Reinforcement Learning: Policy Updates & Policy Gradients

Ethan B. Mehta
ethanbmehta@berkeley.edu

Sean Lin
seanlin2000@berkeley.edu

Jaiveer Singh
j.singh@berkeley.edu

1 Introduction

In this note, we will review notions of state space and action space, present a number of learning algorithms on discrete spaces, and then delve into state-of-the-art DeepRL algorithms like VPG and A3C. This note accompanies the project and provides depth to topics broached in the project specification.

The goal of this note is to impart a high-level understanding of policy updates that connects to your understanding from EECS 16AB, without getting bogged down in the nitty-gritty details of RL. However, if you would like to learn more about these fundamentals, we have provided thorough appendices at the end of this note.

1.1 Toy Example: Taxi World

To help make this note more digestible, we will consistently refer to the concrete example of a 'Taxi World' when introducing new ideas.

Taxi World is played on a simple 2-dimensional, 5x5 grid. The Agent plays as the driver of a single taxi that can navigate around the grid by driving Up, Down, Left or Right. A selection of edges in the grid serve as walls that cannot be driven through; instead, the taxi must drive around the walls. In the four corners of the grid are special locations indicated by the letters R, G, B, and Y. A single passenger will appear at one of those 4 RGBY locations, with a desire to be transported to a different RGBY location. Once the taxi reaches the RGBY square with the waiting passenger, the Agent can execute a Pick-up Action. Once the taxi reaches the destination RGBY square with the passenger in the car, the Agent can execute a Drop-Off Action and receive some payment for the service. As soon as the first passenger is dropped off, a second passenger spawns with a random start and desired end RGBY location, and so on for a total of 3 passengers per game played.

You will have the opportunity to explore Taxi World yourself, as well as train several remarkably capable Agents, in the project. Self-driving car, anyone?

2 Solving Reinforcement Learning Problems

Let's begin by refreshing our understanding of what exactly a Reinforcement Learning (RL) problem is.

In a sentence, RL is about training a model (Agent) that continually observes the State of its environment, and then takes the 'best' Actions to maximize Reward.

This idea should parallel the foundations of control built up in EECS 16B. The notion of State is consistent between 16B and this presentation of RL; the State encapsulates the current status of all

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

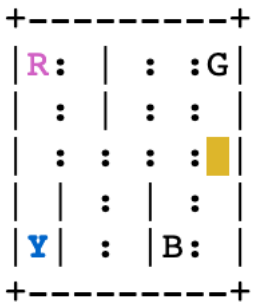

Figure 1: Taxi World

relevant elements in the environment. In closed-loop control, we found a fixed expression for the input $u(t)$ in terms of the State $x(t)$, such that the combined evolution of $x(t+1) = f(x(t), u(t), w(t))$ converged towards our desired State $x^*(t)$. The random disturbances $w(t)$ are modelled in RL as uncertainty in the Transition function. Instead of finding a 'simple' $u(t)$, however, RL involves training an Agent that can respond differently to different States. At a macro level, we are essentially adding more degrees of freedom to the control law, in order to get a better result. RL also concretizes the notion of Reward, which means we can incentivize specific Reward-generating behavior (ex: reach the passenger's pick-up location) instead of simply approaching a fixed target State (ex: navigate to $(1, 4)$).

More concretely, we want to train an Agent that can behave optimally, where optimality is reached when the Agent takes Actions that maximize its total expected Reward. For Taxi World, an optimal Agent would be one that can navigate to the passenger pick-up location and then the passenger drop-off location as quickly as possible, earning the most fares possible over a specific time interval.

Broadly, we refer to "solving RL problems" as the procedure of learning this optimal, reward-maximizing behavior.

2.1 Policy

To formalize the notion of 'behavior', we introduce Policy, which will be the primary focus of this note. A Policy is a mapping from States to Actions, which can be used by the Agent to select which Action to take from its current State. In environments with discrete State and Action Spaces, a Policy can essentially function as a 'lookup table', often implemented as a dictionary.

A complete, deterministic Policy maps each state to a single action, and can be represented as $\pi(s)$. In some situations, especially in the continuous domain, it makes more sense to have a stochastic Policy that instead specifies several actions for each state, with a fixed probability for each action. The optimal Policy function, $\pi^*(s)$, is what we would like to learn. Our goal is to now build up a formal framework that will allow the agents we write to learn that optimal Policy $\pi^*(s)$.

