Learning to play Connect Four through deep reinforcement learning

Clemens Macho

cmacho@stanford.edu

Video: youtu.be/pEbqpMM3kzo

Introduction

In 2015, the British company DeepMind developed a computer program AlphaGo that achieved superhuman performance in the game of Go by utilizing deep neural networks and tree search [1]. Later they created AlphaZero [2], a version of their program that could also play other games such as Chess and Shogi. For this project, I reimplemented the algorithms behind AlphaZero to create a version that plays Connect Four.

The game of Connect Four was solved through other methods in the 1980s, and the correct theoretical value of any position can be computed quickly by using a minmax algorithm with alpha beta pruning and some further optimizations. This allows me to generate many interesting board positions quickly and to evaluate my program in ways that would not be possible for Chess or Go (e.g. by comparing its moves to the theoretically optimal moves). Since Connect Four is much simpler than Chess or Go, I can also use a smaller neural network and less computation than were used by DeepMind.

Components

- **Neural Network:** The heart of the algorithm is a convolutional neural network f_θ with parameters θ. It takes a board position s as input and outputs both a value for the position and a policy (which move to play). $(\mathbf{v}, \mathbf{p}) = f_\theta(s), \mathbf{v} = (x_{\text{win}}, x_{\text{lose}}, x_{\text{draw}}), v = x_{\text{win}} - x_{\text{lose}}$.

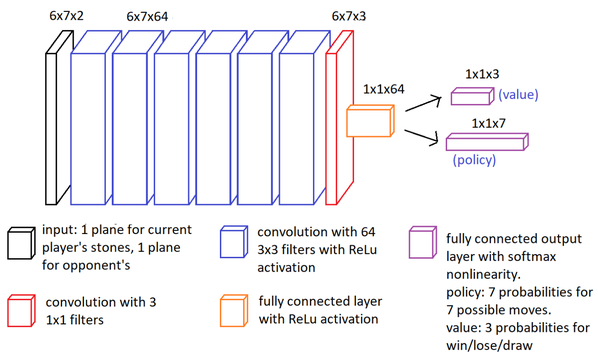

Figure 1: Neural network architecture. We use six convolutional layers with ReLu activations.

In the first phase of my project, the parameters θ were tuned by supervised learning based on a labeled data set that I have created. In a second phase, the program is learning through self play, generating its own training examples.

- **Monte Carlo Tree Search (MCTS):** Given a state s, MCTS chooses an action to play by building up a search tree with root s. Whenever a node s' is added to the search tree, it is evaluated by the neural network $(\mathbf{v}, \mathbf{p}) = f_\theta(s')$. The values and policies that are computed by the neural network guide the tree search. Details are in [2] and in my final report. MCTS only searches a small number of states before deciding on a move. In my implementation, each run of MCTS evaluates $T = 100$ positions during training (self play) or $T = 2000$ positions during actual play.

Supervised Learning

The correct theoretical value and optimal policy of any Connect Four position can be computed within a fraction of a second using an appropriate implementation of minmax with alpha beta pruning and further optimizations. I used such a program to label data for supervised learning and also in order to generate data both for training and for testing. We distinguish between two different minmax values and corresponding minmax policies.

- **weak minmax:** chooses moves to get the optimal outcome for the game (win, lose or draw)
- **strong minmax:** when winning, chooses moves that lead to a victory as early as possible; when losing, chooses moves that lead to losing as late as possible

I used cloud computing (Google Cloud) to generate a training data set of 1,000,000 positions and a validation data set of another 112,000 positions. Each position is labeled with its weak minmax value (1,-1 or 0) and optimal policy according to strong minmax. The positions were sampled from approximately 200,000 games where each move was chosen either according to a strong minmax policy or uniformly at random.

The neural network parameters θ were fit to the training set using stochastic gradient descent with a crossentropy loss function and L2 regularization.

Result: We evaluate the neural network on a validation set of 69,569 samples. These were filtered from a larger validation set of 112,500 positions by first removing all examples that also appear in the training set and then removing all duplicates in the remaining data set. We evaluated four metrics shown in the following table.

accuracy for value	0.821
mean squared error for value	0.320
accuracy for policy	0.843
average probability of choosing an optimal move	0.689

Here, accuracy for value means the fraction of positions where the neural network assigned the highest probability to the correct value (1,-1 or 0). Accuracy for policy means the fraction of positions where the neural network assigned the highest probability to a move that leads to the best strong minmax value. We also evaluated the neural network by having it play full games (with MCTS). See the example games on the right.

Self Play and Reinforcement Learning

To train through self play, the program alternates between two steps:

- **Step1:** Generate data by playing several thousand games against itself, choosing each move with MCTS guided by the current neural network f_θ. From each game one position is sampled and labeled with the improved policy π as computed by MCTS. As a value for the position, we use the value $z \in \{1, -1, 0\}$ of the game's final outcome, from the perspective of the player whose turn it is in the sampled position.
- **Step2:** Update the neural network parameters θ by performing SGD updates using the data generated in Step 1.

Result: The program is currently training on Google Cloud. Once finished, it will be evaluated according to the same metrics as in the table above and it will also play against other versions that were trained through supervised learning. Results will be included in the final report.

Example Games

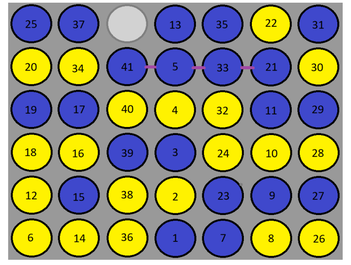

Figure 2: neural network agent (blue) versus strong minmax agent (yellow). Blue wins at move 41.

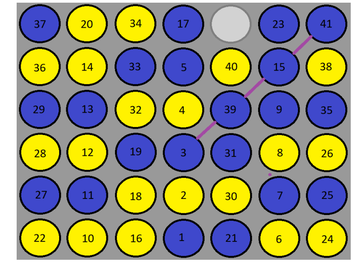

Figure 3: neural network agent (blue) versus strong minmax agent (yellow). Blue wins at move 41.

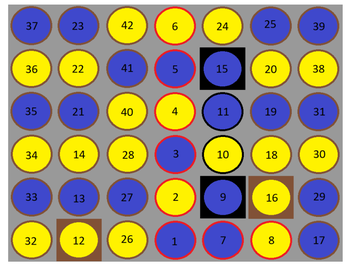

Figure 4: Clemens Macho (blue) versus neural network agent (yellow). Game ends in a draw.

Figure 5: neural network agent (blue) versus Clemens Macho (yellow). Blue wins at move 39.

Each move has been circled in a color corresponding to the (theoretical, weak minmax) value of the resulting position from the perspective of the neural network agent: red for a losing position, black for a winning position and brown for a position that leads to a draw. Moves that were mistakes (and thus changed the value of the game to the detriment of the player making the move) are marked with a square around them.

As we can see, the neural network agent plays perfectly when playing against the minmax agent but does make some mistakes when playing against a human player. This is probably because it was trained on data that was sampled from games by a strong minmax agent (with some randomness added). So the positions which arise during play against a strong minmax agent are more likely to be positions that were also in the training data (or similar to such positions).

References

[1] A. Silver, Davidand Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan. 2016.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.