Simulating Reinforcement Learning

Daniel Changyan Wang

Adviser: Prof. Yoram Singer

1 Introduction and Motivation

Reinforcement learning (RL for short) is a rapidly expanding field of machine learning, which has been important in solving state-of-the-art problems, particularly in games. Despite the rapid developments in the field, there is no framework for generalized testing of reinforcement learning algorithms. Currently, algorithms are only tested in specific games or settings. For RL algorithms designed for a specific setting, this makes sense. For example, Warlop, Lazaric, and Mary's algorithm for recommender systems was tested on real-world ad and movie recommendation data [4]. However, some RL algorithms are meant to be applied in an extremely broad range of settings, yet are only tested in a few specific settings. One example of this is Khadka and Tumer's recent paper introducing Evolutionary Reinforcement Learning (ERL), an algorithm intended to be applied to arbitrary RL tasks [3]. In the paper, Khadka and Tumer demonstrate that ERL performs well in MuJoCo (a physics environment in OpenAI Gym, which is described below). However, MuJoCo, the only environment ERL is demonstrated in, provides a much narrower range of tasks than a general MDP. This is not due to neglect by the authors: it appears that no tool currently exists to test general RL algorithms (such as ERL) within a general MDP. The goal of this project is to create such a tool, enabling better (and more general) testing and evaluation of reinforcement learning algorithms.

2 Introduction to Reinforcement Learning

2.1 The RL Problem

2.1.1 Reward

Reinforcement learning (RL) is a broad category of machine learning, in which the learning agent aims to navigate an environment to maximize some defined reward. In RL problems, the RL agent makes observations of its environment, takes an action, receives a reward, and then repeats this cycle. The goal is to maximize

the total cumulative received reward. The agent can continuously adapt its strategy based on the feedback it is receiving from the environment. For example, if a reinforcement learning agent is playing an arcade game, the reward at a given time point might be the number of points gained/lost in that time point. Or, if a RL agent is in charge of teaching a robot to move, it might get positive reward for moving forward, and get negative reward for crashing.

2.1.2 History and State

RL agents are aiming to maximize their cumulative reward. But, how do they do this? At any point, the RL agent is trying to use the information it has (the history) to decide which action to take next. The "history" is defined as the entire collection of past observations, actions, and rewards. Of course, a RL algorithm rarely utilizes every single piece of the history to make its decision. Instead, from the history, the RL agent only remembers/stores the relevant information it needs to make its future decisions. Unnecessary or outdated information in the history is tossed out. The remaining information, as well as any calculations derived from this information, is what the RL agent keeps track of, and what it uses to decide its actions. This information is called the state of the RL agent, because it is a complete representation of the agent's current decision-making information. (In other words, the agent's state is derived from the history, but once the state is known, the history can be thrown away.) The agent's state is Markov, because given the agent's state, it is not necessary to know the history.

2.1.3 Markov Reward Processes, Discounts, and Return

I assume the reader is familiar with a Markov chain, in which Markov states are traversed, with the probability of transitioning from one state to another expressed using a constant state transition matrix. A Markov reward process (MRP) is a variation on a Markov chain that assigns rewards. At each time step, we receive a reward based on which state we are in. These rewards could be random, but in general we care about the expected value of the reward we receive. Thus, we define a reward function R such that $R(s) = \mathbb{E}[R_t|S_t = s]$, where R_t is the reward received at time t and S_t is the state at time t, so $R(s)$ is the expected reward at a time, given that the current state is s.

Note: We usually discount rewards by a scalar factor $0 \leq \gamma \leq 1$. If we are currently at time t, and time $t + k$ will yield reward R_{t+k}, we devalue this by a factor of γ^k. Instead of valuing R_{t+k} at face value, we value it at $\gamma^k R_{t+k}$. There are many reasons for this, including to account for uncertainty that isn't in the model, and to prevent computing infinite rewards if we have a cyclic Markov process.

Of course, in RL problems we aren't just interested in the reward at a single time step. To evaluate how "good" a state is, we are interested in its return, its total expected discounted reward. The return at a

time-point t is G_t.

$$G_t = R_t + \gamma R_{t+1} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{1}$$

We are usually interested in the expected return based on a given starting state. We call this the state's value, $v(s) = \mathbb{E}[G_t|S_t = s]$. Since this is a Markov process, we can conveniently simplify this equation in terms of the values of other states:

$$v(s) = \mathbb{E}[G_t|S_t = s] = \mathbb{E}[R_t + \gamma v(S_{t+1})|S_t = s] \tag{2}$$

If $\mathcal{S}$ is our set of possible states, P is our state transition matrix, so $P_{s_1 s_2}$ is the probability of moving from s_1 to s_2, then we can write this more explicitly as:

$$v(s) = \mathbb{E}[G_t|S_t = s] = R(s) + \gamma \sum_{s' \in \mathcal{S}} v(s') \tag{3}$$

$$v = R + \gamma P v \tag{4}$$

Equations (3) and (4) are called Bellman equations. From a Bellman equation, we can solve for v in computational time $O(n^3)$, where n is the number of states. This is possible for small MRP's but large ones require other methods (discussed later) to approximate states' values. In many RL algorithms (particularly, those that are not model-free), a large part of an RL agent's goal is to explore the state space of the problem it is trying to solve, and approximately learn the values of the states. This knowledge of the states' values allows the RL agent to act more appropriately.

2.1.4 Markov Decision Process, Actions, and Policy

The overall RL problem is often represented as a Markov decision process (MDP), which is a MRP plus actions. Just like in a MRP, the state transition matrix in a MDP represents the probabilities of moving from state A to state B for every pair of states A and B. However, unlike in a simple Markov chain where the state transition matrix is constant, in a MDP the state transition matrix depends on the agent's actions, which is the agent's behavior function. Thus, depending on which action it chooses, the agent in a Markov decision process is able to influence which states it is likely sent to next. Specifically, each action has a corresponding state transition matrix. Notationally, we denote the set of actions as $\mathcal{A}$. For each $a \in \mathcal{A}$, we have a corresponding state transition matrix P^a, where $P^a_{s_1 s_2}$ is the probability, under action a, of transitioning from s_1 to s_2.

A policy represents how an agent chooses an action. It completely describes an agent's decisions. A

deterministic policy specifies which one action a_s for each state s, so the agent will always take action a_s when in state s. In other words, a deterministic policy is a function π that maps s to a_s. A stochastic policy specifies, for each state, a probability distribution over all possible actions. That is, a stochastic policy is a function π which maps an action a and a state s to the probability that a will be chosen when in state s.

2.1.5 State-value and Action-value Functions

Naturally, we are interested in how much value we will receive from various policies. In RL, the amount of expected value gained from following a particular policy is coded in value functions. The state-value function $v_\pi(s)$ is the expected return starting from a state s, following policy π. It is defined as

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{5}$$

The action-value function $q_\pi(s, a)$ is more specific: it is the expected return starting from a state s, taking action a, and then following policy π. It is defined as

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{6}$$

Similar to the Bellman equation for MRP's, we can use the Bellman equation to break value functions into the immediate reward plus the value functions of other states/actions, as follows:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}[R_t + \gamma v_\pi(S_{t+1})|S_t = s] \tag{7}$$

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}[R_t + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \tag{8}$$

Based on the policy π, we can express $v_\pi(s)$ in terms of the action-value functions of its state:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(a, s) \tag{9}$$

Furthermore, based on the state transition matrix for a given action, we can break down the action-value function in terms of the state-value function!

$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_\pi(s') \tag{10}$$

Combining these two equations, we can get an equation for the state-value function without involving the action-value function, and similarly we can get an equation for the action-value function without involving

the state-value function...

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_\pi(s') \right) \tag{11}$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \left(\sum_{a \in \mathcal{A}} \pi(a|s') q_\pi(a, s') \right) \tag{12}$$

Similar to the calculation of value in an MRP, we can express the calculation of all v_π (that is, $v_\pi(s)$ for all s), in matrix form:

$$v_\pi = \mathcal{R}^\pi + \gamma P^\pi v_\pi$$

$$v_\pi = (I - \gamma P^\pi)^{-1} \mathcal{R}^\pi \tag{13}$$

2.1.6 Finding the Optimal Value Functions

In RL, we're interested in searching for the optimal policy. This usually involves estimating the optimal state-value function and the optimal action-value functions, which are defined as the highest value of the value functions over all policies:

$$v_*(s) = \max_\pi v_\pi(s) \tag{14}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{15}$$

An MDP for which we know the optimal value functions is called "solved". Note that at a given state, the optimal action is always to pick the action which has the highest action-value. Thus, the optimal policy directly corresponds to the set of optimal action-value functions $q_*(s,a)$ for all states s and actions a in the MDP.

We might expect that, similar to how we used the Bellman equations to solve for v_π and q_π, we could solve for q_*. Unfortunately, the Bellman equations for q_* simplify to

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \max_{a'} q_*(s', a') \tag{16}$$

which is non-linear and has no simple solution. Thus, in RL, we will use iterative methods to estimate q_*.

2.1.7 Learning and Planning

Reinforcement learning can often be broken into two problems: learning and planning. Initially, the RL agent has to learn about its environment and learn the behavior of the MDP it finds itself in. As it gains more information, planning becomes more important: the agent must use this information to predict how a policy will perform (prediction) and to choose optimal behavior (control).

2.2 Dynamic Programming

Dynamic programming is used for planning. It can do both prediction (prediction of the performance of a given policy) and control (using present information to choose the optimal policy).

2.2.1 Prediction

We know that $v_\pi = \mathcal{R}^\pi + \gamma P^\pi v_\pi$ for the "true" state-value function $v_p i$, but solving this requires finding a matrix inverse, which is computationally infeasible for large state spaces. Thus, in dynamic programming, to evaluate a policy, we iterate using an approach similar to the Bellman equation. We begin with some estimate v_0 of the state-value function, and let $v_{t+1} = \mathcal{R}^\pi + \gamma P^\pi v_t$. Iterating this way causes v_t to approach v_π. This process is called *synchronous backups*.

2.2.2 Control

Using iteration as above, we obtain an estimate of v_π for the previous policy π. Now we iterate on the policy to improve it. Call the previous policy π_t, so we've estimated v_{π_t}. Call our improved policy π_{t+1}. To improve the policy, we define our next policy as greedy on v_{π_t}. That is, the policy always chooses the action a which maximizes $q(s, a)$ where s is the current state and q is derived from our estimate of v_{π_t}.

2.2.3 Policy Iteration

Now that we have a way to predict how good a policy is, then use that prediction to make a new improved policy, we naturally arrive at a method of iteratively improving the policy. In *policy iteration*, we estimate v_{π_t}, then find π_{t+1} based on v_{π_t}, then estimate $v_{\pi_{t+1}}$, etc. This can be done using any method (not just dynamic programming) of estimating v_{π_t} and any policy improvement algorithm for finding π_{t+1} based on v_{π_t}.

2.2.4 Value Iteration

Policy iteration using dynamic programming is one way to do planning. Another is called *value iteration*. Whereas policy iteration was based on the Bellman equation, value iteration is based on the previously observed dependency between state-value functions, in equation (11):

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_\pi(s') \right)$$

For the optimal value function v_*, we can simplify this further:

$$v_*(s) = \max_{a \in \mathcal{A})} \mathcal{R}_s^a + \gamma P_s^a v_* \tag{17}$$

Based on this intuition, we define our iterative value update as:

$$v_{k+1}(s) = \max_{a \in \mathcal{A})} \mathcal{R}_s^a + \gamma P_s^a v_k \tag{18}$$

Using value iteration, we can estimate v_* without assuming any policy. This can be more effective than policy iteration in complex MDPs.

2.2.5 Efficiency and Asynchronous Dynamic Drogramming

Unfortunately, dynamic programming is not efficient in large MDPs, since each step of iteration requires estimating the state-value function for every single state in the MDP. However, DP's fundamental idea of using previous value functions to infer other value functions is used by many other, more efficient, algorithms. For example, in asynchronous dynamic programming, only some states are updated at each iteration step. Some asynchronous DP algorithms decide which states to back up based on the size of the Bellman update, or based on which states an agent has visited most often.

2.3 Model-Free Methods

Dynamic programming requires us to know the whole MDP (both the expected reward $\mathcal{R}^\pi$ and the state transition matrix $\gamma P^\pi v_t$) in order to predict the performance of a policy in that MDP. In most cases, this is not realistic. This is where model-free methods, such as Monte-Carlo and TD methods, can be useful for estimating the values of states in the MDP. By sampling the MDP many times, model-free methods can learn the values of states in an MDP without prior knowledge about what that MDP looks like. Even when the MDP is known, model-free methods can still be useful in large MDP's, because, unlike DP, they sample the MDP rather than calculating through the whole MDP.

2.3.1 Monte-Carlo Policy Evaluation

Whereas dynamic programming uses the MDP and its knowledge of state transitions to estimate value functions, Monte-Carlo (MC) methods do not assume any knowledge about the MDP underlying the process. Rather, MC bases its estimates of value functions purely on the empirical return. That is, Monte-Carlo methods note every time that a state is visited, and record the subsequent return $G_t = R_t + \gamma R_{t+1} + \dots$ in

a cumulative sum. We then divide the total return received after this state by the number of times we've visited it to find the empirical average return, which is our estimated state-value function for this policy. Essentially, to estimate the state-value functions of a policy, MDP uses the policy many times and averages how well the policy performs at each state.

Two notes: First, in non-stationary problems, it can be useful to keep a *running mean* which gives more weight to more recent results.) Second, note that for continual games which never terminate, Monte-Carlo requires modification to be applicable, since without termination, G_t is never "done" being calculated.

2.3.2 TD Policy Evaluation

In Monte-Carlo prediction, we estimate state-value functions purely based on return, without regard to the subsequent states entered in the process. In TD-learning, we use our estimated state-value functions to help update other state-value functions. The simplest example is TD(0), which updates $V(S_t)$ as follows:

$$V(S_t) \to (1-\alpha)V(S_t) + \alpha\Big(R_{t+1} + \gamma V(S_{t+1})\Big) \tag{19}$$

so the adjustment we make to $V(S_t)$ moves towards our estimate of the value of its successor state, rather than simply the empirical return. In this example, $R_{t+1} + \gamma V(S_{t+1})$ is the *TD target* while $\Big(R_{t+1} + \gamma V(S_{t+1})\Big) - V(S_t)$ is the *TD error*.

