Deep Reinforcement Learning

A beginner's guide to policy optimization

Jack McGaughey

April 16, 2021

As a beginner that has just begun to see the complexities that the fields of Machine Learning and Artificial Intelligence have in store, it is clear that we need brilliant minds from many perspectives pondering into it. With this exponential technological growth in our computational power added with the intellect of the pioneers in deep learning, it is increasingly important for people across disciplines to learn about AI and its capabilities. My goal throughout my career will be to show people that computer science has much more potential than it might look like. Studying artificial intelligence and deep learning is more than the complex math symbols and confusing code; but philosophies of learning, studying biological systems in nature, and bringing ideas from many scientific domains to solve the problems in AI. The solutions to the problems that arise in the creation of AI are inspired by observations about how nature arranges itself. Deriving these solutions from observations about the natural world requires different world views than that of a computer scientist for which this problem has been previously constrained to.

The reason why Artificial Intelligence should have this different approach is due to the subset of machine learning called deep learning. In my opinion, the biggest difference between classical computer science and deep learning is what controlling a system, a model, an algorithm really means. In what most of us think of programming, you could imagine a programmer sitting down to solve a purely logical problem using a perfectly logical system, a computer. Of course the programmer is coding with some language, perhaps Java or Python, which is highly abstracted from the binary computations that are performed on the computer. The programmer has the feeling of having control over the problem, in that he or she understands the axioms that the solution must be built upon. They will trust that the computer will perform logically, that the computer will do exactly what it was told to do. I think what is fundamentally different between artificial intelligence and classical programming is our interpretation of control over the system. At the core of AI is Deep learning, where we enable a model to change its internal parameters and form connections about data that are not reliant on a human understanding them. Because the system is given this autonomy about how to arrange connections within networks, we lose the nice predictability we had with classical systems.

Artificial neural networks make up what we know as deep learning, and as a common theme in this field of deep learning, we model solutions from nature. It says it right in the name, these neural networks are modeled after the biological neural networks in our brain. This is where following purely logic starts to fall apart, we do not know how to translate these biological systems to artificial ones because we do not completely understand the system. Scientists have more or less come to a consensus about how any particular neuron behaves. A neuron will receive some inputs, then perform some linear function on those inputs to produce an output.

This is saying that the output is proportional to the input. So programmers can in fact code individual neurons and connect them to other individual neurons, but what exactly we are trying to mimic is not so obvious. Somehow, when all of these easily understood functions are put together by the thousands and are interconnected, some intelligence arises out of the system. It is that lack of control over the system because of the lack of understanding about it that separates deep learning from the rest of the domains in computer science.

We understand things at a basic level, in this case neurons. Somehow, when they are arranged in a specific way beyond our knowledge, some intelligence of a higher order emerges from the system. I see this as the same difference between understanding neurology and psychology, or even the difference between an individual's psychology and collective psychology. Neuroscientists can understand how things work at a neurological level, but when billions of neurons are put together into an interconnected web of neurons some psychology emerges that is beyond the intellectual domain of the neuroscientist. I think that this is what is happening with the computer scientist or the programmer. The programmers and computer scientists understand how things work at the level of algorithms and logical functions where there is an input and an explainable output from that input. When we get to the point of having thousands of these functions interconnected in some way, some beautiful intelligence emerges from the system. Just as the neuroscientist might find themselves slightly outside of their domain but rightly interested in the phenomenon, so does the computer scientist.

I am not diminishing the position of the computer scientist in deep learning, they have the tasks of communicating mathematics and ideas into code that the computer can then execute. This fundamental change in the basis of algorithms gives way to people with biological, psychological, or philosophical backgrounds to add their opinions and theories to the plethora of information that computer scientists can pull from. The average programmer's job is to take an idea and implement it into code, the best are able to read the newest theoretical research papers in their respective fields and translate it into a language that a computer can read. A good programmer, after reading the most recently published paper can understand what boundary conditions that their code must satisfy, for which circumstances will it work under, and overall the underlying forces that make the system flourish.

I think a lot of people have really good ideas about what makes psychological, biological, sociological systems flourish, in their respective fields of course. Someone who studies evolutionary biology might have an amazing understanding of why certain traits come about in an environment. A sociologist might have an elegant and unique perspective of how certain patterns of interaction within cultures leads to specific group behaviors. A psychiatrist might have a theory about how we think, maybe how certain childhood exposure leads to the

development of mental illnesses later on in life. A philosopher might have these ideas beyond the reach of psychology about how we think, maybe about how our subconscious structures itself. With all of these different people's perspectives and their adequate understanding of the dynamics of their respective professions, with the right tools could they model it? Let us say that the goal here is a human level intelligence, would it not be beneficial for both the person and the field of AI as a whole to give these geniuses tools to model their theories. After all, if the goal is to mimic human intelligence surely computer scientists are not the experts on how we as humans work. This right here is the explicit reason we need to have input from people outside of the computer science domain, because it is really not a computer science question that we are trying to answer. We need people with the understanding of people and their complexities to solve the problem of creating intelligence.

The most important part of coding, writing a book, playing music, is being able to express yourself or your ideas through it. One cannot judge Shakespeare on his playwriting ability from a spelling test just as one cannot judge the potential of a beginner programmer by how fluent they are in a coding language. Obviously, a good understanding of the English language was absolutely integral to Shakespeare being one of the best playwrights of all time. And true for amazing programmers as well, to reach one's true potential it is necessary that they develop fluency in a language that machines can interpret. The potential of the programmer is not held in their ability to write code, but their ability to express their ideas through it. I think that people genuinely do have intellectual predispositions one way or another in the intellectual landscape. Different people are wired differently, this is to say that I think that certain types of people are drawn towards art, towards language studies, certain types towards physics, and certain types towards computer science, and many other intellectual domains of course. All of these different people however were probably taught how to read and write in a language even though the person more inclined towards language studies might have picked it up a bit faster and shown more initial proficiency. This does not change that the purpose of learning a language was to express their ideas through it, and to communicate those ideas. I think that we should look at coding languages the same way, although those inclined towards computer science may show more initial proficiency, it does not change the fact that the core purpose of coding is for people to express their ideas through it.

