Solving the Rubik's Cube with Deep Reinforcement Learning

Name: Justin Cheigh
Email: jhc5@williams.edu
GitHub
LinkedIn

Abstract

In this writeup I describe my process for solving the Rubik's Cube using deep reinforcement learning. My code (TensorFlow/Keras) can be found *here.* This project was inspired by [2].

To solve the Rubik's Cube we begin with an approximate value iteration (AVI) algorithm that involves training a ResNet model to output a value given some configuration of the Cube. We then solve the Cube with Monte Carlo Tree Search (MCTS), where we use the trained network to reduce both the breadth and the depth of the tree search. I also describe a group theoretic way to solve the Rubik's Cube, known as the Kociemba solver.

This writeup aims to describe my process is a fully self-contained fashion. I introduce reinforcement learning, deep learning (ANNs, CNNs, ResNets), deep RL (DQN), MCTS, and how one can combine AVI algorithms and MCTS to create a highly optimized Rubik's Cube solver. Finally I introduce group theory and describe a group-theoretic approach to solving the Cube.

Section 0- Introduction

In this paper, I demonstrate how to solve the Rubik's Cube using deep reinforcement learning. My code (Python/PyTorch) can be found *here.* I give credit to [2]. This is a cool project that I really enjoyed doing, so I decided to create this writeup to detail/explain my work.

A majority of this work follows the ideas in [2]. However, I will introduce the major concepts in an effort to be more self contained than most research papers. Ideally it will be possible to follow this project without any prerequisite knowledge of things like reinforcement learning or deep learning, but occasionally this may be difficult as to not make this writeup too long.

After describing the high level idea behind [2], I will transition to a completely different approach to solving the Rubik's Cube: group theory. Specifically I will discuss the Kociemba [1] method of solving the Cube.

Thanks for reading!

Subsection 0.1- High Level Approach

This subsection describes a high level approach to solving the Rubik's Cube with deep reinforcement learning (DRL).

Reinforcement learning (RL) is all about teaching *agents* to navigate through some *environment*. Ideally we aim to teach the agent which *action* to take at each *state* of the environment. *Q-learning*, a common RL algorithm, works by assessing the "quality" of each state-action pair. However, Q-learning is completely infeasible for larger environments. It turns out there are more than 43 quintillion configurations of the Rubik's Cube, which is completely unreasonable for something as simple as Q-learning. So, we need some way to approximate this Q-function, which is where we need *deep reinforcement learning*. However, before getting to deep RL, we need to discuss *deep learning*.

Deep learning involves using *artificial neural networks* (ANNs). At the simplest level, ANNs involve stacking layers of *artificial neurons*. Each neuron obtains its output typically by applying some non-linearity (ReLU for example) to a linear combination of its input. Finding these *weights* and *biases* involves *training* the neural network, in a process like *backpropagation*. Important theoretical work, like the *universal approximation theorem*, tells us that this is actually a worthwhile thing to do. Great! Now we can talk about deep RL.

Deep RL combines deep learning and reinforcement learning. We know that neural networks can potentially approximate functions well, so it's natural to consider using neural networks to approximate the Q-function. *Deep Q-Learning* works by training a neural network (called a *deep Q network* or DQN) to output the Q-value of each state-action pair (s, a) given the state s. While Deep Q-learning has achieved superhuman performance on things like Atari games [4, 3], we actually need more than just this to solve the Rubik's Cube.

Some engines, such as AlphaGO [5], were able to achieve incredible results by combining some kind of tree search with some kind of *value network* like a DQN. A common tree search is *Monte Carlo Tree Search* (MCTS). MCTS searches the *game tree* by keeping track of the "best" moves and running random simulations that favor these better moves more. One can prove that MCTS converges to *MiniMax* given certain circumstances. Furthermore, MCTS doesn't require a heuristic function and is able to reduce the state space, both of which often make it favorable to similar algorithms like *A* search with alpha-beta pruning*. However, MCTS is still very slow at times, which is why we need to combine the concepts of Deep RL to MCTS.

To solve the Rubik's Cube, we first train a value network that approximates the value of any given state. This process is fast but not as accurate. We then use MCTS, a slower process, to actually solve the Cube. However, we use the value network to reduce both the breadth and the depth of the tree search, effectively using the fast process to help speed up the more accurate slower process. All together, we create an efficient Cube solver without ever inputting any domain knowledge of the Cube!

Section 2- Reinforcement Learning

In this section I'll go over the major topics of reinforcement learning.

I'll start by describing the high level idea behind reinforcement learning through the example of chess. After an intuitive approach to the subject, I will transition to typical concepts of RL, including Markov Decision Processes, the Bellman Equation, and Q-Learning.

Subsection 2.1- Chess as an RL Task

This subsection contains the high level ideas of reinforcement learning. To cling to intuition, I will describe RL through the specific example of learning to play chess. However, most of the italicized terminology is universal and will be used in later sections.

Reinforcement learning deals with *agents* navigating some *environment*. In the case of chess, the agent is the learner or the computer, and the environment is, well, chess. One part of the environment are *states*. For chess, one state could be some random position with black to play. Notice that part of the state is specifying which player's turn it is. The agent starts in some *initial state* (the starting position of the chessboard with white to play).

At each state, there are known *actions* that the agent can take to go from state from state. For example, one action at the initial state of chess is moving the pawn on E2 to E4. Each state comes with some immediate *reward*. While certain environments like video games may have rewards at many states, games like chess may only have nonzero rewards at *terminal/final states* (states where the game is over).