2.3.3 MC vs. TD

In general, MC is guaranteed to converge to the answer, and is simple to use. Unlike TD, MC doesn't assume anything about the underlying Markov process, so it is robust in situations where the situation isn't perfectly Markov. However, MC converges more slowly because it relies on brute force for each state, rather than partially sharing information between states as TD does. This slower convergence is more noticeable when the data is noisy. That is, MC is more sensitive to random variances in the sample.

2.3.4 n-step TD Returns and TD(λ) Learning

MC uses purely the empirical returns, so $G_t = R_t + \gamma R_{t+1} + \ldots$. TD(0) uses the next empirical return, and uses the value of the next state as a substitute for all future returns, so $G_t = R_t + \gamma v(S_{t+1})$. A natural idea is to do something in between, such as using the next n empirical returns and then substituting in the value of the next state. Call this return $G_t^{(n)}$. For example, if $n = 2$, we could let

$$G_t^{(2)} = R_t + \gamma R_{t+1} + \gamma^2 v(S_{t+2}) \tag{20}$$

$G_t^{(n)}$ is called the *n-step TD return*. In TD(0) we used the 1-step TD return $G_t^{(1)}$ as the TD target, but we could substitute another n-step TD return and get a similar algorithm. The ∞-step TD return is equivalent to the empirical return.

The λ-return is used to blend all the n-step returns together in TD learning. We define the λ-return as

$$G_t^\lambda = \sum_{i=1}^{\infty} (1-\lambda)\lambda^{i-1} G_t^{(i)} \tag{21}$$

In TD(λ), we define the TD target as G_t^λ. Note that for $\lambda = 0$, if we let $0^0 = 1$, $G_t^\lambda = G_t^{(1)}$ as expected. In general, larger values of λ make the algorithm increasingly similar to MC and less similar to TD(0).

2.3.5 Backward-View TD

The TD learning described above is called *forward-view TD learning* because it updates state-value estimates based on future rewards. If forward-view TD(λ) were programmed literally, it would not update any state-value estimate until the end of the episode. A variation, called *backward-view TD learning*, has the advantage that, unlike MC or forward-view TD, it updates state-value estimates online (on-the-fly) so it does not need complete episodes in order to learn. Each time a reward is received, backward-view TD updates all previous states. Backward-view TD calculates how much a state s would have updated its state-value function, using a value called the *eligibility trace* $E_t(s)$, which decays by a factor of $\gamma\lambda$ at each time step but increases by 1 if the current state is s:

$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbb{1}(S_t = s) \tag{22}$$

When a reward R_t is received, we set our TD target as $R_t + \gamma V(S_{t+1}) - V(s)$, similar to TD(0). Then, we update *every* state by $\alpha E_t(s)\Big(R_t + \gamma V(S_{t+1}) - V(s)\Big)$, so we multiply the TD target by the eligibility trace and then update the state.

Backward-view TD is roughly equivalent to forward-view TD. In fact, if all updates were only applied at the end, backward-view TD would be exactly equivalent to forward-view TD. The difference is that backward-view TD applies the update at each time step, so its final values are slightly different.

2.4 Model-Free On-Policy Control

Similar to how DP prediction was used to iteratively improve policy (by acting greedily with respect to the previous round of estimated state-value functions), we can use our model-free predictions to iteratively improve policy. Note that we can't copy the method exactly, because state-value functions alone cannot provide guidance in a model-free environment, because we don't have full information about how our actions

influence which states we go to. Instead, we must use action-value estimates $Q(s, a)$ which are calculated similarly to $V(s)$ using MC or TD.

Whereas DP control was simple (just act greedily), a purely greedy strategy is a bad choice for model-free control. A DP policy can act greedily with confidence because it knows the entire MDP. In model-free control, the policy must earn plenty of reward *and* continue to explore in order to identify possibly better strategies. Accordingly, acting greedily could prevent proper exploration and lock the agent in a sub-optimal strategy.

2.4.1 ϵ-Greedy and GLIE Policies

ϵ-greedy policy is simply one that chooses the greedy action with probability $1 - \epsilon$ and a random action with probability ϵ. We can viably perform policy iteration using MC estimates of action-value functions and iterating with ϵ-greedy policies.

ϵ-greedy can be a special case of the general class of GLIE (Greedy in the Limit with Infinite Exploration) policies. A policy is GLIE if it explores each state-action pair and infinite number of times, and the policy converges towards the greedy policy. For example, ϵ-greedy policy where $\epsilon = 1/t$ is GLIE. GLIE policies are guaranteed to converge to the greedy policy on the optimal action-value function.

2.4.2 SARSA

Now that we know how to apply MC to iterate on ϵ-greedy policies, we naturally wonder how TD can be used to do the same. Applying TD to action-value functions is quite simple. The action-value analog to TD(0) is called SARSA(0), and updates in SARSA(0) are calculated in the following manner:

$$Q(S_t, A_t) \to Q(S_t, A_t) + \alpha(R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{23}$$

Similar to n-step TD, n-step SARSA defines the n-step Q return as

$$q_t^{(n)} = R_t + \gamma R_{t+1} + \ldots + \gamma^{n-1} R_{t+n-1} + \gamma^n Q(S_{t+n}, A_{t+n}) \tag{24}$$

and the update rule is

$$Q(S_t, A_t) \to Q(S_t, A_t) + \alpha(q_t^{(n)} - Q(S_t, A_t)) \tag{25}$$

Similar to in TD(λ), we define the λ-return as

$$q_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \left(\lambda^{n-1} q_t^{(n)} \right) \tag{26}$$

And SARSA(λ) updates $Q(S_t, A_t)$ towards q_t^λ. Finally, backwards-view SARSA is similar to backwards-view TD, except the eligibility traces are stored for each state-action pair rather than each state. As with all other algorithms described above, the initial values of $Q(S, A)$ can be arbitrary.

For policy iteration with SARSA to converge to the greedy (optimal) policy on the optimal action-value function, the policies not only must be GLIE, but its values α must also satisfy that the sum of α_t does not converge, but the sum of α_t^2 does converge.

2.5 Model-Free Off-Policy Control

The methods described so far (MC control, SARSA control) rely on policy iteration, where we must apply and test a policy before improving it. However, often it is useful to estimate how good a policy is without actually applying that policy. For example, it would be useful to estimate how well an optimal (e.g. greedy) policy would perform, even while we are currently executing an exploratory (e.g. ϵ-greedy) policy. We call the actual policy we're following the *behavior policy* μ, and the policy we're trying to evaluate the *target policy* π. Our first attempt at off-policy control aims to learn state-value functions, is based on simple probability comparisons, and is called *importance sampling*.

2.5.1 Importance Sampling

Suppose that at time t the agent is at state S_t, and the probability that μ picks action A_t is $\mu(S_t, A_t) = 1/2$, and similarly π picks action A_t with probability $\pi(S_t, A_t) = 1$. Then in MC importance sampling we update $V(S_t)$ as follows:

$$V(S_t) \to V(S_t) + \alpha\left(\frac{\pi(S_t, A_t)}{\mu(S_t, A_t)} \cdots \frac{\pi(S_{t+n}, A_{t+n})}{\pi(S_{t+n}, A_{t+n})}\right) G_t \tag{27}$$

So the state-value update is scaled based on how frequent the taken action was in the behavior and target policies. Similarly in TD(0) importance sampling we update $V(S_t)$ as:

$$V(S_t) \to V(S_t) + \alpha\left(\frac{\pi(S_t, A_t)}{\mu(S_t, A_t)}\big(R_t + \gamma V(S_{t+1})\big) - V(S_t)\right) \tag{28}$$

TD(0) importance sampling has much lower variance than MC importance sampling, because it only compares the frequency of the behavior and target policies at one stage, rather than at all stages like MC does.