The most elegant advancements in AI have been modeled off of nature in its different aspects, including the creation of deep learning itself. One of my favorite innovations in deep learning is the Generative Adversarial Network (GAN) created by Yann Lecun in 2014. In a min max game, two networks work to almost outsmart each other. Imagine a burglar and a sheriff, they both start out really bad at their jobs, but as one gets better it forces the other one to. The burglar's main objective is to not

get caught by the sheriff and the sheriff's main objective is to catch the burglar. The burglar is forced to come up with better and better solutions as the sheriff gets better at his job. In the language of deep learning, the burglar is a neural network that generates images and the sheriff is a neural network that gets better and better at guessing if the images are generated from the burglar or selected from a preexisting set of images. It feels a bit like natural selection to me, how as the environment evolves and becomes more complex, the life forms must evolve to keep up and survive in the environment. Environment in this context means everything an organism might have to deal with including other species. I do not claim to know the motivations behind Lecun devising this method of image generation, but I do know that thinking about it from a biological perspective intuitive and useful in its own right. Perhaps more intuitive to someone who has a passion for biological systems and evolution rather than computer science.

Take a step back and think, what do we think about when we think of artificial intelligence? Robots. Agents that live in our world and can navigate through it, perhaps evil and with hidden agendas, but for now just robots designed to do specific tasks. What could be a solution to this problem of teaching a robot how to interact with an environment? Maybe a good way to deal with this problem is to reinforce certain behaviors, good or bad, analogous to how psychological reinforcement works in humans. This method of machine learning is called reinforcement learning, and when it learns using deep neural networks it is often called deep reinforcement learning. This nice connection between the psychology of reinforcement and the neurology from the deep neural networks is what really fascinates me about this field, and what inspired me to write about this for my final project.

One thing that is unavoidable in explaining something like this is mathematics. If you are not inclined towards mathematics or just hate math in general, it may be hard to keep up with what I am saying throughout, especially towards the end. Think of math as a bridge between intuition and code, we need mathematics to define particular laws and conditions for the models and environments, but they are representative of broader more intuitive ideas. This intuition may get a little bit lost in all of the calculus and greek notation, but do not let that stop you from thinking about it until you understand, it takes practice. For example a huge concept for learning a policy is a gradient, which is fairly complex mathematically, but a gradient is simply a way to adjust variables to most quickly increase or decrease some function. In the context of policy gradients it has some psychological intuition: in which direction can we change the way we are doing things right now to most quickly approach the optimal way of doing them. I will do my best throughout to explain the mathematics to the best of my ability, but I will not linger on any particular concepts for too long.

Deep Reinforcement Learning:

So what exactly is reinforcement learning or deep reinforcement learning (DRL)? I think that there are a few ways to phrase a definition, from a psychological perspective to a mathematical one, but perhaps the most intuitive is the psychological one. Prior to the invention of computers as we think about them now, psychologists and behaviorists were studying the reinforcement of behaviors in animals as a consequence of all sorts of conditioning. One of the most influential American psychologists was B.F. Skinner, and the idea that he brought to the forefront of behavioral psychology was that behavior is determined by its consequences. One might look at this as an optimization of its behavior towards the environment to best fit the environment. Certain behaviors are enforced, these behaviors are those that lead to a positive response, and other actions or behaviors, are avoided. Pavlov's famous experiments with his dogs in the fields of behavioral science have shown us that we can manipulate these biological systems with neutral stimuli. How exactly these stimuli become connected to experiences, and how exactly in our brains experiences have molded our behavior are queries not completely understood, and professionals in these fields will admit it too.

Let's say that we wanted to create a robot named Tim that walks around a world full of dangers: Lava, quicksand, or predators. This hypothetical world also has things in it that the robot needs to take advantage of to survive, maybe it's some water or food. For this robot, Tim, to be successful in the world, he must survive, and his performance is measured by how long he survives. Wouldn't it be reasonable to design this robot's brain in a way that reinforces behaviors that lead to its ultimate survival. Reinforcement learning algorithms will learn how to optimally interact with the world in some trial and error fashion. It may take many iterations of the robot navigating the world, but eventually with the right RL algorithm behind it, the robot will develop some intelligence about how to navigate the world without dying. We might use deep neural networks (DNNs) as a kind of brain for the robot, with the mechanism of reinforcement being signals from the environment. The use of DNNs is more for important pattern recognition, it gives the robot a tool to distill important patterns from its initial high dimensional observations. This implementation of DNNs into the robot is called deep reinforcement learning. Before we go any further into this specific aspect of RL, there are some core ideas, and clever tricks that need to be understood prior to the main topic of the book.

The general term in RL for the robot in this example is an agent, and the world in which the agent lives is called the environment. The words agent and environment are definitely reflective of their roles as a part of a RL system. The agent sees the environment, or a part of it, and then decides which action to take moving forwards. The environment will likely change as a result of this action, but the environment can also change independent of the agent's action. In addition to

seeing a new state in the environment, the agent receives a reward. This reward is some number that is indicative of how good or bad the state is. Typically the agent has some objective to maximize the cumulative reward over the game, this is called the return.

I want to be clear for all of these definitions, you can search them up online to find a clear, concise, to the point definition. My purpose in writing this book is to spark intuition about these topics so that somebody might realize that their particular theories are in the applicability realm of RL or DRL. I am very well aware that there are many more reputable sources than myself, and those sources are where I learned these definitions myself. I am not here presenting myself as an expert, but wanting to reach a crowd not typically inclined towards computer science as a discipline. Because my intended audience are people who are not as inclined towards scientific diction and mathematics, I will prioritize the explanation in the domain of their intuition.

States and Observations

The terms state and observation are commonly interchanged, especially in the Spinning Up literature. However, it is important for beginners to understand the distinction between them to form a more full understanding of RL. A state in the context of RL is the entire description of the environment whatever the environment happens to be. An observation is either a partial or complete description of the state. This distinction is important when we are thinking about things with partial observability. In trying to teach the game of poker to a RL agent, the agent would have a partial observability of the state of the game. The rest of the cards in the players hands are still a part of the state and have consequences for future states, but they are unknown to the agent due to partial observability. If you think about it from the robot example earlier, the state would be the whole world, and the observation would be the part of that world the robot could see. We still must keep in mind that we have to translate these ideas back into things that computers can read, so these states and observations are often expressed by vectors, matrices, or tensors.

