_{r'\in\mathcal{R}} r' p(s', r'|s, a)}{p(s'|s, a)}. \tag{4}$$

We can now define an MDP using the above quantities. An MDP is a tuple given as follows:

$$\text{MDP} = (\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot,\cdot), \gamma). \tag{5}$$

The quantities encompassed by the MDP are defined as:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of actions.
- $\mathcal{P}(s'|s, a)$ is a transition kernel giving the probability that taking action $a$ in state $s$ will lead to state $s'$.
- $\mathcal{R}(s, a, s')$ is a reward kernel giving the reward received from taking action $a$ in state $s$ and ending up in state $s'$.
- $\gamma \in [0, 1]$ is a discounting factor on the rewards representing the importance of immediate and future rewards.

With the problem framework defined we now focus our attention on defining what it means to solve or approximately solve an MDP and methods to do so computationally.

[3]The New York Taxi dataset is available at `https://publish.illinois.edu/dbwork/open-data/`

## 3 Model Based Reinforcement Learning

Recall that the goal in RL is to find the optimal policy. This means we want to learn the optimal action to take in each state in the state space. Methods that use a model of the environment given by an MDP to compute an optimal policy are referred to as model based RL algorithms. These methods utilize dynamic programming principles and are sometimes referred to as planning methods because they are offline in the sense that they do not require explicit interaction with the environment.

RL algorithms almost always estimate value functions. Value functions are functions of states or state-action pairs which estimate how much value a state or state-action pair has. The value here means the expected future rewards from a state or state-action pair. Because the expected future rewards depend on future actions, value functions are defined with respect to a policy. We define a policy as a probability mass function from a state to an action. Formally we will denote a policy as $\pi(a|s)$ and when we drop $a, s$ this denotes following $\pi$ at each state encountered. The value function of state $s$ under a policy $\pi$ is then given by

$$v_\pi(s) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s], \tag{6}$$

and similarly the value function of a state-action pair $s, a$ under a policy $\pi$ is then given by

$$q_\pi(s, a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]. \tag{7}$$

An intermediate result that we will make use of later is found by simply removing the first term from the sum in both the state-value function

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s], \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s], \end{aligned} \tag{8}$$

and in the state-action value function.

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a], \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \end{aligned} \tag{9}$$

Each of these definitions can be completely unrolled through recursive relationships to give the following equivalent formulations

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s', r'} p(s', r'|s, a)[r' + \gamma v_\pi(s')], \tag{10}$$

$$q_\pi(s, a) = \sum_{s', r'} p(s', r'|s, a)[r' + \gamma v_\pi(s')], \tag{11}$$

which are known as the Bellman equations. This formulation is convenient as it is explicitly clear that the sum over $s', r'$ is an expectation over a state-action pair $s, a$ as we noted was the meaning of a value function.

The preceding expressions define value functions for a policy, while the goal of RL is to find the optimal policy. This lends naturally to a set of optimization problems that must be solved.

$$v_*(s) = \max_\pi v_\pi(s) \quad \text{and} \quad q_*(s, a) = \max_\pi q_\pi(s, a). \tag{12}$$

The solutions to these optimization problems give what are known as the Bellman optimality conditions. The derivations follow from the Bellman equations and are provided in Appendix Section C.

$$v_*(s) = \max_a \sum_{s', r'} p(s', r'|s, a)[r' + \gamma v_*(s')], \tag{13}$$

$$q_*(s, a) = \sum_{s', r'} p(s', r'|s, a)[r' + \gamma \max_{a'} q_*(s', a')]. \tag{14}$$

This is a famous result and is detailed in [5]. It is also worth noting that the solution is unique, this argument hinges on formulating the Bellman equation as a fixed point problem and showing that it is a contraction. Additionally, if the dynamics are known the optimality conditions reduce to the problem of solving a system of equations in the dimension of the state space, meaning that any nonlinear system equation solving method can be applied. The optimal policy then naturally follows from this analysis. In the case of the state-value function the optimal policy comes from finding the actions in each state which obtains the maximum of the Bellman optimality condition. Formally this is the following equation

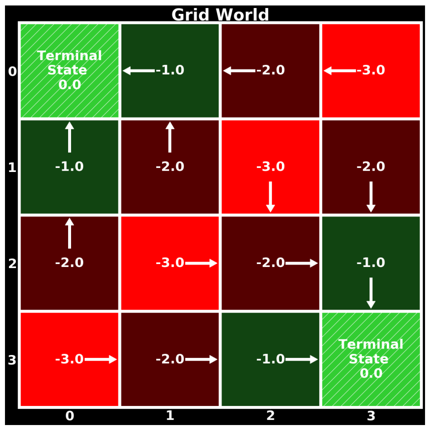
$$\pi^*(s) = \arg\max_a \sum_{s',r'} p(s', r'|s, a)[r' + \gamma v_*(s')]. \tag{15}$$

Similarly, in the case of the state-action value function the optimal policy simply comes from taking the action which maximizes the state-action value function at each state. The optimal policy is thus a greedy policy over the value function.

