Learning to Play Connect Four from Game Images

Rotem Hemo, Beomjoon Kim, Yo Shavit, Aradhana Sinha

***Abstract*— We design a system to play Connect Four given access to an image of a game board. We experiment with three different deep-learning architectures: expert policy estimation, state value estimation, and a hybrid architecture that uses classical planning (MCTS). We compare our trained networks with a variety of standard players and discuss their relative strengths.**

I. Introduction

In this paper we aim to learn to play Connect Four given only the instantaneous image of a game board. This problem requires a combination of machine vision (to parse the board image into an actual board state) and action planning (to choose the best action given a state).

This problem requires both the ability to detect useful features from a given board image, and the ability to plan actions amid long horizon and large branch factor.

In 2016, Google DeepMind's AlphaGo solved a similar challenge for the game of Go. AlphaGo relied on hand-tuned features of a board as an input to a neural network, and focused on the planning problem [1].

Much like AlphaGo, our networks are trained in a supervised fashion using "expert" games between two classical-AI computer players (two Monte-Carlo Tree Search [4] players with half-second thinking time). We train our network to learn to play using three different deep-learning architectures:

1) A **policy network** to predict what action an expert would take in the state
2) A **value network** to predict the likelihood that an expert would win from the current state
3) An **augmented Monte-Carlo Tree Search (AMCTS)** which combines the policy and value networks with a classical tree-search method to estimate the best next move

This problem is challenging in a few major ways. First, extracting a meaningful playing strategy from example experts that choose moves stochasticly is a high-noise, low signal-strength problem. Second, we are trying to learn an estimated likelihood of a far-away future reward without knowing any meaningful game logic. Finally, while Connect Four is technically a solved game [5], choosing a move visually adds state estimation to the already difficult task of policy-learning, a combination that has yet to be meaningfully explored.

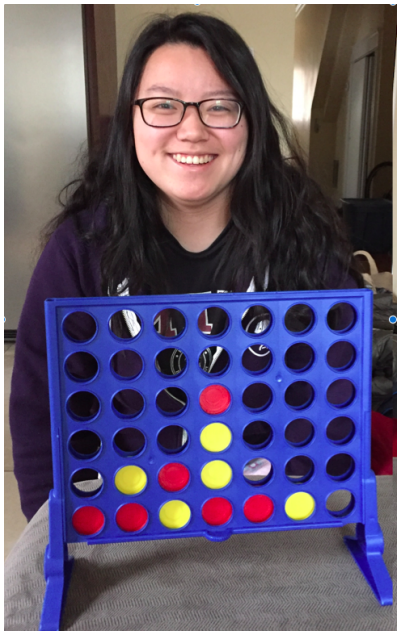

Fig. 1. Example Connect Four Board with beloved TA

II. Related Work

A. Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS), also known as UCT search, is a classical planning strategy which asymmetrically expands the game tree to estimate the quality of different moves [4]. On a high level, this tree describes the set of ways the game could play out: "if red moves here, then black might move here, in which case red should move here..." On a technical level, the algorithm consists of a tree of nodes each representing a game state, and each node can be "expanded" by taking a valid action at that node to reach a new game state. The algorithm consists of two parts: descending through the tree to determine which node to expand next (the "tree policy"), and then quickly estimating the quality of the expanded node (the "default policy"). In classic MCTS, the "tree policy" used is known as the

UCT heuristic (which encourages asymmetric expansion of the tree), while the "default policy" evaluates the current state by randomly assigning actions to each player until one side wins. The tree is expanded for a fixed time limit, and when the time is up the most-expanded action (which is likely the best-performing) is chosen. The longer the algorithm is given to "think", the more it will expand the tree and the better its choice of move will be.

Our AMCTS algorithm improves on the MCTS framework by using trained networks as heuristics [2]. Specifically, the "tree policy" is augmented by warm-starting the UCT estimate using the output of the value network. The "default policy", rather than playing the game out by generating moves randomly, generates moves using the policy network, causing the simulation result to much more closely correlate to the true quality of the state.

B. AlphaGo

AlphaGo [1] is an artificial intelligence system developed by Google DeepMind to play Go. Like Connect Four, Go is a zero-sum game where two players alternate moves until one wins. Go is legendarily complicated, with a 19x19 game-board and frequently-changing game state making it one of the most complicated board games for humans (and AI) to play. To overcome the difficulty of estimating the quality of a game state, AlphaGo combines MCTS planning with three trained networks: two policy networks (trained to predict expert moves and used for the MCTS tree policy and default policy respectively) and one value network (trained to predict the estimate generated by the AlphaGo system in a reinforcement-learning scheme).

