Recitation 14 April 24

1 MCTS: Selection & Expansion In-Class Exercise

Recall the formula for the UCB heuristic for the selection and expansion steps of MCTS at some state s:

$$UCB(s) = \frac{totalscore(s)}{count(s)} + c\sqrt{\frac{\ln N}{count(s)}}$$

where the first term $\frac{totalscore(s)}{count(s)}$ is the average score of the node (dividing total summed score by the number of times we've visited the node).

$count(s)$ is the number of times we've visited child state s, N is the total number of simulations we've run, and c is the exploration parameter (typically $\sqrt{2}$).

We will now run our own exercise of the selection/expansion phase of MCTS for the candy grab game. Recall that we start with 11 candies on the table, and at each turn, alternating players can grab either 1 or 2 candies. The winner is the person who grabs the last 1 or 2 candies.

1. We have 11 candies. We can start our search tree by initializing our $UCB(root) = -\infty$. Let every unexplored node also be initialized with a UCB value of $-\infty$.
2. Our two options are grab 1 or grab 2 candies, leading to either 10 or 9 candies left. In partners, choose one option and perform **rollout**.
   - Each pair represents a simulation. Each pair can do it one time, or multiple times (starting from 11 candies).
   - Record these things: 1) How many simulations you did, 2) Which action you performed (1 or 2) from 11, and 3) Who won (the starting player, or the other player).
3. Now calculate the updated scores and UCB values for both $s = 9$ and $s = 10$, and record those here.
4. What do we backpropagate to the original children and the root after rollouts? Remember that the scores will alternate at every layer (so if Player A starts at $s = 9$ and wins, that means the score at the root decreases, since Player B started at $s = 11$ and lost).

   Return the result of each rollout, and keep incrementing the number of simulations and updating the score (who won), applying it to the matching state accordingly. This updates the UCB value for each child state $s = 9$ and $s = 10$.
5. Which move at $s = 11$ (take 1 or take 2) would MCTS prefer after several simulations, and why?

   Answers will vary. This should be a comparison of $UCB(s = 9)$ and $UCB(s = 10)$.

2 AlphaGo

We need to go over AlphaGo in recitation:
Policy and value heads
What is stored in self play and how is it used to train the network

AlphaGo's core idea is to combine deep neural networks with Monte Carlo Tree Search (MCTS), using learned guidance to dramatically reduce the search space. The architecture revolves around two main neural network outputs: a policy head and a value head. The training process utilizes a powerful data generation strategy through *self-play*.

1. What is self-play and how does it help AlphaGo perform so much better than humans?

   Self-play is a reinforcement learning technique where the model generates its own training data by playing millions of games against itself. In AlphaGo, the neural network plays against itself using MCTS to simulate games. If AlphaGo only learned by copying human experts (a.k.a. imitation learning), its maximum skill would be capped at the human level. Self-play allows it to discover entirely new strategies that humans have never considered.

2. In the AlphaGo architecture, the neural network splits into two heads: a policy head and a value head. What does each head output and what role effect does it have on the learning process?

   Policy Head

   - **Output:** A probability distribution over the possible next actions given the current state, $\pi(a|s)$.
   - **Effect** It guides the MCTS during self-play by narrowing the search space to high-probability moves. This makes the learning process more efficient by generating better quality training data for the network's next iteration.

   Value Head

   - **Output:** A scalar value estimating the worth of the current board state, ranging from $[-1, 1]$.
   - **Effect on Learning:** This replaces the computationally expensive MCTS rollout stepmake. By providing a fast neural network forward pass to predict the winner, it accelerates the generation of self-play data.

3. During AlphaGo's self-play phase, it generates dataset tuples in the form of (s_t, π_t, z_t). What does each component of this tuple represent?

s_t is the board state at time t. π_t is the MCTS-improved policy (search probabilities). z_t is the final actual outcome of the game ($\in [-1, 1]$).

4. AlphaGo's Tree Policy modifies the MCTS action selection to incorporate the policy network. Actions are selected during the search by maximizing the formula:

$$a_t = \text{argmax}_a \left(Q(s_t, a) + c \frac{\sqrt{N(s_t)}}{1 + N(s_t, a)} \pi_\theta(a|s_t) \right)$$

Where $Q(s_t, a)$ is the average reward from simulations, c is the exploration constant, $N(s_t, a)$ is the visit count, and $\pi_\theta(a|s_t)$ is the prior expert probability provided by the policy network.

Explain how the inclusion of $\pi_\theta(a|s_t)$ changes the behavior of MCTS compared to a standard UCB formula. Specifically, when evaluating a node, how does the selection behavior shift from the early stages to the later stages (when the action has already been heavily sampled)?

