Machine Learning in Blokus

Alexander Thannhauser | Hao Li
ztannhauser@gmail.com | hao.81.li@gmail.com

Abstract
Machine Learning is a field of study that focuses on giving the computer abilities to learn. It can be applied to many areas such as pattern recognition, data classification and game-play. In this project we applied machine learning techniques to the board game Blokus. In order to give the computer the ability to learn how to play the game, the neural network was built with 5 hidden layers, each 50 neurons wide. By applying the backpropagation algorithm and a modifed Q-learning algorithm to the neural network, we enable the neural network to develop strategies for play. The project is implemented in Python. The Neural Network was trained with over 1,000 games. Significant improvement can be seen.

Introduction
Nowadays, Machine Learning becomes more and more popular. After the success of AlphaGo, the field catches the attentions all over the world. In 1959, Arthur Samuel defined machine learning as a "Field of study that gives computers the ability to learn without being explicitly programmed." Developing traditional AI applications without using Machine Learning strategies is the process that transfers human knowledge to formal computer algorithms through programming languages. This approach gives the computer the knowledge needed to complete the tasks rather than the abilities to learn the knowledge. The traditional AI approach without Machine Learning, solves the Game-Play problem typically in the following steps:

1. Given the current game state S_0, calculate possible next game states $S_{next} = \{s_i | s_0 \xrightarrow{move_i} s_i\}$
2. Evaluate all game states in S_{next} with the Evaluation Function f. Find the best next game state S_{best} where $f(s_{best}) = max\{f(s_i) \,|\, s_i \epsilon s_{next}\}$ and corresponding move m_{best}. $(s_0 \xrightarrow{move_{best}} s_{best})$
3. Apply $move_{best}$.

Obviously the key element in a Game-Play AI application is the evaluation function. Designing an evaluation function is the process of transferring human knowledge or heuristics into a formula that gives a meaningful evaluation score, to help serve as decision making. The problem with the traditional AI is that

1. The evaluation function is hard to design. There are too many factors needed to be considered in order to give a meaningful score.

2. Typically the evaluation function only considers the current game state without looking at the future, which makes the evaluation score so meaningful. Min-Max search was introduced to solve the problem. But as the number of game states grows exponentially, this approach becomes not practical.

Machine Learning overcomes this problem by giving the computer the abilities to 'learn' how to evaluate the game states. We use a neural network as the model for our Machine Learning. Each future game-state is converted into a input vector, which is then used to create the evaluation score for that move.

Algorithm Description
The basic idea is to feed the neural network with the a future game state and to make a move based on the output of the network, which is the evaluation score of the future game-state. At the end of the game, an error is produced based on the result of the game. The error is used by the backpropagation algorithm to change weights in the network. The neural network improves performance through the weight changes based on the errors of previous games.

1. **Game State Representation**
   For the game Blokus, the game state is the state of the current board. The board is 20×20. So a game board state can be represented by a vector of length 400. Each element is a integer number that represents the state of each cell on the board.

2. **Neural Network**
   The neural network is built with 7 layers (1 input, 5 hidden, 1 output layer). Each layer except the first and last layer contains 50 neurons. The game state vector is fed into the first layer. There is just one neuron in the last layer, which outputs an evaluation score of the game state. All neurons are fully connected between adjacent layers.

3. **Backpropagate The Errors**
   The errors are backpropagated using the following formula.

$$\delta^{x,l} = \left(\left(w^{l+t} \right)^T \delta^{x,l+1} \right) \odot \sigma' \left(z^{x,l} \right)$$

   $\delta^{x,l}$ is the error of x^{th} neuron in l^{th} layer
   w^l is the weight matrix in l^{th} layer
   $\sigma(x) = \frac{1}{1+e^{-x}}$
   $z^{x,l}$ is the weighted input to the x^{th} neuron in the l^{th} layer

4. **Weight Change**
   The weight changes, per neuron, are calculated using the following for-

mula.

$$w^l \to w^l - \frac{\eta}{m} \sum_x \left(\delta^{x,l} \left(a^{x,l-1} \right)^T \right)$$

$a^{x,l}$ is the activation value of x^{th} neuron in l^{th} layer
η is the learning rate.

