A Survey of Fundamental Reinforcement Learning Algorithms

Jacob Pettit

Florida State University

April 27, 2019

1 Introduction

In this Practicum project, my goal was to survey a set of foundational reinforcement learning (RL) algorithms and build up an understanding of each. The algorithms covered are Temporal Difference Learning, Dynamic Programming, and Monte Carlo. Each algorithm is used to solve a different problem or environment. These algorithms are some of the building blocks for understanding many modern RL algorithms and systems, and as such, understanding them is a prerequisite to being able to keep apace of the latest advances in the field.

2 Background

2.1 Overview of Reinforcement Learning Systems

In reinforcement learning, we want the agent to learn what to do in a variety of situations. The goal is to learn how to map from a situation to an action. We say that the best action an agent can take is the one that maximizes its reward.

The agent needs to be able to sense at least part of the state of its environment; it needs some information to make decisions based off of. Actions taken by the agent also must affect the state of its environment. This way the agent will experience the consequences of its actions through the reward it receives.

We require that our agent has a goal that relates to the state of the environment. That is to say, there must be some terminal state that the agent is working towards. If we were training an agent to play tic-tac-toe, it would work towards winning, where winning would be formulated as a terminal state with three of the agents pieces in a row.

An agents goal is to maximize the reward it collects. To do this, the agent must take actions which it knows from past experience yield reward. However, it also must find new reward giving actions, because of the possibility that one of these new actions is better than the previously discovered actions. This is to say that the agent must continuously evaluate new actions with the goal of finding a better option than what it currently believes is best, and it must progressively prefer the actions that appear best.

There are four key sub-elements of an RL system. These are:

- Policy
- Reward signal
- Value function
- Model of the environment

The policy determines our agents behavior. It gives us a mapping from environmental state to a probability distribution over possible actions in that state. We can consider the action the policy believes is best to be the one with highest probability.

We use the reward signal to define the goal in our problem. At each time step, the environment gives a singular reward value to the agent. The agent wants to maximize the reward it collects over a long period of time. When we look at all of the reward an agent receives after taking action A in state S, we call this the return.

The reward signal is a short-term value because it is provided once at each time step. A long-term evaluation of what is good comes from the value function. We define the value of a state to be the expected total reward an agent can collect through the future, when starting from that state. The value function accounts for the most likely following states and the available reward in those subsequent states and uses that information to produce an approximation of the long-term desirability of a state.

We use rewards to make decisions about which actions to take, and values are used for judgment. We want our agent to pick actions that give the highest value, rather than the highest reward. This is because actions with higher value are better actions over the long run. It is significantly easier for us to calculate rewards than values. Rewards are outright given to us by the environment every time step, while we have to estimate values based upon the agents past experiences and past observations.

A model of the environment must either mimic the behavior of the environment or allow us to infer how the environment will behave. For example, given a current state S_t and action A_S, an environment model could be able to predict the next state S_{t+1} and reward. Models are used for action planning. Environmental models are not always necessary for learning a value function or policy.

2.2 *k*-Armed Bandits

A *k*-Armed Bandit problem is one where an agent repeatedly has to choose from *k* different actions. The reward for each action is selected from a stable, unchanging probability distribution. The distribution the reward is picked from depends on the action taken. The goal is to maximize the expected total reward over an arbitrary time period. The *k*-armed bandit problem can be compared to a slot machine with *k* arms instead of one. Each action picked is like choosing to pull on one of the slot machine's levers, and the rewards are like payoff for getting the jackpot.

k-Armed Bandits provide a useful framework for building up to the full reinforcement learning problem. When we solve a *k*-armed bandit problem, we are trying to estimate the true action-values for a set of actions so that we can choose the optimal action at each time step. An action-value tells us how good an action is over the long run. There are three levels of complexity to *k*-armed bandit problems. Stationary cases, nonstationary cases, and associative cases.

In a stationary *k*-armed bandit problem, the reward generating probability distribution and therefore the action-values are unchanging. We can use a simple average of rewards received when taking action A to figure out the action-value for that action.

During a nonstationary problem, the action-values and reward generating distribution change. When we are dealing with such a problem, it makes more sense for us to give more

weight to more recent rewards than to rewards from further into the past. One way to do this is to use an incremental update rule with a constant step-size parameter:

$$Q_{n+1} \doteq Q_n + \alpha * [R_n - Q_n] \tag{2.1}$$

The step-size parameter, $\alpha \in (0, 1]$ is constant. This forces Q_{n+1} to be a weighted average of past rewards.

An associative case of a k-armed bandit is when the true action-values change as an observable trait of the environment changes. To solve this problem, we must learn a policy which maps from the situation of the environment to the optimal action.