Policy in Taxi World We can leverage our knowledge of Taxi World to intuitively identify several of the (State –> Action) mappings in the optimal policy. For example, if the Agent is in the current passenger pick-up square and the passenger has yet to enter the taxi, then obviously the optimal choice is to execute the Pick-Up action. However, there are other situations where the optimal policy is less obvious to a human observer. For example, immediately after executing the Pick-Up Action, it might not be instantly clear to us which direction the Agent should move through the walls in order to achieve the shortest transit time, though with some time, we could manually compute the Maze Distance using standard path-planning techniques. Amazingly, the Agent you will write in the project does no such Maze Distance calculation upfront, and yet it learns the optimal policy regardless! Continued reading will help us understand how this can be the case.

3 Online and Model-Free Learning

In some specific, simplified environments, it is possible to explicitly model the entire evolution of the State in a closed-form way. Virtually all of the simple control problems you have seen in EECS 16B were of this form. The techniques used to solve this kind of problem are referred to as 'Model-Based,' because they operate given some ground-truth understanding of how the environment actually works.

Model-based learning is often a good choice for simple games, especially because we can solve model-based problems entirely offline. Here, 'Offline' refers to the fact that we don't need to actually interact in the real environment at all. Instead, offline methods use the model of the real environment to construct a local 'toy' environment, learn the optimal policy there, and then return to the real environment only after achieving mastery. An analogy for this approach would be if you learned to fly an airplane using only flight simulators, and consequently mastered aviation without ever actually being in the sky.

However, many environments are so large, complex, or unknown in such a way that it is impossible to construct an accurate 'toy' model to learn from. Instead, the only way we can learn in these sorts of environments is by actually trying various Actions and observing what happens as a result. The fact that we choose not to construct a model for the environment makes this type of approach 'Model-Free', and the fact that we must actively engage with the real environment to learn makes this approach 'Online'.

There are many interesting techniques in the model-free, online realm; some of the most foundational are summarized in Appendix C.

4 Introduction to Policy Gradients & Deep RL

4.1 Theory of Policy Gradients

The goal of a reinforcement learning task is simply to determine the optimal policy for the agent to maximize reward. Just like in other machine learning paradigms, an optimal policy that *maximizes* a specific function (reward) can be iteratively computed using gradient ascent! This method of iterative policy optimization is known as **Policy gradients**.

Let θ be our set of parameters, R be our Reward function, and τ be a trajectory of Actions and States that we take in the world. A Policy gradient computes $\nabla_\theta E[R(\tau)|\theta]$. Using gradient ascent, we try to optimize θ using this policy gradient.

The derivation for the Policy gradient is as follows:

$$\begin{aligned}\nabla_\theta E_\theta[R\tau] &= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) \\ &= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) \\ &= \int_\tau \frac{P(\tau|\theta)}{P(\tau|\theta)}\nabla_\theta P(\tau|\theta)R(\tau)\end{aligned}$$

Now, recall that: $\nabla_x \log f(x) = \frac{\nabla_x f(x)}{f(x)}$. We can use this information to say that:

$$\nabla_\theta E_\theta[R\tau] = \int_\tau P(\tau|\theta)\nabla_\theta \log(P(\tau|\theta))R(\tau) = E_\theta[\nabla_\theta \log(P(\tau|\theta))R(\tau)]$$

We can break this down into parts. First of all, we can define the probability of a trajectory given the parameters as follows:

$$P(\tau|\theta) = p_0(s_0)\Pi_{t=1}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

where $p_0(s_0)$ is the probability of starting at State 0, $P(s_{t+1}|s_t, a_t)$ is the probability of transitioning to State s_{t+1} given that we were at State s_t and took Action a_t, and $\pi_\theta(a_t|s_t)$ is the probability that our Policy tells us to take Action a_t given that we were at State s_t. Notice that our Policy is stochastic and not deterministic, another difference from the discrete setting.

Taking the log of both sides and then taking the gradient, we get the following expressions:

$$\log P(\tau|\theta) = \log p_0(s_0) + \sum_{t=1}^{T} \log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)$$

$$\nabla_\theta \log P(\tau|\theta) = \nabla_\theta[\log p_0(s_0) + \sum_{t=1}^{T} \log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)] = \sum_{t=1}^{T} \nabla_\theta \log P(s_{t+1}|s_t, a_t)$$

We can now plug this back into our equation for gradient of expected Reward:

$$J(\theta) = \nabla_\theta E_\theta[R\tau] = E_\theta[\sum_{t=1}^{T} \nabla_\theta \log P(s_{t+1}|s_t, a_t) R(\tau)]$$

to obtain the basic expression for the Policy gradient.

