An Application of Neural Networks to a Non-Deterministic Game of Imperfect Information

Nicholas Lydeen
Faculty Mentor: Dr. Chris Ahrendt
University of Wisconsin-Eau Claire

1. Introduction

Current research into game-playing neural networks emphasizes deterministic games of perfect information and training with "expert knowledge" such as known successful strategies. As an example, Google's *AlphaGo*, a neural network agent designed to play the game Go, was trained with over 30 million different board positions drawn from approximately 160,000 different games. This expert knowledge gives *AlphaGo* a solid baseline for future play.

We studied the application of neural networks to the game *Lost Cities*, a *non-deterministic* game of *imperfect information*, *without* providing such expert knowledge, and designed several neural network agents and implemented an agent using Monte Carlo tree search for comparison.

2. Lost Cities

Lost Cities is a two-player 60-card board game in which the players compete to mount expeditions to any of five "lost cities". The cards are numbered and represent progress toward a particular expedition. Players progress on an expedition by playing cards on their boards. At each turn, a player must either play a card on their board or discard a card and must draw a card from either a discard pile or the draw pile, maintaining a hand of eight cards. Cards must be played onto the player's board in increasing order (although not necessarily consecutively), and there is an investment penalty for starting an expedition.

The game ends when the draw pile is depleted. A player's score is completely determined by the endgame state of their board. Each suit is scored individually. An untouched suit scores no points, and if a player plays even a single card of a suit, they incur a -20 point investment cost. Special investment cards act as multipliers on the suit's final score. The rank of each other card determines the number of points contributed to that suit's score. If eight or more cards are played on a suit, a 20 point bonus is awarded.

Example. Suppose that one suit of a board has two multiplier cards and cards of rank 1, 3, and 5. Then the score of that suit is $(2 + 1)(1 + 3 + 5) - 20 = 7$.

3. Computer Representation

We implemented the game controller in Python. The controller implements the game mechanics and controls the visibility of the game state.

Cards are represented as `(suit,rank)` tuples and moves are represented by `(hand index, action, draw index)` tuples, where `hand index` refers to the index in the agent's hand whence the agent wishes to play a card, `action` determines whether the agent wishes to play or discard the card, and `draw index` indicates whence the agent wishes to draw a new card.

Agents implement a standard interface for interacting with the controller, so that the controller is not bound to a particular agent implementation.

4. Neural Networks – Theory

Neural networks are machine learning models inspired by biological nervous systems.

Neural networks are directed graphs of *neurons* with associated *weights*. These weights are determined by *training* the neural network.

For a single neuron, the output is computed by taking the dot product of the input vector with the neuron's weight vector and then applying an *activation function*. For multi-layered neural networks, the output of the first layer is used as the input to the second layer and so forth.

To train a neural network, we must have a set of input vectors together with their respective desired output vectors, collectively forming the *training set*. A *loss function* determines the extent to which the network's output vector differs from the desired output vector, and the weights of the network are adjusted in proportion to the negative of the gradient of the loss function.

This procedure is straightforward for single-layer networks. Multi-layer networks require a method known as *back-propagation*, in which this method is extended by estimating the extent of the error due to each neuron, and adjusting their weights accordingly.

5. Neural Network – Implementation

We used the *Keras* machine learning library.

The input vector consists of 61 inputs: 60 inputs encode the location of each card, and one input gives the number of cards on the draw pile.

The output vector consists of 96 outputs: one output for each possible move tuple (not all necessarily valid for a given game state), normalized for interpretation as a probability distribution, whence the agent draws its next move.

The network is aggressively trained against illegal moves and positively (negatively) reinforced for positive (negative) scores by adjusting the probabilities in the distribution.

6. Monte Carlo Tree Search

In a *Monte Carlo Tree Search* (MCTS), the agent traverses a random subset of the game tree according to some sampling distribution, and must make a trade-off between exploration (breadth) and exploitation (depth).

The agent maintains a copy of the game state, using sentinel values to represent hidden cards, such as those in the opponent's hand. When a simulation requires knowledge of an unknown card (e.g. in simulating the opponent's play), the corresponding sentinel value is replaced by an actual card drawn from the set of yet-unknown cards.