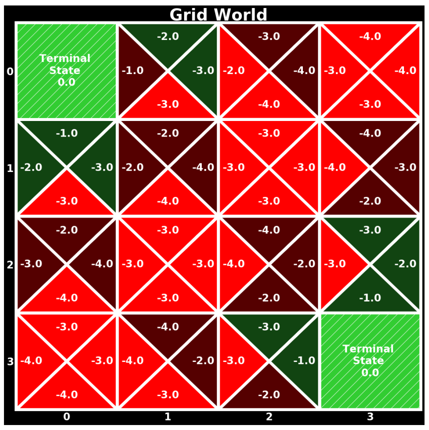
The reason such attention was paid to defining the state value function and the state-action value function and deriving the optimality conditions is that these conditions form the basis for nearly all RL algorithms whether through solving explicitly or through approximation. Model based dynamic programming methods in fact use exactly the Bellman equations and the Bellman optimality conditions. We have implemented the three primary methods: these are policy iteration, value iteration, and $q$-value iteration. In this section we give an overview and refer the reader to Appendix Section D for the explicit algorithms.

Policy iteration sweeps over the state space and alternates between two steps until convergence. These are policy evaluation, given by (10), and policy improvement (15). Value iteration sweeps over the state space and applies the optimality condition in (13) until convergence. $q$-value iteration sweeps over the state space and applies the optimality condition in (14) until convergence. Each of these algorithms are optimal and will arrive at the same solution.

We now provide results testing these algorithms and demonstrate that grid-world environment. To prove that the algorithms are implemented correctly we use a very simple MDP. We designate the state space to be indexes of the grid and actions to be the cardinal directions $\{N, E, S, W\}$. The transition function is deterministic (actions take the agent to the desired state) with the exception that actions that cause the agent to go off the grid result in the agent staying in the same state with probability $1$ and an agent remains in a terminal state when reached, all transitions incur a reward of $-1$ with the exception transitions from a terminal state incur a reward of $0$, and the discount factor is $1$. Thus the problem is to find the shortest path to a terminal state from an initial state which would be the Manhattan Distance.



(a) Value iteration and policy iteration. Arrows indicate the policy in a state and the numbers represent the state values.



(b) $q$-value Iteration. Numbers represent the state-action values.

Figure 1: Model based algorithms in grid-world.

Fig. 1 illustrates the results of the model-based learning algorithms with value and policy iteration in Fig. 1a and $q$-value iteration in Fig. 1b. As was previously mentioned would be the case, each of these methods arrives at the same optimal policy. The evaluation criteria in this setting are the Bellman optimality conditions as well as the cumulative rewards of trials. Because this is a deterministic example we are able to satisfy the Bellman optimality conditions and we will always achieve the maximum possible reward in each trial as a result.

While the algorithms given in this section may appear as the be-all and end-all to RL given that we have shown they are optimal, in reality these methods have significant limitations. This is because as the state space grows very large dynamic programming methods become computationally intractable, creating the need for methods that approximate the Bellman optimality conditions.

## 4 Model Free Reinforcement Learning

Model based dynamic programming methods face two significant limitations. As touched on previously, they become computationally intractable as the state space grows very large. Another limitation is that complete knowledge of the dynamics are required, yet transition probabilities and event outcomes are often unknown a priori in practice. To make up for these shortcomings, we turn our attention towards approximation methods that gradually improve estimates of the environment and the value functions. These methods are know as model free RL because they operate in an online fashion and require no knowledge at the outset of a problem.

The fundamental concept of model free RL is temporal-difference (TD) learning. TD methods are like dynamic programming methods in that they update value estimates using prior estimates before reaching a final outcome. The difference however, is that updates are made using sample feedback from the environment instead of the idealized model of the environment from the MDP. The TD error term for the state value function is given as follows

$$TD_v = R_{t+1} + \gamma V(S_{t+1}) - V(S_t), \tag{16}$$

similarly the state-action value function is given by

$$TD_q = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t). \tag{17}$$

These quantities may look familiar and indeed we can trace them back to a form of the Bellman equations from (8) and (9) which we restate for clarity here.

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s], \tag{8}$$

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \tag{9}$$

It is now evident that the TD terms in equations (16) and (17) are sampling from the expected values of the targets from (8) and (9) and computing the error between the improved estimate of the value function with the current estimate of the value function. Intuitively the TD term can be interpreted as a gradient like term, which then lends naturally to the idea of algorithms using online stochastic gradient descent methods to improve value function estimates. It turns out in fact, that this is exactly how the most basic TD learning algorithms work. We have implemented the three core methods of TD(0), Sarsa, and $q$-learning. The methods differ by the policy that is being sampled from in the TD updates and whether a state value function or a state-action value function is being learned.