We use this expression to perform gradient ascent updates to our parameters θ to try to find the Policy yielding the maximal Reward:

$$\theta_{t+1} = \theta_t + \alpha * (E_\theta[\sum_{t=1}^{T} \nabla_\theta \log P(s_{t+1}|s_t, a_t) R(\tau)])$$

4.2 Practical Methods for Policy Gradients

For Policy optimization algorithms, neural networks are generally used to implement the Policy. Given continuous observations as input, a classification neural network is used to output the logits (log-odds), which are then processed into probabilities that represent the chance that we choose an Action a. Given these probabilities, we then sample from a multinomial distribution defined by these probabilities to pick what Action to take. Neural networks make it easy to compute the gradient and update the parameters using backpropogation.

4.3 Deep RL with A3C: Asynchronous Advantage Actor-Critic

Now that we understand the methodology of Policy gradients, we will explore a more advanced, state-of-the-art DeepRL technique known as Asynchronous Advantage Actor-Critic.

The Asynchronous Advantage Actor-Critic algorithm, henceforth referred to as A3C, was introduced in 2016 and has since become a standard algorithm in the DeepRL toolbox. Credit for the original algorithm is due to Google's DeepMind in this paper: `https://arxiv.org/pdf/1602.01783.pdf`.

To understand A3C, it is helpful to break the name down into each of its component parts:

1. **Asynchronous:** The most naive Policy Gradient approach is called Vanilla Policy Gradient (VPG), which follows the derivation shown above with a single Agent training against a single environment to build a single model. In contrast, A3C achieves significant speedups by employing multiple asynchronous worker Agents. Each worker Agent maintains its own local model and trains against a local copy of the environment, performing a variant of mini-batch gradient descent to update the local model. As each asynchronous worker completes a training episode, it checks its performance against the single, global model. If any local model is better than the current global model, the global model is updated and each worker's local model is reset to match the improvement.

   Intuitively, A3C's asynchronous behavior makes it very parallelizable, which is great for the highly-parallel computers that have become commonplace in recent years.

2. **Advantage:** 'Advantage' is a technical term that has a mathematical definition which is outside the scope of this course. However, we can still appreciate the overall idea.

   A3C is different from earlier Actor-Critic approaches because instead of having the Critic learn a Value function over States, the Critic learns an Advantage function over State-Action pairs. In other words, the algorithm prioritizes exploring States with higher Rewards than expected, instead of simply the States with higher Rewards. This will make more sense after reading the subsequent Actor-Critic definition.

3. **Actor-Critic:** Actor-Critic is a framework in which one part of the learning system attempts to identify the best Action from a given State to a new State, and the other part of the learning system attempts to identify how 'good' that new State is. The Actor serves as a probabilistic version of the Policy function, deciding which Action the agent should take. The Critic then serves as an approximation of the Advantage function, using a neural net to estimate how much better the Actor's Action is against the current Policy. Through this iterative, back-and-forth process, both the Actor and the Critic get better at their respective jobs, and so the Agent as a whole improves.

A3C offers a number of strengths over algorithms like Vanilla Policy Gradient. Firstly, the distributed nature of the workload makes this algorithm efficient on parallel computers with multiple CPU cores. Second, the distributed nature also encourages broader exploration around the problem, reducing the chance that the global model gets stuck in a poor local maximum. Finally, since A3C operates by estimating Advantage instead of Value, the algorithm encourages its workers to spend further effort investigating 'promising' areas where the realized rewards are higher than the expected rewards.

This higher-level understanding of A3C will be sufficient for most use cases, but the intrepid learner is encouraged to refer to the original 2016 paper for a more detailed explanation: `https://arxiv.org/pdf/1602.01783.pdf`.