The next step up in complexity is to consider an associative case where the actions don't only affect the current state, but also affect future states. These actions affect not only the immediate reward, but also all future reward. When we work on this type of problem, this is considered to be a full reinforcement learning problem. To quickly reiterate, the full RL problem is:

- Associative
- Nonstationary
- Allows current actions to affect the future

2.3 Finite Markov Decision Processes

A Markov Decision Process (MDP) is a formulation for sequential decision making. Actions influence current and future rewards, and actions influence future states. Therefore, MDPs involve dealing with delayed reward and the requirement to be able to trade-off between the immediate and delayed rewards. Where in k-armed bandits we estimated $q_*(a)$, the action value of each action a, in MDPs we need to estimate $q_*(s, a)$, the value of each action a in state s. Alternatively, we can estimate $v_*(s)$, which is the long-term value of each state, given that optimal action choices are always made. MDPs are a mathematically idealized form of the full RL problem, and as such, precise theoretical statements can be made about them.

2.3.1 Interface Between Agent and Environment

We intend an MDP to be a simple and direct way to frame the problem of learning from interaction and experience to achieve a goal. We call the thing that learns and makes decisions about which actions to take, the agent. Everything external to the agent, so all of the stuff it interacts with, is called the environment. The agent and the environment interact continuously. The agent picks new actions at each time step and the environment responds to the actions and gives new situations to the agent. The agent also receives rewards from the environment for each action it takes. The agent aims to maximize its rewards over time by choosing actions that yield more reward.

Interaction between the agent and the environment occurs at discrete time steps, i.e. $t = 0, 1, 2, 3, 4, \ldots$. At each time step, the agent receives some representation of the state of the environment, $S_t \in S$ and chooses an action $A_t \in A(S_t)$. One time step later, the agent receives a reward, $R_{t+1} \in R \subset \mathbb{R}$. Then, the agent is in a new state: S_{t+1}. The interaction of the MDP and the agent yield a sequence that looks like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{2.2}$$

In a finite MDP, states, actions and rewards all have finite numbers of elements and so all have well-defined probability distributions. The distribution over next state and reward in next state depends only on the previous state and action. For particular values of the random variables $s' \in S, r \in R$, there exists a probability of the values occurring at time t, given some values from the previous state and action.

$$p(s', r|s, a) \doteq Pr[S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a] \quad \forall s', s \in S; r \in R; a \in A(s) \tag{2.3}$$

The probabilities of the four argument function p completely describe the dynamics of the MDP. Using this function, we can calculate anything we want about the environment. We can calculate the state-transition probabilities, the expected rewards for state-action pairs, or the expected rewards for state, action, next state triplets.

The framework of an MDP is largely flexible and can be applied to a great variety of different

problems. We do not require that time steps occur in real time. Instead, they can occur at arbitrary intervals or can occur every time an action is taken. An action can be any choice that we want the agent to learn how to make and a state can be any information the agent can possibly know that may be useful in making those choices. The general rule for drawing the boundary between environment and agent is that if something cannot be arbitrarily changed by the agent, it is external to it and so must be part of the environment.

The MDP framework is a large abstraction of the problem of goal-oriented learning from experience and interaction. It suggests that the problem of learning goal-directed behavior can be reduced to three signals between an agent and its environment: state signal, action signal, and reward signal. The representation of states and actions between different tasks can have an outsize impact on performance, and at present, the choice of how to best represent these things is more of an art than a science.

2.3.2 Reward and Goal

In RL, the reward determines the goal of the agent. Per time step, the reward is a single number, $R_t \in \mathbb{R}$. The agent strives to maximize the total reward it receives. This means that the agent must be able to prioritize total long-term reward over immediate reward. This idea can be clearly stated as the reward hypothesis:

Every time we talk about a goal, this can be thought of as maximizing the expected value of the cumulative sum of the reward.

The fact that we use a reward signal to formalize the idea of a goal is one of the most distinct features of RL. Perhaps surprisingly, the use of reward signals to define goals has been largely flexible. When making a robot escape a maze, researchers have given it a reward of -1 for every time step before escape. For training RL to play a game, such as Go, chess, or checkers, natural rewards arise as -1 for a loss, 1 for a win, and 0 for all nonterminal or drawing board positions. In each of the examples, the agent will always learn to maximize the reward. In escaping the maze, it'll learn to escape as quickly as possible. When playing the game, it'll learn

to take actions that give it the highest probability of winning. When we want the agent to learn to perform a specific goal, we must provide the reward in such a way that when it maximizes the reward, the resultant behavior will lead it to accomplish the desired goals. The reward signal should not be used to indicate *how* we want the agent to do something, only *what* we want it to do.

2.3.3 Episodes and Returns

When we have a sequence of rewards: $R_t, R_{t+1}, R_{t+2}, \ldots$, we want the agent to learn to maximize some function the whole sequence, called the return, instead of any one part of the sequence. The return can be written, in the simplest case, as the sum of all rewards received by the agent:

$$G_t \doteq R_t + R_{t+1} + R_{t+2} + \cdots + R_T \tag{2.4}$$

Where t_n are time steps during the task, and T is the terminal time step. Formulating the return like this works well for tasks where there exists a natural idea of a terminal time step. Examples of tasks having this characteristic would be something like playing a game, where the terminal time step occurs when there is a terminal game state, or escaping a maze, where the terminal time step happens when the agent has successfully escaped the maze. Tasks with natural 'episodes' like this are called episodic tasks.