TD(0) learns the state value function of a policy that is given. Meaning given policy $\pi$ the state value function $v_\pi$ is learned by sampling from $\pi$ and updating the state value function as follows:

$$V(S) = V(S) + \alpha[R' + \gamma V(S') - V(S)]. \tag{18}$$

Sarsa is an on-policy method, meaning that the policy being followed is the same as the policy being learned, that learns the optimal state-action value function. Typically, the $\epsilon$-greedy policy or the Boltzmann policy is followed. Each of these methods trades off exploration and exploitation. Given the greedy action $a^* = \arg\max_a Q(S,a)$ the $\epsilon$-greedy policy is given by

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A|} & a = a^* \\ \frac{\epsilon}{|A|} & a \neq a^* \end{cases} \tag{19}$$

where $\epsilon \in [0, 1]$ controls the greediness. The Boltzmann policy is given by

$$\pi(a|s) = \frac{e^{\tau Q(s,a)}}{\sum_a e^{\tau Q(s,a)}} \tag{20}$$

where $\tau$ is a temperature parameter $\in [0, \infty)$ that controls the greediness. It is common to use a decay rate on the parameters controlling the greediness to make the algorithm more greedy as the environment is further explored. We have implemented these policies in support of the algorithms. Given this information, the TD update in Sarsa is given as follows:

$$Q(S, A) = Q(S, A) + \alpha[R' + \gamma Q(S', A') - Q(S, A)]. \tag{21}$$

$q$-learning is very similar to Sarsa with the exception that it is an off-policy method, meaning that the policy being followed is different from the policy being learned, that learns the optimal state-action value function. Specifically $q$-learning follows the $\epsilon$-greedy policy or the Boltzmann policy but learns the greedy policy. The TD update is given as follows:

$$Q(S', A') = Q(S, A) + \alpha[R' + \gamma \max_a Q(S', a) - Q(S, A)]. \tag{22}$$

Finally, we note that to guarantee convergence of the temporal difference algorithms the Robbins and Monro conditions [6] on the learning rate must hold

$$\sum_{t=0}^{\infty} \alpha_t(s, a) = \infty \quad \text{and} \quad \sum_{t=0}^{\infty} \alpha_t^2(s, a) < \infty, \ \forall\, s, a, \tag{23}$$

and each state-action pair must be visited infinitely often during exploration (for more on convergence we refer the reader to [7] and [4]). While clearly this is infeasible, TD methods often reach a very good value function approximation and policy quickly. An example is the problem described in 1, when testing the TD methods we find that the same optimal state value function and optimal policy is learned as the model based methods but in the state-action value function there is some error in the suboptimal state-action pairs. However, to an agent who wants to maximize their rewards, not estimating suboptimal values completely correct does not matter.

## 5 Risk-Sensitive Reinforcement Learning

The novel contribution of [2] was to extend the $q$-learning method by applying a transformation to the temporal difference term with a value function (note the conflation of the term value function, in this setting it means a type of function that maps utilities) obeying certain properties and coming from the class of functions that have been developed to model human behavior, while maintaining convergence guarantees. In particular the update in this algorithm is given by the following:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[u(R + \gamma \max_a Q(S', a) - Q(S, A))]. \tag{24}$$

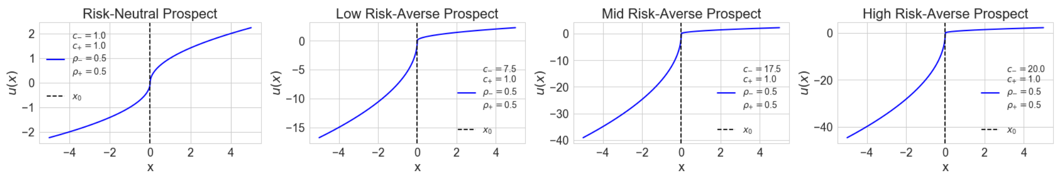
Through applying the value function to the temporal difference term, a nonlinear transformation is applied not only to the rewards, but also to the transition probabilities. This is significant given that this is precisely what humans have been observed to do when making decisions as we discussed in the introduction.

We explore and implement several functions that capture risk-sensitive decision making including a prospect theory value function [8], an entropic map value function, and a logarithm-based value function [9]. We will focus on the prospect value function given by:

$$u(y) = \begin{cases} c_+(x)^{\rho^+} & x > 0 \\ -c_-(-x)^{\rho^+} & x \leq 0 \end{cases} \tag{25}$$

where we are setting the reference point to be 0. The parameters $(c_+, c_-, \rho+, \rho_-)$ control the degree of risk-sensitivity and loss aversion. Typically human decision makers have $0 < \rho_+, \rho_- < 1$. In terms of the shape of the function these preferences correspond to concavity in gains and convexity in losses.