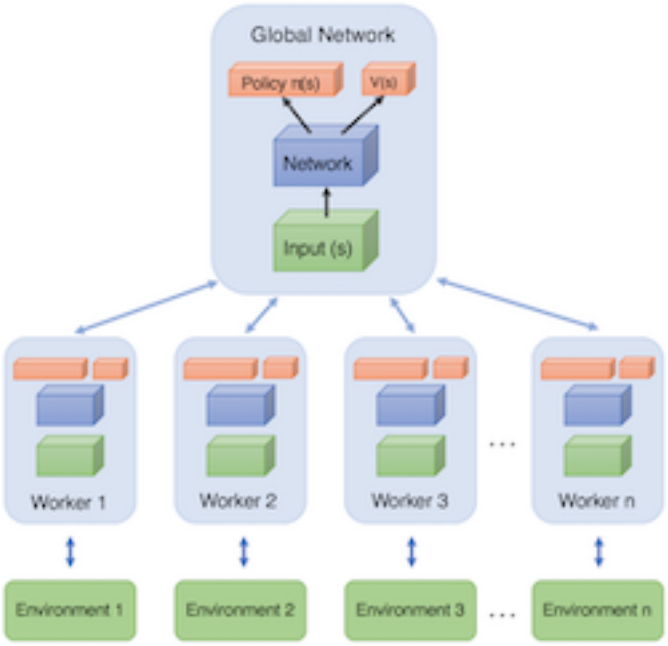

Figure 2: A3C Algorithm Parallelizing RL Exploration

5 Conclusion

In this note, we have built up the basics of RL and understood the parallels between RL's focus on Reward maximization and traditional ML's focus on Loss minimization. We have covered the Vanilla Policy Gradient technique, which operates in a manner akin to Stochastic Gradient Descent for learning the optimal policy. Finally, we introduced a more advanced Deep RL method of A3C, which uses a framework akin to Boosting in order to reach optimality faster.

In the interest of keeping this content accessible from just an EECS 16AB background, we have contained much of the derivation and thorough explanation of other techniques in the following appendices. While these ideas are not directly in scope for this project, reading through the appendices may help you build a stronger intuition for RL.

A Appendix A: Intro to Markov Decision Processes (MDPs)

Recall that our goal with reinforcement learning (RL) is to train our agent to make optimal decisions in its environment. Consequently, a natural starting point for reinforcement learning could be to develop a model of the underlying environment. Our study of Markov Decision Processes is motivated by this observation.

A Markov Decision Process, henceforth referred to as MDP, is a broad model of an environment that accounts for both random and controlled events. A MDP is defined by the following: States; Actions; Transition and Reward functions; and a discount factor. At each time step, the Agent, or decision-maker, chooses an Action available to it at its current State. As a result of that Action, the Agent transitions to a new State determined by the Transition function probability distribution, and the Agent incurs a reward or penalty (negative reward) based on the Reward function probability distribution. The Agent continues choosing Actions, transitioning to new States, and incurring Rewards until it reaches a terminal State, a State from which there are no available Actions.

With this quick overview complete, let's look at each piece of the model in more detail.

A.1 States

A State captures the the current status of each element in the environment. The set of all possible States of the environment is known as an MDP's State Space. For example, in the game of chess, one complete formulation of the State would include the positions of each piece on the board, the current turn (black or white), and whether or not each king has castled. In a simpler game like tic-tac-toe, the State would be the position of all the X's and O's and whether or not it is the Agent's turn. From a State, the Agent chooses an Action, and transitions to the next State with some Transition probability.

A.2 Actions

Depending on the State of the environment, the Agent will have a set of possible Actions available to it. An Action is a controlled choice about what future outcome the Agent desires, based on the Agent's observation of the current State. The set of Actions is known as the Action Space.

Importantly, an MDP allows for modelling environments with uncertainty, and so an Agent's Actions are not the only factor that determine the next State. More precisely, in stochastic environments, it may be the case that there are stochastic Transitions from State to State. In a game like chess, transitions are deterministic; as long as the move is valid, a player may indicate an intention to move their pawn forward (Action) and then know with absolute certainty that the pawn has reached its new square in the subsequent State. However, in the card game Blackjack, an Agent's decision to take the "Hit" Action and receive a new card from the dealer will send them to one State randomly determined out of a number of States based on which exact card has been dealt.

We will address this idea of uncertainty again when we define the Transition function.

A.3 Q-States

At the most basic level, Q-States are a combination of a State and an Action taken from the Action Space available at that State. Q-States are intermediate States that are useful in representing the process of the Agent's decision making. More explicitly, a Q-State is the State the environment is in after an Agent has decided which Action they will take, but before the Agent has actually executed that Action.

A.4 Transition Function

Our earlier discussion about the uncertainty and randomness associated with Actions can be formally expressed using a Transition function. A Transition function, of the form $T(s, a, s')$, specifies the probability of transitioning to State s' given that the agent was previously in State s and took Action a. For deterministic games like chess, each State-Action pair can only result in exactly one next State s'; thus, $T(s, a, s_0) = 1$ for exactly one s_0 and $T(s, a, s_k) = 0$ for $k \neq 0$. In stochastic games, the Transition function is often a probability distribution over multiple destination States s'. Rolling

a six-sided die, for example, has a Transition function of $T(s_{init}, a_{roll}, s_i) = \frac{1}{6}$ for some integer $1 \leq i \leq 6$.