- **Guided Exploration:** Standard MCTS explores unvisited branches blindly. By including $\pi_\theta(a|s_t)$, the neural network is forced to use its prior knowledge, biasing the exploration term toward moves that the policy network already predicts are strong.
- **Early Stages:** When an action has not been visited much ($N(s_t, a)$ is small), the exploration bonus is large and heavily influenced by $\pi_\theta(a|s_t)$, so the search will prioritize exploring the network's top recommended moves.
- **Later Stages:** As an action is heavily sampled, the denominator $1 + N(s_t, a)$ grows so the exploration term decays. Thus, the selection decision becomes dominated by $Q(s_t, a)$, which is the actual empirical value discovered during the search. Intuitively, if a move had a high prior but lead to bad outcomes, the model would learn to abandon it.

3 Deep Q-Networks

Recall that we use the following Q-update in regular reinforcement learning:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

We update the Q-value for a certain (state, action) pair by adding a weighted TD error to move the estimate toward the target.

Now, in deep reinforcement learning, we train Deep Q-Networks (DQN) to update our Q-values. We follow the following procedure:

1. Create new data $\mathcal{D}$ through simulation. Store transitions (s,a,r,s') in dataset $\mathcal{D}$, then sample random mini-batches for training.
2. Training step: We update the main Q-network using gradient descent:

   $$\theta \leftarrow \theta + \alpha(y - Q_\theta(s,a))f(s,a)$$

   Where $f(s,a)$ is a feature approximation function for the state-action pair (s,a).

As with any deep learning model, we need a loss function that guides our updates. In Deep Q-Learning, we use a loss function that compares our Q-value prediction and the target and uses gradient descent to update the weights of our DQN to approximate our Q-values better.

- This is our **sample**, where θ^- are the target network parameters:

  $$y = R + \gamma \max_{a'} Q_{\theta^-}(s',a')$$

  Note that y does not depend on our trainable parameters θ. The target network (parameterized by θ^-) is a copy of the main network. We keep it fixed temporarily so the target value does not move while learning. This is different from θ, which are our trainable parameters

- This is our **loss function**, which is the MSE of the Temporal Difference error:

  $$L(\theta) = (y - Q_\theta(s,a))^2 = (R + \gamma \max_{a'} Q_{\theta^-}(s',a') - Q_\theta(s,a))^2$$

How is this different from regular Q-learning? The difference is how we get $Q(s',a')$. Instead of calculating the successor Q-values $Q(s',a')$ directly, we use our network to predict a Q-value.

For simplicity, we replace a full neural network with a linear model that uses linear feature approximation functions to approximate feature values. Real DQN uses a neural network, but gradient descent follows the same idea.

$$Q_\theta(s,a) = \sum_i \theta_i f_i(s,a)$$

Then, in every step, we update our weights with gradient descent with

$$\theta \leftarrow \theta + \alpha(R + \gamma \max_{a'} Q_{\theta^-}(s',a') - Q_\theta(s,a))f(s,a)$$

Recall that the gradient descent formula is

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

Now, let's show that our update is indeed the gradient descent update, $\theta \leftarrow \theta - \alpha \cdot \nabla_\theta L$. Derive the DQN update from the gradient descent update, using the provided $L(\theta)$.

1. Calculate $\nabla_\theta L(\theta)$, in terms of $\nabla_\theta Q_\theta(s,a)$. *Hint*: y does not depend on θ.

$$\begin{aligned}\nabla_\theta L(\theta) &= \nabla_\theta (y - Q_\theta(s,a))^2 \\ &= 2(y - Q_\theta(s,a)) \cdot -\nabla_\theta Q_\theta(s,a)\end{aligned}$$

2. Calculate $\nabla_\theta Q_\theta(s,a)$, given that we use a linear feature approximation so $Q_\theta(s,a) = \sum_i \theta_i f_i(s,a)$.

$$\begin{aligned}\nabla_\theta Q_\theta(s,a) &= \nabla_\theta \sum_i \theta_i f_i(s,a) \\ &= f(s,a)\end{aligned}$$

3. Now, put them together and show that our update is

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

where

$$\nabla_\theta L(\theta) = -(R + \gamma \max_{a'} Q_{\theta^-}(s',a') - Q_\theta(s,a)) f(s,a)$$

$$\begin{aligned}\nabla_\theta L(\theta) &= -2(y - Q_\theta(s,a)) \nabla_\theta Q_\theta(s,a) \\ &= -2(y - Q_\theta(s,a)) f(s,a)\end{aligned}$$

Then, we can put the gradient into the gradient descent formula:

$$\theta \leftarrow \theta - \alpha(-2(y - Q_\theta(s,a)) f(s,a))$$

which simplifies to the following equation (the constant 2 is absorbed by the learning rate α):

$$\theta \leftarrow \theta + \alpha(R + \gamma \max_{a'} Q_{\theta^-}(s',a') - Q_\theta(s,a)) f(s,a)$$