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

Lecture 7: Presenting Alphazero

January 2, 2025

1 A Recap of MCTS

After achieving several notable successes, the DeepMind team set their sights on a notoriously hard AI milestone: writing a computer Go program that could compete with and beat the strongest human Go players. And to make matters more difficult, they didn't want to use any domain-specific knowledge.

Their starting point was the MCTS algorithm that we covered in a previous lecture. To recap, this is a very general reinforcement learning algorithm that can be applied in a variety of settings. Like the minimax algorithm, it maintains a search tree of possible continuations from the current state, but there are several important differences:

- The search tree isn't searched to a fixed depth. Instead, the algorithm uses a formula (the UCT formula) that prioritizes exploring nodes with a better expected payoff.
- No human-coded eval function is needed! Instead, when the algorithm reaches a new node, it performs a certain number of random "rollouts", that is, quick games where the moves are made randomly or using a simple heuristic.
- The updates to the values of the nodes aren't made using the minimax philosophy. Instead, the value of a node is the average of the payoffs of all its rollouts.

This is a beautiful approach! Unfortunately, simply using UCT does not yield a strong chess or Go player on its own. I'm not a great Go player, but I do play chess and I can explain why this algorithm wouldn't work so well. Consider the following position:

What's the best move? I think that anyone who knows a little about chess might consider playing d3, or 0-0, or g4. Each of these moves have their rationale - they improve White's position a little bit in some way. Then, there are other moves, such as Rb1, or Bf1, that don't seem to do anything at all. A human player wouldn't even consider such moves for a second, even though they aren't actually BAD.

And then there are moves like Bxf7+ and Nxe5. These moves actually lose material, potentially making the position much worse! But an experienced player knows that sometimes such moves can be the start of a *combination*, a precise sequence of moves leading to a win. And in fact, Nxe5 does exactly that! The critical line is Nxe5 Bxd1 Bxf7+ Ke7 Nd5 checkmate! This is a famous sequence known as Legal's mate.

Now, a UCT player should see that Nxe5 leads to the loss of a queen. It's likely that in a **random** game, Black would score very well from the resulting position. In fact, Black is indeed winning, **unless** White plays the precise sequence of moves that lead to mate. Even if the UCT player eventually stumbles onto the winning line, it would only influence the expected payoff of Nxe5 by a tiny bit at first. It would take a VERY long time for the AI to realize that Nxe5 is the best move.

More generally, there are two main issues with UCT:

1. The method of random rollouts is bad for chess (and probably many other problems): A sequence of random moves, or even just sub-par moves guided by a simple heuristic, may not contain a lot of useful information, and is computationally expensive.

2. The UCT formula that decides which nodes to explore, is perhaps TOO general: It doesn't take into account domain specific knowledge. For

example, in chess, strong players develop a pattern recognition that guides them to decide which moves are promising. They still have to calculate concretely to make sure that the move actually works, but they don't waste time on moves that are "obviously" bad.

2 The PUCT algorithm: an overview

Faced with these challenges, the DeepMind team developed a more sophisticated version of UCT that they called PUCT (Predictor + UCT). The basic idea is to use two neural networks, a value network and a policy network, to address the two problems that we described above.

1. Instead of using a random rollout to assess the value of a leaf node, the position is fed into the value network (the "predictor"), which outputs the (estimated) payoff from the current position. This is more accurate than a random rollout would be.

2. The policy network outputs a number for each move, which can be understood as the a-priori probability (without any calculation) that the algorithm will end up choosing that move, after performing the search. During the selection stage, the MCTS uses a formula that combines a node's expected value, the UCT exploration term, and the a-priori probabilities given by the policy network to choose which node to explore.

It turns out that pretty much the same features are needed to get the right policy and value, and so Alphazero ended up having a single network with two separate outputs: The value head and the policy head. This simplifies the architecture and increases the efficiency.

