Reinforcement Learning in AlphaZero

Kevin Fu

May 2019

1 Introduction

Last week, we covered a general overview of reinforcement learning. (See Jenny's "Reinforcement Learning.") This lecture focuses on DeepMind's AlphaZero, one of the most recent developments in the field of reinforcement learning.

2 Background

In 2016, AlphaGo defeated Lee Sedol 4-1 in a Go match, becoming the first computer program to beat a human Go world champion. Go is played on a 19x19 board and has a search space 10^{100} times larger than chess' search space, making it significantly more challenging for computers. DeepMind overcame this with AlphaGo by training a deep neural network on expert human games to develop a policy and value function, then relying on Monte Carlo Tree Search reinforcement learning to improve both functions.

Later that year, DeepMind published details on AlphaGo's successor, AlphaGo Zero, which beat AlphaGo 100-0. The "Zero" part of the name refers to how AlphaGo Zero's neural net was trained entirely from self-play, cutting the first step in AlphaGo's learning process. DeepMind would later generalize their algorithm to learn any game from scratch, provided the rules to the game and enough training time, and brand it AlphaZero.

3 Monte Carlo Tree Search

A Monte Carlo Tree Search (MCTS) is very similar to the Minimax algorithm. Both are applied to deterministic, zero-sum, perfect information games, and both attempt to find the best next move from a position in the game with an internal tree structure.

However, Minimax relies on a full game tree, which is impractical in games with high branching factors, or many possible moves from a position. Typically researchers limit the number of future moves that Minimax searches, and craft evaluation functions to guess at the likelihood of winning from the furthest point reached. This is called a depth limited Minimax.

MCTS is better than Minimax for complex games because it doesn't need a full game tree to pick a best move, and in its most basic form, isn't reliant on a custom-tailored evaluation function. Instead, like other Monte Carlo methods, MCTS relies on the idea that probabilistic systems can be approximated with randomized trials, as well as some statistics principles. With enough trials, MCTS will learn the best move to play. To accomplish this, the algorithm loops through four phases: selection, expansion, simulation, and backpropagation.

3.1 Selection

Let's say we have a game tree with many win/loss stats for every move played from a few nodes. Given this information, if we wanted to maximize our odds of winning, we would play deterministically, picking nodes that maximize win rate:

$$R_i = \frac{w_i}{n_i}$$

- w_i is the wins for a move i
- n_i is the number of plays, or times that move has been selected, up to this point

But this will run into issues when we have little or no trials to work off of. Better moves may be ignored in favor of worse moves with higher win rates by virtue of their n counts. Instead, we want to pick moves stochastically, balancing exploitation (picking based solely on win rate) with exploration (picking moves that are under-explored and potentially better). The way MCTS handles this is with the upper confidence bound (UCB) formula:

$$U_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

- again, w_i and n_i are the win and play counts for a move i
- c is an exploration constant, typically $\sqrt{2}$
- t is the total number of plays up to move i

As you can see, the first part of the UCB formula is the same as the win rate formula from before. The second part prioritizes nodes that haven't been explored much, and the relationship between the two halves can be changed with the exploration constant c. The resulting formula balances exploitation and exploration to select promising nodes.

So for the selection phase, MCTS will run down the game tree, selecting the node with the max UCB value at every step, until it reaches a leaf node, as shown in Fig. 1.

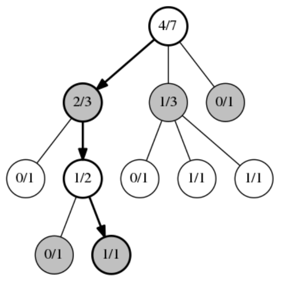

Figure 1: The best node, based on UCB, is selected at each step.

3.2 Expansion

Once a leaf node is reached, it is expanded by instantiating an empty child node for every possible move from that state. (Unless the game ends at that node, in which case there are obviously no more moves to expand with.) Then one of the children is randomly selected for the next phase, simulation. This is illustrated in Fig. 2.

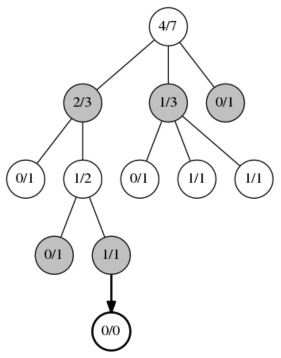

Figure 2: The leaf node is expanded, and a child is randomly selected. (In this example there's only one child.)

3.3 Simulation

This phase is where the Monte Carlo part of MCTS comes in. Like a typical Monte Carlo method, a complete game is simulated with random moves from the selected child node. This is called a light playout. Some implementations of MCTS use heavy playouts, where each move of the simulation is chosen by an evaluation heuristic. A playout continues until the game reaches a terminal state; in other words, if a win/loss/draw occurs.

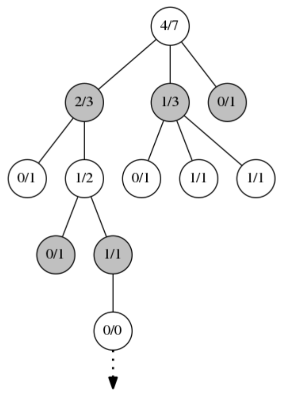

Figure 3: The selected child node is then simulated (represented here by a dotted arrow).

3.4 Backpropagation

Finally, after a simulation is completed, the results are then backpropagated up through the tree. This is not the typical machine learning definition of backpropagation—there are no activation functions here. Instead, the wins and play counts for every parent are simply incremented if necessary. In Fig. 4, the win result from the playout increments both the wins and plays for each of its parents.

