The Fundamentals of Reinforcement Learning

Joel Keyser
Department of Software Engineering
University of Wisconsin – Platteville
keyserj@uwplatt.edu

Abstract

In 1995, Gerald Tesauro published a paper[9] describing his successes with creating a self-learning artificial intelligence program that played Backgammon at a level surpassing the average human. Eighteen years later, a different program, made by DeepMind, was able to learn and play six of seven tested Atari games at levels higher than any previous AI [5]. It was an impressive achievement, but even more impressive is that the program did this without changing its own core architecture or learning algorithms for each game. Fast forward three years, and DeepMind released another groundbreaking program[6] that was able to beat the world champion of the game Go, a game in which there are roughly 10^{170} possible board states (for comparison, there are an estimated 10^{80} atoms in the known universe). Between these three scientific breakthroughs in the field of Artificial Intelligence, there is one core similarity: the application of Reinforcement Learning.

This paper covers the fundamental concepts of Reinforcement Learning that allow the accomplishment of such incredible feats. Additionally mentioned are some important design considerations when applying Reinforcement Learning in progressive software projects.

Introduction

The topic of Reinforcement Learning is a specific one, and to begin to understand how it works one must first understand its scope within the umbrella of Artificial Intelligence.

Artificial Intelligence (AI) is the idea that machines can be programmed to perform activities that require a human-like thought process. A few different branches of AI are recognized, though they are mainly split between programs whose thought processes are hard-coded and programs whose thought processes are learned. The branch for programs that learn their thought processes is called Machine Learning. Machine Learning is split into three major categories: supervised learning, unsupervised learning, and reinforcement learning.

The first two types of Machine Learning, supervised and unsupervised learning, involve the program being given pre-existing datasets to learn from. In contrast, Reinforcement Learning refers to machines that learn based on experiencing situations themselves. An example of

Reinforcement Learning is a program that plays blackjack[8] and, as it plays more games, learns which cases that the better move is to “hit” rather than “stay”, and vice versa.

Markov Decision Processes

To more accurately discuss Reinforcement Learning, each applicable scenario is represented as a Markov Decision Process (MDP) [4].

In a scenario, there is an agent that observes an environment. The environment is in a specific state and the agent can take a specific action. Taking an action transitions the environment to another state and the agent receives a resulting reward that is not necessarily positive. A policy is what the agent uses to decide which action to take, given the current state of the environment.

An MDP consists of the sets of actions (action space) and states (state space), combined with the rules for transitions and rewards. One episode of an MDP is a sequence of states, actions, and resulting rewards from a start state to the final state, called the terminal state, and is illustrated by the equation

$$episode = \{s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_{T-1}, a_{T-1}, r_{T-1}, s_T\}$$

where the subscripts are the time-steps, and time-step T is the time-step of the last, or terminal, state of the episode. It is important to note that an MDP has the Markov property, meaning that the probability of transitioning to a specific next state is based only on the current state and executed action in that state.

Imagine a simplified scenario with a self-driving car. The action space could be to accelerate, brake, or do nothing. The states could include the current GPS coordinates, as well as the current velocity and acceleration of the car. Given these parts of the state as well as a chosen action, the next state could be determined. Perhaps the reward is one point per foot-closer-to-destination; it can be anything chosen by the implementer that lets the agent know what its goal should be. An episode, then, could be the sequence of all states seen, actions taken, and rewards received in order to drive from home to school.

Note that if the velocity and acceleration were not included in the states, this scenario would not be Markov because the next state could not be determined from only GPS coordinates. Knowledge of previous states would be required so that the velocity and acceleration could be calculated in order to determine the next state.

Value Functions

Each MDP has a reward function, whether or not it is known, that describes the amount of reward that an agent should receive, given that a specific action is taken in a specific state. However, in most scenarios, what matters for the agent is the total reward received by the end of

the episode, not just the immediate reward for the current state. In chess, for example, it matters little that white can take black's queen if that sets black up to put white in checkmate.

This is where value functions come in. Value functions are used to estimate the total reward received from a specified state until the terminal state. There are two types of value functions: a state-value function $V(s)$ and an action-value function $Q(s,a)$. The state-value function determines how good it is to be in a specific state, and the action-value function determines how good it is to take a specific action in a specific state. These value functions are either an array or an approximation function.