Our architecture bears similarity to AlphaGo in that we too combine policy and value networks within the framework of a classical MCTS planner to leverage their differing strengths. The problem of this paper, however, is fundamentally different: AlphaGo is provided with an array representing the game grid and is only asked to generate an action, whereas we aim to interpret the raw image of a game board and also generate an action based on that interpretation. On the other hand, Go requires a far more complex state representation than Connect Four does, and has a much longer horizon (many more moves are played each game) and higher branching factor (many more moves can be taken at any one time). Theoretically this suggests that estimating the value of a current position in Connect Four should be easier than it was for AlphaGo, which had to train convolution networks with dozens of layers over the course of several months.

C. Atari

In 2015, DeepMind achieved the first deep learning model that learned to play a game (Atari 2600) better than human players [3], using only the screen pixels as an input and score gains as rewards. Their reinforcement learning model was a CNN trained with a Q-function. Q functions represent the predicted future reward for a given state and action.

The Atari paper and this Connect Four paper both attempt to derive meaningful structure using only pixels as input, and both aim to generate an action. The training set of both involve learning from a series of highly correlated states.

The horizon until a reward, however, is much longer for Connect Four than it is for Atari. The reinforcement learning methods described in the Atari paper do not work for a game with a slightly longer horizon (e.g. Montezuma's Revenge). Hence, in our study we rely on supervised training methods, and also integrate a more robust classical planning strategy (MCTS).

Furthermore, in Connect Four the action used to reach a state does not impact the value of the state. A given board for a given player has the same potential for future success regardless of how it was arrived upon. For this reason, we chose to train value functions that represent the predicted future reward given only a state.

III. Methodology

We take several different approaches to learning to play Connect Four. Most of these methods are motivated by approaches taken in AlphaGo, which have already proven to be effective for a much more complicated game.

First, we present a method that learns to map a board image to an action, where an action is defined as the board column chosen for the next token. This approach, which we call a "supervised policy" network, is similar to AlphaGo's Supervised Learning Policy Networks in that we take as training data expert actions in various states and perform supervised learning on the dataset. The major difference is that AlphaGo's input is a hand-designed feature of the board, while we use an image of the board. Also, unlike DeepMind, our lack of access to a large database of expert game playthroughs forced us to resort to using MCTS-vs.-MCTS playthroughs to generate expert play data.

Second, we present a method that learns to map a board image to a "value" of the board, representing the

expected likely reward from that board position. This "value" is highly dependent on the strategy used to play the game - if one policy does well from a certain state, and another does poorly, they should yield different expected values. AlphaGo trains its value network to predict the performance of its policy-network's policy. In our case, our supervised policy network is not as reliable, so we resort to learning the "value" of a state given an MCTS player.

Lastly, we construct an augmented MCTS with the learned value-network and policy-network used to guide roll-outs (i.e. tree expansions). This too is similar to the approach in AlphaGo, slightly simplified: we linearly combine the heuristic estimate generated by the value network with the value estimated by MCTS, and then use the policy-network for roll-outs.

We trained all our networks using 4 TitanX GPUs for 6 hours each.

A. Supervised Policy Network Learning

Our policy network consists of four convolutional layers with kernel size of four, with no zero padding in the first layer but with zero padding in the latter third in order to preserve the original shape. We have max pooling layers in between each of the convolutional layers of size 4 by 4, followed by a dense layer that outputs seven numbers that represents probability of an expert player choosing each of the columns.

In order to generate a dataset for our policy network, we played two MCTS players with 1 second time limit against each other. We then took winning players' trajectories from these game plays, and created the dataset D^{π} which takes the form

$$D^{\pi} = \{\tau_i\}_{i=1}^{n},$$
$$\tau_i = \{s_t^{(i)}, a_t^{(i)}\}_{t=1}^{H}$$

where s, the state of the board is represented with 144 by 144 and 3 channels RGB image, and action $a \in [0,1]^7$ is a seven dimensional vector that represents the probability that we should select one of the columns in the board s. We trained on a total of 20,000 (s, a) pairs.

We then feed the dataset D to our CNN to learn the policy π that learns a mapping

$$\pi : \mathbb{R}^{144\times144\times3} \to [0,1]^7$$

by training the network as a classifier with cross-entropy loss. The prediction accuracy was about 43%.

