A Review of the AlphaGo game-playing AI

After implementing our first game-playing agent, we couldn't wait to review the DeepMind Team paper on AlphaGo[1]. In our work, we built a game tree of the different possible moves in a game of Isolation, using techniques such as iterative deepening and alpha-beta pruning to evaluate branches without going all the way down. To assign a "value" to each branch, we designed several evaluation functions by hand. On a 7x7 isolation board, the branching factor is fairly small allowing us to go fairly deep down the game tree. In the game of Go, where the branching factor is ~250 and a typical number of moves is ~150, one needs better techniques when deciding which branch of the game tree to select.

Design Goals

Beyond its obvious main goal of reducing the search space of possible moves, it seems to us the DeepMind Team might have had two additional goals, here:

Use as little domain knowledge as possible: The AlphaGo paper shows how **deep learning** can be used to train a model that can assign values to nodes (board positions) in the game tree (using a "value" neural network) and which branch of the game tree to select during play (using a "policy" neural network), without domain knowledge (sophisticated but brittle human-designed heuristics).

Go beyond supervised learning: Supervised learning is great when you have a lot of labels, and the AlphaGo team certainly had access to a lot of them, but they couldn't develop a system better than a "strong amateur" just using those. However, using **reinforcement learning**, pitting the system against other instances of itself, they managed to train a much stronger game-playing agent.

System Design

AlphaGo uses **neural networks** and **Monte Carlo Tree Search (MCTS)**. Three policy neural networks are used to decide which moves to investigate and which ones to play. They were trained to identify promising moves from a 19x19 image of the Go game board. A fourth neural network, called the value network, looks at one of those images and assigns a "goodness" value to the current player position (the equivalent of the evaluation function in our Isolation game-playing agent). MCTS uses the four networks to evaluate the value of each game position in the search tree and identify the most promising move.

Network Design and Training

The **Supervised Learning (SL) policy network** is a 13-layer deep CNN trained on 30 million Go game positions. Given a game position, it predicts the next move, i.e. the *most likely move*. It alternates convolutional layers followed by ReLU activations and is capped by a huge softmax that allocates a probability to each legal move.

The next step uses a **Reinforcement Learning (RL) policy network** to also predict the next move, albeit the *best next move* rather than the most likely one. Not only does it have the same structure as the SL network, it started as the same network. But, by making it play against itself 1.2 million times and beat earlier incarnations of itself, keeping the network weights of the winner, it became much stronger.

There is a third-policy network called the **Fast Rollout (FR) policy network**. Like the SL network it was trained to predict the next move, but it is a thousand times faster than the SL network. While not as accurate, but because it is so much faster, it is used to play out the rest of the game, hence, predicting the most likely outcome following the predicted next move.

The **Value network** estimates the probability that the current position will lead to a win or a loss for the current player. When it was first trained on the same data than the SL policy network, it severely overfitted. To improve its ability to generalize, the AlphaGo Team trained it on the games collected during the during reinforcement learning phase instead (~30M human games vs 1.5B self-play games).

The Game-playing Agent Tree Search

From the current position at the top of the game tree, down edges to possible moves carry an action value Q hat captures how good a potential move is. The game agent searches the tree for the best move in **four different phases**. First, the Agent chooses the edge with the highest Q and explores down that branch (**selection phase**).

When it reaches a leaf node to explore further, it creates a down branch and *runs the slow SL policy network to come up with a strong candidate move* (**expansion phase**). It then does two things (**evaluation phase**): a/ run the value network *once* to evaluate this new position, and b/ use the FR network to playout from that position to the end of the game trying *as many runs as possible* within a time limit. After this, it will propagate the information it collected during those runs and bubble it up the search tree (**backup phase**). If many of the runs ended badly, it will adjust the Q value for that branch down. If it often did well, it will bump it up.

After the game-playing agent has used all of its allocated time evaluating many different branches, it chooses the move that yielded the highest Q value.

Results

The paper presents three sets of results for two different implementations of AlphaGo (one distributed, one not):
* Both versions of AlphaGo significantly outperforms previously existing Go-playing AIs and are better than the best European player.
* Even without using all of its neural networks, just using the value network, AlphaGo performs almost as well as other AIs.
* Finally, by throwing more hardware at the problem, AlphaGo performs even better.

References

[1] Mastering the game of Go with deep neural networks and tree search, by David Silver et al @ https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf