Following in the Footsteps of Alphazero: Reinforcement learning and game AI

Lecture 4: Advanced game AI techniques

December 8, 2024

Last week, we looked at the minimax algorithm and its more efficient cousin, alpha-beta pruning. As we discussed, in many applications where the search space is large, these algorithms are coupled with an *evaluation function*. This is a function that takes a leaf node of the game tree and evaluates it with a numerical score. How? That's the million-dollar question. In the past, evaluation functions were hand-crafted based on human expertise.

The main topic of this lecture is the Monte-Carlo tree search (MCTS) algorithm, an incredible way of doing it that dispenses with the need for an evaluation function altogether. MCTS also fits nicely into the framework of reinforcement learning. As in introduction to the field, we will start today's lecture by presenting a simple reinforcement learning problem, the **multi-armed bandit**.

1 Multi-armed bandits

Consider an agent that has to choose between k possible actions. For each action that the agent takes, it receives a reward sampled from some unknown distribution. We assume that this situation repeats itself a large number of times. The objective is to maximize the total reward.

If we **knew** the distributions in advance, the optimal strategy would be to always choose the action with the maximum expected reward. Conceptually, we can think of the agent as having two tasks - first, to find out which action maximizes the expected reward, and then to choose that action repeatedly in order to reap the benefits of its knowledge.

One strategy could be to try each action once and, at each subsequent step, select the action whose average reward so far has been maximal. This is a valid strategy, but it might not lead to an optimal solution. For example, consider the case where one action consistently yields a reward of 1, while the second action provides a reward of 0 with a probability of $\frac{2}{3}$ and a reward of 6 with a probability of $\frac{1}{3}$. Clearly, the optimal strategy in the long run is to pick the second action. However, if the agent receives a reward of 0 the first time it tries

action 2, it might prematurely conclude that the first action is better and never explore the second action again. With this example in mind, especially if we anticipate many repetitions of this scenario, it makes sense to invest some time in trying each possible action several times to get a better approximation of its expected value.

We call actions that choose the action with the maximal expected value as *exploitation moves*, and actions that select suboptimal actions in the hope that they might turn out to be the best ones *exploration moves*. Broadly speaking, exploration and exploitation are both important goals, and since they often conflict, we can speak of a trade-off between exploration and exploitation. This is a fundamental issue that arises in many reinforcement learning problems.

In what follows, we will describe a few different ways to balance exploration and exploitation. It's worth taking the time to understand them because these strategies can be applied in many RL scenarios that lie far beyond the multi-armed bandit framework.

The ε-greedy algorithm

One way to resolve this trade-off is to choose some small constant $\varepsilon > 0$. At each turn, with probability $1 - \varepsilon$ we choose the action whose average reward so far is maximal, and with probability ε we make a random move. The exploration moves ensure that our value-approximations eventually converge to their expectations. By choosing a small value of ε we ensure that the agent spends most of its time making exploitation moves. This is the ε-greedy algorithm, which forms the backbone of many RL systems.

This algorithm involves maintaining a list Q of expected payoffs - one for each action. At each step, if we selected the action a and received the reward r, we make the update

$$Q(a) = Q(a) + \frac{1}{\#\text{visited}(a)}(r - Q(a)).$$

This ensures that $Q(a)$ is the average payoff that the agent received when selecting the action a. Additionally, if we foresee that the expected values of the actions may change over time, we might want to make the values $Q(a)$ a weighted average, in which more recent attempts receive more weight. This can be accomplished by changing the update rule to

$$Q(a) = Q(a) + \alpha(r - Q(a)),$$

where $0 < \alpha \leq 1$ is a constant.

The Upper Confidence Bound algorithm

The ε-greedy algorithm fails to differentiate between the different suboptimal actions. In practice, clearly there is a difference between actions that have

performed terribly and actions that have either not been tested, or they have been tested and their expected payoff is very close to optimal.

Another approach, which has been wildly influential and which we will encounter again in our discussion of the MCTS algorithm, is to assign a numerical value to each action that balances their expected payoff and the number of times they have been tried. This is the *upper confidence bound* algorithm. The exact formula one should use is a delicate question, but the formula that appears in textbooks is:

$$\text{UCB}(a) = \text{Expected Value}(a) + C\sqrt{\frac{\log(\#\text{Total Visits})}{\#\text{visited}(a)}}.$$

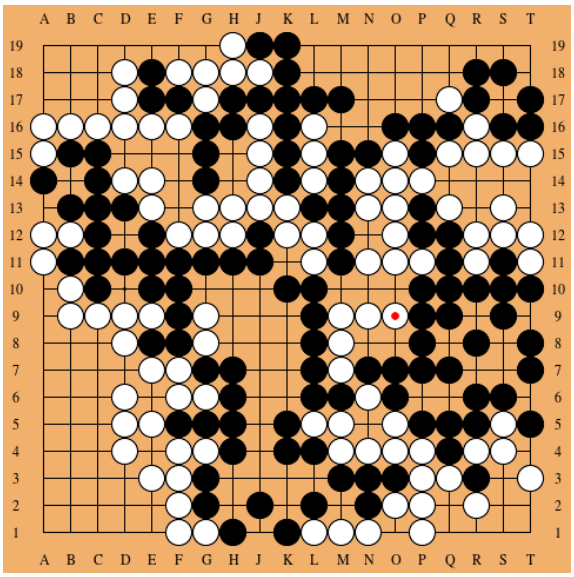
As you can see, the formula consists of two parts: The first term is our current estimate of the expected value of the action a, and the second term corresponds to our uncertainty about the first term - the more we visit an action, the better we can expect our estimate of the expected payoff to be, and so the smaller the second term becomes. At each step, the UCB algorithm deterministically selects the action with the maximal UCB, and then updates the UCB according to the reward that it received.