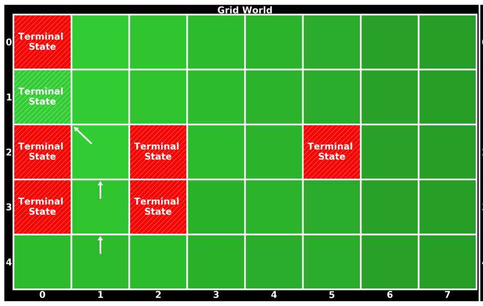
We now experiment with the value function in grid world using the configurations from Fig. 2 to learn more about how risk-sensitivity influences the optimal value function and policy from a specific



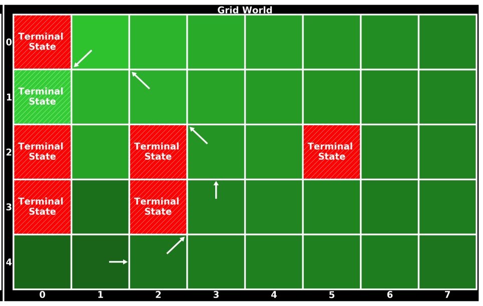
(a) (b) (c) (d)

Figure 2: Prospect Theory value functions with varying risk-sensitivity parameters. In particular we fix $(c_+ = 1, \rho + = .5, \rho_- = .5)$ and vary $c_- \in \{1, 7.5, 17.5, 20\}$ to produce the functions from left to right.

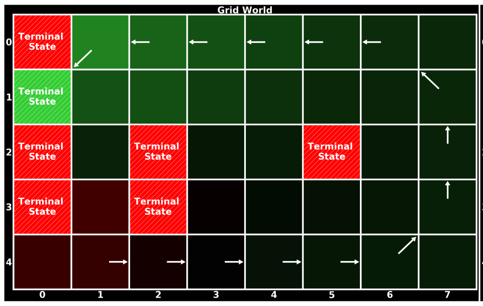
initial state. In this example we designate the state space to be indexes of the grid and actions to be the compass directions $\{N, E, S, W, NE, SE, SW, NW\}$. The transition function is configured such that an action results in the desired state with probability .93 and a random state with probability .07 except a terminal state is reached the agent remains there. All transitions incur a reward of .1 with the exception transitions to and from a terminal state incur a reward of 1 for good terminal states and $-1$ for bad terminal states. The discount factor was set to .95.



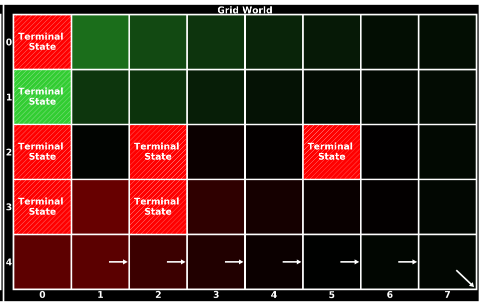
(a) Risk-Neutral.



(b) Low Risk-Averse.



(c) Mid Risk-Averse.



(d) High Risk-Averse.

Figure 3: Risk-Sensitive $q$-learning results with varying risk-sensitivity parameters.

We see in Fig. 3 that a risk-neutral agent is not influenced by the small probability of going into a bad terminal state, but as the agent becomes risk averse they work to avoid the bad terminal states at growing levels by taking longer, safer paths to the good terminal state until they learn to stay far away from the bad terminal states and only collect a small reward. For the complete algorithm and additional details see Appendix Section F.

## 6 New York Taxi Dataset

We formulate the decision process of a taxi driver as a MDP in order to learn the optimal policy to maximize earning rates as well as compare to empirical polices from data to see how well the drivers do at present.

The New York Taxi Dataset covers taxi operations from 2010–2013. The key information that is contained is the hack license (driver ID), pickup date-time, dropoff date-time, trip time in seconds, trip distance in miles, GPS coordinates at starting and ending locations, total fare including tip, and the total cost of tolls. The data required a significant amount of preprocessing to clean the data and obtain the needed quantities for the MDP formulation. In fact, it was estimated by [3] that nearly $10\%$ of trips contained erroneous information. We gave particular attention to cleaning incorrect trip times, as well as infeasible trip locations, distances, and earning rates.

The complete formulation of the finite MDP for a taxi driver we derived is detailed in Appendix Section G. The salient features of the formulation are:

- Each state is a tuple containing the node the driver is in–we discretized location into a grid using district boundaries for New York City–an indicator of whether the taxi currently is full (just picked up passengers) or empty (just dropped off passengers), and the cumulative reward interval the driver is in–we discretized the reward intervals assuming that how a driver behaves may be a function of the earnings at a point in the workday.
- Actions are moving between nodes in the location grid we created.
- The reward functions use values derived from the data, such as earning rate, the expected time searching for a passenger, and the fare between grid nodes. The transition probabilities use empirical transition probabilities as well as expected earning rates.
- The policy is given by the actions taken when looking to pick up new passengers, i.e. empty to full transitions.

There are also a couple of important things we do to create the MDP which we make note of. Drivers work abnormal hours, i.e. the typical driver begins work in the evening and works until the early morning hours. In light of this we shift all transaction times back by 12 hours so that we can observe the full workday within a single date. We estimate the time spent searching for a new trip as the time from dropoff to pickup, but if this time exceeds 20 we assume the driver was taking a break.