B. Supervised Value Network Learning

To represent a value network, we use the same architecture as the policy network but with two additional layers, one that outputs linear activation and the last output that has $\tanh$ activation. In order to train the value network V, we play two MCTS players with 0.5 seconds time limit. Then, we take red stone player's trajectory, and create the dataset

$$D^{V} = \{\tau_i\}_{i=1}^{n},$$
$$\tau_i = \{s_t^{(i)}, v_t^{(i)}\}_{t=1}^{H}$$

where $v_t^{(i)}$ represents whether by at the end of the episode, the red stone player won from the state $s_t^{(i)}$. The total number of (s, a) pairs was 50000. If the red agent has won, it takes a value of 1, and the agent loses, then it takes a value of -1. Using this dataset, the value network learns the mapping

$$V : \mathbb{R}^{144\times144\times3} \to [-1,1]$$

by minimizing the mean squared error on the dataset D^{V}. Our resulting function estimates the expected reward (from -1 to 1) of the red player from the current board image.

C. Augmented MCTS

In order to utilize π and V that we learned, we augment the standard MCTS with π as the rollout policy and V to warm-start MCTS's estimates of move quality (and thus to guide the direction of the expansion of the tree). The intuition here is that we can choose better actions in a shorter amount of time by biasing the MCTS search towards the more promising region of the state space, as estimated by π and V.

More specifically, the Q function that the MCTS algorithm uses to descend the tree is

$$Q(s,a) = \gamma^{n_t} \cdot V(s') + (1-\gamma^{n_t})\frac{\sum_{i=1}^{n_t} z_i}{n_t}$$

This equation initially warm starts with $V(s')$ when we never previously explored the current state, and eventually converges to $\frac{\sum_{i=1}^{n_t} z_i}{n_t}$ as n_t, the number of time we have explored the state, increases.

We would like to use π to guide the game rollouts that estimate the winner from the current board position, but these rollouts must be stochastic to eventually cover the whole search tree. Therefore we execute a random action with probability q and execute the action chosen by π with probability $1-q$. (In practice, we chose $q = 0.3$.)

IV. Experiments

Performance was assessed in three ways:

1) Game performance against computer players of various difficulties
2) Performance under specific board scenarios
3) Analysis of hidden features.

A. Performance Against Other Computers

We play AIs against each other 20 times (alternating which player goes first, as the first mover has an advantage) to see how the players compare. Our baseline players include:

- MCTS with computation times of 0.05, 0.2, 0.5, 1.0, and 2.0 seconds.
- Random Player that randomly chooses an action.
- Random Player that tends to play tokens in the three center columns.

We play each of the above AIs against those we generate:

- Value Network policy
- Policy Network policy
- Augmented MCTS with both the Value network and the Policy network, with computation times of 0.5, 1.0, and 3.0 seconds.
- Augmented MCTS with only the Value network, with a computation time of 1.0 seconds.
- Augmented MCTS with only the Policy network, with a computation time of 1.0 seconds.

B. Performance Against Specific Board Scenarios

To better understand the behavior of a given player, we examine its decisions under very specific conditions where the right move or moves are obvious to a human player (See Figure 2). Through these specific situations, we sought to cover the following scenarios:

- Obvious win for current player (horizontal, diagonal, vertical, varying columns)
- Obvious win for current player in two locations (horizontal, diagonal, vertical, varying columns)
- Block obvious win for opponent (horizontal, diagonal, vertical, varying columns)
- Place a token that creates a trap.
- Place a token to make opponent not create a trap.
- The above scenarios in sparse and crowded environments

C. Policy Learning

Out results suggest that the policy network performs very well against all the other one-step players (random, center-oriented, the value-network, and even short-thinking-time MCTS). It cannot, however, compete with the longer computation time MCTS algorithms, which can likely understand more complicated sequences of future moves. Interestingly, it seems to match the ability of some of the AMCTS algorithms.

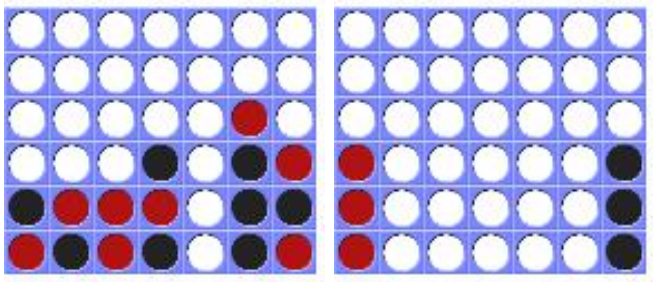

Fig. 2. (left) Example of a situation where Red should play a token in the third column from the right to induce a winning scenario for itself. A correct decision on this image involves the ability to see forward three steps, and do more than simply avoid the opponent's win condition. (right) An example of where the obvious move is to complete the vertical four-in-a-row. This particular scenario is never seen in any of the training data since the MCTS players overwhelmingly pick the center nodes earlier in the game. Correct performance on this image indicates an ability to transfer learning to new states.