Using an array is simple enough: if a state-value function is an array, each state would have its own estimated total future reward in the array; conversely, an action-value function would be a two-dimensional array, because there is a value for each state-action pair.

An approximation function is a bit more complicated, and can be calculated using a number of methods (linear approximation with a feature vector, neural networks, etc. [8]) that will not be discussed here. Notably, these approximation functions are typically differentiable and are functions of a set of parameters $\vec{\theta}$; these properties are important for learning because they allow the use of gradient methods, which will be covered later.

Policies

Behind every action is a thought process. In Reinforcement Learning, the decision-making function that simulates a thought process is called the policy. The policy $\pi(s,a)$ calculates the probability of taking a specific action given a specific state. Unlike value functions, it is not common for the structure of policies to be an array. Simple policies usually involve being Greedy, meaning that the decision of which action to take is determined by whichever action that the value function says has the highest value. Complex policies, on the other hand, are similar to the approximation function of value functions in that they are functions of some set of parameters $\vec{\theta}$ and that they are typically differentiable. Additionally, as with the approximation function, the important parts are these two properties, and they too can be calculated using neural networks or linear approximation.

Components of the Learning Algorithm

The core of the algorithm that the agent uses to learn can be slightly different depending on specifically what the agent wants to learn. There are three goals the agent can have: to learn the true value function (referred to as value-based), learn the optimal policy (referred to as policy-based), or learn both the true value function and the optimal policy (referred to as actor-critic methods).

If the goal is to learn the true value function, then the learning method involves using a simple (greedy) policy, observing the results of the actions taken decided by this policy, and updating

the value function towards the observed results. “Updating towards results” in this context means to change the value function so that, in the future, its estimate will be closer to the results that were seen. The idea behind this goal is that, when the true value function is achieved, the agent can just use a Greedy policy in order to make the best decisions.

In the case that the goal is to learn the optimal policy, the learning method involves stepping through an entire episode to observe each individual reward, then updating parameters of the policy towards the rewards received; meaning, if there is high reward, the parameters will be adjusted so that the action performed to receive that reward is more likely to occur again, and vice versa for if there is low reward.

The third goal, which is to learn both the true value function and the optimal policy, uses a bit of each approach. Actor-critic methods have their name because the value function is treated as a critic and the policy is treated as an actor. The policy is updated based on the direction that the value function tells it, then the value function is updated towards observed results. Both the policy and value function improve over time, and it can make the learning process more efficient.

Actor-critic methods are generally more used than value-based or policy-based methods, but it is worth mentioning that some specific cases do exist in which the less-used methods are more suitable. Regardless of which of these three methods are used, the learning algorithm involves two vital components: target calculation and updating.

Target Calculation

As mentioned before, updating entails a change towards a value. Well, the value that is updated towards is called the target. The target always includes some use of experienced reward because the target is supposed to be a value that is more accurate than the estimate obtained using the currently most accurate value function. There are three different methods used for calculating the target: Monte Carlo, Temporal Difference (0), and Temporal Difference (λ).

Monte Carlo Method

The idea behind Monte Carlo is that the most accurate target is the one that is exactly the total reward from an actual experience starting in a specific state. Therefore, to calculate the target, at least an episode’s worth of experience is accumulated and then an algorithm similar to the one seen in Figure 1 below is executed.

Repeat until terminal state s_T (end of episode)
    Observe state s_t at time t
    Take an action a_t based on the state s_t using policy π
    Observe immediate reward r_t
Repeat for each time t
    Calculate total received reward R_t at time t by accumulating immediate rewards r_t to r_T
    $Target_t = R_t$

Figure 1: Basic Monte Carlo algorithm for calculating targets.

In this algorithm, a target is calculated for each time-step *t* in an episode.