2.5.2 Q-Learning

Off-policy Q-learning aims to learn action-value functions while off-policy. In particular, if A_{t+1} is chosen according to $\mu(S_{t+1})$ (that is, A_{t+1} is the actual action chosen at time $t+1$) then let A'_{t+1} be the hypothetical action chosen by $\pi(S_{t+1})$. Then, at each time step our update rule is:

$$Q(S_t, A_t) \to Q(S_t, A_t) + \alpha\Big(R_t + \gamma Q(S_{t+1}, A'_{t+1}) - Q(S_t, A_t)\Big) \tag{29}$$

so we update the action-value towards $R_t + \gamma Q(S_{t+1}, A'_{t+1})$. Note that for a good result, you should pick a behavior policy μ that actually explores most or all state-action pairs many times, otherwise many Q-values will not be accurate due to low sample size. A simple and common choice for π is the greedy function. In this case, $\gamma Q(S_{t+1}, A'_{t+1}) = \max_a \gamma Q(S_{t+1}, a)$. Q-learning converges to the correct estimation of the action-value function.

2.6 Value Function Approximation

The model-free methods for prediction and control usually store a value function for every state, or every state-action pair. This becomes impossible for very large Markov processes, such as most board or video games. For MDP's too large to record (or properly estimate) every value function, we instead create a *value function approximator* $\hat{v}(s, w) \approx v(s)$ or $\hat{q}(s, a, w) \approx q(s, a)$. w is the parameter that determines how the value function approximator estimates the value of a state or state-action pair. Often, w is the weights in a linear combination of features or a neural network.

Note that we have been vague about what a value function approximator inputs and outputs, exactly. This is because a value function approximator has several options. It can intake a state s and output a value estimate for it $\hat{v}(s, w)$, intake a state s and action a and output $\hat{q}(s, a, w)$, or input a state s and output a predicted state-action value for every action a: $\big\{\hat{q}(s, a, w)\big\}\forall a \in \mathcal{A}$. Broadly speaking, our aim is to use value function approximators as a substitute for learning every single state-value function. Then, just as we used our learned state-value functions to create an improved policy which we then aim to learn the value functions for, we will use our learned value function approximators to create an improved policy, for which we then aim to learn a new value function approximator, and so on. So, how to train our value function approximators?

2.6.1 Incremental (Online) Methods

We would like to train our value function approximator to be close to the true value function. If we knew the true value function, we could do this with methods such as SGD (if our approximator is a linear combination of features) or backprop (if using a neural network). In practice, we can replace the true value function in these algorithms with our estimated value function from MC or TD. Thus, at each iteration we would train the value function using G_t (MC), $R_t + \gamma\hat{v}(S_{t+1}, w)$ (TD(0)), or G_t^λ (TD(λ)) as our target, measuring our error as $target - \hat{v}(S_t)$.

Thus, using MC and forward view TD to train a value function approximator are theoretically quite simple, as one simply plugs in the MC or TD target into the training algorithm of the value function approximator. Backwards view TD can be more difficult, because the calculation of eligibility traces depends on the particular training algorithm of the value function approximator.

However, for linear combinations of features in SGD, backwards view TD is simple. The eligibility traces are stored as a vector, with length equal to the number of features. The eligibility trace decays by $\gamma\lambda$ each time step and increments by the value of the features themselves at each time step. Then, at each time step the update applied is equal to alpha times the target times the eligibility trace:

$$E_t = \gamma\lambda E_{t-1} + \mathbf{x}(S_t) \tag{30}$$

$$\delta w = \alpha\delta_t E_t \tag{31}$$

where δ_t is the target.

The above methods are for state-value estimation. Action-value estimation is nearly identical. Because we are estimating the value of a particular state and action, the feature vector contains features of both the state and the action. Thus, we call the feature vector $\mathbf{v}(S, A)$ instead of $\mathbf{v}(S)$.

Using these methods to train our value function approximator, we can then use it in our previous policy iteration algorithms such as ϵ-greedy policy improvement.

2.6.2 Convergence of Value Function Approximation

One problem with value function approximation is that it is often not guaranteed to converge to the optimal approximator. In particular, on-policy TD-trained value approximation is not guaranteed to converge for non-linear value approximators, and even linear value approximators are not guaranteed to converge when TD training is used off-policy. However, a small adjustment to TD learning (called Gradient TD) does yield convergence to the optimal approximator, including for non-linear approximators.

Control algorithms are trickier. MC, SARSA, and Q-learning are not guaranteed to converge to the optimal approximation for non-linear approximation functions, but are guaranteed to chatter near the optimum for linear approximators. Q-learning using Gradient TD, though, is guaranteed to converge for linear approximators.

2.6.3 Batch Methods

The above methods are for online (incremental) learning. Alternatively, we can run a policy many times to obtain training data, and then fit the value function approximator to our training data. This is called *batch learning*. For example, for estimating state-value functions, we would obtain a series of pairs $< S_t, G_t >$ from running our policy, and we would randomly sample pairs from our experience with which to train our approximator (e.g. using SGD). In general, training using previous experiences is called *experience replay*. Batch learning is better able to take advantage of limited experiential data, because it can replay an experience multiple times. In addition, by randomly sampling past experiences rather than applying an update immediately after the experience, the input data is less self-correlated (is now i.i.d.), which can improve the stability of learning.

A successful example of this is Deepmind's training an agent to play Atari games. There, they used experience replay on action-value functions, where the agent used ϵ-greedy policy iteration on Q values. The function approximator used was a neural network. The experiences recorded were tuples $< s_t, a_t, r_t, s_{t+1} >$ and these tuples were randomly sampled in order to define Q learning targets based on the previous neural network. Then, the next neural network was defined as the neural network which minimized the MSE between its Q values and the Q learning targets. That is,

$$\mathcal{L}(w_i) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s', a'; w_i)\right)\right] \tag{32}$$

where w^- is the weights in the previous network.

One final note: in the case of linear value function approximators, we can directly solve for the least squares solution (rather than using SGD). This can be faster when the number of features is small. However, this does not mean experience replay is useless for linear approximators: it can be faster in cases where the number of features is large.

2.7 Policy Gradient

So far we have approximated the value of states or state-action pairs directly using a set of parameters. However, in some cases it is better to parametrize the policy itself. This can be useful if the preferred policy

is not greedy, but is instead random. For example, in rock paper scissors, a greedy policy would pick a deterministic option, which could certainly be exploited by opponents.

There are two steps to optimizing a policy. We first choose an objective function, and then move the policy towards the maximum of that objective function.