When faced with specific scenarios, the policy network is able to detect the right move to induce a winning scenario for itself within the first three moves. In the situation where an immediate win is not clear, but column choice affects which player eventually wins (players take turns avoiding placing a token in a column that would induce a win for the other player), the policy player is unable to distinguish the right move. That is, it seems to be unable to count forward even the obvious moves for ten steps.

D. Value Learning

Our value network performed very poorly (and likely contributed to the rather poor performance of our AMCTS player).

When faced with specific scenarios, the value network does assign values of one to the immediate win scenarios. It is unable to distinguish obvious win scenarios that are 2-3 steps away. Value network does not discriminate well between the relative probability of success between boards that are crowded–roughly between a third and two-thirds full. This is probably why it performs poorly relative to the other strategies.

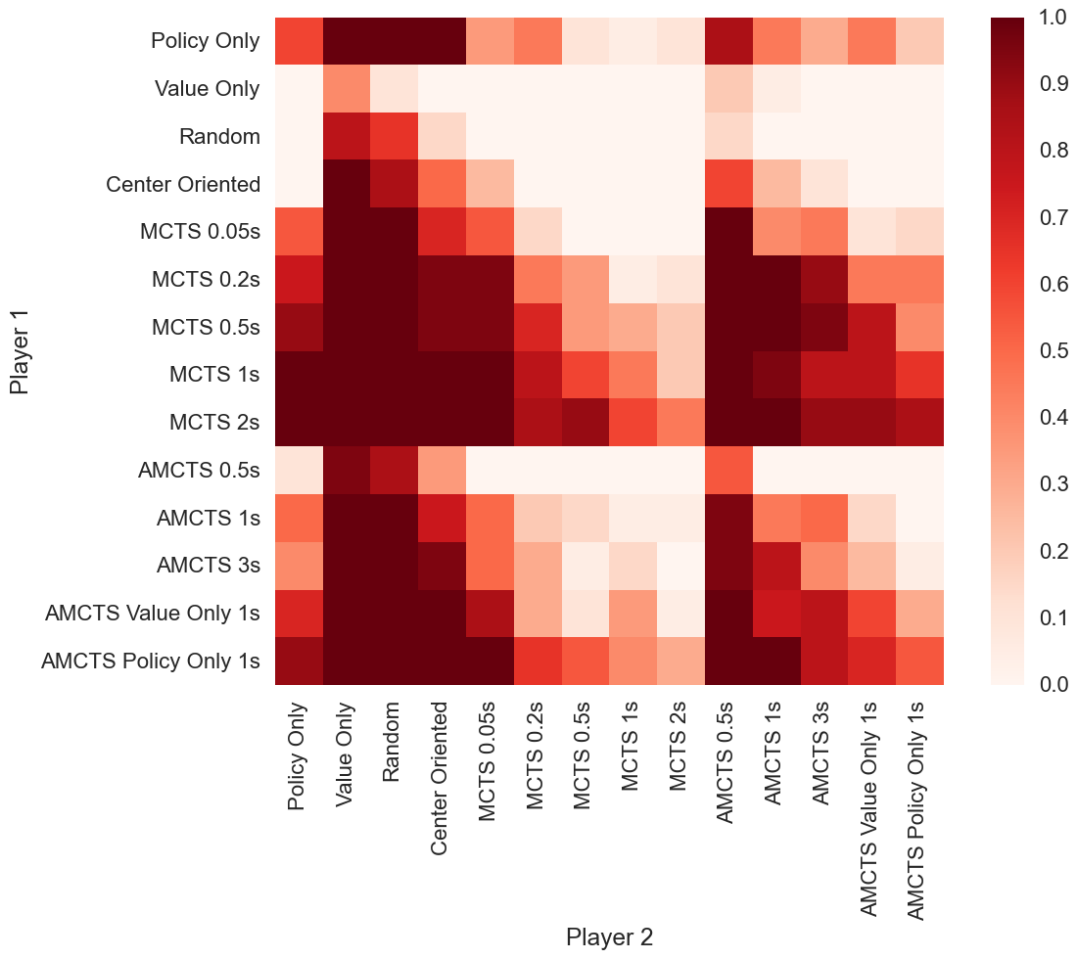

Fig. 3. A chart of comparisons of different players against each other. The value at each grid position is the fraction of games won by "player 1" (y axis) against "player 2" (x axis) when player 1 moves first.