Frequently, we want the agent to solve a task that does not naturally break up into episodes. We call tasks like this 'continuing' tasks. An agent trying to solve a continuing task using the reward function outlined above would likely not learn very well. This is because the final time step in a continuing task is $T = \infty$ and as such, the reward would also be infinite. To deal with infinite time steps, we formulate a discounted return function. We discount rewards with parameter γ, called the discount rate, where $0 \leq \gamma \leq 1$. With this approach, the agent attempts to choose actions that maximize the sum of discounted rewards it receives over the future. Formulation of discounted reward:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.5}$$

The discount rate allows us to set the current value of rewards into the future. Rewards are worth γ^{k-1} times what their worth would be if no discounting were being used, where k is how many time steps into the future a reward is. When $\gamma < 1$, the return function above has a finite value as long as the sequence of rewards is bounded instead of being allowed to progress infinitely. When $\gamma = 0$, the agent does not care at all about future rewards and so only seeks to maximize immediate reward. As we increase γ towards 1, the agent cares more and more about taking future rewards into account when making decisions.

2.3.4 Value Functions and Policies

The vast majority of RL algorithms involve approximating value functions. We approximate the value of how good it is to be in a given state, or how good it is to take a given action in a given state. Goodness is defined in terms of expected return. Future expected rewards depend on the actions the agent takes in the present.

We define value functions with respect to policies. Policies map from a state to a probability distribution over actions that could be taken in that state. If we call a policy π and we have an agent following that policy, then at a time step t, $\pi(a|s)$ is the probability of choosing an action $a \in A(s)$ given a state $s \in S$. $\pi(a|s)$ is an ordinary probability density function over $a \in A(s)$ given $s \in S$. Over time, as the agent gains more experience, RL methods declare how the agent's policy should be changed to continually maximize reward.

The value of a state s under policy π is denoted as $v_\pi(s)$. It is formulated as the expected return when the agent starts in state s and thereafter follows policy π. We can define it as:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{k+t+1}|S_t = s] \quad \forall s \in S \tag{2.6}$$

v_π is called the state-value function for policy π. $\mathbb{E}_\pi$ is the expected return when following policy π.

Let's now look at action values. We can declare the value of taking action a in state s under a policy π as $\pi(q_\pi(s, a))$. This value is the expected return when we start at state s, take action a,

and after that action follow the policy π.

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{k+t+1}|S_t = s, A_t = a] \tag{2.7}$$

q_π is the action-value function for policy π. Both the state and action value functions can be estimated from experience. We could use a Monte Carlo method and maintain averages over large random samples of real returns to generate approximations of the values of each state and action. If there are too many states, we may not be able to use a Monte Carlo method and instead could approximate the state and action value functions as parameterized functions and adjust a parameterized function approximator to match observed returns.

2.3.5 Optimality in Policies and Value Functions

An interesting and fundamental property of value functions and of dynamic programming is that they satisfy a set of recursive relationships. Given a policy π and a state s, the below condition will hold for both the current value of s and the values of every possible subsequent state.

$$\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\
v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
v_\pi(s) &= \textstyle\sum_a \pi(a,s) \sum_s' \sum_r p(s',r|s,a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] \\
v_\pi(s) &= \textstyle\sum_a \pi(a,s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \quad \forall s \in S
\end{aligned} \tag{2.8}$$

All actions are taken from $A(s)$, the set of possible actions in state s, and all states are taken from S or S^+ if it is an episodic task. Rewards are chosen from R, the set of possible rewards.

Equation (2.8) is the Bellman equation for $v_\pi(s)$. It demonstrates a relationship between the value of a current state and values of subsequent states. The Bellman equation looks from a current state into the future of all possible subsequent states, it averages all of those states and the possible actions in each state, and weights each state-action pair by its probability of happening. v_π is the unique solution to its particular Bellman equation. The Bellman equation gives us a basis for a

wide variety of different methods to find v_π.

Since we are dealing with finite MDPs, we can clearly define an optimal policy.

Policy π is said to be better than or equivalent to a policy π' if its expected return across all states is greater than or equal to the expected return of π' over all states.

Writing this idea mathematically:

$$\pi \geq \pi' \quad \text{if and only if } v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in S$$

This is an optimal policy. It is possible for more than one optimal policy to exist, but, we still denote all of the optimal policies by π_*. All of the optimal policies share the same state-value function, called the optimal state-value function. This optimal state-value function is denoted by v_* and is defined by:

$$v_*(s) \doteq max_\pi v_\pi(s) \quad \forall s \in S \tag{2.9}$$

Optimal policies will also share an optimal action-value function, denoted q_* and defined by:

$$q_*(s,a) \doteq max_\pi q_\pi(s,a) \quad \forall s \in S, a \in A \tag{2.10}$$

For a state-action pair (s,a) this function gives us the return we expect when taking action a in state s and afterwards following an optimal policy. So, we can write q_* in terms of v_*:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \tag{2.11}$$

In reality, an agent almost never learns an optimal policy. Normally, we are lacking computational power to compute optimal policies for interesting problems. Even given a complete and accurate model of the environment and its dynamics, we still often cannot compute the optimal policy simply because the state-space is too large. The way that we pose the reinforcement learning problem requires that we settle for approximations. However, it also gives opportunity to

develop some very good approximations. For an example, when we are learning to approximate optimal behavior, there may be some states of the environment that are so extremely unlikely to occur that poor decision making in those few states has a very small impact on the overall amount of reward the agent receives. The online feature of RL makes it possible to put in more work to develop good value approximations for states that occur frequently and put in less work to get good approximations of states that occur rarely. This is a distinguishing property of RL that separates it from other methods to approximately solve MDPs.