2.7.1 Policy Objective Functions

When the start state is well defined (such as in many games), we define the objective function as the expected reward based on following this policy. If s_1 is the starting state, θ is the parameter of the policy, and $J_1(\theta)$ is the objective function on these parameters, this is:

$$J_1(\theta) = V^{\pi_\theta}(s_1) \tag{33}$$

This objective function J_1 is the *start value* of the policy. In continuous environments, we can use the *average value* or the *average reward per time step*. If the MDP is stationary, these can be expressed in terms of the stationary distribution of the MDP.

2.7.2 Optimizing Policy Objectives

Given a policy and an objective function, how do we improve the policy with respect to the objective function? In general we wish to change the policy in a direction that increases the objective. For state-value functions it was easy to tell which direction to move the policy in, because given a set of state-values, it was easy to infer how a policy change would affect total return. However, in policy parametrization/optimization problems it is not immediately clear how changing the policy will affect the objective function. Thus, a key component of solving policy parametrization/optimization is to estimate in which direction we should shift the policy to improve it.

2.7.3 Optimizing Policy Objectives: Finite Differences

The simplest method to do this is called *finite differences*. Suppose the parameters are a vector $\theta \in \mathbb{R}^n$. Then for the first dimension of θ (the first parameter) we perturb that parameter by a small amount ϵ, and run the policy to see how the objective function is affected. We repeat this for each dimension of θ, and combine our results to find the direction to move the policy. The method of finite differences is very inefficient and noisy, but it can work, and it has the advantage of working on any arbitrary parametrized policy, even one that is not differentiable.

2.7.4 Optimizing Policy Objectives: Policy Gradient

Finite differences does not rely on the policy being differentiable, but most policy optimization algorithms do, since the gradient of the policy provides a convenient indication of the direction the policy should move in. If our parametrized policy π_θ is perturbed, we are primarily interested in how $\pi_\theta(s,a)$ (the probability a is selected at s) will change for each state-action pair, in proportion with how valuable that state-action pair is, $Q^{\pi_\theta}(s,a)$.

For example, imagine a one-step MDP with an initial state s. Then

$$J(\theta) = \mathbb{E}_{\pi_\theta}(R_t) = \sum_{a \in \mathcal{A}} \pi_\theta(s,a)\mathcal{R}_{s,a} \tag{34}$$

We are interested in the gradient of our objective function, with respect to θ:

$$\nabla_\theta J(\theta) = \nabla_\theta \left(\sum_{a \in \mathcal{A}} \pi_\theta(s,a)\mathcal{R}_{s,a} \right) = \sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(s,a)\mathcal{R}_{s,a} \tag{35}$$

We could stop here, but in many cases (such as with a softmax policy) it is much easier to find the gradient of the log-likelihood, $\nabla_\theta \log \pi_\theta(s,a)$, than the gradient of the actual likelihood $\nabla_\theta \pi_\theta(s,a)$. Conveniently, we note that

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) \tag{36}$$

by basic calculus, so we can plug this into equation (35):

$$\nabla_\theta J(\theta) = \sum_{a \in \mathcal{A}} \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) \mathcal{R}_{s,a} = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \mathcal{R}_{s,a}] \tag{37}$$

Thus, in the one-step MDP, the gradient of the objective function with respect to θ is equal to the expectation of the gradient of the log-likelihood times the expected reward. This turns out (not proven here) to be true in general as well, except with the action-value $Q^{\pi_\theta}(s,a)$ replacing the immediate reward $\mathcal{R}_{s,a}$. Thus, in general we have the following theorem:

If $\pi_\theta(s,a)$ is differentiable for every s,a, then for any objective function (of the three we mentioned above) the gradient is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)] \tag{38}$$

Given the policy gradient, we can use familiar gradient descent to optimize the policy!

2.7.5 Policy Gradient: Monte-Carlo Estimation

Assuming we know $\mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)]$, policy gradient optimization is easy to perform with SGD. For many policies, $\nabla_\theta \log \pi_\theta(s,a)$ is easy to find. The most elusive quantity in the equation is $Q^{\pi_\theta}(s,a)$ or $\mathbb{E}_{\pi_\theta}[Q^{\pi_\theta}(s,a)]$, since this is not easily calculated. We usually substitute some estimate of $Q^{\pi_\theta}(s,a)$. The Monte-Carlo estimate is the easiest. We simply replace $Q^{\pi_\theta}(s,a)$ with the return G_t observed after that state-action pair. G_t is an unbiased estimate of $Q^{\pi_\theta}(s,a)$.

2.7.6 Actor-Critic Methods

Monte-Carlo policy gradient methods can work well, but despite G_t being an unbiased estimate of $Q^{\pi_\theta}(s,a)$, it can have very high variance. Actor-critic methods aim to mitigate this. Rather than using the return as an estimate of $Q^{\pi_\theta}(s,a)$, we estimate $Q^{\pi_\theta}(s,a)$ directly, using policy evaluation methods we've discussed previously, such as TD(λ). The estimator of $Q^{\pi_\theta}(s,a)$ is called the *critic*, and the agent updating the policy is called the *actor*.

2.7.7 Compatible Function Approximation Theorem

In actor-critic methods, by using approximate Q-values $Q^{\pi_\theta}(s,a)$, we are using approximate values of the policy gradient $\nabla_\theta J(\theta)$. Unlike MC estimation, however, actor-critic estimations are generally *not* unbiased. In fact, choosing a poor critic can prevent us from approaching the optimal policy. Fortunately, there is a useful theorem, the *Compatible Function Approximation Theorem*, that identifies sufficient conditions to guarantee that our policy gradient leads to convergence to the optimal policy.

Let w be the parameters of our value function approximator, so $Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$. Then if the gradient w.r.t. w of $Q_w(s,a)$ is equal to the gradient of the log-likelihood of the policy at (s,a)

$$\nabla_w Q_w(s,a) = \nabla_\theta \pi_\theta(s,a) \tag{39}$$

and the parameter w minimizes the MSE compared to the true action-value

$$w = \arg\min_{w' \in W} \mathbb{E}_{\pi_\theta}\left[(Q^{\pi_\theta}(s,a) - Q_{w'}(s,a))^2\right] \tag{40}$$

then the policy gradient using the estimated Q_w is equivalent to the policy gradient using the true action-value

$$\mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)] = \nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)] \tag{41}$$

Thus, so long as we select a compatible action-value approximator, our policy gradient will work just as well as if we were using the true action-value function.

2.7.8 Improving Actor-Critic Methods

There are several ways to improve actor-critic methods. Oftentimes, we replace $Q^{\pi_\theta}(s,a)$ with $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$ where $A^{\pi_\theta}(s,a)$ is called the *advantage function*. This is allowed because it does not affect the expectation of the gradient, so it does not affect the policy gradient, and using the advantage function often produces lower-variance results.

The *natural policy gradient* is a variation on policy gradient which sometimes converges better. The natural policy gradient updates the actor parameters in the direction of the critic parameters.

2.8 Model-Based RL

So far, we have used model-free methods, where we do not aim to learn the distribution of rewards or the state transition matrices. Now we turn to model-based RL, where we aim both to learn the model and to perform policy iteration. In this section we primarily discuss model estimation, because once we have a model we can use our previously-discussed methods to perform policy iteration (or some other planning algorithm).