A.5 Reward Function

At the same time that a Transition is made, the Agent will receive some Reward based on its previous State, Action taken, and next State; the function that determines the value of this Reward is the Reward function. In an arcade game like Pacman, a simple Reward Function will give the Agent a $+1$ Reward each time the Pacman eats a food pellet, and a $+50$ Reward each time the Pacman eats a vulnerable Ghost.

However, Reward functions can also be more sophisticated. One way to incentivize an Agent to solve games quicker is to introduce what is (counterintuitively) known as a Living Reward. Each timestep that the Agent is alive, it receives a small penalty; tuned correctly, this design will motivate the Agent to find shorter solutions that will end the game sooner, with less accumulated penalty. Reward functions can also be designed to incorporate risk-averse or risk-seeking behaviors, by artificially shrinking down or blowing up Rewards at either extreme. A risk-averse Pacman might exhibit a strong fear of Ghosts and miss out on food pellets that are close to enemies, while a risk-seeking Pacman might attempt a feat of bravery, dashing in and out of a Ghost's path to secure a single pellet. While out of scope for this note, Utility functions are a common mechanism to specify risk tolerance.

A.6 Discount Factor

Optionally, many MDPs include a discount factor. A discount factor, typically notated as the Greek letter 'gamma' γ, specifies a multiplicative "discount" applied towards future rewards. Suppose an Agent is contemplating taking a certain Action with accompanying Reward $+100$, and further suppose that $\gamma = 0.1$. If that Action is taken on the very next timestep, the Agent enjoys the full $+100$ Reward. If, however, the Agent decides to wait a timestep before taking the Action on the second timestep, then the 'current value' of the Reward is only $0.1 \times 100 = 10$. Similarly, the 'current value' of the Reward if the necessary action occurs on the third timestep is $(0.1)^2 \times 100 = 1$, and so on.

The purpose of discounting rewards by the discount factor is to indicate a preference for earlier Reward over later Reward. This concept should be intuitive from a real-world perspective: \$10,000 in hand today is more useful than \$10,000 received fifteen years from now. The sooner the Agent incurs a Reward, the less discounted the Reward will be, and so the greater the total Reward the agent will receive. A discount factor $\gamma < 1$ is also mathematically useful for ensuring convergence of Reward values when dealing with infinite games, though we will not delve further into that math here.

A.7 Markov Property

It is important to note that MDPs must obey the Markov Property, also known as the memoryless property. Essentially, all Transitions, Actions, and Rewards must be based on only the current and immediate next States. For example, consider an Agent that moves from State s_0 to s_1 via Action a_0, and then from s_1 to s_2 via a_1. Now, the probability that the Agent Transitions to state s_3 must only depend on the current state s_2 and action a_2, not on either of the previous states or actions s_0, s_1, a_0, or a_1. Mathematically, $P(s_3|a_2, s_2, a_1, s_1, a_0, s_0) = P(s_3|a_2, s_2)$. While this may seem like a major limitation, the Markov Property is critical to the elegant simplicity of MDPs.

Practically speaking, if one finds that information about previous real-world states is needed, then one can create a larger MDP State that encapsulates past information. For example, in a best-of-three Rock-Paper-Scissors game, we can design the MDP such that the State contains the results of the two previous games at each timestep. In this way, even though the Transitions are determined by events that occurred in the real-world past, we use only information contained in the current State and thus we do not violate the Markov Property.

B Appendix B: Bellman-Ford and Value & Policy Iteration

Our goal in RL to maximize the expected total Reward over the lifetime of the agent. But how can our Agent in the present deal with future Rewards that are uncertain and unrealized?

The discussion of Values and Q-Values will address this question.

B.1 Values and Q-Values

We will start by defining two quantities: $V^*(s)$ and $Q^*(s, a)$.

1. $V^*(s)$ is the expected value of the Reward that an Agent starting from state s and behaving optimally afterwards will receive over its entire lifetime.
2. $Q^*(s, a)$ is the expected value of the Reward that an Agent starting from Q-State (s, a) and behaving optimally afterwards will receive over its entire lifetime. That is, the Agent begins in State s, takes an Action a, and then proceeds optimally after reaching the next State s'.

The astute reader will recognize that there is an inherent recursion at play between the Values and Q-Values. The Value for a State s_0 should conceivably be impacted by the Values of the States $s_1, ...s_n$ that follow s_0. We will explore this more in the next section.