To depict how the Monte Carlo target calculation algorithm works, picture an agent playing Pac Man. The reward is just the value of the score, and each fruit eaten gives one point. When the episode starts, the agent moves ten times, and Pac Man eats a fruit for each of the last five. On the tenth move, Pac Man runs into a ghost, causing a game-over and ending the episode with five points. Using the first loop of the algorithm in Figure 2, the immediate rewards are tracked during the episode and valued as:

$$Rewards = \{r_t = 0 | 2 \leq t \leq 5, r_t = 1 | 6 \leq t \leq 10\}.$$

The total received rewards, calculated in the second loop of the algorithm in Figure 2, are:

$$Total\ Rewards = \{R_t = 5 | 1 \leq t \leq 6, R_7 = 4, R_8 = 3, R_9 = 2, R_{10} = 1\}$$

because they are, at time *t*, the total values received from step *t* onwards.

Though Monte Carlo can be used to calculate the most accurate target, there are specifically two noteworthy downsides to Monte Carlo. One, it requires running through an entire episode step by step before the target can be calculated. If episodes are long, then the target will be calculated infrequently (when going through multiple episodes). Two, it does not account for the intermediate steps of the episode besides through the final result. Picture the scenario of a game of checkers. The agent makes a decision in the first few plays that results in the opponent performing a triple-jump to take three of the agent's pieces. However, the game goes on for long enough and the agent manages to crawl back and win. Well, with Monte Carlo, the target will be calculated at the end of the episode, and will basically imply that, since the agent won, all the moves that it chose were "good", when, in reality, the moves leading to getting triple-jumped were probably not the greatest. Additionally, since the target is determined by so many different steps, it will have a high variance.

Temporal Difference (0) Method

Temporal Difference (0), commonly referred to as TD(0), can be used after just one action is taken, regardless of whether or not the next state happens to be the terminal state. This way,

targets can be calculated after each step, and therefore can reflect the individual value of intermediate steps. Since only one step is used for each calculated target, the variance of the target is low.

The theory behind TD(0) is that the target can be determined by taking one step, observing the immediate reward r_t, and adding the value function's estimate at the next step $t + 1$. Mathematically, this is represented as $target = r_t + V(s_{t+1})$. Since there is one step of actual experience taken, this target is "better" than just taking the value function's estimate at step t, $V(s_t)$. Note that this is shown using a state-value function; using an action-value function Q(s, a) is similar, except a next action a_{t+1} is selected for determining the estimates.

Back to the Pac Man example, assume that the state-value function is an array and it has somehow gotten to the point where its values are:

$$V(s_t) = \begin{cases} 5 | 1 \leq t \leq 7 \\ 3 | 8 \leq t \leq 9. \\ 1 | t = 10 \end{cases}$$

For example, at $t = 8$, the total future reward calculated using TD(0) is $r_8 + V(s_9) = 1 + 3 = 4$. The value is slightly off, seeing as the Monte Carlo method calculated the actual total reward at the same point to be 3. However, since the TD(0) method can determine a target at each step of an episode, an update can be made at each step of the episode, and therefore the agent can improve quicker and account for the value of intermediate steps. Think about it this way: in an episode with ten steps, ten less accurate updates with lower variance can be made, rather than just one more accurate update with higher variance by using Monte Carlo methods. So an agent using TD(0) will learn quicker than an agent using Monte Carlo, but since the TD(0) method is less accurate, the learning may not be as reliable as it otherwise would be using Monte Carlo. There is a clear trade-off here: accuracy for time efficiency and lower variance.

Temporal Difference (λ) Method

There exists a method that can have a bit of all the benefits of accuracy, time efficiency, and low variance though: Temporal Difference (λ), or TD(λ). The idea behind TD(λ) starts with the thought of combining the Monte Carlo and TD(0) methods. For example, take three real steps to observe three immediate rewards, then use the value function to estimate the total future reward from that point onwards. Considering all the different ways to combine Monte Carlo and TD(0) (e.g. take two steps, estimate the rest, take three steps, estimate the rest, …) leaves many different combinations of possible levels of accuracy, time efficiency, and variance. TD(λ) is an attempt to use the most advantageous combination.

Visualize that a target is created for each of these different ways to combine Monte Carlo and TD(0) (target for one step and estimating the rest, target for two steps and estimating the rest, …). Then, each target is weighted based on a constant λ, with the targets that estimate more of the future being weighted higher than the targets that estimate less (and step further). This choice of weighing gives more value to the intermediate steps, trading off the accuracy of the more

Monte Carlo-like targets. The resulting weighted targets are summed together to form a weighted average that is the desired target.