Formally, we define the model $\mathcal{M} =< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$, where the state space $\mathcal{S}$ and action space $\mathcal{A}$ are known, and we aim to estimate the state transition matrices $\mathcal{P}$ and the expected reward distributions $\mathcal{R}$. Typically we assume independence between the next state transition we receive and the reward we receive. That is,

$$\mathbb{P}[S_{t+1}, R_t | S_t, A_t] = \mathbb{P}[S_{t+1} | S_t, A_t] \mathbb{P}[R_t | S_t, A_t] \tag{42}$$

In model-based RL we are given a set of experiences $< S_1, A_1, R_1, S_2, \ldots, S_T >$. We wish to infer $\mathcal{P}$ and $\mathcal{R}$. If we assume (42), we can solve these problems separately, and they become familiar problems. Inferring $\mathcal{R}$ is a regression problem, in which we aim to use the rewards (s, a, r) to infer the reward functions $\mathcal{R}_s^a$. Inferring the state transition matrix is a density estimation problem, in which we aim to use the input (s_t, a_t, s_{t+1}) to infer the probabilities to advance to each possible subsequent state. We generally pick some loss function and parametrize our estimated model $\mathcal{M}_\eta$, and find parameters η which minimize the loss.

2.8.1 Table Lookup Model

Call our estimated state transition matrix $\hat{\mathcal{P}}$ and our estimated reward function $\hat{\mathcal{R}}$. The simplest attempt at estimating the model is simply to use the average transition frequency and the average reward:

$$\hat{\mathcal{P}}^a_{s,s'} = \frac{1}{N(s,a)} \sum_{i=1}^{T} \mathbb{1}(S_t, A_t, S_{t+1} = s, a, s') \tag{43}$$

$$\hat{\mathcal{R}}^a_s = \frac{1}{N(s,a)} \sum_{i=1}^{T} \mathbb{1}(S_t, A_t = s, a) R_t \tag{44}$$

Alternatively, (particularly if the state transitions and rewards are *not* independent), we could record the tuple $< s_t, a_t, r_t, s_{t+1} >$ for each time step t. Then, to predict the next state and reward from (s, a) we simply take a random tuple $< s, a, \cdot, \cdot >$ which matches $s_t = s, a_t = a$.

2.8.2 Sample-Based Planning

Given that we've estimated a model $\mathcal{M}$ in model-based learning, we might be tempted to use exhaustive search or dynamic programming to find the optimal policy, since these methods take advantage of the fact that we have a model. However, this is often computationally inefficient. *Sample-based planning* applies model-free learning to our model. In sample-based planning, we only use the model to generate samples, which are then fed to any of the model-free learning methods we've discussed before (e.g. SARSA).

2.8.3 Dyna and Dyna-Q

So far, we have discussed estimating the model and planning policy as separate steps. For example, in sample-based planning, we generate the model, and then feed samples from our learned model to our planning algorithm (e.g. MC or SARSA). However, a set of algorithms called Dyna blend model learning and planning. In Dyna-Q, for example, the environment is explored through ϵ-greedy behavior on Q-values, based on the current Q-values. Then, the experience gained from exploring the environment is used *both* to update Q-values as in model-free learning *and* to update the learned model. Between exploration steps, Dyna-Q also refines its Q-value estimates by running sample-based planning on its learned model.

2.8.4 Simulation-Based Search

Once we have an estimated model, we can use our model to plan for the future. *Simulation-based search* incorporates sample-based planning to plan the best current action by exploring the tree starting from the current state. In simulation search, the agent simulates many rounds of experience under a given *simulation*

policy π, and uses these simulations to estimate the state-value functions or action-value functions under π.

In Monte-Carlo tree search, the policy π is simulated many times, and action-value functions for π are estimated using average MC return. Then, the policy is improved using something like ϵ-greedy on the action-value functions previously estimated. That is, π_{t+1} is ϵ-greedy on $\hat{Q}_\pi(s, a)$.

2.9 Summary

Having covered many reinforcement learning algorithms and settings, we now provide a summary of using RL algorithms.

As described above, an RL agent's *state* stores information about its environment, the MDP they are navigating. The RL agent uses this knowledge to choose a policy on how to behave in the MPD. The agent acts according to this policy, and thus receives more information about the environment. Then, using this information, the RL agent updates its state/knowledge, picks a new policy, and so on.

In table 1, various model-free RL agents are summarized based on the different ways in which they accomplish the above. The *State* column refers to what information is stored in the agent's state. *Policy Selection* is how the state is used to choose the policy. *Update Process* is how the agent updates its state upon receiving new information. Model-based RL agents don't fit well into the table because they are slightly more complex, since they have the dual tasks of generating their model and choosing a policy based on their generated model.

3 Related Work

In the literature, I found three attempts to improve the testing and evaluation of RL algorithms: these are OpenAI Gym, OpenAI Baselines, and Google's Deepmind Lab. All three are important tools in testing RL algorithms.

3.1 OpenAI Gym

OpenAI Gym is a testing environment for RL released by OpenAI [2]. It provides an API to implement and test RL agents in a variety of environments. OpenAI Gym offers a wide variety of testing environments, and is often used in testing, but it does not provide tools for randomly generating new environments, which is the goal of this project. However, OpenAI Gym's API serves as an example for how to provide useful, accessible RL testing tools.

Agent	State	Policy Selection	Update Process	Other Notes
Monte-Carlo Policy Iteration	Estimated state-value or action-value functions	ϵ-greedy (or similar) policy on estimated state-value or action-value functions	The estimated state-value function or action-value function equals the average empirical return received after reaching that state, or taking that action at that state	
TD(0) Policy Iteration	Estimated state-value or action-value functions	ϵ-greedy (or similar) policy on estimated state-value or action-value functions	The estimated state-value function or action-value function is updated toward current reward plus the discounted values of the next visited state or state-action pair	
TD(λ) Policy Iteration	Estimated state-value or action-value functions	ϵ-greedy (or similar) policy on estimated state-value or action-value functions	The estimated state-value function or action-value function is a blend of the TD(0) and Monte-Carlo estimates	Higher λ values make TD(λ) more similar to Monte-Carlo
Q-Learning	Estimated action-value functions for target policy π	Picks behavior policy μ based on ϵ-greedy (or similar) policy on current action-value estimates for π. Picks π arbitrarily (often greedy)	Update action-value estimates for π in the same manner as in TD learning (i.e. when calculating the TD target, use the Q-value of the action π, not μ, would have taken)	Q-Learning behaves according to μ but updates action-values based on π
Value Function Approximators	A value function approximator that is being trained (e.g. neural net)	ϵ-greedy (or similar) policy on estimates derived from the approximator-in-training	Train approximator based on data received (could use MC or TD target), e.g. using backpropagation or regression	Batch learning and experience replay can improve learning stability
Monte-Carlo Policy Gradient	Estimated action-values, and a parametrized policy with parameters θ	The policy is already selected, and is part of the state.	Estimated action-values are updated by MC. θ is updated by gradient descent, using the gradient from equation (38).	
Actor-Critic Methods	Estimated action-values, and a parametrized policy with parameters	The policy is already selected, and is part of the state.	Estimated action-values are updated by the critic, which runs the policy many times independently and then uses any standard policy evaluation method to estimate action-values for that policy.	Variants include using the advantage function instead of the action-value, and using natural policy gradient instead of the normal policy gradient.