3 Algorithms

The algorithms investigated are Dynamic Programming Policy Iteration, Monte Carlo Policy Iteration, and Temporal-Difference Learning Value Estimation. First, we'll look at the Dynamic Programming approach for Policy Iteration and discuss some of its strengths and weaknesses, then, we'll do the same with Monte Carlo Policy Iteration and finally to Temporal-Difference Learning Value Estimation.

3.1 Dynamic Programming Policy Iteration

Dynamic Programming (DP) requires a perfect model of the environment as a Markov Decision Process, but, once it has this, it can compute an optimal policy for that environment. Unfortunately, due to this requirement of a perfect model of the environment, DP often cannot actually be used in reinforcement learning. Frequently, we do not have a perfect model of the environment and so instead must opt for a method that can handle this. However, the theoretical background of dynamic programming is still important. Optimal policies can be obtained once we've found the optimal value functions that satisfy the Bellman optimality equations.

3.1.1 Policy Evaluation

We compute the state-value function, v_π for policy π by using policy evaluation. Policy evaluation is also referred to as the prediction problem. From section 2.3.5 above:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s]$$
$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s]$$
$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_{t+1}] \tag{3.1}$$
$$v_\pi(s) = \sum_a \pi(a,s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')] \tag{3.2}$$

$\pi(a|s)$ is the probability of taking an action a given a state s following policy π. Our expected returns and values in the above equations are all subscripted by π to indicate that they are what we expect only when the policy is being followed. So, if we break policy, these equations do not hold. When we completely know all environmental dynamics, (3.2) becomes a system of $|S|$ linear equations in $|S|$ unknowns. The unknown components are the values of each state $s \in S$. In this case, an iterative method is best for computing the solutions to the linear system. Our initial value approximation does not matter, so we choose it arbitrarily, and each following approximation is obtained by using the Bellman equation for v_π (eqn. 3.2) as an update rule.

$$v_{k+1} \doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s]$$
$$v_{k+1} = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')] \tag{3.3}$$

$\forall \quad s \in S$

The Bellman equation guarantees that $v_k = v_\pi$ in this case, as $k \leftarrow \infty$. The algorithm described here is called *iterative policy evaluation.*

Iterative policy evaluation applies the same operation to each state: replace the old value of state s with a new value, which is obtained from the old values and from the current expected reward along all possible one-step transitions under the policy that we are evaluating. This is called an expected update operation. Every time there is an iteration of the iterative policy evaluation, it updates each states value one time to produce a new approximation of the value function. Every single update that is done in a dynamic programming algorithm is an expected update since we're updating the value approximation based on a reward expectation over all possible next state rather than on a real, sampled next state.

3.1.2 Policy Improvement

When we calculate the value function for a certain policy, this can help to find better policies. When following a value function v_π and following a deterministic policy π, we'd like to see if we should change the policy when in state s so that it chooses some action $a \neq \pi(s)$. We already know the value of the current policy, so the object is to figure out if it is advantageous to change the policy. We can figure this out by picking a when in s and afterwards following π. The value of this is:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \tag{3.4}$$
$$q_\pi(s,a) = \textstyle\sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

The key criterion to determine whether it is better to change the policy is to see whether the value determined here is greater or less than $V_\pi(s)$. If it is greater, then we can expect that it will be better to pick a every time we are in s and that the updated policy would overall be superior.

The new policy being better than the old is a special case of the *policy improvement theorem*. Let's make π and π' any pair of deterministic policies such that for all $s \in S$:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{3.5}$$

Policy π' has to be at least as good as π, it must get greater or equal expected return as π for all states.

$$v_{\pi'}(s) \geq v_\pi(s) \tag{3.6}$$

This is the policy improvement theorem. We can extend it to look at all states and all actions that could possibly be taken. At each state, we can pick the action which looks best according to $q_\pi(s,a)$. This produces a greedy policy, π'.

$$\pi'(s) \doteq argmax_a q_\pi(s,a)$$
$$\pi'(s) = argmax_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \tag{3.7}$$
$$\pi'(s) = argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

The greedy policy chooses the action with the best short-term reward, according to the value function v_π. This meets the conditions of the policy improvement theorem by construction, so we know π' is either as good as or better than π. The process of making a new policy that is better than the old policy, by making it greedy with respect to the current value function, is called policy improvement.

If the new, greedy, policy is as good as, but not better, than the old policy, then $v_\pi = v_{\pi'}$ and it follows from equation 3.7 that for all states:

$$v_{\pi'}(s) = max_a \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) | S_t = s, A_t = a]$$
$$v_{\pi'} = max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi'}(s')]$$

This matches up with the original Bellman equation, (3.2). This tells us that $v_{\pi'}$ must be v_* and that therefore π and π' are optimal policies. Policy improvement must always give a better policy except for when the original policy is an optimal one.