Visualizing TD(λ) in this manner makes it easier to see that the method tries to get the best of low variance and high accuracy, but it really suffers in time-efficiency. This visualization of TD(λ) is called Forward TD(λ) and it is never used in practice because there is another method, often referred to as Backwards TD(λ), that is proven to achieve the same target values[8] without suffering so much time-efficiency.

Backwards TD(λ) does not calculate the target directly. It uses a TD(0) target and keeps an eligibility trace that is used later, during an update. The eligibility trace is usually an array with a value for each state. When a state is seen during an episode, the eligibility trace value for the state is incremented. During each step of the episode, after incrementing one state's eligibility trace, every state's eligibility trace is decayed. In this manner, the eligibility trace will account for states that are seen more frequently and more recently so that, when an update is performed, such states will receive more credit for the reward that is received.

Performing the Update

The learning algorithm, up to this point, has actually accomplished no learning. Agents learn with each update. Depending on the methods used, updates can be performed on the value function, policy, or both; however, it is not as important to distinguish the update between these as it is to distinguish between updating a function that is an array and updating a function of some set of parameters $\vec{\theta}$.

Updating an Array

Below shows an example[8] of an assignment that is used to update an action-value function that is an array:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)].$$

As seen by the $r + \gamma Q(s',a')$, this assignment is updating the value function towards a target calculated using TD(0) methods, and therefore is likely used for each step of an episode. This assignment can be described as changing the action-value function by an amount determined by the difference between the target and the action-value function itself. Hence, if the target is higher, the action-value function will be increased in order to be more accurate to the target. The α and γ variables are both values inclusively between 0 and 1. α determines how much of a step should be taken towards the target, and is used for decreasing variance (because if the target is entirely considered, and targets have much variance, the updates could go up and down and up and down etc. for a longer period of time than desired). γ is used to discount the value of future reward because, while the estimate may be accurate, the environment could change in a way that the agent does not expect, throwing reward estimates off. For this reason, immediate reward is always valued more than future estimated rewards.

Updating a Function of $\vec{\theta}$

The reason that there is such a difference between updating an array and updating an approximation is that the probability of all actions happening in all states is affected when any of the parameters are changed. With an array, there is no doubt that one update will only affect the probabilities of the possible actions chosen in one state. Fortunately, mathematics provides an effective tool for "nudging" the parameters in the desired direction to focus on improving one specific action-state scenario, while minimizing the effect on all of the other action-state pairs. This tool is called gradient.

The gradient of a function $\nabla f(\vec{\theta})$ is a vector of size equal to the number of parameters in the function; each element of the vector is the partial derivative with respect to one of the function's parameters. How the gradient works is out of the scope of this paper, but the important thing to know is that the gradient vector, when parameter values are passed, always points in the direction of positive slope [2]. Therefore, if the parameter values are adjusted slightly in the direction of the gradient, the returned value of the function will be slightly higher than before.

An example of an update using the gradient is shown through the assignment[7]

$$\Delta\vec{\theta} = \alpha\nabla \log \pi_{\vec{\theta}}(s,a)\, Q(s,a)$$

where $\pi_{\vec{\theta}}$ represents the policy that is a function of the parameters $\vec{\theta}$. Taking the logarithm of the policy can be ignored for understanding purposes; there is a Policy Gradient Theorem[7] that is used to arrive at this equation. The gradient of the policy points in the direction that increases the likelihood of the action *a* being chosen given the state *s*[1] (logically this works because the policy itself returns the probability of taking action *a* in state *s*). This gradient value is multiplied by the action-value function because that function says how good the action *a* is, given the state *s*. Hence, if *a* is bad enough that the action-state function returns a negative value, the parameters will be adjusted so that *a* is less likely to be picked as an action, given the state *s*.

Additional Considerations

Through these basic techniques for learning, an agent might be successful in figuring out an optimal solution to a problem. Sometimes, though, finding a solution requires taking into account a few other relevant issues. One important issue is exploration. Some solutions to other issues that can potentially improve a learning algorithm that will not be discussed here include Off-Policy learning[8], data decorrelation through Experience Replay[4], and using models to simulate experience[8].