It can be confusing in literature what exactly the distinction is between a state and an observation, and the picky details really do not matter too much in the larger scheme of things. For example when an agent decides to take an action given a state, it really acts on the observation of the state and not the state itself. However in the context of many problems the state is identical to the observation so this detail may seem unnecessary. It is still good practice to understand the difference of these the two ideas when thinking about it intuitively. We as humans cannot really know everything about a state in the environment, and it is our pursuit to strengthen our observation of it.

A trajectory is another important term, and it is fairly self explanatory. It is a list of states and actions that the agent takes in a given episode of a game. The trajectory of an agent could look something like: state 1, action1, state2, action2 … up until the termination of the episode. The termination of the episode is dependent on whether the agent is in a terminal state, in the robot example it would be if he died. In a game like ping pong, a terminal state would be either the agent winning or having the ball pass the slider.

Action spaces

The action space can be thought of as the set of all possible actions. For PacMan, this would be something like up, down, left, or right. For ping pong it would be something like move the paddle up or move the paddle down. Notice that in our robot example it is not quite this simple, the robot has more autonomy than going just up, down, left, or right. The robot could choose any combination of these, and maybe even change its speed. Trying to put all of these possible actions into a set is impossible, because there are an infinite amount of actions the robot could take. The first case is what is called a discrete action space, and the second is a continuous action space. Imagine a continuous action space from trying to balance a ball on a platform you can control. The action you can take is often some balance between completely shifting the platform one way and completely shifting the platform another, there exists this continuum of actions that are linked through space. If we were to write out some table of every action for this space, discretizing at some very small value, changes in value would be really small for small changes in the action space. It is important to understand that continuous action spaces are of this nature because we may need to differentiate some function of how well the agent does with respect to the action argument. We may need to figure out how changes in actions would result in the change of performance.

You might be able to see how this could cause a problem when trying to figure out what the best option is given a state. This is part of what makes learning with continuous action spaces harder than discrete action spaces, but also more interesting. Further into the book, this will be brought up again, and a better explanation will be given. Some algorithms can only be applied to action spaces that are discrete, and some only for the other.

Policies

A policy in a RL agent is just as it might sound. A way of life for the agent, a way for the agent to understand and deduce what action to take in a given state. One of the common ways to solve the problem of RL, and what a lot of this paper is about, is to optimize this policy in order to optimize the agent's performance,

judged by the return. This can be thought of as changing the way the agent processes the world and acts given a situation to most efficiently increase the final return of the agent. There are two different types of policies, deterministic and stochastic.

The distinction between deterministic and stochastic policies is legitimately important to developing intuition and fluency when creating new solutions or implementing other people's ideas into code. Something is deterministic if you can determine the outcome given an action. For example, if someone stabs someone in the heart you could determine with certainty that the consequence of this action would be death. For any philosophers or deep thinkers out there, there is this age old question of whether everything in life is predetermined, if humans have free will. Determinism is the philosophy that everything that happens, all events that have ever occurred and will ever occur, is predisposed to happen beyond the reach of our will as humans. A deterministic policy is a policy that arrives at one action with absolute certainty. With our robot example from earlier, a deterministic policy when tasked with choosing an optimal action would arrive at one particular action with absolute certainty. Where a deterministic policy will choose an action with certainty given an observation, a stochastic policy will output a probability distribution of actions given an observation. A stochastic policy is almost necessary for a chaotic, dynamical environments. Something like the stock market, where there exists almost a chaotic and unpredictable environment might be more easily learned from a stochastic policy. In a stock market-like environment, it is so much more efficient for the agent to learn and sample actions from a probability distribution created by a policy than to try to learn by picking an action with absolute certainty.

With this puzzling case of having a stochastic policy output the probabilities of taking a given action at any state in a continuous action space we use diagonal gaussian policies to choose actions. Think why this might be puzzling or confusing at first, if there are infinitely many actions available to take, how can we sample a single action? At first the policy will not know which actions are optimal to the performance of the agent, so one might think that the probability distribution is fairly spread out. Remember again, that there still exists definable actions like up, down, left or right, and the continuous spectrum of actions lies in some combination of the definable actions. The two variables that define the motion of the robot; the first being its horizontal motion, and the second being its vertical motion. So instead of thinking about a continuous action space of a confusing infinite spectrum of values, think about it as some combination of these variables. For geometric intuition, imagine some two dimensional grid that is representative of the action space, two dimensional because there are two variables. Each point on the grid describes an action that is some combination of horizontal and vertical movement. The goal here is to kind of be able to circle an area on this action map given a state, and choose an action from near that circle. At the beginning of the

learning process, the circle from which we choose the action is large, as the agent is not sure as to what actions lead to good returns. However as the game progresses the agent learns to choose better actions by shrinking this action circle around certain values in the action grid. It really is not a circle per se, but a distribution centered around a point with some variance comparable to the radius of the circle. Geometric intuition is lost when there are multiple different variables at play here. Imagine if additional to horizontal and vertical movement, we added it being able to control like noise output and size. This four dimensional space doesn't fit with our intuition as well. In this more complex case, there is still high uncertainty at the beginning, a high variance. An agent learning a policy in this case would be to find out what four dimensional value to center the distribution around, and become more and more sure by decreasing the variance around the mean of the distribution.

There are upsides to both stochastic and deterministic policies. The upside to most policy optimization methods is that they work for stochastic and continuous cases. A deterministic policy is normally denoted by μ and a stochastic policy by π.

Discounted returns and rewards to go

Remember that a return is a sum of all of the rewards across a trajectory. Rewards can either be positive or negative, and are the essence of reinforcement in agent behavior. Discounting rewards is the idea of assigning more importance to rewards close in the future than rewards far away. We as humans might actually do something similar to this, assigning values to rewards close in time rather than weeks away. Take for instance if someone offered you one hundred dollars how versus one hundred dollars a week later. Maybe in this same intuition it can be analogous to inflation, where literally in a few weeks that hundred dollar bill will not be worth as much. The agent's perception of a future reward might be flawed and not as reliable as one closer in time so it would naturally make more sense to trust in the closer one more. In addition to this nice intuitive property, it has a very useful mathematical property. Sometimes we predict the return by taking an infinite sum of the rewards. Having a discount attached to rewards in the future ensures that the sum converges to a finite number.