3.1.3 Policy Iteration

Once we've improved policy π with v_π to give a new and better policy π', we can calculate a new value function, $v_{\pi'}$, and again improve to get an even better policy π''. In this way we can obtain a set of consistently improving policies and value functions. The trajectory of policy iteration would look like this:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

Each E represents a policy evaluation step and each I represents a policy improvement step. Unless the policy is already optimal, each new policy will be an improvement over the old one. A finite MDP has a finite number of policies, so the policy iteration process must converge within a finite number of iterations.

In policy iteration, each policy evaluation step is started with the value function from the previous policy. Often, this increases the speed of convergence.

3.2 Monte Carlo Policy Iteration

Everything discussed in the previous section covering deterministic policy evaluation, improvement, and iteration extends to probabilistic policies as well.

In contrast to the requirements of dynamic programming, Monte Carlo methods do not require perfect knowledge of the environment. In fact, they need only experience in order to be able to learn. This is important because we do not need any knowledge of the dynamics or mechanics of our environment in order to learn optimal behaviors. It becomes particularly helpful when learning from simulated experience, because although we do still need a model of the environment, we need only to use the model to produce transition probabilities for samples, not for all possible transitions.

With Monte Carlo methods, we sample and average returns over state-action pairs. This is similar to the k-armed bandit methods which sampled and averaged rewards for each action. Now, the significant difference is that we are working with the full RL problem; we allow actions to impact future states and future rewards. With this method, whatever problem we are working on becomes nonstationary because we are continuously learning to make different action choices.

We must adapt the idea of policy iteration to handle the nonstationarity of the problem, we'll use samples from our MDP to learn the value function. This contrasts dynamic programming, which directly computed the value function using an update rule and policy iteration. Policy iteration is extended from dynamic programming to Monte Carlo, here we use sample experience to learn the policy π, the state value function $v_\pi(s)$, and the state-action value function $q_\pi(a, s)$.

3.2.1 Monte Carlo Prediction

In Monte Carlo Prediction, we'll use the Monte Carlo method to learn the state-action value function for a given policy. In order to estimate the state-action values from experience, we will average the observed returns for a state following every time that we are in that state. As the number of times visiting each state goes to infinity, that average will converge to the true expected return for each state.

If we have a goal of estimating $v_\pi(s)$, given a set of episodes that we have collected by following π and moving through the set of states, this can be achieved using either the method of first visit Monte Carlo or every visit Monte Carlo. First visit Monte Carlo estimates the value of state s as an average of the return after the first visit to state s, while every visit Monte Carlo estimates the value by averaging the return of state s after each visit made to the state. Both first and every visit Monte Carlo converge to $v_\pi(s)$ as the number of visits goes to infinity.

3.2.2 Estimating Action Values with Monte Carlo

In the case that we are lacking a model of the environment, it is better to estimate action values and use those to form a policy, rather than state values. When we do have a model, state values alone are sufficient to form a policy. To form a policy with state values, we just look one state ahead and choose the one with the highest value. One of the main goals of Monte Carlo is to approximate q_*. In order to achieve this, we'll use policy evaluation for action values.

In policy evaluation for action values, we aim to estimate $q_\pi(s, a)$, the expected return when starting from state s, taking action a, and afterwards following policy π. Here, the Monte Carlo method is basically the same as it was for state values. The only difference now is that we are looking at state-action pairs rather than just at states. Every visit and first visit Monte Carlo still converge to the true expected reward values as the number of visits goes to infinity.

A problem that arises now, however, is that many state-action pairs will never be visited. If we follow a deterministic policy then Monte Carlo will only ever see returns for one action from each state. Since the other actions will never be picked, there will be no estimation of returns from those actions. It is necessary that we estimate returns for all actions in a state, not only the ones that the policy prefers.

This is the problem of maintaining exploration. To ensure that policy evaluation still works for action values, we must make exploration continue. This can be achieved by starting each episode in a random state-action pair and giving every state-action pair a nonzero probability of being picked. This is called the method of exploring starts and it does guarantee that each state-action pair will be visited an infinite number of times as the number of episodes goes to infinity. However, assuming that there are exploring starts is not always a reliable method; we cannot depend on exploring starts when we want to learn from real world experience and interaction. A common alternative to exploring starts is to use stochastic policies and assign a nonzero probability to each action that can be taken in a state.

3.2.3 Monte Carlo Control

We can make use of Monte Carlo estimation to approximate an optimal policy. The general idea of policy iteration from section 3.1.3 will be followed. In a general policy iteration, we maintain an approximate value function and an approximate policy. We iteratively update the value function to be the value function for the current policy, and we iteratively improve the policy compared to our current value function. Together, these processes push the policy and value function towards optimality. To begin, we consider a Monte Carlo version of classical policy iteration. As before, we'll do alternating steps of policy evaluation and policy improvement. We will begin with an arbitrary policy π_0, and end with optimal policy π_* and optimal action-value function q_*.