We select 7 drivers and analyze then over a three month period in 2010. For each driver we solve the MDP using one of our model based RL algorithms and compare the empirical policy to the optimal policy for several discount factors.

Table 1: Percentage of decision states (empty to full and not in a terminal state) that the empirical policy matches the optimal policy learned.

| DriverID | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $\gamma = .9$ | 59.3 | 51.7 | 32.2 | 34.1 | 33.3 | 28.6 | 44.7 |
| $\gamma = .95$ | 60.0 | 44.4 | 17.2 | 21.8 | 26.0 | 16.4 | 36.7 |
| $\gamma = 1$ | 75.0 | 75.6 | 77.8 | 84.7 | 63.3 | 86.4 | 64.7 |

The results show that drivers do not act optimally. Interestingly they are closest to the policy learned with a discount factor of 1. In this case the policy learned was nearly always to stay in the same grid location to look for rides. As the discount factor decreased we saw increased occurrences in the learned policy of going to the grid locations that contained one of the airports. We believe this is because there are longer trips and higher earning rates but this comes at the cost of lost time in traveling there with no ride.

## 7 Conclusions and Future Work

In this project we thoroughly analyzed finite MDPs and RL theoretically and computationally by developing a significant code base of algorithms and infrastructure for experimentation. We also explored an interesting RL paradigm of risk-sensitivity and concluded with formulating the decision process of a taxi driver as an MDP. In future work we would like to apply the risk-sensitive RL to the taxi problem and tune parameters to try and find a policy that matches the empirical policy closely, which would tell us the risk-preferences a driver was acting on.

## References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[2] Yun Shen, Michael J. Tobia, Tobias Sommer, and Klaus Obermayer. Risk-sensitive reinforcement learning. *CoRR*, abs/1311.2097, 2013.

[3] B Donovan and DB Work. New york city taxi trip data (2010–2013), 2014.

[4] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[5] Richard Bellman. *Dynamic programming*. Courier Corporation, 2013.

[6] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[7] John N Tsitsiklis. Asynchronous stochastic approximation and q-learning. *Machine learning*, 16 (3):185–202, 1994.

[8] Amos Tversky and Daniel Kahneman. Advances in prospect theory: Cumulative representation of uncertainty. *Journal of Risk and uncertainty*, 5(4):297–323, 1992.

[9] Lillian J. Ratliff and Eric Mazumdar. Risk-sensitive inverse reinforcement learning via gradient methods. *CoRR*, abs/1703.09842, 2017.

# APPENDIX

## A Implementation Details

All functionality in this project was implemented on our own without the use of machine learning packages. The framework we have developed allows for testing and comparison of many RL algorithms in only a few simple lines of code thanks to the object oriented structure, much like scikit-learn. For evaluation purposes we have created a grid-world environment and made our framework compatible with OpenAI Gym.

The salient features is that we divide up RL approaches into model based, model free, and risk-sensitive and develop classes for each. Since the approaches share many features and we would like to be able to test optimality and visualize results all the same, it is natural that these are derived from base classes. When instantiating one of the objects there are parameters for that can be chosen such as the learning rate, policy type, and decay parameters.

The grid-world environment we created takes in parameters allowing for specification of the states, actions, transition probabilities, and rewards in a very simple way. Moreover it allows for extremely clear visualizations of state value functions, state-action value functions, and policies that are appealing to they eye.

Below we demonstrate how easy it is to use and test the RL algorithms in grid-world with our code. This is how we created the examples in Section 3 of model based algorithms in grid-world.

```python
import RL

# Problem specifications.
grid_rows = 4
grid_cols = 4
num_actions = 4
living_rewards = -1
terminal_states = [0, 15]
terminal_rewards = {0:0, 15:0}

# Creating the MDP.
mdp = RL.GridWorldMDP(grid_rows, grid_cols, num_actions, terminal_states,
                    terminal_rewards, prob_noise=.0,
                    living_rewards=living_rewards, reward_into=False)


# Creating the model based RL object.
model_rl = RL.ModelBasedRL()

# Run, visualize, and test for optimality of policy iteration.
model_rl.policy_iteration(mdp)
display = RL.GridDisplay(model_rl, mdp)
display.show_values(fig_name='policy_iteration.png', save_fig=True)
v_error = model_rl.test_optimal_v(mdp)

# Run, visualize, and test for optimality of value iteration.
model_rl.value_iteration(mdp)
display = RL.GridDisplay(model_rl, mdp)
display.show_values(fig_name='value_iteration.png', save_fig=True)
v_error = model_rl.test_optimal_v(mdp)

# Run, visualize, and test for optimality of q-value iteration.
model_rl.q_value_iteration(mdp)
display = RL.GridDisplay(model_rl, mdp)
display.show_q_values(fig_name='q_value_iteration.png', save_fig=True)
q_error = model_rl.test_optimal_q(mdp)
v_error = model_rl.test_optimal_v(mdp)
```