Exploration

If learning by using the basic techniques, a problem can arise: the lack of exploration. When learning, if the agent sees that going left provides reward and that going right does not, then it has no programmed reason to ever try and explore the action of going right, even though it does not know if going right may be better in the long run. One common solution (that has already been used in parts of this paper) for this issue is to turn all decisions to take actions into probabilities of decisions to take actions; this way, the agent can learn from looking more into actions that appear to be better, but it occasionally will look into less-promising actions, in case they turn into something that could be good.

Designing a Reinforcement Learning Agent

Developing Reinforcement Learning software can be a much more complicated task than developing regular software due to the sheer variety of methods that can be used for learning. At this point, there still is no specific algorithm that is better than the rest, and many developers and researchers are constantly discovering new methods and making improvements on old methods for learning. This current state of Reinforcement Learning makes specific qualities of software more highly valued than the rest and has been helped significantly by well-made frameworks[3].

Valued Qualities of Reinforcement Learning Software

In order to be effective and worthwhile software, Reinforcement Learning software needs to focus on having a few specific qualities[3] (though this does not mean that other qualities are not important). It should be modifiable, reusable, and extensible. Modifiability allows for the software to be tweaked often, and is important because changes will happen frequently when trying to produce the best learning algorithm for a unique environment. Reusability is good for the field of RL so that researchers can share their findings and use the discoveries of others without much hassle. Extensibility encourages simple additions to be made to existing algorithms, and this can significantly help increase the pace at which discoveries are made. Perhaps someone has been learning about others' algorithms and a thought comes to mind about something that the others' algorithms do not account for; well, then the someone that was learning can easily test their thought out if the implemented algorithms are extensible. One more specifically important quality for RL software is performance. In order to train the agents, they must undergo learning for usually thousands of episodes, if not millions. As the saying goes, "time is money". The faster the program can process and learn from each episode, the better.

Frameworks

There is probably the most evidence behind the usefulness of reusability because of frameworks. Frameworks are similar to libraries in that they can provide operations, but different in that their specific purpose is to provide design. An application can use a framework as a skeleton, and then fill it out based on its own intent. Since design is often considered more challenging than implementation, frameworks are key to allowing people to develop RL software without requiring excessive amounts of time and software knowledge. They let users focus specifically

on the RL algorithms, rather than having to think about the software engineering aspects of RL development.

Conclusion

There is a variety of methods for calculating targets, for approximating value functions or policies, for ensuring exploration, and for solving a number of issues that were not explored in detail here. Different combinations of this variety of methods create different outcomes for each type of environment, and no combination has yet proven to be clearly optimal. It is important to understand the fundamentals of these methods and keep in mind key qualities of Reinforcement Learn software in order to effectively create Reinforcement Learning successes.

References

[1] Karpathy, A. (2016, May 31). Deep Reinforcement Learning: Pong from Pixels [Blog post]. Retrieved from http://karpathy.github.io/2016/05/31/rl/

[2] Khan Academy. (2016, May 11). Why the gradient is the direction of steepest ascent [Video file]. Retrieved from https://www.youtube.com/watch?v=TEB2z7ZlRAw

[3] Kovacs, T. & Egginton, R. Mach Learn (2011) 84: 7. doi:10.1007/s10994-011-5237-8

[4] Matiisen, T. (2015, December 19). Demystifying Deep Reinforcement Learning [Blog post]. Retrieved from https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., et al. (2013). Playing Atari with Deep Reinforcement Learning. Technical report. Deepmind Technologies, arXiv:1312.5602 [cs.LG].

[6] Silver D, Huang A, Maddison CJ, Guez A, Sifre L, van den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M, Dieleman S, Grewe D, Nham J, Kalchbrenner N, Sutskever I, Lillicrap T, Leach M, Kavukcuoglu K, Graepel T, Hassabis D (2016) Mastering the game of go with deep neural networks and tree search. Nature 529(7587):484–489

[7] Silver, D. (2015). Lecture 7: Policy Gradient [Lecture notes]. Retrieved from http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf

[8] Sutton, R. S., & Barto, A. G. (1998). Reinforcement Learning: An Introduction (1st ed.). Cambridge, MA: MIT Press.

[9] Tesauro, G. (1995). Temporal difference learning and TD-Gammon. Commun. ACM, 58-68.