In order to reduce the variance of a policy gradient, we use these rewards to go function instead of the entire sum or rewards. We only want the actions taken after the reward to factor into how the agent processes the world. This rewards to go function sums up the rewards from a single time step in time until the termination of the episode. The purpose of implementing the rewards to go, is to only consider the effects after an action was taking when updating the policy. If you remember from earlier, B.F. Skinner realized that changes in actions are a direct

result of their consequences in the environment. It would not make sense to treat a state or reward early in an episode as a consequence of a later action.

Value functions

Almost every RL algorithm uses some variation of a value function. A value function will estimate the value of taking an action in any given state. This value function has many different variations, there is the infinite discounted return or the undercounted finite horizon return. Take again the example of a robot living in some world, this value function would compute the value of an action by taking that action then determining every reward that would happen if the policy was followed. Given some policy π that the robot uses to map states to actions, the value function will give the expected return of the agent following this policy. Notice that this makes sense in the deterministic case, where each state gives exactly one action. In the stochastic case, it does not seem to make as much sense, it seems as though this return could have many different values because of sampling from a distribution of actions. Many different variations of this 'value function' exist in the realm of deep RL algorithms. There are four main types of value functions that are completely necessary to understand before moving forward into other deep RL topics: on-policy value function, on-policy action-value function, optimal value function, and optimal value-action function.

The on-policy value function is saying that with some current policy π, if we continue to abide by that policy the value function is the value of that current state. This is a great way of taking the pure value of the state so the agent can have another tool to construct a proper view of the environment. Where the on-policy value function is on instruction of the current policy, the optimal value function of a state is if the agent were to follow some optimal policy.

$$V^{\pi}(s) = \underset{\tau \sim \pi}{\mathrm{E}} \left[R(\tau) \,|s_0 = s \right]$$

The on-policy action-value function is the infinite discounted reward return of taking an action, then following a given policy. Let's say that Tim the robot is in the middle of learning a policy, and we want some measure of how good an action is. Of course, this prediction's legitimacy will be limited by how good the policy in place is. Let's say for simplicity's sake, the robot has a discrete action set of up, down, left, or right. We could estimate the value of each action like we did with just the value function. Additionally, much like the optimal value function, there is an optimal action-value function that, given any arbitrary action, will output a value. In Tim's case, he would weigh each action based on the action-value function dependent on the current policy and assign each action some value. One

$$Q^{\pi}(s, a) = \underset{\tau \sim \pi}{\mathrm{E}} \left[R(\tau) \,|s_0 = s, a_0 = a \right]$$

might imagine that if the agent knew the optimal policy it could compute the optimal value and the optimal action-values from any particular state.

One of the biggest branches of RL is called Q-learning, dealing with learning an optimal action-value function. Just as the optimal policy function might be approximated by a neural network, so can the Q function. Where the policy π is parameterized by the greek letter θ, the parameters in the value function are denoted by ϕ. The value of any given state is just as it might seem, how good or bad being in a certain state is. Winning a game might be defined simply as seeing as many of these high valued states as possible. You will see later that valuing a state in a stochastic environment, especially a complex and dynamical environment is challenging. A value is under a given policy, and given a stochastic policy, there are many different trajectories the agent could take from any state following the policy. The value of a state is what is expected, and is computed by taking every possible trajectory and weighing the return of that trajectory against it. It is infeasible, a waste of time, to compute all of this. Deep Q-learning uses neural network approximations to estimate the value or quality function.

Both the action-value function and the value function obey the equation to the right, which when looked at can be understood without too much thought. The reward from a state and action added to the value function from the next state is equivalent to the value of the current state. The same is true with the Q function; The quality of taking any arbitrary action in a state is equivalent to the reward of that action added to the quality of the next state action pair. The only way in which the Q-function differs from the value function is that the Value function takes an action given by the policy and the Q function takes all possible actions and picks the best one. Q-learning is dependent on discrete action spaces, because it needs to compute the value function for states after every possible action. Policy optimization does not go the same route as Q learning, policy optimization attempts to solve the problem almost at the root.

$$V^{\pi}(s) = \underset{\substack{a \sim \pi \\ s' \sim P}}{\mathrm{E}} [r(s,a) + \gamma V^{\pi}(s')]$$

In many policy gradient problems, we do not need to get an exact value for each state, or state action pair. What we really care about when learning a policy is the relative value of an action, compared to all of the other actions the agent could take. How good a state is in general is the value of that state, which is approximated using a neural network for complicated environments. We really only care about if taking a different action will lead us to better results than the previous set of actions we were taking. Subtracting the Q function from the Value function gives us this estimate to how much better the action is, which is called the Advantage function. We want to reinforce policies that give us a positive advantage function, and decrease those that lead to bad actions."High-Dimensional Continuous Control

Using Generalized Advantage Estimation" gives a full mathematical intuition for an advantage estimation method called GAE. This specific advantage estimation function is what is used in following policy gradient algorithms.

Credit assignment problem

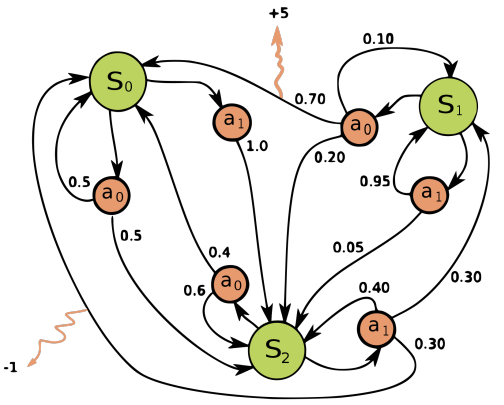
The credit assignment problem is both prevalent in RL and cognitive sciences. For what reason did the agent get a reward, and in turn which actions can we change to push up the cumulative reward. This is not only a reinforcement learning problem, a big part of analysis in many fields is seeing which components of a system led to it working the way it does.