Table 1: Table summarizing RL methods

3.2 OpenAI Baselines

OpenAI Baselines is a repository for reference implementations of RL algorithms, designed to improve standardization of RL implementations. Baselines is not a direct tool for RL testing, but it provides a valuable source of well-tested reference algorithms. Integration with Baselines algorithms is a potential area of future work for this project.

3.3 Google's Deepmind Lab

Google's Deepmind Lab is a 3D testing environment for RL algorithms [1]. Deepmind Lab specifically offers a 3D environment similar to many games: at any time, the actions available to the agent include movement in any cardinal direction, rotating the agent's viewpoint, and other potential actions such as firing a laser. Of the related work I found, Deepmind Lab is the closest to offering generalizable testing. It allows users to procedurally generate new environments, so users can generate randomized environments if desired. However, these environments continue to be specifically 3D environments where the set of actions is rather specific (and different from, for example, board game environments).

4 Simulator Approach and Implementation

To allow general simulation of RL algorithms in MDP's, we must achieve three tasks: we must generate random MDP's with user specifications, allow users to specify RL agents to be tested, and track the algorithm's performance as it is simulated. In the sections below, I discuss how each of these steps was achieved in the project. The code for the project is found on Github at `https://github.com/dcw3/Thesis`.

5 Simulator: Random MDP Generation

One of our goals is to generate a random MDP according to user specifications, aiming to do so in a way that is "realistic", behaving similarly to MDP's seen in real games or challenges. The user of course must specify a, the number of actions available at each state, and m, the number of states. We then create $a \cdot m$ vectors of length m, each of which represents a vector of state-action transition probabilities, each of which is drawn from some Dirichlet distribution.

5.1 Dirichlet Distribution

Broadly speaking, the best distribution from which to draw transition probabilities is the Dirichlet distribution. The Dirichlet distribution produces probability distributions naturally, and is also adjustable to allow changed means and changed concentrations. The Dirichlet distribution takes as input some vector $\vec{\boldsymbol{\alpha}} = [\alpha_1, \alpha_2, \ldots, \alpha_K]$, and outputs a probability vector $[x_1, x_2, \ldots, x_K]$. The PDF of the Dirichlet distribution is

$$\frac{1}{B(\boldsymbol{\alpha})} \prod_{i=1}^{K} x_i^{\alpha_i - 1}$$

where

$$B(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K} \Gamma(\alpha_i)}{\Gamma(\sum_{i=1}^{K} \alpha_i)}$$

The mean of a Dirichlet distribution with parameters $\alpha_1, \alpha_2, \ldots, \alpha_k$ is $[\alpha_1/\alpha_0, \ldots, \alpha_k/\alpha_0]$. Then, for a given vector v, we can make a particular neighbor w similar to v by drawing w from $Dir(v_1, \ldots, v_k)$, i.e. using a Dirichlet distribution with parameters equal to entries of v.

If $X \sim Dir_k(\beta)$,

$$Var(X_i) = \frac{\alpha_i(\alpha_0 - \alpha_i)}{\alpha_0^2(\alpha_0 + 1)}$$

Then, multiplying all α_i by a factor k will approximately multiply the variance of X_i by $1/k^2$. Then, to increase the "strength" of the correlation, we could draw w from $Dir(kv_1, \ldots, kv_k)$ for some $k > 1$, and we could similarly decrease the strength of correlation by choosing $k < 1$.

If there are multiple neighbors $\{v^1, v^2, \ldots\}$, then we could use their average, call it $\overline{v}$. In addition, rather than setting the parameters directly to the average of $\overline{v}$, we could set them to a mixture of $\overline{v}$ and a uniform vector β (which represents the prior distribution). For example, we could set them to be equal to $r\overline{v}+(1-r)\beta$. Thus, in general the Dirichlet distribution is a flexible and natural way to generate transition probabilities for the MDP.

5.2 Simplest MDP

The simplest MDP generator I built takes as input an α-value or a vector $\boldsymbol{\alpha}$, a number of states, and a number of actions. For each state-action pair, the corresponding transition probability vector is drawn i.i.d. from $Dir(\boldsymbol{\alpha})$. This MDP generator (which is 5 lines of code) serves as a simple example for those wishing to write custom generators, and may also be useful for debugging purposes. Figure 1 shows an example of an MDP generated in such a manner.

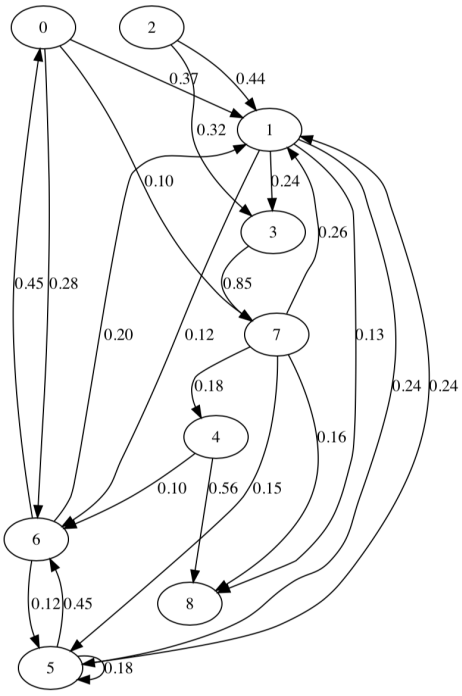

Figure 1: Visualization of transitions generated for one action and ten states, by the simplest MDP generator

5.3 Sparse-Reward MDP's

In this project, I focused on a common sparse-reward setting, where there is no reward in the MDP, except for two terminal states, one of which is high-reward and one of which is low-reward. Further work on this project should aim to extend the range of MDP's generated. As the simulator is highly modular, users are also free to add more MDP generators as is useful.

One of the most unrealistic properties of the simplest MDP method described in Section 5.2 is that, because every transition probability is drawn from the Dirichlet distribution (a continuous distribution over probability vectors), the transition probability between two states is non-zero almost surely (with probability zero). This is very unrealistic for most games and applications of RL. Intuitively, the best way to fix this is to select a set of "successor" states for each state-action pair: the set of k states which can be reached from this state-action pair, while all other states have probability zero. Then, the transition probabilities to the successor states would be drawn from a Dirichlet distribution of length k.

In general, we would like to see that a generated MDP's states are all reachable from the origin state. Otherwise, those states are useless. Unfortunately, some methods of selecting successor states do not guarantee this property. A small example is the following. Suppose there are 4 states (numbered 1, 2, 3, and 4), each with one action, and each state has only one successor state. State 1 is succeeded by state 2, and states 2, 3, and 4 are succeeded by state 1. If state 1 is the origin state, then states 3 and 4 are never reachable.

Furthermore, some methods of selecting successor states do not guarantee that the MDP is even weakly connected. For example, as in the previous example, if there are 4 states and one action, and states 1 and 2 succeed each other and states 3 and 4 succeed each other, then the MDP can essentially be broken into two unrelated MDP's, which is unlikely to occur in games or challenges faced by real RL agents. Thus, when generating MDP's, we must be careful about how to select successor states for each state.

To solve the problem of reachability and connectedness, I viewed generating an MDP as an iterative process. First, we choose a state s whose transition probabilities are not yet defined. Then, for each action a, we choose the states that are reachable from s in one step under action a. Then, from within those destination states, we define a transition probability vector with desired entropy. Then, having chosen the transition probability vectors for all actions from s, we choose a new state s and begin again.