3 PUCT under the microscope

Since the final project of this course is to implement the AlphaZero algorithm, it's super important that you understand exactly how it works. Below, we present pseudocode for the PUCT algorithm. This pseudocode assumes that you have a trained neural network with a policy head and a value head.

The `policy` is a list of probabilities for each legal move in the given position, and `value` $-1 \leq v \leq 1$ is the expected outcome of the game **from the perspective of the current player**. Eg, if the value head outputs $+0.8$ and it's Black's move, then the network thinks that Black will probably win.

Like the MCTS algorithm, the PUCT algorithm builds up a game tree one node at a time. Each iteration consists of three phases: selection, expansion/simulation, and backpropagation. It looks something like this:

```
best_move(position)
    root = Node(position.clone())
    for _ in range(iterations):
        node = select(root)
        node = expand(node)
        backpropagate(node)
    return choose_best_move(root)
```

1. **Selection:** Starting at the root node (the current position), the algorithm traverses the tree until it reaches a leaf node—a node with unexplored children. At each step, it selects the child node with the highest PUCT score, given by:

$$PUCT = Q + c \cdot P \cdot \frac{\sqrt{N_{\text{parent}}}}{1 + N}.$$

   Here Q is the average payoff of the node from the point of view of the current player, P is the a-priori move probability from the policy head, N is the number of visits to the node, and N_{parent} is the number of visits to the parent node. The constant c balances exploration and exploitation.

   As you can see, this formula is very similar to the UCT formula, except that the exploration term is multiplied by the a-priori move probability P. In effect, this formula says that we want to explore nodes where the expected payoff is high, the a-priori probability that they are good is high, and the number of visits N is low.

```
select(node):
    while node.N > 0: # Not a leaf
        node = select_child(node)

select_child(node):
    best_score = -inf
```

```
    best_child = None
    for child in node.children:
        uct_score = (
            child.Q + c * child.P * (sqrt(node.N) / (1 + child.N))
        )
        if uct_score > best_score:
            best_score = uct_score
            best_child = child
    return best_child
```

2. **Expansion & Simulation:** When a leaf node is reached, the algorithm expands one of its unexplored children. The corresponding position is passed to the neural network, which outputs a value for the position and probabilities for all legal moves. The value becomes the initial Q of the new node, and the probabilities are stored as P for its children.

```
expand(node):
    legal_moves = generate_legal_moves(node.position)
    policy, value = neural_network.predict(node.position)
    for move in legal_moves:
        child_position = node.position.clone()
        child_position.make_move(move)
        child_node = Node(parent=node, child_position, move)
        child_node.P = policy[move]
        node.children.append(child_node)
    return node
```

3. **Backpropagation:** After expanding a node, the algorithm backpropagates the value of the newly expanded node up the tree. At each node along the path to the root, the visit count N is increased, and the Q value is updated to reflect the average of its children's values.

```
backpropagate(node, value):
    while node is not None:
        node.N += 1  # Increment visit count
        node.Q += (1/node.N) * (value - node.Q)
        node = node.parent
        value = -value
```

Once some predetermined number of nodes have been explored, the algorithm chooses the move with the maximal number of visits. This is a bit counter-intuitive, because I would have thought that we should choose the move with

the maximal expected payoff. There is actually a long-standing discussion about which approach is better for MCTS algorithms in general, and the consensus is that there isn't much difference between the two choices. My assumption is that the DeepMind team tried out both approaches and chose the one that performed better.

```
choose_best_move(root_node):
    best_child = max(root_node.children, key=lambda child: child.N)
    return best_child.move
```

Note that although some of the elements in the algorithm can be interpreted as probabilities, the algorithm is completely deterministic.

4 What about the network?

Ok. So we've worked out we want our neural network to *do*. But what architecture should we use? And how do we train it?

The architecture chosen by the DeepMind team was to use a CNN to extract features from the board position. After that, the network is split into two paths: One for the policy head, and one for the value head. Each path is essentially a series of fully connected layers followed by the output.