I am a gymnast, if at a competition I bend my legs on the vault, fall on the floor, hit a perfect routine on the pommel horse, and do mediocre on everything else, then at the end I get some all around score. Maybe my mean score from the whole season is 70, if I get a 68.3 at this meet, the credit assignment problem is the problem of determining what exactly led to this. If I was to define rewards more frequently in the trajectory I could more accurately determine what led to the score I got and how to change my actions to benefit the final score. Let's say I got feedback after each routine, a 12.4 for high bar, a 10.3 on floor, and so on. Maybe then I could go back to the gym and train those events especially hard to prepare for the next meet. Even further, there are specific things that led to the scores I got on each of those events, the sum of the difficulties of skills minus deductions. If I got a positive reward for every hard skill I did and a negative reward for every deduction, I would be even more equipped with knowledge to go back to the gym and train.

The problem with credit assignment is that we humans do not always know how to assign a reward to a particular state. Take chess for instance, maybe we could provide more frequent rewards by assigning positive rewards to taking an opponent's piece, and negative rewards to losing a piece. When we think about this in depth, adding too many rewards may obscure the overall objective of the game. The agent might be focused on taking pieces while the opponent has sneakily put the agent into checkmate. This is the credit assignment problem, and it really boils down to how well humans understand systems and environments because we are the ones placing reward on being in states. In problems like these, where there needs to be a full understanding of the dynamics present in an environment, people well versed in their particular field will shine. This is where computer scientists do not have the advantage, they cannot understand a biological or a physical system better than a biologist or a physicist, and therefore are less equipped to assign rewards to states in the environment. I believe communication between all sorts of intellectual domains is so important to creating new technologies throughout deep learning.

Temporal Difference and Monte Carlo learning

In a Markov Decision Process (MDP), the future is independent of that past given all of the information in the present moment. This is to say that the best action from a given state to make in an environment is not dependent on previous actions, states, or rewards. For the agent to take advantage of the MDP properly, it needs to be able to see the whole environment, in other words the observation is the entire state. A flow chart would be a good example of an MDP, given any place that one might land on the flowchart there is a next step independent of how you got there. There might be many ways to get to that particular state in the flowchart, but they don't influence the future. In the image to the left, there are three different states represented by the green circles and two different actions. It may not be obvious, but this is an MDP. Once in a state, regardless of how the agent might have gotten there, there exists a constant reward structure. An effective way of maximizing a score in an environment is learning the MDP, then taking actions from that particular model.



MDP From Wiki User: Waldoalvarez

Maybe there is an environment, like blackjack, that does in fact have a MDP which structures the rewards system of the environment. But because the Agent cannot look at the other player's cards, it cannot observe the complete MDP. These types of situations are called partially observable MDPs. Additionally, there are environments with no underlying MDPs, where the past does have a lasting impact on the future. From this, there are two main systems of learning: Temporal Difference learning, and Monte Carlo learning. Temporal Difference (TD) learning has a built in assumption that the agent is in a fully observable MDP. TD learning provides a learning advantage as an algorithm because it learns an MDP from which it can take actions. Effectively, TD learning learns this MDP map. Monte Carlo (MC) learning is generally less effective in a MDP when compared to TD because it does not take into account the properties of the MDP.

We know that MC is not as good at solving MDPs as a TD algorithm would be, but what exactly is MC learning? A Monte Carlo learning algorithm will learn by sampling from an episode. After completing an episode, or dying in a simulation, the agent will look back over an episode of experience, find the mean return over trajectories and calculate the values of states accordingly. MC is a much more general family of algorithms than TD is. When we do not know anything about the

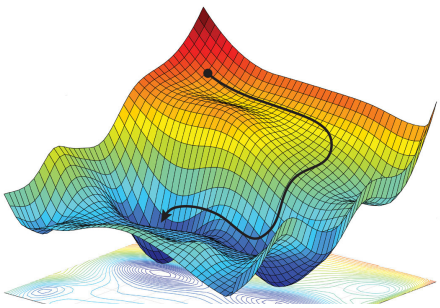
transition qualities within the environment, we must use MC because we cannot assume there is a MDP. It will update the policy after the entire trajectory is stored in memory, at the termination of the episode. Where MC will wait until the end of the episode, temporal difference helps develop a fuller picture for the dynamic of the environment by updating throughout the episode. Let's say that we are measuring the weather over time in Dallas. We have an alright model about how to predict the weather a few weeks into the future, let's say our goal is to predict the weather for week three using this model. It is week one right now, but we can actually change our model sometime during week two because the weather in week two tells us something about the weather in week three. These intermediate changes of the model are what make TD learning successful.

In TD learning, there is this concept of an error for each time step, often represented as δ. This error holds the information about how for how close the value approximation is to the reward structure in the environment. The value of state two should be equal to the value of state one plus the reward from state one. Minimizing this difference, the TD error, is how the value function will approximate the true value function of the environment.

Policy Optimization

How exactly we make a policy optimization algorithm is difficult. The general flow of ideas generally looks something like observations about the world or humans, the formulation of mathematical relationships between them, manipulation of formulas to work, and finally the translation of mathematics into something a computer is able to read. One of these ideas about how to optimize learning is called a policy gradient, a way that the artificial neurons in the deep neural policy network change themselves to most quickly change the behavior of the policy. Imagine it as many small steps shifting the parameters of the policy function closer and closer to an optimal policy.

Most deep neural networks are trained using gradient descent and are guided by some loss function. If a convolutional neural network, a type of DNN, is used for classifying images into different kinds of birds, the loss function would be some metric that tells the last layer of the net how wrong it is. Remember that the last layer is where all of the neurons come together to form some conclusions about the input. After the network is told how wrong it was in its conclusion, by stochastic gradient descent the network tunes all of its parameters in a direction that most quickly decreases the loss function. For this problem of birds, you might imagine some predefined solution space that exists for the problem at hand. With our neural network,



we are able to navigate through this high dimensional solution space, and approximate solutions by stepping towards a minimum of the loss function. This notion of approximation and not perfection is important to keep in mind, and it explains why we need things like a very small learning rate, as not to overstep this preexisting function that we are navigating. In this colorful image to the right from sciencemag.org, you might image this as a solution to a neural network with two parameters. Each combination of these parameters has a corresponding loss value, maybe how accurate it is in classifying some phenomenon. For every instance, every place in time, the parameters of the net are at one point in this solution space, and we are able to calculate the direction to move these two parameters to most quickly decrease the loss value of the parameters. Imagine you are an ant in a complex, hilly, cloudy environment and you want to get down the hill. You can see what is exactly in front of you, but no further. In this intuition, being the ant you would not want to walk for too long before looking again, because the environment will change. This is the gradient, it tells part of the story of the shape of the solution specifically a direction but nothing else.