Below, I elaborate on details in the iterative process of generating an MDP. The simplest part is choosing a transition probability vector once we have chosen the destination states. If there are k destination states, we use a k-dimensional symmetric Dirichlet distribution function, with the concentration parameter α. Higher values of α correspond to higher entropy, and this can be specified from the user. For example, for any state-action pair, α could be drawn from a normal distribution with a certain mean, or α could be constant.

The next state for which to choose probabilities could be chosen either from any of the destination states, or from any of the states whose transitions have not already been decided.

The most interesting choice is choosing the destination states of each state-action pair. There are countless properties users might be interested in testing here, so it is hard to account for all possible MDP's the user may wish to create. In my implementation, we define recurrent, terminal, and "normal" successors. Using these concepts, we offer the following properties:

- How well-structured is the MDP? At one extreme, all actions from a given state could have the same successor states, just with different probabilities. At the opposite extreme, the successor states of different actions could be completely independent though identically distributed (i.i.d.). To implement this, I ask users to specify two functions (which can be constant functions). One function, call it f, takes as input a state number and outputs the number of states that are candidates to be successor states to this state. Then, the other function g takes as input a state-action pair, and outputs the number of states that are candidates to be successor states to that state-action pair. If s, a are the state and action, then the ratio $r = f(s)/g(s, a)$ represents how well-structured the MDP is at state s. $r = 1$ implies the successor states for different actions are all the same. When m is the total number of states, $f(s) = m$ implies that the successor states of different actions are chosen independently from all possible states.

- How frequently do the states feed into the terminal states, and how recurrent is the MDP? For example, in the case of feeding into terminal states, we might contrast an environment in which all states have some chance of having a terminal state as a successor, versus an environment such as a long and winding maze, where only a few states far from the origin have a chance of being adjacent to a terminal state (an exit). The two characteristics of terminal-feeding and recurrence are defined very similarly to the characteristic of well-structuredness discussed in the first bullet point above. For terminal-feeding, the user is asked to specify two functions, one of which specifies the number of terminal connections for each state, and the other of which specifies the number of terminal connections for each state-action pair. Similarly the user specifies two functions to determine how many recurrent connections are from each state and each state-action pair.

6 Simulator: Agent Specification

To specify an agent, the user must write a Python class, inheriting the BaseAgent class provided, that implements several functions. I implemented a Q-learning and a SARSA agent as examples for users. The `reset` function can be called by the user when they wish to reset an agent, forgetting what it has learned. The `begin_episode` and `end_episode` functions are called at the beginning and at the end of each training episode, respectively. They may be used by the agent to synthesize information gained during an episode. For example, a forwards-view TD agent might apply updates in the `begin_episode` or `end_episode` function. Finally, the agent class must implement `step` and `step_eval` functions. Both functions take as a numerical reward, a state number, and an integer time (which increments by one for each step, and resets to zero at the beginning of each episode). The difference is that the `step` function is called during training, while the `step_eval` function is called during evaluation. The `step` function takes the given information and returns the action the agent wishes to take, assuming it is training (so this choice of action might represent the agent's desire to learn as well as its desire to earn reward). The `step_eval` function returns the action which the agent expects will maximize its reward during the evaluation phase, and the agent does not learn when calling `step_eval`. As discussed below, `step_eval` allows the simulator to evaluate the agent's progress and learning as the agent is being trained.

7 Simulator: Simulations and Performance Tracking

Currently, only simulations of terminal MDP's are supported. To train an agent in an MDP, the user instantiates an instance of the `TerminalSimulator` class with the MDP and agent as attributes, and then

calls the `simulate` function with the desired number of episodes (call it n) and the initial state. The function returns a vector of length n containing the amount of reward accumulated over each episode. In addition to viewing the actual reward received in each episode by the agent, users can evaluate the policy determined by the agent's state at each time step, described below. At first glance, it may appear that This is for two reasons.

At any given point in the agent's training, the agent stores some knowledge about the MDP it is navigating. For example, a Q-learning agent stores its estimates of Q-values. The agent must be able to use this knowledge to decide both how to behave when exploring the MDP, and how to behave when trying to maximize reward. Then, for a given agent which is trying to maximize its reward, the state of knowledge it holds about the MDP determines a policy. For example, in a Q-learning agent the determined policy is the greedy policy on Q-values, selecting the action with the highest estimated Q-value. Then, at various points in the agent's training, we would like to estimate the expected reward of the policy determined by the agent's knowledge. In other words, we would like to see how well an agent can perform at various points in its training.

To do this, users can call the `simulate_eval` function. This function is identical to the `simulate` function, except that between each episode, the policy is tested (using the `step_eval` function rather than `step`) for a number of iterations (user-specified) and the average evaluation reward in each episode is recorded, then returned alongside the actual reward received. Figure 2 shows an example of an MDP generated in such a manner.

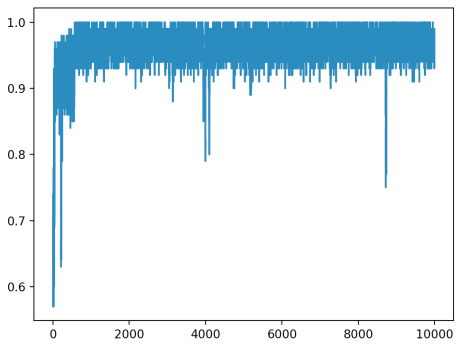

Figure 2: Visualization of the evaluated policy value at each episode of a Q-learning agent in a sparse-reward MDP

8 Conclusion and Future Work

The simulator is currently still in progress. While showing some promise, it requires more work in order to be truly useful to the broader RL research community, which is our goal. One important area for improvement is documentation, about both how to use the simulator and how to contribute to it. In the long run, as an open-source tool, we hope that many users will contribute to the code as well, making documentation of the project particularly important. Furthermore, even non-contributing users will require documentation and examples for how to write custom MDP generators and how to specify custom agents.

Another area in need of improvement is the validation of the custom MDP generators I wrote. As discussed, we would like to generate MDP's that inform researchers about how they should expect their algorithm to perform in the real world. Therefore, we would like to generate MDP's similar to real-world ones. Further study is needed to better characterize what a "realistic" MDP means. Further experimentation correlating algorithms' performance in generated MDP's and real tasks would also help illuminate the actual usefulness of the MDP generators. On a theoretical level, it could be valuable to further study the properties of various rules for randomly generating graphs and Markov processes as well.

Finally, some small quality-of-life improvements could improve the simulator. For example, the current visualization function for MDP's only shows transitions for one action. The visualization function should be improved to overlay multiple actions' transitions, using different colors for clarity.

Hopefully, the improvements described would encourage adoption of this simulator for testing, improving the generalizability of testing RL algorithms and contributing to our understanding of how these algorithms succeed.

References

[1] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab. *CoRR*, abs/1612.03801, 2016.

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

[3] Shauharda Khadka and Kagan Tumer. Evolution-guided policy gradient in reinforcement learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 1188–1200. Curran Associates, Inc., 2018.

[4] Romain Warlop, Alessandro Lazaric, and Jérémie Mary. Fighting boredom in recommender systems with linear reinforcement learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 1757–1768. Curran Associates, Inc., 2018.