An Application of Machine Learning to the Board Game Pentago

Greg Heon
Lilli Oetting

December 14, 2012

Abstract

We present an application of machine learning to the two-player strategy game Pentago. We have explored two different models through which to train our program - a feature-based model and a reinforcement learning model. This paper discusses the relative merits of each, as well as the obstacles we encountered using each model.

Introduction

In the last century, intelligent gameplay has become a classic problem in artificial intelligence. Computers have an unmatched ability to solve problems with brute force, but games often contain too many possibilities for a computer to calculate the best move in a reasonable amount of time. To play a game well in a reasonable amount of time, a computer must transcend brute force and find a smart move without considering every single possible move made in the future; in essence, it must start to understand the game.

Pentago is a two player strategy game composed of four quadrants containing a three by three grid each. A players turn consists of placing one piece in an open space and then rotating one of the quadrants 90 degrees in either direction. Each player must complete both of these actions on a given turn. Play continues until a player achieves a consecutive sequence of five of his or her own pieces or the board is filled, resulting in a tie. The five-in-a-row sequence can be contained in any row, column, or diagonal on the board.

We considered several games before settling on Pentago. It is feasible for a machine learning application because it is deterministic, strategic, and unsolved. We avoided games that involved aspects of randomness, like dice or many card games, since we thought that given our limited computing power we would not be able to run enough training trials to average out the randomness.

We ultimately implemented a feature-based model that trains the weights on each feature, and a reinforcement learning model that trains a policy. Both models are trained through self-play. The details of each are discussed below.

Feature-Based Model

The first model we implemented was based on parameterized features and a greedy policy that evaluated board positions and chose the move that would lead to the most highly valued result. Because Pentago is a deterministic game, this strategy can lead to near perfect results if it is able to look an unlimited number of moves into the future when deciding which move to make. Our implementation is based on a large number of pre-defined features and associated weights that are built up based on game results.

The Model

Feature Selection

A variety of features associated with success in Pentago are used to evaluate potential moves. The final model contained 35 different features of gameboards, ranging from features obviously linked to success (i.e. number of 4-in-a-row sequences) to those whose importance we were unsure of (i.e. number of corner pieces). We decided to include a large number of features and allow the weights developed during self-play determine the relative importance of each feature and downplay features that are not shown to have a strong affect on the game result. Starting with a wide variety of features limits the affect of our personal bias in the eventual performance of the program and allows the relevant features to be selected through the weights determined during gameplay.

Weight Determination

The success of this model is based upon its ability to develop weights that accurately reflect a feature′s correlation with success in the game. Weights are initialized according to reasonable estimates of which features we suspect to be correlated with success and refined through self-play. This process involves two instances of the program playing against each other. In each game, one player uses the current set of weights and the opponent uses this set adapted by a slight random deviation. The two instances play a game, and the winning players weighting set becomes the current set in the next round. Through this process, we hoped to converge on an optimal set of weights.

Results and Setbacks

Ideally, this model would be able to look an unlimited number of games into the future when determining a move. Unfortunately, limits on our computing power make this impossible. We reduced this model to a minimax strategy that evaluates all possible moves for the active player, then all possible moves his or her opponent could make in response, and finally the moves that the active player would be able to make after the opponent has played. The objective is to minimize the threat posed by the opponent on his or her next move. Given that there are more than 250 possible moves per turn in the beginning of the game, evaluating 15.6 million boards in order to make a single move was too computationally intensive for the computers available to us. Even looking only three moves ahead was too slow to run the number of games necessary for weight convergence. We eventually switched to a model which evaluates only the immediately reachable boards and doesnt have information about any future moves. This strategy still produced a program intelligent enough to beat a random move generator but much less intelligent than a model which can examine future moves.

The fact that the program must evaluate every possible board based on such a large feature set causes it to run slowly even when examining only a single move. In order to find a more efficient alternative we turned to a reinforcement learning model.

Reinforcement Learning

The Model

Our reinforcement learning model uses a Markov decision process (MDP) approach. The number of board positions, however, is too large for the computing power we had, so we hashed board positions into buckets and ran the MDP model on these buckets instead of the much larger state space that is all board positions. On each turn, the obtainable boards are hashed into buckets, and the move associated with the highest valued bucket is chosen. The state transition probabilities are initialized such that there is an equal probability of being able to move between all buckets and built up by seeing what moves are made during self-play.