In real problems, this is not a three dimensional space with only two parameters, it is likely a space with a few hundred dimensions. Although this might be hard to wrap your head around, understand that the idea of computing a gradient does not really change when more dimensions are added. In a policy network or a classifier network, there are still local minimums and maximums in the space of all different combinations of the parameters. Convergence means that we find one of these minimums or maximums. Take for instance, a network's parameters are in a local minimum, this makes the gradient zero and that point, meaning that any movement will increase the loss function. But does this mean that we have found the absolute best set of parameters to solve the problem? No. What we have converged upon is a local minimum, meaning that there might be dips in parameter space that have a lower loss value than the minimum we found. In machine learning terms, the process of approaching a local minimum is called training a network. When we begin the process of training we randomly initialize the parameters in the network, normally according to a normal distribution. This initialization corresponds to some loss value, and more importantly a different local shape of the loss function we attempt to solve. This is why, with the exact same gradient descent algorithm we can reach vastly different solutions for the same problem with different initializations. We assume that this loss function is always continuous and differentiable with respect to the loss. Given enough steps for a classifier and properly labeled data, gradient descent will converge on a set of parameters given any arbitrary initialization of parameters. For this reason it is important to do more than one trial to diagnose the effectiveness of an algorithm.

There are a few big differences between a policy network in a RL agent, and the typical DNNs you might see in a classifier. The goal is to minimize some function

in a classifier, to apply gradient descent iteratively in order to approach an optimal solution. In RL, the primary purpose is to maximize a score function dependent on the policy, not to minimize an error. Gradient ascent is taking steps to maximize a function where gradient descent is taking steps to minimize one. Just as we might measure the performance of a classifier with $L(\theta)$, we measure the performance of a policy with the infinite discounted sum of rewards from a given trajectory under π, called $J(\theta)$. This really is not as complicated as it might sound. Given the current policy we have, we measure how good this policy is by collecting the expected sum of the rewards, which we call the expected return. This should be a fairly clear concept, easily grasped. We want to capture this idea of an expectation in the language of mathematics; given this policy π what is the expected value of a state over the trajectory. The "E" symbol means an expectation and the π as a subscript means it is an exception under π. This expectation functional takes a weighted average of whatever variables are calculating the expectation of. This weighted average weights the return of a trajectory by the strength of the probability of that trajectory actually happening: $\pi(\tau;\theta)$. $\pi(\tau;\theta)$ being the probability of taking trajectory τ given the current parameters θ with policy π. So summing the essence of an expectation up, it is weighing the return of every trajectory on the basis of how likely it is to occur given the current policy parameters, and giving the sum of all of them. There are many different ways this trajectory could lay out, different actions and consequently different states, remember this is due to the stochastic nature of the policy. Note that the expectation is not a function, but a functional because it takes functions as inputs and outputs a series of random variables. A fair amount of mathematics, and an even greater amount of math relating to RL, has the property of taking intuitive ideas like expectation and defining them mathematically so that computers may use them. I think that anyone can have a pretty full understanding of the expectation function when pondering on it for a long enough time.

Let's think a little bit more about what taking a gradient means in the context of the expected return. What effects does changing the parameters of the policy have on the performance exactly? Well it is certain that there will be a different action distribution, and this different action distribution will certainly lead to a different state distribution. With changes to the policy, there are changes in the state distribution, therefore the gradient of the performance should really account for how the state distribution changes. Now arises the issue that the function of the environment is unknown, we simply cannot predict what exact changes to the state distribution given a change in the action distribution. There is an essential part of the dynamics of the return we must ignore to derive a policy gradient, which is why we must tread extremely carefully when making updates to the policy. To find a gradient and step in that direction is a linear approximation, which becomes more inaccurate for larger steps. Adding on top of this inaccuracy for higher steps is the uncertainty in how the state distribution will change, and larger and larger steps will

continue to increase that uncertainty. Keep this in mind as we move forward with finding the policy gradient.

Follow along with my explanation, and the mathematical derivation of the policy gradient. Step one is to define the performance, J(θ), as some expected value under x of *f*(x). Step two is to expand the expectation in its integral form, this integral creates a weighted 'sum' of *f*(x), which will represent the trajectory. After this, we move the gradient inside the integral as you see in step three. It is not optimal to have to compute the gradient over the probability p(x;θ) for a few reasons. First of all, we would like to convert this back into an expectation eventually, so we need a standalone probability distribution under π without the gradient. Secondly, it is mathematically extraneous, and when doing these calculations on a computer, a huge number of small probabilities accounts for a lot of the imprecision that comes naturally with float point numbers. Taking the gradient of the logarithm probability distribution actually solves both of these problems. The logarithmic is more numerically stable for computers, it keeps values from rapidly approaching zero which otherwise might result in some error. If the gradient of an argument is divided by that argument, it is equal to the gradient of the logarithm of that argument. This actually takes care of being able to convert the integral back into an expectation. In step 5, it is shown that the final gradient J can be written as an expectation under the policy.

$$
\begin{aligned}
\nabla_\theta E_x[f(x)] &= \frac{\partial E_x[f(x)]}{\partial \theta} \\
&= \nabla_\theta \int f(x)\, p(x \mid \theta)\, dx \\
&= \int f(x)\, \nabla_\theta p(x \mid \theta)\, \frac{p(x \mid \theta)}{p(x \mid \theta)}\, dx \\
&= \int f(x)\, \nabla_\theta log\, p(x \mid \theta)\, p(x \mid \theta)\, dx \\
&= E_x[f(x)\, \nabla_\theta log\, p(x \mid \theta)]
\end{aligned}
$$

This is a linear model of learning, meaning that for any instance of J(θ), we have expanded our vision just a bit to see what moving in any direction will do to the overall reward function. This is only accurate within a short time step because a linear approximation is not really a good estimation of the function. These iterative steps become more and more inaccurate as we increase the learning rate, α, off the function not only because the linear approximation is inaccurate for big steps, and with the uncertainty of the change in the state distribution. Recall from earlier that the state distribution has a significant effect on the reward, and this policy gradient derivation is not involved in the stated distribution. Because the function of the environment is unknown, there is uncertainty with how the state distribution will change. If the agent updates its policy by too much it can be detrimental to the success of the agent. The agent's policy is attempting to climb this hill in parameter space, and taking too large of a step is being thrown off of that hill. The agent might see states totally different to what it has been seeing, and the value of these states might be much lower than previously. This limits the policy updates to an extremely small learning rate, which

$$\theta_{k+1} = \theta_k + \alpha \, \nabla_\theta J(\pi_\theta)|_{\theta_k}$$

has the effect of slowing learning down considerably and makes scaling up to more complex environments unrealistic. The essence of trust region policy optimization and proximal policy optimization is to solve this problem by defining how far is safe for the policy to update.