Like in grid world we can apply any of the methods described in this paper in OpenAI Gym. An example on a task of picking up a passenger and moving them to a desired location in a grid is shown below using both model free methods, and model based methods where we use an estimated MDP that is learned from interacting with the environment.

```python
import gym
import RL

# Gym environment to learn.
env = gym.make('Taxi-v2')

# We simulate the env so we can estimate an MDP to apply model based RL to.
mdp = RL.SimulatedMDP(env)


# Apply value iteration.
model_rl.value_iteration(mdp)

# Simulate using the policy that was learned and the last episode is displayed.
model_rl.simulate_policy(env)

# Plotting the returns at each episode.
model_rl.scatter_epsiode_returns()


# Creating parameters needed for ModelFreeRL class.
n = env.observation_space.n
states = range(n)
m = env.action_space.n
actions = range(m)

# Creating model free RL object.
model_free_rl = RL.ModelFreeRL(n=n, m=m, states=states, actions=actions)

# Running the q-learning algorithm.
model_free_rl.q_learning(env)

# Plotting the returns, epsilon choices, and alpha parameters at each episode.
model_free_rl.plot_epsiode_returns()
model_free_rl.plot_epsilon_parameters()
model_free_rl.plot_alpha_parameters(s=0, a=0)

# Simulate using the policy that was learned and the last episode is displayed.
model_free_rl.simulate_policy(env)
```

## B Risk-Sensitive Decision Making Example

A prevalent example to demonstrate how risk factors into human-decision making including warping of the probability of events as well as losses being weighed more significantly than gains is as follows. When asked to choose between being given \$90 or having a 10% chance of winning \$100 and a 90% chance of winning \$0 most people will choose to take the guaranteed \$90. When this question is framed as a loss however, i.e. to choose between losing \$90 or having a 10% chance of losing \$0 and a 90% chance of losing \$100, most will choose to risk an increased loss for a chance at no loss. A rational, risk-neutral decision maker would be indifferent to the options in both framings of the question since the expected value of each option is the same.

## C Bellman Optimality Conditions Derivation

The bellman optimality conditions represent that the value of a state under an optimal policy must be equal to the expected return for the best action from the state. The derivations follow from the

Bellman equations.

$$\begin{aligned}
v_*(s) &= \max_{a \in A} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi^*}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r'} p(s', r' | s, a)[r' + \gamma v_*(s')],
\end{aligned}$$

The bellman optimality equation for $q_*$ is then

$$\begin{aligned}
q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\
&= \sum_{s',r'} p(s', r' | s, a)[r' + \gamma \max_{a'} q_*(s', a')].
\end{aligned}$$

## D Model Based RL Algorithms

**Algorithm 1** Policy Iteration

```
1: procedure PolicyIteration
2:     Initialize: $v(s) = 0, \forall\, s \in \mathcal{S}, \pi(s) \in \mathcal{A}\, \forall\, s \in \mathcal{S}$
3:     1. Policy Evaluation
4:     while True do
5:         $\delta \leftarrow 0$
6:         for $s \in \mathcal{S}$ do
7:             $v_{temp} \leftarrow v(s)$
8:             $v(s) \leftarrow \sum_{s',r'} p(s', r' | s, \pi(s))[r(s, a, s') + \gamma v(s')]$
9:             $\delta \leftarrow \max(\delta, |v_{temp} - v(s)|)$
10:        if $\delta < \epsilon$ then
11:            break

12:    2. Policy Improvement
13:    stable ← True
14:    for $s \in \mathcal{S}$ do
15:        old-action ← $\pi(s)$
16:        $\pi(s) \leftarrow \arg\max_a \sum_{s',r'} p(s', r' | s, a)[r(s, a, s') + \gamma v(s')]$
17:        if old-action $\neq \pi(s)$ then
18:            stable ← False
19:        else
20:            Begin again at 1.
21:    if stable then
22:        return $v, \pi$ and end
```

**Algorithm 2** Value Iteration

```
1:  procedure ValueIteration
2:      Initialize: $v(s) = 0, \; \forall \; s \in \mathcal{S}$
3:      while True do
4:          $\delta \leftarrow 0$
5:          for $s \in \mathcal{S}$ do
6:              $v_{temp} \leftarrow v(s)$
7:              $v(s) \leftarrow \max_a \sum_{s',r'} p(s',r'|s,a)[r(s,a,s') + \gamma v(s')]$
8:              $\delta \leftarrow \max(\delta, |v_{temp} - v(s)|)$
9:          if $\delta < \epsilon$ then
10:             break
11:     $\pi \leftarrow \operatorname{argmax}_a \sum_{s',r'} p(s',r'|s,a)[r(s,a,s') + \gamma v(s')], \; \forall \; s \in \mathcal{S}$
```