Values in Taxi World Generally speaking, the overall trend of Values across the state space of a simple problem like Taxi World will make sense to a human observer, even if the exact numerical quantities of each Value are not easily verifiable. The Value of a State in which the taxi is about to drop-off a passenger is extremely high, because the Reward in the form of taxi fare will immediately result from that drop-off Action. We would also expect the Value of a State in which the taxi is exactly 1 square away from the destination to also be quite high, though it should be lower than the aforementioned Value at the exact drop-off location. Applying a real-world perspective, when we are in the taxi just one block away with the passenger's destination in sight, the likelihood that we successfully reach that destination and receive the taxi fare is very high. However, there is also a chance that a freak incident could force us to abort the trip and receive no reward: the vehicle suddenly breaks down, some emergency workers block traffic, or a train barrels through along the middle of the street. Like we understand from this real-world example, uncertainty and temporal distance from a potential Reward will both reduce the Value of a State.

B.2 A Tangible Formulation: The Bellman Equation

The Bellman Equation formulates this recursion mathematically, representing the optimal Values and Q-Values of each State as a function of other States' optimal Values and Q-Values. We will present the equation here, and then derive the Bellman Equation through logical reasoning.

$$V^*(s) = \max_{a \in actions} \sum_{s' \in states} T(s, a, s')[R(s, a, s') + \gamma \times V^*(s')]$$

Consider the following thought experiment: You are the Agent in a world modeled by an MDP, and you know the optimal value $V^*(s)$ for all States of the MDP. Currently, you are in a State s_0. How would you choose which Action to take?

Intuitively, you would want to choose the Action that moves yourself to the State with the highest value of $V^*(s)$. The notion of maximizing over Actions suggests the following structure so far:

$$V^*(s) = \max_{a \in actions} (...)$$

However, in an uncertain world, there are multiple resultant States s' that you may find yourself in after taking Action a. You need to consider each of these States s', as well as the probabilities of reaching those States after having taken the Action a. Looping over all s' suggests this:

$$V^*(s) = \max_{a \in actions} \sum_{s' \in states} T(s, a, s')(...)$$

Recall that each Transition may be accompanied by a Reward. Since your goal is to find the Action that maximizes expected Reward, it is obvious that you must include the Reward function in your equation:

$$V^*(s) = \max_{a \in actions} \sum_{s' \in states} T(s, a, s')[R(s, a, s') + ...]$$

Now, recall also that the Reward function only encapsulates the Reward earned while transitioning from s to s' via a. If the game ends immediately after that Transition, then that is all you would need to consider. However, if the game continues, you need some way of representing the expected future Reward from the new State s' you have just arrived in.

This is exactly the definition of $V^*(s')$! This insight yields the following:

$$V^*(s) = \max_{a \in actions} \sum_{s' \in states} T(s, a, s')[R(s, a, s') + (...) \times V^*(s')]$$

Finally, recall that future Rewards are typically less beneficial than sooner Rewards of equal magnitude. As a result, you must incorporate the discount factor γ. The result is the complete Bellman Equation:

$$V^*(s) = \max_{a \in actions} \sum_{s' \in states} T(s, a, s')[R(s, a, s') + \gamma \times V^*(s')]$$

After going through this derivation, the equation for $Q^*(s, a)$ should be clear. Instead of maximizing over all Actions as in the previous equation, we are explicitly given the first Action a. The rest of the equation follows cleanly:

$$Q^*(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \times V^*(s')]$$

Notice that we can combine the two equations by substituting in $Q^*(s, a)$.

$$V^*(s) = \max_{a \in actions} Q^*(s, a)$$

The Bellman Equation does not give us a way to directly solve for optimal values, but it does provide our algorithms with a guarantee. The guarantee is that if our algorithm can find Values $V(s)$ and Q-Values $Q(s, a)$ such that the Bellman equations are satisfied, then we know that we have found the optimal Values $V(s) = V^*(s)$ and Q-Values $Q(s, a) = Q^*(s, a)$.

In other words, the Bellman Equation will not hold if the Values that we have assigned to a State do not match the optimal Values. Because of this feature, we can use the Bellman Equation to tell us whether or not our Values are optimal.

B.3 Solving for Optimal Values: Value Iteration

Now we will introduce Value Iteration, a technique to actually solve for these optimal values.

The idea of Value Iteration is to iteratively improve our estimates of the Values of each State until we have learned the true optimal Values and satisfied the Bellman Equation. The key subproblem that we will use to solve for $V^*(s)$ is $V_k(s)$. $V_k(s)$ is the expected value of the Reward that an optimally behaving Agent starting in State s will receive over the next k timesteps (as opposed to over the Agent's entire lifetime).