Implementation Details
When we were assigned the task of writing a blokus-learning algorithm, our first response was to break our task into pieces:

- Some sort of host-program, game-engine, something that could manage the game board, keep track of it's current state, draw it on the screen, keep track of the players' pieces, and calculate possible moves for the players.
- A function that would 'act' for the player, it would somehow take in the current board, and decide what piece to place. This function would be called by the game-engine each turn.
- Some kind of "learning"-method, it would hopefully change the way the 'act'ing-function would operate.

Before we consided the exact implementation of these pieces, we knew that we could make our lives much easier if we didn't start of scratch, and instead build off of other developers's code. That's when a repository on github that already had developed the game-engine, players, and simple user-interface, was discovered. It out-of-the box ran the user-player aganist a random-computer player, and drew the board on an ASCII grid printed out on the terminal. Each player was an object containing an array of pieces, a score, a name, corner to start from, and a function pointer. That function-pointer was to be called during a player's turn, when it needed to decide what piece to place where (the 'act'-ing function). The player objects are handed their 'act'ing-function pointer on construction. Originally, the program had four 'act'ing-functions defined: one that read from standard input from the user and acted as the human player, another that randomly placed legal pieces, a MinMax algorithm implementation, and a Greedy algorithm implementation. By changing the function pointers of the objects, one can easisly remove the human-player, and have computer-players compete instead. Because neural networks require much more data than the typical 'act'ing function, a new 'NNPlayer' object had to be created to store all of the neural network's specific data, such as weights, and past-turns (used for 'learning' at the end of the game).
The new 'NNPlayer' object 'act'ed by calculating all possible moves with the current board and the availible pieces, and evaluating the neural net with each future possible move as the input-vector. To make the best move, the 'act'ing function simply picked the highest-scored future move. The representation of

the future board can be given as a large set of integers, being that the blokus board is a set of small squares, which can only have 5 possible states. ('1' for Player1, '2' for Player2, ..., '0' for empty space) This makes the 'act'-ing function for the 'NNPlayer' objects equailivent to:

```
possible_moves = getFutureMoves(availible_pieces)
possible_moves_scores = []
for move in possible_moves:
        future_board = current_board.copy()
        future_board.playMove(move);
        input_vector = [];
        for x in range(0, future_board.width):
                for y in range(0, future_board.height):
                        cellvalue = 0;
                        if(future_board.WhoOwns(x, y) == player1):
                                cellvalue = 1;
                        if(future_board.WhoOwns(x, y) == player2):
                                cellvalue = 2;
                        if(future_board.WhoOwns(x, y) == player3):
                                cellvalue = 3;
                        if(future_board.WhoOwns(x, y) == player4):
                                cellvalue = 4;
                        input_vector.append(cellvalue);
        score = evaulate_neural_network(weights, input_vector)
        possible_moves_scores.append(score)
highest_score_index = # highest value's index in possible_moves_scores #
return possible_moves[highest_score_index]
```

Disscussion

At first we considered (the very silly idea of) somehow inputting the current board, and the available pieces, and then having the neural net output which piece, x-y cordinate, and rotation/flip for that piece. That idea quickly fell apart when attempting to implement a variable-length list of shapes, as well as teaching the neural net, without enforcing a 'perfect piece', which would be requried for this implementation.

The second idea was to input a future possible board, and have the neural net output a score on the quality of that choice, or how wise that choice is for the player. 'Error' in the system can be the difference bewteen the neural network's output compared to it's final placement (rank). Because rank has limited resolution, and can therefore be difficult for a learning algorithm learn from, we chose to have the 'rank' of a player equal to the difference of their score, to the highest score of the players, and the player with the highest score is compared with the player with the second highest score. For example, if the final scores were {45, 56, 42, 69}, then the 'rank' would be respectively $\{-24, -13, -27, +13\}$. These ranks where then divided by the highest possible score (89), to normalize the ranks before comparing them to the output of the neural net. This had it's own set of problems, the biggest one of which that since a sigmoid function is used in our neurons, networks can never achieve a negative output. This causes a typical backprop. training algorithm to drastically change weights incorrectly. A new and final ranking system was soon proposed.

