Abstract
Machine Learning is a study field that focusing on giving computer the 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 computer the ability to learn how to play the game, a Neural Network was built with 7 layers. 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 1000 games. Significant improvement could 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 application without using machine learning strategy is the process that transfers human knowledge to formal computer algorithm and implements it in programming languages. This approach gives the computer the knowledge needed to complete the tasks rather than the abilities to learn the knowledge. 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 $f(s_{best}) = max\{f(s_i) \mid s_i \epsilon s_{next}\}$ and corresponding move m_{best}. $(s_0 \xrightarrow{move_{best}} s_{best})$
3. Apply $move_{best}$.

Obviously the key element in Game Play AI application is the Evaluation Function. Designing Evaluation Function is the process to transfer human knowledge or heuristics into a formula that gives a meaningful Evaluation Score to help with decision making.
The problem with the traditional AI

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 no 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 the problem by giving computer the abilities to learning how to evaluate the game states. We use Artificial Neural Network as the model of Machine Learning. A game state is converted to a input vector which gets plugged into the Neural Network. The output of the Network is the Evaluation Score of the game state. We use back-propagation algorithm as our learning algorithm. In the next section, we will talk about our approach in details.

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 gamestate. At the end of the game, a error is produced based on the result of the game. The error is backpropagated to each layers in the network. Finally, the neural network improved performance through the weight changes based on the errors of prevoius 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 pixel on the board.

2. **Neural Network**
   The Neural Network is built with 7 layers. 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 a 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
   $\sigma(\mathrm{x})=\frac{1}{1+e^{-x}}$
   $z^{x,l}$ is the weighted input to the x^{th} neuron in the l^{th} layer

4. **Weight Change**
   The weight change is calculated using the following formula.
   $$w^l \to w^l - \frac{\eta}{m} \sum_x \left(\delta^{x,l}\left(a^{x,l-1}\right)^T\right)$$
   $$b^l \to b^l - \frac{\eta}{m} \sum_x \left(\delta^{x,l}\right)$$
   $a^{x,l}$ 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 we discovered a repository on github that already had developed the game-engine, players, and simple user-interface. 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. We thought this was a perfect starting point, so we began exploring through the code, to find develop a basic idea of the flow of the program, and to start getting an idea of how we can change this program to suit our needs. Each player was an object containing an array of pieces, a score, a name, corner to start from, and a function pointer. Each player was given a function-pointer to call 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 user as a human player, another that randomly placed pieces, Min-Max algorithm, and a Greedy algoritm. By changing the function pointers of the objects, we could easisly remove the human-player, and have two computer-players compete.
Now that we knew we had a soild base, we started to consider in more detail the implementation of our "learning" 'act'ing-function. Even though theorically any "learning" function would do, we felt we would have the best results with a typical netural net desgin. Then came the more difficult part were we had to decide what should be a part of the input layer and output layer, and how it should be represented.

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 choose to have the 'rank' equal to the difference of each player's score, to the player with highest score, 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 from 0 to 1. This had it's own set of problems, the biggest one of which that since a sigmoid function was 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 ranking system was soon proposed.

calculate current game state, as well as . Such input the host-program can easily compute, and a output it can work with to make the best move, simply by picking the highest-scored future move. With this particular output of the neural net, training is very easy. To improve the neural net. we can compare the output of the neural net to the final score of the player at the end of the game. 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 our "learning" players 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]
```

Experimentation/Evaulation

¡¡¡¡¡¡¡ HEAD

======= **Disscussion**

This section talks about all the struggles we had with the project.

¿¿¿¿¿¿¿ 203e23d2f7454c7d83f35b4215dfeb0900556036