A simple policy gradient

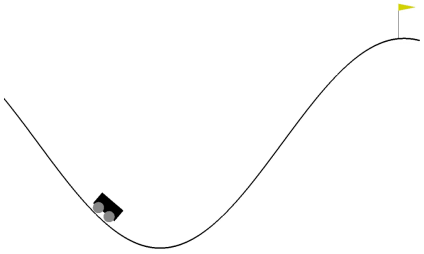
Mathematics is nice, pretty, accounting for things like infinite integrals. Translating ideas from the theoretical language of math into very real applications in code forces us to come up with solutions. These solutions are beautiful, the use of neural networks as function approximation for the value and policy functions, transforming theoretical ideas in math into tangible things that a computer can do. Something that might have not been so obvious earlier is that in fact we do need a neural network for the value function. It doesn't necessarily need to be a value function per se, it could be any measurement of how good a state is. Denoted with parameters ϕ, the value function is trained to approximate the rewards-to-go from any given state. Trained with your typical gradient descent algorithm, it attempts to minimize mean squared error between rewards to go and the old value function. While the policy network learns how to assign actions to states, the value network will learn how to assign values to states. You will see how these two networks work together to solve problems.

If learning python sounds like something that you might want to get into, I encourage you to go look into my code in my GitHub repository: "https://github.com/jackmcgaughey/reinforcement_learning". I think looking at code and trying to understand it is very helpful in understanding what is necessary to writing code. There are so many technicalities in programming, data types that only work with specific things. Sometime things will not work because the wrong type of list or array was used, and the only way to get over this hump is to simply put in the time and effort to learning.

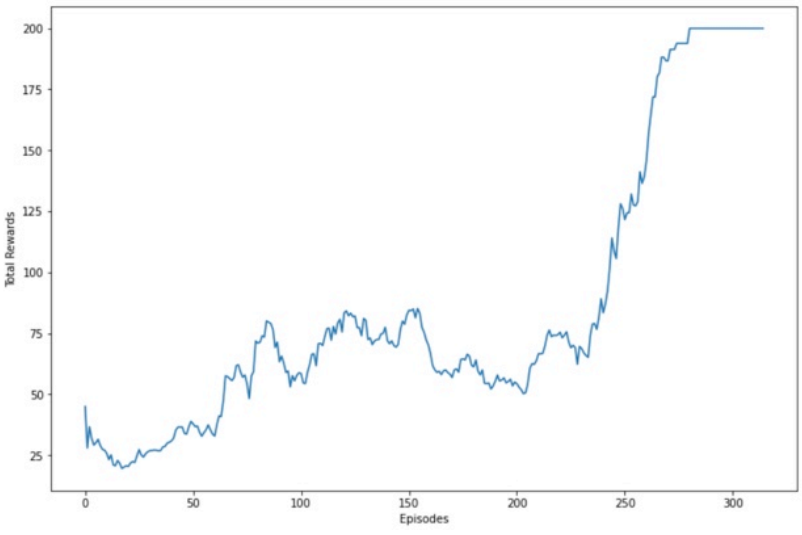
Actor-Critic is a simple way to combine policy gradients and value functions. It consists of two main components which hold neural networks, the actor and the critic. The actor takes steps given what the critic has to say about any state. Looking back to the derivation of the policy gradient, this term $R(\tau)$ is what serves to signify if the trajectory taken was good or bad, then the parameters are updated accordingly. An approximation for the expected return is the critic, and fittingly so, it critiques how good or bad a policy is on the basis of its trajectory. The critic is not always specific to being the expected return, it serves the purpose of being a critic in whatever way might be most beneficial to the algorithm as a whole. You could see the critic as an Advantage function as discussed earlier, a Q function, or a value function. The actor receives this information from the critic about how good the trajectory was under the current policy. The actor then takes steps to change the

parameters of the policy function in a way that moves the agent towards taking a better, higher valued trajectory. What is nice about this Actor-Critic architecture, is that it's really easy to understand and read in python, because most of the time they will have separate classes.

In my GitHub repo, you can find my implementations of the algorithms I did and even run them yourself if you want to. For the first one, I used an Actor-Critic architecture, and designed the algorithm for temporal difference learning. The environment that the agent played in is called continuous mountain climber. The goal of the game is to reach the top of the hill, but the only way the agent is able to accomplish this, is if it swings back and gains momentum to make its way up the hill. A funny thing happened when running my algorithm is that it never actually got a score above zero. This just goes to demonstrate that the agent's method of optimization might not be exactly the most optimal solution you had in your head. Instead of maximizing a positive score by reaching the top of the hill, the agent minimizes negative reward. The agent just sits at the bottom of the hill and tries to move as little as possible, this is how it creates an optimal solution to the environment. This temporal difference algorithm keeps track at every time step of how far the value estimate is from the reward given, and it attempts to minimize this loss over time.



Instead of fitting a neural network to be the value function, another method is to simply sample off of the environment to learn. This algorithm called REINFORCE is the second algorithm I wrote out in code, and it does not use a second neural network for the value function. I decided to try a different environment for my implementation of REINFORCE. From open AI's gym, I chose the classic Cart Pole environment. The goal for the agent is to balance the pole on top of the cart, which the agent has control over. Firstly, I collected the trajectories of the episodes; the observations, actions, and rewards. After calculating the discounted rewards to go, I used the policy gradient theorem to calculate the gradient. The agent was able to solve the environment, achieving a max score of 200 points after about 300 episodes consistently. To the right here is a graph of episodes versus the reward over that episode. It is evident that the agent is



learning how to maximize its score in the environment in that over episodes, the total reward goes up. It levels out when the reward reaches 200 because that is the maximum reward the agent can have. So what does solving the environment actually look like? It looks like balancing the pole on the cart for the whole length of the episode, for a good visual representation go to this YouTube link: "https://www.youtube.com/watch?v=ooOA0q2skmE".