Hashing

Hashing into buckets made reinforcement learning a feasible way of approaching Pentago. Choosing the correct number of buckets and the correct hash function were two of the most difficult problems we encountered with the reinforcement learning approach.

Choosing the number of buckets consisted of balancing time constraints, computing power limitations, and the program′s performance. As we increased the number of buckets, each game took longer and more memory was used. Even implementing a sparse array for the transition probabilities matrix didn′t stop our program from using up all the application memory on an iMac with 1000 buckets and less than 5000 training games. On the other hand, having too few buckets meant too many different states would be hashed into the same bucket. If the value of the two boards was significantly different for the two states within the same bucket, the value associated with that bucket is unrepresentative of the boards it contains, which causes poor gameplay.

One solution to this collision problem is a smart hash function. Hash function choice is a time and intelligence tradeoff that we never found a good middle ground for.

The hash function we created originally takes the sum of the product of the the piece placed in each spot (0 for empty, 1 for player 1, and 2 for player 2) and primes (2, 3, 5, and 7) raised to the power of the piece′s spot in the quadrant (0 through 8). It then takes the sum modulo numBuckets - 2 and sets that as the states bucket number. Buckets 0 and 1 are reserved for player 1 wins and player 2 wins respectively. This essentially hashes boards into buckets randomly since there is no effort to group similarly valued boards together. The benefits of randomly hashing is that it is very fast, but the collisions do detract from gameplay.

The alternative to a random hashing is a hashing based on features. Unfortunately, there were no easily calculable features that indicated who was winning, so we implemented a hash function that counted the sum of cubes of streaks (a four in a row would be a streak of four) of each player and taking the sum of the first player′s and five times the second player′s. [1] While this would hash similarly valued states together, it was prohibitively slow. To create a brilliant pentago program, one would need a smart and quick hash function, but the game of Pentago is arguably too complex for such a function to exist. At the writing of this paper, the authors have been unable to find such a function.

Self-play

The training of our program consisted of playing itself repeatedly, learning from a win or loss each time. Because states for both players are hashed into the same buckets, but can be unrelated in terms of value, we thought it best to train two side-by-side models; the first learns to play first, and the second learns to play second. Each has its own probability matrix and value vector.

Exploratory Moves

To ensure that the two computers learned a more universal strategy, we programmed each to make random moves some ϵ percent of the time during training. These exploratory moves forced the computer to consider states it would not see otherwise and also kept the computer from finding what is essentially a local maximum. Some of the data we took was trying to find the best ϵ in terms of performance against our baselines. This is discussed in the Results section.

Updates

Updates on the transition probabilities matrix and the value of each bucket were made after each game ended. Rewards were given only at the end of games. Wins were awarded 10 and losses -10.

Results

Finding the best parameters

We did tests on the following parameters: the number of buckets hashed into, the number of games, the discount factor γ, the convergence tolerance τ, and the percentage of exploratory

[1] Five was chosen because a streak of five means the game is over

moves ϵ. The best parameters we found given our computing limitations were 1500 buckets, 2000 training games, $\gamma = 0.9$, $\tau = 0.001$, and $\epsilon = 20\%$.

Performance against a baseline

We tested reinforcement learning performance against two baselines. The first was a computer that made random moves and the second was our feature-based algorithm. Since the feature based-algorithm is so slow, fewer test games were played when testing against it. Our results with the optimized parameters were winning about 96% of the time against random and 30% of the time against the greedy algorithm, when it made its first three moves at random. While we wished to see better performance against the greedy algorithm, the 96% against random is a clear indicator of intelligent play.

The only other baseline we used was occasional human testing, when we would first train the program and then the authors would play it. While it never won, there were games where it exhibited intelligence in the moves it chose.

Conclusion

Although neither model yielded ideal results, we were able to successfully implement two methods of developing intelligent computer play that performed significantly better than a random-move generator. Based on our results, we believe that our models would result in much more sophisticated play with more time and computing power in order to run a larger number of training games, increase the number of buckets in our reinforcement learning model, and increase the number of future moves evaluated by the greedy parameter-based model.

This project is significant because it explores fundamental questions of machine learning and the power of computers that extend into much more socially significant realms. The basic aspects of this project teaching a program without human input, solving a problem with unknown parameters, and exploring whether a computer can develop a system of complex strategy are important for a wide variety of applications in machine learning.