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} q_*$$

E shows a policy evaluation step and I shows a policy improvement step. Under the assumptions that we explore an infinite number of episodes and that those episodes are started with exploring starts, Monte Carlo policy iteration can exactly compute each q_{π_k} for any possible π_k.

In this policy improvement, we have an action-value function because we do not have a model of the environment. With any action-value function, the greedy policy is one that for each state chooses the action with the maximal action-value:

$\pi(s) \doteq argmax_a(q(s,a))$

Policy improvement can be performed by building the next policy as the greedy policy for the current action-value function. If we construct the next policy in this way, the policy improvement theorem (3.5) applies to both current and next policies, because for all states in our state space:

$q_{\pi_k} = q_{\pi_k}(s, argmax_a(q(s,a)))$
$q_{\pi_k} = max_a q_{\pi_k}(s,a)$
$q_{\pi_k} \geq q_{\pi_k}(s, \pi_k(s))$
$q_{\pi_k} \geq v_{\pi_k}(s)$

To reiterate the details of the policy improvement theorem, it guarantees that our new policy π_{k+1} will be at least as good as the old policy π_k until an optimal policy is reached. This guarantee means that the overall process of policy iteration will converge to an optimal policy and value function.

3.3 Temporal Difference Learning

Temporal Difference (TD) combines ideas from both Monte Carlo and Dynamic Programming. Similarly to Monte Carlo methods, TD does not require knowledge of an environments mechanics and dynamics in order to learn. TD methods will update their estimates based partially on other estimates they've already learned, and don't wait for a final outcome.

3.3.1 Temporal Difference Prediction

Similarly to Monte Carlo methods, TD uses experience to do prediction. Under policy π, obtain some experience in the form of episodes. Both TD and Monte Carlo will update their estimates V of v_π for each state that is not terminal and occurs in the collected experience set. Monte Carlo

methods wait to update their estimate until they know the return following their visit to state S. The return is then used to update $V(S_t)$. Below is an example of an every-visit Monte Carlo method that works well in nonstationary environments:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{3.8}$$

This every-visit Monte Carlo method can easily be turned into an extremely simple TD update by substituting the return G_t for the value estimate at the next state $V(S_{t+1})$.

$$V(S_t) \leftarrow V(S_t) + \alpha[V(S_{t+1}) - V(S_t)] \tag{3.9}$$

Now, we'll look at what environments these algorithms were used in and the results from each.

4 Discussion

Each algorithm, Dynamic Programming, Monte Carlo, and Temporal Difference, was used to solve a different problem. I'll discuss each problem, starting with gridworld, then blackjack, and finally tic-tac-toe, and why the algorithm used worked well for the problem it was used on, as well as how that algorithm would likely perform on the other problems.

4.1 Solving Gridworld with Dynamic Programming

The gridworld environment is an NxN grid with one terminal state. We randomly initialize the agent to a point within the grid and want it to navigate to the terminal state. Reward for all actions that don't result in reaching the terminal state is -1, and reward for reaching the terminal state is 0. This incentivizes the agent to reach the ending state as quickly as possible.

In gridworld, we have complete knowledge of the environment, its dynamics, and all transition probabilities. This makes it wonderfully well suited for dynamic programming, as long as we do not produce too large of a grid.

We solve dynamic programming by policy iteration, and the algorithm used is outlined in

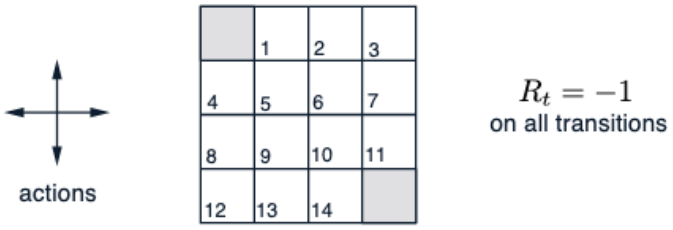


Figure 1: The gridworld environment. [1]

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

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

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

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

Figure 2: A dynamic programming algorithm for policy iteration. [1]

figure 2.

The policy iteration algorithm computes an optimal value function and optimal policy, and those can be used to make decisions about actions to take in gridworld.

The dynamic programming approach to policy iteration successfully terminates for gridworld, and the resulting agent is able to make optimal action choices to move through gridworld.

Both Monte Carlo methods and Temporal Difference learning would work well on gridworld. Monte Carlo and TD require only experience to learn an optimal policy and value function, and in this case, it is easy to generate that experience. There are no characteristics of gridworld that would prevent either Monte Carlo or TD learning from working well on it.

```
Value function reshaped onto gridworld
array([[ 0., -1., -2., -3.],
       [-1., -2., -3., -2.],
       [-2., -3., -2., -1.],
       [-3., -2., -1.,  0.]])
```

Figure 3: Value function for 4 by 4 gridworld found by my code.

4.1.1 Learning to play Blackjack with Monte Carlo methods

Dynamic programming is not as well suited to blackjack as it is to gridworld, because it is much harder to figure out the state transition probabilities in blackjack than in gridworld. However, Monte Carlo methods work well here since they do not require a model of the environment and are capable of learning solely through experience.