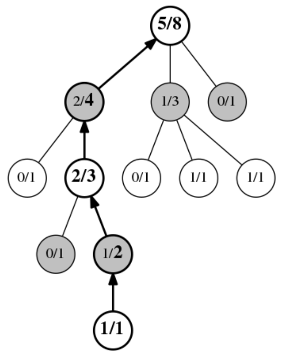

Figure 4: The win from the playout is backpropagated up the tree.

3.5 Looping

MCTS will loop through each of these four phases for a set number of iterations or amount of time. After enough loops, the tree will have a decent number of statistics for the next moves from the root position. Then the best way to pick the next move is deterministically, as our initial problem of not having enough information on some nodes has been resolved. Thus, the algorithm has taught itself what the best move is through intelligent trial-and-error.

4 Improving MCTS

The two primary ways to improve on a basic MCTS search are with a policy function and a value function.

Since games with high branching factors, like chess or Go, have many moves at each turn to evaluate, relying on randomized light playouts would take too long to loop through for accurate results. To improve this, we can create a policy function, p, that estimates the probability of an expert player would play each move from a given position:

$$p(s_i) = (p_0, p_1, p_2, ..., p_i)$$

- s_i is a given state, or position of the game
- p_i is the output of the policy function p for a single action (move) i

This is incorporated in a modified UCB formula for selection:

$$U_i = \frac{w_i}{n_i} + cp_i\sqrt{\frac{\ln t}{n_i}}$$

Now the MCTS' exploration is guided by the policy function, as well as the play counts. If the policy function is accurate, this will lead the MCTS to invest time in searching and simulating good moves of the tree and waste less time on bad moves.

If we wanted, we could use this policy function in simulation and turn our light playouts into more accurate heavy playouts. But to save even more time, we can cut out simulation entirely by approximating playouts with a value function, which outputs a value between -1 and 1, where 1 is a win and -1 is a loss:

$$v(s_i) \in [-1, 1]$$

This is similar to depth-limiting and using evaluation functions in Minimax. As long as our value function gives an accurate estimate of the playout (and takes less time to compute), then our MCTS will be able to search more nodes and pick a better move.

The problem is, obtaining generalized and accurate policy and value functions is difficult, even with expert player input. The breakthrough DeepMind was able to achieve with AlphaZero Go and AlphaZero was developing a method to train a neural network with MCTS to learn these functions.

5 The AlphaZero Algorithm

The algorithm DeepMind developed is based on two key concepts:

1. Accurate policy and value functions for a MCTS can be found by training one deep neural network.
2. This neural network can be trained on probabilities and results from games where the MCTS plays against itself.

To obtain training data, AlphaZero plays thousands of games against itself, and saves every state, s and corresponding MCTS probabilities, π. After a game finishes, it also saves the corresponding final game value, z, to generate training data of the form:

$$(s_i, \pi_i, z_i)$$

Then the network can be trained on this dataset to take a game state as input and output both a policy and value for that state. Mathmatically, this can be expressed as, where a single network f outputs both a policy function p and a value function v from a state s_i:

$$(p(s_i), v(s_i)) = f(s_i)$$

The loss function for this network f is:

$$L = (z - v)^2 - \pi \ln p + \lambda \|\theta_f\|$$

- $(z - v)^2$ is the mean-squared error of the value function, where z is the actual value from a self-play game and v is the value function's output
- $\pi \ln p$ is the cross-entropy loss of the policy function, where π is the actual probabilities from MCTS and p is the policy function's output
- $\lambda\|\theta_f\|$ is L2 weight regularization, where θ_f is the weights of the network and λ is just a constant

Finally, the actual structure of the neural network consists of 19 mini-convolutional neural networks, called blocks, stacked on top of each other, with two final residual blocks for the policy and value function outputs.

5.1 Why does this work?

In an MCTS, after a number of iterations, the root state will have probabilities for each possible move, which is why we can pick deterministically from these actions for the best one. Crucially, the MCTS will provide more accurate probabilites than the neural network alone. If you view a basic, random-playout MCTS as a MCTS with a randomized policy function, it's obvious that the MCTS' result will be better than picking a random move. This holds true even when the policy function is accurate: the MCTS' best move will be better than the policy function on its own. Therefore, the policy function can be trained to match the probabilities of the MCTS it informs.

Similarly, the actual value of a game played between two identical computer agents will be more accurate than the value function's estimates, even when both agents are informed by that value function. So the value function can be trained to match the self-play games' results too.

With this algorithm, supercomputers and ample training time, and a few other tricks like parallelized-MCTS, the researchers at DeepMind were able to achieve superhuman performance with AlphaZero. For a simpler version of the AlphaZero algorithm that plays Othello, see `https://github.com/suragnair/alpha-zero-general` or `https://github.com/kfu02/orez-ahpla`.

6 Sources

1. MCTS images:
    - `https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/`
2. DeepMind's blogs on AlphaGo Zero and AlphaZero (which contain links to their published journals):
    - `https://deepmind.com/blog/alphago-zero-learning-scratch/`
    - `https://deepmind.com/blog/alphazero-shedding-new-light-grand-games-chess-shogi-and-go/`
3. Simple AlphaZero explanation with Tic-Tac-Toe:
    - `http://tim.hibal.org/blog/alpha-zero-how-and-why-it-works/`