E. Augmented MCTS

The AMCTS players which used both value and policy networks significantly under-performed MCTS players with the same amount of computation time. In fact, the 1 second AMCTS seems to have even had trouble beating the "center-oriented" board-independent strategy some portion of the time. This is attributable to both the previously-described problems with the value network, and the surprising effectiveness of MCTS on Connect Four (further detailed in the Discussion section).

However, if we restrict AMCTS to only using the policy network (meaning that the tree-policy is classic MCTS, and the board roll-outs are done using the policy network) the policy-only AMCTS algorithm actually outperforms most of the AMCTS solvers. When given the first move advantage, this policy-only AMCTS given 1-second processing time is able to defeat an MCTS with 2 seconds of processing time 43% of the time (See Figure 3). This suggests that the search tree using the policy network is being expanded much more efficiently.

F. Analysis of Hidden Features

In order to have a better understanding of how the network behaves, we analyzed the hidden layers of our Supervised Policy Convolutional Neural Net (CNN). Using the following approach, we analyzed the hidden layers in order to find neurons that are in charge of specific tasks. First, we feed the network with a few hundreds of different board games and take the top K active neurons. Second, we select the board images which activated these neurons. Lastly, we find the correlation between the active neuron to the board image which triggered it.

As we can see in figure 4, for example, the activated neuron "recognizes" tokens which are blocked by a

opponent's token. We can note the high activation level by the concentration of white pixels, which represent very high value.

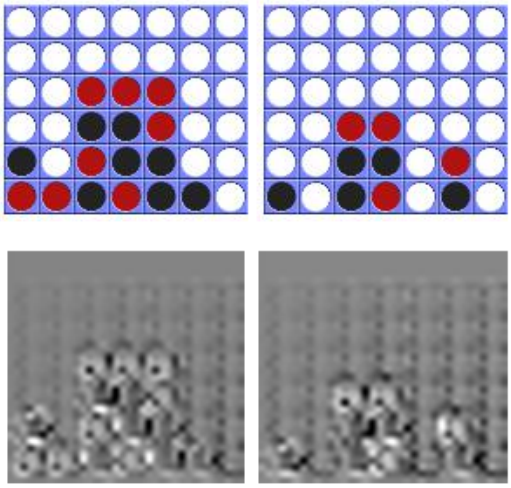

Fig. 4. (Top) Two board images with tokens the are blocked by the opponent tokens (Bottom) the corresponding neuron visualization which shows that the neuron is activated by this specific configuration.

V. DISCUSSION

One of the clearest observations about our AMCTS algorithm is that it generally loses to a pure MCTS algorithm with the same computation time. The sources of this problem are two-fold, both due to an initial noisy dataset, and the structure of Connect Four.

The noise in the initial dataset D^V may have compromised the training. For the same board image, the MCTS agent sometimes leads to a winning board and sometimes it does not. We aggregated all the data together, however, giving noisy labels, and making the value network more difficult to train. For the boards closer to the terminal position, this noise is significantly smaller than the ones that are far away. This is why in AlphaGo, their system played the first few steps of the game using an expert-supervised network and then followed it up with an RL policy network (and then only took part of the data generated by the RL policy network as the training data for their value network). In our case, we did not have a computing resources that Google DeepMind had, and have to use all the data for every training step.

Further, it appears that in Connect Four the horizon and branching factor are sufficiently small that expanding nodes randomly is more time-efficient than expanding nodes intelligently (making MCTS successful over AMCTS with an equivalent thinking time). This is precisely the opposite of the situation in AlphaGo [1]. Basic MCTS fails miserably at Go because the game has such a long horizon that random play-outs are uncorrelated with the true likelihood of winning. Therefore in AlphaGo, unlike in our system, using deep-learned heuristics and rollout policy is a much more time-efficient way to estimate likelihood of winning.

REFERENCES

[1] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484-489.

[2] Gelly, S., & Silver, D. (2007, June). Combining online and offline knowledge in UCT. In Proceedings of the 24th international conference on Machine learning (pp. 273-280). ACM.

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[4] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games, 4(1), 1-43.

[5] Allis, L. V. (1988). A knowledge-based approach of connect-four. Vrije Universiteit, Subfaculteit Wiskunde en Informatica.