A couple remarks: The input to the network is the current position, but it's very important that the inputs are binary. So for example one way to input a chess position is as 12 8 by 8 matrices, one for each possible piece and color. And there are other important inputs! Things like castling rights and OF COURSE whose turn it is!

In some of your projects you will have a game with a score. That HAS to be one of the inputs too!

Next, the outputs: Some of you may have noticed that there is a problem with the policy head: The set of legal moves depends on the position! The solution is to have an output for every POSSIBLE move: ie there is an output for Knight from e6 to f4 even if there is no Knight at e6. Altogether the policy head has over 1000 outputs. This is a bit cumbersome, but considering the large size of the network it's not a major issue.

Essentially, this same basic architecture was used for go, chess, and shogi, with minor changes. For example, the number of filters in the CNN was 73 for chess, but 139 for Shogi.

As we said, the neural network is built using a CNN. I think that this choice was motivated by the desire to beat Go, where the board is big (like an image) and has local dependencies between the stones, which convolutions are good at detecting.

5 The Training Pipeline

Training the network turns out to be a bit complicated. We can use an approach that is very similar to the Monte-Carlo algorithm for training Q-learning agents.

Namely, we let the program play against itself a large number of times. For each position from each game, we know the result of the game and we know which move was chosen. So we can train the policy head to predict which move will be chosen in a given position, and we can use the result of the game to train the value head to predict the expected result of the game.

But there's a problem. As we described it, the PUCT algorithm is completely deterministic. If we leave it as is, we will get the same game played over and over again. It's basically the old exploration vs exploitation problem. Alphazero has two ways of dealing with this:

1. During training, the bast move is chosen randomly from a distribution that is proportional to the visit counts. Eg if there are three legal moves with visit counts 100, 40 and 20 then the first move is chosen with probability $\frac{5}{8}$, the second with probability 14 and the third with probability $\frac{1}{8}$.

2. Again during training, we add some noise to the policy values at the root node. Basically we take the weighted average of the policy vector at the root with a random distribution. The specific distribution used by Alphazero is called "Dirichlet noise", but I don't think the type of distribution is too important.

Another little point - the policy head is trained to predict the MCTS visit counts (not the move actually played). This aligns the network's policy with what the tree search found.

As we mentioned, the value head is trained to predict the final result of the game (+1, 0, or -1). Alternatively, you can use the bootstrapping approach, where the value head predicts the final evaluation of the root position.

The loss function combines two terms:

$$L = \text{Policy Loss} + \lambda \cdot \text{Value Loss},$$

where λ is a weight to balance the two components. The policy loss is the cross-entropy between the predicted policy and the visit counts, and the value loss is the mean squared error (MSE) between the predicted value and the target value.

6 Practical Tips for Your AlphaZero Engine

Here are some tips to make your implementation smoother:

1. **Warm Start:** Start with a simpler MCTS player (random rollouts instead of a neural network) to generate initial training data.

2. **Efficient Training:** Limit the number of MCTS simulations per move early in training. You can increase this as the network improves.

3. **Debugging:** In a big project there is a lot that can go wrong, and it's very important to check and debug everything thoroughly! In particular, it can be very hard to understand what the neural net is actually doing. So I recommend starting at least with a small and simple network (NOT a CNN!). After you train it for a bit, check it on some sample postions. You may find that you forgot an important input (like the score! Or whose move it is!) or that the output is basically random regardless of the position. This means you probably have a bug!

4. **Tuning:** This is an unsung and underappreciated part of any serious work with NNs: You have to fiddle around with them. The architecture, the learning rate, the training pipeline, the weighting in the loss function, and so on. It takes time. Be patient!

5. **Compute:** I haven't told you guys this, but the college has a powerful cluster of computers in the basement, and our tech guy (Efi) has been begging me all semester to tell you to use it. So after everything is all written and debugged, training it intensively on the cluster is the next step.