When the numerical values for $V_k(s) = V_{k+1}(s)$ for all s, we know that the Values have converged. Though we will not show it here, it is possible to prove that the converged Values must be the optimal ones: $V_k(s) = V^*(s)$. Thus, once Value Iteration does not change any Values from one iteration to the next, we have achieved our final Values.

By expanding our time horizon (increasing k) until convergence, we can iteratively find $V^*(s)$. Here is the algorithm:

1. For all states, initialize $V_0(s) = 0$.

2. Repeat the following while $V_k(s) \neq V_{k+1}(s)$ for any s

$$V_{k+1}(s) \leftarrow \max_{a \in actions} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma * V_k(s')]$$

This algorithm is a Dynamic Programming algorithm, which means that it represents problems as a combinations of smaller subproblems. One example of a Dynamic Programming algorithm is the iterative Fibonacci function developed in CS 61A. Dynamic Programming is covered in more depth in classes like CS 170.

This iterative algorithm lets us solve for the Values of an MDP, but our true objective is to discover the optimal policy. Thankfully, given the optimal Values, we can extract the optimal policy.

B.4 Extracting Policies for Optimal Values: Policy Extraction

Now that we have the optimal Values of each State, we would like to use these Values to generate a Policy. The procedure to return a Policy given the optimal Values is referred to as Policy Extraction, and it is very reminiscent of the thought experiment posed earlier.

Recall the Bellman Equation written in terms of Q-Values:

$$V^*(s) = \max_a Q^*(s, a)$$

Implicitly, this equation maximizes over all the available Actions to identify the optimal Action. For Policy Extraction, we no longer care about how much Reward the optimal Action receives; instead, we care about what that optimal Action is. Thus, we can simply replace the $\max$ function with an $\arg\max$:

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

.

Substituting for our definition of $Q^*(s, a)$ from an earlier section, we have:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \times V^*(s')]$$

This equation gives us a way of extracting the optimal Policy from the optimal Values. Thus, Value Iteration followed by Policy Extraction will enable us to obtain the optimal policy.

B.5 Evaluation and Improvement: Policy Iteration

While Value Iteration along with Policy Extraction is a viable method to obtain optimal policies, it becomes apparent that this method leads to significant amounts of redundant computation.

Essentially, the key observation is that while the precise numerical values of $V^*(s)$ converge only gradually, the optimal Policy does not care about the precise numerical values. Instead, the optimal Policy is determined by the State-to-State comparison.

This notion suggests that we would expect a Policy to converge faster than all the Values converge; in other words, we should perform Policy Iteration along with Value Iteration.

Here are the basic steps of the Policy Iteration algorithm:

1. Initialize $\pi_0(s)$ to an arbitrary Policy. Recall that a Policy is simply a mapping from states and actions.
2. Repeat the following while $\pi_k(s) \neq \pi_{k+1}(s)$ for any s:

   Use Value Iteration to find the Value of each State while following Policy π_k (defined as $V^{\pi_k}(s)$, as opposed to the optimal Policy $V^*(s)$), by repeating while $V_j^{\pi_k}(s) \neq V_{j+1}^{\pi_k}(s)$ for any s:

$$V_{j+1}^{\pi_k}(s) = \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma \times V_j^{\pi_k}(s')]$$

Update the Policy using Policy Extraction:

$$\pi_{k+1}(s) = \arg\max_{a} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \times V^{\pi_k}(s')]$$

In practice, Policy Iteration saves significant computation time through the interleaving of Policy Extraction and Value Iteration steps.

C Appendix C: Direct Evaluation, TD Learning, and Q-Learning

Recall that in Model-Free learning, the Agent does not try to learn the Transition and Reward functions; the Agent does not assemble a Model of its environment. Instead, the Agent learns the optimal Policy directly, usually through learning the Values or Q-Values of each State.

Model-Free methods are more complex and more deserving of close focus, so we will devote the rest of this section to such methods. We will learn how to evaluate policies using Direct Evaluation and Temporal Difference Learning, and then we will see how to learn the optimal Policy in Q-Learning.

C.1 Direct Evaluation

Direct Evaluation is the simplest Model-Free method. The goal in Direct Evaluation is to evaluate a Policy $\pi(s)$ by following it for several episodes and recording the results. (An episode is a single run on the environment from initial State to a terminal State.)

As the Agent goes through multiple episodes while following the Policy, it maintains counts of the total Reward it receives from each each State and of how many times each State was visited.