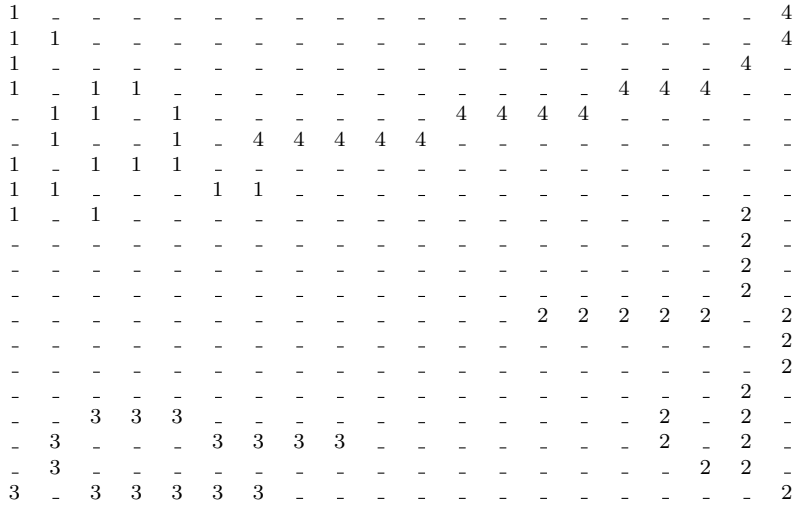
That system was to have the rank of each player simply equal to their normalized final score, this guarantees target values bewteen 0 and 1, as well as a seamless connection bewteen what the neural network *thinks* is going to happen, what *actually* happens, and how the neural network is *changed* by the difference.

Concerns

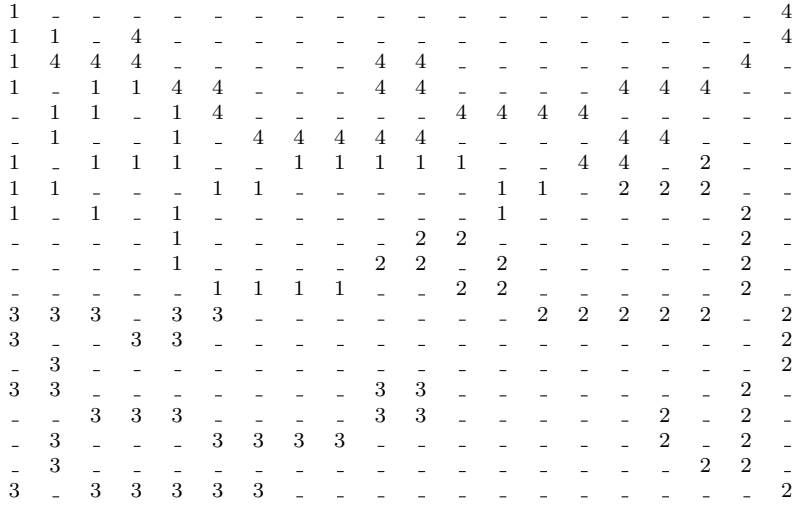
One major concern about the final 'act'ing function, and the 'learning' predicted-rank vs genuine-rank method, is that the neural network will only change the way it 'act's by failures to predict outcomes. That is, if the neural net thought it was going to lose from the very first turn, and it does, no weight-change it made. This problem can theorically lock-up the neural networks, if set to compete against themseleves: each player will settle on a rank, and keep predicting that rank, and keep *performing* that rank, causing no learning for any network. Another area for problems, is the lack of biases in the neural network. Biases are additional weights for each neuron, that can be thought of as a offset, biases are always added in the net-sum, unlike weights, which are dependent on other data. Biases are used in the situation where the input-vector is near zero, that is, the board is practically empty. In the situation where no biases are implemeneted, the first move of each player can be very difficult to tune for, especially since those weights are used for all future moves as well. The result is a very poor first move for all players.