Monte-Carlo Tree Search

Back to our main topic of the day.

The Alpha-beta search algorithm is powerful and versatile, achieving great successes in many settings, notably in chess. However, there is another popular board game, Go, where the formula of alpha-beta search plus handcrafted evaluation was much less successful.

Go is a Japanese game, which may not be as familiar to you as chess. It is a two-player game played on a 19 by 19 board. Players take turns placing white or black stones on the board. The objective is to capture territory. At the end of the game, the winner is the player who has surrounded the most empty squares. Below is an example of a Go position. The points that are surrounded by black count as points for Black, and the same is true for White.



As an aside, for a fun introduction to the game, I recommend the anime "Hikaru no Go."

Now, the chess program Deep Blue beat World champion Garry Kasparov in 1997, but 20 years later, until AlphaGo arrived on the scene, there was no computer program that could play as well as the best Go players. There were two main challenges that Go programs had to overcome:

1. Since the 19×19 Go board is so large and stones can be placed anywhere, the number of legal moves in a position can be huge. The average number of legal moves in a Go position is about 200, whereas in chess it's only about 30. As a result, alpha-beta search can't look too deeply into the game before the number of nodes in the search tree becomes unmanageable.

2. Compared to chess, there are fewer recognizable "signposts" in a Go position to guide the evaluation function. Strong chess players count pieces, look at open files, king safety, pawn structure and so on. They can explain their mental process pretty well, and programmers can use that to craft an accurate evaluation function. In contrast, strong Go players often rely on intuitive assessments that are hard to explain explicitly.

Monte Carlo Tree Search (MCTS) was proposed as a solution to these obstacles. Simplistically, the idea is that instead of using a heuristic evaluation function to decide if a position is good or bad, you have the computer play itself many times from that position and compile statistics on the outcomes.

Each outcome is called a "rollout." For this scheme to be efficient, the rollouts have to be played *fast*. One could imagine that each move is chosen randomly from the set of available moves. More typically, the rollouts are guided by some simple heuristic, so that the game results won't be quite so chaotic.

This approach solves both issues: Since most legal moves are never explored, the breadth of the game tree doesn't really affect the runtime. Additionally, statistics on the results of the games replace the evaluation function, eliminating the need for an explicit set of rules to evaluate a position.

In more detail: MCTS maintains a search tree of moves whose root is the position we want to evaluate. Each step consists of four main operations: selection, expansion, simulation and backpropagation.

- **Selection:** In this step, we move down in the game tree until we get to a leaf node. Of course we will need to find a strategy to select a move in every step. We will later see how such a strategy can be constructed regardless of any knowledge about the game by just using statistics.
- **Expansion:** In this step we create a new node by making a move at the node selected in the selection step. This move could be selected at random, or according to some other strategy.
- **Simulation:** In this step we run a computation to get statistical information about the new node. The most straight-forward way is to play a certain number of random games, i.e. select random moves until the game finishes in a win, draw or loss.
- **Back-Propagation** In this step we take the statistical information that we computed by executing the random games and propagate it back up to the root of the game tree.

In your programming exercise you will create a MCTS player for a simple board game, so it's important that you understand the details of each step.

First, there is one piece of super confusing terminology that I have to explain: Usually, a leaf node in a tree is a node with no children. However, in the context of MCTS, a leaf node is a node that has untried moves. In other words, we stop the selection step the moment we get to a node whose children don't cover all of the legal moves from that position.

Next, assuming that the current node isn't a leaf node, how should we select which child to go to? Once again, we want to strike a balance between exploration and exploitation: We want to mostly explore the best moves (exploitation), but on the other hand we want to sometimes look at suboptimal moves (exploration) because they might turn out to be good after all.

One simple way to do this is the ε-greedy strategy. In this approach, we choose the highest-scoring move with probability $1 - \varepsilon$, and with probability ε we choose a random move. The parameter ε allows us to choose how to balance exploration and exploitation.

However, when you think about the ε-greedy algorithm, you realize that it doesn't make sense to spend the same amount of time exploring the second best option vs the worst option. You want to somehow explore moves which you suspect might be pretty good and haven't been explored much yet. The most

popular way to do this is called UCT (Upper Confidence bound for Trees). In this approach, each node has an associated number:

$$UCT = \frac{\#\text{ wins}}{\#\text{ visits}} + C \cdot \sqrt{\frac{\ln(\#\text{ parent visits})}{\#\text{ visits}}}$$

At each step, the selection algorithm selects the node whose UCT is maximal. The UCT formula has two terms: The first term is just the expected value of the node, and the second is a measure of the uncertainty - it's big when the number of visits is small, so it encourages the algorithm to look at less explored options.

When the selection stage gets to a "leaf" node - a node with unexplored options - we choose one of them and create a new node in the tree. This is the expansion stage.

We then perform a "rollout", which is a simulated game from the node position. In the pure form of MCTS, these rollouts are completely random. However, random rollouts convey relatively little information about the strength of a move. For example, consider a chess position in which I can capture a defended rook with my queen. This would be a bad move - losing a queen for a rook! However, if the rollouts are completely random then the player whose rook has been captured will usually not recapture - ie they will just lose a rook. This would lead the algorithm to (at least initially) think that capturing the rook is a good move.

Therefore it's better if the rollouts are not completely random. Usually a simple and fast heuristic function is used to select the moves for the rollouts.

Finally, the way that we back-propogate depends a lot on the selection strategy. If we use UCT or ϵ-greedy as described above, we just have to update the counts of wins and visits for the ancestors of the new node that we created in the expansion step.