Other methods of policy optimization

If you take a look at either one of my scripts in my repository, you'll see how low the learning rates are. I mentioned this problem earlier, these algorithms are very unstable for larger learning rates. What TRPO and PPO do, is they define boundaries for how far the policy can update given the current policy. Making too large of an update in parameter space can completely knock the agent off of the path it was learning on. It is comparable to climbing up a rocky, steep, dangerous hill. Just looking at the equations for normal policy gradients, by taking the gradient of the expected return with respect to the parameters of the neural network the direction to update is decided, and then the step is taken. The step size, α, is predetermined and the same regardless of the policy. In trust region policy optimization, the maximum step size is determined, then the optimal direction to step in is determined. But why is this method better in terms of updating policies? Partially because it is dynamic, the agent can change the size of this trust region based on the performance of the policy at any given time. The new policy, after an update, should not be too different than the old one.

What size the trust region should be is the main problem TRPO and PPO solves. Remember one of the key problems with updating too much is the change in the state distribution as a consequence of changes in the action distribution. If the rewards structure of the environment is an unknown, which often it is, then there is no known transition model for states. The policy's effect on the action distribution is derivable, but the policy's effect on the states the agent sees is not. The policy gradient theorem is quite a clever trick in the sense that it provides a derivation of the performance with respect to the parameters without differentiating the state distribution. TRPO and PPO (in some instances) measure the divergence between the old policy and the new policy in terms of the state distributions. I got this image to the right from Open AI's spinning up documentation, an amazing and helpful site for learning about policy optimization. This divergence is a KL-divergence, which is a measurement of the "distance" between two distributions. In this case, the KL-divergence is measuring how different the state distributions under policy π with the

$$\bar{D}_{KL}(\theta||\theta_k) = \underset{s\sim\pi_{\theta_k}}{\mathrm{E}}\left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)\right].$$

new and old parameters θ. This dynamic safety measurement keeps the agent from overstepping in parameter space.

If the update safety net is expressed in terms of the KL-divergence, then there must be a measurement of how good or bad the new policy is. By using the same data collected under the old policy, the performance between the new and old policies are compared by dividing new by the end. Taking the expectation of the product of the comparison term and the advantage function for given actions and states, gives the expression of the surrogate advantage. To the right, also from the spinning up documentation might be a bit easier to understand. Think of the KL-divergence term as a kind of a constraint to the surrogate advantage. But to be a constraint, it would need to be constrained to some value. This KL-divergence term must be less than some value. So all and all, a perfect TRPO update would find the best set of parameters θ which maximized the surrogate advantage but the KL-divergence limit condition is still satisfied.

$$\mathcal{L}(\theta_k, \theta) = \underset{s,a \sim \pi_{\theta_k}}{\mathrm{E}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right]$$

Theoretically, this is great and easy to understand. But implementing this into code... is quite a challenge, and a reason why I didn't include it in this paper. Basically, it involves approximating the surrogate advantage and the divergence using a Taylor series expansion. If you really feel like you want to dive into the math, then I recommend reading Trust Region Policy Optimization by Schulman. There is about five pages of math, too complicated for me to sit and sift through. What is more feasible to workout and write into code is the Proximal Policy Optimization algorithm.

PPO solves the same problem that TRPO solves, but in a first order way. There are technically two different PPO methods, there is the KL-divergence penalty and the clipped advantage. The one most easy to understand, and the one which is used the most is the PPO-clip. PPO has no need for this measurement between distributions made by the policy, it has no KL-divergence term, PPO limits moving too far away from previous policies in a different way. Still it is help to visualize this as making some sphere of how far or close the policy is allowed to update. There is a structure of clipping this objective function that constrains how far it can update, it creates a boundary. It will be allowed to update to anything under this condition, and the algorithms are structured in a really simple and easy to understand fashion. There is somewhat of a ceiling, that does not reward or incentivize going past this ceiling.

Citations:

aiSUTRA. *Medium*, 2020, medium.com/@sendamailtokarthik/gradient-descent-611d696451e0.

Breloff, Tom. "Deep Reinforcement Learning with Online Generalized Advantage Estimation." *Deep Reinforcement Learning with Online Generalized Advantage Estimation – Tom Breloff*, 2016, www.breloff.com/DeepRL-OnlineGAE/.

Hui, Jonathan. "RL - Proximal Policy Optimization (PPO) Explained." *Medium*, Medium, 29 Dec. 2018, jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12.

JoshAchiam, Josh. "Welcome to Spinning Up in Deep RL!¶." *Welcome to Spinning Up in Deep RL! - Spinning Up Documentation*, 2018, spinningup.openai.com/en/latest/.

Schulman, John, et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation." *ArXiv.org*, 20 Oct. 2018, arxiv.org/abs/1506.02438.

Schulman, John, et al. "Proximal Policy Optimization Algorithms." *ArXiv.org*, 28 Aug. 2017, arxiv.org/abs/1707.06347.

Schulman, John, et al. "Proximal Policy Optimization Algorithms." *ArXiv.org*, 28 Aug. 2017, arxiv.org/abs/1707.06347.

Schulman, John, et al. "Trust Region Policy Optimization." *ArXiv.org*, 20 Apr. 2017, arxiv.org/abs/1502.05477.

Schulman, John. *HIGH-DIMENSIONAL CONTINUOUS CONTROL USING GENERALIZED ADVANTAGE ESTIMATION*. 2018, arxiv.org/pdf/1506.02438.pdf.

Schulman, john. *OPTIMIZING EXPECTATIONS: FROM DEEP REINFORCEMENT LEARNING TO STOCHASTIC COMPUTATION GRAPHS*. 2016.

Simonini, Thomas. "An Introduction to Policy Gradients with Cartpole and Doom." *Medium*, FreeCodeCamp.org, 5 Feb. 2019, medium.com/free-code-camp/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f.

Sutton. *Policy Gradient Methods for Reinforcement Learning with Function Approximation*. 2000.