**Algorithm 3** Q Value Iteration

```
1:  procedure QValueIteration
2:      Initialize: $q(s,a) = 0, \; \forall \; s \in \mathcal{S}, a \in \mathcal{A}$
3:      while True do
4:          $\delta \leftarrow 0$
5:          for $s \in \mathcal{S}, a \in \mathcal{A}$ do
6:              $q_{temp} \leftarrow q(s,a)$
7:              $q(s,a) \leftarrow \sum_{s',r'} p(s',r'|s,a)[r(s,a,s') + \gamma \max_a q(s',a')]$
8:              $\delta \leftarrow \max(\delta, |q_{temp} - q(s,a)|)$
9:          if $\delta < \epsilon$ then
10:             break
11:     $\pi \leftarrow \operatorname{argmax}_a q(s), \; \forall \; s \in \mathcal{S}$
```

## E Model Free RL Algorithms

**Algorithm 4** TD(0)

```
1:  procedure TD(0)
2:      Initialize: $V(s) = 0, \; \forall \; s \in \mathcal{S}$
3:      for each episode do
4:          Initialize S
5:          for step in episode do
6:              $A \leftarrow$ action given by $\pi$ for $S$
7:              Take action $A$, observe $R, S'$
8:              $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
9:              $S \leftarrow S'$
10:         Until end of the horizon or episode.
```

**Algorithm 5** Sarsa

```
1:  procedure Sarsa
2:      Initialize: $Q(s,a) = 0 \; \forall \; s \in \mathcal{S}, a \in \mathcal{A}$
3:      for each episode do
4:          Initialize S
5:          Choose A from S using policy derived from Q (e.g., $e$-greedy or Boltzmann)
6:          for step in episode do
7:              Take action $A$, observe $R, S'$
8:              Choose A′ from S′ using policy derived from Q (e.g., $e$-greedy or Boltzmann)
9:              $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
10:             $S \leftarrow S'$
11:         Until end of the horizon or episode.
```

**Algorithm 6** Q-Learning

```
1:  procedure Q-LEARNING
2:      Initialize: $Q(s, a) = 0 \; \forall \; s \in \mathcal{S}, a \in \mathcal{A}$
3:      for each episode do
4:          Initialize S
5:          for step in episode do
6:              Choose A from S using policy derived from Q (e.g., $e$-greedy or Boltzmann)
7:              Take action $A$, observe $R, S'$
8:              $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
9:              $S \leftarrow S'$
10:         Until end of the horizon or episode.
```

## F Risk-Sensitive Reinforcement Learning

**Algorithm 7** Risk-Sensitive Q-Learning

```
1:  procedure RISK-SENSITIVEQ-LEARNING
2:      Initialize: $Q(s, a) = 0 \; \forall \; s \in \mathcal{S}, a \in \mathcal{A}$
3:      for each episode do
4:          Initialize S
5:          for step in episode do
6:              Choose A from S using policy derived from Q (e.g., $e$-greedy or Boltzmann)
7:              Take action $A$, observe $R, S'$
8:              $Q(S, A) \leftarrow Q(S, A) + \alpha[u(R + \gamma \max_a Q(S', a) - Q(S, A))]$
9:              $S \leftarrow S'$
10:         Until end of the horizon or episode.
```

## G Taxi MDP Model Description

### G.1 State Space

The state space is

$$\mathcal{X} = \{\mathcal{N} \cup \{x_f\} \times \mathcal{S} \times \mathcal{R}\} \backslash \mathcal{X}_{na}$$

where

- $x_f$ is the terminal state representing the taxi being done for the period,
- $\mathcal{N}$ be the index set for the zones or nodes in the city with $N$ nodes. We use boundaries from the city of New York that divide up the city into approximately 200 neighborhoods. Because the overwhelming percentage of rides begin in small set of states we aggregate this to 10 nodes,
- $\mathcal{S} = \{\mathsf{e}, \mathsf{f}\}$ is an indicator of if the taxi is $\mathsf{e}$=empty or $\mathsf{f}$=full, and
- $\mathcal{R}$ is the discretized cumulative fare value space which has the structure

  $$\mathcal{R} = \{\mathcal{R}_1\} \cup \cdots \cup \{\mathcal{R}_m\} \cup \{\mathcal{R}_f\}$$

  where $\mathcal{R}_i = [a_i, b_i]$ with $a_1 < b_1 \leq a_2 < b_2 \cdots \leq a_m < b_m = \bar{r}$ and $\mathcal{R}_f = [\bar{r}, \infty)$ where $\bar{r}$ is some reference point for period earnings (e.g., if the period of investigation is a day, then this is the daily earnings reference point which could be the median or mean). The purpose of this is to impose some structure on the problem owing to the thought that how a driver acts may be dependent on their cumulative earnings in the day. We use intervals of \$20 apart for our experiments.
- $\mathcal{X}_{na}$ are the states not allowed and is defined by

  $$\mathcal{X}_{na} = \{(x_f, \mathsf{f}, r), r \in \mathcal{R}\} \cup \{(x_f, \mathsf{e}, r), r \notin \mathcal{R}_f\}$$

A state $(i, s, r) \in \mathcal{N} \times \mathcal{S} \times \mathcal{R}$ indicates the taxi is in node $i$ (or terminal state $x_f$ if $i = x_f$), has a empty/full state of $s$ and has current cumulative fare value $r$. The terminal state is reached when the fare value portion of the state is greater than or equal to $\bar{r}$.