Using these values, the agent can estimate $V^{\pi}(s)$. Specifically:

$$V^{\pi}(s) = \frac{\text{total reward achieved from state } s}{\text{times state } s \text{ visited}}$$

Though Direct Evaluation is very simple in implementation, it has a number of glaring flaws that reduce its appeal in practice. The most critical issue is that Direct Evaluation discards all information about the trajectory of States and Actions, only keeping the end results. By treating every single observation the same through the simple average computed at the end of exploration, Direct Evaluation is also vulnerable to poor evaluation caused by a single outlier Transition with extremely high or low incurred Reward.

C.2 Temporal Difference Learning

One step towards solving these problems is Temporal Difference Learning. The core concept of Temporal Difference Learning is that the Agent can learn after each Transition.

In Temporal Difference Learning, the Agent maintains a running estimate of each State's optimal Value which it updates using new samples as it collects them. Importantly, Temporal Difference Learning recognizes that all samples are not equal, and thus uses an exponential average instead of a simple average. This approach addresses some of the biggest problems with Direct Evaluation.

Here is the Temporal Difference Learning algorithm:

1. Start with an initial Value of $V^{\pi}(s) = 0$ for all s.
2. Repeat for the number of exploration episodes:

   Using the Action $a = \pi(s)$ derived from the given Policy and current State s, Transition from s to s' and receive some Reward $R(s, a, s')$ from the environment.

   Using this measured Reward $R(s, a, s')$, construct a sample:

   $$\text{sample} = R(s, a, s') + \gamma \times V^{\pi}(s')$$

   Update $V^{\pi}(s)$ using this new sample and a learning rate $\alpha < 1$:

   $$V^{\pi}(s) \leftarrow (1 - \alpha) \times V^{\pi}(s) + \alpha \times \text{sample}$$

   Optionally, shrink α as exploration continues.

   If the new State s' is a terminal State, advance the episode counter and continue. Otherwise, loop through the Action-Sample-Update process until termination.

The update rule is the main change that allows us to continue learning as we are taking Actions. The update rule also answers the question of how to incorporate our new sample into our running estimate. According to the update rule, we use a learning rate α to guide how much weight to give the new sample.

The reason we favor newer samples is because the $V^{\pi}(s)$ values converge closer to the true $V^{\pi}(s)$ over time; since samples depend on the $V^{\pi}(s)$, newer samples are more accurate, and thus should be weighted higher.

C.3 Q-Learning

Temporal Difference Learning and Direct Evaluation can help us evaluate a learned Policy, but we still need to develop a Policy in the first place. Q-Learning will allow us to do exactly that.

Q-Learning is an algorithm that learns the optimal Policy by learning the optimal Q-Values at each state. Like Temporal Difference Learning, Q-Learning takes samples and incorporates the samples into the running estimates of the Q-States using the learning rate $alpha$. As we have seen, extracting the Policy from the optimal Q-Values is extremely easy: from each State s, take the Action a that maximizes $Q(s, a)$.

The question is how to efficiently navigate through the State Space in order to encounter enough Q-States to build a reliable estimate. Because we maintain the Q-Values at all times, we have access to our current Policy at all times as well. However, if we were to follow our supposedly-best Policy at all times, we would never explore new States that might be pathways to even better Rewards.

The solution used by Q-Learning is known as ϵ-greedy: At each timestep, with some small probability ϵ, take a random Action. Otherwise, use the best Action as determined by the current Policy.

With this, we have enough to create the Q-Learning algorithm in a discrete setting:

1. Initialize $Q(s, a) = 0$ for all s, a.
2. Repeat for the number of exploration episodes:

   Take a random Action a from s with probability ϵ. Otherwise, take the Action $a = \arg\max_a Q(s, a)$ from the current Q-Values.

   Construct a sample using the incurred Reward from reaching the destination s'. (Note that $V(s') = \max_a Q(s', a)$)

   $$\text{sample} = R(s, a, s') + \gamma \times \max_a Q(s', a)$$

   Update $Q(s, a)$ using this new sample and a learning rate $\alpha < 1$:

   $$Q(s, a) \leftarrow (1 - \alpha) * Q(s, a) + \alpha * \text{sample}$$

   Optionally, shrink α as exploration continues.

   If the new State s' is a terminal State, advance the episode counter and continue. Otherwise, loop through the Action-Sample-Update process until termination.

Q-Learning is a powerful tool for environments with discrete State and Action Spaces, and it is the foundation of many more advanced RL algorithms as well. Though we will not delve into it here, Deep Q-Networks (DQN) extends the principle behind Q-Learning using neural networks, and serves as another handy algorithm in a good engineer's Deep RL toolbox.