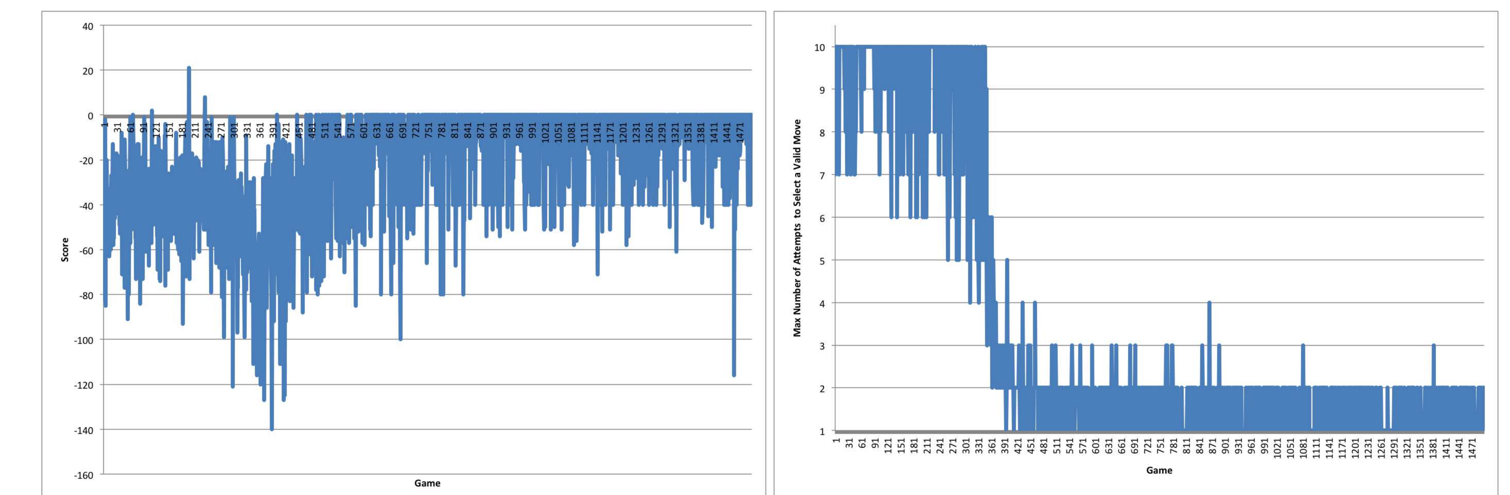
We implemented a neural-network-assisted MCTS, in which the Monte Carlo sampling distribution is learned by a neural network. The neural network helps the MCTS routine focus on branches that are more likely to be favorable.

7. Analysis – Straight Neural Network

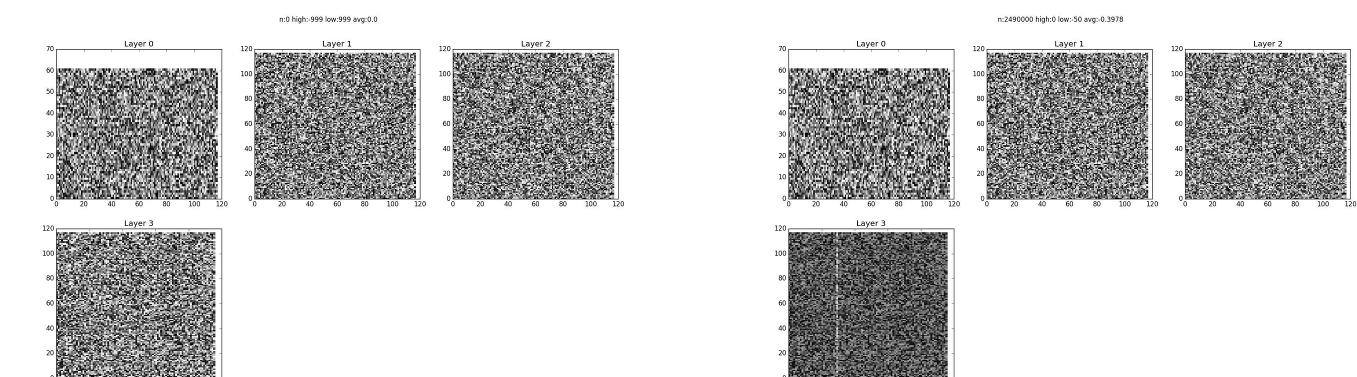
Although the straight neural network agent (without Monte Carlo) does not perform admirably, we made several interesting observations.

We noticed that heavily training illegal moves is successful in the sense that it reduces the number of proposed illegal moves, but at a high cost. The network eventually nearly stops attempting to play cards on the board at all, as, if done randomly, playing a card is often illegal. This consistently results in a "safe" zero score.

The following graphs give an example of how an agent's score and number of sampled illegal moves change over time:



For the sake of comparison, these are the initial and final weights of a particular neural network agent, before and after 2.5 million games:



8. Analysis – Monte Carlo with Neural Network

At present, the Monte Carlo neural network agent is a marginal improvement over the naive Monte Carlo agent. Nonetheless, we made several interesting observations here as well.

For certain training implementations, the Monte Carlo neural network also tends toward the riskless zero score. However, a small number of trials suggests that the Monte Carlo neural network instead begins to learn to manipulate the discard pile to minimize the opponent's score.

References

[1] Chollet, François, *Keras*. GitHub, 2015. `https://github.com/fchollet/keras`

[2] Russell, Stuart and Norvig, Peter, *Artificial Intelligence: A Modern Approach, Third Edition*. Pearson Education, 2010.

[3] Silver, David, et al., *Mastering the Game of Go with Deep Neural Networks and Tree Search*. Nature, 529 (75–87): 484–489.

Acknowledgments

- Department of Mathematics, UW-Eau Claire
- Office of Research and Sponsored Programs, UW-Eau Claire
- Poster created with LaTeX