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 somehow inputting the current board, and the availible pieces, and then having the neural net output which piece, x-y cordinate, and rotation for that piece. That idea quickly fell apart with how to represent a variable-length list of shapes, and how to train rotation and flips, as such operations aren't linearly separable. The second idea was to input a future possible board, and have the neural net output a score on how well it thinks it is doing in the game. Such input the host-program can easily compute, and a output it can work with to make the best move to the turn, by picking 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 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]
```

Things left to talk about:
- implementation of learning algoythm
- talk about backprop
- how we train our nets
- why we don't use short-term and long-term rewards