We define a *policy* as a function that tells the agent what to do at every (non-terminal) state. The goal is have the agent learn the "best" policy in order to maximize the cumulative reward. In other words, we aim to find some *optimal policy*.

It turns out we can use various iterative algorithms to help agents learn some optimal policy. However, these methods often rely on more information than we are typically given: if an agent is navigating a truly new environment, chances are they don't know many things about the environment, including even what will happen if they take action a at state s!

We usually are only given a few things: the current state, the reward of the current state, and the actions we can perform. Under these tighter constraints, there's not much else to do except try things! RL really turns into the art of learning better and better policies *through experience*.

Great! Now that we've covered the high level approach of reinforcement learning, we will dive into formalizing this intuition. This formalization will prove useful in allowing us to achieve much more than we otherwise could. We'll start by covering Markov Decision Processes, which attempt to define *environment*.

Subsection 1.2- Markov Decision Process

In this subsection we will work to providing a definition of a Markov Decision Process (MDP), which will formalize the notion of a general environment.

To begin, we will define a state machine, which captures the notion of states and actions:

Definition 1 (State Machine)**.** A *state machine* is a 5-tuple $M = (S, \Sigma, \Delta, s_0, F)$, where S is a set of *states*, Σ is a set of *actions*, $\Delta \subseteq S \times \Sigma \times S$ is a set of *permitted actions*, $s_0 \in S$ is the *initial state*, and $F \subseteq S$ is a set of *final/terminal states*.

Remark 1. To clarify, some $\delta \in \Delta$ with $\delta = (s_i, \sigma, s_j)$ tells us that an agent taking action σ at state s_i *potentially* takes that agent to s_j.

While many problems can be formulated as a state machine, this definition still is missing a few things. To truly define an environment, one requires a notion of *reward*; how can an agent navigate the environment *well* without some (quantifiable) way of measuring success in an environment? Furthermore, one needs to think of navigating an environment from a probabilistic perspective. Any $(s_i, \sigma, s_j) \in \Delta$ tells us that it is possible to get from state s_i to state s_j by taking action σ. However, one really needs to say: "if an agent takes action σ at state s_i, then the agent will get to state s_j with probability p".

With these things in mind we can create a full definition of a MDP:

Definition 2 (Markov Decision Process)**.** A *Markov Decision Process* (MDP) is a triple (M, R, w), where $M = (S, \Sigma, \Delta, s_0, F)$ is a state machine, $R : S \to \mathbb{R}$ is a reward function, and $w : \Delta \to \mathbb{R}_{>0}$ assigns a position weight such that, for every $s \in S, \sigma \in \Sigma$,

$$\sum_{\substack{\delta \in \Delta \\ \delta = (q, \sigma, q')}} w(\delta) = 1.$$

Remark 2. Let $(s_i, \sigma, s_j) \in \Delta$. Then $w((s_i, \sigma, s_j)) = p$ tells us that the probability of getting to state s_j by taking action σ at state s_i is p. The final constraint is simply asserting that, for each state action pair, w represents a true probability distribution. We may also use the notation $\mathbb{P}[s_j \mid (s_i, \sigma)] = p$ as well.

Subsection 1.3- Bellman Equation

This subsection serves to understand what a "successful" agent is actually trying to accomplish.

Recall a policy maps states to actions. Specifically, a *policy* is a map $\pi : (S \setminus F) \to \Sigma$. We wish to define some notion of an *optimal policy* π^*. However, often there are many policies with infinite value (infinite cumulative reward). One way to avoid this problem is to define some *discounting factor*, which essentially means the present value of a reward in the future is discounted. Specifically, let $\gamma \in (0, 1]$ be our *discounting factor*. Then, for some $s \in S$, we can define the discounted reward $R_t(s) = \gamma^t \cdot R(s)$.

So, how good is a given policy π at a state $s \in S$? We can work on defining the utility of policy π at state s, but what we really care about is determining the best action to take at every non-terminal state. With this, we get the Bellman Equation:

$$U(s) := R(s) + \gamma \max_{\sigma} \sum_{s' \in S} \mathbb{P}[s' \mid (s, \sigma)] \cdot U(s'). \tag{1}$$

In words, the utility at state s is the immediate reward plus the discounted future reward obtained by taking the best action σ. The discounted future reward is essentially an expected utility gain. Notice calculating $U(s)$ using the Bellman Equation involves determining the "best" σ, meaning we can use this information to determine the optimal policy π^*.

It turns out that there are iterative guessing methods like *Value Iteration* that, both in the limit and in practice, use the Bellman Equation to find π^*.

However, if we step back we'll see that this might not be entirely useful. We may not always have access to the full MDP. Often we will only know a few things, like our current state, the reward of the current state, and the actions we can perform. Another reason methods like value iteration tend to be infeasible is that the computation required for these methods is simply not useful for any somewhat large state space.

So, if we assume we know only the bare minimum, determining this optimal policy seems absurd; we don't even know all of the states! It turns out we can still try different things that ultimately work well in practice, and this will be the focus of our next subsection.

Subsection 1.3- TD/Q Learning

Section Group Theory

We'll now cover *group theory*, a subset of *abstract algebra*.

Most pure math majors take abstract algebra as a core course, introducing them to a more rigorous study of algebraic structure

Bibliography

[1] URL: http://www.kociemba.org/math/imptwophase.htm.

[2] Stephen McAleer et al. "Solving the Rubik's Cube with Approximate Policy Iteration". In: *International Conference on Learning Representations*. 2018.

[3] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[4] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (Dec. 2013).

[5] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.