The dimension of the state space is thus,

$$(\dim(\mathcal{N}) + \dim(\{x_f\})) \times \dim(\mathcal{S}) \times \dim(\mathcal{R}) - |\mathcal{X}_{na}|.$$

## G.2 Action Space

Let $\mathcal{U} = \mathcal{U}_a \cup \{\emptyset\}$ where $\mathcal{U}_a = \{u_{i \mapsto j}, (i,j) \in \mathcal{N} \times \mathcal{N}\}$ be the action space where $u_{i \mapsto j}$ indicates the choice of going from node $i$ to node $j$ and where $\emptyset$ is the null action. The admissible actions are state dependent. In particular, if the state is $x = (i, \mathsf{e}, r)$ for any $i \in \mathcal{N}$ and any cumulative fare value $r \in \mathcal{R}$, then $\mathcal{U}(i, \mathsf{e}, r) = \{u_{i \mapsto j}, (i,j) \in \mathcal{N} \times \mathcal{N}\}$ and, on the other hand, if $x = (i, \mathsf{f}, r)$ for any $i \in \mathcal{N}$ and any cumulative fare value $r \in \mathcal{R}$, then $\mathcal{U}(i, \mathsf{f}, r) = \{\emptyset\}$ indicating that the taxi is currently full and is taking a ride from node $i$ to node $k$ with probability $P_{\mathsf{dest}}(i,k)$ (i.e. the probability that a fare picked up in node $i$ will want to go to node $k$).

## G.3 Transition Kernel

Let $\mathcal{P} : \mathcal{X} \times \mathcal{U} \times \mathcal{X} \to [0,1]$ be the transition kernel such that $\mathcal{P}(x_t, u_t, x_{t+1})$ is the probability that state $x_t$ will transition to state $x_{t+1}$ given action $u_t$. Let's consider the different cases.

- First let's look at the e to e transitions for all other state action pairs:

$$\mathcal{P}((i,\mathsf{e},r), u, (j,\mathsf{e},r')) = \begin{cases} 1, & \text{if } r, r' \in \mathcal{R}_f \text{ \& } \{i \in \mathcal{N}, j = x_f\} \vee \{i = x_f, j = x_f\} \\ 0, & \text{otherwise} \end{cases}$$

- Now let's look at the f to f transitions for all other state action pairs:

$$\mathcal{P}((i,\mathsf{f},r), u, (j,\mathsf{f},r')) = 0$$

- Next we will look at the f to e transitions for all other state action pairs:

$$\mathcal{P}((i,\mathsf{f},r), u, (j,\mathsf{e},r')) = \begin{cases} P_{\mathsf{dest}}(i,j) P(E[F(i,j)] + r \geq a_l) & \text{if } r' \in \mathcal{R}_l, r' \geq r \text{ \& } i,j \in \mathcal{N} \\ 0, & \text{otherwise} \end{cases}$$

  where $a_l$ is the lower bound on the interval $\mathcal{R}_l = [a_l, b_l)$

- Finally we look at the e to f transitions (these are all ones where the choices of action dictates the transition probability)

$$\mathcal{P}((i,\mathsf{e},r), u, (j,\mathsf{f},r')) = \begin{cases} 1, & \text{if } u = u_{i \mapsto j} \text{ \& } r = r', r \notin \mathcal{R}_f \\ 0, & \text{otherwise} \end{cases}$$

## G.4 Reward Function

The reward function is a map $R : \mathcal{X} \times \mathcal{U} \times \mathcal{X} \times \to \mathbb{R}$ with $R(x_t, u_t, x_{t+1})$ a random variable such that

- The reward for f to e is

$$R((i,\mathsf{f},r), u, (j,\mathsf{e},r')) = \begin{cases} F(i,j), & \text{if } i,j \in \mathcal{N}, u = \emptyset, r' \geq r, r' \notin \mathcal{R}_f \\ 1000, & \text{if } i,j \in \mathcal{N}, u = \emptyset, r' \geq r, r' \in \mathcal{R}_f \\ 0, & \text{otherwise} \end{cases}$$

  where $F(i,j)$ is a random variable representing the fare from $i$ to $j$. The very large reward in designed to encourage the learning algorithm to determine that the ultimate goal is to reach a cumulative earnings threshold.

- The reward for e to f is

$$R((i,\mathsf{e},r), u, (j,\mathsf{f},r')) = \begin{cases} -t_{\mathsf{seek}}(i,j)/E(i,j)^{-1}, & \text{if } i,j \in \mathcal{N}, u = u_{i \mapsto j}, r = r' \notin \mathcal{R}_f \\ 0, & \text{otherwise} \end{cases}$$

  where $t_{\mathsf{seek}}(i,j)$ is a random variable for the time to travel and find a fare when you go from node $i$ to $j$ under the control action $u_{i \mapsto j}$ and where $E(i,j)$ is a random variable for the earning rate for trips from $i$ to $j$. In practice, we infer the mean values of these quantities from the data.