In blackjack, the object of the game is to obtain a hand of cards with a point total as close to 21 as possible, without going over 21. All numbered cards have a point value equal to the number on their face, all face cards have a value of 10 points, and aces have a value of either 1 or 11. The player is dealt two cards, face up, and can choose to either hit (receive another card) or stick (commit to having only the current cards). The dealer is dealt two cards, one face down, one face up, and the dealer's turn begins once the player chooses to stick. The dealer can take the same actions as the player.

In the implementation of this Monte Carlo method, we sample from an infinite deck (with card replacement) and the dealer follows a deterministic policy. The dealer always sticks on every point total greater than or equal to 17 and always hits on a point total less than 17. The agent is allowed to observe three things about the state of the game: its point total, the dealer's showing card, and whether or not the agent has a usable ace.

We do Monte Carlo Policy Iteration to approximate an optimal policy and optimal value function, and then we can use those to play blackjack against the deterministic dealer.

The algorithm was trained for 1 million iterations. It produces a good approximation of the action-value function, and wins against the dealer approximately 25 percent of the time by the end of training.

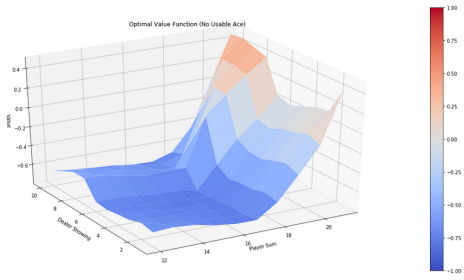
It is likely that TD learning would perform well on this problem too, since it also requires only experience to learn and does not rely on a model of the environment. It is possible that TD would

First-visit MC prediction, for estimating $V \approx v_\pi$

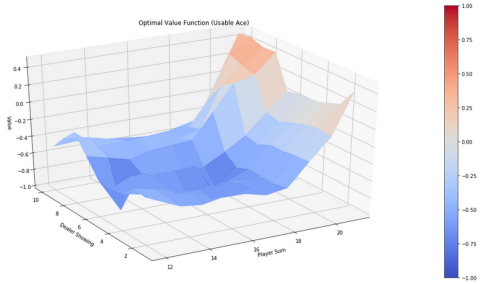
Input: a policy π to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append G to $Returns(S_t)$
            $V(S_t) \leftarrow$ average$(Returns(S_t))$

Figure 4: First Visit Monte Carlo Prediction [1]



(a) Action-value function for usable ace.



(b) Action-value function for no usable ace.

Figure 5: Approximate action-value functions for blackjack.

learn more efficiently than Monte Carlo, since it does not require a truly random sample across all state-action pairs and can instead be formulated as a value update rule throughout training.

4.1.2 Using Temporal Difference Learning to play Tic-Tac-Toe

Let's review the game of tic-tac-toe. The game takes place on a 3 by 3 grid, one player plays as Xs, and the other player plays as Os. The object is to place three of your pieces in a row on the board, and you may only place one piece per turn.

In building the TD agent to learn to play tic-tac-toe, we consider draws and losses to be equally bad. To do this, we start with initializing an array of numbers, where each number in the array corresponds to the estimated value of that board state. In this case, our values are synonymous with win probability in a state. The entire array represents our value function. We initialize all states where the agent (for simplicity, it always plays X) has won to value of 1, all states where the agent loses or the game ends in a draw to value 0, and all other states to 0.5.

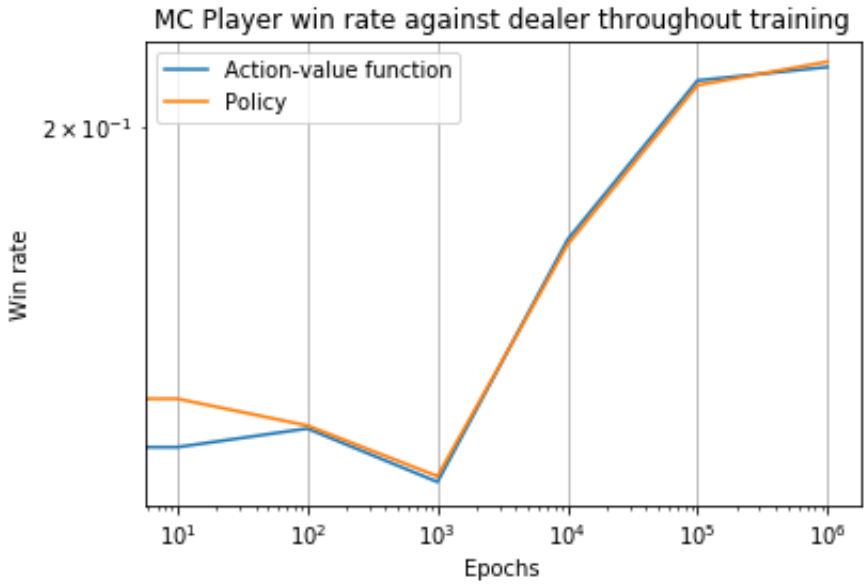


Figure 6: Win rate of the Monte Carlo blackjack agent against the dealer throughout training. Decisions were made with the action-value function (blue) and policy (orange) to see if there was any difference between them.