Experimentation/Evaluation

After thousands of games of four 'NNPlayer's, Player1 seems to be the only player that has it's weights tuned for a large-piece first move, the others, however, have only placed either 1 or 2-square pieces as their first move. This is mostly because of the lack of biasing. (see Concerns) Each player has still developed enough to show basic strategies forming: All Players have elected to begin placing large/sizeable pieces first, a simple sign of an experienced player. Player1, Player2, and Player3 seem to start with a Greedy strategy, were they slowly consume space nearest to them, while Player4 has chosen to spend it's large pieces reaching over to surround Player1:



Farther through the game, one can see that Player4 has reached to the other end of the board, making Player1's reach limited, but possibly hindering itself. Player2 and Player3, have begun spreading out to claim space before resources become scarce.



At the end-game, one can see that since Player4 used it's 1-square piece early in the game, it was difficult for it to claim more space, after it had attempted to surround Player1. Player2 and Player3 used the same general strategy overall, they began compact and Greedy, and eventually spread out to begin claiming more space later in the game. Player1 kept with it's inital Greedy strategy, and used it's smaller pieces later in the game to reach into crevices that other Players could not reach. It was this 'saving small pieces towards the end' skill that Player1 had learned before the others, which was the key to it's success. Overall, very clear signs of learning, and development.

```
1 _ _ _ _ 4 4 4 4 _ _ _ _ 4 _ _ 4 _ _ 4
1 1 _ 4 1 1 _ _ 4 _ _ 4 4 4 _ 4 4 4 _ 4
1 4 4 4 1 1 _ _ 1 4 4 _ _ 4 _ _ _ _ 4 _
1 _ 1 1 4 4 1 1 1 4 4 _ _ _ _ 4 4 4 _ _
_ 1 1 _ 1 4 _ _ 1 _ _ 4 4 4 4 _ _ _ 2 _
_ 1 _ _ 1 _ 4 4 4 4 4 _ _ _ _ 4 4 _ 2 2
1 _ 1 1 1 _ _ 1 1 1 1 1 _ _ 4 4 _ 2 _ 2
1 1 _ _ _ 1 1 2 _ _ _ _ 1 1 _ 2 2 2 _ 2
1 _ 1 _ 1 _ _ 2 2 2 _ _ 1 _ _ _ _ _ 2 _
_ _ _ _ 1 _ _ 2 _ _ 2 2 _ 1 1 1 _ _ 2 _
_ _ 1 _ 1 _ _ _ _ 2 2 _ 2 1 _ _ 1 1 2 _
_ _ 1 1 _ 1 1 1 1 _ _ 2 2 _ 1 1 1 _ 2 _
3 3 3 1 3 3 _ _ _ _ 3 3 3 2 2 2 2 2 3 2
3 _ 1 3 3 _ 3 _ _ 3 3 2 2 3 3 3 _ _ 3 2
_ 3 1 1 1 1 3 3 3 _ _ 2 2 _ 3 _ 3 3 3 2
3 3 _ _ _ _ 3 _ _ 3 3 _ _ 2 2 2 _ _ 2 _
_ _ 3 3 3 _ _ _ _ 3 3 _ _ 2 _ _ 2 _ 2 _
_ 3 _ _ _ 3 3 3 3 _ _ 3 _ 2 _ _ 2 _ 2 _
_ 3 _ _ _ _ _ _ _ _ _ 3 3 3 3 2 _ 2 2 _
3 _ 3 3 3 3 3 _ _ _ _ _ _ _ 2 2 2 _ _ 2
Final Scores:
NeuralNet_Player_4 : 44
NeuralNet_Player_2 : 54
NeuralNet_Player_3 : 54
NeuralNet_Player_1 : 64
```