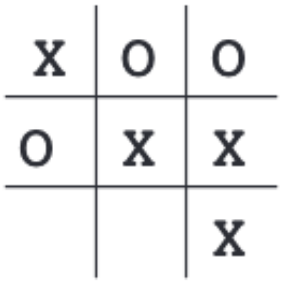


Figure 7: An example of a tic-tac-toe board. X has won in this image. [1]

Once we've approximated our value function, we simply play lots of games against our opponent. In order to choose the next move while in the game, we look at the set of next legal board states and see what the corresponding value is in our array. We will most frequently make random moves, but some of the time must make random, exploratory moves. These exploratory moves enable us to see states that we might otherwise never experience.

Throughout playing, we update the values of the states that we are in. We aim to make our values more accurate estimates of the probability of winning. We'll execute this goal by moving our value estimate of the earlier state a fraction of the way towards the later states value. This is done following this update rule:

$$V(s) \leftarrow V(s) + \alpha[V(s') - V(s)] \tag{4.1}$$

Where s is the state before making a greedy move, s' is the state after making a greedy move, α is a step-size parameter, and $V(s)$ is the estimated value of the state.

If we choose to reduce $\alpha \rightarrow 0$ over time, this algorithm will converge to the true win probabilities for a fixed opponent. That is, an opponent that never changes their style of play. Should we instead reduce α not all the way to zero, this algorithm will play well against opponents that slowly change their style of playing.

As is typical in tic-tac-toe, this agent is allowed to observe the full board state. In about 5,000 games, it trained to a level to consistently draw against adult humans, with the occasional win or loss.

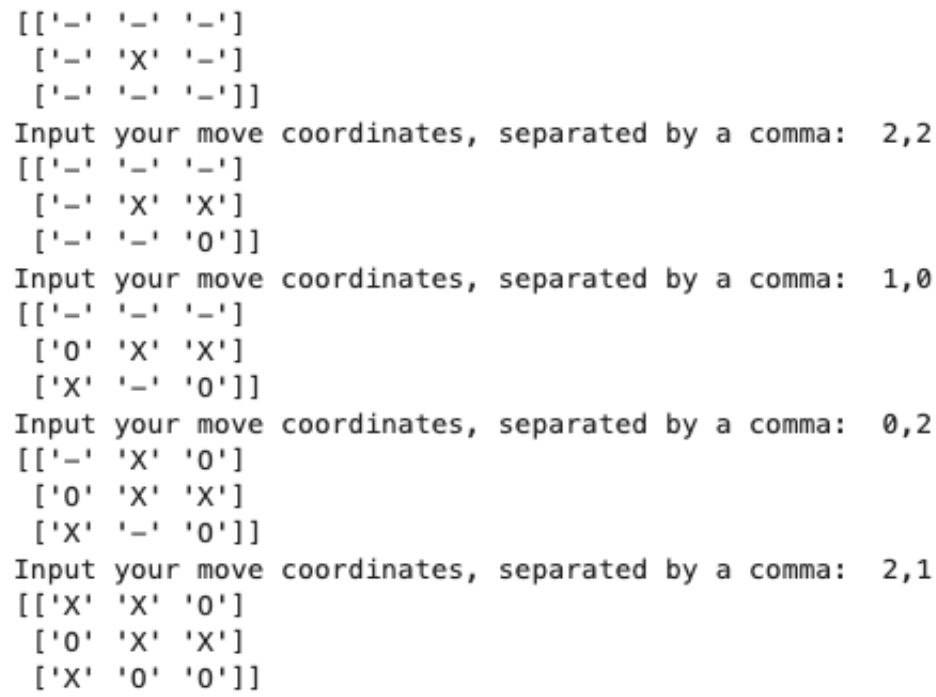


Figure 8: Screenshot of gameplay against the TD tic-tac-toe player. Human is O, agent is X.

5 Conclusion

Throughout this project, I was able to survey a fundamental set of reinforcement learning (RL) algorithms and begin to build up an understanding of the field. Investigations were made into Dynamic Programming, Monte Carlo methods, and Temporal Difference learning and how they could be applied to different problems. In the future, I will continue my work with RL. I'll move on to learning about policy gradients, and from there, some very current and state of the art, methods. Future work will also include an investigation into applying regularization methods to RL, in an effort to make the algorithms generalize better. Finally, moving forward, I hope to become stronger at building simulations for RL agents to learn within.

6 Acknowledgements

Significant gratitude goes to Brian Bartoldson for his continuous guidance and support throughout the semester. A thank you also goes to the rest of the Computational Intelligence Lab.

7 Footnotes

1. My code can be found at my GitHub repository, here: (https://github.com/jfpettit/reinforcement-learning).

References

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. In: The MIT Press, Cambridge, Massachusetts, London, England, 2018.

[2] Denny Britz. *Implementation of Reinforcement Learning Algorithms. Python, OpenAI Gym, Tensorflow. Exercises and Solutions to accompany Sutton's Book and David Silver's course.* In: https://github.com/dennybritz/reinforcement-learning